

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 2.1.282-1-g5e3d89e3, 2024-05-18

# Table of Contents

Lizenz .....	1
Vorwort von Scott Chacon .....	2
Vorwort von Ben Straub .....	4
Widmungen .....	5
Mitwirkende .....	6
Anmerkung der Übersetzer .....	8
Einleitung .....	9
Erste Schritte .....	11
Was ist Versionsverwaltung? .....	11
Kurzer Überblick über die Historie von Git .....	15
Was ist Git? .....	15
Die Kommandozeile .....	20
Git installieren .....	20
Git Basis-Konfiguration .....	23
Hilfe finden .....	26
Zusammenfassung .....	27
Git Grundlagen .....	28
Ein Git-Repository anlegen .....	28
Änderungen nachverfolgen und im Repository speichern .....	30
Anzeigen der Commit-Historie .....	43
Ungewollte Änderungen rückgängig machen .....	50
Mit Remotes arbeiten .....	55
Taggen .....	60
Git Aliases .....	66
Zusammenfassung .....	67
Git Branching .....	68
Branches auf einen Blick .....	68
Einfaches Branching und Merging .....	76
Branch-Management .....	85
Branching-Workflows .....	89
Remote-Banches .....	92
Rebasing .....	102
Zusammenfassung .....	112
Git auf dem Server .....	113
Die Protokolle .....	113
Git auf einem Server einrichten .....	119
Erstellung eines SSH-Public-Keys .....	121
Einrichten des Servers .....	122

Git-Daemon .....	125
Smart HTTP .....	127
GitWeb .....	128
GitLab .....	130
Von Drittanbietern gehostete Optionen .....	135
Zusammenfassung .....	135
Verteiltes Git .....	136
Verteilter Arbeitsablauf .....	136
An einem Projekt mitwirken .....	140
Ein Projekt verwalten .....	164
Zusammenfassung .....	179
GitHub .....	180
Einrichten und Konfigurieren eines Kontos .....	180
Mitwirken an einem Projekt .....	186
Ein Projekt betreuen .....	206
Verwalten einer Organisation .....	221
Skripte mit GitHub .....	224
Zusammenfassung .....	235
Git Tools .....	236
Revisions-Auswahl .....	236
Interaktives Stagen .....	244
Stashen und Bereinigen .....	249
Deine Arbeit signieren .....	255
Suchen .....	259
Den Verlauf umschreiben .....	263
Reset entzaubert .....	271
Fortgeschrittenes Merging .....	293
Rerere .....	311
Debuggen mit Git .....	317
Submodule .....	320
Bundling .....	342
Replace (Ersetzen) .....	345
Anmeldeinformationen speichern .....	354
Zusammenfassung .....	359
Git einrichten .....	360
Git Konfiguration .....	360
Git-Attribute .....	371
Git Hooks .....	379
Beispiel für Git-forcierte Regeln .....	383
Zusammenfassung .....	392
Git und andere VCS-Systeme .....	393

Git als Client .....	393
Migration zu Git .....	427
Zusammenfassung .....	442
<b>Git Interna .....</b>	<b>443</b>
Basisbefehle und Standardbefehle (Plumbing and Porcelain) .....	443
Git Objekte .....	444
Git Referenzen .....	455
Packdateien (engl. Packfiles) .....	459
Die Referenzspezifikation (engl. Refspec) .....	462
Transfer Protokolle .....	465
Wartung und Datenwiederherstellung .....	471
Zusammenfassung .....	484
<b>Appendix A: Git in anderen Umgebungen .....</b>	<b>485</b>
Grafische Schnittstellen .....	485
Git in Visual Studio .....	491
Git in Visual Studio Code .....	491
Git in IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine .....	492
Git in Sublime Text .....	492
Git in Bash .....	493
Git in Zsh .....	494
Git in PowerShell .....	496
Zusammenfassung .....	498
<b>Appendix B: Git in Ihre Anwendungen einbetten .....</b>	<b>499</b>
Die Git-Kommandozeile .....	499
Libgit2 .....	499
JGit .....	505
go-git .....	508
Dulwich .....	510
<b>Appendix C: Git Kommandos .....</b>	<b>512</b>
Setup und Konfiguration .....	512
Projekte importieren und erstellen .....	514
Einfache Snapshot-Funktionen .....	515
Branching und Merging .....	518
Projekte gemeinsam nutzen und aktualisieren .....	520
Kontrollieren und Vergleichen .....	522
Debugging .....	523
Patchen bzw. Fehlerkorrektur .....	524
E-mails .....	525
Externe Systeme .....	526
Administration .....	526
Basisbefehle .....	527

Index .....	529
-------------	-----

# Lizenz

Dieses Werk ist lizenziert unter der „Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported“ Lizenz. Um eine Kopie dieser Lizenz zu lesen, besuche <https://creativecommons.org/licenses/by-nc-sa/3.0> oder sende einen Brief an Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

# Vorwort von Scott Chacon

Herzlich willkommen bei der zweiten Auflage von Pro Git. Seit die erste Auflage vor fast vier Jahren veröffentlicht wurde, hat sich eine Menge in der Welt von Git verändert und doch sind viele wichtige Dinge gleich geblieben. Das Kernteam von Git stellt sicher, und das machen sie ziemlich gut, dass die grundlegenden Befehle und Struktur abwärtskompatibel bleiben. Und doch gab es ein paar bedeutende Ergänzungen in Git und andere Weiterentwicklungen rund um die Git-Community. In der zweiten Auflage des Buches sollen diese Änderungen behandelt werden. Außerdem wurden viele Passagen aktualisiert, so dass vor allem neue Git Benutzer leichter einsteigen können.

Zu der Zeit, als ich die erste Edition geschrieben habe, war Git relativ schwierig zu bedienen und kaum benutzerfreundlich. Es richtete sich eher an fortgeschrittenen Anwender. In einigen Communities wurde es immer beliebter, aber es war lange nicht so allgegenwärtig, wie es heute ist. Inzwischen verwendet nahezu jede im Open Source Bereich tätige Community Git. Es gab unglaubliche Fortschritte auf Windows-Betriebssystemen, zahlreiche grafische Oberflächen für alle Plattformen wurden veröffentlicht und die Integration in Entwicklungsumgebungen und im Geschäftsbereich wurde verbessert. Das hätte ich mir vor vier Jahren nicht vorstellen können. In dieser neuen Auflage möchte ich besonders die zahlreichen, neu erschlossenen Bereiche der Git Community thematisieren.

Die Open Source Community, die Git verwendet, ist sprichwörtlich explodiert. Als ich mich vor fast fünf Jahren hingesetzt habe, um dieses Buch zu schreiben (ja, es hat eine ganze Weile gedauert um das erste Buch zu veröffentlichen), habe ich gerade bei einer eher unbekannten kleinen Firma angefangen zu arbeiten. Diese Firma beschäftigte sich mit der Entwicklung einer Git Hosting Website, genannt GitHub. Zu der Zeit, als das Buch veröffentlicht wurde, gab es vielleicht ein paar tausend Leute, die die Website benutzt haben und wir waren nur zu viert, die an ihr gearbeitet haben. Während ich dieses Vorwort schreibe, kündigt GitHub unser 10-millionstes, gehostetes Projekt an. GitHub hat zu diesem Zeitpunkt über 5 Millionen registrierte Benutzer und unterhält mehr als 230 Mitarbeiter. Man kann es gut oder schlecht finden, aber GitHub hat große Teile der Open Source Community verändert, wie ich es mir in der Zeit, als ich das erste Buch geschrieben habe, in meinen kühnsten Träumen nicht vorstellen können.

In der ersten Auflage von Pro Git habe ich ein kurzes Kapitel über GitHub verfasst. Ich wollte damit zeigen, wie Git Hosting aussehen kann. Ich habe mich beim Schreiben des Kapitels aber nie richtig wohl gefühlt. Ich mochte nicht, dass ich über etwas schreibe, was aus einer Community heraus entstanden ist und in die meine Firma involviert ist. Ich mag diesen Interessenkonflikt immer noch nicht, aber die Bedeutung von GitHub in der Git Community ist unverkennbar. Ich habe mich deshalb dazu entschlossen, dass angesprochene Kapitel umzuschreiben und statt einem Beispiel für Git Hosting möchte ich genauer erklären, was GitHub ist und wie man es effektiv nutzen kann. Wenn man vor hat, sich mit Git zu beschäftigen und man weiß, wie GitHub funktioniert, hilft es einem sehr gut, ein Teil einer riesigen Gemeinschaft zu werden. Das kann sehr wertvoll sein und schlussendlich ist es dann auch egal, für welchen Git Hosting Partner man sich für seinen eigenen Code entscheidet.

Eine weitere, große Änderung im Bereich Git seit dem letzten Erscheinen des Buches, war die Weiterentwicklung und -verbreitung des HTTP Protokolls für die Übertragung von Git Daten. Deshalb habe ich die meisten Beispiele angepasst und statt SSH wird jetzt HTTP verwendet, was

vieles wesentlich einfacher macht.

Es war großartig dabei zuzuschauen, wie sich Git die letzten paar Jahre weiterentwickelt hat, von einem doch eher obskuren Versionskontrollsystem zu einem dominierenden Versionskontrollsystem im Open Source und Geschäftsbereich. Ich bin glücklich, wie es bisher mit Pro Git gelaufen ist und dass es einer der wenigen technischen Bücher auf dem Markt ist, welches sowohl ziemlich erfolgreich als auch uneingeschränkt Open Source ist.

Ich hoffe, du hast viel Spaß mit der neuen Auflage von Pro Git.

# Vorwort von Ben Straub

Die erste Ausgabe dieses Buches war der Grund, warum ich mich für Git begeistern konnte. Es war mein Einstieg in eine Form der Softwareentwicklung, die sich natürlicher anfühlte als alles andere, das ich zuvor gesehen hatte. Ich war bis dahin mehrere Jahre lang Entwickler gewesen, doch jetzt war die richtige Wendung, die mich auf einen viel interessanteren Weg führte als den, auf dem ich bisher unterwegs war.

Jetzt, Jahre später, trage ich etwas zu einer bedeutenden Git-Implementierung bei. Ich habe für das größte Git-Hosting-Unternehmen gearbeitet und ich bin durch die ganze Welt gereist, um Menschen Git beizubringen. Als Scott mich fragte, ob ich Interesse hätte, an der zweiten Ausgabe des Buches mitzuarbeiten, musste ich nicht lange darüber nachdenken.

Es war mir eine große Freude und ein Privileg, an diesem Buch mitzuwirken. Ich hoffe, es wird euch genauso helfen wie mir.

# **Widmungen**

*An meine Frau Becky, ohne die dieses Abenteuer nie begonnen hätte. – Ben*

*Diese Ausgabe ist meinen Mädchen gewidmet. Meiner Frau Jessica, die mich all die Jahre unterstützt hat und meiner Tochter Josephine, die mich unterstützen wird, wenn ich zu alt bin, um noch zu verstehen, was vor sich geht. – Scott*

# Mitwirkende

Da es sich um ein Open-Source-Buch handelt, haben wir im Laufe der Jahre verschiedene Errata und inhaltliche Änderungen erhalten. Nachfolgend sind alle Personen aufgelistet, die zum Open-Source-Projekt der deutschen Version von Pro Git beigetragen haben. Vielen Dank an alle, die mitgeholfen haben, dieses Buch zu verbessern.

Contributors as of 5e3d89e3:

4wk-	Jordan Hayashi	Severino Lorilla Jr
Adam Laflamme	Joris Valette	Shengbin Meng
Adrien Ollier	Josh Byster	Siarhei Bobryk
Akrom K	Joshua Webb	Siarhei Krukau
Aleh Suprunovich	Junjie Yuan	Skyper
Alexander Bezzubov	Justin Clift	Snehal Shekatkar
Alexandre Garnier	Jörg Ewald	Song Li
Alfred Myers	Kaartic Sivaraam	Stefan Pinnow
Andreas Kromke	Katrin Leinweber	Stephan van Maris
Andrei Dascalu	Kausar Mehmood	Steven Roddis
Andrew MacFie	Keith Hill	SudarsanGP
Andrew Metcalf	Kenneth Kin Lum	Suhaib Mujahid
Andrew Murphy	Klaus Frank	Sven Selberg
AndyGee	Kristijan "Fremen" Velkovski	Thanix
AnneTheAgile	Krzysztof Szumny	Thomas Ackermann
Anthony Loiseau	Kyrylo Yatsenko	Thomas Hartmann
Anton Trunov	Lars K.W. Gohlke	Tom Schady
Antonello Piemonte	Lars Vogel	Tomoki Aonuma
Antonino Ingargiola	Laxman	Tvirus
Atul Varma	Lazar95	Tyler Cipriani
Ben Sima	Leonard Laszlo	Ud Yzr
Benjamin Dopplinger	Linus Heckemann	Vadim Markovtsev
Borek Bernard	Logan Hasson	Vangelis Katsikaros
Brett Cannon	Louise Corrigan	Victor Ma
Buzut	Luc Morin	Vitaly Kuznetsov
C Nguyen	Lukas Röllin	William Gathoye
Cadel Watson	Manfred	William Turrell
Carlos Martín Nieto	Marcin Sędzik-Jakubowski	Włodek Bzyl
Casper Clemens Mierau	Marie-Helene Burle	Xavier Bonaventura
Casper	Marius Žilénas	Yann Soubeyrand
Chaitanya Gurrapu	Markus KARG	Yue Lin Ho
Changwoo Park	Markus Rennings	Yunhai Luo
Christoph Bachhuber	Marti Bolivar	Yusuke SATO
Christoph Prokop	Mashrur Mia (Sa'ad)	ajax333221
Christopher Wilson	Masood Fallahpoor	alex-koziell
CodingSpiderFox	Mathieu Dubreuilh	allen joslin
Cory Donnelly	Matthew Miner	arohde69
Cullen Rhodes	Matthieu Moy	atalakam
Cyril	MichaWiedenmann	axmbo
Damien Tournoud	Michael MacAskill	bodo22

Dan Schmidt	Michael Sheaver	bripmccann
Daniel Shahaf	Michael Welch	brotherben
Daniel Stauffer	Michiel van der Wulp	christophprokop
Daniel Sturm	Mike Charles	delta4d
Daniele Tricoli	Mike Pennisi	devwebcl
Daniil Larionov	Mike Thibodeau	dualsky
Danny Lin	Mister-Muffin	evanderiel
David Rogers	Mo-Gul	eyherabh
Davide Angelocola	Niels Widger	flip111
Denis Savitskiy	Nils Reuße	flyingzumwalt
Dexter Morganov	Olleg Samoylov	goekboet
DiamonddeX	Oscar Santiago	graue70
Dieter Ziller	Owen	grgbnc
Dino Karic	Pablo Schläpfer	haripetrov
Dmitri Tikhonov	Pascal Berger	i-give-up
Duncan Dean	Pascal Borreli	iprok
Eden Hochbaum	Patrick Steinhardt	jingsam
Eric Henziger	Pavel Janík	johnhar
Explorare	Paweł Krupiński	knittl
Ezra Buehler	Peter Kokot	luwol03
Felix Nehrke	Pfiffikus	maks
Fernsehkind	Phil Mitchell	mmikeww
Filip Kucharczyk	Philipp Montesano	mosdalsvsocld
Fornost461	Philippe Miossec	nicktime
Frank	Rafi	osantiag
Frank Theile	Ralph Haussmann	pastatopf
Frederico Mazzone	Raphael Kruse	patrick96
Frej Drexhammar	Raphael R	paveljanik
Guthrie McAfee Armstrong	Ray Chen	pedrorijo91
HairyFotr	Rex Kerr	peterwwillis
Hamidreza Mahdavipanah	Reza Ahmadi	petsuter
Haruo Nakayama	Richard Hoyle	rahrah
Helmut K. C. Tessarek	Ricky Senft	rance-muhammitz
HonkingGoose	Rintze M. Zelle	rmzelle
Howard	Rob Blanco	root
Ignacy	Robert P. Goldman	sanders@oreilly.com
Ilker Cat	Robert P. J. Day	saxc
J.M. Rütter	Rohan D'Souza	sp-icy
Jan Groenewald	Roman Kosenko	spacewander
Jan Lemke	Ronald Wampler	stredani
Jaswinder Singh	Rüdiger Herrmann	td2014
Jean-Noël Avila	SATO Yusuke	twekberg
Jeroen Oortwijn	Sam Ford	uerdogan
Jim Hill	Sam Joseph	un1versal
Joel Davies	Sanders Kleinfeld	xJom
Johannes Dewender	Sarah Schneider	xtreak
Johannes Schindelin	Saurav Sachidanand	yakirwin
John Lin	Scott Bronson	zwPapEr
Jon Forrest	Sean Head	၂၀၂၀၂၀၂
Jon Freed	Sebastian Krause	၂၀

# Anmerkung der Übersetzer

Die vorliegende Übersetzung des englischen Original-Textes wurde ausschließlich von freiwilligen, nicht professionellen Übersetzern geleistet.

Wir bitten um Nachsicht, falls die eine oder andere Passage nicht sonderlich elegant übersetzt wurde.

Wir ermuntern jedem, der einen Fehler entdeckt oder eine Verbesserung vorschlagen kann dazu, im deutschen Repository entweder einen [Pull-Request](#) oder ein [Issue](#) zu starten.

## Aktualität

Da auch die Übernahme von Änderungen im englischen Original-Text nur von Zeit zu Zeit erfolgt, kann es vorkommen, dass der deutsche Text auf einer etwas älteren Version der englischen Version basiert.

Falls du im deutschen Buch Unklarheiten finden solltest, dann ist es ratsam, zum Vergleich immer im [englischen Buch](#) nachzuschlagen.

# Einleitung

Du bist im Begriff, viele Stunden mit dem Lesen eines Buches über Git zu verbringen. Nehmen wir uns eine Minute Zeit, um zu erklären, was wir für dich bereithalten. Hier findest du eine kurze Zusammenfassung der zehn Kapitel und drei Anhänge dieses Buches.

In **Kapitel 1** werden wir Version Control Systeme (VCSs) und die Grundlagen von Git behandeln – kein technisches Zeug: was Git ist, warum es in einem Land voller VCSs überhaupt entstanden ist, was es auszeichnet und warum so viele Menschen es benutzen. Dann werden wir beschreiben, wie du Git herunterladen und initial einrichten kannst, wenn du es noch nicht auf deinem System installiert hast.

In **Kapitel 2** gehen wir auf die grundlegende Git-Verwendung ein – wie du Git in den 80% der Fälle verwendest, die dir wahrscheinlich am häufigsten begegnen werden. Nachdem du dieses Kapitel gelesen haben, solltest du in der Lage sein, ein Repository zu klonen, zu sehen, was in seiner Verlaufshistorie passiert ist, Dateien zu modifizieren und Änderungen beizutragen. Wenn das Buch zu diesem Zeitpunkt spontan in Flammen aufgeht, solltest du in der Zeit, die du brauchst, um dir ein weiteres Exemplar zu holen, bereits ziemlich versiert im Umgang mit Git sein.

In **Kapitel 3** geht es um das Branching-Modell von Git, das oft als Gits Killer-Funktion beschrieben wird. Hier erfährst du, was Git wirklich von der Masse abhebt. Wenn du das Kapitel zu Ende gebracht hast, wirst du vielleicht in einem ruhigen Moment darüber nachdenken, wie du bisher ohne Gits Branching das Leben können (du wirst keine Antwort finden :-)).

**Kapitel 4** befasst sich mit Git auf dem Server. Mit diesem Kapitel wenden wir uns an diejenigen von euch, die Git innerhalb Ihrer Organisation oder auf Ihrem eigenen persönlichen Server für die gemeinsame Entwicklung einrichten möchten. Wir werden auch verschiedene Hosting-Optionen erörtern, falls du es vorziehst, dass jemand anderes diese Aufgabe für dich übernimmt.

**Kapitel 5** geht ausführlich auf verschiedene dezentrale Workflows ein und wie man sie mit Git realisieren kann. Wenn du dieses Kapitel durchgearbeitet hast, solltest du in der Lage sein, professionell mit mehreren Remote-Repositorys zu arbeiten, Git über E-Mail zu verwenden und geschickt mit zahlreichen Remote-Banches und beigesteuerten Patches zu hantieren.

**Kapitel 6** befasst sich detailliert mit dem GitHub-Hosting-Service und dem Tooling. Wir behandeln die Anmeldung und Verwaltung eines Kontos, die Erstellung und Nutzung von Git-Repositorys, Git-Workflows, um zu Projekten beizutragen und Beiträge für deine Projekte anzunehmen, die Programmoberfläche von GitHub und viele kleine Tipps, die dir das tägliche Arbeiten im Allgemeinen erleichtern.

**Kapitel 7** befasst sich mit anspruchsvollen Git-Befehlen. Hier erfährst du mehr über Themen wie das Beherrschen des „furchterregenden“ Reset-Befehls, die Verwendung der Binärsuche zur Identifizierung von Fehlern, die Bearbeitung des Verlaufs, die detaillierte Auswahl der Revision und vieles mehr. Dieses Kapitel wird dein Wissen über Git perfektionieren, so dass du zu einem echten Meister wirst.

**Kapitel 8** behandelt die Konfiguration deiner individuellen Git-Umgebung. Dazu gehört die Einrichtung von Hook-Skripten zur Umsetzung oder Unterstützung eigener Regeln und die Verwendung von Konfigurationseinstellungen für die Benutzerumgebung, damit du so arbeiten

kannst, wie du es dir vorstellst. Wir befassen uns auch mit der Erstellung eigener Skripte zur Umsetzung einer benutzerdefinierten Commit-Richtlinie.

**Kapitel 9** beschäftigt sich mit Git und anderen VCS-Systemen. Dazu gehört die Verwendung von Git in einer Subversion-Umgebung (SVN-Umgebung) und die Umwandlung von Projekten aus anderen VCSs nach Git. Viele Unternehmen verwenden immer noch SVN und haben nicht vor, etwas zu ändern. Du hast jedoch die unglaubliche Leistungsfähigkeit von Git kennengelernt. Dieses Kapitel zeigt dir, wie du damit umgehen kannst, wenn du noch einen SVN-Server verwenden musst. Wir besprechen auch, wie man Projekte aus unterschiedlichen Systemen importiert, wenn du dann (hoffentlich) alle davon überzeugt hast, den Sprung nach Git zu wagen.

**Kapitel 10** vertieft die dunklen und zugleich wundervollen Tiefen der Git-Interna. Jetzt, da du alles über Git weißt und es mit Macht und großer Eleganz bedienen kannst, kannst du weiter darüber reden, wie Git seine Objekte speichert, was das Objektmodell ist, Einzelheiten zu Packfiles, Server-Protokollen und vielem mehr. Im gesamten Buch werden wir auf Abschnitte dieses Kapitels Bezug nehmen. Falls du an diesem Punkt das Bedürfnis hast, tiefer in die technischen Details einzutauchen (so wie wir), solltest du vielleicht zuerst Kapitel 10 lesen. Das überlassen wir ganz dir.

In **Anhang A** schauen wir uns eine Reihe von Beispielen für den Git-Einsatz in verschiedenen speziellen Anwendungsumgebungen an. Wir befassen uns mit einer Vielzahl verschiedener GUIs und Entwicklungs-Umgebungen, in denen du Git einsetzen kannst. Wenn du an einem Überblick über die Verwendung von Git in deiner Shell, deiner IDE oder deinem Texteditor interessiert bist, schaue hier nach.

In **Anhang B** erkunden wir das Scripting und die Erweiterung von Git durch Tools wie libgit2 und JGit. Wenn du an der Entwicklung komplexer, schneller und benutzerdefinierter Tools interessiert bist und einen Low-Level-Git-Zugang benötigst, kannst du hier nachlesen, wie so etwas funktioniert.

Schließlich gehen wir in **Anhang C** alle wichtigen Git-Befehle einzeln durch und wiederholen, wo wir sie im Buch behandelt haben und wie wir sie genutzt haben. Wenn du wissen möchtest, wo in dem Buch wir einen bestimmten Git-Befehl verwendet haben, kannst du ihn hier nachschlagen.

Lass uns beginnen.

# Erste Schritte

In diesem Kapitel geht es um den Einstieg in Git. Wir erklären zunächst einige Hintergrundinformationen zu Versionskontrolltools, gehen dann dazu über, wie du Git auf deinem System zum Laufen bringst und wie du es schließlich so einrichtest, dass du damit arbeiten kannst. Am Ende dieses Kapitels solltest du verstehen, wozu Git gut ist und weshalb man es verwenden sollte. Du solltest dann in der Lage sein, mit deiner Arbeit in Git loslegen zu können.

## Was ist Versionsverwaltung?

Was ist „Versionsverwaltung“, und warum sollten Sie sich dafür interessieren? Versionsverwaltung ist ein System, welches die Änderungen an einer oder einer Reihe von Dateien über die Zeit hinweg protokolliert, sodass man später auf eine bestimmte Version zurückgreifen kann. Die Dateien, die in den Beispielen in diesem Buch unter Versionsverwaltung gestellt werden, enthalten Quelltext von Software. Tatsächlich kann in der Praxis nahezu jede Art von Datei per Versionsverwaltung nachverfolgt werden.

Wenn du Grafik- oder Webdesigner bist und jede Version eines Bildes oder Layouts behalten möchtest (was Du mit Sicherheit möchtest), ist die Verwendung eines Versionskontrollsysteins (VCS) eine sehr kluge Sache. Damit kannst du ausgewählte Dateien auf einen früheren Zustand zurücksetzen, das gesamte Projekt auf einen früheren Zustand zurücksetzen, zurückliegende Änderungen vergleichen, sehen wer zuletzt etwas geändert hat, das möglicherweise ein Problem verursacht, wer wann ein Problem verursacht hat und vieles mehr. Die Verwendung eines VCS bedeutet im Allgemeinen auch, dass du problemlos eine Wiederherstellung durchführen kannst, wenn etwas kaputt geht oder Dateien verloren gehen. All diese Vorteile erhält man mit einem nur sehr geringen, zusätzlichen Aufwand.

## Lokale Versionsverwaltung

Viele Menschen betreiben Versionsverwaltung, indem sie einfach all ihre Dateien in ein separates Verzeichnis kopieren (die Schlauerer darunter verwenden ein Verzeichnis mit Zeitstempel im Namen). Dieser Ansatz ist sehr verbreitet, weil er so einfach ist, aber auch unglaublich fehleranfällig. Man arbeitet sehr leicht im falschen Verzeichnis, bearbeitet damit die falschen Dateien oder überschreibt Dateien, die man eigentlich nicht überschreiben wollte.

Aus diesem Grund, haben Programmierer bereits vor langer Zeit, lokale Versionsverwaltungssysteme entwickelt, die alle Änderungen an allen relevanten Dateien in einer Datenbank verwalten.

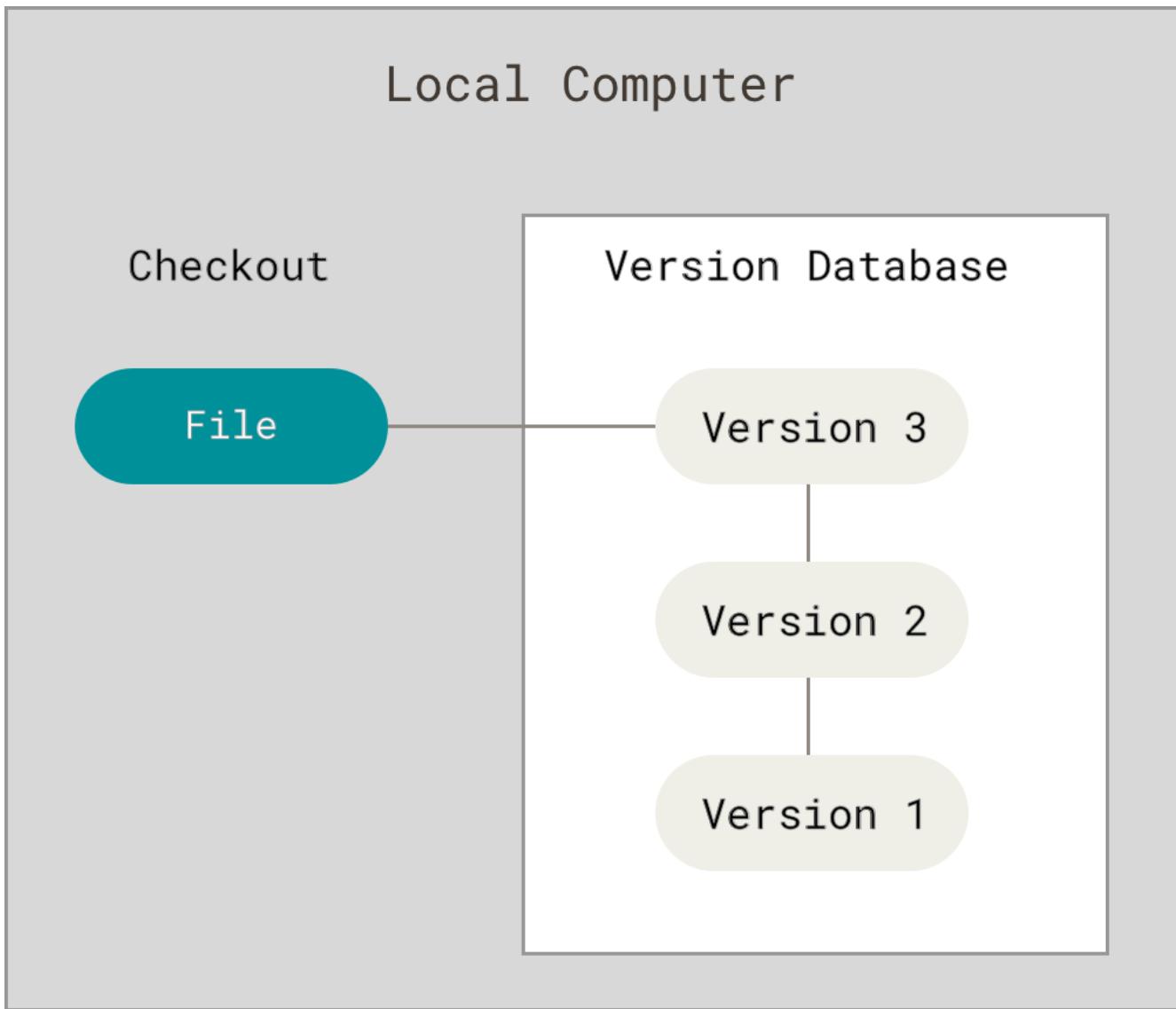


Figure 1. Lokale Versionsverwaltung

Eines der populäreren Versionsverwaltungssysteme war RCS, welches auch heute noch mit vielen Computern ausgeliefert wird. [RCS](#) arbeitet nach dem Prinzip, dass für jede Änderung ein Patch (ein Patch umfasst alle Änderungen an einer oder mehreren Dateien) in einem speziellen Format auf der Festplatte gespeichert wird. Um eine bestimmte Version einer Datei wiederherzustellen, wendet es alle Patches bis zur gewünschten Version an und rekonstruiert damit die Datei in der gewünschten Version.

## Zentrale Versionsverwaltung

Ein weiteres großes Problem, mit dem sich viele Leute konfrontiert sahen, bestand in der Zusammenarbeit mit anderen Entwicklern auf anderen Systemen. Um dieses Problem zu lösen, wurden zentralisierte Versionsverwaltungssysteme entwickelt (engl. Centralized Version Control System, CVCS). Diese Systeme, wozu beispielsweise CVS, Subversion und Perforce gehören, basieren auf einem zentralen Server, der alle versionierte Dateien verwaltet. Die Clients können die Dateien von diesem zentralen Ort abholen und auf ihren PC übertragen. Den Vorgang des Abholens nennt man Auschecken (engl. to check out). Diese Art von System war über viele Jahre hinweg der Standard für Versionsverwaltungssysteme.

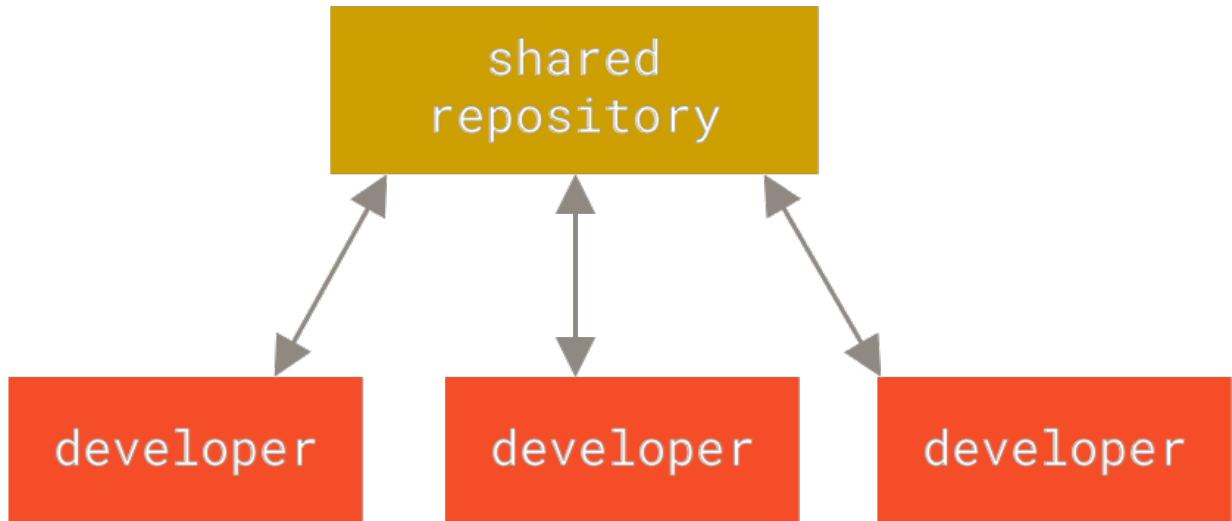


Figure 2. Zentrale Versionsverwaltung

Dieser Ansatz hat viele Vorteile, besonders gegenüber lokalen Versionsverwaltungssystemen. Zum Beispiel weiß jeder mehr oder weniger genau darüber Bescheid, was andere an einem Projekt Beteiligte gerade tun. Administratoren haben die Möglichkeit, detailliert festzulegen, wer was tun kann. Und es ist sehr viel einfacher, ein CVCS zu administrieren als lokale Datenbanken auf jedem einzelnen Anwenderrechner zu verwalten.

Allerdings hat dieser Aufbau auch einige erhebliche Nachteile. Der offensichtlichste Nachteil ist das Risiko eines Systemausfalls bei Ausfall einer einzelnen Komponente, d.h. wenn der zentrale Server ausfällt. Wenn dieser Server nur für eine Stunde nicht verfügbar ist, dann kann in dieser einen Stunde niemand in irgendeiner Form mit anderen zusammenarbeiten oder Dateien, an denen gerade gearbeitet wird, versioniert abspeichern. Wenn die auf dem zentralen Server eingesetzte Festplatte beschädigt wird und keine Sicherheitskopien erstellt wurden, dann sind all diese Daten unwiederbringlich verloren – die komplette Historie des Projektes, abgesehen natürlich von dem jeweiligen Zustand, den Mitstreiter gerade zufällig auf ihrem Rechner noch vorliegen haben. Lokale Versionskontrollsysteme haben natürlich dasselbe Problem: Wenn man die Historie eines Projektes an einer einzigen, zentralen Stelle verwaltet, riskiert man, sie vollständig zu verlieren, wenn irgendetwas an dieser zentralen Stelle schief läuft.

## Verteilte Versionsverwaltung

Und an dieser Stelle kommen verteilte Versionsverwaltungssysteme (engl. Distributed Version Control System, DVCS) ins Spiel. In einem DVCS (wie z.B. Git, Mercurial, Bazaar oder Darcs) erhalten Anwender nicht einfach nur den jeweils letzten Zustand des Projektes von einem Server: Sie erhalten stattdessen eine vollständige Kopie des Repositories. Auf diese Weise kann, wenn ein Server beschädigt wird, jedes beliebige Repository von jedem beliebigen Anwenderrechner zurückkopiert werden und der Server so wiederhergestellt werden. Jede Kopie, ein sogenannter Klon (engl. clone), ist ein vollständiges Backup der gesamten Projektdaten.

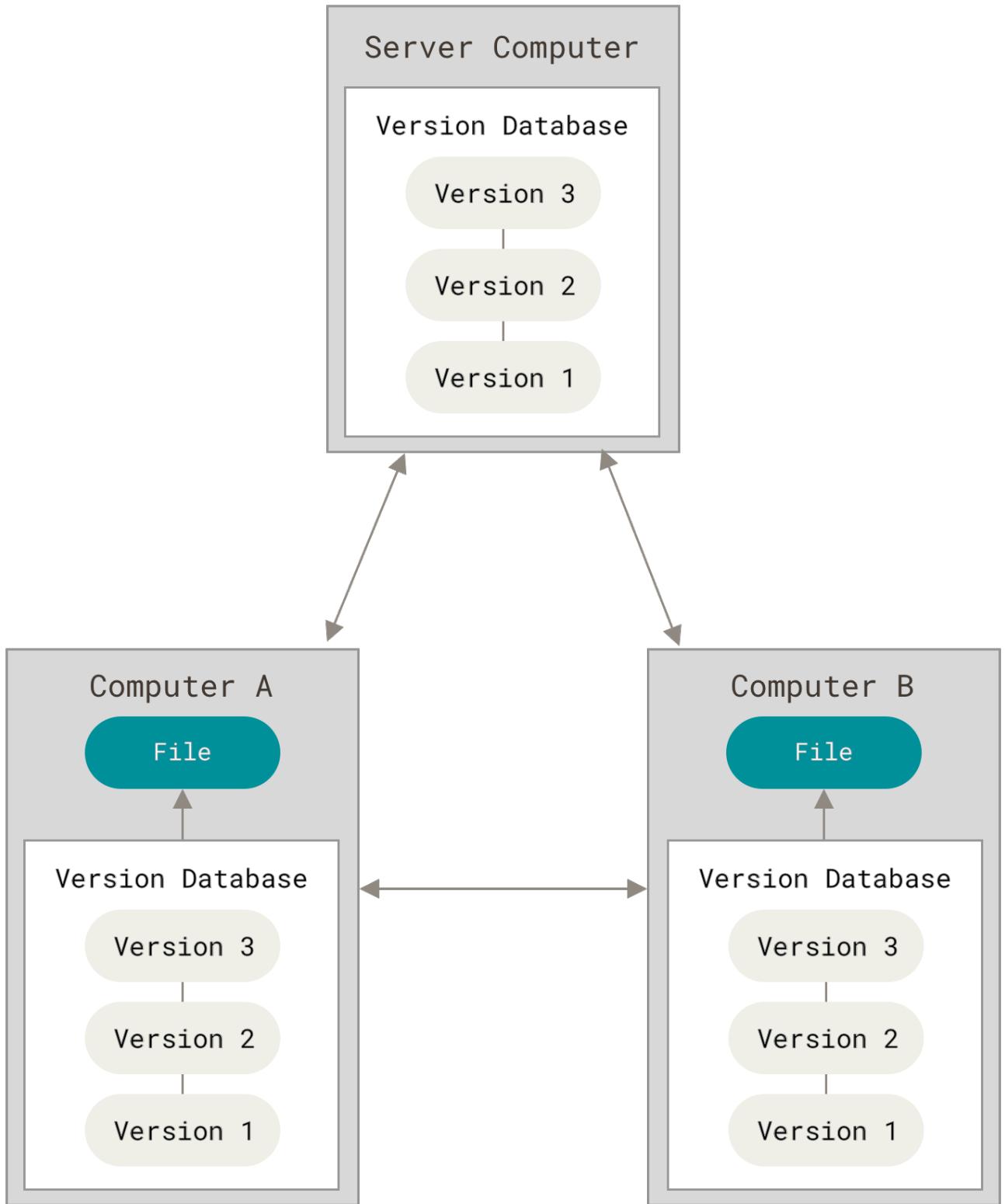


Figure 3. Verteilte Versionsverwaltung

Darüber hinaus können derartige Systeme hervorragend mit verschiedenen externen Repositories, sogenannten Remote-Repositories, umgehen, sodass man mit verschiedenen Gruppen von Leuten simultan auf verschiedene Art und Weise an einem Projekt zusammenarbeiten kann. Damit ist es möglich, verschiedene Arten von Arbeitsabläufen zu erstellen und anzuwenden, welche mit zentralisierten Systemen nicht möglich wären. Dazu gehören zum Beispiel hierarchische Arbeitsabläufe.

# Kurzer Überblick über die Historie von Git

Wie viele großartige Dinge im Leben, entstand Git aus einer Prise kreativem Chaos und hitziger Diskussion.

Der Linux-Kernel ist ein Open-Source-Software-Projekt von erheblichem Umfang. Während der frühen Jahre der Linux-Kernel Entwicklung (1991 - 2002) wurden Änderungen an diesem, in Form von Patches und archivierten Dateien, herumgereicht. 2002 begann man dann, ein proprietäres DVCS mit dem Namen Bitkeeper zu verwenden.

2005 ging die Beziehung zwischen der Community, die den Linux-Kernel entwickelte, und des kommerziell ausgerichteten Unternehmens, das BitKeeper entwickelte, in die Brüche. Die zuvor ausgesprochene Erlaubnis, BitKeeper kostenlos zu verwenden, wurde widerrufen. Dies war für die Linux-Entwickler-Community (und besonders für Linus Torvalds, dem Erfinder von Linux) der Auslöser dafür, ein eigenes Tool zu entwickeln, das auf den Erfahrungen mit BitKeeper basierte. Die Ziele des neuen Systems waren unter anderem:

- Geschwindigkeit
- Einfaches Design
- Gute Unterstützung von nicht-linearer Entwicklung (tausende parallele Entwicklungs-Branches)
- Vollständig dezentrale Struktur
- Fähigkeit, große Projekte, wie den Linux Kernel, effektiv zu verwalten (Geschwindigkeit und Datenumfang)

Seit seiner Geburt 2005 entwickelte sich Git kontinuierlich weiter und reifte zu einem System heran, das einfach zu bedienen ist, die ursprünglichen Ziele dabei aber weiter beibehält. Es ist unglaublich schnell, äußerst effizient, wenn es um große Projekte geht, und es hat ein fantastisches Branching-Konzept für nicht-lineare Entwicklung (siehe Kapitel 3 [Git Branching](#)).

## Was ist Git?

Also, was ist Git denn nun? Dies ist ein wichtiger Teil, den es zu beachten gilt, denn wenn du verstehst, was Git und das grundlegenden Konzept seiner Arbeitsweise ist, dann wird die effektive Nutzung von Git für dich wahrscheinlich viel einfacher sein. Wenn du Git erlernst, solltest du versuchen, deinen Kopf von den Dingen zu befreien, die du über andere VCSs, wie CVS, Subversion oder Perforce, weist. Das wird dir helfen, unangenehme Missverständnisse bei der Verwendung des Tools zu vermeiden. Auch wenn die Benutzeroberfläche von Git den VCSs sehr ähnlich ist, speichert und betrachtet Git Informationen auf eine ganz andere Weise, und das Verständnis dieser Unterschiede hilft dir Unklarheiten bei der Verwendung zu vermeiden.

## Snapshots statt Unterschiede

Der Hauptunterschied zwischen Git und anderen Versionsverwaltungssystemen (insbesondere auch Subversion und vergleichbaren Systemen) besteht in der Art und Weise wie Git die Daten betrachtet. Konzeptionell speichern die meisten anderen Systeme Informationen als eine Liste dateibasierter Änderungen. Diese Systeme (CVS, Subversion, Perforce, Bazaar usw.) betrachtet die Informationen, die sie verwaltet, als eine Reihe von Dateien an denen im Laufe der Zeit

Änderungen vorgenommen werden (dies wird allgemein als *deltabasierte* Versionskontrolle bezeichnet).

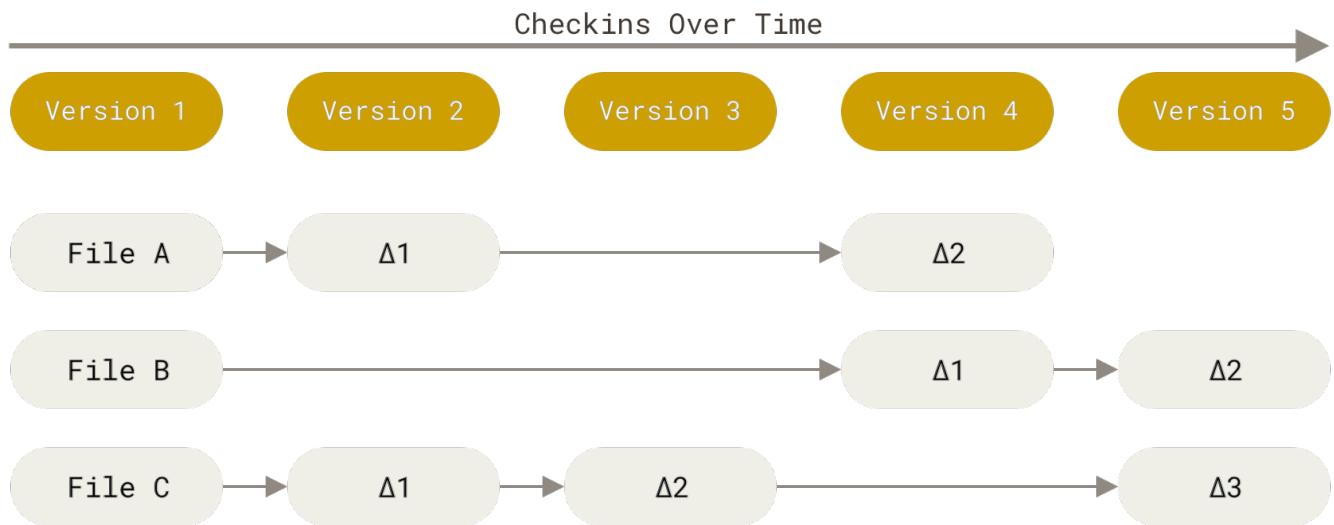


Figure 4. Speichern von Daten als Änderung an einzelnen Dateien auf Basis einer Ursprungsdatei

Git arbeitet nicht auf diese Art. Stattdessen betrachtet Git seine Daten eher wie eine Reihe von Schnappschüssen eines Mini-Dateisystems. Git macht jedes Mal, wenn du den Status deines Projekts committest (das heißt den gegenwärtigen Status deines Projekts als eine Version in Git speicherst) ein Abbild von all deinen Dateien wie sie gerade aussehen und speichert einen Verweis in diesem Schnappschuss. Um dies möglichst effizient und schnell tun zu können, kopiert Git unveränderte Dateien nicht, sondern legt lediglich eine Verknüpfung zu der vorherigen Version der Datei an. Git betrachtet seine Daten eher als einen **Stapel von Schnappschüssen**.

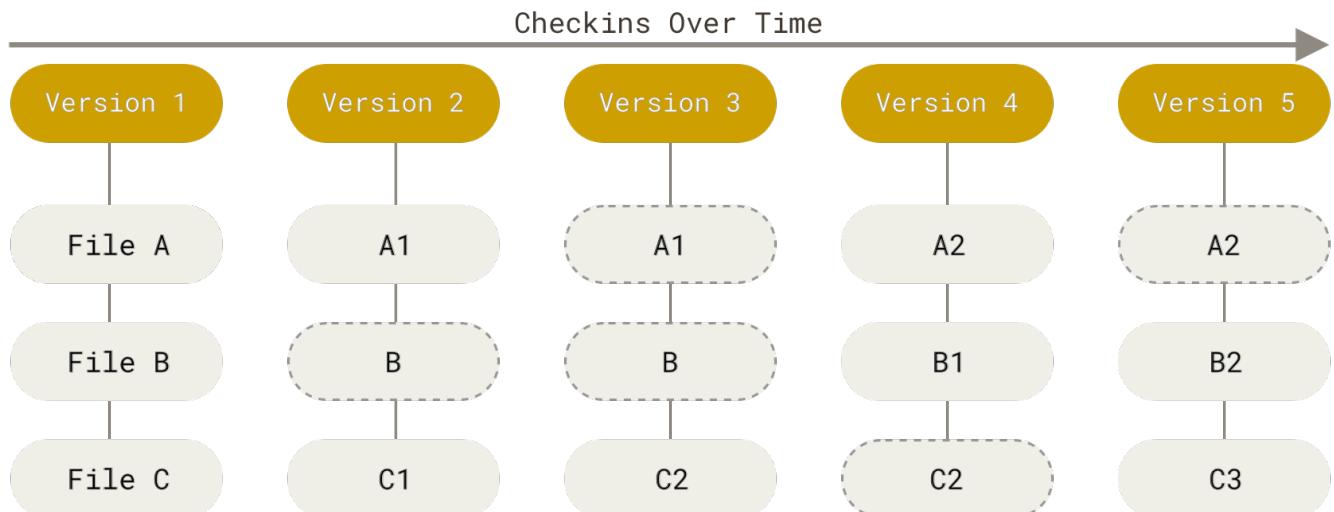


Figure 5. Speichern der Daten-Historie eines Projekts in Form von Schnappschüssen

Das ist ein wichtiger Unterschied zwischen Git und praktisch allen anderen Versionsverwaltungssystemen. In Git wurden fast alle Aspekte der Versionsverwaltung neu überdacht, die in anderen Systemen mehr oder weniger von ihrer jeweiligen Vorgängergeneration übernommen worden waren. Git arbeitet im Großen und Ganzen eher wie ein mit einigen unglaublich mächtigen Werkzeugen ausgerüstetes Mini-Dateisystem, statt nur als Versionsverwaltungssystem. Auf einige der Vorteile, die es mit sich bringt, Daten in dieser Weise zu verwalten, werden wir in Kapitel 3, [Git Branching](#), eingehen, wenn wir das Git Branching-Konzept kennenlernen.

## Fast jede Funktion arbeitet lokal

Die meisten Aktionen in Git benötigen nur lokale Dateien und Ressourcen, um ausgeführt zu werden – im Allgemeinen werden keine Informationen von einem anderen Computer in deinem Netzwerk benötigt. Wenn du mit einem CVCS vertraut bist, bei dem die meisten Operationen durch Netzwerk-Latenzen langsam sind, dann wird diese Eigenschaft von Git glauben lassen, dass Git von mit übernatürlichen Kräften bedacht wurde. Die allermeisten Operationen können nahezu ohne jede Verzögerung ausgeführt werden, da die vollständige Historie eines Projekts bereits auf dem jeweiligen Rechner verfügbar ist.

Um beispielsweise die Historie des Projekts zu durchsuchen, braucht Git sie nicht von einem externen Server zu holen – es liest diese einfach aus der lokalen Datenbank. Das heißt, die vollständige Projekthistorie ist ohne jede Verzögerung verfügbar. Wenn man sich die Änderungen einer aktuellen Version einer Datei im Vergleich zu der vor einem Monat anschauen möchte, dann kann Git den Stand von vor einem Monat in der lokalen Historie nachschlagen und einen lokalen Vergleich zur vorliegenden Datei durchführen. Für diesen Anwendungsfall benötigt Git keinen externen Server, weder um Dateien dort nachzuschlagen, noch um Unterschiede zu finden.

Das bedeutet natürlich außerdem, dass es fast nichts gibt, was man nicht tun kann, nur weil man gerade offline ist oder keinen Zugriff auf ein VPN hat. Wenn man also gerade im Flugzeug oder Zug ein wenig arbeiten will, kann man problemlos seine Arbeit einchecken und dann, wenn man wieder mit einem Netzwerk verbunden ist, die Dateien auf den Server hochladen. Wenn man zu Hause sitzt, aber der VPN-Client gerade mal wieder nicht funktioniert, kann man immer noch arbeiten. Bei vielen anderen Systemen wäre dies unmöglich oder äußerst kompliziert umzusetzen. In Perforce kannst du beispielsweise nicht viel tun, wenn du nicht mit dem Server verbunden bist; in Subversion und CVS kannst du Dateien bearbeiten, aber du kannst keine Änderungen zu deinen Daten übertragen (weil die Datenbank offline ist). Das scheint keine große Sache zu sein, aber du wirst überrascht sein, was für einen großen Unterschied das machen kann.

## Git stellt Integrität sicher

Von allen zu speichernden Daten berechnet Git Prüfsummen (engl. checksum) und speichert diese als Referenz zusammen mit den Daten ab. Das macht es unmöglich, dass sich Inhalte von Dateien oder Verzeichnissen ändern, ohne dass Git das mitbekommt. Git basiert auf dieser Funktionalität und sie ist ein integraler Teil der Git-Philosophie. Man kann Informationen deshalb z.B. nicht während der Übermittlung verlieren oder unwissentlich beschädigte Dateien verwenden, ohne dass Git in der Lage wäre, dies festzustellen.

Der Mechanismus, den Git verwendet, um diese Prüfsummen zu erstellen, heißt SHA-1-Hash. Eine solche Prüfsumme ist eine 40-Zeichen lange Zeichenkette, die aus hexadezimalen Zeichen (0-9 und a-f) besteht. Sie wird von Git aus den Inhalten einer Datei oder Verzeichnisstruktur berechnet. Ein SHA-1-Hash sieht wie folgt aus:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Diese Hashes begegnen einem überall bei der Arbeit, weil sie so ausgiebig von Git genutzt werden. Tatsächlich speichert Git alles in seiner Datenbank nicht nach Dateinamen, sondern nach dem Hash-Wert seines Inhalts.

## Git fügt im Regelfall nur Daten hinzu

Wenn du Aktionen in Git durchführen, werden diese fast immer nur Daten zur Git-Datenbank hinzufügen. Deshalb ist es sehr schwer, das System dazu zu bewegen, irgendetwas zu tun, das nicht wieder rückgängig zu machen ist, oder dazu, Daten in irgendeiner Form zu löschen. Wie in jedem anderen VCS, kann man in Git Daten verlieren oder durcheinander bringen, solange man sie noch nicht eingecheckt hat. Aber sobald man einen Schnappschuss in Git eingecheckt hat, ist es sehr schwierig, diese Daten wieder zu verlieren, insbesondere wenn man regelmäßig seine lokale Datenbank auf ein anderes Repository hochlädt.

Unter anderem deshalb macht es so viel Spaß mit Git zu arbeiten. Man weiß genau, man kann ein wenig experimentieren, ohne befürchten zu müssen, irgendetwas zu zerstören oder durcheinander zu bringen. Wer sich genauer dafür interessiert, wie Git Daten speichert und wie man Daten, die scheinbar verloren sind, wiederherstellen kann, dem sei das Kapitel 2, [Ungewollte Änderungen rückgängig machen](#), ans Herz gelegt.

## Die drei Zustände

Jetzt heißt es aufgepasst! Es folgt die wichtigste Information, die man sich merken muss, wenn man Git erlernen und dabei Fallstricke vermeiden will. Git definiert drei Hauptzustände, in denen sich eine Datei befinden kann: committet (engl. *committed*), geändert (engl. *modified*) und für Commit vorgemerkt (engl. *staged*).

- **Modified** bedeutet, dass eine Datei geändert, aber noch nicht in die lokale Datenbank eingecheckt wurde.
- **Staged** bedeutet, dass eine geänderte Datei in ihrem gegenwärtigen Zustand für den nächsten Commit vorgemerkt ist.
- **Committed** bedeutet, dass die Daten sicher in der lokalen Datenbank gespeichert sind.

Das führt uns zu den drei Hauptbereichen eines Git-Projekts: dem Verzeichnisbaum (engl. Working Tree), der sogenannten Staging-Area und dem Git-Verzeichnis.

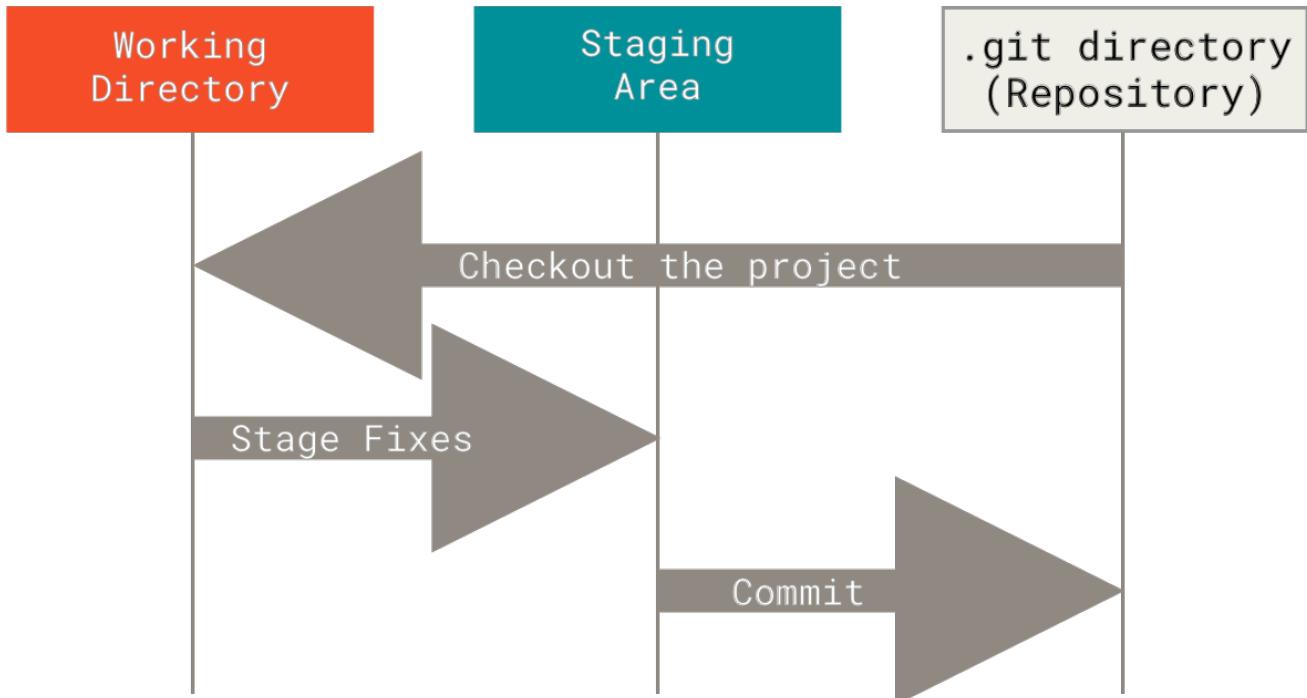


Figure 6. Verzeichnisbaum, Staging-Area und Git-Verzeichnis

Der Verzeichnisbaum ist ein einzelner Checkout einer Version des Projekts. Diese Dateien werden aus der komprimierten Datenbank im Git-Verzeichnis geholt und auf der Festplatte so abgelegt, damit du sie verwenden oder ändern kannst.

Die Staging-Area ist in der Regel eine Datei, die sich in deinem Git-Verzeichnis befindet und Informationen darüber speichert, was in deinem nächsten Commit einfließen soll. Der technische Name im Git-Sprachgebrauch ist „Index“, aber der Ausdruck „Staging-Area“ funktioniert genauso gut.

Im Git-Verzeichnis werden die Metadaten und die Objektdatenbank für dein Projekt gespeichert. Das ist der wichtigste Teil von Git, dieser Teil wird kopiert, wenn man ein Repository von einem anderen Rechner *klont*.

Der grundlegende Git-Arbeitsablauf sieht in etwa so aus:

1. Du änderst Dateien in deinem Verzeichnisbaum.
2. Du stellst selektiv Änderungen bereit, die du bei deinem nächsten Commit berücksichtigen möchtest, wodurch nur diese Änderungen in den Staging-Bereich aufgenommen werden.
3. Du führst einen Commit aus, der die Dateien so übernimmt, wie sie sich in der Staging-Area befinden und diesen Snapshot dauerhaft in deinem Git-Verzeichnis speichert.

Wenn sich eine bestimmte Version einer Datei im Git-Verzeichnis befindet, wird sie als *committed* betrachtet. Wenn sie geändert und in die Staging-Area hinzugefügt wurde, gilt sie als für den Commit *vorgemerkt* (engl. *staged*). Und wenn sie geändert, aber noch nicht zur Staging-Area hinzugefügt wurde, gilt sie als *geändert* (engl. *modified*). Im Kapitel 2, [Git Grundlagen](#), werden diese drei Zustände noch näher erläutert und wie man diese sinnvoll einsetzen kann oder alternativ, wie man den Zwischenschritt der Staging-Area überspringen kann.

# Die Kommandozeile

Es gibt viele verschiedene Möglichkeiten Git einzusetzen. Auf der einen Seite gibt es die Werkzeuge, die per Kommandozeile bedient werden und auf der anderen, die vielen grafischen Benutzeroberflächen (engl. graphical user interface, GUI), die sich im Leistungsumfang unterscheiden. In diesem Buch verwenden wir die Kommandozeile. In der Kommandozeile können nämlich wirklich **alle** vorhandenen Git Befehle ausgeführt werden. Bei den meisten grafischen Oberflächen ist dies nicht möglich, da aus Gründen der Einfachheit nur ein Teil der Git-Funktionalitäten zur Verfügung gestellt werden. Wenn du dich in der Kommandozeilenversion von Git auskennen, findest du dich wahrscheinlich auch in einer GUI-Version relativ schnell zurecht, aber andersherum muss das nicht unbedingt zutreffen. Außerdem ist die Wahl der grafischen Oberfläche eher Geschmackssache, wohingegen die Kommandozeilenversion auf jedem Rechner installiert und verfügbar ist.

In diesem Buch gehen wir deshalb davon aus, dass du weißt, wie man bei einem Mac ein Terminal öffnet, oder wie man unter Windows die Eingabeaufforderung oder die Powershell öffnet. Falls du jetzt nur Bahnhof verstehst, solltest du an dieser Stelle abbrechen und dich schlau machen, was ein Terminal bzw. eine Eingabeaufforderung ist und wie man diese bedient. Nur so ist sichergestellt, dass du unsere Beispiele und Ausführungen im weiteren Verlauf des Buches folgen kannst.

## Git installieren

Bevor du mit Git loslegen kannst, muss es natürlich zuerst installiert werden. Auch wenn es bereits vorhanden ist, ist es vermutlich eine gute Idee, auf die neueste Version zu aktualisieren. Du kannst es entweder als Paket oder über ein anderes Installationsprogramm installieren oder den Quellcode herunterladen und selbst kompilieren.

Dieses Buch wurde auf Basis der Git-Version **2** geschrieben. Da Git hervorragend darin ist, die Abwärtskompatibilität aufrechtzuerhalten, sollte jede neuere Version problemlos funktionieren. Auch wenn die meisten Befehle, die wir anwenden werden, auch in älteren Versionen funktionieren, kann es doch sein, dass die Befehlsausgabe oder das Verhalten leicht anders ist.



### Installation unter Linux

Wenn du die grundlegenden Git-Tools unter Linux über ein Installationsprogramm installieren möchtest, kannst du das in der Regel über das Paketverwaltungstool der Distribution tun. Wenn du mit Fedora (oder einer anderen eng damit verbundenen RPM-basierten Distribution, wie RHEL oder CentOS) arbeitest, kannst du **dnf** verwenden:

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

Auf einem Debian-basierten System, wie Ubuntu, steht **apt** zur Verfügung:

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

Auf der Git-Homepage <https://git-scm.com/download/linux> findet man weitere Möglichkeiten und Optionen, wie man Git unter einem Unix-basierten Betriebssystem installieren kann.

## Installation unter macOS

Es gibt mehrere Möglichkeiten, Git auf einem Mac zu installieren. Am einfachsten ist es wahrscheinlich, die Xcode Command Line Tools zu installieren. Bei Mavericks (10.9) oder neueren Versionen kann man dazu einfach `git` im Terminal eingeben.

```
$ git --version
```

Wenn Git noch nicht installiert ist, erscheint eine Abfrage, ob man es installieren möchte.

Wenn man eine sehr aktuelle Version einsetzen möchte, kann man Git auch über ein Installationsprogramm installieren. Auf der Git-Website <https://git-scm.com/download/mac> findet man die jeweils aktuellste Version und kann sie von dort herunterladen.

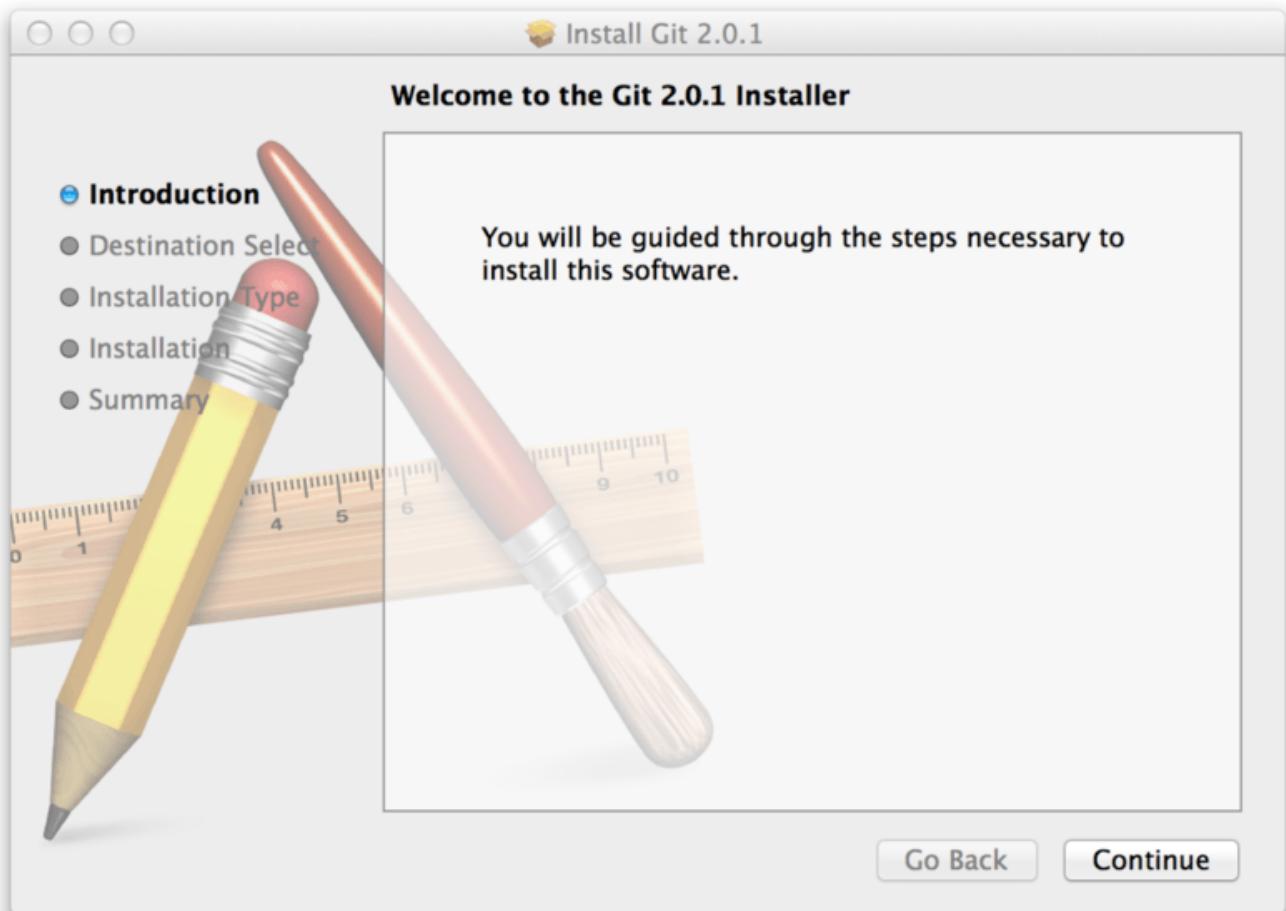


Figure 7. Git macOS Installationsprogramm

## Installation unter Windows

Auch für Windows gibt es einige Möglichkeiten zur Installation von Git. Eine offizielle Windows-Version findet man direkt auf der Git-Homepage. Gehe dazu auf <https://git-scm.com/download/win> und der Download sollte dann automatisch starten. Man sollte dabei beachten, dass es sich hierbei

um das Projekt „Git for Windows“ handelt, welches unabhängig von Git selbst ist. Weitere Informationen hierzu findest du unter <https://msysgit.github.io/>.

Um eine automatisierte Installation zu erhalten, kannst du das [Git Chocolatey Paket](#) verwenden. Beachte, dass das Chocolatey-Paket von der Community gepflegt wird.

## Aus dem Quellcode installieren

Viele Leute kompilieren Git auch auf ihrem eigenen Rechner, weil sie damit die jeweils aktuellste Version erhalten. Die vorbereiteten Pakete hinken meist ein wenig hinterher, obwohl Git in den letzten Jahren ausgereifter geworden ist und dies somit wesentlich besser geworden ist.

Wenn du Git aus dem Quellcode installieren möchtest, benötigst du die folgenden Bibliotheken, von denen Git abhängt: autotools, curl, zlib, openssl, expat und libiconv. Wenn du dich beispielsweise auf einem System befinden, das Paketverwaltungen, wie `dnf` (Fedora) oder `apt-get` (ein Debian-basiertes System) hat, kannst du mit einem dieser Befehle die minimalen Abhängigkeiten für die Kompilierung und Installation der Git-Binärdateien installieren:

```
$ 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 libbz2-dev libssl-dev
```

Um die Dokumentation in verschiedenen Formaten (doc, html, info) zu erstellen, sind weitere Abhängigkeiten notwendig:

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



Benutzer von RHEL und RHEL-Derivaten wie CentOS und Scientific Linux müssen das [EPEL-Repository aktivieren](#), um das Paket `docbook2X` herunterzuladen.

Wenn du eine Debian-basierte Distribution (Debian, Ubuntu oder deren Derivate) verwendest, benötigst du auch das Paket `install-info`:

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

Wenn du eine RPM-basierte Distribution (Fedora, RHEL oder deren Derivate) verwendest, benötigst du auch das Paket `getopt` (welches auf einer Debian-basierten Distribution bereits installiert ist):

```
$ sudo dnf install getopt
```

Wenn du Fedora- oder RHEL-Derivate verwendest, musst du wegen der unterschiedlichen Paketnamen zusätzlich einen Symlink erstellen, indem du folgenden Befehl:

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

aufgrund von binären Namensunterschieden ausführst.

Wenn du alle notwendigen Abhängigkeiten installiert hast, kannst du dir als nächstes die jeweils aktuellste Version als Tarball von verschiedenen Stellen herunterladen. Man findet die Quellen auf der Kernel.org-Website unter <https://www.kernel.org/pub/software/scm/git>, oder einen Mirror auf der GitHub-Website unter <https://github.com/git/git/releases>. Auf der GitHub-Seite ist es einfacher herauszufinden, welches die jeweils aktuellste Version ist. Auf kernel.org dagegen werden auch Signaturen zur Verifikation des Downloads der jeweiligen Pakete zur Verfügung gestellt.

Nachdem man sich so die Quellen beschafft hat, kann man Git kompilieren und installieren:

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

Nachdem dies erledigt ist, kannst du Git für Updates auch über Git selbst beziehen:

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

## Git Basis-Konfiguration

Nachdem Git jetzt auf deinem System installiert ist, solltest du deine Git-Konfiguration noch anpassen. Dies muss nur einmalig auf dem jeweiligen System durchgeführt werden. Die Konfiguration bleibt bestehen, wenn du Git auf eine neuere Version aktualisierst. Die Git-Konfiguration kann auch jederzeit geändert werden, indem die nachfolgenden Befehle einfach noch einmal ausgeführt werden.

Git umfasst das Werkzeug `git config`, welches die Möglichkeit bietet, Konfigurationswerte zu verändern. Auf diese Weise kannst du anpassen, wie Git aussieht und arbeitet. Die Konfiguration ist an drei verschiedenen Orten gespeichert:

1. Die Datei `[path]/etc/gitconfig`: enthält Werte, die für jeden Benutzer auf dem System und alle seine Repositories gelten. Wenn du die Option `--system` an `git config` übergibst, liest und schreibt sie aus dieser Datei. Da es sich um eine Systemkonfigurationsdatei handelt, benötigst du Administrator- oder Superuser-Rechte, um Änderungen daran vorzunehmen.
2. Die Datei `~/.gitconfig` oder `~/.config/git/config`: enthält Werte, die für dich, den Benutzer, persönlich bestimmt sind. Du kannst Git dazu bringen, diese Datei gezielt zu lesen und zu schreiben, indem du die Option `--global` übergibst. Dies betrifft *alle* Repositories, mit denen du auf deinem System arbeitest.
3. Die Datei `config` im Git-Verzeichnis (also `.git/config`) des jeweiligen Repositories, das du gerade

verwendest: Du kannst Git mit der Option `--local` zwingen, aus dieser Datei zu lesen und in sie zu schreiben, das ist in der Regel die Standardoption. (Es dürfte klar sein, dass du dich irgendwo in einem Git-Repository befinden musst, damit diese Option ordnungsgemäß funktioniert.)

Jede Ebene überschreibt Werte der vorherigen Ebene, so dass Werte in `'.git/config'` diejenigen in `'[path]/etc/gitconfig'` aushebelt.

Auf Windows-Systemen sucht Git nach der Datei `.gitconfig` im `$HOME` Verzeichnis (normalerweise zeigt `$HOME` bei den meisten Systemen auf `C:\Users\$USER`). Git schaut immer nach `[path]/etc/gitconfig`, auch wenn die sich relativ zu dem MSys-Wurzelverzeichnis befindet, dem Verzeichnis in das du Git installiert hast. Wenn du eine Version 2.x oder neuer von Git für Windows verwendest, gibt es auch eine Konfigurationsdatei auf Systemebene unter `C:\Dokumente und Einstellungen\Alle Benutzer\Anwendungsdaten\Git\config` unter Windows XP und unter `C:\ProgramData\Git\config` unter Windows Vista und neuer. Diese Konfigurationsdatei kann nur von `git config -f <file>` als Admin geändert werden.

Du kannst dir alle Git Einstellungen ansehen und wo sie herkommen mit:

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

## Deine Identität

Nachdem du Git installiert hast, solltest du als allererstes deinen Namen und deine E-Mail-Adresse in Git konfigurieren. Das ist insofern wichtig, da jeder Git-Commit diese Informationen verwendet und sie unveränderlich in die Commits eingearbeitet werden, die du erstellst:

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

Wie schon erwähnt, brauchst du diese Konfiguration nur einmal vorzunehmen, wenn du die Option `--global` verwendest. Auf Grund der oben erwähnten Prioritäten gilt das dann für alle deine Projekte. Wenn du für ein spezielles Projekt einen anderen Namen oder eine andere E-Mail-Adresse verwenden möchtest, kannst du den Befehl ohne die Option `--global` innerhalb des Projektes ausführen.

Viele der grafischen Oberflächen helfen einem bei diesem Schritt, wenn du sie zum ersten Mal ausführst.

## Dein Editor

Jetzt, da deine Identität eingerichtet ist, kannst du den Standard-Texteditor konfigurieren, der verwendet wird, wenn Git dich zum Eingeben einer Nachricht auffordert. Normalerweise verwendet Git den Standard-Texteditor des jeweiligen Systems.

Wenn du einen anderen Texteditor, z.B. Emacs, verwenden willst, kannst du das wie folgt festlegen:

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

Wenn du auf einem Windows-System einen anderen Texteditor verwenden möchtest, musst du den vollständigen Pfad zu seiner ausführbaren Datei angeben. Dies kann, je nachdem, wie dein Editor eingebunden ist, unterschiedlich sein.

Im Falle von Notepad++, einem beliebten Programmiereditor, solltest du wahrscheinlich die 32-Bit-Version verwenden, da die 64-Bit-Version zum Zeitpunkt der Erstellung nicht alle Plug-Ins unterstützt. Beim Einsatz auf einem 32-Bit-Windows-System oder einem 64-Bit-Editor auf einem 64-Bit-System gibst du etwa Folgendes ein:

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

 Vim, Emacs und Notepad++ sind beliebte Texteditoren, die von Entwicklern häufig auf Unix-basierten Systemen wie Linux und macOS oder einem Windows-System verwendet werden. Wenn du einen anderen Editor oder eine 32-Bit-Version verwenden, findest du in [git config core.editor commands](#) spezielle Anweisungen, wie du deinen bevorzugten Editor mit Git einrichten kannst.

 Wenn du Git nicht so konfigurierst, dass es deinen Texteditor verwendet und du keine Ahnung davon hast, wie man Vim oder Emacs benutzt, wirst du ein wenig überfordert sein, wenn diese zum ersten Mal von Git gestartet werden. Ein Beispiel auf einem Windows-System könnte ein vorzeitig beendeter Git-Vorgang während einer von Git initiierten Editor-Bearbeitung sein.

## Der standardmäßige Branch-Name

In der Voreinstellung wird Git einen Branch mit dem Namen *master* erzeugen, wenn du ein neues Repository mit `git init` erstellst. Ab der Git-Version 2.28 kannst du einen abweichenden Namen für den initialen Branch festlegen.

So konfigurierst du *main* als Vorgabe für den Branch-Namen:

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

## Einstellungen überprüfen

Wenn du deine Konfigurationseinstellungen überprüfen möchtest, kannst du mit dem Befehl `git config --list` alle Einstellungen auflisten, die Git derzeit finden kann:

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

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

Manche Parameter werden möglicherweise mehrfach aufgelistet, weil Git denselben Parameter in verschiedenen Dateien (z.B. `[path]/etc/gitconfig` und `~/.gitconfig`) gefunden hat. In diesem Fall verwendet Git dann den jeweils zuletzt aufgelisteten Wert.

Außerdem kannst du mit dem Befehl `git config <key>` prüfen, welchen Wert Git für einen bestimmten Parameter verwendet:

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

Da Git möglicherweise den gleichen Wert der Konfigurationsvariablen aus mehr als einer Datei liest, ist es möglich, dass du einen unerwarteten Wert für einen dieser Werte erhältst und nicht weißt, warum. In solchen Fällen kannst du Git nach dem *Ursprung* (engl. origin) für diesen Wert fragen. Git wird dir sagen, mit welcher Konfigurationsdatei der Wert letztendlich festgelegt wurde:

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

## Hilfe finden

Falls du einmal Hilfe bei der Verwendung von Git benötigst, gibt es drei Möglichkeiten, die entsprechende Seite aus der Dokumentation (engl. manpage) für jeden Git-Befehl anzuzeigen:

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

Beispielweise erhältst du die Hilfeseite für den Befehl `git config` folgendermaßen:

```
$ git help config
```

Diese Befehle sind nützlich, weil du dir die Hilfe jederzeit anzeigen lassen kannst, auch wenn du einmal offline bist. Wenn die Hilfeseiten und dieses Buch nicht ausreichen und du persönliche Hilfe brauchst, kannst du einen der Kanäle `#git`, `#github` oder `#gitlab` auf dem Libera-Chat-IRC-Server probieren, der unter <https://libera.chat/> zu finden ist. Diese Kanäle sind in der Regel sehr gut besucht. Normalerweise findet sich unter den vielen Anwendern, die oft sehr viel Erfahrung mit Git

haben, irgendjemand, der dir weiterhelfen kann.

Wenn du nicht die vollständige Manpage-Hilfe benötigst, sondern nur eine kurze Beschreibung der verfügbaren Optionen für einen Git-Befehl, kannst du auch in den kompakteren „Hilfeseiten“ mit der Option `-h` nachschauen, wie 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
--renormalize          renormalize EOL of tracked files (implies -u)
-N, --intent-to-add    record only the fact that the path will be added later
-A, --all               add changes from all tracked and untracked files
--ignore-removal        ignore paths removed in the working tree (same as --no
-all)
--refresh              don't add, only refresh the index
--ignore-errors         just skip files which cannot be added because of
errors
--ignore-missing        check if - even missing - files are ignored in dry run
--chmod (+|-)x          override the executable bit of the listed files
--pathspec-from-file <file> read pathspec from file
--pathspec-file-nul     with --pathspec-from-file, pathspec elements are
separated with NUL character
```

## Zusammenfassung

Du solltest nun ein grundlegendes Verständnis davon haben, was Git ist und wie es sich von anderen zentralisierten Versionskontrollsysteinen unterscheidet, die du möglicherweise zuvor verwendet hast. Du solltest nun auch eine funktionierende Installation von Git auf deinem System haben, die mit deiner persönlichen Identität eingerichtet ist. Es ist jetzt an der Zeit, einige Git-Grundlagen zu erlernen.

# Git Grundlagen

Falls du es eilig hast und du nur die Zeit hast ein einziges Kapitel dieses hervorragendes Buches durchzulesen, bist du hier genau richtig.

Dieses Kapitel behandelt alle grundlegenden Befehle, die Du benötigst, um den Großteil der Aufgaben zu erledigen, die mit Git erledigt werden müssen. Am Ende des Kapitels solltest Du in der Lage sein, ein neues Repository anzulegen und zu konfigurieren, Dateien zu versionieren bzw. Sie aus der Versionsverwaltung zu entfernen, Dateien in die Staging-Area hinzuzufügen und einen Commit durchzuführen.

Außerdem wird gezeigt, wie Du Git so konfigurieren kannst, dass es bestimmte Dateien und Dateimuster ignoriert, wie Du Fehler schnell und einfach rückgängig machen, wie Du die Historie eines Projekts durchsuchen und Änderungen zwischen Commits nachschlagen, und wie Du von einem Remote-Repository Daten herunter- bzw. dort hochladen kannst.

## Ein Git-Repository anlegen

Du hast zwei Möglichkeiten, ein Git-Repository auf Deinem Rechner anzulegen.

1. Du kannst ein lokales Verzeichnis, das sich derzeit nicht unter Versionskontrolle befindet, in ein Git-Repository verwandeln, oder
2. Du kannst ein bestehendes Git-Repository von einem anderen Ort aus *klonen*.

In beiden Fällen erhältst Du ein einsatzbereites Git-Repository auf Deinem lokalen Rechner.

## Ein Repository in einem bestehenden Verzeichnis einrichten

Wenn Du ein Projektverzeichnis hast, das sich derzeit nicht unter Versionskontrolle befindet, und Du mit dessen Versionierung über Git beginnen möchten, musst Du zunächst in das Verzeichnis dieses Projekts wechseln. Das sieht je nachdem, welches System verwendet wird, unterschiedlich aus:

für Linux:

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

für macOS:

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

für Windows:

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

Führe dort folgenden Befehl aus:

```
$ git init
```

Der Befehl erzeugt ein Unterverzeichnis `.git`, in dem alle relevanten Git-Repository-Daten enthalten sind, also so etwas wie ein Git-Repository Grundgerüst. Zu diesem Zeitpunkt werden noch keine Dateien in Git versioniert. In Kapitel 10, [Git Interna](#), findest Du weitere Informationen, welche Dateien im `.git` Verzeichnis erzeugt wurden.

Wenn Du bereits existierende Dateien versionieren möchtest (und es sich nicht nur um ein leeres Verzeichnis handelt), dann solltest Du den aktuellen Stand mit einem initialen Commit tracken. Mit dem Befehl `git add` legst Du fest, welche Dateien versioniert werden sollen und mit dem Befehl `git commit` erzeugst Du einen neuen Commit: Du kannst dies mit ein paar `git add`-Befehlen erreichen. Damit markierst du die zu trackende Dateien. Anschließend gibst du ein `git commit` ein:

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

Wir werden gleich noch einmal genauer auf diese Befehle eingehen. Im Moment ist nur wichtig zu verstehen, dass du jetzt ein Git-Repository erzeugt und einen ersten Commit angelegt hast.

## Ein existierendes Repository klonen

Wenn du eine Kopie eines existierenden Git-Repositorys anlegen möchtest – um beispielsweise an einem Projekt mitzuarbeiten – kannst du den Befehl `git clone` verwenden. Wenn du bereits Erfahrung mit einem anderen VCS-System wie Subversion gesammelt hast, fällt dir bestimmt sofort auf, dass der Befehl „clone“ und nicht etwa „checkout“ lautet. Das ist ein wichtiger Unterschied, den du unbedingt verstehen solltest. Anstatt nur eine einfache Arbeitskopie vom Projekt zu erzeugen, lädt Git nahezu alle Daten, die der Server bereithält, auf den lokalen Rechner. Jede Version von jeder Datei der Projekt-Historie wird automatisch heruntergeladen, wenn du `git clone` ausführst. Wenn deine Serverfestplatte beschädigt wird, kannst du nahezu jeden der Klone auf irgendeinem Client verwenden, um den Server wieder in den Zustand zurückzusetzen, in dem er sich zum Zeitpunkt des Klonens befand. (Du wirst vielleicht einige serverseitige Hooks und dergleichen verlieren, aber alle versionierte Daten wären vorhanden – siehe Kapitel 4, [Git auf dem Server](#), für weitere Details.)

Du klonst ein Repository mit dem Befehl `git clone [url]`. Um beispielsweise das Repository der verlinkbaren Git-Bibliothek `libgit2` zu klonen, führst du folgenden Befehl aus:

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

Git legt dann ein Verzeichnis `libgit2` an, initialisiert in diesem ein `.git` Verzeichnis, lädt alle Daten des Repositorys herunter und checkt eine Kopie der aktuellsten Version aus. Wenn du in das neue `libgit2` Verzeichnis wechselst, findest du dort die Projektdateien und kannst gleich damit arbeiten.

Wenn du das Repository in ein Verzeichnis mit einem anderen Namen als `libgit2` klonen möchtest, kannst du das wie folgt durchführen:

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

Dieser Befehl tut das Gleiche wie der vorhergehende, aber das Zielverzeichnis lautet diesmal `mylibgit`.

Git unterstützt eine Reihe unterschiedlicher Übertragungsprotokolle. Das vorhergehende Beispiel verwendet das `https://` Protokoll, aber dir könnten auch die Angaben `git://` oder `user@server:path/to/repo.git` begegnen, welches das SSH-Transfer-Protokoll verwendet. Kapitel 4, [Git auf dem Server](#), stellt alle verfügbaren Optionen vor, die der Server für den Zugriff auf dein Git-Repository hat und die Vor- und Nachteile der möglichen Optionen.

## Änderungen nachverfolgen und im Repository speichern

An dieser Stelle solltest du ein *angemessenes* Git-Repository auf deinem lokalen Computer und eine Checkout- oder Arbeitskopie aller deiner Dateien vor dir haben. Normalerweise wirst du damit beginnen wollen, Änderungen vorzunehmen und Schnappschüsse dieser Änderungen in dein Repository zu committen, wenn das Projekt so weit fortgeschritten ist, dass du es sichern möchten.

Denke daran, dass sich jede Datei in deinem Arbeitsverzeichnis in einem von zwei Zuständen befinden kann: *tracked* oder *untracked* – Änderungen an der Datei werden verfolgt (engl. *tracked*) oder eben nicht (engl. *untracked*). Tracked Dateien sind Dateien, die im letzten Snapshot enthalten sind. Genauso wie alle neuen Dateien in der Staging-Area. Sie können entweder unverändert, modifiziert oder für den nächsten Commit vorgemerkt (*staged*) sein. Kurz gesagt, nachverfolgte Dateien sind Dateien, die Git kennt.

Alle anderen Dateien in deinem Arbeitsverzeichnis dagegen, sind nicht versioniert: Das sind all diejenigen Dateien, die nicht schon im letzten Schnappschuss enthalten waren und die sich nicht in der Staging-Area befinden. Wenn du ein Repository zum ersten Mal klonst, sind alle Dateien versioniert und unverändert. Nach dem Klonen wurden sie ja ausgecheckt und bis dahin hast du auch noch nichts an ihnen verändert.

Sobald du anfängst versionierte Dateien zu bearbeiten, erkennt Git diese als modifiziert, weil sie sich im Vergleich zum letzten Commit verändert haben. Die geänderten Dateien kannst du dann für den nächsten Commit vormerken und schließlich alle Änderungen, die sich in der Staging-Area befinden, einchecken (engl. committen). Danach wiederholt sich dieser Vorgang.

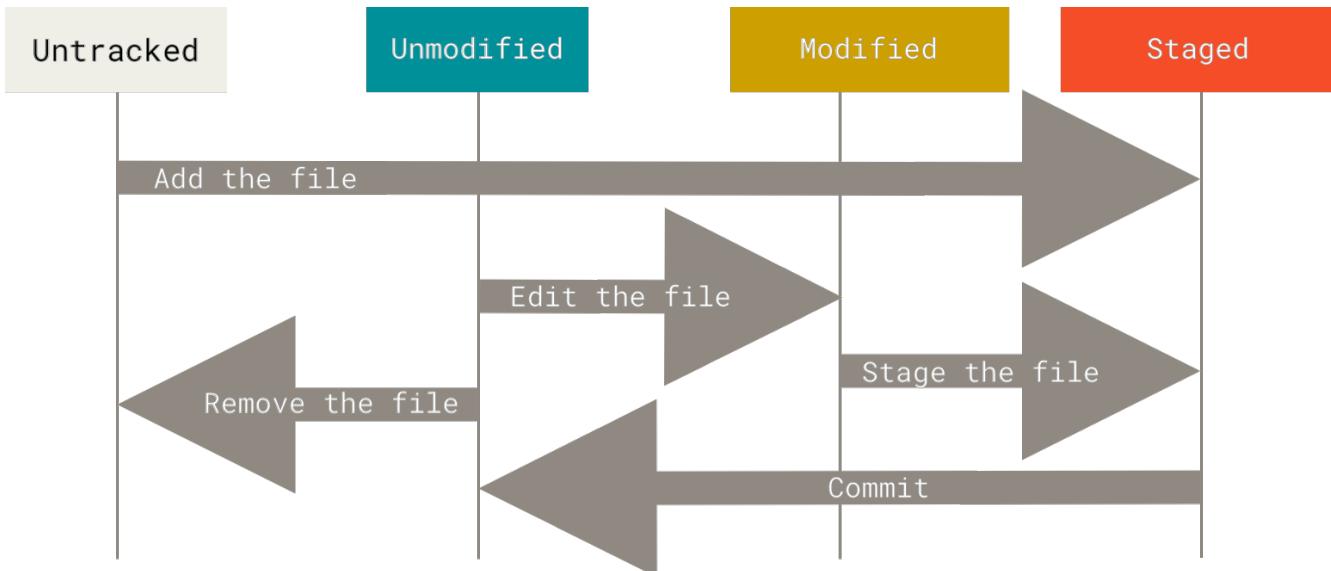


Figure 8. Der Status deiner Dateien im Überblick

## Zustand von Dateien prüfen

Das wichtigste Hilfsmittel, um den Zustand zu überprüfen, in dem sich deine Dateien gerade befinden, ist der Befehl `git status`. Wenn du diesen Befehl unmittelbar nach dem Klonen eines Repositorys ausführst, sollte er in etwa folgende Ausgabe liefern:

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

Dieser Zustand wird auch als sauberes Arbeitsverzeichnis (engl. clean working directory) bezeichnet. Mit anderen Worten, es gibt keine Dateien, die unter Versionsverwaltung stehen und seit dem letzten Commit geändert wurden – andernfalls würden sie hier aufgelistet werden. Außerdem teilt dir der Befehl mit, auf welchem Branch du gerade arbeitest und informiert dich darüber, dass dieser sich im Vergleich zum Branch auf dem Server nicht verändert hat. Momentan ist dieser Branch immer `master`, was der Vorgabe entspricht; Du musst dich jetzt nicht darum kümmern. Wir werden im Kapitel [Git Branching](#) detaillierter auf Branches eingehen.



GitHub änderte Mitte 2020 den Standard-Branch-Namen von `master` in `main`, und andere Git-Hosts folgten diesem Beispiel. Daher wirst du möglicherweise feststellen, dass der Standard-Branch-Name in einigen neu erstellten Repositorys `main` und nicht `master` ist. Außerdem kann der Standard-Branch-Name geändert werden (wie du in [Der standardmäßige Branch-Name](#) gesehen hast), sodass du möglicherweise einen anderen Namen für den Standard-Branch vorfindest.

Git selbst verwendet jedoch immer noch `master` als Standard, also werden wir es auch im gesamten Buch verwenden.

Nehmen wir einmal an, du fügst eine neue Datei mit dem Namen `README` zu deinem Projekt hinzu. Wenn die Datei zuvor nicht existiert hat und du jetzt `git status` ausführst, zeigt Git die bisher nicht

versionierte Datei wie folgt an:

```
$ 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)
```

Alle Dateien, die im Abschnitt „Untracked files“ aufgelistet werden, sind Dateien, die bisher noch nicht versioniert sind. Dort wird jetzt auch die Datei **README** angezeigt. Mit anderen Worten, die Datei **README** wird in diesem Bereich gelistet, weil sie im letzten Schnappschuss nicht enthalten war und noch nicht gestaged wurde. Git nimmt eine solche Datei nicht automatisch in die Versionsverwaltung auf. Du musst Git dazu ausdrücklich auffordern. Ansonsten würden generierte Binärdateien oder andere Dateien, die du nicht in deinem Repository haben möchtest, automatisch hinzugefügt werden. Das möchte man in den meisten Fällen vermeiden. Jetzt wollen wir aber Änderungen an der Datei **README** verfolgen und fügen sie deshalb zur Versionsverwaltung hinzu.

## Neue Dateien zur Versionsverwaltung hinzufügen

Um eine neue Datei zu versionieren, kannst du den Befehl **git add** verwenden. Für deine neue **README** Datei, kannst du folgendes ausführen:

```
$ git add README
```

Wenn du erneut den Befehl **git status** ausführst, wirst du sehen, dass sich deine **README** Datei jetzt unter Versionsverwaltung befindet und für den nächsten Commit vorgemerkt ist:

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

    new file:   README
```

Du kannst erkennen, dass die Datei für den nächsten Commit vorgemerkt ist, weil sie unter der Rubrik „Changes to be committed“ aufgelistet ist. Wenn du jetzt einen Commit anlegst, wird der Schnappschuss den Zustand der Datei festhalten, den sie zum Zeitpunkt des Befehls **git add** hat. Du erinnerst dich vielleicht daran, wie du vorhin **git init** und anschließend **git add <files>** ausgeführt hast. Mit diesen Befehlen hast du die Dateien in deinem Verzeichnis zur Versionsverwaltung hinzugefügt. Der Befehl **git add** akzeptiert einen Pfadnamen einer Datei oder

eines Verzeichnisses. Wenn du ein Verzeichnis angibst, fügt `git add` alle Dateien in diesem Verzeichnis und allen Unterverzeichnissen rekursiv hinzu.

## Geänderte Dateien zur Staging-Area hinzufügen

Las uns jetzt eine bereits versionierte Datei ändern. Wenn du zum Beispiel eine bereits unter Versionsverwaltung stehende Datei mit dem Dateinamen `CONTRIBUTING.md` änderst und danach den Befehl `git status` erneut ausführst, erhältst du in etwa folgende Ausgabe:

```
$ 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
```

Die Datei `CONTRIBUTING.md` erscheint im Abschnitt „Changes not staged for commit“. Das bedeutet, dass eine versionierte Datei im Arbeitsverzeichnis verändert worden ist, aber noch nicht für den Commit vorgemerkt wurde. Um sie vorzumerken, führst du den Befehl `git add` aus. Der Befehl `git add` wird zu vielen verschiedenen Zwecken eingesetzt. Man verwendet ihn, um neue Dateien zur Versionsverwaltung hinzuzufügen, Dateien für einen Commit vorzumerken und verschiedene andere Dinge – beispielsweise einen Konflikt aus einem Merge als aufgelöst zu kennzeichnen. Leider wird der Befehl `git add` oft missverstanden. Viele assoziieren damit, dass damit Dateien zum Projekt hinzugefügt werden. Wie du aber gerade gelernt hast, wird der Befehl auch noch für viele andere Dinge eingesetzt. Wenn du den Befehl `git add` einsetzt, solltest du das eher so sehen, dass du damit einen bestimmten Inhalt für den nächsten Commit vormerkst. Las uns nun mit `git add` die Datei `CONTRIBUTING.md` zur Staging-Area hinzufügen und danach das Ergebnis mit `git status` kontrollieren:

```
$ 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 Dateien sind nun für den nächsten Commit vorgemerkt. Nehmen wir an, du willst jetzt aber

noch eine weitere Änderung an der Datei `CONTRIBUTING.md` vornehmen, bevor du den Commit tatsächlich startest. Du öffnest die Datei erneut, änderst sie entsprechend ab und eigentlich wärst du jetzt bereit den Commit durchzuführen. Las uns vorher noch einmal den Befehl `git status` ausführen:

```
$ 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
```

Was zum Kuckuck ...? Jetzt wird die Datei `CONTRIBUTING.md` sowohl in der Staging-Area als auch als geändert aufgelistet. Wie ist das möglich? Die Erklärung dafür ist, dass Git eine Datei in exakt dem Zustand für den Commit vormerkt, in dem sie sich befindet, wenn du den Befehl `git add` ausführst. Wenn du den Commit jetzt anlegst, wird die Version der Datei `CONTRIBUTING.md` denjenigen Inhalt haben, den sie hatte, als du `git add` zuletzt ausgeführt hast und nicht denjenigen, den sie in dem Moment hat, wenn du den Befehl `git commit` ausführst. Wenn du stattdessen die gegenwärtige Version im Commit haben möchten, müsst du erneut `git add` ausführen, um die Datei der Staging-Area hinzuzufügen:

```
$ 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
```

## Kompakter Status

Die Ausgabe von `git status` ist sehr umfassend und auch ziemlich wortreich. Git hat auch ein Kurzformat, mit dem du deine Änderungen kompakter sehen kannst. Wenn du `git status -s` oder `git status --short` ausführst, erhältst du eine kürzere Darstellung des Befehls:

```
$ git status -s
```

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

Neue Dateien, die nicht versioniert werden, werden mit **??** markiert. Neue Dateien, die der Staging-Area hinzugefügt wurden, haben ein **A**, geänderte Dateien haben ein **M** usw. Es gibt zwei Spalten für die Ausgabe – die linke Spalte zeigt den Status der Staging-Area und die rechte Spalte den Status des Arbeitsverzeichnis. So ist beispielsweise in der Bildschirmausgabe oben die Datei **README** im Arbeitsverzeichnis geändert, aber noch nicht staged, während die Datei **lib/simplegit.rb** geändert und staged ist. Das **Rakefile** wurde modifiziert, staged und dann wieder modifiziert, so dass es Änderungen an ihm gibt, die sowohl staged als auch unstaged sind.

## Ignorieren von Dateien

Häufig gibt es eine Reihe von Dateien, die Git nicht automatisch hinzufügen oder schon gar nicht als „nicht versioniert“ (eng. untracked) anzeigen soll. Dazu gehören in der Regel automatisch generierte Dateien, wie Log-Dateien oder Dateien, die von deinem Build-System erzeugt werden. In solchen Fällen kannst du die Datei **.gitignore** erstellen, die eine Liste mit Vergleichsmustern enthält. Hier ist eine **.gitignore** Beispieldatei:

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

Die erste Zeile weist Git an, alle Dateien zu ignorieren, die auf „.o“ oder „.a“ enden – Objekt- und Archivdateien, die das Ergebnis einer Codegenerierung sein könnten. Die zweite Zeile weist Git an, alle Dateien zu ignorieren, deren Name mit einer Tilde (~) endet, was von vielen Texteditoren wie Emacs zum Markieren temporärer Dateien verwendet wird. Du kannst auch ein Verzeichnis „log“, „tmp“ oder „pid“ hinzufügen, eine automatisch generierte Dokumentation usw. Es ist im Allgemeinen eine gute Idee, die **.gitignore** Datei für dein neues Repository einzurichten, noch bevor du loslegen. So kannst du nicht versehentlich Dateien committen, die du nicht in deinem Git-Repository haben möchtest.

Die Richtlinien für Vergleichsmuster, die du in der Datei **.gitignore** eingeben kannst, lautet wie folgt:

- Leerzeilen oder Zeilen, die mit **#** beginnen, werden ignoriert.
- Standard-Platzhalter-Zeichen funktionieren und werden rekursiv im gesamten Verzeichnisbaum angewendet.
- Du kannst Vergleichsmuster mit einem Schrägstrich (**/**) **beginnen**, um die Rekursivität zu verhindern.
- Du kannst Vergleichsmuster mit einem Schrägstrich (**/**) **beenden**, um ein Verzeichnis anzugeben.
- Du kannst ein Vergleichsmuster negieren, indem es mit einem Ausrufezeichen (!) beginnt.

Platzhalter-Zeichen sind wie einfache, reguläre Ausdrücke, die von der Shell genutzt werden. Ein Sternchen (\*) entspricht null oder mehr Zeichen; [abc] entspricht jedem Zeichen innerhalb der eckigen Klammern (in diesem Fall a, b oder c); ein Fragezeichen (?) entspricht einem einzelnen Zeichen und eckige Klammern, die durch einen Bindestrich ([0-9]) getrennte Zeichen einschließen, passen zu jedem Zeichen dazwischen (in diesem Fall von 0 bis 9). Du kannst auch zwei Sterne verwenden, um verschachtelte Verzeichnisse abzulegen; a/\*\*/z würde zu a/z, a/b/z, a/b/c/z usw. passen.

Hier ist eine weitere `.gitignore` Beispieldatei:

```
# 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 unterhält eine ziemlich umfassende Liste guter `.gitignore` Beispiel-Dateien für Dutzende von Projekten und Sprachen auf <https://github.com/github/gitignore>, falls du einen Ansatzpunkt für dein Projekt suchst.



Im einfachsten Fall kann ein Repository eine einzelne `.gitignore` Datei in seinem Root-Verzeichnis haben, die rekursiv für das gesamte Repository gilt. Es ist aber auch möglich, weitere `.gitignore` Dateien in Unterverzeichnissen anzulegen. Die Regeln dieser verschachtelten `.gitignore` Dateien gelten nur für die in dem Verzeichnis (und unterhalb) liegenden Dateien. Das Linux-Kernel-Source-Repository hat beispielsweise 206 `.gitignore` Dateien.

Es würde den Rahmen dieses Buches sprengen, detaillierter auf den Einsatz mehrerer `.gitignore` Dateien einzugehen; siehe die Manpage `man gitignore` für weitere Informationen.

## Überprüfen der Staged- und Unstaged-Änderungen

Wenn der Befehl `git status` zu wage für dich ist und du genau wissen willst, was du geändert hast, kannst du den Befehl `git diff` verwenden. Wir werden `git diff` später ausführlicher behandeln, aber du wirst es wahrscheinlich oft verwenden, um dir diese beiden Fragen zu beantworten: Was

hat sich geändert, ist aber noch nicht zum Commit vorgemerkt (engl. staged)? Was hast du zum Commit vorgemerkt und wird demnächst committet? Der Befehl `git status` beantwortet diese Fragen ganz allgemein, indem er die Dateinamen auflistet; `git diff` zeigt dir aber genau die hinzugefügten und entfernten Zeilen – sozusagen den aktuellen Patch.

Nehmen wir an, du bearbeitest und stagst die Datei `README` nochmal und bearbeitest dann die Datei `CONTRIBUTING.md`, ohne sie zu stagern. Wenn du den Befehl `git status` ausführst, siehst du erneut so etwas:

```
$ 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
```

Um die Änderungen zu sehen, die du noch nicht zum Commit vorgemerkt hast, gibst du den Befehl `git diff` ohne weitere Argumente, ein:

```
$ 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
```

Dieses Kommando vergleicht, was sich in deinem Arbeitsverzeichnis befindet, mit dem, was sich in deiner Staging-Area befindet. Das Ergebnis gibt dir an, welche Änderungen du vorgenommen hast, die noch nicht gestaged sind.

Wenn du wissen willst, was du zum Commit vorgemerkt hast, das in deinem nächsten Commit einfließt, kannst du `git diff --staged` verwenden. Dieser Befehl vergleicht deine zum Commit

vorgemerkt Änderungen mit deinem letzten 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
```

Es ist wichtig zu wissen, dass `git diff` von sich aus nicht alle Änderungen seit deinem letzten Commit anzeigt – nur die Änderungen, die noch „unstaged“ sind. Wenn du alle deine Änderungen bereits „gestaged“ hast, wird `git diff` dir keine Antwort geben.

Ein weiteres Beispiel: Wenn du die Datei `CONTRIBUTING.md` stagst und dann bearbeitest, kannst du `git diff` verwenden, um die Änderungen in der Datei anzuzeigen, die staged und nicht staged sind. Wenn es bei dir so aussieht:

```
$ 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
```

kannst du jetzt mit `git diff` sehen, was noch nicht zum Commit vorgemerkt ist

```
$ 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
```

und `git diff --cached` zeigt an, was du bisher zum Commit vorgemerkt hast (`--staged` und `--cached` sind Synonyme, sie bewirken das Gleiche):

```
$ 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 mit einem externen Tool*

Wir werden den Befehl `git diff` im weiteren Verlauf des Buches auf vielfältige Weise verwenden. Es gibt eine weitere Methode, diese Diffs zu betrachten, solltest du lieber ein graphisches oder externes Diff-Viewing-Programm bevorzugen. Wenn du `git difftool` anstelle von `git diff` verwendest, kannst du alle diese Unterschiede in einer Software wie emerge, vimdiff und viele andere (einschließlich kommerzieller Produkte) anzeigen lassen. Führe einfach den Befehl `git difftool --tool-help` aus, um zu sehen, was auf deinem System verfügbar ist.



## Die Änderungen committen

Nachdem deine Staging-Area nun so eingerichtet ist, wie du es wünschst, kannst du deine Änderungen committen. Denke daran, dass alles, was noch nicht zum Commit vorgemerkt ist – alle Dateien, die du erstellt oder geändert hast und für die du seit deiner Bearbeitung nicht mehr `git add` ausgeführt hast – nicht in diesen Commit einfließen werden. Sie bleiben aber als geänderte Dateien auf deiner Festplatte erhalten. In diesem Beispiel nehmen wir an, dass du beim letzten Mal, als du `git status` ausgeführt hast, gesehen hast, dass alles zum Commit vorgemerkt wurde und du bereit bist, alle deine Änderungen zu committen. Am einfachsten committen man, wenn `git commit` eingegeben wird:

```
$ git commit
```

Dadurch wird der Editor deiner Wahl gestartet.

 Das wird durch die Umgebungsvariable `EDITOR` deiner Shell festgelegt – normalerweise Vim oder Emacs. Du kannst den Editor aber auch mit dem Befehl `git config --global core.editor` beliebig konfigurieren, wie du es in Kapitel 1, [Erste Schritte](#) gesehen hast.

Der Editor zeigt den folgenden Text an (dieses Beispiel ist eine Vim-Ansicht):

```
# 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
```

Du kannst erkennen, dass die Standard-Commit-Meldung die neueste Ausgabe des auskommentierten Befehls `git status` und eine leere Zeile darüber enthält. Du kannst diese Kommentare entfernen und deine eigene Commit-Nachricht eingeben oder du kannst sie dort stehen lassen, damit du dir merken kannst, was du committed hast.

 Für eine noch bessere Gedächtnisstütze über das, was du geändert hast, kannst du die Option `-v` an `git commit` übergeben. Dadurch wird auch die Änderung in den Editor geschrieben, so dass du genau sehen kannst, welche Änderungen du committest.

Wenn du den Editor verlässt, erstellt Git deinen Commit mit dieser Commit-Nachricht (mit den Kommentaren und ausgeblendetem Diff).

Alternativ kannst du deine Commit-Nachricht auch inline mit dem Befehl `commit -m` eingeben. Das Flag `-m` ermöglicht die direkte Eingabe eines Kommentartextes:

```
$ 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
```

Jetzt hast du deinen ersten Commit erstellt! Du kannst sehen, dass der Commit eine Nachricht über sich selbst ausgegeben hat: in welchen Branch du committet hast (`master`), welche SHA-1-Prüfsumme der Commit hat (`463dc4f`), wie viele Dateien geändert wurden und Statistiken über hinzugefügte und entfernte Zeilen im Commit.

Denke daran, dass der Commit den Snapshot aufzeichnet, den du in deiner Staging-Area eingerichtet hast. Alles, was von dir nicht zum Commit vorgemerkt wurde, ist weiterhin verfügbar. Du kannst einen weiteren Commit durchführen, um es zu deiner Historie hinzuzufügen. Jedes Mal, wenn du einen Commit ausführst, zeichnest du einen Schnappschuss deines Projekts auf, auf den du zurückgreifen oder mit dem du später vergleichen kannst.

## Die Staging-Area überspringen

Obwohl es außerordentlich nützlich sein kann, Commits so zu erstellen, wie du es wünschst, ist die Staging-Area manchmal etwas komplexer, als du es für deinen Workflow benötigst. Wenn du die Staging-Area überspringen möchten, bietet Git eine einfache Kurzform. Durch das Hinzufügen der Option `-a` zum Befehl `git commit` wird jede Datei, die bereits vor dem Commit versioniert war, automatisch von Git zum Commit staged, so dass du den Teil `git add` überspringen kannst:

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

    modified:   CONTRIBUTING.md

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

Beachte, dass du in diesem Fall `git add` nicht für die Datei `CONTRIBUTING.md` ausführen musst, bevor du committest. Das liegt daran, dass das `-a`-Flag alle geänderten Dateien einschließt. Das ist bequem. Aber sei vorsichtig, manchmal führt dieses Flag dazu, dass du ungewollte Änderungen vornimmst.

## Dateien löschen

Um eine Datei aus Git zu entfernen, musst du sie aus der Versionsverwaltung entfernen (genauer gesagt, aus deinem Staging-Bereich löschen) und dann committen. Der Befehl `git rm` erledigt das und entfernt die Datei auch aus deinem Arbeitsverzeichnis, so dass du sie beim nächsten Mal nicht mehr als „untracked“-Datei siehst.

Wenn du die Datei einfach aus deinem Arbeitsverzeichnis entfernst, erscheint sie unter dem „Changes not staged for commit“-Bereich (das ist die *unstaged*-Area) deiner `git status` Ausgabe:

```
$ 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")
```

Wenn du dann `git rm` ausführst, wird die Entfernung der Datei zum Commit vorgemerkt:

```
$ 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
```

Wenn du das nächste Mal einen Commit ausführst, ist die Datei weg und ist nicht mehr versioniert (engl. tracked). Wenn du die Datei geändert oder bereits zur Staging-Area hinzugefügt hast, musst du das Entfernen mit der Option `-f` erzwingen. Hierbei handelt es sich um eine Sicherheitsfunktion, die ein versehentliches Entfernen von Dateien verhindert, die noch nicht in einem Snapshot aufgezeichnet wurden und die nicht von Git wiederhergestellt werden können.

Eine weitere nützliche Sache, die du möglicherweise nutzen möchtest, ist, die Datei in deinem Verzeichnisbaum zu behalten, sie aber aus deiner Staging-Area zu entfernen. Mit anderen Worten, du kannst die Datei auf deiner Festplatte behalten, aber nicht mehr von Git protokollieren/versionieren lassen.

Das ist besonders dann nützlich, wenn du vergessen hast, etwas zu deiner Datei `.gitignore` hinzuzufügen und dies dann versehentlich gestaged hast (eine große Logdatei z.B. oder eine Reihe von .a-kompilierten Dateien). Das erreichst du mit der Option `--cached`:

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

Du kannst Dateien, Verzeichnisse und Platzhalter-Zeichen an den Befehl `git rm` übergeben. Das bedeutet, dass du folgende Möglichkeit hast:

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

Beachten den Backslash (\) vor dem \*. Der ist notwendig, weil Git zusätzlich zur Dateinamen-Erweiterung deiner Shell eine eigene Dateinamen-Erweiterung vornimmt. Mit dem Befehl oben werden alle Dateien entfernt, die die Erweiterung `.log` im Verzeichnis `log/` haben. Oder du kannst Folgendes ausführen:

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

Dieses Kommando entfernt alle Dateien, deren Name mit `~` endet.

## Dateien verschieben

Im Gegensatz zu vielen anderen VCS-Systemen verfolgt (engl. track) Git das Verschieben von Dateien nicht ausdrücklich. Wenn du eine Datei in Git umbenennst, werden keine Metadaten in Git gespeichert, die dem System mitteilen, dass du die Datei umbenannt hast. Allerdings ist Git ziemlich clever, das im Nachhinein herauszufinden – wir werden uns etwas später mit der Erkennung von Datei-Verschiebungen befassen.

Daher ist es etwas verwirrend, dass Git einen Befehl `mv` vorweisen kann. Wenn du eine Datei in Git umbenennen möchten, kannst du beispielsweise Folgendes ausführen:

```
$ git mv file_from file_to
```

Das funktioniert gut. Tatsache ist, wenn du so einen Befehl ausführst und dir den Status ansiehst, wirst du sehen, dass Git es für eine umbenannte Datei hält:

```
$ 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
```

Unabhängig davon, ist dieser Befehl zu dem Folgenden gleichwertig:

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

Git erkennt, dass es sich um eine umbenannte Datei handelt, so dass es egal ist, ob du eine Datei auf diese Weise oder mit dem Befehl `mv` umbenennst. Der alleinige, reale Unterschied ist, dass `git mv` ein einziger Befehl ist statt deren drei – es ist eine Komfortfunktion. Wichtiger ist, dass du jedes beliebige Tool verwenden können, um eine Datei umzubenennen und das du `add/rm` später aufrufen kannst, bevor du committest.

## Anzeigen der Commit-Historie

Nachdem du mehrere Commits erstellt hast, oder wenn du ein Repository mit einer bestehenden Commit-Historie geklont hast, wirst du wahrscheinlich zurückschauen wollen, um zu erfahren, was

geschehen ist. Das wichtigste und mächtigste Werkzeug dafür ist der Befehl `git log`.

Diese Beispiele verwenden ein sehr einfaches Projekt namens „simplegit“. Um das Projekt zu erstellen, führe diesen Befehl aus:

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

Wenn du `git log` in diesem Projekt aufrufst, solltest du eine Ausgabe erhalten, die ungefähr so aussieht:

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

Change version number

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

Remove unnecessary test

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

Initial commit

Standardmäßig, d.h. ohne Argumente, listet `git log` die in diesem Repository vorgenommenen Commits in umgekehrter chronologischer Reihenfolge auf, d.h. die neuesten Commits werden als Erstes angezeigt. Wie du sehen kannst, listet dieser Befehl jeden Commit mit seiner SHA-1-Prüfsumme, dem Namen und der E-Mail-Adresse des Autors, dem Erstellungs-Datum und der Commit-Beschreibung auf.

Eine Vielzahl von Optionen stehen für den Befehl `git log` zur Verfügung, um dir exakt das anzuseigen, wonach du suchst. Hier zeigen wir dir einige der gängigsten.

Eine der hilfreichsten Optionen ist `-p` oder `--patch`. Sie zeigt die Änderungen (die *patch*-Auszug) an, die bei jedem Commit durchgeführt wurden. Du kannst auch die Anzahl der anzugezeigenden Protokolleinträge begrenzen, z.B. mit `-2` werden nur die letzten beiden Einträge dargestellt.

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

Change version number

```

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

```

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

Author: Scott Chacon <schacon@gee-mail.com>

Date: Sat Mar 15 16:40:33 2008 -0700

Remove unnecessary test

```

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

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

```

Diese Option zeigt die gleichen Informationen an, jedoch mit der diff Ausgabe direkt nach jedem Eintrag. Dies ist sehr hilfreich für die Codeüberprüfung oder um schnell zu durchsuchen, was während einer Reihe von Commits passiert ist, die ein Teammitglied hinzugefügt hat. Du kannst auch mehrere Optionen zur Verdichtung der Ausgabe von `git log` verwenden. Wenn du beispielsweise einige gekürzte Statistiken für jeden Commit sehen möchtest, kannst du die Option `--stat` verwenden:

```

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

```

Change version number

```

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

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

    Remove unnecessary test

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

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

Initial commit

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

```

Wie du sehen kannst, gibt die Option `--stat` unter jedem Commit-Eintrag eine Liste der geänderten Dateien aus, wie viele Dateien geändert wurden und wie viele Zeilen in diesen Dateien hinzugefügt und entfernt wurden. Sie enthält auch eine Zusammenfassung am Ende des Berichts.

Eine weitere wirklich nützliche Option ist `--pretty`. Diese Option ändert das Format der Log-Ausgabe in ein anderes als das Standard-Format. Dir stehen einige vorgefertigte Optionswerte zur Verfügung. Der Wert `oneline` für diese Option gibt jeden Commit in einer einzigen Zeile aus, was besonders nützlich ist, wenn du dir viele Commits ansiehst. Darüber hinaus zeigen die Werte `short`, `full` und `fuller` die Ausgabe im etwa gleichen Format, allerdings mit mehr oder weniger Informationen an:

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

Der interessanteste Wert ist `format`, mit dem du dein eigenes Log-Ausgabeformat festlegen kannst. Dieses Verfahren ist besonders nützlich, wenn du Ausgaben für das maschinelle Parsen generierst – da du das Format explizit angibst, weißt du, dass es sich mit Updates von Git nicht ändert:

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

Wichtige Spezifikatoren für `git log --pretty=format` listet einige der nützlichsten Spezifikatoren auf, die `format` bietet.

Table 1. Wichtige Spezifikatoren für `git log --pretty=format`

Spezifikator	Beschreibung der Ausgabe
<code>%H</code>	Commit-Hash
<code>%h</code>	gekürzter Commit-Hash
<code>%T</code>	Hash-Baum
<code>%t</code>	gekürzter Hash-Baum
<code>%P</code>	Parent-Hashes
<code>%p</code>	gekürzte Parent-Hashes
<code>%an</code>	Name des Autors
<code>%ae</code>	E-Mail-Adresse des Autors
<code>%ad</code>	Erstellungs-Datum des Autors (Format berücksichtigt <code>--date=option</code> )
<code>%ar</code>	relatives Erstellungs-Datum des Autors
<code>%cn</code>	Name des Committers
<code>%ce</code>	E-Mail-Adresse des Committers
<code>%cd</code>	Erstellungs-Datum des Committers
<code>%cr</code>	relatives Erstellungs-Datum des Committers
<code>%s</code>	Thema (engl. Subject)

Du fragst dich vielleicht, worin der Unterschied zwischen *Autor* und *Committer* besteht. Der Autor ist die Person, die das Werk ursprünglich geschrieben hat, während der Committer die Person ist, die das Werk zuletzt bearbeitet hat. Wenn du also einen Patch an ein Projekt sendest und eines der Core-Mitglieder den Patch einbindet, erhalten beide die Anerkennung – du als Autor und das Core-Mitglied als Committer. Wir werden diese Unterscheidung näher in Kapitel 5, [Verteiltes Git](#) erläutern.

Die Optionswerte `oneline` und `format` sind vor allem bei einer anderen `log` Option mit der Bezeichnung `--graph` hilfreich. Diese Option fügt ein schönes kleines ASCII-Diagramm hinzu, das deinen Branch und den Merge-Verlauf zeigt:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 Ignore errors from SIGCHLD on trap
* 5e3ee11 Merge branch 'master' of https://github.com/dustin/grit
|\
| * 420eac9 Add method for getting the current branch
* | 30e367c Timeout code and tests
* | 5a09431 Add timeout protection to grit
* | e1193f8 Support for heads with slashes in them
|/
* d6016bc Require time for xmlschema
```

```
* 11d191e Merge branch 'defunkt' into local
```

Dieser Ausgabetyp wird immer interessanter, wenn wir im nächsten Kapitel über das Branching und Merging sprechen.

Das sind nur einige einfache Optionen zur Ausgabe-Formatierung von `git log` – es gibt noch viele mehr. [Allgemeine Optionen für git log](#) listet die bisher von uns behandelten Optionen auf, sowie einige andere gängige Format-Optionen, die sinnvoll sein können, um die Ausgabe des log-Befehls zu ändern.

Table 2. Allgemeine Optionen für `git log`

Option	Beschreibung
<code>-p</code>	Zeigt den Patch (bzw. Änderung) an, der mit den jeweiligen Commits eingefügt wurde.
<code>--stat</code>	Anzeige der Statistiken für Dateien, die in den einzelnen Commits geändert wurden.
<code>--shortstat</code>	Anzeige der geänderten/eingefügten/gelöschten Zeilen des Befehls <code>--stat</code> .
<code>--name-only</code>	Listet nach den Commit-Informationen die geänderten Dateien auf
<code>--name-status</code>	Listet die Dateien mit hinzugefügten/geänderten/gelöschten Informationen auf.
<code>--abbrev-commit</code>	Zeigt nur die ersten paar Zeichen der SHA-1-Prüfsumme an, nicht aber alle 40.
<code>--relative-date</code>	Zeigt das Datum in einem relativen Format an (z.B. „vor 2 Wochen“), anstatt das volle Datumsformat zu verwenden.
<code>--graph</code>	Zeigt neben der Historie ein ASCII-Diagramm des Branch- und Zusammenführungsverlaufs an.
<code>--pretty</code>	Zeigt Commits in einem anderen Format an. Zu den Optionswerten gehören <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> und <code>format</code> (womit du dein eigenes Format angeben kannst).
<code>--oneline</code>	Kurzform für die gleichzeitige Verwendung von <code>--pretty=oneline</code> und <code>--abbrev-commit</code> .

## Einschränken der Log-Ausgabe

Zusätzlich zu den Optionen für die Ausgabe-Formatierung bietet `git log` eine Reihe nützlicher einschränkender Optionen, d.h. Optionen, mit denen du nur eine Teilmenge von Commits anzeigen kannst. Du hast eine solche Option bereits gesehen – die Option `-2`, die nur die letzten beiden Commits anzeigt. Du kannst die Option `-<n>` verwenden, wobei `n` eine beliebige ganze Zahl ist, um die letzten `n` Commits anzuzeigen. In der Praxis wirst du das selten verwenden, da Git standardmäßig alle Ausgaben über einen Pager leitet, so dass du immer nur eine Seite der Log-Ausgabe siehst.

Im Gegensatz dazu sind zeitbeschränkenden Optionen wie `--since` und `--until` sehr nützlich. Dieser Befehl ruft z.B. die Liste der in den letzten beiden Wochen durchgeföhrten Commits ab:

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

Dieser Befehl funktioniert mit vielen Formaten. Du kannst ein bestimmtes Datum wie "2008-01-15" angeben, oder ein relatives Datum wie "vor 2 Jahren 1 Tag 3 Minuten".

Du kannst die Liste auch nach Commits filtern, die bestimmten Suchkriterien entsprechen. Mit der Option `--author` kannst du nach einem bestimmten Autor filtern und mit der Option `--grep` kannst du nach Schlüsselwörtern in den Übertragungsmeldungen suchen.



Du kannst mehr als eine Instanz der Suchkriterien `--author` und `--grep` angeben, was die Commit-Ausgabe auf Commits beschränkt, die *jedem* der `--author` Muster und *jedem* der `--grep` Muster entsprechen; durch Hinzufügen der Option `--all-match` wird die Ausgabe jedoch weiter auf diejenigen Commits beschränkt, die *allen* `--grep` Mustern entsprechen.

Ein weiterer wirklich hilfreicher Filter ist die Option `-S` (umgangssprachlich als Git's „Pickel“-Option bezeichnet), die eine Zeichenkette empfängt und nur die Commits anzeigt, die die Anzahl der Vorkommen dieser Zeichenkette geändert haben. Wenn du beispielsweise den letzten Commit suchen möchtest, der einen Verweis auf eine bestimmte Funktion hinzugefügt oder entfernt hat, kannst du Folgendes aufrufen:

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

Die letzte hier angesprochene nützliche Option, die als Filter an `git log` übergeben werden kann, ist ein Pfad. Wenn du ein Verzeichnis oder einen Dateinamen angibst, kannst du die Log-Ausgabe auf Commits beschränken, die eine Änderung an diesen Dateien vorgenommen haben. Das ist immer die letzte Option und wird in der Regel durch Doppelstriche (--) eingeleitet, um Pfade von den Optionen zu trennen.`

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

In [Optionen zum Anpassen der Ausgabe von git log](#) sind diese und einige andere gängige Optionen als Referenz aufgelistet.

Table 3. Optionen zum Anpassen der Ausgabe von `git log`

Option	Beschreibung
<code>-&lt;n&gt;</code>	Zeigt nur die letzten n Commits an
<code>--since, --after</code>	Begrenzt die angezeigten Commits auf die, die nach dem angegebenen Datum gemacht wurden.
<code>--until, --before</code>	Begrenzt die angezeigten Commits auf die, die vor dem angegebenen Datum gemacht wurden.
<code>--author</code>	Zeigt nur Commits an, bei denen der Autoren-Eintrag mit der angegebenen Zeichenkette übereinstimmt.

Option	Beschreibung
--committer	Zeigt nur Commits an, bei denen der Committer-Eintrag mit der angegebenen Zeichenkette übereinstimmt.
--grep	Zeigt nur Commits an, deren Commit-Beschreibung die Zeichenkette enthält
-S	Zeigt nur Commits an, die solchen Code hinzufügen oder entfernen, der mit der Zeichenkette übereinstimmt

Wenn du zum Beispiel sehen möchtest, welche der Commits die Testdateien in der Git-Quellcode-Historie ändern, die von Junio Hamano im Monat Oktober 2008 committet wurden und keine Merge-Commits sind, kannst du in etwa folgendes aufrufen:

```
$ 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
```

Von den fast 40.000 Commits in der Git-Quellcode-Historie zeigt dieser Befehl die 6 Commits an, die diesen Kriterien entsprechen.



#### *Die Anzeige von Merge-Commits unterdrücken*

Abhängig von dem in deinem Repository verwendeten Workflow ist es möglich, dass ein beträchtlicher Prozentsatz der Commits in deiner Log-Historie nur Merge-Commits sind, die in der Regel nicht sehr informativ sind. Um zu vermeiden, dass die Anzeige von Merge-Commits deinen Log-Verlauf überflutet, füge einfach die Log-Option `--no-merges` hinzu.

## Ungewollte Änderungen rückgängig machen

Es kommt sicherlich irgendwann der Zeitpunkt, an dem du eine Änderung rückgängig (engl. undo) machen willst. Wir werden hier einige grundlegende Werkzeuge besprechen, mit denen du genau das tun kannst. Sei vorsichtig, man kann diese Aktionen nicht immer rückgängig machen. Das ist einer der wenigen Bereiche in Git, in denen du Arbeit verlieren könntest, wenn du etwas falsch machst.

Eines der häufigsten Undos tritt auf, wenn du zu früh committest und möglicherweise vergessen hast, einige Dateien hinzuzufügen, oder wenn du Fehler in deiner Commit-Nachricht hast. Wenn du diesen Commit wiederholen möchtest, nimm zusätzlichen Änderungen vor, die du vergessen hast, stage Sie und committe erneut mit der Option `--amend`:

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

Dieser Befehl übernimmt deine Staging-Area und verwendet sie für den Commit. Wenn du seit deinem letzten Commit keine Änderungen vorgenommen hast (z.B. du führst diesen Befehl unmittelbar nach deinem vorherigen Commit aus), dann sieht dein Snapshot genau gleich aus; du änderst nur deine Commit-Nachricht.

Der gleiche Commit-Message-Editor wird aufgerufen, enthält aber bereits die Nachricht deines vorherigen Commits. Du kannst die Nachricht wie gewohnt bearbeiten, aber sie überschreibt den vorherigen Commit.

Wenn du beispielsweise einen Commit durchführst und dann feststellst, dass du vergessen hast, eine Datei zu ändern, könntest du Folgendes tun:

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

Du erhältst am Ende einen einzigen Commit – der zweite Commit ersetzt die Ergebnisse des Ersten.

Es ist wichtig zu verstehen: wenn du deinen letzten Commit änderst, korrigierst du ihn nicht. Du *ersetzt* ihn durch einen neuen, verbesserten Commit. Der alte Commit wird entfernt und der neue Commit an seine Stelle gesetzt. Tatsächlich ist es so, als ob der letzte Commit nie stattgefunden hätte und er nicht mehr in deinem Repository-Verlauf auftaucht.



Der naheliegendste Nutzen für die Änderung von Commits besteht darin, kleine Verbesserungen an deinem letzten Commit vorzunehmen, ohne dein Repository-Verlauf mit Commit-Nachrichten der Form „Ups, vergessen, eine Datei hinzuzufügen“ oder „Verdammt, einen Tippfehler im letzten Commit behoben“ zu überladen.



Änderne nur lokale Commits, die noch nicht gepusht wurden. Das Ändern zuvor übertragener Commits und das forcierte pushen des Branches verursacht Probleme bei ihren Mitstreitern. Weitere Informationen darüber, was dabei passiert und wie du es wieder gerade ziehen kannst, wenn du dich auf der Empfängerseite befinden, findest du unter [Die Gefahren des Rebasing](#).

## Eine Datei aus der Staging-Area entfernen

Die nächsten beiden Abschnitte erläutern, wie du mit deiner Staging-Area und den Änderungen des Arbeitsverzeichnisses arbeitest. Der angenehme Nebeneffekt ist, dass der Befehl, mit dem du den Zustand dieser beiden Bereiche bestimmst, dich auch daran erinnert, wie du Änderungen an ihnen rückgängig machen kannst. Nehmen wir zum Beispiel an, du hast zwei Dateien geändert und möchtest sie als zwei separate Änderungen übertragen, aber du gibst versehentlich `git add *` ein und stellst sie dann beide in der Staging-Area bereit. Wie kannst du eine der beiden aus der

Staging-Area entfernen? Der Befehl `git status` meldet:

```
$ 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
```

Direkt unter dem Text „Changes to be committed“, steht, dass man `git reset HEAD <file>...` verwenden soll, um die Staging-Area zu entleeren. Lass uns also diesem Rat folgen und die Datei `CONTRIBUTING.md` aus der Staging-Area entfernen:

```
$ 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
```

Der Befehl klingt etwas merkwürdig, aber er funktioniert. Die Datei `CONTRIBUTING.md` ist modifiziert und wieder im Status unstaged überführt.



Es stimmt, dass `git reset` ein riskanter Befehl sein kann, besonders, wenn du das `--hard` Flag mit gibst. In dem oben beschriebenen Szenario wird die Datei in deinem Arbeitsverzeichnis jedoch nicht angetastet, so dass er relativ sicher ist.

Im Moment ist dieser Aufruf alles, was du über den Befehl `git reset` wissen musst. Wir werden viel ausführlicher darauf eingehen, was `reset` bewirkt und wie man damit umgeht, um wirklich interessante Aufgaben zu erledigen, siehe Kapitel 7 [Git Reset](#).

## Änderung in einer modifizierten Datei zurücknehmen

Was ist, wenn du feststellst, dass du deine Änderungen an der Datei `CONTRIBUTING.md` nicht behalten willst? Wie kannst du sie in den Ursprungszustand zurücksetzen, so wie sie beim letzten Commit ausgesehen hat (oder anfänglich geklont wurde, oder wie auch immer du sie in dein

Arbeitsverzeichnis bekommen hast)? Glücklicherweise sagt dir `git status`, wie du das machen kannst. Im letzten Beispiel sieht die Unstaged-Area so aus:

```
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
```

Git erklärt dir, wie du die von dir vorgenommenen Änderungen verwerfen kannst. Lassen es uns ausführen:

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

Wie du siehst, wurde die Änderungen rückgängig gemacht.

Es ist sehr wichtig zu begreifen, dass `git checkout -- <file>` ein riskanter Befehl ist. Alle lokalen Änderungen, die du an dieser Datei vorgenommen hast, sind verloren – Git hat diese Datei einfach durch die zuletzt committete oder gestagte Version ersetzt. Verwenden diesen Befehl nur, wenn du dir absolut sicher bist, dass du diese nicht gespeicherten lokalen Änderungen nicht benötigst.



Wenn du die Änderungen, die du an dieser Datei gemacht hast, beibehalten möchten, sie aber vorerst aus dem Weg räumen willst, sollten wir das Stashing und Branching in Kapitel 3 – [Git Branching](#) durchgehen; das sind im Allgemeinen die besseren Methoden, um das zu erledigen.

Denke daran, dass alles, was in Git *committet* wird, fast immer wiederhergestellt werden kann. Sogar Commits, die auf gelöschten Branches lagen oder Commits, die mit einem `--amend` Commit überschrieben wurden, können wiederhergestellt werden (siehe Kapitel 10 [Daten-Rettung](#) für das Wiederherstellen der Daten). Aber: alles, was du verworfen und nie committet hast, wirst du wahrscheinlich nie wieder sehen.

## Änderungen mit `git restore` Rückgängig machen

Git Version 2.23.0 führte einen neuen Befehl ein: `git restore`. Es ist im Grunde eine Alternative zu `git reset`, die wir gerade behandelt haben. Ab Git Version 2.23.0 verwendet Git für viele dieser Vorgänge `git restore` anstelle von `git reset`.

Lasse uns unsere Schritte wiederholen und die Dinge mit `git restore` anstelle von `git reset` rückgängig machen.

## Eine Datei mit git restore unstagen

Die nächsten beiden Abschnitte zeigen, wie du an Änderungen in deinem Staging-Bereich und im Arbeitsverzeichnisses mit `git restore` arbeitest. Das Schöne daran ist, dass der Befehl, mit dem du den Status dieser beiden Bereiche bestimmt, dir auch zeigt, wie du Änderungen an ihnen rückgängig machen kannst. Angenommen, du hast zwei Dateien geändert und möchtest sie als zwei separate Änderungen committen. Du gibst jedoch versehentlich `git add *` ein und committest beide. Wie kannst du eine der beiden wieder unstaged? Der Befehl `git status` zeigt folgendes:

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

Direkt unter dem Text „Changes to be committed“ steht `git restore --staged <file> ...` zum unstagen. Verwenden wir diesen Rat, um die Datei `CONTRIBUTING.md` zu unstagen:

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

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

Die Datei `CONTRIBUTING.md` ist modifiziert und wieder im Status unstaged überführt.

## Eine geänderte Datei mit git restore rückgängig machen

Was ist, wenn du merkst, dass du deine Änderungen an der Datei `CONTRIBUTING.md` nicht beibehalten willst? Wie kannst du sie in den Ursprungszustand zurücksetzen, so wie sie beim letzten Commit ausgesehen hat (oder anfänglich geklont wurde, oder wie auch immer du sie in dein Arbeitsverzeichnis bekommen hast)? Glücklicherweise sagt dir `git status`, wie du das machen kannst. Im letzten Beispiel sieht die Unstaged-Area so aus:

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

Git erklärt dir, wie du die von dir vorgenommene Änderungen verwerfen kannst. Lassen es uns ausführen:

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

Es ist wichtig zu verstehen, dass `git restore <file>` ein gefährlicher Befehl ist. Alle lokalen Änderungen, die du an dieser Datei vorgenommen hast, sind weg. Git hat diese Datei durch die zuletzt committete oder gestagte Version ersetzt. Verwende diesen Befehl nur, wenn du dir absolut sicher bist, dass du diese nicht gespeicherten lokalen Änderungen nicht benötigst.



## Mit Remotes arbeiten

Um an Git-Projekte mitarbeiten zu können, musst du wissen, wie du deine Remote-Repositorys verwalten kannst. Remote-Repositorys sind Versionen deines Projekts, die im Internet oder im Netzwerk irgendwo gehostet werden. Du kannst Mehrere davon haben, von denen jedes in der Regel entweder schreibgeschützt oder beschreibbar ist. Die Zusammenarbeit mit Anderen erfordert die Verwaltung dieser Remote-Repositorys sowie das Pushing und Pulling von Daten zu und von den Repositorys, wenn du deine Arbeit teilen möchtest. Die Verwaltung von Remote-Repositorys umfasst das Wissen, wie man entfernte Repositorys hinzufügt, nicht mehr gültige Remotes entfernt, verschiedene Remote-Banches verwaltet, sie als versioniert oder nicht versioniert definiert und vieles mehr. In diesem Abschnitt werden wir einige dieser Remote-Management-Fertigkeiten erörtern.

*Remote-Repositorys können auch auf deinem lokalen Rechner liegen.*



Es ist durchaus möglich, dass du mit einem „entfernten“ Repository arbeiten kannst, das sich eigentlich auf demselben Host befindet auf dem du gerade arbeitest. Das Wort „remote“ bedeutet nicht unbedingt, dass sich das Repository an einem anderen Ort im Netzwerk oder Internet befindet, sondern nur, dass es an einem anderen Ort liegt. Die Arbeit mit einem solchen entfernten Repository würde immer noch alle üblichen Push-, Pull- und Fetch-Operationen einschließen, wie bei jedem anderen Remote-Repository.

## Auflisten der Remotes

Um zu sehen, welche Remote-Server bei dir konfiguriert sind, kannst du den Befehl `git remote` aufrufen. Er listet die Kurznamen der einzelnen von dir festgelegten Remote-Handles auf. Wenn du dein Repository geklont hast, solltest du zumindest `origin` sehen – das ist der Standardname, den Git dem Server gibt, von dem du geklont hast:

```
$ 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
```

Du kannst zusätzlich auch `-v` angeben, das dir die URLs anzeigen, die Git für den Kurznamen gespeichert hat, um beim Lesen und Schreiben auf diesen Remote zuzugreifen:

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

Wenn du mehr als einen Remote hast, listet der Befehl sie alle auf. Ein Repository mit mehreren Remotes für die Arbeit mit mehreren Beteiligten könnte beispielsweise so aussehen.

```
$ 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)
```

Das bedeutet, dass wir Beiträge von jedem dieser Benutzer ziemlich einfach abrufen können. Möglicherweise haben wir zusätzlich die Erlaubnis, auf einen oder mehrere von diesen zu pushen, obwohl wir das hier nicht erkennen können.

Beachte: Diese Remotes verwenden unterschiedliche Protokollen; wir werden mehr darüber hier erfahren: [Git auf einem Server installieren](#).

## Hinzufügen von Remote-Repositorys

Wir haben bereits erwähnt und einige Beispiele gezeigt, wie der Befehl `git clone` stillschweigend den Remote `origin` hinzufügt. Im Folgendem beschreiben wir, wie du einen neuen Remote hinzufügen kannst. Um ein neues Remote-Git-Repository als Kurzname hinzuzufügen, auf das du verweisen kannst, führe `git remote add <shortname> <url>` aus:

```
$ 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)
```

Nun kannst du die Zeichenfolge **pb** auf der Kommandozeile anstelle der gesamten URL verwenden. Wenn du beispielsweise alle Informationen fetchen möchtest, die Paul hat, die aber noch nicht in deinem Repository enthalten sind, kannst du **git fetch pb** ausführen:

```
$ 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
```

Pauls **master** Branch ist nun lokal als **pb/master** erreichbar – Du kannst ihn in eine deiner Branches mergen oder ihn in einen lokalen Branch auschecken, wenn du ihn inspizieren möchten. Wir werden in [Git Branching](#) detailliert beschreiben, was Branches sind und wie man sie nutzt.

## Fetchen und Pullen deiner Remotes

Wie du gerade gesehen hast, kannst du Daten aus einem Remote-Projekt abrufen:

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

Der Befehl geht an das Remote-Projekt und zieht (engl. pull) alle Daten von diesem Remote-Projekt, die du noch nicht hast. Danach solltest du Referenzen auf alle Branches dieses Remote haben, die du jederzeit mergen oder inspizieren kannst.

Wenn du ein Repository klonst, fügt der Befehl dieses entfernte Repository automatisch unter dem Namen „origin“ hinzu. So holt **git fetch origin** alle Änderungen, die seit dem Klonen (oder dem letzten Abholen) auf diesen Server abgelegt (engl. pushed) wurden. Es ist jedoch wichtig zu beachten, dass der Befehl **git fetch** nur die Daten in dein lokales Repository herunterlädt – er merged sie nicht automatisch mit deiner Arbeit zusammen oder ändert das, woran du gerade arbeitest. Du musst das Ganze manuell mit deiner Arbeit zusammenführen, wenn du soweit bist.

Wenn dein aktueller Branch so eingerichtet ist, dass er einen entfernten Branch verfolgt (engl. tracking), kannst du den Befehl **git pull** verwenden, um diesen entfernten Branch automatisch zu fetchen und dann mit deinem aktuellen Branch zu mergen (siehe den nächsten Abschnitt und [Git](#)

[Branching](#) für weitere Informationen). Das könnte ein einfacherer oder komfortablerer Workflow sein. Standardmäßig richtet der Befehl `git clone` deinen lokalen `master` Branch automatisch so ein, dass er den entfernten `master` Branch (oder wie auch immer der Standard-Branch genannt wird) auf dem Server versioniert von dem du geklont hast. Wenn du `git pull` ausführst, werden normalerweise Daten von dem Server abgerufen, von dem du ursprünglich geklont hast. Anschliessend wird automatisch versucht diese Daten in deinem Code zu mergen.

Ab der Version 2.27 von Git wird `git pull` eine Warnung ausgeben, wenn die Variable `pull.rebase` nicht gesetzt ist. Git wird dich so lange warnen, bis du die Variable setzt.



Falls du das Standardverhalten (möglichst ein fast-forward, ansonsten einen Merge-Commit erstellen) von Git beibehalten willst: `git config --global pull.rebase "false"`

Wenn du jedoch mit dem Pullen einen Rebase machen willst: `git config --global pull.rebase "true"`

## Zu deinen Remotes Pushen

Wenn du dein Projekt an einem Punkt hast, an dem du es teilen möchtest, können wir es zum Upstream verschieben (engl. pushen). Der Befehl dafür lautet: `git push <remote> <branch>`. Wenn du deinen `master` Branch auf den `origin` Server pushen möchtest (Zur Erinnerung: das Klonen richtet im Regelfall diese beiden Branches automatisch ein), dann kannst du diesen Befehl auch nutzen, um alle Commits, die du durchgeführt hast, auf den Server zu pushen:

```
$ git push origin master
```

Dieser Befehl funktioniert allerdings nur, wenn du von einem Server geklont hast, auf den du Schreibzugriff hast und wenn in der Zwischenzeit noch niemand anderes gepusht hat. Wenn du und ein anderer Benutzer gleichzeitig klonen und beide Upstream pushen wollen, du aber etwas später nach Upstream pushst, dann wird dein Push zu Recht abgelehnt. Du musst zuerst dessen Bearbeitung abholen und in deine einbinden, bevor du pushen kannst. Siehe Kapitel 3 [Git Branching](#) mit ausführlicheren Details zum Pushen auf Remote-Server.

## Inspizieren eines Remotes

Wenn du mehr Informationen über einen bestimmten Remote sehen möchten, kannst du den Befehl `git remote show <remote>` verwenden. Wenn du diesen Befehl mit einem remote Kurznamen ausführen, wie z.B. `origin`, erhältst du bspw. folgende Meldung:

```
$ 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)

```

Er listet die URL für das Remote-Repository sowie die Informationen zu den Tracking-Banches auf. Der Befehl teilt dir mit: Wenn du dich im Master-Branch befindest und `git pull` ausführst, der `master` Branch des remotes nach dem abrufen (engl. fetched) automatisch mit dem lokalen Branch gemerged wird. Er listet auch alle Remote-Referenzen auf, die er abgerufen hat.

Das ist nur ein einfaches Beispiel, auf das du möglicherweise treffen wirst. Wenn du Git hingegen intensiver verwendest, könnte die `git remote show` Ausgabe wesentlich umfangreicher sein:

```

$ 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)

```

Dieser Befehl zeigt an, zu welchem Branch automatisch gepusht wird, wenn du `git push` ausführst, während du dich in bestimmten Branches befindest. Er zeigt dir auch, welche entfernten Branches auf dem Server vorhanden sind, die du noch nicht hast, welche entfernten Branches du hast, die aber vom Server gelöscht wurden und die lokalen Branches, die automatisch mit deinen Remote-Tracking-Branch zusammengeführt (gemerged) werden können, wenn du `git pull` ausführst.

## Umbenennen und Entfernen von Remotes

Du kannst `git remote rename` ausführen, um den Kurznamen eines Remotes zu ändern. Wenn du beispielsweise `pb` in `paul` umbenennen möchtest, kannst du dies mit dem Befehl `git remote rename`

erreichen:

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

Es ist zu beachten, dass dadurch auch alle deine Remote-Tracking-Branchnamen geändert werden. Was vorher unter `pb/master` referenziert wurde, ist jetzt unter `paul/master` zu finden.

Wenn du einen Remote aus irgendwelchen Gründen entfernen möchten – Du hast den Server verschoben oder verwendest einen bestimmten Mirror nicht länger oder ein Beitragender ist nicht mehr dabei – dann kannst du entweder `git remote remove` oder `git remote rm` verwenden:

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

Sobald du die Referenz auf einen Remote auf diese Weise gelöscht hast, werden auch alle mit diesem Remote verbundenen Remote-Tracking-Banches und Konfigurationseinstellungen gelöscht.

## Taggen

Wie die meisten VCSs hat Git die Möglichkeit, bestimmte Punkte in der Historie eines Repositorys als wichtig zu markieren. Normalerweise verwenden Leute diese Funktionalität, um Releases zu markieren (`v1.0`, `v2.0` usw). In diesem Abschnitt erfährst du, wie bestehende Tags aufgelistet, Tags erstellt und gelöscht werden können sowie was die unterschiedlichen Tag-Typen sind.

### Deine Tags auflisten

Die Auflistung der vorhandenen Tags in Git ist unkompliziert. Gib einfach `git tag` (mit optionalem `-l` oder `--list`) ein:

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

Dieser Befehl listet die Tags in alphabetischer Reihenfolge auf. Die Reihenfolge, in der sie angezeigt werden, hat keine wirkliche Bedeutung.

Du kannst auch nach Tags suchen, die einer bestimmten Zeichenfolge entsprechen. Das Git-Source-Repo zum Beispiel enthält mehr als 500 Tags. Wenn du nur daran interessiert bist, dir die 1.8.5-Serie anzusehen, kannst du Folgendes ausführen:

```
$ 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
```

Das Auflisten von Tag-Wildcards erfordert die Option `-l` oder `--list`

 Wenn du lediglich die gesamte Liste der Tags wünschst, geht die Ausführung des Befehls `git tag` implizit davon aus, dass du eine Auflistung haben willst und gibt sie aus; die Verwendung von `-l` oder `--list` ist in diesem Fall optional.

Wenn du jedoch ein Platzhaltermuster angibst, das mit den Tag-Namen übereinstimmt, ist die Verwendung von `-l` oder `--list` obligatorisch.

## Erstellen von Tags

Git unterstützt zwei Arten von Tags: *lightweight* (d.h. nicht-annotiert) und *annotated*.

Ein nicht-annotiertes Tag ist sehr ähnlich eines Branches, der sich nicht ändert – es ist nur ein Zeiger auf einen bestimmten Commit.

Annotierte Tags werden dagegen als vollständige Objekte in der Git-Datenbank gespeichert. Sie werden mit einer Prüfsumme versehen, enthalten den Tagger-Namen, die E-Mail-Adresse und das Datum, haben eine Tagging-Nachricht und können mit GNU Privacy Guard (GPG) signiert und überprüft werden. Es wird allgemein empfohlen annotierte Tags zu erstellen, damit all diese Informationen gespeichert werden; aber wenn du ein temporäres Tag wünschst oder aus irgendwelchen Gründen die anderen Informationen nicht speichern willst, sind auch nicht-annotierte Tags möglich.

## Annotated Tags

Das Erstellen eines annotierten Tags in Git ist einfach. Der einfachste Weg ist die Eingabe von `-a`, beim Ausführen des `tag` Befehls:

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

Ein `-m` spezifiziert eine Tagging-Nachricht, die mit dem Tag gespeichert wird. Wenn du keine Nachricht für ein annotierten Tag angibst, startet Git deinen Editor, damit du eine Nachricht

eingeben kannst.

Du kannst die Tag-Daten zusammen mit dem getaggen Commit sehen, indem du den Befehl `git show` verwendest:

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

my version 1.4

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

    Change version number
```

Es werden die Tagger-Informationen, das Datum an dem der Commit getaggt wurde, und die Annotationsnachricht angezeigt, gefolgt von den Commit-Informationen.

## Lightweight Tags

Eine weitere Möglichkeit, Commits zu markieren, ist ein leichtgewichtiger, nicht-annotierter Tag. Das ist im Grunde genommen die in einer Datei gespeicherte Commit-Prüfsumme – es werden keine weiteren Informationen gespeichert. Um einen leichtgewichtigen Tag zu erstellen, gib keine der Optionen `-a`, `-s` oder `-m` an, sondern nur einen Tag-Namen:

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

Wenn du diesmal `git show` auf dem Tag ausführst, siehst du keine zusätzlichen Tag-Informationen. Der Befehl zeigt nur den Commit an:

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

    Change version number
```

## Nachträgliches Tagging

Du kannst auch ältere Commits markieren. Angenommen, dein Commit-Verlauf sieht so aus:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 Create write support
0d52aaab4479697da7686c15f77a3d64d9165190 One more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbe Add commit function
4682c3261057305bdd616e23b64b0857d832627b Add todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a Create write support
9fce02d0ae598e95dc970b74767f19372d61af8 Update rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc Commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a Update readme
```

Nehmen wir an, du hast vergessen, das Projekt mit v1.2 beim Commit von „Update rakefile“ zu taggen. Du kannst ihn nachträglich hinzufügen. Um diesen Commit zu markieren, gib am Ende des Befehls die Commit-Prüfsumme (oder einen Teil davon) an:

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

Du siehst, dass du den Commit getaggt hast:

```
$ 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

    Update rakefile
...
```

## Tags teilen

Normalerweise überträgt der Befehl `git push` keine Tags an den Remote-Server. Du musst Tags explizit auf einen Server verschieben, nachdem du sie erstellt hast. Dieser Prozess funktioniert genauso wie das Teilen von Remote-Banches – Du musst dazu `git push origin <tagname>` ausführen.

```
$ 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
```

Wenn du viele Tags hast, die du auf einmal pushen willst, kannst du auch die Option `--tags` mit dem Befehl `git push` verwenden. Dadurch werden alle deine Tags auf den Remote-Server übertragen, die sich noch nicht auf dem Server befinden.

```
$ 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
```

Wenn jetzt jemand anderes aus deinem Repository klont oder pullt, erhält er auch alle deine Tags.

`git push` pusht beide Arten von Tags



`git push <remote> --tags` wird sowohl Lightweight- als auch Annotated-Tags pushen. Es gibt zur Zeit keine Möglichkeit, nur Lightweight-Tags zu pushen, aber wenn Sie `git push <remote> --follow-tags` verwenden, werden nur annotierte Tags an den Remote gepusht.

## Tags löschen

Um einen Tag aus dem lokalen Repository zu löschen, verwende `git tag -d <tagname>`. Wir könnten beispielsweise den leichtgewichtigen Tag wie folgt entfernen:

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

Beachte, dass dadurch der Tag nicht von Remote-Servern entfernt wird. Es gibt zwei gängige Varianten, um ein Tag von einem remote Server zu löschen.

Die erste Möglichkeit ist `git push <remote> :refs/tags/<tagname>`:

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

Zur Erklärung des obigen Befehls: Vor dem Doppelpunkt ist nichts und somit als Nullwert zu lesen. Darauf wird der Remote-Tag-Name gepusht, wodurch dieser gelöscht wird.

Der zweite, intuitivere Weg, ein Remote-Tag zu löschen, ist mit:

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

## Tags auschecken

Wenn du die Dateiversion anzeigen möchtest, auf die ein bestimmter Tag zeigt, kannst du `git checkout` auf dieses Tag durchführen. Dies versetzt dein Repository in den Zustand „detached HEAD“ (dt. losgelöst), was einige negative Nebenwirkungen hat:

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

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

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

```
git switch -c <new-branch-name>
```

Or undo this operation with:

```
git switch -
```

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

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

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

Wenn du im Zustand „detached HEAD“ Änderungen wornimmst und dann einen Commit erstellst, bleibt der Tag gleich, aber dein neuer Commit gehört zu keinem Branch und ist unzugänglich, außer mit dem genauen Commit-Hash. Wenn du also Änderungen vornehmen musst – z.B. wenn du

einen Fehler in einer älteren Version behebst – wirst du normalerweise einen Branch erstellen:

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

Wenn du das tust und einen Commit erstellst, wird sich dein Branch **version2** leicht von deinem Tag **v2.0.0** unterscheiden, da er mit deinen neuen Änderungen fortschreitet, sei also vorsichtig.

## Git Aliases

Bevor wir dieses Kapitel über Basic Git abschließen, gibt es noch einen kurzen Tipp, der deine Arbeit mit Git einfacher, leichter und verständlicher machen kann: Aliase. Der Klarheit halber werden wir sie nirgendwo anders in diesem Buch verwenden, aber wenn du Git in Zukunft regelmäßig verwendest, dann sind Aliase etwas, das du kennen solltest.

Git erkennt nicht automatisch deinen abgesetzten Befehl, wenn du ihn nur teilweise eingibst. Wenn du nicht den gesamten Text jedes Git-Befehls eingeben möchtest, kannst du mit Hilfe von **git config** einfach ein Alias für jeden Befehl einrichten. Hier sind ein paar Beispiele, die du einrichten sollten:

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

Das bedeutet, dass du z.B. anstelle von **git commit** einfach **git ci** eingeben kannst. Je mehr du Git verwendest, wirst du vermutlich noch andere Befehle häufiger verwenden; scheue dich nicht, neue Aliase zu erstellen.

Diese Technik kann auch sehr nützlich sein, um Befehle zu erstellen, von denen du glaubst, dass sie vorhanden sein sollten. Um beispielsweise ein Usability-Problem zu beheben, auf das du beim Entfernen einer Datei aus der Staging-Area stößt, kannst du Git deinen eigenen Unstage-Alias hinzufügen:

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

Dadurch sind die folgenden beiden Befehle gleichwertig:

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

Das macht denn Sinn des aliasing hoffentlich klarer. Es ist auch üblich, einen **last** (dt. letzten) Befehl hinzuzufügen, so wie hier:

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

Auf diese Weise kannst du den letzten Commit leicht auffinden:

```
$ 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>
```

Wie du siehst, ersetzt Git den neuen Befehl einfach durch den Alias, den du ihm geben hast. Vielleicht möchtest du jedoch eher einen externen Befehl als einen Git-Subbefehl ausführen. In diesem Fall starte den Befehl mit einem Ausrufezeichen (!). Das ist hilfreich, wenn du deine eigenen Tools schreibst, die mit einem Git-Repository arbeiten. Hier ein Beispiel, in dem wir `git visual` mit `gitk` aliasen:

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

## Zusammenfassung

Du solltest jetzt in der Lage sein, die wichtigsten Git-Befehle einsetzen zu können. Folgendes sollte Dir jetzt gelingen: Erzeugen oder Klonen eines Repositorys, Änderungen vornehmen und zur Staging-Area hinzufügen, Commits anlegen und die Historie aller Commits in einem Repository durchsuchen. Als nächstes werden wir uns mit der „Killer-Funktion“ von Git befassen: dem Branching-Modell.

# Git Branching

Nahezu jedes VCS unterstützt eine Form von Branching. Branching bedeutet, dass du von der Hauptlinie der Entwicklung abzweigen und deine Arbeit fortsetzen kannst, ohne die Hauptlinie durcheinanderzubringen. In vielen VCS-Tools ist das ein etwas aufwändiger Prozess, bei dem du oft eine neue Kopie deines Quellcode-Verzeichnisses erstellen musst, was bei großen Projekten viel Zeit in Anspruch nehmen kann.

Manche Leute bezeichnen Gits Branching-Modell als dessen „Killer-Feature“, was Git zweifellos vom Rest der VCS-Community abhebt. Was ist das Besondere daran? Die Art und Weise, wie Git Branches anlegt ist unglaublich leichtgewichtig. Branch-Operationen werden nahezu verzögerungsfrei ausgeführt und auch das Hin- und Herschalten zwischen einzelnen Entwicklungszweigen läuft meist genauso schnell ab. Im Gegensatz zu anderen VCS ermutigt Git zu einer Arbeitsweise mit häufigem Branching und Merging, sogar mehrmals am Tag. Wenn du diese Funktion verstehst und beherrschst, besitzt du ein mächtiges und besonderes Werkzeug, welches deine Art zu entwickeln vollständig verändern kann.

## Branches auf einen Blick

Um richtig zu verstehen, wie Git das Verzweigen (engl. Branching) realisiert, müssen wir einen Schritt zurück gehen und untersuchen, wie Git seine Daten speichert.

Wie du vielleicht aus Kapitel 1 [Was ist Git?](#) in Erinnerung hast, speichert Git seine Daten nicht als Serie von Änderungen oder Differenzen, sondern statt dessen als eine Reihe von *Snapshots*.

Wenn du einen Commit durchführst, speichert Git ein Commit-Objekt, das einen Zeiger auf den Snapshot des von dir gestagten Inhalts enthält. Dieses Objekt enthält auch den Namen und die E-Mail-Adresse des Autors, die Nachricht, die er eingegeben hat, und zeigt auf den Commit oder die Commits, die direkt vor diesem Commit stattfanden (zu seinem Vorgänger bzw. seinen Vorgängern): keine Vorgänger für den ersten Commit, einen Vorgänger für einen normalen Commit und mehrere Vorgänger für einen Commit, welcher aus dem Zusammenführen (engl. *mergen*) von zwei oder mehr Branches resultiert.

Um das zu veranschaulichen, lass uns annehmen, du hast ein Verzeichnis, welches drei Dateien enthält, und du fügst alle Dateien zur Staging-Area hinzu und führst einen Commit durch. Durch das Hinzufügen der Dateien zur Staging-Area erzeugt Git für jede Datei eine Prüfsumme (den SHA-1-Hashwert, den wir in Kapitel 1 [Was ist Git?](#) erwähnt haben), speichert diese Version der Datei im Git-Repository (Git verweist auf diese als *blobs*) und fügt die Prüfsumme der Staging-Area hinzu:

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

Wenn du mit der Anweisung `git commit` einen Commit erzeugst, berechnet Git für jedes Unterverzeichnis (in diesem Fall nur das Wurzelverzeichnis des Projektes) eine Prüfsumme und speichert diese als *tree*-Objekt im Git-Repository. Git erzeugt dann ein *commit*-Objekt, welches die Metadaten und einen Zeiger zum *tree*-Objekt des Wurzelverzeichnisses enthält, sodass es bei Bedarf den Snapshot erneut erzeugen kann.

Dein Git-Repository enthält jetzt fünf Objekte: drei *blobs* (die jeweils den Inhalt einer der drei Dateien repräsentieren), ein *tree*-Objekt, welches den Inhalt des Verzeichnisses auflistet und angibt, welcher Dateiname zu welchem Blob gehört, und ein *commit*-Objekt mit dem Zeiger, der auf die Wurzel des Projektbaumes und die Metadaten des Commits verweist.

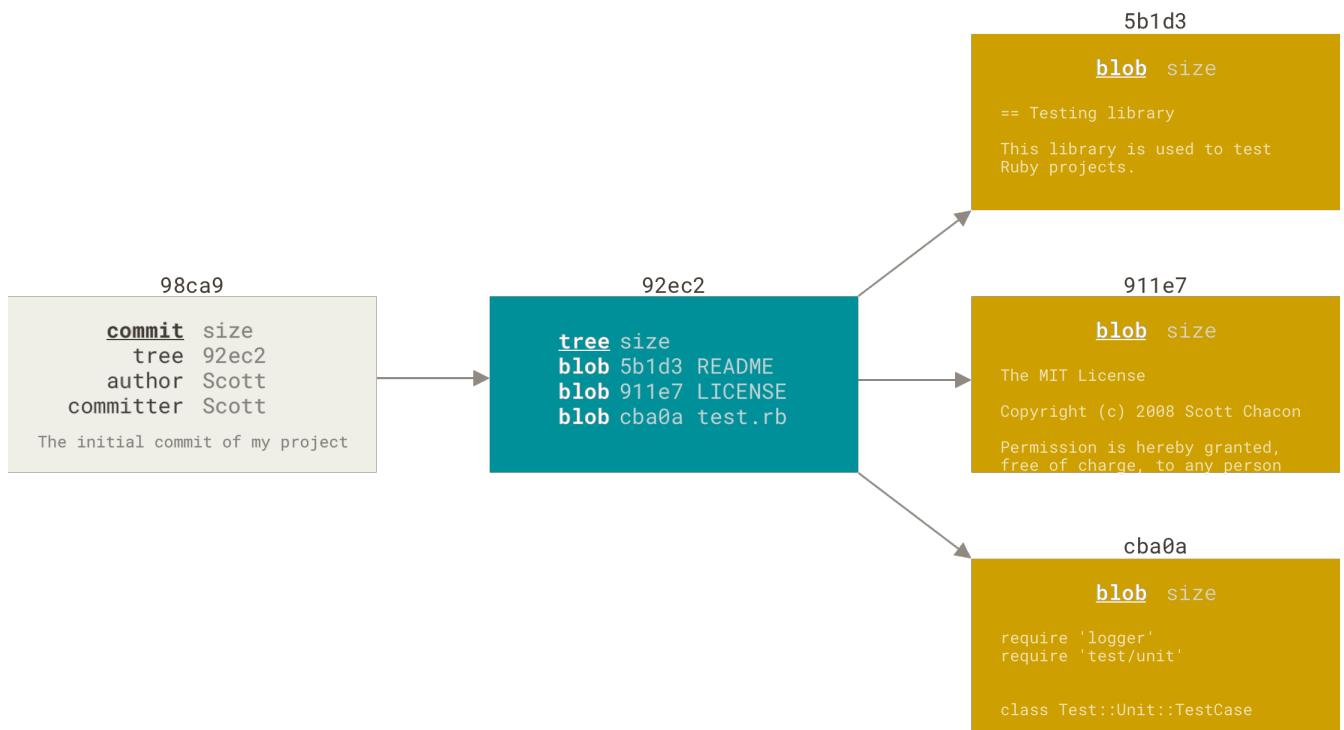


Figure 9. Ein Commit und sein Tree

Wenn du einige Änderungen vornimmst und wieder einen Commit durchführst, speichert dieser einen Zeiger zu dem Commit, der unmittelbar davor gemacht wurde.

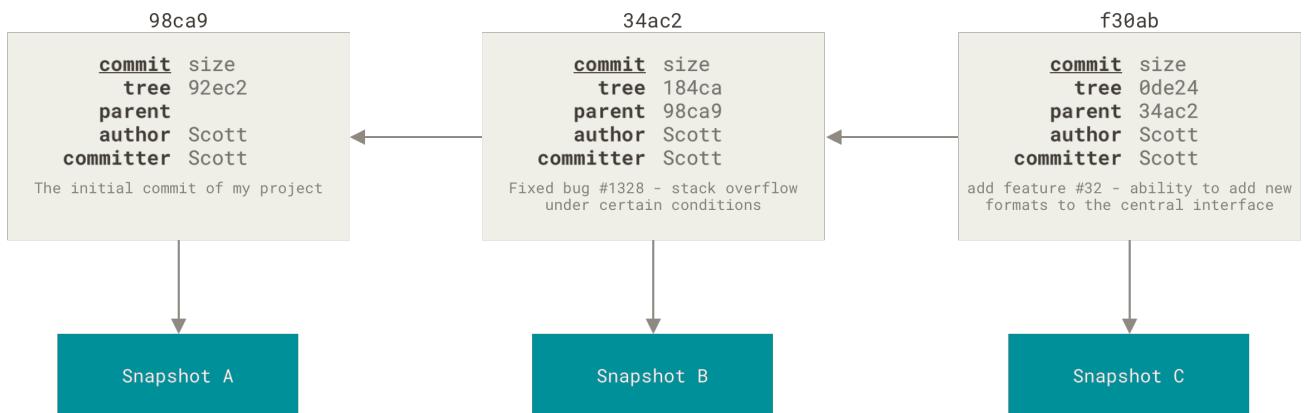


Figure 10. Commits und ihre Vorgänger

Ein Branch in Git ist einfach ein leichter, beweglicher Zeiger auf einen dieser Commits. Die Standardbezeichnung für einen Branch bei Git lautet `master`. Wenn du damit beginnst, Commits durchzuführen, erhältst du einen `master` Branch, der auf den letzten Commit zeigt, den du gemacht hast. Jedes Mal, wenn du einen Commit durchführst, bewegt er sich automatisch vorwärts.



Der „`master`“-Branch in Git ist kein spezieller Branch. Er ist genau wie jeder andere Branch. Der einzige Grund dafür, dass nahezu jedes Repository einen „`master`“-Branch hat, ist der Umstand, dass die Anweisung `git init` diesen

standardmäßig erzeugt und die meisten Leute sich nicht darum kümmern, den Namen zu ändern.

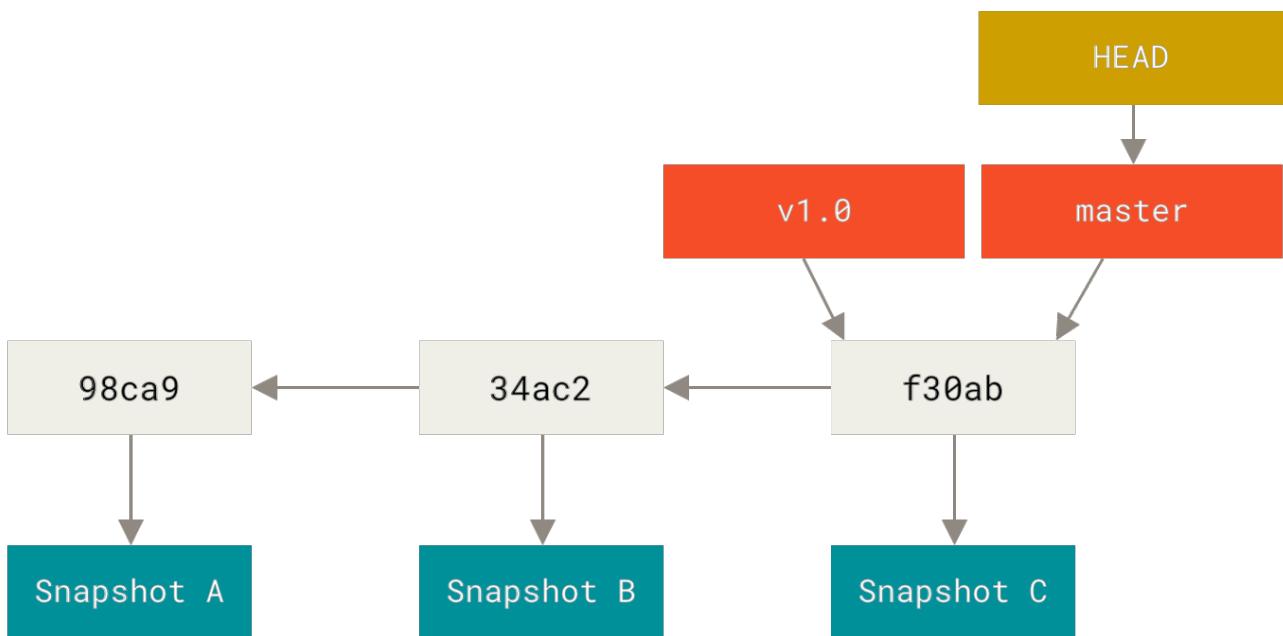


Figure 11. Ein Branch und sein Commit-Verlauf

## Erzeugen eines neuen Branches

Was passiert, wenn du einen neuen Branch anlegst? Nun, wenn du das tust, wird ein neuer Zeiger (Pointer) erstellt, mit dem du dich in der Entwicklung fortbewegen kannst. Nehmen wir an, du erzeugst einen neuen Branch mit dem Namen „testing“. Das machst du mit der Anweisung `git branch`:

```
$ git branch testing
```

Dieser Befehl erzeugt einen neuen Zeiger, der auf denselben Commit zeigt, auf dem du dich gegenwärtig befindest.

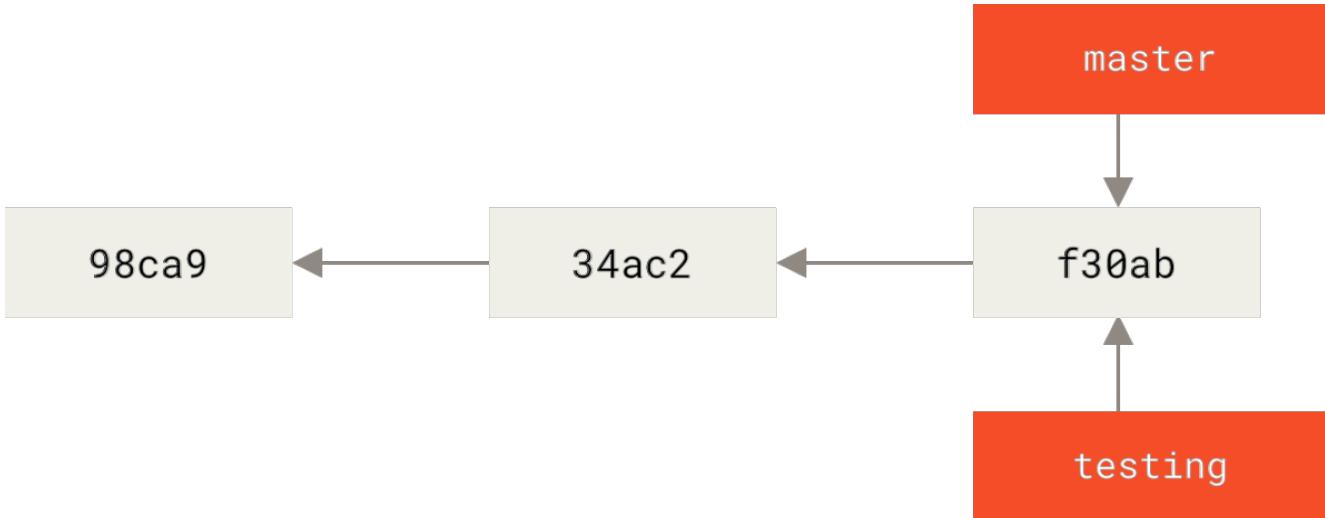


Figure 12. Zwei Branches, die auf dieselbe Serie von Commits zeigen

Woher weiß Git, auf welchem Branch du gegenwärtig bist? Es besitzt einen speziellen Zeiger namens `HEAD`. Beachte, dass dieser `HEAD` sich sehr stark unterscheidet von den `HEAD` Konzepten anderer Versionsverwaltungen, mit denen du vielleicht vertraut bist, wie Subversion oder CVS. Bei Git handelt es sich bei `HEAD` um einen Zeiger auf den lokalen Branch, auf dem du dich gegenwärtig befindst. In diesem Fall bist du weiterhin auf dem `master` Branch. Die Anweisung `git branch` hat den neuen Branch nur *erzeugt*, aber nicht zu diesem gewechselt.



Figure 13. Auf einen Branch zeigender HEAD

Du kannst das leicht nachvollziehen, indem du den einfachen Befehl `git log` ausführst, mit dem du siehst, wohin die Zeiger der Branches zeigen. Diese Option wird `--decorate` genannt.

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

```
98ca9 Initial commit
```

Du kannst die Branches `master` und `testing` sehen, die sich rechts neben dem Commit von `f30ab` befinden.

## Wechseln des Branches

Um zu einem existierenden Branch zu wechseln, führe die Anweisung `git checkout` aus. Lass uns zum neuen `testing` Branch wechseln.

```
$ git checkout testing
```

Dadurch wird `HEAD` verschoben, um auf den Branch `testing` zu zeigen.

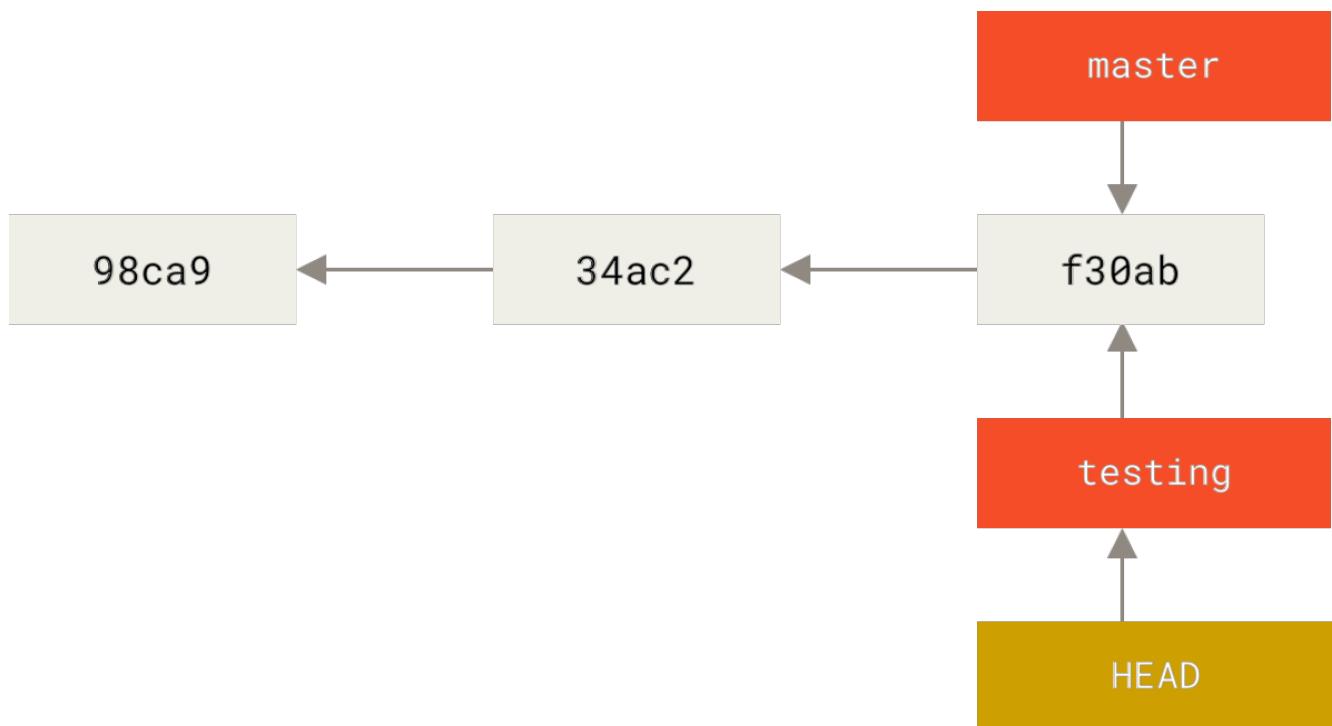


Figure 14. `HEAD` zeigt auf den aktuellen Branch

Was bedeutet das? Nun, lass uns einen weiteren Commit durchführen.

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

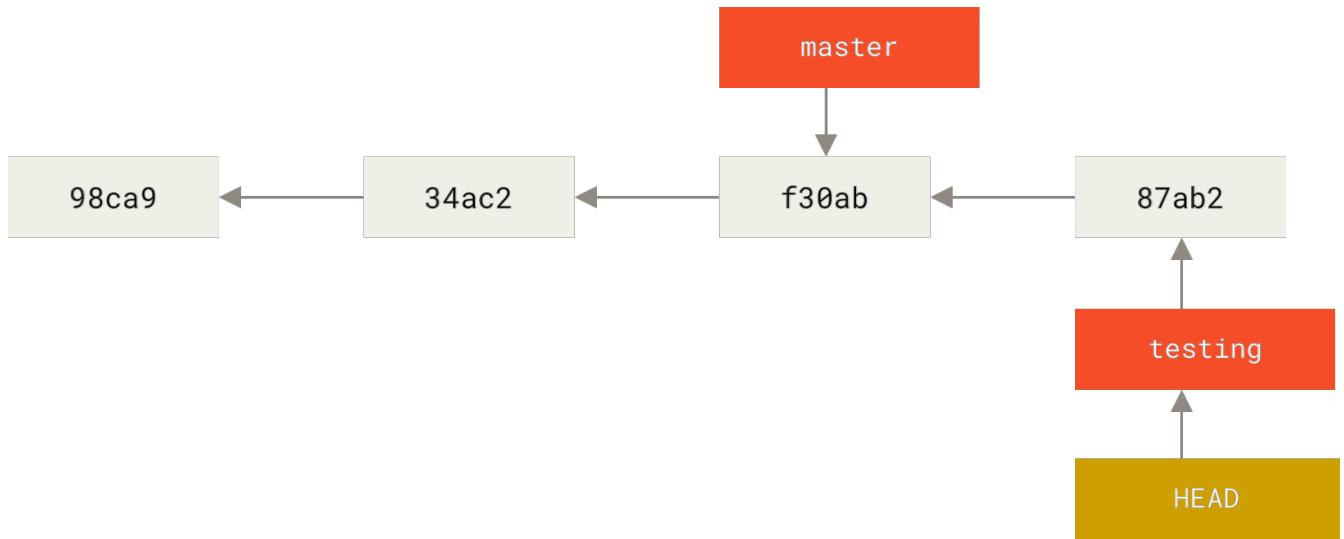


Figure 15. Der Branch, auf den HEAD zeigt, bewegt sich vorwärts, wenn ein Commit gemacht wird

Das ist interessant, weil sich jetzt dein `testing` Branch vorwärts bewegt hat, aber dein `master` Branch noch auf den Commit zeigt, auf dem du dich befandest, als du die Anweisung `git checkout` ausführtest, um die Branches zu wechseln. Lassen uns zum Branch `master` zurückwechseln.

```
$ git checkout master
```

#### `git log` zeigt nicht immer alle Branches

Wenn du jetzt `git log` aufrufen würdest, könntest du dich fragen, wohin der gerade erstellte „`testing`“ Branch verschwunden ist, da er nicht in der Anzeige auftaucht.

**i** Der Branch ist nicht spurlos verschwunden. Git weiß nur nicht, dass du dich für diesen Branch interessierst. Git versucht, dir das zu zeigen, woran du seiner Meinung nach interessiert bist. Anders gesagt, standardmäßig zeigt `git log` nur den Commit-Verlauf des Branches an, den du ausgecheckt hast.

Um die Commit-Historie für den gewünschten Branch anzuzeigen, musst du ihn explizit angeben: `git log testing`. Um alle Branches zu sehen, füge `--all` zu deinem Kommando `git log` hinzu.

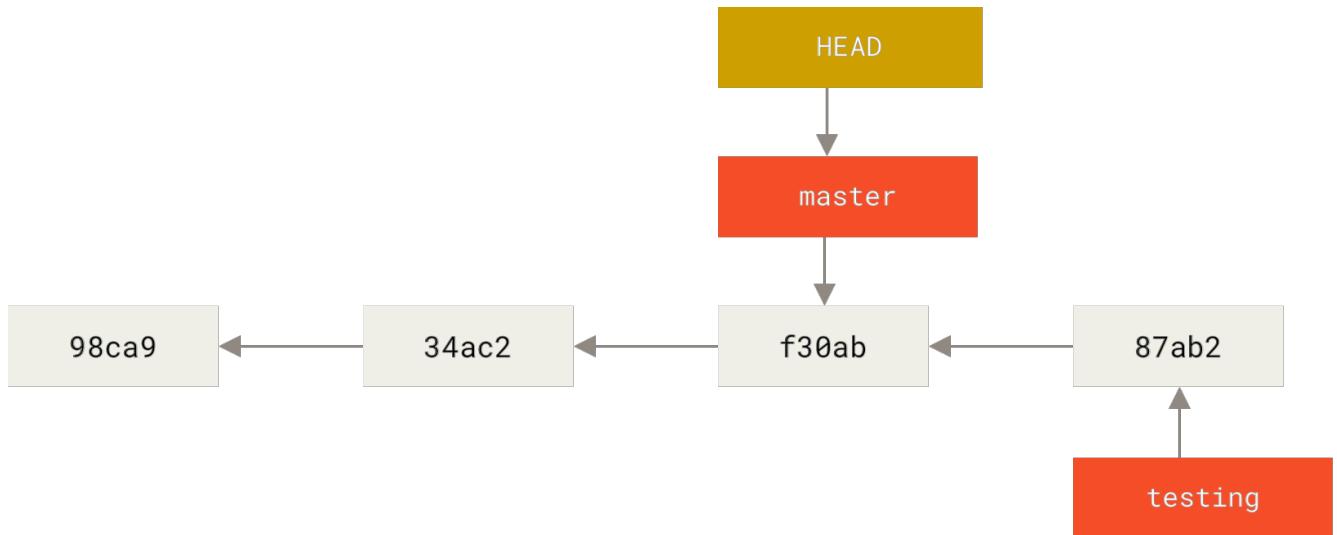


Figure 16. HEAD bewegt sich, wenn du auscheckst

Diese Anweisung hat zwei Dinge bewirkt. Es bewegte den HEAD-Zeiger zurück, um auf den `master` Branch zu zeigen und es setzte die Dateien in deinem Arbeitsverzeichnis auf den Snapshot zurück, auf den `master` zeigt. Das bedeutet auch, dass die Änderungen, die du von diesem Punkt aus vornimmst, von einer älteren Version des Projekts abzweigen werden. Du machst im Grunde genommen die Änderungen rückgängig, die du auf deinem `testing` Branch vorgenommen hast, sodass du in eine andere Richtung gehen kannst.

#### *Das Wechseln der Branches ändert Dateien in deinem Arbeitsverzeichnis*

Es ist wichtig zu beachten, dass sich die Dateien in deinem Arbeitsverzeichnis verändern, wenn du in Git die Branches wechselst. Wenn du zu einem älteren Branch wechselst, wird dein Arbeitsverzeichnis zurückverwandelt, sodass es aussieht wie zu dem Zeitpunkt, als du deinen letzten Commit auf diesem Branch durchgeführt hast. Wenn Git das nicht problemlos durchführen kann, lässt es dich die Branches überhaupt nicht wechseln.



Lass uns ein paar Änderungen vornehmen und noch einen Commit durchführen:

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

Jetzt hat sich dein Projektverlauf verzweigt (siehe [Verzweigter Verlauf](#)). Du hast einen Branch erstellt und bist zu ihm gewechselt, hast einige Arbeiten daran durchgeführt und bist dann wieder zu deinem Haupt-Branch zurückgekehrt, um andere Arbeiten durchzuführen. Beide Änderungen sind in separaten Branches isoliert: Du kannst zwischen den Branches hin und her wechseln sowie sie zusammenführen, wenn du soweit bist. Und das alles mit den einfachen Befehlen `branch`, `checkout` und `commit`.

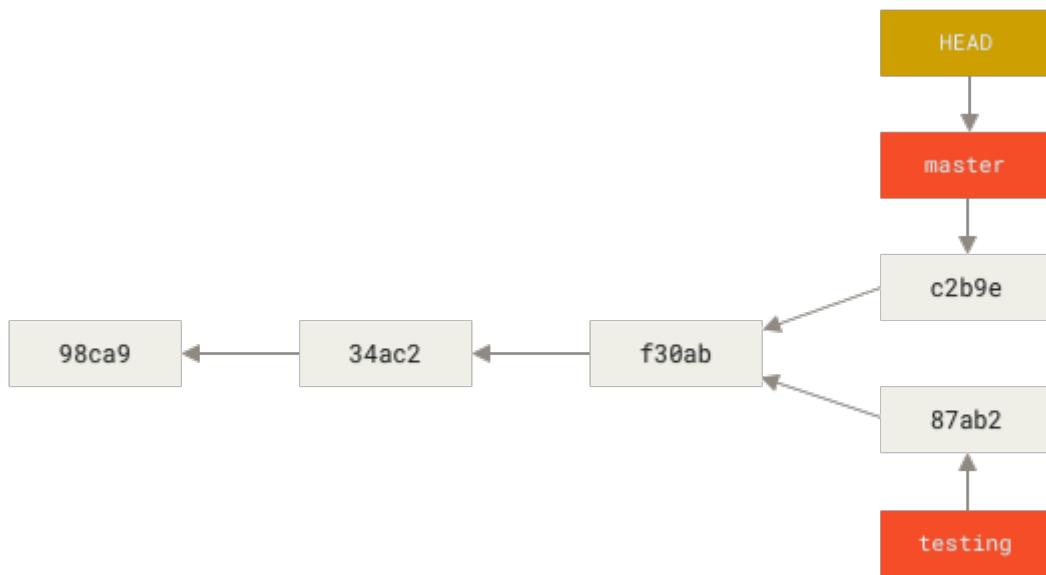


Figure 17. Verzweigter Verlauf

Du kannst dir dies ansehen, wenn du die Anweisung `git log` ausführst. Wenn du die Anweisung `git log --oneline --decorate --graph --all` ausführst, wird dir der Verlauf deiner Commits so angezeigt, dass erkennbar ist, wo deine Branch-Zeiger sich befinden und wie dein Verlauf sich verzweigt hat.

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

Da ein Branch in Git in Wirklichkeit eine einfache Datei ist, welche die 40-Zeichen lange SHA-1-Prüfsumme des Commits enthält, auf dem sie zeigt, können Branches ohne großen Aufwand erzeugt und gelöscht werden. Einen neuen Branch anzulegen, geht so schnell und ist so einfach, wie 41 Bytes in eine Datei zu schreiben (40 Zeichen und einen Zeilenumbruch).

Das steht im krassen Gegensatz zur Art und Weise, wie die meisten älteren Werkzeuge zur Versionsverwaltung Branches anlegen, bei der alle Projektdateien in ein zweites Verzeichnis kopiert werden. Das kann, in Abhängigkeit von der Projektgröße, mehrere Sekunden oder sogar Minuten dauern, während bei Git dieser Prozess augenblicklich erledigt ist. Da wir außerdem immer die Vorgänger mit aufzeichnen, wenn wir einen Commit durchführen, wird die Suche nach einer geeigneten Basis für das Zusammenführen (engl. merging) für uns automatisch durchgeführt, was in der Regel sehr einfach erledigt werden kann. Diese Funktionen tragen dazu bei, dass Entwickler ermutigt werden, häufig Branches zu erstellen und zu nutzen.

Lass uns herausfinden, warum du das tun solltest.

*Einen neuen Branch erzeugen und gleichzeitig dorthin wechseln.*



Es ist üblich, einen neuen Branch zu erstellen und gleichzeitig zu diesem neuen Branch zu wechseln – dies kann in einem Arbeitsschritt mit `git checkout -b <newbranchname>` passieren.

Ab Git version 2.23 kannst du `git switch` anstatt von `git checkout` nutzen um:



- Zu einem bestehendem Branch wechseln mit: `git switch testing-branch`.
- Einen neuen Branch erstellen und zu ihm wechseln mit: `git switch -c new-branch`. Die `-c` Option steht für Create (Anlegen), du kannst auch die komplette Option `--create` nutzen.
- Zurück zu deinem zuletzt ausgecheckten Branch wechseln mit: `git switch -`.

## Einfaches Branching und Merging

Lass uns ein einfaches Beispiel für das Verzweigen und Zusammenführen (engl. branching and merging) anschauen, wie es dir in einem praxisnahen Workflow begegnen könnte. Stell dir vor, du führst folgende Schritte aus:

1. Du arbeitest an einer Website
2. Du erstellst einen Branch für eine neue Anwendergeschichte (engl. User Story), an der du gerade arbeitest
3. Du erledigst einige Arbeiten in diesem Branch

In diesem Moment erhältst du einen Anruf, dass ein anderes Problem kritisch ist und ein Hotfix benötigt wird. Dazu wirst du folgendes tun:

1. Du wechselst zu deinem Produktions-Branch
2. Du erstellst einen Branch, um den Hotfix einzufügen
3. Nachdem der Test abgeschlossen ist, mergst du den Hotfix-Branch und schiebst ihn in den Produktions-Branch
4. Du wechselst zurück zu deiner ursprünglichen Anwenderstory und arbeitest daran weiter

## Einfaches Branching

Lass uns zunächst annehmen, du arbeitest an deinem Projekt und hast bereits ein paar Commits in deinem `master` Branch gemacht.

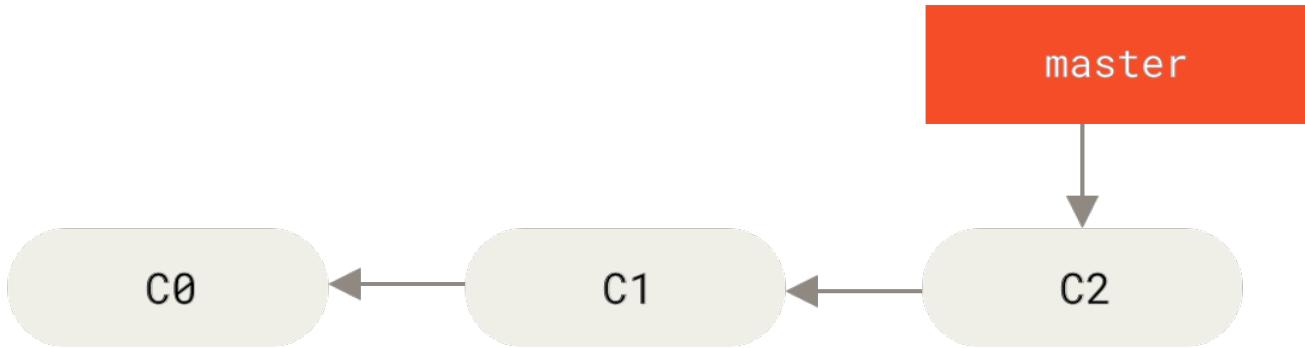


Figure 18. Ein einfacher Commit-Verlauf

Du hast dich dafür entschieden, an „Issue #53“ aus irgendeinem Fehlerverfolgungssystem, das deine Firma benutzt, zu arbeiten. Um einen neuen Branch anzulegen und gleichzeitig zu diesem zu wechseln, kannst du die Anweisung `git checkout` zusammen mit der Option `-b` ausführen:

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

Das ist die Kurzform der beiden folgenden Befehle:

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

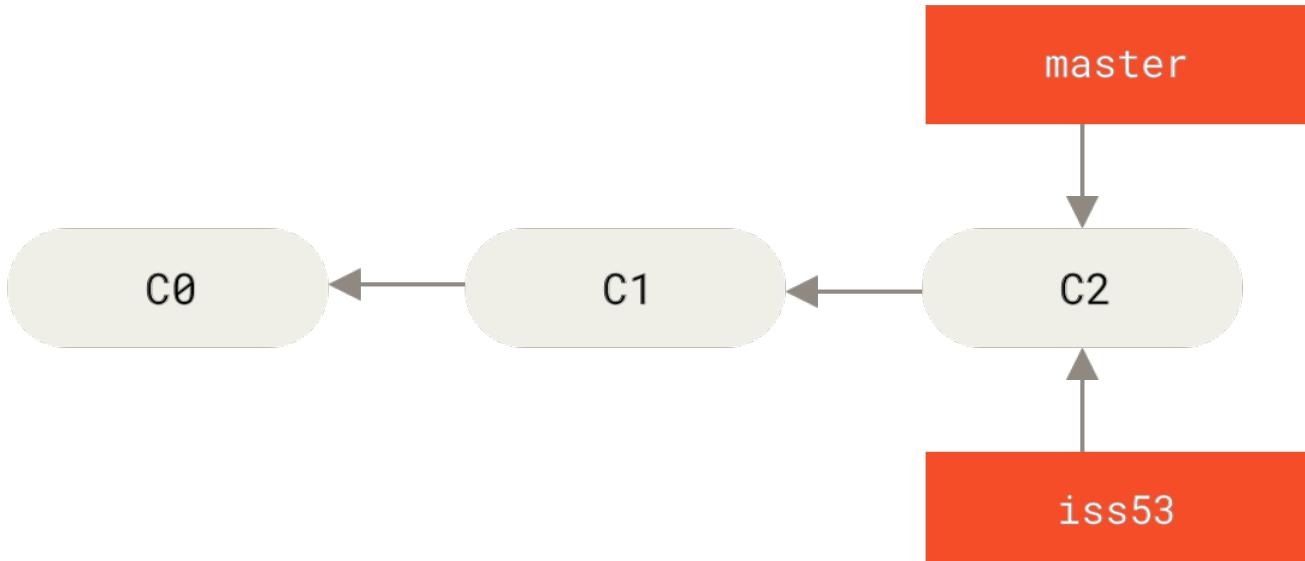


Figure 19. Erstellen eines neuen Branch-Zeigers

Du arbeitest an deiner Website und führst einige Commits durch. Sobald du das machst, bewegt das den `iss53` Branch vorwärts, weil du in ihn gewechselt (engl. checked out) bist. Das bedeutet, dein `HEAD` zeigt auf diesen Branch:

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

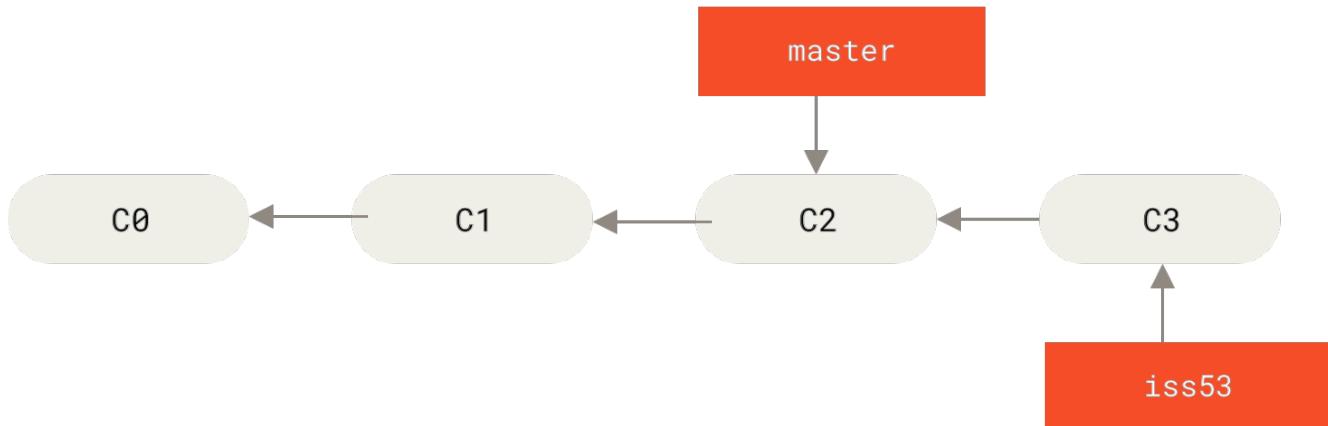


Figure 20. Der `iss53` Branch hat sich bei deiner Arbeit vorwärts bewegt

Jetzt bekommst du einen Anruf, dass es ein Problem mit der Website gibt und du es umgehend beheben musst. Bei Git musst du deinen Fix nicht zusammen mit den Änderungen bereitstellen, die du bereits an `iss53` vorgenommen hast. Du musst auch keinen großen Aufwand betreiben, diese Änderungen rückgängig zu machen, bevor du an den neuen Hotfix arbeiten kannst, um die Produktionsumgebung zu fixen. Alles, was du machen musst, ist zu deinem vorherigen `master` Branch zu wechseln.

Beachten dabei, dass Git das Wechseln zu einem anderen Branch blockiert, falls dein Arbeitsverzeichnis oder dein Staging-Bereich nicht committete Modifikationen enthält, die Konflikte verursachen. Generell ist es am besten, einen sauberen Zustand des Arbeitsbereichs anzustreben, bevor du Branches wechselst. Es gibt Möglichkeiten, das zu umgehen (nämlich das Verstecken (engl. Stashen) und Revidieren (engl. Amending) von Änderungen), die wir später in Kapitel 7 [Git Stashing](#) behandeln werden. Lass uns vorerst annehmen, du hast für alle deine Änderungen Commits durchgeführt, sodass du zu deinem vorherigen `master` Branch wechseln kannst.

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

Zu diesem Zeitpunkt befindet sich das Arbeitsverzeichnis des Projektes in exakt dem gleichen Zustand, in dem es sich befand, bevor du mit der Arbeit an „Issue #53“ begonnen hast und du kannst dich direkt auf den Hotfix konzentrieren. Das ist ein **wichtiger Punkt**, den du unbedingt beachten solltest: Wenn du die Branches wechselst, setzt Git dein Arbeitsverzeichnis zurück, um so auszusehen, wie es das letzte Mal war, als du in den Branch committed hast. Dateien werden automatisch hinzugefügt, entfernt und verändert, um sicherzustellen, dass deine Arbeitskopie auf demselben Stand ist wie zum Zeitpunkt deines letzten Commits auf diesem Branch.

Als Nächstes musst du dich um den Hotfix kümmern. Lass uns einen `hotfix` Branch erstellen, an dem du bis zu dessen Fertigstellung arbeitest:

```
$ git checkout -b hotfix
Switched to a new branch 'hotfix'
$ vim index.html
$ git commit -a -m 'Fix broken email address'
```

```
[hotfix 1fb7853] Fix broken email address  
1 file changed, 2 insertions(+)
```

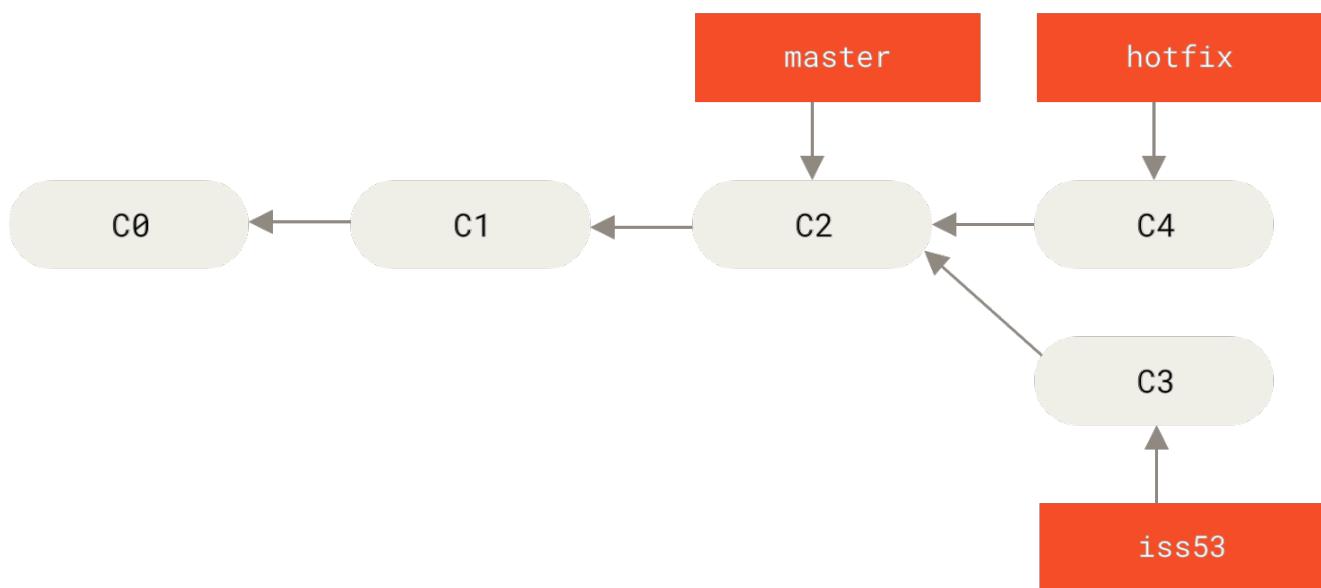


Figure 21. Auf dem `master` Branch basierender Hotfix-Branch

Du kannst deine Tests durchführen, dich vergewissern, dass der Hotfix das macht, was du von ihm erwartest und schließlich den Branch `hotfix` wieder in deinen `master` Branch integrieren (engl. merge), um ihn in der Produktion einzusetzen. Das machst du mit der Anweisung `git merge`:

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

Dir wird bei diesem Zusammenführen der Ausdruck „fast-forward“ auffallen. Da der Commit `C4`, auf den der von dir eingebundene Branch `hotfix` zeigt, direkt vor dem Commit `C2` liegt, auf dem du dich befindest, bewegt Git den Pointer einfach nach vorne. Um es anders auszudrücken: Wenn du versuchst, einen Commit mit einem Commit zusammenzuführen, der durch Verfolgen der Historie des ersten Commits erreicht werden kann, vereinfacht Git die Dinge, indem er den Zeiger nach vorne bewegt, da es keine verzweigte Arbeiten gibt, die miteinander gemerged werden müssen – das wird als „fast-forward“ bezeichnet.

Deine Änderung befindet sich nun im Schnappschuss des Commits, auf den der `master` Branch zeigt und du kannst deinen Fix ausliefern und installieren.

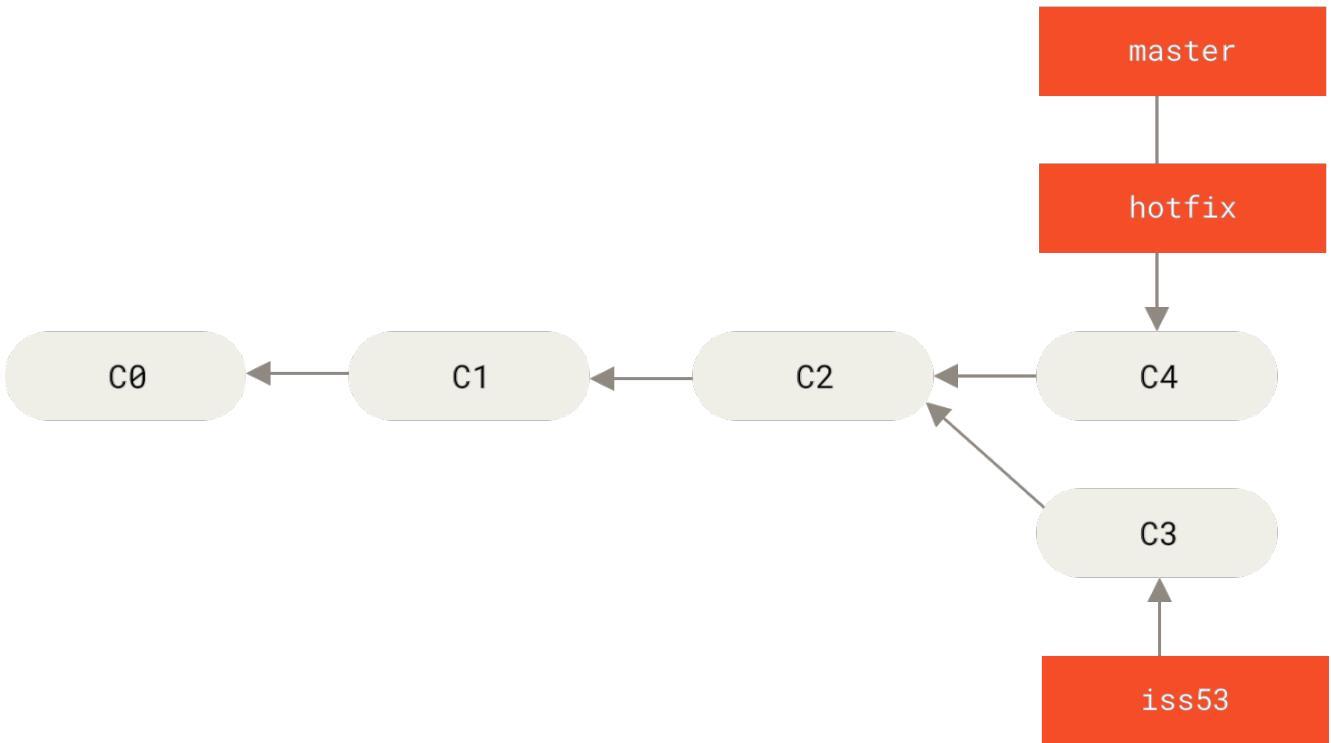


Figure 22. `master` wurde zu `hotfix` „fast-forwarded“

Nachdem deine überaus wichtiger Fix bereitgestellt wurde, kannst du dich wieder dem zuwenden, woran du vorher gearbeitet hast. Zunächst solltest du jedoch den `hotfix` Branch löschen, weil du diesen nicht länger benötigst – schließlich verweist der `master` Branch auf denselben Entwicklungsstand. Du kannst ihn mit der Anweisung `git branch` und der Option `-d` löschen:

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

Jetzt kannst du zum vorherigen Branch wechseln, auf dem du mit deinen Arbeiten an „Issue #53“ begonnen hast und daran weiter arbeiten.

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

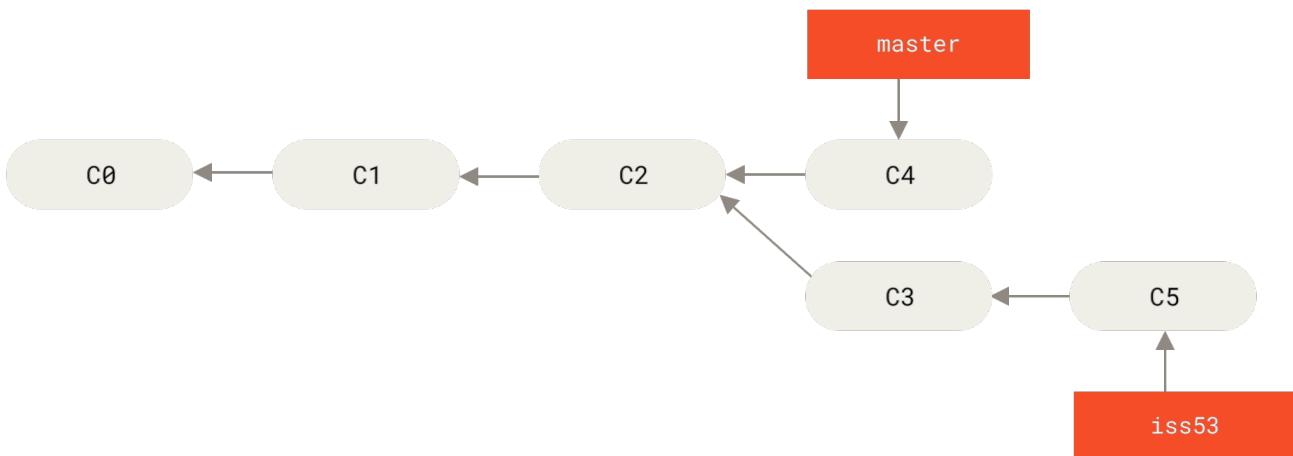


Figure 23. Arbeiten an `iss53` fortsetzen

Erwähnenswert ist, dass die Änderungen, die du in deinem `hotfix` Branch durchgeführt hast, nicht in den Dateien deines `iss53` Branch enthalten sind. Wenn du diese Änderungen übernehmen willst, kannst du deinen `master` Branch in den `iss53` Branch mergen, indem du `git merge master` ausführst. Du kannst aber auch warten und dich später dazu entscheiden, den `iss53` Branch in `master` zu übernehmen (engl. pullen).

## Einfaches Merging

Angenommen, du hast dich entschieden, dass dein Issue #53 abgeschlossen ist und du bereit bist, ihn in deinem Branch `master` zu mergen. Dann wirst du deinen `iss53` Branch in den `master` Branch mergen, so wie du es zuvor mit dem `hotfix` Branch gemacht hast. Du musst nur mit der Anweisung `checkout` zum Branch wechseln, in welchen du etwas einfließen lassen willst und dann die Anweisung `git merge` ausführen:

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

Das sieht ein bisschen anders aus, als das Merging mit dem `hotfix` Branch, welches du zuvor gemacht hast. Hier hat sich der Entwicklungsverlauf an einem früheren Zustand geteilt. Da der Commit auf dem Branch, auf dem du dich gerade befindest, kein unmittelbarer Vorgänger des Branches ist, in den du mergst, muss Git einige Arbeiten erledigen. In diesem Fall führt Git einen einfachen Drei-Wege-Merge durch, indem er die beiden Schnapschüsse verwendet, auf die das Branch-Ende und der gemeinsame Vorgänger der beiden zeigen.

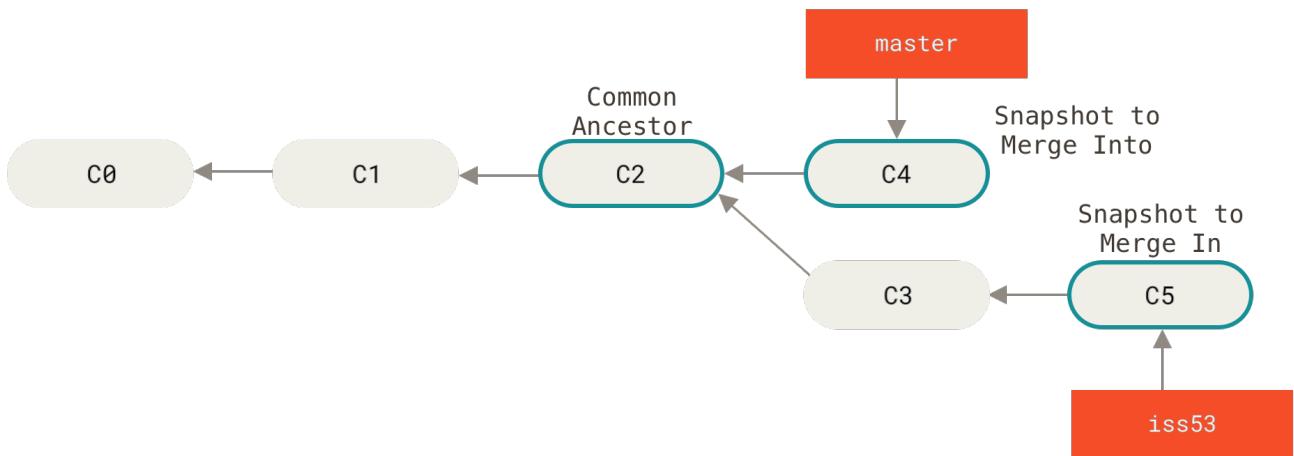


Figure 24. Drei Schnappschüsse, die bei einem typischen `merge` benutzt werden

Anstatt einfach den Zeiger des Branches vorwärts zu bewegen, erstellt Git einen neuen Schnappschuss, der aus dem Drei-Wege-Merge resultiert und erzeugt automatisch einen neuen Commit, der darauf zeigt. Das wird auch als Merge-Commit bezeichnet und ist ein Spezialfall, weil er mehr als nur einen Vorgänger hat.

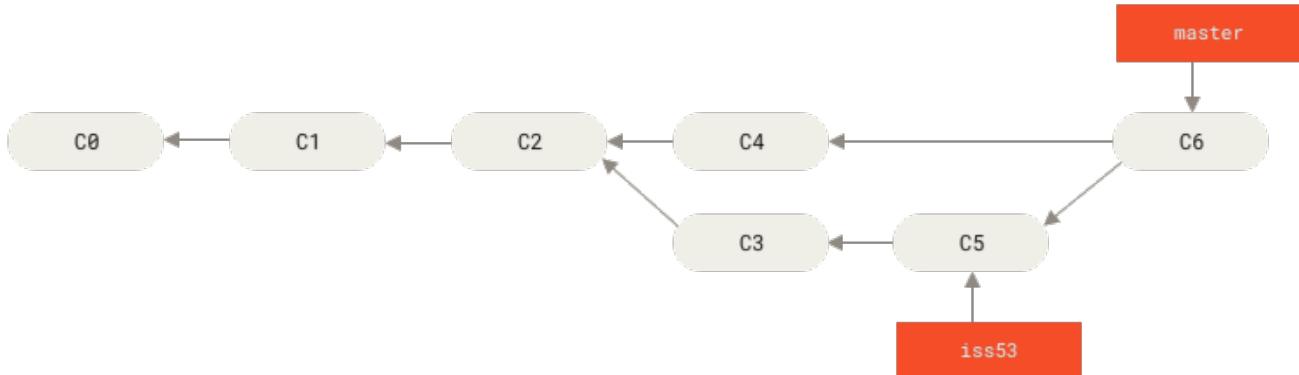


Figure 25. Ein Merge-Commit

Da deine Änderungen jetzt eingeflossen sind, hast du keine weitere Verwendung mehr für den `iss53` Branch. Du kannst den Issue in deinem Issue-Tracking-System schließen und den Branch löschen:

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

## Einfache Merge-Konflikte

Gelegentlich verläuft der Merge-Prozess nicht ganz reibungslos. Wenn du in den beiden Branches, die du zusammenführen willst, an derselben Stelle in derselben Datei unterschiedliche Änderungen vorgenommen hast, wird Git nicht in der Lage sein, diese sauber zusammenzuführen.

Wenn dein Fix für „Issue #53“ den gleichen Teil einer Datei wie der Branch `hotfix` geändert hat, erhältst du einen Merge-Konflikt, der ungefähr so aussieht:

```
$ 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 konnte einen neuen Merge-Commit nicht automatisch erstellen. Es hat den Prozess angehalten, bis du den Konflikt beseitigt hast. Wenn du sehen möchtest, welche Dateien zu irgendeinem Zeitpunkt nach einem Merge-Konflikt nicht zusammengeführt wurden, kannst du `git status` ausführen:

```
$ 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, was Merge-Konflikte ausgelöst hat und nicht behoben wurde, wird als `unmerged` angezeigt. Git fügt den Dateien, die Konflikte haben, Standardmarkierungen zur Konfliktlösung hinzu, so dass du sie manuell öffnen und diese Konflikte lösen können. Deine Datei enthält einen Bereich, der in etwa so aussieht:

```
<<<<< 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
```

Das bedeutet, die Version in `HEAD` (das ist der aktuelle commit `master` Branches, denn der wurde per `checkout` aktiviert, als du den `Merge` gestartet hast) ist der obere Teil des Blocks (alles oberhalb von `=====`) und die Version aus dem `iss53` Branch ist in dem darunter befindliche Teil. Um den Konflikt zu lösen, musst du dich entweder für einen der beiden Teile entscheiden oder du führst die Inhalte selbst zusammen. Du kannst diesen Konflikt beispielsweise lösen, indem du den gesamten Block durch diesen ersetzen:

```
<div id="footer">
```

```
please contact us at email.support@github.com  
</div>
```

Diese Lösung hat von beiden Teilen etwas und die Zeilen mit <<<<<, ===== und >>>>> wurden vollständig entfernt. Nachdem du alle problematischen Bereiche in allen von dem Konflikt betroffenen Dateien beseitigt hast, führst du einfach die Anweisung `git add` für alle betroffenen Dateien aus, um sie als gelöst zu markieren. Dieses „Staging“ der Dateien markiert sie für Git als bereinigt.

Wenn du ein grafisches Tool benutzen möchtest, um die Probleme zu lösen, dann kannst du `git mergetool` verwenden, welches ein passendes grafisches Merge-Tool startet und dich durch die Konfliktbereiche führt:

```
$ 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 ecmmerge
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):
```

Wenn du ein anderes Merge-Tool anstelle des Standardwerkzeugs verwenden möchten (Git wählte in diesem Fall `opendiff`, da die Anweisung auf einem Mac ausgeführt wurde), dann kannst du alle unterstützten Werkzeuge sehen, die oben nach „one of the following tools“ aufgelistet sind. Tippe einfach den Namen des gewünschten Programms ein.



Wenn du fortgeschrittenere Werkzeuge zur Lösung kniffliger Merge-Konflikte benötigst, erfährst du mehr darüber in Kapitel 7 [Fortgeschrittenes Merging](#).

Nachdem du das Merge-Tool beendet hast, wirst du von Git gefragt, ob das Zusammenführen erfolgreich war. Wenn du das bestätigst, wird die Datei der Staging-Area hinzugefügt und der Konflikt als gelöst markiert. Du kannst den Befehl `git status` erneut ausführen, um zu überprüfen, ob alle Konflikte gelöst wurden:

```
$ 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
```

Wenn du damit zufrieden bist und geprüft hast, dass alles, was Konflikte aufwies, der Staging-Area hinzugefügt wurde, kannst du die Anweisung `git commit` ausführen, um den Merge-Commit abzuschließen. Die standardmäßige Commit-Nachricht sieht ungefähr so aus:

```
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
#
```

Du kannst diese Commit-Nachricht noch Details darüber hinzufügen, wie du diesen Merge-Konflikt gelöst hast. Es könnte für künftige Betrachter dieses Commits hilfreich sein, zu verstehen, warum du etwas getan hast, falls es nicht offensichtlich ist.

## Branch-Management

Nachdem du nun einige Branches erzeugt, zusammengeführt und gelöscht hast, lass uns jetzt einige Werkzeuge für das Branch-Management betrachten, die sich als sehr nützlich erweisen werden, wenn du erst einmal Branches benutzt.

Der Befehl `git branch` kann noch mehr, als Branches zu erzeugen und zu löschen. Wenn du die Anweisung ohne Argumente ausführst, bekommst du eine einfache Auflistung deiner aktuellen Branches:

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

Beachte das Sternchen (\*), das dem Branch `master` vorangestellt ist: es zeigt an, welchen Branch du

gegenwärtig ausgecheckt hast (bzw. den Branch, auf den `HEAD` zeigt). Wenn du zu diesem Zeitpunkt einen Commit durchführst, wird der Branch `master` durch deine neue Änderung vorwärts bewegt. Um sich den letzten Commit auf jedem Branch anzeigen zu lassen, kannst du die Anweisung `git branch -v` ausführen:

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

Die nützlichen Optionen `--merged` und `--no-merged` kann diese Liste nach Branches filtern, welche bereits mit dem Branch, auf dem du dich gegenwärtig befindest, zusammengeführt wurden und welche nicht. Um zu sehen, welche Branches schon mit dem Branch zusammengeführt wurden, auf dem du gerade bist, kannst du die Anweisung `git branch --merged` ausführen:

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

Da du den Branch `iss53` schon früher gemerged hast, siehst du ihn in dieser Liste. Branches auf dieser Liste ohne vorangestelltes `*` können für gewöhnlich einfach mit der Anweisung `git branch -d` gelöscht werden; Du hast deren Änderungen bereits zu einem anderen Branch hinzugefügt, sodass du nichts verlieren würdest.

Um alle Branches zu sehen, welche Änderungen enthalten, die du noch nicht integriert hast, kannst du die Anweisung `git branch --no-merged` ausführen:

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

Das zeigt dir einen anderen Branch. Da er Änderungen enthält, die noch nicht integriert wurden, würde der Versuch, ihn mit `git branch -d` zu löschen, fehlschlagen:

```
$ 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'.
```

Wenn du den Branch wirklich löschen und diese Bearbeitungen aufgeben willst, kannst du dies mit der Option `-D` erzwingen, worauf git in seinem Output hinweist.



Wenn du keinen Commit- oder Branch-Namen als Argument angibst, zeigen dir die oben beschriebenen Optionen `--merged` und `--no-merged` was jeweils in deinem *current*-Branch gemerged oder nicht gemerged wurde.

Du kannst immer ein zusätzliches Argument angeben, um nach dem Merge-Status

in Bezug auf einen anderen Branch zu fragen, ohne zu diesen anderen Branch wechseln zu müssen. So wie im Beispiel unten: „Was ist nicht in den Branch `master` integriert?“

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

## Ändern eines Branchnamens



Benenne keine Branches um, die noch von anderen Mitstreitern verwendet werden. Benenne einen Branch wie `master` / `main` / `mainline` nicht um, ohne den Abschnitt „Ändern des Namens des Haupt-Branch“ gelesen zu haben.

Angenommen, du hast einen Branch mit dem Namen `bad-branch-name` und möchtest ihn in `corrected-branch-name` ändern, während die gesamte Historie beibehalten wird. Du möchtest auch den Branchnamen auf dem Remote-Repository ändern (GitHub, GitLab, anderer Server). Wie machst du das?

Benennen den Branch lokal mit dem Befehl `git branch --move` um:

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

Dies ersetzt deinen Branch `bad-branch-name` durch `corrected-branch-name`, aber diese Änderung ist vorerst nur lokal. Um den korrigierten Branchnamen für andere auf dem Remote-Repository sichtbar zu machen, pushe ihn:

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

Jetzt werfen wir einen kurzen Blick darauf, wo wir momentan stehen:

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

Beachte, dass du dich auf dem Branch `corrected-branch-name` befindest und er ist auf dem Remote-Repository verfügbar. Der fehlerhafte Branch ist ebenfalls auf dem Remote-Repository weiterhin vorhanden. Du kannst ihn vom Remote-Repository folgendermaßen löschen:

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

Nun ist der falsche Branchname vollständig durch den korrigierten Branchnamen ersetzt.

## Ändern des Master Branch Namens



Wenn du den Namen eines Branches wie master/main/mainline/default änderst, werden die Integrationen, Dienste, Hilfsprogramme und Build/Release-Skripte, die dein Repository verwendet, höchstwahrscheinlich nicht mehr funktionieren. Bevor du das tust, solltest du dies gründlich mit deinen Mitstreitern besprechen. Stelle außerdem sicher, dass du dein Repo gründlich durchsuchst und alle Verweise auf den alten Branchnamen in deinem Code und in deinen Skripten aktualisierst.

Benennen deinen lokalen `master` Branch mit dem folgenden Befehl in `main` um

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

Es gibt lokal keinen `master` Branch mehr, da er in `main` Branch umbenannt wurde.

Damit andere den neuen `main` Branch sehen können, musst du ihn auf das Remote-Repository pushen. Dadurch wird der umbenannte Branch auf dem Remote Repository verfügbar.

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

Jetzt haben wir folgenden Zustand:

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

Dein lokaler `master` Branch ist weg, da er durch den `main` Branch ersetzt wurde. Der Branch `main` ist nun auch auf dem Remote-Repository verfügbar. Aber im Remote-Repository existiert immer noch eine `master` Branch. Andere Mitstreiter werden weiterhin den Branch `master` als Grundlage für ihre Arbeit verwenden, bis du weitere Änderungen vornimmst.

Jetzt hast du noch ein paar Aufgaben vor dir, um den Übergang abzuschließen:

- Alle Projekte, die von diesem abhängen, müssen ihren Code und/oder ihre Konfiguration aktualisieren.
- Aktualisiere alle Test-Runner Konfigurationsdateien.
- Passe Build- und Release-Skripte an.

- Richte Umleitungen auf deinem Repo-Host für Dinge wie den Standardbranch des Repos, Merge-Regeln und andere Dinge ein, die mit Branchnamen übereinstimmen.
- Aktualisiere die Verweise auf den alten Branch in der Dokumentation.
- Schließe oder Merge alle Pull-Requests, die auf den alten Branch zeigen.

Nachdem du alle diese Aufgaben erledigt hast und sicher bist, dass der `main` Branch genau wie der `master` Branch funktioniert, kannst du den `master` Branch löschen:

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

## Branching-Workflows

Jetzt hast du die Grundlagen des Verzweigens (Branching) und Zusammenführen (Merging) kennengelernt. Was kannst oder solltest du damit anfangen? In diesem Abschnitt werden wir einige gängige Arbeitsabläufe vorstellen, die Gits leichtgewichtiger Branching-Mechanismus ermöglicht. Du kannst selber entscheiden, ob du eins davon in deinen eigenen Entwicklungszyklus integrieren möchtest.

### Langlaufende Branches

Da Git ein einfaches 3-Wege-Merge verwendet, ist mehrmaliges Zusammenführen von einem Branch in einen anderen über einen langen Zeitraum generell einfach zu bewerkstelligen. Das bedeutet, du kannst mehrere Branches haben, die immer offen sind und die du für unterschiedliche Stadien deines Entwicklungszyklus verwendest; du kannst sie regelmäßig mit anderen zusammenführen.

Viele Git-Entwickler folgen einem Workflow, welcher den Ansatz verfolgt, nur stabilen Code im `master` Branch zu haben – möglicherweise auch nur Code, der released wurde oder werden soll. Sie haben einen weiteren parallelen Branches namens `develop` oder `next`, auf denen Sie arbeiten oder die Sie für Stabilitätstests nutzen. Diese sind nicht zwangsläufig stabil, aber wann immer sie einen stabilen Zustand erreichen, können sie mit dem `master` Branch zusammengeführt werden. Sie werden genutzt, um Features (kurzlebige Branches, wie der früher genutzte `iss53` Branch) einfließen zu lassen, wenn diese fertiggestellt sind. Damit wird sichergestellt, dass sie alle Tests bestehen und keine Fehler einführen.

Eigentlich reden wir gerade über Zeiger, die sich in der Reihe der Commits, die du durchführst, vorwärts bewegen. Die stabilen Branches sind weiter hinten und die allerneuesten Branches sind weiter vorn im Verlauf.

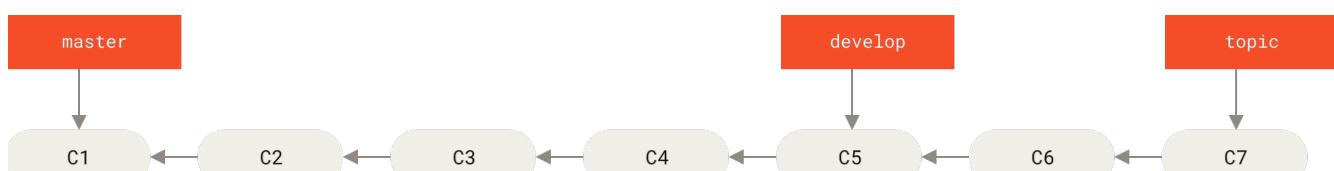


Figure 26. Lineares Modell eines Branchings mit zunehmender Stabilität

Es ist einfacher, sich die verschiedenen Branches als Silos vorzustellen, in denen Gruppen von Commits in stabile Silos aufsteigen, sobald sie vollständig getestet wurden.

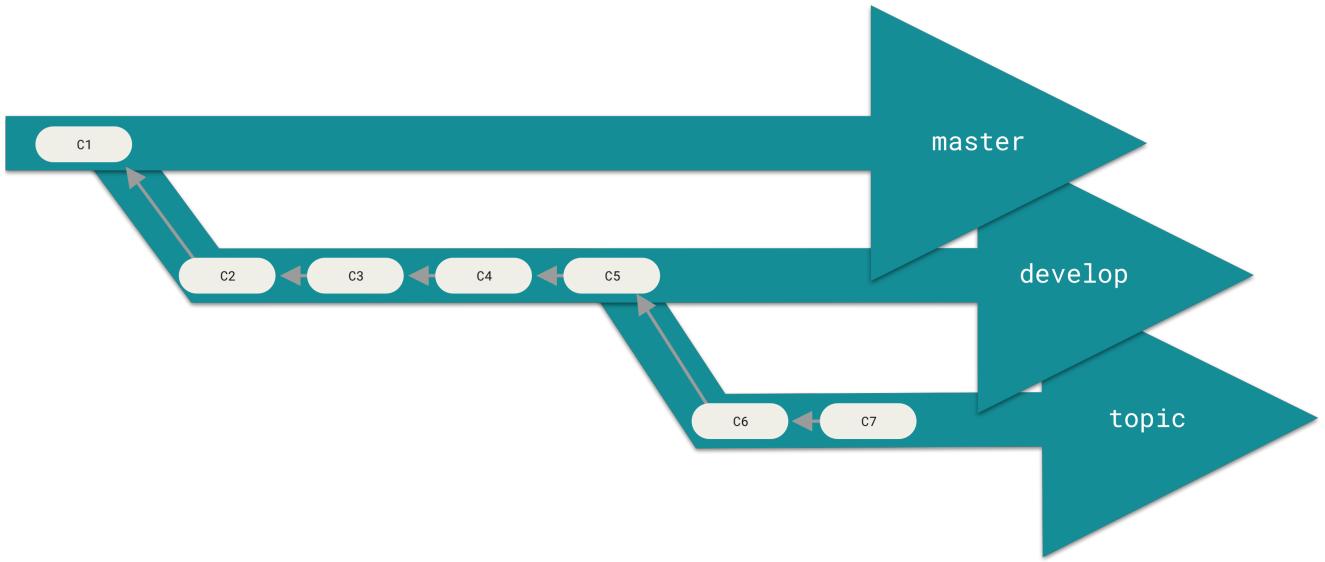


Figure 27. „Silo“-Modell eines Branchings mit zunehmender Stabilität

Du kannst das für mehrere Stabilitätsgrade einrichten. Einige größere Projekte haben auch einen Branch **proposed** (vorgeschlagen) oder **pu** (proposed updates – vorgeschlagene Updates), in welchem Branches integriert sind, die vielleicht noch nicht bereit sind in den Branch **next** oder **master** einzufließen. Die Idee dahinter ist, dass Ihre Branches verschiedene Stabilitäts-Level repräsentieren. Sobald sie einen Grad höherer Stabilität erreichen, werden sie mit dem nächsthöheren Branch zusammengeführt. Zur Erinnerung: Langfristig verschiedene Branches parallel laufen zu lassen, ist nicht notwendig, aber oft hilfreich. Insbesondere dann, wenn man es mit sehr großen oder komplexen Projekten zu tun hat.

## Themen-Branches (Feature-Branches)

Feature-Branches sind in Projekten jeder Größe nützlich. Ein Feature-Branch ist ein kurzlebiger Branch, welchen du für eine ganz bestimmte Funktion oder zusammengehörende Arbeiten erstellst und nutzt. Das ist etwas, was du wahrscheinlich noch nie zuvor mit einem Versionsverwaltungssystem gemacht hast, weil es normalerweise zu aufwändig und mühsam ist, Branches zu erstellen und zusammenzuführen. Aber bei Git ist es vollkommen üblich, mehrmals am Tag Branches zu erstellen, an ihnen zu arbeiten, sie zusammenzuführen und sie anschließend wieder zu löschen.

Du hast das im letzten Abschnitt an den erstellten Branches **iss53** und **hotfix** gesehen. Du führst mehrere Commits auf diesen Branches durch und löschst sie sofort, nachdem du sie mit deinem Hauptbranch zusammengeführt hast. Diese Technik erlaubt es dir, schnell und vollständig den Kontext zu wechseln – da deine Arbeit auf verschiedene Silos aufgeteilt ist, wo alle Änderungen auf diesem Branch unter diese Thematik fallen, ist es leichter nachzuvollziehen, was bei Code-Überprüfungen und Ähnlichem geschehen ist. Du kannst die Änderungen darin für Minuten, Tage oder Monate aufbewahren und sie einfließen lassen (mergen), wenn diese fertig sind, ungeachtet der Reihenfolge, in welcher diese erstellt oder bearbeitet wurden.

Betrachten wir folgendes Beispiel: Du erledigst gerade einige Arbeiten (auf **master**), zweigst davon ab wegen eines Problems (**iss91**), arbeitest daran eine Weile, zweigst davon den zweiten Branch ab, um eine andere Möglichkeit zur Handhabung desselben Problems auszuprobieren (**iss91v2**), wechselst zurück zu deinem **master** Branch und arbeitest dort eine Zeit lang, und zweigst dann dort

nochmal ab, um etwas zu versuchen, bei dem du dir nicht sicher bist, ob es eine gute Idee ist (**dumbidea** Branch). Dein Commit-Verlauf wird in etwa so aussehen:

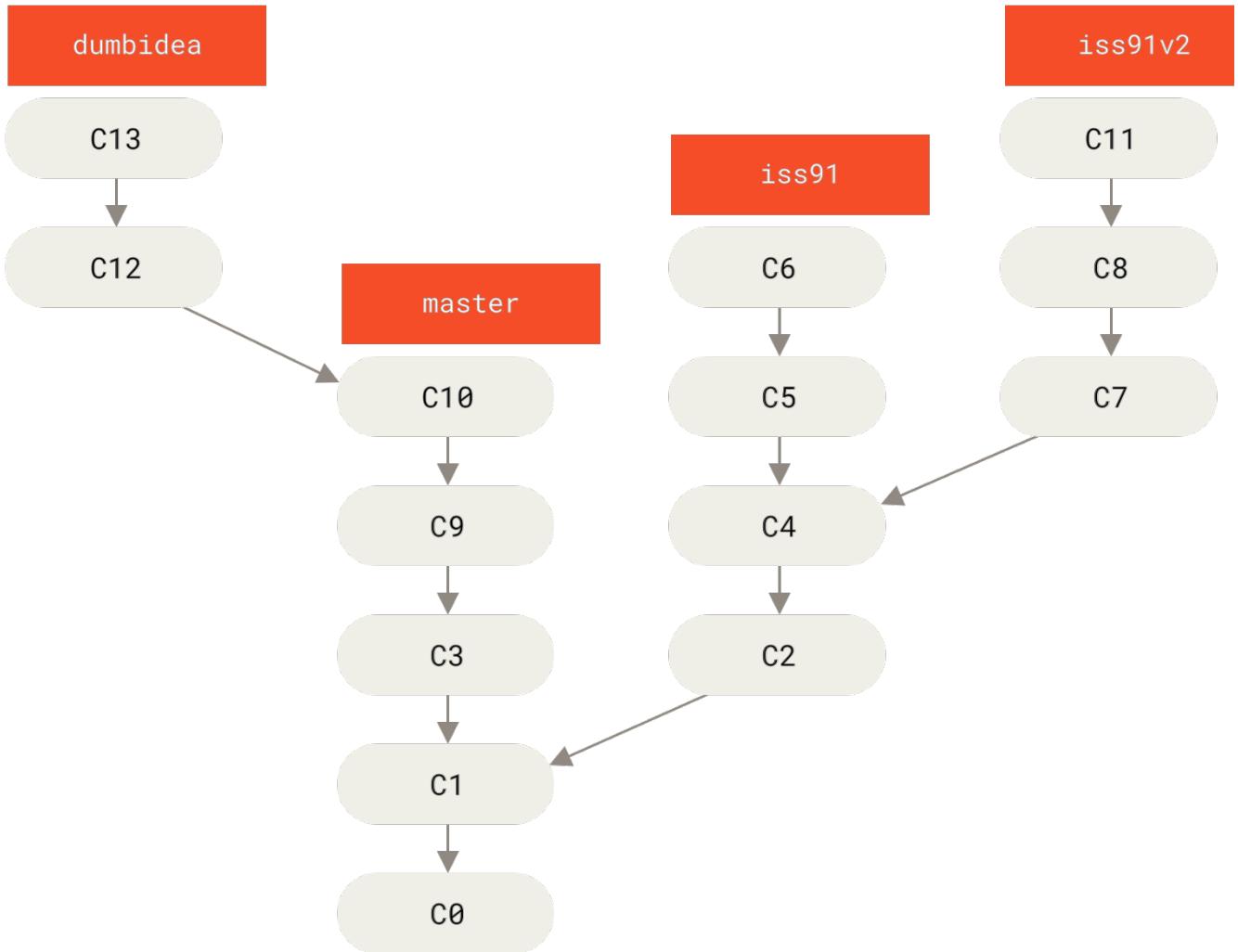


Figure 28. Mehrere Themen-Branches

Angenommen, du hast dich jetzt entschieden, dass dir die zweite Lösung für dein Problem (**iss91v2**) am besten gefällt; und du hast den **dumbidea** Branch deinen Mitstreitern gezeigt und es hat sich herausgestellt, dass er genial ist. Du kannst also den ursprünglichen **iss91** Branch (unter Verlust der Commits **C5** und **C6**) wegwerfen und die anderen beiden einfließen lassen. Dein Verlauf sieht dann so aus:

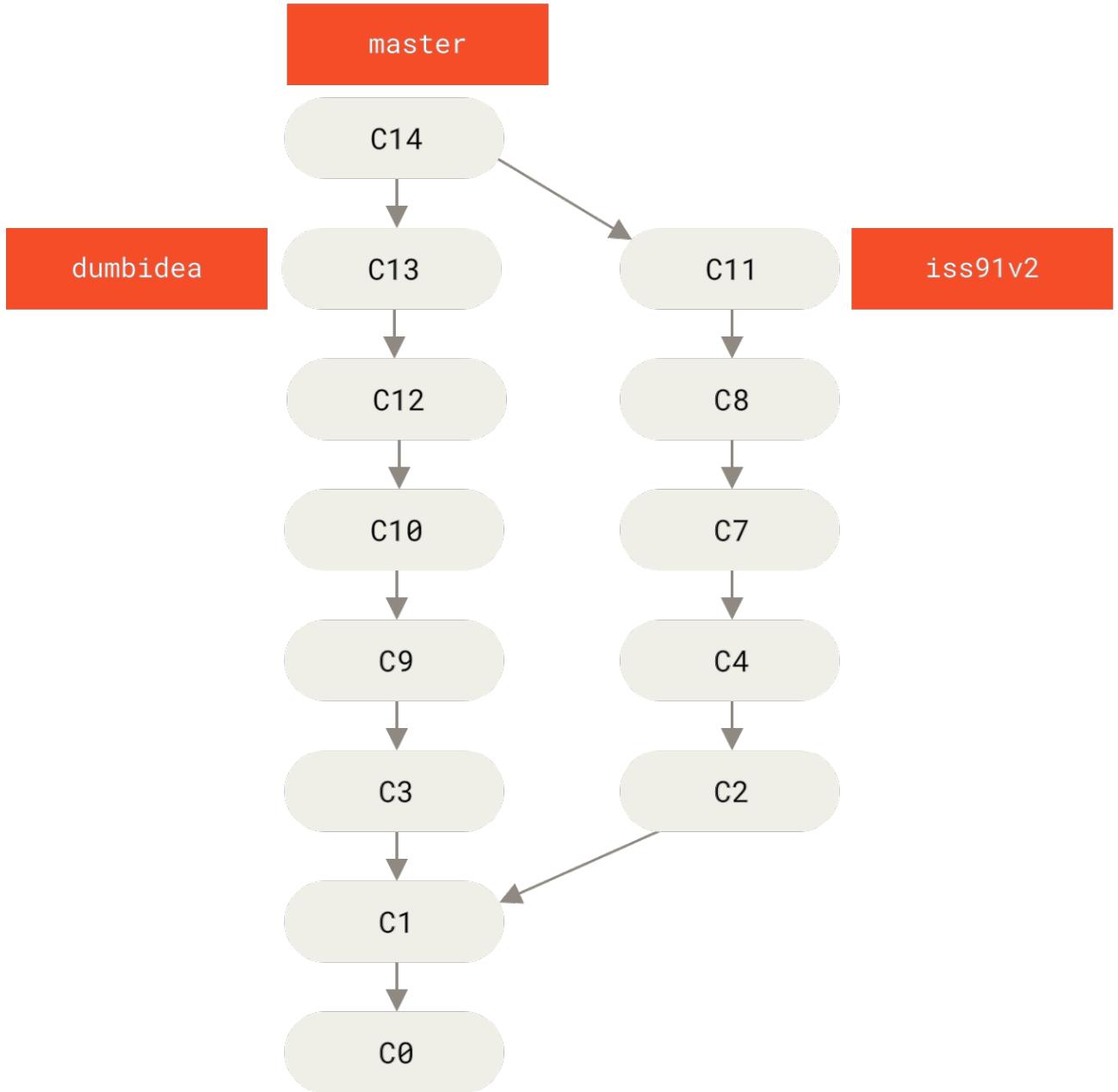


Figure 29. Verlauf nach dem Mergen von `dumbidea` und `iss91v2`

In Kapitel 5 [Verteiltes Git](#) werden wir die verschiedenen möglichen Arbeitsabläufe für dein Git-Projekt noch detaillierter betrachten. Bevor du dich also entscheidest, welches Branching-Modell du für dein nächstes Projekt nutzen willst, solltest du unbedingt dieses Kapitel gelesen haben.

Es ist wichtig, sich bei all dem daran zu erinnern, dass diese Branches nur lokal existieren. Wenn du Branches anlegst und zusammenführst, geschieht das alles nur in deinem lokalen Git-Repository – es findet keine Server-Kommunikation statt.

## Remote-Banches

Remote-Referenzen sind Referenzen (Zeiger) in deinem Remote-Repositorys, einschließlich Branches, Tags usw. Du kannst eine vollständige, ausführliche Liste von Remote-Referenzen bekommen, wenn du die Anweisungen `git ls-remote <remote>` oder `git remote show <remote>` für Remote-Banches ausführst, sowie auch für weitere Informationen. Der gängigerer Ansatz ist

jedoch die Nutzung von Remote-Tracking-Banches.

Remote-Tracking-Banches sind Referenzen auf den Zustand von Remote-Banches. Sie sind lokale Referenzen, die du nicht manuell ändern kannst. Sie werden automatisch für dich geändert, sobald Sie irgendeine Netzwerkkommunikation durchführen. Betrachte sie als Lesezeichen, die daran erinnern, wo die Branches in deinem Remote-Repository das letzte Mal standen, als du dich mit ihnen verbunden hast.

Remote-Tracking-Branch-Namen haben die Form `<remote>/<branch>`. Wenn du beispielsweise wissen möchtest, wie der Branch `master` in deinem Repository `origin` ausgesehen hat, als du zuletzt Kontakt mit ihm hattest, dann würdest du den Branch `origin/master` überprüfen. Wenn du mit einem Mitstreiter an einem Problem gearbeitet hast und dieser bereits einen `iss53` Branch hochgeladen (gepusht) hat, besitzt du möglicherweise deinen eigenen lokalen `iss53` Branch. Der Branch auf dem Server würde jedoch vom Remote-Tracking-Branch `origin/iss53` dargestellt werden.

Das kann ein wenig verwirrend sein, lass uns also ein Beispiel betrachten. Angenommen, du hast in deinem Netzwerk einen Git-Server mit der Adresse `git.ourcompany.com`. Wenn du von diesem klonst, erhält der Server von der Git-Anweisung `clone` automatisch den Namen `origin`, lädt all seine Daten herunter, erstellt einen Zeiger auf dem `master` Branch zeigt und benennt ihn lokal `origin/master`. Git gibt dir auch deinen eigenen lokalen `master` Branch mit der gleichen Ausgangsposition wie der `origin/master` Branch, damit du einen Punkt hast, wo du mit deiner Arbeit beginnen kannst.

*„origin“ ist nichts Besonderes*

Genau wie der Branch-Name „master“ in Git keine besondere Bedeutung hat, hat auch „origin“ keine besondere Bedeutung. Während „master“ die Standardbezeichnung für den Anfangsbranch ist, wenn du die Anweisung `git init` ausführst, was der einzige Grund dafür ist, warum er so weit verbreitet ist, wird „origin“ als Standardbezeichnung für ein entferntes Repository vergeben, wenn du die Anweisung `git clone` ausführst. Wenn du stattdessen die Anweisung `git clone -o booyah` ausführst, erhältst du `booyah/master` als Standard-Remote-Branch.



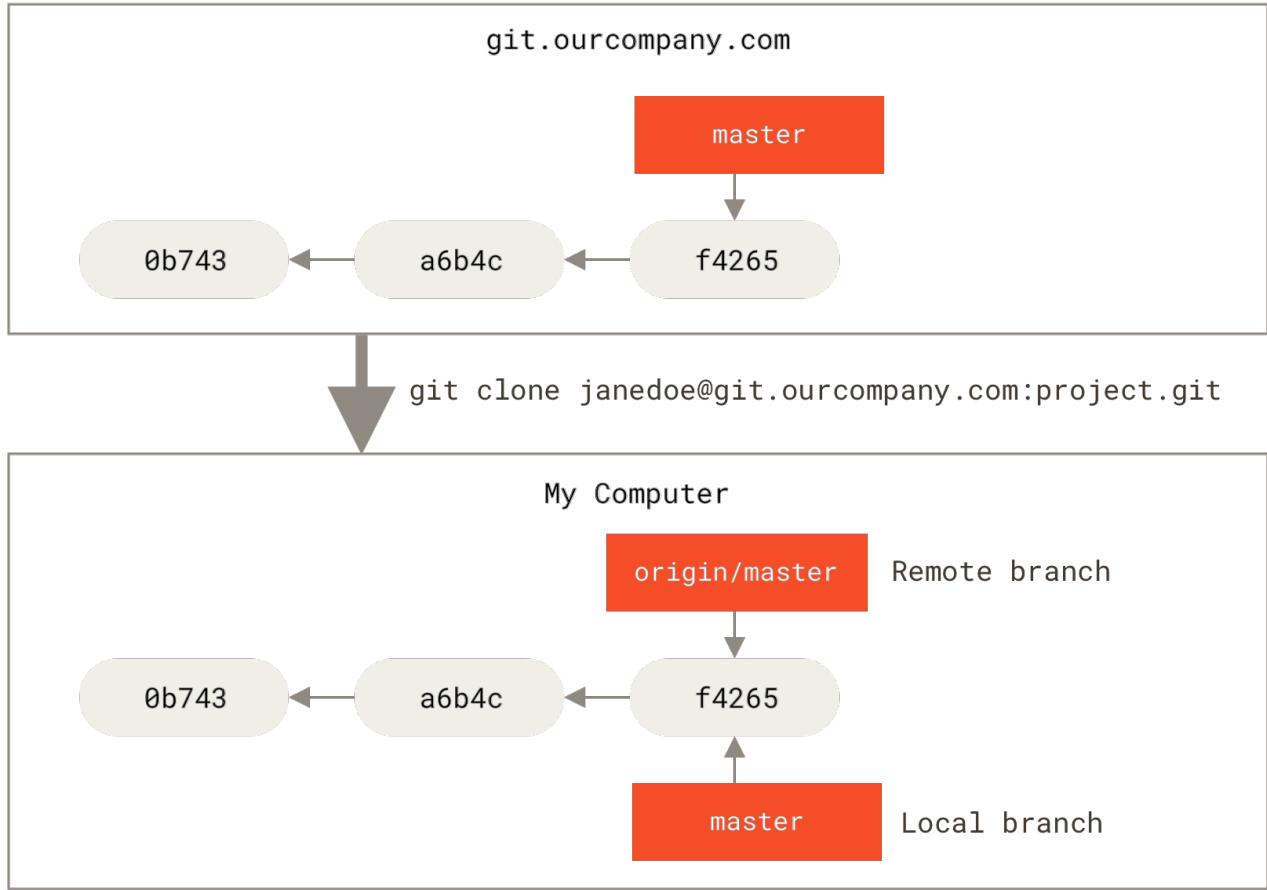


Figure 30. Entfernte und lokale Repositorys nach dem Klonen

Wenn du ein wenig an deinem lokalen **master** Branch arbeitest und in der Zwischenzeit jemand anderes etwas zu **git.ourcompany.com** hochlädt und damit dessen **master** Branch aktualisiert, dann bewegen sich eure Verläufe unterschiedlich vorwärts. Solange du keinen Kontakt mit deinem **origin** Server aufnimmst, bewegt sich dein **origin/master** Zeiger nicht.

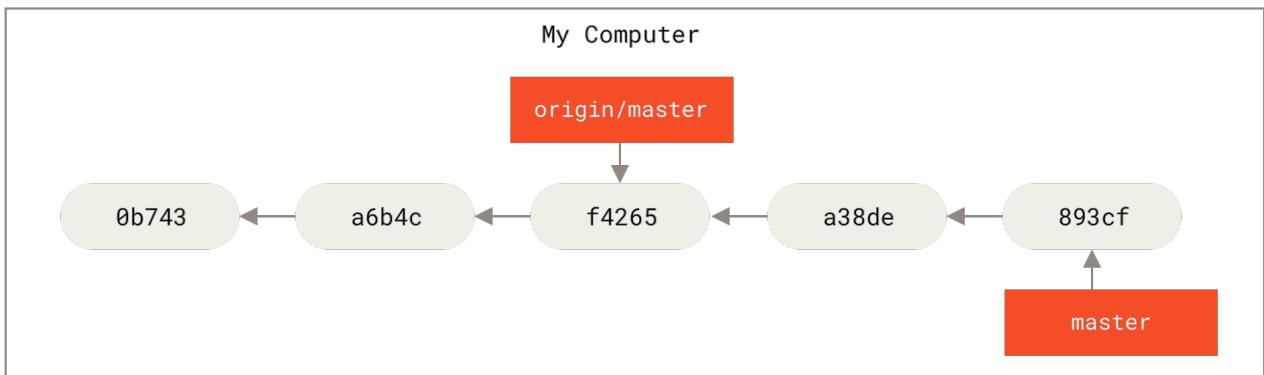
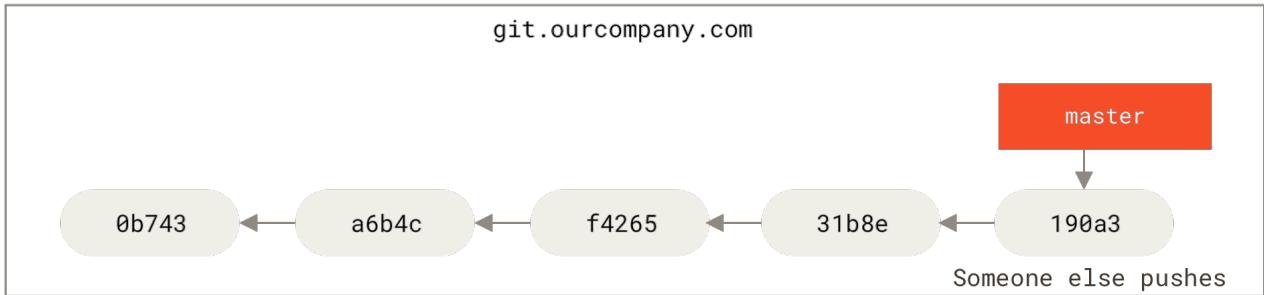


Figure 31. Lokale und entfernte Änderungen können Auseinanderlaufen

Um deine Arbeit mit einem bestimmten Remote zu synchronisieren, führst du den Befehl `git fetch <remote>` aus (in unserem Fall `git fetch origin`). Der Befehl sucht, welcher Server „origin“ ist (in diesem Fall `git.ourcompany.com`), holt alle Daten, die du noch nicht hast, und aktualisierst deine lokale Datenbank, indem er deinen `origin/master` Zeiger auf seine neue, aktuellere Position bewegt.

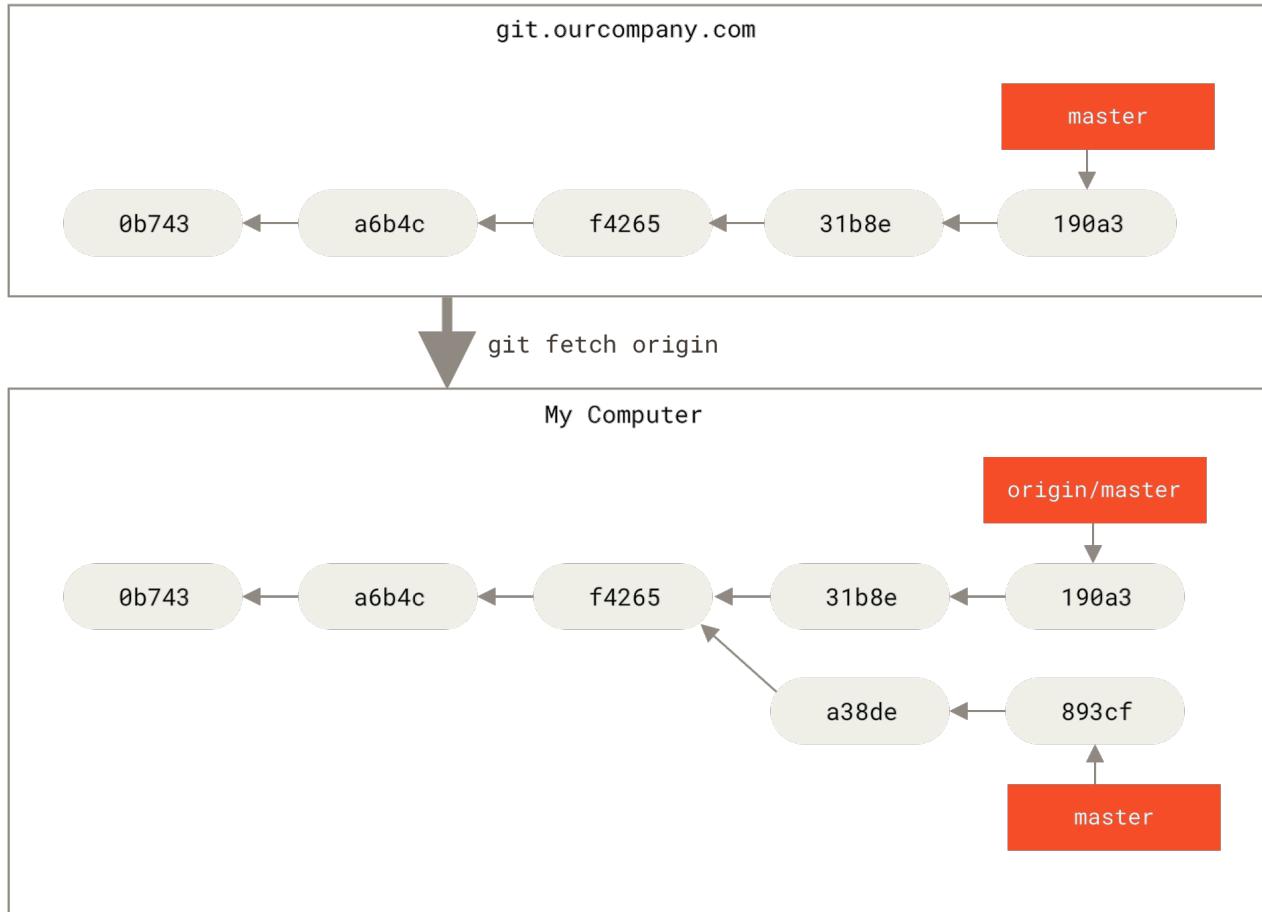


Figure 32. `git fetch` aktualisiert deine Remote-Tracking-Branches

Um den Umgang mit mehreren Remote-Servern zu veranschaulichen und um zu sehen, wie Remote-Branches bei diesen Remote-Projekten aussehen, nehmen wir an, dass du einen weiteren internen Git-Server hast, welcher von einem deiner Sprint-Teams nur zur Entwicklung genutzt wird. Diesen Server erreichen wir unter `git.team1.ourcompany.com`. Du kannst ihn zu dem Projekt, an dem du gegenwärtig arbeitest, als neuen Remote-Server hinzufügen, indem du die Anweisung `git remote add` ausführst, wie wir bereits in Kapitel 2 [Git Grundlagen](#) behandelt haben. Wir nennen diesen Remote-Server `teamone`, was die Kurzbezeichnung für die gesamte URL sein wird.

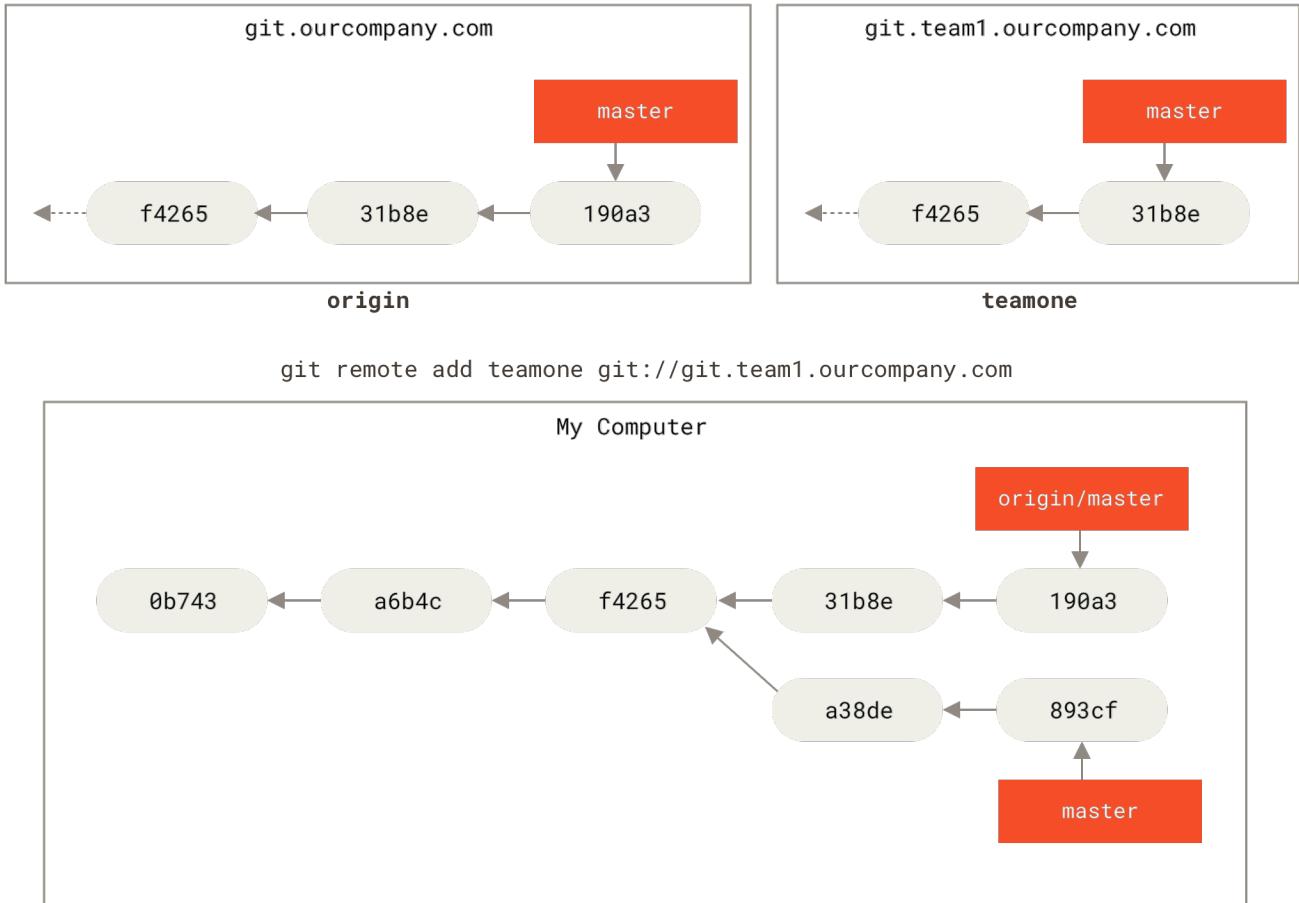


Figure 33. Hinzufügen eines weiteren Remote-Servers

Jetzt kannst du mit der Anweisung `git fetch teamone` alles vom Server holen, was du noch nicht hast. Da auf diesem Server nur eine Teilmenge der Daten ist, die sich genau jetzt auf deinem `origin` Server befinden, holt Git keine Daten ab, aber es erstellt einen Remote-Branch `teamone/master` so, dass er auf den Commit zeigt, den `teamone` als seinen `master` Branch hat.

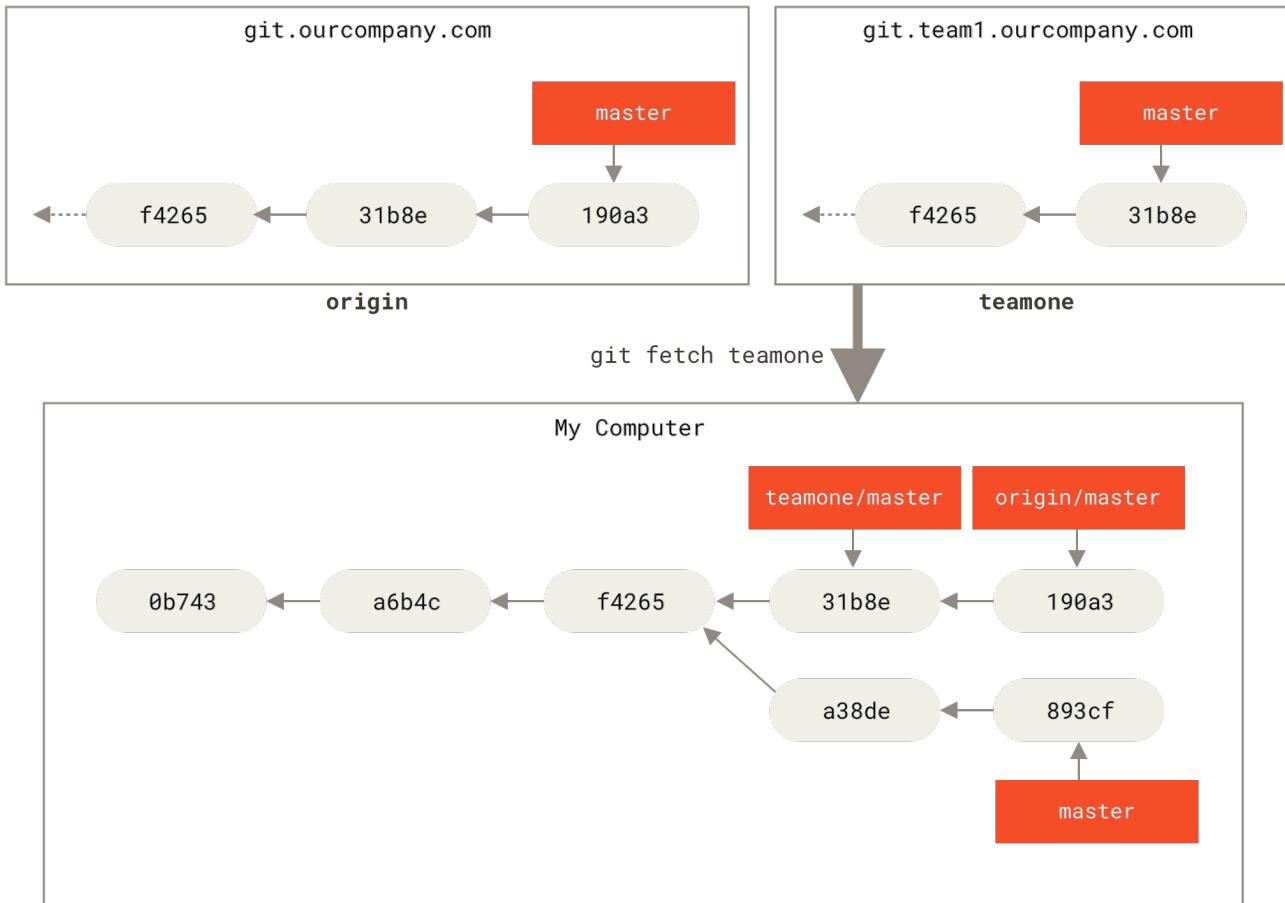


Figure 34. Remote-Tracking-Branch für `teamone/master`

## Pushing/Hochladen

Wenn du einen Branch mit der Welt teilen möchtest, musst du ihn auf einen Remote-Server hochladen, auf dem du Schreibrechte besitzt. Deine lokalen Branches, auf die du schreibst, werden nicht automatisch mit den Remotes synchronisiert – Du musst die Branches, die du freigeben möchtest, explizit pushen. Auf diese Weise kannst du private Branches, die du nicht veröffentlichen willst, zum Arbeiten benutzen und nur die Feature-Branches pushen, an denen du mitarbeiten willst.

Wenn du einen Branch namens `serverfix` besitzt, an dem du mit anderen arbeiten möchtest, dann kannst du diesen auf dieselbe Weise Hochladen wie deinen ersten Branch. Führe die Anweisung `git push <remote> <branch>` aus:

```
$ 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
```

Das ist eine Art Abkürzung. Git erweitert den Branch-Namen `serverfix` automatisch zu

`refs/heads/serverfix:refs/heads/serverfix`, was soviel bedeutet wie „Nimm meinen lokalen Branch `serverfix` und aktualisiere damit den `serverfix` Branch auf meinem Remote-Server“. Wir werden den Teil `refs/heads/` in Kapitel 10 [Git Interna](#) noch näher beleuchten, du kannst ihn aber in der Regel auslassen. Du kannst auch die Anweisung `git push origin serverfix:serverfix` ausführen, was das Gleiche bewirkt – es bedeutet „Nimm meinen `serverfix` und mach ihn zum `serverfix` des Remote-Servers“. Du kannst dieses Format auch benutzen, um einen lokalen Branch in einen Remote-Branch mit anderem Namen zu pushen. Wenn du nicht willst, dass er auf dem Remote als `serverfix` bezeichnet wird, kannst du stattdessen `git push origin serverfix:awesomebranch` ausführen, um deinen lokalen `serverfix` Branch auf den `awesomebranch` Branch im Remote-Projekt zu pushen.

*Geb dein Passwort nicht jedes Mal neu ein*

Wenn du eine HTTPS-URL zum Übertragen verwendest, fragt der Git-Server nach deinem Benutzernamen und Passwort zur Authentifizierung. Standardmäßig wirst du auf dem Terminal nach diesen Informationen gefragt, damit der Server erkennen kann, ob du pushen darfst.



Wenn du es nicht jedes Mal eingeben willst, wenn du etwas hochlädst, dann kannst du einen „credential cache“ einstellen. Am einfachsten ist es, die Informationen nur für einige Minuten im Speicher zu behalten, was du einfach mit der Anweisung `git config --global credential.helper cache` bewerkstelligen kannst.

Weitere Informationen zu den verschiedenen verfügbaren „credential cache“ Optionen findest du in Kapitel 7 [Caching von Anmeldeinformationen](#).

Das nächste Mal, wenn einer deiner Mitstreiter Daten vom Server abholt, wird er eine Referenz auf die Server-Version des Branches `serverfix` unter dem Remote-Branch `origin/serverfix` erhalten:

```
$ 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
```

Es ist wichtig zu beachten, dass du bei einem Abruf, der neue Remote-Tracking-Banches abruft, nicht automatisch über lokale, bearbeitbare Kopien davon verfügst. Mit anderen Worten, in diesem Fall hast du keinen neuen Branch `serverfix` – Du hast nur einen Zeiger `origin/serverfix`, den du nicht ändern kannst.

Um diese Änderungen in deinem gegenwärtigen Arbeitsbranch einfließen zu lassen, kannst du die Anweisung `git merge origin/serverfix` ausführen. Wenn du deinen eigenen `serverfix` Branch haben willst, an dem du arbeiten kannst, kannst du ihn von deinem Remote-Tracking-Branch ableiten (engl. base):

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

Das erstellt dir einen lokalen Branch, an dem du arbeiten kannst, und der dort beginnt, wo `origin/serverfix` derzeit steht.

## Tracking-Banches

Das Auschecken eines lokalen Branches von einem Remote-Branch erzeugt automatisch einen sogenannten „Tracking-Branch“ (oder manchmal einen „Upstream-Branch“). Tracking-Banches sind lokale Branches, die eine direkte Beziehung zu einem Remote-Branch haben. Wenn du dich auf einem Tracking-Branch befindest und `git pull` eingibst, weiß Git automatisch, von welchem Server Daten abzuholen sind und in welchen Branch diese einfließen sollen.

Wenn du ein Repository klonst, wird automatisch ein `master` Branch erzeugt, welcher `origin/master` trackt. Du kannst jedoch auch andere Tracking-Banches erzeugen, wenn du wünschst – welche die Branches auf anderen Remotes folgen. Der einfachste Fall ist das Beispiel, dass du gerade gesehen hast, die Ausführung der Anweisung `git checkout -b <branch> <remotename>/<branch>`. Das ist eine übliche Operation, für die Git die Kurzform `--track` bereitstellt:

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

In der Tat ist dies so weit verbreitet, dass es sogar eine Abkürzung für diese Abkürzung gibt. Wenn der Branch-Name, den du zum Auschecken verwenden möchtest (a), nicht existiert und (b) genau mit einem Namen auf nur einem Remote übereinstimmt, erstellt Git einen Tracking-Branch für dich:

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

Um einen lokalen Branch mit einem anderen Namen als den entfernten Branch einzurichten, kannst du die erste Version mit einem anderen lokalen Branch-Namen verwenden:

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

Nun wird dein lokaler Branch `sf` automatisch von `origin/serverfix` pullen.

Wenn du bereits über einen lokalen Branch verfügst und ihn auf einen Remote-Branch festlegen möchtest, den du gerade gepullt hast, oder auf den von dir gefolgten Upstream-Branch ändern

möchtest, kannst du die Option `-u` oder `--set-upstream-to` für `git branch` verwenden, um es jederzeit festzulegen.

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

#### *Upstream-Kürzel*



Wenn du einen Tracking-Branch eingerichtet hast, kannst du auf seinen Upstream-Branch mit der Kurzform `@{upstream}` oder `@{u}` verweisen. Wenn du also auf dem `master` Branch bist und er `origin/master` folgt, kannst du, so etwas wie `git merge @{u}` anstelle von `git merge origin/master` verwenden.

Wenn du die Tracking-Banches sehen willst, die du eingerichtet hast, kannst du die Anweisung `git branch` zusammen mit der Option `-vv` ausführen. Das listet deine lokalen Branches zusammen mit weiteren Informationen auf, einschließlich was jeder Branch trackt und ob dein lokaler Branch voraus oder zurück liegt, oder auch beides.

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

Hier können wir also sehen, dass unser `iss53` Branch den Branch `origin/iss53` folgt und die Information „ahead 2“ bedeutet, dass wir zwei lokale Commits haben, welche noch nicht auf den Server hochgeladen wurden. Wir können außerdem sehen, dass unser `master` Branch `origin/master` folgt und auf den neuesten Stand ist. Als nächstes sehen wir, dass unser `serverfix` Branch den Branch `server-fix-good` auf unserem Server `teamone` folgt. „ahead 3, behind 1“ bedeutet, dass es einen Commit auf dem Server gibt, den wir noch nicht gemerged haben, und drei lokale Commits existieren, die wir noch nicht gepusht haben. Zum Schluss können wir sehen, dass unser `testing` Branch gar keinen Remote-Branch folgt.

Es ist wichtig zu beachten, dass diese Zahlen den Zustand zu dem Zeitpunkt beschreiben, als du zum letzten Mal Daten vom Server abgeholt hast. Diese Anweisung greift nicht auf die Server zu, sie liefert nur die Informationen, welche beim letzten Server-Kontakt lokal zwischengespeichert wurden. Wenn du gänzlich aktuelle Zahlen von „ahead“ und „behind“ willst, dann musst du, kurz bevor du die Anweisung ausführst, von all deinen Remote-Servern Daten abholen (fetch). Du kannst das so machen:

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

## Pulling/Herunterladen

Die Anweisung `git fetch` holt alle Änderungen auf dem Server ab, die du zurzeit noch nicht hast. In deinem Arbeitsverzeichnis wird sie jedoch überhaupt nichts verändern. Sie wird einfach die Daten

für dich holen und dir das Zusammenführen überlassen. Es gibt jedoch die Anweisung `git pull`, welche im Grunde genommen ein `git fetch` ist, dem in den meisten Fällen augenblicklich ein `git merge` folgt. Wenn du einen Tracking-Branch eingerichtet hast, wie im letzten Abschnitt gezeigt, indem du ihn explizit setzt oder indem du ihn mit den Befehlen `clone` oder `checkout` für dich hast erstellen lassen, dann sucht `git pull` nach dem Server und dem getrackten Branch. Git macht ein `fetch` vom Server und versucht dann diesen remote branch zu mergen.

Generell ist es besser, einfach explizit die Anweisungen `git fetch` und `git merge` zu benutzen, da die Magie der Anweisung `git pull` häufig verwirrend sein kann.

## Remote-Branches entfernen

Stellen wir uns vor, du bist mit deinem Remote-Branch fertig. Du und deine Kollegen sind fertig mit einer neuen Funktion und haben sie in den Branch `master` des Remote-Servers (oder in welchem Branch auch immer sich dein stabiler Code befindet) einfließen lassen. Du kannst einen Remote-Branch löschen, indem du die Anweisung `git push` zusammen mit der Option `--delete` ausführst. Wenn du deinen `serverfix` Branch vom Server löschen willst, führst du folgende Anweisung aus:

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

Im Grunde genommen ist alles, was das bewirkt, dass der Zeiger vom Server entfernt wird. Der Git-Server bewahrt die Daten dort in der Regel eine Weile auf, bis eine Speicherbereinigung läuft. Wenn sie also versehentlich gelöscht wurden, ist es oft einfach, sie wiederherzustellen.

## Rebasing

Es gibt bei Git zwei Wege, um Änderungen von einem Branch in einen anderen zu integrieren: `merge` und `rebase`. In diesem Abschnitt wirst du erfahren, was Rebasing ist, wie du es anwendest, warum es ein ziemlich erstaunliches Werkzeug ist und bei welchen Gelegenheiten du es besser nicht einsetzen solltest.

### Einfacher Rebase

Wenn du dich noch mal ein früheres Beispiel aus [Einfaches Merging](#) anschaugst, kannst du sehen, dass du deine Arbeit verzweigt und Commits auf zwei unterschiedlichen Branches erstellt hast.

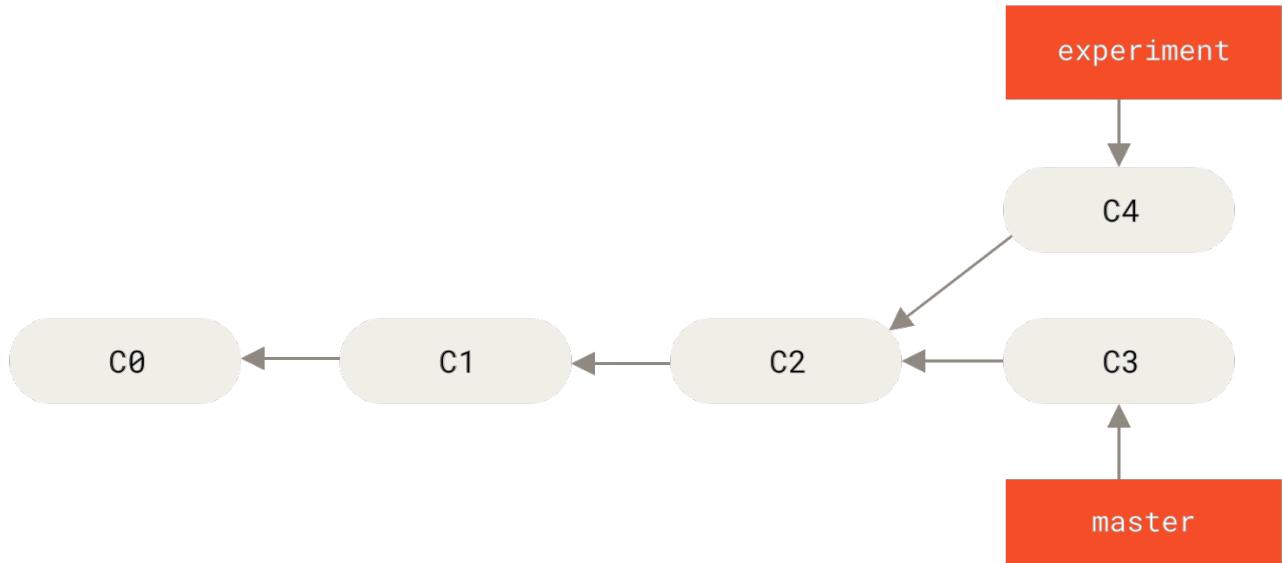


Figure 35. Einfacher verzweigter Verlauf

Der einfachste Weg, die Branches zu integrieren ist der Befehl `merge`, wie wir bereits besprochen haben. Er führt einen Drei-Wege-Merge zwischen den beiden letzten Branch-Snapshots (`C3` und `C4`) und dem jüngsten gemeinsamen Vorgänger der beiden (`C2`) durch und erstellt einen neuen Snapshot (und Commit).

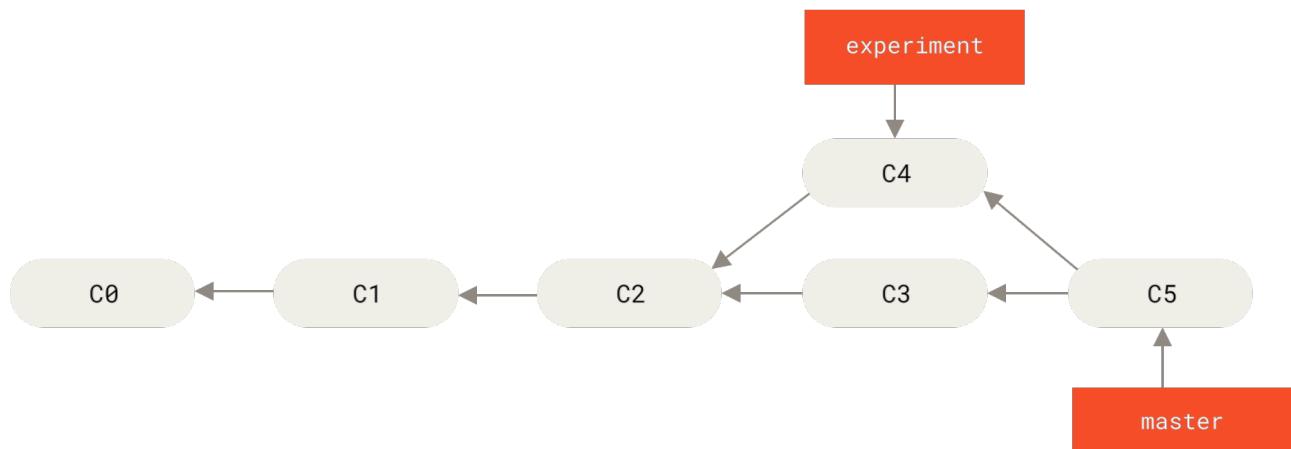


Figure 36. Zusammenführen (Merging) verzweigter Arbeitsverläufe

Allerdings gibt es noch einen anderen Weg: Du kannst den Patch der Änderungen, den wir in `C4` eingeführt haben, nehmen und am Ende von `C3` erneut anwenden. Dieses Vorgehen nennt man in Git `rebasing`. Mit dem Befehl `rebase` kannst du alle Änderungen, die in einem Branch vorgenommen wurden, übernehmen und in einem anderen Branch wiedergeben.

Für dieses Beispiel würdest du den Branch `experiment` auschecken und dann wie folgt auf den `master` Branch restrukturieren (engl. `rebase`):

```

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

```

Dies funktioniert, indem Git zum letzten gemeinsamen Vorgänger der beiden Branches (der, auf dem du arbeitest, und jener, auf den du *rebasen* möchtest) geht, dann die Informationen zu den Änderungen (diffs) sammelt, welche seitdem bei jedem einzelnen Commit des aktuellen Branches gemacht wurden, diese in temporären Dateien speichert, den aktuellen Branch auf den gleichen Commit setzt wie den Branch, auf den du *rebasen* möchtest, und dann alle Änderungen erneut durchführt.

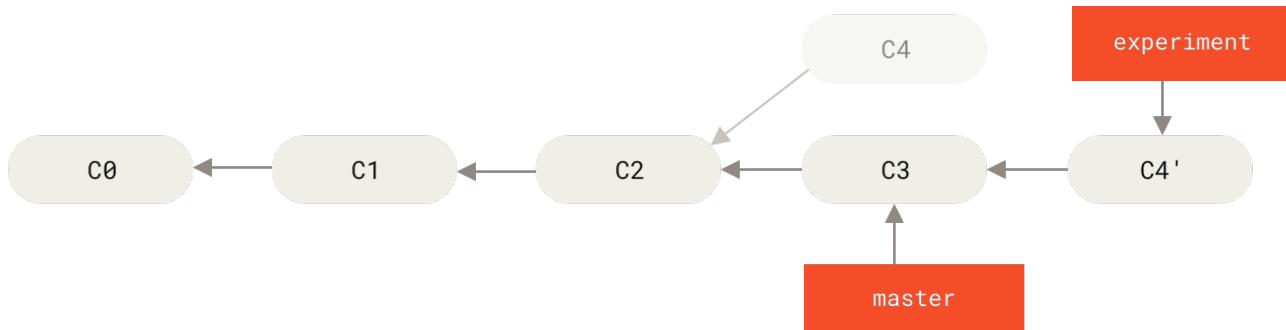


Figure 37. Rebase der in C4 eingeführten Änderung auf C3

An diesem Punkt kannst du zum vorherigen `master` Branch wechseln und einen fast-forward-Merge durchführen.

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

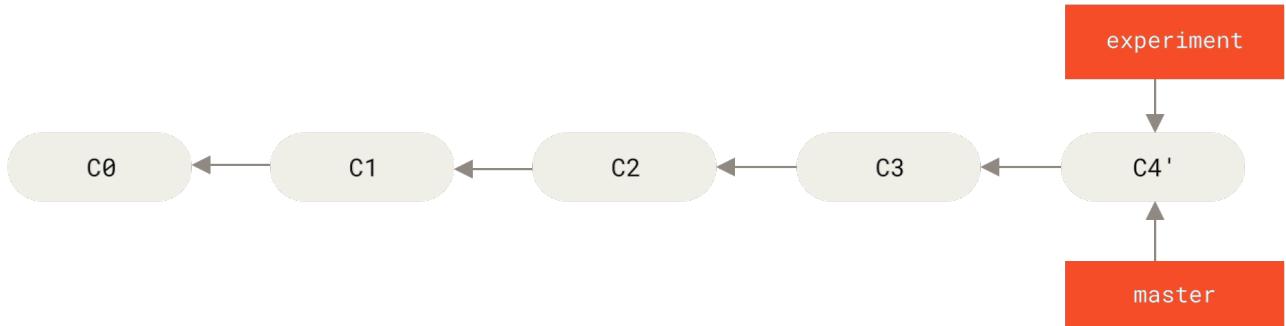


Figure 38. Vorspulen (fast-forwarding) des `master` Branches

Jetzt ist der Schnappschuss, der auf C4' zeigt, exakt derselbe wie derjenige, auf den C5 in dem [Merge-Beispiel](#) gezeigt hat. Es gibt keinen Unterschied im Endergebnis der Integration. Das Rebase sorgt jedoch für eine klarere Historie. Wenn man das Protokoll eines rebase Branches betrachtet, sieht es wie eine lineare Historie aus: Es scheint, dass alle Arbeiten sequentiell stattgefunden hätten, auch wenn sie ursprünglich parallel stattgefunden haben.

Häufig wirst du das anwenden, damit deine Commits sauber auf einen Remote-Branch angewendet werden – vielleicht in einem Projekt, zu dem du beitragen möchtest, das du aber nicht pflegst. Du würdest deine Änderungen in einem lokalen Branch durchführen und diese im Anschluss mittels rebase zu `origin/master` dem Hauptprojekt hinzufügen. Auf diese Weise muss der Maintainer keine Integrationsarbeiten durchführen – nur einen „fast-forward“ oder ein einfaches Einbinden deines Patches.

Beachte, dass der Snapshot, auf welchen der letzte Commit zeigt, ob es nun der letzte des Rebasing-Commits nach einem Rebasing oder der finale Merge-Commit nach einem Merge ist, derselbe Schnappschuss ist. Nur der Verlauf ist ein anderer. Rebasing wiederholt die Änderungsschritte von einer Entwicklungslinie auf einer anderen in der Reihenfolge, in der sie entstanden sind. Dagegen werden beim Merge die beiden Endpunkte der Branches genommen und miteinander gemerged.

## Weitere interessante Rebases

Du kannst dein Rebasing auch auf einen anderen Branch als den Rebasing-Ziel-Branch anwenden. Nimm zum Beispiel einen Verlauf wie im Bild: [Ein Verlauf mit einem Feature-Branch neben einem anderen Feature-Branch](#). Du hast einen Feature-Branch (`server`) angelegt, um ein paar serverseitige Funktionalitäten zu deinem Projekt hinzuzufügen, und hast dann einen Commit gemacht. Anschließend hast du von diesem einen weiteren Branch abgezweigt, um clientseitige Änderungen (`client`) vorzunehmen. Auch hier hast du ein paar Commits durchgeführt. Zum Schluss wechselst du wieder zu deinem vorherigen `server` Branch und machst weitere Commits.

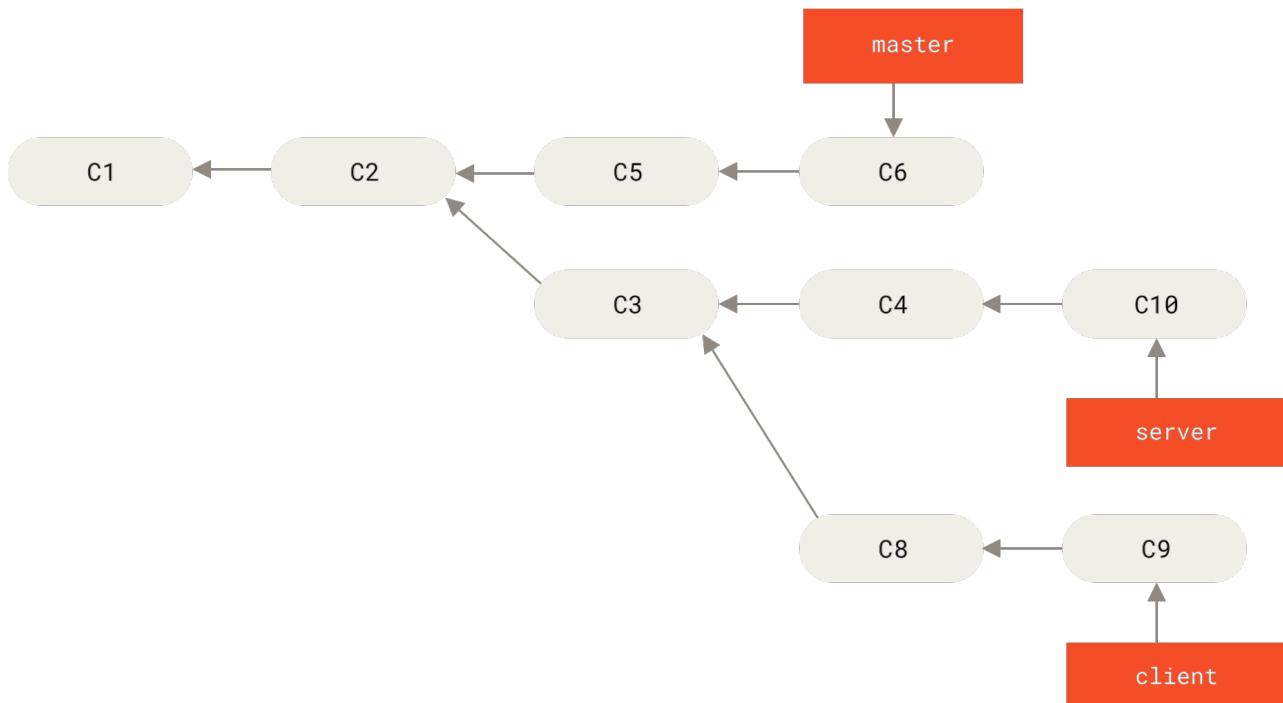


Figure 39. Ein Verlauf mit einem Feature-Branch neben einem anderen Feature-Branch

Angenommen, du entscheidest dich, dass du für einen Release deine clientseitigen Änderungen mit deiner Hauptentwicklungslinie zusammenführst, während du die serverseitigen Änderungen noch zurückhalten willst, bis diese weiter getestet wurden. Du kannst die Änderungen auf dem `client` Branch, die nicht auf dem `server` Branch (`C8` und `C9`) sind, übernehmen und sie in deinem `master` Branch wiedergeben, indem du die Option `--onto` von `git rebase` verwendest:

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

Das bedeutet im Wesentlichen, „Checke den `client` Branch aus, finde die Patches des gemeinsamen Vorgängers der Branches `client` und `server` heraus und wende sie erneut auf den `master` Branch an.“ Das ist ein wenig komplex, aber das Resultat ist ziemlich toll.

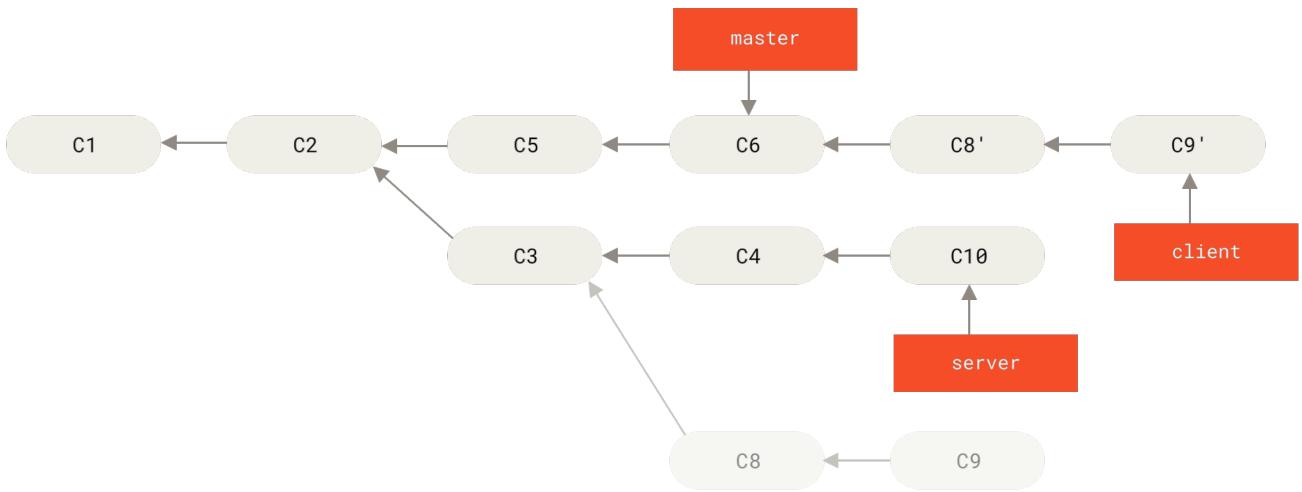


Figure 40. Rebasing eines Themen-Branches aus einem anderen Themen-Branch

Jetzt kannst du deinen Master-Branch vorspulen (engl. fast-forward) (siehe [Vorspulen deines master Branches zum Einfügen der Änderungen des client Branches](#)):

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

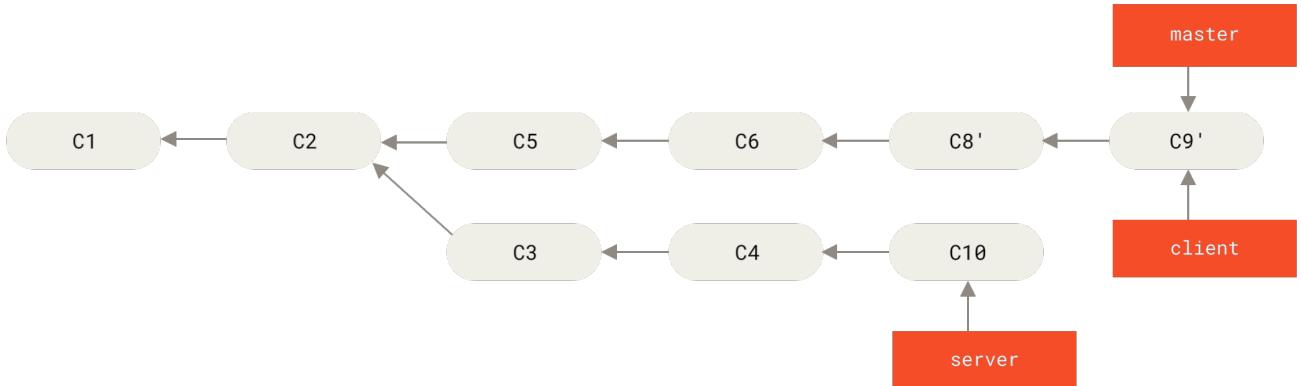


Figure 41. Vorspulen deines master Branches zum Einfügen der Änderungen des client Branches

Lass uns annehmen, du entscheidest dich dazu, deinen `server` Branch ebenfalls einzupflegen. Du kannst das Rebase des `server` Branches auf den `master` Branch anwenden, ohne diesen vorher auschecken zu müssen, indem du die Anweisung `git rebase < Basis-Branch > < Feature-Branch >` ausführst, welche für dich den Feature-Branch auscheckt (in diesem Fall `server`) und ihn auf dem Basis-Branch (`master`) wiederholt:

```
$ git rebase master server
```

Das wiederholt deine Änderungen aus dem `server` Branch am Ende des `master` Branches, wie in [Rebase deines server Branches am Ende deines master Branches](#) gezeigt wird.

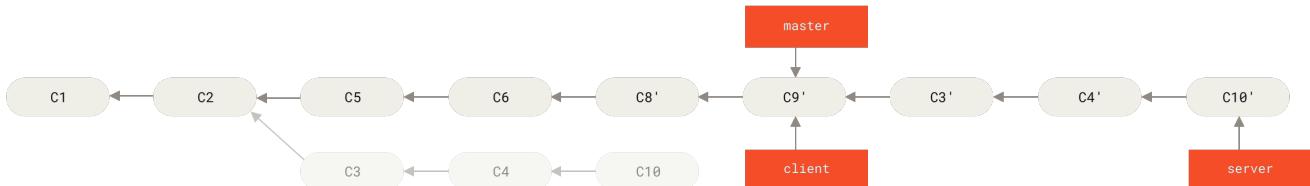


Figure 42. Rebasing deines `server` Branches am Ende deines `master` Branches

Dann kannst du den Basis-Branch (`master`) vorspulen (engl. fast-forward):

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

Du kannst die Branches `client` und `server` löschen, da die ganze Arbeit bereits in `master` integriert wurde und du diese nicht mehr benötigst. Dein Verlauf für diesen gesamten Prozess sieht jetzt wie in [Endgültiger Commit-Verlauf](#) aus:

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

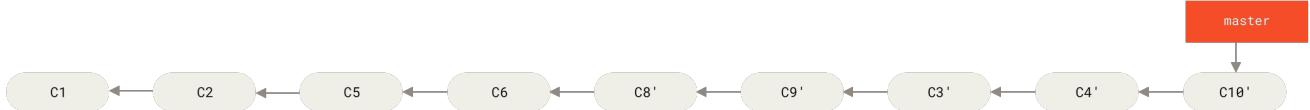


Figure 43. Endgültiger Commit-Verlauf

## Die Gefahren des Rebasing

Ahh, aber der ganze Spaß mit dem Rebasen kommt nicht ohne Schattenseiten und Fallstricke, welche in einer einzigen Zeile zusammengefasst werden können:

**Führe keinen Rebase mit Commits durch, die außerhalb deines Repositorys existieren und auf welche die Arbeit anderer Personen basiert.**

Wenn du dich an diese Leitlinie hältst, wirst du gut zurechtkommen. Wenn du es nicht tust, werden die Leute dich hassen und du wirst von Freunden und Familie verschmäht werden.

Wenn du ein Rebase durchführst, entfernst du bestehende Commits und erstellen stattdessen neue, die zwar ähnlich aber dennoch unterschiedlich sind. Stell dir vor, du lädst diese Commits hoch und andere laden sich diese herunter und nehmen sie als Grundlage für ihre Arbeit. Dann änderst du jedoch ihre commits nochmal und rebasen und pushst sie. Deine Kollegen müssen ihre Änderungen nochmal remergen. Wenn sie nun versuchen diesen remerge bei sich zu pullen, wird das nicht funktionieren und es kommt zu einem heillosen Durcheinander.

Schauen wir uns ein Beispiel an, wie ein Rebase von Arbeiten, die du öffentlich gemacht hast, Probleme verursachen kann. Nehmen wir an, du klonst ein Repository von einem zentralen Server und machst ein paar Änderungen. Dein Commit-Verlauf sieht anschließend so aus:

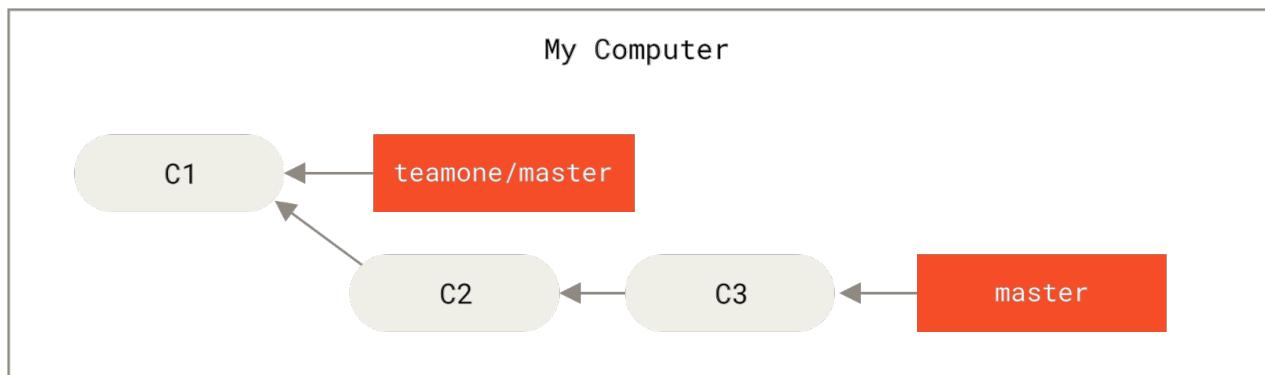
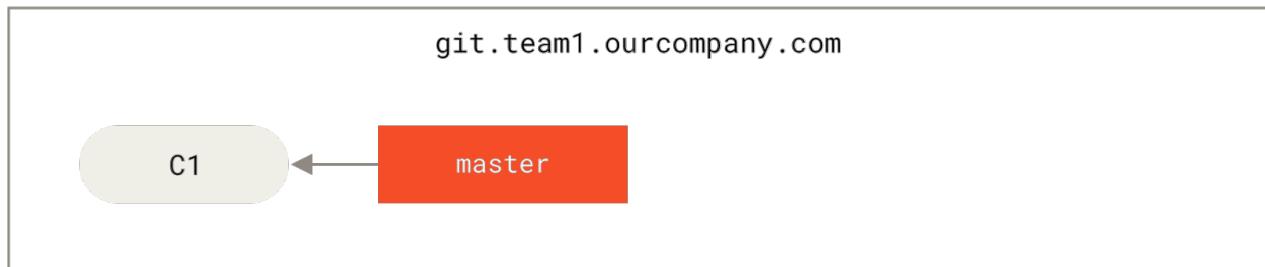


Figure 44. Klonen eines Repositorys und darauf Arbeit aufbauen

Nun macht jemand anderes Änderungen am Code, einschließlich eines Merges und pusht diese dann auf den zentralen Server. Du holst die Änderungen ab und mergest den neuen Remote-Branch mit deiner Arbeit, sodass dein Verlauf wie folgt aussieht.

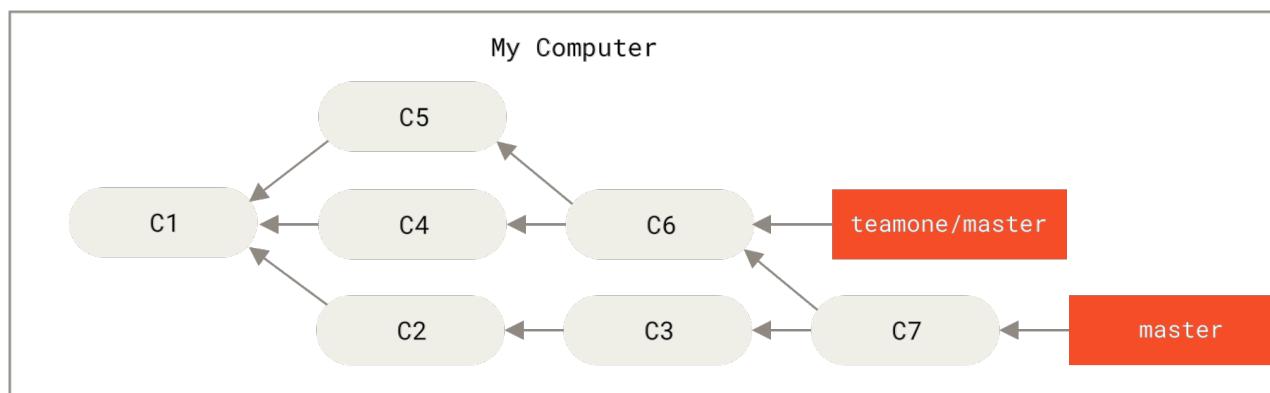
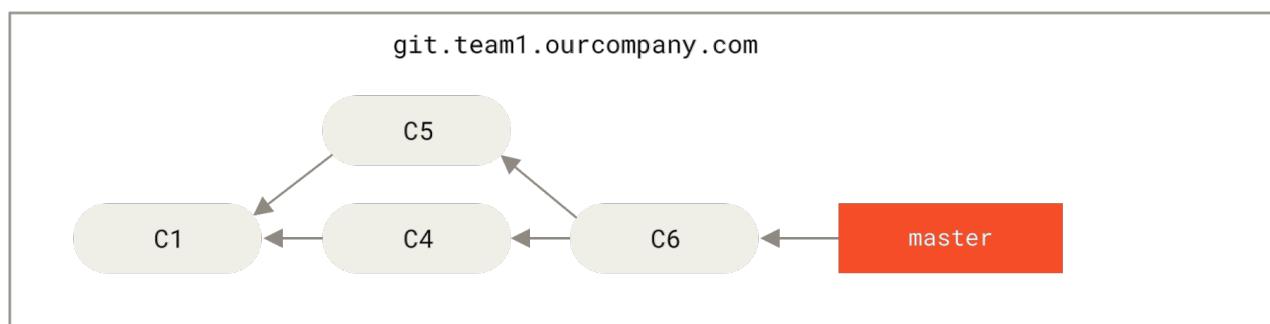


Figure 45. Weitere Commits abholen und mergen mit deiner Arbeit

Als nächstes entscheidet die Person, welche die gemergte Arbeit hochgeladen hat, diese rückgängig

zu machen und stattdessen deine Arbeit mittels Rebase hinzuzufügen. Sie führt dazu die Anweisung `git push --force` aus, um den Verlauf auf dem Server zu überschreiben. Du holst das Ganze dann von diesem Server ab und lädst die neuen Commits herunter.

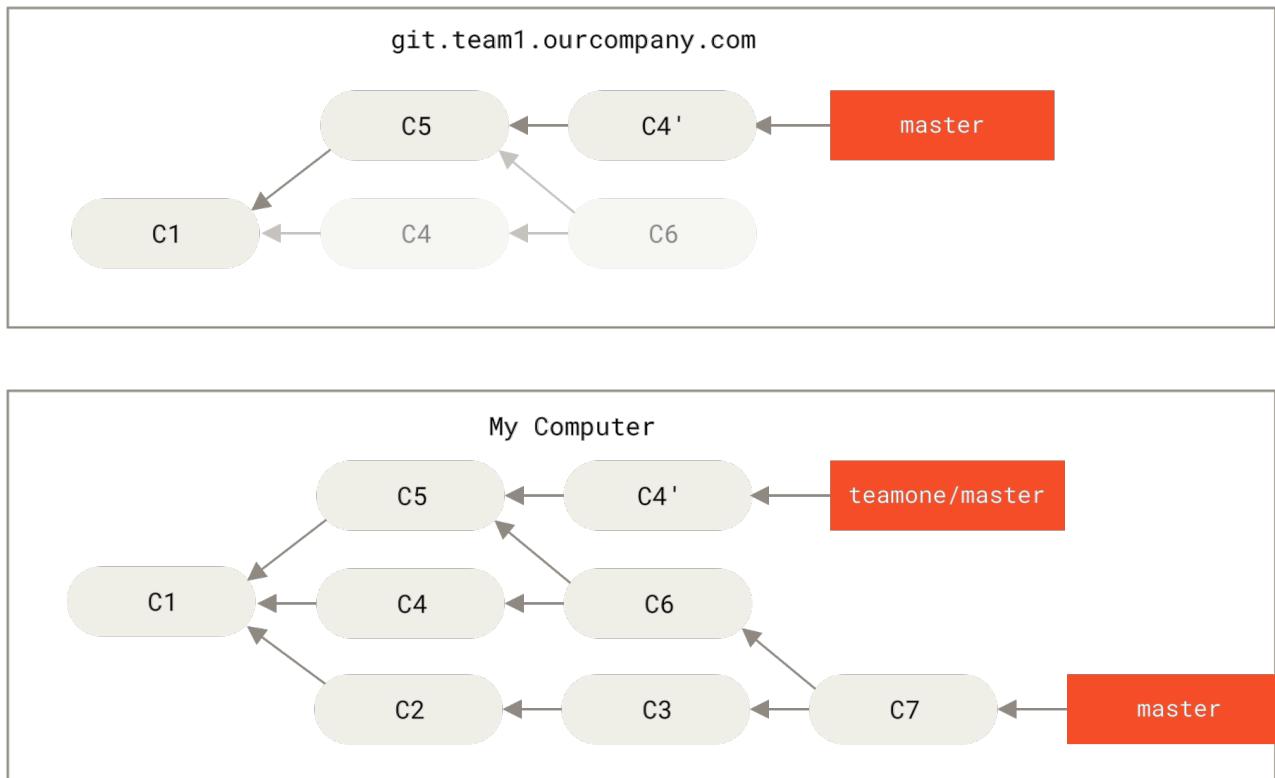


Figure 46. Jemand lädt Commits nach einem Rebase hoch und verwirft damit Commits, auf denen deine Arbeit basiert

Jetzt sitzt ihr beide in der Klemme. Wenn du ein `git pull` durchführst, würdest du einen Merge-Commit erzeugen, welcher beide Entwicklungslinien einschließt und dein Repository würde so aussehen:

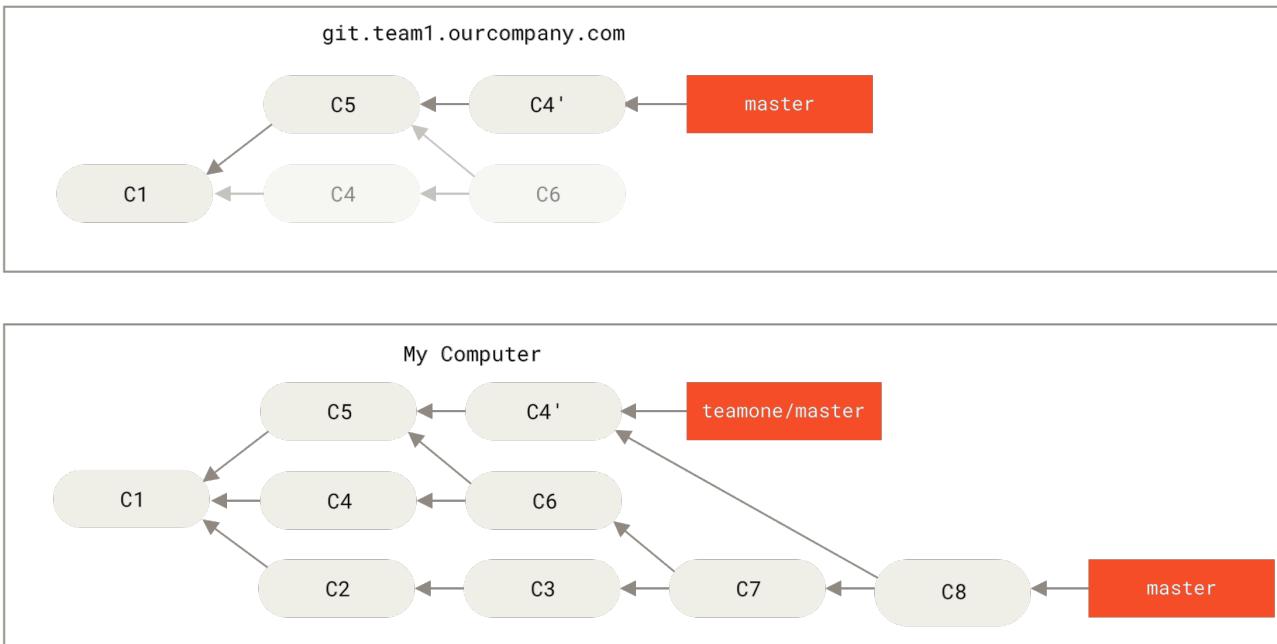


Figure 47. Du lässt die Änderungen nochmals in dieselbe Arbeit einfließen in einen neuen Merge-Commit

Falls du ein `git log` ausführst, wenn dein Verlauf so aussieht, würdest du zwei Commits sehen, bei denen Autor, Datum und Nachricht übereinstimmen, was verwirrend ist. Weiter würdest du, wenn du diesen Verlauf zurück auf den Server pushst, alle diese vom Rebase stammenden Commits auf dem zentralen Server ablegen, was die Kollegen noch weiter durcheinander bringen würde. Man kann ziemlich sicher davon ausgehen, dass der andere Entwickler **C4** und **C6** nicht im Verlauf haben möchte; das ist der Grund, warum derjenige das Rebase überhaupt gemacht hat.

## Rebasen, wenn du Rebase durchführst

Wenn du dich in einer solchen Situation **befindest**, hat Git eine weitere magische Funktion, die dir helfen könnte. Falls jemand in deinem Team forcierte Änderungen pusht, die Arbeiten überschreiben, auf denen deine basiert, besteht deine Herausforderung darin, herauszufinden, was dir gehört und was andere überschrieben haben.

Es stellt sich heraus, dass Git neben der SHA-1-Prüfsumme eine weitere Prüfsumme berechnet, die nur auf den mit dem Commit eingeführten Änderungen basiert. Diese wird „patch-id“ genannt.

Wenn du die neu umgeschriebene Änderungen pullen und ein Rebase auf auf die neuen Commits deines Partners ausführst, kann Git oft erfolgreich herausfinden, was nur von dir ist und kann es entsprechend auf den neuen Branch anwenden.

Sobald wir im vorhergehenden Szenario, beispielsweise bei **Jemand lädt Commits nach einem Rebase hoch und verwirft damit Commits, auf denen deine Arbeit basiert**, die Anweisung `git rebase teamone/master` ausführen, anstatt ein Merge durchzuführen, dann wird Git:

- bestimmen, welche Änderungen an unserem Branch einmalig sind (**C2, C3, C4, C6, C7**),
- bestimmen, welche der Commits keine Merge-Commits sind (**C2, C3, C4**),
- bestimmen, welche Commits nicht neu in den Zielbranch geschrieben wurden (nur **C2** und **C3**, da **C4** der selbe Patch wie **C4'** ist), und

- diese Commits am Ende des `teamone/master` Branches anwenden.

Statt des Ergebnisses, welches wir in `Du lässt die Änderungen nochmals in dieselbe Arbeit einfließen in einen neuen Merge-Commit` sehen, würden wir etwas erhalten, was eher wie `Rebase` am Ende von Änderungen eines „`force-pushed`“-`Rebase` aussieht.

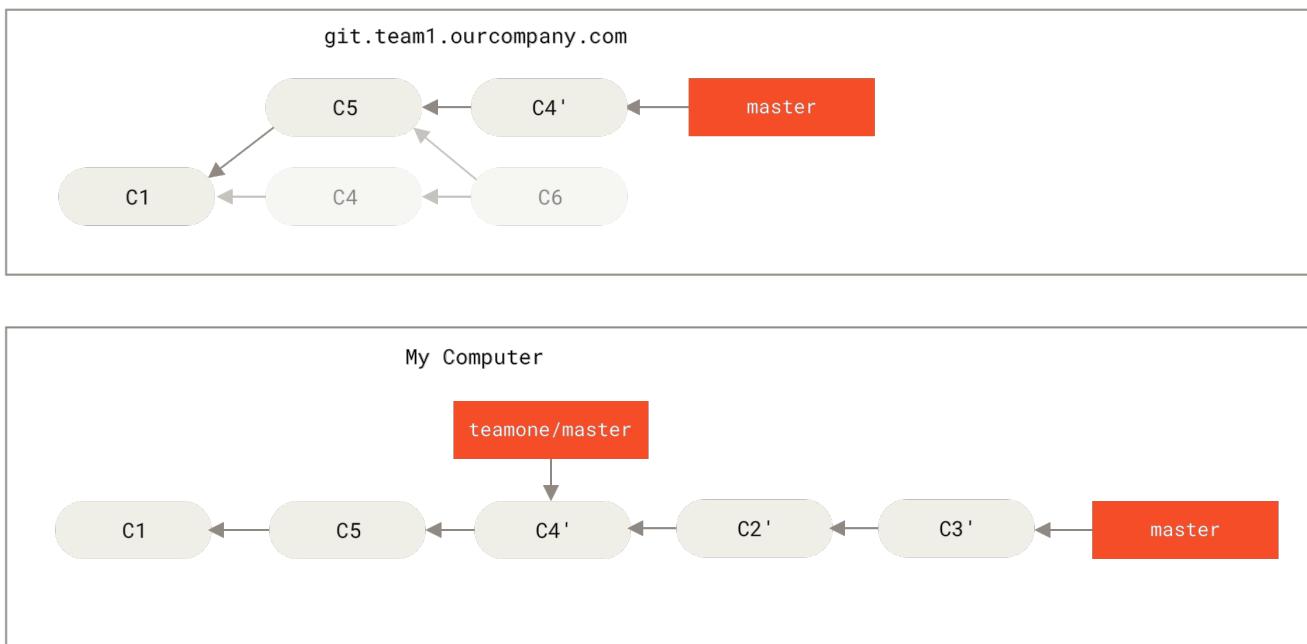


Figure 48. *Rebase am Ende von Änderungen eines „force-pushed“-Rebase*

Das funktioniert nur, wenn es sich bei `C4` und `C4'`, welche dein Teamkollege erstellt hat, um fast genau denselben Patch handelt. Andernfalls kann das rebase nicht erkennen, dass es sich um ein Duplikat handelt und fügt einen weiteren, dem Patch `C4` ähnlichen, hinzu (der wahrscheinlich nicht sauber angewendet werden kann, da die Änderungen bereits vollständig oder zumindest teilweise vorhanden sind).

Du kannst das auch vereinfachen, indem du ein `git pull --rebase` anstelle eines normalen `git pull` verwendest. Oder du kannst es manuell mit einem `git fetch` machen, in diesem Fall gefolgt von einem `git rebase teamone/master`.

Wenn du `git pull` benutzt und `--rebase` zur Standardeinstellung machen willst, kannst du den `pull.rebase` Konfigurationswert mit etwas wie `git config --global pull.rebase true` einstellen.

Wenn du nur Commits rebased, die noch nie deinen eigenen Computer verlassen haben, ist alles in Ordnung. Wenn du Commits, die gepusht wurden, aber niemand sonst hat, basierend auf den Commits, rebased, wird auch alles in Ordnung sein. Wenn du Commits rebased, die gepusht wurden, auf denen aber keine Commits von jemand anderen basieren, ist auch alles in Ordnung. Wenn du Commits, die bereits veröffentlicht wurden, rebased und die Arbeit anderer Leute basiert auf diese Commits, dann könntest du Probleme bekommen und von deinen Teamkollegen verhöhnt werden.

Wenn du oder ein Partner es irgendwann für unbedingt notwendig halten, stelle sicher, dass jeder weiß, dass er anschließend `git pull --rebase` laufen lassen muss. So kann man versuchen, den Schaden einzuschränken, nachdem er passiert ist.

## Rebase vs. Merge

Nachdem du jetzt Rebasing und Merging in Aktion erlebt hast, fragst du dich vielleicht, welches davon besser ist. Bevor wir das beantworten können, lass uns ein klein wenig zurückblicken und darüber reden, was Historie bedeutet.

Ein Standpunkt ist, dass der Commit-Verlauf deines Repositorys eine **Aufzeichnung davon ist, was wirklich passiert ist**. Es ist ein wertvolles Dokument, das nicht manipuliert werden sollte. Aus diesem Blickwinkel ist das Ändern der Commit-Historie fast schon blasphemisch. Man *belügt sich* über das, was tatsächlich passiert ist. Was wäre, wenn es eine verwirrende Reihe von Merge-Commits gäbe? So ist es nun mal passiert, und das Repository sollte das für die Nachwelt beibehalten.

Der entgegengesetzte Standpunkt ist, dass der Commit-Verlauf den **Verlauf deines Projekts** darstellt. Du würdest den ersten Entwurf eines Buches niemals veröffentlichen, warum also deine unordentliche Arbeit? Wenn du an einem Projekt arbeitest, benötigst du möglicherweise eine Aufzeichnung all deiner Fehlritte und Sackgassen. Wenn es jedoch an der Zeit ist, deine Arbeit der Welt zu zeigen, möchtest du möglicherweise eine kohärentere Geschichte darüber erzählen, wie du von A nach B gekommen bist. Die Leute in diesem Camp verwenden Tools wie Rebase und Filter-Branch, um ihre Commits neu zu schreiben, bevor sie in den Haupt-Branch integriert werden. Sie verwenden Tools wie **Rebase** und **Filter-Branch**, um die Geschichte so zu erzählen, wie es für zukünftige Leser am besten ist.

Nun zur Frage, ob Mergen oder Rebasen besser ist. Wie so oft, ist diese Frage nicht so leicht zu beantworten. Git ist ein mächtiges Werkzeug und ermöglicht es dir, viele Dinge mit deinem Verlauf anzustellen. Aber jedes Team und jedes Projekt ist anders. Jetzt, da du weißt, wie diese beiden Möglichkeiten funktionieren, liegt es an dir, zu entscheiden, welche für deine Situation die Beste ist.

Für gewöhnlich lassen sich die Vorteile von beiden Techniken nutzen: Rebase lokale Änderungen vor einem Push, um deinen Verlauf zu bereinigen, aber rebase niemals etwas, das du bereits gepusht hast.

## Zusammenfassung

Wir haben einfaches Branching und Merging mit Git besprochen. Es sollte dir leicht fallen, neue Branches zu erstellen und zu diesen zu wechseln, zwischen bestehenden Branches zu wechseln und lokale Branches zusammenzuführen (engl. mergen). Außerdem solltest du in der Lage sein, deine Branches auf einem gemeinsam genutzten Server bereitzustellen, mit anderen an gemeinsam genutzten Branches zu arbeiten und deren Branches zu rebasen, bevor du diese bereitstellen. Als nächstes werden wir dir zeigen, was du brauchst, um deinen eigenen Git-Repository-Hosting-Server zu betreiben.

# Git auf dem Server

An dieser Stelle solltest du in der Lage sein, die meisten der täglichen Aufgaben zu erledigen, für die du Git verwenden wirst. Um jedoch in Git mit Anderen zusammenarbeiten zu können, benötigst du ein externes Git-Repository. Obwohl du, technisch gesehen, Änderungen in und aus den Repositorys deiner Mitstreiter schieben kannst, ist das nicht empfehlenswert, da das ziemlich leicht zu Verwirrung führen kann, wenn du nicht sehr vorsichtig bist. Darüber hinaus ist es vorteilhaft, dass deine Mitstreiter auch dann auf das Repository zugreifen können, wenn dein Computer offline ist – ein zuverlässigeres gemeinsames Repository ist oft sinnvoll. Daher ist die bevorzugte Methode für die Zusammenarbeit, einen Zwischenspeicher einzurichten, auf den beide Seiten Zugriff haben, und von dem aus sie Push-to and Pull ausführen können.

Das Betreiben eines Git-Servers ist recht unkompliziert. Zuerst bestimmst du, welche Protokolle dein Server unterstützen soll. Der erste Abschnitt dieses Kapitels behandelt die verfügbaren Protokolle und deren Vor- und Nachteile. In den nächsten Abschnitten werden einige typische Setups mit diesen Protokollen erläutert und erklärt, wie du deinen Server mit diesen Protokollen zum Laufen bringst. Zuletzt werden wir ein paar gehostete Optionen durchgehen, falls du nicht den Aufwand der Einrichtung und Wartung deines eigenen Git-Servers auf dich nehmen willst.

Wenn du keinen eigenen Server betreiben möchtest, kannst du direkt zum letzten Abschnitt dieses Kapitels springen. Dort sind einige Optionen zum Einrichten eines gehosteten Kontos zu finden. Anschließend kannst du mit dem nächsten Kapitel fortfahren, in dem die verschiedenen Vor- und Nachteile der Arbeit in einer verteilten Versionskontrollumgebung erläutert werden.

Ein entferntes Repository ist in der Regel ein „*nacktes Repository*“ – ein Git-Repository, das kein Arbeitsverzeichnis hat. Da das Repository nur als Kollaborationspunkt verwendet wird, gibt es erstmal keinen Grund, einen Snapshot auf die Festplatte speichern zu lassen; es enthält nur die Git-(Kontroll-)Daten. Im einfachsten Fall besteht ein nacktes (eng. bare) Repository nur aus dem Inhalt des `.git` Verzeichnisses deines Projekts.

## Die Protokolle

Git kann vier verschiedene Protokolle für die Datenübertragung verwenden: Lokal, HTTP, Secure Shell (SSH) und Git. Hier werden wir klären, worum es sich handelt und unter welchen Rahmenbedingungen du sie verwenden oder nicht verwenden solltest.

### Lokales Protokoll

Das einfachste ist das *lokale Protokoll*, bei dem sich das entfernte Repository in einem anderen Verzeichnis auf demselben Host befindet. Es wird häufig verwendet, wenn jeder in deinem Team Zugriff auf ein freigegebenes Dateisystem wie z.B. ein NFS-Mount hat, oder in dem selteneren Fall, dass sich jeder auf dem gleichen Computer anmeldet. Letzteres wäre nicht ideal, da sich alle deine Code-Repository-Instanzen auf demselben Computer befinden würden, was einen Totalverlust deiner Daten viel wahrscheinlicher macht.

Wenn Du ein geteiltes und gemountetes Dateisystem hast, kannst du von einem lokalem dateibasierten Repository klonen, etwas dort hin verschieben (engl. push to) und etwas daraus ziehen (engl. pull). Um solch ein repository zu klonen oder um es als remote zu einem

existierenden Projekt hinzuzufügen, nutze einfach den Pfad zum Repository als URL. Um beispielsweise ein lokales Repository zu klonen, kannst du Folgendes ausführen:

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

oder auch das:

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

Git verhält sich ein wenig anders, wenn du `file://` explizit am Anfang der URL angibst. Wenn du nur den Pfad angibst, versucht Git, Hardlinks zu verwenden oder die benötigten Dateien direkt zu kopieren. Wenn du `file://` angibst, löst Git Prozesse aus, die normalerweise zum Übertragen von Daten über ein Netzwerk verwendet werden, was im Allgemeinen viel weniger effizient ist. Der Hauptgrund für die Angabe des Präfix `file://` ist, wenn du eine saubere Kopie des Repositorys mit irrelevanten Referenzen oder weggelassenen Objekten wünschst – in der Regel nach einem Import aus einem anderen VCS oder ähnlichem (siehe [Git Interna](#) für Wartungsaufgaben). Wir werden hier den normalen Pfad verwenden, denn das ist fast immer schneller.

Um ein lokales Repository zu einem bestehenden Git-Projekt hinzuzufügen, kann folgendermaßen vorgegangen werden:

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

Dann kannst du über deinen neuen Remote-Namen `local_proj` auf dieses Remote-Repository pushen und von dort abrufen, so wie du es auch über ein Netzwerk tun würdest.

## Vorteile

Die Vorteile dateibasierter Repositorys liegen darin, dass sie simpel sind und vorhandene Datei- und Netzwerk-Berechtigungen verwenden. Wenn du bereits über ein freigegebenes Dateisystem verfügst, auf das dein gesamtes Team Zugriff hat, ist das Einrichten eines solchen Repositorys sehr einfach. Du speicherst die leere Repository-Kopie an einer Stelle, auf die jeder Zugriff hat und legst die Lese- und Schreibberechtigungen wie bei jedem anderen freigegebenen Verzeichnis fest. Informationen zum Exportieren einer Bare-Repository-Kopie für diesen Zweck findest du unter [Git auf einem Server installieren](#).

Das ist auch eine elegante Möglichkeit, um schnell Arbeiten aus dem Arbeits-Repository eines anderen zu holen. Wenn du und ein Mitstreiter am gleichen Projekt arbeiten und du etwas überprüfen möchtest, ist es oft einfacher, einen Befehl wie `git pull /home/john/project` auszuführen, als auf einen Remote-Server zu pushen und anschließend von dort zu holen.

## Nachteile

Der Nachteil dieser Methode ist, dass der gemeinsame Zugriff in der Regel schwieriger einzurichten ist damit man von mehreren Standorten aus erreichbar ist als der einfache Netzwerkzugriff. Wenn du von zu Hause mit deinem Laptop aus pushen möchtest, musst du das entfernte Verzeichnis

einhängen (engl. mounten), was im Vergleich zum netzwerkbasierten Zugriff schwieriger und langsamer sein kann.

Es ist wichtig zu erwähnen, dass es nicht unbedingt die schnellste Option ist, wenn du einen gemeinsamen Mount verwendest. Ein lokales Repository ist nur dann schnell, wenn du schnellen Zugriff auf die Daten hast. Ein Repository auf NFS-Mounts ist oft langsamer als der Zugriff über SSH auf demselben Server, so kann Git Daten auf lokalen Festplatten auf jedem System speichern.

Schließlich schützt dieses Protokoll das Repository nicht vor unbeabsichtigten Schäden. Jeder Benutzer hat vollen Shell-Zugriff auf das „remote“ Verzeichnis, und nichts hindert ihn daran, interne Git-Dateien zu ändern oder zu entfernen und das Repository zu beschädigen.

## HTTP Protokolle

Git kann über HTTP in zwei verschiedenen Modi kommunizieren. Vor Git 1.6.6 gab es nur einen einzigen Weg, der sehr einfach und im Allgemeinen „read-only“ war. Mit der Version 1.6.6 wurde ein neues, intelligenteres Protokoll eingeführt, bei dem Git in der Lage ist, den Datentransfer intelligent auszuhandeln, ähnlich wie bei SSH. In den letzten Jahren ist dieses neue HTTP-Protokoll sehr beliebt geworden, da es für den Benutzer einfacher und intelligenter in der Kommunikation ist. Die neuere Version wird oft als *Smart* HTTP Protokoll und die ältere als *Dumb* HTTP bezeichnet. Wir werden zuerst das neuere Smart HTTP-Protokoll besprechen.

### Smart HTTP

Smart HTTP funktioniert sehr ähnlich wie die Protokolle SSH oder Git, läuft aber über Standard HTTPS-Ports und kann verschiedene HTTP-Authentifizierungsmechanismen verwenden. Das bedeutet, dass es für den Benutzer oft einfacher ist als so etwa SSH, da du Eingaben wie Benutzername/Passwort-Authentifizierung verwenden kannst, anstatt SSH-Schlüssel einrichten zu müssen.

Es ist wahrscheinlich der beliebteste Weg, heutzutage Git zu verwenden, da es so eingerichtet werden kann, dass es sowohl anonym wie das Protokoll `git://` arbeitet, als auch mit Authentifizierung und Verschlüsselung wie das SSH-Protokoll betrieben werden kann. Anstatt dafür verschiedene URLs einrichten zu müssen, kannst du nun eine einzige URL für beides verwenden. Wenn du versuchst, einen Push durchzuführen und das Repository eine Authentifizierung erfordert (was normalerweise der Fall sein sollte), kann der Server nach einem Benutzernamen und einem Passwort fragen. Gleiches gilt für den Lesezugriff.

Für Dienste, wie GitHub, ist die URL, die du verwendest, um das Repository online anzuzeigen (z.B. <https://github.com/schacon/simplegit>), die gleiche URL, mit der du klonen und, wenn du Zugriff hast, pushen kannst.

### Dumb HTTP

Wenn der Server nicht mit einem Git HTTP Smart Service antwortet, versucht der Git Client, auf das einfachere *Dumb* HTTP Protokoll zurückzugreifen. Das Dumb-Protokoll erwartet von dem Bare-Git-Repository, dass der Webserver seine Dateien ganz normal zur Verfügung stellt. Das Schöne an Dumb HTTP ist die einfache Einrichtung. Im Grunde genommen musst du nur ein leeres Git-Repository unter deinem HTTP-Dokument-Root legen und einen bestimmten `post-update` Hook

einrichten, und schon bist du fertig (siehe [Git Hooks](#)). Ab diesem Zeitpunkt kann jeder, der auf den Webserver zugreifen kann, auf dem das Repository liegt, dein Repository klonen. Um Lesezugriff auf dein Repository über HTTP zu ermöglichen, gehe wie folgt vor:

```
$ 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
```

Das war's. Der `post-update` Hook, der standardmäßig mit Git geliefert wird, führt den entsprechenden Befehl (`git update-server-info`) aus, um den Abruf- und Klonvorgang über HTTP ordnungsgemäß zu ermöglichen. Dieser Befehl wird ausgeführt, wenn du in dieses Repository pushst (z.B. über SSH); dann können andere folgendermaßen Klonen:

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

In diesem speziellen Fall verwenden wir den Pfad `/var/www/htdocs`, der für Apache-Installationen üblich ist. Du kannst aber jeden statischen Webserver verwenden – lege einfach das leere Repository in seinen Pfad. Die Git-Daten werden als einfache statische Dateien bereitgestellt (siehe Kapitel [Git Interna](#) für Bedienungsdetails).

Normalerweise würdest du entweder einen Smart HTTP-Server zum Lesen und Schreiben betreiben oder die Dateien einfach schreibgeschützt im Dumb-Modus zur Verfügung stellen. Ein Mix aus beiden Diensten wird selten angeboten.

## Vorteile

Wir wollen uns auf die Vorteile der Smart Version des HTTP-Protokolls konzentrieren.

Die Tatsache, dass eine einzige URL für alle Zugriffsarten und der Server-Prompt nur dann gebraucht wird, wenn eine Authentifizierung erforderlich ist, macht die Sache für den Endbenutzer sehr einfach. Die Authentifizierung mit einem Benutzernamen und einem Passwort ist ebenfalls ein großer Vorteil gegenüber SSH, da Benutzer keine SSH-Schlüssel lokal generieren und ihren öffentlichen Schlüssel auf den Server hochladen müssen, bevor sie mit ihm interagieren können. Für weniger anspruchsvolle Benutzer oder Benutzer auf Systemen, auf denen SSH weniger verbreitet ist, ist dies ein großer Vorteil in der Benutzerfreundlichkeit. Es ist auch ein sehr schnelles und effizientes Protokoll, ähnlich dem SSH-Protokoll.

Du kannst dein Repositorys auch schreibgeschützt über HTTPS bereitstellen, d.h. du kannst die Inhaltsübertragung verschlüsseln oder sogar so weit gehen, dass Clients bestimmte signierte SSL-Zertifikate verwenden müssen.

Eine weitere schöne Sache ist, dass HTTP und HTTPS ein so häufig verwendetes Protokoll ist. Unternehmens-Firewalls sind oft so eingerichtet, dass sie den Datenverkehr über deren Ports erlauben.

## Nachteile

Git über HTTPS kann im Vergleich zu SSH auf einigen Servern etwas komplizierter einzurichten sein. Abgesehen davon gibt es sehr wenig Vorteile, die andere Protokolle gegenüber Smart HTTP für die Bereitstellung von Git-Inhalten haben.

Wenn du HTTP für authentifiziertes Pushen verwendest, ist die Bereitstellung deiner Anmeldeinformationen manchmal komplizierter als die Verwendung von Schlüsseln über SSH. Es gibt jedoch mehrere Tools zum Zwischenspeichern von Berechtigungen, die du verwenden kannst, darunter Keychain-Zugriff auf macOS und Credential Manager unter Windows, um das zu Vereinfachen. Lese den Abschnitt [Credential Storage](#), um zu erfahren, wie du ein sicheres HTTP-Passwort-Caching auf deinem System einrichten kannst.

## SSH Protocol

Ein gängiges Git-Transportprotokoll für Self-Hosting ist SSH. Der SSH-Zugriff auf den Server ist in den meisten Fällen bereits eingerichtet – und wenn nicht, ist es einfach zu bewerkstelligen. SSH ist auch ein authentifiziertes Netzwerkprotokoll, und da es allgegenwärtig ist, ist es im Allgemeinen einfach einzurichten und zu verwenden.

Um ein Git-Repository über SSH zu klonen, kannst du eine entsprechende `ssh://` URL angeben:

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

Oder kannst die kürzere scp-ähnliche Syntax für das SSH-Protokoll verwenden:

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

Wenn du in beiden Fällen oben keinen optionalen Benutzernamen angibst, benutzt Git den User, mit dem du aktuell angemeldet bist.

## Vorteile

Die Vorteile bei der Verwendung von SSH sind vielfältig. Erstens ist SSH relativ einfach einzurichten – SSH-Daemons sind weit verbreitet, viele Netzwerkadministratoren haben Erfahrung mit ihnen und viele Betriebssystem-Distributionen werden mit ihnen eingerichtet oder haben Werkzeuge, um sie zu verwalten. Als nächstes ist der Zugriff über SSH sicher – der gesamte Datentransfer wird verschlüsselt und authentifiziert. Schließlich ist SSH, wie die Protokolle HTTPS, Git und Local effizient und komprimiert die Daten vor der Übertragung so stark wie möglich.

## Nachteile

Die negative Seite von SSH ist, dass es keinen anonymen Zugriff auf dein Git-Repository unterstützt. Wenn du SSH verwendest, müssen Benutzer über einen SSH-Zugriff auf deinen Computer verfügen, auch wenn sie nur über Lesezugriff verfügen. Das macht SSH in Open Source-Projekten ungeeignet. Bspw. wenn die Benutzer dein Repository einfach nur klonen möchten, um es zu überprüfen. Wenn du es nur in deinem Unternehmensnetzwerk verwendest, ist SSH möglicherweise das einzige Protokoll, mit dem du dich befassen musst. Wenn du anonymen schreibgeschützten Zugriff auf

deine Projekte und auch SSH zulassen möchtest, musst du SSH einrichten, damit du Push-Vorgänge ausführen kannst. Zusätzlich musst du jedoch noch etwas anderes konfigurieren damit Benutzer auch anonym abrufen können.

## Git Protokoll

Zuletzt haben wir noch das Git-Protokoll. Es ist ein spezieller Daemon, der mit Git ausgeliefert wird. Er lauscht auf einem dedizierten Port (9418) und stellt einen Dienst bereit, ähnlich dem des SSH-Protokolls, jedoch ohne jegliche Authentifizierung oder Verschlüsselung. Damit ein Repository über das Git-Protokoll bedient werden kann, musst du eine `git-daemon-export-ok` Datei erstellen – der Daemon wird ohne diese Datei kein Repository bedienen, da es ansonsten keine Sicherheit gibt. Entweder ist das Git-Repository für jeden oder für niemanden zum klonen zugänglich. Das bedeutet, dass es in der Regel keinen Push über dieses Protokoll gibt. Du kannst den Push-Zugriff aktivieren, aber angesichts der fehlenden Authentifizierung kann jeder im Internet, der die URL deines Projekts findet, in dieses Projekt pushen. Somit sollte klar sein, dass es sehr selten vorkommt.

### Vorteile

Das Git-Protokoll ist oft als erstes Netzwerkübertragungsprotokoll verfügbar. Wenn du viele Daten für ein öffentliches Projekt bereitstellst oder du ein sehr großes Projekt bedienst, das keine Benutzeroauthentifizierung für den Lesezugriff benötigt, dann kannst du den Git-Daemon einrichten, um das Projekt zu unterstützen. Er verwendet den gleichen Datenübertragungsmechanismus wie das SSH-Protokoll, jedoch ohne den Aufwand der Verschlüsselung und Authentifizierung.

### Nachteile

Aufgrund des Fehlens von TLS oder einer anderer Verschlüsselung kann das Klonen über „git://“ zu der Schwachstelle der Ausführung willkürlichen Codes führen und sollte daher vermieden werden, es sei denn, du weißt genau, was du tust.

- Wenn du `git clone git://example.com/project.git` ausführst, kann ein Angreifer, der z. B. deinen Router kontrolliert, das gerade geklonte Repo modifizieren und bösartigen Code injizieren. Wenn du dann den gerade geklonten Code kompilieren/ausführen möchtest, führst du stattdessen diesen bösartigen Code aus. Das Ausführen von `git clone http://example.com/project.git` sollte aus demselben Grund vermieden werden.
- Das Ausführen von `git clone https://example.com/project.git` hat dieses Problem nicht (es sei denn, der Angreifer kann ein TLS-Zertifikat für example.com bereitstellen). Das Ausführen von `git clone git@example.com:project.git` hat nur dann dieses Problem, wenn du einen falschen ssh-key fingerprint akzeptierst.

Es hat auch keine Authentifizierung, d.h. jeder kann das Repo klonen (obwohl dies oft genau das ist, was du willst). Es ist wahrscheinlich auch das am schwierigsten einzurichtende Protokoll. Ein proprietärer Daemon muss laufen, der eine `xinetd` oder `systemd` Konfiguration oder dergleichen erfordert, was nicht immer einfach ist. Es erfordert auch einen Firewall-Zugang auf Port 9418, der kein Standardport ist, den Unternehmens-Firewalls zulassen. Hinter großen Firmen-Firewalls wird dieser „obskure“ Port häufig blockiert.

# Git auf einem Server einrichten

Nun geht es darum, einen Git-Dienst einzurichten, der diese Protokolle auf deinem eigenen Server ausführt.



Hier zeigen wir dir die Befehle und Schritte, die für die grundlegende, vereinfachte Installation auf einem Linux-basierten Server erforderlich sind. Es ist jedoch auch möglich, diese Dienste auf macOS- oder Windows-Servern auszuführen. Die tatsächliche Einrichtung eines Produktionsservers innerhalb deiner Infrastruktur wird sicherlich Unterschiede in Bezug auf Sicherheitsmaßnahmen oder Betriebssystemwerkzeuge mit sich bringen. Hoffentlich gibt dir dies einen Überblick darüber, worum es geht.

Um einen Git-Server einzurichten, musst du ein bestehendes Repository in ein neues Bare-Repository exportieren – ein Repository, das kein Arbeitsverzeichnis enthält. Das ist im Allgemeinen recht einfach zu realisieren. Um dein Repository zu klonen, um ein neues leeres Repository zu erstellen, führst du den Befehl `clone` mit der Option `--bare` aus. Normalerweise enden Bare-Repository-Verzeichnisnamen mit dem Suffix `.git`, wie hier:

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

Du solltest nun eine Kopie der Git-Verzeichnisdaten in deinem `my_project.git` Verzeichnis haben.

Das ist in etwa das selbe wie:

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

Es gibt ein paar kleine Unterschiede in der Konfigurationsdatei, aber für deinen Zweck ist das fast dasselbe. Es übernimmt das Git-Repository allein, ohne Arbeitsverzeichnis, und erstellt daraus ein eigenes Verzeichnis.

## Das Bare-Repository auf einem Server ablegen

Jetzt, da du eine leere Kopie deines Repositorys hast, musst du es nur noch auf einen Server legen und die gewünschten Protokolle einrichten. Nehmen wir an, du hast einen Server mit der Bezeichnung `git.example.com` eingerichtet, auf dem du SSH-Zugriff hast und du möchtest alle deine Git-Repositorys unter dem Verzeichnis `/srv/git` ablegen. Angenommen, `/srv/git` existiert bereits auf diesem Server, dann kannst du dein neues Repository einrichten, indem du dein leeres Repository kopierst:

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

Ab diesem Zeitpunkt können andere Benutzer, die SSH-basierten Lesezugriff auf das Verzeichnis

/srv/git auf diesem Server haben, dein Repository klonen, indem sie Folgendes ausführen:

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

Wenn sich ein Benutzer über SSH in einen Server einloggt und Schreibrechte auf das Verzeichnis /srv/git/my\_project.git hat, hat er auch automatisch Push-Rechte.

Git fügt automatisch Schreibrechte für Gruppen zu einem Repository hinzu, wenn du den Befehl `git init` mit der Option `--shared` ausführst. Beachte, dass durch die Ausführung dieses Befehls keine Commits, Referenzen usw. im laufenden Prozess zerstört werden.

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

Du siehst, wie einfach es ist, aus einem Git-Repository eine leere Version zu erstellen und diese auf einem Server zu platzieren, auf den du und deine Mitstreiter SSH-Zugriff haben. Jetzt seid ihr der Lage, am gleichen Projekt zusammenzuarbeiten.

Das ist auch schon alles ist, was zu tun ist, um einen brauchbaren Git-Server zu betreiben, auf den mehrere Personen Zugriff haben. Füge einfach SSH-fähige Konten auf einem Server hinzu und lege ein leeres Repository an einen Ort, auf das alle diese Benutzer Lese- und Schreibrechte haben. Ihr seid nun startklar – mehr ist nicht nötig.

In den nächsten Abschnitten erfährst du, wie du dieses Grundkonstrukt zu komplexeren Konfigurationen erweitern kannst. Das beinhaltet, dass man nicht für jeden Benutzer ein Benutzerkonto anlegen muss, öffentlichen Lesezugriff auf Repositorys hinzufügt und Web-UIs einrichtet und vieles mehr. Denke jedoch daran, dass zur Zusammenarbeit mit ein paar Personen bei einem privaten Projekt *nur* ein SSH-Server und ein Bare-Repository benötigt wird.

## Kleine Installationen

Wenn Ihr ein kleines Team seid, die Git nur in einer kleinen Umgebung ausprobieren wollen und nur wenige Entwickler habt, kann es ganz einfach sein. Einer der kompliziertesten Aspekte bei der Einrichtung eines Git-Servers ist die Benutzerverwaltung. Wenn du möchtest, dass einige Repositorys für bestimmte Benutzer schreibgeschützt und für andere lesend und schreibend sind, können Zugriff und Berechtigungen etwas schwieriger zu realisieren sein.

## SSH-Zugang

Wenn du einen Server hast, auf dem alle deine Entwickler bereits SSH-Zugriff haben, ist es in der Regel am einfachsten, dort ein erstes Repository einzurichten. Du musst so gut wie keine zusätzlichen Einstellungen vornehmen, wie wir im letzten Abschnitt beschrieben haben. Wenn du komplexere Zugriffsberechtigungen für deine Repositorys benötigst, kannst du diese mit den normalen Dateisystemberechtigungen des Betriebssystems deines Servers verwalten.

Wenn du deine Repositorys auf einem Server platzieren möchtest, der nicht über Konten für alle

Personen in deinem Team verfügt, denen du Schreibzugriff gewähren möchtest, musst du für sie einen SSH-Zugriff einrichten. Wir gehen davon aus, dass auf deinem Server bereits ein SSH-Server installiert ist und du auf diesen Server zugreifen kannst.

Es gibt einige Möglichkeiten, wie du jedem in deinem Team Zugang gewähren kannst. Die erste besteht darin, Konten für alle einzurichten, was unkompliziert ist, aber recht mühsam sein kann. Unter Umständen ist es ratsam, `adduser` (oder die mögliche Alternative `useradd`) nicht auszuführen und für jeden neuen Benutzer temporäre Passwörter festzulegen.

Eine zweite Möglichkeit besteht darin, ein einzelnes Git-Benutzerkonto auf der Maschine zu erstellen. Fordere anschließend jeden Benutzer, der Schreibrechte haben möchte auf, dir seinen öffentlichen SSH-Schlüssel zu senden. Diesen Schlüssel fügst du nun zur Datei `~/.ssh/authorized_keys` des neuen 'git' Kontos hinzu. Jetzt sollte jeder über das 'git' Konto auf die Maschine zugreifen können. Das hat keinen Einfluss auf die committeten Daten. Der SSH-Benutzer mit dem du dich anmeldest hat keinen Einfluss auf deine aufgezeichneten Commits.

Eine weitere Möglichkeit besteht darin, dass sich dein SSH-Server von einem LDAP-Server oder einer anderen zentralen Authentifizierungsquelle authentifiziert, den du möglicherweise bereits eingerichtet hast. Solange jeder Benutzer Shell-Zugriff auf die Maschine erhalten kann, sollte jeder denkbare SSH-Authentifizierungsmechanismus funktionieren.

## Erstellung eines SSH-Public-Keys

Viele Git-Server authentifizieren sich über öffentliche SSH-Schlüssele. Um einen öffentlichen Schlüssel bereitzustellen, muss jeder Benutzer in seinem System selbst einen generieren, falls er noch keinen hat. Der Ablauf ist für alle Betriebssysteme gleich. Zuerst solltest du überprüfen, ob du noch keinen Schlüssel hast. Standardmäßig werden die SSH-Schlüssele eines Benutzers im Verzeichnis `~/.ssh` dieses Benutzers gespeichert. Du kannst leicht nachsehen, ob du bereits über einen Schlüssel verfügst, indem du in dieses Verzeichnis gehst und den Inhalt auflistest:

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

Suche ein Datei-Paar mit dem Namen `id_dsa` oder `id_rsa` und eine entsprechende Datei mit der Erweiterung `.pub`. Die `.pub` Datei ist dein öffentlicher Schlüssel, und die andere Datei ist der zugehörige private Schlüssel. Wenn du diese Dateien nicht hast (oder nicht einmal ein `.ssh` Verzeichnis vorhanden ist), kannst du sie erstellen, indem du ein Programm namens `ssh-keygen` ausführst, das im SSH-Paket auf Linux/macOS-Systemen enthalten ist und mit Git für Windows installiert wird:

```
$ 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
```

Zuerst wird der Speicherort des Schlüssels (`.ssh/id_rsa`) festgelegt, danach wird zweimal nach einer Passphrase gefragt, die du leer lassen kannst, wenn du beim Verwenden des Schlüssels nicht jedes mal ein Passwort eingeben möchtest. Wenn du jedoch ein Passwort verwendest, fügst du die Option `-o` hinzu; diese speichert den privaten Schlüssel in einem Format, das resistenter gegen Brute-Force-Passwortcracking ist als das Standardformat. Du kannst auch das `ssh-agent` Tool verwenden, um zu vermeiden, dass du das Passwort jedes Mal neu eingeben musst.

Jetzt muss jeder Benutzer seinen öffentlichen Schlüssel an dich oder an einen Administrator des Git-Servers senden (vorausgesetzt, du verwendest ein SSH-Server-Setup, für das öffentliche Schlüssel erforderlich sind). Alles, was man tun muss, ist, den Inhalt der `.pub` Datei zu kopieren und per E-Mail zu versenden. Die öffentlichen Schlüssel sehen in etwa so aus:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAQEAk1OUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPl+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUFrv1QzM7x1ELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMcilq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUF1jQJKprrx88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnPnTPi89ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

Ein ausführliches Tutorial zur Erstellung eines SSH-Schlüssels für unterschiedliche Betriebssysteme findest du in der GitHub-Anleitung für SSH-Schlüssel unter [Generieren und Hinzufügen eines SSH-Schlüssels](#).

## Einrichten des Servers

Lass uns nun durch die Einrichtung des SSH-Zugriffs auf der Serverseite gehen. In diesem Beispiel verwendest du die Methode `authorized_keys` zur Authentifizierung deiner Benutzer. Wir nehmen an, dass du eine Standard-Linux-Distribution wie Ubuntu verwendest.



Viele der hier beschriebenen Vorgänge können mit dem Befehl `ssh-copy-id` automatisiert werden, ohne dass öffentliche Schlüssel manuell kopiert und installiert werden müssen.

Zuerst erstellst du ein `git` Benutzerkonto und ein `.ssh` Verzeichnis für diesen Benutzer:

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh && chmod 700 .ssh
```

```
$ touch .ssh/authorized_keys && chmod 600 .ssh/authorized_keys
```

Als nächstes musst du einige öffentliche SSH-Schlüssel für Entwickler zur `authorized_keys` Datei für den `git` User hinzufügen. Nehmen wir an, du hast einige vertrauenswürdige öffentliche Schlüssel und hast sie in temporären Dateien gespeichert. Auch hier sehen die öffentlichen Schlüssel in etwa so aus:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x41hJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAzuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdf8AcCIicTDWbqLAcU4UpkaX8KyGllwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
07TCUSBdLQlgMVOFq1I2uPWQOkOWQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgTZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Du fügst sie einfach an die Datei `authorized_keys` des `git` Benutzers in dessen `.ssh` Verzeichnis an:

```
$ 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
```

Nun kannst du ein leeres Repository für sie einrichten, indem du `git init` mit der Option `--bare` ausführst, die das Repository ohne Arbeitsverzeichnis initialisiert:

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

Dann können John, Josie oder Jessica die erste Version ihres Projekts in dieses Repository pushen, indem sie es als Remote hinzufügen und dann einen Branch pushen. Beachte, dass jemand auf der Maschine eine Shell ausführen muss und jedes Mal, wenn du ein Projekt hinzufügen möchtest, ein Bare-Repository erstellen muss. Lass uns `gitserver` als Hostname für den Server verwenden, auf dem du deinen `git` Benutzer und dein Repository eingerichtet hast. Wenn du den Server in einem internen Netzwerk betreibst und DNS so einrichtest, dass `gitserver` auf diesen Server zeigt, dann kannst du folgende Befehle verwenden (vorausgesetzt, dass `myproject` ein bestehendes Git-Projekt ist):

```
# on John's computer
$ cd myproject
$ git init
$ git add .
$ git commit -m 'Initial commit'
$ git remote add origin git@gitserver:/srv/git/project.git
```

```
$ git push origin master
```

Jetzt können die anderen es klonen und Änderungen genauso einfach wieder pushen:

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

Mit dieser Methode kannst du kurzfristig einen Read/Write Git-Server für eine Handvoll Entwickler in Betrieb nehmen.

Du solltest beachten, dass sich derzeit alle diese Benutzer auch am Server anmelden und eine Shell als **git** Benutzer erhalten können. Wenn du das einschränken willst, musst du die Shell zu etwas anderem in der Datei **/etc/passwd** ändern.

Du kannst das **git** Benutzerkonto mit einem in Git enthaltenen Shell-Tool mit dem Namen **git-shell** ganz einfach auf Git-bezogene Aktivitäten beschränken. Wenn du diese Option als Anmeldeshell des **git** Benutzerkontos festlegst, hat dieses Konto keinen normalen Shell-Zugriff auf deinen Server. Um das einzurichten, gib **git-shell** anstelle von **bash** oder **csh** für die Login-Shell dieses Kontos an. Um das zu erreichen, musst du zuerst den vollständigen Pfadnamen des **git-shell** Befehls zu **/etc/shells** hinzufügen, falls er nicht bereits vorhanden ist:

```
$ cat /etc/shells  # prüfe ob git-shell bereits vorhanden ist. Wenn nicht...
$ which git-shell  # prüfe, ob git-shell auf deinem System installiert ist.
$ sudo -e /etc/shells  # füge den pfad zu git-shell aus deinem letzten Kommando hinzu
```

Jetzt kannst du die Shell für einen Benutzer mit **chsh <username> -s <shell>** bearbeiten:

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

Nun kann der **git** Benutzer die SSH-Verbindung weiterhin zum Pushen und Pullen von Git-Repositorys verwenden, aber sich sonst nicht mehr auf der Maschine bewegen. Wenn du es versuchst, siehst du eine entsprechende Meldung:

```
$ 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.
```

An dieser Stelle könnten Benutzer noch SSH-Portforwarding verwenden, um auf jeden Host zuzugreifen, den der Git-Server erreichen kann. Wenn du das verhindern möchtest, kannst du die Datei **authorized\_keys** bearbeiten und jedem Schlüssel, den du einschränken möchtest, die folgenden Optionen voranstellen:

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

Das Ergebnis sollte so aussehen:

```
$ cat ~/.ssh/authorized_keys
no-port-forwarding,no-X11-forwarding,no-agent-forwarding,no-pty ssh-rsa
AAAAB3NzaC1yc2EAAAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4LojG6rs6h
PB09j9R/T17/x4l1hJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N
YsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdP6W1GYEIgS9EzSdf8AcC
IicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv07TCUSBd
LQlgMVOFq1I2uPWQ0kOWQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgtzg2AYYgPqdAv8JggJ
ICUvax2T9va5 gsg-keypair
```

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

Jetzt funktionieren die Git-Netzwerkbefehle weiterhin einwandfrei, aber die Benutzer können keine Shell nutzen. Wie oben zu sehen, kannst du auch ein Verzeichnis im Home-Verzeichnis des `git` Benutzers einrichten, das den `git-shell` Befehl ein wenig anpasst. Du kannst beispielsweise die vom Server akzeptierten Git-Befehle einschränken oder die Nachricht anpassen, die Benutzer sehen, wenn sie versuchen, SSH auf diese Weise auszuführen. Führe `git help shell` aus, um weitere Informationen zum Anpassen der Shell zu erhalten.

## Git-Daemon

Als Nächstes richten wir einen Daemon ein, der Repositorys mit dem „Git“-Protokoll versorgt. Das ist eine gängige Option für den schnellen, nicht authentifizierten Zugriff auf deine Git-Daten. Denke daran, dass alles, was du über dieses Protokoll bereitstellst, innerhalb deines Netzwerks öffentlich ist, da dies kein authentifizierter Dienst ist.

Wenn du dieses auf einem Server außerhalb deiner Firewall ausführst, sollte dies nur für Projekte verwendet werden, die für die Welt öffentlich sichtbar sein dürfen. Wenn sich der genutzte Server, hinter deiner Firewall befindet, kannst du es für Projekte verwenden, auf die eine große Anzahl von Personen oder Computern (Continuous Integration oder Build-Server) nur Lesezugriff haben, wenn du nicht für jeden einen SSH-Schlüssel hinzufügen möchtest.

In jedem Fall ist das Git-Protokoll relativ einfach einzurichten. Grundsätzlich solltest du diesen Befehl als Daemon auszuführen:

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

Mit der `--reuseaddr` Option kann der Server neu gestartet werden, ohne dass man auf eine Time-Out für alte Verbindungen warten muß. Mit der `--base-path` Option können Benutzer Projekte klonen, ohne den gesamten Pfad anzugeben. Der abschließende Pfad teilt dem Git-Daemon mit, wo nach zu exportierenden Repositorys gesucht werden soll. Wenn du eine Firewall verwendest, musst du den Port 9418 konfigurieren, um eingehende Verbindungen zuzulassen.

Du kannst diesen Prozess auf verschiedene Arten als Daemon betreiben, je nachdem, welches Betriebssystem du verwendest.

Das `systemd` ist das gebräuchlichste Init-System unter modernen Linux-Distributionen. Das könntest du für diesen Zweck verwenden. Lege einfach eine Datei mit folgendem Inhalt in `/etc/systemd/system/git-daemon.service` ab:

```
[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
```

Du hast vielleicht bemerkt, dass der Git-Daemon hier mit `git` als Gruppe und Benutzer gestartet wird. Passe es an deine Bedürfnisse an und stelle sicher, dass der angegebene Benutzer auf dem System vorhanden ist. Überprüfe auch, ob sich die Git-Binärdatei tatsächlich unter `/usr/bin/git` befindet und ändere gegebenenfalls den Pfad.

Abschließend führst du `systemctl enable git-daemon` aus, um den Dienst beim Booten automatisch zu starten, so dass du den Dienst mit `systemctl start git-daemon` und `systemctl stop git-daemon` starten und stoppen kannst.

Auf anderen Systemen kannst du `xinetd` verwenden oder ein Skript in deinem `sysvinit` System benutzen. Auch andere Lösungen sind möglich, solange der Prozess als Daemon läuft und irgendwie überwacht wird.

Als nächstes musst du Git mitteilen, auf welche Repositorys nicht authentifizierter, serverbasierter Zugriffe auf Git möglich sein soll. Du kannst das in den einzelnen Repositorys tun, indem du eine Datei mit dem Namen `git-daemon-export-ok` erstellst.

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

Das Vorhandensein dieser Datei teilt Git mit, dass es in Ordnung ist, dieses Projekt ohne Authentifizierung zur Verfügung zu stellen.

# Smart HTTP

Wir haben jetzt authentifizierte Zugriffe über SSH und nicht authentifizierte Zugriffe über `git://`, aber es gibt auch ein Protokoll, das beides gleichzeitig kann. Die Einrichtung von Smart HTTP ist im Grunde genommen nur die Aktivierung eines CGI-Skripts, das zusammen mit Git namens `git-http-backend` auf dem Server bereitgestellt wird. Dieses CGI liest den Pfad und die Header, die von einem `git fetch` oder `git push` an eine HTTP-URL gesendet werden, und bestimmt, ob der Client über HTTP kommunizieren kann (was für jeden Client ab Version 1.6.6 gilt). Wenn das CGI sieht, dass der Client intelligent ist, kommuniziert es intelligent mit ihm; andernfalls fällt es auf das dumme Verhalten zurück (also ist es rückwärtskompatibel für Lesezugriffe mit älteren Clients).

Lass uns durch ein sehr einfaches Setup gehen. Wir werden Apache als CGI-Server verwenden. Wenn du kein Apache-Setup hast, kannst du dies auf einem Linux-System, wie nachfolgend beschrieben einrichten:

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

Dadurch werden auch die Module `mod_cgi`, `mod_alias`, und `mod_env` aktiviert, die alle benötigt werden, damit das Ganze ordnungsgemäß funktioniert.

Du solltest auch die Unix-Benutzergruppe im Verzeichnis `/srv/git` auf `www-data` setzen, damit dein Webserver auf die Repositorys lesend und schreibend zugreifen kann. Die Apache-Instanz, auf der das CGI-Skript läuft, wird standardmäßig unter dieser Benutzer laufen:

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

Als nächstes müssen wir der Apache-Konfiguration einige Dinge hinzufügen, um das `git-http-backend` als Handler für alles, was im `git` Pfad deiner Webservers liegt, auszuführen.

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

Wenn du die Umgebungsvariable `GIT_HTTP_EXPORT_ALL` nicht setzt, wird Git unauthentifizierte Clients nur die Repositorys, die die Datei `git-daemon-export-ok` enthalten, zur Verfügung stellen. Das Verhalten ist dann wie beim Git-Daemon.

Abschließend konfigurierst du Apache so, dass er Anfragen an das `git-http-backend` zulassen soll, um Schreibvorgänge zu authentifizieren. Dazu kannst du folgenden Code nutzen:

```
<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>
```

Dazu musst du eine **.htpasswd** Datei erstellen, die die Passwörter aller gültigen Benutzer enthält. Hier ein Beispiel für das Hinzufügen eines Benutzers „schacon“:

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

Es gibt unzählige Möglichkeiten, Benutzer mit Apache zu authentifizieren. Du musst eine von ihnen auswählen und implementieren. Dies ist ein einfaches Beispiel. Du wirst dies wahrscheinlich über SSL konfigurieren wollen, damit auch alle Daten verschlüsselt werden.

Wir wollen nicht zu weit in das Konzept der Apache-Konfigurationsspezifikationen eindringen, da du möglicherweise einen anderen Server verwendst oder andere Authentifizierungsanforderungen hast. Die Idee ist, dass Git mit einem CGI mit dem Namen **git-http-backend** daherkommt, das beim Senden und Empfangen von Daten die Kommunikation über HTTP aushandelt. Es implementiert selbst keine Authentifizierung, aber diese kann über den Webservers gesteuert werden. Du kannst das mit fast jedem CGI-fähigen Webserver tun. Am besten wählst du denjenigen, den du am besten kennst.



Weitere Informationen zur Konfiguration der Authentifizierung in Apache findest du in den Apache-Dokumenten unter: <https://httpd.apache.org/docs/current/howto/auth.html>

## GitWeb

Nun, da du über einen einfachen Lese-/Schreibzugriff und Lesezugriff auf dein Projekt verfügst, kannst eine einfache webbasierten Git-GUI einrichten. Git wird mit einem CGI-Skript namens GitWeb ausgeliefert, das dafür verwendet werden kann.

The screenshot shows the GitWeb interface for a project. At the top, there's a navigation bar with links like 'summary', 'shortlog', 'log', 'commit', 'commitdiff', and 'tree'. On the right, there are buttons for 'commit' (with dropdown), 'search' (with input field and 're' checkbox), and a 'git' logo. Below the navigation, there's a section for 'description' (empty), 'owner' (Ben Straub), and 'last change' (Wed, 11 Jun 2014 12:20:23 -0700). The main area is divided into sections: 'shortlog' (listing commits from June 2014), 'tags' (listing project versions from v0.11.0 to v0.21.0-rc1), and a footer with a '... more' link.

Figure 49. Die webbasierte Benutzeroberfläche von GitWeb

Du kannst sehr einfach ausprobieren, wie GitWeb für dein Projekt aussehen würde. Es gibt einen Befehl, um eine temporäre Instanz zu starten. Dazu benötigst du auf deinem System einen simplen Webserver wie `lighttpd` oder `webrick`. Auf Linux-Maschinen wird `lighttpd` oft installiert, so dass du ihn wahrscheinlich direkt starten kannst, indem du `git instaweb` in deinem Projektverzeichnis eingibst. Wenn du einen Mac verwenden, wird Leopard mit Ruby vorinstalliert geliefert, so dass `webrick` dein Favorit sein sollte. Um `instaweb` mit einem nicht-lighttpd Handler zu starten, kannst du es mit der Option `--httpd` versuchen.

```
$ 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]
```

Dies startet einen HTTPD-Server auf Port 1234 und öffnet ein Webbroweser, der die Seite anzeigt. Das Vorgehen ist ziemlich einfach. Wenn du fertig bist und den Server herunterfahren möchtest, kannst du den gleichen Befehl mit der Option `--stop` ausführen:

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

Wenn du das Web-Interface kontinuierlich auf einem Server für dein Team oder für ein gehostetes Open-Source-Projekt ausführen möchtest, musst du das CGI-Skript so einrichten, dass es von deinem Webserver zur Verfügung gestellt wird. Einige Linux-Distributionen haben ein `gitweb` Paket, das du möglicherweise über `apt` oder `dnf` installieren kannst, so dass du das zuerst

ausprobieren solltest. Wir werden die manuelle Installation von GitWeb nur sehr kurz abhandeln. Zuerst musst du den Git-Quellcode, der im Lieferumfang von GitWeb enthalten ist, herunterladen und das benutzerdefinierte CGI-Skript generieren:

```
$ git clone https://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/
```

Beachte, dass du dem Befehl mit der Variablen `GITWEB_PROJECTROOT` mitteilen musst, wo dein Git-Repository zu finden ist. Nun musst du Apache dazu bringen, CGI für dieses Skript zu verwenden. Dazu kannst du einen VirtualHost hinzufügen:

```
<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>
```

Auch hier kann GitWeb mit jedem CGI- oder Perl-fähigen Webserver zur Verfügung gestellt werden. Wenn du etwas anderes bevorzugst, sollte es nicht schwierig einzurichten sein. Nun solltest du in der Lage sein, <http://gitserver/> zu besuchen, um deine Repositorien online zu betrachten.

## GitLab

GitWeb ist jedoch ein recht übersichtliches Tool. Wenn du einen modernen, voll ausgestatteten Git-Server suchst, gibt es einige Open-Source-Lösungen, die du stattdessen installieren kannst. Da GitLab einer der beliebtesten ist, werden wir uns mit der Installation im Detail befassen und es als Beispiel verwenden. Dies ist etwas schwieriger als die GitWeb-Option und erfordert mehr Wartung, aber es ist eine viel umfassendere Lösung.

## Installation

GitLab ist eine datenbankgestützte Webanwendung, so dass die Installation etwas aufwändiger ist als bei einigen anderen Git-Servern. Glücklicherweise ist dieser Prozess sehr gut dokumentiert und

unterstützt. GitLab empfiehlt dringend, GitLab über das offizielle Omnibus GitLab-Paket zu installieren.

Die anderen Installationsmethoden sind:

- GitLab Helm-Chart zur Verwendung mit Kubernetes.
- Dockerisierte GitLab-Pakete zur Verwendung mit Docker.
- Direkt aus den Quelldateien.
- Cloud-Anbieter wie AWS, Google Cloud Platform, Azure, OpenShift und Digital Ocean.

Weitere Informationen findest du in der Readme-Datei [GitLab Community Edition \(CE\)](#).

## Administration

Die Verwaltungsoberfläche von GitLab wird ist webbasiert. Benutze einfach deinen Browser, um den Hostnamen oder die IP-Adresse, auf der GitLab installiert ist, anzugeben, und melden dich als Admin-Benutzer an. Der Standardbenutzername ist `admin@local.host`, das Standardpasswort ist `5iveL!fe` (dies muss nach der Eingabe **geändert werden**). Klicke nach der Anmeldung im Menü oben rechts auf das Symbol „Admin-Bereich“.

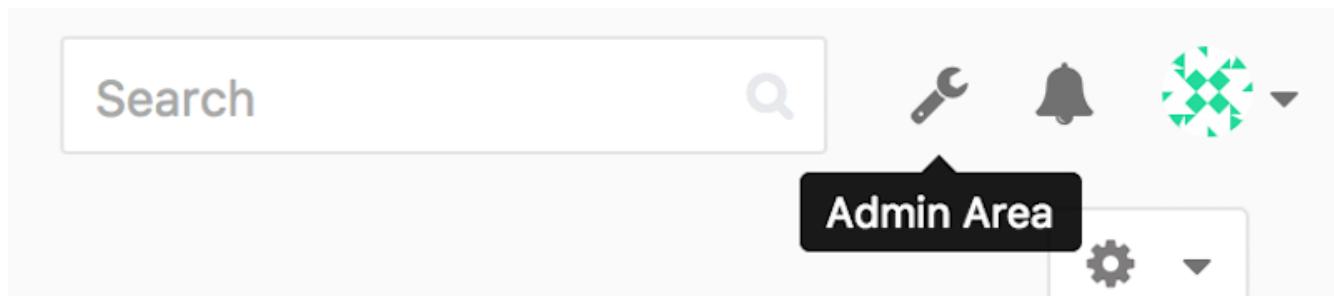


Figure 50. Der „Admin-Bereich“ im GitLab-Menü

## Benutzer

Jeder Gitlab Nutzer muß ein Benutzerkonto besitzen. Benutzerkonten sind recht simple. Hauptsächlich beinhalten sie persönliche Informationen, die an Login-Daten geknüpft sind. Jedes Benutzerkonto hat einen **namespace** (Namensraum), der eine logische Gruppierung von Projekten ist, die diesem Benutzer gehören. Wenn der Benutzer Jane ein Projekt mit dem Namen project hätte, wäre die URL dieses Projekts <http://server/jane/project>.

The screenshot shows the 'Users' section of the GitLab Admin Area. At the top, there are navigation 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 selected), Groups, Builds, and Runners. A search bar and a 'New User' button are also present. The main content area displays a list of users with their names, email addresses, roles (e.g., Admin, User), and status (e.g., Active, Blocked). Each user entry includes an 'Edit' button and a dropdown menu icon.

Name	Email	Role	Status
Administrator	admin@example.com	Admin	Active
Betsy Rutherford II	marlin@lednerlangworth.biz	User	Active
Brenden Hayes	laney_dubuque@cormier.biz	User	Active
Cassandra Kilback	caterina@beer.com	User	Active
Cathryn Leffler DVM	desmond@crooks.ca	User	Active
Cecil Medhurst	winnifred@glover.co.uk	User	Active
Dr. Joany Fisher	milan@hueels.us	User	Active
Jazmin Sipes	juliet.turner@leannon.co.uk	User	Active

Figure 51. Das Fenster der Benutzerverwaltung von GitLab

Das Löschen eines Benutzers kann auf zwei Arten erfolgen. Das „Blockieren“ eines Benutzers verhindert, dass er sich am GitLab anmeldet. Alle Daten unter den Namensraum dieses Benutzers bleiben jedoch erhalten. Mit der E-Mail-Adresse dieses Benutzers signierte Commits werden weiterhin mit seinem Profil verknüpft.

Das „Zerstören“ eines Benutzers hingegen entfernt ihn vollständig aus der Datenbank und dem Dateisystem. Alle Projekte und Daten in seinem Namensraum werden entfernt, und alle Gruppen, die sich in seinem Besitz befinden, werden ebenfalls entfernt. Das ist eine permanente und destruktive Aktion, die in der Regel selten angewendet wird.

## Gruppen

Eine GitLab-Gruppe ist eine Sammlung von Projekten. Zusätzlich beinhaltet sie Daten, wie Benutzer auf diese Projekte zugreifen können. Jede Gruppe hat einen Projektnamensraum (genauso wie Benutzer). Wenn die Gruppe `training` ein Projekt `materials` hat, lautet die URL <http://server/training/materials>.

The screenshot shows the GitLab.org group administration interface. At the top, there's a navigation bar with 'Group' selected, followed by 'Activity', 'Labels', 'Milestones', 'Issues 8,501', 'Merge Requests 701', 'Members', and 'Contribution Analytics'. A search bar says 'This group Search'. Below the navigation is the group logo (@gitlab-org) and the tagline 'Open source software to collaborate on code'. There are buttons for 'Leave group' and 'Global'. The main area shows a list of projects under 'All Projects':

- GitLab Development Kit** (owner): Get started with GitLab Rails development.
- kubernetes-gitlab-demo** (K): Idea to Production GitLab Demo running on Kubernetes.
- omnibus-gitlab**: This project creates full-stack platform-specific downloadable packages for GitLab.
- GitLab Enterprise Edition**: GitLab Enterprise Edition.
- gitlab-shell**: SSH access and repository management app for GitLab.
- gitlab-ci-multi-runner** (runner): GitLab Runner.

Filters at the top right allow 'Filter by name' and 'Last updated' sorting.

Figure 52. Der Admin-Bildschirm für die Gruppenverwaltung von GitLab

Jeder Gruppe ist einer Reihe von Benutzern zugeordnet, von denen jeder eine Berechtigungsstufe für die Projekte der Gruppe und der Gruppe selbst hat. Diese reichen von „Guest“ (nur Themen und Chat) bis hin zu „Owner“ (volle Kontrolle über die Gruppe, ihre Mitglieder und ihre Projekte). Die Arten von Berechtigungen sind zu zahlreich, um sie hier aufzulisten, aber GitLab hat einen hilfreichen Link auf dem Administrationsbildschirm.

## Projekte

Ein GitLab-Projekt entspricht in etwa einem einzelnen Git-Repository. Jedes Projekt gehört zu einem einzigen Namensraum, entweder einem Benutzer oder einer Gruppe. Wenn das Projekt einem Benutzer gehört, hat dieser Projektbesitzer die direkte Kontrolle darüber, wer Zugriff auf das Projekt hat. Falls das Projekt einer Gruppe gehört, werden auch die Berechtigungen der Gruppe auf Benutzerebene wirksam.

Jedes Projekt hat eine Zugriffsebene, die steuert, wer Lesezugriff auf die Seiten und das Repository des Projekts hat. Wenn ein Projekt *privat* ist, muss der Eigentümer des Projekts Benutzern explizit Zugriff gewähren. Ein *internes* Projekt ist für jeden angemeldeten Benutzer sichtbar. Ein *öffentliches* (engl. *public*) Projekt ist für jeden sichtbar. Beachte, dass dies sowohl den Zugriff auf **git fetch** als auch den Zugriff auf die Web-Benutzeroberfläche für dieses Projekt steuert.

## Hooks

GitLab bietet Unterstützung für Hooks, sowohl auf Projekt- als auch auf Systemebene. Für beides führt der GitLab-Server einen HTTP POST mit einem beschreibenden JSON durch, wenn relevante Ereignisse eintreten. Auf diese Weise kannst du deine Git-Repositories und GitLab-Instanzen mit dem Rest deiner Entwicklungsplattform verbinden, wie z.B. CI-Server, Chatrooms oder Deploymenttools.

## Grundlegende Anwendung

Das erste, was du mit GitLab machen solltest, ist das Erstellen eines neuen Projekts. Dies geschieht durch Anklicken des Symbols „+“ in der Symbolleiste. Du wirst nach dem Namen des Projekts gefragt, zu welchem Namensraum es gehören soll und wie seine Sichtbarkeit sein soll. Das meiste, was du hier angibst, ist nicht permanent und kann später über die Konfigurations-Oberfläche angepasst werden. Klicke auf „Projekt erstellen“, und du bist fertig.

Sobald das Projekt existiert, wirst du es vermutlich mit einem lokalen Git-Repository verbinden wollen. Jedes Projekt ist über HTTPS oder SSH zugänglich. Beide können genutzt werden, um ein Git-Remote zu konfigurieren. Die URLs sind oben auf der Startseite des Projekts sichtbar. Für ein bestehendes lokales Repository erstellt dieser Befehl einen Remote mit Namen `gitlab` für die gehostete Instanz:

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

Wenn du noch keine lokale Kopie des Repositorys hast, kannst du das ganz einfach nachholen:

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

Die Web-Benutzeroberfläche bietet Zugriff auf viele nützliche Informationen des Repositorys. Die Homepage jedes Projekts zeigt die letzten Aktivitäten an. Die Links oben zeigen verschiedene Ansichten der Projektdateien und zum Commit-Log.

## Zusammen arbeiten

Die einfachste Art der Zusammenarbeit bei einem GitLab-Projekt besteht darin, jedem Benutzer direkten Push-Zugriff auf das Git-Repository zu ermöglichen. Du kannst einem Benutzer zu einem Projekt hinzufügen, indem du im Abschnitt „Mitglieder“ der Einstellungen dieses Projekts den neuen Benutzer einer Zugriffsebene zuordnest. Die verschiedenen Zugriffsebenen werden in den [Gruppen](#) erläutert. Indem ein Benutzer die Zugriffsebene „Developer“ oder höher erhält, kann dieser Benutzer Commits und Branches direkt und ohne Einschränkung in das Repository pushen.

Eine weitere, stärker entkoppelte Art der Zusammenarbeit ist die Nutzung von Merge-Requests. Diese Funktion ermöglicht es jedem Benutzer, der ein Projekt sehen kann, kontrollierter dazu beizutragen. Benutzer mit direktem Zugriff können einfach einen Branch erstellen, auf ihn committen und einen Merge-Request von ihrem Branch zurück in den `master` oder einen anderen Branch einreichen. Benutzer, die keine Push-Berechtigungen für ein Repository haben, können es „forken“ (ihre eigene Kopie erstellen), Push-Commits für diese Kopie erstellen und einen Merge-Request von ihrer Fork zurück zum Hauptprojekt einreichen. Dieses Modell ermöglicht es dem Eigentümer, die volle Kontrolle darüber zu behalten, was wann in das Repository gelangt, und gleichzeitig Beiträge von unbekannten Benutzern zu ermöglichen.

Merge-Requests und Issues sind die Hauptelemente von langlebigen Diskussionen in GitLab. Jede Merge-Request ermöglicht eine zeilenweise Diskussion der vorgeschlagenen Änderung. Damit kann eine einfache Code-Überprüfung (engl. Code-Review) sowie ein Diskussionsstrang umgesetzt werden. Beide können Benutzern zugeordnet oder in Meilensteine organisiert werden.

Dieser Abschnitt konzentriert sich hauptsächlich auf die Git-bezogenen Funktionen von GitLab. Als ausgereiftes Programm bietet es viele weitere Funktionen, die dir bei der Teamarbeit helfen, wie Projekt-Wikis und System-Wartungstools. Ein Vorteil von GitLab ist, dass du nach der Einrichtung und Inbetriebnahme des Servers selten eine Konfigurationsdatei anpassen oder über SSH auf den Server zugreifen musst. Die überwiegende Verwaltung und allgemeine Nutzung kann über die Browser-Oberfläche erfolgen.

## Von Drittanbietern gehostete Optionen

Wenn du nicht alle Arbeiten zur Einrichtung eines eigenen Git-Servers durchführen möchtest, hast du mehrere Möglichkeiten, deine Git-Projekte auf einer externen dedizierten Hosting-Seite zu hosten. Dies bietet eine Reihe von Vorteilen: Eine Hosting-Site ist in der Regel schnell eingerichtet und in der Lage, Projekte einfach zu starten, ohne dass eine Serverwartung oder -überwachung erforderlich ist. Selbst wenn du deinen eigenen Server intern einrichtest und betreibst, kannst du dennoch eine öffentliche Hosting-Site für deine Open-Source-Code verwenden. Es ist im Allgemeinen einfacher für die Open-Source-Community, dich so zu finden und dich zu unterstützen.

In der heutigen Zeit hast du eine große Anzahl von Hosting-Optionen zur Auswahl, jede mit unterschiedlichen Vor- und Nachteilen. Um eine aktuelle Liste zu sehen, besuche die GitHosting-Seite im Hauptwiki von Git unter <https://git.wiki.kernel.org/index.php/GitHosting>.

Wir werden die Verwendung von GitHub in Kapitel 6 [GitHub](#) im Detail besprechen. GitHub ist der größte Git-Host auf dem Markt ist und du wirst vermutlich mit Projekten darauf arbeiten müssen. Es gibt jedoch noch Dutzende weitere, aus denen du wählen kannst, falls du nicht deinen eigenen Git-Server einrichten willst.

## Zusammenfassung

Du hast mehrere Möglichkeiten, ein entferntes Git-Repository in Betrieb zu nehmen, damit du mit anderen zusammenarbeiten und deine Arbeit teilen kannst.

Der Betrieb eines eigenen Servers gibt dir viel Kontrolle und ermöglicht es dir, den Server innerhalb deiner eigenen Firewall zu betreiben. Ein solcher Server benötigt jedoch in der Regel einen angemessenen Teil deiner Zeit für Einrichtung und Wartung. Wenn du deine Daten auf einem gehosteten Server ablegst, macht es dir dein Leben wesentlich einfacher. Du musst aber die Möglichkeit haben, deinen Code auf fremden Servern zu speichern. Einige Unternehmen erlauben das nicht.

Es sollte ziemlich einfach sein festzustellen, welche Lösung oder Kombination von Lösungen für dich und dein Unternehmen am besten geeignet ist.

# Verteiltes Git

Nachdem du ein entferntes Git-Repository eingerichtet hast, in dem alle Entwickler ihren Code teilen können, und du mit den grundlegenden Git-Befehlen in einem lokalen Arbeitsablauf vertraut bist, wirst du einige der verteilten Arbeitsabläufe verwenden, die Git dir ermöglicht.

In diesem Kapitel erfährst du, wie du mit Git in einer verteilten Umgebung als Mitwirkender (engl. Contributor) und Integrator arbeitest. Das heißtt, du lernst, wie du Quelltext erfolgreich zu einem Projekt beisteuern und es dir und dem Projektbetreuer so einfach wie möglich machst. Außerdem lernst du, wie du ein Projekt erfolgreich verwaltet, in dem mehrere Entwicklern Inhalte beisteuern.

## Verteilter Arbeitsablauf

Im Gegensatz zu CVCSSs (Centralized Version Control Systems – Zentrale Versionsverwaltungs Systeme) kannst du dank der verteilten Struktur von Git die Zusammenarbeit von Entwicklern in Projekten wesentlich flexibler gestalten. In zentralisierten Systemen ist jeder Entwickler ein gleichwertiger Netzknoten, der mehr oder weniger gleichermaßen mit einem zentralen System arbeitet. In Git ist jedoch jeder Entwickler potentiell beides – sowohl Netzknoten als auch zentrales System. Das heißtt, jeder Entwickler kann sowohl Code für andere Repositorys bereitstellen als auch ein öffentliches Repository verwalten, auf dem andere ihre Arbeit aufbauen und zu dem sie beitragen können. Dies bietet eine Fülle von möglichen Arbeitsabläufen (engl. Workflows) für dein Projekt und/oder dein Team, sodass wir einige gängige Paradigmen behandeln, welche die Vorteile dieser Flexibilität nutzen. Wir werden auf die Stärken und möglichen Schwächen der einzelnen Entwürfe eingehen. Du kannst einen einzelnen davon auswählen, um ihn zu nutzen, oder du kannst die Funktionalitäten von allen miteinander kombinieren.

### Zentralisierter Arbeitsablauf

In zentralisierten Systemen gibt es im Allgemeinen ein einziges Modell für die Zusammenarbeit – den zentralisierten Arbeitsablauf. Ein zentraler Hub oder *Repository* kann Quelltext akzeptieren und alle Beteiligten synchronisieren ihre Arbeit damit. Eine Reihe von Entwicklern sind Netzknoten – Nutzer dieses Hubs – und synchronisieren ihre Arbeit mit diesem einen, zentralen Punkt.

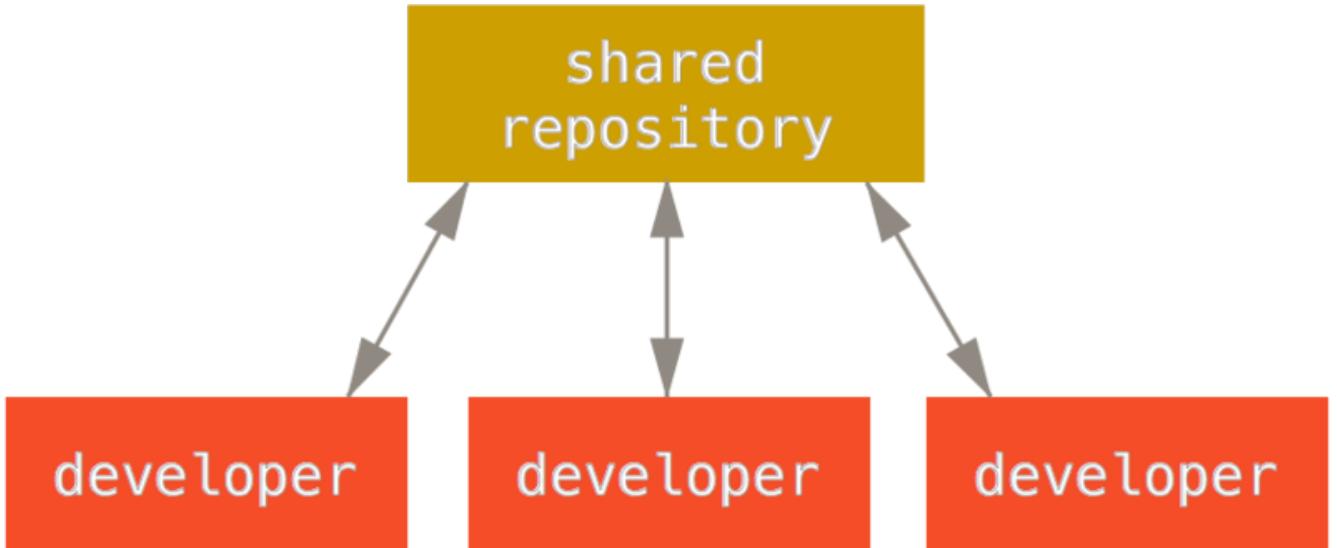


Figure 53. Zentralisierter Arbeitsablauf

Dies bedeutet, wenn zwei Entwickler ein Repository vom Hub klonen und beide Änderungen vornehmen, kann der erste Entwickler seine Änderungen problemlos zurückspielen (pushen). Der zweite Entwickler muss jedoch die Arbeit des ersten Entwicklers bei sich einfließen lassen (mergen), bevor seine Änderungen aufgenommen werden können, damit die Änderungen des ersten Entwicklers nicht überschrieben werden. Dieses Konzept ist in Git genauso wahr wie in Subversion (oder ein anderes beliebiges CVCS), und dieses Konzept funktioniert in Git wunderbar.

Wenn du bereits mit einem zentralisierten Arbeitsablauf in deinem Unternehmen oder Team vertraut bist, kannst du diesen Ablauf problemlos mit Git weiterverwenden. Richte einfach ein einziges Repository ein und gewähre allen Mitgliedern deines Teams Schreib-Zugriff (push). Git lässt nicht zu, dass Benutzer ihre Änderungen gegenseitig überschreiben.

Sagen wir, John und Jessica fangen beide zur gleichen Zeit mit ihrer Arbeit an. John beendet seine Änderung und lädt diese zum Server hoch. Dann versucht Jessica, ihre Änderungen hochzuladen, aber der Server lehnt sie ab. Ihr wird gesagt, dass sie versucht, Änderungen „non-fast-forward“ zu pushen, und dass sie dies erst tun kann, wenn sie die bestehenden Änderungen abgeholt und mit ihrer lokalen Kopie zusammengeführt hat. Dieser Workflow ist für viele Menschen sehr ansprechend, weil er ein bewährtes Modell ist, mit dem viele bereits bekannt und vertraut sind.

Diese Vorgehensweise ist nicht auf kleine Teams beschränkt. Mit dem Verzweigungs-Modell (Branching-Modell) von Git ist es Hunderten von Entwicklern möglich, ein einzelnes Projekt über Dutzende von Branches gleichzeitig erfolgreich zu bearbeiten.

## Arbeitsablauf mit Integrationsmanager

Da du in Git über mehrere Remote-Repositorys verfügen kannst, ist ein Workflow möglich, bei dem jeder Entwickler Schreibzugriff auf sein eigenes, öffentliches Repository und Lesezugriff auf die Repositorys aller anderen Entwickler hat. Dieses Szenario enthält häufig ein zentrales Repository, das das „offizielle“ Projekt darstellt. Um zu diesem Projekt beizutragen, erstellst du deinen eigenen öffentlichen Klon des Projekts und lädst deine Änderungen dort hoch. Anschließend kannst du eine Anfrage an den Betreuer des Hauptprojekts senden, um deine Änderungen zu übernehmen (Pull Request). Der Betreuer kann dann dein Repository als Remote hinzufügen, deine Änderungen lokal testen, diese in seinem Branch einfließen lassen und in sein öffentliches Repository hochladen. Der

Prozess funktioniert wie folgt (siehe [Arbeitsablauf mit Integrationsmanager](#)):

1. Die Projekt Betreuer laden Arbeit in ihr eigenes, öffentlichen Repository hoch.
2. Ein Mitwirkender klonst dieses Repository und nimmt Änderungen vor.
3. Der Mitwirkende lädt diese in sein eigenes öffentliches Repository hoch.
4. Der Mitwirkende sendet dem Betreuer eine E-Mail mit der Aufforderung, die Änderungen zu übernehmen (Pull Request).
5. Der Betreuer fügt das Repository des Mitwirkenden als Remote hinzu und führt die Änderungen lokal zusammen.
6. Der Betreuer lädt die zusammengeführten Änderungen in das Haupt-Repository hoch.

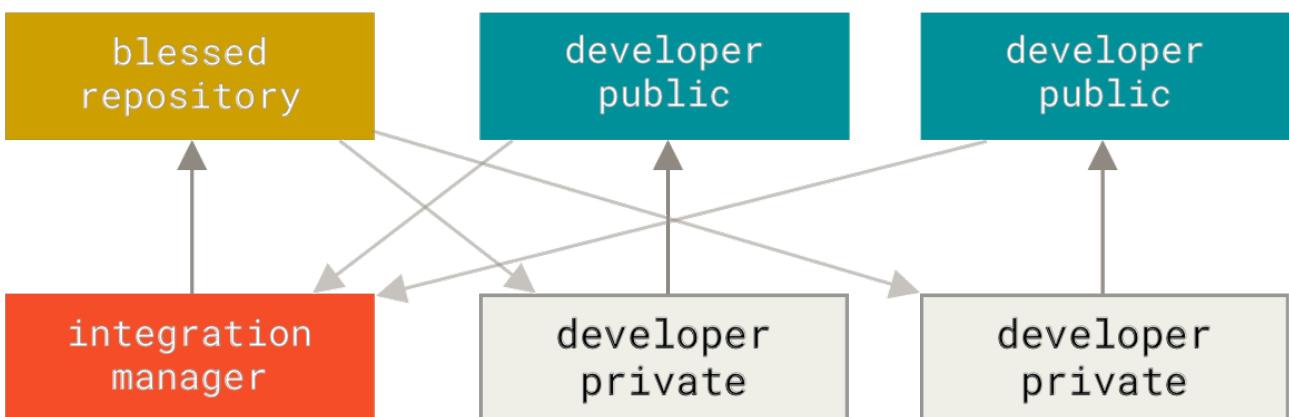


Figure 54. Arbeitsablauf mit Integrationsmanager

Dies ist ein sehr häufiger Workflow mit Hub-basierten Tools wie GitHub oder GitLab, bei dem es einfach ist, ein Projekt zu „forken“ und eigene Änderungen in deinen Fork hochzuladen, damit jeder sie sehen kann. Einer der Hauptvorteile dieses Ansatzes besteht darin, dass du weiterarbeiten kannst und der Verwalter des Haupt-Repositorys deine Änderungen jederzeit übernehmen kann. Die Mitwirkenden müssen nicht warten, bis das Projekt deine Änderungen übernommen hat – jede Partei kann in ihrem eigenen Tempo arbeiten.

## Arbeitsablauf mit Diktator und Leutnants

Dies ist eine Variante eines Workflows mit vielen Repositorys. Sie wird im Allgemeinen von großen Projekten mit Hunderten von Mitstreitern verwendet. Ein berühmtes Beispiel ist der Linux-Kernel. Verschiedene Integrationsmanager sind für bestimmte Teile des Repositorys verantwortlich. Sie heißen *Leutnants*. Alle Leutnants haben einen Integrationsmanager, der als der wohlwollende Diktator (benevolent dictator) bezeichnet wird. Der wohlwollende Diktator pusht von seinem Verzeichnis in ein Referenz-Repository, aus dem alle Beteiligten ihre eigenen Repositorys aktualisieren müssen. Dieser Prozess funktioniert wie folgt (siehe [Arbeitsablauf mit wohlwollendem Diktator](#)):

1. Entwickler arbeiten regelmäßig an ihrem Featurebranch und reorganisieren (rebasen) ihre Arbeit auf `master`. Der `master` Branch ist der des Referenz-Repositorys, in das der Diktator pusht.
2. Die Leutnants mergen die Featurebranches der Entwickler in ihrem `master` Branch.
3. Der Diktator führt die `master` Branches der Leutnants in den Branch `master` des Diktators

zusammen.

4. Schließlich pusht der Diktator diesen **master** Branch in das Referenz-Repository, damit die anderen Entwickler darauf einen Rebase durchführen können.

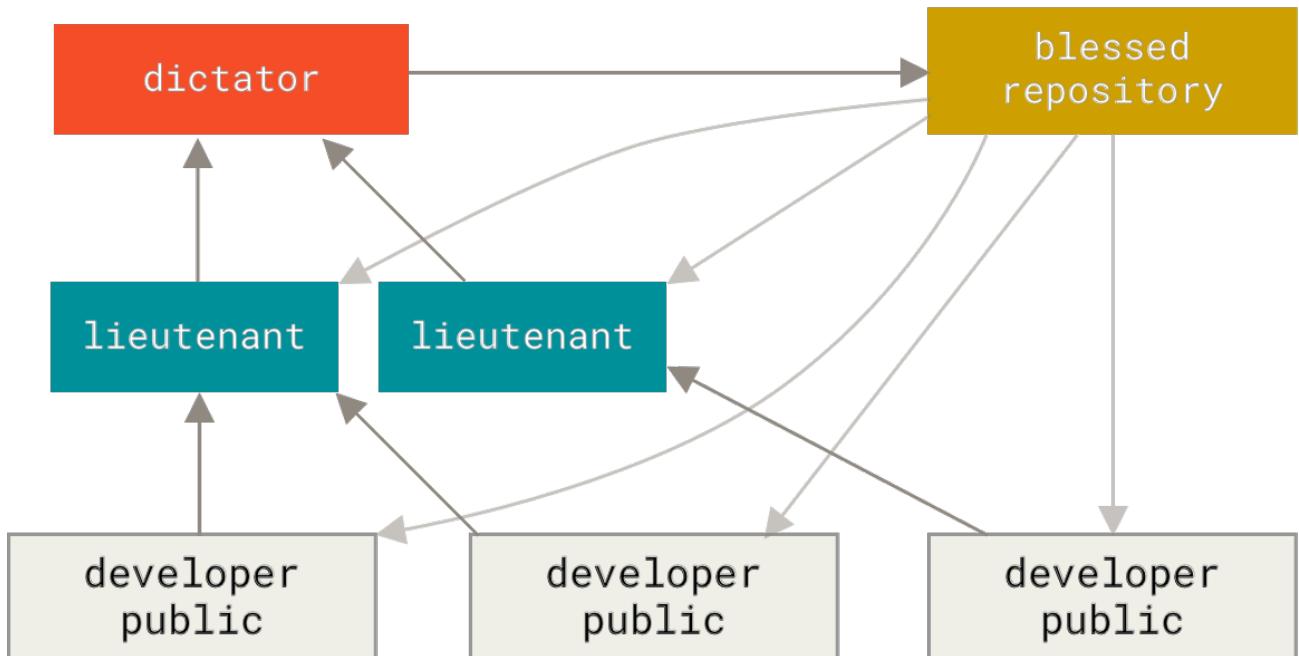


Figure 55. Arbeitsablauf mit wohlwollendem Diktator

Diese Art von Arbeitsablauf ist nicht weit verbreitet, kann jedoch in sehr großen Projekten oder in sehr hierarchischen Umgebungen hilfreich sein. Dies ermöglicht dem Projektleiter (dem Diktator), einen Großteil der Arbeit zu delegieren und große Teilbereiche von Quelltext an mehreren Stellen zu sammeln, bevor diese integriert werden.

## Methoden zur Verwaltung von Quellcode-Banches



Martin Fowler hat den Leitfaden „Patterns for Managing Source Code Branches“ (Methoden zur Verwaltung von Quellcode-Banches) erstellt. Dieser Leitfaden deckt alle gängigen Git-Workflows ab und erklärt, wie und wann sie eingesetzt werden sollten. Es gibt auch einen Abschnitt, in dem hohe und niedrige Integrationsfrequenzen verglichen werden.

<https://martinfowler.com/articles/branching-patterns.html>

## Zusammenfassung

Dies sind einige häufig verwendete Workflows, die mit einem verteilten System wie Git möglich sind. Allerdings sind auch viele Variationen möglich, um deinen eigenen Arbeitsabläufen gerecht zu werden. Jetzt, da du (hoffentlich) bestimmen kannst, welche Kombination von Arbeitsabläufen bei dir funktionieren würde, werden wir einige spezifischere Beispiele davon betrachten, wie man die Hauptaufgaben durchführen kann, welche die unterschiedliche Abläufe ausmachen. Im nächsten Abschnitt erfährst du etwas über gängige Formen der Mitarbeit an einem Projekt.

# An einem Projekt mitwirken

Die größte Schwierigkeit bei der Beschreibung, wie man an einem Projekt mitwirkt, sind die zahlreichen Varianten, wie man das tun könnte. Da Git sehr flexibel ist, können die Beteiligten auf viele Arten zusammenarbeiten. Es ist nicht einfach zu beschreiben, wie du dazu beitragen solltest, da jedes Projekt ein wenig anders ist. Einige der wichtigen Unbekannten sind die Anzahl der aktiven Mitwirkenden, der ausgewählte Arbeitsablauf, deine Zugriffsberechtigung und möglicherweise auch die Methode, wie externe Beiträge verwaltet werden sollen.

Die erste Unbekannte ist die Anzahl der aktiven Mitwirkenden – wie viele Benutzer tragen aktiv Quelltext zu diesem Projekt bei und wie oft? In vielen Fällen hast du zwei oder drei Entwickler mit ein paar Commits pro Tag oder möglicherweise weniger für etwas schlummernde Projekte. Bei größeren Unternehmen oder Projekten kann die Anzahl der Entwickler in die Tausende gehen, wobei jeden Tag Hunderte oder Tausende von Commits getätigt werden können. Das ist deshalb von Bedeutung, da du mit einer wachsenden Anzahl von Entwicklern auch sicherstellen musst, dass sich der Code problemlos anwenden lässt und leicht zusammengeführt werden kann. Von dir übermittelte Änderungen können durch Arbeiten, die während deiner Arbeit oder während deiner Änderungen genehmigt oder eingearbeitet wurden, veraltet sein oder nicht mehr funktionieren. Wie kannst du deinen Quelltext konsistent auf dem neuesten Stand halten und dafür sorgen, dass deine Commits gültig sind?

Die nächste Unbekannte ist der für das Projekt verwendete Arbeitsablauf. Ist er zentralisiert, wobei jeder Entwickler den gleichen Schreibzugriff auf die Hauptentwicklungslinie hat? Verfügt das Projekt über einen Betreuer oder Integrationsmanager, der alle Patches überprüft? Sind alle Patches von Fachleuten geprüft und genehmigt? Bist du selbst in diesen Prozess involviert? Ist ein Leutnant System vorhanden und musst du deine Arbeit zuerst bei diesen einreichen?

Die nächste Unbekannte ist deine Zugriffsberechtigung. Der erforderliche Arbeitsablauf, um zu einem Projekt beizutragen, unterscheidet sich erheblich, wenn du Schreibzugriff auf das Projekt hast, als wenn du diesen nicht hast. Wenn du keinen Schreibzugriff hast, in welcher Reihenfolge erfolgt die Annahme von beigetragener Arbeit? Gibt es überhaupt eine Richtlinie? Wie umfangreich sind die Änderungen, die du jeweils beisteuerst? Wie oft trägst du etwas bei?

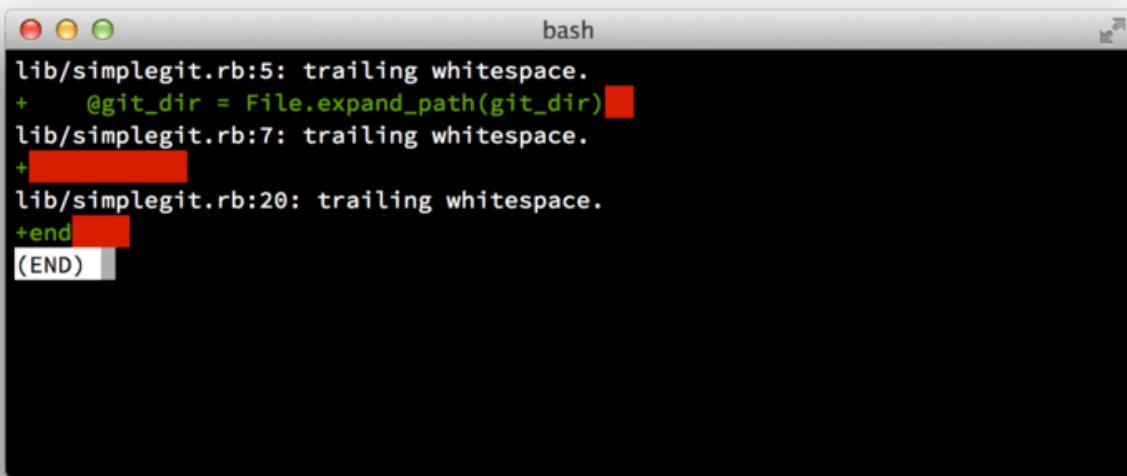
All diese Fragen können sich darauf auswirken, wie du effektiv zu einem Projekt beitragen und welche Arbeitsabläufe bevorzugt oder überhaupt für sie verfügbar sind. Wir werden jeden dieser Aspekte in einer Reihe von Anwendungsfällen behandeln, wobei wir mit simplen Beispielen anfangen und später komplexere Szenarios besprechen. Du solltest in der Lage sein, anhand dieser Beispiele die spezifischen Arbeitsabläufe zu erstellen, die du in der Praxis benötigst.

## Richtlinien zur Zusammenführung (engl. Commits)

Bevor wir uns mit den spezifischen Anwendungsfällen befassen, findest du hier einen kurzen Hinweis zu Commit-Nachrichten. Ein guter Leitfaden zum Erstellen von Commits und das Befolgen desselben, erleichtert die Arbeit mit Git und die Zusammenarbeit mit anderen erheblich. Das Git-Projekt selber enthält ein Dokument, in dem einige nützliche Tipps zum Erstellen von Commits für die Übermittlung von Patches aufgeführt sind. Du findest diese Tipps im Git-Quellcode in der Datei [Documentation/SubmittingPatches](#).

Deine Einsendungen sollten keine Leerzeichenfehler enthalten. Git bietet eine einfache

Möglichkeit, dies zu überprüfen. Führe vor dem Commit `git diff --check` aus, um mögliche Leerzeichenfehler zu identifizieren und diese aufzulisten.



```
lib/simplegit.rb:5: trailing whitespace.
+    @git_dir = File.expand_path(git_dir)
lib/simplegit.rb:7: trailing whitespace.
+ [REDACTED]
lib/simplegit.rb:20: trailing whitespace.
+end [REDACTED]
(END) [REDACTED]
```

Figure 56. Ausgabe von `git diff --check`

Wenn du diesen Befehl vor einem Commit ausführst, kannst du feststellen, ob Leerzeichen-Probleme auftreten, die andere Entwickler stören könnten.

Als Nächstes, versuchst du, aus jedem Commit einen logisch getrennten Satz von Änderungen zu machen. Versuche deine Änderungen leicht verständlich zu machen. Arbeiten nicht ein ganzes Wochenende an fünf verschiedenen Features und übermittel all diese Änderungen in einem massiven Commit am Montag. Auch wenn du am Wochenende keine Commits durchführst, nutze am Montag die Staging-Area, um deine Änderungen aufzuteilen in wenigstens einen Commit für jeden Teilaспект mit jeweils einer sinnvollen Nachricht. Wenn einige der Änderungen dieselbe Datei modifizieren, benutze die Anweisung `git add --patch`, um Dateien partiell zur Staging-Area hinzuzufügen (detailliert dargestellt im Abschnitt [Interaktives Stagen](#)). Der Schnappschuss vom Projekt am Ende des Branches ist der Selbe, ob du einen oder fünf Commits durchgeführt hast, solange nur all die Änderungen irgendwann hinzugefügt werden. Versuche also, die Dinge für deine Entwicklerkollegen zu vereinfachen, die deine Änderungen begutachten müssen.

Dieser Ansatz macht es außerdem einfacher, einen Satz von Änderungen zu entfernen oder rückgängig zu machen, falls das später nötig sein sollte. [Den Verlauf umschreiben](#) beschreibt eine Reihe nützlicher Git-Tricks zum Umschreiben des Verlaufs oder um interaktiv Dateien zur Staging-Area hinzuzufügen. Verwende diese Werkzeuge, um einen sauberen und leicht verständlichen Verlauf aufzubauen, bevor du deine Arbeit jemand anderem schickst.

Als letztes darf die Commit-Nachricht nicht vergessen werden. Macht man es sich zur Gewohnheit, qualitativ hochwertige Commit-Nachrichten zu erstellen, erleichtert dies die Verwendung und die Zusammenarbeit mit Git erheblich. In der Regel sollte deine Nachrichten mit einer einzelnen Zeile beginnen, die nicht länger als 50 Zeichen ist. Diese sollte deine Änderungen kurz und bündig beschreiben. Darauf folgen eine leere Zeile und eine ausführliche Erläuterung. Für das Git-Projekt ist es erforderlich, dass die ausführliche Erläuterung deine Motivation für die Änderung enthält. Außerdem sollte das Ergebnis deiner Implementierung mit dem vorherigen Verhalten des Projekts

gegenüber gestellt werden. Dies ist eine gute Richtlinie, an die man sich halten sollte. Es empfiehlt sich außerdem, die Gegenwartsform des Imperativs in diesen Nachrichten zu benutzen. Mit anderen Worten, verwende Anweisungen. Anstatt „Ich habe Test hinzugefügt für“ oder „Tests hinzufügend für“ benutze „Füge Tests hinzu für“. Hier ist eine Vorlage, die du nutzen kannst. Wir haben sie leicht angepasst. Das Original findest du hier: [E-Mail-Vorlage, die ursprünglich von Tim Pope geschrieben wurde](#):

#### Kurzfassung der Änderung (50 Zeichen oder weniger)

Gegebenenfalls ausführlicher, erläuternder Text. Pro Zeile etwa 72 Zeichen. In einigen Kontexten wird die erste Zeile wie der Betreff einer E-Mail gehandelt und der Rest des Textes als Textkörper. Die Leerzeile, welche die Zusammenfassung vom Text trennt ist von entscheidender Bedeutung (es sei denn, du lässt den Textkörper ganz weg). Werkzeuge wie rebase können Fehler machen, wenn du diese Leerzeile nicht einhältst.

Schreiben deine Commit Nachrichten im Imperativ: "Fix bug" und nicht "Fixed bug" oder "Fixes bug". Diese Konvention stimmt mit Commit-Nachrichten überein, die von Befehlen wie git merge und git revert generiert werden.

Weitere Absätze folgen nach Leerzeilen.

- Aufzählungszeichen sind auch in Ordnung
- In der Regel wird für das Aufzählungszeichen ein Bindestrich oder ein Sternchen verwendet, gefolgt von einem Leerzeichen.  
Zwischen den einzelnen Aufzählungen werden Leerzeilen eingefügt. Diese Konventionen variieren jedoch.

#### Verwende einen hängenden Einzug

Wenn alle deine Commit-Nachrichten diesem Modell folgen, wird es für dich und die Entwickler, mit denen du zusammenarbeitest, viel einfacher sein. Das Git-Projekt selber enthält gut formatierte Commit-Nachrichten. Versuche, `git log --no-merges` auszuführen, um zu sehen, wie ein gut formatierter Commit-Verlauf des Projekts aussieht.

*Tue das, was wir sagen und nicht das, was wir tun.*

Der Kürze halber haben viele der Beispiele in diesem Buch keine gut formatierten Commit-Nachrichten. Stattdessen verwenden wir einfach die Option `-m`, um `git commit` auszuführen.

Kurz gesagt, tue es wie wir es sagen und nicht wie wir es tun.

## Kleines, privates Team

Das einfachste Setup, auf das du wahrscheinlich stoßen wirst, ist ein privates Projekt mit einem oder zwei anderen Entwicklern. Privat bedeutet in diesem Zusammenhang „closed source“ – es ist für die Außenwelt nicht öffentlich zugänglich. Du und die anderen Entwickler haben alle

## Schreibzugriff (Push-Zugriff) auf das Repository.

In dieser Umgebung kannst du einem Arbeitsablauf folgen, der dem ähnelt, den du mit Subversion oder einem anderen zentralisierten System ausführen würdest. Du hast immer noch die Vorteile von Dingen wie Offline-Commit und ein wesentlich einfacheres Verzweigungs- (engl.branching) und Zusammenführungsmodel (engl. merging), aber der Arbeitsablauf kann sehr ähnlich sein. Hauptunterschied ist, dass das Zusammenführen eher auf der Client-Seite stattfindet als auf dem Server beim Durchführen eines Commits. Mal sehen, wie es aussehen könnte, wenn zwei Entwickler beginnen, mit einem gemeinsam genutzten Repository zusammenzuarbeiten. Der erste Entwickler, John, klont das Repository, nimmt eine Änderung vor und commitet es lokal. Die Protokollnachrichten wurden in diesen Beispielen durch `...` ersetzt, um sie etwas zu verkürzen.

```
# 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(-)
```

Die zweite Entwicklerin, Jessica, tut dasselbe — sie klont das Repository, ändert etwas und führt einen Commit durch.

```
# 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(-)
```

Nun lädt Jessica ihre Änderungen auf den Server hoch. Das funktioniert problemlos:

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

Die letzte Zeile der obigen Ausgabe zeigt eine nützliche Rückmeldung der Push Operation. Das Grundformat ist `<oldref> .. <newref> fromref → toref`, wobei `oldref` die alte Referenz bedeutet, `newref` die neue Referenz bedeutet, `fromref` der Name der lokalen Referenz ist, die übertragen wird, und `toref` ist der Name der entfernten Referenz, die aktualisiert werden soll. Eine ähnliche

Ausgabe findest du weiter unten in den Diskussionen. Wenn du also ein grundlegendes Verständnis der Bedeutung dieser Angaben hast, dann kannst du die verschiedenen Zustände der Repositorys besser verstehen. Weitere Informationen dazu findest du in der Dokumentation für [git-push](#).

Wenn wir mit diesem Beispiel fortfahren, nimmt John einige Änderungen vor, schreibt sie in sein lokales Repository und versucht, sie auf den gleichen Server zu übertragen:

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

John ist es nicht gestattet, seine Änderungen hochzuladen, weil Jessica vorher *ihre* hochgeladen hat. Dies ist wichtig zu verstehen, wenn du an Subversion gewöhnt bist. Wie du sicherlich bemerkt hast, haben die beiden Entwickler nicht dieselbe Datei bearbeitet. Obwohl Subversion eine solche Zusammenführung automatisch auf dem Server durchführt, wenn verschiedene Dateien bearbeitet werden, musst du bei Git die Commits *zuerst* lokal zusammenführen. Mit anderen Worten, John muss zuerst Jessicas Änderungen abrufen und in seinem lokalen Repository zusammenführen, bevor ihm das Hochladen gestattet wird.

Als ersten Schritt holt John, Jessicas Änderungen (dies holt nur Jessicas Änderungen, diese werden noch nicht mit Johns Änderungen zusammengeführt):

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

Zu diesem Zeitpunkt sieht Johns lokales Repository ungefähr so aus:

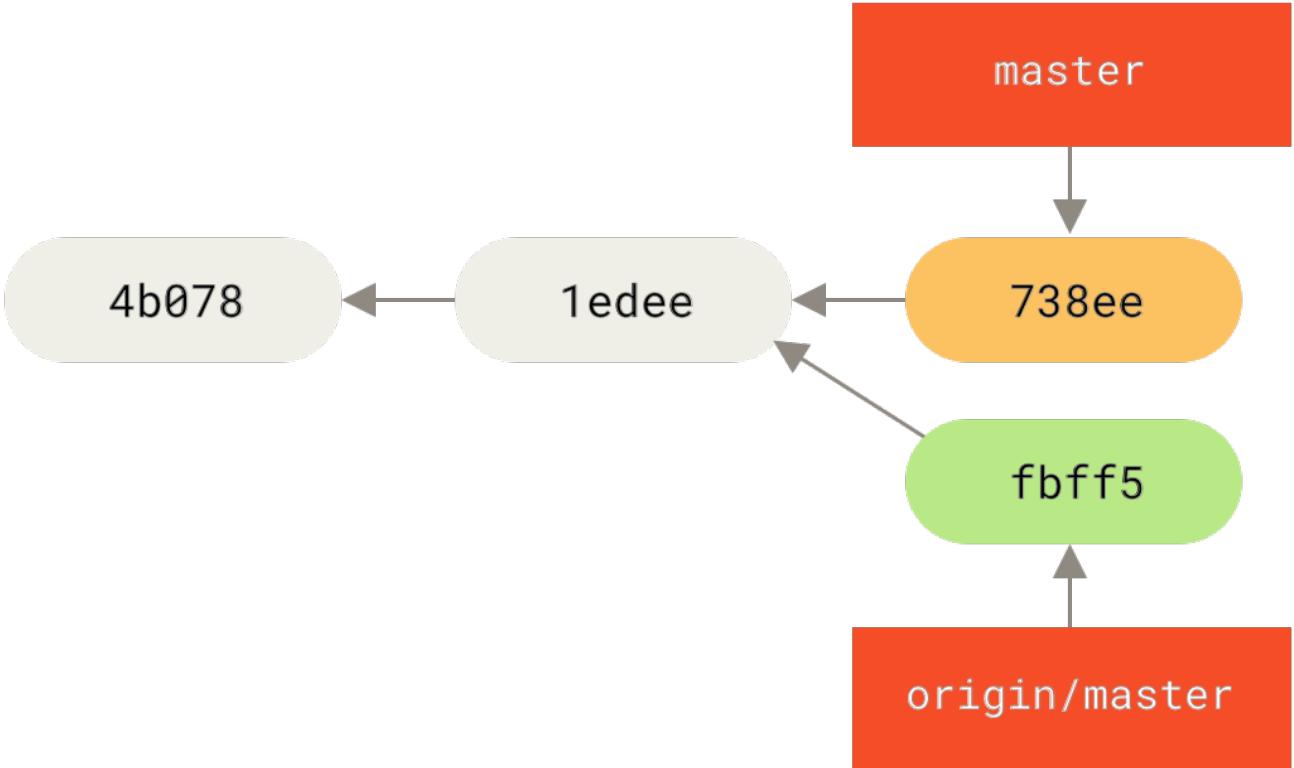


Figure 57. Johns abzweigender Verlauf

Jetzt kann John, Jessicas abgeholt Änderungen, zu seinen eigenen lokalen Änderungen zusammenführen:

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

Wenn diese lokale Zusammenführung reibungslos verläuft, sieht der aktualisierte Verlauf von John nun folgendermaßen aus:

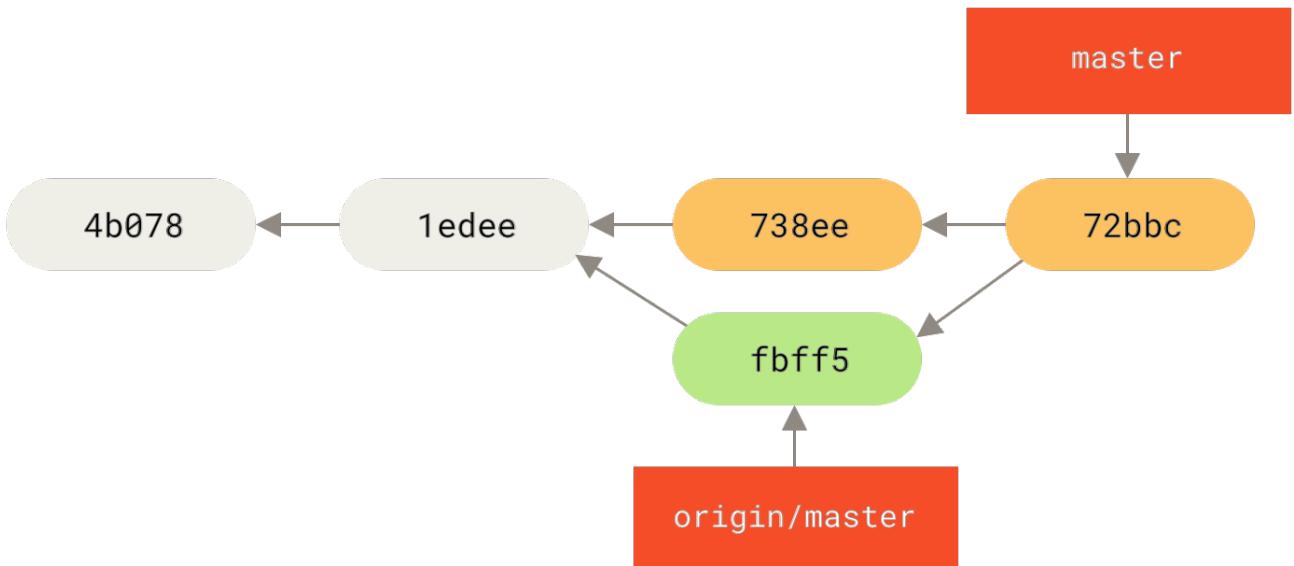


Figure 58. Johns Repository nach der Zusammenführung origin/master

Zu diesem Zeitpunkt möchte John möglicherweise diesen neuen Code testen, um sicherzustellen, dass sich keine der Arbeiten von Jessica auf seine auswirkt. Wenn alles in Ordnung ist, kann er die neu zusammengeführten Änderungen schließlich auf den Server übertragen:

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

Am Ende sieht Johns Commit-Verlauf so aus:

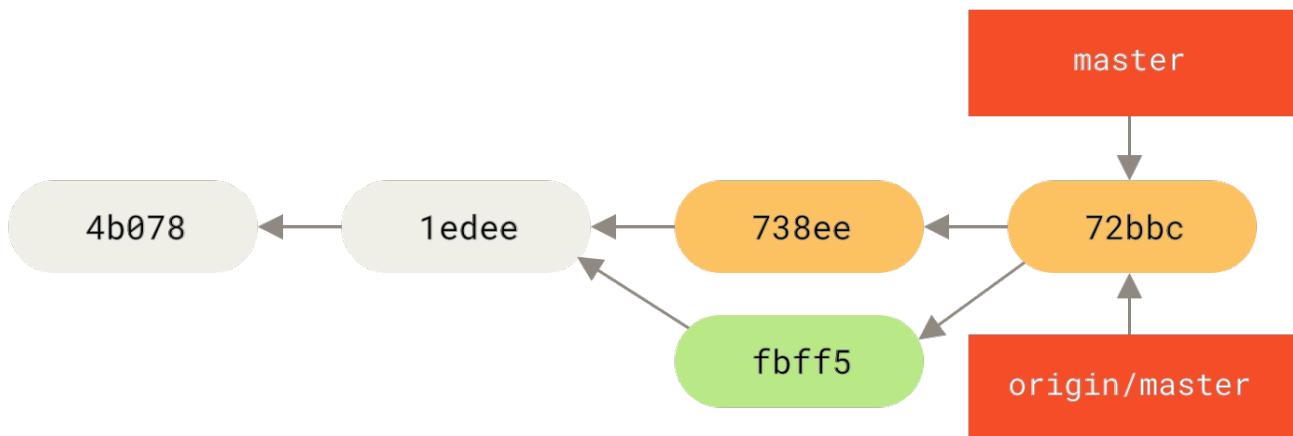


Figure 59. Johns Verlauf nach Hochladen auf den origin Server

In der Zwischenzeit hat Jessica einen neuen Branch mit dem Namen `issue54` erstellt und drei Commits auf diesem Branch vorgenommen. Sie hat Johns Änderungen noch nicht abgerufen, daher sieht ihr Commit-Verlauf folgendermaßen aus:

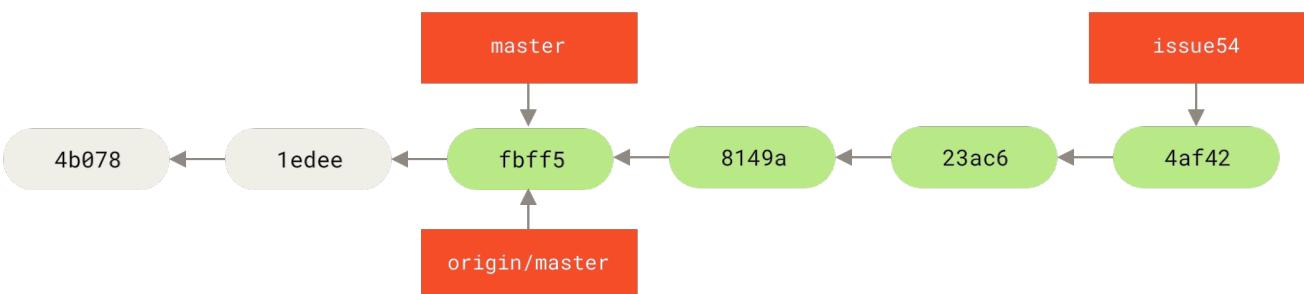


Figure 60. Jessicas Featurebranch

Nun erfährt Jessica, dass John einige neue Arbeiten auf den Server geschoben hat und sie möchte sich diese ansehen. Sie kann alle neuen Inhalte von dem Server abrufen über die sie noch nicht verfügt:

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

Dies zieht die Arbeit herunter (Pull), die John in der Zwischenzeit hochgeladen hat. Jessicas Verlauf sieht jetzt so aus:

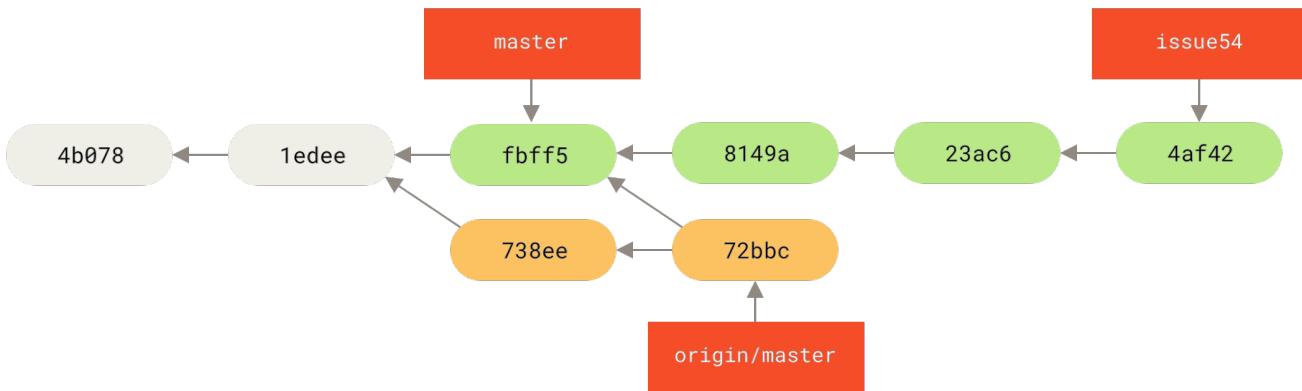


Figure 61. Jessicas Verlauf nach dem Abholen von Johns Änderungen

Jessica denkt, dass ihr Feature Branch nun fertig ist. Sie möchte jedoch wissen, welchen Teil von Johns abgerufenen Arbeiten sie in ihre Arbeit einbinden muss, damit sie hochladen kann. Sie führt `git log` aus, um das herauszufinden:

```
$ git log --no-merges issue54..origin/master
commit 738ee872852dfa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 16:01:27 2009 -0700

    Remove invalid default value
```

Die Syntax `issue54..origin/master` ist ein Logfilter, der Git anweist, nur die Commits anzuzeigen, die sich im letzterem Branch befinden (in diesem Fall `origin/master`) und nicht im ersten Branch (in diesem Fall `issue54`). Wir werden diese Syntax in [Commit-Bereiche](#) genauer erläutern.

Aus der obigen Ausgabe können wir sehen, dass es einen einzigen Commit gibt, den John gemacht hat, welchen Jessica nicht in ihre lokale Arbeit eingebunden hat. Wenn sie `origin/master` zusammenführt, ist dies der einzige Commit, der ihre lokale Arbeit verändert.

Jetzt kann Jessica ihre Arbeit in ihrem `master` Branch zusammenführen, Johns Arbeit (`origin/master`) in ihrem `master` Branch zusammenführen und dann wieder auf den Server hochladen.

Als erstes wechselt Jessica (nachdem sie alle Änderungen in ihrem FeatureBranch `issue54` committet hat) zurück zu ihrem `master` Branch, um diese Integration vorzubereiten:

```
$ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
```

Jessica kann entweder `origin/master` oder `issue54` mit ihrem lokalem `master` zusammenführen – beide sind ihrem `master` vorgelagert. Daher spielt die Reihenfolge keine Rolle. Der finale

Schnappschuss sollte unabhängig von der gewählten Reihenfolge identisch sein. Nur der Verlauf wird anders sein. Sie beschließt, zuerst den Branch **issue54** zusammenzuführen:

```
$ git merge issue54
Updating fbfff5bc..4af4298
Fast forward
 README          |    1 +
 lib/simplegit.rb |    6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Es treten keine Probleme auf. Wie du siehst, handelte es sich um eine einfache Schnellvorlauf Zusammenführung (engl. Fast-Forward). Jessica schließt nun den lokalen Zusammenführungsprozess ab, indem sie Johns zuvor abgerufene Arbeit zusammenführt, die sich im Branch **origin/master** befindet:

```
$ 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 kann sauber zusammengeführt werden. Jessicas Verlauf sieht nun so aus:

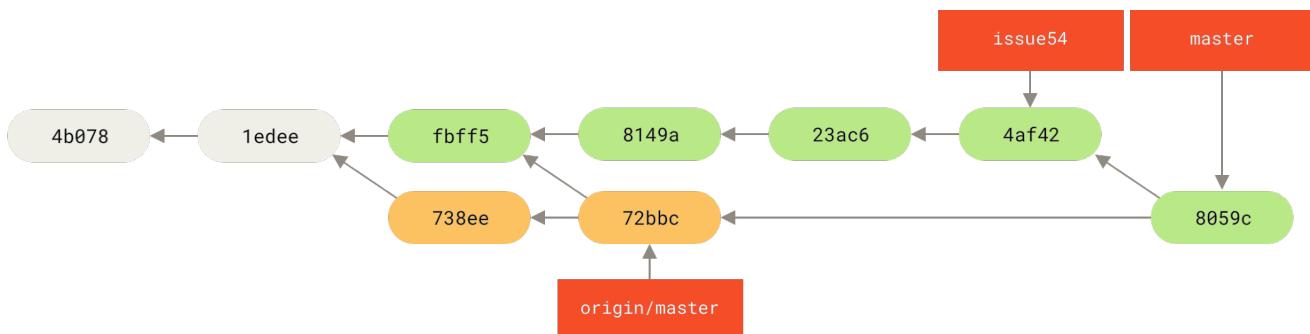


Figure 62. Jessicas Verlauf nach Zusammenführung mit Johns Änderungen

Jetzt ist **origin/master** über Jessicas **master** Branch erreichbar, sodass sie erfolgreich pushen kann (vorausgesetzt, John hat in der Zwischenzeit keine weiteren Änderungen hochgeladen):

```
$ git push origin master
...
To jessica@githost:simplegit.git
 72bbc59..8059c15  master -> master
```

Jeder Entwickler hat einige Commits durchgeführt und die Arbeit des jeweils anderen erfolgreich zusammengeführt.

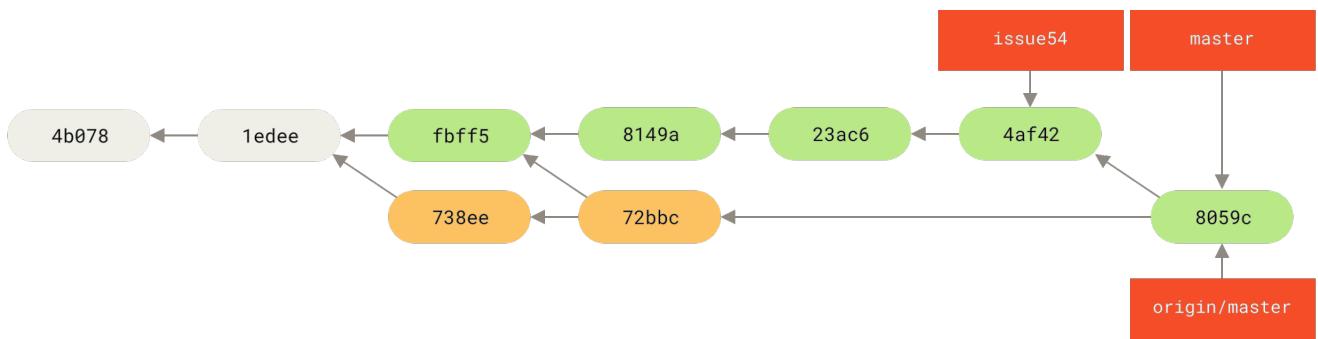


Figure 63. Jessicas Verlauf nach hochladen aller Änderungen auf den Server

Das ist einer der einfachsten Arbeitsabläufe. Du arbeitest eine Weile (in der Regel in einem Feature Branch) und führst diese Arbeiten in deinem Branch `master` zusammen, sobald sie für die Integration bereit sind. Wenn du diese Arbeit teilen möchtest, rufst du deinen `master` von `origin/master` ab und führst ihn zusammen, falls er sich geändert hat. Anschließend pushst du ihn in den `master` Branch auf dem Server. Die allgemeine Reihenfolge sieht in etwa so aus:

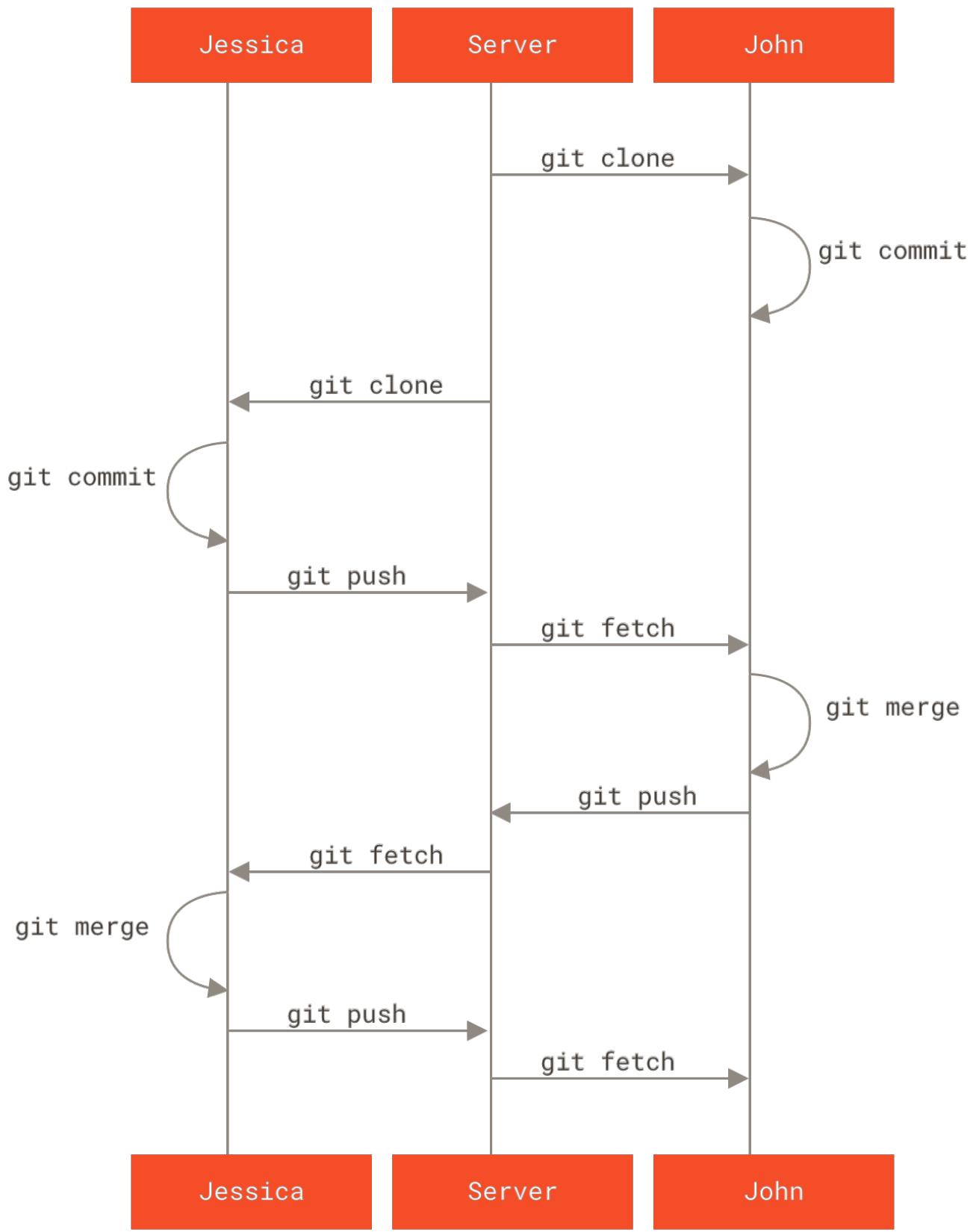


Figure 64. Allgemeine Abfolge von Ereignissen für einen einfachen Arbeitsablauf mit mehreren Entwicklern

## Geführtes, privates Team

In diesem Szenario sehen wir uns die Rollen der Mitwirkenden in einem größeren, geschlossenen Team an. Du lernst, wie Du in einer Umgebung arbeitest, in der kleine Gruppen an der Entwicklung einzelner Funktionen zusammenarbeiten. Anschließend werden diese teambasierten Beiträge von

einem anderen Beteiligten integriert.

Nehmen wir an, John und Jessica arbeiten gemeinsam an einem Feature (nennen wir dieses **FeatureA**), während Jessica und Josie, eine dritte Entwicklerin, an einem zweiten Feature arbeiten (sagen wir **FeatureB**). In diesem Fall verwendet das Unternehmen einen Arbeitsablauf mit Integrationsmanager. Bei diesem kann die Arbeit der einzelnen Gruppen nur von bestimmten Beteiligten integriert werden. Der Master-Branch des Haupt Repositorys kann nur von diesen Beteiligten aktualisiert werden. In diesem Szenario werden alle Arbeiten in teambasierten Branches ausgeführt und später vom Integrationsmanager zusammengeführt.

Folgen wir Jessicas Arbeitsablauf, während sie an ihren beiden Features tätig ist und parallel mit zwei verschiedenen Entwicklern in dieser Umgebung arbeitet. Wir nehmen an, sie hat ihr Repository bereits geklont. Zuerst beschließt sie an **featureA** zu arbeiten. Sie erstellt einen neuen Branch für das Feature und führt dort einige Änderungen aus:

```
# 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(-)
```

Zu diesem Zeitpunkt muss sie ihre Arbeit mit John teilen, also lädt sie ihre **featureA** Branch Commits auf den Server hoch. Jessica hat keinen Push-Zugriff auf den **master** Branch, nur die Integrationsmanager haben das. Sie muss daher auf einen anderen Branch hochladen, um mit John zusammenzuarbeiten:

```
$ git push -u origin featureA
...
To jessica@githost:simplegit.git
 * [new branch]      featureA -> featureA
```

Jessica schickt John eine E-Mail, um ihm mitzuteilen, dass sie einige Arbeiten in einen Branch mit dem Namen **featureA** hochgeladen hat. Er kann sie sich jetzt ansehen. Während sie auf Rückmeldung von John wartet, beschließt Jessica, mit Josie an **featureB** zu arbeiten. Zunächst startet sie einen neuen Feature-Branch, der auf dem **master** Branch des Servers basiert:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch 'featureB'
```

Jetzt macht Jessica ein paar Commits auf dem Branch **featureB**:

```
$ vim lib/simplegit.rb
```

```
$ git commit -am 'Make ls-tree function recursive'
[featureB e5b0fdc] Make ls-tree function recursive
 1 files changed, 1 insertions(+), 1 deletions(-)
$ vim lib/simplegit.rb
$ git commit -am 'Add ls-files'
[featureB 8512791] Add ls-files
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Jessicas Repository sieht nun folgendermaßen aus:

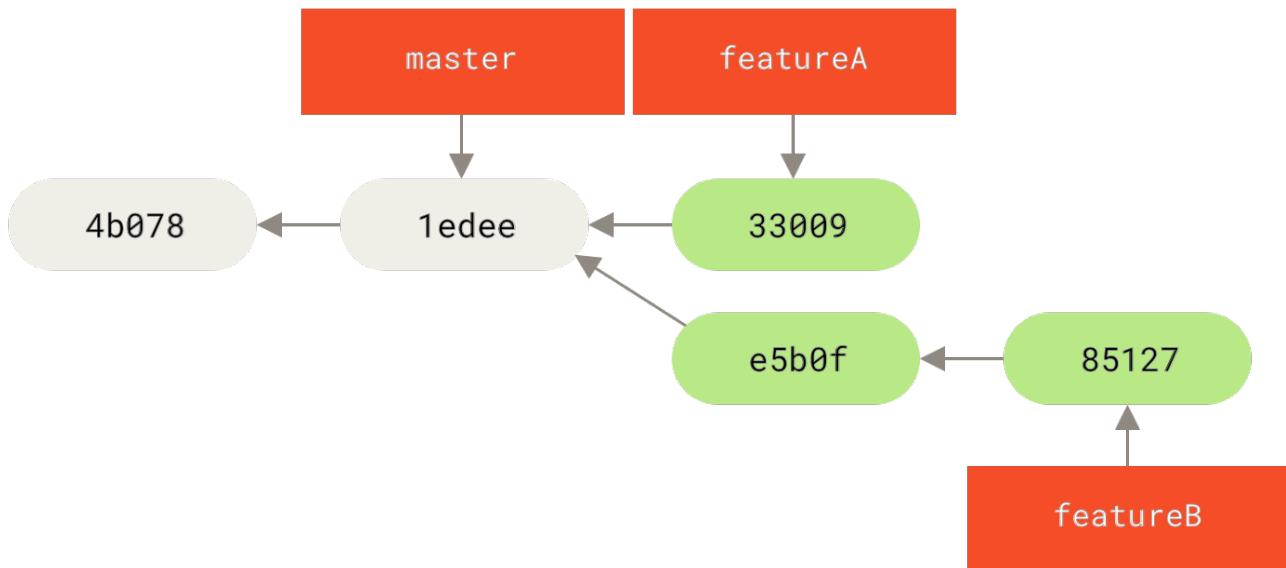


Figure 65. Jessicas initialer Commit Verlauf

Sie ist bereit, ihre Arbeit hochzuladen, erhält jedoch eine E-Mail von Josie, dass ein Branch mit einigen anfänglichen `featureB` Aufgaben bereits als `featureBee` Branch auf den Server übertragen wurde. Jessica muss diese Änderungen mit ihren eigenen zusammenführen, bevor sie ihre Arbeit auf den Server übertragen kann. Jessica holt sich zuerst Josies Änderungen mit `git fetch`:

```
$ git fetch origin
...
From jessica@githost:simplegit
 * [new branch]      featureBee -> origin/featureBee
```

Angenommen Jessica befindet sich noch in ihrem ausgecheckten `featureB` Branch. Dann kann sie nun Josies Arbeit mit `git merge` in diesen Branch zusammenführen:

```
$ 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(-)
```

Nun möchte Jessica die gesamte zusammengeführte Arbeit an `featureB` zurück auf den Server übertragen. Jedoch möchte sie nicht einfach ihren eigenen Branch `featureB` übertragen. Da Josie

bereits einen Upstream Branch `featureBee` gestartet hat, möchte Jessica auf diesen Branch hochladen, was sie auch folgendermaßen tut:

```
$ git push -u origin featureB:featureBee
...
To jessica@githost:simplegit.git
  fba9af8..cd685d1  featureB -> featureBee
```

Dies wird als *refspec* bezeichnet. Unter [Die Referenzspezifikation \(engl. Refspec\)](#) findest du eine detailliertere Beschreibung der Git-Refspecs und der verschiedenen Möglichkeiten, die du damit hast. Beachte auch die `-u` Option. Dies ist die Abkürzung für `--set-upstream`, mit der die Branches so konfiguriert werden, dass sie später leichter gepusht und gepulkt werden können.

Als nächstes erhält Jessica eine E-Mail von John, der ihr mitteilt, dass er einige Änderungen am Branch `featureA` vorgenommen hat, an dem sie zusammenarbeiten. Er bittet Jessica, sie sich anzusehen. Wieder führt Jessica ein einfaches `git fetch` durch, um *alle* neuen Inhalte vom Server abzurufen, einschließlich (natürlich) Johns neuester Arbeit:

```
$ git fetch origin
...
From jessica@githost:simplegit
  3300904..aad881d  featureA -> origin/featureA
```

Jessica kann sich Johns neue Arbeit ansehen, indem sie den Inhalt des neu abgerufenen Branches `featureA` mit ihrer lokalen Kopie desselben Branches vergleicht:

```
$ git log featureA..origin/featureA
commit aad881d154acdaeb2b6b18ea0e827ed8a6d671e6
Author: John Smith <jsmith@example.com>
Date:   Fri May 29 19:57:33 2009 -0700

  Increase log output to 30 from 25
```

Wenn ihr Johns neue Arbeit gefällt, kann sie sie mit ihrem lokalen Branch `featureA` zusammenführen:

```
$ 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(-)
```

Schließlich möchte Jessica noch ein paar geringfügige Änderungen an dem gesamten, zusammengeführten Inhalt vornehmen. Sie kann diese Änderungen vornehmen, indem sie in ihren

lokalen Branch **featureA** comittet und das Endergebnis zurück auf den Server überträgt (pusht):

```
$ git commit -am 'Add small tweak to merged content'
[featureA 774b3ed] Add small tweak to merged content
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
 3300904..774b3ed  featureA -> featureA
```

Jessicas commit Verlauf sieht nun in etwa so aus:

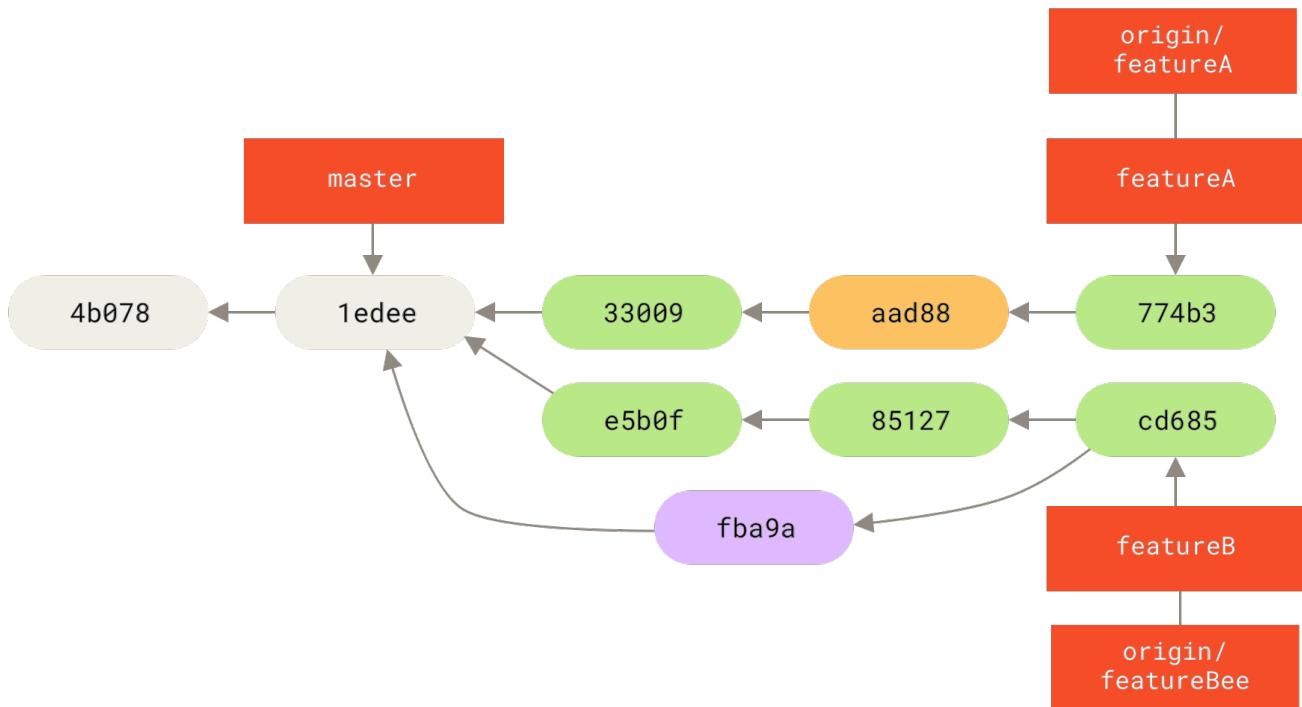


Figure 66. Jessicas Verlauf nach committen auf einem Feature Branch

Irgendwann informieren Jessica, Josie und John die Integratoren, dass die Branches **featureA** und **featureBee** auf dem Server für die Integration in die Hauptlinie bereit sind. Nachdem die Integratoren diese Branches in der Hauptlinie zusammengeführt haben, holt ein Abruf den neuen Zusammenführungs-Commit ab, sodass der Verlauf wie folgt aussieht:

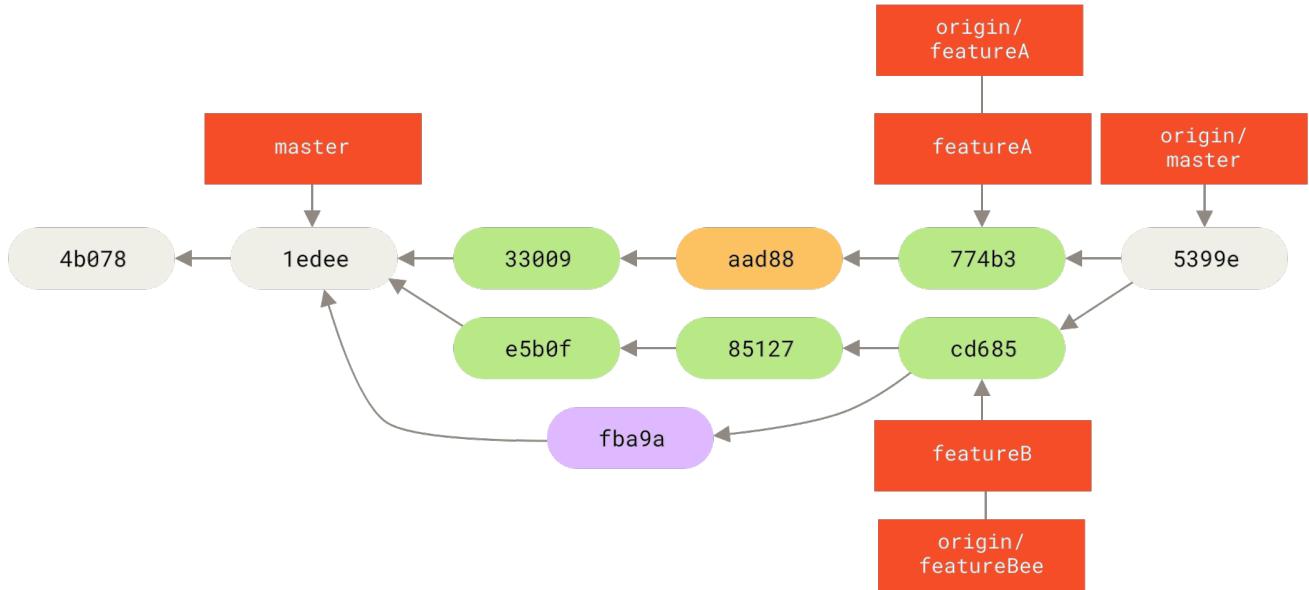


Figure 67. Jessicas Verlauf nach Zusammenführung ihrer beiden Featurebranches

Viele Teams wechseln zu Git, da sie parallel arbeiten können und die verschiedenen Entwicklungslinien zu einem späteren Zeitpunkt zusammengeführt werden können. Ein großer Vorteil von Git besteht darin, dass man in kleinen Untergruppen eines Teams über entfernte Branches zusammenarbeiten kann, ohne notwendigerweise das gesamte Team zu involvieren oder zu behindern. Die Vorgehensweise dieses Arbeitsablaufes, sieht in etwa so aus:

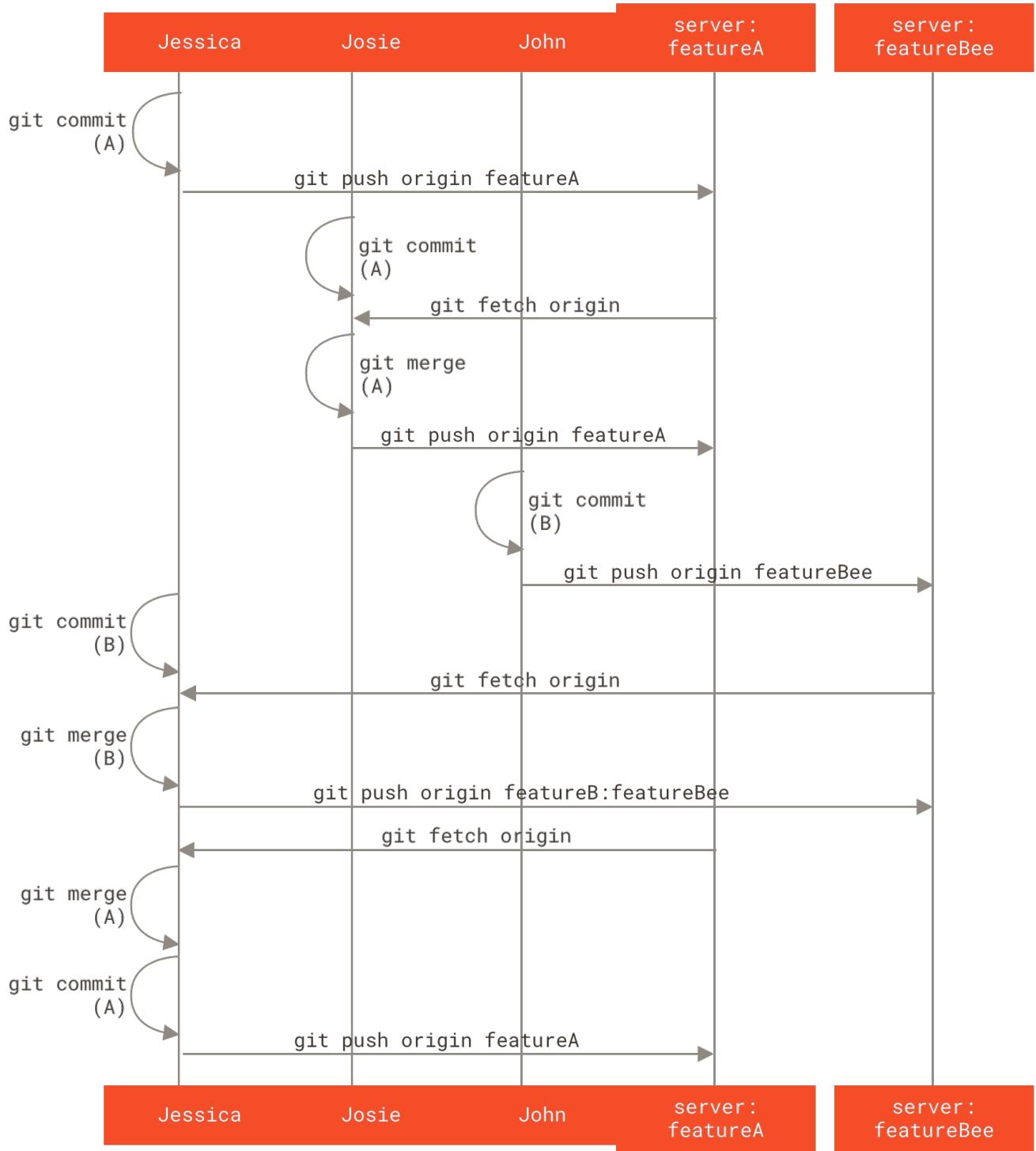


Figure 68. Grundlegende Vorgehensweise bei einem geführten Team

## Verteiltes, öffentliches Projekt

An öffentlichen Projekten mitzuwirken ist ein wenig anders. Da du nicht die Berechtigung hast, Branches im Projekt direkt zu aktualisieren, musst du deine Arbeit auf andere Weise an die Projektbetreuer weiterleiten. In diesem ersten Beispiel wird beschrieben, wie du auf Git Hosts, die einfaches „Forking“ unterstützen, via „Forking“ mitwirken kannst. Viele Hosting-Sites unterstützen dies (einschließlich GitHub, BitBucket, repo.or.cz und andere) und viele Projektbetreuer erwarten diese Art der Mitarbeit. Der nächste Abschnitt befasst sich mit Projekten, die bereitgestellte Patches bevorzugt per E-Mail akzeptieren.

Zunächst möchtest du wahrscheinlich das Hauptrepository klonen, einen Branch für den Patch oder die Patch Serien erstellen, die du beisteuern möchtest, und dort deine Arbeit erledigen. Der Prozess sieht im Grunde so aus:

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
... work ...
$ git commit
... work ...
$ git commit
```

 Du kannst `rebase -i` verwenden, um deine Arbeit auf ein einzelnen Commit zu reduzieren. Du kannst auch die Änderungen in den Commits neu anordnen, damit der Betreuer den Patch einfacher überprüfen kann – siehe [Den Verlauf umschreiben](#) für weitere Informationen zum interaktiven Rebasing.

Wenn deine Arbeit am Branch abgeschlossen ist und du bereit bist, sie an die Betreuer weiterzuleiten, wechsel zur ursprünglichen Projektseite. Dort klickst du auf die Schaltfläche `Fork`, um deinen eigenen schreibbaren Fork des Projekts zu erstellen. Anschließend musst du diese Repository-URL als neue Remote-Adresse deines lokalen Repositorys hinzufügen. Nennen wir es in diesem Beispiel `myfork`:

```
$ git remote add myfork <url>
```

Anschließend musst du deine neue Arbeit in dieses Repository hochladen. Es ist am einfachsten, den Branch, an dem du arbeitest, in dein geforktes Repository hochzuladen, anstatt diese Arbeit in deinem 'master'-Branch zusammenzuführen und diesen hochzuladen. Der Grund dafür ist, dass du deinen `master` Branch nicht zurücksetzen musst, wenn deine Arbeit nicht akzeptiert bzw. nur teilweise übernommen (cherry-pick) wurde (die Git-Operation zum `cherry-pick` wird ausführlicher in [Rebasing und Cherry-Picking Workflows](#) behandelt). Wenn die Betreuer deine Arbeit per `merge`, `rebase` oder `cherry-pick` übernehmen, erhältst du deine Arbeit sowieso zurück, wenn du aus dem Repository der Betreuer pullst.

Auf jedem Fall kannst du deine Arbeit hochladen mit:

```
$ git push -u myfork featureA
```

Sobald deine Arbeit an deinem Fork des Repositorys hochgeladen wurde, musst du den Betreuern des ursprünglichen Projekts mitteilen, dass es Änderungen gibt, die sie zusammenführen möchten. Dies wird oft als *Pull Request* bezeichnet. Du generierst eine solche Anfrage entweder über die Website – GitHub hat einen eigenen „Pull-Request-Mechanismus“, den wir in [GitHub](#) behandeln werden, oder du kannst den Befehl `git request-pull` ausführen und die nachfolgende Ausgabe manuell per E-Mail an den Projektbetreuer senden.

Der Befehl `git request-pull` verwendet den Basis Branch, in den dein Feature Branch abgelegt

werden soll. Außerdem wird die Git-Repository-URL angegeben aus dem er gezogen werden soll. Er erstellt damit eine Zusammenfassung aller Änderungen, um deren Übernahme du bittest. Wenn bspw. Jessica an John eine Pull Request senden möchte und sie zwei Commits für den gerade hochgeladenen Featurebranch ausgeführt hat, kann sie folgendes ausführen:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
Jessica Smith (1):
    Create new function

are available in the git repository at:

    https://githost/simplegit.git featureA

Jessica Smith (2):
    Add limit to log function
    Increase log output to 30 from 25

lib/simplegit.rb | 10 ++++++----
1 files changed, 9 insertions(+), 1 deletions(-)
```

Diese Ausgabe kann an den Betreuer gesendet werden. Sie teilt ihm mit, von wo die Arbeit gebranched wurde, fasst die Commits zusammen und gibt an, von wo die neue Arbeit abgerufen werden soll.

Bei einem Projekt, für das du nicht der Betreuer bist, ist es im Allgemeinen einfacher, einen Branch wie `master` zu haben, der immer `origin/master` folgt. Deine Arbeit kannst du dann in Feature Branches erledigen, die du einfach verwerfen kannst, wenn deine Änderungen abgelehnt werden. Durch das Isolieren von Änderungen in Feature Branches wird es für dich auch einfacher, deine Arbeit neu zu strukturieren. Falls sich das Haupt-Repository in der Zwischenzeit weiter entwickelt hat und deine Commits nicht mehr sauber angewendet werden können. Wenn du beispielsweise ein zweites Feature an das Projekt senden möchtest, arbeitest du nicht weiter an dem Branch, den du gerade hochgeladen hast. Beginne erneut im `master` Branch des Haupt-Repositorys:

```
$ 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
```

Jetzt ist jedes deiner Features in einer Art Silo enthalten, ähnlich wie bei einer Patch-Warteschlange. Dieses kannst du umarbeiten, zurücksetzen oder ändern, ohne dass die Features sich gegenseitig stören oder voneinander abhängig sind.

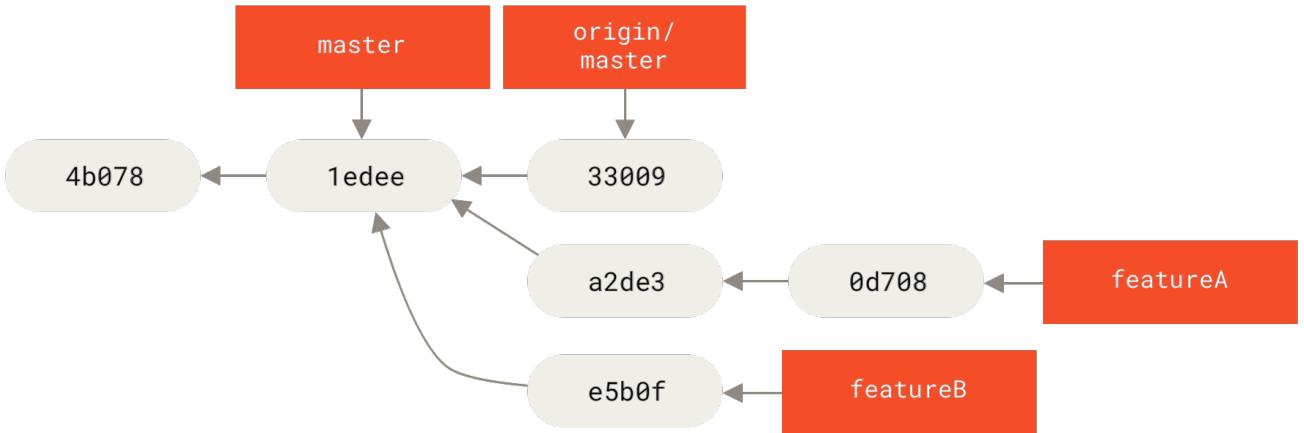


Figure 69. Initialer Commit Verlauf mit **featureB** Änderungen

Nehmen wir an, der Projektbetreuer hat eine Reihe weiterer Patches übernommen und deinen ersten Branch einfließen lassen, der jedoch nicht mehr ordnungsgemäß zusammengeführt werden kann. In diesem Fall kannst du versuchen, diesen Branch auf `origin/master` zu reorganisieren, die Konflikte für den Betreuer zu lösen und deine Änderungen erneut zu übermitteln:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

Dadurch wird dein Verlauf so umgeschrieben, sodass er jetzt folgendermaßen aussieht [Commit Verlauf nach featureA Änderungen](#).

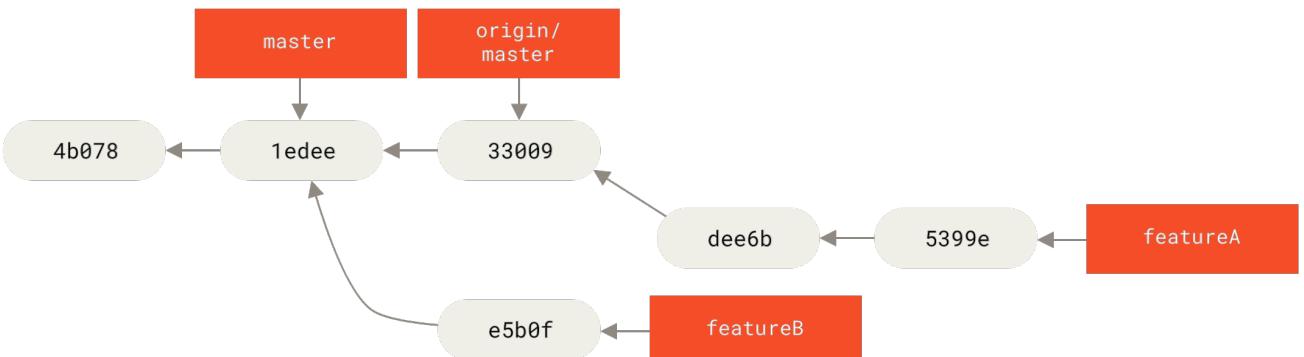


Figure 70. Commit Verlauf nach **featureA** Änderungen

Da du den Branch reorganisiert hast, musst du das **-f** für deinen Push-Befehl angeben, um den **featureA** Branch auf dem Server mit einem Commit ersetzen zu können, der nicht vom gegenwärtig letzten Commit des entfernten Branches abstammt. Eine Alternative wäre, diese neue Arbeit in einen anderen Branch auf dem Server hochzuladen (beispielsweise als **featureAv2**).

Schauen wir uns ein weiteres mögliches Szenario an: Der Betreuer hat sich die Arbeit in deinem zweiten Branch angesehen und mag dein Konzept, möchte aber, dass du ein Implementierungsdetail änderst. Du nutzt diese Gelegenheit, um deine Änderungen zu verschieben, damit diese auf dem aktuellen **master** Branch des Projektes basieren. Du startest einen neuen Branch, der auf den aktuellen Branch **origin/master** basiert und fasst die Änderungen an

**featureB** dort zusammen. Dabei löst du etwaige Konflikte, machst die Implementierungsänderungen und lädst diese Arbeiten als neuen Branch hoch:

```
$ git checkout -b featureBv2 origin/master
$ git merge --squash featureB
... change implementation ...
$ git commit
$ git push myfork featureBv2
```

Mit der Option `--squash` wird die gesamte Arbeit an dem zusammengeführten Branch in einen Änderungssatz komprimiert. Dadurch wird ein Repository-Status erzeugt, als ob eine echter Commit stattgefunden hätte, ohne dass tatsächlich ein Merge-Commit durchgeführt wurde. Dies bedeutet, dass dein zukünftiger Commit nur einen übergeordneten Vorgänger hat. Das erlaubt dir alle Änderungen aus einem anderen Branch einzuführen und weitere Änderungen vorzunehmen, bevor du den neuen Commit aufnimmst. Auch die Option `--no-commit` kann nützlich sein, um den Merge-Commit im Falle des Standard-Merge-Prozesses zu verzögern.

Nun kannst du den Betreuer darüber informieren, dass du die angeforderten Änderungen vorgenommen hast und dass du diese Änderungen in deinem Branch **featureBv2** findest.

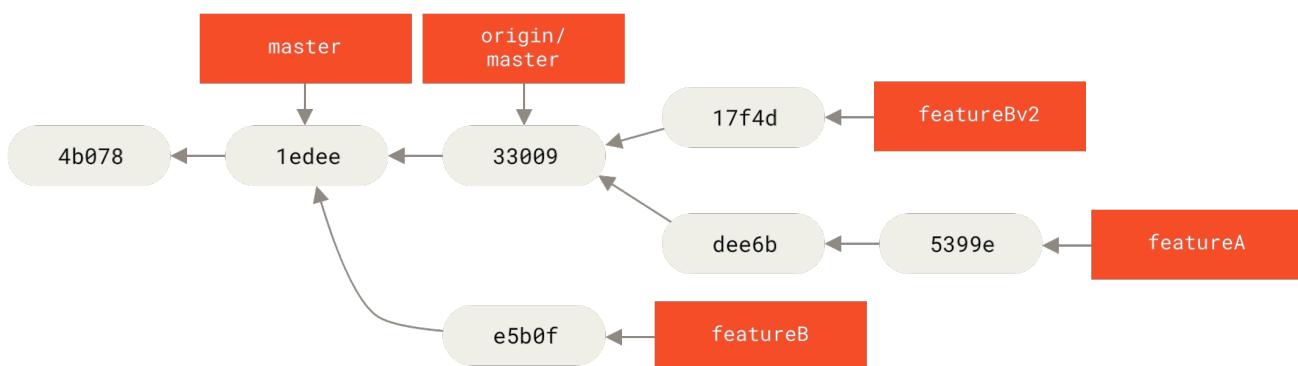


Figure 71. Commit Verlauf nach getaner **featureBv2** Arbeit

## Öffentliche Projekte via Email

Viele Projekte haben fest definierte Prozesse, um Änderungen entgegenzunehmen. Du musst die spezifischen Regeln dieser Projekte kennen, da sie sich oft unterscheiden. Da es viele alte und große Projekte gibt, die Änderungen über eine Entwickler-Mailingliste akzeptieren, werden wir jetzt solch ein Beispiel durchgehen.

Der Workflow ähnelt dem vorherigen Anwendungsfall: Du erstellst Feature Branches für jede Patch Serie, an der du arbeitest. Der Unterschied besteht darin, wie du diese Änderungen an das Projekt sendest. Anstatt das Projekt zu forken und auf dein eigenes geforktes Repository hochzuladen, generierst du E-Mail-Versionen jeder Commit-Serie und sendest diese per E-Mail an die Entwickler-Mailingliste:

```
$ git checkout -b topicA
```

```
... work ...
$ git commit
... work ...
$ git commit
```

Jetzt hast du zwei Commits, die du an die Mailingliste senden kannst. Du verwendest `git format-patch`, um die mbox-formatierten Dateien zu generieren, die du anschließend per E-Mail an die Mailingliste sendest. Dabei wird jeder Commit in eine E-Mail-Nachricht umgewandelt. Die erste Zeile der Commit-Nachricht wird als Betreff verwendet. Der Rest der Commit-Nachricht plus den Patch, den der Commit einführt wird als Mail-Körper verwendet. Der Vorteil daran ist, dass durch das Anwenden eines Patches aus einer mit `format-patch` erstellten E-Mail alle Commit-Informationen ordnungsgemäß erhalten bleiben.

```
$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-increase-log-output-to-30-from-25.patch
```

Der Befehl `format-patch` gibt die Namen der von ihm erstellten Patch-Dateien aus. Die `-M` Option weist Git an, nach Umbenennungen zu suchen. Die Dateien sehen am Ende folgendermaßen aus:

```
$ 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
```

Du kannst diese Patch-Dateien auch bearbeiten, um weitere Informationen für die E-Mail-Liste hinzuzufügen, die nicht in der Commit-Nachricht angezeigt werden sollen. Wenn du Text zwischen der Zeile `---` und dem Beginn des Patches (der Zeile `diff --git`) einfügst, können die Entwickler diesen Text lesen. Der Inhalt wird jedoch vom Patch-Vorgang ignoriert.

Um dies nun per E-Mail an eine Mailingliste zu senden, kannst du die Datei entweder an eine Mail anhängen oder über ein Befehlszeilenprogramm direkt versenden. Das Einfügen von Text führt häufig zu Formatierungsproblemen, insbesondere bei „intelligenten“ Clients, bei denen Zeilenumbrüche und andere Leerzeichen nicht ordnungsgemäß beibehalten werden. Glücklicherweise bietet Git ein Tool, mit dem du ordnungsgemäß formatierte Patches über IMAP senden kannst, was einfacher für dich sein könnte. Wir zeigen dir, wie du einen Patch über Google Mail sendest. Dies ist der E-Mail-Agent, mit dem wir uns am besten auskennen. Detaillierte Anweisungen für eine Reihe von anderen Mail-Programmen findest du am Ende der oben genannten Datei [Documentation/SubmittingPatches](#) im Git-Quellcode.

Zuerst musst du den Abschnitt `imap` in deiner `~/.gitconfig` Datei einrichten. Du kannst jeden Wert separat mit einer Reihe von `git config` Befehlen festlegen oder manuell hinzufügen. Am Ende sollte deine Konfigurationsdatei ungefähr so aussehen:

```
[imap]
  folder = "[Gmail]/Entwürfe"
  host = imaps://imap.gmail.com
  user = user@gmail.com
  pass = YX]8g76G_2^sFbd
  port = 993
  sslverify = false
```

Wenn dein IMAP-Server kein SSL verwendet, sind die letzten beiden Zeilen wahrscheinlich nicht erforderlich. Der Hostwert lautet dann `imap://` anstelle von `imaps://`. Wenn dies eingerichtet ist, kannst du `git imap-send` verwenden, um die Patch-Reihe im Ordner `Entwürfe` des angegebenen IMAP-Servers abzulegen:

```
$ 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
```

Zu diesem Zeitpunkt solltest du in der Lage sein, in deinem Entwurfsordner zu wechseln und dort das Feld An der generierten Email in die Mailinglist-Adresse zu ändern, an die du den Patch senden willst. Möglicherweise willst du auch den Betreuer oder die Person in Kopie nehmen, die für diesen Abschnitt verantwortlich ist. Anschließend kannst du die Mail versenden.

Du kannst die Patches auch über einen SMTP-Server senden. Wie zuvor kannst du jeden Wert separat mit einer Reihe von `git config` Befehlen festlegen oder manuell im Abschnitt `sendemail` in deiner `~/.gitconfig` Datei hinzufügen:

```
[sendemail]
smtpencryption = tls
smtpserver = smtp.gmail.com
smtpuser = user@gmail.com
smtpserverport = 587
```

Danach kannst du deine Patches mit `git send-email` versenden:

```
$ git send-email *.patch
0001-add-limit-to-log-function.patch
0002-increase-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Git gibt anschließend für jeden Patch, den du versendest, eine Reihe von Protokollinformationen aus, die in etwa so aussehen:

```
(mbox) Adding cc: Jessica Smith <jessica@example.com> from
      \line 'From: Jessica Smith <jessica@example.com>'
OK. Log says:
Sendmail: /usr/sbin/sendmail -i jessica@example.com
From: Jessica Smith <jessica@example.com>
To: jessica@example.com
Subject: [PATCH 1/2] Add limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```



Weitere Informationen zum Konfigurieren deines Systems und deiner E-Mail-Adresse, weitere Tipps und Tricks sowie eine Sandbox zum Senden eines Test-Patches per E-Mail findest du unter [git-send-email.io](http://git-send-email.io).

## Zusammenfassung

In diesem Abschnitt haben wir mehrere Workflows behandelt und über die Unterschiede zwischen der Arbeit an Closed-Source-Projekten in einem kleinen Teams und der Mitarbeit an einem großen öffentlichen Projekt gesprochen. Du musst vor dem Committen nach White-Space-Fehlern suchen und kannst großartige Commit-Beschreibungen hinzufügen. Du hast gelernt, wie du Patches formatieren und per E-Mail an eine Entwickler-Mailingliste senden kannst. Der Umgang mit Merges wurde auch im Zusammenhang mit den verschiedenen Arbeitsabläufen behandelt. Du bist jetzt gut

vorbereitet, an jedem Projekt mitzuarbeiten.

Als Nächstes erfährst du, wie du auf der anderen Seite arbeitest: als Verwalter (Maintainer) eines Git-Projektes. Du lernst, wie man als wohlwollender Diktator oder Integrationsmanager korrekt arbeitet.

## Ein Projekt verwalten

Du musst nicht nur wissen, wie du effektiv zu einem Projekt etwas beitragen kannst. Du solltest auch wissen, wie du ein Projekt verwaltet. Du musst bspw. wissen wie du Patches akzeptierst und anwendest, die über `format-patch` generiert und per E-Mail an dich gesendet wurden. Weiterhin solltest du wissen wie du Änderungen in Remote-Banches für Repositorys integrierst, die du als Remotes zu deinem Projekt hinzugefügt hast. Unabhängig davon, ob du ein zentrales Repository verwalten oder durch Überprüfen oder Genehmigen von Patches helfen möchtest, musst du wissen, wie du die Arbeit Anderer so akzeptierst, dass es für andere Mitwirkende transparent und auf lange Sicht auch nachhaltig ist.

### Arbeiten in Featurebranches

Wenn man vorhat, neuen Code zu integrieren, ist es im Allgemeinen eine gute Idee, diesen in einem *Feature Branch* zu testen. Das ist ein temporärer Branch, der speziell zum Ausprobieren dieser neuen Änderungen erstellt wurde. Auf diese Weise ist es einfach, einen Patch einzeln zu optimieren und ihn nicht weiter zu bearbeiten, wenn er nicht funktioniert, bis du wieder Zeit hast, dich wieder damit zu befassen. Du solltest einen einfachen Branchnamen erstellen, der auf dem Thema der Arbeit basiert, die du durchführst, wie z.B. `ruby_client` oder etwas ähnlich Aussagekräftiges. Dann kannst du dich später leichter daran erinnern, falls du den Branch für eine Weile hast ruhen lassen und später daran weiter arbeitest. Der Betreuer des Git-Projekts neigt auch dazu, diese Branches mit einem Namespace zu versehen – wie z. B. `sc/ruby_client`, wobei `sc` für die Person steht, die die Arbeit beigesteuert hat. Wie du dich erinnerst, kannst du den Branch basierend auf deinem `master` Branch wie folgt erstellen:

```
$ git branch sc/ruby_client master
```

Wenn du anschließend sofort zum neuen Branch wechseln möchtest, kannst du auch die Option `checkout -b` verwenden:

```
$ git checkout -b sc/ruby_client master
```

Jetzt kannst du die getätigte Arbeit zu diesem Branch hinzufügen und festlegen, ob du ihn mit deinen bestehenden Branches zusammenführen möchtest.

### Integrieren von Änderungen aus E-Mails

Wenn du einen Patch per E-Mail erhältst, den du in deinem Projekt integrieren möchtest, musst du den Patch in deinem Feature Branch einfließen lassen, damit du ihn prüfen kannst. Es gibt zwei Möglichkeiten, einen per E-Mail versendeten Patch anzuwenden: mit `git apply` oder mit `git am`.

## Änderungen mit `apply` integrieren

Wenn du den Patch von jemandem erhalten hast, der ihn mit `git diff` oder mit einer Variante des Unix-Befehls `diff` erzeugt hat (was nicht empfohlen wird; siehe nächster Abschnitt), kannst du ihn mit dem Befehl `git apply` integrieren. Angenommen du hast den Patch unter `/tmp/patch-ruby-client.patch` gespeichert. Dann kannst du den Patch folgendermaßen integrieren:

```
$ git apply /tmp/patch-ruby-client.patch
```

Hierdurch werden die Dateien in deinem Arbeitsverzeichnis geändert. Es ist fast identisch mit dem Ausführen eines `patch -p1` Befehls zum Anwenden des Patches, obwohl es vorsichtiger ist und unscharfe Übereinstimmungen selektiver als `patch` akzeptiert. Damit kann man auch Dateien Hinzufügen, Löschen und Umbenennen, wenn diese im `git diff` Format beschrieben sind, was mit `patch` nicht möglich ist. Zu guter Letzt ist `git apply` ein „wende alles oder nichts an“ Modell, bei dem entweder alles oder nichts übernommen wird. `patch` hingegen integriert Patchdateien eventuell nur teilweise und kann dein Arbeitsverzeichnis in einem undefinierten Zustand versetzen. `git apply` ist insgesamt sehr viel konservativer als `patch`. Es wird kein Commit erstellen. Nach dem Ausführen musst du die eingeführten Änderungen manuell bereitstellen und comitten.

Du kannst `git apply` verwenden, um zu prüfen, ob ein Patch ordnungsgemäß integriert werden kann, bevor du versuchst, ihn tatsächlich anzuwenden. Du kannst `git apply --check` auf den Patch ausführen:

```
$ git apply --check 0001-see-if-this-helps-the-gem.patch
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
```

Wenn keine Ausgabe erfolgt, sollte der Patch ordnungsgemäß angewendet werden können. Dieser Befehl wird auch mit einem Rückgabewert ungleich Null beendet, wenn die Prüfung fehlschlägt. So kannst du ihn bei Bedarf in Skripten verwenden.

## Änderungen mit `am` integrieren

Wenn der Beitragende ein Git-Benutzer ist und den Befehl `format-patch` zum Generieren seines Patches verwendet hat, ist deine Arbeit einfacher. Der Patch enthält bereits Informationen über den Autor und eine entsprechende Commitnachricht. Wenn möglich, ermutige die Beitragenden `format-patch` anstelle von `diff` zum Erstellen von Patches zu verwenden. Du solltest `git apply` nur für ältere Patches und ähnliche Dinge verwenden.

Um einen von `format-patch` erzeugten Patch zu integrieren, verwende `git am` (der Befehl heißt `am`, da er verwendet wird, um „eine Reihe von Patches aus einer Mailbox anzuwenden“). Technisch gesehen ist `git am` so aufgebaut, dass eine mbox-Datei gelesen werden kann. Hierbei handelt es sich um ein einfaches Nur-Text-Format zum Speichern einer oder mehrerer E-Mail-Nachrichten in einer Textdatei. Das sieht in etwa so aus:

```
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

Dies ist der Anfang der Ausgabe des Befehls `git format-patch`, den du im vorherigen Abschnitt gesehen hast. Es zeigt ein gültiges mbox Email Format. Wenn dir jemand den Patch ordnungsgemäß mit `git send-email` per E-Mail zugesendet hat und du ihn in ein mbox-Format herunterlädst, kannst du `git am` auf diese mbox-Datei ausführen. Damit werden alle angezeigten Patches entsprechend angewendet. Wenn du einen Mail-Client ausführst, der mehrere E-Mails im Mbox-Format speichern kann, kannst du ganze Patch-Serien in einer Datei speichern. Diese können anschließend mit `git am` einzeln angewendet werden.

Wenn jemand eine mit `git format-patch` erzeugte Patch-Datei in ein Ticketsystem oder ähnliches hochgeladen hat, kannst du die Datei lokal speichern. Die Datei kannst du dann an `git am` übergeben, um sie zu integrieren:

```
$ git am 0001-limit-log-function.patch
Applying: Add limit to log function
```

Wie du sehen kannst, wurde der Patch korrekt integriert und es wurde automatisch ein neuer Commit für dich erstellt. Die Autoreninformationen werden aus den Kopfzeilen `From` und `Date` der E-Mail entnommen und die Commitnachricht wird aus dem `Subject` und dem Textkörper (vor dem Patch) der E-Mail entnommen. Wenn dieser Patch bspw. aus dem obigen mbox-Beispiel angewendet würde, würde der erzeugte Commit in etwa so aussehen:

```
$ 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

Die `Commit` Informationen gibt die Person an, die den Patch angewendet hat und den Zeitpunkt, wann er angewendet wurde. Die `Author` Information gibt die Person an, die den Patch ursprünglich erstellt hat und wann er ursprünglich erstellt wurde.

Es besteht jedoch die Möglichkeit, dass der Patch nicht sauber angewendet werden kann. Möglicherweise ist dein Hauptbranch zu weit vom Branch entfernt, von dem aus der Patch erstellt wurde. Oder aber der Patch hängt noch von einem anderen Patch ab, den du noch nicht angewendet hast. In diesem Fall schlägt der Prozess `git am` fehl und du wirst gefragt, was du tun möchtest:

```
$ git am 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

Dieser Befehl fügt Konfliktmarkierungen in alle Dateien ein, in denen Probleme auftreten. Ähnlich wie bei einem Konflikt bei der Zusammenführung (engl. merge) bzw. bei der Reorganisation (engl. rebase). Du löst dieses Problem auf die gleiche Art: Bearbeite die Datei, um den Konflikt zu lösen. Anschließend füge die neue Datei der Staging-Area hinzu und führe dann `git am --resolved` aus, um mit dem nächsten Patch fortzufahren:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: See if this helps the gem
```

Wenn du möchtest, dass Git den Konflikt etwas intelligenter löst, kannst du ihm die Option „-3“ übergeben, wodurch Git versucht, eine Drei-Wege-Merge durchzuführen. Diese Option ist standardmäßig nicht aktiviert, da sie nicht funktioniert, wenn sich der Commit, auf dem der Patch basiert, nicht in deinem Repository befindet. Wenn du diesen Commit hast – wenn der Patch auf einem öffentlichen Commit basiert –, ist die Option „-3“ im Allgemeinen viel intelligenter beim Anwenden eines Patch mit Konflikten:

```
$ git am -3 0001-see-if-this-helps-the-gem.patch
Applying: See if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

In diesem Fall wäre der Patch ohne die Option `-3` als Konflikt gewertet worden. Da die Option `-3` verwendet wurde, konnte der Patch sauber angewendet werden.

Wenn du mehrere Patches aus mbox anwendest, kannst du auch den Befehl `am` im interaktiven Modus ausführen. Bei jedem gefundenen Patch wird angehalten und du wirst gefragt, ob du ihn anwenden möchtest:

```
$ git am -3 -i mbox
Commit Body is:
-----
See if this helps the gem
```

```
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Dies ist hilfreich, wenn du eine Reihe von Patches gespeichert hast. Du kannst dir den Patch anzeigen lassen, wenn du dich nicht daran erinnerst, worum es genau geht. Oder du wendest den Patch nicht an, weil du es bereits getan hast.

Wenn alle Patches für dein Feature angewendet und in deinem Branch committet wurden, kannst du auswählen, ob und wie du sie in einen Hauptbranch integrieren möchtest.

## Remote Branches auschecken

Wenn du einen Beitrag von einem Git-Nutzer erhältst, der sein eigenes Repository eingerichtet, eine Reihe von Änderungen vorgenommen und dir dann die URL zum Repository und den Namen des Remote-Branchs gesendet hat, in dem sich die Änderungen befinden, dann kannst du diesen als remote hinzufügen und die Änderungen lokal zusammenführen.

Wenn Jessica dir bspw. eine E-Mail sendet, die besagt, dass sie eine großartige neue Funktion im `ruby-client` Branch ihres Repositorys hat, kannst du diese testen, indem du den Branch als remote hinzufügst und ihn lokal auscheckst:

```
$ git remote add jessica https://github.com/jessica/myproject.git  
$ git fetch jessica  
$ git checkout -b rubyclient jessica/ruby-client
```

Wenn Jessica dir später erneut eine E-Mail mit einem anderen Branch sendet, der eine weitere großartige Funktion enthält, kannst du diese direkt abrufen und auschecken, da du bereits über das Remote Repository verfügst.

Dies ist vor allem dann nützlich, wenn du durchgängig mit einer Person arbeitest. Wenn jemand nur selten einen Patch empfängt, ist das Akzeptieren über E-Mail möglicherweise weniger zeitaufwendig. Andernfalls müsste jeder seinen eigenen Server unterhalten und Remotes hinzufügen und entfernen, um diese wenige Patches zu erhalten. Es ist auch unwahrscheinlich, dass du Hunderte von Remotes einbinden möchtest für Personen, die nur ein oder zwei Patches beisteuern. Skripte und gehostete Dienste können dies jedoch vereinfachen – dies hängt weitgehend davon ab, wie du und die Mitwirkenden entwickeln.

Der andere Vorteil dieses Ansatzes ist, dass du auch die Historie der Commits erhältst. Obwohl du möglicherweise Probleme bei der Zusammenführungen hast, wissen du, worauf in deiner Historie deren Arbeit basiert. Eine ordnungsgemäße Drei-Wege-Zusammenführung ist die Standardeinstellung, anstatt ein „-3“ einzugeben, und zu hoffen, dass der Patch aus einem öffentlichen Commit generiert wurde, auf den du Zugriff hast.

Wenn du nicht durchgängig mit einer Person arbeitest, aber dennoch auf diese Weise von dieser Person abrufen möchtest, kannst du die URL des Remote-Repositorys für den Befehl `git pull` angeben. Dies führt einen einmaligen Abruf durch und speichert die URL nicht als Remote-Referenz:

```
$ git pull https://github.com/onetimeguy/project
From https://github.com/onetimeguy/project
 * branch           HEAD      -> FETCH_HEAD
Merge made by the 'recursive' strategy.
```

## Bestimmen, was übernommen wird

Stell dir vor, du hast einen Feature Branch mit neuen Beiträgen. An dieser Stelle kannst du festlegen, was du damit machen möchtest. In diesem Abschnitt werden einige Befehle noch einmal behandelt. Mit diesen kannst du genau überprüfen, was du übernimmst, wenn du die Beiträge in deinem Hauptbranch zusammenführst.

Es ist oft hilfreich, eine Überprüfung über alle Commits zu erhalten, die sich in diesem Branch jedoch nicht in deinem `master` Branch befinden. Du kannst Commits im `master` Branch ausschließen, indem du die Option `--not` vor dem Branchnamen hinzufügst. Dies entspricht dem Format `master..contrib`, welches wir zuvor verwendet haben. Wenn deine Mitstreiter dir bspw. zwei Patches senden und du einen Branch mit dem Namen `contrib` erstellst und diese Patches dort anwendest, kannst du Folgendes ausführen:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700
```

See if this helps the gem

```
commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700
```

Update gemspec to hopefully work better

Denke daran, dass du die Option `-p` an `git log` übergeben kannst, um zu sehen, welche Änderungen jeder Commit einführt.

Um einen vollständigen Überblick darüber zu erhalten, was passieren würde, wenn du diesen Branch mit einem anderen Branch zusammenführen würdest, musst du möglicherweise einen ungewöhnlichen Kniff anwenden, um die richtigen Ergebnisse zu erzielen. Eventuell denkst du daran folgendes auszuführen:

```
$ git diff master
```

Dieser Befehl gibt die Unterschiede zurück. Dies kann jedoch irreführend sein. Wenn dein Masterbranch vorgerückt ist, seit du den Feature-Branch daraus erstellt hast, erhältst du scheinbar unerwartete Ergebnisse. Dies geschieht, weil Git den Snapshots des letzten Commits des Branches, in dem du dich befindest, und den Snapshot des letzten Commits des Branches `master` direkt

miteinander vergleicht. Wenn du bspw. eine Zeile in eine Datei im Branch `master` eingefügt hast, sieht ein direkter Vergleich des Snapshots so aus, als würde der Feature Branch diese Zeile entfernen.

Wenn `master` ein direkter Vorgänger deines Feature-Branches ist, ist dies kein Problem. Wenn aber beiden Historien voneinander abweichen, sieht es so aus, als würdest du alle neuen Inhalte in deinem Featurebranch hinzufügen und alles entfernen, was im `master` Branch eindeutig ist.

Was du wirklich sehen möchtest, sind die Änderungen, die dem Featurebranch hinzugefügt wurden. Die Arbeit, die du hinzufügst, wenn du den neuen Branch mit `master` zusammenführst. Du tust dies, indem Git den letzten Commit in deinem Featurebranch mit dem ersten gemeinsamen Vorgänger aus dem `master` Branch vergleicht.

Technisch gesehen könntest du dies tun, indem du den gemeinsamen Vorgänger explizit herausfindest und dann dein `diff` darauf ausführst:

```
$ git merge-base contrib master  
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649  
$ git diff 36c7db
```

oder kurzgefasst:

```
$ git diff $(git merge-base contrib master)
```

Beides ist jedoch nicht besonders praktisch, weshalb Git einen anderen Weg für diesen Vorgang bietet: die Drei-Punkt-Syntax (engl. Triple-Dot-Syntax). Im Kontext des Befehls `git diff` kannst du drei Punkte nach einem anderen Branch einfügen, um ein `diff` zwischen dem letzten Commit deines aktuellen Branch und dem gemeinsamen Vorgänger eines anderen Branches zu erstellen:

```
$ git diff master...contrib
```

Dieser Befehl zeigt dir nur die Arbeit an, die dein aktueller Branch seit dem gemeinsamen Vorgänger mit `master` eingeführt hat. Dies ist eine sehr nützliche Syntax, die du dir merken solltest.

## Beiträge integrieren

Wenn dein Featurebranch bereit ist, um in einen Hauptbranch integriert zu werden, lautet die Frage, wie du dies tun kannst. Welchen Workflow möchtest du verwenden, um dein Projekt zu verwalten? Du hast eine Reihe von Möglichkeiten, daher werden wir einige davon behandeln.

### Zusammenführungs Workflow (engl. mergen)

Ein grundlegender Workflow besteht darin, all diese Arbeiten einfach direkt in deinem `master` Branch zusammenzuführen. In diesem Szenario hast du einen `master` Branch, der stabilen Code enthält. Wenn du in einem Branch arbeitest, von dem du glaubst, dass du ihn abgeschlossen hast, oder von jemand anderem beigesteuert und überprüft hast, führst du ihn in deinem Hauptbranch

zusammen. Lösche anschließend diesen gerade zusammengeführten Branch und wiederholen den Vorgang bei Bedarf.

Wenn wir zum Beispiel ein Repository mit zwei Branches namens `ruby_client` und `php_client` haben, die wie [Historie mit mehreren Topic Branches](#) aussehen, und wir `ruby_client` gefolgt von `php_client` zusammenführen, sieht dein Verlauf so aus [Status nach einem Featurebranch Merge](#).

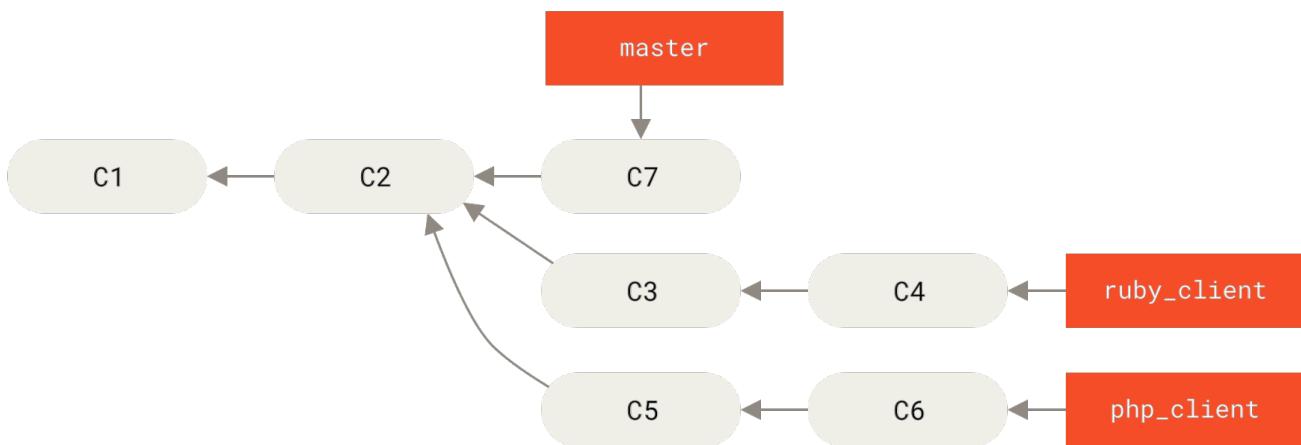


Figure 72. Historie mit mehreren Topic Branches

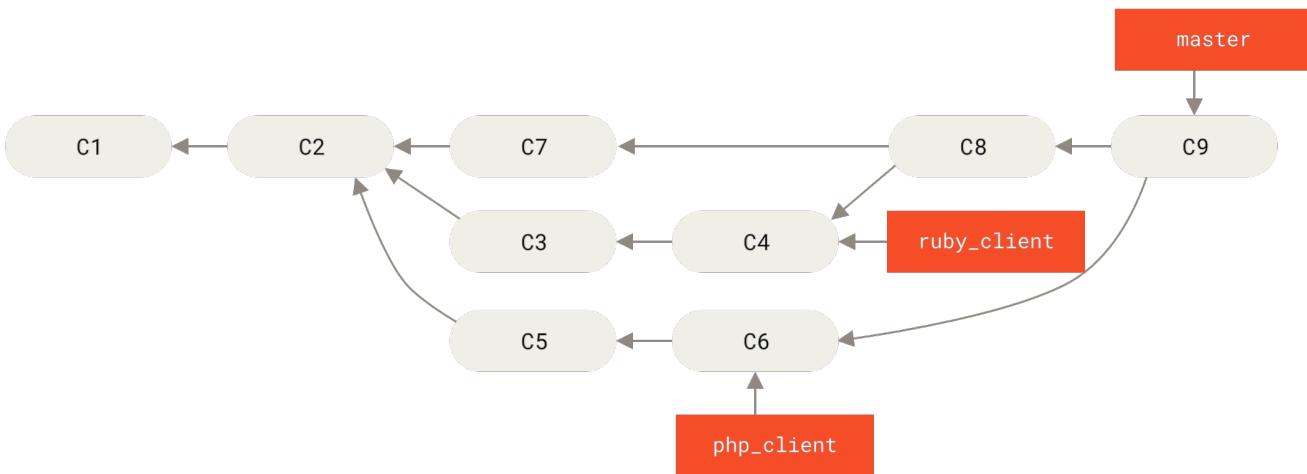


Figure 73. Status nach einem Featurebranch Merge

Das ist wahrscheinlich der einfachste Workflow. Es kann jedoch zu Problemen kommen, wenn du größere oder stabilere Projekte bearbeitest. Bei diesen musst du mit der Einführung von Änderungen sehr vorsichtig sein.

Wenn du ein wichtiges Projekt hast, möchtest du möglicherweise einen zweistufigen Merge Prozess verwenden. In diesem Szenario hast du zwei lange laufende Branches namens `master` und `develop`. Du legst fest, dass `master` nur dann aktualisiert wird, wenn eine sehr stabile Version vorhanden ist und der gesamte neue Code in den Branch `develop` integriert wird. Du pushst diese beiden Branches regelmäßig in das öffentliche Repository. Jedes Mal, wenn du einen neuen Branch zum Zusammenführen hast ([Vor einem Featurebranch Merge](#)), führst du ihn in `develop` ([Nach einem Featurebranch Merge](#)) zusammen. Wenn du nun ein Release mit einem Tag versiehst, spulst du `master` an dieser Stelle weiter, an der sich der jetzt stabile `develop` Branch befindet ([Nach einem Projekt Release](#)).

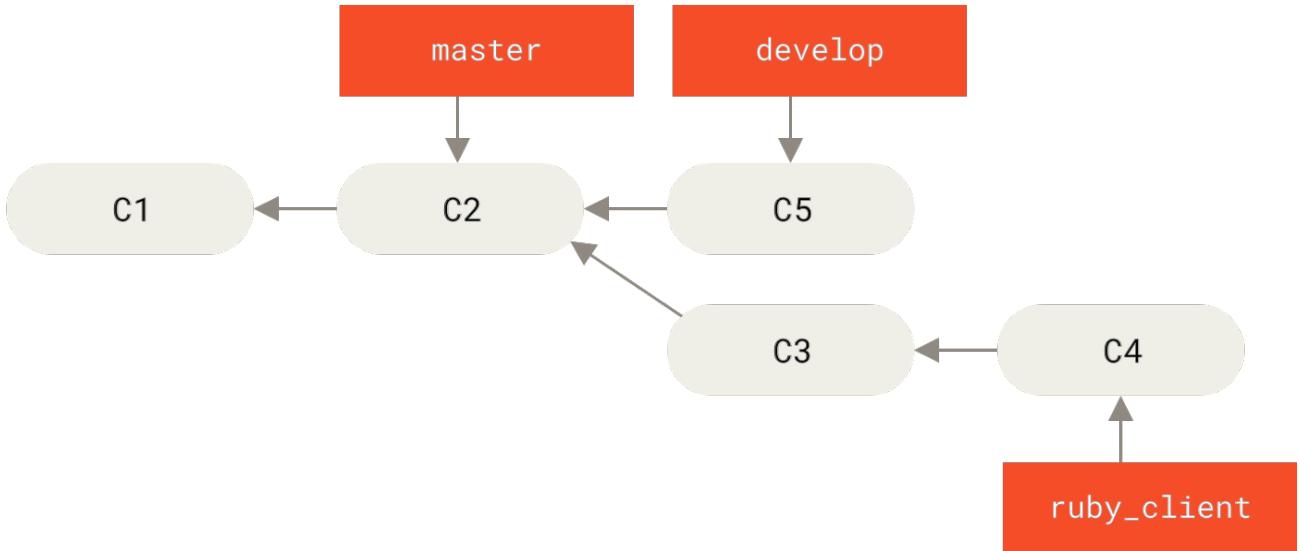


Figure 74. Vor einem Featurebranch Merge

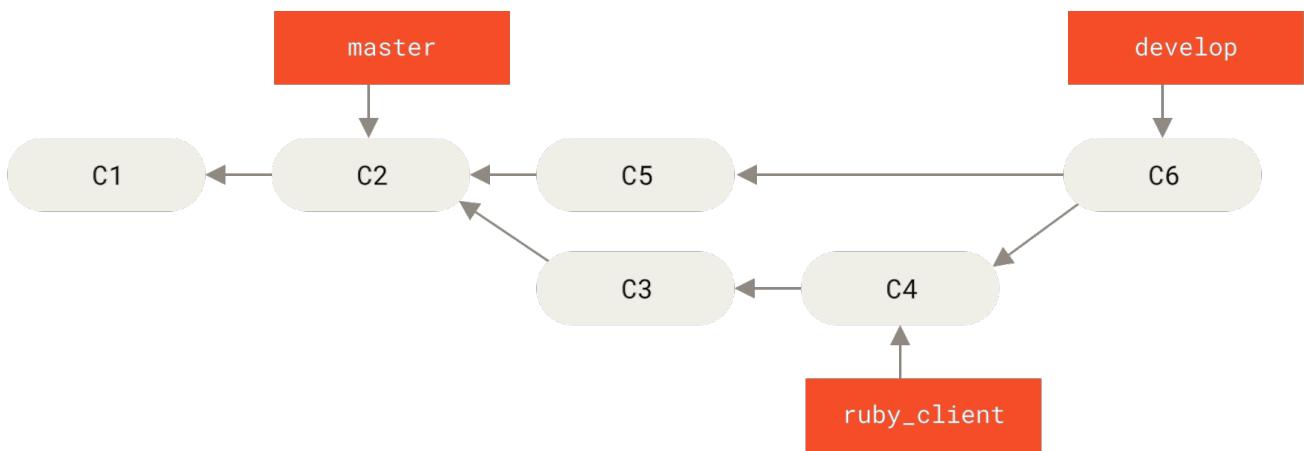


Figure 75. Nach einem Featurebranch Merge

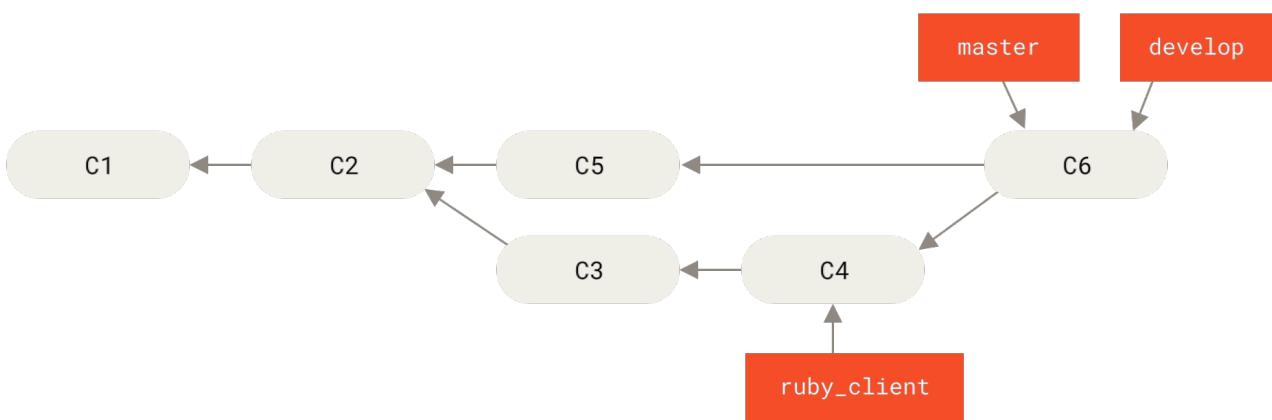


Figure 76. Nach einem Projekt Release

Auf diese Weise können Benutzer, die das Repository deines Projekts klonen, entweder `master` oder `develop` auschecken. Mit `master` können sie die neueste stabile Version erstellen und somit recht einfach auf dem neuesten Stand bleiben. Oder sie können `develop` auschecken, welchen den aktuellsten Inhalt enthält. Du kannst dieses Konzept auch erweitern, indem du einen `integrate` Branch einrichtest, in dem alle Arbeiten zusammengeführt werden. Wenn die Codebasis auf diesem

Branch stabil ist und die Tests erfolgreich sind, kannst du sie zu einem Entwicklungsbranch zusammen führen. Wenn sich dieser dann als stabil erwiesen hat, kannst du deinen `master` Branch fast-forwarden.

## Workflows mit umfangreichen Merges

Das Git-Projekt selber hat vier kontinuierlich laufende Branches: `master`, `next` und `seen` (vormals 'pu' – vorgeschlagene Updates) für neue Arbeiten und `maint` für Wartungs-Backports. Wenn neue Arbeiten von Mitwirkenden eingereicht werden, werden sie in ähnlicher Weise wie oben beschrieben in Featurebranches im Projektrepository des Betreuers gesammelt (siehe [Verwaltung einer komplexen Reihe paralleler Featurebranches](#)). Zu diesem Zeitpunkt werden die Features evaluiert, um festzustellen, ob sie korrekt sind und zur Weiterverarbeitung bereit sind oder ob sie Nacharbeit benötigen. Wenn sie korrekt sind, werden sie zu `next` zusammengeführt, und dieser Branch wird dann gepusht, damit jeder die miteinander integrierten Features testen kann.

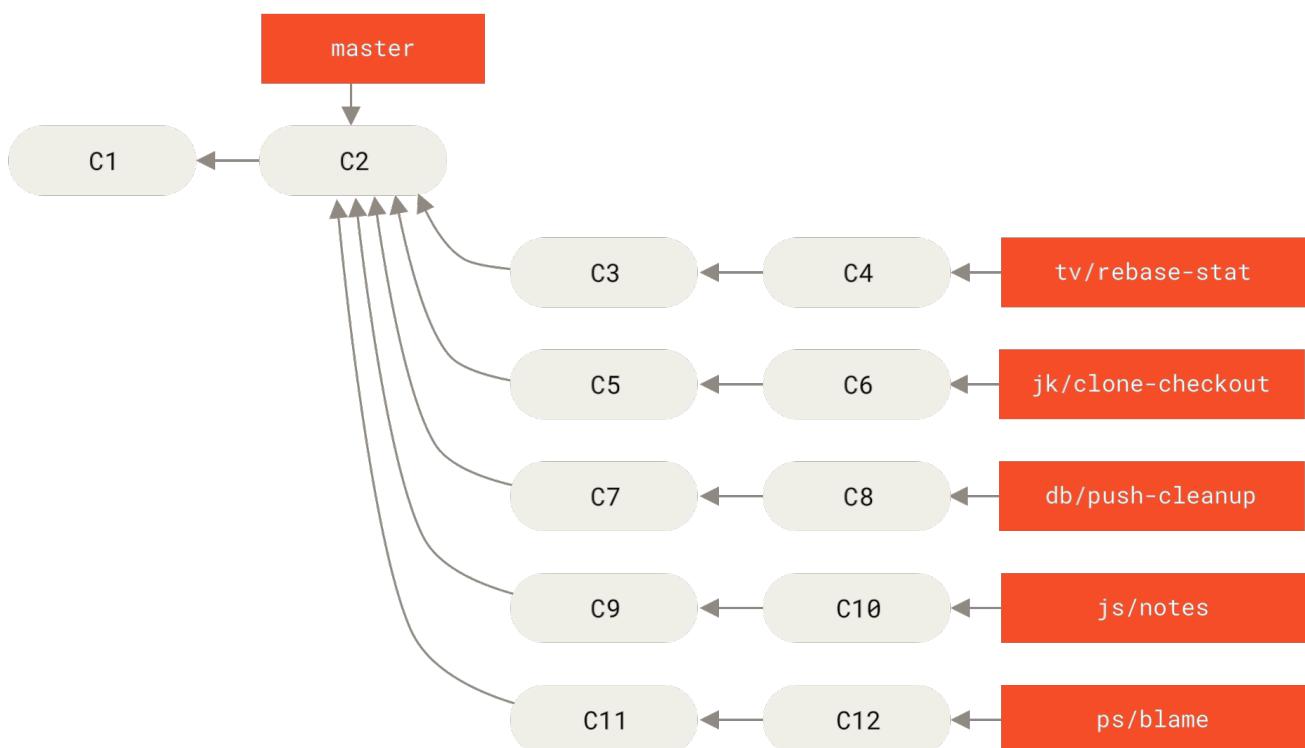


Figure 77. Verwaltung einer komplexen Reihe paralleler Featurebranches

Wenn die Features noch bearbeitet werden müssen, werden sie in `seen` gemerged. Wenn festgestellt wird, dass sie absolut stabil sind, werden die Features wieder zu `master` zusammengeführt. Die Branches `next` und `seen` werden dann von `master` neu aufgebaut. Dies bedeutet, dass `master` fast immer vorwärts geht, `next` wird gelegentlich und `seen` häufiger rebased:

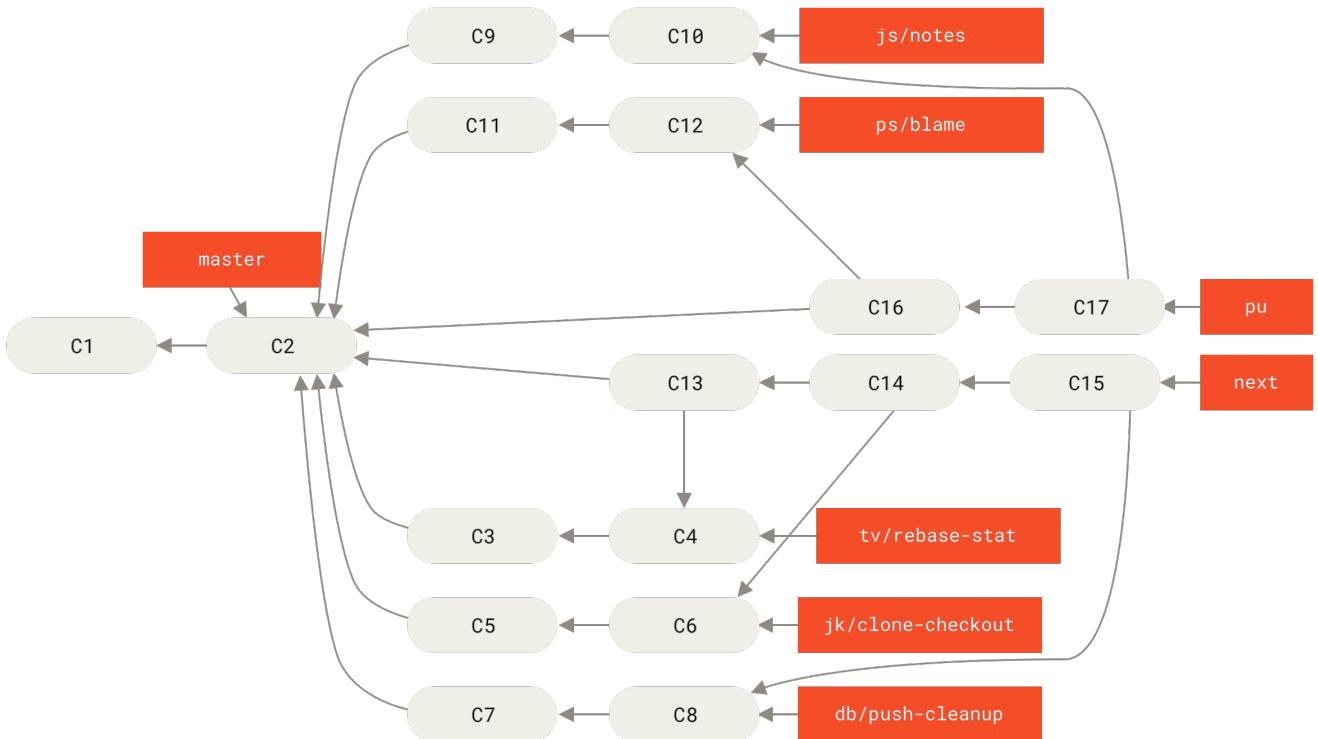


Figure 78. Zusammenführen von Featurebranches in langfristige Integrationsbranches

Wenn ein Branch schließlich zu `master` zusammengeführt wurde, wird er aus dem Repository entfernt. Das Git-Projekt hat auch einen `maint` Branch, der von letzten release geforkt wurde, um für den Fall, dass eine Wartungsversion erforderlich ist, Backport-Patches bereitzustellen. Wenn du das Git-Repository also klonst, stehen dir vier Branches zur Verfügung, mit denen du das Projekt in verschiedenen Entwicklungsstadien bewerten kannst, je nachdem, wie aktuell du sein möchtest oder wie du einen Beitrag leisten möchtest. Der Betreuer verfügt über einen strukturierten Workflow, der ihm hilft, neue Beiträge zu überprüfen. Der Workflow des Git-Projekts ist sehr speziell. Um dies zu verstehen, kannst du das [Git Maintainer's guide](#) lesen.

## Rebasing und Cherry-Picking Workflows

Andere Betreuer bevorzugen es, die Arbeit auf ihrem `master` Branch zu rebasen oder zu cherry-picken, anstatt sie zusammenzuführen, um einen linearen Verlauf beizubehalten. Wenn du in einem Featureranch arbeitest und dich dazu entschlossen hast, ihn zu integrieren, wechselst du in diesen Branch und führst den `rebase` Befehl aus, um die Änderungen auf deinen `master` (oder `develop` Branch usw.) aufzubauen. Wenn das gut funktioniert, kannst du deinen `master` Branch fast-forwarden, und du erhältst eine lineare Projekthistorie.

Eine andere Möglichkeit, die eingeführte Arbeit von einem Branch in einen anderen zu verschieben, besteht darin, sie zu cherry-picken. Ein Cherry-Pick in Git ist wie ein Rebase für ein einzelnes Commit. Es nimmt den Patch, der in einem Commit eingeführt wurde, und versucht ihn erneut auf den Branch anzuwenden, auf dem du dich gerade befindest. Dies ist nützlich, wenn du eine Reihe von Commits für einen Branch hast und nur einen davon integrieren möchtest. Oder aber wenn du nur einen Commit für einen Branch hast und es vorziebst, diesen zu cherry-picken, anstatt ein Rebase auszuführen. Angenommen, du hast ein Projekt, das folgendermaßen aussieht:

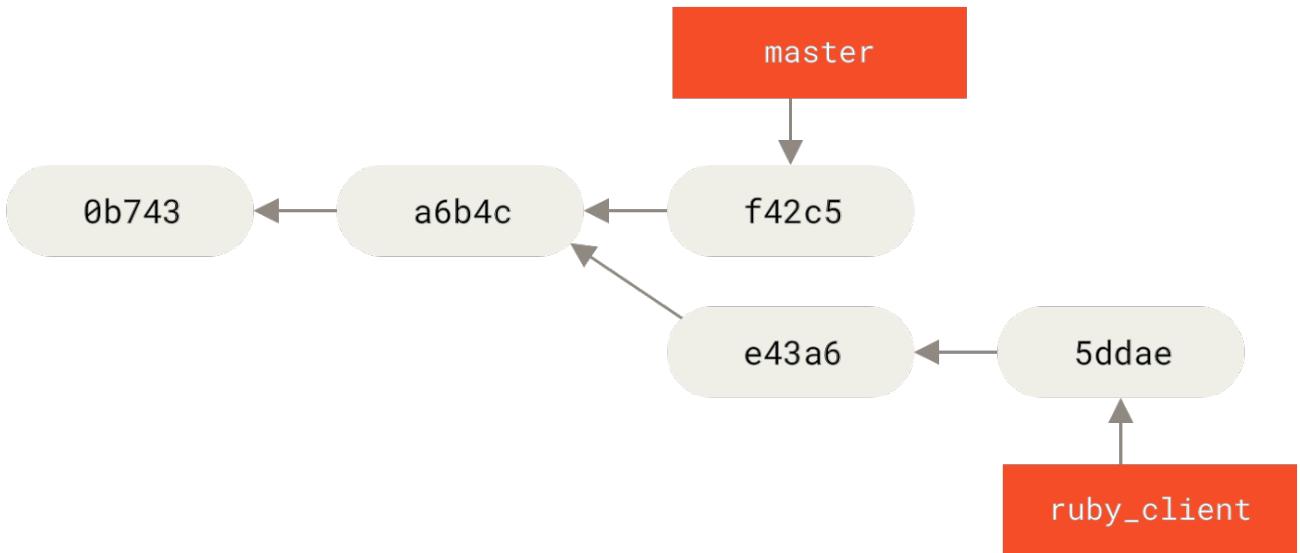


Figure 79. Beispiel Historie vor einem Cherry-Pick

Wenn du den Commit „e43a6“ in deinem `master` Branch ziehen möchtest, kannst du folgendes ausführen:

```
$ 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(-)
```

Dies zieht die gleiche Änderung nach sich, die in `e43a6` eingeführt wurde. Du erhältst jedoch einen neuen Commit SHA-1-Wert, da das angewendete Datum unterschiedlich ist. Nun sieht die Historie so aus:

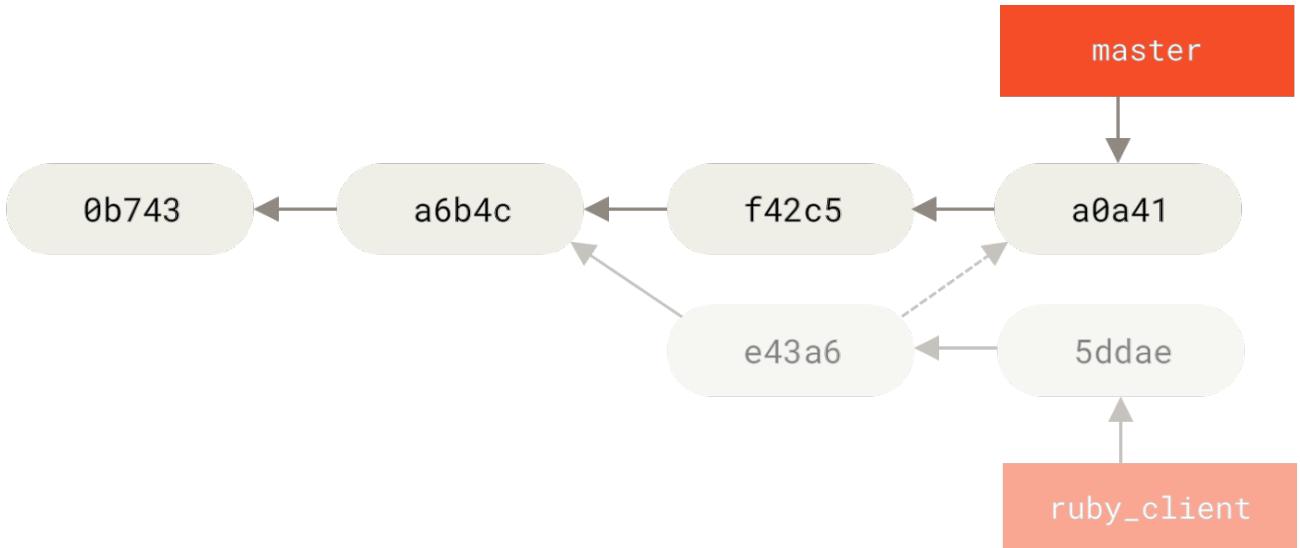


Figure 80. Historie nach Cherry-Picken eines Commits auf einen Featurebranch

Jetzt kannst du deinen Featurebranch entfernen und die Commits löschen, die du nicht einbeziehen willst.

## Rerere

Wenn du viel mergst und rebased oder einen langlebigen Featurebranch pflegst, hat Git eine Funktion namens „rerere“, die nützlich sein kann.

Rerere steht für „reuse recorded resolution“ (deutsch „Aufgezeichnete Lösung wiederverwenden“). Es ist eine Möglichkeit, die manuelle Konfliktlösung zu verkürzen. Wenn rerere aktiviert ist, behält Git eine Reihe von Pre- und Postimages von erfolgreichen Commits bei. Wenn es feststellt, dass ein Konflikt genauso aussieht, wie der, den du bereits behoben hast, wird die Korrektur vom letzten Mal verwendet, ohne nochmal nachzufragen.

Diese Funktion besteht aus zwei Teilen: einer Konfigurationseinstellung und einem Befehl. Die Konfigurationseinstellung lautet `rerere.enabled`. Man kann sie in die globale Konfiguration eingeben:

```
$ git config --global rerere.enabled true
```

Wenn du nun einen merge durchführst, der Konflikte auflöst, wird diese Auflösung im Cache für die Zukunft gespeichert.

Bei Bedarf kannst du mit dem rerere Cache mittels des Befehls `git rerere` interagieren. Wenn der Befehl ausgeführt wird, überprüft Git seine Lösungsdatenbank und versucht eine Übereinstimmung mit aktuellen Mergekonflikten zu finden und diesen zu lösen (dies geschieht jedoch automatisch, wenn `rerere.enabled` auf `true` gesetzt ist). Es gibt auch Subbefehle, um zu sehen, was aufgezeichnet wird, um eine bestimmte Konfliktlösung aus dem Cache zu löschen oder um den gesamten Cache zu löschen. Wir werden uns in [Rerere](#) eingehender mit rerere beschäftigen.

## Tagging deines Releases

Wenn du dich entschieden hast, ein Release zu erstellen, dann möchtest du wahrscheinlich einen Tag zuweisen, damit du dieses Release in Zukunft jederzeit neu erstellen kannst. Du kannst einen neuen Tag erstellen, wie in [Git Grundlagen](#) beschrieben. Wenn du den Tag als Betreuer signieren möchtest, sieht der Tag möglicherweise folgendermaßen aus:

```
$ 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
```

Wenn du deinen Tags signierst, hast du möglicherweise das Problem, den öffentlichen PGP-Schlüssel zu verteilen, der zum Signieren deines Tags verwendet wird. Der Betreuer des Git-Projekts hat dieses Problem behoben, indem er seinen öffentlichen Schlüssel als Blob in das Repository aufgenommen und anschließend einen Tag hinzugefügt hat, der direkt auf diesen Inhalt verweist. Um dies zu tun, kannst du herausfinden, welchen Schlüssel du möchtest, indem du `gpg --list-keys` ausführst:

```
$ 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]
```

Anschließend kannst du den Schlüssel direkt in die Git-Datenbank importieren, indem du ihn exportierst und diesen über `git hash-object` weiterleitest. Dadurch wird ein neuer Blob mit diesen Inhalten in Git geschrieben und du erhältst den SHA-1 des Blobs zurück:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin  
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Nachdem du nun den Inhalt deines Schlüssels in Git hast, kannst du einen Tag erstellen, der direkt darauf verweist. Dies tust du indem du den neuen SHA-1-Wert angibst, den du mit dem Befehl `hash-object` erhalten hast:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Wenn du `git push --tags` ausführst, wird der Tag `maintainer-pgp-pub` für alle freigegeben. Wenn jemand einen Tag verifizieren möchte, kann er deinen PGP-Schlüssel direkt importieren, indem er den Blob direkt aus der Datenbank zieht und in GPG importiert:

```
$ git show maintainer-pgp-pub | gpg --import
```

Mit diesem Schlüssel können alle deine signierten Tags überprüfen. Wenn du der Tag-Nachricht Anweisungen hinzufügst, kannst du dem Endbenutzer mit `git show <tag>` genauere Anweisungen zur Tag-Überprüfung geben.

## Eine Build Nummer generieren

Git verfügt nicht über ansteigende Zahlen wie 'v123' oder ähnliches für jeden Commit. Wenn du einen lesbaren Namen für deinen Commit benötigst, dann kannst du für diesen Commit den Befehl `git describe` ausführen. Als Antwort generiert Git eine Zeichenfolge, die aus dem Namen des jüngsten Tags vor diesem Commit besteht, gefolgt von der Anzahl der Commits seit diesem Tag, gefolgt von einem partiellen SHA-1-Wert des beschriebene Commits (vorangestelltem wird dem Buchstaben „g“ für Git):

```
$ git describe master  
v1.6.2-rc1-20-g8c5b85c
```

Auf diese Weise kannst du einen Snapshot exportieren oder einen Build erstellen und einen verständlichen Namen vergeben. Wenn du Git aus den Quellcode erstellst, der aus dem Git-

Repository geklont wurde, erhältst du mit `git --version` etwas, das genauso aussieht. Wenn du einen Commit beschreibst, den du direkt getaggt hast, erhältst du einfach den Tag-Namen.

Standardmäßig erfordert der Befehl `git describe` annotierte (mit Anmerkungen versehene) Tags (die mit dem Flag `-a` oder `-s` erstellt wurden). Wenn du auch leichtgewichtige (nicht mit Anmerkungen versehene) Tags verwenden möchtest, fügst du dem Befehl die Option `--tags` hinzu. Du kannst diese Zeichenfolge auch als Ziel der Befehle `git checkout` oder `git show` verwenden, obwohl sie auf dem abgekürzten SHA-1-Wert am Ende basiert, sodass sie möglicherweise nicht für immer gültig ist. Zum Beispiel hat der Linux-Kernel kürzlich einen Sprung von 8 auf 10 Zeichen gemacht, um die Eindeutigkeit von SHA-1-Objekten zu gewährleisten, sodass ältere Ausgabenamen von `git describe` ungültig wurden.

## Ein Release vorbereiten

Als nächstes möchtest du einen Build freigeben. Eines der Dinge, die du tun möchtest, ist ein Archiv des neuesten Schnappschusses deines Codes für die Unglücklichen zu erstellen, die Git nicht verwenden. Der Befehl dazu lautet `git archive`:

```
$ git archive master --prefix='project/' | gzip > 'git describe master'.tar.gz  
$ ls *.tar.gz  
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Wenn jemand dieses Archiv öffnet, erhält er den neuesten Schnappschuss deines Projekts in einem `projekt` Verzeichnis. Du kannst auch ein zip-Archiv auf die gleiche Weise erstellen, indem du jedoch die Option `--format=zip` an `git archive` übergibst:

```
$ git archive master --prefix='project/' --format=zip > 'git describe master'.zip
```

Du hast jetzt einen schönen Tarball und ein Zip-Archiv deiner Projektversion, die du auf deiner Website hochladen oder per E-Mail an andere Personen senden kannst.

## Das Shortlog

Es ist Zeit, eine E-Mail an die Personen deiner Mailingliste zu senden, die wissen möchten, was in deinem Projekt vor sich geht. Mit dem Befehl `git shortlog` kannst du schnell eine Art Änderungsprotokoll dessen abrufen, was deinem Projekt seit deiner letzten Veröffentlichung oder deiner letzte E-Mail hinzugefügt wurde. Es fasst alle Commits in dem von dir angegebenen Bereich zusammen. Im Folgenden findest du als Beispiel eine Zusammenfassung aller Commits seit deiner letzten Veröffentlichung, sofern deine letzte Veröffentlichung den Namen v1.0.1 hat:

```
$ git shortlog --no-merges master --not v1.0.1  
Chris Wanstrath (6):  
    Add support for annotated tags to Grit::Tag  
    Add packed-refs annotated tag support.  
    Add Grit::Commit#to_patch  
    Update version and History.txt  
    Remove stray 'puts'
```

```
Make ls_tree ignore nils
```

Tom Preston-Werner (4):

- fix dates in history
- dynamic version method
- Version bump to 1.0.2
- Regenerated gemspec for version 1.0.2

Du erhältst eine übersichtliche Zusammenfassung aller Commits seit Version 1.0.1, gruppiert nach Autoren, die du per E-Mail an deine Mailingliste senden kannst.

## Zusammenfassung

Du solltest nun vertraut damit sein, in einem Git Projekt mitzuwirken, dein eigenes Projekt zu pflegen oder die Beiträge anderer Entwickler zu integrieren. Herzlichen Glückwunsch, du bist nun ein erfolgreichen Git-Entwickler! Im nächsten Kapitel erfährst du, wie du den größten und beliebtesten Git-Hosting-Dienst, GitHub, verwendest.

# GitHub

GitHub ist der größte Hoster für Git-Repositorys und der zentrale Punkt der Teamarbeit für Millionen von Entwicklern und Projekten. Ein großer Teil aller Git-Repositorys wird auf GitHub gehostet und viele Open-Source-Projekte nutzen es für Git-Hosting, Issue Tracking, Code-Review und andere Dinge. Obwohl es kein direkter Bestandteil des Git Open-Source-Projekts ist, ist es sehr wahrscheinlich, dass du irgendwann mit GitHub arbeiten möchtest oder musst.

In diesem Kapitel geht es um die effektive Nutzung von GitHub. Wir behandeln die Anmeldung und Verwaltung eines Kontos, die Erstellung und Nutzung von Git-Repositorys und Workflows für die Zusammenarbeit. Außerdem geht es darum, wie man Zuarbeiten für Projekte annimmt, die Programmoberfläche von GitHub und viele kleine Tipps, um dir das Leben im Allgemeinen zu erleichtern.

Wenn du nicht daran interessiert bist, GitHub zu verwenden, um deine eigenen Projekte zu hosten oder mit anderen an Projekten zusammenzuarbeiten, die auf GitHub gehostet sind, kannst du getrost zum nächsten Kapitel [Git Tools](#) springen.

## *Schnittstellen Änderung*



Bitte beachte, dass sich die Weboberfläche in diesen Screenshots, wie bei vielen aktiven Websites, mit der Zeit ändern können. Hoffentlich wird die generelle Idee von dem was wir hier zu zeigen versuchen, immer noch verständlich sein. Wenn du neuere Versionen der Weboberfläche benötigst, könnte die Online-Versionen diese Buchse aktuellere Screenshots enthalten.

## Einrichten und Konfigurieren eines Kontos

Das erste, was du tun musst, ist ein kostenloses Benutzerkonto einzurichten. Besuche einfach <https://github.com>, wähle einen noch unbenutzten Username, gib eine E-Mail-Adresse und ein Passwort ein und klicke auf die große grüne Schaltfläche „Bei GitHub Anmelden“.

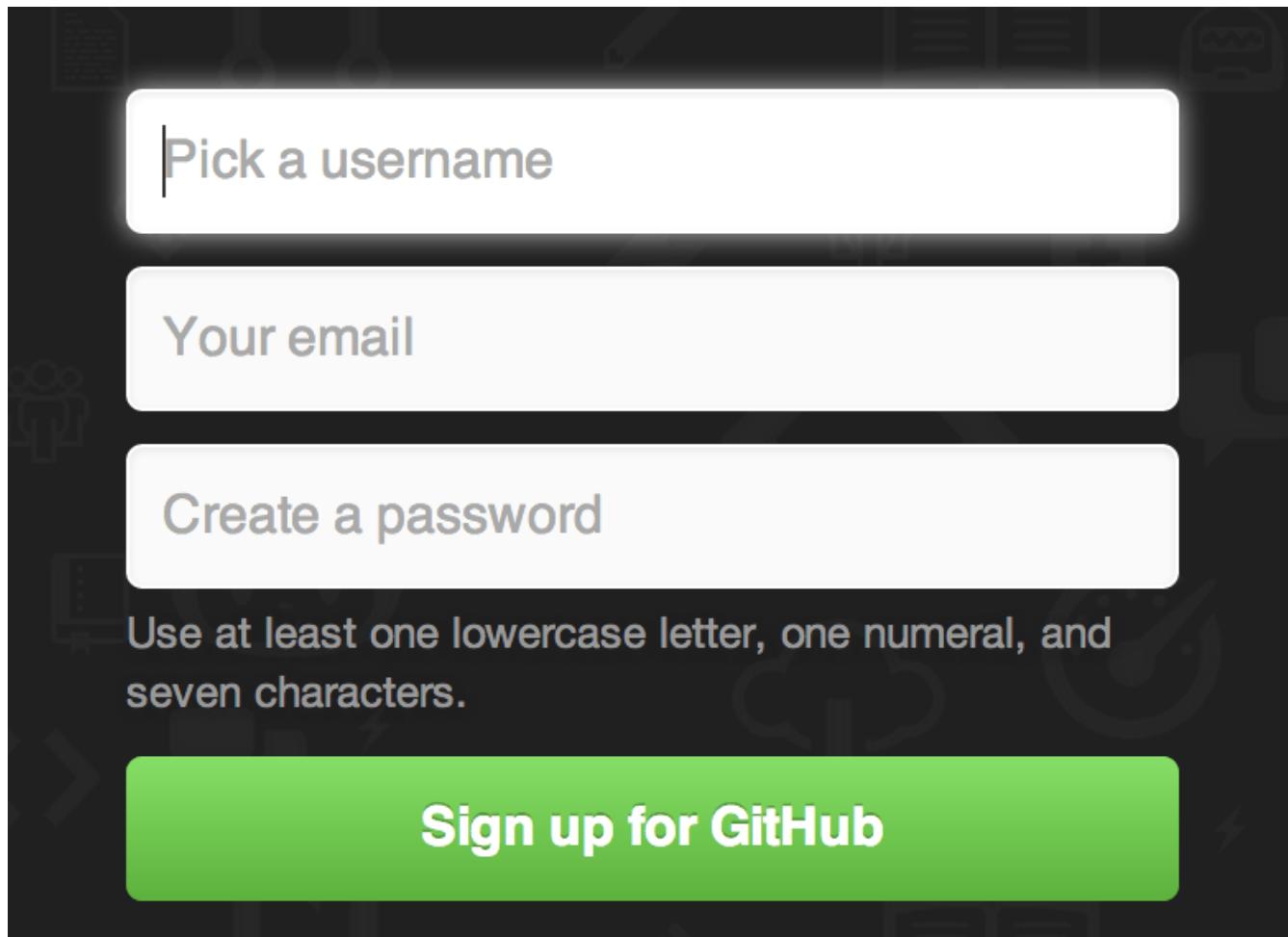


Figure 81. Das GitHub Anmeldeformular

Das nächste was du sehen wirst, ist die Preisseite für Upgrade-Pakete. Du kannst diese jedoch vorerst ignorieren. GitHub sendet dir eine E-Mail, um die von dir angegebene Adresse zu bestätigen. Fahre fort indem du die erhaltene E-Mail bestätigst. Das ist ziemlich wichtig, wie wir später sehen werden.

GitHub bietet fast alle Funktionen kostenlos an, mit Ausnahme einiger erweiterter Funktionen.



Die kostenpflichtigen Tarife von GitHub umfassen erweiterte Tools und Funktionen sowie erhöhte Limits für kostenlose Dienste. Diese werden in diesem Buch jedoch nicht behandelt. Weitere Informationen zu verfügbaren Tarifen und deren Vergleich erhältst du unter <https://de.github.com/pricing.html>.

Wenn du auf das Octocat-Logo oben links auf dem Bildschirm klickst, gelangst du zu deiner Dashboard-Seite. Ab jetzt kannst du GitHub benutzen.

## SSH-Zugang

Ab sofort kannst du dich uneingeschränkt mit Git-Repositorys über das <https://> Protokoll verbinden und dich mit dem gerade eingerichteten Benutzernamen und Passwort authentifizieren. Um jedoch öffentliche Projekte einfach zu klonen, musst du dich nicht einmal anmelden. Das Konto, das wir gerade erstellt haben, benötigen wir nur, wenn wir Projekte forken und später zu unseren Forks etwas pushen.

Wenn du SSH-Remotes verwenden möchtest, musst du einen öffentlichen Schlüssel konfigurieren. Falls du noch keinen hast, siehe [öffentlichen SSH-Schlüssel generieren](#). Öffne deine Kontoeinstellungen über den Link oben rechts im Fenster:

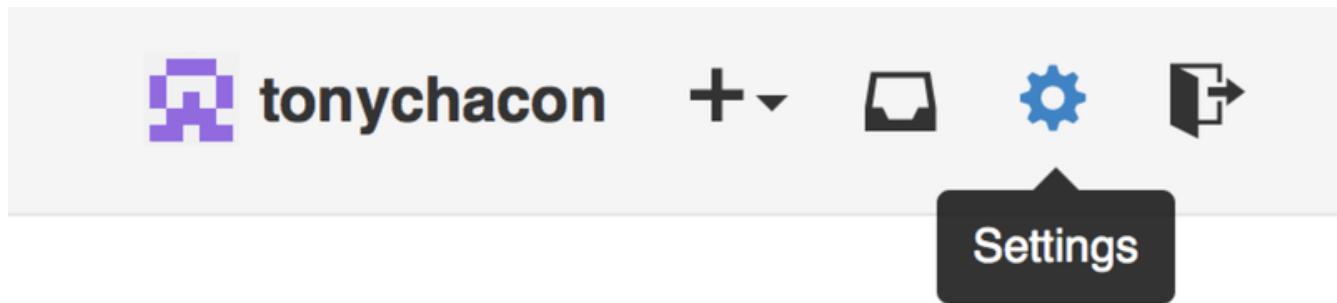


Figure 82. Der Link „Account-Einstellungen“

Wähle dann auf der linken Seite den Bereich „SSH-Schlüssel“.

A screenshot of the "SSH keys" section in the GitHub account settings. On the left, a sidebar lists "Profile", "Account settings", "Emails", "Notification center", "Billing", "SSH keys" (which is selected and highlighted in orange), "Security", "Applications", "Repositories", and "Organizations". The main area has a heading "SSH Keys" with a sub-section "Add an SSH Key". It contains fields for "Title" (an empty input field) and "Key" (a large text area). A green "Add key" button is at the bottom. Above the main area, a message says "Need help? Check out our guide to [generating SSH keys](#) or troubleshoot [common SSH Problems](#)". There's also a "Add SSH key" button in the top right of the "SSH Keys" section.

Figure 83. Der Link „SSH-Schlüssel“

Klicke von dort aus auf die Schaltfläche „Add an SSH key“, gib deinem Schlüssel einen Namen, füge den Inhalt deiner `~/.ssh/id_rsa.pub` Public-Key-Datei (oder wie auch immer du sie genannt hast) in das Textfeld ein und klicke auf „Add key“.



Achte darauf, dass du deinem SSH-Schlüssel einen Namen gibst, an den du dich gut erinnern kannst. Du kannst jeden deiner Schlüssel (z.B. „Mein Laptop“ oder „Arbeitskonto“) benennen, so dass du, falls du einen Schlüssel später widerrufen musst, leichter erkennen kannst, nach welchem du suchst.

## Dein Avatar-Bild

Als nächstes kannst du, wenn du möchtest, den für dich generierten Avatar durch ein Bild deiner Wahl ersetzen. Gehe zunächst auf die Registerkarte „Profil“ (oberhalb der Registerkarte SSH-Schlüssel) und klicke auf „Neues Bild hochladen“.

The screenshot shows the GitHub profile settings interface. On the left, a sidebar lists various account management options: Profile (which is selected and highlighted in orange), Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications, Repositories, and Organizations. The main content area is titled "Public profile". It includes fields for "Profile picture" (with a placeholder image of a purple Git logo and a "Upload new picture" button), "Name" (empty input field), "Email (will be public)" (empty input field), "URL" (empty input field), "Company" (empty input field), and "Location" (empty input field). At the bottom is a green "Update profile" button.

Figure 84. Der Link „Profile“

In diesem Beispiel wählen wir eine Kopie des Git-Logos, das sich auf unserer Festplatte befindet. Anschließend haben wir die Möglichkeit, es zurecht zu schneiden.

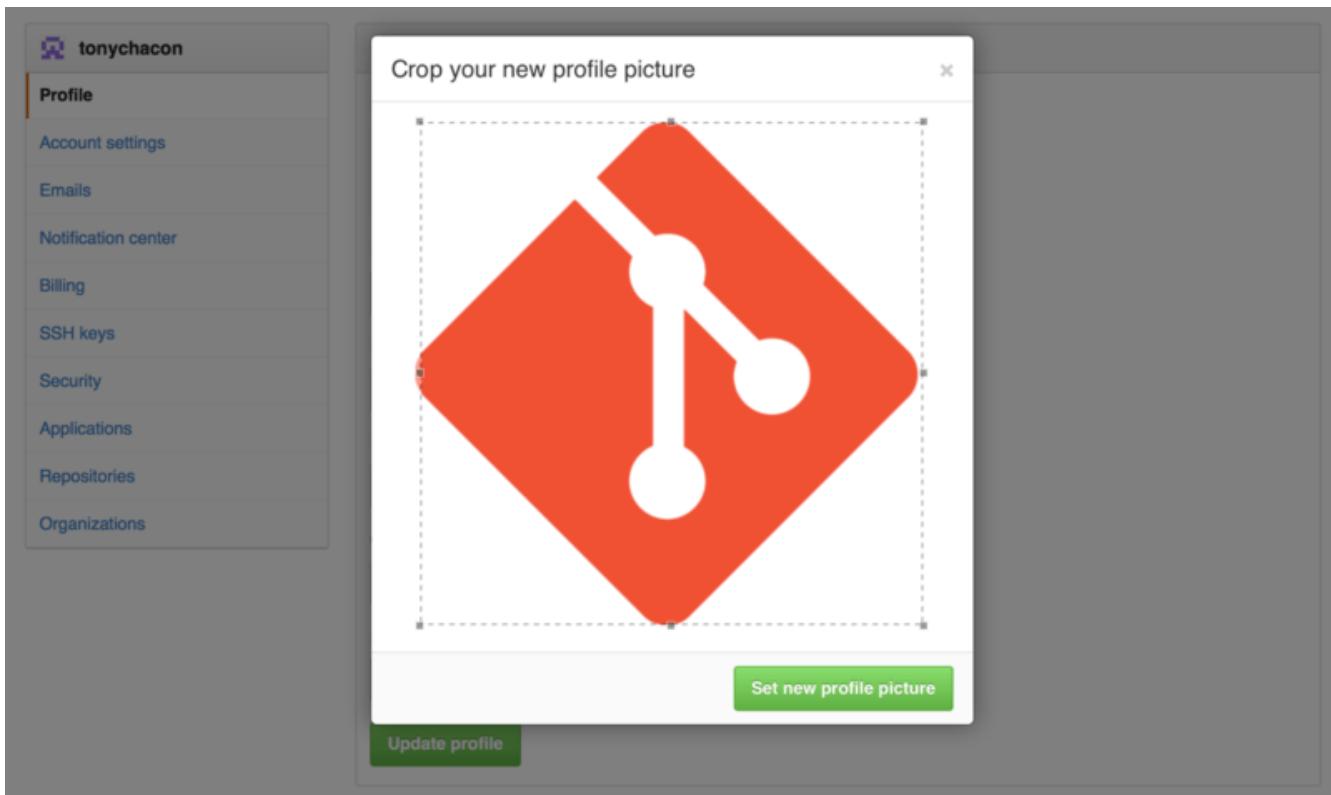


Figure 85. Dein Avatar-Bild beschneiden

Nun sehen die Betrachter überall dort, wo du auf der Website agierst, dein Avatar-Bild neben deinem Benutzernamen.

Wenn du beim beliebten Gravatar-Dienst (der oft für Wordpress-Konten verwendet wird) einen Avatar hochgeladen hast, wird dieser standardmäßig verwendet und du musst diesen Schritt nicht mehr ausführen.

## Deine Email-Adressen

GitHub bildet deine Git Commits auf deinen Account ab, wobei die Zuordnung per E-Mail erfolgt. Wenn du mehrere E-Mail-Adressen in deine Commits verwendest und möchtest, dass GitHub diese korrekt verknüpft, musst du alle von dir verwendeten E-Mail-Adressen in den E-Mail-Bereich des Admin-Bereichs aufnehmen.

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).

tonychacon@example.com Primary Public

tchacon@example.com Set as primary

tony.chacon@example.com Unverified

Add email address

Keep my email address private  
We will use **tonychacon@users.noreply.github.com** when performing Git operations and sending email on your behalf.

Figure 86. E-Mail-Adressen hinzufügen

Unter [E-Mail-Adressen hinzufügen](#) kannst du den Status einer E-Mail Adresse einsehen. Einige der möglichen Zustände sind oben abgebildet. Die oberste Adresse ist verifiziert und als Hauptadresse (engl. Primary) eingestellt, d.h. an diese Adresse gehen alle Benachrichtigungen und Empfangsbestätigungen. Die zweite Adresse ist verifiziert und kann, wenn du sie wechseln möchtest, als primär eingestellt werden. Die letzte Adresse ist noch nicht verifiziert, was bedeutet, dass du sie nicht zu deiner Hauptadresse machen kannst. Wenn GitHub eine davon in Commit-Nachrichten in einem beliebigen Repository auf der Website sieht, wird dieser mit deinem Benutzer-Konto verknüpft.

## Zwei-Faktor-Authentifizierung

Zuletzt solltest du aus Sicherheitsgründen auf jeden Fall die Zwei-Faktor-Authentifizierung oder „2FA“ einrichten. Die Zwei-Faktor-Authentifizierung ist ein Authentifizierungs-Mechanismus, der in letzter Zeit immer beliebter wird. Damit wird das Risiko verringert, dass dein Account durch den Diebstahl deines Passworts Schaden erleidet. Wenn du die Funktion einschaltest, fragt GitHub nach zwei verschiedenen Authentifizierungsmethoden, so dass ein Angreifer, wenn eine davon kompromittiert wird, nicht auf dein Konto zuzugreifen kann.

Du findest die Einstellungen für die Zwei-Faktor-Authentifizierung unter der Registerkarte „Password and authentication“ in deinen Kontoeinstellungen.

The screenshot shows the GitHub 'Security' settings page. On the left, a sidebar lists options: Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security (which is selected and highlighted in orange), Applications, Repositories, and Organizations. The main content area has a header 'Two-factor authentication' with a status of 'Off' and a red 'X'. A button 'Set up two-factor authentication' is visible. Below it, a note explains that two-factor authentication provides another layer of security. The 'Sessions' section lists a current session: 'Paris 85.168.227.34' (Your current session) running 'Safari on OS X 10.9.4' at 'Location: Paris, Ile-de-France, France' and 'Signed in: September 30, 2014'.

Figure 87. „2FA“ auf der Security-Registerkarte

Wenn du auf die Schaltfläche „Zwei-Faktor-Authentifizierung einrichten“ klickst, gelangst du zu einer Konfigurationsseite, auf der du eine Handy-App wählen kannst, um deine sekundären Code zu generieren (ein „zeitbasiertes Einmalpasswort“). Alternativ kannst du dir bei jedem Login von GitHub einen Code per SMS zusenden lassen.

Nachdem du dich für eine der beiden Methoden entschieden hast und den Anweisungen zur Einrichtung von 2FA gefolgt bist, ist dein Konto etwas sicherer. Du musst nun bei jedem Login in GitHub neben deinem Passwort einen zusätzlichen Code eingeben.

## Mitwirken an einem Projekt

Nun, da unser Konto eingerichtet ist, lass uns einige Details durchgehen, die nützlich sein könnten, um dir zu helfen, an einem bestehenden Projekt mitzuarbeiten.

### Forken von Projekten

Wenn du zu einem bestehenden Projekt beitragen möchtest, zu dem du keine Push-Berechtigungen hast, kannst du das Projekt „forken“. Wenn du ein Projekt „forkst“, erstellt GitHub eine Kopie des Projekts, die ganz dir gehört. Es befindet sich in deinem Namensraum (engl. namespace), und du können Daten dorthin pushen.



In der Vergangenheit war der Begriff „Fork“ in diesem Zusammenhang etwas Negatives und bedeutete, dass jemand ein Open-Source-Projekt in eine andere Richtung lenkt, manchmal ein konkurrierendes Projekt erstellt und die Beitragenden aufgespaltet hat. In GitHub ist ein „Fork“ schlichtweg das gleiche Projekt in deinem eigenen Namensraum, so dass du Änderungen an einem Projekt öffentlich vornehmen kannst und dabei einen transparenten Ansatz verfolgest.

Auf diese Weise müssen sich Projekte nicht darum kümmern, Benutzer als Beteiligte hinzuzufügen,

um ihnen Push-Zugriff zu geben. Jeder kann ein Projekt forken, dorthin pushen und seine Änderungen wieder in das originale Repository einbringen, indem er einen sogenannten Pull-Request erstellt, den wir als nächstes behandeln werden. Das eröffnet einen Diskussionsfaden mit Code-Review. Der Eigentümer und der Mitwirkende können dann über die Änderung kommunizieren, bis der Eigentümer mit ihr zufrieden ist und sie daraufhin zusammenführen (engl. merge) kann.

Um ein Projekt abzuspalten (engl. fork), gehst du auf die Projektseite und klickst auf die Schaltfläche „Fork“ oben rechts auf der Seite.



Figure 88. Die Schaltfläche „Fork“

Nach ein paar Sekunden wirst du auf deine neue Projektseite weitergeleitet, mit deiner eigenen beschreibbaren Kopie des Codes.

## Der GitHub Workflow

GitHub ist auf einen bestimmten Collaboration-Workflow ausgerichtet, der sich auf Pull-Requests konzentriert. Dieser Ablauf funktioniert unabhängig davon, ob du eng in einem Team, in einem einzigen gemeinsamen Repository, mit einem global verteilten Unternehmen oder einem Netzwerk von Fremden, über Dutzende von Forks, zusammenarbeitest und zu einem Projekt beiträgst. Er richtet sich nach dem [Themen-Branches](#) Workflow, der in Kapitel 3 [Git Branching](#) ausführlich besprochen wurde.

Im Prinzip funktioniert der Ablauf so:

1. Forke das Projekt.
2. Erstelle lokal einen Feature-Branch aus `master`.
3. Mache einige Commits, um das Projekt zu bearbeiten.
4. Pushe diesen Branch in dein GitHub-Projekt (den Fork).
5. Eröffne einen Pull-Request auf GitHub.
6. Diskutiere deine Änderung, optional: mache weitere Commits.
7. Der Projekteigentümer merged oder schließt den Pull Request.
8. Synchronisiere den aktualisierten `master` wieder mit deinem Fork.

Das ist im Grunde genommen der Integration-Manager-Workflow aus Kapitel 5 [Integrationsmanager](#). Aber anstatt E-Mails zur Kommunikation und Überprüfung von Änderungen zu verwenden, verwenden Teammitglieder die webbasierten Tools von GitHub.

Schauen wir uns ein Beispiel an, wie man mit diesem Workflow eine Änderung an einem Open-Source-Projekt vorschlägt, das auf GitHub gehostet wird.

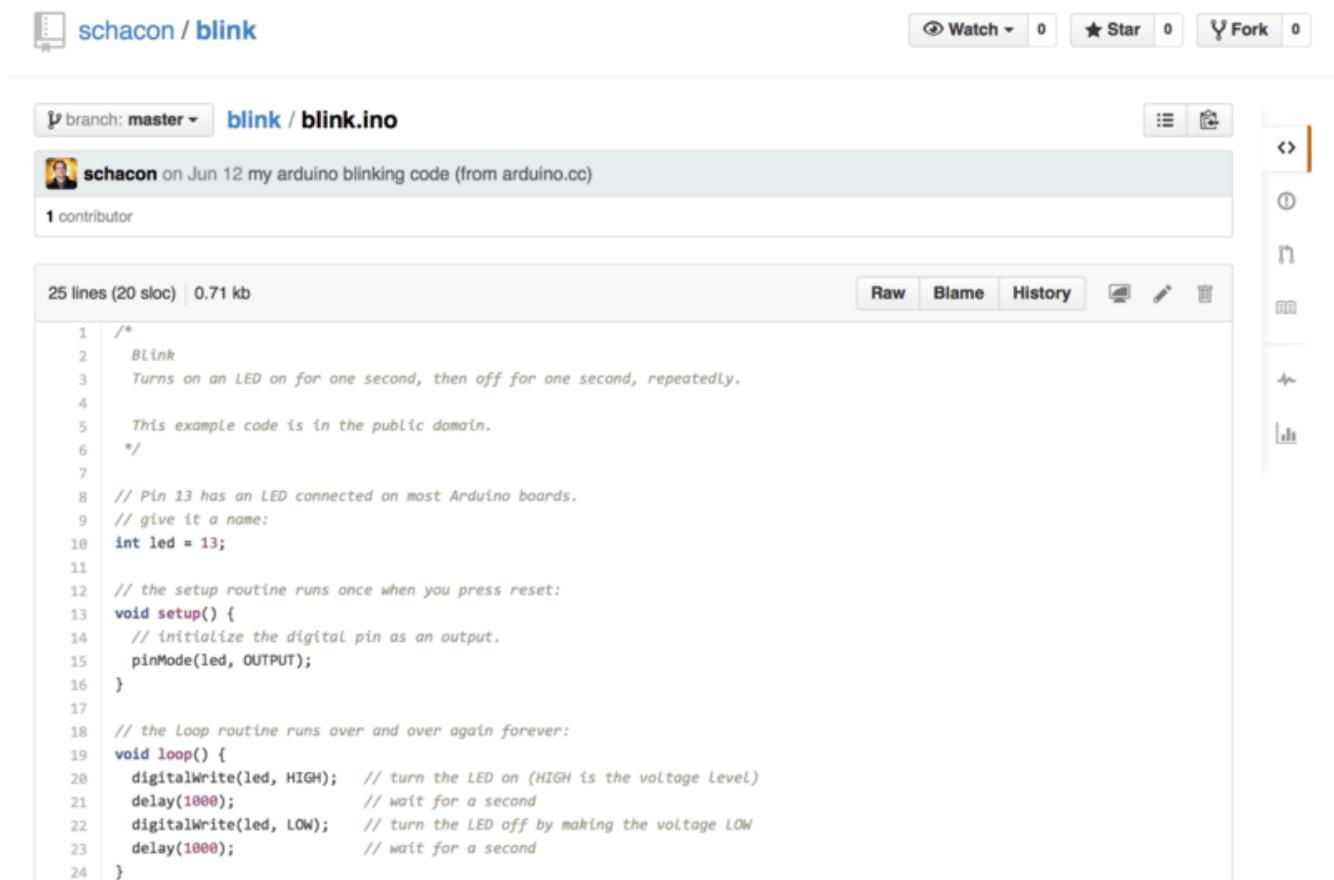


Du kannst für die meisten Aktionen das offizielle Werkzeug **GitHub CLI** anstatt die GitHub Weboberfläche nutzen. Das Werkzeug kann auf Windows, MacOS und

Linux Systemen installiert werden. Gehe zu [GitHub CLI homepage](#) für weitere Informationen, Installationsanleitungen und Handbücher.

## Anlegen eines Pull-Requests

Tony ist auf der Suche nach Code, der auf seinem programmierbaren Arduino-Mikrocontroller läuft und hat auf GitHub unter <https://github.com/schacon/blink> eine tolle Programmdatei gefunden.



The screenshot shows the GitHub repository page for the 'blink' project by 'schacon'. The repository has 0 stars, 0 forks, and 1 contributor. The code is a simple Arduino sketch named 'blink.ino' with 25 lines (20 sloc) and 0.71 kb. The code itself is as follows:

```
/*
 * 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 89. Das Projekt, zu dem wir beitragen wollen

Das einzige Problem ist, dass die Blinkfrequenz zu schnell ist. Wir finden es viel angenehmer, 3 Sekunden statt 1 Sekunde zwischen den einzelnen Zustandsänderungen zu warten. Lass uns also das Programm verbessern und es als Änderungsvorschlag an das Projekt senden.

Zuerst klicken wir, wie bereits erwähnt, auf die Schaltfläche 'Fork', um unsere eigene Kopie des Projekts zu erhalten. Unser Benutzername hier ist „tonychacon“, also ist unsere Kopie dieses Projekts unter <https://github.com/tonychacon/blink> zu finden und dort könnten wir es bearbeiten. Wir werden es aber lokal klonen, einen Feature-Branch erstellen, den Code ändern und schließlich diese Änderung wieder auf GitHub pushen.

```
$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
```

```
Switched to a new branch 'slow-blink'
```

```
$ sed -i '' 's/1000/3000/' blink.ino (macOS) ③
# If you're on a Linux system, do this instead:
# $ sed -i 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
[-delay(1000);-]{+delay(3000);+} // wait for a second
    digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
[-delay(1000);-]{+delay(3000);+} // wait for a second
}

$ git commit -a -m 'Change delay to 3 seconds' ⑤
[slow-blink 5ca509d] Change delay to 3 seconds
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink
```

① Wir klonen unseren Fork des Projekts lokal.

② Wir erstellen einen Branch mit prägnantem Namen.

③ Wir nehmen unsere Anpassung am Code vor.

④ Wir überprüfen, ob die Änderung gut ist.

⑤ Wir committen unsere Änderung in den Feature-Branch.

⑥ Wir pushen unseren neuen Feature-Branch zurück zu unserer GitHub-Fork.

Wenn wir nun zu unserem Fork auf GitHub zurückkehren, können wir sehen, dass GitHub bemerkt hat, dass wir einen neuen Feature-Branch gepusht haben. GitHub zeigt uns einen großen grünen Button, um unsere Änderungen zu überprüfen und einen Pull Request zum ursprünglichen Projekt zu öffnen.

Du kannst alternativ auch die Seite „Branches“ bei <https://github.com/<user>/<project>/branches>

aufzurufen, um deinen Branch auszuwählen und von dort aus einen neuen Pull Request zu öffnen.

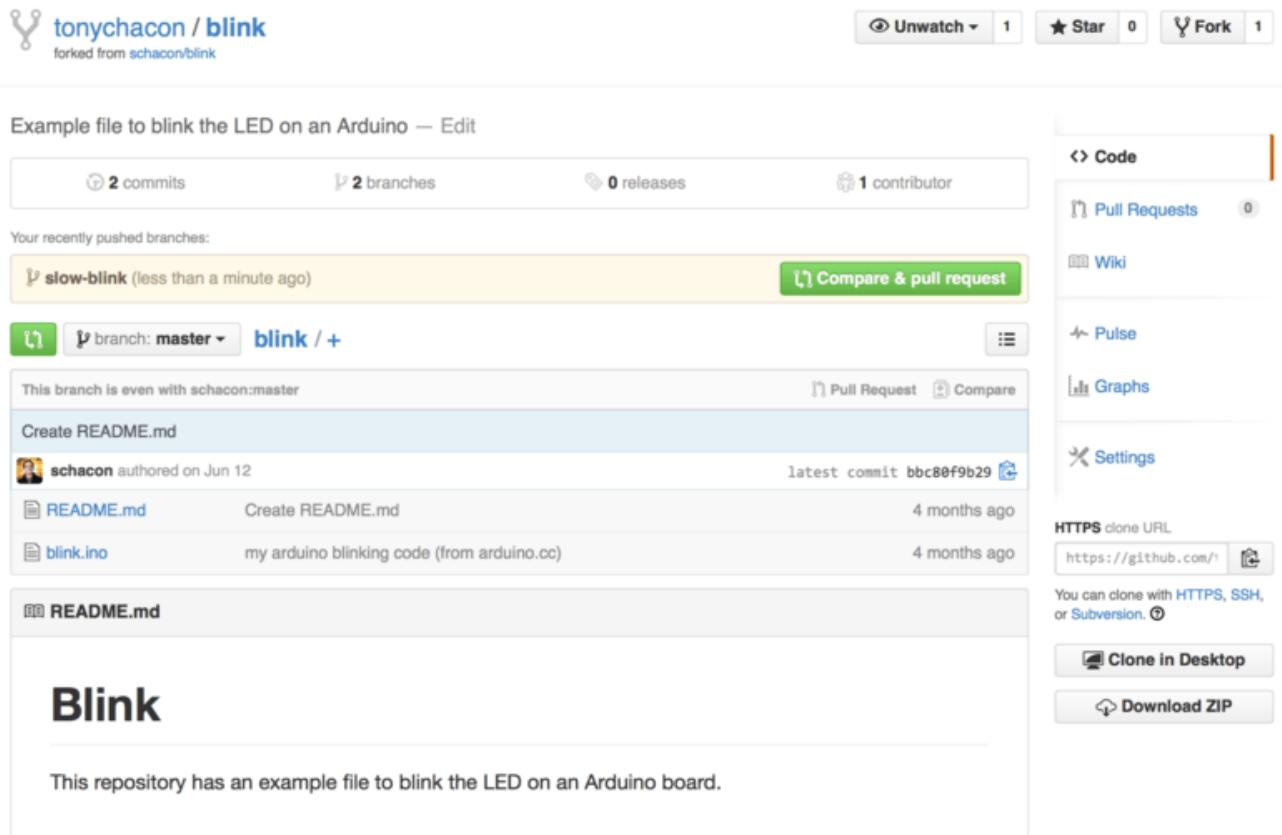
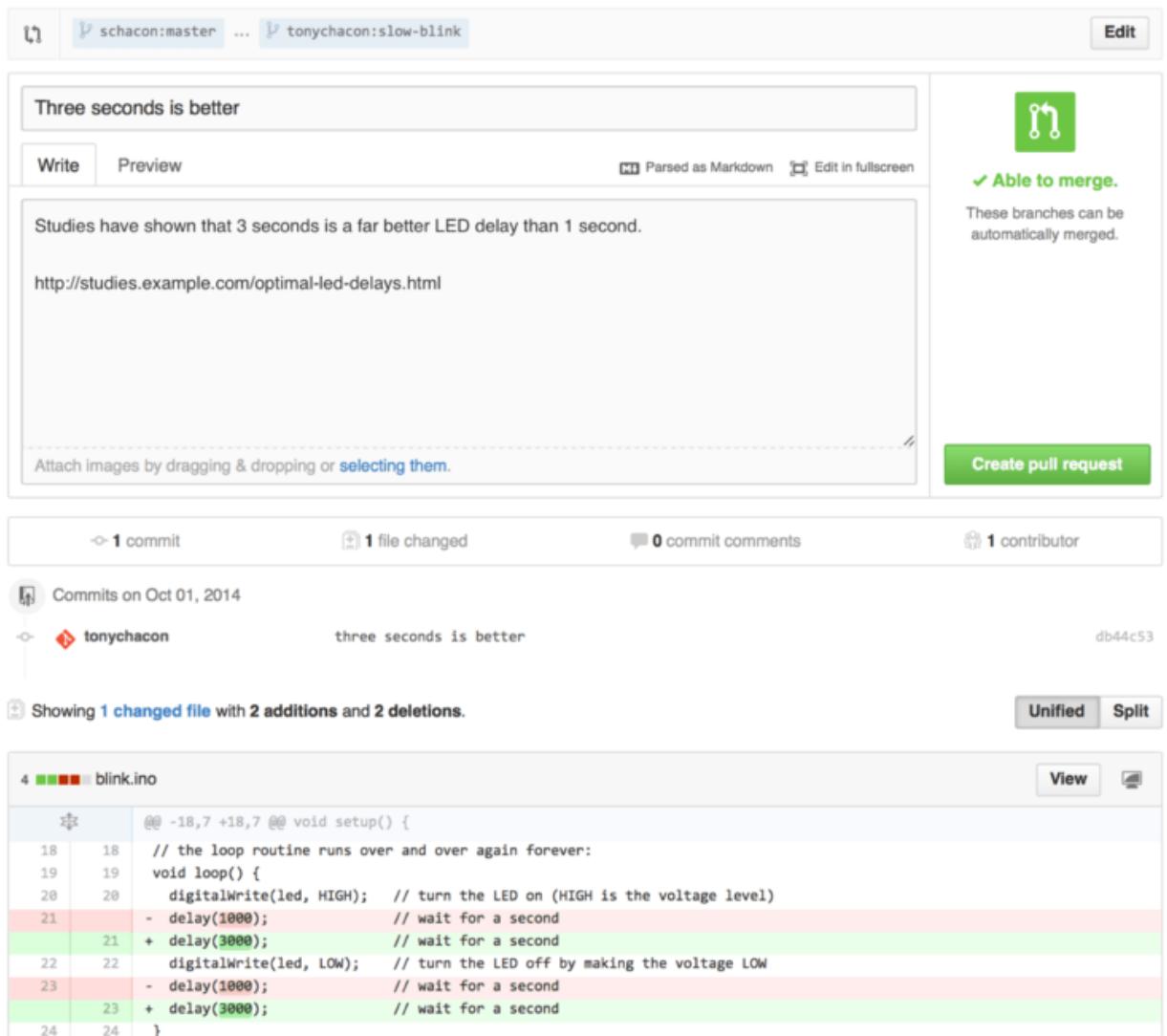


Figure 90. Die Schaltfläche Pull Request

Wenn wir auf die grüne Schaltfläche klicken, sehen wir einen Bildschirm, auf dem du aufgefordert wirst, deinen Pull Request einen Titel und eine Beschreibung zu geben. Es ist fast immer sinnvoll, sich hierfür ein bisschen Mühe zu geben. Eine gute Beschreibung hilft dem Eigentümer des ursprünglichen Projekts, festzustellen, warum du die Änderungen machst, ob deine vorgeschlagenen Änderungen korrekt sind und ob die Annahme der Änderungen das ursprüngliche Projekt verbessern würden.

Wir erhalten auch eine Liste der Commits in unserem Feature-Branch, die dem **master** Branch „**voraus**“ (engl. ahead) sind (in diesem Fall nur der eine) und ein vereinheitlichtes Diff aller Änderungen, die vorgenommen werden, falls dieser Branch vom Projektleiter gemerged wird.



The screenshot shows the GitHub pull request creation interface. At the top, there's a header with the repository name 'tonychacon / blink' and a 'forked from schacon/blink' note. Below the header, the pull request details are shown: the base branch is 'schacon:master' and the target branch is 'tonychacon:slow-blink'. The title of the pull request is 'Three seconds is better'. The description states: 'Studies have shown that 3 seconds is a far better LED delay than 1 second.' A link to 'http://studies.example.com/optimal-led-delays.html' is provided. On the right side, there's a green button with a gear icon labeled 'Able to merge.' and a note: 'These branches can be automatically merged.' Below the description, there's a section for attachments with the placeholder 'Attach images by dragging & dropping or selecting them.' At the bottom of the pull request view, there are summary statistics: 1 commit, 1 file changed, 0 commit comments, and 1 contributor. The commit details show a single commit from 'tonychacon' on Oct 01, 2014, with the commit message 'three seconds is better' and the commit hash 'db44c53'. Below the commit details, it says 'Showing 1 changed file with 2 additions and 2 deletions.' The code diff view shows changes made to 'blink.ino':

```

4 4 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 }

```

Figure 91. Seite zur Erstellung von Pull-Requests

Wenn du in diesem Fenster auf die Schaltfläche 'Create pull request' klickst, wird der Eigentümer/Leiter des Projekts, für das du eine Abspaltung (engl. Fork) vorgenommen hast, benachrichtigt. Er wird informiert, dass jemand eine Änderung vorschlägt, und verlinkt auf eine Seite, die alle diese Informationen enthält.



Pull-Requests werden häufig für öffentliche Projekte wie dieses verwendet, wenn die Änderungen der Beitragende vollständige Änderung sind. Sie werden jedoch auch zu Beginn des Entwicklungszyklus in internen Projekten verwendet. Da du den Feature-Branch auch **nach** dem Öffnen des Pull-Requests weiter bearbeiten kannst, wird er oft früh geöffnet. Er bietet die Möglichkeit, um die Arbeit als Team in einem Gesamtkontext zu iterieren, anstatt ihn erst am Ende des Prozesses zu öffnen.

## Iteration eines Pull-Requests

An dieser Stelle kann sich der Projektverantwortliche die vorgeschlagene Änderung ansehen und mergen, ablehnen oder kommentieren. Nehmen wir an, er mag die Idee, würde aber eine etwas

längere Zeit bevorzugen, in der das Licht aus ist.

Während diese Unterhaltung über E-Mail in den in Kapitel 5 [Verteiltes Git](#) dargestellten Workflows stattfinden kann, geschieht dies bei GitHub online. Der Projektverantwortliche kann das vereinheitlichte Diff überprüfen und einen Kommentar hinterlassen, indem er auf eine der Zeilen klickt.

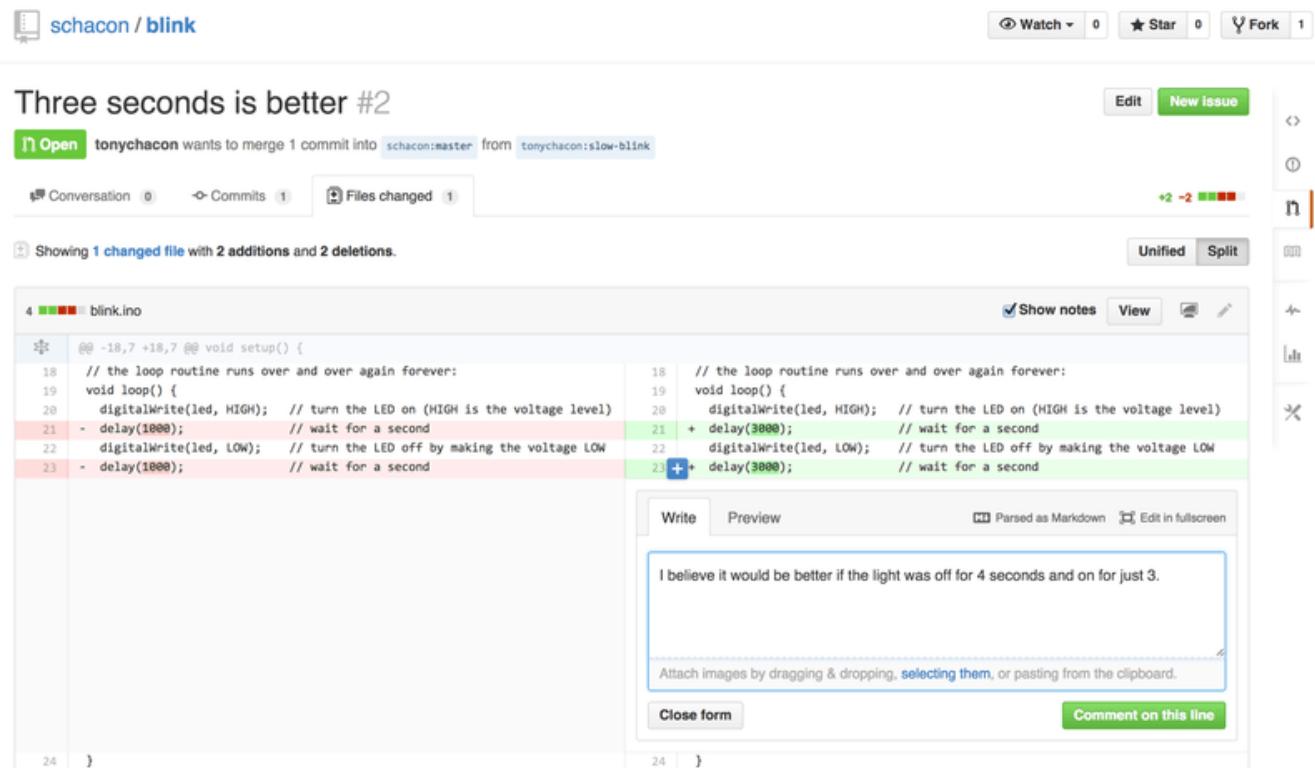


Figure 92. Kommentar zu einer bestimmten Codezeile in einem Pull-Request

Sobald der Betreuer diesen Kommentar abgibt, erhält die Person, die den Pull-Request geöffnet hat (und auch alle anderen, die das Repository beobachten), eine Benachrichtigung. Wir werden das später noch einmal anpassen, aber wenn er E-Mail-Benachrichtigungen eingeschaltet hat, bekommt Tony eine E-Mail wie diese:



Figure 93. Kommentare, als E-Mail-Benachrichtigungen gesendet

Jeder kann auch allgemeine Kommentare zum Pull Request hinterlassen. Auf der [Pull Request Diskussions-Seite](#) sehen wir ein Beispiel dafür, wie der Projektleiter sowohl eine Zeile Code kommentiert als auch einen allgemeinen Kommentar im Diskussionsbereich hinterlässt. Du siehst,

dass auch die Code-Kommentare in das Diskussionsfenster sichtbar sind.

## Three seconds is better #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	((6 lines not shown))
23	23	digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
		- delay(1000); // wait for a second
		+ 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 94. Pull Request Diskussions-Seite

Jetzt kann der Beitragende sehen, was er tun muss, damit seine Änderung akzeptiert wird. Zum Glück ist das sehr einfach. Wo du über E-Mail deine Daten erneut in die Mailingliste eintragen musst, committest du mit GitHub einfach erneut in den Feature-Branch und pushst, wodurch der Pull-Request automatisch aktualisiert wird. Im [finalen Pull-Request](#) siehst du auch, dass der alte Code-Kommentar in dem aktualisierten Pull-Request zusammengeklappt wurde, da er auf eine inzwischen geänderte Zeile basiert.

Das Hinzufügen von Commits zu einem bestehenden Pull Request löst keine weitere Benachrichtigung aus. Nachdem Tony seine Korrekturen gepusht hat, beschließt er, einen Kommentar zu hinterlassen, um den Projektträger darüber zu informieren, dass er die gewünschte Änderung vorgenommen hat.

## 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's first comment (11 minutes ago) links to a study on optimal LED delays. schacon comments on an outdated diff (5 minutes ago). tonychacon adds some commits (2 minutes ago) for longer off time and removes trailing whitespace. tonychacon's final comment (10 seconds ago) asks for more changes. At the bottom, a message says "This pull request can be automatically merged." and "Merge pull request".

Figure 95. finaler Pull-Request

Eine bemerkenswerte Besonderheit ist, dass du den „vereinheitlichten“ Diff erhältst, wenn du auf die Registerkarte „Files Changed“ in diesem Pull-Request klickst – d.h. die gesamte aggregierte Differenz, die in deinen Hauptbranch eingebracht würde, wenn dieser Feature-Branch gemerged würde. Im Sinne von `git diff` zeigt es dir grundsätzlich automatisch `git diff master...<branch>` für den Branch, auf dem dieser Pull Request basiert. Siehe Kapitel 5 [Festlegen, was eingebracht wird](#) für weitere Informationen über diese Art von Diffs.

Außerdem prüft GitHub, ob der Pull-Request sauber mergen würde und stellt eine Schaltfläche zur Verfügung, mit der du den Merge auf dem Server durchführen kannst. Diese Schaltfläche erscheint nur, wenn du Schreibzugriff auf das Repository hast und ein trivialer Merge möglich ist. Wenn du darauf klickst, führt GitHub einen „non-fast-forward“-Merge durch, was bedeutet, dass selbst wenn es sich bei dem Merge um einen schnellen Vorlauf (engl. fast-forward) handeln könnte, immer noch ein Merge-Commit erstellt wird.

Wenn du möchtest, kannst du den Branch einfach herunterladen (engl. pull) und lokal zusammenführen (engl. merge). Wenn du diesen Branch mit dem **master** Branch mergst und ihn nach GitHub pushen willst, wird der Pull Request automatisch geschlossen.

Das ist der grundsätzliche Workflow, den die meisten GitHub-Projekte verwenden. Feature-Branches werden erstellt, Pull-Requests werden geöffnet, es folgt eine Diskussion, möglicherweise wird eine weitere Überarbeitung des Branches durchgeführt und schließlich wird der Request entweder geschlossen oder zusammengeführt.

#### *Nicht nur Forks*



Es ist wichtig zu erwähnen, dass du auch einen Pull-Request zwischen zwei Branches im selben Repository öffnen können. Wenn du mit jemandem an einer Funktion arbeitest und beide Schreibrechte auf das Projekt haben, kannst du einen Feature-Branch in das Repository verschieben und einen Pull-Request darauf an den **master** Branch desselben Projekts öffnen, um den Code-Review- und Diskussionsprozess einzuleiten. Es ist kein Forking notwendig.

## Erweiterte Pull-Requests

Nachdem wir nun die Grundlagen für einen Beitrag zu einem Projekt auf GitHub erläutert haben, möchten wir dir einige interessante Tipps und Tricks zu Pull-Requests geben, damit du diese effektiver nutzen kannst.

### Pull-Requests als Patches

Es ist wichtig zu verstehen, dass viele Projekte Pull-Requests nicht wirklich als eine Reihe perfekter Patches betrachten, die sauber angewendet werden sollten, so wie es bei den meisten Mailinglisten-basierten Projekten mit Patch-Serienbeiträgen ist. Die meisten GitHub-Projekte betrachten Pull-Request-Branches als iterative Dialoge für vorgesehene Änderungen. Dies führt in der Regel zu einem vereinheitlichten Diff, der am Ende gemerged wird.

Das ist ein wichtiger Unterschied, denn im Allgemeinen wird die Änderung vorgeschlagen, bevor der Code fertig ist, was bei Beiträgen von Patch-Serien auf Mailinglistenbasis weitaus seltener ist. So kann früher mit den Projekt-Betreuern gesprochen werden und somit ist die richtige Lösung eher eine Art Gemeinschaftsarbeit. Wenn Code mit einem Pull-Request vorgeschlagen wird und die Maintainer oder die Community eine Änderung vorschlagen, wird die Patch-Serie im Allgemeinen nicht neu aufgerollt. Es wird der Unterschied als neuer Commit an den Branch weitergegeben (gepusht), wodurch der Dialog an dieser Stelle im intakten Kontext der vorherigen Arbeit fortgesetzt wird.

Wenn du beispielsweise zurückgehen und dir den **finalen Pull-Request** erneut ansehen willst, wirst du feststellen, dass der Beitragende seinen Commit nicht rebased und einen weiteren Pull-Request geöffnet hat. Stattdessen wurden neue Commits hinzugefügt und sie in den bestehenden Branch gepusht. Wenn du dir also zu einem späterem Zeitpunkt den Pull Request nochmal anschau, kannst du den gesamten Kontext finden, in dem die Entscheidungen getroffen wurden. Wenn du auf der Website auf die Schaltfläche „Merge“ klickst, wird ein Merge-Commit erstellt, der auf den Pull-Request verweist. Somit kannst du bei Bedarf sehr schnell die ursprüngliche Diskussion begutachten.

## Mit dem Upstream Schritt halten

Wenn dein Pull-Request veraltet ist oder anderweitig nicht sauber zusammengeführt wird, solltest du ihn reparieren, damit der Betreuer ihn leicht mergen kann. GitHub wird das für dich testen und dich am Ende jedes Pull Requests darüber informieren, ob das Merge trivial ist oder nicht.

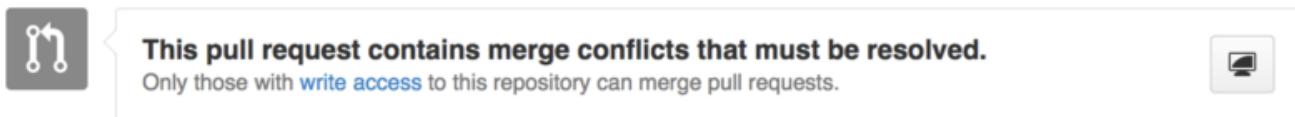


Figure 96. Pull Request lässt sich nicht sauber mergen

Wenn du etwa „[Pull Request lässt sich nicht sauber mergen](#)“ siehst, solltest du deinen Branch so reparieren, dass er grün wird und der Maintainer keine zusätzliche Arbeit leisten muss.

Du hast grundsätzlich zwei Möglichkeiten, wie du das realisieren kannst. Du kannst deinen Branch entweder auf den Ziel-Branch rebasen (normalerweise den `master` Branch des von dir geforkten Repositorys), oder du kannst den Ziel-Branch mit deinem Branch mergen.

Die meisten Entwickler auf GitHub werden sich aus den gleichen Gründen für Letzteres entscheiden, die wir im vorherigen Abschnitt erläutert haben. Was wichtig ist, ist die Verlaufskontrolle und der endgültige Merge, so dass das Rebasing nicht viel mehr bringt als eine etwas aufgeräumtere Historie. Im Gegenzug ist es **viel** schwieriger und fehleranfälliger.

Wenn du im Ziel-Branch mergen willst, um deinen Pull-Request zusammenzuführen zu können, solltest du das ursprüngliche Repository als neuen Remote hinzufügen. Dann machst du ein `git fetch` darauf, führst den Haupt-Branch dieses Repositorys in deinem Feature-Branch zusammen, behebst alle Probleme und pushst es schließlich wieder in den gleichen Branch, in dem du den Pull-Request geöffnet hast.

Nehmen wir zum Beispiel an, dass der ursprüngliche Autor in dem von uns zuvor verwendeten Beispiel „tonychacon“ eine Änderung vorgenommen hat, die zu einem Konflikt im Pull-Request führt. Gehen wir diese Schritte einzeln durch.

```
$ 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 Das Original-Repository als Remote mit der Bezeichnung `upstream` hinzufügen.
- ② Die neueste Arbeit von diesem Remote abrufen (engl. `fetch`).
- ③ Den Haupt-Branch dieses Repositorys mit dem Feature-Branch mergen.
- ④ Den aufgetretenen Konflikt beheben.
- ⑤ Zum gleichen Feature-Branch zurück pushen.

Sobald du das getan hast, wird der Pull-Request automatisch aktualisiert und erneut überprüft, um festzustellen, ob er sauber zusammengeführt werden kann.

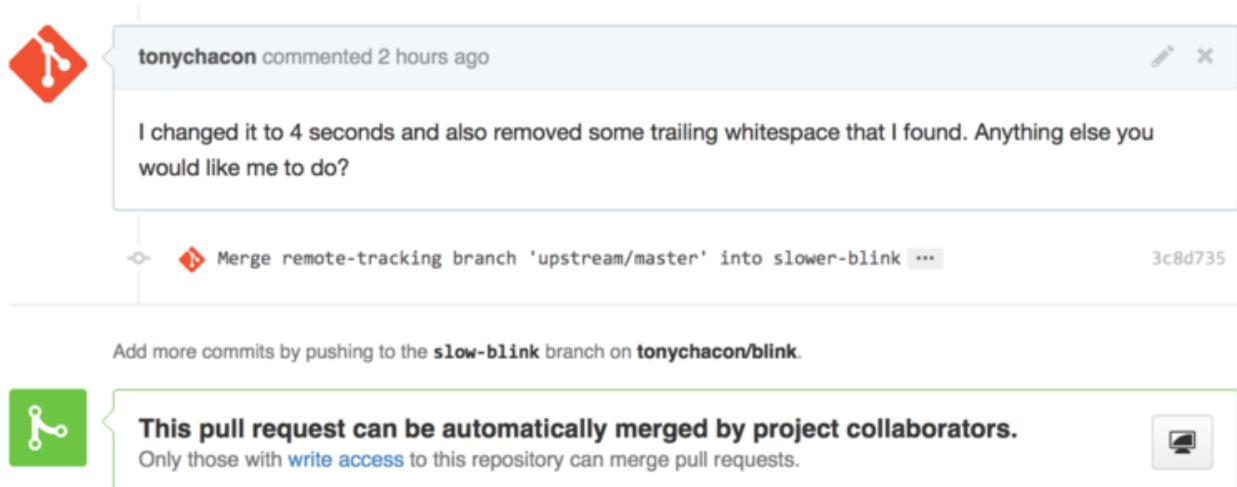


Figure 97. Pull-Request wird nun sauber zusammengeführt

Einer der großen Pluspunkte von Git ist, dass man das kontinuierlich tun kann. Wenn du ein sehr lang laufendes Projekt hast, kannst du leicht immer wieder aus dem Zielbranch heraus mergen und müsstest dich nur mit Konflikten befassen, die seit dem letzten Mal, als du zusammengeführt hast, aufgetreten sind; was den Prozess sehr überschaubar macht.

Wenn du den Branch unbedingt rebasen willst, um ihn aufzuräumen, kannst du das natürlich tun. Es wird jedoch dringend empfohlen, den Branch, auf dem der Pull-Request bereits geöffnet ist, nicht zu force-pushen. Wenn von Anderen gepullt wurde und weitere Arbeiten daran durchgeführt wurden, stößt man auf alle in Kapitel 3, [Die Gefahren des Rebasing](#) beschriebenen Probleme. Schiebe stattdessen den rebasten Branch zu einem neuen Branch auf GitHub und öffne einen

brandneuen Pull Request mit Bezug auf den alten Branch und schließen den originalen Pull Request.

## Referenzen

Möglicherweise lautet deine nächste Frage: „Wie verweise ich auf den alten Pull-Request?“. Es stellt sich heraus, dass es viele, viele Möglichkeiten gibt, auf andere Dinge Bezug zu nehmen, und zwar fast überall dort, wo man in GitHub schreiben kann.

Beginnen wir mit der Frage, wie man einen anderen Pull-Request oder ein Issue vergleicht. Alle Pull-Requests und Issues sind mit Nummern versehen und innerhalb des Projekts eindeutig. Beispielsweise ist es nicht möglich, Pull Request #3 *und* Issue #3 anzulegen. Wenn du auf einen Pull-Request oder ein Issue von einem anderen verweisen möchtest, kannst du einfach <num> in einen Kommentar oder eine Beschreibung eingeben. Du kannst auch präziser sein, wenn die Issue- oder Pull-Anforderung an einem anderen Ort liegen. Schreibe Benutzername<num>, wenn du dich auf ein Issue oder Pull Request beziehst, in einen Fork des Repositorys, in dem du dich befindest. Oder schreibe Benutzername/repo#<num>, um auf etwas in einem anderen Repository zu verweisen.

Schauen wir uns ein Beispiel an. Angenommen, wir haben den Branch aus dem vorherigen Beispiel rebased, einen neuen Pull-Request für ihn erstellt und jetzt wollen wir die alte Pull-Anforderung aus der neuen aufrufen. Wir möchten auch auf ein Problem im Fork des Repositorys und auf ein Problem in einem ganz anderen Projekt verweisen. Wir können die Beschreibung wie bei Querverweise in einem Pull-Request eingeben.

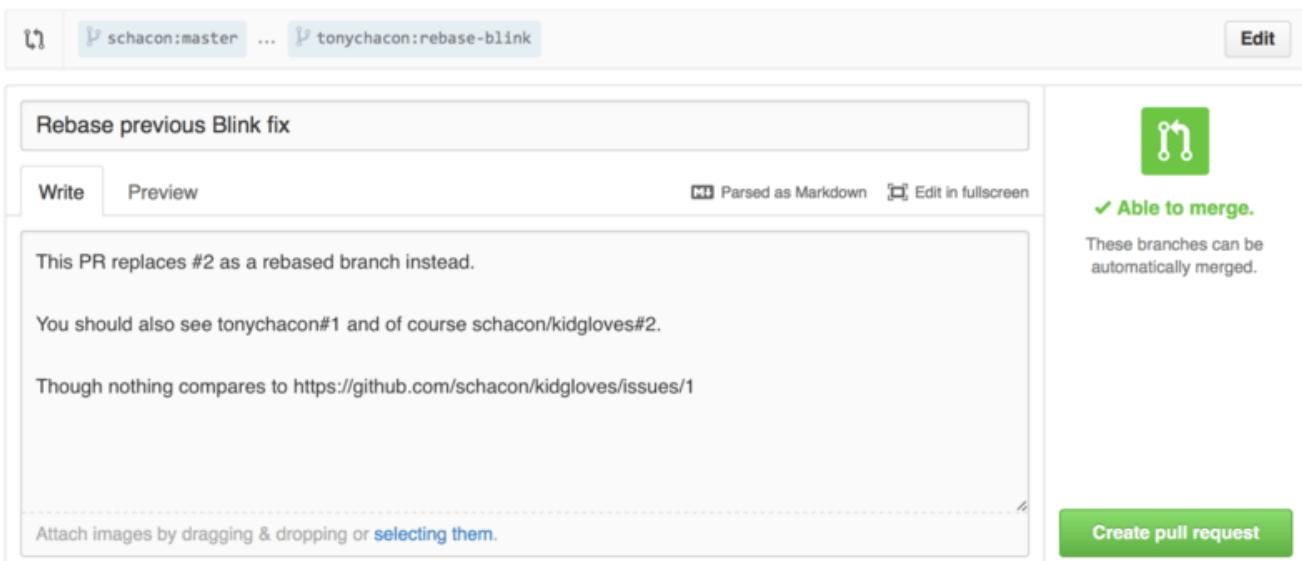


Figure 98. Querverweise in einem Pull-Request

Wenn wir diese Pull-Anfrage einreichen, werden wir sehen, dass alles wie in „Querverweise, die in einem Pull-Request erzeugt wurden“ dargestellt wird.

## Rebase previous Blink fix #4

The screenshot shows a GitHub pull request page. At the top, there's a green button labeled "Open" and a status message: "tonychacon wants to merge 2 commits into schacon:master from tonychacon:rebase-blink". Below this, there are tabs for "Conversation" (0), "Commits" (2), and "Files changed" (1). A comment from "tonychacon" is visible, stating: "This PR replaces #2 as a rebased branch instead. You should also see tonychacon#1 and of course schacon/kidgloves#2. Though nothing compares to schacon/kidgloves#1". Below the comment, a commit history shows two commits by "tonychacon": "three seconds is better" (commit hash afe904a) and "remove trailing whitespace" (commit hash a5a7751).

Figure 99. Querverweise, die in einem Pull-Request erzeugt wurden

Bitte beachte, dass die vollständige GitHub-URL, die wir dort eingegeben haben, auf die benötigten Informationen gekürzt wurde.

Wenn Tony nun zurück geht und den ursprünglichen Pull-Request schließt, können wir sehen, dass GitHub automatisch ein Trackback-Ereignis in der Pull-Request Zeitleiste erstellt hat, indem er ihn im neuen Pull-Request erwähnt. Das bedeutet, dass jeder, der diesen Pull-Request aufruft, sieht, dass er geschlossen ist. Er kann leicht auf denjenigen zurückgreifen, der ihn ersetzt hat. Der Link wird in etwa wie in „Zurück zum neuen Pull-Request in der geschlossenen Pull-Request Zeitleiste“ aussehen.

The screenshot shows the timeline of a GitHub pull request. It includes a commit from "tonychacon" to "Merge remote-tracking branch 'upstream/master' into slower-blink" (commit hash 3c8d735). A note indicates "tonychacon referenced this pull request 2 minutes ago" and links to "Rebase previous Blink fix #4". This entry has a green "Open" button. Below it, another note shows "tonychacon closed this just now". A summary box at the bottom states "Closed with unmerged commits" and "This pull request is closed, but the tonychacon:slow-blink branch has unmerged commits.", with a "Delete branch" button next to it.

Figure 100. Zurück zum neuen Pull-Request in der geschlossenen Pull-Request Zeitleiste

Zusätzlich zu den Issue-Nummern kannst du auch auf einen bestimmten Commit per SHA-1 referenzieren. Du musst einen vollen 40-stelligen SHA-1 angeben. Wenn GitHub das in einem Kommentar sieht, wird er direkt auf den Commit verlinken. Wie bei Issues kannst du auch hier auf Commits in Forks oder anderen Repositorys verweisen.

## GitHub-Variante von Markdown

Die Verknüpfung mit anderen Issues ist nur der Anfang von interessanten Dingen, die du mit fast jeder Textbox auf GitHub machen kannst. In Issue- und Pull-Request-Beschreibungen, Kommentaren, Code-Kommentaren und mehr, kannst du das sogenannte „GitHub Flavored Markdown“ verwenden. Markdown fühlt sich an wie das Schreiben in Klartext, hat aber umfangreiche Darstellungs-Optionen.

Im [Beispiel für GitHub-Variante von Markdown, geschrieben und gerendert](#) findest du ein Muster, wie Kommentare oder Text geschrieben und dann mit Markdown gerendert werden können.

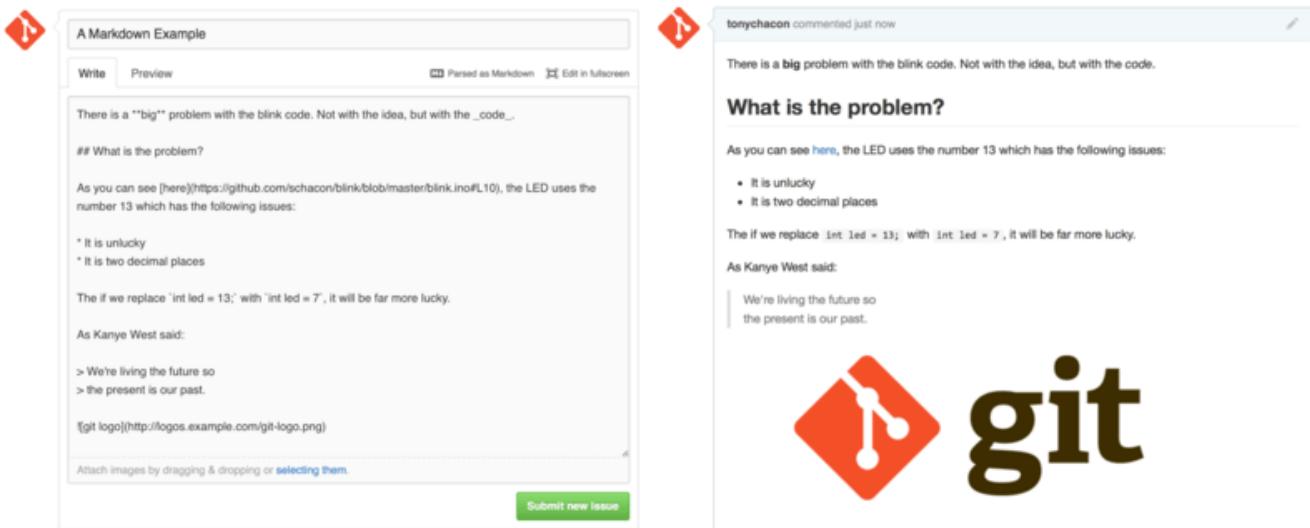


Figure 101. Beispiel für GitHub-Variante von Markdown, geschrieben und gerendert

Die GitHub-Variante von Markdown bietet mehr Möglichkeiten, als die normale Markdown-Syntax. Alle diese Funktionen können sehr nützlich sein, wenn du hilfreiche Pull-Request, Issue-Kommentare oder Beschreibungen erstellst.

## Aufgabenlisten

Die wirklich praktische GitHub-spezifische Markdown-Funktion, vor allem für die Verwendung in Pull-Requests, ist die Aufgabenliste. Eine Aufgabenliste ist eine Liste von Kontrollkästchen für alle Vorgänge, die du erledigen möchtest. Wenn du sie in einen Issue oder Pull-Request einfügst, werden normalerweise Punkte angezeigt, die du erledigen sollst, bevor du den gesamten Vorgang als erledigt betrachten kannst.

Du kannst eine Task-Liste wie folgt anlegen:

- [X] Write the code
- [ ] Write all the tests
- [ ] Document the code

Wenn wir das in die Beschreibung unseres Pull Request oder Issue aufnehmen, sehen wir, dass es wie in [Aufgabenliste in Markdown-Kommentar, gerendert](#) dargestellt wird.



Figure 102. Aufgabenliste in Markdown-Kommentar, gerendert

Diese Funktion wird häufig in Pull Requests verwendet, um zu verdeutlichen, was du alles auf dem Branch erledigen möchtest, bevor der Pull Request bereit für die Zusammenführung ist. Der wirklich coole Teil ist, dass du einfach auf die Kontrollkästchen klicken kannst, um den Kommentar zu aktualisieren. Du musst nicht den Markdown bearbeiten, um Aufgaben als erledigt zu markieren.

Darüber hinaus sucht GitHub nach Aufgabenlisten in deinen Issues und Pull Requests und zeigt sie als Metadaten zu den Seiten an, auf denen sie stehen. Wenn du z.B. einen Pull-Request mit Aufgabenliste hast und dir die Übersichtsseite aller Pull-Requests ansiehst, kannst du sehen, wie weit er abgearbeitet ist. Das hilft den Teilnehmern, Pull-Requests in Teilaufgaben aufzuschlüsseln und anderen Teilnehmern, den Fortschritt des Branch zu verfolgen. Ein Beispiel dafür findest du bei: [Task-Liste \(Zusammenfassung\) in der Pull-Request-Liste](#).

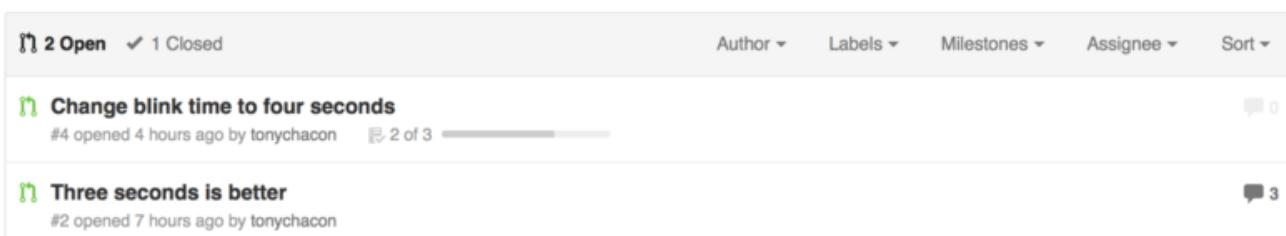


Figure 103. Task-Liste (Zusammenfassung) in der Pull-Request-Liste

Diese Funktion ist unglaublich nützlich, wenn du einen Pull Request frühzeitig öffnest und damit deinen Fortschritt bei der Realisierung des Features verfolgst (engl. track).

## Code-Schnipsel

Du kannst auch Code-Schnipsel zu Kommentaren hinzufügen. Es ist dann besonders praktisch, wenn du etwas präsentieren möchtest, das du versuchen *könntest*, bevor du es tatsächlich als Commit auf deinem Branch einbaust. Das wird auch oft verwendet, um Beispielcode hinzuzufügen, der verrät, was nicht funktioniert oder was dieser Pull-Request umsetzen könnte.

Um ein Code-Schnipsel hinzuzufügen, musst du ihn in zwei 3-fach Backticks (`) „einzäunen“.

```
```java
for(int i=0 ; i < 5 ; i++)
{
    System.out.println("i is : " + i);
}
```



Wenn du den Namen der Programmiersprache hinzufügst, wie wir es mit `java` getan haben, wird GitHub versuchen, die Syntax hervorzuheben. Wie im Beispiel oben, würde es zu „eingezäuntes“, gerendertes Code-Schnipsel-Beispiel gerendert werden.



Figure 104. „eingezäuntes“, gerendertes Code-Schnipsel-Beispiel

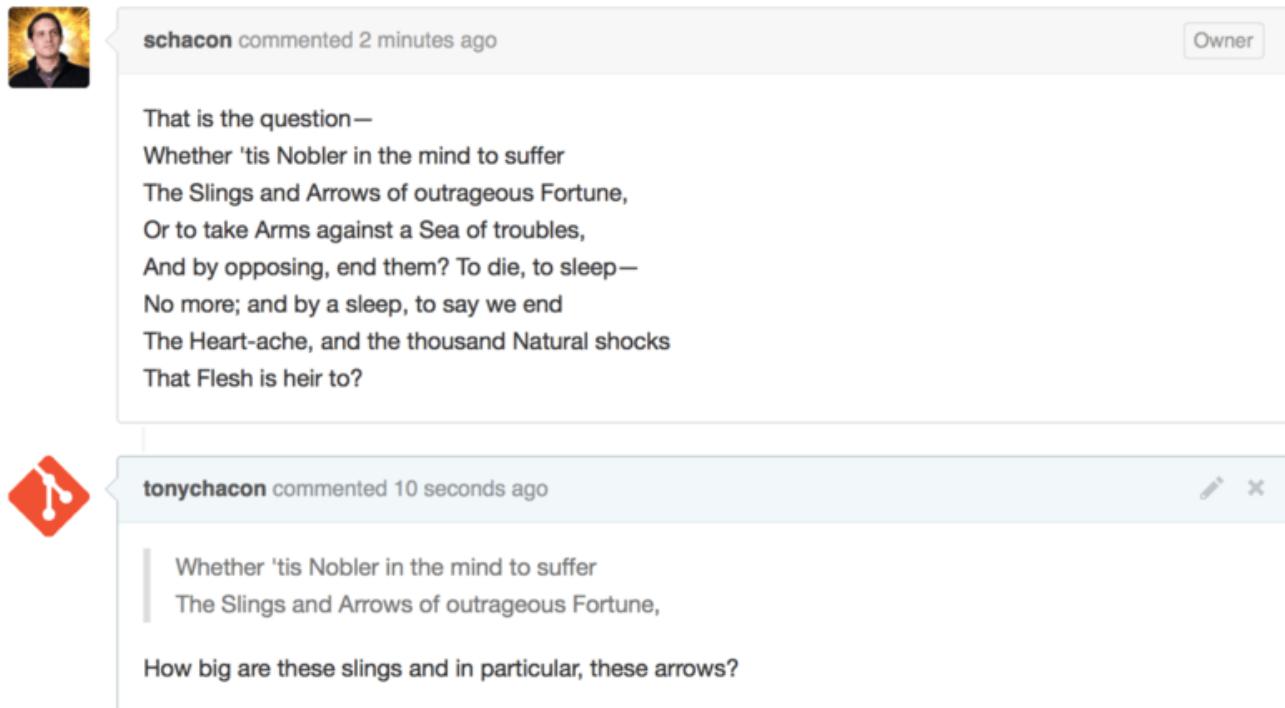
## Quoten (Zitieren)

Wenn du auf einen kleinen Teil eines langen Kommentars reagieren willst, kannst du selektiv aus dem anderen Kommentar zitieren, indem du den Zeilen das Zeichen `>` voranstellst. Es ist sogar so häufig und nützlich, dass es eine Tastenkombination dafür gibt. Wenn du Text in einem Kommentar markierst, auf den du direkt antworten möchtest, und die Taste `r` drückst, wird dieser Text in der Kommentarbox für dich zitiert.

Zitate (engl. quotes) sehen in etwa so aus:

```
> 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?
```

Nach dem Rendern sieht der Kommentar wie folgt aus: [gerendertes Zitat-Beispiel](#).



The screenshot shows a GitHub comment thread. The first comment, by user schacon, contains a blockquote of a famous soliloquy from Hamlet:

```

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,  

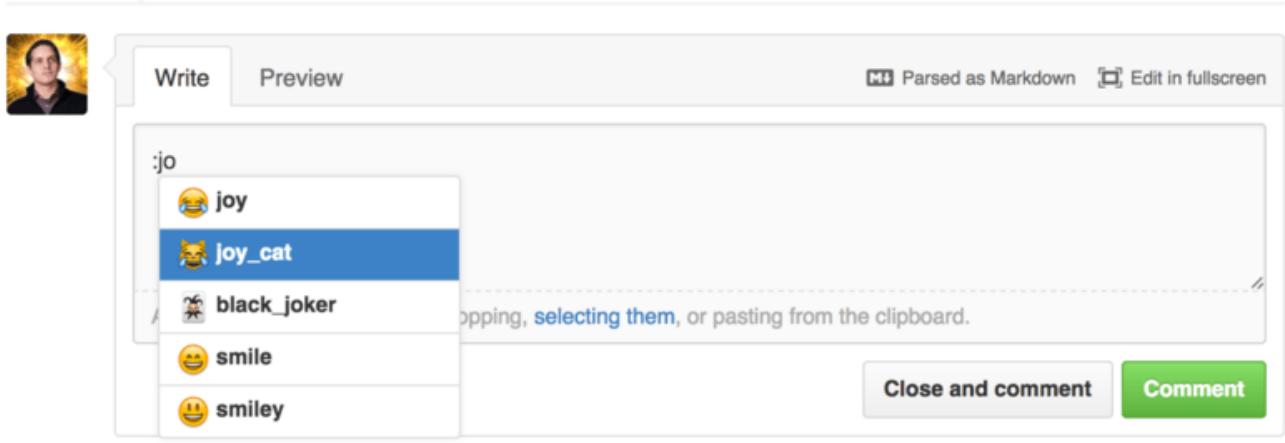
How big are these slings and in particular, these arrows?

```

Figure 105. gerendertes Zitat-Beispiel

## Emojis

Abschließend kannst du auch Emojis in deinen Kommentaren verwenden. Das wird in der Praxis sehr häufig verwendet. Du wirst es in vielen GitHub-Issues und Pull-Requests sehen. Es gibt sogar einen Emoji-Helfer in GitHub. Wenn du einen Kommentar eingibst und mit einem : (Doppelpunkt) beginnst, hilft dir ein Autokomplettierer, das Gesuchte schnell zu finden.



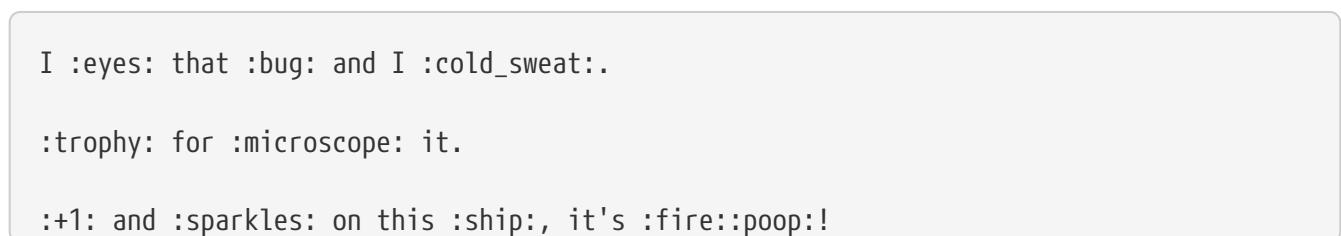
The screenshot shows the GitHub emoji autocompletions interface. A user has typed ':jo' and a dropdown menu appears with suggestions:

- :joy: joy
- :joy\_cat: joy\_cat
- :black\_joker: black\_joker
- :smile: smile
- :smiley: smiley

At the bottom right are two buttons: 'Close and comment' and 'Comment'.

Figure 106. Autokomplettierer für Emojis in Aktion

Emojis haben die Erscheinungsform von :<name>: irgendwo im Kommentar. Zum Beispiel kannst du so etwas schreiben:



The screenshot shows a GitHub comment with the following text:

```

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:

Gerendert würde es in etwa so aussehen: [Massive Emoji-Kommentare](#).

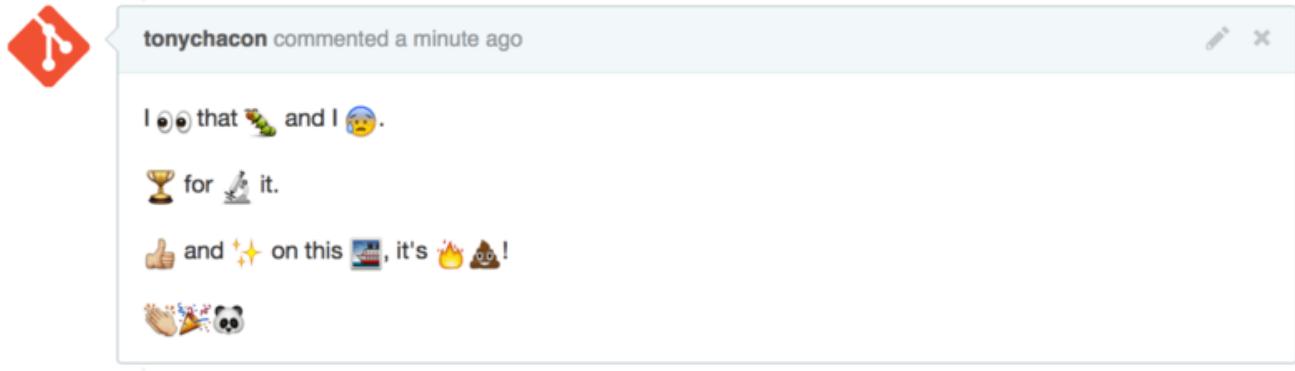


Figure 107. Massive Emoji-Kommentare

Nicht, dass das äußerst sinnvoll wäre, aber es ergänzt ein Medium mit Spaß und Emotionen, was sonst nur schwer zu vermitteln wäre.



Es gibt derzeit eine ganze Reihe von Webservices, die Emoji-Zeichen verwenden. Ein großartiger Spickzettel, zum Nachschlagen, um ein Emoji zu finden, das ausdrückt, was du sagen willst, findest du unter:

<https://www.webfx.com/tools/emoji-cheat-sheet/>

## Bilder

Technisch gesehen ist das keine GitHub-Variante von Markdown, aber es ist sehr praktisch. Neben dem Hinzufügen von Markdown-Bildlinks zu Kommentaren, für die es schwierig sein kann, URLs zum Einbetten zu finden, kannst du mit GitHub Bilder per Drag&Drop in Textbereiche ziehen und so einbinden.

This is the wrong version of Git for the website:



**Git.png**

Attach images by dragging & dropping or [selecting them](#).

**Comment**

---

This is the wrong version of Git for the website:

![git](https://cloud.githubusercontent.com/assets/7874698/4481741/7b87b8fe-49a2-11e4-817d-8023b752b750.png)

Attach images by dragging & dropping or [selecting them](#).

**Comment**

Figure 108. Bilder per Drag&Drop hochladen und automatisch einbetten

Wenn du dir [Bilder per Drag&Drop hochladen und automatisch einbetten](#) ansiehst, wirst du einen kleinen Hinweis, „Parsed as Markdown“, über dem Textfeld sehen. Wenn du darauf klickst, erhältst du einen vollständiges Cheat-Sheet (Spickzettel) mit allem, was du mit Markdown auf GitHub machen kannst.

## Dein öffentliches GitHub-Repository aktuell halten

Sobald du ein GitHub-Repository geforkt hast, existiert dein Repository (dein „fork“) unabhängig vom Original. Insbesondere dann, wenn das ursprüngliche Repository neue Commits bekommen hat, informiert dich GitHub in der Regel durch eine Meldung wie:

This branch is 5 commits behind progit:master.

Aber dein GitHub-Repository wird nie automatisch von GitHub aktualisiert. Das ist etwas, was du selbst tun musst. Glücklicherweise ist das sehr einfach umzusetzen.

Die eine Möglichkeit erfordert keine Konfiguration. Wenn du z.B. von <https://github.com/progit/progit2.git> geforkt hast, kannst du deinen `master` Branch so auf dem neuesten Stand halten:

```
$ git checkout master ①
$ git pull https://github.com/progit/progit2.git ②
$ git push origin master ③
```

- ① Wenn du dich in einem anderen Branch befindest, kehrst du zu `master` zurück
- ② Hole (engl. fetch) Änderungen von <https://github.com/progit/progit2.git> und merge sie in den `master`
- ③ Pushen deinen `master` Branch nach `origin`

Das funktioniert, aber es ist lästig, die Fetch-URL jedes Mal neu eingeben zu müssen. Du kannst diese Arbeit mit ein wenig Konfiguration automatisieren:

```
$ git remote add progit https://github.com/progit/progit2.git ①
$ git fetch progit ②
$ git branch --set-upstream-to=progit/master master ③
$ git config --local remote.pushDefault origin ④
```

- ① Füge das Quell-Repository hinzu und gib ihm einen Namen. Hier habe ich mich entschieden, es `progit` zu nennen.
- ② Hole (engl. fetch) dir eine Referenz zu den Branches von `progit`, genauer gesagt vom `master`.
- ③ Konfiguriere deinen `master` Branch so, dass er von dem `progit` Remote abgeholt (engl. fetch) wird.
- ④ Definiere das standardmäßige Push-Repository auf `origin`

Sobald das getan ist, wird der Workflow viel einfacher:

```
$ git checkout master ①
$ git pull ②
$ git push ③
```

- ① Wenn du dich in einem anderen Branch befindest, kehre zum `master` zurück
- ② Fetche die Änderungen von `progit` und merge sie in `master`
- ③ Pushe deinen `master` Branch nach `origin`

Dieser Herangehensweise kann nützlich sein, aber sie ist nicht ohne Nachteile. Git wird diese Aufgabe gerne im Hintergrund für dich erledigen, aber es wird dich nicht benachrichtigen, wenn du einen Commit zum `master` machst, von `progit` pullst und dann zu `origin` pushst – alle diese Operationen sind mit diesem Setup zulässig. Du musst also darauf achten, nie direkt an den `master` zu committen, da dieser Branch faktisch zum Upstream-Repository gehört.

## Ein Projekt betreuen

Nachdem wir nun zu einem Projekt beitragen können, schauen wir uns die andere Seite an: die Erstellung, Wartung und Verwaltung deines eigenen Projekts.

### Ein neues Repository erstellen

Erstellen wir ein neues Repository, in dem wir unseren Projekt-Code freigeben können. Klicke zunächst auf die Schaltfläche „New repository“ auf der rechten Seite des Dashboards oder auf die

Schaltfläche **+** in der oberen Symbolleiste neben deinem Benutzernamen, wie in [Das Dropdown-Menü „New repository“ zu sehen.](#)

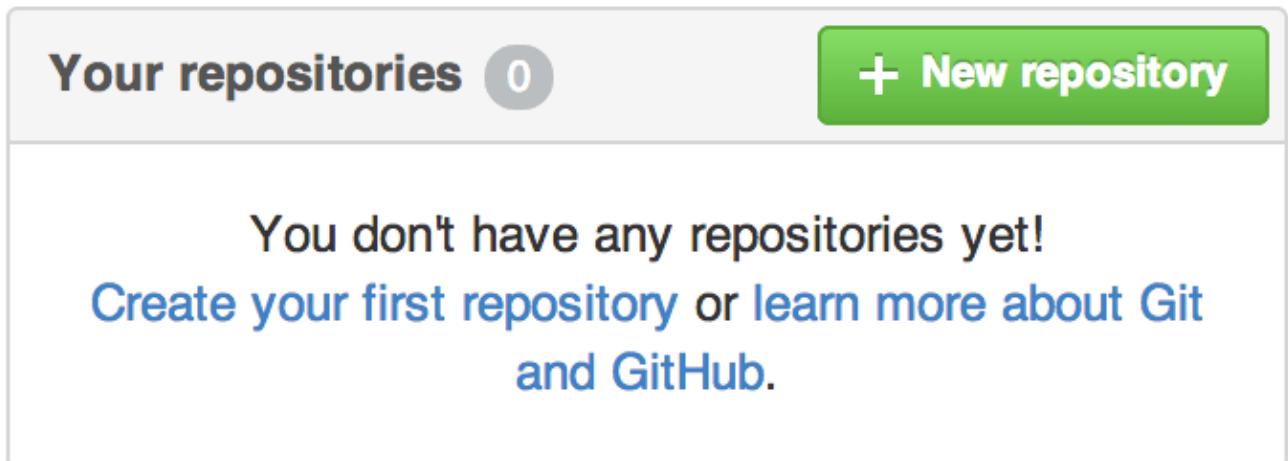


Figure 109. Der Bereich „Your repositories“

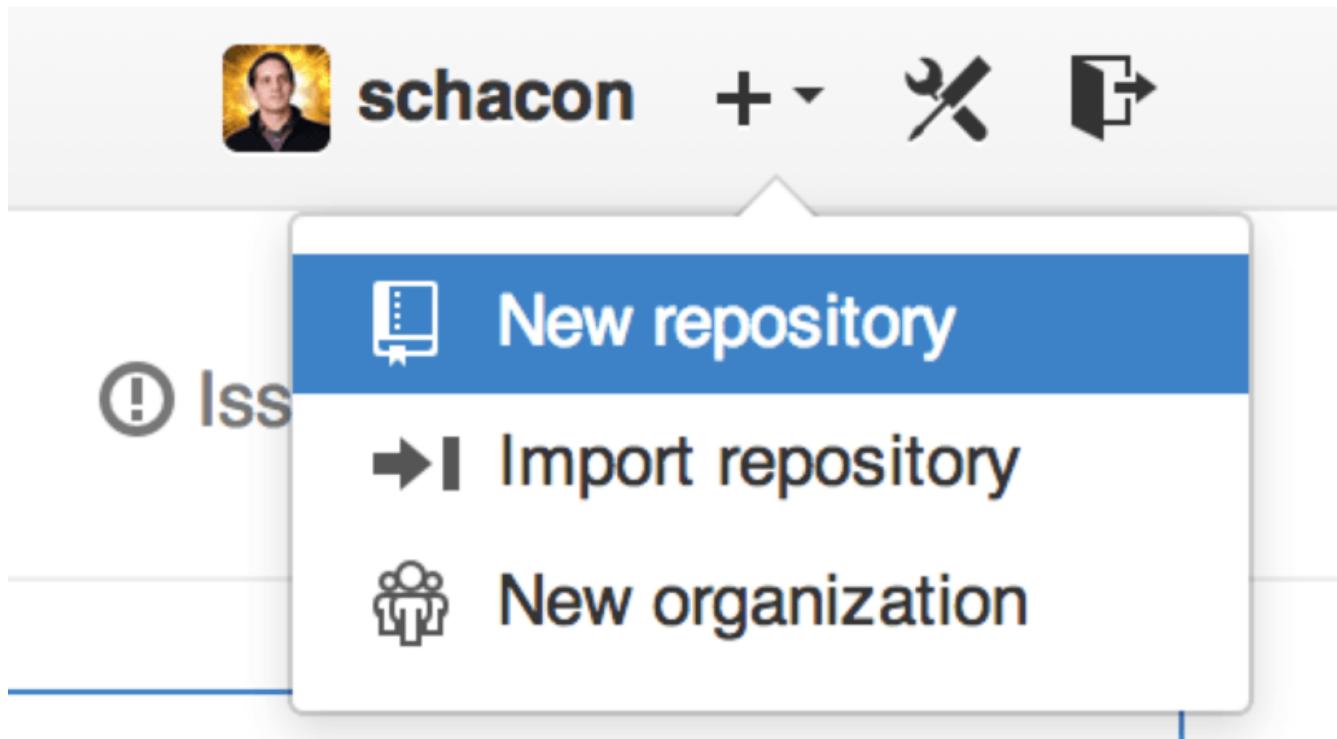


Figure 110. Das Dropdown-Menü „New repository“

Du wirst zum Formular „new repository“ weitergeleitet:

Owner      Repository name

PUBLIC  ben / iOSApp 

Great repository names are short and memorable. Need inspiration? How about [drunken-dubstep](#).

Description (optional)

iOS project for our mobile group

 **Public**  
Anyone can see this repository. You choose who can commit.

 **Private**  
You choose who can see and commit to this repository.

**Initialize this repository with a README**  
This will allow you to `git clone` the repository immediately. Skip this step if you have already run `git init` locally.

Add .gitignore: **None** | Add a license: **None** | ⓘ

**Create repository**

Figure 111. Das Formular „new repository“

Alles, was du hier wirklich tun musst, ist, einen Projektnamen anzugeben. Die restlichen Felder sind optional. Fürs Erste klicke einfach auf die Schaltfläche „Create Repository“, und bäm—du verfügst über ein neues Repository auf GitHub mit dem Namen <User>/<Projekt\_Name>.

Da du dort noch keinen Code vorfindest, zeigt dir GitHub Anleitungen an, wie du ein brandneues Git-Repository einrichten oder es zu einem bestehenden Git-Projekt verbinden kannst. Wir werden das hier nicht weiter vertiefen. Wenn du eine Auffrischung benötigst, schaue dir noch einmal [Kapitel 2, Git Grundlagen](#) an.

Da dein Projekt jetzt auf GitHub gehostet wird, kannst du die URL an jeden weitergeben, mit dem du dein Projekt teilen möchtest. Jedes Projekt auf GitHub ist über HTTPS als [https://github.com/<User>/<Projekt\\_Name>](https://github.com/<User>/<Projekt_Name>) und über SSH als [git@github.com:<User>/<Projekt\\_Name>](git@github.com:<User>/<Projekt_Name>) erreichbar. Git kann von diesen beiden URLs abholen/fetchen und auf sie hochladen/pushen. Auf Basis der Anmeldedaten des Benutzers, der sich verbindet, werden die Zugriffsrechte entsprechend beschränkt.



Häufig ist es sinnvoll, die HTTPS-basierte URL für ein öffentliches Projekt zu verwenden, da der Anwender zum Klonen kein GitHub-Konto haben muss. Wenn User per SSH-URL auf dein Projekt zugreifen wollen, müssen sie über ein GitHub-Konto und einen hochgeladenen SSH-Schlüssel verfügen. Die HTTPS-Adresse ist genau die gleiche URL, die du in einen Browser einfügen würdest, um das Projekt dort anzuzeigen.

## Hinzufügen von Mitwirkenden

Wenn du mit anderen Personen zusammenarbeitest, denen du die Erlaubnis zum Committen gewähren möchtest, musst du diese als „collaborators“ (dt. Mitwirkende) eintragen. Ben, Jeff und Louise besitzen ein GitHub-Konto. Wenn du ihnen Push-Zugriff auf dein Repository gewähren

möchtest, kannst du sie deinem Projekt hinzufügen. Dadurch erhalten sie „Push“-Zugriff, d.h. sie haben sowohl Lese- als auch Schreibzugriff auf das Projekt und das Git-Repository.

Klicke auf den Link „Settings“ unten rechts in der Seitenleiste.

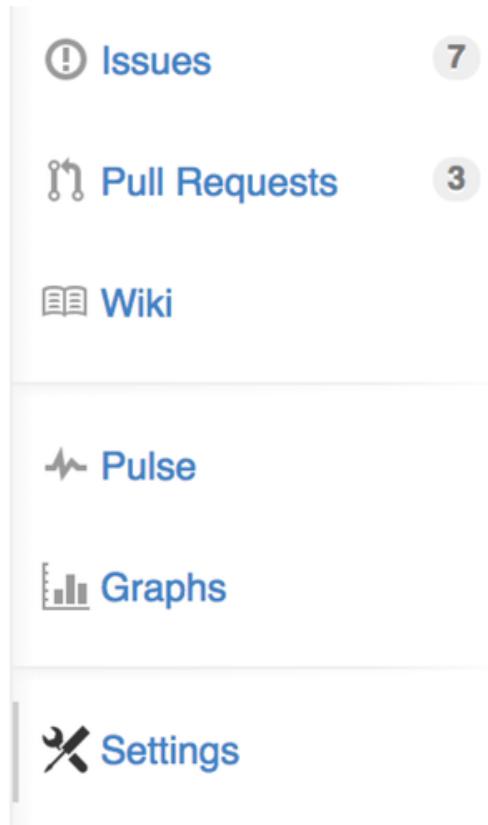


Figure 112. Der Settings-Link des Repositorys

Wähle aus dem Menü auf der linken Seite „Collaborators“. Gib dann einfach einen Benutzernamen in das Feld ein und klicke auf „Add Collaborator“. Du kannst diese Prozedur beliebig oft wiederholen, um so jedem Zugriff zu gewähren. Wenn du die Zugangsberechtigung widerrufen möchtest, klicke einfach auf das "X" auf der rechten Seite der entsprechenden Zeile.

A screenshot of the GitHub 'Collaborators' page. On the left, there's a sidebar with 'Options', 'Collaborators' (which is selected and highlighted in orange), 'Webhooks &amp; Services', and 'Deploy keys'. The main area shows a table of collaborators: Ben Straub (ben) with 'Full access to the repository' and a delete 'X' button; Jeff King (peff); and Louise Corrigan (LouiseCorrigan). At the bottom, there's a search bar 'Type a username' and a 'Add collaborator' button.

Figure 113. Mitwirkende im Repository

## Pull-Requests verwalten

Da es jetzt ein Projekt mit einem Code und vielleicht sogar ein paar Mitwirkenden gibt, die auch Push-Zugriff haben, lass uns noch einmal darüber nachdenken, was zu tun ist, wenn du einen Pull Request erhältst.

Pull-Requests können entweder von einem Branch in einem Fork deines Repositorys kommen oder von einem anderen Branch im selben Repository. Der einzige Unterschied besteht darin, dass Fork-Pull-Requests oft von Personen stammen, auf deren Branch du nicht pushen kannst und sie nicht auf deinen. Bei internen Pull-Requests haben im Allgemeinen beide Parteien Zugriff auf alle Branches.

Für diese Beispiele nehmen wir an, du bist „tonychacon“ und hast ein neues Arduino-Code-Projekt mit der Bezeichnung „fade“ erstellt.

## E-Mail Benachrichtigungen

Irgendjemand bearbeitet deinen Code und sendet dir einen Pull-Request. In diesem Fall solltest du eine E-Mail erhalten, in der du über den neuen Pull-Request informiert wirst. Dieser sollte etwa so aussehen wie in [E-Mail Benachrichtigung über einen neuen Pull-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 114. E-Mail Benachrichtigung über einen neuen Pull-Request

Es gibt ein paar Punkte, die man bei dieser E-Mail beachten sollte. Es gibt ein kleines diffstat – eine Liste von Dateien, die sich im Pull Request geändert haben und um wie sehr sie sich geändert haben. Du erhältst einen Link zum Pull Request auf GitHub. Du erhältst auch ein paar URLs, die du von der Kommandozeile aus verwenden kannst.

Wenn du die Zeile siehst, die `git pull <url> patch-1` lautet, ist das eine einfache Möglichkeit, aus einer entfernten Branch zu mergen, ohne einen Remote hinzufügen zu müssen. Wir haben das in Kapitel 5, [Remote-Banches auschecken](#) kurz besprochen. Wenn du möchtest, kannst du einen Feature-Branch erstellen und in diesen wechseln (engl. checkout). Anschließend kannst du diesen Befehl ausführen, um die Änderungen in dem Pull-Request zu mergen.

Die anderen interessanten URLs sind die `.diff` und `.patch` URLs, die, wie du vielleicht vermutest, vereinheitlichte Diff- und Patch-Versionen des Pull Requests bereitstellen. Technisch gesehen kannst du die Arbeit im Pull-Request mit einem Befehl wie diesem zusammenführen:

```
$ curl https://github.com/tonychacon/fade/pull/1.patch | git am
```

## Mitwirkung beim Pull Request

Wie in [Github Workflow](#) beschrieben, kannst du nun eine Diskussion mit der Person führen, die den Pull Request geöffnet hat. Du kannst bestimmte Code-Zeilen kommentieren, ganze Commits kommentieren oder den gesamten Pull-Request selbst kommentieren, indem du „GitHub Flavored Markdown“ an beliebiger Stelle verwendest.

Jedes Mal, wenn jemand den Pull-Request kommentiert, erhältst du weitere E-Mail-Benachrichtigungen, damit du weißt, dass Aktivitäten stattfinden. Du wirst jeweils einen Link zum Pull Request erhalten, in dem die Aktivität stattfindet. Auf die E-Mails kannst du auch direkt antworten, um den Pull-Request-Thread zu kommentieren.

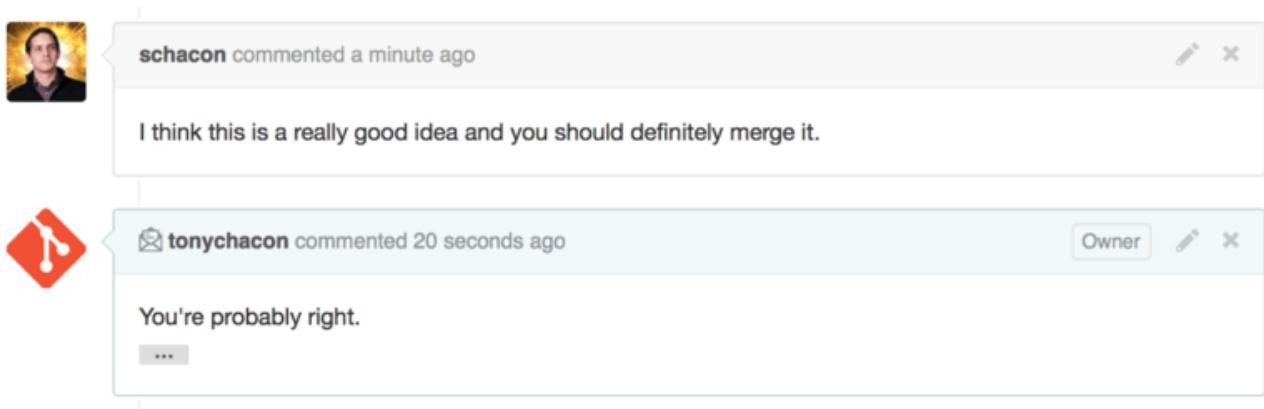


Figure 115. Antworten auf E-Mails sind in den Thread integriert

Sobald der Quellcode soweit korrekt ist und du ihn zusammenführen möchtest, kannst du ihn mit der Anweisung `git pull <url> <branch>` (wie zuvor gesehen) pullen und lokal mergen. Oder du fügst den Fork als Remote hinzu, fetchst den Patch und mergest ihn anschließend.

Wenn das Zusammenführen trivial ist, kannst du auch einfach auf der GitHub-Seite auf die Schaltfläche „Merge“ klicken. Dadurch wird ein „non-fast-forward“ Merge durchgeführt, wodurch ein Merge-Commit erstellt wird, auch wenn ein „fast-forward“ Merge möglich gewesen wäre. Das bedeutet, dass unabhängig davon, was du tust, jedes Mal, wenn du auf die Schaltfläche Merge klickst, ein Merge-Commit erstellt wird. Wie du in [Merge-Button und Anweisungen zum manuellen Zusammenführen eines Pull-Requests](#) sehen kannst, gibt dir GitHub all diese Informationen, wenn du auf den Hinweis-Link klickst.

This pull request can be automatically merged.  
You can also merge branches on the [command line](#).

**Merging via command line**  
If you do not want to use the merge button or an automatic merge cannot be performed, you can perform a manual merge on the command line.

HTTP Git Patch <https://github.com/schacon/fade.git>

**Step 1:** From your project repository, check out a new branch and test the changes.

```
git checkout -b schacon-patch-1 master
git pull https://github.com/schacon/fade.git patch-1
```

**Step 2:** Merge the changes and update on GitHub.

```
git checkout master
git merge --no-ff schacon-patch-1
git push origin master
```

Figure 116. Merge-Button und Anweisungen zum manuellen Zusammenführen eines Pull-Requests

Wenn du dich entscheidest, dass du ihn nicht zusammenführen möchtest, kannst du auch einfach den Pull-Request schließen und die Person, die ihn geöffnet hat, wird benachrichtigt.

### Pull Request Refs (Referenzen)

Wenn du **mehr** viele Pull-Requests erhältst und nicht jedes Mal einen ganzen Batzen Remotes hinzufügen oder einmalige Pulls durchführen willst, gibt es einen tollen Kniff in GitHub. Es handelt sich um einen fortgeschrittenen Trick, den wir in [Kapitel 10, Refspecs](#) ausführlicher durchgehen werden. Er kann jedoch recht nützlich sein.

GitHub beschreibt die Pull-Request-Banches für ein Repository als eine Art Pseudo-Branch auf dem Server. Normalerweise werden sie beim Klonen nicht angezeigt- Sie sind jedoch verdeckt vorhanden und man kann recht einfach darauf zugreifen.

Um das zu verdeutlichen, werden wir den Low-Level-Befehl `ls-remote` verwenden, der oft als Git-Basisbefehl (engl. plumbing) bezeichnet wird und über den wir in [Basisbefehle und Standardbefehle \(Plumbing and Porcelain\)](#) mehr lesen werden. Dieses Kommando wird im Regelfall nicht im täglichen Gebrauch von Git verwendet, aber es ist zweckmäßig, um uns zu zeigen, welche Referenzen auf dem Server vorhanden sind.

Wenn wir diesen Befehl gegen das „blink“ Repository ausführen, das wir vorhin benutzt haben, erhalten wir eine Liste aller Branches und Tags, sowie anderer Referenzen im 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
3c8d735ee16296c242be7a9742ebfb2665adec1      refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d      refs/pull/2/merge
```

```
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a    refs/pull/4/head  
31a45fc257e8433c8d8804e3e848cf61c9d3166c    refs/pull/4/merge
```

Wenn du dich in deinem Repository befindest und `git ls-remote origin` ausführst (oder auch einen beliebigen anderen Remote, den du überprüfen möchtest), dann sieht die Ausgabe ähnlich aus.

Wenn sich das Repository auf GitHub befindet und es offene Pull-Requests gibt, werden diese Referenzen mit vorangestelltem `refs/pull/` angezeigt. Das sind im Prinzip Branches, die aber nicht unter `refs/heads/` stehen. Sie werden normalerweise nicht angezeigt, wenn du klonst oder vom Server fetchst – der Fetching-Prozess ignoriert sie normalerweise.

Es gibt zwei Referenzen pro Pull-Request – eine endet mit `/head`. Sie zeigt auf genau den gleichen Commit wie der letzte Commit im Pull-Request-Branch. Wenn also jemand einen Pull-Request in unserem Repository öffnet und sein Branch `bug-fix` heißt, der auf `a5a775` zeigt, dann haben wir in **unserem** Repository keinen `bug-fix` Branch (weil der in **seinem** Fork liegt). Wir haben jedoch `pull/<pr#>/head`, der auf `a5a775` zeigt. Das heißt, wir können jeden Pull-Request-Branch herunterladen, ohne lokal einen Menge Remotes hinzufügen zu müssen.

Jetzt kannst du das Fetchen dieser Referenz direkt durchführen.

```
$ git fetch origin refs/pull/958/head  
From https://github.com/libgit2/libgit2  
 * branch          refs/pull/958/head -> FETCH_HEAD
```

Damit sagt man Git: „Verbinde dich mit dem `origin` Remote und laden die Referenz `refs/pull/958/head` herunter.“ Git macht das mit Vergnügen und lädt alles herunter, was du brauchst, um diesen Ref zu erstellen. Es setzt einen Zeiger auf den Commit, den du unter `.git/FETCH_HEAD` haben willst. Du kannst die Bearbeitung mit `git merge FETCH_HEAD` in einen Branch fortsetzen. In diesem könntest du die Änderung testen. Die Merge-Commit-Nachricht sieht jedoch ein wenig merkwürdig aus. Wenn du **viele** Pull-Requests überprüfen musst, wird das etwas umständlich.

Es gibt auch eine Möglichkeit wie du *alle* Pull-Requests abrufen und aktuell halten kannst, wann immer du dich mit dem Remote verbindest. Öffne `.git/config` in deinem bevorzugten Editor und suche nach dem `origin` Remote. Es sollte etwa so aussehen:

```
[remote "origin"]  
  url = https://github.com/libgit2/libgit2  
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Die Zeile, die mit `fetch =` beginnt, ist eine „refspec.“ Es ist eine Möglichkeit, Namen auf dem Remote auf Namen in deinem lokalen `.git` Verzeichnis zuzuordnen. Dieser hier sagt git: „die Sachen auf dem Remote, die unter `refs/heads` liegen, sollten in meinem lokalen Repository unter `refs/remotes/origin` abgelegt werden.“ Du kannst diesen Teil ändern, um eine weitere Referenz (refspec) hinzuzufügen:

```
[remote "origin"]
  url = https://github.com/libgit2/libgit2.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  fetch = +refs/pull/*/head:refs/remotes/origin/pr/*
```

Diese letzte Zeile sagt Git: „Alle Referenzen, die wie `refs/pull/123/head` aussehen, sollten lokal als `refs/remotes/origin/pr/123` gespeichert werden.“ Wenn du diese Änderung speicherst und ein `git fetch` ausführst, passiert folgendes:

```
$ 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
# ...
```

Nun werden alle Remote-Pull-Requests lokal mit Referenzen (refs) abgebildet, die sich ähnlich wie das Tracken von Branches verhalten. Sie sind schreibgeschützt und werden aktualisiert, wenn du ein Fetch durchführst. Dadurch ist es besonders einfach, den Code aus einer Pull-Anforderung lokal auszuprobieren:

```
$ 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'
```

Diejenigen mit Adleraugen unter euch werden das `head` am Ende des Remote-Abschnitts der `refspec` bemerkt haben. Es gibt auch ein `refs/pull/#/merge` ref auf der GitHub-Seite, der den Commit darstellt, der sich ergeben würde, wenn du auf die Schaltfläche „merge“ klickst. Auf diese Weise kannst du das Ergebnis des Pull-Requests testen, bevor du überhaupt auf die Schaltfläche geklickt hast.

## Pull-Requests auf Pull-Requests

Du kannst nicht nur Pull-Requests öffnen, die auf den Haupt- oder `master` Branch zeigen. Du kannst Pull-Request, gegen jeden beliebigen Branch im Netzwerk stellen. Du kannst sogar einen anderen Pull-Request als Ziel wählen.

Wenn du einen Pull-Request siehst, der dir gefällt und du eine Idee für eine Änderung hast, die von ihm abhängt; oder du dir nicht sicher bist, ob dieser Pull-Request überhaupt eine gute Idee ist; oder du keine Push-Berechtigung zum Zie-Branch hast – kannst du einen Pull-Request auf diesen anderen Pull-Request öffnen.

Wenn du einen Pull-Request öffnest, befindet sich oben auf der Seite ein Feld, das angibt, von welchem Branch du pullen und in welchen du den Pull-Vorgang ausführen möchtest. Wenn du auf die Schaltfläche „Edit“ (Bearbeiten), rechts neben diesem Feld, klickst, kannst du nicht nur die

Branches, sondern auch den Fork ändern.

The screenshot shows two GitHub pull request comparison pages side-by-side. The top page compares 'schacon:master' and 'tonychacon:patch-2'. It displays 2 commits, 1 file changed, 0 commit comments, and 2 contributors. The bottom page shows a dropdown menu titled 'Choose a base branch' with options like 'master' and 'patch-1'. The 'master' option is selected. This indicates that the user is manually specifying the target fork and branch for the pull request.

Figure 117. Manuelles Ändern der Pull-Request Ziel-Fork und der Branch

Hier kannst du relativ einfach angeben, ob dein neuer Branch in einen anderen Pull-Request oder in einen anderen Fork des Projekts zusammengeführt werden soll.

## Erwähnen und Benachrichtigen

GitHub hat auch ein ziemlich praktisches Benachrichtigungssystem integriert, das bei Fragen oder Rückmeldungen von einzelnen Personen oder Teams eine große Hilfe sein kann.

In jedem Kommentar kannst du mit der Eingabe eines @-Zeichens beginnen und es wird automatisch mit den Namen und Benutzernamen von Personen vervollständigt, die Mitstreiter oder Beitragende des Projektes sind.

The screenshot shows the GitHub comment input interface. The 'Write' tab is active, and the input field contains '@'. A dropdown menu lists suggestions: 'ben Ben Straub', 'peff Jeff King', 'jlehmann Jens Lehmann', and 'LouiseCorrigan Louise Corrigan'. Below the suggestions, a note says 'At', 'selecting them, or pasting from the clipboard.' At the bottom right are 'Close and comment' and 'Comment' buttons.

Figure 118. Mit der Eingabe von @ anfangen, um jemanden zu erwähnen

Du kannst auch einen Benutzer angeben, der sich nicht in diesem Dropdown-Menü befindet, aber oft ist der Auto-Komplettierer schneller.

Sobald du einen Kommentar mit einer Benutzererwähnung postest, wird dieser Benutzer benachrichtigt. Damit ist es möglich, Andere effektiv in Gespräche einbinden, anstatt sie einfach

nur zu befragen. Häufig werden bei Pull-Requests auf GitHub andere Personen in Teams oder Unternehmen einbezogen, um ein Issue oder Pull-Requests zu überprüfen.

Wenn jemand in einem Pull-Request oder Issue erwähnt wird, wird er darauf „abonniert“ (engl. subscribed) und erhält immer dann weitere Benachrichtigungen, wenn eine Aktivität dort stattfindet. Du wirst auch abonniert, wenn der Pull-Request von dir geöffnet wurde, wenn du das Repository beobachten oder wenn du etwas kommentierst. Wenn du keine weiteren Benachrichtigungen mehr erhalten möchtest, kannst du auf die Schaltfläche „Unsubscribe“ klicken, um dich von den Benachrichtigungen abzumelden.

## Notifications



**✖️ Unsubscribe**

You're receiving notifications because you commented.

Figure 119. Von einer Issue- oder Pull-Request-Benachrichtigung abmelden

### Die Benachrichtigungs-Seite

Wenn wir hier „Benachrichtigungen“ (engl. notifications) erwähnen, meinen wir wie GitHub versucht dich zu informieren, falls ein Ereignis eintritt. Es gibt ein paar Einstellungen, die du konfigurieren kannst. Wenn du von der Settings-Seite aus auf die Registerkarte "Notifications" gehst, siehst du einige der verfügbaren Optionen.

Figure 120. Benachrichtigungs-Optionen

Die beiden Möglichkeiten sind: über „E-Mail“ oder „Web“ benachrichtigen. Du kannst entweder eine, keine oder beide Optionen wählen, wenn du dich an den Aktivitäten in Repositorys beteiligst, die du gerade beobachtet.

### Web Benachrichtigungen

Web-Benachrichtigungen landen nur auf GitHub und du kannst sie nur auf GitHub überprüfen. Wenn du diese Option in deinen Einstellungen ausgewählt hast und eine Benachrichtigung für dich ausgelöst wird, siehst du oben auf deinem Bildschirm einen kleinen blauen Punkt über deinem Benachrichtigungssymbol, wie in [Benachrichtigungen](#) zu sehen ist.

Figure 121. Benachrichtigungen

Wenn du darauf klickst, siehst du eine Liste aller Elemente, über die du informiert wurdest, gruppiert nach Projekten. Du kannst nach den Benachrichtigungen eines bestimmten Projekts

filtern, indem du auf dessen Namen in der linken Seitenleiste klickst. Du kannst deine Benachrichtigung als Gelesen markieren indem du das Häkchens neben einer Meldung anklickst oder *alle* Benachrichtigungen in einem Projekt bestätigen indem du das Häkchens oben in der Gruppe anklickst. Es gibt auch eine Mute-Taste neben jedem Häkchen, die du anklicken kannst, um keine weiteren Mitteilungen zu diesem Thema zu erhalten.

Diese Tools sind alle sehr praktisch für die Organisation vieler Benachrichtigungen in GitHub. Viele GitHub Power-User schalten E-Mail-Benachrichtigungen einfach komplett aus und verwalten ihre gesamten Benachrichtigungen über die Web-Oberfläche.

### E-Mail Benachrichtigungen

E-Mail-Benachrichtigungen sind die zweite Variante, mit der du Benachrichtigungen über GitHub erhalten kannst. Wenn du diese Option aktiviert hast, erhältst du E-Mails für jede Mitteilung. Wir haben Beispiele dafür in [E-Mail Benachrichtigungen](#) und [E-Mail Benachrichtigung über einen neuen Pull-Request](#) gesehen. Die E-Mails werden auch nach Thema sortiert, was sehr hilfreich ist, wenn dein E-Mail-Client entsprechend konfiguriert ist.

In den Headern der E-Mails, die GitHub dir sendet, sind auch eine ganze Reihe von Metadaten eingebettet, was bei der Einrichtung angepasster Filter und Regeln sehr nützlich sein kann.

Wenn wir uns zum Beispiel die aktuellen E-Mail-Header ansehen, der in [E-Mail Benachrichtigung über einen neuen Pull-Request](#) angezeigten E-Mail, die an Tony gesendet wurde, werden wir unter den gesendeten Informationen Folgendes sehen:

```
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
```

Es gibt hier noch ein paar interessante Kleinigkeiten. Möchtest du E-Mails zu einem Projekt oder Pull Request hervorheben oder umleiten, erhältst du alle notwendige Daten in **Message-ID** im **<user>/<project>/<type>/<id>** Format. Wenn das, zum Beispiel, ein Issue wäre, dann wäre das Feld **<type>** eher „Issues“ als „Pull“ gewesen.

Die **List-Post** und **List-Unsubscribe** Felder bedeuten, dass du, wenn du einen Mail-Client haben, der das versteht, ganz einfach in die Liste posten oder dich vom Thread „abmelden“ (engl. unsubscribe) kannst. Das wäre im Wesentlichen dasselbe wie das Anklicken des „Mute“ Buttons in Web-Benachrichtigungen oder „Unsubscribe“ auf der Issue- oder Pull-Request-Seite selbst.

Es ist auch wichtig zu wissen, dass, wenn du sowohl E-Mail- als auch Web-Benachrichtigungen aktiviert hast und du die E-Mail-Version der Benachrichtigung liest, die Web-Version auch als gelesen markiert wird, falls du in deinem Mail-Client Bilder erlaubt hast.

## Besondere Dateien

Es gibt ein paar besondere Dateien, die GitHub erkennt, wenn sie in deinem Repository vorhanden sind.

### README

Zuerst ist da die **README** Datei, die in nahezu jedem Dateiformat vorliegen kann, das GitHub als Text erkennt. Zum Beispiel könnte es sich um **README**, **README.md**, **README.asciidoc** usw. handeln. Wenn GitHub eine README-Datei in deinem Projekt findet, wird sie auf der Startseite des Projekts angezeigt.

Viele Teams verwenden diese Datei, um alle relevanten Projektinformationen für Personen zu sammeln, die neu im Repository oder Projekt sind. Dazu gehören in der Regel Sachen wie:

- Wofür ist das Projekt vorgesehen
- Wie wird es konfiguriert und installiert
- Ein Beispiel, wie man es anwendet oder zum Laufen bringt
- Die Lizenz, unter der das Projekt zur Verfügung steht
- Wie man dazu beitragen kann

Da GitHub diese Datei rendern, kannst du Bilder oder Links in sie einbetten, um sie besser verständlich zu machen.

### CONTRIBUTING

Die andere von GitHub erkannte, spezielle Datei ist die Datei **CONTRIBUTING**. Wenn du eine **CONTRIBUTING** Datei mit einer beliebigen Dateiendung verwendest, zeigt GitHub sie wie in [Einen Pull-Request starten, wenn eine CONTRIBUTING-Datei existiert](#) gezeigt an, wenn irgend jemand einen Pull-Request öffnet.

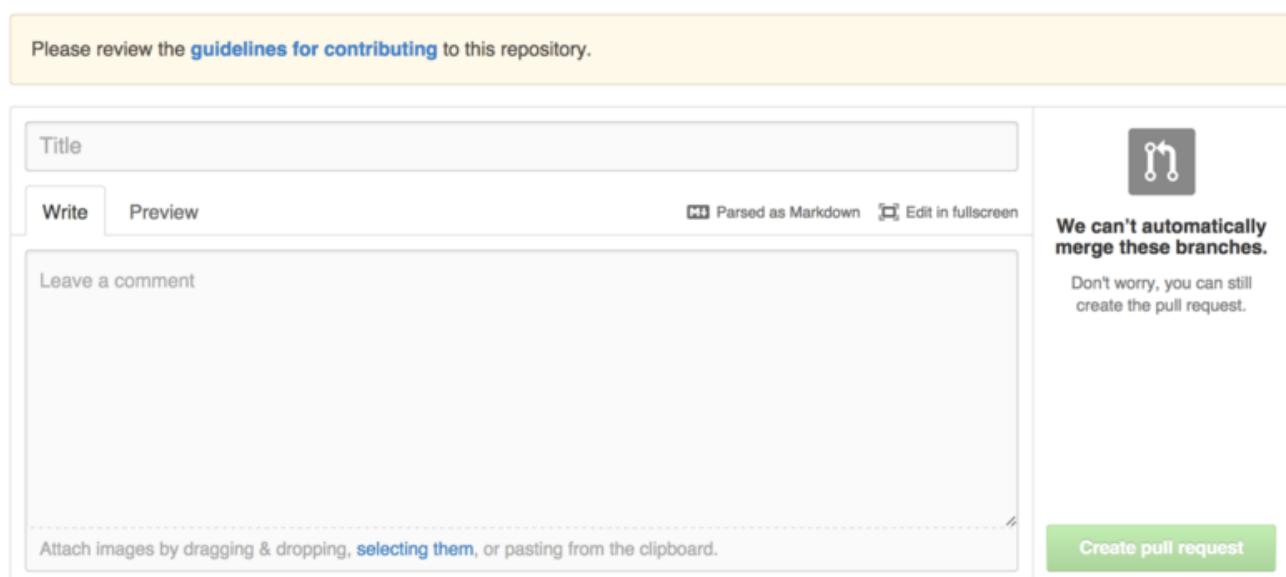


Figure 122. Einen Pull-Request starten, wenn eine CONTRIBUTING-Datei existiert

Die Absicht dabei ist, dass du bestimmte Punkte spezifizieren kannst, die du benötigst oder nicht wünschst, für Pull-Requests, die an deinem Projekt gesendet werden. Auf diese Weise kann ein Benutzer die Leitlinien auch wirklich lesen, bevor er den Pull-Request eröffnet.

## Projekt-Administration

Generell gibt es nur wenige administrative Aufgaben, die du mit einem einzelnen Projekt durchführen kannst, aber ein paar Punkte könnten interessant sein.

### Ändern des Standard-Branchs

Wenn du einen anderen Branch als „master“ als Standard-Branch verwenden willst, auf den die Teilnehmer Pull-Requests öffnen oder ihn standardmäßig sehen sollen, dann kannst du das auf der Settings-Seite deines Repositorys unter der Registerkarte „Optionen“ ändern.

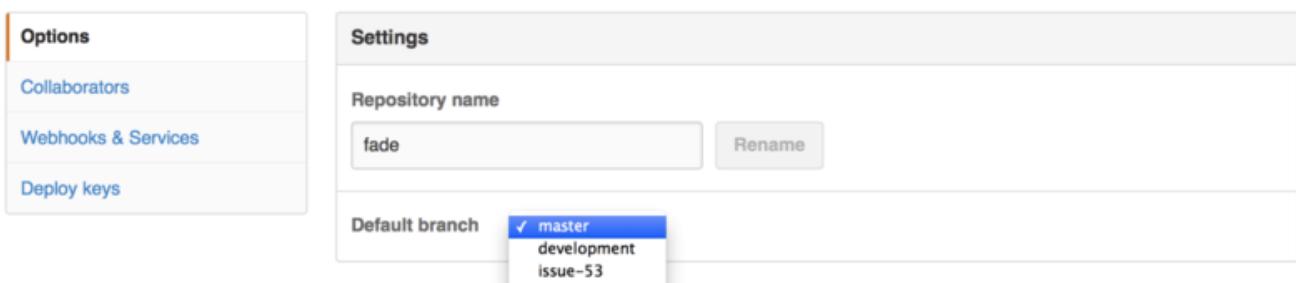


Figure 123. Ändern des Standard-Branchs eines Projekts

Ändere einfach den Standard-Branch in der Dropdown-Liste und das ist dann der Vorgabewert für alle wichtigen Operationen, einschließlich des Branchs, der standardmäßig ausgecheckt wird, wenn jemand das Repository klonnt.

### Übertragen eines Projektes

Wenn du ein Projekt auf einen anderen Benutzer oder eine Organisation in GitHub übertragen möchtest, gibt es unten auf der gleichen Registerkarte „Optionen“ deiner Repository-Einstellungen eine Option „Eigenum übertragen“ (engl. Transfer ownership), die das ermöglicht.

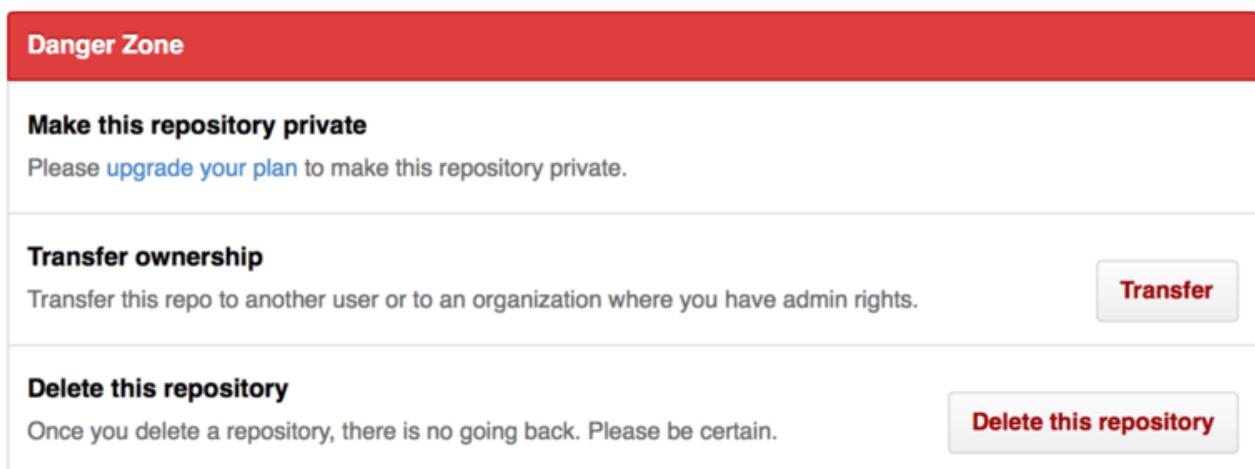


Figure 124. Übertragen eines Projekts auf einen anderen GitHub-User oder eine andere Organisation

Diese Option ist sinnvoll, wenn du ein Projekt aufgibst und es von jemandem übernommen werden soll oder wenn dein Projekt größer wird und du es in eine andere Organisation verlagern möchtest.

Dadurch wird nicht nur das Repository zusammen mit all seinen Beobachtern und Sternen an einen anderen Ort verschoben, sondern es wird auch ein Redirect von deiner URL an den neuen Ort eingerichtet. Es wird auch die Klone und Fetches von Git umleiten, nicht nur die Web-Anfragen.

## Verwalten einer Organisation

Neben den Einzelbenutzer-Konten gibt es bei GitHub auch so genannte Organisationen. Wie bei persönlichen Konten haben auch Organisations-Konten einen Namensraum, in dem alle ihre Projekte gespeichert sind. Aber andere Details sind unterschiedlich. Diese Konten stellen eine Gruppe von Personen dar, die gemeinsam an Projekten beteiligt sind. Es gibt viele Funktionen zum Verwalten ihrer Untergruppen. Normalerweise werden diese Konten für Open-Source-Gruppen (wie „perl“ oder „rails“) oder Unternehmen (wie „google“ oder „twitter“) verwendet.

### Wesentliches zu der Organisation

Eine Organisation ist ziemlich einfach zu erstellen. Klicke einfach auf das „+“ Symbol oben rechts auf jeder GitHub-Seite und wähle „Neue Organisation“ aus dem Menü.

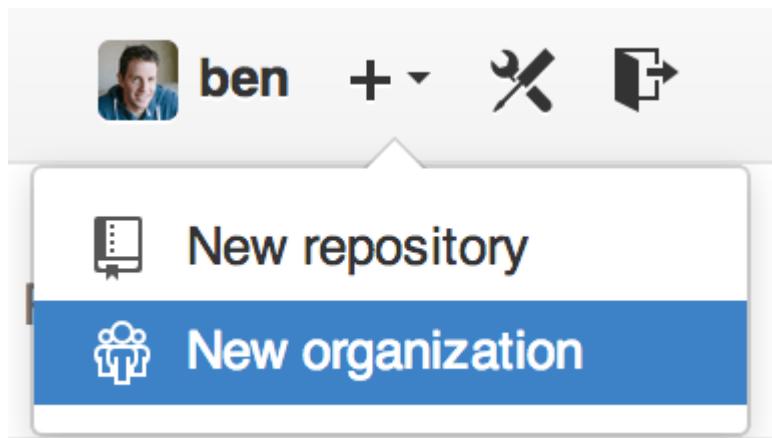


Figure 125. Der Menüpunkt „Neue Organisation“

Zuerst musst du deiner Organisation einen Namen geben und eine E-Mail-Adresse für den Hauptansprechpartner der Gruppe angeben. Dann kannst du andere Benutzer einladen, um Miteigentümer des Accounts zu werden.

Befolge die Anweisungen und du wirst bald Eigentümer einer brandneuen Organisation sein. Wie persönliche Konten sind Organisationen kostenlos, wenn alles, was du dort ablegen willst, Open Source sein wird.

Als Eigentümer in einer Organisation hast du beim Forken eines Repository die Wahl, es in den Namensraum deiner Organisation zu übertragen. Wenn du neue Repositorys erstellst, kannst du diese entweder unter deinem persönlichen Konto oder unter dem einer der Organisationen erstellen, deren Eigentümer du bist. Du „beobachtest“ (engl. watch) auch automatisch jedes neue Repository, das unter diesen Unternehmen erstellt wird.

Wie in [Ihr Avatar-Bild](#) gezeigt, kannst du ein Symbol-Bild für deine Organisation hochladen, um sie

ein wenig zu personalisieren. Wie bei persönlichen Konten hast du auch eine Startseite für die Organisation, die alle deine Repositorys auflistet und von anderen eingesehen werden kann.

Lass uns jetzt einige der Punkte ansprechen, die mit einem Organisationskonto etwas anders sind.

## Teams

Organisationen werden mit einzelnen Personen über Teams verbunden, die lediglich eine Gruppe von einzelnen Benutzer-Accounts und Repositorys innerhalb der Organisation sind. Diese Personen haben unterschiedliche Rechte beim Zugriff in diesen Repositorys.

Angenommen, dein Unternehmen verfügt über drei Repositorys: `frontend`, `backend`, und `deployscripts`. Du möchtest, dass deine HTML/CSS/JavaScript-Entwickler Zugriff auf das Frontend und eventuell das Backend haben und deine Operations-Mitarbeiter Zugriff auf das Backend und die Bereitstellungs-Skripte. Mit Teams ist es einfach, den Beteiligten für jedes einzelne Repository die passende Gruppe zuzuweisen, ohne sie einzeln verwalten zu müssen.

Die Seite Organisation zeigt dir ein übersichtliches Dashboard mit allen Repositorys, Benutzern und Teams, die zu dieser Organisation gehören.

The screenshot shows the GitHub organization page for 'chaconcorp'. On the left, there's a sidebar with a purple logo, the organization name 'chaconcorp', and a gear icon. Below it are sections for 'deployscripts' (scripts for deployment), 'backend' (Backend Code), and 'frontend' (Frontend Code). Each repository has a star rating of 0, 0 reviews, and was updated 16 hours ago. To the right, there are two main panels: 'People' and 'Teams'. The 'People' panel lists three members: dragonchacon (Dragon Chacon), schacon (Scott Chacon), and tonychacon (Tony Chacon), each with a small profile picture. There's also a button to 'Invite someone'. The 'Teams' panel shows three teams: 'Owners' (1 member - 3 repositories), 'Frontend Developers' (2 members - 2 repositories), and 'Ops' (3 members - 1 repository). There's a button to 'Create new team'.

Figure 126. Die Seite Organisation

Um deine Teams zu verwalten, kannst du in [Die Seite Organisation](#) auf die Team-Seitenleiste auf der rechten Seite klicken. So gelangst du zu der Seite, auf der du Mitglieder zum Team hinzufügen, Repositorys zum Team hinzufügen oder die Einstellungen und Zugriffskontrollstufen für das Team verwalten kannst. Jedes Team kann Lesezugriff, Lese-/Schreibzugriff oder administrativen Zugriff

auf die Repositorys haben. Du kannst die Stufe ändern, indem du auf die Schaltfläche „Einstellungen“ in [Die Seite Team](#) klickst.

The screenshot shows the GitHub interface for the 'Frontend Developers' team under the organization 'chaconcorp'. At the top, there are links for 'People 3', 'Teams 3', and 'Audit log'. The 'Members' tab is selected, showing two members: 'tonychacon' (Tony Chacon) and 'schacon' (Scott Chacon). Each member has a small profile picture, their GitHub handle, their name, and a 'Remove' button. Below the members, a note states: 'This team grants Admin access: members can read from, push to, and add collaborators to the team's repositories.' On the left, there's a summary box for the team, showing '2 MEMBERS' and '2 REPOSITORIES', with 'Leave' and 'Settings' buttons.

Figure 127. Die Seite Team

Wenn du einen Benutzer in ein Team einlädst, erhält er eine E-Mail, die ihn darüber informiert, dass er eingeladen wurde.

Zusätzlich funktionieren Team-@mentions (wie `@acmecorp/frontend`) ähnlich wie bei einzelnen Benutzern, nur dass dann **alle** Mitglieder des Teams den Thread abonniert haben. Das ist praktisch, wenn du die Unterstützung von einem Teammitglied wünschst, aber du nicht genau weißt, wen du fragen sollst.

Ein Benutzer kann zu einer beliebigen Anzahl von Teams gehören, also beschränke dich nicht nur auf die Zugriffskontrolle der Teams. Special-Interest-Teams wie `ux`, `css` oder `refactoring` sind für bestimmte Arten von Fragen sinnvoll, andere wie `legal` und `colorblind` für eine völlig andere Kategorie.

## Audit-Logbuch

Organisationen geben den Besitzern auch Zugang zu allen Informationen darüber, was im Rahmen der Organisation vor sich geht. Du kannst auf der Registerkarte [Audit Log](#) sehen, welche Ereignisse auf Organisationsebene stattgefunden haben, wer sie durchgeführt hat und wo in der Welt sie durchgeführt wurden.



Recent events		Filters ▾		
		Search...		
 dragonchacon	added themselves to the <a href="#">chaconcorp/ops</a> team		member	32 minutes ago
 schacon	added themselves to the <a href="#">chaconcorp/ops</a> team			
 tonychacon	invited <a href="#">dragonchacon</a> to the <a href="#">chaconcorp</a> organization		member	33 minutes ago
 tonychacon	invited <a href="#">schacon</a> to the <a href="#">chaconcorp</a> organization			
 tonychacon	gave <a href="#">chaconcorp/ops</a> access to <a href="#">chaconcorp/backend</a>		member	16 hours ago
 tonychacon	gave <a href="#">chaconcorp/frontend-developers</a> access to <a href="#">chaconcorp/backend</a>			
 tonychacon	gave <a href="#">chaconcorp/frontend-developers</a> access to <a href="#">chaconcorp/frontend</a>			16 hours ago
 tonychacon	created the repository <a href="#">chaconcorp/deployscripts</a>			16 hours ago
 tonychacon	created the repository <a href="#">chaconcorp/backend</a>			16 hours ago

Figure 128. Das Audit-Log

Du kannst auch nach bestimmten Ereignissen, bestimmten Orten oder Personen filtern.

## Skripte mit GitHub

Jetzt haben wir alle wichtigen Funktionen und Workflows von GitHub kennengelernt, aber jede große Gruppe oder jedes Projekt wird Anpassungen haben, die sie vornehmen möchte, oder externe Dienste, die sie integrieren möchte.

Glücklicherweise ist GitHub, in vielerlei Hinsicht, ziemlich leicht anzupassen. In diesem Abschnitt erfährst du, wie du das GitHub-Hook-System und seine API verwendest, damit GitHub so funktioniert, wie wir es uns wünschen.

## Dienste und Hooks

Der Bereich Hooks und Services der GitHub-Repository-Administration ist der einfachste Weg, um GitHub mit externen Systemen interagieren zu lassen.

### Dienste

Schauen wir uns zuerst die Services (Dienste) an. Sowohl die Hooks- als auch die Dienste-Integration findest du im Abschnitt Einstellungen deines Repositorys. Dort haben wir uns zuvor mit dem Hinzufügen von Mitwirkenden und dem Ändern des Standard-Branche deines Projekts beschäftigt. Unter der Registerkarte „Webhooks und Dienste“ siehst du so etwas wie [Konfiguration von Diensten und Hooks](#).

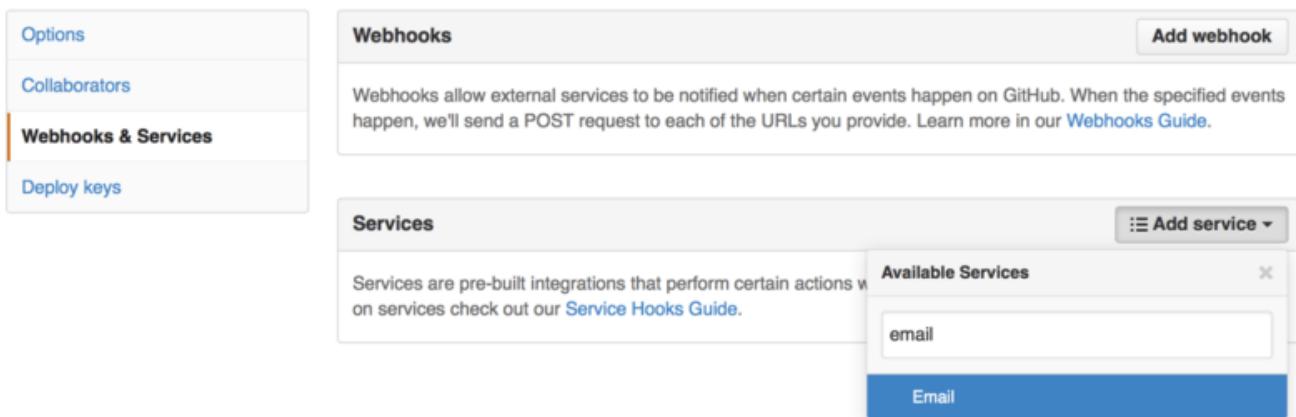


Figure 129. Konfiguration von Diensten und Hooks

Es gibt Dutzende von Diensten, aus denen du wählen kannst. Die meisten davon sind Integrationen in andere kommerzielle und Open-Source-Systeme. Die meisten von ihnen betreffen kontinuierliche Integrationsdienste (engl. Continuous-Integration-Services), Bug- und Issue-Tracker, Chatroom-Systeme und Dokumentationssysteme. Wir werden uns mit der Konfiguration eines sehr einfachen Systems befassen, dem E-Mail-Hook. Wenn du „E-Mail“ aus der Auswahlliste „Add Service“ wählst, erhältst du einen Konfigurationsbildschirm wie unter [E-Mail-Service-Konfiguration](#).

The screenshot shows the 'Services / Add Email' configuration page on GitHub. On the left, a sidebar menu lists 'Options', 'Collaborators', 'Webhooks & Services' (which is selected and highlighted in orange), and 'Deploy keys'. The main content area has a header 'Install Notes' with three numbered points: 1. address whitespace separated email addresses (at most two), 2. secret fills out the Approved header to automatically approve the message in a read-only or moderated mailing list, and 3. send\_from\_author uses the commit author email address in the From address of the email. Below this are fields for 'Address' (containing 'tchacon@example.com'), 'Secret' (an empty field), and a checkbox for 'Send from author' (unchecked). A section labeled 'Active' contains the text 'We will run this service when an event is triggered.' followed by a checked checkbox. At the bottom is a green 'Add service' button.

Figure 130. E-Mail-Service-Konfiguration

Wenn wir in diesem Fall auf die Schaltfläche „Dienst hinzufügen“ klicken, erhält die von uns angegebene Mail-Adresse jedes Mal eine E-Mail, wenn jemand in das Repository pusht. Dienste können auf viele verschiedene Arten von Ereignissen lauschen, aber die meisten sind ausschließlich auf Push-Events spezialisiert und bearbeiten diese Daten dann.

Wenn es ein System gibt, das du verwendest und das du mit GitHub integrieren möchtest, solltest du hier überprüfen, ob es eine bestehende Service-Integration gibt. Angenommen, du verwendest Jenkins, um auf deiner Code-Basis Tests durchzuführen. Du kannst die eingebaute Service-Integration von Jenkins aktivieren, um jedes Mal einen Testlauf zu starten, wenn jemand in dein Repository pusht.

## Hooks

Wenn du eine speziellere Lösung benötigst oder mit einem Dienst oder einer Website interagieren möchtest, der nicht in dieser Liste enthalten ist, kannst du stattdessen das generischere Hooks-System verwenden. GitHub Repository-Hooks sind denkbar einfach. Gib eine URL an und GitHub wird bei jedem gewünschten Event über HTTP Nutz-Daten an diese URL senden.

Im Regelfall kannst du einen kleinen Webservice einrichten, um nach einem GitHub-Hook-Inhalt zu suchen und dann die empfangenen Daten weiter zu verarbeiten.

Um einen Hook zu aktivieren, klicke in [Konfiguration von Diensten und Hooks](#) auf die Schaltfläche „Webhook hinzufügen“. Das führt dich zu einer Seite, die wie [Web-Hook Konfiguration](#) aussieht.

The screenshot shows the GitHub settings interface for managing webhooks. 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' field is populated with 'https://example.com/postreceive'. The 'Content type' dropdown is set to 'application/json'. A 'Secret' field is present but empty. Below these, a section asks 'Which events would you like to trigger this webhook?' with three options: 'Just the push event.' (selected), 'Send me everything.', and 'Let me select individual events.'. A checked 'Active' checkbox indicates the hook will deliver event details when triggered. At the bottom is a green 'Add webhook' button.

Figure 131. Web-Hook Konfiguration

Die Konfiguration für einen Web-Hook ist relativ einfach. In den meisten Fällen gibst du einfach eine URL und einen geheimen Schlüssel ein und klicken auf „Webhook hinzufügen“. Es gibt ein paar Optionen, bei denen GitHub veranlasst wird dir eine Payload zu senden – die Vorgabe ist, eine Payload nur für das **push** Ereignis senden, wenn jemand neuen Code in einen beliebigen Branch deines Repositorys schiebt.

Schauen wir uns ein kleines Beispiel für einen Webservice an, den du für die Verwaltung eines Web-Hooks einrichten kannst. Wir verwenden das Ruby Web-Framework Sinatra, da es relativ übersichtlich ist und du leicht sehen kannst, was wir tun.

Nehmen wir an, wir wollen eine E-Mail erhalten, wenn eine bestimmte Person zu einem bestimmten Branch unseres Projekts pusht und eine bestimmte Datei ändert. Mit einem solchen Code könnten wir das ziemlich einfach machen:

```
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 nehmen wir den JSON-Inhalt, den GitHub uns liefert, und schauen nach, wer sie zu welchem Branch gepusht hat und welche Dateien bei allen Commits, die gepusht wurden, angefasst wurden. Dann überprüfen wir das anhand unserer Kriterien und senden eine E-Mail, wenn sie den Anforderungen entspricht.

Um so etwas zu entwickeln und zu testen, hast du eine ansprechende Entwicklerkonsole auf dem gleichen Bildschirm, auf dem du den Hook eingerichtet hast. Du kannst die jüngsten Aktualisierungen sehen, die GitHub für diesen Webhook vorgenommen hat. Für jeden Hook kannst du nachvollziehen, wann er zugestellt wurde, ob er erfolgreich war und Body und Header für Anfrage (engl. request) und Antwort (engl. response) prüfen. Das ermöglicht ein unglaublich einfaches Testen und Debuggen deines Hooks.

**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 132. Web-Hook Debug Information

Das andere großartige Feature ist, dass du jede der Payloads neu ausliefern kannst, um deinen Service einfach zu testen.

Weitere Informationen wie man Webhook schreiben kann und welche Event-Typen man überwachen kann, findest du in der GitHub-Developer-Dokumentation unter <https://developer.github.com/webhooks/>.

## Die GitHub API

Dienste und Hooks bieten die die Möglichkeit, Push-Benachrichtigungen über Ereignisse zu erhalten, die in deinem Repository stattfinden. Aber was ist, wenn du weitere Informationen über diese Ereignisse benötigst? Was ist, wenn du eine Automatisierung benötigst, wie z.B. das

Hinzufügen von Mitwirkenden oder das Labeln von Issues?

Hier kommt die GitHub API zum Zug. GitHub verfügt über eine Vielzahl von API-Endpunkten, um fast alles zu automatisieren, was du manuell auf der Website tun kannst. In diesem Abschnitt erfahren wir, wie man sich authentifiziert und mit der API verbündet, wie man ein Issue kommentiert und wie man den Status eines Pull-Requests über die API ändert.

## Grundlegende Anwendung

Die elementarste Aufgabe, die du lösen kannst, ist eine einfache GET-Anfrage an einen Endpunkt, der keine Authentifizierung erfordert. Das kann ein Benutzer- oder Lese-Informationen in einem Open-Source-Projekt sein. Wenn wir beispielsweise mehr über einen Benutzer mit Namen „schacon“ erfahren möchten, können wir so etwas verwenden:

```
$ 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"
}
```

Es gibt unzählige Endpunkte wie diesen, um Informationen über Organisationen, Projekte, Issues, Commits zu erhalten – so ziemlich alles, was du öffentlich auf GitHub sehen kannst. Du kannst die API sogar verwenden, um beliebige Markdown-Funktionen zu rendern oder eine [.gitignore](#) Vorlage zu finden.

```
$ curl https://api.github.com/gitignore/templates/Java
{
  "name": "Java",
  "source": "*.class

# Mobile Tools for Java (J2ME)
.mtj.tmp/

# Package Files #
*.jar
*.war
*.ear

# virtual machine crash logs, see
https://www.java.com/en/download/help/error_hotspot.xml
hs_err_pid*
"
```

}

## Ein Issue kommentieren

Wenn du jedoch eine Aktivität auf der Website durchführen möchtest, wie z.B. einen Kommentar zu einem Issue oder Pull Request oder wenn du private Inhalte einsehen oder mit diesen interagieren möchtest, musst du sich authentifizieren.

Es gibt mehrere Möglichkeiten, sich zu authentifizieren. Du kannst die Basisauthentifizierung nur mit deinem Benutzernamen und Passwort verwenden, aber im Allgemeinen ist es eine bessere Idee, einen persönlichen Zugriffstoken zu verwenden. Du kannst den über die Registerkarte „Anwendungen“ auf deiner Einstellungsseite generieren.

The screenshot shows the GitHub Settings page for the user 'tonychacon'. On the left, there's a sidebar with links like Profile, Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications (which is selected), Repositories, and Organizations. Below the sidebar is another user profile for 'chaconcorp'. The main content area has several sections: 'Developer applications' (with a link to 'Register new application'), 'Personal access tokens' (with a link to 'Generate new token' and a note about their function), 'Authorized applications' (noting 'You have no applications authorized to access your account.'), and 'GitHub applications' (listing 'GitHub Team' with a 'Last used on Oct 6, 2014' timestamp and a 'Revoke' button).

Figure 133. Generieren eines Zugriffstokens auf der Registerkarte „Anwendungen“ der Settings-Seite

Du wirst gefragt, welchen Geltungsbereich du für dieses Token möchtest und es wird eine Beschreibung angezeigt. Achte darauf, eine gute Beschreibung zu verwenden, damit du dir sicher bist, das Token entfernen zu können, wenn dein Skript oder deine Anwendung nicht mehr verwendet wird.

GitHub zeigt dir den Token nur ein einziges Mal an, also kopiere und speicher ihn sorgfältig ab. Du kannst diese Funktion nun verwenden, um dich in deinem Skript zu authentifizieren, anstatt einen Benutzernamen und ein Passwort zu verwenden. Das ist angenehm, weil du den Umfang dessen, was du tun möchtest, einschränken kannst und das Token widerruflich ist.

Das hat auch den Vorteil, dass die Abfrage-Rate erhöht wird. Ohne Authentifizierung bist du auf 60 Anfragen pro Stunde beschränkt. Wenn du dich authentifizierst, kannst du bis zu 5.000 Anfragen pro Stunde stellen.

Also nutzen wir es, um einen Kommentar zu einem unserer Issues abzugeben. Nehmen wir an, wir wollen einen Kommentar zu einem bestimmten Problem, Issue #6, abgeben. Dazu müssen wir

einen HTTP POST Request an `repos/<user>/<repo>/issues/<num>/comments` mit dem Token stellen, den wir gerade als Autorisierungs-Header generiert haben.

```
$ 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:"
}
```

Wenn du jetzt zu diesem Issue gehst, kannst du den Kommentar sehen, den wir gerade erfolgreich gepostet haben, wie in [Kommentar, veröffentlicht von der GitHub API](#) zu sehen ist.

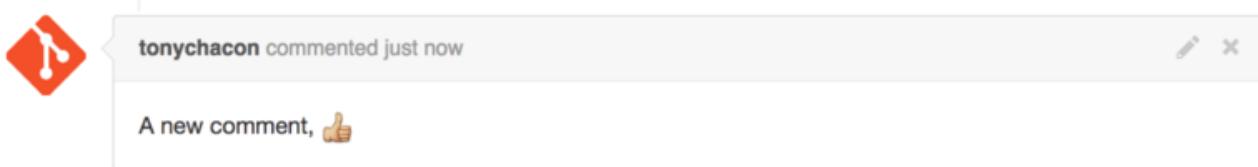


Figure 134. Kommentar, veröffentlicht von der GitHub API

Du kannst die API verwenden, um so ziemlich alles zu tun, was du auf der Website tun kannst – das Erstellen und Setzen von Meilensteinen, das Zuweisen von Personen zu Issues und Pull-Requests, das Erstellen und Ändern von Labels, auf Commit-Daten zugreifen, das Erstellen neuer Commits und Branches, das Öffnen, Schließen oder Mergen von Pull-Requests, das Erstellen und Bearbeiten von Teams, das Kommentieren von Code-Zeilen in einem Pull-Request, das Durchsuchen der Website und so weiter und so fort.

## Den Status eines Pull-Requests ändern

Ein abschließendes Beispiel werden wir uns ansehen, da es wirklich praktisch ist, wenn du mit Pull-Requests arbeitest. Jeder Übertragung können ein oder mehrere Zustände zugeordnet sein. Es gibt eine API für das Hinzufügen und Abfragen dieser Stati.

Die meisten der Dienste für kontinuierliche Integration und Tests nutzen diese API, um auf Pushes zu reagieren, indem sie den Code testen, der gepusht wurde, und dann Bericht erstatten, wenn dieser Commit alle Tests bestanden hat. Du kannst damit auch überprüfen, ob die Commit-Nachricht korrekt formatiert ist, ob der Einreicher alle deine Contributions-Richtlinien befolgt hat,

ob die Übertragung gültig signiert wurde – und vieles mehr.

Angenommen, du richtest einen Webhook in deinem Repository ein, der einen kleinen Webdienst aufruft, der in der Commit-Nachricht nach einer Zeichenkette **Signed-off-by** sucht.

```
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
```

Das ist hoffentlich relativ einfach zu verstehen. In diesem Web-Hook-Handler schauen wir uns jeden Commit an, der gerade gepusht wurde, wir suchen nach der Zeichenkette 'Signed-off-by' in der Commit-Nachricht und POST(en) via HTTP den Status an den API-Endpunkt [/repos/<user>/<repo>/statuses/<commit\\_sha>](#).

In diesem Fall kannst du einen Zustand ('success', 'failure', 'error'), eine Beschreibung des Geschehens, eine Ziel-URL, auf die der Benutzer für weitere Informationen zugreifen kann, und einen „Kontext“ senden, falls es mehrere Zustände für einen einzelnen Commit gibt. So kann beispielsweise ein Testdienst einen Status liefern und ein Validierungsdienst wie dieser ebenfalls einen Status – das Feld „Kontext“ zeigt, wie sie sich voneinander unterscheiden.

Wenn jemand einen neuen Pull-Request auf GitHub öffnet und dieser Hook eingerichtet ist, siehst du vielleicht etwas wie [Commit-Status via API](#).

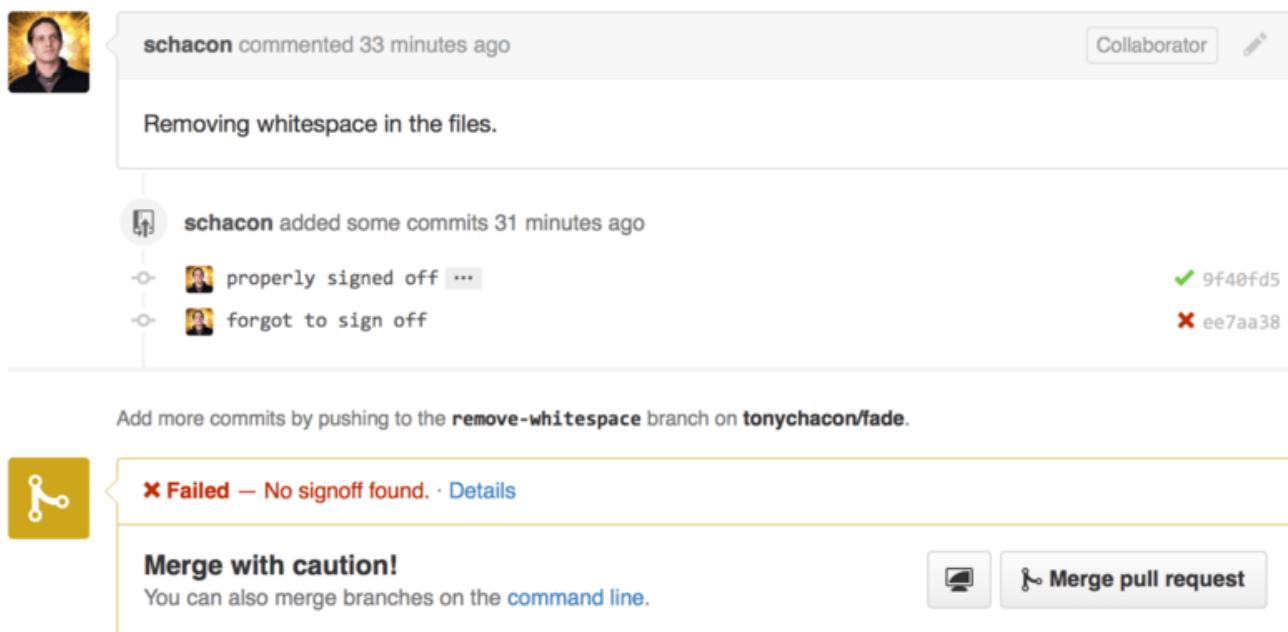


Figure 135. Commit-Status via API

Du siehst nun ein kleines grünes Häkchen neben dem Commit, der in der Nachricht die Zeichenkette „Signed-off-by“ enthält und ein rotes Kreuz, wenn der Autor vergessen hat, den Commit zu signieren. Du kannst auch sehen, dass der Pull-Request den Status des letzten Commits auf dem Branch annimmt und dich warnt, falls es einen Fehler gibt. Das ist besonders nützlich, wenn du diese API für Prüfergebnisse verwendest, damit du nicht versehentlich etwas zusammenführst, bei dem der letzte Commit die Tests nicht besteht.

## Octokit

Obwohl wir in diesen Beispielen fast alles durch `curl` und einfache HTTP-Requests gemacht haben, gibt es mehrere Open-Source-Bibliotheken, die diese API auf eine eigenständigere Form verfügbar machen. Zum Zeitpunkt des Entstehen dieses Buchs umfassen die unterstützten Sprachen Go, Objective-C, Ruby und .NET. Besuche <https://github.com/octokit> für weitere Informationen zu diesen Themen, da sie einen großen Teil des HTTP-Codes für dich übernehmen.

Hoffentlich können dir diese Tools helfen, GitHub anzupassen und zu modifizieren, um sich so besser an deinem individuellen Workflows anzupassen. Eine vollständige Dokumentation der gesamten API sowie Anleitungen für häufige Aufgaben findest du unter <https://developer.github.com>.

# Zusammenfassung

Du bist nun ein GitHub-User. Du weißt, wie man ein Konto einrichtet, eine Organisation (in GitHub) verwaltet, Repositorys erstellt und betreibt, zu Projekten Anderer beiträgt und Beiträge von Anderen entgegennimmt. Im nächsten Kapitel lernst du weitere mächtige Werkzeuge und Tipps für den Umgang mit komplexen Situationen kennen, die dich zu einem kompetenten Git-Experten machen werden.

# Git Tools

Inzwischen hast du die meisten der gängigen Befehle und Workflows kennen gelernt. Du benötigst sie, um ein Git-Repository für deine Quellcode-Kontrolle zu verwalten oder zu pflegen. Du hast die grundlegenden Aufgaben des Tracking und Committens von Dateien gelernt und du hast die Leistungsfähigkeit der Staging-Area und Branching- und Merging-Funktionen mit Feature-Banches genutzt.

Jetzt wirst du eine Reihe von anderen nützlichen Git-Funktionen entdecken. Du wirst diese nicht unbedingt im Alltag einsetzen müssen, aber vielleicht irgendwann einmal benötigen.

## Revisions-Auswahl

Es gibt eine Reihe von Wegen um auf einen einzelnen Commit, einen Satz von Commits oder einen Bereich von Commits zu verweisen. Nicht alle sind offensichtlich, aber es ist nützlich sie zu kennen.

### Einzelne Revisionsstände

Du kannst dich natürlich auf jeden einzelnen Commit mit seinem vollen, 40-stelligen SHA-1-Hash beziehen, aber es gibt auch benutzerfreundlichere Möglichkeiten, sich auf Commits zu beziehen. Dieses Kapitel beschreibt die verschiedenen Möglichkeiten, wie du auf jeden Commit verweisen kannst.

### Kurz-SHA-1

Git ist intelligent genug, um herauszufinden, auf welchen Commit du dich beziehst, wenn du die ersten paar Zeichen des SHA-1-Hash angibst, solange dieser Teil-Hash mindestens vier Zeichen lang und eindeutig ist. D.h. kein anderes Objekt in der Objektdatenbank darf einen Hash haben, der mit dem gleichen Präfix beginnt.

Wenn du z.B. einen bestimmten Commit untersuchen möchtest, kannst du den Befehl `git log` ausführen, um den Commit zu finden:

```
$ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
```

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Dec 11 14:58:32 2008 -0800

Add some blame and merge stuff

Angenommen, du bist an dem Commit interessiert, dessen Hash mit `1c002dd…` beginnt. Du kannst den Commit mit einer der folgenden Varianten von `git show` überprüfen (vorausgesetzt, die verkürzten Hash-Versionen sind eindeutig):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b  
$ git show 1c002dd4b536e7479f  
$ git show 1c002d
```

Git kann eine eindeutige Abkürzung für deine SHA-1-Hashs ermitteln. Wenn du `--abbrev-commit` mit dem Befehl `git log` nutzt, verwendet die Ausgabe kürzere und eindeutige Hash-Werte. Standardmäßig werden sieben Zeichen verwendet. Diese werden jedoch bei Bedarf länger, um die Eindeutigkeit des SHA-1-Hashs zu gewährleisten:

```
$ git log --abbrev-commit --pretty=oneline  
ca82a6d Change the version number  
085bb3b Remove unnecessary test code  
a11bef0 Initial commit
```

In der Regel sind acht bis zehn Zeichen mehr als genug, um innerhalb eines Projekts eindeutig zu sein. Zum Beispiel hat der Linux-Kernel (ein ziemlich großes Projekt) seit Februar 2019 über 875.000 Commits und fast sieben Millionen Objekte in seiner Objektdatenbank, wobei keine zwei Objekte vorhanden sind, deren SHA-1s in den ersten 12 Zeichen identisch sind.

#### *Eine kurze Anmerkung zu SHA-1*

Viele Leute machen sich zu einem bestimmten Zeitpunkt Sorgen, dass sie zufällig zwei verschiedene Objekte in ihrem Repository haben könnten, die den gleichen SHA-1-Wert haben. Was dann?

Wenn du ein Objekt, das auf den gleichen SHA-1-Wert wie ein vorhergehendes *unterschiedliches* Objekt in deinem Repository hasht, committest, wird Git das vorhergehende Objekt bereits in deiner Git-Datenbank sehen und davon ausgehen, dass es bereits geschrieben wurde und es einfach wiederverwenden. Wenn du versuchst, dieses Objekt irgendwann wieder auszuchecken, erhältst du immer die Daten des ersten Objekts.

Du solltest dir jedoch bewusst sein, wie unwahrscheinlich dieses Szenario ist. Der SHA-1 Hashwert beträgt 20 Bytes oder 160 Bit. Die Anzahl der zufällig gehaschten Objekte, die benötigt werden, um eine 50%ige Wahrscheinlichkeit einer einzelnen Kollision zu erreichen, beträgt etwa  $2^{80}$  (Die Formel zur Bestimmung der Kollisionswahrscheinlichkeit ist  $p = (n(n-1)/2) * (1/2^{160})$ ).  $2^{80}$  sind  $1.2 \times 10^{24}$  oder 1 Million Milliarden Milliarden. Das ist das 1.200-fache der Anzahl der

Sandkörner auf der Erde.

Hier ist ein Beispiel, um dir eine Vorstellung davon zu geben, was nötig wäre, um eine SHA-1-Kollision zu erhalten. Wenn alle 6,5 Milliarden Menschen auf der Erde programmierten würden und jeder jede Sekunde soviel Code produzieren würde, die der Menge des gesamten Verlaufs des Linux-Kernels entspräche (6,5 Millionen Git-Objekte) und dann alles in ein riesiges Git-Repository geschoben wird, dann würde es etwa 2 Jahre dauern, bis dieses Repository genügend Objekte enthielte, um eine 50%ige Wahrscheinlichkeit einer einzelnen SHA-1-Objektkollision zu erzielen. Somit ist eine organische SHA-1-Kollision unwahrscheinlicher, als wenn jedes Mitglied deines Programmierer-Teams in der gleichen Nacht von Wölfen angegriffen und auf eine andere Art in der selben Nacht getötet würde.

Wenn du Rechenleistung im Wert von mehreren Tausend US-Dollar dafür bereitstellst, kannst du zwei Dateien mit demselben Hash künstlich generieren, wie im Februar 2017 unter <https://shattered.io/> nachgewiesen wurde. Git geht dazu über, SHA256 als Standard-Hashing-Algorithmus zu verwenden, der gegenüber Kollisionsangriffen wesentlich widerstandsfähiger ist und Code beinhaltet, um diesen Angriff abzuschwächen (obwohl er nicht vollständig beseitigt werden kann).

## Branch Referenzen

Eine unkomplizierte Methode, auf einen bestimmten Commit zu verweisen, ist, wenn es sich um den Commit am Ende eines Branches handelt. In diesem Fall kannst du einfach den Branch-Namen in jedem Git-Befehl verwenden, der eine Referenz auf einen Commit erwartet. Wenn du beispielsweise das letzte Commit-Objekt in einem Branch untersuchen möchtest, sind die folgenden Befehle gleichwertig, vorausgesetzt, der Branch `topic1` zeigt auf den Commit `ca82a6d…`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949  
$ git show topic1
```

Wenn du sehen willst, auf welchen spezifischen SHA-1 ein Branch zeigt, oder wenn du sehen willst, worauf sich die folgenden Beispiele in Bezug auf SHA-1s verkürzen, kannst du ein Git Basis-Befehl (engl. plumbing tool) mit dem Namen `rev-parse` verwenden. Du kannst in [Git Interna](#) weitere Details über Basisbefehl-Tools nachlesen. Der Befehl `rev-parse` ist eine Low-Level-Befehl und ist normalerweise nicht für den täglichen Einsatz notwendig. Allerdings kann es gelegentlich hilfreich sein, wenn man herausfinden muss, was eigentlich passiert ist. So kannst du `rev-parse` auf deinem Branch ausführen.

```
$ git rev-parse topic1  
ca82a6dff817ec66f44342007202690a93763949
```

## RefLog Kurzformen

Eine der Dinge, die Git im Hintergrund macht, während du arbeitest, ist einen „Reflog“

aufzuzeichnen – ein Protokoll darüber, wo sich deine HEAD- und Branch-Referenzen in den letzten Monaten befunden haben.

Du kannst dein Reflog sehen, indem du `git reflog` benutzt:

```
$ git reflog
734713b HEAD@{0}: commit: Fix refs handling, add gc auto, update tests
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the 'recursive' strategy.
1c002dd HEAD@{2}: commit: Add some blame and merge stuff
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD
95df984 HEAD@{4}: commit: # This is a combination of two commits.
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Jedes Mal, wenn das Ende deines Branch aus irgendeinem Grund aktualisiert wird, speichert Git diese Informationen für dich in dieser temporären Historie. Du kannst deine Reflog-Daten auch verwenden, um auf ältere Commits zu verweisen. Wenn du beispielsweise den fünft-letzten Wert des HEADs deines Repositorys sehen möchtest, kannst du den Verweis `@{5}` benutzen, damit du diese Reflog-Ausgabe erhältst:

```
$ git show HEAD@{5}
```

Du kannst diese Syntax auch verwenden, um zu sehen, wo sich ein Branch vor einer bestimmten Zeit befand. Um zum Beispiel zu sehen, wo dein `master` Branch gestern war, kannst du folgendes eingeben:

```
$ git show master@{yesterday}
```

Das würde dir zeigen, wo das Ende deines `master` Branchs gestern war. Diese Technik funktioniert nur für Daten, die sich noch in deinem Reflog befinden. Daher kannst du sie nicht verwenden, um nach Commits zu suchen, die älter als ein paar Monate sind.

Um die Reflog-Informationen so zu formatieren, wie die Ausgabe von `git log`, kannst du `git log -g` aufrufen:

```
$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: Fix refs handling, add gc auto, update tests
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

        Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
```

Reflog message: merge phedders/rdocs: Merge made by recursive.

Author: Scott Chacon <schacon@gmail.com>

Date: Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'

Es ist jedoch wichtig festzuhalten, dass die Reflog-Informationen ausschließlich lokale Informationen sind – es ist nur ein Protokoll dessen, was *du* in *deinem* Repository getan hast. Diese Referenzen sind nicht die gleichen wie auf einer anderen Kopie des Repositorys. Gleich nachdem du ein Repository geklont hast, hast du ein leeres Reflog, da noch keine Aktivität in deinem lokalen Repository stattgefunden hat. Wenn du `git show HEAD@{2.months.ago}` ausführst, wird dir der passende Commit nur angezeigt, wenn du das Projekt vor mindestens zwei Monaten geklont haben. Wenn du es aber erst vor kurzem geklont hast, siehst du nur deinen ersten lokalen Commit.

*Betrachte das Reflog als die Shell-Historie von Git.*



Wenn du UNIX- oder Linux-Kenntnisse hast, kannst du dir das Reflog als die Git-Version der Shell-Historie vorstellen. Diese zeigt jedoch wie bei der Shell-Historie nur die Daten für deine „Sitzung“, die mit niemand anderem etwas zu tun hat, der am gleichen Client arbeiten könnte.

*Klammern in PowerShell maskieren*

Bei Verwendung von PowerShell sind geschweifte Klammern wie `{` und `}` Sonderzeichen und müssen maskiert werden. Du kannst sie mit einem Backtick `\`` maskieren oder die Commit-Referenz in Anführungszeichen setzen:



```
$ git show HEAD@{0}      # wird nicht funktionieren
$ git show HEAD@\`{0`\}    # OK
$ git show "HEAD@{0}"     # OK
```

## Abstammung der Referenzen

Die andere Methode, um einen Commit anzugeben, ist über seine Abstammung. Wenn du ein `^` (Zirkumflex) am Ende einer Referenz anhängt, löst Git es auf, um das Eltern (engl. parent) Element dieses Commits anzusprechen. Angenommen, du schaust auf den Verlauf deines Projekts:

```
$ git log --pretty=format:'%h %s' --graph
* 734713b Fix refs handling, add gc auto, update tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd Add some blame and merge stuff
|/
* 1c36188 Ignore *.gem
* 9b29157 Add open3_detach to gemspec file list
```

Dann kannst du den vorherigen Commit sehen, indem du `HEAD^` angibst, das das „Elternteil von HEAD“ bedeutet:

```
$ 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'
```

#### *Den Zirkumflex (^) in Windows umgehen*

In der Eingabeaufforderung von Windows (`cmd.exe`) ist `^` ein Sonderzeichen und muss anders behandelt werden. Du kannst es entweder verdoppeln oder die Commit-Referenz in Anführungszeichen setzen:



```
$ git show HEAD^      # wird in Windows NICHT funktionieren  
$ git show HEAD^^    # OK  
$ git show "HEAD^"   # OK
```

Du kannst auch eine Zahl nach dem `^` angeben, um den gewünschten Elternteil zu identifizieren. So bedeutet beispielsweise `d921970^2` den „zweiten Elternteil von `d921970`“. Diese Syntax ist nur für Merge-Commits nützlich, die mehr als einen Elternteil haben – der *erste* Elternteil eines Merge-Commits stammt aus dem Branch, in dem du beim Mergen warst (in der Regel `master`). Der *zweite* Elternteil eines Merge-Commits stammt aus dem Branch, der zusammengeführt wurde (z.B. `topic`):

```
$ git show d921970^  
commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b  
Author: Scott Chacon <schacon@gmail.com>  
Date: Thu Dec 11 14:58:32 2008 -0800
```

Add some blame and merge stuff

```
$ git show d921970^2  
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548  
Author: Paul Hedderly <paul+git@mjr.org>  
Date: Wed Dec 10 22:22:03 2008 +0000
```

Some rdoc changes

Die andere wichtige Abstammungsangabe ist die `~` (Tilde). Sie bezieht sich auch auf den ersten Elternteil, so dass `HEAD~` und `HEAD^` gleichbedeutend sind. Der Unterschied wird deutlich, wenn du eine Zahl angibst. `HEAD~2` meint den „ersten Elternteil des ersten Elternteils“ oder „den Großelternteil“ – er passiert den ersten Elternteil so oft wie du angegeben hast. In der zuvor aufgelisteten Historie wäre z. B. `HEAD~3` folgendes gewesen:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

Ignore *.gem
```

Das kann auch als `HEAD~~~` geschrieben werden. Auch hier handelt es sich um den ersten Elternteil des ersten Elternteils des ersten Elternteils:

```
$ git show HEAD~~~
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date:   Fri Nov 7 13:47:59 2008 -0500

Ignore *.gem
```

Du kannst diese Syntax auch kombinieren – so kannst du den zweiten Elternteil der vorhergehenden Referenz (vorausgesetzt, es handelt sich um einen Merge Commit) erhalten, indem du `HEAD~3^2` verwendest, und so weiter.

## Commit-Bereiche

Nachdem du jetzt einzelne Commits angeben kannst, möchten wir dir zeigen, wie du einen Bereich von Commits festlegen kannst. Besonders nützlich ist das für die Verwaltung deiner Branches. Bei vielen Branches kannst du mit Hilfe von Bereichs(engl. Range)-Spezifikationen Fragen beantworten wie: „Welche Arbeit ist in diesem Branch, die ich noch nicht mit meiner Haupt-Branch zusammengeführt habe?“

### Doppelter Punkt

Die gebräuchlichste Bereichsspezifikation ist die Doppelte-Punkt-Syntax. Hiermit wird Git im Wesentlichen aufgefordert, eine Reihe von Commits aufzulösen, die von einem bestimmten Commit erreichbar sind, aber von einem anderen nicht. Angenommen, du hast eine Commit-Historie, die wie [Beispiel – Verlauf zur Bereichsauswahl](#) aussieht.

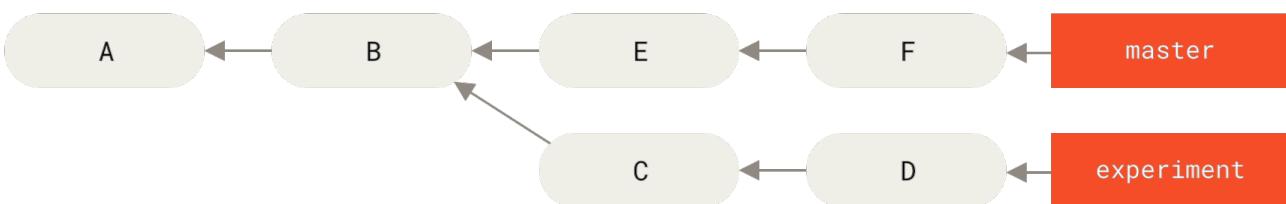


Figure 136. Beispiel – Verlauf zur Bereichsauswahl

Angenommen, du willst nun wissen, was sich in deinem Branch `experiment` befindet, das noch nicht mit deinem Branch `master` gemerged wurde. Du kannst Git fragen, ob es dir ein Log der Commits mit `master..experiment` anzeigen kann – d.h. „alle Commits, die von experiment aus erreichbar sind,

von master aus aber nicht“. Um die Kürze und Übersichtlichkeit dieser Beispiele zu erhalten, werden die Buchstaben der Commit-Objekte aus der Abbildung anstelle der eigentlichen Protokollausgabe verwendet, in der Reihenfolge, in der sie angezeigt werden:

```
$ git log master..experiment  
D  
C
```

Wenn du das Gegenteil sehen wollen – alle Commits in `master`, die nicht in `experiment` sind – dann kannst du die Branch-Namen umkehren. `experiment..master` zeigt dir alles in `master`, was von `experiment` nicht erreichbar ist:

```
$ git log experiment..master  
F  
E
```

Dieses Vorgehen ist praktisch, wenn du den Branch `experiment` auf dem aktuellen Stand halten und eine Vorschau darauf erhalten möchtest, was du gerade mergen willst. Eine weitere häufige Anwendung dieser Syntax besteht darin, zu überprüfen, was du auf einen Remote pushen möchtest:

```
$ git log origin/master..HEAD
```

Dieser Befehl zeigt dir alle Commits in deinem aktuellen Branch an, die sich nicht im Branch `master` auf deinem Remote `origin` befinden. Wenn du ein `git push` ausführst und dein aktueller Branch trackt `origin/master`, dann sind die Commits, die mit `git log origin/master..HEAD` aufgelistet werden, die Commits, die an den Server übertragen werden. Du kannst auch eine Seite der Syntax weglassen, so dass Git `HEAD` auf der fehlenden Seite annimmt. Zum Beispiel kannst du die gleichen Ergebnisse wie im vorherigen Beispiel erhalten, indem du `git log origin/master..` angibst. Git ersetzt in diesem Fall `HEAD`, wenn eine Seite fehlt.

## Mehrere Punkte

Die Doppelte-Punkt-Syntax ist als Kurzform nützlich, aber möglicherweise möchtest du mehr als zwei Branches angeben, um deinen Revisions-Stand anzuzeigen. So könntest du beispielsweise feststellen, welche Commits in einem oder mehreren Branches vorhanden sind aber sich nicht in dem Branch befinden, in dem du dich gerade aufhältst. Git ermöglicht dir dies mit dem Zeichen `^` oder dem Zusatz `--not` vor einer Referenz, von der du keinen Commits sehen möchtest. Die folgenden drei Befehle sind daher vergleichbar:

```
$ git log refA..refB  
$ git log ^refA refB  
$ git log refB --not refA
```

Das ist auch deshalb interessant, weil du mit dieser Syntax mehr als zwei Referenzen in deiner

Abfrage angeben kannst. Das ist mit der Doppelte-Punkt-Syntax (engl. Double-Dot-Syntax) nicht möglich. Wenn du zum Beispiel alle Commits sehen möchtest, die von `refA` oder `refB` aus erreichbar sind, aber nicht von `refC` aus, kannst du eine der folgenden Optionen verwenden:

```
$ git log refA refB ^refC  
$ git log refA refB --not refC
```

Das sorgt für ein sehr leistungsfähiges Revisions-Abfragesystem, das dir dabei helfen sollte, festzustellen, was in deinen Branches gerade enthalten ist.

### Dreifacher Punkt

Die letzte wichtige Syntax für die Bereichsauswahl ist die Dreifach-Punkt-Syntax (engl. Triple-Dot-Syntax), die alle Commits angibt, die durch *eine* der beiden Referenzen erreichbar sind, aber nicht durch beide. Schaue dir dazu die Commit-Historie in [Beispiel – Verlauf zur Bereichsauswahl](#) an. Wenn du wissen willst, was sich in `master` oder `experiment` befindet, aber nicht deren gemeinsamen Referenzen, kannst du diese Funktion ausführen:

```
$ git log master...experiment  
F  
E  
D  
C
```

Auch hier erhältst du eine normale `log` Ausgabe. Es werden dir jedoch nur die Commit-Informationen für diese vier Commits angezeigt, die in der normalen Reihenfolge der Commit-Daten erscheinen.

Ein gängiger Parameter, der hier mit dem `log` Befehl verwendet werden kann ist `--left-right`. Er zeigt dir, auf welcher Seite des Bereichs sich der Commit gerade befindet. Auf diese Weise wird die Ausgabe besser auswertbar:

```
$ git log --left-right master...experiment  
< F  
< E  
> D  
> C
```

Mit diesen Tools kannst du Git viel einfacher mitteilen, welche Commits du überprüfen möchtest.

## Interaktives Stagen

In diesem Abschnitt wirst du einige interaktive Git-Befehle kennenlernen, mit denen du deine Commits so gestalten kannst, dass sie nur bestimmte Kombinationen und Teile von Dateien enthalten. Diese Tools sind nützlich, um zu entscheiden, ob eine Vielzahl von umfassend modifizierten Dateien in mehrere gezielte Commits aufgeteilt oder in einem großen

unübersichtlichen Commit übertragen werden sollen. Auf diese Weise kannst du sicherstellen, dass deine Commits in logisch getrennten Changesets vorliegen, die von deinen Entwicklern leichter überprüft werden können.

Wenn du `git add` mit der Option `-i` oder `--interactive` ausführst, wechselt Git in einen interaktiven Shell-Modus, der so etwas wie das folgende anzeigt:

```
$ git add -i
      staged      unstaged path
1: unchanged      +0/-1 TODO
2: unchanged      +1/-1 index.html
3: unchanged      +5/-1 lib/simplegit.rb

*** Commands ***
1: [s]tatus      2: [u]pdate      3: [r]evert      4: [a]dd untracked
5: [p]atch      6: [d]iff       7: [q]uit       8: [h]elp
What now>
```

Du kannst erkennen, dass dieser Befehl dir eine ganz neue Darstellung deiner Staging-Area zeigt, als du es gewohnt bist. Im Grunde genommen zeigt er die gleichen Informationen, die du mit `git status` erhältst, aber etwas kompakter und informativer. Er listet auf der linken Seite die gestagten und auf der rechten Seite die nicht gestagten Änderungen auf.

Danach folgt der Bereich „Commands“ (Befehle), in dem du eine Reihe von Aktionen ausführen kannst, wie z.B. Staging und Unstaging von Dateien, Staging von Teilen von Dateien, Hinzufügen von nicht getrackten Dateien und das Anzeigen von Diffs (Unterschieden) der zuvor gestagten Dateien.

## Staging und Unstaging von Dateien

Wenn du `u` oder `2` (für Update) an der Eingabeaufforderung `What now>` eingibst, wirst du aufgefordert die Dateien anzugeben, die du zur Staging-Area hinzufügen möchtest:

```
What now> u
      staged      unstaged path
1: unchanged      +0/-1 TODO
2: unchanged      +1/-1 index.html
3: unchanged      +5/-1 lib/simplegit.rb
Update>>
```

Um die Dateien `TODO` und `index.html` zu stagern, kannst du die entsprechenden Ziffern eingeben:

```
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>>

Das \* (Sternchen) neben den Dateien bedeutet, dass die Datei zum Stagen ausgewählt wurde. Wenn du die Enter-Taste drückst, ohne dass du an der Eingabeaufforderung nach **Update>>** etwas eingegeben hast, übernimmt Git alles, was ausgewählt ist und stagt es:

```
Update>>
updated 2 paths

*** Commands ***
1: [s]tatus      2: [u]pdate      3: [r]evert      4: [a]dd untracked
5: [p]atch      6: [d]iff       7: [q]uit       8: [h]elp
What now> s
          staged      unstaged path
1:        +0/-1      nothing TODO
2:        +1/-1      nothing index.html
3:    unchanged      +5/-1 lib/simplegit.rb
```

Jetzt kannst du sehen, dass die Dateien **TODO** und **index.html** gestagt sind und die Datei **simplegit.rb** noch nicht zur Staging-Area hinzugefügt ist. Wenn du die Datei **TODO** an dieser Stelle unstagen möchtest, verwendest du die Option **r** oder **3** (für revert/rückgängig):

```
*** Commands ***
1: [s]tatus      2: [u]pdate      3: [r]evert      4: [a]dd untracked
5: [p]atch      6: [d]iff       7: [q]uit       8: [h]elp
What now> r
          staged      unstaged path
1:        +0/-1      nothing TODO
2:        +1/-1      nothing index.html
3:    unchanged      +5/-1 lib/simplegit.rb
Revert>> 1
          staged      unstaged path
* 1:        +0/-1      nothing TODO
2:        +1/-1      nothing index.html
3:    unchanged      +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Wenn du dir deinen Git-Status noch einmal anschaugst, siehst du, dass du die Datei **TODO** unstaged hast:

```
*** Commands ***
1: [s]tatus      2: [u]pdate      3: [r]evert      4: [a]dd untracked
5: [p]atch      6: [d]iff       7: [q]uit       8: [h]elp
What now> s
          staged      unstaged path
1:    unchanged      +0/-1 TODO
```

```
2:      +1/-1      nothing index.html  
3:  unchanged      +5/-1 lib/simplegit.rb
```

Um den Diff von dem zu sehen, was du gestagt hast, kannst du den Befehl **d** oder **6** (für diff) verwenden. Er zeigt dir eine Liste deiner gestagten Dateien an, aus der du auswählen kannst, für welche Dateien du die gestagte Differenz sehen möchtest. Das ist so ähnlich wie die Angabe von **git diff --cached** auf der Kommandozeile:

```
*** Commands ***  
1: [s]tatus      2: [u]pdate      3: [r]evert      4: [a]dd untracked  
5: [p]atch       6: [d]iff        7: [q]uit        8: [h]elp  
What now> d  
      staged      unstaged path  
1:      +1/-1      nothing index.html  
Review diff>> 1  
diff --git a/index.html b/index.html  
index 4d07108..4335f49 100644  
--- a/index.html  
+++ b/index.html  
@@ -16,7 +16,7 @@ Date Finder  
  
<p id="out">...</p>  
  
-<div id="footer">contact : support@github.com</div>  
+<div id="footer">contact : email.support@github.com</div>  
  
<script type="text/javascript">
```

Mit diesen grundlegenden Befehlen kannst du den interaktiven Einfügen-Modus nutzen, um deine Staging-Area etwas komfortabler zu verwalten.

## Staging von Patches

Es ist auch möglich, dass Git nur bestimmte *Teile* der Dateien stagt, ohne die restlichen Teile. Wenn du beispielsweise zwei Änderungen an deiner Datei **simplegit.rb** vornimmt und eine davon stagern möchtest und die andere nicht, so ist das in Git sehr einfach. Gib auf der gleichen interaktiven Eingabeaufforderung, die im vorherigen Abschnitt erläutert wurde, **p** oder **5** (für Patch) ein. Git wird dich fragen, welche Dateien du teilweise stagern möchtest. Anschließend zeigt es für jeden Abschnitt der ausgewählten Dateien Diffs an und fragt dich nacheinander, Stück für Stück, was du stagern möchtest:

```
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,?]?

```

Du hast an dieser Stelle viele Möglichkeiten. Die Eingabe von `?` zeigt eine Auflistung aller Aktionen, die durchführbar sind:

```

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

```

Normalerweise tippst du `y` oder `n`, wenn du die einzelnen Abschnitte stagern möchtest. Auch das Staging aller Abschnitte in bestimmten Dateien oder das Überspringen einer Abschnitts bis zu einem späteren Zeitpunkt kann sinnvoll sein. Wenn du einen Teil der Datei stagern möchtest und einen anderen Teil nicht, sieht die Ausgabe deines Status so aus:

```

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

```

Der Status der Datei `simplegit.rb` ist sehr interessant. Es zeigt dir, dass ein paar Zeilen gestagert sind und andere nicht. Du hast diese Datei teilweise zur Staging-Area hinzugefügt. An diesem Punkt kannst du das interaktive Einfüge-Skript verlassen und `git commit` ausführen, um die teilweise bereitgestellten Dateien zu committen.

Du brauchst auch nicht im interaktiven Einfüge-Modus zu sein, um mit einem Teil der Datei Staging durchzuführen. Du kannst das gleiche Skript starten, indem du `git add -p` oder `git add --patch` auf der Kommandozeile verwendest.

Darüber hinaus kannst du den Patch-Modus verwenden, um Dateien mit dem Befehl `git reset`

`--patch` teilweise zurückzusetzen, um Teile von Dateien mit dem Befehl `git checkout --patch` auszuchecken und um Teile von Dateien mit dem Befehl `git stash save --patch` zu speichern. Wir werden auf jeden dieser Befehle näher eingehen, wenn wir zu komplexeren Anwendungen dieser Befehle kommen.

## Stashen und Bereinigen

Es passiert manchmal, dass dein Projekt nach einer Bearbeitung in einem unordentlichen Zustand. Stelle dir vor du willst in einen anderen Branch wechseln, um an etwas anderem zu arbeiten. Das Problem ist, dass du keinen Commit mit halbfertiger Arbeit machen willst, nur um später an diesen Punkt zurückkehren zu können. Die Antwort auf dieses Problem ist der Befehl `git stash`.

Stashing nimmt den unsauberer Zustand deines Arbeitsverzeichnisses – das heißt, deine geänderten, getrackten Dateien und gestagte Änderungen – und speichert ihn in einem Stapel unvollendeter Änderungen, die du jederzeit (auch auf einen anderen Branch) wieder anwenden kannst.

### Migrieren zu `git stash push`

Ende Oktober 2017 gab es eine ausführliche Diskussion innerhalb der Git-Mailingliste, bei der der Befehl `git stash save` zugunsten der bestehenden Alternative `git stash push` als veraltet eingestuft wurde. Der Hauptgrund dafür ist, dass `git stash push` die Möglichkeit bietet, ausgewählte *pathspecs* zu speichern, was `git stash save` nicht unterstützt.

`git stash save` wird in naher Zukunft nicht verschwinden, also mache dir keine Sorgen, dass es plötzlich nicht mehr vorhanden ist. Wir empfehlen dir, wegen der neuen Funktionalität, zu der neuen `push` Alternative zu migrieren.



## Deine Arbeit stashen

Um das Stashen zu demonstrieren, gehe in dein Projekt und beginne mit der Arbeit an ein paar Dateien. Du kannst dann eine der Änderungen der Staging-Area hinzufügen. Wenn du `git status` ausführst, kannst du den aktuellen Status sehen:

```
$ 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
```

Du möchtest nun den Branch wechseln, aber du willst das bisherige noch nicht committen, also

solltest du die Änderungen stashen. Um einen neuen Stash zu erstellen, führe `git stash` oder `git stash push` aus:

```
$ git stash
Saved working directory and index state \
    "WIP on master: 049d078 Create index file"
HEAD is now at 049d078 Create index file
(To restore them type "git stash apply")
```

Anschliessend siehst du, dass dein Arbeitsverzeichnis bereinigt ist:

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

An dieser Stelle kannst du die Branches wechseln und anderswo arbeiten. Deine Änderungen werden solange gespeichert. Um zu sehen, welche Stashes du gespeichert hast, kannst du `git stash list` verwenden:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
```

Hier wurden vorher schon zwei Stashes gespeichert, so dass du Zugriff auf drei verschiedene gestashte Arbeiten hast. Du kannst den soeben versteckten Stash erneut aufrufen, indem du den Befehl verwendest, der in der Hilfe-Anzeige des ursprünglichen Stash-Befehls angezeigt wird: `git stash apply`. Wenn du einen der früheren Stashes anwenden möchtest, kannst du ihn durch einen Namen angeben, etwa so: `git stash apply stash@{2}`. Wenn du keinen Stash angibst, nimmt Git den neuesten Stash und versucht, ihn zu übernehmen:

```
$ 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")
```

Du kannst sehen, dass Git die Dateien, die du beim Speichern des Stashes zurückgesetzt hast, erneut modifiziert. In diesem Fall hastest du ein sauberes Arbeitsverzeichnis, als du versucht hast, den Stash anzuwenden den du auf den gleichen Branch anwenden wolltest, aus dem du ihn erzeugt

hast. Ein sauberes Arbeitsverzeichnis und dessen Anwendung auf denselben Branch sind nicht nötig, um einen Stash erfolgreich anzulegen. Du kannst einen Stash in einem Branch speichern, später in einen anderen Branch wechseln und erneut versuchen, die Änderungen zu übernehmen. Du kannst auch geänderte und nicht übertragene Dateien in deinem Arbeitsverzeichnis haben, wenn du einen Stash anwendest. Git meldet dir Merge-Konflikte, wenn etwas nicht mehr sauber funktioniert.

Die Änderungen an deinen Dateien wurden erneut angewendet, aber die Datei, die du zuvor bereitgestellt hast, wurde nicht neu eingestellt. Um das zu erreichen, musst du den Befehl `git stash apply` mit der Option `--index` ausführen und so dem Befehl anweisen, dass er versuchen soll, die gestagten Änderungen erneut anzuwenden. Hättest du stattdessen diesen Befehl ausgeführt, wärst du an deine ursprüngliche Position zurückgekehrt:

```
$ 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
```

Die `apply`-Option versucht nur, die gestashte Arbeit zu übernehmen. Du hast sie weiterhin in deinem Stack. Um sie zu entfernen, kann man `git stash drop` mit dem Namen des zu entfernenden Stash ausführen:

```
$ git stash list
stash@{0}: WIP on master: 049d078 Create index file
stash@{1}: WIP on master: c264051 Revert "Add file_size"
stash@{2}: WIP on master: 21d80a5 Add number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

Man kann auch `git stash pop` ausführen, um den Stash einzubringen und ihn dann sofort vom Stack zu entfernen.

## Kreatives Stashing

Es gibt ein paar Stash-Varianten, die ebenfalls nützlich sein können. Die erste, recht beliebte Option ist die `--keep-index` Option zum `git stash` Befehl. Diese weist Git an, nicht nur alle bereitgestellten Inhalte in den zu erstellenden Stash aufzunehmen, sondern sie gleichzeitig im Index zu belassen.

```
$ 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

```

Eine weitere gebräuchliche Funktion von `stash` ist die Ablage der nicht getrackten sowie der getrackten Dateien. Standardmäßig wird `git stash` nur modifizierte und gestagte, *getrackte* Dateien aufnehmen. Wenn du `--include-untracked` oder `-u` angeben, wird Git ungetrackte Dateien in den zu erstellenden Stash einschließen. Trotzdem wird das Einfügen von nicht getrackten Dateien in den Stash weiterhin keine explizit *zu ignorierenden* Dateien enthalten. Um zusätzlich ignorierte Dateien einzubeziehen, verwende `--all` (oder nur `-a`).

```

$ git status -s
M index.html
M lib/simplegit.rb
?? new-file.txt

$ git stash -u
Saved working directory and index state WIP on master: 1b65b17 added the index file
HEAD is now at 1b65b17 added the index file

$ git status -s
$
```

Wenn du das `--patch` Flag angibst, wird Git nicht alles, was modifiziert wurde, in den Stash aufnehmen, sondern dich fragen, welche der Änderungen du sicher verwahren willst und welche du noch in deinem Arbeitsverzeichnis behalten möchtest.

```

$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
      return `#{git_cmd} 2>&1`.chomp
    end
  end
+
+  def show(treeish = 'master')
+    command("git show #{treeish}")
+  end
end
```

```
test
Stash this hunk [y,n,q,a,d,/,e,?]? y

Saved working directory and index state WIP on master: 1b65b17 added the index file
```

## Einen Branch aus einem Stash erzeugen

Wenn Du etwas Arbeit stashen, sie eine Weile dort belassen und dann auf dem Branch weiter machen willst, aus dem du die Arbeit gestashet hast, könntest du ein Problem bekommen, die Änderungen wieder anzuwenden. Wenn man versucht, eine Datei zu ändern, die man zwischenzeitlich schon bearbeitet hatte, erhält man einen Merge-Konflikt und muss versuchen, diesen aufzulösen. Wenn du einen einfacheren Weg bevorzugst, um die gespeicherten Änderungen noch einmal zu testen, kannst du `git stash branch <new branchname>` ausführen. Damit wird ein neuer Branch mit dem gewählten Branch-Namen erzeugt, der Commit, an der du gerade gearbeitet hast, auscheckt, deine Arbeit dort wieder eingesetzt und dann den Stash verworfen, wenn er erfolgreich angewendet wurde:

```
$ 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)
```

Das ist ein interessanter Weg, mit dem man die gestasheten Arbeiten leicht wiederherstellen und in einem neuen Branch bearbeiten kann.

## Bereinigung des Arbeitsverzeichnisses

Letztendlich möchtest du vielleicht einige Arbeiten oder Dateien nicht in deinem Arbeitsverzeichnis ablegen, sondern sie einfach nur loswerden. Dafür ist der Befehl `git clean` gedacht.

Einige gängige Beispiele, in denen du dein Arbeitsverzeichnis bereinigen musst, sind das Entfernen von überflüssigem Programmcode, der durch Merges oder externe Tools erzeugt wurde oder das Entfernen von Build-Artefakten, um einen sauberen Build zu ermöglichen.

Du solltest mit diesem Befehl sehr vorsichtig sein, da er darauf ausgelegt ist, Dateien aus deinem Arbeitsverzeichnis zu entfernen, die nicht getrackt werden. Wenn du deine Absicht änderst, gibt es oft keine Möglichkeit mehr, den Inhalt dieser Dateien wiederherzustellen. Eine bessere Option ist, `git stash --all` auszuführen um alles zu entfernen, aber es in einem Stash zu speichern.

Angenommen, du willst unerwünschte Dateien entfernen oder dein Arbeitsverzeichnis bereinigen, dann kannstest du das mit `git clean` erledigen. Um alle ungetrackten Dateien in deinem Arbeitsverzeichnis zu entfernen, kannst du `git clean -f -d` ausführen, das alle Dateien entfernt, auch aus Unterverzeichnissen, die dadurch leer werden könnten. Das `-f` bedeutet „force“ (dt. „erzwingen“ oder „unbedingt ausführen“) und wird benötigt, falls die Git-Konfigurationsvariable `clean.requireForce` explizit nicht auf `false` gesetzt ist.

Wenn du wissen willst, was der Befehl bewirken könnte, dann führe ihn mit der Option `--dry-run` (oder `-n`) aus. Das bedeutet: „Mach einen Probelauf und berichte mir, was du gelöscht hättest“.

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

Standardmäßig entfernt der Befehl `git clean` nur die ungetrackten Dateien, die nicht ignoriert werden. Jede Datei, die mit einem Suchmuster in deiner `.gitignore` oder anderen Ignore-Dateien übereinstimmt, wird nicht entfernt. Wenn du diese Dateien ebenfalls entfernen willst, z.B. um alle `.o` Dateien zu entfernen, die von einem Build erzeugt wurden, kannst du dem clean-Befehl ein `-x` hinzufügen.

```
$ 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/
```

Wenn du nicht weißt, was der `git clean` Befehl bewirken wird, führe ihn immer mit einem `-n` aus, um ihn zu überprüfen, bevor du das `-n` in ein `-f` änderst und ihn dann wirklich auszuführen. Der andere Weg, wie du dich vorsehen kannst, ist den Prozess mit dem `-i` oder „interactive“ Flag auszuführen.

Dadurch wird der Clean-Befehl im interaktiven Modus ausgeführt.

```
$ 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>
```

Auf diese Weise kannst du jede Datei einzeln durchgehen oder interaktiv das zu entsprechende Lösch-Pattern festlegen.



Es gibt eine ungewöhnliche Situation, in der man Git besonders energisch auffordern muss, das Arbeitsverzeichnis zu bereinigen. Wenn du dich in einem Arbeitsverzeichnis befindest, unter dem du andere Git-Repositorys (vielleicht als Submodule) kopiert oder geklont hast, wird selbst `git clean -fd` sich weigern, diese Verzeichnisse zu löschen. In solchen Fällen musst du eine zweite `-f` Option zur Verstärkung hinzufügen.

## Deine Arbeit signieren

Git ist kryptografisch sicher, aber nicht idiotensicher. Wenn du Arbeit von anderen im Internet übernehmen und überprüfen willst, dass diese Commits tatsächlich von einer vertrauenswürdigen Quelle stammen, gibt es in Git einige Möglichkeiten, die Arbeit mit GPG zu signieren und zu überprüfen.

### Einführung in GPG

Wenn du etwas signieren willst, musst du zuerst GPG konfigurieren und deine persönlichen Schlüssel installieren.

```
$ 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
```

Wenn du noch keinen Schlüssel installiert hast, kannst du einen mit `gpg --gen-key` generieren.

```
$ gpg --gen-key
```

Sobald du einen privaten Schlüssel zum Signieren hast, kannst du Git so konfigurieren, dass er zum Signieren verwendet wird, indem du die Konfigurationseinstellung `user.signingkey` setzt.

```
$ git config --global user.signingkey 0A46826A!
```

Jetzt wird Git standardmäßig deinen Schlüssel benutzen, um Tags und Commits zu signieren, falls du es wünschst.

## Tags signieren

Wenn du einen privaten GPG-Schlüssel eingerichtet hast, kannst du diesen nun zum Signieren neuer Tags verwenden. Alles, was du tun musst, ist `-s` statt `-a` zu verwenden:

```
$ 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
```

Wenn du `git show` auf dieses Tag ausführst, kannst du deine GPG-Signatur daran angehängt sehen:

```
$ git show v1.5  
tag v1.5  
Tagger: Ben Straub <ben@straub.cc>  
Date: Sat May 3 20:29:41 2014 -0700  
  
my signed 1.5 tag  
-----BEGIN PGP SIGNATURE-----  
Version: GnuPG v1  
  
iQEcBAABAgAGBQJTzbQ1AAoJEF0+sviABDDrZbQH/09PfE51KPVPlanr6q1v4/Ut  
LQxfojUWiLQdg2ESJItkcuweYg+kC3HCyFejeDIBw9dpXt00rY26p05qrpnG+85b  
hM1/PswpPLuBSr+oCIDj5GMC2r2iEKsfv2fJbNW8iWAXVLoWZRF8B0MfqX/YTMbm  
ecorc4iXzQu7tupRihslbNkfvcimnSDeSvzCpWAH17h8Wj6hhqePmLm91AYqnKp  
8S5B/1SSQuEAjRZgI4IexpZoeKGVDptPHxLLS38fozsyi0QyDyzEgJxcJQVMXxVi  
RUysgqjcpT8+iQM1PblGfHR4XAh0qN5Fx06PSaFZhqvWFezJ28/CLyX5q+oIVk=  
=EFTF  
-----END PGP SIGNATURE-----  
  
commit ca82a6dff817ec66f44342007202690a93763949  
Author: Scott Chacon <schacon@gee-mail.com>  
Date: Mon Mar 17 21:52:11 2008 -0700  
  
Change version number
```

## Überprüfen der Tags

Um ein signiertes Tag zu prüfen, benutzt man `git tag -v <tag-name>`. Dieser Befehl verwendet GPG, um die Signatur zu verifizieren. Du benötigst den öffentlichen Schlüssel des Unterzeichners in deinem Schlüsselbund, damit das korrekt funktioniert:

```
$ 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
```

Wenn du den öffentlichen Schlüssel des Unterzeichners nicht hast, bekommst du stattdessen so etwas wie das hier zu sehen:

```
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 signieren

In neueren Versionen von Git (v1.7.9 und neuer) kannst du nun auch einzelne Commits signieren. Wenn du daran interessiert bist, Commits direkt, anstatt nur die Tags zu signieren, musst du nur ein `-S` zu deinem `git commit` Befehl hinzufügen.

```
$ 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
```

Um das Signaturen zu sehen und zu überprüfen, gibt es für `git log` auch die Option `--show-signature`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun 4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>"
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700
```

## Signed commit

Zusätzlich kannst du `git log` konfigurieren, um alle gefundenen Signaturen zu überprüfen und sie in seiner Ausgabe im `%G?` Format aufzulisten.

```
$ git log --pretty=format:%h %G? %aN  %s"  
5c3386c G Scott Chacon Signed commit  
ca82a6d N Scott Chacon Change the version number  
085bb3b N Scott Chacon Remove unnecessary test code  
a11bef0 N Scott Chacon Initial commit
```

Hier können wir feststellen, dass nur der letzte Commit signiert und gültig ist und die vorherigen Commits nicht.

In Git 1.8.3 und neuer können `git merge` und `git pull` beim Mergen angewiesen werden, einen Commit mit der Befehlsoption `--verify-signatures` zu prüfen und zurückzuweisen, wenn dieser keine vertrauenswürdige GPG-Signatur trägt.

Wenn du diese Option verwendest, während du einen Branch mergst und dieser Commits enthält, die nicht signiert und gültig sind, wird der Merge nicht ausgeführt.

```
$ git merge --verify-signatures non-verify  
fatal: Commit ab06180 does not have a GPG signature.
```

Wenn der Vorgang nur gültige signierte Commits enthält, zeigt dir der Merge-Befehl alle geprüften Signaturen an und fährt dann mit dem Merge fort.

```
$ 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(+)
```

Du kannst die Option `-S` mit dem `git merge` Befehl auch verwenden, um den resultierenden Merge-Commit selbst zu signieren. Das folgende Beispiel verifiziert bei jedem Commit in dem zusammenzuführenden Branch, dass er signiert ist und signiert darüber hinaus den resultierenden Merge-Commit.

```
$ 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(+)
```

## Jeder muss signieren

Das Signieren von Tags und Commits ist zwar ganz schön, aber wenn du dich dazu entschieden hast, das in deinem normalen Arbeitsablauf zu verwenden, musst du sicherstellen, dass jeder in deinem Team weiß, wie das geht. Wenn du das nicht tust, wirst du am Ende viel Zeit damit verbringen, den Leuten zu helfen herauszufinden, wie sie ihre Commits mit signierten Versionen neu schreiben können. Stelle sicher, dass du GPG und die Vorteile des Signierens von Dingen verstehst, bevor du dies als Teil deines Standard-Workflows übernimmst.

## Suchen

Bei fast jeder Codebasis musst du oft herausfinden, wo eine Funktion aufgerufen oder definiert wird, oder die Historie einer Methode anzeigen. Git bietet eine Reihe nützlicher Werkzeuge, um den Code und die in seiner Datenbank gespeicherten Commits schnell und einfach zu durchsuchen. Im Folgenden gehen wir ein paar davon durch.

### Git Grep

Git wird mit einem Befehl namens `grep` ausgeliefert, der es dir ermöglicht, auf einfache Weise einen beliebigen Verzeichnisbaum, das Arbeitsverzeichnis oder sogar die Staging-Area nach einer Zeichenkette (engl. string) oder einem regulären Ausdruck (engl. regular expression) zu durchsuchen. Für die folgenden Beispiele werden wir den Quellcode von Git selbst durchsuchen.

Standardmäßig durchsucht `git grep` die Dateien in deinem Arbeitsverzeichnis. Als erste Variante kannst du eine der Optionen `-n` oder `--line-number` verwenden, um die Zeilenummern anzuzeigen, bei denen Git Übereinstimmungen gefunden hat:

```
$ 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
```

Zusätzlich zur oben gezeigten einfachen Suche unterstützt `git grep` eine Vielzahl weiterer interessanter Optionen.

Anstatt beispielsweise alle Übereinstimmungen anzuzeigen, kannst du die Ausgabe von `git grep` mit der Option `-c` oder `--count` zusammenfassen. Git zeigt dir dann nur an, welche Dateien den Suchbegriff enthalten und wie viele Übereinstimmungen es in jeder Datei gibt:

```
$ 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
```

Wenn du dich für den *Kontext* eines Suchbegriffs interessierst, kannst du die umschließende Methode oder Funktion für jeden passenden Suchbegriff mit einer der Optionen `-p` oder `--show-function` anzeigen:

```
$ 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 */
```

Wie du sehen kannst, wird die Routine `gmtime_r` sowohl von den Funktionen `match_multi_number` als auch `match_digit` in der Datei `date.c` aufgerufen (die dritte angezeigte Übereinstimmung stellt nur den String dar, der in einem Kommentar erscheint).

Du kannst mit `--and` nach komplexen Kombinationen von Strings suchen, was sicherstellt, dass mehrere Übereinstimmungen in der gleichen Textzeile vorkommen müssen. Suchen wir zum Beispiel nach Zeilen, die eine Konstante definieren (den Teilstring `#define` enthalten), deren Name einen der Teilstrings `LINK` oder `BUF_MAX` enthält. Wir suchen hier in einer älteren Version der Git-Codebasis, die durch den Tag v1.8.0 repräsentiert wird (wir werden die Optionen `--break` und `--heading` hinzufügen, um die Ausgabe in ein besser lesbares Format aufzuteilen):

```
$ 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 */

```

Der Befehl `git grep` hat einige Vorteile gegenüber normalen Suchbefehlen wie `grep` und `ack`. Der erste Vorteil ist, dass es sehr schnell ist, der zweite, dass du jeden Baum in Git durchsuchen kannst, nicht nur das Arbeitsverzeichnis. Wie wir im obigen Beispiel gesehen haben, haben wir nach Begriffen in einer älteren Version des Git-Quellcodes gesucht, nicht in der Version, die gerade ausgecheckt war.

## Stichwortsuche in Git Log

Vielleicht suchst du nicht, *wo* ein Begriff existiert, sondern *wann* er existiert oder eingeführt wurde. Der Befehl `git log` verfügt über eine Reihe leistungsfähiger Werkzeuge, um bestimmte Commits anhand des Inhalts ihrer Nachrichten, oder sogar anhand des Inhalts des von ihnen eingeführten Diffs zu finden.

Wenn wir zum Beispiel herausfinden wollen, wann die Konstante `ZLIB_BUF_MAX` ursprünglich eingeführt wurde, können wir die Option `-S` (umgangssprachlich als Git „pickaxe“ Option bezeichnet) verwenden, um Git anzuweisen, uns nur die Commits anzuzeigen, in denen die Anzahl der Vorkommen dieses Strings geändert wurde.

```

$ git log -S ZLIB_BUF_MAX --oneline
e01503b zlib: allow feeding more than 4GB in one go
ef49a7a zlib: zlib can only process 4GB at a time

```

Wenn wir uns den Unterschied dieser Commits ansehen, können wir sehen, dass die Konstante in `ef49a7a` eingeführt und in `e01503b` geändert wurde.

Wenn du spezifischer sein willst, kannst du mit der Option `-G` einen regulären Ausdruck für die Suche angeben.

## Zeilen- und Funktionssuche in Git Log

Eine weitere ziemlich fortgeschrittene Logsuche, die wahnsinnig nützlich ist, ist die Suche nach dem Zeilen- und Funktionsverlauf. Führe einfach `git log` mit der Option `-L` aus, und es wird dir die Historie einer Funktion oder Codezeile in deiner Codebasis anzeigen.

Wenn wir zum Beispiel jede Änderung an der Funktion `git_deflate_bound` in der Datei `zlib.c` sehen

wollten, könnten wir `git log -L :git_deflate_bound:zlib.c` ausführen. Dies wird versuchen, die Grenzen dieser Funktion herauszufinden und dann die Historie durchzusehen und uns jede Änderung, die an der Funktion vorgenommen wurde, als eine Reihe von Patches bis zum Zeitpunkt der ersten Erstellung der Funktion zu zeigen.

```
$ 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);
+}
+
```

Wenn Git nicht herausfinden kann, wie man eine Funktion oder Methode in deiner Programmiersprache abgleicht, kannst du Git auch einen regulären Ausdruck (engl. regular expression oder regex) mitgeben. Zum Beispiel hätte Folgendes das Gleiche getan wie das obige Beispiel: `git log -L '/unsigned long git_deflate_bound/','/^}:/zlib.c`. Du kannst Git auch einen Bereich von Zeilen oder eine einzelne Zeilennummer angeben und du erhältst die gleiche Art von Ausgabe.

# Den Verlauf umschreiben

Bei der Arbeit mit Git möchtest du manchmal deinen lokalen Commit-Verlauf überarbeiten. Eine der genialen Eigenschaften von Git ist, dass es einem ermöglicht, Entscheidungen im letztmöglichen Moment zu treffen. Du kannst bestimmen, welche Dateien in welche Commits gehen, kurz bevor du in die Staging-Area committest. Du kannst mit `git stash` festlegen, dass du jetzt noch nicht an etwas arbeiten willst und du keine Commits, die bereits durchgeführt wurden, so umschreiben, dass es so aussieht, als wären sie auf eine ganz andere Art und Weise erfolgt. Das kann eine Änderung der Reihenfolge der Commits umfassen, das Ändern von Nachrichten oder das Modifizieren von Dateien in einem Commit, das Zusammenfügen oder Aufteilen von Commits, oder das komplette Entfernen von Commits. Alles bevor du deine Arbeit mit anderen teilst.

In diesem Abschnitt zeigen wir, wie du diese Aufgaben erledigen kannst, damit du deine Commit-Historie so aussehen lassen kannst, wie du es wünschst, bevor du sie mit anderen teilst.

*Du solltest deine Arbeit nicht pushen, solange du damit nicht zufrieden bist.*

Eine der wichtigsten Eigenschaften von Git ist die Möglichkeit die Verlaufshistorie, *innerhalb deines lokalen Klons*, nach deinen Wünschen umzuschreiben, weil der größte Teil der Arbeit vor Ort geschieht. Wenn du deine Arbeit jedoch einmal gepusht hast, ist das eine ganz andere Geschichte und du solltest die gepushte Arbeit als endgültig betrachten. Es sei denn, du hast gute Gründe, diese zu ändern. Um es kurz zu machen: Vermeide es, deine Arbeit so lange zu推, bis du mit ihr zufrieden bist und bereit bist, sie mit dem Rest der Welt zu teilen.



## Den letzten Commit ändern

Das Ändern des letzten Commits ist vermutlich der häufigste Grund für die Änderung der Versionshistorie. Du wirst oft zwei wesentliche Änderungen an deinem letzten Commit vornehmen wollen: die Commit-Nachricht oder den eigentlichen Inhalt des Commits ändern, indem du Dateien hinzufügst, entfernst oder modifizierst.

Wenn du lediglich die letzte Commit-Beschreibung ändern willst, ist das einfach:

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

Der obige Befehl lädt die vorherige Commit-Beschreibung in eine Editorsitzung, in der du Änderungen an der Meldung vornehmen, diese Änderungen speichern und die Sitzung beenden kannst. Wenn du die Nachricht speicherst und schließt, schreibt der Editor einen neuen Commit, der diese aktualisierte Commit-Beschreibung enthält, und macht ihn zu deiner neuen letzten Commit-Beschreibung.

Wenn du andererseits den eigentlichen *Inhalt* deines letzten Commits ändern willst, funktioniert der Prozess im Prinzip auf die gleiche Weise. Mache zuerst die Änderungen, die du glaubst, vergessen zu haben, stage diese Änderungen und der anschließende `git commit --amend` ersetzt diesen letzten Commit durch deinen neuen, verbesserten Commit.

Du musst mit dieser Technik vorsichtig sein, da die Änderung den SHA-1 des Commits ändert. Es ist

wie ein sehr kleiner Rebasing – ändere deinen letzten Commit nicht, wenn du ihn bereits gepusht hast.

*Ein geänderter Commit kann (eventuell) eine geänderte Commit-Beschreibung benötigen*

Wenn du einen Commit änderst, hast du die Möglichkeit, sowohl die Commit-Beschreibung als auch den Inhalt des Commits zu ändern. Wenn du den Inhalt des Commits maßgeblich änderst, solltest du die Commit-Beschreibung aktualisieren, um den geänderten Inhalt widerzuspiegeln.



Wenn deine Änderungen andererseits trivial ist (ein dummer Tippfehler wurde korrigiert oder eine Datei hinzugefügt, die du vergessen hast zu stagern) und die frühere Commit-Beschreibung ist in Ordnung, dann kannst du einfach die Änderungen vornehmen, sie stagern und die unnötige Editorsitzung vermeiden:

```
$ git commit --amend --no-edit
```

## Ändern mehrerer Commit-Beschreibungen

Um einen Commit zu ändern, der weiter zurückliegt, musst du komplexeren Werkzeuge nutzen. Git hat kein Tool zum Ändern der Historie, aber du kannst das Rebasing-Werkzeug verwenden, um eine Reihe von Commits auf den HEAD zu übertragen, auf dem sie ursprünglich basieren, anstatt sie auf einen anderen zu verschieben. Mit dem interaktiven Rebasing-Werkzeug kannst du dann nach jedem Commit pausieren und die Beschreibung ändern, Dateien hinzufügen oder was immer du willst. Du kannst Rebasing interaktiv ausführen, indem du die Option `-i` mit `git rebase` verwendest. Du musst angeben, wie weit du die Commits umschreiben willst, indem du dem Kommando den Commit nennst, auf den du rebasen willst.

Wenn du zum Beispiel die letzten drei Commit-Beschreibungen oder eine der Commit-Beschreibungen in dieser Gruppe ändern willst, gebe als Argument für `git rebase -i` das Elternteil der letzten Commit-Beschreibung, die du bearbeiten willst, an (`HEAD~2^` oder `HEAD~3`). Es ist vielleicht einfacher, sich die `~3` zu merken, weil du versuchst, die letzten drei Commits zu bearbeiten. Bedenke aber, dass du eigentlich vier Commits angeben musst, den Elternteil des letzten Commits, den du bearbeiten willst:

```
$ git rebase -i HEAD~3
```

Bitte vergiss nicht, dass es sich hierbei um einen Rebasing-Befehl handelt. Jeder Commit im Bereich `HEAD~3..HEAD` mit einer geänderten Beschreibung und *allen seinen Nachfolgern* wird neu geschrieben. Füge keinen Commit ein, den du bereits auf einen zentralen Server gepusht hast. Das wird andere Entwickler verwirren, weil sie eine neue Version der gleichen Änderung übermitteln.

Wenn du diesen Befehl ausführst, erhältst du eine Liste von Commits in deinem Texteditor, die ungefähr so aussieht:

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
```

```

pick a5f4a0d Add cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [<oneline>]
# .      create a merge commit using the original merge commit's
# .      message (or the oneline, if no original merge commit was
# .      specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out

```

Es ist wichtig zu erwähnen, dass diese Commits in der umgekehrten Reihenfolge aufgelistet werden, als du sie normalerweise mit dem `log` Befehl siehst. Wenn du ein `log` ausführst, siehst du etwas wie das hier:

```

$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d Add cat-file
310154e Update README formatting and add blame
f7f3f6d Change my name a bit

```

Beachte die entgegengesetzte Reihenfolge. Der interaktive Rebase stellt dir ein Skript zur Verfügung, den er ausführen wird. Es beginnt mit dem Commit, den du auf der Kommandozeile angibst (`HEAD~3`) und gibt die Änderungen, die in jedem dieser Commits eingeführt wurden, von oben nach unten wieder. Es listet die ältesten oben auf, nicht die neuesten, weil es die ersten sind, die es wiedergibt.

Du musst das Skript so bearbeiten, dass es bei dem Commit anhält, den du bearbeiten willst. Ändere dazu das Wort `pick` in das Wort `edit` für jeden Commit, nach dem das Skript anhalten soll. Um beispielsweise nur die dritte Commit-Beschreibung zu ändern, ändere die Datei so, dass sie wie folgt aussieht:

```
edit f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

Wenn du speicherst und den Editor verlässt, springt Git zum letzten Commit in dieser Liste zurück und zeigt dir die folgende Meldung an der Kommandozeile:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... Change my name a bit
You can amend the commit now, with

    git commit --amend

Once you're satisfied with your changes, run

    git rebase --continue
```

Diese Hinweise sagen dir genau, was zu tun ist. Schreibe:

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

ändere die Commit-Beschreibung und verlasse den Editor. Dann rufe folgenden Befehl auf:

```
$ git rebase --continue
```

Damit setzt du die anderen beiden Commits automatisch fort und du bist fertig. Falls du „pick“ zum Bearbeiten in mehreren Zeilen zu „edit“ änderst, kannst du diese Schritte für jeden zu bearbeitenden Commit wiederholen. Jedes Mal hält Git an, lässt dich den Commit ändern und fährt fort, sobald du fertig bist.

## Commits umsortieren

Du kannst interaktive Rebases auch verwenden, um Commits neu anzurichten oder ganz zu entfernen. Wenn du unten den „Add cat-file“ Commit entfernst und die Reihenfolge ändern willst, in der die anderen beiden Commits aufgeführt werden, kannst du das Rebase-Skript so anpassen (vorher):

```
pick f7f3f6d Change my name a bit
pick 310154e Update README formatting and add blame
pick a5f4a0d Add cat-file
```

nachher:

```
pick 310154e Update README formatting and add blame
```

```
pick f7f3f6d Change my name a bit
```

Wenn du gespeichert und den Editor verlassen hast, spult Git deinen Branch zum Elternteil dieser Commits zurück, wendet **310154e** und dann **f7f3f6d** an und stoppt dann. Du änderst effektiv die Reihenfolge dieser Commits und entfernst den „Add cat-file“ Commit komplett.

## Commits zusammenfassen

Es ist auch möglich, eine Reihe von Commits zu mit dem interaktiven Rebasing-Werkzeug zu einem einzigen Commit zusammenzufassen (engl. to squash). Das Skript fügt hilfreiche Anweisungen in die Rebasemeldung ein:

```
#  
# Commands:  
# p, pick <commit> = use commit  
# r, reword <commit> = use commit, but edit the commit message  
# e, edit <commit> = use commit, but stop for amending  
# s, squash <commit> = use commit, but meld into previous commit  
# f, fixup <commit> = like "squash", but discard this commit's log message  
# x, exec <command> = run command (the rest of the line) using shell  
# b, break = stop here (continue rebase later with 'git rebase --continue')  
# d, drop <commit> = remove commit  
# l, label <label> = label current HEAD with a name  
# t, reset <label> = reset HEAD to a label  
# m, merge [-C <commit> | -c <commit>] <label> [<oneline>]  
# .      create a merge commit using the original merge commit's  
# .      message (or the oneline, if no original merge commit was  
# .      specified). Use -c <commit> to reword the commit message.  
#  
# These lines can be re-ordered; they are executed from top to bottom.  
#  
# If you remove a line here THAT COMMIT WILL BE LOST.  
#  
# However, if you remove everything, the rebase will be aborted.  
#  
# Note that empty commits are commented out
```

Wenn du statt „pick“ oder „edit“ „squash“ angibst, wendet Git sowohl diese Änderung als auch die Änderung direkt davor an und lässt dich die Commit-Beschreibungen zusammenfügen. Wenn du also einen einzelnen Commit aus diesen drei Commits machen willst, musst du das Skript wie folgt anpassen:

```
pick f7f3f6d Change my name a bit  
squash 310154e Update README formatting and add blame  
squash a5f4a0d Add cat-file
```

Wenn du speicherst und den Editor schließt, wendet Git alle drei Änderungen an und öffnet dann

wieder den Editor, um die drei Commit-Beschreibungen zusammenzuführen:

```
# This is a combination of 3 commits.  
# The first commit's message is:  
Change my name a bit  
  
# This is the 2nd commit message:  
  
Update README formatting and add blame  
  
# This is the 3rd commit message:  
  
Add cat-file
```

Wenn du das speicherst, hast du einen einzigen Commit, der die Änderungen aller drei vorherigen Commits einbringt.

## Aufspalten eines Commits

Das Aufteilen eines Commits macht einen Commit rückgängig und stagt dann partiell so viele Commits, wie du am Ende haben willst. Nehmen wir beispielsweise an, du willst den mittleren Commit deiner drei Commits teilen. Statt „Update README formatting and add blame“ willst du ihn in zwei Commits aufteilen: „Update README formatting“ für die erste und „Add blame“ für die zweite. Du kannst das mit dem `rebase -i` Skript tun, indem du die Anweisung für den Commit, den du aufteilen willst, in „edit“ änderst:

```
pick f7f3f6d Change my name a bit  
edit 310154e Update README formatting and add blame  
pick a5f4a0d Add cat-file
```

Wenn das Skript dich dann auf die Befehlszeile zurückführt, setze diesen Commit zurück, übernehmen die zurückgesetzten Änderungen und erstelle daraus mehrere Commits. Wenn du speichern und den Editor verlässt, springt Git zum Elternteil des ersten Commits in deiner Liste zurück, wendet den ersten Commit an (`f7f3f6d`), wendet den zweiten an (`310154e`) und führt dich zurück auf die Konsole. Dort kannst du ein kombiniertes Zurücksetzen dieses Commits mit `git reset HEAD^` durchführen, was praktisch den Commit rückgängig macht und die modifizierten Dateien unstaged lässt. Jetzt kannst du Dateien so lange stagern und committen, bis du mehrere Commits ausgeführt hast, und danach, wenn du fertig bist, `git rebase --continue` starten:

```
$ git reset HEAD^  
$ git add README  
$ git commit -m 'Update README formatting'  
$ git add lib/simplegit.rb  
$ git commit -m 'Add blame'  
$ git rebase --continue
```

Git wendet den letzten Commit (**a5f4a0d**) im Skript an, und dein Verlauf sieht dann so aus:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd Add cat-file
9b29157 Add blame
35cfb2b Update README formatting
f7f3f6d Change my name a bit
```

Dies ändert die SHA-1s der drei jüngsten Commits in deiner Liste. Stelle also sicher, dass kein geänderter Commit in dieser Liste auftaucht, den du bereits in ein gemeinschaftliches Repository verschoben hast. Beachte, dass der letzte Commit (**f7f3f6d**) in der Liste nicht geändert wurde. Trotzdem wird dieser Commit im Skript angezeigt, da er als „pick“ markiert war und vor jeglichen Rebase-Änderungen angewendet wurde. Git lässt den Commit unverändert.

## Commit löschen

Wenn du einen Commit entfernen möchtest, kannst du ihn mit dem Skript **rebase -i** löschen. Füge in der Liste der Commits das Wort „drop“ vor dem Commit ein, den du löschen möchtest (oder lösche einfach diese Zeile aus dem Rebase-Skript):

```
pick 461cb2a This commit is OK
drop 5aec10 This commit is broken
```

Aufgrund der Art und Weise, wie Git Commit-Objekte erstellt, werden beim Löschen oder Ändern eines Commits alle darauf folgenden Commits neu geschrieben. Je weiter du in der Historie deines Repos zurück gehst, desto mehr Commits müssen neu erstellt werden. Dies kann zu vielen Mergekonflikten führen, wenn es viele Commits in der Historie gibt, die von dem gerade gelöschten abhängen.

Wenn du ein solches Rebase teilweise durchläufst und feststellst, dass dies keine gute Idee ist, kannst du jederzeit damit aufhören. Gib **git rebase --abort** ein und dein Repo wird in den Zustand zurückversetzt, in dem es sich befand, bevor du das Rebase gestartet hast.

Wenn du ein Rebase beendest und feststellst, dass es nicht das ist, was du wolltest, kannst du **git reflog** verwenden, um eine frühere Version deines Branches wiederherzustellen. Weitere Informationen zum Befehl **reflog** findest du unter [Datenwiederherstellung](#).



Drew DeVault hat einen praktischen Leitfaden mit Übungen erstellt, um die Verwendung von **git rebase** zu erlernen. Er ist unter <https://git-rebase.io/> zu finden.

## Die Nuklear-Option: filter-branch

Es gibt noch eine weitere Option zum Überschreiben der Historie, wenn du eine größere Anzahl von Commits auf eine skriptfähige Art und Weise umschreiben musst. Wenn du, zum Beispiel, deine E-Mail-Adresse global änderst oder eine Datei aus jedem Commit entfernen willst. Der Befehl heißt **filter-branch** und kann große Teile deines Verlaufs neu schreiben. Du solltest ihn deshalb

besser nicht verwenden. Es sei denn, dein Projekt ist noch nicht veröffentlicht und andere Leute haben noch keine Arbeiten an den Commits durchgeführt, die du gerade neu schreiben willst. Wie auch immer, er kann sehr nützlich sein. Du wirst ein paar der häufigsten Verwendungszwecke kennenlernen, damit du eine Vorstellung gewinnen kannst, wofür er geeignet ist.



`git filter-branch` hat viele Fallstricke und wird nicht mehr empfohlen, um die Historie umzuschreiben. Stattdessen solltest du die Verwendung von `git-filter-repo` in Betracht ziehen. Das ist ein Python-Skript, das für die meisten Aufgaben besser geeignet ist, bei denen du normalerweise auf `filter-branch` zurückgreifen würdest. Die zugehörige Dokumentation und den Quellcode findest du unter <https://github.com/newren/git-filter-repo>.

## Eine Datei aus jedem Commit entfernen

Das kommt relativ häufig vor. Jemand übergibt versehentlich eine riesige Binärdatei mit einem gedankenlosen `git add .` und du willst sie überall entfernen. Vielleicht hast du versehentlich eine Datei übergeben, die ein Passwort enthält und du willst dein Projekt zu Open Source machen. `filter-branch` ist das Mittel der Wahl, um deinen gesamten Verlauf zu säubern. Um eine Datei namens `passwords.txt` aus deinem gesamten Verlauf zu entfernen, kannst du die Option `--tree-filter` mit `filter-branch` verwenden:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

Die Option `--tree-filter` führt den angegebenen Befehl nach jedem Checkout des Projekts aus und überträgt die Ergebnisse erneut. In diesem Fall entfernst du die Datei `passwords.txt` aus jedem Schnapschuss, unabhängig davon, ob sie existiert oder nicht. Wenn du alle versehentlich übertragene Editor-Backup-Dateien entfernen möchtest, kannst du beispielsweise `git filter-branch --tree-filter 'rm -f *~' HEAD` ausführen.

Du wirst in der Lage sein, Git beim Umschreiben der Bäume und Commits zu beobachten und am Ende den Branch-Zeiger zu bewegen. Generell ist es ratsam, das in einem Test-Branch zu tun und den `master` Branch hart zurückzusetzen, wenn das Ergebnis so ist, wie du es erwartet hast. Um `filter-branch` auf allen deinen Branches auszuführen, kannst du die Option `--all` an den Befehl übergeben.

## Ein Unterverzeichnis zum neuen Root machen

Nehmen wir an, du hast einen Import aus einem anderen Versionsverwaltungssystem durchgeführt und verfügst über Unterverzeichnisse, die keinen Sinn machen (`trunk`, `tags` usw.). Wenn du das `trunk` Unterverzeichnis zum neuen Stamm-Verzeichnis des Projekts für jeden Commit machen willst, kann dir `filter-branch` auch dabei helfen:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Jetzt ist dein neues Projekt-Stammverzeichnis das, was sich vorher im Unterverzeichnis `trunk` befand. Git wird automatisch Commits entfernen, die sich nicht auf das Unterverzeichnis auswirken.

## Globales Ändern von E-Mail-Adressen

Ein weiterer häufiger Fall ist, dass du vergessen hast, `git config` auszuführen, um deinen Namen und deine E-Mail-Adresse vor Beginn der Arbeit festzulegen oder vielleicht willst du ein Open-Source-Projekt eröffnen und alle deine Arbeits-E-Mail-Adressen auf deine persönliche Adresse ändern. In jedem Fall kannst du die E-Mail-Adressen in mehreren Commits in einem Batch mit `filter-branch` ebenfalls ändern. Du musst darauf achten, nur die E-Mail-Adressen zu ändern, die dir gehören. Deshalb solltest du `--commit-filter` verwenden:

```
$ 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
```

Dadurch wird jeder Commit umgeschrieben, um deine neue Adresse zu erhalten. Da die Commits die SHA-1-Werte ihrer Eltern enthalten, ändert dieser Befehl jeden Commit SHA-1 in deinem Verlauf, nicht nur diejenigen, die die passende E-Mail-Adresse haben.

## Reset entzaubert

Bevor wir zu spezialisierteren Werkzeugen übergehen, sollten wir über die Befehle `reset` und `checkout` sprechen. Diese Befehle sind, wenn man ihnen zum ersten Mal begegnet, die beiden verwirrendsten Teile von Git. Sie erledigen so viele Aufgaben, dass es aussichtslos erscheint, sie wirklich zu verstehen und richtig anzuwenden. Deshalb empfehlen wir eine einfache Metapher.

## Die drei Bäume

Eine bessere Methode, um über `reset` und `checkout` zu reflektieren, ist der gedankliche Ansatz, dass Git ein Inhaltsmanager von drei verschiedenen Bäumen ist. Mit „Baum“ meinen wir hier in Wahrheit eine „Sammlung von Dateien“, nicht speziell die Datenstruktur. Es gibt ein paar Fälle, in denen sich der Inhalt nicht genau wie ein Baum verhält, aber für unsere Zwecke ist es vorerst einfacher, auf diese Weise darüber nachzudenken.

Als System verwaltet Git im regulären Modus drei Bäume:

Baum	Rolle
HEAD	letzter Commit-Snapshot, nächstes Elternteil
Index (Staging-Area)	nächster, geplanter Commit-Snapshot

Baum	Rolle
Arbeitsverzeichnis	Sandbox

## Der HEAD

HEAD ist der Verweis auf die aktuelle Branch-Referenz, die wiederum ein Pointer zu dem letzten Commit auf dem aktuellen Branch ist. Das bedeutet, dass HEAD das Elternteil des nächsten Commits ist, der erzeugt wird. Es ist generell am einfachsten, sich HEAD als den Schnappschuss **deines letzten Commits auf dem aktuellen Branch** vorzustellen.

Es ist ziemlich einfach zu erkennen, wie dieser Schnappschuss aussieht. Hier ist ein Beispiel, wie man die aktuelle Verzeichnisliste und die SHA-1-Prüfsummen für jede Datei im HEAD-Snapshot erhält:

```
$ 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
```

Die Git-Befehle `cat-file` und `ls-tree` sind „Basisbefehle“, die für Aufgaben auf low-level Ebene verwendet werden und nicht wirklich in der täglichen Arbeit eingesetzt werden. Es hilft jedoch sie zu verstehen, was sie genau tun.

## Der Index

*Index* ist dein **nächster, geplanter Commit**. Wir haben diesen Konzept auch als Git's „Staging-Area“ bezeichnet, da Git darauf schaut, wenn du `git commit` ausführst.

Git füllt den Index mit allen Dateiinhalten, die du zuletzt in dein Arbeitsverzeichnis ausgecheckt hast und zeigt dir, wie sie beim letzten Auschecken ausgesehen haben. Du tauschst dann einige dieser Dateien mit neueren Versionen aus, und `git commit` konvertiert diese in den Baum für einen neuen Commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README
100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb
```

Nochmals, wir verwenden hier `git ls-files`, ein Kommando, das eher ein Basisbefehl ist, welcher dir anzeigt, wie dein Index derzeit aussieht.

Der Index ist technisch gesehen keine hierarchische Struktur. Er ist eigentlich als flaches Manifest umgesetzt, aber für unsere Zwecke ist das ausreichend genau.

## Das Working Directory oder Arbeitsverzeichnis

Abschließend gibt es dein *Arbeitsverzeichnis* (engl. „working directory“ oder „working tree“). Die beiden anderen Bäume speichern deinen Inhalt auf effiziente, aber unpraktische Weise innerhalb des `.git` Ordners. Das Arbeitsverzeichnis entpackt sie in echte Dateien, was es wesentlich einfacher macht, sie zu bearbeiten. Stelle dir das Arbeitsverzeichnis wie einen **Sandkasten** (engl. sandbox) vor, in der du Änderungen ausprobieren kannst, bevor du sie in deiner Staging-Area und dann in den Verlauf überträgst.

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

1 directory, 3 files
```

## Der Workflow

Der typische Arbeitsablauf von Git sieht vor, dass du durch die Bearbeitung dieser drei Bäume nach und nach bessere Momentaufnahmen deines Projekts erzeugst.

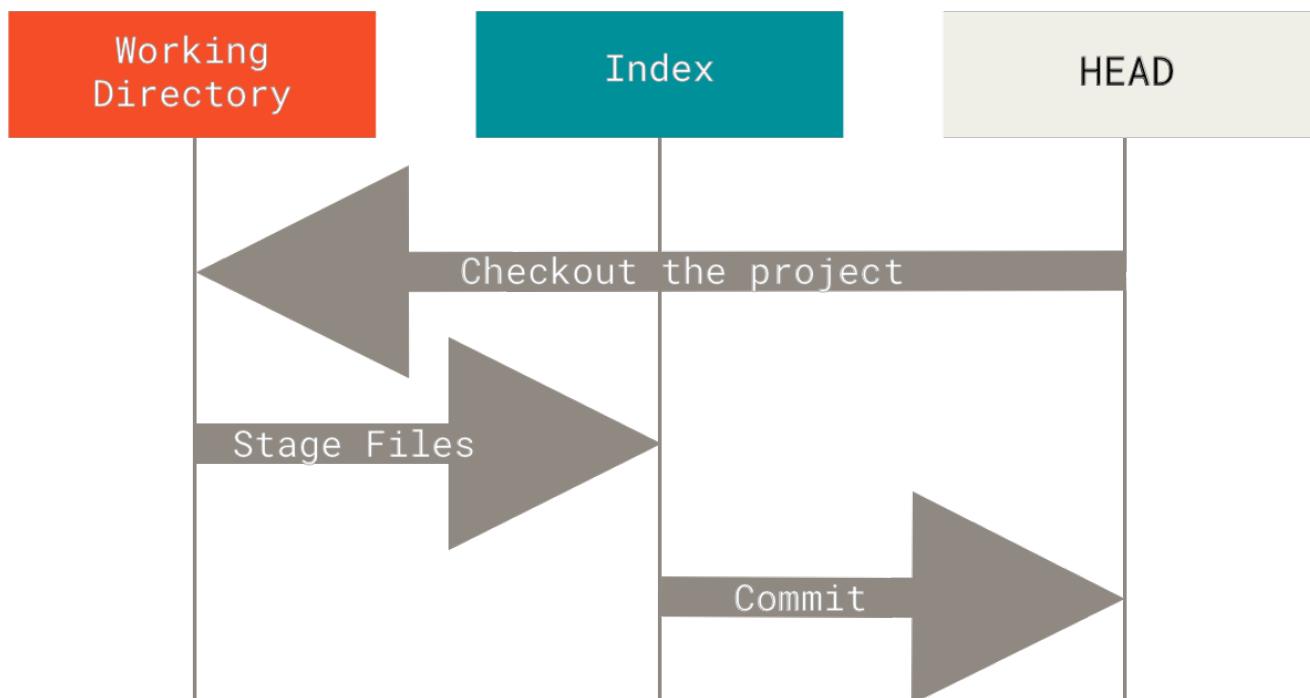


Figure 137. Git's standard workflow

Stellen wir uns folgenden Ablauf vor: Angenommen, du wechselst in ein neues Verzeichnis, in dem sich eine einzige Datei befindet. Wir nennen das die **v1** der Datei und kennzeichnen sie in blau.

Nun führen wir `git init` aus, das ein Git-Repository mit einer HEAD-Referenz erzeugt, die auf den noch nicht existierenden `master` Branch zeigt.

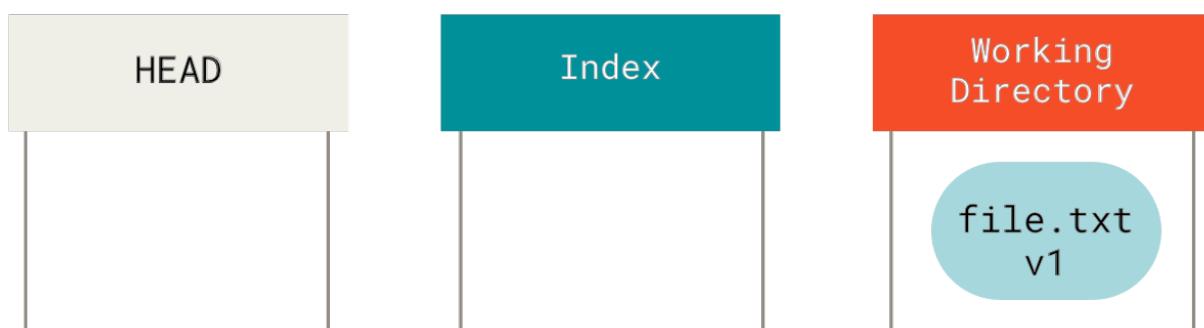
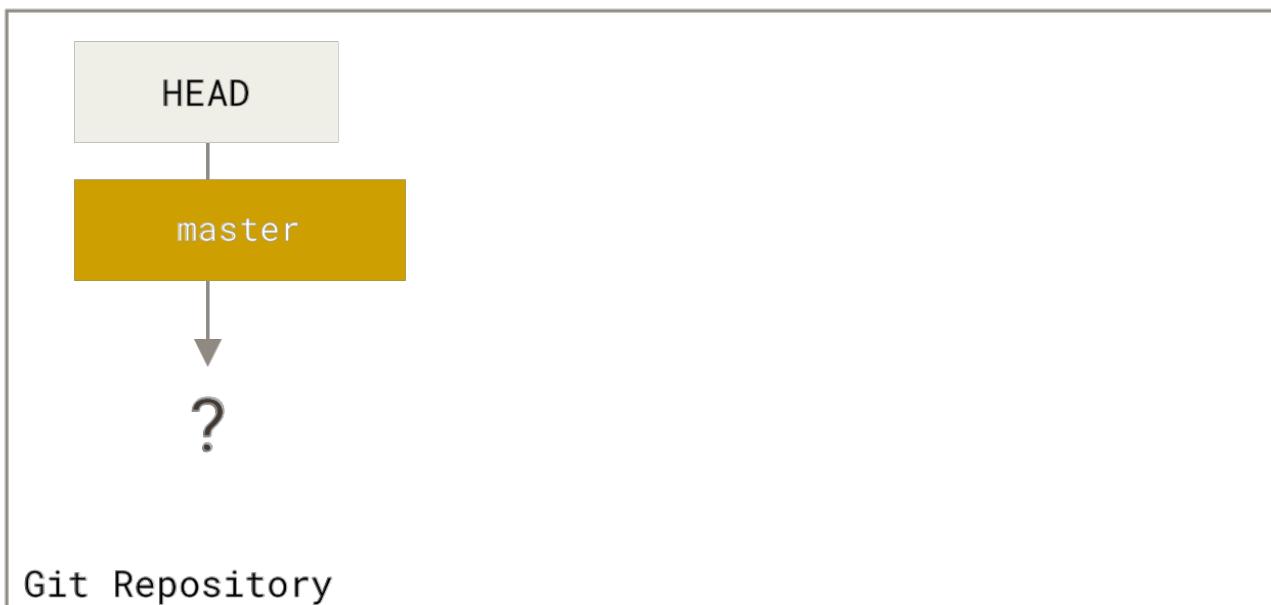


Figure 138. Neu-Instaziertes Git Repository mit unstaged Datei im Arbeitsverzeichnis

Zu diesem Zeitpunkt hat nur der Verzeichnisbaum (engl working tree) des Arbeitsverzeichnisses (engl. working directory) irgendeinen Inhalt.

Nun wollen wir diese Datei committen, also benutzen wir `git add`, um den Inhalt im Arbeitsverzeichnis zu übernehmen und in den Index zu kopieren.

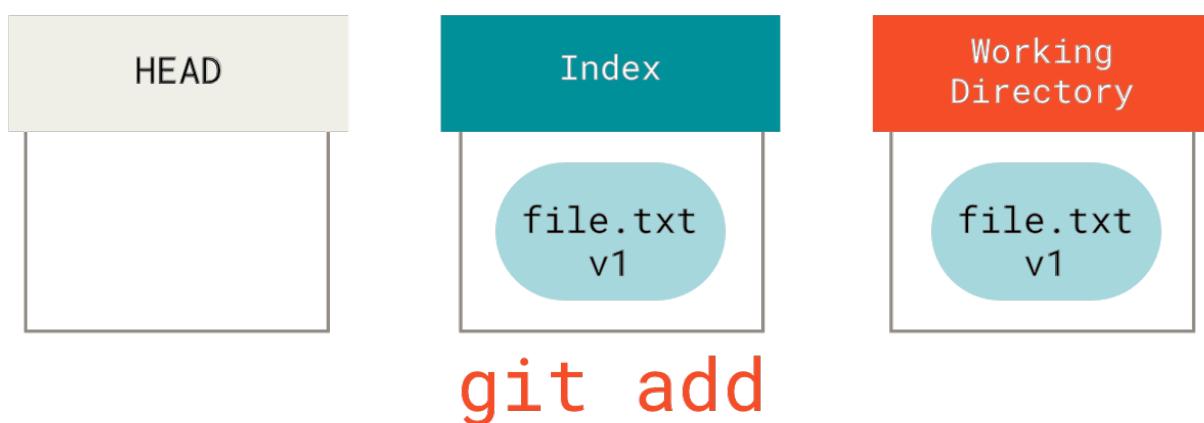
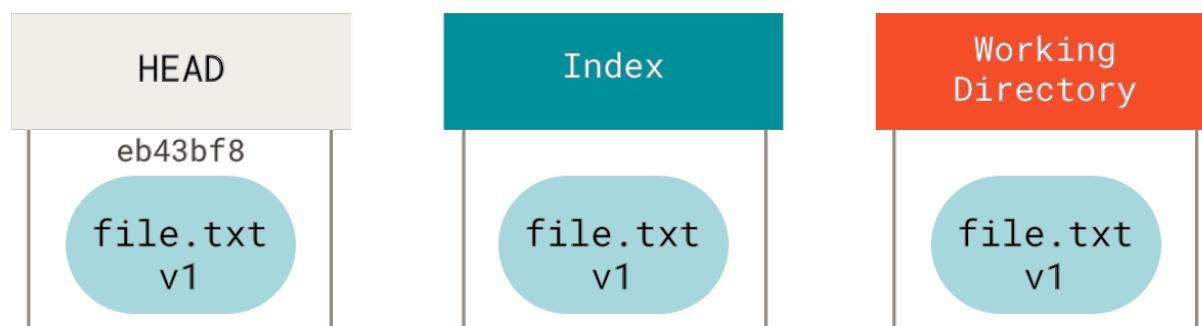
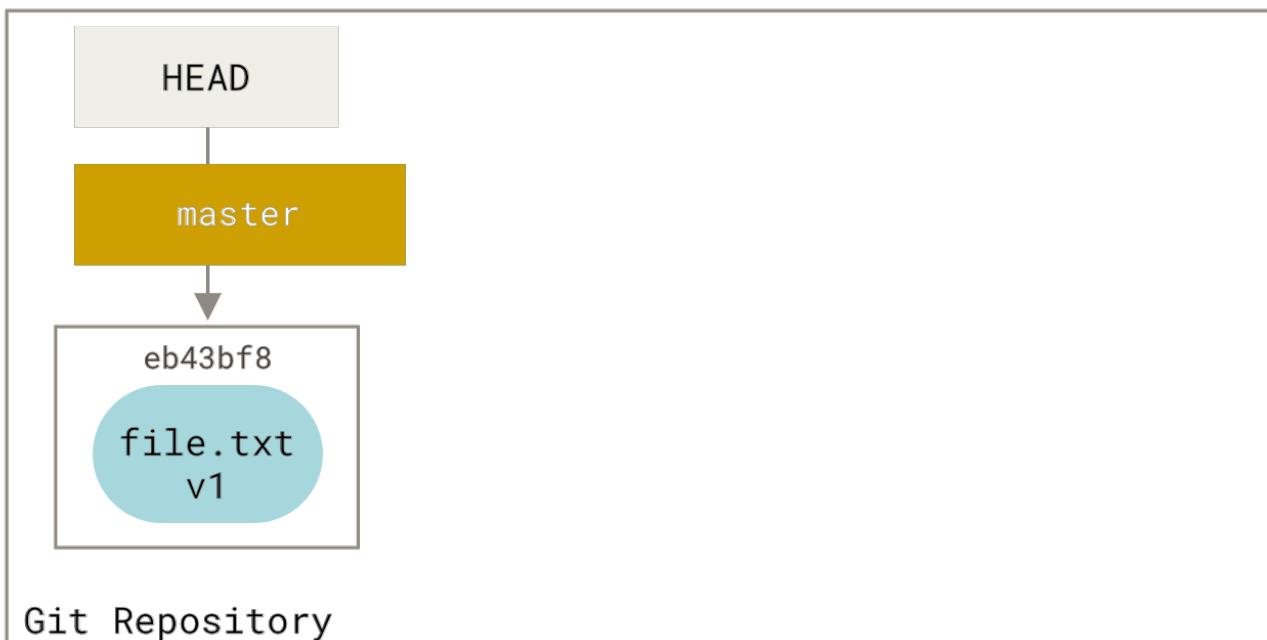


Figure 139. Datei wird bei `git add` auf den Index kopiert

Dann führen wir `git commit` aus, das den Inhalt der Staging-Area (oder Index) als endgültigen Snapshot speichert, ein Commit-Objekt erzeugt, das auf diesen Snapshot zeigt, und den Branch `master` aktualisiert, um auf diesen Commit zu zeigen.



## git commit

Figure 140. Der `git commit` Schritt

Wenn wir jetzt `git status` ausführen, werden wir keine Änderungen sehen, weil alle drei Bäume gleich sind.

Nun wollen wir eine Änderung an dieser Datei vornehmen und sie übertragen. Wir führen den gleichen Vorgang durch. Zuerst ändern wir die Datei in unserem Arbeitsverzeichnis. Wir nennen sie **v2** dieser Datei und markieren sie in rot.

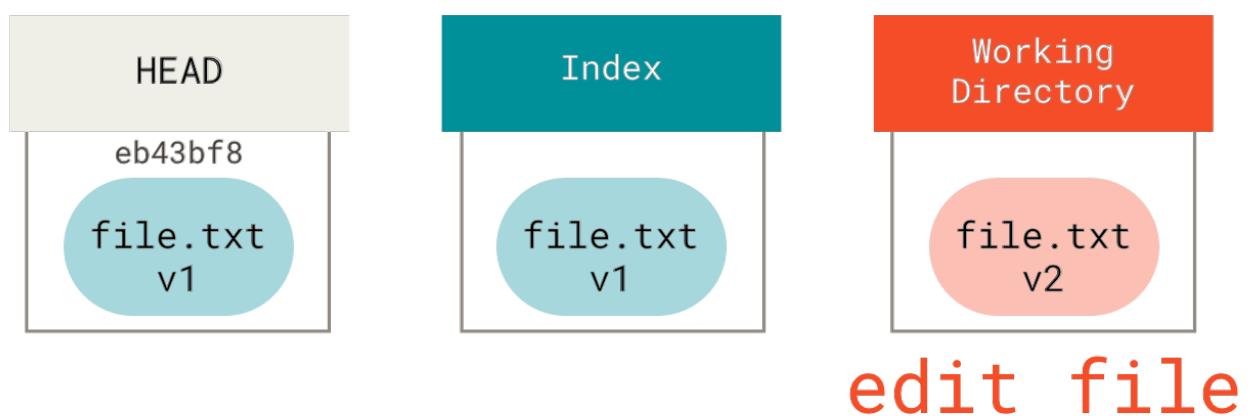
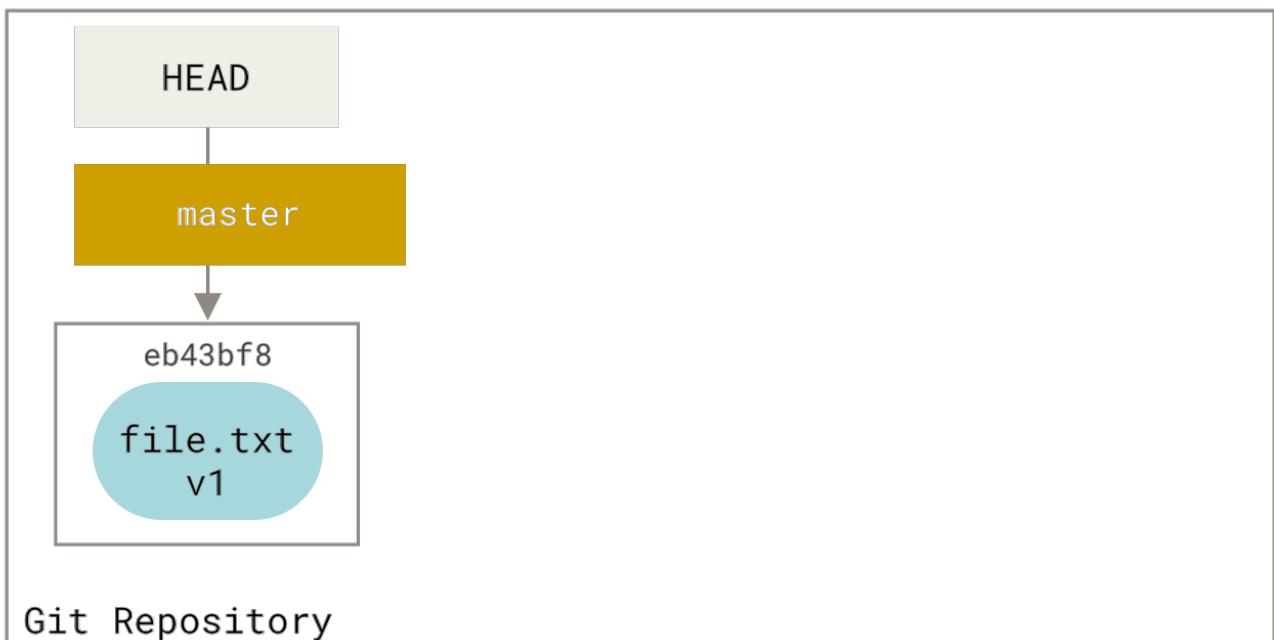


Figure 141. Git Repository mit geänderten Dateien im Arbeitsverzeichnis

Wenn wir jetzt den Befehl `git status` aufrufen, sehen wir die Datei in rot als „Changes not staged for commit“ (dt. Änderungen nicht zum Commit vorgemerkt), weil sich dieser Eintrag im Index zu dem im Arbeitsverzeichnis unterscheidet. Als nächstes führen wir `git add` aus, um sie in unseren Index zu übernehmen, d.h zur Staging-Area hinzuzufügen.

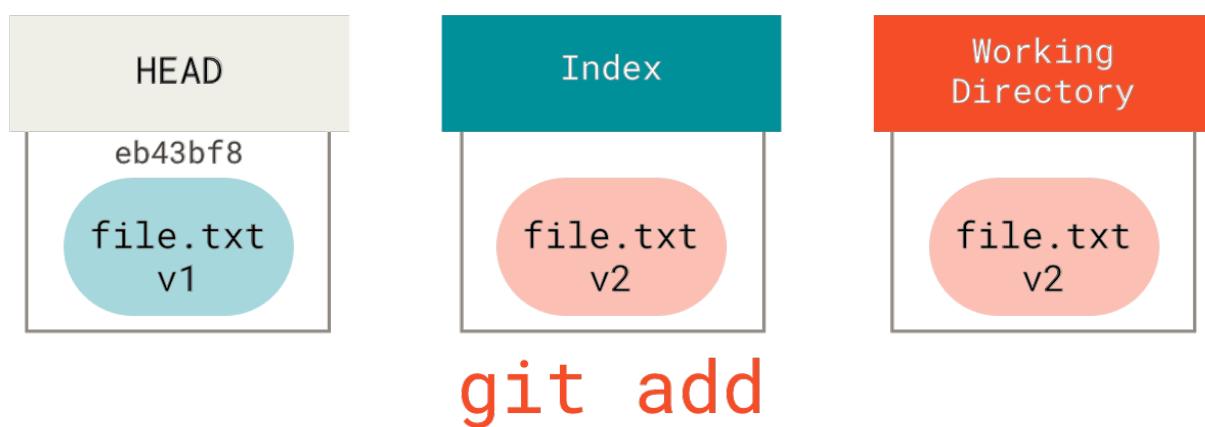
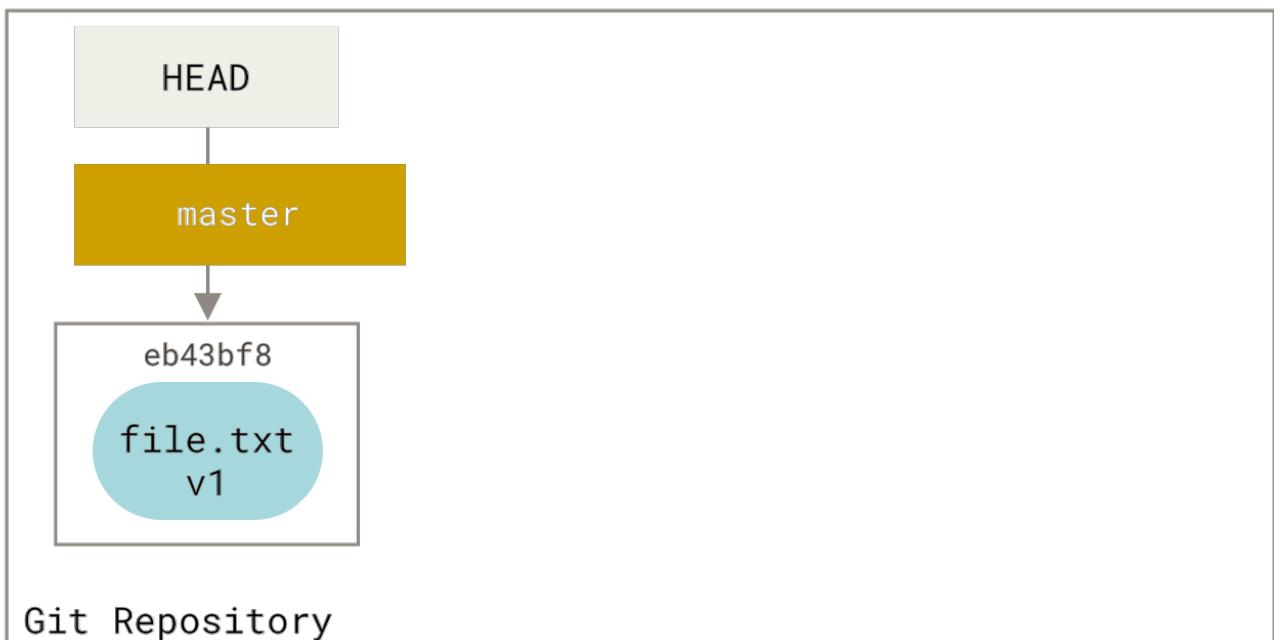
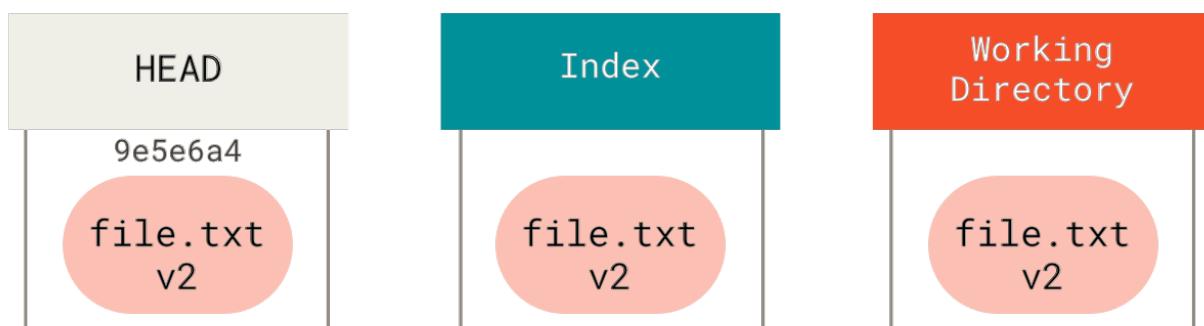
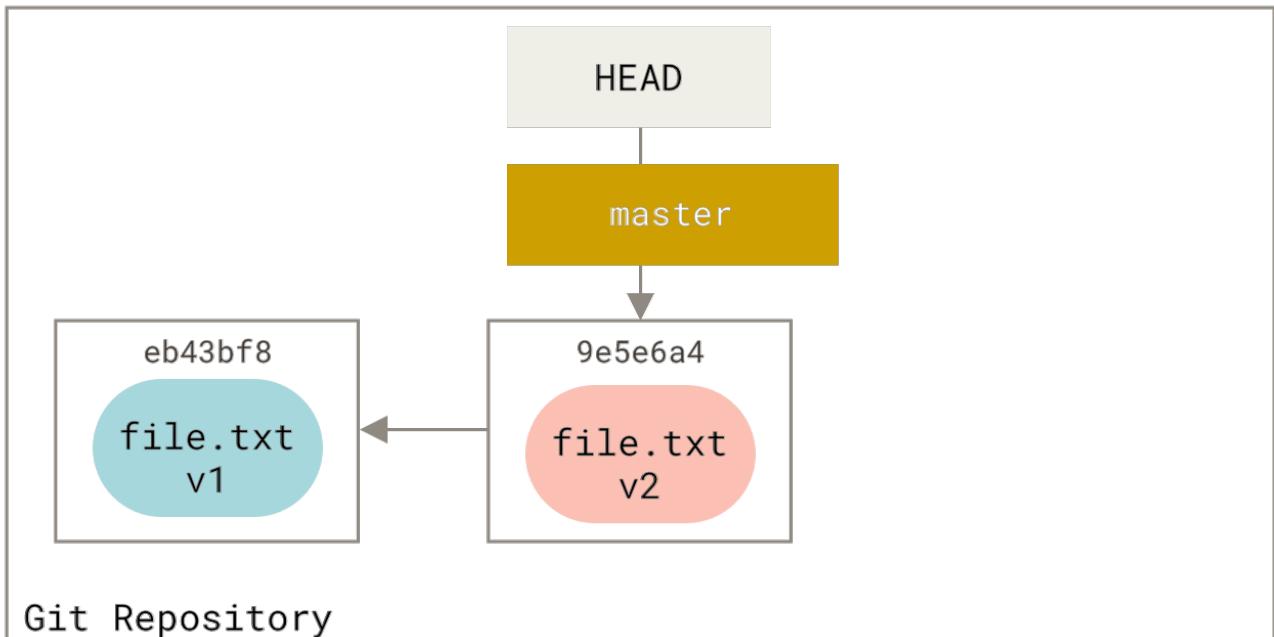


Figure 142. Staging Änderungen am Index

Wenn wir zu diesem Zeitpunkt `git status` ausführen, sehen wir die Datei in grün unter „Changes to be committed“ (dt. Änderungen zum Commit vorgemerkt), weil sich der Index und der HEAD unterscheiden – d.h. unser geplanter nächster Commit unterscheidet sich nun von unserem letzten Commit. Schließlich führen wir `git commit` aus, um die Daten zu übertragen.



## git commit

Figure 143. Der `git commit` Schritt mit geänderter Datei

Nun wird uns `git status` keine Ergebnisse liefern, weil alle drei Bäume wieder gleich sind.

Das Wechseln von Branches oder das Klonen geht ähnlich vor sich. Wenn du einen Branch auscheckst, ändert er **HEAD** so, dass er auf den neuen Branch-Ref zeigt, füllt deine **Staging-Area** (bzw. Index) mit dem aktuellen Schnapschuss dieses Commits und kopiert dann den Inhalt des **Index** in dein **Arbeitsverzeichnis**.

## Die Bedeutung von Reset

Der Befehl `reset` macht mehr Sinn, wenn wir folgenden Fall betrachten.

Für diesen Zweck nehmen wir an, dass wir `file.txt` erneut modifiziert und ein drittes Mal committet hätten. Nun sieht unser Verlauf so aus:

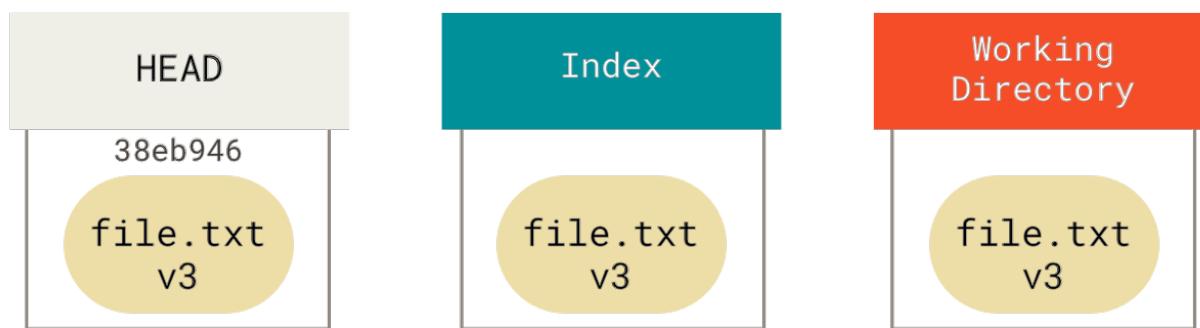
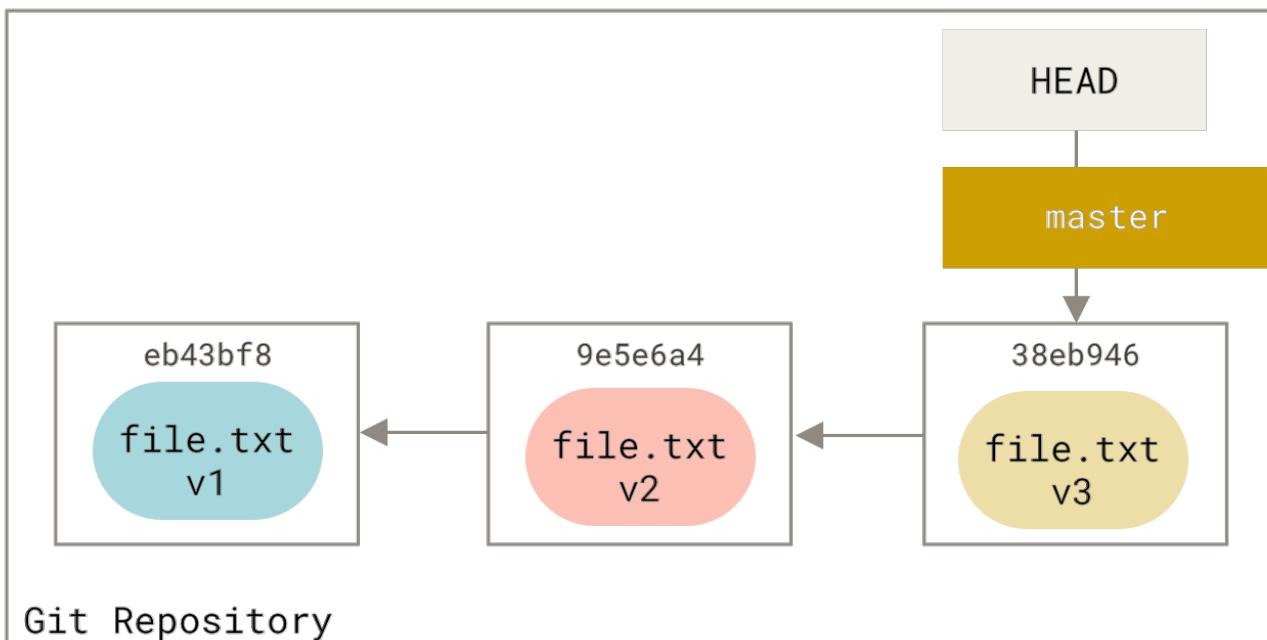
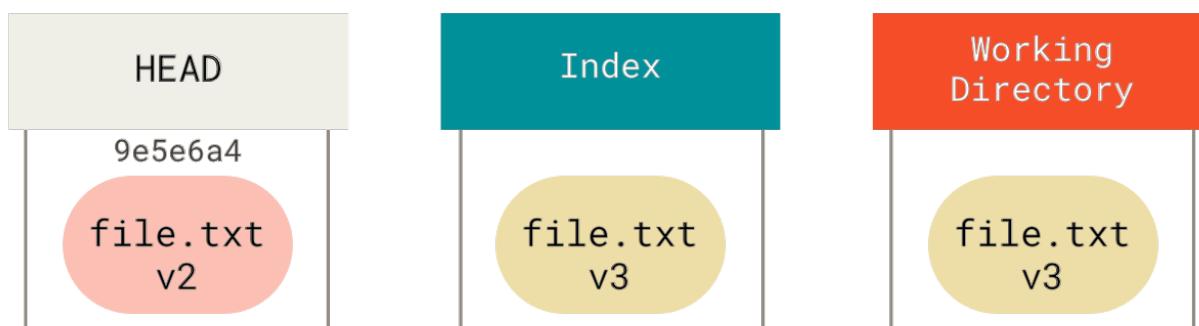
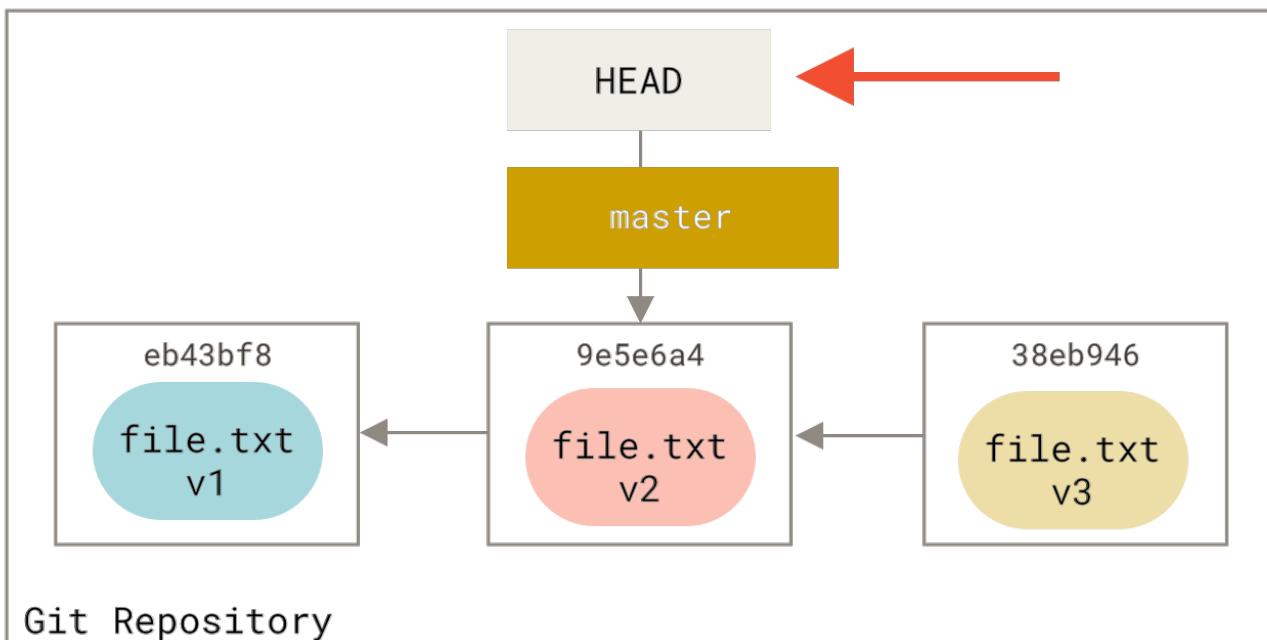


Figure 144. Git Repository mit drei Commits

Lass uns nun genau untersuchen, was `reset` bewirkt, wenn du es aufrufst. Es manipuliert die drei Bäume auf einfache und kalkulierbare Weise direkt. Es führt bis zu drei einfache Operationen aus.

### Step 1: Den HEAD verschieben

Als erstes wird `reset` das verschieben, worauf HEAD zeigt. Das ist nicht dasselbe wie HEAD selbst zu ändern (was `checkout` macht). `reset` verschiebt den Branch, auf den HEAD zeigt. Das bedeutet, wenn HEAD auf den Branch `master` gesetzt ist (d.h. du befindest dich gerade auf dem `master` Branch), wird die Ausführung von `git reset 9e5e6a4` damit starten, dass `master` auf `9e5e6a4` zeigt.



**git reset --soft HEAD~**

Figure 145. Soft Reset

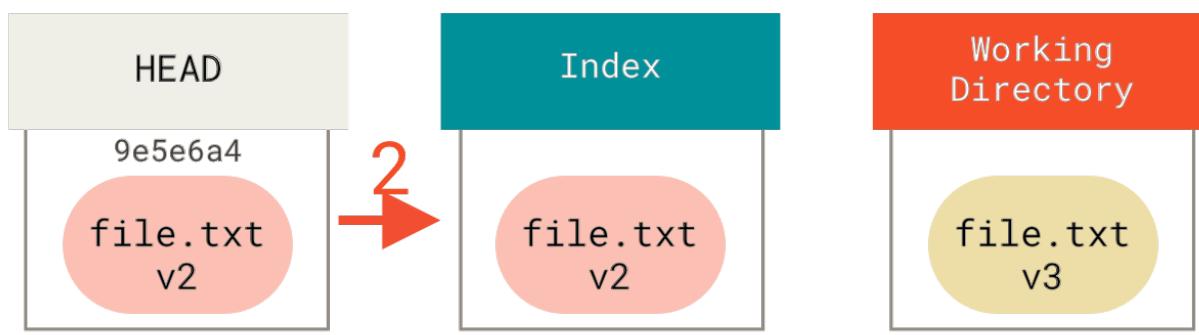
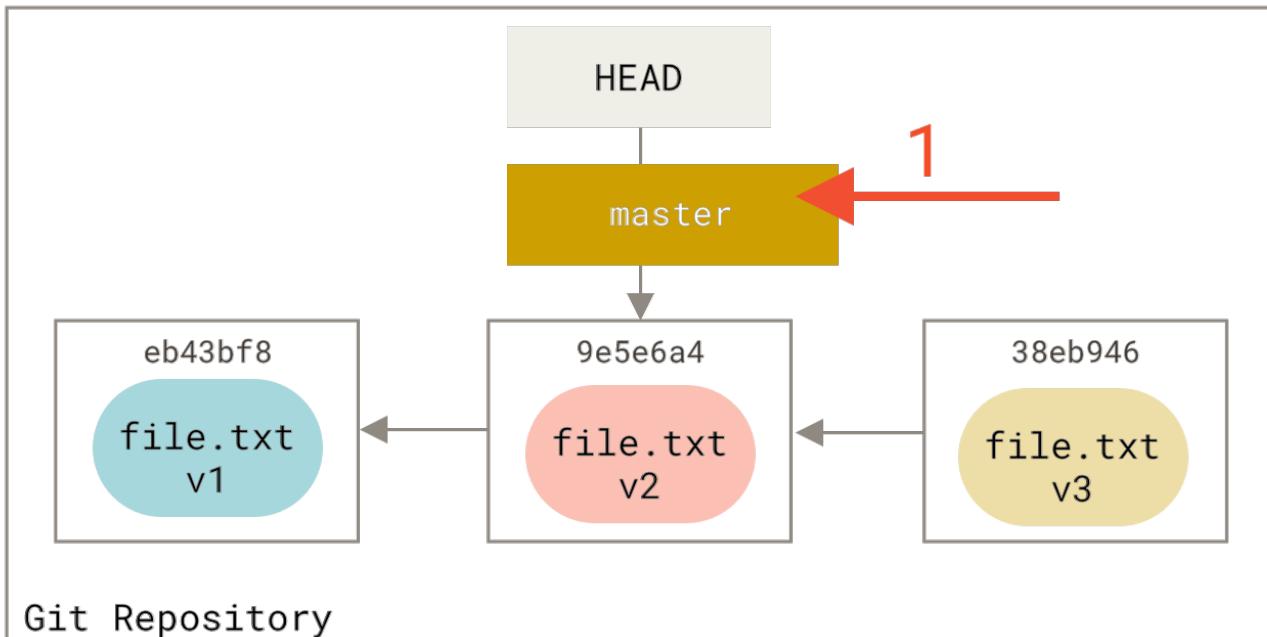
Egal, mit welcher Methode du `reset` bei einem Commit aufrufst. Dies ist immer die erste Aktion, die versucht wird auszuführen. Mit `reset --soft` wird es dort einfach stoppen.

Nimm dir nun eine Minute Zeit, um dir diese Abbildung anzusehen und dich zu fragen, was da passiert ist. Es hat im Wesentlichen den letzten `git commit` Befehl rückgängig gemacht. Wenn du `git commit` ausführst, erzeugt Git einen neuen Commit und verschiebt den Branch, auf den HEAD zeigt, dorthin. Wenn du auf `HEAD~` (das Elternteil von HEAD) zurücksetzt, verschiebst du den Branch wieder an seine ursprüngliche Stelle, ohne den Index oder das Arbeitsverzeichnis zu ändern. Du kannst nun den Index aktualisieren und `git commit` erneut ausführen, um das zu erreichen, was `git commit --amend` getan hätte (siehe auch [Den letzten Commit ändern](#)).

### Step 2: Den Index aktualisieren (--mixed)

Bitte berücksichtige, dass du bei Ausführung von `git status` in grün den Unterschied zwischen dem Index und dem neuen HEAD sehen wirst.

Als nächstes wird `reset` den Index mit dem Inhalt des Schnappschusses aktualisieren, auf den HEAD jetzt zeigt.



**git reset [--mixed] HEAD~**

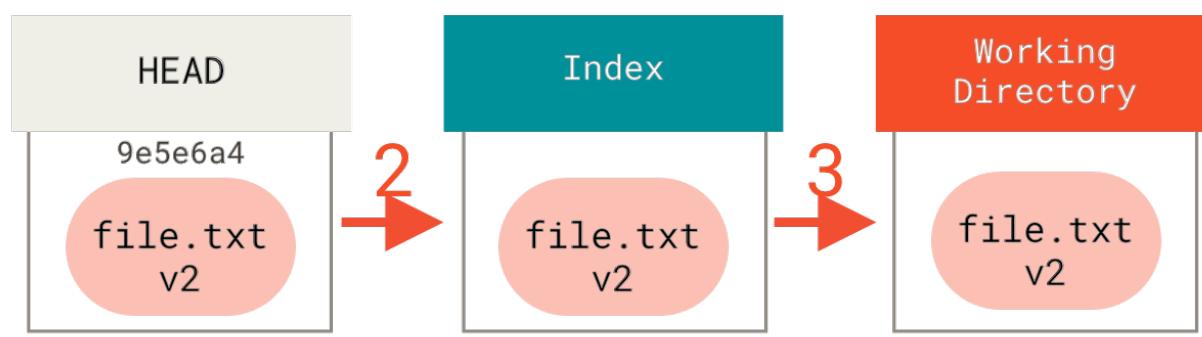
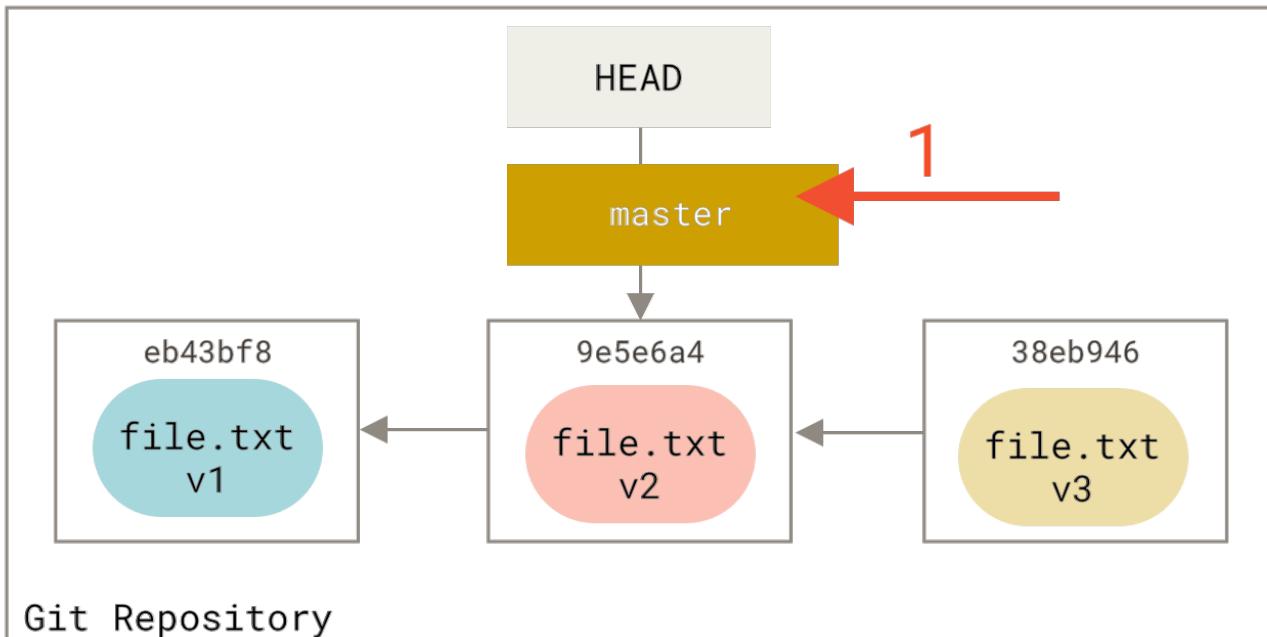
Figure 146. Mixed Reset

Wenn du die Option `--mixed` angibst, wird `reset` an dieser Stelle beendet. Das ist auch die Voreinstellung, wenn du überhaupt keine Option angibst (in diesem Fall nur `git reset HEAD~`), wird der Befehl dort enden.

Nir noch noch eine Minute Zeit, um dir jetzt diese Abbildung anzuschauen und zu erkennen, was passiert ist: Es hat deinen letzten `commit` rückgängig gemacht, aber auch alles auf `unstaged` gesetzt. Du wurdest auf den Stand zurück versetzt, bevor du alle deine `git add` und `git commit` Befehle ausgeführt hast.

### Step 3: Das Working Directory (Arbeitsverzeichnis) aktualisieren (--hard)

Als Drittes wird das Arbeitsverzeichnis durch `reset` zurückgesetzt, damit es dem Index entspricht. Wenn du die Option `--hard` verwendest, wird es bis zu diesem Schritt fortgesetzt.



**git reset --hard HEAD~**

Figure 147. Hard Reset

Denken wir also darüber nach, was gerade passiert ist. Du hast deinen letzten Commit rückgängig gemacht, die Befehle `git add` und `git commit` und dazu noch die gesamte Arbeit, die du in deinem Arbeitsverzeichnis geleistet hast.

Es ist sehr wichtig zu wissen, dass das Flag (`--hard`) die einzige Möglichkeit ist, den Befehl `reset` gefährlich zu machen und einer der wenigen Fälle, in denen Git tatsächlich Daten vernichtet. Jeder andere Aufruf von `reset` kann ziemlich leicht rückgängig gemacht werden, aber nicht die Option `--hard`, da sie Dateien im Arbeitsverzeichnis zwingend überschreibt. In diesem speziellen Fall haben wir noch immer die `v3` Version unserer Datei in einem Commit in unserer Git-Datenbank. Wir könnten sie durch einen Blick auf unser `reflog` zurückholen. Hätten wir sie aber nicht committet, dann hätte Git die Datei überschrieben und sie wäre nicht wiederherstellbar.

## Zusammenfassung

Der Befehl `reset` überschreibt diese drei Bäume in einer bestimmten Reihenfolge und stoppt, wann du es willst:

1. Verschiebe den Branch-HEAD und (*stoppt hier, wenn --soft*).

2. Lasse den Index wie HEAD erscheinen (*hier stoppen, wenn nicht --hard*).
3. Lasse das Arbeitsverzeichnis wie den Index erscheinen.

## Zurücksetzen (reset) mit Pfadangabe

Das deckt das Verhalten von `reset` in seiner Basisform ab, aber du kannst ihm auch einen Pfad angeben, auf dem er aktiv werden soll. Wenn du einen Pfad festlegst, überspringt `reset` Step 1 und beschränkt die restlichen Aktionen auf eine bestimmte Datei oder eine Gruppe von Dateien. Das macht tatsächlich Sinn. HEAD ist nur ein Pointer. Du kannst nicht auf den einen Teil eines Commits und auf einen Teil eines anderen zeigen. Der Index und das Arbeitsverzeichnis *können* jedoch teilweise aktualisiert werden, so dass das Zurücksetzen mit den Schritten 2 und 3 fortgesetzt wird.

Nehmen wir also an, wir führen ein `git reset file.txt` aus. Da du hier keinen Commit-SHA-1 oder -Branch angegeben hast und auch nicht die Optionen `--soft` oder `--hard` verwendet hast, ist das die Kurzform für `git reset --mixed HEAD file.txt`. Der Befehl wird Folgendes bewirken:

1. Verschiebt den Branch, HEAD zeigt auf (*übersprungen*).
2. Passt den Index an HEAD an (*stopt hier*).

Er kopiert also im Endeffekt nur `file.txt` von HEAD in den Index.

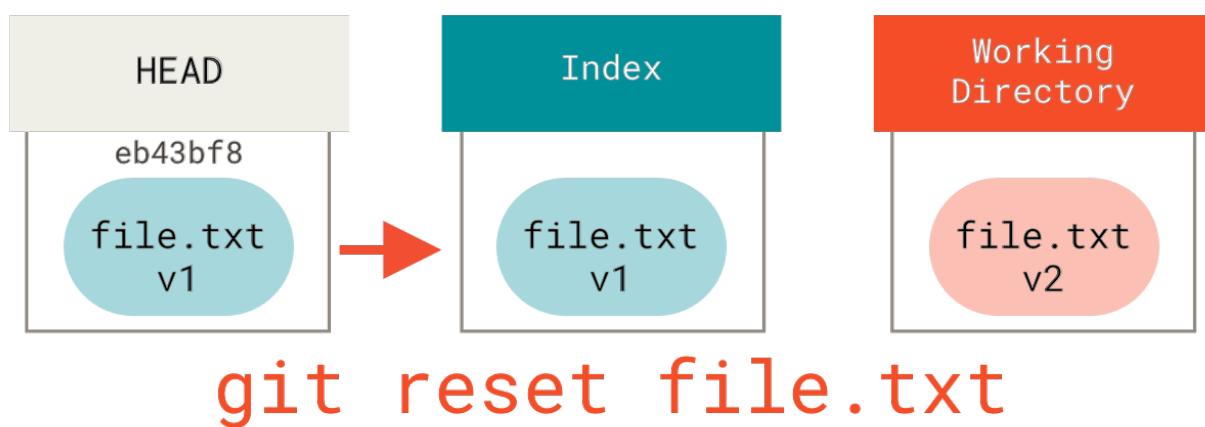
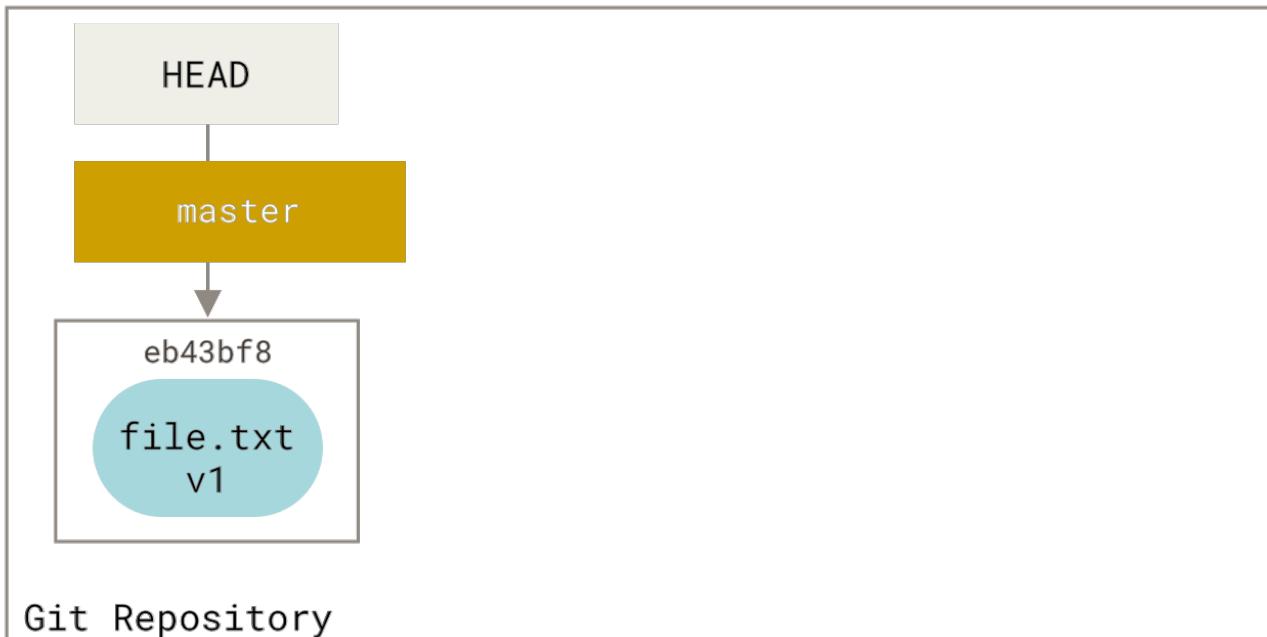


Figure 148. Mixed Reset mit einem Pfad

Das hat den praktischen Effekt, dass die Datei *aus der Staging-Area entfernt* wird (engl. *unstage*). Wenn wir uns die Abbildung für diesen Befehl ansehen und überlegen, was `git add` macht, sind die beiden Befehle genau gegensätzlich.

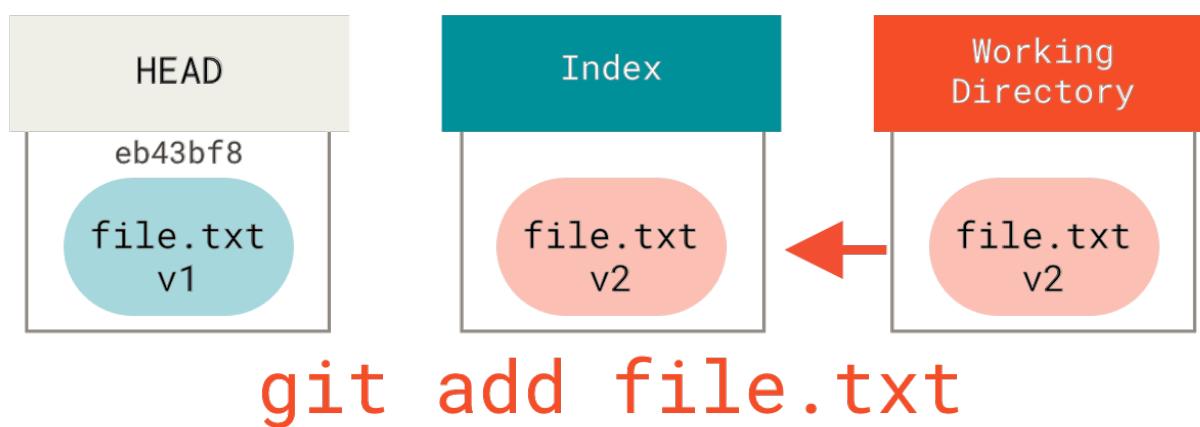
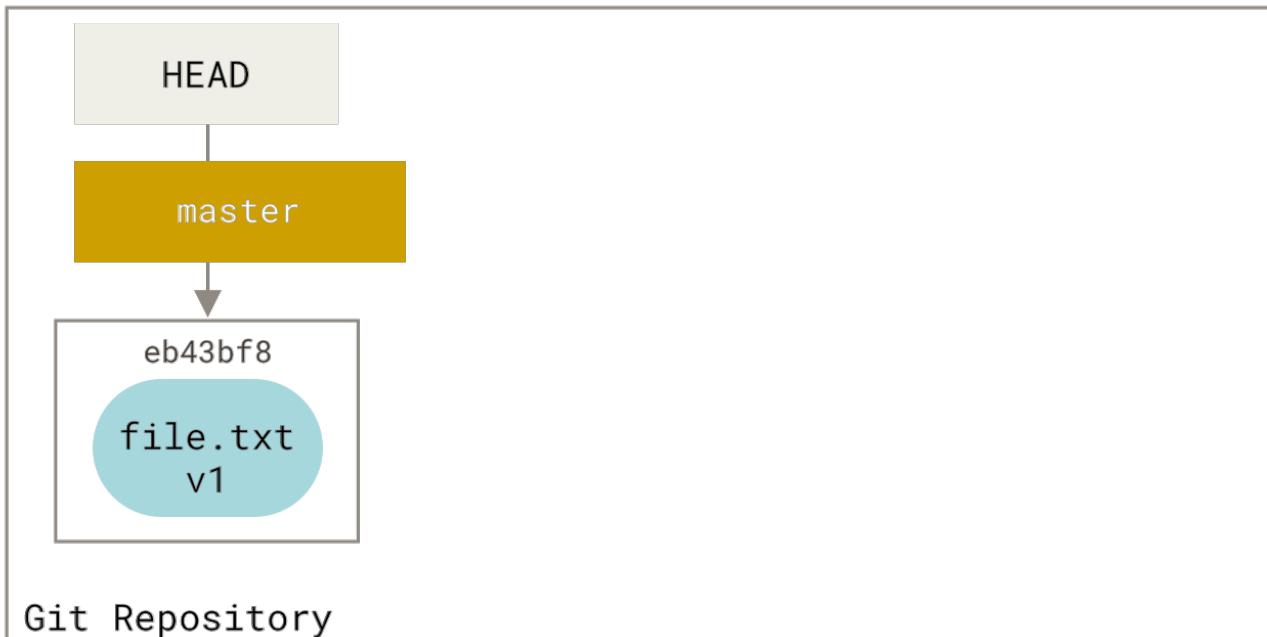
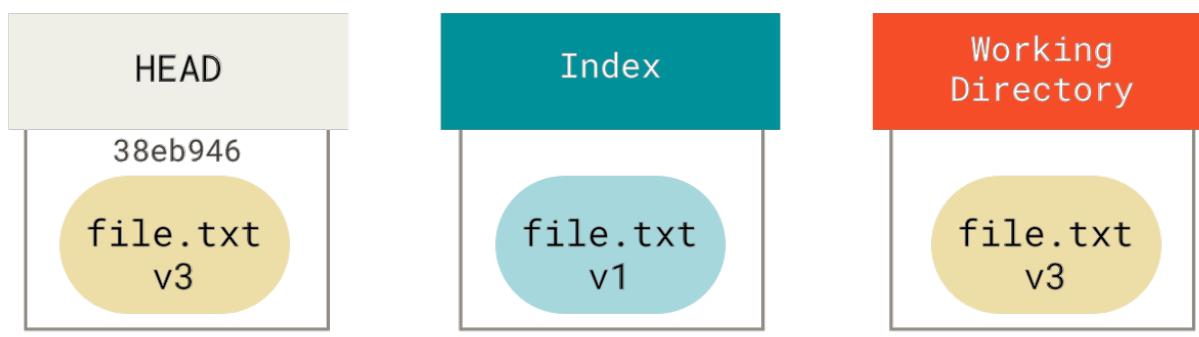
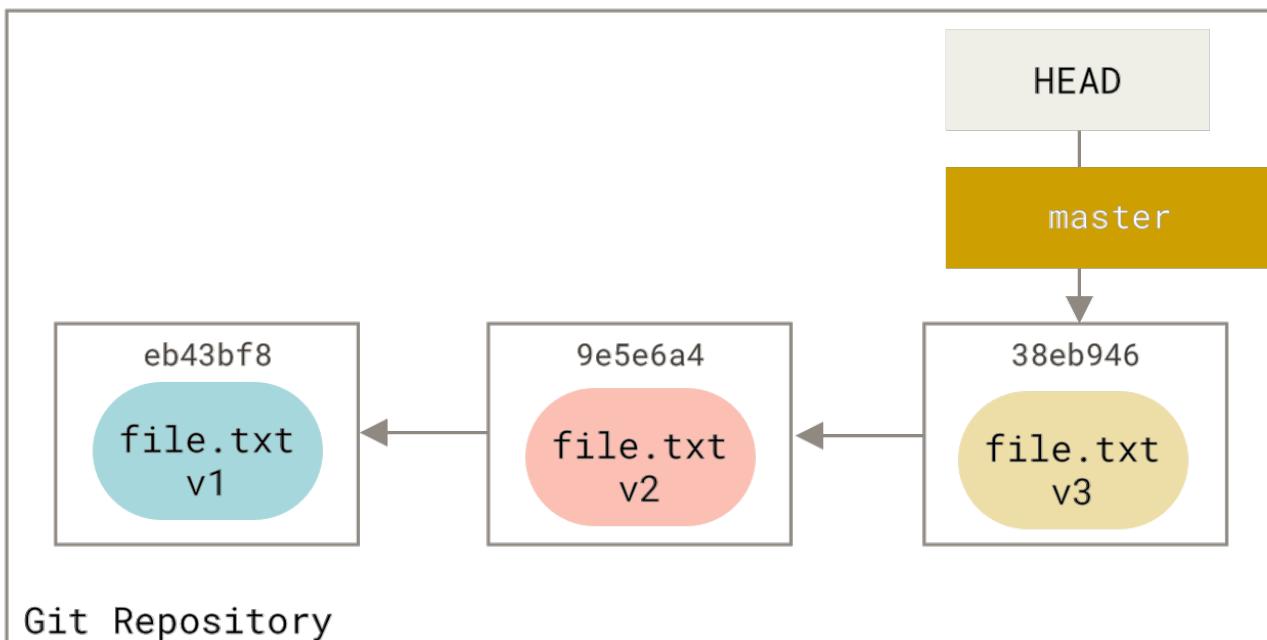


Figure 149. Datei wird auf den Index gestaged

Deshalb schlägt die Anzeige des Befehls `git status` vor, dass du den Befehl `git reset` ausführst, um eine Datei aus der Staging-Area zu entfernen. Siehe auch Kapitel 2 [Eine Datei aus der Staging-Area entfernen](#) für weitere Informationen.

Wir könnten ebenso einfach, Git nicht annehmen lassen, dass wir damit meinen, es soll „die Daten aus dem HEAD pullen“, indem wir einen bestimmten Commit angeben, aus dem diese Dateiversion gezogen werden soll. Stattdessen würden wir einfach etwas wie `git reset eb43bf file.txt` ausführen.



**git reset eb43 -- file.txt**

Figure 150. Soft Reset mit Pfad auf einen spezifischen Commit

Das macht effektiv dasselbe, als ob wir den Inhalt der Datei im Arbeitsverzeichnis auf **v1** geändert, `git add` darauf ausgeführt und dann wieder auf **v3** zurückgewandelt hätten (ohne wirklich alle diese Schritte zu durchlaufen). Wenn wir jetzt `git commit` aufrufen, wird er eine Modifikation registrieren, die diese Datei wieder auf **v1** zurücksetzt, obwohl wir sie nie wieder in unserem Arbeitsverzeichnis hatten.

Interessant ist auch, dass der `reset` Befehl wie auch `git add` die Option `--patch` akzeptiert, um Inhalte schrittweise zu entfernen. Du kannst also selektiv Inhalte aufheben oder zurücksetzen.

## Squashing (Zusammenfassen)

Schauen wir uns an, was wir mit dieser neu entdeckten Möglichkeit machen können – das Zusammenfassen von Commits.

Angenommen, du hast eine Reihe von Commits mit Nachrichten wie „Ups“, „WIP“ und „Diese Datei vergessen“. Du kannst `reset` verwenden, um diese schnell und einfach in einem einzigen Commit zusammenzufassen, der dich wirklich clever aussehen lässt. [Commits zusammenfassen](#) zeigt dir eine andere Möglichkeit auf, aber in diesem Fall ist es einfacher `reset` zu verwenden.

Stellen wir uns vor, du hast ein Projekt, bei dem der erste Commit eine Datei enthält, der zweite Commit eine neue Datei hinzufügt und die erste ändert, und der dritte Commit die erste Datei erneut ändert. Der zweite Commit war eine unfertige Arbeit und du willst diese zusammenfassen.

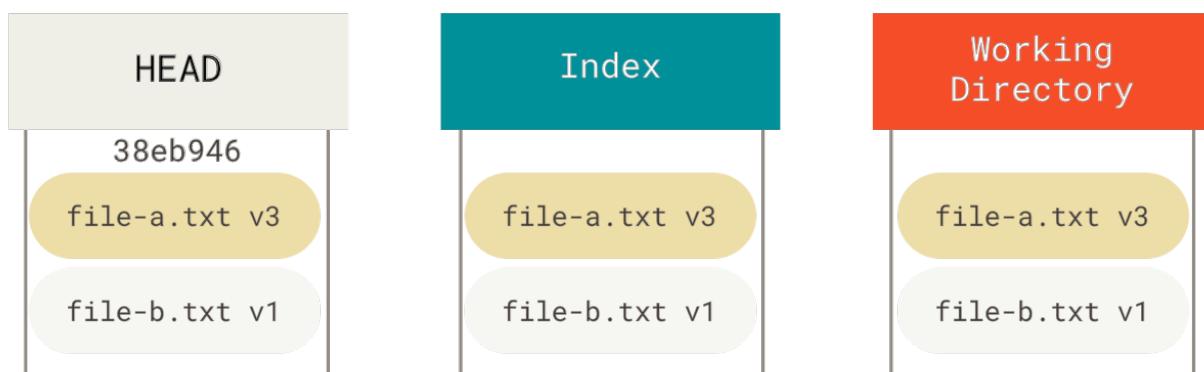
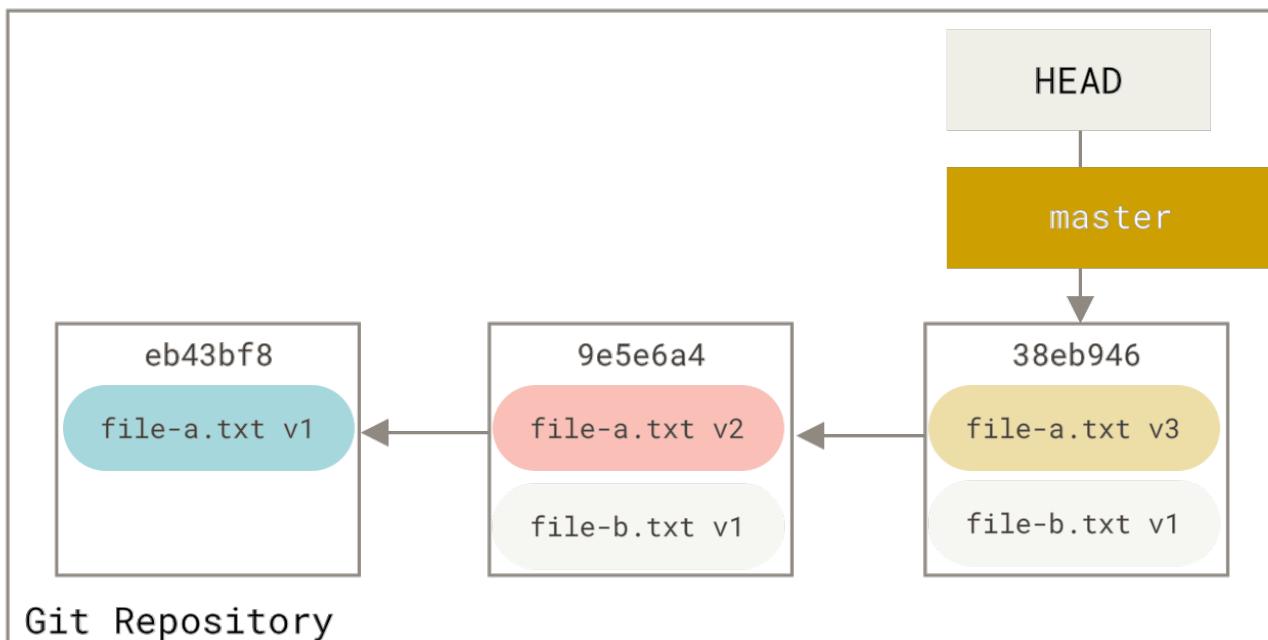
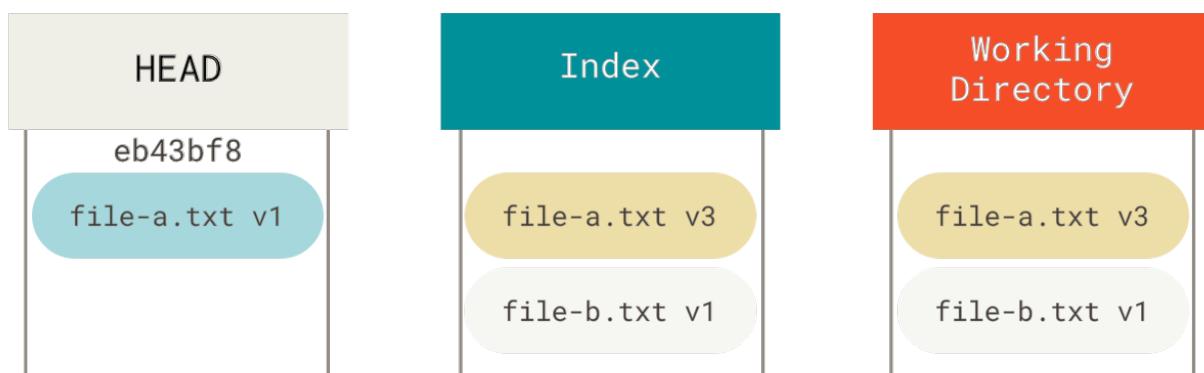
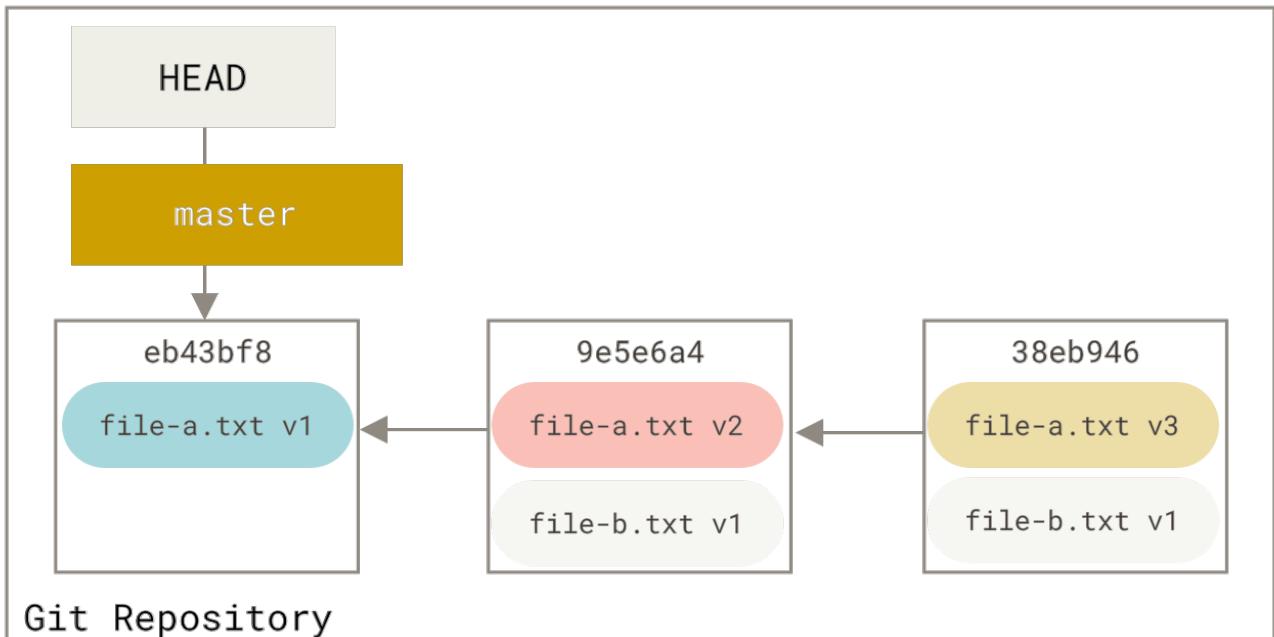


Figure 151. Git Repository

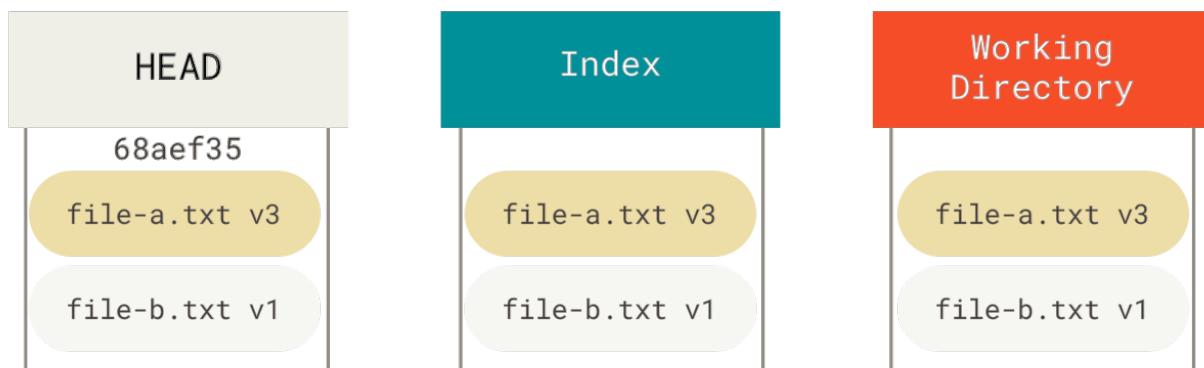
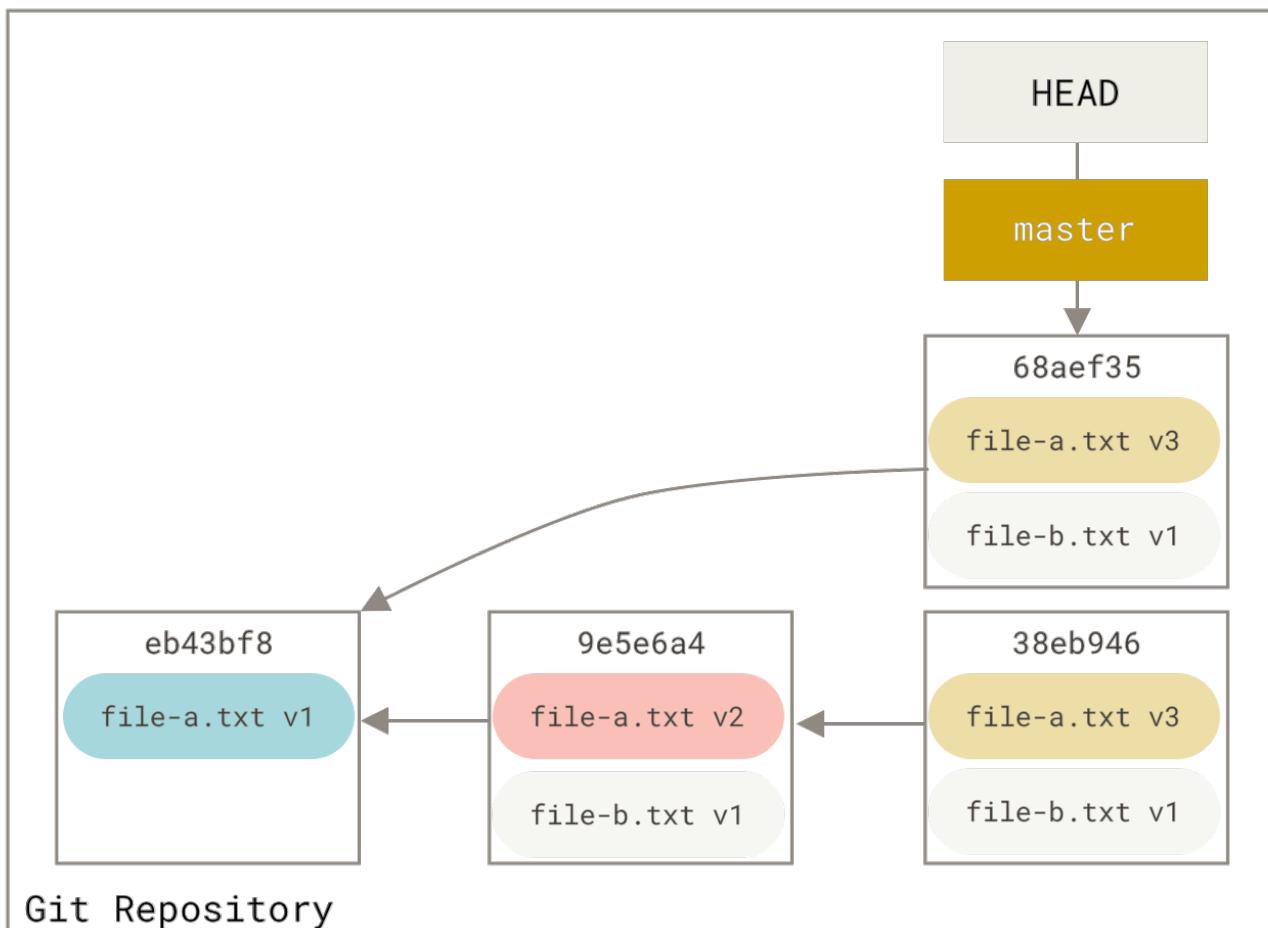
Du kannst `git reset --soft HEAD~2` ausführen, um den HEAD-Branch zurück zu einem älteren Commit (dem neuesten Commit, den du behalten willst) zu verschieben:



**git reset --soft HEAD~2**

Figure 152. HEAD Verschiebung mit Soft Reset

Danach einfach erneut `git commit` ausführen:



## git commit

Figure 153. Git Repository mit Squashed Commit

Jetzt kannst du sehen, dass dein gewünschter Verlauf, der Verlauf, den du pushen möchtest, jetzt so aussieht, als hättest du einen Commit mit `file-a.txt v1` gemacht. Anschließend hättest du einen zweiten gemacht, der sowohl `file-a.txt` zu v3 modifiziert als auch `file-b.txt` hinzugefügt hat. Der Commit mit der Version v2 der Datei ist nicht mehr im Verlauf enthalten.

## Auschecken (checkout)

Zum Schluss wirst du dich vielleicht fragen, was der Unterschied zwischen `checkout` und `reset` ist. Wie `reset` manipuliert `checkout` die drei Bäume. Es ist ein bisschen unterschiedlich, je nachdem, ob du dem Befehl einen Dateipfad mit gibst oder nicht.

## Ohne Pfadangabe

Das Benutzen von `git checkout [branch]` ist dem Ausführen von `git reset --hard [branch]` ziemlich ähnlich, da es alle drei Bäume aktualisiert, damit sie wie `[branch]` aussehen, aber es gibt zwei wichtige Unterschiede.

Erstens, anders als bei `reset --hard`, ist bei `checkout` das Arbeitsverzeichnis sicher. Es wird geprüft, ob Dateien, die Änderungen enthalten, nicht gelöscht werden. Eigentlich ist es noch etwas intelligenter. Es versucht, eine triviales Merge im Arbeitsverzeichnis durchzuführen, so dass alle Dateien, die du *nicht* geändert hast, aktualisiert werden. `reset --hard` hingegen, wird alles ohne Überprüfung einfach ersetzen.

Der zweite wichtige Unterschied ist die Frage, wie `checkout` den HEAD aktualisiert. Während `reset` den Branch verschiebt, auf den HEAD zeigt, so bewegt `checkout` den HEAD selbst, um auf einen anderen Branch zu zeigen.

Angenommen, wir haben `master` und `develop` Branches, die zu verschiedenen Commits zeigen und wir befinden uns gerade im `develop` Branch (also weist HEAD dorthin). Sollten wir `git reset master` ausführen, wird `develop` selbst nun auf den gleichen Commit zeigen, den `master` durchführt. Wenn wir stattdessen `git checkout master` ausführen, ändert sich `develop` nicht, HEAD selbst bewegt sich. HEAD zeigt nun auf `master`.

In beiden Fällen verschieben wir also HEAD, um auf Commit A zu zeigen, *aber die Methode* ist sehr unterschiedlich. `reset` verschiebt den Branch zum HEAD, `checkout` dagegen verschiebt den HEAD selbst (nicht den Branch).

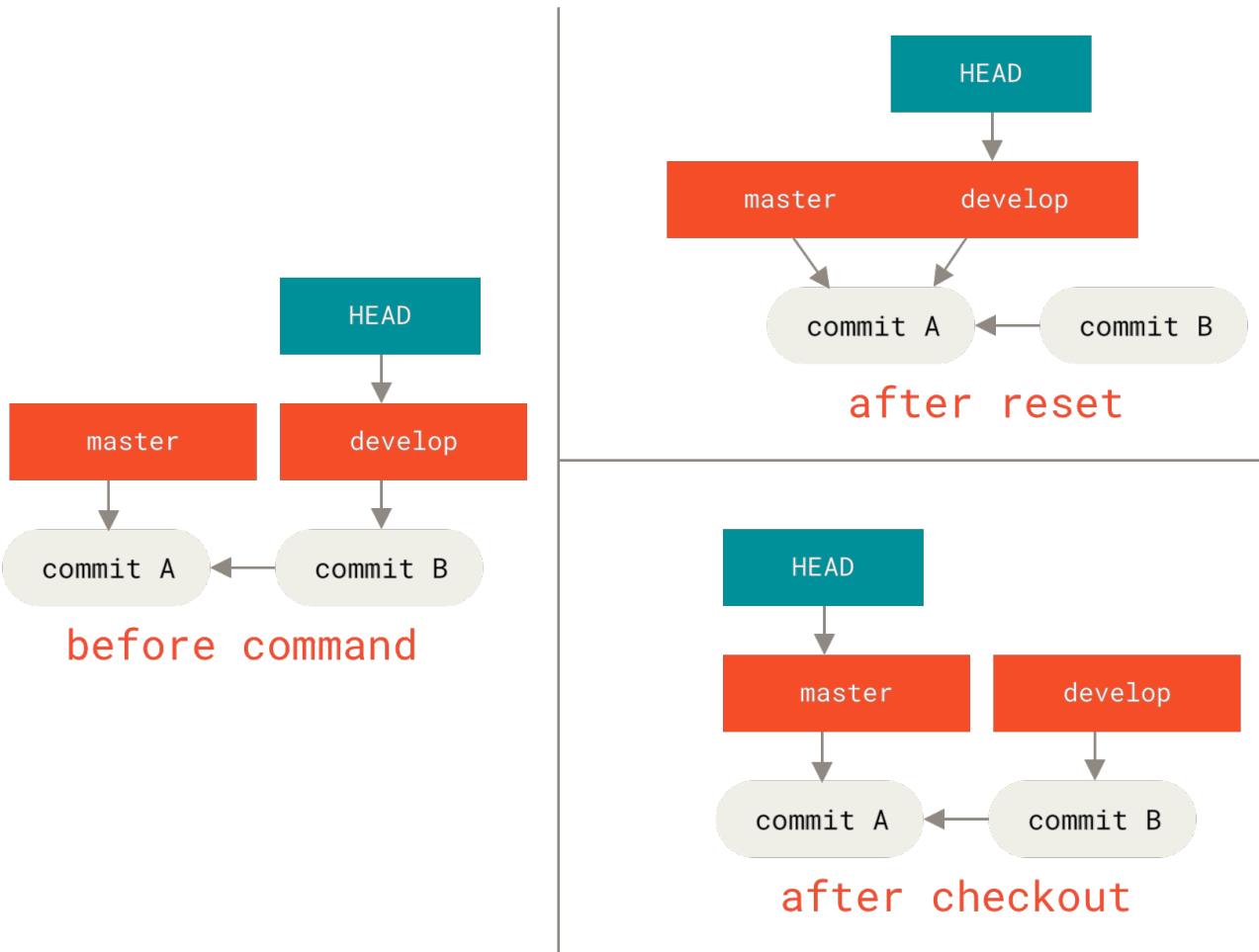


Figure 154. `git checkout` und `git reset`

### Mit Pfadangabe

Die andere Möglichkeit, das Auschecken (`checkout`) auszuführen, ist inkl. der Angabe eines Dateipfades, der, wie bei `reset`, den HEAD nicht verschiebt. Es ist genau wie bei `git reset [branch] Datei`, indem es den Index mit dieser Datei beim Commit aktualisiert, aber es überschreibt auch die Datei im Arbeitsverzeichnis. Es wäre genau wie `git reset --hard [branch] Datei` (wenn `reset` dich das ausführen lassen würde). Das Arbeitsverzeichnis ist nicht sicher und der Befehl verschiebt den HEAD nicht.

Ebenso wie `git reset` und `git add` akzeptiert `checkout` die Option `--patch`, die es dir erlaubt, den Inhalt von Dateien auf Basis von einzelnen Teilen selektiv zurückzusetzen.

### Zusammenfassung

Wir hoffen, dass du jetzt den Befehl `reset` besser kennen und anwenden kannst. Wahrscheinlich bist du aber immer noch etwas unsicher, wie genau er sich von `checkout` unterscheidet. Du kannst dir vermutlich nicht alle Regeln der verschiedenen Aufrufe merken.

Hier ist eine Tabelle, die zeigt, welche Befehle sich auf welche Bäume auswirken. In der Spalte „HEAD“ bedeutet „REF“, dass dieser Befehl die Referenz (den Branch) verschiebt, auf die HEAD zeigt. „HEAD“ signalisiert, dass er HEAD selbst verschiebt. Achte besonders auf die Spalte „WD sicher?“. Wenn dort „NO“ steht, überlege dir genau, ob du diesen Befehl ausführen willst.

	HEAD	Index	Workdir	WD sicher?
<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>

## Fortgeschrittenes Merging

Das Mergen ist mit Git in der Regel ziemlich einfach. Da es in Git recht einfach ist, einen Branch mehrfach zu mergen, kannst du manchmal sehr langlebigen Branch haben. Du solltest ihn aber während deiner Arbeit immer wieder aktualisieren und dabei mögliche kleine Konflikte lösen, anstatt am Ende der Arbeit von enorm großen Konflikten überrascht zu werden.

Manchmal kommt es jedoch zu heiklen Konflikten. Im Gegensatz zu einigen anderen Versionskontrollsystemen versucht Git nicht, bei der Lösung von Merge-Konflikten eigenständig vorzugehen. Die Philosophie von Git ist es, intelligent zu erkennen, wann eine Merge-Lösung eindeutig ist, doch wenn es einen Konflikt gibt, versucht es nicht, automatisch eine Lösung zu finden. Wenn du also zu lange mit dem Zusammenführen zweier Branches wartest, die sich auseinander entwickeln, so kannst du auf einige Probleme stoßen.

In diesem Abschnitt stellen wir dir vor, welche Probleme sich daraus ergeben könnten und welche Werkzeuge Git dir zur Verfügung stellt, um diese heiklen Situationen zu bewältigen. Wir werden auch einige der verschiedenen, atypischen Merges vorstellen, die dir zur Verfügung stehen. Außerdem werden wir dir zeigen, wie du Merges wieder rückgängig machen kannst, die bereits durchgeführt wurden.

### Merge-Konflikte

Während wir einige Grundlagen zur Lösung von Merge-Konflikten in Kapitel 3, [Einfache Merge-Konflikte](#), behandelt haben, bietet Git für komplexere Konflikte einige Werkzeuge, die dir helfen, herauszufinden, was passiert ist und wie du den Konflikt am besten lösen kannst.

Vergewissere dich zunächst einmal, dass dein Arbeitsverzeichnis sauber ist, bevor du einen Merge startest bei dem Konflikte auftreten können. Committe aktuelle Arbeiten auf einem temporären Branch oder stashe sie. Auf diese Weise kannst du **alles**, was du hier ausprobierst, wieder rückgängig machen. Falls du nicht gespeicherte Änderungen in deinem Arbeitsverzeichnis hast, während du einen Merge ausführst, können dir einige dieser Tipps helfen, diese Arbeit zu nicht zu verlieren.

Lass uns ein sehr einfaches Beispiel betrachten. Hier haben wir eine extrem einfache Ruby-Datei, die „hello world“ ausgibt.

```

#!/usr/bin/env ruby

def hello
    puts 'hello world'
end

hello()

```

In unserem Repository erstellen wir einen neuen Branch mit der Bezeichnung `whitespace`. Dann ändern wir alle Unix-Zeilenumbrüche in DOS-Zeilenumbrüche für jede Zeile der Datei, also ändern wir nur mit Leerzeichen. Dann ändern wir die Zeile „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 'Convert hello.rb to DOS'
[whitespace 3270f76] Convert hello.rb to DOS
1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

def hello
- puts 'hello world'
+ puts 'hello mundo'^M
end

hello()

$ git commit -am 'Use Spanish instead of English'
[whitespace 6d338d2] Use Spanish instead of English
1 file changed, 1 insertion(+), 1 deletion(-)

```

Jetzt wechseln wir wieder zu unserem Branch `master` und fügen einige Kommentare für die Funktion hinzu.

```

$ git checkout master
Switched to branch 'master'

$ vim hello.rb

```

```

$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
#! /usr/bin/env ruby

+# prints out a greeting
def hello
  puts 'hello world'
end

$ git commit -am 'Add comment documenting the function'
[master bec6336] Add comment documenting the function
1 file changed, 1 insertion(+)

```

Jetzt versuchen wir unseren **whitespace** branch zu mergen, es kommt zu einem Konflikt wegen der Änderungen an den Whitespaces (dt. Leerzeichen).

```

$ git merge whitespace
Auto-merging hello.rb
CONFLICT (content): Merge conflict in hello.rb
Automatic merge failed; fix conflicts and then commit the result.

```

### Einen Merge abbrechen

Wir haben jetzt einige Optionen. Zunächst einmal sollten wir uns überlegen, wie wir aus dieser Situation herauskommen können. Vielleicht hast du nicht mit Konflikten gerechnet und willst dich noch nicht ganz mit der Situation auseinandersetzen, dann kannst du einfach mit **git merge --abort** den Merge abbrechen.

```

$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master

```

Die Option **git merge --abort** versucht, zu dem Zustand zurückzukehren, der vor dem Merge bestand. Die einzigen Ausnahmen, in denen diese Funktion nicht perfekt funktioniert, wären nicht gestashete oder nicht committete Einträge in deinem Arbeitsverzeichnis. Wenn das nicht der Fall ist, sollte sie einwandfrei funktionieren.

Wenn du aus irgendeinem Grund noch einmal von vorne anfangen möchtest, kannst du auch **git**

`reset --hard HEAD` ausführen. Dein Repository wird dann wieder in den Zustand direkt nach dem letzten Commit versetzt. Denke daran, dass jede nicht committete Arbeit verloren geht. Stelle also sicher, dass du diese Arbeit wirklich nicht mehr benötigst.

## Leerzeichen ignorieren

In diesem speziellen Fall sind die Konflikte durch Leerzeichen verursacht. Dieser Konflikt ist ziemlich eindeutig. Aber auch in anderen Fällen ist der Konflikt ziemlich leicht zu erkennen, da jede Zeile auf der einen Seite gelöscht und auf der anderen Seite wieder hinzugefügt wird. Standardmäßig sieht Git alle diese Zeilen als geändert an, weshalb es die Dateien nicht miteinander mergen kann.

Der Standard Merge-Strategie kann man Argumente mitgeben. Einige beziehen sich auf die Nichtbeachtung von Leerzeichen-Änderungen. Wenn du weißt, dass du viele Leerzeichen-Änderungen bei einem Merge hast, kannst du ihn einfach abbrechen und erneut durchführen, diesmal mit den Optionen `-Xignore-all-space` oder `-Xignore-space-change`. Die erste Option ignoriert Leerzeichen beim Vergleich von Zeilen **komplett**, die zweite betrachtet Folgen von einem oder mehreren Leerzeichen als identisch.

```
$ 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(-)
```

Da in diesem Fall die tatsächlichen Dateiänderungen keinen Konflikt darstellen, mergen die einzelnen Dateien problemlos miteinander, wenn wir die Änderungen der Leerzeichen ignorieren.

Das ist sehr praktisch, falls du jemand in deinem Team hast, der gerne alles von Leerzeichen auf Tabulatoren oder umgekehrt formatiert.

## Manuelles Re-Mergen von Dateien

Obwohl Git die Behandlung von Leerzeichen vorab relativ gut verarbeitet, gibt es andere Änderungen, die Git nicht automatisch übernehmen kann, die jedoch verskriptet werden können. Nehmen wir beispielsweise an, dass Git nicht mit der Änderung der Leerzeichen umgehen konnte und wir es von Hand machen müssten.

Was wir unbedingt machen müssen, ist die Datei, die wir mergen wollen, durch das Programm `dos2unix` laufen zu lassen, noch vor dem eigentlichen Mergen der Datei. Also, wie sollten wir das realisieren?

Als Erstes geraten wir in einen Merge-Konflikt beim Aufruf des Befehls. Anschließend wollen wir Kopien von unserer Version der Datei, deren Version (aus dem Branch, in den wir mergen wollen) und der gemeinsamen Version (von der beide Seiten abstammen) bekommen. Danach sollten wir entweder deren oder unsere Seite korrigieren und den Merge für diese einzelne Datei noch ein zweites Mal versuchen.

Es ist eigentlich ganz einfach, die drei Dateiversionen zu erhalten. Git speichert alle diese Versionen

im Index unter „stages“ (dt. Stufe), die jeweils mit Ziffern versehen sind. Stage 1 ist der gemeinsame Vorgänger, Stage 2 ist deine Version und Stage 3 stammt aus dem MERGE\_HEAD, „deren“ Version, zu der die Datei(en) gemerged werden.

Du kannst eine Kopie jeder dieser Versionen der Konfliktdatei mit dem Befehl `git show` mit einer speziellen Syntax extrahieren.

```
$ git show :1:hello.rb > hello.common.rb  
$ git show :2:hello.rb > hello.ours.rb  
$ git show :3:hello.rb > hello.theirs.rb
```

Wenn du es etwas mehr Hard-Core willst, kannst du auch den Basisbefehl `ls-files -u` verwenden, um die aktuellen SHA-1s der Git-Blobs für jede dieser Dateien zu erhalten.

```
$ git ls-files -u  
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1  hello.rb  
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2  hello.rb  
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3  hello.rb
```

Das `:1:hello.rb` ist nur eine Kurzform für die Suche nach dem Blob SHA-1.

Nachdem wir jetzt den Inhalt von allen drei Stufen in unserem Arbeitsverzeichnis haben, können wir deren es manuell korrigieren, um das Problem mit den Leerzeichen zu beheben und die Datei mit dem dafür vorgesehenen, kaum bekannten Befehl `git merge-file` neu zu mergen.

```
$ 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()
```

An dieser Stelle haben wir die Datei geschickt miteinander gemerged. Das funktioniert sogar besser als die Option `ignore-space-change`, weil dadurch die Änderungen an den Leerzeichen vor dem Zusammenführen korrigiert werden, statt sie einfach zu ignorieren. Beim Merge mit `ignore-space-change` haben wir sogar ein paar Zeilen mit DOS-Zeilenden erhalten, wodurch die Sache uneinheitlich wurde.

Wenn du dir vor dem Abschluss dieses Commits ein Bild davon machen willst, was tatsächlich auf der einen oder anderen Stufe geändert wurde, kannst du mit `git diff` vergleichen, was sich in deinem Arbeitsverzeichnis befindet, das du als Ergebnis der Zusammenführung in einer dieser Stufen übergeben willst. Wir können sie alle nacheinander durchspielen.

Um das Resultat mit dem zu vergleichen, was du vor dem Merge in deinem Branch hattest oder anders gesagt, um herauszufinden, was durch den Merge entstanden ist, kannst du `git diff --ours` folgendermaßen ausführen:

```
$ 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()
```

Hier können wir leicht erkennen, dass das, was in unserem Branch entstanden ist, was wir mit diesem Merge eigentlich in diese Datei aufnehmen, diese einzelne Zeile ändert.

Wenn man sehen will, wie sich das Ergebnis des merges von dem unterscheidet, was in der Remote-Version war, kannst du `git diff --theirs` nutzen. In diesem und dem folgenden Beispiel müssen wir `-b` verwenden, um die Leerzeichen zu löschen. Wir vergleichen das Ergebnis mit dem, was in Git enthalten ist, nicht mit der bereinigten Datei `hello.theirs.rb`.

```
$ git diff --theirs -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index e85207e..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
#!/usr/bin/env ruby

+# prints out a greeting
```

```
def hello
  puts 'hello mundo'
end
```

Schließlich kannst du mit `git diff --base` sehen, wie sich die Datei beiderseits geändert hat.

```
$ 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()
```

An diesem Punkt können wir mit dem Befehl `git clean` die zusätzlichen Dateien löschen, die wir für den manuellen Merge angelegt haben und nicht mehr benötigen.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

## Die Konflikte austesten

Womöglich sind wir aus irgendeinem Grund nicht zufrieden mit der vorliegenden Lösung oder die manuelle Korrektur einer oder beider Seiten hat noch immer nicht gut funktioniert und wir benötigen weitere Informationen.

Lass uns das Beispiel etwas verändern. Hier haben wir zwei schon länger existierende Branches, die jeweils einige Commits enthalten, aber bei einem Merge einen begründeten inhaltlichen Konflikt erzeugen.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) Update README
* 9af9d3b Create README
* 694971d Update phrase to 'hola world'
| * e3eb223 (mundo) Add more tests
| * 7cff591 Create initial testing script
```

```
| * c3fffff1 Change text to 'hello mundo'  
|/  
* b7dcc89 Initial hello world code
```

Wir haben jetzt drei individuelle Commits, die nur auf dem Branch `master` existieren und drei weitere, die auf dem Branch `mundo` liegen. Wenn wir nun versuchen, den `mundo` Branch zu integrieren, bekommen wir einen Konflikt.

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

Wir möchten jetzt wissen, was den Merge-Konflikt verursacht. Beim Öffnen der Datei sehen wir etwas wie das hier:

```
#! /usr/bin/env ruby  
  
def hello  
<<<<< HEAD  
  puts 'hola mundo'  
=====  
  puts 'hello mundo'  
>>>>> mundo  
end  
  
hello()
```

Beide Seiten des Merges haben dieser Datei inhaltlich etwas verändert. Manche der Commits haben die Datei jedoch an gleicher Stelle verändert, wodurch dieser Konflikt entstanden ist.

Lass uns einige Tools näher betrachten, die dir zur Verfügung stehen, um herauszufinden, wie es zu diesem Konflikt gekommen ist. Vielleicht ist es nicht ganz eindeutig, wie genau du diesen Konflikt lösen solltest. Du benötigst weitere Informationen.

Ein hilfreiches Werkzeug ist die Funktion `git checkout` mit der Option `--conflict`. Dadurch wird die Datei erneut ausgecheckt und die Konfliktmarkierungen für den Merge-Prozess ersetzt. Das kann praktisch sein, wenn du die Markierungen zurücksetzen und erneut versuchen willst, sie aufzulösen.

Du kannst mit `--conflict` die Werte `diff3` oder `merge` (der Default-Wert) übergeben. Wenn du `diff3` übergibst, wird Git eine andere Variante von Konfliktmarkern verwenden und dir nicht nur die „ours“ and „theirs“ Version, sondern auch die „base“-Version mit zur Verfügung stellen, um dir mehr Kontext zu geben.

```
$ git checkout --conflict=diff3 hello.rb
```

Danach sollte die Datei wie folgt aussehen:

```
#!/usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=====
  puts 'hello mundo'
>>>>> theirs
end

hello()
```

Wenn dir dieses Format gefällt, kannst du es als Standard für zukünftige Merge-Konflikte festlegen, indem du die Einstellung `merge.conflictstyle` auf `diff3` setzt.

```
$ git config --global merge.conflictstyle diff3
```

Der Befehl `git checkout` kann auch die Optionen `--ours` und `--theirs` nutzen, wodurch man sehr schnell entweder nur die eine oder die andere Seite wählen kann, ohne die beiden Seiten zu vermischen.

Das kann speziell bei Konflikten mit Binärdateien nützlich sein, bei denen du einfach die eine Seite wählen kannst oder bei denen du nur bestimmte Dateien aus einem anderen Branch einbinden willst. Du kannst den Merge starten und dann bestimmte Dateien von der einen oder anderen Seite auschecken, bevor du sie committest.

## Merge-Protokoll

Ein weiteres nützliches Werkzeug bei der Lösung von Merge-Konflikten ist `git log`. So kannst du den Bezug zu dem herstellen, was zu den Konflikten beigetragen haben könnte. Es ist manchmal sehr nützlich, die Historie ein wenig Revue passieren zu lassen, um sich zu erinnern, warum zwei Entwicklungslinien den gleichen Bereich des Quellcodes tangieren.

Um eine vollständige Liste aller eindeutigen Commits zu erhalten, die in den beiden an dieser Zusammenführung beteiligten Branches enthalten waren, können wir die „triple dot“-Syntax verwenden, die wir in [Dreifacher Punkt](#) kennengelernt haben.

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 Update README
< 9af9d3b Create README
< 694971d Update phrase to 'hola world'
> e3eb223 Add more tests
> 7cff591 Create initial testing script
```

```
> c3ffff1 Change text to 'hello mundo'
```

Das ist eine detaillierte Liste der insgesamt sechs beteiligten Commits und der Entwicklungslinie, in der jeder Commit erfolgte.

Wir können das aber noch weiter vereinfachen, um uns einen viel spezifischeren Hintergrund zu geben. Fügen wir die Option `--merge` zu `git log` hinzu, zeigt sie nur die Commits auf beiden Seiten des Merges an, die eine Datei betreffen, bei der ein Konflikt vorliegt.

```
$ git log --oneline --left-right --merge
< 694971d Update phrase to 'hola world'
> c3ffff1 Change text to 'hello mundo'
```

Wenn du diesen Befehl stattdessen mit der Option `-p` ausführst, erhältst du nur die Diffs zu der Datei, die in Konflikt steht. Das kann **äußerst** wertvoll sein, um dir schnell den notwendigen Kontext zu liefern. So kannst du besser verstehen, warum sich etwas in Widerspruch befindet und wie du es auf intelligenter Weise auflösen kannst.

## Kombiniertes Diff-Format

Da Git alle Ergebnisse der erfolgreichen Merges staged, wenn du `git diff` ausführst, solange du dich in einem Merge-Konflikt-Zustand befindest, wird nur das angezeigt, was derzeit auch tatsächlich noch im Konflikt steht. Das hilft dir beim Erkennen der noch zu lösenden Fehler.

Wenn du `git diff` direkt nach einem Merge-Konflikt ausführst, erhältst du Informationen in einem ziemlich eindeutigen diff-Ausgabeformat.

```
$ 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 mundo'
+=====
+ puts 'hello mundo'
++>>>>> mundo
end

hello()
```

Das Format wird „Combined Diff“ genannt und liefert dir zwei Datenspalten neben jeder Zeile. Die erste Spalte zeigt dir an, ob diese Zeile zwischen dem Branch „ours“ und der Datei in deinem

Arbeitsverzeichnis unterschiedlich (hinzugefügt oder gelöscht) ist. Die zweite Spalte macht das Gleiche zwischen dem Branch „theirs“ und deiner Kopie des Arbeitsverzeichnisses.

In diesem Beispiel kannst du sehen, dass die Zeilen <<<<<< und >>>>> in der Arbeitskopie sind, aber nicht auf beiden Seiten des Merges stehen. Das macht Sinn, weil das Merge-Tool sie für unseren Kontext dort eingefügt hat. Es wird aber von uns erwartet, dass wir sie wieder entfernen.

Wenn der Konflikt gelöst wird und wir wieder `git diff` laufen lassen, werden wir dasselbe sehen, nun allerdings etwas hilfreicher.

```
$ 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()
```

Das zeigt uns, dass „hola world“ auf unserer Seite enthalten war, aber nicht in der Arbeitskopie, dass „hello mundo“ auf deren Seite, aber nicht in der Arbeitskopie war, und schließlich, dass „hola mundo“ auf keiner Seite existierte, aber jetzt in der Arbeitskopie ist. Es kann sinnvoll sein, dieses vor dem Committen der Lösung zu überprüfen.

Das kannst du auch aus dem `git log` für jeden Merge entnehmen, um zu verfolgen, wie etwas im Anschluss daran gelöst wurde. Git wird dieses Format dann ausgeben, wenn du `git show` bei einem Merge-Commit ausführst oder wenn du die Option `--cc` zu einem `git log -p` hinzufügst (welches standardmäßig nur Patches für Non-Merge-Commits anzeigt).

```
$ 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 annullieren

Nachdem du jetzt weißt, wie man einen Merge-Commit erstellt, wirst du vermutlich ein paar davon versehentlich machen. Eine der besten Eigenschaften von Git ist, dass es völlig in Ordnung ist, Fehler zu machen. Es ist immer möglich ist, sie rückgängig zu machen (und in vielen Fällen ist das auch sehr leicht).

Merge-Commits sind da keine Ausnahme. Angenommen, du hast mit der Arbeit an einem Feature-Branch begonnen, ihn versehentlich in `master` gemerged und jetzt sieht deine Commit-Historie so aus:

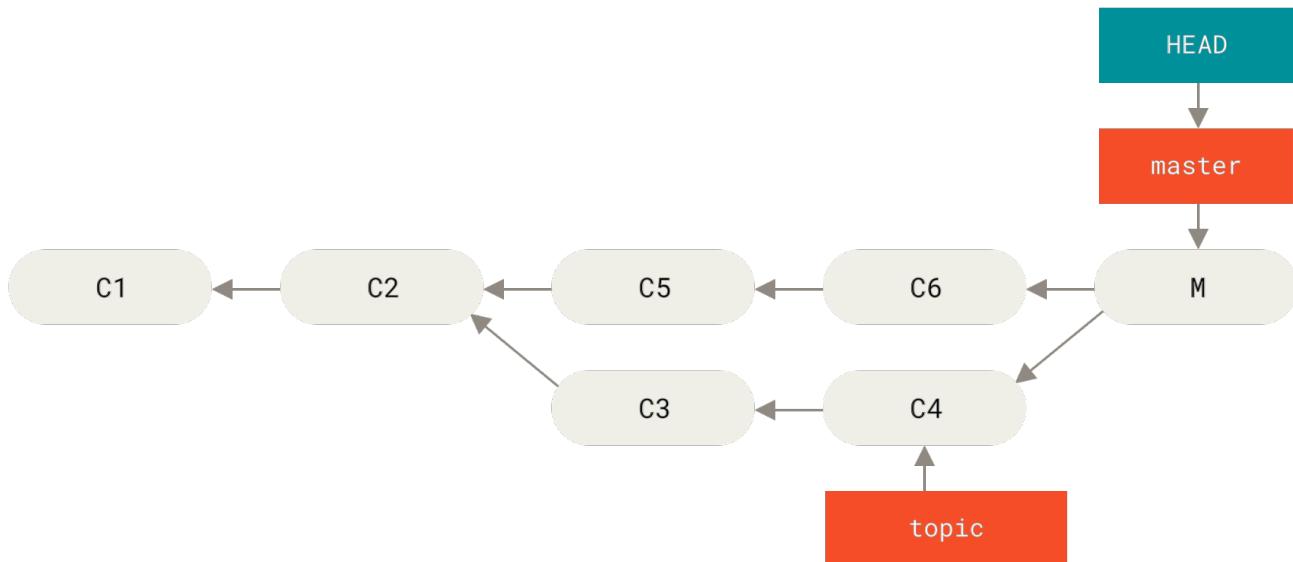


Figure 155. Versehentlicher Merge-Commit

Es gibt zwei Möglichkeiten, dieses Problem zu lösen, je nachdem, welches Ergebnis du dir vorstellst.

### Reparieren der Referenzen

Wenn der unerwünschte Merge-Commit nur auf deinem lokalen Repository existiert, ist die einfachste und beste Lösung, die Branches so zu verschieben, dass sie dorthin zeigen, wo du sie haben wolltest. Meistens, wenn dem irrtümlichen `git merge` ein `git reset --hard HEAD~` folgt, wird

das die Branch-Pointer zurücksetzen, so dass sie wie folgt aussehen:

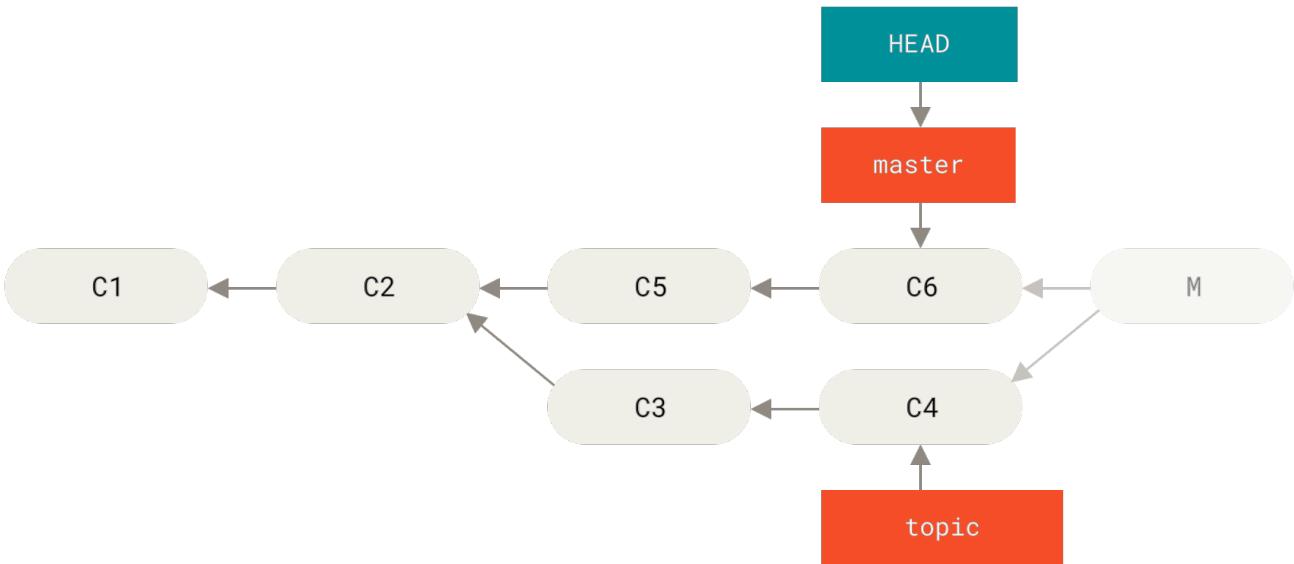


Figure 156. Verlauf nach `git reset --hard HEAD~`

Wir haben bereits in [Reset entzaubert](#) über `reset` geschrieben, also sollte es nicht allzu schwer sein, zu verstehen, was hier passiert. Hier nur zur Erinnerung: `reset --hard` durchläuft normalerweise drei Phasen:

1. Verschieben, wohin der Branch HEAD zeigt. Hier wollen wir `master` dorthin verschieben, wo er vor dem Merge-Commit stand (`C6`).
2. Den Index wie HEAD aussehen lassen.
3. Das Arbeitsverzeichnis an den Index anpassen.

Der Nachteil dieser Vorgehensweise ist, dass der Verlauf umgeschrieben wird, was mit einem gemeinsam genutzten Repository problematisch sein kann. Sieh dir [Die Gefahren des Rebasing](#) an, um Näheres zu erfahren, was passieren kann. Die Kurzfassung ist: Wenn andere Benutzer Commits vorliegen haben, die du gerade umschreiben willst, dann solltest du besser ein `reset` vermeiden. Dieser Vorgang wird auch dann nicht funktionieren, wenn seit dem Merge weitere Commits erstellt wurden. Ein Verschieben der Refs würde zum Verlust dieser Änderungen führen.

## Den Commit umkehren

Falls das Verschieben der Branch-Pointer für dich nicht in Frage kommt, bietet dir Git die Option, einen neuen Commit durchzuführen, der alle Änderungen eines bestehenden Commits rückgängig macht. Git nennt diese Operation „revert“. In diesem konkreten Szenario würdest du sie wie folgt aufrufen:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

Das `-m 1` Flag gibt an, welches Elternteil auf der „Hauptlinie“ ist und beibehalten werden sollte. Wenn du einen Merge zu `HEAD` (`git merge topic`) startest, hat der neue Commit zwei Elternteile: der erste ist `HEAD` (`C6`), und der zweite ist das Ende des Branchs, der gemerged wird (`C4`). Hier wollen wir

alle Änderungen rückgängig machen, die durch das Mergen im übergeordneten Teil #2 (**C4**) vorgenommen wurden, wobei der gesamte Inhalt des übergeordneten Teils #1 (**C6**) beibehalten wird.

Der Verlauf mit dem Revert-Commit sieht so aus:

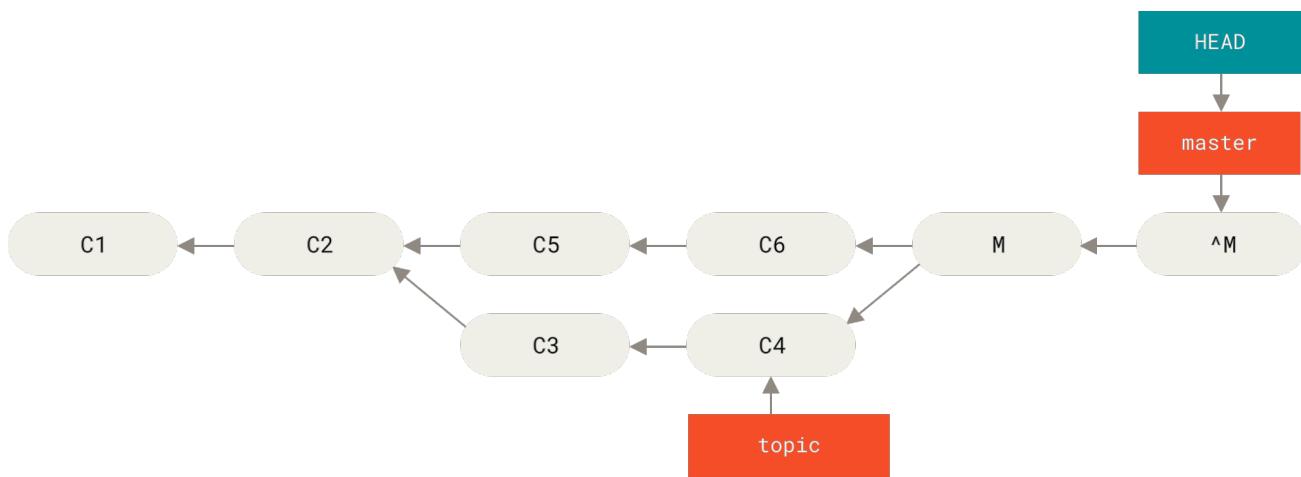


Figure 157. Verlauf nach `git revert -m 1`

Der neue Commit **^M** hat genau den gleichen Inhalt wie **C6**, so dass es von hier an so aussieht, als ob der Merge nie stattgefunden hätte, außer dass die jetzt nicht mehr gemergeten Commits noch im Verlauf von **HEAD** stehen. Git würde verwirrt werden, wenn du erneut versuchen solltest, **topic** in **master** zu mergern:

```
$ git merge topic
Already up-to-date.
```

Es gibt nichts in **topic**, was nicht schon von **master** aus erreichbar wäre. Was viel schlimmer wäre, wenn du Arbeiten zu **topic** beitragen und erneut mergen sollst. Dann wird Git nur die Änderungen seit dem rückgängig gemachten Merge vornehmen:

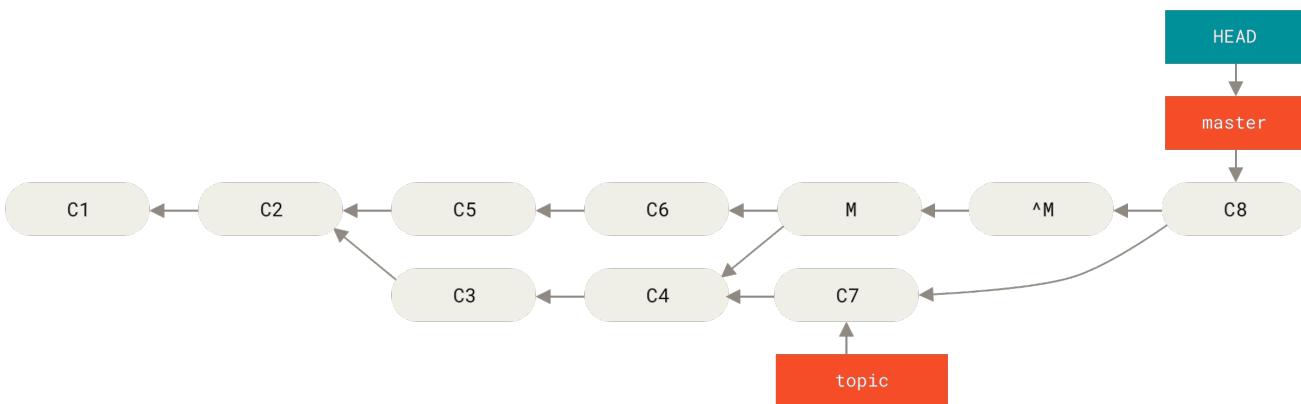


Figure 158. Verlauf mit einem fehlerhaften Merge

Der optimale Weg dies zu umgehen besteht darin, den ursprüngliche Merge rückgängig zu machen. Da du jetzt die Änderungen übernehmen möchtest, die annulliert wurden, erstellst du **dann** einen neuen Merge-Commit:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'''"
$ git merge topic
```

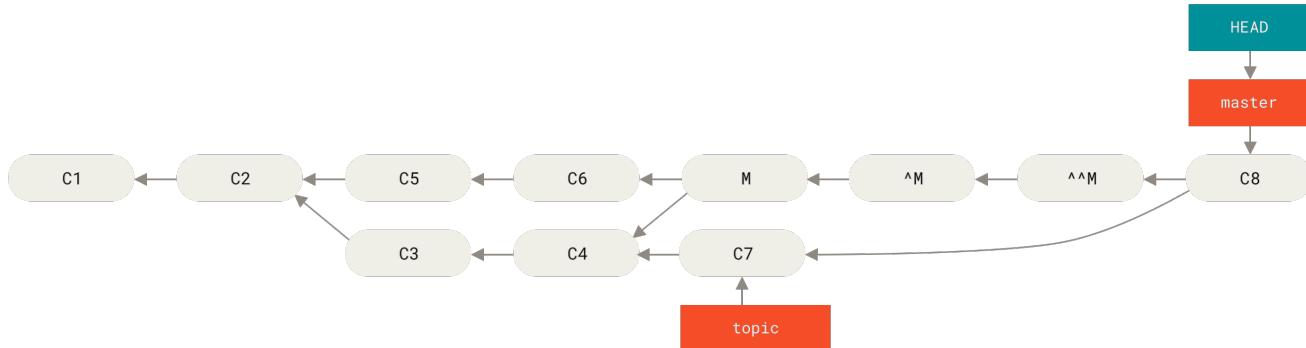


Figure 159. Verlauf nach erneutem Merge eines rückgängig gemachten Merges

In diesem Beispiel heben sich **M** und **^M** gegenseitig auf. Der Commit **^^M** merged wirksam mit den Änderungen von **C3** und **C4**, und **C8** merged mit den Änderungen von **C7**, so dass jetzt das **topic** vollständig gemerged ist.

## Andere Arten von Merges

Bisher haben wir den normalen Merge von zwei Branches behandelt, der normalerweise mit der so genannten "rekursiven" Strategie des Mergens bearbeitet wird. Es gibt jedoch auch andere Möglichkeiten, um Branches miteinander zu mergen. Lass uns kurz einige davon untersuchen.

### Unsere oder deren Präferenz

Erstens, es gibt noch eine weitere praktische Funktion, die wir mit dem normalen „rekursiven“ Modus des Merging anwenden können. Wir haben uns bereits die Optionen `ignore-all-space` und `ignore-space-change` angesehen, die mit `-X` übergeben werden. Wir können Git aber auch anweisen, die eine oder die andere Seite zu bevorzugen, falls ein Konflikt auftritt.

Wenn Git einen Konflikt zwischen zwei zu mergende Branches feststellt, fügt es standardmäßig Konfliktmarkierungen in deinem Code ein und markiert die Datei als im Konflikt stehend und ermöglicht dir, den Konflikt zu beheben. Solltest du es vorziehen, dass Git einfach eine bestimmte Version auswählt und die andere ignorieren soll, anstatt dich den Konflikt manuell auflösen zu lassen, kannst du den `merge` Befehl entweder mit `-Xours` oder `-Xtheirs` übergeben.

Erkennt Git diese Einstellung, wird es keine Konfliktmarkierungen hinzufügen. Alle miteinander in Einklang zu bringenden Abweichungen werden gemerged. Bei allen sich widersprechenden Änderungen wird einfach die von dir angegebene Version ausgewählt, einschließlich binärer Dateien.

Wenn wir auf das „hello world“-Beispiel zurückkommen, das wir zuvor benutzt haben, können wir erkennen, dass der Merge in unserem Branch Konflikte verursacht.

```
$ 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.
```

Wenn wir das Beispiel aber mit `-Xours` oder `-Xtheirs` ausführen, dann funktioniert das Merging.

```
$ 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
```

Dann würden keine Konfliktmarkierungen bei „hello mundo“ auf der einen Seite und „hola world“ auf der anderen in die Datei eingefügt, sondern einfach „hola world“ übernommen. Alle anderen, konfliktfreien Änderungen auf dieser Branch werden jedoch korrekt in die Version aufgenommen.

Diese Option kann auch an den Befehl `git merge-file` übergeben werden, den wir weiter oben beschrieben haben, als wir `git merge-file --ours` für individuelle Datei-Merges ausgeführt haben.

Wenn man so etwas machen will, aber Git nicht einmal versucht, Änderungen von der anderen Seite einzubinden, gibt es eine drastischere Option, nämlich die „ours“ Merge-Strategie. Die unterscheidet sich von der rekursiven, „ours“ Merge-Option.

Dadurch wird eigentlich ein Schein-Merge ausgeführt. Es wird ein neuer Merge-Commit mit beiden Branches als Elternteil protokolliert, aber es wird noch nicht einmal der betreffende Branch angesehen. Als Ergebnis des Merges wird einfach der exakte Code in deinem aktuellen Branch gespeichert.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

Du kannst erkennen, dass es keinen Unterschied zwischen dem Branch, auf dem wir uns befanden und dem Resultat des Merges gibt.

Oft kann das nützlich sein, um Git bei einem späteren Merge im Grunde zu täuschen, dass ein Branch bereits gemerged ist. Nehmen wir zum Beispiel an, du hast einen Branch `release` abgespalten und dort einige Änderungen vorgenommen, die du irgendwann wieder in deinem Branch `master` integrieren möchtest. Zwischenzeitlich müssen einige Bugfixes auf `master` in deinem Branch `release` zurückportiert werden. Du kannst den Bugfix-Branch in den Branch `release` mergen und ebenso den gleichen Branch mit `merge -s ours` in deinem `master` Branch mergen (auch wenn der Fix bereits vorhanden ist). Wenn du später den `release` Branch wieder mergst, gibt es keine Konflikte durch den Bugfix.

## Subtree Merging

Der Grundgedanke des Teilbaum-Merge (engl. Subtree-Merge) besteht darin, dass du zwei Projekte hast und eines der Projekte auf ein Unterverzeichnis des anderen Projekts verweist. Wenn du einen Teilbaum-Merge durchführst, ist Git so versiert zu erkennen, dass das ein Unterverzeichnis ein Teilbaum des anderen ist und es entsprechend merged.

Wir werden ein Beispiel durcharbeiten, bei dem ein separates Projekt in ein bestehendes Projekt eingefügt und dann der Code des zweiten Projekts in ein Unterverzeichnis des ersten Projekts gemerged wird.

Zunächst werden wir die Anwendung „Rack“ zu unserem Projekt hinzufügen. Wir werden das Rack-Projekt in unserem eigenen Projekt als Remote-Referenz einbinden und es dann in einem eigenen Branch auschecken:

```
$ 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"
```

Jetzt haben wir das Root-Verzeichnis des Rack-Projekts in unserem `rack_branch` und unser eigenes Projekt im `master` Branch. Wenn du die beiden Branches prüfst, kannst du sehen, dass sie unterschiedliche Projekt-Roots haben:

```
$ ls
AUTHORS      KNOWN-ISSUES  Rakefile    contrib      lib
COPYING      README        bin         example      test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Das ist irgendwie ein merkwürdiges Konzept. Nicht alle Branches in deinem Repository müssen unbedingt Branches desselben Projektes sein. Es ist nicht allgemein üblich, weil es selten hilfreich ist. Allerdings ist es ziemlich wahrscheinlich, dass die Branches völlig unterschiedliche Verläufe enthalten.

In unserem Fall wollen wir das Rack-Projekt als Unterverzeichnis in unser Projekt `master` einbringen. Das können wir in Git mit `git read-tree` tun. Mehr über den Befehl `read-tree` und seiner Verwandten erfährst du in [Git Interna](#). Vorab solltest du erfahren, dass er den Root-Tree eines Branchs in deine aktuelle Staging-Area und dein aktuelles Arbeitsverzeichnis einliest. Wir sind gerade zu deinem Branch `master` zurückgewechselt und ziehen den Branch `rack_branch` in das Unterverzeichnis `rack` unseres `master` Branchs des Hauptprojektes:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Bei einem Commit sieht es so aus, als befänden sich alle Rack-Dateien unterhalb dieses Unterverzeichnisses — als ob wir sie aus einem Tarball hineinkopiert hätten. Interessant ist, dass wir Änderungen in einem der Branches relativ leicht mit anderen Branches mergen können. Falls das Rack-Projekt aktualisiert wird, können wir die Änderungen einbinden, indem wir zu diesem Branch wechseln und pullen:

```
$ git checkout rack_branch  
$ git pull
```

Dann können wir diese Änderungen wieder in unserem Branch `master` zusammenführen. Um die Änderungen zu pullen und die Commit-Nachricht vorzubereiten, verwendet man die Option `--squash` sowie die Option `-Xsubtree` der rekursiven Merge-Strategie. Die rekursive Strategie ist hier die Voreinstellung, aber wir fügen sie der Klarheit halber ein.

```
$ 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
```

Alle Änderungen aus dem Rack-Projekt werden in das Projekt gemerged und können lokal committet werden. Du kannst auch das Gegenteil tun — Änderungen im Unterverzeichnis `rack` deines `master` Branchs vornehmen und sie dann später in deinem Branch `rack_branch` mergen, um sie den Autoren zu übermitteln oder sie zum Upstream zu pushen.

Dadurch haben wir einen Workflow, der dem Submodul-Workflow ähnelt, ohne Submodule zu verwenden (die wir in [Submodule](#) behandeln werden). Wir können Branches mit anderen verwandten Projekten in unserem Repository vorhalten und sie gelegentlich in unserem Projekt mergen. In gewisser Weise ist das nützlich. Beispielsweise kann der gesamte Code an einen einzigen Ort übermittelt werden. Allerdings gibt es auch Nachteile. Es ist etwas komplizierter und somit leichter, Fehler bei der Re-Integration von Änderungen zu machen oder versehentlich einen Branch in ein nicht relevantes Repository zu pushen.

Eine weitere etwas eigenartige Eigenschaft ist, dass du den Unterschied zwischen dem, was in deinem Unterverzeichnis `rack` steht, und dem Code in deinem Branch `rack_branch` nicht mit dem normalen `diff` Befehl erhalten können (um zu prüfen, ob du sie mergen musst). Stattdessen musst du `git diff-tree` auf dem Branch, mit dem du vergleichen willst, ausführen:

```
$ git diff-tree -p rack_branch
```

Um den Inhalt deines `rack` Unterverzeichnisses mit dem Branch `master` auf dem Remote-Server zu vergleichen, kannst du auch folgendes ausführen:

```
$ git diff-tree -p rack_remote/master
```

## Rerere

Der Befehl `git rerere` ist eine eher versteckte Funktion. Der Name steht für „reuse recorded resolution“ (dt. „gespeicherte Ergebnisse wiederverwenden“). Der Name bedeutet, dass du Git auffordern kannst sich zu erinnern, wie du einen bestimmten Konflikt in der Vergangenheit gelöst hast. Wenn Git das nächste Mal den gleichen Konflikt sieht, kann es ihn automatisch lösen.

Es gibt eine Reihe von Szenarien, in denen diese Funktionalität wirklich nützlich sein kann. Eines der Beispiele, das in der Dokumentation erwähnt wird, ist sicher zu stellen, dass ein langlebiger Feature-Branch am Ende sauber gemerged wird. Dabei willst du jedoch nicht, dass eine Menge zwischenzeitlicher Merge-Commits deine Commit-Historie durcheinander bringt. Wenn `rerere` aktiviert ist, kannst du ab und zu einen Merge starten, die Konflikte lösen und dann den Merge-Prozess stoppen. Falls du das kontinuierlich tust, sollte der finale Merge recht unkompliziert sein, denn `rerere` kann alles für dich automatisch erledigen.

Dieselbe Vorgehensweise kann angewendet werden, wenn du einen Branch rebased, damit du dich nicht jedes Mal mit denselben Konflikten beim Rebase auseinandersetzen musst. Oder wenn du einen Branch, den du schon gemerged und eine Reihe von Konflikten behoben hast dich dann jedoch ein Rebase entscheidest. Dann musst du wahrscheinlich nicht alle Konflikte nochmal lösen.

Eine weitere Einsatzmöglichkeit von 'rerere' ist, wenn man eine Reihe sich fortentwickelnden Feature-Banches gelegentlich zu einem testbaren Head zusammenfügt, so wie es das Git-Projekt oft selbst praktiziert.

Wenn die Tests fehlschlagen, kannst du die Merges rückgängig machen und sie ohne den fehlerhaften Feature-Branch, erneut starten, ohne die Konflikte erneut auflösen zu müssen.

Um die Funktion `rerere` zu aktivieren, musst du nur die folgende Config-Einstellung verwenden:

```
$ git config --global rerere.enabled true
```

Du kannst sie auch einschalten, indem du das Verzeichnis `.git/rr-cache` in einem konkreten Repository erstellst. Die Konfigurationseinstellung ist allerdings eindeutiger und aktiviert diese Funktion global.

Sehen wir uns nun ein einfaches Beispiel an, das unserem vorherigen ähnlich ist. Nehmen wir an, wir haben eine Datei namens `hello.rb`, die so aussieht:

```

#!/usr/bin/env ruby

def hello
  puts 'hello world'
end

```

In dem einen Branch ändern wir das Wort „hello“ in „hola“, in dem anderen Branch ändern wir „world“ in „mundo“, wie gehabt.

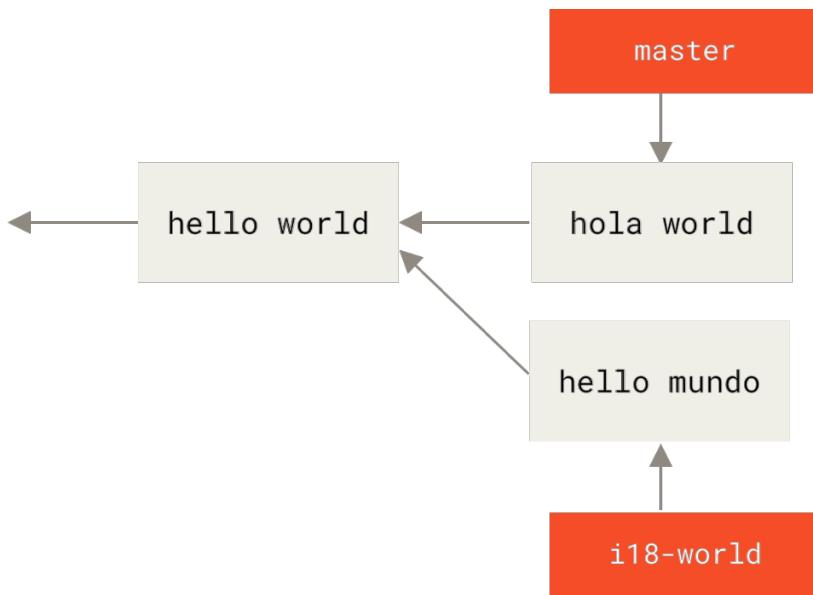


Figure 160. Zwei Branches ändern die selbe Stelle unterschiedlich

Wenn wir beiden Branches mergen, bekommen wir einen Merge-Konflikt:

```

$ 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.

```

Beachte die neue Zeile **Recorded preimage for FILE**. Der Rest sollte genauso wie bei ein normaler Merge-Konflikt aussehen. An dieser Stelle kann **rerere** uns ein paar Dinge sagen. Normalerweise kannst du an diesem Punkt **git status** ausführen, um alle Konflikte zu sehen:

```

$ 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

```

```
#
```

Allerdings wird dir `git rerere` auch mitteilen, was es im Pre-Merge Status mit `git rerere status` aufgezeichnet hat:

```
$ git rerere status  
hello.rb
```

Ein `git rerere diff` zeigt den aktuellen Status der Lösung – womit du angefangen hast und welche Lösung du gefunden hast.

```
$ 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
```

Außerdem (und das hat nicht wirklich etwas mit `rerere` zu tun) kannst du `git ls-files -u` verwenden, um dir die in Konflikt stehenden Dateien anzusehen (inklusive der vorherigen, linken und rechten Version):

```
$ git ls-files -u  
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1  hello.rb  
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2  hello.rb  
100644 54336ba847c3758ab604876419607e9443848474 3  hello.rb
```

Jetzt kannst du es einfach zu `puts 'hola mundo'` auflösen. Dann kannst du noch einmal `git rerere diff` starten, um zu sehen, woran rerere sich erinnern wird:

```
$ 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

```

Das heißt im Grunde genommen: wenn Git in einer `hello.rb` Datei, die „hello mundo“ auf der einen Seite und „hola world“ auf der anderen Seite enthält, einen Konflikt erkennt und ihn zu „hola mundo“ auflöst.

Jetzt können wir ihn als gelöst markieren und committen:

```

$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'

```

Du kannst sehen, dass es die „Lösung für DATEI gespeichert hat“ (Recorded resolution for FILE).

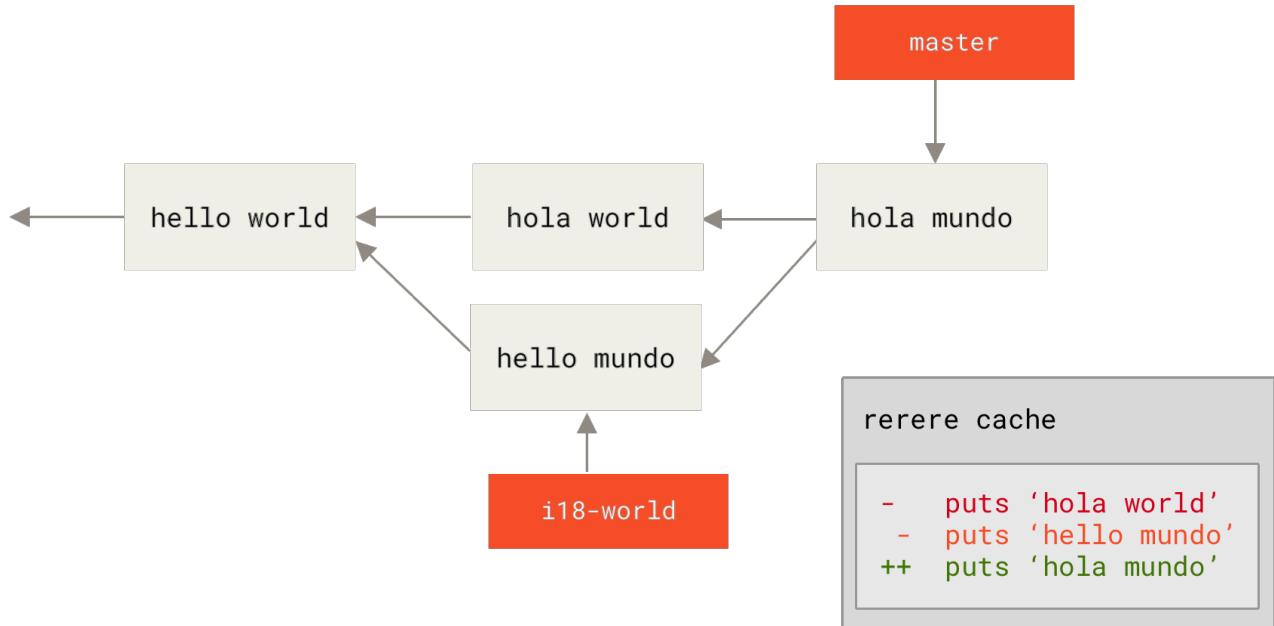


Figure 161. Aufgezeichnete Auflösung für FILE

Machen wir jetzt diesen Merge rückgängig und legen ihn stattdessen dann auf unseren Branch `master`. Wir können unseren Branch zurücksetzen, indem wir `git reset` anwenden, wie wir es in [Reset entzaubert](#) beschrieben haben.

```
$ git reset --hard HEAD^
```

```
HEAD is now at ad63f15 i18n the hello
```

Unser Merge wurde rückgängig gemacht. Lass uns jetzt den Feature-Branch 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
```

Nun haben wir den erwarteten Merge-Konflikt, aber schaue dir die Zeile **Resolved FILE using previous resolution** an. Wenn wir die Datei betrachten, sehen wir, dass der Konflikt bereits gelöst ist. Es gibt keine Marker für den Merge-Konflikt in der Datei.

```
#! /usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Zudem wird dir **git diff** zeigen, wie es erneut automatisch gelöst wurde:

```
$ 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
```

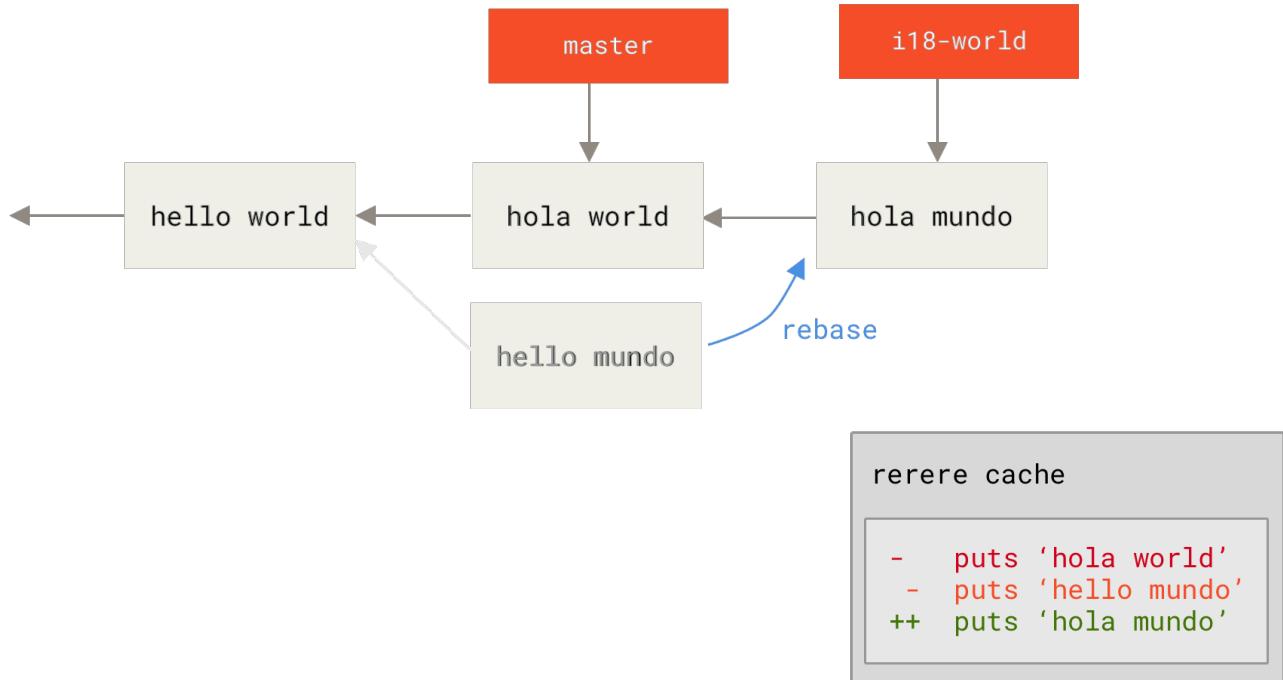


Figure 162. Automatisch aufgelöster merge Konflikt, der eine vorherige Auflösung nutzt

Du kannst den Status der Konfliktdatei auch mit `git checkout` wiederherstellen:

```

$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#!/usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>> theirs
end

```

Ein Beispiel dafür haben wir in [Fortgeschrittenes Merging](#) kennengelernt. Vorerst sollten wir das Problem dadurch lösen, dass wir `git rerere` noch einmal starten:

```

$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#!/usr/bin/env ruby

def hello
  puts 'hola mundo'
end

```

Wir haben die Datei automatisch mit der mit `rerere` zwischengespeicherten Lösung erneut gelöst.

Du kannst es nun hinzufügen und den Rebase fortsetzen, um ihn fertigzustellen.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

Wenn du also viele Re-Merges machst oder einen Topic-Branch mit deinem Branch `master` aktuell halten willst, ohne dass eine Unmenge von Merges durchgeführt werden sollen. Oder wenn du häufig einen Rebase machst, solltest du `rerere` aktivieren, um dir das Leben ein wenig leichter zu machen.

## Debuggen mit Git

Git ist zwar primär ein Tool für die Versionskontrolle, es bietet jedoch auch ein paar Befehle, die dir beim Debuggen deiner Quellcode-Projekte helfen können. Da Git für fast jede Art von Inhalt entwickelt wurde, sind diese Werkzeuge ziemlich allgemein gehalten. Sie können dir jedoch oft helfen, nach einem Fehler oder Code-Autor zu suchen, wenn etwas schief läuft.

### Datei-Annotationen

Wenn du einen Fehler in deinem Code findest und wissen willst, wann und warum er eingeführt wurde, ist die Datei-Annotation oft dein bestes Werkzeug. Es zeigt dir, welcher Commit als letzter jede Zeile einer Datei geändert hat. Wenn du also bemerkst, dass eine Methode in deinem Code fehlerhaft ist, kannst du die Datei mit `git blame` annotieren um festzustellen, welcher Commit für die Einführung dieser Zeile verantwortlich war.

Das folgende Beispiel verwendet `git blame`, um zu bestimmen, welcher Commit und Committer für die Zeilen im `Makefile` des Linux-Kernels der obersten Ebene verantwortlich war. Außerdem verwendet es die Option `-L`, um die Ausgabe der Annotation auf die Zeilen 69 bis 82 dieser Datei zu beschränken:

```
$ 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
```

Beachte, dass das erste Feld der partielle SHA-1 des Commits ist, der diese Zeile zuletzt geändert hat. Die nächsten beiden Felder sind Werte, die aus diesem Commit extrahiert wurden — der Name des Autors und das Datum dieses Commits — so dass du leicht sehen kannst, wer diese Zeile wann geändert hat. Danach folgen die Zeilennummer und der Inhalt der Datei. Beachte auch die `^1da177e4c3f4` Commit-Zeilen, wobei das `^`-Präfix Zeilen angibt, daß diese Code-Zeilen mit dem allerersten Commit des Repositorys eingeführt wurden und seitdem unverändert geblieben sind. Das ist ein bisschen verwirrend, denn jetzt hast du mindestens drei verschiedene Möglichkeiten gesehen, wie Git das Zeichen `^` verwendet, um einen Commit SHA-1 zu modifizieren. Hier ist die Bedeutung eine andere.

Eine weitere coole Sache an Git ist, dass es Dateiumbenennungen nicht explizit verfolgt. Es zeichnet die Snapshots auf und versucht dann im Nachhinein herauszufinden, was implizit umbenannt wurde. Eines der interessanten Features ist, dass man Git bitten kann, auch alle Arten von Codebewegungen herauszufinden. Wenn du `-C` an `git blame` über gibst, analysiert Git die Datei, die du annotierst und versucht herauszufinden, woher die Codeschnipsel darin ursprünglich kamen, wenn sie von woanders kopiert wurden. Nehmen wir an, du zerlegst eine Datei namens `GITServerHandler.m` in mehrere Dateien, von denen eine `GITPackUpload.m` ist. Indem du `GITPackUpload.m` mit `git blame -C` aufrufst, annst du sehen, wo Teile des Codes ursprünglich herkamen:

```
$ 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)
```

Das ist wirklich nützlich. Normalerweise erhältst du als Original-Commit den Commit, mit dem du den Code kopiert hast, denn das ist das erste Mal, dass du diese Zeilen in dieser Datei angefasst hast. Mit der Option `-C` gibt dir Git den ursprünglichen Commit, in welchem du diese Zeilen geschrieben hast, auch wenn das in einer anderen Datei war.

## Binärsuche

Das Annotieren einer Datei hilft, wenn du weißt, wo das Problem liegt. Wenn du nicht weißt, was kaputt ist und es Dutzende oder Hunderte von Commits seit dem letzten funktionierendem Zustand gab, kannst du es mit `git bisect` versuchen. Der Befehl `bisect` führt eine binäre Suche durch deine Commit-Historie durch, um dir zu helfen so schnell wie möglich zu identifizieren, welcher Commit das Problem eingeführt hat.

Nehmen wir an, du hast gerade eine Version deines Codes in eine Produktionsumgebung released. Du erhältst einen Fehlerberichte über ein Verhalten, das so nicht in deiner Entwicklungsumgebung aufgetreten ist und du kannst dir nicht erklären, warum der Code sich so verhält. Du siehst dir deinem Code an und du kannst den Fehler reproduzieren. Du kannst jedoch nicht herausfinden, was schief läuft. Du kannst den Code *teilen* (engl. bisect), um es herauszufinden. Zuerst rufst du `git bisect start` auf. Dann benutzt du `git bisect bad`, um dem System mitzuteilen, dass der aktuelle Commit, auf dem du dich befindet nicht funktioniert. Dann musst du `git bisect` sagen, wann der letzte bekannte funktionierende Zustand war, indem du `git bisect good <good_commit>` verwendest:

```
$ 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 hat herausgefunden, dass etwa 12 Commits zwischen dem Commit, den du als letzten guten Commit (v1.0) markiert hast, und der aktuellen schlechten Version liegen. Anschließend ist Git zu dem mittleren Commit gewechselt (interner `git checkout`). Jetzt kannst du deinen Test durchführen, um zu sehen, ob der Fehler zum Zeitpunkt dieses Commits existiert. Wenn ja, dann wurde er irgendwann vor diesem mittleren Commit eingeführt. Wenn nicht, dann wurde das Problem irgendwann nach dem mittleren Commit eingeführt. In diesem Beispiel stellt sich heraus, dass es hier kein Problem gibt. Du sagst Git das, indem du `git bisect good` tipps und so deine Reise fortsetzt:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] Secure this thing
```

Jetzt bist du auf einem anderen Commit. Du bist auf halbem Weg zwischen dem, den du gerade getestet hast und dem schlechten Commit. Du führst deinen Test noch einmal durch und stellst fest, dass dieser Commit fehlerhaft ist. Also sagst du Git das mit `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] Drop exceptions table
```

Dieser Commit ist in Ordnung, und jetzt hat Git alle Informationen, die es braucht, um festzustellen, wo das Problem eingeführt wurde. Es gibt dir den SHA-1 des ersten fehlerhaften Commits und zeigt einige der Commit-Informationen und welche Dateien in diesem Commit verändert wurden, so dass du herausfinden kannst, was diesen Fehler eingeführt haben könnte:

```
$ 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
```

Wenn du fertig bist, solltest du `git bisect reset` ausführen, um deinen HEAD wieder auf den Stand vor dem Start zurückzusetzen (ansonsten landest du in einem undefinierten Zustand):

```
$ git bisect reset
```

Dies ist ein mächtiges Werkzeug, das dir helfen kann, hunderte von Commits in Minuten auf einen Fehler zu überprüfen. Wenn du ein Skript hast, das mit 0 als Rückgabewert, wenn das Projekt funktioniert und mit ungleich 0 als Rückgabewert, wenn das Projekt nicht funktioniert, kannst du `git bisect` vollständig automatisieren. Zuerst teilst du Git wieder den Umfang der Bisection mit, indem du die bekannten schlechten und guten Commits angibst. Du kannst dies tun, indem du dem Befehl `bisect start` den bekannten schlechten Commit als Ersten und den bekannten guten Commit als zweiten Parameter mit gibst:

```
$ git bisect start HEAD v1.0  
$ git bisect run test-error.sh
```

Dabei wird automatisch `test-error.sh` bei jedem ausgecheckten Commit ausgeführt, bis Git den ersten fehlerhaften Commit findet. Du kannst auch etwas wie `make` or `make tests` oder was auch immer du hast, das automatische Tests für dich ausführt, nutzen.

## Submodule

Es kommt oft vor, dass du während der Arbeit an einem Projekt ein anderes Projekt verwenden musst. Möglicherweise handelt es sich dabei um eine Bibliothek, die von einem Dritten entwickelt wurde oder die du selber entwickelst und in mehreren übergeordneten Projekten verwenden willst. In diesen Szenarien tritt ein typisches Problem auf: Du möchtest die beiden Projekte getrennt halten und dennoch das eine vom anderen aus nutzen können.

Dazu ein Beispiel. Angenommen, du entwickelst eine Website und erstellst Atom-Feeds. Statt deinem eigenen Atom-generierenden Code zu schreiben, entscheidest du dich für die Verwendung einer Bibliothek. Wahrscheinlich musst du entweder den Code aus einer gemeinsam genutzten Bibliothek wie eine CPAN-Installation bzw. RubyGems einbinden oder den Quellcode in deinem eigenen Projektbaum kopieren. Die Schwierigkeit beim Einbinden von Bibliotheken besteht darin, dass es nicht einfach ist, die Bibliothek in anzupassen. Noch schwieriger ist es, sie zu deployen, da sichergestellt werden muss, dass jedem Kunden diese Bibliothek zur Verfügung steht. Das Problem mit dem Kopieren des Codes in dein eigenes Projekt ist, dass alle von dir vorgenommenen Änderungen nur schwer gemerged werden können, wenn Änderungen im Upstream verfügbar werden.

Git löst dieses Problem mit Hilfe von Submodulen. Submodule ermöglichen es dir, ein Git-Repository als Unterverzeichnis eines anderen Git-Repositorys zu führen. Dadurch kannst du ein anderes Repository in dein Projekt klonen und deine Commits getrennt halten.

## Erste Schritte mit Submodulen

Wir durchlaufen beispielhaft die Entwicklung eines einfachen Projekts, das in ein Hauptprojekt und mehrere Unterprojekte aufgeteilt wurde.

Beginnen wir mit dem Einfügen eines bestehenden Git-Repositorys als Submodul des in Arbeit befindlichen Repositorys. Ein neues Untermodul kannst du mit dem Befehl `git submodule add` und der absoluten oder relativen URL des zu trackenden Projekts hinzufügen. In diesem Beispiel fügen wir eine Bibliothek mit der Bezeichnung „DbConnector“ hinzu.

```
$ 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.
```

Standardmäßig fügt der Befehl Submodul als Subprojekt in ein Verzeichnis mit dem gleichen Namen wie das Repository, hier „DbConnector“, ein. Du kannst am Ende des Befehls einen anderen Pfad angeben, wenn du es woanders ablegen willst.

Wenn du an dieser Stelle `git status` ausführst, werden dir ein paar Dinge auffallen.

```
$ 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
```

Als erstes wirst du die neue Datei `.gitmodules` bemerken. Das ist eine Konfigurationsdatei, die die Zuordnung zwischen der URL des Projekts und dem lokalen Unterverzeichnis, in das du es kopiert hast, speichert:

```
[submodule "DbConnector"]
  path = DbConnector
  url = https://github.com/chaconinc/DbConnector
```

Wenn du mehrere Submodule hast, wirst du mehrere Einträge in dieser Datei haben. Beachte, dass

diese Datei zusammen mit deinen anderen Dateien, wie z.B. der `.gitignore` Datei, der Versionskontrolle unterliegen. Sie wird zusammen mit dem Rest deines Projekts gepusht und gepullt. So wissen auch andere Entwickler, die dieses Projekt klonen, wo sie die Submodul-Projekte beziehen können.



Da die in der `.gitmodules`-Datei enthaltene URL der Ort ist, wo Andere zuerst versuchen werden, das Submodul zu erreichen, vergewissere dich, dass du eine URL verwendest, auf die ein Zugriff möglich ist. Wenn du z.B. eine unterschiedliche URLs zum Pushen und als Andere zum Pullen verwendest, dann benutze die, auf die auch die Andere Zugriff haben. Du kannst diesen Wert lokal mit `git config submodule.DbConnector.url PRIVATE_URL` für den eigenen Einsatz überschreiben. Eine relative URL kann unter Umständen hilfreich sein.

Der andere Punkt in der Ausgabe von `git status` ist der Eintrag für den Projektordner. Wenn du darauf `git diff` ausführst, siehst du etwas Merkwürdiges:

```
$ 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
```

Obwohl `DbConnector` ein Unterverzeichnis in deinem Arbeitsverzeichnis ist, versteht es Git als ein Submodul und überwacht (engl. track) seinen Inhalt nicht, solange du dich nicht in diesem Verzeichnis befindest. Git sieht es vielmehr als einen besonderen Commit dieses Repositorys an.

Wenn du eine etwas informativere `diff`-Ausgabe benötigst, kannst du an `git diff` die Option `--submodule` übergeben.

```
$ 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)
```

Wenn du einen Commit durchführst, siehst du folgendes:

```
$ git commit -am 'Add DbConnector module'
```

```
[master fb9093c] Add DbConnector module
2 files changed, 4 insertions(+)
create mode 100644 .gitmodules
create mode 160000 DbConnector
```

Beachte den Modus **160000** für den Eintrag **DbConnector**. Das ist ein spezieller Modus in Git, der im Prinzip bedeutet, dass du einen Commit als Verzeichniseintrag und nicht als Unterverzeichnis oder Datei erfasst.

Schließlich solltest du diese Änderungen pushen:

```
$ git push origin master
```

## Ein Projekt mit Submodulen klonen

Jetzt klonen wir ein Projekt mit einem enthaltenen Submodul. Wenn du ein solches Projekt klonst, erhältst du standardmäßig die Verzeichnisse, die Submodule enthalten, aber keine der darin enthaltenen Dateien:

```
$ 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
$
```

Das Verzeichnis **DbConnector** ist vorhanden, aber leer. Du musst deshalb zwei Befehle ausführen: **git submodule init**, um deine lokale Konfigurationsdatei zu initialisieren und **git submodule update**, um alle Daten dieses Projekts zu fatchen und die entsprechenden Commits auszuchecken, die in deinem Hauptprojekt aufgelistet sind:

```
$ 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'
```

Nun hat dein Unterverzeichnis `DbConnector` exakt denselben Inhalt, als du es zum letzten Mal committed hast.

Es gibt noch eine andere, unkompliziertere Möglichkeit. Wenn du dem Befehl `git clone` die Option `--recurse-submodules` übergibst, wird jedes Submodul im Repository automatisch initialisiert und aktualisiert. Einschließlich verschachtelter Submodule, falls eines der Submodule im Repository selbst Submodule hat.

```
$ 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'
```

Wenn du das Projekt bereits geklont und `--recurse-submodules` vergessen hast, kannst du die Schritte `git submodule init` und `git submodule update` zur Aktualisierung auch kombinieren, indem du `git submodule update --init` aufrufst. Damit auch verschachtelte Submodule initialisiert, gefetched und ausgecheckt werden, kannst du das narrensichere `git submodule update --init --recursive` verwenden.

## Arbeiten an einem Projekt mit Submodulen

Wir haben jetzt eine Projektkopie mit Submodulen und werden sowohl beim Haupt- als auch beim Submodulprojekt mit unseren Teamkollegen zusammenarbeiten.

## Übernehmen von Upstream-Änderungen aus dem Remote-Submodul

Das Einfachste bei der Verwendung von Submodulen in einem Projekt, ist ein Subprojekt nur zu benutzen und gelegentlich Aktualisierungen zu erhalten, die aber nichts an deinem Checkout ändern. Gehen wir ein einfaches Beispiel durch.

Wenn du in einem Untermodul nach neuen Inhalten suchen willst, kannst du ins entsprechende Verzeichnis gehen und `git fetch` und `git merge` auf dem Upstream-Branch ausführen, um den lokalen Code zu aktualisieren.

```
$ 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(+)
```

Wenn du nun zurück in das Hauptprojekt gehst und `git diff --submodule` ausführst, kannst du sehen, dass das Submodul aktualisiert wurde und erhältst eine Liste der Commits, die ihm hinzugefügt wurden. Um nicht jedes Mal, wenn du `git diff` ausführst, `--submodule` einzugeben, kannst du es als Standardformat festlegen, indem du den Konfigurationswert `diff.submodule` auf „log“ setzen.

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

Wenn du nun committest, zwingst du das submodule neuen code zu zu beinhalten, wenn andere Personen ihn updaten.

Es gibt auch einen einfacheren Weg, das zu erreichen. Falls du es vorziehst, nicht manuell in das Unterverzeichnis zu fatchen und zu mergen. Wenn du `git submodule update --remote` ausführst, wird Git in dein Submodul wechseln und die Aktualisierung abholen und durchführen.

```
$ 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'
```

Dieser Befehl geht standardmäßig davon aus, dass du den Checkout auf den default branch des entfernten Submodul-Repositorys aktualisieren möchtest (derjenige, auf den **HEAD** auf remote zeigt). Du kannst es, sofern du möchtest, auch anders konfigurieren. Wenn du beispielsweise möchtest, dass das DbConnector-Submodul den Branch „stable“ dieses Repositorys tracken soll, dann kannst du es entweder in deiner **.gitmodules** Datei eintragen (damit jeder andere es auch trackt) oder es einfach in deiner lokalen **.git/config** Datei setzen. Lass es uns in der **.gitmodules** Datei einrichten:

```
$ 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'
```

Wenn du die Option **-f .gitmodules** weglässt, wird die Änderung nur für dich vorgenommen. Es ist aber wahrscheinlich sinnvoller, diese Informationen im Repository zu speichern, damit alle anderen es genauso machen.

Wenn wir hier **git status** ausführen, wird Git uns anzeigen, dass wir „neue Commits“ im Submodul haben.

```
$ 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")
```

Wenn du die Konfigurationseinstellung **status.submodulesummary** setzt, zeigt dir Git auch eine kurze Übersicht der Änderungen in den Submodulen an:

```
$ 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
```

Sobald du jetzt `git diff` ausführst, kannst du erkennen, dass sowohl unsere `.gitmodules` Datei modifiziert wurde und dass es eine Reihe von Commits gibt, die wir gepulkt haben und die bereit sind, an unser Submodul-Projekt committet zu werden.

```
$ 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
```

Das ist ziemlich beeindruckend, denn wir können das Log-Protokoll der Commits sehen, die wir in unserem Submodul vornehmen wollen. Nach dem erfolgten Commit kannst du diese Informationen auch nachträglich anzeigen lassen, indem du `git log -p` aufrufst.

```
$ 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
```

Standardmäßig wird Git versuchen, **alle** Submodule zu aktualisieren, wenn du `git submodule update --remote` ausführst. Wenn du viele Submodule hast, solltest du also den genauen Namen des Submoduls angeben, das du gerade aktualisieren möchtest.

### Upstream-Änderungen des Projekts vom Remote aus pullen

Lass uns nun aus Sicht deiner Mitstreiters agieren, der einen eigenen lokalen Klon des Hauptprojekt-Repositorys besitzt. Einfach nur `git pull` auszuführen, um die von dir eingereichten Änderungen abzurufen, wird nicht ausreichen:

```
$ git pull
From https://github.com/chaconinc/MainProject
  fb9093c..0a24cfc master      -> origin/master
Fetching submodule DbConnector
From https://github.com/chaconinc/DbConnector
  c3f01dc..c87d55d stable      -> origin/stable
Updating fb9093c..0a24cfc
Fast-forward
 .gitmodules      | 2 ++
 DbConnector      | 2 ++
 2 files changed, 2 insertions(+), 2 deletions(-)

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

    modified:   DbConnector (new commits)

Submodules changed but not updated:

* DbConnector c87d55d...c3f01dc (4):
  < catch non-null terminated lines
  < more robust error handling
  < more efficient db routine
  < better connection routine
```

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

Der Befehl `git pull` holt standardmäßig rekursiv die Änderungen der Submodule, wie wir in der Ausgabe des ersten Befehls oben sehen können. Er **aktualisiert** die Submodule jedoch nicht. Dies wird durch die Ausgabe des `git status` Befehls gezeigt, aus dem hervorgeht, dass das Submodul „modifiziert“ ist und „neue Commits“ enthält. Außerdem zeigen die spitzen Klammern der neuen Commits nach links (<) und bedeuten, dass diese Commits im Hauptprojekt, aber nicht im lokalen Checkout von DbConnector vorhanden sind. Um das Update abzuschließen, musst du `git submodule update` ausführen:

```
$ git submodule update --init --recursive
Submodule path 'vendor/plugins/demo': checked out
'48679c6302815f6c76f1fe30625d795d9e55fc56'

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

Sicherheitshalber solltest du `git submodule update` mit dem `--init` Flag ausführen. Für den Fall, dass das Hauptprojekt einen Commit durchführt, bei dem du gerade neue Submodule hinzugefügt hast. Wenn ein Submodul verschachtelte Submodule enthält, solltest du das `--recursive` Flag setzen.

Wenn du diesen Prozess automatisieren möchtest, kannst du das Flag `--recurse-submodules` zum Befehl `git pull` hinzufügen (seit Git 2.14). Dadurch wird Git dazu veranlasst `git submodule update` direkt nach dem Pull-Kommando zu starten, wodurch die Submodule in die korrekte Version versetzt werden. Wenn du in Git immer mit dem Flag `--recurse-submodules` pullen willst, kannst du die Konfigurations-Option `submodule.recurse` auf `true` setzen (dies funktioniert für `git pull` seit Git 2.15). Diese Option bewirkt, dass Git das Flag `--recurse-submodules` für alle Befehle verwendet, die es unterstützen (außer `clone`).

Es gibt eine besondere Situation, die beim Abrufen von Aktualisierungen des Hauptprojekts auftreten kann: Es könnte sein, dass das Upstream-Repository die URL des Submoduls in der Datei `.gitmodules` in einem der von dir abgerufenen Commits geändert hat. Das kann zum Beispiel passieren, wenn das Submodul-Projekt seine Hosting-Plattform ändert. In diesem Fall ist es möglich, dass `git pull --recurse-submodules` oder `git submodule update` fehlschlägt, wenn das Hauptprojekt auf einen Submodul-Commit verweist, der nicht in dem lokal konfigurierten Submodul in deinem Repository gefunden wird. Um diese Situation zu beheben, ist der Befehl `git submodule sync` erforderlich:

```
# copy the new URL to your local config
$ git submodule sync --recursive
# update the submodule from the new URL
$ git submodule update --init --recursive
```

## An einem Submodul arbeiten

Wahrscheinlich verwendest du Submodule, weil du gleichzeitig am Code im Submodul und im Hauptprojekt (oder modulübergreifend in mehreren Submodulen) arbeiten willst. Andernfalls würdest du wahrscheinlich stattdessen ein einfacheres System zur Verwaltung von Abhängigkeiten (wie Maven oder Rubygems) verwenden.

Betrachten wir nun ein Beispiel, bei dem Änderungen am Submodul gleichzeitig mit dem Hauptprojekt vorgenommen werden und diese Änderungen zur gleichen Zeit committed und veröffentlicht werden.

Wenn wir bisher den Befehl `git submodule update` ausgeführt haben, um Änderungen aus den Submodul-Repositorys zu holen, würde Git die Änderungen erhalten und die Dateien im Unterverzeichnis aktualisieren, aber das Sub-Repository in einem sogenannten "detached HEAD" Status belassen. Das bedeutet, dass es keinen lokalen Arbeits-Branch (wie z.B. den `master`) gibt, der die Änderungen trackt. Ohne einen Arbeits-Branch verändert sich die Nachverfolgung, d.h. selbst wenn du Änderungen an das Untermodul committest, gehen diese Änderungen möglicherweise verloren, wenn du das nächste Mal `git submodule update` ausführst. Du musst einige zusätzliche Schritte ausführen, wenn du willst, dass diese Änderungen in einem Submodul getrackt werden.

Um dein Submodul so einzurichten, dass du es leichter kannst, musst du zwei Dinge tun. Du musst in jedes Submodul gehen und einen Branch auschecken, an dem du arbeiten willst. Dann musst du Git mitteilen, was zu tun ist, wenn du Änderungen vorgenommen hast. Anschließend pullt der Befehl `git submodule update --remote` neue Daten vom Upstream. Die Optionen sind, dass du diese entweder in deine lokale Bearbeitung mergst oder du kannst deine lokale Arbeit auf die neuen Änderungen rebasen.

Gehen wir zunächst in unser Submodul-Verzeichnis und wechseln in einen Branch.

```
$ cd DbConnector/  
$ git checkout stable  
Switched to branch 'stable'
```

Versuchen wir, unser Submodul mit der Option „merge“ zu aktualisieren. Wenn du es manuell starten möchtest, können wir einfach die Option `--merge` zu unserem `update` Aufruf hinzufügen. Wir erkennen hier, dass es eine Änderung auf dem Server für dieses Submodul gegeben hat und es wird zusammengeführt.

```
$ cd ..  
$ git submodule update --remote --merge  
remote: Counting objects: 4, done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 4 (delta 2), reused 4 (delta 2)  
Unpacking objects: 100% (4/4), done.  
From https://github.com/chaconinc/DbConnector  
  c87d55d..92c7337  stable    -> origin/stable  
Updating c87d55d..92c7337  
Fast-forward  
  src/main.c | 1 +
```

```
1 file changed, 1 insertion(+)
Submodule path 'DbConnector': merged in '92c7337b30ef9e0893e758dac2459d07362ab5ea'
```

Das Verzeichnis DbConnector hat die neuen Änderungen bereits in unserem lokalen **stable** Branch gemerged. Jetzt wollen wir beobachten, was passiert, wenn wir unsere eigene lokale Änderung an der Bibliothek vornehmen und jemand anderes gleichzeitig eine andere Änderung im Upstream-Bereich vornimmt.

```
$ cd DbConnector/
$ vim src/db.c
$ git commit -am 'Unicode support'
[stable f906e16] Unicode support
1 file changed, 1 insertion(+)
```

Bei der Aktualisierung unseres Submoduls erfahren wir jetzt, was passiert, wenn wir eine lokale Änderung vorgenommen haben und auch im Upstream-Bereich eine Änderung vorliegt, die wir einbauen müssen.

```
$ cd ..
$ git submodule update --remote --rebase
First, rewinding head to replay your work on top of it...
Applying: Unicode support
Submodule path 'DbConnector': rebased into '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Wenn du **--rebase** oder **--merge** vergessen hast, wird Git einfach das Submodul auf das aktualisieren, was auch immer auf dem Server vorhanden ist und dein Projekt in einen abgekoppelten (engl. detached) HEAD-Zustand zurücksetzen.

```
$ git submodule update --remote
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

Wenn das passiert, brauchst du dir keine Sorgen zu machen. Du kannst einfach ins Verzeichnis zurückgehen und deinen Branch wieder auschecken (der immer noch deine Arbeit enthält) und **origin/stable** (oder welchen Remote-Branch auch immer) manuell mergen oder rebasen.

Solltest du deine Änderungen nicht in deinem Submodul committed haben und ein Submodul-Update ausführen, das Probleme verursachen würde, holt Git die gemachten Änderungen, überschreibt aber nicht die ungesicherte Arbeit in deinem Submodul-Verzeichnis.

```
$ 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'
```

Wenn du Änderungen vorgenommen hast, die mit einer Änderung im Upstream-Bereich in Konflikt stehen, wird Git dich darüber informieren, sobald du das Update ausführst.

```
$ 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'
```

Du kannst in das Submodul-Verzeichnis wechseln und den Konflikt wie gewohnt beheben.

## Änderungen am Submodul veröffentlichen

Wir haben jetzt einige Änderungen in unserem Submodul-Verzeichnis gemacht. Einige davon wurden von unseren Updates aus dem Upstream-Bereich eingebracht, andere wurden lokal vorgenommen und stehen noch niemandem zur Verfügung, da wir sie noch nicht gepusht haben.

```
$ git diff  
Submodule DbConnector c87d55d..82d2ad3:  
  > Merge from origin/stable  
  > Update setup script  
  > Unicode support  
  > Remove unnecessary method  
  > Add new option for conn pooling
```

Wenn wir im Hauptprojekt einen Commit machen und ihn nach pushen, ohne gleichzeitig die Änderungen an den Submodulen zu pushen, werden andere Entwickler, die versuchen, unsere Änderungen zu überprüfen, in Schwierigkeiten geraten, da sie keine Möglichkeit haben, die davon abhängigen Änderungen an den Submodulen zu erhalten. Diese Änderungen werden nur auf unserer lokalen Kopie existieren.

Um sicherzustellen, dass dies nicht passiert, kannst du Git auffordern, zu überprüfen, ob alle deine Submodule ordnungsgemäß gepusht wurden, bevor du das Hauptprojekt pushst. Der Befehl `git push` übernimmt das Argument `--recurse-submodules`, das entweder auf „check“ oder „on-demand“ (dt. bei Bedarf) gesetzt werden kann. Die Option „check“ lässt `push` einfach fehlschlagen, wenn eine der eingereichten Submodul-Änderungen noch nicht gepusht wurde.

```
$ 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.

Wie du feststellen kannst, erhalten wir dadurch auch einige hilfreiche Tipps, was wir als nächstes tun könnten. Die einfache Lösung besteht darin, in jedes Submodul zu wechseln und manuell auf den Remote zu pushen. So stellst du sicher, dass die Submodule extern verfügbar sind. Danach kann man diesen Push erneut versuchen. Wenn du möchtest, dass dieses Verhalten für alle Pushs durchgeführt wird, kannst du dieses Vorgehen zur Standardeinstellung machen. Dazu musst du den Befehl `git config push.recurseSubmodules check` ausführen.

Die andere Möglichkeit ist die Verwendung der Option „on-demand“, die das für dich erledigt.

```
$ 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
```

Wie du oben sehen konntest, ist Git in das DbConnector-Modul gewechselt und hat es gepusht, bevor das Hauptprojekt gepusht wurde. Wenn dieser Push des Submoduls aus irgendeinem Grund fehlschlägt, wird auch der Push des Hauptprojekts fehlschlagen. Du kannst dieses Verhalten zur Standardeinstellung machen, indem du `git config push.recurseSubmodules on-demand` ausführst.

## Änderungen an den Submodulen mergen

Wenn du die Referenz eines Submoduls gleichzeitig mit anderen Personen änderst, könntest du auf Probleme stoßen. Sollten die Historien der Submodule divergieren und du dorthin zu einem

Hauptprojekt mit divergierenden Branches committed haben, kann es schwierig sein, das zu beheben.

Ist einer der Commits ein direkter Vorgänger des anderen (ein fast-forward Merge), dann wählt Git einfach den letzteren für den Merge aus, so dass das gut funktioniert.

Git wird nicht einmal einen trivialen Merge versuchen. Wenn die Submodule Commits divergieren und gemerged werden müssen, erhältst du folgende Ausgabe:

```
$ 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.
```

Im Wesentlichen ist hier Folgendes passiert: Git hat herausgefunden, dass die beiden Branches, im Verlauf der Submodule, Einträge aufzeichnen, die voneinander abweichen und gemerged werden müssen. Es erklärt es als „merge following commits not found“ (Merge nach Commits nicht gefunden), was erstmal verwirrend erscheint. Wir werden gleich erklären, warum das so passiert.

Um das Problem zu lösen, musst du ermitteln, in welchem Status sich das Submodul befinden sollte. Merkwürdigerweise liefert Git hier nicht wirklich viele Informationen, die dir helfen können, nicht einmal die SHA-1s der Commits beider Seiten der Historie. Glücklicherweise ist es aber ganz leicht, das herauszufinden. Wenn du `git diff` ausführst, kannst du die SHA-1s der Commits der beiden Branches, die du zusammenführen willst, anzeigen lassen.

```
$ git diff
diff --cc DbConnector
index eb41d76,c771610..0000000
--- a/DbConnector
+++ b/DbConnector
```

In diesem Fall ist `eb41d76` der Commit in unserem Submodul, den **wir** hatten, und `c771610` ist der Commit, den der Upstream hatte. Wenn wir in unser Submodul-Verzeichnis gehen, sollte es bereits `eb41d76` enthalten, da es durch den Merge nicht angetastet wurde. Sollte das aus irgendeinem Grund nicht der Fall sein, kannst du einfach einen neuen Branch erstellen und auschecken, der auf dieses Verzeichnis zeigt.

Wichtig ist der SHA-1 des Commits von der Gegenseite. Diesen wirst du mergen und auflösen müssen. Du kannst entweder direkt versuchen, den Merge mit dem SHA-1 durchzuführen oder du kannst einen Branch dafür erstellen und dann versuchen, diesen zu mergen. Wir empfehlen

letzteres – und sei es nur, um eine bessere Merge-Commit-Meldung zu erhalten.

Wir werden also in unser Submodul-Verzeichnis wechseln, einen Branch namens „try-merge“, basierend auf diesem zweiten SHA-1 aus `git diff` erstellen und manuell mergen.

```
$ cd DbConnector  
  
$ git rev-parse HEAD  
eb41d764bccf88be77aced643c13a7fa86714135  
  
$ git branch try-merge c771610  
  
$ git merge try-merge  
Auto-merging src/main.c  
CONFLICT (content): Merge conflict in src/main.c  
Recorded preimage for 'src/main.c'  
Automatic merge failed; fix conflicts and then commit the result.
```

Wir haben hier einen echten Merge-Konflikt. Wenn wir diesen lösen und beheben, dann können wir das Hauptprojekt einfach mit dem Resultat updaten.

```
$ 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 Zuerst lösen wir den Konflikt.

② Dann wechseln wir zurück zum Hauptprojekt-Verzeichnis.

③ Wir könnten den SHA-1 noch einmal überprüfen.

④ Wir lösen den Konflikt im Submodul-Eintrag.

⑤ Wir committen unseren Merge.

Es kann etwas verwirrend sein, aber es ist eigentlich nicht sehr schwer.

Interessanterweise gibt es einen weiteren Fall, den Git handhaben kann. Wenn ein Merge-Commit im Submodul-Verzeichnis existiert, der **beide** Commits in seinem Verlauf enthält, wird Git dir dies als mögliche Lösung vorschlagen. Git sieht, dass irgendwann im Submodul-Projekt jemand Branches gemerged hat, die diese beiden Commits enthalten. Möglicherweise möchtest du also diesen einen haben wollen.

Deshalb lautete die Fehlermeldung von vorhin „merge following commits not found“, weil **dies** nicht möglich war. Es ist verwirrend, denn wer würde erwarten, dass es das **versucht**?

Wenn Git einen einzelnen, akzeptablen Merge-Commit findet, siehst du wahrscheinlich Folgendes:

```
$ 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.
```

Der von Git vorgeschlagene Befehl wird den Index aktualisieren, als ob du **git add** (was den Konflikt aufhebt) und dann **commit** ausführen würdest. Allerdings solltest du das nicht tun. Du kannst genauso einfach in das Submodul-Verzeichnis wechseln, den Unterschied prüfen, zu diesem Commit springen, ihn ordnungsgemäß testen und ihn dann committen.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward

$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forward to a common submodule child'
```

Damit wird dasselbe erreicht, aber zumindest kannst du auf diese Weise überprüfen, ob es wirklich funktioniert und du hast den Code in deinem Submodul-Verzeichnis, wenn du fertig bist.

## Tipps für Submodule

Es gibt ein paar Dinge, die du tun kannst, um dir die Arbeit mit den Untermodulen ein wenig zu

erleichtern.

## Submodul Foreach

Es gibt das Submodul-Kommando `foreach`, um in jedem Submodul ein beliebiges Kommando auszuführen. Das kann wirklich hilfreich sein, wenn du mehrere Submodule im gleichen Projekt hast.

Nehmen wir zum Beispiel an, wir wollen ein neues Feature starten oder einen Bugfix durchführen und arbeiten an mehreren Submodulen. Wir können leicht die gesamte Arbeit in all unseren Submodulen 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
```

Dann können wir einen neuen Branch erstellen und von allen unseren Submodulen zu diesem wechseln.

```
$ git submodule foreach 'git checkout -b featureA'  
Entering 'CryptoLibrary'  
Switched to a new branch 'featureA'  
Entering 'DbConnector'  
Switched to a new branch 'featureA'
```

Du verstehst die Idee dahinter? Eine wirklich sinnvolle Methode, die dir hilft, ein gutes, einheitliches Diff zwischen den Änderungen in deinem Hauptprojekt und all deinen Subprojekten zu erstellen.

```
$ git diff; git submodule foreach 'git diff'  
Submodule DbConnector contains modified content  
diff --git a/src/main.c b/src/main.c  
index 210f1ae..1f0acdc 100644  
--- a/src/main.c  
+++ b/src/main.c  
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)  
  
    commit_pager_choice();  
  
+    url = url_decode(url_orig);  
+  
/* build alias_argv */  
alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));  
alias_argv[0] = alias_string + 1;
```

```

Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
    return url_decode_internal(&url, len, NULL, &out, 0);
}

+char *url_decode(const char *url)
+{
+    return url_decode_mem(url, strlen(url));
+}
+
char *url_decode_parameter_name(const char **query)
{
    struct strbuf out = STRBUF_INIT;

```

Hier kannst du sehen, dass wir eine Funktion in einem Submodul definieren und sie im Hauptprojekt aufrufen. Das ist natürlich ein vereinfachtes Beispiel, aber es gibt dir hoffentlich eine Vorstellung davon, wie hilfreich das sein könnte.

## Nützliche Aliase

Vielleicht möchtest du Aliase für manche dieser Befehle einrichten, da sie ziemlich lang sein können. Du kannst für die meisten dieser Befehle keine Konfigurationsoptionen festlegen, um sie zu Standardeinstellungen zu machen. Wir haben die Einrichtung von Git-Aliassen in [Git Aliases](#) behandelt. Hier ist ein Beispiel dafür, was du vielleicht einrichten solltest, wenn du viel mit Submodulen in Git arbeiten willst.

```

$ git config alias.sdiff '!'"git diff && git submodule foreach 'git diff''"
$ git config alias.spush 'push --recurse-submodules=on-demand'
$ git config alias.supdate 'submodule update --remote --merge'

```

Auf diese Weise kannst du einfach `git supdate` ausführen, wenn du deine Submodule aktualisieren willst. Oder du kannst `git spush` nutzen, um einen Push mit der Überprüfung der Submodul-Abhängigkeiten durchzuführen.

## Probleme mit Submodulen

Die Verwendung von Submodulen ist jedoch nicht ohne Schwierigkeiten.

### Branches wechseln

So kann beispielsweise das Wechseln von Branches mit darin enthaltenen Submodulen bei älteren Git-Versionen (vor Git 2.13) etwas knifflig sein. Wenn du einen neuen Branch erstellst, dort ein Submodul hinzufügst und dann wieder zu einem Branch ohne dieses Submodul wechselst, hast du darin das Submodul-Verzeichnis immer noch als ungetracktes Verzeichnis:

```

$ git --version
git version 2.12.2

$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    CryptoLibrary/

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

```

Das Entfernen des Verzeichnisses ist nicht schwierig, aber es kann etwas verwirrend sein, was darin enthalten ist. Wenn du es entfernst und dann wieder zum Branch wechselst, der dieses Submodul besitzt, musst du **submodule update --init** ausführen, um es neu zu befüllen.

```

$ 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

```

Auch das ist nicht wirklich schwierig. Aber auch das kann ein wenig verwirrend sein.

Neuere Git-Versionen (ab Git 2.13) vereinfachen das alles, indem sie das Flag `--recurse-submodules` zum Befehl `git checkout` hinzufügen, der sich darum kümmert, die Submodule in den richtigen Zustand für den Branch zu bringen, auf den wir wechseln.

```
$ git --version
git version 2.13.3

$ git checkout -b add-crypto
Switched to a new branch 'add-crypto'

$ git submodule add https://github.com/chaconinc/CryptoLibrary
Cloning into 'CryptoLibrary'...
...

$ git commit -am 'Add crypto library'
[add-crypto 4445836] Add crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout --recurse-submodules master
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

nothing to commit, working tree clean
```

Die Verwendung des Flags `--recurse-submodules` mit `git checkout` kann auch praktisch sein, wenn du auf mehreren Branches im Hauptprojekt arbeitest, wobei jedes deiner Submodule auf unterschiedliche Commits zeigt. Wenn du zwischen den Branches wechselst, die das Submodul bei verschiedenen Commits enthalten, wird das Submodul bei der Ausführung von `git status` als „modifiziert“ erscheinen und „neue Commits“ anzeigen. Das liegt daran, dass der Submodul-Status beim Wechseln der Branches standardmäßig nicht mit übertragen wird.

Das kann äußerst verwirrend sein, deshalb ist es besser, immer den Befehl `git checkout --recurse-submodules` zu verwenden, wenn dein Projekt Submodule hat. Für ältere Git-Versionen, die das Flag `--recurse-submodules` nicht kennen, verwende nach dem Auschecken `git submodule update --init --recursive`, um die Submodule in den richtigen Zustand zu versetzen.

Glücklicherweise kannst du Git ( $\geq 2.14$ ) anweisen, immer das Flag `--recurse-submodules` zu verwenden, indem du die Konfigurationsoption `submodule.recurse` setzt mit: `git config submodule.recurse true`. Wie oben schon erwähnt, wird das auch dazu führen, dass Git in Submodulen jeden Befehl entsprechend umwandelt, der eine `--recurse-submodules` Option hat (außer bei `git clone`).

## Von Unterverzeichnissen zu Submodulen wechseln

Die andere Falle, in die viele Anwender tappen, ist der Wechsel von Unterverzeichnissen zu Submodulen. Wenn du Dateien in deinem Projekt getrackt hast und sie in ein Submodul verschieben willst, musst du vorsichtig sein. Andernfalls wird Git dir das nicht verzeihen. Angenommen, du hast Dateien in einem Unterverzeichnis deines Projekts und willst sie in ein Untermodul verschieben. Wenn du das Unterverzeichnis löschen und dann `submodule add` ausführst, wird Git dich in etwa so ausschimpfen:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

Du musst zuerst das `CryptoLibrary` Verzeichnis aus der Staging-Area unstage. Danach kannst du das Submodul hinzufügen:

```
$ 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.
```

Nehmen wir jetzt an, du hast das in einem Branch getan. Wenn du versuchst, wieder zu einem Branch zu wechseln, in der sich diese Dateien noch im aktuellen Verzeichnisbaum und nicht in einem Submodul befinden, erhältst du diesen Fehler:

```
$ 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
```

Du kannst den Wechsel mit `checkout -f` erzwingen, aber achte darauf, dass dort keine ungesicherten Änderungen enthalten sind, da diese mit dem Befehl überschrieben werden könnten.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Wenn du zurückwechselst, erhältst du aus bestimmten Gründen ein leeres `CryptoLibrary`

Verzeichnis. Auch ein `git submodule update` kann es dann nicht beheben. Möglicherweise musst du in dein Submodul-Verzeichnis gehen und `git checkout .` ausführen, um alle deine Dateien zurück zu bekommen. Du kannst das mit einem `submodule foreach` Skript erledigen, um es für mehrere Submodule anzuwenden.

Es ist wichtig zu wissen, dass Submodule heutzutage alle ihre Git-Daten im `.git` Verzeichnis des Hauptprojekts speichern. So gehen im Gegensatz zu vielen älteren Git-Versionen, mit dem Löschen eines Submodul-Verzeichnisses keine Commits oder Branches verloren, die du zuvor schon hattest.

Mit diesen Werkzeugen können Submodule eine ziemlich einfache und effektive Methode sein, um an mehreren verwandten, aber dennoch unterschiedlichen Projekten gleichzeitig zu arbeiten.

## Bundling

Wir haben zwar die üblichen Methoden zur Übertragung von Git-Daten über ein Netzwerk (HTTP, SSH usw.) behandelt, aber es gibt noch eine weitere Möglichkeit. Diese wird zwar nicht häufig verwendet, aber sie kann durchaus nützlich sein.

Git ist in der Lage, seine Daten in einer einzigen Datei zu „bündeln“. Das kann in verschiedenen Situationen nützlich sein. Vielleicht ist dein Netzwerk ausgefallen und du möchtest Änderungen an deine Mitstreiter senden. Vielleicht arbeitest du irgendwo außerhalb deines Unternehmens und hast aus Sicherheitsgründen keinen Zugang zum Firmen-Netzwerk. Möglicherweise ist deine Wireless-/Ethernet-Karte einfach kaputt. Oder du hast im Moment keinen Zugang zu einem gemeinsamen Server. Du willst jemandem Updates per E-Mail schicken und keine 40 Commits per `format-patch` übertragen.

Hier kann die Funktion `git bundle` behilflich sein. Der Befehl `bundle` packt alles, was normalerweise mit einem `git push` Befehl über die Leitung geschoben wird, in eine Binärdatei, die du an jemanden per E-Mail oder auf einem Flash-Laufwerk schicken kannst, um es dann in ein anderes Repository zu entpacken.

Lass uns ein einfaches Beispiel anschauen. Angenommen, du hast ein Repository mit zwei Commits:

```
$ 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

Wenn du dieses Repository an jemanden schicken willst und du keinen Zugriff auf ein Repository

hast, um es zu pushen, oder wenn du einfach keins einrichten willst, kannst du es mit `git bundle create` bündeln.

```
$ 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)
```

Du hast jetzt eine Datei mit der Bezeichnung `repo.bundle`, die alle Daten enthält, die benötigt werden, um den `master` Branch des Repositorys neu zu erstellen. Mit dem Kommando `bundle` musst du jede Referenz oder jeden spezifischen Bereich von Commits auflisten, die du einbeziehen möchtest. Wenn du beabsichtigst, diese Datei irgendwo anders zu klonen, solltest du HEAD als Referenz hinzufügen, wie wir es hier getan haben.

Du kannst diese `repo.bundle` Datei per E-Mail an eine andere Person schicken oder sie auf einem USB-Laufwerk speichern und übergeben.

Nun nehmen wir an, dass du diese `repo.bundle` Datei erhalten hast und an dem Projekt mitarbeiten willst. Du kannst die Binärdatei in ein Verzeichnis klonen, ähnlich wie du es von einer URL aus tun würdest.

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 Second commit
b1ec324 First commit
```

Wenn du HEAD nicht in die Referenzen integrierst, musst du `-b master` oder einen beliebigen anderen Branch angeben, da der Befehl sonst nicht weiß, welchen Branch er auschecken soll.

Nehmen wir an, du machst drei Commits darauf und willst die neuen Commits über ein Bündel auf einem USB-Stick oder per E-Mail zurückschicken.

```
$ git log --oneline
71b84da Last commit - second repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
9a466c5 Second commit
b1ec324 First commit
```

Zuerst müssen wir die Commits bestimmen, die wir in das Bündel aufnehmen wollen. Im Gegensatz zu den Netzwerkprotokollen, die für uns den minimalen Datensatz für die Übertragung über das Netzwerk festlegen, müssen wir das hier manuell herausfinden. Du könntest einfach das gesamte

Repository bündeln. Das wird zwar funktionieren, aber es ist besser, nur die Differenz zu bündeln – einfach nur die drei Commits, die wir gerade lokal gemacht hatten.

Dazu musst du die Differenz berechnen. Wie wir in [Commit-Bereiche](#) beschrieben haben, kannst du die Commits auf verschiedene Weise festlegen. Um die drei Commits zu bestimmen, die wir in unserem `master` Branch vorliegen haben und die nicht in dem Branch waren als wir ihn geklont haben, könnten wir zum Beispiel `origin/master..master` oder `master ^origin/master` benutzen. Du kannst die Ausführung mit dem Befehl `log` überprüfen und testen.

```
$ git log --oneline master ^origin/master
71b84da Last commit - second repo
c99cf5b Fourth commit - second repo
7011d3d Third commit - second repo
```

Jetzt haben wir unsere Liste der Commits, die wir in das Bundle aufnehmen wollen. Wir machen das mit dem Befehl `git bundle create`, indem wir ihm einen Dateinamen angeben (wie das Bündel heißen soll) und den Umfang der Commits angeben, die wir aufnehmen wollen.

```
$ 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)
```

Jetzt haben wir eine `commits.bundle` Datei in unserem Verzeichnis. Wenn wir diese Datei an unsere Partnerin schicken, kann sie diese in das originale Repository importieren, auch wenn dort zwischenzeitlich weitere Arbeiten stattgefunden haben.

Wenn sie das Bündel erhält, kann sie den Inhalt prüfen, bevor sie es in ihr Repository importiert. Der erste Befehl ist der Befehl `bundle verify`, der sicherstellt, dass die Datei tatsächlich ein gültiges Git-Bundle ist und dass diese alle notwendigen Vorgänger hat, um sie korrekt wiederherzustellen.

```
$ 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
```

Hätte der Bündel-Ersteller nur die beiden letzten Commits gebündelt und nicht alle drei, wäre das ursprüngliche Repository nicht in der Lage, es zu importieren, da ihm der erforderliche Verlauf fehlt. Das Kommando `verify` hätte stattdessen so ausgesehen:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
```

```
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 Third commit - second repo
```

Unser erstes Bündel ist jedoch gültig, so dass wir daraus die Commits abrufen können (engl. `fetch`). Wenn du sehen möchtest, welche Branches aus dem Bündel importiert werden können, gibt es auch einen Befehl, um nur die HEADS aufzulisten:

```
$ git bundle list-heads ../commits.bundle  
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

Der Unterbefehl `verify` wird dir auch die HEADS anzeigen. Es geht darum, zu sehen, was man übernehmen kann. Du kannst dann die Befehle `fetch` oder `pull` verwenden, um Commits aus diesem Bündel zu importieren. Hier holen wir den `master` Branch aus dem Bündel in einen Branch mit dem Namen `other-master` in unser Repository:

```
$ git fetch ../commits.bundle master:other-master  
From ../commits.bundle  
* [new branch]      master      -> other-master
```

Jetzt können wir sehen, dass wir die importierten Commits auf dem Branch `other-master` haben, sowie alle Commits, die wir in der Zwischenzeit in unserem eigenen `master` Branch gemacht haben.

```
$ 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
```

Der Befehl `git-bundle` kann also sehr nützlich sein, um Arbeit weiterzugeben oder um netzwerkähnliche Operationen durchzuführen, falls du nicht über ein gemeinsames Netzwerk oder ein gemeinsam genutztes Repository verfügst.

## Replace (Ersetzen)

Wie wir bereits betont haben, sind die Objekte in der Objektdatenbank von Git unveränderbar. Git bietet aber eine interessante Möglichkeit, so *zu tun, als ob* man Objekte in der Datenbank durch andere Objekte ersetzen würde.

Der Befehl `replace` ermöglicht es dir, ein Objekt in Git zu bestimmen und zu sagen „jedes Mal, wenn ich auf *dieses* Objekt verweise, behandle es so, als wäre es ein *anderes* Objekt“. Das wird am häufigsten zum Ersetzen eines Commits in deinem Verlauf durch einen anderen genutzt. Dadurch musst du nicht die gesamte Historie neu aufbauen, wie z.B. mit `git filter-branch`.

Nehmen wir zum Beispiel an, du hast einen riesigen Code-Verlauf und möchtest dein Repository aufsplitten in eins mit kurzen Verlauf für neue Entwickler und eins mit viel längeren und ausführlicheren Verlauf für Leute, die sich für Data Mining interessieren. Du kannst eine Historie in eine andere einpflanzen, indem du den frühesten Commit in der neuen Zeile durch den neuesten Commit in der älteren Zeile „ersetzt“. Das ist angenehm, weil es bedeutet, dass du nicht wirklich jeden Commit in der neuen Historie neu erstellen musst, wie du es normalerweise tun müsstest, um sie zusammenzufügen (weil die Elternabstammung die SHA-1-Werte beeinflusst).

Probieren wir das einmal aus. Nehmen wir ein vorhandenes Repository, teilen es in zwei Repositorys auf, ein aktuelles und ein altes. Dann untersuchen wir, wie wir sie rekombinieren können, ohne die aktuellen SHA-1-Werte des Repositorys durch `replace` zu verändern.

Wir werden ein kleines Repository mit fünf einfachen Commits verwenden:

```
$ git log --oneline  
ef989d8 Fifth commit  
c6e1e95 Fourth commit  
9c68fdc Third commit  
945704c Second commit  
c1822cf First commit
```

Wir wollen dieses in zwei unterschiedliche Historien aufteilen. Eine Linie geht von Commit eins bis Commit vier – das wird die historische Linie sein. Die zweite Linie wird nur aus den Commits vier und fünf bestehen – das wird die jüngere Historie sein.

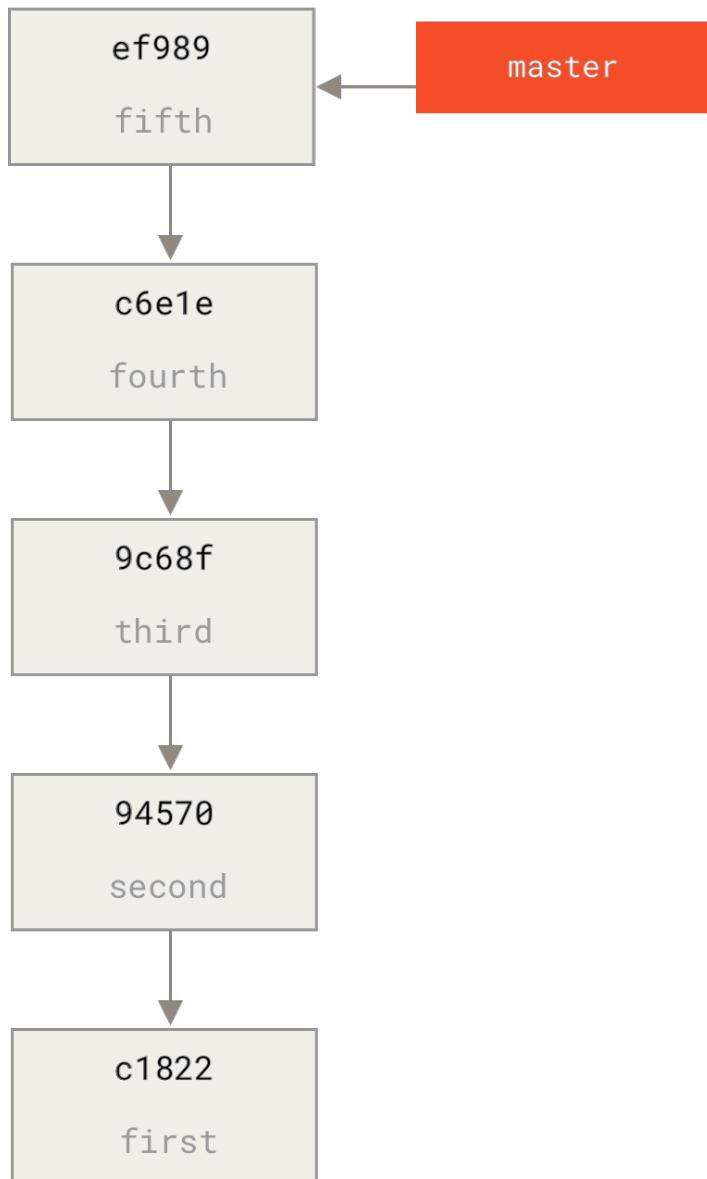


Figure 163. Beispiel Git Historie

Die Erstellung des historischen Verlaufs ist einfach. Wir können einen Branch in den Verlauf einfügen und dann diesen Branch auf den `master` Branch eines neuen Remote-Repositorys pushen.

```

$ 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

```

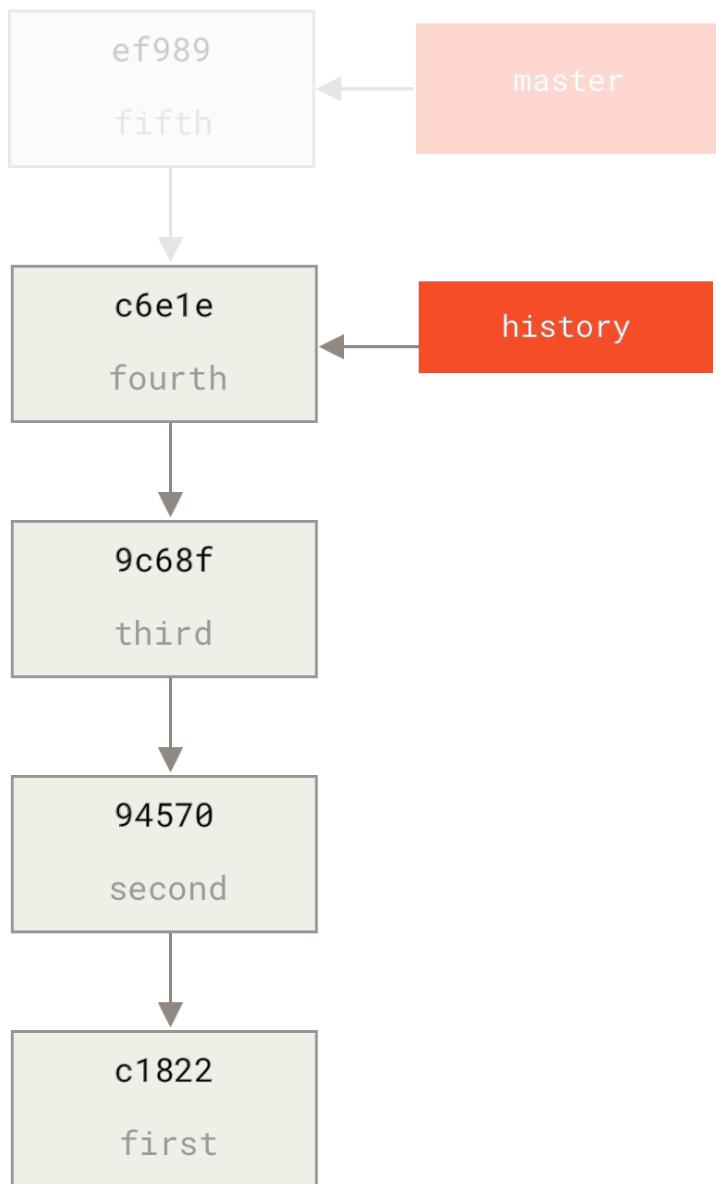


Figure 164. Erstellen eines neuen `history` Branches

Jetzt können wir den neuen Branch `history` in den `master` Branch unseres neuen Repositorys 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

```

Damit ist unsere Historie veröffentlicht. Nun ist der schwierigere Teil, unsere jüngere Historie nach hinten zu kürzen, damit sie kleiner wird. Wir brauchen eine Überlappung, damit wir einen Commit in der einen durch einen gleichwertigen Commit in der anderen ersetzen können. Deshalb werden wir diesen Teil auf die Commits vier und fünf kürzen (so dass sich Commit vier überlappt).

```
$ git log --oneline --decorate  
ef989d8 (HEAD, master) Fifth commit  
c6e1e95 (history) Fourth commit  
9c68fdc Third commit  
945704c Second commit  
c1822cf First commit
```

In diesem Fall ist es nützlich, einen Basis-Commit zu erstellen, der Anweisungen zum Erweitern der Historie enthält, damit andere Entwickler wissen, was zu tun ist, wenn sie auf den ersten Commit in der getrennten Historie treffen und mehr brauchen. Was wir also vornehmen werden, ist ein erstes Commit-Objekt als unseren Basispunkt mit Anweisungen zu erstellen und dann die restlichen Commits (vier und fünf) darauf zu rebasen.

Dazu müssen wir einen Punkt wählen, an dem wir abspalten möchten, der in unserem Beispiel der dritte Commit ist. Er lautet **9c68fdc** im SHA-Vokabular. Unser Basis-Commit wird also auf diesem Baum basieren. Wir können unseren Basis-Commit mit dem Befehl **commit-tree** erstellen, der einfach einen Baum nimmt und uns ein brandneues, elternloses SHA-1-Commit-Objekt zurückgibt.

```
$ echo 'Get history from blah blah blah' | git commit-tree 9c68fdc^{tree}  
622e88e9cbfbacfb75b5279245b9fb38dfa10cf
```

 Das Kommando **commit-tree** gehört zu einer Reihe von Befehlen, die allgemein als „Basis“-Befehle (engl. 'plumbing' commands) bezeichnet werden. Diese Befehle sind im Allgemeinen nicht für den direkten Einsatz gedacht, sondern werden, eingebettet in **andere** Git-Befehle, um kleinere Aufgaben verwendet. Wenn wir bei derartigen Gelegenheiten etwas Ungewöhnliches durchführen müssen, dann erlauben sie uns, echte low-level Aufgaben zu erledigen, sind aber nicht für den täglichen Gebrauch gedacht. Du kannst mehr über Basisbefehle in [Basisbefehle und Standardbefehle \(Plumbing and Porcelain\)](#) nachlesen.



Figure 165. Erstellen eines Basis-Commits mit `commit-tree`

Jetzt, wo wir einen Basis-Commit haben, können wir den Rest unseres Verlaufs mit `git rebase --onto` darauf rebasen. Das Argument `--onto` ist der SHA-1, den wir gerade von `commit-tree` zurückbekommen haben. Der Rebase-Punkt wird der dritte Commit sein (das Elternteil des ersten Commits, den wir behalten wollen, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```

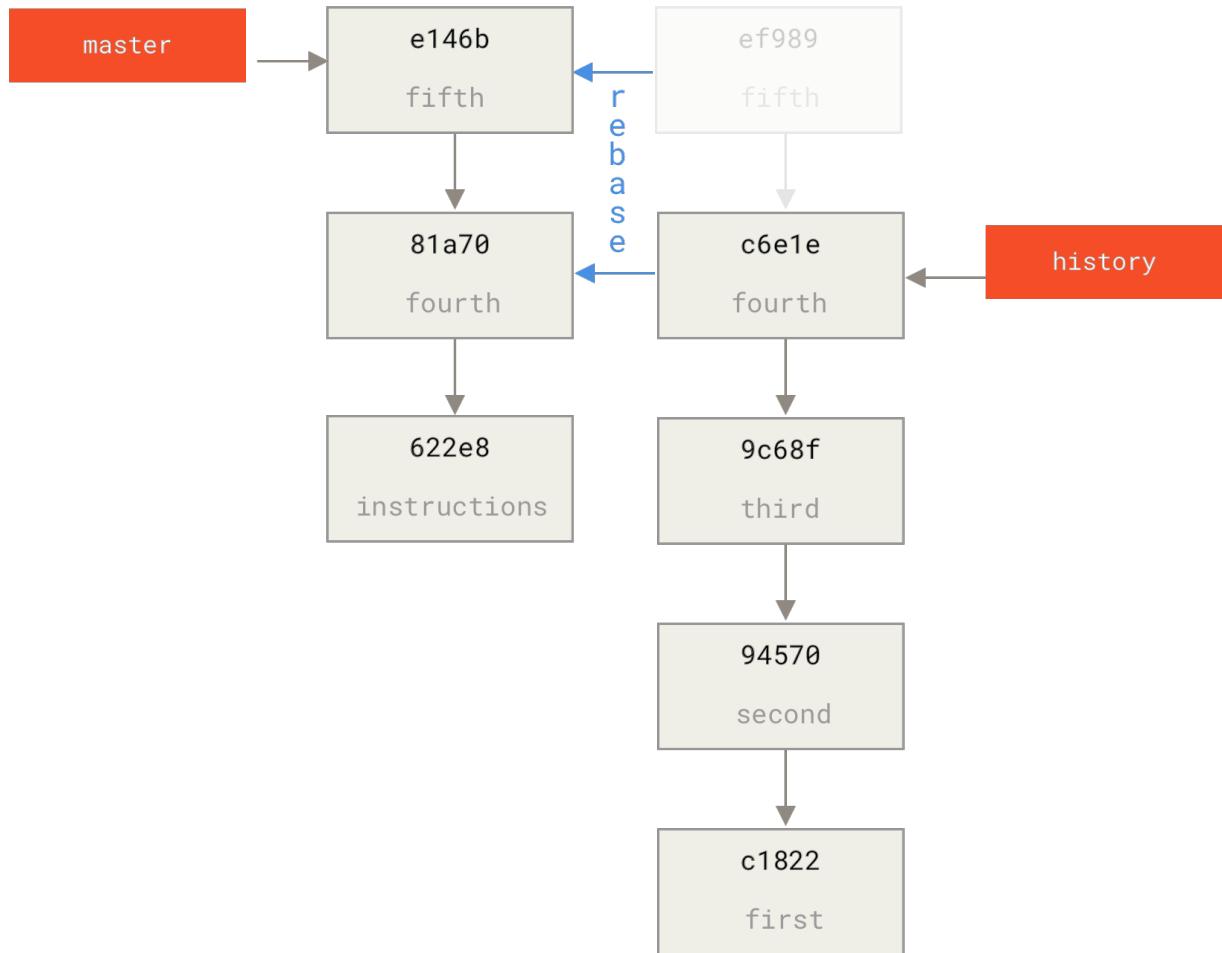


Figure 166. Rebasen der Historie auf den Basis-Commit

Nun haben wir also unseren jüngsten Verlauf auf einer Übergabebasis neu geschrieben, die jetzt Anweisungen enthält, wie wir die gesamte Historie rekonstruieren könnten, wenn wir es wollen. Wir können diesen neuen Verlauf auf ein neues Projekt übertragen. Wenn die Anwender jetzt dieses Repository klonen, sehen sie nur die beiden letzten Commits und dann einen Basis-Commit mit Anweisungen.

Lass uns nun die Rolle tauschen mit jemandem, der das Projekt zum ersten Mal klonen und den gesamten Verlauf des Projekts haben will. Um die Verlaufsdaten nach dem Klonen dieses abgetrennten Repositorys zu erhalten, müsste man einen zweiten Remote für das historische Repository hinzufügen und 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
```

Nun würde die andere Person deine jüngsten Commits im `master` Branch und die historischen Commits im `project-history/master` Branch erhalten.

```
$ 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
```

Um sie zu vereinen, kannst du einfach `git replace` mit dem Commit, den du ersetzen willst, und dann den Commit, mit dem du ihn ersetzen willst, aufrufen. Wir wollen also den „vierten“ Commit im `master` Branch durch den "vierten" Commit im `project-history/master` Branch ersetzen:

```
$ git replace 81a708d c6e1e95
```

Wenn man sich nun den Verlauf des `master` Branch anschaut, sieht er so aus:

```
$ git log --oneline master
e146b5f Fifth commit
81a708d Fourth commit
9c68fdc Third commit
945704c Second commit
c1822cf First commit
```

Klasse, oder? Ohne alle SHA-1s im Upstream ändern zu müssen, konnten wir einen Commit in unserer Historie durch einen ganz anderen ersetzen, und alle normalen Werkzeuge (`bisect`, `blame`, usw.) werden so funktionieren, wie wir es erwarten.

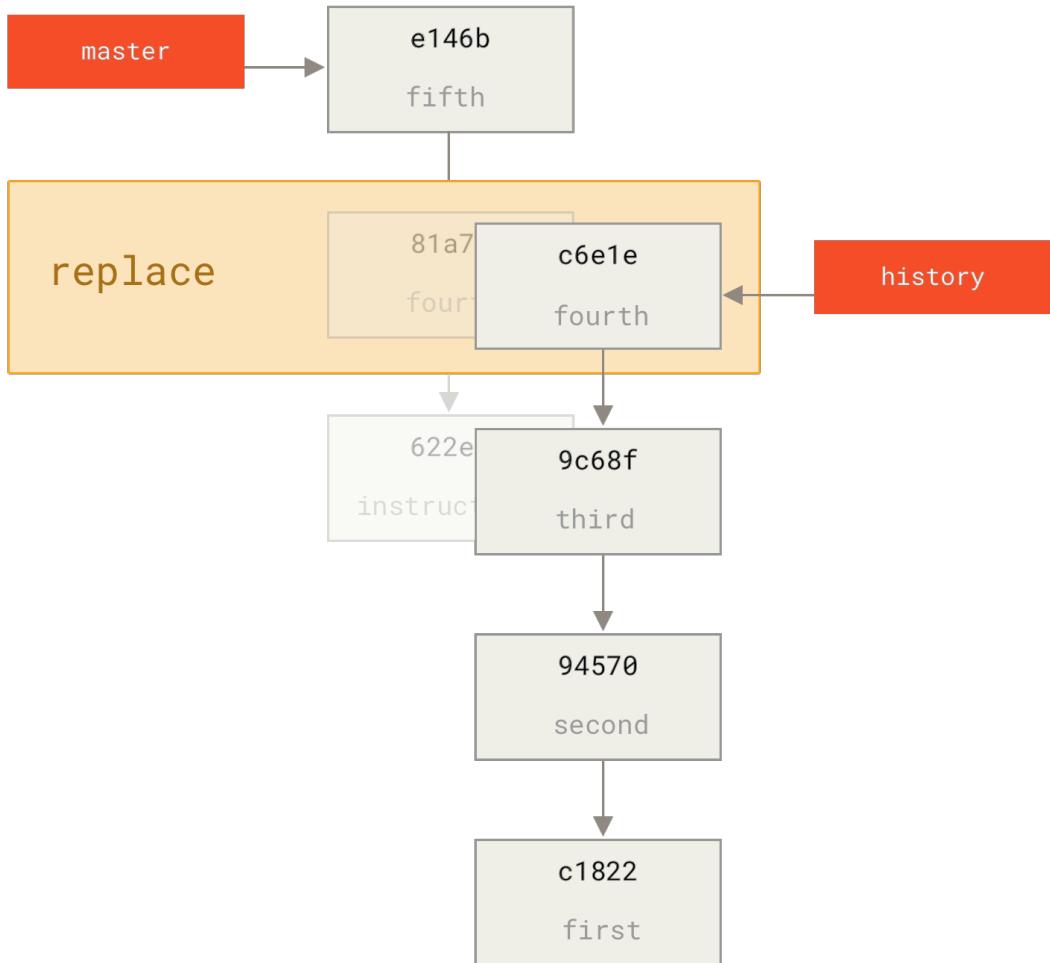


Figure 167. Kombinieren der Commits mit `git replace`

Interessanterweise zeigt das Protokoll ('log') immer noch `81a708d` als SHA-1 an, obwohl es tatsächlich die `c6e1e95`-Commit-Daten verwendet, durch die wir es ersetzt haben. Selbst wenn du einen Befehl wie `cat-file` ausführst, zeigt er dir die ersetzen Daten an:

```
$ 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
```

Vergiss nicht, dass das eigentliche Elternteil von `81a708d` unser Platzhalter-Commit (`622e88e`) war, nicht `9c68fdce`, wie es hier steht.

Interessant ist, dass diese Daten in unseren Referenzen gespeichert sind:

```
$ 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
```

Das bedeutet, dass es einfach ist, unseren Ersatz mit anderen zu teilen, weil wir diesen auf unseren Server pushen können und andere Anwender ihn leicht herunterladen können. Das ist in dem Szenario zur Verlaufsoptimierung, das wir hier durchgespielt haben, nicht so hilfreich (da jeder sowieso beide Historien herunterladen würde. Warum also beide trennen?). Es kann aber unter anderen Umständen sinnvoll sein.

## Anmeldeinformationen speichern

Wenn du das SSH-Protokoll für die Verbindung zu Remotes verwendest, kannst du einen Schlüssel ohne Passwort verwenden. Damit kannst du Daten sicher übertragen, ohne deinen Benutzernamen und ein Passwort einzugeben. Mit den HTTP-Protokollen ist das aber nicht möglich. Jede Verbindung benötigt einen Benutzernamen und ein Passwort. Noch schwieriger wird das bei Systemen mit Zwei-Faktor-Authentifizierung, bei denen das Token, das du für ein Kennwortwendest, zufällig generiert wird und nicht aussprechbar ist.

Glücklicherweise hat Git ein Anmeldesystem, das hier weiterhelfen kann. Git hat dazu ein paar Optionen im Angebot:

- Standardmäßig wird überhaupt nicht zwischengespeichert. Bei jeder Verbindung wird nach deinem Benutzernamen und Passwort gefragt.
- Der „Cache“-Modus hält die Anmeldedaten für eine gewisse Zeitspanne im Zwischenspeicher. Keines der Passwörter wird jemals auf der Festplatte abgelegt und nach 15 Minuten werden sie aus dem Cache gelöscht.
- Der „Speicher“-Modus speichert die Zugangsdaten in einer Klartextdatei auf der Festplatte und sie verfallen nie. Das bedeutet, dass du deine Anmeldedaten nie wieder eingeben musst, bis du dein Passwort für den Git-Host änderst. Der Nachteil dieses Ansatzes ist, dass deine Passwörter im Klartext in einer einfachen Datei in deinem Homeverzeichnis gespeichert werden.
- Wenn du einen Mac verwendest, verfügt Git über einen „osxkeychain“-Modus, der die Anmeldeinformationen im sicheren Schlüsselbund, der an deinem Systemkonto angehängt ist, zwischenspeichert. Diese Methode speichert die Zugangsdaten, ohne Laufzeitbegrenzung auf der Festplatte. Sie werden mit dem gleichen Verfahren verschlüsselt, das auch HTTPS-Zertifikate und Safari-Auto-Vervollständigungen verwaltet.
- Wenn du Windows verwendest, kannst du das Feature **Git Credential Manager** aktivieren. Dazu musst du [Git für Windows](#) installieren oder [the latest GCM](#) als eigenständiger Dienst installieren. Dies ähnelt dem oben beschriebenen ``osxkeychain``-Hilfsprogramm. Es verwendet jedoch den Windows-Anmeldeinformationsspeicher, um vertrauliche Informationen zu verwalten. Es kann auch Anmeldeinformationen für WSL1 oder WSL2 bereitstellen. Weitere Informationen findest du unter [GCM-Installationsanweisungen](#).

Du kannst eine dieser Methoden durch Setzen eines Git-Konfigurationswertes wählen:

```
$ git config --global credential.helper cache
```

Einige dieser Hilfsmittel haben Optionen. Der „store“-Assistent kann das Argument `--file <path>` benutzen, das den Speicherort der Klartextdatei anpasst (der Standard ist `~/.git-credentials`). Der „cache“-Assistent akzeptiert die Option `--timeout <seconds>`, welche die Zeitspanne ändert, in der der Dämon läuft (die Vorgabe ist „900“, oder 15 Minuten). Hier folgt ein Beispiel, wie du den „store“-Assistent mit einem benutzerdefinierten Dateinamen konfigurieren kannst:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

Mit Git kannst du auch mehrere Assistenten konfigurieren. Wenn du nach Anmeldeinformationen für einen bestimmten Host suchst, fragt Git diese der Reihe nach ab und stoppt nach der ersten, erhaltenen Antwort. Beim Speichern der Zugangsdaten sendet Git den Benutzernamen und das Passwort an **alle** Assistenten der Liste, diese entscheiden, was damit zu tun ist. So würde eine `.gitconfig` aussehen, wenn du eine Datei mit Zugangsdaten auf einem USB-Stick hättest. Damit kannst du den Zwischenspeicher nutzen, um dir die Eingabe zu sparen, wenn das Laufwerk nicht angeschlossen ist:

```
[credential]
  helper = store --file /mnt/thumbdrive/.git-credentials
  helper = cache --timeout 30000
```

## Unter der (Motor-)Haube

Wie funktioniert das alles? Der Hauptbefehl für das Anmelde-System ist `git credential`, der einen Befehl als Argument und dann weitere Eingaben über `stdin` (Standardeingabe=Tastatur) entgegennimmt.

Mit einem Beispiel ist das vielleicht leichter verständlich. Nehmen wir an, dass ein Assistent für die Anmeldung konfiguriert wurde und er Anmelddaten für `mygithost` gespeichert hat. Die folgende Sitzung verwendet den Befehl „fill“, der aufgerufen wird, wenn Git versucht, Zugangsdaten für einen Host zu finden:

```
$ 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 Diese Befehlszeile leitet die eigentliche Interaktion ein.
- ② Das Anmeldesystem wartet auf die Eingabe von stdin. Wir geben das „Protokoll“ und den „Hostnamen“ ein.
- ③ Eine leere Zeile signalisiert, dass die Eingabe vollständig ist und das Anmeldesystem sollte mit dem antworten, was ihm bekannt ist.
- ④ Dann übernimmt das Programm Git-Credential und gibt auf stdout die gefundenen Informationen aus.
- ⑤ Falls keine Anmeldeinformationen gefunden werden, fragt Git den Benutzer nach dem Benutzernamen und dem Kennwort und gibt sie an den aufrufenden stdout zurück (hier sind sie an dieselbe Konsole angeschlossen).

Das Anmeldesystem ruft in Wirklichkeit ein Programm auf, das von Git selbst unabhängig ist und bei dem der Konfigurationswert `credential.helper` bestimmt, was und auf welche Weise es aufgerufen wird. Es kann unterschiedliche Varianten anbieten:

Konfiguration-Wert	Verhalten
<code>foo</code>	führt <code>git-credential-foo</code> aus
<code>foo -a --opt=bcd</code>	führt <code>git-credential-foo -a --opt=bcd</code> aus
<code>/absolute/path/foo -xyz</code>	führt <code>/absolute/path/foo -xyz</code> aus
<code>!f() { echo "password=s3cre7"; }; f</code>	Code nach <code>!</code> wird in der Shell ausgewertet

Die oben beschriebenen Hilfsprogramme heißen also eigentlich `git-credential-cache`, `git-credential-store` usw.. Wir können sie so konfigurieren, dass sie Befehlszeilenargumente übernehmen. Die allgemeine Form dafür ist „`git-credential-foo [Argument] <Action>`“. Das stdin/stdout-Protokoll ist das Gleiche wie `git-credential`, aber sie verwenden einen etwas anderen Befehls-Satz:

- `get` ist die Abfrage nach einem Benutzernamen/Passwort-Paar.
- `store` ist die Aufforderung, einen Satz von Anmeldeinformationen im Speicher dieses Assistenten zu hinterlegen.
- `erase` löscht die Anmeldeinformationen für die angegebenen Einstellungen aus dem Speicher dieses Assistenten.

Für die `store` und `erase` Aktionen ist keine Reaktion erforderlich (Git ignoriert sie ohnehin). Für die Aktion `get` ist Git allerdings stark daran interessiert, was der Assistent sagt. Wenn das Hilfsprogramm nichts Sinnvolles kennt, kann es einfach ohne Ausgabe abbrechen. Weiß es aber etwas, sollte es die bereitgestellten Informationen mit den gespeicherten Informationen ergänzen. Die Ausgabe wird wie eine Reihe von Assignment-Statements behandelt. Alles, was zur Verfügung

gestellt wird, ersetzt das, was Git bereits kennt.

Hier ist das gleiche Beispiel von oben, aber ohne `git-credential` und direkt mit `git-credential-store`:

```
$ git credential-store --file ~/git.store store ①
protocol=https
host=mygithost
username=bob
password=s3cre7
$ git credential-store --file ~/git.store get ②
protocol=https
host=mygithost

username=bob ③
password=s3cre7
```

- ① Hier weisen wir `git-credential-store` an, einige Anmelddaten zu speichern: der Benutzername „bob“ und das Passwort „s3cre7“ sollen verwendet werden, wenn auf `https://mygithost` zugegriffen wird.
- ② Jetzt rufen wir diese Anmelddaten ab. Wir geben die bereits bekannten Teile der Internetadresse (`https://mygithost`) und eine leere Zeile ein.
- ③ `git-credential-store` antwortet mit dem Benutzernamen und dem Passwort, die wir beide oben gespeichert haben.

So sieht die Datei `~/git.store` jetzt aus:

```
https://bob:s3cre7@mygithost
```

Sie besteht nur aus einer Reihe von Zeilen, von denen jede eine mit einem Anmeldeinformationen versehene URL enthält. Die Assistenten `osxkeychain` und `wincred` verwenden das native Format ihrer Zwischenspeicher, während `cache` ein eigenes Speicherformat verwendet (das kein anderer Prozess lesen kann).

## Benutzerdefinierter Cache für Anmeldeinformationen.

Angenommen, dass `git-credential-store` und seine Freunde von Git getrennte Programme sind. Dann ist es nicht schwierig zu erkennen, dass *jedes* Programm ein Hilfsprogramm für die Git-Anmeldung sein kann. Die von Git bereitgestellten Assistenten decken viele gewöhnliche Anwendungsfälle ab, aber nicht alle. Nehmen wir zum Beispiel an, dein Team hat einige Anmelddaten, die dem gesamten Team zur Verfügung gestellt werden sollen, z.B. für die Entwicklung. Diese werden in einem freigegebenen Verzeichnis gespeichert. Du möchtest sie jedoch nicht in dein eigenes Anmeldesystem kopieren, da sie sich häufig ändern. Keines der vorhandenen Hilfsprogramme ist auf diesen Fall anwendbar. Schauen wir mal, was nötig wäre, um unser Eigenes zu schreiben. Dieses Programm muss mehrere Schlüsselfunktionen haben:

1. Die einzige Aktion, auf die wir achten müssen, ist `get`. Die Aktionen `store` und `erase` sind

Schreiboperationen, also werden wir sie einfach sauber beenden, sobald sie auftauchen.

2. Das Dateiformat der Datei für die gemeinsamen Anmelddaten ist das Gleiche wie es von `git-credential-store` verwendet wird.
3. Der Speicherort dieser Datei ist eigentlich standardisiert, aber wir sollten es dem Benutzer erlauben, einen benutzerdefinierten Pfad anzugeben, nur für den Fall, dass er es wünscht.

Wir werden diese Erweiterung wieder in Ruby schreiben, aber jede Programmiersprache wird funktionieren, solange Git das fertige Produkt ausführen kann. Hier ist der vollständige Quellcode unseres neuen Anmeldehelpers:

```
#!/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 wir die Befehlszeilenoptionen und erlauben dem Benutzer, die Eingabedatei zu spezifizieren. Die Vorgabe ist `~/.git-credentials`.

② Dieses Programm antwortet nur, wenn die Aktion `get` lautet und die Backup-Speicherdatei existiert.

- ③ Die Schleife hier, liest von stdin, bis die erste leere Zeile erkannt wird. Die Eingaben werden im Hash `known` zur späteren Referenz gespeichert.
- ④ Diese Schleife liest den Inhalt der Speicherdatei und sucht nach Übereinstimmungen. Wenn die Protokolldaten, der Host und der Benutzername mit `known` in dieser Zeile übereinstimmen, gibt das Programm die Ergebnisse auf stdout aus und beendet sich.

Wir speichern unser Hilfsprogramm als `git-credential-read-only`, legen es irgendwo in unserem `PATH` ab und markieren es als ausführbar. So sieht dann eine interaktive Sitzung aus:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost
username=bob

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Da der Name mit „git-“ beginnt, können wir die einfache Syntax für den Konfigurationswert verwenden:

```
$ git config --global credential.helper 'read-only --file /mnt/shared/creds'
```

Wie du siehst, ist die Erweiterung dieses Systems ziemlich unkompliziert und kann einige typische Probleme für dich und dein Team lösen.

## Zusammenfassung

Du hast eine Reihe von fortschrittlichen Tools kennengelernt, mit denen du deine Commits und die Staging-Area präzise verändern kannst. Wenn du Probleme hast, solltest du in der Lage sein leicht herauszufinden, was, wann und von wem der Commit eingebracht wurde. Wenn du Subprojekte in deinem Projekt verwenden möchtest, hast du gelernt, wie du damit umgehst. An diesem Punkt solltest du auf der Kommandozeile die meisten der täglichen Aufgaben in Git erledigen können und du solltest dich dabei sicher fühlen.

# Git einrichten

Bisher haben wir die grundlegende Funktionsweise und die Verwendung von Git erläutert, sowie eine Reihe von Tools vorgestellt, die dir helfen sollen, Git einfach und effizient zu nutzen. In diesem Kapitel werden wir zeigen, wie du Git noch individueller einsetzen kannst, indem du einige wichtige Konfigurationen anpasst und das Hook-System anwendest. Mit diesen Tools ist es einfach, Git genau so einzurichten, wie du, dein Unternehmen oder deine Gruppe es benötigen.

## Git Konfiguration

Wie du in Kapitel 1, [Erste Schritte](#) bereits kurz gelesen hast, kannst du die Git-Konfigurationseinstellungen mit dem Befehl `git config` anpassen. Eine der ersten Dinge, die du vorgenommen hast, war die Einrichtung deines Namens und deiner E-Mail-Adresse:

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

Jetzt lernst du einige der interessanteren Optionen kennen, die du auf diese Weise festlegen kannst, um deine Git-Nutzung zu optimieren.

Zunächst eine kleine Übersicht: Git verwendet eine Reihe von Konfigurationsdateien, um das Standard-Verhalten, wie gewünscht, zu verändern. Der erste Ort, in der Git nach solchen Werten sucht, ist die globale Datei `[path]/etc/gitconfig`. Dort sind die Einstellungen enthalten, die auf jeden Benutzer auf dem System und alle seiner Repositorys angewendet werden. Wenn du die Option `--system` an `git config` über gibst, wird diese Datei gezielt ausgelesen und geschrieben.

Der nächste Ort, an dem Git nachschaut, ist die Datei `~/.gitconfig` (oder `~/.config/git/config`), die spezifisch für jeden Benutzer ist. Du kannst Git veranlassen, diese Datei zu lesen und zu schreiben, indem du die Option `--global` über gibst.

Schließlich sucht Git nach Informationen in der Konfigurations-Datei im Git-Verzeichnis (`.git/config`) des jeweiligen Repositorys, das du gerade verwendest. Diese Werte sind spezifisch für dieses spezielle Repository und werden bei Übergabe der Option `--local` an `git config` angewendet. Wenn du nichts angibst, welchen Level du ansprechen möchtest, ist das die Voreinstellung.

Jeder dieser „Level“ (system, global, lokal) überschreibt Werte der vorigen Ebene. Daher werden beispielsweise Werte in `.git/config` jene in `[path]/etc/gitconfig` überschreiben.



Die Konfigurationsdateien von Git sind reine Textdateien, so dass du diese Werte auch ändern kannst, wenn du die Datei manuell bearbeitest. Es ist jedoch generell einfacher und sicherer, den `git config` Befehl zu benutzen.

## Grundeinstellungen des Clients

Die von Git erkannten Einstelloptionen lassen sich in zwei Kategorien einteilen: client-seitig und serverseitig. Die meisten Optionen beziehen sich auf die Clientseite – die Konfiguration deiner persönlichen Arbeitseinstellungen. *Sehr sehr viele* Konfigurationsoptionen stehen zur Verfügung,

aber ein großer Teil davon ist nur in bestimmten Grenzfällen sinnvoll. Wir werden hier nur die gebräuchlichsten und nützlichsten Optionen behandeln. Wenn du eine Liste aller Optionen sehen möchtest, die deine Version von Git kennt, kannst du Folgendes aufrufen:

```
$ man git-config
```

Dieser Befehl listet alle verfügbaren Optionen detailliert auf. Du findest dieses Referenzmaterial auch unter <https://git-scm.com/docs/git-config>.



Für fortgeschrittene Anwendungsfälle kannst du in der oben erwähnten Dokumentation nach „Conditional includes“ suchen.

### core.editor

Um deine Commit- und Tag-Beschreibungen erstellen/bearbeiten zu können verwendet Git das von dir als Standard-Text-Editor eingestellte Programm aus einer der Shell-Umgebungs-Variablen **VISUAL** oder **EDITOR**. Alternativ greift Git auf den vi-Editor zurück. Diesen Standard kannst du ändern, indem du die Einstellung **core.editor** verwendest:

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

Jetzt wird Git Emacs starten, unabhängig davon, was als Standard-Shell-Editor eingestellt ist, um die Texte zu bearbeiten.

### commit.template

Wenn du dieses Attribut auf die Pfadangabe einer Datei auf deinem System setzt, verwendet Git diese Datei als initiale Standard-Nachricht, wenn du einen Commit durchführst. Der Vorteil bei der Erstellung einer benutzerdefinierten Commit-Vorlage besteht darin, dass du sie verwenden kannst, um sich (oder andere) beim Erstellen einer Commit-Nachricht an das richtige Format und den richtigen Stil zu erinnern.

Nehmen wir z.B. eine Template-Datei unter `~/.gitmessage.txt`, die so aussieht:

```
Subject line (try to keep under 50 characters)
```

```
Multi-line description of commit,  
feel free to be detailed.
```

```
[Ticket: X]
```

Diese Commit-Vorlage erinnert den Committer daran, die Betreffzeile kurz zu halten (für die `git log --oneline` Ausgabe), weitere Details hinzuzufügen und sich auf ein Problem oder eine Bug-Tracker-Ticketnummer zu beziehen, falls vorhanden.

Um Git anzuweisen, das als Standardnachricht zu verwenden, die in deinem Editor erscheint, wenn du `git commit` ausführst, setze den Konfigurationswert **commit.template**:

```
$ git config --global commit.template ~/.gitmessage.txt  
$ git commit
```

Dann öffnet sich dein Text-Editor in etwa so für deine Commit-Beschreibung, wenn du committest:

Subject line (try to keep under 50 characters)

Multi-line description of commit,  
feel free to be detailed.

[Ticket: X]

```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
# modified:   lib/test.rb
#
~  
~  
.git/COMMIT_EDITMSG" 14L, 297C
```

Mit einer eigenen Regel für Commit-Beschreibungen und dem Einfügen einer entsprechenden Vorlage in die Git-Konfiguration deines Systems erhöht sich die Wahrscheinlichkeit, dass diese Regel regelmäßig eingehalten wird.

### core.pager

Diese Einstellung bestimmt, welcher Pager genutzt werden soll, wenn git den Output von `log` und `diff` seitenweise ausgeben soll. Den Wert kann auf `more` oder wie von dir bevorzugt eingestellt werden (Standard ist `less`). Du kannst ihn deaktivieren, indem du eine leere Zeichenkette verwendest:

```
$ git config --global core.pager ''
```

Wenn du diese Konfiguration nutzt, wird Git die komplette Ausgabe aller Befehle anzeigen, unabhängig davon, wie lang sie sind.

### user.signingkey

Wenn du signierte und annotierte Tags erstellst (wie in Kapitel 7, [Ihre Arbeit signieren](#) beschrieben), erleichtert die Definition deines GPG-Signierschlüssels in der Konfigurations-Einstellung die Arbeit. Stelle deine Schlüssel-ID so ein:

```
$ git config --global user.signingkey <gpg-key-id>
```

Jetzt kannst du Tags signieren, ohne jedes Mal deinen Schlüssel mit dem Befehl `git tag` angeben zu müssen:

```
$ git tag -s <tag-name>
```

### core.excludesfile

Du kannst Suchmuster in die `.gitignore` Datei deines Projekts einfügen. Damit kannst du verhindern, dass Git bestimmte Dateien als nicht in der Versionsverwaltung erfasste Dateien erkennt und sie zum Commit vormerken will, wenn du `git add` darauf ausführst, wie in Kapitel 2, [Ignorieren von Dateien](#) beschrieben.

In manchen Fällen sollen bestimmte Dateien für alle Repositorys ignoriert werden, in denen du arbeitest. Falls du macOS verwendest, kennst du vermutlich die `.DS_Store` Dateien. Bei Emacs oder Vim kennst du vielleicht Dateien, die mit einer Tilde (~) oder auf `.swp` enden.

Mit dieser Einstellung kannst du eine Art globale `.gitignore` Datei schreiben. Erstelle eine `~/.gitignore_global` Datei mit diesem Inhalt:

```
*~  
.*.swp  
.DS_Store
```

... und dann führst du `git config --global core.excludesfile ~/.gitignore_global` aus. Git wird sich nie wieder um diese Dateien kümmern.

### help.autocorrect

Wenn du dich bei einem Befehl vertippst, zeigt das System dir so etwas wie das hier:

```
$ git chekcout master  
git: 'chekcout' is not a git command. See 'git --help'.
```

```
The most similar command is  
checkout
```

Git hilft dir dabei herauszufinden, was du gemeint hast, aber es führt den Befehl nicht aus. Wenn du `help.autocorrect` auf 1 setzt, dann führt Git den vorgeschlagenen Befehl auch aus:

```
$ 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...
```

Beachte die „0,1 Sekunden“ Angabe. `help.autocorrect` ist eigentlich eine Ganzzahl, die Zehntelsekunden repräsentiert. Wenn du den Wert auf 50 setzt, gibt dir Git 5 Sekunden Zeit deine

Meinung zu ändern, bevor der autokorrigierte Befehl ausgeführt wird.

## Farben in Git

Git unterstützt die farbige Terminalausgabe, was sehr nützlich ist, um die Befehlsausgabe schnell und einfach visuell zu verarbeiten. Eine Reihe von Optionen können dir helfen, die Farbgestaltung nach deinen Wünschen einzustellen.

### color.ui

Git färbt den größten Teil seiner Ausgabe automatisch ein, aber es gibt einen Hauptschalter, wenn dir dieses Verhalten nicht gefällt. Um alle farbigen Terminalausgaben von Git auszuschalten, nutze Folgendes:

```
$ git config --global color.ui false
```

Die Standardeinstellung ist `auto`, das die Ausgabe von Farben ausgibt, wenn es direkt zu einem Terminal geht, aber die Farbkontrollcodes weglässt, wenn die Ausgabe in eine Pipe oder eine Datei umgeleitet wird.

Du kannst es auch auf `always` einstellen, dass der Unterschied zwischen Terminals und Pipes ignoriert. Das willst du in der Regel nicht. In den meisten Szenarien kannst du stattdessen ein `--color` Flag an den Git-Befehl übergeben, um ihn zu zwingen, Farbcodes zu verwenden, wenn du Farbcodes in deine umgeleiteten Ausgabe wünschst. Die Voreinstellung ist fast immer die Richtige.

### color.\*

Wenn du genauer bestimmen möchtest, welche Befehle wie eingefärbt werden, dann bietet Git sehr spezifische Farbeinstellungen. Die einzelnen Befehle können auf `true`, `false`, oder `always` gesetzt werden:

```
color.branch  
color.diff  
color.interactive  
color.status
```

Darüber hinaus hat jede dieser Optionen Untereinstellungen, mit denen du bestimmte Farben für Teile der Ausgabe festlegen kannst, falls du die Farben überschreiben willst. Um beispielsweise die Metainformationen in deiner Diff-Ausgabe auf blauen Vordergrund, schwarzen Hintergrund und fetten Text zu setzen, kannst du Folgendes ausführen:

```
$ git config --global color.diff.meta "blue black bold"
```

Du kannst die Farbe auf einen der folgenden Werte setzen: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` oder `white`. Wenn du ein Attribut wie im vorherigen Beispiel fett wünschst, kannst du zwischen `bold` (fett), `dim` (abgedunkelt), `ul` (unterstrichen), `blink` (blinken) und `reverse` (Vorder- und Hintergrund vertauschen) wählen.

## Externe Merge- und Diff-Tools

Obwohl Git eine interne Diff-Implementierung hat, die wir in diesem Buch vorgestellt haben, kannst du stattdessen ein externes Tool verwenden. Du kannst auch ein grafisches Tool zum Mergen und Lösen von Konflikten einrichten, anstatt Konflikte manuell lösen zu müssen. Wir zeigen dir, wie du das Tool Perforce Visual Merge (P4Merge) einrichtest, um deine Diffs und Merge-Ansichten zu analysieren. P4Merge ist ein praktisches grafisches Tool und dazu noch kostenlos.

Es zu installieren, sollte kein Problem sein, denn P4Merge läuft auf allen wichtigen Plattformen. Wir verwenden Pfadnamen in den Beispielen, die auf macOS- und Linux-Systemen funktionieren. Unter Windows musst du `/usr/local/bin` in einen ausführbaren Pfad in deiner Umgebung umändern.

Starte mit [P4Merge von Perforce downloaden](#). Richte danach externe Wrapper-Skripte ein, um deine Befehle auszuführen. Wir verwenden hier den macOS-Pfad für die ausführbare Datei. In anderen Systemen sollte er so angepasst werden, dass er auf den Ordner verweist, in dem dein `p4merge` Binary installiert ist. Richte ein Merge-Wrapper-Skript mit dem Namen `extMerge` ein, das deine Binärdatei mit allen angegebenen Argumenten aufruft:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Der diff Wrapper überprüft, ob sieben Argumente angegeben sind und übergibt zwei davon an dein Merge-Skript. Standardmäßig übergibt Git die folgenden Argumente an das Diff-Programm:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Da du nur die Argumente `old-file` und `new-file` benötigst, verwendest du das Wrapper-Skript, um diese zu übergeben.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Du musst außerdem darauf achten, dass diese Tools ausführbar sind:

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Jetzt kannst du deine Konfigurationsdatei so einrichten, dass sie deine benutzerdefinierte Merge-Lösung und Diff-Tools nutzt. Dazu sind eine Reihe von benutzerdefinierten Einstellungen erforderlich: `merge.tool`, um Git mitzuteilen, welche Strategie zu verwenden ist, `mergetool.<tool>.cmd`, um anzugeben, wie der Befehl ausgeführt werden soll, `mergetool.<tool>.trustExitCode`, um Git mitzuteilen, ob der Exit-Code dieses Programms eine

erfolgreiche Merge-Lösung anzeigt oder nicht, und `diff.external`, um Git mitzuteilen, welchen Befehl es für diffs ausführen soll. Du kannst also entweder die vier Konfigurationsbefehle ausführen:

```
$ 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
```

oder du kannst die `~/.gitconfig` Datei bearbeiten und diese Zeilen hinzufügen:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"
  trustExitCode = false
[diff]
  external = extDiff
```

Wenn das alles konfiguriert ist, kannst du diff Befehle wie diesen ausführen:

```
$ git diff 32d1776b1^ 32d1776b1
```

Statt die Diff-Ausgabe auf der Kommandozeile zu erhalten, wird P4Merge von Git gestartet, was folgendermaßen aussieht:

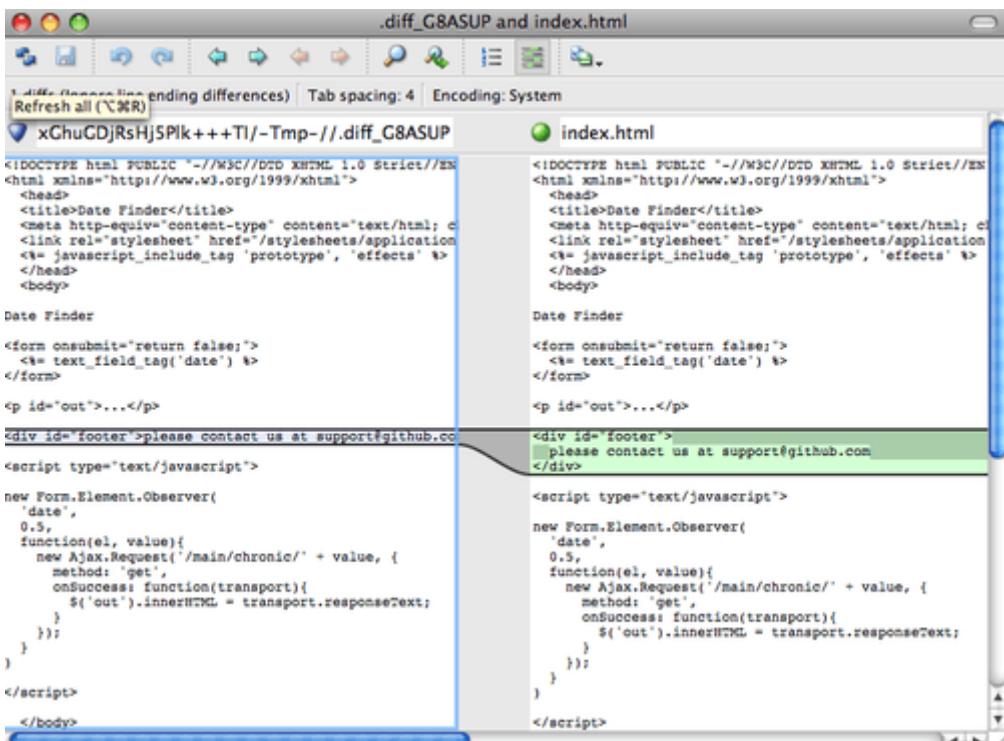


Figure 168. P4Merge

Wenn du versuchst, zwei Branches zu mergen und dabei Merge-Konflikte hast, kannst du den Befehl `git mergetool` ausführen. Er startet P4Merge, um diese Konflikte über das GUI-Tool zu lösen.

Das Tolle an diesem Wrapper-Setup ist, dass du deine Diff- und Merge-Tools einfach ändern kannst. Um beispielsweise deine Tools `extDiff` und `extMerge` so zu ändern, dass das KDiff3-Tool stattdessen ausgeführt wird, musst du nur deine `extMerge` Datei bearbeiten:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Nun wird Git das KDiff3-Tool für das Diff-Viewing und die Lösung der Merge-Konflikte verwenden.

Git ist standardmäßig so eingestellt, dass es eine Reihe anderer Tools zum Auflösen von Merges verwendet, ohne dass du die cmd-Konfiguration einrichten musst. Um eine Liste der unterstützten Tools anzuzeigen, versuche es wie folgt:

```
$ 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
    delwalker
    diffmerge
    diffuse
    ecmerge
    kdiff3
    meld
    tkdiff
    tortoisermerge
    xxdiff
```

Some of the tools listed above only work in a windowed environment. If run in a terminal-only session, they will fail.

Wenn du nicht daran interessiert bist, KDiff3 für diff zu verwenden, sondern es nur für die Merge-Auflösung verwenden willst und der Befehl `kdiff3` in deinem Pfad steht, dann kannst du Folgendes ausführen:

```
$ git config --global merge.tool kdiff3
```

Wenn du diese Methode verwendest, ohne die Dateien `extMerge` und `extDiff` einzurichten, verwendet Git KDiff3 für die Merge-Auflösung und das normale Git diff-Tool für Diffs.

## Formatierung und Leerzeichen

Formatierungs- und Leerzeichen-Probleme sind einige der frustrierendsten und schwierigsten Probleme, auf die viele Entwickler bei der Teamarbeit stoßen, vor allem in plattformübergreifenden Projekten. Es ist sehr einfach für Patches oder andere kollaborative Arbeiten, geringfügige Änderungen an Leerzeichen vorzunehmen, da die Editoren diese im Hintergrund einfügen. Falls deine Dateien jemals mit einem Windows-System in Kontakt kommen, werden ihre Zeilenenden möglicherweise ersetzt. Git hat einige Konfigurationsoptionen, um bei diesen Schwierigkeiten zu helfen.

### `core.autocrlf`

Wenn du unter Windows programmierst und mit Leuten arbeitest, die andere Systeme verwenden (oder umgekehrt), wirst du wahrscheinlich irgendwann auf Probleme mit den Zeilenenden treffen. Der Grund hierfür ist, dass Windows in seinen Dateien sowohl ein Carriage-Return-Zeichen als auch ein Linefeed-Zeichen für Zeilenumbrüche verwendet, während macOS- und Linux-Systeme nur das Linefeed-Zeichen verwenden. Viele Editoren unter Windows ersetzen im Hintergrund bestehende Zeilenenden im LF-Stil durch CRLF oder fügen beide Zeilenendezeichen ein, wenn der Benutzer die Eingabetaste drückt.

Git kann das durch automatische Konvertierung von CRLF-Zeilenden in LF übernehmen, wenn du eine Datei zum Index hinzufügst, und umgekehrt, wenn es Code auf dein Dateisystem auscheckt. Du kannst diese Funktionen mit der `core.autocrlf` Einstellung einschalten. Wenn du dich auf einem Windows-Computer befindest, setze die Einstellung auf `true`. Das konvertiert LF-Endungen in CRLF, wenn du Code auscheckst:

```
$ git config --global core.autocrlf true
```

Auf einem Linux- oder macOS-System, das LF-Zeilenden verwendet, solltest du nicht zulassen, dass Git Dateien automatisch konvertiert, wenn du sie auscheckst. Falls jedoch versehentlich eine Datei mit CRLF-Endungen hinzugefügt wird, kannst du dafür sorgen, dass Git sie korrigiert. Du kannst Git anweisen, CRLF beim Commit in LF zu konvertieren, aber nicht umgekehrt, indem du `core.autocrlf` auf `input` setzt:

```
$ git config --global core.autocrlf input
```

Dieses Setup sollte deine CRLF-Endungen in Windows-Checkouts nicht ändern, aber LF-Endungen auf macOS- und Linux-Systemen und im Repository.

Wenn du ein Windows-Programmierer bist, der ein reines Windows-Projekt durchführt, dann kannst du diese Funktionalität deaktivieren und die Carriage Returns im Repository aufzeichnen,

indem du den Konfigurationswert auf `false` setzt:

```
$ git config --global core.autocrlf false
```

### core.whitespace

Git wird von Hause aus mit einer Voreinstellung zur Erkennung und Behebung einiger Leerzeichen-Probleme installiert. Es kann nach sechs primären Leerzeichen-Problemen suchen. Drei sind standardmäßig aktiviert und können deaktiviert werden, und drei sind standardmäßig deaktiviert, können aber aktiviert werden.

Die drei, bei denen die Standardeinstellung eingeschaltet ist, sind `blank-at-eol`, das nach Leerzeichen am Ende einer Zeile sucht; `blank-at-eof`, das leere Zeilen am Ende einer Datei bemerkt und `space-before-tab`, das nach Leerzeichen vor Tabulatoren am Anfang einer Zeile sucht.

Die drei, die standardmäßig deaktiviert sind, aber eingeschaltet werden können, sind `indent-with-non-tab`, das nach Zeilen sucht, die mit Leerzeichen anstelle von Tabs beginnen (und von der Option `tabwidth` kontrolliert werden); `tab-in-indent`, das nach Tabs im Einzug einer Zeile sucht und `cr-at-eol`, das Git mitteilt, dass Carriage Returns am Ende von Zeilen OK sind.

Du kannst mit Git bestimmen, welche dieser Optionen aktiviert werden sollen. Setze dazu, getrennt durch Kommas, `core.whitespace` auf den gewünschten Wert (ein oder aus). Du kannst eine Option deaktivieren, indem du ein `-` (Minus-Zeichen) dem Namen voranstellst, oder den Standardwert verwendest, indem du ihn ganz aus der Zeichenkette der Einstellung entfernst. Wenn du z.B. möchtest, dass alles außer `space-before-tab` gesetzt wird, kannst du das so machen (wobei `trailing-space` eine Kurzform ist, um sowohl `blank-at-eol` als auch `blank-at-eof` abzudecken):

```
$ git config --global core.whitespace \
    trailing-space,-space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Oder du kannst nur den anzupassenden Teil angeben:

```
$ git config --global core.whitespace \
    -space-before-tab,indent-with-non-tab,tab-in-indent,cr-at-eol
```

Git wird diese Punkte erkennen, wenn du einen `git diff` Befehl ausführst und versuchen sie einzufärben, damit du sie gegebenenfalls vor dem Commit beheben kannst. Git wird diese Werte auch verwenden, um dir zu helfen, wenn du Patches mit `git apply` einspielst. Wenn du Patches installieren, kannst du Git bitten, dich zu warnen, wenn es Patches mit den angegebenen Leerzeichen-Problemen anwendet:

```
$ git apply --whitespace=warn <patch>
```

Du kannst auch Git versuchen lassen, das Problem automatisch zu beheben, bevor du den Patch einspielst:

```
$ git apply --whitespace=fix <patch>
```

Diese Optionen gelten auch für den Befehl `git rebase`. Wenn du Leerzeichen-Probleme committet hast, aber noch nicht zum Upstream gepusht hast, kannst du `git rebase --whitespace=fix` ausführen, damit Git Leerzeichen-Probleme automatisch behebt, während es die Patches neu schreibt.

## Server-Konfiguration

Für die Serverseite von Git stehen nicht annähernd so viele Konfigurationsoptionen zur Verfügung. Es gibt jedoch einige wichtige Einstellungen, die du beachten solltest.

### `receive.fsckObjects`

Git ist in der Lage zu kontrollieren, ob jedes während eines Pushs empfangene Objekt noch mit seiner SHA-1-Prüfsumme übereinstimmt und auf gültige Objekte zeigt. Standardmäßig tut es das jedoch nicht. Es ist eine ziemlich aufwändige Operation und kann den Betrieb verlangsamen, insbesondere bei großen Repositorys oder Pushes. Wenn du möchtest, dass Git bei jedem Push die Objektkonsistenz überprüft, kannst du es dazu zwingen, indem du `receive.fsckObjects` auf `true` setzt:

```
$ git config --system receive.fsckObjects true
```

Jetzt prüft Git die Integrität deines Repositorys, noch bevor jeder Push akzeptiert wird, um sicherzustellen, dass fehlerhafte (oder böswillige) Clients keine korrupten Daten einschleusen.

### `receive.denyNonFastForwards`

Wenn du Commits rebased, die du bereits gepusht hast, und dann versuchst, erneut zu pushen, oder anderweitig versuchst, einen Commit an einen Remote-Branch zu senden, der nicht den Commit enthält, auf den der Remote-Branch aktuell zeigt, wird dies abgelehnt. Im Allgemeinen ist das eine gute Richtlinie. Bei dem Rebase kannst du festlegen den Remote-Branch mit einem `-f` Flag in deinem Push-Befehl zu aktualisieren, wenn du weißt, was du tust.

Um Git anzuweisen, Force-Pushes abzulehnen, setze `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

Die andere Möglichkeit ist, das über serverseitige Receive-Hooks tun, die wir im weiteren Verlauf behandeln werden. Dieser Ansatz ermöglicht es dir, komplexere Dinge zu tun, wie z.B. einem bestimmten Teil der Benutzer die Möglichkeit „non-fast-forwards“ zu verweigern.

### `receive.denyDeletes`

Ein möglicher Workaround für die `denyNonFastForwards` Regel besteht darin, dass der Benutzer den Branch löscht und ihn dann mit einer neuen Referenz wieder pusht. Um das zu verhindern, setze `receive.denyDeletes` auf `true`:

```
$ git config --system receive.denyDeletes true
```

Dadurch wird das Löschen von Branches oder Tags verhindert – kein User darf das dann tun. Um dann Remote-Banches zu entfernen, musst du die ref-Dateien manuell vom Server entfernen. Es gibt weitere interessante Möglichkeiten, das auf Benutzerebene über ACLs zu realisieren, wie du in [Beispiel für Git-forcierte Regeln](#) erfahren wirst.

## Git-Attribute

Einige dieser Einstellungen können auch für einen Pfad angegeben werden, so dass Git diese Einstellungen nur für ein Unterverzeichnis oder eine Teilmenge von Dateien anwendet. Diese pfadspezifischen Einstellungen heißen Git-Attribute und werden entweder in einer `.gitattributes` Datei in einem deiner Verzeichnisse (normalerweise dem Stammverzeichnis deines Projekts) oder in der `.git/info/attributes` Datei festgelegt, falls du nicht willst, dass die Attributdatei mit deinem Projekt verknüpft wird.

Mit Hilfe von Attributen kannst du beispielsweise separate Merge-Strategien für einzelne Dateien oder Verzeichnisse eines Projekts festlegen, Git sagen, wie man Nicht-Text-Dateien unterscheidet oder den Inhalt von Git filtern lassen, bevor du ihn in Git ein- oder auscheckst. In diesem Abschnitt erfährst du mehr über einige der Attribute, die du in deinem Git-Projekt auf deine Pfade setzen kannst, und siehst einige Beispiele für die praktische Anwendung dieser Funktion.

## Binäre Dateien

Ein toller Kniff, für den du Git-Attribute verwenden kannst, ist das Erkennen von binären Dateien (falls es sonst nicht möglich ist, es herauszufinden) und Git anzuweisen, wie man mit diesen Dateien umgeht. So können beispielsweise einige Textdateien maschinell erzeugt und nicht mehr gedifft werden, während andere Binärdateien gedifft werden können. Du wirst sehen, wie du Git mitteilen kannst, welche welche ist.

### Binärdateien identifizieren

Einige Dateien sehen aus wie Textdateien, sind aber im Grunde genommen wie Binärdateien zu behandeln. Xcode-Projekte unter macOS enthalten beispielsweise eine Datei, die mit `.pbxproj` endet. Diese ist im Grunde genommen ein JSON-Datensatz (Klartext JavaScript Datenformat), der von der IDE auf die Festplatte geschrieben wird, deine Build-Einstellungen aufzeichnet usw. Obwohl es sich technisch gesehen um eine Textdatei handelt (weil sie eigentlich komplett UTF-8 ist), willst du sie nicht als solche behandeln, denn es handelt sich hierbei um eine sehr einfache Datenbank. Du kannst den Inhalt nicht mergen, wenn zwei Personen ihn ändern. Diffs sind im Allgemeinen nicht hilfreich. Die Datei ist für den internen Betrieb auf einem Rechner bestimmt. Im Prinzip solltest du sie wie eine Binärdatei behandeln.

Um Git anzuweisen, alle `.pbxproj` Dateien als Binärdaten zu behandeln, fügst du die folgende Zeile zu deiner `.gitattributes` Datei hinzu:

```
*.pbxproj binary
```

Jetzt wird Git nicht versuchen, CRLF-Probleme zu konvertieren oder zu beheben; noch wird es versuchen, ein Diff für Änderungen in dieser Datei zu berechnen oder auszugeben, wenn du `git show` oder `git diff` bei deinem Projekt ausführst.

## Unterschiede in Binärdateien vergleichen

Du kannst auch die Git-Attribut-Funktionalität verwenden, um Binärdateien effektiv zu unterscheiden. Dafür musst du Git mitteilen, wie es deine Binärdateien in ein Textformat konvertieren soll, das über das normale `diff` verglichen werden kann.

Vielleicht wirst du diese Technik nutzen, um eines der läufigsten Probleme zu lösen, die es gibt: die Versionskontrolle von Microsoft Word-Dokumenten. Wenn du Word-Dokumente versionieren willst, kannst du sie in ein Git-Repository legen und ab und zu committen; aber was nützt das? Wenn du `git diff` wie gewohnt startest, siehst du nur so etwas wie das hier:

```
$ git diff  
diff --git a/chapter1.docx b/chapter1.docx  
index 88839c4..4afcb7c 100644  
Binary files a/chapter1.docx and b/chapter1.docx differ
```

Du kannst zwei Versionen nicht direkt vergleichen, es sei denn, du checkst sie aus und scannst sie manuell, richtig? Wie sich herausstellt, kann man das ziemlich gut mit Git-Attributen machen. Füge die folgende Zeile in deine `.gitattributes` Datei ein:

```
*.docx diff=word
```

Hiermit erfährt Git, dass jede Datei, die mit diesem Suchmuster übereinstimmt (`.docx`), den Filter „word“ verwenden sollte, wenn du einen Diff anzeigen lassen willst, der Änderungen enthält. Was ist der Filter „word“? Du musst ihn einrichten. Nachfolgend konfigurierst du Git so, dass es das Programm `docx2txt` verwendet, um Word-Dokumente in lesbare Textdateien zu konvertieren, die es dann richtig auswertet.

Zuerst musst du `docx2txt` installieren. Du kannst es von <https://sourceforge.net/projects/docx2txt> herunterladen. Folgen den Anweisungen in der `INSTALL` Datei, um es an einen Ort zu platzieren, an dem deine Shell es finden kann. Als nächstes schreibst du ein Wrapper-Skript, um die Ausgabe in das von Git erwartete Format zu konvertieren. Erstellen eine Datei namens `docx2txt`, irgendwo innerhalb deines Pfads und füge diesen Inhalt hinzu:

```
#!/bin/bash  
docx2txt.pl "$1" -
```

Vergiss nicht, die Dateirechte mit `chmod a+x` zu ändern. Schließlich kannst du Git so einrichten, dass es dieses Skript verwendet:

```
$ git config diff.word.textconv docx2txt
```

Jetzt ist Git darüber informiert, dass es bei dem Versuch, einen Diff zwischen zwei Snapshots zu machen, der Dateien enthält, die auf `.docx` enden, diese Dateien durch den „word“ Filter laufen lassen soll, der als `docx2txt` Programm definiert ist. Auf diese Weise entstehen im Vorfeld gute Textversionen deiner Word-Dateien, die du leichter diffen kannst.

Hier ist ein Beispiel: Kapitel 1 dieses Buches wurde in das Word-Format konvertiert und in ein Git-Repository committet. Dann wurde ein neuer Absatz hinzugefügt. Hier ist das, was `git diff` zeigt:

```
$ 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 sagt uns sehr treffend und klar, dass wir die Zeichenkette „Testing: 1, 2, 3.“ hinzugefügt haben, was richtig ist. Was aber nicht perfekt ist: Formatierungsänderungen würden hier nicht angezeigt – aber es funktioniert.

Ein weiteres spannendes Problem, das du auf diese Weise lösen kannst, ist die diffen von Bilddateien. Die eine Methode besteht darin, Bilddateien durch einen Filter zu leiten, der ihre EXIF-Informationen extrahiert – das sind Metadaten, die mit den meisten Bildformaten aufgezeichnet werden. Nach dem Herunterladen und Installieren des `exiftool` Programms kannst du damit die Metadaten deiner Bilder in Text umwandeln. So zeigt dir das Diff zumindest eine schriftliche

Darstellung der vorgenommenen Änderungen an. Füge die folgende Zeile in deine `.gitattributes` Datei ein:

```
*.png diff=exif
```

So konfigurierst du Git, um dieses Tool zu verwenden:

```
$ git config diff.exif.textconv exiftool
```

Wenn du ein Bild in deinem Projekt ersetzt und dann `git diff` ausführst, siehst du etwas ähnliches wie hier:

```
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
```

Du kannst leicht erkennen, dass sich sowohl die Dateigröße als auch die Bildgröße geändert hat.

## Schlüsselwort-Erweiterung

Die SVN- oder CVS-artige Schlüsselwort-Erweiterung wird oft von Entwicklern gefordert, die mit diesen Systemen arbeiten. Das Hauptproblem in Git ist, dass du eine Datei mit Informationen über einen Commit nach dem Committen nicht ändern kannst, da zuerst die Prüfsumme der Datei von Git ermittelt wird. Wenn die Datei ausgecheckt ist kannst du jedoch Text in sie einfügen und ihn wieder entfernen, bevor er zu einem Commit hinzugefügt wird. Die Git-Attribute stellen dir dafür zwei Möglichkeiten zur Verfügung.

Erstens kannst du die SHA-1-Prüfsumme eines Blobs automatisch in ein `$Id$` Feld in der Datei einfügen. Wird dieses Attribut auf eine Datei oder eine Gruppe von Dateien gesetzt, dann wird Git beim nächsten Auschecken dieses Branchs das entsprechende Feld durch den SHA-1 des Blobs ersetzen. Dabei muss man beachten, dass es nicht das SHA-1 des Commits ist, sondern das des Blobs an sich. Füge die folgende Zeile in deine `.gitattributes` Datei ein:

```
*.txt ident
```

Füge eine \$Id\$ Referenz einer Testdatei hinzu:

```
$ echo '$Id$' > test.txt
```

Beim nächsten Auschecken dieser Datei fügt Git den SHA-1 des Blobs ein:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Dieses Ergebnis ist jedoch von begrenztem Nutzen. Wenn du die Ersetzung von Schlüsselwörtern in CVS oder Subversion verwendet hast, kannst du einen Datums-Stempel hinzufügen. Der SHA-1 ist nicht allzu hilfreich, weil er eher willkürlich ist und du nicht erkennen kannst, ob ein SHA-1 älter oder neuer als ein anderer ist.

Es zeigt sich, dass du deine eigenen Filter schreiben kannst, um Ersetzungen in Dateien bei einem Commit/Checkout durchzuführen. Diese werden als „saubere“ (engl. clean) und „verschmutzte“ (engl. smudge) Filter bezeichnet. In der Datei [.gitattributes](#) kannst du einen Filter für bestimmte Pfade setzen und dann Skripte einrichten, die die Dateien verarbeiten, kurz bevor sie ausgecheckt werden (für „smudge“, siehe [Der „smudge“ Filter wird beim Auschecken ausgeführt](#)) und kurz bevor sie zum Commit vorgemerkt werden (für „clean“, siehe [Der „clean“ Filter wird ausgeführt, wenn Dateien zum Commit vorgemerkt werden](#)). Diese Filter können so eingestellt werden, um damit alle möglichen interessanten Aufgaben zu erledigen.

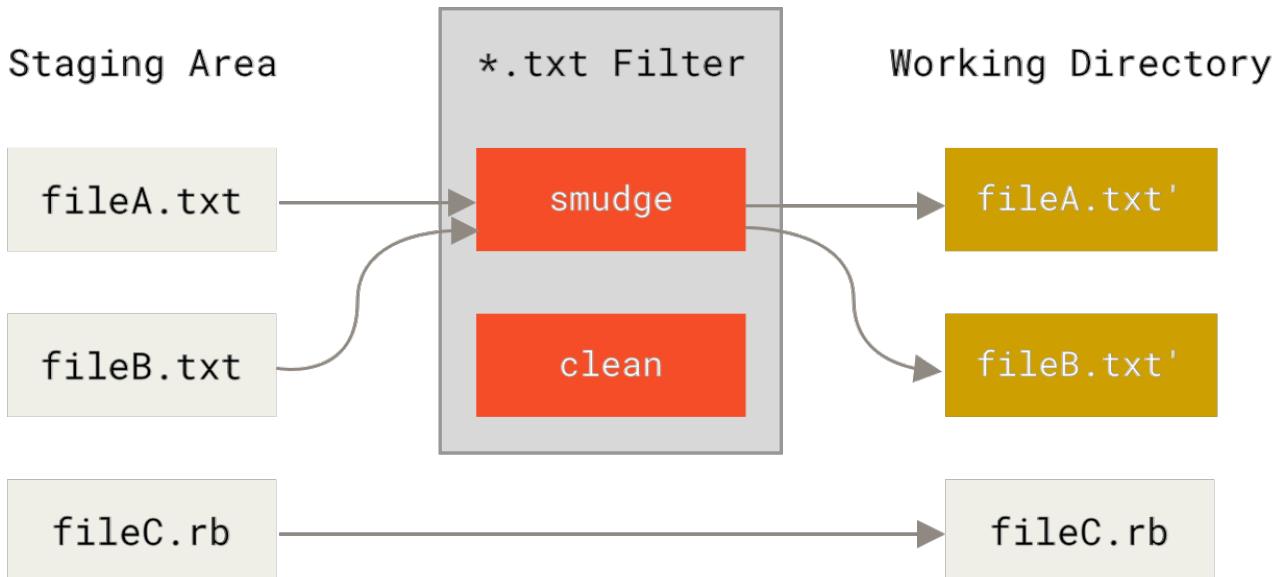


Figure 169. Der „smudge“ Filter wird beim Auschecken ausgeführt

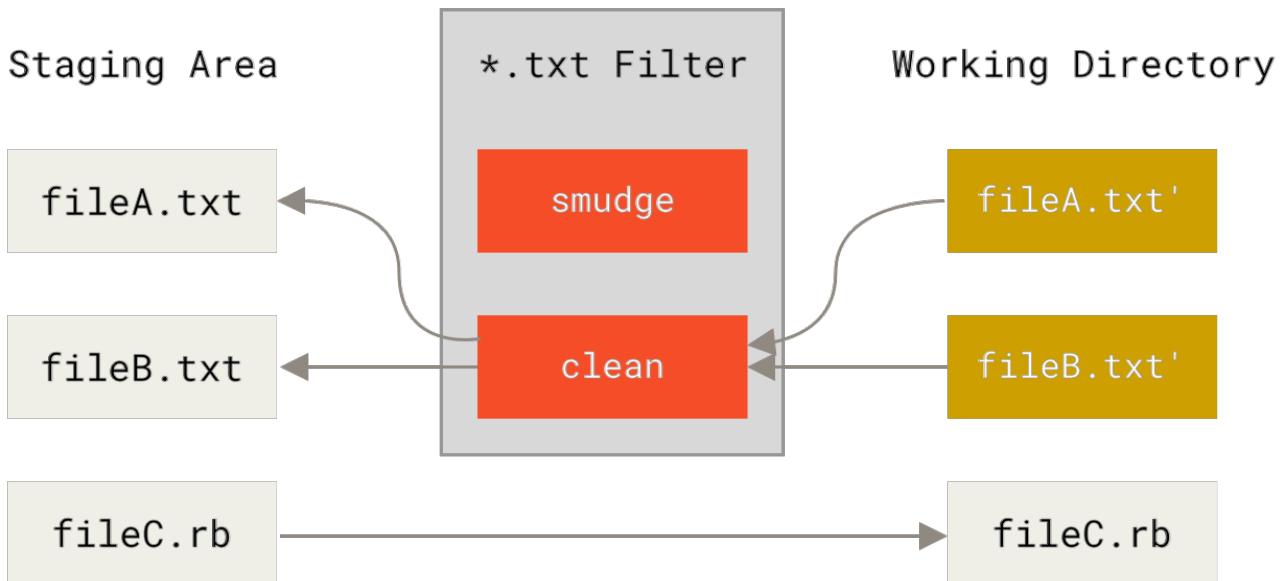


Figure 170. Der „clean“ Filter wird ausgeführt, wenn Dateien zum Commit vorgemerkt werden

Die ursprüngliche Commit-Meldung dieser Funktion zeigt ein einfaches Anwendungsbeispiel, wie du deinen C-Quellcode vor dem Commit durch das `indent` Programm laufen lassen kannst. Du kannst es so einrichten, dass das Filterattribut in deiner `.gitattributes` Datei so gesetzt ist, dass `*.c` Dateien mit dem Filter „`indent`“ gefiltert werden:

```
*.c filter=indent
```

Dann weist du Git an, was der „`indent`“ Filter bei `smudge` und `clean` bewirkt:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

Wenn du nun Dateien committest, die mit `*.c` übereinstimmen, wird Git sie das `Indent`-Programm durchlaufen lassen, bevor es sie der Staging-Area hinzufügt. Dann werden sie durch das `cat` Programm laufen, bevor es sie wieder auf die Festplatte auscheckt. Das `cat` Programm macht im Grunde genommen nichts: Es übergibt die gleichen Daten, die es bekommt. Diese Kombination filtert effektiv alle C-Quellcode-Dateien durch Einrückung (engl. `indent`), bevor sie committet werden.

Ein weiteres interessantes Beispiel ist die `$Date$` Schlüsselwort-Erweiterung im RCS-Stil. Um das richtig anzuwenden, benötigst du ein kleines Skript, das einen Dateinamen entgegennimmt, das letzte Commit-Datum für dieses Projekt ermittelt und das Datum in die Datei einfügt. Hier ist ein kleines Ruby-Skript, das das erledigt:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Alles, was das Skript macht, ist das neueste Commit-Datum aus dem `git log` Befehl zu ermitteln, es in alle `$Date$` Zeichenketten zu ersetzen, die es in stdin erkennt und das Ergebnis auszugeben. Es sollte nicht schwierig sein, in der Programmiersprache zu erstellen mit der du am besten zurechtkommst. Du kannst diese Datei `expand_date` nennen und in deinem Pfad aufnehmen. Nun musst du einen Filter in Git einrichten (nenne ihn z.B. `dater`) und ihm sagen, dass er deinen `expand_date` Filter verwenden soll, um die Dateien beim Auschecken zu bearbeiten (engl. smudge). Ein Perl-Ausdruck dient dazu, das beim Commit zu bereinigen:

```
$ git config filter.dater.smudge expand_date  
$ git config filter.dater.clean 'perl -pe "s/\$\$Date[^\$\$]*\$\$/\$Date\\\$/"'
```

Dieses Perl-Snippet entfernt alles, was es in einer `$Date$` Zeichenkette sieht, um an den Ausgangspunkt zurückzukehren. Nachdem dein Filter jetzt fertig ist, kannst du ihn testen, indem du ein Git-Attribut für diese Datei einrichtest, das den neuen Filter aktiviert und eine Datei mit deinem `$Date$` Schlüsselwort erstellst:

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

Wenn du diese Änderungen committest und die Datei erneut auscheckst, ist das Schlüsselwort ordnungsgemäß ersetzt:

```
$ git add date_test.txt .gitattributes  
$ git commit -m "Test date expansion in Git"  
$ rm date_test.txt  
$ git checkout date_test.txt  
$ cat date_test.txt  
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Hier siehst du, wie leistungsfähig diese Technik für maßgeschneiderte Anwendungen sein kann. Du solltest jedoch vorsichtig sein, da die `.gitattributes` Datei committed und mit dem Projekt weitergegeben wird, der Treiber (hier: `dater`) aber nicht. Es wird also nicht überall funktionieren. Wenn du diese Filter entwirfst, sollten deine Mitstreiter, bei fehlerhaften Ergebnissen, dennoch problemlos in der Lage sein das Projekt einwandfrei laufen zu lassen.

## Exportieren deiner Repositorys

Mit Git-Attribut-Daten kannst du interessante Aufgaben beim Exportieren deines Projekts erledigen.

### export-ignore

Du kannst Git anweisen, bestimmte Dateien oder Verzeichnisse nicht zu exportieren, wenn du ein Archiv erstellst. Wenn es ein Unterverzeichnis oder eine Datei gibt, die du nicht in deine

Archivdatei aufnehmen möchtest, aber in dein Projekt einchecken möchtest, kannst du diese Dateien über das Attribut **export-ignore** markieren.

Angenommen, du hast einige Testdateien in einem **test/** Unterverzeichnis und es ist nicht sinnvoll, sie in den Tarball-Export deines Projekts aufzunehmen. Man kann die folgende Zeile zu der Datei mit den Git-Attributen hinzufügen:

```
test/ export-ignore
```

Wenn du nun **git archive** ausführst, um einen Tarball deines Projekts zu erstellen, wird dieses Verzeichnis nicht in das Archiv aufgenommen.

### **export-subst**

Beim Exportieren von Dateien für das Deployment kannst du die Formatierung und Schlüsselwort-Erweiterung von **git log** auf ausgewählte Teile von Dateien anwenden, die mit dem Attribut **export-subst** markiert sind.

Wenn du beispielsweise eine Datei mit dem Namen **LAST\_COMMIT** in dein Projekt aufnehmen möchtest und Metadaten über den letzten Commit automatisch in das Projekt eingespeist werden sollen, sobald **git archive** läuft, kannst du beispielsweise deine **.gitattributes** und **LAST\_COMMIT** Dateien so einrichten:

```
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'
```

Wenn du **git archive** ausführst, sieht der Inhalt der archivierten Datei so aus:

```
$ git archive HEAD | tar xCf ../deployment-testing -
$ cat ../deployment-testing/LAST_COMMIT
Last commit date: Tue Apr 21 08:38:48 2009 -0700 by Scott Chacon
```

Die Ersetzungen können z.B. die Commit-Meldung und beliebige **git notes** enthalten, und **git log** kann einfache Zeilenumbrüche durchführen:

```
$ 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.
```

Das so entstandene Archiv ist für das Deployment geeignet, aber wie jedes andere exportierte Archiv ist es nicht zur weiteren Entwicklungsarbeit verwendbar.

## Merge-Strategien

Du kannst Git-Attribute auch verwenden, um Git anzuweisen, verschiedene Merge-Strategien für spezifische Dateien in deinem Projekt zu verwenden. Eine sehr nützliche Option ist es, Git anzuweisen, dass es nicht versuchen soll, bestimmte Dateien zusammenzuführen, wenn sie Konflikte haben, sondern deine Version der Daten zu benutzen, anstelle der von jemand anderem.

Das ist nützlich, wenn ein Branch in deinem Projekt auseinander gelaufen ist oder sich geändert hat, du jedoch weiterhin Änderungen von ihm mergen möchtest und dabei einige Dateien ignorieren möchtest. Angenommen, du hast eine Datenbank-Einstellungsdatei namens `database.xml`, die sich in zwei Branches voneinander unterscheidet. Du willst in deinem zweiten Branch mergen, ohne die Datenbankdatei zu beschädigen. Dann kannst du so ein Attribut einrichten:

```
database.xml merge=ours
```

Dann definierst du die Dummy-Merge-Strategie `ours` mit:

```
$ git config --global merge.ours.driver true
```

Wenn du in dem zweiten Branch mergst, siehst du statt Merge-Konflikte mit der Datei `database.xml` folgendes:

```
$ git merge topic
Auto-merging database.xml
Merge made by recursive.
```

In diesem Fall bleibt die Datei `database.xml` auf der Version, die sie ursprünglich gehabt hat.

## Git Hooks

Wie viele andere Versionskontrollsysteme hat Git eine Option, um benutzerdefinierte Skripte zu starten, wenn wichtige Ereignisse eintreten. Es gibt zwei Gruppen dieser Hooks: client-seitig und server-seitig. Client-seitige Hooks werden durch Operationen wie Commit und Merging ausgelöst, während server-seitige Hooks bei Netzwerkoperationen wie dem Empfangen von Push-Commits

ausgeführt werden. Du kannst diese Hooks für die unterschiedlichsten Dinge verwenden.

## Einen Hook installieren

Alle Hooks werden im Unterverzeichnis `hooks` des Git-Verzeichnisses gespeichert. In den meisten Projekten ist das `.git/hooks`. Wenn du ein neues Repository mit `git init` einrichtest, füllt Git das `hooks`-Verzeichnis mit einer Reihe von Beispielskripten, von denen viele für sich allein genommen recht nützlich sind. Sie dokumentieren aber auch die Input-Werte jedes Skripts. Alle Beispiele sind als shell-Skripte verfasst, wobei in einigen Perl eingebaut ist. Jedes richtig benannten ausführbaren Skripte sollte funktionieren. Du kannst sie in Ruby oder Python oder einer beliebigen anderen Sprache schreiben, mit der du vertraut bist. Wenn du die mitgelieferten Hook-Skripte verwenden möchtest, musst du sie umbenennen. Ihre Dateinamen enden alle mit `.sample`.

Füge zum Aktivieren eines Hook-Skripts eine Datei in das `hooks` Unterverzeichnis deines `.git`-Verzeichnisses ein, die einen passenden Namen trägt (ohne Erweiterung) und ausführbar ist. Ab diesem Zeitpunkt sollte es aufgerufen werden können. Wir werden die meisten wichtigen Hook-Dateinamen hier besprechen.

## Clientseitige Hooks

Es gibt viele clientseitige Hooks. Dieser Abschnitt unterteilt sie in Committing-Workflow-Hooks, E-Mail-Workflow-Skripte und alles andere.



Beachte, dass clientseitige Hooks **nicht** kopiert werden, wenn du ein Repository klonst. Wenn du mit diesen Skripten beabsichtigen, die Einhaltung einer Richtlinie durchzusetzen, solltest du es auf der Serverseite machen. Sieh dir am Besten das Beispiel in [Beispiel für Git-forcierte Regeln](#) an.

### Committing-Workflow Hooks

Die ersten vier Hooks beziehen sich auf den Committing-Prozess.

Als erstes wird der `pre-commit` Hook ausgeführt, bevor du eine Commit-Nachricht eintippen kannst. Er dient dazu, den Snapshot zu untersuchen, der übertragen werden soll. Oder du kannst herausfinden, ob du etwas vergessen hast, um sicherzustellen, dass Tests ausgeführt werden oder was immer du im Code überprüfen möchtest. Falls ein Status ungleich Null erkannt wird, bricht der Hook den Commit ab. Du kannst das aber mit `git commit --no-verify` umgehen. Du kannst verschiedene Optionen ausführen, wie z.B. die Suche nach dem Code Style (`lint` oder ähnliches ausführen), die Suche nach Leerzeichen am Ende des Textes (der Standard-Hook macht genau das) oder die Suche nach einer geeigneten Dokumentation zu neuen Methoden.

Der `prepare-commit-msg` Hook wird ausgeführt, bevor der Commit-Message-Editor gestartet wird, aber nachdem die Standardmeldung erstellt wurde. Du kannst die Standardnachricht bearbeiten, noch bevor der Commit-Autor sie sieht. Dieser Hook verwendet einige Parameter: den Pfad zur Datei, die die Commit-Nachricht bisher enthält, die Art des Commits und den Commit SHA-1, wenn es sich um einen geänderten Commit handelt. In der Regel ist dieser Hook für normale Commits nicht geeignet. Er ist eher für Commits gedacht, bei denen die Standardnachricht automatisch generiert wird, wie z.B. vordefinierte Commit-Nachrichten, Merge Commits, Squashed Commits und modifizierte Commits. Du kannst es in Verbindung mit einer Commit-Vorlage verwenden, um

Informationen automatisiert einzufügen.

Der `commit-msg` Hook übernimmt einen Parameter, der wiederum den Pfad zu einer temporären Datei angibt, die die vom Entwickler geschriebene Commit-Beschreibung enthält. Wenn dieses Skript mit Status ungleich Null endet, bricht Git den Commit-Prozess ab. So kannst du deinen Projektstatus oder deine Commit-Beschreibung überprüfen, bevor du einen Commit zulässt. Im letzten Abschnitt dieses Kapitels werden wir demonstrieren, wie du mit diesem Hook überprüfen kannst, ob deine Commit-Beschreibung mit einem vorgegebenen Muster übereinstimmt.

Nachdem der gesamte Commit-Prozess abgeschlossen ist, wird der `post-commit` Hook gestartet. Er benötigt keine Parameter, aber du kannst den letzten Commit aufrufen, indem du `git log -1 HEAD` aufrufst. Im Allgemeinen wird dieses Skript für Benachrichtigungen oder ähnliches verwendet.

## E-Mail-Workflow-Hooks

Du kannst drei clientseitige Hooks für einen E-Mail-basierten Workflow einrichten. Alle werden vom `git am` Befehl aufgerufen, so dass du ohne weiteres zum nächsten Abschnitt springen kannst, wenn du diesen Befehl in deinem Workflow nicht verwendest. Wenn du Patches per E-Mail erhältst, die von `git format-patch` vorbereitet wurden, dann könnten sich einige davon für dich als nützlich erweisen.

Der zuerst ausgeführte Hook ist `applypatch-msg`. Dafür wird ein einziges Argument verwendet: der Name der temporären Datei, die die vorgeschlagene Commit-Beschreibung enthält. Git bricht den Patch ab, wenn der Status dieses Skripts mit ungleich Null endet. Du kannst das verwenden, um sicherzugehen, dass eine Commit-Beschreibung korrekt formatiert ist oder um die Nachricht zu normalisieren, indem du sie vom Skript bearbeiten lässt.

Der nächste Hook, der beim Anwenden von Patches über `git am` läuft, ist `pre-applypatch`. Etwas verwirrend ist, dass er *nach* dem Einspielen des Patches, aber *vor* einem Commit ausgeführt wird. Das ermöglicht es dir den Snapshot zu untersuchen, bevor du den Commit durchführst. Mit diesem Skript kannst du Tests durchführen oder den Verzeichnisbaum anderweitig durchsuchen. Wenn etwas fehlt oder die Tests nicht erfolgreich waren, wird das `git am` Skript durch Exit ungleich Null abgebrochen, ohne den Patch zu übertragen.

Der letzte Hook, der während einer `git am` Operation läuft, ist `post-applypatch`, der nach dem Commit ausgeführt wird. Du kannst ihn verwenden, um eine Gruppe oder den Autor des Patches darüber zu informieren, dass du einen Patch gepulled hast. Mit diesem Skript kannst du den Patch-Prozess nicht stoppen.

## Andere Client-Hooks

Der `pre-rebase` Hook läuft, noch bevor du etwas rebased und kann den Prozess stoppen, wenn du ihn mit einem Wert ungleich Null beendest. Du kannst diesen Hook dazu nutzen, das Rebasing von bereits gepushten Commits zu unterbinden. Der Beispiel-Hook `pre-rebase`, den Git installiert macht das, obwohl er einige Voreinstellungen enthält, die möglicherweise nicht mit deinem Workflow harmonieren.

Der `post-rewrite` Hook wird von Befehlen durchgeführt, die Commits ersetzen, wie z.B. `git commit --amend` und `git rebase` (allerdings nicht mit `git filter-branch`). Sein einziges Argument ist der Befehl, der das Rewrite ausgelöst hat, und er empfängt eine Liste der Rewrites in `stdin`. Dieser Hook

hat die gleichen Einsatzmöglichkeiten wie `post-checkout` und `post-merge`.

Nachdem du einen erfolgreichen `git checkout` durchgeführt hast, läuft der `post-checkout` Hook. Du kannst damit dein Arbeitsverzeichnis für deine Projektumgebung entsprechend einrichten. Dies kann bedeuten, große Binärdateien zu verschieben, für die du keine Quellcodeverwaltung benötigst oder automatisch Dokumentation zu generieren oder etwas in dieser Richtung.

Der `post-merge` Hook läuft nach einem erfolgreich abgeschlossenen `merge` Befehl. Damit kannst du Daten im Verzeichnisbaum wiederherstellen, die Git nicht überwachen kann, wie z.B. die Zugriffsrechte. Dieser Hook kann das Vorhandensein von Dateien außerhalb der Git-Kontrolle überprüfen, die du möglicherweise kopieren möchtest, wenn sich der Arbeitsbaum ändert.

Der `pre-push` Hook wird während des `git push` aufgerufen, nachdem die Remote-Refs aktualisiert wurden, aber noch bevor irgendwelche Objekte übertragen wurden. Er empfängt den Namen und die Position des Remotes als Parameter und eine Liste der zu aktualisierenden Referenzen über `stdin`. Du kannst damit eine Reihe von Referenz-Updates validieren, bevor ein Push stattfindet (ein Exit-Code ungleich Null bricht den Push ab).

Git führt gelegentlich eine automatische Speicherbereinigung (engl. garbage collection) als Teil seiner regulären Funktion durch, indem es `git gc --auto` aufruft. Der `pre-auto-gc` Hook wird kurz vor der Garbage Collection aufgerufen und kann verwendet werden, um dich darüber zu informieren oder um die Speicherbereinigung abzubrechen, wenn der Zeitpunkt nicht günstig ist.

## Serverseitige Hooks

Zusätzlich zu den clientseitigen Hooks kannst du als Systemadministrator einige wichtige serverseitige Hooks verwenden, um nahezu jede Art von Richtlinien für dein Projekt durchzusetzen. Diese Skripte werden vor und nach dem Push auf dem Server ausgeführt. Die Pre-Hooks können jederzeit durch einen Exit-Code ungleich Null den Push zurückweisen und eine Fehlermeldung an den Client zurücksenden. So kannst du beliebig komplexe Push-Richtlinie einrichten.

### `pre-receive`

Das erste Skript, das ausgeführt wird, wenn ein Push von einem Client verarbeitet wird, ist `pre-receive`. Es wird eine Liste von Referenzen übernommen, die von `stdin` gepusht werden. Wenn der Exit-Code ungleich Null ist, wird keine von ihnen akzeptiert. Du kannst diesen Hook benutzen, um bestimmte Aktionen auszuführen, wie z.B. sicherzustellen, dass keine der aktualisierten Referenzen „non-fast-forwards“ sind oder um die Zugriffskontrolle für alle mit dem Push geänderten Refs und Dateien durchzuführen.

### `update`

Das `update` Skript ist dem `pre-receive` Skript sehr ähnlich, nur dass es für jeden Branch einmal ausgeführt wird, den der Pusher versucht zu aktualisieren. Wenn der Pusher versucht, in verschiedene Branches zu pushen, läuft `pre-receive` nur einmal, während das Update einmal pro Branch läuft, auf den gepusht wird. Satt aus `stdin` zu lesen, verwendet dieses Skript drei Argumente: den Namen der Referenz (Branch), den SHA-1, auf den die Referenz vor dem Push zeigte und den SHA-1, den der Benutzer zu pushen versucht. Wenn das Aktualisierungsskript den Status ungleich Null (engl. non-zero) ausgibt, wird nur diese Referenz abgelehnt; andere

Referenzen können noch aktualisiert werden.

### post-receive

Der `post-receive` Hook wird nach Abschluss des gesamten Prozesses ausgeführt und kann zur Aktualisierung anderer Dienste oder zur Benachrichtigung von Benutzern verwendet werden. Er verwendet die gleichen stdin-Daten wie auch der `pre-receive` Hook. Beispiele umfassen das Mailen an eine Liste, die Benachrichtigung eines Continuous Integration Servers oder die Aktualisierung eines Ticket-Tracking-Systems. Du kannst sogar die Commit-Nachrichten analysieren, um zu sehen, ob Tickets geöffnet, geändert oder geschlossen werden müssen. Dieses Skript kann den Push-Prozess nicht stoppen und der Client trennt die Verbindung erst, wenn er mit seiner aktuellen Aktivität fertig ist. Passe daher auf, wenn du eine Aktion durchführst, die sehr lang dauern kann.



Wenn du ein Skript/einen Hook schreibst, den andere lesen sollen, bevorzuge die langen Versionen der Befehlszeilenflags. In sechs Monaten wirst du dafür dankbar sein.

## Beispiel für Git-forcierte Regeln

In diesem Abschnitt wirst du das Erlernte nutzen, um einen Git-Workflow einzurichten, der ein benutzerdefiniertes Commit-Beschreibungs-Format prüft und nur bestimmten Benutzern erlaubt, ausgewählte Unterverzeichnisse in einem Projekt zu ändern. Du erstellst Client-Skripte, die den Entwickler erkennen lassen, ob ihr Push abgelehnt wird. Ebenso werden Server-Skripte erstellt, die die Einhaltung von Richtlinien erzwingen.

Die hier vorgestellten Skripte sind in Ruby geschrieben, teils wegen unserem Know-How, teils aber auch weil Ruby leicht zu lesen ist (auch wenn man es nicht schreiben kann). Allerdings kann jede beliebige Programmiersprache verwendet werden. Alle mit Git vertriebenen Beispiel-Hook-Skripte sind entweder in Perl oder Bash verfasst. Diese kannst du dir bei Bedarf anschauen und anpassen.

### Serverseitiger Hook

Die gesamte serverseitige Arbeit wird in die `update` Datei in deinem `hooks` Verzeichnis übernommen. Der `update` Hook läuft einmal pro gepushtem Branch und benötigt drei Argumente:

- Der Name der Referenz, zu der gepusht wird
- Die alte Revision, in der sich dieser Branch befunden hat
- Die neue Revision, die gepusht werden soll

Wenn der Push über SSH ausgeführt wird, hast du auch Zugriff auf den Benutzer, der den Push durchführt. Auch wenn du es jedem erlaubt hast, sich mit einem bestimmten Benutzer (z.B. „git“) über die Public-Key-Authentifizierung zu verbinden, musst du diesem Benutzer möglicherweise einen Shell-Wrapper zur Verfügung stellen, der anhand des öffentlichen Schlüssels ermittelt, welcher Benutzer sich verbündet, und eine entsprechende Umgebungsvariable festlegen. Hier gehen wir davon aus, dass sich der verbündende Benutzer in der Umgebungsvariablen `$USER` befindet, so dass dein Update-Skript mit dem Sammeln aller benötigten Informationen beginnt:

```

#!/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, das sind globale Variablen. Verurteile es nicht – es ist auf diese Weise leichter zu demonstrieren.

### Ein bestimmtes Commit-Message-Format erzwingen

Deine erste Herausforderung besteht darin, die Einhaltung eines bestimmten Formats für jede Commit-Nachricht durchzusetzen. Nur um ein Ziel zu haben, gehst du davon aus, dass jede Nachricht eine Zeichenkette enthalten muss, die wie „ref: 1234“ aussieht. Du möchtest, dass jeder Commit auf einen Arbeitsschritt in deinem Ticketing-System verweist. Du musst dir jeden Commit ansehen, der gepusht wird. Du prüfst, ob sich diese Zeichenkette in der Commit-Beschreibung befindet. Falls die Zeichenkette bei einem der Commits fehlt, wird der Push mit einem Exit-Status ungleich Null abgelehnt.

Du kannst eine Liste der SHA-1-Werte aller gepushten Commits erhalten, indem du die Werte `$newrev` und `$oldrev` verwendest und sie an das Git-Basiskommando (engl. Plumbing Command) `git rev-list` übergibst. Das ist im Grunde genommen der Befehl `git log`, der aber standardmäßig nur die SHA-1-Werte und keine anderen Informationen ausgibt. Um also eine Liste aller Commit SHA-1-en zu erhalten, die zwischen zwei verschiedenen Commit SHA-1 liegen, kannst du in etwa so vorgehen:

```

$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cf0e475

```

Du kannst diese Ausgabe nehmen, die Nachricht für jeden dieser Commit SHA-1s abgreifen und diese Nachricht gegen einen regulären Ausdruck testen, der nach einem Zeichen-Muster sucht.

Du musst dir überlegen, wie du die Commit-Nachricht von jedem dieser Commits erhältst. Um die unformatierten Commit-Daten zu ermitteln, kannst du ein anderes Basiskommando namens `git cat-file` verwenden. Wir werden alle diese Basiskommandos in Kapitel 10, [Git Interna](#) im Detail betrachten. Dieser Befehl gibt Folgendes aus:

```

$ git cat-file commit ca82a6
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7

```

```
author Scott Chacon <schacon@gmail.com> 1205815931 -0700
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

Change the version number

Eine gibt eine einfache Möglichkeit, um die Commit Nachricht mit einem SHA-1-Wert zu erhalten. Nimm dazu einfach den Text nach der ersten Leerzeile. Auf Unix-Systemen kannst du mit dem **sed** Befehl arbeiten:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
Change the version number
```

Du kannst auf diese Weise die Commit-Nachricht von jedem Commit, der gepusht werden soll, übernehmen und mit einem Exit-Code ungleich Null beenden, wenn du etwas findest, das nicht mit deinen Regeln übereinstimmt. Um das Skript zu verlassen und den Push abzulehnen, gib einen Rückgabewert ungleich Null aus. Die gesamte Methode sieht dann so aus:

```
$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
```

Wenn du das in dein **update** Skript einfügst, werden Updates mit Commit-Nachrichten, die nicht deinen Regel entsprechen, abgelehnt.

## Ein benutzerbasiertes ACL-System einrichten

Angenommen, du möchtest einen Prozess hinzufügen, der eine Zugriffskontrollliste (ACL) verwendet, die festlegt, welche Benutzer Änderungen an welchen Teilen deines Projekts vornehmen dürfen. Einige Personen haben vollen Zugriff, andere können Änderungen nur zu bestimmten Unterzeichnissen oder bestimmten Dateien pushen. Um das zu erreichen, musst du diese Regeln in eine **acl** Datei schreiben, die in deinem Git-Repository auf dem Server liegt. Mit dem **update** Hook kannst du diese Regeln prüfen. So kannst du feststellen, welche Dateien für die zu übertragenden Commits eingeführt werden und entscheiden, ob der Benutzer, der den Push durchführt, Zugriff hat, um diese Dateien zu aktualisieren.

Zuerst musst du deine ACL-Datei erstellen. Hier verwendest du ein Format ähnlich dem CVS ACL-

Mechanismus: Es verwendet eine Reihe von Zeilen, wobei das erste Feld **avail** (verfügbar) oder **unavail** (nicht verfügbar) ist. Das nächste Feld ist eine kommagetrennte Liste der gültigen Benutzer und das letzte Feld ist der Pfad, für den die Regel gilt (blank bedeutet Open Access). Alle diese Felder werden durch ein Pipe-Zeichen (|) getrennt.

In diesem Beispiel hast du ein Reihe von Administratoren, einige Dokumentations-Autoren mit Zugriff auf das **doc** Verzeichnis und einen Entwickler, der nur Zugriff auf die Verzeichnisse **lib** und **tests** hat. deine ACL-Datei sieht dann wie folgt aus:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests
```

Du beginnst damit, diese Daten in eine Struktur einzulesen, die du weiterverwenden kannst. In diesem Beispiel wirst du, um das Beispiel einfach zu halten, nur die **avail** Anweisungen einführen. Hier folgt eine Methode, die dir ein assoziatives Array liefert, wobei der Schlüssel der Benutzername und der Wert ein Array von Pfaden ist, auf die der Benutzer Schreibzugriff hat:

```
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
```

In der zuvor angesehenen ACL-Datei gibt die **get\_acl\_access\_data** Methode eine Datenstruktur zurück, die wie folgt aussieht:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Nachdem du nun die Berechtigungen geklärt hast, musst du ermitteln, welche Pfade die gepuschten

Commits geändert haben. So kannst du sicherstellen, dass der Benutzer, der gepusht hat, Zugriff auf alle diese Pfade erhält.

Mit der Option `--name-only` des `git log` Befehls kannst du relativ einfach sehen, welche Dateien in einem einzelnen Commit geändert wurden (wurde in Kapitel 2, [Git Grundlagen](#) kurz erwähnt):

```
$ git log -1 --name-only --pretty=format:'' 9f585d
```

```
README  
lib/test.rb
```

Wenn du die verwendete ACL-Struktur, die von der `get_acl_access_data` Methode zurückgegeben wird, mit den aufgelisteten Dateien in jedem der Commits vergleichst, kannst du feststellen, ob der Benutzer die Berechtigung hat, um alle seine Commits zu 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
```

Du erhältst eine Liste der neuen Commits, die mit `git rev-list` auf deinen Server gepusht werden. Dann stellst du für jeden dieser Commits fest, welche der Dateien geändert werden sollen und stellst sicher, dass der Benutzer, der den Push ausführt, Zugriff auf alle zu ändernden Pfade hat.

Jetzt können deine Benutzer keine Commits mit unstrukturierten Nachrichten oder Dateien außerhalb der erlaubten Pfade pushen.

## Austesten

Wenn du `chmod u+x.git/hooks/update` ausführst, welches die Datei ist in die du deinen Code einfügen solltest, und dann versuchst, ein Commit mit einer nicht konformen Beschreibung zu pushen, dann erhältst du wahrscheinlich das hier:

```
$ 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 sind noch ein paar interessante Details zu finden. Erstens, du siehst, so der Hook gestartet ist.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Denke daran, dass das ganz am Anfang deines Update-Skripts ausgegeben hast. Alles, was dein Skript an `stdout` weitergibt, wird an den Client übertragen.

Das nächste, was du beachten solltest, ist die Fehlermeldung.

```
[POLICY] Your message is not formatted correctly
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

Die erste Zeile wurde von dir, die anderen beiden wurden von Git ausgegeben. Die teilt dir mit, dass das Update-Skript ungleich Null beendet wurde. Das ist der Grund, warum dein Push abgelehnt wurde. Zum Schluss noch Folgendes:

```
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Du erhältst eine Remote-Ablehnungs-Nachricht für jede Referenz, die von deinem Hook abgelehnt wurde. Sie zeigt dir an, dass sie genau wegen diesem Hook-Fehlers abgelehnt wurde.

Wenn außerdem jemand versucht, eine Datei, auf die er keinen Zugriff hat, zu bearbeiten und einen Commit damit pusht, dann wird er etwas Ähnliches sehen. Versucht ein Dokumentations-Author zum Beispiel, einen Commit zu pushen, indem er etwas im `lib` Verzeichnis ändert, wird ihm folgendes angezeigt:

```
[POLICY] You do not have access to push to lib/test.rb
```

Von jetzt an, solange dieses `update` Skript verfügbar und ausführbar ist, wird dein Repository nie eine Commit-Beschreibung ohne dein eigenes Schema haben, und deine Benutzer werden in einer „Sandbox“ untergebracht sein.

## Clientseitige Hooks

Der Nachteil dieses Konzepts ist das Gejammer, das unweigerlich entsteht, wenn die Commit-Pushes deiner Benutzer abgelehnt werden. Die Tatsache, dass die sorgfältig gestalteten Arbeiten in letzter Minute abgelehnt werden, kann äußerst frustrierend und irritierend sein. Darüber hinaus müssen sie ihre Verlaufsdaten bearbeiten, um sie zu korrigieren, was nicht unbedingt etwas für schwache Nerven ist.

Die Antwort auf dieses Dilemma ist, einige clientseitige Hooks bereitzustellen, die Benutzer ausführen können, um sie darüber zu informieren, dass sie etwas unternehmen, das der Server wahrscheinlich ablehnen wird. Auf diese Weise können sie mögliche Probleme vor dem Commit klären, bevor es schwieriger wird sie zu beheben. Da Hooks nicht mit einem Klon eines Projekts übertragen werden, musst du diese Skripte auf andere Weise bereitstellen. Dann müssen deine Benutzer sie in ihr `.git/hooks` Verzeichnis kopieren und sie ausführbar machen. Du kannst diese Hooks innerhalb des Projekts oder in einem separaten Projekt weitergeben, aber Git wird sie nicht automatisch einrichten.

Am Anfang solltest du deine Commit-Beschreibung kurz vor jeder Übertragung überprüfen, damit du dir sicher bist, dass der Server deine Änderungen nicht aufgrund schlecht formatierter Commit-Beschreibungen ablehnt. Dazu kannst du den `commit-msg` Hook hinzufügen. Wenn du die Nachricht aus der als erstes Argument übergebenen Datei liest und mit dem Patternmuster vergleichst, kannst du Git zwingen, die Übertragung abzubrechen, wenn es keine Übereinstimmung gibt:

```
#!/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
```

Wenn dieses Skript (in `.git/hooks/commit-msg`) vorhanden und ausführbar ist und du mit einer Nachricht committest, die nicht korrekt formatiert ist, siehst du das hier:

```
$ git commit -am 'Test'  
[POLICY] Your message is not formatted correctly
```

In diesem Fall wurde kein Commit durchgeführt. Wenn deine Nachricht jedoch das richtige Muster enthält, erlaubt dir Git die Übertragung:

```
$ git commit -am 'Test [ref: 132]'  
[master e05c914] Test [ref: 132]  
1 file changed, 1 insertions(+), 0 deletions(-)
```

Als nächstes solltest du sicherstellen, dass du keine Dateien änderst, die sich außerhalb deines ACL-Bereichs befinden. Wenn das `.git` Verzeichnis deines Projekts eine Kopie der ACL-Datei enthält, die du zuvor verwendet hast, dann wird das folgende `pre-commit` Skript diese Einschränkungen für dich erzwingen:

```
#!/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
```

Das ist ungefähr das gleiche Skript wie der serverseitige Teil, allerdings mit zwei wichtigen Unterschieden. Erstens befindet sich die ACL-Datei an einer anderen Stelle, da dieses Skript aus deinem Arbeitsverzeichnis und nicht aus deinem `.git` Verzeichnis läuft. Du musst den Pfad zur ACL-Datei ändern, von:

```
access = get_acl_access_data('acl')
```

zu:

```
access = get_acl_access_data('.git/acl')
```

Der andere wichtige Unterschied ist die Art und Weise, wie du eine Liste der Dateien erhältst, die geändert wurden. Da die serverseitige Methode das Log des Commits betrachtet und der Commit an dieser Stelle noch nicht aufgezeichnet wurde, musst du stattdessen deine Dateiliste aus der Staging-Area holen. Anstelle von:

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

musst du Folgendes benutzen:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Aber das sind die einzigen beiden Unterschiede – ansonsten funktioniert das Skript wie gehabt. Ein Nachteil ist, dass es erwartet, dass du lokal mit dem gleichen Benutzer arbeitest, den du auf dem Remotesystem verwendest. Wenn das anders ist, musst du die Variable `$user` manuell setzen.

Außerdem können wir hier sicherstellen, dass der Benutzer keine „non-fast-forwarded“ Referenzen pusht. Um eine Referenz zu erhalten, die kein „fast-forward“ ist, musst du entweder über einen Commit hinaus rebasen, den du bereits hochgeladen hast oder versuchen, einen anderen lokalen Branch auf den gleichen Remote-Branch zu pushen.

Wahrscheinlich ist der Server bereits mit `receive.denyDeletes` und `receive.denyNonFastForwards` konfiguriert, um diese Richtlinie zu erzwingen. Somit ist die einzige unbeabsichtigte Aktion, die du abfangen kannst ein Rebase-Commit, welcher bereits gepusht wurde.

Hier folgt ein Beispiel für ein Pre-Rebase-Skript, das das überprüft. Es erhält eine Liste aller Commits, die du gerade neu schreiben willst, und prüft, ob sie in einer deiner Remote-Referenzen vorhanden sind. Wenn es einen findet, der von einer deinen Remote-Referenzen aus erreichbar ist, bricht es den Rebase ab.

```
#!/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

```

Dieses Skript verwendet eine Syntax, die in Kapitel 7, [Revisions-Auswahl](#) nicht behandelt wurde. Du erhältst eine Liste der Commits, die bereits gepusht wurden, wenn du diese Anweisung aufrufst:

```
'git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}'
```

Die `SHA^@` Syntax löst alle Vorgänger (engl. parents) dieses Commits auf. Du suchst nach jedem Commit, der vom letzten Commit auf dem Remote aus erreichbar ist und der von keinem Parent einer der SHA-1s, die du hochladen möchten, erreichbar ist – was bedeutet, dass es ein „fast-forward“ ist.

Der größte Nachteil dieses Ansatzes ist, dass er sehr langsam sein kann und oft unnötig ist. Wenn du nicht versuchst, den Push mit `-f` zu erzwingen, wird der Server dich warnen und den Push nicht akzeptieren. Es ist aber ein interessanter Test und kann dir theoretisch helfen, einen Rebase zu vermeiden, den du später vielleicht wieder zurücknehmen und reparieren musst.

## Zusammenfassung

Wir haben die meisten der wichtigen Möglichkeiten besprochen, mit denen du deinen Git-Client und -Server so anpassen kannst, dass er am besten zu deinem Workflow und deinem Projekten passt. Du hast die verschiedensten Konfigurations-Einstellungen, dateibasierte Attribute und Ereignis-Hooks kennengelernt und einen Beispiel-Server zur Einhaltung von Regeln erstellt. In der Regel kannst du Git nun an nahezu jeden Workflow anpassen, den du dir vorstellen kannst.

# Git und andere VCS-Systeme

Die Welt ist nicht perfekt. Normalerweise kannst du nicht jedes Projekt, mit dem du arbeitest, sofort auf Git umstellen. Manchmal steckt man in einem Projekt mit einem anderen VCS fest und wünscht sich, man könnte zu Git wechseln. Wir werden im ersten Teil dieses Kapitels die Möglichkeiten kennenlernen, Git als Client zu verwenden, falls das Projekt, an dem du gerade arbeitest, ein anderes System nutzt.

Irgendwann wirst du vielleicht dein bestehendes Projekt nach Git migrieren wollen. Der zweite Teil dieses Kapitels behandelt die Migration deines Projekts zu Git aus verschiedenen Systemen sowie eine funktionierende Methode, wenn kein vorgefertigtes Import-Tool vorhanden ist.

## Git als Client

Git bietet Entwicklern eine so reizvolle Umgebung, dass viele Anwender schon herausgefunden haben, wie man es auf den Arbeitsplätzen nutzen kann, auch wenn der Rest des Teams ein völlig anderes VCS einsetzt. Es gibt eine Vielzahl dieser Schnittstellen, die sogenannten „Brücken“. Hier werden wir die vorstellen, denen du am ehesten in der „freien Wildbahn“ begegnen wirst.

## Git und Subversion

Ein großer Teil der Open-Source-Entwicklungsprojekte und eine ganze Reihe von Unternehmensprojekten nutzen Subversion zur Verwaltung ihres Quellcodes. Es gibt Subversion seit mehr als einem Jahrzehnt, und die meiste Zeit war es *erste Wahl* für ein VCS im Bereich Open-Source-Projekte. Es ist auch in vielen Aspekten sehr ähnlich zu CVS, das vorher der wichtigste Vertreter der Versionsverwaltung war.

Eines der herausragenden Merkmale von Git ist die bidirektionale Brücke zu Subversion, genannt `git svn`. Mit diesem Tool kannst du Git als geeigneter Client für einen Subversion-Server verwenden, so dass du alle lokalen Funktionen von Git nutzen und dann auf einen Subversion-Server pushen kannst, als ob du Subversion lokal einsetzen würdest. Das bedeutet, dass du lokale Branching- und Merging-Aktivitäten vornimmst, die Staging-Area nutzt, Rebasing- und Cherry-Picking-Aktivitäten durchführen kannst, während deine Mitstreiter weiterhin in ihrer dunklen und altertümlichen SVN-Umgebung tätig sind. Es ist eine gute Möglichkeit, Git in die Unternehmensumgebung einzuschleusen, deine Entwicklerkollegen zu helfen, effizienter zu werden und gleichzeitig die Infrastruktur so zu ändern, dass Git vollständig unterstützt wird. Die Subversion-Brücke ist das Portal zur D(istributed)-VCS-Welt.

### `git svn`

Der Hauptbefehl in Git für sämtliches Subversion-Bridging ist `git svn`. Es sind ziemlich wenige Befehle erforderlich, so dass wir die gängigsten aufzeigen und dabei einige einfache Workflows durchgehen werden.

Es ist wichtig zu beachten, dass du bei der Verwendung von `git svn` mit Subversion interagierst. SVN ist ein System, das ganz anders funktioniert als Git. Obwohl du lokales Branching und Merging durchführen **kannst**, ist es im Allgemeinen ratsam, deinen Verlauf so linear wie möglich zu gestalten, indem du deine Arbeiten rebased. Vermeide dabei die gleichzeitige Interaktion mit einem

## Git Remote-Repository

Schreibe deinen Verlauf nicht um und versuche nicht, diesen geänderten Verlauf erneut zu pushen. Pushe nicht in ein paralleles Git-Repository, um gleichzeitig mit anderen Git-Entwicklern zusammenzuarbeiten. Subversion kann nur einen einzigen linearen Verlauf haben und es ist sehr einfach, diesen Verlauf durcheinanderzubringen. Wenn du mit einem Team arbeitest und einige verwenden SVN und andere Git, stelle sicher, dass alle den SVN-Server für die gemeinsame Arbeit verwenden – das erleichtert dir deinen Alltag.

## Einrichtung

Um diese Funktionalität zu demonstrieren, benötigst du ein SVN-Repository mit Schreibzugriff. Wenn du die Beispiele kopieren möchtest, musst du eine beschreibbare Kopie eines SVN-Test Repository erstellen. Um das zu tun, kannst du das Tool `svnsync` verwenden, das in einer Subversion-Installation enthalten ist.

Um dem Beispielen zu folgen, musst du zunächst ein neues lokales Subversion-Repository erstellen:

```
$ mkdir /tmp/test-svn  
$ svnadmin create /tmp/test-svn
```

Aktiviere dann alle Benutzer, um revprops zu ändern. Der einfachste Weg ist, ein `pre-revprop-change` Skript hinzuzufügen, das immer den exit-Wert 0 hat:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change  
#!/bin/sh  
exit 0;  
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Du kannst dieses Projekt nun auf deinem lokalen Rechner synchronisieren, indem du `svnsync init` mit den Repositorys to und from aufrufst.

```
$ svnsync init file:///tmp/test-svn \  
http://your-svn-server.example.org/svn/
```

Dadurch werden die Eigenschaften für die Ausführung der Synchronisierung festgelegt. Anschliessend kannst du den Code klonen, indem du Folgendes ausführst:

```
$ 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.  
[...]
```

Obwohl dieser Vorgang nur wenige Minuten in Anspruch nehmen könnte, dauert der Prozess fast eine Stunde, wenn du versuchen solltest, das ursprüngliche Repository in ein anderes Remote-Repository anstelle eines lokalen zu kopieren, obwohl es weniger als 100 Commits gibt. Subversion muss eine Revision nach der anderen klonen und sie dann wieder in ein anderes Repository verschieben. Es ist äußerst ineffizient, aber es ist der einzige einfache Weg, das zu erreichen.

## Erste Schritte

Jetzt, da du ein beschreibbares Subversion-Repository hast, kannst du einen normalen Workflow nutzen. Beginne mit dem Befehl `git svn clone`, der ein komplettes Subversion-Repository in ein lokales Git-Repository importiert. Beachte, dass du beim Import aus einem echten Subversion-Repository `file:///tmp/test-svn` durch die URL deines Subversion-Repositorys ersetzen musst:

```
$ 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
```

Das entspricht zwei Befehlen – `git svn init` gefolgt von `git svn fetch` – auf der von dir angegebenen URL. Dieser Vorgang kann einige Zeit dauern. Wenn beispielsweise das Testprojekt nur etwa 75 Commits hat und die Code-Basis nicht so groß ist, muss Git dennoch jede Version einzeln auschecken und einzeln committen. Bei einem Projekt mit Hunderten oder Tausenden von Commits kann es Stunden oder gar Tage dauern.

Der Teil `-T trunk -b branches -t tags` teilt Git mit, dass dieses Subversion-Repository den grundlegenden Branching- und Tagging-Konventionen folgt. Wenn du deinen Trunk, deine Branches oder Tags anders benennst, können sich diese Optionen ändern. Da dies so häufig vorkommt, kannst du den gesamten Teil durch `-s` ersetzen, was Standardlayout bedeutet und all diese Optionen beinhaltet. Das folgende Kommando ist dabei gleichwertig:

```
$ git svn clone file:///tmp/test-svn -s
```

An dieser Stelle solltest du über ein valides Git-Repository verfügen, das deine Branches und Tags importiert hat:

```
$ 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
```

Beachte, dass dieses Tool Subversion-Tags als Remote-Referenzen (engl. refs) verwaltet. Werfen wir einen genaueren Blick auf den Git Low-Level-Befehl `show-ref`:

```
$ git show-ref
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/heads/master
0fb585761df569eaecd8146c71e58d70147460a2 refs/remotes/origin/my-calc-branch
bfd2d79303166789fc73af4046651a4b35c12f0b refs/remotes/origin/tags/2.0.2
285c2b2e36e467dd4d91c8e3c0c0e1750b3fe8ca refs/remotes/origin/tags/release-2.0.1
cbda99cb45d9abcb9793db1d4f70ae562a969f1e refs/remotes/origin/tags/release-2.0.2
a9f074aa89e826d6f9d30808ce5ae3ffe711feda refs/remotes/origin/tags/release-2.0.2rc1
556a3e1e7ad1fde0a32823fc7e4d046bcfd86dae refs/remotes/origin/trunk
```

Git macht das nicht, wenn man von einem Git-Server geklont hat. So sieht ein Repository mit Tags nach einem frischen Klon aus:

```
$ 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 die Tags direkt in `refs/tags`, anstatt sie mit entfernten Branches zu verknüpfen.

## Zurück zu Subversion committen

Jetzt, da du ein Arbeitsverzeichnis hast, kannst du an dem Projekt arbeiten und deine Commits wieder zum Upstream pushen, indem du Git als SVN-Client verwendest. Wenn du eine der Dateien bearbeitest und überträgst, hast du einen Commit, der in Git lokal existiert aber nicht auf dem Subversion-Server vorhanden ist:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 4af61fd] Adding git-svn instructions to the README
 1 file changed, 5 insertions(+)
```

Als nächstes musst du deine Änderung zum Upstream pushen. Beachte, wie sich dies auf deine

Arbeitsweise mit Subversion auswirkt. Du kannst mehrere Commits offline durchführen und diese dann alle auf einmal auf den Subversion-Server übertragen. Um zu einem Subversion-Server zu pushen, führe den Befehl `git svn dcommit` aus:

```
$ 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
```

Dies erstellt für jeden Commit, den du auf deinem Subversion Server-Codes gemacht hast, einen eigenen Subversion-Commit und schreibt deinen lokalen Git-Commit um, um einen eindeutigen Identifier einzufügen. Das ist wichtig, weil es bedeutet, dass sich alle SHA-1-Prüfsummen für deine Commits ändern. Aus diesem Grund ist es keine gute Idee, gleichzeitig mit Git-basierten Remotes deines Projekts und einem Subversion-Server zu arbeiten. Wenn du dir den letzten Commit ansiehst, siehst du die neu hinzugefügte `git-svn-id`:

```
$ git log -1
commit 95e0222ba6399739834380eb10afcd73e0670bc5
Author: ben <ben@0b684db3-b064-4277-89d1-21af03df0a68>
Date:   Thu Jul 24 03:08:36 2014 +0000

    Adding git-svn instructions to the README

    git-svn-id: file:///tmp/test-svn/trunk@77 0b684db3-b064-4277-89d1-21af03df0a68
```

Es ist zu beachten, dass die SHA-1-Prüfsumme, die ursprünglich mit `4af61fd` begann, als du die Daten übertragen hast, nun mit `95e0222` beginnt. Wenn du sowohl auf einen Git-Server als auch auf einen Subversion-Server pushen möchtest, musst du zuerst auf den Subversion-Server pushen (`dcommit`), da diese Aktion deine Commit-Daten ändert.

## Neue Änderungen pullen

Wenn du mit anderen Entwicklern zusammenarbeitest, wird irgendwann einer von euch pushen, und andere versuchen, eine Änderung voranzutreiben, die Konflikte verursacht. Diese Änderung wird abgelehnt, bis du deren Arbeit mergst. In `git svn` sieht das so aus:

```
$ 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.
```

Um diese Situation zu lösen, kannst du `git svn rebase` ausführen, das alle Änderungen auf dem Server, die du noch nicht hast, pullt und alle deine lokalen Arbeiten auf den gepulnten Code rebased:

```
$ 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.
```

Jetzt ist deine gesamte Arbeit auf den Code des Subversion-Servers rebased, so dass du `dcommit` erfolgreich einsetzen kannst:

```
$ 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
```

Im Unterschied zu Git, das voraussetzt, dass du Upstream-Arbeiten, die du noch nicht lokal hast, zuerst mergst, bevor du pushen kannst, zwingt dich `git svn` dazu nur dann, wenn die Änderungen im Konflikt stehen (ähnlich wie bei Subversion). Wenn jemand anderes eine Änderung an einer Datei macht und du eine Änderung an einer anderen Datei machst, funktioniert dein `dcommit` gut:

```
$ 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: a0253d06732169107aa020390d9fef2b1d92806 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...
```

Es ist sehr wichtig, sich daran zu halten, denn das Ergebnis ist ein Projektstatus, der beim Push auf keinem deiner Computer vorhanden war. Wenn die Änderungen inkompatibel sind, aber keine Konflikte verursachen, können Probleme auftreten, die schwer zu diagnostizieren sind. Das ist ein Unterschied gegenüber der Nutzung eines Git-Servers. In Git kannst du den Zustand auf deinem Client-System vor der Veröffentlichung vollständig testen, während du in SVN nie sicher sein kannst, dass die Zustände unmittelbar vor dem Commit und nach dem Commit identisch sind.

Du solltest diesen Befehl auch ausführen, um Änderungen vom Subversion-Server einzubinden, auch wenn du noch nicht bereit bist, selbst zu committen. Es ist ratsam, `git svn fetch` auszuführen, um die neuen Daten zu holen, aber `git svn rebase` übernimmt bereits den Fetch und aktualisiert dann deine lokalen Commits.

```
$ git svn rebase
    M  autogen.sh
r88 = c9c5f83c64bd755368784b444bc7a0216cc1e17b (refs/remotes/origin/trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/origin/trunk.
```

Wenn du ab und zu `git svn rebase` laufen lässt, stellst du sicher, dass dein Code immer auf dem neuesten Stand ist. Du solltest jedoch überprüfen, ob dein Arbeitsverzeichnis sauber ist, wenn du diese Funktion ausführst. Wenn du lokale Änderungen hast, musst du deine Arbeit entweder verstecken (engl. stash) oder temporär committen, bevor du `git svn rebase` ausführst. Andernfalls wird der Befehl angehalten, wenn er erkennt, dass das Rebase zu einem Merge-Konflikt führen wird.

## Git Branching Probleme

Sobald du dich mit einem Git-Workflow vertraut gemacht hast, wirst du wahrscheinlich Topic-Banches erstellen, an ihnen arbeiten und sie dann mergen wollen. Wenn du über `git svn` auf einen Subversion-Server pushst, kannst du deine Arbeit jedes Mal auf einen einzigen Branch rebasen, anstatt Branches zu mergen. Die Begründung für ein Rebasing ist, dass Subversion eine lineare Historie hat und sich nicht wie Git mit Merges beschäftigt. So folgt `git svn` bei der Konvertierung der Snapshots in Subversion Commits nur dem ersten Elternteil.

Nehmen wir an, dein Verlauf sieht wie folgt aus: Du hast einen `experiment` Branch erstellt, zwei Commits durchgeführt und diese dann wieder mit dem `master` zusammengeführt. Wenn du `dcommit` aufrufst, erscheint folgende Anzeige:

```
$ 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
```

Das Ausführen von `dcommit` auf einem Branch mit zusammengeführtem Verlauf funktioniert gut, außer wenn du dir dein Git-Projekt-Historie ansiehst. Es wurde keiner der Commits, die du auf dem `experiment` Branch gemacht hast, neu geschrieben – statt dessen erscheinen alle diese Änderungen in der SVN-Version eines einzelnen Merge-Commits.

Wenn jemand anderes diese Arbeit klont, sieht man nur den Merge-Commit mit der gesamten Arbeit, die in ihn zusammengeführt wurde, als ob du `git merge --squash` ausgeführt hättest. Man sieht die Commit-Daten nicht, weder woher sie stammen noch wann sie committed wurden.

## Subversion Branching

Branching in Subversion ist nicht dasselbe wie Branching in Git. Es ist wahrscheinlich das Beste, wenn du es so gut es geht vermeidest. Du kannst aber mit `git svn` in Subversion Branches anlegen und dorthin committen.

### Erstellen eines neuen SVN Branches

Um einen neuen Branch in Subversion zu erstellen, führe `git svn branch [new-branch]` aus:

```
$ 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)
```

Dadurch wird das Äquivalent des Befehls `svn copy trunk branches/opera` in Subversion ausgeführt und auf dem Subversion-Server angewendet. Es ist wichtig zu beachten, dass dieser Branch nicht ausgecheckt wird. Wenn du an diesem Punkt einen Commit durchführst, geht dieser Commit in den `trunk` auf dem Server, nicht in `opera`.

## Aktive Branches wechseln

Git findet heraus, in welchen Branch deine dcommits hingehen, indem es nach dem Ende eines deiner Subversion Branches in deinem Verlauf sucht – du solltest nur einen haben, und es sollte der Letzte sein, der eine `git-svn-id` in deinem aktuellen Branchverlauf hat.

Wenn du an mehr als einem Branch gleichzeitig arbeiten möchtest, kannst du lokale Branches so einrichten, dass du `dcommit` für bestimmte Subversion-Banches ausführst, indem du diese beim importierten Subversion-Commit für diesen Branch startest. Einen `opera` Branch, mit dem du separat arbeiten kannst, kannst du folgendermaßen initialisieren:

```
$ git branch opera remotes/origin/opera
```

Um deinen `opera` Branch in `trunk` (deinen `master` Branch) zu mergen, kannst du das mit einem normalen `git merge` machen. Aber du solltest unbedingt eine beschreibende Commit-Meldung (via `-m`) angeben, sonst wird beim Merge anstelle von etwas Vernünftigem „Merge branch `opera`“ angezeigt.

Obwohl du für diese Operation `git merge` verwendest und der Merge wahrscheinlich viel einfacher ist als in Subversion (da Git automatisch die entsprechende Merge-Basis für dich erkennt), ist es kein normaler Git Merge-Commit. Du musst diese Daten an einen Subversion-Server zurück pushen, der keinen Commit mit mehr als einem Elternteil verarbeiten kann. Nachdem du ihn zum Server gepusht hast, sieht er also aus wie ein einzelner Commit, der die gesamte Arbeit eines anderen Branchs unter einem einzigen Commit zusammenfasst. Nachdem du einen Branch in einem anderen zusammengeführt haben, kannst du nicht einfach zurückgehen und an diesem Branch weiterarbeiten, wie du es normalerweise in Git tust. Das `dcommit` Kommando, das du ausführst, löscht alle Informationen, die zeigen, in welchen Branch zusammengeführt wurde, so dass nachfolgende Berechnungen der Merge-Basis falsch sind – `dcommit` lässt dein `git merge` Ergebnis aussehen, als ob du `git merge --squash` ausgeführt hättest. Leider gibt es keine gute Methode, diese Situation zu vermeiden – Subversion kann diese Informationen nicht speichern, daher wirst du immer von den Einschränkungen des Systems behindert, während du es als deinen Server verwendest. Um Fehler zu vermeiden, solltest du den lokalen Branch (in diesem Fall `opera`) löschen, nachdem du ihn in `trunk` eingefügt hast.

## Subversion Kommandos

Das `git svn` Toolset bietet eine Reihe von Befehlen, die den Übergang zu Git erleichtern, indem es einige Funktionen bereitstellt, die denen ähneln, die du aus Subversion kennst. Wir haben hier ein paar Befehle, mit denen du das bekommst, was Subversion vorher konnte.

## Verlauf im SVN-Format

Wenn du an Subversion gewöhnt bist und deinen Verlauf im SVN-Stil sehen möchtest, kannst du `git`

`svn log` ausführen, um deinen Commit-Verlauf in SVN-Formatierung anzuzeigen:

```
$ 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
```

Du solltest zwei wichtige Dinge über `git svn log` wissen. Erstens funktioniert es offline, im Unterschied zum echten `svn log` Befehl, der den Subversion-Server nach den Daten fragt. Zweitens zeigt es dir nur Commits an, die zum Subversion-Server übertragen wurden. Lokale Git-Commits, die du noch nicht mit `dcommit` bestätigt hast, werden nicht angezeigt; ebenso wenig wie Commits, die von Leuten in der Zwischenzeit auf dem Subversion-Server gemacht wurden. Es ist mehr wie der letzte bekannte Zustand der Commits auf dem Subversion-Server.

### SVN Annotation

So wie der Befehl `git svn log` den Befehl `svn log` offline simuliert, kannst du das Äquivalent von `svn annotate` abrufen, indem du `git svn blame [FILE]` ausführst. Die Ausgabe sieht wie folgt aus:

```
$ 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 temporal Committing in git-svn.
78 temporal
2 temporal To build and install the C++ Protocol Buffer runtime and the Protocol
2 temporal Buffer compiler (protoc) execute the following:
2 temporal
```

Nochmal: Auch hier werden keine Commits angezeigt, die du lokal in Git gemacht hast oder die in der Zwischenzeit in Subversion verschoben wurden.

## SVN Server-Information

Wenn du `git svn info` ausführst, kannst du die gleiche Art von Informationen erhalten, die dir `svn info` liefert:

```
$ 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)
```

Das ist wie bei `blame` und `log`, denn es läuft offline und ist nur ab der letzten Kommunikation mit dem Subversion-Server auf dem neuesten Stand.

## Ignorieren, was Subversion ignoriert

Wenn du ein Subversion-Repository klonst, in dem irgendwo `svn:ignore` Eigenschaften gesetzt sind, wirst du wahrscheinlich entsprechende `.gitignore` Dateien setzen wollen, damit du nicht versehentlich Dateien überträgst, die nicht übertragen werden sollen. `git svn` verfügt über zwei Befehle, um bei diesem Problem zu helfen. Der Erste ist `git svn create-ignore`, der automatisch entsprechende `.gitignore` Dateien für dich erstellt, damit sie bei deinem nächsten Commit berücksichtigt werden.

Der zweite Befehl ist `git svn show-ignore`, der die Zeilen nach stdout ausgibt, die du in eine `.gitignore` Datei einfügen musst, damit du die Ausgabe in die Ausschlussdatei deines Projekts umleiten kannst:

```
$ git svn show-ignore > .git/info/exclude
```

Auf diese Weise überhäufst du das Projekt nicht mit `.gitignore` Dateien. Das ist eine gute Option, wenn du der einzige Git-Benutzer in einem Subversion-Team bist und deine Teamkollegen keine `.gitignore` Dateien im Projekt haben wollen.

## git svn Zusammenfassung

Die `git svn` Tools sind nützlich, wenn du gezwungen werden musst mit einem Subversion-Server arbeiten. Du solltest es jedoch als verkapptes Git betrachten, oder du wirst Probleme in der Umsetzung haben, die dich und deine Mitstreiter verwirren könnten. Um keine Schwierigkeiten zu bekommen, versuche dich an diese Hinweise zu halten:

- Führe einen linearen Git-Verlauf, der keine Merge-Commits von `git merge` enthält. Rebasing alle Arbeiten, die du außerhalb deines Haupt-Branchs durchführst, wieder in diesen ein; merge sie

nicht.

- Richte keinen separaten Git-Server ein und arbeite nicht mit einem zusammen. Möglicherweise hast du einen, um Klone für neue Entwickler zu starten, aber pushe nichts, was nicht über einen `git-svn-id` Eintrag verfügt. Du kannst eventuell einen `pre-receive` Hook hinzufügen, der jede Commit-Nachricht auf einen `git-svn-id` überprüft und Pushes, die Commits ohne ihn enthalten, ablehnt.

Wenn du diese Leitlinien befolgst, kann die Arbeit mit einem Subversion-Server leichter umsetzbar sein. Mit einem Umstieg auf einen echten Git-Server kann dein Team erheblich mehr an Effizienz gewinnen.

## Git und Mercurial

Das DVCS-Universum besteht nicht nur aus nur Git. In diesem Bereich gibt es viele andere Systeme, jedes hat seinen eigenen Ansatz, wie eine verteilte Versionskontrolle zu funktionieren hat. Neben Git ist Mercurial am populärsten und die beiden sind sich in vielerlei Hinsicht sehr ähnlich.

Die gute Nachricht, wenn du Gits clientseitiges Verhalten bevorzugst, aber mit einem Projekt arbeitest, dessen Quellcode mit Mercurial verwaltet wird, dann ist es möglich, Git als Client für ein von Mercurial gehostetes Repository zu verwenden. Da die Art und Weise, wie Git über Remotes mit Server-Repositories kommuniziert, sollte es nicht überraschen, dass diese Bridge als Remote-Helper implementiert ist. Der Name des Projekts lautet `git-remote-hg` und ist unter <https://github.com/felipec/git-remote-hg> zu finden.

### git-remote-hg

Zuerst musst du `git-remote-hg` installieren. Im Wesentlichen geht es darum, die Datei irgendwo in deinem Pfad abzulegen, so wie hier:

```
$ curl -o ~/bin/git-remote-hg \
  https://raw.githubusercontent.com/felipec/git-remote-hg/master/git-remote-hg
$ chmod +x ~/bin/git-remote-hg
```

...vorausgesetzt (in einer Linux-Umgebung), `~/bin` ist in deinem `$PATH`. `Git-remote-hg` hat noch eine weitere Abhängigkeit: die `mercurial` Library für Python. Wenn du Python schon installiert hast, ist das einfach:

```
$ pip install mercurial
```

Wenn du Python noch nicht installiert hast, besuche <https://www.python.org/> und installiere es.

Als Letztes brauchst du den Mercurial-Client. Gehe zu <https://www.mercurial-scm.org/> und installiere ihn, falls du es noch nicht getan hast.

Jetzt bist du bereit zum abrocken. Alles, was du benötigst, ist ein Mercurial-Repository, auf das du zugreifen kannst. Glücklicherweise kann sich jedes Mercurial-Repository so verhalten. Also verwenden wir einfach das „hello world“-Repository, das jeder benutzt, um Mercurial zu lernen:

```
$ hg clone http://selenic.com/repo/hello /tmp/hello
```

## Erste Schritte

Nun, da wir über ein geeignetes „serverseitiges“ Repository verfügen, können wir einen typischen Workflow durchlaufen. Wie du sehen wirst, sind diese beiden Systeme ähnlich genug, dass es keine großen Abweichungen gibt.

Wie immer mit Git, wir klonen zuerst:

```
$ 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
```

Wie du siehst, verwendet man bei der Arbeit mit einem Mercurial-Repository den Standardbefehl `git clone`. Das liegt daran, dass `git-remote-hg` auf einem relativ niedrigen Level arbeitet und einen ähnlichen Mechanismus verwendet, wie es die Implementierung des HTTP/S-Protokolls in Git ist (Remote-Helper). Da Git und Mercurial beide so konzipiert sind, dass jeder Client eine vollständige Kopie der Repository-Historie hat, erstellt dieser Befehl relativ schnell einen vollständigen Klon, einschließlich der gesamten Projekthistorie.

Der `log`-Befehl zeigt zwei Commits, von denen der letzte von einer ganzen Reihe von Refs angeführt wird. Wie sich herausstellt, sind einige davon nicht wirklich da. Werfen wir einen Blick darauf, was sich wirklich im `.git` Verzeichnis befindet:

```
$ 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 versucht sich idiomatisch (begrifflich) an Git anzunähern, aber im Hintergrund verwaltet es die konzeptionelle Zuordnung zwischen zwei leicht unterschiedlichen Systemen. Im Verzeichnis `refs/hg` werden die aktuellen Remote-Referenzen gespeichert. Zum Beispiel ist die `refs/hg/origin/branches/default` eine Git ref-Datei, die das SHA-1 enthält und mit „ac7955c“ beginnt. Das ist der Commit, auf den `master` zeigt. Das Verzeichnis `refs/hg` ist also eine Art gefälschtes `refs/remotes/origin`, aber es unterscheidet zusätzlich zwischen Lesezeichen und Branches.

Die Datei `notes/hg` ist der Ausgangspunkt dafür, wie git-remote-hg Git-Commit-Hashes auf Mercurial-Changeset-IDs abbildet. Lass uns ein wenig experimentieren:

```
$ 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
```

So zeigt `refs/notes/hg` auf einen Verzeichnisbaum, der in der Git-Objektdatenbank eine Liste anderer Objekte mit Namen ist. `git ls-tree` gibt Modus, Typ, Objekt-Hash und Dateiname für Elemente innerhalb eines Baums aus. Sobald wir uns auf eines der Baumelemente festgelegt haben, stellen wir fest, dass sich darin ein „ac9117f“ Blob (der SHA-1-Hash des Commit, auf den `master` zeigt) befindet. Inhaltlich ist er identisch mit „0a04b98“ (das ist die ID des Mercurial-Changesets am Ende des `default` Branch).

Die gute Nachricht ist, dass wir uns darüber meistens keine Sorgen machen müssen. Der typische Arbeitsablauf unterscheidet sich nicht wesentlich von der Arbeit mit einem Git-Remote.

Noch eine Besonderheit, um die wir uns kümmern sollten, bevor wir fortfahren: Das Ignorieren. Mercurial und Git verwenden dafür einen sehr ähnlichen Mechanismus, aber es ist durchaus möglich, dass du eine `.gitignore` Datei nicht wirklich in ein Mercurial Repository übertragen willst. Glücklicherweise hat Git eine Möglichkeit, Dateien zu ignorieren, die lokal in einem On-Disk-Repository liegen. Das Mercurial-Format ist kompatibel mit Git, so dass du sie nur kopieren musst:

```
$ cp .hgignore .git/info/exclude
```

Die Datei `.git/info/exclude` verhält sich wie eine `.gitignore`, wird aber nicht in den Commits aufgenommen.

## Workflow

Nehmen wir an, wir haben einige Arbeiten erledigt und einige Commits auf den `master` Branch gemacht und du bist so weit, ihn in das Remote-Repository zu pushen. Nun sieht unser Repository momentan so aus:

```
$ 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
```

Unser `master` Branch ist zwei Commits vor dem `origin/master`, aber diese beiden Commits existieren nur auf unserem lokalen Rechner. Schauen wir mal nach, ob jemand anderes zur gleichen Zeit wichtige Arbeit geleistet hat:

```
$ 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
```

Da wir das `--all` Flag verwendet haben, sehen wir die „notes“ Refs, die intern von `git-remote-hg` verwendet werden, die wir aber ignorieren können. Den Rest haben wir erwartet. `origin/master` ist um einen Commit fortgeschritten. Unser Verlauf hat sich dadurch verändert. Anders als bei anderen Systemen, mit denen wir in diesem Kapitel arbeiten, ist Mercurial in der Lage, Merges zu verarbeiten, so dass wir nichts Ausgefallenes tun müssen.

```
$ 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
```

Perfekt. Wir führen die Tests durch und alles passt, also sind wir so weit, dass wir unsere Arbeit mit dem Rest des Teams teilen können:

```
$ git push  
To hg::/tmp/hello  
 df85e87..0c64627 master -> master
```

Das wars! Wenn du einen Blick auf das Mercurial-Repository wirfst, wirst du feststellen, dass genau das passiert ist, was wir erwarten haben:

```
$ 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
```

Das Change-Set mit der Nummer 2 wurde von Mercurial vorgenommen, und die Change-Sets mit der Nummer 3 und 4 wurden von git-remote-hg durchgeführt, indem Commits mit Git gepusht wurden.

## Branches und Bookmarks

Git hat nur eine Art von Branch: eine Referenz, die sich verschiebt, wenn Commits gemacht werden. In Mercurial wird diese Art von Referenz als „bookmark“ (dt. Lesezeichen) bezeichnet, und sie verhält sich ähnlich wie ein Git-Branch.

Das Konzept von Mercurial eines „Branchs“ ist schwergewichtiger. Der Branch, auf den ein

Changeseit durchgeführt wird, wird *zusammen mit dem Changeseit* aufgezeichnet, d.h. er befindet sich immer im Repository-Verlauf. Hier ist ein Beispiel für einen Commit, der auf dem **develop** Branch gemacht wurde:

```
$ 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
```

Achte auf die Zeile, die mit „branch“ beginnt. Git kann das nicht wirklich nachahmen (und muss es auch nicht; beide Arten von Branches können als Git ref dargestellt werden), aber git-remote-hg muss den Unterschied erkennen, denn für Mercurial ist er wichtig.

Das Anlegen von Mercurial-Lesezeichen ist so einfach wie das Erstellen von Git-Banches. Auf der Seite vom Git machst du:

```
$ git checkout -b featureA
Switched to a new branch 'featureA'
$ git push origin featureA
To hg::/tmp/hello
 * [new branch]      featureA -> featureA
```

Das ist alles, was es dazu zu sagen gibt. Auf der Mercurial-Seite sieht es dann folgendermaßen aus:

```
$ 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
```

Beachte den neuen [featureA] Tag auf Revision 5. Diese verhalten sich genau wie Git-Branches auf der Git-Seite, mit einer Ausnahme: Du kannst ein Lesezeichen auf der Git-Seite nicht löschen (das ist eine Einschränkung des Remote-Helpers).

Du kannst auch an einem „schwergewichtigen“ Mercurial-Branch arbeiten: Bringe einfach einen Branch in den branches Namensraum:

```
$ 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
```

So sieht das dann auf der Mercurial-Seite aus:

```
$ 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'
[...]
```

Der Branch-Name „permanent“ wurde mit dem Change-Set 7 eingetragen.

Seitens von Git ist die Arbeit mit einem dieser Branch-Stile die gleiche: Einfach auschecken, committen, fetchen, mergen, pullen, und pushen, wie du es normalerweise auch machen würdest. Eine Sache, die du wissen solltest, ist, dass Mercurial das Überschreiben der Historie nicht unterstützt, sondern nur hinzufügt. Das Mercurial-Repository sieht nach einem interaktiven Rebbase und einem Force-Push so aus:

```
$ 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
```

Die Changesets 8, 9 und 10 wurden angelegt und gehören zum **permanent** Branch, aber die alten Changesets sind immer noch vorhanden. Das kann für deine Teamkollegen, die Mercurial verwenden, **sehr verwirrend** sein, also versuche es zu vermeiden.

## Mercurial Zusammenfassung

Git und Mercurial sind sich ähnlich genug, um über die eigene Umgebung hinaus schmerzlos

miteinander zu arbeiten. Wenn du vermeidest, den Verlauf zu ändern, der deinen Computer bereits verlassen hat (was allgemein empfohlen wird), merkst du wahrscheinlich nicht einmal, dass am anderen Ende Mercurial verwendet wird.

## Git und Perforce

Perforce ist ein sehr beliebtes Versionskontrollsyste in Unternehmen. Es existiert seit 1995 und ist damit das älteste in diesem Kapitel behandelte System. Altersbedingt hat das Konzept aus heutiger Sicht einige Einschränkungen. Es geht davon aus, dass du immer mit einem einzigen zentralen Server verbunden bist und nur eine Version auf der lokalen Festplatte gespeichert ist. Sicherlich sind seine Funktionen und Einschränkungen gut für einige spezielle Probleme geeignet, aber es gibt viele Projekte mit Perforce, bei denen Git möglicherweise besser geeignet wäre.

Es gibt zwei Möglichkeiten, wenn du Perforce und Git zusammen nutzen möchtest. Als erstes stellen wir die „Git Fusion“ Bridge des Herstellers von Perforce vor, mit der du Teilbäume deines Perforce-Depots als Read-Write-Git-Repository freigeben kannst. Bei der zweiten handelt es sich um git-p4, eine client-seitige Bridge, mit der du Git als Perforce-Client verwenden kannst, ohne dass der Perforce-Server neu konfiguriert werden muss.

### Git Fusion

Preforce bietet mit Git Fusion ein Produkt (verfügbar unter <https://www.perforce.com/git-fusion>), das einen Perforce-Server mit Git-Repositorys auf der Serverseite synchronisiert.

#### Git Fusion einrichten

Für unsere Beispiele verwenden wir die einfachste Installationsmethode für Git Fusion, indem wir eine virtuelle Maschine herunterladen, auf der der Perforce-Daemon und Git Fusion laufen. Du kannst das Image der virtuellen Maschine von <https://www.perforce.com/downloads/Perforce/20-User> herunterladen. Wenn der Download abgeschlossen ist, importierst du es in deiner bevorzugten Virtualisierungssoftware (wir verwenden VirtualBox).

Beim ersten Start des Rechners wirst du aufgefordert, das Passwort für drei Linux-Benutzer (**root**, **perforce**, **git**) festzulegen und einen Instanznamen anzugeben, mit dem du diese Installation von anderen im selben Netzwerk unterscheiden kannst. Wenn das alles abgeschlossen ist, solltest du folgendes sehen:

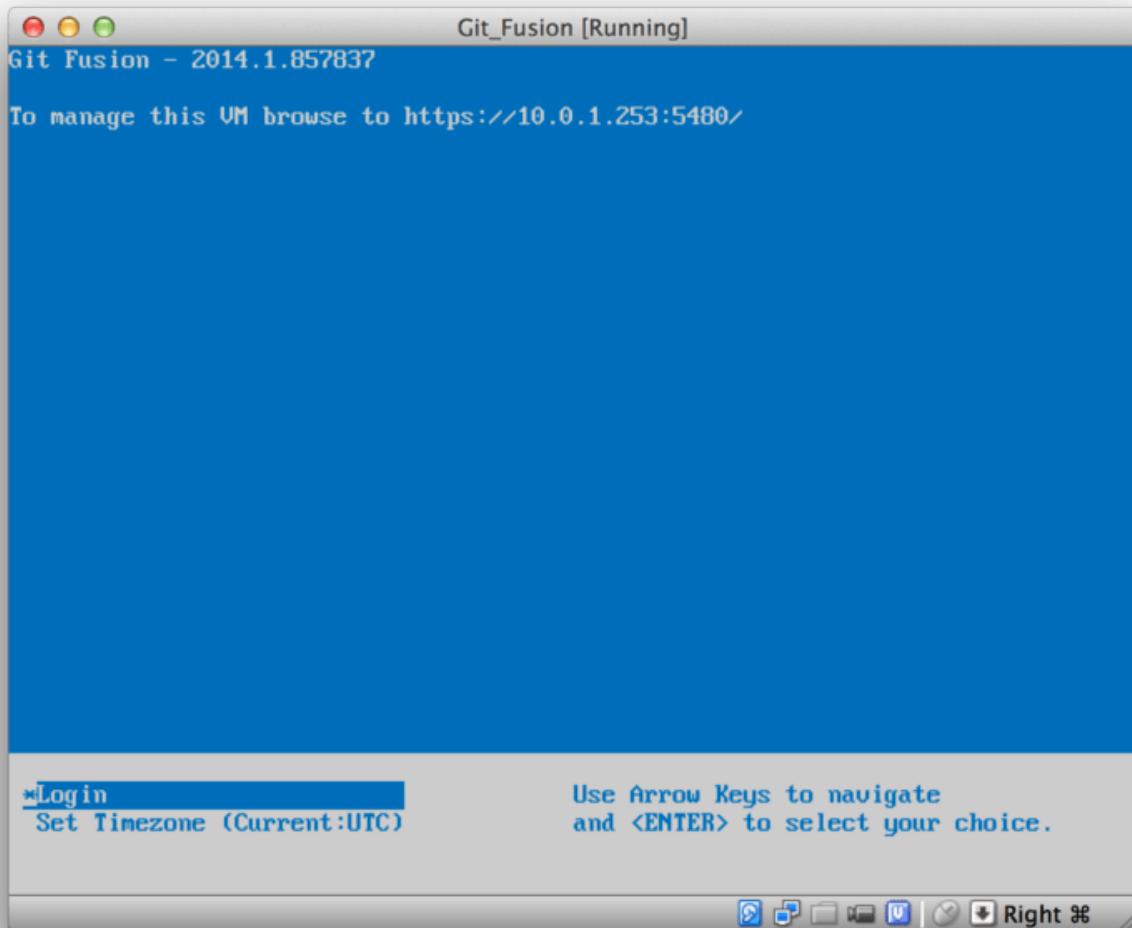


Figure 171. Boot-Bildschirm der virtuellen Maschine von Git Fusion

Du solltest die hier angezeigte IP-Adresse notieren, wir werden sie später benutzen. Als nächstes erstellen wir einen Perforce-Benutzer. Wähle unten die Option „Login“ und drücke die Eingabetaste (oder verbinde dich per SSH mit dem Computer) und melde dich als `root` an. Verwende anschliessend folgenden Befehle, um einen Benutzer anzulegen:

```
$ p4 -p localhost:1666 -u super user -f john  
$ p4 -p localhost:1666 -u john passwd  
$ exit
```

Der erste öffnet einen VI-Editor, um den Benutzer zu personalisieren. Du kannst auch die Vorgaben übernehmen, indem du `:wq` eingibst und auf Enter drückst. Der zweite wird dich auffordern, ein Passwort zweimal einzugeben. Das ist alles, was wir mit einem Shell-Prompt zu tun haben werden, also beende die Sitzung.

Als nächstes musst du Git mitteilen, dass es keine SSL-Zertifikate überprüfen soll. Das Git Fusion-Image wird mit einem Zertifikat geliefert, aber es bezieht sich auf eine Domäne, die nicht mit der IP-Adresse deiner virtuellen Maschine übereinstimmt, weshalb Git die HTTPS-Verbindung abweisen würde. Wenn dies eine permanente Installation sein soll, lies das Handbuch von Perforce

Git Fusion, um ein anderes Zertifikat zu installieren. Für unsere Beispielzwecke genügt diese Angabe:

```
$ export GIT_SSL_NO_VERIFY=true
```

Jetzt können wir testen, ob alles funktioniert.

```
$ 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.
```

Das Virtual-Machine-Image ist mit einem Beispielprojekt ausgestattet, das du klonen kannst. Hier klonen wir über HTTPS, mit dem Benutzer **john**, den wir oben erstellt haben. Git fragt nach Anmeldeinformationen für diese Verbindung, aber der Credential-Cache erlaubt es uns, diesen Schritt für alle nachfolgenden Anfragen zu überspringen.

### Fusion Konfiguration

Sobald du Git Fusion installiert hast, solltest du die Konfiguration anpassen. Mit deinem favorisierten Perforce-Client ist das ganz einfach. Weise das Verzeichnis **//.git-fusion** auf dem Perforce-Server einfach deinem Arbeitsbereich zu. Die Dateistruktur sieht wie folgt aus:

```
$ tree
.
├── objects
│   ├── repos
│   │   └── [...]
│   └── trees
│       └── [...]
|
└── p4gf_config
    ├── repos
    │   └── Talkhouse
    │       └── p4gf_config
    └── users
        └── p4gf_usermap

498 directories, 287 files
```

Das Verzeichnis **objects** wird intern von Git Fusion verwendet, um Perforce-Objekte auf Git

abzubilden und umgekehrt, so dass du es ignorieren kannst. In diesem Verzeichnis gibt es eine globale `p4gf_config` Datei sowie eine für jedes Repository – das sind die Konfigurationsdateien, die das Verhalten von Git Fusion bestimmen. Werfen wir einen Blick auf die Datei im Root:

```
[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
```

Wir werden hier nicht auf die Bedeutung all dieser Flags eingehen. Bedenke jedoch, dass es sich hierbei nur um eine INI-formatierte Textdatei handelt, ähnlich wie bei der Konfiguration mit Git. Diese Datei legt die globalen Optionen fest, die dann von repository-spezifischen Konfigurationsdateien wie `repos/Talkhouse/p4gf_config` überschrieben werden können. Wenn du diese Datei öffnest, siehst du einen Abschnitt `[@repo]` mit einigen Einstellungen, die sich von den globalen Standardeinstellungen unterscheiden. Du wirst auch Abschnitte sehen, die so aussehen:

```
[Talkhouse-master]
git-branch-name = master
view = //depot/Talkhouse/main-dev/... ...
```

Dabei handelt es sich um eine Zuordnung zwischen einem Perforce-Branch und einem Git-Branch. Der Abschnitt kann beliebig benannt werden, solange der Name eindeutig ist. Der `git-branch-name` ermöglicht dir, einen Depot-Pfad, der unter Git umständlich wäre, in einen benutzerfreundlicheren Namen zu konvertieren. Die Anzeige-Einstellung steuert, wie Perforce-Dateien in das Git-Repository mit Hilfe der Standard-Syntax für das View-Mapping abgebildet werden. Es kann mehr als ein Mapping angegeben werden, wie in diesem Beispiel:

```
[multi-project-mapping]
git-branch-name = master
view = //depot/project1/main/... project1/...
      //depot/project2/mainline/... project2/...
```

Wenn dein normales Workspace-Mapping Änderungen in der Struktur der Verzeichnisse enthält, kannst du das auf diese Weise mit einem Git-Repository replizieren.

Die letzte Datei, die wir hier behandeln, ist `users/p4gf_usermap`, die Perforce-Benutzer auf Git-Benutzer abbildet und die du möglicherweise nicht einmal benötigst. Bei der Konvertierung eines Perforce Change-Sets in einen Git Commit sucht Git Fusion standardmäßig nach dem Perforce-Benutzer und verwendet die dort gespeicherte E-Mail-Adresse und den vollständigen Namen für das Autor/Committer-Feld in Git. Bei der umgekehrten Konvertierung wird standardmäßig der Perforce-Benutzer mit der E-Mail-Adresse gesucht, die im Autorenfeld des Git-Commits gespeichert ist, und das Änderungsset als dieser Benutzer übermittelt (mit entsprechenden Berechtigungen). In den meisten Fällen wird dieses Verhalten gut funktionieren, aber beachte die folgende Mapping-Datei:

```
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"
```

Jede Zeile hat das Format `<user> <email> "<full name>"` und erstellt eine einzige Benutzerzuordnung. Die beiden ersten Zeilen ordnen zwei verschiedene E-Mail-Adressen demselben Perforce-Benutzerkonto zu. Das ist praktisch, wenn du Git-Commits unter mehreren verschiedenen E-Mail-Adressen erstellt hast (oder sich deine E-Mail-Adresse ändert), diese aber dem gleichen Perforce-Benutzer zugeordnet werden sollen. Beim Erstellen eines Git-Commits aus einem Perforce Change-Set wird die erste Zeile, die dem Perforce-Benutzer entspricht, für die Angaben zur Git-Autorschaft verwendet.

Die letzten beiden Zeilen überdecken Bob und Joe's tatsächliche Namen und E-Mail-Adressen aus den Git-Commits, die erstellt werden. Das ist sinnvoll, wenn du ein internes Projekt open-source-fähig machen willst, aber dein Mitarbeiterverzeichnis nicht auf der ganzen Welt veröffentlichen willst. Beachte, dass die E-Mail-Adressen und vollständigen Namen eindeutig sein sollten, es sei denn, du möchtest alle Git-Commits einem einzigen fiktiven Autor zuordnen.

### Workflow (Arbeitsablauf)

Perforce Git Fusion ist eine bidirektionale Brücke zwischen der Perforce- und Git-Versionskontrolle. Betrachten wir die Arbeit von der Git-Seite aus. Wir gehen davon aus, dass wir im Projekt „Jam“ mit einer oben gezeigten Konfigurationsdatei gemapped sind, die wir so klonen können:

```
$ git clone https://10.0.1.254/Jam
Cloning into 'Jam'...
Username for 'https://10.0.1.254': john
Password for 'https://john@10.0.1.254':
```

```

remote: Counting objects: 2070, done.
remote: Compressing objects: 100% (1704/1704), done.
Receiving objects: 100% (2070/2070), 1.21 MiB | 0 bytes/s, done.
remote: Total 2070 (delta 1242), reused 0 (delta 0)
Resolving deltas: 100% (1242/1242), done.
Checking connectivity... done.
$ git branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/rel2.1
$ git log --oneline --decorate --graph --all
* 0a38c33 (origin/rel2.1) Create Jam 2.1 release branch.
| * d254865 (HEAD, origin/master, origin/HEAD, master) Upgrade to latest metrowerks on Beos -- the Intel one.
| * bd2f54a Put in fix for jam's NT handle leak.
| * c0f29e7 Fix URL in a jam doc
| * cc644ac Radstone's lynx port.
[...]

```

Wenn du das zum ersten Mal machst, kann es einige Zeit in Anspruch nehmen. Git Fusion konvertiert alle anwendbaren Changesets in der Perforce-Historie in Git-Commits. Das passiert lokal auf dem Server, ist also relativ schnell, aber wenn man einen langen Verlauf hat, kann es trotzdem einige Zeit dauern. Nachfolgende Fetches führen eine inkrementelle Konvertierung durch, so dass es sich schon eher wie die native Geschwindigkeit von Git anfühlt.

Wie du siehst, sieht unser Repository genauso aus wie jedes andere Git-Repository, mit dem du arbeiten kannst. Es gibt drei Branches. Git hat einen lokalen `master` Branch erstellt, der `origin/master` tracked. Wir werden ein wenig arbeiten und ein paar neue Commits erstellen:

```

# ...
$ 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.
[...]

```

Wir haben zwei neue Commits. Nun lass uns überprüfen, ob jemand anderes auch daran gearbeitet hat:

```

$ 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.
[...]

```

Anscheinend hat jemand das tatsächlich getan! Du würdest es aus dieser Sicht nicht erkennen, aber der **6afeb15** Commit wurde mit einem Perforce Client erstellt. Es sieht aus der Perspektive von Git wie ein weiterer Commit aus. Das ist genau der Punkt. Betrachten wir, wie der Perforce-Server mit einem Merge-Commit umgeht:

```

$ git merge origin/master
Auto-merging README
Merge made by the 'recursive' strategy.
 README | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (9/9), 917 bytes | 0 bytes/s, done.
Total 9 (delta 6), reused 0 (delta 0)
remote: Perforce: 100% (3/3) Loading commit tree into memory...
remote: Perforce: 100% (5/5) Finding child commits...
remote: Perforce: Running git fast-export...
remote: Perforce: 100% (3/3) Checking commits...
remote: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
 6afeb15..89cba2b master -> master

```

Git glaubt, dass es funktioniert hat. Werfen wir einen Blick auf den Verlauf der **README** Datei aus der Perspektive von Perforce, indem wir die Revisionsgraphenfunktion von **p4v** verwenden:

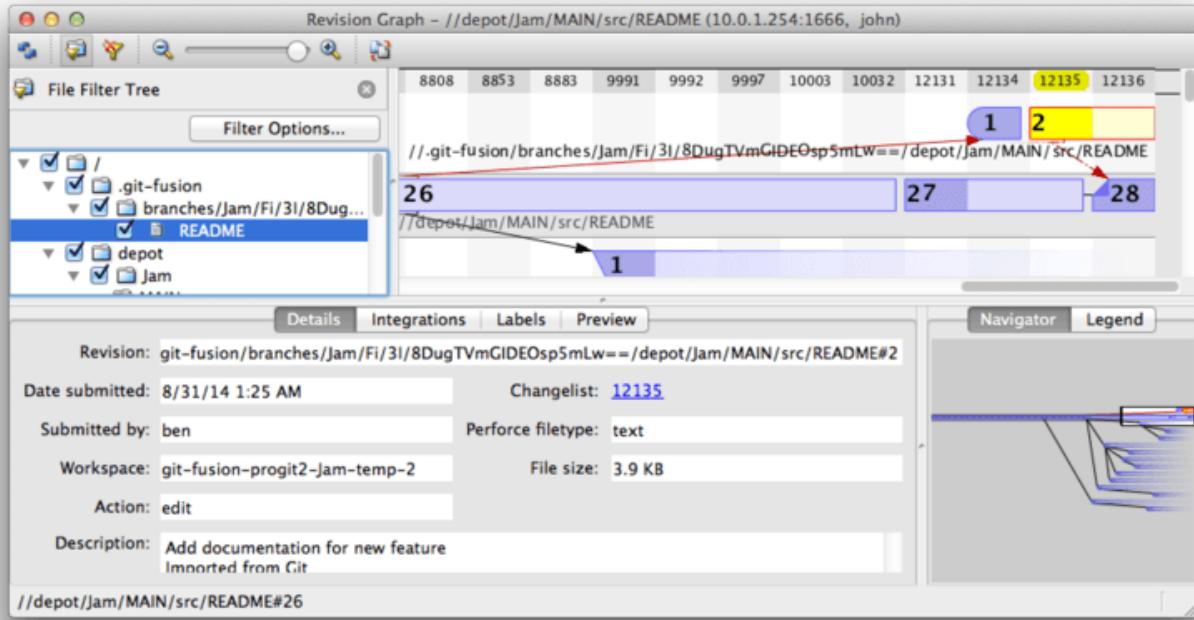


Figure 172. Perforce-Revisionsgraph resultierend aus dem Git-Push

Wenn du diese Darstellung noch nie zuvor gesehen hast, mag sie verwirrend erscheinen, aber sie zeigt die gleichen Konzepte wie ein grafischer Viewer für die Git-Historie. Wir betrachten die Historie der `README` Datei, so dass der Verzeichnisbaum oben links nur diese Datei anzeigt, wenn sie in verschiedenen Branches auftaucht. Oben rechts haben wir ein Diagramm, das veranschaulicht, wie verschiedene Revisionen der Datei zusammenhängen, und die vergrößerte Ansicht dieses Diagramms befindet sich unten rechts. Der Rest der Darstellung wird der Detailansicht für die ausgewählte Revision (in diesem Fall 2) übergeben.

Auffällig ist, dass die Grafik genau so aussieht wie die in dem Verlauf von Git. Perforce hatte keinen namentlich benannten Branch, um die Commits 1 und 2 zu speichern. Also wurde ein „anonymer“ Branch im `.git-fusion` Verzeichnis erstellt, um sie zu speichern. Das gilt auch für benannte Git-Branches, die keinem benannten Perforce-Branch entsprechen (Du kannst sie später über die Konfigurationsdatei einem Perforce-Branch zuordnen).

Das meiste geschieht hinter den Kulissen, und das Ergebnis ist, dass eine Person in der Gruppe Git und eine andere Perforce verwenden kann wobei keine von ihnen von der Entscheidung der anderen Person weiß.

### Git-Fusion Zusammenfassung

Wenn du Zugang zu deinem Perforce-Server hast (oder erhalten kannst), ist Git Fusion eine gute Möglichkeit, Git und Perforce zum gegenseitigen Austausch zu bewegen. Es ist ein wenig Konfiguration erforderlich, aber die Lernkurve ist nicht sehr steil. Dieses ist einer der wenigen Abschnitte in diesem Kapitel, in denen Warnungen über die Verwendung von Gits voller Leistung nicht erscheinen. Das heißt nicht, dass Perforce mit allem, was du ihm zumutest, zufrieden sein wirst – wenn du versuchst, eine bereits gepushte Historie neu zu schreiben, wird Git Fusion sie ablehnen – aber Git Fusion gibt sich sehr große Mühe, sich nativ anzufühlen. Du kannst sogar Git-Submodule verwenden (obwohl sie für Perforce-Anwender seltsam aussehen werden) und Branches mergen (das wird als Integration auf der Perforce-Seite erfasst).

Wenn du den Administrator deines Servers nicht davon überzeugen kannst, Git Fusion einzurichten, gibt es noch eine weitere Möglichkeit, diese Tools gemeinsam zu nutzen.

## Git-p4

Git-p4 ist eine bidirektionale Brücke zwischen Git und Perforce. Es läuft vollständig in deinem Git-Repository, so dass du keinen Zugriff auf den Perforce-Server benötigst (mit Ausnahme der Benutzer-Anmeldeinformationen). Git-p4 ist nicht so flexibel und keine Komplettlösung wie Git Fusion, aber es ermöglicht dir, das meiste von dem zu tun, was du tun möchtest, ohne die Serverumgebung zu beeinträchtigen.



Du brauchst das **p4** Tool an einer beliebigen Stelle in deinem **PATH**, um mit git-p4 zu arbeiten. Zur Zeit ist es unter <https://www.perforce.com/downloads/Perforce/20-User> frei verfügbar.

### Einrichtung

So werden wir beispielsweise den Perforce-Server wie oben gezeigt von der Git Fusion OVA (Open-Virtualization-Archive-Datei) aus verwenden, aber wir umgehen den Git Fusion-Server und gehen direkt zur Perforce-Versionskontrolle.

Um den von git-p4 benötigten Befehlszeilen-Client **p4** verwenden zu können, musst du ein paar Umgebungsvariablen setzen:

```
$ export P4PORT=10.0.1.254:1666  
$ export P4USER=john
```

### Erste Schritte

Wie bei allem in Git ist das Klonen der erste Befehl:

```
$ 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
```

Dadurch entsteht ein Git-technisch „flacher“ Klon. Nur die allerletzte Perforce-Revision wird in Git importiert. Denke daran, Perforce ist nicht dazu gedacht, jedem Benutzer alle Revisionen zu übergeben. Das ist ausreichend, um Git als Perforce-Client zu verwenden, ist aber für andere Zwecke ungeeignet.

Sobald es abgeschlossen ist, haben wir ein voll funktionsfähiges 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
```

Beachte, dass es für den Perforce-Server einen „p4“ Remote gibt, aber alles andere sieht aus wie ein Standardklon. Eigentlich ist das etwas irreführend; es gibt dort nicht wirklich einen Remote.

```
$ git remote -v
```

In diesem Repository existieren überhaupt keine Remotes. Git-p4 hat einige Referenzen erstellt, um den Zustand des Servers darzustellen. Für `git log` sehen sie aus wie Remote-Referenzen, aber sie werden nicht von Git selbst verwaltet. Man kann **nicht** zu ihnen pushen.

## Workflow

Okay, lass uns ein paar Arbeiten erledigen. Nehmen wir an, du hast einige Änderungen bei einem sehr wichtigen Feature gemacht und bist bereit, es dem Rest deines Teams zu zeigen.

```
$ 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
```

Wir haben zwei neue Commits erstellt, die wir an den Perforce-Server übermitteln können. Schauen wir mal, ob heute noch jemand anderes Änderungen gemacht hat:

```
$ git p4 sync
git p4 sync
Performing incremental import into refs/remotes/p4/master git branch
Depot paths: //depot/www/live/
Import destination: refs/remotes/p4/master
Importing revision 12142 (100%)
$ git log --oneline --all --graph --decorate
* 75cd059 (p4/master, p4/HEAD) Update copyright
| * 018467c (HEAD, master) Change page title
| * c0fb617 Update link
|
* 70eaf78 Initial import of //depot/www/live/ from the state at revision #head
```

Tatsächlich wurde etwas geändert. Die Branches `master` und `p4/master` sind auseinander gelaufen. Das Branching-System von Perforce ist *nicht* wie das von Git, so dass das Übertragen von Merge-Commits keinen Sinn macht. Git-p4 empfiehlt, dass du deine Commits rebased und bietet sogar eine Kurzform dafür:

```
$ 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(-)
```

Vermutlich kannst du das an der Ausgabe erkennen: `git p4 rebase` ist eine Abkürzung für `git p4 sync` gefolgt von `git rebase p4/master`. Das ist zwar noch ein bisschen cleverer, besonders bei der Arbeit mit mehreren Branches, ist aber eine gute Annäherung.

Jetzt ist unser Verlauf wieder linear und bereit, in Perforce eingereicht zu werden. Der Befehl `git p4 submit` versucht, für jeden Git-Commit zwischen `p4/master` und `master`, eine neue Perforce-Revision zu erstellen. Beim Ausführen werden wir in unseren bevorzugten Editor weitergeleitet, der Inhalt der Datei sieht dann ungefähr so aus:

```
# 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>
```

Das ist im Wesentlichen derselbe Inhalt, den du bei der Ausführung von `p4 submit` sehen würdest. Ausgenommen sind die Dinge am Ende, die git-p4 sinnvollerweise mit aufgenommen hat. Git-p4 versucht, deine Git- und Perforce-Einstellungen individuell zu berücksichtigen, wenn es einen Namen für ein Commit- oder Changeset angeben muss. In einigen Fällen willst du ihn eventuell überschreiben. Wenn beispielsweise der Git-Commit, den du importierst, von einem Mitwirkenden geschrieben wurde, der kein Perforce-Benutzerkonto hat, kannst du trotzdem wollen, dass sich das daraus ergebende Changeset so aussieht, als hätte er es geschrieben (und nicht du).

Git-p4 hat die Nachricht aus dem Git-Commit zweckmäßigerweise als Inhalt für dieses Perforce Changeset importiert. Alles, was wir tun müssen, ist zweimal speichern (einmal für jeden Commit) und beenden. Die resultierende Shell-Ausgabe sieht in etwa so aus:

```
$ 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
```

Das Ergebnis sieht so aus, als ob wir gerade ein `git push` gemacht hätten. Das kommt dem, was tatsächlich passiert sehr nahe.

Beachte, dass während dieses Prozesses jeder Git-Commit in einen Perforce Changeset umgewandelt wird. Wenn du ihn in einen einzelnen Changeset zusammenfassen möchtest, kannst du dies mit einem interaktiven Rebase tun, bevor du `git p4 submit` ausführst. Bitte bedenke, dass die SHA-1-Hashes aller Commits, die als Changesets eingereicht wurden, sich geändert haben. Der Grund dafür ist, dass git-p4 am Ende jedes Commits, den es konvertiert, eine Zeile hinzufügt:

```
$ 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]
```

Was passiert, wenn du versuchst, einen Merge-Commit einzureichen? Versuchen wir es einmal. Dies ist der Zustand, in die wir jetzt sind:

```
$ 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
```

Der Verlauf von Git und Perforce weicht nach 775a46f voneinander ab. Die Git-Seite hat zwei Commits, dann einen Merge-Commit mit dem Perforce Head, dann einen weiteren Commit. Wir werden versuchen, diese auf einem einzelnen Changeset auf der Perforce-Seite einzureichen. Schauen wir mal was passiert, wenn wir versuchen, ihn jetzt einzubringen:

```
$ 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
```

Das `-n` Flag ist die Abkürzung für `--dry-run`. Das bedeutet der Befehl wird nur simuliert und nicht wirklich ausgeführt. Es erfolgt jedoch eine Ausgabe, was passieren würde, wenn man ihn wirklich ausführen, was sehr hilfreich ist. Hier sieht es so aus, als hätten wir drei Perforce-Änderungssets erstellt, die den drei Nicht-Merge-Commits entsprechen, die noch nicht auf dem Perforce-Server vorhanden sind. Das hört sich nach genau dem an, was wir wollen. Lass uns jetzt wirklich ausführen und sehen was passiert:

```
$ 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
```

Unsere Historie wurde linearisiert, genau so, als hätten wir vor dem Einreichen rebased, was auch tatsächlich passiert ist. So kannst du auf der Git-Seite selbst Branches erstellen, darauf arbeiten, verwerfen und mergen, ohne befürchten zu müssen, dass dein Verlauf sich mit Perforce nicht mehr verträgt. Wenn du einen Rebase durchführen kannst, ist es auch möglich, ihn zu einem Perforce-Server beizusteuern.

## Branching

Wenn dein Perforce-Projekt mehrere Branches hat, hast du Glück gehabt. `git-p4` kann damit so umgehen, dass es sich wie Git anfühlt. Angenommen, dein Perforce-Depot ist so aufgebaut:

```
//depot
└── project
    ├── main
    └── dev
```

Nehmen wir weiter an, du hast einen `dev` Branch, der eine View-Spezifikation hat, die so aussieht:

```
//depot/project/main/... //depot/project/dev/...
```

Git-p4 kann diese Situation automatisch erkennen und das Richtige tun:

```
$ git p4 clone --detect-branches //depot/project@all
Importing from //depot/project@all into project
Initialized empty Git repository in /private/tmp/project/.git/
Importing revision 20 (50%)
    Importing new branch project/dev

    Resuming with change 20
Importing revision 22 (100%)
Updated branches: main dev
$ cd project; git log --oneline --all --graph --decorate
* eae77ae (HEAD, p4/master, p4/HEAD, master) main
| * 10d55fb (p4/project/dev) dev
| * a43cfae Populate //depot/project/main/... //depot/project/dev/....
|/
* 2b83451 Project init
```

Beachte den „@all“ Spezifikator im Depotpfad, der git-p4 anweist, nicht nur das neueste Changeset für diesen Teilbaum, sondern alle Changesets, die jemals diese Pfade beeinflusst haben, zu klonen. Das ist näher an Gits Konzept des Klonens, aber wenn du an einem Projekt mit einer langen Historie arbeitest, könnte es einige Zeit dauern den Klon zu kopieren.

Das **--detect-branches** Flag weist git-p4 an, die Branch-Spezifikationen von Perforce zu verwenden, um die Branches den Git refs zuzuordnen. Sind diese Zuordnungen nicht auf dem Perforce-Server vorhanden (was eine absolut zulässige Methode zur Verwendung von Perforce ist), kannst du git-p4 mitteilen, wie die Zuordnung der Branches zu sein hat. Du erhältst dann das gleiche Ergebnis:

```
$ 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 .
```

Das Setzen der Konfigurationsvariablen **git-p4.branchList** auf **main:dev** teilt git-p4 mit, dass „main“ und „dev“ beides Branches sind, wobei der zweite ein untergeordnetes Element des ersten ist.

Machen wir jetzt neben **git checkout -b dev p4/project/dev** noch einige Commits, dann ist git-p4 klug genug, um den richtigen Branch anzusprechen, wenn wir anschließend **git p4 submit** ausführen. Leider kann git-p4 keine flachen Klone mit mehreren Branches mischen. Wenn du ein großes Projekt hast und an mehr als einem Branch arbeiten willst, musst du jeden Branch, den du einreichen möchtest, einmal mit **git p4 clone** erstellen.

Für die Erstellung oder Integration von Branches musst du einen Perforce-Client verwenden. Git-p4 kann nur synchronisieren und an bestehende Branches senden und es kann nur einen linearen

Changeset auf einmal durchführen. Bei dem Mergen zweier Branches in Git und dem Versuch, das neue Changeset einzureichen, wird nur ein Bündel von Dateiänderungen aufgezeichnet. Die Metadaten über die an der Integration beteiligten Branches gehen dabei verloren.

## Git und Perforce, Zusammenfassung

Git-p4 ermöglicht die Verwendung eines Git-Workflows mit einem Perforce-Server und ist darin ziemlich gut. Es ist jedoch wichtig, sich klar zu machen, dass Perforce für den Quellcode verantwortlich ist und du Git nur für die lokale Arbeit verwendest. Sei einfach sehr vorsichtig bei der Weitergabe von Git-Commits. Wenn du einen Remote hast, den auch andere Benutzer verwenden, pushe keine Commits, die zuvor noch nicht an den Perforce-Server übertragen wurden.

Möchtest du die Verwendung von Perforce und Git als Clients für die Versionskontrolle frei kombinieren, dann musst du den Server-Administrator davon überzeugen, Git zu installieren. Git Fusion macht die Verwendung von Git zu einem erstklassigen Versionskontroll-Client für einen Perforce-Server.

# Migration zu Git

Wenn du eine bestehende Quelltext-Basis in einem anderen VCS hast, aber dich für die Verwendung von Git entschieden hast, musst du dein Projekt auf die eine oder andere Weise migrieren. Dieser Abschnitt geht auf einige Importfunktionen für gängige Systeme ein und zeigt anschließend, wie du deinen eigenen benutzerdefinierten Importeur entwickeln kannst. Du lernst, wie man Daten aus einigen der größeren, professionell genutzten SCM-Systeme importiert. Sie werden von der Mehrheit der Benutzer, die wechseln wollen genutzt. Für diese Systeme sind oft hochwertige Migrations-Tools verfügbar.

## Subversion

Wenn Du den vorherigen Abschnitt über die Verwendung von `git svn` gelesen hast, kannst du die Anweisungen zu `git svn clone` dazu benutzen, um ein Repository zu klonen. Anschliessend benötigst du den Subversion-Servers nicht mehr. Pushe alles zu einem neuen Git-Server und nutze dann nur noch diesen. Der Verlauf kann dabei aus dem Subversion-Server gezogen werden, was einige Zeit in Anspruch nehmen kann – abhängig von der Geschwindigkeit, mit der dein SVN-Server die Historie ausliefern kann.

Allerdings ist der Import nicht perfekt. Da er aber so lange dauert, kannst du es auch direkt richtig machen. Das erste Problem sind die Autoreninformationen. In Subversion hat jede Person, die einen Commit durchführt, auch einen Benutzer-Account auf dem System, der in den Commit-Informationen erfasst wird. Die Beispiele im vorherigen Abschnitt zeigen an einigen Stellen `schacon`, wie z.B. der `blame` Output und das `git svn log`. Wenn du diese auf bessere Git-Autorendaten abbilden möchtest, benötigst du eine Zuordnung der Subversion-Benutzer zu den Git-Autoren. Erstelle eine Datei mit Namen `users.txt`, die diese Zuordnung in folgendem Format vornimmt:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Um eine Liste der Autorennamen zu erhalten, die SVN verwendet, kannst du diesen Befehl ausführen:

```
$ svn log --xml --quiet | grep author | sort -u | \
perl -pe 's/.*/>(.*)<.*/$1 = /'
```

Das erzeugt die Protokollausgabe im XML-Format, behält nur die Zeilen mit Autoreninformationen, verwirft Duplikate und entfernt die XML-Tags. Natürlich funktioniert das nur auf einem Computer, auf dem `grep`, `sort` und `perl` installiert sind. Leite diese Ausgabe dann in deine `users.txt` Datei um, damit du die entsprechenden Git-Benutzerdaten neben jedem Eintrag hinzufügen kannst.



Wenn du dies auf einem Windows-Computer versuchen solltest, treten an dieser Stelle Probleme auf. Microsoft hat unter <https://docs.microsoft.com/en-us/azure/devops/repos/git/perform-migration-from-svn-to-git> einige gute Ratschläge und Beispiele bereitgestellt.

Du kannst diese Datei an `git svn` übergeben, um die Autorendaten genauer abzubilden. Außerdem kannst du `git svn` anweisen, die Metadaten, die Subversion normalerweise importiert, nicht zu berücksichtigen. Übergebe dazu `--no-metadata` an den `clone` oder `init` Befehl. Die Metadaten enthalten eine `git-svn-id` in jeder Commit-Nachricht, die Git während des Imports generiert. Dies kann dein Git-Log aufblähen und es möglicherweise etwas unübersichtlich machen.



Du musst die Metadaten beibehalten, wenn du im Git-Repository vorgenommene Commits wieder in das ursprüngliche SVN-Repository spiegeln möchtest. Wenn du die Synchronisierung nicht in deinem Commit-Protokoll haben möchtest, kannst du den Parameter `--no-metadata` weglassen.

Dadurch sieht dein `import` Befehl so aus:

```
$ git svn clone http://my-project.googlecode.com/svn/ \
--authors-file=users.txt --no-metadata --prefix "" -s my_project
$ cd my_project
```

Nun solltest du einen passenderen Subversion-Import in deinem `my_project` Verzeichnis haben. Anstelle von Commits, die so aussehen:

```
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
```

sehen diese jetzt so aus:

```
commit 03a8785f44c8ea5cdb0e8834b7c8e6c469be2ff2
Author: Scott Chacon <schacon@geemail.com>
Date:   Sun May 3 00:12:22 2009 +0000

fixed install - go to trunk
```

Nicht nur das Autorenfeld sieht viel besser aus, auch die `git-svn-id` ist nicht mehr vorhanden.

Du solltest auch eine Bereinigung nach dem Import durchführen. Zum einen solltest du die seltsamen Referenzen bereinigen, die `git svn` eingerichtet hat. Verschiebe zuerst die Tags so, dass sie echte Tags und nicht merkwürdige Remote-Banches darstellen. Dann verschiebe den Rest der Branches auf lokale Branches.

Damit die Tags zu richtigen Git-Tags werden, führe folgenden Befehl aus:

```
$ 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
```

Dabei werden die Referenzen, die Remote-Banches waren und mit `refs/remotes/tags/` begonnen haben zu richtigen (leichten) Tags gemacht.

Als nächstes verschiebe den Rest der Referenzen unter `refs/remotes` in lokale Branches:

```
$ 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
```

Es kann vorkommen, dass du einige zusätzliche Branches siehst, die durch `@xxx` ergänzt sind (wobei xxx eine Zahl ist), während du in Subversion nur einen Branch siehst. Es handelt sich hierbei um eine Subversion-Funktion mit der Bezeichnung „peg-revisions“, für die Git kein syntaktisches Gegenstück hat. Daher fügt `git svn` einfach die SVN-Versionsnummer zum Branch-Namen hinzu, genau so, wie du es in SVN geschrieben hättest, um die peg-Revision dieses Branchs anzusprechen. Wenn du dich nicht mehr um die peg-Revisions sorgen willst, entferne diese einfach:

```
$ for p in $(git for-each-ref --format='%(refname:short)' | grep @); do git branch -D
$p; done
```

Jetzt sind alle alten Branches echte Git-Banches und alle alten Tags sind echte Git-Tags.

Da wäre noch eine letzte Sache zu klären. Leider erstellt `git svn` einen zusätzlichen Branch mit dem Namen `trunk`, der auf den Standard-Branch von Subversion gemappt wird, aber die `trunk` Referenz zeigt auf die gleiche Position wie `master`. Da `master` in Git eher idiomatisch ist, hier die Anleitung zum Entfernen des extra Branchs:

```
$ git branch -d trunk
```

Das Letzte, was du tun musst, ist deinem neuen Git-Server als Remote hinzuzufügen und zu ihm zu pushen. Hier ist ein Beispiel für das hinzufügen deines Servers als Remote:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Um alle deine Branches und Tags zu aktualisieren, kannst du jetzt diese Anweisungen ausführen:

```
$ git push origin --all  
$ git push origin --tags
```

Alle deine Branches und Tags sollten sich nun auf deinem neuen Git-Server in einem schönen, sauberen Import befinden.

## Mercurial

Merkurial und Git haben ziemlich ähnliche Modelle für die Darstellung von Versionen. Außerdem ist Git etwas flexibler, so dass die Konvertierung eines Repositorys von Merkurial nach Git ziemlich einfach ist. Dazu wird ein Tool mit der Bezeichnung „hg-fast-export“ verwendet, dass du aus github beziehen kannst:

```
$ git clone https://github.com/frej/fast-export.git
```

Der erste Schritt bei der Umstellung besteht darin, einen vollständigen Klon des zu konvertierenden Mercurial-Repository zu erhalten:

```
$ hg clone <remote repo URL> /tmp/hg-repo
```

Der nächste Schritt ist die Erstellung einer Autor-Mapping-Datei. Mercurial ist etwas toleranter als Git für das, was es in das Autorenfeld für Changesets stellt. Das ist daher ein guter Zeitpunkt, um das ganze Projekt zu bereinigen. Das lässt sich mit einem einzeiligen Befehl in einer **bash** Shell erreichen:

```
$ cd /tmp/hg-repo  
$ hg log | grep user: | sort | uniq | sed 's/user: *//' > ../authors
```

Das dauert nur ein paar Sekunden, abhängig davon, wie umfangreich der Verlauf deines Projekts ist. Danach wird die Datei **/tmp/authors** in etwa so aussehen:

```
bob  
bob@localhost  
bob <bob@company.com>  
bob jones <bob <AT> company <DOT> com>  
Bob Jones <bob@company.com>
```

In diesem Beispiel hat die gleiche Person (Bob) Changesets unter vier verschiedenen Namen erstellt, von denen einer tatsächlich korrekt aussieht und einer für einen Git-Commit völlig ungültig wäre. Mit hg-fast-export können wir das beheben. Jede Zeile wird in eine Regel umgewandelt: "<input>"="<output>", wobei ein <input> auf einen <output> abgebildet wird. Innerhalb der Zeichenketten <input> und <output> werden alle Escape-Sequenzen unterstützt, die von Python `string_escape` Encoding verstanden werden. Wenn die Autor-Mapping-Datei keinen passenden <input> enthält, wird dieser Autor unverändert an Git übergeben. Wenn alle Benutzernamen korrekt aussehen, werden wir diese Datei überhaupt nicht brauchen. In diesem Beispiel soll unsere Datei so aussehen:

```
"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>"
```

Die gleiche Art von Mapping-Datei kann zum Umbenennen von Branches und Tags verwendet werden, wenn der Mercurial-Name in Git nicht zulässig ist.

Der nächste Schritt ist die Erstellung unseres neuen Git-Repository und das Ausführen des Exportskripts:

```
$ git init /tmp/converted  
$ cd /tmp/converted  
$ /tmp/fast-export/hg-fast-export.sh -r /tmp/hg-repo -A /tmp/authors
```

Das `-r` Flag informiert hg-fast-export darüber, wo das Mercurial-Repository zu finden ist, das wir konvertieren möchten. Das `-A` Flag sagt ihm, wo es die Autor-Mapping-Datei findet (Branch- und Tag-Mapping-Dateien werden jeweils durch die `-B` und `-T` Flags definiert). Das Skript analysiert Mercurial Change-Sets und konvertiert sie in ein Skript für Gits „fast-import“ Funktion (auf die wir später noch näher eingehen werden). Das dauert ein wenig (obwohl es *viel* schneller ist, als wenn es über das Netzwerk laufen würde). Der Output ist ziemlich umfangreich:

```
$ /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

```

Das ist so ziemlich alles, was es dazu zu sagen gibt. Alle Mercurial-Tags wurden in Git-Tags umgewandelt, und Mercurial-Banches und -Lesezeichen wurden in Git-Banches umgewandelt. Jetzt kannst du das Repository in das neue serverseitige System pushen:

```
$ git remote add origin git@my-git-server:myrepository.git
$ git push origin --all
```

## Perforce

Bei dem nächsten System, aus dem du importieren könntest, handelt es sich um Perforce. Wie bereits erwähnt, gibt es zwei Möglichkeiten, wie Git und Perforce miteinander kommunizieren

können: git-p4 und Perforce Git Fusion.

## Perforce Git Fusion

Git Fusion macht diesen Prozess relativ unkompliziert. Konfiguriere einfach deine Projekteinstellungen, Benutzerzuordnungen und Branches mit Hilfe einer Konfigurationsdatei (wie in [Git Fusion](#) beschrieben) und klon das Repository. Git Fusion bietet dir ein natives Git-Repository, mit dem du nach Belieben auf einen nativen Git-Host wechseln kannst. Du kannst Perforce sogar als deinen Git-Host verwenden, wenn du möchtest.

### Git-p4

Git-p4 kann auch als Import-Tool fungieren. Als Beispiel werden wir das Jam-Projekt aus dem Perforce Public Depot importieren. Um deinen Client einzurichten, musst du die Umgebungsvariable P4PORT exportieren und auf das Perforce-Depot verweisen:

```
$ export P4PORT=public.perforce.com:1666
```



Zur weiteren Bearbeitung benötigst du ein Perforce-Depot, mit dem du dich verbinden kannst. Wir werden das öffentliche Depot unter public.perforce.com für unsere Beispiele verwenden. Du kannst aber jedes Depot nutzen, zu dem du Zugang hast.

Führe den Befehl `git p4 clone` aus, um das Jam-Projekt vom Perforce-Server zu importieren, wobei du das Depot, den Projektpfad und den Pfad angibst, in den du das Projekt importieren möchtest:

```
$ 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%)
```

Dieses spezielle Projekt hat nur einen Branch, aber wenn du Branches hast, die mit Branch Views (oder nur einer Gruppe von Verzeichnissen) eingerichtet sind, kannst du ergänzend zum Befehl `git p4 clone` das Flag `--detect-branches` verwenden, um alle Branches des Projekts zu importieren. Siehe [Branching](#) für ein paar weitere Details.

Jetzt bist du fast fertig. Wenn du in das Verzeichnis `p4import` wechselst und `git log` ausführst, kannst du dein importiertes Projekt sehen:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800
```

Correction to line 355; change </UL> to </OL>.

```
[git-p4: depot-paths = "//public/jam/src/": change = 8068]
```

```
commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98  
Author: kwirth <kwirth@perforce.com>  
Date: Tue Jul 7 01:35:51 2009 -0800
```

Fix spelling error on Jam doc page (cummulative -> cumulative).

```
[git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

Du kannst sehen, dass **git-p4** in jeder Commit-Nachricht eine Kennung hinterlassen hat. Es ist gut, diese Kennung dort zu behalten, falls du später auf die Perforce-Änderungsnummer verweisen musst. Wenn du den Identifier jedoch entfernen möchtest, ist jetzt der richtige Zeitpunkt – bevor du mit der Arbeit am neuen Repository beginnst. Du kannst **git filter-branch** verwenden, um die Identifikations-Strings in großer Anzahl zu entfernen:

```
$ git filter-branch --msg-filter 'sed -e "/^\\[git-p4:/d"'  
Rewrite e5da1c909e5db3036475419f6379f2c73710c4e6 (125/125)  
Ref 'refs/heads/master' was rewritten
```

Wenn du **git log** ausführst, kannst du sehen, dass sich alle SHA-1-Prüfsummen für die Commits geändert haben, aber die **git-p4** Zeichenketten sind nicht mehr in den Commit-Nachrichten enthalten:

```
$ git log -2  
commit b17341801ed838d97f7800a54a6f9b95750839b7  
Author: giles <giles@giles@perforce.com>  
Date: Wed Feb 8 03:13:27 2012 -0800  
  
Correction to line 355; change </UL> to </OL>.  
  
commit 3e68c2e26cd89cb983eb52c024ecdfba1d6b3fff  
Author: kwirth <kwirth@perforce.com>  
Date: Tue Jul 7 01:35:51 2009 -0800
```

Fix spelling error on Jam doc page (cummulative -> cumulative).

Dein Import ist bereit, um ihn auf deinen neuen Git-Server zu pushen.

## Benutzerdefinierter Import

Wenn dein System nicht zu den vorgenannten gehört, solltest du online nach einer Import-Schnittstelle suchen – hochwertige Importer sind für viele andere Systeme verfügbar, darunter CVS, Clear Case, Visual Source Safe, sogar für ein Verzeichnis von Archiven. Wenn keines dieser Tools für dich geeignet ist, du ein obskures Tool nutzt oder anderweitig einen benutzerdefinierten Importprozess benötigst, solltest du **git fast-import** verwenden. Dieser Befehl liest die einfachen Anweisungen von „stdin“ aus, um bestimmte Git-Daten zu schreiben. Es ist viel einfacher, Git-

Objekte auf diese Weise zu erstellen, als die Git-Befehle manuell auszuführen oder zu versuchen, Raw-Objekte zu erstellen (siehe Kapitel 10, [Git Interna](#) für weitere Informationen). Auf diese Weise kannst du ein Import-Skript schreiben, das die notwendigen Informationen aus dem System liest, aus dem du importierst, und Anweisungen direkt auf „stdout“ ausgibt. Du kannst dann dieses Programm ausführen und seine Ausgaben über `git fast-import` pipen.

Um das kurz zu demonstrieren, schreibe eine einfache Import-Anweisung. Angenommen, du arbeitest im `current` Branch, du sicherst dein Projekt, indem du das Verzeichnis gelegentlich in ein mit Zeitstempel versehenes `back_YYYY_MM_DD` Backup-Verzeichnis kopieren und dieses in Git importieren möchtest. Deine Verzeichnisstruktur sieht wie folgt aus:

```
$ ls /opt/import_from
back_2014_01_02
back_2014_01_04
back_2014_01_14
back_2014_02_03
current
```

Damit du ein Git-Verzeichnis importieren kannst, musst du dir ansehen, wie Git seine Daten speichert. Wie du dich vielleicht erinnerst, ist Git im Grunde genommen eine verknüpfte Liste von Commit-Objekten, die auf einen Schnapschuss des Inhalts verweisen. Alles, was du tun musst, ist `fast-import` mitzuteilen, worum es sich bei den Content-Snapshots handelt, welche Commit-Datenpunkte zu ihnen gehören und in welcher Reihenfolge sie in den jeweiligen Ordner gehören. Deine Strategie besteht darin, die Snapshots einzeln durchzugehen und Commits mit dem Inhalt jedes Verzeichnisses zu erstellen. Dabei wird jeder Commit mit dem vorherigen verknüpft.

Wie wir es in Kapitel 8, [Beispiel für Git-forcierte Regeln](#) getan haben, werden wir das in Ruby schreiben, denn damit arbeiten wir normalerweise und es ist eher leicht zu lesen. Du kannst dieses Beispiel sehr leicht in jedem Editor schreiben, den du kennst – er muss nur die entsprechenden Informationen nach `stdout` ausgeben können. Unter Windows musst du besonders darauf achten, dass du am Ende deiner Zeilen keine Zeilenumbrüche einfügst – `git fast-import` ist da sehr empfindlich, wenn es darum geht, nur Zeilenvorschübe (LF) und nicht die von Windows verwendeten Zeilenvorschübe (CRLF) zu verwenden.

Zunächst wechselst du in das Zielverzeichnis und identifizierst jene Unterverzeichnisse, von denen jedes ein Snapshot ist, den du als Commit importieren möchtest. Du wechselst in jedes Unterverzeichnis und gibst die für den Export notwendigen Befehle aus. Deine Hauptschleife sieht so aus:

```
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

```

Führe `print_export` in jedem Verzeichnis aus, das das Manifest und die Markierung des vorherigen Snapshots enthält und das Manifest und die Markierung dieses Verzeichnisses zurückgibt. So kannst du sie richtig verlinken. „Mark“ ist der `fast-import` Begriff für eine Kennung, die du einem Commit mit gibst. Wenn du Commits erstellst, gibst du jedem eine Markierung, mit dem du ihn von anderen Commits aus verlinken kannst. Daher ist das Erste, was deine `print_export` Methode tut, die Generierung einer Markierung aus dem Verzeichnisnamen:

```
mark = convert_dir_to_mark(dir)
```

Du wirst dazu ein Array von Verzeichnissen erstellen und den Indexwert als Markierung verwenden. Eine Markierung muss nämlich eine Ganzzahl (Integer) sein. Deine Methode sieht so aus:

```

$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end

```

Nachdem du nun eine ganzzahlige Darstellung deines Commits hast, benötigst du ein Datum für die Commit-Metadaten. Das Datum wird im Namen des Verzeichnisses ausgewiesen, daher wirst du es auswerten. Die nächste Zeile in deiner `print_export` Datei lautet:

```
date = convert_dir_to_date(dir)
```

wobei `convert_dir_to_date` definiert ist als:

```

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

```

Das gibt einen ganzzahligen Wert für das Datum jedes Verzeichnisses zurück. Die letzte Meta-

Information, die du für jeden Commit benötigst, sind die Committer-Daten, die du in einer globalen Variable hartkodierst:

```
$author = 'John Doe <john@example.com>'
```

Damit bist du startklar für die Ausgabe der Commit-Daten für deinen Importer. Die ersten Informationen beschreiben, dass du ein Commit-Objekt definierst und in welchem Branch es sich befindet, gefolgt von der Markierung, die du generiert hast, den Committer-Informationen und der Commit-Beschreibung und dann, falls vorhanden, der vorherige Commit. Der Code sieht jetzt so aus:

```
# 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
```

Du kannst die Zeitzone (-0700) hartkodieren, da das einfach ist. Wenn du sie aus einem anderen System importierst, musst du die Zeitzone als Offset angeben. Die Commit-Beschreibung muss in einem speziellen Format ausgegeben werden:

```
data (size)\n(contents)
```

Das Format besteht aus den Wortdaten, der Größe der zu lesenden Daten, einer neuen Zeile und schließlich den Daten. Da du später das gleiche Format verwenden musst, um den Datei-Inhalt festzulegen, erstellst du mit `export_data` eine Hilfs-Methode:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Nun musst du nur noch den Dateiinhalt für jeden Schnappschuss angeben. Das ist einfach, denn du hast jeden in einem eigenen Verzeichnis. Du kannst den Befehl `deleteall` ausgeben, gefolgt vom Inhalt jeder Datei im Verzeichnis. Git zeichnet dann jeden Schnappschuss entsprechend auf:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Hinweis: Da viele Systeme ihre Revisionen als Änderungen von einem Commit zum anderen betrachten, kann fast-import auch Befehle mit jedem Commit übernehmen, um anzugeben, welche

Dateien hinzugefügt, entfernt oder geändert wurden und was die neuen Inhalte sind. Du kannst die Unterschiede zwischen den Snapshots berechnen und nur diese Daten bereitstellen, aber das ist komplizierter – in diesem Fall solltest du Git alle Daten übergeben und sie auswerten lassen. Sollte diese Option für deine Daten besser geeignet sein, informiere dich in der [fast-import](#) Man-Page, wie du deine Daten auf diese Weise bereitstellen kannst.

Das Format für die Auflistung des neuen Datei-Inhalts oder die Angabe einer modifizierten Datei mit dem neuen Inhalt lautet wie folgt:

```
M 644 inline path/to/file  
data (size)  
(file contents)
```

Im Beispiel ist es der Modus 644 (wenn du ausführbare Dateien hast musst du stattdessen 755 ermitteln und spezifizieren), und inline besagt, dass der Inhalt unmittelbar nach dieser Zeile aufgelistet wird. Das Verfahren [inline\\_data](#) sieht so aus:

```
def inline_data(file, code = 'M', mode = '644')  
  content = File.read(file)  
  puts "#{code} #{mode} inline #{file}"  
  export_data(content)  
end
```

Bei der Wiederverwendung der Methode [export\\_data](#), die du zuvor definiert hast, handelt es sich um das gleiche Verfahren wie bei der Angabe deiner Commit-Message-Daten.

Als Letztes musst du die aktuelle Markierung an das System zurückgeben, damit sie an die nächste Iteration weitergegeben werden kann:

```
return mark
```

 Wenn du unter Windows arbeitest, musst du unbedingt einen zusätzlichen Arbeitsschritt hinzufügen. Wie bereits erwähnt, verwendet Windows CRLF für Zeilenumbrüche, während [git fast-import](#) nur LF erwartet. Um dieses Problem zu umgehen und [git fast-import](#) gerecht zu werden, musst du Ruby anweisen, LF anstelle von CRLF zu verwenden:

```
$stdout.binmode
```

Das war's. Das Skript ist jetzt komplett:

```
#!/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

```

Wenn du dieses Skript ausführst, solltest du eine Ausgabe wie die folgende erhalten:

```

$ 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
(...)
```

Um den Importer aufzurufen, übergibst du diese Output-Pipe an `git fast-import`, während du dich in dem Git-Verzeichnis befindest, in das du importieren willst. Du kannst ein neues Verzeichnis erstellen und dort `git init` für einen Anfangspunkt ausführen und danach dein Skript ausführen:

```

$ 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
-----
```

Wie du siehst, gibt es nach erfolgreichem Abschluss eine Reihe von Statistiken über den erreichten Status. In diesem Fall hast du 13 Objekte mit insgesamt 4 Commits in einen Branch importiert. Jetzt kannst du `git log` ausführen, um deine neue Historie zu sehen:

```
$ 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
```

So ist es richtig – ein ordentliches, sauberes Git-Repository. Es ist wichtig zu beachten, dass nichts ausgecheckt ist – du hast zunächst keine Dateien in dein Arbeitsverzeichnis. Um sie zu erhalten, musst du deinen Branch auf den aktuellen `master` zurücksetzen:

```
$ ls
$ git reset --hard master
HEAD is now at 3caa046 imported from current
```

```
$ ls  
README.md main.rb
```

Mit dem [fast-import](#) Tool kannst du noch viel mehr tun – bearbeiten von unterschiedlichen Modi, binären Daten, multiplen Branches und Merges, Tags, Verlaufsindikatoren und mehr. Eine Reihe von Beispielen für komplexere Szenarien findest du im [contrib/fast-import](#) Verzeichnis des Git-Quellcodes.

## Zusammenfassung

Du solltest jetzt in der Lage sein, Git als Client für andere Versionskontrollsysteme zu verwenden oder fast jedes vorhandene Repository in Git zu importieren, ohne dabei Daten zu verlieren. Im nächsten Kapitel werden wir die internen Abläufe in Git beschreiben, so dass du jedes einzelne Byte nach Bedarf selbst erzeugen kannst.

# Git Interna

Sie sind möglicherweise von einem der vorherigen Kapitel direkt zu diesem Kapitel gesprungen. Oder aber Sie sind jetzt hier gelandet, nachdem Sie das gesamte Buch chronologisch bis zu diesem Punkt gelesen haben. Ganz egal, wir hier das Innenleben und die Implementierung von Git behandeln. Wir finden, dass das Verstehen dieser Informationen von grundlegender Bedeutung ist, um zu verstehen, wie hilfreich und extrem leistungsfähig Git ist. Andere haben jedoch argumentiert, dass es für Anfänger verwirrend und unnötig komplex sein kann. Daher haben wir diese Informationen zum letzten Kapitel des Buches gemacht, damit Sie sie früh oder später in Ihrem Lernprozess lesen können. Wir überlassen es Ihnen, das zu entscheiden.

Jetzt wo sie hier sind, lassen sie uns anfangen. Erstens, wenn es noch nicht klar ist, ist Git grundsätzlich ein inhaltsadressierbares Dateisystem mit einer aufgesetzten VCS-Benutzeroberfläche. Sie werden in Kürze mehr darüber erfahren, was dies bedeutet.

In den Anfängen von Git (meist vor 1.5) war die Benutzeroberfläche sehr viel komplexer, da dieses Dateisystem mehr im Vordergrund stand als ein hochglänzendes VCS. In den letzten Jahren wurde die Benutzeroberfläche weiterentwickelt, bis sie so aufgeräumt und benutzerfreundlich ist wie in vielen anderen Systemen auch. Die Vorurteile gegenüber der früheren Git-Benutzeroberfläche, die komplex und schwierig zu erlernen war, blieben jedoch erhalten.

Die inhaltsadressierbare Dateisystemsicht ist erstaunlich abgefahren, deshalb werden wir es als Erstes in diesem Kapitel behandeln. Anschließend lernen Sie die Transportmechanismen und die Repository-Wartungsaufgaben kennen, mit denen Sie sich möglicherweise befassen müssen.

## Basisbefehle und Standardbefehle (Plumbing and Porcelain)

In diesem Buch wird in erster Linie beschrieben, wie Git mit etwa 30 Standardbefehlen wie `checkout`, `branch`, `remote` usw. verwendet wird. Git war ursprünglich ein Werkzeug für ein Versionskontrollsystem und kein benutzerfreundliches VCS. Somit verfügt es über eine Reihe von Basisbefehlen, die auf niedriger Ebene ausgeführt werden und so konzipiert sind, dass sie im UNIX-Stil verkettet oder über Skripte aufgerufen werden können. Diese Befehle werden im Allgemeinen als Basisbefehle von Git bezeichnet, während die benutzerfreundlicheren Befehle als Standardbefehle bezeichnet werden.

Wie Sie bereits bemerkt haben, befassen sich die ersten neun Kapitel dieses Buches fast ausschließlich mit Standardbefehlen. In diesem Kapitel werden Sie sich jedoch hauptsächlich mit den Basisbefehlen der niedrigeren Ebene befassen. Diese ermöglichen Ihnen den Zugriff auf das Innenleben von Git und helfen Ihnen dabei zu demonstrieren, wie und warum Git das tut, was es tut. Viele dieser Befehle sollten nicht manuell in der Befehlszeile verwendet werden, sondern als Bausteine für neue Tools und benutzerdefinierte Skripts genutzt werden.

Wenn Sie `git init` in einem neuen oder vorhandenen Verzeichnis ausführen, erstellt Git das `.git` Verzeichnis, in dem sich fast alles befindet, was Git speichert und bearbeitet. Wenn Sie Ihr Repository sichern oder klonen möchten, erhalten Sie beim Kopieren dieses einzelnen Verzeichnisses fast alles, was Sie benötigen. Dieses gesamte Kapitel befasst sich im Wesentlichen

mit dem, was Sie in diesem Verzeichnis sehen können. So sieht ein neu initialisiertes `.git`-Verzeichnis normalerweise aus:

```
$ ls -F1
config
description
HEAD
hooks/
info/
objects/
refs/
```

Abhängig von Ihrer Git-Version sehen Sie dort möglicherweise zusätzlichen Inhalt, aber dies ist ein neu erstelltes `git init` Repository – das sehen sie standardmäßig. Die `description` Datei wird nur vom GitWeb-Programm verwendet, machen Sie sich also keine Sorgen darum. Die `config` Datei enthält Ihre projektspezifischen Konfigurationsoptionen, und das `info` Verzeichnis enthält eine globale Ausschlussdatei für ignorierte Muster, die Sie nicht in einer `.gitignore` Datei verfolgen möchten. Das `hooks` Verzeichnis enthält Ihre client- oder serverseitigen Hook-Skripte, die ausführlich in [Git Hooks](#) beschrieben werden.

Dies hinterlässt vier wichtige Einträge: die `HEAD` – und (noch zu erstellenden) 'index` Dateien sowie die `objects` – und `refs` Verzeichnisse. Dies sind die Kernelemente von Git. Das `objects`-Verzeichnis speichert den gesamten Inhalt für Ihre Datenbank, das `refs` Verzeichnis speichert Zeiger auf Commit-Objekte in diesen Daten (Branches, Tags, Remotes, usw.) und die `HEAD` Datei zeigt auf den Branch, den Sie gerade ausgecheckt haben. In der `index` Datei speichert Git Ihre Staging-Bereichsinformationen. Sie werden sich nun jeden dieser Abschnitte genauer ansehen, um zu sehen, wie Git funktioniert.

## Git Objekte

Git ist ein inhalts-adressierbares Dateisystem. Toll. Was bedeutet das? Dies bedeutet, dass der Kern von Git ein einfacher Schlüssel-Wert-Datenspeicher ist. Dies bedeutet, dass Sie jede Art von Inhalt in ein Git-Repository einfügen können. Git gibt Ihnen einen eindeutigen Schlüssel zurück, mit dem Sie den Inhalt später abrufen können.

Schauen wir uns als Beispiel den Installationsbefehl `git hash-object` an, der einige Daten aufnimmt, sie in Ihrem `.git/objects` Verzeichnis (der *object database*) speichert und Ihnen den eindeutigen Schlüssel zurückgibt, der nun auf dieses Datenobjekt verweist.

Zunächst initialisieren Sie ein neues Git-Repository und stellen sicher, dass sich (vorhersehbar) nichts im `objects` Verzeichnis befindet:

```
$ 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 hat das Verzeichnis **objects** initialisiert und darin die Unterverzeichnisse **pack** und **info** erstellt, aber es gibt darin keine regulären Dateien. Jetzt erstellen wir mit **git hash-object** ein neues Datenobjekt und speichern es manuell in Ihrer neuen Git-Datenbank:

```
$ echo 'test content' | git hash-object -w --stdin  
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

In seiner einfachsten Form würde **git hash-object** den Inhalt, den Sie ihm übergeben haben, nehmen und *würde* lediglich den eindeutigen Schlüssel zurückgeben, der zum Speichern in Ihrer Git-Datenbank verwendet werden soll. Die Option **-w** weist dann den Befehl an, den Schlüssel nicht einfach zurückzugeben, sondern das Objekt in die Datenbank zu schreiben. Schließlich weist die Option **--stdin git hash-object** an, den zu verarbeitenden Inhalt von **stdin** abzurufen. Andernfalls würde der Befehl ein Dateinamenargument am Ende des Befehls erwarten, das den zu verwendenden Inhalt enthält.

Die Ausgabe des obigen Befehls ist ein Prüfsummen-Hash mit 40 Zeichen. Dies ist der SHA-1-Hash – eine Prüfsumme des Inhalts, den Sie speichern, sowie eine Kopfzeile, über die Sie in Kürze mehr erfahren werden. Jetzt können Sie sehen, wie Git Ihre Daten gespeichert hat:

```
$ find .git/objects -type f  
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Wenn Sie Ihr Verzeichnis **objects** erneut untersuchen, können Sie feststellen, dass es jetzt eine Datei für diesen neuen Inhalt enthält. Auf diese Weise speichert Git den Inhalt initial — als einzelne Datei pro Inhaltselement, benannt mit der SHA-1-Prüfsumme des Inhalts und seiner Kopfzeile. Das Unterverzeichnis wird mit den ersten 2 Zeichen des SHA-1 benannt, und der Dateiname enthält die verbleibenden 38 Zeichen.

Sobald Sie Inhalt in Ihrer Objektdatenbank haben, können Sie diesen Inhalt mit dem Befehl **git cat-file** untersuchen. Dieser Befehl ist eine Art Schweizer Taschenmesser für die Inspektion von Git-Objekten. Wenn Sie **-p** an **cat-file** übergeben, wird der Befehl angewiesen, zuerst den Inhaltstyp zu ermitteln und ihn dann entsprechend anzuzeigen:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4  
test content
```

Jetzt können Sie Inhalte zu Git hinzufügen und wieder herausziehen. Sie können dies auch mit Inhalten in Dateien tun. Sie können beispielsweise eine einfache Versionskontrolle für eine Datei durchführen. Erstellen Sie zunächst eine neue Datei und speichern Sie deren Inhalt in Ihrer Datenbank:

```
$ echo 'version 1' > test.txt
```

```
$ git hash-object -w test.txt  
83baae61804e65cc73a7201a7252750c76066a30
```

Schreiben Sie dann neuen Inhalt in die Datei und speichern Sie ihn erneut:

```
$ echo 'version 2' > test.txt  
$ git hash-object -w test.txt  
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Ihre Objektdatenbank enthält nun beide Versionen dieser neuen Datei (sowie den ersten Inhalt, den Sie dort gespeichert haben):

```
$ find .git/objects -type f  
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a  
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30  
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Zu diesem Zeitpunkt können Sie Ihre lokale Kopie dieser `test.txt` Datei löschen und dann mit Git entweder die erste gespeicherte Version aus der Objektdatenbank abrufen:

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt  
$ cat test.txt  
version 1
```

oder die zweite Version:

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt  
$ cat test.txt  
version 2
```

Es ist jedoch nicht sinnvoll, sich den SHA-1-Schlüssel für jede Version Ihrer Datei zu merken. Außerdem speichern Sie den Dateinamen nicht in Ihrem System, sondern nur den Inhalt. Dieser Objekttyp wird als *blob* bezeichnet. Git kann Ihnen den Objekttyp jedes Objekts in Git mitteilen, wenn sie den SHA-1-Schlüssel mit `git cat-file -t` angegeben:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a  
blob
```

## Baum Objekte

Die nächste Art von Git-Objekte, die wir untersuchen, ist der *baum*, der das Problem des Speicherns des Dateinamens löst und es Ihnen auch ermöglicht, eine Gruppe von Dateien zusammen zu speichern. Git speichert Inhalte auf ähnliche Weise wie ein UNIX-Dateisystem, jedoch etwas vereinfacht. Der gesamte Inhalt wird als Baum- und Blob-Objekte gespeichert, wobei Bäume UNIX-

Verzeichniseinträgen entsprechen und Blobs mehr oder weniger Inodes oder Dateiinhalten entsprechen. Ein einzelnes Baumobjekt enthält einen oder mehrere Einträge, von denen jeder der SHA-1-Hash eines Blobs oder Teilbaums mit dem zugehörigen Modus, Typ und Dateinamen ist. Angenommen sie haben ein Projekt, in dem der aktuellste Baum folgendermaßen aussieht:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

Die `master^{tree}` Syntax gibt das Baumobjekt an, auf das durch das letzte Commit in Ihrem `master` Branch verwiesen wird. Beachten Sie, dass das Unterverzeichnis `lib` kein Blob ist, sondern ein Zeiger auf einen anderen Baum:

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Je nachdem, welche Shell Sie verwenden, können bei der Verwendung der `master^{tree}` Syntax Fehler auftreten.

In CMD unter Windows wird das Zeichen `^` für das Escapezeichen verwendet, daher müssen Sie es verdoppeln, um dies zu vermeiden: `git cat-file -p master^^{tree}`. Bei Verwendung von PowerShell müssen Parameter mit {} Zeichen in Anführungszeichen gesetzt werden, um eine fehlerhafte Syntaxanalyse des Parameters zu vermeiden: `git cat-file -p 'master^{tree}'`.

Wenn Sie ZSH verwenden, wird das Zeichen `^` zum Verschieben verwendet, daher müssen Sie den gesamten Ausdruck in Anführungszeichen setzen: `git cat-file -p "master^{tree}"`.

Konzeptionell sehen die von Git gespeicherten Daten ungefähr so aus:



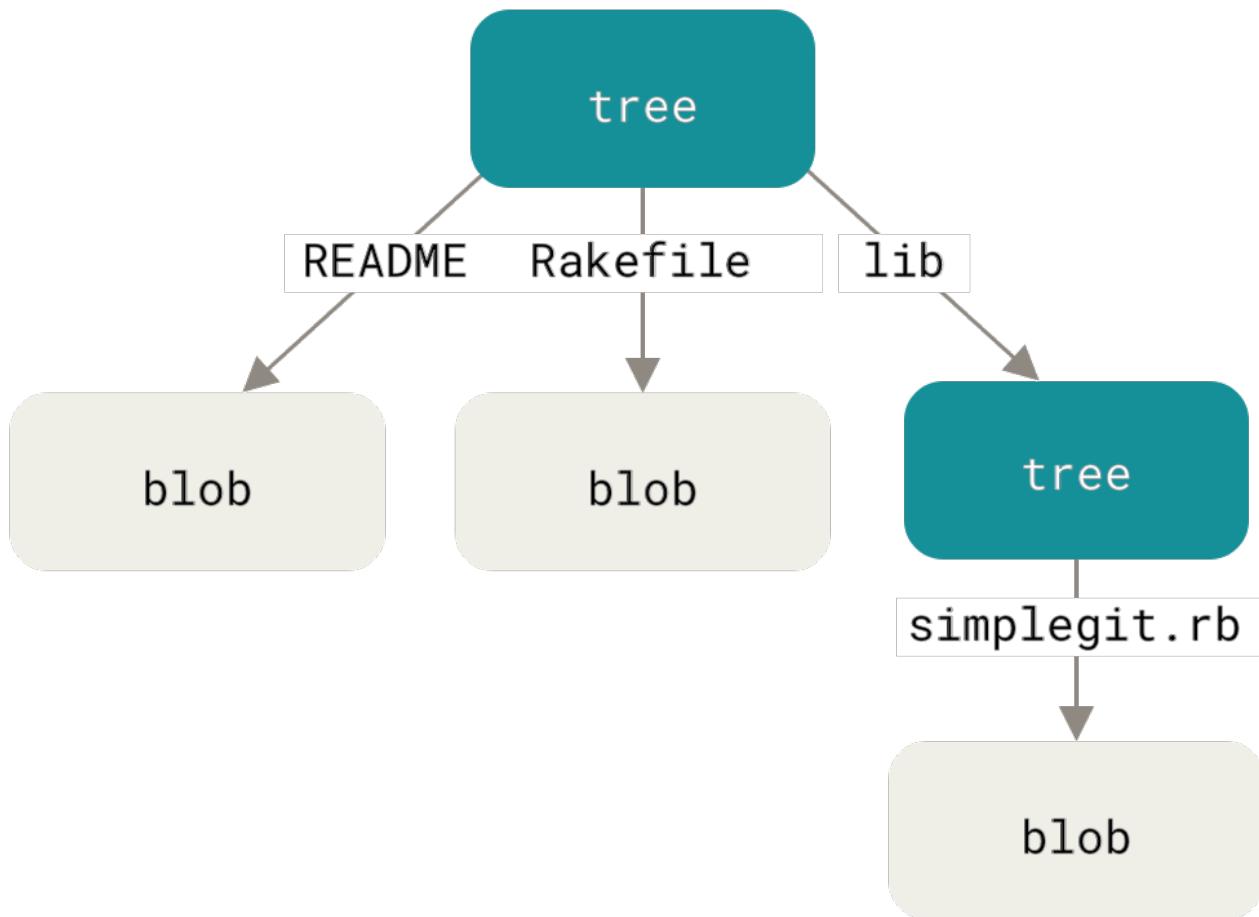


Figure 173. Einfache Version des Git-Datenmodells

Sie können ziemlich einfach Ihren eigenen Baum erstellen. Git erstellt normalerweise einen Baum, indem es den Status Ihres Staging-Bereichs oder Index übernimmt und eine Reihe von Baumobjekten daraus schreibt. Um ein Baumobjekt zu erstellen, müssen Sie zunächst einen Index einrichten, indem Sie einige Dateien bereitstellen. Um einen Index mit einem einzigen Eintrag zu erstellen — der ersten Version Ihrer `test.txt` Datei — können Sie den Basisbefehl `git update-index` verwenden. Mit diesem Befehl fügen Sie die frühere Version der Datei `test.txt` künstlich einem neuen Staging-Bereich hinzu. Sie müssen die Option `--add` übergeben, da die Datei in Ihrem Staging-Bereich noch nicht vorhanden ist (Sie haben noch nicht einmal einen Staging-Bereich eingerichtet). `--cacheinfo` müssen sie angeben, weil die hinzugefügte Datei sich nicht in Ihrem Verzeichnis, sondern in Ihrer Datenbank befindet. Dann geben Sie den Modus, SHA-1 und Dateinamen an:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

In diesem Fall geben Sie den Modus `100644` an, was bedeutet, dass es sich um eine normale Datei handelt. Andere Optionen sind `100755`, was bedeutet, dass es sich um eine ausführbare Datei handelt und `120000`, was einen symbolischen Link angibt. Der Modus stammt aus normalen UNIX-Modi, ist jedoch viel weniger flexibel — diese drei Modi sind die einzigen, die für Dateien (Blobs) in Git gültig sind (obwohl andere Modi für Verzeichnisse und Submodule verwendet werden).

Jetzt können Sie `git write-tree` verwenden, um den Staging-Bereich in ein Baumobjekt zu

schreiben. Es ist keine Option `-w` erforderlich. Durch Aufrufen dieses Befehls wird automatisch ein Baumobjekt aus dem Status des Index erstellt, wenn dieser Baum noch nicht vorhanden ist:

```
$ git write-tree  
d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Sie können auch überprüfen, ob es sich um ein Baumobjekt handelt, indem Sie denselben Befehl `git cat-file` verwenden, den Sie zuvor gesehen haben:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
tree
```

Sie erstellen jetzt einen neuen Baum mit der zweiten Version von `test.txt` und einer neuen Datei:

```
$ echo 'new file' > new.txt  
$ git update-index --cacheinfo 100644 \  
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt  
$ git update-index --add new.txt
```

Ihr Staging-Bereich enthält jetzt die neue Version von `test.txt` sowie die neue Datei `new.txt`. Schreiben Sie diesen Baum aus (zeichnen Sie den Status des Staging-Bereichs oder des Index für ein Baumobjekt auf) und sehen Sie, wie er aussieht:

```
$ git write-tree  
0155eb4229851634a0f03eb265b69f5a2d56f341  
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341  
100644 blob fa49b077972391ad58037050f2a75f74e3671e92 new.txt  
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a test.txt
```

Beachten Sie, dass dieser Baum beide Dateieinträge enthält und dass der SHA-1 von `test.txt` der SHA-1 „Version 2“ von früher (`1f7a7a`) entspricht. Aus Spaß fügen Sie diesem den ersten Baum als Unterverzeichnis hinzu. Sie können Bäume in Ihren Staging-Bereich einlesen, indem Sie `git read-tree` aufrufen. In diesem Fall können Sie einen vorhandenen Baum als Teilbaum in Ihren Staging-Bereich einlesen, indem Sie die Option `--prefix` mit diesem Befehl verwenden:

```
$ 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
```

Wenn Sie aus dem soeben erstellten Baum ein Arbeitsverzeichnis erstellen, erhalten Sie die beiden Dateien in der obersten Ebene des Arbeitsverzeichnisses und ein Unterverzeichnis mit dem Namen `bak`, das die erste Version der Datei `test.txt` enthält. Sie können sich die Daten, die Git für diese Strukturen enthält, so vorstellen:

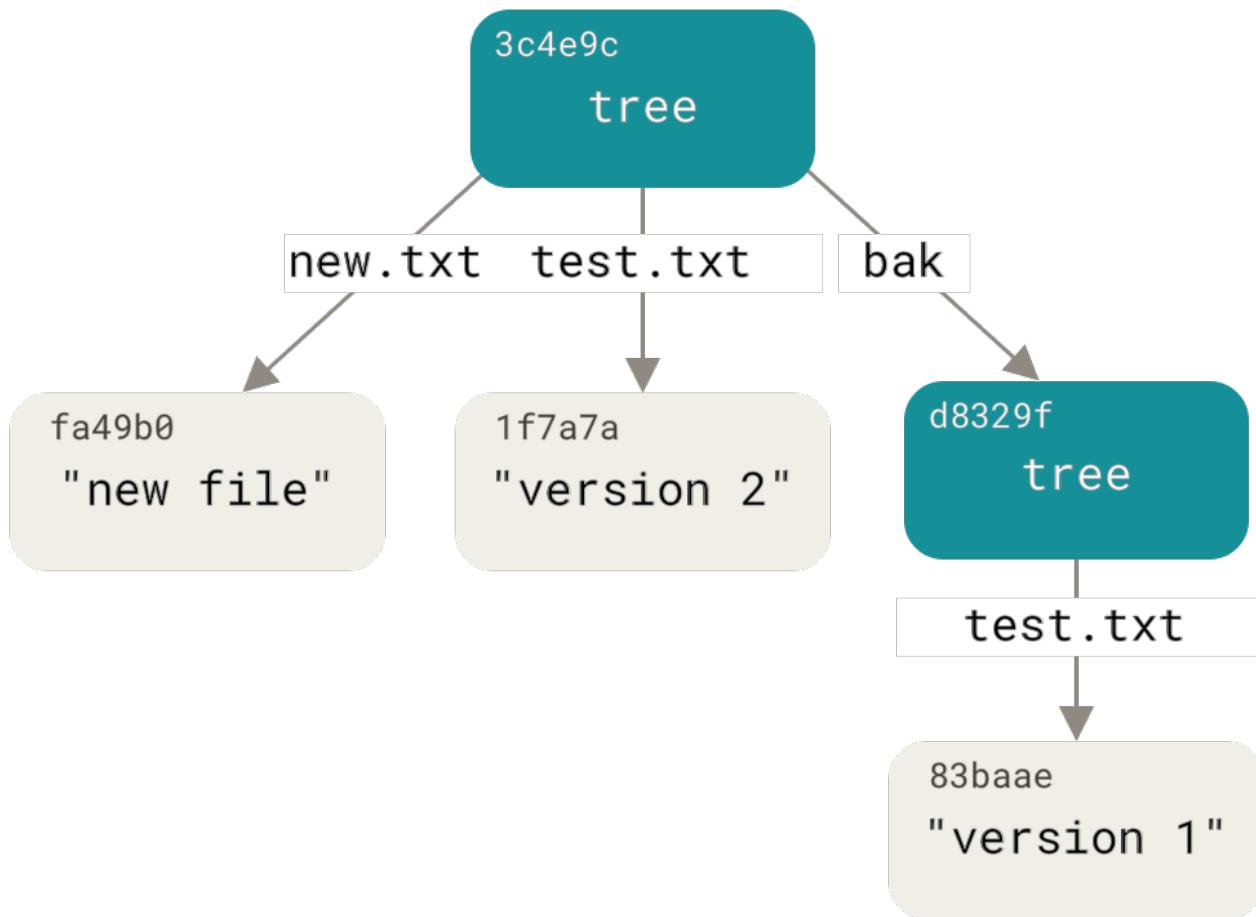


Figure 174. Die Inhaltsstruktur Ihrer aktuellen Git-Daten

## Commit Objekte

Wenn Sie alle oben genannten Schritte ausgeführt haben, verfügen Sie nun über drei Bäume, die die verschiedenen Schnappschüsse Ihres Projekts darstellen, die Sie verfolgen möchten. Das bisherige Problem bleibt jedoch bestehen: Sie müssen sich alle drei SHA-1-Werte merken, um die Schnappschüsse abzurufen. Sie haben auch keine Informationen darüber, wer die Schnappschüsse gespeichert hat, wann sie gespeichert wurden oder warum sie gespeichert wurden. Dies sind die grundlegenden Informationen, die das Commit Objekt für Sie speichert.

Um ein Commit Objekt zu erstellen, rufen Sie `commit-tree` auf und geben einen einzelnen Baum SHA-1 an. Außerdem geben sie an welche Commit Objekte die direkten Vorgänger sind (falls überhaupt welche vorhanden sind). Beginnen Sie mit dem ersten Baum, den Sie geschrieben haben:

```
$ echo 'First commit' | git commit-tree d8329f  
fdf4fc3344e67ab068f836878b6c4951e3b15f3d
```



Sie werden vermutlich einen anderen Hash-Wert erhalten, da die Erstellungszeit und die Autorendaten unterschiedlich sind. Darüber hinaus kann zwar prinzipiell jedes Commit-Objekt genau reproduziert werden. Allerdings könnten aufgrund der technischen Details beim Erstellen dieses Buches die abgedruckten Commit-Hashes im Vergleich zu den entsprechenden Commits abweichen. Ersetzen Sie die Commit- und Tag-Hashes durch Ihre eigenen Prüfsummen in diesem Kapitel.

Nun können Sie sich Ihr neues Commit-Objekt mit `git cat-file` ansehen:

```
$ 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

Das Format für ein Commit Objekt ist einfach: Es gibt den Baum der obersten Ebene für den Snapshot des Projekts zu diesem Zeitpunkt an; Das übergeordnete Objekt, falls vorhanden, (das oben beschriebene Commit Objekt hat keine übergeordneten Objekte); die Autoren-/Committer-Informationen (genutzt werden Ihre Konfigurationseinstellungen `user.name` und `user.email` sowie einen Zeitstempel); eine leere Zeile und dann die Commit-Nachricht.

Als Nächstes schreiben Sie die beiden anderen Commit Objekte, die jeweils auf das Commit verweisen, welches direkt davor erfolgte:

```
$ echo 'Second commit' | git commit-tree 0155eb -p fdf4fc3
cac0cab538b970a37ea1e769cbbde608743bc96d
$ echo 'Third commit' | git commit-tree 3c4e9c -p cac0cab
1a410efbd13591db07496601ebc7a059dd55cfe9
```

Jedes der drei Commit-Objekte verweist auf einen der drei von Ihnen erstellten Snapshot-Bäume. Seltsamerweise haben Sie jetzt einen echten Git-Verlauf, den Sie mit dem Befehl `git log` anzeigen können, wenn Sie ihn beim letzten Commit SHA-1 ausführen:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700
```

Third commit

```
bak/test.txt | 1 +
1 file changed, 1 insertion(+)
```

```
commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700
```

Second commit

```
new.txt | 1 +  
test.txt | 2 +-  
2 files changed, 2 insertions(+), 1 deletion(-)
```

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d

Author: Scott Chacon <schacon@gmail.com>

Date: Fri May 22 18:09:34 2009 -0700

First commit

```
test.txt | 1 +  
1 file changed, 1 insertion(+)
```

Wundervoll. Sie haben gerade Basisbefehle der niedrigen Ebene ausgeführt, um einen Git-Verlauf zu erstellen, ohne einen der Standardbefehle zu verwenden. Dies ist im Wesentlichen das, was Git tut, wenn Sie die Befehle `git add` und `git commit` ausführen — es speichert Blobs für die geänderten Dateien, aktualisiert den Index, schreibt Bäume und schreibt Commit-Objekte, die auf Bäume der oberste Ebene verweisen und die Commits, die unmittelbar vor ihnen aufgetreten sind. Diese drei Haupt-Git-Objekte — der Blob, der Baum und das Commit — werden anfänglich als separate Dateien in Ihrem `.git/objects` Verzeichnis gespeichert. Hier sind jetzt alle Objekte im Beispielverzeichnis, kommentiert mit dem, was sie speichern:

```
$ find .git/objects -type f  
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2  
.git/objects/1a/410efbd13591db0749601ebc7a059dd55cfe9 # commit 3  
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2  
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3  
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1  
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2  
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'  
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1  
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt  
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Wenn Sie allen internen Zeigern folgen, erhalten Sie eine Objektgrafik in etwa wie folgt:

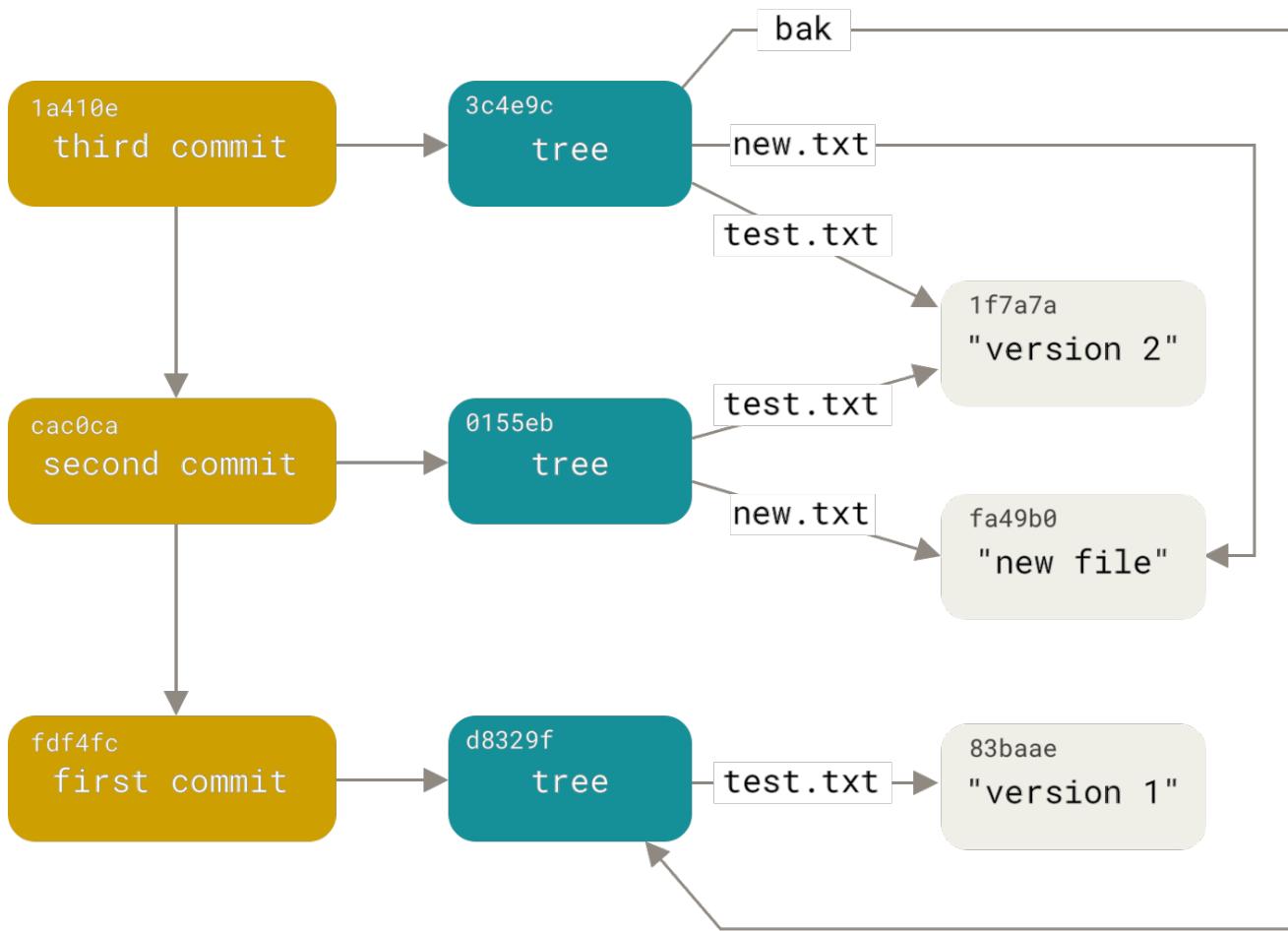


Figure 175. Alle erreichbaren Objekte in Ihrem Git-Verzeichnis

## Objektspeicher

Wir haben bereits erwähnt, dass für jedes Objekt, das Sie in Ihre Git-Objektdatenbank übernehmen, ein Header gespeichert ist. Nehmen wir uns eine Minute Zeit, um zu sehen, wie Git seine Objekte speichert. Sie werden sehen, wie Sie ein Blob-Objekt — in diesem Fall die Zeichenfolge „what is up, doc?“ — interaktiv in der Ruby-Skriptsprache speichern.

Sie können den interaktiven Ruby-Modus mit dem Befehl `irb` starten:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git erstellt zuerst einen Header, der mit der Identifizierung des Objekttyps beginnt — in diesem Fall ein Blob. Zu diesem ersten Teil des Headers fügt Git ein Leerzeichen hinzu, gefolgt von der Größe des Inhalts in Bytes, und fügt ein letztes Nullbyte hinzu:

```
>> header = "blob #{content.bytesize}\0"
=> "blob 16\0000"
```

Git verkettet den Header und den ursprünglichen Inhalt und berechnet dann die SHA-1-Prüfsumme

dieses neuen Inhalts. Sie können den SHA-1-Wert einer Zeichenfolge in Ruby berechnen, indem Sie die SHA1-Digest-Bibliothek mit dem Befehl `require` hinzufügen und dann `Digest::SHA1hexdigest()` mit der Zeichenfolge aufrufen:

```
>> store = header + content  
=> "blob 16\u0000what is up, doc?"  
>> require 'digest/sha1'  
=> true  
>> sha1 = Digest::SHA1hexdigest(store)  
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Vergleichen wir dies mit der Ausgabe von `git hash-object`. Hier verwenden wir `echo -n`, um zu verhindern, dass der Eingabe eine neue Zeile hinzugefügt wird.

```
$ echo -n "what is up, doc?" | git hash-object --stdin  
bd9dbf5aae1a3862dd1526723246b20206e5fc37
```

Git komprimiert den neuen Inhalt mit zlib, was Sie in Ruby mit der zlib-Bibliothek tun können. Zuerst müssen Sie die Bibliothek hinzufügen und dann `Zlib::Deflate.deflate()` auf den Inhalt ausführen:

```
>> require 'zlib'  
=> true  
>> zlib_content = Zlib::Deflate.deflate(store)  
=> "x\x9C\xCA\xC90R04c(\xC9H,Q\xC8,V(-\xD0QH\xC90\xB6\aa\x00_\x1C\aa\x9D"
```

Schließlich schreiben Sie Ihren zlib-entpackten Inhalt auf ein Objekt auf der Festplatte. Sie bestimmen den Pfad des Objekts, das Sie ausschreiben möchten (die ersten beiden Zeichen des SHA-1-Werts sind der Name des Unterverzeichnisses und die letzten 38 Zeichen der Dateiname in diesem Verzeichnis). In Ruby können Sie die Funktion `FileUtils.mkdir_p()` verwenden, um das Unterverzeichnis zu erstellen, falls es nicht vorhanden ist. Öffnen Sie dann die Datei mit `File.open()` und schreiben Sie den zuvor mit zlib komprimierten Inhalt mit einem `write()`-Aufruf auf das resultierende Dateihandle in die Datei:

```
>> 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
```

Lassen Sie uns den Inhalt des Objekts mit `git cat-file` überprüfen:

```
---
```

```
$ git cat-file -p bd9dbf5aae1a3862dd1526723246b20206e5fc37
what is up, doc?
---
```

Das war's – Sie haben ein gültiges Git-Blob-Objekt erstellt.

Alle Git-Objekte werden auf dieselbe Weise gespeichert, nur mit unterschiedlichen Typen. Anstelle des String-Blobs beginnt der Header mit commit oder tree. Auch wenn der Blob-Inhalt nahezu beliebig sein kann, sind Commit- und Tree-Inhalt sehr spezifisch formatiert.

## Git Referenzen

Wenn Sie den Verlauf Ihres Repositorys sehen möchten, der über Commit erreichbar ist, z. B. **1a410e**, können Sie so etwas wie `git log 1a410e` ausführen, um diesen Verlauf anzuzeigen. Dennoch müssen Sie sich weiterhin daran erinnern, dass **1a410e** der Commit ist den Sie als Ausgangspunkt für diese Historie verwenden möchten. Es wäre aber einfacher, wenn Sie eine Datei hätten, in der Sie diesen SHA-1-Wert unter einem einfachen Namen speichern könnten, sodass Sie diesen einfachen Namen anstelle des unformatierten SHA-1-Werts verwenden könnten.

In Git werden diese einfachen Namen „Referenzen“ oder „Refs“ genannt. Sie finden die Dateien, die diese SHA-1-Werte enthalten, im Verzeichnis `.git/refs`. Im aktuellen Projekt enthält dieses Verzeichnis keine Dateien, es enthält eine einfache Struktur:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

Um eine neue Referenz zu erstellen, die Ihnen hilft, sich zu erinnern, wo sich Ihr letztes Commit befindet, können Sie einfach folgende machen:

```
$ echo 1a410efbd13591db07496601ebc7a059dd55cfe9 > .git/refs/heads/master
```

Jetzt können Sie die soeben erstellte Kopfreferenz anstelle des SHA-1-Werts in Ihren Git-Befehlen verwenden:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Es wird nicht empfohlen, die Referenzdateien direkt zu bearbeiten. Stattdessen bietet Git den sichereren **Befehl git update-ref**, um dies zu tun, wenn Sie eine Referenz aktualisieren möchten:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

Das ist im Grunde genommen ein Branch in Git: ein einfacher Zeiger oder ein Verweis auf den Kopf einer Arbeitslinie. So erstellen Sie eine Verzweigung beim zweiten Commit:

```
$ git update-ref refs/heads/test cac0ca
```

Ihr Branch enthält nur Arbeiten von diesem Commit an abwärts:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Nun sieht Ihre Git-Datenbank konzeptionell ungefähr so aus:

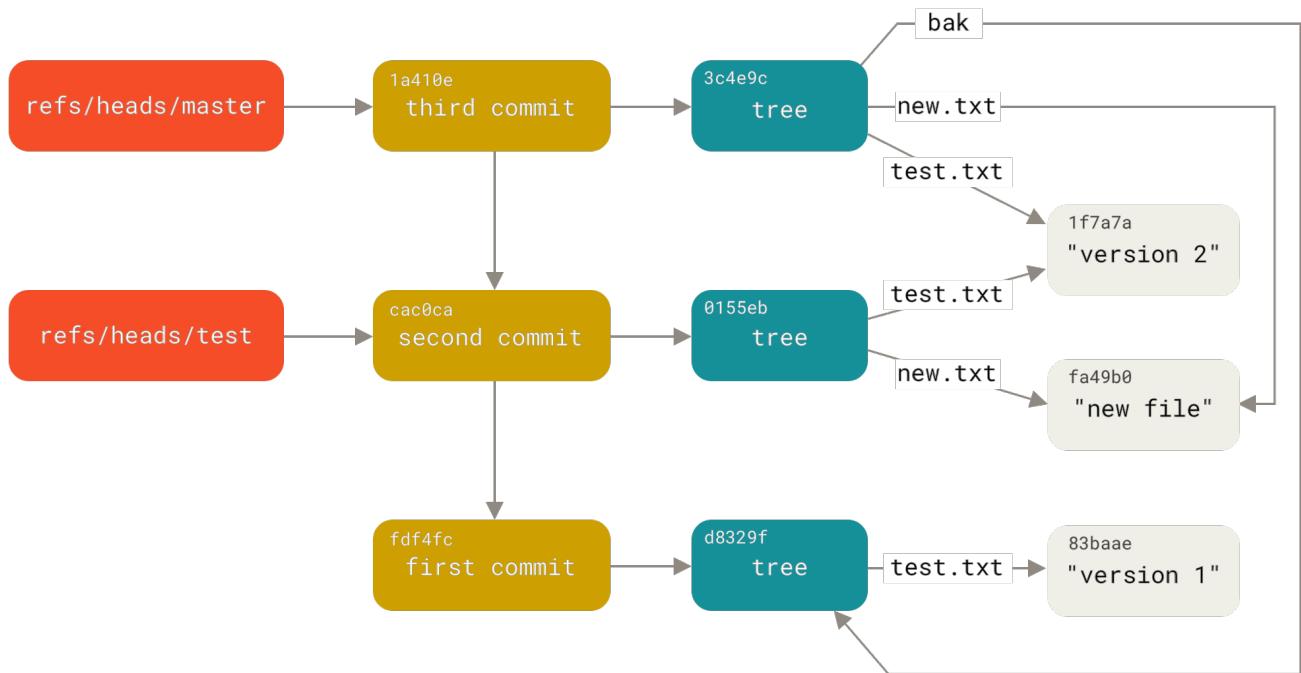


Figure 176. Git-Verzeichnisobjekte mit Branch Head Referenzen

Wenn Sie Befehle wie `git branch <branch>` ausführen, führt Git grundsätzlich den Befehl `update-ref` aus, um den SHA-1 des letzten Commits des Branches, in dem Sie sich befinden, in die neue Referenz einzufügen, die Sie erstellen möchten.

## HEAD

Die Frage ist nun, wenn Sie `git branch <branch>` ausführen, woher kennt Git den SHA-1 des letzten Commits? Die Antwort ist die HEAD-Datei.

Normalerweise ist die HEAD-Datei ein symbolischer Verweis auf den Branch, in dem Sie sich gerade befinden. Mit symbolischer Referenz meinen wir, dass sie im Gegensatz zu einer normalen Referenz einen Zeiger auf eine andere Referenz enthält.

In einigen seltenen Fällen kann die HEAD-Datei jedoch den SHA-1-Wert eines Git-Objekts enthalten. Dies geschieht beim Auschecken eines Tags, Commits oder eines Remote-Banches, wodurch Ihr Repository in den Status "[detached HEAD](#)" versetzt wird.

Wenn Sie sich die Datei ansehen, sehen Sie normalerweise Folgendes:

```
$ cat .git/HEAD  
ref: refs/heads/master
```

Wenn Sie `git checkout test` ausführen, aktualisiert Git die Datei folgendermaßen:

```
$ cat .git/HEAD  
ref: refs/heads/test
```

Wenn Sie `git commit` ausführen, wird das Commitobjekt erstellt, wobei das übergeordnete Objekt dieses Commitobjekts als der SHA-1-Wert angegeben wird, auf den die Referenz in HEAD verweist.

Sie können diese Datei auch manuell bearbeiten, es gibt jedoch wieder einen sichereren Befehl: `git symbolic-ref`. Sie können den Wert Ihres HEAD über diesen Befehl lesen:

```
$ git symbolic-ref HEAD  
refs/heads/master
```

Sie können den Wert von HEAD auch mit demselben Befehl festlegen:

```
$ git symbolic-ref HEAD refs/heads/test  
$ cat .git/HEAD  
ref: refs/heads/test
```

Sie können keine symbolische Referenz außerhalb des Refs-Stils festlegen:

```
$ git symbolic-ref HEAD test  
fatal: Refusing to point HEAD outside of refs/
```

## Tags

Wir haben gerade die drei Hauptobjekttypen von Git (*blobs*, *trees* und *commits*) besprochen, aber es gibt einen vierten. Das *tag*-Objekt ähnelt stark einem Commitobjekt — es enthält einen Tagger, ein Datum, eine Nachricht und einen Zeiger. Der Hauptunterschied besteht darin, dass ein Tag-Objekt im Allgemeinen eher auf ein Commit als auf einen Baum verweist. Es ist wie eine Branchreferenz, aber es bewegt sich nie — es zeigt immer auf das gleiche Commit, gibt ihm aber einen lesbareren Namen.

Wie in [Git Grundlagen](#) beschrieben, gibt es zwei Arten von Tags: Annotierte- und Leichtgewichtige-

Tags. Sie können einen leichtgewichtigen Tag erstellen, indem Sie Folgendes ausführen:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Das ist alles, was ein leichtgewichtiger Tag ist — eine Referenz, die sich nie bewegt. Ein annotiertes Tag ist jedoch komplexer. Wenn Sie ein annotiertes Tag erstellen, erstellt Git ein Tag-Objekt und schreibt dann einen Verweis, um darauf zu zeigen, anstatt direkt auf das Commit. Sie können dies sehen, indem Sie ein annotiertes Tag (mit der Option `-a`) erstellen:

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'Test tag'
```

Hier ist der Wert für das Objekt SHA-1, das erstellt wurde:

```
$ cat .git/refs/tags/v1.1  
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Führen Sie nun `git cat-file -p` für diesen SHA-1-Wert aus:

```
$ 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
```

Beachten Sie, dass der Objekteintrag auf den Commit SHA-1-Wert verweist, den Sie getagged haben. Beachten Sie auch, dass es nicht auf ein Commit verweisen muss. Sie können jedes Git-Objekt taggen. Beispielsweise hat der Betreuer im Git-Quellcode seinen öffentlichen GPG-Schlüssel als Blob-Objekt hinzugefügt und dann mit Tags versehen. Sie können den öffentlichen Schlüssel anzeigen, indem Sie diesen in einem Klon des Git-Repositorys ausführen:

```
$ git cat-file blob junio-gpg-pub
```

Das Linux-Kernel-Repository verfügt auch über ein Tag-Objekt, das nicht auf Commits verweist. Das erste erstellte Tag verweist auf den ursprünglichen Baum des Imports des Quellcodes.

## Remotes

Der dritte Referenztyp, den Sie sehen, ist eine Remoterefenz. Wenn Sie ein Remote hinzufügen und darauf pushen, speichert Git den Wert, den Sie zuletzt an diesen Remote gesendet haben, für jeden Branch im Verzeichnis `refs/remotes`. Zum Beispiel können Sie eine Remote mit dem Namen `origin` hinzufügen und Ihren `master` -Branch darauf 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
```

Anschließend können Sie in der Datei `refs/remotes/origin/master` sehen, in welcher `master` Branch auf dem `origin` Remote Sie das letzte Mal mit dem Server kommuniziert haben:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Remote Referenzen unterscheiden sich von Branches (`refs/heads` Referenzen) hauptsächlich darin, dass sie als schreibgeschützt gelten. Sie können `git checkout` darauf ausführen, aber HEAD wird nicht symbolisch darauf referenzieren, so dass Sie es niemals mit einem `commit` Befehl aktualisieren können. Git verwaltet sie als Lesezeichen für den letzten bekannten Status, in dem sich diese Branches auf diesen Servern befinden.

## Packdateien (engl. Packfiles)

Wenn Sie alle Anweisungen im Beispiel aus dem vorherigen Abschnitt befolgt haben, sollten Sie jetzt ein Test-Git-Repository mit 11 Objekten haben – vier Blobs, drei Bäumen, drei Commits und einem 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 komprimiert den Inhalt dieser Dateien mit zlib. Alle diese Dateien zusammen belegen nur 925 Byte. Jetzt fügen Sie dem Repository einige größere Inhalte hinzu, um eine interessante Funktion von Git zu demonstrieren. Zur Veranschaulichung fügen wir die Datei `repo.rb` aus der Grit-Bibliothek hinzu. Hierbei handelt es sich um eine 22-KB-Quellcodedatei:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git checkout master
$ git add repo.rb
$ git commit -m 'Create repo.rb'
[master 484a592] Create repo.rb
 3 files changed, 709 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Wenn Sie sich den resultierenden Baum ansehen, sehen Sie den SHA-1-Wert, der für Ihr neues repo.rb-Blob-Objekt berechnet wurde:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Sie können dann `git cat-file` verwenden, um zu sehen, wie groß das Objekt ist:

```
$ git cat-file -s 033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5
22044
```

Ändern Sie nun diese Datei ein wenig und sehen Sie, was passiert:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'Modify repo.rb a bit'
[master 2431da6] Modify repo.rb a bit
 1 file changed, 1 insertion(+)
```

Überprüfen Sie den von diesem letzten Commit erstellten Baum. Dabei werden Sie etwas Interessantes sehen:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob b042a60ef7dff760008df33cee372b945b6e884e      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Der Blob ist jetzt ein anderer Blob. Das bedeutet, dass Git, obwohl Sie nur eine einzelne Zeile am Ende einer Datei mit 400 Zeilen hinzugefügt haben, diesen neuen Inhalt als ein komplett neues Objekt gespeichert hat:

```
$ git cat-file -s b042a60ef7dff760008df33cee372b945b6e884e
```

Sie haben jetzt zwei nahezu identische 22-KB-Objekte auf Ihrer Festplatte (jedes auf ca. 7 KB komprimiert). Wäre es nicht schön, wenn Git eines davon vollständig speichern könnte, aber dann das zweite Objekt nur als Delta zwischen dem ersten und dem anderen?

Wie sich herausstellen wird, geht das. Das ursprüngliche Format, in dem Git Objekte auf der Festplatte speichert, wird als „loses“ Objektformat bezeichnet. Allerdings packt Git gelegentlich mehrere dieser Objekte in eine einzige Binärdatei namens „packfile“, um Platz zu sparen und effizienter zu sein. Git tut dies, wenn Sie zu viele lose Objekte haben, wenn Sie den Befehl `git gc` manuell ausführen oder wenn Sie einen Push an einen Remote-Server senden. Um zu sehen, was passiert, können Sie Git manuell auffordern, die Objekte zu packen, indem Sie den Befehl `git gc` aufrufen:

```
$ 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)
```

Wenn Sie in Ihr `objects` Verzeichnis schauen, werden Sie feststellen, dass die meisten Ihrer Objekte verschwunden sind und ein paar neue Dateien auftauchen:

```
$ 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
```

Die verbleibenden Objekte sind die Blobs, auf die von keinem Commit referenziert werden. In diesem Fall die Blobs „what is up, doc?“ Und „test content“, die Sie zuvor erstellt haben. Da Sie sie nie zu Commits hinzugefügt haben, gelten sie als unreferenziert und sind nicht in Ihrem neuen Packfile gepackt.

Die anderen Dateien sind Ihr neues packfile und ein Index. Das packfile ist eine einzelne Datei, die den Inhalt aller Objekte enthält, die aus Ihrem Dateisystem entfernt wurden. Der Index ist eine Datei, die Offsets in diesem packfile enthält, sodass Sie schnell nach einem bestimmten Objekt suchen können. Ein weiterer Vorteil ist, dass, obwohl die Objekte auf der Festplatte vor dem Ausführen des Befehls `gc` insgesamt etwa 15 KB groß waren, die neue Paketdatei nur 7 KB groß ist. Sie haben die Festplattennutzung halbiert, indem Sie Ihre Objekte gepackt haben.

Wie macht Git das? Wenn Git Objekte packt, sucht es nach Dateien mit ähnlichen Namen und Größen und speichert nur die Deltas von einer Version der Datei zur nächsten. Sie können im packfile schauen und sehen, was Git getan hat, um Platz zu sparen. Mit dem Befehl `git verify-pack` können Sie sehen, was gepackt wurde:

```
$ git verify-pack -v .git/objects/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/978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Hier verweist der Blob **033b4**, der, wenn Sie sich erinnern, die erste Version Ihrer `repo.rb` Datei war, auf den Blob **b042a**, der die zweite Version der Datei war. Die dritte Spalte in der Ausgabe ist die Größe des Objekts im Paket. Sie können also sehen, dass **b042a** 22 KB der Datei belegt, **033b4** jedoch nur 9 Byte. Interessant ist auch, dass die zweite Version der Datei als Ganzes gespeichert wird, während die Originalversion als Delta gespeichert wird. Dies liegt daran, dass Sie mit größter Wahrscheinlichkeit einen schnelleren Zugriff auf die neueste Version der Datei benötigen.

Das Schöne daran ist, dass es jederzeit umgepackt werden kann. Git packt Ihre Datenbank gelegentlich automatisch neu und versucht dabei immer, mehr Speicherplatz zu sparen. Sie können das Packen jedoch auch jederzeit manuell durchführen, indem Sie `git gc` manuell ausführen.

## Die Referenzspezifikation (engl. Refspec)

In diesem Buch haben wir simple Zuordnungen von remote Branches zu lokalen Referenzen verwendet, diese können jedoch komplexer sein. Angenommen, Sie haben die letzten Abschnitte mitverfolgt und ein kleines lokales Git-Repository erstellt, und möchten nun eine `remote` hinzufügen:

```
$ git remote add origin https://github.com/schacon/simplegit-progit
```

Durch Ausführen des obigen Befehls wird ein Abschnitt zur `.git/config` Datei Ihres Repositorys hinzugefügt. Es wird der Name des Remote-Repositorys (`origin`), die URL des Remote-Repositorys und die `refspec` angegeben sind, die zum Abrufen verwendet werden soll:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Das Format der Refspec ist zunächst ein optionales `', gefolgt von '<src>:<dst>'`, wobei `'<src>'` das Muster für Referenzen auf der Remote-Seite ist. `'<dst>'` gibt an wo diese Referenzen lokal nachverfolgt werden. Das `'` weist Git an, die Referenz zu aktualisieren, auch wenn es sich nicht um einen fast-forward handelt.

In der Standardeinstellung, die automatisch von einem Befehl `git remote add origin` geschrieben wird, ruft Git alle Referenzen unter `refs/heads/` auf dem Server ab und schreibt sie lokal in `refs/remotes/origin/`. Wenn sich auf dem Server also ein `master` Branch befindet, können Sie auf das Log dieses Branches lokal zugreifen, indem Sie eine der folgenden Aktionen ausführen:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Sie sind alle gleichwertig, da Git sie zu `refs/remotes/origin/master` erweitert.

Wenn Git stattdessen jedes Mal nur den `master` Branch und nicht jeden anderen Branch auf dem Remote-Server pullen soll, können Sie die Abrufzeile so ändern, dass sie nur auf diesen Branch verweist:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Dies ist nur die Standard Refspec für `git fetch` für diesen Remote. Wenn Sie einen einmaligen Abruf durchführen möchten, können Sie die spezifische Refspec auch in der Befehlszeile angeben. Um den `master` Branch auf dem Remote lokal nach `origin/mymaster` zu pullen, können Sie Folgendes ausführen:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Sie können auch mehrere Refspecs angeben. In der Befehlszeile können Sie mehrere Branches wie folgt 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 diesem Fall wurde der `master` Branch pull abgelehnt, da er nicht als Fast-Forward aufgeführt war. Sie können dies überschreiben, indem Sie das `+` vor der Refspec angeben.

Sie können auch mehrere Refspecs zum Abrufen in Ihrer Konfigurationsdatei angeben. Wenn Sie immer die Branches `master` und `experiment` vom `origin` Remote abrufen möchten, fügen Sie zwei Zeilen hinzu:

```
[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
```

Seit Git 2.6.0 können Sie partielle Globs in Pattern verwenden, um mehrere Branches anzupassen, dann funktioniert das hier:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Noch besser, Sie können Namespaces (oder Verzeichnisse) verwenden, um dasselbe mit mehr Struktur zu erreichen. Wenn Sie ein QS-Team haben, das eine Reihe von Branches pusht, und Sie möchten nur den `master` Branch und einen der Branches des QS-Teams erhalten, dann können Sie einen Konfigurationsabschnitt wie diesen verwenden:

```
[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/*
```

Wenn Sie über einen komplexen Workflow-Prozess verfügen, bei dem QS-Team und Entwickler Branches pushen und Integrationsteams auf Remotebranches pushen bzw. daran zusammenarbeiten, können Sie sie auf diese Weise problemlos mit Namespaces versehen.

## Pushende Refspecs

Es ist schön, dass Sie auf diese Weise Referenzen mit Namespaces abrufen können, aber wie bringt das QS-Team seine Branches überhaupt an die erste Stelle eines Namespace `qa/`? Sie erreichen dies, indem Sie Refspecs zum Pushen verwenden.

Wenn das QS-Team seinen `master` Branch auf `qa/master` auf dem Remote-Server verschieben möchte, kann folgendes ausgeführt werden:

```
$ git push origin master:refs/heads/qa/master
```

Wenn Git dies bei jedem Start von `git push origin` automatisch ausführen soll, können Sie ihrer

Konfigurationsdatei einen **push** Wert hinzufügen:

```
[remote "origin"]
  url = https://github.com/schacon/simplegit-progit
  fetch = +refs/heads/*:refs/remotes/origin/*
  push = refs/heads/master:refs/heads/qa/master
```

Dies wird wiederum dazu führen, dass ein **git push origin** den lokalen **master** Branch standardmäßig zum remote **qa/master** Branch pusht.



Sie können die Refspec nicht zum Abrufen von einem Repository und zum Verschieben auf ein anderes Repository verwenden. Ein Beispiel wie das geht finden Sie unter [Dein öffentliches GitHub-Repository aktuell halten](#).

## Löschen von Referenzen

Sie können mit Refspec auch Verweise vom Remote-Server löschen, indem Sie Folgendes ausführen:

```
$ git push origin :topic
```

Die Syntax der Refspezifikation lautet **<src>:<dst>**. Das Weglassen des **<src>** Teils bedeutet, im Grunde genommen, dass der **topic** Branch auf dem Remote leer bleibt, wodurch er gelöscht wird.

Oder Sie können die neuere Syntax verwenden (verfügbar seit Git v1.7.0):

```
$ git push origin --delete topic
```

## Transfer Protokolle

Git kann Daten zwischen zwei Repositorys hauptsächlich auf zwei Arten übertragen: mittels dem „dummen“ Protokoll und dem „intelligenten“ Protokoll. In diesem Abschnitt wird kurz erläutert, wie diese beiden Hauptprotokolle funktionieren.

### Das dumme Protokoll

Wenn Sie ein Repository einrichten, das schreibgeschützt über HTTP bereitgestellt wird, wird wahrscheinlich das dumme Protokoll verwendet. Dieses Protokoll wird als „dumm“ bezeichnet, da es während des Transportvorgangs keinen Git-spezifischen Code auf der Serverseite erfordert. Der Abrufprozess besteht aus einer Reihe von HTTP **GET** Abfragen, bei denen der Client das Layout des Git-Repositorys auf dem Server übernehmen kann.



Das dumme Protokoll wird heutzutage ziemlich selten verwendet. Es ist schwierig, es sicher oder privat einzurichten, daher lehnen die meisten Git-Hosts (sowohl cloudbasiert als auch on-premise) die Verwendung ab. Es wird generell empfohlen, das smarte Protokoll zu verwenden, welches wir weiter unten

beschreiben.

Folgen wir dem **http-fetch** Prozess für die simplegit-Bibliothek:

```
$ git clone http://server/simplegit-progit.git
```

Das erste, was dieser Befehl tut, ist das pullen der Datei **info/refs**. Diese Datei wird mit dem Befehl **update-server-info** geschrieben. Aus diesem Grund müssen Sie diesen als **post-receive** Hook aktivieren, damit der HTTP-Transport ordnungsgemäß funktioniert:

```
=> GET info/refs  
ca82a6dff817ec66f44342007202690a93763949      refs/heads/master
```

Jetzt haben Sie eine Liste der remote Referenzen und der SHA-1s. Als Nächstes suchen Sie nach der HEAD-Referenz, damit Sie wissen, was Sie abschließend überprüfen müssen:

```
=> GET HEAD  
ref: refs/heads/master
```

Sie müssen den 'master'-Branch auschecken, wenn Sie den Vorgang abgeschlossen haben. Jetzt können Sie mit dem laufenden Prozess beginnen. Da Ihr Ausgangspunkt das Commit-Objekt **ca82a6** ist, das Sie in der Datei **info/refs** gesehen haben, rufen Sie zunächst Folgendes ab:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949  
(179 bytes of binary data)
```

Sie erhalten ein Objekt zurück – das Objekt befindet sich in losem Format auf dem Server und Sie haben es über einen statischen HTTP-GET-Aufruf abgerufen. Sie können es zlib-dekomprimieren, den Header entfernen und den Commit-Inhalt anzeigen:

```
$ git cat-file -p ca82a6dff817ec66f44342007202690a93763949  
tree cfda3bf379e4f8dba8717dee55aab78aef7f4daf  
parent 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
author Scott Chacon <schacon@gmail.com> 1205815931 -0700  
committer Scott Chacon <schacon@gmail.com> 1240030591 -0700
```

Change version number

Als nächstes müssen Sie zwei weitere Objekte abrufen – **cfda3b**, das ist der Inhaltsbaum, auf den das gerade abgerufene Commit verweist; und **085bb3**, welches das übergeordnete Commit ist:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
(179 bytes of data)
```

Damit haben Sie Ihr nächstes Commit-Objekt. Holen Sie sich nun das Baumobjekt:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf  
(404 - Not Found)
```

Hoppla – es sieht so aus, als ob das Baum-Objekt auf dem Server nicht im losen Format vorliegt, sodass Sie eine 404-Antwort erhalten. Dafür gibt es mehrere Gründe: Das Objekt befindet sich möglicherweise in einem alternativen Repository oder in einem packfile Paketdatei in diesem Repository. Git sucht zuerst nach allen gelisteten Alternativen:

```
=> GET objects/info/http-alternates  
(empty file)
```

Wenn dies mit einer Liste alternativer URLs zurückgibt, sucht Git dort nach losen Dateien und packfiles. Dies ist ein nützlicher Mechanismus für Projekte, die sich gegenseitig forken, um Objekte auf der Festplatte zu teilen. Da in diesem Fall jedoch keine Alternativen aufgeführt sind, muss sich Ihr Objekt in einem packfile befinden. Um zu sehen, welche packfiles auf diesem Server verfügbar sind, müssen Sie die Datei [objects/info/packs](#) abrufen, die eine Liste dieser Dateien enthält (dies wird ebenfalls mittels [update-server-info](#) generiert):

```
=> GET objects/info/packs  
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Es gibt nur ein packfile auf dem Server, sodass sich Ihr Objekt offensichtlich dort befindet. Überprüfen Sie jedoch den Index, um dies auch sicherzustellen. Dies ist ebenfalls nützlich, wenn sich mehrere packfiles auf dem Server befinden. Sie können so prüfen, welches packfile das gewünschte Objekt enthält:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx  
(4k of binary data)
```

Nachdem Sie nun den packfile-Index haben, können Sie sehen, ob sich Ihr Objekt darin befindet. Da Index listet die SHA-1-Werte der in dem packfile enthaltenen Objekte und die Offsets zu diesen Objekten. Ihr Objekt ist vorhanden, also holen sie sich das komplette packfile:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack  
(13k of binary data)
```

Nun haben Sie Ihr Baumobjekt und gehen Ihre Commits weiter durch. Sie befinden sich alle in dem gerade heruntergeladenen packfile, sodass Sie keine weiteren Anfragen an Ihren Server stellen müssen. Git checkt eine Arbeitskopie des [master](#) Branches aus, auf die in der HEAD-Referenz verwiesen wurde, die Sie zu Beginn heruntergeladen haben.

## Das smarte Protokoll

Das dumme Protokoll ist einfach, aber ein bisschen ineffizient. Es kann keine Daten vom Client auf den Server schreiben. Das Smart-Protokoll wird oft zum Übertragen von Daten genutzt. Es erfordert jedoch einen intelligenten Git-Prozess auf der Remote-Seite. Es kann lokale Daten lesen, herausfinden, was der Client hat und benötigt, und eine benutzerdefiniertes packfile dafür generieren. Es gibt zwei Arten von Prozessen um Daten zu übertragen: zwei zum Hochladen von Daten und zwei zum Herunterladen von Daten.

## Daten hochladen

Um Daten remote hochzuladen, verwendet Git die Prozesse `send-pack` und `receive-pack`. Der `send-pack` Prozess wird auf dem Client ausgeführt und stellt eine Verbindung zu einem `receive-pack` Prozess auf der Remote-Seite her.

SSH

Angenommen, Sie führen in Ihrem Projekt `git push origin master` aus, und `origin` ist als URL definiert, die das SSH-Protokoll verwendet. Git startet den `send-pack` Prozess, der eine Verbindung über SSH zu Ihrem Server aufbaut. Es wird versucht, einen Befehl auf dem Remote-Server über einen SSH-Aufruf auszuführen, der in etwa so aussieht:

```
$ 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
```

Der Befehl `git-receive-pack` antwortet sofort mit einer Zeile für jede Referenz, die er derzeit hat – in diesem Fall nur den `master` Branch und seinen SHA-1. Die erste Zeile enthält auch eine Liste der Serverfunktionen (hier `report-status`, `delete-refs` und einige andere, einschließlich der Client-ID).

Die Daten werden in Paketen (eng. chunks) übertragen. Jeder Chunk beginnt mit einem 4-stelligen Hex-Wert, der angibt, wie lang der Chunk ist (einschließlich der 4 Bytes der Gesamtlänge). Die Pakete enthalten in der Regel eine einzige Zeile mit Daten und einen abschließenden Zeilenvorschub. Ihr erster Chunk beginnt mit 00a5, also hexadezimal für 165, womit das Paket 165 Bytes lang ist. Der nächste Chunk ist 0000, d.h. der Server ist mit seiner Referenzliste fertig.

Jetzt, da der Server-Status bekannt ist, bestimmt Ihr `send-pack` Prozess, welche Commits er hat, die der Server nicht hat. Für jede Referenz, die durch diesen Push aktualisiert wird, teilt der `send-pack` Prozess dem `receive-pack` Prozess diese Informationen mit. Wenn Sie beispielsweise den `master` Branch aktualisieren und einen `experiment` Branch hinzufügen, sieht die Antwort von `send-pack` möglicherweise so aus:

```
0076ca82a6dff817ec66f44342007202690a93763949 15027957951b64cf874c3557a0f3547bd83b3ff6  
\  
    refs/heads/master report-status  
006c00 cdfdb42577e2506715f8cfeacdbabc092bf63e8d
```

```
\  
refs/heads/experiment  
0000
```

Git sendet eine Zeile für jede Referenz, die Sie aktualisieren, mit der Länge der Zeile, dem alten SHA-1, dem neuen SHA-1 und der Referenz, die aktualisiert wird. Die erste Zeile enthält auch die Funktionen des Clients. Der SHA-1-Wert aller Nullen bedeutet, dass zuvor nichts vorhanden war, da Sie die experiment-Referenz hinzufügen. Wenn Sie eine Referenz löschen, sehen Sie das Gegenteil: Alle Nullen auf der rechten Seite.

Als nächstes sendet der Client ein packfile mit allen Objekten, die der Server noch nicht hat. Schließlich antwortet der Server mit einer Erfolgs- oder Fehlermeldung:

```
000eunpack ok
```

## HTTP(S)

Der Prozess über HTTP ist fast derselbe, nur das Handshaking ist ein wenig anders. Die Verbindung wird mit folgender Anfrage initiiert:

```
=> GET http://server/simplegit-progit.git/info/refs?service=git-receive-pack  
001f# service=git-receive-pack  
00ab6c5f0e45abd7832bf23074a333f739977c9e8188 refs/heads/master\report-status \  
    delete-refs side-band-64k quiet ofs-delta \  
    agent=git/2:2.1.1~vmpg-bitmaps-bugaloo-608-g116744e  
0000
```

Das ist das Ende der ersten Client-Server-Unterhaltung. Der Client stellt dann eine weitere Anfrage, diesmal einen **POST**, mit den Daten, die **send-pack** liefert.

```
=> POST http://server/simplegit-progit.git/git-receive-pack
```

Die **POST** Abfrage enthält die **send-pack** Ausgabe und das packfile als Nutzdaten. Der Server zeigt dann mit seiner HTTP-Antwort Erfolg oder Fehler an.

Denken Sie daran, dass das HTTP-Protokoll diese Daten möglicherweise zusätzlich in einen „chunked transfer“ Code verpackt.

## Daten herunterladen

Wenn Sie Daten herunterladen, sind die Prozesse **fetch-pack** und **upload-pack** beteiligt. Der Client initiiert einen **fetch-Pack** Prozess, der eine Verbindung zu einem **upload-pack** Prozess auf der Remote-Seite herstellt, um auszuhandeln, welche Daten nach übertragen werden sollen.

## SSH

Wenn Sie den Abruf über SSH ausführen, führt **fetch-pack** in etwa Folgendes aus:

```
$ ssh -x git@server "git-upload-pack 'simplegit-progit.git'"
```

Nachdem **fetch-pack** eine Verbindung hergestellt hat, sendet **upload-pack** in etwas Folgendes zurück:

```
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
```

Dies ist sehr ähnlich zu dem, was **receive-pack** antwortet, aber die Einsatzmöglichkeiten sind unterschiedlich. Außerdem wird zurückgesendet, worauf HEAD verweist (**symref=HEAD:refs/heads/master**), sodass der Client weiß, was er überprüfen muss, wenn es sich um einen Klon handelt.

Zu diesem Punkt prüft der **fetch-pack** Prozess, über welche Objekte er verfügt, und antwortet mit den Objekten, die er benötigt, indem er „want“ und dann den gewünschten SHA-1 sendet. Es sendet alle Objekte, die es bereits hat, mit „have“ und dann den SHA-1. Am Ende dieser Liste wird „done“ geschrieben, um den `upload-pack“-Prozess einzuleiten, der das packfile mit den benötigten Daten sendet:

```
003cwant ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0009done
0000
```

## HTTP(S)

Der Handshake für einen Abrufvorgang benötigt zwei HTTP Aufrufe. Das erste ist ein **GET** zum selben Endpunkt, der im dummen Protokoll verwendet wird:

```
=> GET $GIT_URL/info/refs?service=git-upload-pack
001e# service=git-upload-pack
00e7ca82a6dff817ec66f44342007202690a93763949 HEAD multi_ack thin-pack \
side-band side-band-64k ofs-delta shallow no-progress include-tag \
multi_ack_detailed no-done symref=HEAD:refs/heads/master \
agent=git/2:2.1.1+github-607-gfba4028
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
0000
```

Dies ist dem Aufrufen von `git-upload-pack` über eine SSH-Verbindung sehr ähnlich. Der zweite Austausch wird als separater Aufruf ausgeführt:

```
=> POST $GIT_URL/git-upload-pack HTTP/1.0
0032want 0a53e9ddeadad63ad106860237bbf53411d11a7
0032have 441b40d833fd9a93eb2908e52742248faf0ee993
0000
```

Auch dies ist das gleiche Format wie oben. Die Antwort auf diese Anfrage zeigt Erfolg oder Fehler an und enthält das packfile.

## Zusammenfassung der Protokolle

Dieser Abschnitt enthält eine sehr grundlegende Übersicht über die Transfer Protokolle. Das Protokoll enthält viele andere Funktionen, wie z.B. `multi_ack` oder `side-band`, die jedoch nicht in diesem Buch behandelt werden. Wir haben versucht, Ihnen ein Gefühl für das allgemeine Hin und Her zwischen Client und Server zu vermitteln. Wenn Sie mehr Informationen diesbezüglich benötigen, sollten Sie sich den Git-Quellcode ansehen.

# Wartung und Datenwiederherstellung

Möglicherweise müssen sie hin und wieder Bereinigungen durchführen. Bspw. müssen sie ein Repository komprimieren, ein importiertes Repository bereinigen oder verlorene Arbeit wiederherstellen. In diesem Abschnitt werden einige dieser Szenarien behandelt.

## Wartung

Gelegentlich führt Git automatisch einen Befehl namens „auto gc“ aus. Meistens macht dieser Befehl gar nichts. Wenn sich jedoch zu viele lose Objekte (Objekte, die nicht in einem Packfile enthalten sind) oder zu viele Packfiles gibt, startet Git einen vollständigen `git gc` Befehl. Das „gc“ steht für Garbage Collect und der Befehl führt eine Reihe von Aktionen aus: Er sammelt alle losen Objekte und legt sie in Packfiles ab. Er fasst Packfiles zu einem großen Packfile zusammen. Außerdem entfernt er Objekte, die nicht von einem Commit erreichbar und ein paar Monate alt sind.

Sie können `auto gc` folgendermaßen manuell ausführen:

```
$ git gc --auto
```

Auch dies tut im Allgemeinen nichts. Sie müssen ungefähr 7.000 lose Objekte oder mehr als 50 Packfiles haben, damit Git einen echten gc-Befehl ausführt. Sie können diese Grenzwerte mit den Konfigurationseinstellungen `gc.auto` und `gc.autopacklimit` ändern.

Die andere Aktion, die `gc` durchführt, ist Ihre Referenzen in eine einzige Datei zu packen. Angenommen, Ihr Repository enthält die folgenden Branches und Tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Wenn Sie `git gc` ausführen, befinden sich diese Dateien nicht mehr im Verzeichnis `refs`. Git verschiebt sie aus Gründen der Effizienz in eine Datei mit dem Namen `.git/packed-refs`, die so aussieht:

```
$ 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
```

Wenn Sie eine Referenz aktualisieren, bearbeitet Git diese Datei nicht, sondern schreibt eine neue Datei in `refs/heads`. Um das passende SHA-1 für eine angegebene Referenz zu erhalten, prüft Git diese Referenz im `refs` Verzeichnis und prüft dann die `packed-refs` Datei als Fallback. Wenn Sie jedoch keine Referenz im `refs` Verzeichnis finden können, befindet sich diese wahrscheinlich in Ihrer `packed-refs` Datei.

Beachten Sie die letzte Zeile der Datei, die mit einem `^` beginnt. Dies bedeutet, dass das darüberliegende Tag ein annotiertes Tag ist und dass diese Zeile das Commit ist, auf das das annotierte Tag verweist.

## Datenwiederherstellung

Es wird der Punkt auf Ihrer Git-Reise kommen, an dem Sie versehentlich einen oder mehrere Commits verlieren. Im Allgemeinen geschieht dies, weil Sie einen Branch mittels `force` Option löschen und es sich herausstellt, dass Sie den Branch doch noch benötigten. Oder aber es passiert, dass Sie einen Branch hart zurückgesetzt haben und noch benötigte Commits verworfen haben. Was können Sie tun, um Ihre Commits zurückzuerhalten?

In diesem Beispiel wird der `master` Branch in Ihrem Test-Repository auf einen älteren Commit zurückgesetzt und die verlorenen Commits wiederhergestellt. Lassen Sie uns zunächst überprüfen, wo sich Ihr Repository zu diesem Zeitpunkt befindet:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
^1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Verschieben Sie nun den 'master'-Branch zurück zum mittleren 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
```

Sie haben die beiden obersten Commits verloren. Sie haben jetzt keinen Branch, von dem aus diese Commits erreichbar sind. Sie müssen das letzte Commit-SHA-1 finden und anschließend einen Branch hinzufügen, der darauf verweist. Der Trick ist, das letzte Commit SHA-1 zu finden. Leider ist es nicht so, als hätten Sie es auswendig gelernt, oder?

Oft ist der schnellste Weg die Nutzung eines Tools namens `git reflog`. Während sie arbeiten, zeichnet Git lautlos im Hintergrund auf, was ihr HEAD ist und worauf es zeigt. Jedes Mal, wenn Sie Branches commiten oder ändern, wird das Reflog aktualisiert. Das reflog wird auch durch den Befehl `git update-ref` aktualisiert. Dies ist ein weiterer Grund, warum Sie diesen Befehl verwenden sollten, anstatt nur den SHA-1-Wert in Ihre ref-Dateien zu schreiben, wie in [Git Referenzen](#) beschrieben. Sie können jederzeit sehen, wo Sie waren, indem Sie `git reflog` ausführen:

```
$ git reflog  
1a410ef HEAD@{0}: reset: moving to 1a410ef  
ab1afef HEAD@{1}: commit: Modify repo.rb a bit  
484a592 HEAD@{2}: commit: Create repo.rb
```

Hier sehen wir die beiden Commits, die wir ausgecheckt haben, jedoch gibt es hier nicht viele Informationen. Um die gleichen Informationen auf eine viel nützlichere Weise anzuzeigen, können wir `git log -g` ausführen, wodurch Sie eine normale Logausgabe für Ihr Reflog erhalten.

```
$ 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  
  
Modify repo.rb a bit
```

Es sieht so aus, als ob Sie das unterste Commit verloren haben. Sie können es also wiederherstellen, indem Sie bei diesem Commit einen neuen Branch erstellen. Beispielsweise können Sie einen Branch mit dem Namen **recover-branch** bei diesem Commit (ab1afef) starten:

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b Modify repo.rb a bit
484a59275031909e19aadb7c92262719cfcdf19a Create repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 Third commit
cac0cab538b970a37ea1e769cbbde608743bc96d Second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d First commit
```

Sehr gut - jetzt haben Sie einen Branch namens **recover-branch**, in dem sich früher Ihr **master** Branch befand, wodurch die ersten beiden Commits wieder erreichbar sind. Angenommen, Ihr Verlust war aus irgendeinem Grund nicht im Reflog - Sie können dies simulieren, indem Sie **recover-branch** entfernen und das Reflog löschen. Jetzt sind die ersten beiden Commits nicht mehr erreichbar:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Da sich die Reflog-Daten im Verzeichnis **.git/logs/** befinden, haben Sie praktisch kein Reflog. Wie können Sie dieses Commit zu diesem Zeitpunkt wiederherstellen? Eine Möglichkeit ist die Verwendung des Hilfsprogramms **git fsck**, mit dem Ihre Datenbank auf Integrität überprüft wird. Wenn Sie es mit der Option **--full** ausführen, werden alle Objekte angezeigt, auf die kein anderes Objekt zeigt:

```
$ 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 diesem Fall können Sie Ihr fehlendes Commit nach dem String „Dangling Commit“ sehen. Sie können es auf die gleiche Weise wiederherstellen, indem Sie einen Branch hinzufügen, der auf diesen SHA-1 verweist.

## Objekte löschen

Es gibt viele großartige Dinge an Git. Eine Funktion, die jedoch Probleme verursachen kann, ist die Tatsache, dass ein Git-Klon den gesamten Verlauf des Projekts herunterlädt, einschließlich jeder Version jeder Datei. Dies ist in Ordnung, wenn das Ganze Quellcode ist. Git ist stark darin, diese Daten effizient zu komprimieren. Wenn jedoch zu einem beliebigen Zeitpunkt im Verlauf Ihres Projekts eine einzelne große Datei hinzugefügt wird, muss jeder Klon für alle Zeiten diese große

Datei herunterladen, auch wenn sie beim nächsten Commit aus dem Projekt entfernt wurde. Weil sie von der Historie aus erreichbar ist, wird sie immer da sein.

Dies kann ein großes Problem sein, wenn Sie Subversion- oder Perforce-Repositorys nach Git konvertieren. Da Sie in diesen Systemen nicht den gesamten Verlauf herunterladen, hat dieses Modell des Hinzufügens nur wenige Konsequenzen. Wenn Sie einen Import von einem anderen System durchgeführt haben oder auf andere Weise feststellen, dass Ihr Repository viel größer ist, als es sein sollte, können Sie große Objekte folgendermaßen finden und entfernen.

**Seien Sie gewarnt: Diese Technik wirkt sich zerstörerisch auf Ihren Commit-Verlauf aus.** Es schreibt jedes Commitobjekt neu, seit dem frühesten Baum, den Sie ändern müssen, um einen Verweis auf eine große Datei zu entfernen. Wenn Sie dies unmittelbar nach einem Import tun, bevor jemand damit begonnen hat, sich auf das Commit zu stützen, ist alles in Ordnung. Andernfalls müssen Sie alle Mitwirkenden benachrichtigen, dass sie ihre Arbeit auf Ihre neuen Commits rebasen müssen.

Zur Veranschaulichung fügen Sie Ihrem Test-Repository eine große Datei hinzu, entfernen Sie sie beim nächsten Commit. Anschließend suchen Sie sie und entfernen sie dauerhaft aus dem Repository. Fügen Sie Ihrer Historie zunächst ein großes Objekt hinzu:

```
$ curl -L https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'Add git tarball'
[master 7b30847] Add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Hoppla - Sie wollten Ihrem Projekt keinen riesigen Tarball hinzufügen. Besser wäre, es loszuwerden:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'Oops - remove large tarball'
[master dadf725] Oops - remove large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Nun wenden sie **gc** auf Ihre Datenbank an und sehen sie, wie viel Speicherplatz Sie verwenden:

```
$ 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)
```

Sie können den Befehl **count-objects** ausführen, um schnell zu sehen, wie viel Speicherplatz Sie

verwenden:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

Der `size-pack` Eintrag gibt die Größe Ihrer Packdateien in Kilobyte an. Somit verwenden Sie fast 5 MB an Speicherplatz. Vor dem letzten Commit haben Sie einen Wert von ungefähr 2 KB belegt. Wenn Sie die Datei aus dem vorherigen Commit entfernen, würde sie offensichtlich nicht aus Ihrem Verlauf entfernt. Jedes Mal, wenn jemand dieses Repository klonen, muss er 5 MB klonen, um dieses winzige Projekt zu erhalten, da Sie versehentlich eine große Datei hinzugefügt haben. Lass sie es uns loswerden.

Als erstes müssen wir es finden. In diesem Fall wissen Sie bereits, um welche Datei es sich handelt. Angenommen, Sie wissen es nicht. Wie würden Sie feststellen, welche Datei oder welche Dateien so viel Speicherplatz beanspruchen? Wenn Sie `git gc` ausführen, befinden sich alle Objekte in einer Packdatei. Sie können die großen Objekte identifizieren, indem Sie einen anderen Installationsbefehl namens `git verify-pack` ausführen und die Ausgabe nach dem dritten Feld sortieren, das die Dateigröße ist. Sie können es auch über den Befehl `tail` pipen, da Sie nur an den letzten, großen Dateien interessiert sind:

```
$ 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
```

Das große Objekt befindet sich unten: 5 MB. Um herauszufinden, um welche Datei es sich handelt, verwenden Sie den Befehl `rev-list`, den Sie kurz in [Ein bestimmtes Commit-Message-Format erzwingen](#) verwendet haben. Wenn Sie `--objects` an `rev-list` übergeben, werden alle festgeschriebenen SHA-1s und auch die BLOB-SHA-1s mit den ihnen zugeordneten Dateipfaden aufgelistet. Sie können dies verwenden, um den Namen Ihres Blobs zu finden:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Jetzt müssen Sie diese Datei von allen Bäumen in Ihrer Historie entfernen. Sie können leicht sehen, welche Commits diese Datei geändert haben:

```
$ git log --oneline --branches -- git.tgz
dadf725 Oops - remove large tarball
7b30847 Add git tarball
```

Sie müssen alle Commits hinter [7b30847](#) neu schreiben, um diese Datei vollständig aus Ihrem Git-Verlauf zu entfernen. Verwenden Sie dazu [filter-branch](#), den Sie in [Den Verlauf umschreiben](#) verwendet haben:

```
$ 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
```

Die Option [--index-filter](#) ähnelt der Option [--tree-filter](#), die in [Den Verlauf umschreiben](#) verwendet wurde. Jedoch ändern sie jedes Mal Ihren Staging-Bereich oder Index anstatt daß sie einen Befehl zum Modifizieren von ausgecheckten Dateien auf ihrer Platte übergeben.

Anstatt eine bestimmte Datei mit etwas wie [rm file](#) zu entfernen, müssen Sie sie mit [git rm --cached](#) entfernen - Sie müssen sie aus dem Index entfernen, nicht von der Festplatte. Der Grund dafür ist die Geschwindigkeit - da Git nicht jede Revision auf der Festplatte auschecken muss, bevor der Filter ausgeführt wird, kann der Prozess sehr viel schneller sein. Sie können die gleiche Aufgabe mit [--tree-filter](#) ausführen, wenn Sie möchten. Die Option [--ignore-unmatch](#) für [git rm](#) weist an, dass keine Fehler auftreten, wenn das zu entfernende Muster nicht vorhanden ist. Schließlich fordern Sie [filter-branch](#) auf, Ihre Historie erst ab dem Commit [7b30847](#) neu zu schreiben, da Sie wissen, wo dieses Problem begann. Andernfalls wird es von vorne beginnen und unnötig länger dauern.

Ihr Verlauf enthält nun keinen Verweis mehr auf diese Datei. Ihr Reflog und eine neue Gruppe von Refs, die Git hinzugefügt hat, als Sie den [filter-branch](#) unter [.git/refs/original](#) ausgeführt haben, enthält jedoch weiterhin Verweise, sodass Sie sie entfernen und die Datenbank neu packen müssen. Sie müssen alles loswerden, das einen Zeiger auf diese alten Commits enthält, bevor Sie neu packen:

```
$ 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)
```

Mal sehen, wie viel Platz Sie gespart haben.

```
$ git count-objects -v
```

```
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

Die Größe des gepackten Repositorys beträgt nur noch 8 KB, was viel besser als 5 MB ist. Sie können anhand der Größe erkennen, dass sich das große Objekt noch in Ihren losen Objekten befindet, sodass es nicht verschwunden ist. Es wird jedoch nicht auf einen Push oder nachfolgenden Klon übertragen, was wichtig ist. Wenn Sie es wirklich wollten, können Sie das Objekt vollständig entfernen, indem Sie `git prune` mit der Option `--expire` ausführen:

```
$ 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
```

` === Umgebungsvariablen

Git läuft immer in einer `bash` Shell und verwendet eine Reihe von Shell-Umgebungsvariablen, über die man steuern kann, wie es sich verhält. Gelegentlich ist es hilfreich zu wissen, welche diese sind und wie Sie Git dazu bringen können, sich so zu verhalten, wie Sie es möchten. Dies ist keine vollständige Liste aller Umgebungsvariablen auf die Git achtet, aber wir werden die nützlichsten behandeln.

## Globals Verhalten

Einige der generellen Eigenschaften von Git als Computerprogramm hängen von Umgebungsvariablen ab.

`GIT_EXEC_PATH` bestimmt, wo Git nach seinen Unterprogrammen sucht (wie `git-commit`, `git-diff` und andere). Sie können die aktuelle Einstellung überprüfen, indem Sie `git --exec-path` ausführen.

`HOME` wird normalerweise nicht als anpassbar angesehen (zu viele andere Dinge hängen davon ab), aber hier sucht Git nach der globalen Konfigurationsdatei. Wenn Sie eine wirklich portable Git-Installation mit globaler Konfiguration wünschen, können Sie `HOME` im portablen Git Shell-Profil überschreiben.

`PREFIX` ist ähnlich, jedoch für die systemweite Konfiguration. Git sucht nach dieser Datei unter

`$PREFIX/etc/gitconfig`.

**GIT\_CONFIG\_NOSYSTEM**, falls gesetzt, deaktiviert die Verwendung der systemweiten Konfigurationsdatei. Dies ist nützlich, wenn Ihre Systemkonfiguration Ihre Befehle beeinträchtigt, Sie jedoch keinen Zugriff haben, um diese zu ändern oder zu entfernen.

**GIT\_PAGER** steuert das Programm, mit dem mehrseitige Ausgaben in der Befehlszeile angezeigt werden. Ist dies nicht gesetzt, wird **PAGER** als Fallback verwendet.

**GIT\_EDITOR** ist der Editor, den Git startet, wenn der Benutzer Text bearbeiten muss (z.B. eine Commit-Nachricht). Wenn nicht gesetzt, wird **EDITOR** verwendet.

## Speicherort des Repositorys

Git verwendet mehrere Umgebungsvariablen, um die Verbindung zum aktuellen Repository herzustellen.

**GIT\_DIR** ist der Speicherort des Ordners `.git`. Wenn dies nicht angegeben ist, geht Git nach oben durch den Verzeichnisbaum, bis es zu `~` oder `/` gelangt, und sucht bei jedem Schritt nach einem `.git` Verzeichnis.

**GIT\_CEILING\_DIRECTORIES** steuert das Verhalten bei der Suche nach einem `.git` Verzeichnis. Wenn Sie auf Verzeichnisse zugreifen, die nur langsam geladen werden können (z.B. auf einem Bandlaufwerk oder über eine langsame Netzwerkverbindung), möchten Sie möglicherweise, dass Git den Versuch vorzeitig abbricht, insbesondere wenn Git in der Kommandozeile aufgerufen wird.

**GIT\_WORK\_TREE** ist der Speicherort des Stammverzeichnisses eines Arbeitsverzeichnisses für ein non-bare Repository. Wenn `--git-dir` oder **GIT\_DIR** angegeben ist, jedoch nicht `--work-tree`, **GIT\_WORK\_TREE** oder `core.worktree`, wird das aktuelle Arbeitsverzeichnis als oberste Ebene Ihres Arbeitsbaums betrachtet.

**GIT\_INDEX\_FILE** ist der Pfad zur Indexdatei (nur für non-bare Repositorys).

**GIT\_OBJECT\_DIRECTORY** kann verwendet werden, um den Speicherort des Verzeichnisses anzugeben, das sich normalerweise in `.git/objects` befindet.

**GIT\_ALTERNATE\_OBJECT\_DIRECTORIES** ist eine durch Doppelpunkte getrennte Liste (also im Format `/dir/one:/dir/two:...`), die Git mitteilt, wo nach Objekten gesucht werden soll, wenn sie sich nicht in `GIT_OBJECT_DIRECTORY` befinden. Wenn Sie viele Projekte mit großen Dateien haben, die genau den gleichen Inhalt haben, können Sie damit vermeiden, dass zu viele Kopien davon gespeichert werden.

## Pfadspezifikation (engl. Pathspec)

„Pathspec“ bezieht sich darauf, wie Sie Pfade in Git angeben, einschließlich der Verwendung von Platzhaltern. Diese werden in der Datei `.gitignore` aber auch in der Befehlszeile (`git add *.c`) verwendet.

**GIT\_GLOB\_PATHSPECS** und **GIT\_NOGLOB\_PATHSPECS** steuern das Standardverhalten von Platzhaltern in Pfadangaben. Wenn **GIT\_GLOB\_PATHSPECS** auf 1 gesetzt ist, werden Platzhalterzeichen als Platzhalter

verwendet (dies ist die Standardeinstellung). Wenn `GIT_NOGLOB_PATHSPECS` auf 1 gesetzt ist, stimmen Platzhalterzeichen nur mit sich selbst überein. Dies bedeutet, dass `.c nur mit einer Datei namens .c` übereinstimmt und nicht mit einer Datei, deren Name mit `.c` endet. Sie können dies in Einzelfällen überschreiben, indem Sie die Pfadangabe mit `:(glob)` oder `:(literal)` beginnen, wie in `:(glob)*.c`.

`GIT_LITERAL_PATHSPECS` deaktiviert beide oben genannten Verhaltensweisen. Es können keine Platzhalterzeichen verwendet werden, und die Präfixe zum Überschreiben sind ebenfalls deaktiviert.

`GIT_ICASE_PATHSPECS` setzt alle Pfadangaben so, dass zwischen Groß- und Kleinschreibung nicht unterschieden wird.

## Committen

Die endgültige Erstellung eines Git-Commit-Objekts erfolgt normalerweise über `git-commit-tree`, das diese Umgebungsvariablen als primäre Informationsquelle verwendet und nur dann auf Konfigurationswerte zurückgreift, wenn diese nicht vorhanden sind.

`GIT_AUTHOR_NAME` ist der für Menschen lesbare Name im Feld „author“.

`GIT_AUTHOR_EMAIL` ist die E-Mail-Adresse für das Feld „author“.

`GIT_AUTHOR_DATE` ist der Zeitstempel für das Feld „author“.

`GIT_COMMITTER_NAME` legt den für Menschen lesbaren Namen für das Feld „Committer“ fest.

`GIT_COMMITTER_EMAIL` ist die E-Mail-Adresse für das Feld „Committer“.

`GIT_COMMITTER_DATE` wird für den Zeitstempel im Feld „Committer“ verwendet.

`EMAIL` ist die Ersatz-E-Mail-Adresse für den Fall, dass der Konfigurationswert `user.email` nicht festgelegt ist. Wenn *this* nicht festgelegt ist, greift Git auf die Systembenutzer und Hostnamen zurück.

## Netzwerk

Git verwendet die Bibliothek `curl`, um Netzwerkoperationen über HTTP durchzuführen. `GIT_CURL_VERBOSE` weist Git an, alle von dieser Bibliothek generierten Nachrichten auszugeben. Dies ähnelt dem Ausführen von `curl -v` in der Befehlszeile.

`GIT_SSL_NO_VERIFY` weist Git an, SSL-Zertifikate nicht zu verifizieren. Dies kann manchmal erforderlich sein, wenn Sie ein selbstsigniertes Zertifikat verwenden, um Git-Repositorys über HTTPS bereitzustellen, oder wenn Sie gerade einen Git-Server einrichten, aber noch kein vollständiges Zertifikat installiert haben.

Wenn die Datenrate einer HTTP-Operation unter `GIT_HTTP_LOW_SPEED_LIMIT` Bytes pro Sekunde und länger als `GIT_HTTP_LOW_SPEED_TIME` Sekunden anhält, bricht Git diese Operation ab. Diese Werte überschreiben die Konfigurationswerte `http.lowSpeedLimit` und `http.lowSpeedTime`.

`GIT_HTTP_USER_AGENT` legt die User-Agent-Zeichenfolge fest, die von Git bei der Kommunikation über

HTTP verwendet wird. Der Standardwert ist ein Wert wie `git/2.0.0`.

## Vergleichen und Zusammenführen

`GIT_DIFF_OPTS` ist eigentlich ein unzutreffender Name. Die einzigen gültigen Werte sind `-u<n>` oder `--unified=<n>`, wodurch die Anzahl der in einem `git diff` Befehl angezeigten Kontextzeilen konfiguriert wird.

`GIT_EXTERNAL_DIFF` überschreibt den Konfigurationswert `diff.external`. Wenn diese Variable gesetzt ist, ruft Git dieses Programm auf, wenn `git diff` aufgerufen wird.

`GIT_DIFF_PATH_COUNTER` und `GIT_DIFF_PATH_TOTAL` sind innerhalb des durch `GIT_EXTERNAL_DIFF` oder `diff.external` angegebenen Programms nützlich. Erstes gibt an, welche Datei in einer Reihe von Dateien verglichen wird (beginnend mit 1), und Letzteres gibt die Gesamtzahl der Dateien im Stapel an.

`GIT_MERGE_VERBOSITY` steuert die Ausgabe für die rekursive Merge-Strategie. Folgende Werte sind zulässig:

- 0 gibt nichts aus, außer einer einzelnen Fehlermeldung (möglicherweise).
- 1 zeigt nur Konflikte.
- 2 zeigt auch Dateiänderungen an.
- 3 zeigt an, wann Dateien übersprungen werden, weil sie sich nicht geändert haben.
- 4 zeigt alle Pfade, während sie verarbeitet werden.
- 5 und höher zeigen detaillierte Debugging-Informationen.

Der Standardwert ist 2.

## Debugging

Möchten Sie wirklich wissen, was in Git abgeht? In Git ist eine umfangreiche Sammlung von Traces eingebettet, alles was Sie tun müssen, ist sie einzuschalten. Die möglichen Werte dieser Variablen lauten wie folgt:

- „true“, „1“ oder „2“ - die Trace-Kategorie wird nach stderr geschrieben.
- Ein absoluter Pfad, der mit / beginnt - die Trace-Ausgabe wird in diese Datei geschrieben.

`GIT_TRACE` steuert allgemeine Traces, die keiner bestimmten Kategorie zugeordnet werden können. Dies umfasst die Erweiterung von Aliasen und die Delegierung an andere Unterprogramme.

```
$ 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** steuert das Tracing der Packfile-Zugriffe. Das erste Feld ist die Packdatei, auf die zugegriffen wird, das zweite Feld ist der Offset in dieser Datei:

```
$ 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** aktiviert die Paketverfolgung für Netzwerkoperationen.

```
$ 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** steuert die Protokollierung von Performance-Daten. Die Ausgabe zeigt, wie lange jeder einzelne Aufruf von **git** dauert.

```
$ 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** zeigt Informationen darüber an, was Git über das Repository und die Umgebung, mit denen es interagiert, herausfindet.

```

$ 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

```

## Sonstiges

**GIT\_SSH**, falls angegeben, ist ein Programm, das anstelle von `ssh` aufgerufen wird, um eine Verbindung zu einem SSH-Host herzustellen. Es wird folgendermaßen aufgerufen: `$GIT_SSH [username@]host [-p <port>] <befehl>`. Beachten Sie, dass dies nicht der einfachste Weg ist, um zu konfigurieren, wie `ssh` aufgerufen wird. Es werden keine zusätzlichen Befehlszeilenparameter unterstützt, daher müssen Sie ein Wrapper-Skript schreiben und `GIT_SSH` so einstellen, dass es darauf verweist. Es ist wahrscheinlich einfacher, dafür einfach die Datei `~/.ssh/config` zu verwenden.

**GIT\_ASKPASS** dient zur Überschreibung des Konfigurationswertes `core.askpass`. Dies ist das Programm, das immer dann aufgerufen wird, wenn Git den Benutzer nach Anmeldeinformationen fragen muss, wobei eine Eingabeaufforderung als Befehlszeilenargument erwartet wird und die eine Antwort auf `stdout` zurückgeben soll. Weitere Informationen zu diesem Subsystem finden Sie unter [Anmeldeinformationen speichern](#).

**GIT\_NAMESPACE** steuert den Zugriff auf namenspaced refs und entspricht dem Flag `--namespace`. Dies ist vor allem auf der Serverseite nützlich, wo Sie möglicherweise mehrere Forks eines einzelnen Repositorys in einem Repository speichern möchten, wobei nur die Refs getrennt bleiben.

**GIT\_FLUSH** kann verwendet werden, um Git zu zwingen, nicht gepuffertes I/O zu verwenden, wenn inkrementell in stdout geschrieben wird. Ein Wert von 1 bewirkt, dass Gits Puffer öfter geleert wird. Ein Wert von 0 bewirkt, dass alle Ausgaben gepuffert werden. Der Standardwert (falls diese Variable nicht festgelegt ist) ist die Auswahl eines geeigneten Pufferschemas abhängig von Aktivität und Ausgabemodus.

Mit **GIT\_REFLOG\_ACTION** können Sie den beschreibenden Text angeben, der in das Reflog geschrieben wird. Hier ein Beispiel:

```
$ 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
```

## Zusammenfassung

Zu diesem Zeitpunkt sollten Sie ein ziemlich gutes Verständnis dafür haben, was Git im Hintergrund macht und bis zu einem gewissen Grad auch, wie es implementiert ist. Dieses Kapitel hat eine Reihe von Basisbefehlen behandelt – Befehle, die niedriger und einfacher sind als die Porzellanbefehle, die Sie im Rest des Buches kennengelernt haben. Wenn Sie verstehen, wie Git auf einer niedrigeren Ebene funktioniert, sollten Sie leichter verstehen, warum es das tut, was es tut. Sie könnten nun Ihre eigenen Tools und Hilfsskripten schreiben, damit Ihr spezifischer Workflow für Sie funktioniert.

Git als inhaltsadressierbares Dateisystem ist ein sehr leistungsfähiges Tool, das Sie problemlos als mehr als nur ein einfaches VCS verwenden können. Wir hoffen, dass Sie Ihr neu gewonnenes Wissen über Git-Interna nutzen können, um Ihre eigene coole Anwendung dieser Technologie zu implementieren und sich auf fortgeschrittenere Weise mit Git vertraut zu machen.

# Appendix A: Git in anderen Umgebungen

Wenn Sie das ganze Buch durchgelesen haben, haben Sie viel darüber gelernt, wie man Git auf der Kommandozeile benutzt. Sie können mit lokalen Dateien arbeiten, Ihr Repository über ein Netzwerk mit anderen verbinden und effektiv mit anderen zusammenarbeiten. Aber die Geschichte endet nicht dort; Git wird normalerweise als Teil eines größeren Ökosystems verwendet, und die Kommandozeile ist nicht immer die beste Möglichkeit, damit zu arbeiten. Jetzt werden wir uns einige der anderen Arten von Umgebungen ansehen, in denen Git nützlich sein kann, und wie andere Anwendungen (einschließlich Ihrer) neben Git funktionieren.

## Grafische Schnittstellen

Die native Umgebung von Git ist das Terminal. Dort werden zuerst neue Funktionen implementiert, und nur über die Befehlszeile steht Ihnen die volle Leistung von Git zur Verfügung. Die Befehlszeile ist jedoch nicht für alle Aufgaben die beste Wahl. Manchmal benötigen Sie eine visuelle Darstellung, und einige Benutzer sind mit einer grafischen -Oberfläche viel besser vertraut.

Es ist wichtig zu beachten, dass unterschiedliche Schnittstellen auf unterschiedliche Workflows zugeschnitten sind. Einige Clients stellen nur eine sorgfältig zusammengestellte Teilmenge der Git-Funktionalität zur Verfügung, um eine bestimmte Arbeitsweise zu unterstützen, die der Autor für effektiv hält. Mit diesem Hintergrund kann keines dieser Tools als „besser“ bezeichnet werden als die anderen, sie sind einfach besser für den beabsichtigten Zweck geeignet. Beachten Sie auch, dass diese grafischen Clients nichts tun können, was der Befehlszeilenclient nicht kann. In der Befehlszeile haben Sie immer noch die größte Leistung und Kontrolle, wenn Sie mit Ihren Repositorys arbeiten.

### gitk und git-gui

Wenn Sie Git installieren, erhalten Sie auch die visuellen Tools `gitk` und `git-gui`.

`gitk` ist ein grafischer Verlaufsbetrachter. Stellen Sie sich das wie eine leistungsstarke GUI-Shell über `git log` und `git grep` vor. Dies ist das Tool, welches sie nutzen sollten, wenn sie versuchen, etwas zu finden, das in der Vergangenheit passiert ist oder wenn sie den Verlauf Ihres Projekts visualisieren wollen.

Gitk lässt sich am einfachsten über die Befehlszeile aufrufen. Geben Sie einfach eine CD in ein Git-Repository ein und geben Sie Folgendes ein:

```
$ gitk [git log options]
```

Gitk akzeptiert viele Befehlszeilenoptionen, von denen die meisten an die zugrunde liegende Aktion `git Log` übergeben werden. Wahrscheinlich eines der nützlichsten ist das `--all` Flag, das gitk anweist, Commits anzuzeigen, die von jedem Ref erreichbar sind, nicht nur von HEAD. Die Benutzeroberfläche von Gitk sieht folgendermaßen aus:

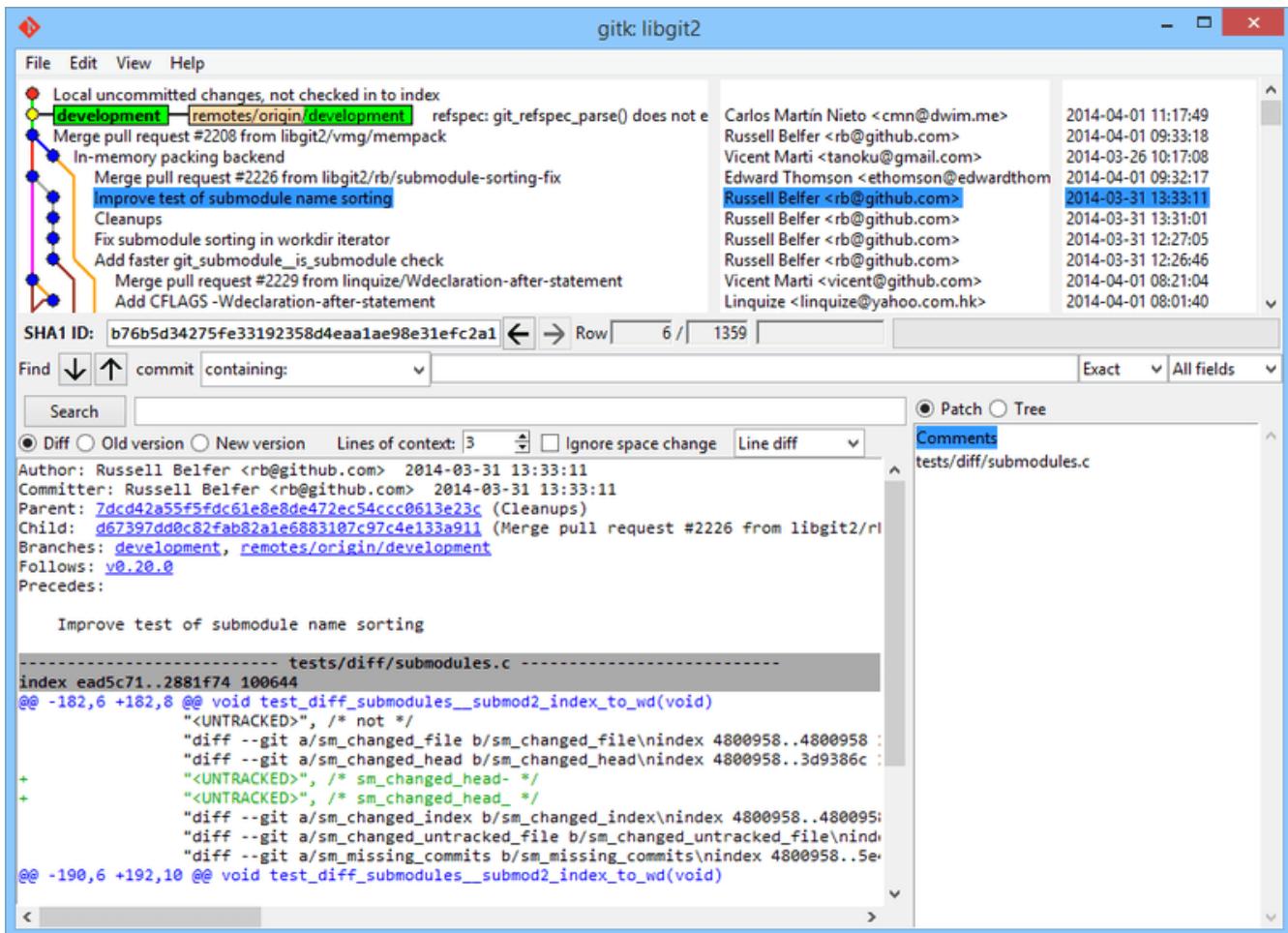


Figure 177. Der `gitk` Verlaufsbetrachter

Im oberen Teil sieht man etwas, das ein bisschen wie die Ausgabe von `git log --graph` aussieht. Jeder Punkt steht für ein Commit, die Linien für übergeordnete Beziehungen und Refs werden als farbige Kästchen angezeigt. Der gelbe Punkt steht für HEAD und der rote Punkt für Änderungen, die noch nicht festgeschrieben wurden. Unten sehen Sie eine Ansicht des ausgewählten Commits. Die Kommentare und Patches auf der linken Seite und eine zusammenfassende Ansicht auf der rechten Seite. Dazwischen befindet sich eine Sammlung von Steuerelementen, die zum Durchsuchen des Verlaufs verwendet werden können.

`git-gui` hingegen ist in erster Linie ein Werkzeug zum Erstellen von Commits. Es kann ebenfalls sehr einfach über die Befehlszeile aufgerufen werden:

```
$ git gui
```

Es sieht in etwa so aus:

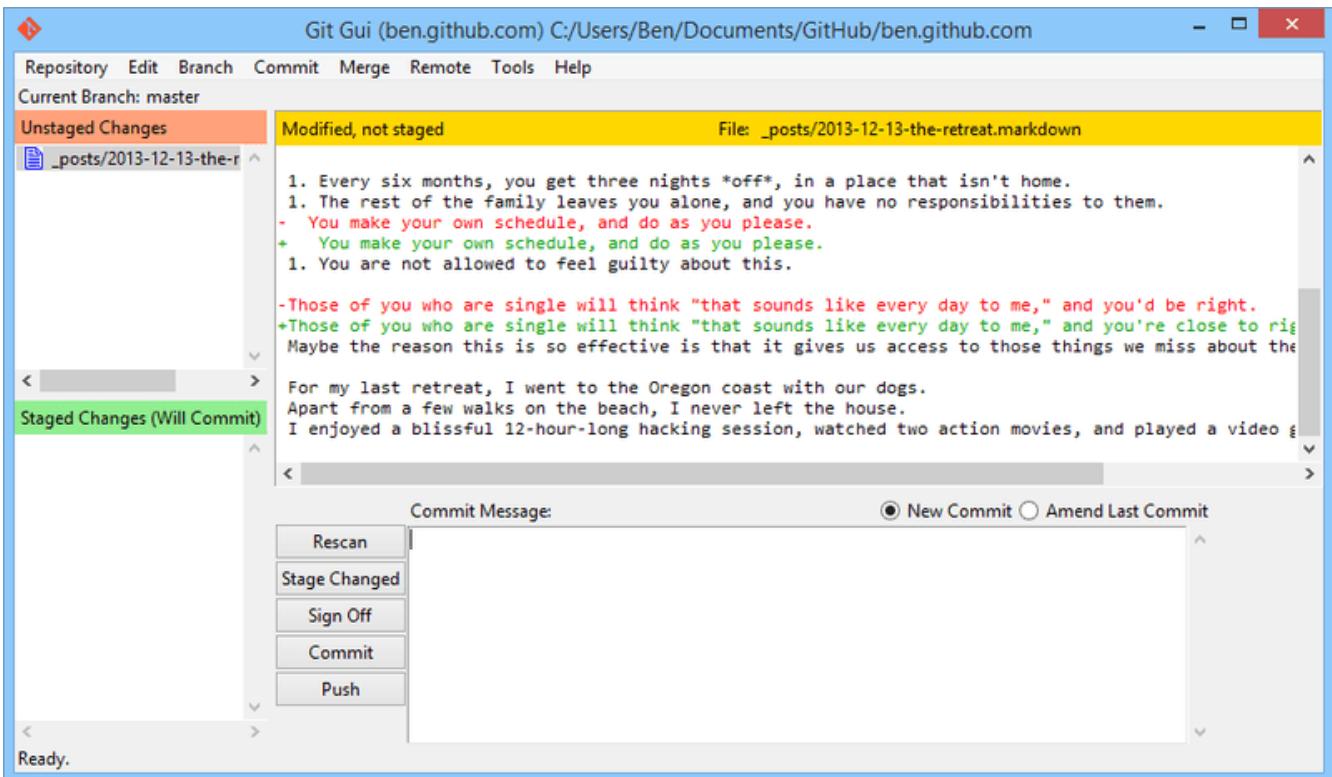


Figure 178. Das `git-gui` Commit-Tool

Links ist der Index. „Unstage“ Änderungen befinden sich oben, „staged“ Änderungen unten. Sie können ganze Dateien zwischen den beiden Status verschieben, indem Sie auf deren Symbole klicken. Weiterhin können Sie eine Datei zum Anzeigen auswählen, indem Sie auf ihren Namen klicken.

Oben rechts befindet sich die Diff-Ansicht, in der die Änderungen für die aktuell ausgewählte Datei angezeigt werden. Sie können einzelne Bereiche (oder einzelne Linien) stagieren, indem Sie mit der rechten Maustaste in diesen Bereich klicken.

Unten rechts befindet sich der Nachrichten- und Aktionsbereich. Geben Sie Ihre Nachricht in das Textfeld ein und klicken Sie auf „Commit“, um etwas Ähnliches wie `git commit` zu tun. Sie können das letzte Commit auch ändern, indem Sie das Optionsfeld „Ändern“ aktivieren, um den Bereich „Staged Changes“ mit dem Inhalt des letzten Commits zu aktualisieren. Anschließend können Sie einfach einige Änderungen aktivieren oder deaktivieren, die Commit-Nachricht ändern und erneut auf „Commit“ klicken, um den alten commit durch einen neuen zu ersetzen.

`gitk` und `git-gui` sind Beispiele für aufgabenorientierte Tools. Jedes von ihnen ist auf einen bestimmten Zweck zugeschnitten (Anzeigen des Verlaufs bzw. Erstellen von Commits) und lässt die für diese Aufgabe nicht erforderlichen Funktionen aus.

## GitHub für macOS und Windows

GitHub hat zwei Workflow-orientierte Git-Clients erstellt: einen für Windows und einen für macOS. Diese Clients sind ein gutes Beispiel für Workflow-orientierte Tools. Anstatt alle Funktionen von Git zu implementieren, konzentrieren sie sich stattdessen auf eine Reihe häufig verwendeter Funktionen, die gut zusammenarbeiten. Sie sehen so aus:

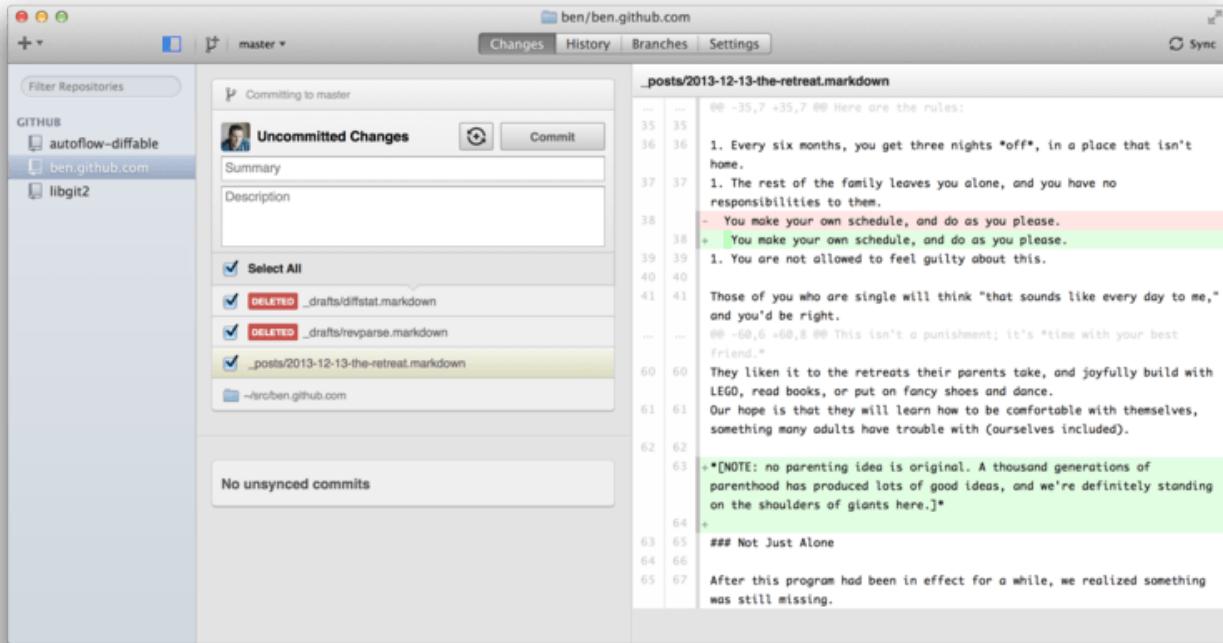


Figure 179. GitHub für macOS

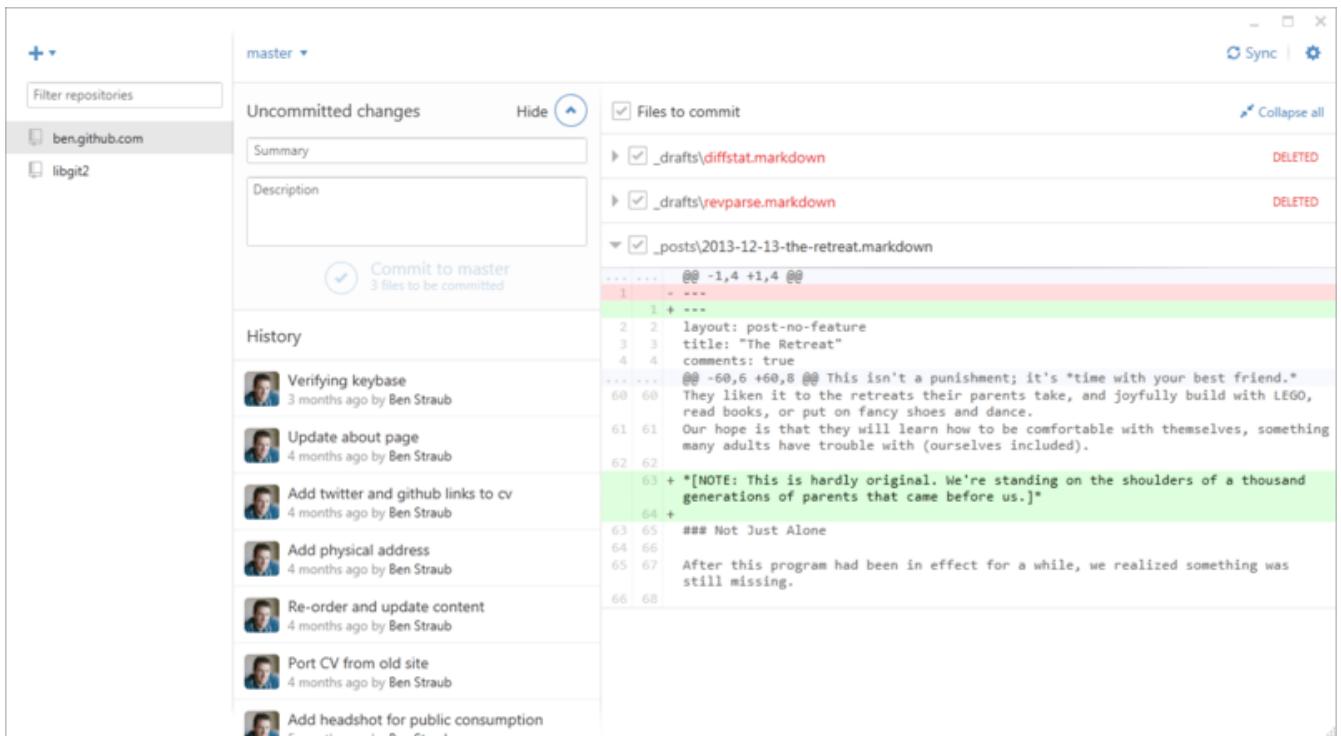


Figure 180. GitHub für Windows

Sie sind so konzipiert, dass sie sehr ähnlich aussehen und funktionieren. Daher werden wir sie in diesem Kapitel wie ein einziges Produkt behandeln. Wir werden keinen detaillierten Überblick über diese Tools geben (sie haben ihre eigene Dokumentation), aber kurz auf die Ansicht „Änderungen“ (in der Sie die meiste Zeit verbringen werden) werden wir eingehen.

- Auf der linken Seite befindet sich die Liste der Repositorys, die der Client verfolgt. Sie können ein Repository hinzufügen (entweder durch Klonen oder lokales Anhängen), indem Sie oben in diesem Bereich auf das Symbol "+" klicken.

- In der Mitte befindet sich ein Commit-Eingabebereich, in den Sie eine Commit-Nachricht eingeben und auswählen können, welche Dateien enthalten sein sollen. Unter Windows wird die Commit-Historie direkt darunter angezeigt. Unter macOS befindet sie sich auf einer separaten Registerkarte.
- Auf der rechten Seite befindet sich eine Diff-Ansicht, die zeigt, was in Ihrem Arbeitsverzeichnis geändert wurde oder welche Änderungen im ausgewählten Commit enthalten waren.
- Zu guter Letzt gibt es oben rechts die Schaltfläche „Synchronisieren“. Hierüber werden sie primär über das Netzwerk interagieren.



Sie benötigen kein GitHub-Konto, um diese Tools verwenden zu können. Das Tool ist designed, um Githubs Services und deren empfohlenen Workflow zu nutzen, sie können damit jedoch problemlos mit jedem Repository arbeiten und Netzwerkoperationen mit jedem Git-Host ausführen.

## Installation

GitHub für Windows kann von <https://windows.github.com> und GitHub für macOS von <https://mac.github.com> heruntergeladen werden. Wenn die Anwendungen zum ersten Mal ausgeführt werden, werden Sie durch alle erstmaligen Git-Einstellungen geführt, z. B. durch die Konfiguration Ihres Namens und Ihrer E-Mail-Adresse. Beide richten vernünftige Standardeinstellungen für viele gängige Konfigurationsoptionen ein, z. B. Caches für Anmeldeinformationen und CRLF-Verhalten.

Beides sind „Evergreen“-Programme – Alle Updates werden automatisch heruntergeladen und im Hintergrund installiert, sobald die Anwendungen geöffnet sind. Dies beinhaltet eine gebündelte Version von Git, was bedeutet, dass Sie sich wahrscheinlich nicht darum kümmern müssen, sie manuell erneut zu aktualisieren. Unter Windows enthält der Client eine Verknüpfung zum Starten von PowerShell mit Posh-git, auf die wir später in diesem Kapitel näher eingehen werden.

Der nächste Schritt besteht darin, dem Tool einige Repositorys zur Verfügung zu stellen, mit denen es arbeiten kann. Der Client zeigt Ihnen eine Liste der Repositorys, auf die Sie auf GitHub zugreifen können, und kann sie in einem Schritt klonen. Wenn Sie bereits über ein lokales Repository verfügen, ziehen Sie dessen Verzeichnis einfach aus dem Finder oder Windows Explorer in das GitHub-Clientfenster. Es wird dann in die Liste der Repositorys auf der linken Seite aufgenommen.

## Empfohlener Workflow

Sobald es installiert und konfiguriert ist, können Sie den GitHub-Client für viele gängige Git-Aufgaben verwenden. Der beabsichtigte Workflow für dieses Tool wird manchmal als „GitHub Flow“ bezeichnet. Wir behandeln dies ausführlicher in [Der GitHub Workflow](#), aber der Kern des Ganzen ist, dass (a) Sie sich auf einen Branch festlegen und (b) sich mit einer remote Repository regelmäßig synchronisieren.

Das Branchmanagement ist einer der Bereiche, in denen sich die beiden Tools unterscheiden. Unter macOS gibt es oben im Fenster eine Schaltfläche zum Erstellen eines neuen Branches:

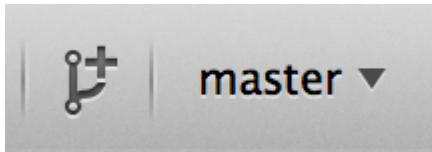


Figure 181. „Create Branch“ Knopf auf macOS

Unter Windows wird dazu der Name des neuen Branches in das Branchwechsel-Widget eingegeben:

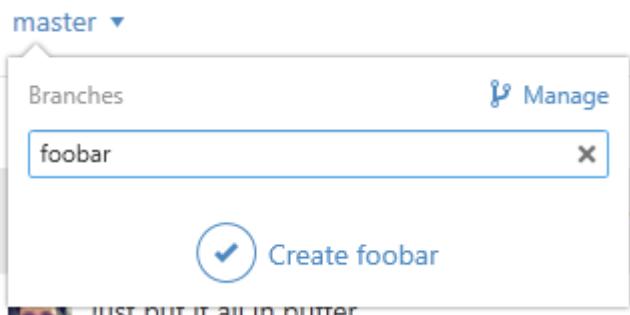


Figure 182. Einen Branch auf Windows erstellen

Sobald Ihre Branch erstellt ist, ist das Erstellen neuer Commits ziemlich einfach. Nehmen Sie einige Änderungen in Ihrem Arbeitsverzeichnis vor. Wenn Sie zum GitHub-Client-Fenster wechseln, wird angezeigt, welche Dateien geändert wurden. Geben Sie eine Commitnachricht ein, wählen Sie die Dateien aus, die Sie einschließen möchten, und klicken Sie auf die Schaltfläche „Commit“ (Strg-Eingabe oder ⌘-Eingabe).

Die Hauptmethode für die Interaktion mit anderen Repositorys über das Netzwerk ist die Funktion „Synchronisieren“. Git verfügt intern über separate Vorgänge zum Verschieben, Abrufen, Zusammenführen und Rebasen. Die GitHub-Clients reduzieren jedoch alle diese Vorgänge zu einer mehrstufigen Funktion. Folgendes passiert, wenn Sie auf die Schaltfläche „Synchronisieren“ klicken:

1. `git pull --rebase`. Wenn das wegen eines merge Konfliktes fehlschlägt, gehe zurück zu `git pull --no-rebase`.
2. `git push`.

Dies ist die häufigste Folge von Netzwerkbefehlen, wenn Sie auf diese Art arbeiten. Wenn Sie sie also zu einem Befehl zusammenfassen, sparen Sie viel Zeit.

## Zusammenfassung

Diese Tools eignen sich sehr gut für den Workflow, für den sie entwickelt wurden. Entwickler und Nichtentwickler können innerhalb von Minuten an einem Projekt zusammenarbeiten, und viele der Best Practices für diese Art von Workflow sind in die Tools integriert. Wenn Ihr Workflow jedoch anders ist oder Sie mehr Kontrolle darüber haben möchten, wie und wann Netzwerkvorgänge ausgeführt werden, empfehlen wir Ihnen, einen anderen Client oder die Befehlszeile zu verwenden.

## Andere GUIs

Es gibt eine Reihe anderer grafischer Git-Clients, die von spezialisierten Einzweck-Tools bis hin zu Apps reichen, die versuchen, alles zu beinhalten, was Git kann. Die offizielle Git-Website enthält eine kuratierte Liste der beliebtesten Clients unter <https://git-scm.com/downloads/guis>. Eine umfassendere Liste finden Sie auf der Git-Wiki-Website unter [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

Visual Studio hat seit Visual Studio 2019 Version 16.8 Git-Tools direkt in die IDE integriert.

Das Tooling unterstützt folgenden Git-Funktionen:

- Erstellen oder klonen von Repositorys.
- Öffnen und durchsuchen des Verlaufs eines Repositorys.
- Branches und Tags erstellen und auschecken.
- Stash-, Stage- und Commit-Änderungen.
- Commits fetchen, pullen, pushen oder synchronisieren.
- Branches zusammenführen und rebasen.
- Zusammenführungskonflikte lösen.
- Unterschiede anzeigen.
- ... und vieles mehr!

Unter [offizielle Dokumentation](#) gibt es weitere Informationen dazu.

## Git in Visual Studio Code

In Visual Studio Code ist Git-Unterstützung bereits integriert. Sie müssen dafür Git Version 2.0.0 (oder höher) installiert haben.

Die Hauptfeatures sind:

- Sehen Sie den Unterschied der Datei, die Sie bearbeiten, in der Seitenleiste.
- In der Git-Statusleiste (unten links) werden der aktuelle Branch, die Dirty-Indikatoren sowie eingehende und ausgehende Commits angezeigt.
- Sie können die gebräuchlichsten Git-Operationen im Editor ausführen:
  - Initialisiere ein Repository.
  - Klonen Sie ein Repository.
  - Erstellen Sie Branches und Tags.
  - Änderungen bereitstellen und festschreiben.
  - Push/Pull/Sync mit einem entfernten Branch.

- Zusammenführungskonflikte lösen.
- Unterschiede anzeigen.
- Mit einer Erweiterung können Sie auch GitHub Pull-Requests bearbeiten: <https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github>.

Die offizielle Dokumentation finden Sie hier: <https://code.visualstudio.com/Docs/editor/versioncontrol>.

## Git in IntelliJ / PyCharm / WebStorm / PhpStorm / RubyMine

JetBrains-IDEs (wie IntelliJ IDEA, PyCharm, WebStorm, PhpStorm, RubyMine und andere) werden mit einem Git-Integrations-Plugin ausgeliefert. Es bietet eine dedizierte Ansicht in der IDE, um mit Git- und GitHub-Pull-Anforderungen zu arbeiten.

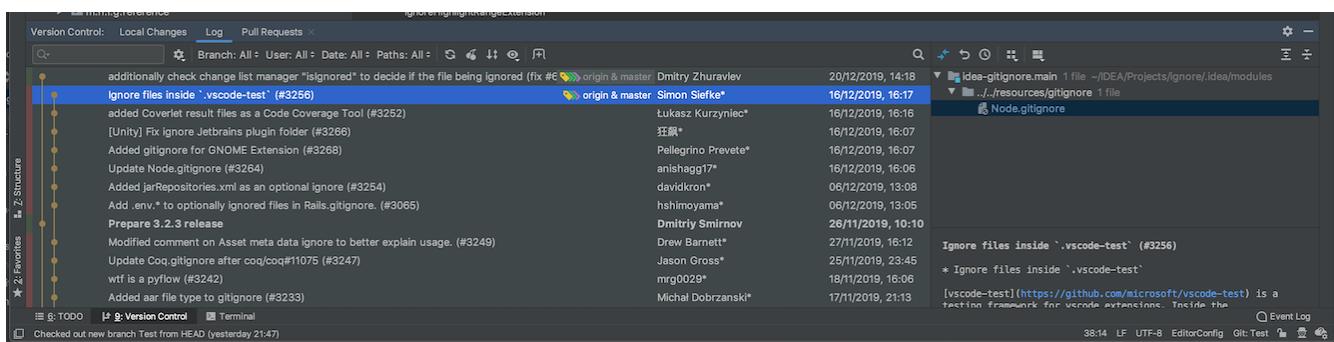


Figure 183. Version Control ToolWindow in JetBrains IDEs

Die Integration basiert auf dem Befehlszeilen-Git-Client und erfordert die Installation eines solchen. Die offizielle Dokumentation finden Sie unter <https://www.jetbrains.com/help/idea/using-git-integration.html>.

## Git in Sublime Text

Sublime Text hat ab Version 3.2 eine Git-Integration im Editor.

Die Funktionen sind:

- In der Seitenleiste wird der Git-Status von Dateien und Ordnern mit einem Abzeichen/Symbol angezeigt.
- Dateien und Ordner in Ihrer .gitignore-Datei werden in der Seitenleiste ausgeblendet.
- In der Statusleiste sehen Sie den aktuellen Git-Branch und wie viele Änderungen Sie vorgenommen haben.
- Alle Änderungen an einer Datei sind jetzt über Markierungen in der Seitenleiste sichtbar.
- Sie können einen Teil der Funktionen des Sublime Merge Git-Clients in Sublime Text verwenden. Dies setzt voraus, dass Sublime Merge installiert ist. Siehe: <https://www.sublimemerge.com/>.

Die offizielle Dokumentation zu Sublime Text finden Sie hier:  
[https://www.sublimetext.com/docs/3/git\\_integration.html](https://www.sublimetext.com/docs/3/git_integration.html).

## Git in Bash

Wenn Sie ein Bash-Benutzer sind, können Sie einige der Funktionen Ihrer Shell nutzen, um Ihren Umgang mit Git viel angenehmer zu gestalten. Git wird mit Plugins für mehrere Shells ausgeliefert. Sie sind jedoch nicht standardmäßig aktiviert.

Zunächst müssen Sie eine Kopie der Vervollständigungsdatei aus dem Quellcode der von Ihnen verwendeten Git-Version abrufen. Überprüfen Sie Ihre Version, indem Sie `git version` eingeben, und verwenden Sie dann `git checkout tags vX.Y.Z`, wobei vX.Y.Z der Version von Git entspricht, die Sie verwenden. Kopieren Sie die Datei `contrib/completion/git-completion.bash` an einen geeigneten Ort, z.B. in Ihr Home-Verzeichnis und fügen Sie folgendes zu Ihrer `.bashrc` hinzu:

```
. ~/git-completion.bash
```

Wechseln Sie anschließend in ein Git-Repository und geben Sie Folgendes ein:

```
$ git chec<tab>
```

...und Bash vervollständigt dies automatisch zu `git checkout`. Dies funktioniert mit allen Unterbefehlen, Befehlszeilenparametern sowie Remote und Referenznamen von Git, sofern dies erforderlich ist.

Es ist auch nützlich, die Eingabeaufforderung so anzupassen, dass Informationen zum Git-Repository des aktuellen Verzeichnisses angezeigt werden. Dies kann so einfach oder komplex sein, wie Sie möchten. Im Allgemeinen gibt es jedoch einige wichtige Informationen, die die meisten Benutzer benötigen, z. B. den aktuellen Branch und den Status des Arbeitsverzeichnisses. Um diese zu Ihrer Eingabeaufforderung hinzuzufügen, kopieren Sie einfach die Datei `contrib/completion/git-prompt.sh` aus dem Quellrepository von Git in Ihr Home Verzeichnis. Fügen Sie Ihre `.bashrc` Datei folgendes hinzu:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(_git_ps1 "(%s)")\$ '
```

Das `\w` bedeutet das Ausgeben des aktuellen Arbeitsverzeichnisses, das `\$` gibt den `$` Teil der Eingabeaufforderung aus und `_git_ps1 "(% s)"` ruft die von `git-prompt.sh` bereitgestellte Funktion mit einem Formatierungsargument auf. Jetzt sieht Ihre Bash-Eingabeaufforderung so aus, wenn Sie sich irgendwo in einem Git Projekt befinden:



Figure 184. Customized bash prompt

Beide Skripte werden mit hilfreicher Dokumentation geliefert. Weitere Informationen finden Sie in den Skripten `git-completion.bash` und `git-prompt.sh` selbst.

## Git in Zsh

Zsh wird ebenfalls mit einer Tab-Completion-Bibliothek für Git ausgeliefert. Um es zu benutzen, fügen Sie einfach `autoload -Uz compinit && compinit` in Ihrer `.zshrc` Datei ein. Die Oberfläche von Zsh ist etwas leistungsfähiger als die von Bash:

```
$ git che<tab>
check-attr      -- display gitattributes information
check-ref-format -- ensure that a reference name is well formed
checkout        -- checkout branch or paths to working tree
checkout-index   -- copy files from index to working directory
cherry          -- find commits not merged upstream
cherry-pick     -- apply changes introduced by some existing commits
```

Mehrdeutige Tab-Vervollständigungen werden nicht nur aufgelistet. Sie haben hilfreiche Beschreibungen und Sie können durch die Liste navigieren, indem Sie wiederholt auf Tab drücken. Dies funktioniert mit Git-Befehlen, ihren Argumenten und Namen von Objekten im Repository (wie Refs und Remotes) sowie mit Dateinamen und all den anderen Dingen, die Zsh mit Tabulatoren vervollständigen kann.

Zsh wird mit einem Framework zum Abrufen von Informationen von Versionskontrollsystmen namens „vcs\_info“ ausgeliefert. Fügen Sie Ihrer `~/.zshrc` Datei die folgenden Zeilen hinzu, um den Namen des Branches in die Eingabeaufforderung auf der rechten Seite aufzunehmen:

```
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'
```

Dies führt zu einer Anzeige des aktuellen Branches auf der rechten Seite des Terminalfensters, wenn sich Ihre Shell in einem Git-Repository befindet. Die linke Seite wird natürlich auch unterstützt; heben Sie einfach die Kommentierung zur PROMPT-Zuweisung auf. Es sieht in etwa so aus:

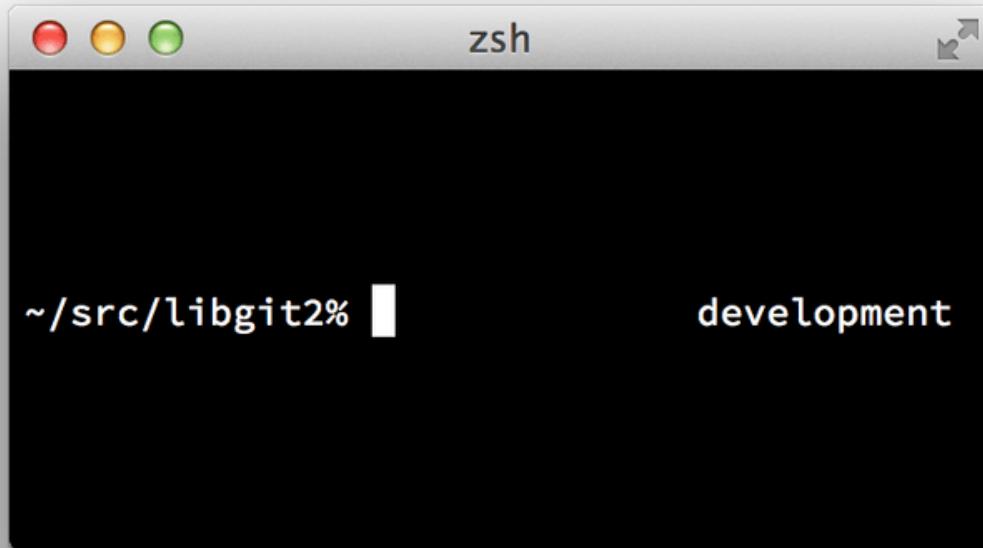


Figure 185. Benutzerdefinierter zsh Prompt

Weitere Informationen zu `vcs_info` finden Sie in der Dokumentation in der `zshcontrib(1)` Manpage oder online unter <https://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information>.

Anstelle von `vcs_info` bevorzugen Sie möglicherweise das im Lieferumfang von Git enthaltene Skript zur Anpassung der Eingabeaufforderung mit dem Namen `git-prompt.sh`. Weitere Informationen finden Sie unter <https://github.com/git/git/blob/master/contrib/completion/git-prompt.sh>. `git-prompt.sh` ist sowohl mit Bash als auch mit Zsh kompatibel.

Zsh ist mächtig genug, dass es ganze Frameworks gibt, um es besser zu machen. Eins von ihnen heißt "oh-my-zsh" und ist unter <https://github.com/robbyrussell/oh-my-zsh> zu finden. Das Plugin-System von oh-my-zsh verfügt über eine leistungsstarke Git-Tab-Vervollständigung und eine Vielzahl von PROMPT „Themen“, von denen viele Versionskontrolldaten anzeigen. Ein Beispiel für ein oh-my-zsh Theme ist nur ein Beispiel dafür, was mit diesem System möglich ist.

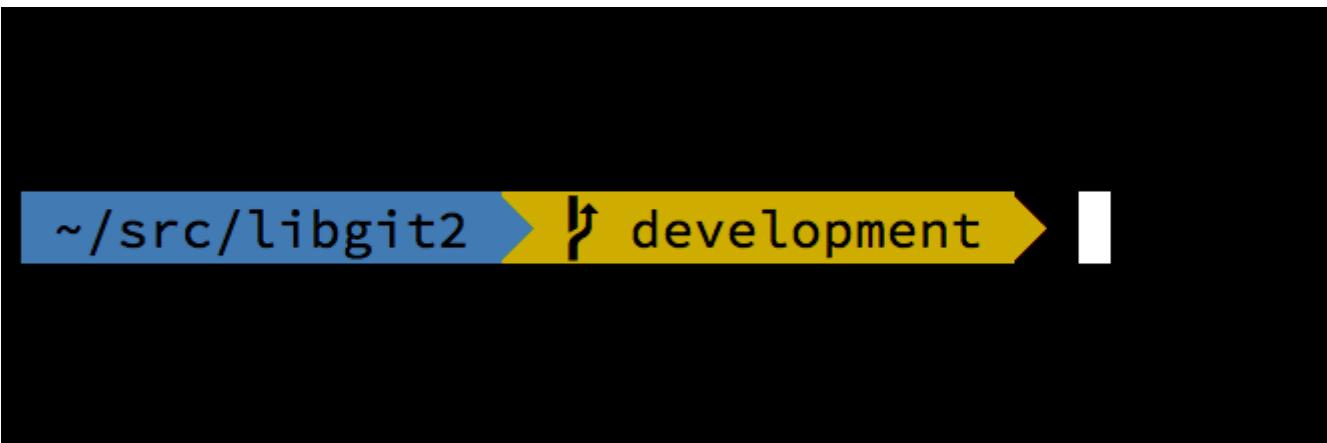


Figure 186. Ein Beispiel für ein oh-my-zsh Theme

## Git in PowerShell

Das Legacy-Befehlszeilenterminal unter Windows ([cmd.exe](#)) bietet keine Git Integration, aber wenn Sie PowerShell verwenden, haben Sie Glück. Dies funktioniert auch, wenn Sie PowerShell Core unter Linux oder macOS ausführen. Ein Paket mit dem Namen posh-git (<https://github.com/dahlbyk/posh-git>) bietet leistungsstarke Funktionen zur Tab Vervollständigung sowie eine erweiterte Eingabeaufforderung, mit der Sie den Überblick über Ihren Repository-Status behalten. Folgendermaßen sieht es aus:

```
posh-git [master] ~ PowerShell 6.1.1 64-bit (18588)
~\GitHub\dahlbyk\posh-git [master ≡ +1 ~0 -0 | +0 ~1 -0 !]>
```

Figure 187. PowerShell with Posh-git

## Installation

### Voraussetzungen (Nur für Windows)

Bevor Sie PowerShell-Skripte auf Ihrem Computer ausführen können, müssen Sie Ihre lokale ExecutionPolicy auf RemoteSigned (im Allgemeinen alles außer Undefiniert und Eingeschränkt) einstellen. Wenn Sie AllSigned anstelle von RemoteSigned auswählen, müssen auch lokale Skripte (Ihre eigenen) digital signiert sein, damit sie ausgeführt werden können. Mit RemoteSigned müssen nur Skripte signiert werden, bei denen „ZoneIdentifier“ auf Internet eingestellt ist (sie wurden aus dem Web heruntergeladen), andere nicht. Wenn Sie ein Administrator sind und es für alle Benutzer auf diesem Computer festlegen möchten, verwenden Sie „-Scope LocalMachine“. Wenn Sie ein normaler Benutzer ohne Administratorrechte sind, können Sie „-Scope CurrentUser“ verwenden,

um es nur für sich selber festzulegen.

Weitere Informationen zu PowerShell-Bereichen unter: [https://docs.microsoft.com/de-de/powershell/module/microsoft.powershell.core/about/about\\_scopes](https://docs.microsoft.com/de-de/powershell/module/microsoft.powershell.core/about/about_scopes).

Weitere Informationen zu PowerShell ExecutionPolicy unter: <https://docs.microsoft.com/de-de/powershell/module/microsoft.powershell.security/set-executionpolicy>.

Um den Wert von **ExecutionPolicy** für alle Benutzer auf **RemoteSigned** zu setzen, verwenden Sie den nächsten Befehl:

```
> Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy RemoteSigned -Force
```

## PowerShell Galerie

Wenn Sie mindestens PowerShell 5 oder PowerShell 4 mit PackageManagement installiert haben, können Sie posh-git mithilfe des Paketmanagers installieren.

Weitere Informationen zur PowerShell-Galerie unter: <https://docs.microsoft.com/de-de/powershell/scripting/gallery/overview>.

```
> Install-Module posh-git -Scope CurrentUser -Force  
> Install-Module posh-git -Scope CurrentUser -AllowPrerelease -Force # Newer beta  
version with PowerShell Core support
```

Wenn Sie posh-git für alle Benutzer installieren möchten, verwenden Sie stattdessen „-Scope AllUsers“ und führen Sie den Befehl über eine PowerShell-Konsole mit admin Rechten aus. Wenn der zweite Befehl fehlschlägt und ein Fehler wie **Module 'PowerShellGet' was not installed by using Install-Module**, müssen Sie zuerst einen anderen Befehl ausführen:

```
> Install-Module PowerShellGet -Force -SkipPublisherCheck
```

Dann können Sie zurückgehen und es erneut versuchen. Dies liegt daran, dass die mit Windows PowerShell gelieferten Module mit einem anderen Veröffentlichungszertifikat signiert sind.

## Aktualisierung der PowerShell Eingabeaufforderung

Um Git-Informationen in Ihre Eingabeaufforderung aufzunehmen, muss das Posh-Git-Modul importiert werden. Wenn posh-git bei jedem Start von PowerShell importiert werden soll, führen Sie den Befehl Add-PoshGitToProfile aus, mit dem die import-Anweisung in Ihr \$profile-Skript eingefügt wird. Dieses Skript wird jedes Mal ausgeführt, wenn Sie eine neue PowerShell-Konsole öffnen. Beachten Sie, dass es mehrere **\$profile** Skripte gibt. Z.B. eins für die Konsole und ein separates für die ISE.

```
> Import-Module posh-git  
> Add-PoshGitToProfile -AllHosts
```

## Vom Quellcode

Laden Sie einfach eine posh-git-Version von (<https://github.com/dahlbyk/posh-git>) herunter und entpacken Sie sie. Importieren Sie dann das Modul unter Verwendung des vollständigen Pfads zur Datei posh-git.ps1:

```
> Import-Module <path-to-uncompress-folder>\src\posh-git.ps1  
> Add-PoshGitToProfile -AllHosts
```

Dies fügt die richtige Zeile zu Ihrer `profile.ps1` Datei hinzu und posh-git wird beim nächsten Öffnen von PowerShell aktiviert.

Eine Beschreibung der in der Eingabeaufforderung angezeigten Git-Statuszusammenfassungsinformationen finden Sie unter: <https://github.com/dahlbyk/posh-git/blob/master/README.md#git-status-summary-information> Weitere Informationen zum Anpassen Ihrer posh-git-Eingabeaufforderung finden Sie unter: <https://github.com/dahlbyk/posh-git/blob/master/README.md#customization-variables>.

## Zusammenfassung

Sie haben gelernt, wie Sie Gits Leistungsfähigkeit in den Tools nutzen, die Sie während Ihrer täglichen Arbeit verwenden, und wie Sie aus Ihren eigenen Programmen auf Git-Repositorys zugreifen können.

# Appendix B: Git in Ihre Anwendungen einbetten

Wenn Ihre Anwendung für Software-Entwickler gedacht ist, stehen die Chancen gut, dass sie von der Integration mit der Quellcode-Versionsverwaltung profitieren kann. Auch Anwendungen, die nicht für Entwickler bestimmt sind, wie z.B. Texteditoren, könnten potenziell von Funktionen der Versionskontrolle profitieren. Das Git-System funktioniert sehr gut für viele unterschiedliche Einsatzszenarien.

Wenn Sie Git in Ihre Anwendung integrieren müssen, haben Sie im Wesentlichen zwei Möglichkeiten: eine Shell zu erzeugen und damit das Git-Kommandozeilenprogramm aufzurufen oder eine Git-Bibliothek in Ihre Anwendung einzubetten. Hier werden wir die Befehlszeilenintegration und einige der beliebtesten, integrierbaren Git-Bibliotheken behandeln.

## Die Git-Kommandozeile

Die eine Möglichkeit besteht darin, einen Shell-Prozess zu erzeugen und das Git-Kommandozeilen-Tool zu verwenden, um die Arbeit zu erledigen. Das hat den Vorteil, dass es kanonisch ist und alle Funktionen von Git unterstützt werden. Außerdem ist dieses Verfahren ganz einfach, da die meisten Laufzeit-Umgebungen eine vergleichsweise unkomplizierte Möglichkeit haben, einen Prozess mit Kommandozeilen-Argumenten aufzurufen. Dieser Weg hat jedoch auch einige Nachteile.

Zum einen ist die gesamte Textausgabe in Klartext. Das bedeutet, dass Sie das gelegentlich wechselnde AusgabefORMAT von Git analysieren müssen, um Fortschritts- und Ergebnisinformationen zu erfassen, was ineffizient und fehleranfällig sein kann.

Ein weiterer Nachteil ist die unzureichende Fehlerkorrektur. Wenn ein Repository irgendwie beschädigt ist oder der Benutzer einen fehlerhaften Konfigurationswert eingestellt hat, verweigert Git einfach die Durchführung vieler Operationen.

Noch ein anderer ist das Prozessmanagement. Git erfordert die Verwaltung einer Shell-Umgebung in einem separaten Prozess, was zu unerwünschter Komplexität führen kann. Der Versuch, viele dieser Prozesse zu koordinieren (insbesondere wenn mehrere Prozesse auf das gleiche Repository zugreifen wollen), kann eine ziemliche Herausforderung darstellen.

## Libgit2

Eine weitere Möglichkeit, die Ihnen zur Verfügung steht, ist die Verwendung von Libgit2. Mit Libgit2 ist eine von Abhängigkeiten freie Implementierung von Git, wobei der Schwerpunkt auf einer ansprechenden API zur Integration in andere Anwendungen liegt. Sie finden das Programm unter <https://libgit2.org>.

Lassen Sie uns zunächst einen Blick auf die C-API werfen. Hier ist eine schnelle Tour:

```
// Open a repository
```

```

git_repository *repo;
int error = git_repository_open(&repo, "/path/to/repository");

// Dereference HEAD to a commit
git_object *head_commit;
error = git_revparse_single(&head_commit, repo, "HEAD^{commit}");
git_commit *commit = (git_commit*)head_commit;

// Print some of the commit's properties
printf("%s", git_commit_message(commit));
const git_signature *author = git_commit_author(commit);
printf("%s <%s>\n", author->name, author->email);
const git_oid *tree_id = git_commit_tree_id(commit);

// Cleanup
git_commit_free(commit);
git_repository_free(repo);

```

Die ersten beiden Zeilen öffnen ein Git-Repository. Der `git_repository` Typ repräsentiert einen Handle auf ein Repository mit Cache im Arbeitsspeicher. Diese Methode ist die am einfachsten anzuwendende, wenn man den genauen Pfad zum Arbeitsverzeichnis oder zum `.git` Ordner eines Repositorys kennt. Es gibt auch `git_repository_open_ext`, das Optionen für die Suche enthält, `git_clone` und seine Verwandten für das Erstellen eines lokalen Klons eines Remote-Repositorys und `git_repository_init` für das Erstellen eines völlig neuen Repositorys.

Der zweite Teil des Codes verwendet die `rev-parse` Syntax (siehe auch [Branch Referenzen](#)), um den Commit zu erhalten, auf den HEAD letztlich zeigt. Der zurückgegebene Typ ist ein `git_object` Pointer, der ein Objekt repräsentiert, das in der Git-Objekt-Datenbank für ein Repository steht. Bei `git_object` handelt es sich eigentlich um einen „Parent“-Typ für mehrere verschiedene Arten von Objekten. Das Speicher-Layout für jeden der „Child“-Typen ist das Gleiche wie bei `git_object`, so dass Sie sicher auf den richtigen Typ verweisen können. In diesem Fall würde `git_object_type(commit)` ein `GIT_OBJ_COMMIT` zurückgeben, so dass es sicher auf einen `git_commit` Pointer zu zeigt.

Der nächste Abschnitt zeigt, wie auf die Eigenschaften des Commits zugegriffen werden kann. Die letzte Zeile, hier, verwendet einen `git_oid` Typ; das ist das Anzeige-Format von Libgit2 für einen SHA-1-Hash.

From this sample, a couple of patterns have started to emerge:

- Wenn Sie einen Pointer definieren und ihm eine Referenz in einem Libgit2-Aufruf übergeben, wird dieser Aufruf wahrscheinlich einen ganzzahligen Fehlercode zurückgeben. Ein Wert von `0` zeigt den Erfolg an; alles andere signalisiert einen Fehler.
- Auch wenn Libgit2 einen Zeiger für Sie erstellt, Sie sind dafür verantwortlich, ihn freizugeben.
- Wenn Libgit2 einen `const` Pointer aus einem Aufruf zurückgegeben hat, müssen Sie ihn nicht freigeben, aber er wird ungültig, wenn das Objekt, zu dem er gehört, freigegeben wird.
- Das Schreiben in **C** kann ein bisschen mühselig sein.

Das letzte Argument bedeutet, dass es nicht sehr wahrscheinlich ist, dass Sie einen C-Code schreiben werden, wenn Sie Libgit2 verwenden. Glücklicherweise gibt es eine Reihe von sprachspezifischen Anbindungen, die es ziemlich einfach machen, mit Git-Repositorys in Ihrer spezifischen Programmier-Sprache und -Umgebung zu arbeiten. Schauen wir uns das obige Beispiel an, das mit den Ruby-Anbindungen für Libgit2 geschrieben wurde, die Rugged genannt werden und unter <https://github.com/libgit2/rugged> zu finden sind.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

Wie Sie sehen können, ist der Code viel weniger verwirrend. Erstens verwendet Rugged Ausnahmeregeln; es kann Dinge wie `ConfigError` oder `ObjectError` aktivieren, um so Fehlerzustände zu signalisieren. Zweitens gibt es keine explizite Freigabe von Ressourcen, da in Ruby eine automatische Speicherbereinigung aktiv ist. Schauen wir uns ein etwas komplexeres Beispiel an: einen Commit von Grund auf neu zu erstellen

```
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 Einen neuen Blob erstellen, der den Inhalt einer neuen Datei enthält.

② Ergänzen Sie den Index mit dem ersten Wert des Commit-Baums und fügen Sie die Datei im Pfad `newfile.txt` hinzu.

③ Damit wird ein neuer Baum in der Objektdatenbank (ODB) erstellt und für den neuen Commit verwendet.

- ④ Wir verwenden die gleiche Signatur für das Autoren- und das Committer-Feld.
- ⑤ Die Commit-Beschreibung.
- ⑥ Wenn Sie einen Commit erstellen, müssen Sie die Elternteile des neuen Commits angeben. Dazu wird das Ende von HEAD für den einzelnen Elternteil verwendet.
- ⑦ Rugged (und Libgit2) können optional eine Referenz aktualisieren, wenn sie einen Commit erstellen.
- ⑧ Der Rückgabewert entspricht dem SHA-1-Hash eines neuen Commit-Objekts, mit dem Sie dann ein **Commit** Objekt erzeugen können.

Der Ruby-Code ist zwar ganz nett und klar und da Libgit2 die Ausführung übernimmt, wird dieser Code auch relativ schnell laufen. Falls Sie kein Ruby-Anhänger sind, dann zeigen wir Alternativen in [Andere Anbindungen](#).

## Erweiterte Funktionalität

Libgit2 verfügt über einige Funktionen, die nicht zum eigentlichen Umfang von Git gehören. Ein Beispiel ist die Erweiterbarkeit: Libgit2 erlaubt es Ihnen, benutzerdefinierte „Backends“ für unterschiedliche Betriebsarten anzubieten, so dass Sie Dinge auf eine andere Art speichern können, als es mit Git möglich ist. Libgit2 erlaubt benutzerdefinierte Backends unter anderem zum Konfigurieren, die Ref-Speicherung und für die Objektdatenbank.

Schauen wir uns an, wie das genau funktioniert. Der untenstehende Code ist aus einer Reihe von Backend-Beispielen des Libgit2-Teams entlehnt (die unter <https://github.com/libgit2/libgit2-backends> zu finden sind). So wird ein benutzerdefiniertes Backend für die Objektdatenbank eingerichtet:

```
git_odb *odb;
int error = git_odb_new(&odb); ①

git_odb_backend *my_backend;
error = git_odb_backend_mine(&my_backend, /*...*/); ②

error = git_odb_add_backend(odb, my_backend, 1); ③

git_repository *repo;
error = git_repository_open(&repo, "some-path");
error = git_repository_set_odb(repo, odb); ④
```

*Beachten Sie, dass Fehler zwar erfasst, aber nicht bearbeitet werden. Wir hoffen, dass Ihr Code besser als unserer ist.*

- ① Erzeugen einer leeren Objektdatenbank (ODB) „Frontend“, die als eine Art Container für die „Backends“ dient, mit denen die eigentliche Arbeit erledigt wird.
- ② Ein benutzerdefiniertes ODB-Backend einrichten.
- ③ Das Backend dem Frontend hinzufügen.

- ④ Ein Repository öffnen und so einrichten, dass es unsere ODB zum Suchen von Objekten verwendet.

Aber was ist das für ein `git_odb_backend_mine`? Nun, das ist der Konstruktor für Ihre eigene ODB-Implementation. Sie können dort machen, was immer Sie wollen, solange Sie die `git_odb_backend` Struktur richtig eingeben. Hier sieht man, wie es ausschauen könnte:

```
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;
}
```

Die wichtigste Bedingung dabei ist, dass das erste Element von `my_backend_struct` eine `git_odb_backend` Struktur sein muss. Dadurch wird sichergestellt, dass das Speicherlayout dem entspricht, was der Libgit2-Code erwartet. Der restliche Teil ist beliebig. Diese Struktur kann so groß oder klein sein, wie Sie es brauchen.

Die Initialisierungsfunktion weist der Struktur Speicherplatz zu, richtet den benutzerdefinierten Kontext ein und fügt dann die Mitglieder der von ihr unterstützten `parent` Struktur ein. Schauen Sie sich die Datei `include/git2/sys/odb_backend.h` im Libgit2-Quelltext an, um einen vollständigen Satz von Aufrufsignaturen zu erhalten. Ihr spezieller Anwendungszweck wird Ihnen helfen, zu bestimmen, welche dieser Signaturen Sie dann benötigen.

## Andere Anbindungen

Libgit2 hat Anbindungen für viele Programmier-Sprachen. Hier zeigen wir eine kleine Auswahl der wichtigsten Pakete, die zum Zeitpunkt der Erstellung dieses Textes verfügbar waren. Es gibt Bibliotheken für viele andere Sprachen, darunter C++, Go, Node.js, Erlang und die JVM, die alle in unterschiedlich weit entwickelt sind. Die offizielle Kollektion von Anbindungen kann in den

Repositorys unter <https://github.com/libgit2> gefunden werden. Der Code, den wir schreiben werden, wird die Commit-Nachricht des Commits zurückliefern, auf den HEAD letztlich zeigt (etwa so wie `git log -1`).

## LibGit2Sharp

Wenn Sie eine .NET- oder Mono-Anwendung schreiben, ist LibGit2Sharp (<https://github.com/libgit2/libgit2sharp>) das Richtige für Sie. Die Anbindungen sind in C# geschrieben und es wurde große Sorgfalt darauf verwendet die unverarbeiteten Libgit2-Aufrufe in CLR-APIs mit der ursprünglichen Funktionalität zu verpacken. So sieht unser Beispielprogramm aus:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

Für Windows-Desktop-Anwendungen gibt es außerdem ein NuGet-Paket, das Ihnen einen schnellen Einstieg ermöglicht.

## objective-git

Wenn Ihre Anwendung auf einer Apple-Plattform läuft, verwenden Sie wahrscheinlich Objective-C als Programmiersprache. Objective-Git (<https://github.com/libgit2/objective-git>) ist der Name der Libgit2-Anbindungen für diese Umgebung. Das Beispielprogramm sieht wie folgt aus:

```
GTRepository *repo =  
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"]  
error:NULL];  
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git ist vollständig kompatibel mit Swift. Sie brauchen also keine Angst zu haben, wenn Sie Objective-C aufgegeben haben.

## pygit2

Die Anbindungen für Libgit2 in Python heißen Pygit2 und sind unter <https://www.pygit2.org> zu finden. Hier ist unser Beispielprogramm:

```
pygit2.Repository("/path/to/repo") # open repository  
    .head                      # get the current branch  
    .peel(pygit2.Commit)        # walk down to the commit  
    .message                   # read the message
```

## Weiterführende Informationen

Eine vollständige Beschreibung der Funktionen von Libgit2 liegt natürlich nicht im Rahmen dieses Buches. Wenn Sie weitere Informationen über Libgit2 selbst wünschen, finden Sie eine API-Dokumentation unter <https://libgit2.github.com/libgit2> und eine Reihe von Anleitungen unter <https://libgit2.github.com/docs>. Für die anderen Anbindungen schauen Sie sich das enthaltene

README und die Tests an. Häufig gibt es auch kleine Tutorials und Hinweise zum Weiterlesen.

## JGit

Wenn Sie Git aus einem Java-Programm heraus verwenden möchten, gibt es eine voll funktionsfähige Git-Bibliothek mit der Bezeichnung JGit. Dabei handelt es sich um eine vergleichsweise vollständige Implementierung von Git, die ausschließlich in Java geschrieben wurde und in der Java-Community weit verbreitet ist. Das JGit-Projekt ist unter dem Dach von Eclipse angesiedelt, und seine Homepage ist unter <https://www.eclipse.org/jgit/> zu finden.

### Die Einrichtung

Es gibt eine Reihe von Möglichkeiten, Ihr Projekt mit JGit zu verbinden und mit dem Schreiben von Code dafür zu beginnen. Die wahrscheinlich einfachste ist die Verwendung von Maven – die Integration wird durch das Hinzufügen des folgenden Snippets zum `<dependencies>` Tag (dt. Abhängigkeiten) in Ihrer pom.xml Datei erreicht:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

Die `version` wird höchstwahrscheinlich schon weiter fortgeschritten sein, wenn Sie das hier lesen. Unter <https://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit> finden Sie aktuelle Informationen zum Repository. Sobald dieser Schritt abgeschlossen ist, wird Maven automatisch die von Ihnen benötigten JGit-Bibliotheken herunterladen und verwenden.

Wenn Sie die binären Abhängigkeiten lieber selbst verwalten möchten, sind vorkompilierte JGit-Binärdateien unter <https://www.eclipse.org/jgit/download> erhältlich. Sie können diese in Ihr Projekt einbauen, indem Sie einen Befehl wie den folgenden ausführen:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

### Plumbing (Basisbefehle)

JGit hat zwei grundsätzliche API-Ebenen: Basis und Standard (plumbing und porcelain). Die Terminologie dafür stammt von Git direkt und JGit ist in etwa die gleichen Bereiche unterteilt. Standardbefehl-APIs bieten ein benutzerfreundliches Front-End für allgemeine Funktionen auf Benutzerebene (die Art von Aktionen, für die ein normaler Benutzer das Git-Befehlszeilen-Tool verwenden würde). Die Basisbefehl-APIs dienen der direkten Interaktion mit Repository-Objekten auf der unteren Anwendungsebene.

Der Ausgangspunkt für die meisten JGit-Sitzungen ist die Klasse `Repository`. Das erste, was Sie tun sollten, ist davon eine Instanz zu erstellen. Für ein dateisystem-basiertes Repository (ja, JGit erlaubt

andere Speichermodelle) wird das mit dem `FileRepositoryBuilder` erreicht:

```
// 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();
```

Der Builder verfügt über ein flexibles API, um alle notwendigen Funktionen zum Auffinden eines Git-Repositorys bereitzustellen, unabhängig von der Frage, wo Ihr Programm sich genau befindet. Er kann Umgebungsvariablen verwenden (`.readEnvironment()`), von einem Ort im Arbeitsverzeichnis starten und suchen (`.setWorkTree(...).findGitDir()`) oder einfach, wie oben beschrieben, ein bekanntes `.git` Verzeichnis öffnen.

Sobald Sie eine `Repository` Instanz eingerichtet haben, können Sie alles Erdenkliche damit machen. Hier ist eine kurze Aufstellung:

```
// 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");
```

Hier gibt es eine Menge zu sagen, lassen Sie uns die Abschnitte nacheinander durchgehen.

Die erste Zeile erhält einen Pointer auf die Referenz `master`. JGit erfasst automatisch den *aktuellen master Ref*, der bei `refs/heads/master` liegt und gibt ein Objekt zurück, mit dem Sie Informationen über die Referenz fatchen können. Sie können den Namen (`.getName()`) und entweder das Zielobjekt einer direkten Referenz (`.getObjectId()`) oder die Referenz, auf die eine symbolische Referenz zeigt (`.getTarget()`), erhalten. Ref-Objekte werden auch zur Darstellung von Tag-Refs und -Objekten verwendet, so dass Sie abfragen können, ob der Tag „gepeelt“ ist, d.h. ob er auf das endgültige Ziel einer (potenziell langen) Kette von Tag-Objekten zeigt.

Die zweite Zeile ermittelt das Ziel der `master` Referenz, die als ObjectId-Instanz zurückgegeben wird. ObjectId repräsentiert den SHA-1-Hash eines Objekts, der in der Objektdatenbank von Git möglicherweise vorhanden sein könnte. Die dritte Zeile ist vergleichbar, zeigt aber, wie JGit die Rev-Parse-Syntax behandelt (mehr dazu in [Branch Referenzen](#)). Sie können jeden beliebigen Objektbezeichner übergeben, den Git versteht und JGit gibt entweder eine gültige ObjectId für dieses Objekt oder `null` zurück.

Die nächsten beiden Zeilen zeigen, wie der Rohinhalt eines Objekts geladen wird. In diesem Beispiel rufen wir `ObjectLoader.copyTo()` auf, um den Inhalt des Objekts direkt nach stdout zu übertragen. Der ObjectLoader verfügt jedoch auch über Funktionen, um den Typ und die Größe eines Objekts zu lesen und es als Byte-Array zurückzugeben. Für größere Objekte (bei denen `true` den Wert `.isLarge()` zurückgibt) können Sie `.openStream()` aufrufen, um ein InputStream-ähnliches Objekt zu erhalten, das die Rohdaten des Objekts lesen kann, ohne alles auf einmal in den Arbeitsspeicher zu ziehen.

Die nächsten paar Zeilen beschreiben, was man für die Erstellung eines neuen Branchs benötigt. Wir generieren eine RefUpdate-Instanz, konfigurieren einige Parameter und rufen `.update()` auf, um die Änderung anzustoßen. Direkt danach folgt der Code zum Löschen desselben Branches. Beachten Sie, dass `.setForceUpdate(true)` erforderlich ist, damit das funktioniert. Ansonsten gibt der Aufruf von `.delete()` den Wert `REJECTED` zurück, und es passiert nichts.

Die letzten Beispielzeilen zeigen, wie der Wert `user.name` aus den Git-Konfigurationsdateien abgerufen werden kann. Diese Config-Instanz verwendet das Repository, das wir zuvor für die lokale Konfiguration geöffnet haben, erkennt auch die Dateien der Global- und System-Konfiguration. Sie übernimmt automatisch die Werte aus diesen Dateien.

Das ist nur ein kleiner Ausschnitt der vollständigen API für die Sanitärtechnik. Es sind noch viele weitere Methoden und Klassen verfügbar. Auch die Art und Weise, wie JGit Fehler behandelt, wird hier nicht aufgezeigt. Das geschieht nämlich über die Verwendung von Exceptions. JGit-APIs werfen manchmal Standard-Java-Exceptions aus (wie `IOException`), aber es gibt eine Vielzahl von JGit-spezifischen Exception-Typen, die ebenfalls zur Verfügung stehen (wie z.B. `NoRemoteRepositoryException`, `CorruptObjectException` und `NoMergeBaseException`).

## Porcelain (Standardbefehle)

Die Basisbefehl-APIs sind fast komplett. Es kann allerdings umständlich sein, sie hintereinander aufzureihen, um allgemeine Aufgaben zu erfüllen, wie das Hinzufügen einer Datei zum Index oder ein neuer Commit. JGit bietet einen erweiterten Satz von APIs, die dabei helfen können. Der Ausgangspunkt für diese APIs ist die Klasse `Git`:

```
Repository repo;  
// construct repo...  
Git git = new Git(repo);
```

Die Git-Klasse verfügt über einen feinen Satz von high-level *Builder*-Style-Methoden, die zur Erstellung einiger ziemlich komplexer Verhaltensweisen verwendet werden können. Schauen wir uns ein Beispiel an – wir wollen so etwas wie `git ls-remote` machen:

```
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());  
}
```

Das ist ein typisches Muster mit der Git-Klasse. Die Methoden geben ein Befehlsobjekt zurück, mit dem Sie Methodenaufrufe verketteten können, um Parameter zu setzen, die beim Aufruf von `.call()` ausgeführt werden. Hier befragen wir den `origin` Remote nach Tags, nicht nach Heads. Beachten Sie auch die Verwendung des Objekts `CredentialsProvider` zur Authentifizierung.

Viele andere Befehle sind über die Git-Klasse verfügbar, einschließlich, aber nicht beschränkt auf `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert` und `reset`.

## Weiterführende Informationen

Das ist lediglich ein kleiner Ausschnitt der umfassenden Funktionalität von JGit. Wenn Sie Interesse haben und mehr erfahren möchten, finden Sie hier Informationen und Anregungen:

- Die offizielle JGit-API-Dokumentation ist unter <https://www.eclipse.org/jgit/documentation> zu finden: Es handelt sich dabei um Standard-Javadoc, so dass Ihre bevorzugte JVM-IDE in der Lage sein wird, diese auch lokal zu installieren.
- Das JGit Cookbook bei <https://github.com/centic9/jgit-cookbook> enthält viele Beispiele, wie bestimmte Aufgaben mit JGit erledigt werden können.

## go-git

Für den Fall, dass Sie Git in einen in Golang geschriebenen Service integrieren wollen, gibt es auch eine direkte Umsetzung der Go-Bibliothek. Diese Implementierung hat keine eigenen Abhängigkeiten und ist daher nicht anfällig für manuelle Fehler in der Speicherverwaltung. Sie ist auch transparent für die standardmäßigen Golang-Tools zur Leistungsanalyse wie CPU, Memory-Profiler, Race-Detektor usw.

go-git konzentriert sich auf Erweiterbarkeit und Kompatibilität. Es unterstützt die meisten APIs für die Basisbefehle (engl. plumbing), die auf <https://github.com/go-git/go-git/blob/master/COMPATIBILITY.md> dokumentiert sind.

Hier ist ein einfaches Beispiel für die Verwendung der Go-APIs:

```
import "github.com/go-git/go-git/v5"

r, err := git.PlainClone("/tmp/foo", false, &git.CloneOptions{
    URL:      "https://github.com/go-git/go-git",
    Progress: os.Stdout,
})
```

Sobald Sie eine **Repository** Instanz haben, können Sie auf Informationen zugreifen und Änderungen daran vornehmen:

```
// 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)
}
```

## Erweiterte Funktionalität

go-git hat nur wenige nennenswerte fortgeschrittene Funktionen, von denen eine ein erweiterbares Speichersystem ist, das den Libgit2-Backends ähnlich ist. Die Standard-Implementierung ist der In-Memory Storage, welcher sehr effizient ist.

```
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{
    URL: "https://github.com/go-git/go-git",
})
```

Die anpassbare Speicherlösung bietet viele interessante Optionen. So können Sie beispielsweise mit [https://github.com/go-git/go-git/tree/master/\\_examples/storage](https://github.com/go-git/go-git/tree/master/_examples/storage) Referenzen, Objekte und Konfiguration in einer Aerospike-Datenbank speichern.

Ein anderes Feature ist eine flexible Abstraktion des Dateisystems. Mit <https://pkg.go.dev/github.com/go-git/go-billy/v5?tab=doc#Filesystem> ist es einfach, alle Dateien auf unterschiedliche Weise zu speichern, d.h. alle Dateien in ein einziges Archiv auf der Festplatte zu komprimieren

oder sie alle im Arbeitsspeicher zu halten.

Ein weiterer fortgeschritten Verwendungszweck enthält einen fein anpassbaren HTTP-Client, wie er bei [https://github.com/go-git/go-git/blob/master/\\_examples/custom\\_http/main.go](https://github.com/go-git/go-git/blob/master/_examples/custom_http/main.go) zu finden ist.

```
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})
```

## Weiterführende Informationen

Eine vollständige Behandlung der Fähigkeiten von go-git liegt außerhalb des eigentlichen Ziels dieses Buches. Wenn Sie weitere Informationen über go-git wünschen, finden Sie die API-Dokumentation auf <https://pkg.go.dev/github.com/go-git/go-git/v5> und eine Reihe von Anwendungsbeispielen unter [https://github.com/go-git/go-git/tree/master/\\_examples](https://github.com/go-git/go-git/tree/master/_examples).

## Dulwich

Es gibt auch eine pure Python-Implementierung in Git – Dulwich. Das Projekt wird unter <https://www.dulwich.io/> gehostet. Es zielt darauf ab, eine Schnittstelle zu Git-Repositorys (lokal und remote) bereitzustellen, die nicht direkt Git aufruft, sondern stattdessen reines Python verwendet. Es hat dennoch eine optionale C-Erweiterung, die die Leistung erheblich verbessert.

Dulwich folgt dem Git-Design und trennt die beiden grundlegenden API-Bereiche: Basis- und Standardbefehle (engl. plumbing and porcelain).

Das folgende Beispiel zeigt die API der low-level (Basis-)Ebene, um auf die Commit-Beschreibung des letzten Commits zuzugreifen:

```
from dulwich.repo import Repo
r = Repo('.')
r.head()
# '57fbe010446356833a6ad1600059d80b1e731e15'

c = r[r.head()]
c
```

```
# <Commit 015fc1267258458901a94d228e39f0a378370466>
```

```
c.message  
# 'Add note about encoding.\n'
```

Zum Drucken eines Commit-Protokolls mit der high-level Standard-API kann man folgendes verwenden:

```
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
```

## Weiterführende Informationen

API-Dokumentation, Tutorials und viele Beispiele, wie bestimmte Aufgaben mit Dulwich erledigt werden können, finden sind auf der offiziellen Homepage unter <https://www.dulwich.io>.

# Appendix C: Git Kommandos

Im Laufe des Buches haben wir Dutzende von Git-Befehlen vorgestellt und uns bemüht, sie in eine Art Erzählung einzuführen und der Handlung langsam weitere Befehle hinzuzufügen. Das führt jedoch dazu, dass wir Beispiele für die Verwendung der Befehle im ganzen Buch verstreut wiederfinden.

In diesem Anhang werden die im gesamten Buch behandelten Git-Befehle genauer beschreiben, grob gruppiert nach ihren Einsatzgebieten. Wir werden darüber reden, was jeder Befehl ganz allgemein tut, und dann darauf hinweisen, wo wir ihn im Buch benutzt haben.



Sie können lange Optionen abkürzen. Zum Beispiel können Sie `git commit --a` eingeben, was sich so verhält, als ob Sie `git commit --amend` eingegeben hätten. Dies funktioniert nur, wenn die Buchstaben nach `--` für eine Option eindeutig sind. Verwenden Sie beim Schreiben von Skripten die vollständige Option.

## Setup und Konfiguration

Es gibt zwei Befehle, die von den ersten Git-Aufrufen bis hin zum täglichen Optimieren und Referenzieren wirklich oft verwendet werden: die `config` und `help` Befehle.

### git config

Git hat eine Standard-Methode, um Hunderte von Aufgaben zu erledigen. Für viele dieser Aufgaben können Sie Git anweisen, sie auf eine andere Weise auszuführen oder Ihre persönlichen Einstellungen vorzunehmen. Das reicht von der Angabe Ihres Namens bis hin zu bestimmten Terminal-Farbeinstellungen oder dem von Ihnen verwendeten Editor. Bei diesem Befehl werden mehrere Dateien gelesen und beschrieben, so dass Sie Werte global oder bis zu einem bestimmten Repository festlegen können.

Der `git config` Befehl wird in fast jedem Kapitel des Buches benutzt.

In [Git Basis-Konfiguration](#) haben wir ihn verwendet, um unseren Namen, unsere E-Mail-Adresse und unsere Editor-Einstellung zu bestimmen, bevor wir mit Git anfangen konnten.

In [Git Aliases](#) haben wir gezeigt, wie man damit Kurzbefehle erstellen kann, die sich zu langen Optionssequenzen ausbauen, damit man sie nicht jedes Mal eingeben muss.

In [Rebasing](#) haben wir ihn verwendet, um `--rebase` zum Standard der Anwendung zu machen, wenn Sie `git pull` ausführen.

In [Anmeldeinformationen speichern](#) haben wir damit einen Standard-Speicherservice für Ihre HTTP-Passwörter eingerichtet.

In [Schlüsselwort-Erweiterung](#) haben wir gezeigt, wie Sie Smudge- und Clean-Filter für Inhalte einrichten können, die in Git ein- und ausgelesen werden.

Im Prinzip ist der gesamte Abschnitt [Git Konfiguration](#) dem Befehl gewidmet.

## git config core.editor commands

Neben den Konfigurationsanweisungen in [Ihr Editor](#) können viele Editoren wie folgt eingerichtet werden:

Table 4. Exhaustive list of `core.editor` configuration commands

Editor	Configuration command
Atom	<code>git config --global core.editor "atom --wait"</code>
BBEdit (Mac, mit Befehlszeilen-Tools)	<code>git config --global core.editor "bbedit -w"</code>
Emacs	<code>git config --global core.editor emacs</code>
Gedit (Linux)	<code>git config --global core.editor "gedit --wait --new-window"</code>
Gvim (Windows 64-bit)	<code>git config --global core.editor "'C:\Program Files\Vim\vim72\gvim.exe' --nofork '%*'"</code> (Siehe auch Anmerkung unten)
Kate (Linux)	<code>git config --global core.editor "kate"</code>
nano	<code>git config --global core.editor "nano -w"</code>
Notepad (Windows 64-bit)	<code>git config core.editor notepad</code>
Notepad++ (Windows 64-bit)	<code>git config --global core.editor "'C:\Program Files\Notepad\notepad.exe' -multiInst -notabbar -nosession -noPlugin"</code> (Siehe auch Anmerkung unten)
Scratch (Linux)	<code>git config --global core.editor "scratch-text-editor"</code>
Sublime Text (macOS)	<code>git config --global core.editor "/Applications/Sublime Text.app/Contents/SharedSupport/bin/subl --new-window --wait"</code>
Sublime Text (Windows 64-bit)	<code>git config --global core.editor "'C:\Program Files\Sublime Text 3\sublime_text.exe' -w"</code> (Siehe auch Anmerkung unten)
TextEdit (macOS)	<code>git config --global --add core.editor "open --wait-apps --new -e"</code>
Textmate	<code>git config --global core.editor "mate -w"</code>
Textpad (Windows 64-bit)	<code>git config --global core.editor "'C:\Program Files\TextPad 5\TextPad.exe' -m</code> (Siehe auch Anmerkung unten)
UltraEdit (Windows 64-bit)	<code>git config --global core.editor Uedit32</code>
Vim	<code>git config --global core.editor "vim --nofork"</code>
Visual Studio Code	<code>git config --global core.editor "code --wait"</code>
VSCodium (Free/Libre Open Source Software Binaries of VSCode)	<code>git config --global core.editor "codium --wait"</code>
WordPad	<code>git config --global core.editor '"C:\Program Files\Windows NT\Accessories\wordpad.exe"'"</code>
Xi	<code>git config --global core.editor "xi --wait"</code>



Wenn Sie einen 32-Bit-Editor auf einem Windows 64-Bit-System verwenden, wird das Programm in `C:\Program Files (x86)\` und nicht in `C:\Program Files\` wie in der vorstehenden Tabelle installiert.

## git help

Der `git help` Befehl zeigt Ihnen zu einem beliebigen Befehl die gesamte Dokumentation, wie sie mit Git ausgeliefert wird. Während wir in diesem Anhang nur einen groben Überblick über die meisten der gängigsten Befehle geben können, erhalten Sie jederzeit, für jeden Befehl eine komplette Aufstellung aller möglichen Optionen und Flags, wenn Sie `git help <command>` ausführen.

Wir haben den `git help` Befehl in [Hilfe finden](#) vorgestellt und Ihnen gezeigt, wie Sie damit mehr Informationen über `git shell` in [Einrichten des Servers](#) erhalten können.

# Projekte importieren und erstellen

Es gibt zwei Möglichkeiten, ein Git-Repository zu erhalten. Der eine ist, es aus einem bestehenden Repository im Netzwerk oder von irgendwo her zu kopieren und der andere ist, ein eigenes in einem existierenden Verzeichnis zu erstellen.

## git init

Um ein Verzeichnis zu übernehmen und es in ein neues Git-Repository umzuwandeln, so dass Sie die Versionskontrolle starten können, müssen Sie nur `git init` ausführen.

Wir haben das erstmals in [Ein Git-Repository anlegen](#) präsentiert, wo wir zeigen, wie ein neues Repository erstellt wird, mit dem man dann arbeiten kann.

Wir besprechen in [Remote-Banches](#) kurz, wie Sie den Standard-Branch-Namen „master“ ändern können.

Mit diesem Befehl erstellen wir für einen Server ein leeres Bare-Repository in [Das Bare-Repository auf einem Server ablegen](#).

Zum Schluss werden wir in [Basisbefehle und Standardbefehle \(Plumbing and Porcelain\)](#) einige Details der Funktionsweise im Hintergrund erläutern.

## git clone

Der `git clone` Befehl ist in Wahrheit ein Art Wrapper für mehrere andere Befehle. Er erstellt ein neues Verzeichnis, wechselt dort hin und führt `git init` aus. So wird es zu einem leeren Git-Repository umgewandelt. Dann fügt er zu der übergebenen URL einen Remote (`git remote add`) hinzu (standardmäßig mit dem Namen `origin`). Er ruft ein `git fetch` von diesem Remote-Repository auf und holt mit `git checkout` den letzten Commit in Ihr Arbeitsverzeichnis.

Der `git clone` Befehl wird an Dutzenden von Stellen im ganzen Buch verwendet, wir werden aber nur ein paar interessante Stellen auflisten.

Er wird im Wesentlichen in [Ein existierendes Repository klonen](#) eingeführt und beschrieben, wobei

wir einige Beispiele durchgehen.

In [Git auf einem Server einrichten](#) untersuchen wir die Verwendung der Option `--bare`, um eine Kopie eines Git-Repository ohne Arbeitsverzeichnis zu erstellen.

In [Bundling](#) verwenden wir ihn, um ein gebündeltes Git-Repository zu entpacken.

Schließlich lernen wir in [Ein Projekt mit Submodulen klonen](#) die Option `--recurse-submodules` kennen, die das Klonen eines Repositorys mit Submodulen etwas einfacher macht.

Obwohl der Befehl noch an vielen anderen Stellen im Buch verwendet wird, sind das jene, die etwas eigenständig sind oder bei denen er auf eine etwas andere Weise verwendet wird.

## Einfache Snapshot-Funktionen

Für den grundlegenden Workflow der Erstellung von Inhalten und dem Committen in Ihren Verlauf gibt es nur wenige einfache Befehle.

### git add

Der `git add` Befehl fügt, für den nächsten Commit, Inhalte aus dem Arbeitsverzeichnis der Staging-Area (bzw. „Index“) hinzu. Bei der Ausführung des Befehls `git commit` wird standardmäßig nur diese Staging-Area betrachtet, so dass mit `git add` festgelegt wird, wie Ihr nächster Commit-Schnappschuss aussehen soll.

Dieser Befehl ist ein unglaublich wichtiges Kommando in Git und wird in diesem Buch mehrfach erwähnt oder verwendet. Wir werden kurz auf einige der einzigartigen Verwendungen eingehen, die es gibt.

Wir stellen `git add` zunächst in [Neue Dateien zur Versionsverwaltung hinzufügen](#) vor und beschreiben ihn ausführlich.

Wir besprechen in [Einfache Merge-Konflikte](#), wie man damit Konflikte beim Mergen löst.

Wir fahren in [Interaktives Stagen](#) damit fort, bestimmte Teile einer modifizierten Datei interaktiv zur Staging-Area hinzuzufügen.

Schließlich emulieren wir ihn in [Baum Objekte](#) auf einem unteren Level, so dass Sie sich vorstellen können, was er im Hintergrund bewirkt.

### git status

Der `git status` Befehl wird Ihnen die verschiedenen Dateizustände in Ihrem Arbeitsverzeichnis und der Staging-Area anzeigen. Er zeigt welche Dateien modifiziert und nicht bereitgestellt und welche bereitgestellt (eng. staged), aber noch nicht committet sind. In seiner üblichen Form werden Ihnen auch einige grundlegende Tipps gegeben, wie Sie Dateien zwischen diesen Stufen verschieben können.

Wir behandeln `status` zunächst in [Zustand von Dateien prüfen](#), sowohl in seinen grundlegenden als auch in seinen kompakten Formen. Im Buch wird so ziemlich alles angesprochen, was man mit

dem Befehl `git status` machen kann.

## git diff

Der `git diff` Befehl wird verwendet, wenn Sie Unterschiede zwischen zwei beliebigen Bäumen feststellen möchten. Das könnte der Unterschied zwischen Ihrer Arbeitsumgebung und Ihrer Staging-Area (`git diff` an sich), zwischen Ihrer Staging-Area und Ihrem letzten Commit (`git diff --staged`) oder zwischen zwei Commits (`git diff master branchB`) sein.

In [Überprüfen der Staged- und Unstaged-Änderungen](#) betrachten wir zunächst die grundsätzliche Anwendung von `git diff` und zeigen dort, wie man feststellen kann, welche Änderungen bereits der Staging-Area hinzugefügt wurden und welche noch nicht.

Wir verwenden ihn mit der Option `--check` in [Richtlinien zur Zusammenführung \(engl. Commits\)](#), um nach möglichen Leerzeichen-Problemen zu suchen, bevor wir committen.

Wir sehen, wie man die Unterschiede zwischen Branches mit der Syntax `git diff A…B` in [Bestimmen, was übernommen wird](#) effektiver überprüfen kann.

Wir verwenden ihn, um Leerzeichen-Differenzen mit `-b` herauszufiltern und wie man verschiedene Stufen von Konfliktdateien mit `--theirs`, `--ours` und `--base` in [Fortgeschrittenes Merging](#) vergleicht.

Zuletzt verwenden wir ihn, um Submodul-Änderungen effektiv mit `--submodule` in [Erste Schritte mit Submodulen](#) zu vergleichen.

## git difftool

Der Befehl `git difftool` startet ein externes Tool, um Ihnen den Unterschied zwischen zwei Bäumen zu zeigen, falls Sie einen anderen Befehl als das eingebaute `git diff` bevorzugen.

Das erwähnen wir nur kurz in [Überprüfen der Staged- und Unstaged-Änderungen](#).

## git commit

Der `git commit` Befehl erfasst alle Dateiinhalte, die mit `git add` zur Staging-Area hinzugefügt wurden und speichert einen neuen permanenten Schnapschuss in der Datenbank. Anschließend bewegt er den Branch-Pointer der aktuellen Branch zu diesem hinauf.

Wir erklären zunächst die Grundlagen des Commitings in [Die Änderungen committen](#). Dort zeigen wir auch, wie man mit dem `-a` Flag den Schritt `git add` im täglichen Arbeitsablauf überspringt und wie man mit dem `-m` Flag eine Commit-Meldung in der Kommandozeile übergibt, anstatt einen Editor zu starten.

In [Ungewollte Änderungen rückgängig machen](#) befassen wir uns mit der Verwendung der Option `--amend`, um den letzten Commit wieder herzustellen.

In [Branches auf einen Blick](#) gehen wir sehr viel detaillierter darauf ein, was `git commit` bewirkt und warum es das so macht.

In [Commits signieren](#) haben wir uns angesehen, wie man Commits kryptographisch mit dem `-S`

Flag signiert.

Schließlich werfen wir einen Blick darauf, was der Befehl `git commit` im Hintergrund macht und wie er tatsächlich eingebunden ist in [Commit Objekte](#).

## git reset

Der `git reset` Befehl wird in erster Linie verwendet, um Aktionen rückgängig zu machen, wie man am Kommando erkennen kann. Er verschiebt den `HEAD` Pointer und ändert optional den `index` oder die Staging-Area und kann optional auch das Arbeitsverzeichnis ändern, wenn Sie `--hard` verwenden. Bei falscher Verwendung der letzten Option kann mit diesem Befehl auch Arbeit verloren gehen, vergewissern Sie sich daher, dass Sie ihn verstehen, bevor Sie ihn verwenden.

Wir befassen uns zunächst mit der einfachsten Anwendung von `git reset` in [Eine Datei aus der Staging-Area entfernen](#). Dort benutzen wir es, um eine Datei, die wir mit `git add` hinzugefügt haben, wieder aus der Staging-Area zu entfernen.

Wir gehen dann in [Reset entzaubert](#) detailliert auf diesen Befehl ein, der sich ganz der Beschreibung dieses Befehls widmet.

Wir verwenden `git reset --hard`, um einen Merge in [Einen Merge abbrechen](#) abzubrechen, wo wir auch `git merge --abort` verwenden, das eine Art Wrapper für den `git reset` Befehl ist.

## git rm

Der `git rm` Befehl wird verwendet, um Dateien aus dem Staging-Bereich und dem Arbeitsverzeichnis von Git zu entfernen. Er ähnelt `git add` dahingehend, dass er das Entfernen einer Datei für den nächsten Commit vorbereitet.

Wir behandeln den Befehl `git rm` in [Dateien löschen](#) ausführlich, einschließlich des rekursiven Entfernens von Dateien und des Entfernens von Dateien aus der Staging-Area, wobei sie jedoch, mit `--cached`, im Arbeitsverzeichnis belassen werden.

Die einzige andere abweichende Verwendung von `git rm` im Buch ist in [Objekte löschen](#) beschrieben, wo wir kurz die `--ignore-unmatch` beim Ausführen von `git filter-branch` verwenden und erklären, was es einfach nicht fehlerfrei macht, wenn die Datei, die wir zu entfernen versuchen, nicht existiert. Das kann bei der Erstellung von Skripten nützlich sein.

## git mv

Der `git mv` Befehl ist ein schlanker komfortabler Befehl, um eine Datei zu verschieben und dann `git add` für die neue Datei und `git rm` für die alte Datei auszuführen.

Wir beschreiben diesen Befehl nur kurz in [Dateien verschieben](#).

## git clean

Der Befehl `git clean` wird verwendet, um unerwünschte Dateien aus Ihrem Arbeitsverzeichnis zu entfernen. Dazu kann das Entfernen von temporären Build-Artefakten oder das Mergen von Konfliktdateien gehören.

Wir behandeln viele der Optionen und Szenarien, in denen Sie den clean-Befehl verwenden könnten in [Bereinigung des Arbeitsverzeichnisses](#).

## Branching und Merging

Es gibt nur eine Handvoll Befehle, die die meisten Branching- und Merging-Funktionen in Git bereitstellen.

### git branch

Der `git branch` Befehl ist eigentlich so etwas wie ein Branch-Management-Tool. Er kann die von Ihnen vorhandenen Branches auflisten, einen neuen Branch erstellen, Branches löschen und umbenennen.

Der größte Teil von [Git Branching](#) ist dem Befehl `branch` gewidmet und wird im gesamten Kapitel verwendet. Wir stellen ihn zuerst in [Erzeugen eines neuen Branches](#) vor und betrachten die meisten seiner anderen Funktionen (das Auflisten und Löschen) in [Branch-Management](#).

In [Tracking-Banches](#) verwenden wir die Option `git branch -u`, um einen Tracking-Branch einzurichten.

Schließlich werden wir einige der Funktionen, die im Hintergrund ausgeführt werden, in [Git Referenzen](#) durchgehen.

### git checkout

Der `git checkout` Befehl wird benutzt, um Branches zu wechseln und Inhalte in Ihr Arbeitsverzeichnis auszuchecken.

Wir sind in [Wechseln des Branches](#) zum ersten Mal dem `git branch` Befehl begegnet.

Wir zeigen in [Tracking-Banches](#), wie man das Tracking von Branches mit dem `--track` Flag startet.

Wir verwenden ihn in [Die Konflikte austesten](#), um Dateikonflikte mit `--conflict=diff3` wieder zu integrieren.

Wir gehen auf die Beziehung zu `git reset` in [Reset entzaubert](#) näher ein.

Abschließend gehen wir auf einige Details der Umsetzung in [HEAD](#) ein.

### git merge

Das `git merge` Tool wird benutzt, um einen oder mehrere Branches in den in den ausgecheckten Branch zusammenzuführen. Es wird dann der aktuelle Branch zum Ergebnis des Merge-Vorgangs weitergeführt.

Der Befehl `git merge` wurde zunächst in [Einfaches Branching](#) vorgestellt. Obwohl er an verschiedenen Stellen im Buch verwendet wird, gibt es nur sehr wenige Variationen des Befehls `merge`. In der Regel nur `git merge <branch>` mit dem Namen des einzelnen Branches, in dem Sie zusammenführen möchten.

Wir haben am Ende von [Verteiltes, öffentliches Projekt](#) beschrieben, wie man ein Squashed Merge macht (bei dem Git die Arbeit zusammenführt, sich aber so verhält, als wäre es nur ein neuer Commit, ohne die Historie des Branches, in dem man zusammenführt, aufzuzeichnen).

Wir haben in [Fortgeschrittenes Merging](#) viel über den Merge-Prozess und -Befehl berichtet, einschließlich des Befehls `-Xignore-space-change` und des Flags `--abort`, um ein Merge-Problem abzubrechen.

Wir haben in [Commits signieren](#) gelernt, wie man Signaturen vor dem Zusammenführen überprüft, wenn Ihr Projekt GPG-Signaturen verwendet.

Schließlich haben wir in [Subtree Merging](#) das Mergen von Sub-Trees kennengelernt.

## git mergetool

Der `git mergetool` Befehl startet lediglich einen externen Merge-Helfer, falls Sie Probleme mit einer Zusammenführung in Git haben.

Wir erwähnen ihn kurz in [Einfache Merge-Konflikte](#) und gehen ausführlich in [Externe Merge- und Diff-Tools](#) darauf ein, wie Sie Ihr eigenes externes Merge-Tool integrieren können.

## git log

Der `git log` Befehl wird verwendet, um den verfügbaren, aufgezeichneten Verlauf eines Projekts, ab des letzten Commit-Snapshots, rückwärts anzuzeigen. Standardmäßig wird nur die Historie des Branchs angezeigt, in dem Sie sich gerade befinden, kann aber mit verschiedenen oder sogar mehreren Heads oder Branches belegt werden, mit denen Sie Schnittmengen haben können. Er wird häufig verwendet, um Unterschiede zwischen zwei oder mehr Branches auf der Commit-Ebene anzuzeigen.

Dieses Kommando wird in fast jedem Kapitel des Buches verwendet, um die Verlaufshistorie eines Projekts zu demonstrieren.

Wir stellen den Befehl in [Anzeigen der Commit-Historie](#) vor und gehen dort etwas ausführlicher darauf ein. Wir betrachten die Option `-p` und `--stat`, um eine Übersicht darüber zu erhalten, was in jedem Commit enthalten ist, und die Optionen `--pretty` und `--oneline`, um die Historie, zusammen mit einigen einfachen Datums- und Autoren-Filteroptionen, übersichtlicher wiederzugeben.

In [Erzeugen eines neuen Branches](#) verwenden wir ihn mit der Option `--decorate`, um leichter zu verdeutlichen, wo unser Branch-Pointer sich gerade befindet und wir benutzen auch die `--graph` Option, um zu sehen, wie die unterschiedlichen Verläufe aussehen.

In [Kleines, privates Team](#) und [Commit-Bereiche](#) behandeln wir die Syntax `branchA..branchB`, um mit dem `git log` Befehl zu überprüfen, welche Commits, relativ zu einem anderen Branch, eindeutig sind. In [Commit-Bereiche](#) gehen wir ausführlicher darauf ein.

In [Merge-Protokoll](#) und [Dreifacher Punkt](#) wird das Format `branchA...branchB` und die Syntax `--left-right` verwendet, um zu sehen, was in dem einen oder anderen Branch vorhanden ist, aber nicht in beiden. In [Merge-Protokoll](#) untersuchen wir auch, wie Sie die Option `--merge` verwenden können, um beim Debugging von Merge-Konflikten zu helfen, sowie die Option `--cc`, um Merge-Commit-

Konflikte in Ihrem Verlauf zu betrachten.

In [RefLog Kurzformen](#) benutzen wir die Option `-g`, um den Git-RefLog über dieses Tool anzuzeigen, anstatt eine Branch-Überquerung durchzuführen.

In [Suchen](#) schauen wir uns die Verwendung der `-S` und `-L` Optionen an, um eine relativ komplexe Suche nach etwas durchzuführen, was während der Entwicklung des Codes passiert ist, wie z.B. den Fortschritt in einer Funktion wahrzunehmen.

In [Commits signieren](#) sehen wir, wie man mit Hilfe der Option `--show-signature` jedem Commit in der `git log` Ausgabe eine Validierungs-Zeichenkette hinzufügt, abhängig von der Gültigkeit der Signatur.

## git stash

Der Befehl `git stash` wird verwendet, um nicht fertiggestellte Arbeit vorübergehend zu speichern, um Ihr Arbeitsverzeichnis aufzuräumen, ohne unfertige Arbeit auf einem Branch committen zu müssen.

Im Wesentlichen wird dieses Thema in [Stashen und Bereinigen](#) vollständig behandelt.

## git tag

Der Befehl `git tag` wird verwendet, um ein permanentes Lesezeichen an einen bestimmten Punkt in der Code-Historie zu setzen. Im Allgemeinen wird das für die Erstellung von Releases verwendet.

Dieser Befehl wird in [Taggen](#) eingeführt und ausführlich behandelt; wir benutzen ihn in der Praxis in [Tagging deines Releases](#).

Wir behandeln in [Deine Arbeit signieren](#) auch, wie man einen GPG-signierten Tag mit dem `-s` Flag erstellt und einen mit dem `-v` Flag verifiziert.

# Projekte gemeinsam nutzen und aktualisieren

Es gibt nicht besonders viele Befehle in Git, die auf das Netzwerk zugreifen, fast alle Befehle arbeiten mit der lokalen Datenbank. Wenn Sie Ihre Arbeit freigeben oder Änderungen von anderswo beziehen wollen, gibt es eine kleine Anzahl von Befehlen, die sich mit Remote-Repositorys beschäftigen.

## git fetch

Der Befehl `git fetch` kommuniziert mit einem entfernten Repository und holt alle Informationen, die sich in diesem Repository befinden, aber nicht in Ihrem aktuellen Repository und speichert sie in Ihrer lokalen Datenbank.

Wir sehen uns diesen Befehl zunächst in [Fetchen und Pullen deiner Remotes](#) an und betrachten anschließend weitere Beispiele für seine Verwendung in [Remote-Branches](#).

Wir benutzen ihn auch bei einigen Beispielen in [An einem Projekt mitwirken](#).

Wir verwenden ihn in [Pull Request Refs \(Referenzen\)](#), um eine einzelne konkrete Referenz zu beziehen, die außerhalb des standardmäßigen Bereichs liegt und wir sehen, in [Bundling](#), wie man sie aus einem Packet herausholen kann.

Wir richten in [Die Referenzspezifikation \(engl. Refspec\)](#) eigene Referenzspezifikationen ein, damit `git fetch` etwas anderes als die Standardeinstellung macht.

## git pull

Der `git pull` Befehl ist im Grunde eine Kombination aus den `git fetch` und `git merge` Befehlen, wobei Git von dem angegebenen Remote holt und dann sofort versucht, es in den Branch, auf dem Sie gerade sind, zu integrieren.

Wir führen ihn in [Fetchen und Pullen deiner Remotes](#) ein und zeigen in [Inspizieren eines Remotes](#) auf, was alles gemerged wird, wenn Sie ihn benutzen.

Wir erfahren in [Rebasen, wenn du Rebase durchführst](#) auch, wie Sie damit bei Schwierigkeiten während des Rebasings umgehen können.

Wir zeigen in [Remote Branches auschecken](#), wie man ihn mit einer URL verwendet, um Änderungen einmalig einzupflegen.

Schließlich erwähnen wir in [Commits signieren](#) noch kurz, wie Sie die Option `--verify-signatures` verwenden können, um beim Abrufen/Pullen von Commits überprüfen können, ob diese mit GPG signiert wurden.

## git push

Der `git push` Befehl wird benutzt, um mit einem anderen Repository zu kommunizieren, zu ermitteln, was die lokale Datenbank enthält, die die entfernte nicht hat und dann die Differenz in das entfernte Repository zu pushen. Es erfordert Schreibzugriff auf das entfernte Repository und wird daher in der Regel auf irgend eine Weise authentifiziert.

Wir sehen uns zuerst den `git push` Befehl in [Zu deinen Remotes Pushen](#) an. Hier beschreiben wir die grundlegenden Aspekte des Pushens einer Branch zu einem Remote-Repository. In [Pushing/Hochladen](#) gehen wir ein wenig detaillierter auf das Pushen bestimmter Branches ein und in [Tracking-Banches](#) sehen wir, wie man Tracking-Banches einrichtet, um dorthin automatisch zu pushen. In [Remote-Banches entfernen](#) benutzen wir die Option `--delete`, um einen Branch auf dem Server mit `git push` zu löschen.

Im Kapitel 5 [An einem Projekt mitwirken](#) können Sie einige Beispiele für die Verwendung von `git push` finden, wie Sie Ihre Arbeit an Branches mit mehreren Remotes teilen können.

Wir sehen in [Tags teilen](#), wie Sie diesen Befehl benutzen können, um Tags, die Sie mit der `--tags` Option erstellt haben, gemeinsam zu nutzen.

In [Änderungen am Submodul veröffentlichen](#) verwenden wir die Option `--recurse-submodules`, um zu überprüfen, ob alle unsere Submodule funktionieren, bevor wir zum Hauptprojekt pushen, was bei der Verwendung von Submodulen sehr hilfreich sein kann.

In [Andere Client-Hooks](#) sprechen wir kurz über den `pre-push` Hook, ein Skript, das wir so einrichten können, dass es vor dem Abschluss eines Pushs ausgeführt wird, um zu prüfen, ob es zulässig sein sollte, zu pushen.

Schließlich betrachten wir in [Pushende Refspecs](#) das Pushen mit einer vollständigen Referenzspezifikation (engl. refspec) anstelle der allgemeinen Abkürzungen, die normalerweise verwendet werden. Das kann Ihnen helfen, sehr spezifisch zu entscheiden, welche Arbeit Sie teilen möchten.

## git remote

Der Befehl `git remote` ist ein Management-Tool für die Verwaltung Ihrer Datensätze in Remote-Repositorys. Er erlaubt Ihnen, lange URLs als kurze Handles zu speichern, wie z.B. „origin“, damit Sie diese nicht ständig abtippen müssen. Sie können auch mehrere solcher Adressen einrichten und der Befehl `git remote` wird verwendet, um sie hinzuzufügen, zu ändern oder zu löschen.

Dieser Befehl wird ausführlich in [Mit Remotes arbeiten](#) behandelt, einschließlich des Auflistens, Hinzufügens, Entfernens und Umbenennens.

Er wird auch in fast jedem der nachfolgenden Kapitel des Buchs verwendet, aber immer im Standardformat `git remote add <name> <url>`.

## git archive

Der Befehl `git archive` wird verwendet, um eine Archivdatei von einem bestimmten Snapshot des Projekts zu erstellen.

Wir benutzen `git archive` in [Ein Release vorbereiten](#), um einen Tarball eines Projektes für die gemeinsame Nutzung zu erstellen.

## git submodule

Der Befehl `git submodule` dient dazu, externe Repositorys innerhalb eines normalen Repositorys zu verwalten. Das kann für Bibliotheken oder andere Arten von gemeinsam genutzten Ressourcen nötig sein. Das `submodule` Kommando hat mehrere Unterbefehle (`add`, `update`, `sync`, usw.) für die Verwaltung dieser Ressourcen.

Dieser Befehl wird nur in [Submodule](#) erwähnt und dort ausführlich beschrieben.

# Kontrollieren und Vergleichen

## git show

Der Befehl `git show` kann ein Git-Objekt auf eine einfache und für den Benutzer lesbare Weise darstellen. Normalerweise würden Sie diesen Befehl verwenden, um die Informationen über ein Tag oder einen Commit anzuzeigen.

Wir verwenden ihn erstmals in [Annotated Tags](#), um annotierte Tag-Informationen anzuzeigen.

Danach, in [Revisions-Auswahl](#), verwenden wir ihn mehrfach, um die Commits zu dokumentieren, die unsere verschiedenen Revisionsauswahlen betreffen.

Eines der interessanteren Dinge, die wir in [Manuelles Re-Mergen von Dateien](#) mit `git show` machen, ist das Extrahieren bestimmter Dateiinhalte in verschiedenen Abschnitten bei einem Merge-Konflikt.

## git shortlog

Der Befehl `git shortlog` wird verwendet, um die Ausgabe von `git log` zu verdichten. Dieses Kommando kennt ähnliche Optionen wie `git log`, aber anstatt alle Commits aufzulisten, wird eine Zusammenfassung der Commits, gruppiert nach Autor angezeigt.

Wir haben in [Das Shortlog](#) gezeigt, wie man damit ein schönes Changelog erstellt.

## git describe

Der Befehl `git describe` wird verwendet, um alles zu übernehmen, das zu einem Commit führt und er erzeugt eine Zeichenkette, die einigermaßen menschenlesbar ist und sich nicht ändern wird. Es ist eine Möglichkeit, eine Beschreibung eines Commits zu erhalten, die so eindeutig wie ein Commit SHA-1 ist, dafür aber auch verständlicher.

Wir verwenden `git describe` in [Eine Build Nummer generieren](#) und [Ein Release vorbereiten](#), um einen String zu erzeugen, der unsere Release-Datei benennt.

# Debugging

Git hat ein paar Befehle, die Ihnen helfen, ein Problem im Code zu debuggen. Das geht vom Feststellen, wann etwas eingefügt wurde, bis zum Erkennen, wer es eingereicht hat.

## git bisect

Das Tool `git bisect` ist ein unglaublich hilfreiches Debugging-Tool, das eingesetzt wird, um mit Hilfe einer automatischen Binärsuche herauszufinden, welcher bestimmte Commit als erster einen Fehler oder ein Problem verursacht hat.

Es wird in [Binärsuche](#) vollständig dokumentiert und nur in diesem Abschnitt behandelt.

## git blame

Der Befehl `git blame` kommentiert die Zeilen einer Datei, bei denen ein Commit zuletzt eine Änderung vorgenommen hat. Zudem wird vermerkt, wer der Autor des Commits ist. Das hilft Ihnen, denjenigen zu ermitteln, der weitere Angaben zu einem bestimmten Abschnitt Ihres Codes machen kann.

Er wird in [Datei-Annotationen](#) behandelt und nur in diesem Kapitel beschrieben.

## git grep

Der Befehl `git grep` kann Ihnen bei der Suche nach einer beliebigen Zeichenfolge oder einem regulären Ausdruck in irgendeiner der Dateien Ihres Quellcodes behilflich sein, selbst in älteren Fassungen Ihres Projekts.

Er wird in [Git Grep](#) behandelt und nur dort beschrieben.

## Patchen bzw. Fehlerkorrektur

Ein paar Befehle in Git fokussieren sich um die konzeptionelle Überlegung, wie Commits sich in Bezug auf die Änderungen verhalten, die sie einführen, wenn die Commit-Serie eine Reihe von Patches wäre. Diese Befehle helfen Ihnen, Ihre Branches auf dieser Grundlage zu organisieren.

## git cherry-pick

Die `git cherry-pick` Anweisung wird benutzt, um die Änderung, die in einem einzelnen Git-Commit vorgenommen wurde, als neuen Commit auf dem Branch, auf dem Sie sich gerade befinden, erneut vorzunehmen. Das kann sinnvoll sein, um nur ein oder zwei Commits aus einem Branch individuell zu übernehmen, anstatt sie in den Branch einzubringen, der sämtliche geänderten Daten enthält.

Das „Kirschenpflücken“ (engl. cherry picking) wird in [Rebasing und Cherry-Picking Workflows](#) beschrieben und demonstriert.

## git rebase

Der Befehl `git rebase` ist im Grunde genommen ein automatisches `cherry-pick`. Er ermittelt eine Reihe von Commits und nimmt sie dann nacheinander, in der gleichen Reihenfolge, an anderer Stelle wieder auf.

Rebasing wird ausführlich in [Rebasing](#) behandelt, einschließlich der Problematik bei der Zusammenarbeit im Zusammenhang mit Rebasing von bereits veröffentlichten Branches.

Wir verwenden ihn bei einem praktischen Beispiel in [Replace \(Ersetzen\)](#) für die Aufteilung Ihres Verlaufs in zwei getrennte Repositorys, wobei auch das Flag `--onto` benutzt wird.

In [Rerere](#) kommt es bei einem Rebaset zu einem Merge-Konflikt.

In [Ändern mehrerer Commit-Beschreibungen](#) verwenden wir ihn auch in einem interaktiven Skripting-Modus mit der Option `-i`.

## git revert

Der `git revert` Befehl ist im Prinzip ein umgekehrter `git cherry-pick`. Er erzeugt einen neuen Commit, der das genaue Gegenteil der Änderung bewirkt, die in dem Commit, auf den Sie gerade zugreifen, eingeführt wurde, d.h. er macht ihn rückgängig.

In [Den Commit umkehren](#) verwenden wir diesen Befehl, um einen Merge-Commit rückgängig zu machen.

# E-mails

Viele Git-Projekte, einschließlich Git selbst, werden vollständig über Mailinglisten verwaltet. Git hat eine Reihe von integrierten Tools, die diesen Prozess erleichtern. Angefangen bei der Erstellung von Patches, die Sie einfach per E-Mail versenden können, bis hin zur Anwendung dieser Patches aus einem E-Mail-Postfach heraus.

## git apply

Der Befehl `git apply` wendet einen Patch an, der mit dem Befehl `git diff` oder auch mit GNU diff erstellt wurde. Das ist vergleichbar mit dem, was der Befehl `patch` macht, mit ein paar kleinen Unterschieden.

In [Integrieren von Änderungen aus E-Mails](#) zeigen wir Ihnen die Handhabung und die Bedingungen, unter denen Sie das tun sollten.

## git am

Der Befehl `git am` wird für die Übernahme von Patches aus einem Email-Postfach verwendet, konkret aus einem mbox-formatierten Email-Postfach. Dadurch können Sie Patches per E-Mail erhalten und sie einfach in Ihrem Projekt einsetzen.

In [Änderungen mit am integrieren](#) haben wir die Bedienung und den Umgang mit `git am` behandelt, einschließlich der Optionen `--resolved`, `-i` und `-3`.

Es gibt auch eine Reihe von Hooks, die Sie zur Vereinfachung des Workflows rund um `git am` verwenden können, die alle in [E-Mail-Workflow-Hooks](#) behandelt werden.

Wir verwenden ihn in [E-Mail Benachrichtigungen](#) ebenfalls, um patch-formatierte Anpassungen in GitHub Pull-Request anzuwenden.

## git format-patch

Der Befehl `git format-patch` wird verwendet, um eine Reihe von Patches im mbox-Format zu erzeugen, die Sie an eine Mailingliste, korrekt formatiert, senden können.

Wir zeigen anhand eines Beispiels in [Öffentliche Projekte via Email](#) wie Sie mit dem Tool `git format-patch` zu einem Projekt beitragen können .

## git imap-send

Der Befehl `git imap-send` lädt eine mit `git format-patch` erzeugte Mailbox in einen IMAP-Entwurfsordner hoch.

Wir betrachten in [Öffentliche Projekte via Email](#) ein Beispiel, wie Sie durch Senden von Patches mit dem Tool `git imap-send` zu einem Projekt beitragen können.

## git send-email

Mit dem Befehl `git send-email` werden Korrekturen, die mit `git format-patch` erzeugt wurden, über E-Mail verschickt.

Wir sehen in [Öffentliche Projekte via Email](#) ein Beispiel für einen Projektbeitrag durch das Versenden von Patches mit dem Tool `git send-email`.

## git request-pull

Der Befehl `git request-pull` wird lediglich dazu verwendet, einen exemplarischen Nachrichtentext zu generieren, der an eine Person per E-Mail gesendet werden kann. Wenn Sie einen Branch auf einem öffentlichen Server haben und jemanden wissen lassen wollen, wie man diese Änderungen integriert, ohne dass die Patches per E-Mail verschickt werden, können Sie diesen Befehl ausführen und die Ausgabe an die Person senden, die die Änderungen einspielen soll.

Wir zeigen in [Verteiltes, öffentliches Projekt](#), wie man `git request-pull` verwendet, um eine Pull-Nachricht zu erzeugen.

# Externe Systeme

Git enthält einige Kommandos mit denen eine Integration mit anderen Versionskontrollsystmen möglich ist.

## git svn

Mit Hilfe der Funktion `git svn` kann man als Client mit dem Versionskontrollsystem Subversion kommunizieren. Das bedeutet, dass Sie Git zum Auschecken von und zum Committen an einen Subversion-Server verwenden können.

Dieser Befehl wird in [Git und Subversion](#) ausführlich erläutert.

## git fast-import

Für andere Versionskontrollsystme oder den Import aus beinahe jedem Format können Sie `git fast-import` verwenden. So können Sie das andere Format einfach auf etwas umwandeln, das Git problemlos verarbeiten kann.

In [Benutzerdefinierter Import](#) wird diese Funktion eingehend untersucht.

# Administration

Wenn Sie ein Git-Repository verwalten oder etwas in größerem Umfang reparieren müssen, bietet Git eine Reihe von Verwaltungsbefehlen, die Sie dabei unterstützen.

## git gc

Der Befehl `git gc` führt „garbage collection“ (dt. Speicherbereinigung) auf Ihrem Repository aus. Er entfernt unnötige Dateien aus Ihrer Datenbank und packt die verbleibenden Dateien in ein

effizientes Format.

Dieser Befehl läuft normalerweise im Hintergrund ab. Wenn Sie wollen, können Sie ihn aber auch manuell ausführen. Wir werden einige Beispiele dafür in [Wartung](#) näher betrachten.

## git fsck

Der Befehl `git fsck` wird zur Überprüfung der internen Datenbank auf Probleme oder Inkonsistenzen verwendet.

Wir beschreiben ihn nur kurz in [Datenwiederherstellung](#), um nach verwaisten Objekten zu suchen.

## git reflog

Der Befehl `git reflog` untersucht ein Log-Protokoll, in dem alle Heads Ihrer Branches aufgezeichnet sind, während Sie daran gearbeitet haben. So können Sie Commits finden, die Sie durch das Umschreiben der Historie verloren haben könnten.

Wir beschäftigen uns mit diesem Befehl hauptsächlich in [RefLog Kurzformen](#). Dort zeigen wir die normale Benutzung und die Verwendung von `git log -g`, um die gleichen Informationen so zu formatieren damit sie wie mit der `git log` Ausgabe aussehen.

Wir stellen in [Datenwiederherstellung](#) ein praktisches Beispiel für die Wiederherstellung einer derart verloren gegangener Branch vor.

## git filter-branch

Der Befehl `git filter-branch` dient dazu, eine Vielzahl von Commits nach bestimmten Kriterien umzuschreiben. Sie können beispielsweise eine Datei überall entfernen oder das gesamte Repository in ein einziges Unterverzeichnis filtern, zum Extrahieren eines Projekts.

In [Eine Datei aus jedem Commit entfernen](#) erklären wir den Befehl und untersuchen verschiedene Optionen wie `--commit-filter`, `--subdirectory-filter` und `--tree-filter`.

In [Git-p4](#) verwenden wir ihn, um importierte externe Repositorys zu berichtigen.

# Basisbefehle

Es gibt zudem eine ganze Reihe von Basisbefehlen, auf die wir in diesem Buch gestoßen sind.

Zuerst begegnen wir `ls-remote` in [Pull Request Refs \(Referenzen\)](#), das wir zum Betrachten der Rohdaten auf dem Server verwenden.

Wir verwenden `ls-files` in [Manuelles Re-Mergen von Dateien](#), [Rerere](#) und [Der Index](#), um einen groben Einblick in Ihre Staging-Area zu erhalten.

Wir beziehen uns in [Branch Referenzen](#) auch auf `rev-parse`, um so gut wie jede beliebige Zeichenkette zu verwenden und sie in ein SHA-1 Objekt zu konvertieren.

Die meisten der von uns beschriebenen Low-Level Basisbefehle sind in [Git Interna](#) enthalten,

worauf sich das Kapitel mehr oder weniger konzentriert. Wir haben versucht, sie im restlichen Teil des Buches nicht zu verwenden.

# Index

## @

\$EDITOR, 361  
\$VISUAL  
  siehe \$EDITOR, 361  
.gitignore, 363  
.NET, 504  
@{upstream}, 101  
@{u}, 101

## A

Aliasnamen, 66  
Anmeldeinformationen, 354  
Apache, 127  
Apple, 504  
Arbeitsablauf siehe Workflows, 136  
Archivierung, 377  
Attribute, 371  
Auto-Korrektur, 363

## B

Bash, 493  
Beitragen, 140  
  zu gemanagtem privatem Team, 150  
  zu kleinem privatem Team, 142  
  zu öffentlichem großen Projekt, 160  
  zu öffentlichem kleinen Projekt, 156  
Beiträge integrieren, 170  
Binärdateien, 371  
BitKeeper, 15  
Branches, 68  
  anlegen, 70  
  diffing, 169  
  einfacher Workflow, 76  
  langlebige, 89  
  mergen, 81  
  Remote, 92  
  remote, 168  
  Remote entfernen, 102  
Themen-Branches, 164  
Topic, 90, 164  
Tracking, 100  
Upstream, 100  
verwalten, 85  
wechseln, 72  
Build Nummern, 177

## C

C, 499  
C#, 504  
Cocoa, 504  
color, 364  
Commit-Vorlagen, 361  
credentials, 354  
crlf, 368  
CVS, 12

## D

Datei-Ausschlüsse, 363, 444  
Dateien  
  entfernen, 41  
  ignorieren, 35  
  verschieben, 43  
difftool, 365  
distributed git, 136  
Dulwich, 510

## E

E-Mail, 162  
  Patches anwenden, 164  
Editor  
  ändern der Voreinstellung, 40  
excludes, 363, 444

## F

Farbe, 364  
Forken, 138, 186  
Freigeben, 178

## G

Git als Client, 393  
Git Befehle  
  add, 32, 32, 33  
  am, 165  
  apply, 165  
  archive, 178  
  branch, 70, 85  
  checkout, 72  
  cherry-pick, 174  
  clone, 29  
    bare, 119  
  commit, 39, 68

config, 23, 26, 40, 66, 162, 360  
credential, 354  
daemon, 125  
describe, 177  
diff, 36  
    check, 140  
fast-import, 434  
fetch, 57  
fetch-pack, 469  
filter-branch, 434  
format-patch, 161  
gitk, 485  
gui, 485  
help, 26, 125  
http-backend, 127  
init, 29, 32  
    bare, 120, 123  
instaweb, 129  
log, 44  
merge, 79  
    squash, 160  
mergetool, 84  
p4, 420, 433  
pull, 58  
push, 58, 64, 98  
rebase, 103  
receive-pack, 468  
remote, 55, 56, 58, 59  
request-pull, 157  
rerere, 176  
send-pack, 468  
shortlog, 178  
show, 62  
show-ref, 396  
status, 31, 39  
svn, 393  
tag, 60, 61, 63  
upload-pack, 469  
git-svn, 393  
GitHub, 180  
    API, 229  
    Benutzerkonten, 180  
    Organisation, 221  
    Pull-Requests, 190  
    Workflow, 187  
GitHub für macOS, 487  
GitHub für Windows, 487  
gitk, 485  
GitLab, 130  
GitWeb, 128  
Go, 508  
go-git, 508  
GPG, 362  
Graphische Tools, 485  
GUIs, 485

**H**

Hooks, 379  
    post-update, 116

**I**

ignorieren von Dateien, 35  
Importieren  
    aus anderen VCSs, 434  
    aus Mercurial, 430  
    aus Perforce, 432  
    aus Subversion, 427  
Interoperation mit anderen VCSs  
    Mercurial, 404  
    Perforce, 412  
    Subversion, 393  
IRC-Chat, 27

**J**

Java, 505  
jgit, 505

**L**

Leerzeichen, 368  
libgit2, 499  
line endings, 368  
Linus Torvalds, 15  
Linux, 15  
    Installation, 20  
Log filtern, 50  
Log formatieren, 46

**M**

macOS  
    Installation, 21  
master, 69  
Mercurial, 404, 430  
mergetool, 365  
Merging, 81  
    -Konflikte, 82  
    -Strategien, 379

vs. Rebasen, 112

Migration zu Git, 427

Mono, 504

## O

Objective-C, 504

origin, 93

## P

Pager, 362

Perforce, 12, 15, 412, 432

  Git Fusion, 412

Policy

  Beispiel, 383

posh-git, 496

PowerShell, 496

Projektpflege Wartung, 164

Protokolle

  Dumb-HTTP, 115

  git, 118

  lokal, 113

  Smart-HTTP, 115

  SSH, 117

Pulling, 101

Pushen, 98

Python, 504, 510

## R

Rebasen, 102

  Fallstricke, 107

  vs. Merging, 112

Referenzen

  Remote, 92

Regeln

  Beispiel, 383

Releasen, 178

Rerere, 176

Ruby, 501

## S

Schlüsselwort-Erweiterung, 374

Server-Repositorys, 113

  Git-Protocol, 125

  GitLab, 130

  GitWeb, 128

  HTTP, 127

  SSH, 120

SHA-1, 17

Shell-Prompts

  bash, 493

  PowerShell, 496

  zsh, 494

SSH, 120

SSH Keys, 121

SSH Schlüssel, 121

  mit GitHub, 181

Staging-Area

  überspringen, 41

Subversion, 12, 15, 137, 393, 427

## T

Tab-Komplettierung

  bash, 493

  PowerShell, 496

  zsh, 494

Tags, 60, 176

  annotiert, 61

  lightweight, 62

  signieren, 176

## V

Versionsverwaltung, 11

  lokal, 11

  verteilt, 13

  zentral, 12

Verteiltes Git, 136

Visual Studio, 491

## W

whitespace, 368

Windows

  Installation, 21

Workflows, 136

  Diktator und Leutnants, 138

  Integrations-Manager, 137

  mergen, 170

  mergen (große), 173

  rebasen und cherry-picking, 174

  zentralisierte, 136

## X

Xcode, 21

## Z

Zeilen

  Endung, 368

[neue](#), [368](#)

[zsh](#), [494](#)