

THE EXPERT'S VOICE®

SECOND EDITION

# Pro Git

*EVERYTHING YOU NEED TO  
KNOW ABOUT GIT*

Scott Chacon and Ben Straub

Apress®

# Pro Git

Scott Chacon, Ben Straub

Version 2.1.109, 2018-10-03

# Table of Contents

Licence .....	1
Voorwoord door Scott Chacon .....	2
Voorwoord door Ben Straub .....	4
Opdrachten .....	5
Mensen die een bijdrage hebben geleverd .....	6
Inleiding .....	7
Aan de slag .....	9
Over versiebeheer .....	9
Een kort historisch overzicht van Git .....	13
De grondbeginselen van Git .....	13
De commando-regel .....	17
Git installeren .....	17
Git klaarmaken voor eerste gebruik .....	21
Hulp krijgen .....	24
Samenvatting .....	24
Git Basics .....	26
Een Git repository verkrijgen .....	26
Wijzigingen aan de repository vastleggen .....	28
De commit geschiedenis bekijken .....	40
Dingen ongedaan maken .....	47
Werken met remotes .....	50
Taggen (Labelen) .....	56
Git aliassen .....	61
Samenvatting .....	62
Branchen in Git .....	63
Branches in vogelvlucht .....	63
Eenvoudig branchen en mergen .....	70
Branch-beheer .....	78
Branch workflows .....	80
Branches op afstand (Remote branches) .....	83
Rebasen .....	93
Samenvatting .....	103
Git op de server .....	104
De protocollen .....	104
Git op een server krijgen .....	109
Je publieke SSH sleutel genereren .....	112
De server opzetten .....	113
Git Daemon .....	115

Slimme HTTP .....	117
GitWeb .....	119
GitLab .....	121
Hosting oplossingen van derden .....	126
Samenvatting .....	126
Gedistribueerd Git .....	127
Gedistribueerde workflows .....	127
Bijdragen aan een project .....	130
Het beheren van een project .....	153
Samenvatting .....	168
GitHub .....	169
Account setup en configuratie .....	169
Aan een project bijdragen .....	174
Een project onderhouden .....	193
Een organisatie beheren .....	208
GitHub Scripten .....	211
Samenvatting .....	221
Git Tools .....	222
Revisie Selectie .....	222
Interactief stagen .....	230
Stashen en opschonen .....	234
Je werk tekenen .....	240
Zoeken .....	244
Geschiedenis herschrijven .....	248
Reset ontrafeld .....	256
Mergen voor gevorderden .....	277
Rerere .....	296
Debuggen met Git .....	302
Submodules .....	306
Bundelen .....	325
Vervangen .....	329
Het opslaan van inloggegevens .....	337
Samenvatting .....	343
Git aanpassen .....	344
Git configuratie .....	344
Git attributen .....	355
Git Hooks .....	364
Een voorbeeld van Git-afgedwongen beleid .....	367
Samenvatting .....	377
Git en andere systemen .....	378
Git als een client .....	378

Migreren naar Git .....	425
Samenvatting .....	445
Git Binnenwerk .....	446
Binnenwerk en koetswerk (plumbing and porcelain) .....	446
Git objecten .....	447
Git Referenties .....	458
Packfiles .....	462
De Refspec .....	466
Uitwisseling protocollen .....	469
Onderhoud en gegevensherstel .....	475
Omgevingsvariabelen .....	482
Samenvatting .....	488
Appendix A: Git in andere omgevingen .....	489
Grafische interfaces .....	489
Git in Visual Studio .....	495
Git in Eclipse .....	496
Git in Bash .....	497
Git in Zsh .....	498
Git in Powershell .....	500
Samenvatting .....	502
Appendix B: Git in je applicaties inbouwen .....	503
Commando-regel Git .....	503
Libgit2 .....	503
JGit .....	508
go-git .....	512
Dulwich .....	514
Appendix C: Git Commando's .....	516
Setup en configuratie .....	516
Projecten ophalen en maken .....	517
Basic Snapshotten .....	518
Branchen en mergen .....	520
Projecten delen en bijwerken .....	523
Inspectie en vergelijking .....	525
Debuggen .....	526
Patchen .....	526
Email .....	527
Externe systemen .....	529
Beheer .....	529
Binnenwerk commando's (plumbing commando's) .....	530
Index .....	531

# Licence

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/3.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

# Voorwoord door Scott Chacon

Welkom bij de tweede editie van Pro Git. De eerste editie is nu meer dan vier jaar geleden uitgegeven. Sindsdien is er veel veranderd en tegelijkertijd zijn veel belangrijke zaken dat niet. Waar de meeste kern commando's en concepten vandaag de dag nog steeds valide zijn omdat het kernteam van Git ongelofelijk goed is in het backward compatible houden van zaken, zijn er een aantal significante toevoegingen en wijzigingen in de omgeving van Git geweest. De tweede editie van dit boek is bedoeld om deze wijzigingen een plek te geven en het boek bij te werken zodat het nog behulpzamer kan zijn voor de nieuwe gebruiker.

Toen ik de eerste editie schreef, was Git nog steeds een relatief moeilijk te gebruiken en nauwelijks aangepast stuk gereedschap voor de meer gevorderde hacker. Het begon in bepaalde gezelschappen wel meer ingevoerd te raken, maar het was nog zeker niet de mate van aanwezigheid die het tegenwoordig heeft. Sinds die tijd, heeft bijna elke open source gezelschap het geadopteerd. Git heeft ongelofelijke vooruitgang geboekt op Windows, in de explosieve toename van grafische gebruikers interfaces ervoor op alle platformen en in de ondersteuning door IDE's en in het zakelijk gebruik. De Pro Git van vier jaar geleden is daar onwetend van. Een van de hoofddoelen van deze nieuwe uitgave is om aan al deze nieuwe ontwikkelingen in de Git gemeenschap enige aandacht te besteden.

De Open source gemeenschap die Git gebruikt is ook explosief toegenomen. Toen ik in het begin ging zitten om het boek te schrijven, nu ongeveer vijf jaar geleden (het kostte me nogal wat tijd om de eerste versie te schrijven), was ik net begonnen met werken bij een nogal onbekend bedrijf die een Git hosting website aan het ontwikkelen was dat GitHub heette. Ten tijde van publicatie waren er misschien een paar duizend mensen die de site gebruikten en maar vier van ons die eraan werkten. Op het moment dat ik deze introductie schrijf, kondigt GitHub onze 10 miljoenste gehoste project aan, met bijna 5 miljoen geregistreerde ontwikkelaars accounts en meer dan 230 medewerkers. Of je het leuk vindt of niet, GitHub heeft grote invloed gehad op grote gedeelten van de Open Source gemeenschap op een manier die nauwelijks te bevroeden was toen ik me aan de eerste editie ging wijden.

Ik schreef een kleine paragraaf in de originele versie van Pro Git over GitHub als een voorbeeld van een gehoste Git waar ik eigenlijk nooit erg tevreden over was. Het voelde niet prettig dat ik aan het schrijven was over iets waarvan ik vond dat niet meer was dan een gemeenschapsbron en ook nog eens over mijn bedrijf schreef. En hoewel het nog steeds niet lekker voelt vanwege de verstengeling van belangen, is het belang van GitHub in de Git gemeenschap niet te ontkennen. In plaats van een voorbeeld van Git hosting, heb ik besloten om dat deel van het boek te wijzigen in een meer gedetailleerde beschrijving van wat GitHub is en hoe het efficiënt te gebruiken. Als je gaat leren hoe Git te gebruiken, dan gaat de kennis hoe GitHub te gebruiken je helpen om deel te nemen in een enorme gemeenschap, wat waardevol is ongeacht welke Git host je besluit te gaan gebruiken voor je eigen code.

De andere grote wijziging sinds de tijd van de laatste publicatie is de ontwikkeling en de opkomst van het HTTP protocol voor Git netwerk transacties. De meeste voorbeelden in het boek zijn gewijzigd naar HTTP van SSH omdat het zoveel eenvoudiger is.

Het is prachtig om Git in de afgelopen paar jaar te hebben zien groeien van een relatief obscure versiebeheer systeem tot iets wat commerciële versies en open source versiebeheer domineert. Ik

ben blij dat Pro Git het zo goed gedaan heeft en dat het in staat is geweest om een van de weinige verkrijgbare technische boeken te zijn die zowel redelijk succesvol als volledig open source is.

Ik hoop dat je van deze bijgewerkte editie van Pro Git zult genieten.

# Voorwoord door Ben Straub

De eerste editie van dit boek is wat me verslaafd maakte aan Git. Dit was mijn introductie in een manier van software maken die natuurlijker aanvoelde dan alles wat ik daarvoor gezien had. Ik had al een aantal jaren ervaring als ontwikkelaar achter de rug, maar dit boek was mijn reisgids naar een veel interessantere weg dan die waar ik eerst op bevond.

Nu, jaren later, lever ik een bijdrage aan een belangrijke Git implementatie, ik ben gaan werken voor het grootste Git hosting bedrijf, en ik heb de wereld rondgereisd om mensen te leren over Git. Toen Scott me vroeg of ik interesse had om aan de tweede uitgave mee te werken, hoefde ik er geen seconde over na te denken.

Het was me een groot plezier en een voorrecht om aan dit boek te werken. Ik hoop dat het jou net zoveel zal helpen als het mij gedaan heeft.

# Opdrachten

*Aan mijn vrouw, Becky, zonder wie dit avontuur nooit zou zijn begonnen. — Ben*

*Deze uitgave draag ik op aan mijn meisjes. Aan mijn vrouw Jessica die me al deze jaren heeft gesteund en aan mijn dochter Josephine, die mij zal ondersteunen wanneer ik te oud ben om te beseffen wat er gebeurt. — Scott*

# Mensen die een bijdrage hebben geleverd

Omdat dit een Open Source boek is, hebben we in de loop der jaren allerhande errata en wijzigingen op de inhoud gekregen. Hier zijn alle mensen die aan de Engelse versie van Pro Git als open source project hebben bijgedragen. Dank aan iedereen voor het helpen om dit een beter boek voor iedereen te maken.

Aleh Suprunovich	Dan Schmidt	Masood Fallahpoor	Ronald Wampler
Alexandre Garnier	Dmitri Tikhonov	Matthew Miner	Sanders
Kleinfeld			
Andrew MacFie	dualsky	Mike Thibodeau	
sanders@oreilly.com			
Antonino Ingargiola	Haruo Nakayama	Pablo Schläpfer	Sarah
Schneider			
Carlos Martín Nieto	Jean-Noël Avila	Pascal Borreli	Siarhei Krukau
Christopher Wilson	Jon Forrest	paveljanik	Stephan van
Maris			
Christoph Prokop	Justin Clift	Pavel Janík	Thomas
Ackermann			
cor	Louise Corrigan	petsuter	Tom Schady
Cor Takken	Luc Morin	Philippe Miossec	Xander Smeets
Damien Tournoud	Marti Bolivar	rahrah	

# Inleiding

Je staat op het punt om een aantal uren van je leven te besteden aan het lezen over Git. Laten we in het kort even uitleggen wat we voor je in petto hebben. Hier is een korte samenvatting van de tien hoofdstukken en drie appendices van dit boek.

In **Hoofdstuk 1** gaan we Versiebeheer systemen (VBSsen) (Version Control Systems (VCSsen)) behandelen en de grondbeginselen van Git—geen technische dingen, alleen wat Git is, hoe het tot stand is gekomen in een landschap vol met VCSsen, waarin het zich onderscheidt en waarom zoveel mensen het gebruiken. Daarna zullen we uitleggen hoe Git te downloaden en het voor de eerste keer op te zetten als je het al niet op je systeem hebt staan.

In **Hoofdstuk 2** zullen het simpele gebruik van Git behandelen—hoe Git te gebruiken in 80% van de gevallen die je het meest zult tegenkomen. Na het lezen van dit hoofdstuk zou je in staat moeten zijn om een repository te klonen, zien wat er in de geschiedenis van het project is gebeurd, bestanden wijzigen, en wijzigingen bij te dragen. Als het boek op dat punt spontaan zou ontbranden, zou je al aardig in staat moeten zijn om Git te blijven gebruiken tot moment dat je een ander exemplaar weet te bemachtigen.

**Hoofdstuk 3** gaat over het branching model in Git, vaak beschreven als de *killer feature* van Git. Hier zal je leren wat Git echt onderscheidt van de rest. Als je hiermee klaar bent, zou je misschien de behoefte kunnen voelen om even in stilte te overpeinen hoe je het kunt volhouden voordat branchen met Git een onderdeel van je leven was.

**Hoofdstuk 4** vertelt het verhaal van Git op de server. Dit hoofdstuk is voor die personen onder jullie die Git willen opzetten binnen je organisatie of je eigen persoonlijke server voor samenwerking. We zullen ook een aantal gehoste oplossingen verkennen als je dat liever aan iemand anders overlaat.

In **Hoofdstuk 5** wordt in volledige detail verschillende gedistribueerde workflows behandeld en hoe je ze kunt bereiken met Git. Als je klaar bent met dit hoofdstuk, zou je in staat moeten zijn om als een expert te werken met meerdere remote repositories, Git via e-mail te gebruiken en volleerd verschillende remote branches en bijgedragen patches in de lucht te houden.

**Hoofdstuk 6** behandelt in detail de GitHub hosting service en gereedschappen. We behandelen het aanmelden voor en het onderhouden van een account, het maken en gebruiken van Git repositories, de gewone workflows om aan projecten bij te dragen en om bijdragen aan jouw project te accepteren, de programmatische interface van GitHub en vele kleine tips om je leven in het algemeen eenvoudiger te maken.

**Hoofdstuk 7** gaat over geavanceerde Git commando's. Hier zal je over een aantal onderwerpen leren zoals het onder de knie krijgen van het enge *reset* commando lezen, het gebruik van binaire zoeken om bugs te vinden, historie te wijzigen, het selecteren van revisies en veel, veel meer. Dit hoofdstuk zal je kennis van Git verdiepen zodat je een echte meester zult worden.

**Hoofdstuk 8** behandelt het configureren van je aangepaste Git environment. Dit is inclusief het opzetten van hook scripts voor het afdwingen van of het aanmoedigen tot het volgen van aangepast beleid en het gebruiken van configuratie instellingen uit de omgeving zodat je op de manier kunt werken die jij wilt. We behandelen ook het maken van je eigen verzameling scripts om een

aangepaste commit-beleid af te dwingen.

**Hoofdstuk 9** verhaalt over Git en andere VCSsen. Daarin ook het gebruik van Git in een Subversion (SVN) wereld en het converteren van projecten uit andere VCSsen naar Git. Een groot aantal organisaties gebruikt nog steeds SVN en zullen dit voorlopig nog niet veranderen, maar op dit moment zal je al op de hoogte zijn van de ongelofelijke kracht van Git—en dit hoofdstuk laat je zien hoe je hiermee kunt omgaan als je nog steeds een SVN server moet gebruiken. We behandelen ook hoe projecten van diverse verschillende systemen kunt importeren voor het geval dat je toch in staat bent geweest iedereen ervan te overtuigen de sprong te wagen.

In **Hoofdstuk 10** duiken we in de duistere doch prachtige diepten van het binnenwerk van Git. Nu je alles weet over Git en het met macht en gratie kunt gebruiken, kan je doorgaan en bespreken we hoe Git zijn objecten opslaat, wat het objectmodel is, detail van packfiles, server protocollen, en meer. Door het boek heen, zullen we naar paragrafen van dit hoofdstuk verwijzen voor het geval dat je op dat punt de diepte in wilt gaan; maar als je ook maar een beetje op ons lijkt en meteen in de technische details wilt duiken, zou je ook meteen hoofdstuk 10 kunnen beginnen lezen. We laten dat geheel aan jou over.

In **Appendix A** bekijken we een aantal voorbeelden van hoe Git te gebruiken in diverse specifieke omgevingen. We behandelen een aantal verschillende GUIs en IDE programmeer omgevingen waarin je Git zou kunnen willen gebruiken en wat er zoal voor je beschikbaar is. Als je geïnteresseerd bent in een overzicht hoe Git in je shell te gebruiken, in Visual Studio of Eclipse, neem je hier een kijkje.

In **Appendix B** verkennen we scripting en het uitbreiden van Git door gereedschappen als libgit2 en JGit. Als je geïnteresseerd bent in het schrijven van complexe en snelle aangepaste gereedschappen en technische toegang nodig hebt, is dit waar je kunt zien hoe dat landschap eruit ziet.

Tot slot behandelen we in **Appendix C** alle belangrijke Git commando's een voor een en kijken terug naar waar in het boek we ze hebben behandeld en wat we ermee gedaan hebben. Als je wilt weten waar in het boek we een specifieke Git commando hebben gebruikt kan je dat daar opzoeken.

Laten we beginnen.

# Aan de slag

In dit hoofdstuk wordt uitgelegd hoe je aan de slag kunt gaan met Git. We zullen beginnen met wat achtergrondinformatie over versiebeheersystemen te geven, daarna een korte uitleg hoe je Git werkend kan krijgen op je computer en sluiten af met uit te leggen hoe je Git kan inrichten zodat je ermee aan het werk kunt. Aan het einde van dit hoofdstuk zou je moeten kunnen begrijpen waarom Git er is, waarom je het zou moeten gebruiken en zal je helemaal gereed zijn om er mee aan de slag te gaan.

## Over versiebeheer

Wat is versiebeheer, en waarom zou je je er druk over maken? Versiebeheer is het systeem waarin veranderingen in een bestand of groep van bestanden over de tijd wordt bijgehouden, zodat je later specifieke versies kan opvragen. In de voorbeelden in dit boek is het broncode van software waarvan de versies beheerd worden, maar in praktijk kan elk soort bestand op een computer aan versiebeheer worden onderworpen.

Als je een grafisch ontwerper bent of websites ontwerpt en elke versie van een afbeelding of opmaak wilt bewaren (wat je vrijwel zeker zult willen), is het zeer verstandig een versiebeheersysteem (Version Control System in het Engels, afgekort tot VCS) te gebruiken. Het gebruik hiervan stelt je in staat eerdere versies van bestanden of het hele project terug te halen, wijzigingen tussen twee momenten in de tijd te bekijken, zien wie het laatst iets aangepast heeft wat een probleem zou kunnen veroorzaken, wie een probleem heeft veroorzaakt en wanneer en nog veel meer. Een VCS gebruiken betekent meestal ook dat je de situatie gemakkelijk terug kan draaien als je een fout maakt of bestanden kwijtraakt. Daarbij komt nog dat dit allemaal heel weinig extra werk met zich mee brengt.

## Lokale versiebeheersystemen

De voorkeursmethode van veel mensen voor versiebeheer is om bestanden naar een andere map te kopiëren (en als ze slim zijn, geven ze die map ook een datum in de naam). Deze methode wordt veel gebruikt omdat het zo simpel is, maar het is ook ongelofelijk foutgevoelig. Het is makkelijk te vergeten in welke map je zit en naar het verkeerde bestand te schrijven, of onbedoeld over bestanden heen te kopiëren.

Om met dit probleem om te gaan hebben programmeurs lang geleden lokale VCSen ontwikkeld die een simpele database gebruiken om alle veranderingen aan bestanden te beheren.

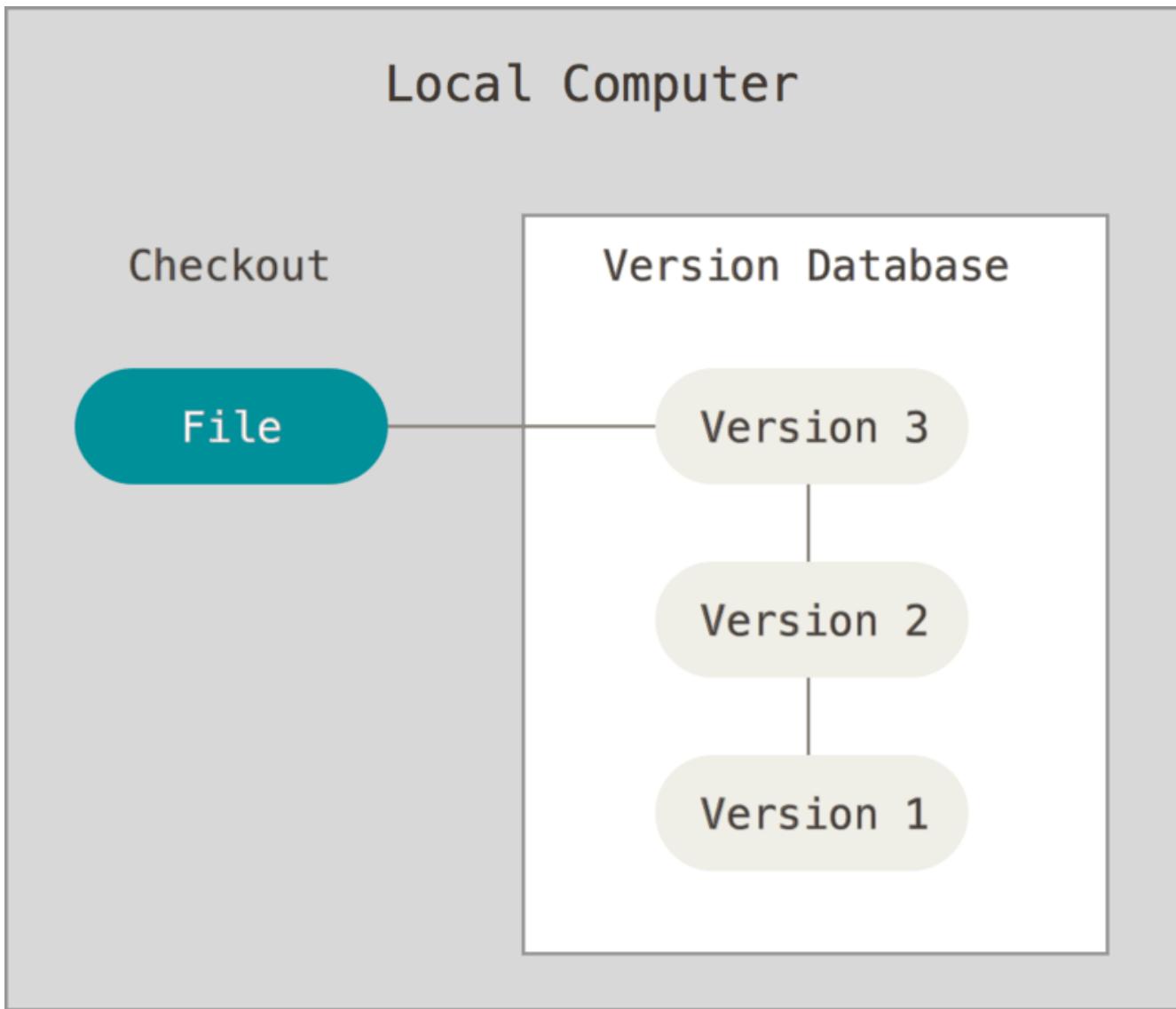


Figure 1. Lokale versiebeheer.

Een populair gereedschap voor VCS was een systeem genaamd RCS, wat vandaag de dag nog steeds met veel computers wordt meegeleverd. RCS werkt door verzamelingen van *patches* (dat zijn de verschillen tussen bestanden) van de opvolgende bestandsversies in een speciaal formaat op de harde schijf op te slaan. Zo kan je een bestand reproduceren zoals deze er uitzag op elk willekeurig moment in tijd door alle patches bij elkaar op te tellen.

## Gecentraliseerde versiebeheersystemen

De volgende belangrijke uitdaging waar mensen mee te maken krijgen is dat ze samen moeten werken met ontwikkelaars op andere computers. Om deze uitdaging aan te gaan ontwikkelde men Gecentraliseerde Versiebeheersystemen (Centralized Version Control Systems, afgekort CVCSs). Deze systemen, zoals CVS, Subversion en Perforce, hebben één centrale server waarop alle versies van de bestanden staan en een aantal werkstations die de bestanden daar van ophalen (*check out*). Vele jaren was dit de standaard voor versiebeheer.

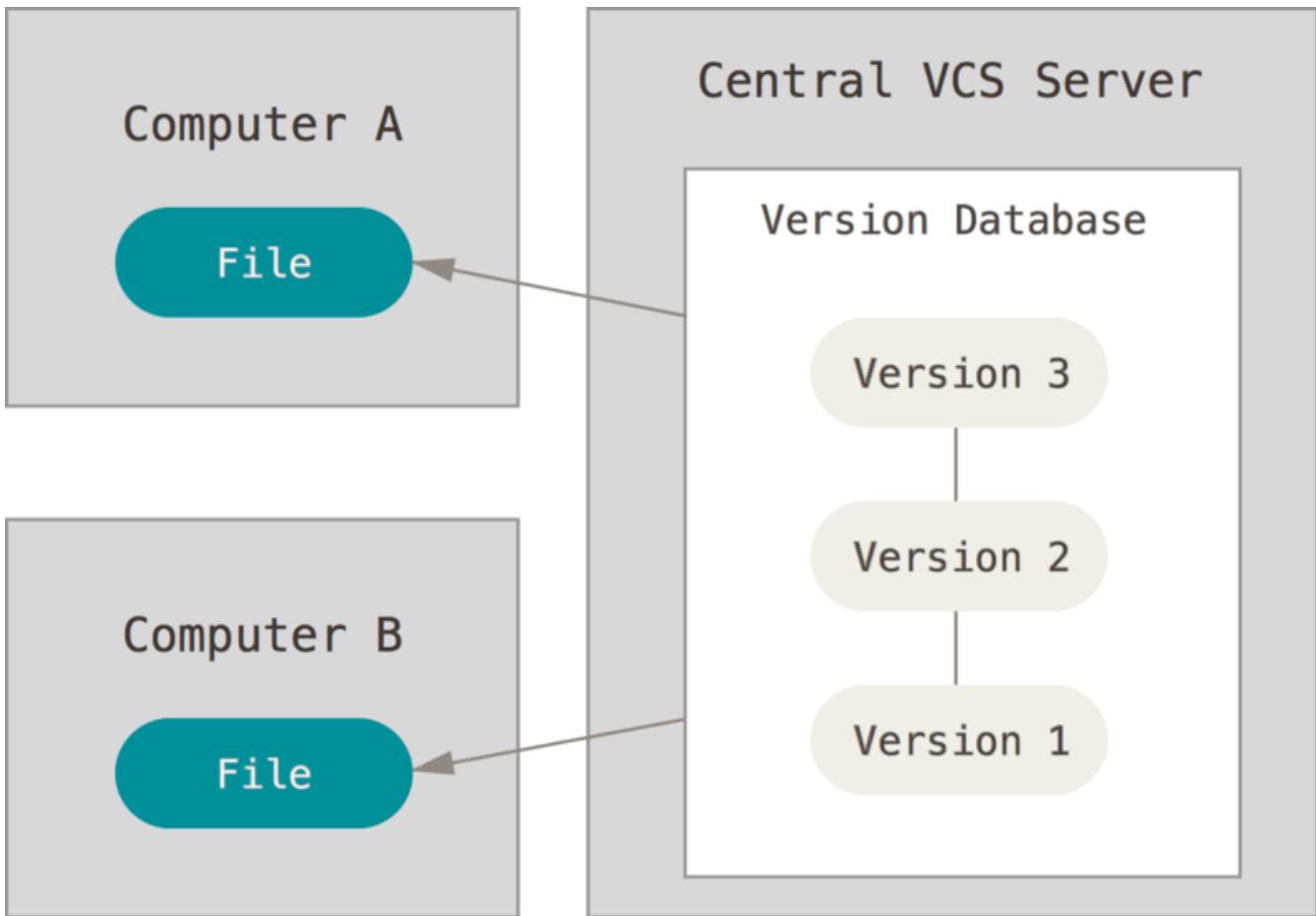


Figure 2. Gecentraliseerde versiebeheer.

Deze manier van versiebeheer biedt veel voordeelen, zeker ten opzichte van lokale VCSen. Bijvoorbeeld: iedereen weet, tot op zekere hoogte, wat de overige project-medewerkers aan het doen zijn. Beheerders hebben een hoge mate van controle over wie wat kan doen, en het is veel eenvoudiger om een CVCS te beheren dan te moeten werken met lokale databases op elke werkstation.

Maar helaas, deze methode heeft ook behoorlijke nadelen. De duidelijkste is de *single point of failure*: als de centrale server plat gaat en een uur later weer terug online komt kan niemand in dat uur samenwerken of versies bewaren van de dingen waar ze aan werken. Als de harde schijf waar de centrale database op staat corrupt raakt en er geen backups van zijn, verlies je echt alles; de hele geschiedenis van het project, op de toevallige momentopnames na die mensen op hun eigen computers hebben staan. Lokale VCS systemen hebben hetzelfde probleem: als je de hele geschiedenis van het project op één enkele plaats bewaart, loop je ook kans alles te verliezen.

## Gedistribueerde versiebeheersystemen

En hier verschijnen Gedistribueerde versiebeheersystemen (Distributed Version Control Systems, DVCSs) ten tonele. In een DVCS (zoals Git, Mercurial, Bazaar of Darcs) downloaden werkstations niet simpelweg de laatste momentopnames van de bestanden; de hele opslagplaats (de *repository*) wordt gekopieerd. Dus als een willekeurige server uitvalt en deze systemen werken via die server samen dan kan de repository van eender welke werkstation terug worden gekopieerd naar de server om deze te herstellen. Elke kloon is dus in feite een complete backup van alle data.

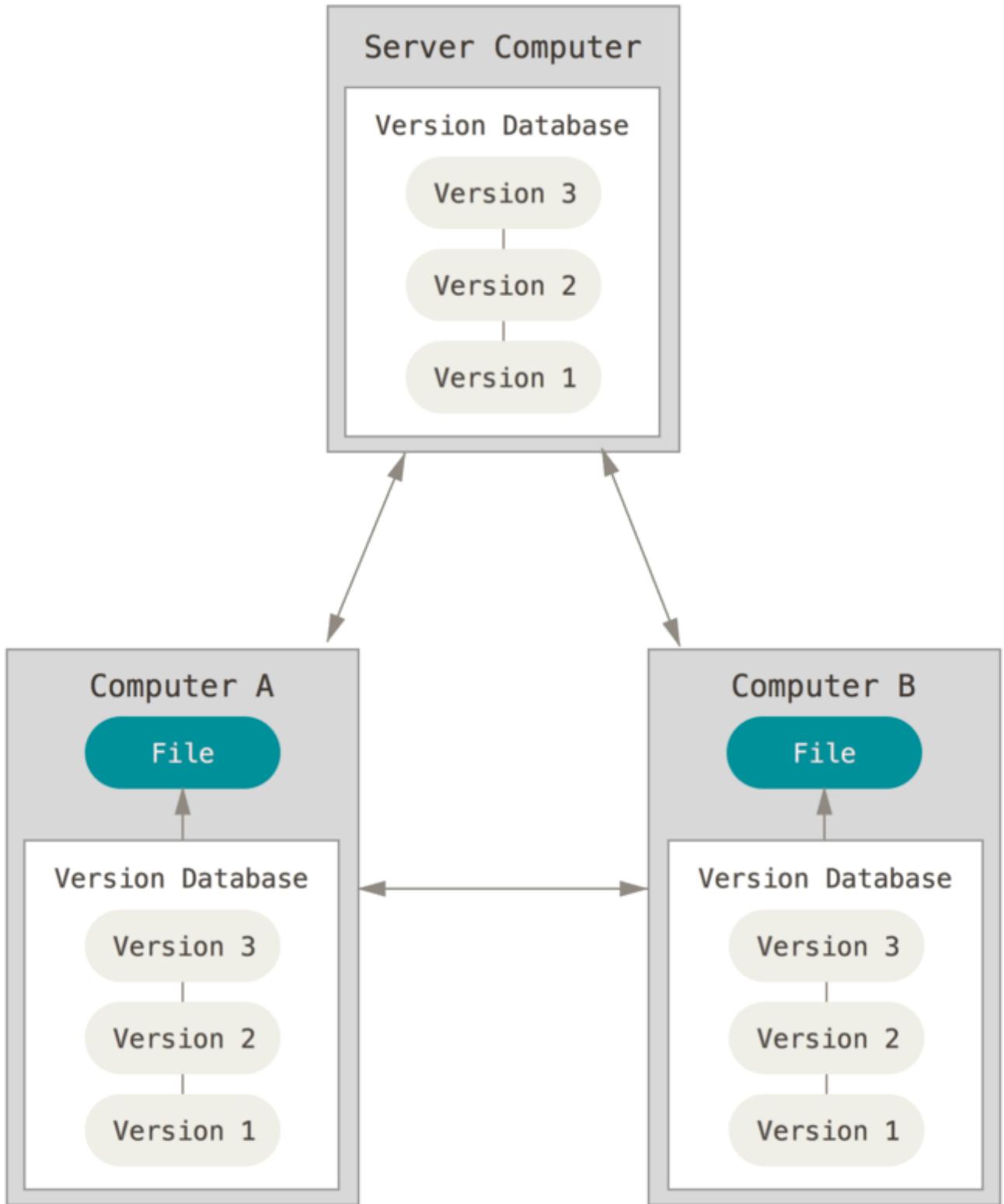


Figure 3. Gedistribueerde versiebeheer.

Bovendien kunnen veel van deze systemen behoorlijk goed omgaan met meerdere (niet-lokale) repositories tegelijk, zodat je met verschillende groepen mensen op verschillende manieren tegelijk aan hetzelfde project kan werken. Hierdoor kan je verschillende werkprocessen (*workflows*) zoals hiërarchische modellen opzetten die niet mogelijk waren geweest met gecentraliseerde systemen.

# Een kort historisch overzicht van Git

Zoals zoveel goede dingen in het leven begon Git met een beetje creatieve destructie en een diepzittende controverse.

De Linux kernel is een open source softwareproject met een behoorlijk grote omvang. Voor een lange tijd tijdens het onderhoud van de Linux kernel (1991-2002), werden aanpassingen aan de software voornamelijk verspreid via patches en gearchiveerde bestanden. In 2002 begon het project een gesloten DVCS genaamd BitKeeper te gebruiken.

In 2005 viel de relatie tussen de gemeenschap die de Linux kernel ontwikkelde en het commerciële bedrijf dat BitKeeper maakte uiteen, en het programma mocht niet langer meer gratis worden gebruikt. Dit was de aanleiding voor de Linux ontwikkel-gemeenschap (en Linus Torvalds, de maker van Linux, in het bijzonder) om hun eigen gereedschap te ontwikkelen, gebaseerd op een aantal lessen die waren geleerd toen ze nog BitKeeper gebruikten. Een aantal van de doelen die ze hadden voor het nieuwe systeem waren als volgt:

- Snelheid
- Eenvoudig ontwerp
- Goede ondersteuning voor niet-lineaire ontwikkeling (duizenden parallelle takken (branches) )
- Volledig gedistribueerd
- In staat om efficiënt om te gaan met grote projecten als de Linux kernel (voor wat betreft snelheid maar ook opslagruimte)

Sinds het ontstaan in 2005 is Git gegroeid tot zijn huidige vorm: het is eenvoudig te gebruiken en heeft toch die oorspronkelijke eigenschappen behouden. Het is ongelofelijk snel, enorm efficiënt met grote projecten en bezit een ongeëvenaard branch-systeem voor het ondersteunen van niet-lineaire ontwikkelen (zie [Branchen in Git](#)).

## De grondbeginselen van Git

Dus, wat is Git in een notendop? Dit is een belangrijke paragraaf om in je op te nemen omdat, als je goed begrijpt wat Git is en de fundamenten van de interne werking begrijpt, het een stuk makkelijker wordt om Git effectief te gebruiken. Probeer, als je Git aan het leren bent, te vergeten wat je al weet over andere VCSen zoals Subversion en Perforce; zo kan je verwarring bij gebruik door de subtiele verschillen voorkomen. Git gaat op een hele andere manier met informatie om dan andere systemen, ook al lijken de verschillende commando's behoorlijk op elkaar. Als je die verschillen begrijpt, kan je voorkomen dat je verward raakt als je Git gebruikt.

## Momentopnames, geen verschillen

Een groot verschil tussen Git en elke andere VCS (inclusief Subversion en consorten) is hoe Git denkt over data. Conceptueel bewaren de meeste andere systemen informatie als een lijst van veranderingen per bestand. Deze systemen (CVS, Subversion, Perforce, Bazaar, enzovoort) zien de informatie die ze bewaren als een groep bestanden en de veranderingen die aan die bestanden zijn aangebracht in de loop der tijd (dit wordt gewoonlijk beschreven als versiebeheer d.m.v. deltas).

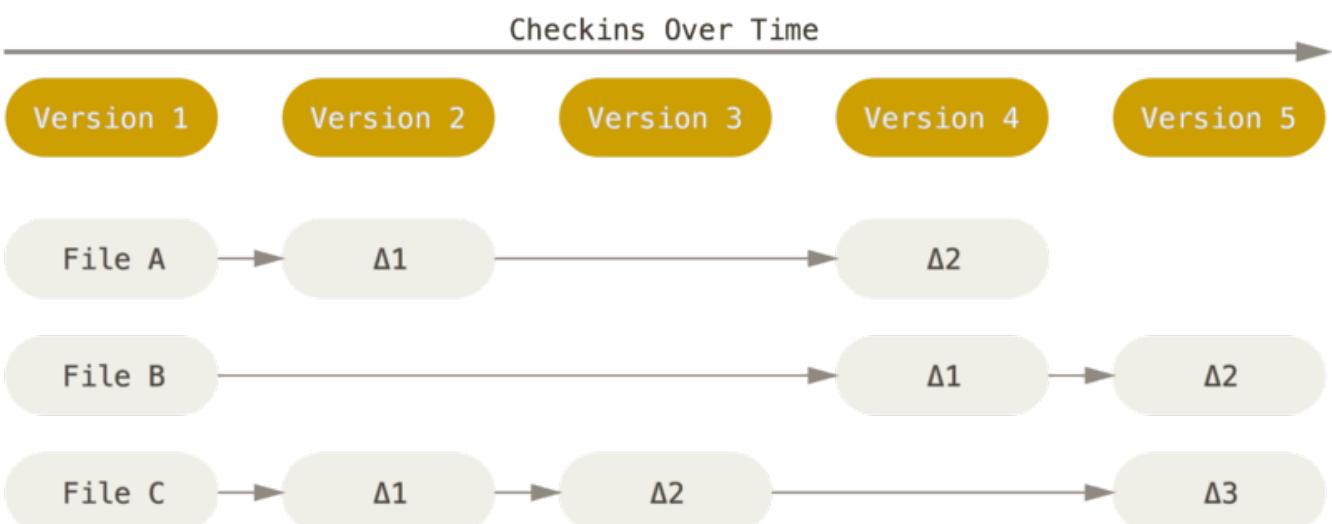


Figure 4. Het bewaren van data als veranderingen aan een basisversie van elk bestand.

Git ziet en bewaart data heel anders. De kijk van Git op data kan worden uitgelegd als een reeks momentopnames (snapshots) van een miniatuurbestandssysteem. Elke keer dat je *commit* (de status van je project in Git opslaat) wordt er als het ware foto van de toestand van al je bestanden op dat moment genomen en wordt er een verwijzing naar die foto opgeslagen. Uit oogpunt van efficiëntie slaat Git ongewijzigde bestanden niet elke keer opnieuw op, alleen een verwijzing naar het eerdere identieke bestand dat het eerder al opgeslagen had. Git beschouwt gegevens meer als een **reeks van snapshots**.

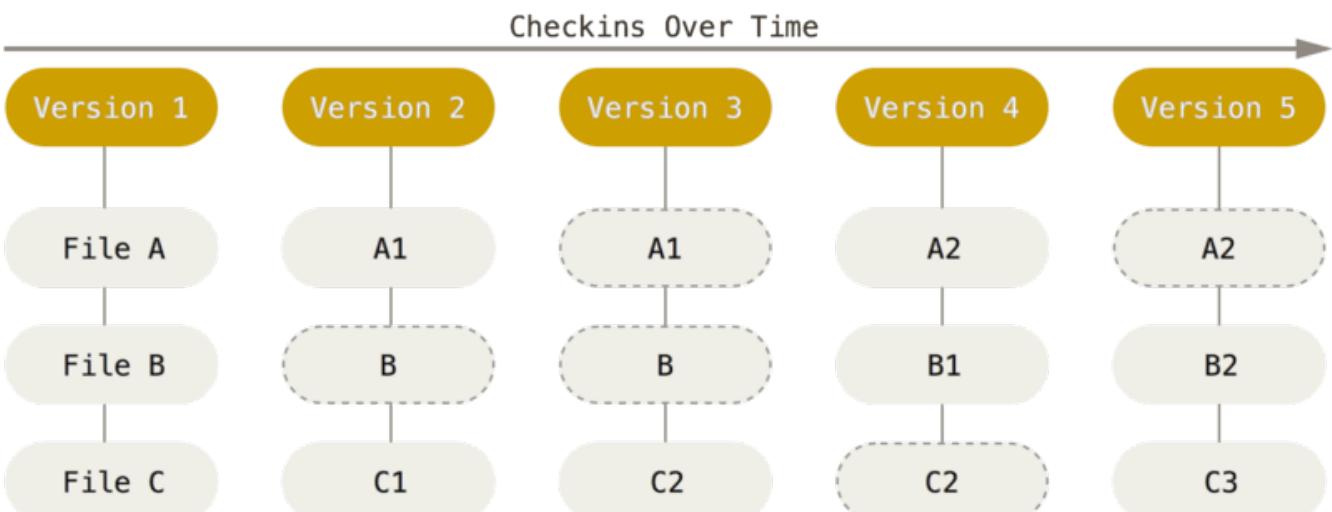


Figure 5. Het bewaren van gegevens als snapshots van het project gedurende de tijd.

Dat is een belangrijk onderscheid tussen Git en bijna alle overige VCSen. Hierdoor moet Git bijna elk aspect van versiebeheer heroverwegen, terwijl de meeste andere systemen het hebben overgenomen van voorgaande generaties. Dit maakt Git meer een soort mini-bestandssysteem met een paar ongelooflijk krachtige gereedschappen, in plaats van niets meer een VCS. We zullen een paar van de voordelen die je krijgt als je op die manier over data denkt gaan onderzoeken, als we *branchen* (gesplitste ontwikkeling) toelichten in [Branchen in Git](#).

## Bijna alle handelingen zijn lokaal

De meeste handelingen in Git hebben alleen lokale bestanden en hulpmiddelen nodig. Normaalgesproken is geen informatie nodig van een andere computer uit je netwerk. Als je gewend bent aan een CVCS, waar de meeste handelingen vertraagd worden door het netwerk, lijkt Git door

de goden van snelheid begenadigd met bovennatuurlijke krachten. Omdat je de hele geschiedenis van het project op je lokale harde schijf hebt staan, lijken de meeste acties geen tijd in beslag te nemen.

Een voorbeeld: om de geschiedenis van je project te doorlopen hoeft Git niet bij een andere server de geschiedenis van je project op te vragen; het leest simpelweg jouw lokale database. Dat betekent dat je de geschiedenis van het project bijna direct te zien krijgt. Als je de veranderingen wilt zien tussen de huidige versie van een bestand en de versie van een maand geleden kan Git het bestand van een maand geleden opzoeken en lokaal de verschillen berekenen, in plaats van aan een niet-lokale server te moeten vragen om het te doen, of de oudere versie van het bestand ophalen van een server om het vervolgens lokaal te doen.

Dit betekent ook dat er maar heel weinig is dat je niet kunt doen als je offline bent of zonder VPN zit. Als je in een vliegtuig of trein zit en je wilt nog even wat werken, kan je vrolijk doorgaan met commits maken (naar je *lokale* kopie, weet je nog?) tot je een netwerkverbinding hebt en je dat werk kunt uploaden. Als je naar huis gaat en je VPN client niet aan de praat krijgt kan je nog steeds doorwerken. Bij veel andere systemen is dat onmogelijk of zeer onaangenaam. Als je bijvoorbeeld Perforce gebruikt kan je niet zo veel doen als je niet verbonden bent met de server. Met Subversion en CVS kun je bestanden bewerken maar je kunt geen commits maken naar je database (omdat die offline is). Dat lijkt misschien niet zo belangrijk maar je zult nog versteld staan wat voor een verschil het kan maken.

## Git heeft integriteit

Alles in Git krijgt een controlegetal (*checksum*) voordat het wordt opgeslagen en er wordt later met dat controlegetal naar gerefereerd. Dat betekent dat het onmogelijk is om de inhoud van een bestand of directory te veranderen zonder dat Git er weet van heeft. Deze functionaliteit is in het diepste niveau van Git ingebouwd en staat centraal in zijn filosofie. Je kunt geen informatie kwijtraken tijdens transport en bestanden kunnen niet corrupt raken zonder dat Git het kan opmerken.

Het mechanisme dat Git gebruikt voor deze controlegetallen heet een SHA-1-hash . Dat is een tekenreeks van 40 karakters lang, bestaande uit hexadecimale tekens (0-9 en a-f) en wordt berekend uit de inhoud van een bestand of directory-structuur in Git. Een SHA-1-hash ziet er als volgt zo uit:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Je zult deze hashwaarden overal tegenkomen omdat Git er zoveel gebruik van maakt. Sterker nog, Git bewaart alles in haar database niet onder een bestandsnaam maar met de hash van de inhoud als sleutel.

## Git voegt normaal gesproken alleen data toe

Bijna alles wat je in Git doet, leidt tot *toevoeging* van data in de Git database. Het is erg moeilijk om het systeem iets te laten doen wat je niet ongedaan kan maken of het de gegevens te laten wissen op wat voor manier dan ook. Zoals met elke VCS kun je wijzigingen verliezen of verhaspelen als je deze nog niet hebt gecommit; maar als je een snapshot hebt gecommit, is het erg moeilijk om die

data te verliezen, zeker als je de lokale database regelmatig uploadt (met *push*) naar een andere repository.

Dit maakt het gebruik van Git zo plezierig omdat je weet dat je kunt experimenteren zonder het gevaar te lopen jezelf behoorlijk in de nesten te werken. Zie [Dingen ongedaan maken](#) voor een iets diepgaandere uitleg over hoe Git gegevens bewaart en hoe je de gegevens die verloren lijken kunt terughalen.

## De drie toestanden

Let nu even goed op, dit is het belangrijkste wat je over Git moet weten als je wilt dat de rest van het leerproces gladjes verloopt. Git heeft drie hoofdtoestanden waarin bestanden zich kunnen bevinden: *gecommit (committed)*, *gewijzigd (modified)* en *voorbereid voor een commit (staged)*:

- Committed houdt in dat alle data veilig opgeslagen is in je lokale database.
- Modified betekent dat je het bestand hebt gewijzigd maar dat je nog niet naar je database gecommit hebt.
- Staged betekent dat je al hebt aangegeven dat de huidige versie van het aangepaste bestand in je volgende commit meegenomen moet worden.

Dit brengt ons tot de drie hoofdonderdelen van een Gitproject: de Git directory, de werk directory (*working tree*), en de wachtrij voor een commit (*staging area*).

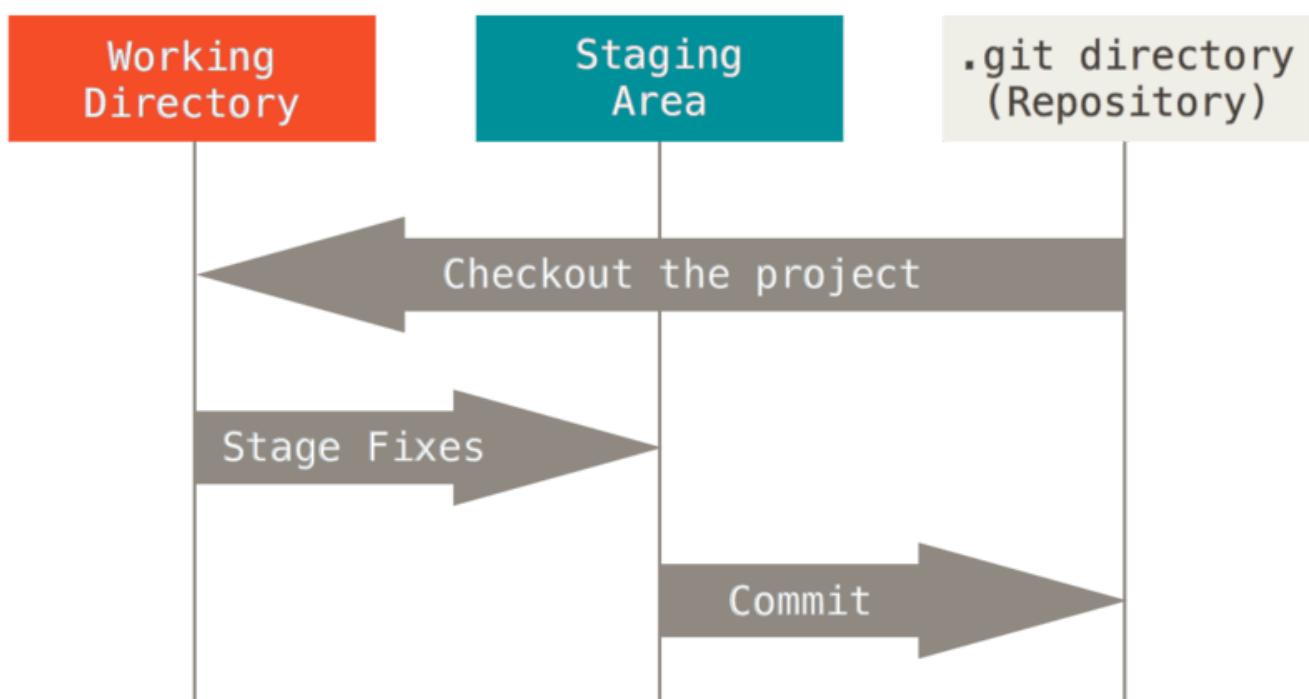


Figure 6. Working tree, staging area en Git directory.

De Git directory is waar Git de metadata en objectdatabase van je project opslaat. Dit is het belangrijkste deel van Git, deze directory wordt gekopieerd wanneer je een repository *kloont* (*clone*) vanaf een andere computer.

De working tree is een checkout van een bepaalde versie van het project. Deze bestanden worden uit de gecomprimeerde database in de Git directory gehaald en op de harde schijf geplaatst waar jij

ze kunt gebruiken of bewerken.

De staging area is een simpel bestand, dat zich normaalgesproken in je Git directory bevindt, waar informatie opgeslagen wordt over wat in de volgende commit meegaat. In Git vaktaal wordt dit de “index” genoemd, maar de zin “staging area” werkt net zo goed.

De algemene workflow met Git gaat ongeveer zo:

1. Je bewerkt bestanden in je working tree.
2. Je staged een selectie van die wijzigingen die je in de volgende commit wilt hebben, dit voegt *alleen* die wijzigingen in de staging area toe.
3. Je maakt een commit, hierbij worden snapshots gemaakt van alle bestanden in de staging area en verzameld en deze worden voorgoed in je Git directory bewaard.

Als een bepaalde versie van een bestand in de Git directory staat, wordt het beschouwd als committed. Als het is aangepast, maar wel al aan de staging area is toegevoegd, is het staged. En als het veranderd is sinds het was uitgechecked maar niet gestaged is, is het modified. In [Git Basics](#) leer je meer over deze toestanden en hoe je er je voordeel mee kunt doen, maar ook hoe je het staggen helemaal over kunt slaan.

## De commando-regel

Er zijn veel verschillende manieren om Git te gebruiken. Er zijn de oorspronkelijke commando-regel tools, en er zijn vele grafische hulpmiddelen (GUI) met verscheidene mogelijkheden. In dit boek zullen we Git gebruiken op de commando regel. Dit omdat de commando-regel de enige plaats is waar je *alle* Git commando's kan aanroepen - de meeste GUIs implementeren maar een deel van de Git functionaliteit voor de eenvoud. Als je de commando-regel versie kan gebruiken, kan je waarschijnlijk ook wel aanvoelen hoe een GUI versie te gebruiken, terwijl het omgekeerde niet altijd het geval is. Als laatste: de keuze van grafisch hulpmiddel is een kwestie van persoonlijke smaak, *alle* gebruikers hebben de commando-regel tools geïnstalleerd en tot hun beschikking.

We gaan er dus van uit dat je weet hoe een Terminal in Mac te openen of een Command Prompt of Powershell in Windows. Als je niet weet waar we het hier over hebben, is het wellicht verstandig om hier te even stoppen en dit op te zoeken zodat je de rest van de voorbeelden en beschrijvingen in dit boek kunt volgen.

## Git installeren

Voordat je Git kunt gaan gebruiken, moet je het eerst beschikbaar maken op je computer. Zelfs als het al is geïnstalleerd, is het waarschijnlijk een goed idee om de laatste update te installeren. Je kunt het installeren als een los pakket of via een andere installatieprocedure, of de broncode downloaden en zelf compileren.



Dit boek is geschreven uitgaande van Git versie **2.0.0**. Alhoewel de meeste commando's die we gebruiken zelfs zouden moeten werken in heel erg oude versie van Git, zouden sommige niet kunnen werken of iets anders reageren als je een oudere versie gebruikt. Omdat Git redelijk goed is in het bewaken van *backwards compatibility*, zou elke versie later dan 2.0 prima moeten werken.

## Installeren op Linux

Als je direct de uitvoerbare bestanden van Git op Linux wilt installeren, kun je dat over het algemeen doen via het standaard pakketbeheersysteem dat meegeleverd is met jouw distributie. Als je Fedora gebruikt (of een direct gerelateerde RPM-gebaseerde distributie, zoals RHEL of CentOS) kan je `dnf` gebruiken:

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

Als je op een Debian-gerelateerde distributie zit, zoals Ubuntu, kan je `apt` proberen:

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

Voor meer opties, er zijn instructies voor het installeren op diverse Unix distributies op de Git webpagina op <http://git-scm.com/download/linux>.

## Installeren op Mac

Er zijn diverse manieren om Git op een Mac te installeren. De simpelste is om de Xcode command line tools te installeren. Op Mavericks (10.9) of hoger kan je dit eenvoudigweg doen door `git` aan te roepen vanaf de Terminal op de allereerste regel.

```
$ git --version
```

Als je het al niet geïnstalleerd hebt, zal het je vragen om te gaan installeren.

Als je een meer recentere versie wilt installeren, kan je het via een binaire installer doen. Een macOS Git installer wordt onderhouden en is beschikbaar voor download op de Git webpagina, op <http://git-scm.com/download/mac>.

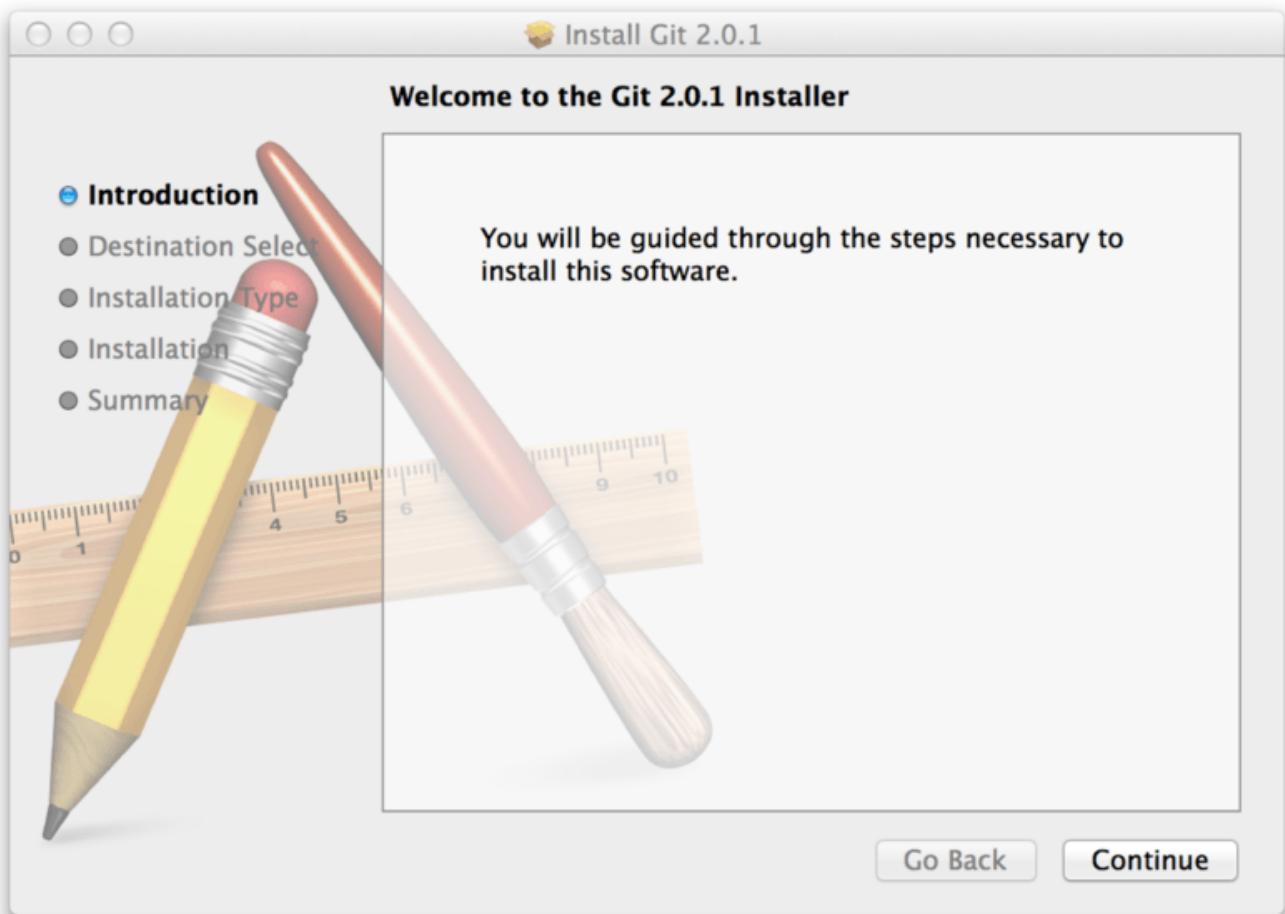


Figure 7. Git macOS Installer.

Je kunt het ook installeren als onderdeel van de GitHub voor Mac installatie. Hun GUI Git tool heeft een optie om de command line tools ook te installeren. Je kunt die tool van de GitHub voor Mac webpagina downloaden, op <http://mac.github.com>.

## Installeren op Windows

Er zijn ook een aantal manieren om Git te installeren op Windows. De meest officiële versie is beschikbaar voor download op de Git webpagina. Gewoon naar <http://git-scm.com/download/win> gaan en de download begint automatisch. Merk op dat dit een project is die Git for Windows heet, die gescheiden leeft van Git zelf; voor meer informatie hieromtrent, ga naar <http://msysgit.github.io/>.

Om een geautomatiseerde installatie te verkrijgen, kan je het [Git Chocolatey pakket](#) gebruiken. Merk op dat het Chocolatey pakket door vrijwilligers wordt onderhouden.

Een andere eenvoudige manier om Git te installeren is door de GitHub Desktop te gebruiken. De installer bevat een command line versie van Git zowel als de GUI. Het werkt ook goed met Powershell, en zet een degelijke credential cache op en goede CRLF instellingen. We zullen later meer vertellen over deze zaken, neem voor nu aan dat het zaken zijn die je wilt hebben. Je dit downloaden van de [GitHub Desktop website](#).

## Installeren vanaf broncode

Sommige mensen vinden het echter nuttig om Git vanaf de broncode te installeren, omdat je dan de meest recente versie krijgt. De binaire installers lopen vaak wat achter, alhoewel dit minder problemen oplevert, omdat Git in de laatste jaren behoorlijk volwassen geworden is.

Als je Git vanaf de broncode wilt installeren, dien je de volgende libraries te hebben waar Git van afhankelijk is: autotools, curl, zlib, openssl, expat, en libiconv. Bijvoorbeeld, als je op een systeem zit dat `dnf` heeft (zoals Fedora) of `apt-get` (zoals een op Debian gebaseerd systeem), kan je een van de volgende commando's gebruiken om alle minimale afhankelijkheden te installeren voor het compileren en installeren van de Git binaire bestanden:

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

Om ook de documenten in de verschillende formaten (doc, html, info) te kunnen toevoegen, zijn deze bijkomende afhankelijkheden nodig (Merk op: gebruikers van RHEL en RHEL-afgeleiden zoals CentOS en Scientific Linux zullen [de EPEL repository moeten activeren](#) om het `docbook2X` pakket te downloaden):

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

Als je een RPM-gebaseerde distributie (Fedora/RHEL/RHEL-afgeleiden), kan je ook het `getopt` pakket (die al is geïnstalleerd op een Debian-gebaseerde distro):

```
$ sudo dnf install getopt
$ sudo apt-get install getopt
```

Aanvullend, als je Fedora/RHEL/RHEL-afgeleide gebruikt, moet je ook dit doen

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

vanwege binaire naamsafwijkingen.

Als je alle benodigde afhankelijkheden hebt, kan je doorgaan en de laatst getagde release tarball oppakken van een van de vele plaatsen. Je kunt het via de kernel.org pagina krijgen, op <https://www.kernel.org/pub/software/scm/git>, of de mirror op de GitHub web pagina, op <https://github.com/git/git/releases>. Het is over het algemeen iets duidelijker aangegeven wat de laatste versie is op de GitHub pagina, en de kernel.org pagina heeft ook release signatures als je de download wilt verifiëren.

Daarna, compileren en installeren:

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

Als dit gebeurd is, kan je Git ook via Git zelf verkrijgen voor updates:

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

## Git klaarmaken voor eerste gebruik

Nu je Git op je computer hebt staan, is het handig dat je een paar dingen doet om je Git omgeving aan jouw voorkeuren aan te passen. Je hoeft deze instellingen normaliter maar één keer te doen, ze blijven hetzelfde als je een nieuwe versie van Git installeert. Je kunt ze ook op elk moment weer veranderen door de commando's opnieuw uit te voeren.

Git bevat standaard een stuk gereedschap genaamd `git config`, waarmee je de configuratie-eigenschappen kunt bekijken en veranderen, die alle aspecten van het uiterlijk en gedrag van Git regelen. Deze eigenschappen kunnen op drie verschillende plaatsen worden bewaard:

1. Het bestand `/etc/gitconfig`: Bevat eigenschappen voor elk gebruiker op de computer en al hun repositories. Als je de optie `--system` meegeeft aan `git config`, zal het de configuratiegegevens in dit bestand lezen en schrijven. (Omdat dit een systeem configuratiebestand betreft, moet je administrative of superuser privileges hebben om deze te kunnen wijzigen.)
2. Het bestand `~/.gitconfig` of `~/.config/git/config`: Eigenschappen voor jouw account. Je kunt Git dit bestand laten lezen en schrijven door de optie `--global` mee te geven, en dit heeft gevolgen voor *alle* repositories waarmee je op je systeem werkt.
3. Het configuratiebestand in de Gitdirectory (dus `.git/config`) van de repository die je op het moment gebruikt: Specifiek voor die ene repository. Je kunt git dit bestand laten lezen en schrijven door de optie `--local` mee te geven, maar dat is eigenlijk de standaard waarde. (Niet onverwacht: je moet ergens in een Git repository staan om deze optie juist te laten werken.)

Elk niveau heeft voorrang boven het voorgaande, dus waarden die in `.git/config` zijn gebruikt zullen worden gebruikt in plaats van die in `/etc/gitconfig`.

Op systemen met Windows zoekt Git naar het `.gitconfig`-bestand in de `$HOME` directory (`C:\Users\$USER` voor de meeste mensen). Het zoekt ook nog steeds naar `/etc/gitconfig`, maar dan gerelateerd aan de plek waar je MSys staat, en dat is de plek waar je Git op je Windowscomputer geïnstalleerd hebt. Als je versie 2.x of later gebruikt van Git for Windows, is er ook een systeem-niveau configuratie bestand in `C:\Documents and Settings\All Users\Application Data\Git\config` op Windows XP, en in `C:\ProgramData\Git\config` op Windows Vista en later. Dit configuratiebestand kan alleen gewijzigd worden met `git config -f <file>` als een admin.

## Jouw identiteit

Het eerste wat je zou moeten doen nadat je Git geïnstalleerd hebt, is je gebruikersnaam en e-mail adres invullen. Dat is belangrijk omdat elke commit in Git deze informatie gebruikt, en het onveranderlijk ingebed zit in de commits die je zult gaan maken:

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

Nogmaals, dit hoef je maar één keer te doen als je de `--global` optie erbij opgeeft, omdat Git die informatie zal gebruiken voor alles wat je doet op dat systeem. Als je een andere naam of e-mail wilt gebruiken voor specifieke projecten, kun je het commando uitvoeren zonder de `--global` optie als je in de directory van dat project zit.

Veel van de GUI tools zullen je helpen dit te doen als je ze voor de eerste keer aanroeft.

## Je editor

Nu Git weet wie je bent, kun je de standaard editor instellen die gebruikt zal worden als Git je een bericht in wil laten typen. Als dat niet is ingesteld, gebruikt Git de standaard editor van je systeem.

Als je een andere editor wilt gebruiken, zoals Emacs, kun je het volgende doen:

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

Als je, op een Windows systeem, een andere text editor wilt gebruiken, moet je het volledige pad naar de executable invullen. Dit kan verschillen, afhankelijk van hoe je editor is geleverd.

In het geval van Notepad++, een populaire editor, zal je waarschijnlijk de 32-bit versie willen gebruiken, omdat -op het moment van schrijven- niet alle plug-ins worden ondersteund door de 64-bit versie. Als je op een 32-bit Windows machine werkt, of je hebt een 64-bit editor op een 64-bit machine, zal je iets als dit typen:

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

Als je een 32-bit editor op een 64-bit systeem hebt, wordt het programma in `C:\Program Files (x86)` geïnstalleerd:

```
$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe'  
-multiInst -nosession"
```

 Vim, Emacs en Notepad++ zijn populaire tekst editors die vaak gebruikt worden door ontwikkelaars op Unixachtige systemen als Linux en macOS of een Windows systeem. Als je niet bekend bent met deze editors, zal je misschien moeten zoeken naar specifieke instructies hoe jouw favoriete editor voor Git in te richten.

 Als je je editor niet op deze manier inricht, zou het kunnen gebeuren dat je erg in verwarring raakt als Git deze probeert op te starten. Een voorbeeld op een Windows systeem zou een voortijdig beeindigde Git operatie kunnen zijn tijdens een edit die door Git opgestart is.

## Je instellingen controleren

Als je je instellingen wilt controleren, kan je het `git config --list` commando gebruiken voor een lijst met alle instellingen die Git vanaf die locatie kan vinden:

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

Je zult sommige sleutels misschien meerdere keren langs zien komen, omdat Git dezelfde sleutel uit verschillende bestanden heeft gelezen (bijvoorbeeld `/etc/gitconfig` en `~/.gitconfig`). In dit geval gebruikt Git de laatste waarde van elke unieke sleutel die het tegenkomt.

Je kan ook bekijken wat Git als instelling heeft bij een specifieke sleutel door `git config {sleutel}` in te voeren:

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

 Omdat Git dezelfde configuratie waarde kan lezen van meer dan een bestand, is het mogelijk dat je een onverwachte waarde ziet voor een van deze waarden en je niet weet waarom. In die gevallen kan je Git vragen naar de *bron* (origin) van die waarde, en het zal je vertellen welk configuratie bestand het laatste woord had in de bepaling van die waarde:

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

# Hulp krijgen

Als je ooit hulp nodig hebt met het gebruik van Git, zijn er twee gelijkwaardige manieren om de uitgebreide gebruiksaanwijzing (manpage) voor elk Git commando te krijgen:

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

Bijvoorbeeld, je kunt de gebruikershandleiding voor het `git config` commando krijgen door het volgende te typen:

```
$ git help config
```

Deze commando's zijn prettig omdat je ze overal kunt opvragen, zelfs als je offline bent. Als de manpage en dit boek niet genoeg zijn en je persoonlijke hulp nodig hebt, kan je de kanalen `#git` of `#github` (beiden Engelstalig) op het Freenode IRC netwerk (<irc.freenode.net>) proberen. In deze kanalen zijn regelmatig honderden mensen aangemeld die allemaal zeer ervaren zijn met Git en vaak bereid om te helpen.

Aanvullend, als je niet de uitgebreide manpage help nodig hebt, maar je alleen je geheugen wilt opfrissen over de beschikbare opties voor een Git commando, kan je de meer compacte “help” uitvoer opvragen met de `-h` of `--help` opties, zoals in:

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

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

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

# Samenvatting

Je zou nu een idee moeten hebben wat Git is en op welke manieren het verschilt van het centrale versiebeheersysteem dat je misschien eerder gebruikte. Je zou nu ook een werkende versie

van Git op je systeem moeten hebben dat is ingesteld met jouw persoonlijk identiteit. Nu is het tijd om een aantal grondbeginselen van Git te gaan leren.

# Git Basics

Als je slechts één hoofdstuk kunt lezen om met Git aan de slag te gaan, dan is deze het. In dit hoofdstuk worden alle basiscommando's behandeld die je nodig hebt om het leeuwendeel van de dingen te doen waarmee je uiteindelijk je tijd met Git zult doorbrengen. Als je dit hoofdstuk doorgenomen hebt, zul je een repository kunnen configureren en initialiseren, bestanden beginnen en stoppen te volgen en veranderingen te *stagen* en *committen*. We laten ook zien hoe je Git kunt instellen zodat het bepaalde bestanden en bestandspatronen negeert, hoe je vergissingen snel en gemakkelijk ongedaan kunt maken, hoe je de geschiedenis van je project kan doorlopen en wijzigingen tussen commits kunt zien, en hoe je kunt pushen naar en pullen van niet lokale repositories.

## Een Git repository verkrijgen

Je kunt op twee manieren een Git project verkrijgen.

1. Je kunt een lokale directory nemen die nog niet onder versiebeheer zit, en deze in een Git repository veranderen, of
2. Je kunt een bestaande Git repository *klonen* (clone) van een andere plek

## Een repository initialiseren in een bestaande directory

Als je een project directory hebt, dat op dit moment nog niet onder versiebeheer staat en je wilt het met Git beheren, moet je eerst in die project directory gaan. Als je dit nog nooit eerder gedaan hebt, zit het er iets anders uit afhankelijk van het systeem waaronder je werkt:

voor Linux:

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

voor Mac:

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

voor Windows:

```
$ cd /c/user/my_project
```

en type:

```
$ git init
```

Dit maakt een nieuwe subdirectory met de naam `.git` aan, die alle noodzakelijke repository bestanden bevat, een Git repository raamwerk. Op dit moment wordt nog niets in je project

gevolgd. (Zie [Git Binnenwerk](#) voor meer informatie over welke bestanden er precies in de `.git` directory staan, die je zojuist gemaakt hebt.)

Als je de versies van bestaande bestanden wilt gaan beheren (in plaats van een lege directory), dan zul je die bestanden moeten beginnen te tracken en een eerste commit doen. Dit kun je bereiken door een paar `git add` commando's waarin je de te volgen bestanden specificeert, gevolgd door een `git commit`:

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

We zullen zodadelijk beschrijven wat deze commando's doen. Op dit punt heb je een Git repository met gevogelde (tracked) bestanden en een initiële commit.

## Een bestaande repository klonen

Als je een kopie wilt van een bestaande Git repository, bijvoorbeeld een project waaraan je wilt bijdragen, dan is `git clone` het commando dat je nodig hebt. Als je bekend bent met andere versiebeheersystemen zoals Subversion, dan zal het je opvallen dat het commando "clone" is en niet "checkout". Dit is een belangrijk onderscheid: in plaats van alleen maar een werk-kopie, ontvangt Git een volledige kopie van bijna alle gegevens die de server heeft. Elke versie van ieder bestand in de hele geschiedenis van een project wordt standaard binnengehaald als je `git clone` aanroeft. In feite kun je, als de schijf van de server kapot gaat, een kloon van een willekeurig werkstation gebruiken om de server terug in de status te brengen op het moment van klonen (al zou je wel wat hooks aan de kant van de server en dergelijke verliezen, maar alle versies van alle bestanden zullen er zijn; zie [Git op een server krijgen](#) voor meer informatie).

Je kloont een repository met `git clone <url>`. Bijvoorbeeld, als je de linkbare Git bibliotheek genaamd libgit2 wilt klonen, kun je dit als volgt doen:

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

Dat maakt een directory genaamd `libgit2` aan, initialiseert hierin een `.git` directory, haalt alle data voor die repository binnen en doet een checkout van een werk-kopie van de laatste versie. Als je in de nieuwe `libgit2` directory gaat kijken zal je de project bestanden vinden, klaar om gebruikt of aan gewerkt te worden. Als je de repository in een directory met een andere naam dan `libgit2` wilt klonen, dan kun je dit met een extra argument meegeven met de nieuwe directorynaam:

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

Dat commando doet hetzelfde als het vorige, maar dan heet de doeldirectory `mylibgit`.

Git heeft een aantal verschillende transport protocollen die je kunt gebruiken. Het vorige voorbeeld maakt gebruik van het `https://` protocol, maar je kunt ook `git://` of `gebruiker@server:/pad/naar/repo.git` tegenkomen, dat het SSH transport protocol gebruikt. [Git op](#)

[een server krijgen](#) zal alle beschikbare opties introduceren die de server kan aanbieden om je toegang tot de Git repositories te geven, met daarbij de voors en tegens van elk.

## Wijzigingen aan de repository vastleggen

Je hebt een *echte* Git repository op je lokale machine en een checkout of *werkopname* van alle bestanden binnen dat project voor je neus. In een reguliere situatie, ga je wijzigingen maken en begin je snapshots te committen naar je repository elke keer dat het project een status bereikt die je wilt vastleggen.

Onthoud dat elk bestand in je werkdirectory in een van twee statussen kan verkeren: *tracked* of *untracked*. Tracked bestanden zijn bestanden die in de laatste snapshot zaten; ze kunnen ongewijzigd (unmodified), gewijzigd (modified) of klaargezet (staged) zijn. Met andere woorden: tracked bestanden zijn bestanden waarvan Git het bestaan kent.

Untracked bestanden zijn al het andere; elk bestand in je werkdirectory dat niet in je laatste snapshot en niet in je staging area zit. Als je een repository voor het eerst kloont, zullen alle bestanden tracked en unmodified zijn, omdat je ze zojuist uitgechecked hebt en nog niets gewijzigd hebt.

Zodra je bestanden wijzigt, ziet Git ze als modified omdat je ze veranderd hebt sinds je laatste commit. Als je aan het werk bent, ga je bepaalde gewijzigde bestanden staggen en daarna commit al deze gestagede bestanden, en de cyclus begint weer van voor af aan.

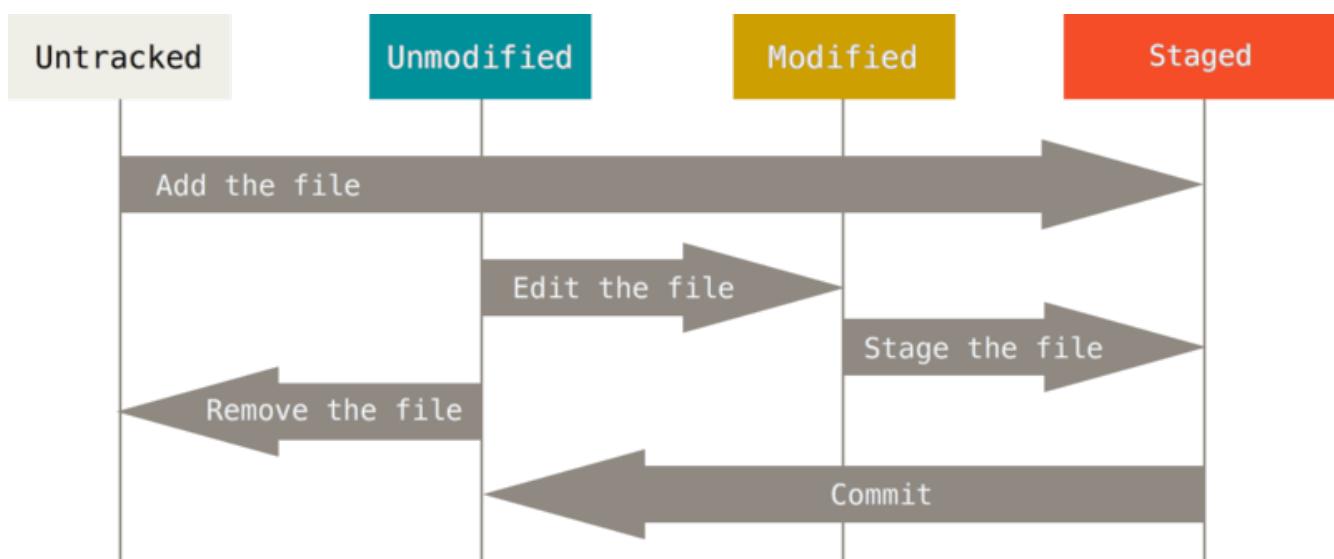


Figure 8. De levenscyclus van de status van je bestanden.

## De status van je bestanden controleren

Het commando dat je voornamelijk zult gebruiken om te bepalen welk bestand zich in welke status bevindt is `git status`. Als je dit commando direct na het clonen uitvoert, dan zal je zoiets als het volgende zien:

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

Dit betekent dat je een schone werkdirectory hebt, met andere woorden er zijn geen tracked bestanden die gewijzigd zijn. Git ziet ook geen untracked bestanden, anders zouden ze hier getoond worden. Als laatste vertelt het commando op welke tak (branch) je nu zit en informeert je dat het niet is afgeweken van dezelfde branch op de server. Voor nu is deze branch altijd “master”, dat is de standaard; besteed daar voor nu nog geen aandacht aan. In [Branchen in Git](#) wordt gedetailleerd ingegaan op branches en referenties.

Stel dat je een nieuw bestand toevoegt aan je project, een simpel `README` bestand. Als het bestand voorheen nog niet bestond, en je doet `git status`, dan zul je het niet getrackte bestand op deze manier zien:

```
$ echo 'My Project' > README
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Untracked files:
  (use "git add <file>..." to include in what will be committed)

    README

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

Je kunt zien dat het nieuwe `README` bestand untracked is, omdat het onder de “Untracked files” kop staat in je status uitvoer. Untracked betekent eigenlijk dat Git een bestand ziet dat je niet in de vorige snapshot (commit) had; Git zal het niet aan je commit snapshots toevoegen totdat jij dit expliciet aangeeft. De reden hiervoor is dat je niet per ongeluk gegenereerde binaire bestanden toevoegt, of andere bestanden die je niet had willen toevoegen. Je wilt dit `README` bestand in het vervolg wel meenemen, dus laten we het gaan tracken.

## Nieuwe bestanden volgen (tracking)

Om een nieuw bestand te beginnen te tracken, gebruik je het commando `git add`. Om de `README` te tracken, voer je dit uit:

```
$ git add README
```

Als je het status commando nogmaals uitvoert, zie je dat je `README` bestand nu tracked en staged is voor committen:

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

    new file: README
```

Je kunt zien dat het gestaged is, omdat het onder de kop “Changes to be committed” staat. Als je nu een commit doet, zal de versie van het bestand zoals het was ten tijde van je `git add` commando in de historische snapshot toegevoegd worden. Je zult je misschien herinneren dat, toen je `git init` eerder uitvoerde, je daarna `git add (bestanden)` uitvoerde; dat was om bestanden in je directory te beginnen te tracken. Het `git add` commando beschouwt een padnaam als een bestand of een directory. Als de padnaam een directory is, dan voegt het commando alle bestanden in die directory recursief toe.

## Gewijzigde bestanden stagen

Laten we een getrackte bestand wijzigen. Als je een reeds getrackt bestand genaamd `CONTRIBUTING.md` wijzigt, en dan het `git status` commando nog eens uitvoert, krijg je iets dat er zo uitziet:

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

    new file: README

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

    modified: CONTRIBUTING.md
```

Het “`CONTRIBUTING.md`” bestand verschijnt onder een sectie genaamd “Changes not staged for commit”, wat inhoudt dat een bestand dat wordt getrackt is gewijzigd in de werkdirctory, maar nog niet is gestaged. Om het te staggen, voer je het `git add` commando uit. `git add` is een veelzijdig commando: je gebruikt het om nieuwe bestanden te laten tracken, om bestanden te staggen, en om andere dingen zoals een bestand met een mergeconflict als opgelost te markeren. Het kan behulpzaam zijn om het commando te zien als “voeg deze inhoud toe aan de volgende commit” in plaats van “voeg dit bestand toe aan het project”. Laten we nu `git add` uitvoeren om het `CONTRIBUTING.md` bestand te staggen, en dan nog eens `git status` uitvoeren:

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

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

Beide bestanden zijn gestaged en zullen met je volgende commit meegaan. Stel nu dat je je herinnert dat je nog een kleine wijziging in **CONTRIBUTING.md** wilt maken voordat je het commit. Je opent het opnieuw en maakt die wijziging, en dan ben je klaar voor de commit. Alhoewel, laten we **git status** nog een keer uitvoeren:

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

    new file: README
    modified: CONTRIBUTING.md

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

    modified: CONTRIBUTING.md
```

Asjemenou?! Nu staat **CONTRIBUTING.md** bij de staged *en* unstaged genoemd. Hoe is dat mogelijk? Het blijkt dat Git een bestand precies zoals het is staged wanneer je het **git add** commando uitvoert. Als je nu commit, dan zal de versie van **CONTRIBUTING.md** zoals het was toen je voor 't laatst **git add** uitvoerde worden toegevoegd in de commit, en niet de versie van het bestand zoals het eruit ziet in je werkdirectory toen je **git commit** uitvoerde. Als je een bestand wijzigt nadat je **git add** uitvoert, dan moet je **git add** nogmaals uitvoeren om de laatste versie van het bestand te staggen:

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

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

## Korte status

Alhoewel de `git status` uitvoer redelijk uitgebreid is, is het ook nogal breedsprakig. Git heeft ook een vlag voor een korte status, zodat je je wijzigingen in een meer compact overzicht ziet. Als je `git status -s` of `git status --short` typt krijg je een veel simpeler uitvoer van het commando.

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

Nieuwe bestanden die nog niet worden getractt hebben een `??` naast zich, nieuwe bestanden die aan de staging area zijn toegevoegd hebben een `A`, gewijzigde bestanden een `M` enzovoort. Er zijn twee kolommen in de uitvoer: de linker kolom geeft de status van de staging area weer en de rechter kolom de status van je werk directory. Als voorbeeld in de uitvoer is het `README` bestand gewijzigd in de werk directory maar nog niet gestaged, terwijl het `lib/simplegit.rb` bestand gewijzigd en gestaged is. `Rakefile` is gewijzigd, gestaged en weer gewijzigd, dus er zijn daar wijzigingen die zowel gestaged als ongestaged zijn.

## Bestanden negeren

Vaak zul je een klasse bestanden hebben waarvan je niet wilt dat Git deze automatisch toekoegt of zelfs maar als untracked toont. Dit zijn doorgaans automatisch gegenereerde bestanden zoals logbestanden of bestanden die geproduceerd worden door je bouwsysteem. In die gevallen kun je een bestand genaamd `.gitignore` maken, waarin patronen staan die die bestanden passen. Hier is een voorbeeld van een `.gitignore` bestand:

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

De eerste regel vertelt Git om ieder bestand te negeren waarvan de naam eindigt op een “.o” of “.a” (object en archief bestanden die het product kunnen zijn van het bouwen van je code). De tweede regel vertelt Git dat ze alle bestanden moet negeren die eindigen op een tilde (`~`), wat gebruikt

wordt door editors zoals Emacs om tijdelijke bestanden aan te geven. Je kunt ook log, tmp of een pid directory toevoegen, automatisch gegenereerde documentatie, enzovoort. Een `.gitignore` bestand aanmaken voordat je gaat beginnen is over het algemeen een goed idee, zodat je niet per ongeluk bestanden commit die je echt niet in je Git repository wilt hebben.

De regels voor patronen die je in het `.gitignore` bestand kunt zetten zijn als volgt:

- Lege regels of regels die beginnen met een `#` worden genegeerd.
- Standaard expansie (glob) patronen worden geaccepteerd, en worden recursief toegepast over de gehele werk directory.
- Je kunt patronen met een schuine streep (`/`) laten beginnen om recursie te voorkomen.
- Je mag patronen laten eindigen op een schuine streep (`/`) om een directory aan te duiden.
- Je mag een patroon ontkennend maken door het te laten beginnen met een uitroepteken (`!`).

Expansie (`glob`) patronen zijn vereenvoudigde reguliere expressies die in shell-omgevingen gebruikt worden. Een asterisk (\*) komt overeen met nul of meer karakters, `[abc]` komt overeen met ieder karakter dat tussen de blokhaken staat (in dit geval a, b of c), een vraagteken (?) komt overeen met een enkel karakter en blokhaken waartussen karakters staan die gescheiden zijn door een streepje (`[0-9]`) komen overeen met ieder karakter dat tussen die karakters zit (in dit geval 0 tot en met 9). Je kunt ook twee asteriskken gebruiken om geneste directories aan te geven: `a/**/z` komt overeen met `a/z, a/b/z, a/b/c/z` en zo verder.

Hier is nog een voorbeeld van een `.gitignore` bestand:

```
# ignore all .a files
*.a

# but do track lib.a, even though you're ignoring .a files above
!lib.a

# only ignore the TODO file in the current directory, not subdir/TODO
/TODO

# ignore all files in any directory named build
build/

# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt

# ignore all .pdf files in the doc/ directory and any of its subdirectories
doc/**/*.pdf
```



GitHub onderhoudt een redelijk uitgebreide lijst van goede voorbeeld `.gitignore` bestanden van projecten en talen op <https://github.com/github/gitignore> als je een goed beginpunt voor jouw project wilt hebben.

## Je staged en unstaged wijzigingen bekijken

Als het `git status` commando te vaag is voor je - je wilt precies weten wat je veranderd hebt en niet alleen welke bestanden veranderd zijn - dan kan je het `git diff` commando gebruiken. We zullen `git diff` later in meer detail bespreken, maar je zult dit commando het meest gebruiken om deze twee vragen te beantwoorden: Wat heb je veranderd maar nog niet gestaged? En wat heb je gestaged en sta je op het punt te committen? Waar `git status` deze vragen heel algemeen beantwoordt door de bestandsnamen te tonen, laat `git diff` je de exacte toegevoegde en verwijderde regels zien, de patch, als het ware.

Stel dat je het `README` bestand opnieuw verandert en staget, en dan het `CONTRIBUTING.md` bestand verandert zonder het te stagen. Als je het `git status` commando uitvoert, dan zie je nogmaals zo iets als dit:

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

    modified: README

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

    modified: CONTRIBUTING.md
```

Om te zien wat je gewijzigd maar nog niet gestaged hebt, type je `git diff` in zonder verdere argumenten:

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

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

Dat commando vergelijkt wat er in je werkdirectory zit met wat er in je staging area zit. Het

resultaat laat je zien welke wijzigingen je gedaan hebt, die je nog niet gestaged hebt.

Als je wilt zien wat je gestaged hebt en in je volgende commit zal zitten, dan kun je `git diff --staged` gebruiken. Dit commando vergelijkt je gestagede wijzigingen met je laatste commit:

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

Het is belangrijk om op te merken dat `git diff` zelf niet alle wijzigingen sinds je laatste commit laat zien, alleen wijzigingen die nog niet gestaged zijn. Als je al je wijzigingen gestaged hebt, zal `git diff` geen uitvoer geven.

Nog een voorbeeld. Als je het `CONTRIBUTING.md` bestand staget en vervolgens verandert, dan kun je `git diff` gebruiken om de wijzigingen in het bestand te zien dat gestaged is en de wijzigingen die niet gestaged zijn. Stel onze omgeving ziet er zo uit:

```
$ git add CONTRIBUTING.md
$ echo '# test line' >> CONTRIBUTING.md
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   CONTRIBUTING.md

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

    modified:   CONTRIBUTING.md
```

Nu kun je `git diff` gebruiken om te zien wat nog niet gestaged is:

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

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

en `git diff --cached` om te zien wat je tot nog toe gestaged hebt (--staged en --cached zijn synonym):

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

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

#### *Git Diff in een externe tool*

We zullen in de rest van het boek doorgaan met het op verschillende manieren gebruiken van het `git diff` commando. Er is een andere manier om naar deze diffs te kijken als je een grafische of externe diff viewer prefereert. Als je de `git difftool` aanroeft in plaats van `git diff`, kan je elk van deze diffs in software als emerge, vimdiff and vele andere zien (inclusief commerciële producten). Roep het `git difftool --tool-help` aan om te zien wat er op jouw systeem beschikbaar is.



## Je wijzigingen committen

Nu je staging area gevuld is zoals jij het wilt, kun je de wijzigingen committen. Onthoud dat alles wat niet gestaged is, dus elk bestand dat je gemaakt of gewijzigd hebt en waarop je nog geen `git add` uitgevoerd hebt, niet in deze commit mee zal gaan. Ze zullen als gewijzigde bestanden op je schijf blijven staan. Stel in dit geval dat, toen je de laatste keer `git status` uitvoerde, je zag dat alles gestaged was. Dus je bent er klaar voor om je wijzigingen te committen. De makkelijkste manier om te committen is om `git commit` in te typen:

```
$ git commit
```

Dit start de door jou gekozen editor op. (Dit wordt bepaald door de `EDITOR` omgevingsvariabele in je shell, meestal vim of emacs, alhoewel je kunt instellen op welke editor je wilt gebruiken met het `git config --global core.editor` commando zoals je hier [Aan de slag](#) gezien hebt).

De editor laat de volgende tekst zien (dit voorbeeld is een Vim scherm):

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
#
# Changes to be committed:
#   new file: README
#   modified: CONTRIBUTING.md
#
~
~
~
~
".git/COMMIT_EDITMSG" 9L, 283C
```

Je kunt zien dat de standaard commit boodschap de laatste output van het `git status` commando als commentaar bevat en een lege regel bovenaan. Je kunt deze commentaren verwijderen en je eigen commit boodschap intypen, of je kunt ze laten staan om je eraan te helpen herinneren wat je aan het committen bent. (Om een meer expliciete herinnering van je wijzigingen te zien kun je de `-v` optie meegeven aan `git commit`. Als je dit doet zet Git de diff van je veranderingen in je editor zodat je precies kunt zien welke wijzigingen je gaat committen.) Als je de editor verlaat, creëert Git je commit boodschap (zonder de commentaren of de diff).

Als alternatief kun je de commit boodschap met het `commit` commando meegeven door hem achter de `-m` vlag te specificeren, zoals hier:

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

Nu heb je je eerste commit gemaakt! Je kunt zien dat de commit je wat uitvoer over zichzelf heeft gegeven: op welke branch je gecommit hebt (`master`), welke SHA-1 checksum de commit heeft (`463dc4f`), hoeveel bestanden er veranderd zijn, en statistieken over toegevoegde en verwijderde regels in de commit.

Onthoud dat commit de snapshot, die je in je staging area hebt gezet, opslaat. Alles wat je niet gestaged hebt staat nog steeds gewijzigd; je kunt een volgende commit doen om het aan je geschiedenis toe te voegen. Elke keer dat je een commit doet, leg je een snapshot van je project vast

waarnaar je later terug kunt draaien of waarmee je kunt vergelijken.

## De staging area overslaan

Alhoewel het ontzettend nuttig kan zijn om commits precies zoals je wilt te maken, is de staging area soms iets ingewikkelder dan je in je workflow nodig hebt. Als je de staging area wilt overslaan, dan kan je met Git makkelijk de route inkorten. Door de `-a` optie aan het `git commit` commando mee te geven zal Git automatisch ieder bestand dat al getrackt wordt voor de commit staggen, zodat je het `git add` gedeelte kunt overslaan:

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

    modified:   CONTRIBUTING.md

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

Merk op dat je nu geen `git add` op het “CONTRIBUTING.md” bestand hoeft te doen voordat je commit. Dat is omdat de `-a` vlag alle gewijzigde bestanden insluit. Dat is handig, maar wees voorzichtig; soms zal deze vlag veroorzaken dat je wijzigingen meeneemt die je eigenlijk niet had gewild.

## Bestanden verwijderen

Om een bestand uit Git te verwijderen, moet je het van de getrackte bestanden verwijderen (of om precies te zijn: verwijderen van je staging area) en dan een commit doen. Het `git rm` commando doet dat, en verwijdert het bestand ook van je werk directory zodat je het de volgende keer niet als een untracked bestand ziet.

Als je het bestand simpelweg verwijdert uit je werk directory, zal het te zien zijn onder het “Changes not staged for commit” - (dat wil zeggen, *unstaged*) gedeelte van je `git status` uitvoer:

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

    deleted:   PROJECTS.md

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

Als je daarna `git rm` uitvoert, zal de verwijdering van het bestand gestaged worden:

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

    deleted:   PROJECTS.md
```

Als je de volgende keer een commit doet, zal het bestand verdwenen zijn en niet meer getract worden. Als je het bestand veranderd hebt en al aan de staging area toegevoegd, dan zul je de verwijdering moeten forceren met de `-f` optie. Dit is een veiligheidsmaatregel om te voorkomen dat je per ongeluk data die nog niet in een snapshot zit, en dus niet teruggehaald kan worden uit Git, weggooit.

Een ander handigheidje dat je misschien wilt gebruiken is het bestand in je werkdirctory houden, maar van je staging area verwijderen. Met andere woorden, je wilt het bestand misschien op je harde schijf bewaren, maar niet dat Git het bestand nog trackt. Dit is erg handig als je iets vergeten bent aan je `.gitignore` bestand toe te voegen, en het per ongeluk gestaged hebt, zoals een groot logbestand, of een serie `.a` gecompileerde bestanden. Gebruik de `--cached` optie om dit te doen:

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

Je kunt bestanden, directories en bestandspatronen aan het `git rm` commando meegeven. Dat betekent dat je zoets als dit kunt doen:

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

Let op de backslash (\) voor de \*. Dit is nodig omdat Git zijn eigen bestandsnaam-expansie doet, naast die van je shell. Dit commando verwijdert alle bestanden die de `.log` extensie hebben in de

`log/` directory. Of, je kunt zo iets als dit doen:

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

Dit commando verwijdert alle bestanden die eindigen met `~`.

## Bestanden verplaatsen

Anders dan vele andere VCS systemen, traceert Git niet expliciet verplaatsingen van bestanden. Als je een bestand een nieuwe naam geeft in Git, is er geen metadata opgeslagen in Git die vertelt dat je het bestand hernoemd hebt. Maar Git is slim genoeg om dit alsnog te zien, we zullen bestandsverplaatsing-detectie wat later behandelen.

Het is daarom een beetje verwarringend dat Git een `mv` commando heeft. Als je een bestand wilt hernoemen in Git, kun je zo iets als dit doen

```
$ git mv file_from file_to
```

en dat werkt prima. Sterker nog, als je zo iets als dit uitvoert en naar de status kijkt, zul je zien dat Git het als een hernoemd bestand beschouwt:

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

    renamed:    README.md -> README
```

Maar dat is gelijk aan het uitvoeren van het volgende:

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

Git komt er impliciet achter dat het om een hernoemd bestand gaat, dus het maakt niet uit of je een bestand op deze manier hernoemt of met het `mv` commando. Het enige echte verschil is dat het `mv` commando slechts één commando is in plaats van drie - het is een gemaksfunctie. Belangrijker nog is dat je iedere applicatie kunt gebruiken om een bestand te hernoemen, en de add/rm later kunt afhandelen voordat je commit.

## De commit geschiedenis bekijken

Nadat je een aantal commits gemaakt hebt, of als je een repository met een bestaande commit-

geschiedenis gekloond hebt, zul je waarschijnlijk terug willen zien wat er gebeurd is. De meest eenvoudige en krachtige tool om dit te doen is het `git log` commando.

Deze voorbeelden maken gebruik van een eenvoudig project genaamd “simplegit”. Om het project op te halen, voer je dit uit

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

Als je `git log` in dit project uitvoert, zou je uitvoer moeten krijgen die er ongeveer zo uitziet:

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

    changed the version number

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

    removed unnecessary test

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

    first commit
```

Zonder argumenten toont `git log` standaard de commits die gedaan zijn in die repository, in omgekeerd chronologische volgorde, dus de meest recente wijziging bovenaan. Zoals je kunt zien, toont dit commando iedere commit met zijn SHA-1 checksum, de naam van de auteur en zijn e-mail, de datum van opslaan, en de commit boodschap.

Een gigantisch aantal en variëteit aan opties zijn beschikbaar voor het `git log` commando om je precies te laten zien waar je naar op zoek bent. Hier laten we je een aantal van de meest gebruikte opties zien.

Een van de meest behulpzame opties is `-p` of `--patch`, wat het verschil (de *patch*) uitvoer laat zien van de dingen die in iedere commit geïntroduceerd zijn. Je kunt ook `-2` gebruiken, om alleen de laatste twee items te laten zien:

```

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

    changed the version number

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

```

```

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

```

```

    removed unnecessary test

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
  end

end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end

```

Deze optie toont dezelfde informatie, maar dan met een diff direct volgend op elk item. Dit is erg handig voor een code review, of om snel te zien wat er tijdens een reeks commits gebeurd is die een medewerker toegevoegd heeft. Je kunt ook een serie samenvattende opties met `git log` gebruiken. Bijvoorbeeld, als je wat verkorte statistieken bij elke commit wilt zien, kun je de `--stat` optie gebruiken:

```

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

    changed the version number

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

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

    removed unnecessary test

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

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

    first commit

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

```

Zoals je ziet, drukt de `--stat` optie onder elke commit een lijst gewijzigde bestanden af, hoeveel bestanden gewijzigd zijn, en hoeveel regels in die bestanden zijn toegevoegd en verwijderd. Het toont ook een samenvatting van de informatie aan het einde.

Een andere handige optie is `--pretty`. Deze optie verandert de log uitvoer naar een ander formaat dan de standaard. Er zijn al een paar voorgebouwde opties voor je beschikbaar. De `oneline` optie drukt elke commit op een eigen regel af, wat handig is als je naar veel commits kijkt. Daarnaast tonen de `short`, `full` en `fuller` opties de output in grofweg hetzelfde formaat, maar respectievelijk met minder of meer informatie:

```

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

```

De meest interessante optie is `format`, waarmee je je eigen log-uitvoer-formaat kunt specificeren. Dit is in het bijzonder handig als je output aan het genereren bent voor automatische verwerking; omdat je explicet het formaat kunt specificeren, weet je dat het niet zal veranderen bij volgende

versies van Git:

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

Nuttige opties voor `git log --pretty=format` toont een aantal handige opties die aan format gegeven kunnen worden.

Table 1. Nuttige opties voor `git log --pretty=format`

Optie	Omschrijving van de Output
%H	Commit hash
%h	Afgekorte commit hash
%T	Tree hash
%t	Afgekorte tree hash
%P	Parent hashes
%p	Afgekorte parent hashes
%an	Auteur naam
%ae	Auteur e-mail
%ad	Auteur datum (format respecteert de --date= optie)
%ar	Auteur datum, relatief
%cn	Committer naam
%ce	Committer email
%cd	Committer datum
%cr	Committer datum, relatief
%s	Onderwerp

Je zult je misschien afvragen wat het verschil is tussen *auteur* en *committer*. De auteur is de persoon die het werk oorspronkelijk geschreven heeft, terwijl de committer de persoon is die de patch als laatste heeft toegepast. Dus als je een patch naar een project stuurt en een van de kernleden past de patch toe, krijgen jullie beiden de eer, jij als de auteur en het kernlid als de committer. We gaan hier wat verder op in in [Gedistribueerd Git](#).

De `oneline` en `format` opties zijn erg handig in combinatie met een andere `log` optie genaamd `--graph`. Deze optie maakt een mooie ASCII grafiek waarin je branch- en merge-geschiedenis getoond worden:

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

Dit type uitvoer zal interessanter worden als we branching en merging gaan behandelen in het volgende hoofdstuk.

Dit zijn slechts een paar simpele uitvoer formaat opties voor `git log`; er zijn er nog veel meer. [Gebruikelijke opties bij git log](#) toont de opties waarover we het tot nog toe gehad hebben, en wat veelvoorkomende formaat opties die handig kunnen zijn, samen met hoe ze de uitvoer van het `log` commando veranderen.

*Table 2. Gebruikelijke opties bij git log*

Optie	Omschrijving
<code>-p</code>	Toon de patch geïntroduceerd bij iedere commit
<code>--stat</code>	Toon statistieken voor gewijzigde bestanden per commit.
<code>--shortstat</code>	Toon alleen de gewijzigde/ingevoegde/verwijderde regel van het <code>--stat</code> commando.
<code>--name-only</code>	Toon de lijst van bestanden die gewijzigd zijn na de commit-informatie.
<code>--name-status</code>	Toon ook de lijst van bestanden die beïnvloed zijn door de toegevoegde/gewijzigde/verwijderde informatie.
<code>--abbrev-commit</code>	Toon alleen de eerste paar karakters van de SHA-1 checksum in plaats van alle 40.
<code>--relative-date</code>	Toon de datum in een relatief formaat (bijvoorbeeld, “2 weeks ago”), in plaats van het volledige datum formaat.
<code>--graph</code>	Toon een ASCII grafiek van de branch- en merge-geschiedenis naast de log uitvoer.
<code>--pretty</code>	Toon commits in een alternatief formaat. De opties bevatten <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> , en <code>format</code> (waarbij je je eigen formaat specificeert).
<code>--oneline</code>	Kort voor <code>--pretty=oneline --abbrev-commit</code> in combinatie met elkaar.

## Log uitvoer beperken

Naast het formatteren van de uitvoer, heeft `git log` nog een aantal nuttige beperkende opties; dat wil zeggen, opties die je een subset van de commits tonen. Je hebt zo'n optie al gezien: de `-2` optie, die slechts de laatste twee commits laat zien. Sterker nog: je kunt `-<n>` doen, waarbij `n` een geheel getal is om de laatste `n` commits te laten zien. In de praktijk zul je deze vorm weinig gebruiken,

omdat Git standaard alle uitvoer door een pager (pagineer applicatie) stuurt zodat je de log-uitvoer pagina voor pagina ziet.

Dat gezegd hebbende, zijn de tijd-limiterende opties zoals `--since` en `--until` erg handig. Dit commando bijvoorbeeld, geeft een lijst met commits die gedaan zijn gedurende de afgelopen twee weken:

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

Dit commando werkt met veel formaten: je kunt een specifieke datum kiezen zoals "2008-01-15", of een relatieve datum zoals "2 years 1 day 3 minutes ago".

Je kunt ook de lijst met commits filteren op bepaalde criteria. De `--author` optie laat je filteren op een specifieke auteur, en de `--grep` optie laat je op bepaalde zoekwoorden filteren in de commit boodschappen.



Je kunt meer dan een instantie van zowel de `--author` en `-grep` zoekcriteria opgeven, wat de uitvoer tot commits beperkt die *enig* patroon van de `--author` en *enig* patroon van de `--grep` past; echter, het toevoegen van de `--all-match` optie beperkt de uitvoer nog eens extra tot die commits die *alle* `--grep` patronen passen.

Een andere echt handige om als filter mee te geven is de `-S` optie (beter bekend als Git's "beitel" optie), die een tekenreeks accepteert en alleen de commits laat zien met een wijziging aan de code die aantal voorkomsten van die reeks wijzigt. Bijvoorbeeld, als je de laatste commit zou willen vinden die een referentie aan een specifieke functie heeft toegevoegd of verwijderd, zou je dit kunnen aanroepen:

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

De laatste echt handige optie om aan `git log` als filter mee te geven is een pad. Als je een directory of bestandsnaam opgeeft, kun je de log output limiteren tot commits die een verandering introduceren op die bestanden. Dit is altijd de laatste optie en wordt over het algemeen vooraf gegaan door dubbele streepjes `(--)` om de paden van de opties te scheiden.

In [Opties om de uitvoer van `git log` te beperken](#) laten we deze en een paar andere veel voorkomende opties zien als referentie.

Table 3. Opties om de uitvoer van `git log` te beperken

Optie	Omschrijving
<code>-(n)</code>	Laat alleen de laatste n commits zien
<code>--since, --after</code>	Beperk de commits tot degenen waarvan de CommitDate op of na de gegeven datum/tijd ligt.
<code>--until, --before</code>	Beperk de commits tot degenen waarvan de CommitDate op of voor de gegeven datum/tijd ligt.

Optie	Omschrijving
--author	Laat alleen de commits zien waarvan de auteur bij de gegeven tekst past.
--committer	Laat alleen de commits zien waarvan de committer bij de gegeven tekst past.
--grep	Laat alleen de commits zien met een commit bericht met daarin de gegeven tekst
-S	Laat alleen de commits zien waarbij de gegeven tekst werd toegevoegd of verwijderd

Als voorbeeld, als je de commits zou willen zien waarin test bestanden in de Git broncode historie gecommit zijn door Junio Hamano die niet gemerged waren in oktober 2008, zou je zo iets als dit kunnen opgeven:

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

Van de bijna 40.000 commits in de Git broncode historie, laat dit commando de 6 zien die aan deze criteria voldoen.

#### *Het tonen van merge commits voorkomen*



Afhankelijk van de workflow die in jouw repository wordt gebruikt, is het mogelijk dat een significant percentage van de commits in jouw log history alleen maar merge commits zijn, die over het algemeen weinig informatie toevoegen. Om te voorkomen dat deze merge commits je uitvoer van de log history vervuilen, kan je simpelweg de optie **--no-merges** toevoegen.

## Dingen ongedaan maken

Op enig moment wil je misschien iets ongedaan maken. Hier zullen we een aantal basis-tools laten zien om veranderingen die je gemaakt hebt weer ongedaan te maken. Maar pas op, je kunt niet altijd het ongedaan maken weer ongedaan maken. Dit is één van de weinige situaties in Git waarbij je werk kwijt kunt raken als je het verkeerd doet.

Een van de veel voorkomende acties die ongedaan gemaakt moeten worden vindt plaats als je te vroeg commit en misschien vergeten bent een aantal bestanden toe te voegen, of je verknalt je commit boodschap. Als je opnieuw wilt committen, de aanvullende wijzigingen maken die je was vergeten, deze staggen en weer committen, dan kun je commit met de **--amend** optie uitvoeren:

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

Dit commando neemt je staging area en gebruikt dit voor de commit. Als je geen veranderingen sinds je laatste commit hebt gedaan (bijvoorbeeld, je voert dit commando meteen na je laatste commit uit), dan zal je snapshot er precies hetzelfde uitzien en zal je commit boodschap het enige zijn dat je verandert.

Dezelfde commit-boodschap editor start op, maar deze bevat meteen de boodschap van je vorige commit. Je kunt de boodschap net als andere keren aanpassen, maar het overschrijft je vorige commit.

Bijvoorbeeld, als je commit en je dan realiseert dat je vergeten bent de veranderingen in een bestand dat je wilde toevoegen in deze commit te stagen, dan kun je zo iets als dit doen:

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

Na deze drie commando's eindig je met één commit; de tweede commit vervangt de resultaten van de eerste.

Het is belangrijk te beseffen dat als je de laatste commit amendeert, je deze niet zo zeer repareert als wel in z'n geheel *vervangt* met een nieuwe, verbeterde commit die de oude commit uit de weg duwt en de nieuwe daarvoor in de plaats zet. Effectief is het alsof de vorige commit nooit heeft plaatsgevonden, en het zal niet in je repository geschiedenis te zien zijn.



De overduidelijke waarde van amenderende commits is om kleine verbeteringen aan te brengen aan je laatste commit, zonder je repository geschiedenis te vervuilen met commit berichten van het soort "Oeps, weer een file vergeten toe te voegen" of "Verdraaid, een typefout in de laatste commit gecorrigeerd".

## Een gestaged bestand unstagen

De volgende twee paragrafen laten zien hoe je de staging area en veranderingen in je werkdictionaries aanpakt. Het prettige hier is dat het commando dat je gebruikt om de status van die gebieden te bepalen, je er ook aan herinnert hoe je de veranderingen eraan weer ongedaan kunt maken. Bijvoorbeeld, stel dat je twee bestanden gewijzigd hebt en je wilt ze committen als twee aparte veranderingen, maar je typt per ongeluk `git add *` en staget ze allebei. Hoe kun je één van de twee nu unstagen? Het `git status` commando herinnert je eraan:

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

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

Direct onder de “Changes to be committed” tekst, staat dat je `git reset HEAD <file>...` moet gebruiken om te unstagen. Laten we dat advies volgen om het `CONTRIBUTING.md` bestand te unstagen:

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

    renamed: README.md -> README

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

    modified: CONTRIBUTING.md
```

Het commando is een beetje vreemd, maar het werkt. Het `CONTRIBUTING.md` bestand is gewijzigd maar weer unstaged.



Hoewel `git reset` een gevaarlijk commando *kan* zijn, zeker als je deze met de `--hard` vlag aanroeft. Echter, in het bovenstaande scenario wordt het bestand in je werkdirectory niet geraakt, dus het is relatief ongevaarlijk.

Voor nu is deze toverspreuk alles wat je hoeft te weten van het `git reset` commando. We zullen nog veel meer details behandelen over wat `reset` doet en hoe dit onder te knie te krijgen zodat je werkelijke heel interessante dingen kunt doen in [Reset ontrafeld](#).

## Een gewijzigd bestand weer ongewijzigd maken

Wat als je bedenkt dat je de wijzigingen aan het `CONTRIBUTING.md` bestand niet wilt behouden? Hoe kun je dit makkelijk ongedaan maken; terugbrengen in de staat waarin het was toen je voor het laatst gecommit hebt (of initieel gekloond, of hoe je het ook in je werkdirectory gekregen hebt)? Gelukkig vertelt `git status` je ook hoe je dat moet doen. In de laatste voorbeeld-uitvoer, ziet het unstaged gebied er zo uit:

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

Het vertelt je behoorlijk expliciet hoe je je veranderingen moet weggooien. Laten we eens doen wat er staat:

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

Je kunt zien dat de veranderingen teruggedraaid zijn.



Het is belangrijk om te beseffen dat `git checkout -- [bestand]` een gevaarlijk commando is. Alle veranderingen die je aan dat bestand gedaan hebt zijn weg; je hebt er zojuist een ander bestand overheen gezet. Gebruik dit commando dan ook nooit, tenzij je heel zeker weet dat je het bestand niet wilt.

Als je het voor nu alleen maar even uit de weg wilt hebben, gebruik dan branching of stashing wat we behandelen in [Branchen in Git](#); dit zijn vaak de betere opties.

Onthoud, alles dat in Git *gecommit* is kan bijna altijd weer hersteld worden. Zelfs commits die op reeds verwijderde branches gedaan zijn, of commits die zijn overschreven door een `--amend` commit, kunnen weer hersteld worden (zie [Gegevensherstel](#) voor data herstel). Maar, alles wat je verliest dat nog nooit was gecommitted is waarschijnlijk voorgoed verloren.

## Werken met remotes

Om samen te kunnen werken op eender welke Git project, moet je weten hoe je jouw remote repositories moet beheren. Remote repositories zijn versies van je project, die worden gehost op het Internet of ergens op een netwerk. Je kunt er meerdere hebben, waarvan over het algemeen ieder ofwel alleen leesbaar, of lees- en schrijfbaar is voor jou. Samenwerken met anderen houdt in dat je deze remote repositories kunt beheren en data kunt pushen en pullen op het moment dat je werk moet delen. Remote repositories beheren houdt ook in weten hoe je ze moet toevoegen, ongeldige repositories moet verwijderen, meerdere remote branches moet beheren en ze als getracked of niet kunt definiëren, en meer. In dit gedeelte zullen we deze remote-beheer vaardigheden behandelen.

*Remote repositories kunnen op je lokale machine staan.*



Het is goed mogelijk dat je met een “remote” repository werkt die, in alle werkelijkheid, op dezelfde host staat als waar je op werkt. Het woord “remote” impliceert niet per se dat de repository ergens op het netwerk of het internet staat, alleen dat het elders is. Het werken op zo een remote repository houdt nog steeds alle standaard push, pull en fetch handelingen in als met elke andere remote.

## Laat je remotes zien

Om te zien welke remote servers je geconfigureerd hebt, kun je het `git remote` commando uitvoeren. Het laat de verkorte namen van iedere remote alias zien die je gespecificeerd hebt. Als je de repository gekloond hebt, dan zul je op z’n minst de oorsprong (origin) zien; dat is de standaard naam die Git aan de server geeft waarvan je gekloond hebt:

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

Je kunt ook `-v` specificeren, wat je de URL laat zien die Git bij de verkorte naam heeft opgeslagen om gebruikt te worden wanneer er van die remote moet worden gelezen of geschreven:

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

Als je meer dan één remote hebt, dan laat het commando ze allemaal zien. Bijvoorbeeld, een repository met meerdere remotes om met meerdere medewerkers samen te werken zou er ongeveer zo uit kunnen zien:

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

Dit betekent dat we vrij gemakkelijk de bijdragen van ieder van deze gebruikers naar binnen kunnen pullen. We zouden ook toestemming kunnen hebben om naar een of meerdere van deze te kunnen pushen, maar dat kunnen we hier niet zien.

Merk ook op dat deze remotes een veelheid aan protocollen gebruiken, we zullen hierover meer behandelen in [Git op een server krijgen](#).

## Remote repositories toevoegen

We hebben het feit dat het `git clone` commando impliciet de `origin` remote voor je toevoegt benoemd en gedemonstreerd. Hier laat ik zien hoe dat explicet gedaan wordt. Om een nieuw Git remote repository als een makkelijk te refereren alias toe te voegen, voer je `git remote add <verkorte naam> <url>` uit:

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

Nu kun je de naam `pb` op de commandoregel gebruiken in plaats van de hele URL. Bijvoorbeeld, als je alle informatie die Paul wel, maar jij niet in je repository hebt wilt fetchen, dan kun je `git fetch pb` uitvoeren:

```
$ git fetch pb
remote: Counting objects: 43, done.
remote: Compressing objects: 100% (36/36), done.
remote: Total 43 (delta 10), reused 31 (delta 5)
Unpacking objects: 100% (43/43), done.
From https://github.com/paulboone/ticgit
 * [new branch]      master      -> pb/master
 * [new branch]      ticgit     -> pb/ticgit
```

De master branch van Paul is lokaal toegankelijk als `pb/master`; je kunt het in een van jouw branches mergen, of je kunt een lokale branch uitchecken op dat punt als je het wil inzien. (We zullen in meer detail zien wat branches precies zijn, en hoe je ze moet gebruiken in [Branchen in Git](#).)

## Van je remotes fetchen en pullen

Zoals je zojuist gezien hebt, kun je om data van je remote projecten te halen dit uitvoeren:

```
$ git fetch <remote-name>
```

Het commando gaat naar het remote project en haalt alle data van dat remote project op dat jij nog niet hebt. Nadat je dit gedaan hebt, zou je references (referenties) naar alle branches van dat remote project moeten hebben, die je op ieder tijdstip kunt mergen en bekijken.

Als je een repository kloont, voegt dat commando die remote repository automatisch toe onder de naam “origin”. Dus `git fetch origin` fetcht (haalt) al het nieuwe werk dat gepusht is naar die server sinds je gekloond hebt (of voor het laatst gefetcht hebt). Het is belangrijk om te weten dat het fetch commando de data naar je locale repository haalt; het merget niet automatisch met je werk of verandert waar je momenteel aan zit te werken. Je moet het handmatig met jouw werk mergen wanneer je er klaar voor bent.

Als je een branch geconfigureerd hebt om een remote branch te volgen (tracken) (zie de volgende paragraaf en [Branchen in Git](#) voor meer informatie), dan kun je het `git pull` commando gebruiken om automatisch een remote branch te fetchen en mergen in je huidige branch. Dit kan makkelijker of een meer comfortabele workflow zijn voor je; en standaard stelt het `git clone` commando je lokale master branch zo in dat het de remote master branch van de server waarvan je gekloond hebt volgt (of hoe de standaard branch ook heet). Over het algemeen zal een `git pull` data van de server waarvan je origineel gekloond hebt halen en proberen het automatisch in de code waar je op dat moment aan zit te werken te mergen.

## Naar je remotes pushen

Wanneer je binnen jouw project op een punt zit waarop je het wilt delen, dan moet je het stroomopwaarts pushen. Het commando hiervoor is simpel: `git push <remote-name> <branch-name>`. Als je de master branch naar je `origin` server wilt pushen (nogmaals, over het algemeen zet klonen beide namen automatisch goed voor je), dan kun je dit uitvoeren om je werk terug op de server te pushen:

```
$ git push origin master
```

Dit commando werkt alleen als je gekloond hebt van een server waarop je schrijfrechten hebt, en als niemand in de tussentijd gepusht heeft. Als jij en iemand anders op hetzelfde tijdstip gekloond hebben en zij pushen eerder stroomopwaarts dan jij, dan zal je push terecht geweigerd worden. Je zult eerst hun werk moeten pullen en in jouw werk verwerken voordat je toegestaan wordt te pushen. Zie [Branchen in Git](#) voor meer gedetailleerde informatie over hoe je naar remote servers moet pushen.

## Een remote inspecteren

Als je meer informatie over een bepaalde remote wilt zien, kun je het `git remote show <remote>` commando gebruiken. Als je dit commando met een bepaalde alias uitvoert, zoals `origin`, dan krijg je zo iets als dit:

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

Het toont de URL voor de remote repository zowel als de tracking branch informatie. Het commando vertelt je behulpzaam dat als je op de master branch zit en je voert `git pull` uit, dat Git dan automatisch de master branch van de remote zal mergen nadat het alle remote references opgehaald heeft. Het toont ook alle remote referenties die het gepulld heeft.

Dat is een eenvoudig voorbeeld dat je vaak zult tegenkomen. Als je Git echter intensiever gebruikt, zul je veel meer informatie van `git remote show` krijgen:

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

Dit commando toont welke branch automatisch naar gepusht wordt als je `git push` uitvoert op als je op bepaalde branches staat. Het toont je ook welke remote branches op de server je nog niet hebt, welke remote branches je hebt die verwijderd zijn van de server, en meerdere lokale branches die automatisch gemerged worden met hun remote-tracking branch als je `git pull` uitvoert.

## Remotes verwijderen en hernoemen

Je kunt `git remote rename` uitvoeren om de korte naam van een remote te wijzigen. Bijvoorbeeld, als je `pb` wilt hernoemen naar `paul`, dan kun je dat doen met `git remote rename`:

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

Het is de moeite waard om te melden dat dit al je remote-tracking branch namen ook verandert. Waar voorheen aan gerefereerd werd als `pb/master` is nu `paul/master`.

Als je om een of andere reden een referentie wilt verwijderen, je hebt de server verplaatst of je gebruikt een bepaalde mirror niet meer, of een medewerker werkt niet meer mee, dan kun je `git remote rm` gebruiken:

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

Als je eenmaal de referentie naar een remote op deze manier verwijderd, worden alle remote-tracking branches en configuratie instellingen die met deze remote te maken hebben ook verwijderd.

## Taggen (Labelen)

Zoals de meeste VCS'en, heeft Git de mogelijkheid om specifieke punten in de historie als belangrijk te taggen (labelen). Over het algemeen gebruiken mensen deze functionaliteit om versie-punten te markeren (v1.0, enz.). In deze paragraaf zul je leren hoe de beschikbare tags te tonen, hoe nieuwe tags te creëren, en wat de verschillende typen tags zijn.

### Jouw tags laten zien

De beschikbare tags in Git laten zien is heel eenvoudig. Type gewoon `git tag` (met optioneel `-l` of `--list`):

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

Dit commando toont de tags in alfabetische volgorde; de volgorde waarin ze verschijnen heeft geen echte betekenis.

Je kunt ook zoeken op tags met een bepaald patroon. De Git bron-repository, bijvoorbeeld, bevat meer dan 500 tags. Als je alleen geïnteresseerd bent om naar de 1.8.5 serie te kijken, kun je dit uitvoeren:

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

*Tag wildcards uitlijsten vereist het gebruik van de `-l` of `--list` optie*



Als je alleen de hele lijst van tags wilt zien, gaat het commando `git tag` er impliciet van uit dat je een uitlijsting wilt en geeft er een; het gebruik van `-l` of `--list` is in dat geval optioneel. Echter, als je een wildcard patroon meegeeft om tag-namen te filteren, is het gebruik van `-l` of `--list` verplicht.

## Tags creëren

Git gebruikt twee tags: *lightweight* (lichtgewicht) en *annotated* (beschreven).

Een lightweight tag vertoont veel overeenkomst met een branch die niet verandert: het is slechts een wijzer naar een specifieke commit.

Annotated tags daarentegen, zijn als volwaardige objecten in de Git database opgeslagen. Ze worden gechecksumd, bevatten de naam van de tagger, e-mail en datum, hebben een tag boodschap, en kunnen gesigneerd en geverifieerd worden met GNU Privacy Guard (GPG). Het wordt over het algemeen aangeraden om annotated tags te maken zodat je al deze informatie hebt; maar als je een tijdelijke tag wilt of om een of andere reden de andere informatie niet wilt houden, dan zijn er lightweight tags.

## Annotated tags

Een annotated tag in Git maken is eenvoudig. Het makkelijkste is om de `-a` optie te specificeren als je het `tag` commando uitvoert:

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

De `-m` specificeert een tag boodschap, die bij de tag opgeslagen wordt. Als je geen boodschap voor een beschreven tag opgeeft, dan opent Git je editor zodat je deze kunt typen.

Je kunt de tag data zien, samen met de commit die getagd was, door het `git show` commando te gebruiken:

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

my version 1.4

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

    changed the version number
```

Dat toont informatie over de tagger, de datum waarop de commit getagd is, en de beschrijvende boodschap alvorens de commit informatie te laten zien.

## Lichtgewicht tags

Een andere manier om commits te taggen zijn lichtgewicht (lightweight) tags. Eigenlijk is dit de checksum van de commit die in een bestand opgeslagen wordt, er wordt geen enkele andere informatie bewaard. Om een lightweight tag te maken, geef je geen van de de **-a**, **-s** of **-m** opties mee:

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

Dit keer, als je `git show` op de tag runt, krijg je niet de extra tag informatie te zien. Het commando laat alleen de commit zien:

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

    changed the version number
```

## Later taggen

Je kunt ook commits taggen als je al veel verder bent. Stel dat je commit historie er als volgt uit ziet:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbe added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fce02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

En stel nu dat je bent vergeten het project op v1.2 te taggen, wat bij de commit van “updated rakefile” was. Je kunt dat achteraf toevoegen. Om die commit te taggen, moet je de commit checksum (of een deel daarvan) toevoegen aan het eind van het commando:

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

Je kunt zien dat je commit getagd hebt:

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

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

version 1.2
commit 9fce02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date:   Sun Apr 27 20:43:35 2008 -0700

        updated rakefile
...
```

## Tags delen

Standaard zal het `git push` commando geen tags naar remote servers versturen. Je zult expliciet tags naar een gedeelde server moeten pushen, nadat je ze gemaakt hebt. Dit proces is hetzelfde als remote branches delen - je kunt `git push origin <tagnaam>` uitvoeren.

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

Als je veel tags hebt die je ineens wilt pushen, kun je ook de `--tags` optie aan het `git push` commando toevoegen. Dit zal al je tags, die nog niet op de remote server zijn, in één keer er naartoe sturen.

```
$ git push origin --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 160 bytes | 0 bytes/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new tag]           v1.4 -> v1.4
 * [new tag]           v1.4-lw -> v1.4-lw
```

Als nu iemand anders van jouw repository kloont of pullt, dan zullen zij al jouw tags ook krijgen.

## Tags verwijderen

Om een tag uit je lokale repository te verwijderen, kan je `git tag -d <tagnaam>` gebruiken. Als voorbeeld, we kunnen onze lichtgewicht tag hierboven als volgt verwijderen:

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

Merk op dat dit niet de tag van enig remote server verwijdert. Om ook de remotes bij te werken, moet je `git push <remote> :refs/tags/<tagnaam>` gebruiken:

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

## Tags uitchecken

Als je de lijst van bestandsversies wilt zien waar een tag naar verwijst, kan je een `git checkout` doen, maar dit zet je repository wel in een “detached HEAD” status, wat een aantal nadelige bijeffecten heeft:

```
$ git checkout 2.0.0  
Note: checking out '2.0.0'.
```

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

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

```
git checkout -b <new-branch>
```

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

```
$ git checkout 2.0-beta-0.1
```

```
Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-final
```

```
HEAD is now at df3f601... add atlas.json and cover image
```

In "detached HEAD" status, als je wijzigingen maakt en dan een commit maakt, blijft de tag hetzelfde, maar je nieuwe commit zal niet tot enige branch behoren en zal onbereikbaar zijn, behalve bij de exacte hash van de commit. Dus als je wijzigingen moet maken - stel dat je een bug op een oudere versie oplost - zal je over het algemeen een branch willen maken:

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

Als je dit doet en dan een commit maakt, zal je **version2**-branch een beetje anders zijn dan je **v2.0.0** tag omdat het voortgaat met jouw nieuwe wijzigingen, dus wees voorzichtig.

## Git aliassen

Voordat we dit hoofdstuk over de basis van Git afsluiten, is er nog een kleine tip dat jouw Git beleving simpeler, eenvoudiger en meer eigen kan maken: aliassen. We zullen hier niet meer aan refereren of later in het boek aannemen dat je ze gebruikt hebt, maar we vinden dat we je moeten vertellen hoe ze werken.

Git zal niet automatisch commando's afleiden uit wat je gedeeltelijk intypt. Als je niet de hele tekst van elke Git commando wilt intypen, kun je gemakkelijk een alias voor elke commando configureren door **git config** te gebruiken. Hier zijn een aantal voorbeelden die je misschien wilt instellen:

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

Dit betekent dat je, bijvoorbeeld, in plaats van `git commit` je alleen `git ci` hoeft in te typen. Als je verder gaat in het gebruik van Git, zul je waarschijnlijk andere commando's ook vaker gaan gebruiken; in dat geval: schroom niet om nieuwe aliassen te maken.

Deze techniek kan ook makkelijk zijn om commando's te maken waarvan je vindt dat ze hadden moeten bestaan. Bijvoorbeeld, om het gebruikprobleem dat je ondervond met het `unstage` van een bestand, kan je je eigen `unstage` alias aan Git toevoegen:

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

Dit maakt de volgende twee commando's gelijkwaardig:

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

Het lijkt wat duidelijker te zijn. Het is ook gebruikelijk om een `last` commando toe te voegen:

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

Op deze manier kun je de laatste commit makkelijk zien:

```
$ git last  
commit 66938dae3329c7aebe598c2246a8e6af90d04646  
Author: Josh Goebel <dreamer3@example.com>  
Date: Tue Aug 26 19:48:51 2008 +0800  
  
    test for current head  
  
Signed-off-by: Scott Chacon <schacon@example.com>
```

Zoals je kunt zien, vervangt Git eenvoudigweg het nieuwe commando met hetgeen waarvoor je het gealiassed hebt. Maar, misschien wil je een extern commando uitvoeren, in plaats van een Git subcommando. In dat geval begin je het commando met een `!` karakter. Dit is handig als je je eigen applicaties maakt die met een Git repository werken. We kunnen dit demonstreren door `git visual` te aliassen met het uitvoeren van `gitk`:

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

## Samenvatting

Op dit punt kun je alle basis lokale Git operaties doen: een repository creëren of clonen, wijzigingen maken, de wijzigingen staggen en committen en de historie bekijken van alle veranderingen die de repository ondergaan heeft. Als volgende gaan we de beste eigenschap van Git bekijken: het branching model.

# Branchen in Git

Bijna elk VCS ondersteunt een bepaalde vorm van branchen. Branchen komt erop neer dat je een tak afsplitst van de hoofd-ontwikkellijn en daar verder mee gaan werken zonder aan de hoofdlijn te komen. Bij veel VCS'en is dat nogal een duur proces, vaak wordt er een nieuwe kopie gemaakt van de directory waar je broncode in staat, wat lang kan duren voor grote projecten.

Sommige mensen verwijzen naar het branch model in Git als de "killer eigenschap", en het onderscheidt Git zeker in de VCS-gemeenschap. Waarom is het zo bijzonder? De manier waarop Git brancht is ongelooflijk lichtgewicht, waardoor branch operaties vrijwel direct uitgevoerd zijn en het wisselen tussen de branches over het algemeen net zo snel. In tegenstelling tot vele andere VCS's, moedigt Git juist een workflow aan waarbij vaak gebrancht en gemerged wordt, zelfs meerdere keren per dag. Deze eigenschap begrijpen en de techniek beheersen geeft je een krachtig en uniek gereedschap en kan letterlijk de manier waarop je ontwikkelt veranderen.

## Branches in vogelvlucht

Om de manier waarop Git brancht echt te begrijpen, moeten we een stap terug doen en onderzoeken hoe Git zijn gegevens opslaat.

Zoals je je misschien herinnert van [Aan de slag](#), slaat Git zijn gegevens niet op als een reeks van wijzigingen of delta's, maar in plaats daarvan als een serie *snapshots*.

Als je in Git commit, dan slaat Git een commit object op dat een verwijzing bevat naar het snapshot van de inhoud die je gestaged hebt. Dit object bevat ook de naam en mailadres van de auteur, het merge bericht dat ingetypt was, verwijzingen naar de commit of commits die de directe ouders van deze commit waren: nul ouders voor de eerste commit, één ouder voor een normale commit, en meerdere ouders voor een commit die het resultaat is van een merge van twee of meer branches.

Om dit te visualiseren, gaan we aannemen dat je een directory hebt met drie bestanden, en je staget en commit ze allemaal. Door de bestanden te staggen krijgen ze allemaal een checksum (de SHA-1 hash waar we het in [Aan de slag](#) over hadden), worden die versies de bestanden in het Git repository (Git noemt ze *blobs*) opgeslagen, en worden die checksums aan de staging area toegevoegd:

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

Als je de commit aanmaakt door `git commit` uit te voeren zal Git iedere directory in het project (in dit geval alleen de root) van een checksum voorzien en slaat ze als boomstructuur (`tree`) objecten in de Git repository op. Daarna creëert Git een `commit` object dat de metagegevens bevat en een verwijzing naar de hoofd-`tree`-object van het project zodat Git deze snapshot opnieuw kan oproepen als dat nodig is.

Je Git repository bevat nu vijf objecten: een blob voor de inhoud van ieder van de drie bestanden, een tree die de inhoud van de directory weergeeft en specificeert welke bestandsnamen opgeslagen zijn als welke blob, en een commit met de verwijzing naar die hoofd-tree en alle commit-

metagegevens.

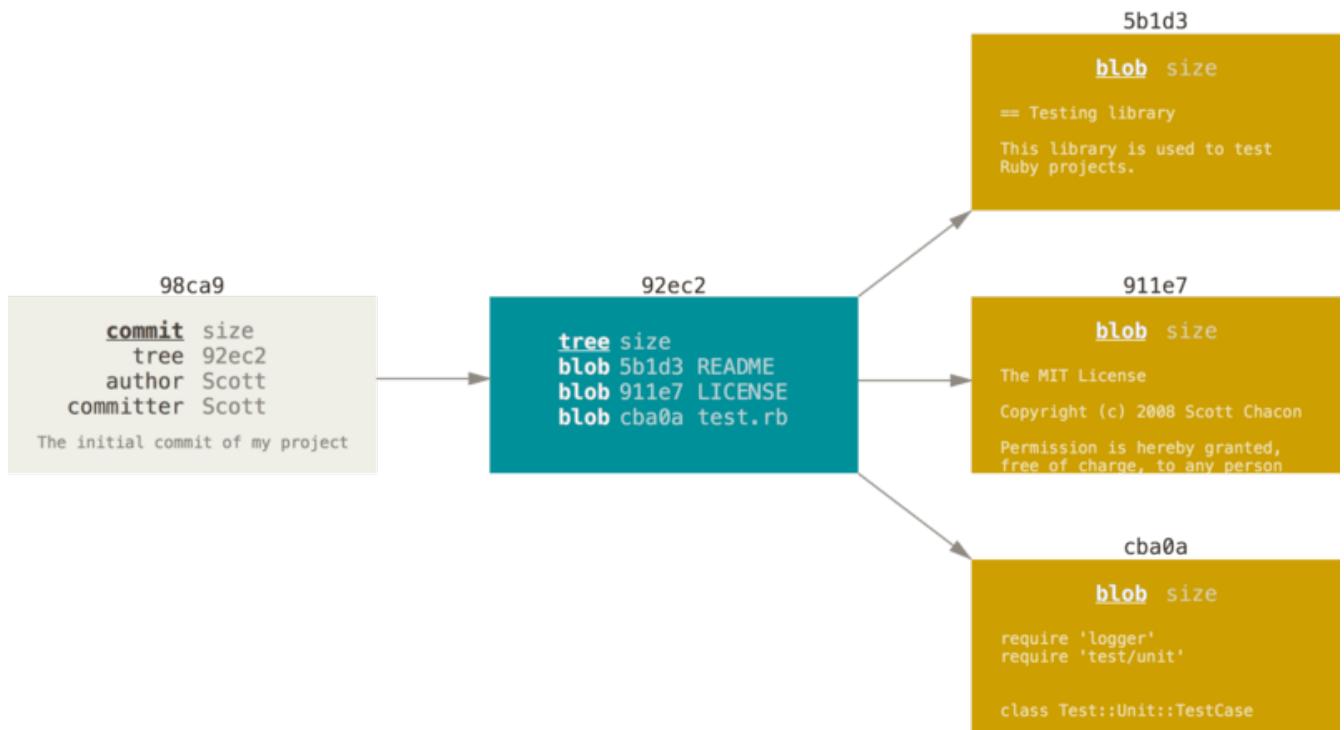


Figure 9. Een commit en zijn tree

Als je wat wijzigingen maakt en nogmaals commit, dan slaat de volgende commit een verwijzing op naar de commit die er direct aan vooraf ging.

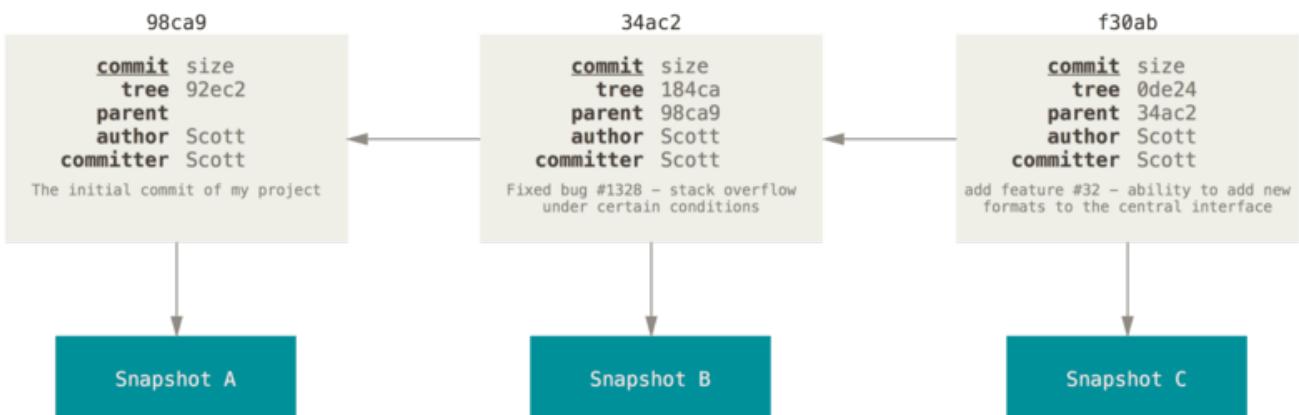


Figure 10. Commits en hun ouders

Een branch in Git is simpelweg een lichtgewicht verplaatsbare verwijzing naar een van deze commits. De standaard branch-naam in Git is `master`. Als je commits begint te maken, krijg je een `master`-branch die wijst naar de laatste commit die je gemaakt hebt. Iedere keer als je commit, beweegt de pointer van de `master`-branch automatisch vooruit.



De “master” branch in Git is geen speciale branch. Het is gelijk aan elke andere branch. De enige reden waarom vrijwel elke repository er een heeft is dat de `git init` commando er standaard een maakt en de meeste mensen geen moeite doen om deze te wijzigen.

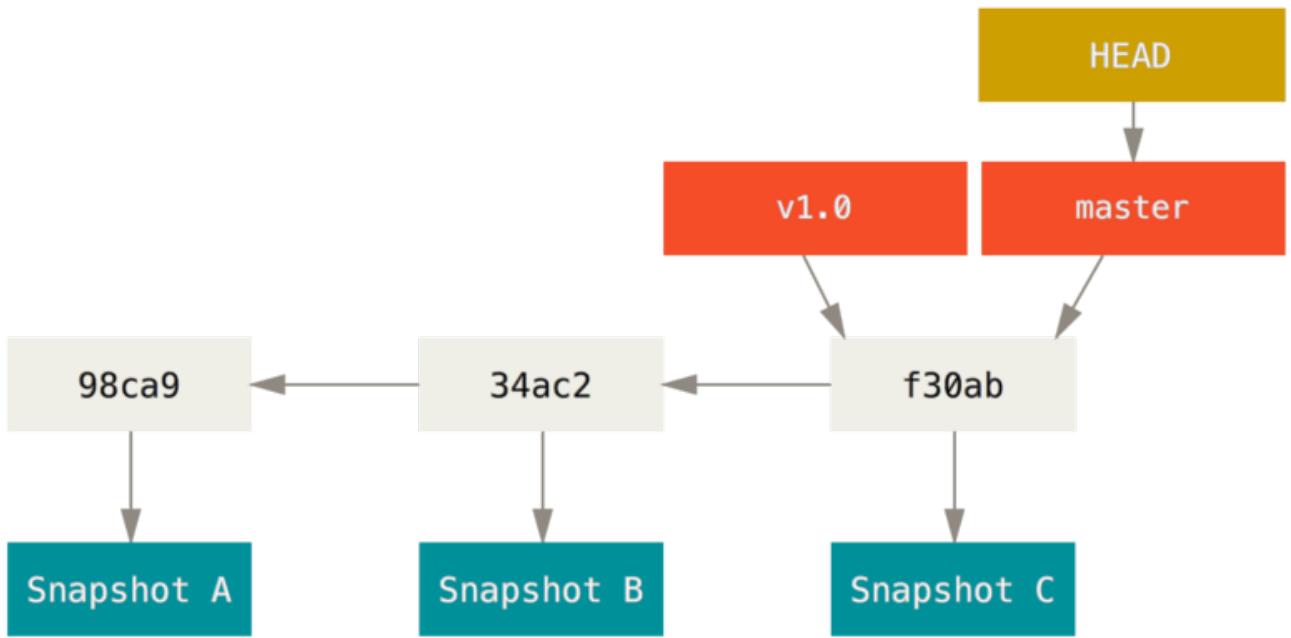


Figure 11. Een branch en zijn commit-historie

## Een nieuwe branch maken

Wat gebeurt er als je een nieuwe branch maakt? Nou, het aanmaken zorgt ervoor dat er een nieuwe verwijzing (pointer) voor je wordt gemaakt die je heen en weer kan bewegen. Stel dat je een nieuwe branch maakt en die `testing` noemt. Je doet dit met het `git branch` commando:

```
$ git branch testing
```

Dit maakt een nieuwe pointer op dezelfde commit als waar je nu op staat.

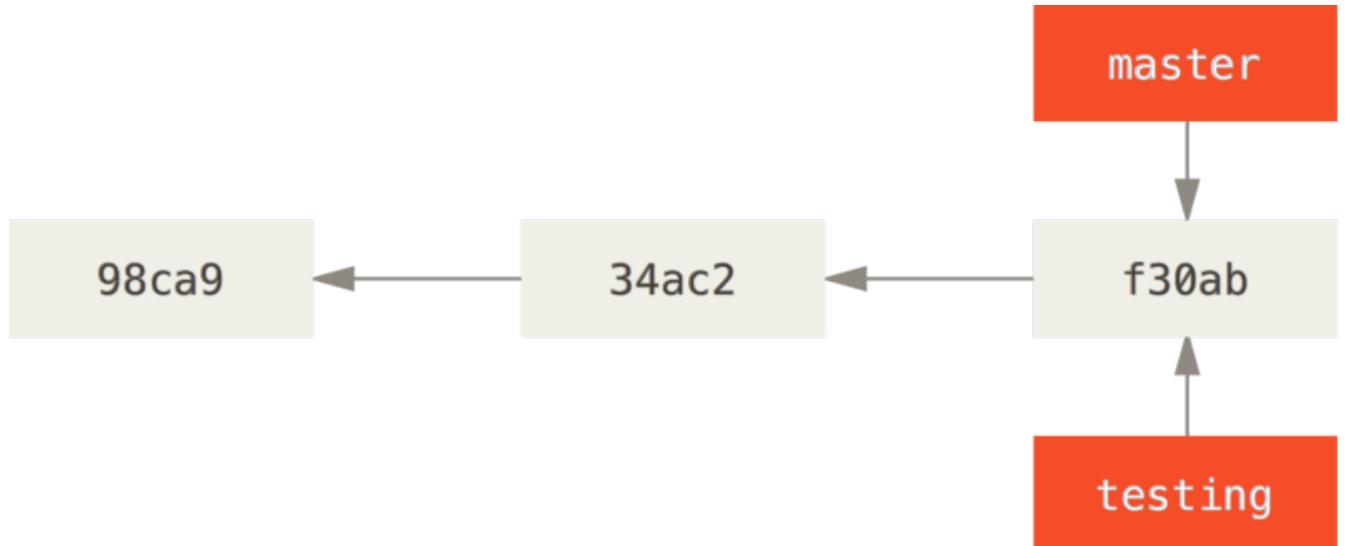


Figure 12. Twee branches die naar dezelfde reeks van commits verwijzen

Hoe weet Git op welke branch je nu zit? Het houdt een speciale verwijzing bij genaamd `HEAD`. Let op dat dit heel anders is dan het concept van `HEAD` in andere VCS's waar je misschien gewend aan bent, zoals Subversion of CVS. In Git is dit een verwijzing naar de lokale branch waar je nu op zit. In dit

geval zit je nog steeds op `master`. Het `git branch`-commando heeft alleen een nieuwe branch **aangemaakt** - we zijn nog niet overgeschakeld naar die branch.

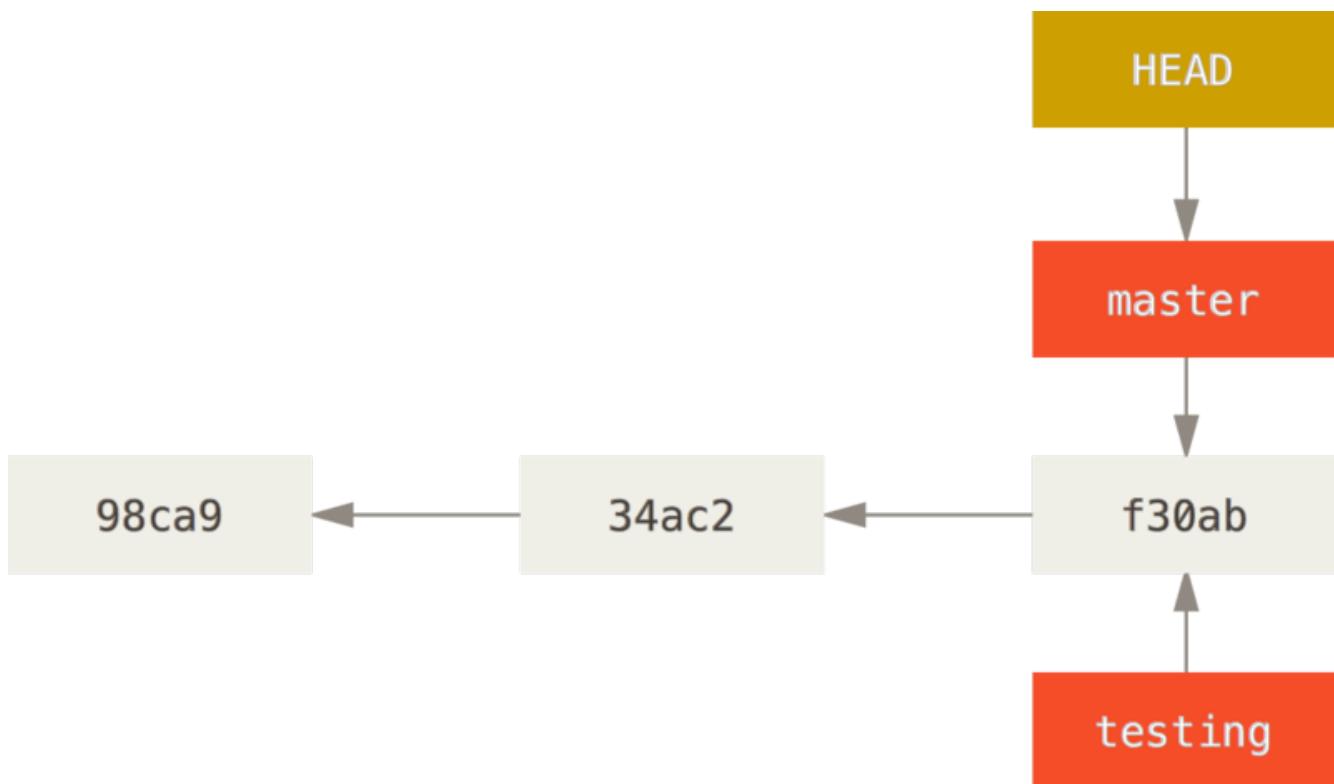


Figure 13. HEAD verwijzend naar een branch

Je kunt dit simpelweg zien door een eenvoudige `git log` commando uit te voeren wat je laat zien waar de branch pointers naar verwijzen. Deze optie heet **--decorate**.

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

Je kunt de "master" en "testing" branches zien die direct naast de `f30ab` commit staan.

## Tussen branches schakelen (switching)

Om te schakelen (switch) naar een bestaande branch, kan je het `git checkout` commando gebruiken. Laten we eens switchen naar de nieuwe `testing`-branch:

```
$ git checkout testing
```

Dit vertelt `HEAD` om te verwijzen naar de `testing`-branch.

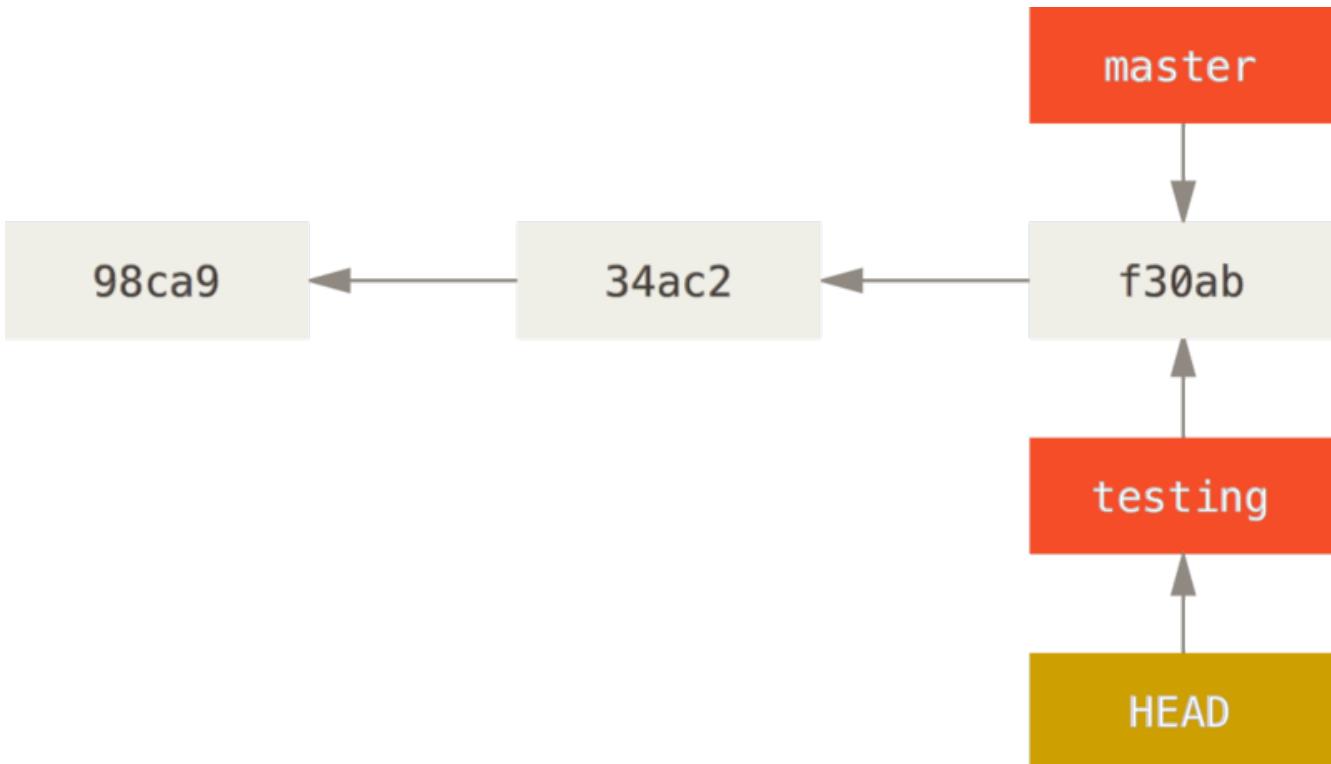


Figure 14. HEAD verwijst naar de huidige branch

Wat is daar nu belangrijk aan? Nou, laten we nog eens een commit doen:

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

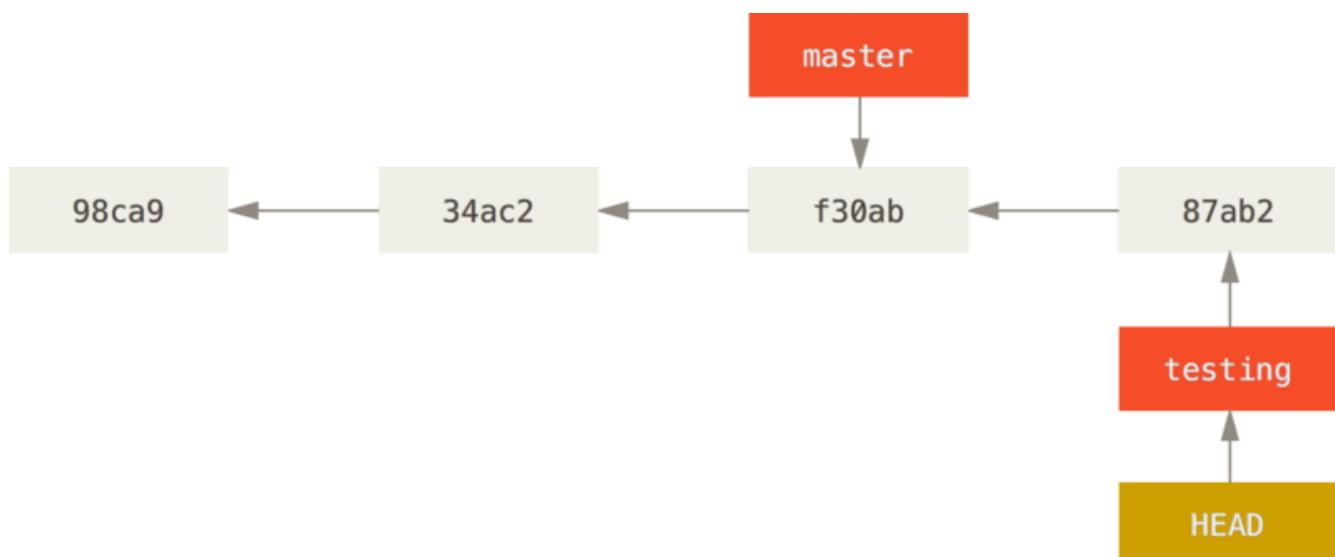


Figure 15. De HEAD branch beweegt voorwaarts als een commit wordt gedaan

Dit is interessant: omdat je **testing**-branch nu naar voren is bewogen, maar je **master** branch nog steeds op het punt staat waar je was toen je **git checkout** uitvoerde om van branch te switchen. Laten we eens terug switchen naar de **master**-branch:

```
$ git checkout master
```

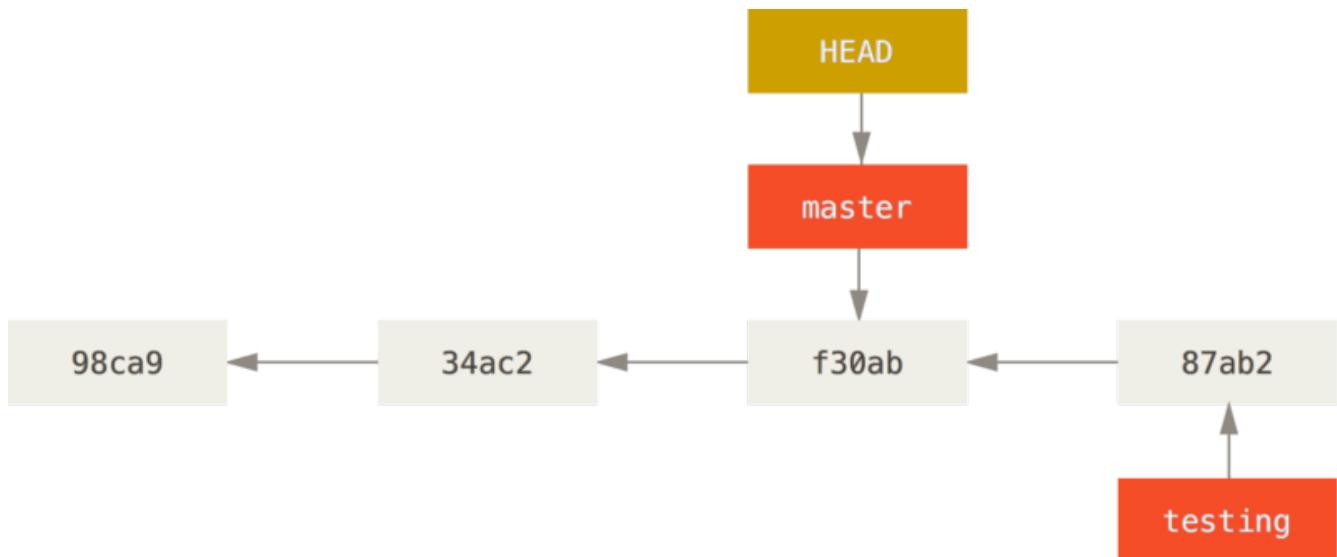


Figure 16. HEAD verplaatst als je checkout uitvoert

Dat commando deed twee dingen. Het verplaatste de HEAD pointer terug om te verwijzen naar de **master**-branch, en het draaide de bestanden terug in je werk directory naar de stand van de snapshot waar **master** naar verwijst. Dit betekent ook dat de wijzigingen die je vanaf nu maakt zullen afwijken van een oudere versie van het project. In essentie draait dit het werk terug dat je in je **testing**-branch gedaan hebt zodat je in een andere richting kunt bewegen.

#### *Branches switchen wijzigt bestanden in je directory*



Het is belangrijk op te merken dat wanneer je tussen branches switcht in Git, bestanden in je werk directory zullen wijzigen. Als je naar een oudere branch switcht, zal je werk directory teruggedraaid worden zodat de inhoud gelijk is aan hoe het eruit zag toen je voor het laatst committe op die branch. Als Git dat niet op een nette manier kan doen, zal het je niet laten switchen.

Laten we een paar wijzigingen aanbrengen en opnieuw committen:

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

Nu is je project historie uiteengelopen (zie [Uiteengelopen histories](#)). Je hebt een branch gemaakt en bent er naartoe overgeschakeld, hebt er wat werk op gedaan, en bent toen teruggeschakeld naar je hoofd-branch en hebt nog wat ander werk gedaan. Al die veranderingen zijn geïsoleerd van elkaar in aparte branches: je kunt heen en weer schakelen tussen de branches en ze mergen als je klaar bent. En je hebt dat alles gedaan met eenvoudige **branch**, **checkout** en **commit** commando's.

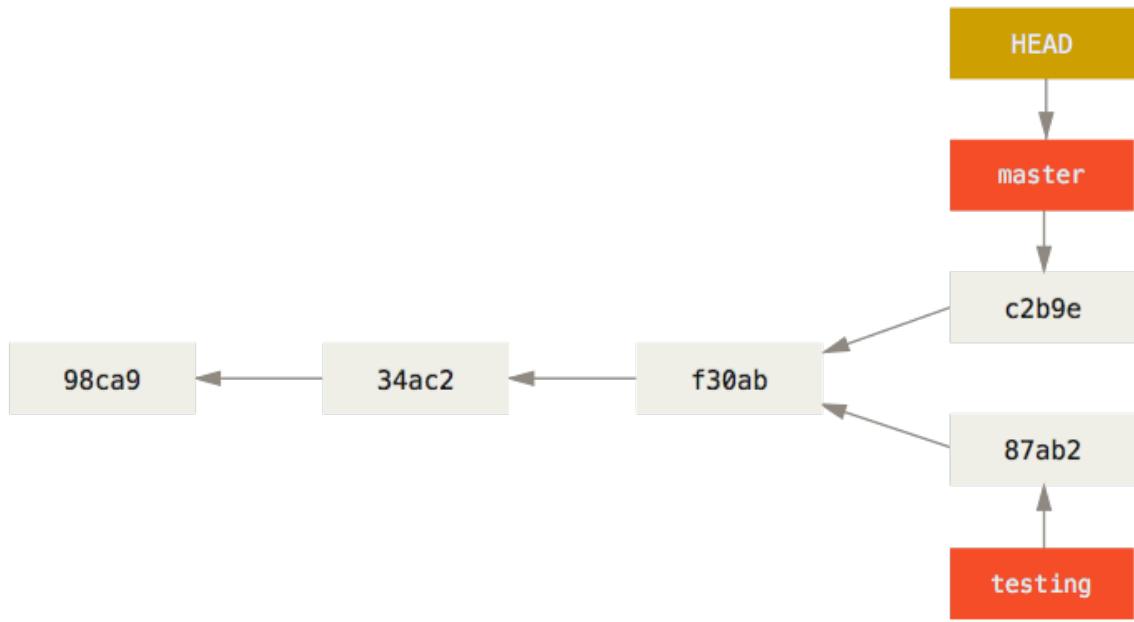


Figure 17. Uiteengelopen histories

Je kunt dit ook eenvoudig zien met het `git log` commando. Als je `git log --oneline --decorate --graph --all` uitvoert zal het de historie van jouw commits afdrukken, laten zien waar jouw branch pointers zijn en hoe je historie uiteengelopen is.

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

Omdat een branch in Git in werkelijkheid een eenvoudig bestand is dat de SHA-1 checksum van 40 karakters bevat van de commit waar het naar verwijst, zijn branches goedkoop om te maken en te verwijderen. Een nieuwe branch maken is net zo snel en simpel te maken als 41 bytes naar een bestand te schrijven (40 karakters en een newline).

Dit is in schril contrast met de manier waarop de meeste oudere VCS applicaties branchen, wat het kopiëren van alle bestanden van het project in een tweede directory inhoudt. Dit kan een aantal seconden of zelfs minuten duren, afhankelijk van de grootte van het project. Dit terwijl bij Git het proces altijd onmiddelijk gereed is. Daarbij komt dat, omdat we de ouders bijhouden als we een commit maken, het vinden van een goede merge basis voor het merge proces automatisch voor ons gedaan is en dit doorgaans eenvoudig te doen is. Deze kenmerken helpen ontwikkelaars aan te moedigen om vaak en veel branches aan te maken.

Laten we eens kijken waarom je dat zou moeten doen.

## Eenvoudig branchen en mergen

Laten we eens door een eenvoudig voorbeeld van branchen en mergen stappen met een workflow die je zou kunnen gebruiken in de echte wereld. Je zult deze stappen volgen:

1. Werken aan een website.
2. Een branch aanmaken voor een nieuw verhaal waar je aan werkt.
3. Wat werk doen in die branch.

Dan ga je een telefoontje krijgen dat je een ander probleem direct moet repareren; je moet een snelle reparatie (hotfix) maken. Je zult het volgende doen:

1. Switchen naar je productie-branch.
2. Een branch aanmaken om de hotfix toe te voegen.
3. Nadat het getest is de hotfix-branch mergen, en dat naar productie pushen.
4. Terugswitchen naar je originele verhaal en doorgaan met werken.

### Eenvoudig branchen

Als eerste, laten we zeggen dat je aan je project werkt en al een paar commits hebt gemaakt op de **master**-branch.

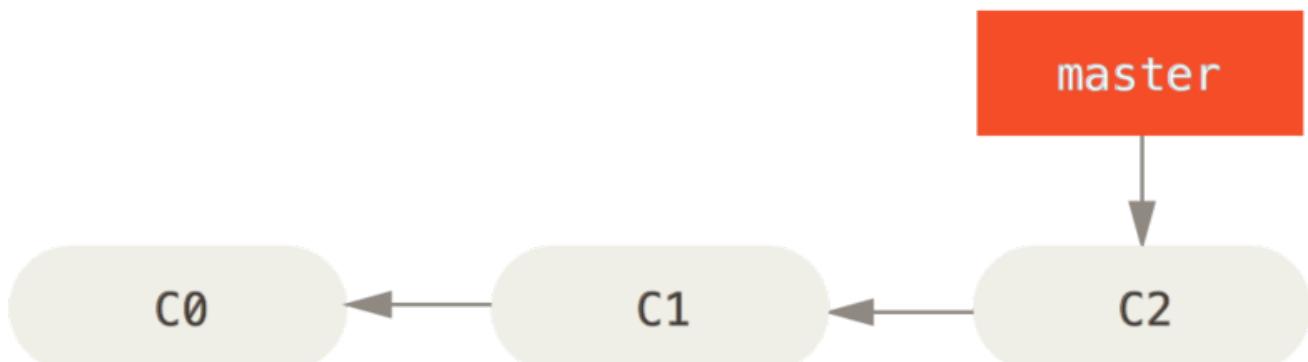


Figure 18. Een eenvoudige commit historie

Je hebt besloten dat je gaat werken aan probleem #53 in wat voor systeem je bedrijf ook gebruikt om problemen te registreren. Om een branch aan te maken en er meteen naartoe te schakelen, kun je het **git checkout** commando uitvoeren met de **-b** optie:

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

Dit is een afkorting voor:

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

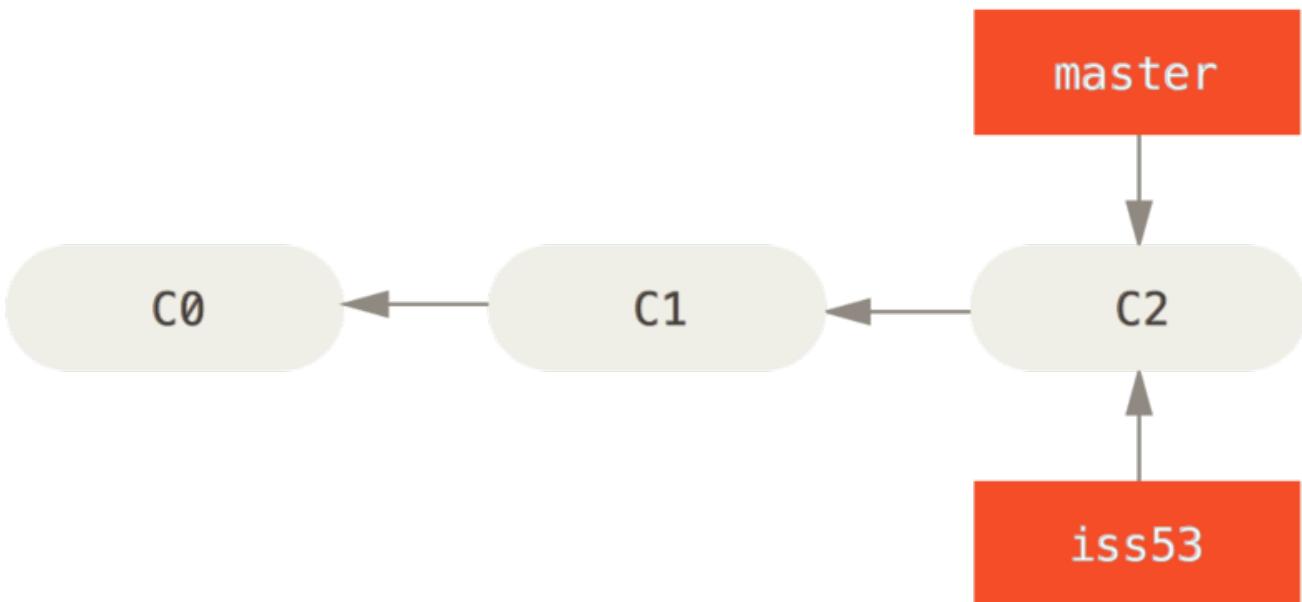


Figure 19. Een nieuwe branch pointer maken

Je doet wat werk aan je website en doet wat commits. Door dat te doen beweegt de `iss53`-branch vooruit, omdat je het uitgecheckt hebt (dat wil zeggen, je `HEAD` wijst ernaar):

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

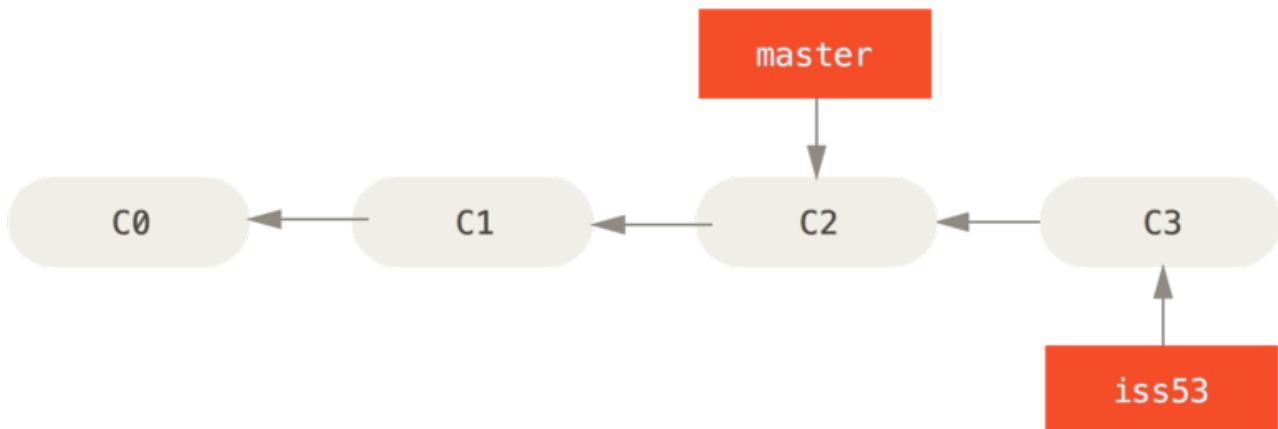


Figure 20. De `iss53`-branch is naar voren verplaatst met je werk

Nu krijg je het telefoonje dat er een probleem is met de website, en je moet het meteen repareren. Met Git hoef je de reparatie niet tegelijk uit te leveren met de `iss53` wijzigingen die je gemaakt hebt, en je hoeft ook niet veel moeite te doen om die wijzigingen terug te draaien voordat je kunt werken aan het toepassen van je reparatie in productie. Het enige wat je moet doen is terug schakelen naar je `master`-branch.

Maar voordat je dat doet, weet dat als je werk-directory of staging area wijzigingen bevatten die nog niet gecommit zijn en conflicteren met de branch die je gaat uitchecken, Git je niet laat omschakelen. Het beste is om een schone werkstatus te hebben als je tussen branches gaat schakelen. Er zijn manieren om hier mee om te gaan (te weten, stashen en commit ammending) die we later gaan behandelen in [Stashen en opschonen](#). Voor nu laten we aannemen dat je alle wijzigingen gecommit hebt, zodat je kunt switchen naar je **master**-branch:

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

Hierna is je project-werk-directory precies zoals het was voordat je begon te werken aan probleem #53, en je kunt je concentreren op je hotfix. Dit is een belangrijk punt om te onthouden: als je van branch overschakelt, herstelt Git je werk-directory zodanig dat deze eruit ziet als het toen je voor de laatste keer daarnaar hebt gecommit. Het voegt automatisch bestanden toe, verwijdert en wijzigt ze om er zeker van te zijn dat je werkkopie eruit ziet zoals de branch eruit zag toen je er voor het laatst op commiteerde.

Vervolgens heb je een hotfix te doen. Laten we een **hotfix**-branch maken om op te werken totdat het af is:

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

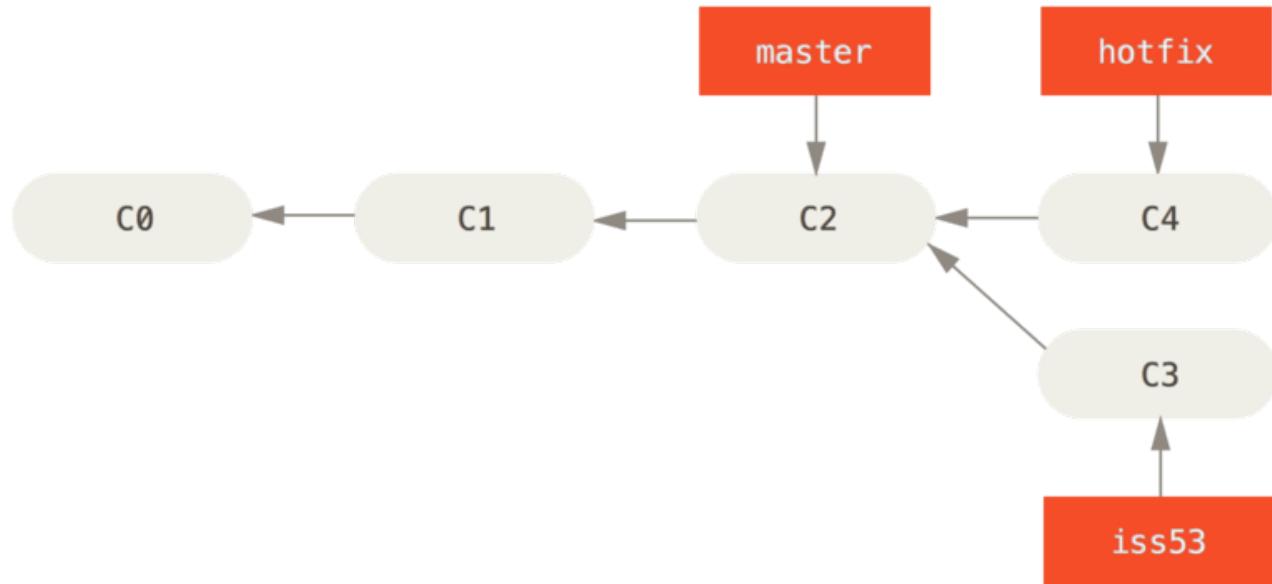


Figure 21. Hotfix branch gebaseerd op **master**

Je kunt je tests draaien, jezelf ervan verzekeren dat de hotfix is wat je wilt, en uiteindelijk de **hotfix**-branch mergen in je **master**-branch en het naar productie uitrollen. Je doet dit met het **git merge**

commando:

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

Je zult de uitdrukking “Fast forward” zien in die merge. Omdat de commit **C4** waar de branch **hotfix** die je in-mergede naar wees direct voor de commit **C2** ligt, waar je op staat, zet Git simpelweg de pointer naar voren. Om het op een andere manier te zeggen: als je een commit probeert te mergen met een commit die bereikt kan worden door de historie van eerste commit te volgen, zal Git de dingen vereenvoudigen door de verwijzing vooruit te verplaatsen omdat er geen afwijkend werk is om te mergen; dit wordt een “fast forward” genoemd.

Je wijziging zit nu in de snapshot van de commit waar de **master**-branch naar wijst, en je kunt je wijziging uitrollen.

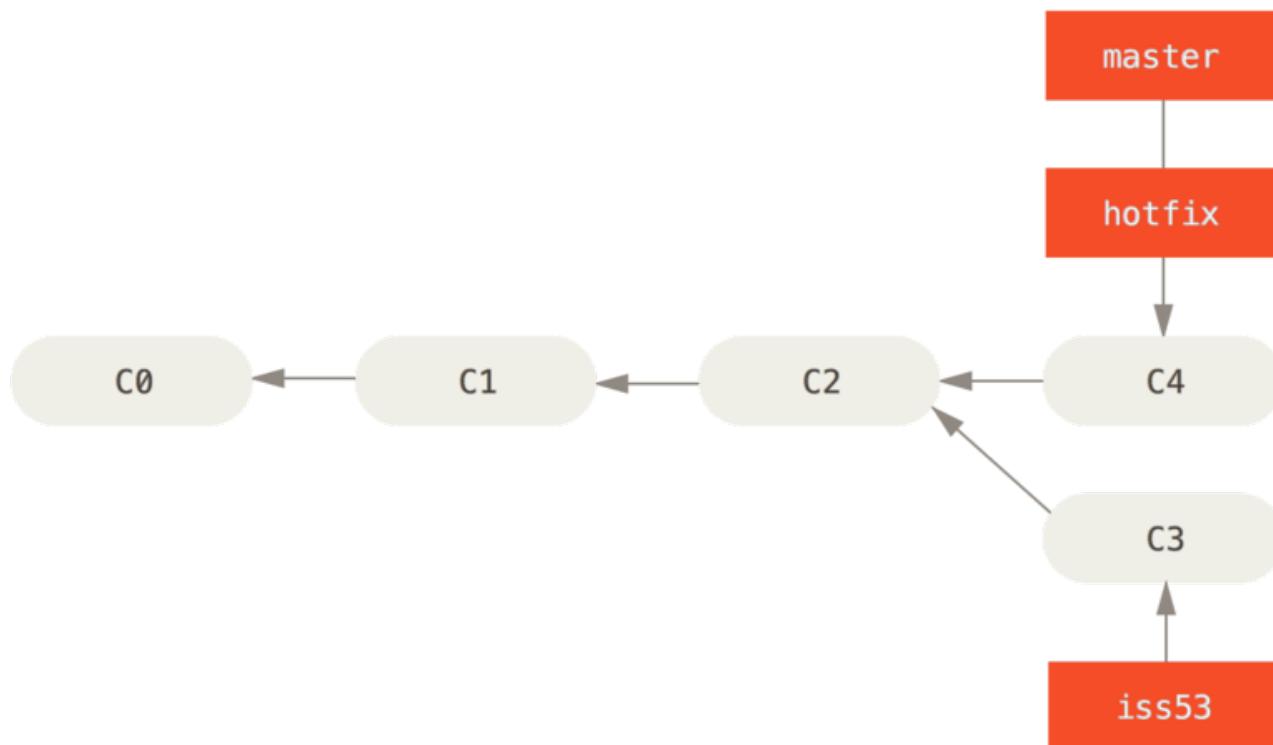


Figure 22. **master** is fast-forwarded naar **hotfix**

Nadat je super-belangrijke reparatie uitgerold is, ben je klaar om terug te schakelen naar het werk dat je deed voordat je onderbroken werd. Maar, eerst ga je de **hotfix**-branch verwijderen, omdat je die niet langer nodig hebt - de **master**-branch wijst naar dezelfde plek. Je kunt het verwijderen met de **-d** optie op **git branch**:

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

Nu kun je switchen naar je werk-in-uitvoering-branch voor probleem #53 en doorgaan met daaraan te werken.

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

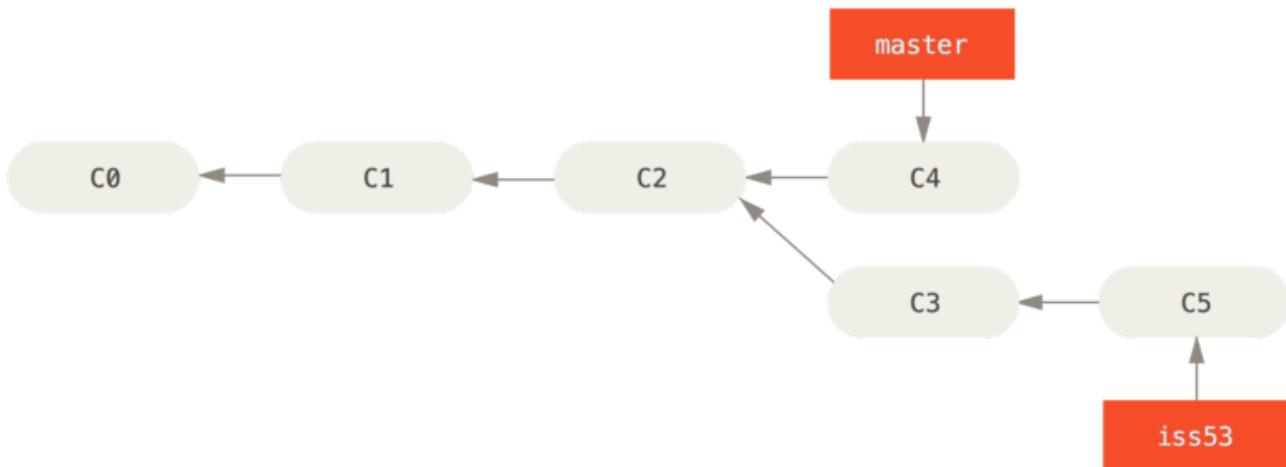


Figure 23. Werk gaat door op iss53

Het is nuttig om hier op te merken dat het werk dat je in de **hotfix**-branch gedaan hebt, niet in de bestanden van je **iss53**-branch zit. Als je dat binnen moet halen, dan kun je de **master**-branch in de **iss53**-branch mergen door `git merge master` uit te voeren, of je kunt wachten met die wijzigingen te integreren tot het moment dat je het besluit neemt de **iss53**-branch in de **master** te trekken.

## Eenvoudig mergen (samenvoegen)

Stel dat je besloten hebt dat je probleem #53 werk gereed is en klaar bent om het te mergen in de **master**-branch. Om dat te doen, zul je de **iss53**-branch mergen zoals je die **hotfix**-branch eerder hebt gemerged. Het enige dat je hoeft te doen is de branch uit te checken waar je in wenst te mergen en dan het `git merge` commando uit te voeren:

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

Dit ziet er iets anders uit dan de **hotfix** merge die je eerder gedaan hebt. In dit geval is je

ontwikkelhistorie afgeweken van een eerder punt. Omdat de commit op de branch waar je op zit geen directe voorouder is van de branch waar je in merget, moet Git wat werk doen. In dit geval doet Git een eenvoudige drieweg-merge, gebruikmakend van de twee snapshots waarnaar gewezen wordt door de uiteinden van de branch en de gezamenlijke voorouder van die twee.

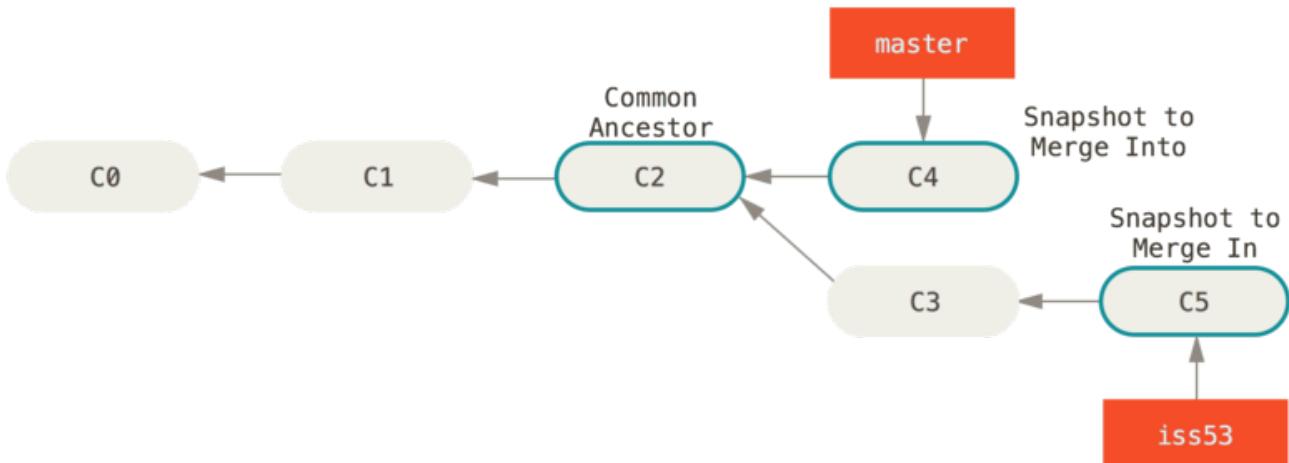


Figure 24. Drie snapshots gebruikt in een typische merge

In plaats van de branch-pointer alleen maar vooruit te verplaatsen, maakt Git een nieuw snapshot dat het resultaat is van deze drieweg-merge en maakt automatisch een nieuwe commit die daarnaar wijst. Dit wordt een merge-commit genoemd, en is bijzonder in die zin dat het meer dan één ouder heeft.

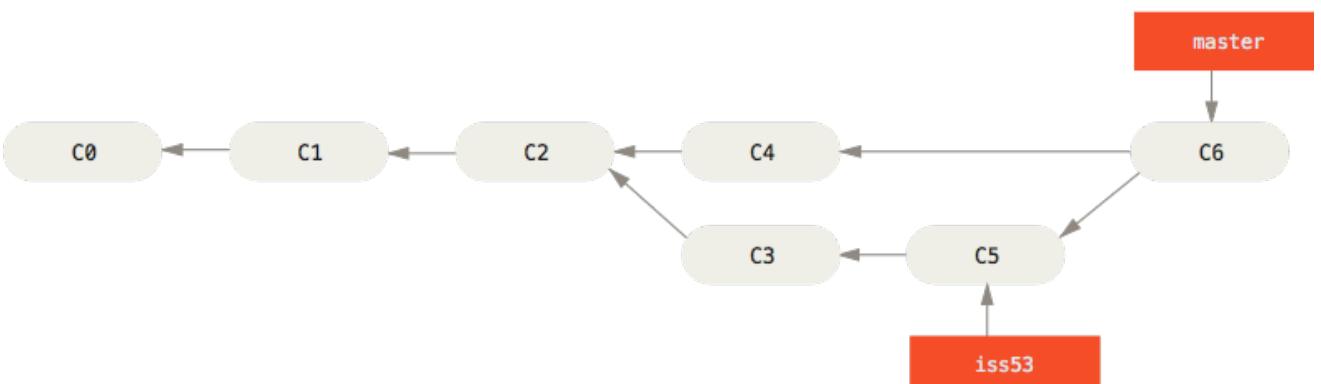


Figure 25. Een merge commit

Nu dat je werk gemerged is, is er geen noodzaak meer voor de `iss53`-branch. Je kunt deze verwijderen en daarna handmatig de ticket in het ticket-volgsysteem sluiten:

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

## Eenvoudige merge conflicten

Af en toe verloopt dit proces niet zo soepel. Als je hetzelfde gedeelte van hetzelfde bestand op een andere manier hebt gewijzigd in twee branches die je merget, dan zal Git niet in staat zijn om ze netjes te mergen. Als je reparatie voor probleem #53 hetzelfde gedeelte van een bestand heeft gewijzigd als de [hotfix](#), dan krijg je een merge conflict dat er ongeveer zo uit ziet:

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

Git heeft niet automatisch een nieuwe merge-commit gemaakt. Het heeft het proces gepauzeerd zolang jij het conflict aan het oplossen bent. Als je wilt zien welke bestanden nog niet zijn gemerged op enig punt na een merge conflict, dan kun je [git status](#) uitvoeren:

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

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

    both modified:      index.html

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

Alles wat merge-conflicten heeft en wat nog niet is opgelost wordt getoond als unmerged. Git voegt standaard conflict-oplossingsmarkeringen toe aan de bestanden die conflicten hebben, zodat je ze handmatig kunt openen en die conflicten kunt oplossen. Je bestand bevat een sectie die er zo ongeveer uit ziet:

```
<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=====
<div id="footer">
  please contact us at support@github.com
</div>
>>>>> iss53:index.html
```

Dit betekent dat de versie in **HEAD** (jouw **master**-branch, omdat dat degene was die je uitgecheckt had toen je het merge-commando uitvoerde) is het bovenste gedeelte van dat blok (alles boven de **=====**), terwijl de versie in je **iss53**-branch eruit ziet zoals alles in het onderste gedeelte. Om het conflict op te lossen, moet je één van de twee gedeeltes kiezen of de inhoud zelf mergen. Je zou bijvoorbeeld dit conflict op kunnen lossen door het hele blok met dit te vervangen:

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

Deze oplossing bevat een stukje uit beide secties, en de <<<<<, =====, en >>>>> regels zijn volledig verwijderd. Nadat je elk van deze secties opgelost hebt in elk conflicterend bestand, voer dan `git add` uit voor elk van die bestanden om het als opgelost te markeren. Het bestand staget markeert het als opgelost in Git.

Als je een grafische applicatie wil gebruiken om deze problemen op te lossen, kun je `git mergetool` uitvoeren, wat een toepasselijk grafische merge-applicatie opstart dat je door de conflicten heen leidt:

```
$ git mergetool  
  
This message is displayed because 'merge.tool' is not configured.  
See 'git mergetool --tool-help' or 'git help config' for more details.  
'git mergetool' will now attempt to use one of the following tools:  
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge  
p4merge araxis bc3 codecompare vimdiff emerge  
Merging:  
index.html  
  
Normal merge conflict for 'index.html':  
 {local}: modified file  
 {remote}: modified file  
Hit return to start merge resolution tool (opendiff):
```

Als je een andere dan de standaard merge-tool wilt gebruiken (Git koos `opendiff` in dit geval, omdat het commando uitgevoerd werd op een Mac), dan kun je alle ondersteunde applicaties opgesomd zien na “one of the following tools.” Type de naam van de applicatie die je liever gebruikt.



Als je een meer geavanceerde applicatie wilt gebruiken om ingewikkelde merge conflicten op te lossen: we behandelen merging uitgebreider in [Mergen voor gevorderden](#).

Nadat je de merge applicatie afsluit, vraagt Git je of de merge succesvol was. Als je het script vertelt dat dit het geval is, dan staget dit script het bestand voor je om het als opgelost te markeren. Je kunt `git status` nogmaals uitvoeren om je ervan te verzekeren dat alle conflicten opgelost zijn:

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

Changes to be committed:

```
modified: index.html
```

Als je het daar mee eens bent, en je gecontroleerd hebt dat alles waar conflicten in zaten gestaged is, dan kun je `git commit` typen om de merge-commit af te ronden. Het commit-bericht ziet er standaard ongeveer zo uit:

```
Merge branch 'iss53'

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

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

Je kunt dat bericht aanpassen met details over hoe je het conflict opgelost hebt, als je denkt dat dat behulpzaam zal zijn voor anderen die in de toekomst naar deze merge kijken - waarom je hebt gedaan wat je gedaan hebt, als dat niet vanzelfsprekend is.

## Branch-beheer

Nu heb je wat branches aangemaakt, gemerged, en verwijderd. Laten we eens kijken naar welke branch-beheer-toepassingen handig zijn als je vaker branches gaat gebruiken.

Het `git branch` commando doet meer dan alleen branches aanmaken en verwijderen. Als je het zonder argumenten uitvoert, dan krijg je een eenvoudige lijst van de huidige branches:

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

Merk op dat het **\*** karakter vooraf gaat aan de **master**-branch: het geeft de branch aan die je op dit moment uitgecheckt hebt (d.i. de branch waar **HEAD** nu naar verwijst). Dit betekent dat als je op dit punt commit, de **master**-branch vooruit zal gaan met je nieuwe werk. Om de laatste commit op iedere branch te zien, kun je **git branch -v** uitvoeren:

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

De handige **--merged** en **--no-merged** opties zijn beschikbaar om te zien welke branches al dan niet gemerged zijn in de branch waar je nu op zit. Om te zien welke branches al gemerged zijn in de branch waar je nu op zit, kan je **git branch --merged** draaien:

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

Omdat je **iss53** al eerder hebt gemerged, zie je het terug in je lijst. Branches op deze lijst zonder de **\*** ervoor zijn over het algemeen zonder problemen te verwijderen met **git branch -d**; je hebt hun werk al in een andere branch zitten, dus je zult niets kwijtraken.

Om alle branches te zien die werk bevatten die je nog niet gemerged hebt, kan je **git branch --no-merged** draaien:

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

Dit toont je andere branch. Omdat het werk bevat dat nog niet gemerged is, zal het verwijderen met **git branch -d** falen:

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

Als je de branch echt wilt verwijderen en dat werk wilt verliezen, dan kun je het forceren met **-D**, zoals het behulpzame bericht je al meldt.

De bovenstaande opties `--merged` en `--no-merged` zullen, als je niet een commit of branchnaam als argument meegeeft, laten zien wat er respectievelijk wel of niet gemerged is in je *huidige* branch.

Je kunt altijd een extra argument meegeven om te vragen over de merge status van een andere branch zonder deze eerst uit te hoeven checken, als in, wat is er nog niet in de `master`-branch gemerged?

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

## Branch workflows

Nu je de basis van het branchen en mergen onder de knie hebt, wat kan je of zou je daarmee kunnen doen? In deze deel gaan we een aantal veel voorkomende workflows die deze lichtgewicht branches mogelijk maken behandelen, zodat je kunt besluiten of je ze wilt toepassen in je eigen ontwikkelcyclus.

### Langlopende branches

branches, long-running) Omdat Git gebruik maakt van een eenvoudige drieweg-merge, is het meerdere keren mergen vanuit een branch in een andere gedurende een langere periode over het algemeen eenvoudig te doen. Dit houdt in dat je meerdere branches kunt hebben, die altijd open staan en die je voor verschillende fasen van je ontwikkelcyclus gebruikt; je kunt regelmatig vanuit een aantal mergen in andere.

Veel Git-ontwikkelaars hebben een workflow die deze aanpak omarmt, zoals het hebben van alleen volledig stabiele code in hun `master`-branch — mogelijk alleen code die is of zal worden vrijgegeven. Ze hebben een andere parallelle branch `develop` of `next` genaamd waarop ze werken of die ze gebruiken om stabiliteit te testen — het is niet noodzakelijkerwijs altijd stabiel, maar zodra het in een stabiele status verkeert, kan het worden gemerged in `master`. Deze wordt gebruikt om topic branches (branches met een korte levensduur, zoals jou eerdere `iss53`-branch) te pullen zodra die klaar zijn, om zich ervan te overtuigen dat alle tests slagen en er geen fouten worden geïntroduceerd.

Feitelijk praten we over verwijzingen die worden verplaatst over de lijn van de commits die je maakt. De stabiele branches zijn verder stroomafwaarts in je commit-historie, en de splinternieuwe branches zijn verder naar voren in de historie.

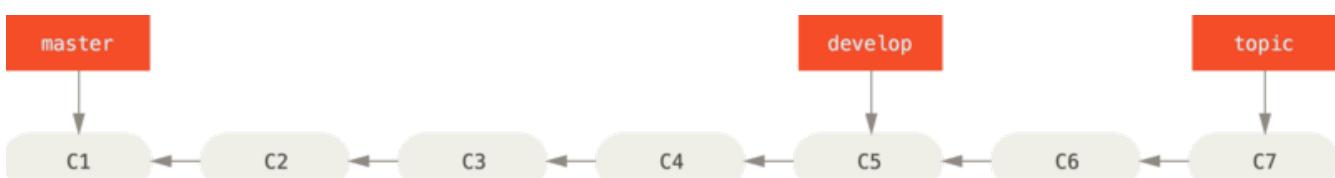


Figure 26. Een lineare kijk op progressief-stabiliteits branchen

Ze zijn misschien makkelijker voor te stellen als silo's, waar sets van commits stapsgewijs naar een meer stabiele silo worden gepromoveerd als ze volledig getest zijn.

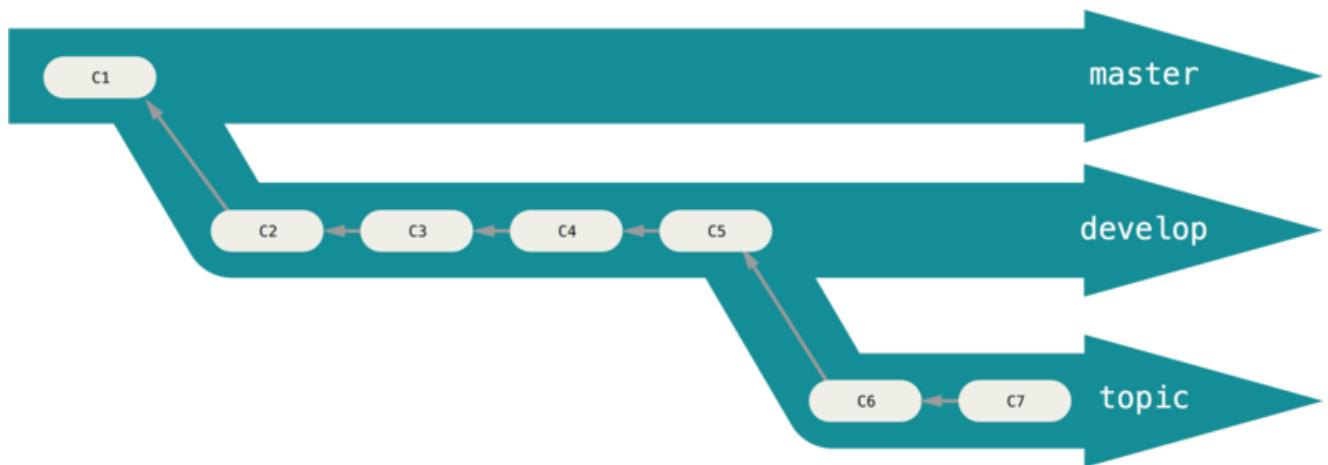


Figure 27. Een “silo” kijk op progressief-stabiliteits branchen

Je kunt dit blijven doen voor elk niveau van stabiliteit. Sommige grotere projecten hebben ook een **proposed** of **pu** (proposed updates) branch die branches geïntegreerd heeft die wellicht nog niet klaar zijn om in de **next** of **master**-branch te gaan. Het idee er achter is dat de branches op verschillende niveaus van stabiliteit zitten. Zodra ze een stabielere niveau bereiken, worden ze in de branch boven hen gemerged. Nogmaals, het hebben van meerdere langlopende branches is niet noodzakelijk, maar het helpt vaak wel; in het bijzonder als je te maken hebt met zeer grote of complexe projecten.

## Topic branches

Topic branches zijn nuttig in projecten van elke grootte. Een topic branch is een kortlopende branch die je maakt en gebruikt om een specifieke functie te realiseren of daaraan gerelateerd werk te doen. Dit is iets wat je waarschijnlijk nooit eerder met een VCS gedaan hebt, omdat het over het algemeen te duur is om branches aan te maken en te mergen. Maar in Git is het niet ongebruikelijk om meerdere keren per dag branches aan te maken, daarop te werken, en ze te verwijderen.

Je zag dit in de vorige paragraaf met de **iss53** en **hotfix**-branches die je gemaakt had. Je hebt een aantal commits op ze gedaan en ze meteen verwijderd nadat je ze gemerged had in je hoofdbranch. Deze techniek stelt je in staat om snel en volledig van context te veranderen— omdat je werk is onderverdeeld in silo's waar alle wijzigingen in die branch te maken hebben met dat onderwerp, is het makkelijker te zien wat er is gebeurd tijdens een code review en dergelijke. Je kunt de wijzigingen daar minuten-, dagen- of maandenlang bewaren, en ze mergen als ze er klaar voor zijn, ongeacht de volgorde waarin ze gemaakt zijn of er aan gewerkt is.

Neem als voorbeeld een situatie waarbij wat werk gedaan wordt (op **master**), er wordt een branche gemaakt voor een probleem (**iss91**) en daar wordt wat aan gewerkt, er wordt een tweede branch gemaakt om op een andere manier te proberen hetzelfde op te lossen (**iss91v2**); weer even wordt teruggegaan naar de master branch om daar een tijdje te werken, en dan vanaf daar wordt gebrancht om wat werk te doen waarvan je niet zeker weet of het wel zo'n slim idee is (**dumbidea**-branch). Je commit-historie zal eruitzien als volgt:

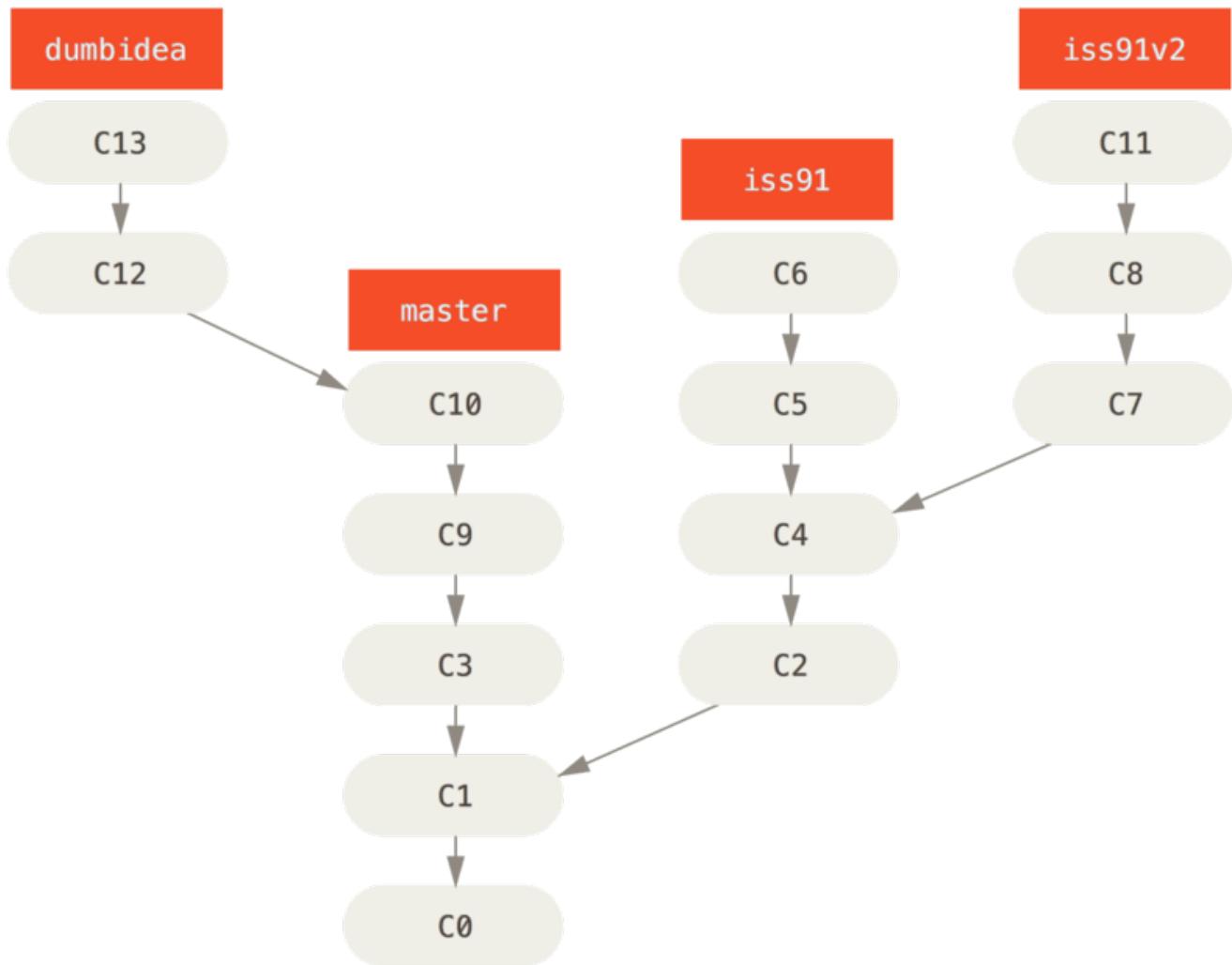


Figure 28. Meerdere topic branches

Laten we zeggen dat je besluit dat je de tweede oplossing voor je probleem het beste vindt (**iss91v2**), en je hebt de **dumbidea**-branch aan je collega's laten zien en het blijkt geniaal te zijn. Je kunt dan de oorspronkelijke **iss91** weggooien (waardoor je commits **C5** en **C6** kwijt raakt), en de andere twee mergen. Je historie ziet er dan uit volgt:

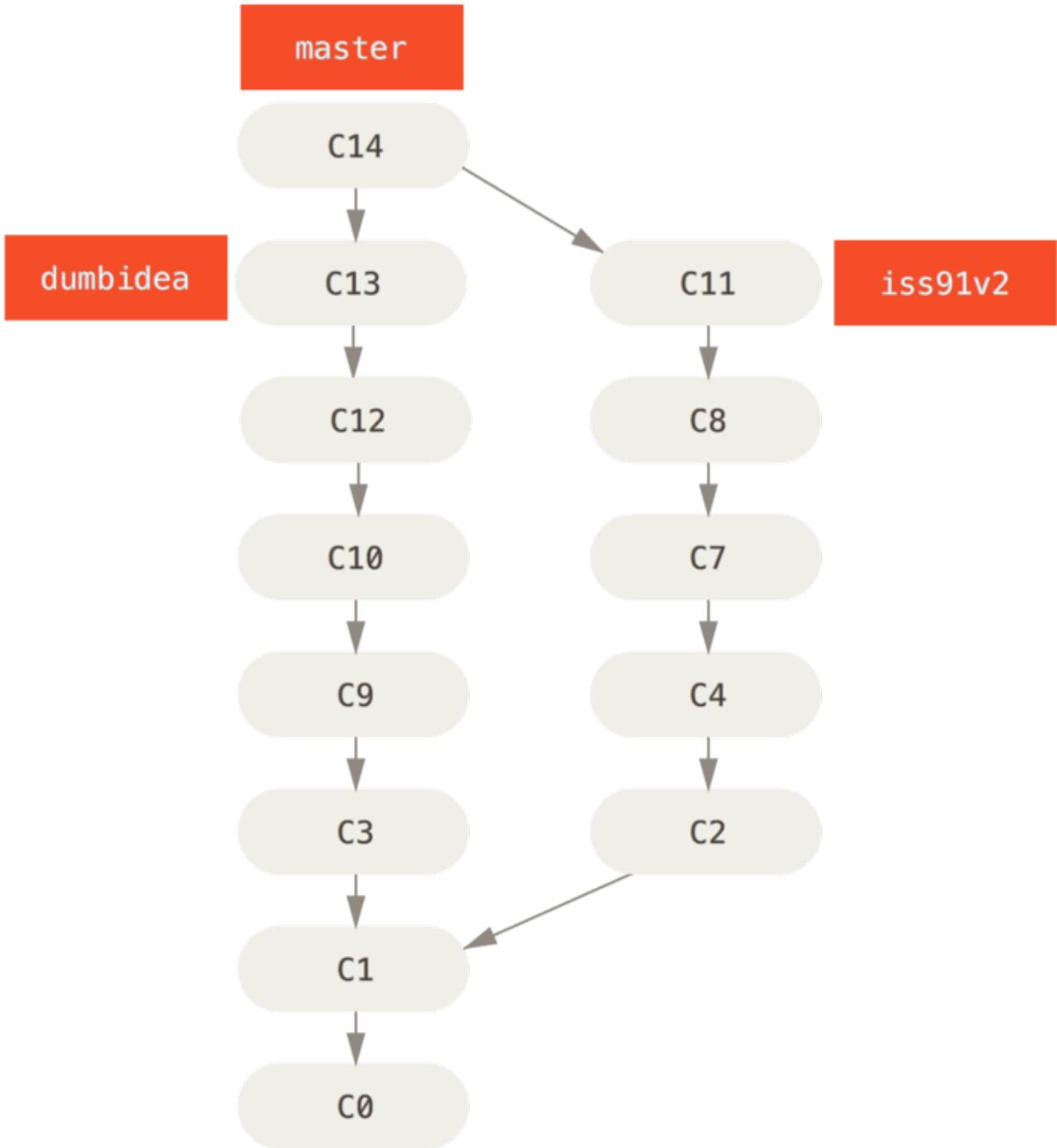


Figure 29. Historie na het mergen van `dumbidea` en `iss91v2`

We zullen in meer detail behandelen wat de verschillende mogelijke workflows zijn voor jouw Git project in [Gedistribueerd Git](#), dus voordat je besluit welk branching schema je voor jouw volgende project wilt gebruiken, zorg dat je dat hoofdstuk gelezen hebt.

Het is belangrijk om te beseffen dat tijdens al deze handelingen, al deze branches volledig lokaal zijn. Als je aan het branchen of mergen bent, dan wordt alles alleen in jouw Git repository gedaan — dus er vindt geen server communicatie plaats.

## Branches op afstand (Remote branches)

Remote branches zijn referenties (pointers) naar jouw remote repositories, inclusief branches, tags,

enz. Je kunt een volledige lijst van remote referenties expliciet krijgen met `git ls-remote [remote]` of ook `git remote show [remote]` voor zowel remote branches als voor meer informatie. Niettemin is het gebruiklijker om te profiteren van remote-tracking branches.

Remote-tracking branches zijn referenties naar de staat van remote branches. Het zijn lokale referenties die je niet kunt verplaatsen; Git verplaatst ze automatisch voor je op de momenten dat je een vorm van netwerk communicatie uitvoert, dit om te verzekeren dat ze een accurate representatie zijn van de staat van de remote repository. Remote-tracking branches gedragen zich als boekenleggers om je eraan te helpen herinneren wat de staat van de branches was van je remote repositories toen je voor het laatst met ze in contact was.

Remote-tracking branches hebben de vorm `<remote>/<branch>`. Bijvoorbeeld, als je wil zien hoe de `master`-branch op je `origin` remote er uitzag de laatste keer dat je er mee communiceerde, dan zal je de `origin/master`-branch moeten bekijken. Als je samen met een partner aan het werk bent met een probleem en zij heeft een `iss53`-branch gepusht, is het niet onmogelijk dat je zelf een eigen lokale `iss53` hebt, maar de branch op de server zal vertegenwoordigd zijn door de remote-tracking branch `origin/iss53`.

Dit kan wat verwarrend zijn, dus laten we eens naar een voorbeeld kijken. Stel dat je een Git-server in je netwerk hebt op `git.ourcompany.com`. Als je hiervan kloont dan wordt die door het `clone` commando van Git automatisch `origin` voor je genoemd, Git haalt alle gegevens binnen, maakt een verwijzing naar waar de `master`-branch is en noemt dat lokaal `origin/master`. Git geeft je ook een eigen lokale `master`-branch, beginnend op dezelfde plaats als de `master`-branch van `origin`, zodat je iets hebt om vanaf te werken.

*“origin” is niets speciaal*



Net als de branch naam “master” geen enkele speciale betekenis heeft in Git, heeft “origin” dat ook niet. Waar “master” de standaard naam is voor een branch die dient als beginpunt als je `git init` aanroeft, wat de enige reden is waarom het zo vaak wordt gebruikt, is “origin” de standaard naam voor een remote als je `git clone` aanroeft. Als je `git clone -o booyah` gebruikt, krijg je `booyah/master` als je standaard remote branch.

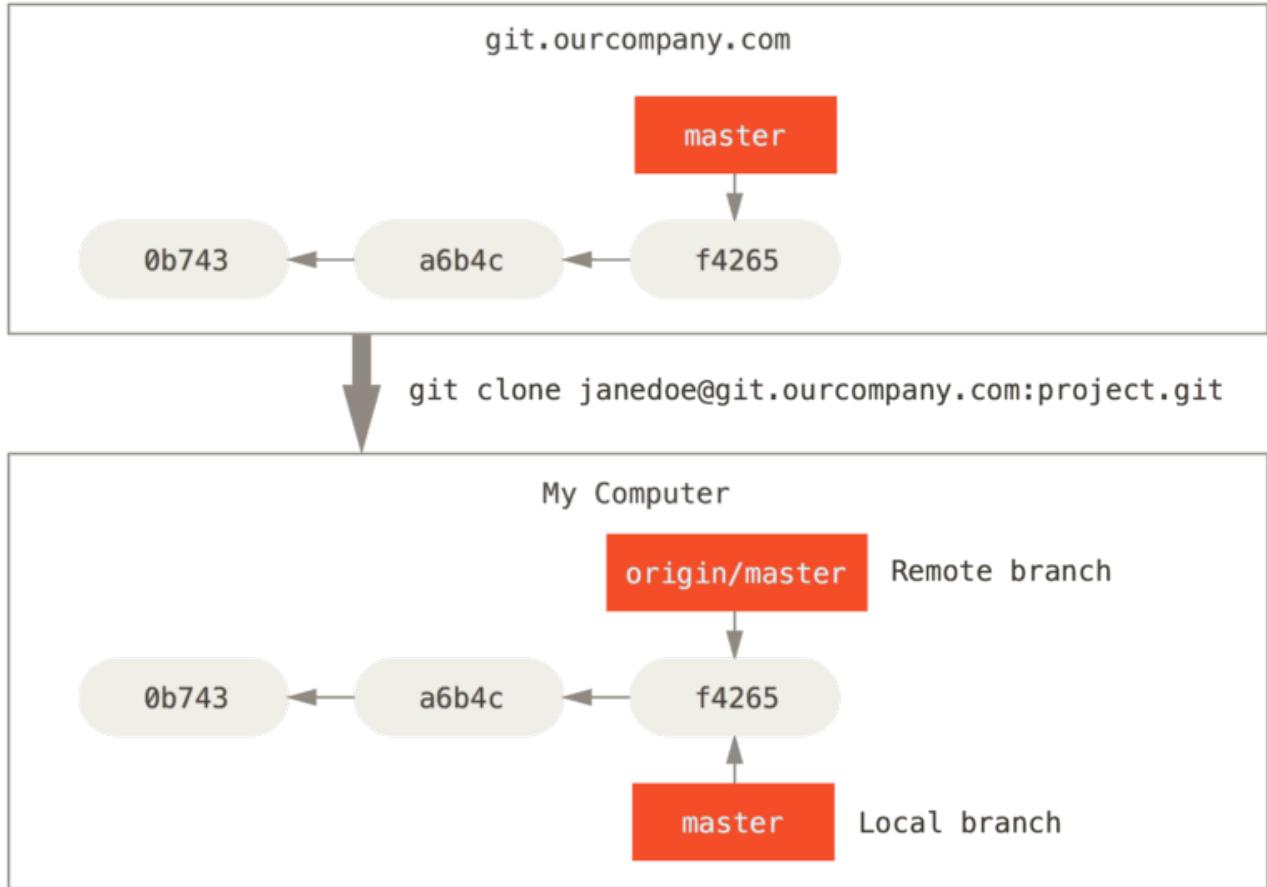


Figure 30. Server en lokale repositories na klonen

Als je wat werk doet op je lokale **master**-branch, en in de tussentijd pusht iemand anders iets naar **git.ourcompany.com** waardoor die **master**-branch wordt vernieuwd, dan zijn jullie histories verschillend vooruit geschoven. En zolang je geen contact hebt met de origin server, zal jouw **origin/master** verwijzing niet verplaatsen.

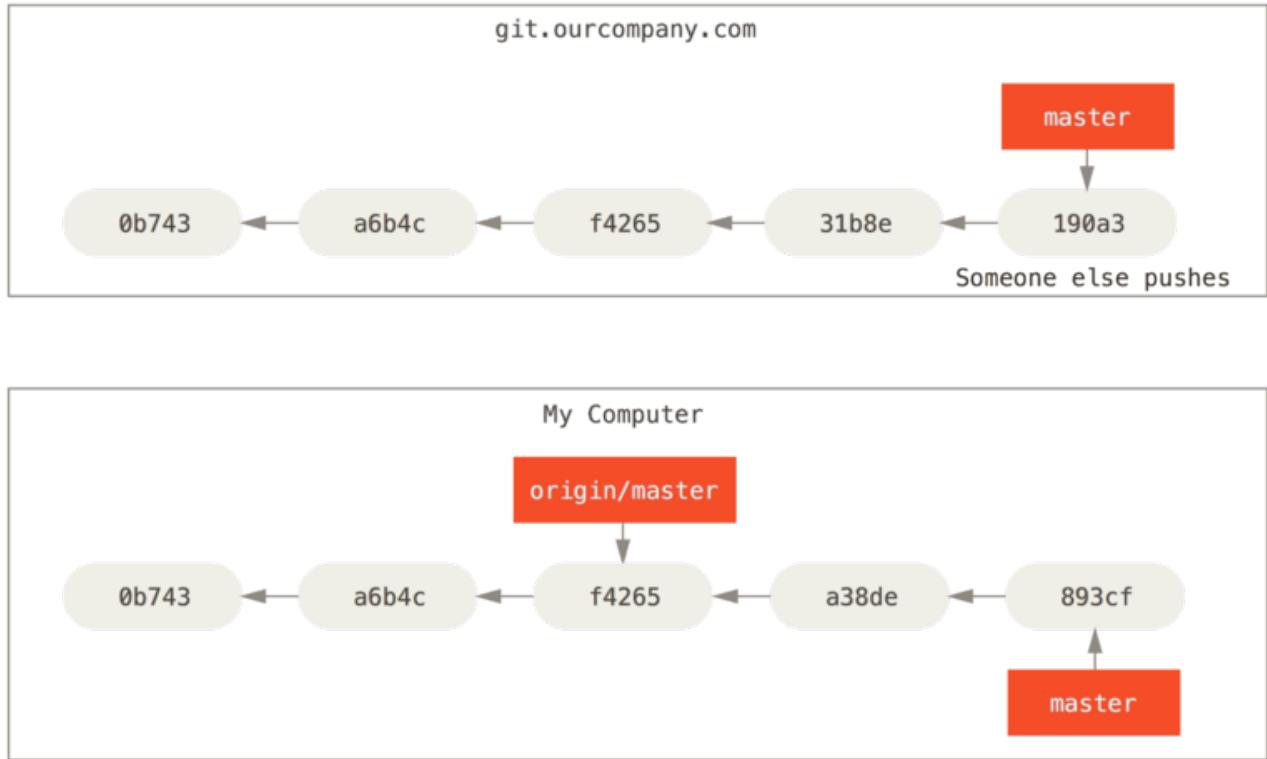


Figure 31. Lokaal en remote werk kan gaan afwijken

Om je werk te synchroniseren, voer je een `git fetch origin` commando uit. Dit commando bekijkt welke server “origin” is (in dit geval is het `git.ourcompany.com`), haalt gegevens er vanaf die je nog niet hebt en vernieuwt je lokale database, waarbij je `origin/master`-verwijzing naar zijn nieuwe positie verplaatst wordt die meer up-to-date is.

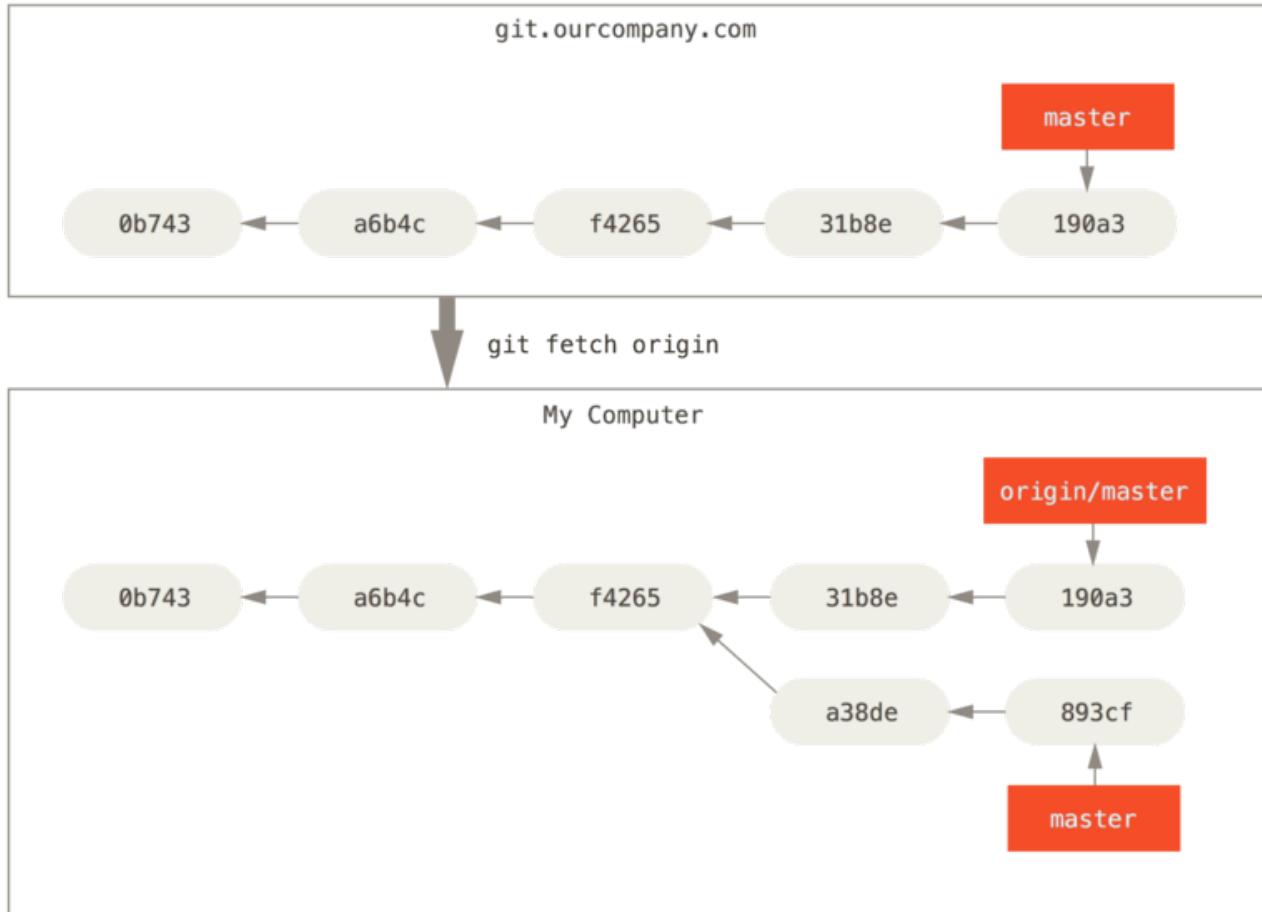


Figure 32. `git fetch` vernieuwt je remote referenties

Om het hebben van meerdere remote servers te demonstreren en hoe remote branches voor die remote projecten er uitzien, zullen we aannemen dat je nog een interne Git-server hebt die alleen wordt gebruikt voor ontwikkelingen gedaan door een van je sprint teams. Deze server bevindt zich op `git.team1.ourcompany.com`. Je kunt het als een nieuwe remote referentie toevoegen aan het project waar je nu aan werkt door het `git remote add` commando uit te voeren, zoals we behandeld hebben in [Git Basics](#). Noem deze remote `teamone`, wat jouw afkorting voor die hele URL wordt.

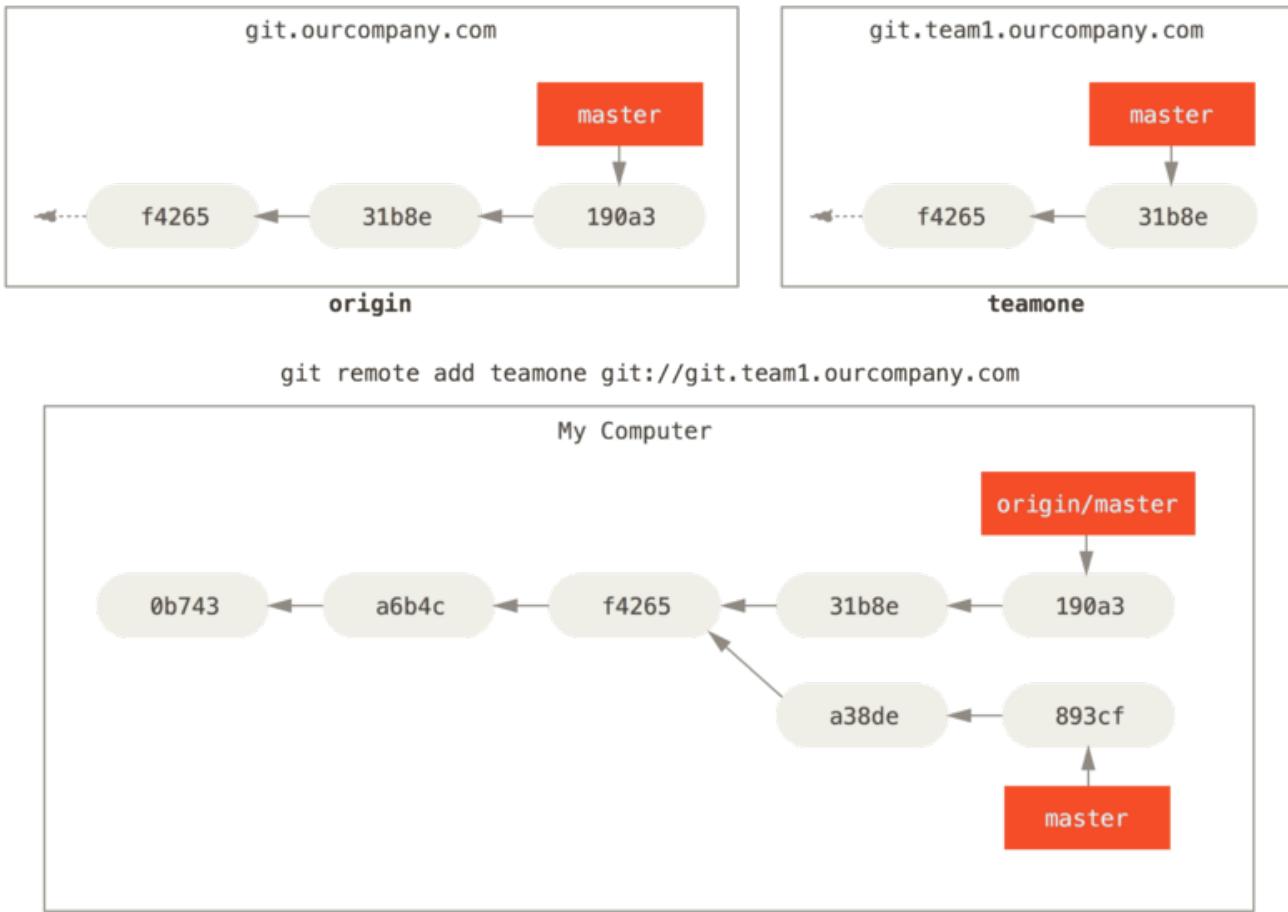


Figure 33. Een andere server toevoegen als remote

Nu kun je `git fetch teamone` uitvoeren om alles op te halen dat wat de `teamone` remote server heeft en jij nog niet. Omdat die server een subset heeft van de gegevens die jouw `origin` server op dit moment heeft, haalt Git geen gegevens op maar maakt een remote-tracking branch genaamd `teamone/master` aan en laat die wijzen naar de commit die `teamone` heeft als zijn `master`-branch.

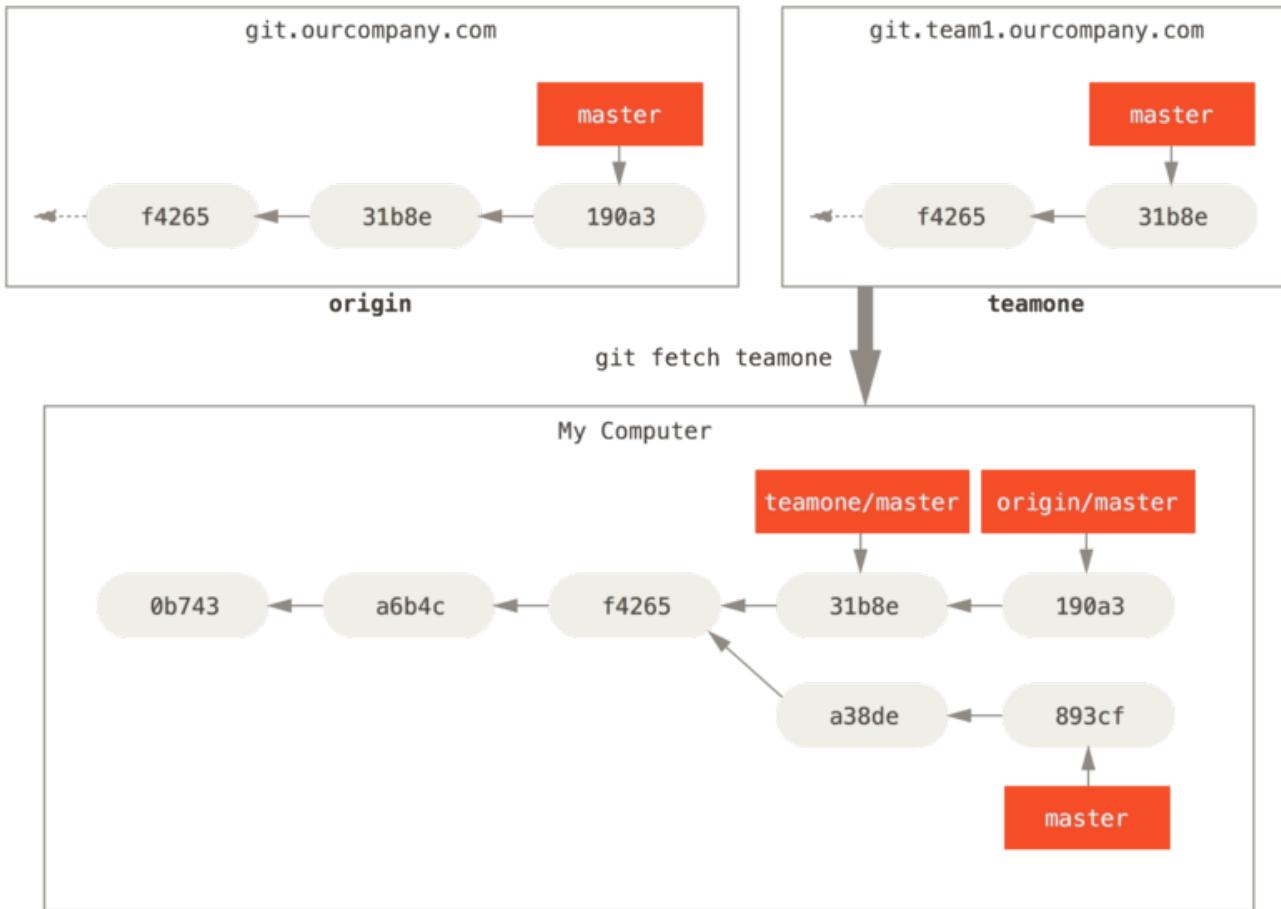


Figure 34. Remote tracking branch voor `teamone/master`

## Pushen

Als je een branch wil delen met de rest van de wereld, dan moet je het naar een remote terugzetten waar je schrijfttoegang op hebt. Je lokale branches worden niet automatisch gesynchroniseerd met de remotes waar je naar schrijft—je moet de branches die je wilt delen explicet pushen. Op die manier kun je privé branches gebruiken voor het werk dat je niet wil delen, en alleen die topic branches pushen waar je op wilt samenwerken.

Als je een branch genaamd `serverfix` hebt waar je met anderen aan wilt werken, dan kun je die op dezelfde manier pushen als waarop je dat voor de eerste branch hebt gedaan. Voer `git push <remote> <branch>` uit:

```
$ git push origin serverfix
Counting objects: 24, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (15/15), done.
Writing objects: 100% (24/24), 1.91 KiB | 0 bytes/s, done.
Total 24 (delta 2), reused 0 (delta 0)
To https://github.com/schacon/simplegit
 * [new branch]      serverfix -> serverfix
```

Dit is wel een beetje de bocht afsnijden. Git zal de `serverfix`-branchnaam automatisch expanderen naar `refs/heads/serverfix:refs/heads/serverfix`, wat staat voor “Neem mijn lokale `serverfix`

branch en push die om de serverfix branch van de remote te vernieuwen.”. We zullen het `refs/heads` gedeelte gedetailleerd behandelen in [Git Binnenwerk](#), maar je kunt het normaalgesproken weglaten. Je kun ook `git push origin serverfix:serverfix` doen, wat hetzelfde doet. Dit staat voor “Neem mijn serverfix en maak het de serverfix van de remote.” Je kunt dit formaat gebruiken om een lokale branch te pushen naar een remote branch die anders heet. Als je niet wil dat het `serverfix` heet aan de remote kant, kan je in plaats daarvan `git push origin serverfix:awesomebranch` gebruiken om je lokale `serverfix`-branch naar de `awesomebranch` op het remote project te pushen.

*Type niet elke keer je wachtwoord*

Als je een HTTPS URL gebruikt om mee te pushen, zal de Git server je elke keer vragen naar je usernaam en wachtwoord voor authenticatie. Standaard zal het je via de terminal vragen (prompten) om deze informatie zodat de server kan vaststellen of je mag pushen.



Als je dit niet elke keer wilt intypen als je pusht, kan je een “credential cache” opzetten. Het eenvoudigste is om het gewoon in het geheugen te houden voor een aantal minuten. Dit kan je simpel opzetten door `git config --global credential.helper cache` aan te roepen.

Voor meer informatie over de verschillende beschikbare credential caching opties, zie [Het opslaan van inloggegevens](#).

De volgende keer dat één van je medewerkers van de server fetcht zal deze een referentie krijgen naar de versie van `serverfix` op de server, onder de remote branch `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://github.com/schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Het is belangrijk om op te merken dat wanneer je een fetch doet die nieuwe remote-tracking branches ophaalt, je niet automatisch lokale aanpasbare kopieën daarvan hebt. In andere woorden, in dit geval heb je geen nieuwe `serverfix`-branch—je hebt alleen een `origin/serverfix` verwijzing die je niet kunt aanpassen.

Om dit werk in je huidige werk branch te mergen, kun je `git merge origin/serverfix` uitvoeren. Als je een eigen `serverfix`-branch wilt waar je op kunt werken, dan kun je deze op je remote-tracking branch baseren:

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

Dit maakt een lokale branch aan waar je op kunt werken, die begint waar `origin/serverfix` is.

## Tracking branches

Een lokale branch uitchecken van een remote-tracking branch creëert automatisch een zogenoemde “tracking branch” (“volg branch”, en de branch die hij volgt heet een “upstream branch”). Tracking branches zijn lokale branches die een directe relatie met een remote branch hebben. Als je op een tracking branch zit en `git pull` typt, dat weet Git automatisch naar welke server moet gaan om de wijzigingen op te halen en in welke branch deze moeten worden gemerged.

Als je een repository kloont, zal het over het algemeen automatisch een `master`-branch aanmaken die `origin/master` trackt. Maar je kan ook andere tracking branches aanmaken als je dat wilt—andere die branches op andere remotes tracken, of niet de `master`-branch tracken. Een eenvoudig voorbeeld is wat je zojuist gezien hebt: `git checkout -b <branch> <remotenaam>/<branch>` uitvoeren. Deze operatie komt dusdanig vaak voor dat Git de `--track` afkorting levert:

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

Sterker nog, dit is zo gewoon dat er zelfs een afkorting voor de afkorting is. Als de branch naam die je uit wilt checken (a) niet bestaat en (b) dezelfde naam maar op een remote voorkomt, zal Git een tracking branch voor je aanmaken:

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

Om een lokale branch te maken met een andere naam dan de remote branch, kun je simpelweg de eerste variant met een andere lokale branchnaam gebruiken:

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

Nu zal je lokale `sf`-branch automatisch pullen van `origin/serverfix`.

Als je al een lokale branch hebt en je wilt deze koppelen aan een remote branch die je zojuist gepulld hebt, of de stroomopwaartse branch die je trackt wijzigen, kan je de `-u` of `--set-upstream-to` optie gebruiken bij de `git branch` om het expliciet te zetten op de momenten dat jij het wilt.

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

### *Upstream verwijzing*



Als je een tracking branch ingericht hebt, kan je hiernaar refereren met de `@{upstream}` of `@{u}` verwijzing. Dus als je op de `master`-branch zit en deze `origin/master` trackt, kan je iets als `git merge @{u}` opgeven in plaats van `git merge origin/master` als je zou willen.

Als je wilt zien welke tracking branches je ingericht hebt, kan je de `-vv` optie aan `git branch` meegeven. Dit zal jouw lokale branches afdrukken met meer informatie, inclusief wat elk van de branches trackt en of je lokale branch voorloopt, achterloopt of beide.

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

Dus hier kunnen we zien dat onze `iss53 origin/iss53` trackt en twee “voorloopt”, wat inhoudt dat we lokaal twee commits hebben die nog niet naar de server zijn gepusht. We kunnen ook zien dat onze `master`-branch `origin/master` trackt en up-to-date is. Vervolgens zien we dat onze `serverfix`-branch de `server-fix-good`-branch trackt op onze `teamone` server en drie voorloopt en een achterloopt. Dit betekent dat er een commit op de server staat die we nog niet hebben gemerged en er drie commits lokaal staan die we nog niet gepusht hebben. Tenslotte kunnen we zien dat onze `testing`-branch geen enkele remote branch trackt.

Het is belangrijk om op te merken dat deze getallen accuraat zijn op het moment dat je voor het laatst gefetcht hebt van elke server. Dit commando maakt geen contact met de servers, het vertelt je wat het van deze servers lokaal heeft opgeslagen. Als je volledige up-to-date gegevens wilt hebben over voorsprong en achterstand zal je van alle remotes moeten fetchen voordat je dit aanroeft. Je zou iets als volgt kunnen doen:

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

## Pullen

Waar het `git fetch` commando alle wijzigingen van de server zal ophalen die je nog niet hebt, zal het je werk directory helemaal niet wijzigen. Het haalt simpelweg de gegevens voor je op en laat het mergen aan jou over. Er is echter een commando `git pull` geheten die feitelijk een `git fetch` onmiddelijk gevolgd door een `git merge` is in de meeste gevallen. Als je een tracking branch opgezet hebt zoals in de vorige paragraaf getoond, expliciet opgezet of voor je opgezet door de `clone` of `checkout` commando's, zal `git pull` kijken welke server en branch je huidige branch trackt, van die server fetchen en de betreffende remote branch daarin mergen.

Over het algemeen is het beter om de `fetch` en `merge` commando's expliciet te gebruiken omdat de magie van `git pull` vaak verwarrend kan zijn.

## Remote branches verwijderen

Stel dat je klaar bent met een remote branch - zeg maar, jij en je medewerkers zijn klaar met een feature en hebben het gemerged in de `master`-branch van de remote (of welke branch jullie stabiele code ook in zit). Je kunt een remote branch verwijderen met de `--delete` optie bij `git push`. Als je de `serverfix`-branch van de server wilt verwijderen, dan voer je het volgende uit:

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

Alles wat dit doet is de pointer van de server verwijderen. De Git server zal over het algemeen de gegevens nog een poos behouden totdat de garbage collection draait, dus als het per ongeluk verwijderd is, is het vaak eenvoudig terug te halen.

## Rebasen

In Git zijn er twee hoofdmanieren om wijzigingen te integreren van de ene branch in een andere: de `merge` en de `rebase`. In deze paragraaf ga je leren wat rebasen is, hoe je dat moet doen, waarom het een zeer bijzonder stukje gereedschap is en in welke gevallen je het niet wilt gebruiken.

### De simpele rebase

Als je het eerdere voorbeeld van [Eenvoudig mergen \(samenvoegen\)](#) erop terugslaat, dan zul je zien dat je werk is uiteengelopen en dat je commits hebt gedaan op de twee verschillende branches.

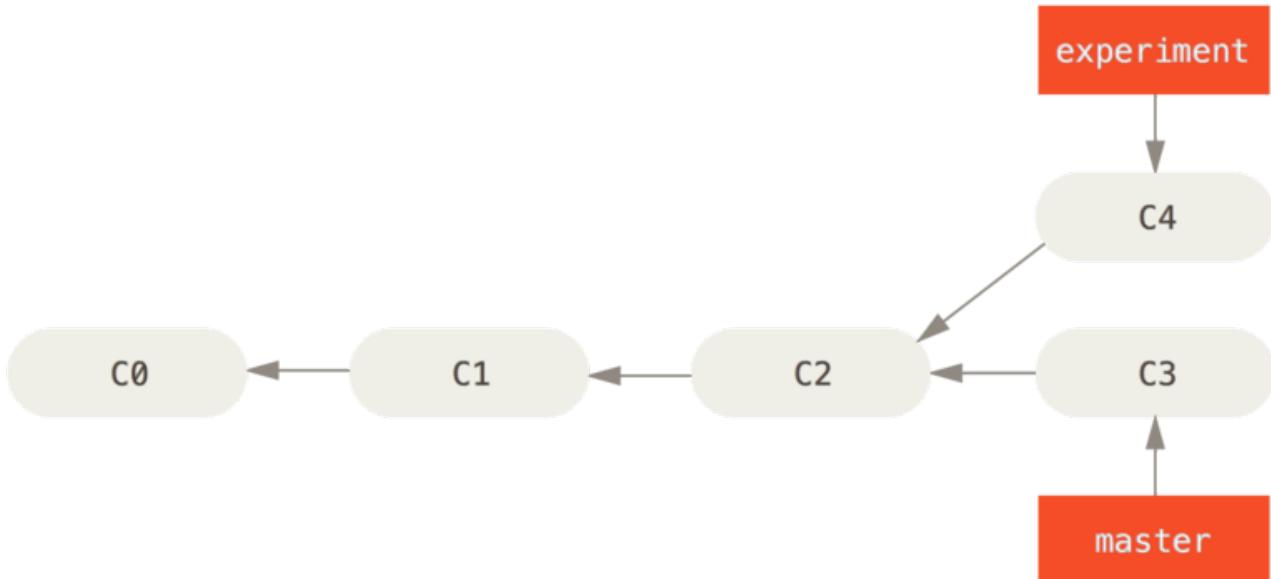


Figure 35. Eenvoudige uiteengelopen historie

De simpelste manier om de branches te integreren, zoals we al hebben besproken, is het `merge` commando. Het voert een drieweg-merge uit tussen de twee laatste snapshots van de branches (`C3` en `C4`), en de meest recente gezamenlijke voorouder van die twee (`C2`), en maakt een nieuw snapshot (en commit).

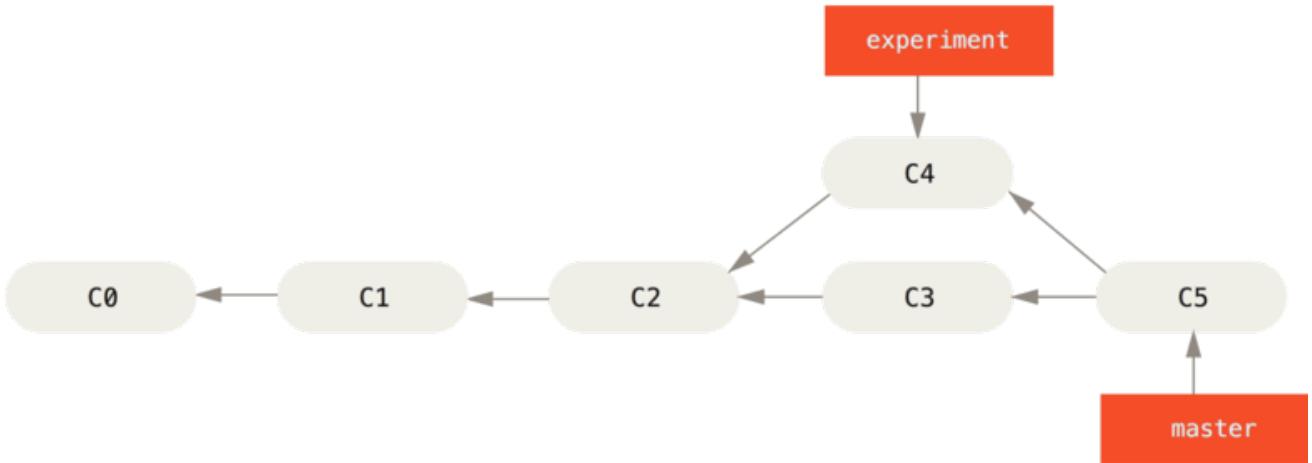


Figure 36. Mergen om uiteengelopen werk historie te integreren

Maar, er is nog een manier: je kunt de patch van de wijziging die werd geïntroduceerd in **C4** pakken en die opnieuw toepassen op **C3**. In Git, wordt dit *rebase* genoemd. Met het `rebase` commando kan je alle wijzigingen pakken die zijn gecommit op de ene branch, en ze opnieuw afspelen op een andere.

In dit voorbeeld zou je het volgende uitvoeren:

```

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

```

Het gebeurt door naar de gezamenlijke voorouder van de twee branches te gaan (degene waar je op zit en degene waar je op rebaset), de diff te nemen die geïntroduceerd is door elke losse commit op de branch waar je op zit, die diffs in tijdelijke bestanden te bewaren, de huidige branch terug te zetten naar dezelfde commit als de branch waar je op rebaset, en uiteindelijk elke diff een voor een te applyen.

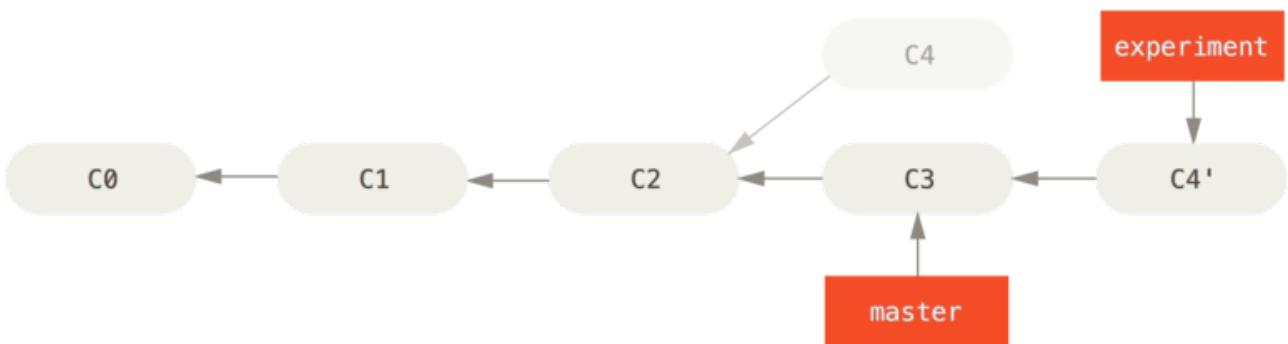


Figure 37. De wijziging gemaakt in **C4** rebasen naar **C3**

En nu kan je teruggaan naar de master branch en een fast-forward merge uitvoeren.

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

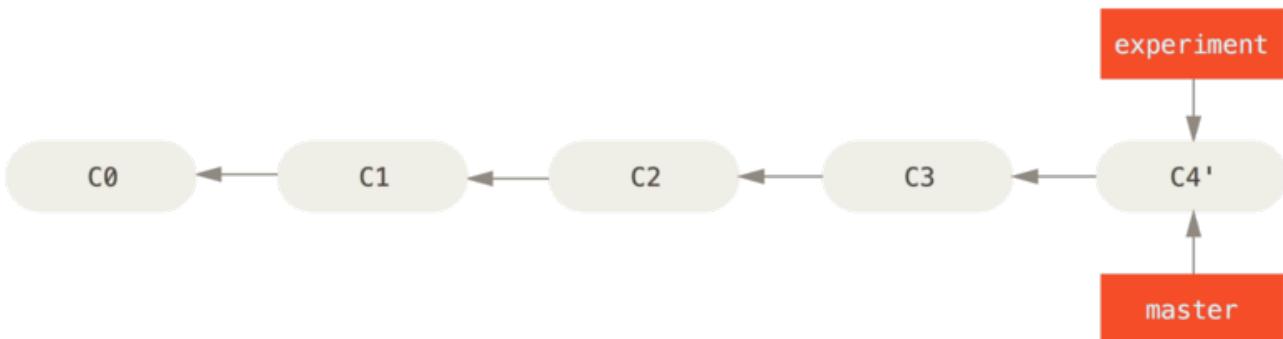


Figure 38. De master branch fast-forwarden

Nu is het snapshot waar **C4** naar wijst precies dezelfde als degene waar **C5** naar wees in het merge voorbeeld. Er zit geen verschil in het eindresultaat van de integratie, maar rebasen zorgt voor een duidelijker historie. Als je de log van een branch die gerebased is bekijkt, ziet het eruit als een lineaire historie: het lijkt alsof al het werk volgordeelijker is gebeurd, zelfs wanneer het in werkelijkheid parallel eraan gedaan is.

Vaak zal je dit doen om er zeker van te zijn dat je commits netjes toegepast kunnen worden op een remote branch - misschien in een project waar je aan probeert bij te dragen, maar welke je niet beheert. In dit geval zou je het werk in een branch uitvoeren en dan je werk rebasen op **origin/master** als je klaar ben om je patches in te sturen naar het hoofdproject. Op die manier hoeft de beheerder geen integratiewerk te doen - gewoon een fast-forward of een schone apply.

Merk op dat de snapshot waar de laatste commit op het eind naar wijst, of het de laatste van de gerebasede commits voor een rebase is of de laatste merge commit na een merge, detzelfde snapshot is - alleen de historie is verschillend. Rebasen speelt veranderingen van een werklijn opnieuw af op een andere, in de volgorde waarin ze gemaakt zijn, terwijl mergen de eindresultaten pakt en die samenvoegt.

## Interessantere rebases

Je kunt je rebase ook opnieuw laten afspelen op iets anders dan de rebase doel branch. Pak een historie zoals in [Een historie met een topic branch vanaf een andere topic branch](#), bijvoorbeeld. Je hebt een topic branch afgesplitst (**server**) om wat server-kant functionaliteit toe te voegen aan je project en toen een keer gecommit. Daarna heb je daar vanaf gebranched om de client-kant wijzigingen te doen (**client**) en een paar keer gecommit. Als laatste, ben je teruggegaan naar je server branch en hebt nog een paar commits gedaan.

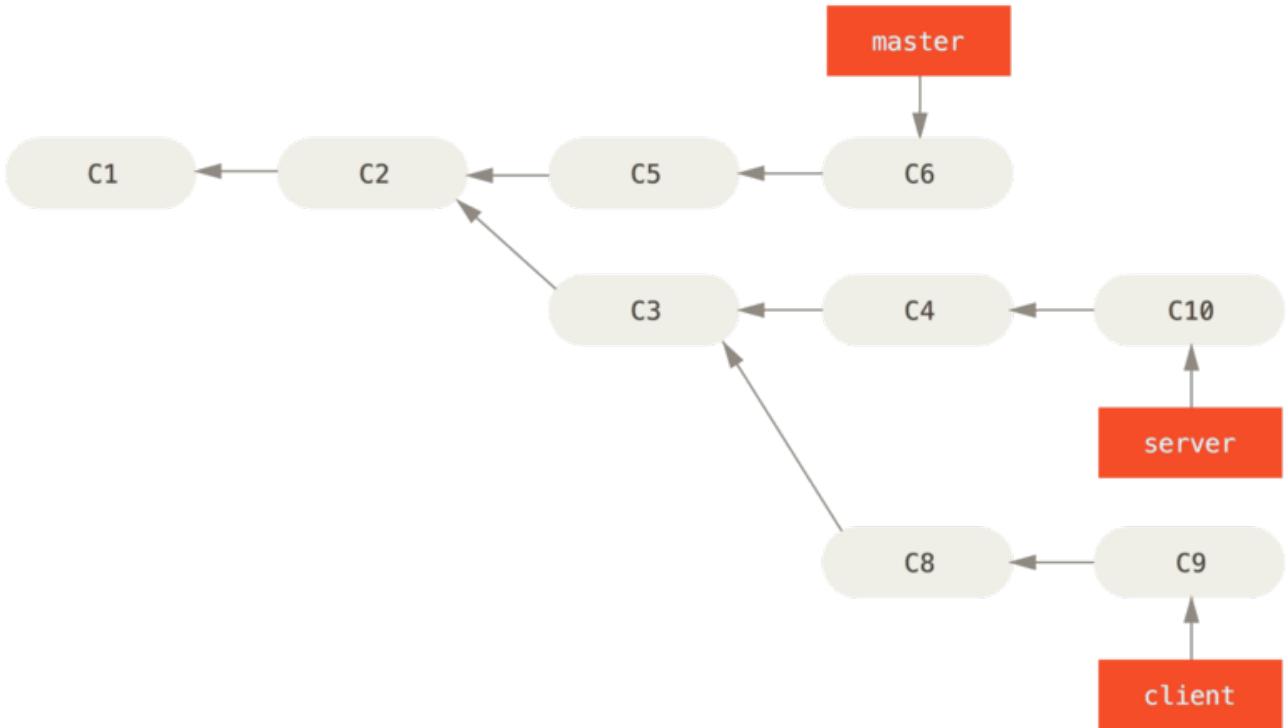


Figure 39. Een historie met een topic branch vanaf een andere topic branch

Stel nu, je besluit dat je de client-kant wijzigingen wilt mergen in je hoofdlijn voor een release, maar je wilt de server-kant wijzigingen nog vasthouden totdat het verder getest is. Je kunt de wijzigingen van client pakken, die nog niet op server zitten (`C8` en `C9`) en die opnieuw afspelen op je `master`-branch door de `--onto` optie te gebruiken van `git rebase`:

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

Dit zegt in feite, '*Check de `client`-branch uit, verzamel de patches van de gezamenlijke voorouder van de `client` en de `server`-branches, en speel die opnieuw af in de `client`-branch alsof deze direct afgeleid was van de `master`-branch.*' Het is een beetje ingewikkeld, maar het resultaat is best wel gaaf.

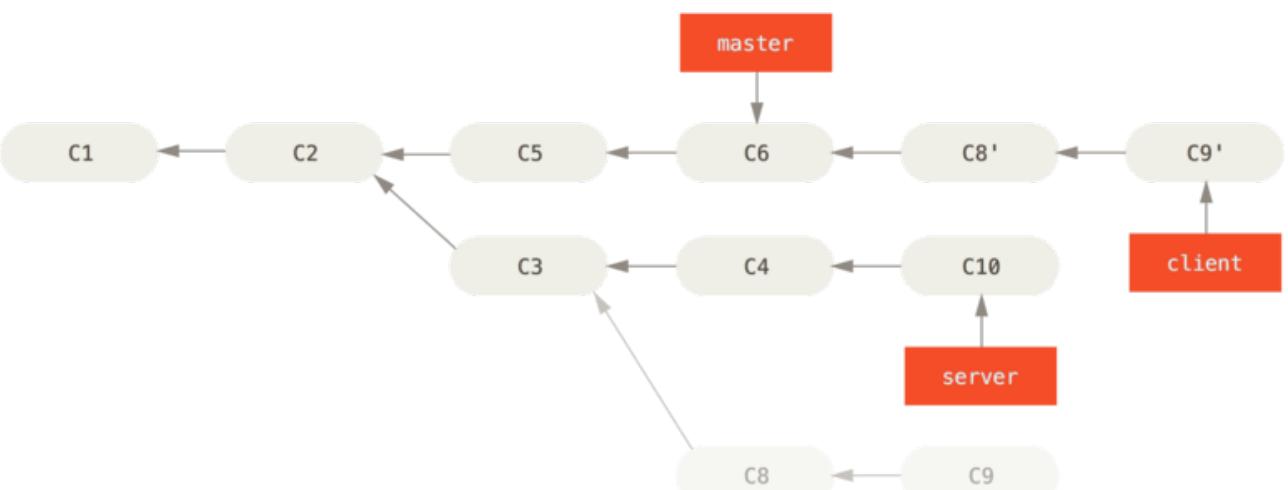


Figure 40. Een topic branch rebasen vanaf een andere topic branch

Nu kun je een fast-forward doen van je `master`-branch (zie [Je master branch fast-forwarden om de](#)

client branch wijzigingen mee te nemen):

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

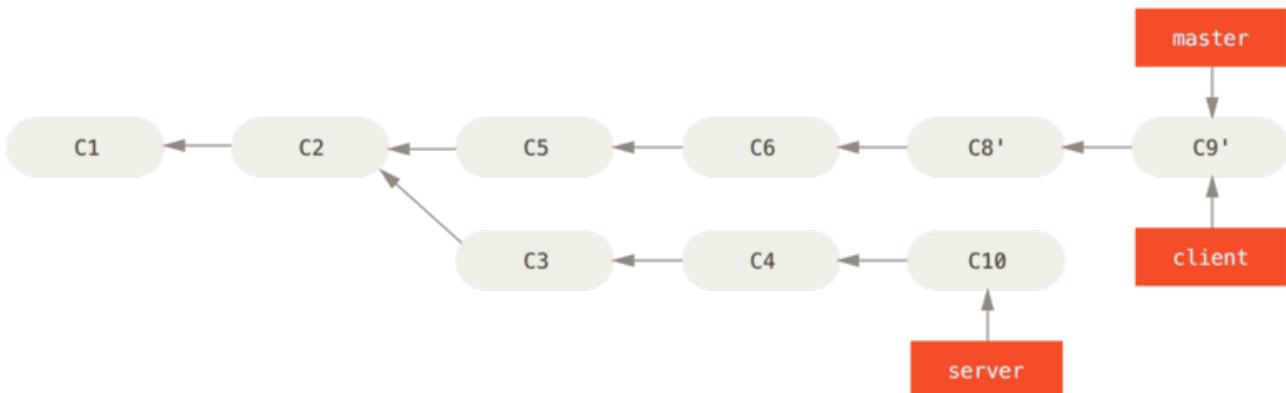


Figure 41. Je master branch fast-forwarden om de client branch wijzigingen mee te nemen

Stel dat je besluit om de server branch ook te pullen. Je kunt de server branch rebasen op de master branch zonder het eerst te hoeven uitchecken door `git rebase <basisbranch> <topicbranch>` uit te voeren - wat de topic branch voor je uitcheckt (in dit geval, `server`) en het opnieuw afspeelt op de basis branch (`master`):

```
$ git rebase master server
```

Dit speelt het `server` werk opnieuw af op het `master` werk, zoals getoond in [Je server branch op je master branch rebasen](#).

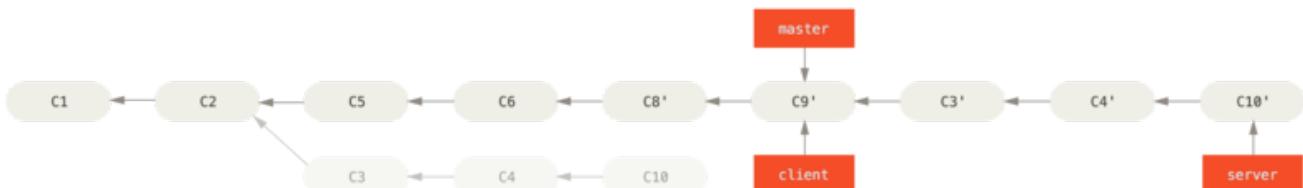


Figure 42. Je server branch op je master branch rebasen

Daarna kan je de basis branch (`master`) fast-forwarden:

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

Je kunt de `client` en `server`-branches verwijderen, omdat al het werk geïntegreerd is en je ze niet meer nodig hebt, en de historie voor het hele proces ziet eruit zoals in [Uiteindelijke commit historie](#):

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



Figure 43. Uiteindelijke commit historie

## De gevaren van rebasen

Ahh, maar de zegeningen van rebasen zijn niet geheel zonder nadelen, samengevat in één enkele regel:

**Rebase geen commits die buiten je repository bekend zijn.**

Als je die richtlijn volgt, kan je weinig gebeuren. Als je dat niet doet, zullen mensen je gaan haten en je zult door vrienden en familie uitgehoond worden.

Als je spullen rebaset, zet je bestaande commits buiten spel en maak je nieuwe aan die vergelijkbaar zijn maar anders. Wanneer je commits ergens pusht en andere pullen deze en baseren daar werk op, en vervolgens herschrijf je die commits met `git rebase` en pusht deze weer, dan zullen je medewerkers hun werk opnieuw moeten mergen en zal het allemaal erg vervelend worden als je hun werk probeert te pullen in het jouwe.

Laten we eens kijken naar een voorbeeld hoe werk rebasen dat je publiek gemaakt hebt problemen kan veroorzaken. Stel dat je van een centrale server clonet en dan wat werk aan doet. Je commit-historie ziet eruit als volgt:

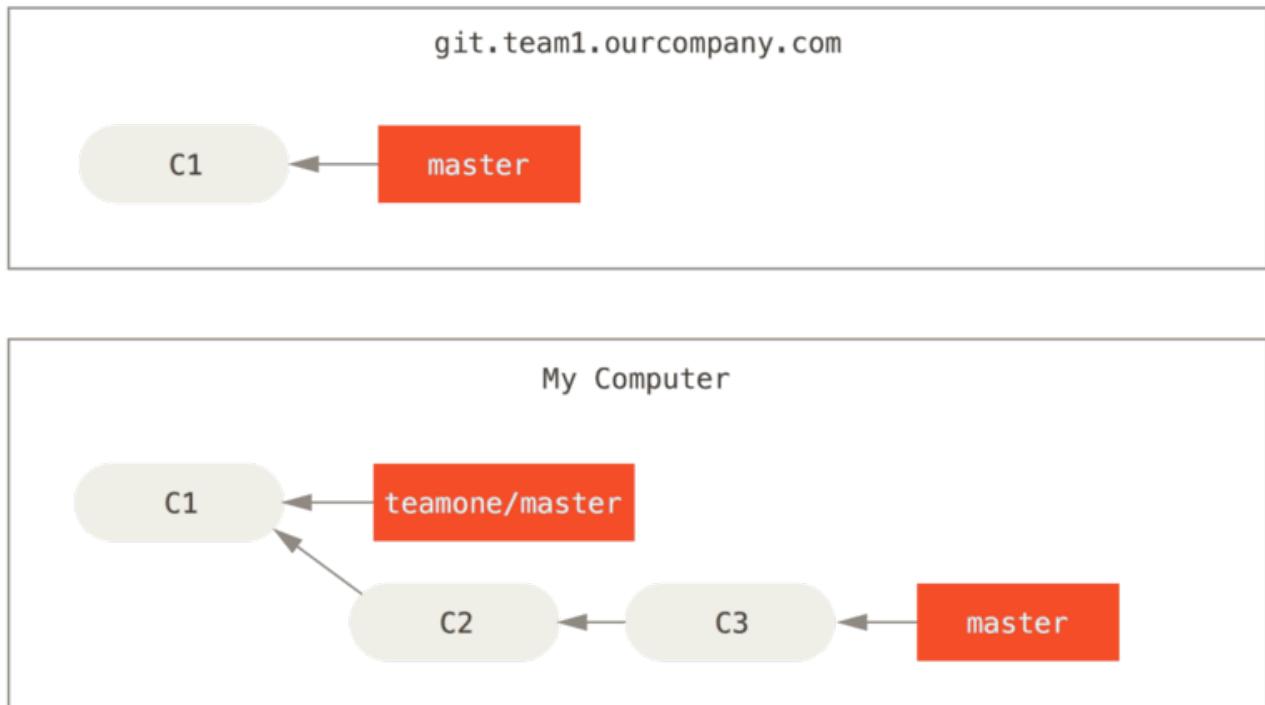


Figure 44. Clone een repository, en baseer er wat werk op

Nu doet iemand anders wat meer werk wat een merge bevat, en pusht dat werk naar de centrale server. Je fetcht dat en merget de nieuwe remote branch in jouw werk, zodat je historie eruitziet zoals dit:

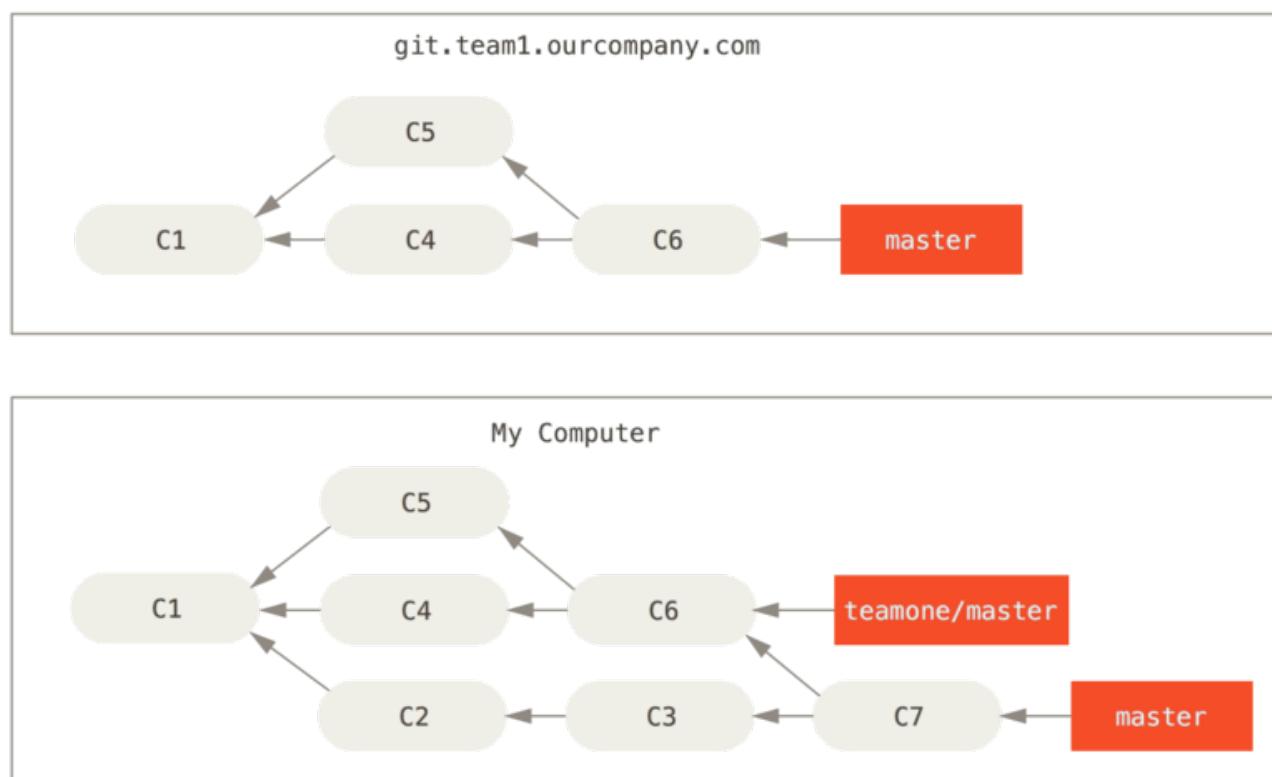


Figure 45. Fetch meer commits, en merge ze in jouw werk

Daarna, beslist de persoon die het werk gepusht heeft om erop terug te komen en in plaats daarvan

zijn werk te gaan rebasen; hij voert een `git push --force` uit om de historie op de server te herschrijven. Je pullt daarna van die server, waarbij je de nieuwe commits binnen krijgt.

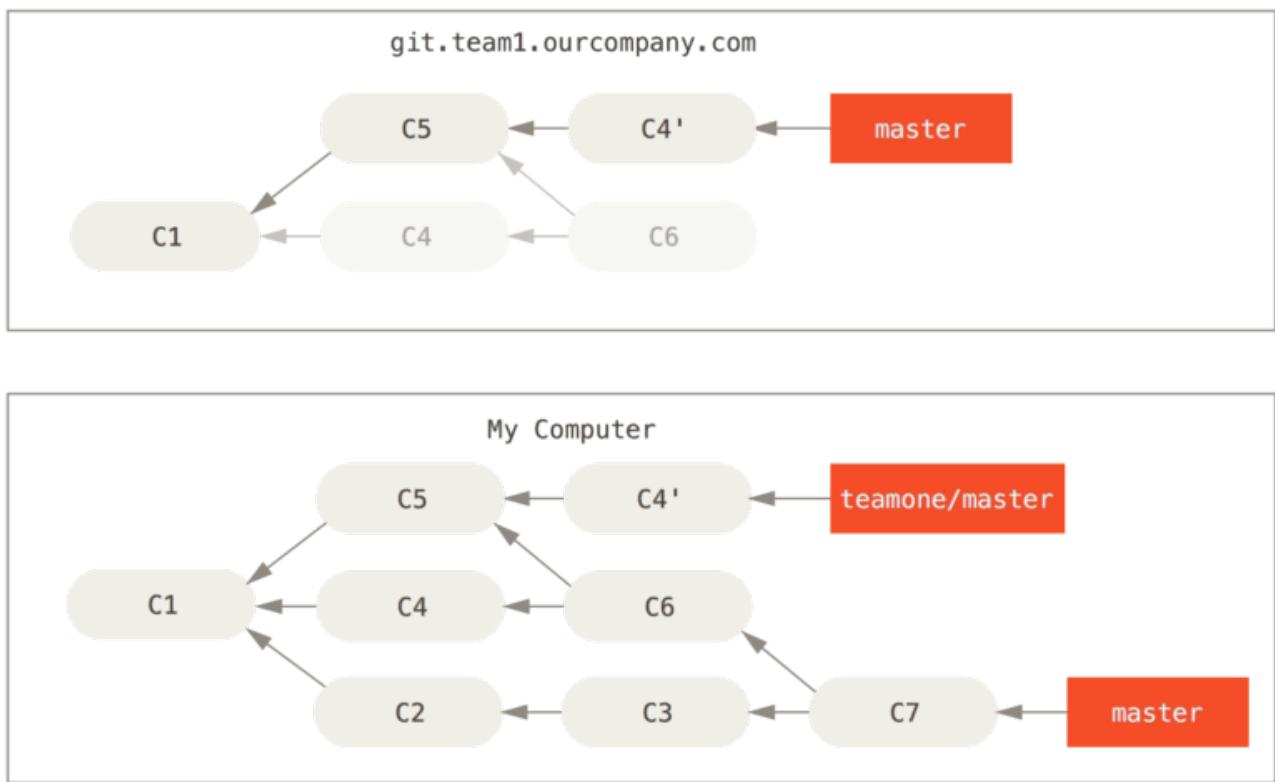


Figure 46. Iemand pusht gerebaseerde commits, daarbij commits buiten spel zettend waar jij werk op gebaseerd hebt

Nu zitten jullie beiden in de penarie. Als jij een `git pull` doet, ga je een commit merge maken waar beide tijdslijnen in zitten, en je repository zal er zo uit zien:

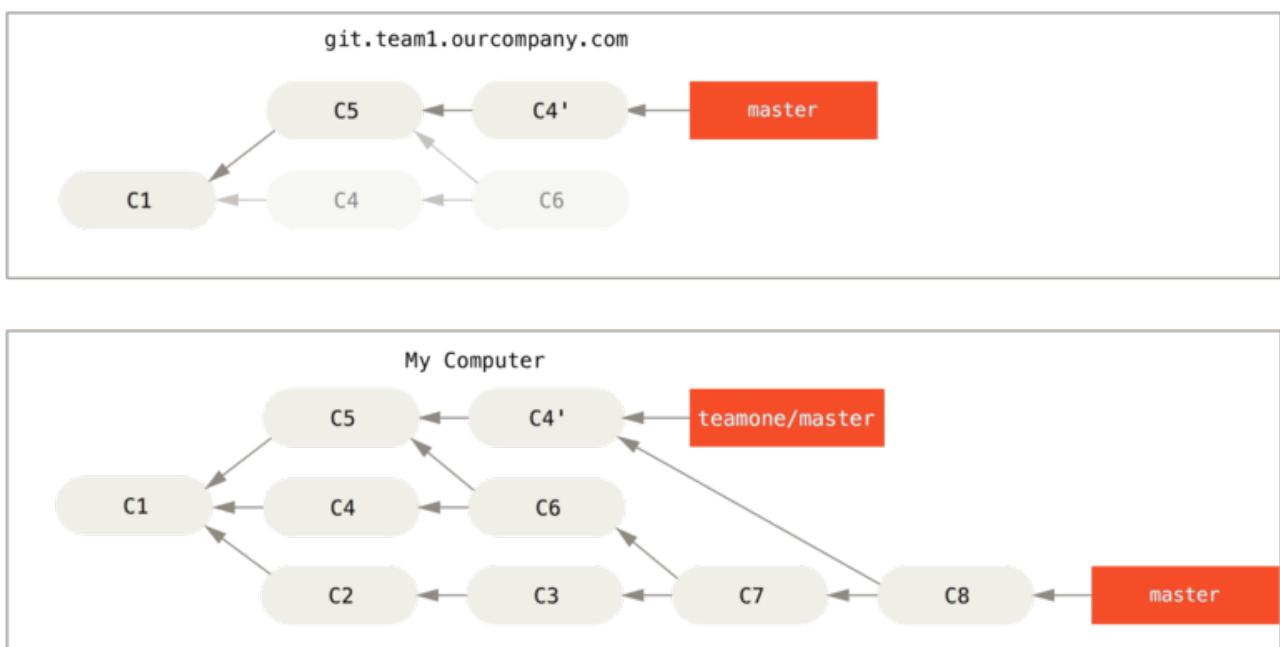


Figure 47. Je merget hetzelfde werk opnieuw in een nieuwe merge commit

Als je een `git log` uitvoert als je historie er zo uitziet, zie je twee commits die dezelfde auteur,

datum en bericht hebben, wat verwarrend is. Daarnaast, als je deze historie naar de server terugpusht, zal je al deze gerebaseerde commits opnieuw herintroduceren op centrale server, wat weer andere mensen zou kunnen verwarreren. Het is redelijk veilig om aan te nemen dat de andere ontwikkelaar C4 en C6 niet in de historie wil, dat is juist de reden waarom ze heeft gerebased.

## Rebaset spullen rebasen

**Mocht** je in zo'n situatie belanden, heeft Git nog wat tovertrucs in petto die je kunnen helpen. Als iemand of een aantal mensen in jouw team met pushes wijzigingen hebben geforceerd die werk overschrijven waar jij je werk op gebaseerd hebt, is het jouw uitdaging om uit te vinden wat jouw werk is en wat zij herschreven hebben.

Het komt zo uit dat naast de SHA-1 checksum van de commit, Git ook een checksum berekent die enkel is gebaseerd op de patch die is geïntroduceerd met de commit. Dit heet een "patch-id".

Als je werk pullt die was herschreven en deze rebased op de nieuwe commits van je partner, kan Git vaak succesvol uitvinden wat specifiek van jou is en deze opnieuw afspelen op de nieuwe branch.

Bijvoorbeeld in het vorige scenario, als in plaats van een merge te doen we in een situatie zijn die beschreven is in [Iemand pusht gerebaseerde commits, daarbij commits buitenspel zettend waar jij werk op gebaseerd hebt en we git rebase teamone/master aanroepen](#), zal Git:

- Bepalen welk werk uniek is in onze branch (C2, C3, C4, C6, C7)
- Bepalen welke geen merge commits zijn (C2, C3, C4)
- Bepalen welke nog niet herschreven zijn in de doel-branch (alleen C2 en C3, omdat C4 dezelfde patch is als C4')
- Deze commits op [teamone/master](#) afspelen

Dus in plaats van het resultaat dat we zien in [Je merget hetzelfde werk opnieuw in een nieuwe merge commit](#), zouden we eindigen met iets wat meer lijkt op [Rebase op een force-pushed rebase werk..](#)

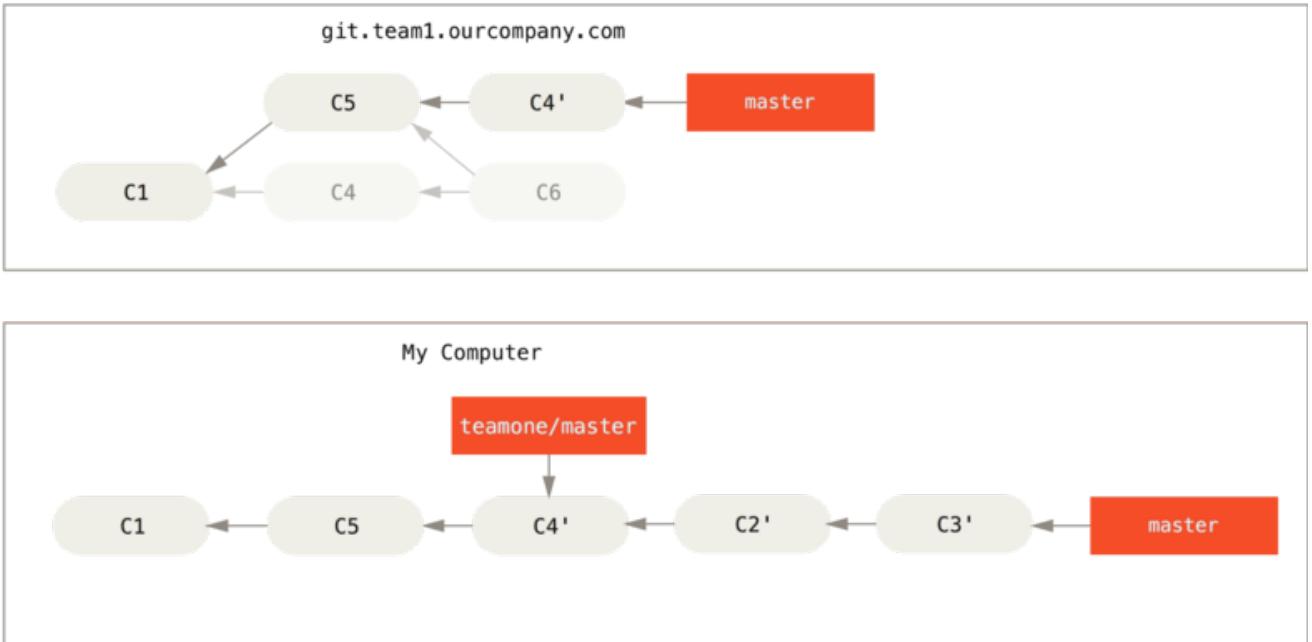


Figure 48. Rebase op een force-pushed rebase werk.

Dit werkt alleen als de door je partner gemaakte C4 en C4' vrijwel dezelfde patch zijn. Anders kan de rebase niet achterhalen dat het een dupliekaat is en zal dan een andere C4-achtige patch toevoegen (die waarschijnlijk niet schoon kan worden toegepast, omdat wijzigingen ongeveer hetzelfde daar al staan).

Je kunt dit versimpelen door een `git pull --rebase` in plaats van een gewone `git pull` te draaien. Of in dit geval kan je handmatig een `git fetch` gevolgd door een `git rebase teamone/master` uitvoeren.

Als je `git pull` gebruikt en `--rebase` de standaard maken, kan je de `pull.rebase` configuratie waarde zetten op `git config --global pull.rebase true`.

Als je rebases behandelt als een manier om op te schonen en met de commits te werken voordat je ze pusht, en als je alleen commits rebased die nooit eerder publiekelijk beschikbaar waren, is er geen probleem. Als je commits rebaset die al publiekelijk gepusht zijn, en mensen kunnen hun werk gebaseerd hebben op die commits, dan heb je de basis gelegd voor wat frustrerende problemen, en de hoon van je teamgenoten.

Als jij of een partner het nodig vinden op een gegeven moment, verzeker je ervan dat iedereen weet dat ze een `git pull --rebase` moeten draaien om de pijn te verzachten nadat dit gebeurt is.

## Rebase vs. Merge

Nu we rebases en mergen in actie hebben laten zien, kan je je afvragen welk van de twee beter is. Voordat we die vraag kunnen beantwoorden, laten we eerst een stapje terug nemen en bespreken wat historie eigenlijk inhoudt.

Een standpunt is dat de commit historie van jouw repository een **vastlegging is van wat daadwerkelijk gebeurd is**. Het is een historisch document, op zichzelf waardevol, waarmee niet mag worden gerommeld. Vanuit dit gezichtspunt, is het wijzigen van de commit historie bijna

vloeken in de kerk; je bent aan het *liegen* over wat er werkelijk gebeurd is. Wat hindert het dat er een slorige reeks merge commits waren? Dat is hoe het gebeurd is, en de repository moet dat bewaren voor het nageslacht.

Een ander standpunt is dat de commit historie het **verhaal is hoe jouw project tot stand is gekomen**. Je puliceert ook niet het eerste manuscript van een boek, en de handleiding hoe je software te onderhouden verdient zorgvuldig samenstellen. Dit is het kamp dat gereedschappen als rebase en filter-branch gebruikt om het verhaal te vertellen dat het beste is voor toekomstige lezers.

Nu, terug naar de vraag of mergen of rebasen beter is: hopelijk snap je nu dat het niet zo eenvoudig ligt. Git is een krachtig instrument, en stelt je in staat om veel dingen te doen met en middels je historie, maar elk team en elk project is anders. Nu je weet hoe beide werken, is het aan jou om te besluiten welke het beste is voor jouw specifieke situatie.

Om het beste van beide aanpakken te krijgen is het over het algemeen het beste om lokale wijzigingen die je nog niet gedeeld hebt te rebasen voordat je ze pusht zodat je verhaal het schoonste blijft, maar nooit iets te rebasen wat je elders gepusht hebt.

## Samenvatting

We hebben de basis van branchen en mergen in Git behandeld. Je zou je op je gemak moeten voelen met het maken en omschakelen naar nieuwe branches, omschakelen tussen branches, en lokale branches te mergen. Je zou ook in staat moeten zijn om je branches te delen door ze naar een gedeelde server te pushen, met anderen op gedeelde branches samen te werken en je branches te rebasen voordat ze gedeeld worden. In het volgende deel gaan we de materie behandelen wat je nodig gaat hebben om jouw eigen Git repository-hosting server op te zetten.

# Git op de server

Je zou nu de alledaagse taken waarvoor je Git zult gebruiken moeten kunnen uitvoeren. Echter, om enige vorm van samenwerking te hebben in Git is een remote Git repository nodig. Technisch gezien kan je wijzigingen pushen naar en pullen van de persoonlijk repositories van anderen, maar dat wordt afgeraden omdat je vrij gemakkelijk het werk waar anderen mee bezig zijn in de war kunt schoppen als je niet oppast. Daarnaast wil je dat je medewerkers de repository kunnen bereiken, zelfs als jouw computer van het netwerk is; het hebben van een meer betrouwbare gezamenlijke repository is vaak nuttig. De voorkeursmethode om met iemand samen te werken is daarom een tussenliggende repository in te richten waar alle partijen toegang tot hebben en om daar naartoe te pushen en vandaan te pullen.

Een Git server draaien is relatief eenvoudig. Als eerste kies je met welke protocollen je de server wilt laten communiceren. In het eerste gedeelte van dit hoofdstuk zullen we de beschikbare protocollen bespreken met de voor- en nadelen van elk. De daarop volgende paragrafen zullen we een aantal veel voorkomende opstellingen bespreken die van die protocollen gebruik maken en hoe je je server ermee kunt opzetten. Als laatste laten we een paar servers van derden zien, als je het niet erg vindt om je code op de server van een ander te zetten en niet het gedoe wilt hebben van het opzetten en onderhouden van je eigen server.

Als je niet van plan bent om je eigen server te draaien, dan kun je de direct naar de laatste paragraaf van dit hoofdstuk gaan om wat mogelijkheden van online accounts te zien en dan door gaan naar het volgende hoofdstuk, waar we diverse zaken bespreken die komen kijken bij het werken met een gedistribueerde versiebeheer omgeving.

Een remote repository is over het algemeen een *bare repository* (kale repository): een Git repository dat geen werkdirectory heeft. Omdat de repository alleen gebruikt wordt als een samenwerkingspunt, is er geen reden om een snapshot op de schijf te hebben; het is alleen de Git data. Een kale repository is eenvoudigweg de inhoud van de `.git` directory in je project, en niets meer.

## De protocollen

Git kan vier verschillende netwerk protocollen gebruiken om data uit te wisselen: Lokaal, HTTP, Beveiligde Shell (Secure Shell, SSH) en Git. Hier bespreken we wat deze zijn en in welke omstandigheden je ze zou willen gebruiken (of juist niet).

### Lokaal protocol

Het simpelste is het *Lokale protocol*, waarbij de remote repository in een andere directory op de schijf van dezelfde host staat. Deze opzet wordt vaak gebruikt als iedereen in het team toegang heeft op een gedeeld bestandssysteem zoals een NFS mount, of in het minder waarschijnlijke geval dat iedereen op dezelfde computer werkt. Het laatste zou niet ideaal zijn, want dan zouden alle repositories op dezelfde computer staan, wat de kans op een catastrofaal verlies van gegevens veel groter maakt.

Als je een gedeeld bestandssysteem hebt, dan kun je clonen, pushen en pullen van een op een lokale bestands-gebaseerde repository. Om een dergelijk repository te clonen, of om er een als een remote

aan een bestaand project toe te voegen, moet je het pad naar het repository als URL gebruiken. Bijvoorbeeld, om een lokaal repository te clonen, kun je zo iets als het volgende uitvoeren:

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

Of je kunt dit doen:

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

Git werkt iets anders als je expliciet `file://` aan het begin van de URL zet. Als je alleen het pad specificeert, probeert Git hardlinks te gebruiken naar de bestanden die het nodig heeft of direct de bestanden te kopiëren. Als je `file://` specificeert, dan start Git de processen die het normaal gesproken gebruikt om data uit te wisselen over een netwerk, wat over het algemeen een veel minder efficiënte methode is om gegevens over te dragen. De belangrijkste reden om `file://` wel te specificeren is als je een schone kopie van de repository wilt met de systeemvrije referenties of objecten eruit gelaten — over het algemeen na een import uit een ander versiebeheer systeem of iets dergelijks (zie [Git Binnenwerk](#) voor onderhoudstaken). We zullen het normale pad hier gebruiken, omdat het bijna altijd sneller is om het zo te doen.

Om een lokale repository aan een bestaand Git project toe te voegen, kun je iets als het volgende uitvoeren:

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

Daarna kan je op gelijke wijze pushen naar, en pullen van die remote via je nieuwe reomte genaamd `local_proj` zoals je over een netwerk zou doen.

## De voordelen

De voordelen van bestands-gebaseerde repositories zijn dat ze eenvoudig zijn en ze maken gebruik van bestaande bestandspermissies en netwerk toegang. Als je al een gedeeld bestandssysteem hebt waar het hele team al toegang toe heeft, dan is een repository opzetten heel gemakkelijk. Je stopt de kale repository ergens waar iedereen gedeelde toegang tot heeft, en stelt de lees- en schrijfrechten in zoals je dat bij iedere andere gedeelde directory zou doen. We zullen de manier om een kopie van een kale repository te exporteren voor dit doel bespreken in [Git op een server krijgen](#).

Dit is ook een prettige optie om snel werk uit een repository van iemand anders te pakken. Als jij en een collega aan hetzelfde project werken, en hij wil dat je iets bekijkt, dan is het uitvoeren van een commando zoals `git pull /home/john/project` vaak makkelijker dan wanneer hij naar een remote server moet pushen, en jij er van moet fetchen.

## De nadelen

Eén van de nadelen van deze methode is dat gedeelde toegang over het algemeen moeilijker op te zetten en te bereiken is vanaf meerdere locaties dan simpele netwerk toegang. Als je wilt pushen van je laptop als je thuis bent, dan moet je de remote schijf aankoppelen, wat moeilijk en langzaam

kan zijn in vergelijking met met netwerk gebaseerde toegang.

Het is ook belangrijk om te vermelden dat het niet altijd de snelste optie is als je een gedeeld koppelpunt (mount) of iets dergelijks gebruikt. Een lokale repository is alleen snel als je snelle toegang tot de data hebt. Een repository op NFS is vaak langzamer dan een repository via SSH op dezelfde server omdat dit laatste Git in staat stelt op lokale schijven te werken op elk van de betrokken systemen.

## De HTTP protocollen

Git kan op twee verschillende manieren via HTTP communiceren. Voor Git versie 1.6.6 was er maar een manier waarop dit kon en dat was erg basaal en over het algemeen kon je alleen lezen. In versie 1.6.6 en later, is een slimmere protocol geïntroduceerd waardoor Git in staat is gesteld om de gegevens-uitwisseling iets slimmer aan te pakken, op een manier die lijkt op de SSH aanpak. In de laatste paar jaar is dit nieuwe HTTP protocol erg populair geworden omdat het eenvoudiger is voor de gebruiker en slimmer in de manier waarop het communiceert. Aan deze nieuwere versie wordt vaak gerefereerd als het *slimme* HTTP protocol en het oude als het *domme* HTTP. We zullen eerst het nieuwere slimme HTTP protocol behandelen.

### Slimme HTTP

Het slimme HTTP protocol werkt ongeveer gelijk aan het SSH of Git protocol, maar verloopt via standaard HTTPS poorten en kan verscheidene HTTP authenticatie mechanismen gebruiken, wat betekent dat het vaak eenvoudiger is voor de gebruiker dan iets als SSH, omdat je zaken als basale gebruikersnaam/wachtwoord authenticatie kunt gebruiken in plaats van SSH sleutels in te richten.

Het is waarschijnlijk de meest populaire manier om Git te gebruiken geworden, omdat het zowel ingericht kan worden om anoniem gebruik te faciliteren zoals bij het `git://` protocol, maar ook met authenticatie en encryptie zoals bij het SSH protocol. In plaats van verschillende URLs op te zetten voor deze dingen, kan je nu een enkele URL voor beide gebruiken. Als je probeert te pushen en de repository heeft authenticatie nodig (wat normaalgesproken wel zou moeten), kan de server om gebruikernaam en wachtwoord vragen. Hetzelfde geldt voor lees-toegang.

Sterker nog, voor services als GitHub is de URL die je gebruikt om de repository online te bekijken (bijvoorbeeld, <https://github.com/schacon/simplegit>) is dezelfde URL die je kunt gebruiken om te clonen en, als je toegang hebt, om te pushen.

### Domme HTTP

Als de server niet werkt met een Git HTTP slimme service, zal de Git client terug proberen te vallen op het simpelere *domme* HTTP protocol. Het domme protocol verwacht dat de kale Git repository moet worden verspreid als gewone bestanden van de web server. Het mooie van het domme HTTP protocol is de eenvoud van het inrichten. Alles wat je hoeft te doen is een kale Git repository onder je HTTP document-root neer te zetten en een speciale `post-update` hook in te richten, en je bent klaar (Zie [Git Hooks](#)). Vanaf dat moment, kan iedereen die de webserver waarop je de repository hebt gezet kan bereiken ook je repository klonen. Om lees-toegang tot je repository toe te staan over HTTP, kan je iets als dit doen:

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

Dat is alles. De `post-update` hook dat standaard bij Git geleverd wordt activeert het juiste commando (`git update-server-info`) om HTTP fetching en cloning correct te laten werken. Dit commando wordt gedraaid als je naar deze repository pusht (misschien met SSH); en andere mensen kunnen clonen met zo iets als

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

In dit specifieke geval gebruiken we het `/var/www/htdocs` pad dat gebruikelijk is voor Apache setups, maar je kunt elke statische web server gebruiken — zet gewoon de kale repository in het pad. De Git data wordt verspreid als gewone statische bestanden (zie [Git Binnenwerk](#) voor details hoe precies het wordt verspreid).

Over het algemeen zou je kunnen kiezen om een lees/schrijf slimme HTTP server te draaien of om de bestanden beschikbaar te stellen als alleen lezen op de domme manier. Het is zeldzaam om een combinatie van beide services te draaien.

## De voordelen

Ze zullen ons voornamelijk richten op de voordelen van de slimme versie van het HTTP protocol.

De eenvoud van het hebben van een enkele URL voor alle typen van toegang en de server alleen te laten vragen om authenticatie wanneer het noodzakelijk is maakt het heel erg eenvoudig voor de eindgebruiker. Het kunnen authentificeren met gebruikersnaam en wachtwoord is ook een groot voordeel ten opzichte van SSH, omdat de gebruikers geen SSH sleutels lokaal hoeven te genereren en hun publieke sleutels niet naar de server hoeven te uploaden voordat ze in staat zijn met deze te communiceren. Voor minder kundige gebruikers, of gebruikers op systemen waar SSH minder gebruikelijk is, is dit een groot voordeel in bruikbaarheid. Het is ook een erg snel en efficiënt protocol, vergelijkbaar met SSH.

Je kunt ook je repositories met alleen leesrechten verspreiden middels HTTPS, wat inhoudt dat je de inhoud versleuteld verstuurt; of je kunt zelfs zover gaan dat je de clients speciaal getekende SSL certificaten laat gebruiken.

Een ander aardigheid is dat het HTTPS protocol zo gebruikelijk is dat de firewalls van bedrijven vaak zo zijn opgezet dat ze verkeer via deze poorten toestaan.

## De nadelen

Git via HTTPS kan op sommige servers iets moeilijker op te zetten zijn in vergelijking tot SSH. Anders dan dat, is er weinig dat andere protocollen in het voordeel laat zijn als we het hebben over het slimme HTTP protocol om Git inhoud te bedienen.

Als je HTTP gebruikt voor geauthenticeerde pushen, is het invullen van credentials (gebruikersnaam en wachtwoord) soms ingewikkelder dan het gebruik sleutels via SSH. Er zijn echter een aantal credential caching tools die je kunt gebruiken, waaronder Keychain toegang op macOS en Credential Manager op Windows om dit redelijk gladjes te laten verlopen. Lees [Het opslaan van inloggegevens](#) om te zien hoe veilige HTTP wachtwoord caching op jouw systeem op te zetten.

## Het SSH Protocol

Een gebruikelijk transport protocol voor Git in het geval van zelf-hosting is SSH. Dit is omdat SSH toegang tot servers in de meeste gevallen al ingericht is—en als dat niet het geval is, is het eenvoudig te doen. SSH is ook een authenticerend netwerk protocol, en omdat het alom aanwezig is, is het over het algemeen eenvoudig om in te richten en te gebruiken.

Om een Git repository via SSH te clonen, kan je een `ssh://` URL specificeren op deze manier:

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

Of je kunt het kortere scp-achtige syntax voor het SSH protocol gebruiken:

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

Als je in beide bovenstaande gevallen de optionele gebruikersnaam niet opgeeft, neemt Git de gebruikernaam over waarmee je op dat moment ingelogd bent.

### De voordelen

Er zijn vele voordelen om SSH te gebruiken. Ten eerste is SSH relatief eenvoudig op te zetten—SSH daemons komen veel voor, veel systeembeheerders hebben er ervaring mee, en veel operating systemen zijn er mee uitgerust of hebben applicaties om ze te beheren. Daarnaast is toegang via SSH veilig—all data transporten zijn versleuteld en geauthenticeerd. En als laatste is SSH efficiënt, net zoals bij het HTTPS, Git en lokale protocol worden de gegevens zo compact mogelijk gemaakt voordat het getransporteerd wordt.

### De nadelen

Het negatieve aspect van SSH is dat je er geen anonieme toegang naar je repository over kunt geven. Mensen *moeten* via SSH toegang hebben om er gebruik van te kunnen maken zelfs als het alleen lezen is, dit maakt dat SSH toegang niet echt bevordelijker is voor open source projecten. Als je het alleen binnen je bedrijfsnetwerk gebruikt, is SSH wellicht het enige protocol waar je mee in aanraking komt. Als je anonieme alleen-lezen toegang wilt toestaan tot je projecten, dan moet je SSH voor jezelf instellen om over te pushen, maar iets anders voor anderen om te fetchen.

## Het Git Protocol

Vervolgens is er het Git protocol. Dit is een speciale daemon dat met Git mee wordt geleverd, het luistert naar een toegewezen poort (9418) en biedt een service die vergelijkbaar is met het SSH

protocol, maar zonder enige vorm van authenticatie. Om een repository via het Git protocol te laten verspreiden, moet je eerst een `git-daemon-export-ok` bestand maken—de daemon zal een repository zonder dat bestand niet verspreiden, maar anders dan dat is er geen beveiliging. De Git repository is beschikbaar voor iedereen om te klonen of het is het voor niemand. Dit betekent dat er over het algemeen niet wordt gepusht met dit protocol. Je kunt push-toegang beschikbaar maken, maar gegeven het ontbreken van authenticatie kan iedereen op het internet die de vingers kan leggen op het URL van je project naar jouw project kan pushen. Laten we zeggen dat dit zelden voorkomt.

## De voordelen

Het Git protocol is vaak het snelste netwerk overdrachtsprotocol beschikbaar. Als je veel verkeer voor een publiek project moet bedienen of een erg groot project dat geen authenticatie behoeft voor lees-toegang, is het waarschijnlijk dat je een Git daemon wilt opzetten om je project te bedienen. Het gebruikt dezelfde data-transfer mechanisme als het SSH protocol, maar zonder de encryptie en authenticatie-overhead.

## De nadelen

Het nadeel van het Git protocol is het ontbreken van authenticatie. Het is over het algemeen niet wenselijk om alleen middels het Git protocol toegang te geven tot je project. Over het algemeen zal je dit koppelen met SSH of HTTPS toegang voor de enkele ontwikkelaars die push (schrijf)rechten hebben en voor alle anderen het `git://` protocol voor leestoegang. Daarbij is het waarschijnlijk het moeilijkste protocol om in te richten. Het moet zijn eigen daemon draaien, wat betekent dat de `xinetd` configuratie (of iets vergelijkbaars) wat niet het eenvoudigste is om te doen. Ook moet de firewall toegang tot poort 9418 worden opgezet, wat geen standaard poort is die in de firewalls van bedrijven wordt toegestaan. In de firewall van grote bedrijven is deze obscure poort doorgaans geblokkeerd.

# Git op een server krijgen

We gaan nu het inrichten van een Git service op je eigen server behandelen waarin deze protocols worden gebuikt.



We zullen hier de commando's en stappen laten zien om een eenvoudige, versimpelde installatie op een op Linux gebaseerde server op te zetten, alhoewel het ook mogelijk is deze services op een Mac of Windows server te draaien. Het daadwerkelijk opzetten van een productie server binnen jouw infrastructuur zal vrijwel zeker verschillen in de manier waarop de veiligheidsmaatregelen zijn ingericht of de gebruikte tooling van het besturingssysteem, maar hopelijk zal dit je een indruk geven van wat er allemaal bij komt kijken.

Om initieel een Git server op te zetten, moet je een bestaande repository naar een nieuwe kale repository exporteren—een repository die geen werk directory bevat. Dit is over het algemeen eenvoudig te doen. Om je repository clonen om daarmee een nieuwe kale repository te maken, draai je het clone commando met de `--bare` optie. De conventie is om directories waar kale repositories in staan te laten eindigen met `.git`, zoals hier:

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

Je zou nu een kopie van de Git directory gegevens in je `my_project.git` directory moeten hebben

Dit is grofweg gelijk aan

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

Er zijn een paar kleine verschillen in het configuratie bestand, maar het komt op hetzelfde neer. Het neemt de Git repository zelf, zonder een werkdirectory, en maakt een directory specifiek hiervoor aan.

## De kale repository op een server zetten

Nu je een kale kopie van je repository hebt, is het enige dat je moet doen het op een server zetten en je protocollen instellen. Laten we aannemen dat je een server ingericht hebt die `git.example.com` heet, waar je SSH toegang op hebt, en waar je al je Git repositories wilt opslaan onder de `/srv/git` directory. Aangenomen dat `/srv/git` bestaat op die server, kan je deze nieuwe repository beschikbaar stellen door je kale repository ernaartoe te kopiëren:

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

Vanaf dat moment kunnen andere gebruikers, die SSH toegang hebben tot dezelfde server en lees-toegang hebben tot de `/srv/git` directory, jouw repository clonen door dit uit te voeren:

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

Als een gebruiker met SSH op een server inlogt en schrijfttoegang heeft tot de `/srv/git/my_project.git` directory, dan hebben ze automatisch ook push toegang.

Git zal automatisch de correcte groep schrijfrechten aan een repository toekennen als je het `git init` commando met de `--shared` optie uitvoert.

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

Je ziet hoe eenvoudig het is om een Git repository te nemen, een kale versie aan te maken, en het op een server plaatsen waar jij en je medewerkers SSH toegang tot hebben. Nu zijn jullie klaar om aan hetzelfde project samen te werken.

Het is belangrijk om op te merken dat dit letterlijk alles is wat je moet doen om een bruikbare Git

server te draaien waarop meerdere mensen toegang hebben: maak alleen een paar accounts met SSH toegang aan op een server, en stop een kale repository ergens waar al die gebruikers lees- en schrijftoegang toe hebben. Je bent er klaar voor — je hebt niets anders nodig.

In de volgende paragrafen zul je zien hoe je meer ingewikkelde opstellingen kunt maken. Deze besprekking zal het niet hoeven aanmaken van gebruikers accounts voor elke gebruiker, publieke leestoegang tot repositories, grafische web interfaces en meer omvatten. Maar, hou in gedachten dat om samen te kunnen werken met mensen op een privé project, alles wat je *nodig* hebt een SSH server is en een kale repository.

## Kleine opstellingen

Als je met een kleine groep bent of net begint met Git in je organisatie en slechts een paar ontwikkelaars hebt, dan kunnen de dingen eenvoudig voor je zijn. Een van de meest gecompliceerde aspecten van een Git server instellen is het beheren van gebruikers. Als je sommige repositories alleen-lezen voor bepaalde gebruikers wilt hebben, en lees/schrijf voor anderen, dan kunnen toegang en permissies een beetje lastiger te regelen zijn.

### SSH toegang

Als je al een server hebt waar al je ontwikkelaars SSH toegang op hebben, dan is het over het algemeen het eenvoudigste om je eerste repository daar op te zetten, omdat je dan bijna niets hoeft te doen (zoals beschreven in de vorige paragraaf). Als je meer complexe toegangscontrole wilt op je repositories, dan kun je ze instellen met de normale bestandssysteem permissies van het operating systeem dat op je server draait.

Als je je repositories op een server wilt zetten, die geen accounts heeft voor iedereen in je team die je schrijftoegang wilt geven, dan moet je SSH toegang voor ze opzetten. We gaan er vanuit dat je een server hebt waarmee je dit kunt doen, dat je reeds een SSH server geïnstalleerd hebt, en dat de manier is waarop je toegang hebt tot de server.

Er zijn een paar manieren waarop je iedereen in je team toegang kunt geven. De eerste is voor iedereen accounts aanmaken, wat rechttoe rechtaan is maar bewerkelijk kan zijn. Je wilt vermoedelijk niet `adduser` uitvoeren en tijdelijke wachtwoorden instellen voor iedere gebruiker.

Een tweede methode is een enkele *git* gebruiker aan te maken op de machine, aan iedere gebruiker die schrijftoegang moet hebben vragen of ze je een publieke SSH sleutel sturen, en die sleutel toevoegen aan het `~/.ssh/authorized_keys` bestand van die nieuwe gebruiker. Vanaf dat moment zal iedereen toegang hebben op die machine via de *git* gebruiker. Dit tast de commit data op geen enkele manier aan — de SSH gebruiker waarmee je inlogt zal de commits die je opgeslagen hebt niet beïnvloeden.

Een andere manier waarop je het kunt doen is je SSH server laten authenticeren middels een LDAP server of een andere gecentraliseerde authenticatie bron, die misschien al ingericht is. Zolang iedere gebruiker shell toegang kan krijgen op de machine, zou ieder SSH authenticatie mechanisme dat je kunt bedenken moeten werken.

# Je publieke SSH sleutel genereren

Er zijn vele Git servers die authenticeren met een publieke SSH sleutel. Om een publieke sleutel te kunnen aanleveren, zal iedere gebruiker in je systeem er een moeten genereren als ze er nog geen hebben. Dit proces is bij alle operating systemen vergelijkbaar. Als eerste moet je controleren of je er niet al een hebt. Standaard staan de SSH sleutels van de gebruikers in hun eigen `~/.ssh` directory. Je kunt makkelijk nagaan of je al een sleutel hebt door naar die directory te gaan en de inhoud te bekijken:

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

Je bent op zoek naar een aantal bestanden genaamd als `id_dsa` of `id_rsa` met een bestand met gelijke naam en een `.pub` extensie. Het `.pub` bestand is je publieke sleutel en het andere bestand is je private sleutel. Als je deze bestanden niet hebt (of als je zelfs geen `.ssh` directory hebt), dan kun je ze aanmaken door een applicatie genaamd `ssh-keygen` uit te voeren, deze wordt met het SSH pakket op Linux/Mac systemen meegeleverd en met Git for Windows pakket:

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

Eerst wordt de lokatie waar je de sleutel wordt opgeslagen (`.ssh/id_rsa`) aangegeven, en vervolgens vraagt het tweemaal om een wachtwoord, die je leeg kunt laten als je geen wachtwoord wilt intypen op het moment dat je de sleutel gebruikt. Echter, als je echt een wachtwoord gebruikt, zorg ervoor dat je de `-o` optie toevoegt; daarmee bewaar je de private key in een formaat dat beter bestand is tegen *brute-force* wachtwoord kraken dan het standaard formaat. Je kunt ook de het `ssh-agent` tool gebruiken om te voorkomen dat je elke keer je wachtwoord moet intypen.

Iedere gebruiker die dit doet, moet zijn sleutel sturen naar jou of degene die de Git server beheert (aangenomen dat je een SSH server gebruikt die publieke sleutels vereist). Het enige dat ze hoeven doen is de inhoud van het `.pub` bestand kopiëren en e-mailen. De publieke sleutel ziet er ongeveer zo uit:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
GPl+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUFrv1QzM7xLELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyB1WXFCR+HAo3FXRitBqxiX1nKhXpHAZsMcilq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprrX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnPPI89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTLMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Voor een uitgebreidere tutorial over het aanmaken van een SSH sleutel op meerdere operating systemen, verwijzen we je naar de GitHub handleiding over SSH sleutels op <https://help.github.com/articles/generating-ssh-keys>.

## De server opzetten

Laten we het opzetten van SSH toegang aan de server kant eens doorlopen. In dit voorbeeld zul je de **authorized\_keys** methode gebruiken om je gebruikers te authenticeren. We gaan er ook vanuit dat je een standaard Linux distributie gebruikt zoals Ubuntu.



Het leeuwendeel van wat er hier beschreven is kan worden geautomatiseerd door het **ssh-copy-id**-commando, in plaats van het handmatig kopieren en installeren van publieke sleutels.

Als eerste maak je een **git** gebruiker aan en een **.ssh** directory voor die gebruiker.

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

Vervolgens moet je een aantal publieke SSH sleutels van ontwikkelaars aan het **authorized\_keys** bestand toevoegen voor de **git** gebruiker. Laten we aannemen dat je een aantal sleutels per e-mail ontvangen hebt en ze hebt opgeslagen in tijdelijke bestanden. Nogmaals, de publieke sleutels zien er ongeveer zo uit:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x41hJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAzuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1KKI9MAQLMdpgW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
07TCUSBdLQlgMVOfq1I2uPWQ0k0WQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgTZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Je voegt ze eenvoudigweg toe aan het **authorized\_keys** bestand van de **git** gebruiker in de **.ssh** directory:

```
$ 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
```

Nu kun je een lege repository voor ze instellen door `git init` uit te voeren met de `--bare` optie, wat de repository initialiseert zonder een werkdirectory:

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

Daarna kunnen John, Josie of Jessica de eerste versie van hun project in de repository pushen door het als een remote toe te voegen en een branch te pushen. Merk op dat iemand met een shell op de machine zal moeten inloggen en een kale repository moet creëren elke keer dat je een project wilt toevoegen. Laten we `gitserver` als hostnaam gebruiken voor de server waar je de `git` gebruiker en repository hebt aangemaakt. Als je het binnenshuis draait, en je de DNS instelt zodat `gitserver` naar die server wijst, dan kun je de commando's vrijwel ongewijzigd gebruiken (aangenomen dat `myproject` een bestaand project is met bestanden):

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

Vanaf dat moment kunnen de anderen het klonen en wijzigingen even gemakkelijk terug pushen:

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

Op deze manier kun je snel een lees/schrijf Git server draaiend krijgen voor een handjevol ontwikkelaars.

Je moet je ervan bewust zijn dat al deze gebruikers ook de server op kunnen en een shell als de `git` gebruiker kunnen krijgen. Als je dat wilt beperken, moet je de shell in iets anders veranderen in het `passwd` bestand.

Je kunt eenvoudig de `git` gebruiker beperken tot alleen Git activiteiten met een beperkte shell-tool

genaamd **git-shell** wat met Git wordt meegeleverd. Als je dit als login shell voor je **git** gebruiker instelt, dan kan de **git** gebruiker geen normale shell toegang hebben op je server. Specificeer **git-shell** in plaats van bash of csh voor je gebruikers login shell om dit te gebruiken. Om dit te doen moet je eerst **git-shell** aan **/etc/shells/** moeten toevoegen als dat al niet gebeurd is:

```
$ cat /etc/shells  # see if 'git-shell' is already in there. If not...
$ which git-shell  # make sure git-shell is installed on your system.
$ sudo vim /etc/shells  # and add the path to git-shell from last command
```

Nu kan je de shell wijzigen voor een gebruiker met **chsh <username> -s <shell>**:

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

Nu kan de **git** gebruiker alleen de SSH verbinding gebruiken om Git repositories te pullen en pushen en kan hij niet naar een shell op de machine gaan. Als je het probeert, zie je een login-weigering als dit:

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

De Git netwerk commando's zullen nog steeds blijven werken, maar gebruikers zullen niet in staat zijn een reguliere shell te starten. Zoals de uitvoer aangeeft, kan je ook een directory opgeven in de home directory van de **git** gebruiker die het **git-shell** commando een beetje aanpast. Als voorbeeld, je kunt de Git commando's die de server accepteert beperken of je kunt het bericht aanpassen dat de gebruikers zien als ze op deze manier met SSH verbinding maken. Roep **git help shell** aan voor meer informatie hoe de shell aanpassing te doen.

## Git Daemon

Nu gaan we een daemon opzetten die repositories verspreid via het “Git” protocol. Dit is een gebruikelijke keuze voor snelle, ongeauthenticeerde toegang tot je Git data. Onthoud dat omdat dit geen geauthenticeerde service is, alles wat je verspreid publiek is in het netwerk.

Als je deze server buiten je firewall draait, zou het alleen gebruikt moeten worden voor projecten die publiekelijk zichtbaar zijn naar de wereld. Als de server die je draait binnen je firewall staat, zou je het kunnen gebruiken voor projecten met een groot aantal mensen of computers (continue integratie of build servers) die alleen-lees-toegang hebben, waarvoor je niet een SSH sleutel wilt toevoegen voor elk van deze.

In elk geval, het Git protocol is relatief simpel om op te zetten. Het enige wat je hoeft te doen is dit commando in de achtergrond (daemonize) draaien:

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

De `--reuseaddr` optie staat de server toe om te herstarten zonder te wachten tot oude connecties een time-out krijgen, de `--base-path` optie staat mensen toe om projecten te clonen zonder het volledige pad te specificeren, en het pad aan het einde vertelt de Git daemon waar hij moet kijken voor de te exporteren repositories. Als je een firewall draait, zul je er poort 9418 open moeten zetten op de machine waar je dit op gaat doen.

Je kunt dit proces op een aantal manieren daemoniseren, afhankelijk van het besturingssysteem waarop je draait.

Omdat `systemd` het meest gebruikte init systeem is onder de moderne Linux distributies, kan je dat gebruiken voor dat doel. Simpelweg een bestand in `/etc/systemd/system/git-daemon.service` zetten met deze inhoud:

```
[Unit]
Description=Start Git Daemon

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

Restart=always
RestartSec=500ms

StandardOutput=syslog
StandardError=syslog
SyslogIdentifier=git-daemon

User=git
Group=git

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

Het zal je op zijn gevallen dat de Git daemon hier wordt gestart met `git` als zowel de group als user. Pas het aan aan jouw standaard en org ervoor dat de gebruiker bestaat op het systeem. Controleer ook dat de Git binary inderdaad op `/usr/bin/git` staat en wijzig het pad indien noodzakelijk.

Als laatste roep je `systemctl enable git-daemon` aan om de service automatisch te starten op boot-tijd, en je kunt de service starten en stoppen met respectievelijk `systemctl start git-daemon` en `systemctl stop git-daemon`.

Tot LTS 14.04, gebruikte Ubuntu upstart service unit configuraties. Daarom gebruik je op Ubuntu ≤ 14.04 een Upstart script. Dus, in het volgende bestand:

```
/etc/init/local-git-daemon.conf
```

zet je dit script:

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

Om veiligheidsredenen, wordt het sterk aangeraden om deze daemon te draaien als een gebruiker met alleen lees rechten op de repositories — je kunt dit simpelweg doen door een nieuwe gebruiker *git-ro* aan te maken en de daemon als deze te draaien. Om het simpel te houden draaien we het onderd de zelfde *git*-user waar de **git-shell** onder draait.

Als je de machine herstart, zal de Git daemon automatisch opstarten en herstarten als de server onderuit gaat. Om het te laten draaien zonder te herstarten, kun je dit uitvoeren:

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

Op andere systemen zul je misschien **xinetd** willen gebruiken, een script in je **sysvinit** systeem, of iets anders — zolang je dat commando maar ge-daemoniseerd krijgt en deze op een of andere manier in de gaten gehouden wordt.

Vervolgens zul je Git moeten vertellen op welke repositories je onauthenticeerde Gitserver gebaseerde toegang toestaat. Je kunt dit doen voor elke repository door een bestand genaamd **git-daemon-export-ok** te maken.

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

Het feit dat dit bestand er is geeft aan dat Git dat project zonder authenticatie kan verspreiden.

## Slimme HTTP

We hebben nu geauthenticeerde toegang via SSH en ongeauthenticeerde toegang met **git://**, maar er is een protocol die tot beide in staat is. Slimme HTTP opzetten is eigenlijk gewoon op de server een CGI script **git-http-backend** geheten activeren die met Git wordt geleverd. Deze CGI leest het pad en de headers die door een **git fetch** of een **git push** worden gestuurd aan een HTTP URL en bepaalt of de client via HTTP kan communiceren (wat elke client sinds versie 1.6.6 kan). Als de CGI ziet dat de client "slim" is, zal het op de slimme manier met deze communiceren, anders zal het terugvallen op het domme gedrag (dus het is "backward compatible" met lees acties van oudere clients).

Laten we een heel eenvoudige opzet doornemen. We zullen het gaan opzetten met Apache als de

CGI server. Als je geen Apache hebt geïnstalleerd, kan je dit op een Linux machine doen met iets wat lijkt op:

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

Dit zet `mod_cgi`, `mod_alias` en `mod_env` modules aan, die alle nodig zijn om dit goed te laten werken.

Je zult ook nog de Unix user groep van de `/srv/git` directories naar `www-data` te zetten zodat je webserver lees- en schrijfrechten heeft op de repositories, omdat de Apache instantie die de CGI script draait (standaard) draait onder die user:

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

Vervolgens moeten we een aantal dingen aan de Apache configuratie toevoegen om `git-http-backend` als afhandelaar te identificeren voor alles wat in het `/git` pad van je web server binnentkomt.

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

Als je de `GIT_HTTP_EXPORT_ALL` uit je omgevingsvariabele laat, zal Git alleen de repositories met het `git-daemon-export-ok` bestand erin verspreiden, net zoals de Git daemon deed.

Als laatste moet je Apache vertellen om verzoeken naar paden toe te staan die er zo uit zien, optioneel met een Auth block zoals hier:

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

Dat verplicht je een `.htaccess` bestand aan te maken met daarin de wachtwoorden van al de geldige gebruikers. Hier is een voorbeeld van hoe een “schacon” gebruiker toe te voegen aan het bestand:

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

Er zijn tig manieren om geauthenticeerde gebruikers in Apache aan te geven, je zult een keuze moeten maken en een van deze implementeren. Dit is gewoon een van de eenvoudigste voorbeelden die we konden verzinnen. Je zult dit waarschijnlijk ook over SSL willen opzetten,

zodat alle gegevens zijn versleuteld.

We willen ons niet te veel bochten wringen om de specifieke zaken van Apache configuraties uit de doeken te doen, je gebruikt misschien een andere server of andere authenticatie behoeften hebben. De clou is dat Git met een CGI geleverd wordt die **git-http-backend** heet die, wanneer geactiveerd, alle onderhandelingen doet teneinde bestanden te sturen en te ontvangen over HTTP. Deze implementeert het authenticeren zelf niet, maar dat kan eenvoudig worden beheerd op het niveau van de server die dit aanroeft. Je kunt dit met vrijwel elke web server met CGI capaciteiten, dus pas het vooral toe op hetgeen waar je het meest bekend mee bent.



Voor meer informatie over het configureren van authenticatie in Apache, verwijzen we je naar de volgende Apache documentatie: <http://httpd.apache.org/docs/current/howto/auth.html>

## GitWeb

Nu je gewone lees/schrijf en alleen-lezen toegang tot je project hebt, wil je misschien een eenvoudige web-gebaseerde visualisatie instellen. Git levert een CGI script genaamd GitWeb mee, dat soms hiervoor gebruikt wordt.

The screenshot shows the GitWeb interface for a repository. At the top, there's a navigation bar with links for 'summary', 'shortlog', 'log', 'commit', 'commitdiff', and 'tree'. On the right, there are buttons for 'commit', 'search', and a 're' button. Below the navigation, there's a 'description' field containing 'Unnamed repository; edit this file 'description' to name the repository.', an 'owner' field with 'Ben Straub', and a 'last change' timestamp. The main area is divided into sections: 'shortlog' (listing commits from June 2014), 'tags' (listing various versions from v0.11.0 to v0.21.0-rc1), and 'commitdiff' (a link to the commitdiff page). The 'shortlog' section includes details like author, date, commit message, and a list of files affected.

Figure 49. De GitWeb web-based gebruikers interface.

Als je wilt zien hoe GitWeb eruit ziet voor jouw project, kan je een commando wat met Git geleverd wordt gebruiken om een tijdelijke instantie op te starten als je een lichtgewicht server op je systeem hebt als **lighttpd** of **webrick**. Op Linux machines is **lighttpd** vaak geïnstalleerd, dus je zou in staat moeten zijn om het te laten lopen door **git instaweb** in te typen in je project directory. Als je een

Mac gebruikt, Leopard heeft Ruby voor-geïnstalleerd, zou **webrick** de beste gok kunnen zijn. Om **instaweb** te starten met een niet-lighttpd handler, kan je het aanroepen met de **--httpd** optie.

```
$ 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]
```

Daarmee wordt een HTTPD server op poort 1234 opgestart en daarna een webbrowser die opent op die pagina. Voor jou stelt dit niet veel voor. Als je klaar bent en je de server weer wilt afsluiten, kan je hetzelfde commando met de **--stop** optie aanroepen:

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

Als je de web interface permanent op een server wilt hebben draaien voor je team of voor een open source project die je host, moet je je reguliere web server inrichten om het CGI script te serveren. Sommige Linux distributies hebben een **gitweb** package dat je wellicht met **apt** of **yum** kunt installeren, wellicht kan je dat eerst proberen. We zullen spoedig het handmatig installeren van GitWeb bespreken. Eerst zal je de Git broncode, waar GitWeb mee geleverd wordt, moeten verkrijgen, en het volgende maatwerk CGI script genereren:

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

Merk op dat je het commando moet vertellen waar je Git repositories gevonden kunnen worden met de **GITWEB\_PROJECTROOT** variabele. Vervolgens moet je ervoor zorgen dat Apache CGI gebruikt voor dat script, daarvoor kan je een VirtualHost toevoegen:

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

Nogmaals: GitWeb kan worden geserveerd met elke web server die CGI of Perl ondersteunt, als je toch iets anders wilt gebruiken zou het niet al te moeilijk moeten zijn dit in te richten. Nu zou je in staat moeten zijn om <http://gitserver/> op te zoeken en je repositories online te zien.

## GitLab

GitWeb is echter nogal simplistisch. Als je op zoek bent naar een meer moderne Git server met alle toeters en bellen, zijn er een aantal open source oplossingen die je als alternatief kunt installeren. Omdat GitLab een van de meer populaire alternatieven is, zullen we het installeren en gebruiken als voorbeeld bespreken. Dit is iets complexer dan de GitWeb optie en vergt waarschijnlijk meer onderhoud, maar het is een optie met veel meer mogelijkheden.

### Installatie

GitLab is een applicatie die door een database wordt ondersteund, de installatie houdt daarom iets meer in dan andere Git servers. Gelukkig is dit proces zeer goed gedocumenteerd en ondersteund.

Er zijn een aantal methoden die je kunt volgen om GitLab te installeren. Om iets snel in de lucht te krijgen, kan je een image van een virtuele machine downloaden of een klik op de knop installatie programma van <https://bitnami.com/stack/gitlab>, en de configuratie wat bijstellen voor jouw specifieke omgeving. Bitnami heeft een prettig detail toegevoegd endat is het login scherm (bereikbaar door alt→ te typen); het geeft je het IP adres en standaard gebruikersnaam en wachtwoord voor de geïnstalleerde GitLab.

```

[----] [----] [----]
[----] [----] [----]

*** Welcome to the BitNami Gitlab Stack ***
*** Built using Ubuntu 12.04 - Kernel 3.2.0-53-virtual (tty2). ***

*** You can access the application at http://10.0.1.17 ***
*** The default username and password is 'user@example.com' and 'bitnami1'. ***
*** Please refer to http://wiki.bitnami.com/Virtual_Machines for details. ***

linux login: -

```

Figure 50. Het Bitnami GitLab virtual machine login scherm.

Voor de rest, volg de handleiding in het GitLab Community Edition readme-bestand, welke gevonden kan worden op <https://gitlab.com/gitlab-org/gitlab-ce/tree/master>. Daar vind je ondersteuning voor het installeren van GitLab gebruikmakend van Chef-recepten, een virtual machine op Digital Ocean en RPM en DEB-pakketten (die, ten tijde van schrijven, in beta zijn). Er is ook een “onofficieel” handleiding over hoe GitLab op een niet standaard besturingssysteem en database aan de praat te krijgen, een installatie-script voor het volledig handmatig installeren, en vele andere onderwerpen.

## Beheer

De beheer interface van GitLab is via het web benaderbaar. Simpelweg de hostnaam of IP adres waar GitLab is geïnstalleerd in je browser invullen, en inloggen als beheer-gebruiker. De standaard gebruikersnaam is `admin@local.host`, en het standaard wachtwoord is `5iveL!fie` (en je wordt er aan herinnerd om dit te wijzigen zodra je het intypt). Zodra je aangemeld bent, klik op het “Admin area” icoon in het menu rechts boven.

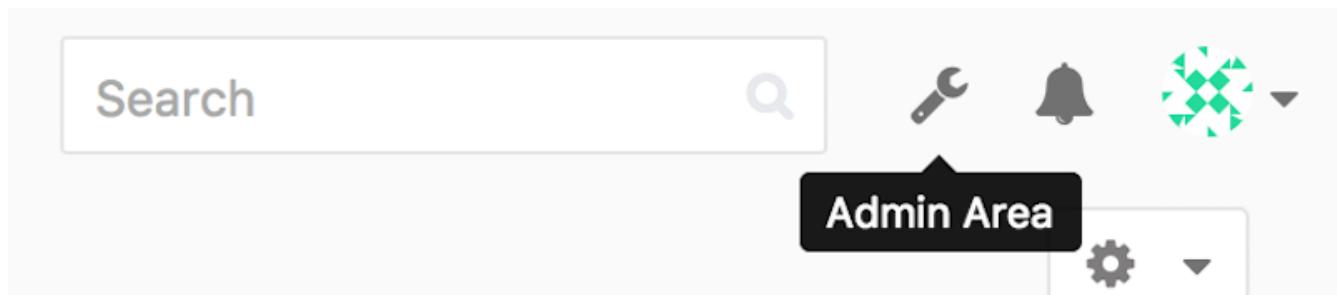
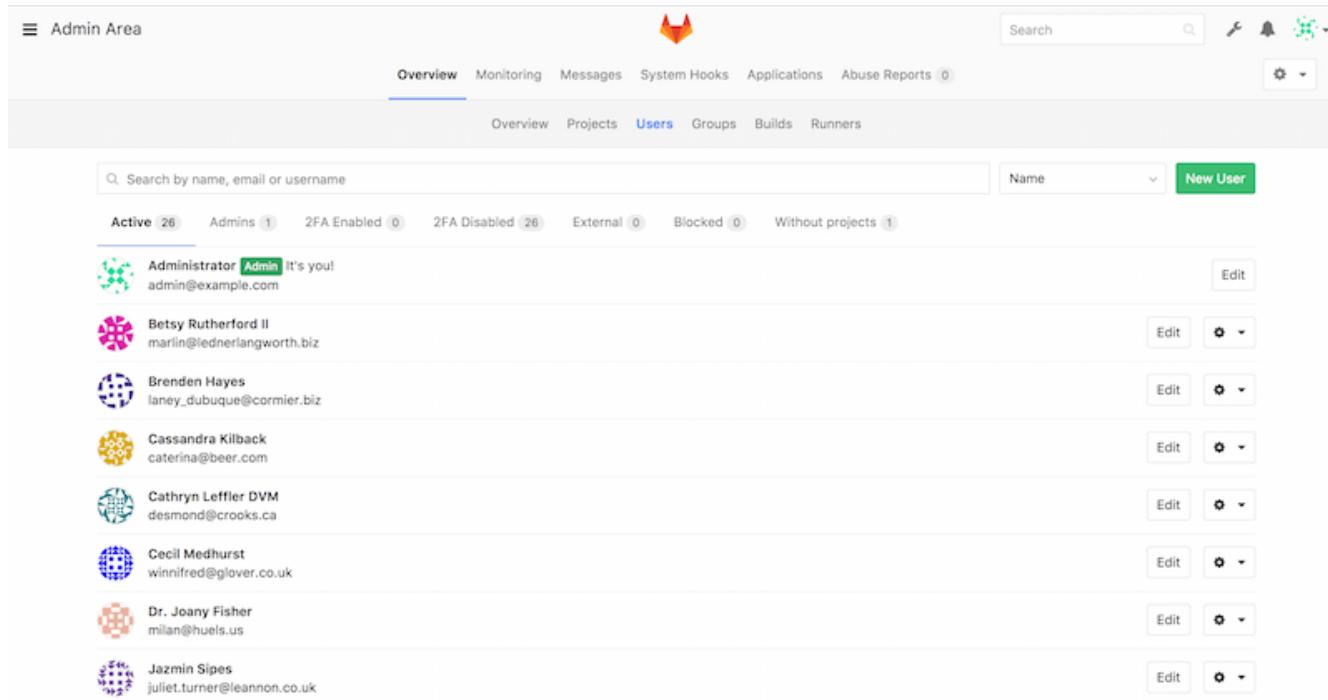


Figure 51. Het “Admin area” item in het GitLab menu.

## Gebruikers

Gebruikers in GitLab zijn accounts die overeenkomen met personen. Gebruiker accounts hebben niet veel complexiteit; het is voornamelijk een verzameling van persoonlijke gegevens die bij login-

gegevens horen. Elke gebruiker heeft een **namespace**, wat een logische groepering van projecten is die bij die gebruiker horen. Als de gebruiker **jane** een project genaamd **project** zou hebben, zou de URL van dat project <http://server/jane/project> zijn.



The screenshot shows the 'Users' section of the GitLab Admin Area. At the top, there are tabs for Overview, Monitoring, Messages, System Hooks, Applications, and Abuse Reports. Below that is a secondary navigation bar with links for Overview, Projects, Users (which is active), Groups, Builds, and Runners. A search bar and a 'New User' button are also present. The main area displays a list of users with their names, email addresses, roles (e.g., Admin, Developer), and profile icons. Each user entry includes an 'Edit' button and a dropdown menu. The user list includes:

- Administrator Admin It's you! admin@example.com
- Betsy Rutherford II marlin@ednnerlangworth.biz
- Brenden Hayes laney\_dubuque@cormier.biz
- Cassandra Kilback caterina@beer.com
- Cathryn Leffler DVM desmond@crooks.ca
- Cecil Medhurst winnifred@glover.co.uk
- Dr. Joany Fisher milan@huels.us
- Jazmin Sipes juliet.turner@leannon.co.uk

Figure 52. Het GitLab gebruiker beheer scherm.

Een gebruiker kan op twee manieren worden verwijderd. Een gebruiker “Blocken” (blokken) verhindert ze om in te loggen in deze GitLab instantie, maar alle gegevens onder de namespace van die gebruiker blijven intact, en commits met het email adres van die gebruiker zullen nog steeds naar die profiel terugverwijzen.

Een gebruiker “Destroyen” (vernietigen) echter verwijdert deze volledig van de database en het bestandssysteem. Alle projecten en gegevens in de namespace worden verwijderd, en alle groepen die met die gebruiker als eigenaar worden ook verwijderd. Dit is duidelijk een aktie met meer permanente en vernietigende gevolgen, en het wordt ook zelden gebruikt.

## Groepen

Een GitLab groep is een verzameling van projecten, samen met gegevens hoe gebruikers deze projecten kunnen benaderen. Elke groep heeft een project namespace (op gelijke manier waarop gebruikers dit hebben), dus als de groep **training** een project genaamd **materials** heeft, zou de url <http://server/training/materials> zijn.

The screenshot shows the GitLab.org group management interface for the '@gitlab-org' group. At the top, there's a navigation bar with links for Group, Activity, Labels, Milestones, Issues (8,501), Merge Requests (701), Members, and Contribution Analytics. Below the navigation is the group logo, which is a stylized orange fox head, followed by the handle '@gitlab-org'. A subtitle reads 'Open source software to collaborate on code'. There are buttons for 'Leave group' and 'Global'. The main area displays a list of projects under 'All Projects'. The projects listed are:

- GitLab Development Kit
- kubernetes-gitlab-demo
- omnibus-gitlab
- GitLab Enterprise Edition
- gitlab-shell
- gitlab-ci-multi-runner

Each project entry includes a small icon, a name, a brief description, and two circular icons on the right.

Figure 53. Het GitLab groep beheer scherm.

Elke groep heeft een relatie met een aantal gebruikers, elk van hen heeft een mate van permissies op de projecten van de groep en de groep zelf. Deze varieren van “Guest” (gast) (alleen problemen en chat) tot “Owner” (eigenaar) (volledig beheer over de groep, haar leden en projecten). De lijst van permissies is te groot om hier weer te geven, maar GitLab heeft een behulpzame link op het beheerscherm.

## Projecten

Een GitLab project komt grofweg overeen met een enkele Git repository. Elk project behoort tot één enkele namespace, ofwel een gebruiker of een groep. Als het project bij een gebruiker hoort, heeft de eigenaar van het project direct controle over wie toegang heeft tot het project; als het project tot een groep behoort, beginnen de permissies van de gebruikers binnen die groep ook een rol te spelen.

Elk project heeft een niveau van zichtbaarheid, welke bepaalt wie lees rechten tot de pagina's van het project en de repository heeft. Als een project *Private* is, moet de eigenaar van dat project expliciet toegang verlenen aan specifieke gebruikers. Als een project *Internal* (intern) is, is deze zichtbaar voor elke aangemelde gebruiker, en een *Public* project is zichtbaar voor iedereen. Let wel dat dit zowel de `git fetch` toegang als de toegang middels de web gebruikers interface voor dat project regelt.

## Hooks (haken)

GitLab ondersteunt het gebruik van hooks, zowel op een project als systeem niveau. Voor beiden zal de GitLab server een HTTP POST uitvoeren met wat beschrijvende JSON elke keer als er relevante gebeurtenissen plaatsvinden. Dit is een goede manier om je Git repositories en de GitLab instantie te verbinden met de rest van je ontwikkel-automatisering, zoals CI servers, chat rooms of deployment tools.

## Eenvoudig gebruik

Het eerste wat je binnen GitLab zult willen doen is het maken van een nieuw project. Dit wordt gedaan door het “+” icoon te klikken in de toolbar. Er zal worden gevraagd naar de naam van het project, welke namespace het toe behoort en wat het niveau van zichtbaarheid het moet hebben. Het meeste wat je hier aangeeft is niet permanent, het kan later veranderd worden middels het settings (instellingen) interface. Klik op ‘‘Create Project’’, en je bent klaar.

Als het project eenmaal bestaat, zal je het waarschijnlijk met een lokale Git repository willen verbinden. Elk project is toegangkelijk via HTTPS of SSH, beide kunnen worden gebruikt om ee Git remote op te zetten. De URLs zijn te zien aan de bovenkant van de thuis-pagina van het project. Voor een bestaande lokale repository zal dit commando een remote genaamd `gitlab` aanmaken naar de gehoste locatie:

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

Als je geen lokale kopie hebt van de repository, kan je simpelweg dit doen:

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

De web gebruikersinterface geeft toegang tot een aantal nuttige kijken op de repository. Elke thuispagina van een project laat de recente activiteit zien, en links aan de bovenkant zullen je naar overzichten voeren van de bestanden van het project en de commit log.

## Samenwerken

De eenvoudigste manier van samenwerken op een GitLab project is door een andere gebruiker direct push-toegang te geven tot de Git repository. Je kunt een gebruiker aan een project toevoegen door naar het “Members” (leden) gedeelte te gaan van de instellingen van dat project, en de nieuwe gebruiker een toegangsniveau toe te wijzen (de verschillende niveaus worden een beetje besproken in [Groepen](#)). Door een gebruiker een toegangsniveau van “Developer” of hoger te geven, kan die gebruiker commits pushen en straffeloos branches direct aan de repository toevoegen.

Een andere, meer ontkoppelde manier van samenwerken is door gebruik te maken van merge-requests (verzoeken tot samenvoeging). Deze mogelijkheid maakt het mogelijk dat elke gebruiker die het project kan zien eraan kan bijdragen op een meer beheerde manier. Gebruikers met directe toegang kunnen simpelweg een branch maken, hier commits naar pushen en een merge request openen om vanuit hun branch naar `master` of elke ander branch te mergen. Gebruikers die geen push-toestemming hebben voor een repository kunnen deze “forken” (hun eigen kopie maken), commits naar *die* kopie pushen, en een merge request openen vanuit hun fork terug naar het hoofdproject. Dit model stelt de eigenaar in staat om alles wat en wanneer er in de repository gebeurt volledig te beheersen, en tegelijkertijd bijdragen van niet vertrouwde gebruikers toe te staan.

Merge requests en issues (problemen) zijn de hoofdbestanddelen van langlopende discussies in GitLab. Elke merge request staat een regel-voor-regel discussie toe van de voorgestelde wijziging (wat de mogelijkheid opent voor een lichtgewicht code-review), alsook een generieke discussie

thread. Beide kunnen aan gebruikers worden toegewezen of gegroepeerd in mijlpalen.

Deze paragraaf is voornamelijk gericht op Git-gerelateerde mogelijkheden van GitLab; maar als een volwassen project biedt het vele andere mogelijkheden om je team samen te laten werken, zoals project wiki's en systeem onderhoudsinstrumenten. Een voordeel van GitLab is dat, zodra de server is ingericht en loopt, je nauwelijks nog aanpassingen aan de configuratie bestand hoeft te doen of de server via SSH moet benaderen. De meeste beheer en generieke gebruikshandelingen kunnen via de browser interface plaatsvinden.

## Hosting oplossingen van derden

Als je niet al het werk wilt verzetten wat samenhangt met het opzetten van je eigen Git server, heb je een aantal opties om je Git projecten te laten hosten op een site van een externe partij die zich hierop volledig heeft toegelegd. Als je dit doet geeft dit je een aantal voordelen: een hosting site is over het algemeen snel opgezet en het is eenvoudig om hierop projecten te beginnen, er komt geen server onderhoud of bewaking bij kijken. Zelfs als je je eigen interne server opzet en draait, wil je misschien toch een openbare hosting site voor je open source code gebruiken - het is over het algemeen makkelijker voor de open source gemeenschap om je te vinden en je ermee te helpen.

Vandaag de dag heb je een groot aantal hosting opties waaruit je kunt kiezen, elk met verschillende voor- en nadelen. Om een recentelijk bijgewerkte lijst te zien, kijk even op de GitHosting pagina van de hoofd Git wiki op <https://git.wiki.kernel.org/index.php/GitHosting>

We zullen het gebruik van GitHub tot in detail bespreken in [GitHub](#), daar dit zo'n beetje de grootste Git host is die er is en je waarschijnlijk toch in aanraking zult komen met projecten die daar gehost worden; maar er zijn nog tientallen waar je uit kunt kiezen als je niet je eigen Git server wilt inrichten.

## Samenvatting

Je hebt meerdere opties om een remote Git repository werkend te krijgen zodat je kunt samenwerken met anderen of je werk kunt delen.

Je eigen server draaien geeft je veel controle en stelt je in staat om de server binnen je firewall te draaien, maar zo'n server vraagt over het algemeen een behoorlijke hoeveelheid tijd om in te stellen en te onderhouden. Als je je gegevens op een beheerde server plaatst, is het eenvoudig in te stellen en te onderhouden; maar je moet wel willen dat je code op de server van een derde opgeslagen is, en sommige organisaties staan dit niet toe.

Het zou redelijk eenvoudig moeten zijn om te bepalen welke oplossing of combinatie van oplossingen van toepassing is op jou en jouw organisatie.

# Gedistribueerd Git

Nu je een remote Git repository hebt ingesteld als een plaats waar alle ontwikkelaars hun code kunnen delen, en je bekend bent met fundamentele Git commando's in een lokale workflow, zul je hier zien hoe je enkele van de gedistribueerde workflows kunt gebruiken waar Git je toe in staat stelt.

In dit hoofdstuk zul je zien hoe je met Git kunt werken in een gedistribueerde omgeving als een bijdrager (contributor) en als een integrator. Dat wil zeggen, je zult leren hoe je succesvol code kunt bijdragen aan een project en hoe je het zo makkelijk mogelijk maakt voor jou en de beheerder van het project, en ook hoe je een project succesvol kunt onderhouden waar een aantal ontwikkelaars aan bijdragen.

## Gedistribueerde workflows

In tegenstelling tot gecentraliseerde versiebeheersystemen (CVCSen), stelt de gedistribueerde aard van Git je in staat om veel flexibeler te zijn in de manier waarop ontwikkelaars samenwerken in projecten. Bij gecentraliseerde systemen is iedere ontwikkelaar een knooppunt dat min of meer gelijkwaardig werkt op een centraal punt. In Git echter is iedere ontwikkelaar zowel een knooppunt als een spil — dat wil zeggen, iedere ontwikkelaar kan zowel code bijdragen aan andere repositories, als ook een publiek repository beheren waarop andere ontwikkelaars hun werk baseren en waaraan zij kunnen bijdragen. Dit stelt je project en/of je team in staat om een enorm aantal workflows er op na te houden, dus ik zal een aantal veel voorkomende manieren behandelen die gebruik maken van deze flexibiliteit. We zullen de sterke en mogelijke zwakke punten van ieder ontwerp behandelen; je kunt er een kiezen om te gebruiken, of je kunt van iedere wijze een paar eigenschappen overnemen en mengen.

### Gecentraliseerde workflow

In gecentraliseerde systemen is er over het algemeen een enkel samenwerkingsmodel — de gecentraliseerde workflow. Eén centraal punt, of *repository*, kan code aanvaarden en iedereen synchroniseert zijn werk daarmee. Een aantal ontwikkelaars zijn knopen — gebruikers van dat centrale punt - en synchroniseren met die ene plaats.

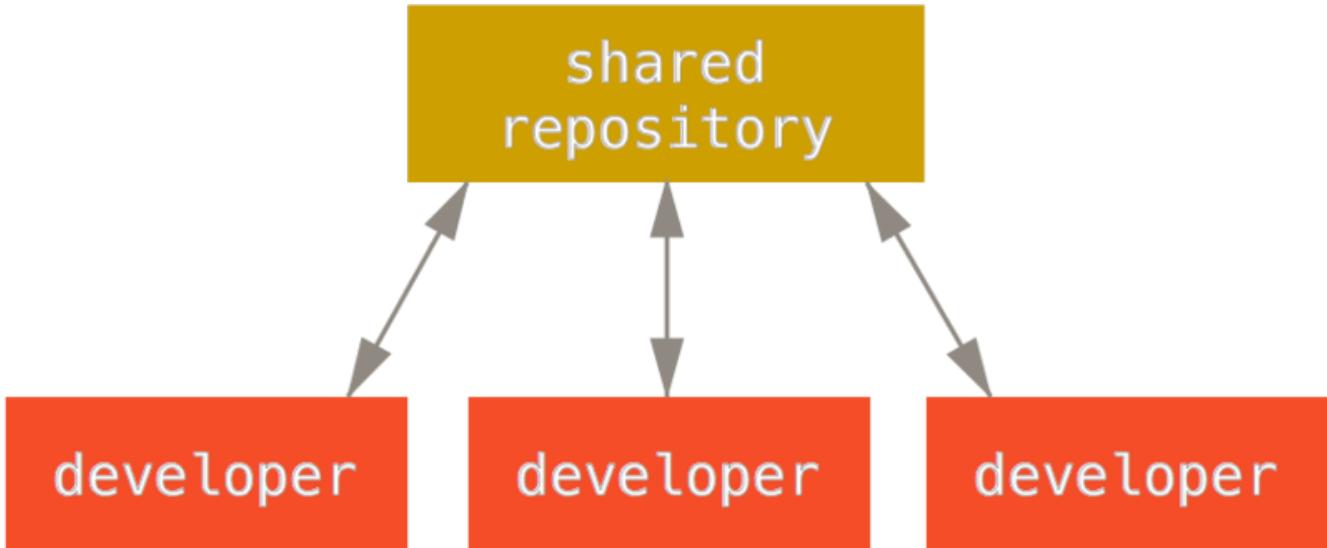


Figure 54. Gecentraliseerde workflow.

Dit houdt in dat als twee ontwikkelaars klonen van het gecentraliseerde punt en beiden wijzigingen doen, de eerste ontwikkelaar zijn wijzigingen zonder problemen kan pushen. De tweede ontwikkelaar zal het werk van de eerste in het zijne moeten mergen voordat hij het zijne kan pushen, om zo niet het werk van de eerste te overschrijven. Dit concept werkt in Git zoals het ook werkt in Subversion (of ieder ander CVCS), en dit model werkt prima in Git.

Als je al vertrouwd bent met een gecentraliseerde workflow in je bedrijf of team, dan kun je eenvoudigweg doorgaan met het gebruiken van die workflow met Git. Stel eenvoudigweg een enkele repository in, en geef iedereen in je team push-toegang; Git zal gebruikers niet toestaan om elkaar wijzigingen te overschrijven. Stel dat John en Jessica beiden tegelijkertijd beginnen te werken. John is klaar met zijn wijziging en pusht deze naar de server. Vervolgens probeert Jessica haar wijzigingen te pushen, maar de server weigert deze. Haar wordt verteld dat ze non-fast-forward wijzigingen probeert te pushen, en dat ze niet kan committen totdat ze gefetched en gemerged heeft. Deze workflow spreekt veel mensen aan omdat het een werkwijze is waarmee veel mensen bekend zijn en zich hierbij op hun gemak voelen.

Deze workflow is echter niet beperkt tot alleen kleine teams. Met het branching model van Git is het mogelijk om honderden ontwikkelaars tegelijkertijd succesvol te laten werken op een enkel project middels tientallen branches.

## Integratie-manager workflow

Omdat Git je toestaat om meerdere remote repositories te hebben, is het mogelijk om een workflow te hebben waarbij iedere ontwikkelaar schrijfttoegang heeft tot zijn eigen publieke repository en leestoegang op die van de anderen. In dit scenario is er vaak een gezagdragend (canonical) repository dat het “officiële” project vertegenwoordigt. Om bij te kunnen dragen tot dat project, maak je je eigen publieke kloon van het project en pusht je wijzigingen daarin terug. Daarna stuur je een verzoek naar de eigenaar van het hoofdproject om jouw wijzigingen binnen te halen. Hij kan je repository toevoegen als een remote, je wijzigingen lokaal testen, ze in zijn branch mergen, en dan naar zijn repository pushen. Het proces werkt als volgt (zie [Integratie-manager workflow](#)):

1. De projecteigenaar pusht naar de publieke repository.

2. Een bijdrager kloont die repository en maakt wijzigingen.
3. De bijdrager pusht naar zijn eigen publieke kopie.
4. De bijdrager stuurt de eigenaar een e-mail met de vraag om de wijzigingen binnen te halen.
5. De eigenaar voegt de repo van de bijdrager toe als een remote en merget lokaal.
6. De eigenaar pusht de gemergde wijzigingen terug in de hoofdrepository.

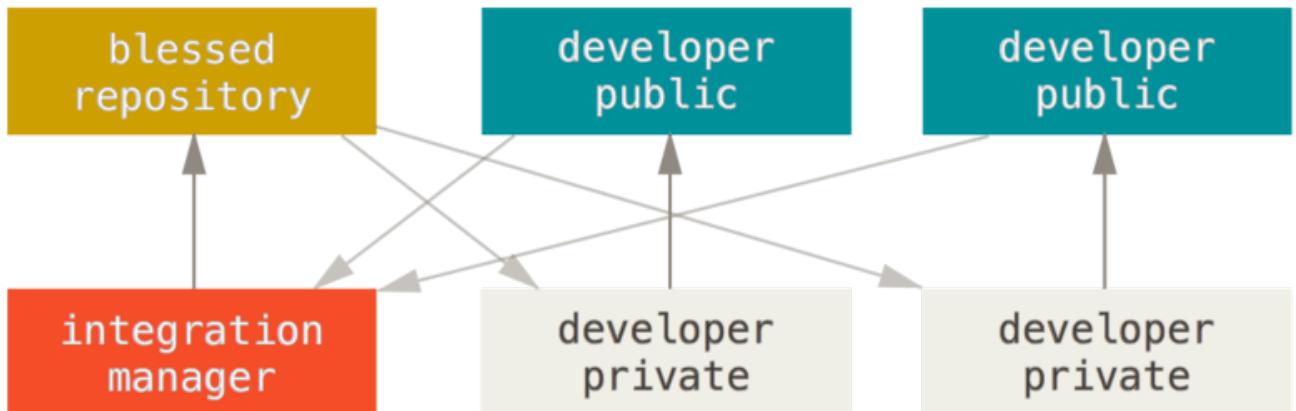


Figure 55. Integratie-manager workflow.

Dit is een veel voorkomende workflow bij websites zoals GitHub of GitLab, waarbij het eenvoudig is om een project af te splitsen (fork) en je wijzigingen te pushen in jouw afgesplitste project waar iedereen ze kan zien. Een van de grote voordelen van deze aanpak is dat je door kunt gaan met werken, en de eigenaar van de hoofdrepository jouw wijzigingen op ieder moment kan binnengenomen. Bijdragers hoeven niet te wachten tot het project hun bijdragen invoegt—iedere partij kan in zijn eigen tempo werken.

## Dictator en luitenanten workflow

Dit is een variant op de multi-repository workflow. Het wordt over het algemeen gebruikt bij enorme grote projecten met honderden bijdragers; een bekend voorbeeld is de Linux-kernel. Een aantal integrators hebben de leiding over bepaalde delen van de repository, zij worden *luitenanten* genoemd. Alle luitenanten hebben één integrator die bekend staat als de welwillende dictator (benevolent dictator). De repository van de welwillende dictator dient als het referentie-repository vanwaar alle bijdragers dienen te pullen. Het proces werkt als volgt (zie [Benevolent dictator workflow](#)):

1. Reguliere ontwikkelaars werken op hun eigen onderwerp (topic) branch en rebasen hun werk op de `master`. De `master`-branch is die van de referentie-repository waar de dictator naar toe pusht.
2. Luitenanten mergen de topic branches van de ontwikkelaars in hun `master`-branch.
3. De dictator merged de `master`-branches van de luitenanten in de `master`-branch van de dictator.
4. De dictator pusht zijn `master`-branch ten slotte terug naar het referentie-repository zodat de andere ontwikkelaars op deze kunnen rebasen.

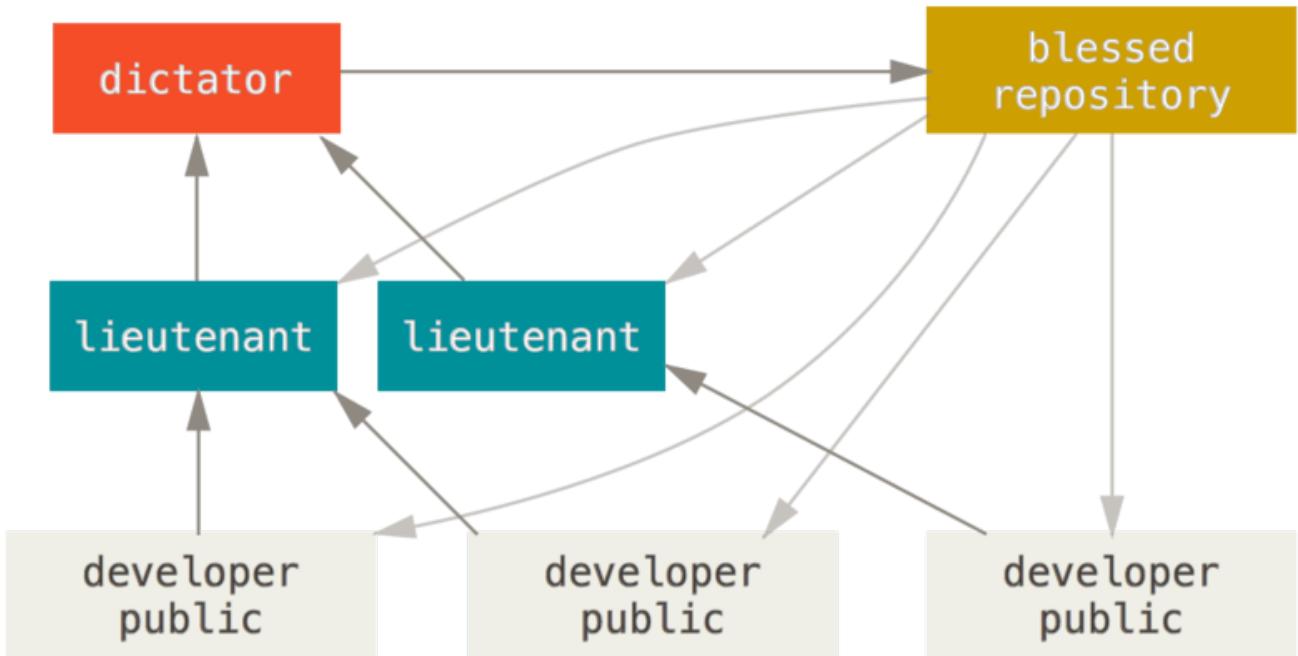


Figure 56. Benevolent dictator workflow.

Deze manier van werken is niet gewoon, maar kan handig zijn in hele grote projecten of in zeer hiërarchische omgevingen. Het stelt de projectleider (de dictator) in staat het meeste werk te delegeren en grote subsets van code te verzamelen op meerdere punten alvorens ze te integreren.

## Workflows samenvatting

Dit zijn een aantal veel voorkomende workflows die mogelijk zijn met een gedistribueerd systeem als Git, maar je kunt zien dat er veel variaties mogelijk zijn om ze te laten passen bij jouw specifieke workflow. Nu dat je (naar we hopen) in staat bent om te bepalen welke combinatie van workflows voor jou werkt, zullen we wat specifiekere voorbeelden behandelen hoe je de belangrijkste rollen kunt vervullen die in de verschillende workflows voorkomen. In de volgende paragraaf zal je kennis maken met een aantal reguliere patronen voor het bijdragen aan een project.

## Bijdragen aan een project

De grote moeilijkheid bij het beschrijven van dit proces is dat er een enorm aantal variaties mogelijk zijn in hoe het gebeurt. Om dat Git erg flexibel is, kunnen en zullen mensen op vele manieren samenwerken, en het is lastig om te beschrijven hoe je zou moeten bijdragen aan een project — ieder project is net weer een beetje anders. Een aantal van de betrokken variabelen zijn het aantal actieve bijdragers, gekozen workflow, je commit toegang, en mogelijk de manier waarop externe bijdragen worden gedaan.

De eerste variabele is het aantal actieve bijdragers. Hoeveel gebruikers dragen actief code bij aan dit project, en hoe vaak? In veel gevallen zal je twee of drie ontwikkelaars met een paar commits per dag hebben, of misschien minder voor wat meer slapende projecten. Voor zeer grote bedrijven of projecten kan het aantal ontwikkelaars in de duizenden lopen, met tientallen of zelfs honderden patches die iedere dag binnenkomen. Dit is belangrijk omdat met meer en meer ontwikkelaars, je meer en meer problemen tegenkomt bij het je verzekeren dat code netjes gepatched of eenvoudig gemerged kan worden. Wijzigingen die je indient kunnen verouderd of zwaar beschadigd raken

door werk dat gemerged is terwijl je ermee aan het werken was, of terwijl je wijzigingen in de wacht stonden voor goedkeuring of toepassing. Hoe kun je jouw code consequent up-to-date en je patches geldig houden?

De volgende variabele is de gebruikte workflow in het project. Is het gecentraliseerd, waarbij iedere ontwikkelaar gelijkwaardige schrijftoegang heeft tot de hoofd codebasis? Heeft het project een eigenaar of integrator die alle patches controleert? Worden alle patches ge(peer)reviewed en goedgekeurd? Ben jij betrokken bij dat proces? Is er een luitenanten systeem neergezet, en moet je je werk eerst bij hen inleveren?

De volgende variabele is je commit toegang. De benodigde workflow om bij te dragen aan een project is heel anders als je schrijftoegang hebt tot het project dan wanneer je dat niet hebt. Als je geen schrijftoegang hebt, wat is de voorkeur van het project om bijdragen te ontvangen? Is er überhaupt een beleid? Hoeveel werk draag je per keer bij? Hoe vaak draag je bij?

Al deze vragen kunnen van invloed zijn op hoe je effectief bijdraagt aan een project en welke workflows de voorkeur hebben of beschikbaar zijn. We zullen een aantal van deze aspecten behandelen in een aantal voorbeelden, waarbij we van eenvoudig tot complexzullen gaan. Je zou in staat moeten zijn om de specifieke workflows die je in jouw praktijk nodig hebt te kunnen herleiden naar deze voorbeelden.

## Commit richtlijnen

Voordat we gaan kijken naar de specifieke gebruiksscenario's, volgt hier een kort stukje over commit berichten. Het hebben van een goede richtlijn voor het maken commits en je daar aan houden maakt het werken met Git en samenwerken met anderen een stuk makkelijker. Het Git project levert een document waarin een aantal goede tips staan voor het maken van commits waaruit je patches kunt indienen—je kunt het lezen in de Git broncode in het [Documentation/SubmittingPatches](#) bestand.

Als eerste wil je geen witruimte-fouten indienen. Git geeft je een eenvoudige manier om hierop te controleren—voordat je commit, voer `git diff --check` uit, wat mogelijke witruimte-fouten identificeert en ze voor je afdrukt.

```
lib/simplegit.rb:5: trailing whitespace.
+ @git_dir = File.expand_path(git_dir)
lib/simplegit.rb:7: trailing whitespace.
+
lib/simplegit.rb:20: trailing whitespace.
+end
(END)
```

Figure 57. Uitvoer van `git diff --check`.

Als je dat commando uitvoert alvorens te committen, kun je al zien of je op het punt staat witruimte problemen te committen waaraan andere ontwikkelaars zich zouden kunnen ergeren.

Probeer vervolgens om van elke commit een logische set wijzigingen te maken. Probeer, als het je lukt, om je wijzigingen verterbaar te houden — ga niet het hele weekend zitten coderen op vijf verschillende problemen om dat vervolgens op maandag als één gigantische commit in te dienen. Zelfs als je gedurende het weekend niet commit, gebruik dan het staging gebied op maandag om je werk in ten minste één commit per probleem op te splitsen, met een bruikbaar bericht per commit. Als een paar van de wijzigingen hetzelfde bestand betreffen, probeer dan `git add --patch` te gebruiken om bestanden gedeeltelijk te staggen (in detail behandeld in [Interactief staggen](#)). De snapshot aan de kop van het project is gelijk of je nu één commit doet of vijf, zolang alle wijzigingen op een gegeven moment maar toegevoegd zijn, probeer dus om het je medeontwikkelaars makkelijk te maken als ze je wijzigingen moeten beoordelen.

Deze aanpak maakt het ook makkelijker om één wijziging eruit te selecteren of terug te draaien, mocht dat later nodig zijn. [Geschiedenis herschrijven](#) beschrijft een aantal handige Git trucs om geschiedenis te herschrijven en bestanden interactief te staggen — gebruik deze instrumenten als hulp om een schone en begrijpelijke historie op te bouwen voordat deze naar iemand anders wordt gestuurd.

Het laatste om in gedachten te houden is het commit bericht. Als je er een gewoonte van maakt om commit berichten van goede kwaliteit aan te maken, dan maakt dat het gebruik van en samenwerken in Git een stuk eenvoudiger. In het algemeen zouden je berichten moeten beginnen met een enkele regel die niet langer is dan 50 karakters en die de wijzigingen beknopt omschrijft, gevolgd door een lege regel en daarna een meer gedetailleerde uitleg. Het Git project vereist dat de meer gedetailleerde omschrijving ook je motivatie voor de verandering bevat, en de nieuwe implementatie tegen het oude gedrag afzet — dit is een goede richtlijn om te volgen. Het is ook een goed idee om de gebiedende wijs te gebruiken in deze berichten. Met andere woorden, gebruik commando's. In plaats van "Ik heb testen toegevoegd voor" of "Testen toegevoegd voor" gebruik je "Voeg testen toe voor". Hier is een sjabloon dat origineel geschreven is door Tim Pope:

Short (50 chars or less) summary of changes

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

Further paragraphs come after blank lines.

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

Als al je commit berichten er zo uit zien, dan zullen de dingen een stuk eenvoudiger zijn voor jou en de ontwikkelaars waar je mee samenwerkt. Het Git project heeft goed geformatteerde commit berichten - ik raad je aan om `git log --no-merges` uit te voeren om te zien hoe een goed geformatteerde project-commit historie eruit ziet.

*Doe wat wij zeggen, niet wat wij doen.*



In de volgende voorbeelden en verder door de rest van dit boek, zijn omwille van bondigheid, de berichten niet zo netjes geformateerd als dit; in plaats daarvan gebruiken we de `-m` optie voor `git commit`.

Kortom: doe wat wij zeggen, niet wat wij doen.

## Besloten klein team

De eenvoudigste opzet die je waarschijnlijk zult tegenkomen is een besloten project met één of twee andere ontwikkelaars. Met “besloten” bedoel ik gesloten broncode — zonder leestoegang voor de buitenwereld. Jij en de andere ontwikkelaars hebben allemaal push toegang op de repository.

In deze omgeving kan je een workflow aanhouden die vergelijkbaar is met wat je zou doen als je Subversion of een andere gecentraliseerd systeem zou gebruiken. Je hebt nog steeds de voordelen van zaken als offline committen en veel eenvoudiger branchen en mergen, maar de workflow kan erg vergelijkbaar zijn. Het grootste verschil is dat het mergen aan de client-kant gebeurt en niet tijdens het committen aan de server-kant. Laten we eens kijken hoe het er uit zou kunnen zien als twee ontwikkelaars samen beginnen te werken met een gedeelde repository. De eerste ontwikkelaar, John, kloont de repository, maakt een wijziging, en commit lokaal. (De protocol berichten zijn met `...` vervangen in deze voorbeelden om ze iets in te korten.)

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

De tweede ontwikkelaar, Jessica, doet hetzelfde — kloont de repository en commit een wijziging:

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

Nu pusht Jessica haar werk naar de server, en dat werkt prima:

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

De laatste regel van de uitvoer hierboven laat een bruikbaar bericht zien van de push-operatie. Het basis-formaat is `<oldref>..<newref> fromref -> toref`, waar `oldref` de oude reference inhoudt, `newref` betekent de nieuwe referentie, `fromref` is de naam van de lokale referentie die gepusht wordt, en `toref` is de naam van de remote referentie die wordt geupdate. Je zult een vergelijkbare uitvoer zien in de behandeling hieronder, dus een basis inzicht hebben in de betekenis zal helpen met het begrijpen van de verschillende stadia van de repositories. Meer details zijn beschikbaar in de documentatie van [git-push](#).

We gaan door met het voorbeeld, John maakt wat wijzigingen, commit deze naar zijn lokale repository en probeert ze naar dezelfde server te pushen:

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

In dit geval zal de push van John falen, vanwege de eerdere push van Jessica met *haar wijzigingen*. Dit is belangrijk om te begrijpen als je gewend bent aan Subversion, omdat het je zal opvallen dat de twee ontwikkelaars niet hetzelfde bestand hebben aangepast. Waar Subversion automatisch zo'n merge op de server doet als verschillende bestanden zijn aangepast, moet je in Git de commits *eerst* lokaal mergen. Met andere woorden: John moet eerst Jessica's wijzigingen ophalen en ze in zijn lokale repository mergen voor hij mag pushen:

Als eerste stap, gaat John Jessica's werk eerst fetchen (dit zal alleen Jessica's werk *fetchen*, het zal het nog niet in John's werk mergen):

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

Hierna ziet de lokale repository van John er ongeveer zo uit:

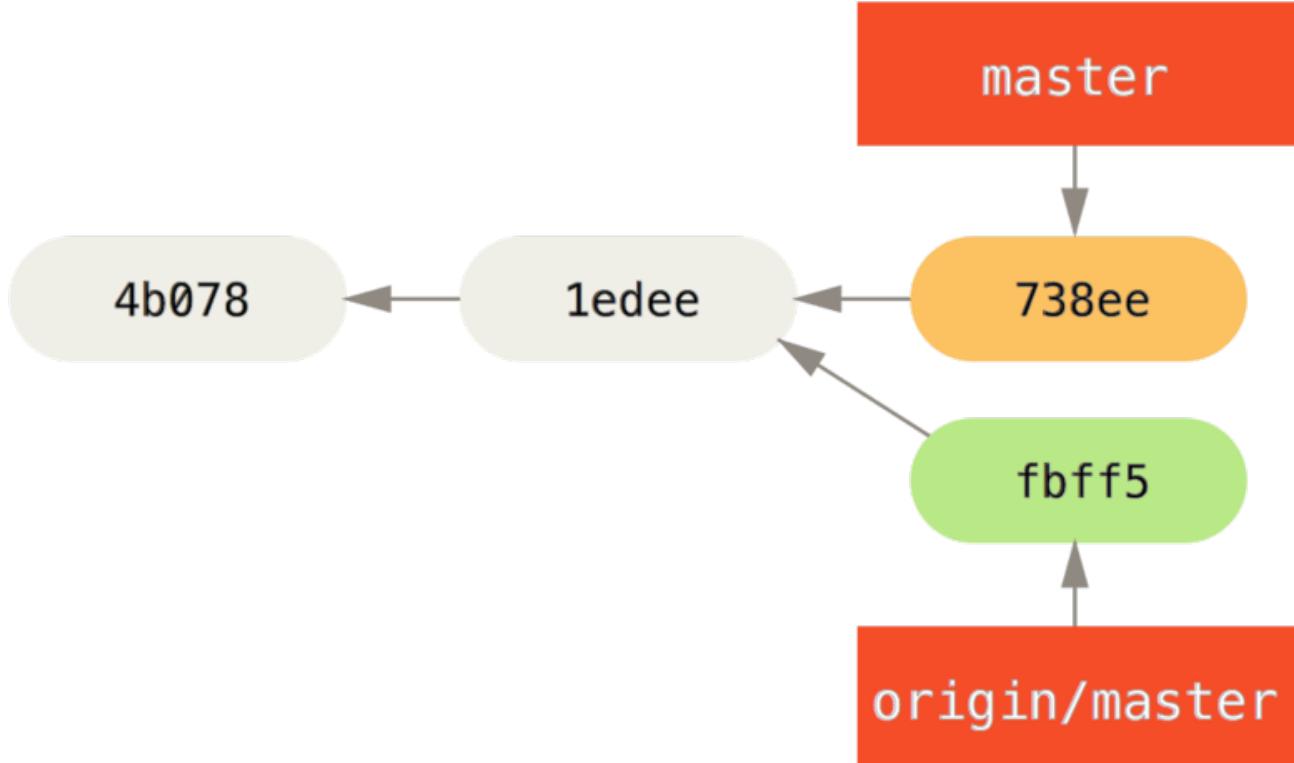


Figure 58. De afwijkende historie van John.

Nu kan John het werk van Jessica dat hij heeft gefetcht gaan mergen in zijn lokale werk.

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

Zo lang als de lokale merge gladjes verloopt, zal John's geupdate history er zo uit zien:

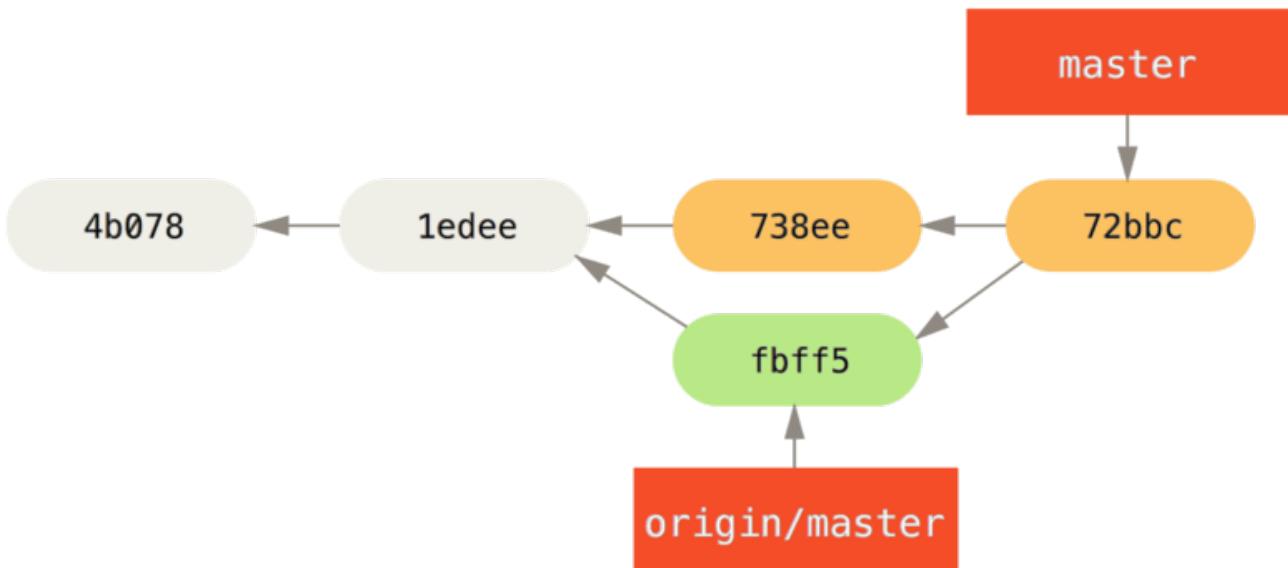


Figure 59. De repository van John na `origin/master` te hebben gemerged.

Nu zou John zijn code moeten testen om er zeker van te zijn dat alles nog steeds goed werkt, en dan kan hij zijn nieuwe gemergede werk pushen naar de server:

```
$ git push origin master
...
To john@githost:simplegit.git
fbfff5bc..72bbc59  master -> master
```

Uiteindelijk ziet de commit historie van John er zo uit:

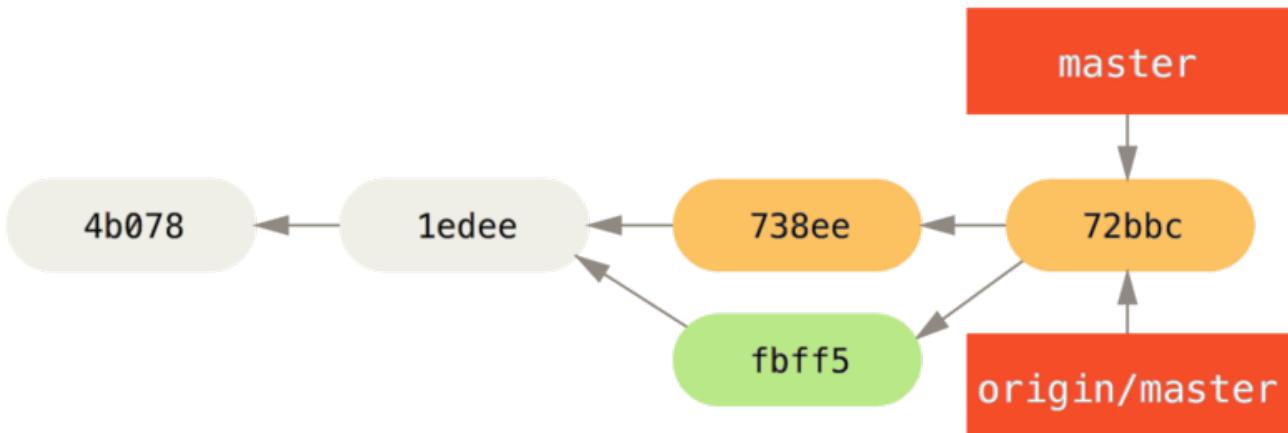


Figure 60. De historie van John na te hebben gepusht naar de `origin` server.

In de tussentijd heeft Jessica een topic branch genaamd `issue54` aangemaakt en daar drie commits op gedaan. Ze heeft John's wijzigingen nog niet opgehaald, dus haar commit historie ziet er zo uit:

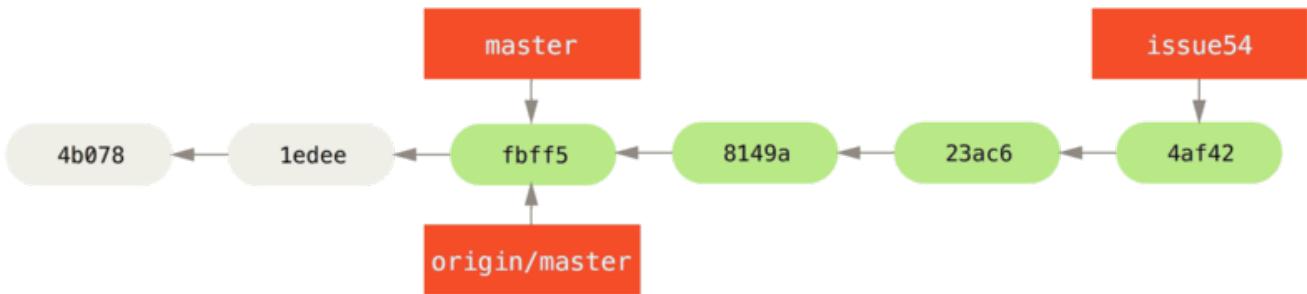


Figure 61. Topic branch van Jessica.

Ineens hoort Jessica dat John nieuw werk heeft gepusht op de server en ze wil dit even gaan bekijken, zodat ze alle inhoud van de server kan ophalen dat ze nog niet heeft met:

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

Dit haalt het werk op dat John in de tussentijd gepusht heeft. De historie van Jessica ziet er nu zo uit:

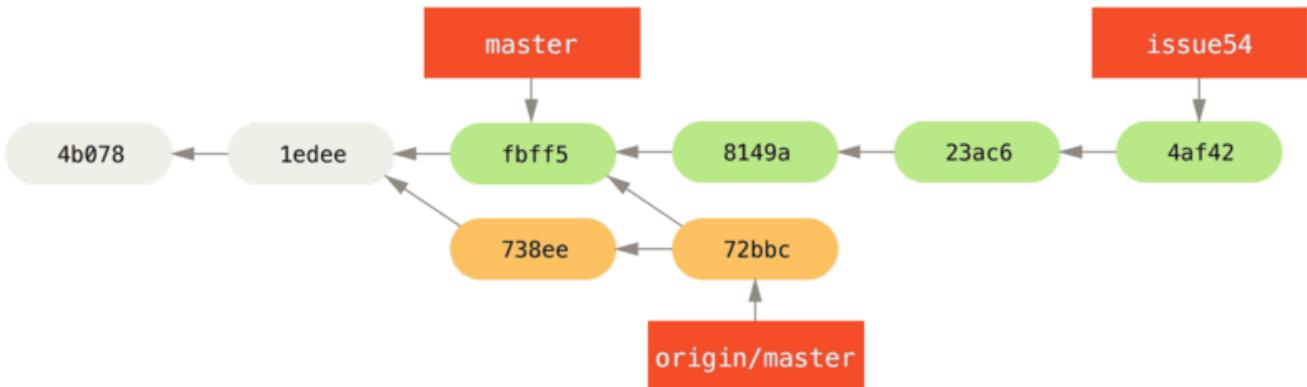


Figure 62. Historie van Jessica na het fetchen van de wijzigingen van John.

Jessica denkt dat haar topic branch nu klaar is, maar ze wil weten wat ze in haar werk moet mergen zodat ze kan pushen. Ze voert `git log` uit om dat uit te zoeken:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    removed invalid default value
```

De `issue54..origin/master` syntax is een log filter dat Git vraagt om alleen de lijst van commits te

tonen die op de laatstgenoemde branch (in dit geval `origin/master`) staan die niet in de eerstgenoemde (in dit geval `issue54`) staan. We zullen deze syntax in detail bespreken in [Commit reeksen](#).

Nu zien we in de uitvoer dat er een commit is die John gemaakt heeft die Jessica nog niet gemerged heeft. Als ze `origin/master` merged, is dat de enige commit die haar lokale werk zal wijzigen.

Nu kan Jessica het werk van haar topic branch mergen in haar master branch, het werk van John (`origin/master`) in haar `master`-branch mergen, en dan naar de server pushen.

Eerst (nadat ze al haar werk op de `issue54` topic branch heeft gecommit) schakelt Jessica terug naar haar master branch als voorbereiding op het integreren van al dit werk:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

Ze kan `origin/master` of `issue54` als eerste mergen — ze zijn beide stroomopwaarts dus de volgorde maakt niet uit. Uiteindelijk zou de snapshot gelijk moeten zijn ongeacht welke volgorde ze kiest; alleen de geschiedenis zal iets verschillen. Ze kiest ervoor om `issue54` eerst te mergen:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README          |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Er doen zich geen problemen voor, zoals je kunt zien was het een eenvoudige fast-forward. Jessica voltooit het lokale merge proces met het mergen van John's eerder opgehaalde werk die in de `origin/master`-branch zit:

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
 lib/simplegit.rb |    2 ++
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Alles merget netjes, en de historie van Jessica ziet er nu zo uit:

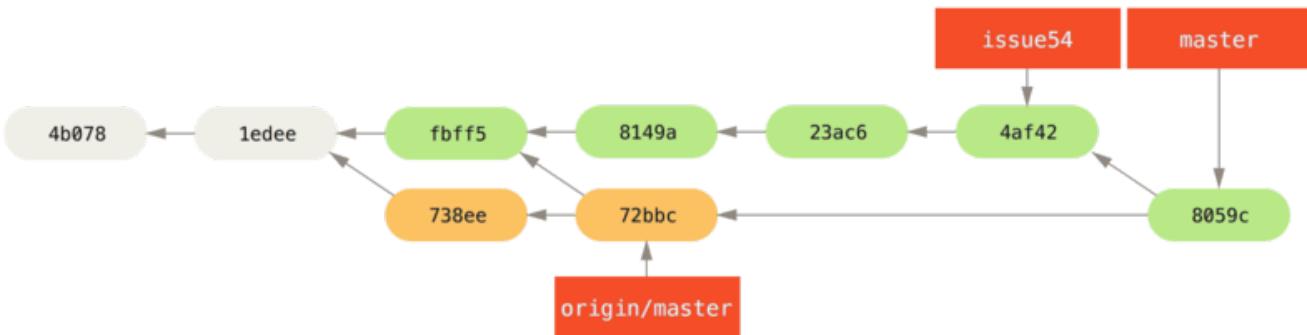


Figure 63. Historie van Jessica na het mergen van de wijzigingen van John.

Nu is `origin/master` bereikbaar vanuit Jessica's `master`-branch, dus ze zou in staat moeten zijn om succesvol te pushen (even aangenomen dat John in de tussentijd niet weer iets gepusht heeft):

```
$ git push origin master
...
To jessica@githost:simplegit.git
  72bbc59..8059c15  master -> master
```

Iedere ontwikkelaar heeft een paar keer gecommit en elkaars werk succesvol gemerged.

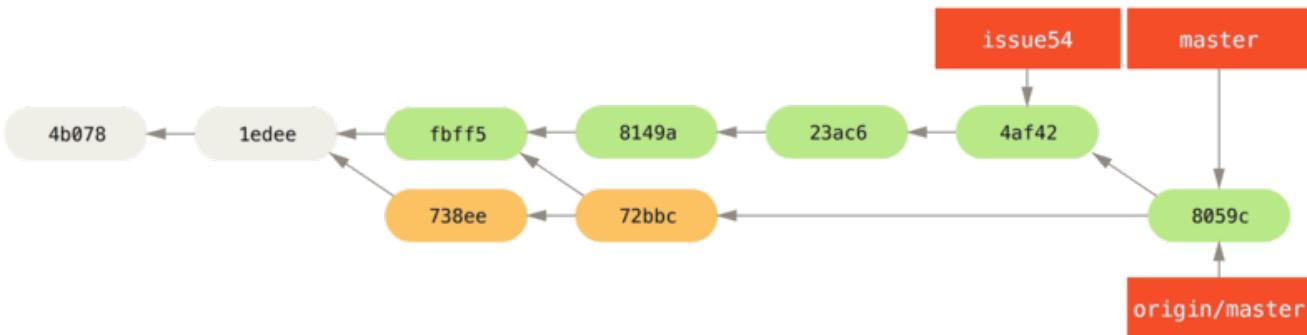


Figure 64. Historie van Jessica na al haar wijzigingen naar de server te hebben gepusht.

Dit is één van de eenvoudigste workflows. Je werkt een tijdje (over het algemeen in een topic branch) en merget dit in je `master`-branch als het klaar is om te worden geïntegreerd. Als je dat werk wilt delen, dan fetch merge je je eigen `master`-branch met de `origin/master` als die is gewijzigd, en als laatste push je naar de `master`-branch op de server. De algemene volgorde is als volgt:

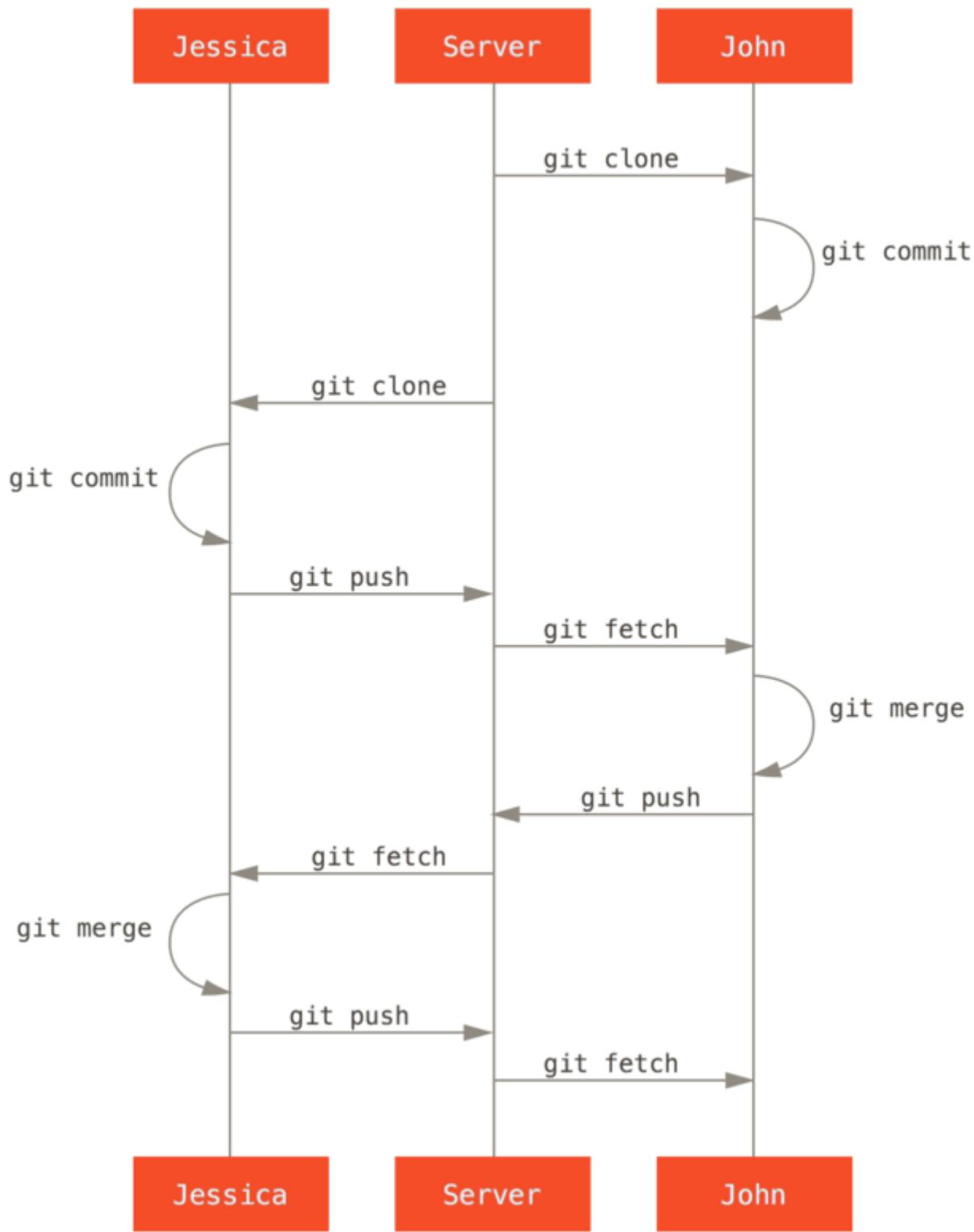


Figure 65. Algemene volgorde van gebeurtenissen voor een eenvoudige multi-ontwikkelaar Git workflow.

## Besloten aangestuurd team

In het volgende scenario zul je kijken naar de rol van de bijdragers in een grotere besloten groep. Je zult leren hoe te werken in een omgeving waar kleine groepen samenwerken aan functies, waarna die team-gebaseerde bijdragen worden geïntegreerd door een andere partij.

Stel dat John en Jessica samen werken aan een functie (laten we zeggen “featureA”), terwijl Jessica en een derde ontwikkelaar, Josie, aan een tweede (zeg, “featureB”) aan het werken zijn. In dit geval

gebruikt het bedrijf een integratie-manager achtige workflow, waarbij het werk van de individuele groepen alleen wordt geïntegreerd door bepaalde ontwikkelaars, en de `master`-branch van het hoofd repo alleen kan worden vernieuwd door die ontwikkelaars. In dit scenario wordt al het werk gedaan in team-gebaseerde branches en later door de integrators samengevoegd.

Laten we Jessica's workflow volgen terwijl ze aan haar twee features werkt, in parallel met twee verschillende ontwikkelaars in deze omgeving. We nemen even aan dat ze haar repository al gekloond heeft, en dat ze besloten heeft als eerste te werken aan `featureA`. Ze maakt een nieuwe branch aan voor de functie en doet daar wat werk:

```
# Jessica's Machine
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ vim lib/simplegit.rb
$ git commit -am 'add limit to log function'
[featureA 3300904] add limit to log function
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Op dit punt, moet ze haar werk delen met John, dus ze pusht haar commits naar de `featureA`-branch op de server. Jessica heeft geen push toegang op de `master`-branch — alleen de integratoren hebben dat — dus ze moet naar een andere branch pushen om samen te kunnen werken met John:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica mailt John om hem te zeggen dat ze wat werk gepusht heeft in een branch genaamd `featureA` en dat hij er nu naar kan kijken. Terwijl ze op terugkoppeling van John wacht, besluit Jessica te beginnen met het werken aan `featureB` met Josie. Om te beginnen start ze een nieuwe feature branch, gebaseerd op de `master`-branch van de server:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Nu doet Jessica een paar commits op de `featureB`-branch:

```
$ vim lib/simplegit.rb
$ git commit -am 'made the ls-tree function recursive'
[featureB e5b0fdc] made the ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'add ls-files'
[featureB 8512791] add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Jessica's repository ziet eruit als volgt:

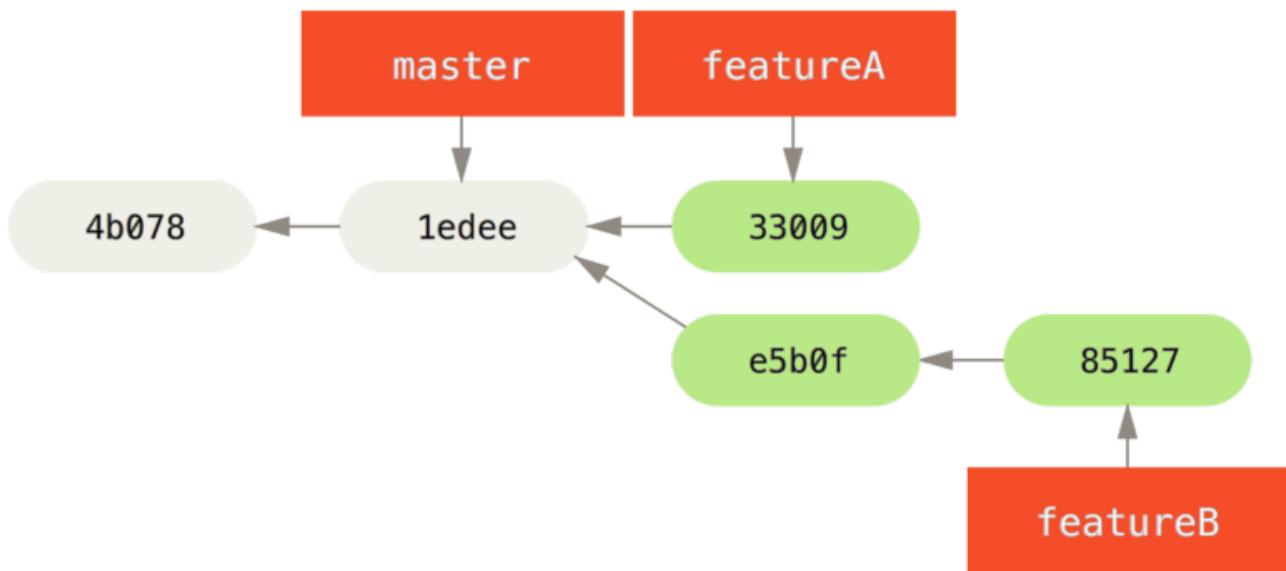


Figure 66. Initiële commit historie van Jessica.

Ze is klaar om haar werk te pushen, maar ze krijgt een mail van Josie dat een branch met wat initieel werk voor `featureB` erin al gepusht is naar de server in de `featureBee`-branch. Jessica moet die wijzigingen eerst mergen met die van haar voordat ze kan pushen naar de server. Ze kan dan Josie's wijzigingen ophalen met `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Aangenomen dat Jessica nog op haar uitgecheckte `featureB`-branch zit, kan Jessica dit nu mergen in het werk wat zij gedaan heeft met `git merge`:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by the 'recursive' strategy.
lib/simplegit.rb |    4 +---
1 files changed, 4 insertions(+), 0 deletions(-)
```

Nu wil Jessica al dit gemergede “featureB” werk terug pushen naar de server, maar ze wil niet eenvoudigweg haar eigen **featureB**-branch pushen. Omdat Josie al een **featureBee**-branch gemaakt heeft, zou Jessica op *die* branch willen pushen en dat doet ze met:

```
$ git push -u origin featureB:featureBee  
...  
To jessica@githost:simplegit.git  
fba9af8..cd685d1  featureB -> featureBee
```

Dit wordt een *refspec* genoemd. Zie [De Refspec](#) voor een meer gedetailleerde behandeling van Git refspecs en de verschillende dingen die je daarmee kan doen. Merk ook de **-u** vlag op; dit is een verkorte notatie voor **-set-upstream**, wat de branches voor eenvoudigere pushen en pullen op een later moment inricht.

Plotseling mailt John naar Jessica om te zeggen dat hij wat wijzigingen naar de **featureA**-branch waar ze op samenwerken gepusht heeft, om haar te vragen die te bekijken. Wederom voert Jessica een eenvoudige **git fetch** uit om *alle* nieuwe wijzigingen van de server op te halen, inclusief (uiteraard) het werk van John:

```
$ git fetch origin  
...  
From jessica@githost:simplegit  
3300904..aad881d  featureA -> origin/featureA
```

Jessica kan de log bekijken van John’s nieuwe werk door de inhoud van de zojuist opgehaalde **featureA**-branch met haar lokale kopie van dezelfde branch:

```
$ git log featureA..origin/featureA  
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6  
Author: John Smith <jsmith@example.com>  
Date:   Fri May 29 19:57:33 2009 -0700  
  
        changed log output to 30 from 25
```

Als wat Jessica ziet haar bevalt, kan ze het nieuwe werk van John in haar lokale **featureA**-branch mergen met:

```
$ git checkout featureA  
Switched to branch 'featureA'  
$ git merge origin/featureA  
Updating 3300904..aad881d  
Fast forward  
 lib/simplegit.rb |  10 ++++++++--  
 1 files changed, 9 insertions(+), 1 deletions(-)
```

Jessica kan tot slot nog wat kleine wijzigingen aanbrengen in de gemergede inhoud, dus ze het staat

haar vrij om dat te doen, deze wijzigingen committen in haar lokale `featureA`-branch en dan het eindresultaat terug pushen op de server.

```
$ git commit -am 'small tweak'  
[featureA 774b3ed] small tweak  
 1 files changed, 1 insertions(+), 1 deletions(-)  
$ git push  
...  
To jessica@githost:simplegit.git  
 3300904..774b3ed featureA -> featureA
```

De commit historie van Jessica ziet er nu zo uit:

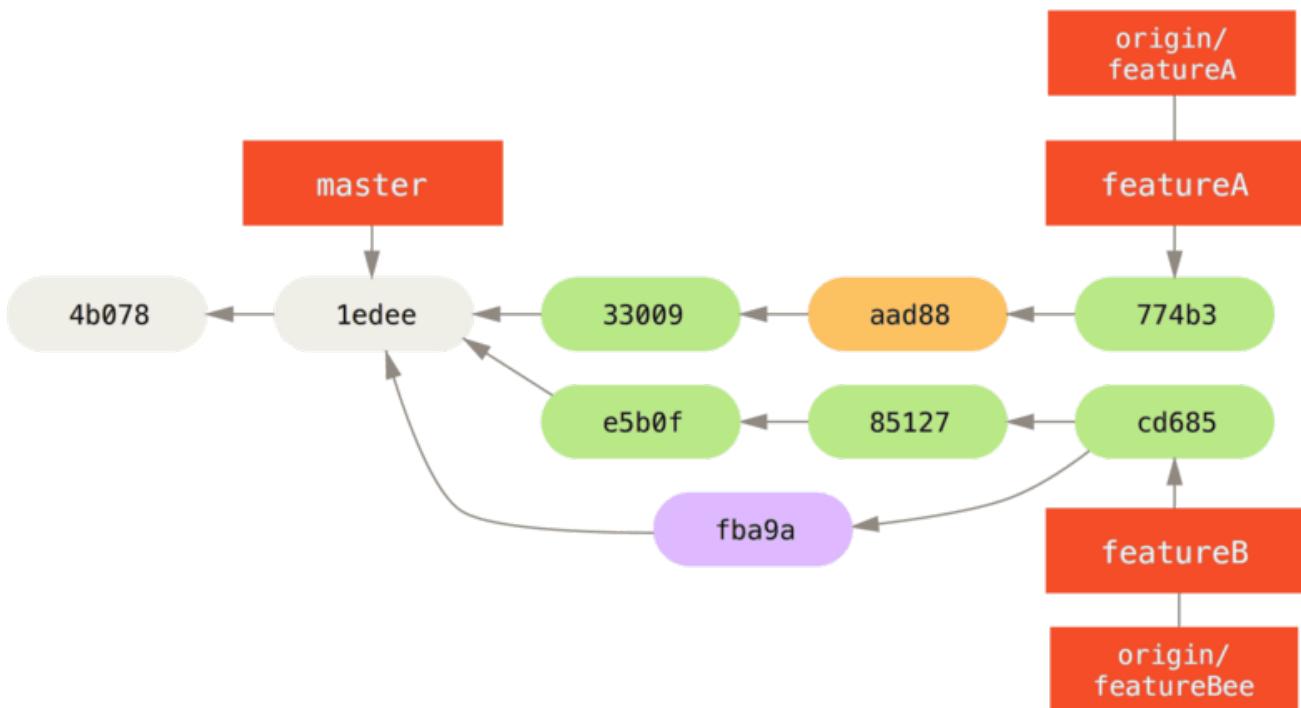


Figure 67. De historie van Jessica na het committen op een feature branch.

Jessica, Josie en John informeren de integrators nu dat de `featureA` en `featureBee`-branches op de server klaar zijn voor integratie in de hoofdlijn. Nadat zij die branches in de hoofdlijn geïntegreerd hebben, zal een fetch de nieuwe merge commits ophalen, waardoor de commit historie er zo uit ziet:

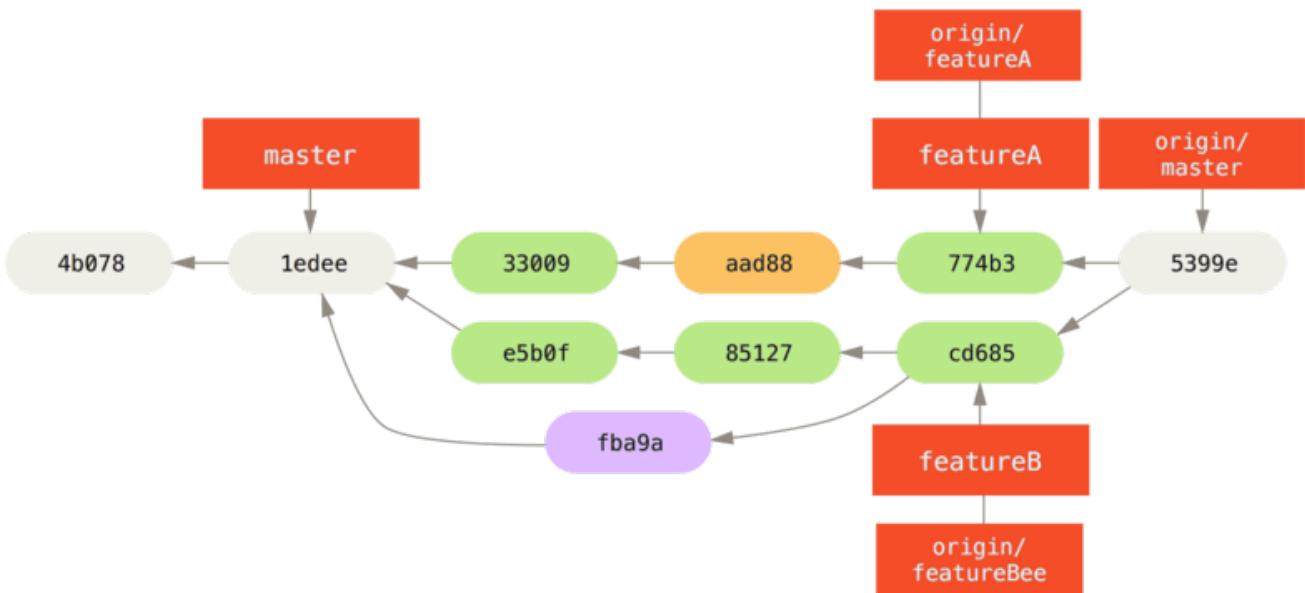


Figure 68. De historie van Jessica na het mergen van haar beide topic branches.

Veel groepen schakelen om naar Git juist vanwege de mogelijkheid om meerdere teams in parallel te kunnen laten werken, waarbij de verschillende lijnen van werk laat in het proces gemerged worden. De mogelijkheid van kleinere subgroepen of een team om samen te werken via remote branches zonder het hele team erin te betrekken of te hinderen is een enorm voordeel van Git. De volgorde van de workflow die je hier zag is ongeveer dit:

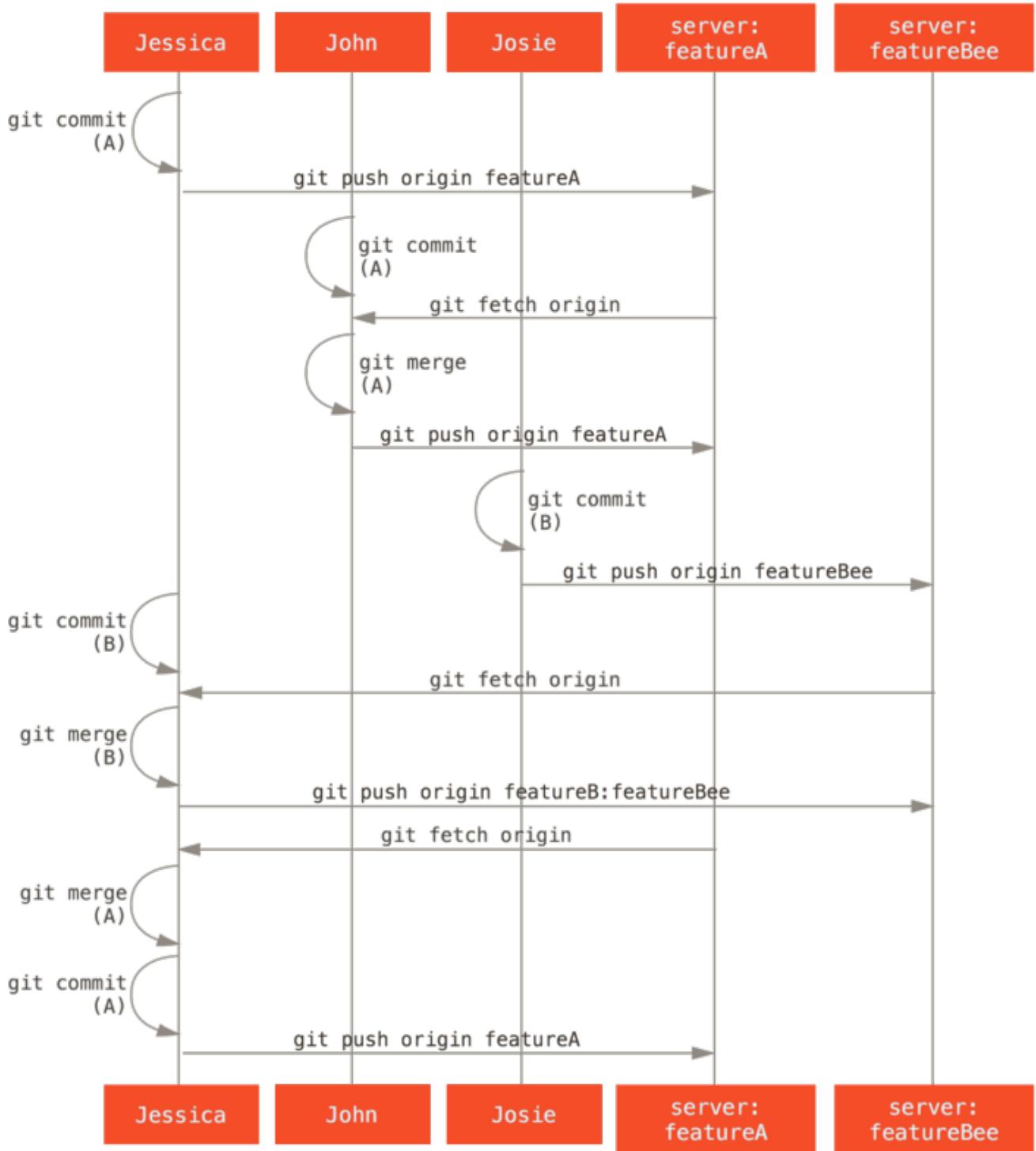


Figure 69. Eenvoudige volgorde in de workflow van dit aangestuurde team.

## Gevorkt openbaar project

Het bijdragen aan openbare, of publieke, projecten gaat op een iets andere manier. Omdat je niet de toestemming hebt om de branches van het project rechtstreeks te updaten, moet je het werk op een andere manier naar de beheerders krijgen. Dit eerste voorbeeld beschrijft het bijdragen via afsplitsen (forken) op Git hosts die het eenvoudig aanmaken van forks ondersteunen. Vele hosting sites ondersteunen dit (waaronder GitHub, BitBucket, Google Code, repo.or.cz, en andere), en veel project beheerders verwachten deze manier van bijdragen. De volgende paragraaf behandelt projecten die de voorkeur hebben om bijdragen in de vorm van patches via e-mail te ontvangen.

Eerst zal je waarschijnlijk de hoofdrepository klonen, een topic branch maken voor de patch of

reeks patches die je van plan bent bij te dragen, en je werk daarop doen. De te volgen stappen zien er in principe zo uit:

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
... work ...
$ git commit
... work ...
$ git commit
```



Je kunt eventueel besluiten `rebase -i` te gebruiken om je werk in één enkele commit samen te persen (squash), of het werk in de commits te herschikken om de patch eenvoudiger te kunnen laten reviewen door de beheerders—zie [Geschiedenis herschrijven](#) voor meer informatie over het interactief rebasen.

Als je werk op de branch af is, en je klaar bent om het over te dragen aan de beheerders, ga je naar de originele project pagina en klik op de “Fork” knop. Hiermee maak je een eigen overschrijfbare fork van het project. Je moet de URL van deze nieuwe repository URL toevoegen als een tweede remote, en laten we deze `myfork` noemen:

```
$ git remote add myfork <url>
```

Dan moet je je werk daar naartoe pushen. Het is het makkelijkst om de topic branch waar je op zit te werken te pushen naar je repository, in plaats van het te mergen in je master branch en die te pushen. De reden hiervan is, dat als het werk niet wordt geaccepteerd of alleen ge-cherry picked (deels overgenomen), je jouw master branch niet hoeft terug te draaien (de Git `cherry-pick` operatie wordt meer gedetailleerd behandeld in [Rebasende en cherry pick workflows](#)). Als de beheerders je werk mergen, rebasen of cherry picken, dan krijg je het uiteindelijk toch binnen door hun repository te pullen.

Hoe dan ook, je kunt je werk pushen met:

```
$ git push -u myfork featureA
```

Als jouw werk gepusht is naar jouw fork van de repository, dan moet je de beheerder van het oorspronkelijke project inlichten dat je werk hebt dat je ze wil laten mergen. Dit wordt een *pull request* (haal-binnen-verzoek) genoemd, en je kunt deze via de website genereren - GitHub heeft een eigen “Pull Request” mechanisme die we verder zullen behandelen in [GitHub](#) of je roept het `git request-pull` commando aan en stuurt een mail met de uitvoer handmatig naar de projectbeheerder.

Het `request-pull` commando neemt de basis branch waarin je de topic branch gepulld wil krijgen, en de URL van de Git repository waar je ze uit wil laten pullen, en maakt een samenvatting van alle wijzigingen die je gepulld wenst te hebben. Bijvoorbeeld, als Jessica John een pull request wil sturen, en ze heeft twee commits gedaan op de topic branch die ze zojuist gepusht heeft, dan kan ze

dit uitvoeren:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
Jessica Smith (1):
    added a new function

are available in the git repository at:

git://githost/simplegit.git featureA

Jessica Smith (2):
    add limit to log function
    change log output to 30 from 25

lib/simplegit.rb | 10 ++++++--
1 files changed, 9 insertions(+), 1 deletions(-)
```

De uitvoer kan naar de beheerders gestuurd worden—het vertelt ze waar het werk vanaf gebracht is, vat de commits samen en vertelt waar vandaan ze dit werk kunnen pullen.

Bij een project waarvan je niet de beheerder bent, is het over het algemeen eenvoudiger om een branch zoals `master` altijd de `origin/master` te laten tracken, en je werk te doen in topic branches die je eenvoudig weg kunt gooien als ze geweigerd worden. Als je je werkthema's gescheiden houdt in topic branches maakt dat het ook eenvoudiger voor jou om je werk te rebasen als de punt van de hoofd-repository in de tussentijd verschoven is en je commits niet langer netjes toegepast kunnen worden. Bijvoorbeeld, als je een tweede onderwerp wilt bijdragen aan een project, ga dan niet verder werken op de topic branch die je zojuist gepusht hebt—begin opnieuw vanaf de `master`-branch van de hoofd repository:

```
$ git checkout -b featureB origin/master
... work ...
$ git commit
$ git push myfork featureB
$ git request-pull origin/master myfork
... email generated request pull to maintainer ...
$ git fetch origin
```

Nu zijn al je onderwerpen opgeslagen in een silo — vergelijkbaar met een patch reeks (queue) — die je kunt herschrijven, rebasen en wijzigen zonder dat de onderwerpen elkaar beïnvloeden of van elkaar afhankelijk, hier is hoe je het kunt doen:

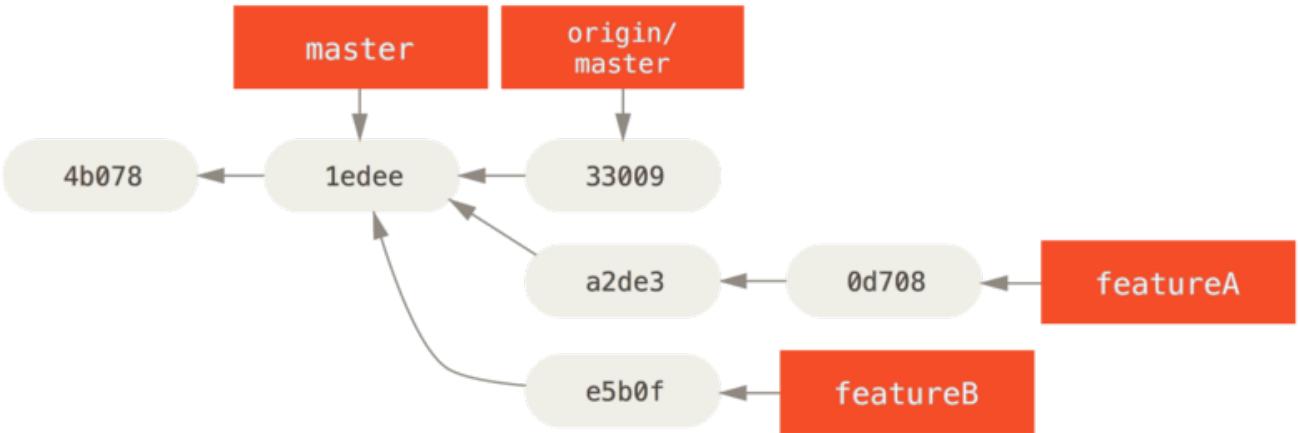


Figure 70. Initiële commit historie met werk van `featureB`.

Stel dat de project-beheerder een verzameling andere patches binnengehaald heeft en jouw eerste branch geprobeerd heeft, maar dat die niet meer netjes merged. In dat geval kun je proberen die branch te rebasen op `origin/master`, de conflicten op te lossen voor de beheerder, en dan je wijzigingen opnieuw aanbieden:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

Dit herschrijft je historie zodat die eruit ziet als in [Commit historie na werk van `featureA`](#).

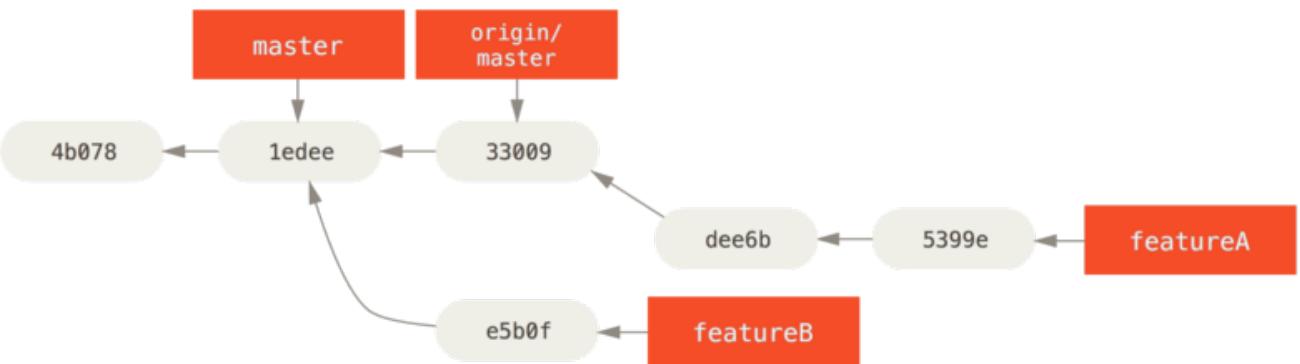


Figure 71. Commit historie na werk van `featureA`.

Omdat je de branch gerebased hebt, moet je de `-f` specificeren met je push commando om in staat te zijn de `featureA`-branch op de server te vervangen met een commit die er geen afstammeling van is. Een alternatief zou zijn dit nieuwe werk naar een andere branch op de server te pushen (misschien `featureAv2` genaamd).

Laten we eens kijken naar nog een mogelijk scenario: de beheerder heeft je werk bekeken in je tweede branch en vindt het concept goed, maar zou willen dat je een implementatie detail verandert. Je gebruikt deze gelegenheid meteen om het werk te baseren op de huidige `master`-branch van het project. Je begint een nieuwe branch gebaseerd op de huidige `origin/master`-branch, squasht de `featureB` wijzigingen er naartoe, lost eventuele conflicten op, doet de implementatie wijziging en pusht deze terug als een nieuwe branch:

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
... change implementation ...
$ git commit
$ git push myfork featureBv2
```

De `--squash` optie pakt al het werk op de gemerged branch en perst dat samen in één wijzigingsset (changeset) die de staat van de repository geeft alsof er een echte merge zou hebben plaatsgevonden, zonder feitelijk een merge commit te doen. Dit betekent dat je toekomstige commit maar één ouder heeft en geeft je de ruimte om alle wijzigingen te introduceren uit een andere branch en daarna meer wijzigingen te maken voordat de nieuwe commit wordt vastgelegd. Daarnaast kan de `--no-commit` handig zijn door de merge commit uit te stellen in plaats van het standaard merge proces.

Je kunt de beheerder nu een bericht sturen dat je de gevraagde wijzigingen gemaakt hebt en dat ze die wijzigingen kunnen vinden in je `featureBv2`-branch.

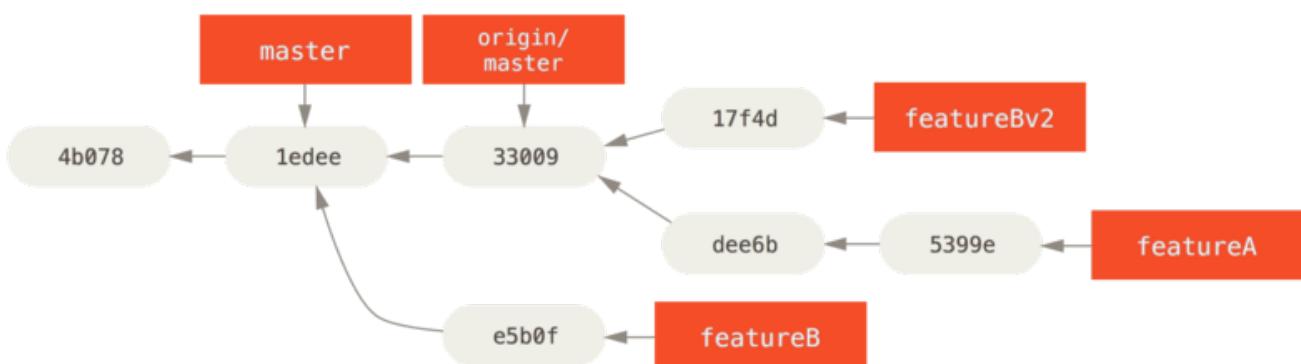


Figure 72. Commit historie na het `featureBv2` werk.

## Openbaar project per e-mail

Veel projecten hebben vastgestelde procedures voor het accepteren van patches—je zult de specifieke regels voor elk project moeten controleren, ze zullen namelijk verschillen. Omdat er verscheidene oudere, grotere projecten zijn die patches accepteren via ontwikkelaar-maillijsten, zullen we nu een voorbeeld hiervan behandelen.

De workflow is vergelijkbaar met het vorige geval—je maakt topic branches voor iedere patch waar je aan werkt. Het verschil is hoe je die aanlevert bij het project. In plaats van het project te fork en naar je eigen schrijfbare versie te pushen, genereer je e-mail versies van iedere reeks commits en mailt die naar de ontwikkelaar-maillijst:

```
$ git checkout -b topicA
... work ...
$ git commit
... work ...
$ git commit
```

Nu heb je twee commits die je naar de maillijst wilt sturen. Je gebruikt `git format-patch` om de mbox-geformatteerde bestanden te genereren die je kunt mailen naar de lijst. Dit maakt van iedere commit een e-mail bericht met de eerste regel van het commit bericht als het onderwerp, en de rest van het bericht plus de patch die door de commit wordt geïntroduceerd als de inhoud. Het prettige hieraan is dat met het toepassen van een patch uit een mail die gegenereerd is met `format-patch` alle commit informatie blijft behouden.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
```

Het `format-patch` commando drukt de namen af van de patch bestanden die het maakt. De `-M` optie vertelt Git te letten op naamswijzigingen. De bestanden komen er uiteindelijk zo uit te zien:

```
$ cat 0001-add-limit-to-log-function.patch
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function

Limit log functionality to the first 20

---
lib/simplegit.rb |    2 ++
1 files changed, 1 insertions(+), 1 deletions(-)

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 76f47bc..f9815f1 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -14,7 +14,7 @@ class SimpleGit
  end

  def log(treeish = 'master')
-    command("git log #{treeish}")
+    command("git log -n 20 #{treeish}")
  end

  def ls_tree(treeish = 'master')
-- 
2.1.0
```

Je kunt deze patch bestanden ook aanpassen om meer informatie, die je niet in het commit bericht wilt laten verschijnen, voor de maillijst toe te voegen. Als je tekst toevoegt tussen de `---` regel en het begin van de patch (de `diff --git` regel), dan kunnen ontwikkelaars dit lezen, maar tijdens het toepassen van de patch wordt dit genegeerd.

Om dit te mailen naar een maillijst, kan je het bestand in je mail-applicatie plakken of het sturen via een commandoregel programma. Het plakken van de tekst veroorzaakt vaak formateringsproblemen, in het bijzonder bij “slimmere” clients die de newlines en andere witruimte niet juist behouden. Gelukkig levert Git een gereedschap die je helpt om juist geformatteerde patches via IMAP te versturen, wat het alweer een stuk makkelijker voor je kan maken. We zullen je laten zien hoe je een patch via Gmail stuurt, wat de mail-applicatie is waar we het meest bekend mee zijn. Je kunt gedetailleerde instructies voor een aantal mail programma’s vinden aan het eind van het eerder genoemde [Documentation/SubmittingPatches](#) bestand in de Git broncode.

Eerst moet je de imap sectie in je `~/.gitconfig` bestand instellen. Je kunt iedere waarde apart instellen met een serie `git config` commando’s, of je kunt ze handmatig toevoegen, maar uiteindelijk moet je config bestand er ongeveer zo uitzien:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = YX]8g76G_2^sFbd
  port = 993
  sslverify = false
```

Als je IMAP server geen SSL gebruikt, zijn de laatste twee regels waarschijnlijk niet nodig, en de waarde voor host zal `imap://` zijn in plaats van `imaps://`. Als dat ingesteld is, kun je `git imap-send` gebruiken om de patch reeks in de Drafts map van de gespecificeerde IMAP server te zetten:

```
$ cat *.patch |git imap-send
Resolving imap.gmail.com... ok
Connecting to [74.125.142.109]:993... ok
Logging in...
sending 2 messages
100% (2/2) done
```

Nu zou je in staat moeten zijn om naar je Drafts folder te gaan, het To veld te veranderen in het adres van de mail lijst waar je de patch naar toe stuurt, en mogelijk de onderhouder of de persoon verantwoordelijk voor dat deel te CC-en, en het te versturen.

Je kunt de patches ook via een SMTP server sturen. Net als hierboven, kan je elke waarde apart zetten met een reeks van `git config` commando’s, of je kunt ze handmatig in de sendemail sectie in je `~/.gitconfig` bestand toevoegen:

```
[sendemail]
smtpencryption = tls
smtpserver = smtp.gmail.com
smtpuser = user@gmail.com
smtpserverport = 587
```

Als dit gedaan is, kan je `git send-email` gebruiken om je patches te sturen:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Dan zal Git een berg log-informatie oplepelen die er ongeveer zo uitziet voor elke patch die je aan het versturen bent:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
      \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

## Samenvatting

In dit hoofdstuk is een aantal veel voorkomende workflows behandeld, die je kunt gebruiken om te kunnen werken in een aantal zeer verschillende typen Git projecten die je kunt tegenkomen. En er zijn een aantal nieuwe gereedschappen geïntroduceerd die je helpen om dit proces te beheren. Wat hierna volgt zal je laten zien hoe je aan de andere kant van de tafel werkt: een Git project beheren. Je zult leren hoe een welwillende dictator of integratie manager te zijn.

## Het beheren van een project

Naast weten hoe effectief bij te dragen aan een project, is het ook handig om te weten hoe je er een beheert. Dit kan bestaan uit het accepteren en toepassen van patches die met `format-patch` gemaakt

en naar je gemaild zijn, of het integreren van wijzigingen in de remote branches van repositories die je hebt toegevoegd als remotes van je project. Of je nu een canonieke repository beheert, of wilt bijdragen door het controleren of goedkeuren van patches, je moet weten hoe werk te ontvangen op een manier die het duidelijkst is voor andere bijdragers en voor jou op langere termijn vol te houden.

## Werken in topic branches

Als je overweegt om nieuw werk te integreren, is het over het algemeen een goed idee om het uit te proberen in een topic branch — een tijdelijke branch, speciaal gemaakt om dat nieuwe werk uit te proberen. Op deze manier is het handig om een patch individueel te bewerken en het even opzij te zetten als het niet lukt, totdat je tijd hebt om er op terug te komen. Als je een eenvoudige branchnaam maakt, gebaseerd op het onderwerp van het werk dat je aan het proberen bent, bijvoorbeeld `ruby_client` of iets dergelijks, dan is het makkelijk om te herinneren als je het voor een tijdje opzij legt en er later op terug komt. De beheerder van het Git project heeft de neiging om deze branches ook van een naamsruimte (namespace) te voorzien — zoals `sc/ruby_client`, waarbij `sc` een afkorting is van de persoon die het werk heeft bijgedragen. Zoals je je zult herinneren, kun je de branch gebaseerd op je `master`-branch zo maken:

```
$ git branch sc/ruby_client master
```

Of, als je er ook meteen naar wilt omschakelen, kun je de `checkout -b` optie gebruiken:

```
$ git checkout -b sc/ruby_client master
```

Nu ben je klaar om het bijgedragen werk in deze topic branch toe te voegen, en te bepalen of je het wilt mergen in je meer permanente branches.

## Patches uit e-mail toepassen

Als je een patch per e-mail ontvangt, en je moet die integreren in je project, moet je de patch in je topic branch toepassen om het te evalueren. Er zijn twee manieren om een gemailde patch toe te passen: met `git apply` of met `git am`.

### Een patch toepassen met `apply`

Als je de patch ontvangen hebt van iemand die het gegenereerd heeft met de `git diff` of een Unix `diff` commando (wat niet wordt aangeraden, zie de volgende paragraaf), kun je het toepassen met het `git apply` commando. Aangenomen dat je de patch als `/tmp/patch-ruby-client.patch` opgeslagen hebt, kun je de patch als volgt toepassen:

```
$ git apply /tmp/patch-ruby-client.patch
```

Dit wijzigt de bestanden in je werk directory. Het is vrijwel gelijk aan het uitvoeren van een `patch -p1` commando om de patch toe te passen, alhoewel het meer paranoïde is en minder "fuzzy matches" accepteert dan patch. Het handelt ook het toevoegen, verwijderen, en hernoemen van

bestanden af als ze beschreven staan in het `git diff` formaat, wat `patch` niet doet. Als laatste volgt `git apply` een “pas alles toe of laat alles weg” model waarbij alles of niets wordt toegepast. Dit in tegenstelling tot `patch` die gedeeltelijke patches kan toepassen, waardoor je werkdirctory in een vreemde status achterblijft. Over het algemeen is `git apply` terughoudender dan `patch`. Het zal geen commit voor je aanmaken—na het uitvoeren moet je de geïntroduceerde wijzigingen handmatig staggen en committen.

Je kunt ook `git apply` gebruiken om te zien of een patch netjes kan worden toepast voordat je het echt doet—je kunt `git apply --check` uitvoeren met de patch:

```
$ git apply --check 0001-seeing-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Als er geen uitvoer is, dan zou de patch netjes moeten passen. Dit commando retourneert ook een niet-nul status als de controle faalt, zodat je het kunt gebruiken in scripts als je dat zou willen.

### Een patch met `am` toepassen

Als de bijdrager een Git gebruiker is en zo vriendelijk is geweest om het `format-patch` commando te gebruiken om de patch te genereren, dan is je werk eenvoudiger omdat de patch de auteur-informatie en een commit-bericht voor je bevat. Als het enigzins kan, probeer dan je bijdragers aan te moedigen om `format-patch` te gebruiken in plaats van `diff` om patches voor je te genereren. Je zou alleen ‘`git apply` hoeven te gebruiken voor oude patches en dergelijke zaken.

Om een patch gegenereerd met `format-patch` toe te passen, gebruik je `git am` (het commando wordt `am` genoemd, omdat het wordt gebruikt om “een serie van patches toe te passen [apply] uit een mailbox”. Technisch is `git am` gemaakt om een mbox bestand te lezen, wat een eenvoudig gewone platte tekstformaat is om één of meer e-mail berichten in een tekstbestand op te slaan. Het ziet er ongeveer zo uit:

```
From 330090432754092d704da8e76ca5c05c198e71a8 Mon Sep 17 00:00:00 2001
From: Jessica Smith <jessica@example.com>
Date: Sun, 6 Apr 2008 10:17:23 -0700
Subject: [PATCH 1/2] add limit to log function
```

```
Limit log functionality to the first 20
```

Dit is het begin van de uitvoer van het `git format-patch` commando dat je gezien hebt in de vorige paragraaf. Dit is ook een geldig mbox e-mail formaat. Als iemand jou de patch correct gemaild heeft door gebruik te maken van `git send-email` en je download dat in een mbox formaat, dan kan je `git am` naar dat mbox bestand verwijzen, en het zal beginnen met alle patches die het tegenkomt toe te passen. Als je een mail client gebruikt die meerdere e-mails kan opslaan in mbox formaat, dan kun je hele reeksen patches in een bestand opslaan en dan `git am` gebruiken om ze één voor één toe te passen.

Maar, als iemand een patch bestand heeft geüpload die gegenereerd is met `git format-patch` naar

een ticket systeem of zoets, kun je het bestand lokaal opslaan en dan dat opgeslagen bestand aan `git am` doorgeven om het te applyen:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

Je ziet dat het netjes is toegepast, en automatisch een nieuwe commit voor je heeft aangemaakt. De auteursinformatie wordt gehaald uit de `From` en `Date` velden in de kop van de e-mail, en het bericht van de commit wordt gehaald uit de `Subject` en de inhoud (voor de patch) van het mailbericht zelf. Bijvoorbeeld, als deze patch was toegepast van het mbox voorbeeld hierboven, dan zou de gegenereerde commit er ongeveer zo uit zien:

```
$ git log --pretty=fuller -1
commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Author: Jessica Smith <jessica@example.com>
AuthorDate: Sun Apr 6 10:17:23 2008 -0700
Commit: Scott Chacon <schacon@gmail.com>
CommitDate: Thu Apr 9 09:19:06 2009 -0700

    add limit to log function

    Limit log functionality to the first 20
```

De `Commit` informatie toont de persoon die de patch toegepast heeft en de tijd waarop het is toegepast. De `Author` informatie toont de persoon die de patch oorspronkelijk gemaakt heeft en wanneer het gemaakt is.

Maar het is mogelijk dat de patch niet netjes toegepast kan worden. Misschien is jouw hoofdbranch te ver afgeweken van de branch waarop de patch gebouwd is, of is de patch afhankelijk van een andere patch, die je nog niet hebt toegepast. In dat geval zal het `git am` proces falen en je vragen wat je wilt doen:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Dit commando zet conflict markeringen in alle bestanden waar het problemen mee heeft, net zoals een conflicterende merge of rebase operatie. Je lost dit probleem op een vergelijkbare manier op—wijzig het bestand om het conflict op te lossen, stage het bestand en voer dan `git am --resolved` uit om door te gaan met de volgende patch:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

Als je wilt dat Git iets meer intelligentie toepast om het conflict op te lossen, kun je een **-3** optie eraan meegeven, dit zorgt ervoor dat Git een driewegs-merge probeert. Deze optie staat standaard niet aan omdat het niet werkt als de commit waarvan de patch zegt dat het op gebaseerd is niet in je repository zit. Als je die commit wel hebt—als de patch gebaseerd was op een gepubliceerde commit—dan is de **-3** over het algemeen veel slimmer in het toepassen van een conflicterende patch:

```
$ git am -3 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

In dit geval, zonder de **-3** optie zou het als een conflict zijn beschouwd. Omdat de **-3** optie gebruikt is, is de patch netjes toegepast.

Als je een aantal patches van een mbox toepast, kun je ook het **am** commando in een interactieve modus uitvoeren, wat bij iedere patch die het vindt stopt en je vraagt of je het wilt applyen:

```
$ git am -3 -i mbox
Commit Body is:
-----
seeing if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Dit is prettig wanneer je een aantal patches opgespaard hebt, omdat je de patch eerst kunt zien als je je niet kunt herinneren wat het is, of de patch niet toepassen omdat je dat al eerder gedaan hebt.

Als alle patches voor je topic branch zijn toegepast en gecommit zijn op je branch, kan je besluiten of en hoe ze te integreren in een branch met een langere looptijd.

## Remote branches uitchecken

Als je bijdrage van een Git gebruiker komt die zijn eigen repository opgezet heeft, een aantal patches daarin gepusht heeft, en jou de URL naar de repository gestuurd heeft en de naam van de remote branch waarin de wijzigingen zitten, kan je ze toevoegen als een remote en het mergen lokaal doen.

Bijvoorbeeld, als Jessica je een e-mail stuurt waarin staat dat ze een prachtig nieuwe feature

in de `ruby-client`-branch van haar repository heeft, kun je deze testen door de remote toe te voegen en die branch lokaal te bekijken:

```
$ git remote add jessica git://github.com/jessica/myproject.git  
$ git fetch jessica  
$ git checkout -b rubyclient jessica/ruby-client
```

Als ze je later opnieuw mailt met een andere branch die weer een andere mooie feature bevat, dan kun je die meteen ophalen (`fetch`) en bekijken (`checkout`) omdat je de remote al ingesteld hebt.

Dit is meest praktisch als je vaak met een persoon werkt. Als iemand eens in de zoveel tijd een enkele patch bij te dragen heeft, dan is het accepteren per mail misschien minder tijdrovend dan te eisen dat iedereen hun eigen server moet beheren, en daarna voortdurend remotes te moeten toevoegen en verwijderen voor die paar patches. Je zult daarbij waarschijnlijk ook niet honderden remotes willen hebben, elk voor iemand die maar een patch of twee bijdraagt. Aan de andere kant, scripts en gehoste diensten maken het wellicht eenvoudiger — het hangt sterk af van de manier waarop je ontwikkelt en hoe je bijdragers ontwikkelen.

Een bijkomend voordeel van deze aanpak is dat je de historie van de commits ook krijgt. Alhoewel je misschien terechte merge-problemen hebt, weet je op welk punt in de historie hun werk is gebaseerd; een echte drieweg merge is de standaard in plaats van een `-3` te moeten meegeven en hopen dat de patch gegenereerd was van een publieke commit waar je toegang toe hebt.

Als je maar af en toe met een persoon werkt, maar toch op deze manier van hen wilt pullen, dan kun je de URL van de remote repository geven aan het `git pull` commando. Dit doet een eenmalig pull en bewaart de URL niet als een remote referentie:

```
$ git pull https://github.com/onetimeguy/project  
From https://github.com/onetimeguy/project  
 * branch HEAD      -> FETCH_HEAD  
Merge made by the 'recursive' strategy.
```

## Bepalen wat geïntroduceerd is geworden

Je hebt een topic branch dat bijgedragen werk bevat. Nu kan je besluiten wat je er mee wilt doen. Deze paragraaf worden een aantal commando's nogmaals behandeld om te laten zien hoe je ze kunt gebruiken om precies te reviewen wat je zult introduceren als je dit merged in je hoofd-branch.

Het is vaak handig om een overzicht te krijgen van alle commits die in deze branch zitten, maar die niet in je master-branch zitten. Je kunt commits weglaten die al in de master branch zitten door de `--not` optie mee te geven voor de branch naam. Dit doet hetzelfde als het `master..contrib` formaat dat we eerder gebruikt hebben. Bijvoorbeeld, als je bijdrager je twee patches stuurt, je hebt een branch genaamd `contrib` gemaakt en hebt die patches daar toegepast, dan kun je dit uitvoeren:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700
```

seeing if this helps the gem

```
commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700
```

updated the gemspec to hopefully work better

Om te zien welke wijzigingen door een commit worden geïntroduceerd, onthoud dan dat je de `-p` optie kunt meegeven aan `git log` en dan zal de geïntroduceerde diff bij elke commit erachter geplakt worden.

Om een volledige diff te zien van wat zou gebeuren als je deze topic branch merged met een andere branch, zul je misschien een vreemde truc moeten toepassen om de juiste resultaten te krijgen. Je zult misschien dit uit willen voeren:

```
$ git diff master
```

Dit commando geeft je een diff, maar het kan misleidend zijn. Als je `master`-branch vooruit geschoven is sinds je de topic branch er vanaf hebt gemaakt, dan zul je ogenschijnlijk vreemde resultaten krijgen. Dit gebeurt omdat Git de snapshots van de laatste commit op de topic branch waar je op zit vergelijkt met de snapshot van de laatste commit op de `master`-branch. Bijvoorbeeld, als je een regel in een bestand hebt toegevoegd op de `master`-branch, dan zal een directe vergelijking van de snapshots eruit zien alsof de topic branch die regel gaat verwijderen.

Als `master` een directe voorganger is van je topic branch is dit geen probleem, maar als de twee histories uit elkaar zijn gegaan, zal de diff eruit zien alsof je alle nieuwe spullen in je topic-branch toevoegt en al hetgeen wat alleen in de `master`-branch staat weghaalt.

Wat je eigenlijk had willen zien zijn de wijzigingen die in de topic branch zijn toegevoegd — het werk dat je zult introduceren als je deze branch met `master` merget. Je doet dat door Git de laatste commit op je topic branch te laten vergelijken met de eerste gezamenlijke voorouder die het heeft met de `master`-branch.

Technisch, kun je dat doen door de gezamenlijke voorouder op te zoeken en daar je diff op uit te voeren:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

of, beknopter:

```
$ git diff $(git merge-base contrib master)
```

Echter, dat is niet handig, dus levert Git een andere verkorte manier om hetzelfde te doen: de driedubbele punt syntax. In de context van het **diff** commando, kun je drie punten achter een andere branch zetten om een **diff** te doen tussen de laatste commit van de branch waar je op zit en de gezamenlijke voorouder met een andere branch:

```
$ git diff master...contrib
```

Dit commando laat alleen het werk zien dat je huidige topic branch heeft geïntroduceerd sinds de gezamenlijke voorouder met master. Dat is een erg handige syntax om te onthouden.

## Bijgedragen werk integreren

Als al het werk in je onderwerp branch klaar is om te worden geïntegreerd in een hogere branch, dan is de vraag hoe dat te doen. En daarbij, welke workflow wil je gebruiken om je project te beheren? Je hebt een aantal alternatieven, dus we zullen er een aantal behandelen.

### Mergende workflows

Een eenvoudige workflow is om al dat werk direct in de **master**-branch te mergen. In dit scenario heb je een **master**-branch die feitelijk de stabiele code bevat. Als je werk in een topic branch hebt waaraan je gewerkt hebt, of dat iemand anders heeft bijgedragen en je hebt dat nagekeken, dan merge je het in de master branch, verwijdert de topic branch en herhaalt het proces.

Bijvoorbeeld, als we een repository hebben met werk in twee branches genaamd **ruby\_client** en **php\_client**, wat eruit ziet zoals [Historie met een aantal topic branches](#). en mergen eerst **ruby\_client** en daarna **php\_client**, dan zal je historie er uit gaan zien zoals in [Na het mergen van een topic branch..](#)

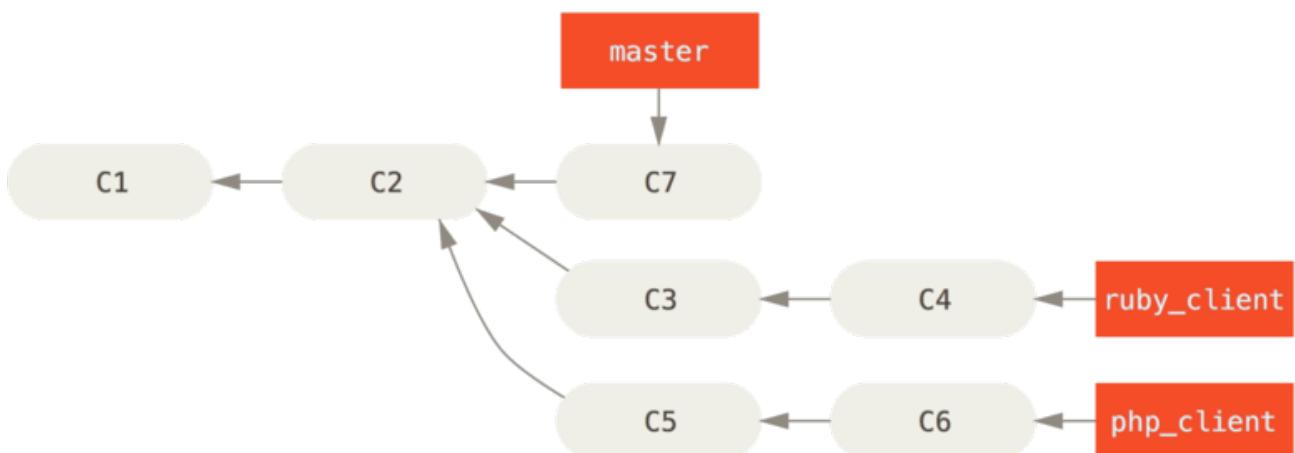


Figure 73. Historie met een aantal topic branches.

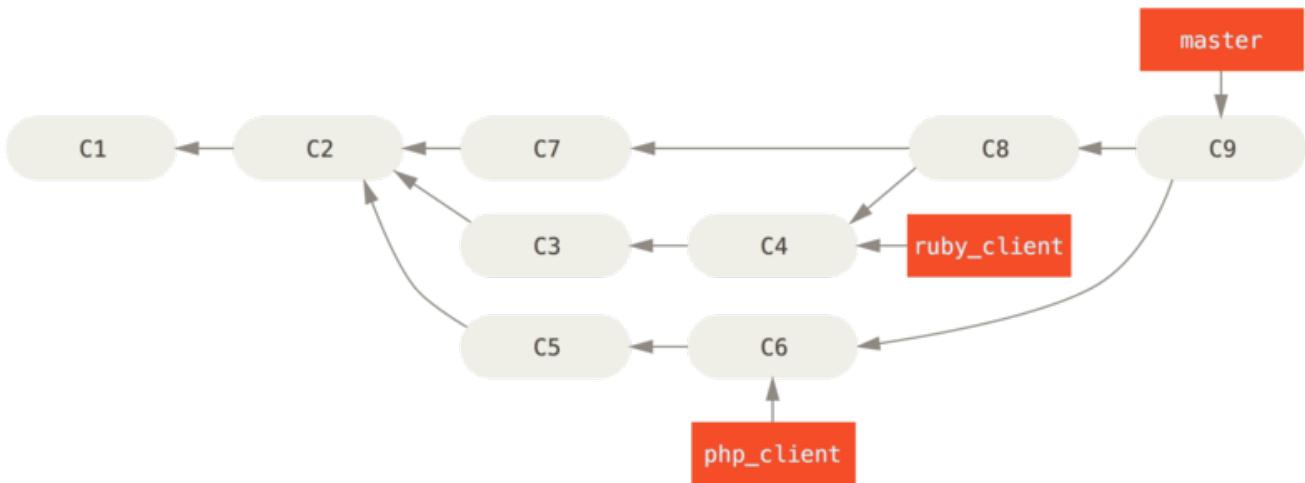


Figure 74. Na het mergen van een topic branch.

Dat is waarschijnlijk de eenvoudigste workflow, maar het wordt problematisch als je werkt met grotere of stabielere projecten waar je heel voorzichtig moet zijn met wat je introduceert.

Als je belangrijkere projecten hebt, zou je kunnen denken aan een twee-fasen merge cyclus. In dit scenario heb je twee langlopende branches, `master` en `develop`, waarin je bepaalt dat `master` alleen wordt geupdate als een hele stabiele release wordt gemaakt en alle nieuwe code wordt geïntegreerd in de `develop`-branch. Je pusht beide branches regelmatig naar de publieke repository. Elke keer heb je een nieuwe topic branch om in te mergen ([Voor de merge van een topic branch.](#)), je merget deze in `develop` ([Na de merge van een topic branch.](#)); daarna, als je een release tagt, fast-forward je `master` naar waar de nu stabiele `develop`-branch is ([Na een project release.](#)).

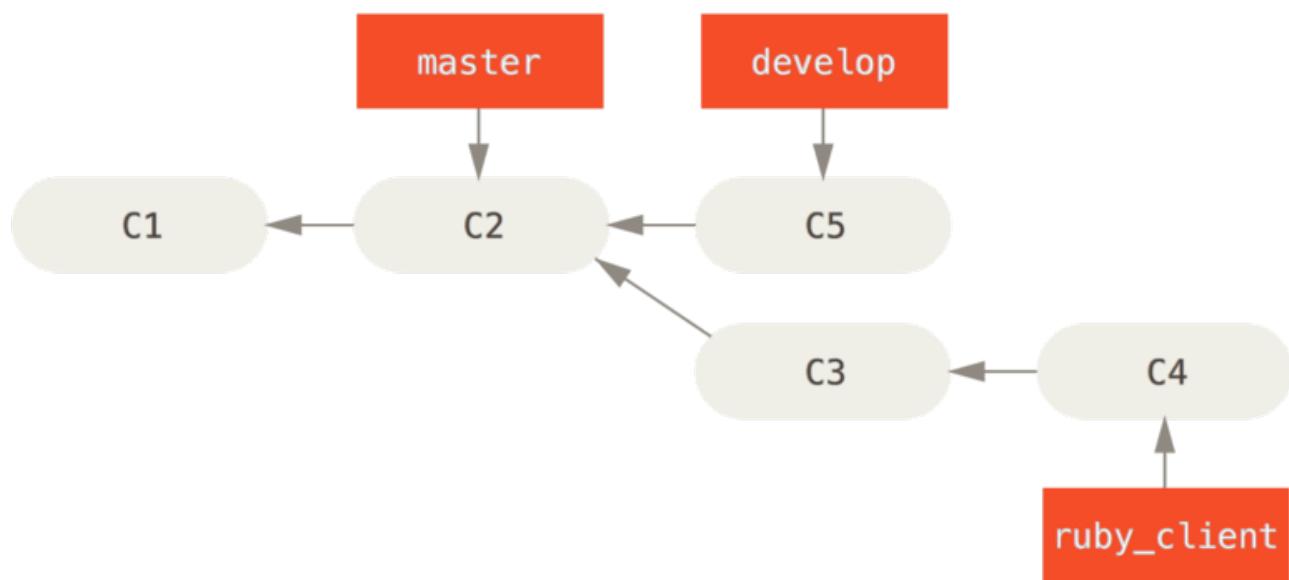


Figure 75. Voor de merge van een topic branch.

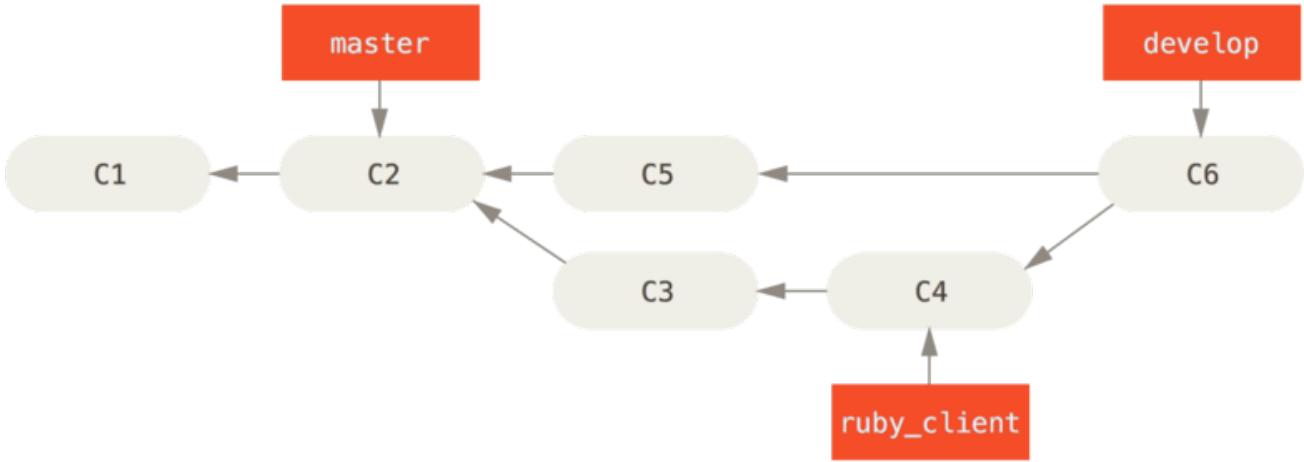


Figure 76. Na de merge van een topic branch.

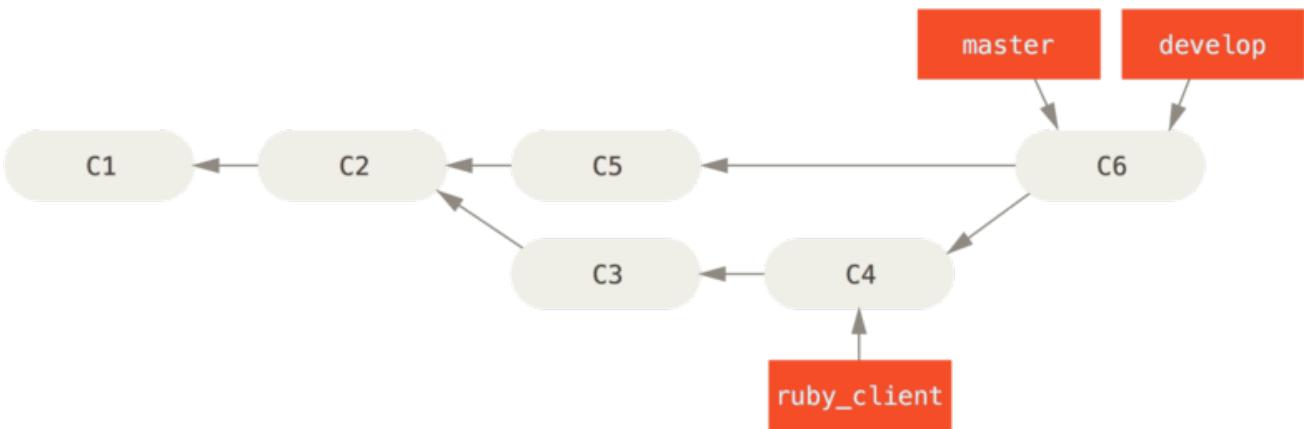


Figure 77. Na een project release.

Op deze manier kunnen mensen, wanneer ze de repository van jouw project klonen, ervoor kiezen om je master uit te checken om de laatste stabiele versie te maken en eenvoudig up-to-date te blijven op die versie, of ze kunnen develop uitchecken, die het nieuwste materiaal bevat. Je kunt dit concept ook doortrekken, en een **integrate**-branch hebben waar al het werk wordt gemerged. Dan, als de code op die branch stabiel is en de tests passeert, kan je dit op de **develop**-branch mergeren, als dat dan enige tijd stabiel is gebleken kan je de **master**-branch fast-forwarden.

## Workflows met grote merges

Het Git project heeft vier langlopende branches: **master**, **next**, en **pu** (proposed updates, voorgestelde wijzigingen) voor nieuw spul, en **maint** voor onderhoudswerk (maintenance backports). Als nieuw werk wordt geïntroduceerd door bijdragers, wordt het samengeraapt in topic branches in de repository van de beheerder op een manier die lijkt op wat we omschreven hebben (zie [Een complexe reeks van parallelle bijgedragen topic branches beheren](#)). Hier worden de topics geëvalueerd om te bepalen of ze veilig zijn en klaar voor verdere verwerking of dat ze nog wat werk nodig hebben. Als ze veilig zijn, worden ze in **next** gemerged, en wordt die branch gepusht zodat iedereen de geïntegreerde topics kan uitproberen.

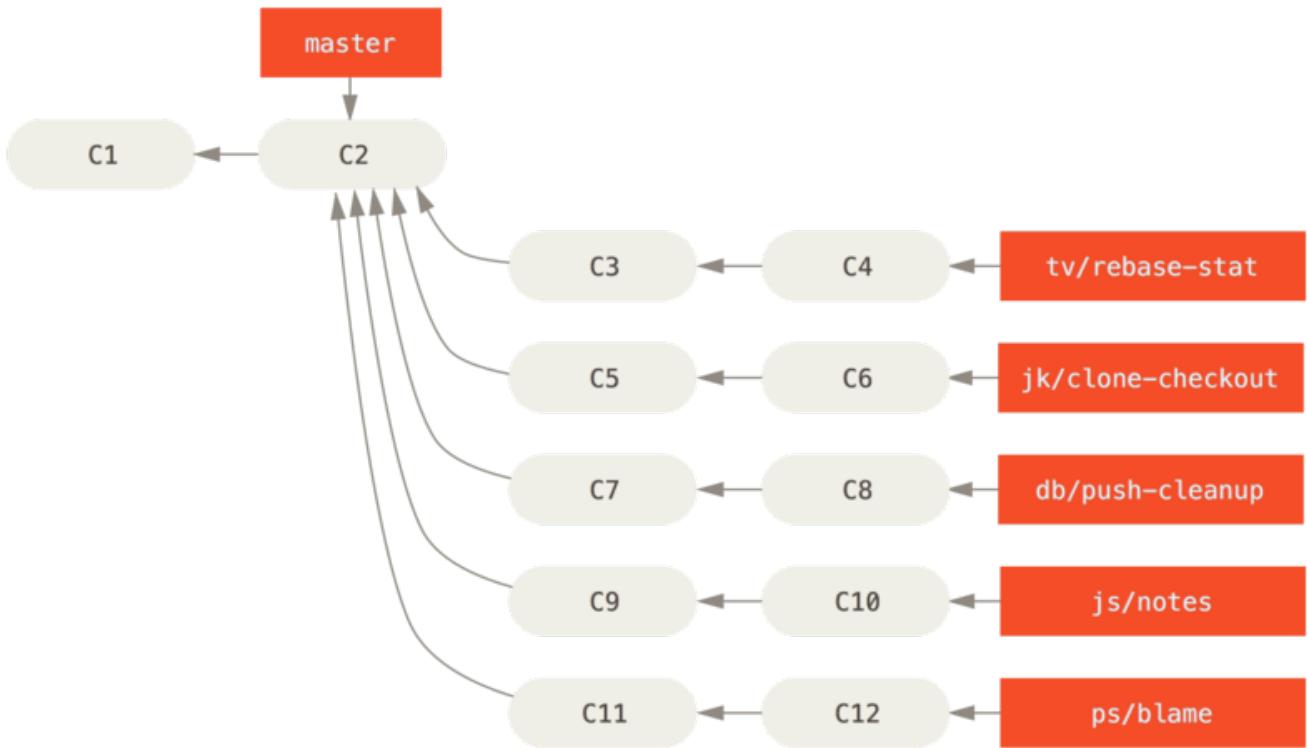


Figure 78. Een complexe reeks van parallele bijgedragen topic branches beheren.

Als de topics nog werk nodig hebben, dan worden ze in plaats daarvan gemerged in `pu`. Zodra vastgesteld is dat ze helemaal stabiel zijn, dan worden de topics opnieuw gemerged in `master`. De `next` en `pu`-branches worden dan opnieuw opgebouwd vanaf de `master`. Dit betekent dat `master` vrijwel altijd vooruit beweegt, `next` eens in de zoveel tijd gerebased wordt, en `pu` nog vaker gerebased wordt:

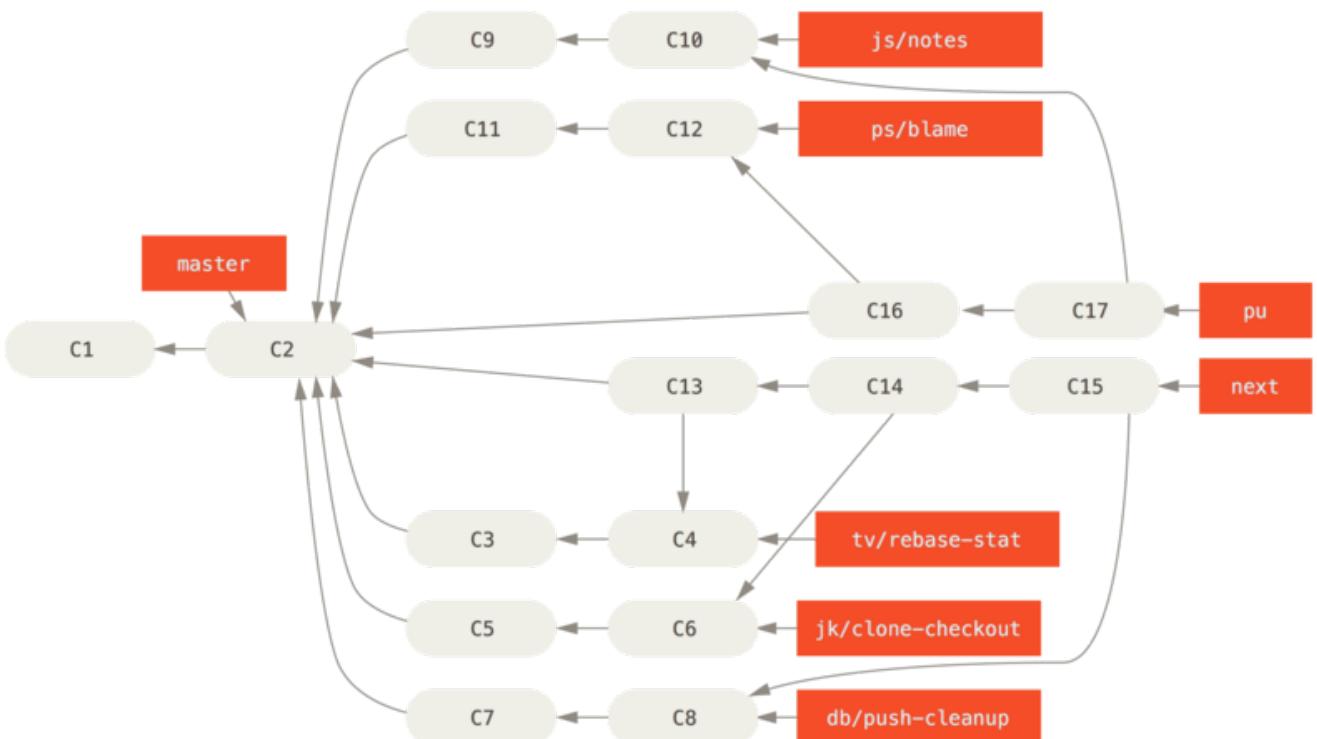


Figure 79. Bijgedragen topic branches mergen in langlopende integratie branches.

Als een topic branch uiteindelijk is gemerged in `master`, dan wordt het verwijderd van de

repository. Het Git project heeft ook een `maint`-branch, die geforkt is van de laatste release om teruggewerkte (backported) patches te leveren in het geval dat een onderhoudsrelease nodig is. Dus als je de Git repository kloont, dan heb je vier branches die je kunt uitchecken om het project in verschillende stadia van ontwikkeling te evalueren, afhankelijk van hoe nieuw je alles wilt hebben of hoe je wil bijdragen; en de onderhouder heeft een gestructureerde workflow om nieuwe bijdragen aan de tand te voelen. De workflow van het Git project is gespecialiseerd. Om dit goed te begrijpen zou je het [Git Beheerdersgids](#) kunnen lezen.

## Rebasende en cherry pick workflows

Andere beheerders geven de voorkeur aan rebasen of bijgedragen werk te cherry-picken naar hun master branch in plaats van ze erin te mergen, om een vrijwel lineaire historie te behouden. Als je werk in een topic branch hebt en hebt besloten dat je het wil integreren, dan ga je naar die branch en voert het rebase commando uit om de wijzigingen op je huidige master branch te baseren (of `develop`, enzovoorts). Als dat goed werkt, dan kun je de `master`-branch fast-forwarden, en eindig je met een lineaire project historie.

De andere manier om geïntroduceerd werk van de ene naar de andere branch te verplaatsen is om het te cherry-picken. Een cherry-pick in Git is een soort rebase voor een enkele commit. Het pakt de patch die was geïntroduceerd in een commit en probeert die weer toe te passen op de branch waar je nu op zit. Dit is handig als je een aantal commits op een topic branch hebt en je er slechts één van wilt integreren, of als je alleen één commit op een topic branch hebt en er de voorkeur aan geeft om het te cherry-picken in plaats van rebase uit te voeren. Bijvoorbeeld, stel dat je een project hebt dat eruit ziet als dit:

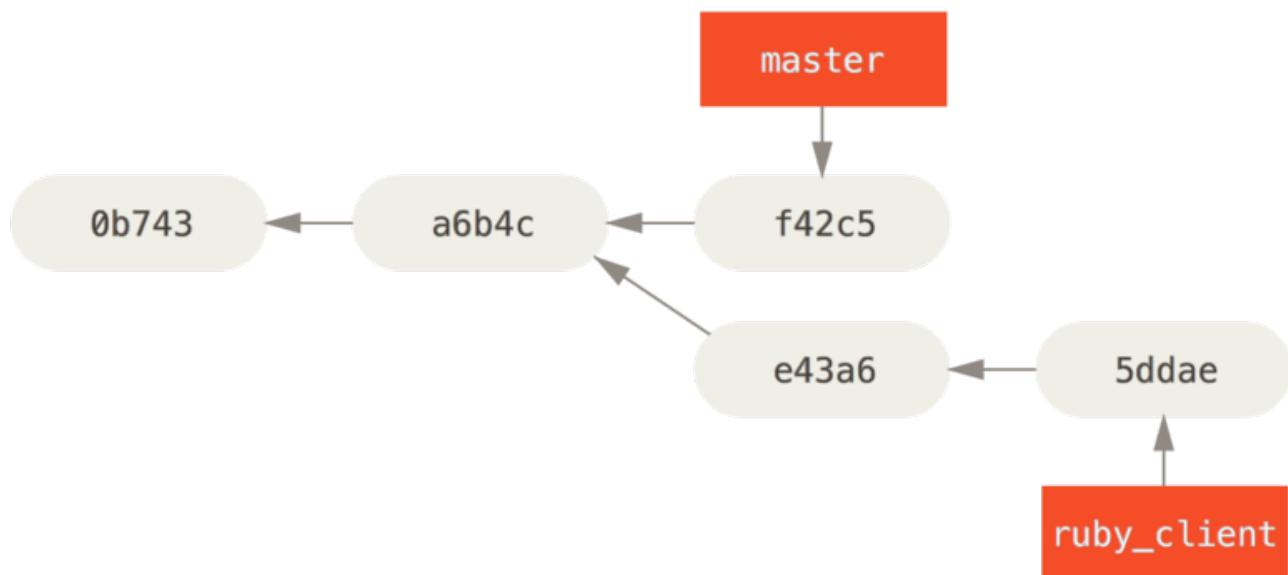


Figure 80. Voorbeeld historie voor een cherry-pick.

Als je commit `e43a6` in je master branch wilt pullen, dan kun je dit uitvoeren

```
$ git cherry-pick e43a6
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
 3 files changed, 17 insertions(+), 3 deletions(-)
```

Dit pullt dezelfde wijziging zoals geïntroduceerd in `e43a6`, maar je krijgt een nieuwe SHA-1 waarde, omdat de toepassingsdatum anders is. Nu ziet je historie er zo uit:

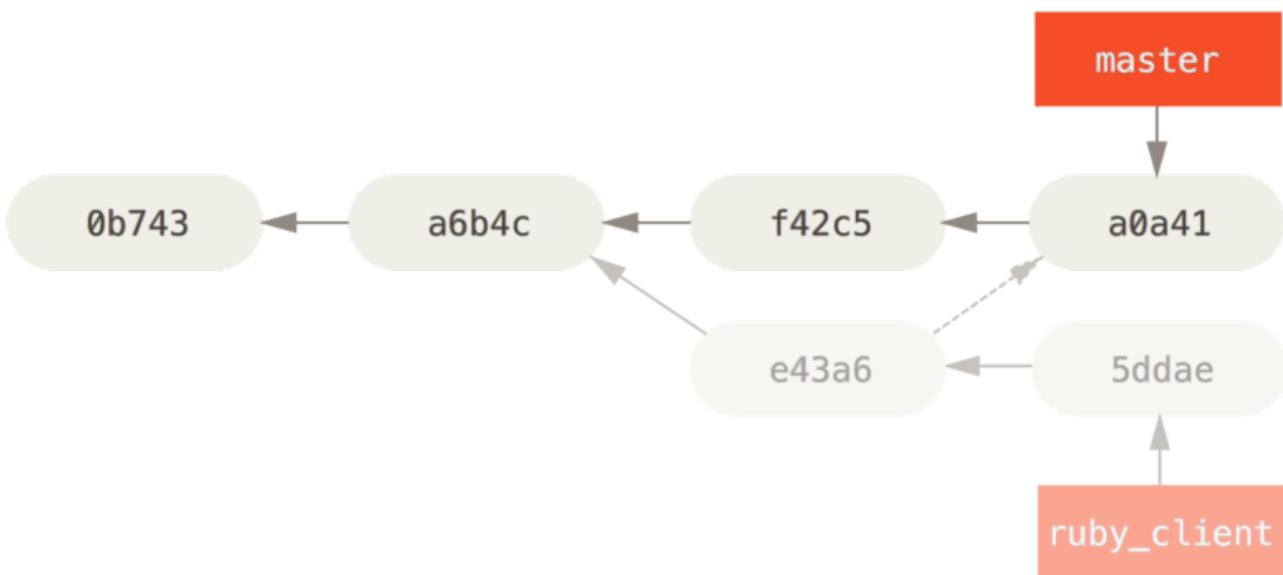


Figure 81. Historie na het cherry-picken van een commit op een topic branch.

Nu kun je de topic branch verwijderen en de commits die je niet wilde pullen weggooien.

## Rerere

Als je veel gaat mergen en rebasen, of als je een langlopende topic branch onderhoudt heeft Git een optie genaamd “rerere” die je kan helpen.

Rerere staat voor “hergebruik het opgeslagen besluit” (Reuse Recorded Resolution) - het is een manier om het andmatige conflict situatie oplossen in te korten. Als rerere is ingeschakeld, zal Git een reeks van ervoor- en erna-plaatjes van succesvolle merges bijhouden en als het opmerkt dat er een conflict is dat precies lijkt op een die je al opgelost hebt, zal het gewoon de oplossing van de vorige keer gebruiken, zonder je ermee lastig te vallen.

Deze optie komt in twee delen: een configuratie instelling en een commando. De configuratie is een `rerere.enabled` instelling, en het is handig genoeg om in je globale configuratie te zetten:

```
$ git config --global rerere.enabled true
```

Vanaf nu, elke keer als je een merge doet dat een conflict oplöst, zal de oplossing opgeslagen worden in de cache voor het geval dat je het in de toekomst nodig gaat hebben.

Als het nodig is kan je interacteren met de rerere cache met het `git rerere` commando. Als het op zich aangeroepen wordt, zal Git de database met oplossingen controleren en probeert een passende te vinden voor alle huidige merge conflicten en deze proberen op te lossen (alhoewel dit automatisch wordt gedaan als `rerere .enabled` op `true` is gezet). Er zijn ook sub-commando's om te zien wat er opgeslagen gaan worden, om specifieke oplossingen uit de cache te verwijderen, en om de gehele cache te legen. We zullen rerere meer gedetailleerd behandelen in [Rerere](#).

## Je releases taggen

Als je hebt besloten om een release te maken, zul je waarschijnlijk een tag willen aanmaken zodat je die release op elk moment in de toekomst opnieuw kunt maken. Je kunt een nieuwe tag maken zoals ik heb beschreven in [Git Basics](#). Als je besluit om de tag als de beheerder te tekenen, dan ziet het taggen er wellicht zo uit:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'  
You need a passphrase to unlock the secret key for  
user: "Scott Chacon <schacon@gmail.com>"  
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Als je tags tekent, dan heb je misschien een probleem om de publieke PGP sleutel, die gebruikt is om de tags te tekenen, te verspreiden. De beheerder van het Git project heeft dit probleem opgelost door hun publieke sleutel als een blob in de repository mee te nemen en een tag te maken die direct naar die inhoud wijst. Om dit te doen kun je opzoeken welke sleutel je wilt gebruiken door `gpg --list-keys` uit te voeren:

```
$ gpg --list-keys  
/Users/schacon/.gnupg/pubring.gpg  
-----  
pub 1024D/F721C45A 2009-02-09 [expires: 2010-02-09]  
uid Scott Chacon <schacon@gmail.com>  
sub 2048g/45D02282 2009-02-09 [expires: 2010-02-09]
```

Daarna kun je de sleutel direct in de Git database importeren, door het te exporteren en te "pipen" naar `git hash-object`, wat een nieuwe blob schrijft in Git met die inhoud en je de SHA-1 van de blob teruggeeft:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin  
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Nu je de inhoud van je sleutel in Git hebt, kun je een tag aanmaken die direct daarnaar wijst door de nieuwe SHA-1 waarde die het `hash-object` commando je gaf te specificeren:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Als je `git push --tags` uitvoert, zal de `maintainer-pgp-pub` tag met iedereen gedeeld worden. Als iemand een tag wil verifiëren, dan kunnen ze jouw PGP sleutel direct importeren door de blob direct uit de database te halen en het in GPG te importeren:

```
$ git show maintainer-pgp-pub | gpg --import
```

Ze kunnen die sleutel gebruiken om al je getekende tags te verifiëren. En als je instructies in het tag

bericht zet, dan zal `git show <tag>` je eindgebruikers meer specifieke instructies geven over tag verificatie.

## Een bouw nummer genereren

Omdat Git geen monotoon oplopende nummers heeft zoals `v123` of iets gelijkwaardigs om bij iedere commit mee te worden genomen, en je een voor mensen leesbare naam wilt hebben bij een commit, kan je `git describe` uitvoeren op die commit. Git geeft je de naam van de dichtstbijzijnde tag met het aantal commits achter die tag en een gedeeltelijke SHA-1 waarde van de commit die je omschrijft:

```
$ git describe master  
v1.6.2-rc1-20-g8c5b85c
```

Op deze manier kun je een snapshot of "build" exporteren en het vernoemen naar iets dat begrijpelijk is voor mensen. Sterker nog: als je Git, gekloont van de Git repository, vanaf broncode gebouwd hebt geeft `git --version` je iets dat er zo uitziet. Als je een commit beschrijft die je direct getagged hebt, dan krijg je de tag naam.

Het `git describe` commando geeft beschreven tags de voorkeur (tags gemaakt met de `-a` of `-s` vlag), dus release tags zouden op deze manier aangemaakt moeten worden als je `git describe` gebruikt, om er zeker van te zijn dat de commit de juiste naam krijgt als het omschreven wordt. Je kunt deze tekst ook gebruiken als het doel van een checkout of show commando, met de aantekening dat het afhankelijk is van de verkorte SHA-1 waarde aan het einde, dus het zou niet voor altijd geldig kunnen blijven. Als voorbeeld, de Linux kernel sprong recentelijk van 8 naar 10 karakters om er zeker van te zijn dat de SHA-1 uniek zijn, oudere `git describe` commando uitvoernamen werden daardoor ongeldig.

## Een release voorbereiden

Nu wil je een build vrijgeven. Een van de dingen die je wilt doen is een archief maken van de laatste snapshot van je code voor de arme stumperds die geen Git gebruiken. Het commando om dit te doen is `git archive`:

```
$ git archive master --prefix='project/' | gzip > 'git describe master'.tar.gz  
$ ls *.tar.gz  
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Als iemand die tarball opent, dan krijgen ze de laatste snapshot van je project onder een project directory. Je kunt op vrijwel dezelfde manier ook een zip archief maken, maar dan door de `format=zip` optie mee te geven aan `git archive`:

```
$ git archive master --prefix='project/' --format=zip > 'git describe master'.zip
```

Je hebt nu een mooie tarball en een zip archief van je project release, die je kunt uploaden naar je website of naar mensen kunt e-mailen.

## De shortlog

De tijd is gekomen om de maillijst met mensen die willen weten wat er gebeurt in je project te mailen. Een prettige manier om een soort van wijzigingsverslag te krijgen van wat er is toegevoegd in je project sinds je laatste release of e-mail is om het `git shortlog` commando te gebruiken. Het vat alle commits samen binnen de grenswaarden die je het geeft. Het volgende, bijvoorbeeld, geeft je een samenvatting van alle commits sinds je vorige release, als je laatste release de naam v1.0.1 had:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
    Add packed-refs annotated tag support.
    Add Grit::Commit#to_patch
    Update version and History.txt
    Remove stray 'puts'
    Make ls_tree ignore nils

Tom Preston-Werner (4):
    fix dates in history
    dynamic version method
    Version bump to 1.0.2
    Regenerated gemspec for version 1.0.2
```

Je krijgt een opgeschoonde samenvatting van alle commits sinds v1.0.1, gegroepeerd op auteur, die je naar je lijst kunt e-mailen.

## Samenvatting

Je zou je nu redelijk op je gemak moeten voelen om aan een project bij te dragen met Git, maar ook om je eigen project te beheren of de bijdragen van andere gebruikers te integreren. Gefeliciteerd, je bent nu een effectieve Git ontwikkelaar! In het volgende hoofdstuk zullen we je vertellen hoe de grootste en meest populaire Git hosting service te gebruiken: GitHub.

# GitHub

GitHub is de grootste host voor Git repositories, en het is het middelpunt van samenwerking voor miljoenen ontwikkelaars en projecten. Het leeuwendeel van alle Git repositories worden op GitHub gehost, en veel open-source projecten gebruiken het voor Git hosting, issue tracking, code review en andere zaken. Dus alhoewel het geen direct onderdeel is van het Git open source project, is er een grote kans dat je op enig moment met GitHub wilt of moet interacteren als je beroepsmatig Git gebruikt.

Dit hoofdstuk bespreekt hoe je GitHub doelmatig gebruiken kunt. We zullen het inschrijven en het beheren van een account, het maken en gebruiken van Git repositories behandelen, veel gebruikte workflows voor het bijdragen aan projecten en het accepteren van bijdragen aan jouw project, de programmatische interface van GitHub bespreken en veel kleine tips geven om je het leven in het algemeen makkelijker te maken.

Als je niet geïnteresseerd bent in het gebruik van GitHub als host voor je eigen projecten of om samen te werken met andere projecten die op GitHub zijn gehost, kan je zonder problemen doorgaan naar [Git Tools](#).

## *Interfaces veranderen*



Het is belangrijk om op te merken dat, zoals veel actieve websites, het onvermijdelijk is dat de gebruikers interface elementen in deze screenshots mettertijd zullen wijzigen. Hopelijk komt het algemene beeld van wat we hier proberen te bereiken nog steeds overeen met de huidige situatie, maar als je meer recente versie van deze schermen wilt, de online versie van dit boek zou nieuwe screenshots kunnen bevatten.

## Account setup en configuratie

Het eerste wat je dient te doen is om een gratis user account aan te maken. Eenvoudigweg <https://github.com> bezoeken, een gebruikersnaam kiezen die nog niet in gebruik is, een email adres en een wachtwoord opgeven, en op de grote groene “Sign up for GitHub” knop klikken.

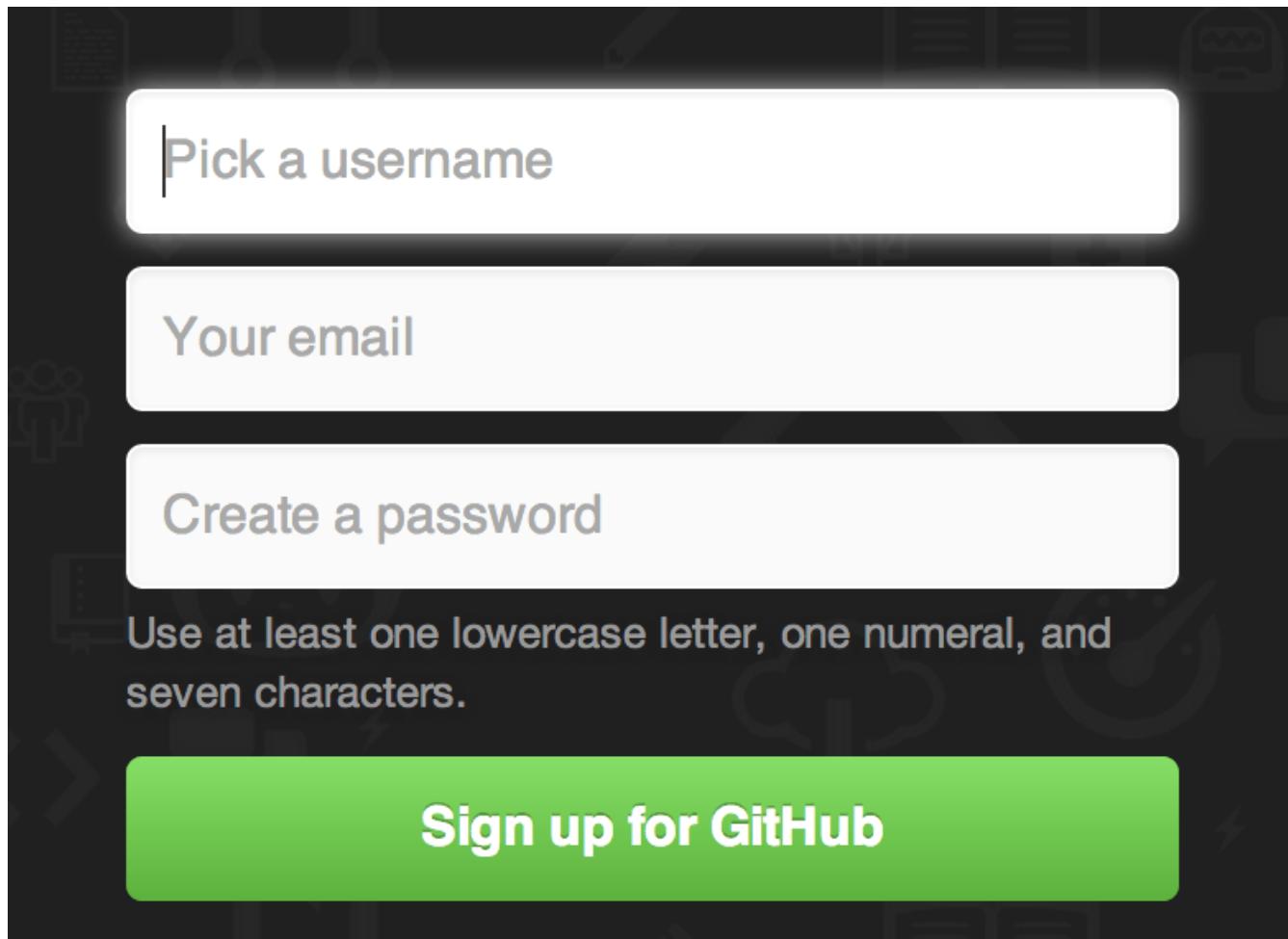


Figure 82. Het GitHub aanmeldings-formulier.

Het volgende wat je zult zien is de prijslijst voor opwaardeer schema's, maar deze kan je op dit moment veilig negeren. GitHub zal je een email sturen om het opgegeven adres te verifiëren. Ga dit nu even doen, het is nogal belangrijk (zoals we straks zullen zien).



GitHub levert al haar functionaliteit met gratis account, met de beperking dat al je projecten compleet openbaar zijn (iedereen heeft leesrechten). De betaalde diensten van GitHub bevatten een aantal besloten projecten, maar we zullen deze in dit boek niet behandelen.

Door op het Octocat logo links boven op het scherm te klikken wordt je naar je dashboard-pagina geleid. Je bent nu klaar om GitHub te gebruiken.

## SSH Toegang

Vanaf nu ben je compleet in staat om contact met Git repositories te maken met het <https://> protocol, je bekend makend met de gebruikersnaam en wachtwoord die je zojuist opgezet hebt. Echter, om eenvoudigweg openbare projecten te klonen hoeft je niet eens in te schrijven - het account dat we zojuist gemaakt hebben gaat een rol spelen als we straks projecten gaan forken en als we naar onze forks gaan pushen.

Als je SSH remotes wilt gebruiken, zal je een publieke sleutel moeten configureren. (Als je er nog geen hebt, zie [Je publieke SSH sleutel genereren](#).) Open je account instellingen met de link rechtsboven op het scherm:

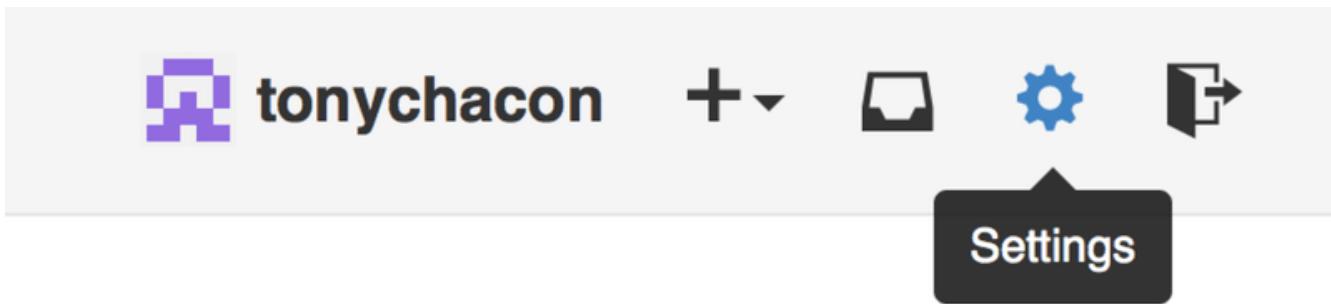


Figure 83. De “Account settings” link.

Selecteer dan de “SSH keys” sectie aan de linker kant.

A screenshot of the "SSH Keys" section on GitHub. On the left, a sidebar shows navigation links: Profile, Account settings (which is selected), Emails, Notification center, Billing, SSH keys (selected), Security, Applications, Repositories, and Organizations. The main area has a heading "SSH Keys" with a sub-section "Need help? Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#)". Below this, a message says "There are no SSH keys with access to your account." A "Add SSH key" button is located in the top right. A modal window titled "Add an SSH Key" is open, containing fields for "Title" (with an empty input field) and "Key" (with a large text area). A green "Add key" button is at the bottom of the modal.

Figure 84. De “SSH keys” link.

Van daar, klik op de “Add an SSH key” knop, geef je sleutel een naam, plak de inhoud van je `~/.ssh/id_rsa.pub` (of hoe je 'm genoemd hebt) public-key bestand in het tekstgebied en klik “Add key”.



Zorg ervoor de je je SSH sleutels een naam geeft die je kunt onthouden. Je kunt elk van je sleutels benoemen (bijv. "Mijn laptop" of "Werk account") zodat je, als je een sleutel moet innemen, je eenvoudig kunt zien welke je moet hebben.

## Jouw avatar

Daarna kan je, als je dat wilt, de gegenereerde avatar vervangen met een afbeelding van jouw keuze. Ga eerst naar de “Profile” tab (boven de SSH Keys tab) en klik op “Upload new picture”.

The screenshot shows the GitHub profile settings interface. On the left, a sidebar lists options: Profile (selected), Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications, Repositories, and Organizations. The main area is titled 'Public profile' and contains fields for 'Profile picture' (with a placeholder image and 'Upload new picture' button), 'Name', 'Email (will be public)', 'URL', 'Company', and 'Location'. At the bottom is a green 'Update profile' button.

Figure 85. De “Profile” link.

We zullen een kopie van het Git logo gebruiken dat op je harde schijf staat en dan krijgen we de kans om het bij te snijden.

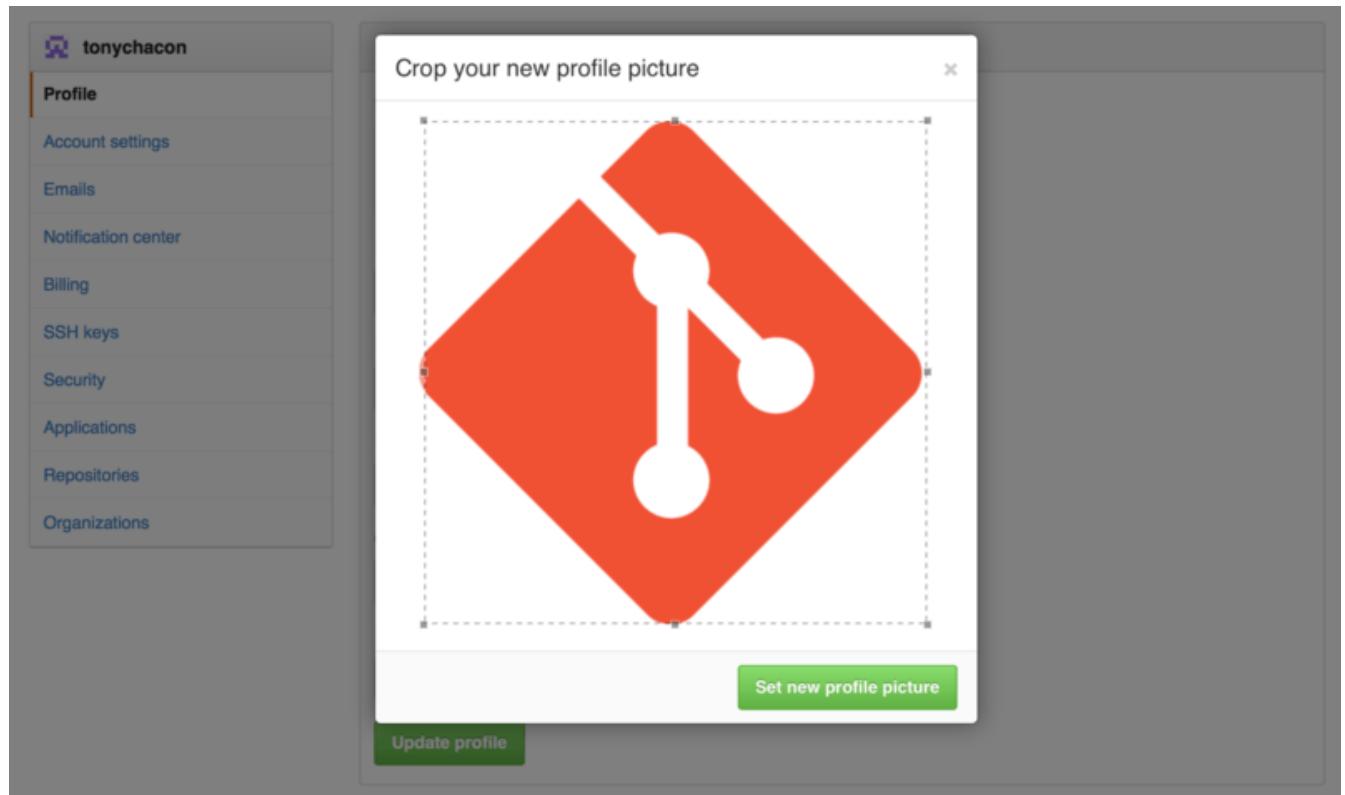


Figure 86. Je avatar bijnissen

Nu zal iedereen jouw avatar naast je gebruikersnaam zien.

Als je toevallig een geüploade avatar op de populaire Gravatar dienst hebt (vaak gebruikt voor

Wordpress accounts), zal die avatar standaard worden gebruikt en hoeft je deze stap niet te doen.

## Je email adressen

De manier waarop GitHub je Git commits koppelt aan je gebruiker is via het email adres. Als je meerdere email adressen gebruikt in je commits en je wilt dat GitHub ze juist koppelt, moet je alle email adressen die je gebruikt hebt toevoegen in het Emails deel van de admin sectie.

The screenshot shows the 'Email' section of the GitHub account settings. On the left, a sidebar lists options like Profile, Account settings, and Emails (which is selected). The main area has a heading 'Email' with a note: 'Your primary GitHub email address will be used for account-related notifications (e.g. account changes and billing receipts) as well as any web-based GitHub operations (e.g. edits and merges)'. It lists three email addresses: 'tonychacon@example.com' (Primary, Public), 'tchacon@example.com' (with a 'Set as primary' button), and 'tony.chacon@example.com' (Unverified, with a 'Send verification email' button). Below is a form to 'Add email address' with an 'Add' button. At the bottom is a checkbox for 'Keep my email address private' with the note: 'We will use [tonychacon@users.noreply.github.com](#) when performing Git operations and sending email on your behalf.'

Figure 87. Voeg email adressen toe

In [Voeg email adressen toe](#) kunnen we een aantal van de verschillende statussen zien die mogelijk zijn. Het bovenste adres is geverifieerd en is gezet als het primaire adres, wat inhoudt dat notificaties en ontvangstbewijzen naar dit adres gestuurd zullen worden. Het tweede adres is geverifieerd en kan dus als primair worden gekozen als je ze zou willen wisselen. Het laatste adres is niet geverifieerd, wat inhoudt dat je deze niet je primaire adres kunt maken. Als GitHub een van deze ziet in commit berichten in een van de repositories op de site zal deze nu aan jouw gebruiker worden gekoppeld.

## Dubbele factor authenticatie

Tot slot, als extra beveiliging, zou je zeker de dubbele factor authenticatie of “2FA” moeten inrichten. Dubbele factor authenticatie is een authenticatie mechanisme die de laatste tijd steeds populairder wordt om het risico te ondervangen dat jouw account wordt misbruikt als je wachtwoord op de een of andere manier wordt gestolen. Door dit aan te zetten zal GitHub je naar twee manieren van authenticeren vragen, zodat wanneer een van deze gecompromitteerd is een aanvaller niet in staat zal zijn je account te gebruiken.

Je kunt de dubbele factor authenticatie instelling vinden onder de Security tab van je Account instellingen.

The screenshot shows the GitHub 'Security' tab for the user 'tonychacon'. On the left, a sidebar lists account settings like Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security (which is selected), Applications, Repositories, and Organizations. The main area has a 'Two-factor authentication' section with a status of 'Off' and a 'Set up two-factor authentication' button. A note explains that 2FA provides another layer of security. Below this is a 'Sessions' section listing a current session from a computer in Paris (IP 85.168.227.34) using Safari on OS X 10.9.4, located in Paris, Ile-de-France, France, signed in on September 30, 2014.

Figure 88. 2FA in de Security Tab

Als je de “Set up two-factor authentication”-knop klikt, zal dit je naar een configuratie-pagina leiden waar je kunt kiezen om een telefoon app te gebruiken om een tweede code te genereren (een “time based one-time password” - een tijdsgerelateerde eenmalig wachtwoord), of je kunt GitHub elke keer als je moet inloggen een code laten SMSsen.

Nadat je een voorkeursmethode hebt gekozen en de instructies volgt voor het instellen van 2FA, zal je account iets veiliger zijn en zal je een aanvullende code moeten opgeven bij je wachtwoord elke keer als je in GitHub inlogt.

## Aan een project bijdragen

Nu het account is ingericht, laten we eens door de details lopen die je kunnen helpen bij het bijdragen bij bestaande projecten.

### Projecten afsplitsen (forken)

Als je wilt bijdragen aan een bestaand project waar je geen push toegang tot hebt, kan je het project “forken”. Dat houdt in dat GitHub je een kopie laat maken van het project die geheel van jouw is; het bestaat in de namespace van jouw gebruiker en jij kunt ernaar pushen.



Historisch gezien is de term “fork” een beetje negatief in context, in die zin dat iemand een open source project in een andere richting leidde, soms een concurrerend project makend en de bijdragers onderling verdeelde. In GitHub is een “fork” eenvoudigweg hetzelfde project in jouw namespace, wat jou toestaat om wijzigingen aan een project openbaar te maken met als doel om op een meer open manier bij te dragen.

Op deze manier hoeven projecten zich geen zorgen te maken over gebruikers als bijdragers toe te voegen om ze push-toegang te geven. Mensen kunnen een project forken, ernaar pushen, en hun

wijzigingen terug naar de oorspronkelijke project bij te dragen door een zogenoemde Pull Request te maken, wat we straks zullen behandelen. Dit opent een discussie *thread* met code review en de eigenaar en de bijdrager kunnen dan over de wijziging communiceren totdat de eigenaar ermee tevreden is, op welk moment de eigenaar deze kan mergen.

Om een project te forken, bezoek je de projectpagina en klikt op de “Fork” knop rechtboven op de pagina.



Figure 89. De “Fork” knop.

Na enkele seconden zal je naar jouw nieuwe projectpagina worden geleid, met je eigen schrijfbare kopie van de code.

## De GitHub flow

GitHub is ontworpen rond een specifieke samenwerkings workflow die draait om pull-verzoeken (Pull Requests). Deze workflow werkt of je nu samenwerkt met een hecht team in een enkel gedeelde repository of een bedrijf dat wereldwijd verspreid is of een netwerk van onbekenden die bijdragen aan een project door middel van vele forks. Het is gericht op de [Topic branches](#) workflow die behandeld is in [Branchen in Git](#).

Hier is hoe het over het algemeen werkt:

1. Fork het project
2. Maak een topic branch van [master](#).
3. Doe een aantal commits om het project te verbeteren.
4. Push deze branch naar jouw GitHub project.
5. Open een Pull Request op GitHub.
6. Bespreek en blijf committen zo je wilt.
7. De project eigenaar merget of sluit de Pull Request.

Dit is eigenlijk de Integratie Manager workflow zoals deze behandeld is in [Integratie-manager workflow](#), maar in plaats van mail te gebruiken om te communiceren en wijzigingen te reviewen, gebruiken teams de web-gebaseerde instrumenten van GitHub.

Laten we eens een voorbeeld bespreken van een voorstel tot wijziging aan een open source project die op GitHub gehost wordt die deze workflow gebruikt.

### Een Pull Request maken

Tony is op zoek naar code om op zijn Arduino programmeerbare microcontroller te draaien en heeft een fantastisch project gevonden op GitHub op <https://github.com/schacon/blink>.

```
/*
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
}
```

Figure 90. Het project waar we aan willen bijdragen.

Het enige probleem is dat het lichtje te snel knippert, we vinden dat het veel beter is als we 3 seconden wachten in plaats van 1 tussen elke status wijziging. Laten we dus het programma verbeteren en het terugsturen aan het project als een wijzigingsvoorstel.

Eerst klikken we de *Fork* knop zoals eerder gezegd om onze eigen kopie van het project te krijgen. Onze gebruikersnaam is in dit geval “tonychacon” dus onze kopie van dit project is op <https://github.com/tonychacon/blink> en dat is waar we het kunnen wijzigen. We clonen het lokaal, maken een topic branch en doen de codewijziging en tot slot pushen we de wijziging weer naar GitHub.

```

$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino (macOS) ③
# If you're on a Linux system, do this instead:
# $ sed -i 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
    [-delay(1000);-]{+delay(3000);+} // wait for a second
    digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
    [-delay(1000);-]{+delay(3000);+} // wait for a second
}

$ git commit -a -m 'three seconds is better' ⑤
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink

```

① Kloon onze fork van het project lokaal

② Maak een omschrijvende topic branch

③ Voer onze wijzigingen aan de code uit

④ Controleer dat de wijziging juist is

⑤ Commit de wijziging naar de topic branch

⑥ Push onze nieuwe topic branch terug naar onze GitHub fork

Als we nu teruggaan naar onze fork op GitHub, kunnen we zien dat GitHub heeft opgemerkt dat we

een nieuwe topic branch hebben gepusht en laat ons een grote groene knop zien om onze wijzigingen te bekijken en een Pull Request te openen naar het oorspronkelijke project.

Als alternatief zou je naar de “Branches” pagina kunnen gaan op <https://github.com/<user>/<project>/branches> en jouw branch opzoeken en een Pull Request vanuit die locatie openen.

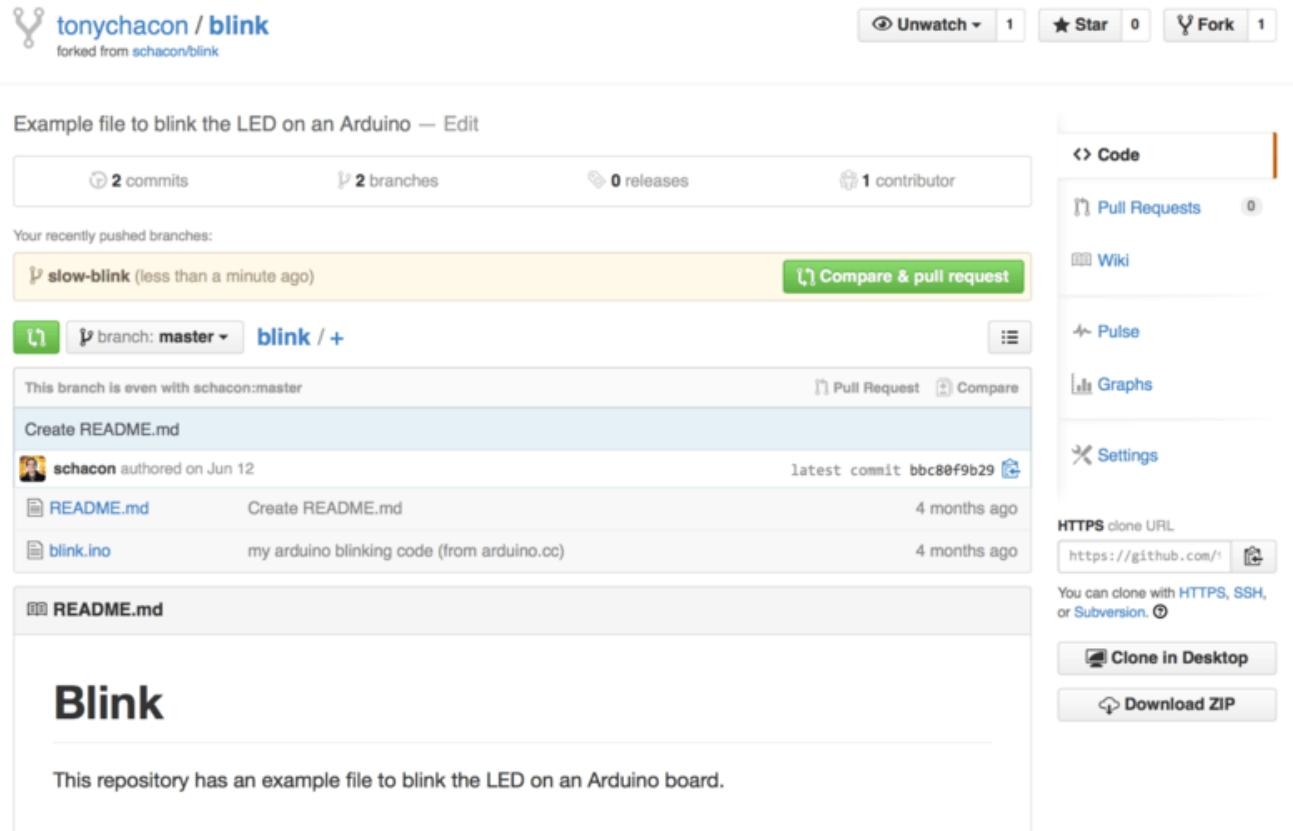
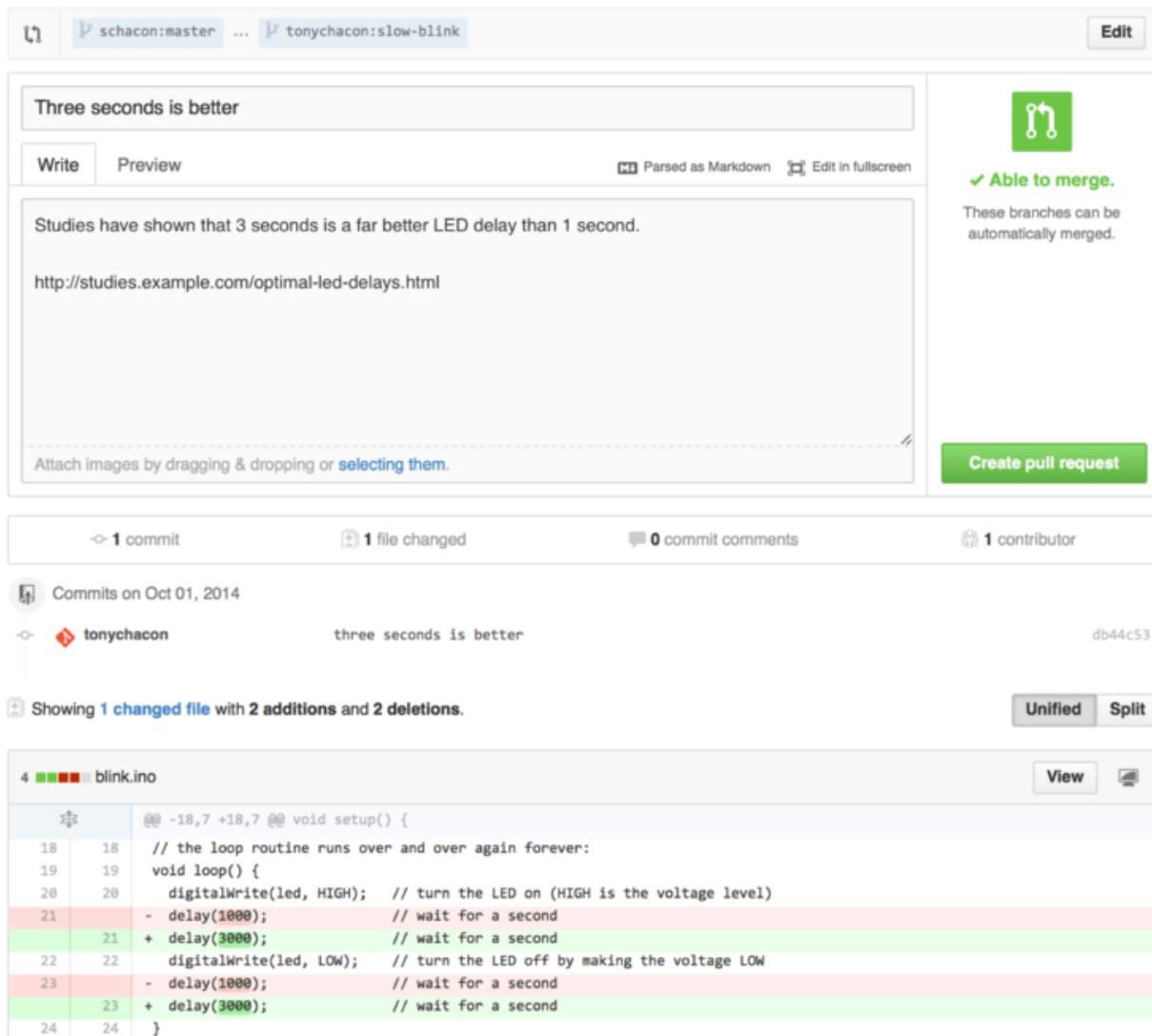


Figure 91. Pull Request knop

Als we op die groene knop klikken, zullen we een scherm zien die ons in staat stelt een titel en een omschrijving voor de wijziging die we willen aanvragen. Het is over het algemeen een goed idee om wat moeite te steken in het maken van een zo goed mogelijke omschrijving zodat de eigenaar van het originele project weet waarom dit wordt gesuggereert, dat je wijziging correct is, en waarom het een waardevolle wijziging is als deze wordt geaccepteerd.

We zien ook een lijst van de commits in onze topic branch die “voorlopen” op de **master**-branch (in dit geval, alleen deze ene) en een *unified diff* van alle wijzigingen die gemaakt zullen worden als deze branch gemerged gaat worden door de project eigenaar.



The screenshot shows a GitHub pull request page for the repository 'tonychacon / blink'. The branch being reviewed is 'tonychacon:slow-blink' against the 'schacon:master' branch. The pull request title is 'Three seconds is better'.

The main content area contains a single commit message:

```
Studies have shown that 3 seconds is a far better LED delay than 1 second.  
  
http://studies.example.com/optimal-led-delays.html
```

Below the commit message is a note: "Attach images by dragging & dropping or [selecting them](#)".

On the right side of the page, there is a green icon with two interlocking gears and the text "Able to merge." followed by the subtext "These branches can be automatically merged." A large green "Create pull request" button is also present.

At the bottom of the pull request page, there is a summary of the changes:

- 1 commit
- 1 file changed
- 0 commit comments
- 1 contributor

The commit details show a single commit from 'tonychacon' on Oct 01, 2014, with the commit hash 'db44c53'. The commit message is 'three seconds is better'.

The code diff section shows the changes made to the 'blink.ino' file:

```
diff --git a/blink.ino b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
 18   18   // the loop routine runs over and over again forever:
 19   19   void loop() {
 20     20     digitalWrite(led, HIGH);    // turn the LED on (HIGH is the voltage level)
 21     21     - delay(1000);          // wait for a second
 22     22     + delay(3000);          // wait for a second
 23     23     digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
 24     24     - delay(1000);          // wait for a second
 25     25     + delay(3000);          // wait for a second
 26     26 }
```

Below the code diff, there are 'Unified' and 'Split' view options.

Figure 92. Pull Request aanmaak pagina

Als je op de *Create pull request* knop drukt op deze pagina, zal de eigenaar van het project waar jij vanaf hebt geforked een berichtje krijgen dat iemand een wijziging voorstelt en zal naar een pagina verwijzen waar al deze informatie op vermeld staat.



Alhoewel Pull Requests gewoonlijk gebruikt worden voor openbare projecten zoals deze als de bijdrager een volledige wijziging klaar heeft staan, is het ook vaak gebruikt in interne projecten *aan het begin* van de ontwikkel-cyclus. Omdat je kunt blijven pushen naar de topic branch zelfs **nadat** de Pull Request is geopend, wordt deze vaak vroeg geopend en gebruikt als een manier om op werk te itereren als een team binnen een context, in plaats van te worden geopend helemaal aan het eind van het proces.

## Iteraties op een Pull Request

Op dit moment kan de project eigenaar naar de voorgedragen wijziging kijken en deze mergen, afwijzen of er op reageren. Laten we zeggen dat het idee hem aanspreekt, maar dat hij het lichtje iets langer uit wil hebben dan aan.

Waar deze discussie via mail zou kunnen plaatsvinden in de workflows die we hebben laten zien in [Gedistribueerd Git](#), heeft het bij GitHub online plaats. De project eigenaar kan de unified diff bekijken en een commentaar achterlaten door op een of meer regels te klikken.

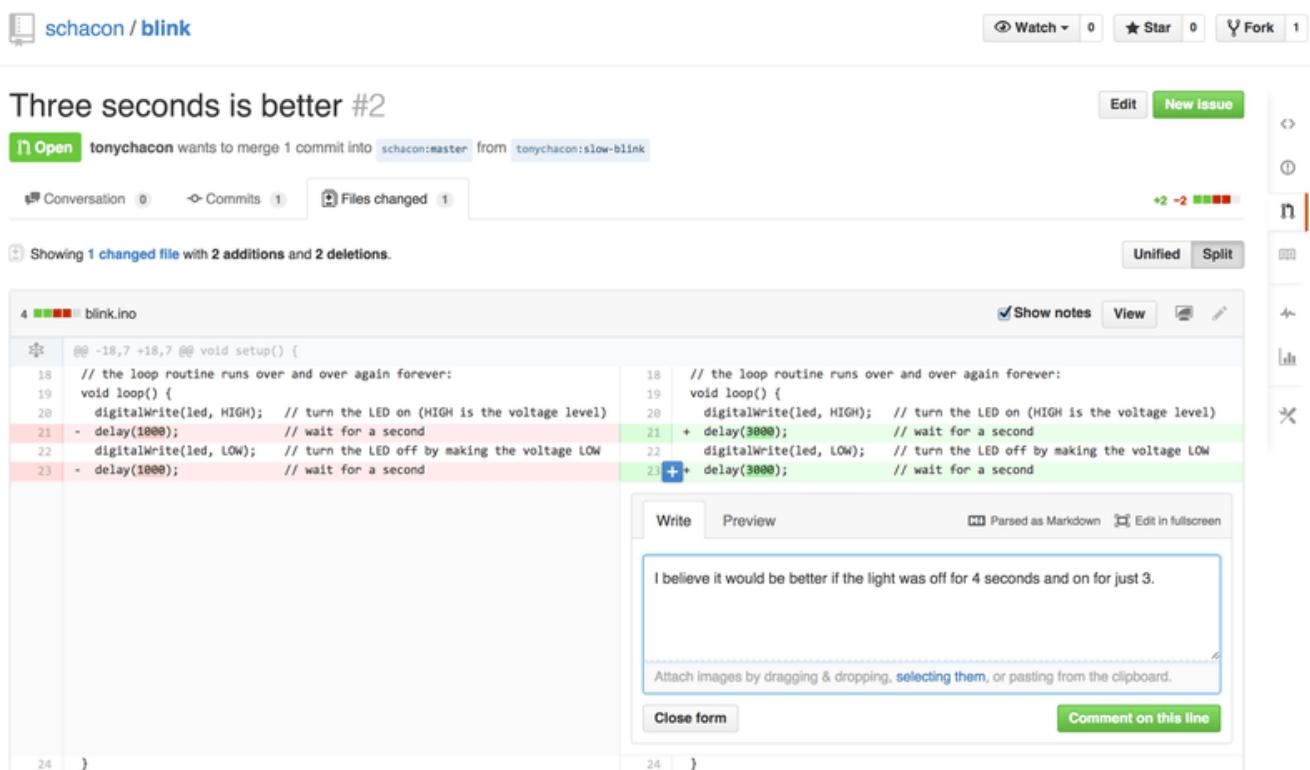


Figure 93. Commentariëren op een specifieke code regel in een Pull Request

Als de onderhouder dit commentaar eenmaal heeft gemaakt zal de persoon die de Pull Request heeft geopend (en verder iedereen die deze repository volgt) een berichtje krijgen. We zullen straks de manier waarop dit kan worden aangepast behandelen, maar als hij email notificaties aan heeft staan, zou Tony een mail zoals deze krijgen:



Figure 94. Commentaar verstuurd als email notificaties

Het is iedereen toegestaan om algemene commentaren op het Pull Request te maken. In [Pull Request discussie pagina](#) kunnen we een voorbeeld zien van de project eigenaar die zowel een regel code becommentariëert en daarna een algemeen commentaar achterlaat in het discussie gedeelte. Je kunt zien dat de code commentaren ook in de conversatie worden gevoegd.

## Three seconds is better #2

Edit New issue

Open tonychacon wants to merge 1 commit into schacon:master from tonychacon:slow-blink

Conversation 1 Commits 1 Files changed 1 +2 -2

tonychacon commented 6 minutes ago  
Studies have shown that 3 seconds is a far better LED delay than 1 second.  
<http://studies.example.com/optimal-led-delays.html>

three seconds is better db44c53

schacon commented on the diff just now

blink.ino View full changes

```
22 22 digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
23 - delay(1000); // wait for a second
23 + delay(3000); // wait for a second
```

schacon added a note just now  
I believe it would be better if the light was off for 4 seconds and on for just 3.

Add a line note

schacon commented just now  
If you make that change, I'll be happy to merge this.

Labels None yet

Milestone No milestone

Assignee No one—assign yourself

Notifications Unsubscribe You're receiving notifications because you commented.

2 participants

Lock pull request

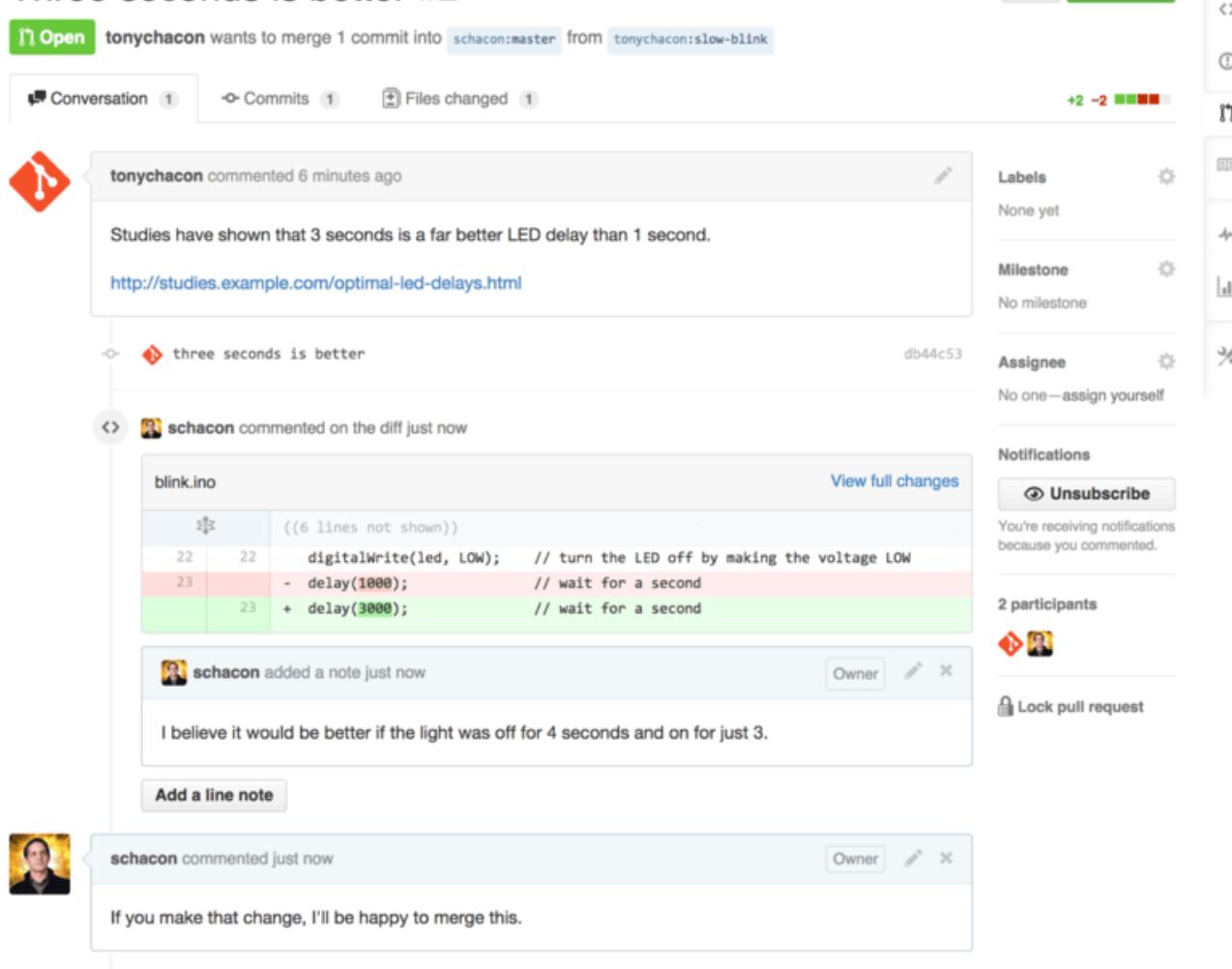


Figure 95. Pull Request discussie pagina

Nu kan de bijdrager zien wat hij moet doen om hun wijziging geaccepteerd te krijgen. Gelukkig is dit ook erg eenvoudig om te doen. Waar je bij email je reeks opnieuw moet samenstellen en opnieuw submitten naar de mail lijst, hoef je met GitHub alleen opnieuw naar de topic branch te committen en te pushen. In [Pull Request final](#) is te zien dat de oude code commentaar is ingeklappt in de bijgewerkte Pull Request, omdat deze is gemaakt dat sindsdien is gewijzigd.

Het toevoegen van commits in een bestaande Pull Request veroorzaakt geen notificatie, dus als Tony eenmaal zijn correcties heeft gepusht besluit hij om een commentaar achter te laten om de project eigenaar te informeren dat hij de gevraagde wijziging gemaakt heeft.

## Three seconds is better #2

The screenshot shows a GitHub pull request interface. At the top, a green button says "Open" and indicates "tonychacon wants to merge 3 commits into schacon:master from tonychacon:slow-blink". Below this, there are tabs for "Conversation" (3), "Commits" (3), and "Files changed" (1). The main area shows a conversation between tonychacon and schacon. tonychacon comments: "Studies have shown that 3 seconds is a far better LED delay than 1 second. http://studies.example.com/optimal-led-delays.html". schacon replies: "If you make that change, I'll be happy to merge this.". tonychacon adds some commits: "longer off time" and "remove trailing whitespace". In the final message, tonychacon says: "I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?". A summary at the bottom states: "This pull request can be automatically merged. You can also merge branches on the command line." and includes a "Merge pull request" button.

Figure 96. Pull Request final

Een interessant iets om op te merken is dat wanneer je de "Files Changed" tab klikt op deze Pull Request, je de "unified" diff krijgt — daarmee wordt het uiteindelijke geaggregeerde verschil bedoeld die geïntroduceerd wordt met je hoofd branch als deze topic branch zou zijn gerged. In `git diff` terminologie, het laat feitelijk automatisch de `git diff master...<branch>` zien voor de branch waar deze Pull Request op is gebaseerd. Zie [Bepalen wat geïntroduceerd is geworden](#) voor meer informatie over dit type diff.

Het andere wat je zult zien is dat GitHub controleert of de Pull Request goed zou mergen en een knop biedt om de merge voor je te doen op de server. Deze knop krijg je alleen te zien als je schrijfrechten hebt op de repository en een triviale merge mogelijk is. Als je de knop klikt zal GitHub een "non-fast-forward" merge uitvoeren, wat inhoudt dat zelfs als de merge een fast-forward **zou kunnen** zijn, het nog steeds een merge commit maakt.

Als je dat liever hebt, kan je de branch eenvoudigweg pullen en het lokaal mergen. Als je deze branch merged in de `master`-branch en deze naar GitHub pusht, wordt de Pull Request automatisch gesloten.

Dit is de eenvoudige workflow dat de meeste GitHub projecten gebruiken. Topic branches worden gemaakt, Pull Requests worden hierop geopend, een discussie volgt, mogelijk wordt er meer werk op de branch gedaan en uiteindelijk wordt het request gesloten of gemerged.

#### *Niet alleen forks*



Het is belangrijk op te merken dat je ook een Pull Request kunt openen tussen twee branches in dezelfde repository. Als je met iemand samenwerkt aan een feature en je hebt beiden schrijfrechten op het project, kan je een topic branch pushen naar de repository en een Pull Request openen naar de `master`-branch van hetzelfde project om het code review en discussie proces te starten. Forken is niet noodzakelijk.

## Pull Requests voor gevorderden

Nu we de grondbeginselen van bijdragen aan een project op GitHub hebben behandeld, laten we een paar interessante tips en truks zien met betrekking tot Pull Requests zodat je nog effectiever kunt zijn in het gebruik.

### Pull Requests als patches

Het is belangrijk om te begrijpen dat veel projecten Pull Requests niet echt zien als stapels met perfecte patches die altijd netjes achter elkaar zullen kunnen worden toegepast, zoals de meeste maillijst-gebaseerde projecten de reeks bijgedragen patches zien. De meeste GitHub projecten zien Pull Request branches als iteratieve conversaties rond een voorgestelde wijziging, uitmondend in een unified diff die met een merge wordt toegepast.

Dit is een belangrijk onderscheid, omdat de wijziging over het algemeen wordt voorgesteld voordat de code als perfect wordt beschouwd, wat zeldzamer is bij de reeks patch bijdragen in maillijsten. Dit maakt een vroeg gesprek mogelijk met de beheerders zodat het vinden van een goede oplossing meer een inspanning wordt van de hele gemeenschap. Als code wordt voorgesteld met een Pull Request en de beheerders of de gemeenschap een wijziging voorstellen wordt de reeks patches niet opnieuw samengesteld, maar daarentegen wordt het verschil gepusht als een nieuwe commit op de branch, waarbij de conversatie doorgaat met behoud van de context van het vorige werk.

Bijvoorbeeld, als je [Pull Request final](#) erop terugslaat, zal je zien dat de bijdrager zijn commit niet heeft gerebased en een andere Pull Request heeft gestuurd. In plaats daarvan zijn er nieuwe commits toegevoegd en deze zijn naar de bestaande branch gepusht. Op deze manier kan je in de toekomst teruggaan naar deze Pull Request en alle context terugvinden waarop besluiten zijn genomen. De “Merge” knop indrukken op de site maakt opzettelijk een merge commit die aan de Pull Request refereert zodat het eenvoudig is om terug te gaan en de oorspronkelijke conversatie te onderzoeken mocht het nodig zijn.

## Met de upstream bijblijven

Als je Pull Request veroudert raakt of om een andere reden niet schoon merget, zal het willen corrigeren zodat de onderhouder deze eenvoudig kan mergen. GitHub zal dit voor je controleren en je aan de onderkant van elke Pull Request laten weten of de merge triviaal is of niet.

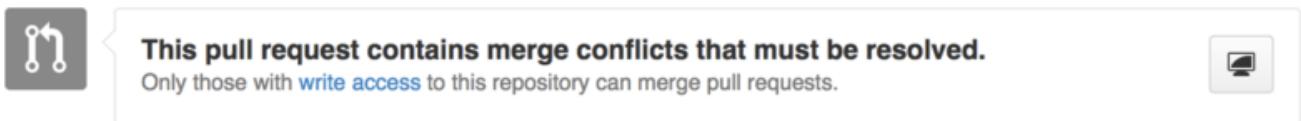


Figure 97. Pull Request zal niet netjes mergen

Als je zoiets als [Pull Request zal niet netjes mergen](#) ziet, zal je jouw branch willen repareren zodat het groen wordt en de onderhouder geen extra werk hoeft te doen.

Je hebt twee voor de hand liggende opties om dit te doen. Je kunt je branch rebasen op waar de target branch is (normaalgesproken de `master`-branch van de repository die je hebt geforked), of je kunt de target branch in je eigen branch mergen.

De meeste ontwikkelaars op GitHub zullen het laatste doen, omdat dezelfde redenen die we behandeld hebben in de vorige paragraaf. Waar het om draait is de historie en de laatste merge, dus rebasen geeft je niet veel meer dan een enigzins schonere historie en is aan de andere kant **veel** moeilijker en foutgevoeliger.

Als je de target branch wilt mergen om je Pull Request merge-baar te maken, moet je de oorspronkelijke repository als een nieuwe remote moeten toevoegen, ervan fetchen, de hoofdbranch van die repository in jouw topic branch mergen, de problemen oplossen als ze er zijn en daarna je topic branch weer terugpushen naar dezelfde branch als waar je de Pull Request op geopend hebt.

Als voorbeeld, stel dat in het “tonychacon” voorbeeld dat we hiervoor gebruikt hebben, de oorspronkelijke auteur een wijziging gemaakt heeft die een conflict in het Pull Request veroorzaakt. Laten we de stappen eens doorlopen.

```

$ git remote add upstream https://github.com/schacon/blink ①

$ git fetch upstream ②
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (3/3), done.
Unpacking objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
From https://github.com/schacon/blink
 * [new branch]      master      -> upstream/master

$ git merge upstream/master ③
Auto-merging blink.ino
CONFLICT (content): Merge conflict in blink.ino
Automatic merge failed; fix conflicts and then commit the result.

$ vim blink.ino ④
$ git add blink.ino
$ git commit
[slow-blink 3c8d735] Merge remote-tracking branch 'upstream/master' \
  into slower-blink

$ git push origin slow-blink ⑤
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 682 bytes | 0 bytes/s, done.
Total 6 (delta 2), reused 0 (delta 0)
To https://github.com/tonychacon/blink
  ef4725c..3c8d735  slower-blink -> slow-blink

```

① Voeg de oorspronkelijke repository als remote toe met de naam “upstream”

② Fetch het nieuwste werk van die remote

③ Merge de main branch in jouw topic branch

④ Los het conflict op dat optrad

⑤ Push naar dezelfde topic branch

Als je dat gedaan hebt zal de Pull Request automatisch geupdate worden en opnieuw gecontroleerd of het zuiver merget.

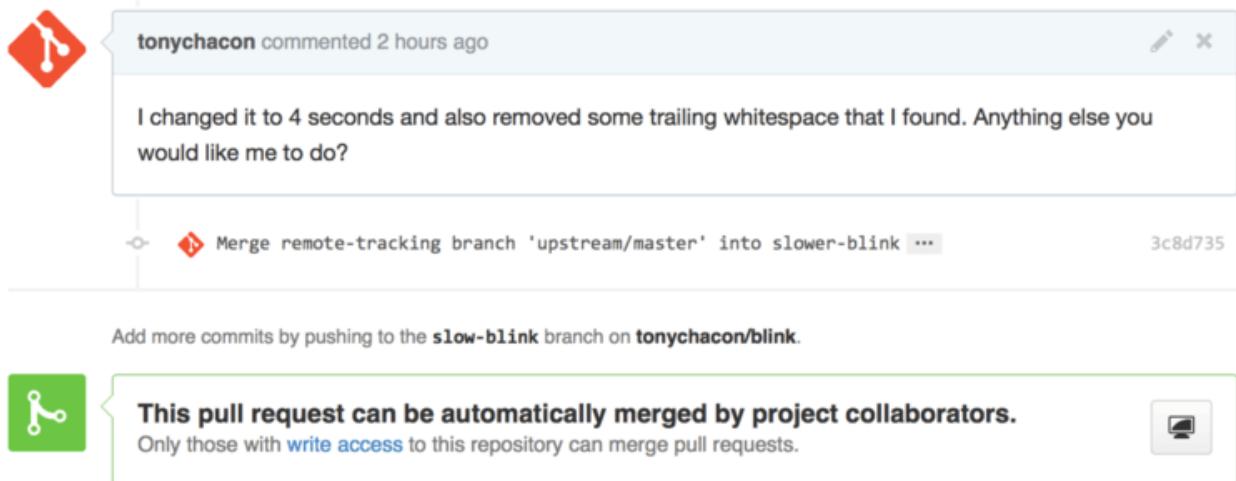


Figure 98. Pull Request merge goed

Een van de geweldige dingen aan Git is dat je dat constant kunt blijven doen. Als je een erg langlopend project hebt, kan je eenvoudig keer op keer de target branch mergen en hoef je alleen de conflicten op te lossen die zijn opgetreden sinds de laatste keer dat je gemerged hebt, wat het proces erg beheerbaar maakt.

Als je perse de branch wilt rebasen om het op te schonen, kan je dat zeker doen, maar het wordt je sterk aangeraden om niet te force pushen naar de branch waar al een Pull Request op is geopend. Als andere mensen deze hebben gepulld en er op zijn gaan doorwerken, krijg je te maken met alle problemen die zijn genoemd in [De gevaren van rebasen](#). In plaats daarvan push je de rebased branch naar een nieuwe branch op GitHub en open je een gloednieuwe Pull Request waarin je aan de oude refereert, en sluit daarna het originele request.

## Referenties

Je volgende vraag zou "Hoe refereert ik aan het oude Pull Request?" kunnen zijn. Er blijken vele, vele manieren te zijn waarop je aan andere dingen kunt refereren zo ongeveer overal waar je kunt schrijven in GitHub.

Laten we beginnen met hoe naar een andere Pull Request of Issue te verwijzen. Alle Pull Requests en Issues hebben een nummer toegewezen gekregen en deze zijn uniek binnen het project. Bijvoorbeeld, je kunt geen Pull Request #3 *en* Issue #3 hebben. Als je aan enig Pull Request of Issue wilt refereren vanuit een andere, kan je eenvoudigweg `#<num>` in elke commentaar of omschrijving neerzetten. Je kunt specifieker zijn als het Issue of Pull Request elders leeft; schrijf `gebruikersnaam#<num>` als je aan een Issue of Pull Request refereert in een fork of repository waar je in zit, of `gebruikersnaam/repo#<num>` om te refereren aan iets in een andere repository.

Laten we naar een voorbeeld kijken. Stel we hebben de branch in het vorige voorbeeld gerebased, een nieuwe pull request ervoor gemaakt en nu willen we verwijzen naar de oude pull request vanuit de nieuwe. We willen ook refereren naar een issue in de fork van de repository in een heel ander project. We maken de beschrijving als in [Verwijzingen in een Pull Request..](#)

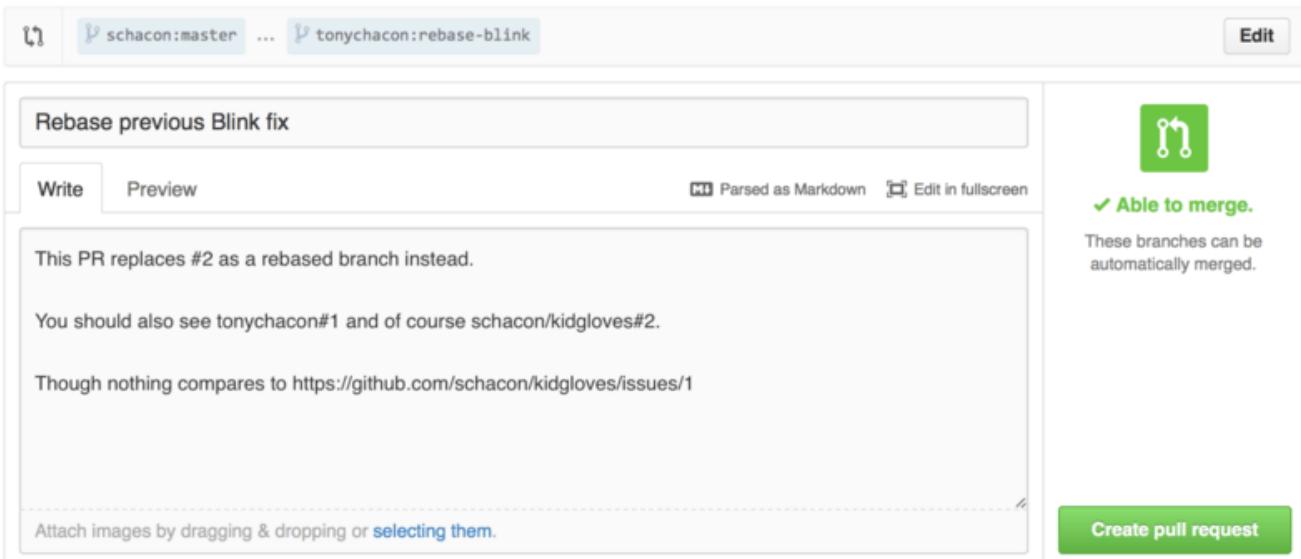


Figure 99. Verwijzingen in een Pull Request.

Als we deze pull request indienen, zien we dit alles getoond als Verwijzingen getoond in een Pull Request..

## Rebase previous Blink fix #4

Figure 100. Verwijzingen getoond in een Pull Request.

Merk op dat de volledige GitHub URL die we erin gezet hebben afgekort is tot alleen de benodigde informatie.

Als Tony nu het orginele Pull Request gaat sluiten, zien we dit doordat we het vermelden in de nieuwe, GitHub heeft automatisch een terugslag gebeurtenis aangemaakt in de tijdlijn van het Pull Request. Dit betekent dat iedereen die dit Pull Request bezoekt en ziet dat het is gesloten eenvoudig kan teruglinken naar degene die het overschrijft. De link zal eruit zien als Verwijzing getoond in een Pull Request..

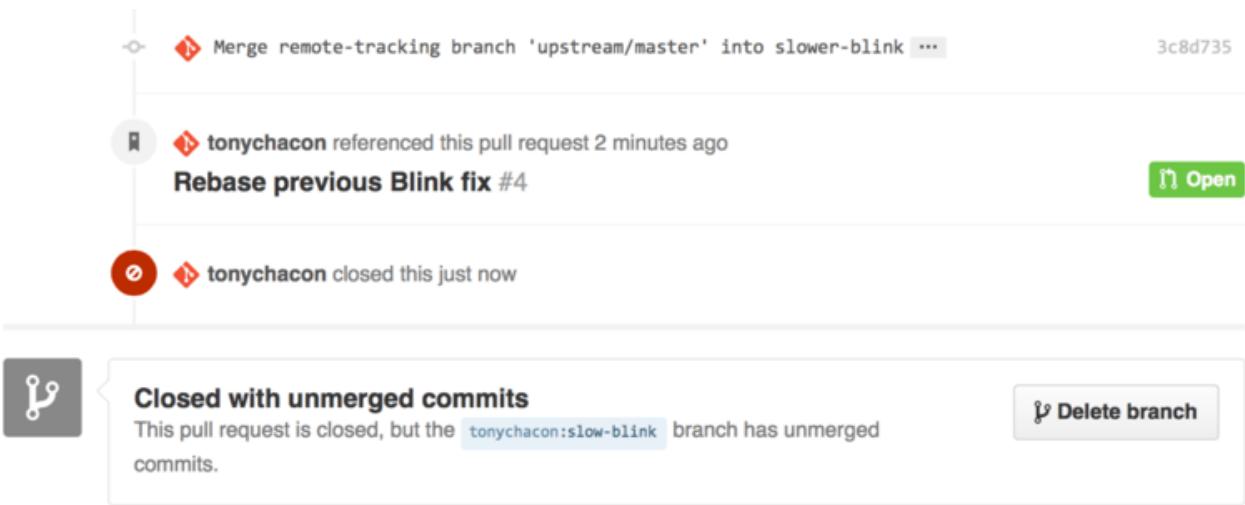


Figure 101. Verwijzing getoond in een Pull Request.

Naast issue nummers, kan je ook refereren aan een specifieke commit door middel van de SHA-1. Je moet een volledige 40 positie SHA-1 vermelden, maar als GitHub dat in een commentaar ziet, zal het direct linken naar de commit. Nogmaals, je kunt aan commits refereren in forks of andere repositories op dezelfde manier als je deed met issues.

## Markdown met een GitHub smaakje

Linken naar andere Issues is maar het begin van de interessante dingen die je met bijna elke tekstbox op GitHub kan doen. In Issue en Pull Request omschrijvingen, commentaren, code commentaren en andere zaken kan je de zogenoemde “GitHub Flavored Markdown” (Markdown met een GitHub smaakje) gebruiken. Markdown is als schrijven in platte tekst maar wat meer functionaliteit wordt getoond.

Zie [Een voorbeeld van Markdown zoals geschreven en getoond](#). voor een voorbeeld van hoe commentaar of tekst kan worden geschreven en daarna getoond met Markdown.

The left side of the image shows a GitHub Markdown editor. It has tabs for "Write" and "Preview", and buttons for "Parsed as Markdown" and "Edit in fullscreen". The preview area shows a sample Markdown document with code snippets, lists, and a Kanye West quote. The right side shows a GitHub comment from "tonychacon" with the text: "There is a big problem with the blink code. Not with the idea, but with the code...". Below this, a section titled "What is the problem?" discusses the number 13 being unlucky and having two decimal places. It includes a quote from Kanye West: "We're living the future so the present is our past." At the bottom right is a large "git" logo with a red diamond icon.

Figure 102. Een voorbeeld van Markdown zoals geschreven en getoond.

Het smaakje wat GitHub aan Markdown meegeeft is meer dan wat je met de standaard Markdown krijgt. Deze smaakjes kunnen alle heel nuttig zijn als je bruikbare Pull Requests of Issue commentaar of omschrijvingen maakt.

## Taaklijsten

Het eerste echt bruikbare GitHub specifieke Markdown optie, vooral in het gebruik in Pull Requests, is de taaklijst. Een taaklijst is een lijst van checkboxen met dingen die je gedaan wilt hebben. Het neerzetten ervan in een Issue of Pull Request geeft normaalgesproken de dingen weer die je gedaan wilt hebben voordat je het onderwerp voor gesloten beschouwt.

Je kunt op deze manier een taaklijst maken:

- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code

Als we deze in de omschrijving van een Pull Request of Issue zetten, zullen we het als [Taaklijsten zoals getoond een Markdown commentaar](#). getoond zien

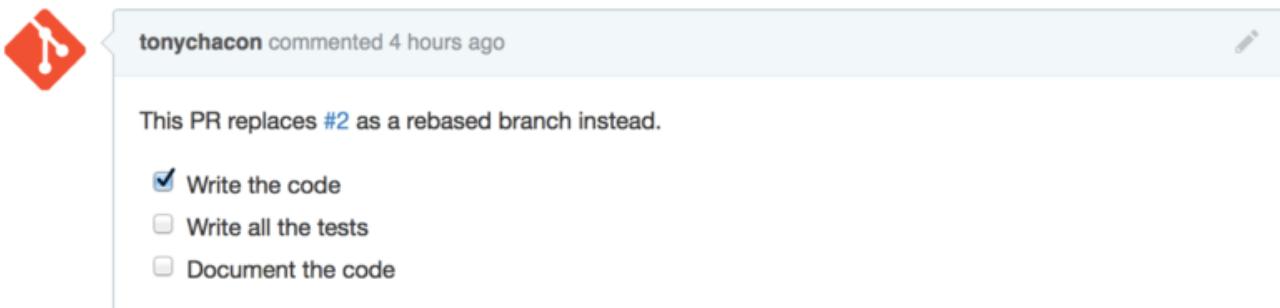


Figure 103. Taaklijsten zoals getoond een Markdown commentaar.

Dit wordt vaak in een Pull Request gebruikt om aan te geven wat je allemaal gedaan wilt zien op de branch voordat de Pull Request klaar is om te mergen. Het echte gave hiervan is dat je eenvoudig op de checkboxen kunt klikken om het commentaar bij te werken—je hoeft de tekst van de Markdown zelf niet te wijzigen om de taken af te tikken.

En er is meer: GitHub zal naar taaklijsten zoeken in je Issues en Pull Requesten en deze als metadata op de pagina tonen die ze bevatten. Bijvoorbeeld, als je een Pull Request hebt met taken en je kijkt naar de overzichtspagina van alle Pull Requesten, kan je zien in hoeverre het klaar is. Dit helpt mensen om Pull Requests in te delen naar subtaken en andere mensen om de voortgang van de branch te volgen. Je kunt een voorbeeld hiervan zien in [Samenvatting van taaklijsten in de Pull Request lijst..](#)

A screenshot of a GitHub pull request list. At the top, it shows '2 Open' and '1 Closed'. Below that, two pull requests are listed:

- #4 Change blink time to four seconds opened 4 hours ago by tonychacon 2 of 3
- #2 Three seconds is better opened 7 hours ago by tonychacon 3

Each pull request has its own summary of the checklist items.

Figure 104. Samenvatting van taaklijsten in de Pull Request lijst.

Dit is ontzettend handig als je vroeg een Pull Request opent en deze gebruikt om de voortgang te

volgen tijdens de implementatie van de feature.

## Code Snippets (code knipsels)

Je kunt ook code knipsels aan commentaren toevoegen. Dit is in het bijzonder handig als je iets wilt voorstellen wat je *zou kunnen* proberen te doen voordat je het daadwerkelijk implementeert als een commit in je branch. Dit wordt ook vaak gebruikt om een voorbeeld te geven de code die niet werkt of wat in deze Pull Request zou kunnen worden geïmplementeerd.

Om een code knipsel toe te voegen moet je het met *backticks* omsluiten.

```
```java
for(int i=0 ; i < 5 ; i++)
{
    System.out.println("i is : " + i);
}
```

```

Als je de naam van de taal toevoegt, zoals we hier met *java* gedaan hebben, zal GitHub proberen ook de syntax te markeren. In het bovenstaande voorbeeld zou het worden getoond als [Getoonde omsloten code voorbeeld..](#)



Figure 105. Getoonde omsloten code voorbeeld.

## Quoting (Citeren)

Als je reageert op een klein deel van een lang commentaar, kan je naar keuze citeren uit het andere commentaar door de regel te laten beginnen met het `>` teken. Dit is zelfs zo gewoon en bruikbaar dat er een sneltoets combinatie voor gemaakt is. Als je de tekst selecteert in het commentaar waar je direct op wilt reageren en de `r` toets indrukt, wordt deze direct voor je als citaat in de commentaar ruimte geplaatst.

De citaten zien er zo ongeveer uit:

```
> Whether 'tis Nobler in the mind to suffer
> The Slings and Arrows of outrageous Fortune,
How big are these slings and in particular, these arrows?
```

Zodra getoond, zal het commentaar er als [Getoond citaat voorbeeld](#). uitzien.

The screenshot shows two GitHub comments. The first comment, by user schacon, contains a sonnet by William Shakespeare:

```
That is the question—  
Whether 'tis Nobler in the mind to suffer  
The Slings and Arrows of outrageous Fortune,  
Or to take Arms against a Sea of troubles,  
And by opposing, end them? To die, to sleep—  
No more; and by a sleep, to say we end  
The Heart-ache, and the thousand Natural shocks  
That Flesh is heir to?
```

The second comment, by user tonychacon, is a reply:

```
Whether 'tis Nobler in the mind to suffer  
The Slings and Arrows of outrageous Fortune,  
  
How big are these slings and in particular, these arrows?
```

Figure 106. Getoond citaat voorbeeld.

## Emoji

Als laatste, kan je ook emoji in je commentaar gebruiken. Dit wordt eigenlijk best wel vaak gebruikt in de commentaren die je bij veel GitHub issues en Pull Requests ziet. Er is zelfs een emoji hulp in GitHub. Als je een commentaar intypt en je begint met een : teken, zal een automatische voltooihulp je komen helpen met vinden wat je zoekt.

The screenshot shows the GitHub emoji suggestion interface. A user has typed ':jo' into the comment input field. A dropdown menu appears with several emoji options:

- joy (laughing face)
- joy\_cat (cat face)
- black\_joker (joker face)
- smile (smiling face)
- smiley (neutral face)

The 'joy\_cat' option is highlighted with a blue background. At the bottom right of the interface are two buttons: 'Close and comment' and a green 'Comment' button.

Figure 107. Emoji voltooi hulp in actie.

Emojis komen er als :<naam>: uit te zien ergens in je commentaar. Je zou bijvoorbeeld iets als dit kunnen schrijven:

```
I :eyes: that :bug: and I :cold_sweat:.  
:trophy: for :microscope: it.  
:+1: and :sparkles: on this :ship:, it's :fire::poop!:!  
:clap::tada::panda_face:
```

Als het wordt getoond, komt het er als **Zwaar emoji commentaar**. uit te zien.

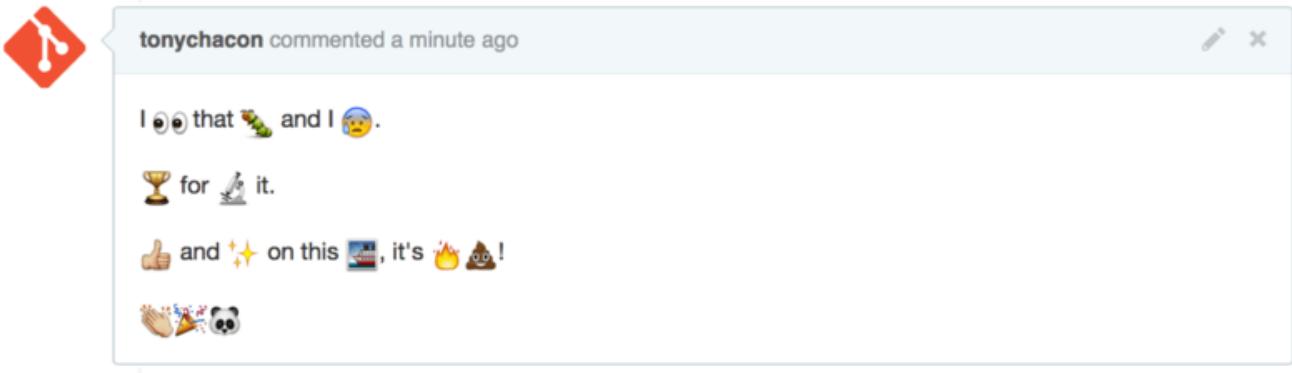


Figure 108. Zwaar emoji commentaar.

Niet dat het erg veel inhoudelijk toevoegt, maar het geeft wel wat sjeu en emotie aan een medium dat normaalgesproken nogal moeilijk emoties laat weergeven.



Er zijn vandaag de dag nogal wat webservices die gebruik maken van emoji. Een goede spiekbrief voor het vinden van emoji die uitdrukken wat je wilt zeggen kan je vinden op:

<http://www.emoji-cheat-sheet.com>

## Plaatjes

Technisch gezien is het geen GitHub smaak van Markdown, maar het is erg handig. Naast het toevoegen van Markdown plaatjes links aan commentaar, waarvan het moeilijk kan zijn om URLs voor te vinden en in te voegen, staat GitHub je toe om plaatjes te slepen en lossen (drag & drop) in tekstgebieden om ze in te voegen.

The image consists of two vertically stacked screenshots of the GitHub 'Write' interface. Both screenshots show a text input field containing the text: 'This is the wrong version of Git for the website:' followed by a placeholder image icon and the file name 'Git.png'. Below the text input is a dashed green border indicating where an image can be dropped or selected. A green 'Comment' button is visible in the bottom right corner of each screenshot.

Figure 109. Slepen en lossen van plaatjes om ze te uploaden en automatisch in te voegen.

Als je terugkijkt naar [Verwijzingen in een Pull Request](#), kan je een kleine “Parsed as Markdown” hint boven het tekstgebied zien. Als je hierop klikt zal dit je een complete spiekbrieven laten zien van alles wat met Markdown in GitHub kunt doen.

## Een project onderhouden

Nu we ons op ons gemak voelen met bij te dragen aan een project, laten we het eens van de andere kant bekijken: je eigen project aanmaken, onderhouden en beheren.

### Een nieuwe repository aanmaken

Laten we eens een nieuwe repository aanmaken waarmee we de code van ons project delen. Begin met het klikken op de “New repository” knop aan de rechterkant van het dashboard, of vanaf de + knop in de bovenste toolbar naast je gebruikersnaam zoals te zien is in [De “New repository” dropdown..](#)

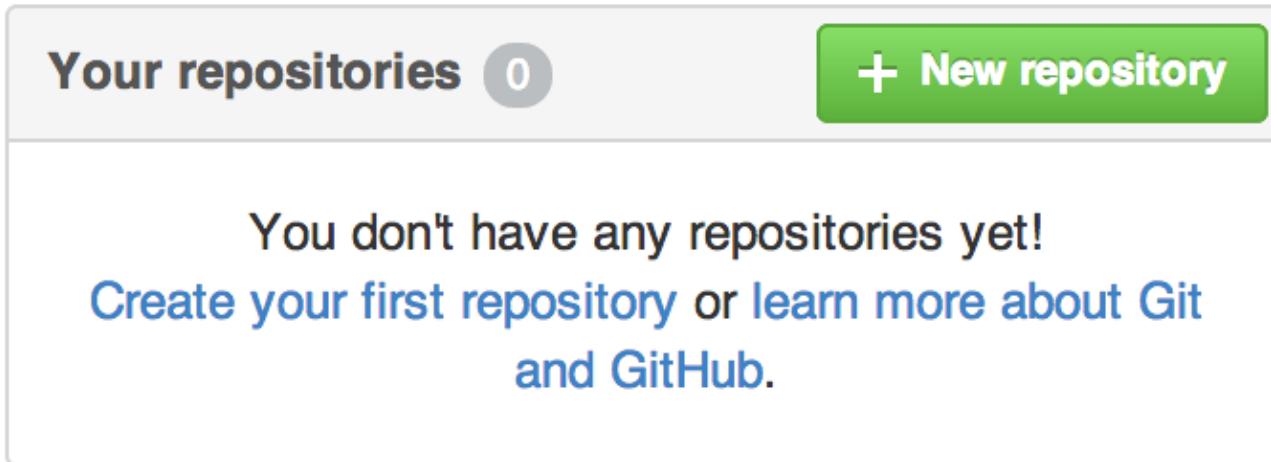


Figure 110. Het “Your repositories” gebied.

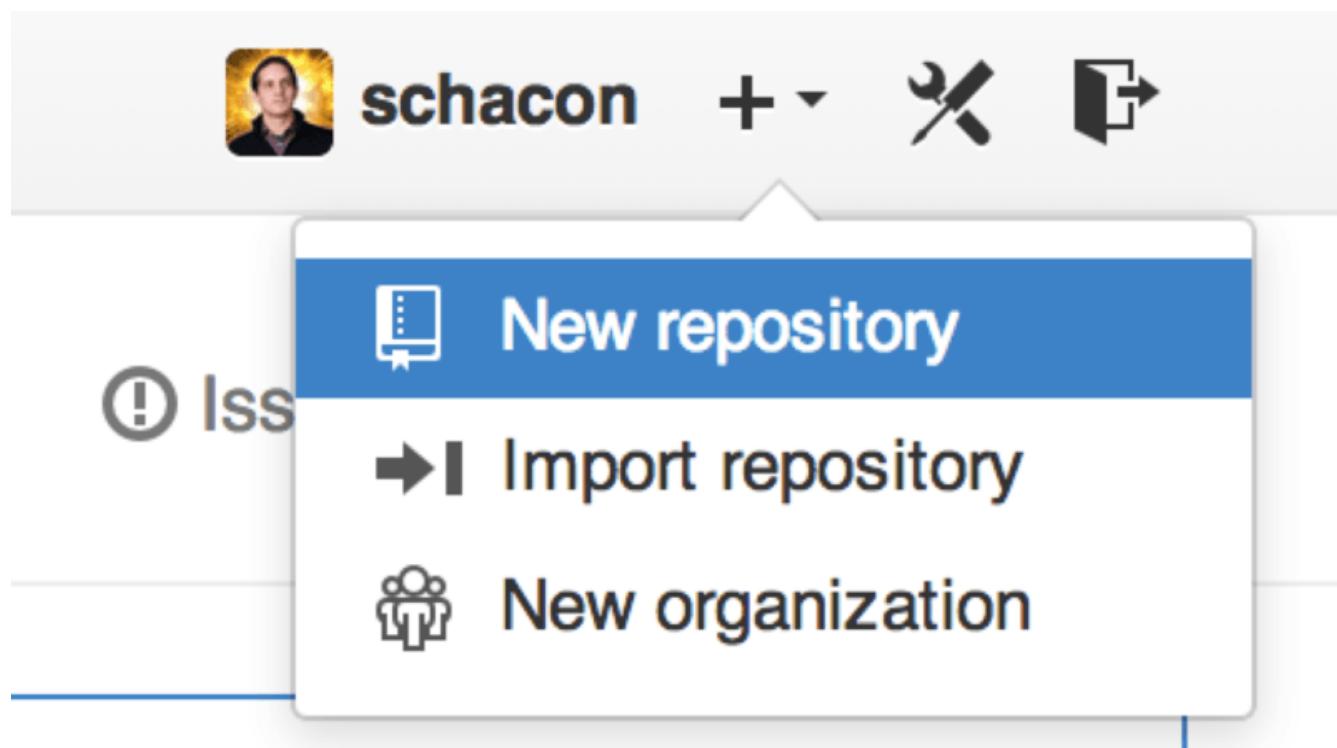


Figure 111. De “New repository” dropdown.

Dit leidt je naar het “new repository” formulier:

Owner: ben / Repository name: iOSApp

Great repository names are short and memorable. Need inspiration? How about [drunken-dubstep](#).

Description (optional): iOS project for our mobile group

**Public**  
Anyone can see this repository. You choose who can commit.

**Private**  
You choose who can see and commit to this repository.

**Initialize this repository with a README**  
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None** | ⓘ

**Create repository**

Figure 112. Het “new repository” formulier.

Alles wat je echt moet doen is je project een naam geven, de overige velden zijn volledig optioneel. Voor nu klik je gewoon de “Create Repository” knop en boem - je hebt een nieuwe repository op GitHub, genaamd `<gebruiker>/<project_naam>`.

Omdat er nog geen code is, zal GitHub je aanwijzigen geven hoe je een gloednieuwe Git repository maakt, of verbindt met een bestaand Git project. We zullen het hier nog niet uitwerken, als je een opfrisser nodig hebt kijk dan nog eens naar [Git Basics](#).

Nu je project op GitHub gehost wordt, kan je het URL aan iedereen geven waarmee je je project wilt delen. Elk project op GitHub is toegankelijk via HTTP als [https://github.com/<gebruiker>/<project\\_naam>](https://github.com/<gebruiker>/<project_naam>), en via SSH als [git@github.com:<gebruiker>/<project\\_naam>](git@github.com:<gebruiker>/<project_naam>). Git kan fetchen van en pushen naar beide URLs, maar toegang wordt bepaald op basis van de gebruikersgegevens van de gebruiker die ze benadert.



Voor openbare projecten wordt vaak de voorkeur gegeven aan het delen middels de HTTP URL, omdat de gebruiker daarvoor niet perse een GitHub account nodig heeft om het te klonen. Gebruikers moeten wel een account hebben en een geüploade SSH sleutel om je project te benaderen als je ze het SSH URL geeft. De HTTPS variant is exact dezelfde URL die ze in hun browser zouden plakken als ze het project daar zouden willen bekijken.

## Medewerkers toevoegen

Als je met andere mensen werkt die je commit toegang wilt geven, moet je ze als “collaborators” toevoegen. Als Ben, Jeff en Louise allemaal GitHub accounts aanvragen, en je wilt ze push toegang geven op jouw repository, kan je ze toevoegen aan je project. Als je dit doet geeft je ze “push” toegang, wat inhoudt dat ze zowel lees als schrijf toegang hebben tot het project en de Git repository.

Klik op de “Settings” link onderaan in de rechter zijkolom.

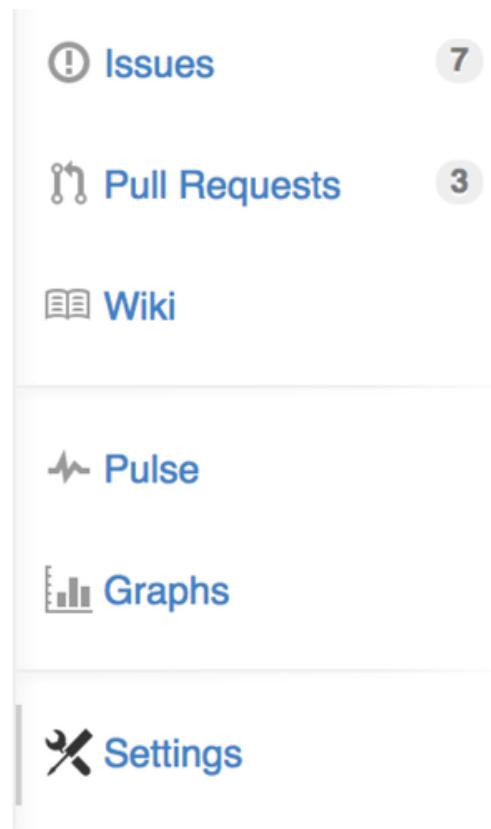


Figure 113. De repository settings link.

Kies daarna “Collaborators” op het menu aan de linker kant. Daarna type je gewoon een gebruikersnaam in het veld en klikt “Add collaborator.” Je kunt dit zo vaak herhalen als je toegang wilt verlenen aan iedereen die je maar wilt. Als je toegang wilt ontzeggen, klik je gewoon de “X” rechts van de betreffende rij.

A screenshot of the GitHub 'Collaborators' page. On the left, there is a sidebar with 'Options' (selected), 'Collaborators' (highlighted with an orange border), 'Webhooks &amp; Services', and 'Deploy keys'. The main area is titled 'Collaborators' and shows three users: Ben Straub (ben), Jeff King (peff), and Louise Corrigan (LouiseCorrigan). Each user has a small profile picture, their name, and their GitHub handle. To the right of the users, the text 'Full access to the repository' is displayed. At the bottom of the list, there is a search bar labeled 'Type a username' and a button labeled 'Add collaborator'.

Figure 114. Repository medewerkers.

## Pull Requests beheren

Nu je een project hebt met wat code erin en misschien zelfs een paar medewerkers die ook push toegang hebben, laten we eens behandelen wat je moet doen als je zelf een Pull Request ontvangt.

Pull Requests kunnen komen van een branch in een fork van jouw repository of ze kunnen komen van een andere branch in dezelfde repository. Het enige verschil is dat ze in een fork vaak van mensen komen waar jij niet naar hun branch kan pushen en zij niet naar de jouwe, terwijl bij

interne Pull Requests beide partijen over het algemeen de branch kunnen benaderen.

Voor de volgende voorbeelden, laten we aannemen dat jij “tonychacon” bent en dat je een nieuwe Arduino code project genaamd “fade” gemaakt hebt.

## Email berichten

Er is nu iemand die een wijziging op je code gemaakt heeft en die je een Pull Request stuurt. Je zou een email bericht moeten krijgen die je over het nieuwe Pull Request bericht en die er ongeveer zo uitziet als [Email bericht van een nieuwe Request..](#)

The screenshot shows an email from GitHub. The subject is "[fade] Wait longer to see the dimming effect better (#1)". The recipient is Scott Chacon <notifications@github.com>. The email was sent at 10:05 AM (0 minutes ago). The message content includes:

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](#).

Figure 115. Email bericht van een nieuwe Request.

Er zijn een aantal dingen te zien aan deze email. Het geeft je een kleine diffstat — een lijst met bestanden die gewijzigd zijn in de Pull Request en hoezeer ze zijn gewijzigd. Het geeft je een link naar de Pull Request op GitHub. Het geeft je ook een aantal URLs die je vanaf de commando regel kunt gebruiken.

Als je de regel bekijkt waar `git pull <url> patch-1` staat, is dit een eenvoudige manier om een remote branch te mergen zonder een remote toe te voegen. We hebben dit kort behandeld in [Remote branches uitchecken](#). Als je wilt, kan je een topic branch maken en ernaar switchen en dit commando aanroepen om de Pull Request wijzigingen erin te mergen.

De andere interessante URLs zijn de `.diff` en `.patch` URLs, die zoals je zult vermoeden, de unified diff en patch versies van de Pull Request geven. Je zou technisch gesproken de Pull Request in je werk mergen met zoiets als dit:

```
$ curl http://github.com/tonychacon/fade/pull/1.patch | git am
```

## Samenwerken op basis van de Pull Request

Zoals behandeld in [De GitHub flow](#), kan je nu een conversatie voeren met de persoon die de Pull Request geopend heeft. Je kunt commentaar geven op specifieke regels code, commentaar geven op hele commits of commentaar geven over de gehele Pull Request; waarbij de GitHub smaak van Markdown overal gebruikt kan worden.

Elke keer als iemand anders commentaar geeft op de Pull Request blijf je e-mails ontvangen zodat je weet dat er activiteit plaatsvindt. Alle betrokkenen krijgen een link naar de Pull Request waar de activiteit op plaatsvindt en je kunt ook direct de mail beantwoorden (reply-to) om commentaar op de Pull Request conversatie te geven.

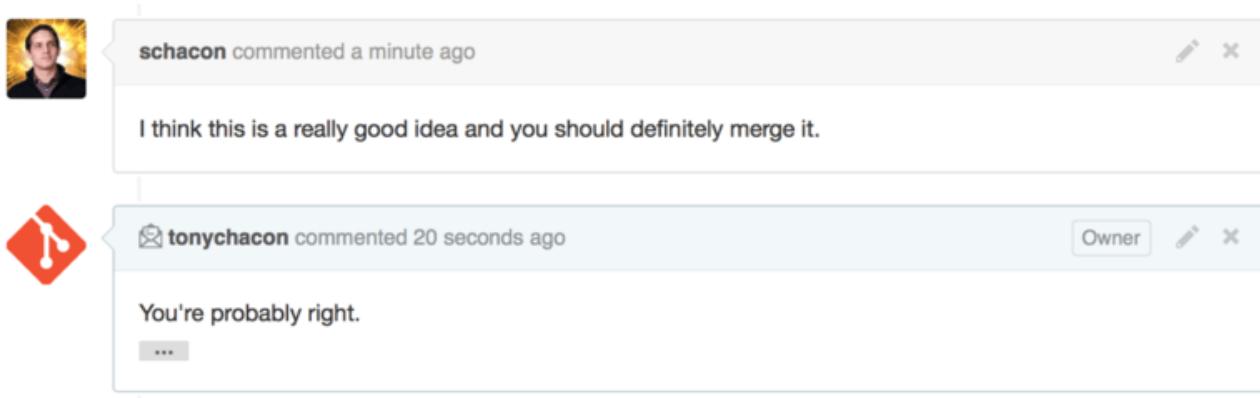


Figure 116. Antwoorden op de emails worden in de converstatie betrokken.

Zodra de code goed genoeg is en je het wilt mergen, kan je ofwel de code lokaal pullen en mergen met de `git pull <url> <branch>` syntax die we eerder gezien hebben, of door de fork als remote toe te voegen en dan te fetchen en mergen.

Als de merge triviaal is, kan je ook gewoon de “Merge” knop op de GitHub site klikken. Dit zal een “non-fast-forward” merge uitvoeren, waardoor een merge commit wordt gemaakt zelfs als er een fast-forward merge mogelijk was. Dit houdt in dat hoe dan ook, elke keer als je de merge knop klikt een merge commit gemaakt wordt. Zoals je kunt zien in [Merge knop en instructies hoe je een Pull Request handmatig merged.](#), geeft GitHub je al deze informatie als je de hint link klikt.

The screenshot shows a GitHub pull request page. At the top, there's a green icon with a person icon and the text "This pull request can be automatically merged." Below it, a message says "You can also merge branches on the command line." To the right is a "Merge pull request" button with a computer monitor icon. A horizontal bar below the header includes tabs for "HTTP", "Git", and "Patch", followed by a URL "https://github.com/schacon/fade.git".

**Merging via command line**

If you do not want to use the merge button or an automatic merge cannot be performed, you can perform a manual merge on the command line.

**Step 1:** From your project repository, check out a new branch and test the changes.

```
git checkout -b schacon-patch-1 master  
git pull https://github.com/schacon/fade.git patch-1
```

**Step 2:** Merge the changes and update on GitHub.

```
git checkout master  
git merge --no-ff schacon-patch-1  
git push origin master
```

Figure 117. Merge knop en instructies hoe je een Pull Request handmatig merged.

Als je besluit dat je het niet wilt mergen, kan je ook gewoon het Pull Request sluiten en de persoon die het geopend heeft krijgt een berichtje.

### Pull Request Refs (Pull Request referenties)

Als je te maken hebt met **veel** Pull Requests en niet elke keer een aantal remotes wilt toevoegen of eenmalige pulls wilt doen, is er een aardige truuk die GitHub je toestaat te doen. Dit is een beetje een truuk voor gevorderden en we zullen de details beter behandelen in [De Refspec](#), maar het kan behoorlijk handig zijn.

GitHub presenteert de Pull Request branches voor een repository als een soort van pseudo-branches op de server. Standaard zal je ze niet krijgen als je kloont, maar ze zijn er wel op een verborgen manier en je kunt ze op een redelijk eenvoudige wijze benaderen.

Om dit te demonstreren, zullen we een low-level (laag niveau) commando (waar vaak aan wordt gerefereerd als een “sanitaire voorziening” (plumbing) commando, waar we meer over zullen lezen in [Binnenwerk en koetswerk \(plumbing and porcelain\)](#)) genaamd `ls-remote` gebruiken. Dit commando wordt over het algemeen niet gebruikt in de dagelijks Git operaties maar het is handig om te laten zien welke referenties er zich op de server bevinden.

Als we dit commando gebruiken met de “blink” repository die we eerder zagen, krijgen we een lijst van alle branches, tags en andere referenties in de repository.

```
$ git ls-remote https://github.com/schacon/blink
10d539600d86723087810ec636870a504f4fee4d    HEAD
10d539600d86723087810ec636870a504f4fee4d    refs/heads/master
6a83107c62950be9453aac297bb0193fd743cd6e    refs/pull/1/head
afe83c2d1a70674c9505cc1d8b7d380d5e076ed3    refs/pull/1/merge
3c8d735ee16296c242be7a9742ebfbc2665adec1    refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d    refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a    refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c    refs/pull/4/merge
```

Natuurlijk, als je in je repository bent en je gebruikt `git ls-remote origin` of welke remote je ook wilt controleren, zal het je iets vergelijkbaars laten zien.

Als de repository op GitHub is en er zijn Pull Requests die open staan, zal je deze referenties krijgen die een prefix `refs/pull/` hebben. Dit zijn eigenlijk gewoon branches, maar omdat deze niet onder `refs/heads/` vermeld staan krijg je ze normaalgesproken niet als je van de server kloont of fetcht — het proces van fetchen negeert ze gewoonlijk.

Er zijn twee referenties per Pull Request - degene die op `/head` eindigt wijst naar precies dezelfde commit als de laatste commit in de Pull Request branch. Dus als iemand in onze repository een Pull Request opent en zijn branch heeft de naam `bug-fix`` en deze wijst naar commit `a5a775`, dan zal in **onze** repository geen branch `bug-fix` aanwezig zijn (omdat deze in hun fork zit), maar we hebben wel `pull/<pr#>/head` die wijst naar `a5a775`. Dit betekent dat we relatief eenvoudig in één keer elke Pull Request branch kunnen pullen zonder daarvoor een heleboel remotes hoeven toe te voegen.

Nu kan je iets doen wat lijkt op het direct fetchen van de referentie.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch                  refs/pull/958/head -> FETCH_HEAD
```

Dit zegt tegen Git, “Maak contact met de `origin` remote, en download de ref genaamd `refs/pull/958/head`.” Git gehoorzaamt trouw, en zal alles downloaden wat je nodig hebt om die ref samen te stellen, en zal een verwijzing naar de commit die je wilt onder `.git/FETCH_HEAD` zetten. Je kunt daarna een `git merge FETCH_HEAD` doen in een branch waar je dit in wilt testen, maar dat merge commit bericht zal er wat vreemd uitzien. Daarnaast, als je **veel** pull requests zal reviewen, wordt dit erg vervelend.

Er is ook een manier om *alle* pull requests te fetchen en ze up-to-date te houden elke keer als je contact maakt met de remote. Open `.git/config` in je favoriete editor, en ga op zoek naar de `origin` remote. Dat zou er zo ongeveer uit moeten zien:

```
[remote "origin"]
url = https://github.com/libgit2/libgit2
fetch = +refs/heads/*:refs/remotes/origin/*
```

De regel die begint met `fetch =` is een “refspec.” Het is een manier om namen op de remote te

mappen met namen in je lokale `.git` directory. Deze specifieke zegt tegen Git: "de spullen op de remote die onder `refs/heads` staan moeten in mijn lokale repository onder `refs/remotes/origin` worden geplaatst." Je kan deze sectie wijzigen door een andere refspec toe te voegen:

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Die laatste regel vertelt Git, "Alle refs die er als `refs/pull/123/head` uitzien moeten lokaal worden opgeslagen als `refs/remotes/origin/pr/123`." Als je dit bestand nu opslaat en een `git fetch` uitvoert:

```
$ git fetch
# ...
* [new ref]          refs/pull/1/head -> origin/pr/1
* [new ref]          refs/pull/2/head -> origin/pr/2
* [new ref]          refs/pull/4/head -> origin/pr/4
# ...
```

Nu worden alle remote pull request lokaal vertegenwoordigd met refs die zich vrijwel hetzelfde gedragen als tracking branches: ze zijn alleen-lezen, en ze worden geüpdatet als je een fetch doet. Dat maakt het enorm makkelijk om de code van een pull request lokaal uit te checken:

```
$ git checkout pr/2
Checking out files: 100% (3769/3769), done.
Branch pr/2 set up to track remote branch pr/2 from origin.
Switched to a new branch 'pr/2'
```

De oplettende lezers tussen jullie zullen de `head` aan het eind van het remote deel van de refspec hebben opgemerkt. Er is ook een `refs/pull/#/merge` ref aan de GitHub kant, welke de commit vertegenwoordigt die het resultaat zou zijn als je op de "merge" knop op de site zou klikken. Dit stelt je in staat de merge te testen zelfs voordat je de knop klikt.

## Pull Requests op Pull Requests

Je kunt niet alleen Pull Requests openen die de main of `master`-branch betreffen, je kunt zelfs een Pull Request openen die betrekking heeft op elke willekeurige branch in het netwerk. Je kunt zelfs een Pull Request openen op een andere Pull Request.

Als je een Pull Request ziet die de juiste kant op gaat en je hebt een idee voor een verandering die daarvan afhankelijk is of je weet niet zeker of het een goed idee is, of je hebt domweg geen push-toegang op de doelbranch, kan je een Pull Request direct op de Pull Request openen.

Als je een Pull Request opent, is er een invoerveld bovenaan op de pagina die aangeeft naar welke branch je wilt laten pullen en welke je het van wilt laten pullen. Als je de "Edit" knop rechts van dat invoerveld klikt kan je niet alleen de branches wijzigen, maar ook welke fork.

The screenshot shows a GitHub pull request comparison between 'schacon:master' and 'tonychacon:patch-2'. At the top, there are buttons for 'Create pull request' and 'Edit'. Below that, a summary shows '2 commits', '1 file changed', '0 commit comments', and '2 contributors'. The commit list shows two commits from 'schacon' and 'tonychacon' on Oct 02, 2014. A dropdown menu titled 'Choose a base branch' is open, showing options like 'master' and 'patch-1'. The 'master' option is selected.

Figure 118. Handmatig de Pull Request doelfork en -branch wijzigen.

Hier kan je redelijk eenvoudig aangeven om jouw nieuwe branch in een andere Pull Request te mergen of in een andere fork van het project.

## Vermeldingen en meldingen

GitHub heeft ook een redelijk aardig ingebouwde meldingen systeem die handig kan worden als je vragen hebt of terugkoppeling wilt van specifiek individuen of teams.

In elk commentaar kan je een @ teken typen en een automatische aanvulling met de namen en gebruikersnamen van mensen die medewerkers of bijdragers in het project zal beginnen.

The screenshot shows a GitHub comment input field. The 'Write' tab is active, and the 'Preview' tab is visible. The input field starts with '@'. A dropdown menu shows suggestions for users: 'ben Ben Straub', 'peff Jeff King', 'jlehmann Jens Lehmann', and 'LouiseCorrigan Louise Corrigan'. The 'ben Ben Straub' entry is highlighted. Below the dropdown, a note says 'selecting them, or pasting from the clipboard.' There are 'Close and comment' and 'Comment' buttons at the bottom.

Figure 119. Begin @ te typen om iemand te vermelden.

Je kunt ook een gebruiker vermelden die niet in die dropdown staat, maar vaak kan de automatische aanvuller het sneller maken.

Als je eenmaal een commentaar hebt gepost met een gebruikersvermelding, zal die gebruiker een berichtje krijgen. Dat houdt in dat dit een erg effectieve manier is om iemand in een discussie te betrekken in plaats van ze actief hierop te laten controleren. Het gebeurt heel vaak op GitHub dat mensen via Pull Requests anderen in hun team of bedrijf erbij betrekken om een Issue of Pull Request te laten reviewen.

Als iemand vermeld wordt in een Pull Request of Issue, worden ze erop “geabonneerd” en blijven meldingen krijgen voor elke activiteit die erop plaatsvindt. Je wordt ook geabonneerd op iets wat je geopend hebt, als je een repository volgt of als je ergens commentaar op geeft. Als je niet langer deze meldingen wilt krijgen, is er een “Unsubscribe” (Afmeld) knop op de pagina die je kunt klikken om de meldingen te stoppen.

## Notifications

 **Unsubscribe**

You're receiving notifications because you commented.

Figure 120. Afmelden van een Issue of Pull Request.

### De meldingen pagina

Als we hier over “meldingen” spreken in de context van GitHub, bedoelen we de specifieke manier waarop GitHub probeert om met je in contact te blijven als er gebeurtenissen zijn en er zijn een aantal manieren waarop je dit kunt configureren. Als je naar de “Notification center” tab van de instellingen pagina gaat, kan je een aantal opties zien die je hebt.

Figure 121. Notification center opties.

De twee keuzes zijn om de berichten via “Email” en via “Web” te ontvangen en je kunt een van twee, geen of beide selecteren als je actief deelneemt aan zaken en voor activiteiten op repositories die je volgt.

## Web Meldingen

Web meldingen bestaan alleen binnen GitHub en je kunt ze alleen op GitHub controleren. Als je deze optie geselecteerd hebt in je voorkeuren en een bericht wordt voor je gemaakt, zie je een kleine blauwe stip boven je meldingen ikoon boven in je scherm zoals getoond in [Notification center..](#)

Figure 122. Notification center.

Als je hierop klikt, zal je een lijst met alle items zien waar je voor wordt bericht, gegroepeerd per project. Je kunt de meldingen van een specifiek project filteren door op de naam te klikken in de

kolom links. Je kunt ook de meldingen bevestigen door het vink-icoon te klikken naast elke melding, of *alle* meldingen in een project bevestigen door de vink boven in de groep te klikken. Er is ook een demp (mute) knop naast elke vinkteken die je kunt klikken om geen enkel bericht meer voor dat bericht te ontvangen.

Al deze instrumenten zijn erg handig voor het afhandelen van grote aantallen meldingen. Veel gevorderde GitHub gebruikers zullen eenvoudigweg alle email berichten uitschakelen en al hun meldingen via dit scherm afhandelen.

### E-mail meldingen

E-mail meldingen zijn de andere manier waarop je berichten kunt afhandelen via GitHub. Als je deze ingeschakeld hebt, krijg je per melding een e-mail. We hebben hiervan voorbeelden gezien in [Email berichten](#) en [Email bericht van een nieuwe Request](#). De e-mails zullen ook juist geketend (threaded) worden, wat handig is als je een zgn. threading e-mail client gebruikt.

Er zit ook een behoorlijke hoeveelheid metadata in de headers van de e-mails die GitHub je stuurt, deze kunnen heel handig zijn voor het inrichten van zelfgemaakte filters en regels.

Bijvoorbeeld, als we een kijkje nemen naar de daadwerkelijke e-mail headers die aan Tony zijn gestuurd in de e-mail in [Email bericht van een nieuwe Request](#), zullen we hetvolgende zien in de gestuurde gegevens:

```
To: tonychacon/fade <fade@noreply.github.com>
Message-ID: <tonychacon/fade/pull/1@github.com>
Subject: [fade] Wait longer to see the dimming effect better (#1)
X-GitHub-Recipient: tonychacon
List-ID: tonychacon/fade <fade.tonychacon.github.com>
List-Archive: https://github.com/tonychacon/fade
List-Post: <mailto:reply+i-4XXX@reply.github.com>
List-Unsubscribe: <mailto:unsub+i-XXX@reply.github.com>,...
X-GitHub-Recipient-Address: tchacon@example.com
```

Hier zijn een aantal interessante dingen. Als je e-mails wilt vlaggen of routeren naar dit specifieke project of zelfs Pull Request, geeft de informatie in **Message-ID** je alle gegevens in **<user>/<project>/<type>/<id>** formaat. Als dit bijvoorbeeld een issue zou zijn zou het **<type>** veld “issues” zijn niet “pull”.

De **List-Post** en **List-Unsubscribe** velden houden in dat als je een mail client hebt die deze velden begrijpt, je eenvoudig een bericht kan sturen naar de thread of ervan kunt “Unsubscribe”. Dat zou effectief hetzelfde zijn als de “mute” knop klikken op de web versie van de melding of “Unsubscribe” op het Issue of Pull Request pagina zelf.

Het is ook de moeite om te vermelden dat als je zowel e-mail als web meldingen aan hebt staan en je de e-mail versie van de melding gelezen hebt, de webversie ook als gelezen wordt gemarkeerd als je het laden van plaatjes toestaat in je mail client.

## Speciale Bestanden

Er zijn een aantal speciale bestanden die GitHub zal opmerken als ze in je repository staan.

### README

Het eerste is de **README** file, die kan ongeveer elk formaat hebben die GitHub herkent als vrije tekst. Bijvoorbeeld het zou **README**, **README.md**, **README.asciidoc**, etc. kunnen zijn. Als GitHub een README bestand in je broncode vindt, zal het dit op de ingangspagina van het project tonen.

Veel teams gebruiken dit bestand om alle relevante project informatie te bevatten ter informatie voor iemand die nieuw is binnen de repository of het project. Meestal bevat het zaken als:

- Waartoe dient het project
- Hoe deze te configureren en te installeren
- Een voorbeeld hoe het te gebruiken of het aan te lopen te krijgen
- De licentie waaronder het project wordt aangeboden
- Hoe er aan bij te dragen

Omdat GitHub dit bestand toont, kan je plaatjes of links er in opnemen voor het verhogen van het begrip.

### CONTRIBUTING

Het andere speciale bestand dat GitHub herkent is het **CONTRIBUTING** bestand. Als je een bestand genaamd **CONTRIBUTING** met een willekeurige extensie, zal GitHub **Een Pull Request openen als er een CONTRIBUTING file bestaan.** tonen als iemand een Pull Request gaat openen.

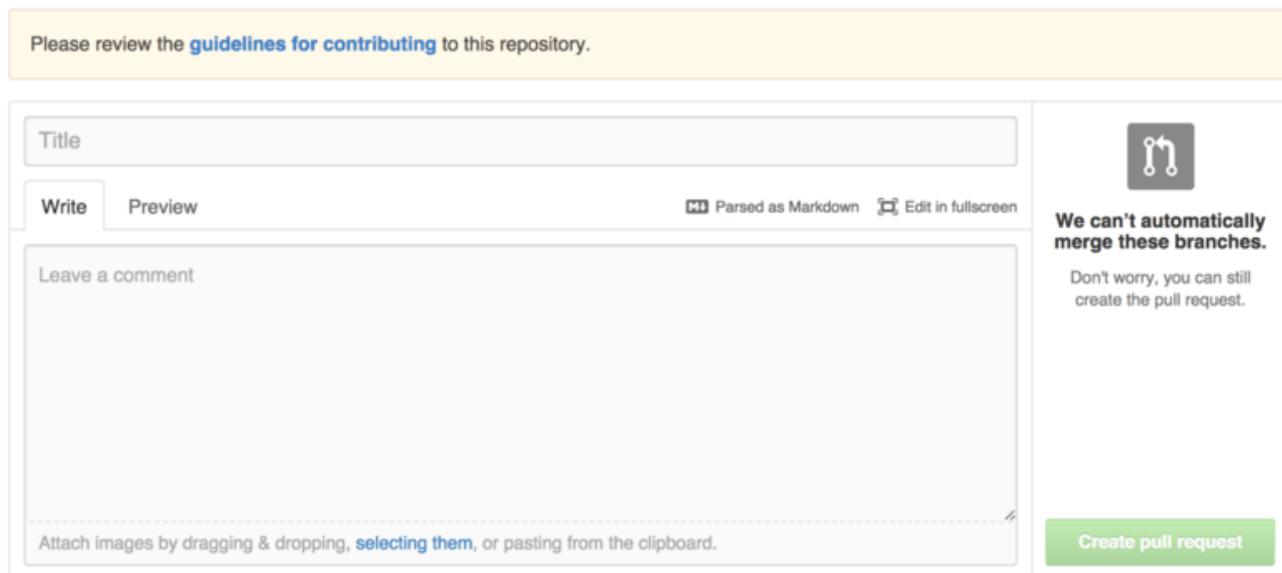


Figure 123. Een Pull Request openen als er een CONTRIBUTING file bestaan.

Het idee hierachter is dat je specifieke zaken kunt aangeven die je wel of niet wilt in een Pull Request die aan je project wordt gestuurd. Op deze manier zouden mensen de richtlijnen misschien echt lezen voordat ze een Pull Request openen.

# Project Beheer

Over het algemeen zijn er niet veel beheersmatige zaken die je kunt doen met een enkel project, maar er zijn een aantal dingen die van belang kunnen zijn.

## De standaard branch wijzigen

Als je een branch anders dan “master” gebruikt als je standaard branch waarvan je wilt dat mensen er Pull Requests op openen of die ze standaard zien, kan je dat in de instellingen pagina van je repository wijzigen onder de “Options” tab.

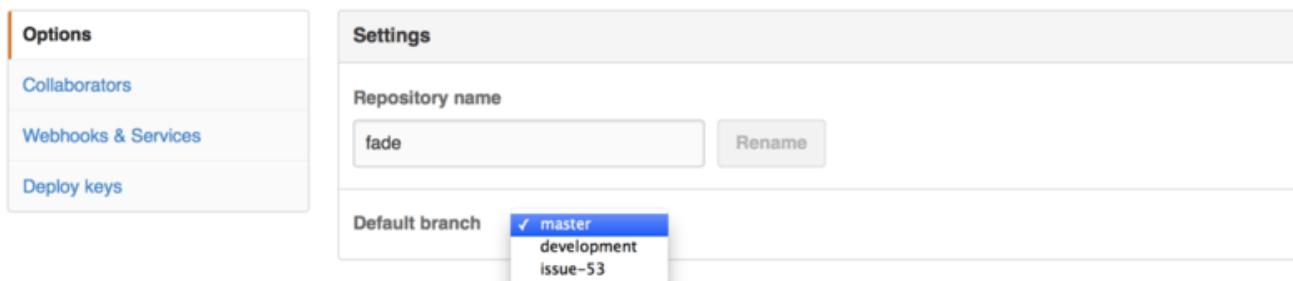


Figure 124. De default branch van een project wijzigen.

Eenvoudigweg de standaard branch in de dropdown wijzigen en dat wordt vanaf dat moment de standaard-branch voor alle belangrijke handelingen, inclusief welke branch er standaard uitgechecked wordt als iemand de repository kloont.

## Een project overdragen

Als je een project wilt overdragen aan een andere gebruiker of organisatie in GitHub, is er een “Transfer ownership” optie onderaan dezelfde “Options” tab van de instellingen pagina van je repository die je dat kan laten doen.

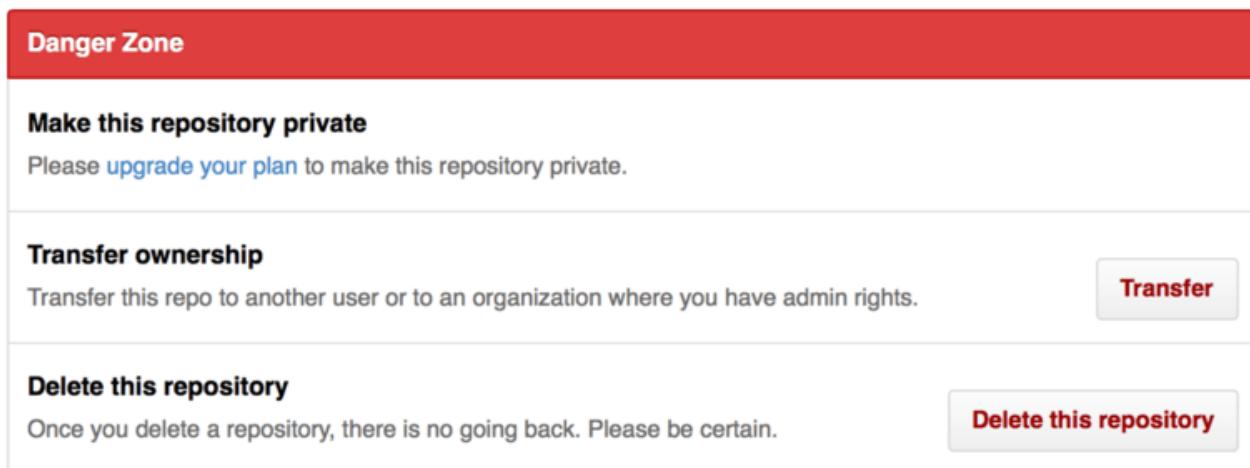


Figure 125. Draag een project over aan een andere GitHub gebruiker of organisatie.

Dit is handig als je een project verlaat en iemand anders wilt het overnemen, of je project wordt groter en je wilt het in een organisatie onderbrengen.

Niet alleen verplaatst dit de repository met al zijn volgers en sterren naar een andere plaats, het

richt ook een redirect (doorverwijzing) van jouw URL naar de nieuwe plaats. Het zal ook de clones en fetches van Git doorverwijzen, niet alleen de web-aanvragen.

## Een organisatie beheren

Aanvullend aan accounts voor personen heeft GitHub ook zogenoemde Organisaties (Organizations). Net als persoonlijke accounts, hebben Organizational accounts een namespace waar al hun projecten bestaan, maar veel andere dingen zijn anders. Deze accounts vertegenwoordigen een groep van mensen met gedeelde eigenaarschap van projecten, en er zijn veel instrumenten om subgroepen met die mensen te beheren. Normaalgesproken worden deze accounts gebruikt voor Open Source groepen (zoals “perl” of “rails”) of bedrijven (zoals “google” of “twitter”).

### Grondbeginsele van Organizations

Een organisatie is vrij eenvoudig te maken; gewoon op het “+” icoon rechtsboven op elke GitHub pagina, en “New organization” in het menu kiezen.

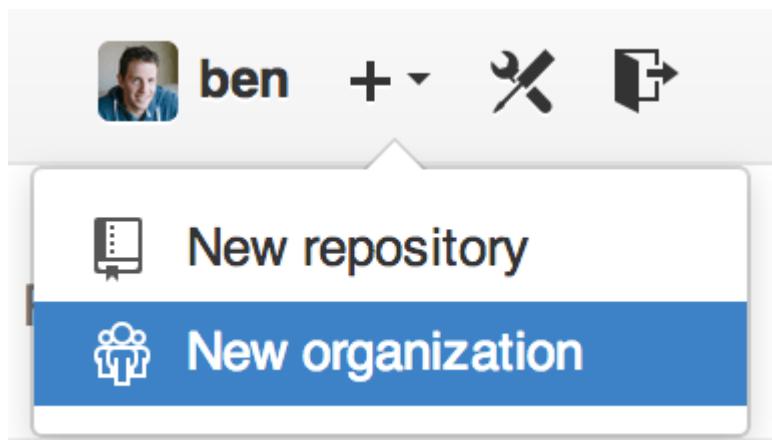


Figure 126. De “New organization” menu keuze.

Eerst moet je jouw organisatie een naam geven en een e-mail adres geven die als primaire contact adres dient voor de groep. Daarna kan je, als je wilt, andere gebruikers uitnodigen om mede-eigenaren te zijn van het account.

Volg deze stappen en je ben al snel de eigenaar van een gloednieuwe organisatie. Net als persoonlijke accounts, zijn organisaties gratis als alles wat je van plan bent er op te slaan open source zal zijn.

Als een eigenaar in een organisatie, als je een repository forkt, hebt je de keuze om deze naar de namespace van je organisatie te forken. Als je nieuwe repositories maakt kan je ze onder je eigen persoonlijke account aanmaken of onder een van de organisaties waar je een eigenaar van bent. Je “volgt” ook automatisch elke nieuwe repository die onder deze organisaties wordt aangemaakt.

Net als in [Jouw avatar](#), kan je een avatar uploaden voor je organisatie om het een persoonlijk tintje te geven. En ook net zoals bij persoonlijke accounts heb je een landingspagina voor de organisatie die een lijst bevat van al je repositories en die door andere mensen kunnen worden bekeken.

Laten we nu eens een aantal andere zaken behandelen die wat anders zijn bij een account van een organisatie.

## Teams

Organisaties worden geassocieerd met individuele mensen door middel van teams, die eenvoudigweg een verzameling van individuele persoonlijke accounts en repositories zijn binnen die organisatie en welk soort van toegang deze mensen in deze repositories hebben.

Als voorbeeld, stel dat je bedrijf die repositories heeft: **frontend**, **backend**, en **deployscripts**. Je zou je HTML/CSS/Javascript ontwikkelaars toegang willen geven tot **frontend** en misschien **backend**, en de mensen van Operations toegang tot **backend** en **deployscripts**. Met teams wordt dit makkelijk, zonder de medewerkers voor elke individuele repository te beheren.

De Organization pagina laat je een eenvoudig dashboard van al de repositories, gebruikers en teams zien die onder deze organisatie vallen.

The screenshot shows the GitHub Organization page for the user 'chaconcorp'. On the left, there's a list of repositories:

- deployscripts**: Scripts for deployment, updated 16 hours ago.
- backend**: Backend Code, updated 16 hours ago.
- frontend**: Frontend Code, updated 16 hours ago.

On the right, there are two sections: 'People' and 'Teams'.

**People** section:

- dragonchacon (Dragon Chacon)
- schacon (Scott Chacon)
- tonychacon (Tony Chacon)

**Teams** section:

- Owners**: 1 member - 3 repositories
- Frontend Developers**: 2 members - 2 repositories
- Ops**: 3 members - 1 repository

Buttons for 'Invite someone' and 'Create new team' are also visible.

Figure 127. De Organization pagina.

Om je teams te beheren, kan je op de Teams kolom aan de rechterkant van de pagina in [De Organization pagina](#). klikken. Dit leidt je naar een pagina die je kunt gebruiken om leden aan het team toe te voegen, repositories aan het team toe te voegen of de instellingen en toegangs niveaus voor het team te beheren. Elk team kan alleen lees-, lees-/schrijf- of beheertoegang tot de repositories hebben. Je kunt dat niveau wijzigen door de “Settings” knop in [De Team pagina](#). te klikken.

The screenshot shows the GitHub Team page for 'Frontend Developers'. On the left, there's a sidebar with team statistics: 2 MEMBERS and 2 REPOSITORIES. Below that are 'Leave' and 'Settings' buttons. The main area has tabs for 'Members' (selected) and 'Repositories'. Under 'Members', it lists two users: 'tonychacon' (Tony Chacon) and 'schacon' (Scott Chacon). Each user entry includes a small profile picture, the username, the full name, and a 'Remove' button. At the top right of the main area is a button to 'Invite or add users to team'.

Figure 128. De Team pagina.

Als je iemand uitnodigt bij een team, krijgen ze een e-mail waarin staat dat je ze hebt uitgenodigd.

Daarbij, werken team **@vermeldingen** (zoals **@acmecorp/frontend**) vrijwel gelijk als bij individuele gebruikers behalve dat **alle** leden van het team dan geabonneerd zijn op die thread. Dit is handig als je de aandacht wilt van iemand in een team, maar je weet niet precies wie te vragen.

Een gebruiker kan tot een willekeurig aantal teams behoren, dus beperk jezelf niet tot alleen tot toegangs-beheer teams. Thematische teams zoals **ux**, **css**, of **refactoring** zijn handig voor bepaalde type vragen, en andere zoals **wetgeving** en **kleurenblindheid** voor compleet andere.

## Audit Log

Organisaties geven eigenaren ook toegang tot alle informatie over wat binnen die organisatie is gebeurd. Je kunt naar de *Audit Log* tab gaan en zien welke gebeurtenissen er hebben plaatsgevonden op een organisationeel niveau, wie ze in gang heeft gezet en waar in de wereld dit gebeurde.

**Recent events**

| Filters ▾     |                         |        |                | Search... |
|---------------|-------------------------|--------|----------------|-----------|
|               | Yesterday's activity    | member | 32 minutes ago |           |
|               | Organization membership | member | 33 minutes ago |           |
|               | Team management         | member | 33 minutes ago |           |
|               | Repository management   |        |                |           |
|               | Billing updates         | member | 16 hours ago   |           |
|               | Hook activity           |        |                |           |
| <b>France</b> |                         |        | 16 hours ago   |           |
|               | tonychacon              |        | 16 hours ago   |           |
|               | tonychacon              |        | 16 hours ago   |           |
|               | tonychacon              |        | 16 hours ago   |           |
|               | tonychacon              |        | 16 hours ago   |           |
|               | tonychacon              |        | 16 hours ago   |           |

Figure 129. De audit log.

Je kunt ook specifieke gebeurtenissen filteren, specifieke plaatsen of specifieke mensen.

## GitHub Scripten

We hebben nu alle hoofdfuncties en workflows van GitHub hebben behandeld, maar elke grote groep of project zal aanpassingen hebben die ze willen maken of services van buitenaf die ze willen integreren.

Gelukkig voor ons is GitHub echt op vele manieren redelijk te hacken. In deze paragraaf zullen we behandelen hoe het zgn. GitHub haken (hooks) en API systeem te gebruiken om GitHub zich te laten gedragen zoals we willen.

## Services en Hooks

Het Hooks en Services deel van het GitHub repository beheer is de eenvoudigste manier om GitHub te laten samenwerken met externe systemen.

### Services

Eerst zullen we naar Services kijken. Zowel Hooks als Services integratie kunnen in het Settings gedeelte van je repository gevonden worden, waar we eerder naar gekeken hebben bij het toevoegen van medewerkers en het wijzigen van de standaard-branch van je project. Onder de “Webhooks and Services” tab zul je iets als [Services and Hooks configuratie deel](#). zien.

The screenshot shows the GitHub repository settings page. On the left, a sidebar menu includes 'Options', 'Collaborators', **'Webhooks & Services'** (which is currently selected), and 'Deploy keys'. The main content area has two tabs: 'Webhooks' and 'Services'. The 'Webhooks' tab contains a brief description of what webhooks are and how they work, with a 'Add webhook' button. The 'Services' tab contains a brief description of what services are and how they perform actions, with a 'Available Services' dropdown menu open. The dropdown menu lists 'email' and has a 'Email' button at the bottom.

Figure 130. Services and Hooks configuratie deel.

Er zijn tientallen services waar je uit kunt kiezen, de meeste zijn integraties naar andere commerciële en open source systemen. De meeste daarvan zijn Continuous Integration services, bug en issue trackers, chat room systemen en documentatie systemen. We zullen je door het opzetten van een eenvoudige leiden: de Email-hook. Als je “email” kiest uit de “Add Service” dropdown, krijg je een configuratie scherm zoals [E-mail service configuratie..](#)

Options

Collaborators

**Webhooks & Services**

Deploy keys

Services / Add Email

## Install Notes

- address whitespace separated email addresses (at most two)
- secret fills out the Approved header to automatically approve the message in a read-only or moderated mailing list.
- `send_from_author` uses the commit author email address in the From address of the email.

**Address**

**Secret**

Send from author

Active  
We will run this service when an event is triggered.

**Add service**

Figure 131. E-mail service configuratie.

In dit geval, als we de “Add service” knop klikken, zal het e-mail adres die we intypen elke keer een bericht ontvangen als iemand naar de repository pusht. Services kunnen luisteren naar verschillende type gebeurtenissen, maar de meeste luisteren alleen naar push-events en doen dan iets met die gegevens.

Als er een systeem is dat je gebruikt en die je wilt integreren met GitHub, zou je hier moeten kijken om te zien of er een bestaande service integratie beschikbaar is. Je zou, als je bijvoorbeeld Jenkins gebruikt om tests te draaien op je codebase, de ingebouwde Jenkins service integratie kunnen aanzetten om een testrun af te trappen elke keer als iemand naar jouw repository pusht.

## Hooks

Als je iets meer specifieker nodig hebt, of je wilt een service of site integreren die niet in de lijst staat, kan je in plaats daarvan het meer generieke hooks systeem gebruiken. GitHub repository hooks zijn redelijk eenvoudig. Je geeft een URL op en GitHub zal een HTTP payload posten op die URL bij elke gebeurtenis dat je maar wilt.

Hoe dit globaal werkt is dat je een kleine web service kunt opzetten die naar een GitHub hook payload luistert en dan iets met de gegevens doet als het is ontvangen.

Om een hook aan te zetten, klik je de “Add webhook” knop in [Services and Hooks configuratie](#) deel.. Dit leidt je naar een pagina die eruit ziet als [Web hook configuratie..](#)

The screenshot shows the 'Webhooks & Services' section of a GitHub repository settings page. On the left, a sidebar lists 'Options', 'Collaborators', 'Webhooks & Services' (which is selected and highlighted in orange), and 'Deploy keys'. The main content area is titled 'Webhooks / Add webhook'. It contains instructions about sending POST requests to the specified URL with event details. A 'Payload URL' input field contains 'https://example.com/postreceive'. A 'Content type' dropdown is set to 'application/json'. A 'Secret' input field is empty. Below these, a section asks 'Which events would you like to trigger this webhook?' with three radio button options: 'Just the push event.' (selected), 'Send me everything.', and 'Let me select individual events.'. A checked checkbox labeled 'Active' indicates that event details will be delivered when the hook is triggered. At the bottom is a green 'Add webhook' button.

Figure 132. Web hook configuratie.

De configuratie van een web hook is redelijk eenvoudig. In de meeste gevallen voer je een URL in en een geheime sleutel en klikt “Add webhook”. Er zijn een paar opties voor welke gebeurtenissen je wilt waarvoor GitHub je een payload stuurt—standaard is om alleen een payload te ontvangen voor de **push** gebeurtenis, als iemand nieuwe code naar een van de branches uit je repository pusht.

Laten we een kort voorbeeld van een web service bekijken die je zou kunnen opzetten om een web hook te verwerken. We zullen het Ruby web framework Sinatra gebruiken omdat dit redelijk bondig is en je in staat zou moeten zijn om snel te zien wat we aan het doen zijn.

Laten we stellen dat we een e-mail willen ontvangen als een bepaald persoon naar een specifieke branch van ons project pusht waarin een zeker bestand wordt gewijzigd. We kunnen dat relatief eenvoudig doen met code zoals deze:

```

require 'sinatra'
require 'json'
require 'mail'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON

  # gather the data we're looking for
  pusher = push["pusher"]["name"]
  branch = push["ref"]

  # get a list of all the files touched
  files = push["commits"].map do |commit|
    commit['added'] + commit['modified'] + commit['removed']
  end
  files = files.flatten.uniq

  # check for our criteria
  if pusher == 'schacon' &&
    branch == 'ref/heads/special-branch' &&
    files.include?('special-file.txt')

    Mail.deliver do
      from      'tchacon@example.com'
      to        'tchacon@example.com'
      subject   'Scott Changed the File'
      body      "ALARM"
    end
  end
end

```

Hier nemen we de JSON payload die GitHub ons levert en kijken na wie gepusht heeft, naar welke branch hij gepusht heeft en welke bestanden geraakt zijn in alle commits die zijn gepusht. Dan houden we die gegevens tegen onze criteria en sturen een e-mail als ze passen.

Om zo iets te kunnen ontwikkelen en testen, heb je een nette ontwikkelaarsscherm in hetzelfde scherm waar je ook de hook ingesteld hebt. Je kunt de laatste paar leveranties die GitHub heeft proberen te maken voor die webhook zien. Voor elke hook kan je uitvinden wanneer het was afgeleverd, of dit succesvol was en de body en headers voor zowel de vraag en het antwoord. Dit maakt het ongelofelijk eenvoudig om je hooks te testen en te debuggen.

**Recent Deliveries**

|                                      |                                      |                     |     |
|--------------------------------------|--------------------------------------|---------------------|-----|
| <span style="color: red;">!</span>   | 4aeae280-4e38-11e4-9bac-c130e992644b | 2014-10-07 17:40:41 | ... |
| <span style="color: green;">✓</span> | aff20880-4e37-11e4-9089-35319435e08b | 2014-10-07 17:36:21 | ... |
| <span style="color: green;">✓</span> | 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": "99d4fe5bfffaf827f8a9e7cde00cbb0ab06a35e48",
  "after": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
  "created": false,
  "deleted": false,
  "forced": false,
  "base_ref": null,
  "compare": "https://github.com/tonychacon/fade/compare/99d4fe5bfffaf...9370a6c33493",
  "commits": [
    {
      "id": "9370a6c3349331bac7e4c3c78c10bc8460c1e3e8",
      "distinct": true,
      "message": "remove whitespace",
      "timestamp": "2014-10-07T17:35:22+02:00",
      "url": "https://github.com/tonychacon/fade/commit/9370a6c3349331bac7e4c3c78c10bc8460c1e3e8"
    }
  ]
}
```

Figure 133. Web hook debug informatie.

De andere geweldige mogelijkheid van dit is dat je elk van de payloads opnieuw kan laten afleveren om je service makkelijk te testen.

Voor meer informatie over hoe webhooks te schrijven en alle verschillende type gebeurtenissen waar je naar kunt verwijzen we je naar de GitHub Developer documentatie op: <https://developer.github.com/webhooks/>

## De GitHub API

Services en hooks bieden je een manier om push berichten te ontvangen van gebeurtenissen die plaatsvinden op je repositories, maar wat nu als je meer informatie hierover nodig hebt? Wat als je iets wilt automatiseren zoals medewerkers toevoegen of issues markeren?

Dit is waar de GitHub API handig bij gebruikt kan worden. GitHub heeft ongelofelijk veel API endpoints om bijna alles wat je op de website kan doen op een geautomatiseerde manier kan doen. In deze paragraaf zullen we leren hoe te authenticeren en te verbinden met de API, hoe te reageren op een issue en hoe de status van een Pull Request te wijzigen middels de API.

## Eenvoudig gebruik

Het meest basale wat je kunt doen is een simpele GET request op een endpoint die geen authenticatie behoeft. Dit zou een gebruiker of read-only informatie op een open source project kunnen zijn. Bijvoorbeeld, als we meer willen weten van een gebruiker genaamd “schacon”, kunnen we zo iets uitvoeren:

```
$ 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"
}
```

Er zijn honderden van soortgelijke endpoints als deze om informatie over organisaties, projecten, issues, commits te verkrijgen — zo ongeveer alles wat je publiekelijk kan zien op GitHub. Je kunt de API zelfs gebruiken om willekeurige MarkDown te tonen of een [.gitignore](#) template vinden.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see
http://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
}
```

## Reageren op een issue

Echter, als je een actie wilt uitvoeren op de website zoals reageren op een Issue of Pull Request of als je gesloten informatie wilt zien of ermee interacteren, zal je je moeten authenticeren.

Er zijn verschillende manieren om je te authenticeren. Je kunt eenvoudige authenticatie gebruiken met gewoon je gebruikersnaam en wachtwoord, maar over het algemeen is het een beter idee om een persoonlijke toegangs bewijs (access token) te gebruiken. Deze kan je genereren vanaf de “Applications” tab van je instellingen pagina.

The screenshot shows the GitHub user settings page for 'tonychacon'. The left sidebar has sections for Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications (which is selected), Repositories, and Organizations. The main content area is titled 'Developer applications' with a 'Register new application' button. Below it, a message says 'Do you want to develop an application that uses the GitHub API? [Register an application](#) to generate OAuth tokens.' The 'Personal access tokens' section is expanded, showing a 'Generate new token' button and a note: 'Need an API token for scripts or testing? [Generate a personal access token](#) for quick access to the GitHub API.' A detailed note explains: '② Personal access tokens function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).' The 'Authorized applications' and 'GitHub applications' sections are also shown, both currently empty.

Figure 134. Genereer je access token vanaf de “Applications” tab op je instellingen pagina.

Het zal je vragen welke contexten je wilt voor dit token en een omschrijving. Zorg ervoor dat je een goede omschrijving gebruikt zo dat je met vertrouwen het token kan weggooien als je script of applicatie niet langer meer in gebruik is.

GitHub laat je het token maar één keer zien, dus zorg ervoor dat je het kopiëert. Je kunt deze nu gebruiken om te authenticeren in je script in plaats van een gebruikersnaam en wachtwoord. Dit is prettig omdat je de context waarin je iets wilt doen kan beperken en het token is weer intrekbaar.

Het heeft ook het bijkomende voordeel dat het je aanvraag limiet verhoogt. Zonder authenticatie ben je gelimiteerd tot 60 aanvragen per uur. Als je authenticert kan je tot 5.000 aanvragen per uur doen.

Dus laten we het gebruiken om een reactie te geven op een van onze issues. Stel dat we een reactie willen geven op een specifieke issue, Issue #6. Om dit te doen moeten we een HTTP POST request op `repos/<user>/<repo>/issues/<num>/comments` uitvoeren met het token wat we zojuist gegenereerd hebben als een Authorization header.

```

$ curl -H "Content-Type: application/json" \
-H "Authorization: token TOKEN" \
--data '{"body":"A new comment, :+1:"}' \
https://api.github.com/repos/schacon/blink/issues/6/comments
{
  "id": 58322100,
  "html_url": "https://github.com/schacon/blink/issues/6#issuecomment-58322100",
  ...
  "user": {
    "login": "tonychacon",
    "id": 7874698,
    "avatar_url": "https://avatars.githubusercontent.com/u/7874698?v=2",
    "type": "User",
  },
  "created_at": "2014-10-08T07:48:19Z",
  "updated_at": "2014-10-08T07:48:19Z",
  "body": "A new comment, :+1:"
}

```

Als je nu naar dat issue gaat, kan je de reactie zien dat we zojuist succesvol gepost hebben in [Een commentaar gepost via de GitHub API..](#)

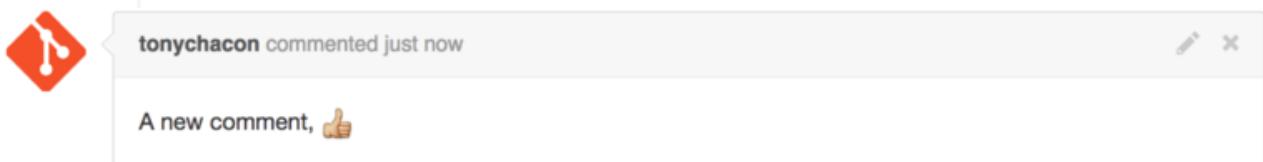


Figure 135. Een commentaar gepost via de GitHub API.

Je kunt de API gebruiken om zo ongeveer alles te doen wat je op website kunt doen—mijlpalen maken en zetten, mensen aan issues en pull requests toewijzen, labels maken en wijzigen, commit gegevens benaderen, nieuwe commits en branches maken, Pull Requests openen, sluiten of mergen, teams maken en wijzigen, reageren op regels code in een Pull Request, op de site zoeken enzovoorts, etcetera.

## De status van een Pull Request wijzigen

We zullen nog een laatste voorbeeld bekijken omdat het echt handig is als je werkt met Pull Requests. Elke commit kan een of meerdere statussen hebben en er is een API om een status toe te voegen en deze uit te vragen.

De meeste Continuous Integration en test services gebruiken deze API om op pushes te reageren door de code die is gepusht te testen, en dan terug te melden of die commit alle tests heeft gepasseerd. Je kunt dit ook gebruiken om te kijken of het commit bericht juist is geformatteerd, of de indiener al je bijdrage richtlijnen heeft gevuld, of de commit juist getekend was—verzin het maar.

Laten we stellen dat je een webhook op je repository ingericht hebt die een kleine webservice aanroeft die controleert of er een **Signed-off-by** letterreeks in het commit bericht voorkomt.

```

require 'httparty'
require 'sinatra'
require 'json'

post '/payload' do
  push = JSON.parse(request.body.read) # parse the JSON
  repo_name = push['repository']['full_name']

  # look through each commit message
  push["commits"].each do |commit|

    # look for a Signed-off-by string
    if /Signed-off-by/.match commit['message']
      state = 'success'
      description = 'Successfully signed off!'
    else
      state = 'failure'
      description = 'No signoff found.'
    end

    # post status to GitHub
    sha = commit["id"]
    status_url = "https://api.github.com/repos/#{repo_name}/statuses/#{sha}"

    status = {
      "state"      => state,
      "description" => description,
      "target_url"  => "http://example.com/how-to-signoff",
      "context"     => "validate/signoff"
    }
    HTTParty.post(status_url,
      :body => status.to_json,
      :headers => {
        'Content-Type'  => 'application/json',
        'User-Agent'    => 'tonychacon/signoff',
        'Authorization' => "token #{ENV['TOKEN']}" })
  end
end

```

Hopelijk is dit redelijk eenvoudig te volgen. In deze webhook-verwerker kijken we door elke commit die zojuist is gepusht, we zoeken naar de reeks *Signed-off-by* in het commit bericht en tenslotten POSTen we via HTTP naar de [/repos/<user>/<repo>/statuses/<commit\\_sha>](#) API endpoint met de status.

In dit geval kan je een status (*success, failure, error*) sturen, een omschrijving wat er gebeurd is, een doel URL waar de gebruiker heen kan gaan voor meer informatie en een “context” in geval er meerdere statussen voor een enkele commit zijn. Bijvoorbeeld, een test-service kan een status aangeven en een validatie service zoals deze kan ook een status aangeven — het “context” veld

maakt hierin het onderscheid.

Als iemand een nieuwe Pull Request op GitHub opent en deze hook is opgezet, kan je iets zoals [Commit status via de API](#) zien.

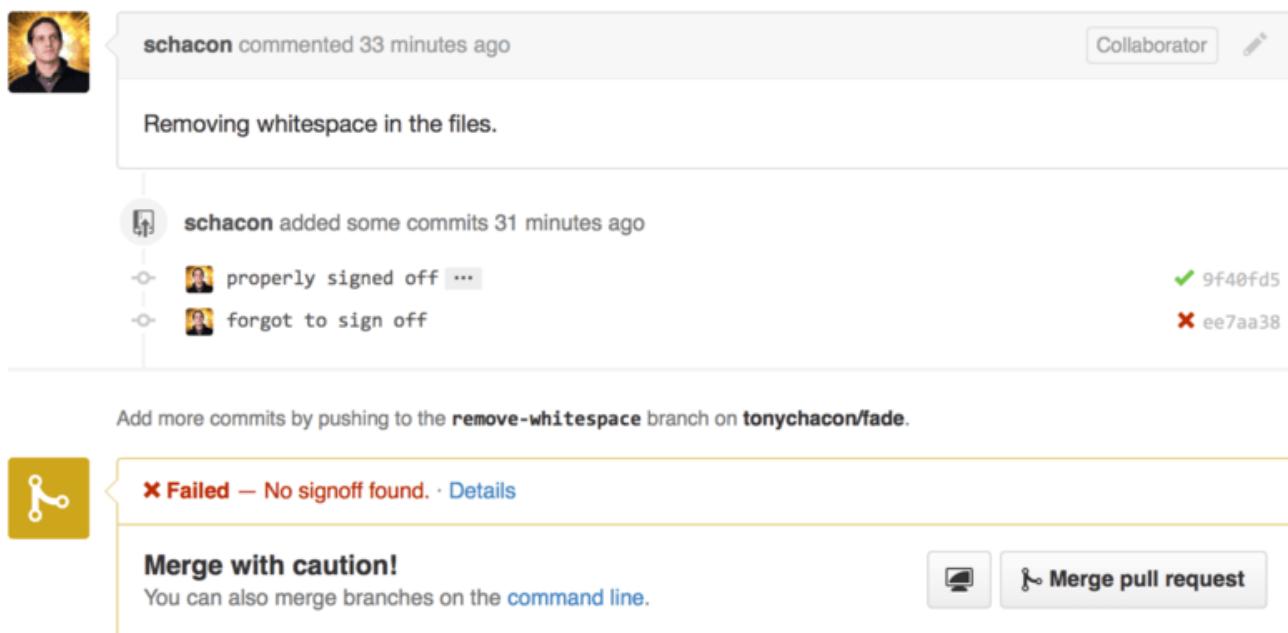


Figure 136. Commit status via de API.

Je kunt nu een klein groen vinkje zien naast de commit die "Signed-off-by" in het bericht heeft staan en een rode kruis door die waar de auteur is vergeten te tekenen. Je kunt ook zien dat de Pull Request de status krijgt van de laatste commit op de branch en waarschuwt je als het niet geslaagd is. Dit is erg handig als je deze API gebruikt voor test resultaten, zodat je niet per ongeluk iets merget waar de laatste commit tests laat falen.

## Octokit

Alhoewel we ongeveer alles middels `curl` en eenvoudige HTTP requests hebben gedaan in deze voorbeelden, bestaan er diverse open source libraries die deze API in een meer taalspecifieke manier beschikbaar maken. Op het moment van schrijven, zijn de ondersteunde talen onder andere Go, Objective-C, Ruby en .NET. Kijk op <http://github.com/octokit> voor meer informatie hiervoor, omdat ze veel van de HTTP voor je afhandelen.

Hopelijk kunnen deze instrumenten je helpen om GitHub aan te passen en te wijzigen zodat deze beter werkt voor jouw specifieke workflows. Voor volledige documentatie over de hele API zowel als handleidingen voor veelvoorkomende taken, verwijzen we je naar <https://developer.github.com>.

## Samenvatting

Je bent nu een GitHub gebruiker. Je weet hoe je een account aanmaakt, een organisatie kunt onderhouden, repositories aanmaken en er naar pushen, hoe je bijdraagt aan projecten van anderen en bijdragen van anderen kunt accepteren. In het volgende hoofdstuk zal je kennismaken met krachtigere instrumenten en tips krijgen hoe met complexe situaties om te gaan waarna je een echte meester zult zijn in het gebruik van Git.

# Git Tools

Op dit moment heb je de meeste van de alledaagse commando's en workflows geleerd die je nodig hebt om een Git repository te onderhouden of te beheren voor het bijhouden van je eigen bron code. Je hebt de basale taken van tracken en committen van bestanden volbracht, en je hebt de kracht van de staging area onder de knie gekregen en het lichtgewicht topic branchen en mergen.

Nu zal je een aantal erg krachtige dingen die Git kan doen onderzoeken die je niet perse elke dag zult doen maar die je op een gegeven moment wel nodig kunt hebben.

## Revisie Selectie

Git laat je op verschillende manieren specifieke commits of een reeks van commits angeven. Ze zijn niet echt voor de hand liggend, maar zijn zeer nuttig om te kennen.

### Enkele revisions

Je kunt uiteraard refereren aan een commit met de SHA-1 hash die eraan is gegeven, maar er zijn ook meer mens-vriendelijke manieren om aan commits te referen. Deze paragraaf toont de verschillende manieren waarmee je kunt refereren aan een enkele commit.

#### Verkorte SHA-1

Git is slim genoeg om uit te knobbelen welke commit je wilde typen als je de eerste paar karakters geeft, zolang als je verkorte SHA-1 op z'n minst 4 karakters lang is en eenduidig; dus, slechts één object in de huidige repository begint met dat deel-SHA-1.

Bijvoorbeeld, om een specifieke commit te zien, stel dat je een `git log` commando gebruikt en de commit waar je een bepaalde functie hebt toegevoegd identificeert:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

Stel dat we geïnteresseerd zijn in de commit waarvan de hash begint met **1c002dd.....**. Je kunt deze commit bekijken met elk van de volgend variaties van het **git show** commando (aangenomen dat de kortere versies uniek zijn):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git kan met een korte, unieke afkorting van je SHA-1 waarden overweg. Als je **--abbrev-commit** doorgeeft aan het **git log** commando, zal de uitvoer de korte waarden gebruiken, maar ze uniek houden; het gebruikt standaard zeven karakters, maar zal ze langer maken als dat nodig is om de SHA-1 eenduidig te houden:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

Over het algemeen zijn acht tot tien karakters meer dan genoeg om binnen een project uniek te zijn. Om een voorbeeld te geven: per juni 2018 bevat de Linux kernel (wat een redelijk groot project is) meer dan 700.000 commits en bijna zes en half miljoen objecten in de object database, waarin geen twee objecten waavan de SHA-1s in de eerste 11 karakters gelijk zijn.

## EEN KORTE NOOT OVER SHA-1

Veel mensen zijn bezorgd dat op een gegeven moment ze, door domme toeval, twee objecten in hun repository hebben waarvan de hash dezelfde SHA-1 waarde is. Wat dan?

Als het je overkomt dat je een object commit dat naar dezelfde SHA-1 waarde hasht als een vorig *ander* object in je repository, zal Git het vorige object in je Git database zien, aannemen dat het al was weggeschreven en simpelweg herbruiken. Als je op dat moment dat object weer zou gaan uitchecken, zal je altijd de gegevens van het eerste object krijgen.

Echter, je moet beseffen hoe belachelijk onwaarschijnlijk dit scenario is. De SHA-1 cijferruimte is 20 bytes of 160 bits. De hoeveelheid willekeurig gehashde objecten die nodig zijn om een 50% waarschijnlijkheid van een enkele *botsing* te garanderen is ongeveer  $2^{80}$  (de formule om de waarschijnlijkheid van een botsing te bepalen is  $p = (n \cdot (n-1)/2) * (1/2^{160})$ ).  $2^{80}$  is  $1,2 \times 10^{24}$  of 1 miljoen miljard miljard. Dat is 1.200 keer het aantal zandkorrels op de aarde.

Hier is een voorbeeld om je een idee te geven wat er nodig is om een SHA-1 botsing te krijgen. Als alle 6,5 miljard mensen op Aarde zouden programmeren, en elke seconde zou elk van hen code opleveren ter grootte aan de gehele Linux kernel historie (6,5 miljoen Git objecten) en deze pushen naar een gigantische Git repository, zou het ongeveer 2 jaar duren voordat de repository genoeg objecten zou bevatten om een 50% waarschijnlijkheid te krijgen van een enkele SHA-1 object botsing. Er is een grotere kans dat elk lid van je programmeerteam wordt aangevallen en gedood door wolven in ongerelateerde gebeurtenissen op dezelfde avond.



## Branch referenties

Een directe manier om naar een specifieke commit te verwijzen is als het de commit is aan de punt van een branch; in dat geval kan je de branchnaam in elke Git commando gebruiken die een referentie naar een commit verwacht. Bijvoorbeeld, als je het laatste commit object op een branch wilt laten zien, zijn de volgende commando's gelijk, aangenomen dat de `topic1`-branch wijst naar `ca82a6d`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949  
$ git show topic1
```

Als je wilt zien naar welke specifieke SHA-1 een branch wijst, of als je wilt zien waar elk van deze voorbeelden op neerkomt in termen van SHA-1s, kan je het Git binnenvwerk instrument (plumbing tool) geheten `rev-parse` gebruiken. Je kunt [Git Binnenwerk](#) bekijken voor meer informatie over plumbing tools; het komt erop neer dat `rev-parse` er is voor onder-water operaties en dat het niet bedoeld is voor het dagelijks gebruik. Dit gezegd hebbende, het kan soms handig zijn als het nodig is om te zien wat er echt gebeurt. Hier kan je `rev-parse` op je branch laten lopen.

```
$ git rev-parse topic1  
ca82a6dff817ec66f44342007202690a93763949
```

## RefLog verkorte namen

Een van de dingen die Git op de achtergrond doet als je aan het werk bent is een “reflog” bijhouden—een logboek waarin wordt bijgehouden waar je HEAD en branch referenties in de afgelopen paar maanden zijn geweest.

Je kunt de reflog zien door `git reflog` te gebruiken:

```
$ git reflog  
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated  
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the 'recursive' strategy.  
1c002dd HEAD@{2}: commit: added some blame and merge stuff  
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD  
95df984 HEAD@{4}: commit: # This is a combination of two commits.  
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD  
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Elke keer als de punt van je branch voor welke reden dan ook wordt bijgewerkt, slaat Git die informatie voor je op in deze tijdelijke historie. En je kunt ook aan oudere commits refereren met deze gegevens. Als je bijvoorbeeld de vijfde vorige waarde van de HEAD van je repository wilt zien, kan je de `@{5}` referentie gebruiken die je in de reflog uitvoer ziet:

```
$ git show HEAD@{5}
```

Je kunt deze syntax ook gebruiken om te zien waar een branch was op een specifieke moment in het verleden. Als je bijvoorbeeld wilt zien waar je `master`-branch gister was, kan je dit typen

```
$ git show master@{yesterday}
```

Dat toont je waar de punt van de `master`-branch gister was. Deze techniek werkt alleen voor gegevens die nog steeds in je reflog staan, dus je kunt het niet gebruiken om commits op te zoeken die ouder dan een paar maanden zijn.

Om de reflog informatie geformatteerd te tonen zoals de `git log` uitvoer, kan je `git log -g` uitvoeren:

```

$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'

```

Het is belangrijk op te merken dat de reflog informatie puur lokaal is—het is een log van wat *jij* gedaan hebt in *jouw* repository. De referentie zal niet hetzelfde zijn op de kopie van een ander van de repository, en direct nadat je initieel een kloon van een repository gemaakt hebt zal je een lege reflog hebben, omdat er nog geen activiteiten hebben plaatsgevonden op jouw repository. Het uitvoeren van `git show HEAD@{2.months.ago}` zal alleen werken als je het project op z'n minst 2 maanden geleden hebt gekloond—als je het recentelijker geleden hebt gekloond zal je alleen je eerste lokale commit zien.



*Zie de reflog als Git's versie van shell history*

Als je een UNIX of Linux achtergrond hebt, kan je de reflog zien als Git's versie van shell history, wat benadrukt dat wat er hier te zien is duidelijk alleen relevant is voor jou en jouw “sessie” en niets te doen heeft met anderen die misschien toevallig op dezelfde machine werken.

## Voorouder referenties

De andere veelgebruikte manier om een commit te specificeren is via zijn voorouders. Als je een `^` (caret) aan het eind van een referentie plaatst, zal Git dit interpreteren als een referentie aan de ouder van deze commit. Stel dat je naar de historie van je project kijkt:

```

$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list

```

Dan kan je de vorige commit zien door `HEAD^` te specificeren, wat “de ouder van HEAD” betekent:

```
$ git show HEAD^  
commit d921970aadf03b3cf0e71becdaab3147ba71cdef  
Merge: 1c002dd... 35cfb2b...  
Author: Scott Chacon <schacon@gmail.com>  
Date: Thu Dec 11 15:08:43 2008 -0800  
  
Merge commit 'phedders/rdocs'
```

#### *De caret escappen op Windows*

Voor `cmd.exe` op Windows, is `^` een speciaal teken en moet anders behandeld worden. Je kunt het verdubbelen of de referentie naar de commit in quotes zetten:



```
$ git show HEAD^      # werkt NIET op Windows  
$ git show HEAD^^    # OK  
$ git show "HEAD^"   # OK
```

Je kunt ook een getal aangeven na de `^`, bijvoorbeeld: `d921970^2` wat “de tweede ouder van d921970” betekent. Deze syntax is alleen nuttig voor merge commits, waar je meer dan een ouder hebt. De eerste ouder is de branch waar je op stond toen je mergede, en de tweede is de commit op de branch die je aan het in mergen was:

```
$ git show d921970^  
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b  
Author: Scott Chacon <schacon@gmail.com>  
Date: Thu Dec 11 14:58:32 2008 -0800  
  
        added some blame and merge stuff  
  
$ git show d921970^2  
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548  
Author: Paul Hedderly <paul+git@mjr.org>  
Date: Wed Dec 10 22:22:03 2008 +0000  
  
        Some rdoc changes
```

De andere belangrijke voorouder specificatie is de `~` (tilde). Deze refereert ook aan de eerste ouder, dus `HEAD~` en `HEAD^` zijn aan elkaar gelijk. Het verschil wordt duidelijk wanneer je een getal specificeert. `HEAD~2` betekent “de eerste ouder van de eerste ouder”, of “de grootouder”—het loopt het aantal keren terug over de eerste ouders dat je specificeert. Even weer als voorbeeld, in de historie van hiervoor, `HEAD~3` zou dit opleveren:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Dit kan ook als `HEAD~~~` worden geschreven, wat wederom de eerste ouder van de eerste ouder van de eerste ouder aanduidt:

```
$ git show HEAD~~~
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

    ignore *.gem
```

Je kunt deze syntaxen ook combineren; je kunt de tweede ouder van de vorige referentie krijgen (aangenomen dat het een merge commit was) door `HEAD~3^2` te gebruiken, en zo voort.

## Commit reeksen

Nu dat je individuele commits kunt aanwijzen, laten we eens kijken hoe je een reeks van commits kunt aanduiden. Dit is in het bijzonder nuttig voor het beheren van je branches—als je veel branches hebt, kan je reeks-specificaties gebruiken om vragen te beantwoorden als “Welk werk zit op deze branch die ik nog niet gemerged heb in mijn hoofdbranch?”.

### Tweevoudige punt

De meest gebruikte reeks specificatie is de tweevoudige punt (double-dot) syntax. Dit vraagt Git gewoon om een reeks commits op te halen die bereikbaar zijn van de ene commit maar niet vanaf een andere. Bijvoorbeeld, stel dat je een commit historie hebt die eruit ziet als [Voorbeeld historie voor reeks selectie..](#)

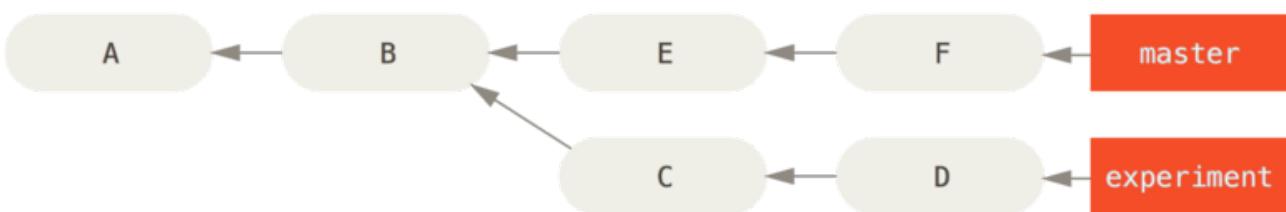


Figure 137. Voorbeeld historie voor reeks selectie.

Je wilt zien wat er in je `experiment`-branch zit wat nog niet in je `master`-branch gemerged is. Je kunt Git vragen om een log te laten zien van alleen die commits met `master..experiment`—hiermee wordt bedoeld “alle commits bereikbaar voor `experiment` die niet bereikbaar zijn voor `master`.” Om het kort en duidelijk te houden in deze voorbeelden, gebruik ik de letters van de commit objecten uit het diagram in plaats van de eigenlijke log uitvoer in de volgorde dat ze getoond

zouden zijn:

```
$ git log master..experiment  
D  
C
```

Als je, aan de andere kant, je het tegenovergestelde wilt zien—alle commits in `master` die niet in `experiment` zitten, kan je de branch namen omdraaien. `experiment..master` laat je alles in `master` zien wat niet vanuit `experiment` bereikbaar is:

```
$ git log experiment..master  
F  
E
```

Dit is nuttig als je de `experiment`-branch bij wilt houden en alvast wilt zien wat je op het punt staat in te mergen. Waar deze syntax ook vaak voor wordt gebruikt is om te zien wat je op het punt staat te pushen naar een remote:

```
$ git log origin/master..HEAD
```

Dit commando laat je alle commits in je huidige branch zien die niet in de `master`-branch zitten op je `origin`-remote. Als je een `git push` laat lopen en je huidige branch trackt `origin/master`, zijn de commits die worden getoond door `git log origin/master..HEAD` de commits die naar de server zullen worden gestuurd. Je kunt ook een kant van deze syntax weglaten om Git te laten aannemen dat hier `HEAD` wordt bedoeld. Bijvoorbeeld, kan je dezelfde resultaten bereiken als in het vorige voorbeeld door `git log origin/master..` te typen—Git gebruikt `HEAD` als een van de twee kanten ontbreekt.

## Meerdere punten

De twee-punten syntax is nuttig als een afkorting; maar misschien wil je meer dan twee branches aanwijzen om je revisie aan te geven, zoals het zien welke commits er zijn in een willekeurig aantal branches die niet in de branch zitten waar je nu op zit. Git staat je toe dit te doen door ofwel het `^` karakter te gebruiken of `--not` voor elke referentie waarvan je niet de bereikbare commits wilt zien. Dus deze drie commando's zijn gelijkwaardig:

```
$ git log refA..refB  
$ git log ^refA refB  
$ git log refB --not refA
```

Dit is nuttig omdat je met deze syntax meer dan twee referenties in je query kunt aangeven, wat je niet kunt doen met de dubbele-punt syntax. Als je bijvoorbeeld alle commits wilt zien die bereikbaar zijn vanaf `refA` of `refB` maar niet van `refC`, kan je een van deze intypen:

```
$ git log refA refB ^refC  
$ git log refA refB --not refC
```

Dit vormt een hele krachtige revisie-uitvraag-systeem die je kan helpen om uit te vinden wat er in je branches zit.

### Drievoudige punt

De laatste belangrijke reeks-selectie syntax is de drievoudige punt (triple dot) syntax, welke alle commits aanduidt die door *een van beide* referenties bereikbaar is maar niet door beide. Kijk even terug naar het voorbeeld van commit historie in [Voorbeeld historie voor reeks selectie..](#) Als je wilt zien wat in `master` of `experiment` zit maar geen gedeelde referenties kan je dit laten lopen

```
$ git log master...experiment  
F  
E  
D  
C
```

Wederom, dit geeft je een normale `log` uitvoer, maar laat je alleen de commit informatie zien voor deze vier commits, getoond op de reguliere commit datum-volgorde.

Een gebruikelijke optie om te gebruiken bij het `log` commando in dit geval is `--left-right`, welke je laat zien welke zijde van de reeks elke commit in zit. Dit helpt om de gegevens meer bruikbaar te maken:

```
$ git log --left-right master...experiment  
< F  
< E  
> D  
> C
```

Met deze instrumenten kan je eenvoudiger Git laten weten welke commit of commits je wilt onderzoeken.

## Interactief stagen

Hier gaan we een blik werpen op een paar interactieve Git commando's die je kunnen helpen om eenvoudig in je commits alleen bepaalde combinaties en delen van bestanden op te nemen. Deze instrumenten zijn zeer handig als je een aantal bestanden wijzigt en dan besluit dat je deze wijzigingen alleen in een aantal gerichte commits wilt hebben in plaats van een grote warrige commit. Op deze manier kan je ervoor zorgen dat je commits logisch onderscheiden wijzigingsgroepen (change sets) zijn en daarmee eenvoudig te reviewen door de ontwikkelaars die met je samenwerken.

Als je `git add` aanroeft met de `-i` of `--interactive` optie, zal Git in een interactieve schil modus

schakelen, en je iets als het volgende tonen:

```
$ git add -i
      staged      unstaged path
1: unchanged      +0/-1 TODO
2: unchanged      +1/-1 index.html
3: unchanged      +5/-1 lib/simplegit.rb

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now>
```

Je kunt zien dat dit commando je een heel andere kijk op je staging area geeft — eigenlijk dezelfde informatie die je krijgt met `git status` maar wat beknopter en meer informatief. Het lijst de wijzigingen die je gestaged hebt aan de linker- en de unstagede wijzigingen aan de rechterkant.

Hierna volgt een Commands gedeelte. Hier kan je een aantal dingen doen, waaronder bestanden staggen en unstagen, delen van bestanden staggen, untracked bestanden toevoegen en diffs zien van wat gestaged is.

## Bestanden staggen en unstagen

Als je `2` of `u` typt achter de `What now>` prompt, zal het script je vragen welke bestanden je wilt staggen:

```
What now> 2
      staged      unstaged path
1: unchanged      +0/-1 TODO
2: unchanged      +1/-1 index.html
3: unchanged      +5/-1 lib/simplegit.rb
Update>>
```

Om de `TODO` en `index.html` bestanden te staggen, kan je de nummers typen:

```
Update>> 1,2
      staged      unstaged path
* 1: unchanged      +0/-1 TODO
* 2: unchanged      +1/-1 index.html
3: unchanged      +5/-1 lib/simplegit.rb
Update>>
```

De `*` naast elk bestand geeft aan dat het bestand geselecteerd is om te worden gestaged. Als je Enter drukt als je niets typt achter de `Update>>` prompt, zal Git alles wat geselecteerd is voor je staggen:

```

Update>>
updated 2 paths

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now> 1
          staged      unstaged path
1:          +0/-1      nothing TODO
2:          +1/-1      nothing index.html
3:      unchanged      +5/-1 lib/simplegit.rb

```

Nu kan je zien dat de **TODO** en **index.html** bestanden gestaged zijn en het **simplegit.rb** bestand nog steeds unstaged is. Als je het **TODO** bestand op dit moment wilt unstagen, kan je de **3** of **r** (voor revert) optie gebruiken:

```

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now> 3
          staged      unstaged path
1:          +0/-1      nothing TODO
2:          +1/-1      nothing index.html
3:      unchanged      +5/-1 lib/simplegit.rb
Revert>> 1
          staged      unstaged path
* 1:          +0/-1      nothing TODO
2:          +1/-1      nothing index.html
3:      unchanged      +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path

```

Als je weer naar je Git status kijkt, kan je zien dat het **TODO** bestand unstaged is:

```

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch      6: diff        7: quit        8: help
What now> 1
          staged      unstaged path
1:      unchanged      +0/-1 TODO
2:          +1/-1      nothing index.html
3:      unchanged      +5/-1 lib/simplegit.rb

```

Om de diff te zien van wat je gestaged hebt, kan je het **6** of **d** (voor diff) commando gebruiken. Het laat je een lijst zien van staged bestanden, en je kunt diegenen selecteren waar je de staged diff van wilt bekijken. Dit is vergelijkbaar met het specificeren van **git diff --cached** op de command line:

```

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch       6: diff        7: quit        8: help
What now> 6
          staged      unstaged path
1:           +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

<p id="out">...</p>

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">

```

Met deze simpele commando's kan je de interactieve toevoeg methode gebruiken om je staging area iets eenvoudiger te bewerken.

## Patches stagen

Het is ook mogelijk om Git bepaalde *delen* van bestanden te laten staggen en de rest niet. Bijvoorbeeld, als je twee wijzigingen maakt in het `simplegit.rb` bestand en de ene wilt staggen en de andere niet, is dit erg eenvoudig te doen in Git. Vanaf dezelfde interactieve prompt die is uitgelegd in de vorige paragraaf, type je `5` of `p` (voor patch). Git zal je vragen welke bestanden je deels wilt staggen; vervolgens, voor elke sectie van de geselecteerde bestanden, zal het een voor een brokken van de bestands-diff laten zien en vragen of je deze wilt staggen:

```

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
  end

  def log(treeish = 'master')
-   command("git log -n 25 #{treeish}")
+   command("git log -n 30 #{treeish}")
  end

  def blame(path)
Stage this hunk [y,n,a,d/,,j,J,g,e,?]?

```

Je hebt hier veel keuzes. Het typen van `?` laat je een lijst zien van wat je kunt doen:

```

Stage this hunk [y,n,a,d,/,,j,J,g,e,?] ? 
y - stage this hunk
n - do not stage this hunk
a - stage this and all the remaining hunks in the file
d - do not stage this hunk nor any of the remaining hunks in the file
g - select a hunk to go to
/ - search for a hunk matching the given regex
j - leave this hunk undecided, see next undecided hunk
J - leave this hunk undecided, see next hunk
k - leave this hunk undecided, see previous undecided hunk
K - leave this hunk undecided, see previous hunk
s - split the current hunk into smaller hunks
e - manually edit the current hunk
? - print help

```

Over het algemeen zal je `y` of `n` typen als je elk deel wilt staggen, maar alle delen in bepaalde bestanden staggen of een besluit om een deel voor nu even over te slaan kan ook handig zijn. Als je het ene deel van het bestand staged en een ander deel unstaged laat, zal je status uitvoer er zo uitzien:

```

What now> 1
      staged      unstaged path
1:   unchanged      +0/-1 TODO
2:      +1/-1      nothing index.html
3:      +1/-1      +4/-0 lib/simplegit.rb

```

De status van het `simplegit.rb` bestand is interessant. Het laat je zien dat een aantal regels staged zijn en een aantal unstaged. Je hebt het bestand deels gestaged. Nu kan je het interactieve toevoegscript verlaten en `git commit` uitvoeren om de deels gestagede bestanden te committen.

Je hoeft overigens niet in de interactieve toevoeg modus te zijn om het staggen van bestandsdelen te doen — je kunt hetzelfde script starten door `git add -p` of `git add --patch` op de commando regel te typen.

Bovendien kan je de patch modus gebruiken om bestanden deels te resetten met het `reset --patch` commando, om delen van bestanden uit te checken met het `checkout --patch` commando en om delen van bestanden te stashen met het `stash save --patch` commando. We zullen meer details geven van elk van deze als we de meer gevorderde toepassingen van deze commando's gaan behandelen.

## Stashen en opschonen

Vaak, als je aan het werk bent geweest aan een deel van je project, zijn zaken in een rommelige staat en wil je naar andere branches omschakelen om wat aan iets anders te werken. Het probleem hier is dat je geen commit van half-compleet werk wilt doen, met als enige reden om later hiernaar terug te kunnen keren. Het antwoord op dit probleem is het `git stash` commando.

Stashen (even opzij leggen) pakt de onafgewerkte status van je werk directory—dat wil zeggen: je gewijzigde tracked bestanden en gestagede wijzigingen—en bewaart dit op een stapel met incomplete wijzigingen die je op elk willekeurig moment kunt afspelen.

#### Migreren naar `git stash push`

Per achterin oktober 2017 was er een uitvoerige discussie op de Git maillijst, waarbij het commando `git stash save` achterhaald (deprecated) is verklaard ter faveure van het bestaande alternatief `git stash push`. De belangrijkste reden hiervoor is dat `git stash push` de optie introduceert van het stashen van geselecteerde \_pathspecs~, iets wat `git stash save` niet ondersteund.

`git stash save` blijft voorlopig nog wel rondhangen, dus maak je er geen zorgen over dat het plotseling verdwijnt. Maar je kunt erover nadenken om te beginnen migreren naar het `push` alternatief voor de nieuwe functionaliteit.

## Je werk stashen

Om de werking te laten zien, ga je naar je project en begint te werken aan een aantal bestanden en mogelijk stage je een van de wijzigingen. Als je `git status` gebruikt, kan je je *vuile* status zien:

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

    modified:   index.html

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

    modified:   lib/simplegit.rb
```

Nu wil je naar een andere branch omschakelen, maar je wilt hetgeen waar je aan hebt gewerkt nog niet committen; dus ga je de wijzigingen stashen. Om een nieuwe stash op de stapel (stack) te duwen, roep je `git stash` of `git stash save` aan:

```
$ git stash
Saved working directory and index state \
  "WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

Je nu zien dat je werk directory schoon is:

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

Op dit moment kan je eenvoudige branches switchen en werk elders doen; je wijzigingen zijn bewaard op je stack. Om te zien welke stashes je bewaard hebt, kan je `git stash list` gebruiken:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

In dit voorbeeld, zijn er twee stashes eerder gedaan, dus je hebt toegang tot drie verschillende gestashde werken. Je kunt die ene die je zojuist gestashd hebt weer afspelen door het commando zoals getoond in de help uitvoer van het oorspronkelijke stash commando: `git stash apply`. Als je een van de oudere stashes wilt afspelen, kan je deze aangeven door de naam gebruiken, zoals dit: `git stash apply stash@{2}`. Als je geen stash aangeeft, gebruikt Git de meest recente stash en probeert deze af te spelen:

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

    modified:   index.html
    modified:   lib/simplegit.rb

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

Je kunt zien dat Git de bestanden weer wijzigt die je teruggedraaid had toen je de stash bewaarde. In dit geval had je een schone werk directory toen je de stash probeerde af te spelen en je probeerde deze af te spelen op dezelfde branch als waarvan je het had bewaard. Maar het hebben van een schone werk directory en deze op dezelfde branch af te spelen zijn niet nodig om een stash succesvol af te spelen. Je kunt een stash op de ene branch bewaren, later overschakelen naar een andere branch en daar de wijzigingen proberen opnieuw af te spelen. Je kunt ook gewijzigde en ongecommitte bestanden in je werkdirectory hebben als je een stash afspeelt — Git geeft je merge conflicten als iets niet meer netjes kan worden toegepast.

De wijzigingen aan je bestanden werden opnieuw gemaakt, maar het bestand dat je voorheen gestaged had wordt niet opnieuw gestaged. Om dat te doen, moet de het `git stash apply` commando met een `--index` optie gebruiken om het commando te vertellen om de gestagede wijzigingen opnieuw af te spelen. Als je deze variant had gebruikt, zou je je oorspronkelijke situatie terug hebben gekregen:

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

    modified:   index.html

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

    modified:   lib/simplegit.rb
```

De `apply` optie probeert alleen het gestashde werk opnieuw af te spelen — je blijft het behouden op je stack. Om het weg te halen, kan je `git stash drop` gebruiken met de naam van de stash die moet worden verwijderd:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Je kunt ook `git stash pop` gebruiken om de stash af te spelen en dan meteen te verwijderen uit je stack.

## Creatief stashen

Er zijn een paar stash varianten die ook handig kunnen zijn. De eerste variant die redelijk populair is, is de `--keep-index` optie bij het `stash save` commando. Dit vertelt Git niet alleen alle gestagede inhoud mee te nemen in de te creeren stash, en tegelijkertijd het in de index te laten.

```
$ git status -s
M  index.html
M  lib/simplegit.rb

$ git stash --keep-index
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
M  index.html
```

Een ander veel voorkomende toepassing van stash is om de niet getrackte bestanden te stashen zowel als de getrackte. Standaard zal `git stash` alleen *tracked* bestanden die gewijzigd en staged zijn opslaan. Als je `--include-untracked` of `'-u'` gebruikt zal git ook alle niet getrackte bestanden die

je gemaakt hebt stashen.

```
$ git status -s
M index.html
M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

Als laatste, als je de `--patch` optie gebruikt, zal Git niet alles wat is gewijzigd stashen maar zal in plaats je daarvan interactief vragen welk van de wijzigingen je wilt stashen en welke je in je werk directory wilt behouden.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
      return '#{git_cmd} 2>&1'.chomp
    end
  end
+
+  def show(treeish = 'master')
+    command("git show #{treeish}")
+  end
end
test
Stash this hunk [y,n,q,a,d,,e,?]?
```

Saved working directory and index state WIP on master: 1b65b17 added the index file

## Een branch vanuit een stash maken

Als je wat werk gestashed hebt, dit daar een tijdje bewaard, en doorgaat op de branch van waaruit je het werk gestashd hebt, kan je een probleem krijgen bij het weer afspelen hiervan. Als het afspelen een bestand wil wijzigen wat je daarna weer gewijzigd hebt, zal je een merge conflict krijgen en zul je het moeten proberen op te lossen. Als je een makkelijker manier wilt om de gestashde wijzigingen weer te testen, kan je `git stash branch <branch>` gebruiken, wat een nieuwe branch voor je aanmaakt, checkt de commit uit waar je op stond toen je je werk stashde, speelt de wijzigingen af op je werk daar en verwijdert de stash als het succesvol heeft kunnen toepassen:

```
$ git stash branch testchanges
M index.html
M lib/simplegit.rb
Switched to a new branch 'testchanges'
On branch testchanges
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   index.html

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

    modified:   lib/simplegit.rb

Dropped refs/stash@{0} (29d385a81d163dfd45a452a2ce816487a6b8b014)
```

Dit is een mooie manier om gestashde werk eenvoudig terug te halen en eraan te werken in een nieuwe branch.

## Je werk directory opruimen

Als laatste wil je misschien bepaald werk of sommige bestanden in je werk directory stashen, maar dit simpelweg verwijderen. Het `git clean` commando zal dit voor je doen.

Een eenvoudige reden hiervoor kan zijn dat het verwijderen van restjes zijn die zijn gegenereerd door merges of andere tools of om bouw-producten te verwijderen om een nette build te maken.

Je moet wel erg voorzichtig zijn met dit commando, het is namelijk gemaakt om bestanden van je werk directory weg te halen die niet worden getracked. Als je je later bedenkt, is er vaak niet mogelijk om de inhoud van die bestanden terug te halen. Een veiligere optie is om `git stash --all` te gebruiken om alles weg te halen, en dit alles in een stash te bewaren.

Aangenomen dat je de overbodige bestanden wilt verwijderen of je werk directory wilt opruimen, kan je dat doen met `git clean`. Om alle ongetrackte bestanden uit je werk directory te verwijderen kan je het `git clean -f -d` commando aanroepen, die alle bestanden verwijdert en ook alle subdirectories die leeg zijn geraakt als gevolg hiervan. De `-f` betekent *forceer* of "doe dit nou maar gewoon".

Als je ooit wilt zien wat het zou gaan doen, kan je het commando met de `-n` optie aanroepen, wat "doe een generale repetitie en vertel me wat je zou hebben verwijderd".

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

Standaard zal het `git clean` commando alleen ongetrackte bestanden verwijderen die niet

genegeerd worden. Elk bestand waarvan de naam overeenkomt met een patroon in je `.gitignore` of andere negeer (ignore) bestanden zullen niet verwijderd worden. Als je ook deze bestanden wilt verwijderen, bijvoorbeeld om alle `.o` bestanden te verwijderen die zijn gegenereerd tijdens een build zodat je een compleet schone build kunt doen kan je een `-x` toevoegen aan het opschoon-commando.

```
$ git status -s
M lib/simplegit.rb
?? build.TMP
?? tmp/

$ git clean -n -d
Would remove build.TMP
Would remove tmp/

$ git clean -n -d -x
Would remove build.TMP
Would remove test.o
Would remove tmp/
```

Als je niet weet wat het `git clean` commando zal gaan doen, roep deze dan altijd eerst aan met een `-n` voor de zekerheid voordat je de `-n` in een `-f` wijzigt en het definitief doet. Een alternatief voor het voorzichtig uitvoeren van dit proces is om het aan te roepen met de `-i` of “interactive” optie.

Dit zal het opschoon-commando in een interactieve modus aanroepen.

```
$ git clean -x -i
Would remove the following items:
  build.TMP  test.o
*** Commands ***
  1: clean           2: filter by pattern   3: select by numbers   4: ask
each            5: quit
  6: help
What now>
```

Op deze manier kan je alle bestanden een voor een afgaan of bepaalde patronen voor verwijdering aangeven.

 Er is een specifiek vreemde situatie waarin je extra overtuigend moet zijn om Git te vragen je werk directory schoon te maken. Als je in een werk directory zit waaronder je een andere Git repository hebt gekopieerd of gekloond (misschien als submodules), zal zelfs `git clean -fd` weigeren deze directories te verwijderen. In zulke gevallen zal je een tweede `-f` optie moeten toevoegen voor extra nadruk.

## Je werk tekenen

Git is cryptografisch veilig, maar beschermt je niet tegen dommigheden. Als je werk overneemt van

anderen op het internet en je wilt verifiëren dat commits van een betrouwbare bron komen, heeft Git een aantal manieren om werk te tekenen en verifiëren met GPG.

## PGP Introductie

Allereerst, als je iets wilt tekenen zal je eerst GPG moeten hebben geconfigureerd en je persoonlijke sleutel geïnstalleerd.

```
$ gpg --list-keys  
/Users/schacon/.gnupg/pubring.gpg  
-----  
pub 2048R/0A46826A 2014-06-04  
uid Scott Chacon (Git signing key) <schacon@gmail.com>  
sub 2048R/874529A9 2014-06-04
```

Als je geen sleutel geïnstalleerd hebt, kan je er een genereren met `gpg --gen-key`.

```
gpg --gen-key
```

Als je eenmaal een privé sleutel hebt om mee te tekenen kan je Git configureren om deze te gebruiken bij het tekenen van spullen door de `user.signingkey` configuratie sleutel te zetten.

```
git config --global user.signingkey 0A46826A
```

Nu zal Git standaard jouw sleutel gebruiken om tags en commits te tekenen als je dat wilt.

## Tags tekenen

Als je een GPG privé sleutel aangemaakt hebt, kan je deze gebruiken om nieuwe tags te tekenen. Al wat je hoeft te doen is `-s` te gebruiken in plaats van `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'  
  
You need a passphrase to unlock the secret key for  
user: "Ben Straub <ben@straub.cc>"  
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

Als je nu `git show` aanroeft op die tag, kan je jouw GPG handtekening erbij zien staan:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcbAABAgAGBQJTzbQIAoJEF0+sviABDDrZbQH/09PfE51KPVPanr6q1v4/Ut
LQxfojUWiLQdg2ESJItkcuweYg+kC3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm
ecorc4ixZQu7tupRihslbNkfvcimnSDeSvzCpWAH17h8Wj6hhqePmLm91AYqnKp
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi
RUysgqjcpT8+iQM1Pb1GfHR4XAh0qN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=
=EFTF
-----END PGP SIGNATURE-----
```

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

changed the version number

## Tags verifiëren

Om een getekende tag te verifiëren, gebruik je `git tag -v <tag-naam>`. Dit commando gebruikt GPG om de handtekening te verifiëren. Je hebt de publieke sleutel van de tekenaar nodig in je sleutelbos (keyring) om dit goed te laten werken:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"
gpg:                               aka "[jpeg image of size 1513]"
Primary key fingerprint: 3565 2A26 2040 E066 C9A7  4A7D C0C6 D9A4 F311 9B9A
```

Als je de publieke sleutel van de tekenaar niet hebt, krijg je iets als dit:

```
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A
gpg: Can't check signature: public key not found
error: could not verify the tag 'v1.4.2.1'
```

## Commits tekenen

In meer recente versie van Git (v1.7.9 en later), kan je ook individuele commits tekenen. Als je geïnteresseerd bent in het tekenen van ook de commits in plaats van alleen de tags, is alles wat je hoeft te doen een `-S` toe te voegen aan je `git commit` commando.

```
$ git commit -a -S -m 'signed commit'

You need a passphrase to unlock the secret key for
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"
2048-bit RSA key, ID 0A46826A, created 2014-06-04

[master 5c3386c] signed commit
 4 files changed, 4 insertions(+), 24 deletions(-)
 rewrite Rakefile (100%)
 create mode 100644 lib/git.rb
```

Om deze handtekeningen te zien en te verifiëren, is er ook een `--show-signature` optie bij `git log`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun 4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

      signed commit
```

Bovendien kan je `git log` configureren om alle handtekeningen die het vind te controleren en deze te laten zien in de uitvoer met het `%G?` formaat.

```
$ git log --pretty=format:%h %G? %aN  %s"
5c3386c G Scott Chacon  signed commit
ca82a6d N Scott Chacon  changed the version number
085bb3b N Scott Chacon  removed unnecessary test code
a11bef0 N Scott Chacon  first commit
```

Hier kunnen we zien dat alleen de laatste commit is getekend en geldig en de eerdere commits niet.

In Git 1.8.3 en later, kunnen "git merge" en "git pull" verteld worden om te controleren tijdens het mergen van een commit en deze af te wijzen als die geen geverifieerde GPG handtekening heeft met

het `--verify-signature` commando.

Als je deze optie gebruikt tijdens het mergen van een branch en die commits bevat die niet getekend en geldig zijn zal de merge niet slagen.

```
$ git merge --verify-signatures non-verify  
fatal: Commit ab06180 does not have a GPG signature.
```

Als de merge alleen maar geldig getekende commits bevat, zal het merge commando je alle handtekeningen laten zien die het heeft gecontroleerd en daarna doorgaan met de merge.

```
$ git merge --verify-signatures signed-branch  
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)  
<schacon@gmail.com>  
Updating 5c3386c..13ad65e  
Fast-forward  
 README | 2 ++  
 1 file changed, 2 insertions(+)
```

Je kunt ook de `-S` optie gebruiken met het `git merge` commando om de merge die het resultaat hiervan is te tekenen. Het volgende voorbeeld zal zowel elke commit in de te mergen branch verifiëren als de resulterende merge commit tekenen.

```
$ git merge --verify-signatures -S signed-branch  
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)  
<schacon@gmail.com>  
  
You need a passphrase to unlock the secret key for  
user: "Scott Chacon (Git signing key) <schacon@gmail.com>"  
2048-bit RSA key, ID 0A46826A, created 2014-06-04  
  
Merge made by the 'recursive' strategy.  
 README | 2 ++  
 1 file changed, 2 insertions(+)
```

## Iedereen moet tekenen

Tags en commits tekenen is geweldig, maar als je besluit dit te gaan gebruiken in je reguliere workflow, moet je er zeker van zijn dat iedereen in je team begrijpt hoe dit te doen. Als je dat niet doet, ga je erg veel tijd kwijt zijn met het uitleggen aan mensen hoe ze hun commits moeten vervangen met getekende versies. Zorg ervoor dat je GPG begrijpt en het voordeel van getekende dingen voordat je dit gaat gebruiken als onderdeel van je reguliere workflow.

## Zoeken

Met zo ongeveer elke formaat codebase, zal je vaak de behoeft hebben om uit te zoeken waar een

functie wordt aangeroepen of gedefinieerd, of de historie van een methode laten zien. Git heeft een aantal handige instrumenten om snel en eenvoudig door de code en commits die in de database staan te zoeken. We zullen er hier een aantal behandelen.

## Git Grep

Git wordt geleverd met een commando genaamd `grep` wat je in staat stelt om eenvoudig door elke gecommitte boomstructuur (tree), de werk directory of zelfs de index te zoeken naar een woord of reguliere expressie (regular expression). Voor de volgende voorbeelden zoeken we door de broncode van Git zelf.

Standaard zal het door de bestanden in je werk directory zoeken. Je kunt `-n` of `--line-number` optie doorgeven om de regelnummers af te drukken waar Git resultaten heeft gevonden.

```
$ git grep -n gmtime_r
compat/gmtime.c:3:#undef gmtime_r
compat/gmtime.c:8:    return git_gmtime_r(timep, &result);
compat/gmtime.c:11:struct tm *git_gmtime_r(const time_t *timep, struct tm *result)
compat/gmtime.c:16:    ret = gmtime_r(timep, result);
compat/mingw.c:826:struct tm *gmtime_r(const time_t *timep, struct tm *result)
compat/mingw.h:206:struct tm *gmtime_r(const time_t *timep, struct tm *result);
date.c:482:            if (gmtime_r(&now, &now_tm))
date.c:545:            if (gmtime_r(&time, tm)) {
date.c:758:            /* gmtime_r() in match_digit() may have clobbered it */
git-compat-util.h:1138:struct tm *git_gmtime_r(const time_t *, struct tm *);
git-compat-util.h:1140:#define gmtime_r git_gmtime_r
```

Er zijn een groot aantal interessante opties die je aanvullend bij het `git grep` commando kunt gebruiken.

Bijvoorbeeld: in plaats van alle vondsten af te drukken kan je `git grep` vragen om de uitvoer samen te vatten door alleen te laten zien welke files het zoek-argument bevatte en hoeveel keer het in elk bestand gevonden werd met de `-c` of `--count` optie:

```
$ 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
```

Als je geïnteresseerd bent in de *context* van een zoek argument, kan je de omsluitende methode of functie voor elke gevonden string laten zien met een van de opties `-p` of `--show-function`:

```
$ git grep -p gmtime_r *.c
date.c=static int match_multi_number(timestamp_t num, char c, const char *date,
date.c:           if (gmtime_r(&now, &now_tm))
date.c=static int match_digit(const char *date, struct tm *tm, int *offset, int
*tm_gmt)
date.c:           if (gmtime_r(&time, tm)) {
date.c=int parse_date_basic(const char *date, timestamp_t *timestamp, int *offset)
date.c:           /* gmtime_r() in match_digit() may have clobbered it */
```

Hier kunnen we dus zien dat `gmtime_r` wordt aangeroepen in de `match_multi_number` en `match_digit` functies in het bestand date.c (de derde match laat alleen de gevonden string in een commentaar zien).

Je kunt ook zoeken naar ingewikkelde combinaties van woorden met de `--and` vlag, die ervoor zorgt dat er meerdere treffers moeten zijn op dezelfde regel. Bijvoorbeeld, laten we kijken of er regels zijn die een constante definiëren met het woord “LINK” of “BUF\_MAX” in de Git codebase met de tag `v1.8.0` versie (we zullen er ook nog de opties `--break` en `--heading` erbij gebruiken die helpen de uitvoer in een meer leesbaar formaat weer te geven):

```
$ git grep --break --heading \
-n -e '#define' --and \(-e LINK -e BUF_MAX \) v1.8.0
v1.8.0:builtin/index-pack.c
62:#define FLAG_LINK (1u<<20)

v1.8.0:cache.h
73:#define S_IFGITLINK 0160000
74:#define S_ISGITLINK(m) (((m) & S_IFMT) == S_IFGITLINK)

v1.8.0:environment.c
54:#define OBJECT_CREATION_MODE OBJECT_CREATIONUSES_HARDLINKS

v1.8.0:strbuf.c
326:#define STRBUF_MAXLINK (2*PATH_MAX)

v1.8.0:symlinks.c
53:#define FL_SYMLINK (1 << 2)

v1.8.0:zlib.c
30:/* #define ZLIB_BUF_MAX ((uInt)-1) */
31:#define ZLIB_BUF_MAX ((uInt) 1024 * 1024 * 1024) /* 1GB */
```

Het `git grep` commando heeft een aantal voordeelen boven reguliere zoek commando's zoals `grep` en `ack`. De eerste is dat het erg snel is, de tweede is dat je door elke tree in Git kunt zoeken, niet alleen de werk directory. Zoals we zagen in het bovenstaande voorbeeld, zochten we naar termen in een oudere versie van de Git broncode, niet de versie die op dat moment uitgecheckt was.

## Zoeken in de Git log

Misschien ben je niet op zoek naar *waar* een term bestaat, maar *wanneer* het bestond of was geïntroduceerd. Het `git log` commando heeft een aantal krachtige instrumenten om specifieke commits te vinden gebruik makende van hun berichten of zelfs de inhoud van de diff die erdoor werd geïntroduceerd.

Als we bijvoorbeeld willen uitzoeken wanneer de constante `ZLIB_BUF_MAX` voor het eerst werd geïntroduceerd, kunnen we Git vragen ons alleen de commits te tonen waarin dit woord werd toegevoegd of verwijderd met de `-S` optie.

```
$ git log -SZLIB_BUFS_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time
```

Als we naar de diff van deze commits kijken kunnen we zien dat in `ef49a7a` deze constante werd geïntroduceerd en in `e01503b` werd gewijzigd.

Als je nog specifieker moet zijn, kan je een regular expression opgeven om mee te zoeken met de `-G` optie.

## Zoeken in de regel-log

Een andere nogal gevorderde log zoekmethode die ongelofelijk nuttig is, is de regel historie zoekmethode. Gewoon `git log` aanroepen met de '-L optie, en het laat je de historie fan een functie of een regel code in je codebase zien.`

Bijvoorbeeld, als we elke wijziging willen zien die gemaakt is aan de functie `git_deflate_bound` in het bestand `zlib.c`, kunnen we `git log -L :git_deflate_bound:zlib.c` aanroepen. Dit zal proberen uit te vinden wat het begin en einde is van die functie en dan door de historie gaan en ons elke wijziging laten zien die aan deze functie gemaakt is als een reeks van patches tot waar de functie als eerste gemaakt was.

```
$ git log -L :git_deflate_bound:zlib.c
commit ef49a7a0126d64359c974b4b3b71d7ad42ee3bca
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:52:15 2011 -0700

    zlib: zlib can only process 4GB at a time

diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -85,5 +130,5 @@
-unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+unsigned long git_deflate_bound(git_zstream *strm, unsigned long size)
{
-    return deflateBound(strm, size);
+    return deflateBound(&strm->z, size);
}


```

```
commit 225a6f1068f71723a910e8565db4e252b3ca21fa
Author: Junio C Hamano <gitster@pobox.com>
Date:   Fri Jun 10 11:18:17 2011 -0700

    zlib: wrap deflateBound() too
```

```
diff --git a/zlib.c b/zlib.c
--- a/zlib.c
+++ b/zlib.c
@@ -81,0 +85,5 @@
+unsigned long git_deflate_bound(z_streamp strm, unsigned long size)
+{
+    return deflateBound(strm, size);
+}
+
```

Als Git niet kan uitvinden hoe een functie of methode in jouw programmeertaal kan worden gevonden, kan je het ook een reguliere expressie (of *regex*) meegeven. Bijvoorbeeld, dit zou tot hetzelfde resultaat als hierboven geleid hebben: `git log -L '/unsigned long git_deflate_bound/,/^}:/zlib.c`. Je kunt het ook een aantal regels als grens meegeven of een enkele regelnummer en je zult een vergelijkbare uitvoer krijgen.

## Geschiedenis herschrijven

Vaak zal je, als je met Git werkt, je commit geschiedenis om een of andere reden willen aanpassen. Eén van de mooie dingen van Git is dat het je in staat stelt om beslissingen op het laatst mogelijke moment te maken. Je kunt bepalen welke bestanden in welke commits gaan vlak voordat je commit, door middel van de staging area, je kunt besluiten dat je ergens toch nog niet aan had willen beginnen met het stash commando en je kunt commits herschrijven ook al zijn ze al

gebeurd, waardoor het lijkt alsof ze op een andere manier gebeurd zijn. Dit kan bijvoorbeeld de volgorde van de commits betreffen, berichten of bestanden in een commit wijzigen, commits samenpersen (squashen) of opsplitsen, of complete commits weghalen — en dat allemaal voordat je jouw werk met anderen deelt.

In deze paragraaf zal je leren hoe je deze handige taken uitvoert, zodat je jouw commit geschiedenis er uit kunt laten zien zoals jij dat wilt, voordat je het met anderen deelt.



Een van de meest belangrijke regels van Git is dat, omdat zoveel werk lokaal gedaan wordt binnen jouw kloon, een een hele hoge graad van vrijheid hebt om je history *lokaal* te herschrijven. Echter, als je je werk eenmaal hebt gepusht, wordt dit een heel ander verhaal, en je zult je gepushte werk moeten beschouwen als definitief tenzij je een hele goede reden hebt om het te wijzigen. Kortom: je moet het pushen van je werk vermijden tot je ermee tevreden bent en klaar bent om het met de rest van de wereld te delen.

## De laatste commit veranderen

De laatste commit veranderen is waarschijnlijk de meest voorkomende geschiedenis-wijziging die je zult doen. Vaak wil je twee basale dingen aan je laatste commit wijzigen: het commit bericht, of de snapshot dat je zojuist opgeslagen hebt, veranderen door het toevoegen, wijzigen of weghalen van bestanden.

Als je alleen je laatste commit bericht wilt wijzigen, dan is dat heel eenvoudig:

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

Dat plaatst je in de teksteditor met je laatste commit bericht erin, klaar voor je om het bericht te wijzigen en de editor sluiten. Als je opslaat en de editor sluit, dan schrijft de editor een nieuwe commit met dat bericht en maakt dat je nieuwe laatste commit.

Als je echter de echte *inhoud* van je laatste commit wilt wijzigen, werkt het proces feitelijk hetzelfde — eerst de wijzigingen doen die je dacht te hebben vergeten, deze staggen en de daaropvolgende `git commit --amend` vervangt die laatste commit met je nieuwe verbeterde commit.

Je moet wel oppassen met deze techniek, omdat het amenderen de SHA-1 van de commit wijzigt. Het is vergelijkbaar met een hele kleine rebase — niet je laatste commit wijzigen als je deze al gepusht hebt.

*Een geamendeerde commit kan (of niet) een geammendeerd commit-bericht nodig hebben*

Als je een commit ammendeert, heb je de kans om zowel het commit-bericht als de inhoud van de commit te wijzigen. Als je de inhoud van de commit substantieel wijzigt, moet je bijna zeker de commit-bericht wijzigen om deze gewijzigde inhoud weer te geven.



Aan de andere kant, als je ammenderingen erg triviaal zijn (een stomme typefout hertsellen of een bestand toevoegen die je was vergeten te staggen) waarbij het eerdere commit-bericht nog steeds juist is, kan je simpelweg de wijzigingen maken, deze staggen en de onnodige editor-sessie helemaal vermijden met:

```
$ git commit --amend --no-edit
```

## Meerdere commit berichten wijzigen

Om een commit te wijzigen die verder terug in je geschiedenis ligt, moet je meer complexe instrumenten gebruiken. Git heeft geen geschiedenis-wijzig tool, maar je kunt de rebase tool gebruiken om een serie commits op de HEAD te rebasen waarop ze origineel gebaseerd, in plaats van ze naar een andere te verhuizen. Met de interactieve rebase tool kun je dan na iedere commit die je wilt wijzigen stoppen en het bericht wijzigen, bestanden toevoegen, of doen wat je ook maar wilt. Je kunt rebase interactief uitvoeren door de `-i` optie aan `git rebase` toe te voegen. Je moet aangeven hoe ver terug je commits wilt herschrijven door het commando te vertellen op welke commit het moet rebasen.

Bijvoorbeeld, als je de laatste drie commit berichten wilt veranderen, of een van de commit berichten in die groep, dan geef je de ouder van de laatste commit die je wilt wijzigen mee als argument aan `git rebase -i`, wat `HEAD~2^` of `HEAD~3` is. Het kan makkelijker zijn om de `~3` te onthouden, omdat je de laatste drie commits probeert te wijzigen; maar hou in gedachten dat je eigenlijk vier commits terug aangeeft; de ouder van de laatste commit die je wilt veranderen:

```
$ git rebase -i HEAD~3
```

Onthoud, nogmaals, dat dit een rebase commando is - iedere commit in de reeks `HEAD~3..HEAD` zal worden herschreven, of je het bericht nu wijzigt of niet. Voeg geen commit toe die je al naar een centrale server gepusht hebt - als je dit doet breng je andere gebruikers in de war omdat je ze een alternatieve versie van dezelfde wijziging te geeft.

Dit commando uitvoeren geeft je een lijst met commits in je tekst editor die er ongeveer zo uit ziet:

```

pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out

```

Het is belangrijk om op te merken dat deze commits in de tegengestelde volgorde getoond worden dan hoe je ze normaliter ziet als je het `log` commando gebruikt. Als je een `log` uitvoert, zie je zo iets als dit:

```
$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit
```

Merk de omgekeerde volgorde op. De interactieve rebase geeft je een script dat het gaat uitvoeren. Het zal beginnen met de commit die je specificeert op de commando regel (`HEAD~3`) en de wijzigingen in elk van deze commits van voor naar achter opnieuw afspelen. Het toont de oudste het eerst in plaats van de nieuwste, omdat dat deze de eerste is die zal worden afgespeeld.

Je moet het script zodanig aanpassen dat het stopt bij de commit die je wilt wijzigen. Om dat te doen moet je het woord `pick` veranderen in het woord `edit` voor elke commit waarbij je het script wilt laten stoppen. Bijvoorbeeld, om alleen het derde commit bericht te wijzigen verander je het bestand zodat het er zo uitziet:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Als je dit opslaat en de editor sluit, spoelt Git terug naar de laatste commit van die lijst en zet je op de commando regel met de volgende boodschap:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with
```

```
git commit --amend
```

Once you're satisfied with your changes, run

```
git rebase --continue
```

Deze instructies vertellen je precies wat je moet doen. Type

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

Wijzig het commit bericht en verlaat de editor. Voer vervolgens dit uit

```
$ git rebase --continue
```

Dit commando zal de andere twee commits automatisch toepassen, en je bent klaar. Als je pick op meerdere regels in edit verandert, dan kan je deze stappen herhalen voor iedere commit die je in edit veranderd hebt. Elke keer zal Git stoppen, je de commit laten wijzigen en verder gaan als je klaar bent.

## Commits anders rangschikken

Je kunt een interactieve rebase ook gebruiken om commits anders te rangschikken of ze geheel te verwijderen. Als je de “added cat-file” commit wilt verwijderen en de volgorde waarin de andere twee commits zijn geïntroduceerd wilt veranderen, dan kun je het rebase script van dit

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

veranderen in dit:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

Als je dan opslaat en de editor sluit, zal Git je branch terugzetten naar de ouder van deze commits, eerst **310154e** en dan **f7f3f6d** toepassen, en dan stoppen. Feitelijk verander je de volgorde van die commits en verwijder je de “added cat-file”-commit volledig.

## Een commit samenpersen (squashing)

Het is ook mogelijk een serie commits te pakken en ze in één enkele commit samen te persen (squash) met de interactieve rebase tool. Het script stopt behulpzame instructies in het rebase bericht:

```
#  
# Commands:  
# p, pick = use commit  
# r, reword = use commit, but edit the commit message  
# e, edit = use commit, but stop for amending  
# s, squash = use commit, but meld into previous commit  
# f, fixup = like "squash", but discard this commit's log message  
# x, exec = run command (the rest of the line) using shell  
#  
# These lines can be re-ordered; they are executed from top to bottom.  
#  
# If you remove a line here THAT COMMIT WILL BE LOST.  
#  
# However, if you remove everything, the rebase will be aborted.  
#  
# Note that empty commits are commented out
```

Als je in plaats van “pick” of “edit”, “squash” specificeert zal Git zowel die verandering als de verandering die er direct aan vooraf gaat toepassen, en je helpen om de commit berichten samen te voegen. Dus als je een enkele commit van deze drie commits wil maken, laat je het script er zo uit zien:

```
pick f7f3f6d changed my name a bit  
squash 310154e updated README formatting and added blame  
squash a5f4a0d added cat-file
```

Als je de editor opslaat en sluit, zal Git alle drie wijzigingen toepassen en je terug in de editor brengen om de drie commit berichten samen te voegen:

```
# This is a combination of 3 commits.  
# The first commit's message is:  
changed my name a bit  
  
# This is the 2nd commit message:  
  
updated README formatting and added blame  
  
# This is the 3rd commit message:  
  
added cat-file
```

Als je dat opslaat, heb je een enkele commit die de veranderingen van alle drie vorige commits introduceert.

## Een commit splitsen

Een commit opsplitsen zal een commit ongedaan maken, en dan net zo vaak gedeeltelijk staggen en committen als het aantal commits waar je mee wilt eindigen. Bijvoorbeeld, stel dat je de middelste van je drie commits wilt splitsen. In plaats van “updated README formatting and added blame” wil je het splitsen in twee commits: “updated README formatting” als eerste, en “added blame” als tweede. Je kunt dat doen in het `rebase -i` script door de instructie van de commit die je wilt splitsen te veranderen in “edit”:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Dan, als het script je naar de commando regel neemt, reset je die commit, neemt de wijzigingen die zijn gereset en maakt meerdere commits ervan. Als je opslaat en de editor verlaat, spoelt Git terug naar de parent van de eerste commit in de lijst, past de eerste commit toe (`f7f3f6d`), past de tweede toe (`310154e`), en zet je dan in de console. Daar kan je een gemengde reset doen van die commit met `git reset HEAD^`, wat effectief de commit terugdraait en de gewijzigde bestanden unstaged laat. Nu kan je de wijzigingen die gereset zijn nemen en er meerdere commits van maken, en dan `git rebase --continue` uitvoeren zodra je klaar bent:

```
$ git reset HEAD^
$ git add README
$ git commit -m 'updated README formatting'
$ git add lib/simplegit.rb
$ git commit -m 'added blame'
$ git rebase --continue
```

Git zal de laatste commit (`a5f4a0d`) in het script toepassen, en je geschiedenis zal er zo uitzien:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Nogmaals, dit verandert alle SHA's van alle commits in de lijst, dus zorg er voor dat er geen commit in die lijst zit die je al naar een gedeelde repository gepusht hebt.

## De optie met atoomkracht: filter-branch

Er is nog een geschiedenis-herschrijvende optie, die je kunt gebruiken als je een groter aantal commits moet herschrijven op een gescripte manier. Bijvoorbeeld, het globaal veranderen van je e-

mail adres of een bestand uit iedere commit verwijderen. Dit commando heet **filter-branch** en het kan grote gedeelten van je geschiedenis herschrijven, dus je moet het niet gebruiken tenzij je project nog niet publiek is gemaakt, en andere mensen nog geen werk hebben gebaseerd op de commits die je op het punt staat te herschrijven. Echter het kan heel handig zijn. Je zult een paar veel gebruikte toepassingen zien zodat je een idee krijgt waar het toe in staat is.

### Een bestand uit iedere commit verwijderen

Dit gebeurt nog wel eens. Iemand voegt per ongeluk een enorm binair bestand toe met een achteloze `git add .`, en je wilt het overal weghalen. Misschien heb je per ongeluk een bestand dat een wachtwoord bevat gecommit, en je wilt dat project open source maken. **filter-branch** is dan het middel dat je wilt gebruiken om je hele geschiedenis schoon te poetsen. Om een bestand met de naam `passwords.txt` uit je hele geschiedenis weg te halen, kun je de `--tree-filter` optie toevoegen aan **filter-branch**:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

De `--tree-filter` optie voert het gegeven commando uit na elke checkout van het project, en commit de resultaten weer. In dit geval verwijder je een bestand genaamd `passwords.txt` van elke snapshot, of het bestaat of niet. Als je alle abusievelijk toegevoegde editor backup bestanden wilt verwijderen, kan je bijvoorbeeld dit uitvoeren `git filter-branch --tree-filter 'rm -f *~' HEAD`.

Je zult Git objectbomen en commits zien herschrijven en aan het eind de branch wijzer zien verplaatsen. Het is over het algemeen een goed idee om dit in een test branch te doen, en dan je master branch te hard-resetten nadat je gecontroleerd hebt dat de uitkomst echt is als je het wilt hebben. Om **filter-branch** op al je branches uit te voeren, moet je `--all` aan het commando meegeven.

### Een subdirectory de nieuwe root maken

Stel dat je een import vanuit een ander versiebeheersysteem hebt gedaan, en subdirectories hebt die nergens op slaan (`trunk`, `tags`, enzovoort). Als je de `trunk` subdirectory de nieuwe root van het project wilt maken voor elke commit, kan **filter-branch** je daar ook mee helpen:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cd8e8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Nu is de nieuwe project root elke keer de inhoud van de `trunk` subdirectory. Git zal ook automatisch commits verwijderen die geen betrekking hadden op die subdirectory.

### E-mail adressen globaal veranderen

Een ander veel voorkomende toepassing is het geval dat je vergeten bent om `git config` uit te voeren om je naam en e-mail adres in te stellen voordat je begon met werken, of misschien wil je

een project op het werk open source maken en al je werk e-mail adressen veranderen naar je persoonlijke adres. Hoe dan ook, je kunt e-mail adressen in meerdere commits ook in één klap veranderen met **filter-branch**. Je moet wel oppassen dat je alleen die e-mail adressen aanpast die van jou zijn, dus gebruik je **--commit-filter**:

```
$ git filter-branch --commit-filter '
  if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
  then
    GIT_AUTHOR_NAME="Scott Chacon";
    GIT_AUTHOR_EMAIL="schacon@example.com";
    git commit-tree "$@";
  else
    git commit-tree "$@";
  fi' HEAD
```

Dit gaat alle commits door en herschrijft ze zodat deze jouw nieuwe adres bevatten. Om dat commits de SHA-1 waarde van hun ouders bevatten, zal dit commando iedere commit SHA in jouw geschiedenis veranderen, niet alleen diegene die het gezochte e-mailadres bevatten.

## Reset ontrafeld

Voordat we doorgaan naar de meer gespecialiseerde instrumenten, laten we eerst de **reset** en **checkout** commando's bespreken. Deze commando's zijn twee van de meest verwarringe delen van Git als je ze voor het eerst tegenkomt. Ze doen zo veel dingen, dat het bijkans onmogelijk is om ze echt te begrijpen en juist toe te passen. Hiervoor stellen we een eenvoudige metafoor voor.

### De drie boomstructuren

Een eenvoudiger manier om je **reset** en **checkout** voor te stellen is door je voor te stellen dat Git een gegevensbeheerder is van drie boomstructuren. Met "boom" bedoelen we hier eigenlijk "verzameling van bestanden", en niet een bepaalde gegevensstructuur. (Er zijn een paar gevallen waarbij de index zich niet echt als een boomstructuur gedraagt, maar voor dit doeleinde is het eenvoudiger om het je als zodanig voor te stellen).

Git als systeem beheert en manipuleert deze boomstructuren bij de gewone operaties:

| Boom              | Rol                                     |
|-------------------|---|
| HEAD              | Laatste commit snapshot, volgende ouder |
| Index             | Voorgestelde volgende commit snapshot   |
| Working Directory | Speeltuin                               |

### De HEAD

HEAD is de verwijzing naar de huidige branch referentie, wat op zijn beurt een verwijzing is naar de laatste commit die gemaakt is op die branch. Dat houdt in dat HEAD de ouder zal zijn van de volgende commit die wordt gemaakt. Het is over het algemeen het eenvoudigste om HEAD te zien als de snapshot van **je laatste commit op die branch**.

Het is in feite redelijk eenvoudig om te zien hoe die snapshot eruit ziet. Hier is een voorbeeld hoe de echte directory inhoud en SHA-1 checksum voor elk bestand in de HEAD snapshot te krijgen:

```
$ git cat-file -p HEAD
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
author Scott Chacon 1301511835 -0700
committer Scott Chacon 1301511835 -0700

initial commit

$ git ls-tree -r HEAD
100644 blob a906cb2a4a904a152... README
100644 blob 8f94139338f9404f2... Rakefile
040000 tree 99f1a6d12cb4b6f19... lib
```

De `cat-file` en `ls-tree` commando's zijn "binnenwerk" (plumbing) commando's die gebruikt worden door de lagere functies en niet echt gebruikt worden in dagelijkse toepassingen, maar ze helpen ons om te zien wat er eigenlijk gebeurt.

## De index

De index is je **voorstel voor de volgende commit**. We hebben hieraan ook gerefereerd als de "staging area" van Git, omdat dit is waar Git naar kijkt als je `git commit` aanroept.

Git vult deze index met een lijst van de inhoud van alle bestanden die als laatste waren uitgechecked naar je werk directory en hoe ze eruit zagen toen ze oorspronkelijk waren uitgechecked. Je vervangt enkele van deze bestanden met nieuwe versies ervan, en `git commit` converteert dit dan naar de boomstructuur voor een nieuwe commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README
100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb
```

Hier gebruiken we nogmaals `ls-files`, wat meer een achter-de-schermen commando is dat je laat zien hoe je index er op dit moment uitziet.

Technisch gesproken is de index geen boomstructuur — het wordt eigenlijk geïmplementeerd als een geplette manifest — maar voor dit doeleinde is het goed genoeg.

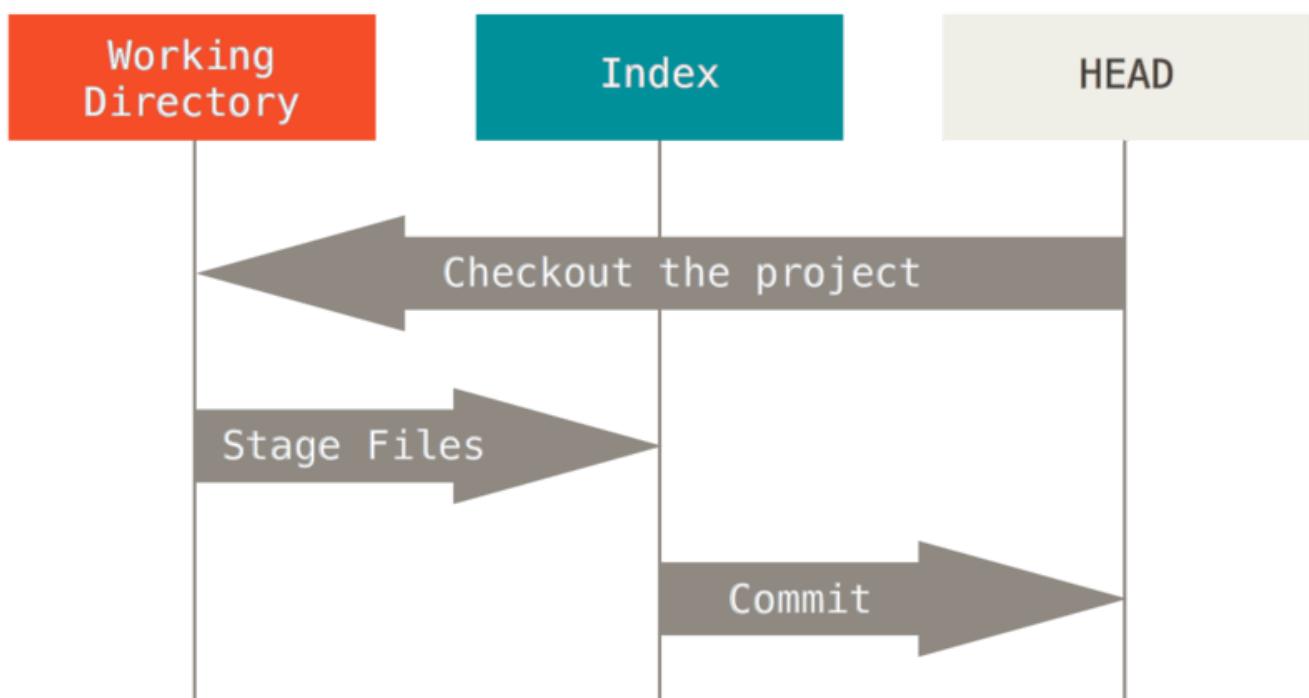
## De werk directory

En als laatste is er je werk directory. De andere twee boomstructuren slaan hun inhoud op een efficient maar onhandige manier op, in de `.git` directory. De werk directory pakt ze uit in echte bestanden, wat het makkelijker voor je maakt om ze te bewerken. Zie de werk directory als een **speeltuin**, waar je wijzigingen kunt uitproberen voordat je ze naar je staging area (index) commit en daarna naar de historie.

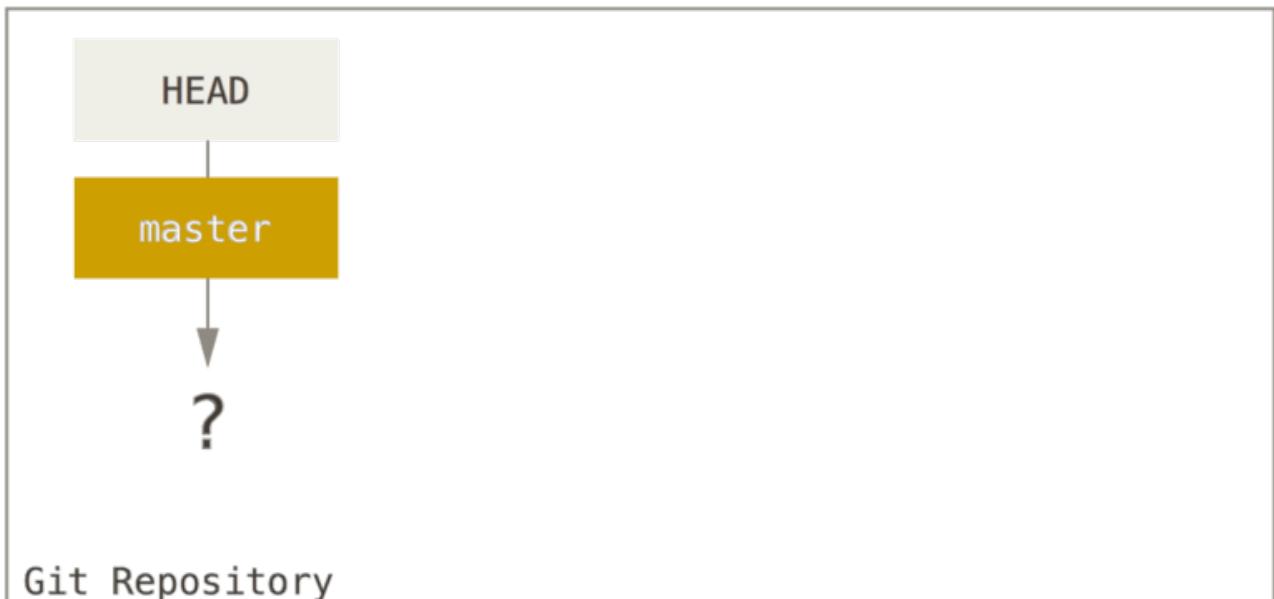
```
$ tree  
.  
├── README  
├── Rakefile  
└── lib  
    └── simplegit.rb  
  
1 directory, 3 files
```

## De Workflow

Het voornaamste doel van Git is om opeenvolgende snapshots te op te slaan van verbeteringen aan je project, door deze drie bomen te manipuleren.



Laten we dit proces eens visualiseren: stel je gaat een nieuwe directory in waarin een enkel bestand staat. We noemen dit **v1** van het bestand, en we geven het in blauw weer. Nu roepen we **git init** aan, wat een Git repository aanmaakt met een HEAD referentie die verwijst naar een ongeboren branch (**master** bestaat nog niet).

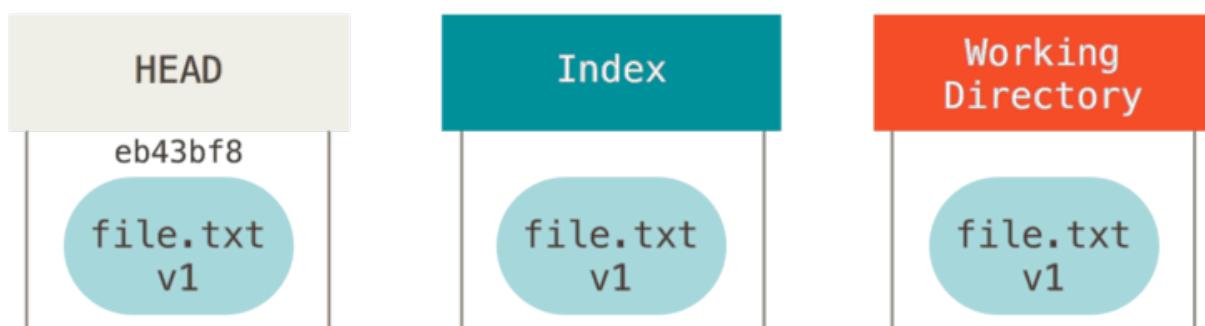
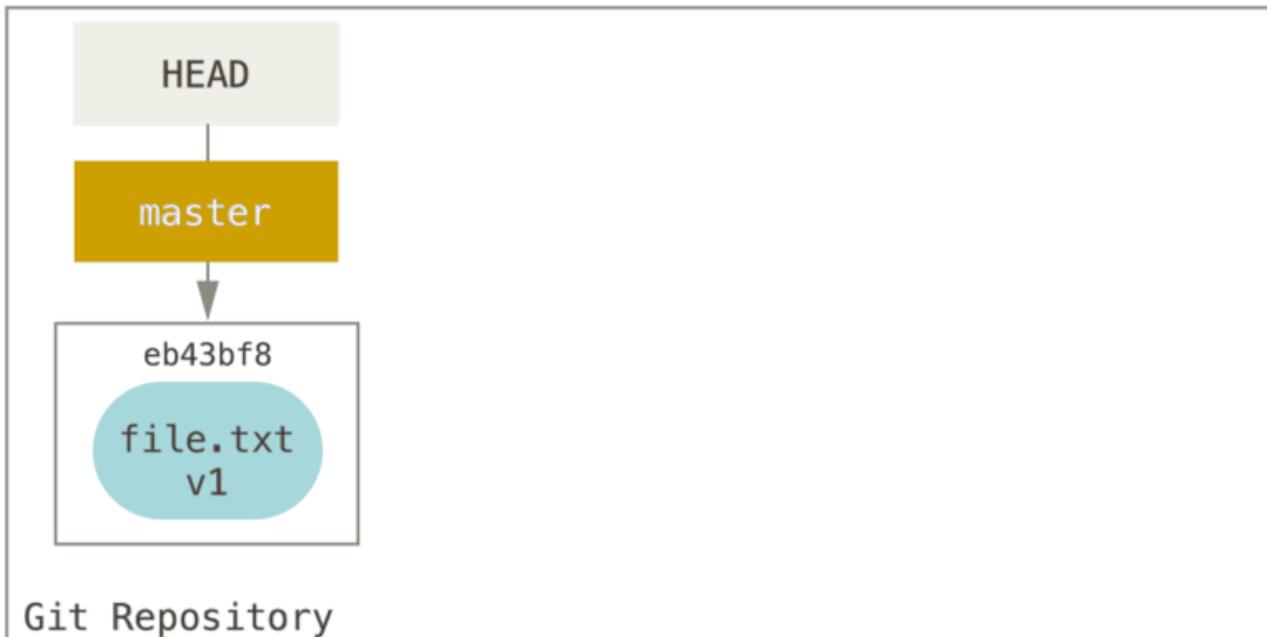


Op dit moment heeft alleen de boom van de werk directory inhoud.

Nu willen we dit bestand committen, dus we gebruiken `git add` om de inhoud van de werk directory te pakken en dit in de Index te kopiëren.



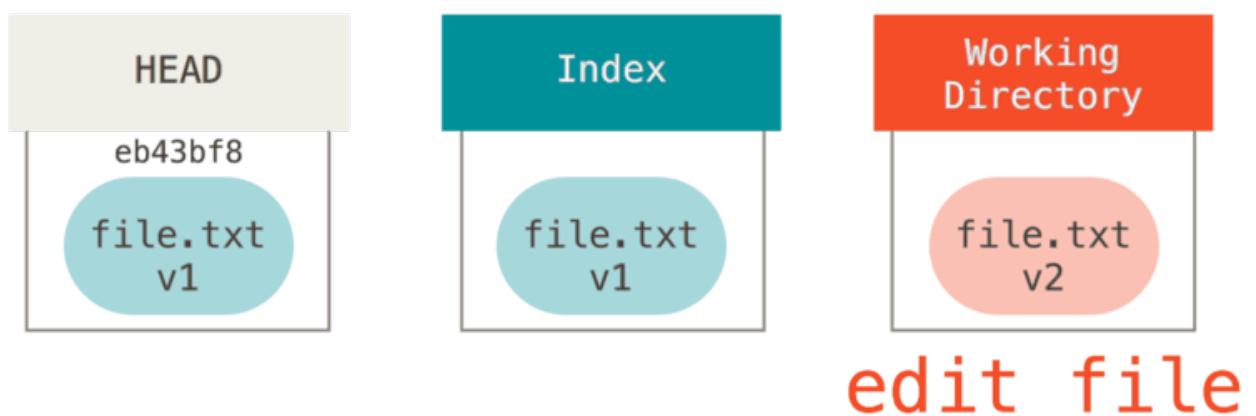
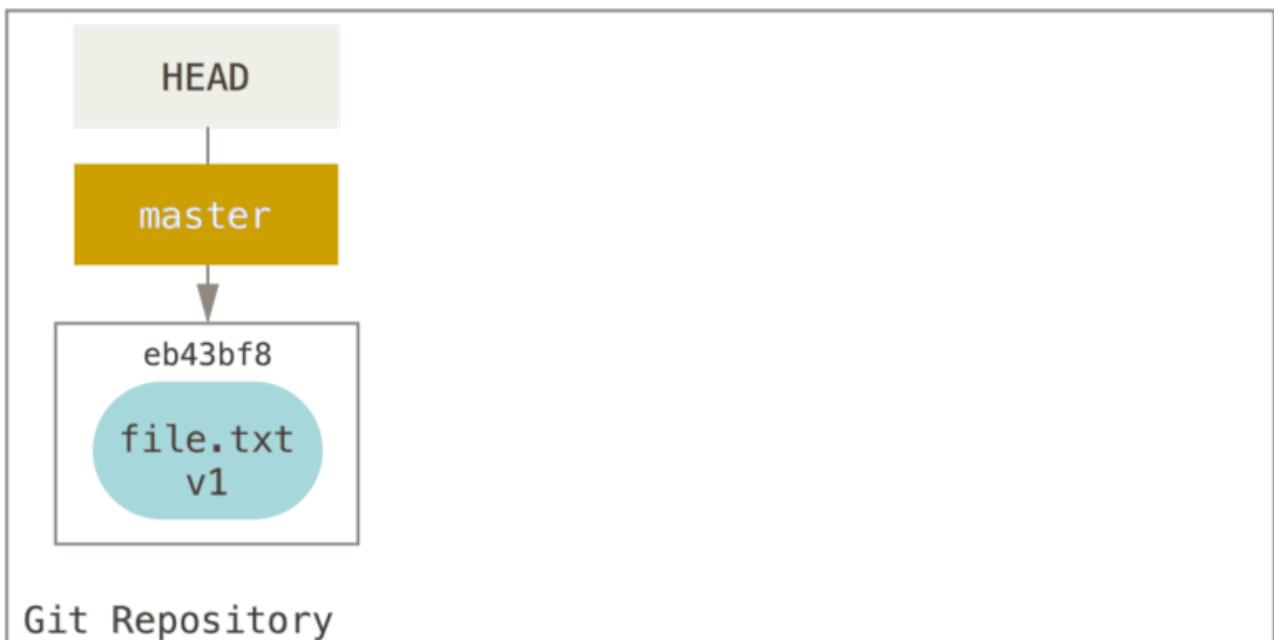
Dan roepen we `git commit` aan, wat de inhoud van de Index pakt en deze bewaart als een permanente snapshot, een commit object aanmaakt die naar die snapshot wijst en dan `master` update die naar die commit wijst.



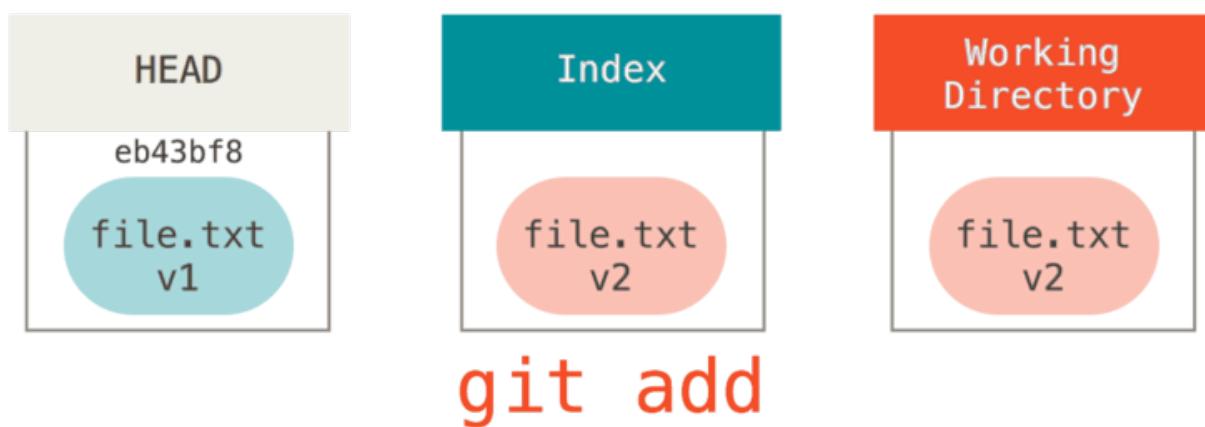
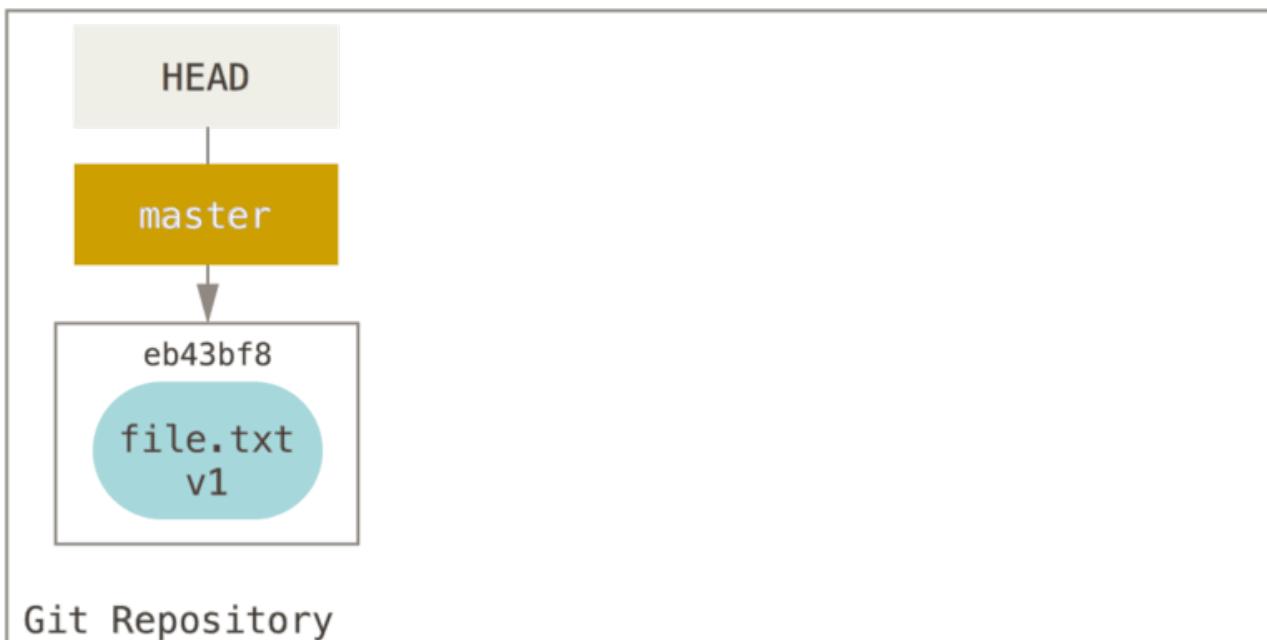
## git commit

Als we nu `git status` aanroepen zien we geen wijzigingen, omdat alle drie bomen hetzelfde zijn.

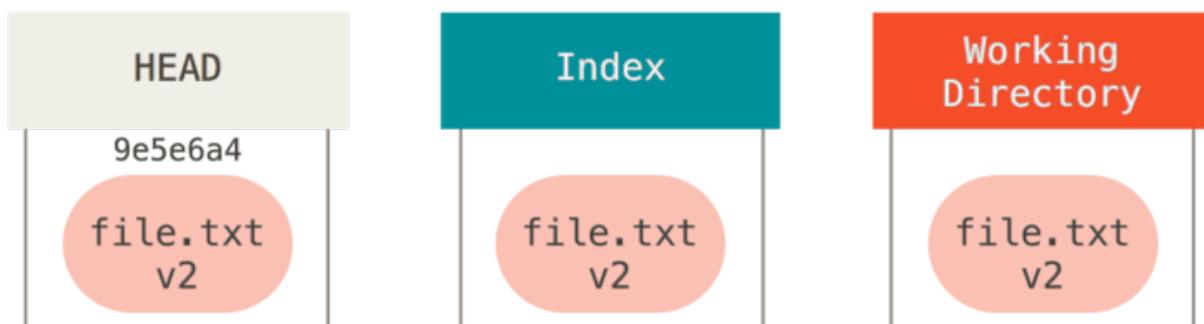
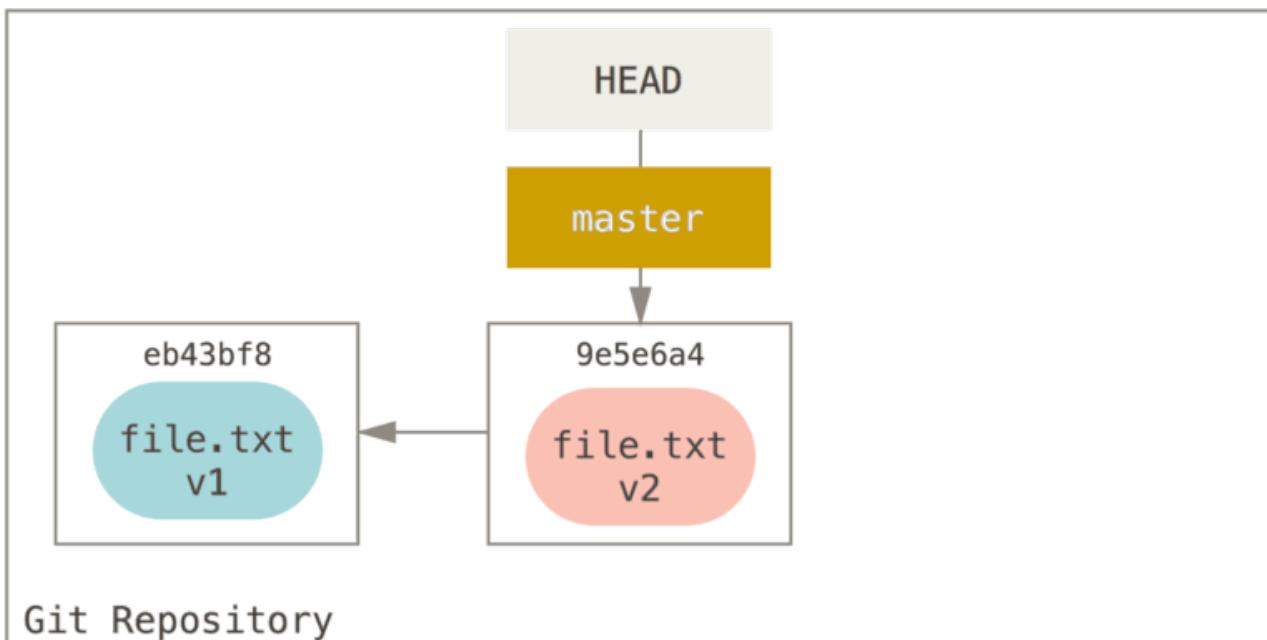
Nu willen we dat bestand wijzigen en deze committen. We volgen hetzelfde proces; eerst wijzigen we het bestand in onze werk directory. Laten we deze **v2** van het bestand noemen, en deze in rood weergeven.



Als we nu `git status` aanroepen, zullen we het bestand in het rood zien als “Changes not staged for commit,” omdat deze versie van het bestand verschilt tussen de index en de werk directory. Nu gaan we `git add` aanroepen om het in onze index te staggen (“to stage”: klaarzetten).



Als we op dit moment `git status` aanroepen zullen we het bestand in het groen zien onder “Changes to be committed” omdat de Index en HEAD verschillen—dat wil zeggen, onze voorgestelde volgende commit verschilt nu van onze laatste commit. Tot slot roepen we `git commit` aan om de commit af te ronden.



## git commit

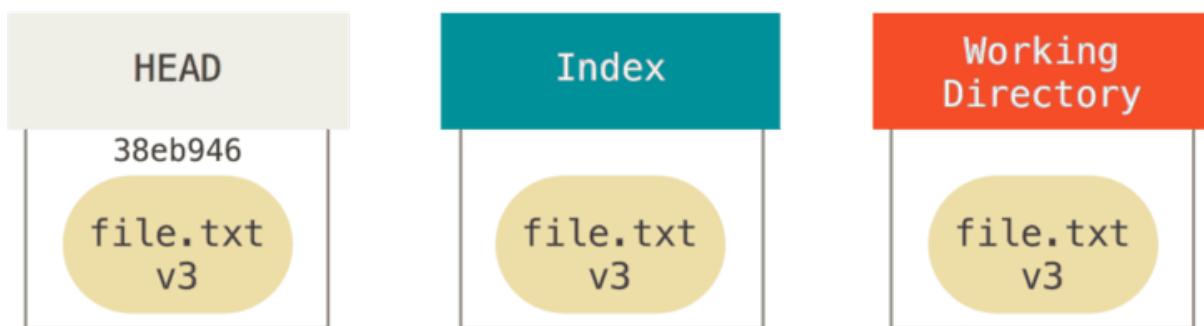
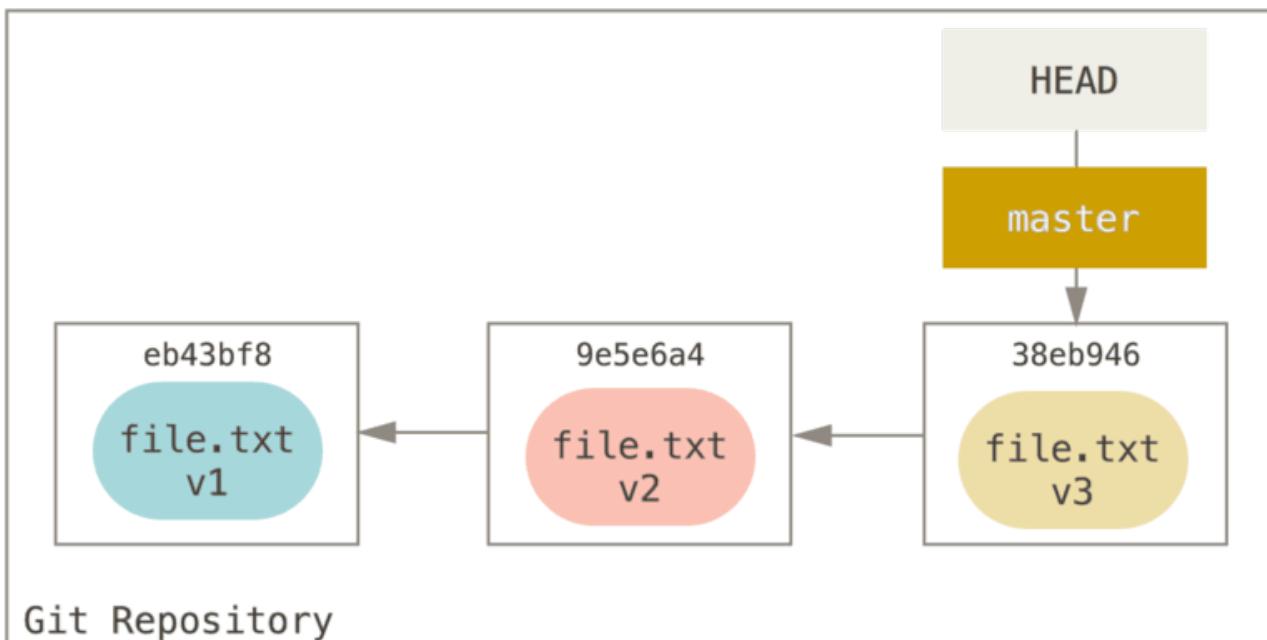
Nu zal `git status` geen uitvoer laten zien, omdat alle drie bomen weer hetzelfde zijn.

Tussen branches overschakelen of klonen volgen een vergelijkbaar proces. Als je een branch uitcheckt, wijzigt dit **HEAD** door het te laten wijzen naar de nieuwe branch referentie, het vult je **Index** met de snapshot van die commit, en kopieert dan de inhoud van de **Index** naar je **werk directory**.

## De rol van reset

Het `reset` commando krijgt in dit licht meer betekenis.

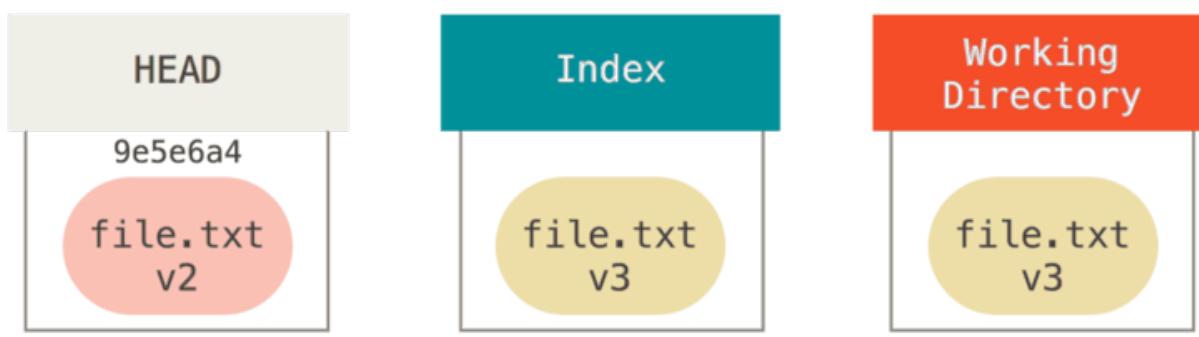
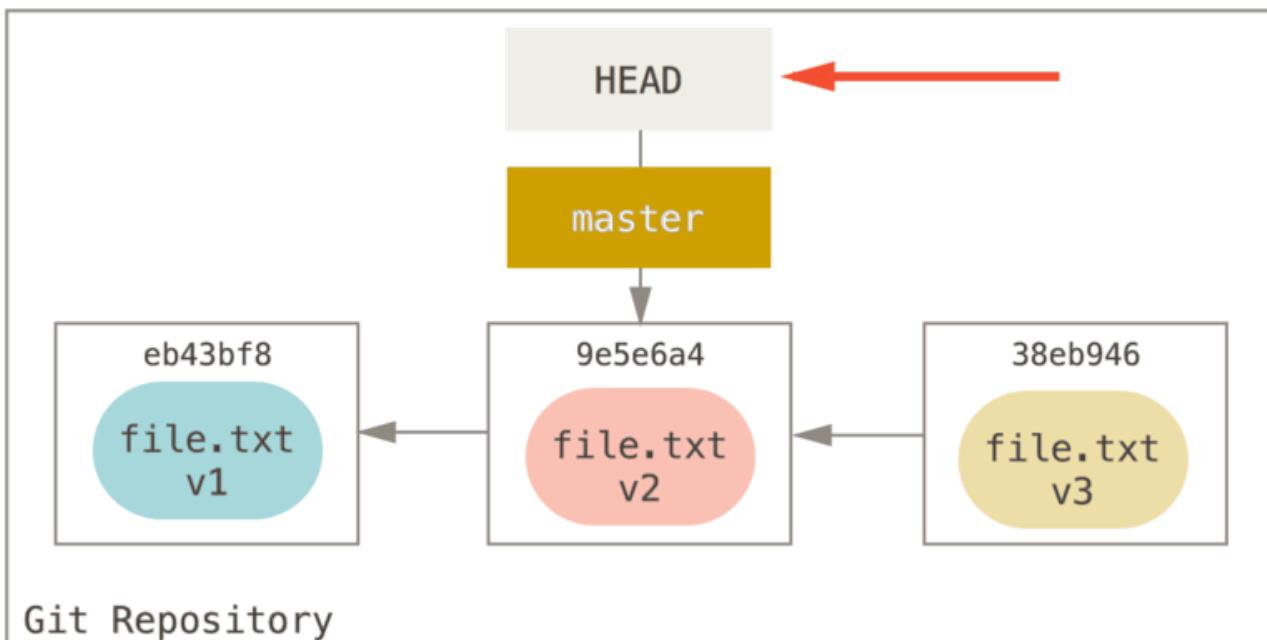
Laten we, voor het doel van deze voorbeelden, stellen dat we `file.txt` weer gewijzigd hebben en het voor de derde keer gecommit. Nu ziet onze historie er dus als volgt uit:



Laten we nu een stap voor stap bespreken wat `reset` doet als je het aanroeft. Het manipuleert deze drie bomen op een eenvoudige en voorspelbare manier. Het voert tot drie basale handelingen uit.

### Stap 1: Verplaats HEAD

Het eerste wat `reset` zal doen is hetgeen waar HEAD naar verwijst verplaatsen. Dit is niet hetzelfde als HEAD zelf wijzigen (dat is wat `checkout` doet); `reset` verplaats de branch waar HEAD naar verwijst. Dit houdt in dat als HEAD naar de `master`-branch wijst (d.i. je bent nu op de `master`-branch), het aanroepen van `git reset 9e5e6a4` zal beginnen met `master` naar `9e5e6a4` te laten wijzen.



**git reset --soft HEAD~**

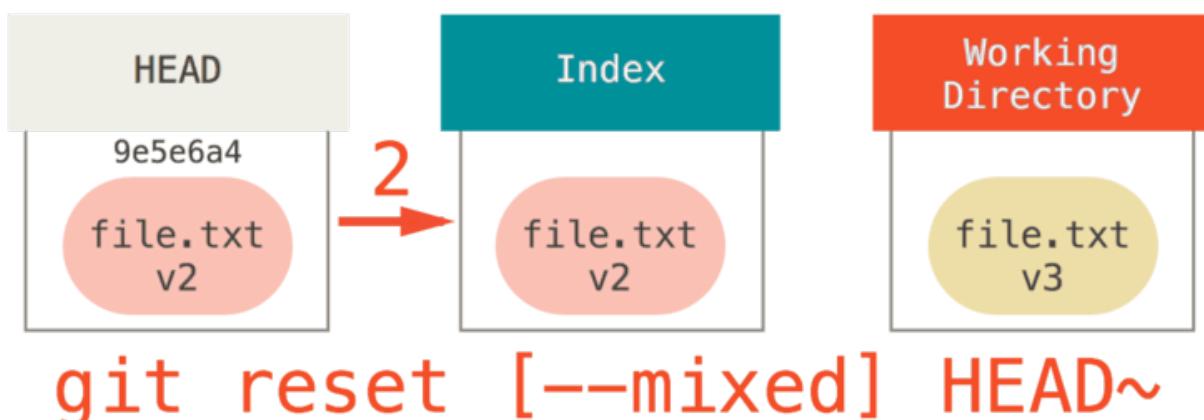
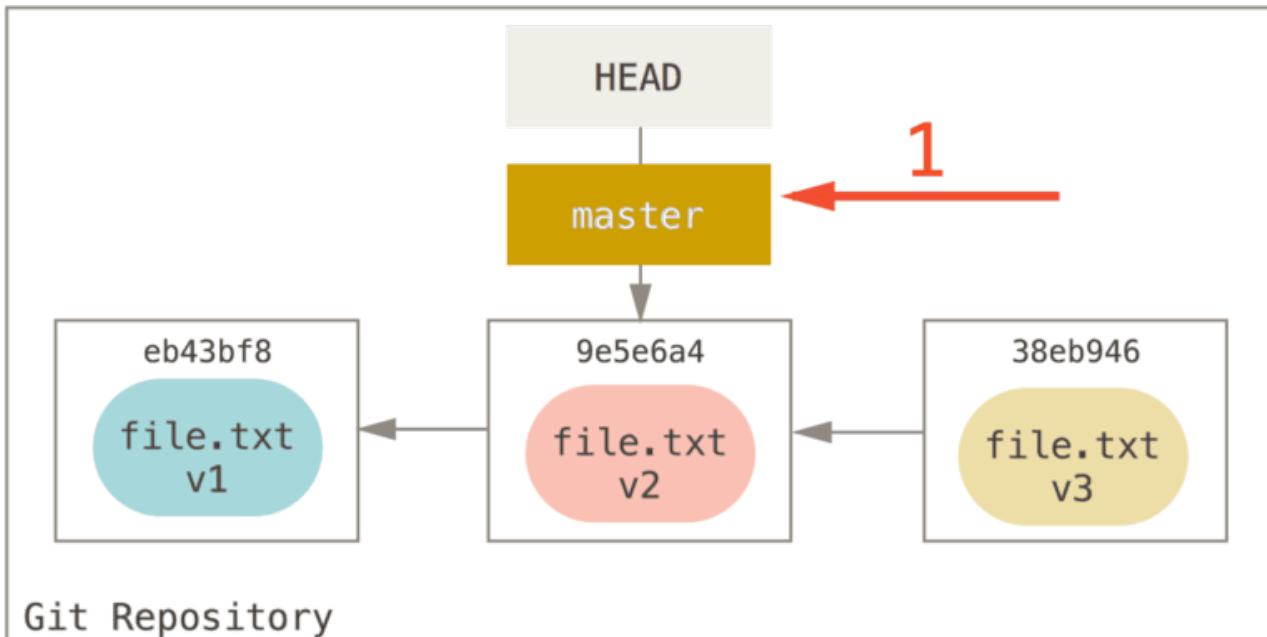
Het maakt niet uit welke variant van `reset` met een commit je aanroeft, dit is het eerste wat het altijd zal proberen te doen. Met `reset --soft`, zal het eenvoudigweg daar stoppen.

Kijk nu nog een keer naar het diagram en besef wat er gebeurd is: het heeft feitelijk de laatste `git commit` comando ongedaan gemaakt. Als je `git commit` aanroeft, maakt Git een nieuwe commit en verplaatst de branch waar HEAD naar wijst daarnaar toe. Als je naar `HEAD~` (de ouder van HEAD) terug `reset`, verplaats je de branch terug naar waar het was, zonder de Index of werk directory te wijzigen. Je kunt de Index nu bijwerken en `git commit` nogmaals aanroepen om te bereiken wat `git commit --amend` gedaan zou hebben (zie [De laatste commit veranderen](#)).

### Stap 2: De Index bijwerken (--mixed)

Merk op dat als je `git status` nu aanroeft dat je het verschil tussen de Index en wat de nieuwe HEAD is in het groen ziet.

Het volgende wat `reset` zal gaan doen is de Index bijwerken met de inhoud van de snapshot waar HEAD nu naar wijst.

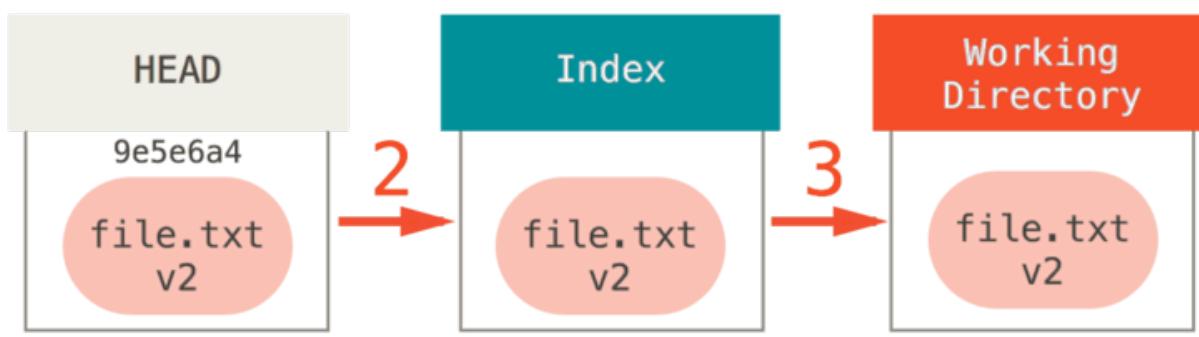
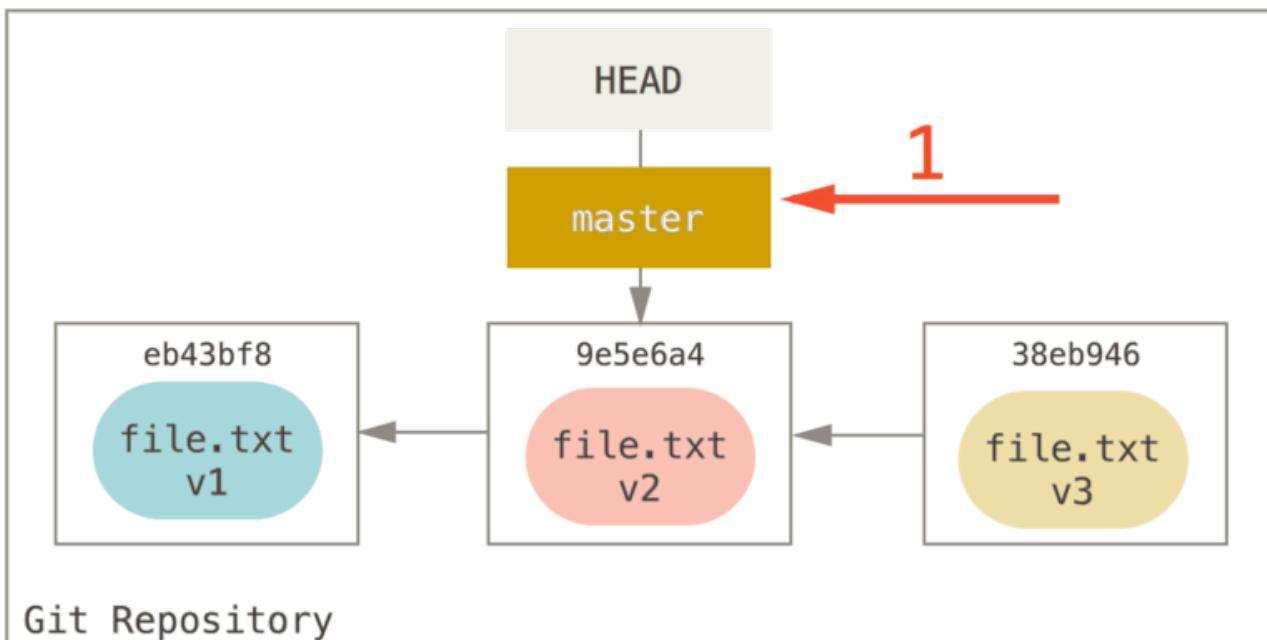


Als je de `--mixed` optie hebt opgegeven, zal `reset` op dit punt stoppen. Dit is ook het standaard gedrag, dus als je geen enkele optie hebt opgegeven (dus in dit geval alleen `git reset HEAD~`), is dit waar het commando zal stoppen.

Kijk nu nog een keer naar het diagram en besef wat er gebeurd is: het heeft nog steeds je laatste `commit` ongedaan gemaakt, maar nu ook alles *unstaged*.

### Stap 3: De working directory bijwerken (-hard)

Het derde wat `reset` zal doen is ervoor zorgen dat de werk directory gaat lijken op de Index. Als je de `--hard` optie gebruikt, zal het doorgaan naar dit stadium.



Laten we eens overdenken wat er zojuist is gebeurd. Je hebt je laatste commit ongedaan gemaakt, de `git add` en `git commit` commando's, **en** al het werk wat je in je werk directory gedaan hebt.

Het is belangrijk op te merken dat deze vlag (`--hard`) de enige manier is om het `reset` commando gevaarlijk te maken, en een van de weinige gevallen waar Git daadwerkelijk gegevens zal vernietigen. Elke andere aanroep van `reset` kan redelijk eenvoudig worden teruggedraaid, maar de `--hard` optie kan dat niet, omdat het keihard de bestanden in de werk directory overschrijft. In dit specifieke geval, hebben we nog steeds de **v3** versie van ons bestand in een commit in onze Git database, en we zouden het kunnen terughalen door naar onze `reflog` te kijken, maar als we het niet zouden hebben gecommit, zou Git het bestand nog steeds hebben overschreven en zou het niet meer te herstellen zijn.

## Samenvattend

Het `reset` commando overschrijft deze drie bomen in een vastgestelde volgorde, en stopt waar je het toe opdraagt:

1. Verplaats de branch waar HEAD naar wijst (*stop hier als `--soft`*)
2. Laat de Index eruit zien als HEAD (*stop hier tenzij `--hard`*)

### 3. Laat de werk directory eruit zien als de Index

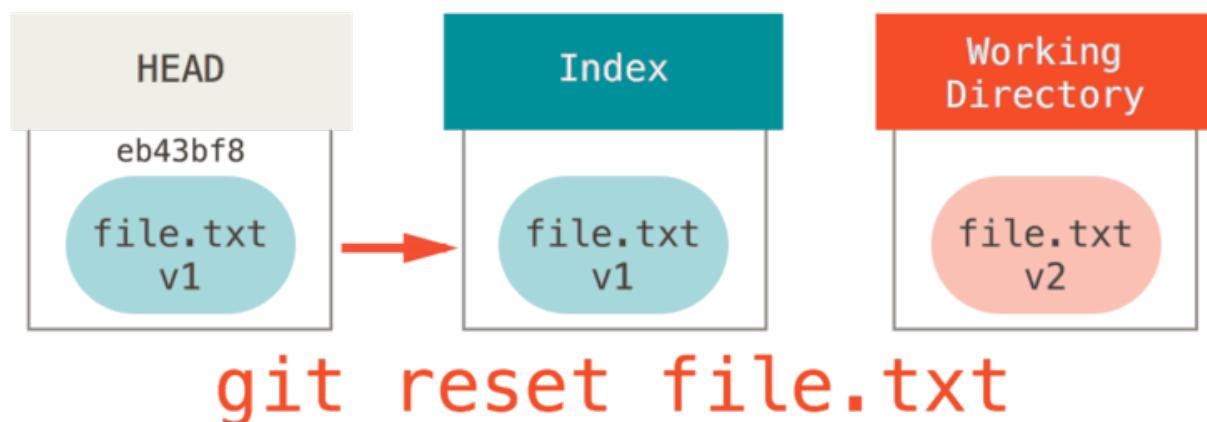
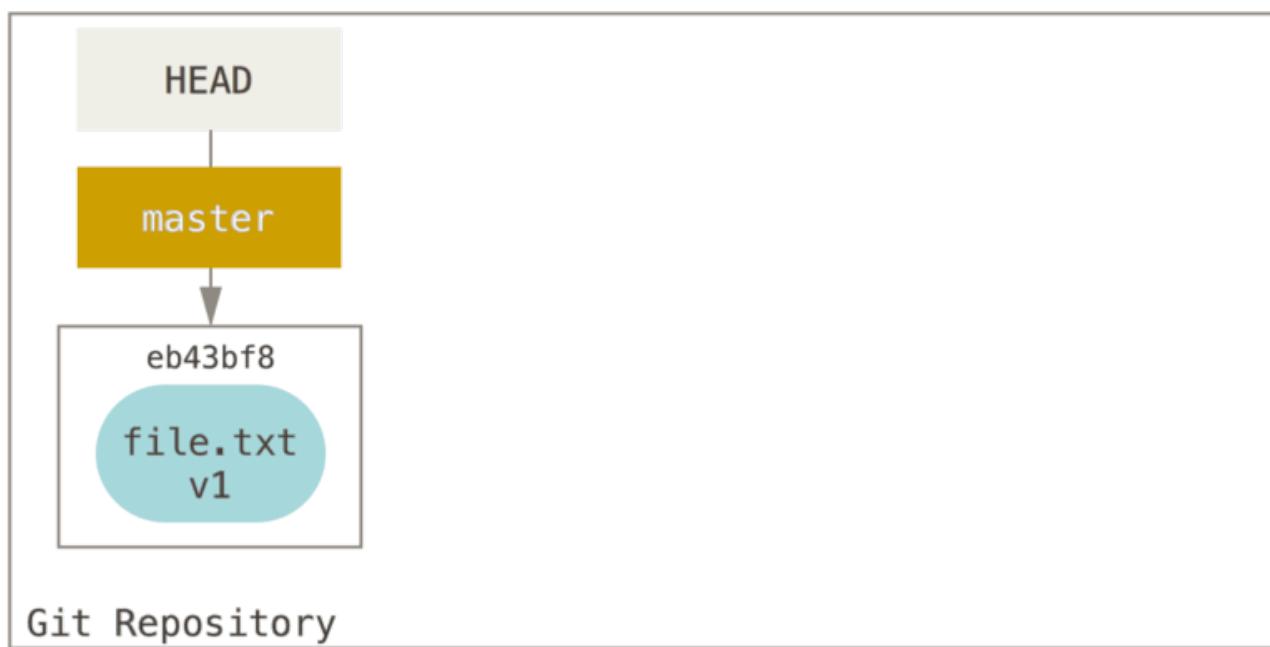
## Reset met een pad (path)

Dit dekt het gedrag van `reset` in zijn eenvoudige vorm, maar je kunt er ook een path bij opgeven waar het op moet acteren. Als je een path opgeeft, zal `reset` stap 1 overslaan, en de rest van de acties beperken tot een specifiek bestand of groep van bestanden. Dit is ergens wel logisch — HEAD is maar een verwijzing, en je kunt niet naar een deel van een commit wijzen en deels naar een andere. Maar de Index en werk directory *kunnen* deels worden bijgewerkt, dus `reset` gaat verder met stappen 2 en 3.

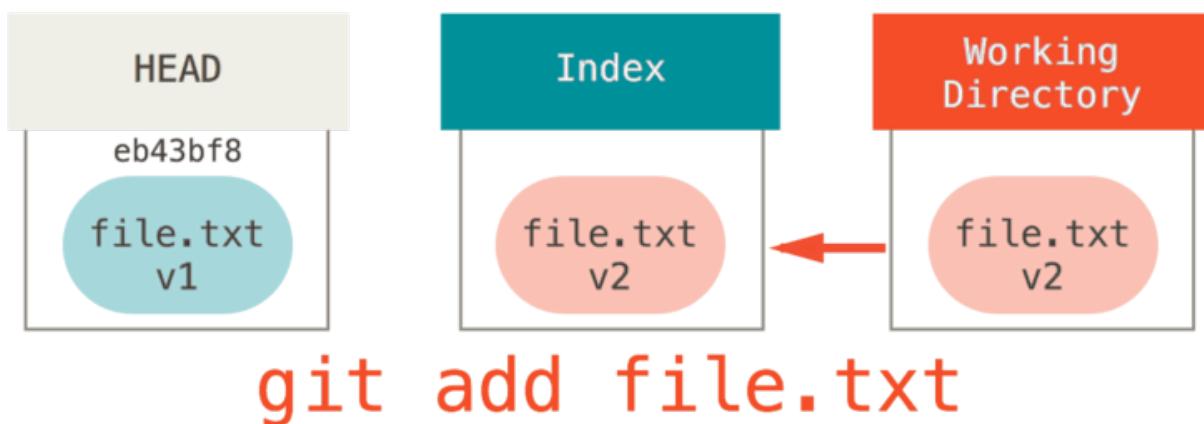
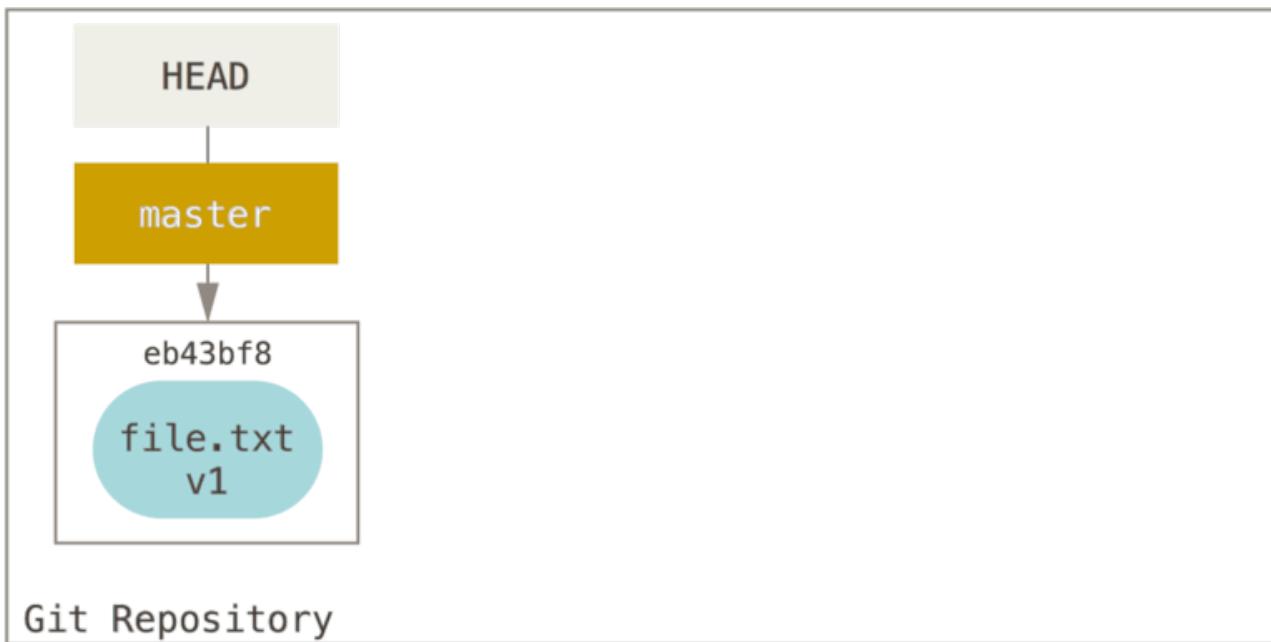
Dus, laten we aannemen dat we `git reset file.txt` aanroepen. Deze vorm (omdat je niet een specifieke SHA-1 van een commit of branch meegeeft, en je hebt geen `--soft` of `--hard` meegegeven) is dit een verkorte vorm van `git reset --mixed HEAD file.txt` en dit zal:

1. De branch waar HEAD naar wijst verplaatsen (*overgeslagen*)
2. De Index eruit zien als HEAD (*stop hier*)

Dus effectief wordt alleen `file.txt` van HEAD naar de Index gekopieerd.

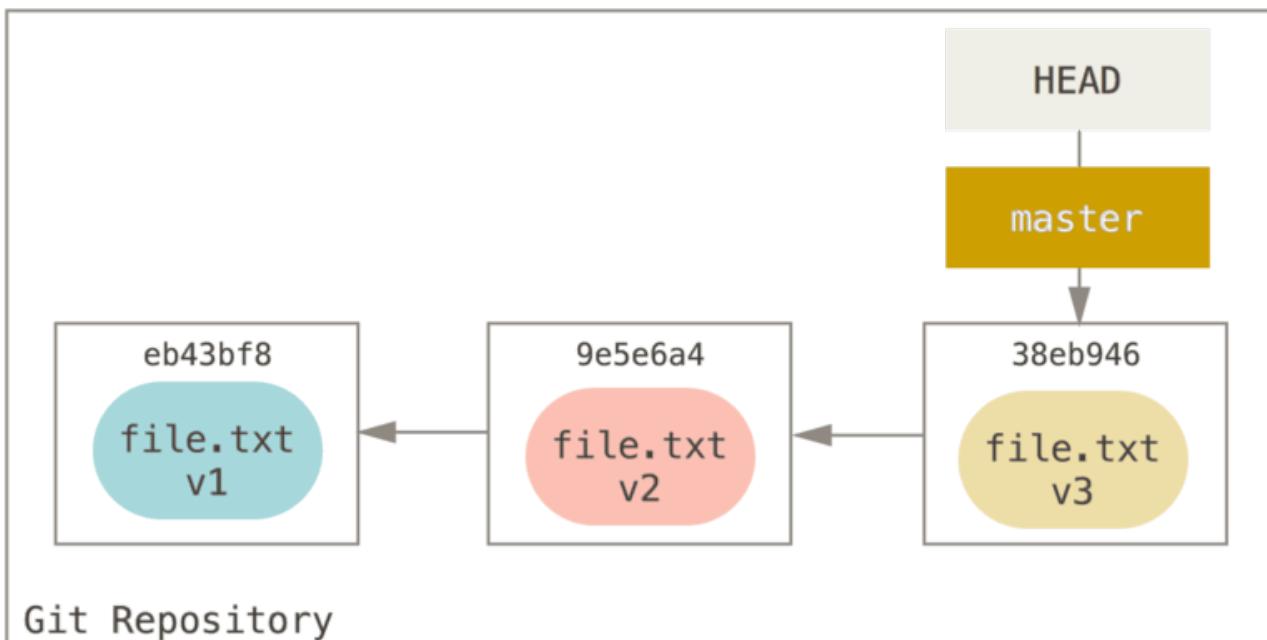


Dit heeft het praktische effect van het bestand *unstage*. Als we kijken naar het diagram voor dat commando en denken aan wat `git add` doet, zijn ze exact elkaars tegenpolen.



Dit is de reden waarom de uitvoer van het `git status` commando je aanraadt om dit aan te roepen om een bestand te unstagen. (Zie [Een gestaged bestand unstagen](#) voor meer hierover.)

We hadden net zo makkelijk Git niet laten aannemen dat we “pull de data van HEAD” bedoelen door een specifieke commit op te geven om die versie van het bestand te pullen. We hadden ook iets als `git reset eb43bf file.txt` kunnen aanroepen.



**git reset eb43 -- file.txt**

Feitelijk gebeurt hier hetzelfde als wanneer we de inhoud van het bestand naar **v1** in de werk directory hadden teruggedraaid, `git add` ervoor hadden aangeroepen, en daarna het weer hadden teruggedraaid naar **v3** (zonder daadwerkelijk al deze stappen te hebben gevuld). Als we nu `git commit` aanroepen, zal het een wijziging vastleggen die het bestand naar **v1** terugdraait, ook al hebben we het nooit echt weer in onze werk directory gehad.

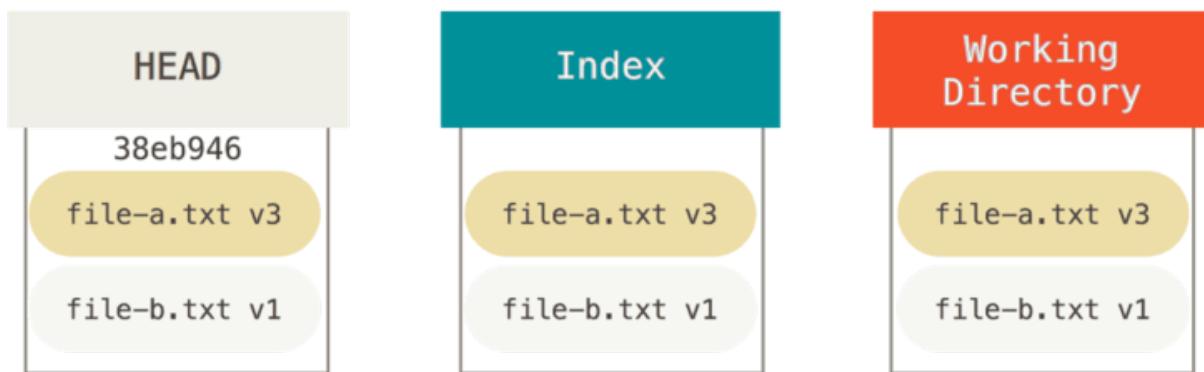
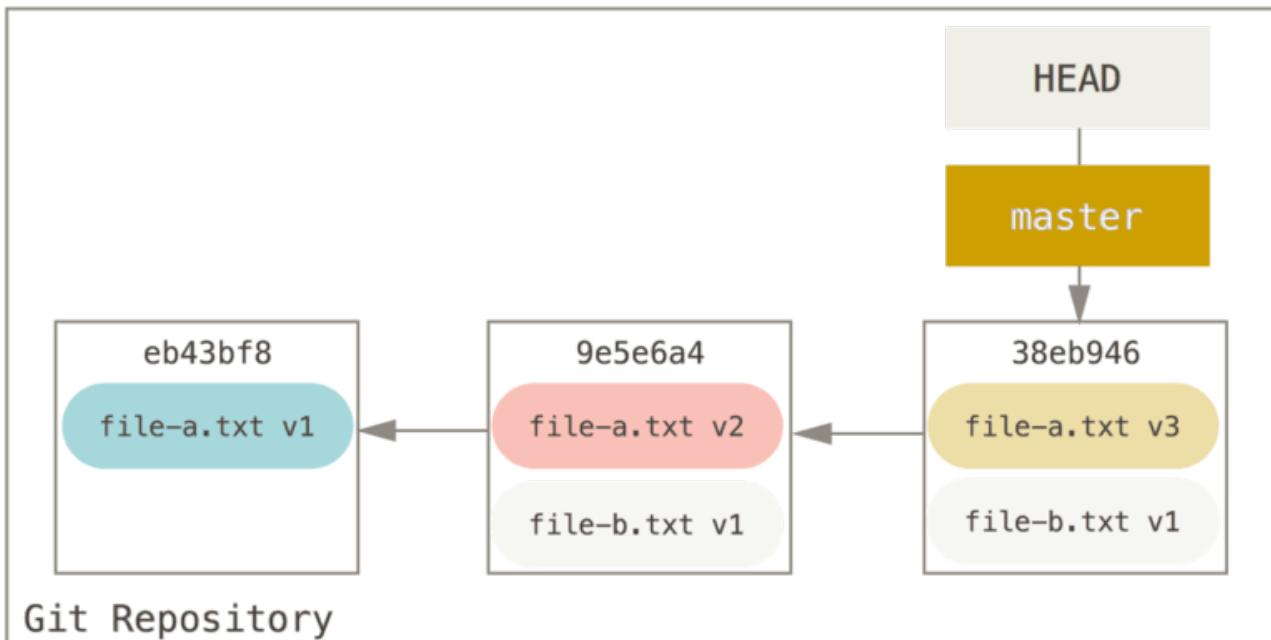
Het is ook interessant om op te merken dat net als `git add`, het `reset` commando een `--patch` optie accepteert om inhoud in deelsgewijs te unstagen. Dus je kunt naar keuze inhoud unstagen of terugdraaien (revert).

## Samenpersen (Squashing)

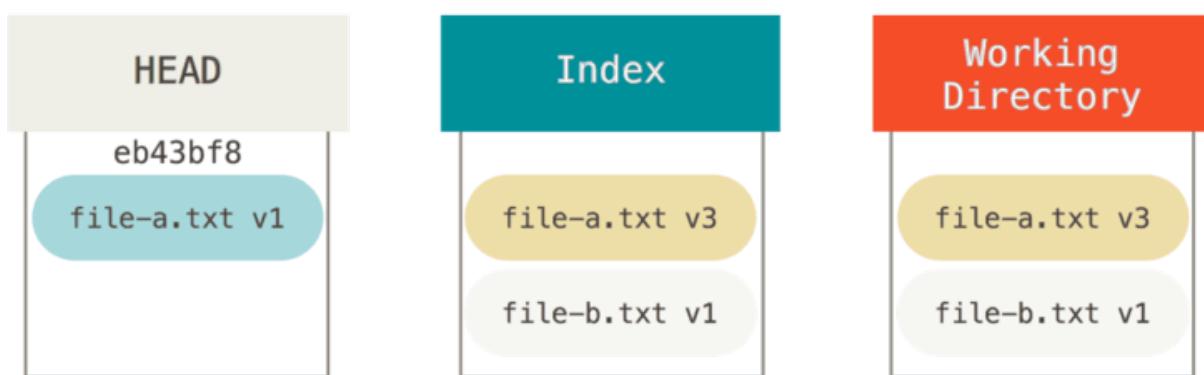
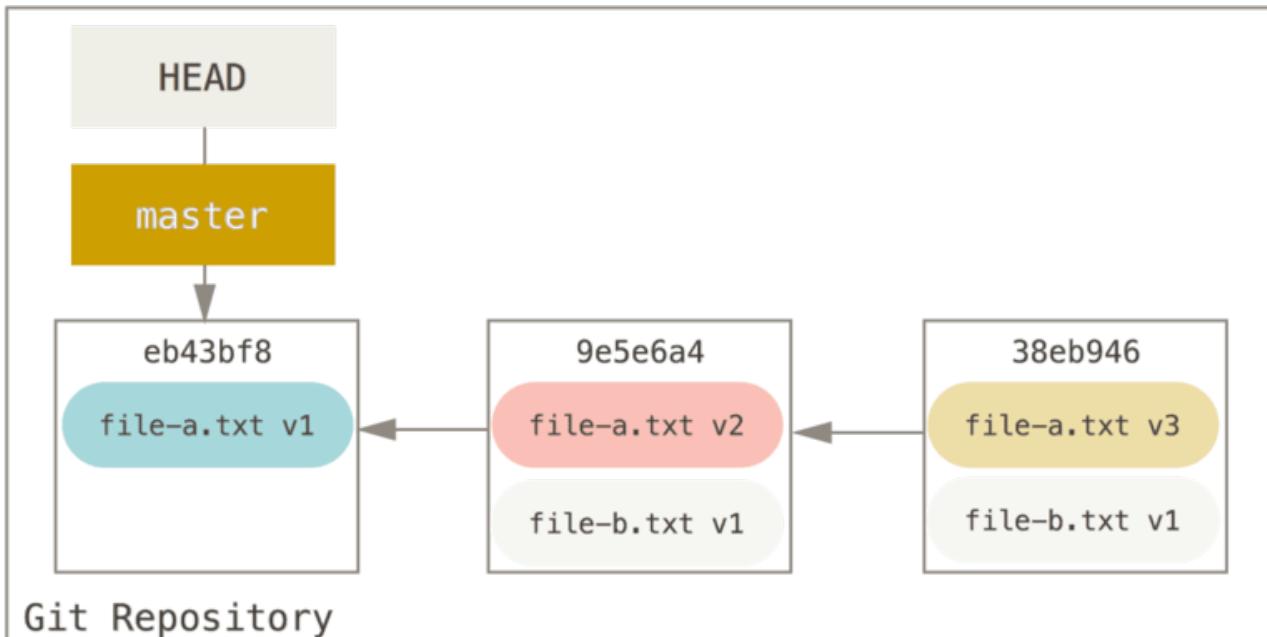
Laten we nu kijken hoe we iets interessants kunnen doen met deze vers ontdekte krachten — commits samenpersen (squashen).

Stel dat je een reeks van commits met berichten als “oops.”, “WIP” en “dit bestand vergeten”. Je kunt `reset` gebruiken om deze snel en makkelijk in een enkele commit te samenpersen waardoor je ontzettend slim zult lijken. ([Een commit samenpersen \(squashing\)](#) laat je een andere manier zien om dit te doen, maar in dit voorbeeld is het makkelijker om `reset` te gebruiken.)

Stel dat je een project hebt waar de eerste commit een bestand heeft, de tweede commit een nieuw bestand toevoegde en het eerste wijzigde, en de derde commit het eerste bestand weer wijzigde. De tweede commit was een onderhanden werk en je wilt het samenpersen.

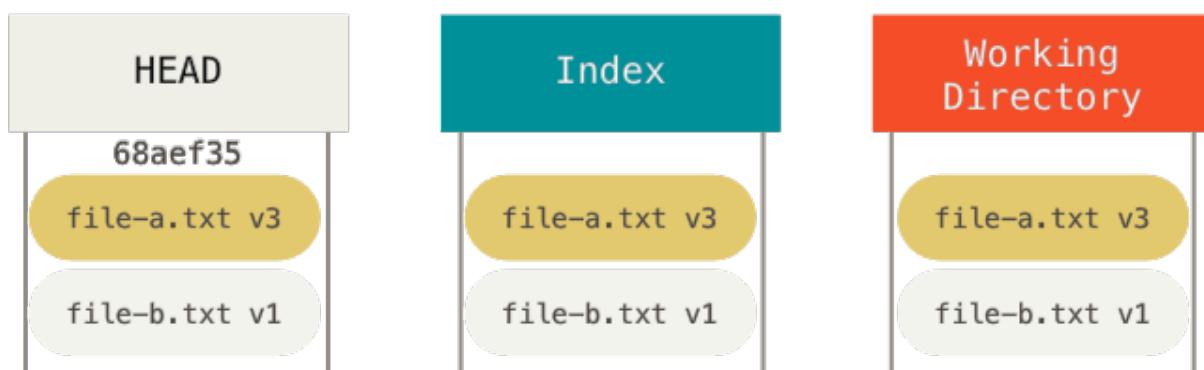
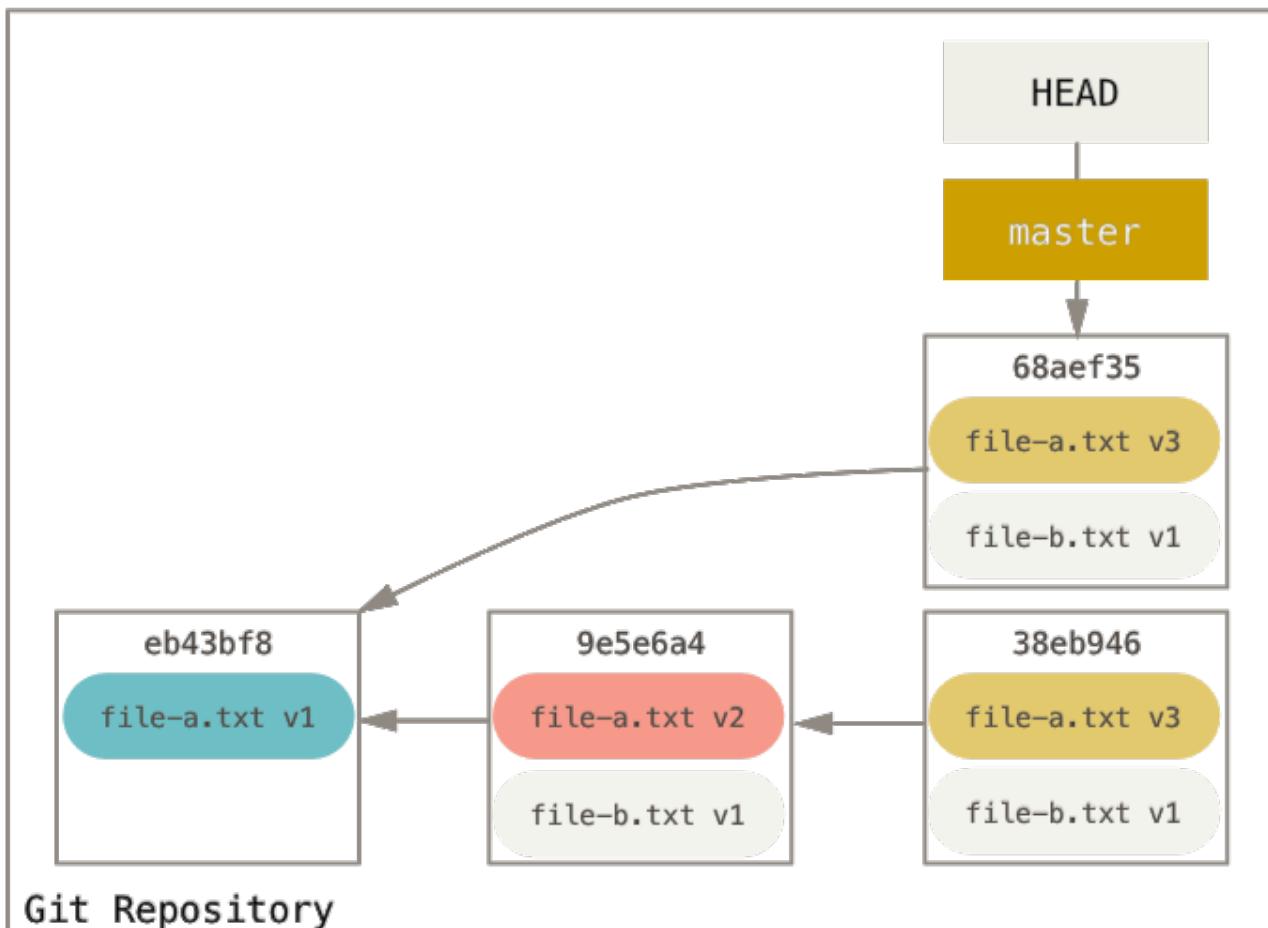


Je kan `git reset --soft HEAD~2` uitvoeren om de HEAD branch terug naar een oudere commit te verplaatsen (de eerste commit wil je behouden):



**git reset --soft HEAD~2**

En daarna eenvoudigweg `git commit` weer aanroepen:



## git commit

Je kunt nu zien dat je bereikbare historie, de historie die je zou gaan pushen, nu eruit ziet alsof je een commit had met **file-a.txt v1**, dan een tweede die zowel **file-a.txt** naar v3 wijzigt en **file-b.txt** toevoegt. De commit met de v2 versie van het bestand is niet meer in de historie aanwezig.

## Check It Out

Als laatste, je kunt je afvragen wat het verschil is tussen **checkout** en **reset**. Net als **reset**, bewerkt **checkout** de drie bomen, en het verschilt enigszins afhankelijk van of je het commando een bestandsnaam geeft of niet.

## Zonder paths

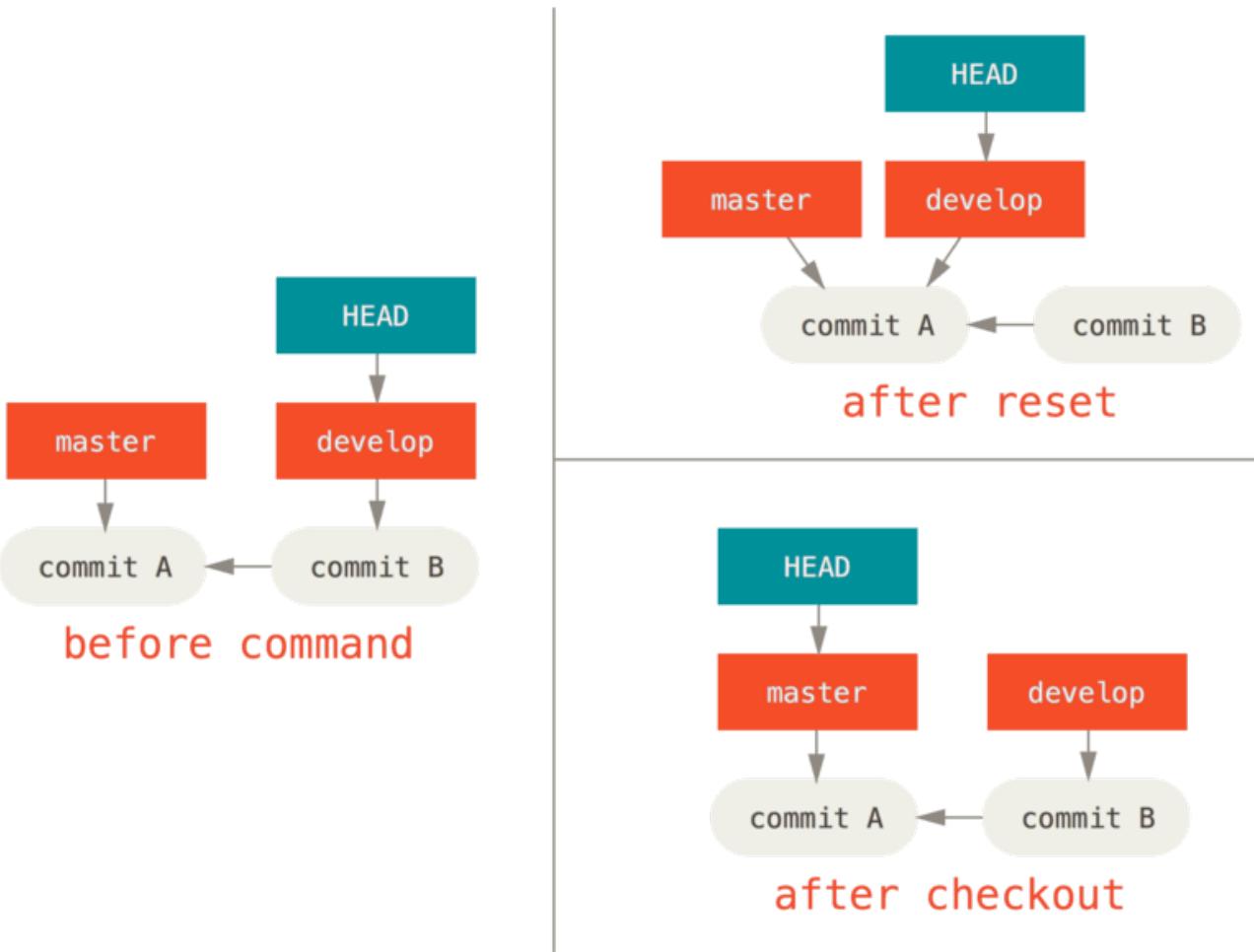
Het aanroepen van `git checkout [branch]` is vergelijkbaar met het aanroepen van `git reset --hard [branch]` in die zin dat het alle drie bomen voor je laat uitzien als `[branch]`, maar er zijn twee belangrijke verschillen.

Ten eerste, in tegenstelling tot `reset --hard`, is `checkout` veilig voor de werk-directory; het zal controleren dat het geen bestanden weggooit waar wijzigingen in gemaakt zijn. Eigenlijk is het nog iets slimmer dan dat—het probeert een triviale merge in de werk directory te doen, zodat alle bestanden die je *niet* gewijzigd hebt bijgewerkt worden. Aan de ander kant zal `reset --hard` eenvoudigweg alles zonder controleren vervangen.

Het tweede belangrijke verschil is hoe het HEAD update. Waar `reset` de branch waar HEAD naar verwijst zal verplaatsen, zal `checkout` de HEAD zelf verplaatsen om naar een andere branch te wijzen.

Bijvoorbeeld, stel dat we `master` en `develop`-branches hebben die naar verschillende commits wijzen, en we staan op dit moment op `develop` (dus HEAD wijst daar naar). Als we `git reset master` aanroepen, zal `develop` zelf wijzen naar dezelfde commit waar `master` naar wijst. Als we echter `git checkout master` aanroepen, zal `develop` niet verplaatsen, HEAD wordt zelf verplaatst. HEAD zal nu naar `master` wijzen.

Dus in beide gevallen verplaatsen we HEAD om naar commit A te wijzen, maar *hoe* we dit doen verschilt enorm. `reset` zal de branch waar HEAD naar verwijst verplaatsen, `checkout` verplaatst HEAD zelf.



## Met paths

De andere manier om `checkout` aan te roepen is met een bestands path die, zoals `reset`, HEAD niet verplaatst. Het is precies als `git reset [branch] file` in die zin dat het de index update met dat bestand op die commit, maar het overschrijft ook het bestand in de werk directory. Het zou precies zijn als `git reset --hard [branch] file` (als `reset` je dat zou toestaan) - het is niet veilig voor de werk directory, en het verplaatst HEAD niet.

En, zoals `git reset` en `git add`, accepteert `checkout` een `--patch` optie zodat je selectief stukje bij beetje bestandsinhoud kunt terugdraaien.

## Samenvatting

Hopelijk begrijp je nu het `reset` commando en voel je je er meer mee op je gemak, maar je zult waarschijnlijk nog een beetje in verwarring zijn in hoe het precies verschilt van `checkout` en zul je je waarschijnlijk ook niet alle regels van verschillende aanroepen herinneren.

Hier is een spiekbrief voor welke commando's welke bomen beïnvloeden. In de "HEAD" kolom staat "REF" als dat commando de referentie (branch) waar HEAD naar wijst verplaatst, en "HEAD" als het HEAD zelf verplaatst. Let met name op de *WD Safe?* kolom - als daar NO in staat, bedenk je een tweede keer voordat je dat commando gebruikt.

|  | HEAD | Index | Workdir | WD Safe?  |
|--|------|-------|---------|-----------|
| <b>Commit Level</b>                          |      |       |         |           |
| <code>reset --soft [commit]</code>           | REF  | NO    | NO      | YES       |
| <code>reset [commit]</code>                  | REF  | YES   | NO      | YES       |
| <code>reset --hard [commit]</code>           | REF  | YES   | YES     | <b>NO</b> |
| <code>checkout &lt;commit&gt;</code>         | HEAD | YES   | YES     | YES       |
| <b>File Level</b>                            |      |       |         |           |
| <code>reset [commit] &lt;paths&gt;</code>    | NO   | YES   | NO      | YES       |
| <code>checkout [commit] &lt;paths&gt;</code> | NO   | YES   | YES     | <b>NO</b> |

## Mergen voor gevorderden

Mergen met Git is normaalgesproken redelijk eenvoudig. Omdat Git het je gemakkelijk maakt om meerdere malen de ene branch met de andere te mergen, betekent dit dat je branches met een lange levensduur kunt hebben maar dat je het gaandeweg up to date kunt houden, vaak kleine conflicten oplossend, in plaats van verrast te worden door een enorme conflict aan het eind van de reeks.

Echter, soms zullen lastige conflicten optreden. In tegenstelling tot andere versie beheer systemen probeert Git niet al te slim te zijn bij het oplossen van merge conflicten. De filosofie van Git is om slim te zijn over het bepalen wanneer een merge oplossing eenduidig is, maar als er een conflict is probeert het niet slim te zijn en het automatisch op te lossen. Dit is de reden dat als je te lang wacht met het meren van twee branches die snel uiteenlopen, dat je tegen een aantal situaties zult aanlopen.

In dit hoofdstuk zullen we van een aantal van die situaties laten zien wat de oorzaak kan zijn en welke instrumenten Git je geeft om je te helpen deze meer lastige situaties op te lossen. We zullen ook een aantal van de afwijkende, niet standaard type merges behandelen die je kunt uitvoeren, zowel als aangeven hoe je merges die je hebt uitgevoerd weer kunt terugdraaien.

### Merge conflicten

We hebben een aantal beginselen van het oplossen van merge conflicten in [Eenvoudige merge conflicten](#) behandeld, voor meer complexe conflicten geeft Git je een aantal instrumenten om uit te vinden wat er aan de hand is en hoe beter met het conflict om te gaan.

Als eerste, als het enigszins mogelijk is, probeer je werk directory op te schonen voor je een merge uitvoert die conflicten zou kunnen bevatten. Als je onderhanden werk hebt, commit dit naar een tijdelijke branch of stash het. Dit zorgt ervoor dat je **alles** kunt terugdraaien wat je hier probeert. Als je niet bewaarde wijzigingen in je werk-directory hebt als je een merge probeert, kunnen een aantal tips die we geven ervoor zorgen dat je dit verliest.

Laten we een erg eenvoudig voorbeeld doorlopen. We hebben een super simpel Ruby bestand dat *hello world* afdrukt.

```
#!/usr/bin/env ruby

def hello
    puts 'hello world'
end

hello()
```

In onze repository maken we een nieuwe branch genaamd `whitespace` en vervolgen we door alle Unix regel-einden te vervangen met DOS regel-einden, eigenlijk gewoon elke regel van het bestand wijzigend, maar alleen met witruimte. Dan wijzigen we de regel “hello world” in “hello mundo”.

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

def hello
- puts 'hello world'
+ puts 'hello mundo'^M
end

hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
1 file changed, 1 insertion(+), 1 deletion(-)
```

Nu switchen we terug naar onze `master`-branch en voegen wat documentatie aan de functie toe.

```

$ git checkout master
Switched to branch 'master'

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
    puts 'hello world'
end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)

```

Nu gaan we proberen onze **whitespace**-branch te mergen en we zullen conflicten krijgen vanwege de witruimte wijzigingen.

```

$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.

```

### Een merge afbreken

We kunnen nu kiezen. Als eerste, laten we bekijken hoe we uit deze situatie kunnen komen. Als je geen conflicten had verwacht en je wilt nu nog even niet met deze situatie te maken hebben, kan je eenvoudig de merge terugdraaien met **git merge --abort**.

```

$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master

```

De **git merge --abort** optie probeert de status terug te halen van voor je probeerde de merge te draaien. De enige gevallen waar dit misschien niet helemaal mogelijk is zou zijn als je `git-unstash`, onecommitte wijzigingen in je werk-directory zou hebben staan toen je het aanriep, in alle andere

gevallen zou het prima moeten werken.

Als je voor wat voor reden ook jezelf in een enorme bende weet te krijgen en je wilt gewoon opnieuw beginnen, kan je ook `git reset --hard HEAD` aanroepen of waar je ook naartoe wilt terugkeren. Onthoud: al het niet gecommitte werk gaat verloren, dus verzekер je ervan dat je hier geen onderhanden werk wilt hebben.

## Witruimtes negeren

In dit specifieke geval hebben de conflicten met witruimtes te maken. We weten dit omdat dit geval eenvoudig is, maar het is ook redelijk eenvoudig te achterhalen in praktijksituaties als je naar een conflict kijkt, omdat elke regel is verwijderd aan de ene kant en weer aan de andere kant wordt toegevoegd. Standaard ziet Git al deze regels als gewijzigd en kan het dus de bestanden niet mergen.

De standaard merge strategie kan echter argumenten meekrijgen, en een aantal van deze gaan over het op een nette manier negeren van witruimte wijzigingen. Als je ziet dat je een groot aantal witruimte issues hebt in een merge, kan je deze eenvoudigweg afbreken en nogmaals uitvoeren, deze keer met `-Xignore-all-space` of `-Xignore-space-change`. De eerste optie negeert alle witruimte **volledig** bij het vergelijken van de regels, en de tweede beschouwt volgorderlijke witruimte karakters als gelijk.

```
$ git merge -Xignore-space-change whitespace
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
hello.rb | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
```

Omdat in dit geval de eigenlijke bestandswijzigingen niet conflicteren, zal het negeren van de witruimte wijzigingen leiden tot een geslaagde merge.

Dit bespaart je veel werk als je iemand in je team hebt die regelmatig graag alles herformateert van spaties naar tabs of omgekeerd.

## Handmatig bestanden opnieuw mergen

Hoewel Git het voorbewerken van witruimtes redelijk goed doet, zijn er andere soorten van wijzigingen waar Git misschien niet automatisch mee kan gaan, maar zijn er oplossingen die met scripts gevonden kunnen worden. Als voorbeeld, stel dat Git de witruimte wijziging niet had kunnen verwerken en we hadden dit met de hand moeten doen.

Wat we echt zouden moeten doen is het bestand dat we proberen te mergen met een `dos2unix` programma bewerken voordat we de echte bestandsmerge uitvoeren. Dus, hoe zouden we dat doen?

Allereerst moeten we in de situatie van de merge conflict geraken. Dan willen we kopieën maken van mijn versie van het bestand, hun versie van het bestand (van de branch die we proberen te mergen) en de gezamelijke versie (van waar beide branches zijn afgesplitst). Daarna willen we een van de twee kanten verbeteren (die van hun of van ons) en dan de merge opnieuw proberen maar

dan voor alleen deze ene bestand.

Het krijgen van de drie bestandversies is eigenlijk vrij makkelijk. Git bewaart al deze versies in de index onder “stages” die elk met getallen zijn geassocieerd. Stage 1 is de gezamelijke voorouder, stage 2 is jouw versie en stage 3 is van de `MERGE_HEAD`, de versie die je probeert te mergen (“theirs”).

Je kunt een kopie van elk van deze versie van het conflicterende bestand met het `git show` commando gecombineerd met een speciale syntax verkrijgen.

```
$ git show :1:hello.rb > hello.common.rb  
$ git show :2:hello.rb > hello.ours.rb  
$ git show :3:hello.rb > hello.theirs.rb
```

Als je iets dichter *op het ijzer* wilt werken, kan je ook het `ls-files -u` binnentwerk commando gebruiken om de echte SHA-1 nummers op te zoeken van de Git blobs voor elk van deze bestanden.

```
$ git ls-files -u  
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1 hello.rb  
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2 hello.rb  
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3 hello.rb
```

Het `:1:hello.rb` is gewoon een verkorte manier om de SHA-1 van de blob op te zoeken.

Nu we de inhoud van de drie stages in onze werk directory hebben, kunnen we handmatig die van hun opknappen om het witruimte probleem op te lossen en de bestanden opnieuw te mergen met het vrij onbekende `git merge-file` commando die precies dat doet.

```
$ dos2unix hello.theirs.rb
dos2unix: converting file hello.theirs.rb to Unix format ...

$ git merge-file -p \
    hello.ours.rb hello.common.rb hello.theirs.rb > hello.rb

$ git diff -b
diff --cc hello.rb
index 36c06c8,e85207e..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,8 -1,7 +1,8 @@@
  #! /usr/bin/env ruby

+## prints out a greeting
def hello
-  puts 'hello world'
+  puts 'hello mundo'
end

hello()
```

We hebben op dit moment het bestand fijn gemerged. Sterker nog, dit werkt eigenlijk beter dan de `ignore-space-change` optie omdat dit echt de witruimte wijzigingen verbetert voor we mergen inplaats van ze gewoon te negeren. In de `ignore-space-change` merge, eindigen we uiteindelijk met een paar regels met DOS regel-einden, waardoor we dingen vermengen.

Als je voordat de commit wordt beeindigd een indruk wilt krijgen van wat daadwerkelijk gewijzigd is tussen de ene en de andere kant, kan je `git diff` vragen om wat in je werk directory zit te vergelijken met wat je van plan bent te committen als resultaat van de merge met elk van deze stages. Laten we ze eens allemaal bekijken.

Om het resultaat te vergelijken met wat je in je branch had voor de merge, met andere woorden om te zien wat de merge geïntroduceerd heeft, kan je `git diff --ours` aanroepen

```

$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@
 
 # prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()

```

Dus hier kunnen we makkelijk zien wat er in onze branch gebeurd is, wat we daadwerkelijk in dit bestand introduceren met deze merge is de wijziging in die ene regel.

Als we willen zien hoe het resultaat van de merge verschilt met wat er op hun kant stond, kan je `git diff --theirs` aanroepen. In deze en het volgende voorbeeld, moeten we `-b` gebruiken om de witruimtes te verwijderen omdat we het vergelijken met wat er in Git staat, niet onze opgeschoonde `hello.their.rb` bestand.

```

$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
#! /usr/bin/env ruby

+# prints out a greeting
def hello
    puts 'hello mundo'
end

```

Als laatste kan je zien hoe het bestand is veranderd ten opzichte van beide kanten met `git diff --base`.

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
#! /usr/bin/env ruby

+# prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()
```

Op dit moment kunnen we het `git clean` commando gebruiken om de extra bestanden te verwijderen die we hebben gemaakt om de handmatige merge uit te voeren maar die we niet langer meer nodig hebben.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

## Conflicten beter bekijken

Misschien zijn we om de een of andere reden niet tevreden met de huidige oplossing, of misschien werkt het handmatig wijzigen van een of beide kanten nog steeds niet goed en moeten we meer van de omstandigheden te weten komen.

Laten we het voorbeeld een beetje veranderen. In dit voorbeeld hebben we twee langer doorlopende branches die elk een aantal commits hebben maar die, wanneer ze worden gemoed, een echt conflict op inhoud opleveren.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|
* b7dcc89 initial hello world code
```

We hebben nu drie unieke commits die alleen in de `master`-branch aanwezig zijn en drie andere die

op de `mundo`-branch zitten. Als we de `mundo`-branch willen mergen krijgen we een conflict.

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

We zouden willen zien wat de merge conflict is. Als we het bestand openen, zie we iets als dit:

```
#! /usr/bin/env ruby

def hello
<<<<< HEAD
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>> mundo
end

hello()
```

Beide kanten van de merge hebben inhoud aan dit bestand toegevoegd, maar een aantal van de commits hebben het bestand op dezelfde plaats gewijzigd waardoor dit conflict optreedt.

Laten we een aantal instrumenten verkennen die je tot je beschikking hebt om te bepalen hoe dit conflict tot stand is gekomen. Misschien is het niet duidelijk hoe je dit conflict precies moet oplossen. Je hebt meer kennis van de context nodig.

Een handig instrument is `git checkout` met de `--conflict` optie. Dit zal het bestand opnieuw uitchecken en de merge conflict markeringen vervangen. Dit kan nuttig zijn als je de markeringen wilt verwijderen en de conflicten opnieuw wilt oplossen.

Je kunt aan `--conflict` of `diff3` of `merge` doorgeven (de laatste is de standaard). Als je het `diff3` doorgeeft zal Git andere soorten conflict markeringen gebruiken, waarbij je niet alleen de “ours” en “theirs” versies krijgt, maar ook de “base” versie *inline* waardoor je meer context krijgt.

```
$ git checkout --conflict=diff3 hello.rb
```

Als we dat nu aanroepen, zal het bestand er nu zo uit zien:

```

#!/usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=====
  puts 'hello mundo'
>>>>> theirs
end

hello()

```

Als dit formaat je bevalt, kan je dit als standaard instellen voor toekomstige merge conflicten door de `merge.conflictstyle` instelling op `diff3` te zetten.

```
$ git config --global merge.conflictstyle diff3
```

Het `git checkout` commando kan ook de opties `--ours` en `--theirs` verwerken, wat een ontzettend snelle manier kan zijn om gewoon een van de twee kanten te kiezen waarbij er gewoon niet gemerged zal worden.

Dit is in het bijzonder handig voor conflicten tussen binaire bestanden waar je gewoon een kant kiest, of waar je alleen bepaalde bestanden wilt mergen van een andere branch - je kunt de merge uitvoeren en dan bepaalde bestanden bekijken dan een of de andere kant voordat je commit.

## Merge log

Een ander nuttig instrument bij het oplossen van merge conflicten is `git log`. Dit kan je helpen bij het verkrijgen van inzicht in wat kan hebben bijgedragen tot het conflict. Een stukje historie nakijken om boven water te krijgen waarom twee ontwikkelingen dezelfde gebieden raakten kan soms erg behulpzaam zijn.

Om een complete lijst te krijgen van alle unieke commits die in elk van beide branches zitten en die betrokken zijn bij deze merge, kunnen we de “drievoudige punt” syntax gebruiken die we geleerd hebben in [Drievoudige punt](#).

```

$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo

```

Dat is een mooie lijst van de in totaal zes betrokken commits, zowel als bij welke ontwikkellijn elke

commit gedaan is.

We kunnen dit echter verder vereenvoudigen om ons een meer specifieke context te geven. Als we de `--merge` optie gebruiken bij `git log`, zal het alleen de commits tonen van beide kanten van de merge die een bestand raken dat op dit moment in een conflict betrokken is.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

Als je het daarentegen met de `-p` optie aanroeft, krijg je alleen de diffs met het bestand dat in een conflict betrokken is geraakt. Dit kan **heel** handig zijn bij het snel verkrijgen van de context die je nodig hebt om te begrijpen waarom iets conflicteert en hoe het beter overwogen op te lossen.

### Gecombineerde diff formaat

Omdat Git alle merge resultaten die succesvol zijn staget, zal het aanroepen van `git diff` terwijl je in een conflicterende merge status zit, je alleen laten zien wat op dit moment zich nog steeds in een conflicterende status bevindt. Dit kan heel handig zijn om te zien wat je nog steeds moet oplossen.

Als je `git diff` direct aanroeft na een merge conflict, zal het je informatie geven in een nogal unieke diff uitvoer formaat.

```
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,11 @@@
 #! /usr/bin/env ruby

 def hello
++<<<<< HEAD
+   puts 'hola world'
+=====
+   puts 'hello mundo'
++>>>>> mundo
 end

hello()
```

Dit formaat heet “Gecombineerde Diff” (Combined Diff) en geeft je twee kolommen met gegevens naast elke regel. De eerste kolom laat je zien dat die regel anders is (toegevoegd of verwijderd) tussen de “ours” branch en het bestand in je werk directory en de tweede kolom doet hetzelfde tussen de “theirs” branch en de kopie in je werk directory.

Dus in het voorbeeld kan je zien dat de `<<<<<` en `>>>>>` regels in de werk kopie zitten maar in geen van beide kanten van de merge. Dit is logisch omdat de merge tool ze daar in heeft gezet voor

onze context, maar het wordt van ons verwacht dat we ze weghalen.

Als we het conflict oplossen en `git diff` nogmaals aanroepen, zien we hetzelfde, maar het is iets bruikbaarder.

```
$ vim hello.rb
$ git diff
diff --cc hello.rb
index 0399cd5,59727f0..0000000
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
#! /usr/bin/env ruby

def hello
- puts 'hola world'
- puts 'hello mundo'
++ puts 'hola mundo'
end

hello()
```

Het laat ons zien dat “hola world” aan onze kant zat maar niet in de werk kopie, dat “hello mundo” aan hun kant stond maar niet in de werk kopie en uiteindelijk dat “hola mundo” in geen van beide kanten zat maar nu in de werk kopie staat. Dit kan nuttig zijn om na te kijken voordat de oplossing wordt gecommit.

Je kunt dit ook krijgen van de `git log` voor elke merge nadat deze is gedaan om achteraf te zien hoe iets was opgelost. Git zal deze uitvoer-vorm kiezen als je `git show` aanroept op een merge commit, of als je de `--cc` optie gebruikt bij een `git log -p` (die standaard alleen patces voor non-merge commits laat zien).

```

$ git log --cc -p -1
commit 14f41939956d80b9e17bb8721354c33f8d5b5a79
Merge: f1270f7 e3eb223
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Sep 19 18:14:49 2014 +0200

Merge branch 'mundo'

Conflicts:
  hello.rb

diff --cc hello.rb
index 0399cd5,59727f0..e1d0799
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
 #! /usr/bin/env ruby

 def hello
-   puts 'hola mundo'
-   puts 'hello mundo'
++   puts 'hola mundo'
 end

hello()

```

## Merges ongedaan maken

Nu je weet hoe merge commits te maken, zal je waarschijnlijk er een aantal per ongeluk maken. Een van de mooie dingen van het werken met Git is dat het niet erg is om vergissingen te begaan, omdat het mogelijk is (en in veel gevallen makkelijk) om ze te herstellen.

Merge commits zijn niet anders. Stel dat je bent begonnen met werken op een topic branch, deze abusiefelijk in `master` heb gemerged, en nu zie je commit historie er zo uit:

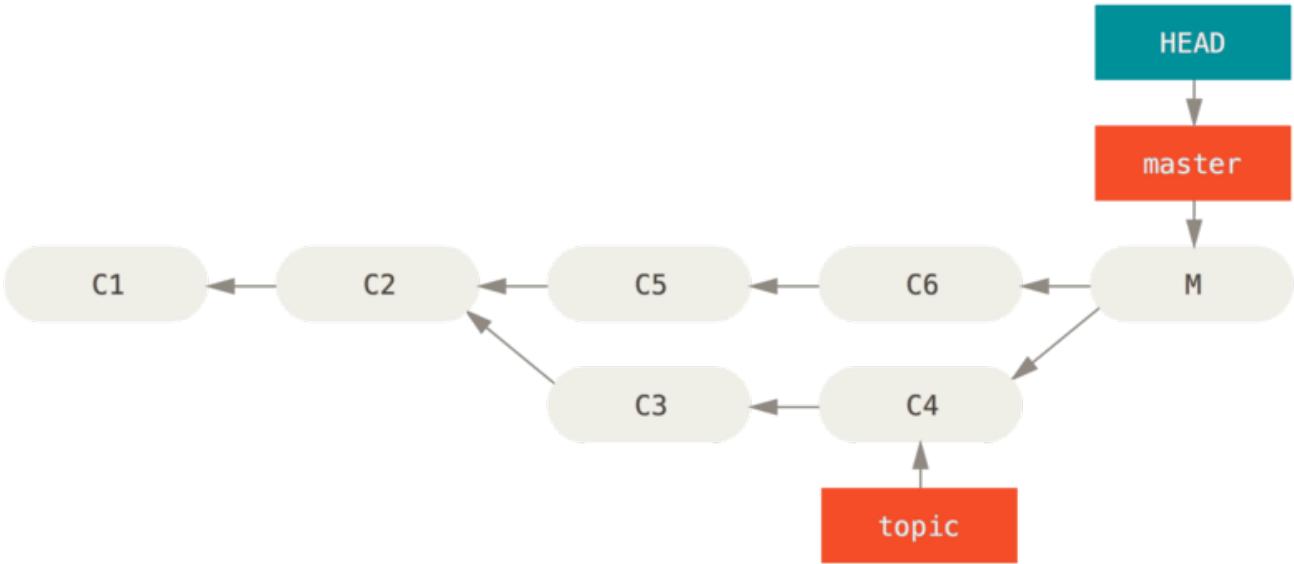


Figure 138. Abusievelijke merge commit

Er zijn twee manieren om dit probleem te benaderen, afhankelijk van wat de gewenste uitkomst is.

### De referenties herstellen

Als de ongewilde merge commit alleen bestaat in je lokale repository, is de eenvoudigste en beste oplossing om de branches dusdanig te verplaatsen dat ze wijzen naar waar je wilt hebben. In de meeste gevallen, als je de foutieve `git merge` opvolgt met `git reset --hard HEAD~`, zal dit de branch verwijzingen herstellen zodat ze er zo uit zien:

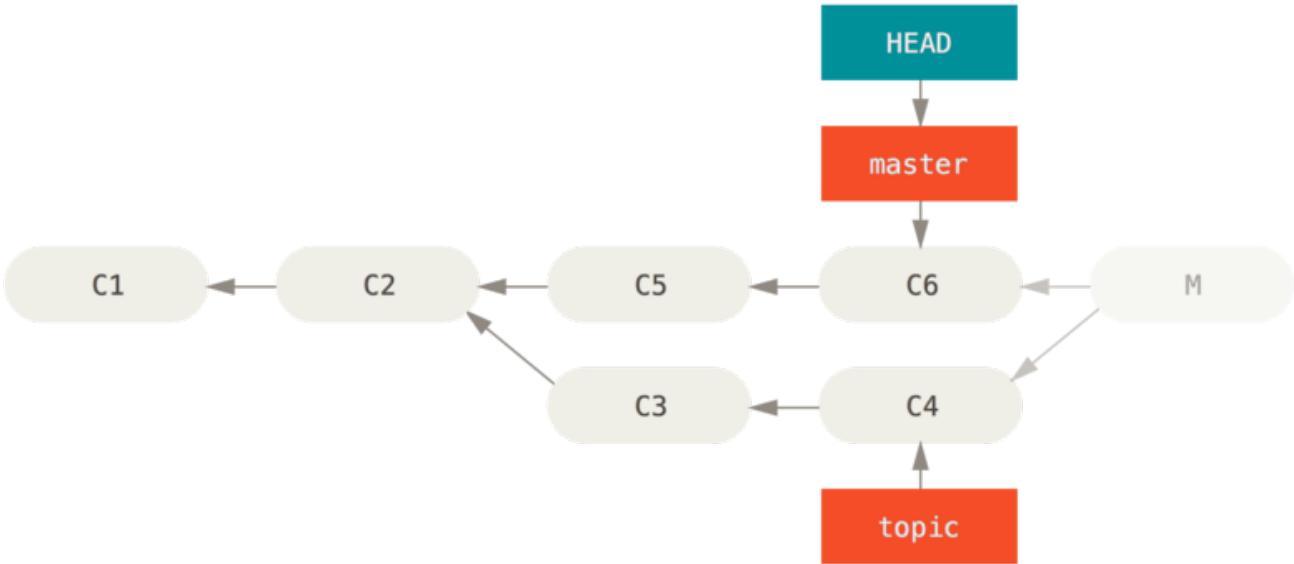


Figure 139. Historie na `git reset --hard HEAD~`

We hebben `reset` in [Reset ontrafeld](#) behandeld, dus het zou niet al te moeilijk uit te vinden wat hier gebeurt. Hier is een snelle opfrisser: `reset --hard` volgt normaalgesproken de volgende drie stappen:

1. Verplaats de branch waar `HEAD` naar wijst In dit geval willen we `master` verplaatsen naar waar het was voor de merge commit (`C6`).

2. Laat de index eruit zien als HEAD.
3. Laat de werk directory eruit zien als de index.

Het nadeel van deze aanpak is dat het de historie herschrijft, wat problematisch kan zijn met een gedeelde repository. Bestudeer [De gevaren van rebasen](#) om te zien wat er dan gebeuren kan; in het kort houdt het in dat als andere mensen de commits hebben die jij aan het herschrijven bent, je [reset](#) eigenlijk wilt vermijden. Deze aanpak zal ook niet werken als er andere commits gemaakt zijn sinds de merge; het verplaatsen van de referenties doet deze wijzigingen ook teniet.

## De commit terugdraaien

Als het verplaatsen van de branch verwijzingen niet gaat werken voor je, geeft Git je de optie van het maken van een nieuwe commit die alle wijzigingen van een bestaande terugdraait. Git noemt deze operatie een “revert”, en in dit specifieke scenario zou je het als volgt aanroepen:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

De `-m 1` vlag geeft aan welke ouder de “hoofdlijn” (mainline) is en behouden moet blijven. Als je een merge naar `HEAD` begint (`git merge topic`), heeft de nieuwe commit twee ouders: de eerste is `HEAD` (`C6`), en de tweede is de punt van de branch die erin wordt gemerged (`C4`). In dit geval, willen we alle wijzigingen die zijn geïntroduceerd door het mergen van ouder #2 (`C4`) terugdraaien, terwijl we de alle inhoud van ouder #1 (`C6`) behouden.

De historie met de terugdraaiende commit ziet er zo uit:

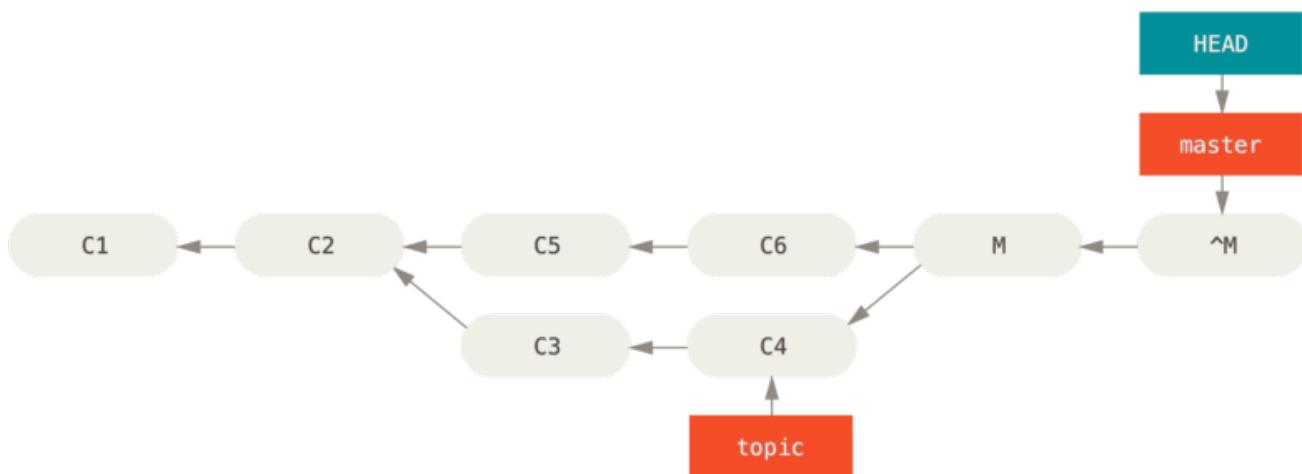


Figure 140. Historie na `git revert -m 1`

De nieuwe commit `^M` heeft precies dezelfde inhoud als `C6`, dus beginnende vanaf hier is het alsof de merge nooit heeft plaatsgevonden, behalve dat de commits die zijn ge-unmerged nog steeds in de geschiedenis van `HEAD` aanwezig zijn. Git zal in de war raken als je probeert `topic` weer in `master` te mergen:

```
$ git merge topic
Already up-to-date.
```

Er is niets in `topic` wat niet al bereikbaar is vanaf `master`. Wat erger is, als je werk toevoegt aan `topic` en weer merged, zal Git alleen de wijzigingen meenemen *sinds* de teruggedraaide merge:

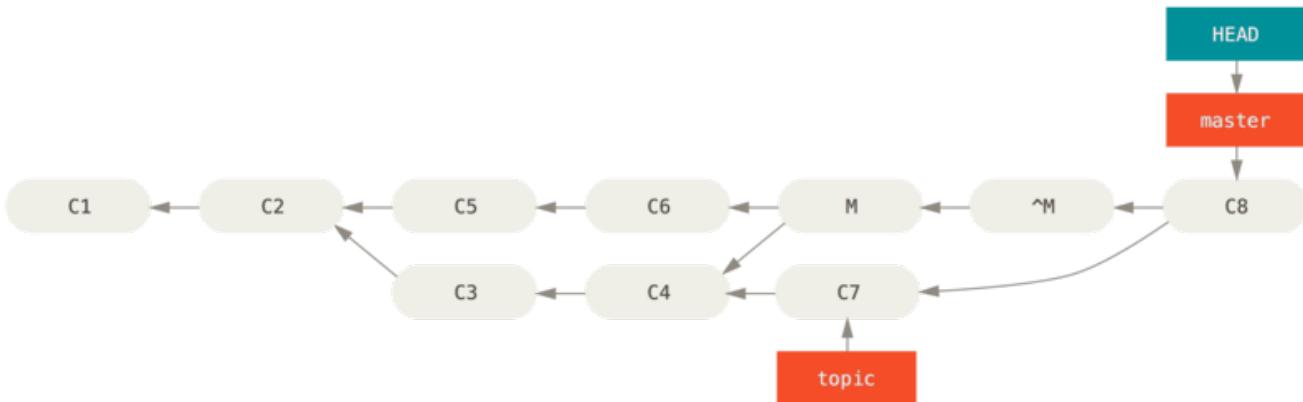


Figure 141. Historie met een slechte merge

De beste manier om hiermee om te gaan is om de originele merge te ont-terugdraaien, omdat je nu de wijzigingen wilt doorvoeren die eruit waren verwijderd door ze terug te draaien, en **dan** een nieuwe merge commit te maken:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'''"
$ git merge topic
```

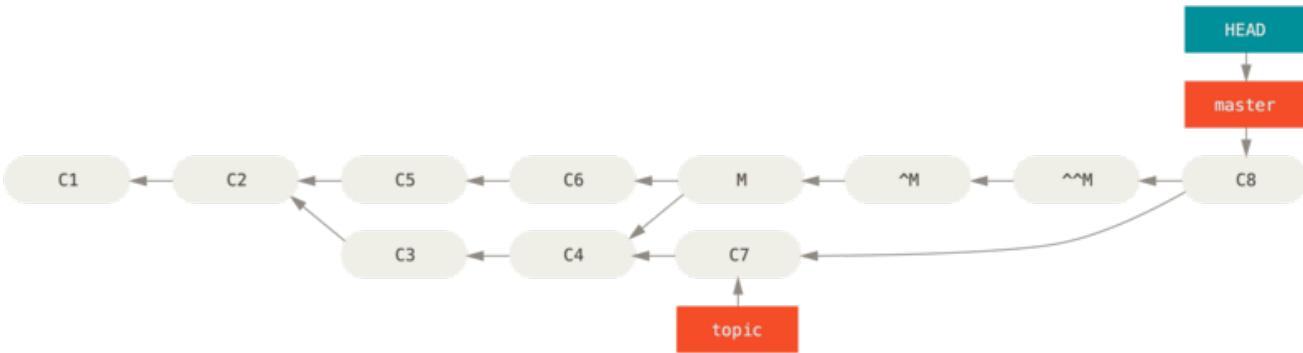


Figure 142. Historie na het opnieuw mergen van een teruggedraaide merge

In dit voorbeeld, neutraliseren `M` en `^M` elkaar. `^^M` merget effectief de wijzigingen van `C3` en `C4`, en `C8` merged de wijzigingen van `C7`, dus nu is `topic` volledig gemerged.

## Andere soorten merges

Tot zover hebben we de reguliere merge van twee branches behandeld, normaalgesproken afgehandeld met wat de “recursive” merge strategie wordt genoemd. Er zijn echter andere manieren om branches samen te voegen. Laten we een aantal van deze snel bespreken.

## Voorkeur voor de onze of de hunne

Allereerst, is er een ander nuttig iets wat we met de normale “recursive” merge wijze kunnen doen. We hebben de `ignore-all-space` en `ignore-space-change` opties gezien die met een `-X` worden doorgegeven, maar we kunnen Git ook vertellen om de ene of de andere kant de voorkeur te geven als het een conflict bespeurt.

Standaard als Git een conflict ziet tussen twee branche die worden gemerged, zal het merge conflict markeringen in je code toevoegen en het bestand als *conflicted* bestempelen en je het laten oplossen. Als je de voorkeur hebt dat Git eenvoudigweg een bepaalde kant kiest en de andere kant negeert in plaats van je handmatig het conflict te laten oplossen kan je het `merge` commando een `-Xours` of `-Xtheirs` doorgeven.

Als Git dit ziet, zal het geen conflict markeringen invoegen. Alle verschillen die mergebaar zijn zal het mergen. Bij alle verschillen die conflicteren zal het simpelweg in het geheel de kant kiezen doe je opgeeft, ook bij binaire bestanden.

Als we terugkijken naar het “hello world” voorbeeld die we hiervoor gaven, kunnen we zien dat mergen in onze branch conflicten gaf.

```
$ git merge mundo
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
```

Echter als we het gebruiken met `-Xours` of `-Xtheirs` gebeurt dit niet.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
hello.rb | 2 ++
test.sh  | 2 ++
2 files changed, 3 insertions(+), 1 deletion(-)
create mode 100644 test.sh
```

In dat geval, in plaats van conflict markeringen te krijgen in het bestand met “hello mundo” aan de ene en “hola world” aan de andere kant, zal het simpelweg “hola world” kiezen. Echter alle andere niet conflicterende wijzigingen in die branch zijn succesvol samengevoegd.

Deze optie kan ook aan het `git merge-file` commando die we eerder zagen worden doorgegeven door iets als `git merge-file --ours` aan te roepen voor individuele file merges.

Als je iets als die wilt doen maar Git niet eens wilt laten proberen wijzigingen van de andere kant te laten samenvoegen, is er een meer draconische optie, en dat is de “ours” merge *strategie*. Dit verschilt met de “ours” recursieve merge *optie*.

Dit zal feitelijk een nep merge uitvoeren. Het zal een nieuwe merge commit vastleggen met beide

branches als ouders, maar het zal niet eens kijken naar de branch die je merget. Het zal eenvoudigweg de exacte code in je huidige branch vastleggen als het resultaat van de merge.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Je kunt zien dat er geen verschil is tussen de branch waar we op zaten en het resultaat van de merge.

Dit kan vaak handig zijn als je Git wilt laten denken dat een branch al ingemerget is wanneer je later een merge aan het doen bent. Bijvoorbeeld, stel dat je een “release” gemaakt hebt en daar wat werk aan gedaan hebt dat je later zeker naar je `master`-branche wilt mergen. In de tussentijd moet er een of andere bugfix op `master` moet teruggebracht (backported) worden naar je `release`-branch. Je kunt de bugfix branch in de `release`-branch mergen en dezelfde branch ook met `merge -s ours` in je `master`-branch mergen (zelfs als de fix daar al aanwezig is) zodat later als je de `release`-branch weer merget, er geen conflicten met de bugfix zijn.

## Het mergen van subtrees

Het idee achter de subtree merge is dat je twee projecten hebt, en een van de projecten verwijst naar een subdirectory van de ander en vice versa. Als je een subtree merge specificeert, is Git vaak slim genoeg om uit te vinden dat de ene een subtree van de ander en zal daarvoor passend mergen.

We zullen een voorbeeld doornemen van het toevoegen van een separaat project in een bestaand project en dan de code mergen van de tweede naar een subdirectory van de eerste.

Eerst zullen we de Rack applicatie aan ons project toevoegen. We gaan het Rack project als een remote referentie in ons project toevoegen en deze dan uitchecken in zijn eigen branch:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Nu hebben we de root van het Rack project in onze `rack_branch`-branch en ons eigen project in de `master`-branch. Als je eerst de ene en dan de andere uitcheckt, kan je zien dat ze verschillende project roots hebben:

```
$ ls
AUTHORS      KNOWN-ISSUES  Rakefile    contrib     lib
COPYING      README        bin         example     test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Dit is een beetje een raar concept. Het is niet verplicht dat alle branches in je repository branches van hetzelfde project zijn. Het is niet iets wat vaak voorkomt, omdat het zelden behulpzaam is, maar het is relatief eenvoudig om branches te hebben die volledig verschillende histories hebben.

In dit geval willen we het Rack project in onze `master`-project binnentrekken (pull) als een subdirectory. We kunnen dat in Git doen met `git read-tree`. Je zult meer over `read-tree` en zijn vriendjes leren in [Git Binnenwerk](#), maar neem voor nu aan dat het de root tree van een branch naar je huidige staging area en werk directory inleest. We zijn zojuist teruggeschakeld naar de `master`-branch, en we pullen de `rack_branch`-branch in de `rack` subdirectory van de `master`-branch van ons hoofdproject:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Als we committen, zal het lijken alsof we alle Rack bestanden onder die subdirectory hebben - alsof we ze vanuit een tarball gekopieerd hebben. Wat dit interessant maakt, is dat we relatief eenvoudig wijzigingen van de ene branch naar de andere kunnen mergen. Dus, als het Rack project wijzigt, kunnen we upstream wijzigingen binnentrekken door naar die branch over te schakelen en te pullen:

```
$ git checkout rack_branch
$ git pull
```

Daarna kunnen we die wijzigingen in onze `master`-branch mergen. Om de wijzigingen binnen te halen en de commit message alvast in te vullen, gebruik je de `--squash` optie zowel als de `-Xsubtree` optie van de recursieve merge strategy. (De recursieve strategie is hier de standaard, maar we voegen het voor de duidelijkheid toe).

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

All de wijzigingen van het Rack project zijn gemerged en klaar om lokaal te worden gecommit. Je kunt ook het tegenovergestelde doen - de wijzigingen in de `rack` subdirectory van je master branch

maken en ze dan later naar je `rack_branch`-branch mergen om ze dan in te dienen bij de beheerders of ze stroomopwaarts te pushen.

Dit is een manier om een workflow te krijgen die lijkt op de submodule workflow zonder submodules te gebruiken (wat we in [Submodules](#) zullen behandelen). We kunnen in onze repository branches aanmaken met andere gerelateerde projecten en ze bij tijd en wijle subtree mergen in ons project. Dit is in sommige opzichten handig, bijvoorbeeld omdat alle code op een enkele plaats wordt gecommit. Het heeft echter ook nadelen in de zin dat het iets complexer is en gevoeliger voor fouten in het herintegreren van wijzigingen of abusievelijk een branch te pushen naar een niet gerelateerde repository.

Een ander gek iets is dat om een diff te krijgen tussen wat je in je `rack` subdirectory hebt en de code in je `rack_branch`-branch - om te zien of je ze moet mergen - kan je niet het normale `diff` commando gebruiken. In plaats daarvan moet je `git diff-tree` aanroepen met de branch waar het je mee wilt vergelijken:

```
$ git diff-tree -p rack_branch
```

Of, om wat in je `rack` subdirectory zit te vergelijken met wat de `master`-branch op de server was de laatste keer dat je gefetcht hebt kan je dit aanroepen

```
$ git diff-tree -p rack_remote/master
```

## Rerere

De functionaliteit van `git rerere` is een beetje onbekend. De naam staat voor “reuse recorded resolution” (hergebruik opgenomen resoluties/oplossingen) en zoals de naam al aangeeft, stelt het je in staat om Git te vragen te onthouden hoe je een bepaald deel van een conflict hebt opgelost zodat Git, als het de volgende keer een vergelijkbaar conflict ziet, deze automatisch voor je kan oplossen.

Er zijn een aantal scenarios waarin deze functionaliteit erg handig zou kunnen zijn. Een van de voorbeelden dat in de documentatie wordt genoemd is dat je ervoor wilt zorgen dat een langlevende topic branch netjes zal mergen maar dat je niet een berg tussenliggende merge commits hoeft te maken die je historie vervuilen. Met `rerere` ingeschakeld kan je af en toe mergen, de conflicten oplossen en dan de merge terugdraaien. Als je dit doorlopend doet, zou de laatste merge eenvoudig moeten zijn omdat `rerere` alles gewoon automatisch voor je kan doen.

Deze zelfde taktiek kan gebruikt worden als je een branch rebased wilt houden zodat je niet elke keer met dezelfde rebasing conflicten te maken krijgt elke keer als je dit doet. Of als je een branch hebt die je hebt gemerged en daar een bergje conflicten hebt opgelost en dan besluit om deze toch maar te rebasen — je zult waarschijnlijk niet dezelfde conflicten willen doorlopen.

Een andere toepassing van `rerere` is er een waar je een af en toe aantal in ontwikkeling zijnde topic branches wilt mergen in een testbare head, zoals in het Git project zelf ook vaak gebeurt. Als de tests falen, kan je de merges terugdraaien en ze weer doen zonder de topic branch die de tests liet falen zonder de conflicten opnieuw te moeten oplossen.

Om de `rerere` functionaliteit in te schakelen, kan je eenvoudigweg de volgende configuratie setting aanroepen:

```
$ git config --global rerere.enabled true
```

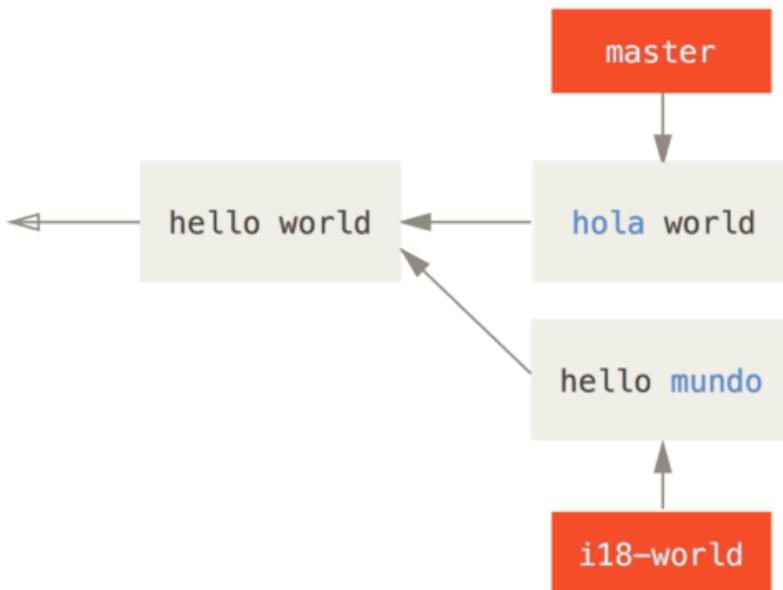
Je kunt het ook inschakelen door de `.git/rr-cache` directory in een specifieke repository aan te maken, maar de configuratie setting is duidelijker en het kan globaal gedaan worden.

Laten we nu eens een eenvoudig voorbeeld bekijken, vergelijkbaar met de vorige. Laten we zeggen dat we een bestand hebben dat er als volgt uitziet:

```
#! /usr/bin/env ruby

def hello
  puts 'hello world'
end
```

In de ene branch hebben we het woord “hello” in “hola” gewijzigd, en daarna in de andere branch veranderen we “world” in “mundo”, net zoals eerder.



Als we de twee branches mergen, zullen we een merge conflict krijgen:

```
$ git merge i18n-world
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Recorded preimage for 'hello.rb'
Automatic merge failed; fix conflicts and then commit the result.
```

Je zult de nieuwe regel `Recorded preimage for FILE` hier opmerken. Verder zou het er precies als een

normale merge conflict uit moeten zien. Op dit moment kan `rerere` ons een aantal dingen vertellen. Normaalgesproken zou je een `git status` kunnen aanroepen om te zien waar de conflicten zitten:

```
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
# both modified:    hello.rb
#
```

Echter, `git rerere` zal je ook vertellen waar het de pre-merge status voor heeft opgenomen met `git rerere status`:

```
$ git rerere status
hello.rb
```

En `git rerere diff` zal ons de huidige staat van de resolutie laten zien—waar je mee begonnen bent met oplossen en waar je het in hebt opgelost.

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,11 @@
 #! /usr/bin/env ruby

 def hello
-<<<<<
- puts 'hello mundo'
-=====
+<<<<< HEAD
    puts 'hola world'
->>>>>
+=====
+ puts 'hello mundo'
+>>>>> i18n-world
 end
```

Daarnaast (en dit is eigenlijk niet gerelateerd aan `rerere`), kan je `ls-files -u` gebruiken om de conflicterende bestanden en de voor, links en rechts versies te zien:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1 hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2 hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3 hello.rb
```

Je kunt het oplossen zodat het alleen `puts 'hola mundo'` wordt en je kunt het `rerere diff` commando nog een keer aanroepen om te zien wat rerere zal onthouden:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
#! /usr/bin/env ruby

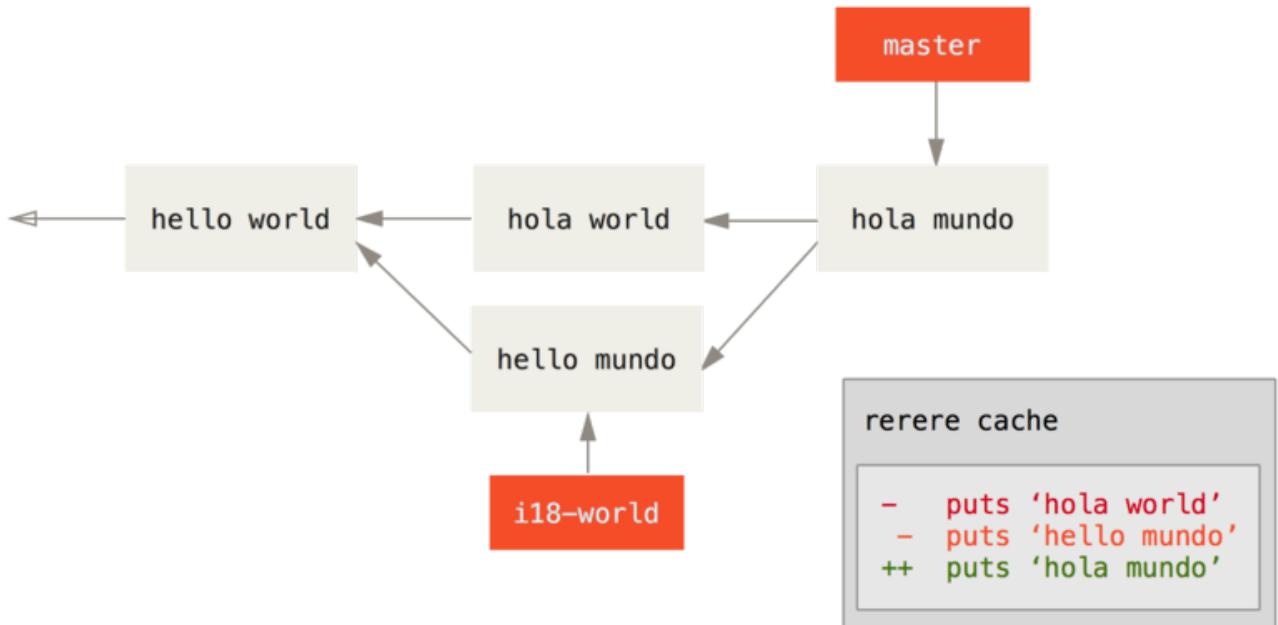
def hello
-<<<<<
- puts 'hello mundo'
-=====
- puts 'hola world'
->>>>>
+ puts 'hola mundo'
end
```

Dit zegt eigenlijk dat, wanneer Git een conflict in een deel van een `hello.rb` bestand ziet waar “hello mundo” aan de ene en “hola world” aan de andere kant staat, het zal oplossen naar “hola mundo”.

Nu kunnen we het als opgelost markeren en committen:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

Je kunt zien aan de boodschap "Recorded resolution for FILE" zien dat het de resolutie voor het bestand heeft opgeslagen.



Laten we nu die merge eens ongedaan maken, en in plaats daarvan deze op onze master branch gaan rebasen. We kunnen onze branch terugzetten door `reset` te gebruiken zoals we zagen in [Reset ontrafeld](#).

```
$ git reset --hard HEAD^  
HEAD is now at ad63f15 i18n the hello
```

Onze merge is ongedaan gemaakt. Laten we de topic branch gaan rebasen.

```
$ git checkout i18n-world
Switched to branch 'i18n-world'

$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: i18n one word
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Resolved 'hello.rb' using previous resolution.
Failed to merge in the changes.
Patch failed at 0001 i18n one word
```

We hebben nu dezelfde merge conflict zoals verwacht, maar kijk eens naar de regel met `Resolved FILE using previous resolution`. Als we nu het bestand bekijken zullen we zien dat het al is opgelost, er staan geen merge conflict markeringen in.

```
#!/usr/bin/env ruby
```

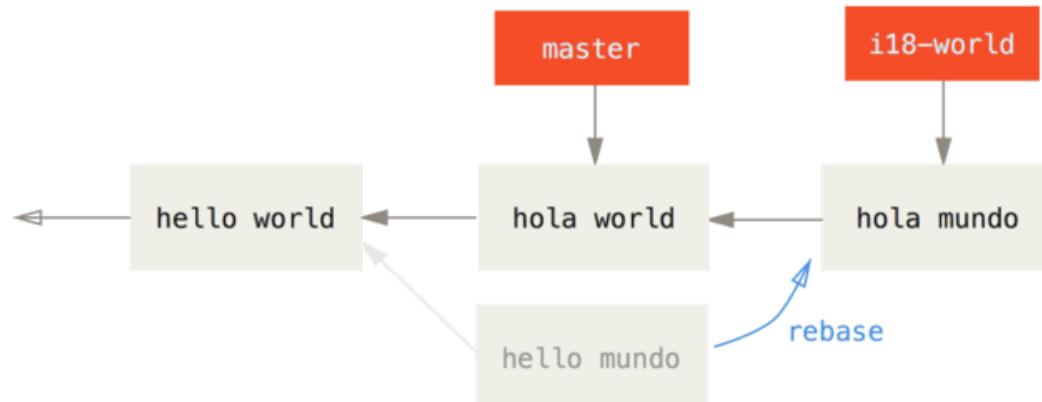
```
def hello
  puts 'hola mundo'
end
```

Ook zal `git diff` je laten zien hoe het automatisch opnieuw was opgelost:

```
$ git diff
diff --cc hello.rb
index a440db6,54336ba..0000000
```

```
--- a/hello.rb
+++ b/hello.rb
@@@ -1,7 -1,7 +1,7 @@@
 #! /usr/bin/env ruby
```

```
def hello
-  puts 'hola world'
-  puts 'hello mundo'
++ puts 'hola mundo'
end
```



`rerere cache`

```
- puts 'hola world'
- puts 'hello mundo'
++ puts 'hola mundo'
```

Je kunt ook de staat van het conflicterende bestand opnieuw creeëren met het `git checkout` commando:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#!/usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola mundo'
=====
  puts 'hello mundo'
>>>>> theirs
end
```

We zagen hier eerder een voorbeeld van in [Mergen voor gevorderden](#). Voor nu echter, laten we het opnieuw oplossen door eenvoudigweg weer **rerere** aan te roepen:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#!/usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

We hebben het bestand automatisch her-opgelost door de opgeslagen **rerere** resolutie te gebruiken. Je kunt het nu toevoegen en de rebase vervolgen om het te voltooien.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

Dus, als je vaak opnieuw merget, of je wilt een topic branch up-to-date houden met je master branch zonder talloze merges, of als je vaak rebaset, kan je **rerere** aanzetten om je leven wat aangenamer te maken.

## Debuggen met Git

Additioneel aan het feit dat het primair voor versiebeheer is, levert Git ook een aantal instrumenten om je te helpen met het debuggen van problemen in je projecten. Omdat Git is ontworpen om te werken met bijna alle soorten projecten, zijn deze instrumenten redelijk generiek, maar ze kunnen je vaak helpen om een fout op te sporen of een schuldige aan te wijzen als dingen fout gaan.

### Bestands annotatie

Als je op zoek bent naar een bug in je code en je wilt weten wanneer deze er in is geslopen en

waarom, is bestands annotatie vaak het beste gereedschap. Het laat voor alle regels in alle bestanden zien welke de commit de laatste was die een wijziging aanbracht. Dus, als je ziet dat een methode in je code labiel is, kan je het bestand annoteren met `git blame` om te zien welke commit verantwoordelijk was voor de introductie van die regel.

Het volgende voorbeeld gebruikt `git blame` om te bepalen welke commit en committer verantwoordelijk zijn voor regels in de top-level Linux kernel `Makefile` en, vervolgens, de `-L` optie om de uitvoer van de geannoteerde regels te beperken tot regels 69 tot en met 82 van dat bestand:

```
$ git blame -L 69,82 Makefile
b8b0618cf6fab (Cheng Renquan 2009-05-26 16:03:07 +0800 69) ifeq ("$(origin V)",
"command line")
b8b0618cf6fab (Cheng Renquan 2009-05-26 16:03:07 +0800 70) KBUILD_VERBOSE = $(V)
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 71) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 72) ifndef KBUILD_VERBOSE
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 73) KBUILD_VERBOSE = 0
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 74) endif
^1da177e4c3f4 (Linus Torvalds 2005-04-16 15:20:36 -0700 75)
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 76) ifeq ($(KBUILD_VERBOSE),1)
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 77) quiet =
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 78) Q =
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 79) else
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 80) quiet=quiet_
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 81) Q = @
066b7ed955808 (Michal Marek 2014-07-04 14:29:30 +0200 82) endif
```

Merk op dat het eerste veld een deel is van de SHA-1 van de commit die het laatste die regel wijzigde. De volgende twee velden zijn waarden die uit die commit zijn gehaald—de naam van de auteur en de schrijfdatum van die commit—zodat je eenvoudig kunt zien wie de regel gewijzigd heeft en wanneer. Daarna volgt het regelnummer en de inhoud van het bestand. Merk ook de `^1da177e4c3f4` commit regels op, waar de `^` prefix de regels aangeeft dit in de initiele commit van de repository aan dit project zijn toegevoegd, en deze regels zijn sindsdien niet gewijzigd. Dit is een beetje verwarring, omdat je nu op z'n minst drie verschillende manieren hebt gezien waarop Git het `^`-teken heeft gebruikt om een SHA-1 van een commit te duiden, maar dat is wat het hier betekent.

Een ander gaaf iets van Git is dat het bestandsnaam wijzigingen niet expliciet bijhoudt. Het slaat de snapshots op en probeert dan impliciet uit te vinden dat het hernoemd is, nadat het gebeurd is. Een van de interessante toepassingen hiervan is dat je het ook kunt vragen allerhande code verplaatsingen uit te vinden. Als je `-C` aan `git blame` meegeeft, zal Git het bestand dat je annoeerd analiseren en probeert het uit te vinden waar delen van de code in dat bestand oorspronkelijk vandaan kwamen als ze van elders waren gekopieerd. Bijvoorbeeld, stel dat je een bestand genaamd `GITServerHandler.m` aan het herstructureren bent in meerdere bestanden, waarvan er een `GITPackUpload.m` heet. Door `GITPackUpload.m` te blamen met de `-C` optie, kan je zien waar delen van de code oorspronkelijk vandaan kwamen:

```
$ git blame -C -L 141,153 GITPackUpload.m
f344f58d GITServerHandler.m (Scott 2009-01-04 141)
f344f58d GITServerHandler.m (Scott 2009-01-04 142) - (void) gatherObjectShasFromC
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144)           //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m (Scott 2009-03-24 145)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 146)           NSString *parentSha;
ad11ac80 GITPackUpload.m (Scott 2009-03-24 147)           GITCommit *commit = [g
ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 149)           //NSLog(@"GATHER COMMI
ad11ac80 GITPackUpload.m (Scott 2009-03-24 150)
56ef2caf GITServerHandler.m (Scott 2009-01-05 151)         if(commit) {
56ef2caf GITServerHandler.m (Scott 2009-01-05 152)         [refDict setOb
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

Dit is erg nuttig. Normaal gesproken krijg je als de oorspronkelijke commit, de commit waar je de code naartoe hebt gekopieerd, omdat dat de eerste keer is dat je deze regels in dit bestand hebt aangeraakt. Git geeft je de oorspronkelijke commit waarin je deze regels hebt geschreven, zelfs als dat in een ander bestand was.

## Binair zoeken

Een bestand annoteren helpt je als je meteen al weet waar het probleem is. Als je niet weet wat er kapot gaat, en er zijn tientallen of honderden commits geweest sinds de laatste staat waarin je weet dat de code werkte, zal je waarschijnlijk `git bisect` inschakelen voor hulp. Het `bisect` commando voert een binair zoektocht uit door je commit historie om je te helpen zo snel als mogelijk de commit te vinden die een probleem heeft geïntroduceerd.

Stel dat je zojuist een release van je code hebt ingevoerd in een productie omgeving, je krijgt fout rapporten over iets wat niet in je ontwikkelomgeving optrad, en je kunt je niet indenken waarom de code zich zo gedraagt. Je duikt in je code en het blijkt dat je het probleem kunt reproduceren, maar je kunt maar niet vinden waar het fout gaat. Je kunt de code *bisecten* om dit op te sporen. Eerst roep je `git bisect start` aan om het proces in gang te zetten, en dan gebruik je `git bisect bad` om het systeem te vertellen dat de huidige commit waar je op staat kapot is. Daarna moet je bisect vertellen waar de laatst bekende goede staat was, door ``git bisect good <goede_commit>` te gebruiken:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git kon opzoeken dat er ongeveer 12 commit zijn geweest tussen de commit die je als de laatst correcte hebt gemarkerd (v1.0) en de huidige slechte versie, en dat het de middelste voor je heeft uitgecheckt. Op dit moment kan je je tests laten lopen om te zien of het probleem op deze commit voorkomt. Als dit het geval is, dan was het ergens voor deze middelste commit erin geslopen; als

dat het niet het geval is, dan is het probleem na deze middelste commit geïntroduceerd. Het blijkt nu dat er hier geen probleem is, en je zegt Git dit door `git bisect good` in te typen en je zoektocht voort te zetten:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Nu zit je op een andere commit, halverwege tussen de ene die je zojuist getest hebt, en je slechte commit. Je gaat weer testen en ziet nu dat deze commit kapot is, dus je zegt dit met `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Deze commit is prima, en nu heeft Git alle informatie die het nodig heeft om te bepalen waar het probleem is begonnen. Het geeft je de SHA-1 van de eerste slechte commit en laat je wat van de commit informatie zien en welke bestanden gewijzigd waren in die commit zodat je kunt uitzoeken wat er gebeurd is waardoor deze fout optreedt:

```
$ git bisect good
b047b02ea83310a70fd603dc8cd7a6cd13d15c04 is first bad commit
commit b047b02ea83310a70fd603dc8cd7a6cd13d15c04
Author: PJ Hyett <pjhyett@example.com>
Date:   Tue Jan 27 14:48:32 2009 -0800

        secure this thing

:040000 040000 40ee3e7821b895e52c1695092db9bdc4c61d1730
f24d3c6ebcf639b1a3814550e62d60b8e68a8e4 M config
```

Als je klaar bent, moet je `git bisect reset` aanroepen om je HEAD terug te zetten naar waar je was voordat je startte, of je verzandt in een hele vreemde status:

```
$ git bisect reset
```

Dit is een krachtig instrument dat je kan helpen met het in enkele minuten doorzoeken van honderden commits voor een opgetreden fout. Als je een script hebt dat met 0 eindigt als het project correct en niet-0 als het project fout is, kan je het `git bisect` proces zelfs volledig automatiseren. Allereerst vertel je het de reikwijdte van de bisect door de bekende goede en slechte commits door te geven. Je kunt dit doen door ze te tonen met de `bisect start` commando als je dit wilt, door de bekende slechte commit als eerste door te geven en de bekende goede commit als tweede:

```
$ git bisect start HEAD v1.0
$ git bisect run test-error.sh
```

Op deze manier wordt `test-error.sh` automatisch aanroepen voor elke uitgecheckte commit totdat Git de eerste kapotte commit vindt. Je kunt ook zo iets als `make` of `make tests` aanroepen of wat je ook maar hebt dat geautomatiseerde tests voor je uitvoert.

## Submodules

Het gebeurt vaak dat terwijl je aan het werk bent op het ene project, je van daar uit een ander project moet gebruiken. Misschien is het een door een derde partij ontwikkelde library of een die je zelf elders aan het ontwikkelen bent en die je gebruikt in meerdere ouder (parent) projecten. Er ontstaat een veelvoorkomend probleem in deze scenario's: je wilt de twee projecten als zelfstandig behandelen maar ondertussen wel in staat zijn om de een vanuit de ander te gebruiken.

Hier is een voorbeeld. Stel dat je een web site aan het ontwikkelen bent en daar Atom feeds maakt. In plaats van je eigen Atom-genererende code te schrijven, besluit je om een library te gebruiken. De kans is groot dat je deze code moet insluiten vanuit een gedeelde library zoals een CPAN installatie of een Ruby gem, of de broncode naar je projecttree moet kopieëren. Het probleem met de library insluiten is dat het lastig is om de library op enige manier aan te passen aan jouw wensen en vaak nog moeilijker om het uit te rollen, omdat je jezelf ervan moet verzekeren dat elke gebruikende applicatie deze library beschikbaar moet hebben. Het probleem met het inbouwen van de code in je eigen project is dat elke eigen aanpassing het je moeilijk zal maken om te mergen als er stroomopwaarts wijzigingen beschikbaar komen.

Git adresseert dit probleem met behulp van submodules. Submodules staan je toe om een Git repository als een subdirectory van een andere Git repository op te slaan. Dit stelt je in staat om een andere repository in je eigen project te klonen en je commits apart te houden.

## Beginnen met submodules

We zullen een voorbeeld nemen van het ontwikkelen van een eenvoudig project die is opgedeeld in een hoofd project en een aantal sub-projecten.

Laten we beginnen met het toevoegen van een bestaande Git repository als een submodule van de repository waar we op aan het werk zijn. Om een nieuwe submodule toe te voegen gebruik je het `git submodule add` commando met de absolute of relatieve URL van het project dat je wilt gaan tracken. In dit voorbeeld voegen we een library genaamd "DbConnector" toe.

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Standaard zal submodules het subplot in een directory met dezelfde naam als de repository toevoegen, in dit geval “DbConnector”. Je kunt aan het eind van het commando een ander pad toevoegen als je het ergens anders wilt laten landen.

Als je `git status` op dit moment aanroeft, zullen je een aantal dingen opvallen.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

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

    new file:   .gitmodules
    new file:   DbConnector
```

Het eerste wat je moet opvallen is het nieuwe `.gitmodules` bestand. Dit is een configuratie bestand waarin de relatie wordt vastgelegd tussen de URL van het project en de lokale subdirectory waar je het in gepulld hebt:

```
[submodule "DbConnector"]
  path = DbConnector
  url = https://github.com/chaconinc/DbConnector
```

Als je meerdere submodules hebt, zal je meerdere van deze regels in dit bestand hebben. Het is belangrijk om op te merken dat dit bestand onder versiebeheer staat samen met je andere bestanden, zoals je `.gitignore` bestand. Het wordt met de rest van je project gepusht en gepulld. Dit is hoe andere mensen die dit project klonen weten waar ze de submodule projecten vandaan moeten halen.



Omdat de URL in het `.gitmodules` bestand degene is waarvan andere mensen als eerste zullen proberen te klonen of fetchen, moet je je ervan verzekeren dat ze er wel bij kunnen. Bijvoorbeeld, als je een andere URL gebruikt om naar te pushen dan waar anderen van zullen pullen, gebruik dan degene waar anderen toegang toe hebben. Je kunt deze waarde lokaal overschrijven met `git config submodule.DbConnector.url PRIVATE_URL` voor eigen gebruik. Waar van toepassing, kan een relatieve URL nuttig zijn.

De andere regel in de `git status` uitvoer is de entry voor de project folder. Als je `git diff` daarop aanroeft, zal je iets opvallends zien:

```
$ git diff --cached DbConnector
diff --git a/DbConnector b/DbConnector
new file mode 160000
index 0000000..c3f01dc
--- /dev/null
+++ b/DbConnector
@@ -0,0 +1 @@
+Subproject commit c3f01dc8862123d317dd46284b05b6892c7b29bc
```

Alhoewel **DbConnector** een subdirectory is in je werk directory, zie Git het als een submodule en zal de inhoud ervan niet tracken als je niet in die directory staat. In plaats daarvan ziet Git het als een specifieke commit van die repository.

Als een een iets betere diff uitvoer wilt, kan je de **--submodule** optie meegeven aan **git diff**.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+    path = DbConnector
+    url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000..c3f01dc (new submodule)
```

Als je commit, zal je iets als dit zien:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Merk de **160000** mode op voor de **DbConnector** entry. Dat is een speciale mode in Git wat gewoon betekent dat je een commit opslaat als een directory entry in plaats van een subdirectory of een bestand.

## Een project met submodules klonen

Hier zullen we een project met een submodule erin gaan klonen. Als je zo'n project kloont, krijg je standaard de directories die submodules bevatten, maar nog geen bestanden die daarin staan:

```

$ git clone https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
$ cd MainProject
$ ls -la
total 16
drwxr-xr-x  9 schacon  staff  306 Sep 17 15:21 .
drwxr-xr-x  7 schacon  staff  238 Sep 17 15:21 ..
drwxr-xr-x 13 schacon  staff  442 Sep 17 15:21 .git
-rw-r--r--  1 schacon  staff   92 Sep 17 15:21 .gitmodules
drwxr-xr-x  2 schacon  staff   68 Sep 17 15:21 DbConnector
-rw-r--r--  1 schacon  staff  756 Sep 17 15:21 Makefile
drwxr-xr-x  3 schacon  staff  102 Sep 17 15:21 includes
drwxr-xr-x  4 schacon  staff  136 Sep 17 15:21 scripts
drwxr-xr-x  4 schacon  staff  136 Sep 17 15:21 src
$ cd DbConnector/
$ ls
$
```

De `DbConnector` directory is er wel, maar leeg. Je moet twee commando's aanroepen: `git submodule init` om het lokale configuratie bestand te initialiseren, en `git submodule update` om alle gegevens van dat project te fetchen en de juiste commit uit te checken die in je superproject staat vermeld:

```

$ git submodule init
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
$ git submodule update
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

Nu is je `DbConnector` subdirectory in precies dezelfde staat als het was toen je het eerder commiteerde.

Er is echter een andere, iets eenvoudiger, manier om dit te doen. Als je `--recursive-submodules` doorgeeft aan het `git clone` commando zal het automatisch elke submodule in de repository initialiseren en updaten.

```
$ git clone --recurse-submodules https://github.com/chaconinc/MainProject
Cloning into 'MainProject'...
remote: Counting objects: 14, done.
remote: Compressing objects: 100% (13/13), done.
remote: Total 14 (delta 1), reused 13 (delta 0)
Unpacking objects: 100% (14/14), done.
Checking connectivity... done.
Submodule 'DbConnector' (https://github.com/chaconinc/DbConnector) registered for path
'DbConnector'
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
Submodule path 'DbConnector': checked out 'c3f01dc8862123d317dd46284b05b6892c7b29bc'
```

## Werken aan een project met submodules

Nu hebben we een kopie van een project met submodules erin en gaan we met onze teamgenoten samenwerken op zowel het hoofdproject als het submodule project.

### Wijzigingen van stroomopwaarts pullen

De eenvoudigste werkwijze bij het gebruik van submodules in een project zou zijn als je eenvoudigweg een subproject naar binnentrekt en de updates ervan van tijd tot tijd binnen haalt maar waarbij je niet echt iets wijzigt in je checkout. Laten we een eenvoudig voorbeeld doornemen.

Als je wilt controleren voor nieuw werk in een submodule, kan je in de directory gaan en `git fetch` aanroepen en `git merge` gebruiken om de wijzigingen uit de branch stroomopwaarts in de lokale code in te voegen.

```
$ git fetch
From https://github.com/chaconinc/DbConnector
  c3f01dc..d0354fc  master      -> origin/master
$ git merge origin/master
Updating c3f01dc..d0354fc
Fast-forward
  scripts/connect.sh | 1 +
  src/db.c           | 1 +
  2 files changed, 2 insertions(+)
```

Als je nu teruggaat naar het hoofdproject en `git diff --submodule` aanroeft kan je zien dat de submodule is bijgewerkt en je krijgt een lijst met commits die eraan is toegevoegd. Als je niet elke keer `--submodule` wilt intypen voor elke keer dat je `git diff` aanroeft, kan je dit als standaard formaat instellen door de `diff.submodule` configuratie waarde op "log" te zetten.

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

Als je nu gaat committen zal je de nieuwe code in de submodule insluiten als andere mensen updaten.

Er is ook een makkelijker manier om dit te doen, als je er de voorkeur aan geeft om niet handmatig te fetchen en mergen in de subdirectory. Als je `git submodule update --remote` aanroeft, zal Git naar je submodules gaan en voor je fetchen en updaten.

```
$ git submodule update --remote DbConnector
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
  3f19983..d0354fc master      -> origin/master
Submodule path 'DbConnector': checked out 'd0354fc054692d3906c85c3af05ddce39a1c0644'
```

Dit commando zal standaard aannemen dat je de checkout wilt updaten naar de `master`-branch van de submodule repository. Je kunt echter dit naar iets anders wijzigen als je wilt. Bijvoorbeeld, als je de DbConnector submodule de “stable” branch van die repository wilt laten tracken, kan je dit aangeven in het `.gitmodules` bestand (zodat iedereen deze ook trackt), of alleen in je lokale `.git/config` bestand. Laten we het aangeven in het `.gitmodules` bestand:

```
$ git config -f .gitmodules submodule.DbConnector.branch stable

$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 2), reused 4 (delta 2)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
  27cf5d3..c87d55d stable -> origin/stable
Submodule path 'DbConnector': checked out 'c87d55d4c6d4b05ee34fbc8cb6f7bf4585ae6687'
```

Als je de `-f .gitmodules` weglaat, zal het de wijziging alleen voor jou maken, maar het is waarschijnlijk zinvoller om die informatie bij de repository te tracken zodat iedereen dat ook zal gaan doen.

Als we nu `git status` aanroepen, zal Git ons laten zien dat we “new commits” hebben op de submodule.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

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

    modified:   .gitmodules
    modified:   DbConnector (new commits)

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

Als je de configuratie instelling `status.submodulesummary` instelt, zal Git je ook een korte samenvatting van de wijzigingen in je submodule laten zien:

```
$ git config status.submodulesummary 1

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

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

    modified:   .gitmodules
    modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c3f01dc...c87d55d (4):
  > catch non-null terminated lines
```

Als je op dit moment `git diff` aanroept kunnen we zien dat zowel we onze `.gitmodules` bestand hebben gewijzigd als dat daarbij er een aantal commmits is die we omlaag hebben gepulld en die klaar staan om te worden gecommit naar ons submodule project.

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
     path = DbConnector
     url = https://github.com/chaconinc/DbConnector
+    branch = stable
Submodule DbConnector c3f01dc..c87d55d:
> catch non-null terminated lines
> more robust error handling
> more efficient db routine
> better connection routine
```

Dit is best wel handig omdat we echt de log met commits kunnen zien waarvan we op het punt staan om ze in onze submodule te committen. Eens gecommit, kan je deze informatie ook achteraf zien als je `git log -p` aanroeft.

```
$ git log -p --submodule
commit 0a24cf121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
     path = DbConnector
     url = https://github.com/chaconinc/DbConnector
+    branch = stable
Submodule DbConnector c3f01dc..c87d55d:
> catch non-null terminated lines
> more robust error handling
> more efficient db routine
> better connection routine
```

Git zal standaard proberen **alle** submodules te updaten als je `git submodule update --remote` aanroeft, dus als je er hier veel van hebt, is het wellicht aan te raden om de naam van alleen die submodule door te geven die je wilt updaten.

## Werken aan een submodule

Het is zeer waarschijnlijk dat als je submodules gebruikt, je dit zult doen omdat je echt aan de code in die submodule wilt werken tegelijk met het werken aan de code in het hoofdproject (of verspreid over verschillende submodules). Anders zou je waarschijnlijk een eenvoudiger afhankelijkheidsbeheer systeem (dependency management system) hebben gebruikt (zoals Maven of Rubygems).

Dus laten we nu eens een voorbeeld behandelen waarin we gelijktijdig wijzigingen aan de submodule en het hoofdproject maken en deze wijzigingen ook gelijktijdig committen en publiceren.

Tot dusverre, als we het `git submodule update` commando aanriepen om met fetch wijzigingen uit de repositories van de submodule te halen, ging Git de wijzigingen ophalen en de files in de subdirectory updaten, maar zou het de subdirectory laten in een staat die bekend staat als “detached HEAD”. Dit houdt in dat er geen lokale werk branch is (zoals “master”, bijvoorbeeld) waar de wijzigingen worden getract. Zonder een werkbranch waarin de wijzigingen worden getract, betekent het dat zelfs als je wijzigingen aan de submodule commit, deze wijzigingen waarschijnlijk verloren zullen gaan bij de volgende keer dat je `git submodule update` aanroeft. Je zult een aantal extra stappen moeten zetten als je wijzigingen in een submodule wilt laten tracken.

Om de submodule in te richten zodat het eenvoudiger is om erin te werken, moet je twee dingen doen. Je moet in elke submodule gaan en een branch uitchecken om in te werken. Daarna moet je Git vertellen wat het moet doen als je wijzigingen hebt gemaakt en daarna zal `git submodule update --remote` nieuw werk van stroomopwaarts pullen. Je hebt nu de keuze om dit in je lokale werk te mergen, of je kunt proberen je nieuwe lokale werk te rebasen bovenop de nieuwe wijzigingen.

Laten we eerst in onze submodule directory gaan en een branch uitchecken.

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

Laten we het eens proberen met de “merge” optie. Om dit handmatig aan te geven kunnen we gewoon de `--merge` optie in onze `update` aanroep toevoegen. Hier zullen we zien dat er een wijziging op de server was voor deze submodule en deze wordt erin gemerged.

```
$ git submodule update --remote --merge  
remote: Counting objects: 4, done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 4 (delta 2), reused 4 (delta 2)  
Unpacking objects: 100% (4/4), done.  
From https://github.com/chaconinc/DbConnector  
  c87d55d..92c7337  stable    -> origin/stable  
Updating c87d55d..92c7337  
Fast-forward  
  src/main.c | 1 +  
  1 file changed, 1 insertion(+)  
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

Als we in de DbConnector directory gaan, hebben we de nieuwe wijzigingen al in onze lokale **stable**-branch gemerged. Laten we nu eens kijken wat er gebeurt als we onze lokale wijziging maken aan de library en iemand anders pusht tegelijk nog een wijziging stroomopwaarts.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'unicode support'
[stable f906e16] unicode support
 1 file changed, 1 insertion(+)
```

Als we nu onze submodule updaten kunnen we zien wat er gebeurt als we een lokale wijziging maken en er stroomopwaarts ook nog een wijziging is die we moeten verwerken.

```
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Als je de **--rebase** of **--merge** bent vergeten, zal Git alleen de submodule updaten naar wat er op de server staat en je lokale project in een detached HEAD status zetten.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Maak je geen zorgen als dit gebeurt, je kunt eenvoudigweg teruggaan naar deze directory en weer je branch uitchecken (die je werk nog steeds bevat) en handmatig **origin/stable** mergen of rebasen (of welke remote branch je wilt).

Als je jouw wijzigingen aan je submodule nog niet hebt gecommit en je roept een submodule update aan die problemen zou veroorzaken, zal Git de wijzigingen ophalen (fetchen) maar het nog onbewaarde werk in je submodule directory niet overschrijven.

```
$ git submodule update --remote
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 4 (delta 0)
Unpacking objects: 100% (4/4), done.
From https://github.com/chaconinc/DbConnector
 5d60ef9..c75e92a stable      -> origin/stable
error: Your local changes to the following files would be overwritten by checkout:
  scripts/setup.sh
Please, commit your changes or stash them before you can switch branches.
Aborting
Unable to checkout 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

Als je wijzigingen hebt gemaakt die conflicteren met wijzigingen die stroomopwaarts zijn gemaakt, zal Git je dit laten weten als je de update uitvoert.

```
$ git submodule update --remote --merge
Auto-merging scripts/setup.sh
CONFLICT (content): Merge conflict in scripts/setup.sh
Recorded preimage for 'scripts/setup.sh'
Automatic merge failed; fix conflicts and then commit the result.
Unable to merge 'c75e92a2b3855c9e5b66f915308390d9db204aca' in submodule path
'DbConnector'
```

Je kunt in de directory van de submodule gaan en de conflicten oplossen op dezelfde manier zoals je anders ook zou doen.

### Submodule wijzigingen publiceren

We hebben nu een aantal wijzigingen in onze submodule directory. Sommige van deze zijn van stroomopwaarts binnengekomen via onze updates en andere zijn lokaal gemaakt en zijn nog voor niemand anders beschikbaar omdat we ze nog niet hebben gepusht.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
  > Merge from origin/stable
  > updated setup script
  > unicode support
  > remove unnecessary method
  > add new option for conn pooling
```

Als we het hoofdproject committen en deze pushen zonder de submodule wijzigingen ook te pushen, zullen andere mensen die willen zien wat onze wijzigingen inhouden problemen krijgen omdat er geen enkele manier is voor hen om de wijzigingen van de submodule te pakken krijgen waar toch op wordt voortgebouwd. Deze wijzigingen zullen alleen in onze lokale kopie bestaan.

Om er zeker van te zijn dat dit niet gebeurt, kan je Git vragen om te controleren dat al je submodules juist gepusht zijn voordat het hoofdproject wordt gepusht. Het `git push` commando leest het `--recurse-submodules` argument die op de waardes “check” of “on-demand” kan worden gezet. De “check” optie laat een `push` eenvoudigweg falen als een van de gecommitte submodule wijzigingen niet is gepusht.

```
$ git push --recurse-submodules=check  
The following submodule paths contain changes that can  
not be found on any remote:  
  DbConnector
```

Please try

```
git push --recurse-submodules=on-demand
```

or cd to the path and use

```
git push
```

to push them to a remote.

Zoals je kunt zien, geeft het ook wat behulpzame adviezen over wat he vervolgens kunnen doen. De eenvoudige optie is om naar elke submodule te gaan en handmatig naar de remotes te pushen om er zeker van te zijn dat ze extern beschikbaar zijn en dan deze push nogmaals te proberen.

De andere optie is om de “on-demand” waarde te gebruiken, wat zal proberen dit voor je te doen.

```
$ git push --recurse-submodules=on-demand  
Pushing submodule 'DbConnector'  
Counting objects: 9, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (8/8), done.  
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.  
Total 9 (delta 3), reused 0 (delta 0)  
To https://github.com/chaconinc/DbConnector  
  c75e92a..82d2ad3 stable -> stable  
Counting objects: 2, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (2/2), done.  
Writing objects: 100% (2/2), 266 bytes | 0 bytes/s, done.  
Total 2 (delta 1), reused 0 (delta 0)  
To https://github.com/chaconinc/MainProject  
  3d6d338..9a377d1 master -> master
```

Zoals je hier kunt zien, ging Git in de DbConnector module en heeft deze gepusht voordat het hoofdproject werd gepusht. Als die push van de submodule om wat voor reden ook faalt, zal de push van het hoofdproject ook falen. Je kunt dit gedrag de standaard maken door `git config push.recurseSubmodules on-demand` te doen.

## Submodule wijzigingen mergen

Als je een submodule-referentie wijzigt op hetzelfde moment als een ander, kan je in enkele problemen geraken. In die zin, dat wanneer submodule histories uit elkaar zijn gaan lopen en naar uit elkaar lopende branches in het superproject worden gecommit, zal het wat extra werk van je

vergen om dit te repareren.

Als een van de commits een directe voorouder is van de ander (een fast-forward merge), dan zal git eenvoudigweg de laatste voor de merge kiezen, dus dat werkt prima.

Git zal echter niet eens een triviale merge voor je proberen. Als de submodule commits uiteen zijn gaan lopen en ze moeten worden gemerged, zal je iets krijgen wat hier op lijkt:

```
$ git pull
remote: Counting objects: 2, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 1), reused 2 (delta 1)
Unpacking objects: 100% (2/2), done.
From https://github.com/chaconinc/MainProject
  9a377d1..eb974f8  master      -> origin/master
Fetching submodule DbConnector
warning: Failed to merge submodule DbConnector (merge following commits not found)
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Dus wat er hier eigenlijk gebeurd is, is dat Git heeft achterhaald dat de twee branches punten in de historie van de submodule hebben opgeslagen die uiteen zijn gaan lopen en die gemerged moeten worden. Het legt dit uit als “merge following commits not found” (merge volgend op commits niet gevonden), wat verwarring kan geven, maar we leggen zo uit waarom dit zo is.

Om dit probleem op te lossen, moet je uit zien te vinden in welke staat de submodule in zou moeten zijn. Vreemdgenoeg geeft Git je niet echt veel informatie om je hiermee te helpen, niet eens de SHA-1 getallen van de commits van beide kanten van de historie. Gelukkig is het redelijk eenvoudig om uit te vinden. Als je `git diff` aanroeft kan je de SHA-1 getallen van de opgeslagen commits krijgen uit beide branches die je probeerde te mergen.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

Dus in dit geval, is `eb41d76` de commit in onze submodule die **wij** hebben en `c771610` is de commit die stroomopwaarts aanwezig is. Als we naar onze submodule directory gaan, moet het al aanwezig zin op `eb41d76` omdat de merge deze nog niet zal hebben aangeraakt. Als deze om welke reden dan ook er niet is, kan je eenvoudigweg een branch die hiernaar wijst aanmaken en uit checken.

Wat nu een belangrijke rol gaat spelen is de SHA-1 van de commit van de andere kant. Dit is wat je in zult moeten mergen en oplossen. Je kunt ofwel de merge met de SHA-1 gewoon proberen, of je kunt een branch hiervoor maken en dan deze proberen te mergen. We raden het laatste aan, al was het maar om een mooiere merge commit bericht te krijgen.

Dus, we gaan naar onze submodule directory, maken een branch gebaseerd op die tweede SHA-1 van `git diff` en mergen handmatig.

```
$ cd DbConnector

$ git rev-parse HEAD
eb41d764bccf88be77aced643c13a7fa86714135

$ git branch try-merge c771610
(DbConnector) $ git merge try-merge
Auto-merging src/main.c
CONFLICT (content): Merge conflict in src/main.c
Recorded preimage for 'src/main.c'
Automatic merge failed; fix conflicts and then commit the result.
```

We hebben een echte merge conflict, dus als we deze oplossen en committen, dan kunnen we eenvoudigweg het hoofdproject updaten met het resultaat.

```
$ vim src/main.c ①
$ git add src/main.c
$ git commit -am 'merged our changes'
Recorded resolution for 'src/main.c'.
[master 9fd905e] merged our changes

$ cd .. ②
$ git diff ③
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
@@@ -1,1 -1,1 +1,1 @@@
- Subproject commit eb41d764bccf88be77aced643c13a7fa86714135
- Subproject commit c77161012afbbe1f58b5053316ead08f4b7e6d1d
++Subproject commit 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
$ git add DbConnector ④

$ git commit -m "Merge Tom's Changes" ⑤
[master 10d2c60] Merge Tom's Changes
```

① Eerst lossen we het conflict op

② Dan gaan we terug naar de directory van het hoofdproject

③ We controleren de SHA-1 getallen nog een keer

④ Lossen de conflicterende submodule entry op

⑤ Committen onze merge.

Dit kan nogal verwarring overkomen, maar het is niet echt moeilijk.

Interessant genoeg, is er een ander geval die Git aankan. Als er een merge commit bestaat in de directory van de submodule die **beide** commits in z'n historie bevat, zal Git je deze voorstellen als mogelijke oplossing. Het ziet dat op een bepaald punt in het submodule project iemand branches heeft gemerged met daarin deze twee commits, dus wellicht wil je die hebben.

Dit is waarom de foutbericht van eerder “merge following commits not found” was, omdat het **dit** niet kon doen. Het is verwarrend omdat, wie verwacht er nu dat Git dit zou **proberen**?

Als het een enkele acceptabele merge commit vindt, zal je iets als dit zien:

```
$ git merge origin/master
warning: Failed to merge submodule DbConnector (not fast-forward)
Found a possible merge resolution for the submodule:
  9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a: > merged our changes
If this is correct simply add it to the index for example
by using:

  git update-index --cacheinfo 160000 9fd905e5d7f45a0d4cbc43d1ee550f16a30e825a
  "DbConnector"

which will accept this suggestion.
Auto-merging DbConnector
CONFLICT (submodule): Merge conflict in DbConnector
Automatic merge failed; fix conflicts and then commit the result.
```

Wat hier gesuggereerd wordt om te doen is om de index te updaten alsof je **git add** zou hebben aangeroepen, wat het conflict opruimt, en dan commit. Echter, je moet dit waarschijnlijk niet doen. Je kunt net zo makkelijk naar de directory van de submodule gaan, kijken wat het verschil is, naar deze commit fast-forwarden, het goed testen en daarna committen.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

Dit bereikt hetzelfde, maar op deze manier kan je verifiëren dat het werkt en je hebt de code in je submodule als je klaar bent.

## Submodule Tips

Er zijn een aantal dingen die je kunt doen om het werken met submodules iets eenvoudiger te maken.

## Submodule Foreach

Er is een **foreach** submodule commando om een willekeurig commando aan te roepen in elke submodule. Dit kan echt handig zijn als je een aantal submodules in hetzelfde project hebt.

Bijvoorbeeld, stel dat we een nieuwe functie willen beginnen te maken of een bugfix uitvoeren en we hebben werkzaamheden in verscheidene submodules onderhanden. We kunnen eenvoudig al het werk in al onze submodules stashen.

```
$ git submodule foreach 'git stash'  
Entering 'CryptoLibrary'  
No local changes to save  
Entering 'DbConnector'  
Saved working directory and index state WIP on stable: 82d2ad3 Merge from  
origin/stable  
HEAD is now at 82d2ad3 Merge from origin/stable
```

Daarna kunnen we een nieuwe branch maken en ernaar switchen in al onze submodules.

```
$ git submodule foreach 'git checkout -b featureA'  
Entering 'CryptoLibrary'  
Switched to a new branch 'featureA'  
Entering 'DbConnector'  
Switched to a new branch 'featureA'
```

Je ziet waar het naartoe gaat. Een heel nuttig ding wat je kunt doen is een mooie unified diff maken van wat er gewijzigd is in je hoofdproject alsmede al je subprojecten.

```

$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

    commit_page_choice();

+    url = url_decode(url_orig);
+
     /* build alias_argv */
     alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
     alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
    return url_decode_internal(&url, len, NULL, &out, 0);
}

+char *url_decode(const char *url)
+{
+    return url_decode_mem(url, strlen(url));
+}
+
char *url_decode_parameter_name(const char **query)
{
    struct strbuf out = STRBUF_INIT;

```

Hier kunnen we zien dat we een functie aan het definieren zijn in een submodule en dat we het in het hoofdproject aanroepen. Dit is overduidelijk een versimpeld voorbeeld, maar hopelijk geeft het je een idee van hoe dit handig kan zijn.

## Bruikbare aliassen

Je wilt misschien een aantal aliassen maken voor een aantal van deze commando's omdat ze redelijk lang kunnen zijn en je geen configuratie opties voor de meeste van deze kunt instellen om ze standaard te maken. We hebben het opzetten van Git aliassen in [Git aliassen](#) behandeld, maar hier is een voorbeeld van iets wat je misschien zou kunnen opzetten als je van plan bent veel met submodules in Git te werken.

```

$ git config alias.sdiff '!"git diff && git submodule foreach \'git diff\'"
$ git config alias.push 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'

```

Op deze manier kan je eenvoudig `git submodule update` aanroepen als je je submodules wilt updaten, of `git submodule push` om te pushen met controle op afhankelijkheden op de submodule.

## Problemen met submodules

Submodules gebruiken is echter niet zonder nukken.

Bijvoorbeeld het switchen van branches met daarin submodulen kan nogal listig zijn. Als je een nieuwe branch maakt, daar een submodule toevoegt, en dan terug switcht naar een branch zonder die submodule, heb je de submodule directory nog steeds als een untrackt directory.

```
$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    CryptoLibrary/

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

Die directory weghalen is niet moeilijk maar het kan nogal verwarringd zijn om hem daar te hebben. Als je het weghaalt en dan weer terug switcht naar de branch die deze submodule heeft, zal je `submodule update --init` moeten aanroepen om het weer te vullen.

```
$ git clean -ffdx
Removing CryptoLibrary/

$ git checkout add-crypto
Switched to branch 'add-crypto'

$ ls CryptoLibrary/
$ git submodule update --init
Submodule path 'CryptoLibrary': checked out 'b8dda6aa182ea4464f3f3264b11e0268545172af'

$ ls CryptoLibrary/
Makefile      includes      scripts      src
```

Alweer, niet echt moeilijk, maar het kan wat verwarring scheppen.

Het andere grote probleem waar veel mensen tegenaan lopen betreft het omschakelen van subdirectories naar submodules. Als je files aan het tracken bent in je project en je wilt ze naar een submodule verplaatsen, moet je voorzichtig zijn omdat Git anders erg boos op je gaat worden. Stel dat je bestanden hebt in een subdirectory van je project, en je wilt er een submodule van maken. Als je de subdirectory verwijdert en dan **submodule add** aanroeft, zal Git tegen je schreeuwen:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Je moet de **CryptoLibrary** directory eerst unstagen. Daarna kan je de submodule toevoegen:

```
$ git rm -r CryptoLibrary
$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

Stel je nu voor dat je dit in een branch zou doen. Als je naar een branch terug zou switchen waar deze bestanden nog steeds in de actuele tree staan in plaats van in een submodule - krijg je deze fout:

```
$ git checkout master
error: The following untracked working tree files would be overwritten by checkout:
CryptoLibrary/Makefile
CryptoLibrary/includes/crypto.h
...
Please move or remove them before you can switch branches.
Aborting
```

Je kunt forceeren om de switch te maken met `checkout -f`, maar wees voorzichtig dat je geen onbewaarde gegevens daar hebt staan omdat deze kunnen worden overschreven met dit commando.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Daarna, als je weer terug switcht, krijg je om de een of andere reden een lege `CryptoLibrary` directory en `git submodule update` zou hier ook geen oplossing voor kunnen bieden. Je zou misschien naar je submodule directory moeten gaan en een `git checkout .` aanroepen om al je bestanden terug te krijgen. Je zou dit in een `submodule foreach` script kunnen doen om het voor meerdere submodules uit te voeren.

Het is belangrijk om op te merken dat submodules tegenwoordig al hun Git data in de `.git` directory van het hoogste project opslaan, dus in tegenstelling tot oudere versies van Git, leidt het vernietigen van een submodule directory niet tot verlies van enig commit of branches die je had.

Met al deze gereedschappen, kunnen submodules een redelijk eenvoudig en effectieve manier zijn om een aantal gerelateerde maar toch aparte projecten tegelijk te ontwikkelen.

## Bundelen

Alhoewel we de reguliere manieren om Git data over een netwerk te transporteren al behandeld hebben (HTTP, SSH, etc), is er eigenlijk nog een weinig gebruikte manier om dit te doen, maar die wel erg nuttig kan zijn.

Git is in staat om zijn gegevens te “bundelen” (bundling) in een enkel bestand. Dit kan handig zijn in verscheidene situaties. Misschien is je netwerk uit de lucht en je wilt wijzigingen naar je medewerkers sturen. Misschien werk je ergens buiten de deur en heb je om beveiligingsredenen geen toegang tot het lokale netwerk. Misschien is je wireless/ethernet kaart gewoon kapot. Misschien heb je op dat moment geen toegang tot een gedeelde server, wil je iemand updates mailen en je wilt niet 40 commits via een `format-patch` sturen.

Dit is waar het `git bundle` commando behulpzaam kan zijn. Het `bundle` commando pakt alles wat normaalgesproken over het netwerk zou worden gepusht met een `git push` commando in een binair bestand die je naar iemand kunt mailen of op een flash drive kunt bewaren, en dan uitpakken in de andere repository.

Laten we een eenvoudig voorbeeld bekijken. Laten we zeggen dat je een repository met twee commits hebt:

```
$ git log
commit 9a466c572fe88b195efd356c3f2bbeccdb504102
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:10 2010 -0800

    second commit

commit b1ec3248f39900d2a406049d762aa68e9641be25
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Mar 10 07:34:01 2010 -0800

    first commit
```

Als je deze repository naar iemand wilt sturen en je hebt geen toegang tot een repository om naar te pushen, of deze gewoon niet wil inrichten, kan je het bundelen met `git bundle create`.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

Nu heb je een bestand die `repo.bundle` heet die alle gegevens heeft die nodig zijn om de `master`-branch van de repository weer op te bouwen. Met het `bundle` commando moet je elke referentie of een reeks van commits opgeven die je erin wilt betrekken. Als de bedoeling is dat deze elders wordt gekloond, moet je ook `HEAD` als referentie meenemen zoals we hier gedaan hebben.

Je kunt dit `repo.bundle` bestand naar iemand mailen, of op een USB schijf zetten en het even langsbrengen.

Aan de andere kant, stel dat je dit `repo.bundle` bestand gestuurd krijgt en je wilt aan het project werken. Je kunt dan van dit binaire bestand naar een directory klonen, vergelijkbaar met hoe je dit zou doen vanaf een URL.

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

Als je de `HEAD` niet in de referenties meeneemt, moet je ook `-b master` opgeven of welke branch er

dan ook in zit, omdat het anders niet duidelijk is welke branch er moet worden uitgechecked.

Laten we nu zeggen dat je drie commits hierop doet en de nieuwe commits terug wilt sturen via een bundel op een USB stick of e-mail.

```
$ git log --oneline  
71b84da last commit - second repo  
c99cf5b fourth commit - second repo  
7011d3d third commit - second repo  
9a466c5 second commit  
b1ec324 first commit
```

Eerst moeten we de reeks van commits vaststellen die we in de bundel willen stoppen. In tegenstelling tot de netwerk protocollen die de minimum set van gegevens die verstuurd moeten worden voor ons kunnen bepalen, moeten we het hier handmatig uitvinden. Je kunt natuurlijk hier hetzelfde doen en de gehele repository bundelen, en dat zou werken, maar het is beter om alleen het verschil te bundelen - alleen de drie commits die we zojuist lokaal gemaakt hebben.

Om dat te doen, moet je het verschil berekenen. Zoals we hebben beschreven in [Commit reeksen](#), kan je op verschillende manieren een reeks van commits aangeven. Om de drie commits te krijgen die we in onze master branch hebben die niet in de originele gekloonde branch zaten, kunnen we zo iets als `origin/master..master` of `master ^origin/master` gebruiken. Je kunt dat verifiëren met het `log` commando.

```
$ git log --oneline master ^origin/master  
71b84da last commit - second repo  
c99cf5b fourth commit - second repo  
7011d3d third commit - second repo
```

Dus nu dat we de lijst met commits hebben die we in de bundel willen pakken, laten we ze dan ook gaan bundelen. We doen dat met het `git bundle create` commando, waaraan we een bestandsnaam meegeven waar we onze bundel in willen pakken en de reeks met commits die we erin willen gaan doen.

```
$ git bundle create commits.bundle master ^9a466c5  
Counting objects: 11, done.  
Delta compression using up to 2 threads.  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (9/9), 775 bytes, done.  
Total 9 (delta 0), reused 0 (delta 0)
```

Nu hebben we een `commits.bundle` bestand in onze directory. Als we deze naar onze partner sturen, kan zij deze importeren in de originele repository, zelfs als daar in de tussentijd weer meer werk aan gedaan is.

Als ze de bundel krijgt, kan ze deze inspecteren om te zien wat erin zit voordat ze deze in haar repository importeert. Het eerste commando is het `bundle verify` commando, dat controleert of het

bestand een geldige Git bundel is en dat je alle benodigde voorouders hebt om het op de juiste wijze te importeren.

```
$ git bundle verify ../commits.bundle
The bundle contains 1 ref
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
The bundle requires these 1 ref
9a466c572fe88b195efd356c3f2bbeccdb504102 second commit
../commits.bundle is okay
```

Als degene die de bundel heeft aangemaakt van alleen de laatste twee commits die ze hadden gedaan, in plaats van alle drie, zou de originele repository niet in staat zijn geweest om deze te importeren, omdat de benodigde historie ontbreekt. Het `verify` commando zou dan iets als dit hebben laten zien:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

Echter, onze eerste bundel is geldig, dus we kunnen de commits ervan gaan fetchen. Als je zou willen zien welke branches er uit de bundel kunnen worden geïmporteerd, is er ook een commando die alleen de heads laat zien:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

Het `verify` sub-commando laat je ook de heads zien. Het belangrijkste is om te zien wat er naar binnen gepulld kan worden, zodat je het `fetch` of `pull` commando kunt gebruiken om de commits van deze bundel kunt importeren. Hier gaan we de *master* branch van de bundel naar een branch met de naam *other-master* in onze repository fetchen:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master    -> other-master
```

Nu kunnen we zien dat we de commits op de *other-master*-branch hebben geïmporteerd zowel als elke andere commit die we in de tussentijd in onze eigen *master*-branch hebben gedaan.

```
$ git log --oneline --decorate --graph --all
* 8255d41 (HEAD, master) third commit - first repo
| * 71b84da (other-master) last commit - second repo
| * c99cf5b fourth commit - second repo
| * 7011d3d third commit - second repo
|/
* 9a466c5 second commit
* b1ec324 first commit
```

Dus `git bundle` kan erg handig zijn voor het delen of netwerk-achtige operaties te doen als je niet de beschikking hebt over een geschikt netwerk of gedeelde repository om te gebruiken.

## Vervangen

Zoals we eerder hebben benadrukt zijn de objecten in de database van Git onwijzigbaar, maar Git heeft een interessante manier om te *doen alsof* je objecten in de database vervangt met andere objecten.

Het `replace` commando laat je een object in Git opgeven en te zeggen dat "elke keer als je *dit* object ziet, doe *alsof* het dit een *ander* object is". Dit is het nuttigst voor het vervangen van een commit in je historie met een andere zonder de gehele historie te vervangen met, laten we zeggen, `git filter-branch`.

Bijvoorbeeld, stel dat je een enorme code historie hebt en je wilt je repository opsplitsen in een korte historie voor nieuwe ontwikkelaars en een veel langere en grotere historie voor mensen die geïnteresseerd zijn in het graven in gegevens (data mining). Je kunt de ene historie op de andere enten door de vroegste commit in de nieuwe lijn met de laatste commit van de oude lijn te "vervangen". Dit is prettig omdat het betekent dat je niet echt alle commits in de nieuwe historie hoeft te herschrijven, wat je normaalgesproken wel zou moeten doen om ze samen te voegen (omdat de voorouderschap de SHA-1's beïnvloedt).

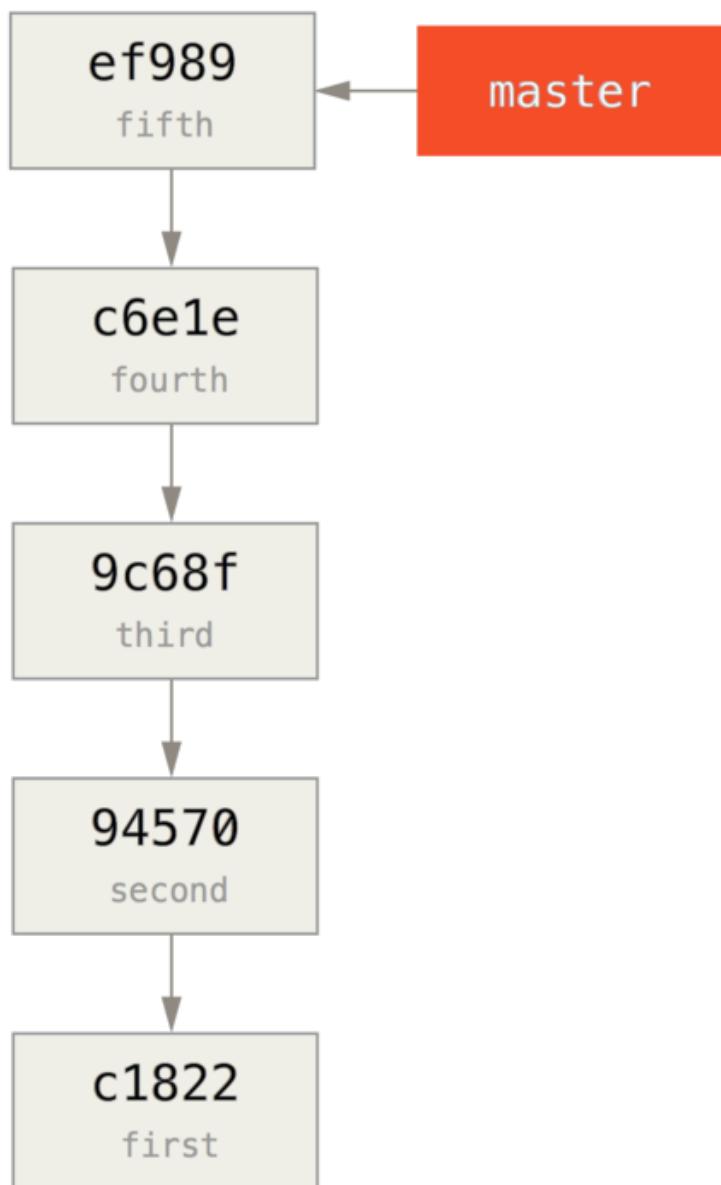
Laten we dat eens uitproberen. Laten we een bestaande repository nemen, en deze in twee repositories splitsen, een recente en een historische, en laten we dan kijken hoe we ze kunnen herschikken zonder de SHA-1 waarden van de recente repository te wijzigen met behulp van `replace`.

We zullen een eenvoudige repository met vijf simpele commits gebruiken:

```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

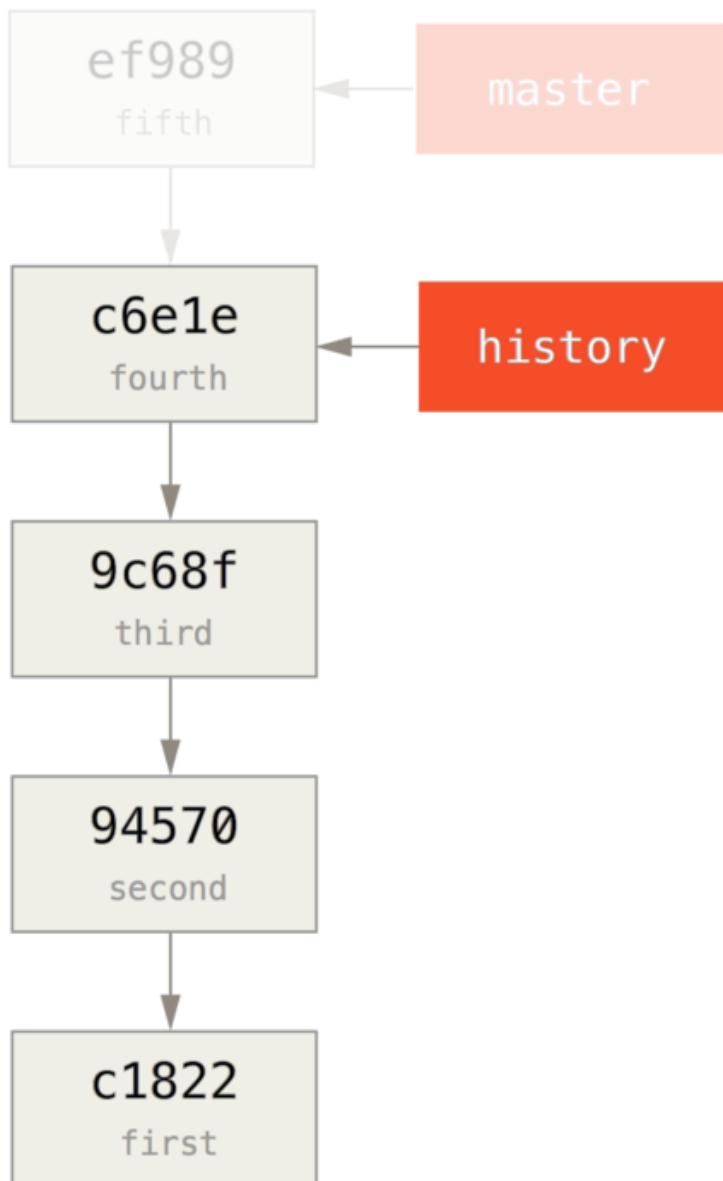
We willen deze opdelen in twee historische lijnen. Een lijn gaat van commit een tot commit vier - dat zal de historische worden. De tweede lijn zal alleen commits vier en vijf zijn - dat is dan de

recente historie.



Nu, de historische historie maken is eenvoudig, we kunnen gewoon een branch in de geschiedenis zetten en dan die branch naar de master branch pushen van een nieuwe remote repository.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```



Nu kunnen we de nieuwe `history`-branch naar de `master`-branch van onze nieuwe repository pushen:

```

$ git remote add project-history https://github.com/schacon/project-history
$ git push project-history history:master
Counting objects: 12, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (12/12), 907 bytes, done.
Total 12 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (12/12), done.
To git@github.com:schacon/project-history.git
 * [new branch]      history -> master

```

Goed, onze historie is nu gepubliceerd. Nu is het moeilijkere gedeelte het terugsnoeien van onze recente historie zodat deze kleiner wordt. We moeten een overlapping maken op zo'n manier dat we een commit kunnen vervangen in een repository die een gelijke commit heeft, dus we gaan deze afkappen tot alleen commits vier en vijf (dus de vierde commit overlapt).

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

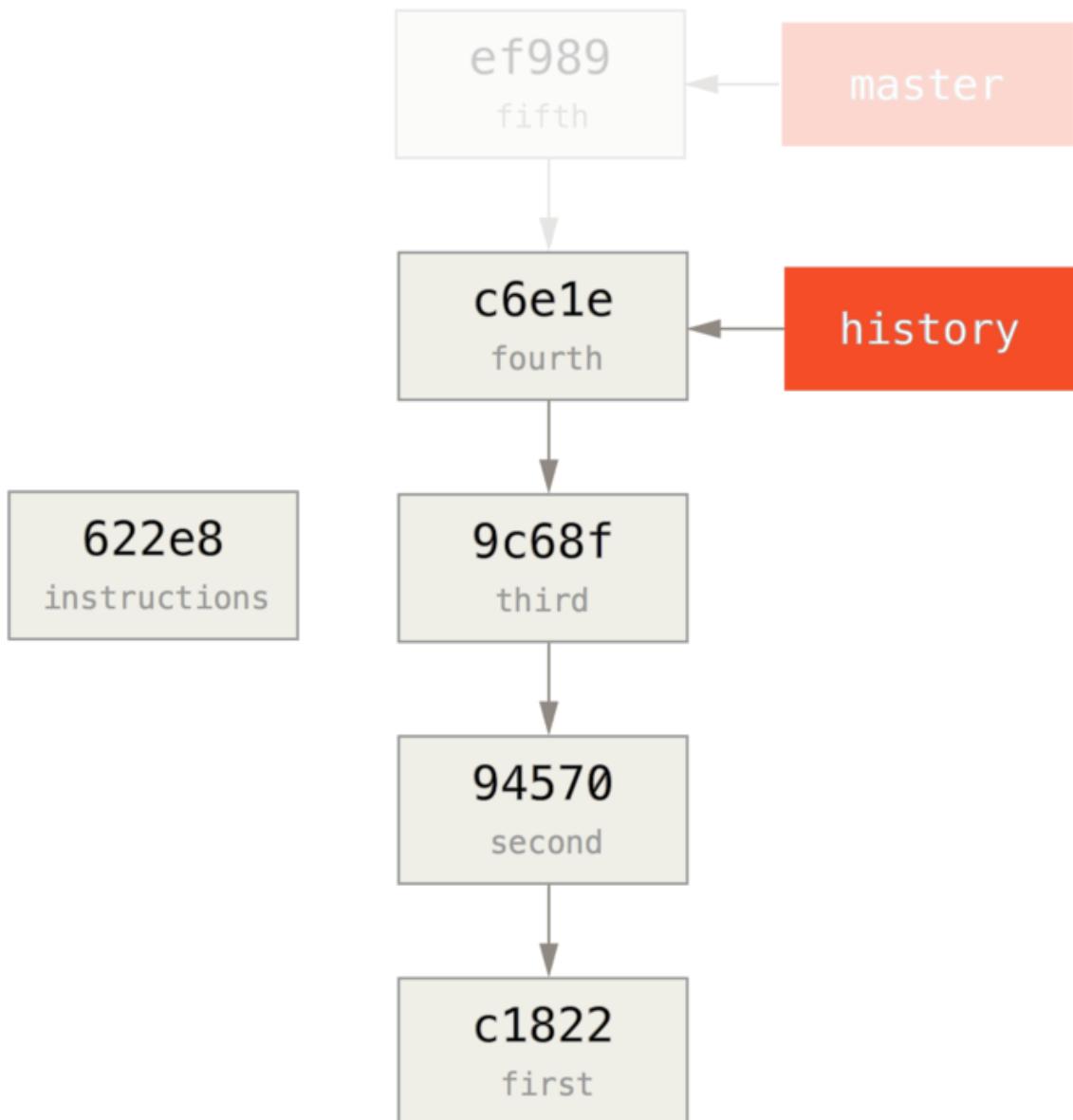
Het is in dit geval handig om een basis commit te maken die instructies bevat hoe de historie uit te breiden, zodat andere ontwikkelaars weten wat te doen als ze de eerste commit in de afgekapselde historie tegenkomen en meer nodig hebben. Dus wat we hier gaan doen is een initieel commit object maken als onze basis en daar instructies in zetten, dan rebasen we de overige commits (vier en vijf) daar bovenop.

Om dat te doen, moeten we een punt kiezen om af te splitsen, wat voor ons de derde commit is, welke **9c68fdc** in SHA-spraak is. Dus onze basis commit zal van die tree af worden getakt. We kunnen onze basis commit maken met het **commit-tree** commando, wat gewoon een tree neemt en ons een SHA-1 teuggeeft van een gloednieuw, ouderloos commit object.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfa10cf
```

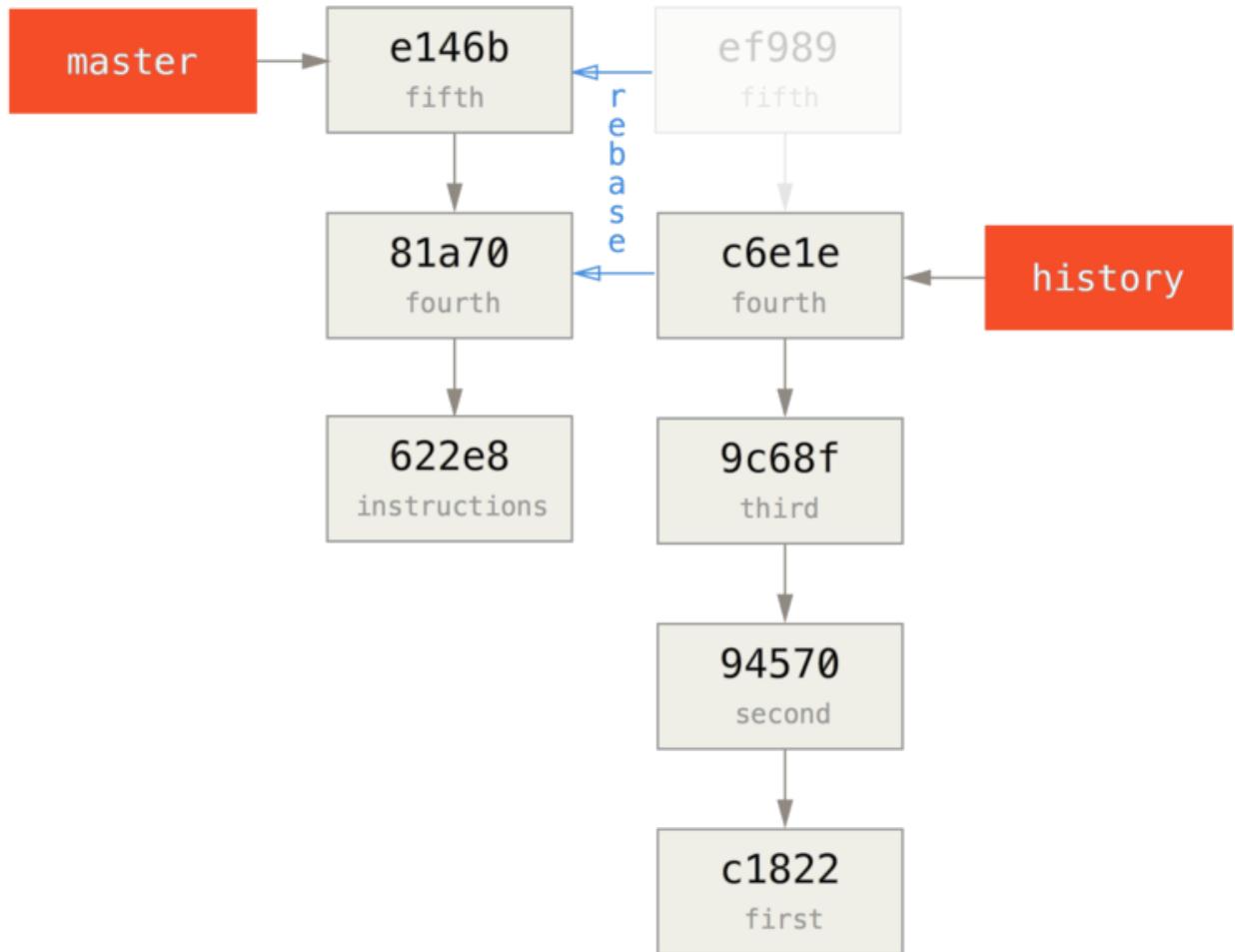


Het **commit-tree** commando is een uit de reeks van commando's die gewoonlijk **binnenwerk** (plumbing) commando's worden genoemd. Dit zijn commando's die niet direct voor normaal gebruik bedoeld zijn, maar die in plaats daarvan door **andere** Git commando's worden gebruikt om kleinere taken uit te voeren. Bij tijd en wijle, als we wat vreemdere zaken dan dit uitvoeren, stellen ze ons in staat om echt *lage* dingen uit te voeren maar ze zijn niet bedoeld voor dagelijks gebruik. Je kunt meer over deze plumbing commando's lezen in [Binnenwerk en koetswerk \(plumbing and porcelain\)](#).



Goed, nu we dus een basis commit hebben, kunnen we de rest van onze historie hier boven op rebasen met `git rebase --onto`. Het `--onto` argument zal de SHA-1 zijn die we zojuist terugkregen van `commit-tree` en het rebase punt zal de derde commit zijn (de ouder van de eerste commit die we willen bewaren: `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```



Mooi, dus we hebben onze recente historie herschreven bovenop een weggooi basis commit die nu onze instructies bevat hoe de gehele historie weer te herbouwen als we dat zouden willen. We kunnen die nieuwe historie op een nieuw project pushen en nu, als mensen die repository klonen, zullen ze alleen de meest recente twee commits zien en dan een basis commit met instructies.

Laten we de rollen nu omdraaien naar iemand die het project voor het eerst kloont en die de hele historie wil hebben. Om de historische gegevens na het klonen van deze gesnoeide repository te krijgen, moet je een tweede remote toevoegen voor de historische repository en dan fetchen:

```
$ git clone https://github.com/schacon/project
$ cd project

$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git remote add project-history https://github.com/schacon/project-history
$ git fetch project-history
From https://github.com/schacon/project-history
 * [new branch]      master      -> project-history/master
```

Nu zal de medewerker hun recente commits in de `master`-branch hebben en de historische commits in de `project-history/master`-branch.

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

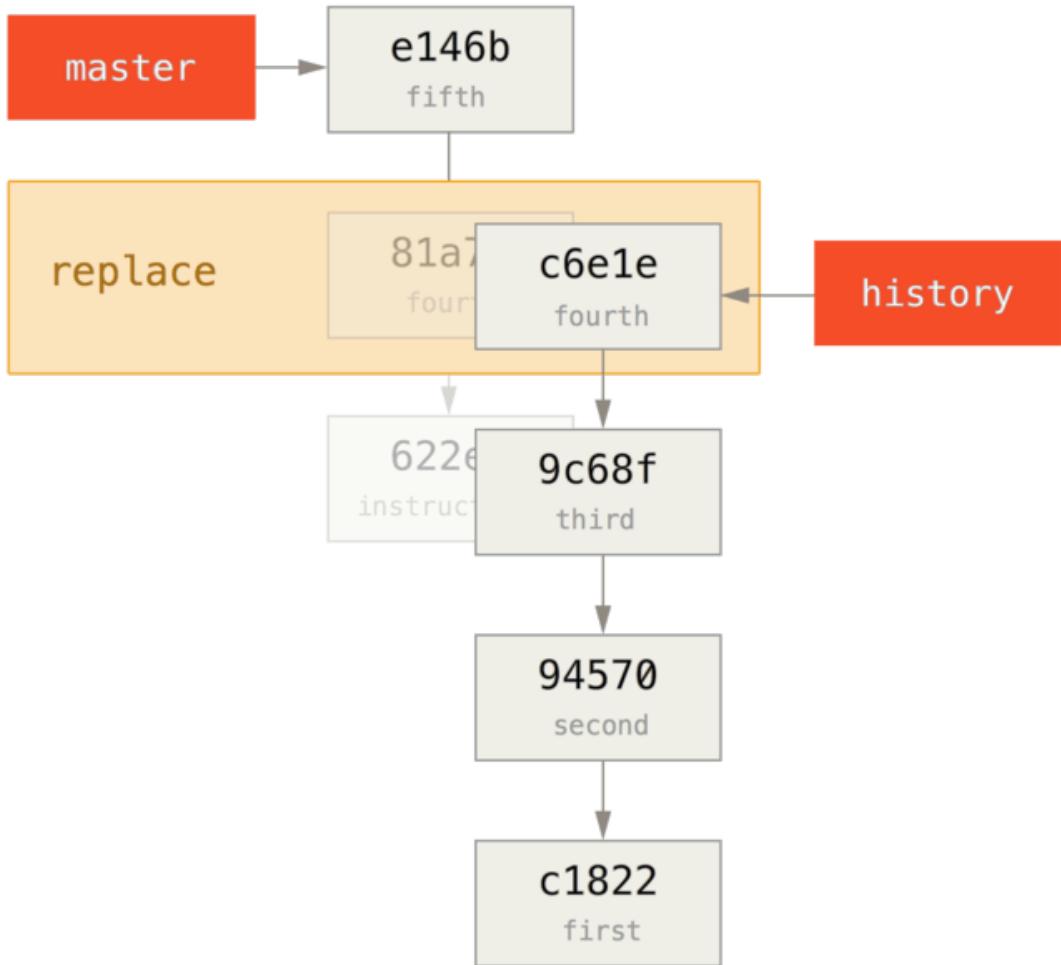
Om deze te combineren, kan je simpelweg `git replace` aanroepen met de commit die je wilt vervangen en dan de commit waarmee je het wilt vervangen. Dus we willen de "fourth" commit in de `master` branch met de "fourth" commit in de `project-history/master`-branch vervangen:

```
$ git replace 81a708d c6e1e95
```

Als je nu naar de historie van de `master`-branch kijkt, lijkt het er zo uit te zien:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Gaaf, toch? Zonder alle SHA-1 stroomopwaarts te hoeven vervangen, waren we toch in staat om een commit in onze history te vervangen met een compleet andere commit en alle normale instrumenten (`bisect`, `blame`, etc.) blijven werken zoals we van ze mogen verwachten.



Interessant genoeg, blijf het nog steeds **81a708d** als de SHA-1 laten zien, zelfs als het in werkelijkheid de gegevens van de **c6e1e95** commit gebruikt waar we het mee hebben vervangen. Zelfs als je een commando als **cat-file** aanroeft, zal het je de vervangen gegevens tonen:

```
$ git cat-file -p 81a708d
tree 7bc544cf438903b65ca9104a1e30345eeee6c083d
parent 9c68fdceee073230f19ebb8b5e7fc71b479c0252
author Scott Chacon <schacon@gmail.com> 1268712581 -0700
committer Scott Chacon <schacon@gmail.com> 1268712581 -0700

fourth commit
```

Onthoud dat de echte ouder van **81a708d** onze plaatsvervangende commit was (**622e88e**), niet **9c68fdce** zoals hier vermeld staat.

Het andere interessante is dat deze gegevens in onze referenties opgeslagen zijn:

```
$ git for-each-ref  
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/heads/master  
c6e1e95051d41771a649f3145423f8809d1a74d4 commit refs/remotes/history/master  
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/HEAD  
e146b5f14e79d4935160c0e83fb9ebe526b8da0d commit refs/remotes/origin/master  
c6e1e95051d41771a649f3145423f8809d1a74d4 commit  
refs/replace/81a708dd0e167a3f691541c7a6463343bc457040
```

Dit houdt in dat het eenvoudig is om onze vervanging met anderen te delen, omdat we deze naar onze server kunnen pushen en andere mensen het eenvoudig kunnen downloaden. Dit is niet zo nuttig in het scenario van historie-enten welke we hier nu behandeld hebben (als iedereen toch beide histories zou gaan downloaden, waarom zouden we ze dan gaan splitsen) maar het kan handig zijn in andere omstandigheden.

## Het opslaan van inloggegevens

Als je het SSH transport gebruikt om verbinding te maken met remotes, is het mogelijk om een sleutel te hebben zonder wachtwoord, wat je in staat stelt veilig gegevens uit te wisselen zonder je gebruikersnaam en wachtwoord in te typen. Dit is echter niet mogelijk met de HTTP protocollen - elke connectie heeft een gebruikersnaam en wachtwoord nodig. Het wordt zelfs lastiger voor systemen met twee-factor authenticatie, waar het token dat je gebruikt voor een wachtwoord willekeurig wordt gegenereerd en onuitspreekbaar is.

Gelukkig heeft Git een credentials systeem die ons daarbij kan helpen. Git heeft standaard een aantal opties in de aanbieding:

- De standaard is om helemaal niets op te slaan. Elke verbinding wordt je gevraagd om je gebruikersnaam en wachtwoord.
- De “cache” modus houdt deze gegevens voor een bepaalde tijd in het geheugen. Geen van de wachtwoorden worden ooit op schijf opgeslagen, en ze worden na 15 minuten uit de cache verwijderd.
- De “store” modus bewaart deze gegevens in bestand in een leesbaar tekstformaat, en ze verlopen nooit. Dit houdt in dat totdat je je wachtwoord wijzigt op de Git host, je nooit meer je gegevens hoeft in te typen. Het nadeel van deze aanpak is dat je wachtwoorden onversleuteld bewaard worden in een gewoon bestand in je home directory.
- Als je een Mac gebruikt, wordt Git geleverd met een “osxkeychain” modus, waarbij de gegevens opgeslagen worden in een beveiligde sleutelring die verbonden is aan je systeem account. Deze methode bewaart je gegevens op schijf, en ze verlopen nooit, maar ze zijn versleuteld met hetzelfde systeem dat de HTTPS certificaten en Safari auto-fills bewaart.
- Als je Windows gebruikt, kan je het hulpprogramma “Git Credential Manager for Windows” installeren. Dit is vergelijkbaar met het “osxkeychain” programma zoals hierboven beschreven, maar het gebruikt de Windows Credential Store om de gevoelige gegevens te beheren. Dit kan gevonden worden op <https://github.com/Microsoft/Git-Credential-Manager-for-Windows>.

Je kunt een van deze methoden kiezen door een Git configuratie waarde in te stellen:

```
$ git config --global credential.helper cache
```

Een aantal van deze hulpprogramma's hebben opties. De "store" helper accepteert een `--file <path>` argument, waarmee je kunt sturen waar het leesbare bestand wordt opgeslagen (standaard is `~/.git-credentials`). De "cache" helper accepteert de `--timeout <seconds>` optie, die de tijdsduur wijzigt gedurende welke de daemon blijft draaien (standaard is dit "900", ofwel 15 minuten). Hier is een voorbeeld van hoe je de "store" helper configureert met een eigen bestandsnaam:

```
$ git config --global credential.helper store --file ~/.my-credentials
```

Git staat het je zelfs toe om meerdere helpers te configureren. Als Git op zoek gaat naar inloggegevens voor een specifieke host, zal Git ze in volgorde afvragen, en stoppen als het eerste antwoord wordt gegeven. Bij het opslaan van de gegevens, zal Git de gebruikersnaam en wachtwoord naar **alle** helpers in de lijst sturen, en zij kunnen besluiten wat met deze gegevens te doen. Hier is hoe een `.gitconfig` eruit zou kunnen zien als je een credentials bestand op een stickie zou hebben staan, maar de opslag in het geheugen zou willen gebruiken om wat typen te besparen als die stick niet ingeplugged is:

```
[credential]
helper = store --file /mnt/thumbdrive/.git-credentials
helper = cache --timeout 30000
```

## Onder de motorkap

Hoe werkt dit nu allemaal? Het basiscommando van Git voor het credential-helper systeem is `git credential`, wat een commando als argument neemt, en daarna meer invoer vanuit stdin.

Dit is misschien beter te begrijpen met een voorbeeld. Laten we zeggen dat een credential helper geconfigureerd is, en de helper heeft gegevens bewaard voor `mygithost`. Hier is een sessie die het "fill" commando gebruikt, wat wordt aangeroepen als Git probeert inloggegevens te vinden voor een host:

```

$ git credential fill ①
protocol=https ②
host=mygithost
③
protocol=https ④
host=mygithost
username=bob
password=s3cre7
$ git credential fill ⑤
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7

```

- ① Dit is de commando regel die de interactie initieert.
- ② Git-credential gaat dan wachten op invoer van stdin. We geven het de dingen die we weten: het protocol en de hostnaam.
- ③ Een blanco regel geeft aan dat de invoer compleet is, en het credential systeem moet nu antwoorden met wat het weet.
- ④ Git-credential neemt het daarna over, en schrijft de stukken informatie het gevonden heeft naar stdout.
- ⑤ Als er geen inloggegevens gevonden zijn, vraag Git de gebruiker om de gebruikersnaam en wachtwoord en stelt die ter beschikking aan de stdout van de aanroeper (hier zitten ze verbonden met dezelfde console).

Het credential systeem roept feitelijk een programma aan dat los staat van Git zelf; welke dat is en hoe hangt af van de waarde die is ingevuld bij `credential.helper`. Deze kan verschillende vormen aannemen:

| Configuratie waarde                              | Gedrag   |
|--|--|
| <code>foo</code>                                 | Roept <code>git-credential-foo</code> aan              |
| <code>foo -a --opt=bcd</code>                    | Roept <code>git-credential-foo -a --opt=bcd</code> aan |
| <code>/absolute/path/foo -xyz</code>             | Roept <code>/absolute/path/foo -xyz</code> aan         |
| <code>!f() { echo "password=s3cre7"; }; f</code> | Code na <code>!</code> wordt in shell geëvalueerd      |

Dus de helpers die hierboven zijn beschreven heten eigenlijk `git-credential-cache`, `git-credential-store`, en zo voorts, en we kunnen ze configureren om commando-regel argumenten te accepteren. De algemene vorm voor dit is “`git-credential-foo [args] <actie>`.” Het stdin/stdout protocol is dezelfde als git-credential, maar deze gebruiken een iets andere set acties:

- `get` is een verzoek voor een gebruikersnaam/wachtwoord paar.

- `store` is een verzoek om een groep van inloggegevens in het geheugen van de helper op te slaan.
- `erase` verwijder de inloggegevens voor de opgegeven kenmerken uit het geheugen van deze helper.

Voor de `store` en `erase` acties, is geen antwoord nodig (Git negeert deze in elk geval). Voor de `get` actie echter is Git zeer geïnteresseerd in het antwoord van de helper. Als de helper geen zinnig antwoord kan geven, kan het simpelweg stoppen zonder uitvoer, maar als het wel een antwoord heeft, moet het de gegeven informatie aanvullen met de gegevens die het heeft opgeslagen. De uitvoer wordt behandeld als een reeks van toewijzings-opdrachten; alles wat wordt aangereikt zal wat Git hierover al weet vervangen.

Hier is hetzelfde voorbeeld als hierboven, maar `git-credential` wordt overgeslagen en er wordt direct naar `git-credential-store` gegaan:

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3cre7
```

① Hier vertellen we `git-credential-store` om wat inloggegevens te bewaren: de gebruikersnaam “bob” en het wachtwoord “s3cre7” moeten worden gebruikt als `https://mygithost` wordt benaderd.

② Nu gaan we deze inloggegevens ophalen. We geven de delen van de verbinding die we al weten (`https://mygithost`) en een lege regel.

③ De `git-credential-store` antwoordt met de gebruikersnaam en wachtwoord die we hierboven hebben opgeslagen.

Hier is hoe het `~/git.store` bestand eruit zal zien:

```
https://bob:s3cre7@mygithost
```

Het is niet meer dan een reeks regels, die elk een van inloggegevens voorziene URL bevat. De `osxkeychain` en `wincred` helpers gebruiken het eigen formaat van hun eigen achterliggende opslag, terwijl `cache` zijn eigen *in-memory* formaat gebruikt (wat geen enkel ander proces kan lezen).

## Een eigen inloggegevens cache

Gegeven dat `git-credential-store` en zijn vriendjes programma's zijn die los staan van Git, is het geen grote stap om te beseffen dat *elk* programma een Git credential helper kan zijn. De helpers die bij Git worden geleverd dekken veel gewone gebruikssituaties, maar niet alle. Bijvoorbeeld, stel nu

dat je team een aantal inloggegevens hebben die met het hele team worden gedeeld, misschien om te deployen. Deze worden opgeslagen in een gedeelde directory, maar je wilt ze niet naar je eigen credential opslagplaats kopiëren, omdat ze vaak veranderen. Geen van de bestaande helpers kan hierin voorzien; laten we eens kijken hoeveel moeite het kost om er zelf een te schrijven. Er zijn een aantal sleutelkenmerken die dit programma moet hebben:

1. De enige actie waar we aandacht aan moeten besteden is `get`; `store` en `erase` zijn schrijf-acties, dus we negeren deze en sluiten gewoon af als ze worden ontvangen.
2. Het bestandsformaat van het gedeelde credential bestand is dezelfde als die wordt gebruikt door `git-credential-store`.
3. De locatie van dat bestand is redelijk standaard, maar we moeten toestaan dat de gebruiker een aangepast pad door geeft, voor het geval dat.

Nogmaals, we schrijven deze extensie in Ruby, maar een andere willekeurige taal werkt ook, zolang Git het uiteindelijke product maar kan aanroepen. Hier is de volledige broncode van onze nieuwe credential helper:

```

#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ①
OptionParser.new do |opts|
  opts.banner = 'USAGE: git-credential-read-only [options] <action>'
  opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
    path = File.expand_path argpath
  end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ②
exit(0) unless File.exists? path

known = {} ③
while line = STDIN.gets
  break if line.strip == ''
  k,v = line.strip.split '=', 2
  known[k] = v
end

File.readlines(path).each do |fileline| ④
  prot,user,pass,host = fileline.scan(/^\:(.*?):(.*)@(.*)$/).first
  if prot == known['protocol'] and host == known['host'] and user == known['username'] then
    puts "protocol=#{prot}"
    puts "host=#{host}"
    puts "username=#{user}"
    puts "password=#{pass}"
    exit(0)
  end
end

```

- ① Hier parsen we de commando-regel opties, waarbij we de gebruiker het invoerbestand kunnen laten aangeven. De standaardwaarde is `~/.git-credentials`.
- ② Dit programma geeft alleen antwoord als de actie `get` is, en het achterliggende bestand bestaat.
- ③ In deze lus wordt stdin gelezen tot de eerste blanco regel wordt bereikt. De invoergegevens worden opgeslagen in de `known` hash voor later gebruik.
- ④ In deze lus wordt de inhoud van het achterliggende bestand gelezen op zoek naar passende inhoud. Als het protocol en de host van `known` gelijk is aan deze regel, drukt het programma de resultaten af op stdout en stopt.

We zullen onze helper als `git-credential-read-only` opslaan, zetten het ergens in onze `PATH` en maken het uitvoerbaar. Hier is hoe een interactieve sessie eruit zou zien:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Omdat de naam met “git-” begint, kunnen we de eenvoudige syntax voor de configuratie waarde gebruiken:

```
$ git config --global credential.helper read-only --file /mnt/shared/creds
```

Zoals je kunt zien, is het uitbreiden van dit systeem redelijk eenvoudig, en we kunnen een aantal gebruikelijke problemen voor jou en je team oplossen.

## Samenvatting

Je hebt een aantal geavanceerde instrumenten gezien die je in staat stellen je commits en staging area met grotere precisie te manipuleren. Als je problemen ziet, zou je in staat moeten zijn om eenvoudig uit te vinden welke commit deze heeft geïntroduceerd, en wie dit gedaan heeft. Als je subprojecten wilt gebruiken in je project, heb je geleerd hoe deze een plaats te geven. Vanaf nu zou je in staat moeten zijn om de meeste dingen in Git te doen die je dagelijks op de command line nodig hebt en je er gemakkelijk bij voelen.

# Git aanpassen

Tot dusver hebben we de grondbeginselen behandeld van de werking van Git en hoe het te gebruiken, en we hebben een aantal instrumenten de revue laten passeren die Git je biedt om het eenvoudig en effectief te gebruiken. In dit hoofdstuk zullen we laten zien hoe je Git op een meer aangepaste manier kan laten werken, door een aantal belangrijke configuratie instellingen te laten zien en het hooks systeem. Met deze gereedschappjes is het eenvoudig om Git te laten werken op de manier waarop jij, je bedrijf of je groep het nodig heeft.

## Git configuratie

Zoals je al kort heb kunnen zien in [Aan de slag](#), kan je Git configuratie settings aangeven met het `git config` commando. Een van de eerste dingen die je hebt gedaan was het inrichten van je naam en e-mail adres:

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

Nu gaan we je een aantal van de meer interessante opties laten zien die je op deze manier kunt instellen om jouw Git gebruik aan te passen.

Eerst, een korte terugblik: Git gebruikt een reeks van configuratie bestanden om te bepalen welke niet standaard gedragingen je hebt aangevraagd. De eerste plek waar Git naar kijkt voor deze waarden is in een `/etc/gitconfig` bestand, welke de waarden bevat voor elke gebruiker op het systeem en al hun repositories. Als je de optie `--system` doorgeeft aan `git config`, leest en schrijft deze naar dit specifieke bestand.

De volgende plaats waar Git kijkt is het `~/.gitconfig` (of `~/.config/git/config`) bestand, die voor elke gebruiker anders kan zijn. Je kunt Git naar dit bestand laten lezen en schrijven door de `--global` optie door te geven.

Als laatste kijkt Git naar configuratie waarden in het configuratie bestand in de Git directory (`.git/config`) van de repository die je op dat moment gebruikt. Deze waarden zijn gebonden aan alleen die ene repository, en vertegenwoordigen het doorgeven van de `--local` optie aan `git config`. (Als je geen waarde doorgeeft voor het niveau, is dit de standaardwaarde.)

Elk van deze “levels” (systeem, globaal, lokaal) overtroeft de waarden van de vorige, dus waarden in ` .git/config` overtroeven die in `/etc/gitconfig` bijvoorbeeld.



De configuratie bestanden van Git zijn gewone tekstbestanden, dus je kunt deze waarden ook aanpassen door het bestand handmatig te bewerken en de juiste syntax te gebruiken. Het is echter over het algemeen eenvoudiger om het `git config` commando te gebruiken.

## Basis werkstation configuratie

De configuratie opties die door Git worden herkend vallen uiteen in twee categoriën: de kant van

het werkstation en die van de server. De meerderheid van de opties zitten aan de kant van het werkstation—welke jouw persoonlijke voorkeuren voor de werking inrichten. Er worden heel, *heel* erg veel configuratie opties ondersteund, maar een groot gedeelte van deze zijn alleen nuttig in bepaalde uitzonderingsgevallen. We zullen alleen de meest voorkomende en nuttigste hier behandelen. Als je een lijst wilt zien van alle opties die jouw versie van Git ondersteunt, kan je dit aanroepen:

```
$ man git-config
```

Dit commando geeft je een lijst van alle beschikbare opties met nogal wat detail informatie. Je kunt dit referentie-materiaal ook vinden op <http://git-scm.com/docs/git-config.html>.

### core.editor

Standaard gebruikt Git hetgeen je hebt ingesteld als je standaard tekstverwerker (`$VISUAL` of `$EDITOR`) en anders valt het terug op de `vi` tekstverwerker om jouw commit en tag berichten te maken en te onderhouden. Om deze standaard naar iets anders te verandereen, kan je de `core.editor` instelling gebruiken:

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

Nu maakt het niet meer uit wat je standaard shell editor is, Git zal Emacs opstarten om je berichten te wijzigen.

### commit.template

Als je dit instelt op het pad van een bestand op je systeem, zal Git dat bestand gebruiken als het standaard bericht als je commit. De waarde van het maken van een standaard bericht voorbeeld is dat je dit kan gebruiken om jezelf (of anderen) eraan kan herinneren om de juiste formattering en stijl te gebruiken voor het maken van een commit-bericht.

Bijvoorbeeld, stel dat je een sjabloon bestand op `~/.gitmessage.txt` hebt gemaakt dat er zo uitziet:

```
Subject line (try to keep under 50 characters)
```

```
Multi-line description of commit,  
feel free to be detailed.
```

```
[Ticket: X]
```

Zie hoe dit sjabloon de committer eraan herinnert om de onderwerpregel kort te houden (ten behoeve van de `git log --oneline`-uitvoer), verdere details eronder te vermelden en om naar een issue of bug-tracker systeem (mocht die bestaan) referentie te verwijzen.

Om Git te vertellen dat het dit als het standaard bericht moet gebruiken dat in je tekstverwerker verschijnt als je `git commit` aanroeft, zet dan de `commit.template` configuratie waarde:

```
$ git config --global commit.template ~/.gitmessage.txt  
$ git commit
```

Dan zal je tekstverwerker het volgende als je commit bericht sjabloon gebruiken als je gaat committen:

Subject line (try to keep under 60 characters)

Multi-line description of commit,  
feel free to be detailed.

[Ticket: X]

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~  
~  
.git/COMMIT_EDITMSG" 14L, 297C
```

Als je team een commit-bericht voorschrift heeft, dan zal het inrichten van een sjabloon voor dat voorschrift op jouw systeem en het inrichten van Git om dit standaard te gebruiken helpen met het vergroten van de kans dat dat voorschrift ook daadwerkelijk wordt opgevolgd.

### core.pager

Het instellen van deze waarde bepaalt welke pagineerhulp wordt gebruikt als Git uitvoer als **log** en **diff** gaan pagineren. Je kunt dit op **more** of jouw favoriete pagineerhulp instellen (standaard is het **less**), of je kunt het uitzetten door een lege waarde te geven:

```
$ git config --global core.pager ''
```

Als je dat aanroept, zal Git de gehele uitvoer van alle commando's tonen, onafhankelijk van de lengte van deze uitvoer.

### user.signingkey

Als je getekende geannoteerde tags aanmaakt (zoals behandeld in [Je werk tekenen](#)), zal het inrichten van je GPG handtekening als configuratie setting dingen eenvoudiger maken. Stel jouw sleutel ID als volgt in:

```
$ git config --global user.signingkey <gpg-key-id>
```

Nu kan je tags tekenen zonder elke keer je sleutel op te hoeven geven als je het `git tag` commando aanroeft:

```
$ git tag -s <tag-name>
```

### core.excludesfile

Je kunt patronen in het `.gitignore` bestand van je project zetten om Git deze niet als untracked bestanden te laten beschouwen of deze zal proberen te staggen als je `git add` op ze aanroeft, zoals beschreven in [Bestanden negeren](#).

Maar soms wil je bepaalde bestanden negeren voor alle repositories waar je in werkt. Als je computer onder Mac OS X draait, ben je waarschijnlijk bekend met `.DS_Store` bestanden. Als je voorkeurs-tekstverwerker Emacs of Vim is, zullen bestanden die eindigen op een `~` of `.swp` je bekend voorkomen.

Deze instelling laat je een soort globale `.gitignore_global` bestand aanmaken. Als je een `~/.gitignore_global` bestand aanmaakt met deze inhoud:

```
*~  
.*.swp  
.DS_Store
```

...en je roept `git config --global core.excludesfile ~/.gitignore_global` aan, zal Git je nooit meer lastig vallen over deze bestanden.

### help.autocorrect

Als je een typefout maakt in een commando, laat het je iets als dit zien:

```
$ git chekcout master  
git: 'chekcout' is not a git command. See 'git --help'.
```

```
Did you mean this?  
  checkout
```

Git probeert behulpzaam uit te vinden wat je bedoeld zou kunnen hebben, maar weigert het wel dit uit te voeren. Als je `help.autocorrect` op 1 instelt, zal Git het commando daadwerkelijk voor je uitvoeren:

```
$ git chekcout master  
WARNING: You called a Git command named 'chekcout', which does not exist.  
Continuing under the assumption that you meant 'checkout'  
in 0.1 seconds automatically...
```

Merk het “0.1 seconden” gedoe op. `help.autocorrect` is eigenlijk een integer waarde die tienden van

een seconde vertegenwoordigt. Dus als je het op 50 zet, zal Git je 5 seconden geven om van gedachten te veranderen voordat het automatisch gecorigeerde commando wordt uitgevoerd.

## Kleuren in Git

Git ondersteunt gekleurde terminal uitvoer volledig, wat enorm helpt in het snel en eenvoudig visueel verwerken van de uitvoer van commando's. Een aantal opties kunnen je helpen met het bepalen van jouw voorkeurs-kleuren.

### color.ui

Git geeft automatisch de meeste van haar uitvoer in kleur weer, maar er is een hoofdschakelaar als dit gedrag je niet bevalt. Om alle kleuren uitvoer van Git uit te zetten, voer je dit uit:

```
$ git config --global color.ui false
```

De standaard waarde is `auto`, wat de uitvoer kleur geeft als het direct naar een terminal wordt gestuurd, maar laat de kleuren-stuurcodes achterwege als de uitvoer naar een pipe of een bestand wordt omgeleid.

Je kunt het ook instellen op `always` om het onderscheid tussen terminals en pipes te negeren. Je zult dit zelden willen in de meeste gevallen, als je kleuren-stuurcodes in je omgeleide uitvoer wilt, kan je in plaats daarvan een `--color` vlag aan het Git commando doorgeven om het te dwingen om kleurcodes te gebruiken. De standaard instelling is vrijwel altijd hetgeen je zult willen.

### color.\*

Als je meer controle wilt hebben over welke commando's gekleurd worden en hoe, heeft Git argument-specifieke kleuren-instellingen. Elk van deze kan worden gezet op `true`, `false` of `always`:

```
color.branch  
color.diff  
color.interactive  
color.status
```

Daarenboven heeft elk van deze specifiekeren instellingen die je kunt gebruiken om specifieke kleuren voor delen van de uitvoer te bepalen, als je elke kleur zou willen herbepalen. Bijvoorbeeld, om de meta-informatie in je diff uitvoer op een blauwe voorgrond, zwarte achtergrond en vetgedrukte tekst in te stellen, kan je het volgende uitvoeren:

```
$ git config --global color.diff.meta "blue black bold"
```

Je kunt de kleur instellen op elk van de volgende waarden: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan`, of `white`. Als je een attribuut als vetgedrukt (`bold`) in het vorige voorbeeld wilt, kan je kiezen uit `bold`, `dim` (minder helder), `ul` (onderstrepen, underline), `blink` (knipperen), en `reverse` (verwissel voor- en achtergrond).

## Externe merge en diff tools

Hoewel Git een interne implementatie van diff heeft, dat is wat we hebben laten zien in dit boek, kan je een externe tool inrichten. Je kunt ook een grafische merge-conflict-oplosgereedschap inrichten in plaats van het handmatig oplossen van de conflicten. We zullen je het inrichten van het Perforce Visual Merge Tool (P4Merge) laten zien om je diffs en merge resoluties te laten doen, omdat het een fijne grafische tool is en omdat het is gratis.

Als je dit wilt proberen, P4Merge werkt op alle meest gebruikte platformen, dus je zou het op deze manier moeten kunnen doen. We zullen in de voorbeelden pad-namen gaan gebruiken die op Mac en Linux systemen werken; voor Windows zal je `/usr/local/bin` in een uitvoerbaar pad moeten veranderen in jouw omgeving.

Om te beginnen, [download P4Merge van Perforce](#). Vervolgens ga je externe wrapper scripts maken om je commando's uit te voeren. We zullen het Mac pad voor de executable gebruiken; op andere systemen zal het zijn waar je het `p4merge` binary bestand hebt geïnstalleerd. Maak een merge wrapper script genaamd `extMerge` dat je binary aanroept met alle gegeven argumenten:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

De diff wrapper verifieert dat er zeven argumenten worden doorgegeven en geeft twee ervan door aan je merge script. Standaard geeft Git de volgende argumenten door aan het diff programma:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Omdat je alleen de `old-file` en `new-file` argumenten wilt, gebruik je het wrapper script om degene door te geven die je nodig hebt.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Je moet ook ervoor zorgen dat deze scripts uitvoerbaar worden gemaakt:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Nu kan je jouw configuratie bestand inrichten om jouw aangepaste merge oplossing en diff instrumenten worden gebruikt. Dit vergt een aantal aangepaste instellingen: `merge.tool` om Git te vertellen welke strategie er gebruikt dient te worden, `mergetool.<tool>.cmd` om aan te geven hoe het commando aan te roepen, `mergetool.<tool>.trustExitCode` om Git te vertellen of de uitvoercode van dat programma een succesvolle merge oplossing aangeeft of niet, en `diff.external` om Git te vertellen welk commando het moet aanroepen voor diffs. Dus je kunt kiezen om vier config

commando's aan te roepen

```
$ 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
```

of je kunt je `~/.gitconfig` bestand aanpassen door deze regels toe te voegen:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

Als dit ingericht is, zal, als je diff commando's als deze aanroeft:

```
$ git diff 32d1776b1^ 32d1776b1
```

Git zal, in plaats van de diff uitvoer op de commando-regel te geven, P4Merge opstarten wat er ongeveer zo uit zal zien:

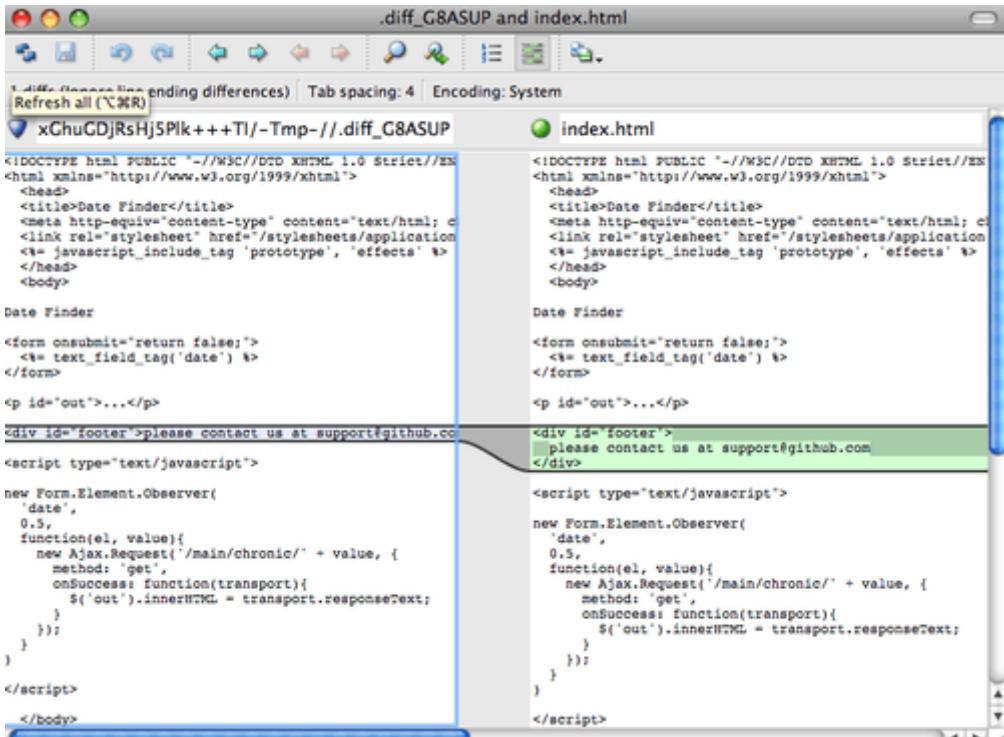


Figure 143. P4Merge.

Als je probeert om twee branches te mergen en je krijgt vervolgens merge conflicten, kan je het

commando `git mergetool` aanroepen; Git zal P4Merge opstarten om je de conflicten middels dat grafische gereedschap op te laten lossen.

Het mooie van deze wrapper inrichting is dat je je diff en merge gereedschappen eenvoudig kan wijzigen. Bijvoorbeeld, om je `extDiff` en `extMerge` gereedschappen te wijzigen om het KDiff3 gereedschap aan te roepen, is het enige wat je hoeft te doen het wijzigen van je `extMerge` bestand:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Nu zal Git het KDiff3 gereedschap gebruiken voor het bekijken van diffs en het oplossen van merge conflicten.

Git is voor-ingericht om een aantal andere merge-oplossings gereedschappen te gebruiken waarbij er geen noodzaak is om de cmd-configuratie op te zetten. Om een lijst te zien van de ondersteunde gereedschappen, probeer eens dit:

```
$ git mergetool --tool-help
```

```
'git mergetool --tool=<tool>' may be set to one of the following:
```

```
    emerge
    gvimdiff
    gvimdiff2
    opendiff
    p4merge
    vimdiff
    vimdiff2
```

```
The following tools are valid, but not currently available:
```

```
    araxis
    bc3
    codecompare
    deltaxwalker
    diffmerge
    diffuse
    ecmerge
    kdiff3
    meld
    tkdiff
    tortoisemerge
    xxdiff
```

```
Some of the tools listed above only work in a windowed
environment. If run in a terminal-only session, they will fail.
```

Als je niet geïnteresseerd bent in het gebruik van KDiff3 voor diff, maar dit liever wilt gebruiken voor alleen het oplossen van merge conflicten, en het kdiff3 commando is op je pad, dan kan je dit aanroepen:

```
$ git config --global merge.tool kdiff3
```

Als je dit uitvoert in plaats van de `extMerge` en `extDiff` bestanden te maken, zal Git KDiff3 gebruiken voor merge conflict oplossingen en het reguliere Git diff gereedschap voor diffs.

## Formatering en witruimtes

Formatering en witruimte problemen zijn een paar van de meest frustrerende en subtiele problemen die veel ontwikkelaars tegenkomen wanneer ze samenwerken, vooral bij verschillende platforms. Het is erg eenvoudig om middels patches of ander gedeeld werk om subtiele witruimte wijzigingen te introduceren omdat tekstverwerkers deze stilletjes invoeren, en als je bestanden ooit in aanraking komen met een Windows systeem, zullen de regeleinden mogelijk vervangen zijn. Git heeft een aantal configuratie opties om je bij deze problemen te helpen.

### `core.autocrlf`

Als je in Windows programmeert en werkt met mensen die dat niet doen (of vice-versa), zal je waarschijnlijk op enig moment met regel-einde problematiek te maken krijgen. Dit is omdat Windows zowel een wagen-terugvoer teken (carriage-return) en een regelopvoer teken (linefeed) gebruikt voor nieuwe regels in haar bestanden, waar Mac en Linux systemen alleen het linefeed teken gebruiken. Dit is een subtiel maar ongelofelijk ergerlijk feit van het werken op verschillende platforms; veel tekstverwerkers op Windows vervangen stilletjes bestaande LF-stijl regeleinden met CRLF, of voegen beide regel-einde karakters in als de gebruiker de enter toets gebruikt.

Git kan dit verwerken door automatisch CRLF regel-einden te converteren in een LF als je een bestand in de index toevoegt, en vice-versa als je code uitcheckt naar je bestandssysteem. Je kunt deze functionaliteit aanzetten met de `core.autocrlf` instelling. Als je op een Windows machine werkt, zet dit dan op `true` - dit zal LF einden naar CRLF vertalen als je code uitcheckt:

```
$ git config --global core.autocrlf true
```

Als je op een Linux of Mac systeem werkt die LF regeleinden gebruikt, dan wil je niet dat Git deze automatisch vertaalt als je bestanden uitcheckt, echter als een bestand met CRLF regeleinden per ongeluk binnenkomt, dan zou je wellicht willen dat Git dit rechtstreekt. Je kunt Git vertellen om CRLF naar LF te vertalen bij een commit, maar niet omgekeerd door de instelling `core.autocrlf` op input te zetten:

```
$ git config --global core.autocrlf input
```

Met deze instelling zou je uit moeten komen met CRLF regeleinden in Windows checkouts, maar LF regeleinden op Mac en Linux systemen en in de repository.

Als je een Windows programmeur bent die aan een project werkt voor alleen Windows, kan je deze functionaliteit uitzetten, waarbij carriage returns in de repository worden vastgelegd door de configuratie waarde op `false` te zetten:

```
$ git config --global core.autocrlf false
```

## core.whitespace

Git wordt geleverd met de instelling om een aantal witruimte problemen te detecteren en op te lossen. Het kan zoeken naar zes meestvoorkomende witruimte problemen—drie ervan zijn standaard ingeschakeld en kunnen worden uitgeschakeld, en drie zijn uitgeschakeld maar kunnen worden geactiveerd.

Degenen die standaard aanstaan zijn **blank-at-eol**, die naar spaties aan het eind van een regel kijkt; **blank-at-eof**, die blanco regels opmerkt aan het eind van een bestand; en **space-before-tab**, die kijkt naar spaties voor tabs aan het begin van een regel.

De drie die standaard uitstaan, maar die kunnen worden aangezet, zijn **indent-with-non-tab**, die kijkt naar regels die beginnen met spaties in plaats van tabs (en wordt geregeld met de **tabwidth** optie); **tab-in-indent**, die kijkt naar tabs in het inspring-gedeelte van een regel; en **cr-at-eol**, welke Git vertelt dat carriage returns aan het eind van regels worden geaccepteerd.

Je kunt Git vertellen welke van deze je ingeschakeld wilt hebben door **core.whitespace** op de waarden te zetten die je aan of uit wilt hebben, gescheiden met komma's. Je kunt instellingen uitzetten door ze weg te laten uit de instelling-reeks of ze vooraf te laten gaan met een **-**. Bijvoorbeeld, als je alles behalve **cr-at-eol** wilt inschakelen kan je dit doen (met **trailing-space** als afkorting die zowel **blank-at-eol** als `blank-at-eof` afdekt):

```
$ git config --global core.whitespace \
    trailing-space,-space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Of je kunt alleen het specificerende gedeelte aangeven:

```
$ git config --global core.whitespace \
    -space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Git zal deze problemen opsporen als je een **git diff** commando aanroeft en ze proberen met een kleur aan te geven zodat je ze mogelijk kunt oplossen voordat je commit. Het zal deze waarden ook gebruiken om je te helpen als je patches toepast met **git apply**. Als je patches aan het toepassen bent, kan je Git ook vragen om je te waarschuwen als het patches toepast met de aangegeven witruimte problematiek:

```
$ git apply --whitespace=warn <patch>
```

Of je kunt Git het probleem automatisch laten proberen op te lossen voordat het de patch toepast:

```
$ git apply --whitespace=fix <patch>
```

Deze opties zijn ook van toepassing voor het `git rebase` commando. Als je witruimte problemen hebt gecommit, maar je hebt ze nog niet stroomopwaarts gepusht, kan je `git rebase --whitespace=fix` aanroepen om Git deze problemen automatisch te laten oplossen als het de patches herschrijft.

## Server configuratie

Er zijn lang niet zoveel configuratie opties beschikbaar voor de server kant van Git, maar er zijn een aantal interessante waar je wellicht naar wilt kijken.

### `receive.fsckObjects`

Git is in staat om te verifiëren dat elk ontvangen object bij een push nog steeds een passende SHA-1 checksum heeft en naar geldige objecten wijst. Dit doet het echter niet standaard; het is een relatief dure operatie, en kan het uitvoeren van de operatie vertragen, zeker bij grote repositories of pushes. Als je wilt dat Git object consistentie bij elke push controleert, kan je dit afdwingen door `receive.fsckObjects` op true te zetten:

```
$ git config --system receive.fsckObjects true
```

Nu zal Git de integriteit van je repository controleren voordat elke push wordt geaccepteerd om er zeker van te zijn dat defecte (of kwaadwillende) werkstations geen corrupte gegevens aanleveren.

### `receive.denyNonFastForwards`

Als je commits rebased die je al gepusht hebt en dan weer probeert te pushen, of op een andere manier probeert eencommit te pushen naar een remote branch die de commit niet bevat waar de remote branch op dit moment naar wijst, zal dit geweigerd worden. Dit is normaalgesproken een goed beleid, maar in het geval van de rebase, kan je besluiten dat je weet wat je aan het doen bent en de remote branch geforceerd updaten met een `-f` vlag bij je push commando.

Om Git te vertellen om geforceerd pushen te weigeren, zet je `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

De andere manier waarop je dit kunt doen is via receive hooks aan de kant van de server, waar we straks meer over gaan vertellen. Die aanpak stelt je ook in staat om iets complexere dingen te doen als het weigeren van non-fast-forwards bij een bepaalde groep van gebruikers.

### `receive.denyDeletes`

Een van de manieren om om het `denyNonFastForwards` beleid heen te werken is om als gebruiker de branch te verwijderen en deze dan weer te pushen met de nieuwe referenties. Om dit te voorkomen, zet `receive.denyDeletes` op true:

```
$ git config --system receive.denyDeletes true
```

Dit weigert het verwijderen van branches of tags — geen enkele gebruiker kan dit doen. Om remote branches te verwijderen, moet je de ref-bestanden handmatig van de server verwijderen. Er zijn nog meer interessante manieren om dit te doen op een gebruikersgerichte manier via ACL's, zoals je zult zien in [Een voorbeeld van Git-afgedwongen beleid](#).

## Git attributen

Een aantal van deze settings kunnen ook worden ingericht voor een pad, zodat Git deze instellingen alleen zal gebruiken voor een subdirectory of een subset van bestanden. Deze pad-specifieke instellingen worden Git attributen genoemd, en worden bewaard in een `.gitattributes` bestand in een van je directories (normaal gesproken de root van je project) of in het `.git/info/attributes` bestand als je dit attributen bestand niet met je project wilt committen.

Met het gebruik van attributen, kan je dingen doen als het specificeren van separate merge strategieën voor individuele bestanden of directories in je project, Git vertellen hoe niet-tekst bestanden moeten worden gedifft, of Git inhoud laten filteren voordat je het naar of van Git in- of uitcheckt. In deze paragraaf zullen we je een en ander vertellen over de attributen die je kunt inrichten op je paden in je Git project en je zult een aantal voorbeelden zien hoe deze mogelijkheden in de praktijk gebruikt kunnen worden.

### Binaire bestanden

Een aardig voorbeeld waar je Git attributen voor kunt gebruiken is Git vertellen welke bestanden binair zijn (in die gevallen waar het niet zelf kan bepalen) en Git speciale instructies te geven over hoe deze bestanden te behandelen. Bijvoorbeeld, sommige tekst bestanden kunnen door applicaties zijn gegenereerd en daarmee niet diffbaar, terwijl sommige binaire bestanden juist wel kunnen worden gedifft. We zullen je laten zien hoe je Git kunt vertellen het onderscheid te maken.

#### Binaire bestanden herkennen

Sommige bestanden zien er uit als tekst bestanden, maar moeten praktisch gezien gewoon behandeld worden als binaire gegevens. Als voorbeeld, Xcode projecten op de Mac bevatten een bestand dat eindigt op `.pbxproj`, wat eigenlijk gewoon een JSON (platte tekst Javascript gegevens formaat) dataset is die door de IDE naar schijf geschreven wordt, waarin je bouwinstellingen staan en zo voorts. Alhoewel het technisch gesproken een tekst bestand is (omdat het allemaal UTF-8 is), wil je het niet als dusdanig behandelen omdat het eigenlijk een lichtgewicht database is - je kunt de inhoud niet mergen als twee personen het wijzigen, en diffs zijn over het algemeen niet echt nuttig. De bedoeling van dit bestand is om door een machine te worden verwerkt. Het komt erop neer dat je het wilt behandelen als een binair bestand.

Om Git te vertellen om alle `.pbxproj` bestanden als binaire gegevens te behandelen, voeg je de volgende regel toe aan je `.gitattributes` bestand:

```
*.pbxproj binary
```

Nu zal Git niet proberen om CRLF gevallen te converteren of te repareren, en ook zal het niet proberen om een diff te bepalen of af te drukken voor wijzigingen in dit bestand als je `git show` of

`git diff` op je project aanroept.

## Binaire bestanden diffen

Je kunt de Git attributen functionaliteit ook gebruiken om binaire bestanden feitelijk te diffen. Je kunt dit doen door Git te vertellen hoe het je binaire gegevens moet converteren naar een tekst formaat die weer via een normale diff kan worden vergeleken.

Allereerst zal je deze techniek gebruiken om een van de meest ergerlijke problemen die er voor de mensheid bestaan op te lossen: het onder versie beheer plaatsen van Microsoft Word documenten. Iedereen weet dat Word een van de afschuwwekkendste tekstverwerkers is die bestaat maar, die vreemd genoeg, nog steeds door iedereen gebruikt wordt. Als je Word documenten onder versie beheer wilt brengen, kan je ze in een Git repository stoppen en eens in de zoveel tijd committen, maar wat is het nut hiervan? Als je `git diff` gewoon zou aanroepen, zou je alleen maar iets als dit zien:

```
$ git diff  
diff --git a/chapter1.docx b/chapter1.docx  
index 88839c4..4afcb7c 100644  
Binary files a/chapter1.docx and b/chapter1.docx differ
```

Je kunt twee versies niet rechtstreeks vergelijken tenzij je ze uitcheckt en dan handmatig vergelijkt, toch? We zullen laten zien dat je dit redelijk goed kunt doen met Git attributen. Zet de volgende regel in je `.gitattributes` bestand:

```
*.docx diff=word
```

Dit vertelt Git dat elk bestand dat past in het patroon (`.docx`) het “word” filter moet gebruiken als je een diff probeert te bekijken waar verschillen in zitten. Wat is het “word” filter? Je moet het zelf inrichten. Hier ga je Git configureren om het `docx2txt` programma te gebruiken om Word documenten in leesbare tekstbestanden te converteren, die vervolgens juist kunnen worden gedifft.

Allereerst moet je `docx2txt` installeren, je kunt het downloaden van <http://docx2txt.sourceforge.net>. Volg de instructies in het `INSTALL` bestand om het ergens neer te zetten waar je shell het kan vinden. Vervolgens, schrijf je een wrapper script om de uitvoer te converteren naar het formaat dat Git verwacht. Maak een bestand ergens in je pad en noem dezer `docx2txt` en zet dit hierin:

```
#!/bin/bash  
docx2txt.pl "$1" -
```

Vergeet dit niet op dit bestand `chmod a+x` aan te roepen. Tot slot kan je Git configureren om dit script te gebruiken:

```
$ git config diff.word.textconv docx2txt
```

Nu weet Git dat als het een diff probeert uit te voeren tussen twee snapshots, en een van de bestandsnamen eindigt op `.docx`, dat het deze bestanden door het “word” filter moet halen die als het `docx2txt` programma gedefinieerd is. Effectief maakt dit mooie tekst-gebaseerde versies van je Word bestanden voordat het probeert deze te diffen.

Hier is een voorbeeld: Hoofdstuk 1 van dit boek was geconverteerd naar een Word formaat en gecommit in een Git repository. Toen is er een nieuwe sectie toegevoegd. Dit is wat `git diff` laat zien:

```
$ git diff  
diff --git a/chapter1.docx b/chapter1.docx  
index 0b013ca..ba25db5 100644  
--- a/chapter1.docx  
+++ b/chapter1.docx  
@@ -2,6 +2,7 @@
```

This chapter will be about getting started with Git. We will begin at the beginning by explaining some background on version control tools, then move on to how to get Git running on your system and finally how to get it setup to start working with. At the end of this chapter you should understand why Git is around, why you should use it and you should be all setup to do so.

### 1.1. About Version Control

What is “version control”, and why should you care? Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. For the examples in this book you will use software source code as the files being version controlled, though in reality you can do this with nearly any type of file on a computer.

+Testing: 1, 2, 3.

If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

#### 1.1.1. Local Version Control Systems

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

Git vertelt ons succesvol en bondig dat we de tekenreeks “Testing: 1, 2, 3,” hebben toegevoegd, wat juist is. Het is niet perfect - wijzigingen in formattering worden niet zichtbaar - maar het werkt wel.

Een ander interessant probleem wat je op deze manier kunt oplossen betreft het diffen van bestanden met afbeeldingen. Een manier om dit te doen is om afbeeldingen door een filter te halen die hun EXIF informatie extraheert - metadata die bij de meeste grafische bestanden worden vastgelegd. Als je het `exiftool` downloadt en installeert, kan je deze gebruiken om je afbeeldingen naar tekst over de metadata te converteren, zodat de diff je in elk geval een tekstuele representatie

van wijzigingen kan laten zien: Zet de volgende regel in je `.gitattributes` bestand:

```
*.png diff=exif
```

Configureer Git om dit tool te gebruiken:

```
$ git config diff.exif.textconv exiftool
```

Als je een afbeelding in je project vervangt en dan `git diff` aanroeft, zie je iets als dit:

```
diff --git a/image.png b/image.png
index 88839c4..4afcb7c 100644
--- a/image.png
+++ b/image.png
@@ -1,12 +1,12 @@
ExifTool Version Number      : 7.74
-File Size                  : 70 kB
-File Modification Date/Time: 2009:04:21 07:02:45-07:00
+File Size                  : 94 kB
+File Modification Date/Time: 2009:04:21 07:02:43-07:00
File Type                   : PNG
MIME Type                   : image/png
-Image Width                : 1058
-Image Height               : 889
+Image Width                : 1056
+Image Height               : 827
Bit Depth                   : 8
Color Type                  : RGB with Alpha
```

Je kunt eenvoudig zien dat de bestandsgrootte en de dimensies van je afbeelding beide zijn veranderd.

## Sleutelwoord expansie (Keyword expansion)

Keyword expansion zoals je dit kan vinden bij SVN of CVS wordt vaak gewenst door ontwikkelaars die gewend zijn aan die systemen. Het grote probleem met dit is in Git is dat je een bestand niet kunt aanvullen met informatie over de commit nadat je het hebt gecommit, omdat Git eerst een checksum van het bestand maakt. Je kunt echter tekst in een bestand injecteren als het uitgecheckt wordt en het weer verwijderen voordat het aan een commit wordt toegevoegd. Git attributen geven je twee manieren om dit te bereiken.

Als eerste kan je automatisch de SHA-1 checksum van een blob in een `$Id$` veld injecteren. Als je dit attribuut op een bestand of een aantal bestanden zet dan zal Git, als je die branch een volgende keer uitcheckt, dat veld vervangen met de SHA-1 van de blob. Het is belangrijk om op te merken dat het niet de SHA-1 van de commit is, maar van de blob zelf.

```
$ echo '*.txt ident' >> .gitattributes  
$ echo '$Id$' > test.txt
```

De volgende keer als je dit bestand uitcheckt, zal Git de SHA-1 van de blob injecteren:

```
$ rm test.txt  
$ git checkout -- test.txt  
$ cat test.txt  
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Dit resultaat is echter van beperkt nut. Als je keyword substitutie in CVS of Subversion gebruikt hebt, kan je een datestamp bijsluiten - de SHA-1 is eigenlijk niet zo nuttig, omdat het redelijk willekeurig is en je kunt aan een SHA-1 niet zien of het ouder of jonger is dan een andere door alleen er naar te kijken.

Je kunt echter je eigen filters schrijven om substituties in bestanden doen bij commit/checkout. Dit worden “clean” (kuis) en “smudge” (besmeur) filters genoemd. In het `.gitattributes` bestand kan je een filter instellen voor specifieke paden en scripts maken die de bestanden bewerken vlak voordat ze worden uitgechecked (“smudge”, zie [Het “smudge” filter wordt bij checkout aangeroepen](#)) en vlak voordat ze worden gestaged (“clean”, zie [Het “clean” filter wordt aangeroepen als bestanden worden gestaged](#)). Deze filters kunnen worden verteld om allerhande leuke dingen te doen.

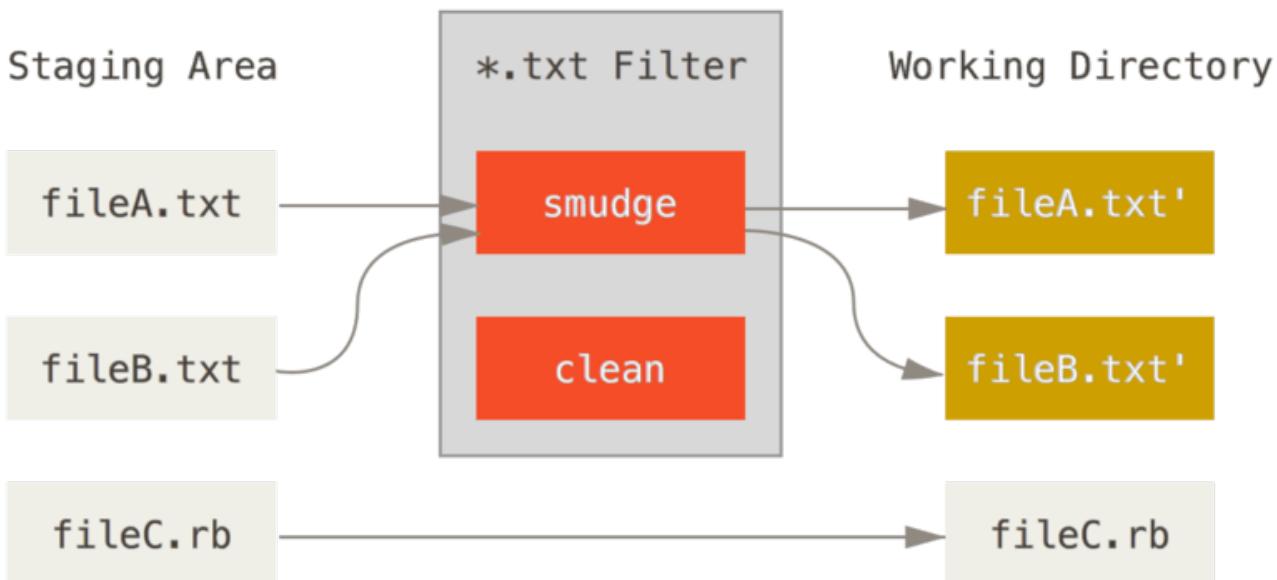


Figure 144. Het “smudge” filter wordt bij checkout aangeroepen.

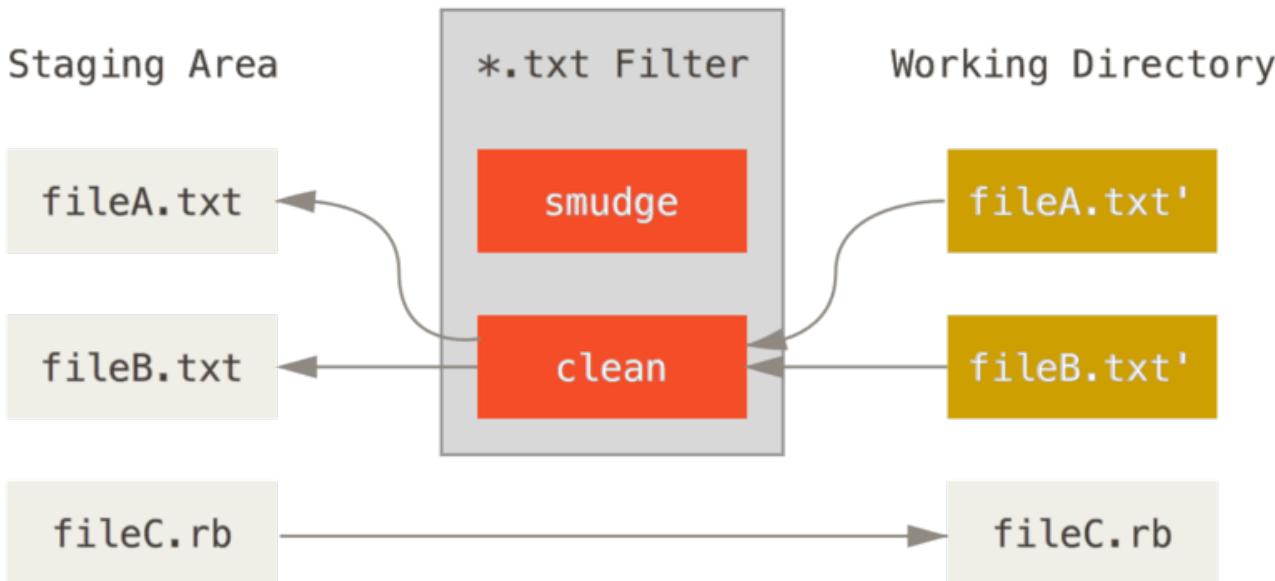


Figure 145. Het “clean” filter wordt aangeroepen als bestanden worden gestaged.

Het orginele commit bericht voor deze functionaliteit geeft een simpel voorbeeld van al je C broncode door het `indent` programma te laten bewerken voor het committen. Je kunt het inrichten door het filter attribuut in je `.gitattributes` bestand `*.c` bestanden te laten filteren met het “indent” filter:

```
*.c filter=indent
```

Daarna vertel je Git wat het “indent” filter moet doen bij smudge en clean:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

In dit geval, als je bestanden commit die lijken op `*.c`, zal Git deze door het `indent` programma halen voordat het deze staget en ze door het `cat` programma halen voordat het ze weer naar schijf uitcheckt. Het `cat` programma doet eigenlijk niets: het geeft de gegevens die binnenkomen ongewijzigd door. Deze combinatie zal effectief alle C broncode door `indent` laten filteren voor het te committen.

Een ander interessant voorbeeld veroorzaakt `$Date$` keyword expansie, zoals bij RCS. Om dit goed te doen heb je een klein scriptje nodig dat een bestandsnaam neemt, de laatste commit datum vindt voor dit project en dan de datum in dat bestand injecteren. Hier is een kort Ruby script dat precies dit doet:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Al wat dit script doet is de laatste commit datum van het `git log` commando uitlezen, en dat in elke `$Date$` tekenreeks die het in stdin ziet zetten, en drukt het resultaat af - het zou eenvoudig moeten zijn om dit te doen in een taal waar je het meest vertrouwd mee bent. Je kunt dit bestand `expand_date` noemen en het op je pad plaatsen. Nu moet je een filter opzetten in Git (noem het `dater`) en het vertellen om je `expand_date` filter te gebruiken om de bestanden te besmeuren bij uitchecken. Je kunt een Perl expressie gebruiken om dat bij commit weer op te kuisen:

```
$ git config filter.dater.smudge expand_date  
$ git config filter.dater.clean 'perl -pe "s/\$\$Date[^\$\$]*\$\$/\$Date\\\$/"'
```

Dit stukje Perl haalt alles weg wat het ziet in een `$Date$` tekenreeks, om terug te halen waar je mee was begonnen. Nu je filter gereed is, kan je het uitproberen door een bestand te maken met je `$Date$` keyword en daarna een Git attribuut op te zetten voor dat bestand die je nieuwe filter activeert:

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

Als je deze wijzigingen commit en het bestand weer uitcheckt, zal je het keyword correct vervangen zien:

```
$ git add date_test.txt .gitattributes  
$ git commit -m "Testing date expansion in Git"  
$ rm date_test.txt  
$ git checkout date_test.txt  
$ cat date_test.txt  
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Je kunt zien hoe krachtig deze techniek kan zijn voor eigen toepassingen. Je moet echter wel voorzichtig zijn, omdat het `.gitattributes` bestand gecommit wordt en met het project wordt verspreid, maar dat het uitvoerende element (in dit geval `dater`) dat niet wordt, zodat het niet overal zal werken. Als je deze filters ontwerpt, moeten ze in staat zijn om netjes te falen en het project nog steeds goed te laten werken.

## Je repository exporteren

De Git attribute gegevens staan je ook toe om interessante dingen te doen als je een archief van je project exporteert.

### export-ignore

Je kunt Git vertellen dat sommige bestanden of directories niet geëxporteerd moeten worden bij het genereren van een archief. Als er een subdirectory of bestand is waarvan je niet wilt dat het wordt meegenomen in je archief bestand, maar dat je wel in je project ingecheckt wil hebben, kan je die

bestanden benoemen met behulp van het `export-ignore` attribuut.

Bijvoorbeeld, stel dat je wat testbestanden in een `test/` subdirectory hebt, en dat het geen zin heeft om die in de tarball export van je project mee te nemen. Dan kan je de volgende regel in je Git attributes bestand toevoegen:

```
test/ export-ignore
```

Als je nu `git archive` uitvoert om een tarball van je project te maken, zal die directory niet meegenomen worden in het archief.

### export-subst

Als je bestanden exporteert voor deployment kan je de formattering en sleutelwoord expansie van `git log` toepassen om delen van bestanden selecteren die met het `export-subst` attribuut zijn gemarkeerd.

Bijvoorbeeld, als je een bestand genaamd `LAST_COMMIT` wilt meenemen in je project, waarin de laatste commit datum op van het moment dat `git archive` liep automatisch wordt geïnjecteerd, kan je het bestand als volgt instellen:

```
LAST_COMMIT exportsubst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Als je `git archive` uitvoert, zal de inhoud van dat bestand er zo uit zien:

```
$ git archive HEAD | tar xf .../deployment-testing -
$ cat .../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

De vervangingen kunnen bijvoorbeeld het commit bericht en elke `git note` omvatten, en `git log` kan eenvoudige word-wrapping uitvoeren:

```
$ echo '$Format:Last commit: %h by %aN at %cd%n%+w(76,6,9)%B$' > LAST_COMMIT
$ git commit -am 'export-subst uses git log\'s custom formatter

git archive uses git log's 'pretty=format:' processor
directly, and strips the surrounding '$Format:' and '$'
markup from the output.

'$

$ git archive @ | tar xf0 - LAST_COMMIT
Last commit: 312ccc8 by Jim Hill at Fri May 8 09:14:04 2015 -0700
    export-subst uses git log's custom formatter

    git archive uses git log's 'pretty=format:' processor directly, and
    strips the surrounding '$Format:' and '$' markup from the output.
```

Het archief wat hieruit komt is bruikbaar voor deployment werkzaamheden, maar zoals elk geëxporteerde archief is het niet echt toepasbaar voor verdere ontwikkel werk.

## Merge strategieën

Je kunt Git attributen ook gebruiken om Git te vertellen dat het verschillende merge strategieën moet gebruiken voor specifieke bestanden in je project. Een erg handige toepassing is Git te vertellen dat het bepaalde bestanden niet moet proberen te mergen als ze conflicten hebben, maar jouw versie moeten gebruiken in plaats van die van de ander.

Dit is handig als een branch in jouw project uiteen is gelopen of gespecialiseerd is, maar je wel in staat wilt zijn om veranderingen daarvan te mergen, en je wilt bepaalde bestanden negeren. Stel dat je een database instellingen-bestand hebt dat `database.xml` heet en tussen twee branches verschilt, en je wilt de andere branch mergen zonder jouw database bestand overhoop te halen. Je kunt dan een attribuut als volgt instellen:

```
database.xml merge=ours
```

En daarna een loze `ours` merge strategie definiëren met:

```
$ git config --global merge.ours.driver true
```

Als je in de andere branch merget, dan zal je in plaats van merge conflicten met het `database.xml` bestand zoiets als dit zien:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In dit geval blijft `database.xml` staan op de versie die je oorspronkelijk al had.

# Git Hooks

Net als veel andere versie beheer systemen, heeft Git een manier om eigen scripts aan te roepen als er bepaalde belangrijke gebeurtenissen plaatsvinden. Er zijn twee groepen van deze haken (hooks): aan de kant van het werkstation en aan de kant van de server. De hooks aan de kant van het werkstation worden afgetrapt door operaties zoals committen en mergen, terwijl die aan de server kant worden afgetrapt door netwerk operaties zoals het ontvangen van gepushte commits. Je kunt deze hooks voor allerhande doeleinden gebruiken.

## Het installeren van een hook

De hooks worden allemaal opgeslagen in de `hooks` subdirectory van de Git directory. In de meeste projecten is dat `.git/hooks`. Als je een nieuwe repository met `git init` initialiseert, vult Git de hooks directory met een aantal voorbeeld scripts, vele ervan zijn op zichzelf al nuttig; maar ze beschrijven ook de invoer waarden van elk script. Al de voorbeelden zijn geschreven als shell scripts, met af en toe wat Perl erdoorheen, maar elk uitvoerbaar script met een goede naam zal prima werken - je kunt ze in Ruby of Python schrijven of verzin maar een taal. Als je de meegeleverde hook scripts wilt gebruiken, zul je ze moeten hernoemen; de bestandsnamen eindigen allemaal met `.sample`.

Om een hook script te activeren, zet een bestand in de `hooks` subdirectory van je Git directory met een juiste naam (geen enkele extensie) en zorg dat 'ie uitvoerbaar is. Vanaf dat moment zou het aangeroepen moeten worden. We zullen de meest belangrijke hook bestandsnamen hier behandelen.

## Hooks bij werkstations

Er zijn veel hooks aan de kant van een werkstation. Deze paragraaf verdeelt ze in hooks voor een committing-workflow, scripts voor een e-mail-workflow en al het andere.



Het is belangrijk om op te weten dat hooks bij werkstations **niet** worden gekopieerd als je een repository kloont. Als het de bedoeling is van deze scripts om een beleid af te dwingen, moet je dat waarschijnlijk aan de kant van de server doen; zie het voorbeeld in [Een voorbeeld van Git-afgedwongen beleid](#).

### Committing-Workflow Hooks

De eerste vier hooks hebben te maken met het proces van committen.

De `pre-commit` hook wordt het eerst aangeroepen, voordat je zelfs een commit bericht begint te typen. Het wordt gebruikt om de snapshot die op het punt staat te worden gecommit te inspecteren, om te zien of je iets vergeten bent, om er zeker van te zijn dat de tests slagen of om de code te controleren op iets waar je het op gecontroleerd wilt hebben. Als dit script niet-nul wordt beëindigd breekt de commit af, al kan je er altijd nog omheen met `git commit --no Verify`. Je kunt dingen doen als de code op stijl controleren (`lint` of iets vergelijkbaar aanroepen), controleren op witruimtes aan het eind van regels (de standaard hook doet precies dat), of controleren of nieuwe methoden goed zijn gedocumenteerd.

De `prepare-commit-msg` hook wordt aangeroepen voordat de commit bericht tekstverwerker wordt opgestart, maar nadat het standaard bericht gemaakt is. Dit geeft je de ruimte om het standaard bericht aan te passen voordat de commit auteur deze ziet. Deze hook heeft een aantal parameters: het pad naar het bestand dat het commit bericht op dat moment bevat, het type commit en de SHA-1 van de commit als het een amended (gewijzigde) commit betreft. Deze hook is over het algemeen niet nuttig voor normale commits; maar is daarentegen goed voor commits waar het standaard bericht automatisch is gegenereerd, zoals commit berichten van een sjabloon, merge commits, squashed commits en amended commits. Je kunt het gebruiken in combinatie met een commit sjabloon om via een programma informatie in te voegen.

De `commit-msg` hook krijgt één parameter, en dat is weer het pad naar een tijdelijk bestand dat het commit bericht bevat die door de ontwikkelaar is geschreven. Als dit script met een niet-nul waarde eindigt, zal Git het commit proces afbreken, en daarmee kan je dit gebruiken om de status van het project of het commit bericht te valideren voordat je toestaat dat een commit doorgaat. In de laatste paragraaf van dit hoofdstuk zullen we laten zien hoe deze hook te gebruiken om te controleren dat het commit bericht voldoet aan een vereist patroon.

Als het gehele commit proces is voltooid wordt de `post-commit` hook aangeroepen. Deze krijgt geen enkele parameter, maar je kunt de laatste commit eenvoudig te pakken krijgen door `git log -1 HEAD` aan te roepen. Meestal wordt dit script gebruikt voor het doen van meldingen of iets vergelijkbaars.

## E-mail workflow hooks

Je kunt drie hooks aan de kant van het werkstation inrichten voor een op e-mail gebaseerde workflow. Ze worden alle aangeroepen door het `git am` commando, dus als je dat commando niet gebruikt in je workflow, kan je dit gedeelte rustig overslaan en doorgaan naar de volgende paragraaf. Als je patches per e-mail ontvangt die door `git format-patch` zijn gemaakt, kunnen een aantal van deze behulpzaam zijn.

De eerste hook die wordt aangeroepen is `applypatch-msg`. Deze krijgt een enkele argument: de naam van het tijdelijke bestand dat het voorgestelde commit bericht bevat. Git breekt het patchen af als dit script met niet-nul eindigt. Je kunt deze gebruiken om je ervan te verzekeren dat het commit bericht juist geformatteerd is, of het bericht normaliseren door met het script deze ter plaatse te bewerken.

De volgende hook die aangeroepen wordt als patches met `git am` worden toegepast is `pre-applypatch`. Ietwat verwarring, wordt deze *na* het toepassen van een patch aangeroepen maar voordat een commit gemaakt wordt, dus je kunt het gebruiken om de snapshot te inspecteren voordat de commit gemaakt wordt. Je kunt tests uitvoeren of op een andere manier de werk-tree met dit script inspecteren. Als er iets ontbreekt of de tests slagen niet, zal het eindigen met niet-nul het `git am` script afbreken zonder de patch te committen.

De laatste hook die wordt aangeroepen met een `git am` operatie is `post-applypatch`, die wordt aangeroepen nadat de commit gemaakt is. Je kunt deze gebruiken om een groep of de auteur van de patch een bericht te sturen van dat je de patch hebt gepulld. Je kunt het patching proces niet stoppen met dit script.

## Overige workstation hooks

De `pre-rebase` hook wordt aangeroepen voordat je ook maar iets rebaset en kan het proces stoppen door met niet-nul te eindigen. Je kunt deze hook gebruiken om het rebasen van commits die al zijn gepusht te weigeren. Het voorbeeld `pre-rebase` hook die Git installeert dit dit, al doet het een aantal aannamen die misschien niet van toepassing zijn op jouw workflow.

De `post-rewrite` hook wordt aangeroepen door commando's die commits vervangen, zoals `git commit --amend` en `git rebase` (echter weer niet door `git filter-branch`). Het enige argument is het commando dat herschrijven heeft veroorzaakt, en het ontvangt een lijst van herschrijvingen op `stdin`. Deze hook heeft veel wat gebruik betreft overeenkomsten met het gebruik van de `post-checkout` en `post-merge` hooks.

Nadat je succesvol een `git checkout` hebt aangeroepen, wordt de `post-checkout` hook aangeroepen; je kunt deze gebruiken om je werk directory juist in te richten voor jouw project omgeving. Dit zou het binnenvullen van grote binair bestanden die je niet in versie beheer wilt hebben kunnen inhouden, het automatisch genereren van documentatie of iets van dien aard.

De `post-merge` hook wordt aangeroepen na een succesvolle `merge` commando. Je kunt deze gebruiken om gegevens terug te zetten in de werk tree die Git niet kan tracken, zoals permissie gegevens. Deze hook kan op vergelijkbare manier de aanwezigheid van bestanden buiten Git valideren die je misschien naar binnen wilt kopiëren als de werk tree wijzigt.

De `pre-push` hook wordt aangeroepen tijdens `git push`, nadat de remote refs zijn ge-update maar voordat er objecten zijn verstuurd. Het ontvangt de naam en locatie van de remote als parameters, en een lijst met refs die op het punt staan te worden geactualiseerd via `stdin`. Je kunt dit gebruiken om een aantal ref updates te valideren voordat er een push plaatsvindt (een niet-nul einde zal de push afbreken).

Git zal nu en dan afval verzamelen (garbage collection) als onderdeel van zijn normale taken, door `git gc --auto` aan te roepen. De `pre-auto-gc` hook wordt aangeroepen vlak voordat de garbage collection plaatsvindt, en kan worden gebruikt om aan te geven dat het staat te gebeuren, of deze operatie af te breken als het nu niet goed uitkomt.

## Hooks aan de kant van de server

Naast de hooks aan de kant van de workstation, kan je als systeem beheerder een aantal belangrijke hooks op de server gebruiken om bijna alle vormen van beleid af te dwingen voor je project. Deze scripts worden voor en na pushes naar de server aangeroepen. De pre hooks kunnen op elk moment niet-nul aflopen om de push af te wijzen zowel als een fout bericht afdrukken die naar het workstation worden gestuurd; je kunt een push beleid opzetten die zo complex is als je zelf wenst.

### `pre-receive`

Het eerste script die wordt aangeroepen als een push wordt afgehandeld van een workstation is `pre-receive`. Deze krijgt een lijst van referenties die worden gepusht vanuit `stdin`; als het niet-nul eindigt wordt geen enkele geaccepteerd. Je kunt deze hook gebruiken om zaken te doen als het verzekeren dat geen van de bijgewerkte referenties non-fast-forwards zijn, of toegangscontroles uit te voeren voor alle refs en bestanden die met de push worden gewijzigd.

## update

Het `update` script lijkt erg op het `pre-receive` script, behalve dat het eenmaal wordt aangeroepen voor elke branch die de pusher probeert bij te werken. Als de pusher meerderes branches probeert te pushen, wordt `pre-receive` maar één keer aangeroepen, terwijl `update` één keer per branch waarnaar wordt gepusht wordt aangeroepen. In plaats van van `stdin` te lezen, krijgt dit script drie argumenten: de naam van de referentie (branch), de SHA-1 waar die referentie naar wees voor de push en de SHA-1 die de gebruiker probeert te pushen. Als het `update` script niet-nul eindigt, wordt alleen die referentie geweigerd, de andere referenties kunnen nog steeds worden geüpdatet.

## post-receive

De `post-receive` hook wordt aangeroepen zodra het hele proces gereed is, en kan worden gebruikt om andere diensten bij te werken of gebruikers een bericht te sturen. Het krijgt dezelfde `stdin` gegevens als de `pre-receive` hook. Voorbeelden omvatten het mailen naar een lijst, continuous integratie services notificeren of een ticket-traceer systeem bij werken - je kunt zelfs de commit berichten doorlezen om te zien of er tickets geopend, gewijzigd of gesloten moeten worden. Dit script kan het push proces niet stoppen, maar het werkstation zal de verbinding niet verbreken voordat dit script geëindigd is, dus wees voorzichtig als je iets probeert te doen wat veel tijd in beslag neemt.

# Een voorbeeld van Git-afgedwongen beleid

In deze paragraaf ga je gebruiken wat je geleerd hebt om een Git workflow te maken, die controleert op een specifiek en alleen bepaalde gebruikers toestaat om bepaalde subdirectories te wijzigen in een project. Je zult client scripts maken die de ontwikkelaar helpen te ontdekken of hun push geweigerd zal worden en server scripts die het beleid afdwingen.

We hebben Ruby gebruikt om deze te schrijven, deels vanwege onze intellectuele inertie, maar ook omdat Ruby eenvoudig te lezen, zelfs als je er niet in zou kunnen schrijven. Echter, elke taal voldoet - alle voorbeeld hook-scripts die met Git geleverd worden zijn in Perl of Bash geschreven, dus je kunt ook genoeg voorbeelden van hooks in deze talen zien door naar deze voorbeelden te kijken.

## Server-kant hook

Al het werk aan de server kant zal in het `update` bestand in je `hooks` directory gaan zitten. De `update` hook zal eens per gepushte branch uitgevoerd worden en accepteert drie argumenten:

- de naam van de referentie waarnaar gepusht wordt
- de oude revisie waar die branch was
- de nieuwe gepushte revisie.

Je hebt ook toegang tot de gebruiker die de push doet als de push via SSH gedaan wordt. Als je iedereen hebt toegestaan om connectie te maken als één gebruiker (zoals `git`) via publieke sleutel authenticatie, dan moet je wellicht die gebruiker een shell wrapper geven die bepaalt welke gebruiker er connectie maakt op basis van de publieke sleutel, en dan een omgevingsvariabele instellen waarin die gebruiker wordt gespecificeerd. Wij gaan er hier van uit dat de gebruiker in de `$USER` omgevingsvariabele staat, dus begint je update script met het verzamelen van alle gegevens die het nodig heeft:

```

#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev = ARGV[1]
$newrev = ARGV[2]
$user = ENV['USER']

puts "Enforcing Policies..."
puts "({$refname}) ($oldrev[0,6]) ($newrev[0,6])"

```

Ja, dat zijn globale variabelen. Niets zeggen - het is eenvoudiger om op deze manier dingen te demonstreren.

### Een specifiek commit-bericht formaat afdwingen

Je eerste uitdaging is afdwingen dat elke commit bericht moet voldoen aan een specifiek formaat. Om iets te hebben om mee te werken, neem even aan dat elk bericht een tekenreeks moet bevatten die er uit ziet als “ref: 1234” omdat je wilt dat iedere commit gekoppeld is aan een werkonderdeel in je ticket systeem. Je moet dus kijken naar iedere commit die gepusht wordt, kijken of die tekst in de commit boodschap zit en als de tekst in één van de commits ontbreekt, met niet nul eindigen zodat de push geweigerd wordt.

Je kunt de lijst met alle SHA-1 waarden van alle commits die gepusht worden verkrijgen door de `$newrev` en `$oldrev` waarden te pakken en ze aan een Git binnenwerk commando genaamd `git rev-list` te geven. Dit is min of meer het `git log` commando, maar standaard voert het alleen de SHA-1 waarden uit en geen andere informatie. Dus, om een lijst te krijgen van alle commit SHA-1's die worden geïntroduceerd tussen één commit SHA-1 en een andere, kun je zo iets als dit uitvoeren:

```

$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cf0e475

```

Je kunt die uitvoer pakken, door elk van die commit SHA's heen lopen, de boodschap daarvan pakken en die boodschap testen tegen een reguliere expressie die op een bepaald patroon zoekt.

Je moet uit zien te vinden hoe je de commit boodschap kunt krijgen van alle te testen commits. Om de echte commit gegevens te krijgen, kun je een andere binnenwerk commando genaamd `git cat-file` gebruiken. We zullen alle binnenwerk commando's in detail behandelen in [Git Binnenwerk](#), maar voor nu is dit wat het commando je geeft:

```
$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

changed the version number

Een simpele manier om de commit boodschap uit een commit waarvan je de SHA-1 waarde hebt te krijgen, is naar de eerste lege regel gaan en alles wat daarna komt pakken. Je kunt dat doen met het **sed** commando op Unix systemen:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
changed the version number
```

Je kunt die toverspreuk gebruiken om de commit boodschap te pakken van iedere commit die geprobeerd wordt te pushen en eindigen als je ziet dat er iets is wat niet past. Om het script te eindigen en de push te weigeren, eindig je met niet nul. De hele methode ziet er zo uit:

```
$regex = /\[ref: (\d+)\]/

# enforced custom commit message format
def check_message_format
  missed_revs = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  missed_revs.each do |rev|
    message = `git cat-file commit #{rev} | sed '1,/^\$/d'`
    if !$regex.match(message)
      puts "[POLICY] Your message is not formatted correctly"
      exit 1
    end
  end
end
check_message_format
```

Door dat in je **update** script te stoppen, zullen updates geweigerd worden die commits bevatten met berichten die niet aan jouw beleid voldoen.

## Een gebruiker-gebaseerd ACL systeem afdwingen

Stel dat je een mechanisme wil toevoegen dat gebruik maakt van een toegangscontrole lijst (ACL) die specificeert welke gebruikers wijzigingen mogen pushen naar bepaalde delen van je project. Sommige mensen hebben volledige toegang, en anderen hebben alleen toestemming om wijzigingen te pushen naar bepaalde subdirectories of specifieke bestanden. Om dit af te dwingen zul je die regels schrijven in een bestand genaamd **acl** dat in je bare Git repository op de server zit. Je zult de **update** hook naar die regels laten kijken, bekijken welke bestanden worden geïntroduceerd voor elke commit die gepusht wordt en bepalen of de gebruiker die de push doet

toestemming heeft om al die bestanden te wijzigen.

Het eerste dat je zult doen is de ACL schrijven. Hier zul je een formaat gebruiken wat erg lijkt op het CVS ACL mechanisme: het gebruikt een serie regels, waarbij het eerste veld `avail` of `unavail` is, het volgende veld een komma gescheiden lijst van de gebruikers is waarvoor de regel geldt en het laatste veld het pad is waarvoor deze regel geldt (leeg betekent open toegang). Alle velden worden gescheiden door een pipe (`|`) karakter.

In dit geval heb je een aantal beheerders, een aantal documentatie schrijvers met toegang tot de `doc` map, en één ontwikkelaar die alleen toegang heeft tot de `lib` en `test` mappen, en je ACL bestand ziet er zo uit:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Je begint met deze gegevens in een structuur in te lezen die je kunt gebruiken. In dit geval, om het voorbeeld eenvoudig te houden, zul je alleen de `avail` richtlijnen handhaven. Hier is een methode die je een associatieve array teruggeeft, waarbij de sleutel de gebruikersnaam is en de waarde een array van paden waar die gebruiker toegang tot heeft:

```
def get_acl_access_data(acl_file)
  # read in ACL data
  acl_file = File.read(acl_file).split("\n").reject { |line| line == '' }
  access = {}
  acl_file.each do |line|
    avail, users, path = line.split('|')
    next unless avail == 'avail'
    users.split(',').each do |user|
      access[user] ||= []
      access[user] << path
    end
  end
  access
end
```

Met het ACL bestand dat je eerder bekeken hebt, zal deze `get_acl_access_data` methode een gegevensstructuur opleverendie er als volgt uit ziet:

```
{"defunkt"=>[nil],  
 "tpw"=>[nil],  
 "nickh"=>[nil],  
 "pjhyett"=>[nil],  
 "schacon"=>["lib", "tests"],  
 "cdickens"=>["doc"],  
 "usinclair"=>["doc"],  
 "ebronte"=>["doc"]}
```

Nu je de rechten bepaald hebt, moet je bepalen welke paden de commits die gepusht worden hebben aangepast, zodat je kunt controleren dat de gebruiker die de push doet daar ook toegang toe heeft.

Je kunt eenvoudig zien welke bestanden gewijzigd zijn in een enkele commit met de `--name-only` optie op het `git log` commando (kort besproken in [Git Basics](#)):

```
$ git log -1 --name-only --pretty=format:'' 9f585d
```

```
README  
lib/test.rb
```

Als je gebruik maakt van de ACL structuur die wordt teruggegeven door de `get_acl_access_data` methode en dat gebruikt met de bestanden in elk van de commits, dan kun je bepalen of de gebruiker toegang heeft om al hun commits te pushen:

```

# only allows certain users to modify certain subdirectories in a project
def check_directory_perms
  access = get_acl_access_data('acl')

  # see if anyone is trying to push something they can't
  new_commits = `git rev-list #{$oldrev}..#{$newrev}`.split("\n")
  new_commits.each do |rev|
    files_modified = `git log -1 --name-only --pretty=format:'' #{rev}`.split("\n")
    files_modified.each do |path|
      next if path.size == 0
      has_file_access = false
      access[$user].each do |access_path|
        if !access_path # user has access to everything
          || (path.start_with? access_path) # access to this path
          has_file_access = true
        end
      end
      if !has_file_access
        puts "[POLICY] You do not have access to push to #{path}"
        exit 1
      end
    end
  end
end

check_directory_perms

```

Je krijgt een lijst met nieuwe commits die gepusht worden naar je server met `git rev-list`. Daarna vind je, voor elk van deze commits, de bestanden die aangepast worden en stelt vast of de gebruiker die pusht toegang heeft tot alle paden die worden aangepast.

Nu kunnen je gebruikers geen commits pushen met slecht vormgegeven berichten of met aangepaste bestanden buiten hun toegewezen paden.

## Het geheel testen

Als je nu `chmod u+x .git/hooks/update` aanroeft, wat het bestand is waar je al deze code zou moeten hebben gezet, en dan een commit probeert te pushen met een ongeldig bericht, krijg je zoets als dit:

```
$ git push -f origin master
Counting objects: 5, done.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 323 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Hier zijn een aantal interessante zaken. Allereerst, je ziet dit zodra de hook begint te lopen.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Onthoud dat je dit afgedrukt hebt helemaal aan het begin van je update script. Alles wat je script naar `stdout` echo't wordt naar het werkstation gestuurd.

Het volgende wat je zult opmerken is het foutbericht.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

De eerste regel was door jou afgedrukt, de andere twee waren van Git die je vertelt dat het update script niet-nul is geëindigd en dat dat hetgeen is wat je push afkeurt. Tot slot heb je dit:

```
To git@gitserver:project.git
 ! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Je zult een remote afwijsbericht zien voor elke referentie die je hook heeft afgewezen, en dat vertelt je specifiek dat het was afgewezen vanwege een hook-falen.

Verder, als iemand probeert een bestand te wijzigen waar ze geen toegang tot hebben en een commit pushen die zo een bestand bevat, zien ze iets vergelijkbaar. Bijvoorbeeld, als een documentatie auteur een commit probeert te pushen waarin hij iets als de `lib` directory wijzigt, ziet deze

```
[POLICY] You do not have access to push to lib/test.rb
```

Vanaf nu, zolang als dat **update** script aanwezig is en uitvoerbaar, zal je repository nooit een commit bericht zonder jouw patroon bevatten, en je gebruikers worden beperkt in hun vrijheid.

## Hooks aan de kant van het werkstation

Het nadeel van deze aanpak is het zeuren dat geheid zal beginnen zodra de commits van je gebruikers geweigerd worden. Het feit dat hun zorgzaam vervaardigde werk op het laatste moment pas geweigerd wordt kan enorm frustrerend en verwarrend zijn, ze zullen hun geschiedenis moeten aanpassen om het te corrigeren, wat niet altijd geschikt is voor de meer onzekere mensen.

Het antwoord op dit dilemma is een aantal werkstation hooks te leveren, die gebruikers kunnen gebruiken om hen te waarschuwen dat ze iets doen dat de server waarschijnlijk gaat weigeren. Op die manier kunnen ze alle problemen corrigeren voordat ze gaan committen en voordat die problemen lastiger te herstellen zijn. Omdat hooks niet overgebracht worden bij het klonen van een project, moet je deze scripts op een andere manier distribueren en je gebruikers ze in hun **.git/hooks** map laten zetten en ze uitvoerbaar maken. Je kunt deze hooks in je project of in een apart project distribueren, maar Git zal ze niet automatisch opzetten.

Om te beginnen zou je de commit boodschap moeten controleren vlak voordat iedere commit opgeslagen wordt, zodat je weet dat de server je wijzigingen niet gaat weigeren omdat de commit boodschap een verkeerd formaat heeft. Om dit te doen, kun je de **commit-msg** hook toevoegen. Als je dat de commit boodschap laat lezen uit het bestand dat als eerste argument opgegeven wordt, en dat vergelijkt met het patroon dan kun je Git dwingen om de commit af te breken als het niet juist is:

```
#!/usr/bin/env ruby
message_file = ARGV[0]
message = File.read(message_file)

$regex = /\[ref: (\d+)\]/

if !$regex.match(message)
  puts "[POLICY] Your message is not formatted correctly"
  exit 1
end
```

Als dat script op z'n plaats staat (in **.git/hooks/commit-msg**), uitvoerbaar is en je commit met een verkeerd geformateerd bericht, dan zie je dit:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

In dat geval is er geen commit gedaan. Maar als je bericht het juiste patroon bevat, dan staat Git je toe te committen:

```
$ git commit -am 'test [ref: 132]'  
[master e05c914] test [ref: 132]  
1 file changed, 1 insertions(+), 0 deletions(-)
```

Vervolgens wil je er zeker van zijn dat je geen bestanden buiten je ACL scope aanpast. Als de `.git` directory van je project een kopie van het ACL bestand bevat dat je eerder gebruikte, dan zal het volgende `pre-commit` script die beperkingen voor je controleren:

```
#!/usr/bin/env ruby  
  
$user      = ENV['USER']  
  
# [ insert acl_access_data method from above ]  
  
# only allows certain users to modify certain subdirectories in a project  
def check_directory_perms  
    access = get_acl_access_data('.git/acl')  
  
    files_modified = `git diff-index --cached --name-only HEAD`.split("\n")  
    files_modified.each do |path|  
        next if path.size == 0  
        has_file_access = false  
        access[$user].each do |access_path|  
            if !access_path || (path.index(access_path) == 0)  
                has_file_access = true  
            end  
            if !has_file_access  
                puts "[POLICY] You do not have access to push to #{path}"  
                exit 1  
            end  
        end  
    end  
end  
  
check_directory_perms
```

Dit is grofweg hetzelfde script als aan de server kant, maar met twee belangrijke verschillen. Als eerste staat het ACL bestand op een andere plek, omdat dit script vanuit je werkdirctory draait, en niet vanuit je `.git` directory. Je moet het pad naar het ACL bestand wijzigen van dit

```
access = get_acl_access_data('acl')
```

naar dit:

```
access = get_acl_access_data('.git/acl')
```

Het andere belangrijke verschil is de manier waarop je een lijst krijgt met bestanden die gewijzigd

is. Omdat de server kant methode naar de log van commits kijkt en nu je commit nog niet opgeslagen is, moet je de bestandslijst in plaats daarvan uit het staging area halen. In plaats van

```
files_modified = 'git log -1 --name-only --pretty=format:'' #{ref}'
```

moet je dit gebruiken

```
files_modified = 'git diff-index --cached --name-only HEAD'
```

Maar dat zijn de enige twee verschillen - verder werkt het script op dezelfde manier. Een aandachtspunt is dat het van je verlangt dat je lokaal werkt als dezelfde gebruiker als waarmee je pusht naar de remote machine. Als dat anders is, moet je de `$user` variabele handmatig instellen.

Het andere wat je moet doen is het controleren dat je niet probeert non-fast-forward referenties te pushen. Om een referentie te krijgen dat non-fast-forward is, moet je voorbij een commit rebasen die je al gepusht hebt, of een andere lokale branch naar dezelfde remote branch proberen te pushen.

We mogen aannemen dat de server 1 ingericht met `receive.denyDeletes` en `receive.denyNonFastForwards` om ditbeleid af te dwingen, dus het enige wat je kunt proberen af te vangen het abusievelijk rebasen van commits die je al gepusht hebt.

Hier is een voorbeeld pre-rebase script dat daarop controleert. Het haalt een lijst met alle commits die je op het punt staat te herschrijven, en controleert of ze al ergens bestaan in één van je remote referenties. Als het er een ziet die bereikbaar is vanuit een van je remote referenties, dan stopt het de rebase.

```

#!/usr/bin/env ruby

base_branch = ARGV[0]
if ARGV[1]
  topic_branch = ARGV[1]
else
  topic_branch = "HEAD"
end

target_shas = `git rev-list #{base_branch}..#{topic_branch}`.split("\n")
remote_refs = `git branch -r`.split("\n").map { |r| r.strip }

target_shas.each do |sha|
  remote_refs.each do |remote_ref|
    shas_pushed = `git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}`
    if shas_pushed.split("\n").include?(sha)
      puts "[POLICY] Commit #{sha} has already been pushed to #{remote_ref}"
      exit 1
    end
  end
end

```

Dit script gebruikt een syntax dat niet behandeld is in [Revisie Selectie](#). Je krijgt een lijst van commits die al gepusht zijn door dit uit te voeren:

```
'git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}'
```

De `SHA^@` syntax wordt vervangen door alle ouders van die commit. Je bent op zoek naar een commit die bereikbaar is vanuit de laatste commit op de remote en die niet bereikbaar is vanuit enige ouder van alle SHA's die je probeert te pushen - wat betekent dat het een fast-forward is.

Het grote nadeel van deze aanpak is dat het erg traag kan zijn en vaak onnodig is, als je de push niet probeert te forceren met de `-f` optie, dan zal de server je al waarschuwen en de push niet accepteren. Maar, het is een aardige oefening en kan je in theorie helpen om een rebase te voorkomen die je later zult moeten herstellen.

## Samenvatting

We hebben de meeste van de belangrijkste manieren behandeld waarop je jouw Git werkstation en server kunt aanpassen zodat deze jouw workflow en projecten het beste ondersteunt. Je hebt over allerhande soorten configuratie instellingen kunnen lezen, bestands-gebaseerde attributen en event hooks, en je hebt een voorbeeld beleidsbewakende server gebouwd. Je zou nu in staat moeten zijn om Git in bijna alle soorten workflows die je kunt verzinnen in te passen.

# Git en andere systemen

Het is geen perfecte wereld. Meestal kan je niet meteen elk project waar je mee in aanraking komt omzetten naar Git. Soms zit je vast op een project dat een ander VCS gebruikt, en wensen dat het Git zou zijn. We zullen het eerste deel van dit hoofdstuk hebben over over manieren om Git als client te gebruiken als het project waar je op werkt op een ander systeem wordt gehost.

Op een gegeven moment zal je een bestaande project misschien willen omzetten naar Git. Het tweede gedeelte van dit hoofdstuk beschrijft hoe je projecten naar Git kunt migreren uit verschillende specifieke systemen, zowel als een manier die je kan helpen als er geen standaard import hulpmiddelen zijn.

## Git als een client

Git biedt zo'n prettige ervaring voor ontwikkelaars dat veel mensen een manier hebben gevonden om het te gebruiken op hun werkstation, zelfs als de rest van hun team een compleet andere VCS gebruikt. Er zijn een aantal van deze adapters, die "bridges" worden genoemd, beschikbaar. We zullen hier degenen behandelen die je het meest waarschijnlijk in de praktijk zult tegenkomen.

### Git en Subversion

Een groot deel van open source ontwikkel projecten en een groot aantal van bedrijfsprojecten gebruiken Subversion om hun broncode te beheren. Het bestaat meer dan 10 jaar en voor een groot gedeelte van die tijd was het de *de facto* VCS keuze voor open source projecten. Het lijkt in vele aspecten ook erg op CVS, die daarvoor een grote naam was in de source-control wereld.

Een van de mooie functies van Git is de bidirectionele brug naar Subversion genaamd `git svn`. Dit instrument staat je toe om Git te gebruiken als een volwaardig client naar een Subversion server, zodat je alle lokale mogelijkheden van Git kunt gebruiken en dan naar een Subversion server kunt pushen alsof je lokaal Subversion gebruikt. Dit houdt in dat je lokaal kunt branchen en mergen, de staging area kunt gebruiken, rebasing en cherry-picking kunt gebruiken, enzovoorts, terwijl de mensen waarmee je samenwerkt blijven werken met hun middelen uit de stenen tijdperk. Het is een goede manier om Git in de bedrijfsmilieu te smokkelen en je mede-ontwikkelaars te helpen om effectiever te worden terwijl jij een lobby voert om de infrastructuur zover te krijgen dat Git volledig wordt ondersteund. De Subversion bridge is de manier om naar de DVCS wereld te groeien.

#### `git svn`

Het basis commando in Git voor alle Subversion bridging commando's is `git svn`. Het accepteert best wel veel commando's, dus we laten de meest gebruikte zien terwijl we een aantal eenvoudige workflows behandelen.

Het is belangrijk om op te merken dat wanneer je `git svn` gebruikt, je interacteert met Subversion, wat een systeem is dat behoorlijk anders werkt dan Git. Alhoewel je lokaal **kunt** branchen en mergen, is het over het algemeen het beste om je historie zo lineair als mogelijk te houden door je werk te rebasen, en te vermijden dat je zaken doet als het tegelijkertijd interacteren met een remote repository in Git.

Ga niet je historie overschrijven en dan weer proberen te pushen, en push niet tegelijk naar een parallelle Git repository om samen te werken met andere Git ontwikkelaars. Subversion kan alleen maar een lineaire historie aan, en het is eenvoudig om het in de war te brengen. Als je met een team samenwerkt, en sommigen gebruiken SVN en anderen gebruiken Git, zorg er dan voor dat iedereen de SVN server gebruikt om samen te werken - als je dit doet wordt je leven een stuk aangenamer.

## Inrichten

Om deze functionaliteit te laten zien, heb je een typische SVN repository nodig waar je schrijfrechten op hebt. Als je deze voorbeelden wilt kopiëren, moet je een schrijfbare kopie maken van een SVN test repository. Om dat eenvoudig te doen, kan je een tool **svnsync** genaamd gebruiken die bij Subversion wordt geleverd.

Om het te volgen, moet je eerst een nieuwe lokale Subversion repository maken:

```
$ mkdir /tmp/test-svn  
$ svnadmin create /tmp/test-svn
```

Daarna moet je alle gebruikers toestaan om revprops te wijzigen - een makkelijke manier is om een **pre-revprop-change** toe te voegen die altijd met 0 afsluit:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change  
#!/bin/sh  
exit 0;  
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Je kunt dit project nu synchroniseren naar je lokale machine door **svnsync init** aan te roepen met de naar en van repositories.

```
$ svnsync init file:///tmp/test-svn \  
http://your-svn-server.example.org/svn/
```

Dit richt de properties in om de synchronisatie te laten lopen. Je kunt dan de code clonen door het volgende te doen

```
$ svnsync sync file:///tmp/test-svn  
Committed revision 1.  
Copied properties for revision 1.  
Transmitting file data .....[...]  
Committed revision 2.  
Copied properties for revision 2.  
[...]
```

Alhoewel deze handeling maar enkele minuten in beslag neemt, zal het proces, als je de orginele repository naar een andere remote repository probeert te kopiëren, bijna een uur in beslag nemen,

zelfs als er minder dan 100 commits zijn. Subversion moet een revisie per keer kopiëren en deze dan naar de andere repository pushen - het is belachelijk inefficiënt, maar het is de enige makkelijke manier om dit te doen.

## Aan de gang gaan

Nu je een Subversion repository hebt waar je schrijfrechten op hebt, kan je een typische workflow gaan volgen. Je begint met het `git svn clone` commando, die een hele Subversion repository importeert naar een lokale Git repository. Onthoud dat als je van een echte gehoste Subversion repository importeert, je de `file:///tmp/test-svn` moet vervangen met de URL van je Subversion repository:

```
$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /private/tmp/progit/test-svn/.git/
r1 = dcfb5891860124cc2e8cc616cded42624897125 (refs/remotes/origin/trunk)
  A  m4/acx_pthread.m4
  A  m4/stl_hash.m4
  A  java/src/test/java/com/google/protobuf/UnknownFieldSetTest.java
  A  java/src/test/java/com/google/protobuf/WireFormatTest.java
...
r75 = 556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae (refs/remotes/origin/trunk)
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/my-calc-branch, 75
Found branch parent: (refs/remotes/origin/my-calc-branch)
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae
Following parent with do_switch
Successfully followed parent
r76 = 0fb585761df569eaecd8146c71e58d70147460a2 (refs/remotes/origin/my-calc-branch)
Checked out HEAD:
  file:///tmp/test-svn/trunk r75
```

Dit roept het equivalent van twee commando's aan - `git svn init` gevolgd door `git svn fetch` - op de URL die je opgeeft. Dit kan even duren. Bijvoorbeeld, als het project maar ongeveer 75 commits heeft en de codebase is niet zo groot, maar Git moet elke versie uitchecken, een voor een, en deze allemaal individueel committen. Voor een project met honderden of duizenden commits, kan dit letterlijk uren of zelfs dagen in beslag nemen voor het klaar is.

Het `-T trunk -b branches -t tags` gedeelte vertelt Git dat deze Subversion repository de normale branch en tag conventies volgt. Als je jouw trunk, branches of tags andere namen geeft, kan je deze opties veranderen. Omdat dit zo gewoonlijk is, kan je dit gehele gedeelte vervangen met `-s`, wat standaard indeling betekent en al die opties impliceert. Het volgende commando doet hetzelfde:

```
$ git svn clone file:///tmp/test-svn -s
```

Op dit punt zou je een valide Git repository moeten hebben die jouw branches en tags heeft geïmporteerd.

```
$ git branch -a
* master
  remotes/origin/my-calc-branch
  remotes/origin/tags/2.0.2
  remotes/origin/tags/release-2.0.1
  remotes/origin/tags/release-2.0.2
  remotes/origin/tags/release-2.0.2rc1
  remotes/origin/trunk
```

Merk op hoe dit instrument Subversion tags als remote refs beheert. Laten we het Git binnenwerk commando `show-ref` gebruiken om het iets nauwkeuriger te bekijken:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git doet dit niet als het van een Git server kloont; zo ziet een repository met tags eruit na een verse clone:

```
$ git show-ref
c3dcbe8488c6240392e8a5d7553bbffcb0f94ef0 refs/remotes/origin/master
32ef1d1c7cc8c603ab78416262cc421b80a8c2df refs/remotes/origin/branch-1
75f703a3580a9b81ead89fe1138e6da858c5ba18 refs/remotes/origin/branch-2
23f8588dde934e8f33c263c6d8359b2ae095f863 refs/tags/v0.1.0
7064938bd5e7ef47bfd79a685a62c1e2649e2ce7 refs/tags/v0.2.0
6dcb09b5b57875f334f61aebed695e2e4193db5e refs/tags/v1.0.0
```

Git fetched de tags direct naar `refs/tags`, in plaats van ze te behandelen als remote branches.

## Terug naar Subversion committen

Nu je een werkende repository hebt, kan je wat werk doen op het project en je commits terug stroomopwaarts pushen, waarbij je Git feitelijk als een SVN client gebruikt. Als je een van de bestanden hebt gewijzigd en deze commit, heb je een commit die lokaal in Git bestaat, maar die niet op de Subversion server bestaat:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

Nu moet je jouw wijziging stroomopwaarts pushen. Merk op dat dit de manier waarop je met

Subversion werkt wijzigt - je kunt verschillende commits offline doen en ze dan allemaal in een keer naar de Subversion server pushen. Om naar een Subversion server te pushen, roep je het `git svn dcommit` commando aan:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M README.txt
Committed r77
M README.txt
r77 = 95e0222ba6399739834380eb10afcd73e0670bc5 (refs/remotes/origin/trunk)
No changes between 4af61fd05045e07598c553167e0f31c84fd6ffe1 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Dit pakt alle commits die je hebt gemaakt bovenop de Subversion server code, maakt een Subversion commit voor elk van deze, en herschrijft je lokale Git commit om een unieke referentienummer in te voegen. Dit is belangrijk omdat dit betekent dat al de SHA-1 checksums voor je lokale commits zal wijzigen. Deels om deze reden, is het werken met Git-gebaseerde remote versies van je project tegelijk met een Subversion server geen goed idee. Als je naar de laatste commit kijkt, kan je het nieuwe `git-svn-id` zien die was toegevoegd:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Merk op dat de SHA-1 checksum die oorspronkelijk begon met `4af61fd` toen je ging committen nu begint met `95e0222`. Als je zowel naar een Git server als een Subversion server wilt pushen, moet je eerst aan de Subversion server pushen (`dcommit`), omdat deze actie je commit gegevens wijzigt.

## Nieuwe wijzigingen pullen

Als je met andere ontwikkelaars werkt, dan zal op een gegeven moment iemand van jullie gaan pushen, en dan zal de ander een wijziging proberen te pushen die conflicteert. Die wijziging zal afgewezen worden totdat je hun werk merget. In `git svn` ziet dit er zo uit:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: d5837c4b461b7c0e018b49d12398769d2bfc240a and refs/remotes/origin/trunk differ,
using rebase:
:100644 100644 f414c433af0fd6734428cf9d2a9fd8ba00ada145
c80b6127dd04f5fcda218730ddf3a2da4eb39138 M README.txt
Current branch master is up to date.
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Om deze situatie op te lossen, kan je `git svn rebase` uitvoeren, die alle wijzigingen op de server pult die je nog niet hebt, en rebaseset al het werk dat je hebt bovenop hetgeen op de server is:

```
$ git svn rebase
Committing to file:///tmp/test-svn/trunk ...

ERROR from SVN:
Transaction is out of date: File '/trunk/README.txt' is out of date
W: eaa029d99f87c5c822c5c29039d19111ff32ef46 and refs/remotes/origin/trunk differ,
using rebase:
:100644 100644 65536c6e30d263495c17d781962cff12422693a
b34372b25ccf4945fe5658fa381b075045e7702a M README.txt
First, rewinding head to replay your work on top of it...
Applying: update foo
Using index info to reconstruct a base tree...
M README.txt
Falling back to patching base and 3-way merge...
Auto-merging README.txt
ERROR: Not all changes have been committed into SVN, however the committed
ones (if any) seem to be successfully integrated into the working tree.
Please see the above messages for details.
```

Nu is al jouw werk uitgevoerd bovenop hetgeen wat op de Subversion server staat, dus je kunt met goed gevolg `dcommit` doen:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
  M README.txt
Committed r85
  M README.txt
r85 = 9c29704cc0bbbed7bd58160cfb66cb9191835cd8 (refs/remotes/origin/trunk)
No changes between 5762f56732a958d6cfda681b661d2a239cc53ef5 and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Merk op dat, in tegenstelling tot Git die vereist dat je werk van stroomopwaarts dat je lokaal nog niet hebt merget voordat je kunt pushen, `git svn` je dat alleen verplicht te doen als de wijzigingen conflicteren (vergelijkbaar met hoe Subversion werkt). Als iemand een wijziging op een bestand pushed en jij pushed een wijziging op een ander bestand, zal je `dcommit` prima werken:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
  M configure.ac
Committed r87
  M autogen.sh
r86 = d8450bab8a77228a644b7dc0e95977ffc61adff7 (refs/remotes/origin/trunk)
  M configure.ac
r87 = f3653ea40cb4e26b6281cec102e35dcba1fe17c4 (refs/remotes/origin/trunk)
W: a0253d06732169107aa020390d9fefdb2b1d92806 and refs/remotes/origin/trunk differ,
using rebase:
:100755 100755 efa5a59965fb5b2b0a12890f1b351bb5493c18
e757b59a9439312d80d5d43bb65d4a7d0389ed6d M autogen.sh
First, rewinding head to replay your work on top of it...
```

Dit is belangrijk om te onthouden, omdat de uitkomst een project status is die niet eerder bestond op een van jullie computers toen jij pushde. Als de wijzigingen niet compatible zijn, maar geen conflict veroorzaken, kan je problemen krijgen die moeilijk te diagnostiseren zijn. Dit verschilt met de manier van werken met een Git server - in Git kan je de situatie op je lokale werkstation testen voordat je het publiceert, terwijl in SVN, je er nooit zeker van kunt zijn dat de situatie direct voor en na een commit gelijk zijn.

Je kunt ook dit commando aanroepen om wijzigingen binnen te halen van de Subversion server, zelfs als je zelf nog niet klaar bent om te committen. Je kunt `git svn fetch` aanroepen om de nieuwe gegevens te pakken, maar `git svn rebase` doet de fetch en werkt daarna je lokale commits bij.

```
$ git svn rebase
  M autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Regelmatig `git svn rebase` aanroepen verzekert je ervan dat je code altijd is bijgewerkt. Je moet er

echter zeker van zijn dat je werk directory schoon is als je dit aanroept. Als je lokale wijzigingen hebt, moet je je werk stashen of tijdelijk committen voordat je `git svn rebase` aanroept - anders zal het commando stoppen als het ziet dat de rebase in een merge conflict zal resulteren.

## Git branching problemen

Als je op je gemak voelt met een Git workflow, zal je waarschijnlijk topic branches maken, daar werk op doen en ze dan weer in mergen. Als je aan een Subversion server pushed met `git svn`, dan is het waarschijnlijk verstandig om je werk elke keer op een enkele branch te rebasen in plaats van branches samen te mergen. De achterliggende reden om rebases te gebruiken is dat Subversion een lineaire historie kent en niet met merges omgaat zoals Git dit doet, dus `git svn` volgt alleen de eerste ouder als het de snapshots naar Subversion commits converteert.

Stel dat je historie er als volgt uitziet: je hebt een `experiment`-branch gemaakt, heb daar twee commits gedaan, en deze daarna terug in `master` gemerged. Als je `dcommit` doet, zie je uitvoer als dit:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M  CHANGES.txt
Committed r89
M  CHANGES.txt
r89 = 89d492c884ea7c834353563d5d913c6adf933981 (refs/remotes/origin/trunk)
M  COPYING.txt
M  INSTALL.txt
Committed r90
M  INSTALL.txt
M  COPYING.txt
r90 = cb522197870e61467473391799148f6721bcf9a0 (refs/remotes/origin/trunk)
No changes between 71af502c214ba13123992338569f4669877f55fd and
refs/remotes/origin/trunk
Resetting to the latest refs/remotes/origin/trunk
```

Het aanroepen van `dcommit` op een branch met gemergde historie werkt prima, behalve als je naar je Git project historie kijkt, heeft het geen van beide commits die je op de `experiment`-branch gemaakt hebt herschreven - in plaats daarvan komen al deze wijzigingen als een enkele merge commit in de SVN versie.

Als iemand anders dat werk kloont, is alles wat ze zien de merge commit met al het werk erin gepropst, alsof je `git merge --squash` aangeroepen hebt; ze zien niet de commit gegevens over waar het vandaan kwam of wanneer het was gecommit.

## Subversion Branching

Branches maken in Subversion is niet hetzelfde als branches maken in Git; als je kunt voorkomen dat je het vaak doet, is dat eigenlijk wel het beste. Echter, je kunt in Subversion branches maken en ernaar committen met `git svn`.

## Een nieuwe SVN branch maken

Om een nieuwe branch in Subversion te maken, roep je `git svn branch [branchnaam]` aan:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r90 to file:///tmp/test-svn/branches/opera...
Found possible branch point: file:///tmp/test-svn/trunk => file:///tmp/test-
svn/branches/opera, 90
Found branch parent: (refs/remotes/origin/opera)
cb522197870e61467473391799148f6721bcf9a0
Following parent with do_switch
Successfully followed parent
r91 = f1b64a3855d3c8dd84ee0ef10fa89d27f1584302 (refs/remotes/origin/opera)
```

Dit is het equivalent van het `svn copy trunk branches/opera` commando in Subversion en wordt uitgevoerd op de Subversion server. Het is belangrijk op te merken dat dit je niet uitcheckt in die branch; als je op dat moment gaat committen, dan zal die commit naar `trunk` gaan op de server, niet `opera`.

## Actieve branches switchen

Git probeert uit te vinden naar welke branch je dcommits gaan door te kijken naar de punt van al je Subversion branches in je historie - je zou er maar een moeten hebben, en het zou de laatste moeten zijn met een `git-svn-id` in je huidige branch historie.

Als je op meer dan een branch tegelijk wilt werken, kan je lokale branches inrichten om te `dcommit`-ten naar specifieke Subversion branches door ze te beginnen op de geïmporteerde Subversion commit voor die branch. Als je een `opera`-branch wilt waar je apart op kunt werken, kan je het volgende aanroepen:

```
$ git branch opera remotes/origin/opera
```

Vervolgens, als je je `opera`-branch wilt mergen naar `trunk` (je `master`-branch), kan je dat doen met een gewone `git merge`. Maar als je een beschrijvende commit bericht (via `-m`) moet meegeven, anders zal de merge “Merge branch `opera`” vermelden in plaats van iets nuttigs.

Onthoud dat hoewel je `git merge` gebruikt om deze handeling uit te voeren, en de merge waarschijnlijk veel makkelijker zal zijn dan het in Subversion zou zijn (omdat Git automatisch de juiste merge basis voor je zal uitzoeken), dit geen normale Git merge commit is. Je zult deze gegevens naar een Subversion server terug moeten pushen die niet in staat is een commit te verwerken die naar meer dan een ouder terug te herleiden is; dus, nadat je het gepusht hebt, zal het eruit zien als een enkele commit waarin al het werk van een andere branch is gepropt onder een enkele commit. Nadat je een branch in de een andere hebt gemerged, kan je niet simpelweg doorgaan met werken op die branch, zoals je in Git zou doen. Het `dcommit` commando dat je hebt aangeroepen verwijdert alle informatie die aangeeft welke branch erin was gemerged, dus daarop volgende merge-basis berekeningen zullen fout gaan - de dcommit maakt dat je `git merge` resultaat eruit ziet alsof je `git merge --squash` had aangeroepen. Jammergenoeg is er geen goede manier om deze situatie te vermijden - Subversion kan deze informatie niet opslaan, dus je zult altijd

gehinderd worden door de beperkingen zolang je het gebruikt als je server. Om problemen te voorkomen, moet je de lokale branch verwijderen (in dit geval, `opera`) nadat je het in de trunk hebt gemerged.

## Subversion commando's

De `git svn` toolset biedt een aantal commando's om de overgang naar Git te vergemakkelijken door wat functionaliteit te leveren die vergelijkbaar is met wat je in Subversion had. Hier zijn een paar commando's die je geven wat Subversion normaal gesproken deed.

### Historie op de manier van SVN

Als je Subversion gewend bent en je wilt je historie zien op de uitvoermanier van SVN, kan je `git svn log` aanroepen om je commit historie in SVN formaat te zien.

```
$ git svn log
-----
r87 | schacon | 2014-05-02 16:07:37 -0700 (Sat, 02 May 2014) | 2 lines
autogen change

-----
r86 | schacon | 2014-05-02 16:00:21 -0700 (Sat, 02 May 2014) | 2 lines
Merge branch 'experiment'

-----
r85 | schacon | 2014-05-02 16:00:09 -0700 (Sat, 02 May 2014) | 2 lines
updated the changelog
```

Je moet twee belangrijke dingen weten over `git svn log`. Ten eerste, het werkt offline, in tegenstelling tot het echte `svn log` commando, die de Subversion server om de gegevens vraagt. Ten tweede, het laat je alleen commits zien die gecommit zijn naar de Subversion server. Lokale Git commits die je niet ge-dcommit hebt worden niet getoond; noch de commits die mensen in de tussentijd naar de Subversion server hebben gemaakt. Je moet het meer zien als de laatst bekende stand van commits op de Subversion server.

### SVN annotatie

Net zoals het `git svn log` commando het `svn log` commando offline simuleert, kan je het equivalent van `svn annotate` krijgen door `git svn blame [FILE]` aan te roepen. De uitvoer ziet er als volgt uit:

```
$ git svn blame README.txt
 2 temporal Protocol Buffers - Google's data interchange format
 2 temporal Copyright 2008 Google Inc.
 2 temporal http://code.google.com/apis/protocolbuffers/
 2 temporal
22 temporal C++ Installation - Unix
22 temporal =====
 2 temporal
79 schacon Committing in git-svn.
78 schacon
 2 temporal To build and install the C++ Protocol Buffer runtime and the Protocol
 2 temporal Buffer compiler (protoc) execute the following:
 2 temporal
```

Nogmaals, het laat je niet de commits zien die je lokaal in Git gemaakt hebt of die in de tussentijd naar Subversion zijn gepusht.

### SVN server informatie

Je kunt ook dezelfde soort informatie krijgen die `svn info` je geeft door `git svn info` aan te roepen:

```
$ git svn info
Path: .
URL: https://schacon-test.googlecode.com/svn/trunk
Repository Root: https://schacon-test.googlecode.com/svn
Repository UUID: 4c93b258-373f-11de-be05-5f7a86268029
Revision: 87
Node Kind: directory
Schedule: normal
Last Changed Author: schacon
Last Changed Rev: 87
Last Changed Date: 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009)
```

Dit is gelijk aan `blame` en `log` in die zin dat het offline loopt en dat het alleen is bijgewerkt tot de laatste keer dat je met de Subversion server contact had.

### Negeren wat Subversion negeert

Als je een Subversion repository cloont die een `svn:ignore` property ergens heeft, zal je waarschijnlijk vergelijkbare `.gitignore` bestanden willen krijgen zodat je niet per ongeluk bestanden commit die je niet had moeten doen. `git svn` heeft twee commando's die je helpen met dit scenario. De eerste is `git svn create-ignore`, die automatisch vergelijkbare `.gitignore` bestanden voor je maakt zodat je volgende commit deze kan bevatten.

Het tweede commando is `git svn show-ignore`, die de regels die je in een `.gitignore` bestand moet zetten naar stdout uitvoert, zodat je deze uitvoer naar je het exclusie bestand in je project kunt leiden:

```
$ git svn show-ignore > .git/info/exclude
```

Op deze manier, vervuil je het project niet met `.gitignore` bestanden. Dit is een goed alternatief als je de enige Git gebruiker in een Subversion team bent, en je teamgenoten geen `.gitignore` bestanden in het project willen hebben.

## Git-Svn samenvatting

De `git svn` instrumenten zijn nuttig als je vastzit aan een Subversion server, of op een andere manier in een ontwikkelteam zit waar het gebruik van een Subversion server noodzakelijk is. Je moet het echter als een gemankeerde Git beschouwen, of je loopt al snel tegen terminologieverschillen aan die jou en je medewerkers zullen verwarreren. Probeer, om niet in de problemen te komen, de volgende richtlijnen te volgen:

- Houd een lineaire Git historie aan die geen merge commits bevat die door `git merge` zijn aangemaakt. Rebasing al het werk die je buiten je hoofd-branch doet hierop terug; merge het niet in.
- Richt niets in voor het samenwerken op een aparte Git server, en werk niet samen met zo'n server. Je kunt er misschien een bijhouden om clones voor nieuwe ontwikkelaars te versnellen, maar ga er niets naar pushen dat geen `git-svn-id` regel heeft. Je zou zelfs een `pre-receive` hook kunnen aanmaken die controleert dat elke commit bericht op een `git-svn-id` controleert en pushes afwijst die commits bevatten die dit niet hebben.

Als je deze richtlijnen volgt, wordt het werken met een Subversion server misschien iets dragelijker. Echter, als het ook maar enigszins mogelijk is om naar een echte Git server te gaan, zal dit je team veel meer opleveren.

## Git en Mercurial

Het DVCS universum is groter dan alleen Git. Eigenlijk zijn er vele andere systemen in deze ruimte, elk met hun eigen aanpak om hoe op de juiste manier om te gaan met gedistribueerd versiebeheer. Buiten Git, is de meest populaire Mercurial, en de twee zijn op vele vlakken erg vergelijkbaar.

Het goede nieuws is, als je het gedrag van Git aan de kant van het werkstation de voorkeur geeft, maar als je werkt met een project waarvan de broncode wordt beheerd met Mercurial, dat er een manier is om Git als een client voor een repository die op een Mercurial-host draait te gebruiken. Omdat de manier voor Git om te praten met server repositories via remotes loopt, moet het niet als een verrassing komen dat deze brug ("bridge") geïmplementeerd is als een remote helper. De naam van het project is `git-remote-hg`, en het kan worden gevonden op <https://github.com/felipec/git-remote-hg>.

### git-remote-hg

Allereerst moet je `git-remote-hg` installeren. Dit houdt niet veel meer in dan het bestand ergens op je pad neer te zetten, op deze manier:

```
$ curl -o ~/bin/git-remote-hg \
https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

...aangenomen dat `~/bin` op je `$PATH` staat. Git-remote-hg heeft een andere afhankelijkheid: de `mercurial` library voor Python. Als je Python geïnstalleerd hebt, is dit zo simpel als:

```
$ pip install mercurial
```

(Als je geen Python geïnstalleerd hebt, bezoek dan <https://www.python.org/> en haal dat eerst op.)

Het laatste wat je nodig hebt is de Mercurial client. Ga naar <https://www.mercurial-scm.org/> en installeer dit als je dat al niet hebt gedaan.

Nu is alles klaar voor gebruik. Al wat je nodig hebt is een Mercurial repository waar je naar kunt pushen. Gelukkig kan elke Mercurial repository zich op deze manier gedragen, dus we hoeven alleen de "hello world" repository die iedereen gebruikt om Mercurial te leren gebruiken:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

## Aan de gang gaan

Nu we een passende "server-side" repository hebben, kunnen we een normale workflow gaan volgen. Zoals je zult zien zijn deze twee systemen voldoende vergelijkbaar dat er niet veel wrijving zal zijn.

Zoals altijd met Git, gaan we eerst clonen:

```
$ git clone hg:///tmp/hello /tmp/hello-git
$ cd /tmp/hello-git
$ git log --oneline --graph --decorate
* ac7955c (HEAD, origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master, master) Create a
makefile
* 65bb417 Create a standard "hello, world" program
```

Je zult opgemerkt hebben dat je bij het werken met een Mercurial repository het standaard `git clone` commando gebruikt. Dat is omdat git-remote-hg op een behoorlijk laag niveau werkt, en gebruik maakt van een vergelijkbaar mechanisme als het HTTP/S protocol dat in Git geïmplementeerd is (remote helpers). Omdat Git en Mercurial beide zijn ontworpen vanuit het principe dat elk werkstation een volledige kopie van de historie van de repository heeft, maakt dit commando een volledige kloon, inclusief alle historie van het project en doet dit redelijk snel.

Het log commando laat twee commits zien, maar naar de laatste daarvan wordt door een hele sloot refs verwezen. Nu is het zo dat een aantal van deze er eigenlijk helemaal niet zijn. Laten we kijken naar wat er eigenlijk in de `.git` directory staat:

```
$ tree .git/refs
.git/refs
├── heads
│   └── master
├── hg
│   └── origin
│       ├── bookmarks
│       │   └── master
│       └── branches
│           └── default
└── notes
    └── hg
└── remotes
    └── origin
        └── HEAD
└── tags
```

9 directories, 5 files

Git-remote-hg probeert dingen meer letterlijk op z'n Gits te maken, maar onder de motorkap beheert het de conceptuele mapping tussen twee marginaal verschillende systemen. De `refs/hg` directory is waar de echte remote refs worden opgeslagen. Bijvoorbeeld, de `refs/hg/origin/branche/default` is een Git ref bestand die de SHA-1 bevat die begint met "ac7955c", wat de commit is waar `master` naar wijst. Dus de `refs/hg` directory is een soort van nep `refs/remotes/origin`, maar het heeft het toegevoegde onderscheid tussen boekenleggers ("bookmarks") en branches.

Het `notes/hg` bestand is het beginpunt van hoe git-remote-hg Git commit hashes mapt op Mercurial changeset IDs. Laten we dit een beetje verkennen:

```
$ cat notes/hg
d4c10386...

$ git cat-file -p d4c10386...
tree 1781c96...
author remote-hg <> 1408066400 -0800
committer remote-hg <> 1408066400 -0800

Notes for master

$ git ls-tree 1781c96...
100644 blob ac9117f... 65bb417...
100644 blob 485e178... ac7955c...

$ git cat-file -p ac9117f
0a04b987be5ae354b710cefeba0e2d9de7ad41a9
```

Dus `refs/notes/hg` wijst naar een tree, wat in de Git object database een lijst van andere objecten

met namen is. `git ls-tree` produceert de mode, type, object hash en bestandsnamen voor items in een tree. Zodra we gaan graven naar een van de tree items, vinden we dat hierbinnen een blog zit met de naam “ac9117f” (de SHA-1 hash van de commit waar `master` naar wijst), met de inhoud “0a04b98” (wat de ID is van de Mercurial changeset aan de punt van de `default`-branch).

Het goede nieuwe is dat we ons over het algemeen hierover geen zorgen hoeven te maken. De typische workflow zal niet veel verschillen van het werken met een Git remote.

Er is een extra onderwerp waar we even aandacht aan moeten schenken voordat we doorgaan: ignores. Mercurial en Git gebruiken een erg vergelijkbare mechanisme hiervoor, maar het is erg aannemelijk dat je een `.gitignore` niet echt in een Mercurial repository wilt committen. Gelukkig heeft Git een manier om bestanden te negeren die lokaal is voor een repository die op schijf staat, en het formaat van Mercurial is compatibel met die van Git, dus je moet het ernaartoe kopiëren:

```
$ cp .hgignore .git/info/exclude
```

Het `.git/info/exclude` bestand gedraagt zich als een `.gitignore`, maar wordt niet in commits meegenomen.

## Workflow

Laten we aannemen dat we wat werk gedaan hebben en een aantal commits op de `master`-branch uitgevoerd hebben, en je bent klaar om dit naar de remote repository te pushen. Zo ziet onze repository eruit op dit moment:

```
$ git log --oneline --graph --decorate
* ba04a2a (HEAD, master) Update makefile
* d25d16f Goodbye
* ac7955c (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Create a makefile
* 65bb417 Create a standard "hello, world" program
```

Onze `master`-branch loopt twee commits voor op `origin/master`, maar deze twee commits bestaan alleen op onze lokale machine. Laten we kijken of iemand anders belangrijk werk heeft gedaan in de tussentijd:

```

$ git fetch
From hg:::/tmp/hello
 ac7955c..df85e87 master      -> origin/master
 ac7955c..df85e87 branches/default -> origin/branches/default
$ git log --oneline --graph --decorate --all
* 7b07969 (refs/notes/hg) Notes for default
* d4c1038 Notes for master
* df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
| * ba04a2a (HEAD, master) Update makefile
| * d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program

```

Omdat we de `--all` vlag gebruikt hebben, zien we de “notes” refs die intern gebruikt worden door git-remote-hg, maar deze kunnen we negeren. De rest is zoals we hadden verwacht; `origin/master` is een commit naar voren gegaan, en onze histories zijn nu uiteengelopen. In tegenstelling tot andere systemen waar we mee werken in dit hoofdstuk, is Mercurial in staat om merges te verwerken, dus we gaan nu geen moeilijke dingen doen.

```

$ git merge origin/master
Auto-merging hello.c
Merge made by the 'recursive' strategy.
 hello.c | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
* 0c64627 (HEAD, master) Merge remote-tracking branch 'origin/master'
|\
| * df85e87 (origin/master, origin/branches/default, origin/HEAD,
refs/hg/origin/branches/default, refs/hg/origin/bookmarks/master) Add some
documentation
* | ba04a2a Update makefile
* | d25d16f Goodbye
|/
* ac7955c Create a makefile
* 65bb417 Create a standard "hello, world" program

```

Perfect. We laten de tests draaien en alles slaagt, dus we zijn klaar om ons werk te delen met de rest van het team:

```

$ git push
To hg:::/tmp/hello
 df85e87..0c64627 master -> master

```

Dat is alles! Als je de Mercurial repository bekijkt, zul je zien dat dit gedaan heeft wat we mogen

verwachten:

```
$ hg log -G --style compact
o 5[tip]:4,2 dc8fa4f932b8 2014-08-14 19:33 -0700 ben
|\ Merge remote-tracking branch 'origin/master'
|
| o 4 64f27bcefc35 2014-08-14 19:27 -0700 ben
| | Update makefile
|
| o 3:1 4256fc29598f 2014-08-14 19:27 -0700 ben
| | Goodbye
|
@ | 2 7db0b4848b3c 2014-08-14 19:30 -0700 ben
| / Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
| Create a standard "hello, world" program
```

De wijzigingsset ("changeset") met nummer 2 is gemaakt door Mercurial, en de changesets nummers 3 en 4 zijn door git-remote-hg gemaakt, door het pushen van de met Git gemaakte commits.

### Branches en Boekenleggers ("Bookmarks")

Git heeft maar een soort branch: een referentie die verplaatst wordt als commits worden gemaakt. In Mercurial, worden deze soorten referenties een “bookmark” genoemd, en het gedraagt zich grotendeels vergelijkbaar met een branch in Git.

Het concept van een “branch” in Mercurial heeft iets meer voeten in de aarde. De branch waar een changeset op is gebaseerd wordt opgeslagen *met de changeset*, wat inhoudt dat het altijd in de historie van de repository aanwezig is. Hier is een voorbeeld van een commit die gemaakt is op de **develop**-branch:

```
$ hg log -l 1
changeset: 6:8f65e5e02793
branch: develop
tag: tip
user: Ben Straub <ben@straub.cc>
date: Thu Aug 14 20:06:38 2014 -0700
summary: More documentation
```

Merk de regel op die begint met “branch”. Git kan dit niet echt simuleren (en hoeft dit ook niet; beide soorten branches kunnen in Git als een ref worden weergegeven), maar git-remote-hg moet het onderscheid kunnen maken, omdat Mercurial hier om geeft.

Het aanmaken van Mercurial bookmarks is net zo eenvoudig als het maken van Git branches. Aan de Git kant:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg:///tmp/hello
 * [new branch]      featureA -> featureA
```

En dat is alles wat nodig is. Aan de kant van Mercurial ziet het er zo uit:

```
$ hg bookmarks
    featureA           5:bd5ac26f11f9
$ hg log --style compact -G
@ 6[tip] 8f65e5e02793 2014-08-14 20:06 -0700  ben
| More documentation
|
o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700  ben
|\ Merge remote-tracking branch 'origin/master'
|
o 4 0434aaa6b91f 2014-08-14 20:01 -0700  ben
| update makefile
|
o 3:1 318914536c86 2014-08-14 20:00 -0700  ben
| goodbye
|
o | 2 f098c7f45c4f 2014-08-14 20:01 -0700  ben
| / Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700  mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700  mpm
| Create a standard "hello, world" program
```

Merk de nieuwe **[featureA]** tag in revisie 5 op. Deze gedragen zich precies als Git branches aan de Git kant, met een uitzondering: je kunt een bookmark niet van de Git kant verwijderen (dit is een beperking van remote helpers).

Je kunt ook op een “zwaargewicht” Mercurial branch werken: gewoon een branch in de **branches** naamsruimte (“namespace”) zetten:

```
$ git checkout -b branches/permanent
Switched to a new branch 'branches/permanent'
$ vi Makefile
$ git commit -am 'A permanent change'
$ git push origin branches/permanent
To hg:::/tmp/hello
 * [new branch]      branches/permanent -> branches/permanent
```

En hier is hoe het er aan de kant van Mercurial uit ziet:

```
$ hg branches
permanent          7:a4529d07aad4
develop            6:8f65e5e02793
default            5:bd5ac26f11f9 (inactive)
$ hg log -G
o changeset: 7:a4529d07aad4
| branch: permanent
| tag: tip
| parent: 5:bd5ac26f11f9
| user: Ben Straub <ben@straub.cc>
| date: Thu Aug 14 20:21:09 2014 -0700
| summary: A permanent change
|
| @ changeset: 6:8f65e5e02793
| / branch: develop
| | user: Ben Straub <ben@straub.cc>
| | date: Thu Aug 14 20:06:38 2014 -0700
| | summary: More documentation
|
o changeset: 5:bd5ac26f11f9
|\ bookmark: featureA
| | parent: 4:0434aaa6b91f
| | parent: 2:f098c7f45c4f
| | user: Ben Straub <ben@straub.cc>
| | date: Thu Aug 14 20:02:21 2014 -0700
| | summary: Merge remote-tracking branch 'origin/master'
[...]
```

De branchnaam “permanent” is opgeslagen met de changeset gemarkeerd met 7.

Aan de kant van Git, is het werken met beide stijlen branch gelijk: gewoon checkout, commit, fetch, merge, pull en push zoals je gewoonlijk zou doen. Een ding wat je moet weten is dat Mercurial het overschrijven van historie niet ondersteunt, alleen eraan toevoegen. Dit is hoe onze Mercurial repository eruit ziet na een interactieve rebase en een force-push:

```
$ hg log --style compact -G
o 10[tip] 99611176cbc9 2014-08-14 20:21 -0700 ben
| A permanent change
|
o 9 f23e12f939c3 2014-08-14 20:01 -0700 ben
| Add some documentation
|
o 8:1 c16971d33922 2014-08-14 20:00 -0700 ben
| goodbye
|
o 7:5 a4529d07aad4 2014-08-14 20:21 -0700 ben
| A permanent change
|
| @ 6 8f65e5e02793 2014-08-14 20:06 -0700 ben
| / More documentation
|
o 5[featureA]:4,2 bd5ac26f11f9 2014-08-14 20:02 -0700 ben
| \ Merge remote-tracking branch 'origin/master'
|
| o 4 0434aaa6b91f 2014-08-14 20:01 -0700 ben
| | update makefile
|
+--o 3:1 318914536c86 2014-08-14 20:00 -0700 ben
| | goodbye
|
| o 2 f098c7f45c4f 2014-08-14 20:01 -0700 ben
| / Add some documentation
|
o 1 82e55d328c8c 2005-08-26 01:21 -0700 mpm
| Create a makefile
|
o 0 0a04b987be5a 2005-08-26 01:20 -0700 mpm
| Create a standard "hello, world" program
```

Changesets 8, 9 en 10 zijn gemaakt voor en behoren bij de **permanent**-branch, maar de oude changesets zijn er nog steeds. Dit kan **erg** verwarringen veroorzaken voor je teamgenoten die Mercurial gebruiken, dus probeer dit te vermijden.

### Mercurial samenvatting

Git en Mercurial zijn gelijk genoeg dat het werken over deze grenzen redelijk goed gaat. Als je het wijzigen van historie die achterblijft op je machine vermeidt (zoals over het algemeen aangeraden wordt), zou je niet eens kunnen zeggen dat Mercurial aan de andere kant staat.

## Git and Bazaar

Onder de DVCSSen, is een andere beroemde: [Bazaar](#). Bazaar is gratis en open source, en het is onderdeel van het [GNU Project](#). Het gedraagt zich heel anders dan Git. Soms, om hetzelfde te bereiken als met Git, gebruik je een ander keyword, en sommige keywords die overeenkomen

hebben niet dezelfde betekenis. In het bijzonder, is het beheren van branches erg anders en kan verwarring scheppen, in het bijzonder wanneer iemand uit het Git universum komt. Niettemin is het mogelijk om op een Bazaar repository te werken vanuit een Git omgeving.

Er zijn veel projecten die toestaan om een Git te gebruiken als een Bazaar client. Hier zullen we het project van Felipe Contreras gebruiken die je kunt vinden op <https://github.com/felipec/git-remote-bzr>. Om het te installeren, hoef je alleen het bestand git-remote-bzr in een folder te downloaden die in je \$PATH staat:

```
$ wget https://raw.github.com/felipec/git-remote-bzr/master/git-remote-bzr -O ~/bin/git-remote-bzr  
$ chmod +x ~/bin/git-remote-bzr
```

Je zult ooko Bazaar geïnstalleerd moeten hebben. Dat is alles!

### Maak een Git repository van een Bazaar repository

Het is eenvoudig in gebruik. Het is voldoende om een Bazaar repository te klonen door het vooraf te laten gaan `bzr::`. Omdat Git en Bazaar beide een volledige kloon doen naar je machine, is het mogelijk om een Git kloon aan je lokale Bazaar kloon te knopen, maar dat wordt niet aangeraden. Het is veel eenvoudiger om je Git kloon direct aan dezelfde plaats te knopen als waar je Bazaar kloon aan verbonden is - de centrale repository.

Laten we er vanuit gaan dat je gewerkt heb met een remote repository die staat op het adres `bzr+ssh://developer@mybazaarserver:myproject`. Dan moet je het op de volgende manier klonen:

```
$ git clone bzr::bzr+ssh://developer@mybazaarserver:myproject myProject-Git  
$ cd myProject-Git
```

Nu is je Git repository gemaakt, maar het is nog niet geoptimaliseerd voor schijf gebruik. Dat is waarom je ook je Git repository moet schoonmaken en optimaliseren, zeker als het een grote is:

```
$ git gc --aggressive
```

### Bazaar branches

Bazaar staat alleen toe om branches te klonen, maar een repository kan verscheidene branches bevatten, en `git-remote-bzr` kan beide klonen. Bijvoorbeeld: om een branch te klonen:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs/trunk emacs-trunk
```

En om de gehele repository te klonen:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs emacs
```

The second command clones all the branches contained in the emacs repository; nevertheless, it is possible to point out some branches:

```
$ git config remote-bzr.branches 'trunk, xwindow'
```

Some remote repositories don't allow you to list their branches, in which case you have to manually specify them, and even though you could specify the configuration in the cloning command, you may find this easier:

```
$ git init emacs
$ git remote add origin bzr::bzr://bzr.savannah.gnu.org/emacs
$ git config remote-bzr.branches 'trunk, xwindow'
$ git fetch
```

## Ignore what is ignored with `.bzrignore`

Since you are working on a project managed with Bazaar, you shouldn't create a `.gitignore` file because you *may* accidentally set it under version control and the other people working with Bazaar would be disturbed. The solution is to create the `.git/info/exclude` file either as a symbolic link or as a regular file. We'll see later on how to solve this question.

Bazaar uses the same model as Git to ignore files, but also has two features which don't have an equivalent into Git. The complete description may be found in [the documentation](#). The two features are:

1. "!!" allows you to ignore certain file patterns even if they're specified using a "!" rule.
2. "RE:" at the beginning of a line allows you to specify a [Python regular expression](#) (Git only allows shell globs).

As a consequence, there are two different situations to consider:

1. If the `.bzrignore` file does not contain any of these two specific prefixes, then you can simply make a symbolic link to it in the repository: `ln -s .bzrignore .git/info/exclude`
2. Otherwise, you must create the `.git/info/exclude` file and adapt it to ignore exactly the same files in `.bzrignore`.

Whatever the case is, you will have to remain vigilant against any change of `.bzrignore` to make sure that the `.git/info/exclude` file always reflects `.bzrignore`. Indeed, if the `.bzrignore` file were to change and contained one or more lines starting with "!!" or "RE:", Git not being able to interpret these lines, you'll have to adapt your `.git/info/exclude` file to ignore the same files as the ones ignored with `.bzrignore`. Moreover, if the `.git/info/exclude` file was a symbolic link, you'll have to first delete the symbolic link, copy `.bzrignore` to `.git/info/exclude` and then adapt the latter. However, be careful with its creation because with Git it is impossible to re-include a file if a parent directory of that file is excluded.

## Fetch the changes of the remote repository

To fetch the changes of the remote, you pull changes as usually, using Git commands. Supposing that your changes are on the `master`-branch, you merge or rebase your work on the `origin/master`-branch:

```
$ git pull --rebase origin
```

## Push your work on the remote repository

Because Bazaar also has the concept of merge commits, there will be no problem if you push a merge commit. So you can work on a branch, merge the changes into `master` and push your work. Then, you create your branches, you test and commit your work as usual. You finally push your work to the Bazaar repository:

```
$ git push origin master
```

## Caveats

Git's remote-helpers framework has some limitations that apply. In particular, these commands don't work:

- `git push origin :branch-to-delete` (Bazaar can't accept ref deletions in this way.)
- `git push origin old:new` (it will push *old*)
- `git push --dry-run origin branch` (it will push)

## Summary

Since Git's and Bazaar's models are similar, there isn't a lot of resistance when working across the boundary. As long as you watch out for the limitations, and are always aware that the remote repository isn't natively Git, you'll be fine.

## Git en Perforce

Perforce is een erg populaire versie-beheer systeem in bedrijfsomgevingen. Het bestaat al sinds 1995, wat het het oudste systeem maakt dat we in dit hoofdstuk behandelen. Zoals het is, is het ontworpen met de beperkingen van die tijd; het gaat er vanuit dat je altijd verbonden bent met een enkele centrale server, en er wordt maar één versie bewaard op je lokale schijf. Het valt niet te ontkennen dat de mogelijkheden en beperkingen goed afgestemd zijn op een aantal specifieke werksituaties, maar er zijn veel projecten die Perforce gebruiken waar Git eigenlijk veel beter zou werken.

Er zijn twee opties als je Perforce en Git samen wilt gebruiken. De eerste die we gaan behandelen is de "Git Fusion" bridge van de makers van Perforce, die je de subtrees van je Perforce depot ter beschikking stelt als lees-schrijf Git repositories. De tweede is git-p4, een bridge op het werkstation die je Git als een Perforce client laat werken, zonder een herconfiguratie van de Perforce server af te dwingen.

## Git Fusion

Perforce stelt een product ter beschikking met de naam Git Fusion (beschikbaar op <http://www.perforce.com/git-fusion>), welke een Perforce server synchroniseert met Git repositories aan de kant van de server.

### Inrichten

Voor onze voorbeelden, zullen we de eenvoudigste installatie methode voor Git Fusion gebruiken, en dat is het downloaden van een virtual machine die de Perforce daemon en Git Fusion draait. Je kunt deze virtual machine image krijgen op <http://www.perforce.com/downloads/Perforce/20-User>, en als het eenmaal gedownload is, importeer je het in je favoriete virtualisatie software (wij zullen VirtualBox gebruiken).

Als de machine voor het eerst opstart, vraag het je om het wachtwoord van de drie Linux gebruikers (`root`, `perforce` en `git`) te wijzigen, en een instantie naam op te geven die kan worden gebruikt om deze installatie van andere te onderscheiden op hetzelfde netwerk. Als dat alles gereed is, zal je het volgende zien:

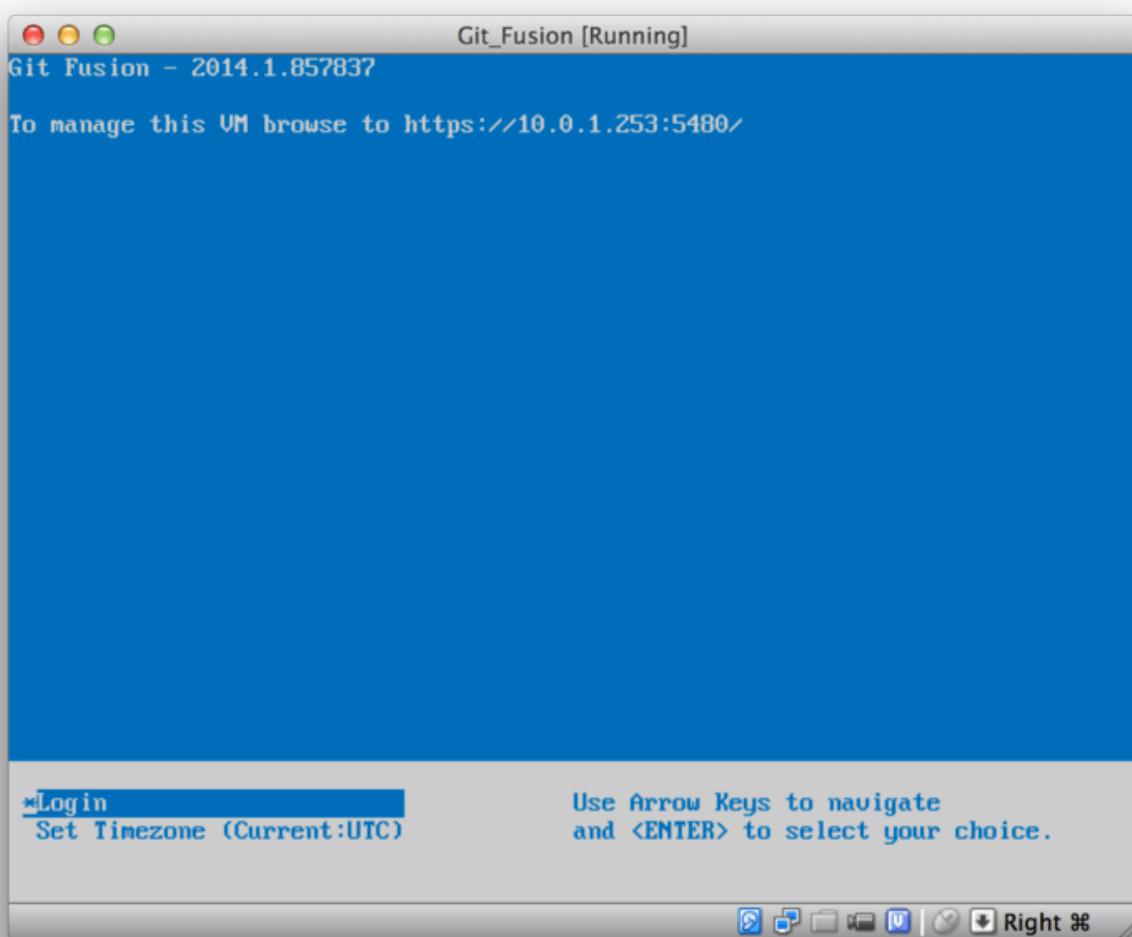


Figure 146. Het Git Fusion virtual machine opstart scherm.

Let even specifiek op het IP adres dat hier wordt getoond, we zullen deze later gaan gebruiken.

Vervolgens maken we een Perforce gebruiker aan. Kies de “Login” optie onder aan het scherm en druk op enter (of SSH naar de machine), en log in als **root**. Gebruik deze commando’s om een gebruiker aan te maken:

```
$ p4 -p localhost:1666 -u super user -f john  
$ p4 -p localhost:1666 -u john passwd  
$ exit
```

Het eerste opent een VI editor om de gebruiker aan te passen, maar je kunt de standaard-instellingen accepteren door :wq te typen en enter te drukken. Het tweede zal je twee keer vragen om een wachtwoord in te typen. Dat is alles wat we hebben te doen met een shell prompt, dus beëindigen we deze sessie.

Wat je daarna moet doen om ons te volgen is Git te vertellen om geen SSL certificaten te verifiëren. Het Git Fusion image wordt met een certificaat geleverd, maar dat is voor een domain die niet zal overeenkomen met het IP adres van je virtuele machine, dus Git zal de HTTPS connectie weigeren. Als het de bedoeling is dat dit een permanente installatie gaat worden, raadpleeg dan het handboek van Perforce Git Fusion om een ander certificaat te installeren; voor het doel van ons voorbeeld zal dit voldoende zijn:

```
$ export GIT_SSL_NO_VERIFY=true
```

Nu kunnen we gaan testen of alles werkt.

```
$ git clone https://10.0.1.254/Talkhouse  
Cloning into 'Talkhouse'...  
Username for 'https://10.0.1.254': john  
Password for 'https://john@10.0.1.254':  
remote: Counting objects: 630, done.  
remote: Compressing objects: 100% (581/581), done.  
remote: Total 630 (delta 172), reused 0 (delta 0)  
Receiving objects: 100% (630/630), 1.22 MiB | 0 bytes/s, done.  
Resolving deltas: 100% (172/172), done.  
Checking connectivity... done.
```

Het virtuele machine image komt met een voorinstalleerd voorbeeld project dat je kunt klonen. Hier klonen we over HTTPS, met de **john** gebruiker die we hierboven aangemaakt hebben; Git vraagt om de inloggegevens voor deze connectie, maar de credential cache staat ons toe om deze stap voor de hierop volgende aanvragen over te slaan.

## Fusion Configuratie

Als je eenmaal Git Fusion geïnstalleerd hebt, zal je de configuratie hier en daar willen aanpassen. Dit is eigenlijk behoorlijk eenvoudig te doen met gebruik van je favoriete Perforce client; map eenvoudigweg de **//.git-fusion** directory op de Perforce server naar je werkruimte. De bestandsstructuur zie er als volgt uit:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
|
└── p4gf_config
    ├── repos
    │   └── Talkhouse
    │       └── p4gf_config
    └── users
        └── p4gf_usermap

498 directories, 287 files
```

De **objects** directory wordt door Git Fusion intern gebruikt om Perforce objecten op Git te mappen en andersom, je zou niet hoeven te rommelen met de inhoud daarvan. Er is een globaal **p4gf\_config** bestand in deze directory, zowel als een voor elke repository – dit zijn de configuratie bestanden die bepalen hoe Git Fusion zich gedraagt. Laten we het bestand in de root eens bekijken:

```
[repo-creation]
charset = utf8

[git-to-perforce]
change-owner = author
enable-git-branch-creation = yes
enable-swarm-reviews = yes
enable-git-merge-commits = yes
enable-git-submodules = yes
preflight-commit = none
ignore-author-permissions = no
read-permission-check = none
git-merge-avoidance-after-change-num = 12107

[perforce-to-git]
http-url = none
ssh-url = none

[@features]
imports = False
chunked-push = False
matrix2 = False
parallel-push = False

[authentication]
email-case-sensitivity = no
```

We zullen niet ingaan op de betekenissen van al deze vlaggen, maar merk op dat dit niet meer is dan een INI-geformatteerd tekstbestand, vergelijkbaar met wat Git gebruikt voor configuratie. Dit bestand bepaalt de globale opties, die kunnen worden overschreven door repository-specifieke configuratie bestanden, zoals `repos/Talkhouse/p4gf_config`. Als je dat bestand opent, zal je een `[@repo]` sectie zien met wat instellingen die anders zijn dan de globale standaard instellingen. Je zult ook secties zien die er zo uit zien:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

Dit is een mapping tussen een Perforce branch en een Git branch. De sectie kan elke naam zijn die je maar kunt verzinnen, zo lang als het maar uniek is. `git-branch-name` stelt je in staat een depot pad die maar moeizaam zou zijn in Git te converteren naar een handigere naam. De `view` instelling bepaalt hoe Perforce bestanden zijn gemapt op de Git repository, waarbij de standaard view mapping syntax wordt gebruikt. Er kunnen meer dan één mapping worden opgegeven, zoals in dit voorbeeld:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
      //depot/project2/mainline/... project2/...
```

Op deze manier kan je, als je reguliere werkruimte mapping wijzigingen in de structuur van de directories in zich heeft, dat met een Git repository repliceren.

Het laatste bestand dat we zullen behandelen is `users/p4gv_usermap`, wat Perforce gebruikers op Git gebruikers mapt, en je zult deze waarschijnlijk niet eens nodig hebben. Bij het converteren van een Perforce changeset naar een Git commit, is het standaard gedrag van Git Fusion om de Perforce gebruiker op te zoeken, en het email adres en volledige naam die daar is opgeslagen voor het auteur/committer veld van Git te gebruiken. Bij het converteren de andere kant op, is de standaard om de Perforce gebruiker op te zoeken met het email adres dat is opgeslagen in het auteur veld in de Git commit, en om de changeset op te sturen als die gebruiker (waarbij de geldende permissies worden gerespecteerd). In de meeste gevallen, zal dit gedrag prima werken, maar bekijk nu eens het volgende mapping bestand:

```
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"
```

Elke regel is van het formaat `<user> <email> "<volledige naam>"`, en vormt de mapping van een enkele gebruiker. De eerste twee regels mappen twee verschillende email adressen naar de naam van dezelfde Perforce gebruiker. Dit is handig als je onder verschillende email adressen Git commits hebt gemaakt (of van email adres bent veranderd), en je deze naar dezelfde Perforce gebruiker wilt mappen. Bij het maken van een Git commit van een Perforce changeset, wordt de

eerste regel die met de Perforce gebruiker overeenkomt in Git gebruikt voor informatie over het auteurschap.

De laatste twee regels voorkomen dat de echte namen en email adressen van Bob en Joe in de commits terechtkomen die voor Git worden gemaakt. Dit is handig als je een intern project openbaar wilt maken, maar je niet de hele personeelsbestand aan de wereld wilt blootstellen. Merk op dat de email adressen en volledige namen uniek moeten zijn, tenzij je wilt dat alle Git commits worden toegeschreven aan een enkele virtuele auteur.

## Workflow

Perforce Git Fusion is een tweewegs bridge tussen Perforce en Git versie beheer. Laten we een kijken hoe het voelt om er vanaf de Git kant mee te werken. We zullen aannemen dat de het “Jam” project gemapt hebben met een configuratie bestand zoals hierboven, en die we kunnen klonen als volgt:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://ben@10.0.1.254':
remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]
```

De eerste keer dat je dit doet kan het eveneens duren. Wat er gebeurt is dat Git Fusion alle van toepassing zijnde changesets in de Perforce historie naar Git commits converteert. Dit gebeurt lokaal op de server, dus het is relatief snel, maar als je veel historie hebt, kan het nog steeds lang duren. Toekomstige fetches voeren incrementele conversies uit, dus zal het meer als de normale snelheid van Git aanvoelen.

Zoals je kunt zien, lijkt onze repository precies op elke andere Git repository waar je mee zou kunnen werken. Er zijn drie branches, en Git heeft heel behulpzaam een lokale `master`-branch gemaakt die `origin/master` trackt. Laten we eens wat werk doen, en een paar nieuwe commits

maken:

```
# ...
$ git log --oneline --decorate --graph --all
* cfd46ab (HEAD, master) Add documentation for new feature
* a730d77 Whitespace
* d254865 (origin/master, origin/HEAD) Upgrade to latest metrowerks on Beos -- the
Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

We hebben twee nieuwe commits. Laten we nu eens controleren of iemand anders aan het werk is geweest:

```
$ git fetch
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From https://10.0.1.254/Jam
  d254865..6afeb15  master      -> origin/master
$ git log --oneline --decorate --graph --all
* 6afeb15 (origin/master, origin/HEAD) Update copyright
| * cfd46ab (HEAD, master) Add documentation for new feature
| * a730d77 Whitespace
|/
* d254865 Upgrade to latest metrowerks on Beos -- the Intel one.
* bd2f54a Put in fix for jam's NT handle leak.
[...]
```

Het ziet er naar uit dat dat het geval was! Je zou het met deze uitvoer niet zeggen, maar de **6afeb15** commit was in gewoon gemaakt met behulp van een Perforce client. Het ziet er net zo uit als elke andere commit wat Git betreft, en dat is nu net de bedoeling. Laten we kijken hoe de Perforce gebruiker met een merge commit omgaat:

```

$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Perforce: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
 6afeb15..89cba2b  master -> master

```

Git denkt dat het gelukt is. Laten we eens kijken naar de historie van het **README** bestand vanuit het oogpunt van Perforce, met het revisiegraaf gereedschap **p4v**.

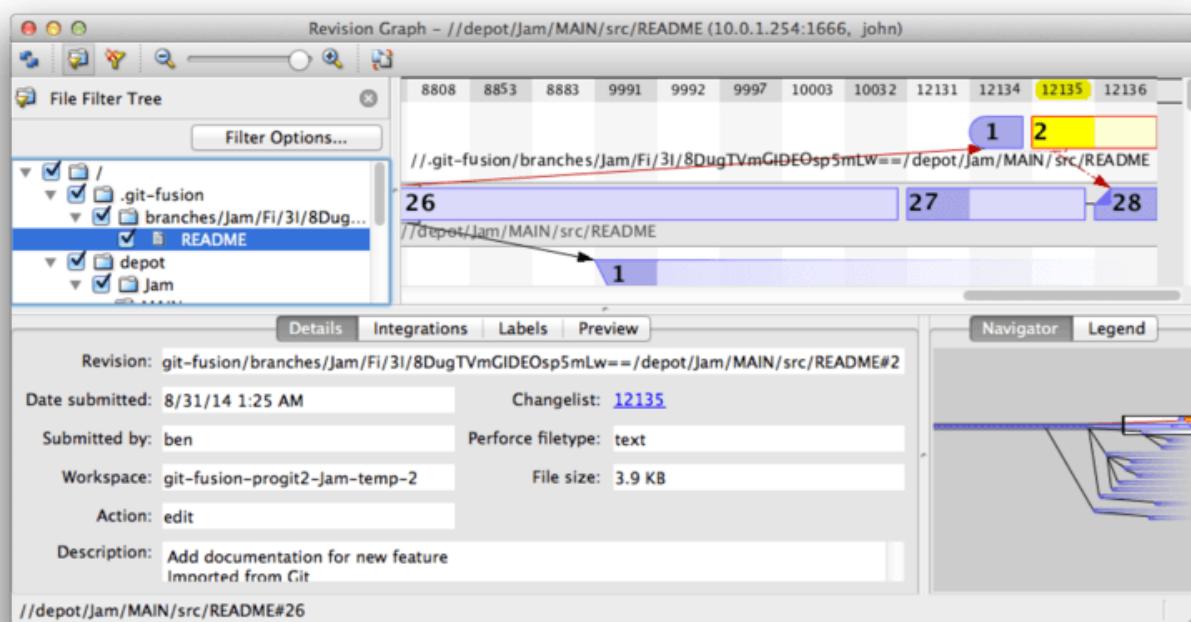


Figure 147. Perforce revisie graaf als resultaat van Git push.

Als je dit scherm nog nooit gezien hebt, kan het er nogal verwarrend uitzien, maar het laat dezelfde concepten zien als een grafisch programma voor Git historie. We kijken naar de geschiedenis van het **README** bestand, dus de directory tree links boven laat alleen dat bestand zien zoals deze voorkomt in de diverse branches. Rechtsboven hebben we een grafische kijk op hoe verschillende

revisies van het bestand aan elkaar zijn gerelateerd, en het hoog-overzicht van dit plaatje is rechts onder. De rest van dit scherm wordt gegeven aan de details-scherm voor de geselecteerde revisie (2 in dit geval).

Een ding om op te merken is dat de graaf er precies hetzelfde uitziet als die in de historie van Git. Perforce had geen benoemde branch om de 1 en 2 commits in op te slaan, dus heeft het een “anonymous” branch aangemaakt in de `.git-fusion` directory om deze in op te slaan. Dit zal ook gebeuren voor Git branches met een naam die niet overeenkomen met een branchnaam in Perforce (en je kunt deze later op een Perforce branch mappen met gebruik van het configuratie bestand).

Het leeuwendeel van dit alles gebeurt achter de schermen, maar het eindresultaat is dat de ene persoon in een team Git kan gebruiken, een ander kan Perforce gebruiken, en geen van beiden heeft weet van de keuze van de ander.

### Git-Fusion Samenvatting

Als je toegang hebt (of kan krijgen) naar je Perforce server, is Git Fusion een hele goede manier om Git en Perforce met elkaar te laten samenwerken. Het vergt een beetje configuratie, maar de leercurve is niet heel erg steil. Dit is een van de weinige paragrafen in dit hoofdstuk waar waarschuwingen over de volledige kracht van Git niet zullen voorkomen. Dat wil niet zeggen dat Perforce erg blij gaat zijn met alles wat je er naartoe stuurt – als je probeert de geschiedenis herschrijven die al gepusht is, zal Git Fusion dit weigeren – maar Git Fusion doet erg z'n best om natuurlijk aan te voelen. Je kunt zelfs Git submodulen gebruiken (al zullen ze er voor Perforce gebruikers vreemd uitzien), en branches mergen (dit wordt aan de Perforce kant als een integratie opgeslagen).

Als je de beheerder niet kunt overtuigen om Git Fusion op te zetten, is er nog steeds een manier om deze instrumenten samen te laten werken.

### Git-p4

Git-p4 is een tweewegs bridge tussen Git en Perforce. Het draait volledig binnen je Git repository, dus je hebt geen enkele vorm van toegang tot de Perforce server nodig (buiten de login gegevens natuurlijk). Git-p4 is niet zo flexibel of compleet als oplossing als Git Fusion, maar het stelt je wel in staat om het meeste wat je zou willen doen uit te voeren zonder afbreuk te doen aan de omgeving van de server.

 Je zult de `p4` tool ergens in je `PATH` moeten zetten om te kunnen werken met git-p4. Op het moment van schrijven is het voor iedereen te verkrijgen op <http://www.perforce.com/downloads/Perforce/20-User>.

### Inrichting

Voor het voorbeeld, zullen we de Perforce server van het Git Fusion OVA als boven gebruiken, maar we slaan de Git Fusion server over en gaan direct naar het versie beheer van Perforce.

Om de `p4` commando-regel client te gebruiken (waar git-p4 van afhankelijk is), zal je een aantal omgevingsvariabelen moeten inrichten:

```
$ export P4PORT=10.0.1.254:1666  
$ export P4USER=john
```

## Op gang komen

Zoals altijd in Git, is het eerste commando het klonen:

```
$ git p4 clone //depot/www/live www-shallow  
Importing from //depot/www/live into www-shallow  
Initialized empty Git repository in /private/tmp/www-shallow/.git/  
Doing initial import of //depot/www/live/ from revision #head into  
refs/remotes/p4/master
```

Dit maakt wat in Git terminologie een “shallow” is; alleen de allerlaatste Perforce revisie wordt in Git geïmporteerd; onthoud dat Perforce niet ontworpen is om elke revisie aan elke gebruiker te geven. Dit is genoeg om Git te laten werken als een Perforce client, maar voor andere toepassingen is dit niet genoeg.

Als dit eenmaal klaar is, hebben we een volledig werkende Git repository:

```
$ cd myproject  
$ git log --oneline --all --graph --decorate  
* 70eaf78 (HEAD, p4/master, p4/HEAD, master) Initial import of //depot/www/live/ from  
the state at revision #head
```

Merk op dat er een “p4” remote is voor de Perforce server, maar al het overige ziet eruit als een standaard clone. Dit is echter een beetje misleidend; er is in het echt geen remote aanwezig.

```
$ git remote -v
```

Er bestaan in deze repository helemaal geen remotes. Git-p4 heeft een aantal refs gemaakt om de staat van de server te vertegenwoordigen, en ze zien er uit als remote refs voor `git log`, maar ze worden niet door Git onderhouden, en je kunt er niet naar pushen.

## Workflow

Okay, laten we wat werk doen. Laten we aannemen dat je wat vorderingen gemaakt hebt op een zeer belangrijke feature, en je bent klaar om het te laten zien aan de rest van je team.

```
$ git log --oneline --all --graph --decorate  
* 018467c (HEAD, master) Change page title  
* c0fb617 Update link  
* 70eaf78 (p4/master, p4/HEAD) Initial import of //depot/www/live/ from the state at  
revision #head
```

We hebben twee nieuwe commits gemaakt die klaar zijn om te worden gestuurd naar de Perforce server. Laten we kijken of er iemand anders aan het werk is geweest vandaag:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|/
* 70eaf78 Initial import of //depot/www/live/ from the state at #head
```

Het ziet er naar uit van wel, en `master` en `p4/master` zijn uiteen gelopen. Het branching systeem van Perforce lijkt *in niets* op die van Git, dus het aanleveren van merge commits zal nergens op slaan. Git-p4 raadt aan dat je je commits rebaset, en levert zelfs een manier om dit snel te doen:

```
$ git p4 rebase
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
No changes to import!
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
Applying: Update link
Applying: Change page title
index.html | 2 ++
1 file changed, 1 insertion(+), 1 deletion(-)
```

Je kunt het uit de uitvoer waarschijnlijk wel afleiden, maar `git p4 rebase` is kort voor `git p4 sync` gevuld door een `git rebase p4/master`. Het is iets slimmer dan dat, vooral in het geval dat je met meerdere branches werkt, maar dit is een goede benadering.

Nu is onze historie weer lineair, en we zijn klaar om onze wijzigingen te delen met Perforce. Het `git p4 submit` commando zal proberen om een nieuwe Perforce revisie te maken voor elke Git commit tussen `p4/master` en `master`. Als we dit aanroepen komen we in onze favoriete editor, en de inhoud van het bestand ziet er ongeveer zo uit:

```

# A Perforce Change Specification.

#
# Change:      The change number. 'new' on a new changelist.
# Date:        The date this specification was last modified.
# Client:      The client on which the changelist was created. Read-only.
# User:        The user who created the changelist.
# Status:      Either 'pending' or 'submitted'. Read-only.
# Type:        Either 'public' or 'restricted'. Default is 'public'.
# Description: Comments about the changelist. Required.
# Jobs:        What opened jobs are to be closed by this changelist.
#               You may delete jobs from this list. (New changelists only.)
# Files:       What opened files from the default changelist are to be added
#               to this changelist. You may delete files from this list.
#               (New changelists only.)

```

Change: new

Client: john\_bens-mbp\_8487

User: john

Status: new

Description:

Update link

Files:

//depot/www/live/index.html # edit

```

##### git author ben@straub.cc does not match your p4 account.
##### Use option --preserve-user to modify authorship.
##### Variable git-p4.skipUserNameCheck hides this message.
##### everything below this line is just the diff #####
--- //depot/www/live/index.html 2014-08-31 18:26:05.000000000 0000
+++ /Users/ben/john_bens-mbp_8487/john_bens-mbp_8487/depot/www/live/index.html 2014-
08-31 18:26:05.000000000 0000
@@ -60,7 +60,7 @@
</td>
<td valign=top>
Source and documentation for
-<a href="http://www.perforce.com/jam/jam.html">
+<a href="jam.html">
Jam/MR</a>,
a software build tool.
</td>
```

Dit is voor een groot gedeelte dezelfde inhoud die je zou zien bij het aanroepen van `p4 submit`, behalve het spul aan het eind dat git-p4 behulpzaam heeft toegevoegd. Git-p4 probeert jouw Git

instellingen en die van Perforce elk te volgen als het een naam moet geven voor een commit of een changeset, maar in sommige gevallen zal je het willen overschrijven. Bijvoorbeeld, als de Git commit die je aan het importeren was geschreven is door iemand die geen Perforce gebruiker account heeft, zal je nog steeds de resulterende changeset eruit willen laten zien alsof zij het geschreven hebben (en niet jij).

Git-p4 heeft het bericht van de Git commit heel behulpzaam geïmporteerd als de inhoud voor deze Perforce changeset, dus alles wat we hoeven te doen is bewaren en stoppen, twee keer (een keer voor elke commit). De uiteindelijke shell uitvoer zal er ongeveer zo uit zien:

```
$ git p4 submit
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Synchronizing p4 checkout...
... - file(s) up-to-date.
Applying dbac45b Update link
//depot/www/live/index.html#4 - opened for edit
Change 12143 created with 1 open file(s).
Submitting change 12143.
Locking 1 files ...
edit //depot/www/live/index.html#5
Change 12143 submitted.
Applying 905ec6a Change page title
//depot/www/live/index.html#5 - opened for edit
Change 12144 created with 1 open file(s).
Submitting change 12144.
Locking 1 files ...
edit //depot/www/live/index.html#6
Change 12144 submitted.
All commits applied!
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12144 (100%)
Rebasing the current branch onto remotes/p4/master
First, rewinding head to replay your work on top of it...
$ git log --oneline --all --graph --decorate
* 775a46f (HEAD, p4/master, p4/HEAD, master) Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Het resultaat is alsof we zojuist een `git push` gedaan hebben, wat beste analogie is van hetgeen er in werkelijkheid is gebeurd.

Merk op dat bij dit proces elke Git commit wordt omgezet in een Perforce changeset; als je deze naar een enkele changeset wilt terugbrengen, kan je dat doen met een interactieve rebase voordat je `git p4 submit` aanroeft. Merk ook op dat de SHA-1 hashes van alle commits die als changesets zijn opgestuurd gewijzigd zijn; dit is omdat git-p4 een regel toevoegt aan het eind van elke commit

die het converteert:

```
$ git log -1
commit 775a46f630d8b46535fc9983cf3ebe6b9aa53145
Author: John Doe <john@example.com>
Date:   Sun Aug 31 10:31:44 2014 -0800

    Change page title

[git-p4: depot-paths = "//depot/www/live/": change = 12144]
```

Wat gebeurt er als je probeert een merge commit op te sturen? Laten we het eens proberen. Hier is de situatie waar we ons in gewerkt hebben:

```
$ git log --oneline --all --graph --decorate
* 3be6fd8 (HEAD, master) Correct email address
*   1dcbf21 Merge remote-tracking branch 'p4/master'
|\ \
| * c4689fc (p4/master, p4/HEAD) Grammar fix
* | cbacd0a Table borders: yes please
* | b4959b6 Trademark
|/
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

De Git en Perforce historie lopen uiteen na 775a46f. Aan de Git kant zijn er twee commits, daarna een merge commit met de Perforce head, en daarna een andere commit. We zullen proberen deze in te sturen bovenop een enkele changeset aan de kant van Perforce. Laten we eens kijken wat er gebeurt als we proberen deze nu in te sturen:

```
$ git p4 submit -n
Perforce checkout for depot path //depot/www/live/ located at /Users/ben/john_bens-
mbp_8487/john_bens-mbp_8487/depot/www/live/
Would synchronize p4 checkout in /Users/ben/john_bens-mbp_8487/john_bens-
mbp_8487/depot/www/live/
Would apply
    b4959b6 Trademark
    cbacd0a Table borders: yes please
    3be6fd8 Correct email address
```

De **-n** vlag staat voor **--dry-run**, wat probeert te rapporteren wat er zou gaan gebeuren als het submit commando in het echt zou worden aangeroepen. In dit geval ziet het er naar uit dat we drie Perforce changesets zouden gaan aanmaken, wat overeenkomt met de drie non-merge commits die nog niet bestaan op de Perforce server. Dat klinkt precies als wat we willen, laten we kijken hoe het uitpakt:

```
$ git p4 submit
[...]
$ git log --oneline --all --graph --decorate
* dadbd89 (HEAD, p4/master, p4/HEAD, master) Correct email address
* 1b79a80 Table borders: yes please
* 0097235 Trademark
* c4689fc Grammar fix
* 775a46f Change page title
* 05f1ade Update link
* 75cd059 Update copyright
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Onze historie werd lineair, precies alsof we hadden gerebased voor het insturen (en dat is precies wat er is gebeurd). Dit houdt in dat je zonder problemen branches kunt maken, erop werken, weggooien en mergen aan de kant van Git zonder bang te zijn dat je historie op de een of andere manier niet meer compatible is met Perforce. Als je het kunt rebasen, kan je het insturen naar een Perforce server.

## Branchen

Als je Perforce project meerdere branches heeft, is niet alles voor je verloren; git-p4 kan het op een manier behandelen die je doet denken aan Git. Laten we zeggen dat je Perforce depot zo is ingericht:

```
//depot
|__ project
    |__ main
    |__ dev
```

En stel nu dat je een **dev**-branch hebt, die een view spec heeft die er zo uit ziet:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 kan deze situatie automatisch herkennen en de juiste handeling uitvoeren:

```

$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
    Importing new branch project/dev

    Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfac Populate //depot/project/main/... //depot/project/dev/....
|
* 2b83451 Project init

```

Merk de “@all” specificatie in het depot pad op; dat vertelt git-p4 om niet alleen de laatste changeset voor die subtree te klonen, maar alle changesets die ooit in aanraking zijn geweest met deze paden. Dit zit dichter bij het Git-concept van een klone, maar als je aan een project werkt met een lange historie, kan dit wel even duren.

De **--detect-branches** vlag instrueert git-p4 om de Perforce branch specificaties te gebruiken om de branches op Git refs te mappen. Als deze mappings niet aanwezig zijn op de Perforce server (wat een heel valide manier is om Perforce te gebruiken), kan je git-p4 aangeven wat de branch mappings zijn, en je krijgt hetzelfde resultaat:

```

$ git init project
Initialized empty Git repository in /tmp/project/.git/
$ cd project
$ git config git-p4.branchList main:dev
$ git clone --detect-branches //depot/project@all .

```

Door de **git-p4.branchList** configuratie variabele op **main:dev** te zetten wordt git-p4 geïnstrueerd dat “main” en “dev” beide branches zijn, en dat de tweede een kind is van de eerste.

Als we nu **git checkout -b dev p4/project/dev** doen en een aantal commits maken, is git-p4 slim genoeg om de juiste branch aan te spreken als we **git p4 submit** uitvoeren. Jammergenoeg kan git-p4 geen *shallow* clones en meerdere branches tegelijk aan; als je een enorm groot project hebt en je wilt aan meer dan één branch werken, zal je **git p4 clone** voor elke branch waarnaar je wilt submitten moeten uitvoeren.

Voor het maken of integreren van branches, zal je een Perforce client moeten gebruiken. Git-p4 kan alleen met bestaande branches synchroniseren of daarnaar submitten, en het kan dit alleen doen voor één lineaire changeset per keer. Als je twee branches in Git merget en probeert de nieuwe changeset in te sturen, is alles wat er wordt opgeslagen een stel bestandswijzigingen; de metadata over welke branches er zijn betrokken bij de integratie gaat verloren.

## Git en Perforce samenvatting

Git-p4 maakt het mogelijk om een Git workflow te gebruiken met een Perforce server, en het is er best wel goed in. Echter, het is belangrijk om te onthouden dat Perforce de baas is over de broncode, en dat je Git alleen maar gebruikt om er lokaal mee te werken. Wees vooral erg voorzichtig om Git commits te delen; als je een remote hebt die andere mensen ook gebruiken, push dan geen enkele commit die niet al eerder naar die Perforce server zijn gestuurd.

Als je vrijelijk zowel de Perforce en Git clients tegelijk wilt gebruiken voor broncode beheer, en je kunt de beheerder van de server ervan overtuigen om het te installeren, zal Git Fusion Git een eersterangs versiebeheer client maken voor een Perforce server.

## Git en TFS

Git wordt steeds populairder onder Windows ontwikkelaars, en als je code schrijft op Windows, is er een grote kans dat je de Team Foundation Server (TFS) van Microsoft gebruikt. TFS is een samenwerkings pakket die een defect- en werkbonvolgsysteem bevat, procesondersteuning voor Scrum en dergelijke, code review en versiebeheer. Er gaat wat verwarring aankomen: **TFS** is de server, die broncode beheer ondersteunt met behulp van zowel Git als hun eigen VCS, die ze **TFVC** (Team Foundation Version Control) hebben gedoopt. Git ondersteuning is een nogal nieuwe mogelijkheid voor TFS (geleverd met de 2013 versie), dus alle instrumenten die van voor die datum stammen verwijzen naar het versie-beheer gedeelte als “TFS”, zelfs als ze voor het grootste gedeelte werken met TFVC.

Als je jezelf in een team zit dat TFVC gebruikt, en je zou liever Git gebruiken als je versie-beheer client, is dit een mooi project voor jou.

### Welk instrument

Er zijn er in feite twee: git-tf en git-tfs.

Git-tfs (te vinden op <https://github.com/git-tfs/git-tfs>) is een .NET project, en (op het tijdstip van schrijven) kan alleen onder Windows draaien. Om met Git repositories te werken, gebruikt het de .NET bindings voor libgit2, een library-oriented implementatie van Git die zeer hoog performant is en die een hoge mate van flexibiliteit biedt met de pretentie van een Git repository. Libgit2 is geen volledige implementatie van Git, dus om dit verschil te compenseren zal git-tfs gewoon de commando-regel versie van de Git client aanroepen om bepaalde operaties uit te voeren, dus er zijn geen kunstmatige beperkingen in wat het met Git repositories kan doen. De ondersteuning van de TFVC functies is zeer volwassen, omdat het de mogelijkheden van Visual Studio gebruikt voor operaties met servers. Dit houdt in dat je toegang nodig hebt tot deze mogelijkheden, wat betekent dat je een recente versie van Visual Studio moet installeren (elke editie sinds versie 2010, inclusief Express sinds versie 2012), of de Visual Studio SDK.

Git-tf (welke huist op <https://gittf.codeplex.com>) is een Java project, en als zodanig loopt het op elke computer met een Java runtime omgeving. Het interacteert met Git repositories middels JGit (een JVM implementatie van Git), wat inhoudt dat het vrijwel geen beperkingen kent in termen van Git functies. Echter, de ondersteuning voor TFVC is beperkt ten opzichte van git-tfs - het ondersteunt bijvoorbeeld geen branches.

Dus elke tool heeft z'n voors en tegens, en er zijn genoeg situaties waarbij de een te prefereren is

boven de ander. We zullen het normale gebruik van beide gaan behandelen in dit boek.



Je zult toegang nodig hebben tot een TFVC-gebaseerde repository om deze instructies mee te doen. Ze zijn niet zo ruim vorhanden als Git of Subversion repositories, dus je zult misschien je eigen moeten gaan maken. Codeplex (<https://www.codeplex.com>) of Visual Studio Online (<http://www.visualstudio.com>) zijn goede opties hiervoor.

### Beginnen met: git-tf

Het eerste wat je doet, net als met elk andere Git project is klonen. Dit is hoe het er uit ziet met `git-tf`:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

Het eerste argument is de URL van een TFVC collectie, het tweede is er een in de vorm `$/project/branch`, en het derde is het pad naar de lokale Git repository die gemaakt moet worden (deze laatste is optioneel). Git-tf kan maar met een branch tegelijk werken; als je checkins wilt maken op een andere TFVC branch, zul je een nieuwe kloon moeten maken van die branch.

Dit maakt een volledig functionele Git repository:

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

Dit wordt een *shallow clone* (oppervlakkige kloon) genoemd, wat inhoudt dat alleen de laatste changeset is gedownload. TFVC is niet ontworpen met het idee dat elk werkstation een volledige kopie van de historie heeft, dus git-tf valt terug op het ophalen van de laatste versie, wat veel sneller is.

Als je de tijd hebt, is het waarschijnlijk de moeite om de gehele project historie te klonen, met de `--deep` optie:

```
$ git tf clone https://tfscodeplex.com:443/tfs/TFS13 $/myproject/Main \
  project_git --deep
Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Merk de tags op met namen als [TFS\\_C35189](#); dit is een functionaliteit die je helpt met het relateren van Git commits aan TFVC changesets. Dit is een aardige manier om het weer te geven, omdat je met een simpele log commando kunt zien welke van jouw commits zijn gerelateerd aan een snapshot die leeft in TFVC. Ze zijn niet noodzakelijk (en je kunt ze gewoon uitschakelen met [git config git-tf.tag false](#)) - git-tf houdt de echte commit-changeset relaties bij in het [.git/git-tf](#) bestand.

### Beginnen met: [git-tfs](#)

Het klonen met git-tfs gedraagt zich iets anders. Neem waar:

```
PS> git tfs clone --with-branches \
  https://username.visualstudio.com/DefaultCollection \
  $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeffb961958b674
```

Merk de [--with-branches](#) vlag op. Git-tfs is in staat de TFVC branches aan Git branches te relateren, en deze vlag geeft aan dat er een lokale Git branch moet worden aangemaakt voor elke TFVC branch. Dit wordt sterk aangeraden als je ooit gebrancht of gemerged hebt in TFS, maar het gaat niet werken met een server die ouder is dan TFS 2010 - voor die versie waren "branches" gewoon folders, dus git-tfs kan ze niet onderscheiden van reguliere folders.

Laten we een kijkje nemen naar de Git repository die het resultaat is:

```

PS> git log --oneline --graph --decorate --all
* 44cd729 (tfv/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfv/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date:   Fri Aug 1 03:41:59 2014 +0000

Hello

git-tfs-id:
[https://username.visualstudio.com/DefaultCollection]$/$/myproject/Trunk;C16

```

Er zijn twee lokale branches, `master` en `featureA`, die staan voor het initiele uitgangspunt van de kloon (`Trunk` in TFVC) en een afgesplitste branch (`featureA` in TFVC). Je kunt ook zien dat de `tfv` “remote” een aantal refs heeft: `default` en `featureA`, die staan voor de TFVC branches. Git-tfs relateert de branch die je hebt gekloond aan `tfv/default`, en de andere krijgen hun eigen namen.

Iets anders om op te merken is de `git-tfs-id:` regels in de commit berichten. In plaats van tags, gebruikt git-tfs deze markeringen om een relatie te leggen tussen TFVC changesets en Git commits. Dit heeft de implicatie dat je Git commits een andere SHA-1 hash zullen hebben voor- en nadat ze naar TFVC zijn gepusht.

## Git-tf[s] Workflow

Onafhankelijk van de tool die je gebruikt, moet je een aantal Git configuratie instellingen inrichten om te voorkomen dat je wat problemen gaat krijgen.



```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

Het meest voor de hand liggende ding wat je hierna zult wilen doen is aan het project werken. TFVC en TFS hebben een aantal kenmerken die je workflow een stuk complexer kunnen maken:

1. Feature branches die niet voorkomen in TFVC maken het wat ingewikkelder. Dit heeft te maken met de **zeer** verschillende manieren waarop TFVC en Git branches weergeven.
2. Wees erop verdacht dat TFVC gebruikers in staat stelt om files van de server uit te checken (“checkout”), en ze op slot te zetten zodat niemand anders ze kan wijzigen. Uiteraard zal dit je niet stoppen om ze in je lokale repository te wijzigen, maar dit kan je in de weg zitten als het tijd wordt om je wijzigingen naar de TFVC server te pushen.
3. TFS kent het concept van “gated” checkins, waarbij een TFS bouw-test stap succesvol moet zijn verlopen voordat de checkin wordt toegelaten. Dit maakt gebruik van de “shelve” functie in TFVC, waar we hier niet in detail op in zullen gaan. Je kunt dit op een handmatige manier naspelen met git-tf, en git-tfs wordt geleverd met het `checkintool` commando die bewust is van

deze gates.

In het belang van beknoptheid, zullen we hier alleen het foutloze pad volgen, die de om de meeste van deze problemen heen leidt of die ze vermindert.

### Workflow: `git-tf`

Stel dat je wat werk gedaan hebt, je hebt een aantal Git commits op `master` gemaakt, en je staat gereed om je voortgang te delen op de TFVC server. Hier is onze Git repository:

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

We willen snapshot die in de `4178a82` commit zit nemen en die pushen naar de TFVC server. Maar laten we bij het begin beginnen: laten we eerst kijken of een van onze teamgenoten iets gedaan heeft sinds we voor het laatst verbonden waren:

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Het ziet er naar uit dat er iemand ook aan het werk is, en we hebben nu een uiteengelopen historie. Dit is waar Git uitblinkt, maar we hebben een keuze uit twee verschillende manieren om door te gaan:

1. Het maken van een merge commit voelt als een Git gebruiker als natuurlijk aan (dat is tenslotte wat `git pull` doet), en `git-tf` kan dit voor je doen met een simpele `git tf pull`. Wees je er echter van bewust dat TFVC zo niet in elkaar zit, en als je merge commits pusht dat je historie er aan beide kanten anders uit gaat zien, wat verwarring kan werken. Echter, als je van plan bent om

al je wijzigingen als één changeset op te sturen, is dit waarschijnlijk de eenvoudigste optie.

2. Rebasen maakt je commit historie lineair, wat inhoudt dat we de optie hebben om elk van onze Git commits te converteren in een TFVC changeset. Omdat dit de meeste opties openhoudt, raden we je aan om het op deze manier te doen; git-tf maakt het je zelfs makkelijk met `git tf pull --rebase`.

De keuze is aan jou. In dit voorbeeld zullen we rebasen:

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
    Team Project Creation Wizard
```

Nu zijn we gereed om een checkin te doen naar de TFVC server. Git-tf geeft je de keuze om een enkele changeset te maken die alle wijzigingen sinds de laatste (`--shallow`, wat standaard is) en het maken van een nieuwe changeset voor elke Git commit (`--deep`). In dit voorbeeld zullen we één changeset maken:

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
    Team Project Creation Wizard
```

Er is een nieuwe `TFS_C35348` tag, wat aangeeft dat TFVC exact dezelfde snapshot heeft opgeslagen als de `5a0e25e` commit. Het is belangrijk om op te merken dat niet elke Git commit perse een exacte evenknie in TFVC dient te hebben; de `6eb3eb5` commit bijvoorbeeld bestaat nergens op de server.

Dat is de belangrijkste workflow. Er zijn een aantal andere overwegingen die je in je achterhoofd

dient te houden:

- Er is geen branching. Git-tf kan alleen Git repositories maken van één TFVC branch per keer.
- Werk samen met TFVC of Git, maar niet beide. Verschillende git-tf clones van dezelfde TFVC repository kunnen verschillende SHA-1 commit-hashes hebben, wat de bron is van een niet aflatende stroom ellende.
- Als de workflow van je team samenwerken met Git inhoudt en periodiek synchroniseren met TFVC, maak met maar één van de Git repositories verbinding met TFVC.

### Workflow: `git-tfs`

Laten we hetzelfde scenario doorlopen waarbij we git-tfs gebruiken. Hier zijn de nieuwe commits die we gemaakt hebben op de `master`-branch in onze Git repository:

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfss/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 (tfss/default) Hello
* b75da1a New project
```

En laten we nu eens kijken of iemand anders werk gedaan heeft terwijl wij lekker aan het kloppen waren:

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfss/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|
| * 44cd729 (tfss/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 Hello
* b75da1a New project
```

Ja, we kunnen zien dat onze collega een nieuwe TFVC changeset heeft toegevoegd, die getoond wordt als de nieuwe `aea74a0` commit, en de `tfss/default` remote branch is verplaatst.

Net als met git-tf, hebben we twee basis opties hoe we deze uiteengelopen histories kunnen verwerken:

1. Rebase om een lineaire historie te behouden
2. Mergen om te bewaren wat er daadwerkelijk gebeurd is.

In dit geval zullen we een “deep” checkin uitvoeren, waar elke Git commit een TFVC changeset wordt, dus we willen gaan rebasen.

```
PS> git rebase tfs/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 Hello
* b75da1a New project
```

Nu zijn we klaar om onze bijdrage te leveren door onze code in te checken bij de TFVC server. Ze zullen hier het `rcheckin` command gebruiken om een TFVC changeset te maken voor elke Git commit in het pad van HEAD naar de eerste `tfs` remote branch die gevonden wordt (het `checkin` commando zou slechts één changeset maken, vergelijkbaar met het squashen van Git commits).

```
PS> git tfs rcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
  add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-
commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
  edit .git\dfs\default\workspace\ConsoleApplication1\ConsoleApplication1\Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-
commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 Hello
* b75da1a New project
```

Merk op hoe na elke succesvolle checkin naar de TFVC server, git-tfs het overblijvende werk rebased op wat het zojuist heeft gedaan. Dat is omdat het **git-tfs-id** veld onderaan de commitberichten wordt toegevoegd, wat de SHA-1 hashes verandert. Dit is precies volgens ontwerp, en er is niets om je zorgen over te maken, maar je moet je ervan bewust zijn dat dit gebeurt, vooral als je Git commits met anderen gaat delen.

TFS heeft veel functionaliteit die integreren met zijn eigen beheer systeem, zoals werkbonnen, aangewezen reviewers, gelaagde checkins (gated checkins) en zo voorts. Het kan nogal omslachtig zijn om met deze functionaliteit te werken met alleen maar een commando-regel tool, maar gelukkig stelt git-tfs je in staat om eenvoudig een grafische checkin tool aan te roepen:

```
PS> git tfs checkintool  
PS> git tfs ct
```

En dat ziet er ongeveer zo uit:

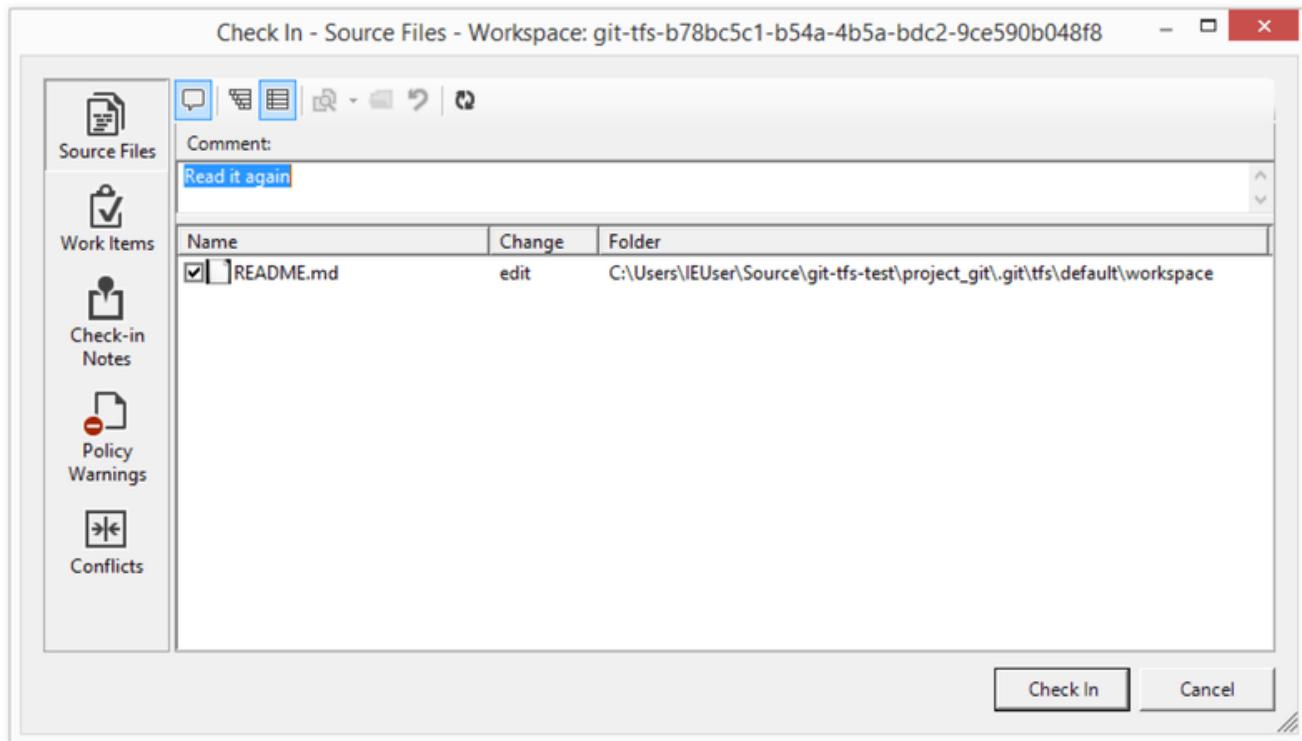


Figure 148. De git-tfs checkin tool.

Dit ziet er vertrouwd uit voor TFS gebruikers, omdat het dezelfde dialoog is die aangeroepen wordt vanuit Visual Studio.

Git-tfs laat je ook TFVC branches beheren vanuit je Git repository. Laten we als voorbeeld er eens een maken:

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfv/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfv/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Een branch maken in TFVC houdt het maken van een changeset in waar die branch nu in bestaat, en dit is voorgesteld als een Git commit. Merk ook op dat git-tfs de `tfv/featureBee` remote branch **aangemaakt** heeft, maar dat `HEAD` nog steeds naar `master` wijst. Als je op de nieuw aangemaakte branch wilt werken, zal je je nieuwe commits op de `1d54865` willen baseren, mogelijk door een topic branch van die commit te maken.

## Git en TFS samenvatting

Git-tf en Git-tfs zijn beide geweldige instrumenten om met een TFVC server te interacteren. Ze stellen je in staat om de kracht van Git lokaal te gebruiken, te voorkomen dat je voortdurend met de centrale TFVC server contact moet leggen, en je leven als ontwikkelaar veel eenvoudiger te maken, zonder je hele team te dwingen over te stappen op Git. Als je op Windows werkt (wat waarschijnlijk is als je team TFS gebruikt), zal je waarschijnlijk git-tfs willen gebruiken, omdat deze de meest complete functionaliteit biedt, maar als je op een ander platform werkt, zal je git-tf gebruiken die beperkter is. Zoals de meeste gereedschappen in dit hoofdstuk, zal je een van deze versie-beheer systemen als leidend kiezen, en de andere in een volgende rol gebruiken - Git of TFVC moet het middelpunt van de samenwerking zijn, niet beide.

# Migreren naar Git

Als je een bestaande codebase in een andere VCS hebt, en je hebt besloten om Git te gaan gebruiken, moet je je project op de een of andere manier migreren. Deze paragraaf behandelt een aantal importeerders voor veelgebruikte systemen, en laat je daarna zien hoe je je eigen importeur kunt ontwikkelen. Je zult kunnen lezen hoe gegevens uit een aantal van de grote professioneel gebruikte SCM systemen te importeren, omdat zij het leeuwendeel van de gebruikers vormen die overgaan, en omdat het eenvoudig is om instrumenten van hoge kwaliteit te pakken te krijgen.

## Subversion

Als je de vorige paragraaf leest over het gebruik van `git svn`, kan je eenvoudigweg deze instructies gebruiken om met `git svn clone` een repository te maken en daarna te stoppen met de Subversion server, naar een nieuwe Git server te pushen en die beginnen te gebruiken. Als je de historie wilt, kan je dat zo snel voor elkaar krijgen als je de gegevens uit de Subversion server kunt krijgen (en

dat kan even duren).

Deze import is echter niet perfect, en omdat het zo lang duurt, kan je eigenlijk ook meteen maar goed doen. Het eerste probleem is de auteur-informatie. In Subversion, heeft elke persoon die commit heeft gedaan een gebruikersnaam op het systeem die wordt opgenomen in de commit-informatie. De voorbeelden in de vorige paragraaf tonen `schacon` in bepaalde plaatsen, zoals de `blame` uitvoer en de `git svn log`. Als je dit beter op Git auteur-gegevens wilt mappen, moet je een relatie leggen van de Subversion gebruikers naar de Git auteurs. Maak een bestand genaamd `users.txt` die deze mapping-informatie heeft in een formaat als deze:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Om een lijst van auteur-namen te krijgen die SVN gebruikt, kan je dit aanroepen:

```
$ svn log --xml --quiet | grep author | sort -u | \
perl -pe 's/.*/$1 = /'
```

Dat maakt de loguitvoer in XML formaat aan, en behoudt vervolgens alleen de regels met auteur-informatie, verwijdert duplicaten en haalt de XML tags weg. (Dit werkt duidelijk alleen op een machine met `grep`, `sort`, en `perl` erop geïnstalleerd). Stuur daarna de uitvoer naar je `users.txt` bestand zodat je de overeenkomstige Git gebruiker gegevens naast elke regel kunt zetten.

Je kunt dit bestand aan `git svn` geven om het te helpen de auteur gegevens beter te mappen. Je kunt `git svn` ook vertellen de meta-data die Subversion normaalgesproken importeert niet mee te nemen, door `--no-metadata` mee te geven aan de `clone` of `init` commando's. Hierdoor ziet je `import` commando er ongeveer zo uit:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
--authors-file=users.txt --no-metadata --prefix "" -s my_project
$ cd my_project
```

Nu zou je een mooiere Subversion import moeten hebben in je `my_project` directory. In plaats van commits die er uit zien als dit

```
commit 37efa680e8473b615de980fa935944215428a35a
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date:  Sun May 3 00:12:22 2009 +0000

fixed install - go to trunk

git-svn-id: https://my-project.googlecode.com/svn/trunk@94 4c93b258-373f-11de-be05-5f7a86268029
```

zien ze er zo uit:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

    fixed install - go to trunk
```

Niet alleen ziet het Author veld er veel beter uit, maar de **git-svn-id** is er ook niet meer.

Je moet ook nog wat opruimen na de import. Onder andere moet je de vreemde referenties opruimen die **git svn** heeft gemaakt. Allereerst ga je de tags verplaatsen zodat ze echte tags zijn, in plaats van vreemde remote branches, en daarna verplaats je de overige branches zodat ze lokaal zijn.

Om de tags te verplaatsen zodat ze echte Git tags worden, roep je dit aan

```
$ for t in $(git for-each-ref --format='%(refname:short)' refs/remotes/tags); do git
tag ${t/tags\//} $t && git branch -D -r $t; done
```

Dit neemt de referenties die remote branches waren en begonnen met **refs/remotes/tags/** en maakt er echte (lichtgewicht) tags van.

Daarna verplaatsen we de overige referenties onder **refs/remotes** om er lokale branches van te maken:

```
$ for b in $(git for-each-ref --format='%(refname:short)' refs/remotes); do git branch
$b refs/remotes/$b && git branch -D -r $b; done
```

Het kan gebeuren dat je een aantal extra branches ziet die vooraf worden gegaan door **@xxx** (waar xxx een getal is), terwijl je in Subversion alleen maar een branch ziet. Dit is eigenlijk een Subversion kenmerk genaamd “peg-revisions”, wat iets is waar Git gewoonweg geen syntactische tegenhanger voor heeft. Vandaar dat **git svn** eenvoudigweg het svn versienummer aan de branchnaam toevoegt op dezelfde manier als jij dit zou hebben gedaan in svn om het peg-revisie van die branch te adresseren. Als je niet meer om de peg-revisies geeft, kan je ze simpelweg verwijderen:

```
$ for p in $(git for-each-ref --format='%(refname:short)' | grep @); do git branch -D
$p; done
```

Nu zijn alle oude branches echte Git branches en alle oude tags echte Git tags.

Er is nog een laatste ding om op te schonen: Helaas maakt **git svn** een extra branch aan met de naam **trunk**, wat overeenkomt met de standaard branch in Subversion, maar de **trunk**-referentie wijst naar dezelfde plek als **master**. Omdat **master** idiomatisch meer Git is, is hier de manier om die extra branch te verwijderen:

```
$ git branch -d trunk
```

Het laatste om te doen is om je nieuwe Git server als een remote toe te voegen en er naar te pushen. Hier is een voorbeeld van een server als een remote toe te voegen:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Omdat al je branches en tags ernaar wilt sturen, kan je nu dit aanroepen:

```
$ git push origin --all  
$ git push origin --tags
```

Al je branches en tags zouden nu op je nieuwe Git server moeten staan in een mooie, schone import.

## Mercurial

Omdat Mercurial and Git een redelijk overeenkomend model hebben om versies te representeren en omdat Git iets flexibeler is, is het converteren van een repository uit Mercurial naar Git minder omslachtig, gebruik makend van een instrument dat "hg-fast-export" heet, waar je een kopie van nodig gaat hebben:

```
$ git clone https://github.com/frej/fast-export.git
```

De eerste stap in de conversie is om een volledige kloon van de Mercurial repository die je wilt converteren te maken:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

De volgende stap is om een auteur-mapping bestand te maken. Mercurial is wat minder streng dan Git voor wat het in het auteur veld zet voor changesets, dus dit is een goed moment om schoon schip te maken. Het aanmaken hiervan is een enkele commando regel in een **bash** shell:

```
$ cd /tmp/hg-repo  
$ hg log | grep user: | sort | uniq | sed 's/user: */' > ../authors
```

Dit duurt een paar tellen, afhankelijk van de lengte van de geschiedenis van je project, en nadien ziet het **/tmp/authors** bestand er ongeveer zo uit:

```
bob
bob@localhost
bob <bob@company.com>
bob jones <bob <AT> company <DOT> com>
Bob Jones <bob@company.com>
Joe Smith <joe@company.com>
```

In dit voorbeeld, heeft dezelfde persoon (Bob) changesets aangemaakt onder vier verschillende namen, waarvan er één er wel correct uitziet, en één ervan zou voor een Git commit helemaal niet geldig zijn. Hg-fast-export laat ons dit corrigeren door elke regel in een instructie te veranderen: "<invoer>"="<uitvoer>", waarbij **<invoer>** in een **<uitvoer>** wordt gewijzigd. Binnen de **<invoer>** en **<uitvoer>** tekenreeksen, worden alle *escaped* reeksen die door de python **string\_escape** encoding worden begrepen ondersteund. Als het auteur mapping-bestand geen passende **<invoer>** regel heeft, wordt deze ongewijzigd doorgestuurd naar Git. Als alle gebruikersnamen er goed uitzien, hebben we dit bestand helemaal niet nodig. In ons voorbeeld, willen we dat ons bestand er zo uit ziet:

```
"bob"="Bob Jones <bob@company.com>"
"bob@localhost"="Bob Jones <bob@company.com>"
"bob <bob@company.com>"="Bob Jones <bob@company.com>"
"bob jones <bob <AT> company <DOT> com>"="Bob Jones <bob@company.com>"
```

Hetzelfde soort mapping bestand kan worden gebruikt om branches en tags te hernoemen als de Mercurial naam niet wordt toegestaan door Git.

De volgende stap is om onze nieuwe Git repository aan te maken en het volgende export script aan te roepen:

```
$ git init /tmp/converted
$ cd /tmp/converted
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

De **-r** vlag vertelt hg-fast-export waar het de Mercurial repository kan vinden die we willen converteren, en de **-A** vlag vertelt het waar het auteur-mapping bestand te vinden is. Het script verwerkt Mercurial changesets en converteert ze in een script voor de "fast-import" functie (die we iets later zullen bespreken). Dit duurt even (al is het *veel* sneller dan het via het netwerk zou zijn), en de uitvoer is nogal breedsprakig:

```

$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
Loaded 4 authors
master: Exporting full revision 1/22208 with 13/0/0 added/changed/removed files
master: Exporting simple delta revision 2/22208 with 1/1/0 added/changed/removed files
master: Exporting simple delta revision 3/22208 with 0/1/0 added/changed/removed files
[...]
master: Exporting simple delta revision 22206/22208 with 0/4/0 added/changed/removed files
master: Exporting simple delta revision 22207/22208 with 0/2/0 added/changed/removed files
master: Exporting thorough delta revision 22208/22208 with 3/213/0
added/changed/removed files
Exporting tag [0.4c] at [hg r9] [git :10]
Exporting tag [0.4d] at [hg r16] [git :17]
[...]
Exporting tag [3.1-rc] at [hg r21926] [git :21927]
Exporting tag [3.1] at [hg r21973] [git :21974]
Issued 22315 commands
git-fast-import statistics:
-----
Alloc'd objects: 120000
Total objects: 115032 ( 208171 duplicates ) )
blobs : 40504 ( 205320 duplicates 26117 deltas of 39602
attempts)
trees : 52320 ( 2851 duplicates 47467 deltas of 47599
attempts)
commits: 22208 ( 0 duplicates 0 deltas of 0
attempts)
tags : 0 ( 0 duplicates 0 deltas of 0
attempts)
Total branches: 109 ( 2 loads )
marks: 1048576 ( 22208 unique )
atoms: 1952
Memory total: 7860 KiB
pools: 2235 KiB
objects: 5625 KiB
-----
pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr = 90430
pack_report: pack_mmap_calls = 46771
pack_report: pack_open_windows = 1 / 1
pack_report: pack_mapped = 340852700 / 340852700
-----
$ git shortlog -sn
 369 Bob Jones
 365 Joe Smith

```

Meer valt er eigenlijk niet over te vertellen. Alle Mercurial tags zijn geconverteerd naar Git tags, en Mercurial branches en boekleggers (bookmarks) zijn geconverteerd naar Git branches. Nu ben je klaar om de repository naar zijn nieuwe server-thuis te sturen:

```
$ git remote add origin git@my-git-server:myrepository.git  
$ git push origin --all
```

## Bazaar

Bazaar is een DVCS tool die veel op Git lijkt, en als gevolg is het redelijk probleemloos om een Bazaar repository in een van Git te converteren. Om dit voor elkaar te krijgen, moet je de **bzr-fastimport**-plugin importeren.

### De bzr-fastimport plugin verkrijgen

De procedure om de fastimport plugin te installeren verschilt op een UNIX-achtige besturingssysteem en op Windows. In het eerste geval, is het het eenvoudigste om het **bzr-fastimport**-pakket te installeren, en dat zal alle benodigde afhankelijkheden installeren.

Bijvoorbeeld, met Debian en afgeleiden, zou je het volgende doen:

```
$ sudo apt-get install bzr-fastimport
```

Met RHEL, zou je het volgende doen:

```
$ sudo yum install bzr-fastimport
```

Met Fedora, vanaf release 22, is dnf de nieuwe package manager:

```
$ sudo dnf install bzr-fastimport
```

Als het pakket niet beschikbaar is, kan je het als een plugin installeren:

```
$ mkdir --parents ~/.bazaar/plugins      # maakt de benodigde folders voor de plugins  
$ cd ~/.bazaar/plugins  
$ bzr branch lp:bzr-fastimport fastimport  # iimporteert de fastimport plugin  
$ cd fastimport  
$ sudo python setup.py install --record=files.txt  # installeert de plugin
```

Om deze plugin te laten werken, heb je ook de **fastimport** Python module nodig. Je kunt controleren of deze aanwezig is of niet en het installeren met de volgende commando's:

```
$ python -c "import fastimport"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named fastimport
$ pip install fastimport
```

Als het niet beschikbaar is, kan je het downloaden van het adres <https://pypi.python.org/pypi/fastimport/>.

In het tweede geval (op Windows), wordt **bzr-fastimport** automatisch geïnstalleerd met de standalone versie en de standaard installatie (laat alle checkboxen aangevinkt). Dus in dit geval hoef je niets te doen.

Vanaf dat moment, is de manier waarop je een Bazaar repository afhankelijk van of je een enkele branch hebt, of dat je op een repository werkt die meerdere branches heeft.

### Project met een enkele branch

Ga met **cd** in de directory die jouw Bazaar repository bevat en initialiseer de Git repository:

```
$ cd /path/to/the/bzr/repository
$ git init
```

Nu kan je eenvoudigweg je Bazaar repository exporteren en converteren in een Git repository met het volgende commando:

```
$ bzr fast-export --plain . | git fast-import
```

Afhankelijk van de grootte van het project, wordt jouw Git repository gebouwd in een periode varierend van een paar seconden tot een paar minuten.

### Het geval van een project met een hoofd-branch en een werk-branch

Je kunt ook een Bazaar repository importeren dat branches bevat. Laten we aannemen dat je twee branches hebt: een vertegenwoordigt de hoofd-branch (`myProject/trunk`), de andere is de werk-branch (`myProject/work`).

```
$ ls
myProject/trunk myProject/work
```

Maak de Git repository en ga er met **cd** erheen:

```
$ git init git-repo
$ cd git-repo
```

Pull de master-branch in git:

```
$ bzr fast-export --export-marks=../marks.bzr ../myProject/trunk | \  
git fast-import --export-marks=../marks.git
```

Pull de werk-branch in Git:

```
$ bzr fast-export --marks=../marks.bzr --git-branch=work ../myProject.work | \  
git fast-import --import-marks=../marks.git --export-marks=../marks.git
```

git branch zal je nu de master-branch alsook de work-branch laten zien. Controleer de logs om je ervan te vergewissen dat ze volledig zijn en verwijder de marks.bzr en de marks.git bestanden.

## De staging area synchroniseren

Onafhankelijk van het aantal branches die je had en de import-methode die je gebruikt hebt, is je staging area niet gesynchroniseerd met HEAD, en met het importeren van verschillende branches, is je werk-directory ook niet gesynchroniseerd. Deze situatie is eenvoudig op te lossen met het volgende commando:

```
$ git reset --hard HEAD
```

## Het negeren van de bestanden die met .bzrignore werden genegeerd

Laten we nu eens kijken naar de te negeren bestanden. Het eerste wat we moeten doen is .bzrignore naar .gitignore hernoemen. Als het .bzrignore bestand een of meerdere regels bevat die beginnen met "!!" of "RE:", zal je deze moeten wijzigen en misschien verscheidene .gitignore-bestanden maken om precies dezelfde bestanden te negeren die Bazaar ook negeerde.

Uiteindelijk zal je een commit moeten maken die deze wijziging voor de migratie bevat:

```
$ git mv .bzrignore .gitignore  
$ # modify .gitignore if needed  
$ git commit -am 'Migration from Bazaar to Git'
```

## Jouw repository naar de server sturen

We zijn er! Je kunt nu de repository naar zijn nieuwe thuis-server pushen:

```
$ git remote add origin git@my-git-server:mygitrepository.git  
$ git push origin --all  
$ git push origin --tags
```

Je Git repository is klaar voor gebruik.

# Perforce

Het volgende systeem waar we naar gaan kijken is het importeren uit Perforce. Zoals hierboven besproken, zijn er twee manieren om Git en Perforce met elkaar te laten praten: git-p4 en Perforce Git Fusion.

## Perforce Git Fusion

Met Git Fusion verloopt dit proces vrijwel pijnloos. Configureer alleen je project settings, user mappings en branches met behulp van een configuratie bestand (zoals behandeld in [Git Fusion](#)), en clone de repository. Git Fusion geeft je iets wat eruit ziet als een echte Git repository, die dan klaar is om naar een reguliere Git host te pushen als je dat wilt. Je kunt zelfs Perforce als je Git host gebruiken als je dat wilt.

### Git-p4

Git-p4 kan ook als een importeer gereedschap werken. Als voorbeeld zullen we het Jam project importeren van de Perforce Public Depot. Om je werkstation in te richten, moet je de P4PORT omgevingsvariabele exporteren zodat deze wijst naar het Perforce depot:

```
$ export P4PORT=public.perforce.com:1666
```



Om dit mee te kunnen doen, moet je een Perforce depot hebben om mee te verbinden. Wij zullen het publieke depot op public.perforce.com gebruiken in onze voorbeelden, maar je kunt elk depot waar jetoegang toe hebt gebruiken.

Roep het `git p4 clone` commando aan om het Jam project van de Perforce server te importeren, waarbij je het pad van het depot en het project en het pad waar je het project in wilt importeren meegeeft:

```
$ git-p4 clone //guest/perforce_software/jam@all p4import
Importing from //guest/perforce_software/jam@all into p4import
Initialized empty Git repository in /private/tmp/p4import/.git/
Import destination: refs/remotes/p4/master
Importing revision 9957 (100%)
```

Dit specifieke project heeft maar een branch, maar als je branches hebt die geconfigureerd zijn met branch views (of alleen een set directories), kan je de `--detect-branches` vlag gebruiken bij `git p4 clone` om alle branches van het project ook te importeren. Zie [Branchen](#) voor wat meer diepgang op dit onderwerp.

Op dit punt ben je bijna klaar. Als je naar de `p4import` directory gaat en `git log` aanroeft, kan je je geïmporteerde werk zien:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800
```

Correction to line 355; change </UL> to </OL>.

```
[git-p4: depot-paths = "//public/jam/src/": change = 8068]
```

```
commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800
```

Fix spelling error on Jam doc page (cummulative -> cumulative).

```
[git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

Je kunt zien dat **git-p4** een identifierend element heeft achtergelaten in elk commit-bericht. Je kunt dit element prima daar laten, in het geval dat je in de toekomst naar het Perforce change nummer moet refereren. Echter, als je dit element wilt weghalen, is dit het moment om het te doen - voordat je begint te werken met de nieuwe repository. Je kunt **git filter-branch** gebruiken om de element-tekenreeksen en masse te verwijderen:

```
$ git filter-branch --msg-filter 'sed -e "/^\\[git-p4:/d"
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)
Ref 'refs/heads/master' was rewritten
```

Als je **git log** aanroeft, kan je zien dat alle SHA-1 checksums voor de commits zijn gewijzigd, maar de **git-p4** tekenreeksen staan niet meer in de commit-berichten:

```
$ git log -2
commit b17341801ed838d97f7800a54a6f9b95750839b7
Author: giles <giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800
```

Correction to line 355; change </UL> to </OL>.

```
commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800
```

Fix spelling error on Jam doc page (cummulative -> cumulative).

Je import is klaar om te worden gepusht naar je nieuwe Git server.

## TFS

Als je team haar versiebeheer uit TFVC naar Git gaat converteren, wil je de hoogst-betrouwbare conversie gebruiken die je maar kunt krijgen. Dit houdt in dat, hoewel we zowel git-tfs als git-tf in de samenwerkings-paragraaf hebben behandeld, zullen we hier alleen git-tfs behandelen, omdat git-tfs branches ondersteunt, en deze beperking het vrijwel onmogelijk maakt om git-tf hiervoor te gebruiken.



Dit is een eenrichtings conversie. De Git repository die hier wordt aangemaakt kan geen verbinding meer leggen met het oorspronkelijk TFVC project.

Het eerste om te doen is gebruikersnamen mappen. TFVC is nogal ruimdenkend met wat er in het auteur veld gaat voor changesets, maar Git wil een voor de mens leesbare naam en email adres hebben. Je kunt deze informatie als volgt van het `tf` commando-regel client krijgen:

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

Dit pakt alle changesets in de geschiedenis van het project en zet dit in het `AUTHORS_TMP` bestand die we verder gaan verwerken om de gegevens uit het *User* kolom (de tweede) te halen. Open het bestand en bekijk op welke karakter de kolom begint en eindigt en vervang, in de volgende commando-regel, de parameters `11-20` van het `cut` commando met de waarden die jij gevonden hebt:

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | sort | uniq > AUTHORS
```

Het `cut` commando behoudt alleen de karakters tussen 11 en 20 van elke regel. Het `tail` commando slaat de eerste twee regels over, die kolom-koppen en ASCII-art onderstrepingen zijn. Het resultaat van dit alles wordt aan `sort` en `uniq` doorgegeven om duplicaten te verwijderen en bewaard in een bestand genaamd `AUTHORS`. De volgende stap is handmatig; om git-tfs van dit bestand gebruik te laten maken, moet elke regel in dit formaat staan:

```
DOMAIN\username = User Name <email@address.com>
```

Het gedeelte links is het “User” veld van TFVC, en het gedeelte rechts van het gelijk-teken is de gebruikersnaam die voor Git commits gaat worden gebruikt.

Als je dit bestand eenmaal hebt, is de volgende stap om te nemen een volledige kloon van het TFVC project waar je in bent geïnteresseerd te maken:

```
PS> git tfs clone --with-branches --authors=AUTHORS  
https://username.visualstudio.com/DefaultCollection $/project/Trunk project_git
```

Vervolgens wil je de `git-tfs-id` gedeeltes aan het eind van de commit-berichten opruimen. Het volgende commando gaat dit doen:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$//g"' '--' --all
```

Dit gebruikt het `sed` commando van de Git-bash omgeving om elke regel die begint met “git-tfs-id:” met leegte te vervangen, dit Git vervolgens dan zal negeren.

Als dit eenmaal gedaan is, ben je klaar om een nieuwe remote toe te voegen, al je branches te pushen, en je team is klaar om met Git te gaan werken.

## Importeren op maat

Als jouw systeem niet een van de bovenstaande is, moet je op het internet gaan zoeken naar een importeerder - goede importeerders zijn beschikbaar voor vele andere systemen, inclusief CVS, Clear Case, Visual Source Safe en zelfs een directory met archieven. Als geen van die tools voor jou geschikt zijn, je hebt een heel obscure tool, of je hebt om een andere reden een meer aangepaste importeer proces nodig, dan moet je `git fast-import` gebruiken. Dit commando leest eenvoudige instructies van stdin om specifieke Git gegevens te schrijven. Het is veel eenvoudiger om op deze manier Git objecten aan te maken dan de rauwe Git commando’s te gebruiken of om zelf de rauwe objecten te schrijven (zie [Git Binnenwerk](#) voor meer informatie). Op deze manier kan je een import script schrijven die de benodigde informatie uit het te importeren systeem leest en op stdout eenvoudige instructies afdrukt. Je kunt dit programma dan aanroepen en de uitvoer doorgeven aan `git fast-import` met een pipe-instructie.

Om een snelle demonstratie te geven, ga je een eenvoudige importeerder schrijven. Stel dat je in `current` aan het werk bent, je maakt op gezette tijden een backup van je project door de directory naar een andere te kopiëren met een datum-tijd indicatie genaamd `back_YYYY_MM_DD`, en je wilt dit importeren in Git. Je directory-structuur ziet er zo uit:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

Om een Git directory te importeren, moet je weten hoe Git haar gegevens opslaat. Zoals je je zult herinneren, is Git in de basis een geschakelde lijst van commit objecten die verwijzen naar een snapshot van de inhoud. Alles wat je hoeft te doen is `fast-import` te vertellen wat de snapshots van de inhoud zijn, welke commit-gegevens hiernaar verwijzen en de volgorde waarin ze staan. Je strategie zal zijn om een voor een door de snapshots te gaan en commits te maken met de inhoud van elke directory, en elke commit terug te laten verwijzen naar de vorige.

Zoals we in [Een voorbeeld van Git-afgedwongen beleid](#) gedaan hebben, schrijven we dit in Ruby, omdat dit is waar we gewoonlijk mee werken en het redelijk eenvoudig te lezen is. Je kunt dit voorbeeld redelijk eenvoudig in iets schrijven waar je bekend mee bent - het moet gewoon de juiste informatie naar `stdout` uitvoeren. En als je op Windows draait, betekent dit dat je ervoor moet zorgen dat je geen carriage returns aan het eind van je regels gebruikt - git fast-import is erg secuur in het alleen accepteren van line feeds (LF) niet de carriage return line feeds (CRLF) die door

Windows wordt gebruikt.

Om te beginnen, spring je in de doel directory en identificeert elke subdirectory, die elk een snapshot is van wat je wilt importeren als een commit. Je springt in elke subdirectory en drukt de commando's af die nodig zijn om het te exporteren. Je hoofdlus ziet er zo uit:

```
last_mark = nil

# loop through the directories
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end
```

Je roept `print_export` in elke directory aan, die de inhoud en kenmerk (mark) van de vorige snapshot neemt en je de inhoud en kenmerk van deze teruggeeft; op die manier kan je ze goed koppelen.

“Mark” is de term die `fast-import` gebruikt voor een identificatie die je aan een commit geeft; tijdens het aanmaken van commit geef je elk een kenmerk die je kunt gebruiken om als koppeling vanaf andere commits. Dus, het eerste wat je moet doen in je `print_export` methode is een kenmerk genereren van de naam van de directory:

```
mark = convert_dir_to_mark(dir)
```

Je doet dit door een reeks (array) directories te maken en de indexwaarde als het kenmerk te gebruiken, omdat een kenmerk een integer dient te zijn. Je methode ziet er zo uit:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Nu je een integer representatie hebt van je commit, heb je een datum nodig voor de commit metadata. Omdat de datum in de naam van de directory zit, ga je het eruit halen. De volgende regel in je `print_export` bestand is:

```
date = convert_dir_to_date(dir)
```

where `convert_dir_to_date` is defined as:

```
def convert_dir_to_date(dir)
  if dir == 'current'
    return Time.now().to_i
  else
    dir = dir.gsub('back_', '')
    (year, month, day) = dir.split('_')
    return Time.local(year, month, day).to_i
  end
end
```

Dat retourneert een integerwaarde voor de datum van elke directory. Het laatste stukje meta-informatie dat je nodig hebt voor elke commit zijn de gegevens van de committer, die je hardgedeerd hebt in een globale variabele:

```
$author = 'John Doe <john@example.com>'
```

Nu ben je klaar om de commit data af te drukken voor je importeerder. De initiële informatie geeft aan dat je een commit object definieert en op welke branch deze staat, gevolgd door het kenmerk die je hebt gegenereert, de informatie van de committer en het commit bericht, en dan de vorige commit als die er is. De code ziet er zo uit:

```
# print the import information
puts 'commit refs/heads/master'
puts 'mark :' + mark
puts "committer #{$author} #{date} -0700"
export_data('imported from ' + dir)
puts 'from :' + last_mark if last_mark
```

Je geeft de tijdzone (-0700) hardgedeerd, omdat dit nu eenmaal makkelijk is. Als je vanuit een ander systeem importeert, moet je de tijdzone als een relatieve verschuiving (offset) weergeven. Het commit bericht moet worden uitgedrukt in een speciaal formaat:

```
data (size)\n(contents)
```

Het formaat bestaat uit het woord `data`, de grootte van de te lezen gegevens, een nieuwe regel en tot slot de gegevens. Omdat je hetzelfde formaat later nodig hebt om de bestandsinhoud te specificeren, maak je een hulpmethode, `export_data`:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Wat er nu nog overblijft is het specificeren van de bestandsinhoud voor elk snapshot. Dit is makkelijk, omdat je elk van deze in een directory hebt - je kunt het `deleteall` commando afdrukken gevuld door de inhoud van elk bestand in de directory. Git zal dan elke snapshot op de juiste manier opslaan:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Let op: Omdat veel systemen hun revisies zien als wijzigingen van de ene commit naar de ander, kan fast-import ook commando's accepteren waarbij elke commit kan worden aangegeven welke bestanden er zijn toegevoegd, verwijderd of gewijzigd en welke nieuwe elementen erbij zijn gekomen. Je kunt de verschillen tussen de snapshots berekenen en alleen deze gegevens meegeven, maar het is veel ingewikkelder om dit te doen - je kunt net zo makkelijk Git alle gegevens meegeven en het hem laten uitzoeken. Als dit beter past bij jouw gegevens, neem dan de `fast-import` man page door voor de details hoe je deze gegevens moet doorgeven in dat geval.

Het formaat om de nieuwe bestandsinhoud weer te geven of om een gewijzigd bestand te specificeren met de nieuwe inhoud is als volgt:

```
M 644 inline path/to/file
data (size)
(file contents)
```

Hier is 644 de mode (als je aanroepbare bestanden hebt, moet je dit detecteren en in plaats daarvan 755 opgeven), en in de code (inline) geef je de inhoud direct achter deze regel weer. Je `inline_data` methode ziet er zo uit:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Je hergebruikt de `export_data` methode die je eerder hebt gedefinieerd, omdat dit formaat hetzelfde is als waarmee je de commit bericht gegevens specificeert.

Het laatste wat je nog moet doen is om het huidige kenmerk te retourneren, zodat het kan worden doorgegeven in de volgende iteratie:

```
return mark
```

Als je op Windows draait moet je je ervan verzekeren dat je een extra stap toevoegt. Zoals eerder opgemerkt, gebruikt Windows CRLF voor nieuwe regel tekens waar `git fast-import` alleen LF verwacht. Om dit probleem te verhelpen en `git fast-import` blij te maken, moet je ruby vertellen om LF te gebruiken in plaats van CRLF:



```
$stdout.binmode
```

Dat is 't. Hier is het script in zijn totaliteit:

```
#!/usr/bin/env ruby

$stdout.binmode
$author = "John Doe <john@example.com>"

$marks = []
def convert_dir_to_mark(dir)
    if !$marks.include?(dir)
        $marks << dir
    end
    ($marks.index(dir)+1).to_s
end

def convert_dir_to_date(dir)
    if dir == 'current'
        return Time.now().to_i
    else
        dir = dir.gsub('back_', '')
        (year, month, day) = dir.split('_')
        return Time.local(year, month, day).to_i
    end
end

def export_data(string)
    print "data #{string.size}\n#{string}"
end

def inline_data(file, code='M', mode='644')
    content = File.read(file)
    puts "#{code} #{mode} inline #{file}"
    export_data(content)
end

def print_export(dir, last_mark)
    date = convert_dir_to_date(dir)
```

```

mark = convert_dir_to_mark(dir)

puts 'commit refs/heads/master'
puts "mark :#{mark}"
puts "committer #{\$author} #{date} -0700"
export_data("imported from #{dir}")
puts "from :#{last_mark}" if last_mark

puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
mark
end

# Loop through the directories
last_mark = nil
Dir.chdir(ARGV[0]) do
  Dir.glob("*").each do |dir|
    next if File.file?(dir)

    # move into the target directory
    Dir.chdir(dir) do
      last_mark = print_export(dir, last_mark)
    end
  end
end

```

Als je dit script aanroept, krijg je inhoud dat er ongeveer zo uit ziet:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer John Doe <john@example.com> 1388649600 -0700
data 29
imported from back_2014_01_02deleteall
M 644 inline README.md
data 28
# Hello

This is my readme.
commit refs/heads/master
mark :2
committer John Doe <john@example.com> 1388822400 -0700
data 29
imported from back_2014_01_04from :1
deleteall
M 644 inline main.rb
data 34
#!/bin/env ruby

puts "Hey there"
M 644 inline README.md
(...)
```

Om de importeerder te laten draaien, geeft je de uitvoer met een pipe door aan `git fast-import` terwijl je in de Git directory staat waar je naartoe wilt importeren. Je kunt een nieuwe directory aanmaken en daarna `git init` daarin aanroepen als een begin, en daarna je script aanroepen:

```

$ git init
Initialized empty Git repository in /opt/import_to/.git/
$ ruby import.rb /opt/import_from | git fast-import
git-fast-import statistics:
-----
Alloc'd objects:      5000
Total objects:       13 (      6 duplicates      )
blobs :             5 (      4 duplicates      ) 3 deltas of 5
attempts)
trees :             4 (      1 duplicates      ) 0 deltas of 4
attempts)
commits:            4 (      1 duplicates      ) 0 deltas of 0
attempts)
tags   :             0 (      0 duplicates      ) 0 deltas of 0
attempts)
Total branches:     1 (      1 loads      )
marks:              1024 (      5 unique      )
atoms:                2
Memory total:       2344 KiB
pools:              2110 KiB
objects:             234 KiB
-----
pack_report: getpagesize() = 4096
pack_report: core.packedGitWindowSize = 1073741824
pack_report: core.packedGitLimit = 8589934592
pack_report: pack_used_ctr = 10
pack_report: pack_mmap_calls = 5
pack_report: pack_open_windows = 2 / 2
pack_report: pack_mapped = 1457 / 1457
-----
```

Zoals je kunt zien, als het succesvol eindigt, geeft het je een bergje statistieken wat het allemaal heeft bereikt. In dit geval, heb je in totaal 13 objecten geïmporteerd voor 4 commits in 1 branch. Je kunt nu `git log` aanroepen om je nieuwe historie te bekijken:

```

$ git log -2
commit 3caa046d4aac682a55867132ccdfbe0d3fdee498
Author: John Doe <john@example.com>
Date:   Tue Jul 29 19:39:04 2014 -0700

    imported from current

commit 4afc2b945d0d3c8cd00556fbe2e8224569dc9def
Author: John Doe <john@example.com>
Date:   Mon Feb 3 01:00:00 2014 -0700

    imported from back_2014_02_03
```

Kijk eens aan - een mooie, schone Git repository. Het is belangrijk om op te merken dat er niets is uitgecheckt - je hebt initieel nog geen enkel bestand in je werk directory. Om deze te krijgen, moet je je branch resetten tot waar `master` nu is:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
$ ls
README.md main.rb
```

Je kunt nog veel meer doen met het `fast-import` gereedschap - verschillende modes behandelen, binaire gegevens, meerdere branches en mergen, tags, voortgangs-indicators en meer. Een aantal voorbeelden van meer ingewikkelde scenarios zijn beschikbaar in de `contrib/fast-import` directory van de Git broncode.

## Samenvatting

Je zou je op je gemak moeten voelen om Git als een client te gebruiken voor andere versiebeheer systemen, of bijna ieder bestaand repository te importeren in een Git versie zonder gegevens te verliezen. Het volgende hoofdstuk zal het ruwe binnenwerk van Git behandelen, zodat je iedere byte kunt bewerken, als dat nodig is.

# Git Binnenwerk

Je bent misschien meteen doorgegaan naar dit hoofdstuk vanuit een eerder hoofdstuk, of je bent misschien hier beland nadat je de rest van het boek gelezen hebt - hoe dan ook, dit is waar we werking onder de motorkap en de implementatie van Git gaan behandelen. We zijn van mening dat het leren van deze informatie fundamenteel belangrijk was om te begrijpen hoe nuttig en krachtig Git is, maar anderen hebben ons duidelijk gemaakt dat het juist verwarrend kan zijn en onnodig complex voor beginners. Daarom hebben we deze behandeling als laatste hoofdstuk in het boek opgenomen zodat je het vroeg of laat in je leerproces kunt gaan lezen. We laten deze beslissing geheel aan jou over.

Maar nu je hier toch bent, laten we beginnen. Ten eerste, als het nog niet duidelijk mocht zijn, Git is in wezen een op inhoud-adresseerbaar bestandssysteem met een VCS (versiebeheer) gebruikers interface erbovenop geschreven. Je zult straks meer lezen over de betekenis hiervan.

In de beginlagen van Git (vooral voor 1.5), was de gebruikers interface veel complexer omdat de nadruk lag op dit bestandssysteem en veel minder op een strak versiebeheer systeem. In de laatste paar jaren is de gebruikersinterface bijgeslepen totdat het net zo gemakkelijk te gebruiken was als ieder ander systeem; maar vaak hangt het stereotype nog rond van de vroege Git interface die zo complex was en moeilijk te leren.

Het op inhoud-adresseerbare bestandssysteemlaag is ongelofelijk gaaf, dus we zullen die eerst in dit hoofdstuk behandelen; daarna zullen je vertellen over de transport mechanismen en het taken voor onderhoud van de repository waar je op den duur mee te maken kunt krijgen.

## Binnenwerk en koetswerk (plumbing and porcelain)

Dit boek behandelt primair hoe Git te gebruiken met ongeveer 30 werkwoorden als `checkout`, `branch`, `remote`, enzovoort. Maar omdat Git initieel een gereedschapkist was voor een versiebeheersysteem in plaats van een compleet gebruikersvriendelijke versiebeheersysteem, heeft het een aantal werkwoorden die het grondwerk verzorgen en die ontworpen zijn om op een UNIX manier te worden gekoppeld of vanuit scripts te worden aangeroepen. Aan deze commando's worden over het algemeen gerefereerd als "plumbing" (binnenwerk) commando's, en de meer gebruikersvriendelijke commando's worden "porcelain" (koetswerk) genoemd.

Je zult nu wel gemerkt hebben dat de eerste negen hoofdstukken van het boek houden zich vrijwel exclusief bezig met porcelain commando's. Maar in dit hoofdstuk ga je meerendeels te maken krijgen met de plumbing commando's op het diepere niveau, omdat deze je toegang geven tot het binnenwerk van Git, en om je te laten zien hoe en waarom Git doet wat het doet. Veel van deze commando's zijn niet bedoeld voor handmatig gebruik op de commando-regel, maar meer bedoeld als gebruik als onderdeel voor nieuwe gereedschappen en eigengemaakte scripts.

Wanneer je `git init` aanroeft in een nieuwe of bestaande directory, maakt Git de `.git` directory aan, waar vrijwel alles wat Git opslaat en bewerkt aanwezig is. Als je een backup wilt maken of je repository wilt klonen, geeft het kopieren van deze ene directory naar elders je vrijwel alles wat je nodig hebt. Deze hele hoofdstuk behandelt eigenlijk het spul dat je in deze directory kunt zien. Dit is hoe een nieuw-gemaakte `.git`-directory er normaal gesproken uitziet:

```
$ ls -F1
config
description
HEAD
hooks/
info/
objects/
refs/
```

Afhankelijk van je versie van Git, zou je wat extra inhoud kunnen zien, maar dit is een versie `git init` repository — dit is wat je standaard ziet. Het `description` bestand wordt alleen gebruikt door het GitWeb programma, dus maak je er geen zorgen over. Het `config` bestand bevat jouw project-specifieke configuratie opties, en de `info` directory bevat een globale exclude bestand voor genegeerde patronen die je niet wilt tracken in een `.gitignore` bestand. De `hooks` directory bevat de hook-scripts voor de werkstation of server kant, waar we dieper op zijn ingegaan in [Git Hooks](#).

Resten vier belangrijke regels: de `HEAD` en (nog te maken) `index` bestanden, en de `objects` en `refs` directories. Dit zijn de kern-onderdelen van Git. De `objects` directory bevat alle inhoud voor jouw database, de `refs` directory bevat verwijzingen naar commit objecten in die gegevens (branches, tags, remotes en zo meer), het `HEAD` bestand verwijst naar de branch die je op dit moment uitgecheckt hebt, en het `index` bestand is waar Git de informatie over je staging gebied bewaart. We zullen hierna dieper ingaan op elk van deze onderdelen om te zien hoe Git werkt.

## Git objecten

Git is een op inhoud-adresseerbaar bestandssysteem. Mooi. Wat betekent dat nu eigenlijk? Het betekent dat in het hart van Git een eenvoudig sleutel-waarde gegevens opslag zit. Je kunt elke vorm van inhoud erin stoppen, en het zal je een sleutel teruggeven die je kunt gebruiken om de inhoud op elk gewenst moment weer op te halen.

Om dit te laten zien, zal je het plumbing commando `hash-object` gebruiken, die gegevens aanneemt, dit opslaat in je `.git` directory en je de sleutel teruggeeft waarmee de gegevens zijn opgeslagen.

Eerst, moet je een nieuwe Git repository initialiseren en vaststellen dat er niets in de `objects` directory zit:

```
$ git init test
Initialized empty Git repository in /tmp/test/.git/
$ cd test
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
```

Git heeft nu de `objects` directory geinitialiseerd en daarin `pack` en `info` subdirectories aangemaakt, maar er zijn geen reguliere bestanden. Nu sla je wat tekst op in je Git database:

```
$ echo 'test content' | git hash-object -w --stdin  
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

In het eenvoudigste vorm neemt het `git hash-object` de inhoud die je hem hebt gegeven en alleen maar de unieke sleutel teruggeven die *zou worden* gebuikt als je het zou opslaan in de Git database. De `-w` vertelt `hash-object` om niet simpelweg de sleutel terug te geven maar om het object op te slaan in de database. Tot slot vertelt de `--stdin` optie het commando om de te verwerken inhoud van stdin te lezen; als je dit niet aangeeft, verwacht het commando een bestands-argument aan het eind van het commando met daarin de inhoud die moet worden verwerkt.

De uitvoer van het commando is een 40-karakter checksum hash. Dit is de SHA-1 hash—een controlegetal van de inhoud die je opslaat plus een header, waar je iets later meer over gaat lezen. Nu kan je zien hoe Git je gegevens heeft opgeslagen:

```
$ find .git/objects -type f  
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Als je weer je `objects` directory gaat bekijken, kan je zien dat het nu een bestand bevat voor die nieuwe inhoud. Dit is hoe Git de inhoud initieel opslaat—als een enkel bestand per stuk inhoud, met het SHA-1 controlegetal als naam die is berekend over de inhoud en de header. De subdirectory heeft de eerste 2 karakters van de SHA-1 als naam, en de bestandsnaam is de overige 38 karakters.

Als je inhoud in je object database hebt, kan je de inhoud bekijken met het `git cat-file` commando. Dit commando is een soort Zwitsers zakmes voor het inspecteren van Git objecten. Met het doorgeven van `-p` vertelt je het `cat-file` commando om het type inhoud uit te zoeken en het netjes aan je te laten zien:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4  
test content
```

Nu ben je in staat om inhoud aan Git toe te voegen en om het er weer uit te halen. Je kunt dit ook met de inhoud van bestanden doen. Bijvoorbeeld, je kunt een eenvoudig versiebeheer op een bestand doen. Eerst, maak een nieuw bestand aan en bewaar de inhoud in je database:

```
$ echo 'version 1' > test.txt  
$ git hash-object -w test.txt  
83baae61804e65cc73a7201a7252750c76066a30
```

Daarna schrijf je wat nieuwe inhoud naar het bestand, en bewaart het opnieuw:

```
$ echo 'version 2' > test.txt  
$ git hash-object -w test.txt  
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Je object database bevat beide versies van dit nieuwe bestand (zowel als de eerste inhoud die je daar bewaard hebt):

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Op dit moment kan je je lokale kopie van dat `test.txt` bestand verwijderen, en dan Git gebruiken om uit de object database de eerste versie die je bewaard hebt:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

of de tweede versie op te halen:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

Maar de SHA-1 sleutel onthouden voor elke versie van je bestand is niet praktisch; plus je bewaart niet de bestandsnaam in je systeem—alleen de inhoud. Dit type object noemen we een *blob*. Je kunt Git je het objecttype laten vertellen van elk object in Git, gegeven de SHA-1 sleutel, met `git cat-file -t`:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

## Boom objecten (tree objects)

Het volgende type waar we naar gaan kijken is de tree, wat het probleem oplost van het opslaan van bestandsnamen en je ook in staat stelt om een groep van bestanden bij elkaar op te slaan. Git slaat de inhoud op een manier die vergelijkbaar is met een UNIX bestandssysteem, maar wat versimpeld. Alle inhoud wordt opgeslagen als tree en blob objecten, waarbij trees overeenkomen met UNIX directory entries en blobs min of meer overeenkomen met inodes of bestandsinhoud. Een enkele tree object bevat een of meer entries, elk daarvan is de SHA-1 hash van een blob of subtree met de bijbehorende mode, type en bestandsnaam. Bijvoorbeeld, de meest recente tree in een project kan er ongeveer zo uitzien:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

De `master^{tree}` syntax geeft het tree object aan waarnaar wordt verwezen door de laatste commit op je `master`-branch. Merk op dat de `lib` subdirectory geen blob is, maar een verwijzing naar een andere tree:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0  
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b simplegit.rb
```

Afhankelijk van de shell die je gebruikt, kan je fouten krijgen als je de `master^{tree}` syntax gebruikt.



In CMD op Windows is het `^` karakter gebruikt voor escapen, dus je moet het verdubbelen om dit te voorkomen: `git cat-file -p master^{tree}`. Als je PowerShell gebruikt, moeten parameters waarin de {} karakters worden gebruikt van quotes worden voorzien om te voorkomen dat ze verkeerd worden geïnterpreteerd: `git cat-file -p 'master^{tree}'`.

Als je ZSH gebruikt, wordt het `^` karakter gebruikt voor globbing, dus je moet de hele expressie in quotes zetten: `git cat-file -p "master^{tree}"`.

Conceptueel zijn de gegevens die Git opslaat ongeveer dit:

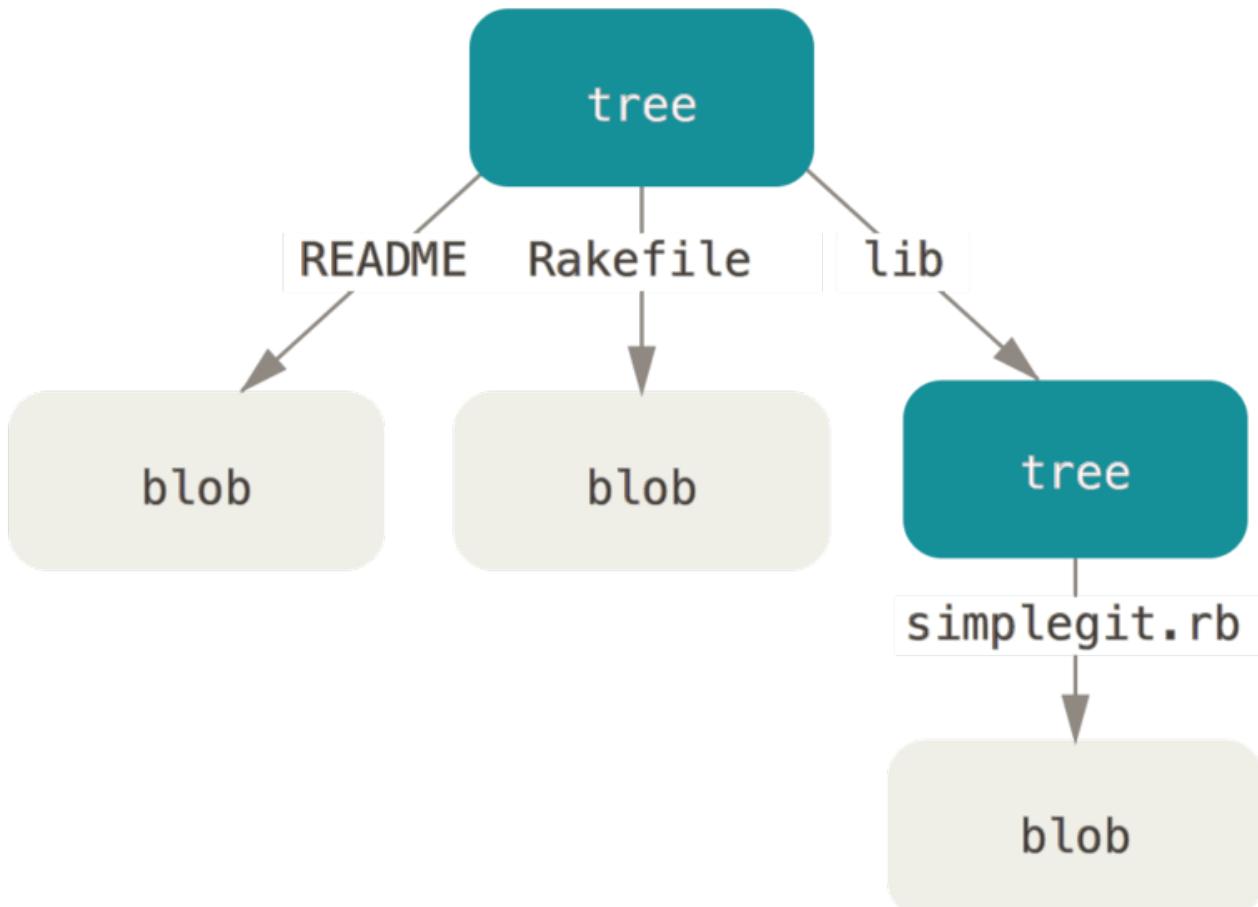


Figure 149. Eenvoudige versie van het Git datamodel.

Je kunt redelijk eenvoudig je eigen boom maken. Git maakt normaalgesproken een tree door de

staat van je staging gebied of index te nemen en daarvan een reeks tree objects te maken. Dus, om een tree object te maken, moet je eerst een index opzetten door wat bestanden te staggen. Om een index te maken met een enkele ingang—de eerste versie van je `test.txt` bestand—kan je het plumbing commando `git update-index` gebruiken. Je gebruikt dit commando om kunstmatig de eerdere versie van het bestand `test.txt` aan een nieuwe staging gebied toe te voegen. Je moet het de optie `--add` doorgeven omdat het bestand nog niet bestaat in je staging gebied (je hebt nog niet eens een staging gebied ingericht) en `--cacheinfo` omdat het bestand dat je toevoegt in in je directory zit maar in je database. Daarna geef je de mode, SHA-1 en bestandsnaam op:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

In dit geval geef je een mode `100644` op, wat aangeeft dat het een normaal bestand is. Andere opties zijn `100755`, wat aangeeft dat het een uitvoerbaar bestand is; en `120000`, wat een symbolische link aangeeft. De modus is afgeleid van normale UNIX modi maar het is minder flexibel—deze drie modi zijn de enige die geldig zijn voor bestanden (blobs) in Git (alhoewel andere modi worden gebruikt voor directories en submodules).

Nu kan je het `git write-tree` commando gebruiken om het staging gebied te schrijven naar een tree object. Hier is geen `-w` optie nodig—het aanroepen van `write-tree` maakt automatisch een tree object aan van de staat van de index als die tree nog niet bestaat:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Je kunt ook verifiëren dat dit een tree object is met hetzelfde `git cat-file` commando die je eerder gezien hebt:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Je gaat nu een nieuwe tree maken met de tweede versie van `test.txt` en ook nog een nieuw bestand:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Je staging gebied heeft nu de nieuwe versie van `test.txt` alsook het nieuwe bestand `new.txt`. Schrijf dat deze tree weg (sla de staat van het staging gebied of index op in een tree object) en kijk hoe dit eruit ziet:

```
$ git write-tree  
0155eb4229851634a0f03eb265b69f5a2d56f341  
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341  
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt  
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Merk op dat deze tree beide bestands entries bevat en ook dat de `test.txt` SHA-1 gelijk aan de “versie 2” SHA-1 van eerder is ([1f7a7a](#)). Puur voor de lol, ga je de eerste tree als een subdirectory toevoegen in deze. Je kunt trees in je staging area lezen door `git read-tree` aan te roepen. In dit geval, kan je een bestaande tree als een subtree in je staging gebied lezen door de `--prefix` optie te gebruiken bij dit commando:

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
$ git write-tree  
3c4e9cd789d88d8d89c1073707c3585e41b0e614  
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614  
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak  
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt  
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

Als je een werk directory van de nieuwe tree maakt die je zojuist geschreven hebt, zou je de twee bestanden op het hoogste niveau van de werk directory krijgen en een subdirectory met de naam `bak` die de eerste versie van het `test.txt` bestand zou bevatten. Je kunt de gegevens die Git bevat voor deze structuren als volgt weergeven:

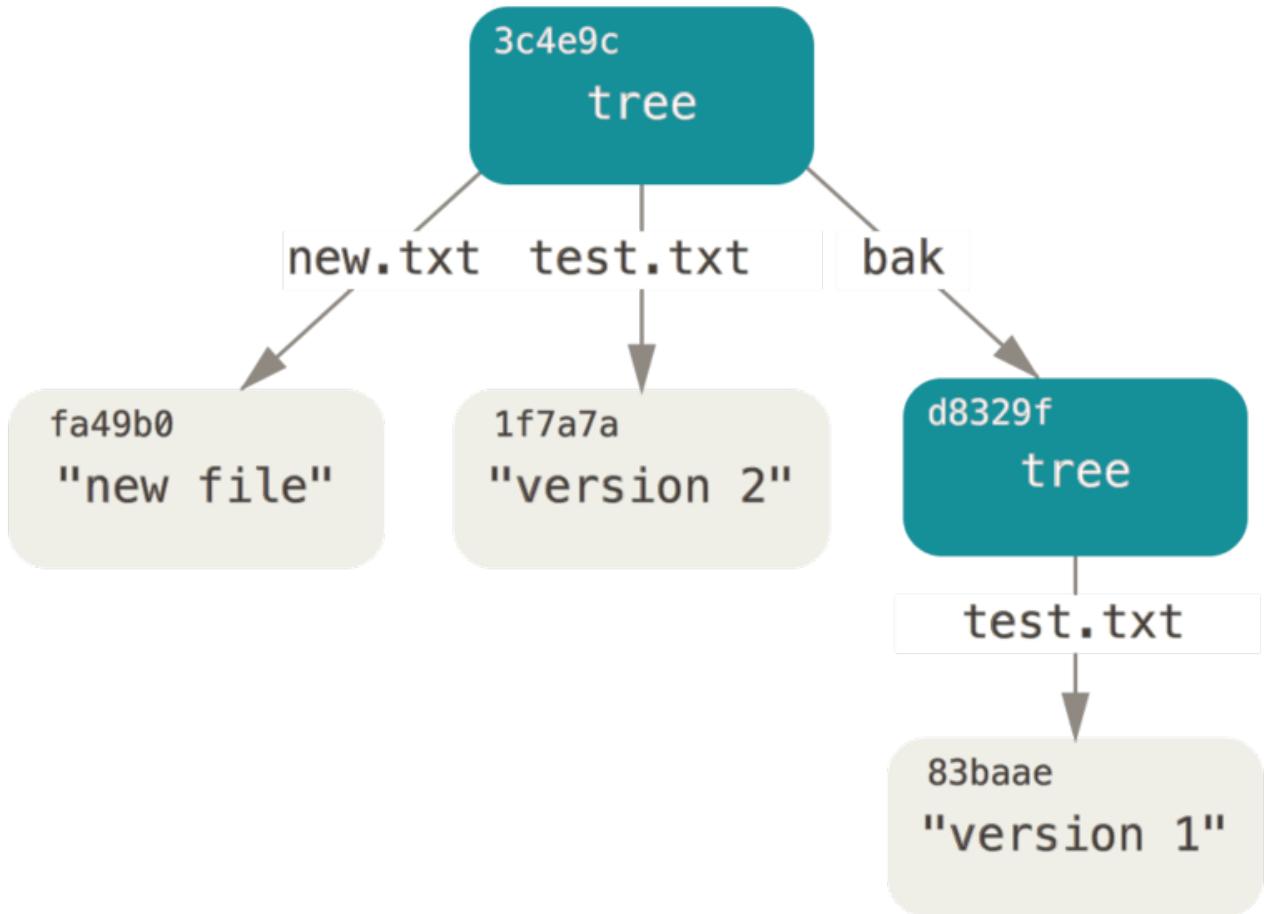


Figure 150. De inhoudsstructuur van je huidige Git gegevens.

## Commit objecten

Je hebt drie trees die de verschillende snapshots weergeven van je project die je wilt volgen, maar het eerdere probleem blijft: je moet alle drie SHA-1 waarden onthouden om de snapshots te kunnen terughalen. Je hebt ook geen informatie over wie de snapshots heeft opgeslagen, wanneer ze zijn opgeslagen of waarom ze waren opgeslagen. Dit is de basis informatie die het commit object voor je opslaat.

Om een commit object te maken, roep je `commit-tree` aan en geef je een de SHA-1 van een enkele tree op en welke commit objecten, indien van toepassing, er direct aan vooraf gaan. Begin met de eerste tree die je geschreven hebt:

```
$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```

Je zult verschillende hash-waarden krijgen omdat er verschillen zijn in aanmaak tijd en auteur-gegevens. Vervang de commit en tag-hashwaarden verderop in dit hoofdstuk met je eigen checksums. Nu kan je naar je nieuwe commit object kijken met `git cat-file`:

```
$ git cat-file -p fdf4fc3
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Scott Chacon <schacon@gmail.com> 1243040974 -0700
committer Scott Chacon <schacon@gmail.com> 1243040974 -0700

first commit
```

Het formaat voor een commit object is eenvoudig: het geeft de tree op het hoogste niveau weer voor de snapshot van het project op dat punt; de auteur/committer informatie (welke je `user.name` en `user.email` configuratie instellingen gebruikt en een timestamp); een blancko regel en dan het commit bericht.

Vervolgens ga je de andere twee commit objects schrijven, die elk naar de commit refereren die er direct aan vooraf ging:

```
$ echo 'second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cf9
```

Elk van de drie commit objecten verwijzen naar een van de drie snapshot trees die je gemaakt hebt. Gek genoeg, heb je nu een echte Git historie die je kunt bekijken met het `git log` commando, als je dit aanroeft op de laatste SHA-1 commit:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    third commit

bак/test.txt | 1 +
1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

new.txt  | 1 +
test.txt | 2 ++
2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

test.txt | 1 +
1 file changed, 1 insertion(+)
```

Verbluffend. Je hebt zojuist de diepere niveau operaties uitgevoerd die een Git historie hebben opgebouwd, zonder gebruik te maken van een van de hogere commando's. In essentie is dit wat Git doet als je de `git add` en `git commit` commando's gebruikt — het slaat blobs op voor de bestanden die zijn gewijzigd, werkt de index bij, schrijft de trees weg en schrijft commit objecten weg die verwijzen naar de trees op het hoogste niveau en de commits die er direct aan vooraf zijn gegaan. Deze drie hoofd Git objecten — de blob, de tree en de commit worden initieel opgeslagen als aparte bestanden in je `.git/objects` directory. Hier zijn alle huidige objecten in de voorbeeld directory, met als commentaar wat ze opslaan:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cf9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c7606a30 # test.txt v1
.git/objects/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
```

Als je al de interne verwijzingen volgt, zal je een objectgraaf krijgen die er ongeveer zo uitziet:

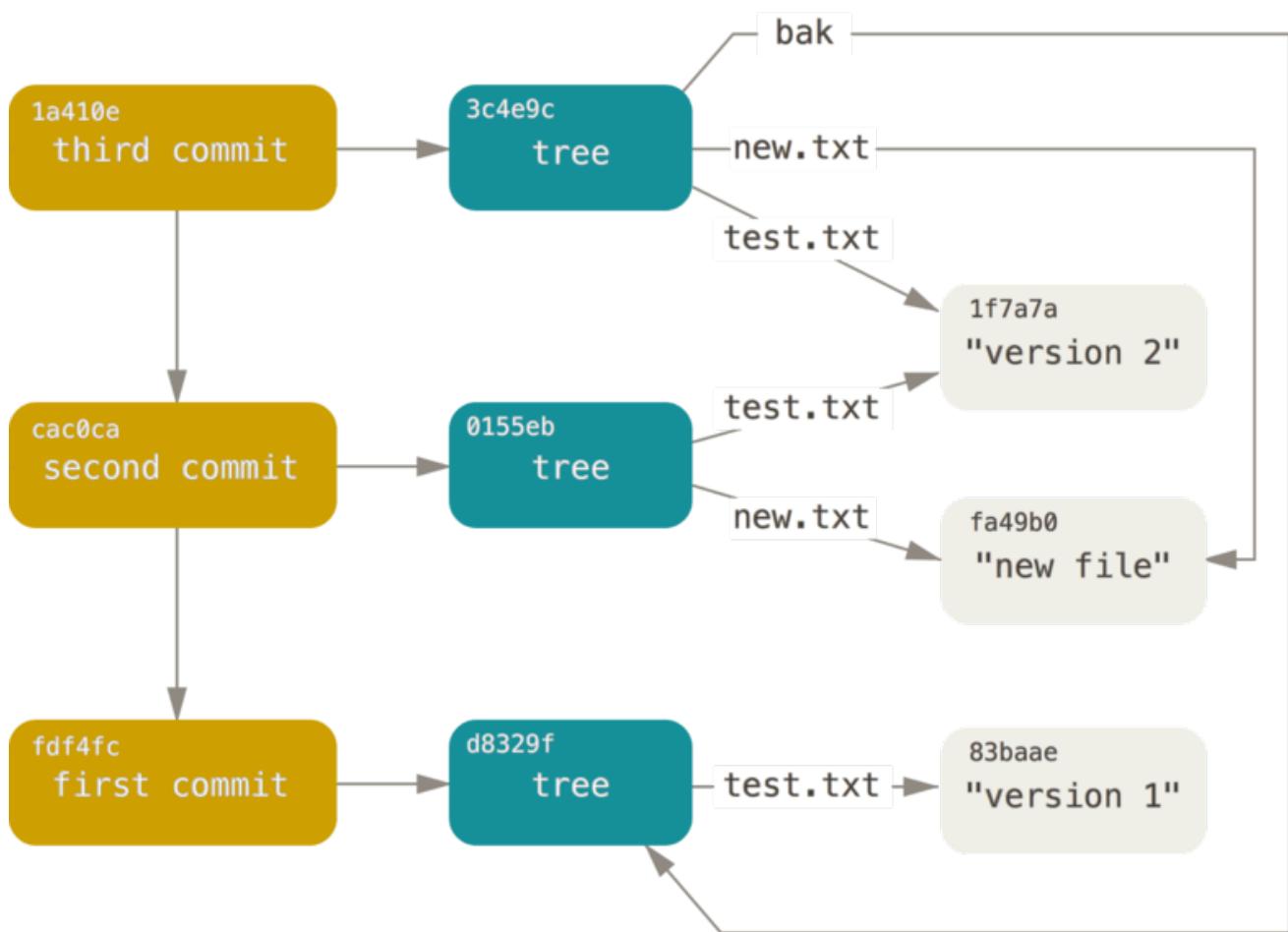


Figure 151. Alle bereikbare objects in je Git directory.

## Object opslag

We hebben eerder gezegd dat er een header wordt opgeslagen bij de inhoud. Laten we nu de tijd nemen om een kijkje te nemen hoe Git haar objecten opslaat. Je gaat zien hoe een blob object interactief wordt opgeslagen — in dit geval, de zin “what is up, doc?” — in de Ruby scripttaal.

Je kunt de interactieve Ruby modus starten met het `irb` commando:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git stelt een header samen die begint met het type van het object, in dit geval een blob. Aan dat eerste gedeelte van de header voegt Git een spatie toe gevolgd door de lengte in bytes van de inhoud en tot slot een null byte:

```
>> header = "blob #{content.length}\0"
=> "blob 16\u0000"
```

Git plakt de header en de oorspronkelijke inhoud samen en berekent het SHA-1 controlegetal van die nieuwe inhoud. Je kunt de SHA-1 waarde van een tekenreeks in Ruby berekenen door de `SHA1` digest library te includen met het `require` commando en daarna `Digest::SHA1hexdigest()` aan te roepen met de tekenreeks:

```
>> store = header + content
=> "blob 16\u0000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Laten we dit vergelijken met de uitvoer van `git hash-object`. Hier gebruiken we `echo -n` om te voorkomen dat er een newline aan de invoer wordt toegevoegd.

```
$ echo -n "what is up, doc?" | git hash-object --stdin
bd9dbf5aae1a3862dd1526723246b20206e5fc37
```

Git comprimeert de nieuwe inhoud met zlib, wat je in Ruby kunt doen met de `zlib` library. Eerst moet je de library `requiren` en daarna `Zlib::Deflate.deflate()` aanroepen op de inhoud:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "x\x9C\xCA\xC9\x04c(\xCFH,Q\xC8,V(-\xD0\xQH\xC9\x0\xB6\x0\x\x1C\x9D"
```

Als laatste schrijf je je zlib-deflated inhoud naar een object op schijf. Je bepaalt het pad van het object die je wilt schrijven (de eerste twee karakters van de SHA-1 waarde als de naam van de subdirectory, en de overige 38 karakters zijnde de bestandsnaam binnen die directory). In Ruby kan je de `FileUtils.mkdir_p()` functie gebruiken om de subdirectory aan te maken als die nog niet bestaat. Daarna open je het bestand met `File.open()` en schrijf je de eerder zlib-gecomprimeerde inhoud naar het bestand met een `write()` aanroep op de filehandle die je krijgt:

```
>> path = '.git/objects/' + sha1[0,2] + '/' + sha1[2,38]
=> ".git/objects/bd/9dbf5aae1a3862dd1526723246b20206e5fc37"
>> require 'fileutils'
=> true
>> FileUtils.mkdir_p(File.dirname(path))
=> ".git/objects/bd"
>> File.open(path, 'w') { |f| f.write zlib_content }
=> 32
```

Laten we de inhoud van het object controleren met `git cat-file`:

```
---
$ git cat-file -p bd9dbf5aae1a3862dd1526723246b20206e5fc37
what is up, doc?
---
```

Dat is alles - je hebt een valide Git blob object gemaakt. Alle Git objecten worden op dezelfde manier opgeslagen, alleen met andere types - in plaats van de tekenreeks blob, begint de header met commit of tree. Daarnaast, alhoewel de inhoud van een blob zo ongeveer alles kan zijn, is de inhoud van een commit en tree zeer specifiek geformatteerd.

## Git Referenties

Als je geïnteresseerd bent in zien van de geschiedenis van je repository vanaf commit, zeg maar, `1a410e`, zou je iets als `git log 1a410e` kunnen aanroepen om die geschiedenis te bekijken, maar je moet nog steeds onthouden dat `1a410e` de commit is die je als beginpunt wilt gebruiken voor die geschiedenis. Het in plaats daarvan handiger zijn als je een bestand zou hebben waarin je die SHA-1 waarde kunt opslaan onder een eenvoudiger naam zodat je die eenvoudiger naam zou kunnen gebruiken in plaats van die kale SHA-1 waarde.

In Git worden deze simpele namen “referenties” of “refs” genoemd, en je kunt de bestanden die deze SHA-1 waarden bevatten vinden in de `.git/refs` directory. In het huidige project, bevat deze directory geen bestanden, maar wat er wel in zit is een simpele structuur:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

Om een nieuwe referentie te maken die je gaat helpen onthouden waar je laatste commit is, kan je technisch gezien iets simpels als dit doen:

```
$ echo 1a410efbd13591db07496601ebc7a059dd55cfec > .git/refs/heads/master
```

Nu kan je de head-referentie die je zojuist gemaakt hebt in je Git commando's gebruiken in plaats van de SHA-1 waarde:

```
$ git log --pretty=oneline master  
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit  
cac0cab538b970a37ea1e769cbbde608743bc96d second commit  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Het wordt je niet aangeraden om de referentiebestanden direct te bewerken. Git voorziet in een veiliger commando genaamd **update-ref** als je een referentie wilt bijwerken:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Dit is wat een branch in Git eigenlijk is: een eenvoudige verwijzing of referentie naar de head van een bepaalde werkomgeving. Om achteraf een branch te maken naar de tweede commit, kan je dit doen:

```
$ git update-ref refs/heads/test cac0ca
```

Je branch zal alleen werk bevatten van die commit en daarvoor:

```
$ git log --pretty=oneline test  
cac0cab538b970a37ea1e769cbbde608743bc96d second commit  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Nu ziet je Git database er conceptueel zo ongeveer uit:

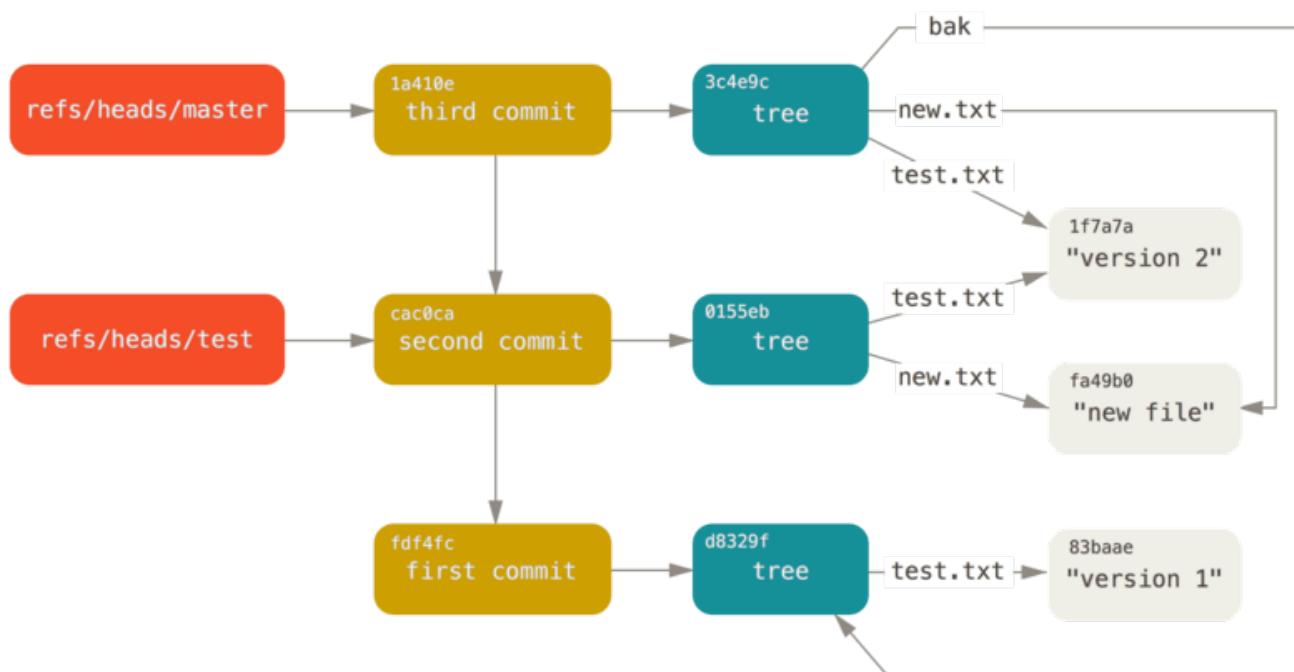


Figure 152. Git directory objecten inclusief branch head referenties.

Als je commando's aanroeft zoals `git branch <branchnaam>`, roept Git feitelijk die `update-ref` commando aan om de SHA-1 van de laatste commit op de branch waar je op zit te plaatsen in de referentie die je op dat moment wilt aanmaken.

## De HEAD

De vraag is nu, als je `git branch <branchnaam>` aanroeft, hoe weet Git de SHA-1 van de laatste commit? Het antwoord is het HEAD bestand.

Het HEAD bestand is een symbolische referentie naar de branch waar je op dit moment op zit. Met symbolische referentie bedoelen we dat, in tegenstelling tot een normale referentie, het over het algemeen geen SHA-1 waarde bevat, maar een verwijzing naar een andere referentie. Als je naar het bestand kijkt, zie je normaalgesproken zo iets als dit:

```
$ cat .git/HEAD  
ref: refs/heads/master
```

Als je `git checkout test` aanroeft, werkt Git het bestand bij om er zo uit te zien:

```
$ cat .git/HEAD  
ref: refs/heads/test
```

Als je `git commit` aanroeft, wordt het commit object aangemaakt, waarbij de ouder van dat commit object wordt gespecificeerd door de SHA-1 waarde die staat in de referentie waar HEAD op dat moment naar verwijst.

Je kunt dit bestand ook handmatig bijwerken, maar alweer is er een veiliger comando om dit te doen: `symbolic-ref`. Je kunt de waarde van je HEAD met dit comando lezen:

```
$ git symbolic-ref HEAD  
refs/heads/master
```

Je kunt ook de waarde van HEAD bepalen met hetzelfde commando:

```
$ git symbolic-ref HEAD refs/heads/test  
$ cat .git/HEAD  
ref: refs/heads/test
```

Je kunt geen symbolische referentie waarde invullen die niet voldoet aan de refs-stijl:

```
$ git symbolic-ref HEAD test  
fatal: Refusing to point HEAD outside of refs/
```

## Tags

We zijn zojuist geëindigt met het bespreken van de drie hoofd objecttypen van Git (*blobs*, *trees* en *commits*), maar er is nog een vierde. Het *tag* object lijkt erg op een commit object—het bevat een tagger, een datum, een bericht en een verwijzing. Het belangrijkste verschil is dat een tag object over het algemeen verwijst naar een commit in plaats van een boom. Het lijkt op een branch referentie, maar het zal nooit bewegen—het verwijst altijd naar dezelfde commit, maar geeft het een vriendelijker naam.

Zoals besproken in [Git Basics](#), zijn er twee soorten tags: geannoteerd en lichtgewicht. Je kunt een lichtgewicht tag aanmaken door iets als het volgende aan te roepen:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769ccbde608743bc96d
```

Dat is alles wat een lichtgewicht tag is—een referentie dat nooit zal bewegen. Een geannoteerde tag is echter veel complexer. Als je een geannoteerde tag aanmaakt, maakt Git een tag object en schrijft daarna een referentie die daarnaar verwijst, in plaats van direct te verwijzen naar de commit. Je kunt dit zien door een geannoteerde tag aan te maken (met de **-a** optie):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Hier is de SHA-1 waarde van het object die is aangemaakt:

```
$ cat .git/refs/tags/v1.1  
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

En roep nu het `git cat-file -p` commando aan op die SHA-1 waarde;

```
$ git cat-file -p 9585191f37f7b0fb9444f35a9bf50de191beadc2  
object 1a410efbd13591db07496601ebc7a059dd55cfe9  
type commit  
tag v1.1  
tagger Scott Chacon <schacon@gmail.com> Sat May 23 16:48:58 2009 -0700  
  
test tag
```

Merk op dat vermelding van het object wijst naar de SHA-1 waarde van de commit die je hebt getagged. Merk ook op dat het niet perse hoeft te verwijzen naar een commit; je kunt elke Git object taggen. In de broncode van Git, bijvoorbeeld, heeft de onderhouder hun GPG publieke sleutel als een blob object toegevoegd en deze toen getagged. Je kunt de publieke sleutel bekijken door het volgende aan te roepen in een cloon van de Git repository:

```
$ git cat-file blob junio-gpg-pub
```

De Linux kernel repository heeft ook een tag die niet naar een commit wijst—de eerste tag die gemaakt is verwijst naar de initiële tree van de import van de broncode.

## Remotes

Het derde type referentie die je zult zien is een remote referentie. Als je een remote toevoegt en ernaar pusht, slaat Git voor elke de waarde die je het laatst naar die remote hebt gepusht in de `refs/remotes` directory. Bijvoorbeeld, je kunt een remote genaamd `origin` toevoegen en daar je `master`-branch naar pushen:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
$ git push origin master
Counting objects: 11, done.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (7/7), 716 bytes, done.
Total 7 (delta 2), reused 4 (delta 1)
To git@github.com:schacon/simplegit-progit.git
  a11bef0..ca82a6d  master -> master
```

Je kunt zien wat de `master`-branch op de `origin` remote was op het moment dat je voor het laatst met de server hebt gecommuniceerd, door het `refs/remotes/origin/master` bestand te bekijken:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Referenties van remotes verschillen van branches (`refs/heads` referenties) voornamelijk door het feit dat ze als *alleen-lezen* worden beschouwd. Je kunt naar een `git checkout` doen, maar Git zal HEAD nooit naar een laten verwijzen, dus je zult er nooit een kunnen bijwerken met een `commit` commando. Git gebruikt ze als boekleggers naar de laatst bekende staat van waar deze branches op stonden op die servers.

## Packfiles

Als je alle instructies in het voorbeeld van de hiervoorstaande sectie hebt gevolgd, zou je nu een test Git repository moeten hebben met 11 objecten—vier blobs, drie trees, drie commits en een tag:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cf9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/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 comprimeert de inhoud van deze bestanden met zlib, en je slaat niet veel op, dus al deze bestanden nemen samen maar 925 bytes in beslag. Je zult wat lijviger inhoud aan de repository toevoegen om een interessante mogelijkheid van Git te demonstreren. Om het te demonstreren zullen we het `repo.rb` bestand van de Grit library toevoegen — dit is een broncode bestand van ongeveer 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 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Als je naar de resulterende tree kijkt, zie je de SHA-1 waarde die je `repo.rb` bestand heeft gekregen voor het nieuwe blob object:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Je kunt daarna `git cat-file` gebruiken om te zien hoe groot dat object is:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Wijzig nu dat bestand een beetje, en kijk wat er gebeurt:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo.rb a bit'
[master 2431da6] modified repo.rb a bit
 1 file changed, 1 insertion(+)
```

Kijk naar de tree die door die commit gemaakt is, en je zult iets interessants zien:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

De blob is nu een andere blob, wat inhoudt dat ondanks dat je maar een enkele regel aan het eind van een bestand van 400 regels hebt toegevoegd, Git de nieuwe inhoud als een volledig nieuw object heeft opgeslagen.

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
22054
```

Je hebt nu twee vrijwel identieke objecten van 22K op je schijf staan (elk gecomprimeerd tot ongeveer 7K). Zou het niet aardig zijn als Git een van deze volledig zou opslaan maar dan het tweede object alleen als de delta tussen deze en de eerste?

Laat dit nu ook het geval zijn. Het initiële formaat waar Git objecten op schijf bewaard wordt een “loose” (los, ruimzittend) object formaat genoemd. Echter, van tijd tot tijd pakt Git een aantal van deze bestanden in een enkele binair bestand “packfile” genaamd in met als doel ruimte te besparen en meer efficiënt te zijn. Git doet dit als je teveel losse bestanden hebt, als je het `git gc` commando handmatig aanroeft, of als je naar een remote server pusht. Om te zien wat er gebeurt, kan je Git handmatig vragen om de objecten in te pakken door het `git gc` commando aan te roepen:

```
$ git gc
Counting objects: 18, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), done.
Total 18 (delta 3), reused 0 (delta 0)
```

Als je in je objecten directory kijkt, zal je zien dat de meeste van je objecten zijn verdwenen en dat er een aantal andere bestanden zijn verschenen:

```
$ 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
```

De objecten die zijn overgebleven zijn de blobs waar geen enkele commit nog naar wijst — in dit geval de blobs van het “what is up, doc?”-voorbeeld en het “test content”-voorbeeld die je eerder hebt gemaakt. Omdat je ze nooit aan enige commit hebt toegevoegd, worden ze beschouwd als bungelend (dangling) en worden ze niet in je nieuwe packfile ingepakt.

De andere bestanden zijn je nieuwe packfile en een index. Het packfile bestand is een enkel bestand met daarin de inhoud van al de objecten die zijn verwijderd van je bestandssysteem. De index is een bestand die de relatieve beginpunten bevat in die packfile zodat je snel naar een specifiek object kunt zoeken. Wat cool is, is dat ondanks dat de bestanden op schijf voordat je het `gc` commando aanriep samen ongeveer 15K groot waren, het nieuwe packfile bestand maar 7K is. Je hebt schijfruimte-gebruik tot de helft verminderd door je objecten in te pakken.

Hoe doet Git dit? Als Git objecten inpakt, kijkt het naar bestanden die vergelijkbare namen en grootte hebben, en slaat daarvan alleen de delta's tussen de ene versie van het bestand en de andere op. Je kunt in het packfile bestand kijken en zien wat Git gedaan heeft om schijfruimte te besparen. Het `git verify-pack` binnenwerk commando stelt je in staat om te zien wat er ingepakt is:

```
$ git verify-pack -v .git/objects/pack/pack-
978e03944f5c581011e6998cd0e9e30000905586.idx
2431da676938450a4d72e260db3bf7b0f587bbc1 commit 223 155 12
69bcdaff5328278ab1c0812ce0e07fa7d26a96d7 commit 214 152 167
80d02664cb23ed55b226516648c7ad5d0a3deb90 commit 214 145 319
43168a18b7613d1281e5560855a83eb8fde3d687 commit 213 146 464
092917823486a802e94d727c820a9024e14a1fc2 commit 214 146 610
702470739ce72005e2edff522fde85d52a65df9b commit 165 118 756
d368d0ac0678cbe6cce505be58126d3526706e54 tag 130 122 874
fe879577cb8cffcdf25441725141e310dd7d239b tree 136 136 996
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree 36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree 136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree 6 17 1314 1 \
    deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 8 19 1331 1 \
    deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob 22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 9 20 7262 1 \
    b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/pack-978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Hier verwijst de **033b4** blob die, als je het je nog kunt herinneren, de eerste versie van je repo.rb bestand is, naar de **b042a** blob, wat de tweede versie van het bestand was. De derde kolom in de uitvoer is de grootte van het object in het packfile bestand, dus je kunt zien dat **b042a** 22K van het bestand inneemt, aar dat **033b4** maar 9 bytes in beslag neemt. Wat ook interessant is, is dat de tweede versie van het bestaand degene is die intact opgeslagen is, terwijl de orginele versie is opgeslagen als een delta — de reden hiervoor is dat het waarschijnlijker is dat je snellere toegang nodig hebt naar de meest recente versie van het bestand.

Het echte leuke van dit is, is dat het op elk gewenste moment weer opnieuw kan worden ingepakt. Git zal van op gezette tijden je database automatisch opnieuw inpakken, altijd met het doel nog meer ruimte te besparen, maar je kunt ook handmatig opnieuw laten inpakken door **git gc** zelf aan te roepen.

## De Refspec

In dit hele boek hebben we eenvoudige verbanden (mappings) gebruikt tussen remote branches naar lokale referenties, maar ze kunnen veel complexer zijn. Stel even dat je met de paar laatste secties hebt meegeedaan en een kleine lokale Git repository gemaakt hebt, en nu een *remote* eraan zou willen toevoegen:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Dit commando voegt een sectie toe aan je `.git/config` bestand, waarbij de naam van de remote (`origin`) wordt opgegeven, de URL van de remote repository en de refspec die gebruikt moet worden voor het fetchen:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Het formaat van de refspec is, eerst, een optionele `+`, gevuld door `<src>:<dst>`, waar '`<src>`' het patroon is voor referenties aan de remote kant, en `<dst>` de plaats is waar deze referenties lokaal zullen worden getracked. De `+` draagt Git op om de referenties bij te werken zelfs als het geen fast-forward is.

In het standaard geval dat automatisch wordt geschreven door een `git remote add` commando, zal Git alle referenties onder `refs/heads` op de server fetchen en ze lokaal naar `refs/remotes/origin` schrijven. Dus, als er een `master`-branch op de server is, kan je de log van deze branch op deze manieren benaderen:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Dit is allemaal gelijkwaardig, omdat Git elk van deze uitwerkt tot `refs/remotes/origin/master`.

Als je in plaats hiervan Git alleen de `master`-branch wilt laten pullen, en niet elke andere branch op de remote server, kan je de fetch regel wijzigen zodat het alleen naar die branch wijst:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Dit is gewoon de standaard refspec voor `git fetch` naar die remote. Als je een eenmalige fetch wilt doen, kan je de refspec ook op de command regel opgeven. Om de `master`-branch te pullen van de remote naar een lokale `origin/mymaster`, kan je dit aanroepen

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Je kunt ook meerdere refspecs opgeven. Op de commando regel kan je meerdere branches als volgt pullen:

```
$ git fetch origin master:refs/remotes/origin/mymaster \
  topic:refs/remotes/origin/topic
From git@github.com:schacon/simplegit
 ! [rejected]      master    -> origin/mymaster (non fast forward)
 * [new branch]    topic     -> origin/topic
```

In dit geval wordt de pull van de `master`-branch geweigerd omdat het niet als een fast-forward referentie was opgegeven. Je kunt dat overschrijven door een `+` voor de refspec op te geven.

Je kunt ook meerdere respecs voor fetchen aangeven in je configuratie bestand. Als je altijd de `master` en `experiment`-branches wilt fetchen, voeg je twee regels toe:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Je kunt niet gedeeltelijke globs in het patroon opgeven, dus dit zou ongeldig zijn:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Echter, je kunt naamsruimten (namespaces) of directories opgeven om zo iets voor elkaar te krijgen. Als je een QA team hebt die een reeks van branches pusht, en je wilt de `master` branch verkrijgen en alle branches van het QA team maar niets anders, kan je een configuratie instelling als deze gebruiken:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/master:refs/remotes/origin/master
  fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

Als je een complexe workflow proces hebt waarbij een QA-team branches pusht, waarbij ontwikkelaar branches pushen en integratie teams die pushen naar en samenwerken op remote branches, kan je ze op deze manier eenvoudig namespaces toewijzen.

## Refspecs pushen

Het is leuk dat je namespaced referenties op deze manier kunt fetchen, maar hoe krijgt het QA-team om te beginnen hun branches in een `qa/` namespace? Je krijgt dit voor elkaar door refsspecs te gebruiken om te pushen.

Als het QA-team hun `master`-branch naar `qa/master` op de remote server wil pushen, kunnen ze dit aanroepen:

```
$ git push origin master:refs/heads/qa/master
```

Als ze willen dat Git dit elke keer automatisch doet als ze `git push origin` aanroepen, kunnen ze een `push` waarde aan hun configuratie bestand toevoegen:

```
[remote "origin"]
url = https://github.com/schacon/simplegit-progit
fetch = +refs/heads/*:refs/remotes/origin/*
push = refs/heads/master:refs/heads/qa/master
```

Nogmaals, dit zal ervoor zorgen dat een `git push origin` de lokale `master`-branch standaard naar de remote `qa/master`-branch pusht.

## References verwijderen

Je kunt de refspec ook gebruiken om referenties te verwijderen van de remote server door iets als dit aan te roepen:

```
$ git push origin :topic
```

Omdat de refspec `<src>:<dst>` is zegt dit, door het weglaten van het `<src>` gedeelte, eigenlijk dat de topic branch van de remote niets moet worden, waarmee het wordt verwijderd.

Of je kunt de nieuwere syntax gebruiken (beschikbaar vanaf Git v1.7.0):

```
$ git push origin --delete topic
```

## Uitwisseling protocollen

Git kan op twee belangrijke manieren gegevens uitwisselen tussen twee repositories: het “domme” (dumb) protocol en het “slimme” (smart) protocol. In dit gedeelte zal de manier van werken van beide snel worden besproken.

### Het domme protocol

Als je een repository opzet die alleen gelezen hoeft te worden via HTTP, is het domme protocol het meest waarschijnlijke zijn die zal worden gebruikt. Dit protocol wordt “dom” genoemd omdat het geen Git-specifieke code nodig heeft aan de kant van de server tijdens het uitwisselingsproces; het fetch proces is een reeks van HTTP `GET` requests, waar het werkstation aannames mag doen over de inrichting van de Git repository op de server.



Het domme protocol wordt tegenwoordig nog maar sporadisch gebruikt. Het is moeilijk te beveiligen of af te schermen, dus de meeste Git hosts (zowel in de cloud als op locatie) zullen het weigeren te gebruiken. Het wordt over het algemeen aangeraden om het slimme protocol te gebruiken, die we iets verderop zullen bespreken.

Laten we het **http-fetch** proces voor de simplegit library eens volgen:

```
$ git clone http://server/simplegit-progit.git
```

Het eerste wat dit commando doet is het **info/refs** bestand pullen. Dit bestand wordt geschreven door het **update-server-info** commando, wat de reden is waarom je dit als **post-receive** hook moet activeren om het uitwisselen via HTTP goed te laten werken:

```
=> GET info/refs  
ca82a6dff817ec66f44342007202690a93763949      refs/heads/master
```

Nu heb je een lijst met de remote referenties en SHA-1 waarden. Vervolgens ga je op zoek naar de HEAD referentie zodat je weet wat je moet uitchecken als je klaar bent:

```
=> GET HEAD  
ref: refs/heads/master
```

Je moet de **master**-branch uitchecken als je het proces hebt voltooid. Op dit punt ben je gereed om het proces te doorlopen. Omdat je vertrekpunt het **ca82a6** commit object is die je in het **info/refs** bestand zag, begin je met die te fetchen:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949  
(179 bytes of binary data)
```

Je krijgt een object terug - dat object is in het losse (loose) formaat op de server, en je hebt het met een statische HTTP GET request gefetcht. Je kunt het nu met zlib-uncompress uitpakken, de header ervan afhalen, en naar de commit inhoud kijken:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949  
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf  
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
author Scott Chacon <schacon@gmail.com> 1205815931 -0700  
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700  
  
changed the version number
```

Vervolgens heb je nog twee objecten op te halen - **cfda3b**, wat de boom met inhoud is waar de commit die we zojuist opgehaald hebben naar wijst, en **085bb3**, wat de ouder-commit is:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
(179 bytes of data)
```

En daarmee krijg je je volgende commit object. Haal het boom-object op:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf  
(404 - Not Found)
```

Oeps, het ziet er naar uit dat het boom-object niet in het losse formaat op de server is, dus je krijgt een 404 antwoord. Hier zijn een aantal mogelijke oorzaken voor - het object kan in een andere repository zitten, of het zou in een packfile in deze repository kunnen zitten. Git controleert eerst of er alternatieven zijn opgegeven:

```
=> GET objects/info/http-alternates  
(empty file)
```

Dit geeft een lijst met alternatieve URLs terug, Git controleert daar op losse bestanden en packfiles - dit is een aardig mechanisme voor projecten die forks zijn van elkaar zijn om objecten te delen op schijf. Echter, omdat er in dit geval geen alternatieven worden gegeven, moet het object in een packfile zitten. Om te zien welke packfiles er beschikbaar zijn op deze server, moet je het **objects/info/packs** bestand te pakken krijgen, waar een opsomming hiervan in staat (wat ook door de **update-server-info** wordt gegenereerd):

```
=> GET objects/info/packs  
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Er is maar één packfile op de server, dus jouw object moet daar wel in zitten, maar je gaat het index bestand toch controleren om er zeker van te zijn. Dit is ook handig als je meerdere packfiles op de server hebt, omdat je dan kan zien welk packfile het object wat je nodig heeft bevat:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx  
(4k of binary data)
```

Nu je de packfile index hebt, kan je kijken of jouw object daar in zit - omdat de index de SHA-1 waarden van de objecten bevat die in de packfile zitten en de relatieve afstand (offset) naar deze objecten. Jouw object is er, dus ga je verder en haalt de hele packfile op:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack  
(13k of binary data)
```

Je hebt je boom object, dus je gaat verder met je commits af te lopen. Die zitten ook allemaal in de packfile die je zojuist hebt gedownload, dus je hoeft geen verzoeken meer te doen naar de server. Git checkt een werk-kopie van de **master**-branch uit waarnaar werd verwiesen door de HEAD

referentie die je aan het begin hebt gedownload.

## Het slimme protocol

Het domme protocol is eenvoudig, maar een beetje inefficiënt, het kan geen gegevens aan die van het werkstation naar de server moet worden geschreven. Het slimme protocol is een meer gebruikelijke methode van gegevens uit te wisselen, maar het heeft een proces aan de remote kant nodig die op de hoogte is van Git - het moet lokale gegevens kunnen lezen, uitvinden wat het werkstation al heeft en nog nodig heeft, en een op maat gemaakte packfile hiervoor maken. Er zijn twee groepen van processen voor het uitwisselen van gegevens: een paar voor het uploaden van gegevens en een paar voor het downloaden van gegevens.

### Het uploaden van gegevens

Om gegevens te uploaden naar een remote proces, gebruikt Git de `send-pack` en `receive-pack` processen. Het `send-pack` proces loopt op het werkstation en maakt verbinding met een `receive-pack` proces aan de remote kant.

#### SSH

Bijvoorbeeld, stel dat je `git push origin master` in jouw project aanroeft en `origin` is als een URL gedefinieerd die het SSH protocol gebruikt. Git start het `send-pack` proces op, die over SSH een verbinding initieert naar je server. Het probeert een commando op de remote server aan te roepen via een SSH call die er ongeveer zo uitziet:

```
$ ssh -x git@server "git-receive-pack 'simplegit-progit.git'"  
00a5ca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status \  
    delete-refs side-band-64k quiet ofs-delta \  
    agent=git/2:2.1.1+github-607-gfba4028 delete-refs  
0000
```

Het `git-receive-pack` commando antwoordt meteen met een regel voor elke referentie die het op dit moment heeft - in dit geval alleen de `master`-branch en zijn SHA-1. De eerste regel heeft ook een lijst met de mogelijkheden van de server (hier, `report-status`, `delete-refs`, en een aantal andere, inclusief de identificatie van de client).

Elke regel begint met een hex waarde van 4 tekens die aangeeft hoe lang de rest van de regel is. Je eerste regel begint met 00a5, wat hexadecimaal is voor 165, wat weer inhoudt dat er nog 165 bytes tot die regel behoren. De volgende regel is 0000, wat betekent dat de server klaar is met het uitlijsten van de referenties.

Nu dat het de status van de server kent, bepaalt je `send-pack` proces welke commits het heeft die de server niet heeft. Voor elke referentie die deze push gaat bijwerken, vertelt het `send-pack` proces het `receive-pack` proces deze informatie. Bijvoorbeeld, als je de `master`-branch aan het bijwerken bent en een `experiment`-branch toevoegt, zal het antwoord van `send-pack` er ongeveer zo uitzien:

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6
 \
   refs/heads/master report-status
006c000000000000000000000000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf63e8d
 \
   refs/heads/experiment
0000
```

Git stuurt een regel voor elke referentie die je bijwerkt met de lengte van de regel, de oude SHA-1, de nieuwe SHA-1 en de referentie die wordt geüpdate. De eerste regel bevat ook de mogelijkheden van het werkstation. De SHA-1 waarde van allemaal '0'en houdt in dat er niets daarvoor was - omdat je de referentie van het experiment aan het toevoegen bent. Als je een referentie aan het verwijderen zou zijn, zou je het omgekeerde zien: alle '0'en aan de rechterkant.

Daarna stuurt het werkstation een packfile van alle objecten die de server nog niet heeft. Als laatste antwoordt de server met een indicatie van succes (of falen):

000eunpack ok

## HTTP(S)

Dit proces is voor het grootste gedeelte hetzelfde als over HTTP, al is de *handshaking* iets anders. De verbinding wordt begonnen met dit verzoek:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack  
001f# service=git-receive-pack  
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master report-status \  
    delete-refs side-band-64k quiet ofs-delta \  
    agent=git/2:2.1.1~vgmgb-bitmaps-bugaloo-608-g116744e  
0000
```

Dat is het einde van de eerste werkstation-server uitwisseling. Het werkstation stuurt daarna een ander verzoek, dit keer een **POST**, met de gegevens die worden geleverd door **send-pack**.

```
=> POST http://server/simplegit-progit.git/qit-receive-pack
```

Het **POST** verzoek bevat de uitvoer van **send-pack** en de packfile als zijn bagage (payload). De server geeft daarna aan of het succesvol of niet heeft verwerkt in zijn HTTP antwoord.

## Gegevens downloaden

Als je gegevens download, zijn de `fetch-pack` en 'upload-pack` processen erbij betrokken. Het werkstation begint een `fetch-pack` proces die verbinding maakt met een `upload-pack` proces op de remote kant om te onderhandelen welke gegevens er naar het werkstation zullen worden gestuurd.

## SSH

Als je de fetch via SSH doet, zal **fetch-pack** ongeveer dit aanroepen:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

Nadat **fetch-pack** verbinding heeft gemaakt, stuurt **upload-pack** zo iets als dit terug:

```
00dfca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag \
  multi_ack_detailed symref=HEAD:refs/heads/master \
  agent=git/2:2.1.1+github-607-gfba4028
003fe2409a098dc3e53539a9028a94b6224db9d6a6b6 refs/heads/master
0000
```

Dit lijkt op waar **receive-pack** mee antwoordt, maar de mogelijkheden zijn anders. Daarenboven stuurt het terug waar HEAD op dit moment naar wijst (**symref=HEAD:refs/heads/master**), zodat het werkstation weet wat het uit moet checken als dit een kloon is.

Op dit punt kijkt het **fetch-pack**-proces naar welke objecten het heeft en antwoordt met de objecten dat het nodig heeft door “want” te sturen en dan de SHA-1 die het wil hebben. Het stuurt alle objecten die het al heeft met “have” en daarna de SHA-1. En aan het einde van deze lijst, schrijft het “done” om het **upload-pack** proces aan te zetten tot het beginnen met sturen van de packfile met de gegevens die het nodig heeft:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

## HTTP(S)

De *handshake* voor een fetch operatie vergt twee HTTP verzoeken. Het eerste is een **GET** naar hetzelfde adres als gebruikt in het domme protocol:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD□multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag \
  multi_ack_detailed no-done symref=HEAD:refs/heads/master \
  agent=git/2:2.1.1+github-607-gfba4028
003fcfa82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Dit lijkt erg op het aanroepen van **git-upload-pack** over een SSH verbinding, maar de tweede uitwisseling wordt uitgevoerd als een separaat verzoek:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeaddad63ad106860237bbf53411d11a7
0032have 441b40d833fdfa93eb2908e52742248faf0ee993
0000
```

Wederom, dit volgt hetzelfde patroon als hierboven. Het antwoord op dit verzoek geeft succes of falen aan, en bevat de packfile.

## Protocollen samenvatting

Deze paragraaf bevat een zeer basale overzicht van de uitwisselings-protocollen. Het protocol omvat vele andere mogelijkheden, zoals `multi_ack` of `side-band` mogelijkheden, maar de behandeling hiervan valt buiten het bestek van dit boek. We hebben geprobeerd je een idee te geven van het globale over-en-weer tussen werkstation en server; als je hierover meer wilt weten, dan kan je bijvoorbeeld een kijkje nemen in de Git broncode.

# Onderhoud en gegevensherstel

Bij tijd en wijle, moet je wat opruimen - een repository iets compacter maken, een geïmporteerde repository opruimen of verloren gegane werk herstellen. Deze paragraaf zal een aantal van deze scenarios behandelen.

## Onderhoud

Op gezette tijden roept Git automatisch een commando genaamd “auto gc” aan. Meestal zal dit commando niets doen. Echter, als er teveel losse objecten zijn (objecten die niet in een packfile zitten) of teveel packfiles, roept Git een volwaardige `git gc` commando aan. Het “gc” staat voor vuil ophalen (garbage collect), en het commando voert een aantal dingen uit: het verzamelt alle losse objecten en zet ze in packfiles, het consolideert packfiles in een grote packfile, en het verwijdert objecten die onbereikbaar zijn vanaf enig commit en een aantal maanden oud zijn.

Je kunt `auto gc` ook handmatig aanroepen als volgt:

```
$ git gc --auto
```

Nogmaals, over het algemeen doet dit niets. Je moet ongeveer 7.000 losse objecten hebben of meer dan 50 packfiles om Git een echte `gc` commando te laten aanroepen. Je kunt deze grenswaarden aanpassen met respectievelijk het `gc.auto` en `gc.autopacklimit` configuratie waarden.

Het andere wat `gc` zal doen is je referenties in een enkel bestand stoppen. Stel dat je repository de volgende branches en tags bevat:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Als je `git gc` aanroeft, zal je deze bestanden niet langer in de `refs` directory hebben staan. Git verplaatst ze allemaal in het kader van efficiëntie naar een bestand met de naam `.git/packed-refs` die er als volgt uit ziet:

```
$ 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
```

Als je een referentie bijwerkt, werkt Git dit bestand niet bij, maar schrijft in plaats daarvan een nieuw bestand naar `refs/heads`. Om de juiste SHA-1 voor een gegeven refentie te pakken te krijgen, zoekt Git de referentie eerst in de `refs` directory en daarna het `packed-refs` bestand als achtervang. Dus, als je een referentie niet kunt vinden in de `refs` directory, staat deze waarschijnlijk in je `packed-refs` bestand.

Let even op de laatste regel van het bestand, die begint met een `^`. Dit geeft aan dat de tag directory erboven een geannoteerde tag is, en deze regel de commit is waar de geannoteerde tag naar verwijst.

## Gegevensherstel

Op een bepaald punt in je Git-reis, kan je per ongeluk een commit kwijt raken. Meestal gebeurt dit omdat je een branch *force-delete* waar werk op zat, en je komt erachter dat je die branch toch nog nodig had; of je hebt een branch *ge-hard-reset*, en daarmee commits laat vallen waar je toch nog iets van wilde gebruiken. Aangenomen dat dit gebeurd is, hoe kan je die commits nog terughalen?

Hier is een voorbeeld die de master branch hard-reset in je test repository naar een oudere commit en daarna de verloren commits herstelt. Laten we eerst eens zien hoe je repository er op dit moment uitziet:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Nu gaan we de `master`-branch terugzetten naar de middelste commit:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cf9  
HEAD is now at 1a410ef third commit  
$ git log --pretty=oneline  
1a410efbd13591db07496601ebc7a059dd55cf9 third commit  
cac0cab538b970a37ea1e769cbbde608743bc96d second commit  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Je bent effectief de bovenste twee commits kwijtgeraakt - je hebt geen branch waar deze commits vanaf bereikt kunnen worden. Je moet de SHA-1 van de laatste commit zien te vinden en dan een branch toevoegen die daarnaar wijst. De truuk is het vinden van de SHA-1 van die laatste commit - we mogen aannemen dat je deze niet uit je hoofd hebt geleerd, toch?

Vaak is de snelste manier om een instrument genaamd `git reflog` te gebruiken. Als je aan het werk bent, houdt Git stilletjes bij wat je HEAD was elke keer als je het verandert. Elke keer als je commit, of branches wijzigt, wordt de reflog bijgewerkt. De reflog wordt ook bijgewerkt door het `git update-ref` commando, wat nog een reden is om dit te gebruiken in plaats van alleen de SHA-1 waarden naar je ref bestanden te schrijven, zoals we besproken hebben in [Git Referenties](#). Je kunt zien waar je op enig moment geweest bent door `git reflog` aan te roepen:

```
$ git reflog  
1a410ef HEAD@{0}: reset: moving to 1a410ef  
ab1afef HEAD@{1}: commit: modified repo.rb a bit  
484a592 HEAD@{2}: commit: added repo.rb
```

Hier kunnen we de twee commits zien die we uitgechecked hebben gehad, maar hier is ook niet veel informatie te zien. Om dezelfde informatie op een veel nuttiger manier te zien, kunnen we `git log -g` aanroepen, die je een normale log uitvoert voor je reflog laat zien.

```
$ git log -g  
commit 1a410efbd13591db07496601ebc7a059dd55cf9  
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)  
Reflog message: updating HEAD  
Author: Scott Chacon <schacon@gmail.com>  
Date: Fri May 22 18:22:37 2009 -0700  
  
third commit  
  
commit ab1afef80fac8e34258ff41fc1b867c702daa24b  
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)  
Reflog message: updating HEAD  
Author: Scott Chacon <schacon@gmail.com>  
Date: Fri May 22 18:15:24 2009 -0700  
  
modified repo.rb a bit
```

Het lijkt erop dat de laatste commit degene is die je kwijt was geraakt, dus je kunt deze terughalen door een nieuwe branch te maken op die commit. Bijvoorbeeld, je kunt een branch genaamd **recover-branch** beginnen op die commit (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Toppie - nu heb je een branch genaamd **recover-branch** die staat waar je **master**-branch heeft gestaan, en de eerste twee commits worden weer bereikbaar. Okay, nu stel dat je verlies om wat voor reden dan ook niet meer in de reflog zichtbaar is - je kunt dat naspelen door **recover-branch** weg te halen en de reflog weg te gooien. Nu kan je op geen enkele manier meer bij die eerste twee commits:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Omdat de reflog gegevens worden bewaard in de **.git/logs/** directory, heb je effectief geen reflog. Hoe kan je nu die commit herstellen? Een manier is om het **git fsck** instrument te gebruiken die je database op integriteit controleert. Als je het aanroeft met de **--full** optie, zal het je alle objecten laten zien waar geen enkele andere object naar verwijst:

```
$ git fsck --full
Checking object directories: 100% (256/256), done.
Checking objects: 100% (18/18), done.
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In dit geval kan je de ontbrekende commit zien achter de tekst “dangling commit”. Je kunt het op dezelfde manier herstellen, door een branch toe te voegen die wijst naar die SHA-1.

## Objecten verwijderen

Er zijn ontzettend veel goede dingen met Git, maar een eigenschap die problemen kan veroorzaken is het feit dat een **git clone** de hele geschiedenis van het project download, inclusief elke versie van elk bestand. Dit is prima als het hele spul broncode is, omdat Git optimaal is ingericht om die gegevens efficiënt te comprimeren. Echter, als iemand op enig moment in de geschiedenis van je project een enorm groot bestand heeft toegevoegd, zal elke kloon voor altijd gedwongen zijn om dat grote bestand te downloaden, zelfs als het in de volgende commit uit het project zou zijn verwijderd. Omdat het vanuit de historie bereikt kan worden, zal het altijd aanwezig zijn.

Dit kan een groot probleem zijn als je Subversion of Perforce repositories naar Git aan het converteren bent. Omdat je in deze systemen niet de hele historie downloadt, heeft dit soort toevoegingen veel minder gevolgen. Als je een import vanuit een ander systeem gedaan hebt, of om een andere reden vindt dat je repository veel groter is dan het zou moeten zijn, volgt hier een manier om uit te vinden hoe je grote objecten kunt vinden en verwijderen.

**Wees gewaarschuwd: deze techniek is destructief voor je commit historie.** Het herschrijft elke commit object sinds de eerste boom die je moet wijzigen om een referentie van een groot bestand te verwijderen. Als je dit direct na een import doet, voordat iemand is begonnen om werk op de commit te baseren zit je nog goed - anders zal je alle bijdragers moeten vertellen dat ze hun werk moeten rebasen op je nieuwe commits.

Om dit te demonstreren, ga je een groot bestand in je test repository toevoegen, deze in de volgende commit verwijderen, het opzoeken en het definitief uit de repository verwijderen. Eerst: voeg een groot object toe aan je historie:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Oeps - je wilde niet een enorme tarball aan je project toevoegen. Laten we 'm maar snel weggooien:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Nu ga je **gc** aanroepen op je database en kijken hoeveel ruimte je nu gebruikt:

```
$ git gc
Counting objects: 17, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Je kunt het **count-objects** commando gebruiken om snel te zien hoeveel ruimte je gebruikt:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

De **size-pack** regel is de grootte van je packfiles in kilobytes, dus je gebruikt bijna 5MB. Voor de laatste commit gebruikte je iets van 2K - het is duidelijk, het verwijderen van het bestand van de vorige commit heeft het niet uit je historie verwijderd. Elke keer als iemand deze repository kloont, zullen ze alle 5MB moeten klonen alleen om dit kleine project te pakken te krijgen, omdat je per ongeluk een groot bestand hebt toegevoegd. Laten we er vanaf komen.

Eerst zal je het moeten vinden. In dit geval, weet je al welk bestand het is. Maar stel dat je het niet zou weten; hoe zou je uitvinden welk bestand of bestanden er zoveel ruimte in beslag nemen? Als je **git gc** aanroeft, komen alle bestanden in een packfile terecht; je kunt de grote objecten vinden door een ander binnenwerk commando **git verify-pack** aan te roepen en de uitvoer te sorteren op het derde veld in de uitvoer, wat de bestandsgrootte is. Je kunt het ook door het **tail** commando pipen, omdat je alleen geïnteresseerd bent in de laatste paar grootste bestanden:

```
$ git verify-pack -v .git/objects/pack/pack-29...69.idx \
| sort -k 3 -n \
| tail -3
dadf7258d699da2c8d89b09ef6670edb7d5f91b4 commit 229 159 12
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 22044 5792 4977696
82c99a3e86bb1267b236a4b6eff7868d97489af1 blob 4975916 4976258 1438
```

Het grote object staat onderaan: 5MB. Om uit te vinden welk bestand dit is, ga je het **rev-list** commando gebruiken, die je al eventjes gebruikt hebt in [Een specifiek commit-bericht formaat afdwingen](#). Als je **--objects** doorgeeft aan **rev-list**, laat het alle SHA-1s zien van commits en ook de blob SHA-1s met het bestandspad die ermee verbonden is. Je kunt dit gebruiken om de naam van jouw blob te vinden:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Nu moet je dit bestand uit alle bomen in je verleden verwijderen. Je kunt eenvoudig zien welke commits dit bestand hebben gewijzigd:

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

Je moet alle commits herschrijven die stroomafwaarts van [7b30847](#) liggen om dit bestand volledig uit je Git historie te verwijderen. Om dit te doen, gebruik je [filter-branch](#), die je gebruikt hebt in [Geschiedenis herschrijven](#):

```
$ git filter-branch --index-filter \  
  'git rm --ignore-unmatch --cached git.tgz' -- 7b30847^..  
Rewrite 7b30847d080183a1ab7d18fb202473b3096e9f34 (1/2)rm 'git.tgz'  
Rewrite dadf7258d699da2c8d89b09ef6670edb7d5f91b4 (2/2)  
Ref 'refs/heads/master' was rewritten
```

De [--index-filter](#) optie is gelijk aan de [--tree-filter](#) optie zoals gebruikt in [Geschiedenis herschrijven](#), behalve dat in plaats van een commando door te geven die bestanden wijzigt die op schijf zijn uitgecheckt, je elke keer het staging gebied of index wijzigt.

In plaats van een specifiek bestand te verwijderen met iets als [rm file](#), moet je het verwijderen met [git rm --cached](#) - je moet het van de index verwijderen, niet van schijf. De reden hierachter is snelheid - omdat Git niet elke revisie hoeft uit te checken naar schijf voordat het je filter aanroept kan het proces veel, veel sneller werken. Je kunt dezelfde resultaat met [--tree-filter](#) bereiken als je wilt. De [--ignore-unmatch](#) optie bij [git rm](#) vertelt het geen fout te genereren als het patroon die je probeert te vinden niet aanwezig is. Als laatste, vraag je [filter-branch](#) om je historie te alleen herschrijven vanaf de [7b30847](#) en later, omdat je weet dat dit de plaats is waar het probleem begon. Anders zou het vanaf het begin starten en onnodig langer zou duren.

Je historie bevat niet langer meer een referentie naar dat bestand. Echter, je reflog en een nieuwe set van refs die Git toegevoegd heeft toen je het [filter-branch](#) gebruikte bestaan nog steeds onder [.git/refs/original](#) nog steeds wel, dus je zult deze moeten verwijderen en dan de database opnieuw inpakken. Je moet afraken van alles wat maar een verwijzing heeft naar die oude commits voordat je opnieuw inpakt.

```
$ rm -Rf .git/refs/original  
$ rm -Rf .git/logs/  
$ git gc  
Counting objects: 15, done.  
Delta compression using up to 8 threads.  
Compressing objects: 100% (11/11), done.  
Writing objects: 100% (15/15), done.  
Total 15 (delta 1), reused 12 (delta 0)
```

Laten we eens kijken hoeveel ruimte je gewonnen hebt.

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

De grootte van de ingepakte repository is gekrompen naar 8K, wat veel beter is dan 5MB. Je kunt aan de waarde van de grootte zien dat het grote object nog steeds in je losse objecten zit, dus het is nog niet weg; maar het wordt niet meer uitgewisseld met een push of een toekomstige kloon, en daar gaat het uiteindelijk om. Als je het echt wilt, zou je het object volledig kunnen verwijderen door `git prune` aan te roepen met de `--expire` optie:

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

## Omgevingsvariabelen

Git draait altijd in een `bash` shell, en gebruikt een aantal shell omgevingsvariabelen om te bepalen hoe het zich moet gedragen. Af en toe is het handig om te weten welke deze zijn, en hoe ze kunnen worden gebruikt om Git zich te laten gedragen op de manier zoals jij dat wilt. Dit is geen uitputtende lijst van alle omgevingsvariabelen waar Git naar kijkt, maar we zullen de meest nuttige behandelen.

### Globaal gedrag

Een bepaald deel van het algemene gedrag van Git als computer programma is afhankelijk van omgevingsvariabelen.

`GIT_EXEC_PATH` bepaalt waar Git zijn sub-programma's zoekt (zoals `git-commit`, `git-diff` en andere). Je kunt de huidige waarde zien door `git --exec-path` aan te roepen.

`HOME` wordt over het algemeen niet beschouwd als aanpasbaar (te veel andere zaken zijn hiervan afhankelijk), maar dit is waar Git op zoek gaat naar het globale configuratie bestand. Als je een echte overdraagbare Git installatie wilt, compleet met globale configuratie, kan je `HOME` overschrijven in het shell profiel van de draagbare Git.

**PREFIX** is hiermee vergelijkbaar, maar voor de systeem-brede configuratie. Git kijkt naar dit bestand op `$PREFIX/etc/gitconfig`.

**GIT\_CONFIG\_NOSYSTEM**, indien gezet, schakelt dit het gebruik van het systeem-brede configuratie bestand uit. Dit kan nuttig zijn als je systeem configuratie je commando's in de weg zit, maar je hebt hier geen toegang toe om het te wijzigen of te verwijderen.

**GIT\_PAGER** bepaalt welk programma er gebruikt wordt om uitvoer van meerdere pagina's op de commando regel te laten zien. Als deze waarde niet gezet is, wordt **PAGER** gebruikt als achtervanger.

**GIT\_EDITOR** is de editor die Git zal aanroepen als de gebruiker tekst moet wijzigen (een commit bericht, bijvoorbeeld). Als deze waarde niet is gezet, wordt **EDITOR** gebruikt.

## Locaties van de repository

Git gebruikt een aantal omgevingsvariabelen om te bepalen hoe het met de huidige repository samenwerkt.

**GIT\_DIR** is de locatie van de `.git` folder. Als dit niet is opgegeven, loopt Git de directory structuur omhoog tot het op `~` of `/` uitkomt, bij elke stap opzoek naar een `.git` directory.

**GIT\_CEILING\_DIRECTORIES** bepaalt het gedrag bij het zoeken naar een `.git` directory. Als je directories in gaan die langzaam zijn (zoals die op een tape drive, of over een langzame netwerkverbinding), zou je ervoor kunnen kiezen om Git te eerder laten stoppen met proberen dan het anders zou doen, zeker als Git wordt aangeroepen als je shell prompt wordt opgebouwd.

**GIT\_WORK\_TREE** is de locatie van de root van de werkdirectory voor een non-bare repository. Als `--git-dir` of **GIT-DIR** wordt gespecificeerd maar geen van `--work-tree`, **GIT\_WORK\_TREE** of **core.worktree** is gegeven wordt de huidige werk-directory beschouwd als het hoogste niveau van je werk-tree.

**GIT\_INDEX\_FILE** is het pad naar het index bestand (alleen bij non-bare repositories).

**GIT\_OBJECT\_DIRECTORY** kan worden gebruikt om de locatie van de directory aan te geven die normaalgesproken onder `.git/objects` te vinden is.

**GIT\_ALTERNATE\_OBJECT\_DIRECTORIES** is een met dubbele punt gescheiden lijst (geformateerd als `/dir/een:/dir/twee:...`) die vertelt waar Git moet zoeken naar objecten als ze niet in **GIT\_OBJECT\_DIRECTORY** te vinden zijn. Als je toevallig veel projecten hebt met grote bestanden die precies dezelfde inhoud hebben, kan dit worden gebruikt om te voorkomen dat er teveel kopieën hiervan worden opgeslagen.

## Pathspecs

Een “pathspec” refereert aan hoe je paden opgeeft bij zaken in Git, inclusief het gebruik van wildcards. Deze worden gebruikt in het `.gitignore` bestand, maar ook op de commando-regel (`git add *.c`).

**GIT\_GLOB\_PATHSPECS** en **GIT\_NOGLOB\_PATHSPECS** bepalen het standaard gedrag van wildcards in pathspecs. Als **GIT\_GLOB\_PATHSPECS** de waarde 1 heeft, gedragen wildcard karakters als wildcards (wat het standaard gedrag is); als **GIT\_NOGLOB\_PATHSPECS** de waarde 1 heeft, passen wildcard

karakters alleen op zichzelf, wat inhoudt dat iets als `*c` alleen een bestand *met de naam “\*.c”* zou passen, in plaats van elk bestand waarvan de naam eindigt op `.c`. Je kunt dit gedrag in specifieke gevallen overschrijven door de pathspecs te laten beginnen met `:(glob)` of `:(literal)`, als in `:(glob)*.c`.

`GIT_LITERAL_PATHSPECS` schakelt beide bovenstaande gedragingen uit; geen enkele wildcard karakter zal werken, en de overschrijvende prefixes worden ook uitgeschakeld.

`GIT_ICASE_PATHSPECS` zet alle pathspecs in voor ongevoelige werkwijze voor hoofdletters en kleine letters.

## Committen

De uiteindelijke aanmaak van een Git commit object wordt normaalgesproken gedaan door `git-commit-tree`, die de omgevingsvariabelen gebruikt als zijn primaire bron van informatie, waarbij wordt teruggevallen op configuratiewaarden alleen als deze niet aanwezig zijn.

`GIT_AUTHOR_NAME` is de mens-leesbare naam in het “author” veld.

`GIT_AUTHOR_EMAIL` is de email voor het “author” veld.

`GIT_AUTHOR_DATE` is de datum/tijd waarde die voor het “author” veld wordt gebruikt.

`GIT_COMMITTER_NAME` bepaalt de mens-leesbare naam voor het “committer” veld.

`GIT_COMMITTER_EMAIL` is het email adres voor het “committer” veld.

`GIT_COMMITTER_DATE` wordt gebruikt voor de datum/tijd waarde in het “committer” veld.

`EMAIL` is de terugvalwaarde voor het email adres in geval de configuratie waarde in `user.email` niet is ingevuld. Als *deze* niet is ingevuld, valt Git terug op de systeemgebruiker en host naam.

## Network communicatie

Git gebruikt de `curl` library om netwerk operaties over HTTP te doen, dus `GIT_CURL_VERBOSE` vertelt Git om alle berichten uit te sturen die worden gegenereerd door deze library. Dit is gelijk aan het intypen van `curl -v` op de commandoregel.

`GIT_SSL_NO_VERIFY` vertelt Git om de SSL certificaten niet te controleren. Dit kan soms nodig zijn als je een door jezelf ondertekende certificaat gebruikt om Git repositories te bedienen via HTTPS, of je bent bezig met het opzetten van een Git server maar je hebt nog geen volwaardig certificaat geïnstalleerd.

Als de gegevenssnelheid van een HTTP operatie lager is dan `GIT_HTTP_LOW_SPEED_LIMIT` bytes per seconde voor langer dan `GIT_HTTP_LOW_SPEED_TIME` seconden, zal Git die operatie afbreken. Deze waarden overschrijven de `http.lowSpeedLimit` en `http.lowSpeedTime` configuratie waarden.

`GIT_HTTP_USER_AGENT` zet de user-agent tekenreeks die Git gebruikt wanneer het communiceert via HTTP. De standaard waarde is iets als `git/2.0.0`.

## Diffen en Mergen

**GIT\_DIFF\_OPTS** is een beetje een misnomer. De enige geldige waarden zijn `-u<n>` of `--unified=<n>`, wat het aantal contextregels bepaalt die worden getoond met een `git diff` commando.

**GIT\_EXTERNAL\_DIFF** wordt gebruikt als een overschrijving voor de `diff.external` configuratie waarde. Als het een waarde heeft, zal Git dit programma gebruiken als `git diff` wordt aangeroepen.

**GIT\_DIFF\_PATH\_COUNTER** en **GIT\_DIFF\_PATH\_TOTAL** zijn nuttig binnen het programma die wordt gespecificeerd door **GIT\_EXTERNAL\_DIFF** of **diff.external**. Het eerste geeft aan welk bestand in een reeks van bestanden wordt gedifft (beginnend met 1), en de laatste is het totaal aantal bestanden in de reeks.

**GIT\_MERGE\_VERBOSITY** bepaalt de uitvoer voor de recursieve merge strategie. De toegestane waarden zijn als volgt:

- 0 voert niets uit, behalve mogelijk een enkele foutbericht.
- 1 laat alleen conflicten zien.
- 2 laat ook bestandswijzigingen zien.
- 3 geeft uitvoer als bestanden worden overgeslagen omdat ze niet zijn gewijzigd.
- 4 laat alle paden zien als ze worden verwerkt.
- 5 en hoger laten gedetailleerd debug informatie zien.

De standaardwaarde is 2.

## Debuggen

Wil je *echt* zien waar Git zoal mee bezig is? Git heeft een redelijk volledige set van traces ingebouwd, en alles wat je hoeft te doen is ze aan te zetten. De mogelijke waarden van deze variabelen zijn als volgt:

- “true”, “1”, of “2” - de trace categorie wordt naar stderr geschreven.
- Een absoluut pad beginnend met `/` - de trace uitvoer wordt naar dat bestand geschreven.

**GIT\_TRACE** bepaalt traces over het algemeen die niet in een andere specifieke categorie vallen. Dit is inclusief de expansie van aliassen, en het delegeren naar andere sub-programma's.

```
$ GIT_TRACE=true git lga
20:12:49.877982 git.c:554          trace: exec: 'git-lga'
20:12:49.878369 run-command.c:341    trace: run_command: 'git-lga'
20:12:49.879529 git.c:282          trace: alias expansion: lga => 'log' '--graph'
'--pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.879885 git.c:349          trace: built-in: git 'log' '--graph' '--
pretty=oneline' '--abbrev-commit' '--decorate' '--all'
20:12:49.899217 run-command.c:341    trace: run_command: 'less'
20:12:49.899675 run-command.c:192    trace: exec: 'less'
```

**GIT\_TRACE\_PACK\_ACCESS** bepaalt het traceren van packfile toegang. Het eerste veld is de packfile die wordt benaderd, het tweede is de relatieve afstand (offset) binnen dat bestand:

```
$ GIT_TRACE_PACK_ACCESS=true git status
20:10:12.081397 sha1_file.c:2088      .git/objects/pack/pack-c3fa...291e.pack 12
20:10:12.081886 sha1_file.c:2088      .git/objects/pack/pack-c3fa...291e.pack 34662
20:10:12.082115 sha1_file.c:2088      .git/objects/pack/pack-c3fa...291e.pack 35175
# [...]
20:10:12.087398 sha1_file.c:2088      .git/objects/pack/pack-e80e...e3d2.pack
56914983
20:10:12.087419 sha1_file.c:2088      .git/objects/pack/pack-e80e...e3d2.pack
14303666
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

**GIT\_TRACE\_PACKET** zet traceren op pakket-niveau voor netwerk operaties aan.

```
$ GIT_TRACE_PACKET=true git ls-remote origin
20:15:14.867043 pkt-line.c:46          packet:          git< # service=git-upload-
pack
20:15:14.867071 pkt-line.c:46          packet:          git< 0000
20:15:14.867079 pkt-line.c:46          packet:          git<
97b8860c071898d9e162678ea1035a8ced2f8b1f HEAD\0multi_ack thin-pack side-band side-
band-64k ofs-delta shallow no-progress include-tag multi_ack_detailed no-done
symref=HEAD:refs/heads/master agent=git/2.0.4
20:15:14.867088 pkt-line.c:46          packet:          git<
0f20ae29889d61f2e93ae00fd34f1cdb53285702 refs/heads/ab/add-interactive-show-diff-func-
name
20:15:14.867094 pkt-line.c:46          packet:          git<
36dc827bc9d17f80ed4f326de21247a5d1341fbc refs/heads/ah/doc-gitk-config
# [...]
```

**GIT\_TRACE\_PERFORMANCE** bepaalt het loggen van performance gegevens aan. De uitvoer laat zien hoe lang elk specifieke git aanroep duurt.

```
$ GIT_TRACE_PERFORMANCE=true git gc
20:18:19.499676 trace.c:414           performance: 0.374835000 s: git command: 'git'
'pack-refs' '--all' '--prune'
20:18:19.845585 trace.c:414           performance: 0.343020000 s: git command: 'git'
'reflog' 'expire' '--all'
Counting objects: 170994, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (43413/43413), done.
Writing objects: 100% (170994/170994), done.
Total 170994 (delta 126176), reused 170524 (delta 125706)
20:18:23.567927 trace.c:414           performance: 3.715349000 s: git command: 'git'
'pack-objects' '--keep-true-parents' '--honor-pack-keep' '--non-empty' '--all' '--
reflog' '--unpack-unreachable=2.weeks.ago' '--local' '--delta-base-offset'
'.git/objects/pack/.tmp-49190-pack'
20:18:23.584728 trace.c:414           performance: 0.000910000 s: git command: 'git'
'prune-packed'
20:18:23.605218 trace.c:414           performance: 0.017972000 s: git command: 'git'
'update-server-info'
20:18:23.606342 trace.c:414           performance: 3.756312000 s: git command: 'git'
'repack' '-d' '-l' '-A' '--unpack-unreachable=2.weeks.ago'
Checking connectivity: 170994, done.
20:18:25.225424 trace.c:414           performance: 1.616423000 s: git command: 'git'
'prune' '--expire' '2.weeks.ago'
20:18:25.232403 trace.c:414           performance: 0.001051000 s: git command: 'git'
'rerere' 'gc'
20:18:25.233159 trace.c:414           performance: 6.112217000 s: git command: 'git'
'gc'
```

**GIT\_TRACE\_SETUP** laat informatie zien over wat Git ontdekt over de repository en de omgeving waar het mee samenwerkt.

```
$ GIT_TRACE_SETUP=true git status
20:19:47.086765 trace.c:315           setup: git_dir: .git
20:19:47.087184 trace.c:316           setup: worktree: /Users/ben/src/git
20:19:47.087191 trace.c:317           setup: cwd: /Users/ben/src/git
20:19:47.087194 trace.c:318           setup: prefix: (null)
On branch master
Your branch is up-to-date with 'origin/master'.
nothing to commit, working directory clean
```

## Diversen

**GIT\_SSH**, indien meegegeven, is een programma dat wordt aangeroepen in plaats van **ssh** als Git probeert verbinding te leggen naar een SSH host. Het wordt aangeroepen als **\$GIT\_SSH [gebruikersnaam@]host [-p <poort>] <commando>**. Merk op dat dit niet de makkelijkste manier is om de manier waarop **ssh** wordt aangeroepen aan te passen; het zal extra commando-regel parameters niet ondersteunen, dus je zult een wrapper script moeten schrijven en **GIT\_SSH** hiernaar moeten laten wijzen. Het is waarschijnlijk makkelijker om gewoon het **~/.ssh/config** bestand hiervoor te

gebruiken.

**GIT\_ASKPASS** overschrijft de waarde van de `core.askpass` configuratie waarde. Dit is het programma dat wordt aangeroepen elke keer als Git de gebruiker moet vragen om de aanloggegevens, die een tekst prompt mag verwachten als een commando-regel argument, en het antwoord moet teruggeven op `stdout`. (Zie [Het opslaan van inloggegevens](#) voor meer over dit subsysteem.)

**GIT\_NAMESPACE** bepaalt de toegang tot refs in een namespace, en is gelijkwaardig aan de `--namespace` vlag. Dit wordt voornamelijk nuttig aan de kant van de server, waar je misschien meerdere forks wilt opslaan van een enkele repository, waarbij alleen de refs apart wordt gehouden.

**GIT\_FLUSH** kan gebruikt worden om Git te dwingen om niet gebufferde I/O te gebruiken als het incrementeel naar `stdout` schrijft. Een waarde van 1 heeft als gevolg dat Git vaker wegschrijft, een waarde van 0 heeft tot gevolg dat alle uitvoer eerst wordt gebufferd. De standaard waarde (als deze waarde niet wordt opgegeven) is om een toepasselijke bufferschema te kiezen afhankelijk van de aktiviteit en de uitvoer-modus.

**GIT\_REFLOG\_ACTION**\* laat je de omschrijvende tekst bepalen die naar de reflog wordt geschreven. Hier is een voorbeeld:

```
$ GIT_REFLOG_ACTION="my action" git commit --allow-empty -m 'my message'  
[master 9e3d55a] my message  
$ git reflog -1  
9e3d55a HEAD@{0}: my action: my message
```

## Samenvatting

Je zou nu een redelijk goed begrip moeten hebben wat Git in de achtergrond uitvoert en, tot op zekere hoogte, hoe het is geïmplementeerd. Dit hoofdstuk heeft een aantal binnenwerk (plumbing) commando's - commando's die een niveau lager en simpeler zijn dan de deftige (porcelain) commando's waarover je in de rest van het boek gelezen hebt. Het begrip over hoe Git op een dieper niveau werkt zou het makkelijker moeten maken om te begrijpen waarom het doet wat het doet en helpen bij het schrijven van je eigen gereedschappen en scripts om jouw specifieke werkwijze te ondersteunen.

Git als een op inhoud-adresseerbaar bestandssysteem is een zeer krachtig gereedschap die je eenvoudig kunt gebruiken voor meer dan alleen een versiebeheersysteem. We hopen dat je je pasverworven kennis over het binnenwerk van Git kunt gebruiken om je eigen gave toepassing van deze technologie te implementeren en je meer op je gemak te voelen bij het gebruik van Git op meer geavanceerde manieren.

# Appendix A: Git in andere omgevingen

Als je het hele boek gelezen hebt, zal je erg veel geleerd hebben over het gebruik van Git op de commando regel. Je kunt met lokale bestanden werken, je repository verbinden met andere via een netwerk, en op een effectieve manier werken met anderen. Maar daarmee houdt het verhaal niet op; Git wordt normaalgesproken gebruikt als onderdeel van een groter ecosysteem, en het werkstation is niet altijd de beste manier om ermee te werken. We zullen nu een kijkje nemen naar andere soorten omgevingen waar Git nuttig kan zijn, en hoe andere applicaties (inclusief de jouwe) zij aan zij samenwerken met Git.

## Grafische interfaces

De natuurlijke omgeving van Git is het werkstation. Nieuwe mogelijkheden verschijnen daar het eerst, en alleen op de commandoregel is de volle kracht van Git volledig tot je beschikking. Maar platte tekst is niet de beste keuze voor alle taken; soms heb je gewoon een meer visuele representatie nodig, en sommige gebruikers voelen zich veel meer op hun gemak bij een *point-and-click* interface.

Het is belangrijk op te merken dat verschillende interfaces zijn toegesneden op andere workflows. Sommige clients laten slechts een verzameling zorgvuldig uitgekozen onderdelen van Git zien, dit om een specifieke manier van werken te ondersteunen die de auteurs ervan beschouwen als effectief. Vanuit dat oogpunt gezien zijn geen van deze gereedschappen “beter” te noemen dan een ander, ze zijn gewoonweg beter in staat om hun specifieke doel te dienen. Merk ook dat er niets is wat deze grafische clients kunnen doen wat niet vanaf de commando-regel te doen zou zijn; de commando-regel is nog steeds de plaats waar je de meeste mogelijkheden en controle hebt als je met je repositories werkt.

### gitk en git-gui

Als je Git installeert, krijg je de visuele gereedschappen erbij, zijnde `gitk` en `git-gui`.

`gitk` is een gereedschap waarmee je de geschiedenis grafisch kan bekijken. Zie het als een krachtige grafische schil over `git log` en `git grep`. Dit is het gereedschap die je gebruikt als je iets probeert te vinden wat in het verleden heeft plaatsgevonden, of de historie van je project probeert te laten zien.

Gitk is het eenvoudigste aan te roepen vanaf de commando-regel. Gewoon `cd` gebruiken om een Git repository in te gaan en dan dit typen:

```
$ gitk [git log options]
```

Gitk accepteert veel opties van de commando-regel, de meeste daarvan worden doorgegeven aan de onderliggende `git log` actie. Een van de meest nuttige hiervan is de `--all` vlag, die gitk vertelt om commits te laten zien die vanuit *elke* ref bereikbaar zijn, niet alleen HEAD. De interface van Gitk ziet er zo uit:

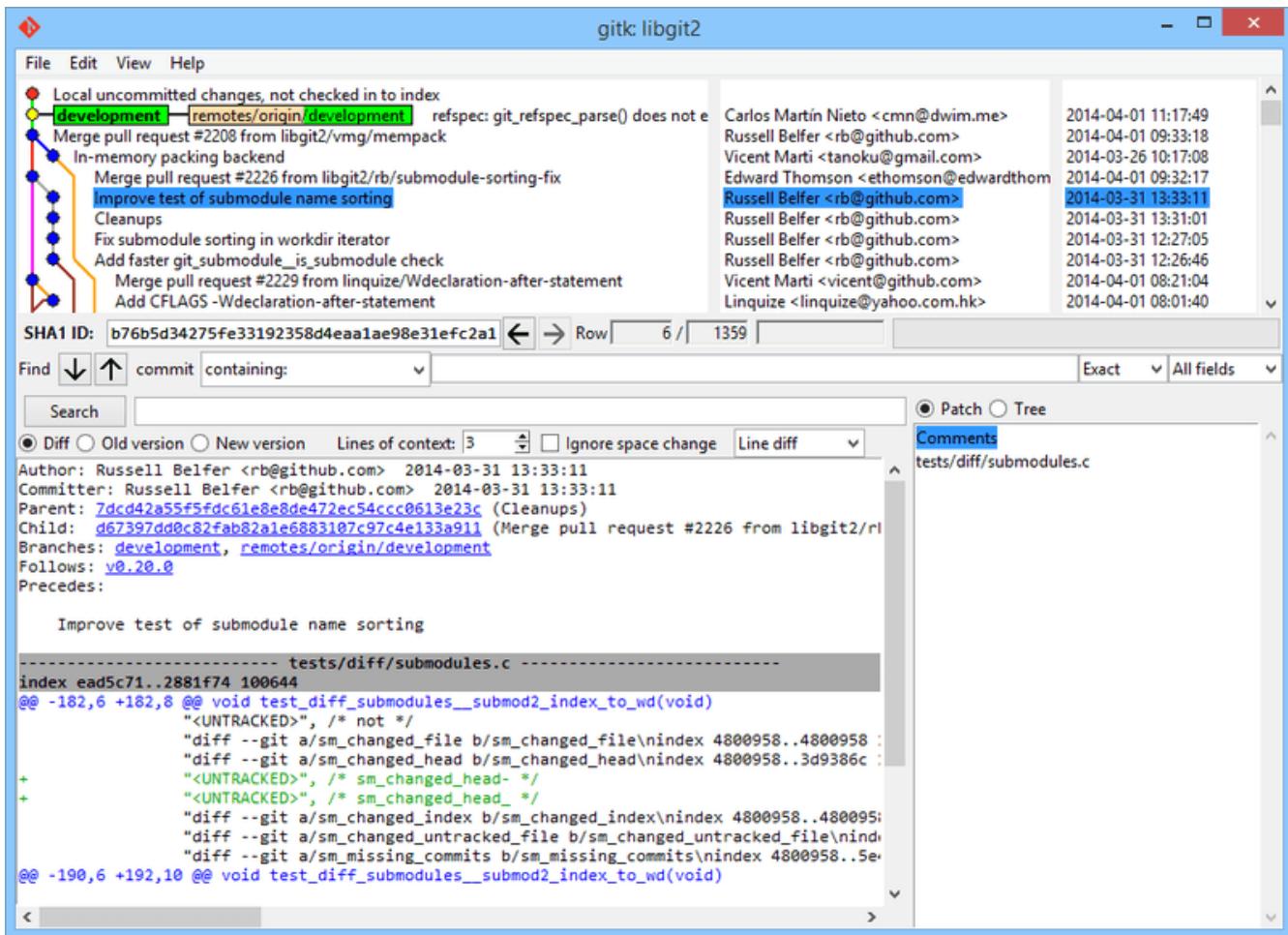


Figure 153. De `gitk` historie viewer.

Bovenaan is iets wat eruit ziet als de uitvoer van `git log --graph`; elke stip staat voor een commit, de lijntjes stellen ouderrelaties voor, en refs worden getoond als gekleurde vierkantjes. De gele stip stelt HEAD voor, en de rode stippen stellen wijzigingen voor die nog een commit moeten worden. Onderaan is een voorstelling van de geselecteerde commit; de commentaren en patch staan links en een samenvatting staat rechts. Hiertussen staat een verzameling van mogelijkheden om in de geschiedenis te zoeken.

`git-gui` daarentegen is primair een gereedschap om commits samen te stellen. Ook dit is het eenvoudigste om aan te roepen van de commando-regel:

```
$ git gui
```

En het ziet er ongeveer zo uit:

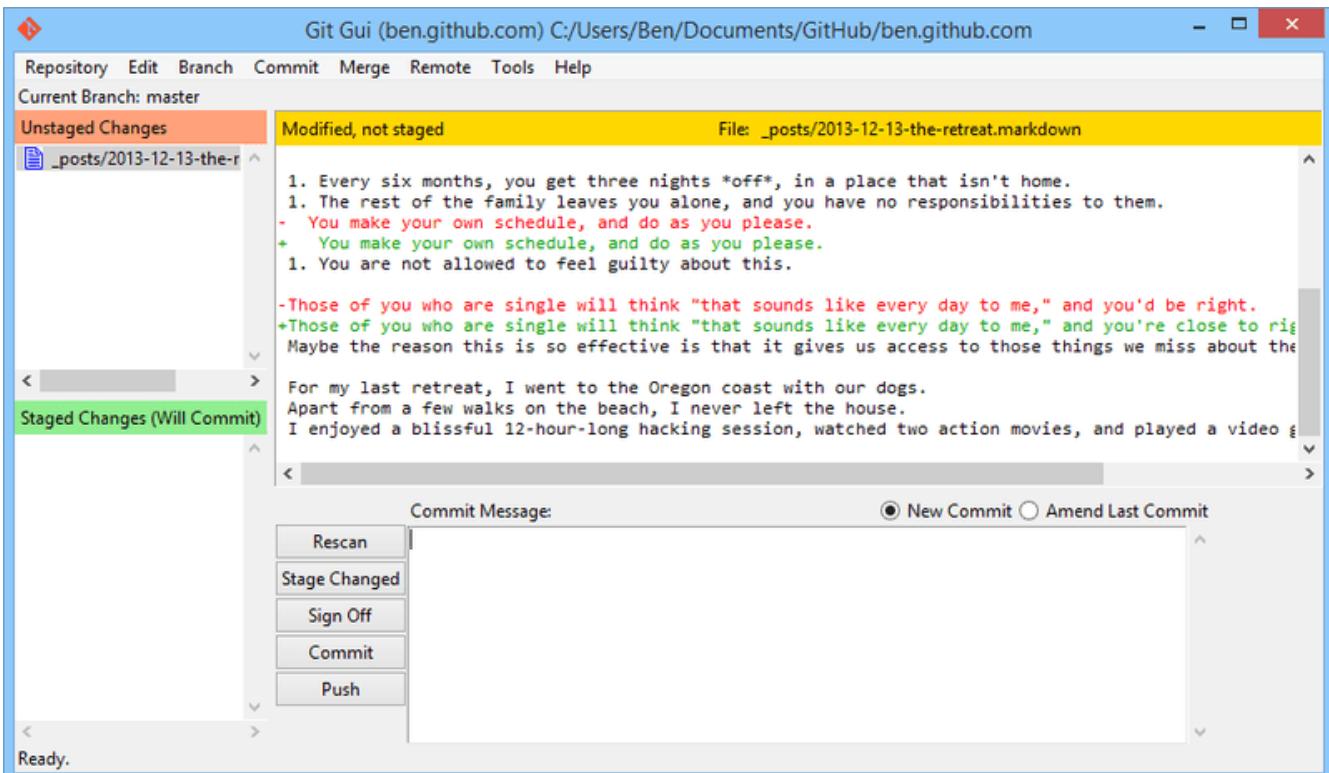


Figure 154. Het `git-gui` commit gereedschap.

Links staat de index; wijzigingen die nog niet zijn gestaged staan bovenaan, wijzigingen die zijn gestaged staan onderaan. Je kunt volledige bestanden tussen deze stadia verplaatsen door op hun icoontjes te klikken, of je selecteert een bestand voor bekijken door op de naam te klikken.

Rechtsboven wordt de diff getoond, hier worden de veranderingen getoond voor het bestand dat op dat moment is geselecteerd. Je kunt individuele hunks stagelen (of individuele regels) door in dit gebied rechts te klikken.

Rechtsonder is het gedeelte waar het bericht en acties kunnen worden ingetypt. Tik je bericht in de tekst-box en klik op "Commit" om iets vergelijkbaars te doen als `git commit`. Je kunt er ook voor kiezen om de laatste commit te amenderen door de "Amend" radio knop te klikken, wat het "Staged Changes" gedeelte vult met de inhoud van de laatste commit. Daarna kan je eenvoudigweg bepaalde wijzigingen kunt stagelen of unstagen, het commit bericht wijzigen en weer "Commit" klikken om de oude commit te vervangen met een nieuwe.

`gitk` en `git-gui` zijn voorbeelden van taak-georiënteerde gereedschappen. Elk van deze is toegesneden op een specifieke doel (respectievelijk geschiedenis bekijken en commits maken), en zaken die niet nodig zijn voor die taak zijn weggelaten.

## GitHub voor Mac en Windows

GitHub heeft twee workflow-georiënteerde clients gemaakt: een voor Windows en een voor Mac. Deze clients zijn goede voorbeelden van workflow-georiënteerde gereedschappen - in plaats van alle functionaliteit van Git beschikbaar te stellen, hebben ze zich gericht op een beperkte set van algemeen gebruikte functies die goed samenwerken. Ze zien er zo uit:

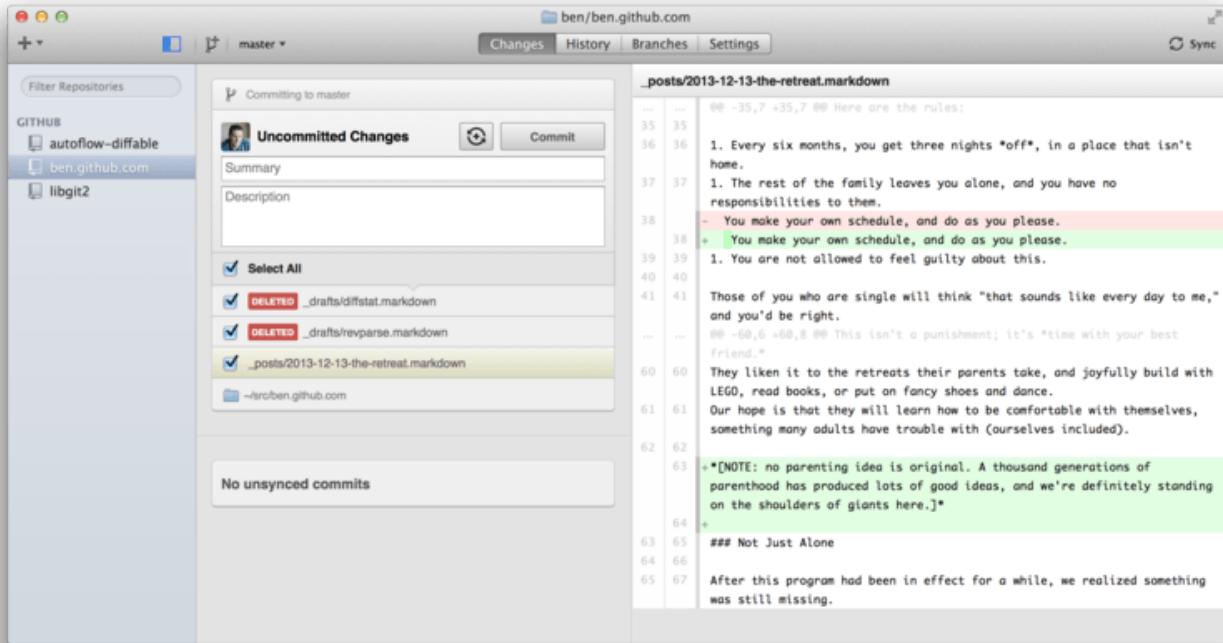


Figure 155. GitHub voor Mac.

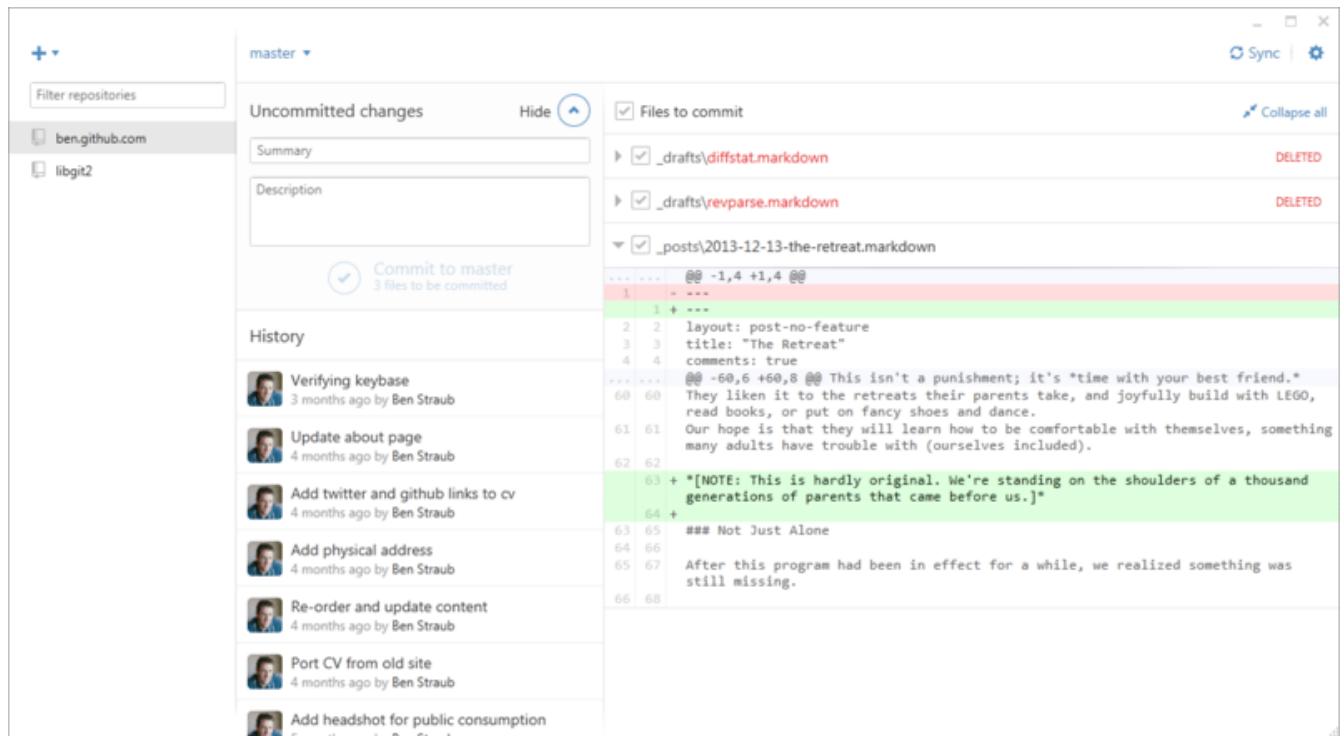


Figure 156. GitHub voor Windows.

Ze zijn ontworpen om zoveel mogelijk op dezelfde manier er uit te zien en te werken, dus we zullen ze in dit hoofdstuk als een enkel product behandelen. We zullen hier geen gedetailleerde behandeling geven van deze gereedschappen (ze hebben hun eigen documentatie), maar een snel overzicht van de “changes” view (waar je het meeste van je tijd zult besteden) is op zijn plaats.

- Links staat de lijst van repositories die de client volgt (trackt); je kunt een repository toevoegen (hetzij door het te klonen of door deze lokaal toe te voegen) door het “+” icoon te klikken die boven dit gebied staat.

- In het midden is een commit-invoer gebied, dit stelt je in staat om een commit bericht in te voeren, en de bestanden die hierin mee moeten te selecteren. (In Windows is de commit historie hier direct onder getoond; op Mac is het een aparte tab.)
- Rechts is een diff view, waar wordt getoond wat er in je werk directory is gewijzigd, of welke wijzigingen er in de geselecteerde commit zitten.
- Het laatste waar je op kunt letten is de “Sync” knop rechtsboven, wat de belangrijkste manier is waarmee je over het netwerk communiceert.



Je hebt geen GitHub account nodig om deze middelen te gebruiken. Alhoewel ze zijn ontworpen om de service van GitHub en aangeraden workflows in de aandacht te brengen, werken ze prima met elke willekeurige ander repository, en zullen netwerk operaties met alle Git hosts uitvoeren.

## Installatie

GitHub voor Windows kan worden gedownload van <https://windows.github.com>, en GitHub voor Mac van <https://mac.github.com>. Als de applicatie voor het eerst wordt aangeroepen, voeren ze je door alle instellingen die nodig zijn om Git te laten werken, zoals het invoeren van je naam en email adres, en beide stellen verstandige standaard instellingen voor, voor veel gebruikte configuratie opties, zoals caches voor inloggegevens en CRLF gedrag.

Beide zijn “overwinteraars” (“evergreen”) – updates worden gedownload en geïnstalleerd terwijl de applicaties in gebruik zijn. Bij deze installatie zit ook een versie van Git ingesloten, wat betekent dat je je waarschijnlijk geen zorgen hoeft te maken om het ooit handmatig te hoeven updaten. Op Windows heeft de client een shortcut om Powershell aan te roepen met Posh-git, waar we later in dit hoofdstuk meer over zullen vertellen.

De volgende stap is om de applicatie een aantal repositories te geven om mee te werken. De client toont je een lijst met repositories waar je toegang toe hebt op GitHub, en je kunt ze in één stap klonen. Als je al een lokale repository hebt, kan je de directory hiervan vanuit de Finder of Windows Explorer in de GitHub client slepen, en het zal in de lijst van repositories aan de linkerkant worden opgenomen.

## Aangeraden workflow

Als het eenmaal is geïnstalleerd en geconfigureerd, kan je de GitHub client voor veel reguliere Git taken gebruiken. De tool is gemaakt met de workflow die “GitHub Flow” heet in gedachten. We behandelen dit in meer detail in [De GitHub flow](#), maar de achterliggende gedachte is dat (a) je naar een branch gaat committen, en (b) dat je vrij regelmatig met een remote repository zult gaan synchroniseren.

Branch beheer is een van de gebieden waar deze twee tools afwijken. Op Mac, is er een knop bovenaan in de window voor het maken van een nieuwe branch:

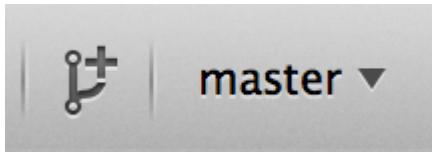


Figure 157. “Create Branch” knop op Mac.

In Windows wordt dit gedaan door de naam van de nieuwe branch in de branch-switching widget in te typen:

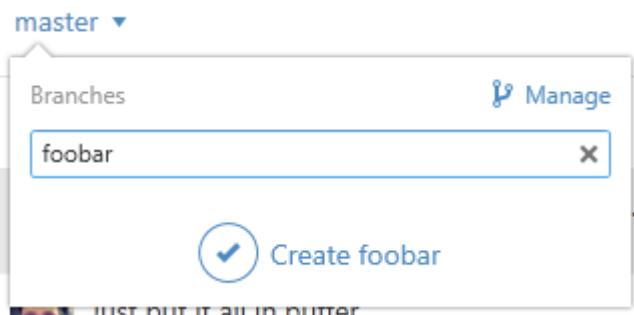


Figure 158. Een branch aanmaken in Windows.

Als je branch eenmaal is aangemaakt, is het maken van nieuwe commits min of meer rechtstreeks. Maak een aantal wijzigingen in je werk directory, en als je naar de GitHub client window gaat, zal het je laten zien welke bestanden zijn veranderd. Vul een commit bericht in, selecteer de bestanden die je erin wilt plaatsen, en klik op de “Commit” knop (ctrl-enter of ⌘-enter).

De belangrijkste manier waarmee je met andere repositories samenwerkt via het netwerk is via de “Sync” functie. Git heeft intern verschillende operaties voor pushen, fetchen, mergen en rebasen, maar de GitHub clients vatten die allemaal samen in een multi-stap functie. Hier is wat er gebeurt als je de Sync-knop klikt:

1. `git pull --rebase`. Als dit mislukt omdat er een merge conflict is, val dan terug op `git pull --no-rebase`.
2. `git push`.

Dit is de meeste gebruikelijke volgorde van netwerk commando's als je op deze manier werkt, dus als je ze in een commando samenvoegt bespaart het je veel tijd.

## Samenvatting

Deze gereedschappen zijn zeer geschikt voor de workflows waarvoor ze zijn ontworpen. Ontwikkelaars zowel als niet-ontwikkelaars kunnen op deze manier binnen een paar minuten met elkaar samenwerken op een project, en veel van de *best practices* voor dit soort van workflows zijn in de gereedschappen ingebakken. Echter, als je workflow hiervan afwijkt, of als je meer controle wilt over hoe en welke netwerk operaties er gedaan worden, raden we je een andere client of de commando-regel aan.

## Andere GUIs

Er zijn een aantal andere grafische Git clients, en ze variëren van gespecialiseerde gereedschappen gemaakt voor één specifiek doel tot aan applicaties die probereen alles wat Git kan beschikbaar te

stellen. De officiële Git website heeft een onderhouden lijst van de meest populaire clients op <http://git-scm.com/downloads/guis>. Een uitgebreidere lijst is beschikbaar op de Git wiki, op [https://git.wiki.kernel.org/index.php/Interfaces,\\_frontends,\\_and\\_tools#Graphical\\_Interfaces](https://git.wiki.kernel.org/index.php/Interfaces,_frontends,_and_tools#Graphical_Interfaces).

## Git in Visual Studio

Vanaf Visual Studio 2013 Update 1, hebben Visual Studio gebruikers een direct in hun IDE ingebouwde Git client. Visual Studio heeft al een hele tijd functionaliteit om te integreren met versie beheer, maar deze was gericht op gecentraliseerde systemen die bestanden blokkeren, en Git pastte niet goed in deze werkwijze. De Git ondersteuning van Visual Studio 2013 is gescheiden opgezet van deze oudere functionaliteit, en het resultaat is een veel betere aansluiting tussen Studio en Git.

Om de functionaliteit te starten, open je een project die wordt beheerd met Git (of type gewoon `git init` in een bestaand project), en selecteer View > Team Explorere in het menu. Je ziet dan de "Connect" view, die er ongeveer zo uit ziet:

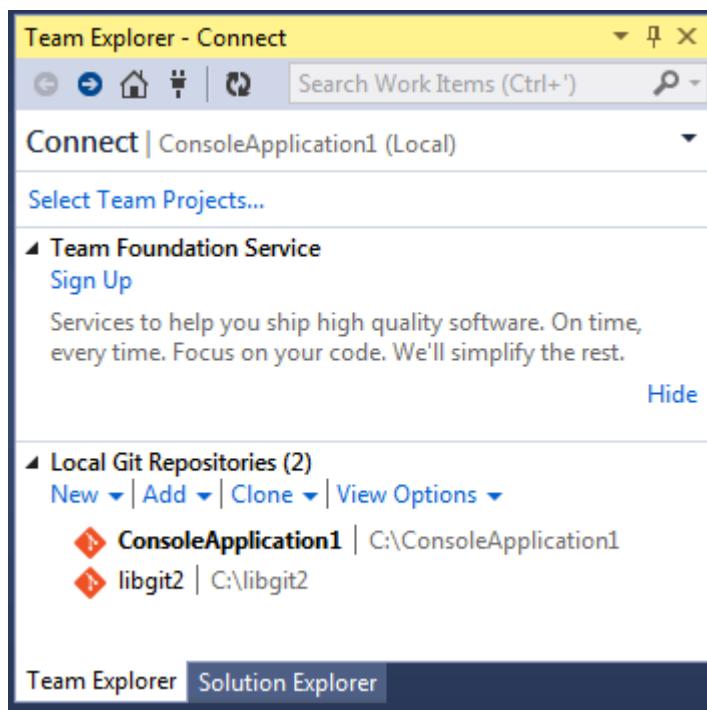


Figure 159. Verbinding maken met een Git repository vanuit Team Explorer.

Visual Studio onthoudt alle door Git beheerde projecten die je geopend hebt, en ze zijn beschikbaar in de lijst onderop. Als je het gewenste project daar niet ziet, klik dan op de "Add" link en type het pad naar de werk directory. Dubbel klikken op een van de lokale Git repositories resulteert in de Home view, die eruit ziet als [De "Home" view voor een Git repository in Visual Studio](#). Dit is een centrale plaats voor het uitvoeren van Git acties; als je code aan het *schrijven* bent, zal je waarschijnlijk de meeste tijd doorbrengen in de "Changes" view, maar wanneer het tijd wordt om wijzigingen die je teamgenoten hebben gemaakt te pullen, zal je de "Unsynched Commits" en "Branches" views gebruiken.

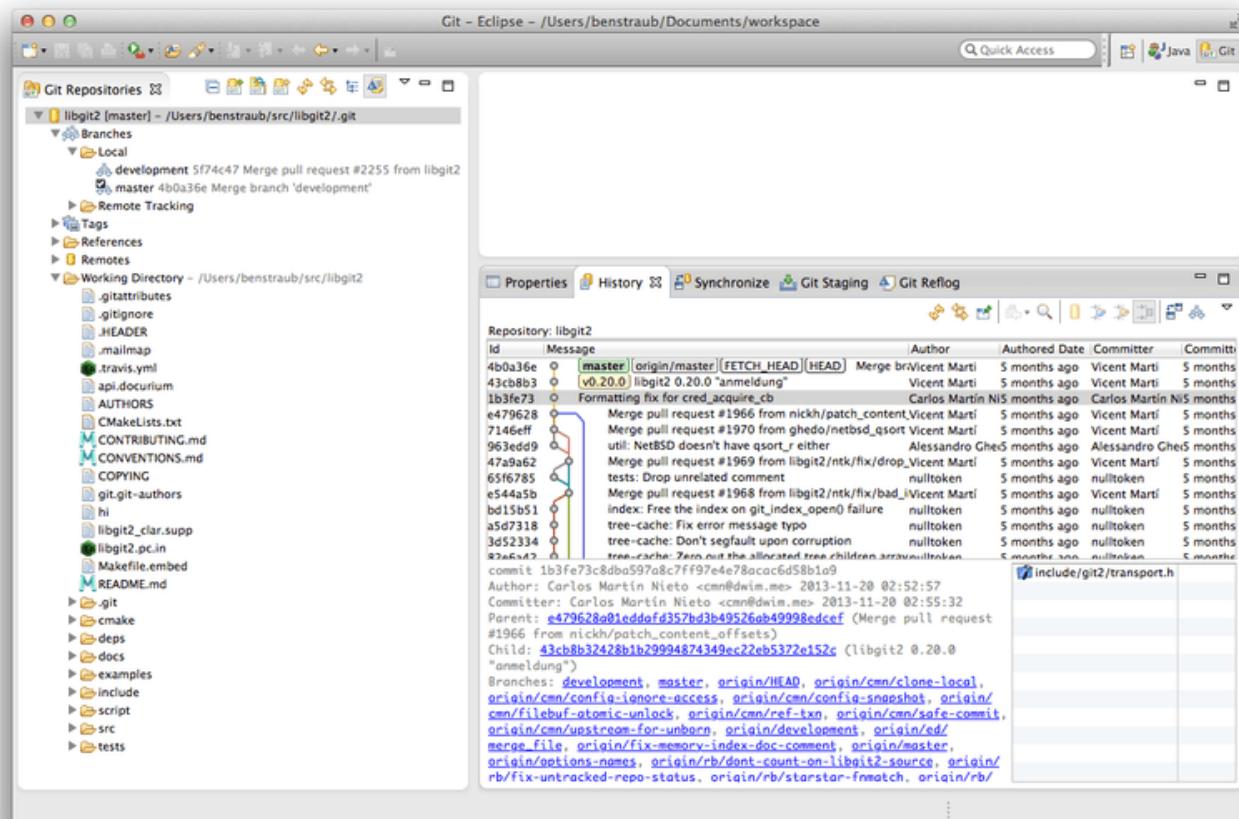


**Figure 160.** De "Home" view voor een Git repository in Visual Studio.

Visual Studio heeft nu een krachtige op taak georiënteerde gebruikers interface voor Git. Het bevat een lineaire historie view, een diff viewer, remote commando's en vele andere mogelijkheden. Voor de volledige documentatie van alle mogelijkheden hiervan (die hier niet past), verwijzen we je naar <http://msdn.microsoft.com/en-us/library/hh850437.aspx>.

## Git in Eclipse

Eclipse wordt geleverd met een plugin met de naam Egit, die een redelijk volledige interface verzorgt naar Git operaties. Je komt er door om te schakelen naar de Git Perspective (Window > Open Perspective > Other..., en selecteer "Git").



**Figure 161.** EGit omgeving van Eclipse.

Bij EGit wordt veel goede documentatie geleverd, die je kunt vinden door naar Help > Help Contents te gaan, en de "EGit Documentation" knoop te kiezen in de inhoudsopgave.

# Git in Bash

Als je een Bash gebruiker bent, kan je gebruik maken van een aantal van de mogelijkheden van je shell om je ervaringen met Git een stuk prettiger te maken. Git wordt namelijk met plugins voor een aantal shells geleverd, maar het staat standaard niet aan.

Allereerst, moet je een kopie van het bestand `contrib/completion/git-completion.bash` uit de Git broncode halen. Kopieer het naar een handige plaats, zoals je home directory, en voeg dit toe aan je `.bashrc`:

```
. ~/git-completion.bash
```

Als dat eenmaal gedaan is, wijzig je directory naar een git repository en type:

```
$ git chec<tab>
```

...en Bash zal automatisch het commando aanvullen naar `git checkout`. Dit werkt met alle subcommando's van Git, commando-regel parameters en namen van remotes en refs waarvan toepassing.

Het is ook nuttig om je prompt te wijzigen om informatie te laten zien over de Git repository in de huidige directory. Dit kan zo eenvoudig of complex worden als je zelf wilt, maar er zijn over het algemeen een paar stukken belangrijke informatie die de meeste mensen willen, zoals de huidige branch en de status van de werk directory. Om dit aan je prompt toe te voegen, kopiéer je simpelweg het `contrib/completion/git-prompt.sh` bestand van de Git broncode repository naar je home directory, en zet zo iets als dit in je `.bashrc`:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(_git_ps1 " (%s)")\$ '
```

De `\w` betekent het afdrukken van de huidige working directory, de `\$` drukt het \$ gedeelte van de prompt af, en `_git_ps1 " (%s)"` roept de functie aan die uitgevoerd wordt door `git-prompt.sh` met een formatterings argument. Nu zal je bash prompt er zo uit zien als je ergens in een door Git beheerd project zit:



Figure 162. Aangepaste bash prompt.

Al deze scripts hebben behulpzame documentatie; neem een kijkje in de documentatie van `git-completion.bash` en `git-prompt.sh` voor meer informatie.

## Git in Zsh

Git wordt ook geleverd met een library voor het voltooien van commando's met tab voor Zsh. Om het te gebruiken, roep je gewoon `autoload -Uz compinit && compinit` aan in je `.zshrc`. De interface van zsh is een stukje krachtiger dan die van Bash:

```
$ git che<tab>
check-attr      -- laat gitattributes informatie zien
check-ref-format -- controleer dat een referentie naam goed is samengesteld
checkout        -- checkout branch of pad naar de working tree
checkout-index   -- kopieer bestanden van de index naar working directory
cherry          -- vind commits die nog niet stroomopwaarts zijn gemerged
cherry-pick     -- pas wijzigingen toe die door enkele bestaande commits zijn
geïntroduceerd
```

Tab-voltooingen die ambigu zijn worden niet alleen getoond; ze hebben behulpzame omschrijvingen, en je kunt de lijst grafisch navigeren door herhaaldelijk tab in te drukken. Dit werkt met Git commando's, hun argumenten, en namen van zaken die zich in de repository bevinden (zoals refs en remotes), zowel als bestandsnamen en alle andere zaken waarvan Zsh weet hoe deze met de tab te voltooien.

Zsh wordt geleverd met een framework om informatie op te halen van een versie beheer systeem, genaamd `vcs_info`. Om de branchnaam rechts te tonen, voeg je deze regels toe aan je `~/.zshrc` bestand:

```
autoload -Uz vcs_info
precmd_vcs_info() { vcs_info }
precmd_functions+=( precmd_vcs_info )
setopt prompt_subst
RPROMPT=\$vcs_info_msg_0_
# PROMPT=\$vcs_info_msg_0_%# '
zstyle ':vcs_info:git:' formats '%b'
```

Dit resulteert in het tonen van de huidige branch aan de rechterkant van de terminal, zodra je shell in een Git repository staat. (De linkerkant wordt ook ondersteund, vanzelfsprekend; gewoon de toewijzing naar PROMPT ontcommentarieren.) Het ziet er ongeveer zo uit:

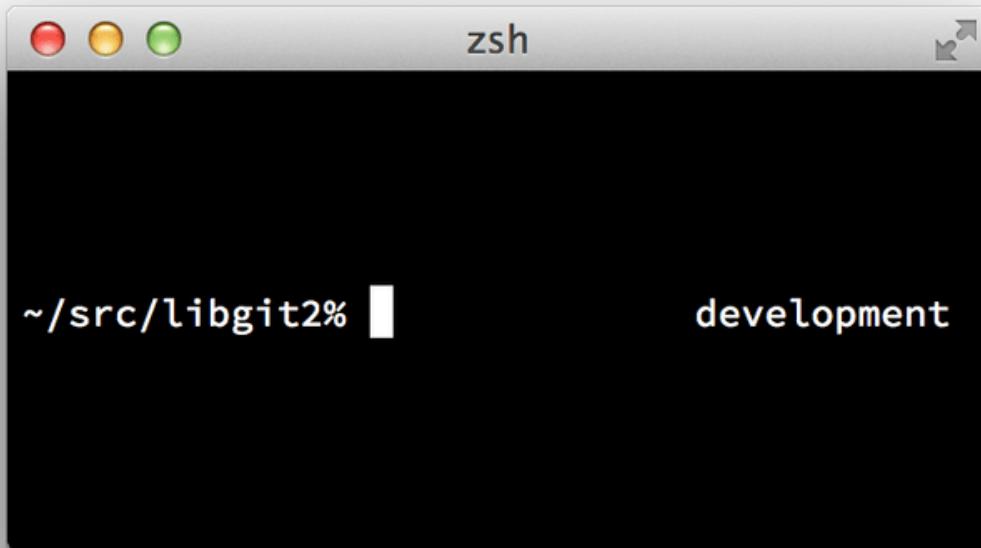


Figure 163. Aangepaste `zsh` prompt.

Voor meer informatie over vcs\_info, kan je de documentatie bekijken in de `zshcontrib(1)` man-page, of online op <http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information>.

In plaats van vcs\_info, heb je misschien voorkeur voor de prompt-aanpassingsscript dat met Git wordt geleverd, deze heet `git-prompt.sh`; zie <https://github.com/git/git/blob/master/contrib/completion/git-prompt.sh> voor details. `git-prompt.sh` is compatible met Bash en Zsh.

Zsh is krachtig genoeg dat er complete frameworks aan zijn gewijd om het beter te maken. Een van deze heet "oh-my-zsh", en deze staat op <https://github.com/robbyrussell/oh-my-zsh>. In het plugin systeem van oh-my-zsh zit een krachtige git tab-voltooiing, en het heeft een rijke verzameling prompt "themes", en vele daarvan tonen versie-beheer gegevens. [Een voorbeeld van een oh-my-zsh thema](#) is maar een voorbeeld van wat gedaan kan worden met dit systeem.

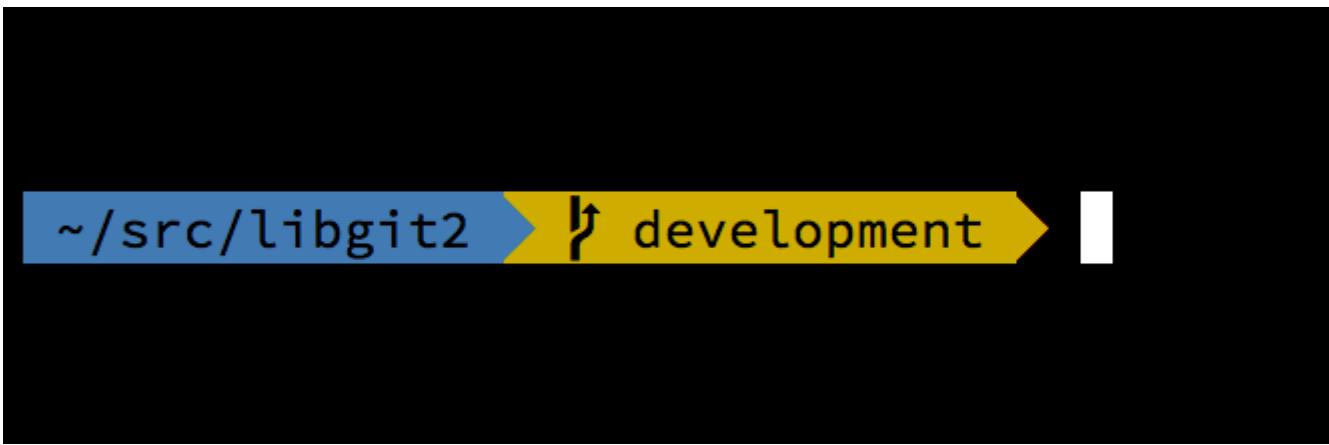


Figure 164. Een voorbeeld van een oh-my-zsh thema.

## Git in Powershell

De standaard commando regel terminal in Windows (`cmd.exe`) is niet echt in staat om een aangepaste Git belevening te ondersteunen, maar als je Powershell gebruikt heb je geluk. Dit werkt ook als je PowerShell op een niet-Windows platform zoals Debian werkt. Een pakket met de naam Posh-Git (<https://github.com/dahlbyk/posh-git>) levert krachtige tab-voltooings functionaliteit, zowel als een uitgebreide prompt om je te helpen bij het zeer nauwlettend volgen van je repository status. Het ziet er zo uit:

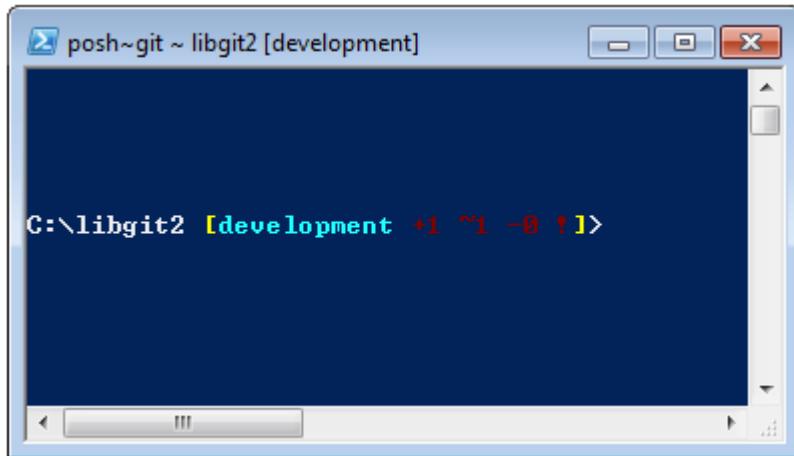


Figure 165. Powershell met Posh-git.

## Installatie

### Voorwaarden (alleen Windows)

Voordat je PowerShell scripts op je machine kunt draaien, moet je je locale ExecutionPolicy op RemoteSigned zetten (Eigenlijk alles behalve Undefined en Restricted). Als je AllSigned kiest in plaats van RemoteSigned, moeten ook lokale scripts (je eigen) digitaal ondertekend worden om te kunnen worden uitgevoerd. Met RemoteSigned moeten alleen Scripts waarvan de "ZoneIdentifier" op Internetgezet is (gedownload vanaf het net) getekend worden, andere niet. Als je een administrator bent, en dit voor alle gebruikers op die machine installen, gebruik dan "-Scope LocalMachine". Als je een normale gebruiker bent, zonder administratieve rechten, kan je "-Scope CurrentUser" gebruiken om het alleen voor jou in te stellen. Meer over PowerShell Scopes:

(<https://technet.microsoft.com/de-de/library/hh847849.aspx>) Meer over PowerShell ExecutionPolicy: (<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy>)

```
> Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy RemoteSigned -Force
```

## PowerShell Gallery

Als je minstens PowerShell 5 of PowerShell 4 met PackageManagement hebt geïnstalleerd, kan je de package manager gebruiken om Posh-Git voor je op te halen. Meer informatie over de benodigdheden: ([https://docs.microsoft.com/en-us/powershell/gallery/psget/get\\_psget\\_module](https://docs.microsoft.com/en-us/powershell/gallery/psget/get_psget_module))

```
> Set-PSRepository -Name PSGallery -InstallationPolicy Trusted  
> Update-Module PowerShellGet -Force  
> Install-Module Posh-Git -Scope AllUsers
```

Als je Posh-Git alleen voor de huidige gebruiker wilt installeren en niet voor iedereen, gebruik dan "-Scope CurrentUser". Als het tweede commando faalt met een fout als **Modules 'PowerShellGet' was not installed by using Install-Module**, moet je eerst een ander commando aanroepen:

```
> Install-Module PowerShellGet -Force -SkipPublisherCheck
```

Dan kan je weer teruggaan en nog een keer proberen. Dit gebeurt, omdat de modules die met Windows PowerShell worden verscheept te gekend zijn met een ander publicatie certificaat.

## Update PowerShell Prompt

Om git informatie in je prompt te laten zien, moet posh-git worden geimporteerd. Om dit automatisch te laten gebeuren moet het import statement in je \$profile script worden opgenomen. Dit script wordt elke keer als je een nieuw PowerShell prompt opent aangeroepen. Onthoud, dat er meerdere \$profile scripts zijn. Bijv. een voor het console en een aparte voor de ISE.

```
> New-Item -Name $($Split-Path -Path $profile) -ItemType Directory -Force  
> 'Import-Module Posh-Git' | Out-File -Append -Encoding default -FilePath $profile
```

## Van broncode

Gewoon een Poshh-Git versie downloaden van (<https://github.com/dahlbyk/posh-git>), en het uitpakken in de **WindowsPowerShell** directory. Dan een Powershell prompt openen als administrator, en dit uitvoeren:

```
> cd ~\Documents\WindowsPowerShell\Module\posh-git  
> .\install.ps1
```

Dit zal de juiste regel toevoegen aan je `profile.ps1` bestand en posh-git wordt actief de volgende keer dat je je prompt opent.

## Samenvatting

Je hebt gezien hoe de kracht van Git kan worden aangewend vanuit de gereedschappen die je in je dagelijkse werkzaamheden gebruikt, en ook hoe je de Git repositories kunt benaderen vanuit je eigen programmatuur.

# Appendix B: Git in je applicaties inbouwen

Als je applicatie bestemd is voor ontwikkelaars, is er een grote kans dat het kan profiteren van integratie met versiebeheer. Zelfs applicaties voor niet-ontwikkelaars, zoals document editors, kunnen potentieel profiteren van versiebeheer functionaliteit, en het model van Git werkt erg goed voor vele verschillende scenarios.

Als je Git moet integreren met je applicatie, heb je eigenlijk twee opties: een shell openen en de Git commando-regel tool gebruiken, of Git als library inbedden in je applicatie. We zullen hier de integratie van de commandoregel en een aantal van de meest populaire inbedbare Git libraries behandelen.

## Commando-regel Git

Een optie is om een shell proces op te starten (spawning) en de Git commando-regel tool gebruiken om het werk te doen. Dit heeft het voordeel van canoniek zijn, en alle mogelijkheden van Git worden ondersteund. Het is toevallig ook nog eens redelijk eenvoudig, omdat de meeste omgevingen waarin het programma draait een relatief eenvoudige manier hebben om een proces met commando-regel argumenten te aan te roepen. Echter, deze aanpak heeft ook een aantal nadelen.

Een ervan is dat alle uitvoer in platte tekst is. Dit houdt in dat je de soms gewijzigde uitvoer van Git moet parsen om de voortgang en resultaat informatie te lezen, wat inefficiënt en foutgevoelig kan zijn.

Een ander is een gebrekkige fout herstel. Als een repository op de een of andere manier corrupt is geraakt, of de gebruiker heeft een configuratiewaarde misvormd, zal Git simpelweg weigeren om een groot aantal operaties uit te voeren.

Weer een ander is proces beheer. Git eist dat je een shell omgeving bijhoudt in een separaat proces, wat ongewilde complexiteit kan toevoegen. Het coördineren van veel van dit soort processen (vooral als de kans bestaat dat dezelfde repository door een aantal processen wordt benaderd) kan een nogal grote uitdaging zijn.

## Libgit2

Een andere optie die je hebt is om Libgit2 te gebruiken. Libgit2 is een implementatie van Git die nergens van afhankelijk is, met een focus op het behouden van een goede API om vanuit andere programma's te gebruiken. Je kunt het vinden op <http://libgit2.github.com>.

Laten we allereerst eens kijken hoe de C API eruit ziet. Hier is een razendsnel overzicht:

```

// Open a repository
git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);

```

De eerste paar regels openen een Git repository. Het `git_repository` type vertegenwoordigt een handvat (handle) naar een repository met een buffer in het geheugen. Dit is de eenvoudigste manier, voor als je het exacte pad naar de werk directory of `.git` folder weet. Er is ook `git_repository_open_ext` die opties voor zoeken in zich heeft, `git_clone` en z'n vriendjes voor het maken van een lokale kloon van een remote repository, en `git_repository_init` voor het maken van een hele nieuwe repository.

Het tweede stuk code gebruikt rev-parse syntax (zie [Branch referenties](#) voor meer details) om de commit te krijgen waar HEAD uiteindelijk naar wijst. Het type dat je terugkrijgt is een `git_object` pointer, wat staat voor iets dat bestaat in de Git object database voor een repository. `git_object` is eigenlijk een “parent” type voor een aantal verschillende soorten objecten; de geheugenindeling voor elk van de “child” typen is hetzelfde als voor `git_object`, dus je kunt het veilig naar de juiste omzetten (casten). In dit geval, zou `git_object_type(commit)` `GIT_OBJ_COMMIT` teruggeven, dus het is veilig te casten naar een `git_commit` pointer.

Het volgende stuk laat zien hoe de kenmerken van een commit kunnen worden benaderd. De laatste regel hier gebruikt een `git_oid` type; dit is hoe Libgit2 een SHA-1 hash represeneert.

Uit dit voorbeeld beginnen een aantal patronen naar boven te komen:

- Als je een pointer declareert en een referentie hieraan doorgeeft met een Libgit2 aanroep, zal die aanroep waarschijnlijk een integer foutcode teruggeven. Een `0` waarde geeft een succes aan; al het andere is een fout.
- Als Libgit2 een pointer voor je vult, ben jij verantwoordelijk deze weer vrij te geven.
- Als Libgit2 een `const` pointer teruggeeft van een aanroep, hoef je deze niet vrij te geven, maar het wordt ongeldig als het object waar het toe behoort vrij wordt gegeven.
- In C schrijven is nogal uitdagend.

Dat laatste houdt in dat het niet erg waarschijnlijk is dat je in C gaat schrijven als je Libgit2

gebruikt. Gelukkig is er een aantal taal-specifieke "bindings" beschikbaar die het redelijk eenvoudig maken om vanuit jouw specifieke taal en omgeving met Git repositories te werken. Laten we naar het bovenstaande voorbeeld kijken die we in Ruby schrijven met gebruikmaking van de bindings voor deze taal die Rugged heet, en gevonden kan worden op <https://github.com/libgit2/rugged>.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"  
tree = commit.tree
```

Zoals je kunt zien, is de code een stuk schoner. Ten eerste, Rugged gebruikt exceptions; het kan dingen gooien als **ConfigError** of `ObjectError` om fout situaties aan te geven. Ten tweede, er is geen expliciete vrij geven van middelen, omdat Ruby een garbage-collector kent. Laten we een kijkje nemen naar een iets ingewikkelder voorbeeld: vanaf het begin een commit samenstellen

```
blob_id = repo.write("Blob contents", :blob) ①

index = repo.index
index.read_tree(repo.head.target.tree)
index.add(:path => 'newfile.txt', :oid => blob_id) ②

sig = {
  :email => "bob@example.com",
  :name => "Bob User",
  :time => Time.now,
}

commit_id = Rugged::Commit.create(repo,
  :tree => index.write_tree(repo), ③
  :author => sig,
  :committer => sig, ④
  :message => "Add newfile.txt", ⑤
  :parents => repo.empty? ? [] : [repo.head.target].compact, ⑥
  :update_ref => 'HEAD', ⑦
)
commit = repo.lookup(commit_id) ⑧
```

- ① Maak een nieuwe blob, die de inhoud van een nieuw bestand bevat.
- ② Vul de index met tree van de commit van de head, en voeg het nieuwe bestand toe aan het pad **newfile.txt**.
- ③ Dit maakt een nieuwe tree aan in de ODB, en gebruikt deze voor de nieuwe commit.
- ④ We gebruiken hetzelfde kenmerk voor de velden van zowel de auteur en de committer.
- ⑤ Het commit bericht.
- ⑥ Als je een commit maakt, moet je de ouders van de nieuwe commit opgeven. Dit gebruikt de punt van HEAD voor de enige ouder.

- ⑦ Rugged (en Libgit2) kan optioneel een referentie updaten als er een commit wordt gemaakt.
- ⑧ De retourwaarde is de SHA-1 has van een nieuw commit object, die je dan weer kan gebruiken om een **Commit** object te krijgen.

De code in Ruby is mooi en schoon, maar omdat Libgit2 het zware werk doet, zal deze code ook redelijk snel draaien. Als je niet zo'n rubyist bent, we behandelen nog een aantal andere bindings in [Andere bindings](#).

## Functionaliteit voor gevorderden

Libgit2 heeft een aantal mogelijkheden die buiten het bereik liggen van de kern van Git. Een voorbeeld is de inhaakmogelijkheid (pluggability): Libgit2 geeft je de mogelijkheid om eigengemaakte "backends" voor een aantal type operaties te verzorgen, zodat je dingen anders kunt opslaan dan Git standaard doet. Libgit2 staat, onder andere, eigengemaakte backends toe voor configuratie, opslag van refs en de object database.

Laten we eens kijken hoe dit in elkaar zit. De onderstaande code is geleend van de set van backend voorbeelden die door het Libgit2 team wordt geleverd (deze kunnen worden gevonden op <https://github.com/libgit2/libgit2-backends>). Hier zie je hoe een eigengemaakte backend voor de objectdatabase wordt opgezet:

```
git_odb *odb;
int error = git_odb_new(&odb); ①

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*...*/); ②

error = git_odb_add_backend(odb, my_backend, 1); ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(repo); ④
```

(Merk op dat fouten worden opgevangen, maar niet afgehandeld. We hopen dat jouw code beter is dan de onze.)

- ① Initialiseer een lege object database (ODB) "frontend", die als een bevatter dient voor de "backends" die het echte werk zullen uitvoeren.
- ② Initialiseer een eigengemaakte ODB backend.
- ③ Voeg de backend toe aan de frontend.
- ④ Open een repository, en stel het in om onze ODB te gebruiken om objecten op te zoeken.

Maar wat is dat `git_odb_backend_mine` voor een ding? Nou, dat is de *constructor* voor je eigen ODB implementatie, en je kunt daarin doen wat je wilt, zolang als je de `git_odb_backend` structuur maar juist vult. Hier is hoe het eruit zou kunnen zien:

```

typedef struct {
    git_odb_backend parent;

    // Some other stuff
    void *custom_context;
} my_backend_struct;

int git_odb_backend_mine(git_odb_backend **backend_out, /*...*/)
{
    my_backend_struct *backend;

    backend = calloc(1, sizeof (my_backend_struct));

    backend->custom_context = ...;

    backend->parent.read = &my_backend__read;
    backend->parent.read_prefix = &my_backend__read_prefix;
    backend->parent.read_header = &my_backend__read_header;
    // ...

    *backend_out = (git_odb_backend *) backend;

    return GIT_SUCCESS;
}

```

Een heel subtile beperking hier is dat de eerste member van `my_backend_struct` een `git_odb_backend` structure moet zijn; dit zorgt ervoor dat de geheugenindeling dat is wat de Libgit2 code ervan verwacht. De rest is vrij in te vullen; deze structuur kan zo groot of zo klein zijn als je het nodig vindt.

De initialisatie functie alloceert wat geheugen voor de structure, richt de eigen context in en vult daarna de de members van de `parent` structure die het ondersteunt. Neem een kijkje in het `include/git2/sys/odb_backend.h` bestand in de Libgit2 broncode voor een volledige set van aanroep-signatures; jou specifieke use-case helpt je bepalen welke van deze je wilt ondersteunen.

## Andere bindings

Libgit2 heeft bindings voor vele talen. Hier laten we je een klein voorbeeld waarbij we een aantal van de meer complete binding pakketten op het moment van schrijven; er bestaan libraries voor vele andere talen, waaronder C++, Go, Node.js, Erlang, en de JVM, alle in verschillende stadia van volwassenheid. De officiële verzameling van bindings kan je vinden door in de repositories te zoeken op <https://github.com/libgit2>. De code die we zullen schrijven gaat het commit bericht teruggeven van de commit waar de HEAD uiteindelijk naar toe wijst (vergelijkbaar met `git log -1`).

### LibGit2Sharp

Als je een applicatie in .NET of Mono schrijft, is LigGit2Sharp (<https://github.com/libgit2/libgit2sharp>) waar je naar op zoek bent. De bindings zijn geschreven in C#, and er is veel zorg besteed aan het inpakken van de ruwe Libgit2 calls met CLR APIs die natuurlijk aanvoelen. Dit is

hoe ons voorbeeld programma eruit ziet:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

Voor Windows werkblad-applicaties is er zelfs een NuGet pakket dat je helpt snel op gang te komen.

### objective-git

Als je applicatie op een Apple platform draait, gebruik je waarschijnlijk Objective-C als je implementatie taal. Objective-Git (<https://github.com/libgit2/objective-git>) is de naam van de Libgit2 bindings voor die omgeving. Ons voorbeeld programma ziet er zo uit:

```
GTRpository *repo =  
    [[GTRpository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"]  
error:NULL];  
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git kan naadloos samenwerken met Swift, dus je hoeft er niet bang voor te zijn als je Objective-C hebt verlaten.

### pygit2

De bindings voor Libgit2 in Python heten Pygit2, en kunnen worden gevonden op <http://www.pygit2.org/>. Ons voorbeeld programma:

```
pygit2.Repository("/path/to/repo") # open repository  
    .head                      # get the current branch  
    .peel(pygit2.Commit)        # walk down to the commit  
    .message                   # read the message
```

## Meer lezen

Een volledige behandeling van de mogelijkheden van Libgit2 ligt natuurlijk buiten het bestek van dit boek. Als je meer informatie wilt over Libgit2 zelf, kan je de API documentatie vinden op <https://libgit2.github.com/libgit2>, en een aantal handboeken op <https://libgit2.github.com/docs>. Voor de andere bindings, kan je de meegeleverde README en testen bekijken; je kunt er vaak kleine tutorials en hints naar meer informatie vinden.

## JGit

Als je Git wilt gebruiken vanuit een Java programma, is er een Git library, Git geheten, die de volledige Git functionaliteit ondersteunt. JGit is een functioneel relatief volledige implementatie van Git geschreven in zuiver Java, en het wordt veel gebruikt in de Java gemeenschap. Het JGit project valt onder de Eclipse paraplu, en het adres waar het kan worden gevonden is <http://www.eclipse.org/jgit>.

## De boel opzetten

Er zijn een aantal manieren om verbinding te maken met je project via JGit, en om er tegenaan te programmeren. Waarschijnlijk is de eenvoudigste manier om Maven te gebruiken - de integratie wordt bereikt door het volgende fragment aan de `<dependencies>` tag in je pom.xml bestand toe te voegen:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

De `versie` zal hoogstwaarschijnlijk al vooruit zijn gegaan tegen de tijd dat je dit leest; kijk even op <http://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit> voor bijgewerkte repository informatie. Als deze stap eenmaal is voltooid, zal Maven automatisch de JGit libraries ophalen en die gebruiken die je nodig zult hebben.

Als je je binaire libraries liever zelf onderhoudt, zijn kant en klaar gebouwde JGit binairies beschikbaar op <http://www.eclipse.org/jgit/download>. Je kunt ze in je project inbouwen door een commando als deze aan te roepen:

```
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
```

## Binnenwerk

JGit heeft twee basale niveaus van API: binnenwerk en koetswerk (plumbing en porcelain). De terminologie hiervoor komt van Git zelf, en JGit is ingedeeld in grofweg dezelfde soorten van gebieden: porcelain API's zijn een vriendelijke voorkant voor reguliere gebruikers acties (het soort van dingen die een gewone gebruiker een commando regel voor zou gebruiken), terwijl de plumbing API's er zijn voor interacties met repository objecten die zich diep in het systeem bevinden.

Het vertrekpunt voor de meeste JGit sessies is de `Repository` klasse, en het eerste wat je zult willen doen is om er een instantie van te maken. Voor een repository die gebaseerd is op een bestandssysteem (ja, JGit ondersteunt ook andere opslagmodellen), wordt dit gedaan door middel van `FileRepositoryBuilder`:

```

// Create a new repository
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Open an existing repository
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();

```

Deze builder heeft een vloeiende API waarmee alles kan worden voorzien die het nodig heeft om een Git repository te vinden, of je programma de preciese locatie weet of niet. Het kan de omgevingsvariabelen (`.readEnvironment()`) gebruiken, ergens van een plaats in de werk directory beginnen te zoeken (`.setWorkTree(...).findGitDir()`), of gewoon een bekende `.git` directory openen zoals hierboven.

Als je eenmaal een `Repository` instantie hebt, kan je er van allerlei dingen mee doen. Hier is een kleine bloemlezing:

```

// Get a reference
Ref master = repo.getRef("master");

// Get the object the reference points to
ObjectId masterTip = master.getObjectId();

// Rev-parse
ObjectId obj = repo.resolve("HEAD^{tree}");

// Load raw object contents
ObjectLoader loader = repo.open(masterTip);
loader.copyTo(System.out);

// Create a branch
RefUpdate createBranch1 = repo.updateRef("refs/heads/branch1");
createBranch1.setNewObjectId(masterTip);
createBranch1.update();

// Delete a branch
RefUpdate deleteBranch1 = repo.updateRef("refs/heads/branch1");
deleteBranch1.setForceUpdate(true);
deleteBranch1.delete();

// Config
Config cfg = repo.getConfig();
String name = cfg.getString("user", null, "name");

```

Er gebeurt hier eigenlijk best wel veel, dus laten we het stukje bij beetje doornemen.

De eerste regel haalt een pointer naar de `master`-referentie op. JGit pakt automatisch de *echte* master ref op, die gedefinieerd is op `refs/heads/master`, en geeft een object terug waarmee je de informatie over de referentie kunt ophalen. Je kunt de naam te pakken krijgen (`.getName()`), en het doel object van een directe referentie (`.getObjectId()`) of de referentie waar een symbolische ref naar wijst (`.getTarget()`). Ref objecten worden ook gebruikt om tag refs en objecten te vertegenwoordigen, dus je kunt vragen of de tag is “geschild” (“peeled”), wat inhoud dat het wijst naar het uiteindelijke doel van een (potentieel lange) tekenreeks van tag objecten.

De tweede regel haalt het doel van de `master`-referentie op, die wordt teruggegeven als een `ObjectId` instantie. `ObjectId` vertegenwoordigt de SHA-1 hash van een object, die al dan niet kan bestaan in de object database van Git. De derde regel is vergelijkbaar, maar laat zien hoe JGit omgaat met de rev-parse syntax (meer hierover in [Branch referenties](#)); je kunt er elke object specificatie aan doorgeven die Git begrijpt, en JGit geeft een geldige `ObjectId` terug voor dat object, of `null`.

De volgende twee regels laten zien hoe de rauwe inhoud van een object wordt ingeladen. In dit voorbeeld, roepen we `ObjectLoader.copyTo()` aan om de inhoud van het object direct naar `stdout` door te geven, maar `ObjectLoader` heeft ook methoden om het type, de grootte van het object te lezen, zowel als om de inhoud als een byte array terug te geven. Voor grote objecten (waar `.isLarge()` de waarde `true` teruggeeft), kan je `.openStream()` aanroepen om een `InputStream`-achtig object terug te krijgen die de data van de rauwe kan lezen zonder het eerst in zijn geheugen te lezen.

De volgende paar regels laten zien wat er voor nodig is om een nieuwe branch te maken. We maken een `RefUpdate` instantie, configureren een paar parameters, en roepen `.update()` aan om de wijziging af te trappen. Direct hierna is de code om dezelfde branch te verwijderen. Merk op dat `.setForceUpdate(true)` nodig is om dit te laten werken; anders zal de aanroep naar `.delete()` de waarde `REJECTED` teruggeven, en er gebeurt helemaal niets.

Het laatste voorbeeld laat zien hoe je de waarde `user.name` uit de Git configuratie bestanden kan ophalen. Deze instantie van `Config` gebruikt de repository die we eerder geopend hebben voor de lokale configuratie, maar pakt automatisch de wijzigingen op in de globale en systeem configuratie bestanden en leest daar ook waarden uit.

Dit is maar een klein voorproefje van de volledige binnenwerk API; er zijn nog veel meer methoden en klassen beschikbaar. Wat we hier ook niet hebben laten zien is de manier waarop JGit fouten behandelt, namelijk door middel van exceptions. De API's van JGit gooien soms standaard Java exceptions (zoals `IOException`), maar er is een heel scala aan JGit-specifieke exceptie typen die ook beschikbaar zijn (zoals `NoRemoteRepositoryException`, `CorruptObjectException`, en `NoMergeBaseException`).

## Porcelein

De binnenwerk API's zijn nogal compleet, maar het kan nogal bewerkelijk zijn om ze aan elkaar te knopen om een regulier doel te bereiken, zoals het toevoegen van een bestand aan de index, of een nieuwe commit maken. JGit levert een aantal API's op een hoger niveau om je hiermee te helpen, en het beginpunt van deze API's is de `Git`-klasse:

```
Repository repo;  
// construct repo...  
Git git = new Git(repo);
```

De Git klasse heeft een leuke verzameling hoog-niveau methoden in de *builder*-stijl die kunnen worden gebruikt om een redelijk ingewikkeld gedrag samen te stellen. Laten we een kijkje nemen naar een voorbeeld - iets doen als `git ls-remote`:

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username",  
"p4ssw0rd");  
Collection<Ref> remoteRefs = git.lsRemote()  
    .setCredentialsProvider(cp)  
    .setRemote("origin")  
    .setTags(true)  
    .setHeads(false)  
    .call();  
for (Ref ref : remoteRefs) {  
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());  
}
```

Dit is een normaal patroon met de Git klasse; de methode geeft een commando object terug waarmee je methode aanroepen aaneen kunt rijgen om parameters te zetten, die worden uitgevoerd als je `.call()` aanroept. In dit geval, vragen we de `origin` remote om tags, maar niet om heads. Merk ook het gebruik van een `CredentialsProvider` object op, voor autenticatie.

Vele andere commando's zijn beschikbaar via de Git klasse, inclusief (maar niet beperkt tot) `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert` en `reset`.

## Meer lezen

Dit is maar een kleine selectie uit de volledige mogelijkheden van JGit. Als je geïnteresseerd bent en meer wilt lezen, kan je hier kijken voor meer informatie en inspiratie:

- De officiële JGit API documentatie is online beschikbaar op <http://download.eclipse.org/jgit/docs/latest/apidocs>. Dit is standaard Javadoc, dus je favoriete JVM IDE zal in staat zijn om ze lokaal te installeren.
- De JGit cookbook op <https://github.com/centic9/jgit-cookbook> heeft veel voorbeelden van hoe de specifieke taken met JGit te doen.
- Er zijn een aantal goede bronnen die genoemd staan op <http://stackoverflow.com/questions/6861881>.

## go-git

In het geval dat je Git in een service die in Golang is geschreven wilt integreren, is er ook een zuivere Go library implementatie. Deze implementatie heeft geen enkele afhankelijkheden met het onderliggende systeem en is dus niet onderhevig aan handmatige geheugenbeheer fouten. Het is

ook nog eens transparant voor de standaard Golang prestatie-analyse gereedschappen zoals CPU, Geheugen profilers, race detectors, etc.

go-git is gefocust op uitbreidbaarheid, compatibiliteit en ondersteund de meeste plumbing APIs, wat is beschreven op <https://github.com/src-d/go-git/blob/master/COMPATIBILITY.md>.

Hier is een eenvoudig voorbeeld van het gebruik van Go APIs:

```
import "gopkg.in/src-d/go-git.v4"

r, err := git.PlainClone("/tmp/foo", false, &git.CloneOptions{
    URL:      "https://github.com/src-d/go-git",
    Progress: os.Stdout,
})
```

Op het moment dat je een **Repository** instantie hebt, kan je de informatie verkrijgen en wijzigingen erop uitvoeren:

```
// retrieves the branch pointed by HEAD
ref, err := r.Head()

// get the commit object, pointed by ref
commit, err := r.CommitObject(ref.Hash())

// retrieves the commit history
history, err := commit.History()

// iterates over the commits and print each
for _, c := range history {
    fmt.Println(c)
}
```

## Gevorderde functionaliteit

go-git heeft een aantal opmerkelijke mogelijkheden, een ervan is een uitbreidbaar opslag systeem, die vergelijkbaar is met Libgit2 backends. De standaard implementatie is in-memory opslag, wat heel snel is.

```
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{
    URL: "https://github.com/src-d/go-git",
})
```

Uitbreidbare opslag biedt veel interessante opties. Bijvoorbeeld, [https://github.com/src-d/go-git/tree/master/\\_examples/storage](https://github.com/src-d/go-git/tree/master/_examples/storage) geeft je de mogelijkheid om referenties, objecten en configuratie in een Aerospike database op te slaan.

Een andere mogelijkheid is een abstractie van een flexibel bestandssysteem. Met <https://godoc.org/>

[github.com/src-d/go-billy#Filesystem](https://github.com/src-d/go-billy#Filesystem) is het eenvoudig om alle bestanden op een andere manier op te slaan, bijv. door ze allemaal in een enkele archief op te slaan of door ze allemaal in het geheugen te houden.

Een andere geavanceerd gebruiksscenario is een erg optimaliseerbare HTTP client, zoals een die gevonden kan worden op [https://github.com/src-d/go-git/blob/master/\\_examples/custom\\_http/main.go](https://github.com/src-d/go-git/blob/master/_examples/custom_http/main.go).

```
customClient := &http.Client{
    Transport: &http.Transport{ // accept any certificate (might be useful for
        testing)
        TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
    },
    Timeout: 15 * time.Second, // 15 second timeout
    CheckRedirect: func(req *http.Request, via []*http.Request) error {
        return http.ErrUseLastResponse // don't follow redirect
    },
}

// Override http(s) default protocol to use our custom client
client.InstallProtocol("https", githttp.NewClient(customClient))

// Clone repository using the new client if the protocol is https://
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{URL: url})
```

## Meer lezen

Een volledige behandeling van de mogelijkheden van go-git ligt buiten het bestek van dit boek. Als je meer informatie wilt hebben over go-git, dan is API documentatie beschikbaar op <https://godoc.org/gopkg.in/src-d/go-git.v4>, en een aantal gebruiksvoorbeelden op [https://github.com/src-d/go-git/tree/master/\\_examples](https://github.com/src-d/go-git/tree/master/_examples).

## Dulwich

Er is ook een zuivere Python Git implementatie - Dulwich. Het project wordt gehost onder <https://www.dulwich.io/>. Het doel van het project is om een interface naar git repositories te bieden (zowel lokaal als remote) die geen directe git aanroepen doet, maar daarentegen puur Python gebruikt. Het heeft echter een optionele C-extensie, die de prestaties behoorlijk versnelt.

Dulwich volgt het git ontwerp en maakt een scheiding tussen twee basisniveaus wat APIs betreft: binnenwerk en koetswerk (plumbing en porcelain).

Hier is een voorbeeld hoe een lager niveau API te gebruiken om toegang tot het commit-bericht van de laatste commit te benaderen:

```
from dulwich.repo import Repo
r = Repo('.')
r.head()
# '57fbe010446356833a6ad1600059d80b1e731e15'

c = r[r.head()]
c
# <Commit 015fc1267258458901a94d228e39f0a378370466>

c.message
# 'Add note about encoding.\n'
```

Om een commit-log af te drukken gebruikmakend van de API van het hogere niveau (porcelein), kan men het volgende doen:

```
from dulwich import porcelain
porcelain.log('.', max_entries=1)

#commit: 57fbe010446356833a6ad1600059d80b1e731e15
#Author: Jelmer Vernooij <jelmer@jelmer.uk>
#Date:   Sat Apr 29 2017 23:57:34 +0000
```

## Meer weten

- De officiële API documentatie is beschikbaar op <https://www.dulwich.io/apidocs/dulwich.html>
- De officiële tutorial op <https://www.dulwich.io/docs/tutorial> heeft veel voorbeelden hoe specifieke taken met Dulwich uit te voeren

# Appendix C: Git Commando's

Overal in dit boek hebben we tientallen Git commando's geïntroduceerd en we hebben ons best gedaan om ze te introduceren met een verhaaltje, en langzaamaan meer commando's toe te voegen. Echter, hiermee zijn we geeindigd met een situatie waarbij de voorbeelden van het gebruik van commando's nogal versnipperd zijn geraakt over het hele boek.

In deze appendix zullen we alle Git commando's die we hebben behandeld in dit boek nogmaals doornemen, grofweg gegroepeerd op het gebruik ervan. We zullen globaal bespreken wat elk commando doet en verwijzen naar de plaats in het boek waar je kunt vinden waar we het hebben gebruikt.

## Setup en configuratie

Er zijn twee commando's die heel vaak gebruikt worden, van de eerste aanroepen van Git tot normale dagelijks gebruikte bijstellingen en referenties, de `config` en `help` commando's.

### git config

Git heeft een standaard manier om vele honderden dingen te doen. Voor veel van deze dingen, kan je Git vertellen om deze dingen standaard anders uit te voeren, of je voorkeuren instellen. Dit kan alles omvatten van Git vertellen wat je naam is, tot voorkeuren voor specifieke werkstation kleuren of welke editor je gebruikt. Er zijn verscheidene bestanden die door dit commando gelezen en geschreven worden zodat je deze waarden globaal kunt instellen of specifiek voor specifieke repositories.

Het `git config` commando is in ongeveer elk hoofdstuk van dit boek gebruikt.

In [Git klaarmaken voor eerste gebruik](#) hebben we het gebruikt om onze naam, email adres en editor voorkeuren aan te geven voordat we zelfs waren begonnen met Git te gebruiken.

In [Git aliassen](#) hebben we laten zien hoe je het kon gebruiken om commando-afkortingen te maken die lange optie-reeksen vervangen zodat je ze niet elke keer hoefde in te typen.

In [Rebasen](#) hebben we het gebruikt om `--rebase` de standaard manier te bepalen voor de aanroep van `git pull`.

In [Het opslaan van inloggegevens](#) hebben we het gebruikt om een standaard bewaarplaats in te richten voor je HTTP wachtwoorden.

In [Sleutelwoord expansie \(Keyword expansion\)](#) hebben we laten zien hoe besmeur en opschoon filters op te zetten voor gegevens die Git in en uit gaan.

Tot slot is heel [Git configuratie](#) toegewijd aan dit commando.

### git help

Het `git help` commando wordt gebruikt om je alle documentatie over elk commando te laten zien die geleverd wordt met Git. Hoewel we een grof overzicht van de meer populaire commando's laten

zien in deze appendix, kan je voor een volledige opsomming van alle mogelijke opties en vlaggen voor elk commando altijd `git help <commando>` aanroepen.

We hebben het `git help` commando in [Hulp krijgen](#) geïntroduceerd, en je laten zien hoe het te gebruiken om meer informatie te verkrijgen over de `git shell` in [De server opzetten](#).

## Projecten ophalen en maken

Er zijn twee manieren om een Git repository op te halen. Een manier is om het te kopiëren van een bestaande repository op het netwerk of ergens anders en de andere is om een nieuwe te maken in een bestaande directory.

### git init

Om een directory in een nieuwe Git repository te veranderen zodat je kunt beginnen met versiebeheer, kan je simpelweg `git init` aanroepen.

We hebben dit voor het eerst behandeld in [Een Git repository verkrijgen](#), waar we laten zien hoe een gloednieuwe repository gemaakt wordt om in te werken.

We hebben kort besproken hoe je de standaard branch van “master” kunt wijzigen in [Branches op afstand \(Remote branches\)](#).

We gebruiken dit commando om een lege bare repository te maken voor een server in [De kale repository op een server zetten](#).

Tot slot, behandelen we een aantal details over wat er achter de schermen gebeurt in [Binnenwerk en koetswerk \(plumbing and porcelain\)](#).

### git clone

Het `git clone` commando is eigenlijk een soort wrapper om een aantal andere commando’s. Het maakt een nieuwe directory, gaat daarin en roept `git init` aan om het een lege Git repository te maken, voegt een remote toe (`git remote add`) naar de URL die je het door hebt gegeven (standaard `origin` genaamd), roept een `git fetch` aan van die remote repository en checkt daarna de laatste commit uit naar je werk directory met `git checkout`.

Het `git clone` commando wordt in tientallen plaatsen in het boek gebruikt, maar we zullen een paar interessante plaatsen opnoemen.

Het wordt kort geïntroduceerd en uitgelegd in [Een bestaande repository klonen](#), waar we een aantal voorbeelden behandelen.

In [Git op een server krijgen](#) bekijken we het gebruik van de `--bare` optie om een kopie van een Git repository te maken zonder een werk directory.

In [Bundelen](#) gebruiken we het om een gebundelde Git repository te ontbundelen.

Tenslotte, in [Een project met submodules klonen](#) hebben we de `--recurse-submodules` optie laten zien waarmee het clonen van een repository met submodules iets te vereenvoudigen.

Alhoewel het op vele andere plaatsen in dit boek wordt gebruikt, zijn dit de toepassingen die nogal uniek zijn, of waar het op manieren wordt gebruikt die een beetje afwijkend zijn.

## Basic Snapshotten

Voor de gewone workflow van het staggen van inhoud en dit committen naar je historie, zijn er maar een paar basis commando's.

### git add

Het `git add` commando voegt inhoud van de werk directory toe aan de staging area (of “index”) voor de volgende commit. Als het `git commit` commando wordt aangeroepen, kijkt het standaard alleen naar deze staging area, dus `git add` wordt gebruikt om de samenstelling van de volgende commit precies naar jouw eigen wens te bepalen.

Dit commando is een ongelofelijk belangrijk commando in Git en het wordt tientallen keren genoemd of gebruikt in dit boek. We zullen heel kort een aantal van de meer incourante gebruiken benoemen die kunnen worden gevonden.

We hebben `git add` voor het eerst geïntroduceerd en in detail uitgelegd in [Nieuwe bestanden volgen \(tracking\)](#).

We beschrijven hoe we het moeten gebruiken om merge conflicten op te lossen in [Eenvoudige merge conflicten](#).

We behandelen het gebruik ervan om interactief alleen specifieke delen van een gewijzigd bestand te staggen in [Interactief staggen](#).

Tot slot spelen we het op een laag niveau na in [Boom objecten \(tree objects\)](#), zodat je een idee krijgt van wat het achter de schermen doet.

### git status

Het `git status` commando laat je de verschillende stadia zien van bestanden in je werk directory en staging area. Welke bestanden er zijn gewijzigd en nog niet gestaged en welke er zijn gestaged maar nog niet gecommit. In zijn reguliere vorm, laat het je ook een aantal hints zien over hoe bestanden tussen deze stadia te verplaatsen.

We behandelen `status` voor het eerst in [De status van je bestanden controleren](#), zowel in zijn basis en versimpelde vormen. Alhoewel we het door het hele boek heen gebruiken, wordt vrijwel alles wat je met het `git status` commando kunt doen daar behandeld.

### git diff

Het `git diff` commando wordt gebruikt als je de verschillen wilt zien tussen elke twee trees. Dit kan het verschil zijn tussen je werk omgeving en je staging area (`git diff` op zichzelf), tussen je staging area en je laatste commit (`git diff --staged`), of tussen twee commits (`git diff master branchB`).

We kijken eerst naar de basis gebruiken van `git diff` in [Je staged en unstaged wijzigingen bekijken](#), waar we laten zien hoe je kunt zien welke wijzigingen er zijn gestaged en welke nog niet.

We gebruiken het om te kijken of er mogelijke witruimte-problemen zijn voor we committen met de `--check` optie in [Commit richtlijnen](#).

We zien hoe we de verschillen tussen branches efficiënter kunnen controleren met de `git diff A...B` syntax in [Bepalen wat geïntroduceerd is geworden](#).

We gebruiken het om de witruimte verschillen weg te filteren met `-b` en hoe de verschillende stadia van conflicterende bestanden te vergelijken met `--theirs`, `--ours` en `--base` in [Mergen voor gevorderden](#).

En tot slot, gebruiken we het om efficiënt submodule wijzigingen te vergelijken met `--submodule` in [Beginnen met submodules](#).

## git difftool

Het `git difftool` commando roept simpelweg een externe tool aan om je de verschillen te tonen tussen twee trees in het geval dat je iets anders wilt gebruiken dan het ingebouwde `git diff` commando.

We vermelden dit slechts kort in [Je staged en unstaged wijzigingen bekijken](#).

## git commit

Het `git commit` commando neemt alle bestandsinhoud die zijn gestaged met `git add` en slaat een nieuwe permanente snapshot in de database op en verplaatst erna de branch verwijzer op de huidige branch daar naar toe.

We behandelen eerst de basis van committen in [Je wijzigingen committen](#). Daar laten we ook zien hoe je de `-a` vlag kunt gebruiken om de `git add` stap over te slaan in de dagelijkse workflows en hoe je de `-m` vlag kunt gebruiken om een commit bericht uit de commando regel kunt doorgeven in plaats van een editor te laten opstarten.

In [Dingen ongedaan maken](#) behandelen we het gebruik van de `--amend` optie om de meest recente commit over te doen.

In [Branches in vogelvlucht](#) gaan we met veel meer detail in op wat `git commit` doet en waarom het doet wat het doet.

We kijken naar hoe commits cryptografisch te tekenen met de `-S` vlag in [Commits tekenen](#).

Tot slot, nemen we een kijkje naar wat het `git commit` commando op de achtergrond doet en hoe het echt is geïmplementeerd in [Commit objecten](#).

## git reset

Het `git reset` commando is voornamelijk gebruikt om zaken terug te draaien, zoals je waarschijnlijk al kunt zien aan het werkwoord. Het verplaatst de `HEAD` verwijzing en verandert

optioneel de `index` of staging area en kan optioneel ook de werk directory wijzigen als je `--hard` gebruikt. Deze laatste optie maakt het mogelijk met dit commando je werk te verliezen als je het niet juist gebruikt, dus verzekер je ervan dat je alles goed begrijpt voordat je het gebruikt.

We hebben feitelijk het eenvoudigste gebruik van `git reset` in [Een gestaged bestand unstagen](#) behandeld, waar we het hebben gebruikt om een bestand te unstagen waar we `git add` op hadden gebruikt.

We behandelen het daarna in behoorlijk meer detail in [Reset ontrafeld](#), die volledig is gewijd aan de uitleg van dit commando.

We gebruiken `git reset --hard` om een merge af te breken in [Een merge afbreken](#), waar we ook `git merge --abort` gebruiken, wat een vorm van een wrapper is voor het `git reset` commando.

## git rm

Het `git rm` commando wordt gebruikt om bestanden te verwijderen van de staging area en de werk directory voor Git. Het lijkt op `git add` in die zin dat het de verwijdering van een bestand staged voor de volgende commit.

We behandelen het `git rm` commando in enige detail in [Bestanden verwijderen](#), inclusief het recursief verwijderen van bestanden en alleen bestanden verwijderen van de staging area maar ze in de werk directory ongemoeid te laten met `--cached`.

Het enige andere alternatieve gebruik van `git rm` in het boek is in [Objecten verwijderen](#) waar we het even gebruiken en de `--ignore-unmatch` uitleggen als we `git filter-branch` aanroepen, wat ervoor zorgt dat we niet met een fout eindigen als het bestand dat we proberen te verwijderen niet bestaat. Dit kan nuttig zijn bij het maken van scripts.

## git mv

Het `git mv` commando is een kleine gemaks-commando om een bestand te verplaatsen en dan `git add` aan te roepen op het nieuwe bestand en `git rm` op het oude bestand.

We noemen dit commando heel kort in [Bestanden verplaatsen](#).

## git clean

Het `git clean` commando wordt gebruikt om ongewenste bestanden te verwijderen uit je werk directory. Dit zou het verwijderen van tijdelijke bouw-artefacten kunnen inhouden of merge conflict bestanden.

We hebben veel van de opties en scenario's besproken waar je het clean commando zou kunnen gebruiken in [Je werk directory opruimen](#).

# Branchen en mergen

Er zijn maar een handjevol commando's die de meeste van de branch en merge functionaliteit in Git implementeren.

## git branch

Het `git branch` commando is eigenlijk een soort branch beheer gereedschap. Het kan de branches die je hebt uitlijsten, een nieuwe branch aanmaken, branches verwijderen en hernoemen.

Het grootste gedeelte van [Branchen in Git](#) is gewijd aan het `branch` commando en het wordt door het gehele hoofdstuk gebruikt. We introduceren het eerst in [Een nieuwe branch maken](#) en we behandelen de meeste van de andere mogelijkheden (uitlijsten en verwijderen) in [Branch-beheer](#).

In [Tracking branches](#) gebruiken we de `git branch -u` optie om een tracking branch op te zetten.

Tot slot, behandelen we een aantal dingen die het op de achtergrond doet in [Git Referenties](#).

## git checkout

Het `git checkout` commando wordt gebruikt om branches te wisselen en inhoud uit te cheken in je werk directory.

We komen het commando voor het eerst tegen in [Tussen branches schakelen \(switching\)](#) samen met de `git branch` commando.

We zien hoe het te gebruiken om tracking branches te starten met de `--track` vlag in [Tracking branches](#).

We gebruiken het om bestandsconflicten te herintroduceren met `--conflict=diff3` in [Conflicten beter bekijken](#).

We behandelen het in nog meer detail in verband met haar relatie met `git reset` in [Reset ontrafeld](#).

Tot slot, behandelen we een aantal implementatie details in [De HEAD](#).

## git merge

Het `git merge` tool wordt gebruikt om een of meerdere branches te mergen naar de branch die je hebt uitgechecked. Het zal daarna de huidige branch voortbewegen naar het resultaat van de merge.

Het `git merge` commando werd voor het eerst geïntroduceerd in [Eenvoudig branchen](#). En hoewel het op diverse plaatsen in het boek wordt gebruikt, zijn er over het algemeen erg weinig variaties op het `merge` commando, normaalgesproken alleen `git merge <branch>` met de naam van die ene branch die je wilt inmergen.

We hebben behandeld hoe een squashed merge te doen (waar Git het werk merged maar doet alsof het niet meer dan een nieuwe commit is zonder de historie van de branch die je in merged op te slaan) aan het einde van [Gevorkt openbaar project](#).

We hebben veel behandeld over het merge proces en commando, inclusief het `-Xignore-space-change` commando en de `--abort` vlag om een problematische merge af te breken in [Mergen voor gevorderden](#).

We hebben gezien hoe handtekeningen te verifiëren als je project GPG tekenen gebruikt in

## [Commits tekenen.](#)

Tot slot, hebben we Subtree mergen behandeld in [Het mergen van subtrees](#).

## **git mergetool**

Het `git mergetool` commando roept simpelweg een externe merge hulp aan in het geval dat je problemen hebt met een merge in Git.

We hebben het kort genoemd in [Eenvoudige merge conflicten](#) en behandelen met detail hoe je je eigen externe merge tool kunt implementeren in [Externe merge en diff tools](#).

## **git log**

Het `git log` commando wordt gebruikt om de bereikbare opgeslagen historie van een project te laten zien vanaf de meeste recente commit snapshot en verder terug. Standaard laat het alleen de historie zien van de branch waar je op dat moment op werkt, maar het kan een andere of zelfs meerdere heads of branches worden gegeven vanwaar het door de geschiedenis zal gaan traceren. Het wordt ook vaak gebruikt om verschillen te laten zien tussen twee of meer branches op het commit niveau.

Dit commando wordt in vrijwel elk hoofdstuk van het boek gebruikt om de historie van het project te tonen.

We introduceren het commando en behandelen het met nogal wat detail in [De commit geschiedenis bekijken](#). Daar nemen we een kijkje naar de `-p` en `--stat` opties om een indruk te krijgen van wat er was geïntroduceerd in elke commit en de `--pretty` en `--oneline` opties om de historie meer beknopt te bekijken, samen met wat eenvoudige datum en auteur filter opties.

In [Een nieuwe branch maken](#) gebruiken we het met de `--decorate` optie om eenvoudig te laten zien waar onze branch verwijzingen zijn en we gebruiken ook de `--graph` optie om te zien hoe uiteenlopende histories eruit zien.

In [Besloten klein team](#) en [Commit reeksen](#) behandelen we hoe de `branchA..branchB` syntax toe te passen om het `git log` commando te gebruiken om te bekijken welke commits er uniek zijn voor een branch in vergelijking met een andere branch. In [Commit reeksen](#) behandelen we dit redelijk uitgebreid.

In [Merge log](#) en [Drievoudige punt](#) behandelen we het gebruik van het `branchA...branchB` formaat en de `--left-right` syntax om te zien wat er in een branch zit of de andere, maar niet in beide. In [Merge log](#) kijken we ook naar hoe de `--merge` optie te gebruiken als hulp om een merge conflict te debuggen en ook het gebruik van de `--cc` optie om naar merge conflicten te kijken in je historie.

In [RefLog verkorte namen](#) gebruiken we de `-g` optie om de Git reflog te bekijken via dit gereedschap in plaats van branch navigatie te doen.

In [Zoeken](#) kijken we naar het gebruik van de `-S` en `-L` opties om een redelijk geraffineerde zoekopdrachten uit te voeren naar iets dat historisch heeft plaatsgevonden in de code, zoals het bekijken van de geschiedenis van een functie.

In [Commits tekenen](#) zien we hoe `--show-signature` te gebruiken om een validatie tekenreeks toe te voegen aan elke commit in de `git log` uitvoer afhankelijk van of het valide is getekend of niet.

## git stash

Het `git stash` commando wordt gebruikt om tijdelijk niet-committed werk op te slaan, zodat je werk directory opgeschoond wordt; hierdoor hoef je geen onvolledig werk te committen naar een branch.

Dit wordt eigenlijk in zijn geheel behandeld in [Stashen en opschonen](#).

## git tag

Het `git tag` commando wordt gebruikt om een blijvende boekwijzer te geven aan een specifiek punt in de code historie. Over het algemeen wordt dit gebruikt voor zaken zoals releases.

Dit commando wordt geïntroduceerd en in detail behandeld in [Taggen \(Labelen\)](#) en we gebruiken het in de praktijk in [Je releases taggen](#).

We behandelen ook hoe een GPG getekende tag te maken met de `-s` vlag en verifiëren er een met de `-v` vlag in [Je werk tekenen](#).

# Projecten delen en bijwerken

Er zijn niet veel commando's in Git die het netwerk benaderen, bijna alle commando's werken op de lokale database. Als je er klaar voor bent om je werk te delen of wijzigingen binnen te halen (pull) van elders, zijn er een handjevol commando's die te maken hebben met remote repositories.

## git fetch

Het `git fetch` commando communiceert met een remote repository en haalt alle informatie binnen die in die repository zit die nog niet in je huidige zit en bewaart dit in je lokale database.

We nemen voor het eerst een kijkje naar dit commando in [Van je remotes fetchen en pullen](#) en we vervolgen met het kijken naar voorbeelden van het gebruik in [Branches op afstand \(Remote branches\)](#).

We gebruiken het ook in diverse voorbeelden in [Bijdragen aan een project](#).

We gebruiken het om een enkele specifieke refentie op te halen die buiten de standaard ruimte is in [Pull Request Refs \(Pull Request referenties\)](#) en we zien hoe we iets uit een bundel kunnen halen in [Bundelen](#).

We richten een aantal zeer eigen refspecs in om `git fetch` iets op een net andere manier te laten doen dan standaard in [De Refspec](#).

## git pull

Het `git pull` commando is feitelijk een combinatie van de `git fetch` en `git merge` commando's, waar Git van de remote die je opgeeft gaat ophalen en daarna direct probeert het opgehaalde te

mergen in de branch waar je op dat moment op zit.

We stellen je het kort voor in [Van je remotes fetchen en pullen](#) en laten zien hoe je kunt bekijken wat het zal gaan mergen als je het aanroeft in [Een remote inspecteren](#).

We laten ook zien hoe het te gebruiken als hulp met bij problemen met rebasen in [Rebaset spullen rebasen](#).

We laten zien hoe het te gebruiken met een URL om wijzigingen op een eenmalige manier te pullen in [Remote branches uitchecken](#).

Tot slot, vermelden we heel snel dat je de `--verify-signature` optie kunt gebruiken om te verifiëren dat commits die je binnenhaalt met GPG getekend zijn in [Commits tekenen](#).

## git push

Het `git push` commando wordt gebruikt om te communiceren met een andere repository, berekenen wat je lokale database heeft en de remote niet, en daarna de verschillen te pushen naar de andere repository. Het vereist wel dat je schrijfrechten hebt op de andere repository en normaalgesproken is dit dus op de een of andere manier geautenticeerd.

We nemen een eerste kijkje naar het `git push` commando in [Naar je remotes pushen](#). Hier behandelen we de basisprincipes van het pushen van een branch naar een remote repository. In [Pushen](#) gaan we iets dieper in op het pushen van specifieke branches en in [Tracking branches](#) zien we hoe tracking branches worden opgezet om automatisch naar te pushen. In [Remote branches verwijderen](#) gebruiken we de `--delete` vlag om een branch te verwijderen op de server met `git push`.

Door heel [Bijdragen aan een project](#) heen zien we een aantal voorbeelden van het gebruik van `git push` om werk op branches te delen middels meerdere remotes.

We zien hoe we het gebruiken om tags te delen die je gemaakt hebt met de `--tags` optie in [Tags delen](#).

In [Submodule wijzigingen publiceren](#) gebruiken we de `--recurse-submodules` optie om te controleren dat al onze submodule werk is gepubliceerd voordat we het superproject pushen, wat zeer behulpzaam kan zijn als je submodules gebruikt.

In [Overige werkstation hooks](#) bespreken we kort de `pre-push` hook, wat een script is die we kunnen opzetten om te draaien voordat een push voltooid is, om te verifiëren dat pushen hiervan toegestaan is.

Tot slot, in [Refspecs pushen](#) kijken we naar pushen met een volledige refspect in plaats van de generieke afkortingen die we normaalgesproken gebruiken. Dit kan je helpen om heel specifiek te zijn over welk werk je wilt delen.

## git remote

Het `git remote` commando is een beheertool voor jouw administratie van remote repositories. Het stelt je in staat om lange URLs op te slaan als afkortingen, zoals "origin" zodat je ze niet elke keer

helemaal hoeft in te typen. Je kunt er een aantal van hebben en het `git remote` commando wordt gebruikt om ze toe te voegen, te wijzigen en te verwijderen.

Dit commando wordt in detail behandeld [Werken met remotes](#), inclusief het uitlijsten, toevoegen, verwijderen en het hernoemen.

Het wordt daarnaast in bijna elk daaropvolgend hoofdstuk van het boek gebruikt, maar altijd in de standaard `git remote add <naam> <url>` formaat.

## git archive

Het `git archive` commando wordt gebruikt om een archiefbestand te maken van een specifieke snapshot van het project.

We gebruiken `git archive` om een tarball te maken van een project om het te delen in [Een release voorbereiden](#).

## git submodule

Het `git submodule` commando wordt gebruikt om externe repositories te beheren binnen normale repositories. Dit kan zijn voor libraries of andere typeren gedeelde hulpmiddelen. Het `submodule` commando heeft een aantal sub-commando's (`add`, `update`, `sync`, etc) om deze hulpmiddelen te beheren.

Dit commando wordt alleen benoemd en in zijn volledigheid behandeld in [Submodules](#).

# Inspectie en vergelijking

## git show

Het `git show` commando kan een Git object in een eenvoudige en voor een mens leesbare manier laten zien. Normaalgesproken gebruik je dit om de informatie over een tag of een commit te laten zien.

We gebruiken het eerst om informatie van een geannoteerde tag te laten zien in [Annotated tags](#).

Later gebruiken we het redelijk vaak in [Revisie Selectie](#) om de commits te tonen waar verscheidene van onze revisie selecties naar toe worden vertaald.

Een van de meer interessante dingen die we met `git show` doen is in [Handmatig bestanden opnieuw mergen](#) waar we specifieke bestandsinhoud ophalen uit verschillende stadia gedurende een merge conflict.

## git shortlog

Het `git shortlog` commando wordt gebruikt om de uitvoer van `git log` samen te vatten. Het accepteert veel van dezelfde opties als het `git log` commando maar zal, in plaats van alle commits uit te lijsten, een samenvatting presenteren van de commits gegroepeerd per auteur.

We hebben laten zien hoe het te gebruiken om een mooie changelog te maken in [De shortlog](#).

## git describe

Het `git describe` commando wordt gebruikt om alles te pakken wat naar een commit kan leiden en produceert een tekenreeks die redelijk mens leesbaar is en niet zal veranderen. Het is een manier om een omschrijving van een commit te krijgen die net zo eenduidig is als een SHA-1, maar meer begrijpelijk.

We gebruiken `git describe` in [Een bouw nummer genereren](#) en [Een release voorbereiden](#) om een tekenreeks te verkrijgen om onze release bestand naar te vernoemen.

## Debuggen

Git heeft een aantal commando's die als hulp worden gebruikt bij het debuggen van een probleem in je code. Dit varieert van uitknobben waar iets is geïntroduceerd tot wie het heeft geïntroduceerd.

### git bisect

Het `git bisect` gereedschap is een ongelofelijk behulpzaam debugging tool die gebruikt wordt om uit te vinden welke specifieke commit de eerste was om een bug of probleem te bevatten door middel van een automatische binaire zoekactie.

Het wordt volledig behandeld in [Binair zoeken](#) en wordt alleen in dat gedeelte genoemd.

### git blame

Het `git blame` commando markeert de regels van elk bestand met welke commit de laatste was die een wijziging invoerde bij elke regel in het bestand en welke persoon de auteur was van die commit. Dit is handig bij het uitzoeken van de persoon zodat er meer informatie kan worden gevraagd over een specifiek gedeelte van je code.

Dit wordt behandeld in [Bestands annotatie](#) en wordt alleen in dat gedeelte genoemd.

### git grep

Het `git grep` commando kan je helpen elke willekeurige tekenreeks of regular expressie te vinden in elk bestand in je broncode, zelfs oudere versies van je project.

Het wordt behandeld in [Git Grep](#) en wordt alleen in dat gedeelte genoemd.

## Patchen

Een aantal commando's in Git draaien om het concept van het zien van commits in termen van de wijzigingen die ze introduceren, alsof de commit reeks een reeks van patches is. Deze commando's helpen je je branches op deze manier te beheren.

## git cherry-pick

Het `git cherry-pick` commando wordt gebruikt om de wijziging die in een enkele Git commit zit te pakken en deze te herintroduceren als een nieuwe commit op de branch waar je op dat moment op zit. Dit kan behulpzaam zijn bij het selectief een of twee commits te nemen van een branch in plaats van de hele branch in te mergen, waarbij alle wijzigingen worden genomen.

Cherry picking wordt beschreven en gedemonstreerd in [Rebasende en cherry pick workflows](#).

## git rebase

Het `git rebase` commando is eigenlijk een geautomatiseerde `cherry-pick`. Het bepaalt een reeks van commits en gaat deze dan een voor een in dezelfde volgorde elders cherry-picken.

Rebasen wordt in detail behandeld in [Rebasen](#), inclusief het behandelen van de samenwerkingsproblematiek waar je mee te maken krijgt als je al publieke branches gaat rebasen.

We gebruiken het in de praktijk tijdens een voorbeeld van het splitsen van je historie in twee aparte repositories in [Vervangen](#), waarbij we ook de `--onto` vlag gebruiken.

We behandelen het in een merge conflict geraken tijdens rebasen in [Rerere](#).

We gebruiken het ook in een interactieve script modus met de `-i` optie in [Meerdere commit berichten wijzigen](#).

## git revert

Het `git revert` commando is feitelijk een omgekeerde `git cherry-pick`. Het maakt een nieuwe commit die de exacte tegenhanger van de wijziging die in de commit zit die je aanwijst toepast, en deze effectief ontdoet of terugdraait.

We gebruiken het in [De commit terugdraaien](#) om een merge commit terug te draaien.

# Email

Veel Git projecten, inclusief Git zelf, worden volledig onderhouden via mail-lijsten. Git heeft een aantal gereedschappen ingebouwd gekregen die helpen dit proces eenvoudiger te maken, van het genereren van patches die je eenvoudig kunt mailen tot het toepassen van deze patches vanuit een email-box.

## git apply

Het `git apply` commando past een patch toe die met het `git diff` of zelfs met het GNU diff commando is gemaakt. Dit is vergelijkbaar met wat het `patch` commando zou kunnen doen met een paar kleine verschillen.

We laten het gebruik zien en de omstandigheden waarin je het zou kunnen toepassen in [Patches uit e-mail toepassen](#).

## git am

Het `git am` commando wordt gebruikt om patches toe te passen vanuit een email inbox, en specifiek een die volgens mbox is geformateerd. Dit is handig voor het ontvangen van patches via email en deze eenvoudig op je project toe te passen.

We hebben de workflow en gebruik rond `git am` behandeld in [Een patch met am toepassen](#) inclusief het gebruik van de `--resolved`, `-i` en `-3` opties.

Er zijn ook een aantal hooks die je kunt gebruiken als hulp in de workflow rond `git am` en ze worden allemaal behandeld in [E-mail workflow hooks](#).

We hebben het ook gebruikt om GitHub Pull Request wijzigingen die als patch zijn geformateerd toe te passen in [Email berichten](#).

## git format-patch

Het `git format-patch` commando wordt gebruikt om een reeks van patches te genereren in mbox formaat die je kunt gebruiken om ze correct geformateerd naar een mail lijst te sturen.

We behandelen een voorbeeld van een bijdrage leveren aan een project met gebruik van het `git format-patch` tool in [Openbaar project per e-mail](#).

## git imap-send

Het `git imap-send` commando zendt een mailbox gegenereerd met `git format-patch` naar een IMAP drafts folder.

We behandelen een voorbeeld van het bijdragen aan een project door het sturen van patches met de `git imap-send` tool in [Openbaar project per e-mail](#).

## git send-email

Het `git send-email` commando wordt gebruikt om patches via email te sturen die zijn gegenereerd met `git format-patch`.

We behandelen een voorbeeld van het bijdragen aan een project door het sturen van patches met het `git send-email` tool in [Openbaar project per e-mail](#).

## git request-pull

Het `git request-pull` commando wordt eenvoudigweg gebruikt om een voorbeeld bericht te genereren om naar iemand te mailen. Als je een branch hebt op een publieke server en iemand wilt laten weten hoe deze wijzigingen kunnen worden geïntegreerd zonder de patches via email te versturen, kan je dit commando aanroepen en de uitvoer sturen naar de persoon die je de wijzigingen wilt laten pullen.

We laten het gebruik van `git request-pull` om een pull message te laten genereren in [Gevorkt openbaar project](#).

# Externe systemen

Git wordt geleverd met een aantal commando's om te integreren met andere versiebeheer systemen.

## git svn

Het `git svn` commando wordt gebruikt om als client te communiceren met het Subversion versiebeheer systeem. Dit houdt in dat je Git kunt gebruiken om checkouts en commits te doen naar en van een Subversion server.

Dit commando wordt gedetailleerd behandeld in [Git en Subversion](#).

## git fast-import

Voor andere versiebeheer systemen of het impoteren van bijna elk formaat, kan je `git fast-import` gebruiken om snel het andere formaat te mappen op iets wat Git eenvoudig kan vastleggen.

Dit commando wordt gedetailleerd behandeld in [Importeren op maat](#).

# Beheer

Als je een Git repository beheert of iets ingrijpend moet repareren, geeft Git je een aantal beheer commando's om je hierbij te helpen.

## git gc

Het `git gc` commando roept “vuilnis ophalen” (“garbage collection”) aan op je repository, waarbij onnodige bestanden in je database worden verwijderd en de overgebleven bestanden op een meer efficiënte manier worden opgeslagen.

Dit commando wordt normaal gesproken op de achtergrond voor jou aangeroepen, maar je kunt het handmatig aanroepen als je dat wilt. We behandelen een paar voorbeelden in [Onderhoud](#).

## git fsck

Het `git fsck` commando wordt gebruikt om de interne database te controleren op problemen of inconsistenties.

We gebruiken het slechts een keer heel kort in [Gegevensherstel](#) om te zoeken naar loshangende (dangling) objecten.

## git reflog

Het `git reflog` commando neemt een log door waarin staat waar alle heads van al je branches hebben gestaan als je op zoek bent naar commits die je misschien bent verloren bij het herschrijven van histories.

We behandelen dit commando voornamelijk in [RefLog verkorte namen](#), waar we het normale

gebruik laten zien en hoe `git log -g` te gebruiken om dezelfde informatie te laten zien als in de `git log` uitvoer.

We behandelen ook een praktisch voorbeeld van het herstellen van zo'n verloren branch in [Gegevensherstel](#).

## git filter-branch

Het `git filter-branch` commando wordt gebruikt om grote hoeveelheden commits te herschrijven volgens een bepaald patroon, zoals het overal verwijderen van een bestand of een hele repository terug te brengen tot een enkele subdirectory voor het extraheren van een project.

In [Een bestand uit iedere commit verwijderen](#) leggen we het commando uit en verkennen een aantal verschillende opties zoals `--commit-filter`, `--subdirectory-filter` en `--tree-filter`.

In [Git-p4](#) en [TFS](#) gebruiken we dit om geïmporteerde externe repositories te op te schonen.

## Binnenwerk commando's (plumbing commando's)

Er zijn ook nog een behoorlijk aantal meer technische binnenwerk commando's die we zijn tegengekomen in het boek.

De eerste die we tegenkomen is `ls-remote` in [Pull Request Refs \(Pull Request referenties\)](#) waar we het gebruiken om te kijken naar de kale referenties op de server.

We gebruiken `ls-files` in [Handmatig bestanden opnieuw mergen](#), [Rerere](#) en [De index](#) om een meer technische kijk te nemen op hoe je staging gebied er eigenlijk uitziet.

We gebruiken `rev-parse` in [Branch referenties](#) om zo ongeveer elke tekenreeks te nemen en het in een object SHA-1 te veranderen.

Echter, de meeste van de technische binnenwerk commando's die we behandelen staan in [Git Binnenwerk](#), wat meteen het belangrijkste onderwerp is waar dit hoofdstuk zich op richt. We hebben geprobeerd het gebruik van deze commando's in de rest van het boek te vermijden.

# Index

@

\$EDITOR, 345

\$VISUAL

zie \$EDITOR, 345

(branches

langlopend, 80

.NET, 507

.gitignore, 347

0 , 92

1 , 92

A

Apache, 118

Apple, 508

aliasen, 61

archiveren, 361

attributen, 355

autocorrect, 347

B

Bazaar, 431

BitKeeper, 13

bash, 497

beleid voorbeeld, 367

bestanden

verplaatsen, 40

verwijderen, 38

bestanden negeren, 32

bijdragen, 130

besloten aangestuurd team, 140

besloten klein team, 133

gevorkt openbare project, 146

openbaar groot project, 150

openbare klein project, 146

binaire bestanden, 355

bitnami, 121

branches, 63

basis workflow, 70

beheren, 78

creating, 65

diffing, 158

maken, 65

managing, 78

mergen, 74

overschakelen, 66

remote, 83, 157

remote verwijderen, 93

samenvoegen, 74

stroomopwaarts, 91

switching, 66

topic, 81, 154

tracking, 91

upstream, 91

branches, long-running, 80

build numbers, 167

C

C, 503

C#, 507

CRLF, 19

CVS, 10

Cocoa, 508

commit templates, 345

contributing, 130

private managed team, 140

private small team, 133

public large project, 150

public small project, 146

credential caching, 19

credentials, 337

crlf, 352

D

Dulwich, 514

difftool, 349

distributed git, 127

E

Eclipse, 496

e-mail, 152

editor

standaard wijzigen, 37

een project beheren, 153

email

patches applyen vanuit, 154

excludes, 347, 447

F

forking, 129, 174

## G

    GPG, 346  
    GUIs, 489  
    Git als een client, 378  
    GitHub, 169  
        API, 216  
        Flow, 175  
        organisaties, 208  
        pull requests, 178  
        user accounts, 169  
    GitHub voor Mac, 491  
    GitHub voor Windows, 491  
    GitLab, 121  
    GitWeb, 119  
    Go, 512  
    Grafische tools, 489  
    git commando's  
        add, 29, 30, 30  
        am, 155  
        apply, 154  
        archive, 167  
        branch, 65, 78  
        checkout, 66  
        cherry-pick, 164  
        clone, 27  
            bare, 109  
        commit, 36, 63  
        config, 23, 37, 61, 152, 344  
        configuratie, 21  
        credential, 337  
        daemon, 115  
        describe, 167  
        diff, 34  
            check, 131  
        fast-import, 437  
        fetch, 53  
        fetch-pack, 473  
        filter-branch, 435  
        format-patch, 151  
        git commando's  
            help, 24  
        gitk, 489  
        gui, 489  
        help, 115  
        http-backend, 117

## I

    init, 27, 30  
        bare, 110, 114  
    instaweb, 120  
    log, 41  
    merge, 73  
        squash, 150  
    mergetool, 77  
    p4, 408, 434  
    pull, 53  
    push, 53, 59, 89  
    rebase, 94  
    receive-pack, 472  
    remote, 51, 52, 54, 55  
    request-pull, 147  
    rerere, 165  
    send-pack, 472  
    shortlog, 168  
    show, 58  
    show-ref, 381  
    status, 28, 36  
    svn, 378  
    tag, 56, 57, 59  
    upload-pack, 473  
    git-svn, 378  
    git-tf, 416  
    git-tfs, 416  
    gitk, 489  
    go-git, 512

## H

    hooks, 364  
        post-update, 107

## I

    IRC, 24  
    Importeren  
        uit Mercurial, 428  
        uit Perforce, 434  
        uit Subversion, 425  
        uit TFS, 436  
        uit andere, 437  
        vanuit Bazaar, 431

## J

    Java, 508

jgit, 508

## K

keyword expansion, 358

kleur, 348

## L

Linus Torvalds, 13

Linux, 13

  installeren, 18

libgit2, 503

log filteren, 47

log formatteren, 44

## M

Mac

  installeren, 18

Mercurial, 389, 428

Migreren naar Git, 425

Mono, 507

master, 64

merge

  conflicten, 76

mergen, 74

  strategieën, 363

mergetool, 349

merging, 74

  conflicts, 76

  vs. rebasing, 102

## O

Objective-C, 508

origin, 84

## P

Perforce, 10, 400, 434

  Git Fusion, 401

Powershell, 19

Python, 508, 514

pager, 346

posh-git, 500

powershell, 500

protocollen

  local, 104

protocols

SSH, 108

domme HTTP, 106

dumb HTTP, 106

git, 108

slimme HTTP, 106

smart HTTP, 106

pullen, 92

pushen, 89

## R

Ruby, 504

rebasen, 93

  gevaren van, 98

rebasing, 93

  vs. merging, 102

references

  remote, 83

referenties

  remote, 83

regel einden

  line endings, 352

releasen, 167

rerere, 165

## S

SHA-1, 15

SSH keys, 112

  met GitHub, 170

SSH sleutels, 112

Samenwerking met andere VCSen

  Mercurial, 389

  Perforce, 400

  Subversion, 378

  TFS, 416

Subversion, 10, 128, 378, 425

serving repositories, 104

  GitLab, 121

  GitWeb, 119

  HTTP, 117

  SSH, 111

  git protocol, 115

shell prompts

  bash, 497

  powershell, 500

  zsh, 498

sleutelwoord expansie, 358  
staging area  
overslaan, 38

## Z

zsh, 498

## T

TFS, 416, 436

TFVC

see=TFS, 416

tab completion

bash, 497

powershell, 500

zsh, 498

tags, 56, 166

annotated, 57

beschreven, 57

lichtgewicht, 58

lightweight, 58

signing, 166

tekenen, 166

## V

Visual Studio, 495

versie beheer, 9

gecentraliseerd

centraal, 10

gedistribueerd, 11

lokaal, 9

## W

Windows

installeren, 19

werk integreren, 160

witruimtes

whitespace, 352

workflows, 127

dictator en luitenanten, 129

gecentraliseerd, 127

integratie manager, 128

mergen, 160

mergen (grote), 162

rebases en cherry-picking, 164

## X

Xcode, 18