

THE EXPERT'S VOICE®

SECOND EDITION

# Pro Git

EVERYTHING YOU NEED TO  
KNOW ABOUT GIT

Scott Chacon and Ben Straub

Apress®

# Pro Git

Scott Chacon, Ben Straub

# Table of Contents

Lizenz .....	1
Vorwort von Scott Chacon .....	2
Vorwort von Ben Straub .....	4
Widmungen .....	5
Mitwirkende .....	6
Einleitung .....	8
Erste Schritte .....	10
Was ist Versionsverwaltung? .....	10
Kurzer Überblick über die Historie von Git .....	14
Was ist Git? .....	14
Die Kommandozeile .....	19
Git installieren .....	19
Git Basis-Konfiguration .....	22
Hilfe finden .....	25
Zusammenfassung .....	26
Git Grundlagen .....	27
Ein Git Repository anlegen .....	27
Änderungen nachverfolgen und im Repository speichern .....	29
Anzeigen der Commit-Historie .....	43
Ungewollte Änderungen rückgängig machen .....	49
Mit Remotes arbeiten .....	52
Tagging .....	58
Git Aliases .....	64
Zusammenfassung .....	65
Git Branching .....	66
Branches auf einen Blick .....	66
Einfaches Branching und Merging .....	72
Branch-Management .....	81
Branching-Workflows .....	83
Remote-Banches .....	86
Rebasing .....	96
Zusammenfassung .....	105
Git auf dem Server .....	106
Die Protokolle .....	106
Git auf einem Server einrichten .....	111
Erstellung eines SSH-Public-Keys .....	114
Einrichten des Servers .....	115
Git-Daemon .....	118

Smart HTTP .....	120
GitWeb .....	121
GitLab .....	123
Von Drittanbietern gehostete Optionen .....	128
Zusammenfassung .....	128
Verteiltes Git .....	130
Verteilter Arbeitsablauf .....	130
An einem Projekt mitwirken .....	133
Ein Projekt verwalten .....	157
Zusammenfassung .....	172
GitHub .....	173
Einrichten und Konfigurieren eines Kontos .....	173
Mitwirken an einem Projekt .....	178
Ein Projekt betreuen .....	198
Verwalten einer Organisation .....	212
Skripte mit GitHub .....	216
Zusammenfassung .....	227
Git Tools .....	229
Revisions-Auswahl .....	229
Interactive Staging .....	237
Stashing and Cleaning .....	241
Ihre Arbeit signieren .....	247
Searching .....	251
Rewriting History .....	255
Reset Demystified .....	263
Fortgeschrittenes Merging .....	283
Rerere .....	302
Debugging with Git .....	308
Submodules .....	311
Bundling .....	333
Replace .....	337
Anmeldeinformationen speichern .....	345
Zusammenfassung .....	350
Git einrichten .....	351
Git Konfiguration .....	351
Git-Attribute .....	362
Git Hooks .....	371
Beispiel für Git-forcierte Regeln .....	374
Zusammenfassung .....	384
Git und andere Systeme .....	385
Git als Client .....	385

Migration zu Git .....	435
Zusammenfassung .....	455
Git Interna .....	456
Basisbefehle und Standardbefehle (Plumbing and Porcelain) .....	456
Git Objekte .....	457
Git Referenzen .....	468
Packdateien (engl. Packfiles) .....	473
Die Referenzspezifikation (engl. Refspec) .....	476
Transfer Protokolle .....	479
Maintenance and Data Recovery .....	485
Environment Variables .....	492
Zusammenfassung .....	497
Appendix A: Git in Other Environments .....	499
Graphical Interfaces .....	499
Git in Visual Studio .....	504
Git in Visual Studio Code .....	506
Git in Eclipse .....	506
Git in Sublime Text .....	507
Git in Bash .....	507
Git in Zsh .....	509
Git in PowerShell .....	511
Summary .....	512
Appendix B: Embedding Git in your Applications .....	513
Command-line Git .....	513
Libgit2 .....	513
JGit .....	518
go-git .....	522
Dulwich .....	523
Appendix C: Git Kommandos .....	525
Setup und Konfiguration .....	525
Projekte importieren und erstellen .....	527
Einfache Snapshot-Funktionen .....	528
Branching und Merging .....	530
Sharing and Updating Projects .....	533
Inspection and Comparison .....	535
Debugging .....	535
Patching .....	536
Email .....	537
External Systems .....	538
Administration .....	538
Plumbing Commands .....	539

Index .....	541
-------------	-----

# Lizenz

Dieses Werk ist lizenziert unter der „Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported“ Lizenz. Um eine Kopie dieser Lizenz zu lesen, besuchen Sie <https://creativecommons.org/licenses/by-nc-sa/3.0> oder senden Sie 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 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 die fortgeschrittenen Anwender. In einigen Communitys 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 das Werkzeug. 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 es 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.

Ein 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 auch viel 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, Sie haben 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, bin ich ein Teil einer bedeutenden Git-Implementierung, ich habe für das größte Git-Hosting-Unternehmen gearbeitet und ich bin durch die ganze Welt gereist, um Menschen über Git zu unterrichten. Als Scott mich fragte, ob ich Interesse hätte, an der zweiten Ausgabe des Buches mitzuarbeiten, musste ich nicht einmal darüber nachdenken.

Es war mir eine große Freude und ein Privileg, an diesem Buch mitzuwirken. Ich hoffe, es wird Ihnen 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.

	Hamidreza Mahdavipanah	paveljanik
4wk-	haripetrov	Pavel Janík
Adam Laflamme	Haruo Nakayama	Paweł Krupiński
Adrien Ollier	Helmut K. C. Tessarek	pedrorijo91
ajax333221	HonkingGoose	Peter Kokot
Akrom K	Howard	peterwwillis
Aleh Suprunovich	i-give-up	petsuter
Alexander Bezzubov	Ignacy	Philippe Miossec
Alexandre Garnier	Ilker Cat	Philipp Montesano
alex-koziell	iprok	Phil Mitchell
Alfred Myers	Jan Groenewald	Rafi
allen joslin	Jaswinder Singh	rahrah
Andrei Dascalu	Jean-Noël Avila	Ralph Haussmann
Andrew MacFie	Jeroen Oortwijn	Raphael R
Andrew Metcalf	Jim Hill	Ray Chen
Andrew Murphy	jingsam	Rex Kerr
AndyGee	J.M. Rütter	Reza Ahmadi
AnneTheAgile	Joel Davies	Richard Hoyle
Anthony Loiseau	Johannes Dewender	Ricky Senft
Antonello Piemonte	Johannes Schindelin	Rintze M. Zelle
Antonino Ingargiola	johnhar	rmzelle
Anton Trunov	John Lin	Rob Blanco
atalakam	Jon Forrest	Robert P. Goldman
Atul Varma	Jon Freed	Robert P. J. Day
axmbo	Jordan Hayashi	Rohan D'Souza
Benjamin Dopplinger	Jörg Ewald	Roman Kosenko
Ben Sima	Joris Valette	Ronald Wampler
Borek Bernard	Josh Byster	root
Brett Cannon	Joshua Webb	Rüdiger Herrmann
bripmccann	Junjie Yuan	Sam Ford
brotherben	Justin Clift	Sam Joseph
Buzut	Kaartic Sivaraam	Sanders Kleinfeld
Cadel Watson	Katrin Leinweber	sanders@oreilly.com
Carlos Martín Nieto	Kausar Mehmood	Sarah Schneider
Chaitanya Gurrapu	Keith Hill	SATO Yusuke
Changwoo Park	Kenneth Kin Lum	Saurav Sachidanand
Christopher Wilson	Klaus Frank	Scott Bronson
Christoph Prokop	Kristijan "Fremen" Velkovski	Sean Head
C Nguyen	Krzysztof Szumny	Sebastian Krause
CodingSpiderFox	Kyrylo Yatsenko	Severino Lorilla Jr
Cory Donnelly	Lars Vogel	Shengbin Meng
Cullen Rhodes	Laxman	Siarhei Bobryk

Cyril	Lazar95	Siarhei Krukau
Damien Tournoud	Leonard Laszlo	Skyper
Daniele Tricoli	Linus Heckemann	Snehal Shekatkar
Daniel Shahaf	Logan Hasson	Song Li
Daniel Stauffer	Louise Corrigan	spacewander
Daniel Sturm	Luc Morin	Stephan van Maris
Daniil Larionov	Lukas Röllin	Steven Roddis
Danny Lin	maks	SudarsanGP
Dan Schmidt	Marcin Sędłak-Jakubowski	Suhaib Mujahid
Davide Angelocola	Marie-Helene Burle	Sven Selberg
David Rogers	Marius Žilėnas	td2014
delta4d	Markus KARG	Thanix
Denis Savitskiy	Marti Bolivar	Thomas Ackermann
devwebcl	Mashrur Mia (Sa'ad)	Thomas Hartmann
DiamonddeX	Masood Fallahpoor	Tomoki Aonuma
Dieter Ziller	Mathieu Dubreuilh	Tom Schady
Dino Karic	Matthew Miner	Tvirus
Dmitri Tikhonov	Matthieu Moy	twekberg
dualsky	Michael MacAskill	Tyler Cipriani
Duncan Dean	Michael Sheaver	Ud Yzr
Eden Hochbaum	Michael Welch	uerdogan
Eric Henziger	Michiel van der Wulp	un1versal
evanderiel	Mike Charles	Vadim Markovtsev
Explorare	Mike Pennisi	Vangelis Katsikaros
eyherabh	Mike Thibodeau	Victor Ma
Ezra Buehler	mmikeww	Vitaly Kuznetsov
Felix Nehrke	mosdalsvsold	William Gathoye
Fernsehkind	nicktime	William Turrell
Filip Kucharczyk	Niels Widger	Wlodek Bzyl
flip111	Nils Reuße	Xavier Bonaventura
flyingzumwalt	Olleg Samoylov	xJom
Fornost461	Oscar Santiago	xtreak
Frank	Owen	yakirwin
Frederico Mazzone	Pablo Schläpfer	Yann Soubeyrand
Frej Drejhämmar	Pascal Berger	Yue Lin Ho
goekboet	Pascal Borreli	Yunhai Luo
grgbnc	pastatopf	Yusuke SATO
Guthrie McAfee Armstrong	patrick96	zwPapEr
HairyFotr	Patrick Steinhardt	狹

# Einleitung

Sie sind 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 Sie auf Lager haben. Auf dieser Seite finden Sie 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 Fachwissen, nur das was mit Git verbunden ist, warum es in einem Land voller VCSs entstanden ist, was es auszeichnet und warum so viele Menschen es benutzen. Dann werden wir beschreiben, wie Sie Git herunterladen und zum ersten Mal einrichten können, wenn Sie es noch nicht auf Ihrem System installiert haben.

In **Kapitel 2** gehen wir auf die grundlegende Git-Verwendung ein – wie Sie Git in den 80% der Fälle verwenden, denen Sie am häufigsten begegnen. Nachdem Sie dieses Kapitel gelesen haben, sollten Sie in der Lage sein, ein Repository zu klonen, zu sehen, was in der Verlaufshistorie des Projekts passiert ist, Dateien zu ändern und Veränderungen beizutragen. Angenommen dieses Buch geht in diesem Augenblick in Flammen auf, dann sollten Sie trotzdem schon in der Lage sein, so weit bei der Anwendung von Git zu helfen, um die Zeit zu überbrücken bis ein neues Exemplar dieses Buches beschafft ist.

In **Kapitel 3** geht es um das Branching-Modell von Git, das oft als seine Killer-Funktion beschrieben wird. Hier erfahren Sie, was Git wirklich von der Masse abhebt. Wenn Sie das Kapitel zu Ende gebracht haben, werden Sie vielleicht in einen ruhigen Moment darüber nachzudenken, wie sie ohne das Branching von Git leben könnten.

**Kapitel 4** befasst sich mit Git auf dem Server. Mit diesem Kapitel wenden wir uns an diejenigen von Ihnen, 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 Sie es vorziehen, dass jemand anderes diese Aufgabe für Sie übernimmt.

**Kapitel 5** geht ausführlich auf verschiedene dezentrale Workflows ein und wie man sie mit Git realisiert. Wenn Sie dieses Kapitel durchgearbeitet haben, sollten Sie in der Lage sein, professionell mit mehreren Remote-Repositories 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 Repositories, gemeinsame Workflows, um zu Projekten beizutragen und Beiträge für Ihre Projekte anzunehmen, die Programmoberfläche von GitHub und viele kleine Tipps, die Ihnen das tägliche Arbeiten im allgemeinen erleichtern.

**Kapitel 7** befasst sich mit anspruchsvollen Git-Befehlen. Hier erfahren Sie 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 Ihr Wissen über Git abrunden, so dass Sie ein echter Meister werden.

**Kapitel 8** behandelt die Konfiguration Ihrer individuellen Git-Umgebung. Dazu gehört die Einrichtung von Hook-Skripten zur Durchsetzung oder Unterstützung angepasster Regeln und die

Verwendung von Konfigurationseinstellungen für die Benutzerumgebung, damit Sie so arbeiten können, wie sie es sich vorstellen. Wir werden auch die Erstellung eines eigenen Skript-Sets für die Umsetzung einer benutzerdefinierten Commit-Richtlinie in die Praxis erörtern.

**Kapitel 9** beschäftigt sich mit Git und anderen VCS-Systemen. Dazu gehört die Verwendung von Git in einer Subversion (SVN) Umgebung und die Umwandlung von Projekten aus anderen VCSs nach Git. Viele Unternehmen verwenden immer noch SVN und sind nicht dabei, das zu ändern, aber an dieser Stelle haben Sie die unglaubliche Leistungsfähigkeit von Git kennen gelernt – dieses Kapitel zeigt Ihnen, wie Sie damit umgehen können, wenn Sie noch einen SVN-Server verwenden müssen. Wir besprechen auch, wie man Projekte aus unterschiedlichen Systemen importiert, falls Sie alle davon überzeugt haben, den Sprung zu wagen.

**Kapitel 10** vertieft die dunklen und zugleich wundervollen Tiefen der Git-Interna. Jetzt, da Sie alles über Git wissen und es mit Macht und Eleganz bedienen können, können Sie 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 Sie an diesem Punkt das Bedürfnis haben, tiefer in die technischen Details einzutauchen und so sind wie wir, sollten Sie vielleicht zuerst Kapitel 10 lesen. Das überlassen wir ganz Ihnen.

In **Anhang A** schauen wir uns eine Reihe von Beispielen für den Git-Einsatz in verschiedenen speziellen Anwendungsumgebungen an. Wir befassen uns mit einer Anzahl verschiedener GUIs und Entwicklungs-Umgebungen, in denen Sie Git einsetzen könnten und welche verfügbar sind. Wenn Sie an einem Überblick über die Verwendung von Git in Ihrer Shell, Ihrer IDE oder Ihrem Texteditor interessiert sind, schauen Sie hier nach.

In **Anhang B** erkunden wir das Scripting und die Erweiterung von Git durch Tools wie libgit2 und JGit. Wenn Sie an der Entwicklung komplexer, schneller und benutzerdefinierter Tools interessiert sind und einen Low-Level-Git-Zugang benötigen, können Sie hier nachlesen, wie diese Szene ausschaut.

Schließlich gehen wir in **Anhang C** alle wichtigen Git-Befehle einzeln durch und wiederholen, wo wir sie in dem Buch behandelt haben und was wir dabei gemacht haben. Wenn die Frage beantwortet werden soll, wo im Buch wir einen bestimmten Git-Befehl verwendet haben, können Sie das hier nachlesen.

Lassen Sie uns beginnen.

# Erste Schritte

In diesem Kapitel wird es darum gehen, wie Sie mit Git zu beginnen können. Wir starten mit einer Erläuterung des Hintergrunds zu Versionskontroll-Systemen, gehen dann weiter zu dem Thema, wie Sie Git auf Ihrem System zum Laufen bringen und schließlich wie Sie es einrichten können, sodass Sie in der Lage sind, die ersten Schritte mit Git zu tun. Am Ende dieses Kapitels sollten Sie verstehen, wozu Git gut ist und weshalb man es verwenden sollte und Sie sollten in der Lage sein, mit 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.

Als Grafik- oder Webdesigner möchte man zum Beispiel in der Lage sein, jede Version eines Bildes oder Layouts nachzuverfolgen zu können. Als solcher wäre es deshalb ratsam, ein Versionsverwaltungssystem (engl. Version Control System, VCS) einzusetzen. Ein solches System erlaubt es, einzelne Dateien oder auch ein ganzes Projekt in einen früheren Zustand zurückzuversetzen, nachzuvollziehen, wer zuletzt welche Änderungen vorgenommen hat, die möglicherweise Probleme verursachen, herauszufinden wer eine Änderung ursprünglich vorgenommen hat und viele weitere Dinge. Ein Versionsverwaltungssystem bietet allgemein die Möglichkeit, jederzeit zu einem vorherigen, funktionierenden Zustand zurückzukehren, auch wenn man einmal Mist gebaut oder aus irgendeinem Grunde Dateien verloren hat. All diese Vorteile erhält man für einen 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). Diese Vorgehensweise ist sehr weit verbreitet und wird gern verwendet, weil sie so einfach ist. Aber sie ist eben 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 dann 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 versionierten 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.

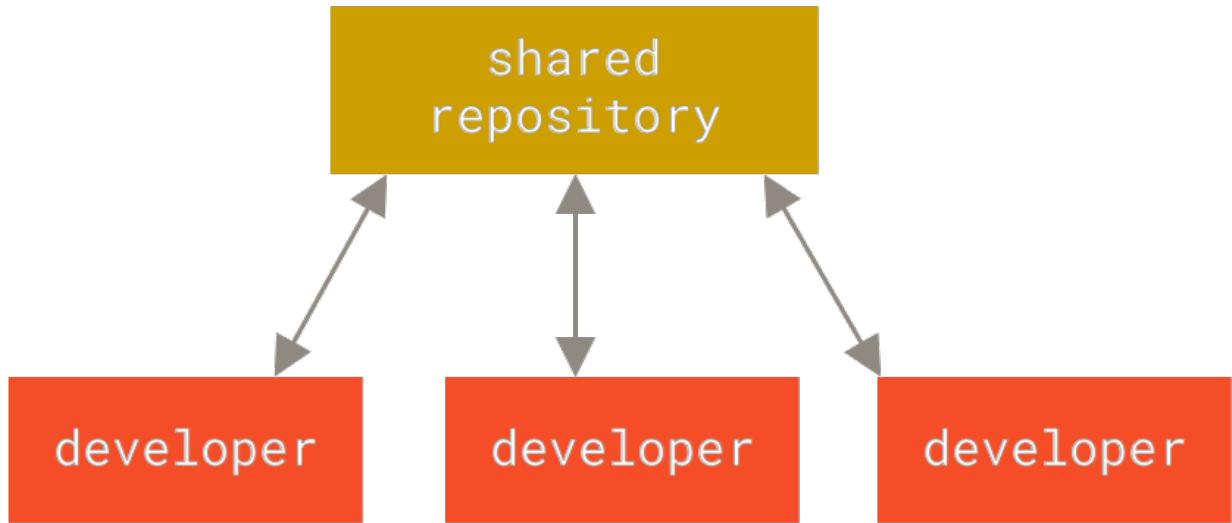


Figure 2. Zentrale Versionsverwaltung

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, nämlich dann wenn der zentralisierte 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 Mitarbeiter 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 ernsthaft 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 Repositorys. 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 Repositorys, sogenannten Remote-Repositorys, 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 gesamten Entwicklungszeit des Linux Kernels von 1991 bis 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, der 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 Entwicklungszweige)
- 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 in Kürze? Das ist ein wichtiger Teil, den es zu beachten gilt, denn wenn Sie verstehen, was Git und das grundlegenden Konzept seiner Arbeitsweise ist, dann wird die effektive Nutzung von Git für Sie wahrscheinlich viel einfacher sein. Wenn Sie Git erlernen, versuchen Sie, Ihren Kopf von den Dingen zu befreien, die Sie über andere VCSs wissen, wie CVS, Subversion oder Perforce – das wird Ihnen helfen, unangenehme Missverständnisse bei der Verwendung des Tools zu vermeiden. Auch wenn die Benutzeroberfläche von Git diesen anderen VCSs sehr ähnlich ist, speichert und betrachtet Git Informationen auf eine ganz andere Weise, und das Verständnis dieser Unterschiede hilft Ihnen Unklarheiten bei der Verwendung zu vermeiden.

## Snapshots und nicht die Unterschiedes

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

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

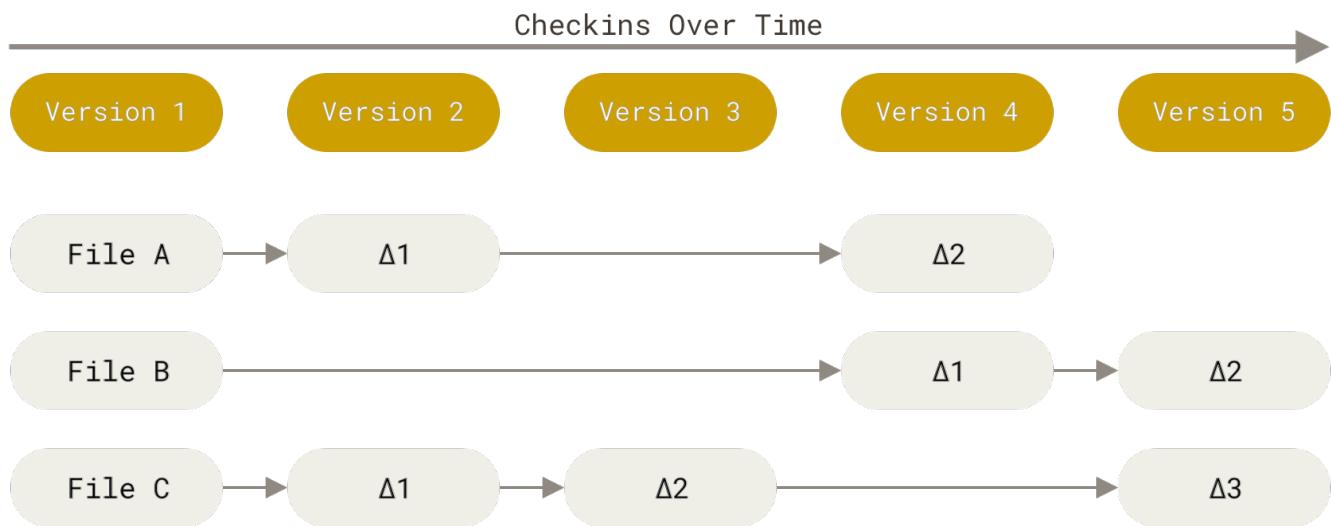


Figure 4. Speichern von Daten als Änderung an einzelnen Dateien auf Basis einer Ursprungsdatei.

Git arbeitet nicht auf diese Art und Weise. Stattdessen betrachtet Git seine Daten eher wie eine Reihe von Schnapschüssen eines Mini-Dateisystems. Git macht jedes Mal, wenn Sie den Status Ihres Projekts committen, das heißt den gegenwärtigen Status Ihres Projekts als eine Version in Git speichern, ein Abbild von all Ihren Dateien wie sie gerade aussehen und speichert einen Verweis in diesem Schnapschuss. 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 wie einen **Stapel von Schnapschüssen**.

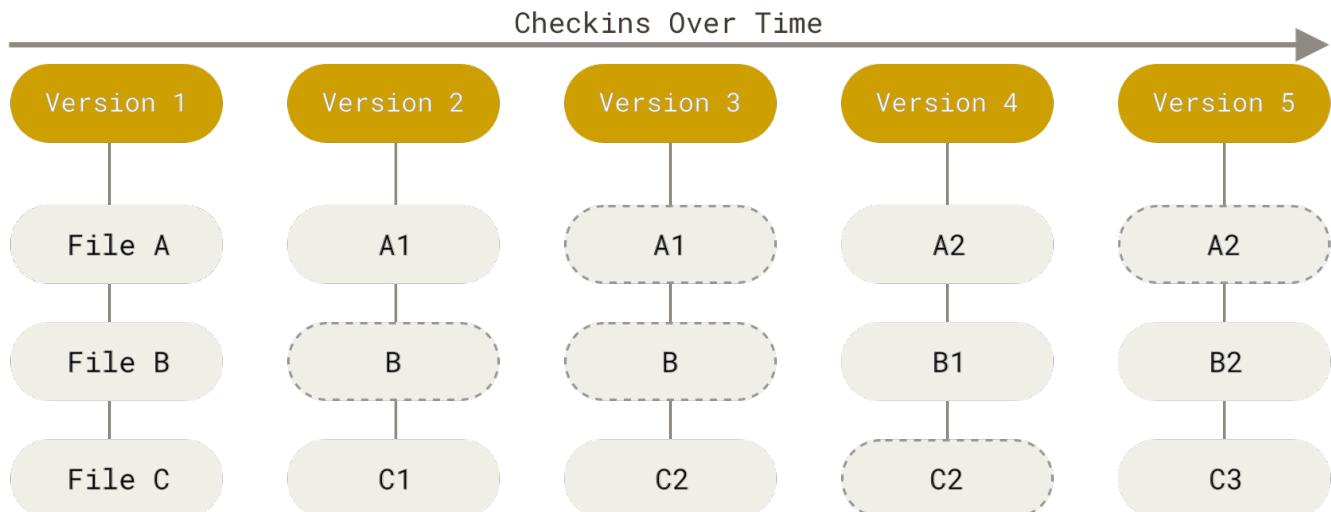


Figure 5. Speichern der Daten-Historie eines Projekts in Form von Schnapschüssen.

Das ist ein wichtiger Unterschied zwischen Git und praktisch allen anderen Versionsverwaltungssystemen. In Git wurden daher 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 Ihrem Netzwerk benötigt. Wenn Sie an ein CVCS gewöhnt sind, in dem die meisten Operationen diesen Netzwerk-Latenz-Overhead haben, wird Sie dieser Aspekt von Git denken lassen, dass die Götter der Geschwindigkeit Git mit außerirdischen Kräften gesegnet haben. 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 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 dort bestimmen zu lassen.

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 einen 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 können Sie beispielsweise nicht viel tun, wenn Sie nicht mit dem Server verbunden sind; in Subversion und CVS können Sie Dateien bearbeiten, aber Sie können keine Änderungen zu Ihren Daten übertragen (weil die Datenbank offline ist). Das scheint keine große Sache zu sein, aber Sie werden überrascht sein, was für einen großen Unterschied es 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 und 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 Sie Aktionen in Git durchführen, werden fast alle von ihnen 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 wird 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, der sogenannten Staging-Area und dem Git-Verzeichnis.

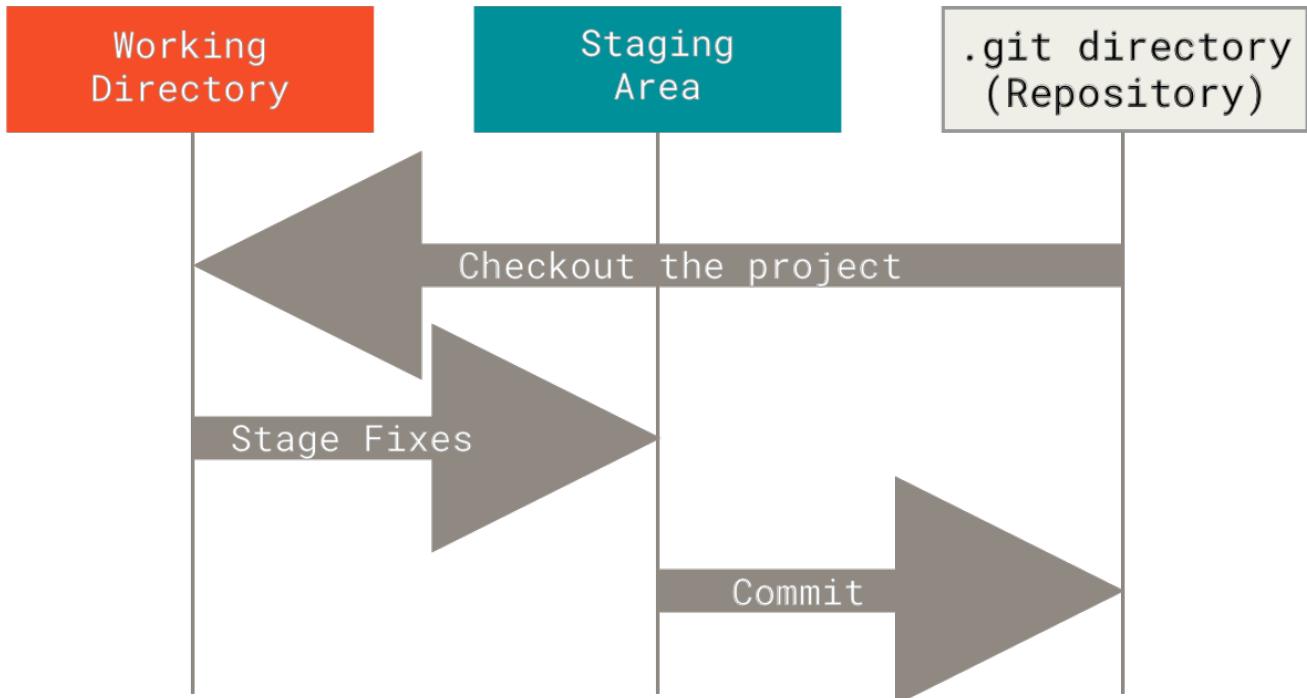


Figure 6. Verzeichnisbaum, Staging-Area und Git-Verzeichnis.

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

Die Staging-Area ist in der Regel eine Datei, die sich in Ihrem Git-Verzeichnis befindet und Informationen darüber speichert, was in Ihren 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 Ihr 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. Sie ändern Dateien in Ihrem Verzeichnisbaum.
2. Sie stellen selektiv Änderungen bereit, die Sie bei Ihrem nächsten Commit berücksichtigen möchten, wodurch nur diese Änderungen in den Staging-Bereich aufgenommen werden.
3. Sie führen einen Commit aus, der die Dateien so übernimmt, wie sie sich in der Staging-Area befinden und diesen Snapshot dauerhaft in Ihrem 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 Sie sich in der Kommandozeilenversion von Git auskennen, finden Sie sich 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 Sie wissen, wie man bei einem Mac ein Terminal öffnet, oder wie man unter Windows die Eingabeaufforderung oder die Powershell öffnet. Sollten Sie jetzt nur Bahnhof verstehen, sollten Sie an dieser Stelle abbrechen und sich schlau machen, was ein Terminal bzw. Eingabeaufforderung ist und wie man diese bedient. Nur so ist sichergestellt, dass Sie unseren Beispielen und Ausführungen im weiteren Verlauf des Buches folgen können.

## Git installieren

Bevor Sie mit Git loslegen können, muss es natürlich zuerst installiert werden. Auch wenn es bereits vorhanden ist, ist es vermutlich eine gute Idee, auf die neueste Version zu aktualisieren. Sie können 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 **2.8.0** geschrieben. 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. Da in Git sehr auf Abwärtskompatibilität geachtet wird, sollten aber alle neueren Versionen nach der Version 2.0 genauso gut funktionieren.

## Installation unter Linux

Wenn Sie die grundlegenden Git-Tools unter Linux über ein Installationsprogramm installieren möchten, können Sie das in der Regel über das Paketverwaltungstool der Distribution tun. Wenn Sie mit Fedora (oder einer anderen eng damit verbundenen RPM-basierten Distribution, wie RHEL oder CentOS) arbeiten, können Sie `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 <http://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 <http://git-scm.com/download/mac> findet man die jeweils aktuellste Version und kann sie von dort herunterladen..



Figure 7. Git macOS Installationsprogramm.

Ein weitere Möglichkeit ist „GitHub for Mac“ zu installieren. Es gibt dort eine Option, dass man neben der grafischen Oberfläche auch gleich die Kommandozeilen Werkzeuge mit installieren

kann. „GitHub for Mac“ kann man unter <http://mac.github.com> herunterladen.

## Installation unter Windows

Auch für Windows gibt es einige, wenige Möglichkeiten zur Installation von Git. Eine offizielle Windows Version findet man direkt auf der Git Homepage. Gehen Sie dazu auf <http://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 finden Sie unter <http://msysgit.github.io/>.

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

Eine weitere einfache Möglichkeit, Git zu installieren, ist die Installation von GitHub Desktop. Das Installationsprogramm enthält neben der GUI, auch eine Kommandozeilenversion von Git. Diese funktioniert zusammen mit der Powershell tadellos, und richtet die wichtigsten Berechtigungs-Caching (engl. credential caching) und Zeilenenden-Konfigurationen (CRLF) vorab ein. Wir werden später etwas mehr darüber erfahren, jetzt reicht es festzustellen, dass es sich um die wünschenswerte Funktionen handelt. Sie können das ganze „GitHub for Windows“ Paket von der [GitHub Desktop Website](#) herunterladen.

## 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. Heutzutage ist dies nicht mehr ganz so schlimm, da Git einen gewissen Reifegrad erfahren hat.

Wenn Sie Git aus dem Quellcode installieren möchten, benötigen Sie die folgenden Bibliotheken, von denen Git abhängt: autotools, curl, zlib, openssl, expat und libiconv. Wenn Sie sich beispielsweise auf einem System befinden, das Paketverwaltungen, wie `dnf` (Fedora) oder `apt-get` (ein Debian-basiertes System) hat, können Sie 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 libbz-dev libssl-dev
```

Um die Dokumentation in verschiedenen Formaten (doc, html, info) zu erstellen, sind weitere Abhängigkeiten notwendig (Hinweis: Benutzer von RHEL und RHEL-Derivaten wie CentOS und Scientific Linux müssen das [EPEL-Repository aktivieren](#), um das Paket `docbook2X` herunterzuladen):

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

Wenn Sie eine Debian-basierte Distribution (Debian, Ubuntu oder deren Derivate) verwenden, benötigen Sie auch das Paket `install-info`:

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

Wenn Sie eine RPM-basierte Distribution (Fedora, RHEL oder deren Derivate) verwenden, benötigen Sie auch das Paket **getopt** (welches auf einer Debian-basierten Distribution bereits installiert ist):

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

Wenn Sie Fedora- oder RHEL-Derivate verwenden, müssen Sie wegen der unterschiedlichen Paketnamen zusätzlich einen Symlink erstellen indem Sie folgenden Befehl ausführen:

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

Wenn Sie alle notwendigen Abhängigkeiten installiert haben, können Sie sich 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-2.0.0.tar.gz  
$ cd git-2.0.0  
$ make configure  
$ ./configure --prefix=/usr  
$ make all doc info  
$ sudo make install install-doc install-html install-info
```

Jetzt nachdem Git installiert ist, kann man sich Git Updates auch per Git beschaffen:

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

## Git Basis-Konfiguration

Nachdem Git jetzt auf Ihrem System installiert ist, sollten Sie Ihre Git Konfiguration noch anpassen. Dies muss nur einmalig auf dem jeweiligen System durchgeführt werden. Die Konfiguration bleibt bestehen, wenn Sie Git auf eine neuere Version aktualisieren. 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 können Sie anpassen, wie Git aussieht und arbeitet. Die Konfiguration ist an drei verschiedenen Orten gespeichert:

1. Die Datei `/etc/gitconfig`: enthält Werte, die für jeden Benutzer auf dem System und alle seine Repositories gelten. Wenn Sie die Option `--system` an `git config` übergeben, liest und schreibt sie spezifisch aus dieser Datei. (Da es sich um eine Systemkonfigurationsdatei handelt, benötigen Sie Administrator- oder Superuser-Rechte, um Änderungen daran vorzunehmen.)
2. Die Datei `~/.gitconfig` oder `~/.config/git/config`: enthält Werte, die für Sie, den Benutzer, persönlich bestimmt sind. Sie können Git dazu bringen, diese Datei gezielt zu lesen und zu schreiben, indem Sie die Option `--global` übergeben, und dies betrifft *alle* der Repositories, mit denen Sie auf Ihrem System arbeiten.
3. Die Datei `config` im Git Verzeichnis (also `.git/config`) des jeweiligen Repositories, das Sie gerade verwenden: Sie können 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 Sie nicht überraschen, dass Sie sich irgendwo in einem Git-Repository befinden müssen, damit diese Option ordnungsgemäß funktioniert.)

Jede Ebene überschreibt Werte der vorherigen Ebene, so dass Werte in `'.git/config'` diejenigen in `'/etc/gitconfig'` aushebeln.

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 `/etc/gitconfig`, auch wenn die sich relativ zu dem MSys-Wurzelverzeichnis befindet, dem Verzeichnis in das Sie Git installiert haben. Wenn Sie eine Version 2.x oder neuer von Git für Windows verwenden, 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.

Sie können sich alle Ihre Einstellungen ansehen sehen, wo sie herkommen:

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

## Ihre Identität

Nachdem Sie Git installiert haben, sollten Sie als aller erstes Ihren Namen und 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 Sie erstellen:

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

Wie schon erwähnt, brauchen Sie diese Konfiguration nur einmal vorzunehmen, wenn Sie die

Option `--global` verwenden. Auf Grund der oben erwähnten Prioritäten gilt das dann für alle Ihre Projekte. Wenn Sie für ein spezielles Projekt einen anderen Namen oder eine andere E-Mail Adresse verwenden möchten, können Sie den Befehl ohne die `--global` Option innerhalb des Projektes ausführen.

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

## Ihr Editor

Jetzt, da Ihre Identität eingerichtet ist, können Sie den Standard-Texteditor konfigurieren, der verwendet wird, wenn Git Sie zum Eingeben einer Nachricht auffordert. Normalerweise verwendet Git den Standard-Texteditor des jeweiligen Systems.

Wenn Sie einen anderen Texteditor, z.B. Emacs, verwenden wollen, können Sie das wie folgt festlegen:

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

Wenn Sie auf einem Windows-System einen anderen Texteditor verwenden möchten, müssen Sie den vollständigen Pfad zu seiner ausführbaren Datei angeben. Dies kann, je nachdem, wie Ihr Editor eingebunden ist, unterschiedlich sein.

Im Falle von Notepad++, einem beliebten Programmiereditor, sollten Sie 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 geben Sie 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 Sie einen anderen Editor oder eine 32-Bit-Version verwenden, finden Sie in [git config core.editor commands](#) spezielle Anweisungen, wie Sie Ihren bevorzugten Editor mit Git einrichten können.



Wenn Sie Git nicht so konfigurieren, dass es Ihren Texteditor verwendet und Sie keine Ahnung davon haben, wie man Vim oder Emacs benutzt, werden Sie ein wenig überfordert sein, wenn diese zum ersten Mal von Git gestartet werden. Ein Beispiel: auf einem Windows-System kann es eine vorzeitig beendete Git-Operation während einer von Git angestoßenen Verarbeitung sein.

## Einstellungen überprüfen

Wenn Sie Ihre Konfigurationseinstellungen überprüfen möchten, können Sie mit dem Befehl `git`

`config --list` alle Einstellungen auflisten, die Git derzeit finden kann:

```
$ git config --list
user.name=John Doe
user.email=johndoe@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. `/etc/gitconfig` und `~/.gitconfig`) gefunden hat. In diesem Fall verwendet Git dann den jeweils zuletzt aufgelisteten Wert.

Außerdem können Sie 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 Sie einen unerwarteten Wert für einen dieser Werte haben und nicht wissen, warum. In solchen Fällen können Sie Git nach dem *Ursprung (origin)* für diesen Wert fragen, und es wird Ihnen sagen, mit welcher Konfigurationsdatei der Wert letztendlich festgelegt wurde:

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

## Hilfe finden

Falls Sie einmal Hilfe bei der Anwendung von Git benötigen, 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 erhalten Sie die Hilfeseite für den `git config` Befehl folgendermaßen:

```
$ git help config
```

Diese Befehle sind nützlich, weil Sie sich die Hilfe jederzeit anzeigen lassen können, auch wenn Sie einmal offline sind. Wenn die Hilfeseiten und dieses Buch nicht ausreichen und Sie persönliche Hilfe brauchen, können Sie den **#git** oder **#github** Kanal auf dem Freenode IRC Server probieren, der unter <https://freenode.net> 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 Ihnen weiterhelfen kann.

Wenn Sie nicht die vollständige Manpage-Hilfe benötigen, sondern nur eine kurze Beschreibung der verfügbaren Optionen für einen Git-Befehl, können Sie auch in den kompakteren „Hilfeseiten“ mit der **-h** Option 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
```

## Zusammenfassung

Sie sollten jetzt ein grundlegendes Verständnis dafür haben, was Git ist und wie es sich von anderen zentralisierten Versionsverwaltungs-Systemen unterscheidet, die Sie ev. bisher eingesetzt haben. Sie sollten nun auch eine funktionierende Version von Git auf Ihrem System haben, die mit Ihrer persönlichen Identität eingerichtet ist. Es ist jetzt an der Zeit, einige Grundlagen von Git zu lernen.

# Git Grundlagen

Wenn Sie nur ein Kapitel durcharbeiten können/wollen, um mit Git zu beginnen, dann ist dieses hier das richtige. Dieses Kapitel behandelt alle grundlegenden Befehle, die Sie benötigen, um die überwiegende Anzahl der Aufgaben zu erledigen, die Sie irgendwann einmal mit Git erledigen werden. Am Ende des Kapitels sollten Sie in der Lage sein, ein neues Repository anzulegen und zu konfigurieren, Dateien zu versionieren bzw. aus der Versionsverwaltung zu entfernen, Dateien in die Staging-Area hinzuzufügen und schließlich einen Commit durchzuführen. Außerdem wird gezeigt, wie Sie Git so konfigurieren können, dass es bestimmte Dateien und Dateimuster ignoriert, wie Sie Fehler schnell und einfach rückgängig machen, wie Sie die Historie eines Projekts durchsuchen und Änderungen zwischen Commits nachschlagen, und wie Sie von einem Remote-Repository Daten herunter- bzw. dort hochladen können.

## Ein Git Repository anlegen

Sie haben zwei Möglichkeiten, ein Git Repository auf Ihrem Rechner anzulegen.

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

In beiden Fällen erhalten Sie ein einsatzbereites Git-Repository auf Ihrem lokalen Rechner.

## Ein Repository in einem bestehenden Verzeichnis einrichten

Wenn Sie ein Projektverzeichnis haben, das sich derzeit nicht unter Versionskontrolle befindet, und Sie mit der Kontrolle über Git beginnen möchten, müssen Sie zunächst in das Verzeichnis dieses Projekts wechseln. Wenn Sie dies noch nie getan haben, sieht es je nachdem, welches System Sie verwenden, etwas anders aus:

für Linux:

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

für macOS:

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

für Windows:

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

dann folgenden Befehl eingeben:

```
$ git init
```

Der Befehl erzeugt ein Unterverzeichnis `.git`, in dem alle relevanten Git Repository Daten enthalten sind, also ein Git Repository Grundgerüst. Zu diesem Zeitpunkt werden noch keine Dateien in Git versioniert. (In Kapitel 10, [Git Interna](#), finden Sie weitere Informationen, welche Dateien im `.git` Verzeichnis enthalten sind und was ihre Aufgabe ist.)

Wenn Sie bereits existierende Dateien versionieren möchten (und es sich nicht nur um ein leeres Verzeichnis handelt), dann sollten Sie den aktuellen Stand in einem initialen Commit starten. Mit dem Befehl `git add` legen Sie fest, welche Dateien versioniert werden sollen und mit dem Befehl `git commit` erzeugen Sie einen neuen Commit:

```
$ 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, dass Sie verstehen, dass Sie jetzt ein Git Repository erzeugt und einen ersten Commit angelegt haben.

## Ein existierendes Repository klonen

Wenn Sie eine Kopie eines existierenden Git Repositorys anlegen wollen – um beispielsweise an einem Projekt mitzuarbeiten – können Sie den Befehl `git clone` verwenden. Wenn Sie bereits Erfahrung mit einem anderen VCS System, wie Subversion, gesammelt haben, fällt Ihnen bestimmt sofort auf, dass der Befehl „clone“ und nicht etwa „checkout“ lautet. Das ist ein wichtiger Unterschied, den Sie unbedingt verstehen sollten – 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 Sie `git clone` ausführen. Wenn Ihre Serverfestplatte beschädigt wird, können Sie oft 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 (Sie werden vielleicht einige serverseitige Hooks und dergleichen verlieren, aber alle versionierten Daten wären vorhanden – siehe Kapitel 4, [Git auf dem Server](#), für weitere Details).

Sie können ein Repository mit dem Befehl `git clone [url]` klonen. Um beispielsweise das Repository der verlinkbaren Git Bibliothek `libgit2` zu klonen, führen Sie den 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 Arbeitskopie der aktuellsten Version aus. Wenn Sie in das neue `libgit2` Verzeichnis wechseln, finden Sie dort die Projektdateien und können gleich damit arbeiten.

Wenn Sie das Repository in ein Verzeichnis mit einem anderen Namen als `libgit2` klonen möchten,

können Sie 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 Ihnen können 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 Ihr Git-Repository hat und die Vor- und Nachteile der einzelnen Optionen, die man einrichten kann.

## Änderungen nachverfolgen und im Repository speichern

An dieser Stelle sollten Sie ein *originalgetreues* Git-Repository auf Ihrem lokalen Computer und eine Checkout- oder Arbeitskopie aller seiner Dateien vor sich haben. Normalerweise werden Sie damit beginnen wollen, Änderungen vorzunehmen und Schnapschüsse dieser Änderungen in Ihr Repository zu übertragen, wenn das Projekt so weit fortgeschritten ist, dass Sie es sichern möchten.

Denken Sie daran, dass sich jede Datei in Ihrem Arbeitsverzeichnis in einem von zwei Zuständen befinden kann: *tracked* oder *untracked* – Änderungen an der Datei werden verfolgt (engl. *tracked*) oder eben nicht (engl. *untracked*). Alle Dateien, die sich im letzten Schnapschuss (Commit) befanden, werden in der Versionsverwaltung nachverfolgt. Sie können entweder unverändert, modifiziert oder für den nächsten Commit vorgemerkt (staged) sein. Kurz gesagt sind getrackte, versionierte Dateien, Daten, die Git kennt.

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

Sobald Sie anfangen, versionierte Dateien zu bearbeiten, erkennt Git diese als modifiziert, weil sie sich im Vergleich zum letzten Commit verändert haben. Die geänderten Dateien können Sie dann für den nächsten Commit vormerken und schließlich alle Änderungen, die sich in der Staging-Area befinden, einchecken/committen. Danach wiederholt sich der Zyklus.

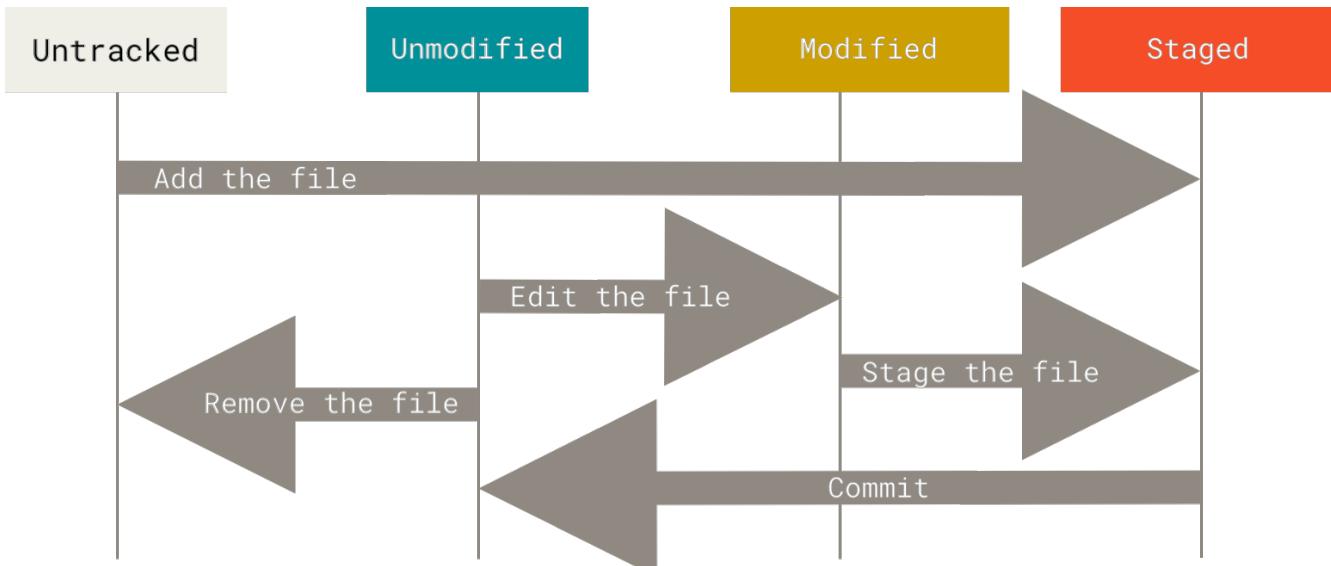


Figure 8. Der Status Ihrer Dateien im Überblick

## Zustand von Dateien prüfen

Das wichtigste Hilfsmittel, um den Zustand zu überprüfen, in dem sich Ihre Dateien gerade befinden, ist der Befehl `git status`. Wenn Sie diesen Befehl unmittelbar nach dem Klonen eines Repositorys ausführen, 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 directory 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 Ihnen der Befehl mit, auf welchem Branch Sie gerade arbeiten und informiert Sie darüber, dass dieser sich im Vergleich zum Branch auf dem Server nicht verändert hat. Momentan ist dieser Zweig immer `master`, was der Vorgabe entspricht; Sie müssen sich jetzt nicht darum kümmern. Wir werden im Kapitel [Git Branching](#) auf Branches detailliert eingehen.

Nehmen wir einmal an, Sie fügen eine neue Datei mit dem Namen `README` zu Ihrem Projekt hinzu. Wenn die Datei zuvor nicht existiert hat und Sie jetzt `git status` ausführen, 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 Schnapschuss nicht enthalten war. Git nimmt eine solche Datei nicht automatisch in die Versionsverwaltung auf, sondern Sie müssen Git dazu ausdrücklich auffordern. Ansonsten würden generierte Binärdateien oder andere Dateien, die Sie nicht in Ihrem Repository haben möchten, 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, können Sie den Befehl **git add** verwenden. Für Ihre neue **README** Datei, können Sie folgendes ausführen:

```
$ git add README
```

Wenn Sie den **git status** Befehl erneut ausführen, werden Sie sehen, dass sich Ihre **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 reset HEAD <file>..." to unstage)

    new file:   README
```

Sie können erkennen, dass die Datei für den nächsten Commit vorgemerkt ist, weil sie unter der Rubrik „Changes to be committed“ aufgelistet ist. Wenn Sie jetzt einen Commit anlegen, wird der Schnapschuss den Zustand der Datei festhalten, den sie zum Zeitpunkt des Befehls **git add** hat. Sie erinnern sich vielleicht daran, wie Sie vorhin **git init** und anschließend **git add <files>** ausgeführt haben. Mit diesen Befehlen haben Sie die Dateien in Ihrem Verzeichnis zur Versionsverwaltung hinzugefügt. Der **git add** Befehl akzeptiert einen Pfadnamen einer Datei oder eines Verzeichnisses. Wenn Sie ein Verzeichnis angeben, fügt **git add** alle Dateien in diesem Verzeichnis und allen Unterverzeichnissen rekursiv hinzu.

## Geänderte Dateien zur Staging-Area hinzufügen

Lassen Sie uns jetzt eine bereits versionierte Datei ändern. Wenn Sie zum Beispiel eine bereits unter Versionsverwaltung stehende Datei mit dem Dateinamen **CONTRIBUTING.md** ändern und danach den Befehl **git status** erneut ausführen, erhalten Sie 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 „Changed but 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, führen Sie 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 hinzufügt werden. Wie Sie aber gerade gelernt haben, wird der Befehl auch noch für viele andere Dinge eingesetzt. Wenn Sie den Befehl **git add** einsetzen, sollten Sie das eher so sehen, dass Sie damit einen bestimmten Inhalt für den nächsten Commit vormerken. Lassen Sie 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, Sie wollen jetzt aber noch eine weitere Änderung an der Datei **CONTRIBUTING.md** vornehmen, bevor Sie den Commit tatsächlich starten. Sie öffnen die Datei erneut, ändern sie entsprechend ab und eigentlich wären Sie ja jetzt bereit den Commit durchzuführen. Allerdings lassen Sie 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 Sie den Befehl `git add` ausführen. Wenn Sie den Commit jetzt anlegen, wird die Version der Datei `CONTRIBUTING.md` denjenigen Inhalt haben, den sie hatte, als Sie `git add` zuletzt ausgeführt haben – und nicht denjenigen, den sie in dem Moment hat, wenn Sie den Befehl `git commit` ausführen. Wenn Sie stattdessen die gegenwärtige Version im Commit haben möchten, müssen Sie 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 Sie Ihre Änderungen kompakter sehen können. Wenn Sie `git status -s` oder `git status --short` ausführen, erhalten Sie 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, haben **??** neben sich, neue Dateien, die dem 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 Verzeichnisbaums. 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 Ihrem Build-System erzeugt werden. In solchen Fällen können Sie 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 der Codegenerierung sein könnten. Die zweite Zeile weist Git an, alle Dateien zu ignorieren, deren Name mit einer Tilde (**~**) enden, was von vielen Texteditoren wie Emacs zum Markieren temporärer Dateien verwendet wird. Sie können auch ein Verzeichnis log, tmp oder pid hinzufügen, eine automatisch generierte Dokumentation usw. Es ist im Allgemeinen eine gute Idee, die **.gitignore** Datei für Ihr neues Repository einzurichten, noch bevor Sie loslegen. So können Sie nicht versehentlich Dateien committen, die Sie wirklich nicht in Ihrem Git-Repository haben möchten.

Die Richtlinien für Vergleichsmuster, die Sie in der Datei **.gitignore** eingeben können, lauten wie folgt:

- Leerzeilen oder Zeilen, die mit **#** beginnen, werden ignoriert.
- Standard Platzhalter-Zeichen funktionieren und werden rekursiv im gesamten Verzeichnisbaum angewendet.
- Sie können Vergleichsmuster mit einem Schrägstrich **(/)** beginnen, um die Rekursivität zu verhindern.
- Sie können Vergleichsmuster mit einem Schrägstrich **(/)** beenden, um ein Verzeichnis anzugeben.

- Sie können ein Vergleichsmuster verbieten, indem Sie es mit einem Ausrufezeichen (!) beginnen.

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). Sie können auch zwei Sterne verwenden, um verschachtelte Verzeichnisse abzulegen; a/\*\*/z würde zu a/z, a/b/z, a/b/c/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 Sie einen Ansatzpunkt für Ihr Projekt suchen.



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` für Sie zu vage ist – Sie wollen genau wissen, was Sie geändert haben, nicht nur welche Dateien geändert wurden – können Sie den Befehl `git diff` verwenden. Wir werden `git diff` später ausführlicher behandeln, aber Sie werden es wahrscheinlich am häufigsten verwenden, um diese beiden Fragen zu beantworten: Was hat sich geändert, ist aber noch nicht zum Commit vorgemerkt (engl. staged)? Und was haben Sie zum Commit vorgemerkt und können es demnächst committen? Der Befehl `git status` beantwortet diese Fragen ganz allgemein, indem er die Dateinamen auflistet, `git diff` zeigt Ihnen aber genau die hinzugefügten und entfernten Zeilen – sozusagen den Patch.

Nehmen wir an, Sie bearbeiten und merken die Datei `README` zum Commit vor (eng. stage) und bearbeiten dann die Datei `CONTRIBUTING.md`, ohne sie zu „stagen“. Wenn Sie den Befehl `git status` ausführen, sehen Sie erneut so etwas wie das hier:

```
$ 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 Sie noch nicht zum Commit vorgemerkt haben, geben Sie 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 Ihrem Arbeitsverzeichnis befindet, mit dem, was sich in Ihrer Staging-Area befindet. Die Bildschirmanzeige gibt Ihnen an, welche Änderungen Sie vorgenommen haben, die noch nicht „gestaged“ sind.

Wenn Sie wissen wollen, was Sie zum Commit vorgemerkt haben, das in Ihren nächsten Commit einfließt, können Sie `git diff --staged` verwenden. Dieser Befehl vergleicht Ihre zum Commit vorgemerkten Änderungen mit Ihrem 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 Ihrem letzten Commit anzeigt – nur die Änderungen, die noch „unstaged“ sind. Wenn Sie alle Ihre Änderungen bereits „gestaged“ haben, wird `git diff` Ihnen keine Antwort geben.

Ein weiteres Beispiel: wenn Sie die Datei `CONTRIBUTING.md` zum Commit vormerken und dann wieder bearbeiten, können Sie mit `git diff` die Änderungen in der „staged“ Datei und die „unstaged“ Änderungen sehen. Wenn Sie folgendes gemacht haben:

```
$ 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
```

Jetzt können Sie 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 Ihnen, was Sie bisher zum Commit vorgemerkt haben (`--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, wenn Sie lieber ein graphisches oder externes Diff-Viewing-Programm bevorzugen. Wenn Sie `git difftool` anstelle von `git diff` verwenden, können Sie alle diese Unterschiede in einer Software wie emerge, vimdiff und viele andere (einschließlich kommerzieller Produkte) anzeigen lassen. Führen Sie den Befehl `git difftool --tool-help` aus, um zu sehen, was auf Ihrem System verfügbar ist.



## Die Änderungen committen

Nachdem Ihre Staging-Area nun so eingerichtet ist, wie Sie es wünschen, können Sie Ihre Änderungen committen. Denken Sie daran, dass alles, was noch nicht zum Commit vorgemerkt ist – alle Dateien, die Sie erstellt oder geändert haben und für die Sie seit Ihrer Bearbeitung nicht mehr `git add` ausgeführt haben – nicht in diesen Commit einfließen werden. Sie bleiben aber als geänderte Dateien auf Ihrer Festplatte erhalten. In diesem Beispiel nehmen wir an, dass Sie beim

letzten Mal, als Sie `git status` ausgeführt haben, gesehen haben, dass alles zum Commit vorgemerkt wurde und bereit sind, Ihre Änderungen zu committen. Am einfachsten ist es, wenn Sie `git commit` eingeben:

```
$ git commit
```

Dadurch wird der Editor Ihrer Wahl gestartet. (Das wird durch die Umgebungsvariable `EDITOR` Ihrer Shell festgelegt – normalerweise vim oder emacs, wie Sie es mit dem Befehl `git config --global core.editor` beliebig konfigurieren können und wie Sie es in Kapitel 1, [Erste Schritte](#), gesehen haben).

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
```

Sie können erkennen, dass die Standard-Commit-Meldung die neueste Ausgabe des auskommentierten Befehls `git status` und eine leere Zeile darüber enthält. Sie können diese Kommentare entfernen und Ihre Commit-Nachricht eingeben oder Sie können sie dort stehen lassen, damit Sie sich merken können, was Sie committen. (Für eine noch bessere Gedächtnisstütze über das, was Sie geändert haben, können Sie die Option `-v` an `git commit` übergeben. Dadurch wird auch die Differenz Ihrer Änderung in den Editor geschrieben, so dass Sie genau sehen können, welche Änderungen Sie committen.) Wenn Sie den Editor verlassen, erstellt Git Ihren Commit mit dieser Commit-Nachricht (mit den Kommentaren und ausgeblendetem diff).

Alternativ können Sie Ihre Commit-Nachricht auch inline mit dem Befehl `commit -m` eingeben. Das `-m` Flag ermöglicht die direkte Eingabe eines Kommentar-Textes:

```
$ 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 haben Sie Ihren ersten Commit erstellt! Sie können sehen, dass der Commit eine Nachricht über sich selbst ausgegeben hat: in welchen Branch Sie committed haben (`master`), welche SHA-1-Prüfsumme der Commit hat (`463dc4f`), wie viele Dateien geändert wurden, und Statistiken über

hinzugefügte und entfernte Zeilen im Commit.

Denken Sie daran, dass der Commit den Snapshot aufzeichnet, den Sie in Ihrer Staging-Area eingerichtet haben. Alles, was von Ihnen nicht zum Commit vorgemerkt wurde, liegt immer noch als modifiziert da. Sie können einen weiteren Commit durchführen, um es zu Ihrer Historie hinzuzufügen. Jedes Mal, wenn Sie einen Commit ausführen, zeichnen Sie einen Schnappschuss Ihres Projekts auf, auf den Sie zurückgreifen oder mit einem späteren Zeitpunkt vergleichen können.

## Die Staging Area überspringen

Obwohl es außerordentlich nützlich sein kann, Commits so zu erstellen, wie Sie es wünschen, ist die Staging-Area manchmal etwas komplexer, als Sie es für Ihren Workflow benötigen. Wenn Sie 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 vorgemerkt (engl. staged), so dass Sie den Teil `git add` überspringen können:

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

    modified:   CONTRIBUTING.md

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

Beachten Sie, dass Sie in diesem Fall `git add` nicht auf der Datei `CONTRIBUTING.md` ausführen müssen, bevor Sie committen. Das liegt daran, dass das `-a` Flag alle geänderten Dateien einschließt. Das ist bequem, aber seien Sie vorsichtig. Manchmal führt dieses Flag dazu, dass Sie ungewollte Änderungen vornehmen.

## Dateien löschen

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

Wenn Sie die Datei einfach aus Ihrem Arbeitsverzeichnis entfernen, erscheint sie unter dem „Changes not staged for commit“ Bereich (das ist die *unstaged*-Area) Ihrer `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 Sie dann `git rm` ausführen, 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 Sie das nächste Mal einen Commit ausführen, ist die Datei weg und ist nicht mehr versioniert (engl. tracked). Wenn Sie die Datei geändert oder bereits zur Staging-Area hinzugefügt haben, müssen Sie 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 Sie möglicherweise nutzen möchten, ist, die Datei in Ihrem Verzeichnisbaum zu behalten, sie aber aus Ihrer Staging-Area zu entfernen. Mit anderen Worten, Sie können die Datei auf Ihrer Festplatte behalten, aber nicht mehr von Git protokollieren/versionieren lassen. Das ist besonders dann nützlich, wenn Sie vergessen haben, etwas zu Ihrer `.gitignore` Datei hinzuzufügen und diese versehentlich „gestaged“ haben, wie eine große Logdatei oder eine Reihe von `.a` kompilierten Dateien. Das erreichen Sie mit der Option `--cached`:

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

Sie können Dateien, Verzeichnisse und Platzhalter-Zeichen an den Befehl `git rm` übergeben. Das bedeutet, dass Sie folgende Möglichkeit haben:

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

Beachten Sie den Backslash (\) vor dem \*. Der ist notwendig, weil Git zusätzlich zur Dateinamen-Erweiterung Ihrer Shell eine eigene Dateinamen-Erweiterung vornimmt. Mit dem Befehl oben werden alle Dateien entfernt, die die Erweiterung .log im Verzeichnis log/ haben. oder, Sie können etwas Ähnliches ausführen:

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

Das Kommando entfernt alle Dateien, deren Name mit einer ~ endet.

## Dateien verschieben

Im Gegensatz zu vielen anderen VCS-Systemen verfolgt (engl. track) Git das Verschieben von Dateien nicht ausdrücklich. Wenn Sie eine Datei in Git umbenennen, werden keine Metadaten in Git gespeichert, die dem System mitteilen, dass Sie die Datei umbenannt haben. 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 Sie eine Datei in Git umbenennen möchten, dann können Sie beispielsweise Folgendes ausführen:

```
$ git mv file_from file_to
```

Das funktioniert gut. Tatsache ist, wenn Sie so einen Befehl ausführen und sich den Status ansehen, werden Sie 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 Sie eine Datei auf diese Weise oder mit dem Befehl mv umbenennen. Der alleinige, reale Unterschied ist, dass git mv ein einziger Befehl ist statt deren drei – es ist eine Komfortfunktion. Wichtiger ist, dass Sie jedes beliebige Tool verwenden können, um eine Datei umzubenennen und das add/rm später aufrufen können, bevor Sie committen.

# Anzeigen der Commit-Historie

Nachdem Sie mehrere Commits erstellt haben oder wenn Sie ein Repository mit einer bestehenden Commit-Historie geklont haben, werden Sie 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 vereinfachtes Projekt namens „simplegit“. Um das Projekt zu erstellen, führen Sie diesen Befehl aus:

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

Wenn Sie `git log` in diesem Projekt aufrufen, sollten Sie 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

    changed the version number

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

    removed unnecessary test

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

    first commit
```

Standardmäßig listet `git log`, ohne Argumente, die in diesem Repository vorgenommenen Commits in umgekehrter chronologischer Reihenfolge auf, d.h. die neuesten Commits werden als erstes angezeigt. Wie Sie sehen können, 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 und Vielfalt von Optionen für den Befehl `git log` stehen zur Verfügung, um Ihnen exakt das anzuseigen, wonach Sie gesucht haben. Hier zeigen wir Ihnen einige der gängigsten.

Eine der hilfreichsten Optionen ist `-p` oder `--patch`. Sie zeigt den Unterschied (die *patch*-Ausgabe) an, der bei jedem Commit eingefügt wird. Sie können auch die Anzahl der anzuseigenden Protokolleinträge begrenzen, z.B. mit `-2` nur die letzten beiden Einträge darstellen.

```

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

    changed the version number

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

```

```

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

```

```

    removed unnecessary test

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

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

```

Diese Option zeigt die gleichen Informationen an, jedoch mit dem Unterschied dass sie direkt hinter jedem Eintrag stehen. Diese Funktion ist sehr hilfreich für die Code-Überprüfung oder zum schnellen Durchsuchen der Vorgänge während einer Reihe von Commits, die ein Teammitglied hinzugefügt hat. Sie können auch eine Reihe von Optionen zur Verdichtung mit `git log` verwenden. Wenn Sie beispielsweise einige gekürzte Statistiken für jede Übertragung sehen möchten, können Sie 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

    changed the version number

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

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

    removed unnecessary test

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

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

    first commit

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

```

Wie Sie sehen können, 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. Ihnen stehen einige vorgefertigte Optionen zur Verfügung. Die Option `oneline` druckt jeden Commit in einer einzigen Zeile, was besonders nützlich ist, wenn Sie sich viele Commits ansehen. Darüber hinaus zeigen die Optionen `short`, `full`, und `fuller` die Ausgabe im etwa gleichen Format, allerdings mit weniger bzw. mehr Informationen:

```

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

```

Die interessanteste Option ist `format`, mit der Sie Ihr eigenes Log-Ausgabeformat festlegen können. Dieses Verfahren ist besonders nützlich, wenn Sie Ausgaben für das maschinelle Parsen generieren – da Sie das Format explizit angeben, wissen Sie, dass es sich mit Updates von Git nicht ändert:

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

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

Table 1. Nützliche Optionen für `git log --pretty=format`

Option	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>	Eltern-Hashes
<code>%p</code>	gekürzte Eltern-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

Sie fragen sich 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 Sie also einen Patch an ein Projekt senden und eines der Core-Mitglieder den Patch einbindet, erhalten Sie beide die Anerkennung – Sie als Autor und das Core-Mitglied als Committer. Wir werden diese Unterscheidung näher erläutern in Kapitel 5, [Verteiltes Git](#).

Die Optionen `oneline` und `format` sind vor allem bei einer anderen `log`-Option mit Bezeichnung `--graph` hilfreich. Diese Option fügt ein schönes kleines ASCII-Diagramm hinzu, das Ihren 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 git://github.com/dustin/grit
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
* 11d191e Merge branch 'defunkt' into local
```

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 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 nur der geänderten/eingefügten/gelöschten Zeile des Befehls <code>--stat</code> .
<code>--name-only</code>	Listet die Dateien auf, die nach den Commit-Informationen geändert wurden.
<code>--name-status</code>	Listet die Dateien auf, die von hinzugefügten, geänderten oder gelöschten Informationen betroffen sind.
<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 ein ASCII-Diagramm der Branch an und verbindet die Historie mit der Log-Ausgabe.
<code>--pretty</code>	Zeigt Commits in einem anderen Format an. Zu den Optionen gehören <code>oneline</code> , <code>short</code> , <code>full</code> , <code>fuller</code> und <code>format</code> (womit Sie Ihr eigenes Format angeben können).
<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 Sie nur eine Teilmenge von Commits anzeigen können. Sie haben eine solche Option bereits gesehen – die Option `-2`, die nur die letzten beiden Commits anzeigt. In Wahrheit können Sie `-<n>` verwenden, wobei `n` eine beliebige ganze Zahl ist,

um die letzten `n` Commits anzuzeigen. In der Praxis werden Sie das kaum verwenden, da Git standardmäßig alle Ausgaben über einen Pager leitet, so dass Sie jeweils nur eine Seite der Log-Ausgabe sehen.

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

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

Dieser Befehl funktioniert mit vielen Formaten – Sie können ein bestimmtes Datum wie „`2008-01-15`“ angeben, oder ein relatives Datum wie „`vor 2 Jahren 1 Tag 3 Minuten`“.

Sie können die Liste auch nach Commits filtern, die bestimmten Suchkriterien entsprechen. Mit der Option `--author` können Sie nach einem bestimmten Autoren filtern, und mit der Option `--grep` können Sie nach Schlüsselwörtern in den Übertragungsmeldungen suchen.



Sie können 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 übernimmt und nur die Commits anzeigt, die die Anzahl der Vorkommen dieses Strings geändert haben. Wenn Sie beispielsweise das letzte Commit suchen möchten, das einen Verweis auf eine bestimmte Funktion hinzugefügt oder entfernt hat, können Sie Folgendes aufrufen:

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

Zuletzt eine wirklich nützliche Option, die Sie als Filter an `git log` übergeben können, der Pfad. Wenn Sie ein Verzeichnis oder einen Dateinamen angeben, können Sie 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.

In [Optionen zur Anpassen der Ausgabe von `git log`](#) werden wir Ihnen diese und einige andere gängige Optionen als Referenz auflisten.

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

Option	Beschreibung
<code>-&lt;n&gt;</code>	Zeigt nur die letzten <code>n</code> 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.

Option	Beschreibung
--author	Zeigt nur Commits an, bei denen der Autoren-Eintrag mit der angegebenen Zeichenkette übereinstimmt.
--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 Sie zum Beispiel sehen möchten, welche der Commits, die Testdateien in der Git-Quellcode-Historie ändern, die von Junio Hamano im Monat Oktober 2008 committed wurden und keine Merge-Commits sind, können Sie etwa so 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 Ihrem Repository verwendeten Workflow ist es möglich, dass ein beträchtlicher Prozentsatz der Commits in Ihrer Log-Historie nur Merge-Commits sind, die in der Regel nicht sehr informativ sind. Um zu vermeiden, dass die Anzeige von Merge-Commits Ihren Log-Verlauf überflutet, fügen Sie einfach die Log-Option **--no-merges** hinzu.

## Ungewollte Änderungen rückgängig machen

Zu jeder Zeit können Sie eine Änderung rückgängig machen. Hier werden wir einige grundlegende Werkzeuge besprechen, die zum Widerrufen von gemachten Änderungen dienen. Seien Sie vorsichtig, denn man kann nicht immer alle diese Annulierungen rückgängig machen. Das ist einer der wenigen Bereiche in Git, in denen Sie etwas Arbeit verlieren könnten, wenn Sie etwas falsch machen.

Eines der häufigsten Undos tritt auf, wenn Sie zu früh committen und möglicherweise vergessen, einige Dateien hinzuzufügen, oder wenn Sie Ihre Commit-Nachricht durcheinander bringen. Wenn Sie den Commit erneut ausführen möchten, nehmen Sie die zusätzlichen Änderungen vor, die Sie vergessen haben, stellen Sie sie bereit (engl. stage) und committen Sie erneut mit der Option

--amend:

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

Dieser Befehl übernimmt Ihre Staging Area und verwendet sie für den Commit. Wenn Sie seit Ihrem letzten Commit keine Änderungen vorgenommen haben (z.B. Sie führen diesen Befehl unmittelbar nach Ihrem vorherigen Commit aus), dann sieht Ihr Snapshot genau gleich aus, Sie ändern nur Ihre Commit-Nachricht.

Der gleiche Commit-Message-Editor wird aufgerufen, enthält aber bereits die Nachricht Ihres vorherigen Commits. Sie können die Nachricht wie gewohnt bearbeiten, aber sie überschreibt den vorherigen Commit.

Wenn Sie zum Beispiel die Änderungen in einer Datei, die Sie zu dieser Übertragung hinzufügen wollten, vergessen haben, können Sie etwas Ähnliches durchführen:

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

Sie erhalten am Ende einen einzigen Commit – der zweite Commit ersetzt die Ergebnisse des ersten.

Es ist wichtig zu verstehen, dass, wenn Sie Ihren letzten Commit ändern, Sie ihn weniger reparieren, als ihn komplett durch einen neuen, verbesserten Commit *ersetzen*. Der alte Commit wird aus dem Weg geräumt und der neue Commit an seine Stelle gesetzt. Tatsächlich ist es so, als ob der letzte Commit nie stattgefunden hätte, der nicht in Ihrem Repository-Verlauf auftauchen wird.



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

## Eine Datei aus der Staging Area entnehmen

Die nächsten beiden Abschnitte erläutern, wie Sie mit Ihrer Staging Area und den Änderungen des Arbeitsverzeichnisses arbeiten. Der angenehme Nebeneffekt ist, dass der Befehl, mit dem Sie den Zustand dieser beiden Bereiche bestimmen, Sie auch daran erinnert, wie Sie Änderungen an ihnen rückgängig machen können. Nehmen wir zum Beispiel an, Sie haben zwei Dateien geändert und möchten sie als zwei separate Änderungen übertragen, aber Sie geben versehentlich `git add *` ein und stellen sie dann beide in der Staging Area bereit. Wie können Sie 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 „Changes to be committed“ Text, steht, dass man `git reset HEAD <file>...` verwenden soll, um die Staging Area zu entleeren. Lassen Sie 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` wurde modifiziert, aber erneut unstaged.



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

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

## Änderung in einer modifizierten Datei zurücknehmen

Was ist, wenn Sie feststellen, dass Sie Ihre Änderungen an der Datei `CONTRIBUTING.md` nicht behalten wollen? Wie können Sie sie leicht wieder ändern – sie wieder so zurücksetzen, wie sie beim letzten Commit ausgesehen hat (oder anfänglich geklont wurde, oder wie auch immer Sie sie in Ihr Arbeitsverzeichnis bekommen haben)? Glücklicherweise sagt Ihnen `git status` auch, wie Sie das machen können. 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
```

Es erklärt Ihnen ziemlich eindeutig, wie Sie die von Ihnen vorgenommenen Änderungen verwerfen können. Lasst Sie uns machen, was da steht:

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

Sie können erkennen, dass die Änderungen rückgängig gemacht wurden.



Es ist sehr wichtig zu begreifen, dass `git checkout -- <file>` ein riskanter Befehl ist. Alle lokalen Änderungen, die Sie an dieser Datei vorgenommen haben, sind hinfällig – Git hat diese Datei einfach durch die zuletzt committete Version ersetzt. Verwenden Sie diesen Befehl niemals, es sei denn, Sie sind sich absolut sicher, dass Sie diese ungesicherten lokalen Änderungen nicht wünschen.

Wenn Sie die Änderungen, die Sie an dieser Datei gemacht haben, beibehalten möchten, sie aber vorerst noch nicht vornehmen möchten, sollten wir das Stashing und Branching in Kapitel 3 – [Git Branching](#) durchgehen; das sind im Allgemeinen die besseren Methoden, um das zu erledigen.

Denken Sie daran, dass alles, was in Git *committed* 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). Allerdings wird alles, was Sie verloren haben und das nie committed wurde, wahrscheinlich nie wieder gesehen werden.

## Mit Remotes arbeiten

Um an jedem Git-Projekt mitarbeiten zu können, müssen Sie wissen, wie Sie Ihre Remote-Repositorys verwalten können. Remote-Repositorys sind Versionen Ihres Projekts, die im Internet oder im Netzwerk irgendwo gehostet werden. Sie können mehrere einrichten, von denen jedes in der Regel entweder schreibgeschützt oder beschreibbar ist. Die Zusammenarbeit mit anderen erfordert die Verwaltung dieser Remote-Repositorys und das Pushing und Pulling von Daten zu und von den Repositorys, wenn Sie Ihre Arbeit teilen möchten. Die Verwaltung von Remote-Repositorys umfasst das Wissen, wie man entfernte Repositorys hinzufügt, nicht mehr gültige Remotes entfernt, verschiedene Remote-Banches verwaltet und 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 Ihrem lokalen Rechner liegen.*

Es ist durchaus möglich, dass Sie mit einem „entfernten“ Repository arbeiten können, das sich tatsächlich auf demselben Host befindet auf dem Sie gerade arbeiten. 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 Sie konfiguriert haben, können Sie den Befehl `git remote` aufrufen. Es listet die Kurznamen der einzelnen von Ihnen festgelegten Remote-Handles auf. Wenn Sie Ihr Repository geklont haben, sollten Sie zumindest `origin` sehen – das ist der Standardname, den Git dem Server gibt, von dem Sie geklont haben:

```
$ 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
```

Sie können zusätzlich auch `-v` angeben, das Ihnen die URLs anzeigen, die Git für den Kurznamen gespeichert hat, der beim Lesen und Schreiben auf diesem Remote verwendet werden soll:

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

Wenn Sie mehr als einen Remote haben, 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.

Beachten Sie, dass diese Remotes eine Vielzahl von Protokollen verwenden; wir werden mehr darüber erfahren, wenn wir [Git auf einem Server installieren](#).

## Hinzufügen von Remote-Repositories

Wir haben bereits erwähnt und einige Beispiele gezeigt, wie der Befehl `git clone` stillschweigend das `origin` Remote für Sie hinzufügt. So können Sie explizit einen neuen Remote hinzufügen. Um ein neues Remote-Git-Repository als Kurzname hinzuzufügen, auf das Sie leicht verweisen können, führen Sie `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)
```

Jetzt können Sie die Zeichenfolge `pb` auf der Kommandozeile anstelle der gesamten URL verwenden. Wenn Sie beispielsweise alle Informationen abrufen möchten, die Paul hat, die aber noch nicht in Ihrem Repository enthalten sind, können Sie `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 – Sie können ihn in eine Ihrer Branches einbinden, oder Sie können an dieser Stelle in einen lokalen Branch wechseln (engl. `checkout`), wenn Sie ihn inspizieren möchten. (Wir werden in [Git Branching](#) näher darauf eingehen, was Branches sind und wie man sie viel präziser nutzen kann.)

## Fetching und Pulling von Ihren Remotes

Wie Sie gerade gesehen haben, können Sie Daten aus Ihren Remote-Projekten abrufen:

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

Der Befehl geht an das Remote-Projekt und zieht (engl. `pull`) alle Daten von diesem Remote-Projekt runter, die Sie noch nicht haben. Danach sollten Sie Referenzen auf alle Branches von diesem Remote haben, die Sie jederzeit einbinden oder inspizieren können.

Wenn Sie ein Repository klonen, fügt der Befehl dieses entfernte Repository automatisch unter dem Namen „origin“ hinzu. So holt `git fetch origin` alle neuen Inhalte, die seit dem Klonen (oder dem letzten Abholen) auf diesen Server verschoben wurden. Es ist jedoch wichtig zu beachten, dass der Befehl `git fetch` nur die Daten in Ihr lokales Repository herunterlädt – er mischt (engl. `merged`) sie nicht automatisch mit Ihrer Arbeit zusammen oder ändert das, woran Sie gerade arbeiten. Sie müssen das Ganze manuell mit Ihrer Arbeit zusammenführen, wenn Sie fertig sind.

Wenn Ihr aktueller Branch so eingerichtet ist, dass er einen entfernten Branch verfolgt (engl. `tracking`), können Sie den Befehl `git pull` verwenden, um diesen entfernten Branch automatisch zu holen und dann mit Ihrem aktuellen Branch zusammenzuführen (siehe den nächsten Abschnitt und [Git Branching](#) für weitere Informationen). Das könnte ein einfacherer oder komfortablerer Workflow für Sie sein. Standardmäßig richtet der Befehl `git clone` Ihren 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 Sie geklont haben. Wenn Sie `git pull` ausführen, werden normalerweise Daten von dem Server abgerufen, von dem Sie ursprünglich geklont haben, und es wird automatisch versucht, sie in den Code zu mergen, an dem Sie gerade arbeiten.

## Pushing zu Ihren Remotes

Wenn Sie Ihr Projekt an einem bestimmten Punkt haben, den Sie teilen möchten, müssen Sie es zum Upstream verschieben (engl. `pushen`). Der Befehl dafür ist einfach: `git push <remote> <bbranch>`. Wenn Sie Ihren Master-Branch auf Ihren `origin` Server verschieben möchten (nochmals, das

Klonen richtet im Regelfall beide dieser Namen automatisch für Sie ein), dann können Sie diesen Befehl auch nutzen, um alle Commits, die Sie durchgeführt haben, auf den Server zu übertragen:

```
$ git push origin master
```

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

## Inspizieren eines Remotes

Wenn Sie mehr Informationen über einen bestimmten Remote sehen möchten, können Sie den Befehl `git remote show <remote>` verwenden. Wenn Sie diesen Befehl mit einem spezifischen Kurznamen ausführen, wie z.B. `origin`, erhalten Sie eine ähnliche 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-Branchen auf. Der hilfreiche Befehl teilt Ihnen mit, dass, wenn Sie sich im Master-Branch befinden und falls Sie `git pull` ausführen, dieser automatisch im Master-Zweig des Remote gemergt wird, nachdem er alle Remote-Referenzen abgerufen (engl. fetched) hat. Er listet auch alle Remote-Referenzen auf, die er abgerufen hat.

Das ist nur ein einfaches Beispiel, auf das Sie vermutlich treffen werden. Wenn Sie Git hingegen intensiver verwenden, können Sie viel mehr Informationen aus `git remote show` herauslesen:

```

$ 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 Zweig automatisch gepusht wird, wenn Sie `git push` ausführen, während Sie sich in bestimmten Branches befinden. Er zeigt Ihnen auch, welche entfernten Branches auf dem Server sind, die Sie noch nicht haben, welche entfernten Branches Sie haben, die aber vom Server entfernt wurden und die lokalen Branches, die automatisch mit ihrem Remote-Tracking-Branch mergen können, wenn Sie `git pull` ausführen.

## Umbenennen und entfernen von Remotes

Sie können `git remote rename` ausführen, um den Kurznamen einer Fernbedienung zu ändern. Wenn Sie beispielsweise `pb` in `paul` umbenennen möchten, können Sie das mit `git remote rename` machen:

```

$ git remote rename pb paul
$ git remote
origin
paul

```

Es ist zu beachten, dass dadurch auch alle Ihre Remote-Tracking-Branchnamen geändert werden. Was früher mit `pb/master` angesprochen wurde, ist jetzt `paul/master`.

Wenn Sie einen Remote aus irgendwelchen Gründen entfernen möchten – Sie haben den Server verschoben oder verwenden einen bestimmten Mirror nicht länger oder ein Beitragender ist nicht mehr dabei – dann können Sie entweder `git remote remove` oder `git remote rm` verwenden:

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

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

## Tagging

Wie die meisten VCSs hat Git die Möglichkeit, bestimmte Punkte in der Historie eines Repositorys als wichtig zu markieren. Normalerweise verwenden Menschen diese Funktionalität, um Releases zu markieren (**v1.0**, **v2.0** usw). In diesem Abschnitt erfahren Sie, wie Sie bestehende Tags auflisten, Tags erstellen und löschen können und was die unterschiedlichen Tag-Typen sind.

### Ihre Tags auflisten

Die Auflistung der vorhandenen Tags in Git ist einfach. Geben Sie 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.

Sie können auch nach Tags suchen, die einer bestimmten Zeichenfolge entsprechen. Das Git Source Repo, zum Beispiel, enthält mehr als 500 Tags. Wenn Sie nur daran interessiert sind, sich die 1.8.5-Serie anzusehen, können Sie 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 Sie lediglich die gesamte Liste der Tags wünschen, geht die Ausführung des Befehls `git tag` implizit davon aus, dass Sie eine Auflistung haben wollen und gibt sie aus; die Verwendung von `-l` oder `--list` ist in diesem Fall optional.

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

## Erstellen von Tags

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

Ein nicht annotiertes Tag ist sehr ähnlich einer Branch, 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 und das Datum, haben eine Tagging-Meldung und können mit GNU Privacy Guard (GPG) signiert und überprüft werden. Es wird allgemein empfohlen, dass Sie annotierte Tags erstellen, damit Sie all diese Informationen erhalten können; aber wenn Sie ein temporäres Tag wünschen oder aus irgendwelchen Gründen die anderen Informationen nicht speichern wollen, sind auch nicht-annotierte Tags möglich.

## Annotierte Tags

Das Erstellen eines annotierten Tags in Git ist einfach. Der einfachste Weg ist die Eingabe von `-a`, wenn Sie den Befehl `tag` ausführen:

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

Ein `-m` spezifiziert eine Tagging-Meldung, die mit dem Tag gespeichert wird. Wenn Sie keine Meldung für ein kommentiertes Tag angeben, startet Git Ihren Editor, damit Sie diese eingeben können.

Sie können die Tag-Daten zusammen mit dem Commit, der mit dem Befehl `git show` getaggt wurde, einsehen:



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

my version 1.4

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

    changed the version number
```

Das zeigt die Tagger-Informationen, das Datum, an dem der Commit getaggt wurde, und die Annotationsmeldung an, bevor die Commit-Informationen angezeigt werden.

## 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, geben Sie 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 Sie diesmal `git show` auf dem Tag ausführen, sehen Sie 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

    changed the version number
```

## Nachträgliches Tagging

Sie können auch Commits markieren, wenn Sie sich bereits an einem anderen Punkt befinden. Angenommen, Ihr Commit-Verlauf sieht so aus:

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

Nehmen wir an, Sie haben vergessen, das Projekt mit v1.2 zu markieren, das bei dem Commit von „updated rakefile“ vorlag. Sie können ihn nachträglich hinzufügen. Um diesen Commit zu markieren, geben Sie am Ende des Befehls die Commit-Prüfsumme (oder einen Teil davon) an:

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

Sie sehen, dass Sie den Commit markiert haben:

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

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

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

    updated rakefile
...
```

## Tags freigeben

Normalerweise überträgt der Befehl `git push` keine Tags an Remote-Server. Sie müssen Tags explizit auf einen freigegebenen Server verschieben, nachdem Sie sie erstellt haben. Dieser Prozess funktioniert genauso wie das Freigeben von Remote-Banches – Sie müssen 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 Sie eine Menge Tags haben, die Sie auf einmal pushen wollen, können Sie auch die Option `--tags` mit dem Befehl `git push` verwenden. Dadurch werden alle Ihre 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 Ihrem Repository klonnt oder pullt, erhält er auch alle Ihre Tags.

#### `git push` pusht beide Arten von Tags



Das Pushen von Tags mit `git push <remote> --tags` unterscheidet nicht zwischen Lightweight- und Annotated-Tags; es gibt keine einfache Option, die es Ihnen erlaubt, nur einen Typ zum Pushen auszuwählen.

## Tags löschen

Um einen Tag aus dem lokalen Repository zu löschen, verwenden Sie `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)
```

Beachten Sie, dass dadurch das Tag nicht von Remote-Servern entfernt wird. Es gibt zwei gängige Varianten, um ein Tag von einem entfernten 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
```

Die Lösung, das oben Gezeigte zu interpretieren, besteht darin, es als Nullwert zu lesen, bevor der Doppelpunkt auf den Remote-Tag-Namen verschoben wird, wodurch es effektiv gelöscht wird.

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

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

## Tags auschecken

Wenn Sie die Dateiversion anzeigen möchten, auf die ein bestimmter Tag zeigt, können Sie `git checkout` auf dieses Tag durchführen, obwohl dies Ihr Repository in den Zustand „detached HEAD“ (dt. losgelöst) versetzt, was einige negative Nebenwirkungen hat:

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

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

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

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

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

$ git checkout 2.0-beta-0.1
Previous HEAD position was 99ada87... Merge pull request #89 from schacon/appendix-final
HEAD is now at df3f601... add atlas.json and cover image
```

Wenn Sie im Zustand „getrennter HEAD“ Änderungen vornehmen und dann einen Commit erstellen, bleibt der Tag gleich, aber Ihr neuer Commit gehört zu keinem Branch und ist unzugänglich, außer mit dem genauen Commit-Hash. Wenn Sie also Änderungen vornehmen müssen – z.B. wenn Sie einen Fehler in einer älteren Version beheben – sollten Sie im Regelfall einen Branch erstellen:

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

Wenn Sie das tun und einen Commit erstellen, wird sich Ihr Zweig `version2` leicht von Ihrem Tag `v2.0.0.0` unterscheiden, da er mit Ihren neuen Änderungen fortschreitet, seien Sie also vorsichtig.

# Git Aliases

Bevor wir dieses Kapitel über Basic Git abschließen, gibt es noch einen kurzen Tipp, der Ihre Arbeit mit Git einfacher, leichter und verständlicher machen kann: Aliase. Wir werden nicht auf sie Bezug nehmen oder annehmen, dass Sie sie später im Buch verwenden, aber Sie sollten wohl wissen, wie man sie benutzt.

Git leitet Ihr Kommando nicht automatisch ab, wenn Sie es teilweise eingeben. Wenn Sie nicht den gesamten Text von jedem der Git-Befehle eingeben möchten, können Sie mit Hilfe von `git config` einfach einen Alias für jeden Befehl einrichten. Hier sind ein paar Beispiele, die Sie 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 Sie z.B. anstelle von `git commit` einfach `git ci` eingeben müssen. Wenn Sie Git nun weiter verwenden, werden Sie vermutlich auch andere Befehle häufig verwenden; scheuen Sie sich nicht, neue Aliase zu erstellen.

Diese Technik kann auch sehr nützlich sein, um Befehle zu erstellen, von denen Sie glauben, dass sie vorhanden sein sollten. Um beispielsweise ein Usability-Problem zu beheben, auf das Sie beim Entfernen einer Datei aus der Staging Area stoßen, können Sie Git Ihren 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 erscheint etwas 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 können Sie 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 Sie feststellen können, ersetzt Git einfach den neuen Befehl durch den Alias, für den Sie ihn verwenden. Vielleicht möchten Sie jedoch eher einen externen Befehl als einen Git-Subbefehl ausführen. In diesem Fall starten Sie den Befehl mit einem ! Zeichen. Das ist hilfreich, wenn Sie Ihre eigenen Tools schreiben, die mit einem Git-Repository arbeiten. Wir können durch Aliasing von `git visual` demonstrieren, um `gitk` auszuführen:

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

## Zusammenfassung

Sie sollten jetzt in der Lage sein, die wichtigsten Git Befehle einsetzen zu können – Folgendes sollte Ihnen jetzt gelingen: Erzeugen oder Klonen eines Repositorys, Änderungen vorzunehmen und zur Staging-Area hinzuzufügen, Commits anzulegen und die Historie aller Commits in einem Repository zu 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 Sie von der Hauptlinie der Entwicklung abzweigen und Ihre Arbeit fortsetzen, ohne an der Hauptlinie herumzubasteln. In vielen VCS-Tools ist das ein etwas aufwändiger Prozess, bei dem Sie oft eine neue Kopie Ihres Quellcode-Verzeichnisses erstellen müssen, 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, wodurch Branch-Operationen nahezu verzögerungsfrei ausgeführt werden und auch das Hin- und Herschalten zwischen einzelnen Entwicklungszweigen meistens genauso schnell abläuft. Im Gegensatz zu anderen VCS ermutigt Git zu einer Arbeitsweise mit häufigem Branching und Merging, sogar mehrmals am Tag. Wenn Sie diese Funktion verstehen und beherrschen, besitzen Sie ein mächtiges und einmaliges Werkzeug, welches Ihre Art zu entwickeln vollständig verändern kann.

## Branches auf einen Blick

Um richtig zu verstehen, wie Git das Verzweigen realisiert, müssen wir einen Schritt zurücktreten und untersuchen, wie Git seine Daten speichert.

Wie Sie vielleicht aus Kapitel 1 [Erste Schritte](#) in Erinnerung haben, speichert Git seine Daten nicht als Serie von Änderungen oder Unterschieden, sondern statt dessen als eine Reihe von *Snapshots*.

Wenn Sie einen Commit durchführen, speichert Git ein Commit-Objekt, das einen Zeiger auf den Snapshot des von Ihnen bereitgestellten Inhalts enthält. Dieses Objekt enthält auch den Namen und die E-Mail-Adresse des Autors, die Nachricht, die Sie eingegeben haben, 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, lassen Sie uns annehmen, Sie haben ein Verzeichnis, welches drei Dateien enthält, und Sie fügen alle Dateien zur Staging-Area hinzu und führen 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 [Erste Schritte](#) 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 'The initial commit of my project'
```

Wenn Sie mit der Anweisung `git commit` einen Commit erzeugen, 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.

Ihr 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 Root des Projektbaumes und die Metadaten des Commits verweist.

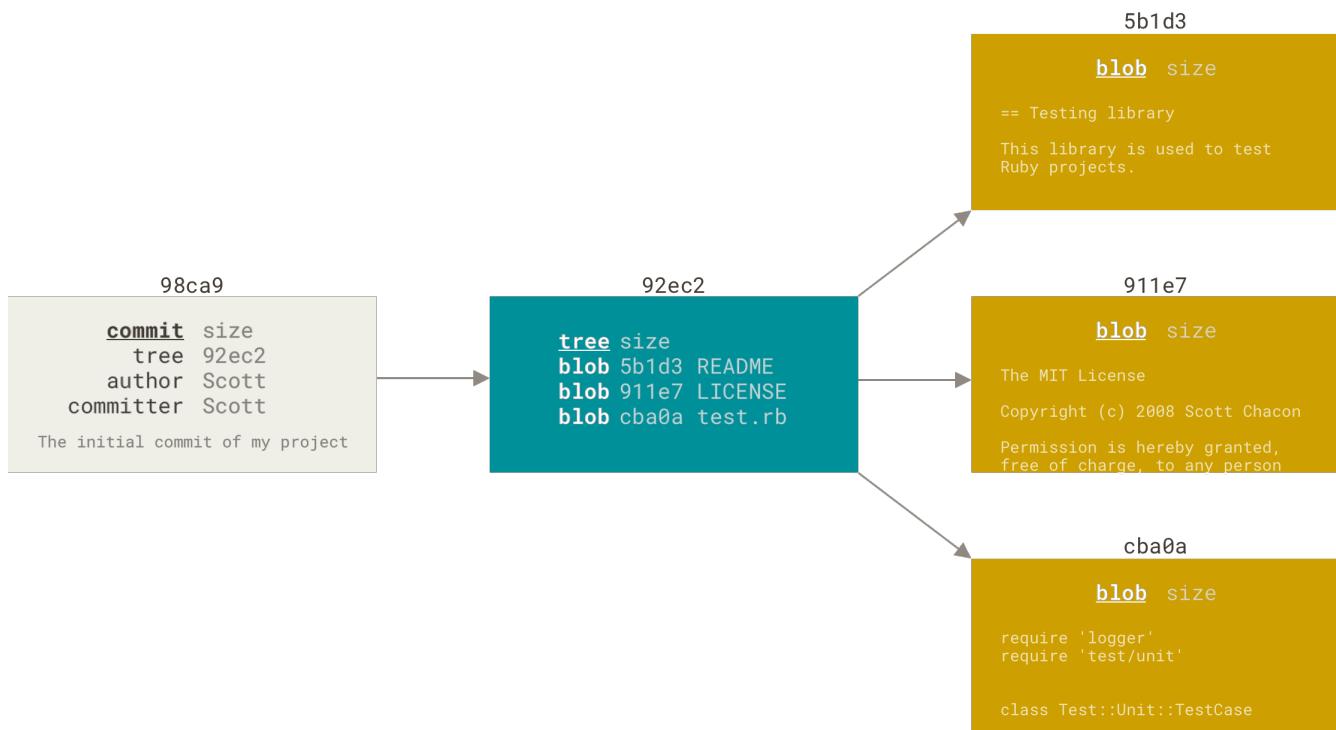


Figure 9. Ein Commit und sein Tree

Wenn Sie einige Änderungen vornehmen und wieder einen Commit durchführen, speichert dieser einen Zeiger zu dem Commit, der unmittelbar davor gemacht wurde.

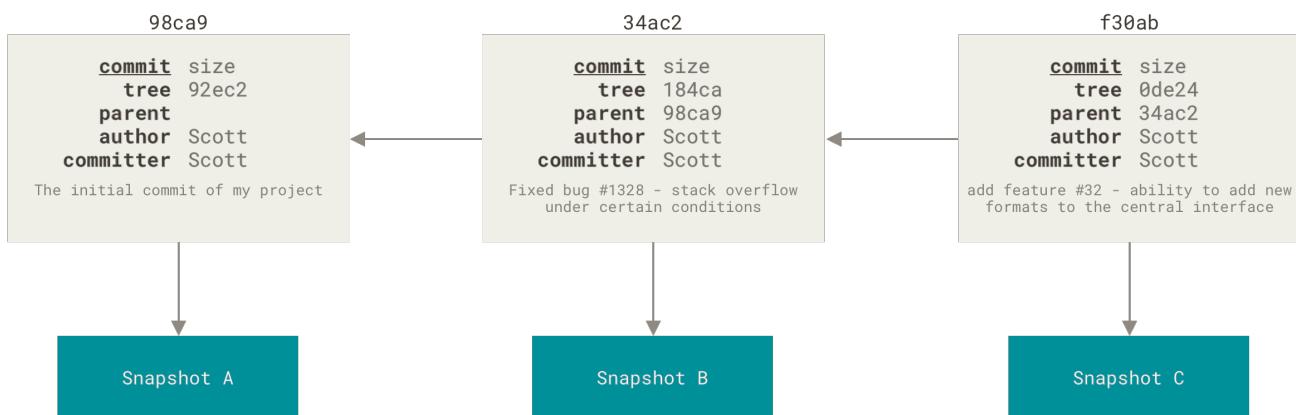


Figure 10. Commits und ihre Vorgänger

Ein Branch in Git ist einfach ein leichter, beweglicher Zeiger auf einen dieser Commits. Die Standardbezeichnung für einen Branch bei Git lautet `master`. Wenn Sie damit beginnen, Commits durchzuführen, erhalten Sie einen `master`-Branch, der auf den letzten Commit zeigt, den Sie gemacht haben. Jedes Mal, wenn Sie einen Commit durchführen, 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.

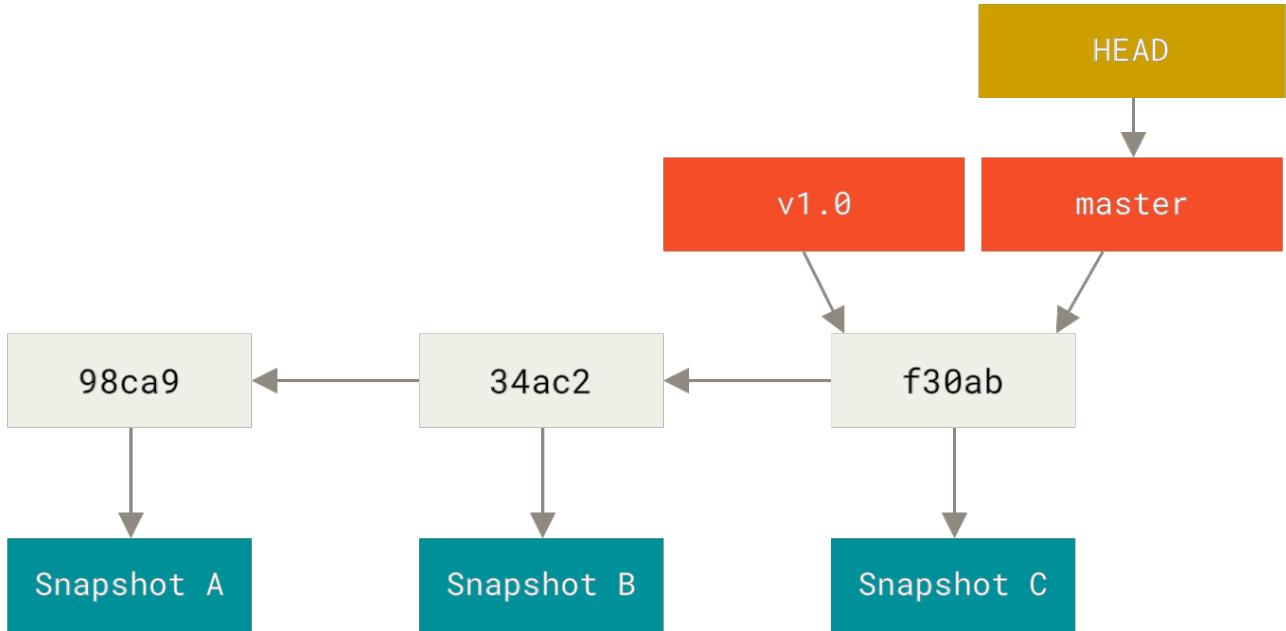


Figure 11. Ein Branch und sein Commit-Verlauf

## Erzeugen eines neuen Branches

Was passiert, wenn Sie einen neuen Branch erzeugen? Nun, wenn Sie das tun, wird ein neuer Zeiger erstellt, mit dem Sie sich in der Entwicklung fortbewegen können. Nehmen wir an, Sie erzeugen einen neuen Branch mit dem Namen „testing“. Das machen Sie mit der Anweisung `git branch`:

```
$ git branch testing
```

Dieser Befehl erzeugt einen neuen Zeiger, der auf den selben Commit zeigt, auf dem Sie sich gegenwärtig befinden.

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

Woher weiß Git, auf welchem Branch Sie gegenwärtig sind? Es besitzt einen speziellen Zeiger namens `HEAD`. Beachten Sie, dass dieser `HEAD` sich sehr stark unterscheidet von den `HEAD`-Konzepten anderer Versionsverwaltungen, mit denen Sie vielleicht vertraut sind, wie Subversion oder CVS. Bei Git handelt es sich bei `HEAD` um einen Zeiger auf den lokalen Branch, auf dem Sie sich gegenwärtig befinden. In diesem Fall sind Sie noch 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

Sie können das leicht nachvollziehen, indem Sie den einfachen Befehl `git log` ausführen, mit dem Sie sehen, 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 Fixed bug #1328 - stack overflow under certain conditions
98ca9 The initial commit of my project
```

Sie können die `master`- und `testing`-Branches sehen, die sich rechts neben dem Commit von `f30ab` befinden.

## Wechseln der Branches

Um zu einem existierenden Branch zu wechseln, führen Sie die Anweisung `git checkout` aus. Lassen Sie uns zu dem neuen `testing`-Branch wechseln.

```
$ git checkout testing
```

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

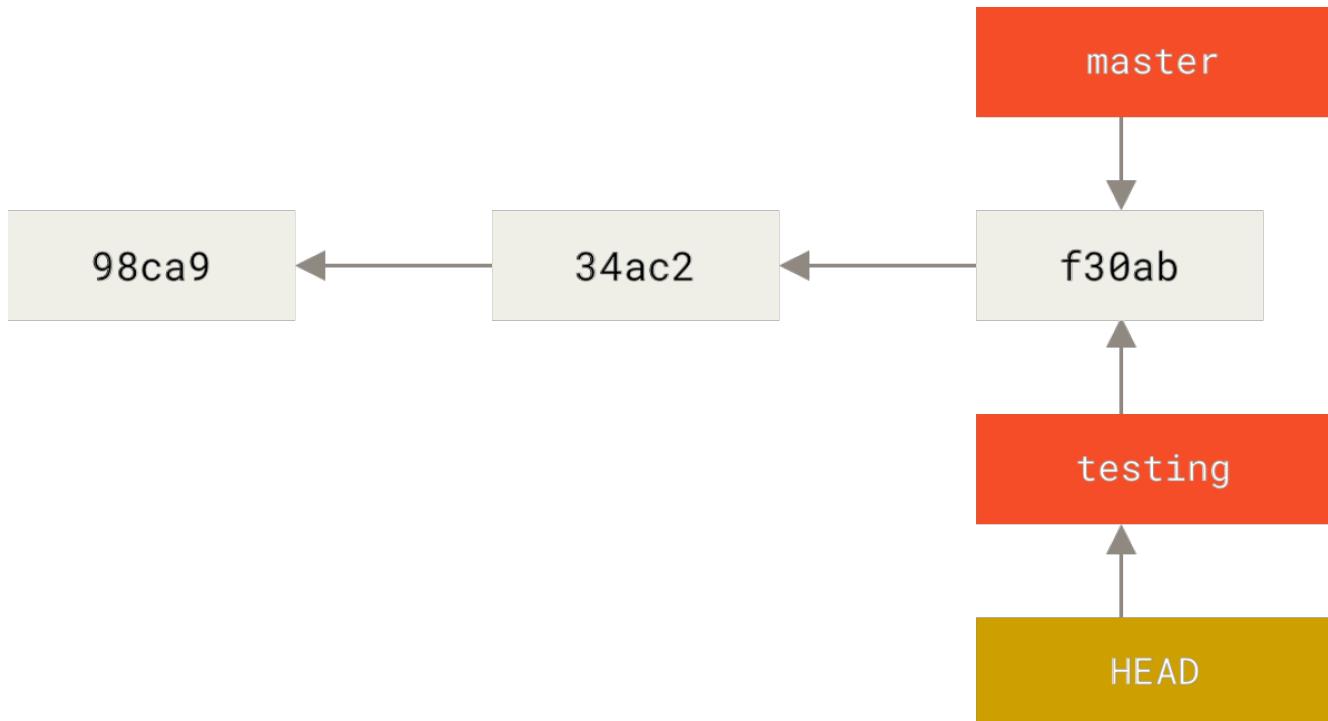


Figure 14. HEAD zeigt auf den aktuellen Branch

Was bedeutet das? Nun, lassen Sie und 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 Ihr `testing`-Branch vorwärts bewegt hat, aber Ihr `master`-Branch noch auf den Commit zeigt, auf dem Sie sich befanden, als Sie die Anweisung `git checkout` ausführten, um die Branches zu wechseln. Lassen Sie uns zurückwechseln zum `master`-Branch.

```
$ git checkout master
```

Figure 16. HEAD bewegt sich, wenn Sie auschecken

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

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



Es ist wichtig zu beachten, dass sich die Dateien in Ihrem Arbeitsverzeichnis verändern, wenn Sie in Git die Branches wechseln. Wenn Sie zu einem älteren Branch wechseln, wird Ihr Arbeitsverzeichnis zurückverwandelt, sodass es aussieht wie zu dem Zeitpunkt, als Sie Ihren letzten Commit auf diesem Branch durchgeführt haben. Wenn Git das nicht problemlos durchführen kann, lässt es Sie die Branches überhaupt nicht wechseln.

Lassen Sie uns ein paar Änderungen vornehmen und noch einen Commit durchführen:

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

Jetzt hat sich Ihr Projektverlauf verzweigt (siehe [Verzweigter Verlauf](#)). You created and switched to a branch, did some work on it, and then switched back to your main branch and did other work. Both of those changes are isolated in separate branches: you can switch back and forth between the branches and merge them together when you're ready. And you did all that with simple [branch](#), [checkout](#), and [commit](#) commands.

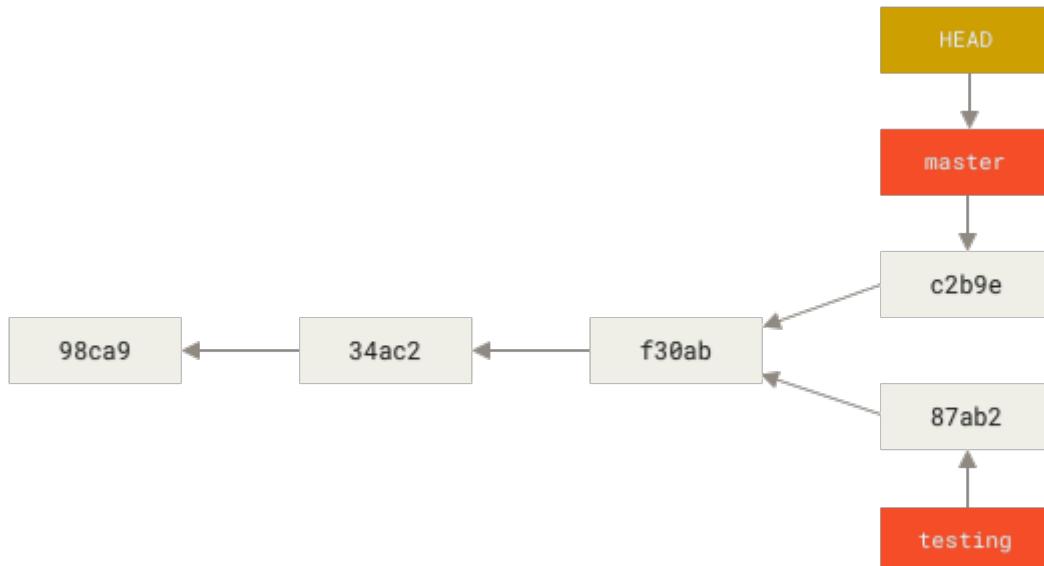


Figure 17. Verzweigter Verlauf

Sie können sich dies auch mühelos ansehen, wenn Sie die Anweisung `git log` ausführen. Wenn Sie die Anweisung `git log --oneline --decorate --graph --all` ausführen, wird Ihnen der Verlauf Ihrer Commits so angezeigt, dass erkennbar ist, wo Ihre Branch-Zeiger sich befinden und wie Ihr 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
* 34ac2 fixed 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, zu dem sie zeigt, können Branches ohne großen Aufwand erzeugt und vernichtet 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 in krassem Gegensatz zu der Art und Weise, wie die meisten älteren Werkzeuge zur Versionsverwaltung Branches anlegen, welches das Kopieren aller Projektdateien in ein zweites Verzeichnis einschließt. 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.

Lassen Sie uns herausfinden, warum Sie so handeln sollten.

*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.

## Einfaches Branching und Merging

Lassen Sie uns ein einfaches Beispiel für das Verzweigen und Zusammenführen (engl. branching and merging) anschauen, wie es Ihnen in einem praxisnahen Workflow begegnen könnte. Führen Sie diese Schritte aus:

1. Arbeiten Sie an einer Website
2. Erstellen Sie einen Branch für eine neue Anwendergeschichte, an der Sie gerade arbeiten
3. Erledigen Sie einige Arbeiten in diesem Branch

In diesem Moment erhalten Sie einen Anruf, dass ein anderes Problem kritisch ist und Sie einen Hotfix benötigen. Dazu werden Sie folgendes machen:

1. Wechseln Sie zu Ihrer Produktions-Branch
2. Erstellen Sie einen Branch, um den Hotfix einzufügen
3. Nachdem der Test abgeschlossen ist, mergen Sie den Hotfix-Branch und schieben ihn in die Produktions-Branch

4. Wechseln Sie zurück zu Ihrer ursprünglichen Anwenderstory und arbeiten Sie daran weiter

## Einfaches Branching

Lassen Sie uns zunächst annehmen, Sie arbeiten an Ihrem Projekt und haben bereits ein paar Commits in Ihre **master**-Branch gemacht.



Figure 18. Ein einfacher Commit-Verlauf

Sie haben sich dafür entschieden, an „Issue #53“ zu arbeiten aus irgendeinem Fehlerverfolgungssystem, das Ihre Firma benutzt. Um einen neuen Branch anzulegen und gleichzeitig zu diesem zu wechseln, können Sie 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
```

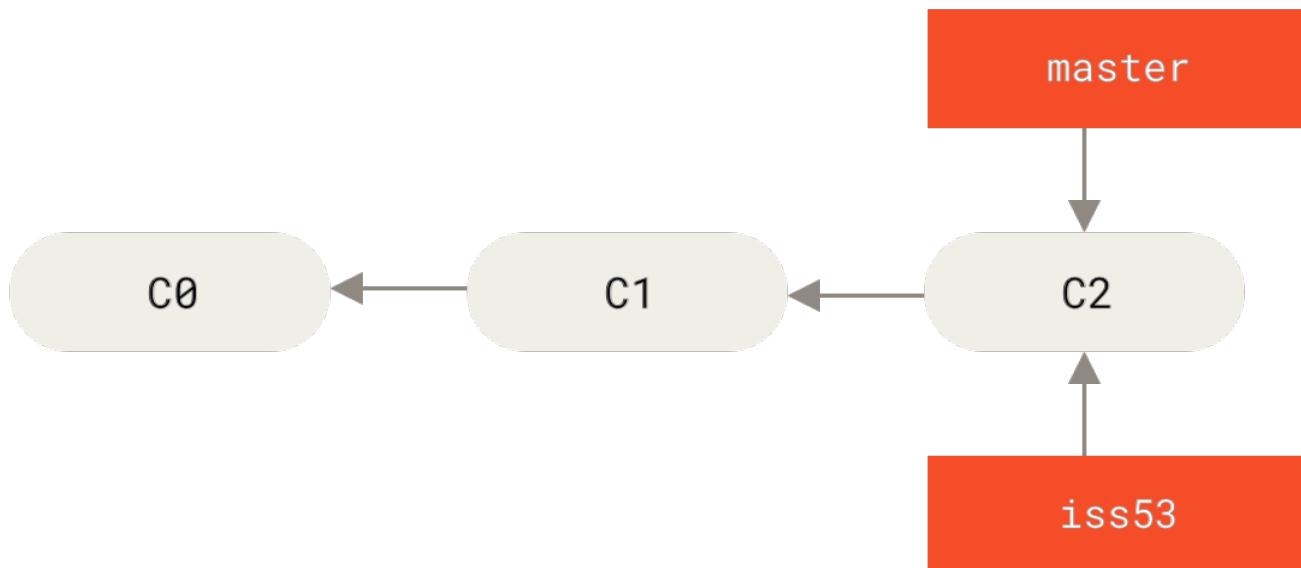


Figure 19. Erstellen eines neuen Branch-Zeigers

Sie arbeiten an Ihrer Website und führen einige Commits durch. Sobald Sie das machen, bewegt das den `iss53`-Branch vorwärts, weil Sie in ihn gewechselt (engl. checked out) haben. Das bedeutet, Ihr `HEAD` zeigt auf diesen Branch:

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

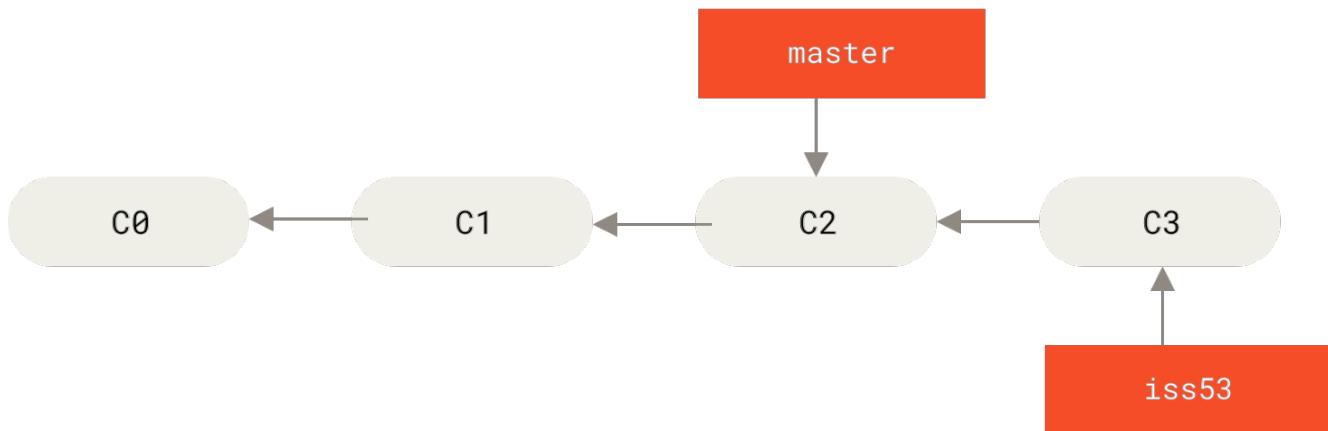


Figure 20. Der `iss53`-Branch hat sich bei Ihrer Arbeit vorwärts bewegt

Jetzt bekommen Sie einen Anruf, dass es ein Problem mit der Website gibt und Sie es umgehend beheben müssen. Bei Git müssen Sie Ihren Fix nicht zusammen mit den Änderungen bereitstellen, die Sie bereits an `iss53` vorgenommen haben, und Sie müssen auch keinen großen Aufwand damit betreiben, diese Änderungen rückgängig zu machen, bevor Sie daran arbeiten können, Ihren Fix auf das anzuwenden, was sich in der Produktionsumgebung befindet. Alles, was Sie machen müssen, ist, zu Ihrem `master`-Branch zurück zu wechseln.

Beachten Sie dabei, dass Git das Wechseln zu einer anderen Branch blockiert, falls Ihr Arbeitsverzeichnis oder Ihr Staging-Bereich nicht committete Modifikationen enthält, die Konflikte verursachen. Es ist am besten, einen sauberer Zustand des Arbeitsbereichs anzustreben, bevor Sie die Zweige wechseln. Es gibt Möglichkeiten, das zu umgehen (nämlich das Verstecken/Stashen und Revidieren/Amending von Änderungen), die wir später in Kapitel 7 [Git Stashing](#) behandeln werden. Lassen Sie uns vorerst annehmen, Sie haben für alle Ihre Änderungen Commits durchgeführt, sodass Sie zu Ihrem `master`-Branch zurück wechseln können.

```
$ 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 Sie mit der Arbeit an „Issue #53“ begonnen haben und Sie können sich direkt auf den Hotfix konzentrieren. Das ist ein **wichtiger Punkt**, den Sie unbedingt beachten sollten: Wenn Sie die Branches wechseln, setzt Git Ihr Arbeitsverzeichnis zurück, um so auszusehen, wie es das letzte Mal war, als Sie in den Branch committed haben. Dateien werden automatisch hinzugefügt, entfernt und verändert, um sicherzustellen, dass Ihre Arbeitskopie auf dem selben Stand ist wie zum Zeitpunkt Ihres letzten Commits auf diesem Branch.

Als Nächstes müssen Sie sich um den Hotfix kümmern. Lassen Sie uns einen `hotfix`-Branch erstellen, an dem Sie bis zu dessen Fertigstellung arbeiten:

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

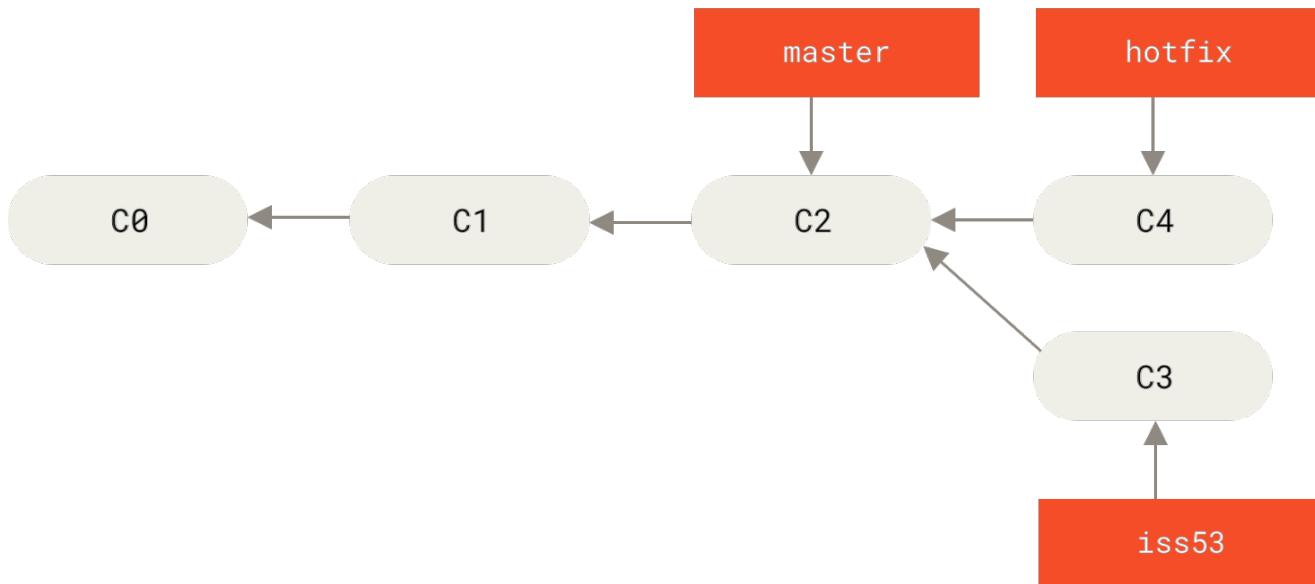


Figure 21. Auf der `master`-Branch basierender `hotfix`-Branch

Sie können Ihre Tests durchführen, sich vergewissern, dass der Hotfix das macht, was Sie von ihm erwarten und schließlich den Branch `hotfix` wieder in Ihren `master`-Zweig integrieren (engl. merge), um ihn in der Produktion einzusetzen. Das machen Sie mit der Anweisung `git merge`:

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

Ihnen wird bei diesem Zusammenführen der Ausdruck „fast-forward“ auffallen. Da der Commit **C4**, auf den der von Ihnen eingebundene Branch `hotfix` zeigt, direkt vor dem Commit **C2** liegt, auf dem Sie sich befinden, bewegt Git den Pointer einfach nach vorne. Um es anders auszudrücken, wenn Sie versuchen, 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 abweichenden Arbeiten gibt, die miteinander gemerget werden müssen – das wird als „fast-forward“ bezeichnet.

Ihre Änderung befindet sich nun im Schnappschuss des Commits, auf den der `master`-Branch zeigt

und Sie können Ihre Fehlerbehebung anwenden.

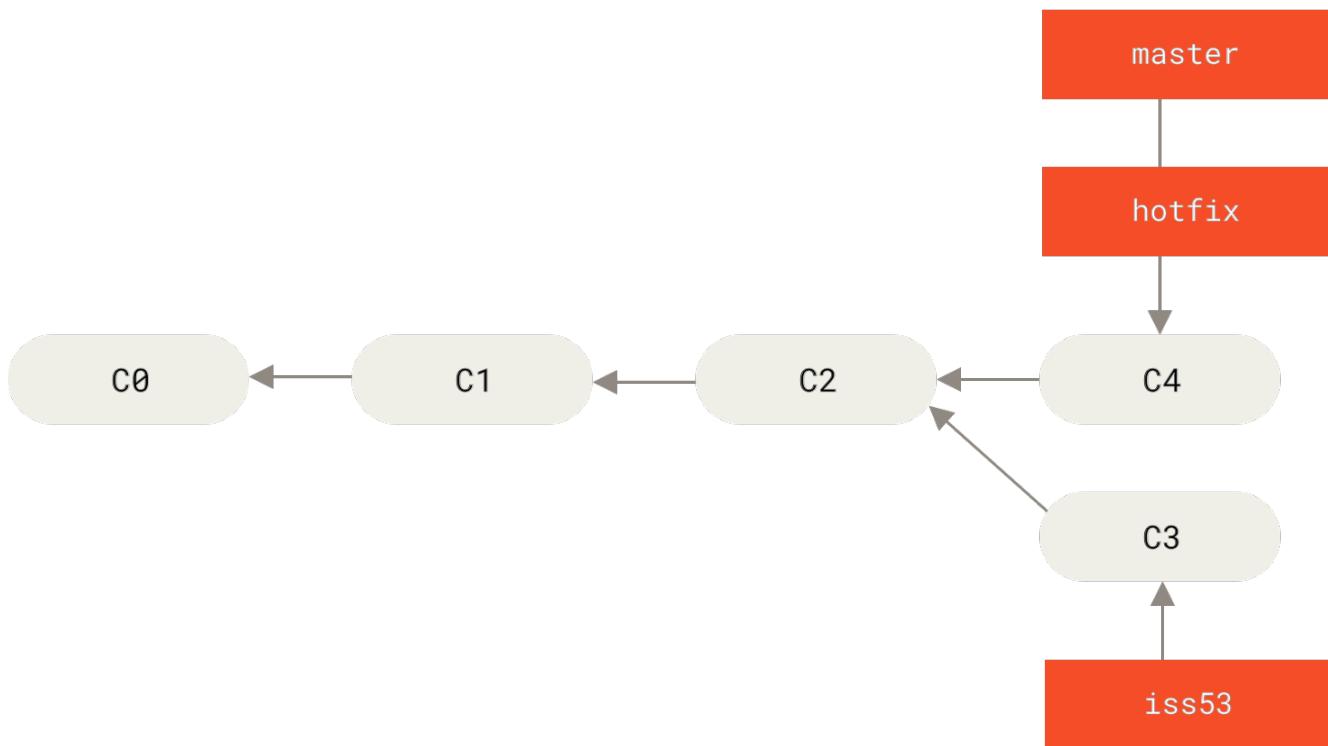


Figure 22. `master` wurde zu `hotfix` „fast-forwarded“

Nachdem Ihre überaus wichtige Fehlerbehebung bereitgestellt wurde, können Sie sich wieder dem zuwenden, woran Sie gerade gearbeitet haben, als Sie unterbrochen wurden. Zunächst sollten Sie jedoch den `hotfix`-Branch löschen, weil Sie diesen nicht länger benötigen – schließlich verweist der `master`-Branch auf den selben Entwicklungsstand. Sie können ihn mit der Anweisung `git branch` und der Option `-d` löschen:

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

Jetzt können Sie zu dem Branch zurückwechseln, auf dem Sie mit Ihren Arbeiten an „Issue #53“ begonnen hatten, und daran weiter arbeiten.

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

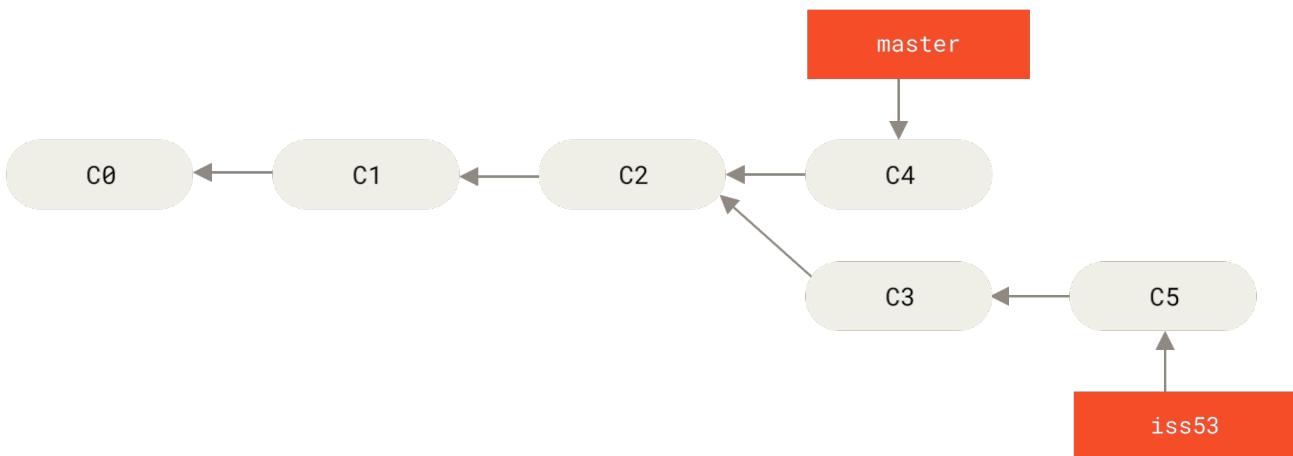


Figure 23. Arbeiten an iss53 weitergeführt

Es ist erwähnenswert, dass die Arbeit, die Sie in Ihrem `hotfix`-Branch durchgeführt haben, nicht in den Dateien in Ihrem `iss53`-Branch enthalten ist. Wenn Sie diese Änderungen übernehmen müssen, können Sie in Ihren `master`-Branch den `iss53`-Zweig einbinden indem Sie `git merge master` ausführen, oder Sie können warten, bis Sie sich später entscheiden, den `iss53`-Zweig wieder zurück nach `master` zu pullen.

## Einfaches Merging

Angenommen, Sie haben entschieden, dass Ihr Issue #53 abgeschlossen ist und Sie bereit sind, ihn in Ihren „Master“ Branch zu integrieren. Dann werden Sie Ihren `iss53`-Branch in den `master`-Branch mergen, so wie Sie es zuvor mit dem `hotfix`-Branch gemacht haben. Sie müssen nur mit der Anweisung `checkout` zum dem Branch zu wechseln, in welchen Sie etwas einfließen lassen wollen 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, das Sie zuvor gemacht haben. Hier hat sich der Entwicklungsverlauf an einem früheren Zustand geteilt. Da der Commit auf dem Branch, auf dem Sie sich gerade befinden, kein unmittelbarer Vorgänger der Branch ist, in den Sie fusionieren, muss Git einige Arbeiten erledigen. In diesem Fall führt Git einen einfachen Drei-Wege-Merge durch, indem er die beiden Schnapschüsse verwendet, auf die die Branch-Spitzen und der gemeinsame Vorfahr der beiden zeigen.

Figure 24. Drei Schnapschüsse, 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.

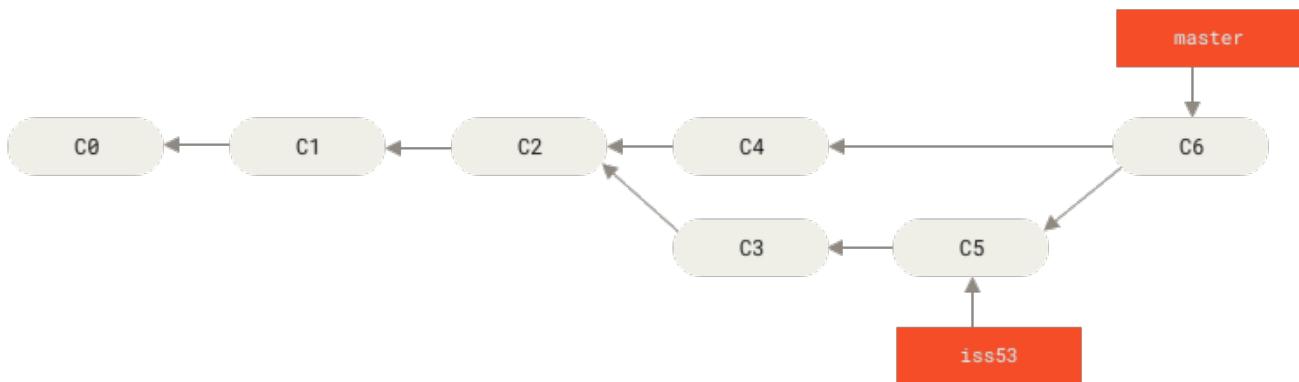


Figure 25. Ein Merge-Commit

Da Ihre Änderungen jetzt eingeflossen sind, haben Sie keinen weiteren Bedarf mehr für den `iss53`-Branch. Sie können das Ticket in Ihrem Ticket-Tracking-System schließen und den Branch löschen:

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

## Einfache Merge-Konflikte

Gelegentlich verläuft der Merge-Prozess nicht ganz reibungslos. Wenn Sie in den beiden Branches, die Sie zusammenführen wollen, an der selben Stelle in der selben Datei unterschiedliche Änderungen vorgenommen haben, wird Git nicht in der Lage sein, diese sauber zusammenzuführen. Wenn Ihr Fix für „Issue #53“ den gleichen Teil einer Datei wie der Branch `hotfix` geändert hat, erhalten Sie 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 automatisch **keinen** neuen Merge-Commit erstellen. Es hat den Prozess angehalten, bis Sie den Konflikt beseitigt haben. Wenn Sie sehen möchten, welche Dateien nach einem Merge-Konflikt an einem bestimmten Punkt nicht zusammengeführt wurden, können Sie `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 Sie sie manuell öffnen und diese Konflikte lösen können. Ihre 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** (Ihres **master**-Branches, denn der wurde per **checkout** aktiviert, als Sie das **merge** gestartet haben) ist der obere Teil des Blocks (alles oberhalb von **=====**), und die Version aus dem **iss53**-Branch sieht wie der darunter befindliche Teil aus. Um den Konflikt zu lösen, müssen Sie sich entweder für einen der beiden Teile entscheiden oder Sie führen die Inhalte selbst zusammen. Sie können diesen Konflikt beispielsweise lösen, indem Sie 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 Sie alle problematischen Bereiche in allen von dem Konflikt betroffenen Dateien beseitigt haben, führen Sie 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 Sie ein grafisches Tool benutzen möchten, um die Probleme zu lösen, dann können Sie **git mergetool** verwenden, welches ein passendes grafisches Merge-Tool startet und Sie 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:
opendifp kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge
p4merge araxis bc3 codecompare vimdiff emerge
Merging:
index.html

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

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



Wenn Sie fortgeschrittenere Werkzeuge zur Lösung kniffliger Merge-Konflikte benötigen, erfahren Sie mehr darüber in Kapitel 7 [Fortgeschrittenes Merging](#).

Nachdem Sie das Merge-Tool beendet haben, werden Sie von Git gefragt, ob das Zusammenführen erfolgreich war. Wenn Sie dem Skript bestätigen, dass es das war, wird die Datei der Staging Area hinzugefügt und der Konflikt als gelöst zu markiert. Sie können 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 Sie damit zufrieden sind und Sie geprüft haben, dass alles, was Konflikte aufwies, zum Index hinzugefügt wurde, können Sie 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
#

```

Sie können dieser Commit-Nachricht noch Details darüber hinzufügen, wie Sie diesen Merge-Konflikt gelöst haben. Es könnte für künftige Betrachter dieses Commits hilfreich sein zu verstehen, warum Sie was getan haben, falls es nicht offensichtlich ist.

## Branch-Management

Nachdem Sie nun einige Branches erzeugt, zusammengeführt und gelöscht haben, lassen Sie uns jetzt einige Werkzeuge für das Branch-Management betrachten, die sich als sehr nützlich erweisen werden, wenn Sie erst einmal ständig Branches benutzen.

Der `git branch` Befehl kann noch mehr, als Branches zu erzeugen und zu löschen. Wenn Sie die Anweisung ohne Argumente ausführen, bekommen Sie eine einfache Auflistung Ihrer aktuellen Branches:

```

$ git branch
  iss53
* master
  testing

```

Beachten Sie das `*`-Zeichen, das dem `master`-Branch vorangestellt ist: es zeigt an, welchen Branch Sie gegenwärtig ausgecheckt haben (bzw. den Branch, auf den `HEAD` zeigt). Wenn Sie zu diesem Zeitpunkt einen Commit durchführen, wird der `master`-Branch durch Ihre neue Änderung vorwärts bewegt. Um sich den letzten Commit auf jedem Branch anzeigen zu lassen, können Sie 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 readmes
```

Die nützlichen Optionen `--merged` und `--no-merged` können diese Liste nach Branches filtern, welche bereits mit dem Branch, auf dem Sie sich gegenwärtig befinden, zusammengeführt wurden und welche nicht. Um zu sehen, welche Branches schon mit dem Branch zusammengeführt wurden, auf dem sie gerade sind, können Sie die Anweisung `git branch --merged` ausführen:

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

Da Sie den Branch `iss53` schon früher gemitigt haben, sehen Sie ihn in Ihrer Liste. Branches auf dieser Liste ohne vorangestelltes `*` können für gewöhnlich einfach mit der Anweisung `git branch -d` gelöscht werden; Sie haben deren Änderungen bereits zu einem anderen Branch hinzugefügt, sodass Sie nichts verlieren würden.

Um alle Branches zu sehen, welche Änderungen enthalten, die Sie noch nicht integriert haben, können Sie die Anweisung `git branch --no-merged` ausführen:

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

Das zeigt Ihnen 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 Sie den Branch wirklich löschen und diese Bearbeitungen aufgeben wollen, können Sie ihn mit der Option `-D` zum Löschen zwingen, wie die Hilfsmeldung anzeigt.

Wenn Sie keinen Commit- oder Branch-Namen als Argument angeben zeigen Ihnen die oben beschriebenen Optionen `--merged` und `--no-merged` was jeweils in Ihren *current*-Branch gemergt oder nicht gemergt wurde.

Sie können immer ein zusätzliches Argument angeben, um nach dem Merge-Status in Bezug auf einen anderen Zweig zu fragen, ohne zu diesen anderen Zweig zuerst wechseln zu müssen. So wie im Beispiel unten: „Was ist nicht in den `master` Branch integriert?“

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

## Branching-Workflows

Jetzt haben Sie die Grundlagen des Verzweigens (Branching) und Zusammenführen (Merging) kennengelernt. Was können oder sollten Sie damit anfangen? In diesem Abschnitt werden wir einige gängige Arbeitsabläufe vorstellen, welche das vereinfachte Branching ermöglichen, so dass Sie entscheiden können, ob Sie es in Ihren eigenen Entwicklungszyklus integrieren möchten.

### Langfristige 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, Sie können mehrere Branches haben, die immer offen sind und die Sie für unterschiedliche Stadien Ihres Entwicklungszyklus verwenden; Sie können sie regelmäßig mit anderen zusammenführen.

Viele Git-Entwickler haben einen Arbeitsablauf, welcher den Ansatz verfolgt, nur vollkommen stabilen Code im `master`-Branch zu haben – möglicherweise auch nur Code, der released wurde oder werden soll. Sie haben weitere parallele `develop` oder `next` Branches, auf dem Sie arbeiten oder den Sie für Stabilitätstests nutzen – dieser ist nicht zwangsläufig stabil, aber wann immer er einen stabilen Zustand erreicht, kann er mit dem `master`-Branch zusammengeführt werden. Es wird benutzt, um Themen-Branches (kurzfristige Branches, wie Ihr früherer `iss53`-Branch) einfließen zu lassen, wenn diese fertiggestellt sind, um sicherzustellen, dass diese alle Tests bestehen und keine Fehler einschleppen.

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

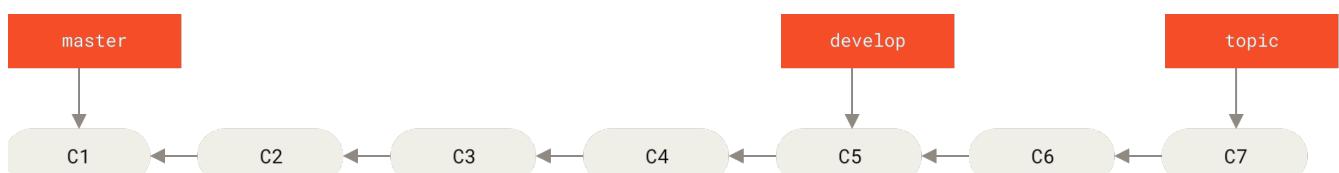


Figure 26. Lineares Modell eines Branchings mit zunehmender Stabilität

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

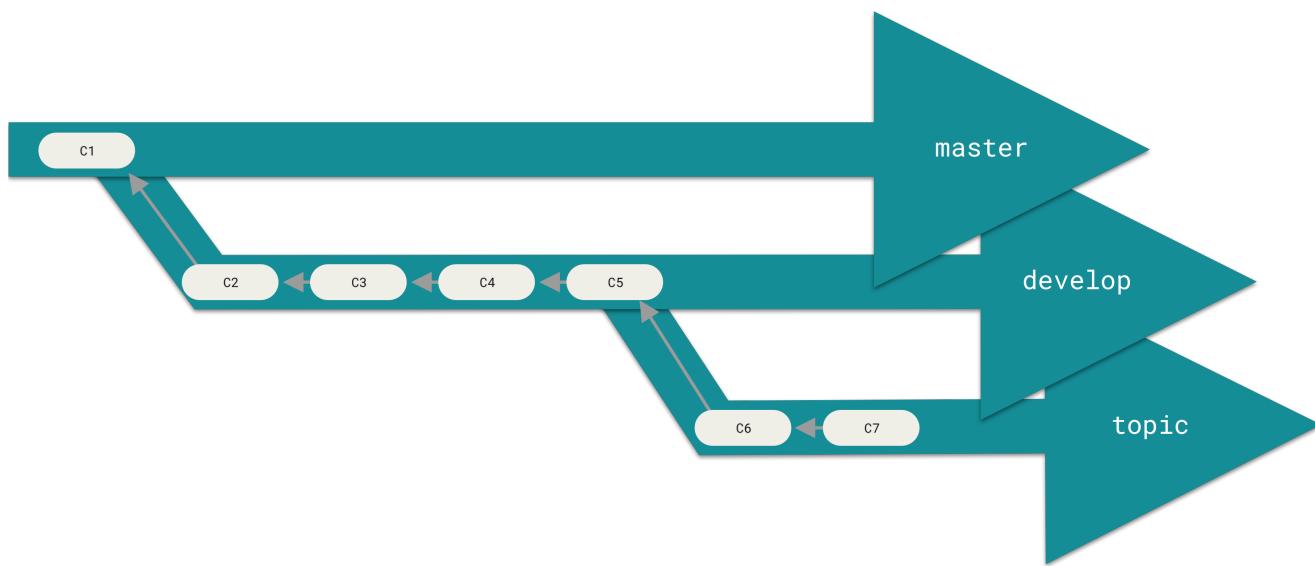


Figure 27. „Silo“-Modell eines Branchings mit zunehmender Stabilität

Sie können 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 **next** oder **master** Branch 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. Nochmal, langfristig verschiedene Branches parallel laufen zu lassen, ist nicht notwendig, aber oft hilfreich, insbesondere wenn man es mit sehr großen oder komplexen Projekten zu tun hat.

## Themen-Branches

Themen-Branches sind in Projekten jeder Größe nützlich. Ein Themen-Branch ist ein kurzlebiger Branch, welchen Sie für eine ganz bestimmte Funktion oder zusammengehörende Arbeiten erstellen und benutzen. Das ist etwas, was Sie wahrscheinlich noch nie zuvor mit einem Versionsverwaltungssystem gemacht haben, 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.

Sie haben das im letzten Abschnitt an den Branches **iss53** und **hotfix** gesehen, die Sie erstellt haben. Sie führten mehrere Commits auf diesen Branches durch und löschten sie sofort, nachdem Sie sie mit Ihrem Hauptbranch zusammengeführt haben. Diese Technik erlaubt es Ihnen, schnell und vollständig den Kontext zu wechseln – da Ihre Arbeit auf verschiedene Depots aufgeteilt ist, wo alle Änderungen auf diesem Branch unter diese Thematik fallen, ist es leichter nachzuvollziehen, was bei Code-Überprüfungen und ähnlichem geschehen ist. Sie können 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: Sie erledigen gerade einige Arbeiten (auf **master**), zweigen davon

ab wegen eines Problems ([iss91](#)), arbeiten daran eine Weile, zweigen davon den zweiten Branch ab, um eine andere Möglichkeit zur Handhabung des selben Problems auszuprobieren ([iss91v2](#)), wechseln zurück zu Ihrem master-Branch und arbeiten dort eine Zeitlang, und zweigen dann dort nochmal ab, um etwas zu versuchen, bei dem Sie sich nicht sind, ob es eine gute Idee ist ([dumbidea](#)-Branch). Ihre Commit-Verlauf wird in etwa so aussehen:

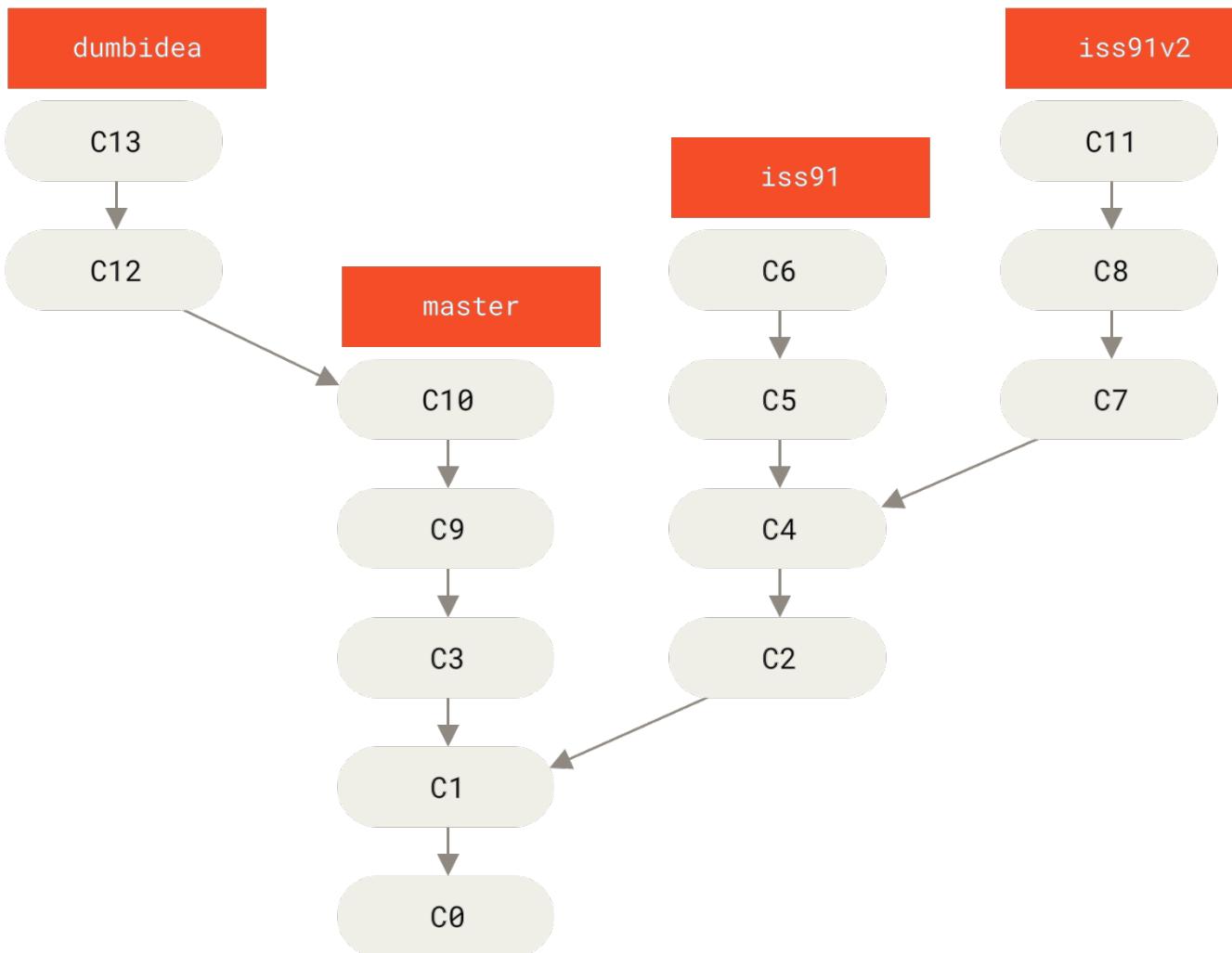


Figure 28. Mehrere Themen-Branches

Angenommen, Sie haben sich jetzt entschieden, dass Ihnen die zweite Lösung für Ihr Problem ([iss91v2](#)) am besten gefällt; und Sie haben den [dumbidea](#)-Branch Ihren Mitarbeitern gezeigt und es hat sich herausgestellt, dass er genial ist. Sie können also den ursprünglichen [iss91](#)-Branch (unter Verlust der Commits [C5](#) und [C6](#)) wegwerfen und die anderen beiden einfließen lassen. Ihr Verlauf sieht dann so aus:

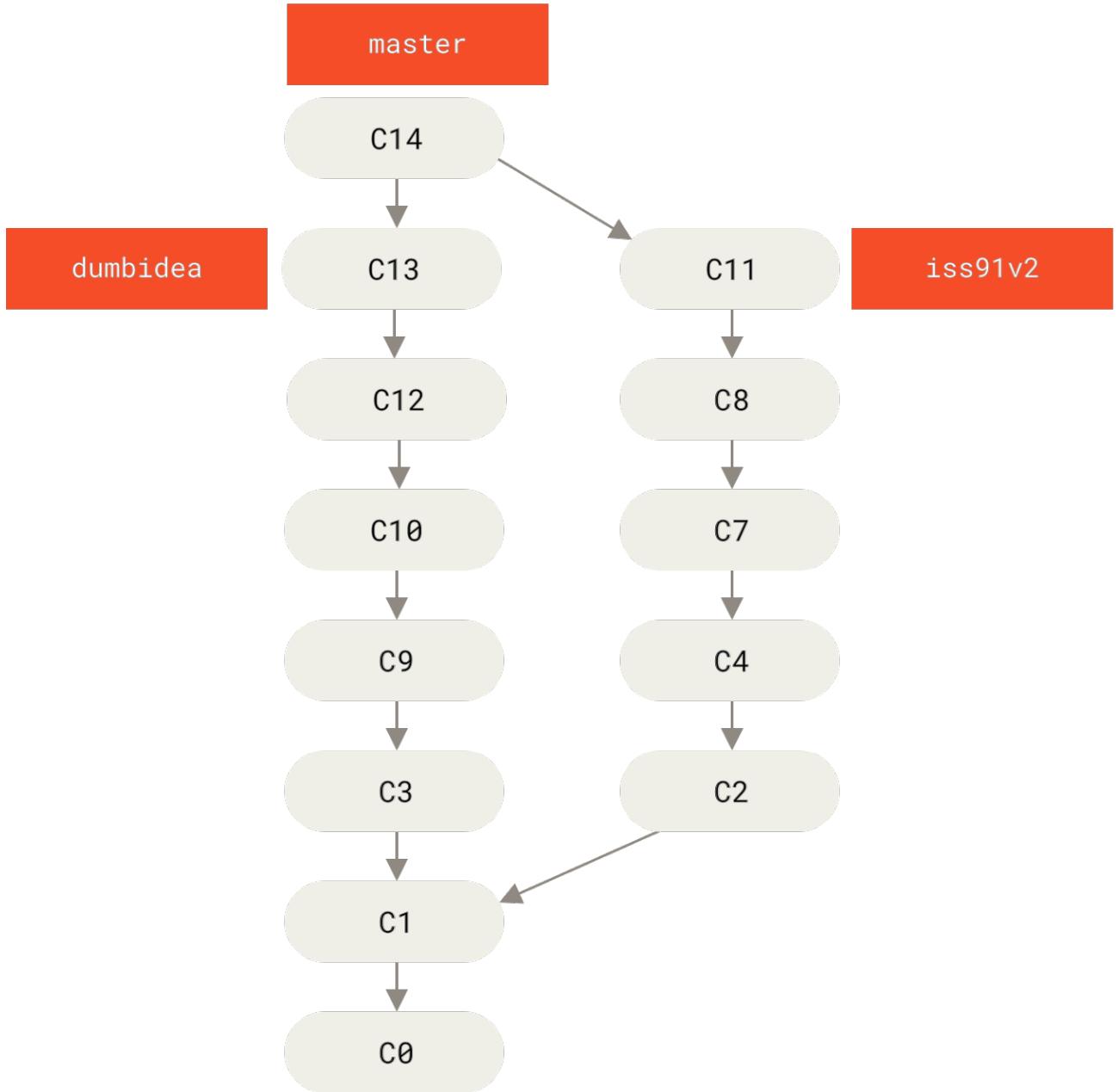


Figure 29. Verlauf nach dem Mergen von `dumbidea` und `iss91v2`

In Kapitel 5 [Verteiltes Git](#) werden wir die verschiedenen möglichen Arbeitsabläufe für Ihr Git-Projekt noch detaillierter betrachten. Bevor Sie sich also entscheiden, welches Branching-Modell Sie für Ihr nächstes Projekt nutzen wollen, sollten Sie unbedingt dieses Kapitel gelesen haben.

Während Sie das alles machen, ist es wichtig, daran zu denken, dass alle diese Branches nur lokal existieren. Wenn Sie Branches anlegen und zusammenführen, geschieht das alles nur in Ihrem lokalen Git-Repository – es findet keine Server-Kommunikation statt.

## Remote-Banches

Remote-Referenzen sind Referenzen (Zeiger) in Ihren Remote-Repositories, einschließlich Branches, Tags usw. Sie können eine vollständige, ausführliche Liste von Remote-Referenzen bekommen, wenn Sie die Anweisungen `git ls-remote (remote)` oder `git remote show (remote)` ausführen, für Remote-Banches sowie für weitere Informationen. Der gebräuchlichste 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 Sie nicht manuell ändern können, sie werden automatisch für Sie geändert, sobald Sie irgendeine Netzwerkkommunikation durchführen. Betrachten Sie sie als Lesezeichen, die Sie daran erinnern, wo die Branches in Ihren Remote-Repositories das letzte Mal standen, als Sie sich mit ihnen verbunden hatten.

Remote-Tracking-Branch-Namen haben die Form `<remote>/<branch>`. Wenn Sie beispielsweise wissen möchten, wie der `master`-Branch in Ihrem `origin`-Repository ausgesehen hat, als Sie zuletzt Kontakt mit ihm hatten, dann würden Sie den `origin/master`-Branch überprüfen. Wenn Sie mit einem Mitarbeiter an einem Problem gearbeitet haben und dieser bereits einen `iss53`-Branch hochgeladen (gepusht) hat, besitzen Sie möglicherweise Ihren eigenen lokalen `iss53`-Branch; aber der Branch auf dem Server würde auf den Remote-Tracking-Branch `origin/iss53` zeigen.

Das kann ein wenig verwirrend sein, lassen Sie uns also ein Beispiel betrachten. Angenommen, Sie haben in Ihrem Netzwerk einen Git-Server mit der Adresse `git.ourcompany.com`. Wenn Sie von diesem klonen, erhält der Server von der Git-Anweisung `clone` automatisch den Namen `origin`, lädt all seine Daten herunter, erstellt einen Zeiger zu dem Commit, auf den dessen `master`-Branch zeigt und benennt ihn lokal `origin/master`. Git gibt Ihnen auch Ihren eigenen lokalen `master`-Branch mit der gleichen Ausgangsposition wie der `origin/master`-Branch, damit Sie einen Punkt, wo Sie mit Ihrer Arbeit beginnen können.

### *„origin“ ist nichts besonderes*



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

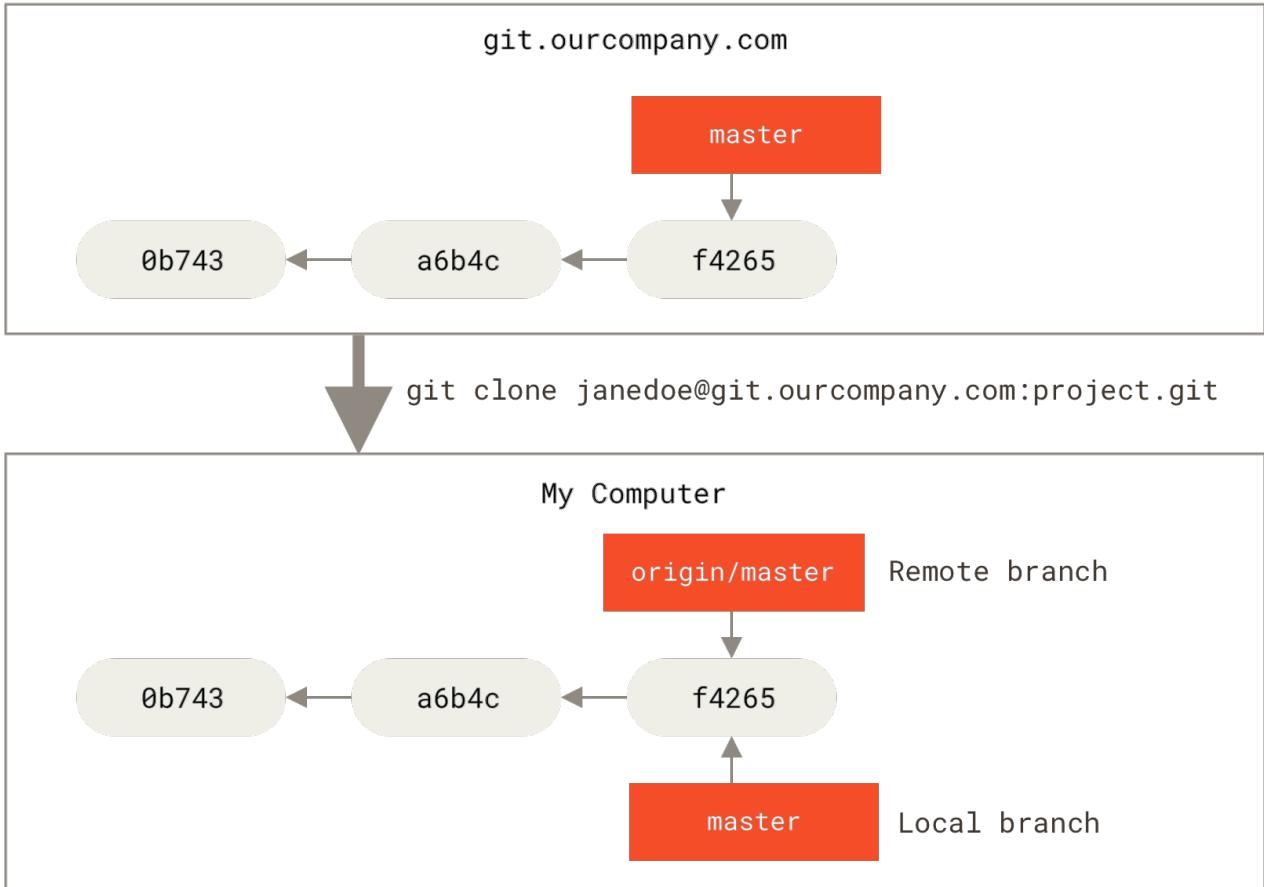


Figure 30. Entfernte und lokale Repositorys nach dem Klonen

Wenn Sie ein wenig an Ihrem lokalen **master**-Branch arbeiten und in der Zwischenzeit jemand anderes etwas zu [git.ourcompany.com](http://git.ourcompany.com) hochlädt und damit dessen **master**-Branch aktualisiert, dann bewegen sich eure Verläufe unterschiedlich vorwärts. Und solange Sie keinen Kontakt mit Ihrem **origin**-Server aufnehmen, bewegt sich Ihr **origin/master**-Zeiger nicht.

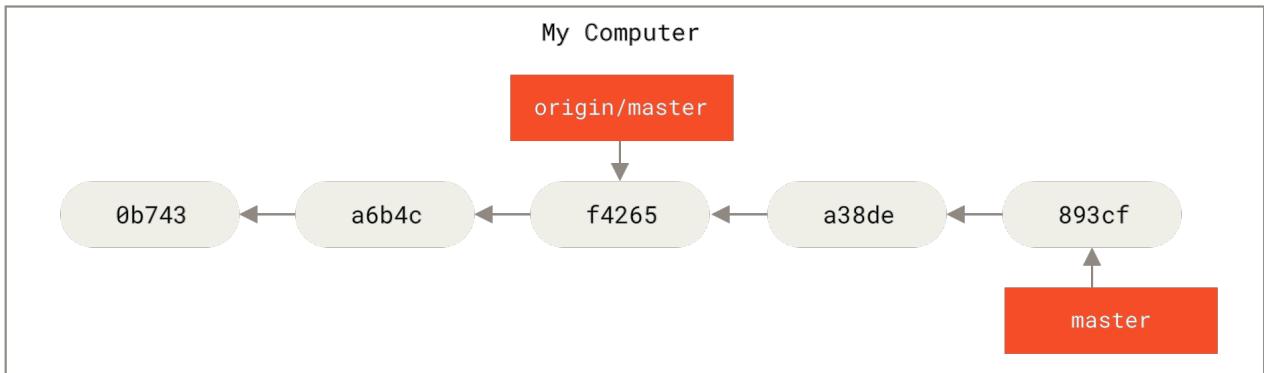
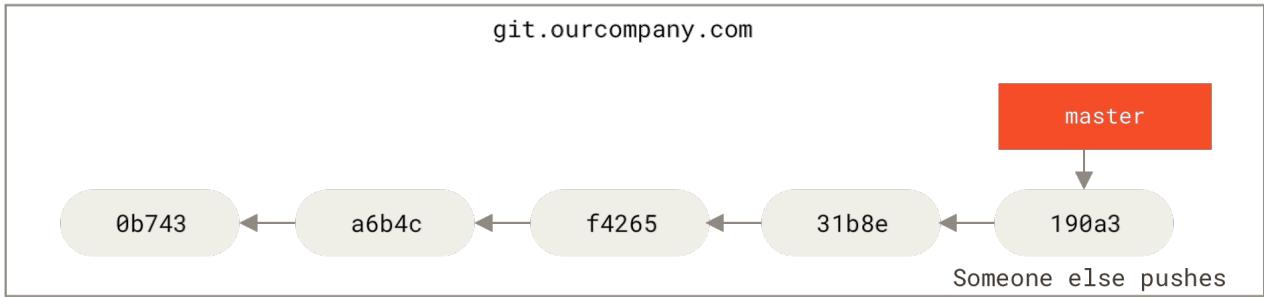


Figure 31. Lokale und entfernte Änderungen können auseinander laufen

Um Ihre Arbeit mit einer bestimmten Remote zu synchronisieren, führen Sie den Befehl `git fetch <remote>` aus (in unserem Fall `git fetch origin`). Der Befehl sucht, welcher Server „orgin“ ist (in diesem Fall `git.ourcompany.com`), holt alle Daten, die Sie noch nicht haben, und aktualisiert Ihre lokale Datenbank, indem es Ihren `origin/master`-Zeiger auf seine neue, aktuellere Position bewegt.

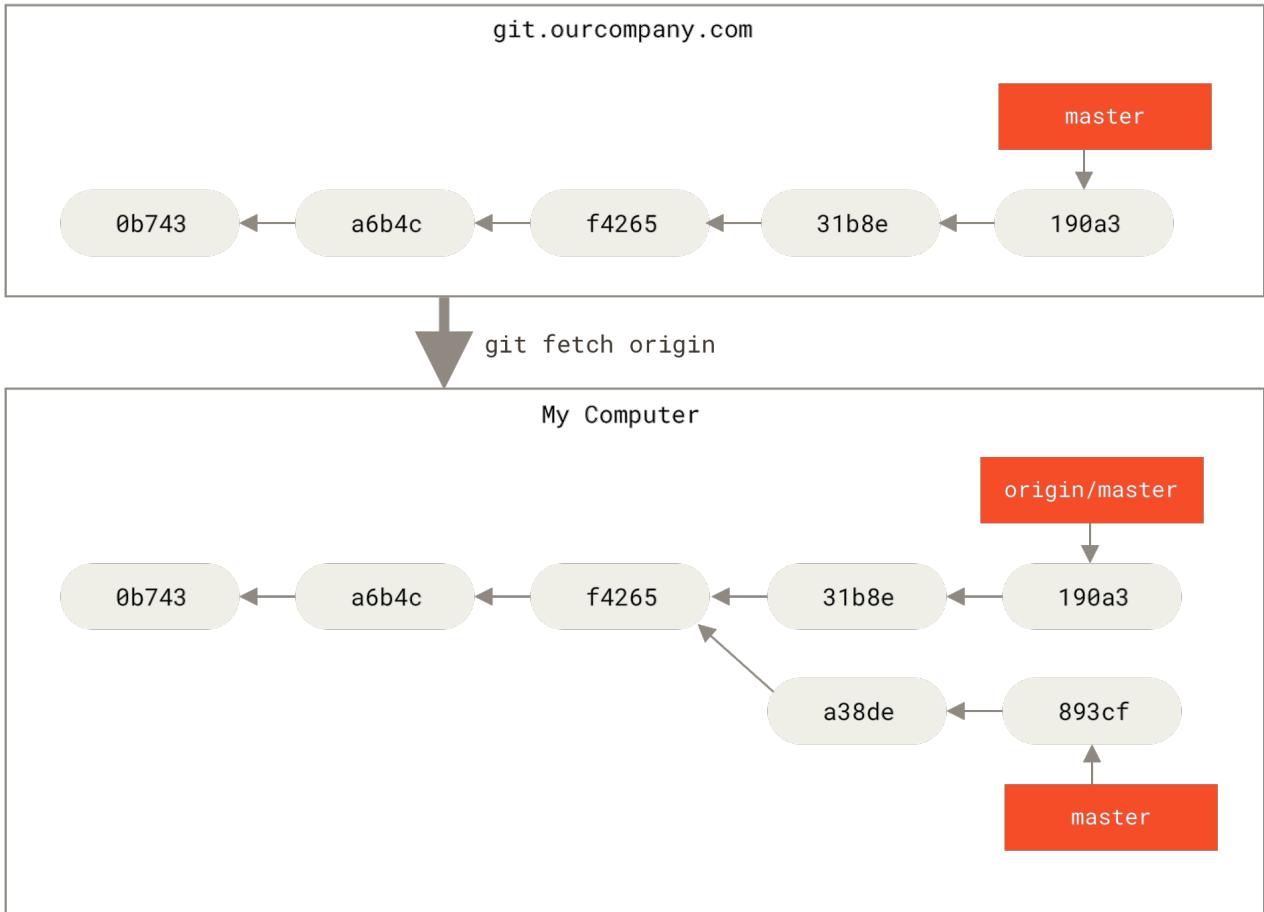


Figure 32. `git fetch` aktualisiert Ihre Remote-Tracking-Branches

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 Sie einen weiteren internen Git-Server haben, welcher von einem Ihrer Sprint-Teams nur zur Entwicklung genutzt wird. Diesen Server erreichen wir unter `git.team1.ourcompany.com`. Sie können ihn zu dem Projekt, an dem Sie gegenwärtig arbeiten, als neuen Remote-Server hinzufügen, indem Sie die Anweisung `git remote add` ausführen, 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.

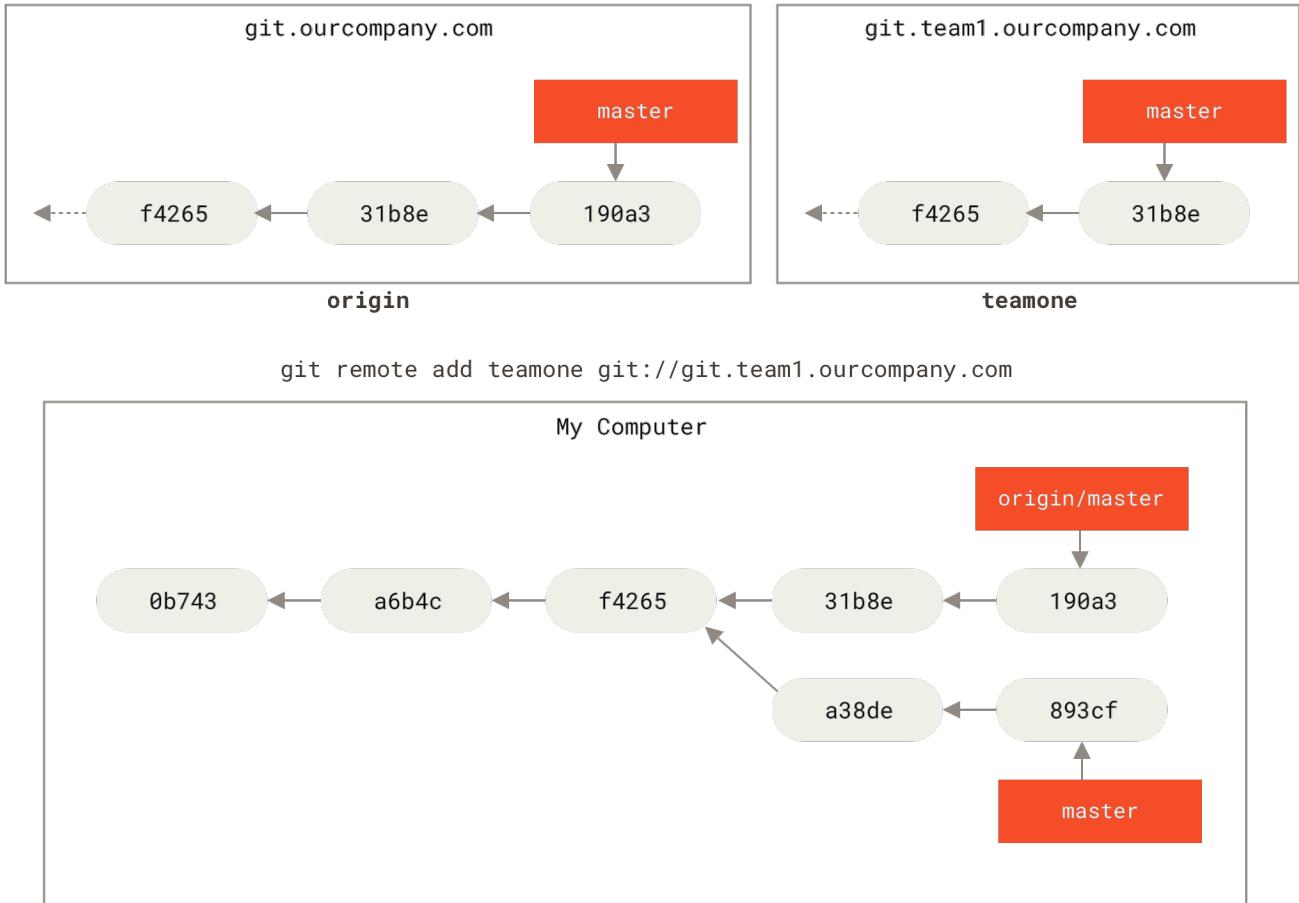


Figure 33. Hinzufügen eines weiteren Remote-Servers

Jetzt können Sie mit der Anweisung `git fetch teamone` alles vom Server holen, was Sie noch nicht haben. Da auf diesem Server nur eine Teilmenge der Daten ist, die sich genau jetzt auf Ihrem `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.

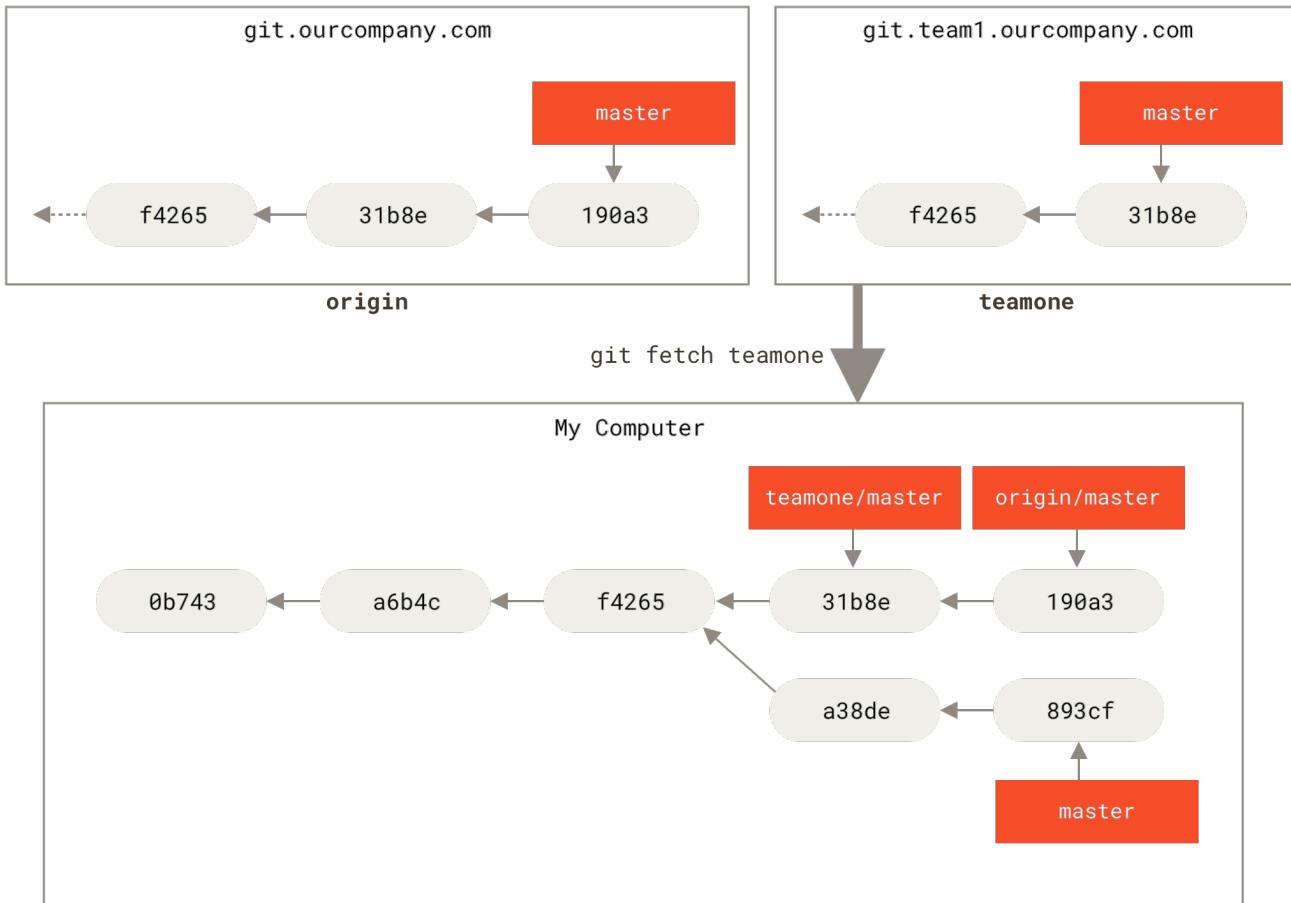


Figure 34. Remote-Tracking-Branch für teamone/master

## Pushing/Hochladen

Wenn Sie einen Branch mit der Welt teilen möchten, müssen Sie ihn auf einen Remote-Server hochladen, auf dem Sie Schreibrechte besitzen. Ihre lokalen Branches werden nicht automatisch mit den Remotes synchronisiert, auf die Sie schreiben – Sie müssen die Branches, die Sie freigeben möchten, explizit pushen. Auf diese Weise können Sie private Branches, die Sie nicht veröffentlichen wollen, zum Arbeiten benutzen und nur die Themen-Branches pushen, an denen Sie mitarbeiten wollen.

Wenn Sie einen Zweig namens `serverfix` besitzen, an dem Sie mit anderen arbeiten möchten, dann können Sie diesen auf dieselbe Weise hochladen wie Ihren ersten Branch. Führen Sie 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 `serverfix`-Branch und aktualisiere damit den `serverfix`-Branch auf meinem Remote-Server“. Wir werden den `refs/heads/-`-Teil in Kapitel 10 [Git Interna](#) noch näher beleuchten, Sie können ihn aber in der Regel auslassen. Sie können 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“. Sie können dieses Format auch benutzen, um einen lokalen Branch in einen Remote-Branch mit anderem Namen zu pushen. Wenn Sie nicht wollten, dass er auf dem Remote als `serverfix` bezeichnet wird, können Sie stattdessen `git push origin serverfix:awesomebranch` ausführen, um Ihren lokalen `serverfix` Branch auf den `awesomebranch` Branch im Remote-Projekt zu pushen.

#### *Geben Sie Ihr Passwort nicht jedes Mal neu ein*

Wenn Sie eine HTTPS-URL zum Übertragen verwenden, fragt Sie der Git-Server nach Ihrem Benutzernamen und Passwort zur Authentifizierung. Standardmäßig werden Sie auf dem Terminal nach diesen Informationen gefragt, damit der Server erkennen kann, ob Sie Push ausführen dürfen.



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

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

Das nächste Mal, wenn einer Ihrer Mitarbeiter Daten von 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 wissen, dass, wenn Sie einen Fetch durchführen, dass neue Remote-Tracking-Branches heruntergeladen werden, Sie nicht automatisch lokale, bearbeitbare Kopien von ihnen haben. Mit anderen Worten, in diesem Fall haben Sie keinen neuen Branch `serverfix` – Sie haben nur einen Zeiger `origin/serverfix`, den Sie nicht ändern können.

Um diese Änderungen in Ihren gegenwärtigen Arbeitsbranch einfließen zu lassen, können Sie die Anweisung `git merge origin/serverfix` ausführen. Wenn Sie Ihren eigenen `serverfix` Branch haben wollen, an dem Sie arbeiten können, können Sie ihn von Ihrem 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 Ihnen einen lokalen Branch, an dem Sie arbeiten können 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 Sie sich auf einem Tracking-Branch befinden und `git pull` eingeben, weiß Git automatisch, von welchem Server Daten abzuholen sind und in welchen Branch diese einfließen sollen.

Wenn Sie ein Repository klonen, wird automatisch ein `master`-Branch erzeugt, welcher `origin/master` trackt. Sie können jedoch auch andere Tracking-Banches erzeugen, wenn Sie wünschen – welche die nicht Zweige auf `origin` und dessen `master` Branch verfolgen. Der einfachste Fall ist das Beispiel, dass Sie gerade gesehen haben, 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 Sie zum Auschecken verwenden möchten (a), nicht existiert und (b) genau mit einem Namen auf nur einem Remote übereinstimmt, erstellt Git einen Tracking-Branch für Sie:

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

Um eine lokale Branch mit einem anderen Namen als die entfernte Branch einzurichten, können Sie 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 Ihre lokale Branch `sf` automatisch von `origin/serverfix` gepullt.

Wenn Sie bereits eine lokale Branch haben und diese auf eine Remote-Branch einstellen möchten, die Sie gerade abgerufen (gepullt) haben, oder wenn Sie die Upstream-Branch ändern möchten, die

Sie versionieren, könnten Sie die Option `-u` oder `--set-upstream-to` zu `git branch` verwenden, um sie zu einem beliebigen Zeitpunkt explizit festzulegen.

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

#### *Upstream Kürzel*



Wenn Sie einen Tracking-Branch eingerichtet haben, können Sie auf seinen Upstream-Branch mit der Kurzform `@{upstream}` oder `@{u}` verweisen. Wenn Sie also auf dem `master`-Zweig sind und er `origin/master` versioniert, können Sie, wenn Sie möchten, so etwas wie `git merge @{u}` anstelle von `git merge origin/master` verwenden.

Wenn Sie die Tracking-Banches sehen wollen, die Sie eingerichtet haben, können Sie die Anweisung `git branch` zusammen mit der Option `-vv` ausführen. Das listet Ihre lokalen Branches zusammen mit weiteren Informationen auf, einschließlich was jeder Branch versioniert und ob Ihr lokaler Branch voraus, hinterher oder beides ist.

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

Hier können wir also sehen, dass unser `iss53`-Branch den Branch `origin/iss53` verfolgt 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` verfolgt und auf den neuesten Stand ist. Als nächstes sehen wir, dass unser `serverfix`-Branch den Branch `server-fix-good` auf unserem `teamone`-Server versioniert und „ahead 3, behind 1“ bedeutet, dass es einen Commit auf dem Server gibt, den wir noch nicht gemitteilt 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 verfolgt.

Es ist wichtig zu beachten, dass diese Zahlen den Zustand zu dem Zeitpunkt beschreiben, als Sie zum letzten Mal Daten vom Server abgeholt haben. Diese Anweisung greift nicht auf die Server zu, sie liefert nur die Informationen, welche beim letzten Server-Kontakt lokal zwischengespeichert wurden. Wenn Sie vollkommen aktuelle Zahlen von „ahead“ und „behind“ wollen, dann müssen Sie, kurz bevor Sie die Anweisung ausführen, von all Ihren Remote-Servern Daten abholen (`fetch`). Sie könnten das so machen: `$ git fetch --all; git branch -vv`

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

## Pulling/Herunterladen

Während die Anweisung `git fetch` alle Änderungen auf dem Server abholt, die Sie zurzeit noch

nicht haben, sie wird aber an Ihrem Arbeitsverzeichnis überhaupt nichts verändern. Sie wird einfach die Daten für Sie holen und Ihnen das Zusammenführen überlassen. Es gibt jedoch die Anweisung namens `git pull`, welche im Grunde genommen ein `git fetch` ist, dem in den meisten Fällen augenblicklich ein `git merge` folgt. Wenn Sie einen Tracking-Branch eingerichtet haben, wie im letzten Abschnitt gezeigt, entweder indem Sie ihn explizit setzen oder indem Sie ihn mit den Befehlen `clone` oder `checkout` für sich haben erstellen lassen, dann sucht `git pull` nach dem Server und der versionierten Branch, verzweigt zu Ihrem aktuellen Branch, pullt von diesem Server und versucht dann, diesen entfernten Branch zu mergen.

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

## Remote-Banches Entfernen

Stellen wir uns vor, Sie sind mit Ihrem Remote-Branch fertig – Sie und Ihre Mitarbeiter sind fertig mit einer neuen Funktion und haben sie in den `master`-Branch des Remote-Servers (oder in welchem Branch auch immer sich Ihr stabiler Code befindet) einfließen lassen. Sie können einen Remote-Branch entfernen, indem die Anweisung `git push` zusammen mit der Option `--delete` ausführen. Wenn Sie Ihren `serverfix`-Branch vom Server löschen wollen, führen Sie 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 wieder herzustellen.

## Rebasing

Es gibt bei Git zwei Wege, um Änderungen von einem Branch in einen anderen zu integrieren: `merge` und `rebase`. In diesem Abschnitt werden Sie erfahren, was Rebasing ist, wie Sie es anwenden, warum es ein verdammt abgefahrenes Werkzeug ist und bei welchen Gelegenheiten Sie es besser nicht einsetzen sollten.

### Einfacher Rebase

Wenn Sie sich noch mal ein früheres Beispiel aus [Einfaches Merging](#) anschauen, können Sie sehen, dass Sie Ihre Arbeit verzweigt und Commits auf zwei unterschiedlichen Branches erstellt haben.

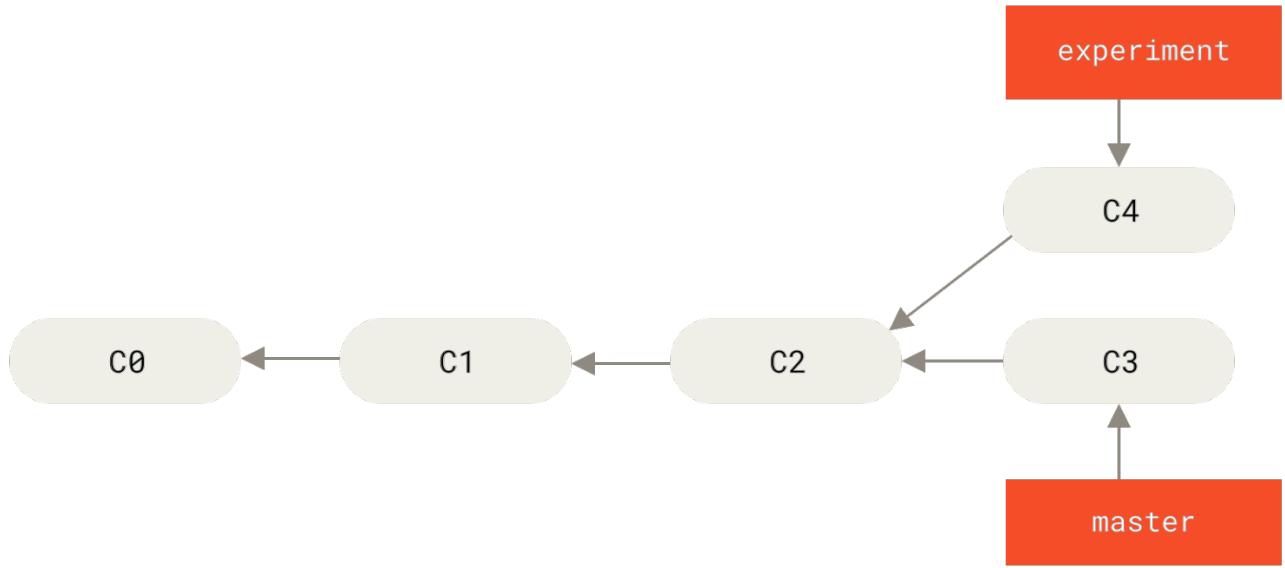


Figure 35. Einfacher verzweigter Verlauf

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

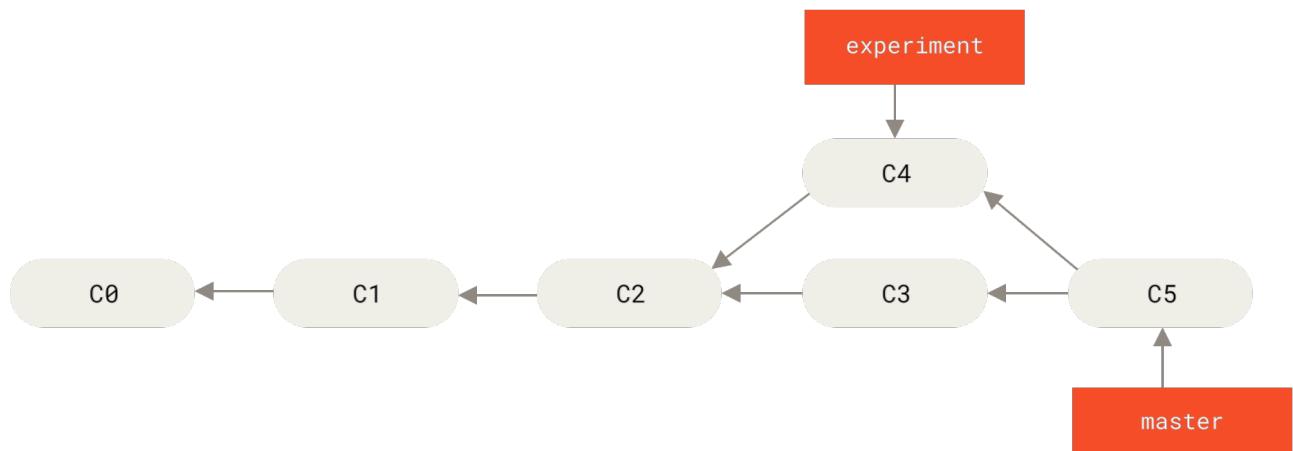


Figure 36. Zusammenführen (Merging) verzweigter Arbeitsverläufe

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

Für dieses Beispiel würden Sie den Branch `experiment` auschecken und dann wie folgt auf den Branch `master` neu ausrichten (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 Vorfahren der beiden Branches (der, auf dem Sie arbeiten, und jener, auf den Sie *rebase* möchten) geht, dann die Informationen zu den Änderungen (diffs) sammelt, welche seit dem 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 Sie *rebase* möchten und dann alle Änderungen erneut durchführt.

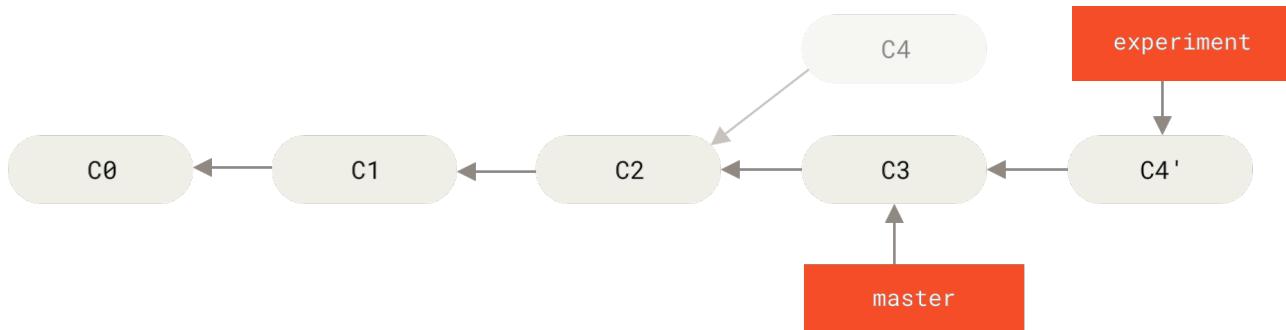


Figure 37. Rebase der in C4 eingeführten Änderung auf C3

An diesem Punkt können Sie zurück zum `master`-Branch wechseln und einen fast-forward-Merge durchführen.

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

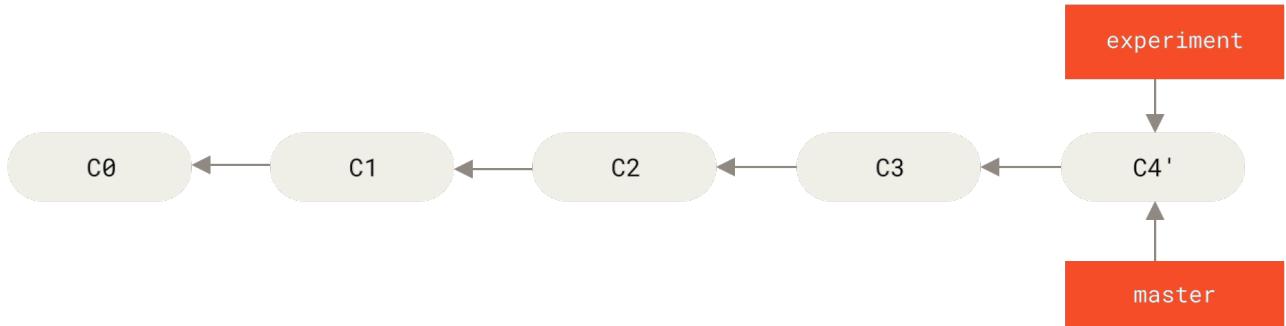


Figure 38. Vorspulen (fast-forwarding) des master-Branche

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, aber das Rebase sorgt für eine klareren Verlauf. Wenn man das Protokoll eines rebasierten Branchs betrachtet, sieht es aus wie eine lineare Historie: Es scheint, dass alle Arbeiten in Serie stattgefunden hätten, auch wenn sie ursprünglich parallel stattgefunden haben.

Häufig werden Sie das anwenden, damit Ihre Commits sauber auf einen Remote-Zweig angewendet werden – vielleicht in einem Projekt, zu dem Sie beitragen möchten, das Sie aber nicht pflegen. In diesem Fall würden Sie Ihre Arbeiten in einem Branch erledigen und im Anschluss Ihre Änderungen mittels Rebase zu `origin/master` hinzufügen, wenn Sie soweit sind, Ihre Patches dem Hauptprojekt zu übermitteln. Auf diese Weise muss der Maintainer keine Integrationsarbeiten durchführen – nur einen „fast-forward“ oder eine saubere Anwendung Ihres Patches.

Beachten Sie, dass der Snapshot, auf welchen der letzte Commit zeigt, ob es nun der letzte der Rebase-Commits nach einem Rebase 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, wohingegen beim Mergen die beiden Endpunkte der Branches genommen und miteinander verschmolzen werden.

## Weitere interessante Rebases

Sie können Ihr Rebase auch auf einen anderen Branch als den Rebase-Ziel-Branch anwenden. Nehmen Sie zum Beispiel einen Verlauf wie im Bild: [Ein Verlauf mit einem Themen-Branch neben einem anderen Themen-Branch](#). Sie haben einen Themen-Branch (`server`) angelegt, um ein paar serverseitige Funktionalitäten zu Ihrem Projekt hinzuzufügen, und haben dann einen Commit gemacht. Dann haben Sie von diesem einen weiteren Branch abgezweigt, um clientseitige Änderungen (`client`) vorzunehmen, und haben ein paar Commits durchgeführt. Zum Schluss wechselten Sie wieder zurück zu Ihrem `server`-Branch und machten ein paar weitere Commits.

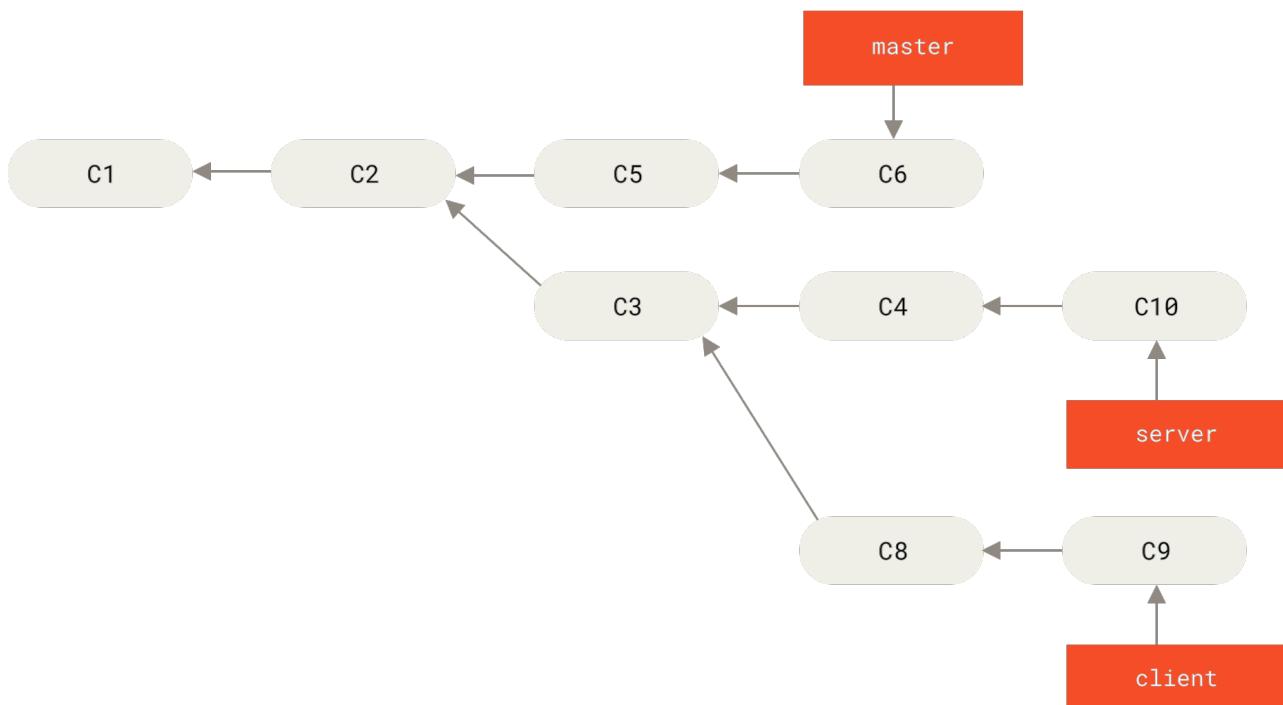


Figure 39. Ein Verlauf mit einem Themen-Branch neben einem anderen Themen-Branch

Angenommen, Sie entscheiden sich, dass Sie für einen Release Ihre clientseitigen Änderungen mit Ihrer Hauptentwicklungslinie zusammenführen, während Sie die serverseitigen Änderungen noch zurückhalten wollen, bis diese weiter getestet wurden. Sie können einfach die Änderungen am `client`-Branch (`C8` und `C9`), die nicht auf `server`-Branch sind, nehmen und mit der Anweisung `git rebase` zusammen mit der Option `--onto` erneut auf den `master`-Branch anwenden:

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

Das bedeutet im Wesentlichen, „Checke den `client`-Branch aus, finde die Patches des gemeinsamen Vorgängers von `client`- und `server`-Branches heraus und wende sie erneut auf den `master`-Branch

an.“ Das ist ein wenig komplex, aber das Resultat ist ziemlich toll.

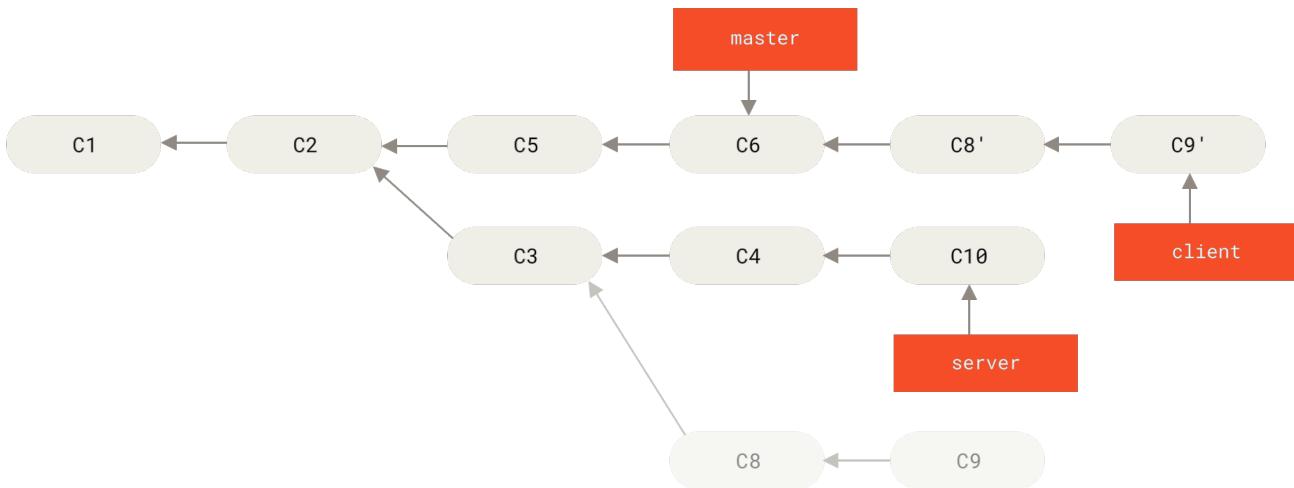


Figure 40. Rebasing eines Themen-Branches aus einem anderen Themen-Branches

Jetzt können Sie Ihren Master-Branch vorrspulen (eng. fast-forward) (siehe [Vorspulen Ihres master-Branches zum Einfügen der Änderungen des client-Branches](#)):

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

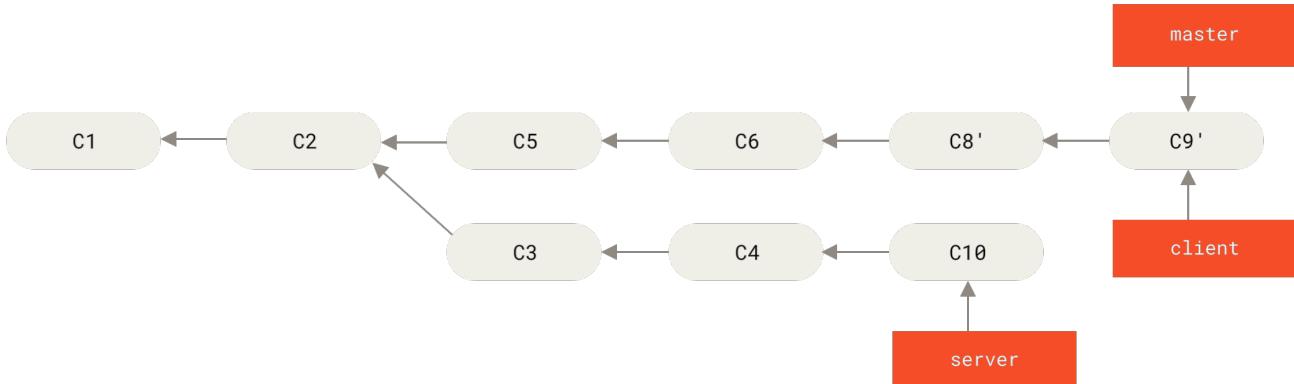


Figure 41. Vorspulen Ihres master-Branches zum Einfügen der Änderungen des client-Branches

Lassen Sie uns annehmen, Sie entscheiden sich dazu, Ihren server-Branch ebenfalls einzupflegen. Sie können das Rebase des server-Branchs auf den `master`-Branch anwenden, ohne diesen vorher auschecken zu müssen, indem Sie die Anweisung `git rebase [Basis-Branch] [Themen-Branch]` ausführen, welche für Sie den Themen-Branch auscheckt (in diesem Fall `server`) und ihn auf dem Basis-Branch (`master`) wiederholt:

```
$ git rebase master server
```

Das wiederholt Ihre Änderungen aus dem `server`-Branch an der Spitze des `master`-Branches, wie in [Rebase Ihres server-Branches an der Spitze Ihres master-Branches](#) gezeigt wird.

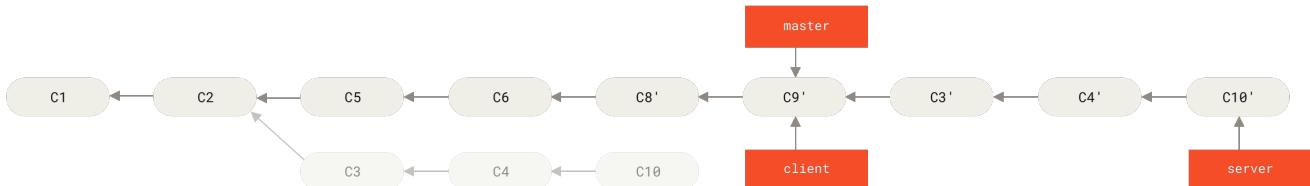


Figure 42. Rebasing Ihres server-Branches an der Spitze Ihres master-Branches

Dann können Sie den Basis-Branch (`master`) vorspalten (engl. fast-forward):

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

Sie können die `client` und `server` Branches entfernen, da die ganze Arbeit bereits integriert in `master` wurde und Sie diese nicht mehr benötigen. Ihr Verlauf für diesen gesamten Prozess sieht jetzt aus wie in [Endgültiger Commit-Verlauf](#):

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

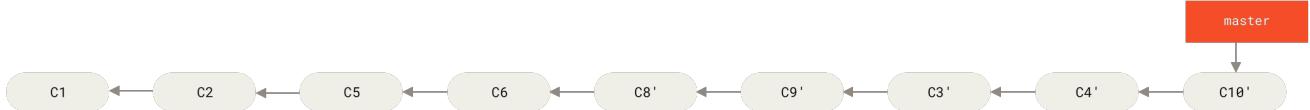


Figure 43. Endgültiger Commit-Verlauf

## Die Gefahren des Rebasing

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

**Führen Sie keinen Rebase mit Commits durch, die außerhalb Ihres Repositorys existieren.**

Wenn Sie sich an diese Leitlinie halten, wird es Ihnen besser gehen. Wenn Sie es nicht tun, werden die Leute Sie hassen, und Sie werden von Freunden und Familie verachtet werden.

Wenn Sie ein Rebase durchführen, heben Sie bestehende Commits auf und erstellen stattdessen neue, die zwar ähnlich aber dennoch unterschiedlich sind. Wenn Sie Commits irgendwohin hochladen und andere ziehen sich diese herunter und nehmen sie als Grundlage für ihre Arbeit, dann müssen Ihre Partner ihre Arbeit jedesmal erneut zusammenführen, sobald Sie Ihre Commits mit einem `git rebase` überschreiben und wieder hochladen. Und richtig chaotisch wird es, wenn Sie versuchen, deren Arbeit in Ihre eigene zu integrieren.

Schauen wir uns ein Beispiel an, wie ein Rebase von von Arbeiten, die Sie öffentlich gemacht haben, Probleme verursachen kann. Angenommen, Sie klonen von einem zentralen Server und arbeiten dann daran. Ihr Commit-Verlauf sieht aus wie dieser:

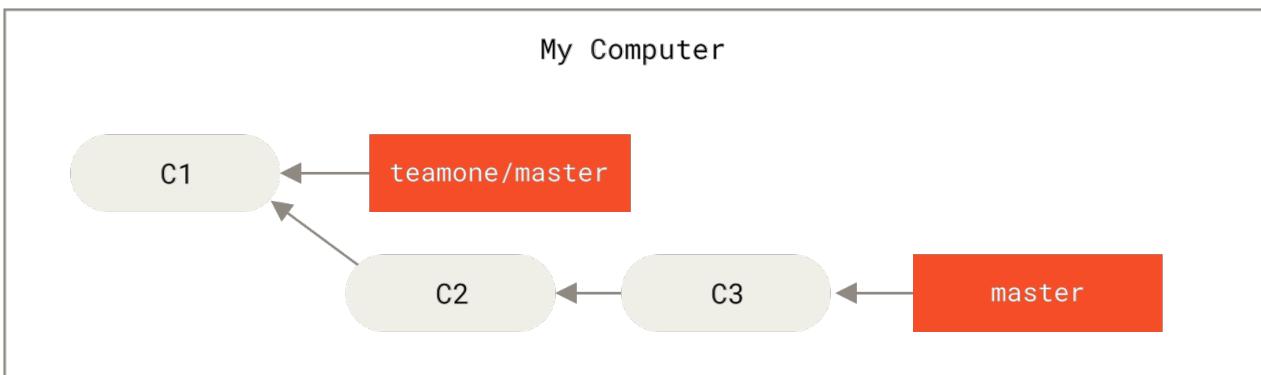
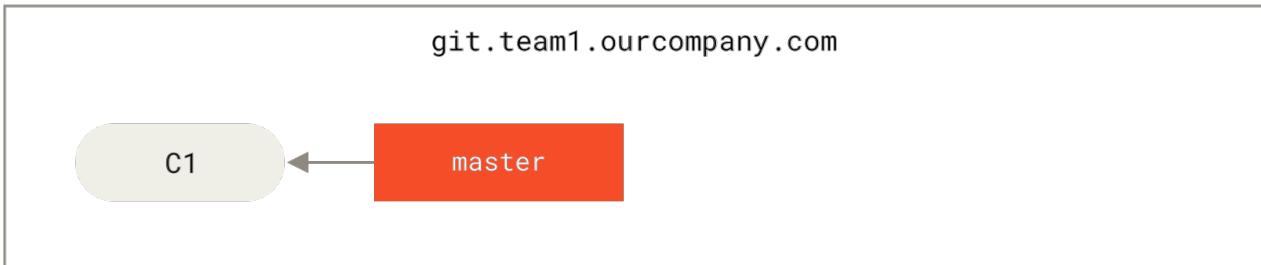


Figure 44. Klonen eines Repositorys und darauf Arbeit aufbauen

Jetzt erledigt jemand anderes eine weitere Arbeit, die einen Merge einschließt, und pusht diese Arbeit auf den zentralen Server. Sie holen die Änderungen ab und mergen den neuen Remote-Branch mit Ihrer Arbeit zusammen, sodass Ihr Verlauf wie so aussieht.

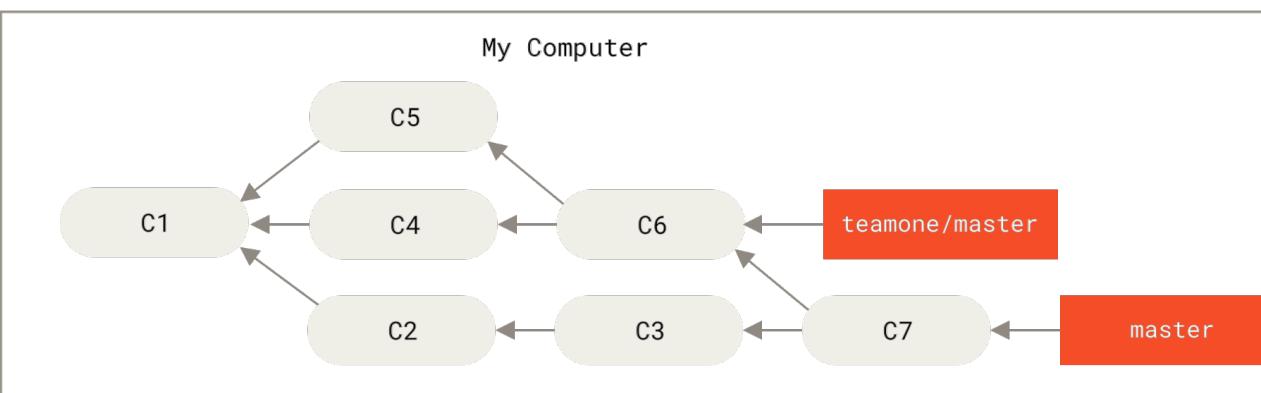
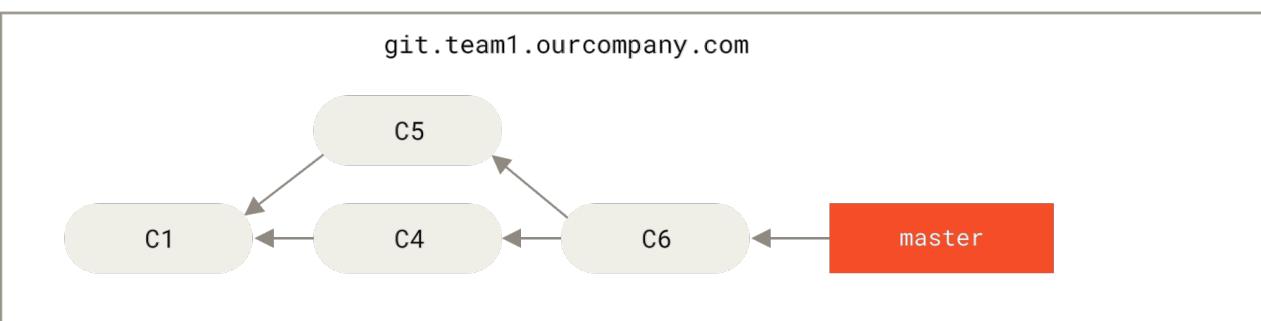


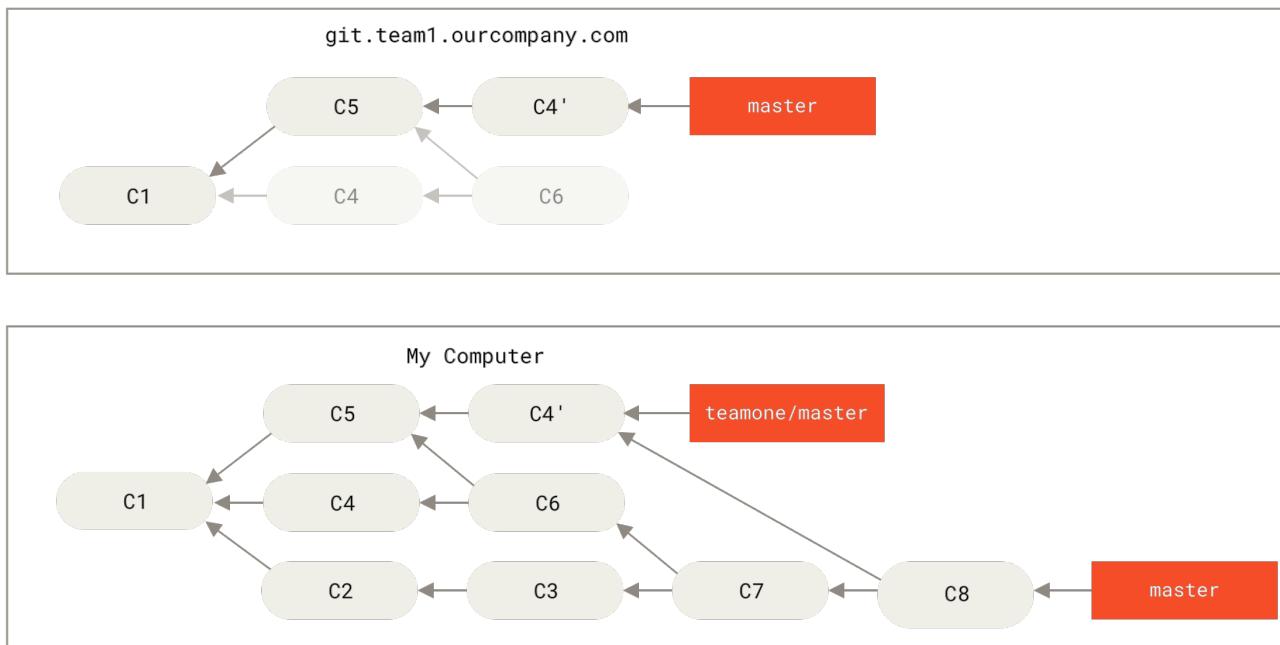
Figure 45. weitere Commits abholen und mergen mit Ihrer Arbeit

Als nächstes entscheidet sich die Person, welche die zusammengeführte Arbeit hochgeladen hat,

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

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

Jetzt sitzen Sie beide in der Klemme. Wenn Sie ein `git pull` durchführen, würden Sie einen Merge-Commit erzeugen, welcher beide Entwicklungslinien einschließt, und Ihr Repository würde so aussehen:



*Figure 47. Sie lassen die Änderungen nochmals in die selbe Arbeit einfließen in einen neuen Merge-Commit*

Falls Sie ein `git log` ausführen, wenn Ihr Verlauf so aussieht, würden Sie zwei Commits sehen, bei denen Autor, Datum und Nachricht übereinstimmen, was verwirrend sein würde. Weiter würden Sie, wenn Sie diesen Verlauf zurück auf den Server pushen, alle diese vom Rebase stammenden Commits auf dem zentralen Server neu einführen, was die Leute noch weiter durcheinander bringen kann. 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 Sie Rebase durchführen

Wenn Sie sich in einer solchen Situation **befinden**, hat Git eine weitere magische Funktion, die Ihnen helfen könnte. Wenn jemand in Ihrem Team gewaltsam Änderungen vorantreibt, die Arbeiten überschreiben, auf denen Sie basiert haben, besteht Ihre Herausforderung darin, herauszufinden, was Ihnen gehört und was andere überschrieben haben.

Es stellt sich heraus, dass Git neben der SHA-1-Prüfsumme auch eine Prüfsumme berechnet, die nur auf dem mit dem Commit eingeführten Patch basiert. Das nennt man eine „patch-id“.

Wenn Sie die neu geschriebene Arbeit pullen und sie mit einem Rebase auf die neuen Commits

Ihres Partners umstellen, kann Git oft erfolgreich herausfinden, was allein von Ihnen ist und kann sie wieder auf die neue Branch anwenden.

Wenn wir im vorhergehenden Szenario beispielsweise bei [Jemand lädt Commits nach einem Rebase hoch und verwirft damit Commits, auf denen Ihre Arbeit basiert](#) die Anweisung `git rebase teamone/master` ausführen, 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 (bloß C2 und C3, da C4 der selbe Patch wie C4' ist)
- diese Commits an der Spitze des Branches `teamone/master` anwenden

Statt des Ergebnisses, welches wir in [Sie lassen die Änderungen nochmals in die selbe Arbeit einfließen in einen neuen Merge-Commit](#) sehen, würden wir etwas erhalten, was irgendwie mehr so wäre wie [Rebase an der Spitze von Änderungen eines „force-pushed“-Rebase..](#)

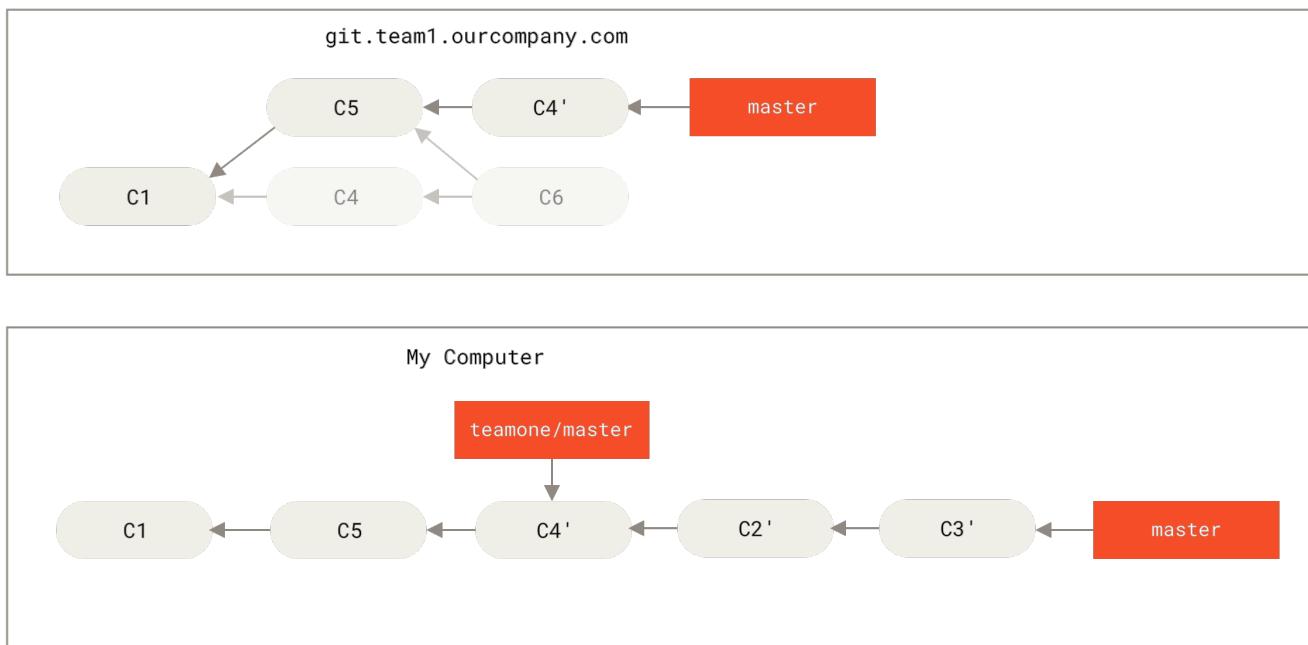


Figure 48. Rebase an der Spitze von Änderungen eines „force-pushed“-Rebase.

Das funktioniert nur, wenn es sich bei C4 und C4', welche Ihr Teamkollege gemacht hat, um fast genau den selben Patch handelt. Andernfalls kann die Datenbank nicht erkennen, dass es sich um ein Duplikat handelt und fügt einen weiteren C4-ähnlichen Patch hinzu (der wahrscheinlich nicht sauber angewendet wird, da die Änderungen bereits oder zumindest teilweise vorhanden wären).

Sie können das auch vereinfachen, indem Sie ein `git pull --rebase` anstelle eines normalen `git pull` verwenden. Oder Sie könnten es manuell mit einem `git fetch` machen, in diesem Fall gefolgt von einem `git rebase teamone/master`.

Wenn Sie `git pull` benutzen und `--rebase` zur Standardeinstellung machen wollen, können Sie den `pull.rebase`-Konfigurationswert mit etwas wie `git config --global pull.rebase true` einstellen.

Wenn Sie nur Commits rebasieren, die noch nie Ihren eigenen Computer verlassen haben, wird es

Ihnen gut gehen. Wenn Sie Commits, die gepusht wurden, aber niemand sonst hat, basierend auf den Commits, rebast, werden Sie auch in Ordnung sein. Wenn Sie Commits, die bereits veröffentlicht wurden, rebasen und Leute die Arbeit auf diesen Commits basieren, dann werden Sie vielleicht frustrierende Probleme und die Verachtung Ihrer Teamkollegen haben.

Wenn Sie oder ein Partner es irgendwann für unbedingt notwendig halten, stellen Sie sicher, dass jeder weiß, dass er anschließend `git pull --rebase` laufen lassen muss. So kann er versuchen den Schaden einzugrenzen, nachdem es passiert ist um alles etwas einfacher zu machen.

## Rebase vs. Merge

Nachdem Sie jetzt Rebasing und Merging in Aktion erlebt haben, fragen Sie sich vielleicht, welches davon besser ist. Bevor wir das beantworten können, lassen Sie uns ein klein wenig zurückblicken und darüber reden, was der Verlauf bedeutet.

Ein Standpunkt ist, dass der Commit-Verlauf Ihres 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 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 beibehalten.

Der entgegengesetzte Standpunkt ist, dass die Commit-Historie die **Story ist, in der beschrieben wird, wie das Projekt realisiert wurde**. Sie würden die erste Fassung eines Buches nicht veröffentlichen, und das Handbuch für die Wartung Ihrer Software verdient eine sorgfältige Bearbeitung. Das ist das Lager, das Tools wie rebase und filter-branch verwendet, um die Story so zu erzählen, wie es für zukünftige Nutzer am sinnvollsten ist.

Nun zur Frage, ob das Mergen oder Rebasen besser ist: hoffentlich werden Sie bemerken, dass es nicht so einfach ist. Git ist ein mächtiges Werkzeug und ermöglicht es Ihnen, viele Dinge mit und an Ihrer eigenen Entwicklung vorzunehmen, aber jedes Team und jedes Projekt ist anders. Jetzt, da Sie wissen, wie diese beiden Möglichkeiten funktionieren, liegt es an Ihnen, zu entscheiden, welche für Ihre spezielle Situation das Beste ist.

Für gewöhnlich lassen sich die Vorteile von beiden Techniken nutzen, indem Sie lokale Änderungen, die Sie gemacht aber noch nicht veröffentlicht haben, mittels Rebase hinzufügen, um den Verlauf zu bereinigen, aber niemals ein Rebase an Commits durchführen, die Sie bereits irgendwohin hochgeladen haben.

## Zusammenfassung

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

# Git auf dem Server

An dieser Stelle sollten Sie in der Lage sein, die meisten der täglichen Aufgaben zu erledigen, für die Sie Git verwenden werden. Um jedoch in Git zusammenarbeiten zu können, benötigen Sie ein externes Git-Repository. Obwohl Sie, technisch gesehen, Änderungen an und aus den individuellen Repositories verschieben können, ist das nicht empfehlenswert, da Sie sich ziemlich leicht irren könnten, woran sie arbeiten, wenn Sie nicht vorsichtig sind. Darüber hinaus ist es vorteilhaft, dass Ihre Mitarbeiter auch dann auf das Repository zugreifen können, wenn Ihr 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 bestimmen Sie, welche Protokolle Ihr 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 Sie Ihren Server mit diesen Protokollen zum Laufen bringen. Zuletzt werden wir ein paar gehostete Optionen durchgehen, wenn es Ihnen nichts ausmacht, Ihren Code auf dem Server eines anderen zu hosten und Sie nicht den Aufwand der Einrichtung und Wartung Ihres eigenen Servers auf sich nehmen wollen.

Wenn Sie keinen eigenen Server betreiben möchten, können Sie zum letzten Abschnitt dieses Kapitels springen, um einige Optionen zum Einrichten eines gehosteten Kontos zu finden und dann 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 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 aus dem Inhalt des `.git` Verzeichnisses Ihres Projekts und nichts anderem.

## 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 Sie sie verwenden könnten (oder nicht sollten).

### 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 Ihrem 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 Ihre Code-Repository-Instanzen auf demselben Computer befinden würden, was einen katastrophalen Verlust viel wahrscheinlicher macht.

Wenn Sie ein gemeinsam genutztes gemountetes Dateisystem haben, können Sie ein lokales dateibasiertes Repository klonen, dort hin verschieben (engl. push to) und daraus ziehen (engl.

pull). Verwenden Sie den Pfad zum Repository als URL, um ein solches Repository zu klonen oder einem vorhandenen Projekt ein Remote-Repository hinzuzufügen. Um beispielsweise ein lokales Repository zu klonen, können Sie Folgendes ausführen:

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

oder auch das:

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

Git funktioniert etwas anders, wenn Sie **file://** explizit am Anfang der URL angeben. Wenn Sie nur den Pfad angeben, versucht Git, Hardlinks zu verwenden oder die benötigten Dateien direkt zu kopieren. Wenn Sie **file://** angeben, 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 Sie eine saubere Kopie des Repositorys mit fremden Referenzen oder weggelassenen Objekten wünschen – 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 so vorgegangen werden:

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

Dann können Sie über Ihren neuen Remote-Namen **local\_proj** auf dieses Remote-Repository pushen und von dort abrufen, als ob Sie dies über ein Netzwerk tun würden.

## Vorteile

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

Das ist auch eine elegante Möglichkeit, um schnell Arbeiten aus dem Arbeits-Repository eines anderen zu holen. Wenn Sie und ein Mitarbeiter am gleichen Projekt arbeiten und Sie etwas überprüfen möchten, 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

Sie von zu Hause mit Ihrem Laptop aus pushen möchten, müssen Sie das entfernte Verzeichnis einhängen (engl. mounten), was im Vergleich zum netzwerkbasierten Zugriff schwierig und langsamer sein kann.

Es ist wichtig zu erwähnen, dass es nicht unbedingt die schnellste Option ist, wenn Sie einen gemeinsamen Mount verwenden. Ein lokales Repository ist nur dann schnell, wenn Sie schnellen Zugriff auf die Daten haben. Ein Repository auf NFS-Mounts ist oft langsamer als der Zugriff auf das Repository über SSH auf demselben Server, während Git lokale Festplatten in jedem System nutzt.

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, was bedeutet, dass es für den Benutzer oft einfacher ist als so etwas wie SSH, da Sie Eingaben wie Benutzername/Passwort-Authentifizierung verwenden können, anstatt SSH-Schlüssel einrichten zu müssen.

Es ist wahrscheinlich der beliebteste Weg, heute 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, können Sie nun eine einzige URL für beides verwenden. Wenn Sie versuchen, 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 Sie verwenden, um das Repository online anzuzeigen (z.B. <https://github.com/schacon/simplegit>), die gleiche URL, mit der Sie klonen und, wenn Sie Zugriff haben, verschieben können.

### 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 es vom Webserver wie normale Dateien handelt wird. Das Schöne an Dumb HTTP ist die Einfachheit der Einrichtung. Im Grunde genommen müssen Sie nur ein leeres Git-Repository

unter Ihre HTTP-Dokument-Root legen und einen bestimmten **post-update** Hook einrichten, und schon sind Sie fertig (siehe [Git Hooks](#)). Ab diesem Zeitpunkt kann jeder, der auf den Webserver zugreifen kann, unter dem Sie das Repository ablegen, auch Ihr Repository klonen. Um Lesezugriff auf Ihr Repository über HTTP zu ermöglichen, gehen Sie 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 das HTTP-Abrufen und -Kloning ordnungsgemäß zu ermöglichen. Dieser Befehl wird ausgeführt, wenn Sie in dieses Repository pushen (vielleicht über SSH); dann können andere Leute klonen über so etwas wie:

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

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

Im Allgemeinen würden Sie sich entweder einen Smart HTTP-Server zum Lesen und Schreiben betreiben oder die Dateien einfach als schreibgeschützt im Dumb-Modus zur Verfügung stellen. Seltener wird ein Mix aus beiden Diensten angeboten.

## Vorteile

Wir werden 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.

Sie können Ihre Repositorys auch schreibgeschützt über HTTPS bereitstellen, d.h. Sie können die Inhaltsübertragung verschlüsseln oder Sie können sogar so weit gehen, dass Clients bestimmte signierte SSL-Zertifikate verwenden müssen.

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

## 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 Sie HTTP für authentifiziertes Pushen verwenden, ist die Bereitstellung Ihrer Anmeldeinformationen manchmal komplizierter als die Verwendung von Schlüsseln über SSH. Es gibt jedoch mehrere Tools zum Zwischenspeichern von Berechtigungen, die Sie verwenden könnten, darunter Keychain-Zugriff auf MacOS und Credential Manager unter Windows, um das ziemlich zu Vereinfachen. Lesen Sie den Abschnitt [Credential Storage](#), um zu erfahren, wie Sie ein sicheres HTTP-Passwort-Caching auf Ihrem System einrichten können.

## SSH Protocol

Ein gängiges Transportprotokoll für Git, wenn das Self-Hosting über SSH läuft. 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, können Sie eine entsprechende `ssh://` URL angeben:

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

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

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

Wenn Sie in beiden Fällen oben keinen optionalen Benutzernamen angeben, benutzt Git den User, mit dem Sie aktuell angemeldet sind.

## 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 Ihr Git-Repository unterstützt. Wenn Sie SSH verwenden, müssen Benutzer über einen SSH-Zugriff auf Ihren Computer verfügen, auch wenn sie nur über Lesezugriff verfügen. Das macht SSH in Open Source-Projekten ungeeignet, wenn, möglicherweise, die Benutzer Ihr Repository einfach nur klonen möchten, um es zu überprüfen. Wenn Sie es nur in Ihrem Unternehmensnetzwerk verwenden, ist SSH möglicherweise das einzige Protokoll, mit dem Sie sich befassen müssen. Wenn Sie anonymen schreibgeschützten

Zugriff auf Ihre Projekte und die Verwendung von SSH zulassen möchten, müssen Sie SSH einrichten, damit Sie Push-Vorgänge ausführen können, aber noch zusätzliche Optionen damit andere Benutzer auch abrufen können.

## Git Protokoll

Und schließlich haben wir das Git-Protokoll. Es ist ein spezieller Daemon, der mit Git ausgeliefert wird, der auf einem dedizierten Port (9418) lauscht und der einen Dienst bereitstellt, ähnlich dem des SSH-Protokolls, aber ohne jegliche Authentifizierung. Damit ein Repository über das Git-Protokoll bedient werden kann, müssen Sie eine `git-daemon-export-ok` Datei erstellen – der Daemon wird ohne diese Datei kein Repository bedienen, weil es sonst keine Sicherheit gibt. Entweder ist das Git-Repository für jeden zugänglich, um zu klonen, oder für Keinen. Das bedeutet, dass es in der Regel keinen Push über dieses Protokoll gibt. Sie können den Push-Zugriff aktivieren, aber angesichts der fehlenden Authentifizierung kann jeder im Internet, der die URL Ihres Projekts findet, zu diesem Projekt pushen. Es reicht aus, zu sagen, dass das selten vorkommt.

### Vorteile

Das Git-Protokoll ist oft als erstes Netzwerkübertragungsprotokoll verfügbar. Wenn Sie viel Traffic für ein öffentliches Projekt bereitstellen oder ein sehr großes Projekt, das keine Benutzeroauthentifizierung für den Lesezugriff benötigt, dann wollen Sie voraussichtlich einen Git-Daemon einrichten, der Ihr Projekt unterstützt. Er verwendet den gleichen Datenübertragungsmechanismus wie das SSH-Protokoll, jedoch ohne den Aufwand für Verschlüsselung und Authentifizierung.

### Nachteile

Der Nachteil des Git-Protokolls ist die fehlende Authentifizierung. Es ist generell nachteilig, wenn das Git-Protokoll der einzige Zugang zu Ihrem Projekt ist. Im Allgemeinen werden Sie es mit einem SSH- oder HTTPS-Zugriff für die wenigen Entwickler koppeln, die Push (Schreib-)Zugriff haben und alle anderen `git://` für den Lesezugriff verwenden lassen. Es ist wahrscheinlich auch das am schwierigsten einzurichtende Protokoll. Es muss seinen eigenen Daemon laufen lassen, der eine `xinetd` oder `systemd` Konfiguration oder dergleichen erfordert, was nicht immer ein Park-Spaziergang ist. Es erfordert auch einen Firewall-Zugang auf Port 9418, der kein Standardport ist, den Unternehmens-Firewalls immer 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 Ihrem eigenen Server ausführt.



Hier zeigen wir Ihnen die Befehle und Schritte, die für die einfache, vereinfachte Installation auf einem Linux-basierten Server erforderlich sind, aber es ist auch möglich, diese Dienste auf MacOS- oder Windows-Servern auszuführen. Die tatsächliche Einrichtung eines Productionsservers innerhalb Ihrer Infrastruktur wird sicherlich Unterschiede in Bezug auf Sicherheitsmaßnahmen oder Betriebssystemwerkzeuge mit sich bringen, aber hoffentlich gibt Ihnen das hier einen Überblick darüber, worum es geht.

Um einen Git-Server einzurichten, müssen Sie ein bestehendes Repository in ein neues Bare-Repository exportieren – ein Repository, das kein Arbeitsverzeichnis enthält. Das ist im Allgemeinen einfach zu realisieren. Um Ihr Repository zu klonen, um ein neues nacktes Repository zu erstellen, führen Sie 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.
```

Sie sollten nun eine Kopie der Git-Verzeichnisdaten in Ihrem `my_project.git` Verzeichnis haben.

Das ist ungefähr so etwas wie

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

Es gibt ein paar kleine Unterschiede in der Konfigurationsdatei, aber für Ihren 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 Sie eine leere Kopie Ihres Repositories haben, müssen Sie es nur noch auf einen Server legen und Ihre Protokolle einrichten. Nehmen wir an, Sie haben einen Server mit der Bezeichnung `git.example.com` eingerichtet, auf den Sie SSH-Zugriff haben und Sie möchten alle Ihre Git-Repositorys unter dem Verzeichnis `/srv/git` speichern. Angenommen, `/srv/git` existiert bereits auf diesem Server, dann können Sie Ihr neues Repository einrichten, indem Sie Ihr leeres Repository kopieren:

```
$ 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, Ihr 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 Sie den Befehl `git init` mit der Option `--shared` ausführen. Beachten Sie, dass Sie durch die Ausführung dieses Befehls keine Commits, Referenzen usw. im laufenden Prozess zerstören werden.

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

Sie sehen, wie einfach es ist, ein Git-Repository zu übernehmen, eine leere Version zu erstellen und sie auf einem Server zu platzieren, auf den Sie und Ihre Mitarbeiter SSH-Zugriff haben. Jetzt sind Sie in der Lage, am gleichen Projekt mitzuarbeiten.

Es ist wichtig zu wissen, dass dies buchstäblich alles ist, was Sie tun müssen, um einen brauchbaren Git-Server zu betreiben, auf den mehrere Personen Zugriff haben – fügen Sie einfach SSH-fähige Konten auf einem Server hinzu und legen Sie ein leeres Repository an einen Ort, auf das alle diese Benutzer Lese- und Schreibrechte haben. Sie sind startklar – mehr ist nicht nötig.

In den nächsten Abschnitten erfahren Sie, wie Sie das zu komplexeren Konfigurationen erweitern können. Diese Betrachtung beinhaltet, dass man nicht für jeden Benutzer ein Benutzerkonto anlegen muss, öffentlichen Lesezugriff auf Repositories hinzufügen und Web-UIs einrichten kann und vieles mehr. Denken Sie 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 Sie ein kleines Team sind, Git nur in Ihrer Umgebung ausprobieren wollen und nur wenige Entwickler haben, kann es ganz einfach sein. Einer der kompliziertesten Aspekte bei der Einrichtung eines Git-Servers ist die Benutzerverwaltung. Wenn Sie möchten, dass einige Repositories für bestimmte Benutzer schreibgeschützt und für andere lesend und schreibend sind, können Zugriff und Berechtigungen etwas schwieriger zu realisieren sein.

### SSH-Zugang

Wenn Sie einen Server haben, auf dem alle Ihre Entwickler bereits SSH-Zugriff haben, ist es in der Regel am einfachsten, dort Ihr erstes Repository einzurichten, da Sie so gut wie keine zusätzlichen Einstellungen vornehmen müssen (wie wir im letzten Abschnitt beschrieben haben). Wenn Sie komplexere Zugriffsberechtigungen für Ihre Repositories benötigen, können Sie diese mit den normalen Dateisystemberechtigungen des Betriebssystems Ihres Servers verwalten.

Wenn Sie Ihre Repositories auf einem Server platzieren möchten, der nicht über Konten für alle Personen in Ihrem Team verfügt, denen Sie Schreibzugriff gewähren möchten, müssen Sie für sie einen SSH-Zugriff einrichten. Wir gehen davon aus, dass auf Ihrem Server bereits ein SSH-Server installiert ist und Sie auf diesen Server zugreifen können.

Es gibt einige Möglichkeiten, wie Sie jedem in Ihrem Team Zugang gewähren können. Die erste besteht darin, Konten für alle einzurichten, was unkompliziert ist, aber schwerfällig 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 Methode besteht darin, ein einzelnes Git-Benutzerkonto auf der Maschine zu erstellen, jeden Benutzer, der Schreibrechte haben soll, aufzufordern, Ihnen einen öffentlichen SSH-Schlüssel zu senden, und diesen Schlüssel zur Datei `~/.ssh/authorized_keys` dieses neuen Git-Kontos hinzuzufügen. Zu dem Zeitpunkt kann jeder über das Git-Konto auf diese Maschine zugreifen. Das hat keinen Einfluss auf die Commit-Daten – den SSH-Benutzer, den Sie anmelden, und auch nicht auf die Commits, die Sie gespeichert haben.

Eine weitere Möglichkeit besteht darin, dass sich Ihr SSH-Server von einem LDAP-Server oder einer anderen zentralen Authentifizierungsquelle authentifiziert, die Sie möglicherweise bereits eingerichtet haben. 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-Schlüssel. Um einen öffentlichen Schlüssel bereitzustellen, muss jeder Benutzer in Ihrem System selbst einen generieren, falls er noch keinen hat. Der Ablauf ist für alle Betriebssysteme gleich. Zuerst sollten Sie überprüfen, ob Sie noch keinen Schlüssel haben. Standardmäßig werden die SSH-Schlüssel eines Benutzers im Verzeichnis `~/.ssh` dieses Benutzers gespeichert. Sie können leicht nachsehen, ob Sie bereits über einen Schlüssel verfügen, indem Sie in dieses Verzeichnis gehen und den Inhalt auflisten:

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

Suchen Sie ein Datei-Paar mit dem Namen `id_dsa` oder `id_rsa` und eine entsprechende Datei mit der Erweiterung `.pub`. Die `.pub`-Datei ist Ihr öffentlicher Schlüssel, und die andere Datei ist der zugehörige private Schlüssel. Wenn Sie diese Dateien nicht haben (oder nicht einmal ein `.ssh`-Verzeichnis vorhanden ist), können Sie sie erstellen, indem Sie ein Programm namens `ssh-keygen` ausführen, 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 Sie leer lassen können, wenn Sie beim Verwenden des Schlüssels kein Passwort eingeben möchten. Wenn Sie jedoch ein Passwort verwenden, fügen Sie die Option `-o` hinzu; sie speichert den privaten Schlüssel in einem Format, das resistenter gegen Brute-Force-Passwortcracking ist als das Standardformat. Sie können auch das `ssh-agent` Tool verwenden, um zu vermeiden, dass Sie das Passwort jedes Mal neu eingeben müssen.

Jetzt muss jeder Benutzer seinen öffentlichen Schlüssel an Sie oder an einen Administrator des Git-Servers senden (vorausgesetzt, Sie verwenden 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 AAAAB3NzaC1yc2EAAAQEAk1OUpkDHrfHY17SbrmTIpNLTGK9Tjom/BWDSU
GPL+nafzlHDTYW7hdI4yZ5ew18JH4JW9jbhUFrvQz7x1ELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyB1WXFCR+HAo3FXRitBqxiX1nKhXpHAZsMcilq8V6RjsNAQwdsdMFVs1VK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUF1jQJKprrx88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnP189ZPmVMLuayrD2cE86Z/i18b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@mylaptop.local
```

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

## Einrichten des Servers

Lassen Sie uns durch die Einrichtung des SSH-Zugriffs auf der Serverseite gehen. In diesem Beispiel verwenden Sie die Methode `authorized_keys` zur Authentifizierung Ihrer Benutzer. Wir nehmen an, dass Sie eine Standard-Linux-Distribution wie Ubuntu verwenden.



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 erstellen Sie 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 müssen Sie einige öffentliche SSH-Schlüssel für Entwickler zur `authorized_keys` Datei für den `git`-User hinzufügen. Nehmen wir an, Sie haben einige vertrauenswürdige öffentliche Schlüssel und haben 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/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1KKI9MAQLMdGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFB1gc+myiv
07TCUSBdLQlgMVOFq1I2uPWQ0kOWQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgfZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Sie fügen 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 können Sie ein leeres Repository für sie einrichten, indem Sie `git init` mit der Option `--bare` ausführen, 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. Beachten Sie, dass jemand auf der Maschine eine Shell ausführen muss und jedes Mal, wenn Sie ein Projekt hinzufügen möchten, ein Bare-Repository erstellen muss. Lassen Sie uns `gitserver` als Hostname für den Server verwenden, auf dem Sie Ihren `git`-Benutzer und Ihr Repository eingerichtet haben. Wenn Sie das intern ausführen und DNS so einrichten, dass `gitserver` auf diesen Server zeigt, dann können Sie die Befehle so verwenden, wie sie wie folgt sind (vorausgesetzt, dass `myproject` ein bestehendes Projekt mit Dateien darin 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 the README file'
$ git push origin master
```

Mit dieser Methode können Sie schnell einen Read/Write Git-Server für eine Handvoll Entwickler in Betrieb nehmen.

Sie sollten beachten, dass sich derzeit alle diese Benutzer auch am Server anmelden und eine Shell als `git`-Benutzer erhalten können. Wenn Sie das einschränken wollen, müssen Sie die Shell zu etwas anderem in der Datei `/etc/passwd` ändern.

Sie können das `git`-Benutzerkonto mit einem in Git enthaltenen Shell-Tool mit dem Namen `git-shell` ganz einfach auf Git-bezogene Aktivitäten beschränken. Wenn Sie diese Option als Anmeldeshell des `git`-Benutzerkontos festlegen, kann dieses Konto keinen normalen Shell-Zugriff auf Ihren Server haben. Um das zu nutzen, geben Sie `git-shell` anstelle von bash oder csh für die Login-Shell dieses Kontos an. Um das zu erreichen, müssen Sie zuerst den vollständigen Pfadnamen des `git-shell`-Befehls zu `/etc/shells` hinzufügen, falls er nicht bereits vorhanden ist:

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

Jetzt können Sie 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 und aber nicht mehr auf der Maschine navigieren. Wenn Sie es versuchen, sehen Sie eine entsprechende Zurückweisung des Logins:

```
$ 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 können Benutzer noch die SSH-Portforwarding verwenden, um auf jeden Host zuzugreifen, den der Git-Server erreichen kann. Wenn Sie dies verhindern möchten, können Sie die Datei `authorized_keys` bearbeiten und jedem Schlüssel, den Sie einschränken möchten, 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/x4lhJA0F3FR1rP6kYBRsWj2aThGw6HXLm9/5zytK6Ztg3RPKK+4kYjh6541N
YsnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdGW1GYEIgS9EzSdf8AcC
IicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtpofwFB1gc+myiv07TCUSBd
LQ1gMV0Fq1I2uPWQ0kOWQAHukE0mfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPqdAv8JggJ
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 abrufen. Wie in der Ausgabe angegeben, können Sie auch ein Verzeichnis im Ausgangsverzeichnis des **git**-Benutzers einrichten, das den **git-shell**-Befehl ein wenig anpasst. Sie können beispielsweise die vom Server akzeptierten Git-Befehle einschränken oder die Nachricht anpassen, die Benutzer sehen, wenn sie versuchen, SSH auf diese Weise auszuführen. Führen Sie **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 Ihre Git-Daten. Denken Sie daran, dass alles, was Sie über dieses Protokoll bereitstellen, innerhalb des Netzwerks öffentlich ist, da dies kein authentifizierter Dienst ist.

Wenn Sie dies auf einem Server außerhalb Ihrer Firewall ausführen, sollte es nur für Projekte verwendet werden, die für die Welt öffentlich sichtbar sind. Wenn sich der Server, auf dem Sie es ausführen, in Ihrer Firewall befindet, können Sie es für Projekte verwenden, auf die eine große Anzahl von Personen oder Computern (Continuous Integration oder Build-Server) nur Lesezugriff haben, wenn Sie dies nicht möchten um jeweils einen SSH-Schlüssel hinzuzufügen.

In jedem Fall ist das Git-Protokoll relativ einfach einzurichten. Grundsätzlich müssen Sie diesen Befehl daemonisiert ausführen:

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

Mit der **--reuseaddr**-Option kann der Server neu gestartet werden, ohne dass das Zeitlimit für alte Verbindungen überschritten wird. Mit der **--base-path** Option können Benutzer Projekte klonen, ohne den gesamten Pfad anzugeben. Der Pfad am Ende teilt dem Git-Dämon mit, wo nach zu exportierenden Repositorys gesucht werden soll. Wenn Sie eine Firewall verwenden, müssen Sie auch an Port 9418 der Box, auf der Sie diese einrichten, ein Loch in die Firewall bohren.

Sie können diesen Prozess auf verschiedene Arten dämonisieren, je nachdem, welches Betriebssystem Sie verwenden.

Da `systemd` das gebräuchlichste Init-System unter modernen Linux-Distributionen ist, können Sie es für diesen Zweck verwenden. Legen Sie 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
```

Sie haben vielleicht bemerkt, dass der Git-Daemon hier mit `git` als Gruppe und Benutzer gestartet wird. Passen Sie es an Ihre Bedürfnisse an und stellen Sie sicher, dass der angegebene Benutzer auf dem System vorhanden ist. Überprüfen Sie auch, ob sich die Git-Binärdatei tatsächlich unter `/usr/bin/git` befindet und ändern Sie gegebenenfalls den Pfad.

Abschließend führen Sie `systemctl enable git-daemon` aus, um den Dienst beim Booten automatisch zu starten, so dass Sie den Dienst mit `systemctl start git-daemon` und `systemctl stop git-daemon` starten und stoppen können.

Auf anderen Systemen können Sie `xinetd` verwenden um ein Skript in Ihrem `sysvinit`-System zu benutzen, oder etwas anderes – solange Sie diesen Befehl aktiviert und irgendwie überwacht bekommen.

Als nächstes müssen Sie Git mitteilen, auf welche Repositorys nicht authentifizierter, serverbasierter Zugriff auf Git möglich sein soll. Sie können das in den einzelnen Repositories tun, indem Sie eine Datei mit dem Namen `git-daemon-export-ok` erstellen.

```
$ 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 zu betreuen.

# Smart HTTP

Wir haben jetzt authentifizierte Zugriff 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 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 seit 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).

Lassen Sie uns durch ein sehr einfaches Setup gehen. Wir werden das mit Apache als CGI-Server einrichten. Wenn Sie kein Apache-Setup haben, können Sie dies auf einer Linux-Box mit so etwas tun:

```
$ 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.

Sie sollten auch die Unix-Benutzergruppe im Verzeichnis `/srv/git` auf `www-data` setzen, damit Ihr Webserver auf die Repositories lesend und schreibend zugreifen kann, da die Apache-Instanz, auf der das CGI-Skript läuft, (standardmäßig) als dieser Benutzer ausgeführt wird:

```
$ 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 in den `/git`-Pfad Ihres Webservers kommt, auszuführen.

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

Wenn Sie die Umgebungsvariable `GIT_HTTP_EXPORT_ALL` weglassen, wird Git nur nicht authentifizierte Clients die Repositorys mit der Datei `git-daemon-export-ok` zur Verfügung stellen, genau wie der Git-Daemon.

Abschließend möchten Sie dem Apache sagen, dass er Anfragen an das `git-http-backend` zulassen soll, damit Schreibvorgänge irgendwie authentifiziert werden, möglicherweise mit einem Auth-Block wie diesem:

```
<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 müssen Sie eine `.htpasswd`-Datei erstellen, die die Passwörter aller gültigen Benutzer enthält. Hier ist ein Beispiel für das Hinzufügen eines „schacon“ Benutzers zur Datei:

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

Es gibt unzählige Möglichkeiten, Benutzer mit Apache zu authentifizieren. Sie müssen eine von ihnen auswählen und implementieren. Das ist nur das einfachste Beispiel, das wir uns vorstellen können. Sie werden dies auch mit ziemlicher Sicherheit über SSL konfigurieren wollen, damit alle Daten verschlüsselt werden.

Wir wollen nicht zu weit in das Konzept der Apache-Konfigurationsspezifikationen eindringen, da Sie möglicherweise einen anderen Server verwenden oder unterschiedliche Authentifizierungsanforderungen haben. Die Idee ist, dass Git mit einem CGI mit dem Namen `git-http-backend` daherkommt, das beim Aufruf alle Vorbereitungen zum Senden und Empfangen von Daten über HTTP trifft. Es implementiert selbst keine Authentifizierung, aber diese kann leicht auf der Ebene des Webservers gesteuert werden, der sie aufruft. Sie können das mit fast jedem CGI-fähigen Webserver tun, also wählen Sie denjenigen, den Sie am besten kennen.



Weitere Informationen zur Konfiguration der Authentifizierung in Apache finden Sie in den Apache-Dokumenten unter: <https://httpd.apache.org/docs/current/howto/auth.html>

## GitWeb

Nun, da Sie über einen einfachen Lese-/Schreibzugriff und Lesezugriff auf Ihr Projekt verfügen, können Sie einen einfachen webbasierten Visualizer einrichten. Git wird mit einem CGI-Skript namens GitWeb geliefert, das manchmal dafür verwendet wird.

The screenshot shows the GitWeb interface for a project. At the top, there's a header with 'projects / .git / summary' and a 'git' logo. Below it is a navigation bar with links like 'summary', 'shortlog', 'log', 'commit', 'commitdiff', and 'tree'. On the right, there are search and refresh buttons. The main content area has sections for 'description', 'owner', and 'last change'. Under 'shortlog', there's a detailed list of commits from June 2014, showing authors like Carlos Martin, Vicent Marti, and Philip Kelley, along with their commit messages and timestamps. The 'tags' section lists various versions from 'v0.11.0' to 'v0.21.0-rc1' with their corresponding commit dates.

Figure 49. Die webbasierte Benutzeroberfläche von GitWeb

Wenn Sie herausfinden möchten, wie GitWeb für Ihr Projekt aussehen würde, gibt Git einen Befehl zum Starten einer temporären Instanz, wenn Sie einen leichten Webserver auf Ihrem System wie `lighttpd` oder `webrick` haben. Auf Linux-Maschinen wird `lighttpd` oft installiert, so dass Sie es möglicherweise zum Laufen bringen können, indem Sie `git instaweb` in Ihr Projektverzeichnis eingeben. Wenn Sie einen Mac verwenden, wird Leopard mit Rubin vorinstalliert geliefert, so dass `webrick` Ihre beste Wahl sein kann. Um `instaweb` mit einem nicht-lighttpd Handler zu starten, können Sie es mit der Option `--httpd` ausführen.

```
$ 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]
```

Das startet einen HTTPD-Server auf Port 1234 und startet dann automatisch einen Webbrowser, der sich auf dieser Seite öffnet. Es ist ziemlich einfach von deiner Seite. Wenn Sie fertig sind und den Server herunterfahren möchten, können Sie den gleichen Befehl mit der Option `--stop` ausführen:

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

Wenn Sie das Web-Interface die ganze Zeit auf einem Server für Ihr Team oder für ein Open-Source-Projekt, das Sie hosten, ausführen möchten, müssen Sie das CGI-Skript so einrichten, dass es von Ihrem normalen Webserver bedient wird. Einige Linux-Distributionen haben ein `gitweb` Paket, das Sie möglicherweise über `apt` oder `dnf` installieren können, so dass Sie das zuerst ausprobieren

sollten. Wir werden die manuelle Installation von GitWeb nur sehr kurz abhandeln. Zuerst müssen Sie den Git-Quellcode, der im Lieferumfang von GitWeb enthalten ist, herunterladen und das benutzerdefinierte CGI-Skript generieren:

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

Beachten Sie, dass Sie dem Befehl mitteilen müssen, wo Sie Ihre Git-Repositories mit der Variablen **GITWEB\_PROJECTROOT** finden können. Nun müssen Sie den Apache dazu bringen, CGI für dieses Skript zu verwenden, zu dem Sie einen VirtualHost hinzufügen können:

```
<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 bedient werden; wenn Sie etwas anderes bevorzugen, sollte es nicht schwierig sein, es einzurichten. An dieser Stelle sollten Sie in der Lage sein, <http://gitserver/> zu besuchen, um Ihre Repositorien online zu betrachten.

## GitLab

GitWeb ist allerdings ziemlich einfach. Wenn Sie nach einem moderneren, voll ausgestatteten Git-Server suchen, gibt es einige Open-Source-Lösungen, die Sie stattdessen installieren können. Da GitLab eines der beliebtesten ist, werden wir uns mit der Installation im Detail befassen und es als Beispiel verwenden. Dies ist etwas komplexer als die GitWeb-Option und erfordert wahrscheinlich 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 wird dieser Prozess sehr gut dokumentiert

und unterstützt.

Es gibt einige Möglichkeiten, wie Sie GitLab installieren können. Um etwas schnell zum Laufen zu bringen, können Sie ein Virtual-Machine-Image oder ein One-Klick-Installationsprogramm von <https://bitnami.com/stack/gitlab> herunterladen und die Konfiguration an Ihre spezielle Umgebung anpassen. Ein netter Punkt, den Bitnami hinzugefügt hat, ist der Anmeldebildschirm (erreichbar durch Eingabe von alt+ →); er sagt Ihnen die IP-Adresse und den standardmäßigen Benutzernamen und das Passwort für das installierte GitLab.



```
*** Welcome to the BitNami Gitlab Stack ***
*** Built using Ubuntu 12.04 - Kernel 3.2.0-53-virtual (tty2). ***

*** You can access the application at http://10.0.1.17 ***
*** The default username and password is 'user@example.com' and 'bitnami1'. ***
*** Please refer to http://wiki.bitnami.com/Virtual_Machines for details. ***

linux login: _
```

Figure 50. Der Anmeldebildschirm für die virtuelle Maschine von Bitnami GitLab

Für alles andere beachten Sie bitte die Hinweise in der Readme-Datei der GitLab Community Edition, die Sie unter <https://gitlab.com/gitlab-org/gitlab-ce/tree/master> finden. Dort finden Sie Hilfe bei der Installation von GitLab mit Kochrezepten, einer virtuellen Maschine auf Digital Ocean und RPM- und DEB-Paketen (die sich derzeit in der Beta-Version befinden). Es gibt auch „inoffizielle“ Anleitungen, wie Sie GitLab mit nicht standardmäßigen Betriebssystemen und Datenbanken zum Laufen bringen können – dazu ein vollständig manuelles Installationsskript und viele andere Inhalte.

## Administration

Die Verwaltungsoberfläche von GitLab wird über das Internet aufgerufen. Benutzen Sie einfach Ihren Browser, um den Hostnamen oder die IP-Adresse, auf der GitLab installiert ist, anzugeben, und melden Sie sich als Admin-Benutzer an. Der Standardbenutzername ist `admin@local.host`, das Standardpassword ist `5iveL!fe` (das Sie nach der Eingabe ändern müssen). Klicken Sie nach der Anmeldung im Menü oben rechts auf das Symbol „Admin-Bereich“.

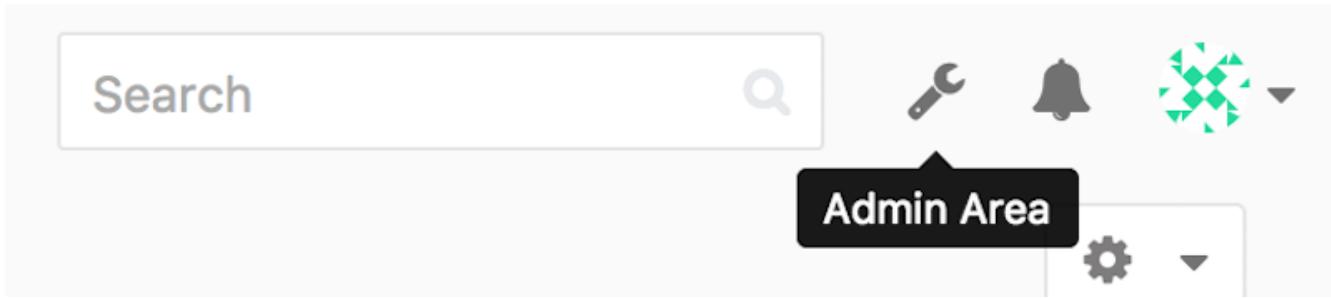


Figure 51. Der „Admin-Bereich“ im GitLab-Menü

## Benutzer

Bei den Anwendern in GitLab handelt es sich um Konten, die Personen zugeordnet sind. Benutzerkonten haben keine große Komplexität; hauptsächlich handelt es sich um eine Sammlung von persönlichen Informationen, die an Login-Daten geknüpft sind. Jedes Benutzerkonto enthält 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>.

Name	Email	Role	Action
Administrator	admin@example.com	Admin	<a href="#">Edit</a>
Betsy Rutherford II	marlin@lednnerlangworth.biz		<a href="#">Edit</a> <a href="#">⋮</a>
Brenden Hayes	laney_dubuque@cormier.biz		<a href="#">Edit</a> <a href="#">⋮</a>
Cassandra Kilback	caterina@beer.com		<a href="#">Edit</a> <a href="#">⋮</a>
Cathryn Leffler DVM	desmond@crooks.ca		<a href="#">Edit</a> <a href="#">⋮</a>
Cecil Medhurst	winnifred@glover.co.uk		<a href="#">Edit</a> <a href="#">⋮</a>
Dr. Joany Fisher	milan@huels.us		<a href="#">Edit</a> <a href="#">⋮</a>
Jazmin Sipes	juliet.turner@leannon.co.uk		<a href="#">Edit</a> <a href="#">⋮</a>

Figure 52. Das Fenster der Benutzerverwaltung von GitLab

Das Entfernen eines Benutzers kann auf zwei Arten erfolgen. Das „Blockieren“ eines Benutzers verhindert, dass er sich am GitLab anmeldet, aber alle Daten unter dem Namensraum dieses Benutzers bleiben erhalten, und 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 natürlich eine viel dauerhaftere und destruktivere Aktion, die kaum angewendet wird.

## Gruppen

Eine GitLab-Gruppe ist eine Zusammenstellung von Projekten, zusammen mit Daten darüber, wie Benutzer auf diese Projekte zugreifen können. Jede Gruppe hat einen Projektnamensraum (genauso wie Benutzer), so dass, wenn die Gruppe `training` ein Projekt `materials` hat, die URL <http://server/training/materials> lautet.

The screenshot shows the GitLab.org interface for managing groups. At the top, there's a navigation bar with 'GitLab.org', a logo, and links for 'Group', 'Activity', 'Labels', 'Milestones', 'Issues 8,501', 'Merge Requests 701', 'Members', and 'Contribution Analytics'. Below the navigation is the group header for '@gitlab-org', featuring the GitLab logo, the group name, a subtitle 'Open source software to collaborate on code', and buttons for 'Leave group' and 'Global'. The main content area displays a list of projects under the heading 'All Projects'. The projects listed are:

- GitLab Development Kit
- kubernetes-gitlab-demo
- omnibus-gitlab
- GitLab Enterprise Edition
- gitlab-shell
- gitlab-ci-multi-runner

Each project entry includes a small icon, the project name, a brief description, and several small circular icons representing different user roles or permissions.

Figure 53. Der Admin-Bildschirm für die Gruppenverwaltung von GitLab.

Jede 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 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 der 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 auch 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 bestimmten Benutzern explizit Zugriff gewähren. Ein *internes* Projekt ist für jeden angemeldeten Benutzer sichtbar, und ein *öffentliches* (engl. *public*) Projekt ist für jeden sichtbar. Beachten Sie, 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 beide führt der GitLab-Server einen HTTP POST mit einem beschreibenden JSON durch, wenn relevante Ereignisse eintreten. Auf diese Weise können Sie Ihre Git-Repositorys und GitLab-Instanzen mit dem Rest Ihrer Entwicklungsverwaltung verbinden, wie z.B. CI-Server, Chatrooms oder Bereitstellungstools.

## Grundlegende Anwendung

Das erste, was Sie mit GitLab anfangen sollten, ist das Erstellen eines neuen Projekts. Dies geschieht durch Anklicken des Symbols „+“ in der Symbolleiste. Sie werden nach dem Namen des Projekts gefragt, zu welchem Namensraum es gehören soll und wie hoch seine Sichtbarkeit sein soll. Das meiste, was Sie hier angeben, ist nicht permanent und kann später über die Einstellungs-Oberfläche neu angepasst werden. Klicken Sie auf „Projekt erstellen“, und Sie sind fertig.

Sobald das Projekt existiert, werden Sie es vermutlich mit einem lokalen Git-Repository verbinden wollen. Jedes Projekt ist über HTTPS oder SSH zugänglich, von denen jede zur Konfiguration einer externen Verbindung mit Git verwendet werden kann. Die URLs sind oben auf der Startseite des Projekts sichtbar. Für ein bestehendes lokales Repository erstellt dieser Befehl einen Remote mit Namen `gitlab` für den gehosteten Standort:

```
$ git remote add gitlab https://server/namespace/project.git
```

Wenn Sie noch keine lokale Kopie des Repositorys haben, können Sie das ganz einfach nachholen:

```
$ git clone https://server/namespace/project.git
```

Die Web-Benutzeroberfläche bietet Zugriff auf mehrere nützliche Anzeigen des Repositorys selbst. Die Homepage jedes Projekts zeigt die letzten Aktivitäten an, und Links oben führen Sie zu Ansichten der Projektdateien und zum Commit-Log.

## Zusammen arbeiten

Die einfachste Art der Zusammenarbeit bei einem GitLab-Projekt besteht darin, einem anderen Benutzer direkten Push-Zugriff auf das Git-Repository zu ermöglichen. Sie können einen Benutzer zu einem Projekt hinzufügen, indem Sie im Abschnitt „Mitglieder“ der Einstellungen dieses Projekts den neuen Benutzer einer Zugriffsebene zuordnen (die verschiedenen Zugriffsebenen werden in den [Gruppen](#) ein wenig erläutert). Indem ein Benutzer eine Zugriffsebene von „Developer“ oder höher erhält, kann dieser Benutzer Commits und Branches direkt und ohne Einschränkung in das Repository verschieben.

Eine weitere, stärker entkoppelte Art der Zusammenarbeit ist die Nutzung von Merge-Anfragen. Diese Funktion ermöglicht es jedem Benutzer, der ein Projekt sehen kann, kontrolliert dazu beizutragen. Benutzer mit direktem Zugriff können einfach einen Branch erstellen, Commits auf ihn verschieben 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 anfordern. 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 der langjährigen Zusammenarbeit in GitLab. Jede Merge-Anfrage ermöglicht eine zeilenweise Diskussion der vorgeschlagenen Änderung (was eine einfache Art der Code-Überprüfung unterstützt), sowie einen allgemeinen Diskussions-Thread. Beide können Benutzern zugeordnet oder in Milestone-Etappen organisiert werden.

Dieser Abschnitt konzentriert sich hauptsächlich auf die Git-bezogenen Funktionen von GitLab, aber als ausgereiftes Programm bietet es viele weitere Funktionen, die Ihnen bei der Teamarbeit helfen, wie Projekt-Wikis und System-Wartungstools. Ein Vorteil für GitLab ist, dass Sie nach der Einrichtung und Inbetriebnahme des Servers selten eine Konfigurationsdatei anpassen oder über SSH auf den Server zugreifen müssen; die überwiegende Verwaltung und allgemeine Nutzung kann über die Browser-Oberfläche erfolgen.

## Von Drittanbietern gehostete Optionen

Wenn Sie nicht alle Arbeiten zur Einrichtung eines eigenen Git-Servers durchführen möchten, haben Sie mehrere Möglichkeiten, Ihre 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 Sie Ihren eigenen Server intern einrichten und betreiben, können Sie dennoch eine öffentliche Hosting-Site für Ihren Open-Source-Code verwenden – es ist im Allgemeinen einfacher für die Open-Source-Community, Sie zu finden und Ihnen zu helfen.

In der heutigen Zeit haben Sie eine große Anzahl von Hosting-Optionen zur Auswahl, jede mit unterschiedlichen Vor- und Nachteilen. Um eine aktuelle Liste zu sehen, besuchen Sie die GitHosting-Seite im Hauptwiki von Git unter [Git Hosting](#).

Wir werden die Verwendung von GitHub in Kapitel 6 [GitHub](#) im Detail besprechen, da es der größte Git-Host auf dem Markt ist und Sie vermutlich mit Projekten interagieren müssen, die auf GitHub gehostet werden, aber es gibt noch Dutzende weitere, aus denen Sie wählen können, falls Sie nicht Ihren eigenen Git-Server einrichten wollen.

## Zusammenfassung

Sie haben mehrere Möglichkeiten, ein entferntes Git-Repository in Betrieb zu nehmen, damit Sie mit anderen zusammenarbeiten oder Ihre Arbeit teilen können.

Der Betrieb eines eigenen Servers gibt Ihnen viel Kontrolle und ermöglicht es Ihnen, den Server innerhalb Ihrer eigenen Firewall zu betreiben, aber ein solcher Server benötigt in der Regel einen angemessenen Teil Ihrer Zeit für die Einrichtung und Wartung. Wenn Sie Ihre Daten auf einem gehosteten Server ablegen, ist es einfach, sie einzurichten und zu warten; Sie müssen aber die Möglichkeit haben, Ihren Code auf fremden Servern zu speichern, und einige Unternehmen erlauben das nicht.

Es sollte ziemlich einfach sein, festzustellen, welche Lösung oder Kombination von Lösungen für Sie und Ihr Unternehmen am besten geeignet ist.

# Verteiltes Git

Nachdem Sie ein entferntes Git Repository eingerichtet haben, in dem alle Entwickler ihren Code teilen können, und Sie mit den grundlegenden Git-Befehlen in einem lokalen Arbeitsablauf vertraut sind, werden Sie einige der verteilten Arbeitsabläufe verwenden, die Git Ihnen ermöglicht.

In diesem Kapitel erfahren Sie, wie Sie mit Git in einer verteilten Umgebung als Mitwirkender und Integrator arbeiten. Das heißtt, Sie lernen, wie Sie Quelltext erfolgreich zu einem Projekt beisteuern und es Ihnen und dem Projektbetreuer so einfach wie möglich machen. Außerdem lernen Sie, wie Sie ein Projekt erfolgreich verwalten, in dem mehrere Entwicklern Inhalte beisteuern.

## Verteilter Arbeitsablauf

Im Gegensatz zu CVCSSs (Centralized Version Control Systems – Zentrale Versionsverwaltungs Systeme) können Sie 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 Ihr Projekt und/oder Ihrem Team, sodass wir einige gängige Paradigmen behandeln, welche die Vorteile dieser Flexibilität nutzen. Wir werden die Stärken und möglichen Schwächen der einzelnen Entwürfe eingehen. Sie können einen einzelnen davon auswählen, um ihn zu nutzen, oder Sie können 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 Netznoten – Nutzer dieses Hubs – und synchronisieren ihre Arbeit mit diesem einen, zentralen Punkt.

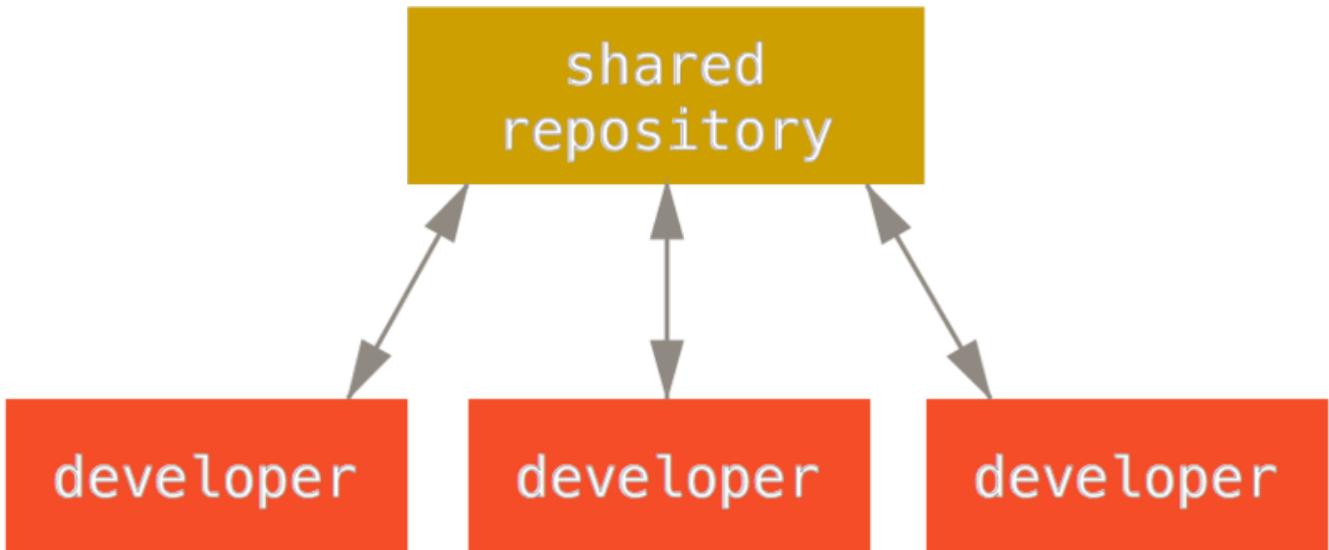


Figure 54. Zentralisierter Arbeitsablauf.

Dies bedeutet, wenn zwei Entwickler ein Repository vom Hub klonen und beide Änderungen vornehmen, der erste Entwickler, der die Änderungen zurückgespielt hat, dies problemlos tun kann. 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 Sie bereits mit einem zentralisierten Arbeitsablauf in Ihrem Unternehmen oder Team vertraut sind, können Sie diesen Ablauf problemlos mit Git weiterverwenden. Richten Sie einfach ein einziges Repository ein und gewähren Sie allen Mitgliedern Ihres Teams Schreib-Zugriff (push). Git lässt nicht zu, dass Benutzer sich 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 bestehende 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 Sie in Git über mehrere Remote-Repositorys verfügen können, 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, erstellen Sie Ihren eigenen öffentlichen Klon des Projekts und laden Ihre Änderungen dort hoch. Anschließend können Sie eine Anforderung an den Betreuer des Hauptprojekts senden, um Ihre Änderungen zu

übernehmen (Pull Request). Der Betreuer kann dann Ihr Repository als Remote hinzufügen, Ihre Änderungen lokal testen, diese in seinem Branch einfließen und in sein öffentliches Repository hochladen. Der Prozess funktioniert wie folgt (siehe [Arbeitsablauf mit Integrationsmanager](#)):

1. Die Projekt Betreuer laden Arbeit 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.

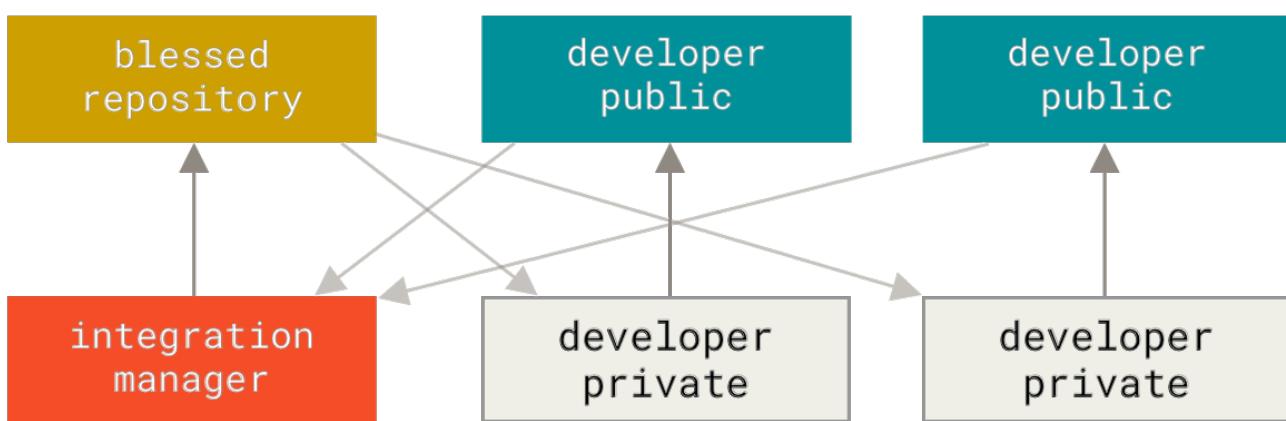


Figure 55. Arbeitsablauf mit Integrationsmanager.

Dies ist ein sehr häufiger Workflow mit Hub-basierten Tools wie GitHub oder GitLab, bei dem es einfach ist, ein Projekt zu „forken“ und Ihre Änderungen in Ihren Fork hochzuladen, damit jeder sie sehen kann. Einer der Hauptvorteile dieses Ansatzes besteht darin, dass Sie weiterarbeiten können und der Verwalter des Haupt-Repositorys Ihre Änderungen jederzeit übernehmen kann. Die Mitwirkenden müssen nicht warten, bis das Projekt ihre 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 Mitarbeitern 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 Themen Branch und reorganisieren (rebasen) ihre Arbeit auf `master`. Der `master`-Branch ist der des Referenz-Repositorys, in das der Diktator pusht.
2. Die Leutnants fassen die Themen Branches der Entwickler mit ihrem `master`-Branch zusammen.

3. Der Diktator führt die `master`-Branches der Leutnants in den `master`-Branch des Diktators zusammen.
4. Schließlich pusht der Diktator diesen `master`-Branch in das Referenz-Repository, damit die anderen Entwickler darauf einen Rebase durchführen können.

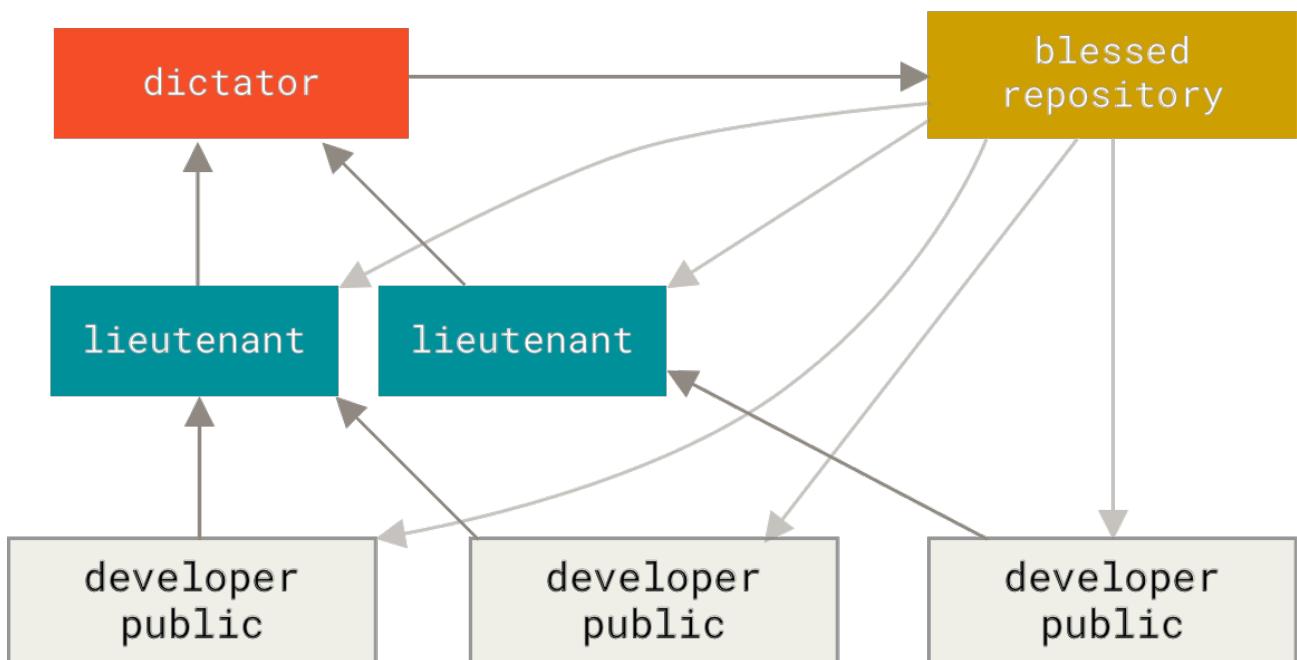


Figure 56. Arbeitsablauf mit wohlwollendem Diktator.

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.

## Zusammenfassung

Dies sind einige häufig verwendete Workflows, die mit einem verteilten System wie Git möglich sind. Allerdings sind auch viele Variationen möglich, um Ihren eigenen Arbeitsabläufen gerecht zu werden. Jetzt, da Sie (hoffentlich) bestimmen können, welche Kombination von Arbeitsabläufen bei Ihnen funktionieren würde, werden wir einige spezifischere Beispiele davon betrachten, wie man die Hauptaufgaben durchführen kann, welche die unterschiedliche Abläufe ausmachen. Im nächsten Abschnitt erfahren Sie 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 geschehen könnte. Da Git sehr flexibel ist, können die Personen auf viele Arten zusammenarbeiten. Es ist nicht einfach zu beschreiben, wie Sie dazu beitragen sollen, da jedes Projekt ein wenig anders ist. Einige der wichtigen Unbekannten sind die Anzahl der aktiven Mitwirkenden, der ausgewählte Arbeitsablauf, Ihre 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 haben Sie zwei oder drei Entwickler mit ein paar Commits pro Tag oder möglicherweise weniger für etwas untätigere 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 Sie mit einer wachsenden Anzahl von Entwicklern auch sicherstellen müssen, dass sich der Code problemlos anwenden lässt und leicht zusammengeführt werden kann. Von Ihnen übermittelte Änderungen können durch Arbeiten, die während Ihrer Arbeit oder während Ihrer Änderungen genehmigt oder eingearbeitet wurden, veraltet sein oder schwer beschädigt werden. Wie können Sie Ihren Quelltext konsistent auf dem neuesten Stand halten und dafür sorgen, dass Ihre 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? Sind Sie selbst in diesen Prozess involviert? Ist ein Leutnant System vorhanden und müssen Sie Ihre Arbeit zuerst bei diesen einreichen?

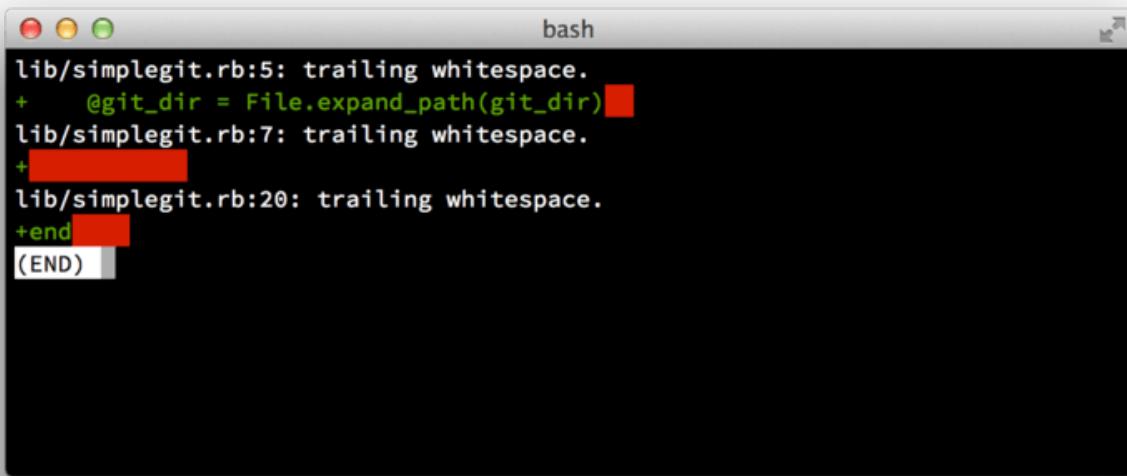
Die nächste Unbekannte ist Ihre Zugriffsberechtigung. Der erforderliche Arbeitsablauf, um zu einem Projekt beizutragen, unterscheidet sich erheblich, wenn Sie Schreibzugriff auf das Projekt haben, als wenn Sie diesen nicht haben. Wenn Sie keinen Schreibzugriff haben, wie bevorzugt das Projekt die Annahme von beigetragener Arbeit? Gibt es überhaupt eine Richtlinie? Wie umfangreich sind die Änderungen, die Sie jeweils beisteuern? Wie oft tragen Sie etwas bei?

All diese Fragen können sich darauf auswirken, wie Sie 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. Sie sollten in der Lage sein, anhand dieser Beispiele die spezifischen Arbeitsabläufe zu erstellen, die Sie in der Praxis benötigen.

## Richtlinien zur Zusammenführung (engl. Commits)

Bevor wir uns mit den spezifischen Anwendungsfällen befassen, finden Sie 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. Sie finden diese Tipps im Git-Quellcode in der Datei [Documentation/SubmittingPatches](#).

Ihre Einsendungen sollten keine Leerzeichenfehler enthalten. Git bietet eine einfache Möglichkeit, dies zu überprüfen. Führen Sie vor dem Commit `git diff --check` aus, um mögliche Leerzeichenfehler zu identifizieren und diese für Sie aufzulisten.



```
lib/simplegit.rb:5: trailing whitespace.  
+ @git_dir = File.expand_path(git_dir)  
lib/simplegit.rb:7: trailing whitespace.  
+  
lib/simplegit.rb:20: trailing whitespace.  
+end  
(END)
```

Figure 57. Ausgabe von `git diff --check`.

Wenn Sie diesen Befehl vor einem Commit ausführen, können Sie feststellen, ob Leerzeichen Probleme auftreten, die andere Entwickler stören könnten.

Als Nächstes, versuchen Sie, aus jedem Commit einen logisch getrennten Satz von Änderungen zu machen. Wenn Sie können, versuchen Sie, Ihre Änderungen leicht verdaulich zu machen – arbeiten Sie nicht ein ganzes Wochenende an fünf verschiedenen Themen und übermitteln Sie dann all diese Änderungen in einem massiven Commit am Montag. Auch wenn Sie am Wochenende keine Commits durchführen, nutzen Sie am Montag die Staging Area, um Ihre Änderungen aufzuteilen in wenigstens einen Commit für jeden Teilapekt mit jeweils einer sinnvollen Nachricht. Wenn einige der Änderungen dieselbe Datei modifizieren, benutzen Sie die Anweisung `git add --patch`, um Dateien partiell zur Staging Area hinzuzufügen (detailliert dargestellt im Abschnitt [Interactive Staging](#)). Der Schnapschuss vom Projekt an der Spitze des Branches ist der Selbe, ob Sie einen oder fünf Commits durchgeführt haben, solange nur all die Änderungen irgendwann hinzugefügt werden. Versuchen Sie also, die Dinge zu vereinfachen für Ihre Entwicklerkollegen, die Ihre Änderungen begutachten müssen.

Dieser Ansatz macht es außerdem einfacher, einen Satz von Änderungen zu entfernen oder rückgängig zu machen, falls das später nötig wäre. [Rewriting History](#) beschreibt eine Reihe nützlicher Git-Tricks zum Umschreiben des Verlaufs oder um interaktiv Dateien zur Staging Area hinzuzufügen. Verwenden Sie diese Werkzeuge, um einen sauberen und leicht verständlichen Verlauf aufzubauen, bevor Sie Ihre Arbeit jemand anderem schicken.

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 sollten Ihre Nachrichten mit einer einzelnen Zeile beginnen, die nicht länger als 50 Zeichen ist. Diese sollte ihre 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 Ihre Motivation für die Änderung enthält. Außerdem sollte das Ergebnis ihre 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, verwenden Sie Anweisungen. Anstatt „Ich habe Test hinzugefügt für“ oder „Tests hinzufügend für“ benutzen Sie „Füge Tests hinzu für“. Hier ist eine [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, Sie lassen den Textkörper ganz weg). Werkzeuge wie rebase können durcheinander kommen, wenn Sie diese Trennung nicht einhalten.

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.

Verwenden Sie einen hängenden Einzug

Wenn alle ihre Commit-Nachrichten diesem Modell folgen, wird es für Sie und die Entwickler, mit denen Sie zusammenarbeiten, viel einfacher sein. Das Git-Projekt selber enthält gut formatierte Commit-Nachrichten. Versuchen Sie, `git log --no-merges` auszuführen, um zu sehen, wie ein gut formatierter Commit-Verlauf des Projekts aussieht.

*Tun sie 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, tun Sie es wie wir es sagen und nicht wie wir es tun.

## Kleines, privates Team

Das einfachste Setup, auf das Sie wahrscheinlich stoßen werden, 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. Sie und die anderen Entwickler haben alle Schreibzugriff (Push-Zugriff) auf das Repository.

In dieser Umgebung können Sie einem Arbeitsablauf folgen, der dem ähnelt, den Sie mit Subversion oder einem anderen zentralisierten System ausführen würden. Sie haben 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 finden Sie weiter unten in den Diskussionen. Wenn Sie also ein grundlegendes Verständnis der Bedeutung dieser Angaben haben, dann können Sie die verschiedenen Zustände der Repositorys besser verstehen. Weitere Informationen dazu finden Sie 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 Sie an Subversion gewöhnt sind. Wie Sie sicherlich bemerkt haben, haben die beiden Entwickler nicht dieselbe Datei bearbeitet. Obwohl Subversion eine solche Zusammenführung automatisch auf dem Server durchführt, wenn verschiedene Dateien bearbeitet werden, müssen Sie 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:

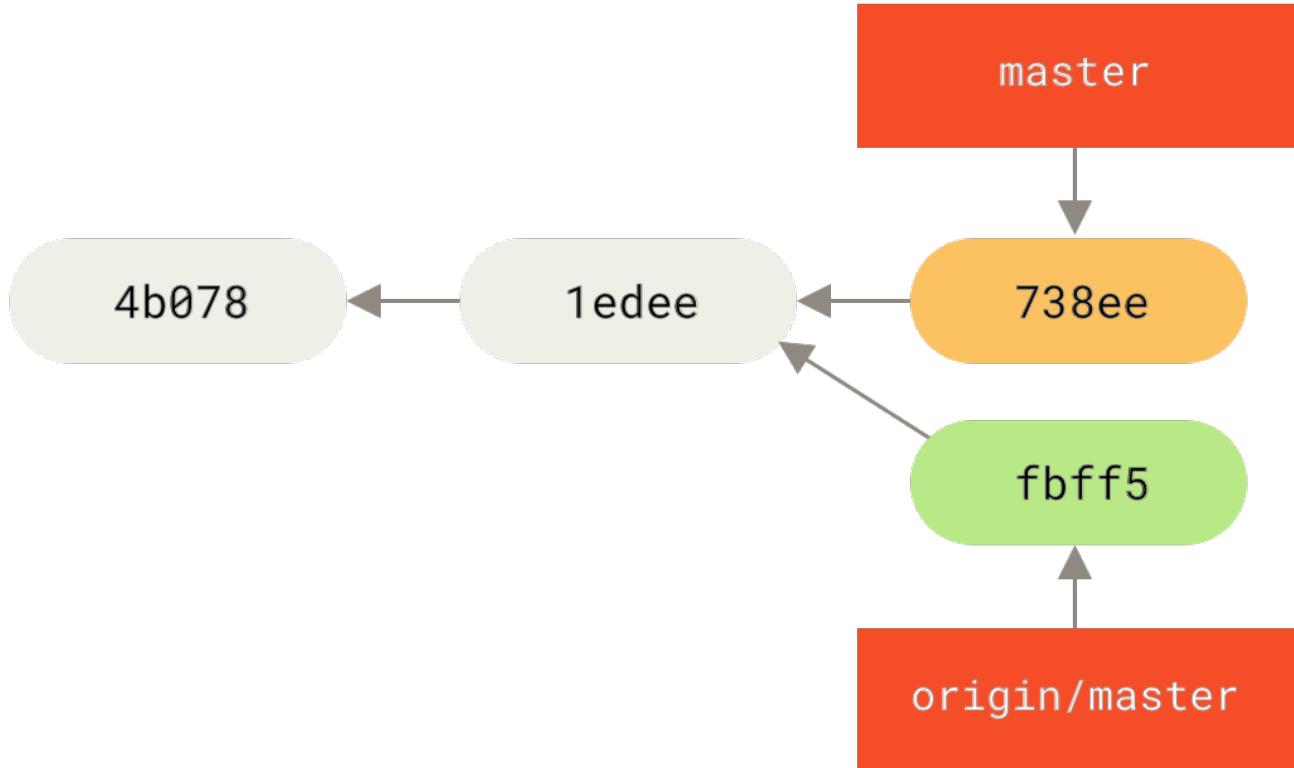


Figure 58. Johns abzweigender Verlauf.

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:

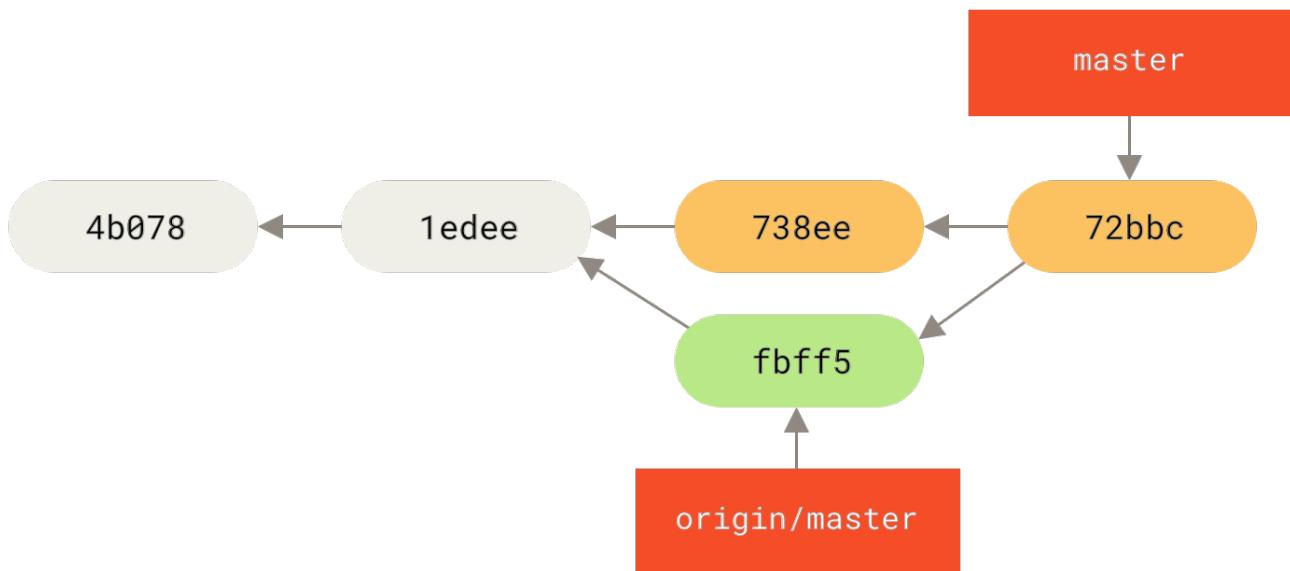


Figure 59. Johns Repository nach der Zusammenführung `origin/master`.

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
fbfff5bc..72bbc59  master -> master
```

Am Ende sieht Johns Commit-Verlauf so aus:

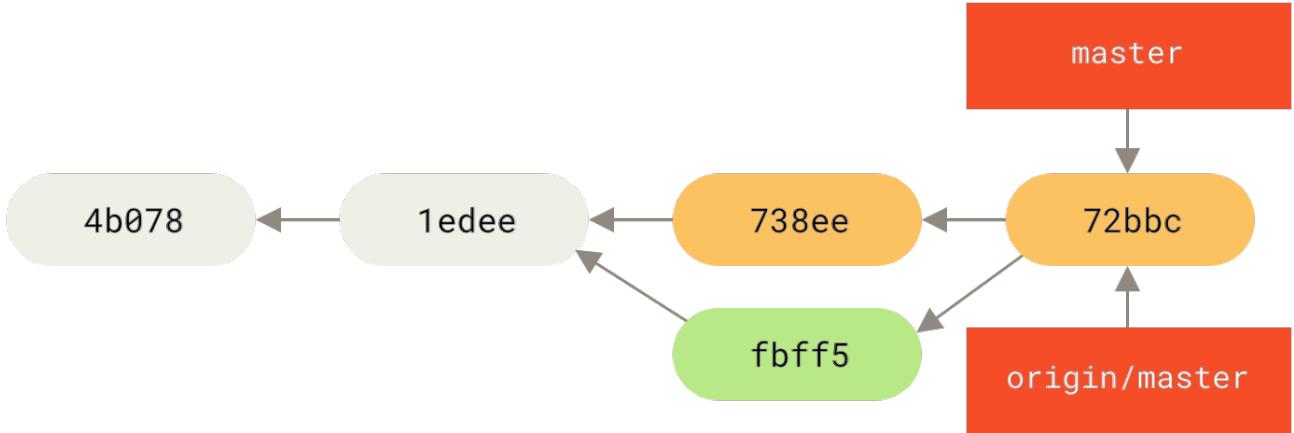


Figure 60. Johns Verlauf nach Hochladen auf den origin-Server.

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:

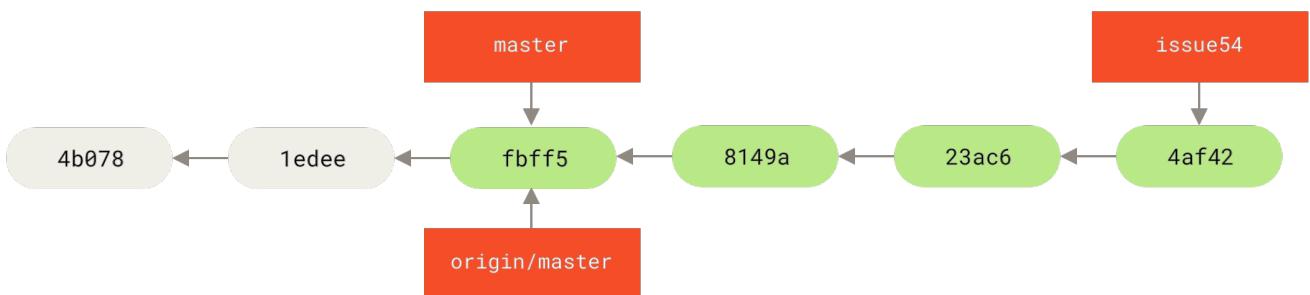


Figure 61. Jessicas Themen Branch.

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
  fffff5bc..72bbc59  master      -> origin/master
```

Dies zieht die Arbeit herunter (Pull), die John in der Zwischenzeit hochgeladen hat. Jessicas Verlauf sieht jetzt so aus:

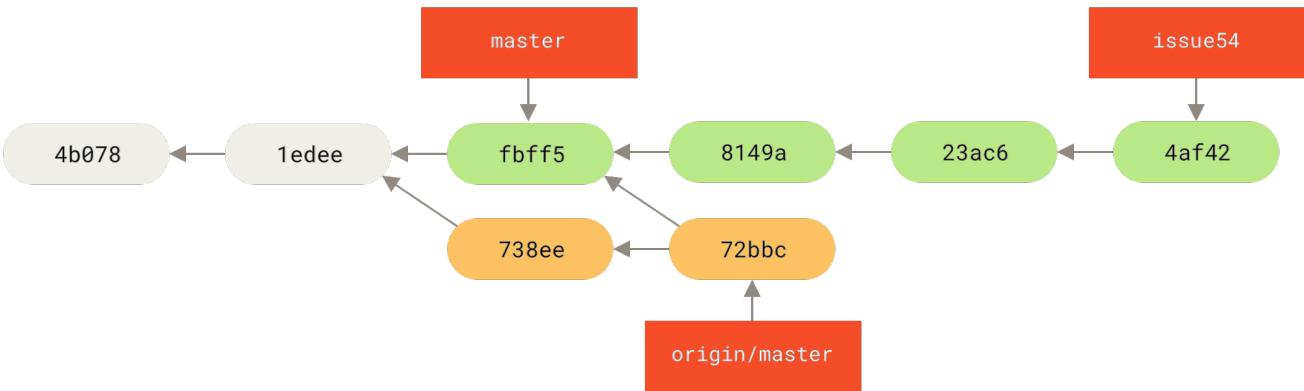


Figure 62. Jessicas Verlauf nach dem Abholen von Johns Änderungen.

Jessica denkt, dass ihr Themen 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 Themen Branch `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 fbf5bc..4af4298
Fast forward
 README          |    1 +
 lib/simplegit.rb |   6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Es treten keine Probleme auf. Wie Sie sehen, handelte es sich um einen einfachen 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:

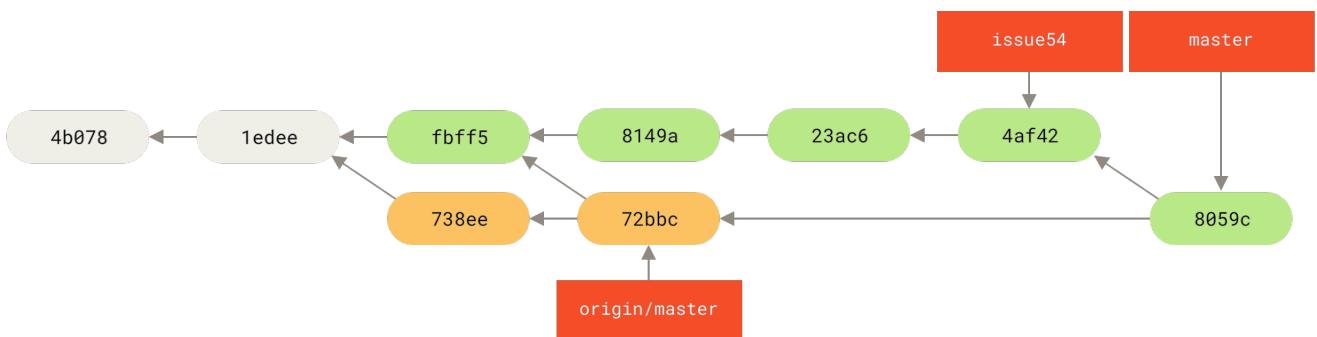


Figure 63. Jessicas Verlauf nach Zusammenführung mit Johns Änderungen.

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.

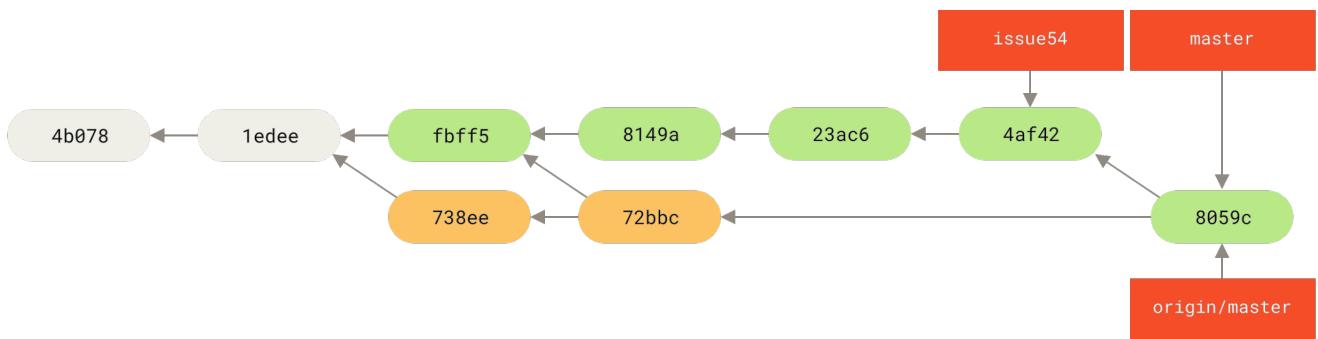


Figure 64. Jessicas Verlauf nach hochladen aller Änderungen auf den Server.

Das ist einer der einfachsten Arbeitsabläufe. Sie arbeiten eine Weile (in der Regel in einem Themen Branch) und führen diese Arbeiten in Ihrem Branch `master` zusammen, sobald sie für die Integration bereit sind. Wenn Sie diese Arbeit teilen möchten, rufen Sie Ihren `master` von `origin/master` ab und führen ihn zusammen, falls er sich geändert hat. Anschließend pushen Sie ihn in den `master` Branch auf dem Server. Die allgemeine Reihenfolge sieht in etwa so aus:

1. Allgemeine Abfolge von Ereignissen für einen einfachen Arbeitsablauf mit mehreren Entwicklern. image::images/small-team-flow.png[Allgemeine Abfolge von Ereignissen für einen einfachen Arbeitsablauf mit mehreren Entwicklern.]

## Geführtes, privates Team

In diesem Szenario sehen Sie sich die Rollen der Mitwirkenden in einem größeren, geschlossenen Team an. Sie lernen, wie Sie in einer Umgebung arbeiten, 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 'made the ls-tree function recursive'  
[featureB e5b0fdc] made the ls-tree function recursive  
 1 files changed, 1 insertions(+), 1 deletions(-)  
$ vim lib/simplegit.rb  
$ git commit -am 'add ls-files'  
[featureB 8512791] add ls-files  
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Jessicas Repository sieht nun folgendermaßen aus:

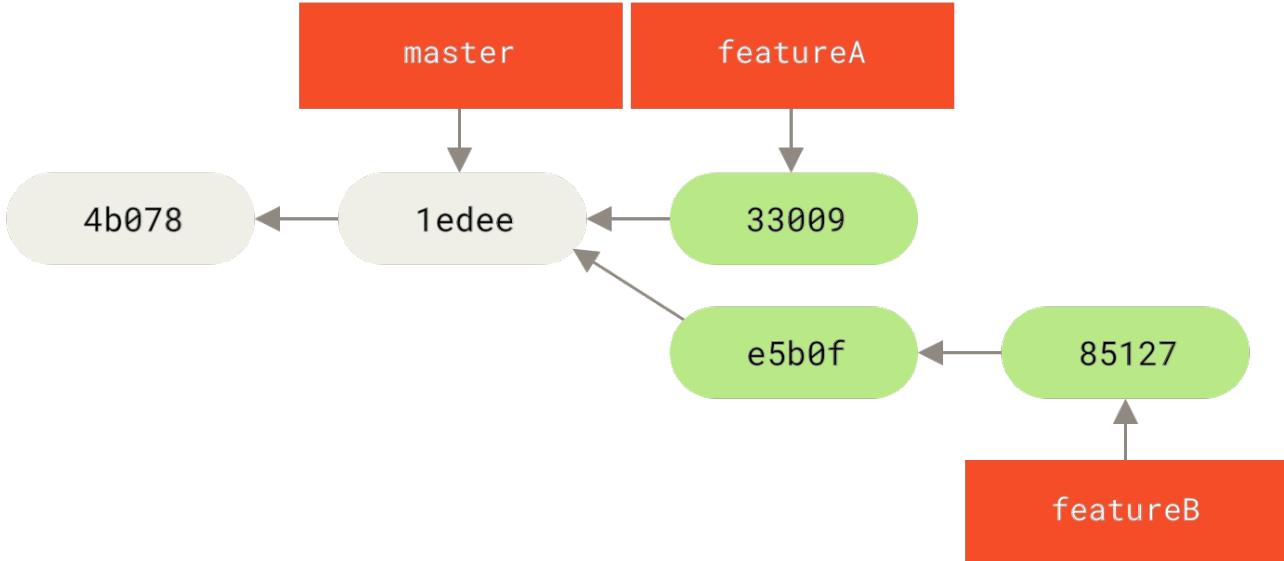


Figure 65. Jessicas initialer Commit Verlauf.

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 Zweig 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 \(engl. Refspec\)](#) finden Sie eine

detailliertere Beschreibung der Git-Refspecs und der verschiedenen Möglichkeiten, die Sie damit haben. Beachten Sie auch die `-u` Option. Dies ist die Abkürzung für `--set-upstream`, mit der die Branches so konfiguriert werden, dass sie später leichter gepusht und gepullt 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

        changed 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, sie in ihren lokalen Branch `featureA` comitten und das Endergebnis zurück auf den Server übertragen.

```

$ git commit -am 'small tweak'
[featureA 774b3ed] small tweak
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
...
To jessica@githost:simplegit.git
 3300904..774b3ed  featureA -> featureA

```

Jessicas commit Verlauf sieht nun in etwa so aus:

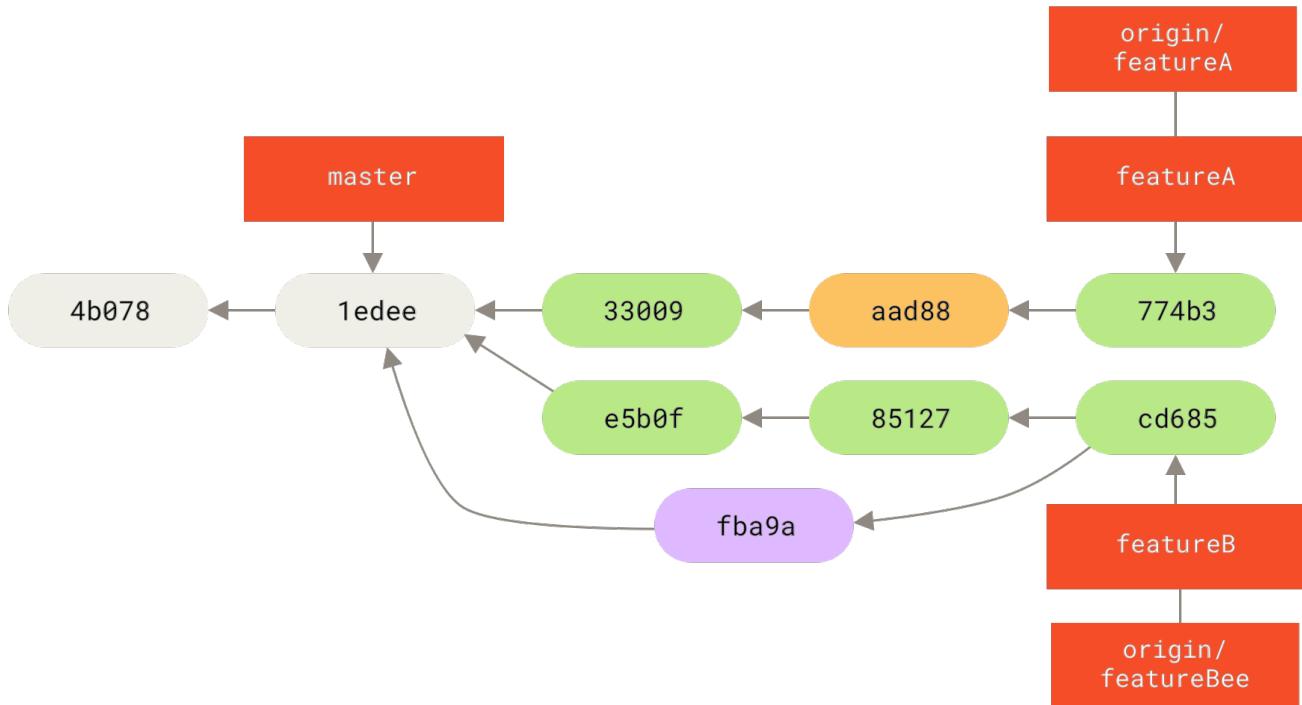


Figure 66. Jessicas Verlauf nach committen auf einem Feature Branch.

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:

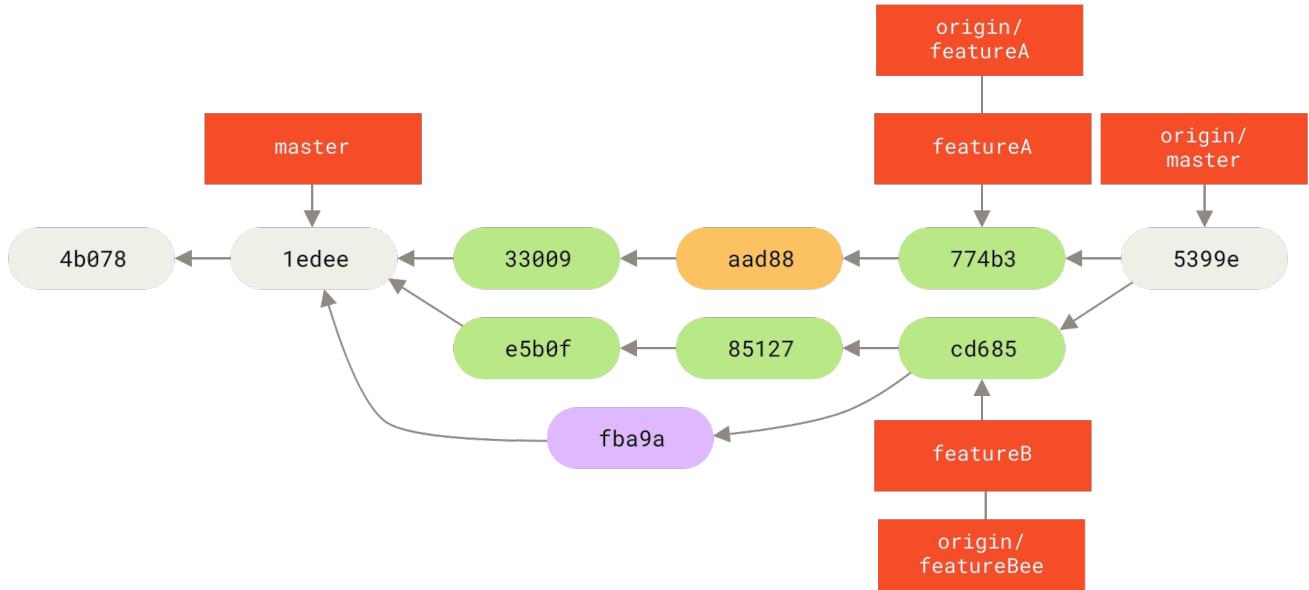


Figure 67. Jessicas Verlauf nach Zusammenführung ihrer beiden Themen Branches.

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:

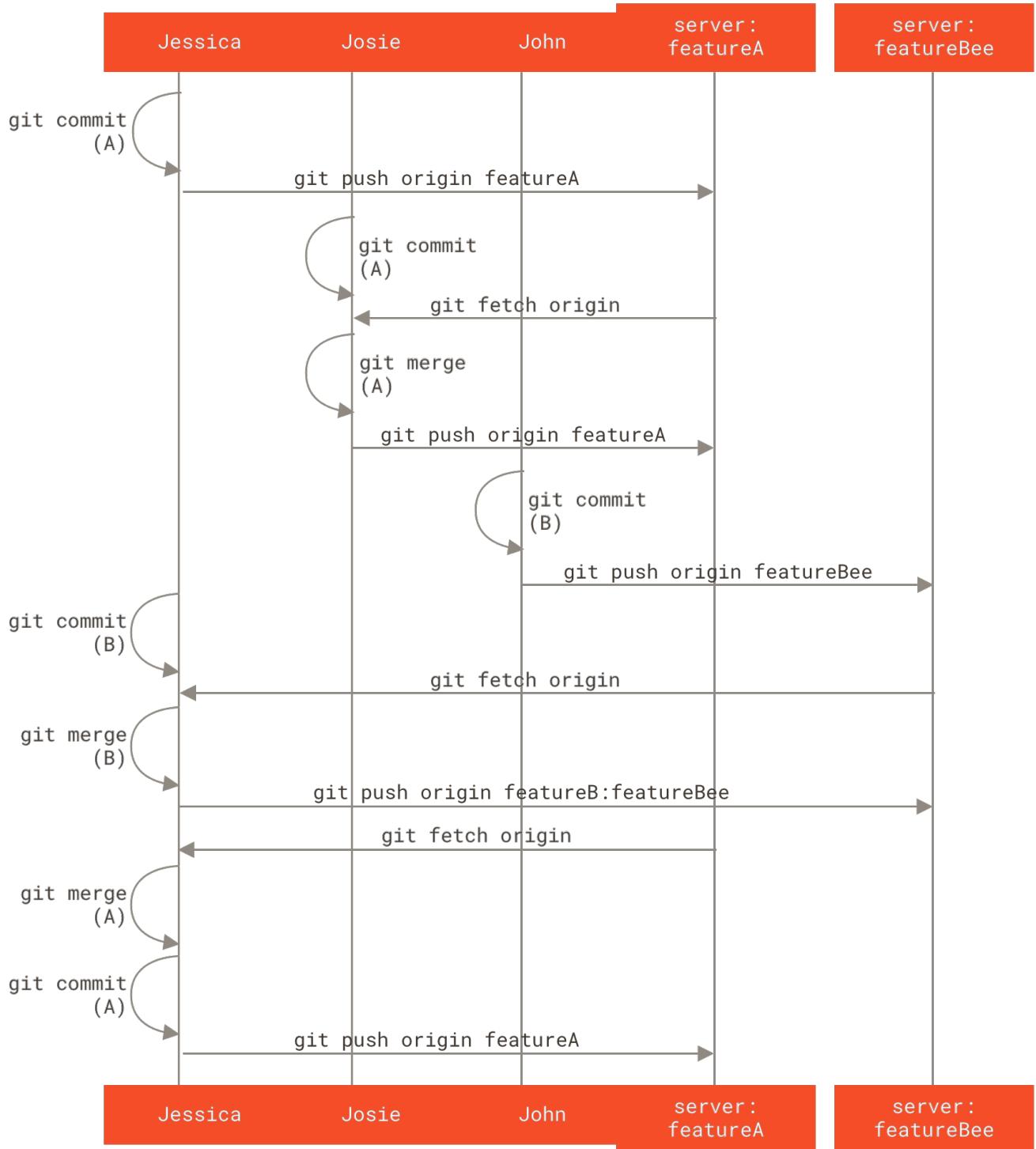


Figure 68. Grundlegende Vorgehensweise bei einem geführten Team.

## Verteiltes, öffentliches Projekt

An öffentlichen Projekten mitzuwirken ist ein wenig anders. Da Sie nicht die Berechtigung haben, Branches im Projekt direkt zu aktualisieren, müssen Sie Ihre Arbeit auf andere Weise an die Projektbetreuer weiterleiten. In diesem ersten Beispiel wird beschrieben, wie Sie auf Git Hosts, die einfaches „Forking“ unterstützen, via „Forking“ mitwirken können. 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 möchten Sie wahrscheinlich das Hauptrepository klonen, einen Branch für den Patch oder die Patch Serien erstellen, die Sie beisteuern möchten, und dort Ihre Arbeit erledigen. Der Prozess sieht im Grunde so aus:

```
$ git clone <url>
$ cd project
$ git checkout -b featureA
... work ...
$ git commit
... work ...
$ git commit
```



Sie können `rebase -i` verwenden, um Ihre Arbeit auf ein einzelnes Commit zu reduzieren. Sie können auch die Änderungen in den Commits neu anordnen, damit der Betreuer den Patch einfacher überprüfen kann – siehe [Rewriting History](#) für weitere Informationen zum interaktiven Rebasing.

Wenn Ihre Arbeit am Branch abgeschlossen ist und Sie bereit sind, sie an die Betreuer weiterzuleiten, wechseln Sie zur ursprünglichen Projektseite. Dort klicken Sie auf die Schaltfläche **Fork**, um Ihren eigenen schreibbaren Fork des Projekts zu erstellen. Anschließend müssen Sie diese Repository-URL als neue Remote-Adresse Ihres lokalen Repositorys hinzufügen. Nennen wir es in diesem Beispiel `myfork`:

```
$ git remote add myfork <url>
```

Anschließend müssen Sie Ihre neue Arbeit in dieses Repository hochladen. Es ist am einfachsten, den Branch, an dem Sie arbeiten, in Ihr geforktes Repository hochzuladen, anstatt diese Arbeit in Ihrem Master-Branch zusammenzuführen und diesen hochzuladen. Der Grund dafür ist, dass Sie Ihren `master`-Branch nicht zurücksetzen müssen, wenn Ihre Arbeit nicht akzeptiert bzw. nur teilweise ausgewählt (cherry-pick) wurde (die Git-Operation zum `cherry-pick` wird ausführlicher in [Rebasing und Cherry-Picking Workflows](#) behandelt). Wenn die Betreuer Ihre Arbeit per `mergen`, `rebase` oder `cherry-pick` übernehmen, erhalten Sie ihre Arbeit sowieso zurück, wenn Sie aus dem Repository der Betreuer pullen.

Auf jedem Fall können Sie Ihre Arbeit hochladen mit:

```
$ git push -u myfork featureA
```

Sobald Ihre Arbeit an Ihren Fork des Repositorys hochgeladen wurde, müssen Sie den Betreuern des ursprünglichen Projekts mitteilen, dass Sie Änderungen haben, die sie zusammenführen möchten. Dies wird oft als *Pull Request* bezeichnet. Sie generieren eine solche Anfrage entweder über die Website – GitHub hat einen eigenen „Pull-Request-Mechanismus“, den wir in [GitHub](#) behandeln werden, oder Sie können 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 Ihr Themen 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 Sie bitten. Wenn bspw. Jessica an John eine Pull Request senden möchte und sie zwei Commits für den gerade hochgeladenen Themen Branch ausgeführt hat, kann sie folgendes ausführen:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
Jessica Smith (1):
    added a new function

are available in the git repository at:

git://githost/simplegit.git featureA

Jessica Smith (2):
    add limit to log function
    change log output to 30 from 25

lib/simplegit.rb | 10 ++++++--
1 files changed, 9 insertions(+), 1 deletions(-)
```

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 Sie nicht der Betreuer sind, ist es im Allgemeinen einfacher, einen Branch wie `master` zu haben, der immer `origin/master` verfolgt. Ihre Arbeit können Sie dann in Themen Branches erledigen, die Sie einfach verworfen können, wenn ihre Änderungen abgelehnt werden. Durch das Isolieren von Änderungen in Themen Branches wird es für Sie auch einfacher, Ihre Arbeit neu zu strukturieren. Falls sich das Haupt-Repositorys in der Zwischenzeit weiter entwickelt hat und Ihre Commits nicht mehr sauber angewendet werden können. Wenn Sie beispielsweise ein zweites Thema an das Projekt senden möchten, arbeiten Sie nicht weiter an dem Branch, den Sie gerade hochgeladen haben. Beginnen Sie 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 Ihrer Themen in einer Art Silo enthalten, ähnlich wie bei einer Patch-Warteschlange. Dieses können Sie umarbeiten, zurücksetzen oder ändern, ohne dass die Themen sich gegenseitig stören oder voneinander abhängig sind.

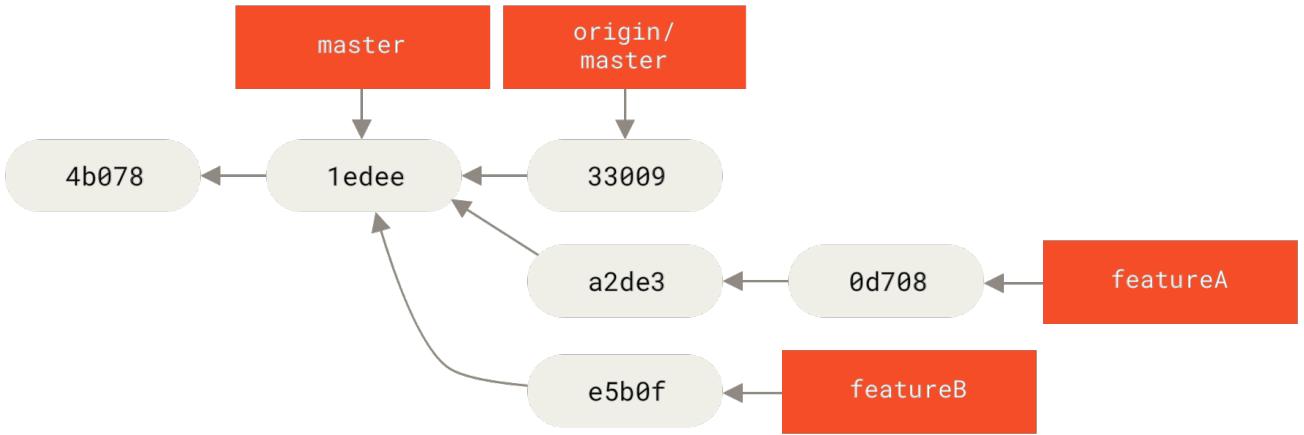


Figure 69. Initialer Commit Verlauf mit **featureB** Änderungen.

Nehmen wir an, der Projektbetreuer hat eine Reihe weiterer Patches übernommen und Ihren ersten Branch einfließen lassen, der jedoch nicht mehr ordnungsgemäß zusammengeführt werden kann. In diesem Fall können Sie versuchen, diesen Branch auf 'origin/master' zu reorganisieren, die Konflikte für den Betreuer zu lösen und Ihre Änderungen erneut zu übermitteln:

```
$ git checkout featureA
$ git rebase origin/master
$ git push -f myfork featureA
```

Dadurch wird Ihr Verlauf so umgeschrieben, sodass er jetzt folgendermaßen aussieht [Commit Verlauf nach featureA Änderungen..](#)

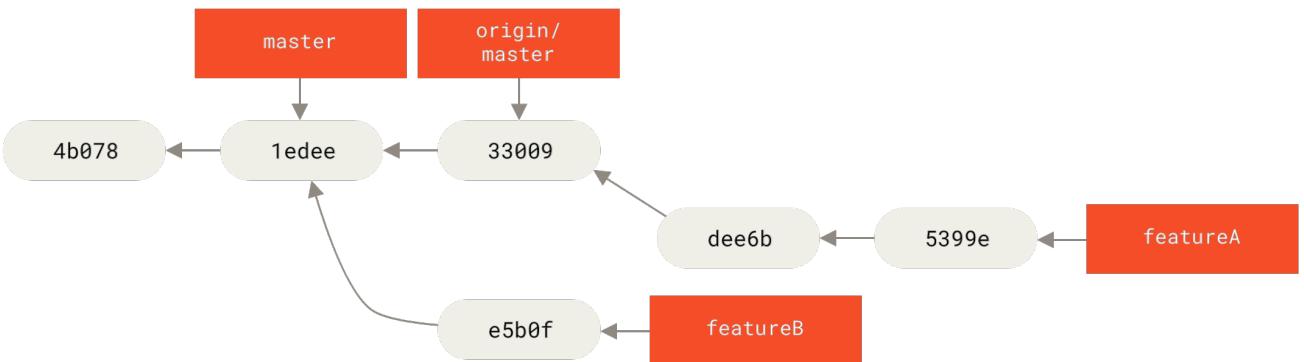


Figure 70. Commit Verlauf nach **featureA** Änderungen.

Da Sie den Branch reorganisiert haben, müssen Sie das **-f** für Ihren 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 Ihrem zweiten Branch angesehen und mag ihr Konzept, möchte aber, dass Sie ein Implementierungsdetail ändern. Sie nutzen diese Gelegenheit, um Ihre Änderungen zu verschieben, damit diese auf dem aktuellen **master** Branch des Projektes basieren. Sie starten einen neuen Branch, der auf den aktuellen Branch **origin/master** basiert und fassen die Änderungen an **featureB** dort zusammen.

Dabei lösen sie etwaige Konflikte, machen die Implementierungsänderungen und laden 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 Ihr zukünftiger Commit nur einen übergeordneten Vorgänger hat. Das erlaubt Ihnen alle Änderungen aus einem anderen Branch einzuführen und weitere Änderungen vorzunehmen, bevor Sie den neuen Commit aufnehmen. Auch die Option `--no-commit` kann nützlich sein, um den Merge-Commit im Falle des Standard-Merge-Prozesses zu verzögern.

Nun können Sie den Betreuer darüber informieren, dass Sie die angeforderten Änderungen vorgenommen haben und dass er diese Änderungen in Ihrem Branch `featureBv2` findet.

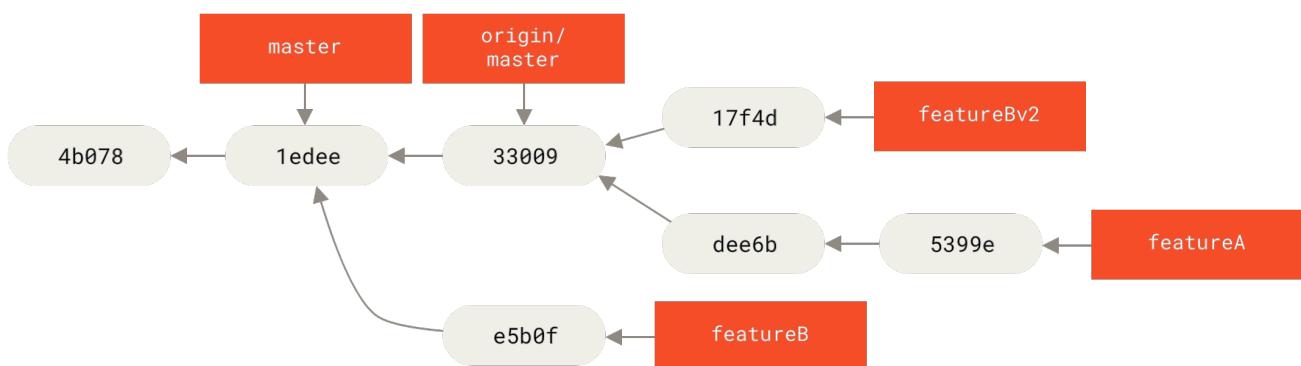


Figure 71. Commit Verlauf nach getaner `featureBv2` Arbeit.

## Öffentliche Projekte via Email

Viele Projekte haben fest definierte Prozesse, um Änderungen entgegenzunehmen. Sie müssen 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: Sie erstellen Themen Branches für jede Patch Serie, an der Sie arbeiten. Der Unterschied besteht darin, wie Sie diese Änderungen an das Projekt senden. Anstatt das Projekt zu forken und auf Ihr eigenes geforktes Repository hochzuladen, generieren Sie E-Mail-Versionen jeder Commit-Serie und senden diese per E-Mail an die Entwickler-Mailingliste:

```
$ git checkout -b topicA
... work ...
$ git commit
... work ...
$ git commit
```

Jetzt haben Sie zwei Commits, die Sie an die Mailingliste senden können. Sie verwenden `git format-patch`, um die mbox-formatierten Dateien zu generieren, die Sie anschließend per E-Mail an die Mailingliste senden. Dabei wird jedes 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-changed-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

```

Sie können diese Patch-Dateien auch bearbeiten, um weitere Informationen für die E-Mail-Liste hinzuzufügen, die nicht in der Commit-Nachricht angezeigt werden sollen. Wenn Sie Text zwischen der Zeile `---` und dem Beginn des Patches (der Zeile `diff --git`) einfügen, 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, können Sie 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 Sie ordnungsgemäß formatierte Patches über IMAP senden können, was einfacher für Sie sein könnte. Wir zeigen Ihnen, wie Sie einen Patch über Google Mail senden. Dies ist der E-Mail-Agent, mit dem wir uns am besten auskennen. Detaillierte Anweisungen für eine Reihe von anderen Mail-Programmen finden Sie am Ende der oben genannten Datei [Documentation/SubmittingPatches](#) im Git-Quellcode.

Zuerst müssen Sie den Abschnitt `imap` in Ihrer `~/.gitconfig` Datei einrichten. Sie können jeden Wert separat mit einer Reihe von `git config` Befehlen festlegen oder manuell hinzufügen. Am Ende sollte Ihre 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 Ihr 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, können Sie `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 sollten Sie in der Lage sein, in Ihren Entwurfsordner zu wechseln und dort das Feld An der generierten Email in die Mailinglist-Adresse zu ändern, an die Sie den Patch senden wollen. Möglicherweise wollen sie auch den Betreuer oder die Person in Kopie nehmen, die für diesen Abschnitt verantwortlich ist. Anschließend können sie die Mail versenden.

Sie können die Patches auch über einen SMTP-Server senden. Wie zuvor können Sie jeden Wert separat mit einer Reihe von `git config` Befehlen festlegen oder manuell im Abschnitt `sendemail` in Ihrer `~/.gitconfig` Datei hinzufügen:

```
[sendemail]
  smtpencryption = tls
  smtpserver = smtp.gmail.com
  smtpuser = user@gmail.com
  smtpserverport = 587
```

Danach können Sie Ihre Patches mit `git send-email` versenden:

```
$ git send-email *.patch
0001-added-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
Who should the emails appear to be from? [Jessica Smith <jessica@example.com>]
Emails will be sent from: Jessica Smith <jessica@example.com>
Who should the emails be sent to? jessica@example.com
Message-ID to be used as In-Reply-To for the first email? y
```

Git gibt anschließend für jeden Patch, den Sie versenden, 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] added limit to log function
Date: Sat, 30 May 2009 13:29:15 -0700
Message-Id: <1243715356-61726-1-git-send-email-jessica@example.com>
X-Mailer: git-send-email 1.6.2.rc1.20.g8c5b.dirty
In-Reply-To: <y>
References: <y>

Result: OK
```

## Zusammenfassung

In diesem Abschnitt wurden eine Reihe gängiger Arbeitsabläufe für den Umgang mit verschiedenen Arten von Git-Projekten behandelt. Außerdem wurden einige neue Tools vorgestellt, die Sie bei der Verwaltung dieser Prozesse unterstützen. Als Nächstes erfahren Sie, wie Sie auf der anderen Seite arbeiten: als Verwalter (Maintainer) eines Git-Projektes. Sie lernen, wie man sich als wohlwollender Diktator oder Integrationsmanager korrekt arbeitet.

# Ein Projekt verwalten

Sie müssen nicht nur wissen, wie Sie effektiv zu einem Projekt etwas beitragen. Sie sollten auch wissen, wie Sie ein Projekt verwalten. Sie müssen bspw. wissen wie sie Patches akzeptieren und anwenden, die über `format-patch` generiert und per E-Mail an Sie gesendet wurden. Weiterhin sollten Sie wissen wie sie Änderungen in Remote-Banches für Repositorys integrieren, die Sie als Remotes zu Ihrem Projekt hinzugefügt haben. Unabhängig davon, ob Sie ein zentrales Repository verwalten oder durch Überprüfen oder Genehmigen von Patches helfen möchten, müssen Sie wissen, wie Sie die Arbeit anderer so akzeptieren, dass es für andere Mitwirkende transparent und auf lange Sicht auch nachhaltig ist.

## Arbeiten in Themen Branches

Wenn man vorhat, neuen Quelltext zu integrieren, ist es im Allgemeinen eine gute Idee, sie in einem *Topic 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 Sie Zeit haben, sich wieder damit zu befassen. Sie sollten einen einfachen Branchnamen erstellen, der auf dem Thema der Arbeit basiert, die Sie durchführen, wie z.B. `ruby_client` oder etwas ähnlich Sprechendes. Dann können Sie sich später leichter daran erinnern, falls Sie den Branch für eine Weile haben ruhen lassen und später daran weiter arbeiten. 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 Sie sich erinnern werden, können Sie den Branch basierend auf Ihrem **master**-Branch wie folgt erstellen:

```
$ git branch sc/ruby_client master
```

Wenn sie anschließend sofort zum neuen Branch wechseln möchten, können Sie auch die Option **checkout -b** verwenden:

```
$ git checkout -b sc/ruby_client master
```

Jetzt können Sie die getätigte Arbeit zu diesem Branch hinzufügen und festlegen, ob Sie ihn mit Ihren bestehenden Branches zusammenführen möchten.

## Integrieren von Änderungen aus E-Mails

Wenn Sie einen Patch per E-Mail erhalten, den Sie in Ihr Projekt integrieren müssen, müssen Sie den Patch in Ihrer Themen Branch einfließen lassen, damit sie ihn prüfen können. Es gibt zwei Möglichkeiten, einen per E-Mail versendeten Patch anzuwenden: mit **git apply** oder mit **git am**.

### Änderungen mit **apply** integrieren

Wenn Sie den Patch von jemandem erhalten haben, der ihn mit **git diff** oder mit einer Variante des Unix-Befehls `diff` erzeugt hat (was nicht empfohlen wird; siehe nächster Abschnitt), können Sie ihn mit dem Befehl **git apply** integrieren. Angenommen Sie haben den Patch unter **/tmp/patch-ruby-client.patch** gespeichert. Dann können Sie den Patch folgendermaßen integrieren:

```
$ git apply /tmp/patch-ruby-client.patch
```

Hierdurch werden die Dateien in Ihrem 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 angewendet wird. **patch** hingegen integriert Patchdateien eventuell nur teilweise und kann Ihr Arbeitsverzeichnis in einem undefinierten Zustand versetzen. **git apply** ist insgesamt sehr viel konservativer als **patch**. Es wird kein Commit erstellen. Nach dem Ausführen müssen Sie die eingeführten Änderungen manuell bereitstellen und comitten.

Sie können **git apply** verwenden, um zu prüfen, ob ein Patch ordnungsgemäß integriert werden kann, bevor Sie versuchen, ihn tatsächlich anzuwenden. Sie können **git apply --check** auf den Patch ausführen:

```
$ git apply --check 0001-seeing-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 können sie 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 Ihre Arbeit einfacher. Der Patch enthält bereits Informationen über den Autor und eine entsprechende Commitnachricht. Wenn möglich, ermutigen Sie die Beitragenden `format-patch` anstelle von `diff` zum Erstellen von Patches zu verwenden. Sie sollten `git apply` nur für ältere Patches und ähnliche Dinge verwenden.

Um einen von `format-patch` erzeugten Patch zu integrieren, verwenden Sie `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 Sie im vorherigen Abschnitt gesehen haben. Es zeigt ein gültiges mbox Email Format. Wenn Ihnen jemand den Patch ordnungsgemäß mit `git send-email` per E-Mail zugesendet hat und Sie ihn in ein mbox-Format herunterladen, können Sie `git am` auf diese mbox-Datei ausführen. Damit werden alle angezeigten Patches entsprechend angewendet. Wenn Sie einen Mail-Client ausführen, der mehrere E-Mails im Mbox-Format speichern kann, können Sie 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 Ticketing-System oder ähnliches hochgeladen hat, können Sie die Datei lokal speichern. Die Datei können Sie dann an `git am` übergeben, um sie integrieren:

```
$ git am 0001-limit-log-function.patch
Applying: add limit to log function
```

Wie Sie können sehen, wurde der Patch korrekt integriert und es wurde automatisch ein neuer Commit für Sie 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 Ihr 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 Sie noch nicht angewendet haben. In diesem Fall schlägt der Prozess **git am** fehl und Sie werden gefragt, was Sie tun möchten:

```
$ git am 0001-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Patch failed at 0001.
When you have resolved this problem run "git am --resolved".
If you would prefer to skip this patch, instead run "git am --skip".
To restore the original branch and stop patching run "git am --abort".
```

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). Sie lösen dieses Problem auf die gleiche Art: Bearbeiten Sie die Datei, um den Konflikt zu lösen. Anschließend fügen sie die neue Datei der Staging Area hinzu und führen dann **git am --resolved** aus, um mit dem nächsten Patch fortzufahren:

```
$ (fix the file)
$ git add ticgit.gemspec
$ git am --resolved
Applying: seeing if this helps the gem
```

Wenn Sie möchten, dass Git den Konflikt etwas intelligenter löst, können Sie ihm die Option „-3“ übergeben, wodurch Git versucht, eine Dreifachzusammenführung durchzuführen. Diese Option ist standardmäßig nicht aktiviert, da sie nicht funktioniert, wenn sich der Commit, auf dem der Patch basiert, nicht in Ihrem Repository befindet. Wenn Sie diesen Commit haben – 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-seeing-if-this-helps-the-gem.patch
Applying: seeing if this helps the gem
error: patch failed: ticgit.gemspec:1
error: ticgit.gemspec: patch does not apply
Using index info to reconstruct a base tree...
Falling back to patching base and 3-way merge...
No changes -- Patch already applied.
```

In 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 Sie mehrere Patches aus mbox anwenden, können Sie auch den Befehl **am** im interaktiven Modus ausführen. Bei jedem gefundenen Patch wird angehalten und Sie werden gefragt, ob Sie ihn anwenden möchten:

```
$ git am -3 -i mbox
Commit Body is:
-----
seeing if this helps the gem
-----
Apply? [y]es/[n]o/[e]dit/[v]iew patch/[a]ccept all
```

Dies ist hilfreich, wenn Sie eine Reihe von Patches gespeichert haben. Sie können sich den Patch zuerst anzeigen lassen, wenn Sie sich nicht daran erinnern, worum es genau geht. Oder sie wenden den Patch nicht an, weil Sie es bereits getan haben.

Wenn alle Patches für Ihr Thema angewendet und in Ihrem Branch committet wurden, können Sie auswählen, ob und wie Sie sie in einen Hauptbranch integrieren möchten.

## Remote Branches auschecken

Wenn sie einen Beitrag von einem Git-Nutzer erhalten, der sein eigenes Repository eingerichtet, eine Reihe von Änderungen vorgenommen und Ihnen dann die URL zum Repository und den Namen des Remote-Zweigs gesendet hat, in dem sich die Änderungen befinden, dann können Sie diesen als remote hinzufügen und die Änderungen lokal zusammenführen.

Wenn Jessica Ihnen bspw. eine E-Mail sendet, die besagt, dass sie eine großartige neue Funktion im **ruby-client** Branch ihres Repositorys hat, können Sie diese testen, indem Sie den Branch als remote hinzufügen und ihn lokal auschecken:

```
$ git remote add jessica git://github.com/jessica/myproject.git
$ git fetch jessica
$ git checkout -b rubyclient jessica/ruby-client
```

Wenn Jessica Ihnen später erneut eine E-Mail mit einem anderen Branch sendet, der eine weitere

großartige Funktion enthält, können Sie diese direkt abrufen und auschecken, da Sie bereits über das Remote Repository verfügen.

Dies ist am nützlichsten, wenn Sie durchgängig mit einer Person arbeiten. Wenn jemand nur selten einen Patch zur Verfügung steht, 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 Sie Hunderte von Remotes einbinden möchten für Personen, die nur ein oder zwei Patches beisteuern. Skripte und gehostete Dienste können dies jedoch vereinfachen – dies hängt weitgehend davon ab, wie Sie und die Mitwirkenden entwickeln.

Der andere Vorteil dieses Ansatzes ist, dass Sie auch die Historie der Commits erhalten. Obwohl Sie möglicherweise berechtigte Probleme bei der Zusammenführungen haben, wissen Sie, wo in Ihrer 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 Sie Zugriff haben.

Wenn Sie nicht durchgängig mit einer Person arbeiten, aber dennoch auf diese Weise von dieser Person abrufen möchten, können Sie 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 eingeführt wird

Sie haben nun einen Themen Branch mit neuen Beiträgen. An dieser Stelle können Sie festlegen, was Sie damit machen möchten. In diesem Abschnitt werden einige Befehle noch einmal behandelt. Mit diesen können sie genau überprüfen, was Sie einführen, wenn Sie die Beiträge in Ihrem Hauptbranch zusammenführen.

Es ist oft hilfreich, eine Überprüfung über alle Commits zu erhalten, die sich in diesem Branch jedoch nicht in Ihrem Masterbranch befinden. Sie können Commits im Masterbranch ausschließen, indem Sie die Option `--not` vor dem Branchnamen hinzufügen. Dies entspricht dem Format `master..contrib`, welches wir zuvor verwendet haben. Wenn Ihr Mitarbeiter Ihnen bspw. zwei Patches sendet und Sie einen Branch mit dem Namen `contrib` erstellen und diese Patches dort anwenden, können Sie Folgendes ausführen:

```
$ git log contrib --not master
commit 5b6235bd297351589efc4d73316f0a68d484f118
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Oct 24 09:53:59 2008 -0700
```

seeing if this helps the gem

```
commit 7482e0d16d04bea79d0dba8988cc78df655f16a0
Author: Scott Chacon <schacon@gmail.com>
Date:   Mon Oct 22 19:38:36 2008 -0700
```

updated the gemspec to hopefully work better

Denken Sie daran, dass Sie die Option `-p` an `git log` übergeben können, um zu sehen, welche Änderungen jeder Commit einführt.

Um einen vollständigen Überblick darüber zu erhalten, was passieren würde, wenn Sie diesen Branch mit einem anderen Branch zusammenführen würden, müssen Sie möglicherweise einen ungewöhnlichen Kniff anwenden, um die richtigen Ergebnisse zu erzielen. Eventuell denken sie daran folgendes auszuführen:

```
$ git diff master
```

Dieser Befehl gibt Ihnen den Unterschied zurück, jedoch kann dies irreführend sein. Wenn Ihr Masterbranch vorgerückt ist, seit Sie den Themenbranch daraus erstellt haben, erhalten Sie scheinbar unerwartete Ergebnisse. Dies geschieht, weil Git den Snapshots des letzten Commits des Branches, in dem Sie sich befinden, und den Snapshot des letzten Commits des Branches `master` direkt miteinander vergleicht. Wenn Sie bspw. eine Zeile in eine Datei im Branch `master` eingefügt haben, sieht ein direkter Vergleich der Snapshots so aus, als würde der Themen Branch diese Zeile entfernen.

Wenn `master` ein direkter Vorgänger Ihres Themenbranches ist, ist dies kein Problem. Wenn aber beiden Historien voneinander abweichen, sieht es so aus, als würden Sie alle neuen Inhalte in Ihrem Themenbranch hinzufügen und alles entfernen, was im `master` Branch eindeutig ist.

Was Sie wirklich sehen möchten, sind die Änderungen, die dem Themenbranch hinzugefügt wurde. Die Arbeit, die Sie hinzufügen, wenn Sie den neuen Branch mit `master` zusammenführen. Sie tun dies, indem Git das letzte Commit in Ihrem Themen Branch mit dem ersten gemeinsamen Vorgänger aus dem `master` Branch vergleicht.

Technisch gesehen können Sie dies tun, indem Sie den gemeinsamen Vorgänger explizit herausfinden und dann Ihr `diff` darauf ausführen:

```
$ 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 eine andere Weg für diesen Vorgang bietet: die Drei-Punkt-Syntax (engl. Triple-Dot-Syntax). Im Kontext des Befehls `git diff` können Sie drei Punkte nach einem anderen Branch einfügen, um ein `diff` zwischen dem letzten Commit ihres aktuellen Branch und dem gemeinsamen Vorgänger eines anderen Branches zu erstellen:

```
$ git diff master...contrib
```

Dieser Befehl zeigt Ihnen nur die Arbeit an, die Ihr aktueller Branch seit dem gemeinsamen Vorgänger mit `master` eingeführt hat. Dies ist eine sehr nützliche Syntax, die sie sich merken sollten.

## Beiträge integrieren

Wenn Ihr Themenbranch bereit ist, um in einen Hauptbranch integriert zu werden, lautet die Frage, wie Sie dies tun können. Welchen Workflow möchten Sie verwenden, um Ihr Projekt zu verwalten? Sie haben 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 Ihrem `master`-Branch zusammenzuführen. In diesem Szenario haben Sie einen `master`-Branch, der stabilen Code enthält. Wenn Sie in einem Branch arbeiten, von dem Sie glauben, dass Sie ihn abgeschlossen haben, oder von jemand anderem beigesteuert und überprüft haben, führen Sie ihn in Ihrem Hauptbranch zusammen. Löschen Sie anschließend diesen gerade zusammengeföhrt 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 Ihr Verlauf so aus [Status nach einem Themen Branch Merge..](#)

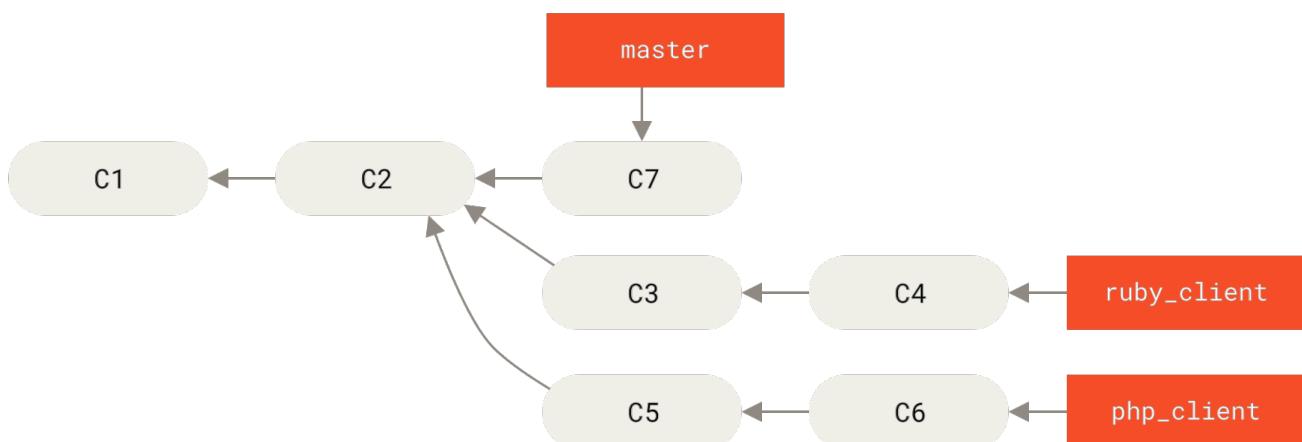


Figure 72. Historie mit mehreren Topic Branches.

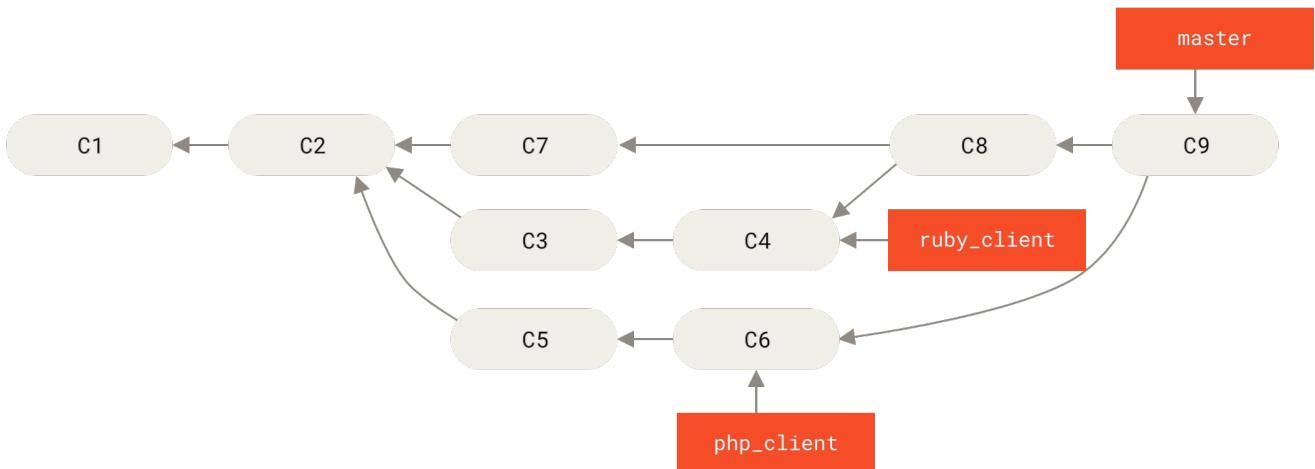


Figure 73. Status nach einem Themen Branch Merge.

Das ist wahrscheinlich der einfachste Workflow. Es kann jedoch zu Problemen können, wenn Sie größere oder stabile Projekte bearbeiten. Bei diesen müssen mit der Einführung von Änderungen sehr vorsichtig sein.

Wenn Sie ein wichtigeres Projekt haben, möchten Sie möglicherweise einen zweistufigen Merge Prozess verwenden. In diesem Szenario haben Sie zwei lange laufende Branches namens `master` und `develop`. Sie legen 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. Sie pushen diese beiden Branches regelmäßig in das öffentliche Repository. Jedes Mal, wenn Sie einen neuen Branch zum Zusammenführen haben ([Vor einem Themen Branch Merge.](#)), führen Sie ihn in `develop` (<<merwf\_d>>) zusammen. Wenn Sie nun ein Release mit einem Tag versehen, spulen Sie `master` an die Stelle weiter, an der sich der jetzt stabile `develop`-Branch befindet ([Nach einem Projekt Release.](#)).

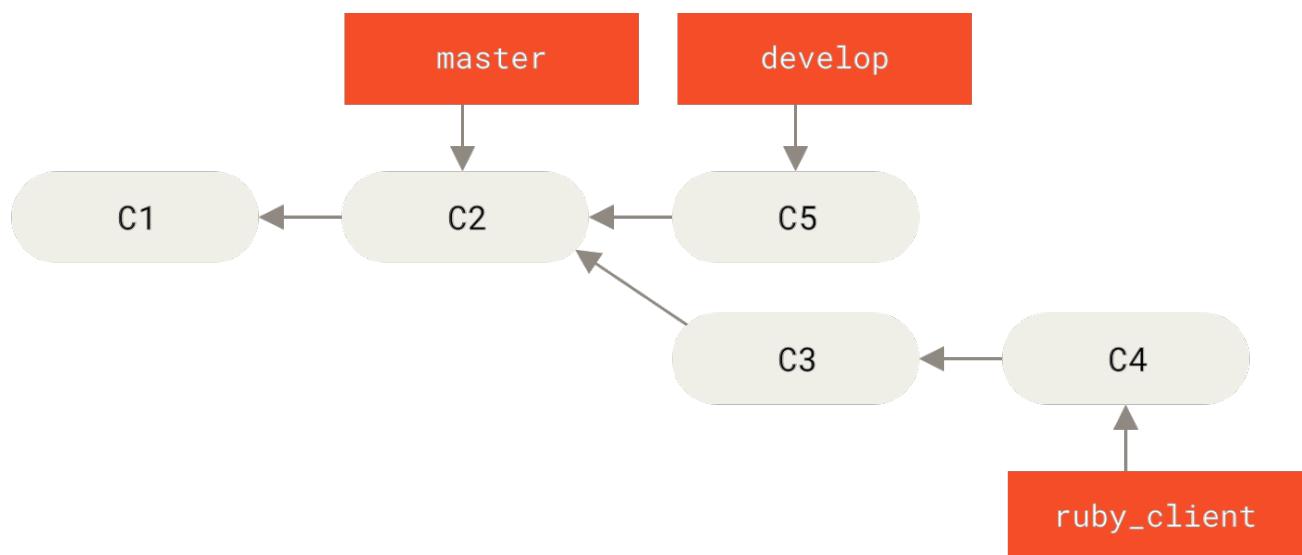


Figure 74. Vor einem Themen Branch Merge.

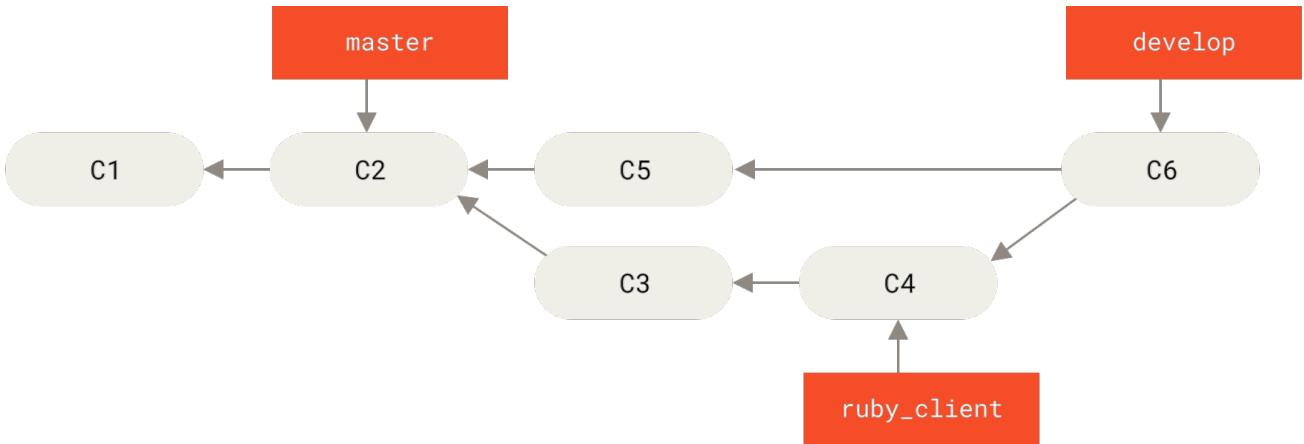


Figure 75. Nach einem Themen Branch Merge.

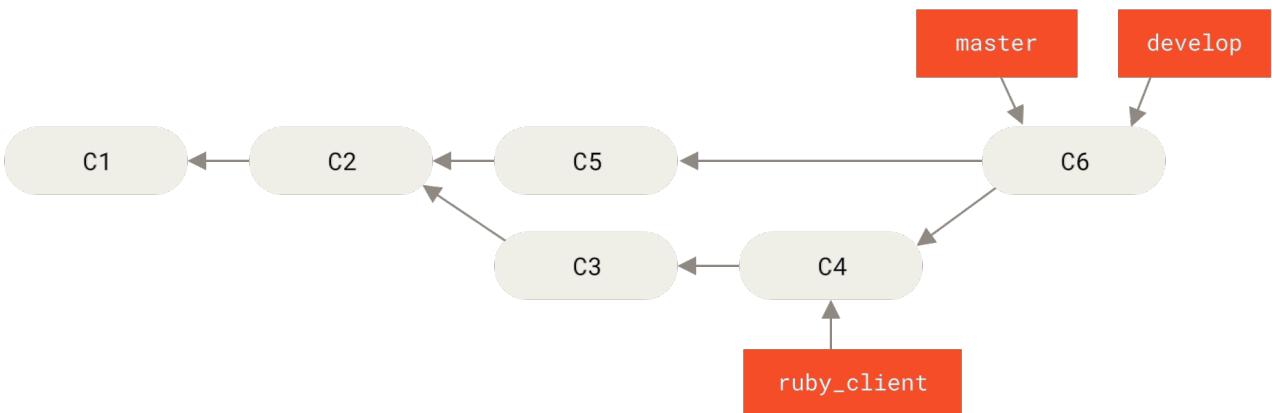


Figure 76. Nach einem Projekt Release.

Auf diese Weise können Benutzer, die das Repository Ihres Projekts klonen, entweder `master` oder `develop` auschecken. Mit `master` können sie die neueste stabile Version erstellen und somit recht einfache auf dem neuesten Stand bleiben. Oder sie können `develop` auschecken, welchen den aktuellsten Inhalt enthält. Sie können dieses Konzept auch erweitern, indem Sie einen `integrate`-Branch einrichten, in dem alle Arbeiten zusammengeführt werden. Wenn die Codebasis auf diesem Branch stabil ist und die Tests erfolgreich sind, können Sie sie zu einem Entwicklungsbranch zusammen führen. Wenn sich dieser dann als stabil erwiesen hat, können sie ihren `master`-Branch fast-forwarden.

## Workflows mit umfangreichen Merges

Das Git-Projekt selber hat vier kontinuierlich laufende Branches: `master`, `next` und `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 Themenbranches im Projektrepository des Betreuers gesammelt (siehe [Verwaltung einer komplexen Reihe paralleler Themenbranches](#)). Zu diesem Zeitpunkt werden die Themen 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 gepushed, damit jeder die miteinander integrierten Themen testen kann.

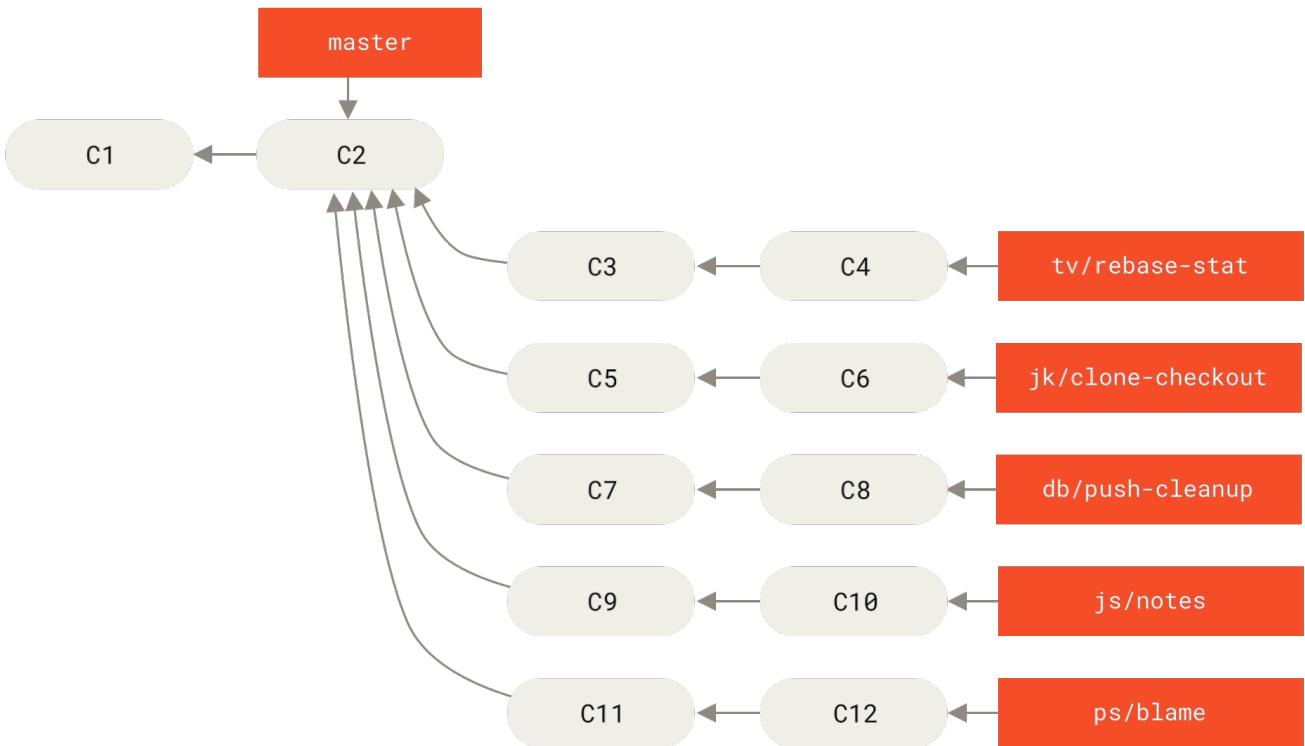


Figure 77. Verwaltung einer komplexen Reihe paralleler Themenbranches.

Wenn die Themen noch bearbeitet werden müssen, werden sie in `pu` gemerged. Wenn festgestellt wird, dass sie absolut stabil sind, werden die Themen wieder zu `master` zusammengeführt. Die Branches `next` und `pu` werden dann von `master` neu aufgebaut. Dies bedeutet, dass `master` fast immer vorwärts geht, `next` wird gelegentlich und `pu` häufiger rebased:

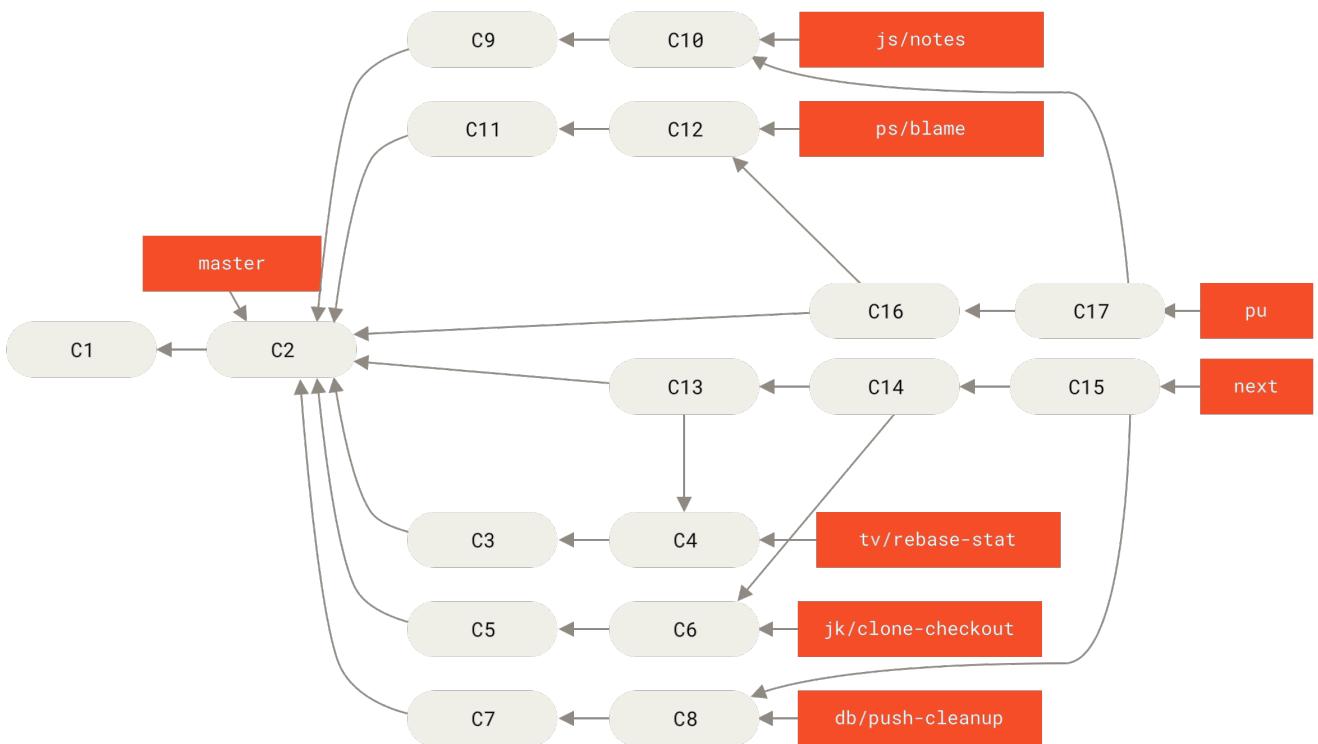


Figure 78. Zusammenführen von Themen Branches in langfristige Integrationsbranches.

Wenn ein Branch schließlich zu `master` zusammengeführt wurde, wird er aus dem Repository entfernt. Das Git-Projekt hat auch einen `maint`-Branch, der von `last release` geforkt wurde, um

für den Fall, dass eine Wartungsversion erforderlich ist, Backport-Patches bereitzustellen. Wenn Sie das Git-Repository also klonen, stehen Ihnen vier Branches zur Verfügung, mit denen Sie das Projekt in verschiedenen Entwicklungsstadien bewerten können, je nachdem, wie aktuell Sie sein möchten oder wie Sie einen Beitrag leisten möchten. 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, können Sie das [Git Maintainer's guide](#) lesen.

## Rebasing und Cherry-Picking Workflows

Andere Betreuer bevorzugen es, die Arbeit auf ihrem Masterbranch zu rebasen oder zu cherry-picken, anstatt sie zusammenzuführen, um einen linearen Verlauf beizubehalten. Wenn Sie in einem Themen Branch arbeiten und sich dazu entschlossen haben, ihn zu integrieren, wechseln Sie in diesen Branch und führen den `rebase` Befehl aus, um die Änderungen auf Ihrem `master` (oder `develop`-Branch usw.) aufzubauen. Wenn das gut funktioniert, können Sie Ihren `master`-Branch fast-forwarden, und Sie erhalten 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 Sie sich gerade befinden. Dies ist nützlich, wenn Sie eine Reihe von Commits für einen Branch haben und nur eine davon integrieren möchten. Oder aber wenn Sie nur einen Commit für einen Branch haben und es vorziehen, diesen zu cherry-picken, anstatt ein Rebase auszuführen. Angenommen, Sie haben ein Projekt, das folgendermaßen aussieht:

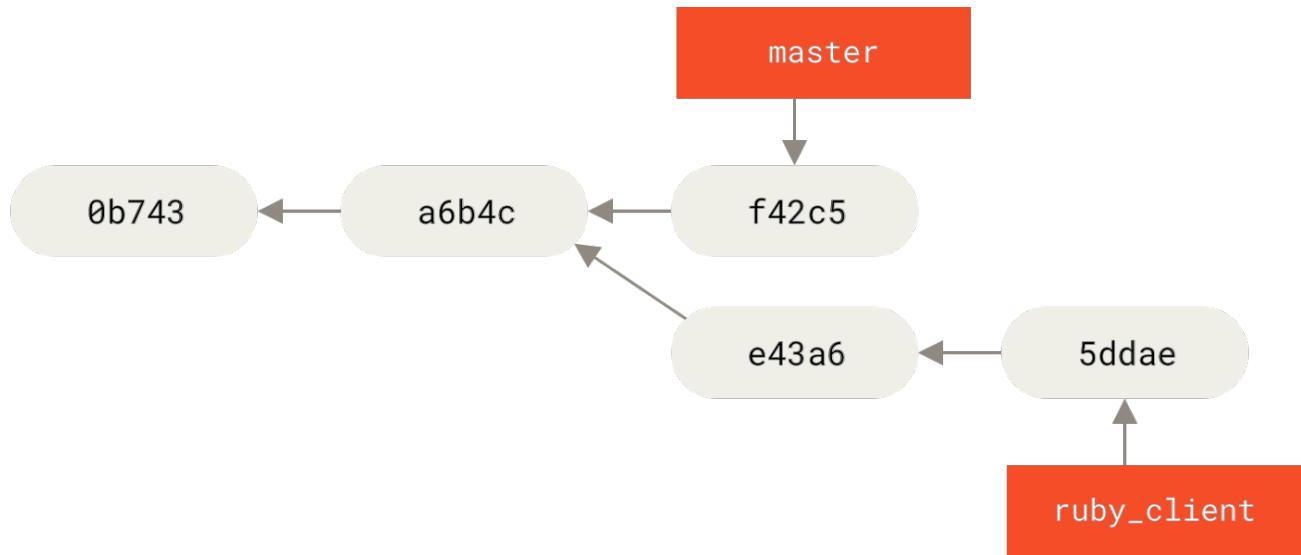


Figure 79. Beispiel Historie vor einem Cherry-Pick.

Wenn Sie das Commit „e43a6“ in Ihren Master-Branch ziehen möchten, können Sie 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. Sie erhalten jedoch einen neuen Commit SHA-1-Wert, da das angewendete Datum unterschiedlich ist. Nun sieht die Historie so aus:

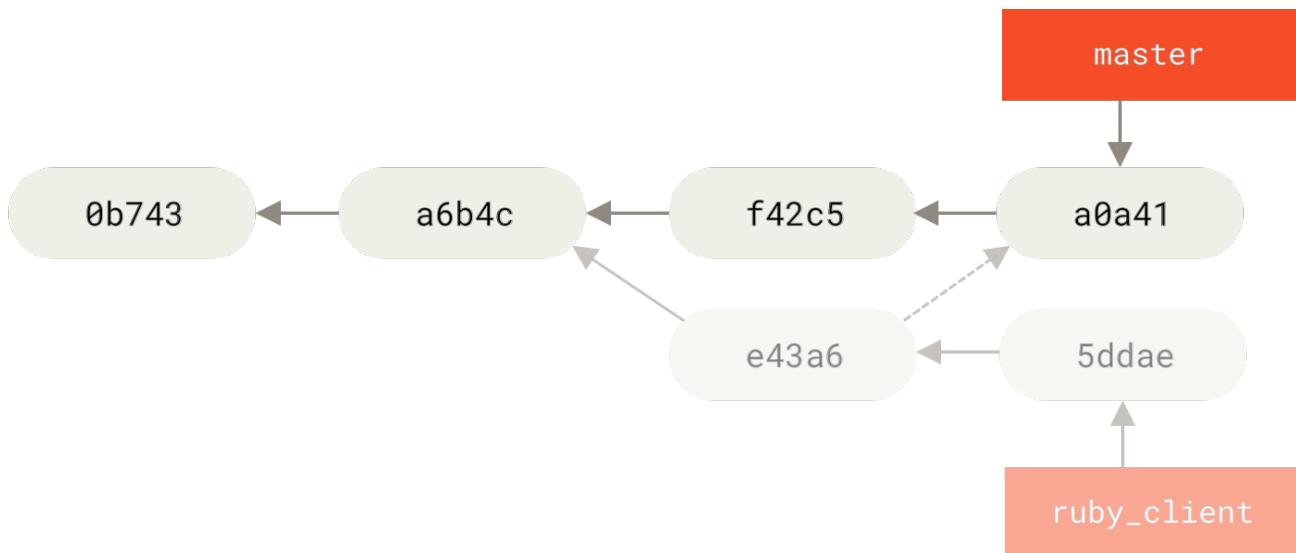


Figure 80. Historie nach Cherry-Picken eines Commits auf einen Themen Branch.

Jetzt können Sie Ihren Themen Branch entfernen und die Commits löschen, die Sie nicht einbeziehen wollten.

## Rerere

Wenn Sie viel mergen und rebasen oder einen langlebigen Themenbranch pflegen, 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 Sie bereits behoben haben, 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 Sie nun einen merge durchführen, der Konflikte auflöst, wird diese Auflösung im Cache gespeichert, falls Sie sie in Zukunft benötigen.

Bei Bedarf können Sie mit dem `rerere` Cache interagieren mittels des Befehls `git rerere`. 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 Unterbefehle, 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 ihres Releases

Wenn Sie sich entschieden haben, ein Release zu erstellen, dann möchten Sie wahrscheinlich einen Tag zuweisen, damit Sie dieses Release in Zukunft jederzeit neu erstellen können. Sie können einen neuen Tag erstellen, wie in [Git Grundlagen](#) beschrieben. Wenn Sie den Tag als Betreuer signieren möchten, 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 Sie Ihre Tags signieren, haben Sie möglicherweise das Problem, den öffentlichen PGP-Schlüssel zu verteilen, der zum Signieren Ihrer 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, können Sie herausfinden, welchen Schlüssel Sie möchten, indem Sie `gpg --list-keys` ausführen:

```
$ 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 können Sie den Schlüssel direkt in die Git-Datenbank importieren, indem Sie ihn exportieren und diesen über `git hash-object` weiterleiten. Dadurch wird ein neuer Blob mit diesen Inhalten in Git geschrieben und Sie erhalten den SHA-1 des Blobs zurück:

```
$ gpg -a --export F721C45A | git hash-object -w --stdin  
659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Nachdem Sie nun den Inhalt Ihres Schlüssels in Git haben, können Sie einen Tag erstellen, der direkt darauf verweist. Dies tun Sie indem Sie den neuen SHA-1-Wert angeben, den Sie mit dem Befehl `hash-object` erhalten haben:

```
$ git tag -a maintainer-pgp-pub 659ef797d181633c87ec71ac3f9ba29fe5775b92
```

Wenn Sie `git push --tags` ausführen, wird der `maintainer-pgp-pub`-Tag für alle freigegeben. Wenn jemand einen Tag verifizieren möchte, kann er Ihren 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 sie alle Ihre signierten Tags überprüfen. Wenn Sie der Tag-Nachricht Anweisungen hinzufügen, können Sie 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 jedes Commit. Wenn sie einen lesbaren Namen für ihren Commit benötigen, dann können Sie für dieses 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 können Sie einen Snapshot exportieren oder einen Build erstellen und einen verständlichen Namen vergeben. Wenn Sie Git aus den Quellcode erstellen, der aus dem Git-Repository geklont wurde, erhalten Sie mit `git --version` etwas, das genauso aussieht. Wenn Sie einen Commit beschreiben, den Sie direkt getaggt haben, erhalten Sie einfach den Tag-Namen.

Standardmäßig erfordert der Befehl `git describe` mit Anmerkungen versehene Tags (Tags, die mit dem Flag `-a` oder `-s` erstellt wurden). Wenn Sie auch leichtgewichtige (nicht mit Anmerkungen versehene) Tags verwenden möchten, fügen Sie dem Befehl die Option `--tags` hinzu. Sie können 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

Nun möchten Sie einen Build freigeben. Eines der Dinge, die Sie tun möchten, ist ein Archiv des neuesten Schnappschusses Ihres Codes für die armen Seelen 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 Schnapschuss Ihres Projekts in einem Projektverzeichnis. Sie können auch ein zip-Archiv auf die gleiche Weise erstellen, indem Sie jedoch die Option `--format=zip` an `git archive` übergeben:

```
$ git archive master --prefix='project/' --format=zip > `git describe master`.zip
```

Sie haben jetzt einen schönen Tarball und ein Zip-Archiv Ihrer Projektversion, die Sie auf Ihre Website hochladen oder per E-Mail an andere Personen senden können.

## Das Shortlog

Es ist Zeit, eine E-Mail an die Personen Ihre Mailingliste zu senden, die wissen möchten, was in Ihrem Projekt vor sich geht. Mit dem Befehl `git shortlog` können Sie schnell eine Art Änderungsprotokoll dessen abrufen, was Ihrem Projekt seit Ihrer letzten Veröffentlichung oder ihrer letzte E-Mail hinzugefügt wurde. Es fasst alle Commits in dem von Ihnen angegebenen Bereich zusammen. Im Folgenden finden Sie als Beispiel eine Zusammenfassung aller Commits seit Ihrer letzten Veröffentlichung, sofern Ihre 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
```

Sie erhalten eine übersichtliche Zusammenfassung aller Commits seit Version 1.0.1, gruppiert nach Autoren, die Sie per E-Mail an Ihre Mailingliste senden können.

## Zusammenfassung

Sie sollten sich nun vertraut damit sein, in einem Git Projekt mitzuwirken, Ihr eigenes Projekt zu pflegen oder die Beiträge anderer Entwickler zu integrieren. Herzlichen Glückwunsch, sie sind nun ein erfolgreichen Git-Entwickler! Im nächsten Kapitel erfahren Sie, wie Sie den größten und beliebtesten Git-Hosting-Dienst, GitHub, verwenden.

# GitHub

GitHub ist der größte Einzelhost für Git-Repositories und der zentrale Punkt der Teamarbeit für Millionen von Entwicklern und Projekten. Ein großer Teil aller Git-Repositories 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 Sie irgendwann mit GitHub in Kontakt treten möchten oder müssen, während Sie Git professionell nutzen.

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-Repositories, gemeinsame Workflows, um zu Projekten beizutragen und Beiträge für Ihre Projekte anzunehmen, die Programmoberfläche von GitHub und viele kleine Tipps, um Ihnen das Leben im allgemeinen zu erleichtern.

Wenn Sie nicht daran interessiert sind, GitHub zu verwenden, um Ihre eigenen Projekte zu hosten oder mit anderen an Projekten zusammenzuarbeiten, die auf GitHub gehostet sind, können Sie getrost zum nächsten Kapitel [Git Tools](#) springen.

## *Schnittstellen Änderung*



Es ist wichtig zu beachten, dass sich die Oberflächenelemente 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 sehen sein, aber wenn Sie aktuellere Versionen dieser Bildschirm-Darstellungen wollen, können die Online-Versionen dieses Buches aktuellere Screenshots enthalten.

## Einrichten und Konfigurieren eines Kontos

Das erste, was Sie tun müssen, ist ein kostenloses Benutzerkonto einzurichten. Besuchen Sie einfach <https://github.com>, wählen Sie einen noch unbenutzten Usernamen, geben Sie eine E-Mail-Adresse und ein Passwort ein und klicken Sie auf die große grüne Schaltfläche „Bei GitHub Anmelden“

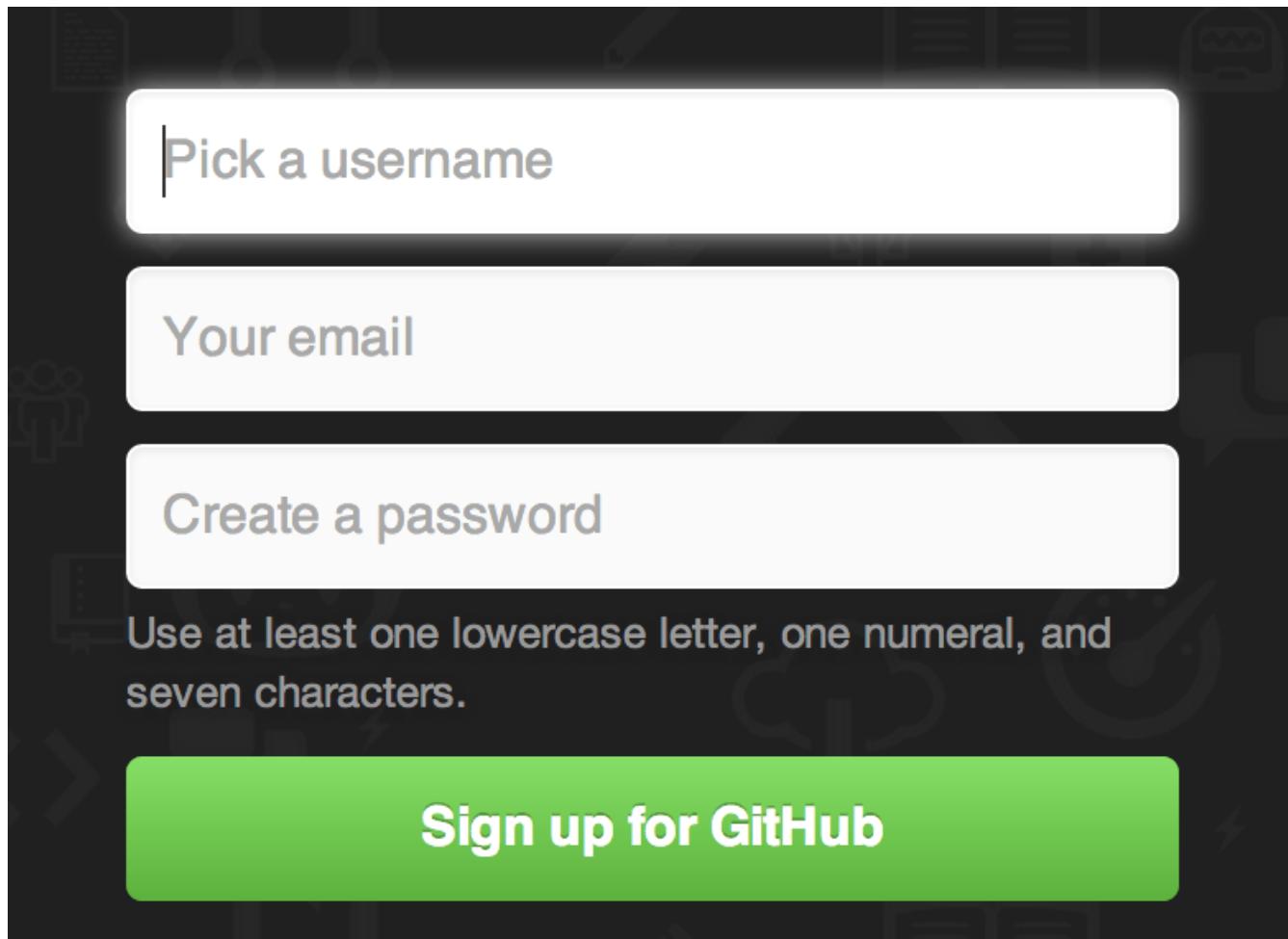


Figure 81. Das GitHub Anmeldeformular

Das nächste, was Sie sehen werden, ist die Preisseite für Upgrade-Pakete. Sie können diese jedoch vorerst ignorieren. GitHub sendet Ihnen eine E-Mail, um die von Ihnen angegebene Adresse zu bestätigen. Fahren Sie fort indem Sie die erhaltene E-Mail bestätigen; das ist ziemlich wichtig, wie wir später sehen werden.



GitHub stellt seine gesamte Funktionalität kostenlos zur Verfügung, mit der Einschränkung, dass alle Ihre Projekte vollständig öffentlich sind (jeder hat Leserechte). Die kostenpflichtigen Angebote von GitHub enthalten auch die Möglichkeit, private Projekte zu erstellen, aber die werden in diesem Buch nicht behandelt.

Wenn Sie auf das Octocat-Logo oben links auf dem Bildschirm klicken, gelangen Sie zu Ihrer Dashboard-Seite. Sie sind ab sofort in der Lage, GitHub zu benutzen.

## SSH-Zugang

Ab sofort können Sie sich uneingeschränkt mit Git-Repositorys über das <https://>-Protokoll verbinden und sich mit dem gerade eingerichteten Benutzernamen und Passwort authentifizieren. Um jedoch öffentliche Projekte einfach zu klonen, müssen Sie sich nicht einmal anmelden – das Konto, das wir gerade erstellt haben, kommt ins Spiel, wenn wir Projekte forken und später zu unseren Forks wechseln.

Wenn Sie SSH-Remotes verwenden möchten, müssen Sie einen öffentlichen Schlüssel konfigurieren

(Wenn Sie noch keinen haben, siehe [öffentlichen SSH-Schlüssel generieren](#)). Öffnen Sie Ihre Kontoeinstellungen über den Link oben rechts im Fenster:

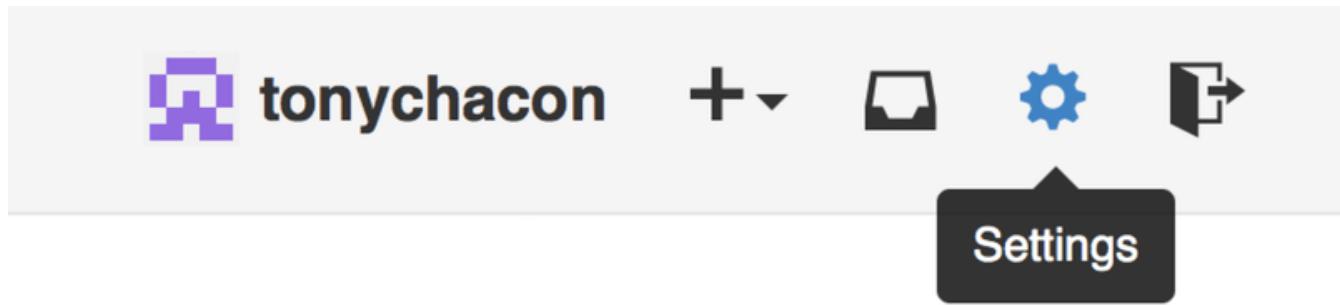


Figure 82. Der Link „Account-Einstellungen“

Wählen Sie dann auf der linken Seite den Bereich „SSH-Schlüssel“.

A screenshot of the 'SSH Keys' section of the GitHub account settings. On the left, there's a sidebar with the same navigation as Figure 82. The 'SSH keys' link is selected. The main area has a heading 'SSH Keys' with a 'Need help?' link and buttons for 'Add SSH key'. Below it says 'There are no SSH keys with access to your account.' A large form titled 'Add an SSH Key' contains fields for 'Title' (with a text input) and 'Key' (with a large text area). At the bottom is a green 'Add key' button.

Figure 83. Der Link „SSH-Schlüssel“

Klicken Sie von dort aus auf die Schaltfläche „Add an SSH key“, geben Sie Ihrem Schlüssel einen Namen, fügen Sie den Inhalt Ihrer `~/.ssh/id_rsa.pub` Public-Key-Datei (oder wie auch immer Sie sie genannt haben) in das Textfeld ein und klicken Sie auf „Add key“.



Achten Sie darauf, dass Sie Ihrem SSH-Schlüssel einen Namen geben, an den Sie sich gut erinnern können. Sie können jeden Ihrer Schlüssel (z.B. „Mein Laptop“ oder „Arbeitskonto“) benennen, so dass Sie, falls Sie einen Schlüssel später widerrufen müssen, leicht erkennen können, nach welchem Sie suchen.

## Ihr Avatar-Bild

Als nächstes können Sie, wenn Sie möchten, den für Sie generierten Avatar durch ein Bild Ihrer

Wahl ersetzen. Gehen Sie zunächst auf die Registerkarte „Profil“ (oberhalb der Registerkarte SSH-Schlüssel) und klicken Sie auf „Neues Bild hochladen“.

The screenshot shows the GitHub profile settings interface. On the left, a sidebar menu is open under the 'Profile' section, listing options like Account settings, Emails, Notification center, Billing, SSH keys, Security, Applications, Repositories, and Organizations. The main content area is titled 'Public profile'. It features a placeholder profile picture with a purple 'X' icon, a button labeled 'Upload new picture', and a note stating 'You can also drag and drop a picture from your computer.' Below this are input fields for 'Name', 'Email (will be public)', 'URL', 'Company', and 'Location'. 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 und anschließend haben wir die Möglichkeit, es zu beschneiden.

The screenshot shows a modal dialog titled 'Crop your new profile picture'. Inside the dialog, there is a red diamond-shaped image of the Git logo, which is being cropped by a white dashed rectangular selection box. The dialog has a close button ('x') in the top right corner and a 'Set new profile picture' button at the bottom right. At the very bottom of the screen, there is a dark grey bar with a green 'Update profile' button.

Figure 85. Ihr Avatar-Bild beschneiden

Nun sehen die Betrachter überall dort, wo Sie auf der Website agieren, Ihr Avatar-Bild neben Ihrem Benutzernamen.

Wenn Sie bei dem beliebten Gravatar-Dienst (der oft für Wordpress-Konten verwendet wird) einen Avatar hochgeladen haben, wird dieser standardmäßig verwendet und Sie müssen diesen Schritt nicht mehr ausführen.

## Ihre Email-Adressen

GitHub bildet Ihre Git Commits auf Ihren Account ab, wobei die Zuordnung per E-Mail erfolgt. Wenn Sie mehrere E-Mail-Adressen in Ihren Commits verwenden und möchten, dass GitHub diese korrekt verknüpft, müssen Sie alle von Ihnen verwendeten E-Mail-Adressen in den E-Mail-Bereich des Admin-Bereichs aufnehmen.

The screenshot shows the 'Email' section of the GitHub account settings. On the left is a sidebar with links: Profile, Account settings, **Emails** (which is selected and highlighted in orange), Notification center, Billing, SSH keys, Security, Applications, Repositories, and Organizations. The main content area has a header 'Email'. It contains a note: 'Your **primary GitHub email address** will be used for account-related notifications (e.g. account changes and billing receipts) as well as any web-based GitHub operations (e.g. edits and merges).'. Below this is a table showing three email addresses:

Email Address	Status	Action
tonychacon@example.com	Primary	trash
tchacon@example.com		Set as primary trash
tony.chacon@example.com	Unverified	Send verification email trash

Below the table is a form to 'Add email address' with a 'Add' button. At the bottom is a checkbox for 'Keep my email address private' with a note: '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](#) können Sie den Status einer E-Mail Adresse ersehen, einige der verschiedenen, möglichen Stadien 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 Sie sie wechseln möchten, als primär eingestellt werden. Die letzte Adresse ist noch nicht verifiziert, was bedeutet, dass Sie sie nicht zu Ihrer Hauptadresse machen können. Wenn GitHub eine davon in Commit-Nachrichten in einem beliebigen Repository auf der Website sieht, wird sie jetzt mit Ihrem Benutzer-Konto verknüpft.

## Zwei-Faktor-Authentifizierung

Schließlich sollten Sie aus Sicherheitsgründen auf jeden Fall die Two-Factor-Authentifizierung oder „2FA“ einrichten. Die Zwei-Faktor-Authentifizierung ist ein Authentifizierungs-Mechanismus, der in letzter Zeit immer beliebter wird, damit das Risiko verringert wird, dass Ihr Account durch den Diebstahl Ihres Passworts Schaden erleidet. Wenn Sie die Funktion einschalten, fragt Sie GitHub nach zwei verschiedenen Authentifizierungsmethoden, so dass ein Angreifer, wenn eine davon beschädigt wird, nicht in der Lage sein wird, auf Ihr Konto zuzugreifen.

Sie finden die Einrichtung der Zwei-Faktor-Authentifizierung unter der Registerkarte „Security“ in

Ihren Kontoeinstellungen.

The screenshot shows the GitHub user interface for security settings. On the left, there's a sidebar with links: 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'. Below it is a button 'Set up two-factor authentication'. A note explains that two-factor authentication provides another layer of security to your account, with a link to 'GitHub Help'. Another section titled 'Sessions' lists devices that have logged into the account. It shows one entry: 'Paris 85.168.227.34' (Your current session), which is a 'Safari' browser on OS X 10.9.4 located in Paris, Ile-de-France, France, signed in on September 30, 2014.

Figure 87. „2FA“ auf der Security-Registerkarte

Wenn Sie auf die Schaltfläche „Zwei-Faktor-Authentifizierung einrichten“ klicken, gelangen Sie zu einer Konfigurationsseite, auf der Sie eine Handy-App wählen können, um Ihren sekundären Code zu generieren (ein „zeitbasiertes Einmalpasswort“), oder Sie können sich bei jedem Login von GitHub einen Code per SMS zusenden lassen.

Nachdem Sie sich für eine der beiden Methoden entschieden haben und den Anweisungen zur Einrichtung von 2FA gefolgt sind, ist Ihr Konto etwas sicherer und Sie müssen bei jedem Login in GitHub einen zusätzlichen Code neben Ihrem Passwort eingeben.

## Mitwirken an einem Projekt

Nun, da unser Konto eingerichtet ist, lassen Sie uns einige Details durchgehen, die nützlich sein könnten, um Ihnen zu helfen, zu einem bestehenden Projekt beizutragen.

### Forken von Projekten

Wenn Sie zu einem bestehenden Projekt beitragen möchten, zu dem Sie keinen Push-Zugang haben, können Sie das Projekt „forken“. Wenn Sie ein Projekt „forken“, erstellt GitHub eine Kopie des Projekts, die ganz Ihnen gehört; es befindet sich in Ihrem Namensraum (engl. namespace), und Sie können Daten dorthin „hochladen“ (engl. push).



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 aufgespalten hat. In GitHub ist ein „Fork“ schlichtweg das gleiche Projekt in Ihrem eigenen Namensraum, so dass Sie Änderungen an einem Projekt öffentlich vornehmen können, um einen transparenten Ansatz zu verfolgen.

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), gehen Sie auf die Projektseite und klicken Sie auf die Schaltfläche „Fork“ oben rechts auf der Seite.



Figure 88. Die Schaltfläche „Fork“

Nach ein paar Sekunden werden Sie auf Ihre neue Projektseite weitergeleitet, mit Ihrer 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 Sie eng in einem Team, in einem einzigen gemeinsamen Repository, mit einem global verteilten Unternehmen oder einem Netzwerk von Fremden, über Dutzende von Forks, zusammenarbeiten und zu einem Projekt beitragen. Es ist um den Workflow aus [Themen-Banches](#) konzentriert, der in Kapitel 3 [Git Branching](#) ausführlich besprochen wird.

Im Prinzip funktioniert der Ablauf so:

1. Forken Sie das Projekt
2. Erstellen Sie lokal einen Themen-Branch aus [master](#).
3. Machen Sie einige Commits, um das Projekt zu überarbeiten.
4. Pushen Sie diesen Branch zu Ihrem GitHub-Projekt.
5. Eröffnen Sie einen Pull-Request auf GitHub.
6. Diskutieren Sie die optionale Fortsetzung des Commits.
7. Der Projekteigentümer mergt oder schließt den Pull Request.
8. Synchronisieren Sie den aktualisierten Master wieder mit Ihrem 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 Anpassung an einem Open-Source-Projekt vorschlägt, das auf GitHub gehostet wird.

## 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.

*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. Lassen Sie uns also das Programm verbessern und es als Änderungsvorschlag an das Projekt zurücksenden.

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 Themenzweig erstellen, den Code ändern und schließlich diese Änderung wieder auf GitHub übertragen.

```

$ git clone https://github.com/tonychacon/blink ①
Cloning into 'blink'...

$ cd blink
$ git checkout -b slow-blink ②
Switched to a new branch 'slow-blink'

$ sed -i '' 's/1000/3000/' blink.ino (macOS) ③
# If you're on a Linux system, do this instead:
# $ sed -i 's/1000/3000/' blink.ino ③

$ git diff --word-diff ④
diff --git a/blink.ino b/blink.ino
index 15b9911..a6cc5a5 100644
--- a/blink.ino
+++ b/blink.ino
@@ -18,7 +18,7 @@ void setup() {
// the loop routine runs over and over again forever:
void loop() {
    digitalWrite(led, HIGH); // turn the LED on (HIGH is the voltage level)
    [-delay(1000);-]{+delay(3000);+} // wait for a second
    digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
    [-delay(1000);-]{+delay(3000);+} // wait for a second
}

$ git commit -a -m 'three seconds is better' ⑤
[slow-blink 5ca509d] three seconds is better
 1 file changed, 2 insertions(+), 2 deletions(-)

$ git push origin slow-blink ⑥
Username for 'https://github.com': tonychacon
Password for 'https://tonychacon@github.com':
Counting objects: 5, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 340 bytes | 0 bytes/s, done.
Total 3 (delta 1), reused 0 (delta 0)
To https://github.com/tonychacon/blink
 * [new branch]      slow-blink -> slow-blink

```

① Wir klonen unsere 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 übernehmen (engl. commit) unsere Änderung in den Themen-Branch

⑥ Wir pushen Sie unseren neuen Themen-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 Themenzweig gepusht haben und zeigt uns einen großen grünen Button, um unsere Änderungen zu überprüfen und einen Pull Request zum ursprünglichen Projekt zu öffnen.

Sie können alternativ auch die Seite „Branches“ bei <https://github.com/<user>/<project>/branches> aufzurufen, um Ihre Branch auszuwählen und von dort aus einen Pull Request zu öffnen.

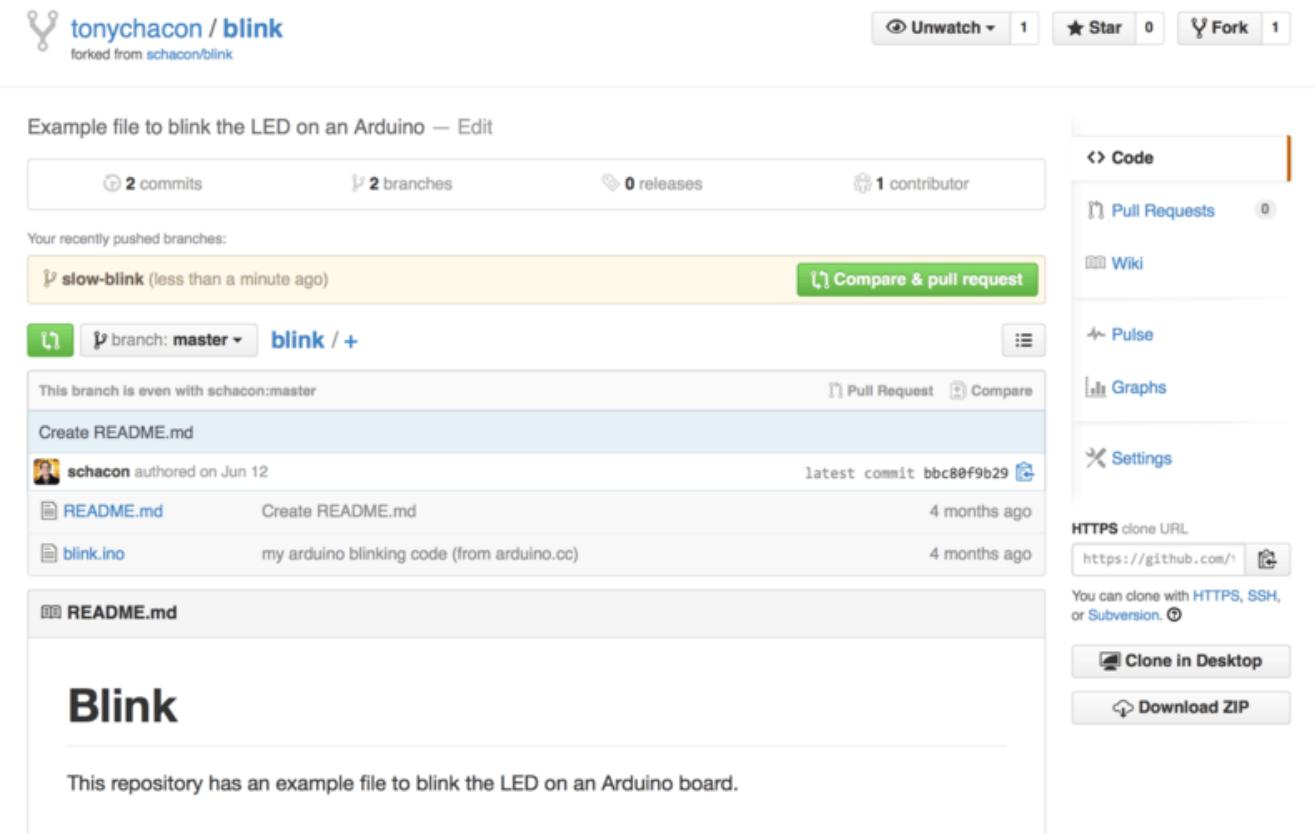
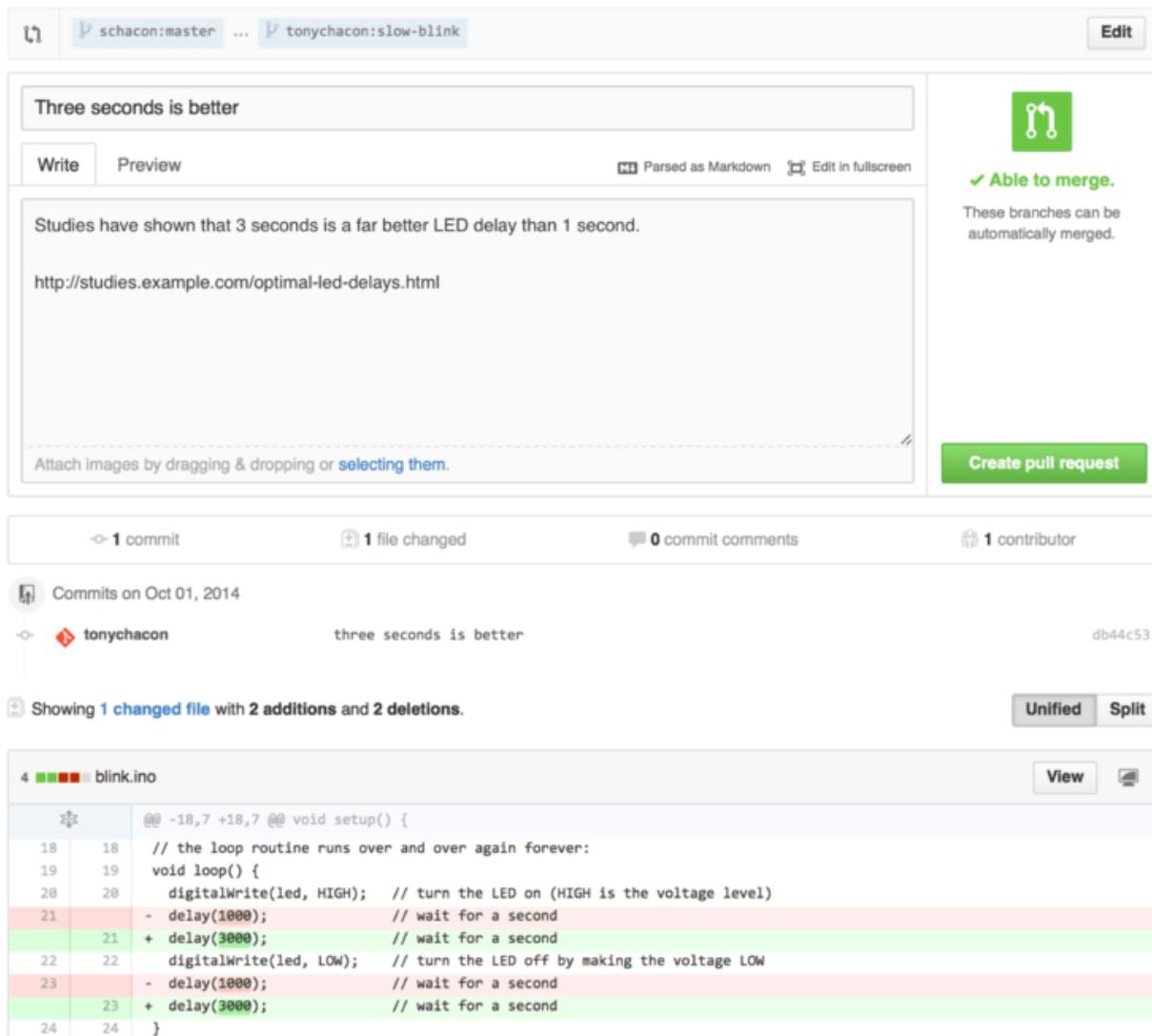


Figure 90. Die Schaltfläche Pull Request

Wenn wir auf die grüne Schaltfläche klicken, sehen wir einen Bildschirm, auf dem Sie aufgefordert werden, Ihrem Pull Request einen Titel und eine Beschreibung zu geben. Es ist fast immer sinnvoll, sich ein bisschen Mühe zu geben, da eine gute Beschreibung dem Eigentümer des ursprünglichen Projekts hilft, festzustellen, was Sie versucht haben, ob Ihre vorgeschlagenen Änderungen korrekt sind und ob die Annahme der Änderungen das ursprüngliche Projekt verbessern würde.

Wir erhalten auch eine Liste der Commits in unserem Themen-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 gemerget wird.



The screenshot shows a GitHub pull request interface. At the top, there's a header with the repository name 'tonychacon / blink' and a forked from 'schacon/blink' note. To the right are buttons for 'Unwatch', 'Star', and 'Fork'. Below the header, the pull request details are shown: 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.' and provides a link: 'http://studies.example.com/optimal-led-delays.html'. On the right side, there's a green icon indicating 'Able to merge' with the note: 'These branches can be automatically merged.' A 'Create pull request' button is visible. Below the main description, there's a summary of the changes: '1 commit', '1 file changed', '0 commit comments', and '1 contributor'. The commit details show it was made on Oct 01, 2014, by 'tonychacon' with the commit message 'three seconds is better' and hash 'db44c53'. The 'diff' section shows a single file 'blink.ino' with 2 additions and 2 deletions. The diff highlights changes in lines 18, 21, 22, and 23.

```

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 Sie in diesem Fenster auf die Schaltfläche *Create pull request* klicken, wird der Eigentümer/Leiter des Projekts, für das Sie eine Abspaltung (engl. Fork) vorgenommen haben, benachrichtigt, dass jemand eine Änderung vorschlägt, und verlinkt auf eine Seite, die alle diese Informationen enthält.



Pull-Requests werden zwar häufig für öffentliche Projekte wie dieses verwendet, wenn der Beitragende eine vollständige Änderung fertig hat, sie werden jedoch auch zu Beginn des Entwicklungszyklus in internen Projekten verwendet. Da Sie den Themenzweig auch **nach** dem Öffnen des Pull-Requests weiter bearbeiten können, wird er oft früh geöffnet und als Möglichkeit genutzt, 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.

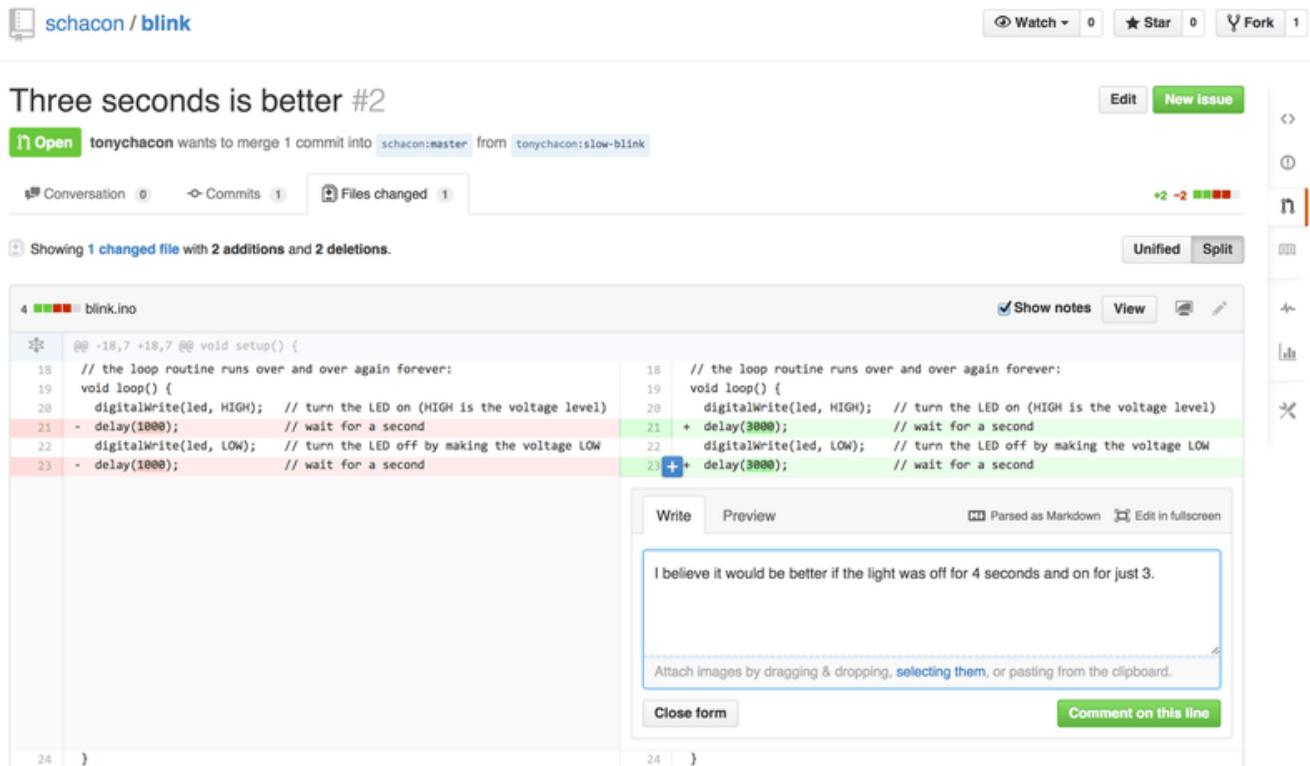


Figure 92. Kommentar zu einer bestimmten Codezeile in einem Pull-Request

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. Sie sehen,

dass auch die Code-Kommentare in das Diskussionsfenster eingebracht werden.

## Three seconds is better #2

The screenshot shows a GitHub pull request page for a file named 'blink.ino'. A comment by 'tonychacon' states: 'Studies have shown that 3 seconds is a far better LED delay than 1 second.' Below this, a commit titled 'three seconds is better' is shown with a diff. The diff highlights changes in line 23:

```
22    22    digitalWrite(led, LOW); // turn the LED off by making the voltage LOW
23 - delay(1000); // wait for a second
23 + delay(3000); // wait for a second
```

A note from 'schacon' suggests changing the delay to 4 seconds. Another comment from 'schacon' expresses willingness to merge if the change is made. The right sidebar shows standard GitHub project management options like Labels, Milestone, Assignee, Notifications, and Participants.

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 Sie über E-Mail Ihre Daten erneut in die Mailingliste eintragen müssen, committen Sie mit GitHub einfach erneut in den Themen-Branch und pushen, wodurch der Pull-Request automatisch aktualisiert wird. Im [finalen Pull-Request](#) sehen Sie auch, dass der alte Code-Kommentar in dem aktualisierten Pull-Request zusammengeklappt wurde, da er auf eine inzwischen geänderte Zeile gemacht wurde.

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, there's a green button labeled "Open" and a status bar indicating "tonychacon wants to merge 3 commits into schacon:master from tonychacon:slow-blink". Below this, there are tabs for "Conversation" (3), "Commits" (3), and "Files changed" (1). The main area shows a conversation between tonychacon and schacon. tonychacon commented 11 minutes ago, linking to a study about LED delays: <http://studies.example.com/optimal-led-delays.html>. schacon commented on an outdated diff 5 minutes ago, with a link to "Show outdated diff". tonychacon added some commits 2 minutes ago, including "longer off time" and "remove trailing whitespace". tonychacon commented 10 seconds ago, saying "I changed it to 4 seconds and also removed some trailing whitespace that I found. Anything else you would like me to do?". A note at the bottom states "This pull request can be automatically merged. You can also merge branches on the command line." with a "Merge pull request" button.

Figure 95. finaler Pull-Request

Eine bemerkenswerte Besonderheit ist, dass Sie den „vereinheitlichten“ Diff erhalten, wenn Sie auf die Registerkarte „Files Changed“ in diesem Pull-Request klicken – d.h. die gesamte aggregierte Differenz, die in Ihren Hauptbranch eingebracht würde, wenn dieser Themenzweig gemitigt würde. Im Sinne von `git diff` zeigt es Ihnen 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 Diff.

Außerdem prüft GitHub, ob der Pull-Request sauber mergen würde und stellt eine Schaltfläche zur Verfügung, mit der Sie den Merge auf dem Server durchführen können. Diese Schaltfläche erscheint nur, wenn Sie Schreibzugriff auf das Repository haben und ein trivialer Merge möglich ist. Wenn Sie darauf klicken, 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 Sie möchten, können Sie den Branch einfach herunterladen (engl. pull) und lokal zusammenführen (engl. merge). Wenn Sie diesen Zweig mit dem **master** Branch verschmelzen und ihn nach GitHub pushen, wird der Pull Request automatisch geschlossen.

Das ist der grundsätzliche Workflow, den die meisten GitHub-Projekte verwenden. Themen-Branches werden erstellt, Pull-Requests werden geöffnet, es folgt eine Diskussion, möglicherweise wird eine weitere Überarbeitung des Zweiges durchgeführt und schließlich wird der Request entweder geschlossen oder zusammengeführt.

#### *Nicht nur Forks*



Es ist wichtig zu erwähnen, dass Sie auch einen Pull-Request zwischen zwei Zweigen im selben Repository öffnen können. Wenn Sie mit jemandem an einer Funktion arbeiten und beide Schreibrechte auf das Projekt haben, können Sie einen Themenzweig 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 Ihnen einige interessante Tipps und Tricks zu Pull-Requests geben, damit Sie diese effektiver nutzen können.

### 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, wobei die meisten Mailinglisten-basierten Projekte an Patch-Serienbeiträge denken. Die meisten GitHub-Projekte betrachten Pull-Request-Branches als iterative Dialoge um eine beabsichtigte Änderung, die zu einem vereinheitlichten Diff führt, welches durch das Mergen angewendet wird.

Das ist ein wichtiger Unterschied, denn im Allgemeinen wird die Änderung vorgeschlagen, bevor der Code perfektioniert ist, was bei Beiträgen von Patch-Serien auf Mailinglistenbasis weitaus seltener ist. So kann früher mit den Betreuern gesprochen werden, damit die richtige Lösung eher eine Gemeinschaftsarbeit ist. 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, sondern der Unterschied wird als neuer Commit an den Branch weitergegeben (gepusht), wodurch der Dialog im intakten Kontext der vorherigen Arbeit fortgesetzt wird.

Wenn Sie beispielsweise zurückgehen und sich den **finalen Pull-Request** erneut ansehen, werden Sie feststellen, dass der Beitragende seinen Commit nicht umbasert hat (engl. rebase) und einen weiteren Pull-Request geöffnet hat. Stattdessen wurden neue Commits hinzugefügt und sie in den bestehenden Zweig gepusht. Wenn Sie also in Zukunft auf diesen Pull Request zurückblicken, können Sie leicht den gesamten Kontext finden, in dem die Entscheidungen getroffen wurden. Wenn Sie auf der Website auf die Schaltfläche „Merge“ klicken, wird gezielt ein Merge-Commit erstellt, der auf den Pull-Request verweist, so dass Sie leicht zurückkehren und bei Bedarf die ursprüngliche Diskussion durchsuchen können.

## Mit dem Upstream Schritt halten

Wenn Ihr Pull-Request veraltet ist oder anderweitig nicht sauber zusammengeführt wird, sollten Sie ihn reparieren, damit der Betreuer ihn leicht mergen kann. GitHub wird das für Sie testen und Sie am Ende jedes Pull Requests darüber informieren, ob das Merge trivial ist oder nicht.



Figure 96. Pull Request lässt sich nicht sauber mergen

Wenn Sie etwa „[Pull Request lässt sich nicht sauber mergen](#)“ sehen, sollten Sie Ihren Branch so reparieren, dass er grün wird und der Maintainer keine zusätzliche Arbeit leisten muss.

Sie haben zwei grundsätzliche Möglichkeiten, wie Sie das realisieren können. Sie können Ihren Branch entweder auf den Ziel-Branch rebasen (normalerweise den `master`-Branch des von Ihnen geforkten Repositorys), oder Sie können den Ziel-Branch mit Ihrem 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 Sie im Ziel-Branch mergen wollen, um Ihren Pull-Request zusammenzuführen zu können, sollten Sie das ursprüngliche Repository als neuen Remote hinzufügen. Dann machen Sie ein `git fetch` davon, führen den Hauptzweig dieses Repositorys in Ihren Themen-Branch zusammen, beheben alle Probleme und pushen Sie es schließlich wieder in den gleichen Branch, in dem Sie den Pull-Request geöffnet hatten.

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 Themen-Branch mergern.
- ④ Den aufgetretenen Konflikt beheben
- ⑤ Zum gleichen Themen-Branch zurück pushen

Sobald Sie das getan haben, wird der Pull-Request automatisch aktualisiert und erneut überprüft, um festzustellen, ob er sauber zusammengeführt werden kann.

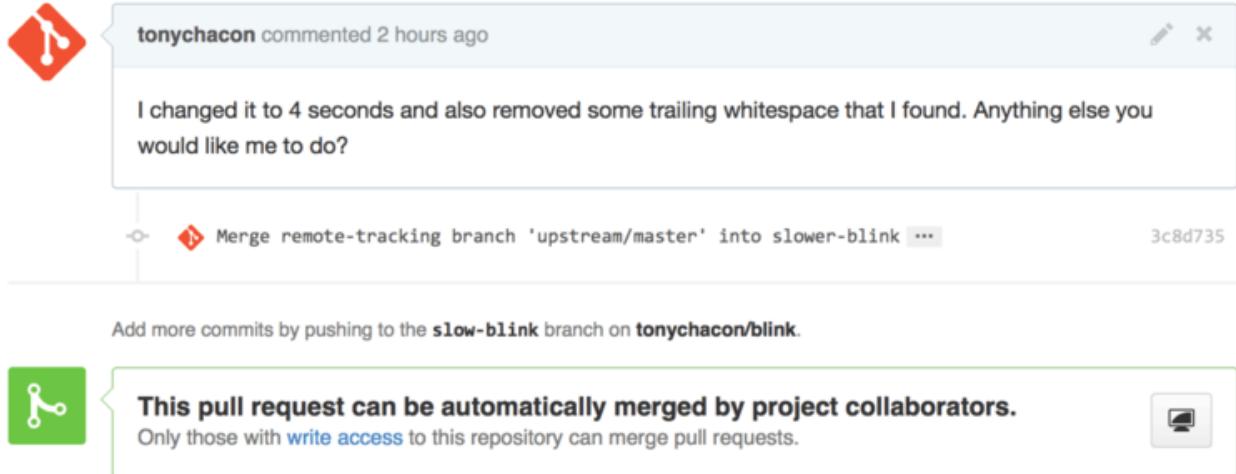


Figure 97. Pull-Request wird nun sauber zusammengeführt

Einer der großen Pluspunkte von Git ist, dass man das kontinuierlich tun kann. Wenn Sie ein sehr lang laufendes Projekt haben, können Sie leicht immer wieder aus dem Zielbranch heraus mergen und müssen sich nur mit Konflikten befassen, die seit dem letzten Mal, als Sie zusammengeführt haben, aufgetreten sind; was den Prozess sehr überschaubar macht.

Wenn Sie den Branch unbedingt rebasen wollen, um ihn aufzuräumen, können Sie das natürlich tun, aber es wird dringend empfohlen, den Branch, auf dem der Pull-Request bereits geöffnet ist, nicht zwangsweise zu 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. Schieben Sie stattdessen den rebasierten Branch zu einem neuen Branch auf GitHub und öffnen Sie einen brandneuen Pull Request mit Bezug auf den alten Branch und schließen Sie dann das Original.

## Referenzen

Möglicherweise lautet Ihre 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 Sie auf einen Pull-Request oder ein Issue von einem anderen verweisen möchten, können Sie einfach #<num> in einen Kommentar oder eine Beschreibung eingeben. Sie können auch präziser sein, wenn die Issue- oder Pull-Anforderung an einem anderen Ort liegt; schreiben Sie Benutzername#<num>, wenn Sie sich auf ein Issue oder Pull Request beziehen, in einen Fork des Repositories, in dem Sie sich befinden, oder 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 rebasiert, 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 in der 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.

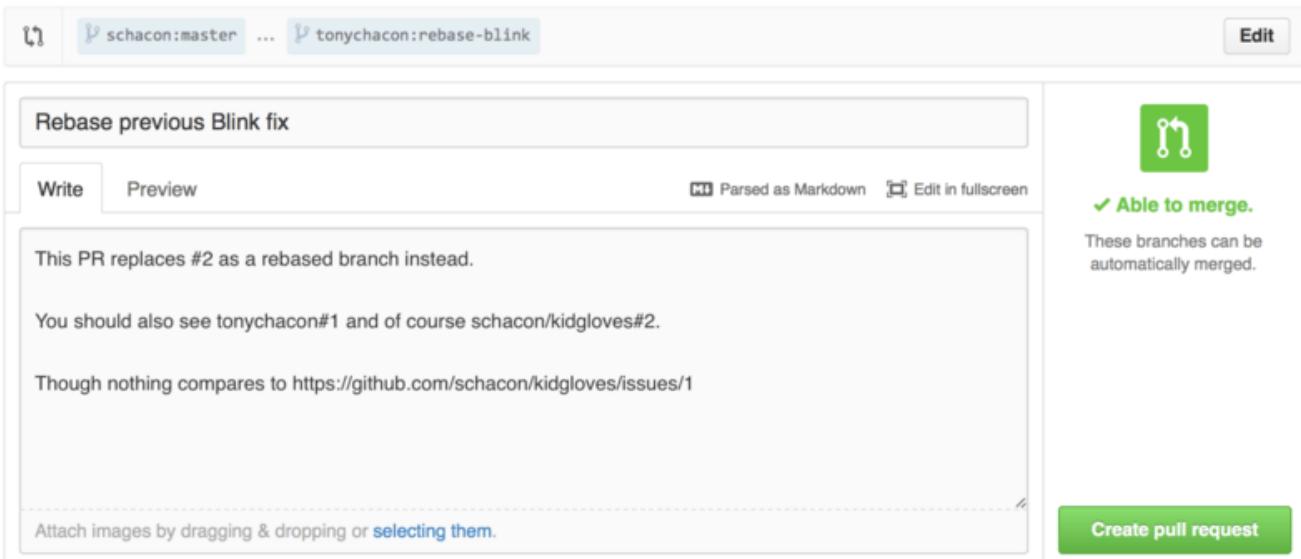


Figure 98. Querverweise in einem Pull-Request

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 the detailed view of a GitHub pull request. At the top, there's an 'Open' button and a note that 'tonychacon wants to merge 2 commits into schacon:master from tonychacon:rebase-blink'. Below this are three tabs: 'Conversation' (0), 'Commits' (2), and 'Files changed' (1). A comment from 'tonychacon' is highlighted, reading: '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, there's a commit history showing 'tonychacon added some commits 4 hours ago' with two commits: 'three seconds is better' (afe904a) and 'remove trailing whitespace' (a5a7751).

Figure 99. Querverweise, die in einem Pull-Request erzeugt wurden

Bitte beachten Sie, 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.

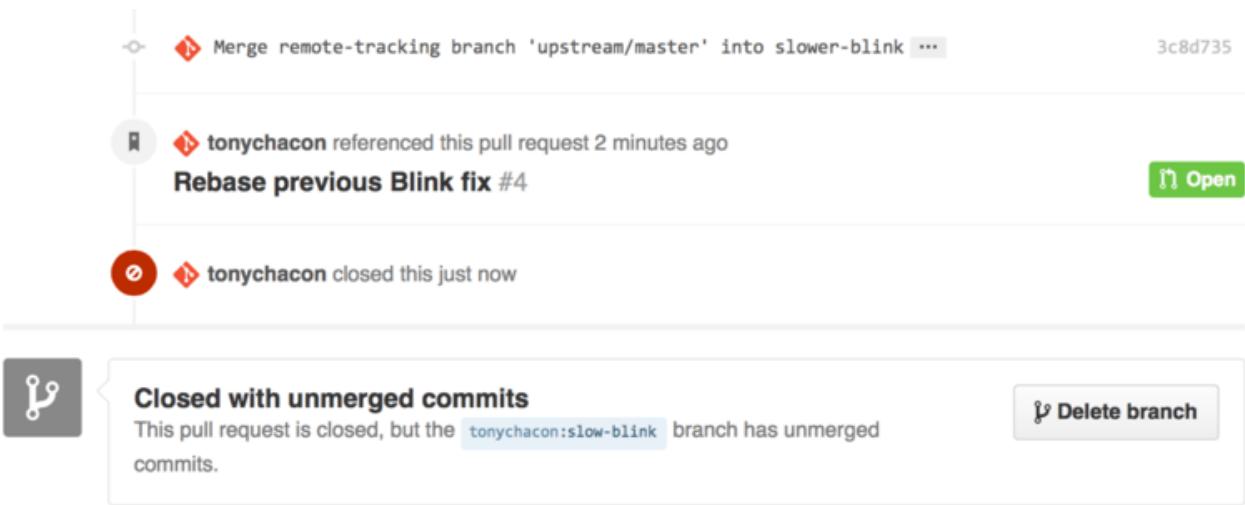


Figure 100. Zurück zum neuen Pull-Request in der geschlossenen Pull-Request Zeitleiste

Zusätzlich zu den Issue-Nummern können Sie auch auf einen bestimmten Commit per SHA-1 referenzieren. Sie müssen einen vollen 40-stelligen SHA-1 angeben, aber wenn GitHub das in einem Kommentar sieht, wird er direkt auf den Commit verlinken. Wie bei Issues können Sie 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 Sie mit fast jeder Textbox auf GitHub machen können. In Issue- und Pull-Request-Beschreibungen, Kommentaren, Code-Kommentaren und mehr, können Sie das sogenannte „GitHub Flavored Markdown“ verwenden. Markdown fühlt sich an wie das Schreiben in Klartext, hat aber umfangreiche Dargestellungs-Optionen.

Im [Beispiel für GitHub-Variante von Markdown, geschrieben und gerendert](#) finden Sie ein Muster, wie Kommentare oder Text geschrieben und dann mit Markdown gerendert werden können.

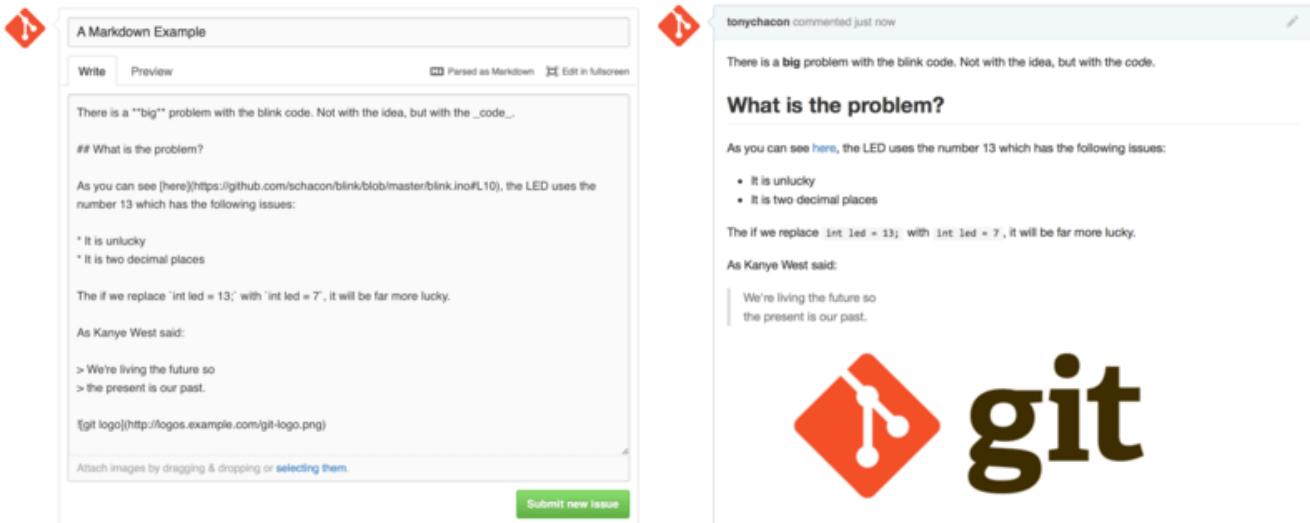


Figure 101. Beispiel für GitHub-Variante von Markdown, geschrieben und gerendert

Die GitHub-Variante von Markdown bietet mehr Möglichkeiten, als die normale Markdown-Syntax. Alle diese Funktionen können sehr nützlich sein, wenn Sie hilfreiche Pull-Request, Issue-Kommentare oder Beschreibungen erstellen.

## 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 Sie erledigen möchten. Wenn Sie sie in einen Issue oder Pull-Request einfügen, werden normalerweise Punkte angezeigt, die Sie erledigen sollten, bevor Sie den Vorgang als erledigt betrachten.

Sie können 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 alles Sie auf dem Branch erledigen möchten, bevor der Pull Request bereit für die Zusammenführung ist. Der wirklich coole Teil ist, dass Sie einfach auf die Kontrollkästchen klicken können, um den Kommentar zu aktualisieren – Sie müssen den Markdown nicht direkt bearbeiten, um Aufgaben abzuwählen.

Darüber hinaus sucht GitHub nach Aufgabenlisten in Ihren Issues und Pull Requests und sie als Metadaten auf den aufgelisteten Seiten anzeigt. Wenn Sie z.B. einen Pull-Request mit Aufgabenliste haben und sich die Übersichtsseite aller Pull-Requests ansehen, können Sie sehen, wie weit er abgearbeitet ist. Das hilft den Teilnehmern, Pull-Requests in Teilaufgaben aufzuschlüsseln und anderen Teilnehmern, den Fortschritt der Branch zu verfolgen. Ein Beispiel dafür finden Sie bei: [Task-Liste \(Zusammenfassung\) in der Pull-Request-Liste](#).

A screenshot of the GitHub pull request list. At the top, there are filters: '2 Open' and '1 Closed'. Below the filters, two pull requests are listed:

- #4 Change blink time to four seconds opened 4 hours ago by tonychacon 2 of 3
- #2 Three seconds is better opened 7 hours ago by tonychacon 3

The first pull request has a task list with one item checked: "Change blink time to four seconds". The second pull request has a task list with one item checked: "Three seconds is better". The entire list is enclosed in a light gray box with rounded corners.

Figure 103. Task-Liste (Zusammenfassung) in der Pull-Request-Liste

Diese Funktion ist unglaublich nützlich, wenn Sie einen Pull Request frühzeitig öffnen und damit Ihren Fortschritt bei der Realisierung des Features verfolgen (engl. track).

## Code-Schnipsel

Sie können auch Code-Schnipsel zu Kommentaren hinzufügen. Es ist dann besonders praktisch, wenn Sie etwas präsentieren möchten, das Sie versuchen *könnten*, bevor Sie es tatsächlich als Commit auf Ihrem Branch einbauen. 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, müssen Sie ihn in zwei 3-fach Backticks (```) „einzäunen“.

```
```java
for(int i=0 ; i < 5 ; i++)
{
    System.out.println("i is : " + i);
}
```

```

Wenn Sie den Namen der Programmiersprache hinzufügen, 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 Sie auf einen kleinen Teil eines langen Kommentars reagieren wollen, können Sie selektiv aus dem anderen Kommentar zitieren, indem Sie den Zeilen das Zeichen > voranstellen. Es ist sogar so häufig und nützlich, dass es eine Tastenkombination dafür gibt. Wenn Sie Text in einem Kommentar markieren, auf den Sie direkt antworten möchten, und die Taste **r** drücken, wird dieser Text in der Kommentarbox für Sie 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: [gerenderes Zitat-Beispiel..](#)

The screenshot shows two GitHub comments. The first comment is by user schacon, posted 2 minutes ago. It contains a rendered quote from Shakespeare's 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 is by user tonychacon, posted 10 seconds ago. It contains a partial quote and a question:

```
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. gerenderes Zitat-Beispiel.

## Emojis

Abschließend, Sie können auch Emojis in Ihren Kommentaren verwenden. Das wird in der Praxis sehr häufig in Kommentaren verwendet, die Sie es bei vielen GitHub-Issues und Pull-Requests sehen. Es gibt sogar einen Emoji-Helfer in GitHub. Wenn Sie einen Kommentar eingeben und mit einem : (Doppelpunkt) beginnen, hilft Ihnen ein Autokomplettierer, das Gesuchte schnell zu finden.

The screenshot shows the GitHub emoji autocompletion interface. A user has typed ':jo' into the comment input field. A dropdown menu appears, listing several emoji suggestions:

- joy (smiling face)
- joy\_cat (smiling cat)
- black\_joker (joker)
- smile (smiling face)
- smiley (smiling face)

The 'joy\_cat' option is highlighted with a blue background. At the bottom right of the input field 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 könnten Sie so ähnlich wie hier schreiben:

```
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](#).

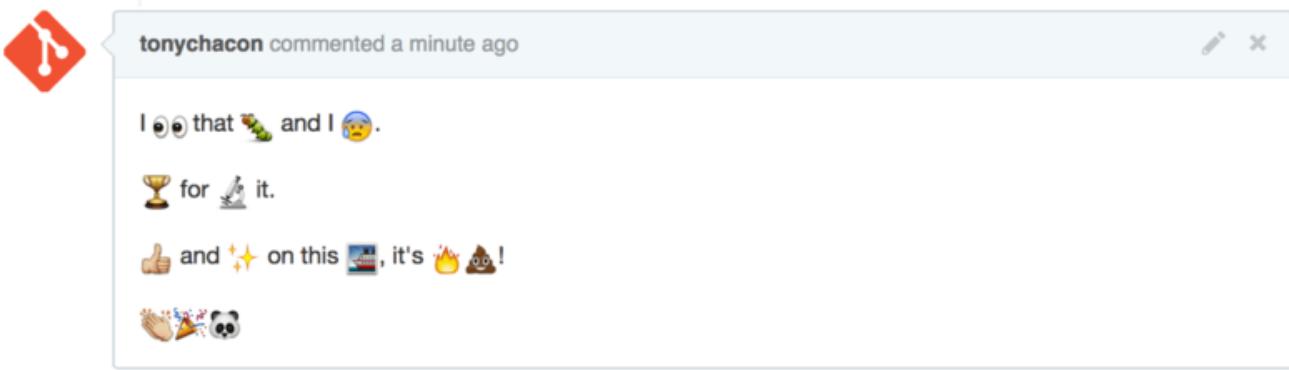


Figure 107. Massive Emoji-Kommentare

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 Sie sagen wollen, finden Sie unter:

<https://www.webfx.com/tools/emoji-cheat-sheet/>

## Bilder

Technisch gesehen ist das keine GitHub-Variante von Markdown, aber es ist unglaublich praktisch. Neben dem Hinzufügen von Markdown-Bildlinks zu Kommentaren, für die es schwierig sein kann, URLs zum Einbetten zu finden, können Sie mit GitHub Bilder per Drag&Drop in Textbereiche ziehen und so einbinden.

The figure consists of two vertically stacked screenshots of the GitHub 'Write' interface.

**Screenshot 1:** Shows a file named 'Git.png' attached to the text 'This is the wrong version of Git for the website:'. A small thumbnail of the image is visible above the file name. Below the text area is a placeholder for attachments: 'Attach images by dragging & dropping or selecting them.'

**Screenshot 2:** Shows the same text and image attachment placeholder as Screenshot 1. Additionally, it includes the Markdown code `![git](https://cloud.githubusercontent.com/assets/7874698/4481741/7b87b8fe-49a2-11e4-817d-8023b752b750.png)` below the text area.

Figure 108. Bilder per Drag&Drop hochladen und automatisch einbetten

Wenn Sie sich [Bilder per Drag&Drop hochladen und automatisch einbetten](#) ansehen, werden Sie einen kleinen Hinweis, „Parsed as Markdown“, über dem Textfeld sehen. Wenn Sie darauf klicken, erhalten Sie einen vollständigen Cheat-Sheet (Spickzettel) mit allem, was Sie mit Markdown auf GitHub machen können.

## Ihr öffentliches GitHub-Repository aktuell halten

Sobald Sie ein GitHub-Repository geforkt haben, existiert Ihr Repository (Ihr „fork“) unabhängig vom Original. Insbesondere dann, wenn das ursprüngliche Repository neue Commits bekommen hat, informiert Sie GitHub in der Regel durch eine Meldung wie:

This branch is 5 commits behind progit:master.

Aber Ihr GitHub-Repository wird nie automatisch von GitHub aktualisiert; das ist etwas, was Sie selbst tun müssen. Glücklicherweise ist das sehr einfach umzusetzen.

Die eine Möglichkeit erfordert keine Konfiguration. Wenn Sie z.B. von <https://github.com/progit/progit2.git> geforkt haben, können Sie Ihren `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 Sie sich in einem anderen Branch befinden, kehren Sie zu `master` zurück.
- ② Holen (engl. `fetch`) Sie sich Änderungen von <https://github.com/progit/progit2.git> und mergen Sie sie in den `master`.
- ③ Pushen Sie Ihren `master` Branch nach `origin`.

Das funktioniert, aber es ist ein lästig, die Fetch-URL jedes Mal neu eingeben zu müssen. Sie können diese Arbeit mit ein wenig Konfiguration automatisieren:

```
$ git remote add progit https://github.com/progit/progit2.git ①
$ git branch --set-upstream-to=progit/master master ②
$ git config --local remote.pushDefault origin ③
```

- ① Fügen Sie das Quell-Repository hinzu und geben Sie ihm einen Namen. Hier habe ich mich entschieden, es `progit` zu nennen.
- ② Konfigurieren Sie Ihren `master` Branch so, dass er von dem `progit` Remote abgeholt (engl. `fetch`) wird.
- ③ Definieren Sie 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 Sie sich in einem anderen Branch befinden, kehren Sie zu `master` zurück.
- ② Fetchen Sie die Änderungen von `progit` und mergen Sie sie in `master`.
- ③ Pushen Sie Ihren `master` Branch nach `origin`.

Dieser Herangehensweise kann nützlich sein, aber sie ist nicht ohne Nachteile. Git wird diese Aufgabe gerne im Hintergrund für Sie erledigen, aber es wird Sie nicht benachrichtigen, wenn Sie einen Commit zum `master` machen, von `progit` pullen und dann zu `origin` pushen – alle diese Operationen sind mit diesem Setup zulässig. Sie müssen 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 Ihres eigenen Projekts.

### Ein neues Repository erstellen

Erstellen wir ein neues Repository, in dem wir unseren Projekt-Code freigeben können. Klicken Sie zunächst auf die Schaltfläche „New repository“ auf der rechten Seite des Dashboards oder auf die Schaltfläche **+** in der oberen Symbolleiste neben Ihrem Benutzernamen, wie in [Das Dropdown-Menü „New repository“](#) zu sehen.

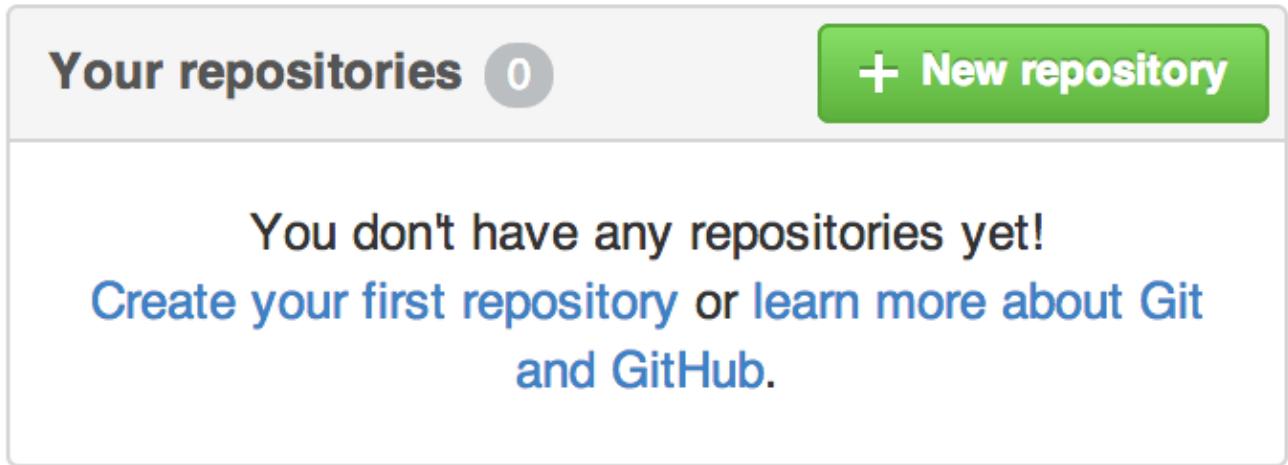


Figure 109. Der Bereich „Your repositories“

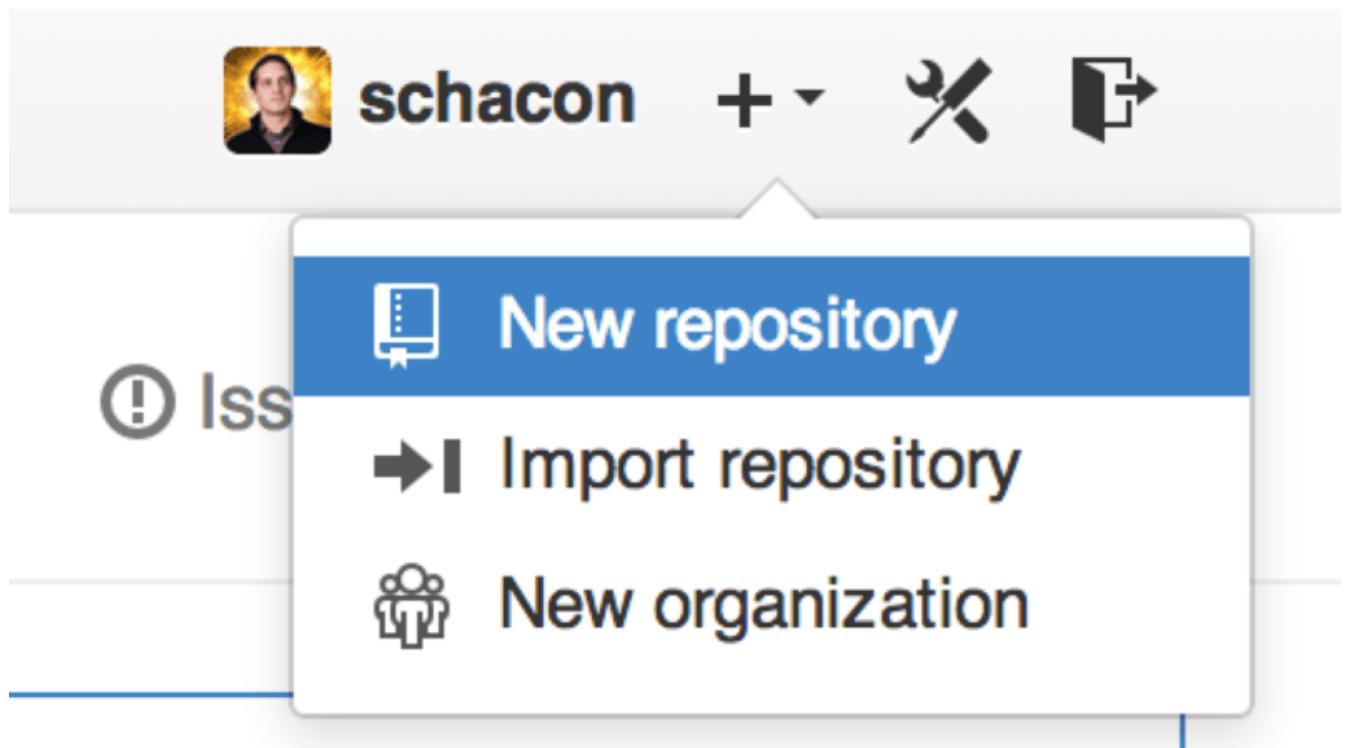


Figure 110. Das Dropdown-Menü „New repository“

Sie werden 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 Sie hier wirklich tun müssen, ist, einen Projektnamen anzugeben; die restlichen Felder sind völlig optional. Fürs Erste klicken Sie einfach auf die Schaltfläche „Create Repository“, und boom – Sie verfügen über ein neues Repository auf GitHub mit dem Namen <User>/<Projekt\_Name>.

Da Sie dort noch keinen Code vorfinden, zeigt Ihnen GitHub Anleitungen an, wie Sie ein brandneues Git-Repository einrichten oder zu einem bestehenden Git-Projekt verbinden können. Wir werden das hier nicht weiter vertiefen; wenn Sie eine Auffrischung benötigen, schauen Sie sich noch einmal [Kapitel 2, Git Grundlagen](#) an.

Da Ihr Projekt jetzt auf GitHub gehostet wird, können Sie die URL an jeden weitergeben, mit dem Sie Ihr Projekt teilen möchten. 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 Anmelddaten des Benutzers, der sich mit ihnen verbündet, 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 Ihr Projekt zugreifen wollen, müssen Sie über ein GitHub-Konto und einen hochgeladenen SSH-Schlüssel verfügen. Die HTTPS-Adresse ist genau die gleiche URL, die sie in einen Browser einfügen würden, um das Projekt dort anzuzeigen.

## Hinzufügen von Mitwirkenden

Wenn Sie mit anderen Personen zusammenarbeiten, denen Sie die Erlaubnis zum Committen gewähren möchten, müssen Sie diese als „collaborators“ (dt. Mitwirkende) eintragen. Ben, Jeff und Louise, Benutzer von GitHub-Konten, denen Sie Push-Zugriff auf Ihr Repository gewähren möchten,

können Sie so zu Ihrem Projekt hinzufügen. Dadurch erhalten sie „Push“-Zugriff, d.h. sie haben sowohl Lese- als auch Schreibzugriff auf das Projekt und das Git-Repository.

Klicken Sie auf den Link „Settings“ unten rechts in der Sidebar.

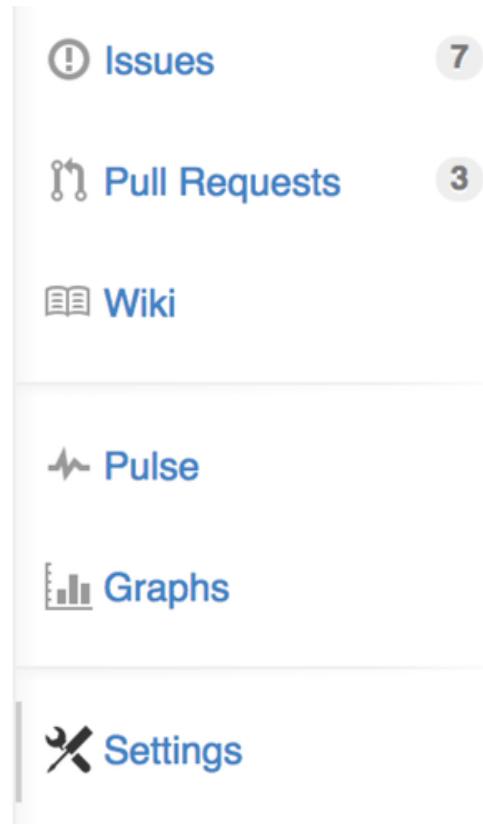


Figure 112. Der Settings-Link des Repositorys

Wählen Sie dann aus dem Menü auf der linken Seite „Collaborators“. Geben Sie dann einfach einen Benutzernamen in das Feld ein und klicken Sie auf „Add Collaborator“. Sie können diese Prozedur beliebig oft wiederholen, um so jedem, den Sie möchten, Zugriff zu gewähren. Wenn Sie die Zugangsberechtigung widerrufen möchten, klicken Sie einfach auf das "X" auf der rechten Seite der entsprechenden Zeile.

A screenshot of the 'Collaborators' section in the GitHub repository settings. 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), Jeff King (peff), and Louise Corrigan (LouiseCorrigan). Each row has a delete icon ('X') on the right. At the bottom, there's a search bar with 'Type a username' and a button 'Add collaborator'.

Figure 113. Mitwirkende im Repository

## Pull-Requests handhaben

Da es jetzt ein Projekt mit einem Code und vielleicht sogar ein paar Mitwirkenden gibt, die auch Push-Zugriff haben, lassen Sie uns noch einmal darüber nachdenken, was zu tun ist, wenn Sie

selbst einen Pull Request erhalten.

Pull-Requests können entweder von einem Branch in einem Fork Ihres Repositories kommen oder von einem anderen Branch im selben Repository. Der einzige Unterschied besteht darin, dass die von einer Fork oft von Personen stammen, die nicht zu ihrem Branch gepusht werden können und sie nicht zu deren, während bei internen Pull-Requests im Allgemeinen beide Parteien Zugriff auf den Branch haben.

Für diese Beispiele nehmen wir an, Sie sind „tonychacon“ und haben ein neues Arduino-Code-Projekt mit der Bezeichnung „fade“ erstellt.

## E-Mail Benachrichtigungen

Jemand meldet sich bei Ihnen, bearbeitet Ihren Code und sendet Ihnen einen Pull-Request. In diesem Fall sollten Sie eine E-Mail erhalten, in der Sie über den neuen Pull-Request informiert werden und dieser sollte etwa so aussehen wie in [E-Mail Benachrichtigung über einen neuen Pull-Request](#).

*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 wieviel. Sie erhalten einen Link zum Pull Request auf GitHub. Sie enthält auch ein paar URLs, die Sie von der Kommandozeile aus verwenden können.

Wenn Sie die Zeile sehen, 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 Sie möchten, können Sie einen Themen-Branch erstellen und in diesen wechseln (engl. checkout) und dann diesen Befehl ausführen, um die Änderungen in dem Pull-Request zu mergen.

Die anderen interessanten URLs sind die `.diff` und `.patch` URLs, die, wie Sie vielleicht vermuten, vereinheitlichte Diff- und Patch-Versionen des Pull Requests bereitstellen. Technisch gesehen könnten Sie 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, können Sie nun eine Diskussion mit der Person führen, die den Pull Request geöffnet hat. Sie können bestimmte Code-Zeilen kommentieren, ganze Commits kommentieren oder den gesamten Pull-Request selbst kommentieren, indem Sie „GitHub Flavored Markdown“ an beliebiger Stelle verwenden.

Jedes Mal, wenn jemand den Pull-Request kommentiert, erhalten Sie weitere E-Mail-Benachrichtigungen, damit Sie wissen, dass Aktivitäten stattfinden. Sie werden jeweils einen Link zum Pull Request enthalten, in dem die Aktivität stattfindet. Auf die E-Mails können Sie auch direkt antworten, um den Pull-Request-Thread zu kommentieren.

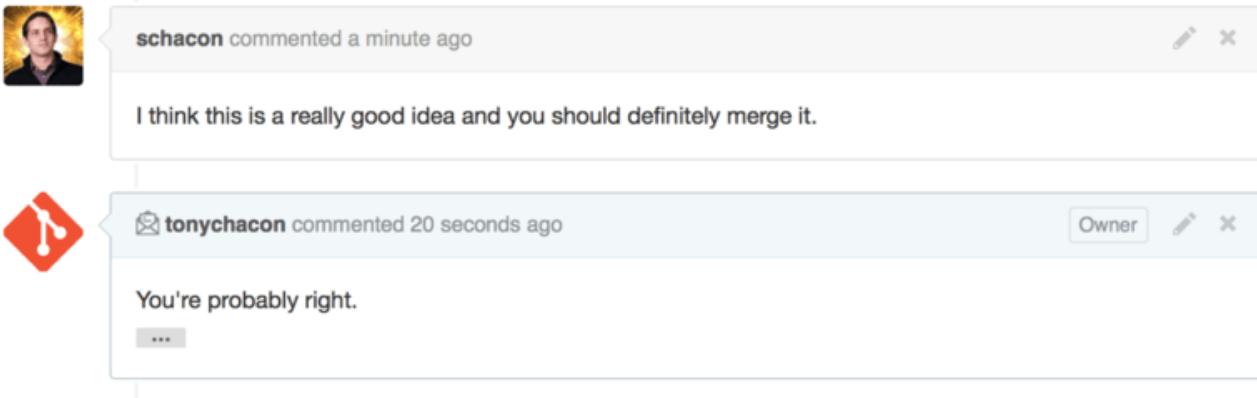


Figure 115. Antworten auf E-Mails sind in den Thread integriert

Sobald der Quellcode richtig platziert ist und Sie ihn zusammenführen möchten, können Sie ihn mit der Anweisung `git pull <url> <branch>` (wie zuvor gesehen) pullen und lokal mergen oder Sie fügen den Fork als Remote hinzu, fetchen den Patch und mergen ihn anschließend.

Wenn das Zusammenführen trivial ist, können Sie 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 Sie tun, jedes Mal, wenn Sie auf die Schaltfläche Merge klicken, ein Merge-Commit erstellt wird. Wie Sie in [Merge-Button und Anweisungen zum manuellen Zusammenführen eines Pull-Requests](#) sehen können, gibt Ihnen GitHub all diese Informationen, wenn Sie auf den Hinweis-Link klicken.

This screenshot shows the 'Merge pull request' section of a GitHub pull request page. It includes a summary message, a command-line merge guide, and step-by-step instructions for manual merging via the command line.

**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 Sie sich entscheiden, dass Sie ihn nicht zusammenführen möchten, können Sie auch einfach den Pull-Request schließen und die Person, die ihn geöffnet hat, wird benachrichtigt.

## Pull Request Refs (Referenzen)

Wenn Sie mit **vielen** Pull-Requests zu kämpfen haben und nicht jedes Mal einen ganzen Batzen

Remotes hinzufügen oder einmalige Pulls durchführen wollen, gibt es einen tollen Kniff, den GitHub Ihnen anbietet. Es handelt sich um einen fortgeschrittenen Kunstgriff, den wir in [Kapitel 10, Refspecs](#) ausführlicher durchgehen werden. Er kann jedoch recht nützlich sein.

GitHub preist die Pull-Request-Banches für ein Repository als eine Art Pseudo-Zweig auf dem Server an. Normalerweise werden sie beim Klonen nicht angezeigt, aber sie sind verdeckt vorhanden und man kann ziemlich leicht darauf zugreifen.

Um das zu verdeutlichen, werden wir den Low-Level-Befehl `ls-remote` verwenden, der oft als „plumbing“ Befehl 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
3c8d735ee16296c242be7a9742ebfbc2665adec1    refs/pull/2/head
15c9f4f80973a2758462ab2066b6ad9fe8dcf03d    refs/pull/2/merge
a5a7751a33b7e86c5e9bb07b26001bb17d775d1a    refs/pull/4/head
31a45fc257e8433c8d8804e3e848cf61c9d3166c    refs/pull/4/merge
```

Wenn Sie sich in Ihrem Repository befinden und `git ls-remote origin` ausführen (oder auch einen beliebigen anderen Remote, den Sie überprüfen möchten), dann wird Ihnen das etwas Vergleichbares angezeigt.

Wenn sich das Repository auf GitHub befindet und Sie irgendwelche geöffnete Pull-Requests haben, 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 Sie klonen oder vom Server fetchen – 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 in der 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` Zweig (weil der in **seinem** Fork liegt). Aber wir *werden* `pull/<pr#>/head` bekommen, der auf `a5a775` zeigt. Das heißt, wir können ziemlich einfach jeden Pull-Request-Branch in einem Rutsch herunterladen, ohne einen Stapel Remotes hinzufügen zu müssen.

Jetzt könnten Sie das Fetchen der Referenz direkt durchführen.

```
$ git fetch origin refs/pull/958/head
From https://github.com/libgit2/libgit2
 * branch                  refs/pull/958/head -> FETCH_HEAD
```

Git erhält die Meldung: „Verbinden Sie sich mit dem `origin`-Remote und laden Sie die Referenz `refs/pull/958/head` herunter.“ Git folgt gern und lädt alles herunter, was Sie brauchen, um diesen Ref zu erstellen. Es setzt einen Zeiger auf den Commit, den Sie unter `.git/FETCH_HEAD` haben wollen. Sie können das mit `git merge FETCH_HEAD` in einen Branch fortsetzen. In diesem wollen Sie testen, aber die Merge-Commit-Nachricht sieht ein wenig merkwürdig aus. Wenn Sie **viele** Pull-Requests überprüfen müssen, wird das umständlich.

Es gibt auch eine Möglichkeit wie Sie *alle* Pull-Requests abrufen und aktuell zu halten können, wann immer Sie sich mit dem Remote verbinden. Öffnen Sie `.git/config` in Ihrem bevorzugten Editor und suchen Sie nach dem `origin` Remote. Es sollte ein bißchen wie folgt 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 den Namen in Ihrem lokalen `.git`-Verzeichnis zuzuordnen. Das sagt Git speziell: „die Sachen auf dem Remote, die unter `refs/heads` liegen, sollten in meinem lokalen Repository unter `refs/remotes/origin` abgelegt werden.“ Sie können 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 meldet Git: „Alle Referenzen, die wie `refs/pull/123/head` aussehen, sollten lokal als `refs/remotes/origin/pr/123` gespeichert werden.“ Wenn Sie jetzt diese Datei speichern und ein `git fetch` auslösen, 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 Sie einen Fetch durchführen. 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 unter Ihnen mit Adleraugen werden das `head` am Ende des Remote-Abschnitts der refspec bemerken. Es gibt auch ein `refs/pull/#/merge` ref auf der GitHub-Seite, der den Commit darstellt, der sich ergeben würde, wenn Sie auf die Schaltfläche „merge“ klicken. Dies kann es Ihnen ermöglichen, das Zusammenführen zu testen, bevor Sie überhaupt auf die Schaltfläche klicken. Auf diese Weise können Sie das Mergen testen, bevor Sie überhaupt auf die Schaltfläche klicken.

## Pull-Requests auf Pull-Requests

Sie können nicht nur Pull-Requests öffnen, die auf den Haupt- oder `master`-Branch gerichtet sind, sondern auch einen Pull-Request, der auf einen beliebigen Branch im Netzwerk ausgerichtet ist. Vielmehr können Sie sogar einen weiteren Pull-Request als Ziel wählen.

Wenn Sie einen Pull-Request sehen, der sich in die richtige Richtung bewegt und Sie eine Idee für eine Änderung haben, die von ihm abhängt; oder Sie sich nicht sicher sind, ob er eine gute Idee ist; oder Sie einfach keinen Push-Zugang zum Zielbranch haben – dann können Sie einen Pull-Request auf diesen direkt öffnen.

Wenn Sie einen Pull-Request öffnen, befindet sich oben auf der Seite ein Feld, das angibt, von welchem Branch Sie pullen und in welchen Sie den Pull-Vorgang ausführen möchten. Wenn Sie auf die Schaltfläche „Edit“ (Bearbeiten), rechts neben diesem Feld, klicken, können Sie nicht nur die Branches, sondern auch den Fork ändern.

The screenshot shows a GitHub pull request comparison interface. At the top, it displays two branches: 'schacon:master' and 'tonychacon:patch-2'. Below this, there's a green button labeled 'Create pull request' and a note: 'Discuss and review the changes in this comparison with others.' To the right is an 'Edit' button. The main area shows a comparison between '2 commits' (from Oct 02, 2014) and '1 file changed'. It lists two commits: one by schacon and one by tonychacon. The commit by schacon is titled 'wait longer to see the dimming effect better' and has a commit hash of 4276e81. The commit by tonychacon is titled 'Update fade.ino' and has a commit hash of c47fc8b. Below this, another pull request interface is shown, featuring a dropdown menu for 'base fork' set to 'schacon/fade', a dropdown for 'base' set to 'patch-1', and a dropdown for 'head fork' set to 'tonychacon/fade'. A modal window titled 'Choose a base branch' is open, showing a dropdown menu with 'master' selected and 'patch-1' as an option. The modal also contains the text 'Branch, tag, commit, or history marker' and a note 'others.'.

Figure 117. Manuelles Ändern der Pull-Request Ziel-Fork und der Branch

Hier können Sie relativ einfach angeben, ob Ihr neuer Branch in einen anderen Pull-Request oder einen anderen Fork des Projekts zusammengeführt werden soll.

## Nennen und Benachrichtigen

GitHub hat auch ein ziemlich praktisches Benachrichtigungssystem integriert, das bei Fragen oder Rückmeldungen von einzelnen Personen oder Teams eine Hilfe sein kann.

In jedem Kommentar können Sie mit der Eingabe eines @-Zeichens beginnen und es wird automatisch mit den Namen und Benutzernamen von Personen vervollständigt, die Mitarbeiter oder Beitragende für das Projekt sind.

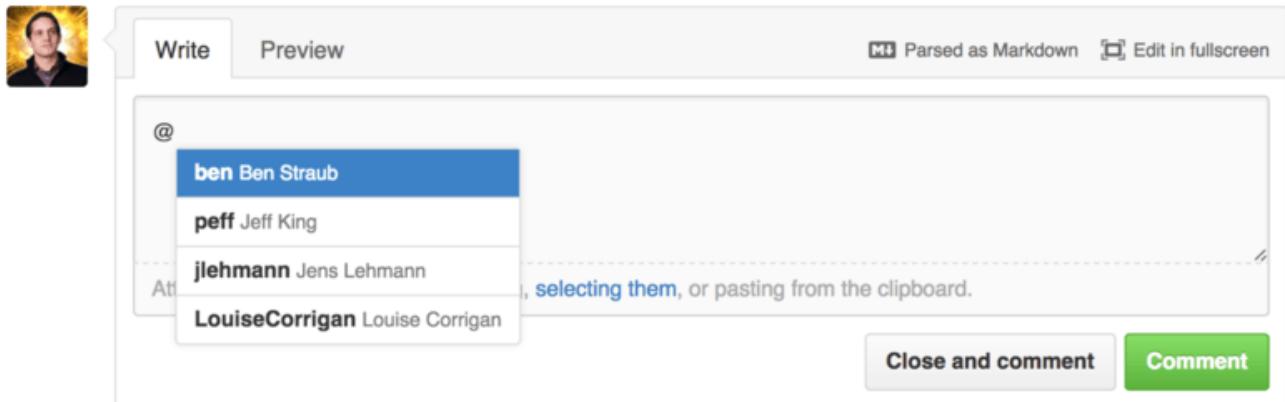


Figure 118. Mit der Eingabe von @ anfangen, um jemanden zu nennen.

Sie können auch einen Benutzer angeben, der sich nicht in diesem Dropdown-Menü befindet, aber oft ist der Auto-Komplettierer schneller.

Sobald Sie einen Kommentar mit einer Benutzererwähnung posten, wird dieser Benutzer benachrichtigt. Damit ist es möglich, Menschen wirklich effektiv in Gespräche zu verwickeln, anstatt sie zur Teilnahme zu drängen. 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. Sie werden auch subskribiert, wenn er von Ihnen geöffnet wurde, wenn Sie das Repository beobachten oder wenn Sie etwas kommentieren. Wenn Sie keine weiteren Benachrichtigungen mehr erhalten möchten, können Sie auf die Schaltfläche „Unsubscribe“ klicken, um sich 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 Sie zu erreichen, falls ein Ereigniss eintritt. Es gibt ein paar Einstellungen, die Sie konfigurieren können. Wenn Sie von der Settings-Seite aus auf die Registerkarte "Notifications" gehen, sehen Sie einige der verfügbaren Optionen.

The screenshot shows the GitHub user profile for tonychacon. On the left, a sidebar lists various account settings: Profile, Account settings, Emails, Notification center (which is selected), Billing, SSH keys, Security, Applications, Repositories, and Organizations. The main content area is titled "How you receive notifications". It contains two sections: "Participating" and "Watching". Under "Participating", it says: "When you participate in a conversation or someone brings you in with an @mention." with checkboxes for "Email" and "Web", both of which are checked. Under "Watching", it says: "Updates to any repositories or threads you're watching." with checkboxes for "Email" and "Web", both of which are checked. Below this, there's a section titled "Notification email" with a "Primary email address" field containing "tchacon@example.com" and a "Save" button. At the bottom, there's a "Custom routing" section with the note: "You can send notifications to different **verified** email addresses depending on the organization that owns the repository."

Figure 120. Benachrichtigungs-Optionen

Die beiden Möglichkeiten sind: über „E-Mail“ oder „Web“ benachrichtigen. Sie können entweder eine, keine oder beide Optionen wählen, wenn Sie sich an den Aktivitäten in Repositoryn beteiligen, die Sie gerade beobachten.

## Web Benachrichtigungen

Web-Benachrichtigungen gibt es nur auf GitHub und Sie können sie nur auf GitHub überprüfen. Wenn Sie diese Option in Ihren Einstellungen ausgewählt haben und eine Benachrichtigung für Sie ausgelöst wird, sehen Sie oben auf Ihrem Bildschirm einen kleinen blauen Punkt über Ihrem Benachrichtigungssymbol, wie in [Benachrichtigungen](#) zu sehen ist.

The screenshot shows the GitHub notifications interface. At the top, there's a search bar, navigation links for Explore, Gist, Blog, Help, and a user profile for tonychacon. A notification badge indicates 'You have unread notifications' with a 'Mark all as read' button. Below the header, there are three main sections:

- Unread:** Shows 4 notifications. One from 'mycorp/project1' about 'SF Corporate Housing Search' posted an hour ago, marked as read. Another from 'git/git-scm.com' about the 'Front Page' posted 3 hours ago, also marked as read.
- Participating:** Shows 3 notifications. One from 'schacon/blink' about 'To Be or Not To Be' posted 5 days ago, and another about 'Three seconds is better' also posted 5 days ago, both marked as read.
- All notifications:** Shows 1 notification from 'mycorp/project1' about the 'Front Page' posted 5 days ago, marked as read.

Figure 121. Benachrichtigungen

Wenn Sie darauf klicken, sehen Sie eine Liste aller Elemente, über die Sie informiert wurden, gruppiert nach Projekten. Sie können nach den Benachrichtigungen eines bestimmten Projekts filtern, indem Sie auf dessen Namen in der linken Seitenleiste klicken. Möglich ist auch die Übernahme der Benachrichtigung durch Anklicken des Häkchens neben einer Meldung oder die Übernahme *aller* Benachrichtigungen in einem Projekt durch Anklicken des Häkchens oben in der Gruppe. Es gibt auch eine Mute-Taste neben jedem Häkchen, die Sie anklicken können, um keine weiteren Mitteilungen zu diesem Thema zu erhalten.

Diese Tools sind alle sehr praktisch für die Bearbeitung einer großen Anzahl von Benachrichtigungen. Viele GitHub Power-User schalten E-Mail-Benachrichtigungen einfach komplett aus und verwalten ihre gesamten Benachrichtigungen über diese Seite.

## E-Mail Benachrichtigungen

E-Mail-Benachrichtigungen sind die zweite Variante, mit der Sie Benachrichtigungen über GitHub auswerten können. Wenn Sie diese Option aktiviert haben, erhalten Sie 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 richtig nach Thema sortiert, was sehr hilfreich ist, wenn Ihr E-Mail-Client entsprechend konfiguriert ist.

In den Headern der E-Mails, die GitHub Ihnen 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 zwischen 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. Möchten Sie E-Mails zu diesem speziellen Projekt oder sogar Pull Request hervorheben oder umleiten, erhalten Sie mit den Informationen in [Message-ID](#) alle Daten 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 Sie, wenn Sie einen Mail-Client haben, der das versteht, ganz einfach in die Liste posten oder sich vom Thread „abmelden“ (engl. unsubscribe) können. 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 Sie sowohl E-Mail- als auch Web-Benachrichtigungen aktiviert haben und Sie die E-Mail-Version der Benachrichtigung lesen, die Web-Version auch als gelesen markiert wird, falls Sie in Ihrem Mail-Client Bilder erlaubt haben.

## Besondere Dateien

Es gibt ein paar besondere Dateien, die GitHub erkennt, wenn sie in Ihrem 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 Ihrem Quellcode 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 rendert, können Sie 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 Sie eine **CONTRIBUTING** Datei mit einer beliebigen Dateiendung verwenden, zeigt GitHub wie in [Einen Pull-Request starten, wenn eine CONTRIBUTING-Datei existiert](#) an, wenn irgend jemand einen Pull-Request öffnet.

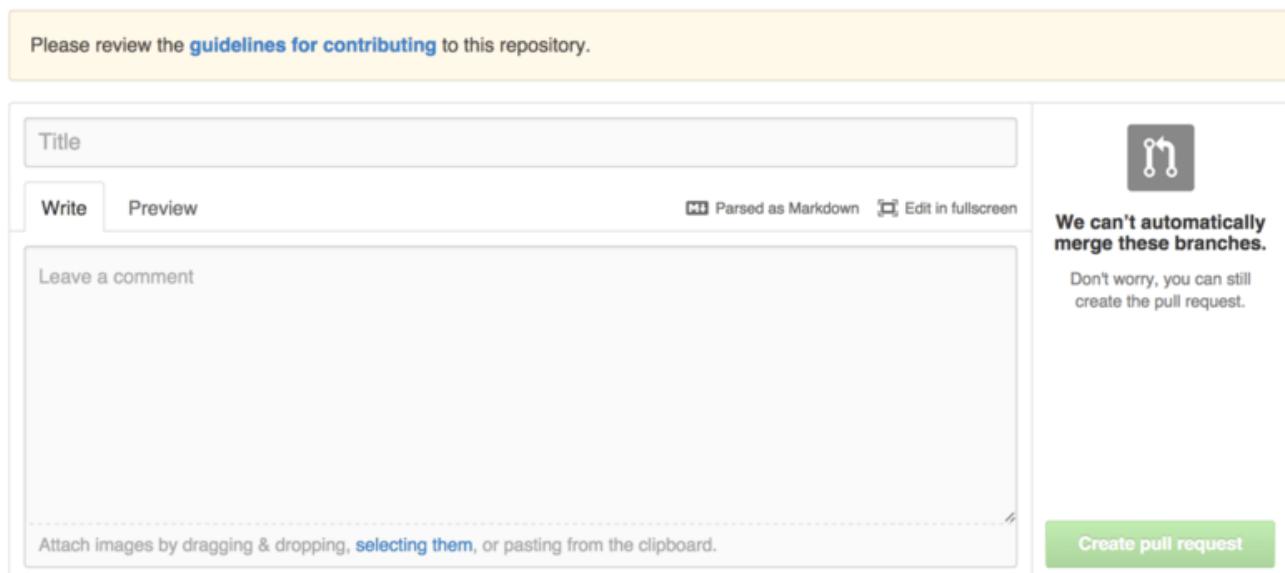


Figure 122. Einen Pull-Request starten, wenn eine CONTRIBUTING-Datei existiert

Die Absicht dabei ist, dass Sie bestimmte Punkte spezifizieren können, die Sie benötigen oder nicht wünschen, für Pull-Requests, die an Ihr Projekt gesendet werden. Auf diese Weise kann ein Benutzer die Leitlinien auch wirklich lesen, bevor er den Pull-Request öffnet.

## Projekt-Administration

Generell gibt es nur wenige administrative Aufgaben, die Sie mit einem einzelnen Projekt durchführen können, aber ein paar Punkte könnten interessant sein.

### Ändern der Standard-Branch

Wenn Sie einen anderen Branch statt „master“ als Standard-Zweig verwenden wollen, auf den die Teilnehmer Pull-Requests öffnen oder ihn standardmäßig sehen sollen, dann können Sie das auf der Settings-Seite Ihres Repositorys unter der Registerkarte „Optionen“ ändern.

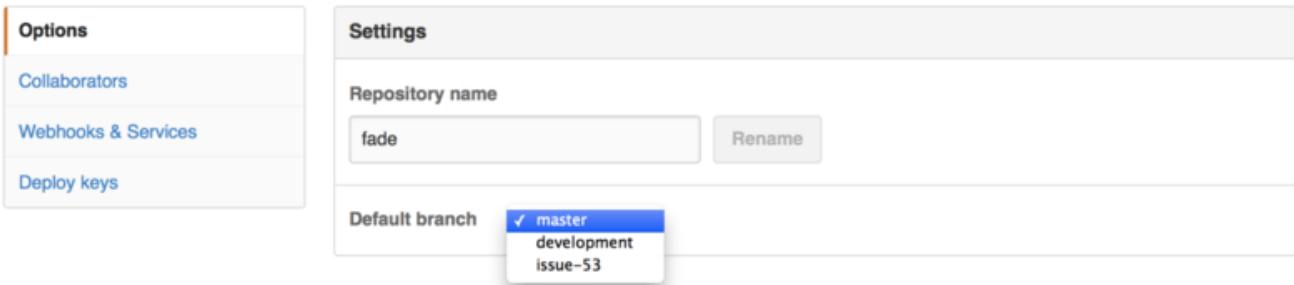


Figure 123. Change the default branch for a project.

Ändern Sie 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 Sie ein Projekt auf einen anderen Benutzer oder eine Organisation in GitHub übertragen möchten, gibt es unten auf der gleichen Registerkarte „Optionen“ Ihrer Repository-Einstellungen eine Option „Eigenum übertragen“ (engl. Transfer ownership), die das ermöglicht.

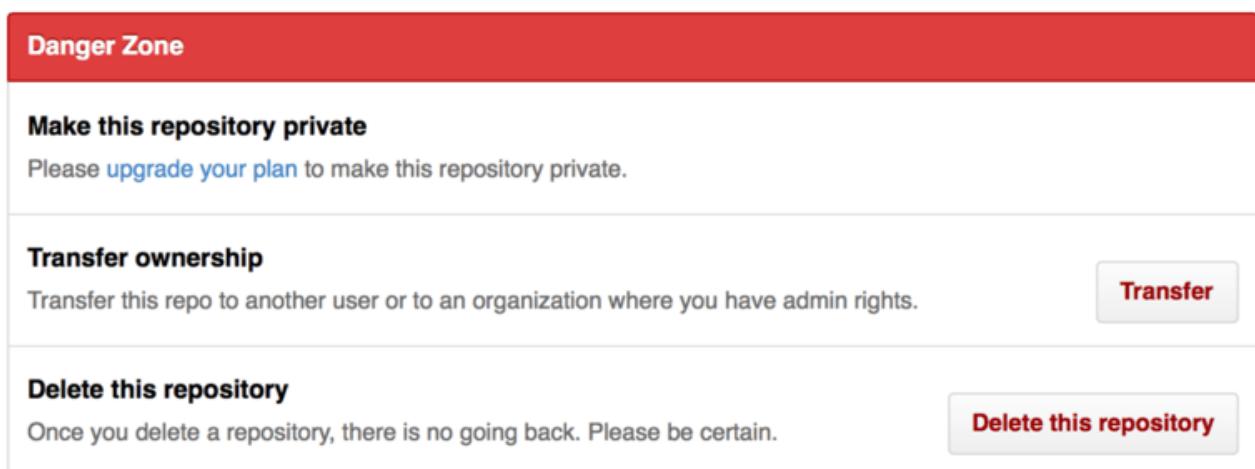


Figure 124. Übertragen eines Projekts auf einen anderen GitHub-User oder eine andere Organisation

Diese Option ist sinnvoll, wenn Sie ein Projekt aufgeben und es von jemandem übernommen werden soll oder wenn Ihr Projekt größer wird und Sie es in eine Organisation verlagern möchten.

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 Ihrer URL an den neuen Ort eingerichtet. Es wird auch die Klonen 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 verschieden. 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. Klicken Sie einfach auf das „+“ Symbol oben rechts auf jeder GitHub-Seite und wählen Sie „Neue Organisation“ aus dem Menü.

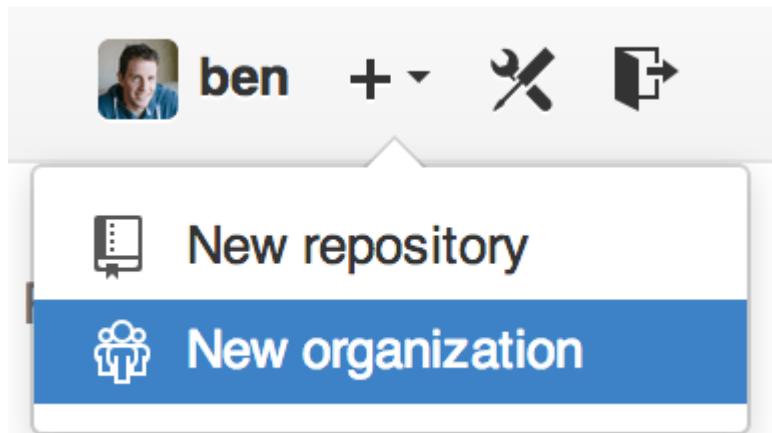


Figure 125. Der Menüpunkt „Neue Organisation“

Zuerst müssen Sie Ihrer Organisation einen Namen geben und eine E-Mail-Adresse für den Hauptansprechpartner der Gruppe angeben. Dann können Sie andere Benutzer einladen, Miteigentümer des Accounts zu werden.

Befolgen Sie diese Anweisungen und Sie werden bald Eigentümer einer brandneuen Organisation sein. Wie persönliche Konten sind Unternehmen kostenlos, wenn alles, was Sie dort ablegen wollen, Open Source sein wird.

Als Eigentümer in einer Organisation haben Sie beim Forken eines Repository die Wahl, es in den Namensraum Ihrer Organisation zu übertragen. Wenn Sie neue Repositories erstellen, können Sie diese entweder unter Ihrem persönlichen Konto oder unter dem einer der Organisationen erstellen, deren Sie Eigentümer sind. Sie „beobachten“ (engl. watch) auch automatisch jedes neue Repository, das unter diesen Unternehmen erstellt wird.

Wie in [Ihr Avatar-Bild](#) gezeigt, können Sie ein Symbol-Bild für Ihre Organisation hochladen, um sie ein wenig zu personalisieren. Wie bei persönlichen Konten haben Sie auch eine Startseite für die Organisation, die alle Ihre Repositorys auflistet und von anderen eingesehen werden kann.

Lassen Sie 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, Ihr Unternehmen verfügt über drei Repositories: [frontend](#), [backend](#), und [deployscripts](#). Sie möchten, dass Ihre HTML/CSS/JavaScript-Entwickler Zugriff auf das Frontend

und eventuell das Backend haben und Ihre 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 Ihnen ein übersichtliches Dashboard mit allen Repositories, Benutzern und Teams, die zu dieser Organisation gehören.

The screenshot shows the GitHub organization page for 'chaconcorp'. At the top, there's a purple logo and the organization name 'chaconcorp'. Below the header, there's a search bar with 'Filters' and a 'Find a repository...' placeholder, and a green '+ New repository' button. The main content area displays three repositories: 'deployscripts' (scripts for deployment, updated 16 hours ago), 'backend' (Backend Code, updated 16 hours ago), and 'frontend' (Frontend Code, updated 16 hours ago). To the right, there are two sections: 'People' (listing 'dragonchacon' (Dragon Chacon), 'schacon' (Scott Chacon), and 'tonychacon' (Tony Chacon) with an 'Invite someone' button) and 'Teams' (listing 'Owners' (1 member - 3 repositories), 'Frontend Developers' (2 members - 2 repositories), and 'Ops' (3 members - 1 repository) with a 'Create new team' button).

Figure 126. Die Seite Organisation

Um Ihre Teams zu verwalten, können Sie in [Die Seite Organisation](#) auf die Team-Seitenleiste auf der rechten Seite klicken. So gelangen Sie zu der Seite, auf der Sie Mitglieder zum Team hinzufügen, Repositorys zum Team hinzufügen oder die Einstellungen und Zugriffskontrollstufen für das Team verwalten können. Jedes Team kann Lesezugriff, Lese-/Schreibzugriff oder administrativen Zugriff auf die Repositorys haben. Sie können die Stufe ändern, indem Sie auf die Schaltfläche „Einstellungen“ in [Die Seite Team](#) klicken.

The screenshot shows the GitHub Team page for 'Frontend Developers'. At the top, there's a navigation bar with icons for People (3), Teams (3), and Audit log. On the left, a sidebar for the team 'Frontend Developers' displays 2 members and 2 repositories, with buttons for Leave and Settings. The main area shows two team members: Tony Chacon (tonychacon) and Scott Chacon (schacon). Each member has a profile picture, name, and a 'Remove' button. A button at the top right says 'Invite or add users to team'.

Figure 127. Die Seite Team

Wenn Sie einen Benutzer in ein Team einladen, 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 Sie die Unterstützung von einem Teammitglied wünschen, aber Sie nicht genau wissen, wen Sie fragen sollen.

Ein Benutzer kann zu einer beliebigen Anzahl von Teams gehören, also beschränken Sie sich nicht nur auf Zugriffskontroll-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 ging. Sie können 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                                       |           | member 33 minutes ago            |
| tonychacon    | invited <a href="#">dragonchacon</a> to the <a href="#">chaconcorp</a> organization               |           | member 16 hours ago              |
| tonychacon    | invited <a href="#">schacon</a> to the <a href="#">chaconcorp</a> organization                    |           | org.invite_member 16 hours ago   |
| tonychacon    | gave <a href="#">chaconcorp/ops</a> access to <a href="#">chaconcorp/backend</a>                  |           | team.add_repository 16 hours ago |
| tonychacon    | gave <a href="#">chaconcorp/frontend-developers</a> access to <a href="#">chaconcorp/backend</a>  |           | team.add_repository 16 hours ago |
| tonychacon    | gave <a href="#">chaconcorp/frontend-developers</a> access to <a href="#">chaconcorp/frontend</a> |           | team.add_repository 16 hours ago |
| tonychacon    | created the repository <a href="#">chaconcorp/deployscripts</a>                                   |           | repo.create 16 hours ago         |
| tonychacon    | created the repository <a href="#">chaconcorp/backend</a>   |           | repo.create 16 hours ago         |

Figure 128. Das Audit-Log

Sie können 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 zu manipulieren. In diesem Abschnitt erfahren Sie, wie Sie das GitHub-Hook-System und seine API verwenden, 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 finden Sie im Abschnitt Einstellungen Ihres Repositorys, wo wir uns zuvor mit dem Hinzufügen von Mitwirkenden und dem Ändern der Standard-Branch Ihres Projekts beschäftigt haben. Unter der Registerkarte „Webhooks und Dienste“ sehen Sie so etwas wie [Konfiguration von Diensten und Hooks](#).

The screenshot shows the 'Services' configuration page in GitHub. On the left, a sidebar menu includes 'Options', 'Collaborators', **'Webhooks & Services'** (which is selected), and 'Deploy keys'. The main content area has two tabs: 'Webhooks' and 'Services'. The 'Webhooks' tab contains a brief description of what Webhooks are and a link to the 'Webhooks Guide'. The 'Services' tab contains a similar description and a link to the 'Service Hooks Guide'. Below these descriptions is a search bar with the word 'email' typed into it. A blue button labeled 'Email' is highlighted at the bottom of the search results. A small 'Available Services' dropdown menu is visible above the search bar.

Figure 129. Konfiguration von Diensten und Hooks

Es gibt Dutzende von Diensten, aus denen Sie wählen können, 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 Sie „E-Mail“ aus der Auswahlliste „Add Service“ wählen, erhalten Sie einen Konfigurationsbildschirm wie [E-Mail-Service-Konfiguration](#).

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 Sie verwenden und das Sie mit GitHub integrieren möchten, sollten Sie hier überprüfen, ob es eine bestehende Service-Integration gibt. Angenommen, Sie verwenden Jenkins, um auf Ihrer Code-Basis Tests durchzuführen, können Sie die eingebaute Service-Integration von Jenkins aktivieren, um jedes Mal einen Testlauf zu starten, wenn jemand in Ihr Repository pusht.

## Hooks

Wenn Sie eine speziellere Lösung benötigen oder mit einem Dienst oder einer Website integrieren möchten, der nicht in dieser Liste enthalten ist, können Sie stattdessen das generischere Hooks-System verwenden. GitHub Repository-Hooks sind denkbar einfach. Geben Sie eine URL an und GitHub wird bei jedem gewünschten Event über HTTP Nutz-Daten an diese URL senden.

Im Regelfall können Sie einen kleinen Webservice einrichten, um nach einer GitHub-Hook-Nutzlast (engl. payload) zu suchen und dann die empfangenen Daten weiter zu verarbeiten.

Um einen Hook zu aktivieren, klicken Sie in [Konfiguration von Diensten und Hooks](#) auf die Schaltfläche „Webhook hinzufügen“. Das führt Sie zu einer Seite, die wie [Web-Hook Konfiguration](#) aussieht.

Options

Collaborators

**Webhooks & Services**

Deploy keys

Webhooks / Add webhook

We'll send a POST request to the URL below with details of any subscribed events. You can also specify which data format you'd like to receive (JSON, x-www-form-urlencoded, etc). More information can be found in [our developer documentation](#).

**Payload URL \***

**Content type**

**Secret**

**Which events would you like to trigger this webhook?**

Just the push event.

Send me **everything**.

Let me select individual events.

Active  
We will deliver event details when this hook is triggered.

**Add webhook**

Figure 131. Web-Hook Konfiguration

Die Konfiguration für einen Web-Hook ist relativ einfach. In den meisten Fällen geben Sie einfach eine URL und einen geheimen Schlüssel ein und klicken auf „Webhook hinzufügen“. Es gibt ein paar Optionen, bei denen GitHub veranlasst wird Ihnen eine Payload zu senden – die Vorgabe ist, eine Payload nur für das **push** Ereignis senden, wenn jemand neuen Code in einen beliebigen Branch Ihres Repositorys schiebt.

Schauen wir uns ein kleines Beispiel für einen Webservice an, den Sie für die Verwaltung eines Web-Hooks einrichten können. Wir verwenden das Ruby Web-Framework Sinatra, da es relativ übersichtlich ist und Sie leicht sehen können sollten, 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 die JSON-„Nutzlast“, die 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, haben Sie eine ansprechende Entwicklerkonsole auf dem gleichen Bildschirm, auf dem Sie den Hook eingerichtet haben. Sie können die jüngsten Aktualisierungen sehen, die GitHub für diesen Webhook vorgenommen hat. Für jeden Hook können Sie 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 Ihrer 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 Sie jede der Payloads neu ausliefern können, um Ihren Service einfach zu testen.

Weitere Informationen wie man Webhook schreiben kann und welche Event-Typen man überwachen kann, finden Sie in der GitHub-Developer-Dokumentation unter <https://developer.github.com/webhooks/>.

## Die GitHub API

Dienste und Hooks bieten Ihnen die Möglichkeit, Push-Benachrichtigungen über Ereignisse zu erhalten, die in Ihren Repositories stattfinden, aber was ist, wenn Sie weitere Informationen über diese Ereignisse benötigen? Was ist, wenn Sie eine Automatisierung benötigen, wie z.B. das

## Hinzufügen von Mitwirkenden oder das Markieren von Problemen?

Hier kommt die GitHub API zum Zug. GitHub verfügt über eine Vielzahl von API-Endpunkten, um fast alles zu tun, was Sie auf der Website automatisiert tun können. 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 Sie lösen können, ist eine einfache GET-Anfrage an einen Endpunkt, der keine Authentifizierung erfordert. Das kann ein Benutzer oder schreibgeschützte Informationen zu 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 Sie öffentlich auf GitHub sehen können. Sie können 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 Sie jedoch eine Aktivität auf der Website durchführen möchten, wie z.B. einen Kommentar zu einem Issue oder Pull Request oder wenn Sie private Inhalte einsehen oder mit diesen interagieren möchten, müssen Sie sich authentifizieren.

Es gibt mehrere Möglichkeiten, sich zu authentifizieren. Sie können die Basisauthentifizierung nur mit Ihrem Benutzernamen und Passwort verwenden, aber im Allgemeinen ist es eine bessere Idee, einen persönlichen Zugriffstoken zu verwenden. Sie können den über die Registerkarte „Anwendungen“ auf Ihrer Einstellungsseite generieren.

The screenshot shows the GitHub user settings interface under the 'Applications' tab. On the left sidebar, 'Applications' is highlighted. The main content area has several sections:

- Developer applications**: A section with a 'Register new application' button.
- Personal access tokens**: A section with a 'Generate new token' button. It includes a note about using tokens for scripts or testing and a link to generate a personal access token.
- Authorized applications**: A section stating 'You have no applications authorized to access your account.'
- GitHub applications**: A section listing 'GitHub Team' with details: 'Last used on Oct 6, 2014' and a 'Revoke' button.

Figure 133. Generieren eines Zugriffstokens auf der Registerkarte „Anwendungen“ der Settings-Seite

Sie werden gefragt, welchen Geltungsbereich Sie für dieses Token möchten und es wird eine Beschreibung angezeigt. Achten Sie darauf, eine gute Beschreibung zu verwenden, damit Sie sich sicher sind, das Token entfernen zu können, wenn Ihr Skript oder Ihre Anwendung nicht mehr verwendet wird.

GitHub zeigt Ihnen den Token nur ein einziges Mal an, also kopieren Sie ihn unbedingt. Sie können diese Funktion nun verwenden, um sich in Ihrem Skript zu authentifizieren, anstatt einen Benutzernamen und ein Passwort zu verwenden. Das ist angenehm, weil Sie den Umfang dessen, was Sie tun möchten, einschränken können und das Token widerruflich ist.

Das hat auch den Vorteil, dass die Rate erhöht wird. Ohne Authentifizierung sind Sie auf 60 Anfragen pro Stunde beschränkt. Wenn Sie sich authentifizieren, können Sie 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 Sie jetzt zu diesem Issue gehen, können Sie den Kommentar sehen, den wir gerade erfolgreich gepostet haben, wie in [Kommentar, veröffentlicht von der GitHub API](#) zu sehen ist.

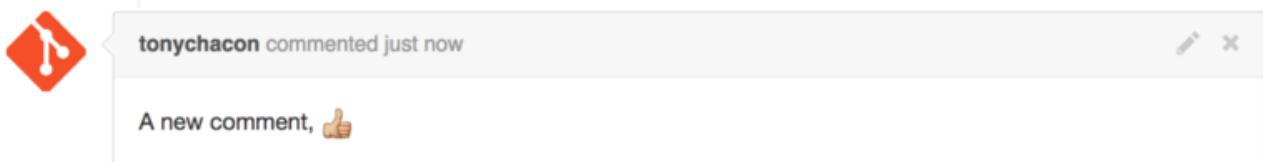


Figure 134. Kommentar, veröffentlicht von der GitHub API

Sie können die API verwenden, um so ziemlich alles zu tun, was Sie auf der Website tun können – das Erstellen und Setzen von Meilensteinen, das Zuweisen von Personen zu Issues und Pull-Requests, das Erstellen und Ändern von Labels, den Zugriff auf Commit-Daten, 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 Sie mit Pull-Requests arbeiten. 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 verschoben wurde, und dann Bericht erstatten, wenn dieser Commit alle Tests bestanden hat. Sie können damit auch überprüfen, ob die Commit-Nachricht korrekt formatiert ist, ob der Einreicher alle Ihre Contributions-Richtlinien befolgt hat, ob die Übertragung gültig signiert wurde – und vieles mehr.

Angenommen, Sie richten einen Webhook in Ihrem 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 an den [`/repos/<user>/<repo>/statuses/<commit\_sha>`](#) API-Endpunkt mit dem Status.

In diesem Fall können Sie 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, sehen Sie vielleicht etwas wie [Commit-Status via API](#).

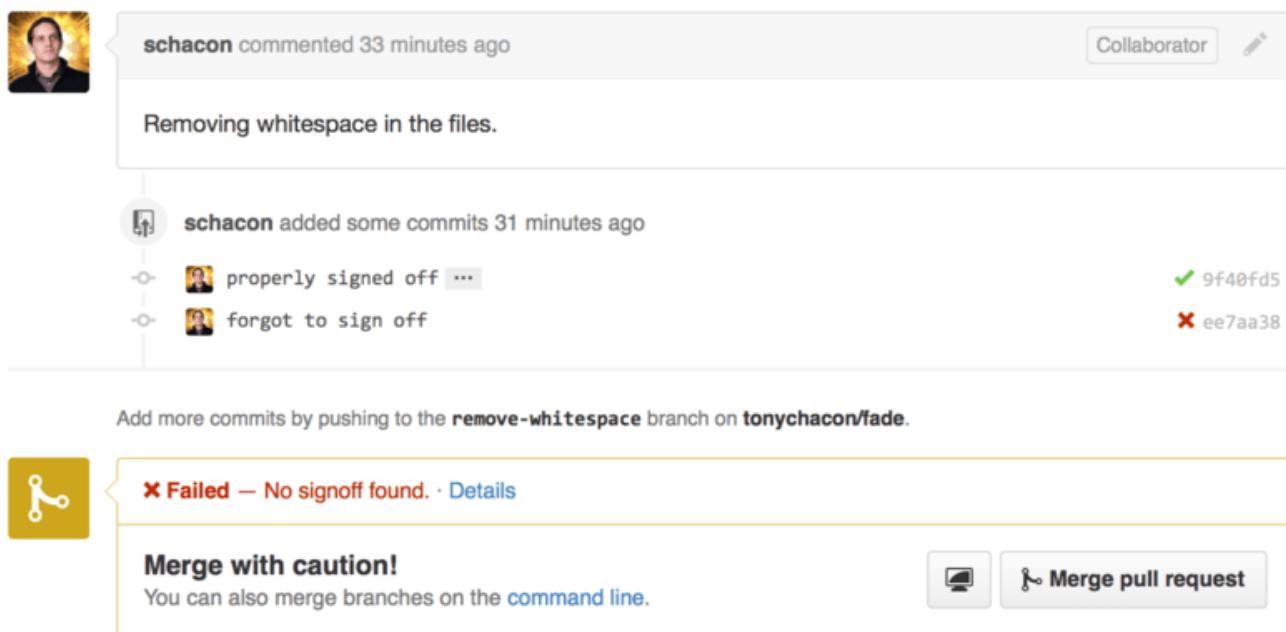


Figure 135. Commit-Status via API

Sie sehen nun ein kleines grünes Häkchen neben dem Commit, das in der Nachricht eine Zeichenkette „Signed-off-by“ und ein rotes Kreuz dasjenige enthält, bei dem der Autor vergessen hat, sich abzumelden. Sie können auch sehen, dass der Pull-Request den Status des letzten Commits auf dem Branch annimmt und Sie warnt, falls es ein Fehler ist. Das ist besonders nützlich, wenn Sie diese API für Prüfergebnisse verwenden, damit Sie nicht versehentlich etwas zusammenführen, 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. Besuchen Sie [Octokit](#) für weitere Informationen zu diesen Themen, da sie einen großen Teil des HTTP-Protokolls für Sie verarbeiten.

Hoffentlich können diese Tools Ihnen helfen, GitHub anzupassen und zu modifizieren, um so besser zu Ihren individuellen Workflows zu passen. Eine vollständige Dokumentation der gesamten API sowie Anleitungen für häufige Aufgaben finden Sie unter <https://developer.github.com>.

## Zusammenfassung

Sie sind nun ein GitHub-User. Sie wissen, wie man ein Konto einrichtet, ein Unternehmen verwaltet, Repositories erstellt und betreibt, zu Projekten anderer beiträgt und Beiträge von anderen entgegennimmt. Im nächsten Kapitel lernen Sie weitere mächtige Werkzeuge und Tipps,

für den Umgang mit komplexen Situationen, kennen, die Sie zu einem kompetenten Experten für Git machen werden.

# Git Tools

Inzwischen haben Sie die meisten der gängigen Befehle und Workflows kennen gelernt. Sie benötigen sie, um ein Git-Repository für Ihre Quellcode-Kontrolle zu verwalten oder zu pflegen. Sie haben die grundlegenden Aufgaben des Tracking und Commitens von Dateien gelöst und Sie haben die Leistungsfähigkeit der Staging Area und der einfachen Branching- und Merging-Funktionen von Topics-Banches genutzt.

Jetzt werden Sie eine Reihe von sehr nützlichen Anwendungen entdecken, die Git ausführen kann. Sie werden 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 zwangsläufig offensichtlich, aber es ist nützlich sie zu kennen.

### Einzelne Revisionsstände

Sie können sich 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 Sie auf jeden Commit verweisen können.

#### Kurz-SHA-1

Git ist intelligent genug, um herauszufinden, auf welchen Commit Sie sich beziehen, wenn Sie die ersten paar Zeichen des SHA-1-Hash angeben, 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 Sie z.B. einen bestimmten Commit untersuchen möchten, von dem Sie wissen, dass Sie gewisse Funktionen hinzugefügt haben, könnten Sie zuerst 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

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    added some blame and merge stuff
```

Angenommen, Sie sind an dem Commit interessiert, dessen Hash mit `1c002dd...` beginnt. Sie können den Commit mit einer der folgenden Varianten von `git show` überprüfen (vorausgesetzt, die verkürzten Versionen sind eindeutig):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git kann eine eindeutige Abkürzung für Ihre SHA-1-Werte ermitteln. Wenn Sie `--abbrev-commit` an den Befehl `git log` übergeben, verwendet die Ausgabe kürzere Werte, aber sie bleiben eindeutig. Es werden standardmäßig sieben Zeichen verwendet, aber bei Bedarf werden sie verlängert, um den SHA-1 eindeutig zu halten:

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

In der Regel sind acht bis zehn Zeichen mehr als genug, um innerhalb eines Projekts unverwechselbar 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 Sie ein Objekt, das auf den gleichen SHA-1-Wert wie ein vorhergehendes *unterschiedliches* Objekt in Ihrem Repository hasht, committen, wird Git das vorhergehende Objekt bereits in Ihrer Git-Datenbank sehen und davon ausgehen, dass es bereits geschrieben wurde und es einfach wiederverwenden. Wenn Sie versuchen, dieses Objekt irgendwann wieder auszuchecken, erhalten Sie immer die Daten des ersten Objekts.

Sie sollten sich jedoch bewusst sein, wie lächerlich unwahrscheinlich dieses Szenario ist. Der SHA-1 Hashwert beträgt 20 Bytes oder 160 Bit. Die Anzahl der zufällig gehashten 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 Ihnen 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 schieben wollte, dann würde es etwa 2 Jahre dauern, bis dieses Repository genügend Objekte enthielt, um eine 50%ige Wahrscheinlichkeit einer einzelnen SHA-1-Objektkollision zu erzielen. Somit ist eine SHA-1-Kollision unwahrscheinlicher, als wenn jedes Mitglied Ihres Programmierer-Teams in der gleichen Nacht von Wölfen angegriffen und bei unabhängigen Zwischenfällen getötet würde.



## Branch Referenzen

Eine unkomplizierte Methode, auf einen bestimmten Commit zu verweisen, ist, wenn es sich um den Commit an der Spitze von einem Branch handelt. In diesem Fall können Sie einfach den Branch-Namen in jedem Git-Befehl verwenden, der eine Referenz auf einen Commit erwartet. Wenn Sie beispielsweise das letzte Commit-Objekt in einem Branch untersuchen möchten, sind die folgenden Befehle gleichwertig, vorausgesetzt, der Branch `topic1` zeigt auf den Commit `ca82a6d....`:

```
$ git show ca82a6dff817ec66f44342007202690a93763949  
$ git show topic1
```

Wenn Sie sehen wollen, auf welchen spezifischen SHA-1 einen Branch zeigt, oder wenn Sie sehen wollen, worauf sich eines dieser Beispiele in Bezug auf SHA-1s verkürzt, können Sie ein Git Basis-Befehl-Tool (engl. plumbing tool) mit dem Namen `rev-parse` verwenden. Sie können in [Git Interna](#) weitere Details über Basisbefehl-Tools nachlesen. Im Grunde genommen gibt es `rev-parse` für Low-Level-Operationen und ist nicht für den Einsatz im täglichen Betrieb konzipiert. Allerdings kann es

gelegentlich hilfreich sein, wenn man herausfinden muss, was eigentlich passiert. So können Sie `rev-parse` auf Ihrem Branch ausführen.

```
$ git rev-parse topic1  
ca82a6dff817ec66f44342007202690a93763949
```

## RefLog Kurzformen

Eine der Dinge, die Git im Hintergrund macht, während Sie arbeiten, ist, einen „Reflog“ zu aufzuzeichnen – ein Protokoll darüber, wo sich Ihre HEAD- und Branch-Referenzen in den letzten Monaten befunden haben.

Sie können Ihr Reflog sehen, indem Sie `git reflog` benutzen:

```
$ git reflog  
734713b HEAD@{0}: commit: fixed refs handling, added gc auto, updated  
d921970 HEAD@{1}: merge phedders/rdocs: Merge made by the 'recursive' strategy.  
1c002dd HEAD@{2}: commit: added some blame and merge stuff  
1c36188 HEAD@{3}: rebase -i (squash): updating HEAD  
95df984 HEAD@{4}: commit: # This is a combination of two commits.  
1c36188 HEAD@{5}: rebase -i (squash): updating HEAD  
7e05da5 HEAD@{6}: rebase -i (pick): updating HEAD
```

Jedes Mal, wenn die Spitze Ihrer Branch aus irgendeinem Grund aktualisiert wird, speichert Git diese Informationen für Sie in dieser temporären Historie. Sie können Ihre Reflog-Daten auch verwenden, um auf ältere Commits zu verweisen. Wenn Sie beispielsweise den fünft-letzten Wert des HEADs Ihres Repositorys sehen möchten, können Sie den Verweis `@{5}` benutzen, damit Sie diese Reflog-Ausgabe erhalten:

```
$ git show HEAD@{5}
```

Sie können diese Syntax auch verwenden, um zu sehen, wo sich ein Branch vor einer bestimmten Zeit befand. Um zum Beispiel zu sehen, wo Ihr `master` Branch gestern war, können Sie folgendes eingeben

```
$ git show master@{yesterday}
```

Das würde Ihnen zeigen, wo die Spitze Ihres `master` Branchs gestern war. Diese Technik funktioniert nur für Daten, die sich noch in Ihrem Reflog befinden, daher können Sie 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`, können Sie `git log -g` aufrufen:

```

$ git log -g master
commit 734713bc047d87bf7eac9674765ae793478c50d3
Reflog: master@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: commit: fixed refs handling, added gc auto, updated
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    fixed refs handling, added gc auto, updated tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Reflog: master@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: merge phedders/rdocs: Merge made by recursive.
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

Merge commit 'phedders/rdocs'

```

Es ist jedoch wichtig festzuhalten, dass die Reflog-Informationen ausschließlich lokale Informationen sind – es ist nur ein Protokoll dessen, was *Sie* in *Ihrem* Repository getan haben. Die Referenzen sind nicht die gleichen wie auf einer anderen Kopie des Repositorys. Gleich nachdem Sie ein Repository geklont haben, haben Sie ein leeres Reflog, da noch keine Aktivität in Ihrem Repository stattgefunden hat. Wenn Sie `git show HEAD@{2.months.ago}` ausführen, wird Ihnen der passende Commit nur angezeigt, wenn Sie das Projekt vor mindestens zwei Monaten geklont haben – wenn Sie es aber erst vor kurzem geklont haben, sehen Sie nur Ihren ersten lokalen Commit.



*Betrachten Sie das Reflog als die Shell-Historie von Git.*

Wenn Sie UNIX- oder Linux-Kenntnisse haben, können Sie sich das Reflog als die Gits-Version der Shell-Historie vorstellen, mit der Betonung, dass das, was es anzeigt, eindeutig nur für Sie und Ihre „Sitzung“ relevant ist und mit niemand anderem etwas zu tun hat, der an der gleichen Maschine arbeiten könnte.

## Abstammung der Referenzen

Die andere Hauptmethode, um einen Commit anzugeben, ist über seine Abstammung. Wenn Sie ein ^ (Zirkumflex) am Ende einer Referenz platzieren, löst Git es auf, um das übergeordnete Element dieses Commits zu bezeichnen. Angenommen, Sie schauen auf den Verlauf Ihres Projekts:

```

$ git log --pretty=format:'%h %s' --graph
* 734713b fixed refs handling, added gc auto, updated tests
*   d921970 Merge commit 'phedders/rdocs'
|\
| * 35cfb2b Some rdoc changes
* | 1c002dd added some blame and merge stuff
|/
* 1c36188 ignore *.gem
* 9b29157 add open3_detach to gemspec file list

```

Dann könnten Sie den vorherigen Commit sehen, indem Sie `HEAD^` angeben, 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. Sie können 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
```

Sie können 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 Sie beim Mergen waren (häufig `master`), während der *zweite* Elternteil eines Merge-Commits aus dem Zweig stammt, 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  
  
        added some blame and merge stuff  
  
$ git show d921970^2  
commit 35cfb2b795a55793d7cc56a6cc2060b4bb732548  
Author: Paul Hedderly <paul+git@mjr.org>  
Date: Wed Dec 10 22:22:03 2008 +0000  
  
        Some rdoc changes
```

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 Sie eine Zahl angeben. `HEAD~2` meint den „ersten Elternteil des ersten Elternteils“ oder „den Großelternteil“ – er passiert den ersten Elternteil so oft wie Sie angegeben haben. 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
```

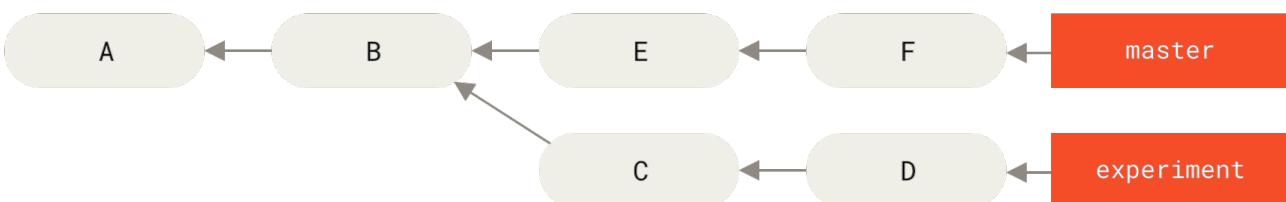
Sie können diese Syntax auch kombinieren – so können Sie den zweiten Elternteil der vorhergehenden Referenz (vorausgesetzt, es handelt sich um einen Merge Commit) erhalten, indem Sie `HEAD~3^2` verwenden, und so weiter.

## Commit-Bereiche

Nachdem Sie jetzt einzelne Commits angeben können, möchten wir Ihnen zeigen, wie Sie einen Bereich von Commits festlegen können. Besonders nützlich ist das für die Verwaltung Ihrer Branches – bei vielen Branches können Sie mit Hilfe von 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 zu bearbeiten, die von einem bestimmten Commit erreichbar sind, aber von einem anderen nicht. Angenommen, Sie haben eine Commit-Historie, die wie [Beispiel – Verlauf zur Bereichsauswahl](#) aussieht.



*Figure 136. Beispiel – Verlauf zur Bereichsauswahl*

Angenommen, Sie wollen wissen, was sich in Ihrer Branch `experiment` befindet, die noch nicht mit Ihrer Branch `master` gemerkt wurde. Sie können Git fragen, ob es Ihnen 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 sie angezeigt werden:

```
$ git log master..experiment  
D  
C
```

Wenn Sie andererseits das Gegenteil sehen wollen – alle Commits in `master`, die nicht in `experiment` sind – dann können Sie die Branch-Namen umkehren. `experiment..master` zeigt Ihnen alles in `master`, was von `experiment` nicht erreichbar ist:

```
$ git log experiment..master  
F  
E
```

Dieses Vorgehen ist praktisch, wenn Sie den Branch `experiment` auf dem aktuellen Stand halten und eine Vorschau darauf erhalten möchten, was Sie gerade verschmelzen wollen. Eine weitere häufige Anwendung dieser Syntax besteht darin, zu überprüfen, was Sie auf einen Remote pushen möchten:

```
$ git log origin/master..HEAD
```

Dieser Befehl zeigt Ihnen alle Commits in Ihrem aktuellen Branch an, die sich nicht im Branch `master` auf Ihrem Remote `origin` befinden. Wenn Sie ein `git push` ausführen und Ihr 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. Sie können auch eine Seite der Syntax weglassen, so dass Git `HEAD` annimmt. Zum Beispiel können Sie die gleichen Ergebnisse wie im vorherigen Beispiel erhalten, indem Sie `git log origin/master..` – Git ersetzt `HEAD`, wenn eine Seite fehlt.

## Mehrere Punkte

Die Doppelte-Punkt-Syntax ist als Kurzform nützlich, aber möglicherweise möchten Sie mehr als zwei Branches angeben, um Ihren Revisions-Stand anzuzeigen. So können Sie beispielsweise feststellen, welche Commits in einem oder mehreren Branches vorhanden sind aber sich nicht in dem Branch befinden, in dem Sie sich gerade aufhalten. Git ermöglicht Ihnen mit dem Zeichen `^` oder dem Zusatz `--not` vor einer Referenz, dass Sie keinen der erreichbaren Commits sehen möchten. 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 Sie mit dieser Syntax mehr als zwei Referenzen in Ihrer

Abfrage angeben können. Das ist mit der Doppelte-Punkt-Syntax (engl. Double-Dot-Syntax) nicht möglich. Wenn Sie zum Beispiel alle Commits sehen möchten, die von `refA` oder `refB` aus erreichbar sind, aber nicht von `refC` aus, können Sie 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 Ihnen dabei helfen sollte, festzustellen, was in Ihren Branches gerade enthalten ist.

### Dreifacher Punkt

Die letzte wichtige Syntax für die Bereichsauswahl ist die Triple-Dot-Syntax (Dreifach-Punkt-Syntax), die alle Commits angibt, die durch *eine* der beiden Referenzen erreichbar sind, aber nicht durch beide. Schauen Sie sich dazu die Commit-Historie in [Beispiel – Verlauf zur Bereichsauswahl](#) an. Wenn Sie wissen wollen, was sich in `master` oder `experiment` befindet, aber nicht in gemeinsamen Referenzen, können Sie diese Funktion ausführen:

```
$ git log master...experiment  
F  
E  
D  
C
```

Auch hier erhalten Sie eine normale `log` Ausgabe. Es werden Ihnen jedoch nur die Commit-Informationen für diese vier Commits angezeigt, die in der traditionellen Reihenfolge der Commit-Daten erscheinen.

Ein gängiger Parameter, der in hier mit dem `log` Befehl verwendet werden kann ist `--left-right`. Er zeigt Ihnen, 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 können Sie Git viel einfacher mitteilen, welche Commits Sie überprüfen möchten.

## Interactive Staging

In this section, you'll look at a few interactive Git commands that can help you craft your commits to include only certain combinations and parts of files. These tools are helpful if you modify a number of files extensively, then decide that you want those changes to be partitioned into several focused commits rather than one big messy commit. This way, you can make sure your commits are

logically separate changesets and can be reviewed easily by the developers working with you.

If you run `git add` with the `-i` or `--interactive` option, Git enters an interactive shell mode, displaying something like this:

```
$ 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>
```

You can see that this command shows you a much different view of your staging area than you're probably used to—basically, the same information you get with `git status` but a bit more succinct and informative. It lists the changes you've staged on the left and unstaged changes on the right.

After this comes a “Commands” section, which allows you to do a number of things like staging and unstaging files, staging parts of files, adding untracked files, and displaying diffs of what has been staged.

## Staging and Unstaging Files

If you type `u` or `2` (for update) at the `What now>` prompt, you're prompted for which files you want to stage:

```
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>>
```

To stage the `TODO` and `index.html` files, you can type the numbers:

```
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>>
```

The `*` next to each file means the file is selected to be staged. If you press Enter after typing nothing at the `Update>>` prompt, Git takes anything selected and stages it for you:

```

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

```

Now you can see that the `TODO` and `index.html` files are staged and the `simplegit.rb` file is still unstaged. If you want to unstage the `TODO` file at this point, you use the `r` or `3` (for revert) option:

```

*** 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

```

Looking at your Git status again, you can see that you've unstaged the `TODO` file:

```

*** 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

```

To see the diff of what you've staged, you can use the `d` or `6` (for diff) command. It shows you a list of your staged files, and you can select the ones for which you would like to see the staged diff. This is much like specifying `git diff --cached` on the command line:

```

*** 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">

```

With these basic commands, you can use the interactive add mode to deal with your staging area a little more easily.

## Staging Patches

It's also possible for Git to stage certain *parts* of files and not the rest. For example, if you make two changes to your `simplegit.rb` file and want to stage one of them and not the other, doing so is very easy in Git. From the same interactive prompt explained in the previous section, type `p` or `5` (for patch). Git will ask you which files you would like to partially stage; then, for each section of the selected files, it will display hunks of the file diff and ask if you would like to stage them, one by one:

```

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,?]?

```

You have a lot of options at this point. Typing `?` shows a list of what you can do:

```
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
```

Generally, you'll type `y` or `n` if you want to stage each hunk, but staging all of them in certain files or skipping a hunk decision until later can be helpful too. If you stage one part of the file and leave another part unstaged, your status output will look like this:

```
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
```

The status of the `simplegit.rb` file is interesting. It shows you that a couple of lines are staged and a couple are unstaged. You've partially staged this file. At this point, you can exit the interactive adding script and run `git commit` to commit the partially staged files.

You also don't need to be in interactive add mode to do the partial-file staging—you can start the same script by using `git add -p` or `git add --patch` on the command line.

Furthermore, you can use patch mode for partially resetting files with the `git reset --patch` command, for checking out parts of files with the `git checkout --patch` command and for stashing parts of files with the `git stash save --patch` command. We'll go into more details on each of these as we get to more advanced usages of these commands.

## Stashing and Cleaning

Often, when you've been working on part of your project, things are in a messy state and you want to switch branches for a bit to work on something else. The problem is, you don't want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is the `git stash` command.

Stashing takes the dirty state of your working directory—that is, your modified tracked files and staged changes—and saves it on a stack of unfinished changes that you can reapply at any time (even on a different branch).

### Migrating to `git stash push`

As of late October 2017, there has been extensive discussion on the Git mailing list, wherein the command `git stash save` is being deprecated in favour of the existing alternative `git stash push`. The main reason for this is that `git stash push` introduces the option of stashing selected *pathspecs*, something `git stash save` does not support.

`git stash save` is not going away any time soon, so don't worry about it suddenly disappearing. But you might want to start migrating over to the `push` alternative for the new functionality.

## Stashing Your Work

To demonstrate stashing, you'll go into your project and start working on a couple of files and possibly stage one of the changes. If you run `git status`, you can see your dirty state:

```
$ 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
```

Now you want to switch branches, but you don't want to commit what you've been working on yet, so you'll stash the changes. To push a new stash onto your stack, run `git stash` or `git stash push`:

```
$ git stash
Saved working directory and index state \
"WIP on master: 049d078 added the index file"
HEAD is now at 049d078 added the index file
(To restore them type "git stash apply")
```

You can now see that your working directory is clean:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

At this point, you can switch branches and do work elsewhere; your changes are stored on your stack. To see which stashes you've stored, you can use `git stash list`:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
```

In this case, two stashes were saved previously, so you have access to three different stashed works. You can reapply the one you just stashed by using the command shown in the help output of the original stash command: `git stash apply`. If you want to apply one of the older stashes, you can specify it by naming it, like this: `git stash apply stash@{2}`. If you don't specify a stash, Git assumes the most recent stash and tries to apply it:

```
$ 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")
```

You can see that Git re-modifies the files you reverted when you saved the stash. In this case, you had a clean working directory when you tried to apply the stash, and you tried to apply it on the same branch you saved it from. Having a clean working directory and applying it on the same branch aren't necessary to successfully apply a stash. You can save a stash on one branch, switch to another branch later, and try to reapply the changes. You can also have modified and uncommitted files in your working directory when you apply a stash — Git gives you merge conflicts if anything no longer applies cleanly.

The changes to your files were reapplied, but the file you staged before wasn't restaged. To do that, you must run the `git stash apply` command with a `--index` option to tell the command to try to reapply the staged changes. If you had run that instead, you'd have gotten back to your original position:

```
$ 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
```

The `apply` option only tries to apply the stashed work—you continue to have it on your stack. To remove it, you can run `git stash drop` with the name of the stash to remove:

```
$ git stash list
stash@{0}: WIP on master: 049d078 added the index file
stash@{1}: WIP on master: c264051 Revert "added file_size"
stash@{2}: WIP on master: 21d80a5 added number to log
$ git stash drop stash@{0}
Dropped stash@{0} (364e91f3f268f0900bc3ee613f9f733e82aaed43)
```

You can also run `git stash pop` to apply the stash and then immediately drop it from your stack.

## Creative Stashing

There are a few stash variants that may also be helpful. The first option that is quite popular is the `--keep-index` option to the `git stash` command. This tells Git to not only include all staged content in the stash being created, but simultaneously leave it in the index.

```
$ 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
```

Another common thing you may want to do with stash is to stash the untracked files as well as the tracked ones. By default, `git stash` will stash only modified and staged *tracked* files. If you specify `--include-untracked` or `-u`, Git will include untracked files in the stash being created. However, including untracked files in the stash will still not include explicitly *ignored* files; to additionally include ignored files, use `--all` (or just `-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
$
```

Finally, if you specify the `--patch` flag, Git will not stash everything that is modified but will instead prompt you interactively which of the changes you would like to stash and which you would like to keep in your working directory.

```
$ git stash --patch
diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index 66d332e..8bb5674 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -16,6 +16,10 @@ class SimpleGit
      return '#{git_cmd} 2>&1'.chomp
    end
  end

+
+  def show(treeish = 'master')
+    command("git show #{treeish}")
+  end

end
test
Stash this hunk [y,n,q,a,d,,e,?]?
```

Saved working directory and index state WIP on master: 1b65b17 added the index file

## Creating a Branch from a Stash

If you stash some work, leave it there for a while, and continue on the branch from which you stashed the work, you may have a problem reapplying the work. If the apply tries to modify a file that you've since modified, you'll get a merge conflict and will have to try to resolve it. If you want an easier way to test the stashed changes again, you can run `git stash branch <new branchname>`, which creates a new branch for you with your selected branch name, checks out the commit you were on when you stashed your work, reapplies your work there, and then drops the stash if it applies successfully:

```
$ 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)
```

This is a nice shortcut to recover stashed work easily and work on it in a new branch.

## Bereinigung des Arbeitsverzeichnisses

Finally, you may not want to stash some work or files in your working directory, but simply get rid of them; that's what the `git clean` command is for.

Some common reasons for cleaning your working directory might be to remove cruft that has been generated by merges or external tools or to remove build artifacts in order to run a clean build.

You'll want to be pretty careful with this command, since it's designed to remove files from your working directory that are not tracked. If you change your mind, there is often no retrieving the content of those files. A safer option is to run `git stash --all` to remove everything but save it in a stash.

Assuming you do want to remove cruft files or clean your working directory, you can do so with `git clean`. To remove all the untracked files in your working directory, you can run `git clean -f -d`, which removes any files and also any subdirectories that become empty as a result. The `-f` means *force* or “really do this,” and is required if the Git configuration variable `clean.requireForce` is not explicitly set to false.

If you ever want to see what it would do, you can run the command with the `--dry-run` (or `-n`) option, which means “do a dry run and tell me what you *would* have removed”.

```
$ git clean -d -n
Would remove test.o
Would remove tmp/
```

By default, the `git clean` command will only remove untracked files that are not ignored. Any file that matches a pattern in your `.gitignore` or other ignore files will not be removed. If you want to

remove those files too, such as to remove all `.o` files generated from a build so you can do a fully clean build, you can add a `-x` to the clean command.

```
$ 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/
```

If you don't know what the `git clean` command is going to do, always run it with a `-n` first to double check before changing the `-n` to a `-f` and doing it for real. The other way you can be careful about the process is to run it with the `-i` or "interactive" flag.

This will run the clean command in an interactive mode.

```
$ 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>
```

This way you can step through each file individually or specify patterns for deletion interactively.



There is a quirky situation where you might need to be extra forceful in asking Git to clean your working directory. If you happen to be in a working directory under which you've copied or cloned other Git repositories (perhaps as submodules), even `git clean -fd` will refuse to delete those directories. In cases like that, you need to add a second `-f` option for emphasis.

## Ihre Arbeit signieren

Git is cryptographically secure, but it's not foolproof. If you're taking work from others on the internet and want to verify that commits are actually from a trusted source, Git has a few ways to sign and verify work using GPG.

## GPG Introduction

First of all, if you want to sign anything you need to get GPG configured and your personal key installed.

```
$ 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
```

If you don't have a key installed, you can generate one with `gpg --gen-key`.

```
$ gpg --gen-key
```

Once you have a private key to sign with, you can configure Git to use it for signing things by setting the `user.signingkey` config setting.

```
$ git config --global user.signingkey 0A46826A
```

Now Git will use your key by default to sign tags and commits if you want.

## Sigining Tags

If you have a GPG private key setup, you can now use it to sign new tags. All you have to do is use `-s` instead of `-a`:

```
$ git tag -s v1.5 -m 'my signed 1.5 tag'  
  
You need a passphrase to unlock the secret key for  
user: "Ben Straub <ben@straub.cc>"  
2048-bit RSA key, ID 800430EB, created 2014-05-04
```

If you run `git show` on that tag, you can see your GPG signature attached to it:

```
$ git show v1.5
tag v1.5
Tagger: Ben Straub <ben@straub.cc>
Date:   Sat May 3 20:29:41 2014 -0700

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1

iQEcbAABAgAGBQJTZbQIAoJEF0+sviABDDrZbQH/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
```

changed the version number

## Verifying Tags

To verify a signed tag, you use `git tag -v <tag-name>`. This command uses GPG to verify the signature. You need the signer's public key in your keyring for this to work properly:

```
$ 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
```

If you don't have the signer's public key, you get something like this instead:

```
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'
```

## Signing Commits

In more recent versions of Git (v1.7.9 and above), you can now also sign individual commits. If you're interested in signing commits directly instead of just the tags, all you need to do is add a `-S` to your `git commit` command.

```
$ 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
```

To see and verify these signatures, there is also a `--show-signature` option to `git log`.

```
$ git log --show-signature -1
commit 5c3386cf54bba0a33a32da706aa52bc0155503c2
gpg: Signature made Wed Jun 4 19:49:17 2014 PDT using RSA key ID 0A46826A
gpg: Good signature from "Scott Chacon (Git signing key) <schacon@gmail.com>
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Jun 4 19:49:17 2014 -0700

      signed commit
```

Additionally, you can configure `git log` to check any signatures it finds and list them in its output with the `%G?` format.

```
$ git log --pretty="format:%h %G? %aN  %s"
5c3386c G Scott Chacon  signed commit
ca82a6d N Scott Chacon  changed the version number
085bb3b N Scott Chacon  removed unnecessary test code
a11bef0 N Scott Chacon  first commit
```

Here we can see that only the latest commit is signed and valid and the previous commits are not.

In Git 1.8.3 and later, `git merge` and `git pull` can be told to inspect and reject when merging a commit that does not carry a trusted GPG signature with the `--verify-signatures` command.

If you use this option when merging a branch and it contains commits that are not signed and valid, the merge will not work.

```
$ git merge --verify-signatures non-verify
fatal: Commit ab06180 does not have a GPG signature.
```

If the merge contains only valid signed commits, the merge command will show you all the signatures it has checked and then move forward with the merge.

```
$ git merge --verify-signatures signed-branch
Commit 13ad65e has a good GPG signature by Scott Chacon (Git signing key)
<schacon@gmail.com>
Updating 5c3386c..13ad65e
Fast-forward
 README | 2 ++
 1 file changed, 2 insertions(+)
```

You can also use the `-S` option with the `git merge` command to sign the resulting merge commit itself. The following example both verifies that every commit in the branch to be merged is signed and furthermore signs the resulting 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(+)
```

## Everyone Must Sign

Signing tags and commits is great, but if you decide to use this in your normal workflow, you'll have to make sure that everyone on your team understands how to do so. If you don't, you'll end up spending a lot of time helping people figure out how to rewrite their commits with signed versions. Make sure you understand GPG and the benefits of signing things before adopting this as part of your standard workflow.

## Searching

With just about any size codebase, you'll often need to find where a function is called or defined, or display the history of a method. Git provides a couple of useful tools for looking through the code and commits stored in its database quickly and easily. We'll go through a few of them.

## Git Grep

Git ships with a command called `grep` that allows you to easily search through any committed tree, the working directory, or even the index for a string or regular expression. For the examples that follow, we'll search through the source code for Git itself.

By default, `git grep` will look through the files in your working directory. As a first variation, you can use either of the `-n` or `--line-number` options to print out the line numbers where Git has found matches:

```
$ 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
```

In addition to the basic search shown above, `git grep` supports a plethora of other interesting options.

For instance, instead of printing all of the matches, you can ask `git grep` to summarize the output by showing you only which files contained the search string and how many matches there were in each file with the `-c` or `--count` option:

```
$ 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
```

If you're interested in the *context* of a search string, you can display the enclosing method or function for each matching string with either of the `-P` or `--show-function` options:

```
$ 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 */
```

As you can see, the `gmtime_r` routine is called from both the `match_multi_number` and `match_digit` functions in the `date.c` file (the third match displayed represents just the string appearing in a comment).

You can also search for complex combinations of strings with the `--and` flag, which ensures that multiple matches must occur in the same line of text. For instance, let's look for any lines that define a constant whose name contains *either* of the substrings “LINK” or “BUF\_MAX”, specifically in an older version of the Git codebase represented by the tag `v1.8.0` (we'll throw in the `--break` and `--heading` options which help split up the output into a more readable format):

```
$ 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 */
```

The `git grep` command has a few advantages over normal searching commands like `grep` and `ack`. The first is that it's really fast, the second is that you can search through any tree in Git, not just the working directory. As we saw in the above example, we looked for terms in an older version of the Git source code, not the version that was currently checked out.

## Git Log Searching

Perhaps you're looking not for *where* a term exists, but *when* it existed or was introduced. The `git log` command has a number of powerful tools for finding specific commits by the content of their messages or even the content of the diff they introduce.

If, for example, we want to find out when the `ZLIB_BUF_MAX` constant was originally introduced, we can use the `-S` option (colloquially referred to as the Git “pickaxe” option) to tell Git to show us only those commits that changed the number of occurrences of that string.

```
$ 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
```

If we look at the diff of those commits, we can see that in `ef49a7a` the constant was introduced and in `e01503b` it was modified.

If you need to be more specific, you can provide a regular expression to search for with the `-G` option.

### Line Log Search

Another fairly advanced log search that is insanely useful is the line history search. Simply run `git log` with the `-L` option, and it will show you the history of a function or line of code in your codebase.

For example, if we wanted to see every change made to the function `git_deflate_bound` in the `zlib.c` file, we could run `git log -L :git_deflate_bound:zlib.c`. This will try to figure out what the bounds of that function are and then look through the history and show us every change that was made to the function as a series of patches back to when the function was first created.

```
$ 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);
+}
+
```

If Git can't figure out how to match a function or method in your programming language, you can also provide it with a regular expression (or *regex*). For example, this would have done the same thing as the example above: `git log -L '/unsigned long git_deflate_bound/','/^}':zlib.c`. You could also give it a range of lines or a single line number and you'll get the same sort of output.

## Rewriting History

Many times, when working with Git, you may want to revise your local commit history. One of the great things about Git is that it allows you to make decisions at the last possible moment. You can decide what files go into which commits right before you commit with the staging area, you can decide that you didn't mean to be working on something yet with `git stash`, and you can rewrite commits that already happened so they look like they happened in a different way. This can involve changing the order of the commits, changing messages or modifying files in a commit, squashing

together or splitting apart commits, or removing commits entirely—all before you share your work with others.

In this section, you’ll see how to accomplish these tasks so that you can make your commit history look the way you want before you share it with others.

*Don’t push your work until you’re happy with it*



One of the cardinal rules of Git is that, since so much work is local within your clone, you have a great deal of freedom to rewrite your history *locally*. However, once you push your work, it is a different story entirely, and you should consider pushed work as final unless you have good reason to change it. In short, you should avoid pushing your work until you’re happy with it and ready to share it with the rest of the world.

## Changing the Last Commit

Changing your most recent commit is probably the most common rewriting of history that you’ll do. You’ll often want to do two basic things to your last commit: simply change the commit message, or change the actual content of the commit by adding, removing and modifying files.

If you simply want to modify your last commit message, that’s easy:

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

The command above loads the previous commit message into an editor session, where you can make changes to the message, save those changes and exit. When you save and close the editor, the editor writes a new commit containing that updated commit message and makes it your new last commit.

If, on the other hand, you want to change the actual *content* of your last commit, the process works basically the same way—first make the changes you think you forgot, stage those changes, and the subsequent `git commit --amend` replaces that last commit with your new, improved commit.

You need to be careful with this technique because amending changes the SHA-1 of the commit. It’s like a very small rebase—don’t amend your last commit if you’ve already pushed it.

*An amended commit may (or may not) need an amended commit message*

When you amend a commit, you have the opportunity to change both the commit message and the content of the commit. If you amend the content of the commit substantially, you should almost certainly update the commit message to reflect that amended content.



On the other hand, if your amendments are suitably trivial (fixing a silly typo or adding a file you forgot to stage) such that the earlier commit message is just fine, you can simply make the changes, stage them, and avoid the unnecessary editor session entirely with:

```
$ git commit --amend --no-edit
```

## Changing Multiple Commit Messages

To modify a commit that is farther back in your history, you must move to more complex tools. Git doesn't have a modify-history tool, but you can use the rebase tool to rebase a series of commits onto the HEAD they were originally based on instead of moving them to another one. With the interactive rebase tool, you can then stop after each commit you want to modify and change the message, add files, or do whatever you wish. You can run rebase interactively by adding the `-i` option to `git rebase`. You must indicate how far back you want to rewrite commits by telling the command which commit to rebase onto.

For example, if you want to change the last three commit messages, or any of the commit messages in that group, you supply as an argument to `git rebase -i` the parent of the last commit you want to edit, which is `HEAD~2^` or `HEAD~3`. It may be easier to remember the `~3` because you're trying to edit the last three commits, but keep in mind that you're actually designating four commits ago, the parent of the last commit you want to edit:

```
$ git rebase -i HEAD~3
```

Remember again that this is a rebasing command—every commit in the range `HEAD~3..HEAD` with a changed message *and all of its descendants* will be rewritten. Don't include any commit you've already pushed to a central server—doing so will confuse other developers by providing an alternate version of the same change.

Running this command gives you a list of commits in your text editor that looks something like this:

```

pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file

# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick <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

```

It's important to note that these commits are listed in the opposite order than you normally see them using the `log` command. If you run a `log`, you see something like this:

```

$ git log --pretty=format:"%h %s" HEAD~3..HEAD
a5f4a0d added cat-file
310154e updated README formatting and added blame
f7f3f6d changed my name a bit

```

Notice the reverse order. The interactive rebase gives you a script that it's going to run. It will start at the commit you specify on the command line (`HEAD~3`) and replay the changes introduced in each of these commits from top to bottom. It lists the oldest at the top, rather than the newest, because that's the first one it will replay.

You need to edit the script so that it stops at the commit you want to edit. To do so, change the word 'pick' to the word 'edit' for each of the commits you want the script to stop after. For example, to modify only the third commit message, you change the file to look like this:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

When you save and exit the editor, Git rewinds you back to the last commit in that list and drops you on the command line with the following message:

```
$ git rebase -i HEAD~3
Stopped at f7f3f6d... changed my name a bit
You can amend the commit now, with

    git commit --amend

Once you're satisfied with your changes, run

    git rebase --continue
```

These instructions tell you exactly what to do. Type

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

Change the commit message, and exit the editor. Then, run

```
$ git rebase --continue
```

This command will apply the other two commits automatically, and then you're done. If you change pick to edit on more lines, you can repeat these steps for each commit you change to edit. Each time, Git will stop, let you amend the commit, and continue when you're finished.

## Reordering Commits

You can also use interactive rebases to reorder or remove commits entirely. If you want to remove the “added cat-file” commit and change the order in which the other two commits are introduced, you can change the rebase script from this

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

to this:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

When you save and exit the editor, Git rewinds your branch to the parent of these commits, applies [310154e](#) and then [f7f3f6d](#), and then stops. You effectively change the order of those commits and remove the “added cat-file” commit completely.

## Squashing Commits

It's also possible to take a series of commits and squash them down into a single commit with the interactive rebasing tool. The script puts helpful instructions in the rebase message:

```
#  
# 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
```

If, instead of “pick” or “edit”, you specify “squash”, Git applies both that change and the change directly before it and makes you merge the commit messages together. So, if you want to make a single commit from these three commits, you make the script look like this:

```
pick f7f3f6d changed my name a bit  
squash 310154e updated README formatting and added blame  
squash a5f4a0d added cat-file
```

When you save and exit the editor, Git applies all three changes and then puts you back into the editor to merge the three commit messages:

```
# This is a combination of 3 commits.  
# The first commit's message is:  
changed my name a bit  
  
# This is the 2nd commit message:  
  
updated README formatting and added blame  
  
# This is the 3rd commit message:  
  
added cat-file
```

When you save that, you have a single commit that introduces the changes of all three previous commits.

## Splitting a Commit

Splitting a commit undoes a commit and then partially stages and commits as many times as commits you want to end up with. For example, suppose you want to split the middle commit of your three commits. Instead of “updated README formatting and added blame”, you want to split it into two commits: “updated README formatting” for the first, and “added blame” for the second. You can do that in the `rebase -i` script by changing the instruction on the commit you want to split to “edit”:

```
pick f7f3f6d changed my name a bit  
edit 310154e updated README formatting and added blame  
pick a5f4a0d added cat-file
```

Then, when the script drops you to the command line, you reset that commit, take the changes that have been reset, and create multiple commits out of them. When you save and exit the editor, Git rewinds to the parent of the first commit in your list, applies the first commit (`f7f3f6d`), applies the second (`310154e`), and drops you to the console. There, you can do a mixed reset of that commit with `git reset HEAD^`, which effectively undoes that commit and leaves the modified files unstaged. Now you can stage and commit files until you have several commits, and run `git rebase --continue` when you’re done:

```
$ git reset HEAD^  
$ git add README  
$ git commit -m 'updated README formatting'  
$ git add lib/simplegit.rb  
$ git commit -m 'added blame'  
$ git rebase --continue
```

Git applies the last commit (`a5f4a0d`) in the script, and your history looks like this:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Once again, this changes the SHA-1s of all the commits in your list, so make sure no commit shows up in that list that you've already pushed to a shared repository.

## The Nuclear Option: filter-branch

There is another history-rewriting option that you can use if you need to rewrite a larger number of commits in some scriptable way—for instance, changing your email address globally or removing a file from every commit. The command is `filter-branch`, and it can rewrite huge swaths of your history, so you probably shouldn't use it unless your project isn't yet public and other people haven't based work off the commits you're about to rewrite. However, it can be very useful. You'll learn a few of the common uses so you can get an idea of some of the things it's capable of.

### Removing a File from Every Commit

This occurs fairly commonly. Someone accidentally commits a huge binary file with a thoughtless `git add .`, and you want to remove it everywhere. Perhaps you accidentally committed a file that contained a password, and you want to make your project open source. `filter-branch` is the tool you probably want to use to scrub your entire history. To remove a file named `passwords.txt` from your entire history, you can use the `--tree-filter` option to `filter-branch`:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

The `--tree-filter` option runs the specified command after each checkout of the project and then recommits the results. In this case, you remove a file called `passwords.txt` from every snapshot, whether it exists or not. If you want to remove all accidentally committed editor backup files, you can run something like `git filter-branch --tree-filter 'rm -f *~' HEAD`.

You'll be able to watch Git rewriting trees and commits and then move the branch pointer at the end. It's generally a good idea to do this in a testing branch and then hard-reset your master branch after you've determined the outcome is what you really want. To run `filter-branch` on all your branches, you can pass `--all` to the command.

### Making a Subdirectory the New Root

Suppose you've done an import from another source control system and have subdirectories that make no sense (`trunk`, `tags`, and so on). If you want to make the `trunk` subdirectory be the new project root for every commit, `filter-branch` can help you do that, too:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Now your new project root is what was in the `trunk` subdirectory each time. Git will also automatically remove commits that did not affect the subdirectory.

## Changing Email Addresses Globally

Another common case is that you forgot to run `git config` to set your name and email address before you started working, or perhaps you want to open-source a project at work and change all your work email addresses to your personal address. In any case, you can change email addresses in multiple commits in a batch with `filter-branch` as well. You need to be careful to change only the email addresses that are yours, so you use `--commit-filter`:

```
$ git filter-branch --commit-filter '
  if [ "$GIT_AUTHOR_EMAIL" = "schacon@localhost" ];
  then
    GIT_AUTHOR_NAME="Scott Chacon";
    GIT_AUTHOR_EMAIL="schacon@example.com";
    git commit-tree "$@";
  else
    git commit-tree "$@";
  fi' HEAD
```

This goes through and rewrites every commit to have your new address. Because commits contain the SHA-1 values of their parents, this command changes every commit SHA-1 in your history, not just those that have the matching email address.

## Reset Demystified

Before moving on to more specialized tools, let's talk about the Git `reset` and `checkout` commands. These commands are two of the most confusing parts of Git when you first encounter them. They do so many things that it seems hopeless to actually understand them and employ them properly. For this, we recommend a simple metaphor.

## The Three Trees

An easier way to think about `reset` and `checkout` is through the mental frame of Git being a content manager of three different trees. By “tree” here, we really mean “collection of files”, not specifically the data structure. (There are a few cases where the index doesn’t exactly act like a tree, but for our purposes it is easier to think about it this way for now.)

Git as a system manages and manipulates three trees in its normal operation:

| Tree              | Role                              |
|-------------------|-----------------------------------|
| HEAD              | Last commit snapshot, next parent |
| Index             | Proposed next commit snapshot     |
| Working Directory | Sandbox                           |

## The HEAD

HEAD is the pointer to the current branch reference, which is in turn a pointer to the last commit made on that branch. That means HEAD will be the parent of the next commit that is created. It's generally simplest to think of HEAD as the snapshot of **your last commit on that branch**.

In fact, it's pretty easy to see what that snapshot looks like. Here is an example of getting the actual directory listing and SHA-1 checksums for each file in the HEAD snapshot:

```
$ 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
```

The Git `cat-file` and `ls-tree` commands are “plumbing” commands that are used for lower level things and not really used in day-to-day work, but they help us see what's going on here.

## The Index

The *index* is your **proposed next commit**. We've also been referring to this concept as Git's “Staging Area” as this is what Git looks at when you run `git commit`.

Git populates this index with a list of all the file contents that were last checked out into your working directory and what they looked like when they were originally checked out. You then replace some of those files with new versions of them, and `git commit` converts that into the tree for a new commit.

```
$ git ls-files -s
100644 a906cb2a4a904a152e80877d4088654daad0c859 0 README
100644 8f94139338f9404f26296befa88755fc2598c289 0 Rakefile
100644 47c6340d6459e05787f644c2447d2595f5d3a54b 0 lib/simplegit.rb
```

Again, here we're using `git ls-files`, which is more of a behind the scenes command that shows you what your index currently looks like.

The index is not technically a tree structure—it's actually implemented as a flattened manifest—but for our purposes it's close enough.

## The Working Directory

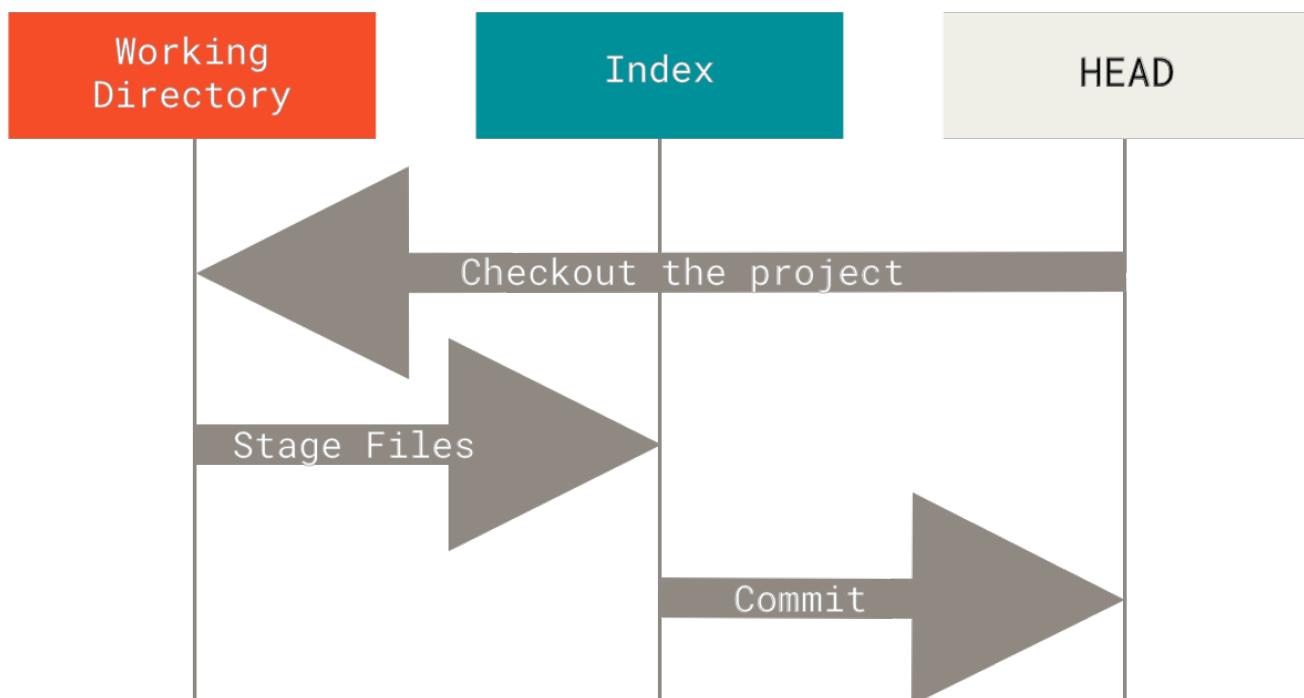
Finally, you have your *working directory* (also commonly referred to as the “working tree”). The other two trees store their content in an efficient but inconvenient manner, inside the `.git` folder. The working directory unpacks them into actual files, which makes it much easier for you to edit them. Think of the working directory as a **sandbox**, where you can try changes out before committing them to your staging area (index) and then to history.

```
$ tree
.
├── README
├── Rakefile
└── lib
    └── simplegit.rb

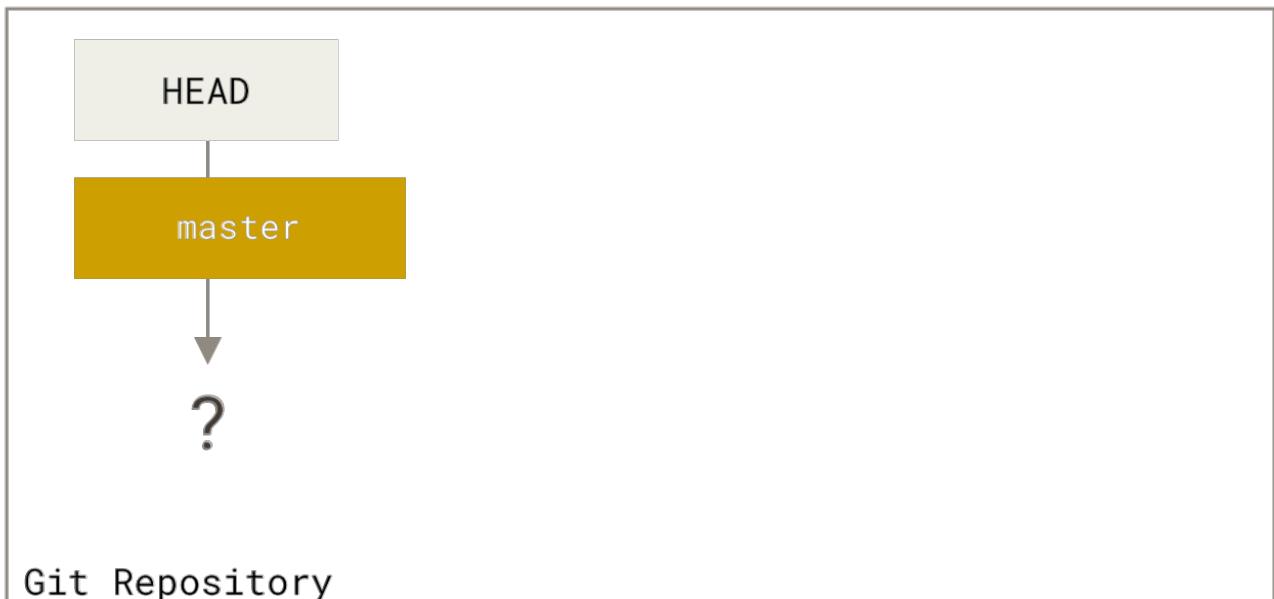
1 directory, 3 files
```

## The Workflow

Git's typical workflow is to record snapshots of your project in successively better states, by manipulating these three trees.

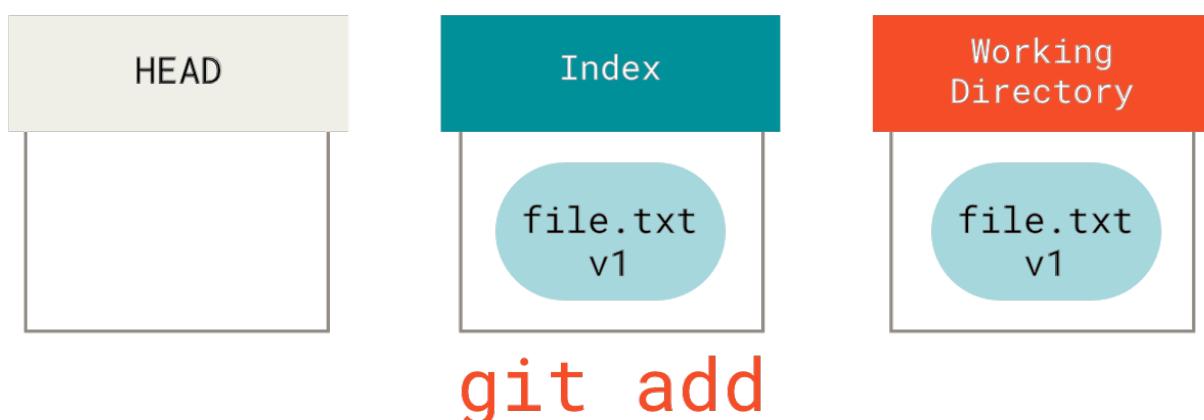


Let's visualize this process: say you go into a new directory with a single file in it. We'll call this `v1` of the file, and we'll indicate it in blue. Now we run `git init`, which will create a Git repository with a `HEAD` reference which points to the unborn `master` branch.

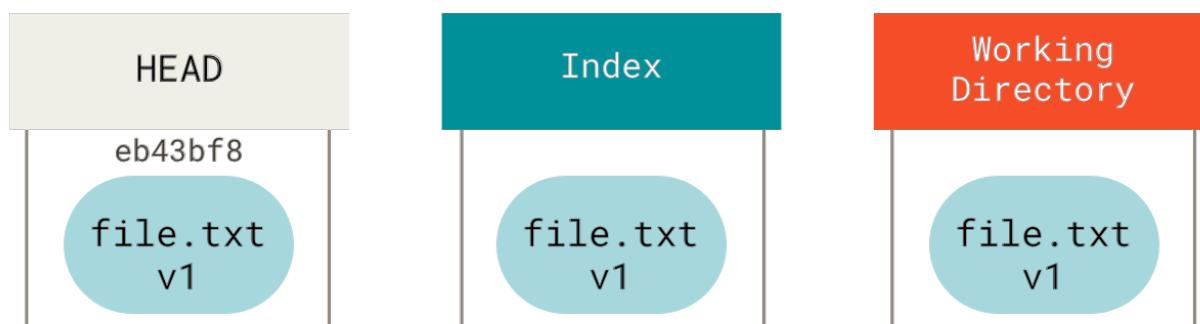
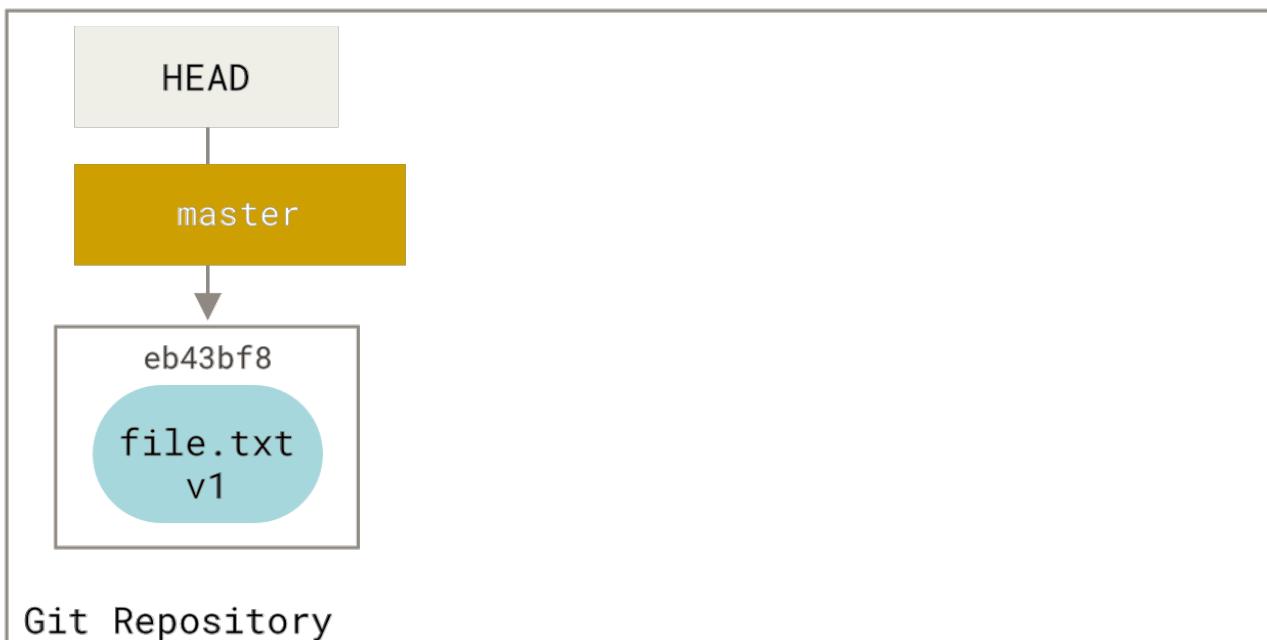


At this point, only the working directory tree has any content.

Now we want to commit this file, so we use `git add` to take content in the working directory and copy it to the index.



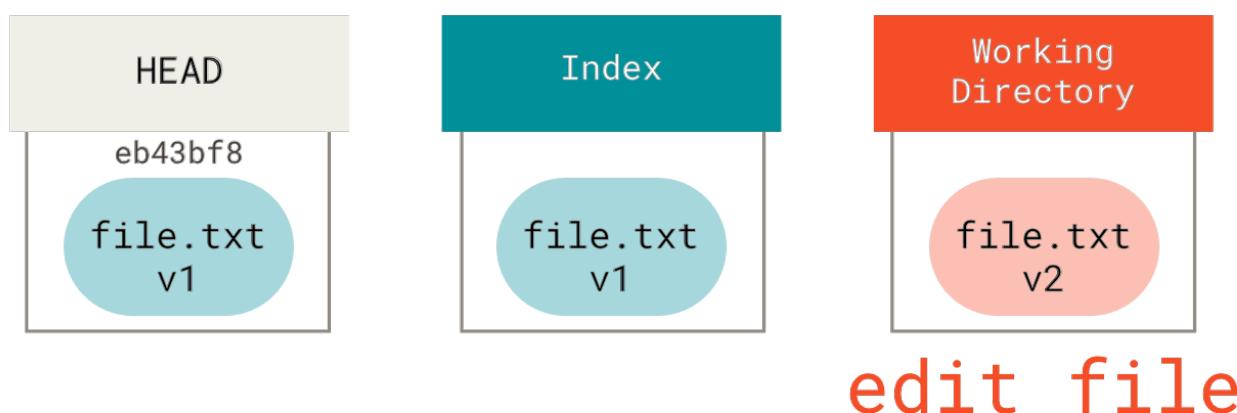
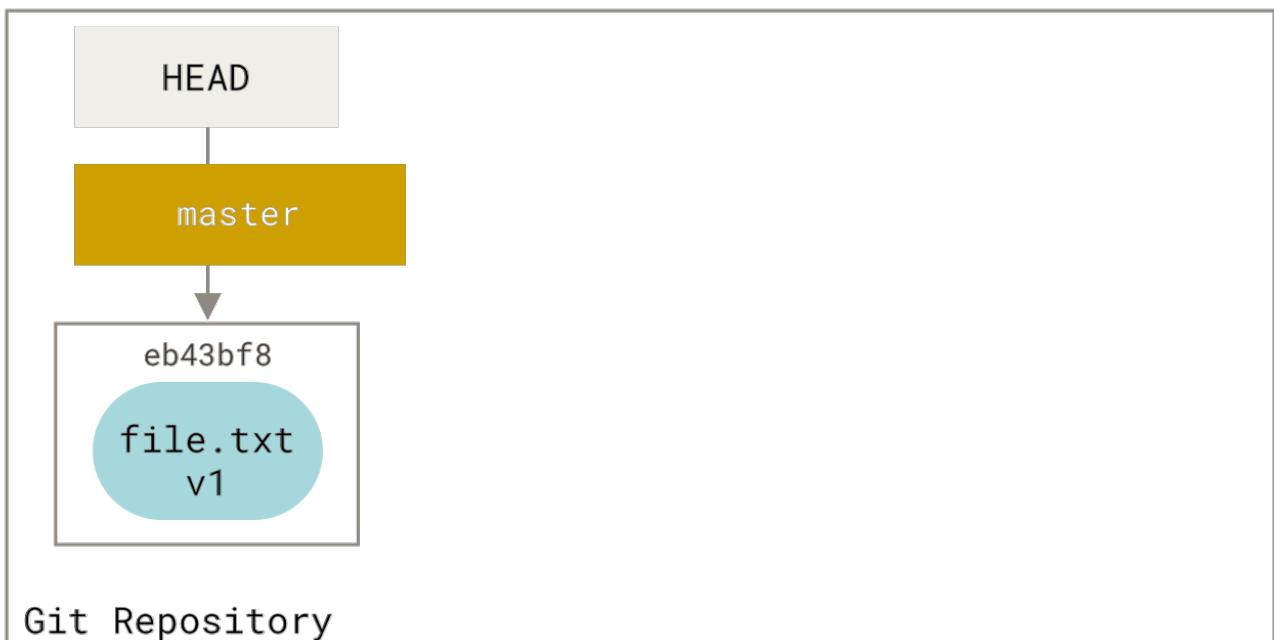
Then we run `git commit`, which takes the contents of the index and saves it as a permanent snapshot, creates a commit object which points to that snapshot, and updates `master` to point to that commit.



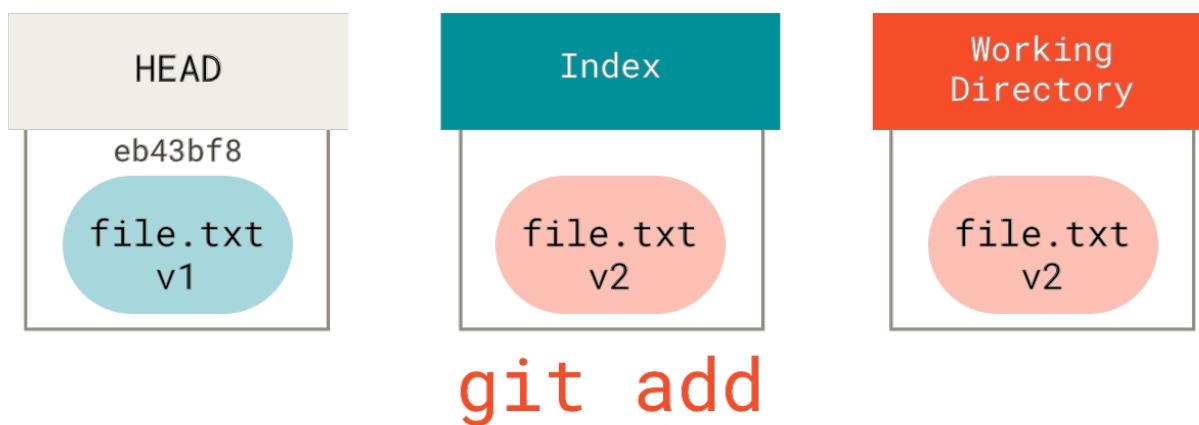
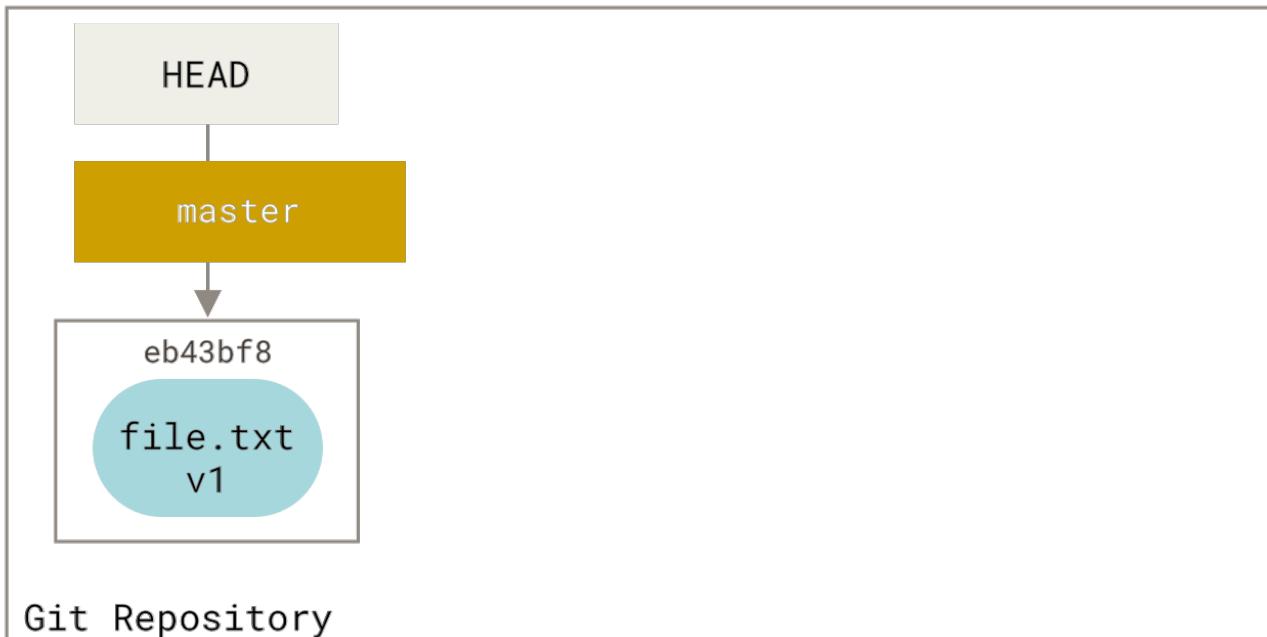
## git commit

If we run `git status`, we'll see no changes, because all three trees are the same.

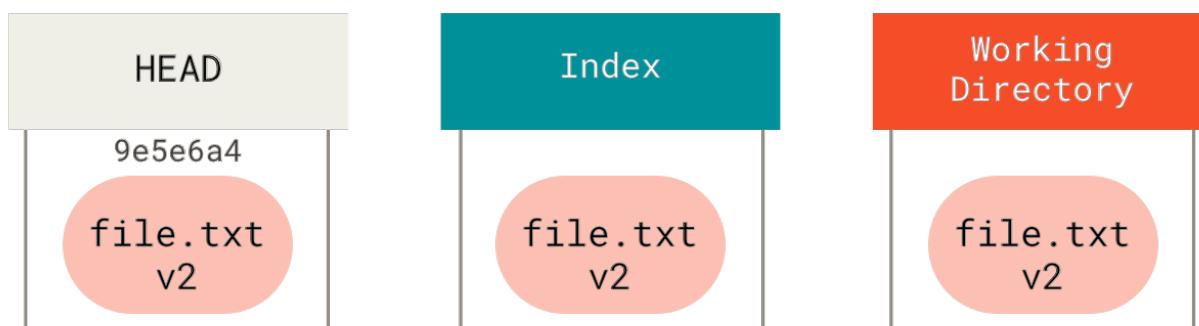
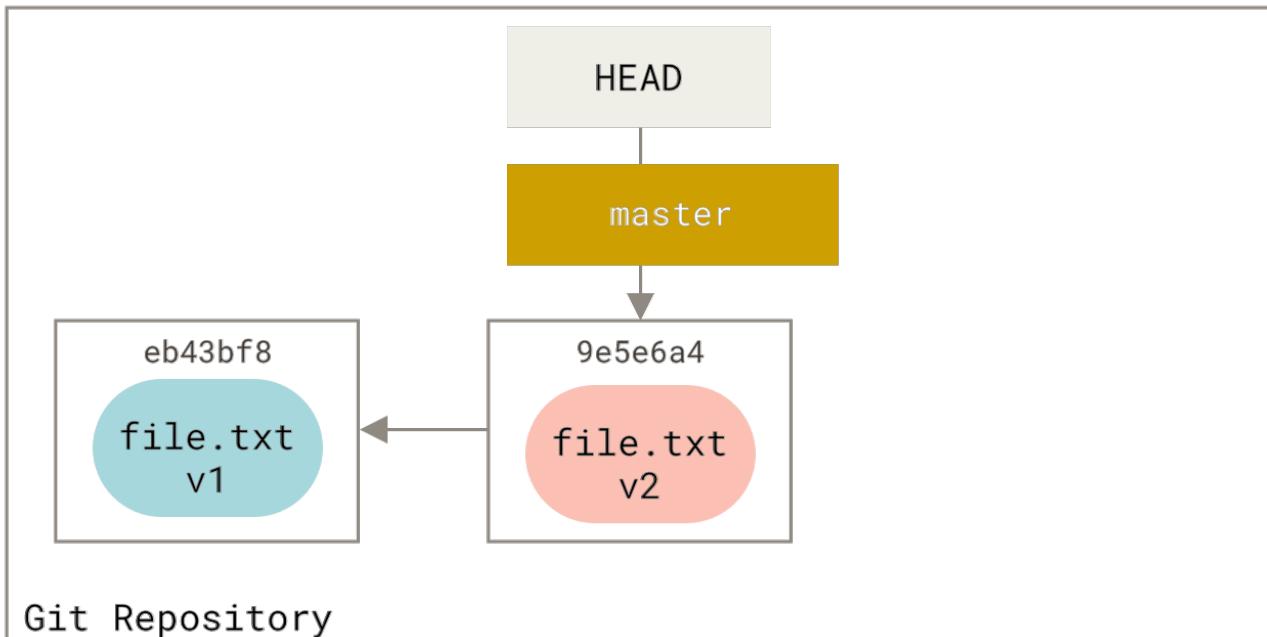
Now we want to make a change to that file and commit it. We'll go through the same process; first, we change the file in our working directory. Let's call this **v2** of the file, and indicate it in red.



If we run `git status` right now, we'll see the file in red as "Changes not staged for commit," because that entry differs between the index and the working directory. Next we run `git add` on it to stage it into our index.



At this point, if we run `git status`, we will see the file in green under “Changes to be committed” because the index and HEAD differ—that is, our proposed next commit is now different from our last commit. Finally, we run `git commit` to finalize the commit.



## git commit

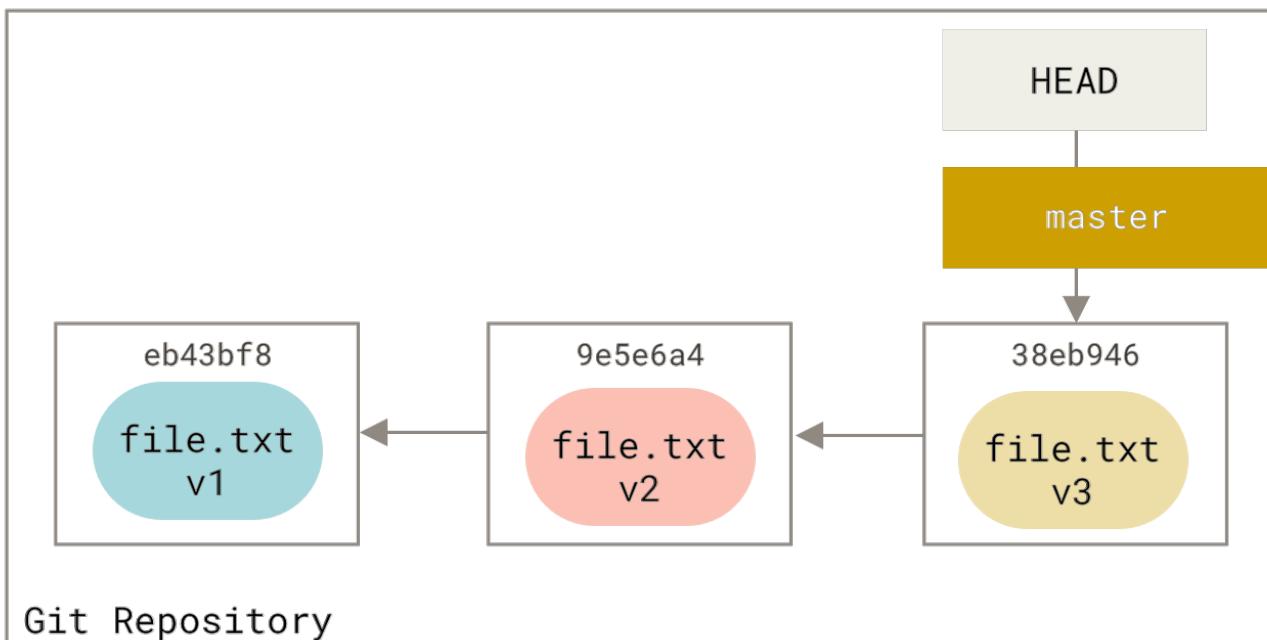
Now `git status` will give us no output, because all three trees are the same again.

Switching branches or cloning goes through a similar process. When you checkout a branch, it changes **HEAD** to point to the new branch ref, populates your **index** with the snapshot of that commit, then copies the contents of the **index** into your **working Directory**.

## The Role of Reset

The `reset` command makes more sense when viewed in this context.

For the purposes of these examples, let's say that we've modified `file.txt` again and committed it a third time. So now our history looks like this:



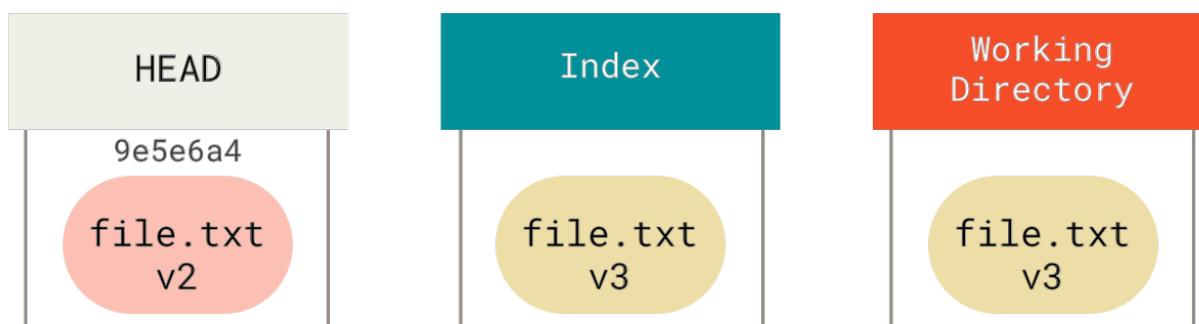
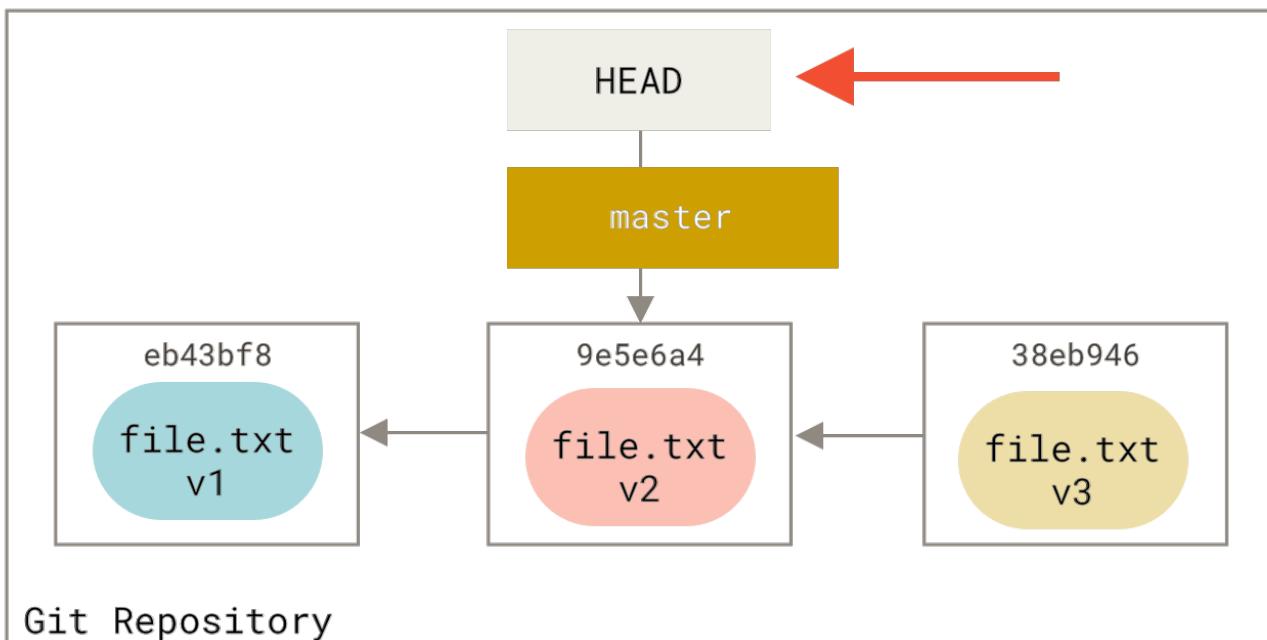
HEAD



Let's now walk through exactly what `reset` does when you call it. It directly manipulates these three trees in a simple and predictable way. It does up to three basic operations.

### Step 1: Move HEAD

The first thing `reset` will do is move what HEAD points to. This isn't the same as changing HEAD itself (which is what `checkout` does); `reset` moves the branch that HEAD is pointing to. This means if HEAD is set to the `master` branch (i.e. you're currently on the `master` branch), running `git reset 9e5e6a4` will start by making `master` point to `9e5e6a4`.



**git reset --soft HEAD~**

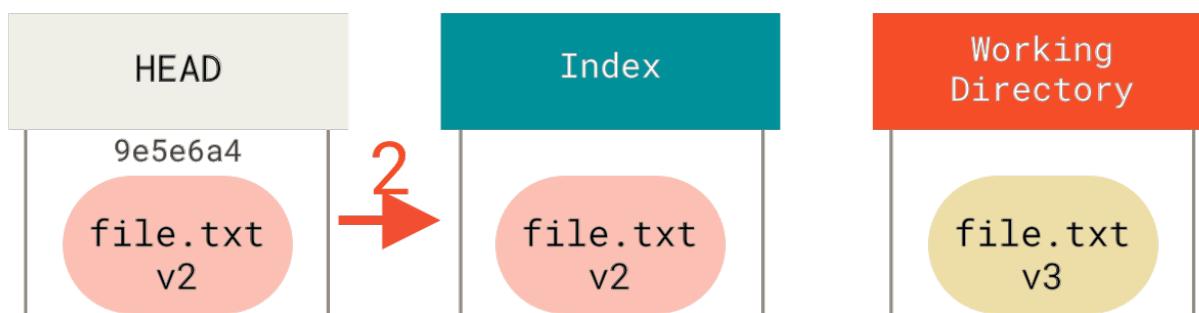
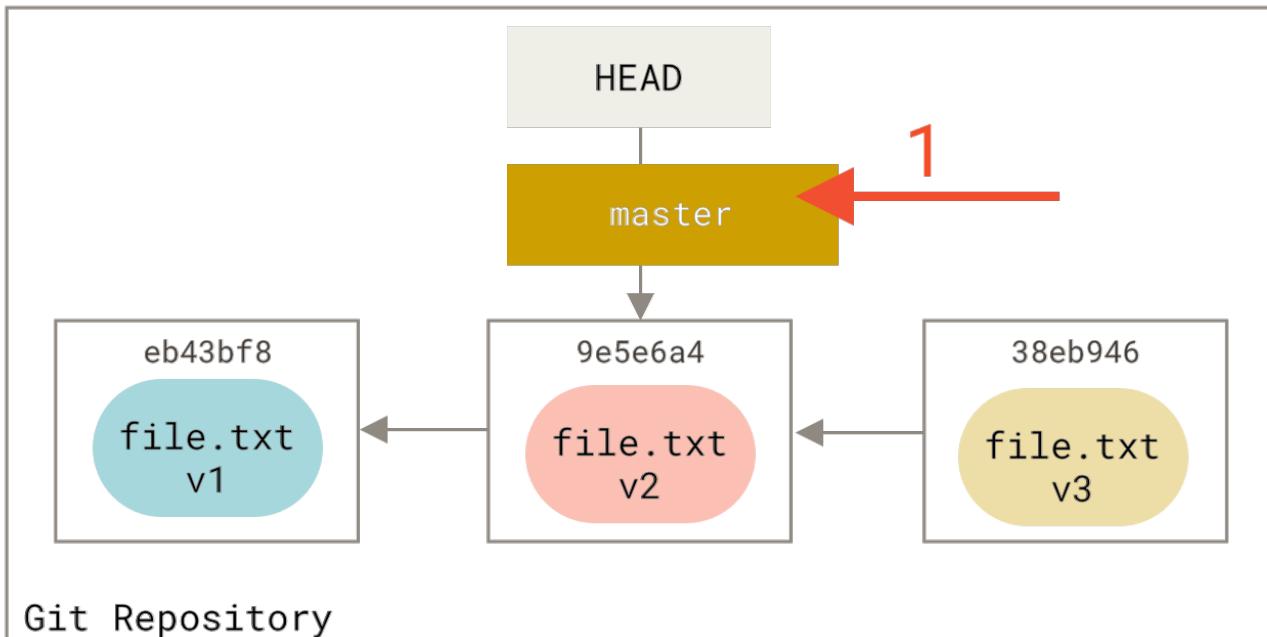
No matter what form of `reset` with a commit you invoke, this is the first thing it will always try to do. With `reset --soft`, it will simply stop there.

Now take a second to look at that diagram and realize what happened: it essentially undid the last `git commit` command. When you run `git commit`, Git creates a new commit and moves the branch that `HEAD` points to up to it. When you `reset` back to `HEAD~` (the parent of `HEAD`), you are moving the branch back to where it was, without changing the index or working directory. You could now update the index and run `git commit` again to accomplish what `git commit --amend` would have done (see [Changing the Last Commit](#)).

### Step 2: Updating the Index (`--mixed`)

Note that if you run `git status` now you'll see in green the difference between the index and what the new `HEAD` is.

The next thing `reset` will do is to update the index with the contents of whatever snapshot `HEAD` now points to.



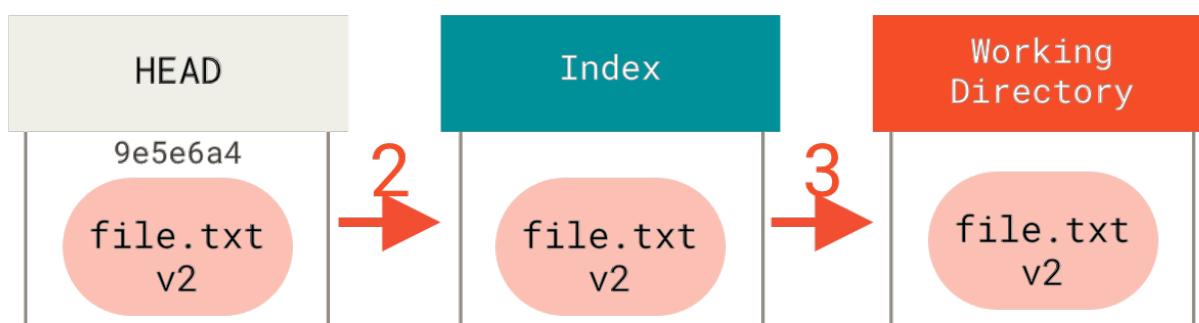
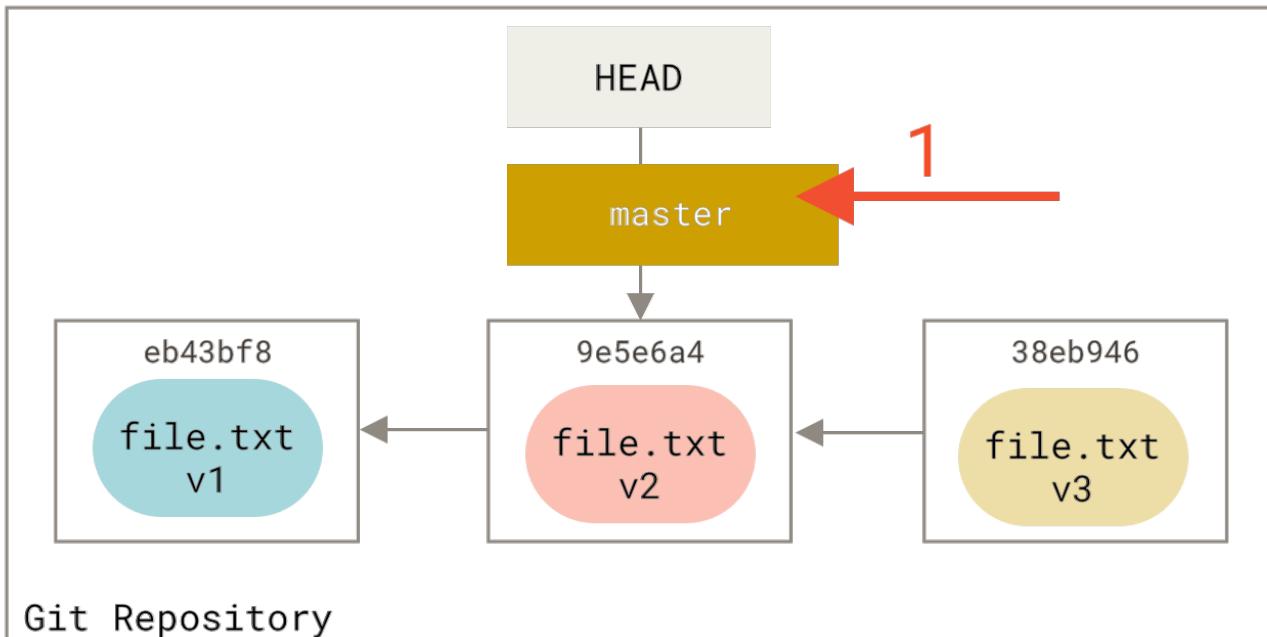
**git reset [--mixed] HEAD~**

If you specify the `--mixed` option, `reset` will stop at this point. This is also the default, so if you specify no option at all (just `git reset HEAD~` in this case), this is where the command will stop.

Now take another second to look at that diagram and realize what happened: it still undid your last `commit`, but also *unstaged* everything. You rolled back to before you ran all your `git add` and `git commit` commands.

### Step 3: Updating the Working Directory (`--hard`)

The third thing that `reset` will do is to make the working directory look like the index. If you use the `--hard` option, it will continue to this stage.



**git reset --hard HEAD~**

So let's think about what just happened. You undid your last commit, the `git add` and `git commit` commands, **and** all the work you did in your working directory.

It's important to note that this flag (`--hard`) is the only way to make the `reset` command dangerous, and one of the very few cases where Git will actually destroy data. Any other invocation of `reset` can be pretty easily undone, but the `--hard` option cannot, since it forcibly overwrites files in the working directory. In this particular case, we still have the `v3` version of our file in a commit in our Git DB, and we could get it back by looking at our `reflog`, but if we had not committed it, Git still would have overwritten the file and it would be unrecoverable.

## Recap

The `reset` command overwrites these three trees in a specific order, stopping when you tell it to:

1. Move the branch `HEAD` points to (*stop here if `--soft`*)
2. Make the index look like `HEAD` (*stop here unless `--hard`*)
3. Make the working directory look like the index

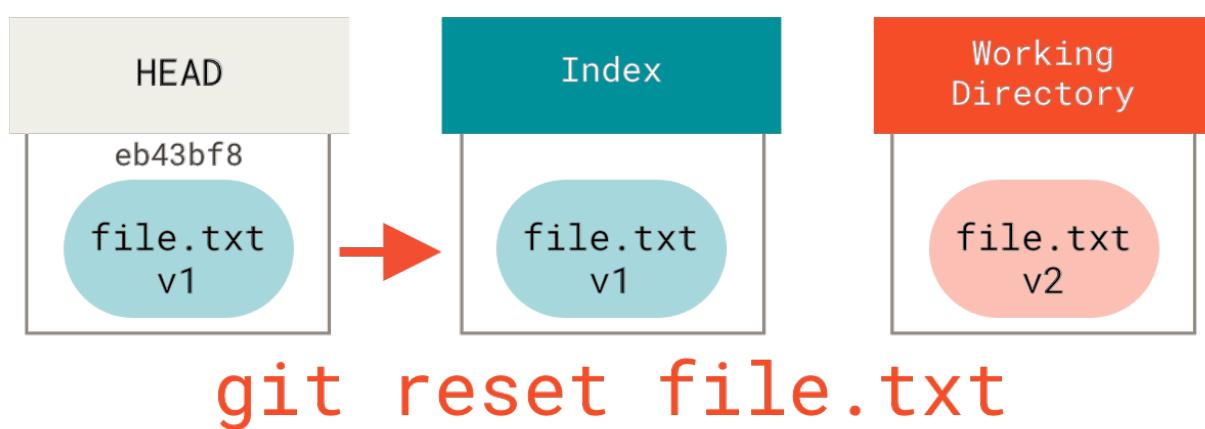
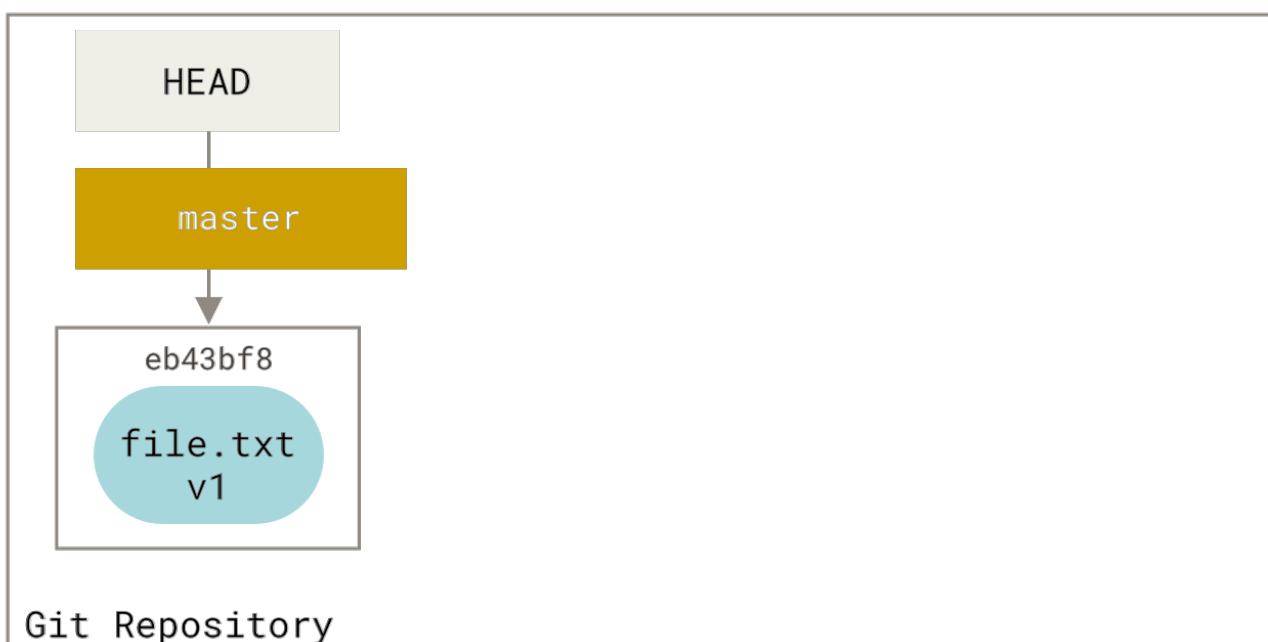
## Reset With a Path

That covers the behavior of `reset` in its basic form, but you can also provide it with a path to act upon. If you specify a path, `reset` will skip step 1, and limit the remainder of its actions to a specific file or set of files. This actually sort of makes sense—HEAD is just a pointer, and you can't point to part of one commit and part of another. But the index and working directory *can* be partially updated, so `reset` proceeds with steps 2 and 3.

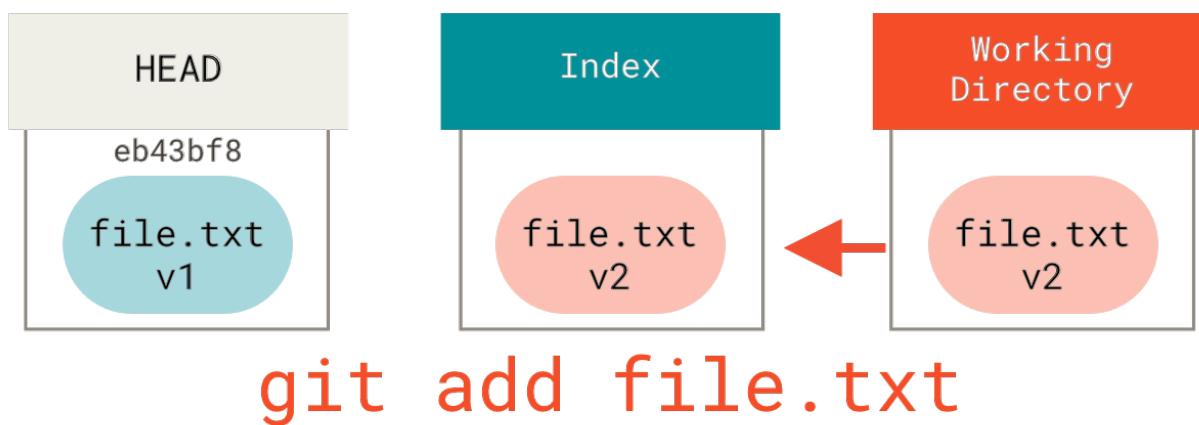
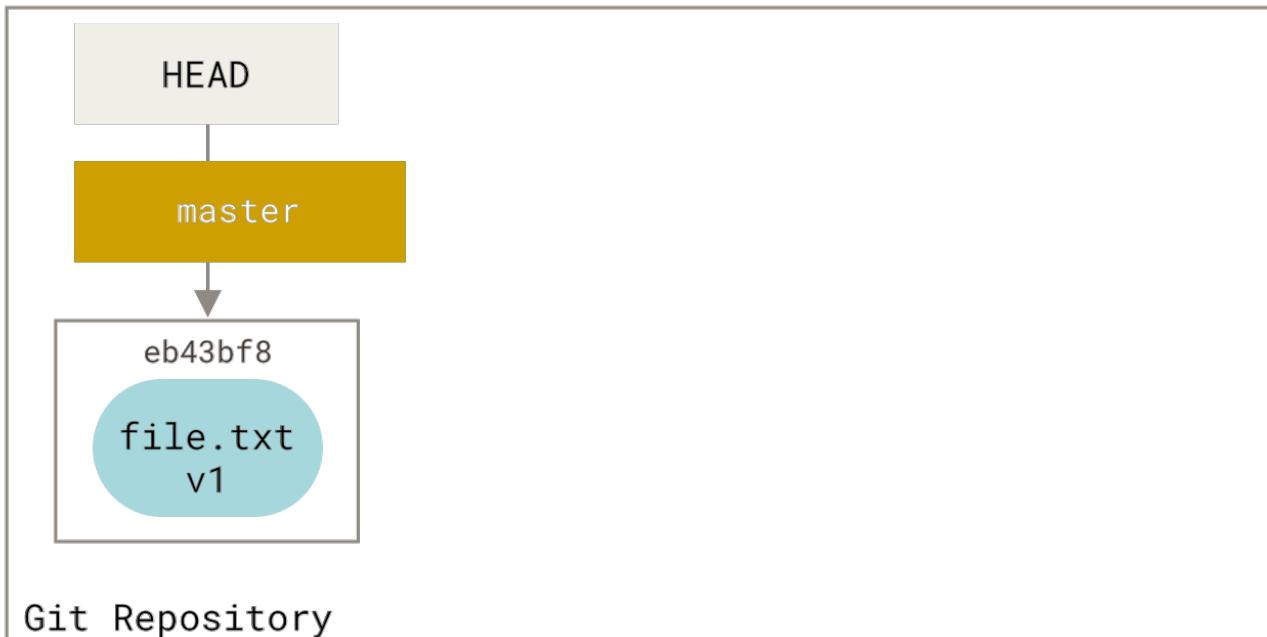
So, assume we run `git reset file.txt`. This form (since you did not specify a commit SHA-1 or branch, and you didn't specify `--soft` or `--hard`) is shorthand for `git reset --mixed HEAD file.txt`, which will:

1. Move the branch HEAD points to (*skipped*)
2. Make the index look like HEAD (*stop here*)

So it essentially just copies `file.txt` from HEAD to the index.

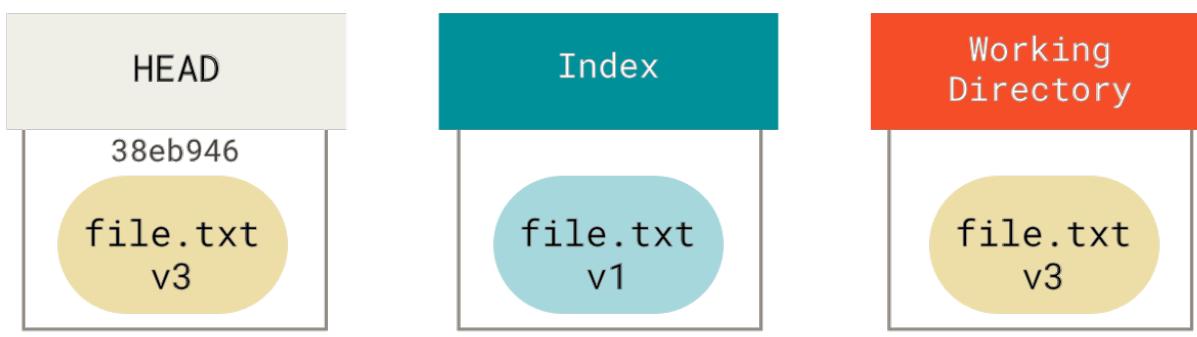
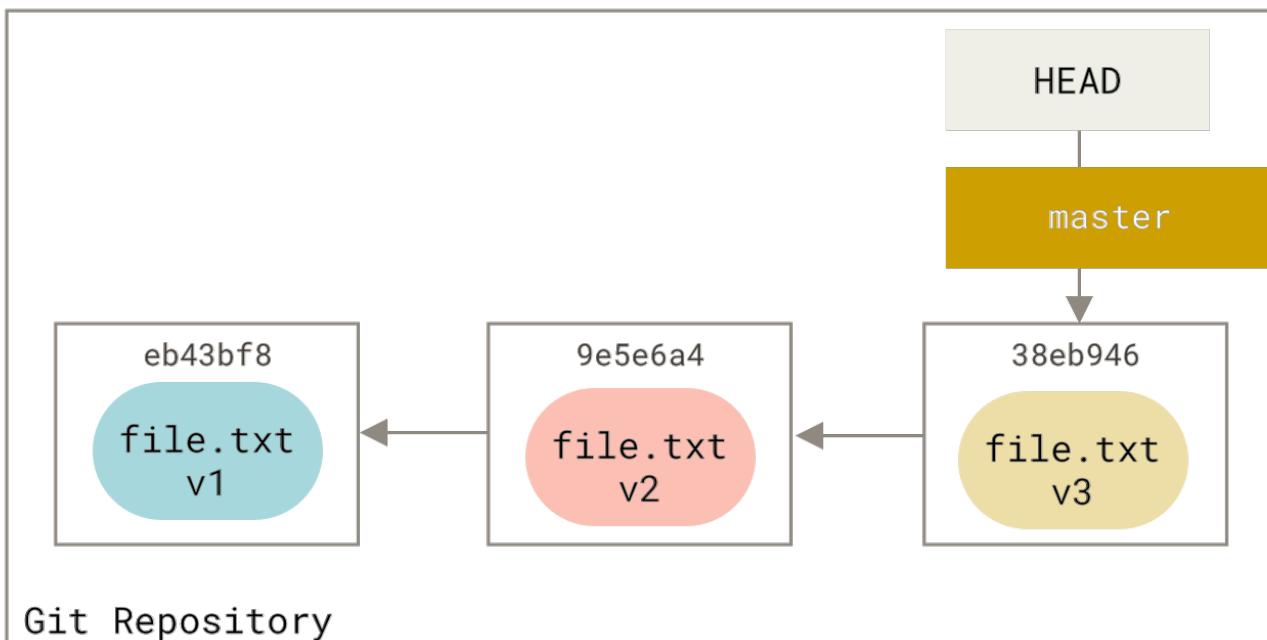


This has the practical effect of *unstaging* the file. If we look at the diagram for that command and think about what `git add` does, they are exact opposites.



This is why the output of the `git status` command suggests that you run this to unstage a file. (See [Eine Datei aus der Staging Area entnehmen](#) for more on this.)

We could just as easily not let Git assume we meant “pull the data from HEAD” by specifying a specific commit to pull that file version from. We would just run something like `git reset eb43bf file.txt`.



**git reset eb43 -- file.txt**

This effectively does the same thing as if we had reverted the content of the file to **v1** in the working directory, ran `git add` on it, then reverted it back to **v3** again (without actually going through all those steps). If we run `git commit` now, it will record a change that reverts that file back to **v1**, even though we never actually had it in our working directory again.

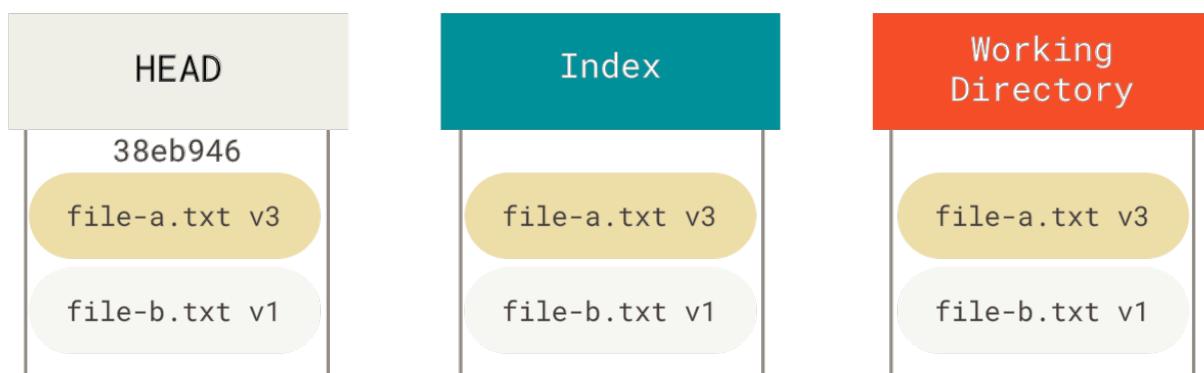
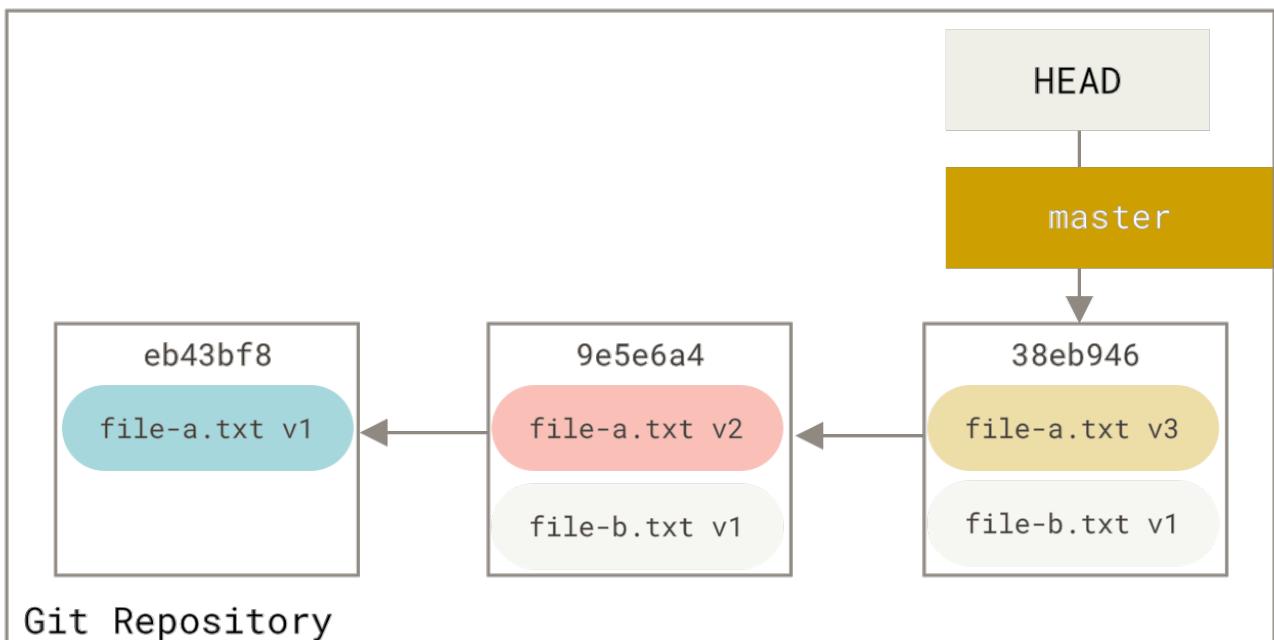
It's also interesting to note that like `git add`, the `reset` command will accept a `--patch` option to unstage content on a hunk-by-hunk basis. So you can selectively unstage or revert content.

## Squashing

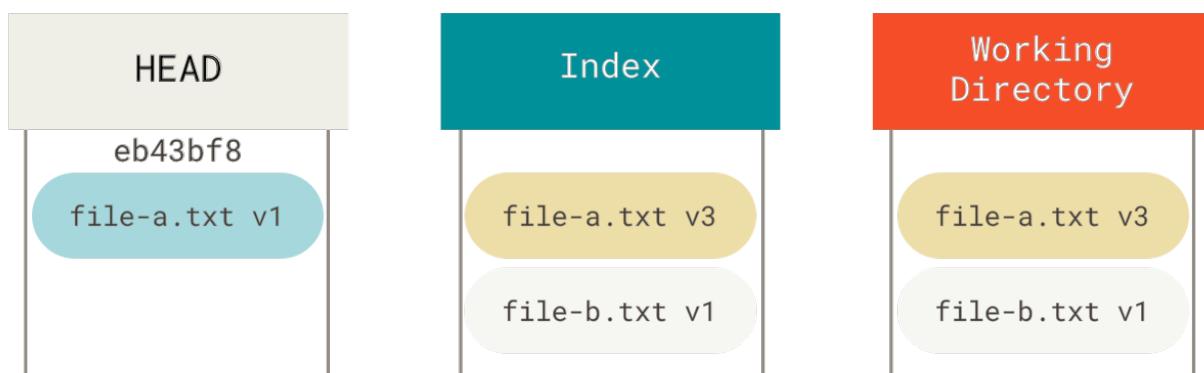
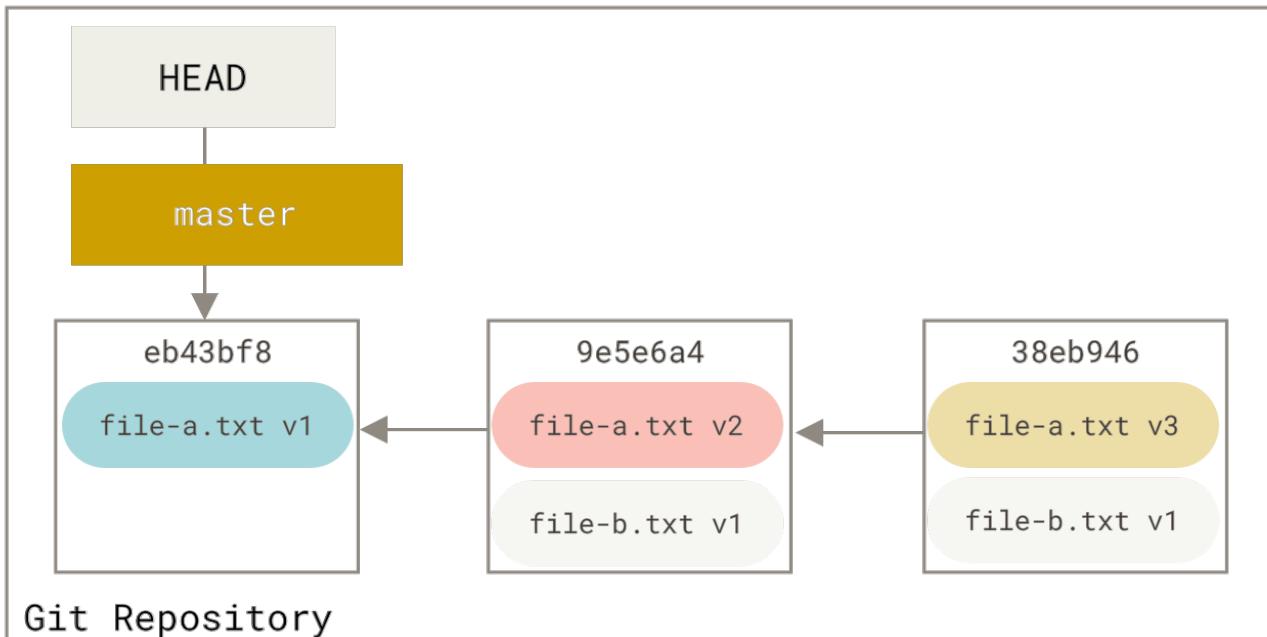
Let's look at how to do something interesting with this newfound power — squashing commits.

Say you have a series of commits with messages like “oops.”, “WIP” and “forgot this file”. You can use `reset` to quickly and easily squash them into a single commit that makes you look really smart. ([Squashing Commits](#) shows another way to do this, but in this example it's simpler to use `reset`.)

Let's say you have a project where the first commit has one file, the second commit added a new file and changed the first, and the third commit changed the first file again. The second commit was a work in progress and you want to squash it down.

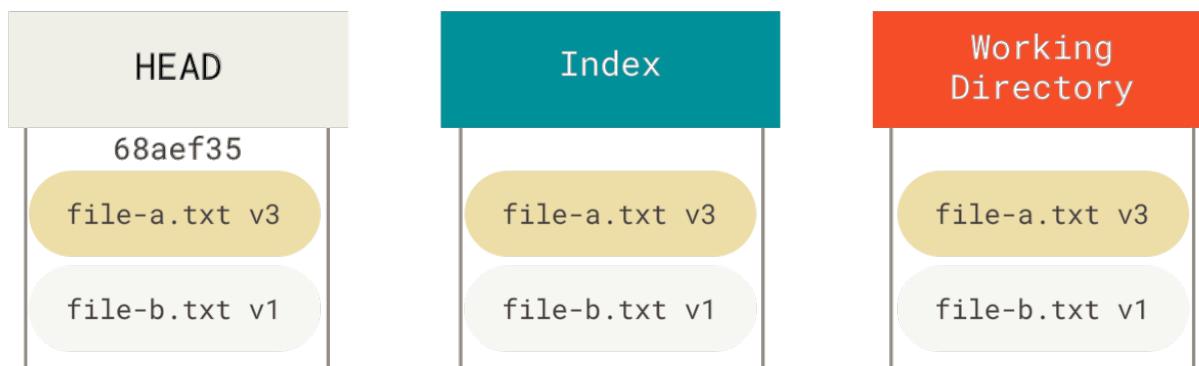
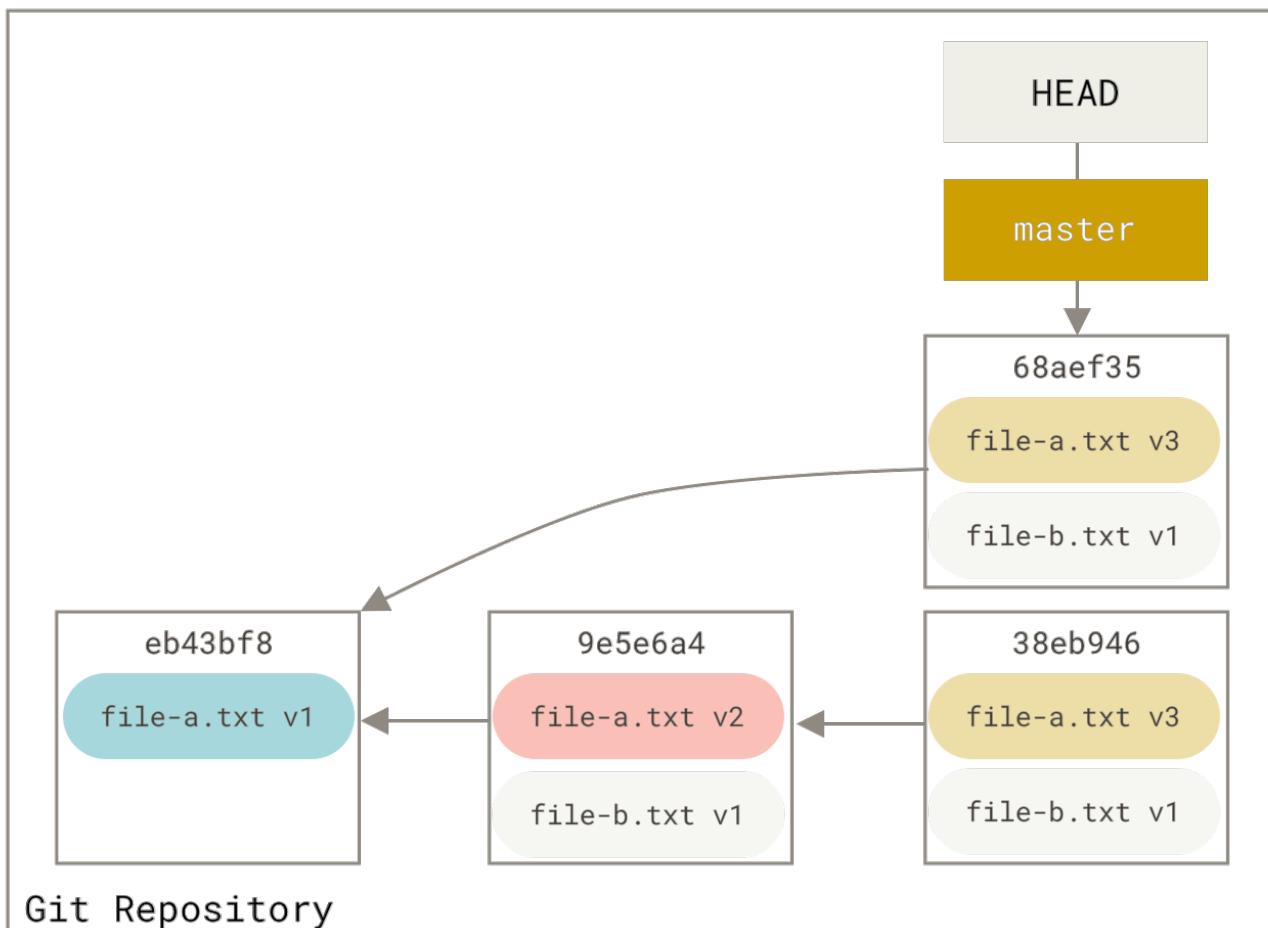


You can run `git reset --soft HEAD~2` to move the HEAD branch back to an older commit (the most recent commit you want to keep):



**git reset --soft HEAD~2**

And then simply run `git commit` again:



## git commit

Now you can see that your reachable history, the history you would push, now looks like you had one commit with `file-a.txt` v1, then a second that both modified `file-a.txt` to v3 and added `file-b.txt`. The commit with the v2 version of the file is no longer in the history.

### Check It Out

Finally, you may wonder what the difference between `checkout` and `reset` is. Like `reset`, `checkout` manipulates the three trees, and it is a bit different depending on whether you give the command a file path or not.

## Without Paths

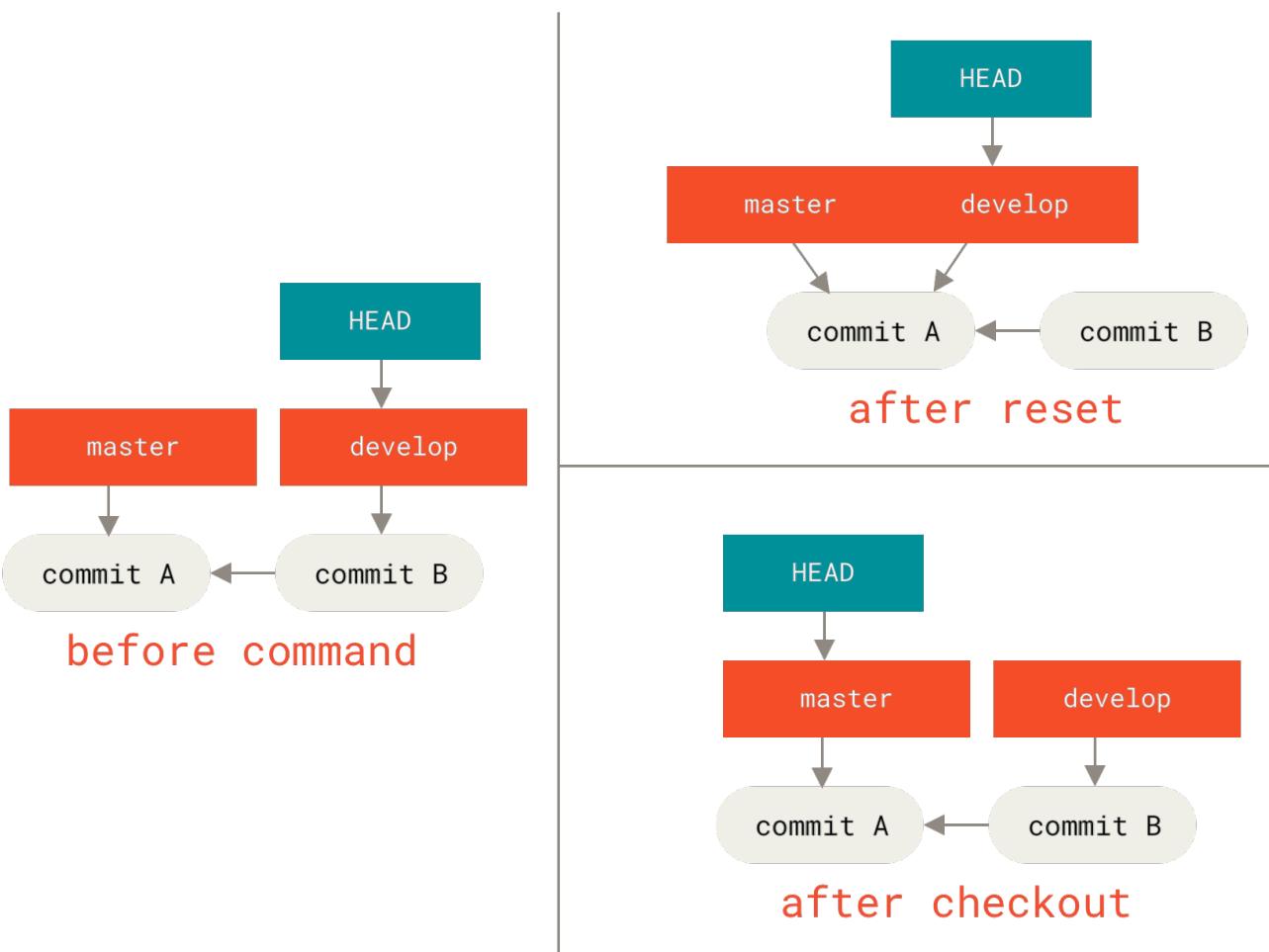
Running `git checkout [branch]` is pretty similar to running `git reset --hard [branch]` in that it updates all three trees for you to look like `[branch]`, but there are two important differences.

First, unlike `reset --hard`, `checkout` is working-directory safe; it will check to make sure it's not blowing away files that have changes to them. Actually, it's a bit smarter than that—it tries to do a trivial merge in the working directory, so all of the files you *haven't* changed will be updated. `reset --hard`, on the other hand, will simply replace everything across the board without checking.

The second important difference is how `checkout` updates HEAD. Whereas `reset` will move the branch that HEAD points to, `checkout` will move HEAD itself to point to another branch.

For instance, say we have `master` and `develop` branches which point at different commits, and we're currently on `develop` (so HEAD points to it). If we run `git reset master`, `develop` itself will now point to the same commit that `master` does. If we instead run `git checkout master`, `develop` does not move, HEAD itself does. HEAD will now point to `master`.

So, in both cases we're moving HEAD to point to commit A, but *how* we do so is very different. `reset` will move the branch HEAD points to, `checkout` moves HEAD itself.



## With Paths

The other way to run `checkout` is with a file path, which, like `reset`, does not move HEAD. It is just like `git reset [branch] file` in that it updates the index with that file at that commit, but it also

overwrites the file in the working directory. It would be exactly like `git reset --hard [branch] file` (if `reset` would let you run that) — it's not working-directory safe, and it does not move HEAD.

Also, like `git reset` and `git add, checkout` will accept a `--patch` option to allow you to selectively revert file contents on a hunk-by-hunk basis.

## Summary

Hopefully now you understand and feel more comfortable with the `reset` command, but are probably still a little confused about how exactly it differs from `checkout` and could not possibly remember all the rules of the different invocations.

Here's a cheat-sheet for which commands affect which trees. The "HEAD" column reads "REF" if that command moves the reference (branch) that HEAD points to, and "HEAD" if it moves HEAD itself. Pay especial attention to the *WD Safe?* column — if it says **NO**, take a second to think before running that command.

|  | HEAD | Index | Workdir | WD Safe?  |
|--|------|-------|---------|-----------|
| <b>Commit Level</b>                          |      |       |         |           |
| <code>reset --soft [commit]</code>           | REF  | NO    | NO      | YES       |
| <code>reset [commit]</code>                  | REF  | YES   | NO      | YES       |
| <code>reset --hard [commit]</code>           | REF  | YES   | YES     | <b>NO</b> |
| <code>checkout &lt;commit&gt;</code>         | HEAD | YES   | YES     | YES       |
| <b>File Level</b>                            |      |       |         |           |
| <code>reset [commit] &lt;paths&gt;</code>    | NO   | YES   | NO      | YES       |
| <code>checkout [commit] &lt;paths&gt;</code> | NO   | YES   | YES     | <b>NO</b> |

## Fortgeschrittenes Merging

Merging in Git is typically fairly easy. Since Git makes it easy to merge another branch multiple times, it means that you can have a very long lived branch but you can keep it up to date as you go, solving small conflicts often, rather than be surprised by one enormous conflict at the end of the series.

However, sometimes tricky conflicts do occur. Unlike some other version control systems, Git does not try to be overly clever about merge conflict resolution. Git's philosophy is to be smart about determining when a merge resolution is unambiguous, but if there is a conflict, it does not try to be clever about automatically resolving it. Therefore, if you wait too long to merge two branches that diverge quickly, you can run into some issues.

In this section, we'll go over what some of those issues might be and what tools Git gives you to help handle these more tricky situations. We'll also cover some of the different, non-standard types of merges you can do, as well as see how to back out of merges that you've done.

## Merge Conflicts

While we covered some basics on resolving merge conflicts in [Einfache Merge-Konflikte](#), for more complex conflicts, Git provides a few tools to help you figure out what's going on and how to better deal with the conflict.

First of all, if at all possible, try to make sure your working directory is clean before doing a merge that may have conflicts. If you have work in progress, either commit it to a temporary branch or stash it. This makes it so that you can undo **anything** you try here. If you have unsaved changes in your working directory when you try a merge, some of these tips may help you preserve that work.

Let's walk through a very simple example. We have a super simple Ruby file that prints *hello world*.

```
#!/usr/bin/env ruby

def hello
    puts 'hello world'
end

hello()
```

In our repository, we create a new branch named **whitespace** and proceed to change all the Unix line endings to DOS line endings, essentially changing every line of the file, but just with whitespace. Then we change the line “hello world” to “hello mundo”.

```
$ git checkout -b whitespace
Switched to a new branch 'whitespace'

$ unix2dos hello.rb
unix2dos: converting file hello.rb to DOS format ...
$ git commit -am 'converted hello.rb to DOS'
[whitespace 3270f76] converted hello.rb to DOS
 1 file changed, 7 insertions(+), 7 deletions(-)

$ vim hello.rb
$ git diff -b
diff --git a/hello.rb b/hello.rb
index ac51efd..e85207e 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,7 @@
 #! /usr/bin/env ruby

 def hello
- puts 'hello world'
+ puts 'hello mundo'^M
end

hello()

$ git commit -am 'hello mundo change'
[whitespace 6d338d2] hello mundo change
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Now we switch back to our `master` branch and add some documentation for the function.

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

$ vim hello.rb
$ git diff
diff --git a/hello.rb b/hello.rb
index ac51efd..36c06c8 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,5 +1,6 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
  puts 'hello world'
end

$ git commit -am 'document the function'
[master bec6336] document the function
 1 file changed, 1 insertion(+)
```

Now we try to merge in our `whitespace` branch and we'll get conflicts because of the whitespace changes.

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

## Aborting a Merge

We now have a few options. First, let's cover how to get out of this situation. If you perhaps weren't expecting conflicts and don't want to quite deal with the situation yet, you can simply back out of the merge with `git merge --abort`.

```
$ git status -sb
## master
UU hello.rb

$ git merge --abort

$ git status -sb
## master
```

The `git merge --abort` option tries to revert back to your state before you ran the merge. The only cases where it may not be able to do this perfectly would be if you had unstashed, uncommitted changes in your working directory when you ran it, otherwise it should work fine.

If for some reason you just want to start over, you can also run `git reset --hard HEAD`, and your repository will be back to the last committed state. Remember that any uncommitted work will be lost, so make sure you don't want any of your changes.

## Ignoring Whitespace

In this specific case, the conflicts are whitespace related. We know this because the case is simple, but it's also pretty easy to tell in real cases when looking at the conflict because every line is removed on one side and added again on the other. By default, Git sees all of these lines as being changed, so it can't merge the files.

The default merge strategy can take arguments though, and a few of them are about properly ignoring whitespace changes. If you see that you have a lot of whitespace issues in a merge, you can simply abort it and do it again, this time with `-Xignore-all-space` or `-Xignore-space-change`. The first option ignores whitespace **completely** when comparing lines, the second treats sequences of one or more whitespace characters as equivalent.

```
$ 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(-)
```

Since in this case, the actual file changes were not conflicting, once we ignore the whitespace changes, everything merges just fine.

This is a lifesaver if you have someone on your team who likes to occasionally reformat everything from spaces to tabs or vice-versa.

## Manual File Re-merging

Though Git handles whitespace pre-processing pretty well, there are other types of changes that perhaps Git can't handle automatically, but are scriptable fixes. As an example, let's pretend that Git could not handle the whitespace change and we needed to do it by hand.

What we really need to do is run the file we're trying to merge in through a `dos2unix` program before trying the actual file merge. So how would we do that?

First, we get into the merge conflict state. Then we want to get copies of my version of the file, their version (from the branch we're merging in) and the common version (from where both sides branched off). Then we want to fix up either their side or our side and re-try the merge again for just this single file.

Getting the three file versions is actually pretty easy. Git stores all of these versions in the index under "stages" which each have numbers associated with them. Stage 1 is the common ancestor, stage 2 is your version and stage 3 is from the `MERGE_HEAD`, the version you're merging in ("theirs").

You can extract a copy of each of these versions of the conflicted file with the `git show` command and a special syntax.

```
$ git show :1:hello.rb > hello.common.rb
$ git show :2:hello.rb > hello.ours.rb
$ git show :3:hello.rb > hello.theirs.rb
```

If you want to get a little more hard core, you can also use the `ls-files -u` plumbing command to get the actual SHA-1s of the Git blobs for each of these files.

```
$ git ls-files -u
100755 ac51efdc3df4f4fd328d1a02ad05331d8e2c9111 1  hello.rb
100755 36c06c8752c78d2aff89571132f3bf7841a7b5c3 2  hello.rb
100755 e85207e04dfdd5eb0a1e9febbc67fd837c44a1cd 3  hello.rb
```

The `:1:hello.rb` is just a shorthand for looking up that blob SHA-1.

Now that we have the content of all three stages in our working directory, we can manually fix up theirs to fix the whitespace issue and re-merge the file with the little-known `git merge-file` command which does just that.

```
$ 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()
```

At this point we have nicely merged the file. In fact, this actually works better than the `ignore-space-change` option because this actually fixes the whitespace changes before merge instead of simply ignoring them. In the `ignore-space-change` merge, we actually ended up with a few lines with DOS line endings, making things mixed.

If you want to get an idea before finalizing this commit about what was actually changed between one side or the other, you can ask `git diff` to compare what is in your working directory that

you're about to commit as the result of the merge to any of these stages. Let's go through them all.

To compare your result to what you had in your branch before the merge, in other words, to see what the merge introduced, you can run `git diff --ours`

```
$ git diff --ours
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index 36c06c8..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -2,7 +2,7 @@
 
 # prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()
```

So here we can easily see that what happened in our branch, what we're actually introducing to this file with this merge, is changing that single line.

If we want to see how the result of the merge differed from what was on their side, you can run `git diff --theirs`. In this and the following example, we have to use `-b` to strip out the whitespace because we're comparing it to what is in Git, not our cleaned up `hello.theirs.rb` file.

```
$ 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
```

Finally, you can see how the file has changed from both sides with `git diff --base`.

```
$ git diff --base -b
* Unmerged path hello.rb
diff --git a/hello.rb b/hello.rb
index ac51efd..44d0a25 100755
--- a/hello.rb
+++ b/hello.rb
@@ -1,7 +1,8 @@
 #! /usr/bin/env ruby

+# prints out a greeting
def hello
- puts 'hello world'
+ puts 'hello mundo'
end

hello()
```

At this point we can use the `git clean` command to clear out the extra files we created to do the manual merge but no longer need.

```
$ git clean -f
Removing hello.common.rb
Removing hello.ours.rb
Removing hello.theirs.rb
```

## Checking Out Conflicts

Perhaps we're not happy with the resolution at this point for some reason, or maybe manually editing one or both sides still didn't work well and we need more context.

Let's change up the example a little. For this example, we have two longer lived branches that each have a few commits in them but create a legitimate content conflict when merged.

```
$ git log --graph --oneline --decorate --all
* f1270f7 (HEAD, master) update README
* 9af9d3b add a README
* 694971d update phrase to hola world
| * e3eb223 (mundo) add more tests
| * 7cff591 add testing script
| * c3ffff1 changed text to hello mundo
|
* b7dcc89 initial hello world code
```

We now have three unique commits that live only on the `master` branch and three others that live on the `mundo` branch. If we try to merge the `mundo` branch in, we get a conflict.

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

We would like to see what the merge conflict is. If we open up the file, we'll see something like this:

```
#! /usr/bin/env ruby

def hello
<<<<< HEAD
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>> mundo
end

hello()
```

Both sides of the merge added content to this file, but some of the commits modified the file in the same place that caused this conflict.

Let's explore a couple of tools that you now have at your disposal to determine how this conflict came to be. Perhaps it's not obvious how exactly you should fix this conflict. You need more context.

One helpful tool is `git checkout` with the `--conflict` option. This will re-checkout the file again and replace the merge conflict markers. This can be useful if you want to reset the markers and try to resolve them again.

You can pass `--conflict` either `diff3` or `merge` (which is the default). If you pass it `diff3`, Git will use a slightly different version of conflict markers, not only giving you the “ours” and “theirs” versions, but also the “base” version inline to give you more context.

```
$ git checkout --conflict=diff3 hello.rb
```

Once we run that, the file will look like this instead:

```
#!/usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola world'
||||||| base
  puts 'hello world'
=====
  puts 'hello mundo'
>>>>> theirs
end

hello()
```

If you like this format, you can set it as the default for future merge conflicts by setting the `merge.conflictstyle` setting to `diff3`.

```
$ git config --global merge.conflictstyle diff3
```

The `git checkout` command can also take `--ours` and `--theirs` options, which can be a really fast way of just choosing either one side or the other without merging things at all.

This can be particularly useful for conflicts of binary files where you can simply choose one side, or where you only want to merge certain files in from another branch - you can do the merge and then checkout certain files from one side or the other before committing.

## Merge Log

Another useful tool when resolving merge conflicts is `git log`. This can help you get context on what may have contributed to the conflicts. Reviewing a little bit of history to remember why two lines of development were touching the same area of code can be really helpful sometimes.

To get a full list of all of the unique commits that were included in either branch involved in this merge, we can use the “triple dot” syntax that we learned in [Dreifacher Punkt](#).

```
$ git log --oneline --left-right HEAD...MERGE_HEAD
< f1270f7 update README
< 9af9d3b add a README
< 694971d update phrase to hola world
> e3eb223 add more tests
> 7cff591 add testing script
> c3ffff1 changed text to hello mundo
```

That's a nice list of the six total commits involved, as well as which line of development each commit was on.

We can further simplify this though to give us much more specific context. If we add the `--merge`

option to `git log`, it will only show the commits in either side of the merge that touch a file that's currently conflicted.

```
$ git log --oneline --left-right --merge
< 694971d update phrase to hola world
> c3ffff1 changed text to hello mundo
```

If you run that with the `-p` option instead, you get just the diffs to the file that ended up in conflict. This can be **really** helpful in quickly giving you the context you need to help understand why something conflicts and how to more intelligently resolve it.

## Combined Diff Format

Since Git stages any merge results that are successful, when you run `git diff` while in a conflicted merge state, you only get what is currently still in conflict. This can be helpful to see what you still have to resolve.

When you run `git diff` directly after a merge conflict, it will give you information in a rather unique diff output format.

```
$ 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()
```

The format is called “Combined Diff” and gives you two columns of data next to each line. The first column shows you if that line is different (added or removed) between the “ours” branch and the file in your working directory and the second column does the same between the “theirs” branch and your working directory copy.

So in that example you can see that the `<<<<<` and `>>>>>` lines are in the working copy but were not in either side of the merge. This makes sense because the merge tool stuck them in there for our context, but we're expected to remove them.

If we resolve the conflict and run `git diff` again, we'll see the same thing, but it's a little more

useful.

```
$ 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()
```

This shows us that “hola world” was in our side but not in the working copy, that “hello mundo” was in their side but not in the working copy and finally that “hola mundo” was not in either side but is now in the working copy. This can be useful to review before committing the resolution.

You can also get this from the `git log` for any merge to see how something was resolved after the fact. Git will output this format if you run `git show` on a merge commit, or if you add a `--cc` option to a `git log -p` (which by default only shows patches for non-merge commits).

```
$ 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()
```

## Undoing Merges

Now that you know how to create a merge commit, you'll probably make some by mistake. One of the great things about working with Git is that it's okay to make mistakes, because it's possible (and in many cases easy) to fix them.

Merge commits are no different. Let's say you started work on a topic branch, accidentally merged it into `master`, and now your commit history looks like this:

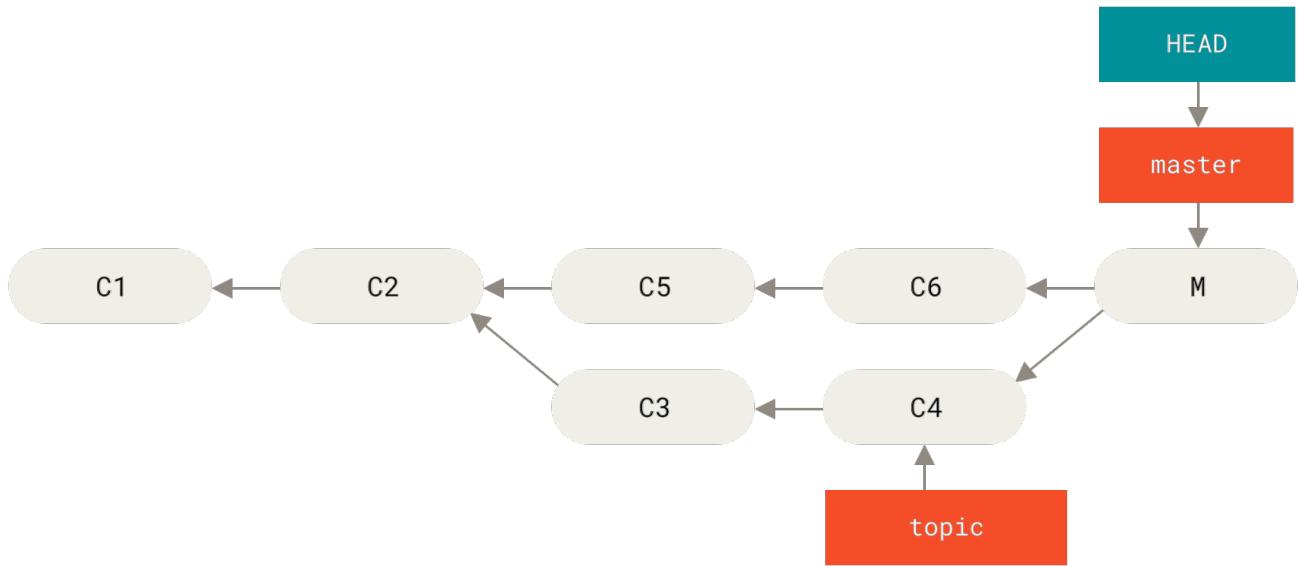


Figure 137. Accidental merge commit

There are two ways to approach this problem, depending on what your desired outcome is.

### Fix the references

If the unwanted merge commit only exists on your local repository, the easiest and best solution is to move the branches so that they point where you want them to. In most cases, if you follow the errant `git merge` with `git reset --hard HEAD~`, this will reset the branch pointers so they look like this:

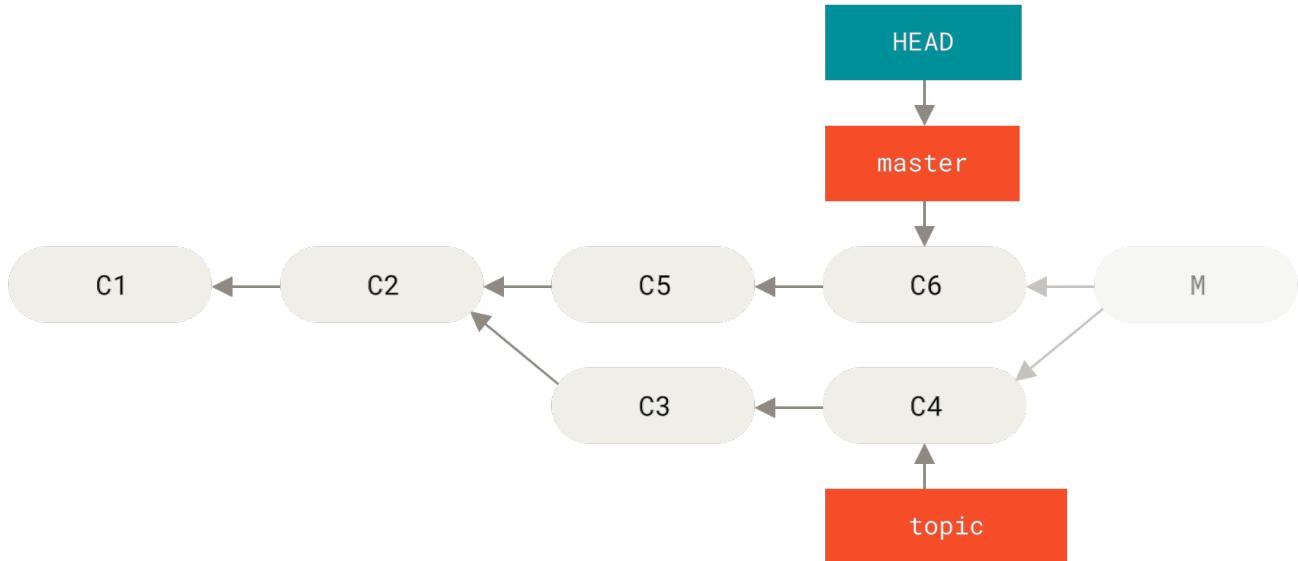


Figure 138. History after `git reset --hard HEAD~`

We covered `reset` back in [Reset Demystified](#), so it shouldn't be too hard to figure out what's going on here. Here's a quick refresher: `reset --hard` usually goes through three steps:

1. Move the branch `HEAD` points to. In this case, we want to move `master` to where it was before the merge commit (`C5`).
2. Make the index look like `HEAD`.

### 3. Make the working directory look like the index.

The downside of this approach is that it's rewriting history, which can be problematic with a shared repository. Check out [Die Gefahren des Rebasing](#) for more on what can happen; the short version is that if other people have the commits you're rewriting, you should probably avoid `reset`. This approach also won't work if any other commits have been created since the merge; moving the refs would effectively lose those changes.

#### Reverse the commit

If moving the branch pointers around isn't going to work for you, Git gives you the option of making a new commit which undoes all the changes from an existing one. Git calls this operation a "revert", and in this particular scenario, you'd invoke it like this:

```
$ git revert -m 1 HEAD
[master b1d8379] Revert "Merge branch 'topic'"
```

The `-m 1` flag indicates which parent is the "mainline" and should be kept. When you invoke a merge into `HEAD` (`git merge topic`), the new commit has two parents: the first one is `HEAD (C6)`, and the second is the tip of the branch being merged in (`C4`). In this case, we want to undo all the changes introduced by merging in parent #2 (`C4`), while keeping all the content from parent #1 (`C6`).

The history with the revert commit looks like this:

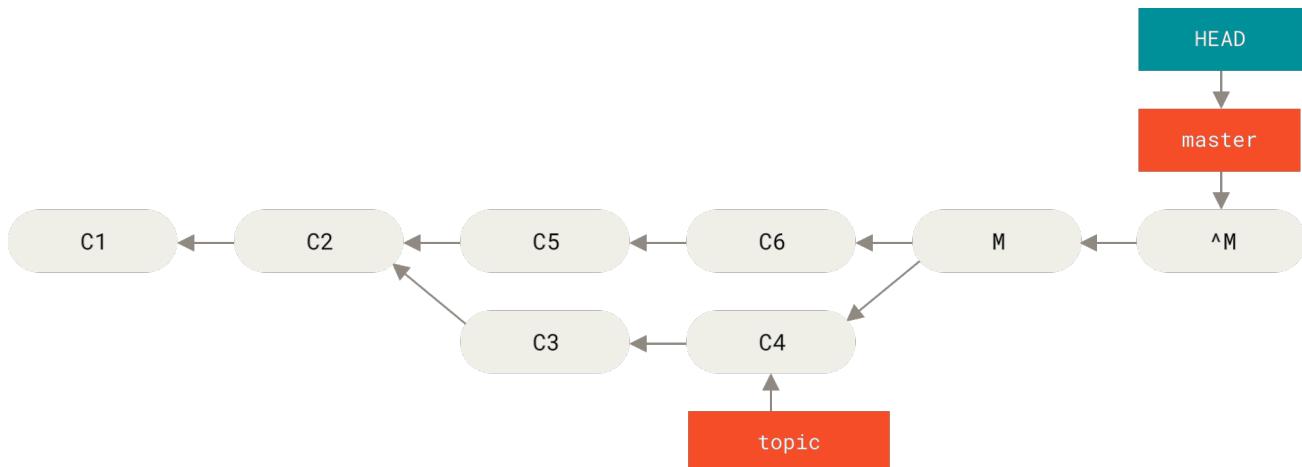


Figure 139. History after `git revert -m 1`

The new commit `^M` has exactly the same contents as `C6`, so starting from here it's as if the merge never happened, except that the now-unmerged commits are still in `HEAD`'s history. Git will get confused if you try to merge `topic` into `master` again:

```
$ git merge topic
Already up-to-date.
```

There's nothing in `topic` that isn't already reachable from `master`. What's worse, if you add work to `topic` and merge again, Git will only bring in the changes *since* the reverted merge:

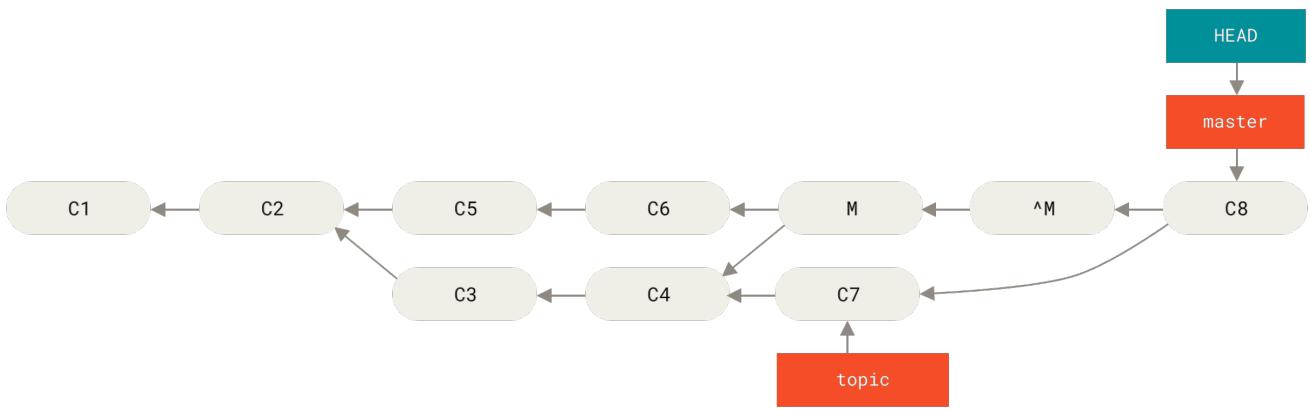


Figure 140. History with a bad merge

The best way around this is to un-revert the original merge, since now you want to bring in the changes that were reverted out, **then** create a new merge commit:

```
$ git revert ^M
[master 09f0126] Revert "Revert "Merge branch 'topic'''"
$ git merge topic
```

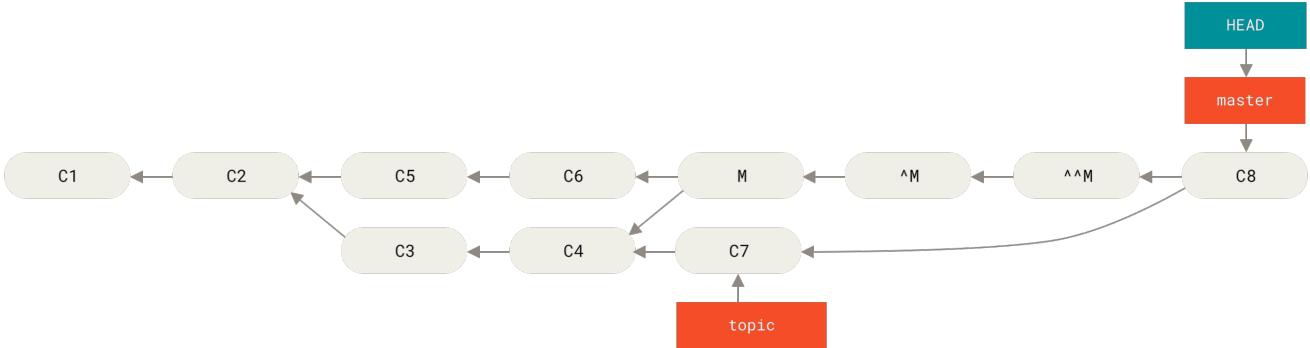


Figure 141. History after re-merging a reverted merge

In this example, **M** and **^M** cancel out. **^^M** effectively merges in the changes from **C3** and **C4**, and **C8** merges in the changes from **C7**, so now **topic** is fully merged.

## Other Types of Merges

So far we've covered the normal merge of two branches, normally handled with what is called the “recursive” strategy of merging. There are other ways to merge branches together however. Let's cover a few of them quickly.

### Our or Theirs Preference

First of all, there is another useful thing we can do with the normal “recursive” mode of merging. We've already seen the **ignore-all-space** and **ignore-space-change** options which are passed with a **-X** but we can also tell Git to favor one side or the other when it sees a conflict.

By default, when Git sees a conflict between two branches being merged, it will add merge conflict markers into your code and mark the file as conflicted and let you resolve it. If you would prefer for Git to simply choose a specific side and ignore the other side instead of letting you manually

resolve the conflict, you can pass the `merge` command either a `-Xours` or `-Xtheirs`.

If Git sees this, it will not add conflict markers. Any differences that are mergeable, it will merge. Any differences that conflict, it will simply choose the side you specify in whole, including binary files.

If we go back to the “hello world” example we were using before, we can see that merging in our branch causes conflicts.

```
$ 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.
```

However if we run it with `-Xours` or `-Xtheirs` it does not.

```
$ git merge -Xours mundo
Auto-merging hello.rb
Merge made by the 'recursive' strategy.
hello.rb | 2 ++
test.sh  | 2 ++
2 files changed, 3 insertions(+), 1 deletion(-)
create mode 100644 test.sh
```

In that case, instead of getting conflict markers in the file with “hello mundo” on one side and “hola world” on the other, it will simply pick “hola world”. However, all the other non-conflicting changes on that branch are merged successfully in.

This option can also be passed to the `git merge-file` command we saw earlier by running something like `git merge-file --ours` for individual file merges.

If you want to do something like this but not have Git even try to merge changes from the other side in, there is a more draconian option, which is the “ours” merge *strategy*. This is different from the “ours” recursive merge *option*.

This will basically do a fake merge. It will record a new merge commit with both branches as parents, but it will not even look at the branch you’re merging in. It will simply record as the result of the merge the exact code in your current branch.

```
$ git merge -s ours mundo
Merge made by the 'ours' strategy.
$ git diff HEAD HEAD~
$
```

You can see that there is no difference between the branch we were on and the result of the merge.

This can often be useful to basically trick Git into thinking that a branch is already merged when doing a merge later on. For example, say you branched off a `release` branch and have done some work on it that you will want to merge back into your `master` branch at some point. In the meantime some bugfix on `master` needs to be backported into your `release` branch. You can merge the bugfix branch into the `release` branch and also `merge -s ours` the same branch into your `master` branch (even though the fix is already there) so when you later merge the `release` branch again, there are no conflicts from the bugfix.

## Subtree Merging

The idea of the subtree merge is that you have two projects, and one of the projects maps to a subdirectory of the other one. When you specify a subtree merge, Git is often smart enough to figure out that one is a subtree of the other and merge appropriately.

We'll go through an example of adding a separate project into an existing project and then merging the code of the second into a subdirectory of the first.

First, we'll add the Rack application to our project. We'll add the Rack project as a remote reference in our own project and then check it out into its own branch:

```
$ git remote add rack_remote https://github.com/rack/rack
$ git fetch rack_remote --no-tags
warning: no common commits
remote: Counting objects: 3184, done.
remote: Compressing objects: 100% (1465/1465), done.
remote: Total 3184 (delta 1952), reused 2770 (delta 1675)
Receiving objects: 100% (3184/3184), 677.42 KiB | 4 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
From https://github.com/rack/rack
 * [new branch]      build      -> rack_remote/build
 * [new branch]      master     -> rack_remote/master
 * [new branch]      rack-0.4   -> rack_remote/rack-0.4
 * [new branch]      rack-0.9   -> rack_remote/rack-0.9
$ git checkout -b rack_branch rack_remote/master
Branch rack_branch set up to track remote branch refs/remotes/rack_remote/master.
Switched to a new branch "rack_branch"
```

Now we have the root of the Rack project in our `rack_branch` branch and our own project in the `master` branch. If you check out one and then the other, you can see that they have different project roots:

```
$ ls
AUTHORS      KNOWN-ISSUES  Rakefile      contrib      lib
COPYING      README        bin          example      test
$ git checkout master
Switched to branch "master"
$ ls
README
```

This is sort of a strange concept. Not all the branches in your repository actually have to be branches of the same project. It's not common, because it's rarely helpful, but it's fairly easy to have branches contain completely different histories.

In this case, we want to pull the Rack project into our `master` project as a subdirectory. We can do that in Git with `git read-tree`. You'll learn more about `read-tree` and its friends in [Git Interna](#), but for now know that it reads the root tree of one branch into your current staging area and working directory. We just switched back to your `master` branch, and we pull the `rack_branch` branch into the `rack` subdirectory of our `master` branch of our main project:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

When we commit, it looks like we have all the Rack files under that subdirectory – as though we copied them in from a tarball. What gets interesting is that we can fairly easily merge changes from one of the branches to the other. So, if the Rack project updates, we can pull in upstream changes by switching to that branch and pulling:

```
$ git checkout rack_branch
$ git pull
```

Then, we can merge those changes back into our `master` branch. To pull in the changes and prepopulate the commit message, use the `--squash` option, as well as the recursive merge strategy's `-Xsubtree` option. (The recursive strategy is the default here, but we include it for clarity.)

```
$ git checkout master
$ git merge --squash -s recursive -Xsubtree=rack rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

All the changes from the Rack project are merged in and ready to be committed locally. You can also do the opposite – make changes in the `rack` subdirectory of your `master` branch and then merge them into your `rack_branch` branch later to submit them to the maintainers or push them upstream.

This gives us a way to have a workflow somewhat similar to the submodule workflow without using submodules (which we will cover in [Submodules](#)). We can keep branches with other related projects in our repository and subtree merge them into our project occasionally. It is nice in some ways, for example all the code is committed to a single place. However, it has other drawbacks in that it's a bit more complex and easier to make mistakes in reintegrating changes or accidentally pushing a branch into an unrelated repository.

Another slightly weird thing is that to get a diff between what you have in your `rack` subdirectory and the code in your `rack_branch` branch – to see if you need to merge them – you can't use the normal `diff` command. Instead, you must run `git diff-tree` with the branch you want to compare to:

```
$ git diff-tree -p rack_branch
```

Or, to compare what is in your `rack` subdirectory with what the `master` branch on the server was the last time you fetched, you can run

```
$ git diff-tree -p rack_remote/master
```

## Rerere

The `git rerere` functionality is a bit of a hidden feature. The name stands for “reuse recorded resolution” and, as the name implies, it allows you to ask Git to remember how you’ve resolved a hunk conflict so that the next time it sees the same conflict, Git can resolve it for you automatically.

There are a number of scenarios in which this functionality might be really handy. One of the examples that is mentioned in the documentation is when you want to make sure a long-lived topic branch will ultimately merge cleanly, but you don’t want to have a bunch of intermediate merge commits cluttering up your commit history. With `rerere` enabled, you can attempt the occasional merge, resolve the conflicts, then back out of the merge. If you do this continuously, then the final merge should be easy because `rerere` can just do everything for you automatically.

This same tactic can be used if you want to keep a branch rebased so you don’t have to deal with the same rebasing conflicts each time you do it. Or if you want to take a branch that you merged and fixed a bunch of conflicts and then decide to rebase it instead—you likely won’t have to do all the same conflicts again.

Another application of `rerere` is where you merge a bunch of evolving topic branches together into a testable head occasionally, as the Git project itself often does. If the tests fail, you can rewind the merges and re-do them without the topic branch that made the tests fail without having to re-resolve the conflicts again.

To enable `rerere` functionality, you simply have to run this config setting:

```
$ git config --global rerere.enabled true
```

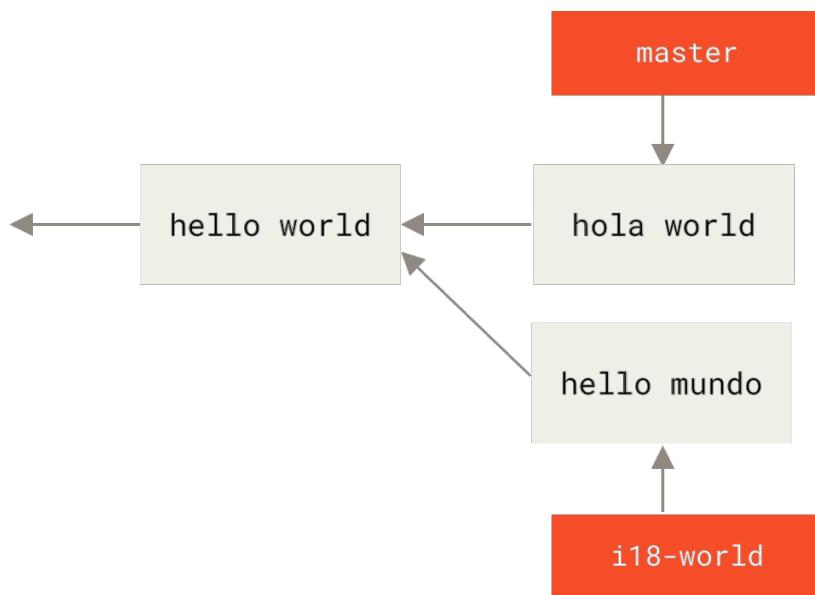
You can also turn it on by creating the `.git/rr-cache` directory in a specific repository, but the config setting is clearer and enables that feature globally for you.

Now let’s see a simple example, similar to our previous one. Let’s say we have a file named `hello.rb` that looks like this:

```
#!/usr/bin/env ruby

def hello
  puts 'hello world'
end
```

In one branch we change the word “hello” to “hola”, then in another branch we change the “world” to “mundo”, just like before.



When we merge the two branches together, we'll get a merge conflict:

```
$ 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.
```

You should notice the new line `Recorded preimage for FILE` in there. Otherwise it should look exactly like a normal merge conflict. At this point, `rerere` can tell us a few things. Normally, you might run `git status` at this point to see what all conflicted:

```
$ 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
#
```

However, `git rerere` will also tell you what it has recorded the pre-merge state for with `git rerere status`:

```
$ git rerere status
hello.rb
```

And `git rerere diff` will show the current state of the resolution—what you started with to resolve and what you've resolved it to.

```
$ 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
```

Also (and this isn't really related to `rerere`), you can use `git ls-files -u` to see the conflicted files and the before, left and right versions:

```
$ git ls-files -u
100644 39804c942a9c1f2c03dc7c5ebcd7f3e3a6b97519 1  hello.rb
100644 a440db6e8d1fd76ad438a49025a9ad9ce746f581 2  hello.rb
100644 54336ba847c3758ab604876419607e9443848474 3  hello.rb
```

Now you can resolve it to just be `puts 'hola mundo'` and you can run `git rerere diff` again to see what `rerere` will remember:

```
$ git rerere diff
--- a/hello.rb
+++ b/hello.rb
@@ -1,11 +1,7 @@
 #! /usr/bin/env ruby

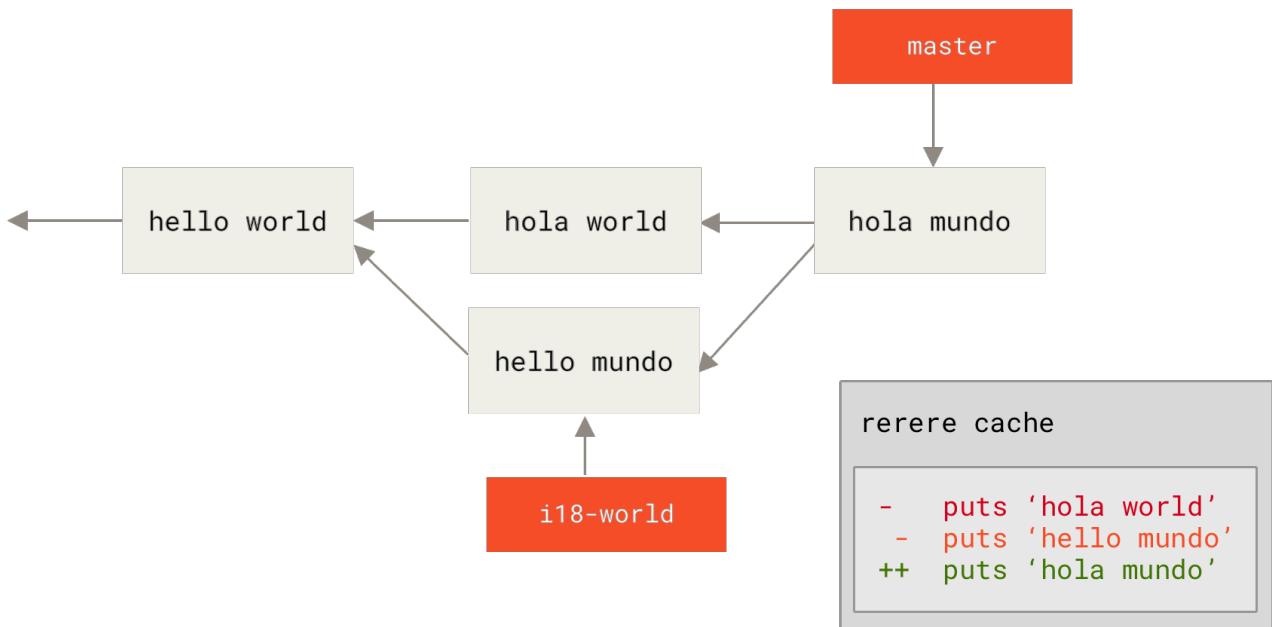
def hello
-<<<<<
- puts 'hello mundo'
-=====
- puts 'hola world'
->>>>>
+ puts 'hola mundo'
end
```

So that basically says, when Git sees a hunk conflict in a `hello.rb` file that has “hello mundo” on one side and “hola mundo” on the other, it will resolve it to “hola mundo”.

Now we can mark it as resolved and commit it:

```
$ git add hello.rb
$ git commit
Recorded resolution for 'hello.rb'.
[master 68e16e5] Merge branch 'i18n'
```

You can see that it "Recorded resolution for FILE".



Now, let's undo that merge and then rebase it on top of our master branch instead. We can move our branch back by using `git reset` as we saw in [Reset Demystified](#).

```
$ git reset --hard HEAD^
HEAD is now at ad63f15 i18n the hello
```

Our merge is undone. Now let's rebase the topic branch.

```
$ 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
```

Now, we got the same merge conflict like we expected, but take a look at the [Resolved FILE using previous resolution](#) line. If we look at the file, we'll see that it's already been resolved, there are no merge conflict markers in it.

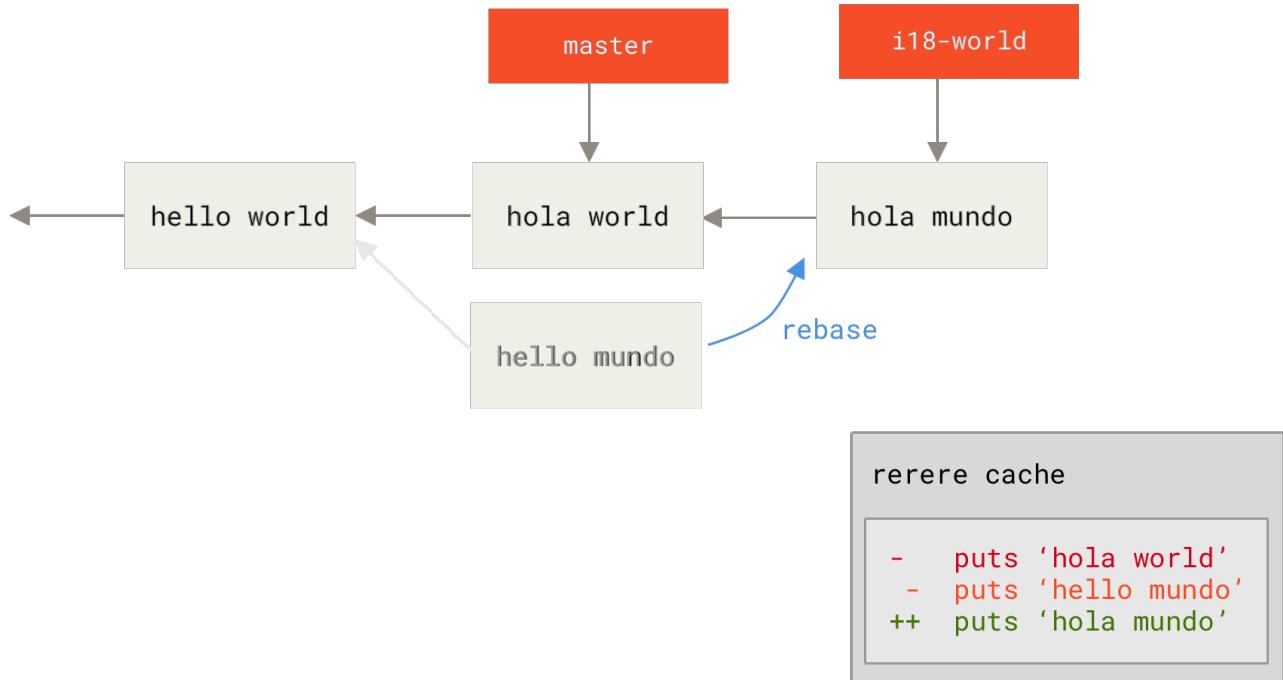
```
#!/usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

Also, [git diff](#) will show you how it was automatically re-resolved:

```
$ 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
```



You can also recreate the conflicted file state with `git checkout`:

```
$ git checkout --conflict=merge hello.rb
$ cat hello.rb
#!/usr/bin/env ruby

def hello
<<<<< ours
  puts 'hola world'
=====
  puts 'hello mundo'
>>>>> theirs
end
```

We saw an example of this in [Fortgeschrittenes Merging](#). For now though, let's re-resolve it by just running `git rerere` again:

```
$ git rerere
Resolved 'hello.rb' using previous resolution.
$ cat hello.rb
#!/usr/bin/env ruby

def hello
  puts 'hola mundo'
end
```

We have re-resolved the file automatically using the `rerere` cached resolution. You can now add and continue the rebase to complete it.

```
$ git add hello.rb
$ git rebase --continue
Applying: i18n one word
```

So, if you do a lot of re-merges, or want to keep a topic branch up to date with your master branch without a ton of merges, or you rebase often, you can turn on `rerere` to help your life out a bit.

## Debugging with Git

In addition to being primarily for version control, Git also provides a couple commands to help you debug your source code projects. Because Git is designed to handle nearly any type of content, these tools are pretty generic, but they can often help you hunt for a bug or culprit when things go wrong.

### File Annotation

If you track down a bug in your code and want to know when it was introduced and why, file annotation is often your best tool. It shows you what commit was the last to modify each line of any file. So if you see that a method in your code is buggy, you can annotate the file with `git blame` to determine which commit was responsible for the introduction of that line.

The following example uses `git blame` to determine which commit and committer was responsible for lines in the top-level Linux kernel `Makefile` and, further, uses the `-L` option to restrict the output of the annotation to lines 69 through 82 of that file:

```
$ 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
```

Notice that the first field is the partial SHA-1 of the commit that last modified that line. The next two fields are values extracted from that commit—the author name and the authored date of that commit—so you can easily see who modified that line and when. After that come the line number and the content of the file. Also note the `^1da177e4c3f4` commit lines, where the `^` prefix designates lines that were introduced in the repository's initial commit and have remained unchanged ever since. This is a tad confusing, because now you've seen at least three different ways that Git uses

the ^ to modify a commit SHA-1, but that is what it means here.

Another cool thing about Git is that it doesn't track file renames explicitly. It records the snapshots and then tries to figure out what was renamed implicitly, after the fact. One of the interesting features of this is that you can ask it to figure out all sorts of code movement as well. If you pass -C to `git blame`, Git analyzes the file you're annotating and tries to figure out where snippets of code within it originally came from if they were copied from elsewhere. For example, say you are refactoring a file named `GITServerHandler.m` into multiple files, one of which is `GITPackUpload.m`. By blaming `GITPackUpload.m` with the -C option, you can see where sections of the code originally came from:

```
$ 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)
```

This is really useful. Normally, you get as the original commit the commit where you copied the code over, because that is the first time you touched those lines in this file. Git tells you the original commit where you wrote those lines, even if it was in another file.

## Binary Search

Annotating a file helps if you know where the issue is to begin with. If you don't know what is breaking, and there have been dozens or hundreds of commits since the last state where you know the code worked, you'll likely turn to `git bisect` for help. The `bisect` command does a binary search through your commit history to help you identify as quickly as possible which commit introduced an issue.

Let's say you just pushed out a release of your code to a production environment, you're getting bug reports about something that wasn't happening in your development environment, and you can't imagine why the code is doing that. You go back to your code, and it turns out you can reproduce the issue, but you can't figure out what is going wrong. You can `bisect` the code to find out. First you run `git bisect start` to get things going, and then you use `git bisect bad` to tell the system that the current commit you're on is broken. Then, you must tell bisect when the last known good state was, using `git bisect good <good_commit>`:

```
$ git bisect start
$ git bisect bad
$ git bisect good v1.0
Bisecting: 6 revisions left to test after this
[ecb6e1bc347ccecc5f9350d878ce677feb13d3b2] error handling on repo
```

Git figured out that about 12 commits came between the commit you marked as the last good commit (v1.0) and the current bad version, and it checked out the middle one for you. At this point, you can run your test to see if the issue exists as of this commit. If it does, then it was introduced sometime before this middle commit; if it doesn't, then the problem was introduced sometime after the middle commit. It turns out there is no issue here, and you tell Git that by typing `git bisect good` and continue your journey:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Now you're on another commit, halfway between the one you just tested and your bad commit. You run your test again and find that this commit is broken, so you tell Git that with `git bisect bad`:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

This commit is fine, and now Git has all the information it needs to determine where the issue was introduced. It tells you the SHA-1 of the first bad commit and show some of the commit information and which files were modified in that commit so you can figure out what happened that may have introduced this bug:

```
$ 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
```

When you're finished, you should run `git bisect reset` to reset your HEAD to where you were before you started, or you'll end up in a weird state:

```
$ git bisect reset
```

This is a powerful tool that can help you check hundreds of commits for an introduced bug in minutes. In fact, if you have a script that will exit 0 if the project is good or non-0 if the project is bad, you can fully automate `git bisect`. First, you again tell it the scope of the bisect by providing the known bad and good commits. You can do this by listing them with the `bisect start` command if you want, listing the known bad commit first and the known good commit second:

```
$ git bisect start HEAD v1.0  
$ git bisect run test-error.sh
```

Doing so automatically runs `test-error.sh` on each checked-out commit until Git finds the first broken commit. You can also run something like `make` or `make tests` or whatever you have that runs automated tests for you.

## Submodules

It often happens that while working on one project, you need to use another project from within it. Perhaps it's a library that a third party developed or that you're developing separately and using in multiple parent projects. A common issue arises in these scenarios: you want to be able to treat the two projects as separate yet still be able to use one from within the other.

Here's an example. Suppose you're developing a website and creating Atom feeds. Instead of writing your own Atom-generating code, you decide to use a library. You're likely to have to either include this code from a shared library like a CPAN install or Ruby gem, or copy the source code into your own project tree. The issue with including the library is that it's difficult to customize the library in any way and often more difficult to deploy it, because you need to make sure every client has that library available. The issue with copying the code into your own project is that any custom changes you make are difficult to merge when upstream changes become available.

Git addresses this issue using submodules. Submodules allow you to keep a Git repository as a subdirectory of another Git repository. This lets you clone another repository into your project and keep your commits separate.

### Starting with Submodules

We'll walk through developing a simple project that has been split up into a main project and a few sub-projects.

Let's start by adding an existing Git repository as a submodule of the repository that we're working on. To add a new submodule you use the `git submodule add` command with the absolute or relative URL of the project you would like to start tracking. In this example, we'll add a library called "DbConnector".

```
$ git submodule add https://github.com/chaconinc/DbConnector
Cloning into 'DbConnector'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 0), reused 11 (delta 0)
Unpacking objects: 100% (11/11), done.
Checking connectivity... done.
```

By default, submodules will add the subproject into a directory named the same as the repository, in this case “DbConnector”. You can add a different path at the end of the command if you want it to go elsewhere.

If you run `git status` at this point, you’ll notice a few things.

```
$ 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
```

First you should notice the new `.gitmodules` file. This is a configuration file that stores the mapping between the project’s URL and the local subdirectory you’ve pulled it into:

```
[submodule "DbConnector"]
  path = DbConnector
  url = https://github.com/chaconinc/DbConnector
```

If you have multiple submodules, you’ll have multiple entries in this file. It’s important to note that this file is version-controlled with your other files, like your `.gitignore` file. It’s pushed and pulled with the rest of your project. This is how other people who clone this project know where to get the submodule projects from.



Since the URL in the `.gitmodules` file is what other people will first try to clone/fetch from, make sure to use a URL that they can access if possible. For example, if you use a different URL to push to than others would pull from, use the one that others have access to. You can overwrite this value locally with `git config submodule.DbConnector.url PRIVATE_URL` for your own use. When applicable, a relative URL can be helpful.

The other listing in the `git status` output is the project folder entry. If you run `git diff` on that, you see something interesting:

```
$ 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
```

Although `DbConnector` is a subdirectory in your working directory, Git sees it as a submodule and doesn't track its contents when you're not in that directory. Instead, Git sees it as a particular commit from that repository.

If you want a little nicer diff output, you can pass the `--submodule` option to `git diff`.

```
$ git diff --cached --submodule
diff --git a/.gitmodules b/.gitmodules
new file mode 100644
index 0000000..71fc376
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "DbConnector"]
+    path = DbConnector
+    url = https://github.com/chaconinc/DbConnector
Submodule DbConnector 0000000...c3f01dc (new submodule)
```

When you commit, you see something like this:

```
$ git commit -am 'added DbConnector module'
[master fb9093c] added DbConnector module
 2 files changed, 4 insertions(+)
 create mode 100644 .gitmodules
 create mode 160000 DbConnector
```

Notice the `160000` mode for the `DbConnector` entry. That is a special mode in Git that basically means you're recording a commit as a directory entry rather than a subdirectory or a file.

Lastly, push these changes:

```
$ git push origin master
```

## Cloning a Project with Submodules

Here we'll clone a project with a submodule in it. When you clone such a project, by default you get the directories that contain submodules, but none of the files within them yet:

```

$ 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
$
```

The `DbConnector` directory is there, but empty. You must run two commands: `git submodule init` to initialize your local configuration file, and `git submodule update` to fetch all the data from that project and check out the appropriate commit listed in your superproject:

```

$ 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'
```

Now your `DbConnector` subdirectory is at the exact state it was in when you committed earlier.

There is another way to do this which is a little simpler, however. If you pass `--recurse-submodules` to the `git clone` command, it will automatically initialize and update each submodule in the repository, including nested submodules if any of the submodules in the repository have submodules themselves.

```
$ 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'
```

If you already cloned the project and forgot `--recurse-submodules`, you can combine the `git submodule init` and `git submodule update` steps by running `git submodule update --init`. To also initialize, fetch and checkout any nested submodules, you can use the foolproof `git submodule update --init --recursive`.

## Working on a Project with Submodules

Now we have a copy of a project with submodules in it and will collaborate with our teammates on both the main project and the submodule project.

### Pulling in Upstream Changes from the Submodule Remote

The simplest model of using submodules in a project would be if you were simply consuming a subproject and wanted to get updates from it from time to time but were not actually modifying anything in your checkout. Let's walk through a simple example there.

If you want to check for new work in a submodule, you can go into the directory and run `git fetch` and `git merge` the upstream branch to update the local code.

```
$ 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(+)
```

Now if you go back into the main project and run `git diff --submodule` you can see that the submodule was updated and get a list of commits that were added to it. If you don't want to type

--submodule every time you run `git diff`, you can set it as the default format by setting the `diff.submodule` config value to “log”.

```
$ git config --global diff.submodule log
$ git diff
Submodule DbConnector c3f01dc..d0354fc:
  > more efficient db routine
  > better connection routine
```

If you commit at this point then you will lock the submodule into having the new code when other people update.

There is an easier way to do this as well, if you prefer to not manually fetch and merge in the subdirectory. If you run `git submodule update --remote`, Git will go into your submodules and fetch and update for you.

```
$ 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'
```

This command will by default assume that you want to update the checkout to the `master` branch of the submodule repository. You can, however, set this to something different if you want. For example, if you want to have the `DbConnector` submodule track that repository’s “stable” branch, you can set it in either your `.gitmodules` file (so everyone else also tracks it), or just in your local `.git/config` file. Let’s set it in the `.gitmodules` file:

```
$ 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'
```

If you leave off the `-f .gitmodules` it will only make the change for you, but it probably makes more sense to track that information with the repository so everyone else does as well.

When we run `git status` at this point, Git will show us that we have “new commits” on the submodule.

```
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

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

    modified:   .gitmodules
    modified:   DbConnector (new commits)

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

If you set the configuration setting `status.submodulesummary`, Git will also show you a short summary of changes to your submodules:

```
$ 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
```

At this point if you run `git diff` we can see both that we have modified our `.gitmodules` file and also that there are a number of commits that we've pulled down and are ready to commit to our submodule project.

```
$ git diff
diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
     path = DbConnector
     url = https://github.com/chaconinc/DbConnector
+    branch = stable
Submodule DbConnector c3f01dc..c87d55d:
> catch non-null terminated lines
> more robust error handling
> more efficient db routine
> better connection routine
```

This is pretty cool as we can actually see the log of commits that we're about to commit to in our submodule. Once committed, you can see this information after the fact as well when you run `git log -p`.

```
$ git log -p --submodule
commit 0a24cfc121a8a3c118e0105ae4ae4c00281cf7ae
Author: Scott Chacon <schacon@gmail.com>
Date:   Wed Sep 17 16:37:02 2014 +0200

    updating DbConnector for bug fixes

diff --git a/.gitmodules b/.gitmodules
index 6fc0b3d..fd1cc29 100644
--- a/.gitmodules
+++ b/.gitmodules
@@ -1,3 +1,4 @@
 [submodule "DbConnector"]
     path = DbConnector
     url = https://github.com/chaconinc/DbConnector
+    branch = stable
Submodule DbConnector c3f01dc..c87d55d:
> catch non-null terminated lines
> more robust error handling
> more efficient db routine
> better connection routine
```

Git will by default try to update **all** of your submodules when you run `git submodule update --remote` so if you have a lot of them, you may want to pass the name of just the submodule you want to try to update.

## Pulling Upstream Changes from the Project Remote

Let's now step into the shoes of your collaborator, who has his own local clone of the MainProject repository. Simply executing `git pull` to get your newly committed changes is not enough:

```
$ git pull
From https://github.com/chaconinc/MainProject
  fb9093c..0a24cf  master      -> origin/master
Fetching submodule DbConnector
From https://github.com/chaconinc/DbConnector
  c3f01dc..c87d55d  stable      -> origin/stable
Updating fb9093c..0a24cf
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")
```

By default, the `'git pull'` command recursively fetches submodules changes, as we can see in the output of the first command above.

However, it does not **\*update\*** the submodules.

This is shown by the output of the `'git status'` command, which shows the submodule is **“modified”**, and has **“new commits”**.

What's more, the brackets showing the new commits point left (<), indicating that these commits are recorded in MainProject but are not present in the local DbConnector checkout.

To finalize the update, you need to run `'git submodule update'`:

```
$ git submodule update --init --recursive
Submodule path 'vendor/plugins/demo': checked out
'48679c6302815f6c76f1fe30625d795d9e55fc56'

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

Note that to be on the safe side, you should run `git submodule update` with the `--init` flag in case the MainProject commits you just pulled added new submodules, and with the `--recursive` flag if any submodules have nested submodules.

If you want to automate this process, you can add the `--recurse-submodules` flag to the `git pull` command (since Git 2.14). This will make Git run `git submodule update` right after the pull, putting the submodules in the correct state. Moreover, if you want to make Git always pull with `--recurse-submodules`, you can set the configuration option `submodule.recurse` to true (this works for `git pull` since Git 2.15). This option will make Git use the `--recurse-submodules` flag for all commands that support it (except `clone`).

## Working on a Submodule

It's quite likely that if you're using submodules, you're doing so because you really want to work on the code in the submodule at the same time as you're working on the code in the main project (or across several submodules). Otherwise you would probably instead be using a simpler dependency management system (such as Maven or Rubygems).

So now let's go through an example of making changes to the submodule at the same time as the main project and committing and publishing those changes at the same time.

So far, when we've run the `git submodule update` command to fetch changes from the submodule repositories, Git would get the changes and update the files in the subdirectory but will leave the sub-repository in what's called a "detached HEAD" state. This means that there is no local working branch (like "master", for example) tracking changes. With no working branch tracking changes, that means even if you commit changes to the submodule, those changes will quite possibly be lost the next time you run `git submodule update`. You have to do some extra steps if you want changes in a submodule to be tracked.

In order to set up your submodule to be easier to go in and hack on, you need to do two things. You need to go into each submodule and check out a branch to work on. Then you need to tell Git what to do if you have made changes and then `git submodule update --remote` pulls in new work from upstream. The options are that you can merge them into your local work, or you can try to rebase your local work on top of the new changes.

First of all, let's go into our submodule directory and check out a branch.

```
$ cd DbConnector/  
$ git checkout stable  
Switched to branch 'stable'
```

Let's try updating our submodule with the "merge" option. To specify it manually, we can just add the `--merge` option to our `update` call. Here we'll see that there was a change on the server for this submodule and it gets merged in.

```
$ 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'
```

If we go into the `DbConnector` directory, we have the new changes already merged into our local `stable` branch. Now let's see what happens when we make our own local change to the library and someone else pushes another change upstream at the same time.

```
$ cd DbConnector/  
$ vim src/db.c  
$ git commit -am 'unicode support'  
[stable f906e16] unicode support  
 1 file changed, 1 insertion(+)
```

Now if we update our submodule we can see what happens when we have made a local change and upstream also has a change we need to incorporate.

```
$ 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'
```

If you forget the `--rebase` or `--merge`, Git will just update the submodule to whatever is on the server and reset your project to a detached HEAD state.

```
$ git submodule update --remote  
Submodule path 'DbConnector': checked out '5d60ef9bbebf5a0c1c1050f242ceeb54ad58da94'
```

If this happens, don't worry, you can simply go back into the directory and check out your branch again (which will still contain your work) and merge or rebase `origin/stable` (or whatever remote branch you want) manually.

If you haven't committed your changes in your submodule and you run a submodule update that would cause issues, Git will fetch the changes but not overwrite unsaved work in your submodule directory.

```
$ 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'
```

If you made changes that conflict with something changed upstream, Git will let you know when you run the update.

```
$ 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'
```

You can go into the submodule directory and fix the conflict just as you normally would.

## Publishing Submodule Changes

Now we have some changes in our submodule directory. Some of these were brought in from upstream by our updates and others were made locally and aren't available to anyone else yet as we haven't pushed them yet.

```
$ git diff
Submodule DbConnector c87d55d..82d2ad3:
> Merge from origin/stable
> updated setup script
> unicode support
> remove unnecessary method
> add new option for conn pooling
```

If we commit in the main project and push it up without pushing the submodule changes up as well, other people who try to check out our changes are going to be in trouble since they will have no way to get the submodule changes that are depended on. Those changes will only exist on our local copy.

In order to make sure this doesn't happen, you can ask Git to check that all your submodules have been pushed properly before pushing the main project. The `git push` command takes the `--recurse-submodules` argument which can be set to either "check" or "on-demand". The "check" option will make `push` simply fail if any of the committed submodule changes haven't been pushed.

```
$ 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.

As you can see, it also gives us some helpful advice on what we might want to do next. The simple option is to go into each submodule and manually push to the remotes to make sure they're externally available and then try this push again. If you want the check behavior to happen for all pushes, you can make this behavior the default by doing `git config push.recurseSubmodules check`.

The other option is to use the "on-demand" value, which will try to do this for you.

```
$ 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
```

As you can see there, Git went into the DbConnector module and pushed it before pushing the main project. If that submodule push fails for some reason, the main project push will also fail. You can make this behavior the default by doing `git config push.recurseSubmodules on-demand`.

## Merging Submodule Changes

If you change a submodule reference at the same time as someone else, you may run into some problems. That is, if the submodule histories have diverged and are committed to diverging branches in a superproject, it may take a bit of work for you to fix.

If one of the commits is a direct ancestor of the other (a fast-forward merge), then Git will simply choose the latter for the merge, so that works fine.

Git will not attempt even a trivial merge for you, however. If the submodule commits diverge and need to be merged, you will get something that looks like this:

```
$ 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.
```

So basically what has happened here is that Git has figured out that the two branches record points in the submodule's history that are divergent and need to be merged. It explains it as "merge

following commits not found”, which is confusing but we’ll explain why that is in a bit.

To solve the problem, you need to figure out what state the submodule should be in. Strangely, Git doesn’t really give you much information to help out here, not even the SHA-1s of the commits of both sides of the history. Fortunately, it’s simple to figure out. If you run `git diff` you can get the SHA-1s of the commits recorded in both branches you were trying to merge.

```
$ git diff  
diff --cc DbConnector  
index eb41d76,c771610..0000000  
--- a/DbConnector  
+++ b/DbConnector
```

So, in this case, `eb41d76` is the commit in our submodule that we had and `c771610` is the commit that upstream had. If we go into our submodule directory, it should already be on `eb41d76` as the merge would not have touched it. If for whatever reason it’s not, you can simply create and checkout a branch pointing to it.

What is important is the SHA-1 of the commit from the other side. This is what you’ll have to merge in and resolve. You can either just try the merge with the SHA-1 directly, or you can create a branch for it and then try to merge that in. We would suggest the latter, even if only to make a nicer merge commit message.

So, we will go into our submodule directory, create a branch based on that second SHA-1 from `git diff` and manually merge.

```
$ cd DbConnector  
  
$ git rev-parse HEAD  
eb41d764bccf88be77aced643c13a7fa86714135  
  
$ git branch try-merge c771610  
(DbConnector) $ git merge try-merge  
Auto-merging src/main.c  
CONFLICT (content): Merge conflict in src/main.c  
Recorded preimage for 'src/main.c'  
Automatic merge failed; fix conflicts and then commit the result.
```

We got an actual merge conflict here, so if we resolve that and commit it, then we can simply update the main project with the result.

```

$ 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 First we resolve the conflict

② Then we go back to the main project directory

③ We can check the SHA-1s again

④ Resolve the conflicted submodule entry

⑤ Commit our merge

It can be a bit confusing, but it's really not very hard.

Interestingly, there is another case that Git handles. If a merge commit exists in the submodule directory that contains **both** commits in its history, Git will suggest it to you as a possible solution. It sees that at some point in the submodule project, someone merged branches containing these two commits, so maybe you'll want that one.

This is why the error message from before was “merge following commits not found”, because it could not do **this**. It's confusing because who would expect it to **try** to do this?

If it does find a single acceptable merge commit, you'll see something like this:

```
$ 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.

The suggested command Git is providing will update the index as though you had run `git add` (which clears the conflict), then commit. You probably shouldn't do this though. You can just as easily go into the submodule directory, see what the difference is, fast-forward to this commit, test it properly, and then commit it.

```
$ cd DbConnector/
$ git merge 9fd905e
Updating eb41d76..9fd905e
Fast-forward
```

```
$ cd ..
$ git add DbConnector
$ git commit -am 'Fast forwarded to a common submodule child'
```

This accomplishes the same thing, but at least this way you can verify that it works and you have the code in your submodule directory when you're done.

## Submodule Tips

There are a few things you can do to make working with submodules a little easier.

### Submodule Foreach

There is a `foreach` submodule command to run some arbitrary command in each submodule. This can be really helpful if you have a number of submodules in the same project.

For example, let's say we want to start a new feature or do a bugfix and we have work going on in several submodules. We can easily stash all the work in all our submodules.

```
$ 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
```

Then we can create a new branch and switch to it in all our submodules.

```
$ git submodule foreach 'git checkout -b featureA'  
Entering 'CryptoLibrary'  
Switched to a new branch 'featureA'  
Entering 'DbConnector'  
Switched to a new branch 'featureA'
```

You get the idea. One really useful thing you can do is produce a nice unified diff of what is changed in your main project and all your subprojects as well.

```

$ git diff; git submodule foreach 'git diff'
Submodule DbConnector contains modified content
diff --git a/src/main.c b/src/main.c
index 210f1ae..1f0acdc 100644
--- a/src/main.c
+++ b/src/main.c
@@ -245,6 +245,8 @@ static int handle_alias(int *argcp, const char ***argv)

    commit_page_choice();

+    url = url_decode(url_orig);
+
/* build alias_argv */
alias_argv = xmalloc(sizeof(*alias_argv) * (argc + 1));
alias_argv[0] = alias_string + 1;
Entering 'DbConnector'
diff --git a/src/db.c b/src/db.c
index 1aaefb6..5297645 100644
--- a/src/db.c
+++ b/src/db.c
@@ -93,6 +93,11 @@ char *url_decode_mem(const char *url, int len)
    return url_decode_internal(&url, len, NULL, &out, 0);
}

+char *url_decode(const char *url)
+{
+    return url_decode_mem(url, strlen(url));
+}
+
char *url_decode_parameter_name(const char **query)
{
    struct strbuf out = STRBUF_INIT;

```

Here we can see that we're defining a function in a submodule and calling it in the main project. This is obviously a simplified example, but hopefully it gives you an idea of how this may be useful.

## Useful Aliases

You may want to set up some aliases for some of these commands as they can be quite long and you can't set configuration options for most of them to make them defaults. We covered setting up Git aliases in [Git Aliases](#), but here is an example of what you may want to set up if you plan on working with submodules in Git a lot.

```

$ 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'

```

This way you can simply run `git supdate` when you want to update your submodules, or `git spush`

to push with submodule dependency checking.

## Issues with Submodules

Using submodules isn't without hiccups, however.

### Switching branches

For instance, switching branches with submodules in them can also be tricky with Git versions older than Git 2.13. If you create a new branch, add a submodule there, and then switch back to a branch without that submodule, you still have the submodule directory as an untracked directory:

```
$ 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 'adding crypto library'
[add-crypto 4445836] adding crypto library
 2 files changed, 4 insertions(+)
 create mode 160000 CryptoLibrary

$ git checkout master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
Your branch is up-to-date with 'origin/master'.

$ git status
On branch master
Your branch is up-to-date with 'origin/master'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    CryptoLibrary/

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

Removing the directory isn't difficult, but it can be a bit confusing to have that in there. If you do remove it and then switch back to the branch that has that submodule, you will need to run `submodule update --init` to repopulate it.

```
$ 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
```

Again, not really very difficult, but it can be a little confusing.

Newer Git versions (Git >= 2.13) simplify all this by adding the `--recurse-submodules` flag to the `git checkout` command, which takes care of placing the submodules in the right state for the branch we are switching to.

```
$ 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 'adding crypto library'
[add-crypto 4445836] adding 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
```

Using the the `--recurse-submodules` flag of `git checkout` can also be useful when you work on several branches in the superproject, each having your submodule pointing at different commits. Indeed, if you switch between branches that record the submodule at different commits, upon

executing `git status` the submodule will appear as “modified”, and indicate “new commits”. That is because the submodule state is by default not carried over when switching branches.

This can be really confusing, so it’s a good idea to always `git checkout --recurse-submodules` when your project has submodules. (For older Git versions that do not have the `--recurse-submodules` flag, after the checkout you can use `git submodule update --init --recursive` to put the submodules in the right state.)

Luckily, you can tell Git ( $>=2.14$ ) to always use the `--recurse-submodules` flag by setting the configuration option `submodule.recurse: git config submodule.recurse true`. As noted above, this will also make Git recurse into submodules for every command that has a `--recurse-submodules` option (except `git clone`).

#### ===== Switching from subdirectories to submodules

The other main caveat that many people run into involves switching from subdirectories to submodules. If you’ve been tracking files in your project and you want to move them out into a submodule, you must be careful or Git will get angry at you. Assume that you have files in a subdirectory of your project, and you want to switch it to a submodule. If you delete the subdirectory and then run `submodule add`, Git yells at you:

```
$ rm -Rf CryptoLibrary/
$ git submodule add https://github.com/chaconinc/CryptoLibrary
'CryptoLibrary' already exists in the index
```

You have to unstage the `CryptoLibrary` directory first. Then you can add the submodule:

```
$ 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.
```

Now suppose you did that in a branch. If you try to switch back to a branch where those files are still in the actual tree rather than a submodule – you get this error:

```
$ 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
```

You can force it to switch with `checkout -f`, but be careful that you don't have unsaved changes in there as they could be overwritten with that command.

```
$ git checkout -f master
warning: unable to rmdir CryptoLibrary: Directory not empty
Switched to branch 'master'
```

Then, when you switch back, you get an empty `CryptoLibrary` directory for some reason and `git submodule update` may not fix it either. You may need to go into your submodule directory and run a `git checkout .` to get all your files back. You could run this in a `submodule foreach` script to run it for multiple submodules.

It's important to note that submodules these days keep all their Git data in the top project's `.git` directory, so unlike much older versions of Git, destroying a submodule directory won't lose any commits or branches that you had.

With these tools, submodules can be a fairly simple and effective method for developing on several related but still separate projects simultaneously.

## Bundling

Though we've covered the common ways to transfer Git data over a network (HTTP, SSH, etc), there is actually one more way to do so that is not commonly used but can actually be quite useful.

Git is capable of "bundling" its data into a single file. This can be useful in various scenarios. Maybe your network is down and you want to send changes to your co-workers. Perhaps you're working somewhere offsite and don't have access to the local network for security reasons. Maybe your wireless/ethernet card just broke. Maybe you don't have access to a shared server for the moment, you want to email someone updates and you don't want to transfer 40 commits via `format-patch`.

This is where the `git bundle` command can be helpful. The `bundle` command will package up everything that would normally be pushed over the wire with a `git push` command into a binary file that you can email to someone or put on a flash drive, then unbundle into another repository.

Let's see a simple example. Let's say you have a repository with two 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
```

If you want to send that repository to someone and you don't have access to a repository to push to, or simply don't want to set one up, you can bundle it with `git bundle create`.

```
$ git bundle create repo.bundle HEAD master
Counting objects: 6, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (6/6), 441 bytes, done.
Total 6 (delta 0), reused 0 (delta 0)
```

Now you have a file named `repo.bundle` that has all the data needed to re-create the repository's `master` branch. With the `bundle` command you need to list out every reference or specific range of commits that you want to be included. If you intend for this to be cloned somewhere else, you should add `HEAD` as a reference as well as we've done here.

You can email this `repo.bundle` file to someone else, or put it on a USB drive and walk it over.

On the other side, say you are sent this `repo.bundle` file and want to work on the project. You can clone from the binary file into a directory, much like you would from a URL.

```
$ git clone repo.bundle repo
Cloning into 'repo'...
...
$ cd repo
$ git log --oneline
9a466c5 second commit
b1ec324 first commit
```

If you don't include `HEAD` in the references, you have to also specify `-b master` or whatever branch is included because otherwise it won't know what branch to check out.

Now let's say you do three commits on it and want to send the new commits back via a bundle on a USB stick or email.

```
$ git log --oneline
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
9a466c5 second commit
b1ec324 first commit
```

First we need to determine the range of commits we want to include in the bundle. Unlike the network protocols which figure out the minimum set of data to transfer over the network for us, we'll have to figure this out manually. Now, you could just do the same thing and bundle the entire repository, which will work, but it's better to just bundle up the difference - just the three commits we just made locally.

In order to do that, you'll have to calculate the difference. As we described in [Commit-Bereiche](#), you can specify a range of commits in a number of ways. To get the three commits that we have in our master branch that weren't in the branch we originally cloned, we can use something like `origin/master..master` or `master ^origin/master`. You can test that with the `log` command.

```
$ git log --oneline master ^origin/master
71b84da last commit - second repo
c99cf5b fourth commit - second repo
7011d3d third commit - second repo
```

So now that we have the list of commits we want to include in the bundle, let's bundle them up. We do that with the `git bundle create` command, giving it a filename we want our bundle to be and the range of commits we want to go into it.

```
$ 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)
```

Now we have a `commits.bundle` file in our directory. If we take that and send it to our partner, she can then import it into the original repository, even if more work has been done there in the meantime.

When she gets the bundle, she can inspect it to see what it contains before she imports it into her repository. The first command is the `bundle verify` command that will make sure the file is actually a valid Git bundle and that you have all the necessary ancestors to reconstitute it properly.

```
$ 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
```

If the bundler had created a bundle of just the last two commits they had done, rather than all three, the original repository would not be able to import it, since it is missing requisite history. The `verify` command would have looked like this instead:

```
$ git bundle verify ../commits-bad.bundle
error: Repository lacks these prerequisite commits:
error: 7011d3d8fc200abe0ad561c011c3852a4b7bbe95 third commit - second repo
```

However, our first bundle is valid, so we can fetch in commits from it. If you want to see what branches are in the bundle that can be imported, there is also a command to just list the heads:

```
$ git bundle list-heads ../commits.bundle
71b84daaf49abed142a373b6e5c59a22dc6560dc refs/heads/master
```

The `verify` sub-command will tell you the heads as well. The point is to see what can be pulled in, so you can use the `fetch` or `pull` commands to import commits from this bundle. Here we'll fetch the *master* branch of the bundle to a branch named *other-master* in our repository:

```
$ git fetch ../commits.bundle master:other-master
From ../commits.bundle
 * [new branch]      master      -> other-master
```

Now we can see that we have the imported commits on the *other-master* branch as well as any commits we've done in the meantime in our own *master* branch.

```
$ 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
```

So, `git bundle` can be really useful for sharing or doing network-type operations when you don't have the proper network or shared repository to do so.

# Replace

As we've emphasized before, the objects in Git's object database are unchangeable, but Git does provide an interesting way to *pretend* to replace objects in its database with other objects.

The `replace` command lets you specify an object in Git and say "every time you refer to *this* object, pretend it's a *different* object". This is most commonly useful for replacing one commit in your history with another one without having to rebuild the entire history with, say, `git filter-branch`.

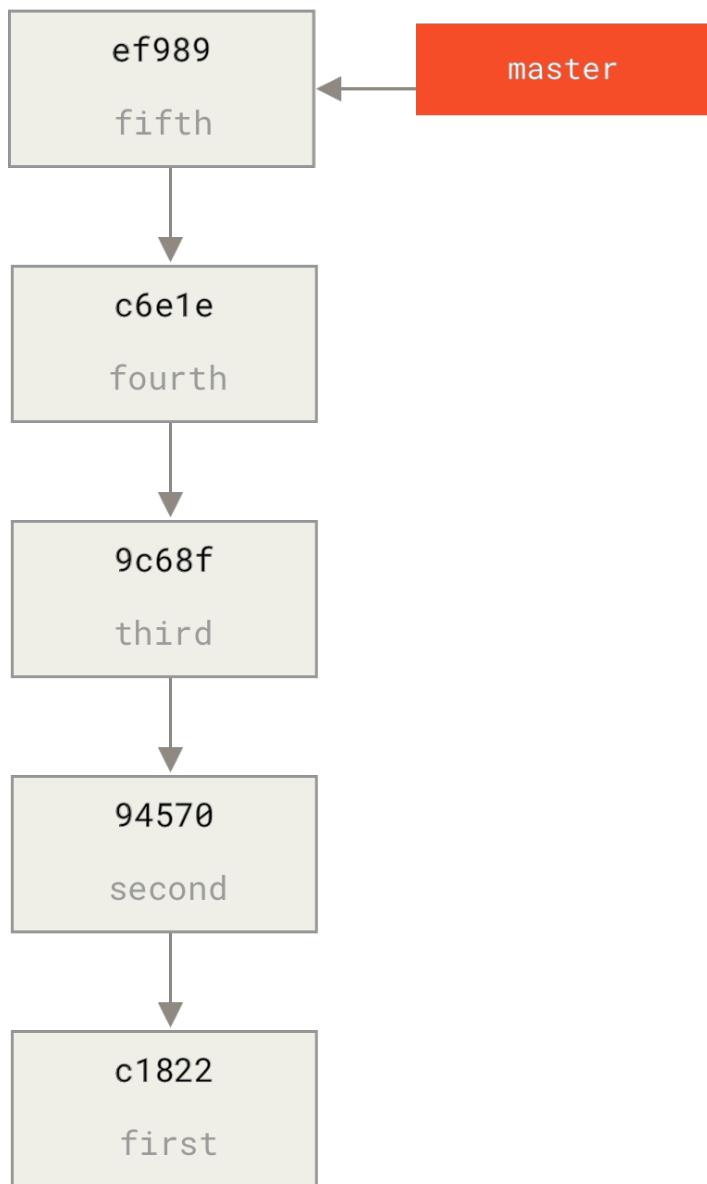
For example, let's say you have a huge code history and want to split your repository into one short history for new developers and one much longer and larger history for people interested in data mining. You can graft one history onto the other by "replacing" the earliest commit in the new line with the latest commit on the older one. This is nice because it means that you don't actually have to rewrite every commit in the new history, as you would normally have to do to join them together (because the parentage affects the SHA-1s).

Let's try this out. Let's take an existing repository, split it into two repositories, one recent and one historical, and then we'll see how we can recombine them without modifying the recent repositories SHA-1 values via `replace`.

We'll use a simple repository with five simple commits:

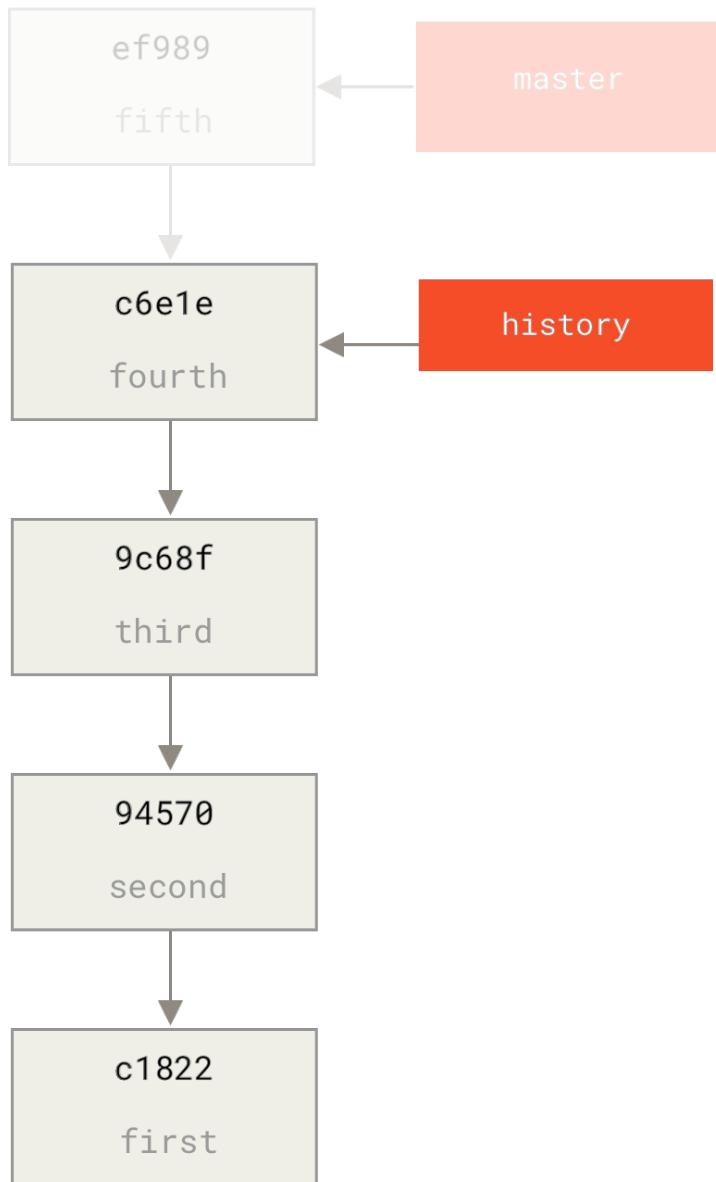
```
$ git log --oneline
ef989d8 fifth commit
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

We want to break this up into two lines of history. One line goes from commit one to commit four - that will be the historical one. The second line will just be commits four and five - that will be the recent history.



Well, creating the historical history is easy, we can just put a branch in the history and then push that branch to the master branch of a new remote repository.

```
$ git branch history c6e1e95
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```



Now we can push the new `history` branch to the `master` branch of our new repository:

```

$ 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

```

OK, so our history is published. Now the harder part is truncating our recent history down so it's

smaller. We need an overlap so we can replace a commit in one with an equivalent commit in the other, so we're going to truncate this to just commits four and five (so commit four overlaps).

```
$ git log --oneline --decorate
ef989d8 (HEAD, master) fifth commit
c6e1e95 (history) fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

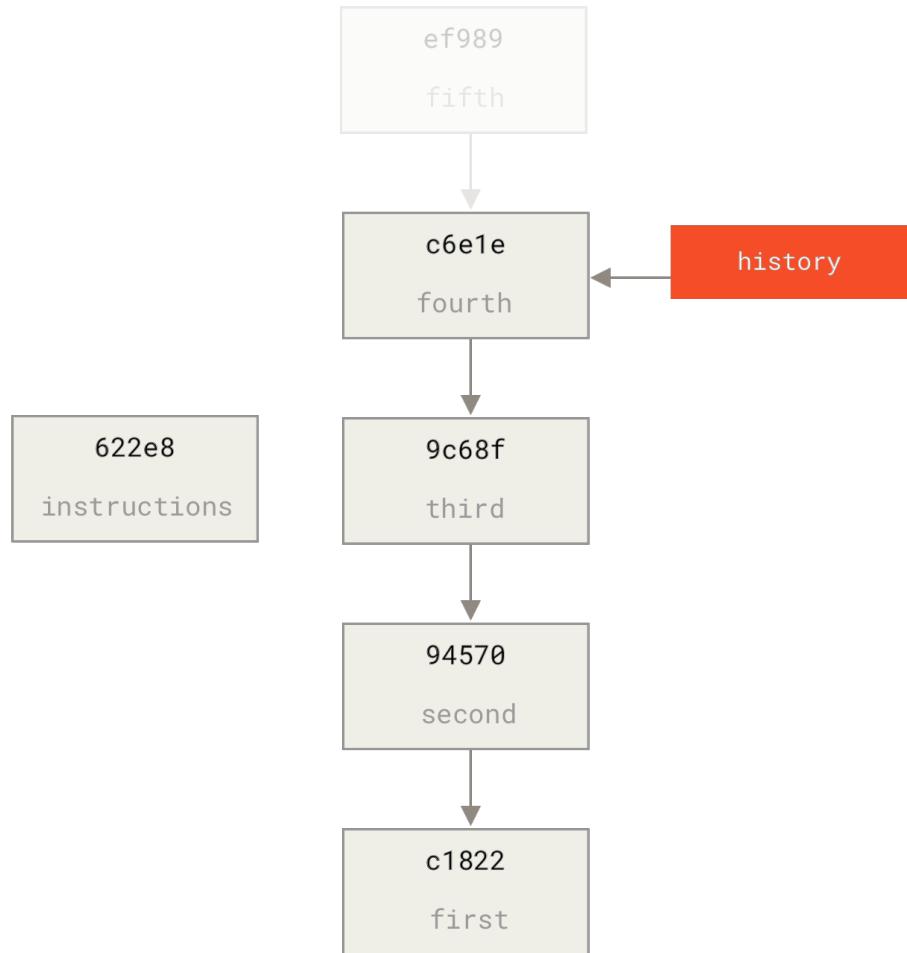
It's useful in this case to create a base commit that has instructions on how to expand the history, so other developers know what to do if they hit the first commit in the truncated history and need more. So, what we're going to do is create an initial commit object as our base point with instructions, then rebase the remaining commits (four and five) on top of it.

To do that, we need to choose a point to split at, which for us is the third commit, which is `9c68fdc` in SHA-speak. So, our base commit will be based off of that tree. We can create our base commit using the `commit-tree` command, which just takes a tree and will give us a brand new, parentless commit object SHA-1 back.

```
$ echo 'get history from blah blah blah' | git commit-tree 9c68fdc^{tree}
622e88e9cbfbacfb75b5279245b9fb38dfa10cf
```

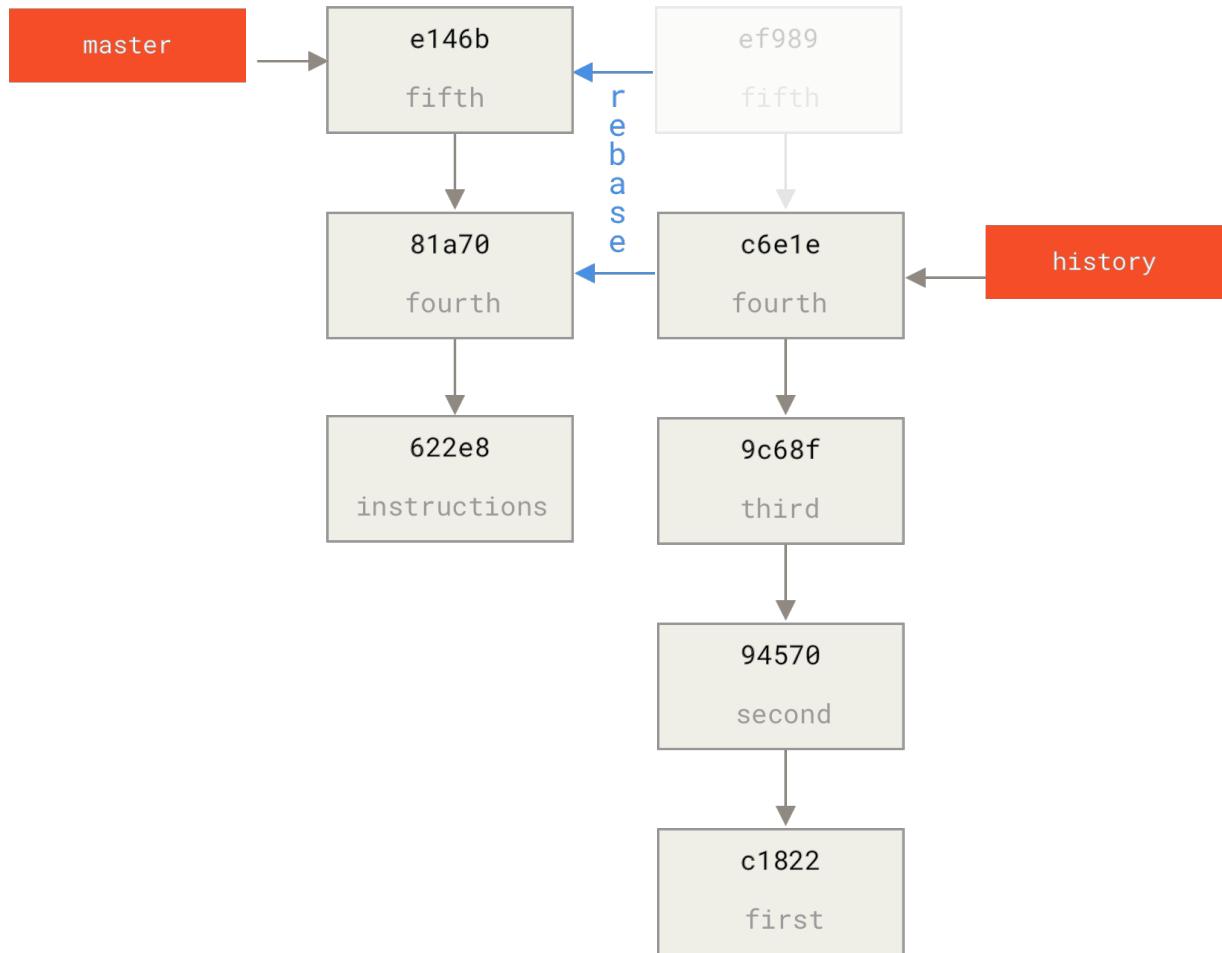


The `commit-tree` command is one of a set of commands that are commonly referred to as *plumbing* commands. These are commands that are not generally meant to be used directly, but instead are used by `other` Git commands to do smaller jobs. On occasions when we're doing weirder things like this, they allow us to do really low-level things but are not meant for daily use. You can read more about plumbing commands in [Basisbefehle und Standardbefehle \(Plumbing and Porcelain\)](#)



OK, so now that we have a base commit, we can rebase the rest of our history on top of that with `git rebase --onto`. The `--onto` argument will be the SHA-1 we just got back from `commit-tree` and the rebase point will be the third commit (the parent of the first commit we want to keep, `9c68fdc`):

```
$ git rebase --onto 622e88 9c68fdc
First, rewinding head to replay your work on top of it...
Applying: fourth commit
Applying: fifth commit
```



OK, so now we've re-written our recent history on top of a throw away base commit that now has instructions in it on how to reconstitute the entire history if we wanted to. We can push that new history to a new project and now when people clone that repository, they will only see the most recent two commits and then a base commit with instructions.

Let's now switch roles to someone cloning the project for the first time who wants the entire history. To get the history data after cloning this truncated repository, one would have to add a second remote for the historical repository and fetch:

```
$ 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
```

Now the collaborator would have their recent commits in the `master` branch and the historical commits in the `project-history/master` branch.

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
622e88e get history from blah blah blah

$ git log --oneline project-history/master
c6e1e95 fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

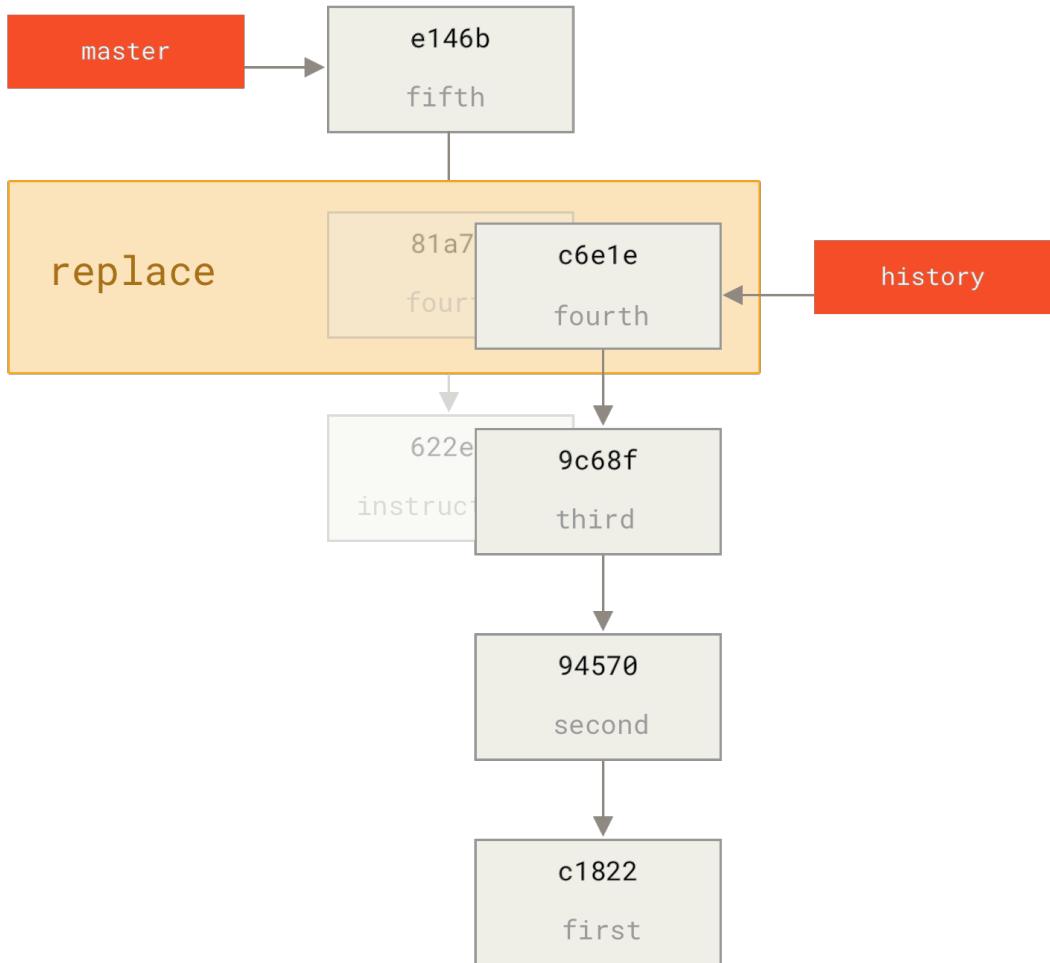
To combine them, you can simply call `git replace` with the commit you want to replace and then the commit you want to replace it with. So we want to replace the "fourth" commit in the `master` branch with the "fourth" commit in the `project-history/master` branch:

```
$ git replace 81a708d c6e1e95
```

Now, if you look at the history of the `master` branch, it appears to look like this:

```
$ git log --oneline master
e146b5f fifth commit
81a708d fourth commit
9c68fdc third commit
945704c second commit
c1822cf first commit
```

Cool, right? Without having to change all the SHA-1s upstream, we were able to replace one commit in our history with an entirely different commit and all the normal tools (`bisect`, `blame`, etc) will work how we would expect them to.



Interestingly, it still shows `81a708d` as the SHA-1, even though it's actually using the `c6e1e95` commit data that we replaced it with. Even if you run a command like `cat-file`, it will show you the replaced data:

```

$ 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

```

Remember that the actual parent of `81a708d` was our placeholder commit (`622e88e`), not `9c68fdce` as it states here.

Another interesting thing is that this data is kept in our references:

```
$ 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
```

This means that it's easy to share our replacement with others, because we can push this to our server and other people can easily download it. This is not that helpful in the history grafting scenario we've gone over here (since everyone would be downloading both histories anyhow, so why separate them?) but it can be useful in other circumstances.

## Anmeldeinformationen speichern

If you use the SSH transport for connecting to remotes, it's possible for you to have a key without a passphrase, which allows you to securely transfer data without typing in your username and password. However, this isn't possible with the HTTP protocols – every connection needs a username and password. This gets even harder for systems with two-factor authentication, where the token you use for a password is randomly generated and unpronounceable.

Fortunately, Git has a credentials system that can help with this. Git has a few options provided in the box:

- The default is not to cache at all. Every connection will prompt you for your username and password.
- The “cache” mode keeps credentials in memory for a certain period of time. None of the passwords are ever stored on disk, and they are purged from the cache after 15 minutes.
- The “store” mode saves the credentials to a plain-text file on disk, and they never expire. This means that until you change your password for the Git host, you won't ever have to type in your credentials again. The downside of this approach is that your passwords are stored in cleartext in a plain file in your home directory.
- If you're using a Mac, Git comes with an “osxkeychain” mode, which caches credentials in the secure keychain that's attached to your system account. This method stores the credentials on disk, and they never expire, but they're encrypted with the same system that stores HTTPS certificates and Safari auto-fills.
- If you're using Windows, you can install a helper called “Git Credential Manager for Windows.” This is similar to the “osxkeychain” helper described above, but uses the Windows Credential Store to control sensitive information. It can be found at <https://github.com/Microsoft/Git-Credential-Manager-for-Windows>.

You can choose one of these methods by setting a Git configuration value:

```
$ git config --global credential.helper cache
```

Some of these helpers have options. The “store” helper can take a `--file <path>` argument, which customizes where the plain-text file is saved (the default is `~/.git-credentials`). The “cache” helper accepts the `--timeout <seconds>` option, which changes the amount of time its daemon is kept running (the default is “900”, or 15 minutes). Here’s an example of how you’d configure the “store” helper with a custom file name:

```
$ git config --global credential.helper 'store --file ~/.my-credentials'
```

Git even allows you to configure several helpers. When looking for credentials for a particular host, Git will query them in order, and stop after the first answer is provided. When saving credentials, Git will send the username and password to **all** of the helpers in the list, and they can choose what to do with them. Here’s what a `.gitconfig` would look like if you had a credentials file on a thumb drive, but wanted to use the in-memory cache to save some typing if the drive isn’t plugged in:

```
[credential]
helper = store --file /mnt/thumbdrive/.git-credentials
helper = cache --timeout 3000
```

## Under the Hood

How does this all work? Git’s root command for the credential-helper system is `git credential`, which takes a command as an argument, and then more input through stdin.

This might be easier to understand with an example. Let’s say that a credential helper has been configured, and the helper has stored credentials for `mygithost`. Here’s a session that uses the “fill” command, which is invoked when Git is trying to find credentials for a host:

```
$ git credential fill ①
protocol=https ②
host=mygithost
③
protocol=https ④
host=mygithost
username=bob
password=s3cre7
$ git credential fill ⑤
protocol=https
host=unknownhost

Username for 'https://unknownhost': bob
Password for 'https://bob@unknownhost':
protocol=https
host=unknownhost
username=bob
password=s3cre7
```

① This is the command line that initiates the interaction.

- ② Git-credential is then waiting for input on stdin. We provide it with the things we know: the protocol and hostname.
- ③ A blank line indicates that the input is complete, and the credential system should answer with what it knows.
- ④ Git-credential then takes over, and writes to stdout with the bits of information it found.
- ⑤ If credentials are not found, Git asks the user for the username and password, and provides them back to the invoking stdout (here they're attached to the same console).

The credential system is actually invoking a program that's separate from Git itself; which one and how depends on the `credential.helper` configuration value. There are several forms it can take:

| Configuration Value                              | Behavior  |
|--|---|
| <code>foo</code>                                 | Runs <code>git-credential-foo</code>              |
| <code>foo -a --opt=bcd</code>                    | Runs <code>git-credential-foo -a --opt=bcd</code> |
| <code>/absolute/path/foo -xyz</code>             | Runs <code>/absolute/path/foo -xyz</code>         |
| <code>!f() { echo "password=s3cre7"; }; f</code> | Code after <code>!</code> evaluated in shell      |

So the helpers described above are actually named `git-credential-cache`, `git-credential-store`, and so on, and we can configure them to take command-line arguments. The general form for this is “`git-credential-foo [args] <action>`.” The stdin/stdout protocol is the same as git-credential, but they use a slightly different set of actions:

- `get` is a request for a username/password pair.
- `store` is a request to save a set of credentials in this helper's memory.
- `erase` purge the credentials for the given properties from this helper's memory.

For the `store` and `erase` actions, no response is required (Git ignores it anyway). For the `get` action, however, Git is very interested in what the helper has to say. If the helper doesn't know anything useful, it can simply exit with no output, but if it does know, it should augment the provided information with the information it has stored. The output is treated like a series of assignment statements; anything provided will replace what Git already knows.

Here's the same example from above, but skipping git-credential and going straight for 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 Here we tell `git-credential-store` to save some credentials: the username “bob” and the password “s3cre7” are to be used when `https://mygithost` is accessed.
- ② Now we’ll retrieve those credentials. We provide the parts of the connection we already know (`https://mygithost`), and an empty line.
- ③ `git-credential-store` replies with the username and password we stored above.

Here’s what the `~/git.store` file looks like:

```
https://bob:s3cre7@mygithost
```

It’s just a series of lines, each of which contains a credential-decorated URL. The `osxkeychain` and `wincred` helpers use the native format of their backing stores, while `cache` uses its own in-memory format (which no other process can read).

## A Custom Credential Cache

Given that `git-credential-store` and friends are separate programs from Git, it’s not much of a leap to realize that *any* program can be a Git credential helper. The helpers provided by Git cover many common use cases, but not all. For example, let’s say your team has some credentials that are shared with the entire team, perhaps for deployment. These are stored in a shared directory, but you don’t want to copy them to your own credential store, because they change often. None of the existing helpers cover this case; let’s see what it would take to write our own. There are several key features this program needs to have:

1. The only action we need to pay attention to is `get`; `store` and `erase` are write operations, so we’ll just exit cleanly when they’re received.
2. The file format of the shared-credential file is the same as that used by `git-credential-store`.
3. The location of that file is fairly standard, but we should allow the user to pass a custom path just in case.

Once again, we’ll write this extension in Ruby, but any language will work so long as Git can execute the finished product. Here’s the full source code of our new credential helper:

```

#!/usr/bin/env ruby

require 'optparse'

path = File.expand_path '~/.git-credentials' ①
OptionParser.new do |opts|
  opts.banner = 'USAGE: git-credential-read-only [options] <action>'
  opts.on('-f', '--file PATH', 'Specify path for backing store') do |argpath|
    path = File.expand_path argpath
  end
end.parse!

exit(0) unless ARGV[0].downcase == 'get' ②
exit(0) unless File.exists? path

known = {} ③
while line = STDIN.gets
  break if line.strip == ''
  k,v = line.strip.split '=', 2
  known[k] = v
end

File.readlines(path).each do |fileline| ④
  prot,user,pass,host = fileline.scan(/^(.*?):\/\/(.*):(.*?)@(.*)$/).first
  if prot == known['protocol'] and host == known['host'] and user == known['username'] then
    puts "protocol=#{prot}"
    puts "host=#{host}"
    puts "username=#{user}"
    puts "password=#{pass}"
    exit(0)
  end
end

```

- ① Here we parse the command-line options, allowing the user to specify the input file. The default is `~/.git-credentials`.
- ② This program only responds if the action is `get` and the backing-store file exists.
- ③ This loop reads from `stdin` until the first blank line is reached. The inputs are stored in the `known` hash for later reference.
- ④ This loop reads the contents of the storage file, looking for matches. If the protocol and host from `known` match this line, the program prints the results to `stdout` and exits.

We'll save our helper as `git-credential-read-only`, put it somewhere in our `PATH` and mark it executable. Here's what an interactive session looks like:

```
$ git credential-read-only --file=/mnt/shared/creds get
protocol=https
host=mygithost

protocol=https
host=mygithost
username=bob
password=s3cre7
```

Since its name starts with “git-”, we can use the simple syntax for the configuration value:

```
$ git config --global credential.helper 'read-only --file /mnt/shared/creds'
```

As you can see, extending this system is pretty straightforward, and can solve some common problems for you and your team.

## Zusammenfassung

Sie haben eine Reihe von fortschrittlichen Tools kennengelernt, mit denen Sie Ihre Commits und die Staging-Area präziser manipulieren können. Wenn Sie Probleme bemerken, sollten Sie in der Lage sein, leicht herauszufinden, was, wann und von wem der Commit eingebracht wurde. Wenn Sie Teilprojekte in Ihrem Projekt verwenden möchten, haben Sie gelernt, wie Sie deren Anforderungen erfüllen können. An diesem Punkt sollten Sie auf der Kommandozeile die meisten der täglichen Aufgaben in Git erledigen können und Sie sollten sich dabei sicher fühlen.

# Git einrichten

Bisher haben wir die grundlegende Funktionsweise und die Verwendung von Git erläutert, sowie eine Reihe von Tools vorgestellt, die Ihnen helfen sollen, Git einfach und effizient zu nutzen. In diesem Kapitel werden wir zeigen, wie Sie Git noch individueller einsetzen können, indem Sie einige wichtige Konfigurationen anpassen und das Hook-System anwenden. Mit diesen Tools ist es einfach, Git genau so einzurichten, wie Sie, Ihr Unternehmen oder Ihre Gruppe es benötigen.

## Git Konfiguration

Wie Sie in Kapitel 1, [Erste Schritte](#) bereits kurz gelesen haben, können Sie die Git-Konfigurationseinstellungen mit dem Befehl `git config` anpassen. Eine der ersten Dinge, die Sie vorgenommen haben, war die Einrichtung Ihres Namens und Ihrer E-Mail-Adresse:

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

Jetzt lernen Sie einige der interessanteren Optionen kennen, die Sie auf diese Weise festlegen können, um Ihre Nutzung von Git zu optimieren.

Zunächst eine kleine Übersicht: Git verwendet eine Reihe von Konfigurationsdateien, um das Standard-Verhalten, wie gewünscht, zu verändern. Die erste Option, in der Git nach solchen Werten sucht, ist die globale Datei `/etc/gitconfig`, die Einstellungen enthält, die auf jeden Benutzer auf dem System und alle seine Repositories angewendet werden. Wenn Sie die Option `--system` an `git config` übergeben, wird diese Datei gezielt ausgelesen und geschrieben.

Der nächste Ort, an dem Git nachschaut, ist die Datei `~/.gitconfig` (oder `~/.config/git/config`), die für jeden Benutzer spezifisch ist. Sie können Git veranlassen, diese Datei zu lesen und zu schreiben, indem Sie die Option `--global` übergeben.

Schließlich sucht Git nach Informationen in der Konfigurations-Datei im Git-Verzeichnis (`.git/config`) des jeweiligen Repositorys, das Sie gerade verwenden. Diese Werte sind spezifisch für dieses spezielle Repository und werden bei Übergabe der Option `--local` an `git config` angewendet. (Wenn Sie nicht angeben, welchen Level Sie ansprechen möchten, ist das die Voreinstellung.)

Jeder dieser „Level“ (system, global, lokal) überschreibt Werte der vorigen Ebene. Daher werden beispielsweise Werte in `.git/config` jene in `/etc/gitconfig` überschreiben.



Die Konfigurationsdateien von Git sind reine Textdateien, so dass Sie diese Werte auch einstellen können, wenn Sie die Datei manuell bearbeiten und dabei die richtige Syntax verwenden. Es ist jedoch generell einfacher, 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 Ihrer

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 Sie eine Liste aller Optionen sehen möchten, die Ihre Version von Git kennt, können Sie Folgendes aufrufen:

```
$ man git-config
```

Dieser Befehl listet alle verfügbaren Optionen detailliert auf. Sie finden dieses Referenzmaterial auch unter [Git-Config](#).

### core.editor

Um Ihre Commit- und Tag-Beschreibungen erstellen/bearbeiten zu können verwendet Git das von Ihnen 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 können Sie ändern, indem Sie die Einstellung `core.editor` verwenden:

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

Jetzt wird Git Emacs starten, unabhängig davon, was als Standard-Shell-Editor eingestellt ist, um die Beschreibungen zu bearbeiten.

### commit.template

Wenn Sie dieses Attribut auf die Pfadangabe einer Datei auf Ihrem System setzen, verwendet Git diese Datei als initiale Standard-Nachricht, wenn Sie einen Commit durchführen. Der Vorteil bei der Erstellung einer benutzerdefinierten Commit-Vorlage besteht darin, dass Sie sie verwenden können, 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]
```

Beachten Sie, wie diese Commit-Vorlage den Committer daran erinnert, 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 Ihrem Editor erscheint, wenn Sie `git commit` ausführen, setzen Sie den Konfigurationswert `commit.template`:

```
$ git config --global commit.template ~/.gitmessage.txt  
$ git commit
```

Dann öffnet sich Ihr Text-Editor in etwa so für Ihre platzhaltergemäße Commit-Beschreibung, wenn Sie committen:

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 Ihres 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 Ihnen bevorzugt eingestellt werden (Standard ist `less`). Sie können ihn deaktivieren, indem Sie eine leere Zeichenkette verwenden:

```
$ git config --global core.pager ''
```

Wenn Sie das benutzen, wird Git die komplette Ausgabe aller Befehle abrufen, unabhängig davon, wie lang sie sind.

### user.signingkey

Wenn Sie signierte kommentierte Tags erstellen (wie in Kapitel 7, [Ihre Arbeit signieren](#) beschrieben), erleichtert die Definition Ihres GPG-Signierschlüssels in der Konfigurations-Einstellung die Arbeit. Stellen Sie Ihre Schlüssel-ID so ein:

```
$ git config --global user.signingkey <gpg-key-id>
```

Jetzt können Sie Tags signieren, ohne jedes Mal Ihren Schlüssel mit dem Befehl `git tag` angeben zu müssen:

```
$ git tag -s <tag-name>
```

#### core.excludesfile

Sie können Suchmuster in die `.gitignore` Datei Ihres Projekts einfügen. Damit können Sie verhindern, dass Git, so bestimmte Dateien, als nicht in der Versionsverwaltung erfasste Dateien erkennt oder sie zum Commit vormerkt, wenn Sie `git add` darauf ausführen, wie in Kapitel 2, [Ignorieren von Dateien](#) beschrieben.

In manchen Fällen sollen bestimmte Dateien für alle Repositorys ignoriert werden, in denen Sie arbeiten. Falls Sie MacOS verwenden, kennen Sie vermutlich die `.DS_Store` Dateien. Bei Emacs oder Vim kennen Sie Dateinamen, die mit einer Tilde (~) oder auf `.swp` enden.

Mit dieser Einstellung können Sie eine gewisse, globale `.gitignore` Datei schreiben. Erstellen Sie eine `~/.gitignore_global` Datei mit diesem Inhalt:

```
*~  
.*.swp  
.DS_Store
```

... und Sie führen `git config --global core.excludesfile ~/.gitignore_global` aus. Git wird sich nie wieder um diese Dateien kümmern.

#### help.autocorrect

Wenn Sie einen Befehl vertippen, zeigt das System Ihnen so etwas wie das hier:

```
$ git chekcout master  
git: 'chekcout' is not a git command. See 'git --help'.  
  
Did you mean this?  
    checkout
```

Git versucht behilflich zu sein, um herauszufinden, was Sie gemeint haben, aber es verweigert es immer noch die Ausführung. Wenn Sie `help.autocorrect` auf 1 setzen, dann führt Git diesen Befehl tatsächlich 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...
```

Beachten Sie die „0,1 Sekunden“ Aktivität. `help.autocorrect` ist eigentlich eine Ganzzahl, die Zehntelsekunden repräsentiert. Wenn Sie den Wert auf 50 setzen, gibt Ihnen Git 5 Sekunden Zeit, Ihre 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 analysieren. Eine Reihe von Optionen können Ihnen helfen, die Farbgestaltung nach Ihren Wünschen einzustellen.

### `color.ui`

Git färbt den größten Teil seiner Ausgabe automatisch ein, aber es gibt einen Hauptschalter, wenn Ihnen dieses Verhalten nicht gefällt. Um alle farbigen Terminalausgaben von Git auszuschalten, verfahren Sie wie folgt:

```
$ 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.

Sie können es auch auf `always` einstellen, dass der Unterschied zwischen Terminals und Pipes immer ignoriert wird. Das werden Sie nur selten wollen; in den meisten Szenarien können Sie stattdessen ein `--color` Flag an den Git-Befehl übergeben, um ihn zu zwingen, Farbcodes zu verwenden, wenn Sie Farbcodes in Ihrer umgeleiteten Ausgabe wünschen. Die Voreinstellung ist fast immer die richtige.

### `color.*`

Wenn Sie genauer bestimmen möchten, welche Befehle wie eingefärbt werden, dann bietet Git 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 Sie bestimmte Farben für Teile der Ausgabe festlegen können, wenn Sie die Farben überschreiben wollen. Um beispielsweise die Metainformationen in Ihrer Diff-Ausgabe auf blauen Vordergrund, schwarzen Hintergrund und

fetten Text zu setzen, können Sie Folgendes ausführen:

```
$ git config --global color.diff.meta "blue black bold"
```

Sie können die Farbe auf einen der folgenden Werte setzen: `normal`, `black`, `red`, `green`, `yellow`, `blue`, `magenta`, `cyan` oder `white`. Wenn Sie ein Attribut wie im vorherigen Beispiel fett wünschen, können Sie zwischen `bold` (fett), `dim` (abgedunkelt), `ul` (unterstrichen), `blink` (blitzen) 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, können Sie stattdessen ein externes Tool verwenden. Sie können auch ein grafisches Tool zum Mergen und Lösen von Konflikten einrichten, anstatt Konflikte manuell lösen zu müssen. Wir zeigen Ihnen, wie Sie das Perforce Visual Merge Tool (P4Merge) einrichten, um Ihre Diffs und Merge-Ansichten zu analysieren, denn es ist ein praktisches grafisches Tool und kostenlos.

Wenn Sie diese Software testen möchten – P4Merge läuft auf allen wichtigen Plattformen. Wir verwenden Pfadnamen in den Beispielen, die auf MacOS- und Linux-Systemen funktionieren. Unter Windows müssen Sie `/usr/local/bin` in einen ausführbaren Pfad in Ihrer Umgebung ändern.

Starten Sie mit [P4Merge von Perforce downloaden](#). Richten Sie danach externe Wrapper-Skripte ein, um Ihre 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 Ihre `p4merge` Binary installiert ist. Richten Sie ein Merge-Wrapper-Skript mit dem Namen `extMerge` ein, das Ihre 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 Ihr 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 Sie nur die Argumente `old-file` und `new-file` benötigen, verwenden Sie das Wrapper-Skript, um die benötigten zu übergeben.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Sie müssen außerdem darauf achten, dass diese Tools lauffähig sind:

```
$ sudo chmod +x /usr/local/bin/extMerge  
$ sudo chmod +x /usr/local/bin/extDiff
```

Jetzt können Sie Ihre Konfigurationsdatei so einrichten, dass sie Ihre 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. Sie können 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 Sie können 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 eingestellt ist, können Sie diff Befehle wie diesen ausführen:

```
$ git diff 32d1776b1^ 32d1776b1
```

Statt die Diff-Ausgabe auf der Kommandozeile zu erhalten, wird P4Merge von Git gestartet, was so ähnlich wie folgt aussieht:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Date Finder</title>
<meta http-equiv="content-type" content="text/html; c
<link rel="stylesheet" href="/stylesheets/application
<!-- javascript_include_tag 'prototype', 'effects' -->
</head>
<body>

Date Finder

<form onsubmit="return false;">
<#> text_field_tag('date') <#>
</form>

<p id="out">...</p>

<div id="footer">please contact us at support@github.co
<script type="text/javascript">

new Form.Element.Observer(
'date',
0.5,
function(el, value){
new Ajax.Request('/main/chronic/' + value, {
method: 'get',
onSuccess: function(transport){
$(el).innerHTML = transport.responseText;
}
});
}
)
</script>
</body>

```

Figure 142. P4Merge

Wenn Sie versuchen, zwei Branches zu verschmelzen und dabei Merge-Konflikte haben, können Sie den Befehl `git mergetool` ausführen. Er startet P4Merge, damit Sie diese Konflikte über das GUI-Tool lösen können.

Das Tolle an diesem Wrapper-Setup ist, dass Sie Ihre Diff- und Merge-Tools einfach ändern können. Um beispielsweise Ihre Tools `extDiff` und `extMerge` so zu ändern, dass das KDiff3-Tool stattdessen ausgeführt wird, müssen Sie nur Ihre `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 Sie die cmd-Konfiguration einrichten müssen. Um eine Liste der unterstützten Tools anzusehen, versuchen Sie 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 Sie nicht daran interessiert sind, KDiff3 für diff zu verwenden, sondern es nur für die Merge-Auflösung verwenden wollen und der Befehl kdiff3 in Ihrem Pfad steht, dann können Sie Folgendes ausführen:

```
$ git config --global merge.tool kdiff3
```

Wenn Sie diese Methode verwenden, 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 schwierigeren 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 Ihre 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 Sie unter Windows programmieren und mit Leuten arbeiten, die andere Systeme verwenden (oder umgekehrt), werden Sie 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 Sie eine Datei zum Index hinzufügen, und umgekehrt, wenn es Code auf Ihr Dateisystem auscheckt. Sie können diese Funktionen mit der `core.autocrlf` Einstellung einschalten. Wenn Sie sich auf einem Windows-Computer befinden, setzen Sie die Einstellung auf `true` – das konvertiert LF-Endungen in CRLF, wenn Sie Code auschecken:

```
$ git config --global core.autocrlf true
```

Auf einem Linux- oder MacOS-System, das LF-Zeilenden verwendet, sollten Sie nicht zulassen, dass Git Dateien automatisch konvertiert, wenn Sie sie auschecken. Falls jedoch versehentlich eine Datei mit CRLF-Endungen hinzugefügt wird, können Sie dafür sorgen, dass Git sie korrigiert. Sie können Git anweisen, CRLF beim Commit in LF zu konvertieren, aber nicht umgekehrt, indem Sie `core.autocrlf` auf `input` setzen:

```
$ git config --global core.autocrlf input
```

Dieses Setup sollte Ihnen CRLF-Endungen in Windows-Checkouts geben, aber LF-Endungen auf MacOS- und Linux-Systemen und im Repository.

Wenn Sie ein Windows-Programmierer sind, der ein reines Windows-Projekt durchführt, dann können Sie diese Funktionalität deaktivieren und die Carriage Returns im Repository aufzeichnen, indem Sie den Konfigurationswert auf `false` setzen:

```
$ git config --global core.autocrlf false
```

## core.whitespace

Git wird ursprünglich mit einer Voreinstellung zur Erkennung und Behebung einiger Leerzeichen-Probleme gestartet. 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.

Sie können mit Git bestimmen, welche dieser Optionen aktiviert werden sollen. Setzen Sie dazu, getrennt durch Kommas, `core.whitespace` auf den gewünschten Wert (ein oder aus). Sie können eine Option deaktivieren, indem Sie ein `-` (Minus-Zeichen) dem Namen voranstellen, oder den Standardwert verwenden, indem Sie ihn ganz aus der Zeichenkette der Einstellung entfernen. Wenn Sie z.B. möchten, dass alles außer `space-before-tab` gesetzt wird, können Sie 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 Sie können 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 Sie einen `git diff` Befehl ausführen und versuchen sie einzufärben, damit Sie sie gegebenenfalls vor der Übertragung beheben können. Es wird diese Werte auch verwenden, um Ihnen zu helfen, wenn Sie Patches mit `git apply` einspielen. Wenn Sie Patches installieren, können Sie Git bitten, Sie zu warnen, wenn es Patches mit den angegebenen Leerzeichen-Problemen anwendet:

```
$ git apply --whitespace=warn <patch>
```

Sie können auch Git versuchen lassen, das Problem automatisch zu beheben, bevor Sie den Patch einspielen:

```
$ git apply --whitespace=fix <patch>
```

Diese Optionen gelten auch für den Befehl `git rebase`. Wenn Sie Leerzeichen-Probleme committet haben, aber noch nicht zum Upstream geschoben haben, können Sie `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 Sie beachten sollten.

## 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 Repositories oder Pushes. Wenn Sie möchten, dass Git bei jedem Push die Objektkonsistenz überprüft, können Sie es dazu zwingen, indem Sie `receive.fsckObjects` auf true setzen:

```
$ git config --system receive.fsckObjects true
```

Jetzt prüft Git die Integrität Ihres Repositorys, noch bevor jeder Push akzeptiert wird, um sicherzustellen, dass fehlerhafte (oder böswillige) Clients keine schädlichen Daten eingeben.

## receive.denyNonFastForwards

Wenn Sie Commits rebasieren, die Sie bereits gepusht haben, und dann versuchen, erneut zu pushen, oder anderweitig versuchen, einen Commit an einen Remote-Branch zu senden, der nicht den Commit enthält, auf den der Remote-Zweig aktuell zeigt, werden Sie abgelehnt. Im Allgemeinen ist das eine gute Richtlinie. Bei dem Rebase können Sie festlegen den Remote-Branch mit einem `-f` Flag in Ihrem Push-Befehl zu aktualisieren, wenn Sie wissen, was Sie tun.

Um Git anzuweisen, Force-Pushes abzulehnen, setzen Sie `receive.denyNonFastForwards`:

```
$ git config --system receive.denyNonFastForwards true
```

Die andere Möglichkeit ist, dass Sie das über serverseitige Receive-Hooks tun, die wir im weiteren Verlauf behandeln werden. Dieser Ansatz ermöglicht es Ihnen, 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 nach oben pusht. Um das zu verhindern, setzen Sie `receive.denyDeletes` auf true:

```
$ git config --system receive.denyDeletes true
```

Dadurch wird das Löschen von Branches oder Tags verhindert – kein User darf das anwenden. Um dann Remote-Banches zu entfernen, müssen Sie die ref-Dateien manuell vom Server entfernen. Es gibt weitere interessante Möglichkeiten, das auf Benutzerebene über ACLs zu realisieren, wie Sie in [Beispiel für Git-forcierte Regeln](#) erfahren werden.

# 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 Ihrer Verzeichnisse (normalerweise dem Stammverzeichnis Ihres Projekts) oder in der `.git/info/attributes` Datei festgelegt, falls Sie nicht wollen, dass die Attributdatei mit Ihrem Projekt verknüpft wird.

Mit Hilfe von Attributen können Sie 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 Sie ihn in Git ein- oder auschecken. In diesem Abschnitt erfahren Sie mehr über einige der Attribute, die Sie in Ihrem Git-Projekt auf Ihre Pfade setzen können, und sehen einige Beispiele für die praktische Anwendung dieser Funktion.

## Binäre Dateien

Ein toller Kniff, für den Sie Git-Attribute verwenden können, ist das Erkennen von binären Dateien (falls es sonst nicht möglich ist, es herauszufinden) und Git speziell anzugeben, wie man mit diesen Dateien umgeht. So können beispielsweise einige Textdateien maschinell erzeugt und nicht mehr unterschieden werden, während andere Binärdateien unterschieden werden können. Sie werden sehen, wie Sie Git mitteilen können, welche die richtige 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, was im Grunde genommen ein JSON-Datensatz (Klartext JavaScript Datenformat) ist, der von der IDE auf die Festplatte geschrieben wird, Ihre Build-Einstellungen aufzeichnet usw. Obwohl es sich technisch gesehen um eine Textdatei handelt (weil sie eigentlich komplett UTF-8 ist), wollen Sie sie nicht als solche behandeln, denn es handelt sich hierbei um eine sehr einfache Datenbank – Sie können den Inhalt nicht mergen, wenn zwei Personen ihn ändern sind Unterschiede im Allgemeinen nicht hilfreich. Die Datei ist für den internen Betrieb auf einem Rechner bestimmt. Im Prinzip sollten Sie sie wie eine Binärdatei behandeln.

Um Git anzugeben, alle `.pbxproj` Dateien als Binärdaten zu behandeln, fügen Sie die folgende Zeile zu Ihrer `.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 Sie `git show` oder `git diff` bei Ihrem Projekt ausführen.

### Unterschiede in Binärdateien vergleichen

Sie können auch die Git-Attribut-Funktionalität verwenden, um Binärdateien effektiv zu unterscheiden. Dafür müssen Sie Git mitteilen, wie es Ihre Binärdateien in ein Textformat konvertieren soll, das über das normale `diff` verglichen werden kann.

Zuerst werden Sie diese Technik nutzen, um eines der läufigsten Probleme zu lösen, die es gibt: die Versionskontrolle von Microsoft Word-Dokumenten. Jeder weiß, dass Word der grauenhafteste

Editor überhaupt ist und dennoch wird er von allen benutzt. Wenn Sie Word-Dokumente versionieren wollen, können Sie sie in ein Git-Repository legen und ab und zu committen; aber was nützt das? Wenn Sie `git diff` wie gewohnt starten, sehen Sie 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
```

Sie können zwei Versionen nicht direkt vergleichen, es sei denn, Sie checken sie aus und scannen sie manuell, richtig? Wie sich herausstellt, kann man das ziemlich gut mit Git-Attributten machen. Fügen Sie die folgende Zeile in Ihre `.gitattributes` Datei ein:

```
*.docx diff=word
```

Hiermit erfährt Git, dass jede Datei, die mit diesem Suchmuster übereinstimmt (`.docx`), den Filter „word“ verwenden sollte, wenn Sie einen Diff anzeigen lassen wollen, der Änderungen enthält. Was ist der Filter „word“? Sie müssen ihn einrichten. Nachfolgend konfigurieren Sie Git so, dass es das Programm `docx2txt` verwendet, um Word-Dokumente in lesbare Textdateien zu konvertieren, die es dann richtig auswertet.

Zuerst müssen Sie `docx2txt` installieren. Sie können es von <https://sourceforge.net/projects/docx2txt> herunterladen. Folgen Sie den Anweisungen in der `INSTALL` Datei, um es an einen Ort zu platzieren, an dem Ihre Shell es finden kann. Als nächstes schreiben Sie ein Wrapper-Skript, um die Ausgabe in das von Git erwartete Format zu konvertieren. Erstellen Sie eine Datei namens `docx2txt`, irgendwo innerhalb Ihres Pfads und fügen Sie diesen Inhalt hinzu:

```
#!/bin/bash  
docx2txt.pl "$1" -
```

Vergessen Sie nicht, die Dateirechte mit `chmod a+x` zu ändern. Schließlich können Sie 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 Ihrer Word-Dateien, die Sie leichter unterscheiden können.

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 offensichtlich, dass es funktioniert.

Ein weiteres spannendes Problem, das Sie auf diese Weise lösen können, ist die Unterscheidung von Bilddateien. Die eine Methode besteht darin, Bilddateien durch einen Filter zu leiten, der ihre EXIF-Informationen extrahiert – Metadaten, die mit den meisten Bildformaten aufgezeichnet werden. Nach dem Herunterladen und Installieren des **exiftool** Programms können Sie damit die Metadaten Ihrer Bilder in Text umwandeln. So zeigt Ihnen das Diff zumindest eine schriftliche Darstellung der vorgenommenen Änderungen an. Fügen Sie die folgende Zeile in Ihre **.gitattributes** Datei ein:

```
*.png diff=exif
```

So konfigurieren Sie Git, um dieses Tool zu verwenden:

```
$ git config diff.exif.textconv exiftool
```

Wenn Sie ein Bild in Ihrem Projekt ersetzen und dann `git diff` ausführen, sehen Sie 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
```

Sie können leicht erkennen, dass sich sowohl die Dateigröße als auch die Bildgröße geändert haben.

## 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 Sie eine Datei mit Informationen über einen Commit nach dem Committen nicht ändern können, da zuerst die Prüfsumme der Datei von Git ermittelt wird. Wenn die Datei ausgecheckt ist können Sie jedoch Text in sie einfügen und ihn wieder entfernen, bevor er zu einem Commit hinzugefügt wird. Die Git-Attribute stellen Ihnen dafür zwei Möglichkeiten zur Verfügung.

Erstens können Sie 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. Fügen Sie die folgende Zeile in Ihre `.gitattributes` Datei ein:

```
*.txt ident
```

Hinzufügen einer `$Id$` Referenz zu einer Testdatei:

```
$ 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 Sie die Ersetzung von Schlüsselwörtern in CVS oder Subversion verwendet haben, können Sie einen Datums-Stempel hinzufügen – der SHA-1 ist nicht allzu hilfreich, weil er eher willkürlich ist und Sie nicht erkennen können, ob ein SHA-1 älter oder neuer als ein anderer ist, den Sie sich ansehen.

Es zeigt sich, dass Sie Ihre eigenen Filter schreiben können, um Ersetzungen in Dateien auf einem Commit/Checkout durchzuführen. Diese werden als „saubere“ (engl. clean) und „verschmutzte“ (engl. smudge) Filter bezeichnet. In der Datei `.gitattributes` können Sie 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.

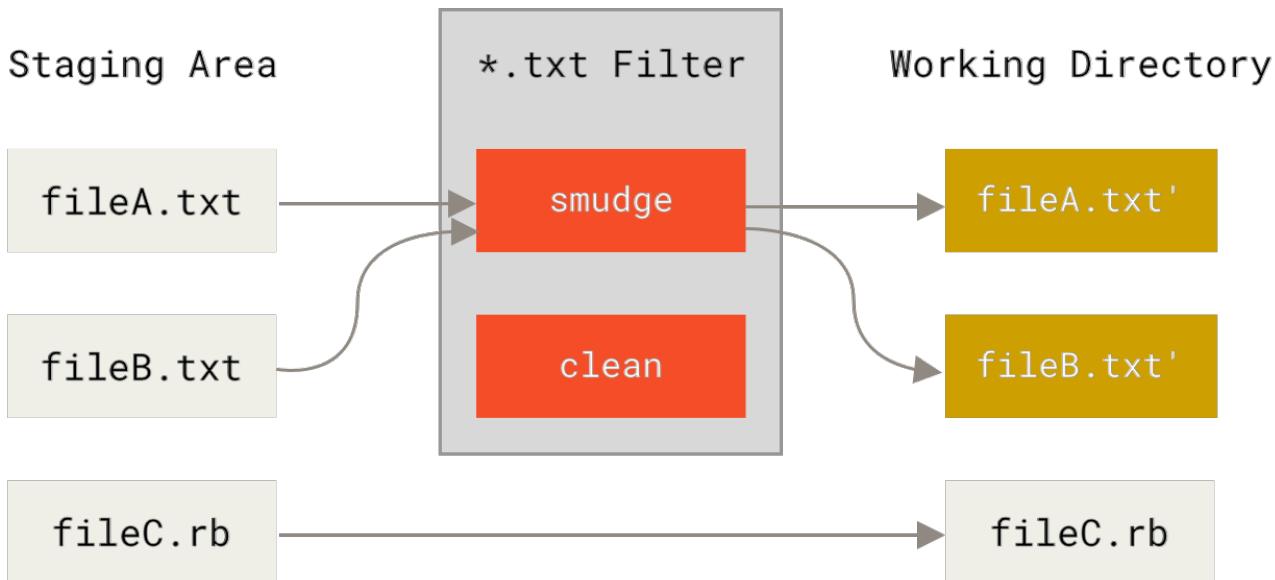


Figure 143. Der „smudge“ Filter wird beim Auschecken ausgeführt

Figure 144. Der „clean“ Filter wird ausgeführt, wenn Dateien zum Commit vorgemerkt werden

Die ursprüngliche Commit-Meldung dieser Funktion zeigt ein einfaches Anwendungsbeispiel, wie Sie Ihren gesamten C-Quellcode vor dem Commit durch das `indent` Programm laufen lassen können. Sie können es so einrichten, dass das Filterattribut in Ihrer `.gitattributes` Datei so gesetzt ist, dass `*.c` Dateien mit dem Filter „indent“ gefiltert werden:

```
*.c filter=indent
```

Dann weisen Sie 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 Sie in diesem Fall Dateien committen, 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, benötigen Sie ein kleines Skript, das einen Dateinamen anlegt, 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 übertragen, die es in stdin erkennt und die Ergebnisse auszugeben. Es sollte einfach sein, es in der Programmiersprache zu erstellen mit der Sie am besten zurechtkommen. Sie können diese Datei `expand_date` nennen und in Ihren Pfad aufnehmen. Jetzt müssen Sie einen Filter in Git einrichten (nennen Sie ihn z.B. `dater`) und ihm sagen, dass er Ihren `expand_date` Filter verwenden soll, um die Dateien beim Auschecken zu bearbeiten (`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 Ihr Filter jetzt fertig ist, können Sie ihn testen, indem Sie ein Git-Attribut für diese Datei einrichten, das den neuen Filter aktiviert und eine Datei mit Ihrem `$Date$` Schlüsselwort erstellen:

```
date*.txt filter=dater
```

```
$ echo '# $Date$' > date_test.txt
```

Wenn Sie diese Änderungen committen und die Datei erneut auschecken, sehen Sie das Schlüsselwort ordnungsgemäß ersetzt:

```
$ git add date_test.txt .gitattributes  
$ git commit -m "Testing date expansion in Git"  
$ rm date_test.txt  
$ git checkout date_test.txt  
$ cat date_test.txt  
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Sie sehen, wie leistungsfähig diese Technik für maßgeschneiderte Anwendungen sein kann. Sie müssen 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 Sie diese Filter entwerfen, sollten sie, bei fehlerhaften Ergebnissen, problemlos in der Lage sein das Projekt trotzdem einwandfrei laufen zu lassen.

## Exportieren Ihres Repositorys

Mit den Daten von Git-Attribute können Sie interessante Aufgaben beim Export eines Archivs Ihres Projekts erledigen.

### export-ignore

Sie können Git anweisen, bestimmte Dateien oder Verzeichnisse nicht zu exportieren, wenn Sie ein Archiv erstellen. Wenn es ein Unterverzeichnis oder eine Datei gibt, die Sie nicht in Ihre Archivdatei aufnehmen möchten, aber in Ihr Projekt einchecken möchten, können Sie diese Dateien über das Attribut `export-ignore` bestimmen.

Angenommen, Sie haben einige Testdateien in einem `test/` Unterverzeichnis und es ist nicht sinnvoll, sie in den Tarball-Export Ihres Projekts aufzunehmen. Man kann die folgende Zeile zu der Datei mit den Git-Attributen hinzufügen:

```
test/ export-ignore
```

Wenn Sie nun `git archive` ausführen, um einen Tarball Ihres Projekts zu erstellen, wird dieses Verzeichnis nicht in das Archiv aufgenommen.

### export-subst

Beim Exportieren von Dateien für die Verteilung können Sie die Formatierung und Schlüsselwort-Erweiterung von `git log` auf ausgewählte Teile von Dateien anwenden, die mit dem Attribut `exportsubst` markiert sind.

Wenn Sie beispielsweise eine Datei mit dem Namen `LAST_COMMIT` in Ihr Projekt aufnehmen möchten und Metadaten über den letzten Commit automatisch in das Projekt eingespeist werden sollen,

sobald `git archive` läuft, können Sie beispielsweise Ihre `.gitattributes` und `LAST_COMMIT` Dateien so einrichten:

```
LAST_COMMIT export-subst
```

```
$ echo 'Last commit date: $Format:%cd by %aN$' > LAST_COMMIT
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Wenn Sie `git archive` ausführen, 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 die Verwendung geeignet, aber wie jedes andere exportierte Archiv ist es nicht zur weiteren Entwicklungsarbeit verwendbar.

## Merge-Strategien

Sie können Git-Attribute auch verwenden, um Git anzulegen, verschiedene Merge-Strategien für spezifische Dateien in Ihrem Projekt zu verwenden. Eine sehr nützliche Option ist es, Git anzulegen, dass es nicht versuchen soll, bestimmte Dateien zusammenzuführen, wenn sie Konflikte haben, sondern Ihre Version der Daten zu benutzen, anstelle der von jemand anderem.

Das ist nützlich, wenn ein Branch in Ihrem Projekt auseinander gelaufen ist oder sich spezialisiert hat. Wenn Sie jedoch Änderungen wieder in ihn einfügen möchten und einige Dateien ignorieren möchten. Angenommen, Sie haben eine Datenbank-Einstellungsdatei namens `database.xml`, die sich

in zwei Branches voneinander unterscheidet. Sie wollen in Ihrer zweiten Branch verschmelzen, ohne die Datenbankdatei zu beschädigen. Dann können Sie so ein Attribut einrichten:

```
database.xml merge=ours
```

Dann definieren Sie die Dummy-Merge-Strategie `ours` mit:

```
$ git config --global merge.ours.driver true
```

Wenn Sie in den zweiten Branch verschmelzen, ohne in Merge-Konflikte mit der Datei `database.xml` zu laufen, sehen Sie so etwas wie hier:

```
$ 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 serverseitig. Client-seitige Hooks werden durch Operationen wie Commit und Merging ausgelöst, während serverseitige Hooks bei Netzwerkoperationen wie dem Empfangen von Push-Commits ausgeführt werden. Sie können diese Hooks aus den unterschiedlichsten Gründen verwenden.

### Einen Hook installieren

Alle Hooks werden im Unterverzeichnis `hooks` des Git-Verzeichnisses gespeichert. In den meisten Projekten ist das `.git/hooks`. Wenn Sie ein neues Repository mit `git init` einrichten, füllt Git das `hooks`-Verzeichnis mit einer Reihe von Beispielskripten, von denen viele für sich allein genommen nützlich sind. Sie dokumentieren aber auch die Input-Werte jedes Skripts. Alle Beispiele sind als shell-Skripte verfasst, wobei in einigen Perl eingebaut ist, aber alle richtig benannten ausführbaren Skripte funktionieren gut – Sie können sie in Ruby oder Python oder einer beliebigen anderen Sprache schreiben, mit der Sie vertraut sind. Wenn Sie die mitgelieferten Hook-Skripte verwenden möchten, müssen Sie sie umbenennen. Ihre Dateinamen enden alle mit `.sample`.

Fügen Sie zum Aktivieren eines Hook-Skripts eine Datei in das `hooks` Unterverzeichnis Ihres `.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-



Beachten Sie, dass clientseitige Hooks **nicht** kopiert werden, wenn Sie ein Repository klonen. Wenn Sie mit diesen Skripten beabsichtigen, die Einhaltung einer Richtlinie durchzusetzen, werden Sie das wahrscheinlich auf der Serverseite machen wollen; sehen Sie sich 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 Sie eine Commit-Nachricht eintippen können. Er dient dazu, den Snapshot zu untersuchen, der übertragen werden soll. Sie können herausfinden, ob Sie etwas vergessen haben, ob Tests ausgeführt werden oder was immer Sie im Code überprüfen müssen. Falls ein Status ungleich Null erkannt wird, bricht der Hook den Commit ab. Sie können das aber mit `git commit --no-verify` umgehen. Sie können verschiedene Optionen ausführen, wie z.B. die Suche nach dem Codestil (`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. Sie können die Standardnachricht bearbeiten, noch bevor sie der Commit-Autor 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. Sie können es in Verbindung mit einer Commit-Vorlage verwenden, um Informationen programmgesteuert einzufügen.

Der `commit-msg` Hook übernimmt einen Parameter, der wiederum der 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, damit Sie Ihren Projektstatus oder Ihre Commit-Beschreibung überprüfen können, bevor Sie einen Commit durchlaufen lassen. Im letzten Abschnitt dieses Kapitels werden wir demonstrieren, wie Sie mit diesen Hook überprüfen können, ob Ihre Commit-Beschreibung mit einem erforderlichen Muster übereinstimmt.

Nachdem der gesamte Commit-Prozess abgeschlossen ist, wird der `post-commit` Hook gestartet. Er benötigt keine Parameter, aber Sie können den letzten Commit mühelos abrufen, indem Sie `git log -1 HEAD` aufrufen. Im Allgemeinen wird dieses Skript für Benachrichtigungen oder ähnliches verwendet.

## E-Mail-Workflow-Hooks

Sie können drei clientseitige Hooks für einen E-Mail-basierten Workflow einrichten. Alle werden vom `git am` Befehl aufgerufen, so dass Sie ohne weiteres zum nächsten Abschnitt springen können, wenn Sie diesen Befehl in Ihrem Workflow nicht verwenden. Wenn Sie Patches per E-Mail erhalten, die von `git format-patch` vorbereitet wurden, dann könnten sich einige davon für Sie 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. Sie können das verwenden, um sicherzugehen, dass eine Commit-Beschreibung korrekt formatiert ist oder um die Nachricht zu normalisieren, indem Sie sie vom Skript bearbeiten lassen.

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 Ihnen den Snapshot zu untersuchen, bevor Sie den Commit durchführen. Mit diesem Skript können Sie Tests durchführen oder den Verzeichnisbaum anderweitig durchsuchen. Wenn etwas fehlt oder die Tests nicht bestanden werden, 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. Sie können ihn verwenden, um eine Gruppe oder den Autor des Patches, den Sie in den Patch eingefügt haben, darüber zu informieren, dass Sie ihn verwendet haben. Mit diesem Skript können Sie den Patch-Prozess nicht stoppen.

## Andere Client-Hooks

Der `pre-rebase` Hook läuft, noch bevor Sie etwas rebasieren und kann den Prozess stoppen, wenn Sie ihn mit einem Wert ungleich Null beenden. Sie können 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 Ihrem Workflow übereinstimmen.

Der `post-rewrite` Hook wird mit Befehlen durchgeführt, die Commits ersetzen, wie z.B. `git commit --amend` und `git rebase` (allerdings nicht mit `git filter-branch`). Sein einziges Argument 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 Sie einen erfolgreichen `git checkout` durchgeführt haben, läuft der `post-checkout` Hook. Sie können damit Ihr Arbeitsverzeichnis für Ihre Projektumgebung entsprechend einrichten. Möglicherweise bedeutet das, dass große Binärdateien bewegen müssen, deren Quellcode nicht kontrolliert werden soll, keine automatisch generierte Dokumentation benötigen oder etwas ähnliches.

Der `post-merge` Hook läuft nach einem erfolgreich abgeschlossenen `merge` Befehl. Damit können Sie Daten im Verzeichnisbaum wiederherstellen, die Git nicht überwachen kann, wie z.B. die Zugriffsrechte. Dieser Hook kann ebenfalls das Vorliegen von Dateien außerhalb der Git-Kontrolle überprüfen, die bei Änderungen im Verzeichnisbaum kopiert werden sollen.

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`. Sie können 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 Sie darüber zu informieren oder um die Speicherbereinigung abzubrechen, wenn der Zeitpunkt nicht günstig ist.

## Serverseitige Hooks

Zusätzlich zu den clientseitigen Hooks können Sie als Systemadministrator einige wichtige serverseitige Hooks verwenden, um nahezu jede Art von Richtlinien für Ihr 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. Sie können so eine 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. Sie können 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. Statt 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 einer Liste, die Benachrichtigung eines Continuous Integration Servers oder die Aktualisierung eines Ticket-Tracking-Systems. Sie können 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 abgeschlossen ist. Passen Sie daher auf, wenn Sie eine Aktion durchführen, die sehr lang dauern kann.

## Beispiel für Git-forcierte Regeln

In diesem Abschnitt werden Sie 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. Sie erstellen Client-Skripte, die den

Entwickler erkennen lassen, ob ihr Push abgelehnt wird, ebenso wie Server-Skripte, die die Einhaltung der Richtlinien erzwingen.

Die hier vorgestellten Skripte sind in Ruby geschrieben, teils wegen unserer geistigen Trägheit, teils aber auch weil Ruby leicht zu lesen ist – auch wenn man es nicht unbedingt selbst schreiben kann. Allerdings kann jede beliebige Programmiersprache verwendet werden. Alle mit Git vertriebenen Beispiel-Hook-Skripte sind entweder in Perl oder Bash verfasst. Sie können daher viele Beispiele für Hooks in diesen Sprachen verstehen, wenn Sie sich die Beispiele ansehen.

## Serverseitiger Hook

Die gesamte serverseitige Arbeit wird in die `update` Datei in Ihrem `hooks` Verzeichnis übernommen. Der `update` Hook läuft einmal pro gepushtem Branch und benötigt drei Argumente:

- Der Name der Referenz, zu der gepusht wird
- Die frühere Version, in der sich dieser Branch befindet
- Die neue Version, die gepusht werden soll

Wenn der Push über SSH ausgeführt wird, haben Sie auch Zugriff auf den Benutzer, der den Push durchführt. Auch wenn Sie es jedem erlaubt haben, sich mit einem bestimmten Benutzer (z.B. „git“) über die Public-Key-Authentifizierung zu verbinden, müssen Sie 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 Ihr 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. Bewerten Sie es nicht – es ist auf diese Weise leichter zu demonstrieren.

## Ein bestimmtes Commit-Message-Format erzwingen

Ihre erste Herausforderung besteht darin, die Einhaltung eines bestimmten Formats für jede Commit-Nachricht durchzusetzen. Nur um ein Ziel zu haben, gehen Sie davon aus, dass jede Nachricht eine Zeichenkette enthalten muss, die wie „ref: 1234“ aussieht, weil Sie möchten, dass jeder Commit auf einen Arbeitsschritt in Ihrem Ticketing-System verweist. Sie müssen sich jeden Commit ansehen, der gepusht wird, um nachzusehen, ob sich diese Zeichenkette in der Commit-Beschreibung befindet und falls die Zeichenkette bei einem der Commits fehlt, mit einem Exit-Status ungleich Null, wird der Push abgelehnt.

Sie können eine Liste der SHA-1-Werte aller Commits erhalten, die verschoben werden, indem Sie die Werte `$newrev` und `$oldrev` verwenden und sie an das Git-Basiskommando (engl. Plumbing Command) `git rev-list` übergeben. 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, können Sie etwa so vorgehen:

```
$ git rev-list 538c33..d14fc7  
d14fc7c847ab946ec39590d87783c69b031bdfb7  
9f585da4401b0a3999e84113824d15245c13f0be  
234071a1be950e2a8d078e6141f5cd20c1e61ad3  
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a  
17716ec0f1ff5c77eff40b7fe912f9f6cf0e475
```

Sie können diese Ausgabe nehmen, jeden dieser Commit SHA-1s durchlaufen, die Beschreibung dafür abgreifen und diese Beschreibung gegen einen regulären Ausdruck testen, der nach einem Zeichen-Muster sucht.

Sie müssen sich überlegen, wie Sie die Commit-Beschreibung von jedem dieser Commits zum Testen erhalten. Um die unformatierten Commit-Daten zu ermitteln, können Sie ein anderes Basiskommando namens `git cat-file` verwenden. Wir werden alle diese Basiskommandos in Kapitel 10, [Git Interna](#) im Detail betrachten – aber vorerst gibt Ihnen dieser Befehl 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  
  
changed the version number
```

Wenn Sie den SHA-1-Wert kennen ist es einfach die Commit-Beschreibung eines Commits zu erhalten. Gehen Sie in die erste leere Zeile und übernehmen alles danach. Auf Unix-Systemen können Sie mit dem `sed` Befehl arbeiten:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'  
changed the version number
```

Sie können auf diese Weise die Commit-Beschreibung von jedem Commit, der gepusht werden soll, übernehmen und mit einem Exit-Code ungleich Null beenden, wenn Sie etwas finden, das nicht übereinstimmt. Um das Skript zu verlassen und den Push abzulehnen, verlassen Sie das Skript mit ungleich Null. 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 Sie das in Ihr **update** Skript einfügen, werden Updates mit Commit-Beschreibungen, die nicht Ihrer Regel entsprechen, abgelehnt.

### Ein benutzerbasiertes ACL-System einrichten

Angenommen, Sie möchten einen Prozess hinzufügen, der eine Zugriffskontrollliste (ACL) verwendet, die festlegt, welche Benutzer Änderungen an welchen Teilen Ihrer Projekte vornehmen dürfen. Einige Personen haben vollen Zugriff, andere können Änderungen nur zu bestimmten Unterverzeichnissen oder bestimmten Dateien pushen. Um das zu erreichen, müssen Sie diese Regeln in eine **acl** Datei schreiben, die in Ihrem blanken Git-Repository auf dem Server liegt. Mit dem **update** Hook können Sie diese Regeln prüfen. Sie können 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 müssen Sie Ihre ACL-Datei erstellen. Hier verwenden Sie 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 haben Sie eine 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. Ihre ACL-Datei sieht dann wie folgt aus:

```

avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
avail|schacon|tests

```

Sie beginnen damit, diese Daten in eine Struktur zu übernehmen, die Sie verwenden können. In diesem Beispiel werden Sie, um das Beispiel einfach zu halten, nur die **avail** Anweisungen einführen. Hier folgt eine Methode, die Ihnen 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 Sie nun die Berechtigungen geklärt haben, müssen Sie ermitteln, welche Pfade die gepuschten Commits geändert haben. So können Sie sicherstellen, dass der Benutzer, der gepusht hat, Zugriff auf alle diese Pfade erhält.

Mit der Option `--name-only` des `git log` Befehls können Sie 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 Sie die verwendete ACL-Struktur, die von der `get_acl_access_data` Methode zurückgegeben wird, mit den aufgelisteten Dateien in jedem der Commits vergleichen, können Sie 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

```

Sie erhalten eine Liste der neuen Commits, die mit `git rev-list` auf Ihren Server gepusht werden. Dann stellen Sie für jeden dieser Commits fest, welche der Dateien geändert werden sollen und stellen sicher, dass der Benutzer, der den Push ausführt, Zugriff auf alle zu ändernden Pfade hat.

Jetzt können Ihre Benutzer keine Commits mit schlecht strukturierten Beschreibungen oder mit modifizierten Dateien außerhalb der zugewiesenen Pfade pushen.

## Austesten

Wenn Sie `chmod u+x.git/hooks/update` ausführen, auf der Datei, in die Sie diesen Code hätten einfügen sollen, und dann versuchen, ein Commit mit einer nicht konformen Beschreibung zu senden, dann erhalten Sie etwa 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, erkennen Sie die Position, an der der Hook startet.

```
Enforcing Policies...
(refs/heads/master) (fb8c72) (c56860)
```

Denken Sie daran, dass das ganz am Anfang Ihres Update-Skripts ausgegeben wird. Alles, was Ihr Skript an `stdout` weitergibt, wird an den Client übertragen.

Das nächste, was Sie beachten sollten, 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 Ihnen, die anderen beiden wurden von Git ausgegeben, die Ihnen mitteilten, dass das Update-Skript ungleich Null beendet wurde. Das war es, was Ihren Push verneint hat. 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'
```

Sie werden eine Nachricht des Remote für jede Referenz erhalten, die von Ihrem Hook abgelehnt wurde. Sie zeigt Ihnen an, dass sie speziell wegen eines 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-Autor 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 Ihr Repository nie eine Commit-Beschreibung ohne Ihr eigenes Schema haben, und Ihre Benutzer werden in einer „Sandbox“ eingeschlossen sein.

## Clientseitige Hooks

Der Nachteil dieses Konzepts ist das Gejammer, das unweigerlich entsteht, wenn die Commit-Pushes Ihrer 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, müssen Sie diese Skripte auf andere Weise bereitstellen. Dann müssen Ihre Benutzer sie in ihr **.git/hooks** Verzeichnis kopieren und sie ausführbar machen. Sie können diese Hooks innerhalb des Projekts oder in einem separaten Projekt weitergeben, aber Git wird sie nicht automatisch einrichten.

Am Anfang sollten Sie Ihre Commit-Beschreibung kurz vor jeder Übertragung überprüfen, damit Sie sich sicher sind, dass der Server Ihre Änderungen nicht aufgrund schlecht formatierter Commit-Beschreibungen ablehnt. Dazu können Sie den **commit-msg** Hook hinzufügen. Wenn Sie die Nachricht aus der als erstes Argument übergebenen Datei lesen und mit dem Patternmuster vergleichen lassen, können Sie 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 Sie mit einer Beschreibung committen, die nicht korrekt formatiert ist, sehen Sie das hier:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

In diesem Fall wurde kein Commit durchgeführt. Wenn Ihre Beschreibung jedoch das richtige Muster enthält, erlaubt Ihnen Git die Übertragung:

```
$ git commit -am 'test [ref: 132]'  
[master e05c914] test [ref: 132]  
1 file changed, 1 insertions(+), 0 deletions(-)
```

Als nächstes sollten Sie sicherstellen, dass Sie keine Dateien ändern, die sich außerhalb Ihres ACL-Bereichs befinden. Wenn das `.git` Verzeichnis Ihres Projekts eine Kopie der ACL-Datei enthält, die Sie zuvor verwendet haben, dann wird das folgende `pre-commit` Skript diese Einschränkungen für Sie durchsetzen:

```
#!/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 Ihrem Arbeitsverzeichnis und nicht aus Ihrem `.git` Verzeichnis läuft. Sie müssen 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 Sie eine Liste der Dateien erhalten, die geändert wurden. Da die serverseitige Methode das Protokoll der Commits betrachtet und der Commit an dieser Stelle noch nicht aufgezeichnet wurde, müssen Sie stattdessen Ihre Dateiliste aus dem Staging-Area holen. Anstelle von

```
files_modified = 'git log -1 --name-only --pretty=format:'' #{ref}'
```

müssen Sie das 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 Sie lokal mit dem gleichen Benutzer arbeiten, den Sie auf dem Remotesystem verwenden. Wenn das anders ist, müssen Sie 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, müssen Sie entweder über einen Commit hinaus rebasieren, den Sie bereits hochgeladen haben 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 sie abfangen können 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 Sie gerade neu schreiben wollen, und prüft, ob sie in einer Ihrer Remote-Referenzen vorhanden sind. Wenn es einen findet, der von einer Ihrer 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. Sie erhalten eine Liste der Commits, die bereits gepusht wurden, wenn Sie diese Anweisung aufrufen:

```
'git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}'
```

Die [SHA<sup>^@</sup>](#) Syntax löst alle Vorgänger (engl. parents) dieses Commits auf. Sie suchen nach jedem Commit, der vom letzten Commit auf dem Remote aus erreichbar ist und der von keinem Parent einer der SHA-1s, die Sie 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 Sie nicht versuchen, den Push mit [-f](#) zu erzwingen, wird der Server Sie warnen und den Push nicht akzeptieren. Es ist aber ein interessanter Test und kann Ihnen theoretisch helfen, einen Rebbase zu vermeiden, den Sie später vielleicht wieder zurücknehmen und reparieren müssen.

## Zusammenfassung

Wir haben die meisten der häufigsten Möglichkeiten besprochen, mit denen Sie Ihren Git-Client und -Server so anpassen können, dass er am besten zu Ihrem Workflow und Ihren Projekten passt. Sie haben die verschiedensten Konfigurations-Einstellungen, dateibasierte Attribute und Ereignis-Hooks kennengelernt und einen Beispiel-Server zur Einhaltung von Regeln erstellt. In der Regel können Sie Git nun an nahezu jeden Workflow anpassen, den Sie sich vorstellen können.

# Git und andere Systeme

Die Welt ist nicht perfekt. Normalerweise können Sie nicht jedes Projekt, mit dem Sie in Berührung kommen, 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 Sie gerade arbeiten, in einem anderen System läuft.

Irgendwann werden Sie vielleicht Ihr bestehendes Projekt in Git umwandeln wollen. Der zweite Teil dieses Kapitels behandelt die Migration Ihres 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 Sie am ehesten in der „freien Wildbahn“ begegnen werden.

## 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 können Sie Git als geeigneter Client für einen Subversion-Server verwenden, so dass Sie alle lokalen Funktionen von Git nutzen und dann auf einen Subversion-Server pushen können, als ob Sie Subversion lokal einsetzen würden. Das bedeutet, dass Sie lokale Branching- und Merging-Aktivitäten vornehmen, die Staging Area nutzen, Rebasing- und Cherry-Picking-Aktivitäten durchführen können, während Ihre Mitstreiter weiterhin in ihrer dunklen und altertümlichen Umgebung tätig sind. Es ist eine gute Möglichkeit, Git in die Unternehmensumgebung einzuschleusen, Ihren Entwicklerkollegen zu helfen, effizienter zu werden und gleichzeitig die Infrastruktur so zu ändern, um Git vollständig zu unterstützen. Die Subversion-Brücke ist das Portal zur DVCS-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 Sie bei der Verwendung von `git svn` mit Subversion interagieren, einem System, das ganz anders funktioniert als Git. Obwohl Sie lokales Branching und Merging durchführen können, ist es im Allgemeinen ratsam, Ihren Verlauf so linear wie möglich zu gestalten, indem Sie Ihre Arbeiten rebasen. Vermeiden Sie dabei auch die gleichzeitige Interaktion

mit einem Git Remote-Repository.

Schreiben Sie Ihren Verlauf nicht um und versuchen Sie nicht, erneut zu pushen. Pushen Sie 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 leicht zu verwirren. Wenn Sie mit einem Team arbeiten und einige verwenden SVN und andere Git, stellen Sie sicher, dass alle den SVN-Server für die gemeinsame Arbeit verwenden – das erleichtert Ihnen den Alltag.

## Einrichtung

Um diese Funktionalität zu demonstrieren, benötigen Sie ein typisches SVN-Repository, auf das Sie Schreibzugriff haben. Wenn Sie die Beispiele kopieren möchten, müssen Sie eine beschreibbare Kopie eines SVN-Test Repository erstellen. Um das einfach zu realisieren, können Sie das Tool `svnsync` verwenden, das in Subversion enthalten ist.

Um weiter zu machen, müssen Sie zunächst ein neues lokales Subversion-Repository erstellen:

```
$ mkdir /tmp/test-svn  
$ svnadmin create /tmp/test-svn
```

Aktivieren Sie 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
```

Sie können dieses Projekt nun auf Ihrem lokalen Rechner synchronisieren, indem Sie `svnsync init` mit den Repositorys to und from aufrufen.

```
$ svnsync init file:///tmp/test-svn \  
http://your-svn-server.example.org/svn/
```

Dadurch werden die Eigenschaften für die Ausführung der Synchronisierung festgelegt. Sie können den Code dann klonen, indem Sie Folgendes ausführen:

```
$ 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 Sie versuchen, 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 unvorstellbar ineffizient, aber es ist der einzige einfache Weg, das zu erreichen.

## Erste Schritte

Jetzt, da Sie ein Subversion-Repository haben, auf das Sie Schreibrechte haben, können Sie einen typischen Work-Flow absolvieren. Beginnen Sie mit dem Befehl `git svn clone`, der ein ganzes Subversion-Repository in ein lokales Git-Repository importiert. Beachten Sie, dass Sie beim Import aus einem echten Subversion-Repository hier `file:///tmp/test-svn` durch die URL Ihres Subversion-Repository ersetzen sollten:

```
$ 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 Ihnen 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 buchstäblich Stunden oder gar Tage dauern, das zu vollenden.

Der Teil `-T trunk -b branches -t tags` teilt Git mit, dass dieses Subversion-Repository den grundlegenden Branching- und Tagging-Konventionen folgt. Wenn Sie Ihren Trunk, Ihre Branches oder Tags anders benennen, können sich diese Optionen ändern. Da dies so häufig vorkommt, können Sie 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 sollten Sie über ein gültiges Git-Repository verfügen, das Ihre 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
```

Beachten Sie, 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 es von einem Git-Server kront. 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 fetcht die Tags direkt in [refs/tags](#), anstatt sie mit entfernten Branches zu verknüpfen.

## Zurück zu Subversion committen

Jetzt, da Sie ein Arbeitsverzeichnis haben, können Sie etwas an dem Projekt arbeiten und Ihre Commits wieder zum Upstream pushen, indem Sie Git als SVN-Client verwenden. Wenn Sie eine der Dateien bearbeiten und übertragen, haben Sie 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 müssen Sie Ihre Änderung zum Upstream pushen. Beachten Sie, wie sich dies auf Ihre

Arbeitsweise mit Subversion auswirkt – Sie können mehrere Commits offline durchführen und diese dann alle auf einmal auf den Subversion-Server übertragen. Um zu einem Subversion-Server zu pushen, führen Sie 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
```

Dabei werden alle Commits ausgeführt, die Sie oberhalb des Subversion Server-Codes gemacht haben, dafür jeweils einen eigenen Subversion-Commit und dann Ihren lokaler Git-Commit umgeschrieben, um einen eindeutigen Identifier einzufügen. Das ist wichtig, weil es bedeutet, dass sich alle SHA-1-Prüfsummen für Ihre Commits ändern. Aus diesem Grund ist es keine gute Idee, gleichzeitig mit Git-basierten Remotes Ihrer Projekte und einem Subversion-Server zu arbeiten. Wenn Sie sich den letzten Commit ansehen, sehen Sie 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 Sie die Daten übertragen haben, nun mit `95e0222` beginnt. Wenn Sie sowohl auf einen Git-Server als auch auf einen Subversion-Server pushen möchten, müssen Sie zuerst auf den Subversion-Server pushen (`dcommit`), da diese Aktion Ihre Commit-Daten ändert.

## Neue Änderungen pullen

Wenn Sie mit anderen Entwicklern zusammenarbeiten, dann wird irgendwann einer von Ihnen pushen, und andere versuchen, eine Änderung voranzutreiben, die Konflikte verursacht. Diese Änderung wird abgelehnt, bis Sie deren Arbeit mergen. 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, können Sie die `git svn rebase` ausführen, das alle Änderungen auf dem Server, die Sie noch nicht haben, pullt und alle ihre lokalen Arbeiten an die Spitze zum Server neu überträgt (engl. rebase):

```
$ 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 Ihre gesamte Arbeit an der Spitze des Subversion-Servers, so dass Sie `dcommit` erfolgreich einsetzen können:

```
$ 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 Sie Upstream-Arbeiten, die Sie noch nicht lokal haben, zuerst mergen, bevor Sie pushen können, zwingt Sie `git svn` dazu nur dann, wenn die Änderungen im Konflikt stehen (ähnlich wie bei Subversion). Wenn jemand anderes eine Änderung an einer Datei vorantreibt und Sie eine Änderung an einer anderen Datei vorantreiben, funktioniert Ihr `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: a0253d06732169107aa020390d9fefd2b1d92806 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 Ihrer 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 können Sie den Zustand auf Ihrem Client-System vor der Veröffentlichung vollständig testen, während Sie in SVN nie sicher sein können, dass die Zustände unmittelbar vor dem Commit und nach dem Commit identisch sind.

Sie sollten diesen Befehl auch ausführen, um Änderungen vom Subversion-Server einzubinden, auch wenn Sie nicht bereit sind, selbst zu committen. Es ist ratsam, `git svn fetch` auszuführen, um die neuen Daten zu holen, aber `git svn rebase` übernimmt den Fetch und aktualisiert dann Ihre 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 Sie ab und zu `git svn rebase` laufen lassen, stellen Sie sicher, dass Ihr Code immer auf dem neuesten Stand ist. Sie sollten jedoch überprüfen, ob Ihr Arbeitsverzeichnis sauber ist, wenn Sie diese Funktion auslösen. Wenn Sie lokale Änderungen haben, müssen Sie Ihre Arbeit entweder verstecken (engl. `stash`) oder temporär committen, bevor Sie `git svn rebase` ausführen – andernfalls wird der Befehl angehalten, wenn er erkennt, dass das Rebase zu einem Merge-Konflikt führen wird.

## Git Branching Probleme

Sobald Sie sich mit einem Git-Workflow vertraut gemacht haben, werden Sie wahrscheinlich Topic-Banches erstellen, an ihnen arbeiten und sie dann verschmelzen (mergen). Wenn Sie über `git svn` auf einen Subversion-Server pushen, können Sie Ihre Arbeit jedes Mal auf einen einzigen Branch rebasieren, 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, Ihr Verlauf sieht wie folgt aus: Sie haben einen `experiment` Branch erstellt, zwei Commits durchgeführt und diese dann wieder mit dem `master` zusammengeführt. Wenn Sie `dcommit` aufrufen, 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 Sie sich Ihre Git-Projekt-Historie ansehen, er hat keinen der Commits, die Sie auf dem `experiment` Branch gemacht haben, 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 hineingedrückt wurde, als ob Sie `git merge --squash` ausgeführt hätten; man sieht die Commit-Daten nicht, woher sie stammen oder wann sie committed wurden.

## Subversion Branching

Branching in Subversion ist nicht dasselbe wie Branching in Git. Es ist wahrscheinlich das Beste, wenn Sie es so oft vermeiden wie möglich. Sie können aber mit `git svn` in Subversion Branches anlegen und dorthin committen.

### Creating a New SVN Branch

Um einen neuen Branch in Subversion zu erstellen, führen Sie `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. Wichtig dabei ist, dass er Sie nicht in diesen Branch auscheckt; wenn Sie an dieser Stelle committen, wird dieser Commit in den `trunk` auf dem Server gehen, nicht in `at this point, that commit will go to trunk on the server, not opera`.

### Aktive Branches wechseln

Git findet heraus, in welchen Branch Ihre dcommits gehen, indem es nach der Spitze eines Ihrer Subversion Branches in Ihrem Verlauf sucht – Sie sollten nur einen haben, und es sollte der Letzte sein, der eine `git-svn-id` in Ihrem aktuellen Branchverlauf hat.

Falls Sie an mehr als einem Branch gleichzeitig arbeiten möchten, können Sie lokale Branches einrichten, um `dcommit` auf bestimmte Subversion Branches zu beschränken, indem Sie diese beim importierten Subversion Commit für diesen Branch starten. Einen `opera` Branch, in dem Sie separat bearbeiten können, können Sie starten mit:

```
$ git branch opera remotes/origin/opera
```

Um Ihren `opera` Branch in `trunk` (Ihren `master` Branch) zu mergen, können Sie das mit einem normalen `git merge` machen. Aber Sie sollten unbedingt eine beschreibende Commit-Meldung (via `-m`) angeben, sonst wird beim Merge anstelle von etwas Vernünftigem „Merge branch `opera`“ angezeigt.

Obwohl Sie für diese Operation `git merge` verwenden und der Merge wahrscheinlich viel einfacher ist als in Subversion (da Git automatisch die entsprechende Merge-Basis für Sie erkennt), ist es kein

normaler Git Merge-Commit. Sie müssen diese Daten an einen Subversion-Server zurück pushen, der keinen Commit mit mehr als einem Elternteil verarbeiten kann; nachdem Sie ihn zum Server gepusht haben, sieht er also aus wie ein einzelner Commit, der die gesamte Arbeit eines anderen Branchs unter einem einzigen Commit zusammenfasst. Nachdem Sie einen Branch in einem anderen zusammengeführt haben, können Sie nicht einfach zurückgehen und an diesem Branch weiterarbeiten, wie Sie es normalerweise in Git tun. Das `dcommit` Kommando, das Sie ausführen, löscht alle Informationen, die zeigen, in welchen Branch zusammengeführt wurde, so dass nachfolgende Berechnungen der Merge-Basis falsch sind – `dcommit` lässt Ihr `git merge` Ergebnis aussehen, als ob Sie `git merge --squash` ausgeführt hätten. Leider gibt es keine gute Methode, diese Situation zu vermeiden – Subversion kann diese Informationen nicht speichern, daher werden Sie immer von den Einschränkungen des Systems behindert, während Sie es als Ihren Server verwenden. Um Fehler zu vermeiden, sollten Sie den lokalen Branch (in diesem Fall `opera`) löschen, nachdem Sie ihn in `trunk` eingefügt haben.

## 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 Sie von Subversion aus kennen. Wir haben hier ein paar Befehle, mit denen Sie das bekommen, was Subversion vorher konnte.

### Verlauf im SVN-Format

Wenn Sie an Subversion gewöhnt sind und Ihren Verlauf im SVN-Stil sehen möchten, können Sie `git svn log` ausführen, um Ihren 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
```

Sie sollten 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 Ihnen nur Commits an, die zum Subversion-Server übertragen wurden. Lokale Git-Commits, die Sie noch nicht mit `dcommit` bestätigt haben, 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 Anmerkung

So wie der Befehl `git svn log` den Befehl `svn log` offline simuliert, können Sie das Äquivalent von `svn annotate` abrufen, indem Sie `git svn blame [FILE]` ausführen. 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 schacon Committing in git-svn.
78 schacon
2 temporal To build and install the C++ Protocol Buffer runtime and the Protocol
2 temporal Buffer compiler (protoc) execute the following:
2 temporal
```

Noch einmal, zur Wiederholung! Auch hier werden keine Commits angezeigt, die Sie lokal in Git gemacht haben oder die in der Zwischenzeit in Subversion verschoben wurden.

## SVN Server-Information

Wenn Sie `git svn info` ausführen, können Sie die gleiche Art von Informationen erhalten, die Ihnen `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 Sie ein Subversion-Repository klonen, in dem irgendwo `svn:ignore` Eigenschaften gesetzt sind, werden Sie wahrscheinlich entsprechende `.gitignore` Dateien setzen wollen, damit Sie nicht versehentlich Dateien übertragen, die Sie nicht sollten. `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 Sie erstellt, damit sie bei Ihrem nächsten Commit berücksichtigt werden.

Der zweite Befehl ist `git svn show-ignore`, der die Zeilen nach stdout ausgibt, die Sie in eine `.gitignore` Datei einfügen müssen, damit Sie die Ausgabe in die Ausschlussdatei Ihres Projekts umleiten können:

```
$ git svn show-ignore > .git/info/exclude
```

Auf diese Weise überhäufen Sie das Projekt nicht mit `.gitignore` Dateien. Das ist eine gute Option, wenn Sie der einzige Git-Benutzer in einem Subversion-Team sind und Ihre Teamkollegen keine `.gitignore` Dateien im Projekt haben wollen.

## git svn Zusammenfassung

Die `git svn` Tools sind nützlich, wenn Sie mit dem Subversion-Server feststecken oder sich anderweitig in einer Entwicklungsumgebung befinden, die den Betrieb eines Subversion-Servers erfordert. Sie sollten es jedoch als verkümmertes Git betrachten, oder Sie werden Probleme in der Umsetzung haben, die Sie und Ihre Mitwirkenden verwirren könnten. Um keine Schwierigkeiten zu bekommen, versuchen Sie sich an diese Hinweise zu halten:

- Führen Sie einen linearen Git-Verlauf, der keine Merge-Commits von `git merge` enthält. Rebasieren Sie alle Arbeiten, die Sie außerhalb Ihres Haupt-Branchs durchführen, wieder in diesen ein; mergen Sie sie nicht.
- Richten Sie keinen separaten Git-Server ein und arbeiten Sie nicht mit einem zusammen. Möglicherweise haben Sie einen, um Klone für neue Entwickler zu starten, aber pushen Sie nichts, was nicht über einen `git-svn-id` Eintrag verfügt. Sie können 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 Sie diese Leitlinien befolgen, kann die Arbeit mit einem Subversion-Server leichter umsetzbar sein. Mit einem Umstieg auf einen echten Git-Server kann Ihr 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 Sie Gits clientseitiges Verhalten bevorzugen, aber mit einem Projekt arbeiten, 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-Helfer 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 müssen Sie `git-remote-hg` installieren. Im Wesentlichen geht es darum, die Datei irgendwo in Ihrem 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 Ihrem `$PATH`. Git-remote-hg hat noch eine weitere Abhängigkeit: die `mercurial` Library für Python. Wenn Sie Python installiert schon haben, ist das einfach:

```
$ pip install mercurial
```

(Wenn Sie Python noch nicht installiert haben, besuchen Sie <https://www.python.org/> und besorgen Sie es sich zuerst.)

Als Letztes brauchen Sie den Mercurial-Client. Gehen Sie zu <https://www.mercurial-scm.org/> und installieren Sie ihn, falls Sie es noch nicht getan haben.

Jetzt sind Sie bereit zu abrocken. Alles, was Sie benötigen, ist ein Mercurial-Repository, auf das Sie zugreifen können. 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 Sie sehen werden, sind diese beiden Systeme ähnlich genug, dass es keine große Überschneidungen 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 Sie sehen, 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-Helfer). 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 Zweigen.

Die Datei `notes/hg` ist der Ausgangspunkt dafür, wie git-remote-hg Git-Commit-Hashes auf Mercurial-Changeset-IDs abbildet. Lassen Sie 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 an der Spitze der `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: Die Auslassungen. Mercurial und Git verwenden dafür einen sehr ähnlichen Mechanismus, aber es ist durchaus möglich, dass Sie eine `.gitignore` Datei nicht wirklich in ein Mercurial Repository übertragen wollen. 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 Sie es nur kopieren müssen:

```
$ 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 Sie sind 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 Sie einen Blick auf das Mercurial-Repository werfen, werden Sie feststellen, dass genau das getan wurde, was wir erwarten durften:

```
$ 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 höher gewichtet. Der Branch, auf den ein Changeset durchgeführt wird, wird *zusammen mit dem Changeset* 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
```

Achten Sie darauf, dass die Zeile mit „branch“ beginnt. Git kann das nicht wirklich nachahmen (und muss es auch nicht; beide Arten von Zweigen 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 machen Sie:

```
$ 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 -6
@ 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

```

Beachten Sie den neuen **[featureA]** Tag auf Revision 5. Die verhalten sich genau wie Git-Branche auf der Git-Seite, mit einer Ausnahme: Sie können ein Lesezeichen auf der Git-Seite nicht löschen (das ist eine Einschränkung des Remote-Helpers).

Sie können auch an einem „schwergewichtigen“ Mercurial-Branch arbeiten: Bringen Sie 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 Sie es üblicherweise machen würden. Eine Sache, die Sie wissen sollten, ist, dass Mercurial das Überschreiben der Historie nicht unterstützt, sondern nur hinzufügt. Das Mercurial-Repository sieht nach einem interaktiven Rebase 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 Ihre Teamkollegen, die Mercurial verwenden, **sehr verwirrend** sein, also versuchen Sie es zu vermeiden.

## Mercurial Zusammenfassung

Git und Mercurial sind sich ähnlich genug, um über die eigene Umgebung hinaus schmerzlos zu arbeiten. Wenn Sie vermeiden, den Verlauf zu ändern, der Ihren Computer verlässt (was allgemein empfohlen wird), merken Sie wahrscheinlich nicht einmal, dass am anderen Ende Mercurial verwendet wird.

## Git und Bazaar

Unter den DVCSS ist **Bazaar** ein weiterer bedeutender Vertreter. Bazaar ist freie Software, Open-Source und ist Teil des [GNU-Projekts](#). Es verhält sich ganz anders als Git. Manchmal muss man, um

das Gleiche wie bei Git machen zu können, ein anderes Schlüsselwort verwenden. Einige gängige Schlüsselwörter haben nicht die gleiche Bedeutung. Insbesondere das Branch-Management ist sehr verschieden und kann zu Verwirrung und Missverständnissen führen, vor allem, wenn jemand aus dem Umfeld von Git kommt. Dennoch ist es von Git aus möglich, an einem Bazaar-Repository zu arbeiten.

Es gibt viele Projekte, die es Ihnen ermöglichen, Git als Bazaar-Client zu nutzen. Hier werden wir das Projekt von Felipe Contreras verwenden, das Sie unter <https://github.com/felipec/git-remote-bzr> finden können. Um es zu installieren, müssen Sie nur die Datei git-remote-bzr in einen Ordner herunterladen, der sich in Ihrem Pfad (**\$PATH**) befindet:

```
$ wget https://raw.github.com/felipec/git-remote-bzr/master/git-remote-bzr -O ~/bin/git-remote-bzr  
$ chmod +x ~/bin/git-remote-bzr
```

Außerdem müssen Sie Bazaar installiert haben. Das ist alles!

### **Erstellen eines Git-Repository aus einem Bazaar-Repository**

Die Bedienung ist einfach. Es genügt, ein Bazaar-Repository zu klonen, dem **bzr::** vorangestellt ist. Da Git und Bazaar beide Vollklone auf Ihrem Computer erstellen, ist es möglich, einen Git-Klon an Ihren lokalen Bazaar-Klon anzuhängen, es wird aber nicht empfohlen. Es ist viel einfacher, Ihren Git-Klon direkt an den gleichen Ort zu hängen, an dem Ihr Bazaar-Klon hängt – das zentrale Repository.

Angenommen, Sie haben mit einem Remote-Repository gearbeitet, das sich unter der Adresse **bzr+ssh://developer@mybazaarserver:myproject** befindet. Dann müssen Sie es wie folgt klonen:

```
$ git clone bzr::bzr+ssh://developer@mybazaarserver:myproject myProject-Git  
$ cd myProject-Git
```

An diesem Punkt wird Ihr Git-Repository erstellt, aber es ist nicht für eine optimale Festplattennutzung komprimiert. Deshalb sollten Sie auch Ihr Git-Repository bereinigen und komprimieren, vor allem wenn es ein großes ist:

```
$ git gc --aggressive
```

### **Bazaar Branches**

Bazaar erlaubt es Ihnen nur, Branches zu klonen, aber ein Repository kann mehrere Branches enthalten, und **git-remote-bzr** kann beides klonen. Um zum Beispiel einen Branch zu klonen:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs/trunk emacs-trunk
```

Und um das gesamte Repository zu klonen:

```
$ git clone bzr::bzr://bzr.savannah.gnu.org/emacs emacs
```

Der zweite Befehl klont alle Branches, die im emacs-Repository enthalten sind; es ist jedoch möglich einige Branches hervorzuheben:

```
$ git config remote-bzr.branches 'trunk, xwindow'
```

Einige Remote-Repositorys erlauben es Ihnen nicht, ihre Branches aufzulisten, in diesem Fall müssen Sie sie manuell angeben, und obwohl Sie die Konfiguration im Klon-Befehl angeben könnten, könnten Sie das leichter feststellen:

```
$ git init emacs  
$ git remote add origin bzr::bzr://bzr.savannah.gnu.org/emacs  
$ git config remote-bzr.branches 'trunk, xwindow'  
$ git fetch
```

### Ignorieren, was mit `.bzrignore` ignoriert wird

Da Sie an einem mit Bazaar verwalteten Projekt arbeiten, sollten Sie keine `.gitignore` Datei erstellen, da Sie diese versehentlich unter Versionskontrolle setzen könnten und die anderen mit Bazaar arbeitenden Benutzer dadurch gestört würden. Die Lösung besteht darin, die `.git/info/exclude` Datei entweder als symbolischen Link oder als normale Datei zu erstellen. Wir werden später sehen, wie wir dieses Problem lösen können.

Bazaar verwendet das gleiche Modell wie Git, um Dateien zu ignorieren, hat aber auch zwei Funktionen, die kein Äquivalent in Git haben. Die vollständige Beschreibung finden Sie in der [Dokumentation](#). Die beiden Merkmale sind:

1. „!!“ ermöglicht es Ihnen, bestimmte Dateimuster zu ignorieren, auch wenn sie mit einer „!“-Regel angegeben werden.
2. „RE:“ am Anfang einer Zeile erlaubt es Ihnen, einen [regulären Python-Ausdruck](#) anzugeben (Git erlaubt nur Shell Globs).

Folglich sind zwei verschiedene Situationen zu prüfen:

1. Wenn die Datei `.bzrignore` keines dieser beiden spezifischen Präfixe enthält, dann können Sie einfach einen symbolischen Link darauf im Repository setzen: `ln -s .bzrignore .git/info/exclude`.
2. Ansonsten müssen Sie die Datei `.git/info/exclude` erstellen und anpassen, um genau die gleichen Dateien in `.bzrignore` zu ignorieren.

Was auch immer der Fall ist, Sie müssen auf jede Änderung von `.bzrignore` achten, um sicherzustellen, dass die Datei `.git/info/exclude` immer `.bzrignore` widerspiegelt. Wenn sich die Datei `.bzrignore` ändert und eine oder mehrere Zeilen enthält, die mit „!!“ oder „RE:“ beginnen, muss die Datei `.git/info/exclude` so angepasst werden, dass sie die gleichen Dateien ignoriert, wie die, die mit `.bzrignore` ignoriert werden. Wenn die Datei `.git/info/exclude` ein symbolischer Link

war, müssen Sie außerdem zuerst den symbolischen Link löschen, `.bzignore` nach `.git/info/exclude` kopieren und diese dann anpassen. Seien Sie jedoch vorsichtig bei der Erstellung, da es mit Git unmöglich ist, eine Datei wieder einzubinden, wenn ein übergeordnetes Verzeichnis dieser Datei ausgeschlossen ist.

## Fetchen der Änderungen aus dem Remote-Repository

Um die Änderungen des Remote zu fetchen, pullen Sie die Änderungen wie gewohnt mit Hilfe von Git-Befehlen. Angenommen, Ihre Änderungen befinden sich im `master` Branch, mergen oder rebasieren Sie Ihre Arbeit auf den `origin/master` Branch:

```
$ git pull --rebase origin
```

## Ihre Arbeit zum Remote-Repository pushen

In Bazaar ist das Konzept der Merge-Commits ebenfalls vorhanden, so dass es kein Problem geben wird, wenn Sie einen Merge Commit pushen. So können Sie an einem Branch arbeiten, Änderungen in `master` zusammenführen und Ihre Arbeit pushen. Dann erstellen Sie Ihre Branches, testen und committen Ihre Arbeit wie gewohnt. Schließlich pushen Sie Ihre Arbeit in das Bazaar-Repository:

```
$ git push origin master
```

## Vorhalte/Einschränkungen

Das Remote-Helper-Framework von Git hat einige gültige Beschränkungen. Vor allem funktionieren diese Befehle nicht:

- `git push origin :branch-to-delete` (Bazaar kann auf diese Weise keine Referenzen löschen)
- `git push origin alt:neu` (es wird *alt* pushen)
- `git push --dry-run origin branch` (es wird pushen)

## Zusammenfassung

Die Modelle von Git und Bazaar sind sehr ähnlich, so dass es beim Arbeiten über die Grenzen keinen großen Aufwand erfordert. Solange Sie auf die Einschränkungen achten und sich immer bewusst sind, dass das Remote-Repository nicht nativ Git ist, werden Sie damit umgehen können.

## Git und Perforce

Perforce ist ein sehr beliebtes Versionskontrollsystem in Unternehmungen. 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 Sie immer mit einem einzigen zentralen Server verbunden sind 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 wirklich besser geeignet ist.

Es gibt zwei Möglichkeiten, wenn Sie die Verwendung von Perforce und Git kombinieren möchten.

Als erstes stellen wir die „Git Fusion“ Bridge des Herstellers von Perforce vor, mit der Sie Teilbäume Ihres Perforce-Depots als Read-Write-Git-Repository freigeben können. Bei der zweiten handelt es sich um git-p4, eine client-seitige Bridge, mit der Sie Git als Perforce-Client verwenden können, ohne dass der Perforce-Server neu konfiguriert werden muss.

## Git Fusion

Preforce bietet mit Git Fusion ein Produkt (verfügbar unter <http://www.perforce.com/git-fusion>), das einen Perforce-Server mit Git-Repositories 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. Sie können das Image der virtuellen Maschine von <http://www.perforce.com/downloads/Perforce/20-User> herunterladen. Wenn der Download abgeschlossen ist, importieren Sie es in Ihre bevorzugte Virtualisierungssoftware (wir verwenden VirtualBox).

Beim ersten Start des Rechners werden Sie aufgefordert, das Passwort für drei Linux-Benutzer (`root`, `perforce`, `git`) festzulegen und einen Instanznamen anzugeben, mit dem Sie diese Installation von anderen im selben Netzwerk unterscheiden können. Wenn das alles abgeschlossen ist, werden Sie das sehen:

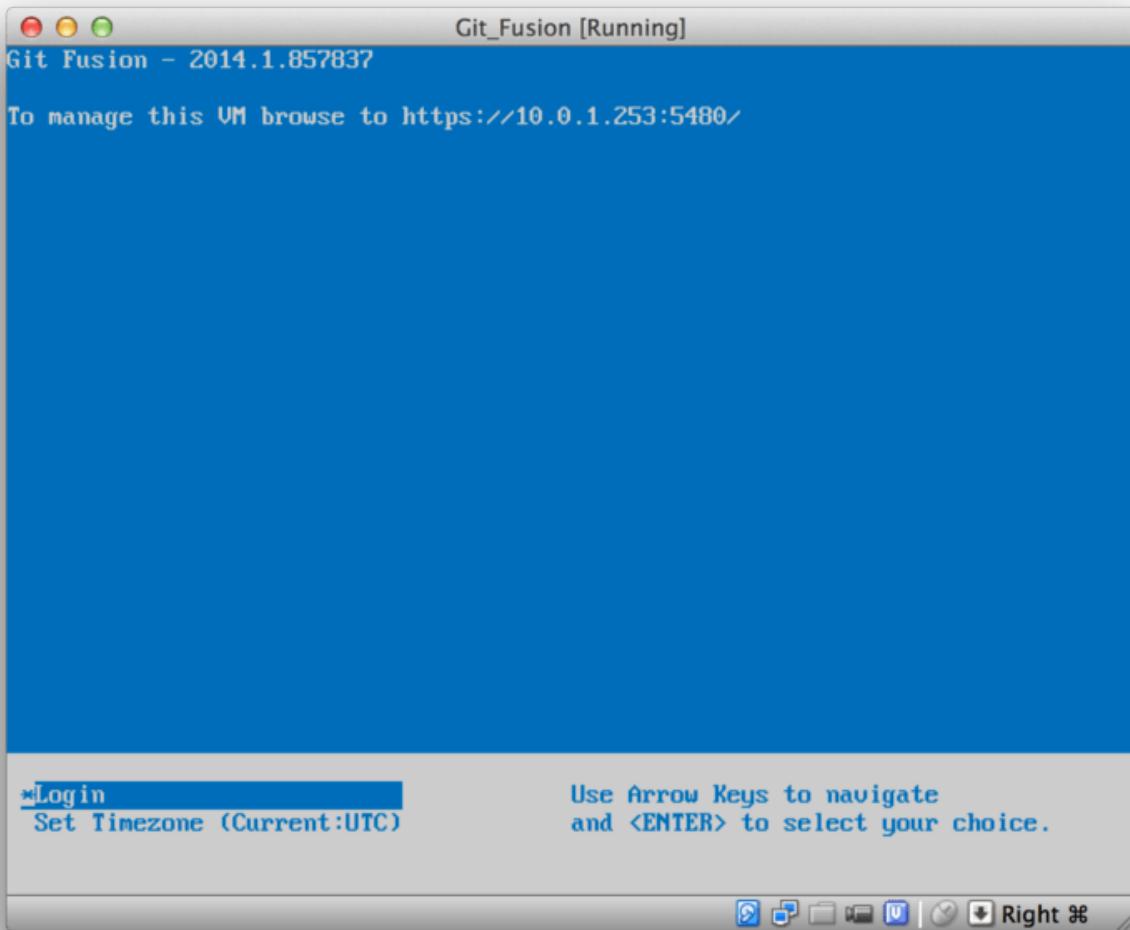


Figure 145. Boot-Bildschirm der virtuellen Maschine von Git Fusion

Sie sollten die hier angezeigte IP-Adresse notieren, wir werden sie später benutzen. Als nächstes erstellen wir einen Perforce-Benutzer. Wählen Sie unten die Option „Login“ und drücken Sie die Eingabetaste (oder verbinden Sie sich per SSH mit dem Computer) und melden Sie sich als **root** an. Verwenden Sie dann diese 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, Sie können auch die Vorgaben übernehmen, indem Sie **:wq** eingeben und auf Enter drücken. Der zweite wird Sie auffordern, ein Passwort zweimal einzugeben. Das ist alles, was wir mit einem Shell-Prompt zu tun haben werden, also beenden Sie die Sitzung.

Als nächstes müssen Sie 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 Ihrer virtuellen Maschine übereinstimmt, weshalb Git die HTTPS-Verbindung abweisen würde. Wenn dies eine permanente Installation sein soll, lesen Sie 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 Sie klonen können. 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 Sie Git Fusion installiert haben, sollten Sie die Konfiguration anpassen. Mit Ihrem favorisierten Perforce-Client ist das ganz einfach. Weisen Sie das Verzeichnis **//.git-fusion** auf dem Perforce-Server einfach Ihrem 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 Sie sich mit nichts darin herumschlagen müssen. 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 dieser Flags eingehen, aber bedenken Sie, 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 Sie diese Datei öffnen, sehen Sie einen Abschnitt `[@repo]` mit einigen Einstellungen, die sich von den globalen Standardeinstellungen unterscheiden. Sie werden 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-Zweig und einem Git-Zweig. Der Abschnitt kann beliebig benannt werden, solange der Name eindeutig ist. Der `git-branch-name` ermöglicht Ihnen, 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 Ihr normales Workspace-Mapping Änderungen in der Struktur der Verzeichnisse enthält, können Sie 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 Sie möglicherweise nicht einmal benötigen. Bei der Konvertierung von 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 beachten Sie 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 Sie Git-Commits unter mehreren verschiedenen E-Mail-Adressen erstellt haben (oder E-Mail-Adressen ändern), 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 Sie ein internes Projekt open-source-fähig machen wollen, aber Ihr Mitarbeiterverzeichnis nicht auf der ganzen Welt veröffentlichen wollen. Beachten Sie, dass die E-Mail-Adressen und vollständigen Namen eindeutig sein sollten, es sei denn, Sie möchten 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 abgebildet 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 Sie das zum ersten Mal vornehmen, 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 Sie sehen können, sieht unser Repository genauso aus wie jedes andere Git-Repository, mit dem Sie arbeiten könnten. Es gibt drei Branches. Git hat einen lokalen **master** Branch erstellt, der **origin/master** trackt. 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 lassen Sie 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! Sie würden es aus dieser Sicht nicht erkennen, aber der **6afeb15** Commit wurde mit einem Perforce Client erstellt. Es sieht aus der Perspektive von Git aus 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: Perforce: Processing will continue even if connection is closed.
remote: Perforce: 100% (3/3) Copying changelists...
remote: Perforce: Submitting new Git commit objects to Perforce: 4
To https://10.0.1.254/Jam
    6afeb15..89cba2b master -> master

```

Git 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:

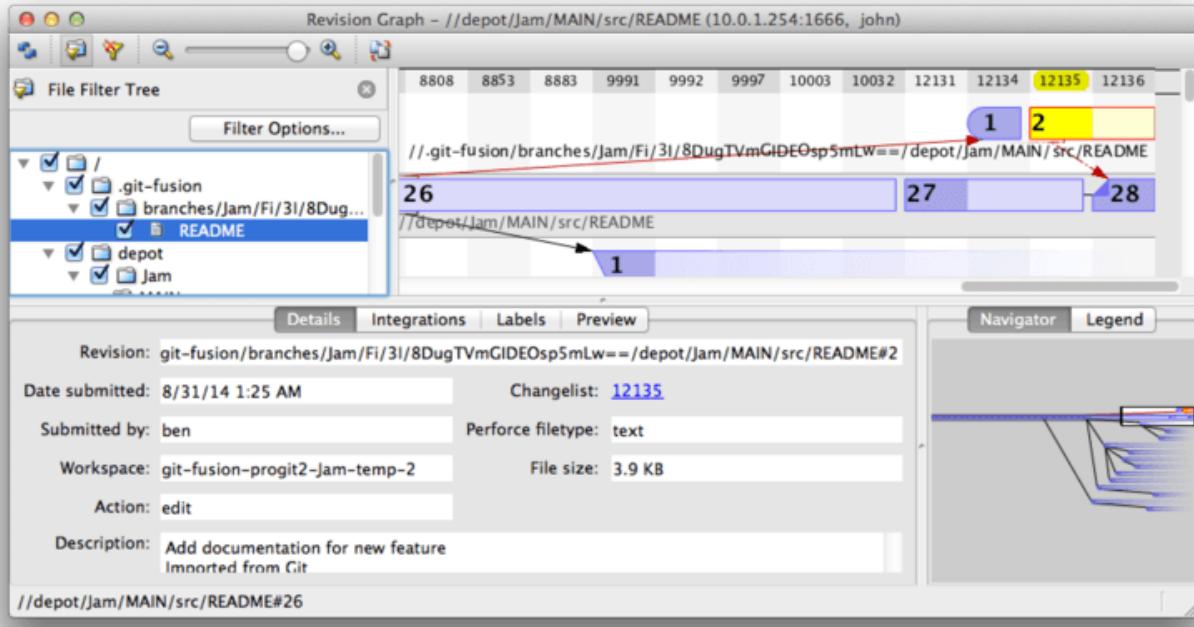


Figure 146. Perforce-Revisionsgraph resultierend aus dem Git-Push

Wenn Sie diese Darstellung noch nie zuvor gesehen haben, mag sie verwirrend erscheinen, aber sie zeigt die gleichen Konzepte wie ein grafischer Viewer für die Git-Historie. Wir betrachten die Geschichte 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.

Auffallend 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-Banches, die keinem benannten Perforce-Branch entsprechen (Sie können 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 Sie Zugang zu Ihrem Perforce-Server haben (oder erhalten können), 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 Sie ihm zumuten, zufrieden sein wird – wenn Sie versuchen, eine bereits gepushte Historie neu zu schreiben, wird Git Fusion sie ablehnen – aber Git Fusion gibt sich sehr große Mühe, sich nativ anzufühlen. Sie können sogar Git-Submodule verwenden (obwohl sie für Perforce-Anwender seltsam aussehen werden) und Branches verschmelzen (das wird als Integration auf der Perforce-Seite erfasst).

Wenn Sie den Administrator Ihres Servers nicht davon überzeugen können, 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 Ihrem Git-Repository, so dass Sie keinen Zugriff auf den Perforce-Server benötigen (mit Ausnahme der Benutzer-Anmeldeinformationen). Git-p4 ist nicht so flexibel und keine Komplettlösung wie Git Fusion, aber es ermöglicht Ihnen, das meiste von dem zu tun, was Sie tun möchten, ohne die Serverumgebung zu beeinträchtigen.



Sie brauchen das **p4** Tool an einer beliebigen Stelle in Ihrem **PATH**, um mit git-p4 zu arbeiten. Zur Zeit ist es unter <http://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, müssen Sie 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. Denken Sie 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
```

Beachten Sie, 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, lassen Sie uns ein paar Arbeiten erledigen. Nehmen wir an, Sie haben einige Fortschritte bei einem sehr wichtigen Feature gemacht und sind bereit, es dem Rest Ihres 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 gearbeitet 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
```

Es sieht so aus, als ob die Commits vorhanden wären, die sich in `master` und `p4/master` aufgeteilt hätten. 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 Sie Ihre Commits rebasieren 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 können Sie das an der Ausgabe erkennen, denn `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, unsere Änderungen in Perforce wieder einzureichen. 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 Sie bei der Ausführung von `p4 submit` sehen würden. Ausgenommen sind die Dinge am Ende, die git-p4 sinnvollerweise mit aufgenommen hat. Git-p4

versucht, Ihre Git- und Perforce-Einstellungen individuell zu berücksichtigen, wenn es einen Namen für ein Commit- oder Changeset angeben muss. In einigen Fällen wollen Sie ihn jedoch überschreiben. Wenn beispielsweise der Git-Commit, den Sie importieren, von einem Mitwirkenden geschrieben wurde, der kein Perforce-Benutzerkonto hat, können Sie trotzdem wollen, dass sich das daraus ergebende Changeset so aussieht, als hätte er es geschrieben (und nicht Sie).

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 ist, als ob wir gerade ein `git push` gemacht hätten, was die nächstliegende Analogie zu dem ist, was tatsächlich passiert ist.

Beachten Sie, dass während dieses Prozesses jeder Git-Commit in einen Perforce Changeset umgewandelt wird. Wenn Sie ihn in einen einzelnen Changeset zusammenfassen möchten, können Sie dies mit einem interaktiven Rebase tun, bevor Sie `git p4 submit` ausführen. Bitte bedenken Sie

auch, 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 Sie versuchen, einen Merge-Commit einzubinden? Versuchen wir es einmal. Das ist die Situation, in die wir uns selbst gebracht haben:

```
$ 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 zusätzlich zu einem einzelnen Changeset auf der Perforce-Seite beizufügen. Schauen wir mal, was passieren würde, wenn wir versuchen würden, 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**, mit dem versucht wird, zu beschreiben, was passieren würde, wenn der submit-Befehl wirklich ausgeführt würde. 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, mal sehen, wie es ausfällt:

```
$ 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 rebasiert, was auch tatsächlich passiert ist. So können Sie auf der Git-Seite selbst Branches erstellen, darauf arbeiten, verwerfen und mergen, ohne befürchten zu müssen, dass Ihr Verlauf sich mit Perforce nicht mehr verträgt. Wenn Sie einen Rebase durchführen können, ist es auch möglich, ihn zu einem Perforce-Server beizusteuern.

## Branching

Wenn Ihr Perforce-Projekt mehrere Branches hat, haben Sie Glück gehabt. git-p4 kann damit so umgehen, dass es sich wie Git anfühlt. Angenommen, Ihr Perforce-Depot ist so aufgebaut:

```
//depot  
└── project  
    ├── main  
    └── dev
```

Nehmen wir weiter an, Sie haben 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
| * a43cfac Populate //depot/project/main/... //depot/project/dev/....
|
* 2b83451 Project init

```

Beachten Sie 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 Sie an einem Projekt mit einer langen Historie arbeiten, 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), können Sie git-p4 mitteilen, wie die Zuordnung der Branches zu sein hat. Sie erhalten 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 machen, 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 Sie ein großes Projekt haben und an mehr als einem Branch arbeiten wollen, müssen Sie für jeden Branch, den Sie einreichen möchten, ihn einmal mit **git p4 clone** erstellen.

Für die Erstellung oder Integration von Branches müssen Sie 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 daran zu erinnern, dass Perforce für den Quellcode verantwortlich ist und Sie Git nur für die lokale Arbeit verwenden. Seien Sie einfach sehr vorsichtig bei der Weitergabe von Git-Commits. Wenn Sie einen Remote haben, den auch andere Benutzer verwenden, pushen Sie keine Commits, die zuvor noch nicht an den Perforce-Server übertragen wurden.

Möchten Sie die Verwendung von Perforce und Git als Clients für die Versionskontrolle frei kombinieren, dann müssen Sie den Server-Administrator davon überzeugen, Git zu installieren. Git Fusion macht die Verwendung von Git zu einem erstklassigen Versionskontroll-Client für einen Perforce-Server.

## Git und TFS

Git wird bei Windows-Entwicklern immer populärer. Falls Sie Code unter Windows schreiben, ist es durchaus möglich, dass Sie den Team Foundation Server (TFS) von Microsoft verwenden. TFS ist eine Collaboration-Suite, die Fehler- und Workitem-Tracking, Prozess-Support für Scrum (und andere), Code-Review und Versionskontrolle umfasst. Es gibt da ein wenig Verwirrung im Vorfeld: **TFS** ist der Server, der die Kontrolle des Quellcodes sowohl mit Git als auch mit einem eigenen, benutzerdefinierten VCS unterstützt, das Microsoft **TFVC** (Team Foundation Version Control) genannt hat. Die Git-Anbindung ist ein neueres Feature für TFS (ausgeliefert mit Version 2013). Daher beziehen sich alle Tools, die älter sind, auf den Versionskontrollteil als „TFS“, auch wenn sie hauptsächlich mit TFVC arbeiten.

Wenn Sie sich in einem Team befinden, das TFVC verwendet, aber lieber Git als Ihren Versionskontroll-Client verwenden möchten, dann gibt es dafür ein Projekt für Sie.

## Welches Tool?

Tatsache: Es gibt zwei: git-tf und git-tfs.

Git-tfs (finden Sie unter <https://github.com/git-tfs/git-tfs>) ist ein .NET-Projekt und läuft (derzeit) nur unter Windows. Um mit Git-Repositories zu arbeiten, verwendet es die .NET-Bindings für libgit2, eine library-orientierte Umsetzung von Git, die hochperformant ist und viel Flexibilität in den Tiefen eines Git-Repositories ermöglicht. Libgit2 ist keine vollständige Implementierung von Git, weshalb git-tfs den Kommandozeilen-Git-Client für einige Operationen aufrufen wird. Es gibt von daher keine expliziten Einschränkungen, was mit den Git-Repositories gemacht werden kann. Die Unterstützung von TFVC-Funktionen ist sehr ausgereift, da es Visual-Studio-Assemblies für Serveroperationen verwendet. Das erfordert den Zugriff auf diese Assemblies und bedeutet, dass Sie eine aktuelle Version von Visual Studio (jede Edition seit Version 2010, einschließlich Express seit Version 2012) oder das Visual Studio SDK installieren müssen.

Git-tf (zu Hause unter <https://gittf.codeplex.com>) ist ein Java-Projekt und läuft als solches auf jedem Computer mit einer Java-Laufzeitumgebung. Die Schnittstelle zu Git-Repositories erfolgt über JGit (eine Git-Implementierung von JVM (Java Virtual Machine)), was bedeutet, dass es praktisch keine Einschränkungen in Bezug auf die Git-Funktionen gibt. Die Unterstützung für TFVC ist jedoch im Vergleich zu git-tfs begrenzt – es werden beispielsweise keine Branches unterstützt.

So hat jedes Tool seine Vor- und Nachteile. Es gibt viele Situationen, eine der beiden gegenüber der anderen zu bevorzugen. Wir werden die prinzipielle Verwendung der beiden in diesem Buch behandeln.



Sie benötigen Zugriff auf ein TFVC-basiertes Repository, um diesen Anweisungen zu folgen. In der „freien Wildbahn“ sind sie nicht so zahlreich wie Git- oder Subversion-Repositorys, so dass Sie möglicherweise selbst eines erstellen müssen. Codeplex (<https://www.codeplex.com>) oder Visual Studio Online (<https://visualstudio.microsoft.com>) sind beide dafür eine gute Wahl.

### Erste Schritte: `git-tf`

Als Erstes werden Sie, wie bei jedem Git-Projekt, klonen. So sieht das mit `git-tf` aus:

```
$ git tf clone https://tfs.codeplex.com:443/tfs/TFS13 $/myproject/Main project_git
```

Das erste Argument ist die URL einer TFVC-Kollektion, das zweite folgt der Form `$/project/branch` und das dritte ist der Pfad zum lokalen Git-Repository, das erstellt werden soll (letzteres ist optional). Git-tf kann nur mit einem einzigen Branch gleichzeitig arbeiten. Falls Sie Checkins auf einem anderen TFVC Branch durchführen wollen, müssen Sie einen neuen Clone aus diesem Branch erstellen.

Dadurch entsteht ein voll funktionsfähiges Git-Repository:

```
$ cd project_git
$ git log --all --oneline --decorate
512e75a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Checkin message
```

Das nennt man einen *flachen* Klon, d.h. es wurde nur der letzte Changeset heruntergeladen. TFVC ist nicht dafür konzipiert, dass jeder Client eine vollständige Kopie der Historie hat, so dass git-tf standardmäßig nur die neueste Version erhält, was viel schneller ist.

Wenn Sie etwas Zeit haben, lohnt es sich vermutlich, die gesamte Projekt-Historie mit der Option `--deep` zu kopieren:

```
$ git tf clone https://tfss.codeplex.com:443/tfs/TFS13 $/myproject/Main \
  project_git --deep
Username: domain\user
Password:
Connecting to TFS...
Cloning $/myproject into /tmp/project_git: 100%, done.
Cloned 4 changesets. Cloned last changeset 35190 as d44b17a
$ cd project_git
$ git log --all --oneline --decorate
d44b17a (HEAD, tag: TFS_C35190, origin_tfs/tfs, master) Goodbye
126aa7b (tag: TFS_C35189)
8f77431 (tag: TFS_C35178) FIRST
0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Beachten Sie die Tags mit Bezeichnungen wie [TFS\\_C35189](#). Diese Eigenschaft hilft Ihnen zu erkennen, welche Git-Commits mit TFVC Change-Sets verknüpft sind. Das ist eine elegante Art, diese anzuzeigen, da Sie mit einem einfachen Log-Befehl sehen können, welcher Ihrer Commits mit einem Snapshot verbunden ist, der auch in TFVC existiert. Sie sind nicht notwendig (Sie können sie mit `git config git-tf.tag false` deaktivieren) – git-tf behält die echten Commit-Changeset-Mappings in der `.git/git-tf` Datei.

### **Erste Schritte: git-tfs**

Das Klonen mit Git-tfs verhält sich etwas anders. Achten Sie darauf:

```
PS> git tfs clone --with-branches \
  https://username.visualstudio.com/DefaultCollection \
  $/project/Trunk project_git
Initialized empty Git repository in C:/Users/ben/project_git/.git/
C15 = b75da1aba1ffb359d00e85c52acb261e4586b0c9
C16 = c403405f4989d73a2c3c119e79021cb2104ce44a
Tfs branches found:
- $/tfvc-test/featureA
The name of the local branch will be : featureA
C17 = d202b53f67bde32171d5078968c644e562f1c439
C18 = 44cd729d8df868a8be20438fdeeffb961958b674
```

Beachten Sie das Flag [--with-branches](#). Git-tfs ist in der Lage, TFVC-Branche auf Git-Branche abzubilden, und dieses Flag sagt ihm, dass es für jeden TFVC-Branch einen lokalen Git-Branch einrichten soll. Dieses Verfahren wird dringend empfohlen, wenn Sie jemals in TFS verzweigt oder gemerkt haben, es wird aber nicht mit einem Server funktionieren, der älter als TFS 2010 ist – vor dieser Version waren „Branches“ nur Verzeichnisse, so dass git-tfs sie nicht von normalen Ordnern unterscheiden konnte.

Werfen wir einen Blick auf das entstandene Git-Repository:

```

PS> git log --oneline --graph --decorate --all
* 44cd729 (tfv/featureA, featureA) Goodbye
* d202b53 Branched from $/tfvc-test/Trunk
* c403405 (HEAD, tfv/default, master) Hello
* b75da1a New project
PS> git log -1
commit c403405f4989d73a2c3c119e79021cb2104ce44a
Author: Ben Straub <ben@straub.cc>
Date:   Fri Aug 1 03:41:59 2014 +0000

Hello

git-tfs-id:
[https://username.visualstudio.com/DefaultCollection]$/.myproject/Trunk;C16

```

Es gibt zwei lokale Branches, `master` und `featureA`, die den Anfangspunkt des Klons darstellen (`Trunk` in TFVC) und einen untergeordneten Branch (`featureA` in TFVC). Sie können auch sehen, dass die `tfs` „remote“ auch ein paar Referenzen hat: `default` und `featureA`, die TFVC-Branches darstellen. Git-tfs ordnet den Branch, von dem aus Sie geklont haben, `tfs/default` zu, während andere ihre eigenen Namen erhalten.

Eine weitere zu beachtende Funktion betrifft die `git-tfs-id:` Zeilen in den Commit-Beschreibungen. Anstelle von Tags verwendet git-tfs diese Marker, um TFVC-Change-Sets mit Git-Commits zu verknüpfen. Das hat den Effekt, dass Ihre Git-Commits einen unterschiedlichen SHA-1-Hash haben, vor und nach dem sie an TFVC übertragen wurden.

## Der Git-tf[s] Workflow

Unabhängig davon, welches Tool Sie verwenden, sollten Sie ein paar Git-Konfigurationswerte festlegen, um Probleme zu vermeiden.



```
$ git config set --local core.ignorecase=true
$ git config set --local core.autocrlf=false
```

Als nächstes wollen Sie natürlich an dem Projekt arbeiten. TFVC und TFS verfügen über mehrere Funktionen, die Ihren Workflow noch komplexer machen können:

1. Themen-Branches, die nicht in TFVC dargestellt werden, erhöhen die Komplexität. Das hängt mit den **sehr** unterschiedlichen Möglichkeiten zusammen, wie TFVC und Git Branches darstellen.
2. Beachten Sie, dass TFVC es Benutzern erlaubt, Dateien vom Server „auszuchecken“ und sie so zu sperren, dass niemand sonst sie bearbeiten kann. Das wird Sie natürlich nicht davon abhalten, sie in Ihrem lokalen Repository zu bearbeiten, aber es könnte Ihnen im Weg stehen, wenn es darum geht, Ihre Änderungen auf den TFVC-Server zu übertragen.
3. TFS hat mit dem Konzept der „eingezäunten“ (engl. gated) Checkins, bei denen ein TFS-Build-Testzyklus erfolgreich abgeschlossen werden muss, bevor das Einchecken erlaubt wird. Dazu

dient die Funktion „shelve“ in TFVC, auf die wir hier nicht näher eingehen werden. Sie können das mit git-tf manuell vortäuschen. Dabei stellt git-tfs den Befehl `checkintool` zur Verfügung, der eine Einzäunung (engl. gate) erkennt.

Kurz gesagt, was wir hier behandeln werden, ist der logische Weg, der die meisten dieser Probleme umgeht oder vermeidet.

### git-tf Workflow:

Nehmen wir an, Sie haben einige Arbeiten erledigt, ein paar Git-Commits auf `master` gemacht und Sie sind fertig, Ihre Ergebnisse auf den TFVC-Server zu übertragen. Das ist unser Git-Repository:

```
$ git log --oneline --graph --decorate --all
* 4178a82 (HEAD, master) update code
* 9df2ae3 update readme
* d44b17a (tag: TFS_C35190, origin_tfs/tfs) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Wir wollen einen Snapshot machen, der sich im Commit `4178a82` befindet und ihn auf den TFVC-Server pushen. Das Wichtigste zuerst: Mal sehen, ob einer unserer Teamkollegen seit unserer letzten Verbindung etwas getan hat:

```
$ git tf fetch
Username: domain\user
Password:
Connecting to TFS...
Fetching $/myproject at latest changeset: 100%, done.
Downloaded changeset 35320 as commit 8ef06a8. Updated FETCH_HEAD.
$ git log --oneline --graph --decorate --all
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
| * 4178a82 (HEAD, master) update code
| * 9df2ae3 update readme
|/
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
  Team Project Creation Wizard
```

Es sieht so aus, als würde auch noch jemand anderes daran arbeiten. Jetzt haben wir einen abweichenden Verlauf. Hier glänzt Git: wir haben zwei Möglichkeiten, wie es weitergehen kann:

1. Einen Merge-Commit zu machen fühlt sich als Git-Benutzer natürlich an (schließlich ist es das, was `git pull` macht) und git-tf kann das mit einem einfachen `git tf pull` für Sie tun. Seien Sie sich jedoch bewusst, dass TFVC nicht auf diese Weise denkt. Wenn Sie die Merge-Commits

pushen, wird Ihr Verlauf auf beiden Seiten unterschiedlich aussehen, was verwirrend sein kann. Aber wenn Sie alle Ihre Änderungen in einem Changeset übertragen, ist das wahrscheinlich der einfachste Weg.

2. Ein Rebase bewirkt, dass unsere Commit-Historie linear wird, was bedeutet, dass wir die Möglichkeit haben, jedes unserer Git-Commits in ein TFVC-Changeset zu konvertieren. Da dadurch die meisten Optionen offen bleiben, empfehlen wir Ihnen diese Vorgehensweise. Mit Hilfe von `git tf` wird es Ihnen mit dem Befehl `git tf pull - rebase` sogar leicht gemacht.

Sie haben die Wahl. Für dieses Beispiel werden wir ein Rebase durchführen:

```
$ git rebase FETCH_HEAD
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320, origin_tfs/tfs) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
    Team Project Creation Wizard
```

Nun können wir einen „checkin“ auf den TFVC-Server durchführen. Git-tf bietet Ihnen die Möglichkeit, ein einziges Changeset durchzuführen, das alle Veränderungen seit der letzten Änderung repräsentiert (`--shallow`, die Standardeinstellung) oder ein neues Changeset für jeden Git-Commit zu erstellen (`--deep`). Für dieses Beispiel werden wir nur ein einziges Changeset erzeugen:

```
$ git tf checkin -m 'Updating readme and code'
Username: domain\user
Password:
Connecting to TFS...
Checking in to $/myproject: 100%, done.
Checked commit 5a0e25e in as changeset 35348
$ git log --oneline --graph --decorate --all
* 5a0e25e (HEAD, tag: TFS_C35348, origin_tfs/tfs, master) update code
* 6eb3eb5 update readme
* 8ef06a8 (tag: TFS_C35320) just some text
* d44b17a (tag: TFS_C35190) Goodbye
* 126aa7b (tag: TFS_C35189)
* 8f77431 (tag: TFS_C35178) FIRST
* 0745a25 (tag: TFS_C35177) Created team project folder $/tfvctest via the \
    Team Project Creation Wizard
```

Es gibt ein neues `TFS_C35348` Tag, das anzeigt, dass TFVC genau den gleichen Snapshot speichert wie

der **5a0e25e** Commit. Wichtig zu beachten ist, dass nicht jeder Git-Commit ein genaues Gegenstück in TFVC haben muss. Der **6eb3eb5** Commit, zum Beispiel, existiert nirgendwo auf dem Server.

So sieht der wichtigste Workflow aus. Es gibt noch ein paar andere Aspekte, die Sie im Auge behalten sollten:

- Es gibt kein Branching. Git-tf kann Git-Repositorys jeweils nur aus einem einzigen TFVC-Branch erstellen.
- Benutzen Sie zur Zusammenarbeit TFVC *oder* Git, aber nicht beides. Unterschiedliche git-tf-Klone desselben TFVC-Repositorys können unterschiedliche Commit SHA-1-Hashes haben, was zu Kopfschmerzen ohne Ende führen wird.
- Wenn der Workflow Ihres Teams die Zusammenarbeit in Git und die regelmäßige Synchronisierung mit TFVC umfasst, verbinden Sie sich mit TFVC ausschließlich mit einem der Git-Repositories.

### git-tfs Workflow:

Lassen Sie uns das dasselbe Szenario mit git-tfs durchgehen. Hier sind die neuen Commits, die wir für den **master** Branch in unserem Git-Repository vorgenommen haben:

```
PS> git log --oneline --graph --all --decorate
* c3bd3ae (HEAD, master) update code
* d85e5a2 update readme
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 (tfv/default) Hello
* b75da1a New project
```

Mal sehen, ob noch jemand anderes gearbeitet hat, während wir abgeschnitten waren:

```
PS> git tfs fetch
C19 = aea74a0313de0a391940c999e51c5c15c381d91d
PS> git log --all --oneline --graph --decorate
* aea74a0 (tfv/default) update documentation
| * c3bd3ae (HEAD, master) update code
| * d85e5a2 update readme
|
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 Hello
* b75da1a New project
```

Ja, es stellt sich heraus, dass ein Co-Autor ein neues TFVC Change-Set hinzugefügt hat, das als neuer **aea74a0** Commit erscheint und der **tfv/default** Remote Branch verändert wurde.

Wie bei git-tf haben wir zwei fundamentale Optionen, um diesen unterschiedlichen Verlauf aufzulösen:

1. Ein Rebase, um den linearen Verlauf zu erhalten.
2. Ein Merge, um zu behalten, was wirklich passiert ist.

In diesem Fall werden wir einen „deep“ checkin durchführen. Dabei wird jeder Git Commit zu einem TFVC Changeset, also sollten wir einen Rebase durchführen.

```
PS> git rebase tfs/default
First, rewinding head to replay your work on top of it...
Applying: update readme
Applying: update code
PS> git log --all --oneline --graph --decorate
* 10a75ac (HEAD, master) update code
* 5cec4ab update readme
* aea74a0 (tfs/default) update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 Hello
* b75da1a New project
```

Nun sind wir soweit, unseren Beitrag zu abzuschließen, indem wir unseren Code in den TFVC-Server einchecken. Wir werden hier den Befehl **rcheckin** verwenden, um ein TFVC Change-Set für jeden Git-Commit im Pfad von HEAD zum ersten gefundenen **tfs** Remote-Branch zu erstellen (der **checkin** Befehl würde nur einen Changeset erzeugen, ähnlich wie beim Squashen von Git-Commits).

```
PS> git tfs rcheckin
Working with tfs remote: default
Fetching changes from TFS to minimize possibility of late conflict...
Starting checkin of 5cec4ab4 'update readme'
  add README.md
C20 = 71a5ddce274c19f8fdc322b4f165d93d89121017
Done with 5cec4ab4b213c354341f66c80cd650ab98dcf1ed, rebasing tail onto new TFS-
commit...
Rebase done successfully.
Starting checkin of b1bf0f99 'update code'
  edit .git\dfs\default\workspace\ConsoleApplication1\ConsoleApplication1\Program.cs
C21 = ff04e7c35dfbe6a8f94e782bf5e0031cee8d103b
Done with b1bf0f9977b2d48bad611ed4a03d3738df05ea5d, rebasing tail onto new TFS-
commit...
Rebase done successfully.
No more to rcheckin.
PS> git log --all --oneline --graph --decorate
* ff04e7c (HEAD, tfs/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfs/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|/
* c403405 Hello
* b75da1a New project
```

Bitte beachten Sie, dass git-tfs nach jedem erfolgreichen checkin auf dem TFVC-Server die verbleibende Arbeit auf das zurücksetzt, was gerade getan wurde. Das liegt daran, dass es das Feld **git-tfs-id** am Ende der Commit-Beschreibungen hinzufügt, was die SHA-1-Hashes ändert. Das ist genau so, wie es entworfen wurde. Es gibt keinen Grund sich Sorgen zu machen, aber Sie sollten sich bewusst sein, dass es geschieht, vor allem wenn Sie Git-Commits mit anderen teilen.

TFS verfügt über viele Funktionen, die sich in sein Versionskontrollsystem integrieren lassen, wie z.B. Workitems, benannte Prüfer, Gated Checkins usw. Es kann umständlich sein, diese Funktionen nur mit einem Kommandozeilen-Tool zu verwenden, aber glücklicherweise gibt es mit git-tfs ein grafisches Checkin-Tool, das Sie sehr einfach starten können:

```
PS> git tfs checkin
PS> git tfs ct
```

Es sieht ungefähr so aus:

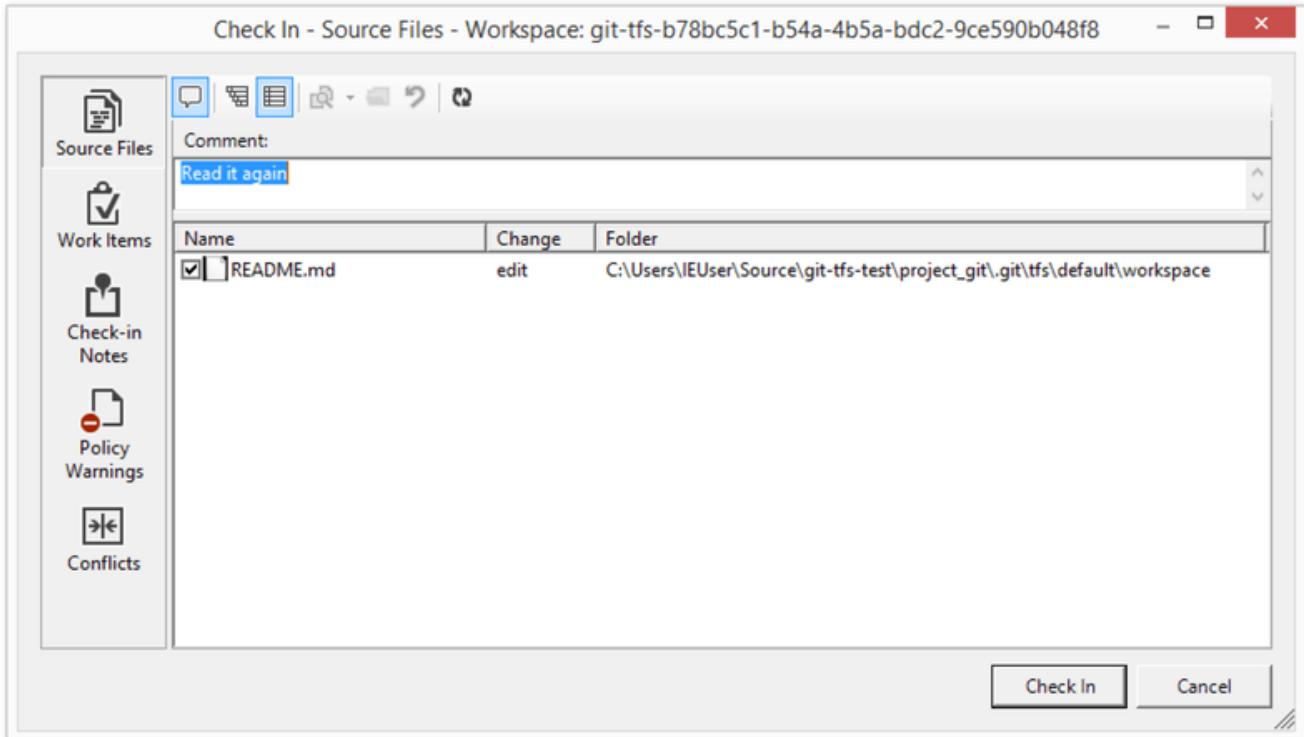


Figure 147. Das git-tfs Checkin-Tool

Das kommt den TFS-Benutzern vertraut vor, da es sich um den gleichen Dialog handelt, der aus Visual Studio heraus gestartet wird.

Mit Git-tfs können Sie auch TFVC-Banches aus Ihrem Git-Repository steuern. Lassen Sie uns zum Beispiel einen Branch erstellen:

```
PS> git tfs branch $/tfvc-test/featureBee
The name of the local branch will be : featureBee
C26 = 1d54865c397608c004a2cadce7296f5edc22a7e5
PS> git log --oneline --graph --decorate --all
* 1d54865 (tfv/featureBee) Creation branch $/myproject/featureBee
* ff04e7c (HEAD, tfv/default, master) update code
* 71a5ddc update readme
* aea74a0 update documentation
| * 44cd729 (tfv/featureA, featureA) Goodbye
| * d202b53 Branched from $/tfvc-test/Trunk
|
* c403405 Hello
* b75da1a New project
```

Das Erstellen eines Branch in TFVC bedeutet, ein Change-Set hinzuzufügen, in dem dieser Branch bereits existiert. Dieser Branch wird als Git-Commit projiziert. Beachten Sie auch, dass git-tfs den **tfv/featureBee** Remote-Branch **erstellt hat**, aber dass **HEAD** immer noch auf **master** zeigt. Wenn Sie an dem neu erstellten Zweig arbeiten möchten, sollten Sie Ihre neuen Commits auf dem **1d54865** Commit basieren, möglicherweise indem Sie einen Topic Branch aus diesem Commit erstellen.

## Git und TFS, Zusammenfassung

Git-tf und Git-tfs sind beides großartige Werkzeuge für die Verbindung zu einem TFVC-Server. Sie ermöglichen es Ihnen, die Leistungsfähigkeit von Git lokal zu nutzen, zu vermeiden, dass Sie ständig auf den zentralen TFVC-Server zurückkehren müssen und Ihren Alltag als Entwickler viel einfacher zu gestalten, ohne Ihr gesamtes Team zur Migration nach Git zu zwingen. Wenn Sie unter Windows arbeiten (was wahrscheinlich ist, wenn Ihr Team TFS verwendet), werden Sie vermutlich git-tfs verwenden wollen, da der Funktionsumfang vollständiger ist, aber wenn Sie auf einer anderen Plattform arbeiten, werden Sie git-tf verwenden, das etwas eingeschränkter ist. Wie bei den meisten Tools in diesem Kapitel sollten Sie eines dieser Versionskontrollsysteme wählen, um eindeutig zu sein, und das andere in einer untergeordneten Form verwenden – entweder Git oder TFVC sollte das Zentrum der Zusammenarbeit sein, aber nicht beide.

## Migration zu Git

Wenn Sie eine bestehende Quelltext-Basis in einem anderen VCS haben, aber sich für die Verwendung von Git entschieden haben, müssen Sie Ihr Projekt auf die eine oder andere Weise migrieren. Dieser Abschnitt geht auf einige Importfunktionen für gängige Systeme ein und zeigt dann, wie Sie Ihren eigenen benutzerdefinierten Importeur entwickeln können. Sie lernen, 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 Sie den vorherigen Abschnitt über die Verwendung von `git svn` gelesen haben, können Sie die Anweisungen zu `git svn clone` leicht dazu benutzen, um ein Repository zu klonen. Beenden Sie dann die Verwendung des Subversion-Servers, pushen Sie zu einem neuen Git-Server und starten Sie dessen Nutzung. Der Verlauf kann in diesem Fall aus dem Subversion-Server gezogen werden (was einige Zeit in Anspruch nehmen kann – abhängig von der Geschwindigkeit, mit der Ihr SVN-Server die Historie ausliefern kann).

Allerdings ist der Import nicht perfekt. Da er aber so lange dauert, können Sie ihn genauso gut auch 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 Sie diese auf bessere Git-Autorendaten abbilden möchten, benötigen Sie eine Zuordnung der Subversion-Benutzer zu den Git-Autoren. Erstellen Sie eine Datei mit Namen `users.txt`, die diese Zuordnung in einem solchen Format vornimmt:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Um eine Liste der Autorennamen zu erhalten, die SVN verwendet, können Sie 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. Leiten Sie diese Ausgabe dann in Ihre `users.txt` Datei um, damit Sie die entsprechenden Git-Benutzerdaten neben jedem Eintrag hinzufügen können.



Wenn Sie dies auf einem Windows-Computer versuchen, 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.

Sie können diese Datei an `git svn` übergeben, um die Autorendaten genauer abzubilden. Außerdem können Sie `git svn` anweisen, die Metadaten, die Subversion normalerweise importiert, nicht zu berücksichtigen. Dazu übergeben Sie `--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 Ihr Git-Log aufblättern und es möglicherweise etwas unübersichtlich machen.



Sie müssen die Metadaten beibehalten, wenn Sie im Git-Repository vorgenommene Commits wieder in das ursprüngliche SVN-Repository spiegeln möchten.

Wenn Sie die Synchronisierung nicht in Ihrem Commit-Protokoll möchten, können Sie den Parameter `--no-metadata` weglassen.

Dadurch sieht Ihr `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 sollten Sie einen passenderen Subversion-Import in Ihrem `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.

Sie sollten auch eine gewisse Bereinigung nach dem Import durchführen. Zum einen sollten Sie die seltsamen Referenzen bereinigen, die `git svn` eingerichtet hat. Verschieben Sie zuerst die Tags so, dass sie echte Tags und nicht merkwürdige Remote-Banches darstellen. Dann verschieben Sie den Rest der Branches auf lokale Tags.

Damit die Tags zu richtigen Git-Tags werden, starten Sie:

```
$ 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 verschieben Sie 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 Sie einige zusätzliche Branches sehen, die durch `@xxx` ergänzt sind (wobei xxx eine Zahl ist), während Sie in Subversion nur einen Branch sehen. Es handelt sich hierbei um eine Subversion-Funktion mit der Bezeichnung „peg-revisions“, für die Git einfach kein syntaktisches Gegenstück hat. Daher fügt `git svn` einfach die SVN-Versionsnummer zum Branch-Namen hinzu, genau so, wie Sie es in SVN geschrieben hätten, um die peg-Revision dieses Branchs anzusprechen. Wenn Sie sich nicht mehr um die peg-Revisions sorgen wollen, entfernen Sie 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-Branches 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 Sie tun müssen, ist, Ihren neuen Git-Server als Remote hinzuzufügen und ihn zu aktivieren. Hier ist ein Beispiel für das Hinzufügen Ihres Servers als Remote:

```
$ git remote add origin git@my-git-server:myrepository.git
```

Um alle Ihre Branches und Tags zu aktualisieren, können Sie jetzt diese Anweisungen ausführen:

```
$ git push origin --all  
$ git push origin --tags
```

Alle Ihre Branches und Tags sollten sich nun auf Ihrem neuen Git-Server in einem schönen, sauberen Import befinden.

## Mercurial

Da Merkural und Git haben ziemlich ähnliche Modelle für die Darstellung von Versionen. Außerdem ist Git etwas flexibler, so dass die Konvertierung eines Repositorys von Merkural nach Git ziemlich einfach ist. Dazu wird ein Tool mit der Bezeichnung „hg-fast-export“ verwendet, von dem Sie eine Kopie benötigen:

```
$ 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 generieren Sie mit einem einzeiligen Befehl in einer **bash** Shell:

```
$ 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 Ihres 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>
Joe Smith <joe@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 können Sie das Repository in das neue serverseitige System pushen:

```
$ git remote add origin git@my-git-server:myrepository.git  
$ git push origin --all
```

## Bazaar

Bazaar ist ein DVCS-Tool ähnlich wie Git. Deshalb ist es relativ unkompliziert, ein Bazaar-Repository in ein Git-Repository zu konvertieren. Um dieses Ziel zu erreichen, müssen Sie das **bzr-fastimport** Plugin einlesen.

### Das bzr-fastimport Plugin herunterladen

The procedure for installing the fastimport plugin is different on UNIX-like operating systems and on Windows. In the first case, the simplest is to install the **bzr-fastimport** package that will install all the required dependencies.

Zum Beispiel, mit Debian (und seinen Derivaten), würden Sie folgendes tun:

```
$ sudo apt-get install bzr-fastimport
```

Mit RHEL würden Sie folgendes tun:

```
$ sudo yum install bzr-fastimport
```

Bei Fedora, seit Release 22, heißt der neue Paketmanager dnf:

```
$ sudo dnf install bzr-fastimport
```

Falls kein Packet verfügbar ist, können Sie es als Plugin installieren:

```
$ mkdir --parents ~/.bazaar/plugins      # creates the necessary folders for the  
plugins  
$ cd ~/.bazaar/plugins  
$ bzr branch lp:bzr-fastimport fastimport  # imports the fastimport plugin  
$ cd fastimport  
$ sudo python setup.py install --record=files.txt  # installs the plugin
```

Damit dieses Plugin funktioniert, benötigen Sie auch das **fastimport** Python-Modul. Sie können überprüfen, ob es vorhanden ist oder nicht und es dann mit den folgenden Befehlen installieren:

```
$ python -c "import fastimport"
Traceback (most recent call last):
  File "<string>", line 1, in <module>
ImportError: No module named fastimport
$ pip install fastimport
```

Wenn es nicht vorhanden ist, können Sie es unter der Adresse <https://pypi.python.org/pypi/fastimport> herunterladen.

Im zweiten Fall (unter Windows) wird **bzr-fastimport** automatisch mit der Standalone-Version und der Default-Installation auf Ihrem Computer mit installiert (alle Kontrollkästchen aktiviert lassen). Deshalb haben Sie in diesem Fall nichts weiter zu tun.

An dieser Stelle unterscheidet sich die Vorgehensweise beim Import eines Bazaar-Repositorys dahingehend, ob Sie einen einzelnen Branch haben oder mit einem Repository arbeiten, das mehrere Branches hat.

### Projekt mit einer einzigen Branch

Wechseln Sie jetzt mit **cd** in das Verzeichnis, das Ihr Bazaar-Repository enthält, und initialisieren Sie das Git-Repository:

```
$ cd /path/to/the/bzr/repository
$ git init
```

Nun können Sie Ihr Bazaar-Repository einfach exportieren und mit dem folgenden Befehl in ein Git-Repository konvertieren:

```
$ bzr fast-export --plain . | git fast-import
```

Abhängig von der Größe des Projekts wird Ihr Git-Repository in einer Zeitspanne von wenigen Sekunden bis einigen Minuten erstellt.

### Projekt mit einem Hauptbranch und einem Arbeitsbranch

Sie können auch ein Bazaar-Repository importieren, das Branches enthält. Angenommen, Sie haben zwei Branches: Einer repräsentiert den Hauptzweig (`myProject/trunk`), der andere ist der Arbeitszweig (`myProject/work`).

```
$ ls
myProject/trunk myProject/work
```

Erstellen Sie das Git-Repository und wechseln Sie jetzt mit **cd** in dieses:

```
$ git init git-repo  
$ cd git-repo
```

Den Master-Branch zu Git pullen:

```
$ bzr fast-export --export-marks=../marks.bzr ../myProject/trunk | \  
git fast-import --export-marks=../marks.git
```

Den Arbeits-Branch zu Git pullen:

```
$ bzr fast-export --marks=../marks.bzr --git-branch=work ../myProject.work | \  
git fast-import --import-marks=../marks.git --export-marks=../marks.git
```

Jetzt zeigt Ihnen **git branch** sowohl den **master** Branch als auch den **work** Branch. Überprüfen Sie die Protokolle, um sicherzustellen, dass sie vollständig sind, und entfernen Sie die Dateien **marks.bzr** und **marks.git**.

## Die Staging Area synchronisieren

Unabhängig von der Anzahl der Branches und der verwendeten Importmethode ist Ihre Staging Area nicht mit **HEAD** synchronisiert, und beim Import mehrerer Branches ist auch Ihr Arbeitsverzeichnis nicht synchronisiert. Diese Situation lässt sich mit dem folgenden Befehl leicht lösen:

```
$ git reset --hard HEAD
```

## Mit **.bzrignore** ignorierte Dateien auslassen

Werfen wir nun einen Blick auf die zu ignorierenden Dateien. Zuerst müssen Sie **.bzrignore** in **.gitignore** umbenennen. Wenn die Datei **.bzrignore** eine oder mehrere Zeilen enthält, die mit „!!“ oder „RE:“ beginnen, müssen Sie sie ändern und vielleicht mehrere **.gitignore** Dateien anlegen, um genau die gleichen Dateien zu ignorieren, die Bazaar ignoriert hat.

Schließlich ist ein Commit zu erstellen, der diese Änderung für die Migration enthält:

```
$ git mv .bzrignore .gitignore  
$ # modify .gitignore if needed  
$ git commit -am 'Migration from Bazaar to Git'
```

## Ihr Repository an den Server übertragen

Hier wären wir! Jetzt können Sie das Repository auf seinen neuen Zielserver pushen:

```
$ git remote add origin git@my-git-server:mygitrepository.git  
$ git push origin --all  
$ git push origin --tags
```

Ihr Git-Repository ist einsatzbereit.

## Perforce

Bei dem nächsten System, aus dem Sie importieren können, 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. Konfigurieren Sie einfach Ihre Projekteinstellungen, Benutzerzuordnungen und Branches mit Hilfe einer Konfigurationsdatei (wie in [Git Fusion](#) beschrieben) und klonen Sie das Repository. Git Fusion bietet Ihnen ein natives Git-Repository, mit dem Sie nach Belieben auf einen nativen Git-Host wechseln können. Sie können Perforce sogar als Ihren Git-Host verwenden, wenn Sie möchten.

### Git-p4

Git-p4 kann auch als Import-Tool fungieren. Als Beispiel werden wir das Jam-Projekt aus dem Perforce Public Depot importieren. Um Ihren Client einzurichten, müssen Sie die Umgebungsvariable P4PORT exportieren und auf das Perforce-Depot verweisen:

```
$ export P4PORT=public.perforce.com:1666
```



Zur weiteren Bearbeitung benötigen Sie ein Perforce-Depot, mit dem Sie sich verbinden können. Wir werden das öffentliche Depot unter public.perforce.com für unsere Beispiele verwenden. Sie können aber jedes Depot nutzen, zu dem Sie Zugang haben.

Führen Sie den Befehl `git p4 clone` aus, um das Jam-Projekt vom Perforce-Server zu importieren, wobei Sie das Depot, den Projektpfad und den Pfad angeben, in den Sie das Projekt importieren möchten:

```
$ 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 Sie Branches haben, die mit Branch Views (oder nur einer Gruppe von Verzeichnissen) eingerichtet sind, können Sie 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.

In diesem Moment sind Sie fast fertig. Wenn Sie in das Verzeichnis `p4import` wechseln und `git log` ausführen, können Sie Ihr importiertes Projekt sehen:

```
$ git log -2
commit e5da1c909e5db3036475419f6379f2c73710c4e6
Author: giles <giles@perforce.com>
Date:   Wed Feb 8 03:13:27 2012 -0800

    Correction to line 355; change </UL> to </OL>.

[git-p4: depot-paths = "//public/jam/src/": change = 8068]

commit aa21359a0a135dda85c50a7f7cf249e4f7b8fd98
Author: kwirth <kwirth@perforce.com>
Date:   Tue Jul 7 01:35:51 2009 -0800

    Fix spelling error on Jam doc page (cummulative -> cumulative).

[git-p4: depot-paths = "//public/jam/src/": change = 7304]
```

Sie können sehen, dass `git-p4` in jeder Commit-Nachricht eine Kennung hinterlassen hat. Es ist gut, diese Kennung dort zu behalten, falls Sie später auf die Perforce-Änderungsnummer verweisen müssen. Wenn Sie den Identifier jedoch entfernen möchten, ist es jetzt der richtige Zeitpunkt – bevor Sie mit der Arbeit am neuen Repository beginnen. Sie können `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 Sie `git log` ausführen, können Sie 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@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).

Ihr Import ist bereit, um ihn auf Ihren neuen Git-Server zu pushen.

## TFS

Wenn Ihr Team seine Versionskontrolle von TFVC nach Git umwandelt, werden Sie die bestmögliche Konvertierung benötigen, die Sie erhalten können. Das bedeutet, dass wir, während wir sowohl git-tfs als auch git-tf für den Interop-Bereich abgedeckt haben, nur git-tfs für diesen Teil behandeln werden, da git-tfs Branches unterstützt, was bei der Verwendung von git-tf äußerst schwierig ist.



Das ist eine Ein-Weg-Konvertierung. Das so entstandene Git-Repository kann sich nicht mit dem ursprünglichen TFVC-Projekt verbinden.

Als erstes müssen Sie die Benutzernamen zuordnen. TFVC ist ziemlich großzügig mit dem, was in das Autorenfeld für Change-Sets aufgenommen wird. Aber Git will einen benutzerfreundlichen Namen und eine E-Mail-Adresse. Sie können diese Informationen vom **tf** Befehlszeilen-Client erhalten, so wie hier:

```
PS> tf history $/myproject -recursive > AUTHORS_TMP
```

Dadurch werden alle Change-Sets im Verlauf des Projekts erfasst und in die Datei **AUTHORS\_TMP** geschrieben. Wir werden diese Datei verarbeiten, um die Daten der Spalte 'User' (die zweite) zu extrahieren. Öffnen Sie die Datei und finden Sie heraus, an welchen Zeichen die Spalte beginnt und endet und ersetzen Sie in der folgenden Befehlszeile die Werte **11-20** des **cut** Befehls durch die gefundenen:

```
PS> cat AUTHORS_TMP | cut -b 11-20 | tail -n+3 | sort | uniq > AUTHORS
```

Der Befehl **cut** berücksichtigt nur die Zeichen zwischen 11 und 20 aus jeder Zeile. Der Befehl **tail** überspringt die ersten beiden Zeilen, die Feld-Header und ASCII-artige Unterstriche sind. Das Resultat davon wird an **sort** und **uniq** weitergeleitet, um Duplikate zu eliminieren und in der Datei **AUTHORS** gespeichert. Der nächste Schritt erfolgt manuell. Damit git-tfs diese Datei effektiv nutzen kann, muss jede Zeile in diesem Format vorliegen:

```
DOMAIN\username = User Name <email@address.com>
```

Der linke Teil ist das Feld „User“ von TFVC und auf der rechten Seite des Gleichheitszeichens ist der Benutzername, der für Git Commits verwendet wird.

Sobald Sie diese Datei haben, ist der nächste Schritt, einen vollständigen Klon des TFVC-Projekts zu erstellen, an dem Sie interessiert sind:

```
PS> git tfs clone --with-branches --authors=AUTHORS  
https://username.visualstudio.com/DefaultCollection $/project/Trunk project_git
```

Als nächstes sollten Sie die `git-tfs-id` Abschnitte unten in den Commit-Nachrichten bereinigen. Die folgende Anweisung erledigt das:

```
PS> git filter-branch -f --msg-filter 'sed "s/^git-tfs-id:.*$/g"' '--' --all
```

Der verwendete `sed` Befehl aus der Git-bash-Umgebung ersetzt jede Zeile, die mit „git-tfs-id:“ beginnt, durch Leerzeichen, die Git dann ignoriert.

Sobald das alles erledigt ist, können Sie einen neuen Remote hinzufügen, alle Ihre Branches nach oben pushen und Ihr Team aus Git heraus arbeiten lassen.

## Benutzerdefinierter Import

Wenn Ihr System nicht zu den vorgenannten gehört, sollten Sie 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 Sie geeignet ist, Sie ein obskures Tool haben oder anderweitig einen benutzerdefinierten Importprozess benötigen, sollten Sie `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 können Sie ein Import-Skript schreiben, das die notwendigen Informationen aus dem System liest, aus dem Sie importieren, und Anweisungen direkt auf „stdout“ ausgibt. Sie können dann dieses Programm ausführen und seine Ausgaben über `git fast-import` pipen.

Um das kurz zu demonstrieren, schreiben Sie eine einfache Import-Anweisung. Angenommen, Sie arbeiten im `current` Branch, Sie sichern Ihr Projekt, indem Sie das Verzeichnis gelegentlich in ein mit Zeitstempel versehenes `back_YYYY_MM_DD` Backup-Verzeichnis kopieren und dieses in Git importieren möchten. Ihre 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 Sie ein Git-Verzeichnis importieren können, müssen Sie sich ansehen, wie Git seine Daten speichert. Wie Sie sich vielleicht erinnern, ist Git im Grunde genommen eine verknüpfte Liste von Commit-Objekten, die auf einen Schnapschuss des Inhalts verweisen. Alles, was Sie tun müssen, 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. Ihre 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. Sie können dieses Beispiel sehr leicht in jedem Editor schreiben, den Sie kennen – er muss nur die entsprechenden Informationen nach **stdout** ausgeben können. Unter Windows müssen Sie besonders darauf achten, dass Sie am Ende Ihrer Zeilen keine Zeilenumbrüche einfügen – **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 wechseln Sie in das Zielverzeichnis und identifizieren jene Unterverzeichnisse, von denen jedes ein Snapshot ist, den Sie als Commit importieren möchten. Sie wechseln in jedes Unterverzeichnis und geben die für den Export notwendigen Befehle aus. Ihre 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
```

Führen Sie **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; auf diese Weise können Sie sie richtig verlinken. „Mark“ ist der **fast-import** Begriff für eine Kennung, die Sie einem Commit mitgeben. Wenn Sie Commits erstellen, geben Sie jedem eine Markierung, mit dem Sie ihn von anderen Commits aus verlinken können. Daher ist das Wichtigste

in Ihrer `print_export` Methode, eine Markierung aus dem Verzeichnisnamen zu erzeugen:

```
mark = convert_dir_to_mark(dir)
```

Sie werden dazu ein Array von Verzeichnissen erstellen und den Indexwert als Markierung verwenden, eine Markierung muss nämlich eine Ganzzahl (Integer) sein. Ihre 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 Sie nun eine ganzzahlige Darstellung Ihres Commits haben, benötigen Sie ein Datum für die Commit-Metadaten. Das Datum wird im Namen des Verzeichnisses ausgewiesen, daher werden Sie es auswerten. Die nächste Zeile in Ihrer `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 Sie für jeden Commit benötigen, sind die Committer-Daten, die Sie in einer globalen Variable hartkodieren:

```
$author = 'John Doe <john@example.com>'
```

Damit sind Sie startklar für die Ausgabe der Commit-Daten für Ihren Importer. Die ersten Informationen beschreiben, dass Sie ein Commit-Objekt definieren und in welchem Branch es sich befindet, gefolgt von der Markierung, die Sie generiert haben, 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
```

Sie können die Zeitzone (-0700) hartkodieren, da das einfach ist. Wenn Sie sie aus einem anderen System importieren, müssen Sie 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 Sie später das gleiche Format verwenden müssen, um den Datei-Inhalt festzulegen, erstellen Sie mit `export_data` eine Hilfs-Methode:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Das ist einfach, denn Sie haben jeden in einem eigenen Verzeichnis. Sie können 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. Sie könnten die Unterschiede zwischen den Snapshots berechnen und nur diese Daten bereitstellen, aber das ist komplizierter – in diesem Fall sollten Sie Git alle Daten übergeben und sie auswerten lassen. Sollte diese Option für Ihre Daten besser geeignet sein, informieren Sie sich in der `fast-import` Man-Page, wie Sie Ihre Daten auf diese Weise bereitstellen können.

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 Sie ausführbare Dateien haben, müssen Sie stattdessen 755 ermitteln und festlegen), 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 Sie zuvor definiert hatten, handelt es sich um das gleiche Verfahren wie bei der Angabe Ihrer Commit-Message-Daten.

Als Letztes müssen Sie die aktuelle Markierung an das System zurückgeben, damit sie an die nächste Iteration weitergegeben werden kann:

```
return mark
```



Wenn Sie unter Windows arbeiten, müssen Sie 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` zufrieden zu stellen, müssen Sie 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
end

```

Wenn Sie dieses Skript ausführen, werden Inhalte mit ähnlichem Aussehen angezeigt:

```
$ 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, übergeben Sie diese Output-Pipe an `git fast-import`, während Sie sich in dem Git-Verzeichnis befinden, in das Sie importieren wollen. Sie können ein neues Verzeichnis erstellen und dort `git init` für einen Anfangspunkt ausführen und danach Ihr 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 Sie sehen können, gibt es nach erfolgreichem Abschluss eine Reihe von Statistiken über den erreichten Status. In diesem Fall haben Sie 13 Objekte mit insgesamt 4 Commits in einen Branch importiert. Jetzt können Sie [git log](#) ausführen, um Ihre 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 – Sie haben zunächst keine Dateien in Ihrem Arbeitsverzeichnis. Um sie zu erhalten, müssen Sie Ihren 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 können Sie viel mehr anfangen – bearbeiten von unterschiedlichen Modi, binären Daten, multiplen Branches und Merges, Tags, Verlaufsindikatoren und mehr. Eine Reihe von Beispielen für komplexere Szenarien finden Sie im `contrib/fast-import` Verzeichnis des Git-Quellcodes.

## Zusammenfassung

Sie sollten sich jetzt wohl dabei fühlen, Git als Client für andere Versionskontrollsysteme zu verwenden oder fast jedes vorhandene Repository in Git zu importieren, ohne Daten zu verlieren. Im nächsten Kapitel werden wir die internen Abläufe in Git beschreiben, so dass Sie jedes einzelne Byte nach Bedarf selbst erzeugen können.

# 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 Dateisystemsenschicht 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 (excludes 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 << ch08-customizing-git#\_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 (Zweige, Tags, Remotes, usw.) und die `HEAD`-Datei zeigt auf den Zweig, 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 **objects**-Verzeichnis 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. Der aktuellste Baum in einem Projekt sieht beispielsweise folgendermaßen aus:

```
$ 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`-Zweig 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:

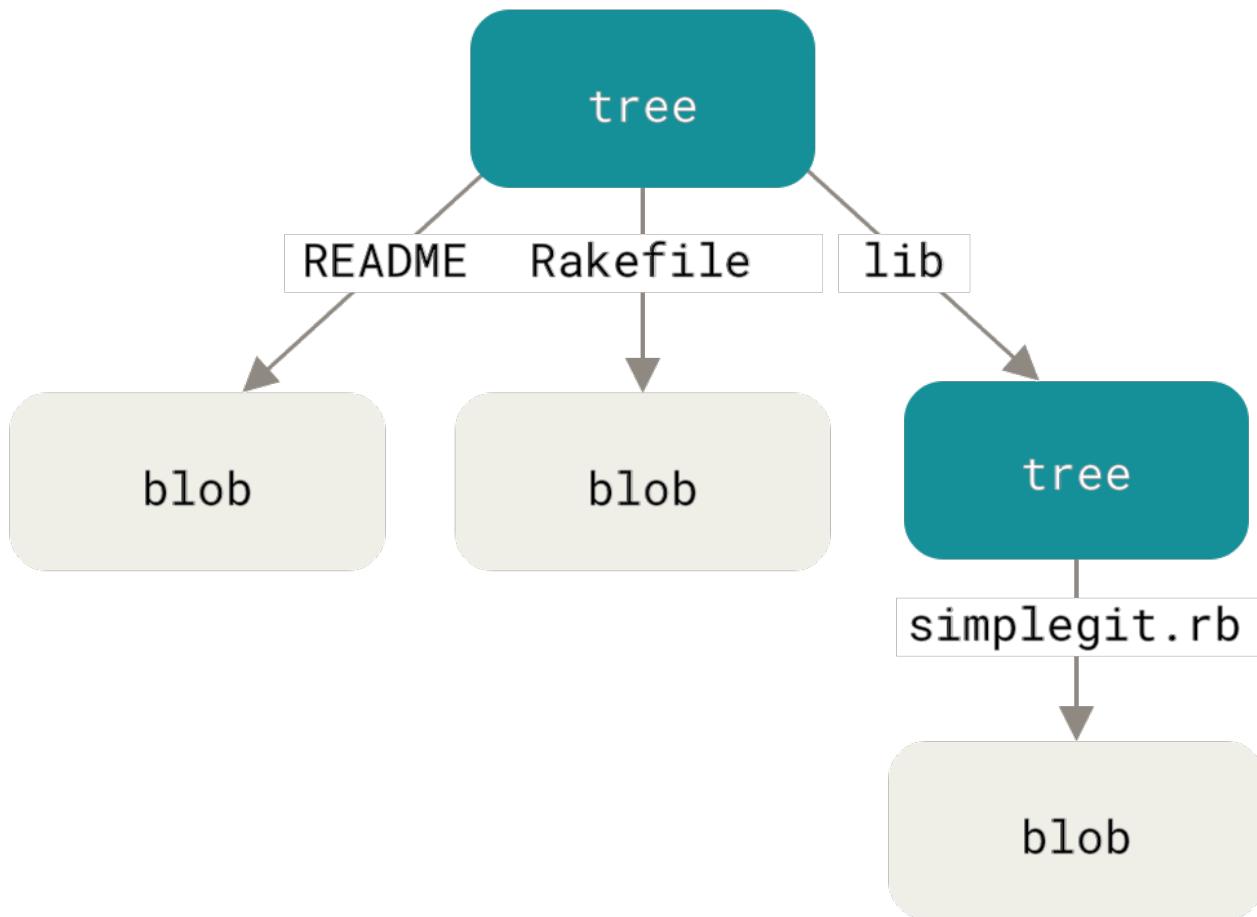


Figure 148. Einfache Version des Git-Datenmodells.

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 --add --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:

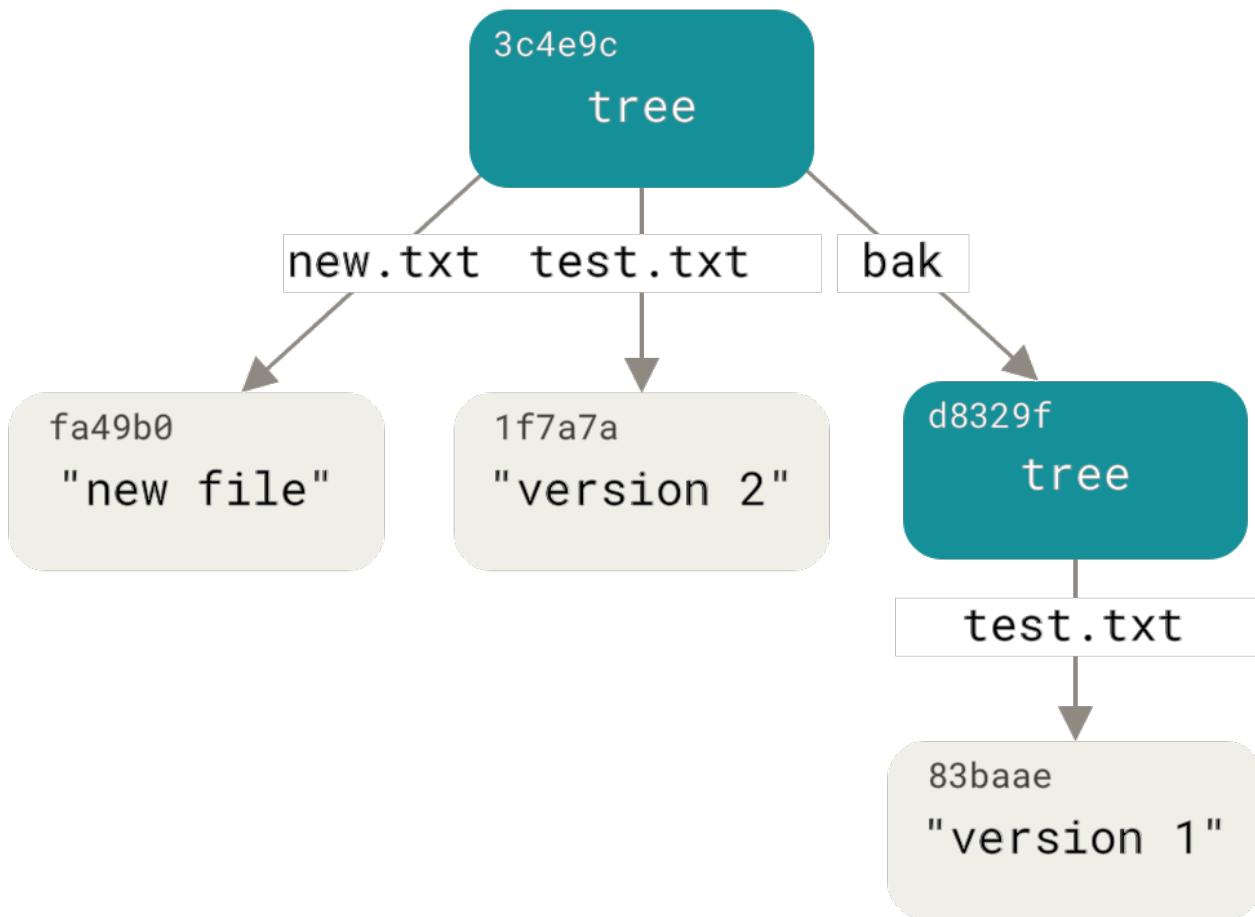


Figure 149. Die Inhaltsstruktur Ihrer aktuellen Git-Daten.

## 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 erhalten einen anderen Hash-Wert, da die Erstellungszeit und die Autorendaten unterschiedlich sind. Ersetzen Sie Commit- und Tag-Hashes durch Ihre eigenen Prüfsummen dieses Kapitels. 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

bак/test.txt | 1 +
1 file changed, 1 insertion(+)

commit cac0cab538b970a37ea1e769cbbde608743bc96d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:14:29 2009 -0700

    second commit

new.txt  | 1 +
test.txt | 2 ++
2 files changed, 2 insertions(+), 1 deletion(-)

commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700

    first commit

test.txt | 1 +
1 file changed, 1 insertion(+)
```

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/410efbd13591db07496601ebc7a059dd55cf9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Wenn Sie allen internen Zeigern folgen, erhalten Sie eine Objektgrafik in etwa wie folgt:

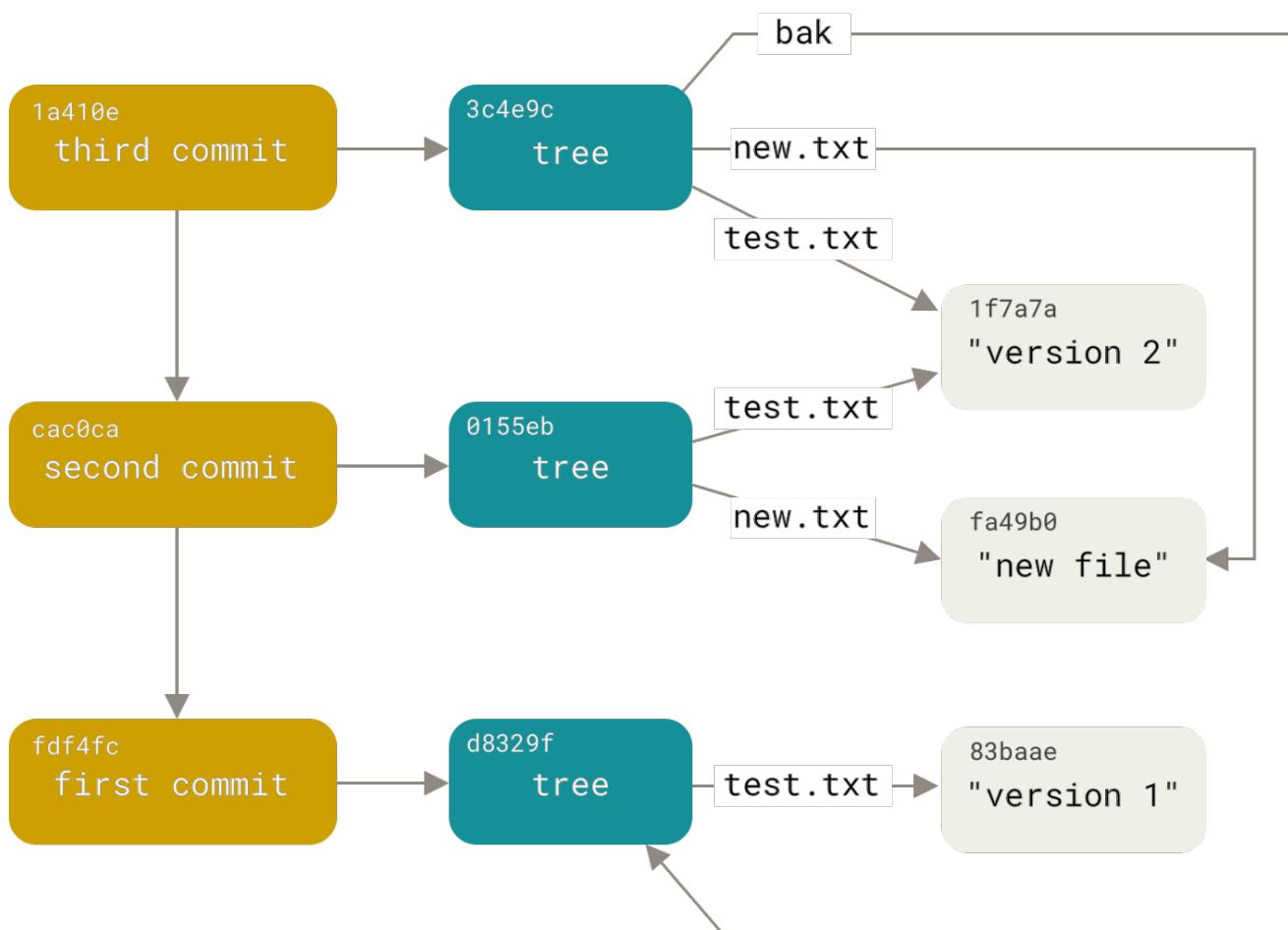


Figure 150. Alle erreichbaren Objekte in Ihrem Git-Verzeichnis.

## 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.length}\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\0000what 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(\xFH,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:

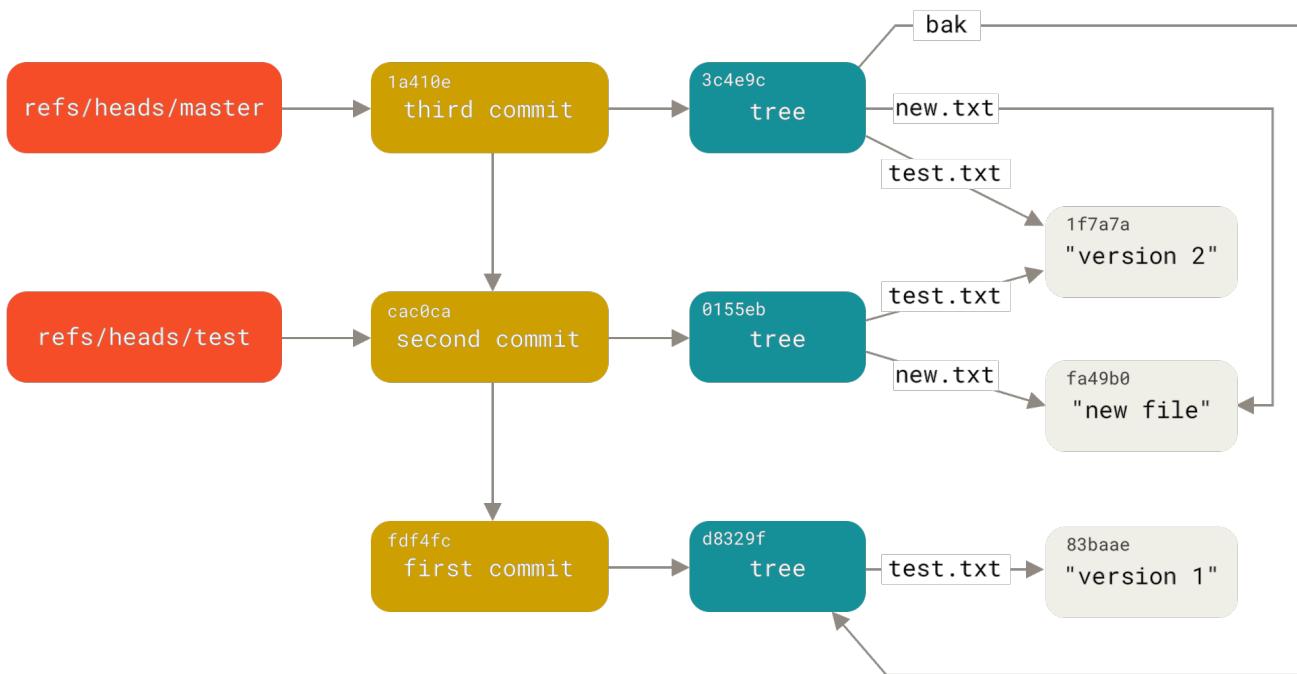


Figure 151. Git-Verzeichnisobjekte mit Branch Head Referenzen.

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-Branches, 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 cac0cab538b970a37ea1e769ccbde608743bc96d
```

Das ist alles, was ein leichtgewichtiger Tag ist — eine Referenz, die sich nie bewegt. Ein annotierter Tag ist jedoch komplexer. Wenn Sie einen annotierten 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 annotierten Tag erstellen (mit der Option `-a`):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cf9 -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 Zweig im Verzeichnis `refs/remotes`. Zum Beispiel können Sie eine Remote mit dem Namen `origin` hinzufügen und Ihren `master`-Zweig 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 darauf zeigen, 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 'added repo.rb'  
[master 484a592] added repo.rb  
 3 files changed, 709 insertions(+), 2 deletions(-)  
 delete mode 100644 bak/test.txt  
 create mode 100644 repo.rb  
 rewrite test.txt (100%)
```

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 'modified repo.rb a bit'
[master 2431da6] modified 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
22054
```

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, dieOffsets 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 Festplattenutzung 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
d8329fc1cc938780ffd9f94e0d364e0ea74f579 tree 36 46 1132
deef2e1b793907545e50a2ea2ddb5ba6c58c4506 tree 136 136 1178
d982c7cb2c2a972ee391a85da481fc1f9127a01d tree 6 17 1314 1 \
    deef2e1b793907545e50a2ea2ddb5ba6c58c4506
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree 8 19 1331 1 \
    deef2e1b793907545e50a2ea2ddb5ba6c58c4506
0155eb4229851634a0f03eb265b69f5a2d56f341 tree 71 76 1350
83baae61804e65cc73a7201a7252750c76066a30 blob 10 19 1426
fa49b077972391ad58037050f2a75f74e3671e92 blob 9 18 1445
b042a60ef7dff760008df33cee372b945b6e884e blob 22054 5799 1463
033b4468fa6b2a9547a70d88d1bbe8bf3f9ed0d5 blob 9 20 7262 1 \
    b042a60ef7dff760008df33cee372b945b6e884e
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob 10 19 7282
non delta: 15 objects
chain length = 1: 3 objects
.git/objects/pack/978e03944f5c581011e6998cd0e9e30000905586.pack: ok
```

Hier verweist der **033b4**-Blob, der, wenn Sie sich erinnern, die erste Version Ihrer `repo.rb`-Datei war, auf den **b042a**-Blob, 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 `git remote add origin`-Befehl 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
```

Sie können keine partiellen Globs im Pattern verwenden, folgendes wäre also ungültig:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Sie können jedoch Namespaces (oder Verzeichnisse) verwenden, um dies 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 `qa`-Namespace? 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 [Ihr öffentliches GitHub-Repository aktuell halten](#).

## Löschende 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  
  
changed the 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).

Jede Zeile beginnt mit einem 4-stelligen Hex-Wert, der angibt, wie lang der Rest der Zeile ist. Ihre erste Zeile beginnt mit 00a5, was hexadezimal für 165 ist. Dies bedeutet, dass 165 Bytes in dieser Zeile verbleiben. Die nächste Zeile ist 0000, was bedeutet, dass der Server mit seiner Referenzliste fertig ist.

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:

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~vgmgb-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/qit-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.

#### 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 0a53e9ddeaddad63ad106860237bbf53411d11a7
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.

# Maintenance and Data Recovery

Occasionally, you may have to do some cleanup – make a repository more compact, clean up an imported repository, or recover lost work. This section will cover some of these scenarios.

## Maintenance

Occasionally, Git automatically runs a command called “auto gc”. Most of the time, this command does nothing. However, if there are too many loose objects (objects not in a packfile) or too many packfiles, Git launches a full-fledged `git gc` command. The “gc” stands for garbage collect, and the command does a number of things: it gathers up all the loose objects and places them in packfiles, it consolidates packfiles into one big packfile, and it removes objects that aren’t reachable from any commit and are a few months old.

You can run `auto gc` manually as follows:

```
$ git gc --auto
```

Again, this generally does nothing. You must have around 7,000 loose objects or more than 50 packfiles for Git to fire up a real `gc` command. You can modify these limits with the `gc.auto` and `gc.autopacklimit` config settings, respectively.

The other thing `gc` will do is pack up your references into a single file. Suppose your repository contains the following branches and tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

If you run `git gc`, you'll no longer have these files in the `refs` directory. Git will move them for the sake of efficiency into a file named `.git/packed-refs` that looks like this:

```
$ 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
```

If you update a reference, Git doesn't edit this file but instead writes a new file to `refs/heads`. To get the appropriate SHA-1 for a given reference, Git checks for that reference in the `refs` directory and then checks the `packed-refs` file as a fallback. However, if you can't find a reference in the `refs` directory, it's probably in your `packed-refs` file.

Notice the last line of the file, which begins with a `^`. This means the tag directly above is an annotated tag and that line is the commit that the annotated tag points to.

## Data Recovery

At some point in your Git journey, you may accidentally lose a commit. Generally, this happens because you force-delete a branch that had work on it, and it turns out you wanted the branch after all; or you hard-reset a branch, thus abandoning commits that you wanted something from. Assuming this happens, how can you get your commits back?

Here's an example that hard-resets the master branch in your test repository to an older commit and then recovers the lost commits. First, let's review where your repository is at this point:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Now, move the `master` branch back to the middle commit:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

You've effectively lost the top two commits – you have no branch from which those commits are reachable. You need to find the latest commit SHA-1 and then add a branch that points to it. The trick is finding that latest commit SHA-1 – it's not like you've memorized it, right?

Often, the quickest way is to use a tool called `git reflog`. As you're working, Git silently records what your HEAD is every time you change it. Each time you commit or change branches, the reflog is updated. The reflog is also updated by the `git update-ref` command, which is another reason to use it instead of just writing the SHA-1 value to your ref files, as we covered in [Git Referenzen](#). You can see where you've been at any time by running `git reflog`:

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to 1a410ef
ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

Here we can see the two commits that we have had checked out, however there is not much information here. To see the same information in a much more useful way, we can run `git log -g`, which will give you a normal log output for your reflog.

```
$ git log -g
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Reflog: HEAD@{0} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:22:37 2009 -0700

    third commit

commit ab1afef80fac8e34258ff41fc1b867c702daa24b
Reflog: HEAD@{1} (Scott Chacon <schacon@gmail.com>)
Reflog message: updating HEAD
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700

    modified repo.rb a bit
```

It looks like the bottom commit is the one you lost, so you can recover it by creating a new branch at that commit. For example, you can start a branch named `recover-branch` at that commit (ab1afef):

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19adb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Cool – now you have a branch named `recover-branch` that is where your `master` branch used to be, making the first two commits reachable again. Next, suppose your loss was for some reason not in the reflog – you can simulate that by removing `recover-branch` and deleting the reflog. Now the first two commits aren't reachable by anything:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Because the reflog data is kept in the `.git/logs/` directory, you effectively have no reflog. How can you recover that commit at this point? One way is to use the `git fsck` utility, which checks your database for integrity. If you run it with the `--full` option, it shows you all objects that aren't pointed to by another object:

```
$ 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 this case, you can see your missing commit after the string “dangling commit”. You can recover it the same way, by adding a branch that points to that SHA-1.

## Removing Objects

There are a lot of great things about Git, but one feature that can cause issues is the fact that a `git clone` downloads the entire history of the project, including every version of every file. This is fine if the whole thing is source code, because Git is highly optimized to compress that data efficiently. However, if someone at any point in the history of your project added a single huge file, every clone for all time will be forced to download that large file, even if it was removed from the project in the very next commit. Because it's reachable from the history, it will always be there.

This can be a huge problem when you're converting Subversion or Perforce repositories into Git. Because you don't download the whole history in those systems, this type of addition carries few consequences. If you did an import from another system or otherwise find that your repository is much larger than it should be, here is how you can find and remove large objects.

**Be warned: this technique is destructive to your commit history.** It rewrites every commit object since the earliest tree you have to modify to remove a large file reference. If you do this immediately after an import, before anyone has started to base work on the commit, you're fine – otherwise, you have to notify all contributors that they must rebase their work onto your new commits.

To demonstrate, you'll add a large file into your test repository, remove it in the next commit, find it, and remove it permanently from the repository. First, add a large object to your history:

```
$ curl https://www.kernel.org/pub/software/scm/git/git-2.1.0.tar.gz > git.tgz
$ git add git.tgz
$ git commit -m 'add git tarball'
[master 7b30847] add git tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.tgz
```

Oops – you didn't want to add a huge tarball to your project. Better get rid of it:

```
$ git rm git.tgz
rm 'git.tgz'
$ git commit -m 'oops - removed large tarball'
[master dadf725] oops - removed large tarball
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 git.tgz
```

Now, **gc** your database and see how much space you're using:

```
$ 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)
```

You can run the **count-objects** command to quickly see how much space you're using:

```
$ git count-objects -v
count: 7
size: 32
in-pack: 17
packs: 1
size-pack: 4868
prune-packable: 0
garbage: 0
size-garbage: 0
```

The `size-pack` entry is the size of your packfiles in kilobytes, so you're using almost 5MB. Before the last commit, you were using closer to 2K – clearly, removing the file from the previous commit didn't remove it from your history. Every time anyone clones this repository, they will have to clone all 5MB just to get this tiny project, because you accidentally added a big file. Let's get rid of it.

First you have to find it. In this case, you already know what file it is. But suppose you didn't; how would you identify what file or files were taking up so much space? If you run `git gc`, all the objects are in a packfile; you can identify the big objects by running another plumbing command called `git verify-pack` and sorting on the third field in the output, which is file size. You can also pipe it through the `tail` command because you're only interested in the last few largest files:

```
$ 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
```

The big object is at the bottom: 5MB. To find out what file it is, you'll use the `rev-list` command, which you used briefly in [Ein bestimmtes Commit-Message-Format erzwingen](#). If you pass `--objects` to `rev-list`, it lists all the commit SHA-1s and also the blob SHA-1s with the file paths associated with them. You can use this to find your blob's name:

```
$ git rev-list --objects --all | grep 82c99a3
82c99a3e86bb1267b236a4b6eff7868d97489af1 git.tgz
```

Now, you need to remove this file from all trees in your past. You can easily see what commits modified this file:

```
$ git log --oneline --branches -- git.tgz
dadf725 oops - removed large tarball
7b30847 add git tarball
```

You must rewrite all the commits downstream from [7b30847](#) to fully remove this file from your Git history. To do so, you use `filter-branch`, which you used in [Rewriting History](#):

```
$ 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
```

The `--index-filter` option is similar to the `--tree-filter` option used in [Rewriting History](#), except that instead of passing a command that modifies files checked out on disk, you're modifying your staging area or index each time.

Rather than remove a specific file with something like `rm file`, you have to remove it with `git rm --cached` – you must remove it from the index, not from disk. The reason to do it this way is speed – because Git doesn't have to check out each revision to disk before running your filter, the process can be much, much faster. You can accomplish the same task with `--tree-filter` if you want. The `--ignore-unmatch` option to `git rm` tells it not to error out if the pattern you're trying to remove isn't there. Finally, you ask `filter-branch` to rewrite your history only from the `7b30847` commit up, because you know that is where this problem started. Otherwise, it will start from the beginning and will unnecessarily take longer.

Your history no longer contains a reference to that file. However, your reflog and a new set of refs that Git added when you did the `filter-branch` under `.git/refs/original` still do, so you have to remove them and then repack the database. You need to get rid of anything that has a pointer to those old commits before you repack:

```
$ 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)
```

Let's see how much space you saved.

```
$ git count-objects -v
count: 11
size: 4904
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

The packed repository size is down to 8K, which is much better than 5MB. You can see from the size value that the big object is still in your loose objects, so it's not gone; but it won't be transferred on a push or subsequent clone, which is what is important. If you really wanted to, you could remove the object completely by running `git prune` with the `--expire` option:

```
$ git prune --expire now
$ git count-objects -v
count: 0
size: 0
in-pack: 15
packs: 1
size-pack: 8
prune-packable: 0
garbage: 0
size-garbage: 0
```

## Environment Variables

Git always runs inside a `bash` shell, and uses a number of shell environment variables to determine how it behaves. Occasionally, it comes in handy to know what these are, and how they can be used to make Git behave the way you want it to. This isn't an exhaustive list of all the environment variables Git pays attention to, but we'll cover the most useful.

### Global Behavior

Some of Git's general behavior as a computer program depends on environment variables.

`GIT_EXEC_PATH` determines where Git looks for its sub-programs (like `git-commit`, `git-diff`, and others). You can check the current setting by running `git --exec-path`.

`HOME` isn't usually considered customizable (too many other things depend on it), but it's where Git looks for the global configuration file. If you want a truly portable Git installation, complete with global configuration, you can override `HOME` in the portable Git's shell profile.

`PREFIX` is similar, but for the system-wide configuration. Git looks for this file at `$PREFIX/etc/gitconfig`.

`GIT_CONFIG_NOSYSTEM`, if set, disables the use of the system-wide configuration file. This is useful if your system config is interfering with your commands, but you don't have access to change or remove it.

`GIT_PAGER` controls the program used to display multi-page output on the command line. If this is unset, `PAGER` will be used as a fallback.

`GIT_EDITOR` is the editor Git will launch when the user needs to edit some text (a commit message, for example). If unset, `EDITOR` will be used.

### Repository Locations

Git uses several environment variables to determine how it interfaces with the current repository.

`GIT_DIR` is the location of the `.git` folder. If this isn't specified, Git walks up the directory tree until it gets to `~` or `/`, looking for a `.git` directory at every step.

**GIT\_CEILING\_DIRECTORIES** controls the behavior of searching for a `.git` directory. If you access directories that are slow to load (such as those on a tape drive, or across a slow network connection), you may want to have Git stop trying earlier than it might otherwise, especially if Git is invoked when building your shell prompt.

**GIT\_WORK\_TREE** is the location of the root of the working directory for a non-bare repository. If `--git-dir` or **GIT\_DIR** is specified but none of `--work-tree`, **GIT\_WORK\_TREE** or `core.worktree` is specified, the current working directory is regarded as the top level of your working tree.

**GIT\_INDEX\_FILE** is the path to the index file (non-bare repositories only).

**GIT\_OBJECT\_DIRECTORY** can be used to specify the location of the directory that usually resides at `.git/objects`.

**GIT\_ALTERNATE\_OBJECT\_DIRECTORIES** is a colon-separated list (formatted like `/dir/one:/dir/two:…`) which tells Git where to check for objects if they aren't in **GIT\_OBJECT\_DIRECTORY**. If you happen to have a lot of projects with large files that have the exact same contents, this can be used to avoid storing too many copies of them.

## Pathspecs

A “pathspec” refers to how you specify paths to things in Git, including the use of wildcards. These are used in the `.gitignore` file, but also on the command-line (`git add *.c`).

**GIT\_GLOB\_PATHSPECS** and **GIT\_NOGLOB\_PATHSPECS** control the default behavior of wildcards in pathspecs. If **GIT\_GLOB\_PATHSPECS** is set to 1, wildcard characters act as wildcards (which is the default); if **GIT\_NOGLOB\_PATHSPECS** is set to 1, wildcard characters only match themselves, meaning something like `*.c` would only match a file named “`*.c`”, rather than any file whose name ends with `.c`. You can override this in individual cases by starting the pathspec with `:(glob)` or `:(literal)`, as in `:(glob)*.c`.

**GIT\_LITERAL\_PATHSPECS** disables both of the above behaviors; no wildcard characters will work, and the override prefixes are disabled as well.

**GIT\_ICASE\_PATHSPECS** sets all pathspecs to work in a case-insensitive manner.

## Committing

The final creation of a Git commit object is usually done by `git-commit-tree`, which uses these environment variables as its primary source of information, falling back to configuration values only if these aren't present.

**GIT\_AUTHOR\_NAME** is the human-readable name in the “author” field.

**GIT\_AUTHOR\_EMAIL** is the email for the “author” field.

**GIT\_AUTHOR\_DATE** is the timestamp used for the “author” field.

**GIT\_COMMITTER\_NAME** sets the human name for the “committer” field.

**GIT\_COMMITTER\_EMAIL** is the email address for the “committer” field.

**GIT\_COMMITTER\_DATE** is used for the timestamp in the “committer” field.

**EMAIL** is the fallback email address in case the `user.email` configuration value isn’t set. If *this* isn’t set, Git falls back to the system user and host names.

## Networking

Git uses the `curl` library to do network operations over HTTP, so **GIT\_CURL\_VERBOSE** tells Git to emit all the messages generated by that library. This is similar to doing `curl -v` on the command line.

**GIT\_SSL\_NO\_VERIFY** tells Git not to verify SSL certificates. This can sometimes be necessary if you’re using a self-signed certificate to serve Git repositories over HTTPS, or you’re in the middle of setting up a Git server but haven’t installed a full certificate yet.

If the data rate of an HTTP operation is lower than **GIT\_HTTP\_LOW\_SPEED\_LIMIT** bytes per second for longer than **GIT\_HTTP\_LOW\_SPEED\_TIME** seconds, Git will abort that operation. These values override the `http.lowSpeedLimit` and `http.lowSpeedTime` configuration values.

**GIT\_HTTP\_USER\_AGENT** sets the user-agent string used by Git when communicating over HTTP. The default is a value like `git/2.0.0`.

## Diffing and Merging

**GIT\_DIFF\_OPTS** is a bit of a misnomer. The only valid values are `-u<n>` or `--unified=<n>`, which controls the number of context lines shown in a `git diff` command.

**GIT\_EXTERNAL\_DIFF** is used as an override for the `diff.external` configuration value. If it’s set, Git will invoke this program when `git diff` is invoked.

**GIT\_DIFF\_PATH\_COUNTER** and **GIT\_DIFF\_PATH\_TOTAL** are useful from inside the program specified by **GIT\_EXTERNAL\_DIFF** or `diff.external`. The former represents which file in a series is being diffed (starting with 1), and the latter is the total number of files in the batch.

**GIT\_MERGE\_VERBOSITY** controls the output for the recursive merge strategy. The allowed values are as follows:

- 0 outputs nothing, except possibly a single error message.
- 1 shows only conflicts.
- 2 also shows file changes.
- 3 shows when files are skipped because they haven’t changed.
- 4 shows all paths as they are processed.
- 5 and above show detailed debugging information.

The default value is 2.

## Debugging

Want to *really* know what Git is up to? Git has a fairly complete set of traces embedded, and all you

need to do is turn them on. The possible values of these variables are as follows:

- “true”, “1”, or “2” – the trace category is written to stderr.
- An absolute path starting with / – the trace output will be written to that file.

**GIT\_TRACE** controls general traces, which don’t fit into any specific category. This includes the expansion of aliases, and delegation to other sub-programs.

```
$ 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** controls tracing of packfile access. The first field is the packfile being accessed, the second is the offset within that file:

```
$ 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** enables packet-level tracing for network operations.

```
$ 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<
36dc827bc9d17f80ed4f326de21247a5d1341fb refs/heads/ah/doc-gitk-config
# [...]
```

**GIT\_TRACE\_PERFORMANCE** controls logging of performance data. The output shows how long each particular `git` invocation takes.

```
$ 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'
'rere' 'gc'
20:18:25.233159 trace.c:414          performance: 6.112217000 s: git command: 'git'
'gc'
```

**GIT\_TRACE\_SETUP** shows information about what Git is discovering about the repository and environment it's interacting with.

```
$ 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
```

## Miscellaneous

**GIT\_SSH**, if specified, is a program that is invoked instead of `ssh` when Git tries to connect to an SSH host. It is invoked like `$GIT_SSH [username@]host [-p <port>] <command>`. Note that this isn't the easiest way to customize how `ssh` is invoked; it won't support extra command-line parameters, so you'd have to write a wrapper script and set `GIT_SSH` to point to it. It's probably easier just to use the `~/.ssh/config` file for that.

**GIT\_ASKPASS** is an override for the `core.askpass` configuration value. This is the program invoked whenever Git needs to ask the user for credentials, which can expect a text prompt as a command-line argument, and should return the answer on `stdout`. (See [Anmeldeinformationen speichern](#) for more on this subsystem.)

**GIT\_NAMESPACE** controls access to namespaced refs, and is equivalent to the `--namespace` flag. This is mostly useful on the server side, where you may want to store multiple forks of a single repository in one repository, only keeping the refs separate.

**GIT\_FLUSH** can be used to force Git to use non-buffered I/O when writing incrementally to `stdout`. A value of 1 causes Git to flush more often, a value of 0 causes all output to be buffered. The default value (if this variable is not set) is to choose an appropriate buffering scheme depending on the activity and the output mode.

**GIT\_REFLOG\_ACTION** lets you specify the descriptive text written to the reflog. Here's an example:

```
$ 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 Other Environments

If you read through the whole book, you've learned a lot about how to use Git at the command line. You can work with local files, connect your repository to others over a network, and work effectively with others. But the story doesn't end there; Git is usually used as part of a larger ecosystem, and the terminal isn't always the best way to work with it. Now we'll take a look at some of the other kinds of environments where Git can be useful, and how other applications (including yours) work alongside Git.

## Graphical Interfaces

Git's native environment is in the terminal. New features show up there first, and only at the command line is the full power of Git completely at your disposal. But plain text isn't the best choice for all tasks; sometimes a visual representation is what you need, and some users are much more comfortable with a point-and-click interface.

It's important to note that different interfaces are tailored for different workflows. Some clients expose only a carefully curated subset of Git functionality, in order to support a specific way of working that the author considers effective. When viewed in this light, none of these tools can be called "better" than any of the others, they're simply more fit for their intended purpose. Also note that there's nothing these graphical clients can do that the command-line client can't; the command-line is still where you'll have the most power and control when working with your repositories.

### gitk and git-gui

When you install Git, you also get its visual tools, `gitk` and `git-gui`.

`gitk` is a graphical history viewer. Think of it like a powerful GUI shell over `git log` and `git grep`. This is the tool to use when you're trying to find something that happened in the past, or visualize your project's history.

Gitk is easiest to invoke from the command-line. Just `cd` into a Git repository, and type:

```
$ gitk [git log options]
```

Gitk accepts many command-line options, most of which are passed through to the underlying `git log` action. Probably one of the most useful is the `--all` flag, which tells gitk to show commits reachable from *any* ref, not just HEAD. Gitk's interface looks like this:

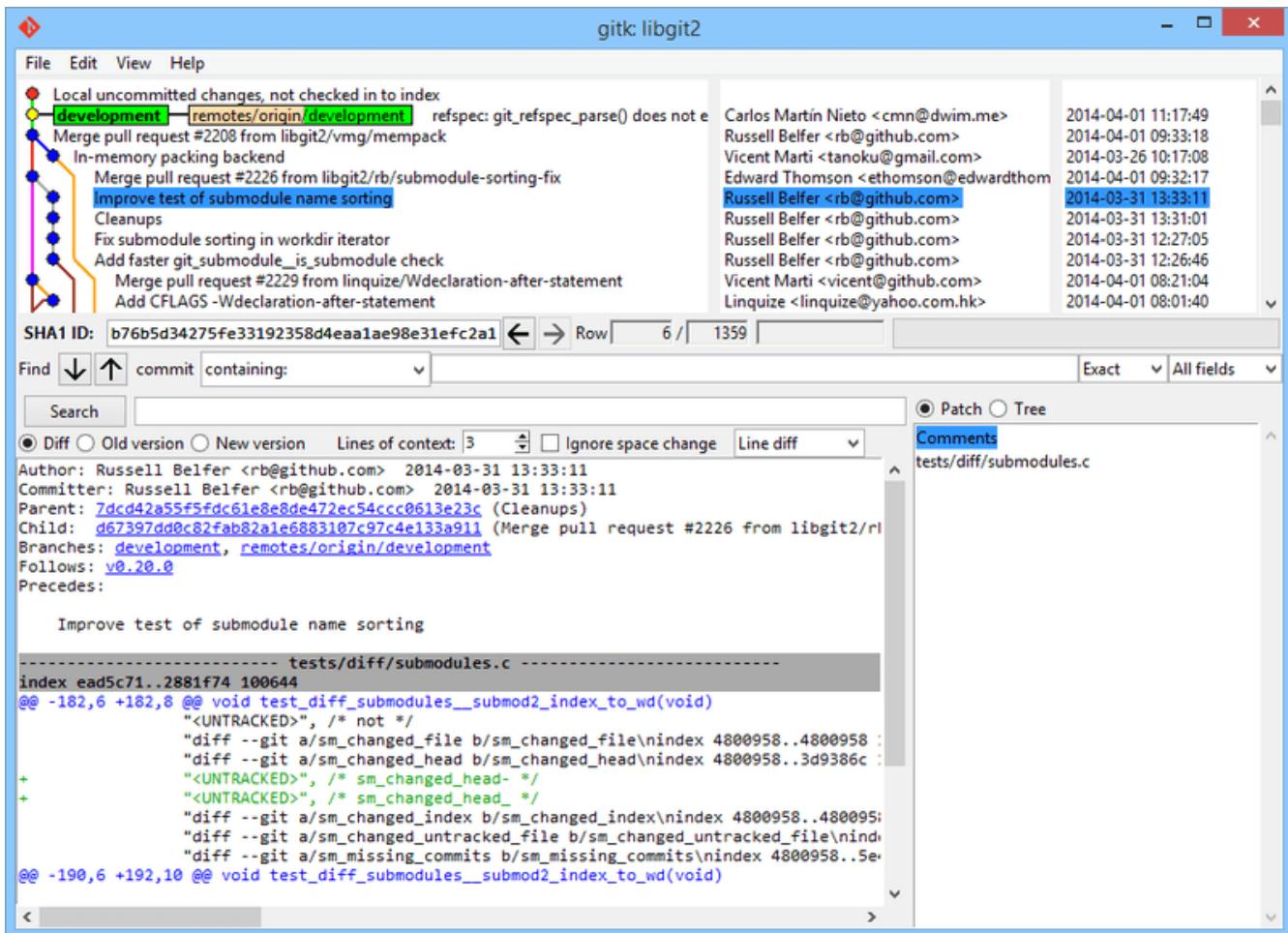


Figure 152. The `gitk` history viewer.

On the top is something that looks a bit like the output of `git log --graph`; each dot represents a commit, the lines represent parent relationships, and refs are shown as colored boxes. The yellow dot represents HEAD, and the red dot represents changes that are yet to become a commit. At the bottom is a view of the selected commit; the comments and patch on the left, and a summary view on the right. In between is a collection of controls used for searching history.

`git-gui`, on the other hand, is primarily a tool for crafting commits. It, too, is easiest to invoke from the command line:

```
$ git gui
```

And it looks something like this:

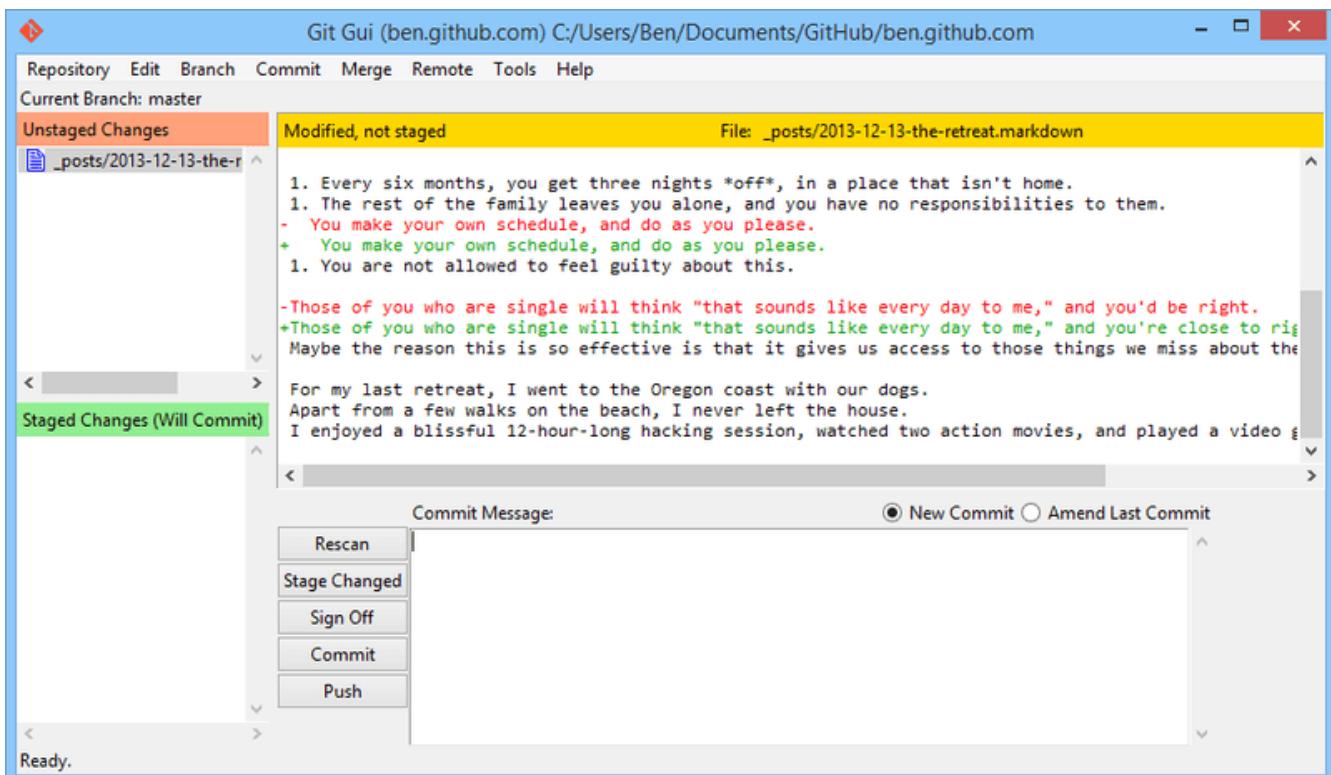


Figure 153. The `git-gui` commit tool.

On the left is the index; unstaged changes are on top, staged changes on the bottom. You can move entire files between the two states by clicking on their icons, or you can select a file for viewing by clicking on its name.

At top right is the diff view, which shows the changes for the currently-selected file. You can stage individual hunks (or individual lines) by right-clicking in this area.

At the bottom right is the message and action area. Type your message into the text box and click “Commit” to do something similar to `git commit`. You can also choose to amend the last commit by choosing the “Amend” radio button, which will update the “Staged Changes” area with the contents of the last commit. Then you can simply stage or unstage some changes, alter the commit message, and click “Commit” again to replace the old commit with a new one.

`gitk` and `git-gui` are examples of task-oriented tools. Each of them is tailored for a specific purpose (viewing history and creating commits, respectively), and omit the features not necessary for that task.

## GitHub for macOS and Windows

GitHub has created two workflow-oriented Git clients: one for Windows, and one for macOS. These clients are a good example of workflow-oriented tools – rather than expose *all* of Git’s functionality, they instead focus on a curated set of commonly-used features that work well together. They look like this:

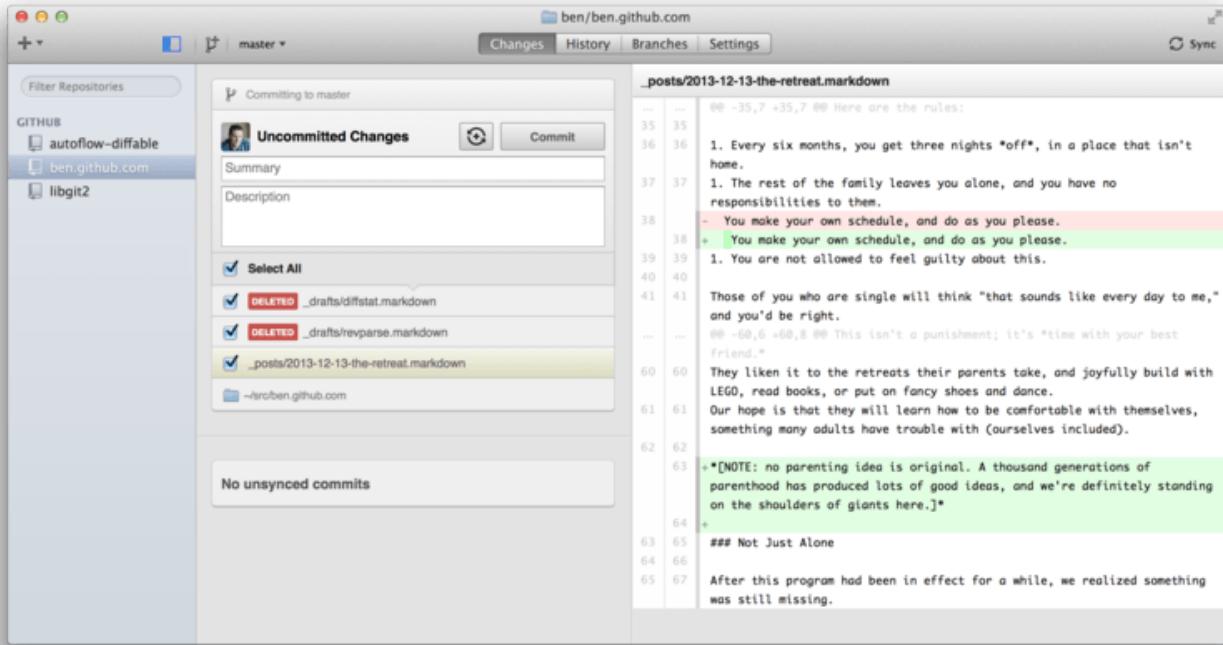


Figure 154. GitHub for macOS.

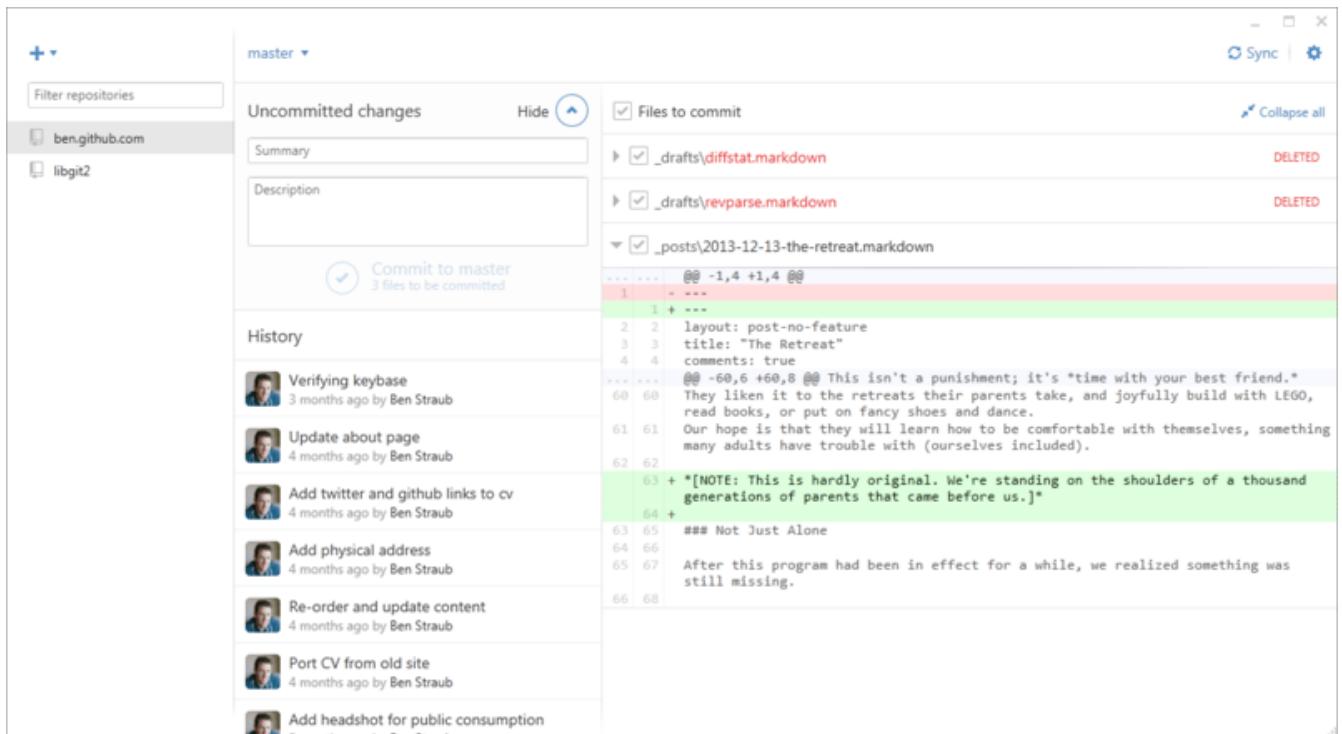


Figure 155. GitHub for Windows.

They are designed to look and work very much alike, so we'll treat them like a single product in this chapter. We won't be doing a detailed rundown of these tools (they have their own documentation), but a quick tour of the "changes" view (which is where you'll spend most of your time) is in order.

- On the left is the list of repositories the client is tracking; you can add a repository (either by cloning or attaching locally) by clicking the "+" icon at the top of this area.
- In the center is a commit-input area, which lets you input a commit message, and select which files should be included. (On Windows, the commit history is displayed directly below this; on

macOS, it's on a separate tab.)

- On the right is a diff view, which shows what's changed in your working directory, or which changes were included in the selected commit.
- The last thing to notice is the “Sync” button at the top-right, which is the primary way you interact over the network.



You don't need a GitHub account to use these tools. While they're designed to highlight GitHub's service and recommended workflow, they will happily work with any repository, and do network operations with any Git host.

## Installation

GitHub for Windows can be downloaded from <https://windows.github.com>, and GitHub for macOS from <https://mac.github.com>. When the applications are first run, they walk you through all the first-time Git setup, such as configuring your name and email address, and both set up sane defaults for many common configuration options, such as credential caches and CRLF behavior.

Both are “evergreen” – updates are downloaded and installed in the background while the applications are open. This helpfully includes a bundled version of Git, which means you probably won't have to worry about manually updating it again. On Windows, the client includes a shortcut to launch PowerShell with Posh-git, which we'll talk more about later in this chapter.

The next step is to give the tool some repositories to work with. The client shows you a list of the repositories you have access to on GitHub, and can clone them in one step. If you already have a local repository, just drag its directory from the Finder or Windows Explorer into the GitHub client window, and it will be included in the list of repositories on the left.

## Recommended Workflow

Once it's installed and configured, you can use the GitHub client for many common Git tasks. The intended workflow for this tool is sometimes called the “GitHub Flow.” We cover this in more detail in [Der GitHub Workflow](#), but the general gist is that (a) you'll be committing to a branch, and (b) you'll be syncing up with a remote repository fairly regularly.

Branch management is one of the areas where the two tools diverge. On macOS, there's a button at the top of the window for creating a new branch:

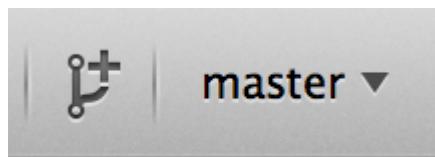


Figure 156. “Create Branch” button on macOS.

On Windows, this is done by typing the new branch's name in the branch-switching widget:

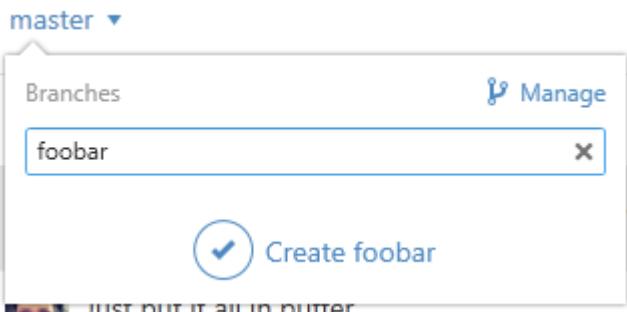


Figure 157. Creating a branch on Windows.

Once your branch is created, making new commits is fairly straightforward. Make some changes in your working directory, and when you switch to the GitHub client window, it will show you which files changed. Enter a commit message, select the files you'd like to include, and click the “Commit” button (ctrl-enter or ⌘-enter).

The main way you interact with other repositories over the network is through the “Sync” feature. Git internally has separate operations for pushing, fetching, merging, and rebasing, but the GitHub clients collapse all of these into one multi-step feature. Here’s what happens when you click the Sync button:

1. `git pull --rebase`. If this fails because of a merge conflict, fall back to `git pull --no-rebase`.
2. `git push`.

This is the most common sequence of network commands when working in this style, so squashing them into one command saves a lot of time.

## Summary

These tools are very well-suited for the workflow they’re designed for. Developers and non-developers alike can be collaborating on a project within minutes, and many of the best practices for this kind of workflow are baked into the tools. However, if your workflow is different, or you want more control over how and when network operations are done, we recommend you use another client or the command line.

## Other GUIs

There are a number of other graphical Git clients, and they run the gamut from specialized, single-purpose tools all the way to apps that try to expose everything Git can do. The official Git website has a curated list of the most popular clients at <https://git-scm.com/downloads/guis>. A more comprehensive list is available on the Git wiki site, at [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

Starting with Visual Studio 2013 Update 1, Visual Studio users have a Git client built directly into their IDE. Visual Studio has had source-control integration features for quite some time, but they were oriented towards centralized, file-locking systems, and Git was not a good match for this workflow. Visual Studio 2013’s Git support has been separated from this older feature, and the

result is a much better fit between Studio and Git.

To locate the feature, open a project that's controlled by Git (or just `git init` an existing project), and select View > Team Explorer from the menu. You'll see the "Connect" view, which looks a bit like this:

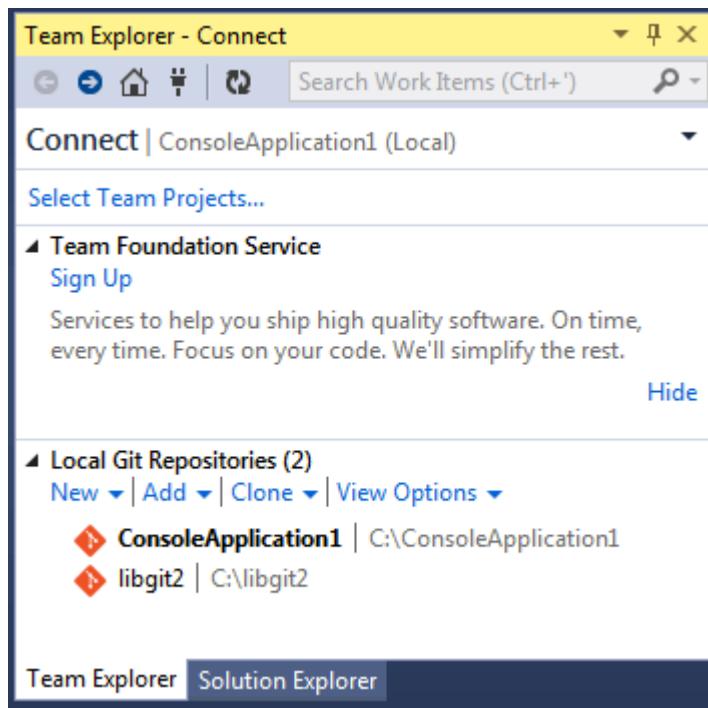


Figure 158. Connecting to a Git repository from Team Explorer.

Visual Studio remembers all of the projects you've opened that are Git-controlled, and they're available in the list at the bottom. If you don't see the one you want there, click the "Add" link and type in the path to the working directory. Double clicking on one of the local Git repositories leads you to the Home view, which looks like [The "Home" view for a Git repository in Visual Studio..](#) This is a hub for performing Git actions; when you're *writing* code, you'll probably spend most of your time in the "Changes" view, but when it comes time to pull down changes made by your teammates, you'll use the "Unsynced Commits" and "Branches" views.

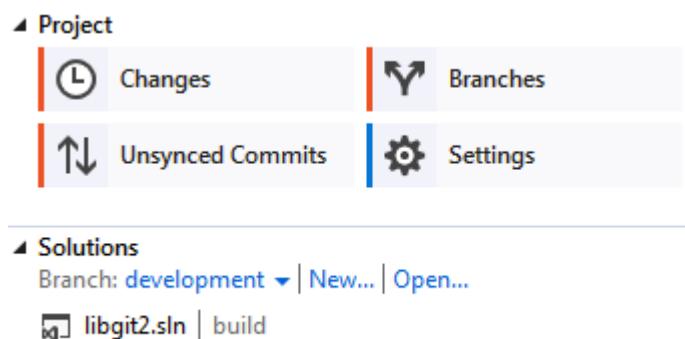


Figure 159. The "Home" view for a Git repository in Visual Studio.

Visual Studio now has a powerful task-focused UI for Git. It includes a linear history view, a diff viewer, remote commands, and many other capabilities. For complete documentation of this feature (which doesn't fit here), go to <http://msdn.microsoft.com/en-us/library/hh850437.aspx>.

# Git in Visual Studio Code

Visual Studio Code has git support built in. You will need to have git version 2.0.0 (or newer) installed.

The main features are:

- See the diff of the file you are editing in the gutter.
- The Git Status Bar (lower left) shows the current branch, dirty indicators, incoming and outgoing commits.
- You can do the most common git operations from within the editor:
  - Initialize a repository.
  - Clone a repository.
  - Create branches and tags.
  - Stage and commit changes.
  - Push/pull/sync with a remote branch.
  - Resolve merge conflicts.
  - View diffs.
- With a extension, you can also handle GitHub Pull Requests: <https://marketplace.visualstudio.com/items?itemName=GitHub.vscode-pull-request-github>

The official documentation can be found here: <https://code.visualstudio.com/Docs/editor/versioncontrol>

# Git in Eclipse

Eclipse ships with a plugin called Egit, which provides a fairly-complete interface to Git operations. It's accessed by switching to the Git Perspective (Window > Open Perspective > Other..., and select "Git").

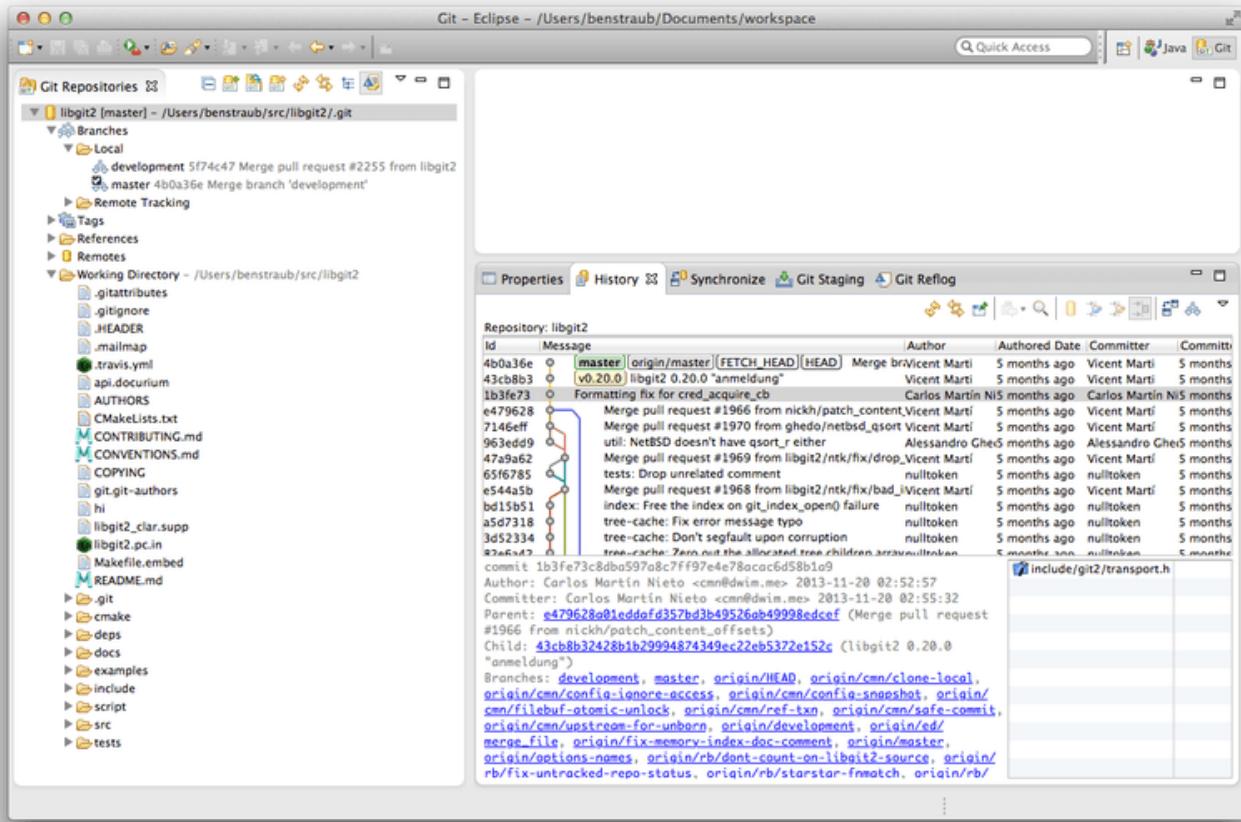


Figure 160. Eclipse's EGit environment.

EGit comes with plenty of great documentation, which you can find by going to Help > Help Contents, and choosing the "EGit Documentation" node from the contents listing.

## Git in Sublime Text

From version 3.2 onwards, Sublime Text has git integration in the editor.

The features are:

- The sidebar will show the git status of files and folders with a badge/icon.
- Files and folders that are in your .gitignore file will be faded out in the sidebar.
- In the status bar, you can see the current git branch and how many modifications you have made.
- All changes to a file are now visible via markers in the gutter.
- You can use part of the Sublime Merge git client functionality from within Sublime Text. (This requires that Sublime Merge is installed. See: <https://www.sublimemerge.com/>)

The official documentation for Sublime Text can be found here: [https://www.sublimetext.com/docs/3/git\\_integration.html](https://www.sublimetext.com/docs/3/git_integration.html)

## Git in Bash

If you're a Bash user, you can tap into some of your shell's features to make your experience with

Git a lot friendlier. Git actually ships with plugins for several shells, but it's not turned on by default.

First, you need to get a copy of the `contrib/completion/git-completion.bash` file out of the Git source code. Copy it somewhere handy, like your home directory, and add this to your `.bashrc`:

```
. ~/git-completion.bash
```

Once that's done, change your directory to a Git repository, and type:

```
$ git chec<tab>
```

...and Bash will auto-complete to `git checkout`. This works with all of Git's subcommands, command-line parameters, and remotes and ref names where appropriate.

It's also useful to customize your prompt to show information about the current directory's Git repository. This can be as simple or complex as you want, but there are generally a few key pieces of information that most people want, like the current branch, and the status of the working directory. To add these to your prompt, just copy the `contrib/completion/git-prompt.sh` file from Git's source repository to your home directory, add something like this to your `.bashrc`:

```
. ~/git-prompt.sh
export GIT_PS1_SHOWDIRTYSTATE=1
export PS1='\w$(_git_ps1 " (%s)")\$ '
```

The `\w` means print the current working directory, the `\$` prints the `$` part of the prompt, and `_git_ps1 " (%s)"` calls the function provided by `git-prompt.sh` with a formatting argument. Now your bash prompt will look like this when you're anywhere inside a Git-controlled project:



Figure 161. Customized `bash` prompt.

Both of these scripts come with helpful documentation; take a look at the contents of `git-completion.bash` and `git-prompt.sh` for more information.

# Git in Zsh

Zsh also ships with a tab-completion library for Git. To use it, simply run `autoload -Uz compinit && compinit` in your `.zshrc`. Zsh's interface is a bit more powerful than Bash's:

```
$ 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
```

Ambiguous tab-completions aren't just listed; they have helpful descriptions, and you can graphically navigate the list by repeatedly hitting tab. This works with Git commands, their arguments, and names of things inside the repository (like refs and remotes), as well as filenames and all the other things Zsh knows how to tab-complete.

Zsh ships with a framework for getting information from version control systems, called `vcs_info`. To include the branch name in the prompt on the right side, add these lines to your `~/.zshrc` file:

```
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'
```

This results in a display of the current branch on the right-hand side of the terminal window, whenever your shell is inside a Git repository. (The left side is supported as well, of course; just uncomment the assignment to `PROMPT`.) It looks a bit like this:

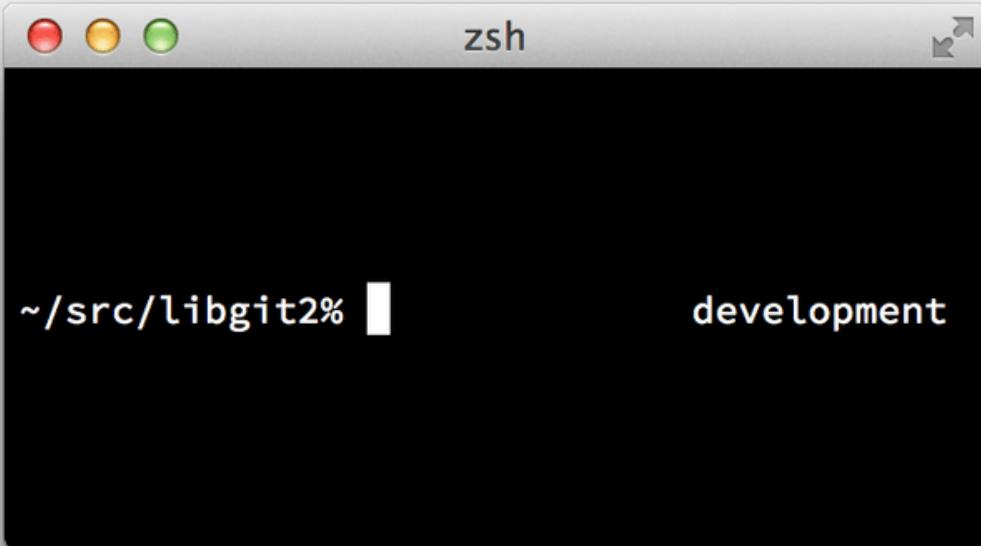


Figure 162. Customized `zsh` prompt.

For more information on `vcs_info`, check out its documentation in the `zshcontrib(1)` manual page, or online at <http://zsh.sourceforge.net/Doc/Release/User-Contributions.html#Version-Control-Information>.

Instead of `vcs_info`, you might prefer the prompt customization script that ships with Git, called `git-prompt.sh`; see <https://github.com/git/git/blob/master/contrib/completion/git-prompt.sh> for details. `git-prompt.sh` is compatible with both Bash and Zsh.

Zsh is powerful enough that there are entire frameworks dedicated to making it better. One of them is called "oh-my-zsh", and it can be found at <https://github.com/robbyrussell/oh-my-zsh>. oh-my-zsh's plugin system comes with powerful git tab-completion, and it has a variety of prompt "themes", many of which display version-control data. [An example of an oh-my-zsh theme.](#) is just one example of what can be done with this system.

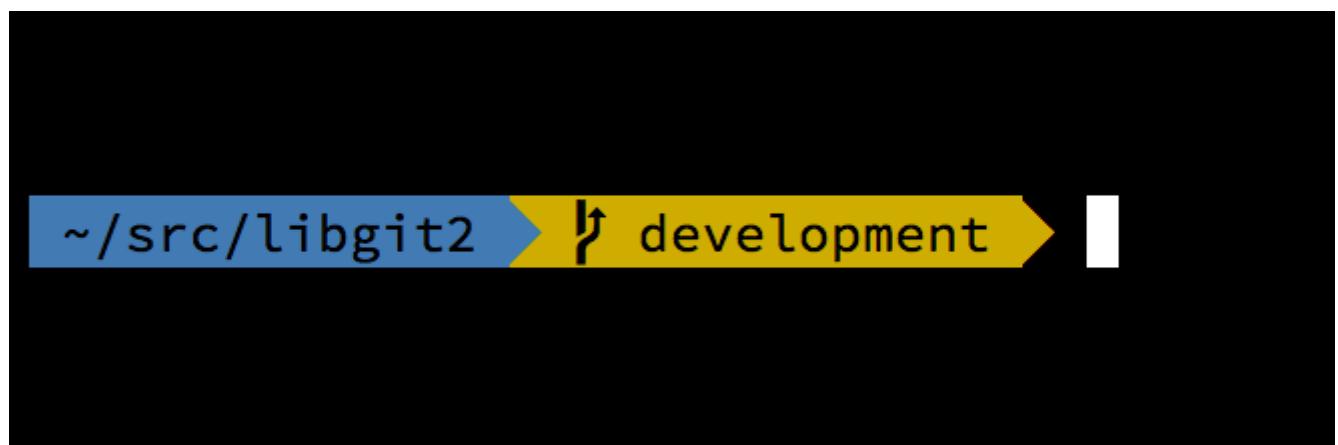


Figure 163. An example of an `oh-my-zsh` theme.

# Git in PowerShell

The legacy command-line terminal on Windows (`cmd.exe`) isn't really capable of a customized Git experience, but if you're using PowerShell, you're in luck. This also works if you're running PowerShell Core on Linux or macOS. A package called posh-git (<https://github.com/dahlbyk/posh-git>) provides powerful tab-completion facilities, as well as an enhanced prompt to help you stay on top of your repository status. It looks like this:

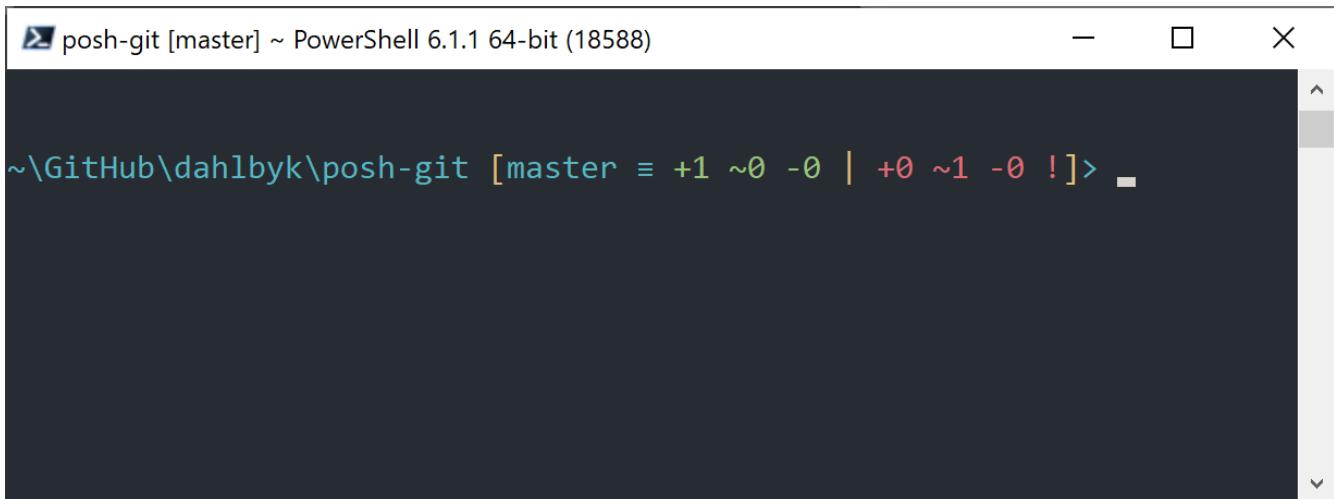
A screenshot of a PowerShell window titled "posh-git [master] ~ PowerShell 6.1.1 64-bit (18588)". The window shows a dark-themed interface with a light-colored status bar at the top. In the main area, there is a single line of text: `~\GitHub\dahlbyk\posh-git [master ≡ +1 ~0 -0 | +0 ~1 -0 !]>`.

Figure 164. PowerShell with Posh-git.

## Installation

### Prerequisites (Windows only)

Before you're able to run PowerShell scripts on your machine, you need to set your local `ExecutionPolicy` to `RemoteSigned` (Basically anything except `Undefined` and `Restricted`). If you choose `AllSigned` instead of `RemoteSigned`, also local scripts (your own) need to be digitally signed in order to be executed. With `RemoteSigned`, only Scripts having the "ZoneIdentifier" set to Internet (were downloaded from the web) need to be signed, others not. If you're an administrator and want to set it for all Users on that machine, use "`-Scope LocalMachine`". If you're a normal user, without administrative rights, you can use "`-Scope CurrentUser`" to set it only for you.

More about PowerShell Scopes: [https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about\\_scopes](https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_scopes)

More about PowerShell ExecutionPolicy: <https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy>

```
> Set-ExecutionPolicy -Scope LocalMachine -ExecutionPolicy RemoteSigned -Force
```

### PowerShell Gallery

If you have at least PowerShell 5 or PowerShell 4 with `PackageManagement` installed, you can use the package manager to install posh-git for you.

More information about PowerShell Gallery: <https://docs.microsoft.com/en-us/powershell/gallery/>

## overview

```
> Install-Module posh-git -Scope CurrentUser -Force  
> Install-Module posh-git -Scope CurrentUser -AllowPrerelease -Force # Newer beta  
version with PowerShell Core support
```

If you want to install posh-git for all users, use "-Scope AllUsers" instead and execute the command from an elevated PowerShell console. If the second command fails with an error like **Module 'PowerShellGet' was not installed by using Install-Module**, you'll need to run another command first:

```
> Install-Module PowerShellGet -Force -SkipPublisherCheck
```

Then you can go back and try again. This happens, because the modules that ship with Windows PowerShell are signed with a different publication certificate.

## Update PowerShell Prompt

To include git information in your prompt, the posh-git module needs to be imported. To have posh-git imported every time PowerShell starts, execute the Add-PoshGitToProfile command which will add the import statement into your \$profile script. This script is executed everytime you open a new PowerShell console. Keep in mind, that there are multiple \$profile scripts. E. g. one for the console and a separate one for the ISE.

```
> Import-Module posh-git  
> Add-PoshGitToProfile -AllHosts
```

## From Source

Just download a posh-git release from (<https://github.com/dahlbyk/posh-git>), and uncompress it. Then import the module using the full path to the posh-git.psd1 file:

```
> Import-Module <path-to-uncompress-folder>\src\posh-git.psd1  
> Add-PoshGitToProfile -AllHosts
```

This will add the proper line to your **profile.ps1** file, and posh-git will be active the next time you open PowerShell. For a description of the Git status summary information displayed in the prompt see: <https://github.com/dahlbyk/posh-git/blob/master/README.md#git-status-summary-information> For more details on how to customize your posh-git prompt see: <https://github.com/dahlbyk/posh-git/blob/master/README.md#customization-variables>

## Summary

You've learned how to harness Git's power from inside the tools that you use during your everyday work, and also how to access Git repositories from your own programs.

# Appendix B: Embedding Git in your Applications

If your application is for developers, chances are good that it could benefit from integration with source control. Even non-developer applications, such as document editors, could potentially benefit from version-control features, and Git's model works very well for many different scenarios.

If you need to integrate Git with your application, you have essentially two options: spawn a shell and call the `git` command-line program, or embed a Git library into your application. Here we'll cover command-line integration and several of the most popular embeddable Git libraries.

## Command-line Git

One option is to spawn a shell process and use the Git command-line tool to do the work. This has the benefit of being canonical, and all of Git's features are supported. This also happens to be fairly easy, as most runtime environments have a relatively simple facility for invoking a process with command-line arguments. However, this approach does have some downsides.

One is that all the output is in plain text. This means that you'll have to parse Git's occasionally-changing output format to read progress and result information, which can be inefficient and error-prone.

Another is the lack of error recovery. If a repository is corrupted somehow, or the user has a malformed configuration value, Git will simply refuse to perform many operations.

Yet another is process management. Git requires you to maintain a shell environment on a separate process, which can add unwanted complexity. Trying to coordinate many of these processes (especially when potentially accessing the same repository from several processes) can be quite a challenge.

## Libgit2

Another option at your disposal is to use Libgit2. Libgit2 is a dependency-free implementation of Git, with a focus on having a nice API for use within other programs. You can find it at <https://libgit2.org>.

First, let's take a look at what the C API looks like. Here's a whirlwind 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);

```

The first couple of lines open a Git repository. The `git_repository` type represents a handle to a repository with a cache in memory. This is the simplest method, for when you know the exact path to a repository’s working directory or `.git` folder. There’s also the `git_repository_open_ext` which includes options for searching, `git_clone` and friends for making a local clone of a remote repository, and `git_repository_init` for creating an entirely new repository.

The second chunk of code uses rev-parse syntax (see [Branch Referenzen](#) for more on this) to get the commit that HEAD eventually points to. The type returned is a `git_object` pointer, which represents something that exists in the Git object database for a repository. `git_object` is actually a “parent” type for several different kinds of objects; the memory layout for each of the “child” types is the same as for `git_object`, so you can safely cast to the right one. In this case, `git_object_type(commit)` would return `GIT_OBJ_COMMIT`, so it’s safe to cast to a `git_commit` pointer.

The next chunk shows how to access the commit’s properties. The last line here uses a `git_oid` type; this is Libgit2’s representation for a SHA-1 hash.

From this sample, a couple of patterns have started to emerge:

- If you declare a pointer and pass a reference to it into a Libgit2 call, that call will probably return an integer error code. A `0` value indicates success; anything less is an error.
- If Libgit2 populates a pointer for you, you’re responsible for freeing it.
- If Libgit2 returns a `const` pointer from a call, you don’t have to free it, but it will become invalid when the object it belongs to is freed.
- Writing C is a bit painful.

That last one means it isn’t very probable that you’ll be writing C when using Libgit2. Fortunately, there are a number of language-specific bindings available that make it fairly easy to work with Git repositories from your specific language and environment. Let’s take a look at the above example written using the Ruby bindings for Libgit2, which are named Rugged, and can be found at

<https://github.com/libgit2/rugged>.

```
repo = Rugged::Repository.new('path/to/repository')
commit = repo.head.target
puts commit.message
puts "#{commit.author[:name]} <#{commit.author[:email]}>"
tree = commit.tree
```

As you can see, the code is much less cluttered. Firstly, Rugged uses exceptions; it can raise things like `ConfigError` or `ObjectError` to signal error conditions. Secondly, there's no explicit freeing of resources, since Ruby is garbage-collected. Let's take a look at a slightly more complicated example: crafting a commit from scratch

```
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 Create a new blob, which contains the contents of a new file.
- ② Populate the index with the head commit's tree, and add the new file at the path `newfile.txt`.
- ③ This creates a new tree in the ODB, and uses it for the new commit.
- ④ We use the same signature for both the author and committer fields.
- ⑤ The commit message.
- ⑥ When creating a commit, you have to specify the new commit's parents. This uses the tip of HEAD for the single parent.
- ⑦ Rugged (and Libgit2) can optionally update a reference when making a commit.
- ⑧ The return value is the SHA-1 hash of a new commit object, which you can then use to get a `Commit` object.

The Ruby code is nice and clean, but since Libgit2 is doing the heavy lifting, this code will run pretty fast, too. If you’re not a rubyist, we touch on some other bindings in [Other Bindings](#).

## Advanced Functionality

Libgit2 has a couple of capabilities that are outside the scope of core Git. One example is pluggability: Libgit2 allows you to provide custom “backends” for several types of operation, so you can store things in a different way than stock Git does. Libgit2 allows custom backends for configuration, ref storage, and the object database, among other things.

Let’s take a look at how this works. The code below is borrowed from the set of backend examples provided by the Libgit2 team (which can be found at <https://github.com/libgit2/libgit2-backends>). Here’s how a custom backend for the object database is set up:

```
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
```

(Note that errors are captured, but not handled. We hope your code is better than ours.)

- ① Initialize an empty object database (ODB) “frontend,” which will act as a container for the “backends” which are the ones doing the real work.
- ② Initialize a custom ODB backend.
- ③ Add the backend to the frontend.
- ④ Open a repository, and set it to use our ODB to look up objects.

But what is this `git_odb_backend_mine` thing? Well, that’s the constructor for your own ODB implementation, and you can do whatever you want in there, so long as you fill in the `git_odb_backend` structure properly. Here’s what it *could* look like:

```

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;
}

```

The subtlest constraint here is that `my_backend_struct`'s first member must be a `git_odb_backend` structure; this ensures that the memory layout is what the Libgit2 code expects it to be. The rest of it is arbitrary; this structure can be as large or small as you need it to be.

The initialization function allocates some memory for the structure, sets up the custom context, and then fills in the members of the `parent` structure that it supports. Take a look at the `include/git2/sys/odb_backend.h` file in the Libgit2 source for a complete set of call signatures; your particular use case will help determine which of these you'll want to support.

## Other Bindings

Libgit2 has bindings for many languages. Here we show a small example using a few of the more complete bindings packages as of this writing; libraries exist for many other languages, including C++, Go, Node.js, Erlang, and the JVM, all in various stages of maturity. The official collection of bindings can be found by browsing the repositories at <https://github.com/libgit2>. The code we'll write will return the commit message from the commit eventually pointed to by HEAD (sort of like `git log -1`).

### LibGit2Sharp

If you're writing a .NET or Mono application, LibGit2Sharp (<https://github.com/libgit2/libgit2sharp>) is what you're looking for. The bindings are written in C#, and great care has been taken to wrap the raw Libgit2 calls with native-feeling CLR APIs. Here's what our example program looks like:

```
new Repository(@"C:\path\to\repo").Head.Tip.Message;
```

For desktop Windows applications, there's even a NuGet package that will help you get started quickly.

## objective-git

If your application is running on an Apple platform, you're likely using Objective-C as your implementation language. Objective-Git (<https://github.com/libgit2/objective-git>) is the name of the Libgit2 bindings for that environment. The example program looks like this:

```
GTRepository *repo =  
    [[GTRepository alloc] initWithURL:[NSURL fileURLWithPath: @"/path/to/repo"]  
error:NULL];  
NSString *msg = [[[repo headReferenceWithError:NULL] resolvedTarget] message];
```

Objective-git is fully interoperable with Swift, so don't fear if you've left Objective-C behind.

## pygit2

The bindings for Libgit2 in Python are called Pygit2, and can be found at <https://www.pygit2.org>. Our example program:

```
pygit2.Repository("/path/to/repo") # open repository  
    .head                      # get the current branch  
    .peel(pygit2.Commit)        # walk down to the commit  
    .message                   # read the message
```

## Further Reading

Of course, a full treatment of Libgit2's capabilities is outside the scope of this book. If you want more information on Libgit2 itself, there's API documentation at <https://libgit2.github.com/libgit2>, and a set of guides at <https://libgit2.github.com/docs>. For the other bindings, check the bundled README and tests; there are often small tutorials and pointers to further reading there.

## JGit

If you want to use Git from within a Java program, there is a fully featured Git library called JGit. JGit is a relatively full-featured implementation of Git written natively in Java, and is widely used in the Java community. The JGit project is under the Eclipse umbrella, and its home can be found at <https://www.eclipse.org/jgit/>.

## Getting Set Up

There are a number of ways to connect your project with JGit and start writing code against it. Probably the easiest is to use Maven – the integration is accomplished by adding the following

snippet to the `<dependencies>` tag in your pom.xml file:

```
<dependency>
    <groupId>org.eclipse.jgit</groupId>
    <artifactId>org.eclipse.jgit</artifactId>
    <version>3.5.0.201409260305-r</version>
</dependency>
```

The `version` will most likely have advanced by the time you read this; check <https://mvnrepository.com/artifact/org.eclipse.jgit/org.eclipse.jgit> for updated repository information. Once this step is done, Maven will automatically acquire and use the JGit libraries that you'll need.

If you would rather manage the binary dependencies yourself, pre-built JGit binaries are available from <https://www.eclipse.org/jgit/download>. You can build them into your project by running a command like this:

```
javac -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App.java
java -cp .:org.eclipse.jgit-3.5.0.201409260305-r.jar App
```

## Plumbing

JGit has two basic levels of API: plumbing and porcelain. The terminology for these comes from Git itself, and JGit is divided into roughly the same kinds of areas: porcelain APIs are a friendly front-end for common user-level actions (the sorts of things a normal user would use the Git command-line tool for), while the plumbing APIs are for interacting with low-level repository objects directly.

The starting point for most JGit sessions is the `Repository` class, and the first thing you'll want to do is create an instance of it. For a filesystem-based repository (yes, JGit allows for other storage models), this is accomplished using `FileRepositoryBuilder`:

```
// Create a new repository
Repository newlyCreatedRepo = FileRepositoryBuilder.create(
    new File("/tmp/new_repo/.git"));
newlyCreatedRepo.create();

// Open an existing repository
Repository existingRepo = new FileRepositoryBuilder()
    .setGitDir(new File("my_repo/.git"))
    .build();
```

The builder has a fluent API for providing all the things it needs to find a Git repository, whether or not your program knows exactly where it's located. It can use environment variables (`.readEnvironment()`), start from a place in the working directory and search (`.setWorkTree(...).findGitDir()`), or just open a known `.git` directory as above.

Once you have a `Repository` instance, you can do all sorts of things with it. Here's a quick sampling:

```

// 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");

```

There's quite a bit going on here, so let's go through it one section at a time.

The first line gets a pointer to the `master` reference. JGit automatically grabs the *actual* master ref, which lives at `refs/heads/master`, and returns an object that lets you fetch information about the reference. You can get the name (`.getName()`), and either the target object of a direct reference (`.getObjectId()`) or the reference pointed to by a symbolic ref (`.getTarget()`). Ref objects are also used to represent tag refs and objects, so you can ask if the tag is “peeled,” meaning that it points to the final target of a (potentially long) string of tag objects.

The second line gets the target of the `master` reference, which is returned as an `ObjectId` instance. `ObjectId` represents the SHA-1 hash of an object, which might or might not exist in Git's object database. The third line is similar, but shows how JGit handles the rev-parse syntax (for more on this, see [Branch Referenzen](#)); you can pass any object specifier that Git understands, and JGit will return either a valid `ObjectId` for that object, or `null`.

The next two lines show how to load the raw contents of an object. In this example, we call `ObjectLoader.copyTo()` to stream the contents of the object directly to stdout, but `ObjectLoader` also has methods to read the type and size of an object, as well as return it as a byte array. For large objects (where `.isLarge()` returns `true`), you can call `.openStream()` to get an `InputStream`-like object that can read the raw object data without pulling it all into memory at once.

The next few lines show what it takes to create a new branch. We create a `RefUpdate` instance,

configure some parameters, and call `.update()` to trigger the change. Directly following this is the code to delete that same branch. Note that `.setForceUpdate(true)` is required for this to work; otherwise the `.delete()` call will return `REJECTED`, and nothing will happen.

The last example shows how to fetch the `user.name` value from the Git configuration files. This Config instance uses the repository we opened earlier for local configuration, but will automatically detect the global and system configuration files and read values from them as well.

This is only a small sampling of the full plumbing API; there are many more methods and classes available. Also not shown here is the way JGit handles errors, which is through the use of exceptions. JGit APIs sometimes throw standard Java exceptions (such as `IOException`), but there are a host of JGit-specific exception types that are provided as well (such as `NoRemoteRepositoryException`, `CorruptObjectException`, and `NoMergeBaseException`).

## Porcelain

The plumbing APIs are rather complete, but it can be cumbersome to string them together to achieve common goals, like adding a file to the index, or making a new commit. JGit provides a higher-level set of APIs to help out with this, and the entry point to these APIs is the `Git` class:

```
Repository repo;  
// construct repo...  
Git git = new Git(repo);
```

The `Git` class has a nice set of high-level *builder-style* methods that can be used to construct some pretty complex behavior. Let's take a look at an example – doing something like `git ls-remote`:

```
CredentialsProvider cp = new UsernamePasswordCredentialsProvider("username",  
"p4ssw0rd");  
Collection<Ref> remoteRefs = git.lsRemote()  
    .setCredentialsProvider(cp)  
    .setRemote("origin")  
    .setTags(true)  
    .setHeads(false)  
    .call();  
for (Ref ref : remoteRefs) {  
    System.out.println(ref.getName() + " -> " + ref.getObjectId().name());  
}
```

This is a common pattern with the `Git` class; the methods return a command object that lets you chain method calls to set parameters, which are executed when you call `.call()`. In this case, we're asking the `origin` remote for tags, but not heads. Also notice the use of a `CredentialsProvider` object for authentication.

Many other commands are available through the `Git` class, including but not limited to `add`, `blame`, `commit`, `clean`, `push`, `rebase`, `revert`, and `reset`.

## Further Reading

This is only a small sampling of JGit's full capabilities. If you're interested and want to learn more, here's where to look for information and inspiration:

- The official JGit API documentation can be found at <https://www.eclipse.org/jgit/documentation>. These are standard Javadoc, so your favorite JVM IDE will be able to install them locally, as well.
- The JGit Cookbook at <https://github.com/centic9/jgit-cookbook> has many examples of how to do specific tasks with JGit.

## go-git

In case you want to integrate Git into a service written in Golang, there also is a pure Go library implementation. This implementation does not have any native dependencies and thus is not prone to manual memory management errors. It is also transparent for the standard Golang performance analysis tooling like CPU, Memory profilers, race detector, etc.

go-git is focused on extensibility, compatibility and supports most of the plumbing APIs, which is documented at <https://github.com/src-d/go-git/blob/master/COMPATIBILITY.md>.

Here is a basic example of using Go APIs:

```
import "gopkg.in/src-d/go-git.v4"

r, err := git.PlainClone("/tmp/foo", false, &git.CloneOptions{
    URL:      "https://github.com/src-d/go-git",
    Progress: os.Stdout,
})
```

As soon as you have a **Repository** instance, you can access information and perform mutations on it:

```
// 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)
}
```

## Advanced Functionality

go-git has few notable advanced features, one of which is a pluggable storage system, which is similar to Libgit2 backends. The default implementation is in-memory storage, which is very fast.

```
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{
    URL: "https://github.com/src-d/go-git",
})
```

Pluggable storage provides many interesting options. For instance, [https://github.com/src-d/go-git/tree/master/\\_examples/storage](https://github.com/src-d/go-git/tree/master/_examples/storage) allows you to store references, objects, and configuration in an Aerospike database.

Another feature is a flexible filesystem abstraction. Using <https://godoc.org/github.com/src-d/go-billy#Filesystem> it is easy to store all the files in different way i.e by packing all of them to a single archive on disk or by keeping them all in-memory.

Another advanced use-case includes a fine-tunable HTTP client, such as the one found at [https://github.com/src-d/go-git/blob/master/\\_examples/custom\\_http/main.go](https://github.com/src-d/go-git/blob/master/_examples/custom_http/main.go).

```
customClient := &http.Client{
    Transport: &http.Transport{ // accept any certificate (might be useful for
        testing)
        TLSClientConfig: &tls.Config{InsecureSkipVerify: true},
    },
    Timeout: 15 * time.Second, // 15 second timeout
    CheckRedirect: func(req *http.Request, via []*http.Request) error {
        return http.ErrUseLastResponse // don't follow redirect
    },
}

// Override http(s) default protocol to use our custom client
client.InstallProtocol("https", githttp.NewClient(customClient))

// Clone repository using the new client if the protocol is https://
r, err := git.Clone(memory.NewStorage(), nil, &git.CloneOptions{URL: url})
```

## Further Reading

A full treatment of go-git's capabilities is outside the scope of this book. If you want more information on go-git, there's API documentation at <https://godoc.org/gopkg.in/src-d/go-git.v4>, and a set of usage examples at [https://github.com/src-d/go-git/tree/master/\\_examples](https://github.com/src-d/go-git/tree/master/_examples).

## Dulwich

There is also a pure-Python Git implementation - Dulwich. The project is hosted under <https://www.dulwich.io/> It aims to provide an interface to git repositories (both local and remote)

that doesn't call out to git directly but instead uses pure Python. It has an optional C extensions though, that significantly improve the performance.

Dulwich follows git design and separate two basic levels of API: plumbing and porcelain.

Here is an example of using the lower level API to access the commit message of the last commit:

```
from dulwich.repo import Repo
r = Repo('.')
r.head()
# '57fbe010446356833a6ad1600059d80b1e731e15'

c = r[r.head()]
c
# <Commit 015fc1267258458901a94d228e39f0a378370466>

c.message
# 'Add note about encoding.\n'
```

To print a commit log using high-level porcelain API, one can use:

```
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
```

## Further Reading

- The official API documentation is available at <https://www.dulwich.io/apidocs/dulwich.html>
- Official tutorial at <https://www.dulwich.io/docs/tutorial> has many examples of how to do specific tasks with Dulwich

# 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.

## 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> (Also see note below)
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> (Also see note below)
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> (Also see note below)
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> (Also see note below)
Vim	<code>git config --global core.editor "vim"</code>
VS Code	<code>git config --global core.editor "code --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 die `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 man ein ganz neues Repository erstellt wird, mit dem man dann arbeiten kann.

Wir besprechen kurz, wie Sie den Standard-Branch „master“ in [Remote-Branche](#)s 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 [Cloning a Project with Submodules](#) die Option `--recurse-submodules` kennen, die das Klonen eines Repositories 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 [Interactive Staging](#) 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 eingeführt 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 [Starting with Submodules](#) 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 [Signing Commits](#) 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 entnehmen](#). 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 Demystified](#) detailliert auf diesen Befehl ein, der sich ganz der Beschreibung dieses Befehls widmet.

Wir verwenden `git reset --hard`, um einen Merge in [Aborting a Merge](#) 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 [Removing Objects](#) 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 der 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 [Checking Out Conflicts](#), um Dateikonflikte mit `--conflict=diff3` wieder zu integrieren.

Wir gehen auf die Beziehung zu `git reset` in [Reset Demystified](#) 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 von 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 `--abort`-Flags, um ein Merge-Problem abzubrechen.

Wir haben in [Signing Commits](#) 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 Log](#) 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 Log](#) 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 [Searching](#) we look at using the `-S` and `-L` options to do fairly sophisticated searches for something that happened historically in the code such as seeing the history of a function.

In [Signing Commits](#) we see how to use `--show-signature` to add a validation string to each commit in the `git log` output based on if it was validly signed or not.

## git stash

The `git stash` command is used to temporarily store uncommitted work in order to clean out your working directory without having to commit unfinished work on a branch.

This is basically entirely covered in [Stashing and Cleaning](#).

## git tag

The `git tag` command is used to give a permanent bookmark to a specific point in the code history. Generally this is used for things like releases.

This command is introduced and covered in detail in [Tagging](#) and we use it in practice in [Tagging ihres Releases](#).

We also cover how to create a GPG signed tag with the `-s` flag and verify one with the `-v` flag in [Ihre Arbeit signieren](#).

# Sharing and Updating Projects

There are not very many commands in Git that access the network, nearly all of the commands operate on the local database. When you are ready to share your work or pull changes from elsewhere, there are a handful of commands that deal with remote repositories.

## git fetch

The `git fetch` command communicates with a remote repository and fetches down all the information that is in that repository that is not in your current one and stores it in your local database.

We first look at this command in [Fetching und Pulling von Ihren Remotes](#) and we continue to see examples of its use in [Remote-Banches](#).

We also use it in several of the examples in [An einem Projekt mitwirken](#).

We use it to fetch a single specific reference that is outside of the default space in [Pull Request Refs \(Referenzen\)](#) and we see how to fetch from a bundle in [Bundling](#).

We set up highly custom refsspecs in order to make `git fetch` do something a little different than the default in [Die Referenzspezifikation \(engl. Refspec\)](#).

## git pull

The `git pull` command is basically a combination of the `git fetch` and `git merge` commands, where Git will fetch from the remote you specify and then immediately try to merge it into the branch you're on.

We introduce it quickly in [Fetching und Pulling von Ihren Remotes](#) and show how to see what it will merge if you run it in [Inspizieren eines Remotes](#).

We also see how to use it to help with rebasing difficulties in [Rebasen, wenn Sie Rebase durchführen](#).

We show how to use it with a URL to pull in changes in a one-off fashion in [Remote Branches auschecken](#).

Finally, we very quickly mention that you can use the `--verify-signatures` option to it in order to verify that commits you are pulling have been GPG signed in [Signing Commits](#).

## git push

The `git push` command is used to communicate with another repository, calculate what your local database has that the remote one does not, and then pushes the difference into the other repository. It requires write access to the other repository and so normally is authenticated somehow.

We first look at the `git push` command in [Pushing zu Ihren Remotes](#). Here we cover the basics of pushing a branch to a remote repository. In [Pushing/Hochladen](#) we go a little deeper into pushing specific branches and in [Tracking-Branche](#)s we see how to set up tracking branches to automatically push to. In [Remote-Branche Entfernen](#) we use the `--delete` flag to delete a branch on the server with `git push`.

Throughout [An einem Projekt mitwirken](#) we see several examples of using `git push` to share work on branches through multiple remotes.

We see how to use it to share tags that you have made with the `--tags` option in [Tags freigeben](#).

In [Publishing Submodule Changes](#) we use the `--recurse-submodules` option to check that all of our submodules work has been published before pushing the superproject, which can be really helpful when using submodules.

In [Andere Client-Hooks](#) we talk briefly about the `pre-push` hook, which is a script we can setup to run before a push completes to verify that it should be allowed to push.

Finally, in [Pushende Refspecs](#) we look at pushing with a full refspec instead of the general shortcuts that are normally used. This can help you be very specific about what work you wish to share.

## git remote

The `git remote` command is a management tool for your record of remote repositories. It allows you to save long URLs as short handles, such as “origin” so you don’t have to type them out all the time. You can have several of these and the `git remote` command is used to add, change and delete them.

This command is covered in detail in [Mit Remotes arbeiten](#), including listing, adding, removing and renaming them.

It is used in nearly every subsequent chapter in the book too, but always in the standard `git remote add <name> <url>` format.

## git archive

The `git archive` command is used to create an archive file of a specific snapshot of the project.

We use `git archive` to create a tarball of a project for sharing in [Ein Release vorbereiten](#).

## git submodule

The `git submodule` command is used to manage external repositories within a normal repositories. This could be for libraries or other types of shared resources. The `submodule` command has several sub-commands (`add`, `update`, `sync`, etc) for managing these resources.

This command is only mentioned and entirely covered in [Submodules](#).

# Inspection and Comparison

## git show

The `git show` command can show a Git object in a simple and human readable way. Normally you would use this to show the information about a tag or a commit.

We first use it to show annotated tag information in [Annotierte Tags](#).

Later we use it quite a bit in [Revisions-Auswahl](#) to show the commits that our various revision selections resolve to.

One of the more interesting things we do with `git show` is in [Manual File Re-merging](#) to extract specific file contents of various stages during a merge conflict.

## git shortlog

The `git shortlog` command is used to summarize the output of `git log`. It will take many of the same options that the `git log` command will but instead of listing out all of the commits it will present a summary of the commits grouped by author.

We showed how to use it to create a nice changelog in [Das Shortlog](#).

## git describe

The `git describe` command is used to take anything that resolves to a commit and produces a string that is somewhat human-readable and will not change. It's a way to get a description of a commit that is as unambiguous as a commit SHA-1 but more understandable.

We use `git describe` in [Eine Build Nummer generieren](#) and [Ein Release vorbereiten](#) to get a string to name our release file after.

# Debugging

Git has a couple of commands that are used to help debug an issue in your code. This ranges from

figuring out where something was introduced to figuring out who introduced it.

## git bisect

The `git bisect` tool is an incredibly helpful debugging tool used to find which specific commit was the first one to introduce a bug or problem by doing an automatic binary search.

It is fully covered in [Binary Search](#) and is only mentioned in that section.

## git blame

The `git blame` command annotates the lines of any file with which commit was the last one to introduce a change to each line of the file and what person authored that commit. This is helpful in order to find the person to ask for more information about a specific section of your code.

It is covered in [File Annotation](#) and is only mentioned in that section.

## git grep

The `git grep` command can help you find any string or regular expression in any of the files in your source code, even older versions of your project.

It is covered in [Git Grep](#) and is only mentioned in that section.

# Patching

A few commands in Git are centered around the concept of thinking of commits in terms of the changes they introduce, as though the commit series is a series of patches. These commands help you manage your branches in this manner.

## git cherry-pick

The `git cherry-pick` command is used to take the change introduced in a single Git commit and try to re-introduce it as a new commit on the branch you're currently on. This can be useful to only take one or two commits from a branch individually rather than merging in the branch which takes all the changes.

Cherry picking is described and demonstrated in [Rebasing und Cherry-Picking Workflows](#).

## git rebase

The `git rebase` command is basically an automated `cherry-pick`. It determines a series of commits and then cherry-picks them one by one in the same order somewhere else.

Rebasing is covered in detail in [Rebasing](#), including covering the collaborative issues involved with rebasing branches that are already public.

We use it in practice during an example of splitting your history into two separate repositories in [Replace](#), using the `--onto` flag as well.

We go through running into a merge conflict during rebasing in [Rerere](#).

We also use it in an interactive scripting mode with the `-i` option in [Changing Multiple Commit Messages](#).

## git revert

The `git revert` command is essentially a reverse `git cherry-pick`. It creates a new commit that applies the exact opposite of the change introduced in the commit you're targeting, essentially undoing or reverting it.

We use this in [Reverse the commit](#) to undo a merge commit.

# Email

Many Git projects, including Git itself, are entirely maintained over mailing lists. Git has a number of tools built into it that help make this process easier, from generating patches you can easily email to applying those patches from an email box.

## git apply

The `git apply` command applies a patch created with the `git diff` or even GNU diff command. It is similar to what the `patch` command might do with a few small differences.

We demonstrate using it and the circumstances in which you might do so in [Integrieren von Änderungen aus E-Mails](#).

## git am

The `git am` command is used to apply patches from an email inbox, specifically one that is mbox formatted. This is useful for receiving patches over email and applying them to your project easily.

We covered usage and workflow around `git am` in [Änderungen mit am integrieren](#) including using the `--resolved`, `-i` and `-3` options.

There are also a number of hooks you can use to help with the workflow around `git am` and they are all covered in [E-Mail-Workflow-Hooks](#).

We also use it to apply patch formatted GitHub Pull Request changes in [E-Mail Benachrichtigungen](#).

## git format-patch

The `git format-patch` command is used to generate a series of patches in mbox format that you can use to send to a mailing list properly formatted.

We go through an example of contributing to a project using the `git format-patch` tool in [Öffentliche Projekte via Email](#).

## git imap-send

The `git imap-send` command uploads a mailbox generated with `git format-patch` into an IMAP drafts folder.

We go through an example of contributing to a project by sending patches with the `git imap-send` tool in [Öffentliche Projekte via Email](#).

## git send-email

The `git send-email` command is used to send patches that are generated with `git format-patch` over email.

We go through an example of contributing to a project by sending patches with the `git send-email` tool in [Öffentliche Projekte via Email](#).

## git request-pull

The `git request-pull` command is simply used to generate an example message body to email to someone. If you have a branch on a public server and want to let someone know how to integrate those changes without sending the patches over email, you can run this command and send the output to the person you want to pull the changes in.

We demonstrate how to use `git request-pull` to generate a pull message in [Verteiltes, öffentliches Projekt](#).

# External Systems

Git comes with a few commands to integrate with other version control systems.

## git svn

The `git svn` command is used to communicate with the Subversion version control system as a client. This means you can use Git to checkout from and commit to a Subversion server.

This command is covered in depth in [Git und Subversion](#).

## git fast-import

For other version control systems or importing from nearly any format, you can use `git fast-import` to quickly map the other format to something Git can easily record.

This command is covered in depth in [Benutzerdefinierter Import](#).

# Administration

If you're administering a Git repository or need to fix something in a big way, Git provides a number of administrative commands to help you out.

## git gc

The `git gc` command runs “garbage collection” on your repository, removing unnecessary files in your database and packing up the remaining files into a more efficient format.

This command normally runs in the background for you, though you can manually run it if you wish. We go over some examples of this in [Maintenance](#).

## git fsck

The `git fsck` command is used to check the internal database for problems or inconsistencies.

We only quickly use this once in [Data Recovery](#) to search for dangling objects.

## git reflog

The `git reflog` command goes through a log of where all the heads of your branches have been as you work to find commits you may have lost through rewriting histories.

We cover this command mainly in [RefLog Kurzformen](#), where we show normal usage to and how to use `git log -g` to view the same information with `git log` output.

We also go through a practical example of recovering such a lost branch in [Data Recovery](#).

## git filter-branch

The `git filter-branch` command is used to rewrite loads of commits according to certain patterns, like removing a file everywhere or filtering the entire repository down to a single subdirectory for extracting a project.

In [Removing a File from Every Commit](#) we explain the command and explore several different options such as `--commit-filter`, `--subdirectory-filter` and `--tree-filter`.

In [Git-p4](#) and [TFS](#) we use it to fix up imported external repositories.

# Plumbing Commands

There were also quite a number of lower level plumbing commands that we encountered in the book.

The first one we encounter is `ls-remote` in [Pull Request Refs \(Referenzen\)](#) which we use to look at the raw references on the server.

We use `ls-files` in [Manual File Re-merging](#), [Rerere](#) and [The Index](#) to take a more raw look at what your staging area looks like.

We also mention `rev-parse` in [Branch Referenzen](#) to take just about any string and turn it into an object SHA-1.

However, most of the low level plumbing commands we cover are in [Git Interna](#), which is more or

less what the chapter is focused on. We tried to avoid use of them throughout most of the rest of the book.

# Index

- @
  - \$EDITOR, 352
  - \$VISUAL
    - see \$EDITOR, 352
  - .NET, 517
  - .gitignore, 354
    - 0 , 95
    - 1 , 95
- A
  - Apache, 120
  - Apple, 518
  - aliases, 64
  - archiving, 369
  - attributes, 362
  - autocorrect, 354
- B
  - Bazaar, 441
  - Berechtigungs-Caching, 21
  - BitKeeper, 14
  - bash, 507
  - binary files, 363
  - branches, 66
    - basic workflow, 73
    - creating, 68
    - deleting remote, 96
    - diffing, 162
    - long-running, 83
    - managing, 81
    - merging, 77
    - remote, 86, 161
    - switching, 69
    - topic, 84, 157
    - tracking, 94
    - upstream, 94
  - build numbers, 171
- C
  - C, 513
  - C#, 517
  - CRLF, 21
  - CVS, 11
  - Cocoa, 518
  - color, 355
- D
  - commit templates, 352
  - contributing, 133
    - private managed team, 143
    - private small team, 136
    - public large project, 153
    - public small project, 149
  - credentials, 345
  - crlf, 360
- E
  - Dulwich, 523
  - difftool, 356
  - distributed git, 130
- F
  - Eclipse, 506
  - editor
    - changing default, 39
  - email, 155
    - applying patches from, 158
  - excludes, 354, 457
- F
  - files
    - moving, 42
    - removing, 40
  - forking, 132, 178
- G
  - PGP, 353
  - GUIs, 499
  - Git Befehle
    - add, 31, 31
    - clone, 28
    - config, 23, 25
    - help, 25
    - init, 28, 31
    - status, 30
  - Git as a client, 385
  - GitHub, 173
    - API, 221
    - Flow, 179
    - organizations, 212
    - pull requests, 182
    - user accounts, 173
- Page-Footer

---

541

GitHub for Windows, 501  
GitHub for macOS, 501  
GitLab, 123  
GitWeb, 121  
Go, 522  
Graphical tools, 499  
git commands  
    add, 32  
    am, 159  
    apply, 158  
    archive, 171  
    branch, 68, 81  
    checkout, 69  
    cherry-pick, 168  
    clone  
        bare, 112  
    commit, 39, 67  
    config, 39, 64, 155, 351  
    credential, 345  
    daemon, 118  
    describe, 171  
    diff, 36  
        check, 134  
    fast-import, 447  
    fetch, 55  
    fetch-pack, 483  
    filter-branch, 445  
    format-patch, 154  
    gitk, 499  
    gui, 499  
    help, 118  
    http-backend, 120  
    init  
        bare, 113, 116  
    instaweb, 122  
    log, 43  
    merge, 75  
        squash, 153  
    mergetool, 79  
    p4, 417, 444  
    pull, 55  
    push, 55, 61, 92  
    rebase, 97  
    receive-pack, 482  
    remote, 53, 54, 56, 57  
    request-pull, 150  
    rerere, 169  
    send-pack, 482  
    shortlog, 172  
    show, 60  
    show-ref, 388  
    status, 39  
    svn, 385  
    tag, 58, 59, 61  
    upload-pack, 483  
    git-svn, 385  
    git-tf, 425  
    git-tfs, 425  
    gitk, 499  
    go-git, 522

**H**

hooks, 371  
    post-update, 109

**I**

IRC, 26  
Importing  
    from Bazaar, 441  
    from Mercurial, 438  
    from Perforce, 444  
    from Subversion, 435  
    from TFS, 446  
    from others, 447

Interoperation with other VCSs  
    Mercurial, 396  
    Perforce, 408  
    Subversion, 385  
    TFS, 425

ignoring files, 34  
integrating work, 164

**J**

Java, 518  
jgit, 518

**K**

keyword expansion, 366

## L

Linux, 14  
  installing, 19  
Linux Torvalds, 14  
libgit2, 513  
line endings, 360  
log filtering, 49  
log formatting, 45

## M

Mercurial, 396, 438  
Migrating to Git, 435  
Mono, 517  
macOS  
  installing, 20  
maintaining a project, 157  
master, 68  
mergetool, 356  
merging, 77  
  conflicts, 78  
  strategies, 370  
  vs. rebasing, 105

## O

Objective-C, 518  
origin, 87

## P

Perforce, 11, 14, 408, 444  
  Git Fusion, 409  
Powershell, 21  
Python, 518, 523  
pager, 353  
policy example, 374  
posh-git, 511  
powershell, 511  
protocols  
  SSH, 110  
  dumb HTTP, 108  
  git, 111  
  local, 106  
  smart HTTP, 108  
pulling, 95  
pushing, 92

## R

Ruby, 514  
rebasing, 96  
  perils of, 101  
  vs. merging, 105  
references  
  remote, 86  
releasing, 171  
rerere, 169

## S

SHA-1, 16  
SSH keys, 114  
  with GitHub, 174  
Subversion, 11, 14, 131, 385, 435  
serving repositories, 106  
  GitLab, 123  
  GitWeb, 121  
  HTTP, 120  
  SSH, 113  
  git protocol, 118  
shell prompts  
  bash, 507  
  powershell, 511  
  zsh, 509  
staging area  
  skipping, 40

## T

TFS, 425, 446  
TFVC  
  see=TFS, 425  
tab completion  
  bash, 507  
  powershell, 511  
  zsh, 509  
tags, 58, 170  
  annotated, 59  
  lightweight, 60  
  signing, 170

## V

Versionsverwaltung, 10  
  verteilt, 12  
  zentral, 11

Versionswerwaltung

  lokal, [10](#)

Visual Studio, [504](#)

## W

Windows

  installing, [21](#)

whitespace, [359](#)

workflows, [130](#)

  centralized, [130](#)

  dictator and lieutenants, [132](#)

  integration manager, [131](#)

  merging, [164](#)

  merging (large), [166](#)

  rebasing and cherry-picking, [168](#)

## X

Xcode, [20](#)

## Z

zsh, [509](#)