

THE EXPERT'S VOICE® IN SOFTWARE DEVELOPMENT

Pro Git

Everything you need to know about
the Git distributed version control tool

 **git**

Scott Chacon

Foreword by Janie E. Hansen, Git project leader

Apress®

Copyright © 2014

Los geht's

In diesem Kapitel wird es darum gehen, wie man mit Git loslegen kann. Wir werden erläutern, wozu Versionskontrollsysteme gut sind, wie man Git auf verschiedenen Systemen installieren und konfigurieren kann, sodass man in der Lage ist, mit der Arbeit anzufangen. Am Ende dieses Kapitels solltest Du verstehen, wozu Git gut ist, weshalb Du es verwenden solltest und wie Du damit loslegen kannst.

Wozu Versionskontrolle?

Was ist die Versionskontrolle, und warum solltest Du Dich dafür interessieren?

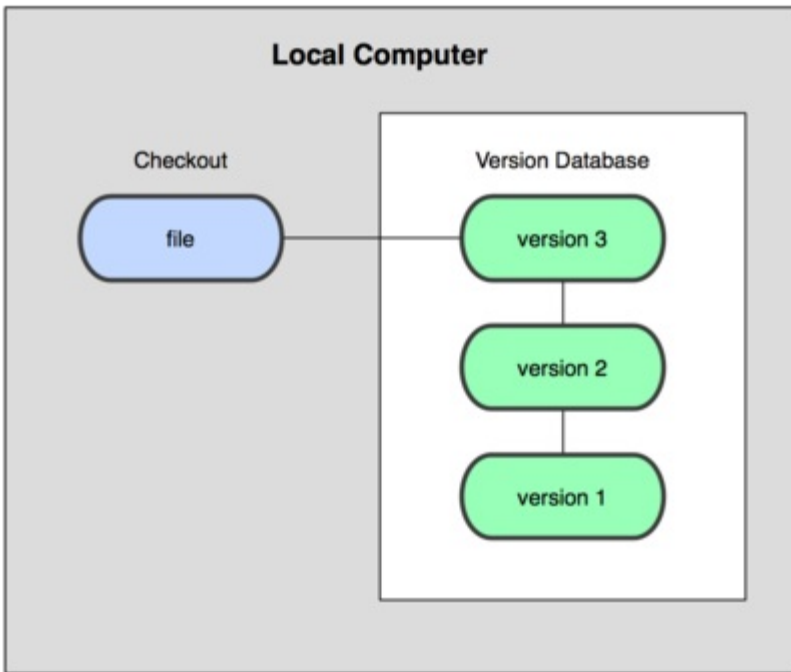
Versionskontrollsysteme (VCS) protokollieren Änderungen an einer Datei oder einer Anzahl von Dateien über die Zeit hinweg, so dass man zu jedem Zeitpunkt auf Versionen und Änderungen zugreifen kann. Die Beispiele in diesem Buch verwenden Software Quellcode, tatsächlich aber kannst Du Änderungen an praktisch jeder Art von Datei per Versionskontrolle nachverfolgen.

Als ein Grafik- oder Webdesigner zum Beispiel willst Du in der Lage sein, jede Version eines Bildes oder Layouts zurückverfolgen zu können. Es wäre daher sehr ratsam, ein Versionskontrollsystem zu verwenden. Ein solches System erlaubt Dir, einzelne Dateien oder auch ein ganzes Projekt in einen früheren Zustand zurückzusetzen, nachzuvollziehen, wer zuletzt welche Änderungen vorgenommen hat, die möglicherweise Probleme verursachen, wer eine Änderung ursprünglich vorgenommen hat usw. Ein Versionskontrollsystem für Deine Arbeit zu verwenden, versetzt Dich in die Lage jederzeit zu einem vorherigen, funktionierenden Zustand zurückzugehen, wenn Du vielleicht Mist gebaut oder aus irgendeinem Grunde Dateien verloren hast. All diese Vorteile erhältst Du für einen nur sehr geringen, zusätzlichen Aufwand.

Lokale Versionskontrollsysteme

Viele Leute kontrollieren Versionen ihrer Arbeit, indem sie einfach Dateien in ein anderes Verzeichnis kopieren (wenn sie clever sind: ein Verzeichnis mit einem Zeitstempel im Namen). Diese Vorgehensweise ist üblich, weil sie so einfach ist. Aber sie ist auch unglaublich fehleranfällig. Man vergisst sehr leicht, in welchem Verzeichnis man sich gerade befindet und kopiert die falschen Dateien oder überschreibt Dateien, die man eigentlich nicht überschreiben wollte.

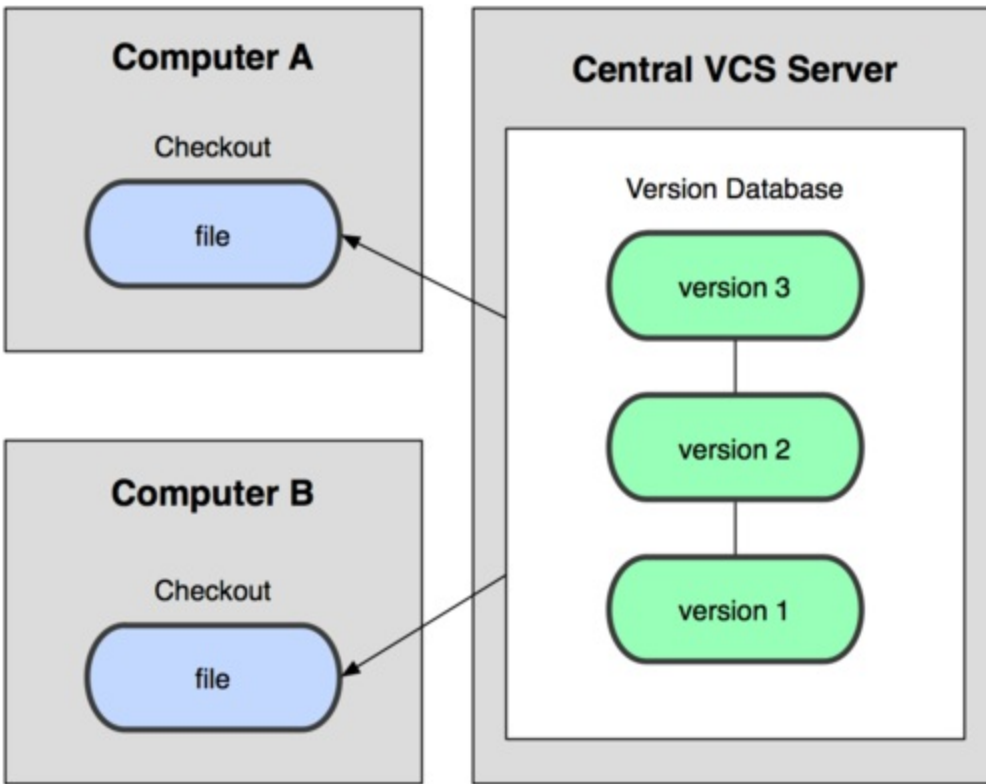
Um diese Arbeit zu erleichtern und sicherer zu machen, haben Programmierer vor langer Zeit Versionskontrollsysteme entwickelt, die alle Änderungen an allen relevanten Dateien in einer lokalen Datenbank verfolgten (siehe Bild 1-1).



Eines der populärsten Versionskontrollsysteme war rcs, und es wird heute immer noch mit vielen Computern ausgeliefert. Z.B. umfasst auch das Betriebssystem Mac OS X den Befehl rcs, wenn Du die Developer Tools installierst. Dieser Befehl arbeitet nach dem Prinzip, dass er für jede Änderung einen Patch (d.h. eine Kodierung der Unterschiede, die eine Änderung an einer oder mehreren Dateien umfasst) in einem speziellen Format in einer Datei auf der Festplatte speichert.

Zentralisierte Versionskontrollsysteme

Das nächste Problem, mit dem Programmierer sich dann konfrontiert sahen, bestand in der Zusammenarbeit mit anderen: Änderungen an dem gleichen Projekt mussten auf verschiedenen Computern, möglicherweise verschiedenen Betriebssystemen vorgenommen werden können. Um dieses Problem zu lösen, wurden Zentralisierte Versionskontrollsysteme (CVCS) entwickelt. Diese Systeme, beispielsweise CVS, Subversion und Perforce, basieren auf einem zentralen Server, der alle versionierten Dateien verwaltet. Wer an diesen Dateien arbeiten will, kann sie von diesem Server abholen („check out“ xxx), auf seinem eigenen Computer bearbeiten und dann wieder auf dem Server abliefern. Diese Art von System war über viele Jahre hinweg der Standard für Versionskontrollsysteme (siehe Bild 1-2).

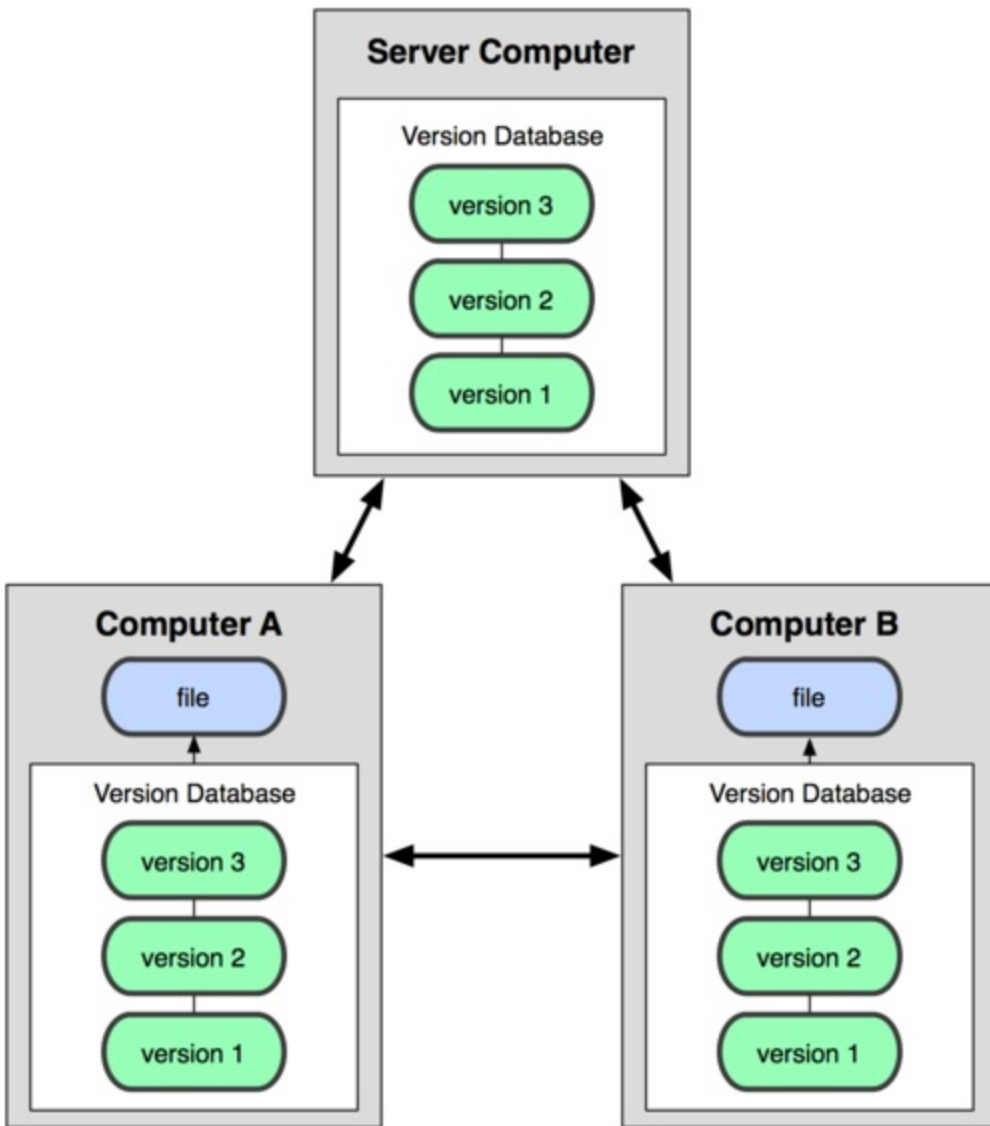


Dieser Aufbau hat viele Vorteile gegenüber Lokalen Versionskontrollsystemen. Zum Beispiel weiß jeder mehr oder weniger genau darüber Bescheid, was andere, an einem Projekt Beteiligte gerade tun. Administratoren haben die Möglichkeit, detailliert festzulegen, wer was tun kann. Und es ist sehr viel einfacher, 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 der „Single Point of Failure“, den der zentralisierte Server darstellt. Wenn dieser Server für nur eine Stunde nicht verfügbar ist, dann kann in dieser Stunde niemand in irgendeiner Form mit anderen arbeiten oder versionierte Änderungen an den Dateien speichern, an denen sie momentan arbeiten. Wenn die auf dem zentralen Server verwendete 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 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 Versionskontrollsysteme

Und an dieser Stelle kommen verteilte Versionskontrollsysteme (DVCS) ins Spiel. In einem DVCS (wie z.B. Git, Mercurial, Bazaar oder Darcs) erhalten Anwender nicht einfach den jeweils letzten Snapshot des Projektes von einem Server: sie erhalten statt dessen eine vollständige Kopie des Repositories. Auf diese Weise kann, wenn ein Server beschädigt wird, jedes beliebige Repository von jedem beliebigen Anwenderrechner zurück kopiert werden und der Server so wieder hergestellt werden.



Darüber hinaus können derartige Systeme hervorragend mit verschiedenen externen („remote“) Repositories umgehen, sodass man mit verschiedenen Gruppen von Leuten simultan in verschiedenen Weisen zusammenarbeiten kann. Das macht es möglich, verschiedene Arten von Arbeitsabläufen (wie Hierarchien) zu integrieren, was mit zentralisierten Systemen nicht möglich ist.

Die Geschichte von Git

Wie viele großartige Dinge im Leben entstand Git aus 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 der Software in Form von Patches (d.h. Änderungen an bestehendem Code) und archivierten Dateien herumgereicht. 2002 began man dann, ein proprietäres DVCS System 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, kaputt. Die zuvor ausgesprochene Erlaubnis, BitKeeper kostenlos zu verwenden, wurde widerrufen. Dies war für die Linux Entwickler Community (und besonders für Linus Torvald, der Erfinder von Linux) der Auslöser dafür, ein eigenes Tool zu entwickeln, das auf den Erfahrungen mit BitKeeper basierte. Ziele des neuen Systems waren unter anderem:

- Geschwindigkeit
- Einfaches Design
- Gute Unterstützung von nicht-linearer Entwicklung (tausende paralleler Branches, d.h. verschiedener Verzweigungen der Versionen)
- Vollständig verteilt
- Fähig, große Projekte wie den Linux Kernel effektiv zu verwalten (Geschwindigkeit und Datenumfang)

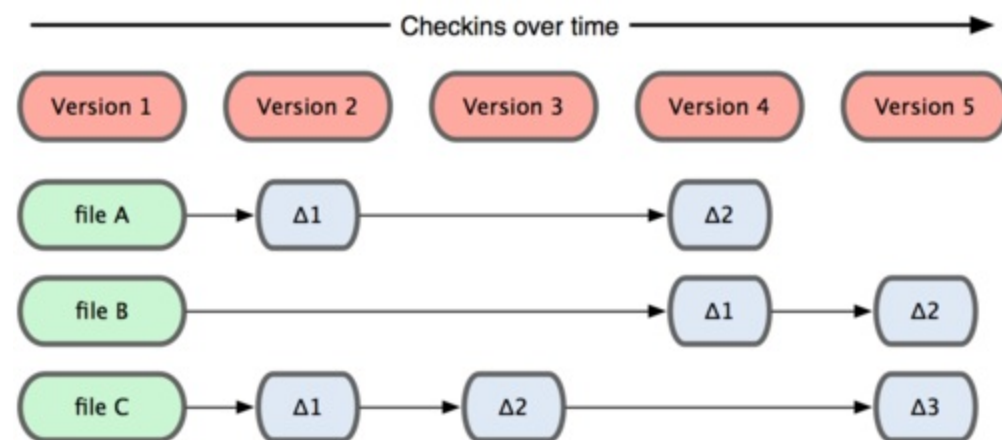
Seit seiner Geburt 2005 entwickelte sich Git kontinuierlich weiter und reifte zu einem System heran, das einfach zu bedienen ist, die ursprünglichen Ziele dabei aber weiter beibehält. Es ist unglaublich schnell, äußerst effizient, wenn es um große Projekte geht, und es hat ein fantastisches Branching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

Git Grundlagen

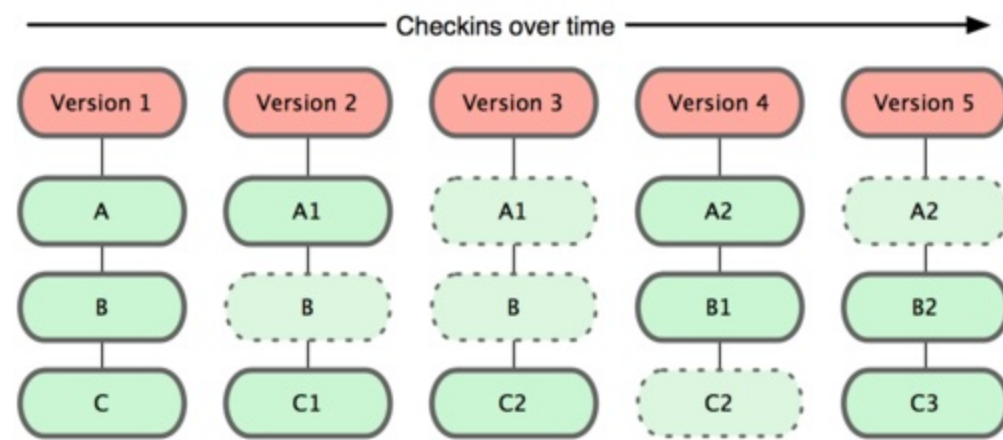
Was also ist Git, in kurzen Worten? Es ist wichtig, den folgenden Abschnitt zu verstehen, in dem es um die grundlegenden Konzepte von Git geht. Das wird Dich in die Lage versetzen, Git einfacher und effektiver anzuwenden. Versuche Dein vorhandenes Wissen über andere Versionskontrollsysteme, wie Subversion oder Perforce, zu ignorieren, während Du Git kennen lernst. Git speichert und konzipiert Information anders als andere Systeme, auch wenn das Interface relativ ähnlich wirkt. Diese Unterschiede zu verstehen wird Dir helfen, Verwirrung bei der Anwendung von Git zu vermeiden.

Snapshots, nicht Diffs

Der Hauptunterschied zwischen Git und anderen Versionskontrollsystemen (auch Subversion und vergleichbaren Systemen) besteht in der Art und Weise wie Git Daten betrachtet. Die meisten anderen Systeme speichern Information als eine fortlaufende Liste von Änderungen an Dateien („Diffs“). Diese Systeme (CVS, Subversion, Perforce, Bazaar usw.) betrachten die Informationen, die sie verwalten, als eine Menge von Dateien und die Änderungen, die über die Zeit hinweg an einzelnen Dateien vorgenommen werden. (Siehe Bild 1-4.)



Git sieht Daten nicht in dieser Weise. Stattdessen betrachtet Git seine Daten eher als eine Reihe von Snapshots eines Mini-Dateisystems. Jedes Mal, wenn Du committest (d.h. den gegenwärtigen Status Deines Projektes als eine Version in Git speicherst), sichert Git den Zustand sämtlicher Dateien in diesem Moment („Snapshot“) und speichert eine Referenz auf diesen Snapshot. 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 Daten also wie in Bild 1-5 dargestellt.



Dies ist ein wichtiger Unterschied zwischen Git und praktisch allen anderen Versionskontrollsystemen. In Git wurden daher fast alle Aspekte der Versionskontrolle neu überdacht, die in anderen Systemen mehr oder weniger von ihren jeweiligen Vorgängergeneration übernommen worden waren. Git arbeitet im Großen und Ganzen eher wie ein (mit einigen unglaublich mächtigen Werkzeugen ausgerüstetes) Mini-Dateisystem, als wie ein gängiges Versionskontrollsystem. Auf einige der Vorteile, die es mit sich bringt, Daten in dieser Weise zu betrachten, werden wir in Kapitel 3 eingehen, wenn wir das Git Branching Konzept diskutieren.

Fast jede Operation ist lokal

Die meisten Operationen in Git benötigen nur die lokalen Dateien und Ressourcen auf Deinem Rechner, um zu funktionieren. D.h. im Allgemeinen werden keine Informationen von einem anderen Rechner im Netzwerk benötigt. Wenn Du mit einem CVCS gearbeitet hast, das für die meisten Operationen einen Network Latency Overhead (also Wartezeiten, die das Netzwerk benötigt werden) hat, dann wirst Du den Eindruck haben, dass die Götter der Geschwindigkeit Git mit unaussprechlichen Fähigkeiten ausgestattet haben. Weil man die vollständige Historie lokal auf dem Rechner hat, werden die allermeisten Operationen ohne jede Verzögerung ausgeführt und sind sehr schnell.

Um beispielsweise die Historie des Projektes zu durchsuchen, braucht Git sie nicht von einem externen Server zu holen – es liest sie einfach aus der lokalen Datenbank. Das heißt, Du siehst die vollständige Projekthistorie ohne jede Verzögerung. Wenn Du sehen willst, worin sich die aktuelle Version einer Datei von einer Version von vor einem Monat unterscheidet, dann kann Git diese Versionen lokal nachschlagen und ihre Unterschiede lokal bestimmen. Es braucht dazu keinen externen Server – weder um Dateien dort nachzuschlagen, noch um Unterschiede dort bestimmen zu lassen.

Dies bedeutet natürlich außerdem, dass es fast nichts gibt, was Du nicht tun kannst, bloß weil Du gerade offline bist oder keinen Zugriff auf ein VPN hast. Wenn Du im Flugzeug oder Zug ein wenig arbeiten willst, kannst Du problemlos Deine Arbeit committen und Deine Arbeit erst auf den Server pushen (hochladen), wenn Du wieder mit dem Internet verbunden bist. Wenn Du zu Hause bist aber nicht auf das VPN zugreifen kannst, kannst Du dennoch arbeiten. Perforce z.B. erlaubt Dir dagegen nicht sonderlich viel zu tun, solange Du nicht mit dem Server verbunden bist. Und in Subversion und CVS kannst Du Dateien zwar ändern, die Änderungen aber nicht in der Datenbank sichern (weil die Datenbank offline ist). Das mag auf den ersten Blick nicht nach

einem großen Problem aussehen, aber Du wirst überrascht sein, was für einen großen Unterschied das ausmachen kann.

Git stellt Integrität sicher

In Git werden Änderungen in Checksummen umgerechnet, bevor sie gespeichert werden. Anschließend werden sie mit dieser Checksumme referenziert. Das macht es unmöglich, dass sich die Inhalte von Dateien oder Verzeichnissen ändern, ohne dass Git das mitbekommt. Git basiert auf dieser Funktionalität und sie ist ein integraler Teil von Gits 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 Checksummen zu erstellen, heißt SHA-1 Hash. Eine solche Checksumme ist eine 40 Zeichen lange Zeichenkette, die aus hexadezimalen Zeichen besteht und diese wird von Git aus den Inhalten einer Datei oder Verzeichnisstruktur kalkuliert. Ein SHA-1 Hash sieht wie folgt aus:

```
24b9da6552252987aa493b52f8696cd6d3b00373
```

Dir werden solche Hash Werte überall in Git begegnen, weil es sie so ausgiebig benutzt. Tatsächlich speichert und referenziert Git Informationen über Dateien in der Datenbank nicht nach ihren Dateinamen sondern nach den Hash Werten ihrer Inhalte.

Git verwaltet fast ausschließlich Daten

Fast alle Operationen, die Du in der täglichen Arbeit mit Git verwendest, fügen Daten jeweils nur zur internen Git Datenbank hinzu. 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. In jedem anderen VCS ist es leicht, Änderungen, die man noch nicht gespeichert hat, zu verlieren oder unbrauchbar zu machen. In Git dagegen ist es schwierig, einen einmal gespeicherten Snapshot zu verlieren, insbesondere wenn man regelmäßig in ein anderes Repository pusht.

U.a. deshalb macht es so viel Spaß, mit Git zu arbeiten. Man kann mit Änderungen experimentieren, ohne befürchten zu müssen, irgendetwas zu zerstören oder durcheinander zu bringen. Einen tieferen Einblick in die Art, wie Git Daten speichert und wie man Daten, die scheinbar verloren sind, wieder herstellen kann, wird Kapitel 9 gewähren.

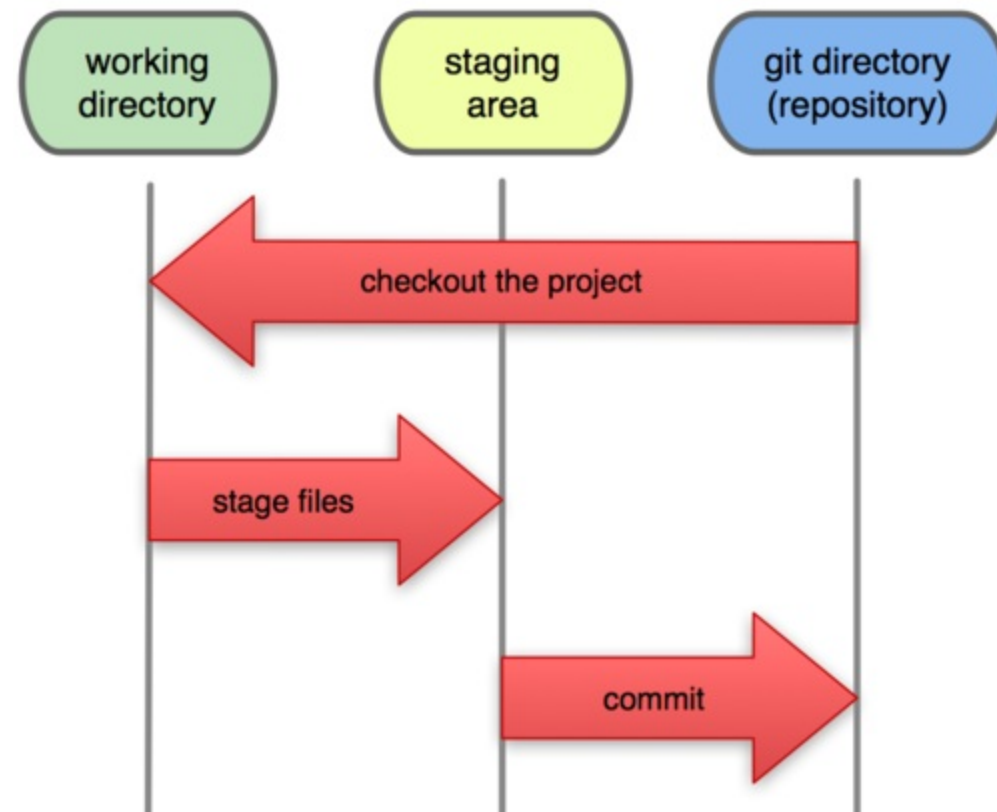
Die drei Zustände

Jetzt aufgepasst. Es folgt die wichtigste Information, die Du Dir merken musst, wenn Du Git kennen lernen willst und Fallstricke vermeiden willst. Git definiert drei Haupt-Zustände, in denen sich eine Datei befinden kann: committed, modified („geändert“) und staged („vorgemerkt“). „Committed“ bedeutet, dass die Daten in der lokalen Datenbank gesichert sind. „Modified“ bedeutet, dass die Datei geändert, diese Änderung aber noch nicht committed wurde. „Staged“

bedeutet, dass Du eine geänderte Datei in ihrem gegenwärtigen Zustand für den nächsten Commit vorgemerkt hast.

Das führt uns zu den drei Hauptbereichen eines Git Projektes: das Git Verzeichnis, das Arbeitsverzeichnis und die Staging Area.

Local Operations



Das Git Verzeichnis ist der Ort, an dem Git Metadaten und die lokale Datenbank für Dein Projekt speichert. Dies ist der wichtigste Teil von Git, und dieser Teil wird kopiert, wenn Du ein Repository von einem anderen Rechner klonst.

Dein Arbeitsverzeichnis ist ein Checkout („Abbild“ xxx) einer spezifischen Version des Projektes. Diese Dateien werden aus der komprimierten Datenbank geholt und auf der Festplatte in einer Form gespeichert, die Du bearbeiten und modifizieren kannst.

Die Staging Area ist einfach eine Datei (normalerweise im Git Verzeichnis), in der vorgemerkt wird, welche Änderungen Dein nächster Commit umfassen soll. Sie wird manchmal auch als „Index“ bezeichnet, aber der Begriff „Staging Area“ ist der gängigere.

Der grundlegend Git Arbeitsprozess sieht in etwa so aus:

1. Du bearbeitest Dateien in Deinem Arbeitsverzeichnis.
2. Du markierst Dateien für den nächsten Commit, indem Du Snapshots zur Staging Area hinzufügst.
3. Du legst den Commit an, wodurch die in der Staging Area vorgemerkten Snapshots dauerhaft im Git Verzeichnis (d.h. der lokalen Datenbank) gespeichert werden.

Wenn eine bestimmte Version einer Datei im Git Verzeichnis liegt, gilt sie als „committed“. Wenn sie geändert und in der Staging Area vorgemerkt ist, gilt sie als „staged“. Und wenn sie geändert, aber noch nicht zur Staging Area hinzugefügt wurde, gilt sie als „modified“. In Kapitel 2 wirst Du mehr über diese Zustände lernen und darüber, wie Du sie sinnvoll einsetzen und wie Du den Zwischenschritt der Staging Area auch einfach überspringen kannst.

Git installieren

Lass uns damit anfangen, Git tatsächlich zu verwenden. Der erste Schritt besteht natürlich darin, Git zu installieren und das kann, wie üblich, auf unterschiedliche Weisen geschehen. Die beiden wichtigsten bestehen darin, entweder den Quellcode herunterzuladen und selbst zu kompilieren oder ein fertiges Paket für Dein Betriebssystem zu installieren.

Vom Quellcode aus installieren

Wenn es Dir möglich ist, empfehlen wir, Git vom Quellcode aus zu installieren, weil Du die jeweils neueste Version erhältst. In der Regel bringt jede Version nützliche Verbesserungen (z.B. am Interface), sodass es sich lohnt die jeweils neueste Version zu verwenden – sofern Du natürlich damit klarkommst, Software aus dem Quellcode zu kompilieren. Viele Linux Distributionen umfassen sehr alte Git Versionen. Wenn Du also keine sehr aktuelle Distribution oder Backports (xxx) verwendest, empfehlen wir, diesen Weg in Erwägung ziehen.

Um Git zu installieren, benötigst Du die folgenden Bibliotheken, die von Git verwendet werden: curl, zlib, openssl, expat und libiconv. Wenn Dir auf Deinem System yum (z.B. auf Fedora) oder apt-get (z.B. auf Debian-basierten Systemen) zur Verfügung steht, kannst Du einen der folgenden Befehle verwenden, um diese Abhängigkeiten zu installieren:

```
$ yum install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel
```

```
$ sudo apt-get install curl-devel expat-devel gettext-devel \
  openssl-devel zlib-devel
```

Nachdem Du die genannten Bibliotheken installiert hast, besorge Dir die aktuelle Version des Git Quellcodes von der Git Webseite:

```
http://git-scm.com/download
```

Danach kannst Du dann Git kompilieren und installieren:

```
$ tar -zxf git-1.6.0.5.tar.gz
$ cd git-1.6.0.5
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Von nun an kannst Du Git mit Hilfe von Git selbst aktualisieren:

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

Installation unter Linux

Wenn Du Git unter Linux mit einem Installationsprogramm installieren willst, kannst Du das normalerweise mit dem Paketmanager tun, der von Deinem Betriebssystem verwendet wird. Unter

Fedora zum Beispiel kannst Du yum verwenden:

```
$ yum install git-core
```

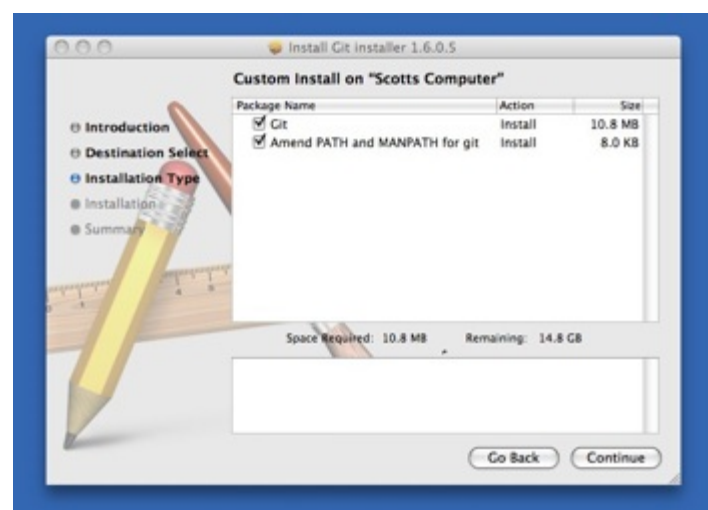
Auf einem Debian-basierten System wie Ubuntu steht Dir apt-get zur Verfügung:

```
$ sudo apt-get install git
```

Installation unter Mac OS X

Auf einem Mac kann man Git auf zwei Arten installieren. Der einfachste ist, das grafische Git Installationsprogramm zu verwenden, den man von der Google Code Webseite herunterladen kann (siehe Bild 1-7)

<http://code.google.com/p/git-osx-installer>



Die andere Möglichkeit ist, Git via MacPorts (<http://www.macports.org>) zu installieren. Wenn Du MacPorts auf Deinem System hast, installiert der folgende Befehl Git:

```
$ sudo port install git-core +svn +doc +bash_completion +gitweb
```

Du brauchst die optionalen Features natürlich nicht mit zu installieren, aber es macht Sinn +svn zu verwenden, falls Du jemals Git mit einem Subversion Repository verwenden willst.

Installation unter Windows

Das msysGit Projekt macht die Installation von Git unter Windows sehr einfach. Lade einfach das Installationsprogramm für Windows von der GitHub Webseite herunter und führe es aus:

<http://msysgit.github.com/>

Danach hast Du sowohl eine Kommandozeilenversion (inklusive eines SSH Clients, der sich später noch als nützlich erweisen wird) als auch die Standard GUI installiert.

Hinweis für Windows Benutzer: Du solltest Git mit der in msysGit enthaltenen Shell (Unix Style)

ausführen. Dies erlaubt es Dir auch die komplexen Kommandozeilenbefehle aus diesem Buch auszuführen. Wenn Du aus irgendeinem Grund die native Windows Shell, also die Eingabeaufforderung, verwenden musst, müssen Gänsefüßchen, statt einzelnen Anführungszeichen verwendet werden (für Parameter, die ein Leerzeichen enthalten). Außerdem müssen alle Parameter, die mit einem Zirkumflex (^) enden und am Ende einer Zeile stehen, mit Gänsefüßchen umschlossen werden. Der Zirkumflex am Ende einer Zeile teilt Windows sonst mit, dass diese Zeile noch nicht beendet ist und in der nächsten Zeile fortgesetzt werden soll.

Git konfigurieren

Nachdem Du jetzt Git auf Deinem System installiert hast, solltest Du Deine Git Konfiguration anpassen. Das brauchst Du nur einmal zu tun, die Konfiguration bleibt auch bestehen, wenn Du Git auf eine neuere Version aktualisierst. Du kannst sie jederzeit ändern, indem Du die folgenden Befehle einfach noch einmal ausführst.

Git umfasst das Werkzeug `git config`, das Dir erlaubt, Konfigurationswerte zu verändern. Auf diese Weise kannst Du anpassen, wie Git aussieht und arbeitet. Diese Werte sind an drei verschiedenen Orten gespeichert:

- Die Datei `/etc/gitconfig` enthält Werte, die für jeden Anwender des Systems und all ihre Projekte gelten. Wenn Du `git config` mit der Option `--system` verwendest, wird diese Datei verwendet.
- Die Werte in der Datei `~/.gitconfig` gelten ausschließlich für Dich und all Deine Projekte. Wenn Du `git config` mit der Option `--global` verwendest, wird diese Datei verwendet.
- Die Datei `.git/config` im Git Verzeichnis eines Projektes enthält Werte, die nur für das jeweilige Projekt gelten. Diese Dateien überschreiben Werte aus den jeweils vorhergehenden Dateien in dieser Reihenfolge. D.h. Werte in beispielsweise `.git/config` überschreiben diejenigen in `/etc/gitconfig`.

Auf Windows Systemen sucht Git nach der `.gitconfig` Datei im `$HOME` Verzeichnis (für die meisten Leute ist das das Verzeichnis `C:\Dokumente und Einstellungen\%USER%`). Git sucht außerdem auch nach dem Verzeichnis `/etc/gitconfig`, aber es sucht relativ demjenigen Verzeichnis, in dem Du Git mit Hilfe des Installers installiert hast.

Deine Identität

Nachdem Du Git installiert hast, solltest Du als erstes Deinen Namen und Deine E-Mail Adresse konfigurieren. Das ist wichtig, weil Git diese Information für jeden Commit verwendet, den Du anlegst, und sie ist unveränderlich in Deine Commits eingebaut (xxx):

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

Du brauchst diese Konfiguration, wie schon erwähnt, nur einmal vorzunehmen, wenn Du die `--global` Option verwendest, weil Git diese Information dann für all Deine Projekte verwenden wird. Wenn Du sie für ein spezielles Projekt mit einem anderen Namen oder einer anderen E-Mail Adresse überschreiben willst, kannst Du dazu den Befehl ohne die `--global` Option innerhalb dieses Projektes ausführen.

Dein Editor

Nachdem Du Deine Identität jetzt konfiguriert hast, kannst Du einstellen, welchen Texteditor Git in Situationen verwenden soll, in denen Du eine Nachricht eingeben musst. Normalerweise

verwendet Git den Standard-Texteditor Deines Systems – das ist üblicherweise Vi oder Vim. Wenn Du einen anderen Texteditor, z.B. Emacs, verwenden willst, kannst Du das wie folgt festlegen:

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

Dein Diff Programm

Eine andere nützliche Einstellung, die Du möglicherweise vornehmen willst, ist welches Diff Programm Git verwendet. Mit diesem Programm kannst Du Konflikte auflösen, die während der Arbeit mit Git manchmal auftreten. Wenn Du beispielsweise vimdiff verwenden willst, kannst Du das so festlegen:

```
$ git config --global merge.tool vimdiff
```

Git kann von Hause aus mit den folgenden Diff Programmen arbeiten: kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, and opendiff. Außerdem kannst Du ein eigenes Programm aufsetzen. Wir werden in Kapitel 7 darauf eingehen, wie das geht.

Deine Einstellungen überprüfen

Wenn Du Deine Einstellungen überprüfen willst, kannst Du mit dem Befehl `git config --list` alle Einstellungen anzuzeigen, die Git an dieser Stelle (z.B. innerhalb eines bestimmten Projektes) bekannt sind:

```
$ git config --list
user.name=Scott Chacon
user.email=schacon@gmail.com
color.status=auto
color.branch=auto
color.interactive=auto
color.diff=auto
...
```

Manche Variablen werden möglicherweise mehrfach aufgelistet, weil Git dieselbe Variable in verschiedenen Dateien (z.B. `/etc/gitconfig` und `~/.gitconfig`) findet. In diesem Fall verwendet Git dann den jeweils zuletzt aufgelisteten Wert.

Außerdem kannst Du mit dem Befehl `git config {key}` prüfen, welchen Wert Git für einen bestimmten Variablennamen verwendet:

```
$ git config user.name
Scott Chacon
```

Hilfe finden

Falls Du jemals Hilfe in der Anwendung von Git benötigst, gibt es drei Möglichkeiten, die entsprechende Seite aus der Dokumentation (manpage) für jeden Git Befehl anzuzeigen:

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

Beispielsweise erhältst Du die Hilfeseite für den `git config` Befehl so:

```
$ git help config
```

Die „manpage“ Dokumentation ist nützlich, weil Du sie Dir jederzeit anzeigen lassen kannst, auch wenn Du offline bist. Wenn Dir die manpages und dieses Buch nicht ausreichen, kannst Du Deine Fragen auch in den Chaträumen `#git` oder `#github` auf dem Freenode IRC Server (irc.freenode.net) stellen. Diese Räume sind in der Regel sehr gut besucht. Normalerweise findet sich unter den hunderten von Anwendern, die oft sehr viel Erfahrung mit Git haben, irgendjemand, der Deine Fragen gern beantwortet.

Zusammenfassung

Du solltest jetzt ein grundlegendes Verständnis davon haben, was Git ist und wie es sich von anderen CVCS unterscheidet, die Du möglicherweise schon verwendet hast. Du solltest außerdem eine funktionierende Git Version auf Deinem Rechner installiert und konfiguriert haben. Jetzt wird es Zeit, einige Git Grundlagen zu besprechen.

Git Grundlagen

Wenn Du nur ein einziges Kapitel aus diesem Buch lesen willst, um mit Git loslegen zu können, dann lies dieses hier. Wir werden hier auf die grundlegenden Git Befehle eingehen, die Du für den größten Teil Deiner täglichen Arbeit mit Git brauchst. Am Ende des Kapitels solltest Du in der Lage sein, ein neues Repository anzulegen und zu konfigurieren, Dateien zur Versionskontrolle hinzuzufügen und wieder aus ihr zu entfernen, Änderungen in der Staging Area für einen Commit vorzumerken und schließlich einen Commit durchzuführen. Wir werden außerdem besprechen, wie Du Git so konfigurieren kannst, dass es bestimmte Dateien und Dateimuster ignoriert, wie Du Fehler schnell und einfach rückgängig machen, wie Du die Historie Deines Projektes durchsuchen und Änderungen zwischen bestimmten Commits nachschlagen, und wie Du in externe Repositories herauf- und von dort herunterladen kannst.

Ein Git Repository anlegen

Es gibt grundsätzlich zwei Möglichkeiten, ein Git Repository auf dem eigenen Rechner anzulegen. Erstens kann man ein existierendes Projekt oder Verzeichnis in ein neues Git Repository importieren. Zweitens kann man ein existierendes Repository von einem anderen Rechner, der als Server fungiert, auf den eigenen Rechner klonen.

Ein existierendes Verzeichnis als Git Repository initialisieren

Wenn Du künftige Änderungen an einem bestehenden Projekt auf Deinem Rechner mit Git versionieren und nachverfolgen willst, kannst Du dazu einfach in das jeweilige Verzeichnis wechseln und diesen Befehl ausführen:

```
$ git init
```

Das 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 9 werden wir genauer darauf eingehen, welche Dateien im `.git` Verzeichnis enthalten sind und was ihre Aufgabe ist.)

Wenn in Deinem Projekt bereits Dateien vorhanden sind (und es sich nicht nur um ein leeres Verzeichnis handelt), willst Du diese vermutlich zur Versionskontrolle hinzufügen, damit Änderungen daran künftig nachverfolgbar sind. Dazu kannst Du die folgenden Git Befehle ausführen um die Dateien zur Versionskontrolle hinzuzufügen. Anschließend kannst Du Deinen ersten Commit anlegen:

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

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

Ein existierendes Repository klonen

Wenn Du eine Kopie eines existierenden Git Repositorys anlegen willst – z.B. um an einem Projekt mitzuarbeiten – dann kannst Du dazu den Befehl `git clone` verwenden. Wenn Du schon mit anderen VCS Systemen wie Subversion gearbeitet hast, wird Dir auffallen, dass der Befehl `clone` heißt und nicht `checkout`. Dies ist ein wichtiger Unterschied, den Du verstehen solltest. Git lädt eine Kopie aller Daten, die sich im existierenden Repository befinden, auf Deinen Rechner. Mit `git clone` wird jede einzelne Version jeder einzelnen Datei in der Historie des Repositorys heruntergeladen. Wenn ein Repository auf einem Server einmal beschädigt wird (z.B. weil die Festplatte beschädigt wird), kann man tatsächlich jeden beliebigen Klon des Repositorys verwenden, um das Repository auf dem Server wieder in dem Zustand wieder herzustellen, in dem es sich befand, als es geklont wurde. (Es kann passieren, dass man einige auf dem Server vorhandenen Hooks verliert, aber alle versionierten Daten bleiben erhalten. In Kapitel 4 gehen wir

darauf noch einmal genauer ein.)

Du kannst ein Repository mit dem Befehl `git clone [url]` klonen. Um beispielsweise das Repository der Ruby Git Bibliothek Grit zu klonen, führst Du den folgenden Befehl aus:

```
$ git clone git://github.com/schacon/grit.git
```

Git legt dann ein Verzeichnis `grit` an, initialisiert ein `.git` Verzeichnis darin, lädt alle Daten des Repositorys herunter, und checkt eine Arbeitskopie der letzten Version aus. Wenn Du in das neue `grit` Verzeichnis wechselst, findest Du dort die in diesem Projekt enthaltenen Dateien und kannst sie benutzen oder bearbeiten. Wenn Du das Repository in ein Verzeichnis mit einem anderen Namen als `grit` klonen willst, kannst Du das wie folgt angeben:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Dieser Befehl tut das gleiche wie der vorhergehende, aber das Zielverzeichnis ist diesmal `mygrit`.

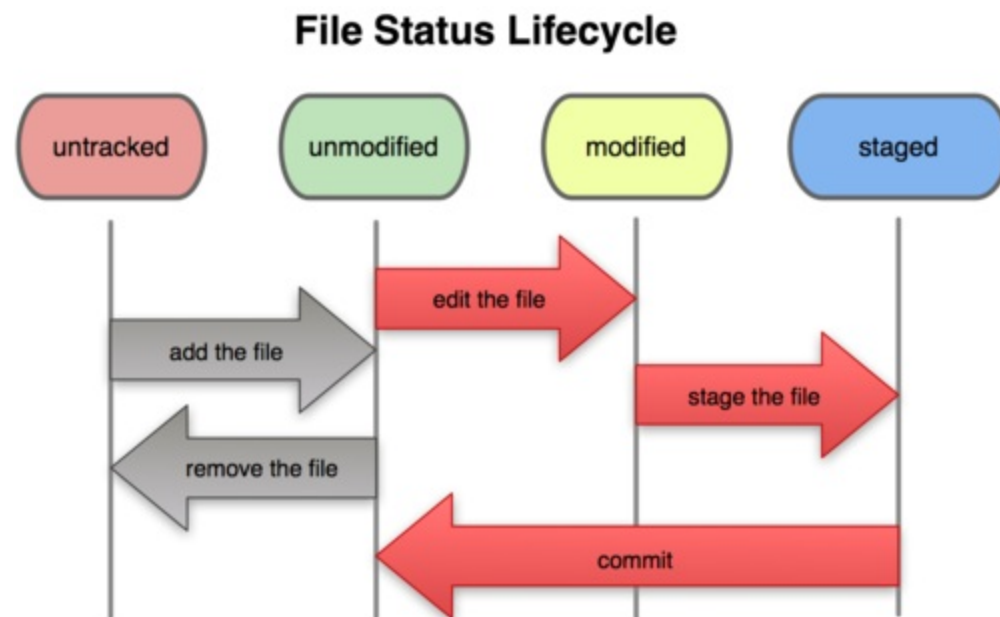
Git unterstützt eine Reihe unterschiedlicher Übertragungsprotokolle. Das vorhergehende Beispiel verwendet das `git://` Protokoll, aber Du wirst auch auf `http(s)://` oder `user@server:/path.git` treffen, die das SSH Protokoll verwenden. In Kapitel 4 gehen wir auf die verfügbaren Optionen (und deren Vor- und Nachteile) ein, die ein Server hat, um Zugriff auf ein Git Repository zu ermöglichen.

Änderungen am Repository nachverfolgen

Du hast jetzt ein voll funktionsfähiges Git Repository und eine Arbeitskopie des Projekts ist in Deinem Verzeichnis ausgecheckt. Du kannst nun die Dateien im Projekt bearbeiten. Immer wenn Dein Projekt einen Zustand erreicht hat, den Du festhalten willst, musst Du diese Änderungen einchecken.

Jede Datei in Deinem Arbeitsverzeichnis kann sich in einem von zwei Zuständen befinden: Änderungen werden verfolgt (engl. tracked) oder nicht (engl. untracked). Alle Dateien, die sich im letzten Snapshot (Commit) befanden, werden in der Versionskontrolle verfolgt. Sie können entweder unverändert (engl. unmodified), modifiziert (engl. modified) oder für den nächsten Commit vorgemerkt (engl. staged) sein. Alle anderen Dateien in Deinem Arbeitsverzeichnis dagegen sind nicht versioniert: das sind all diejenigen Dateien, die nicht schon im letzten Snapshot enthalten waren und die sich nicht in der Staging Area befinden. Wenn Du ein Repository gerade geklont hast, sind alle Dateien versioniert und unverändert – Du hast sie gerade ausgecheckt aber noch nicht verändert.

Sobald Du versionierte Dateien bearbeitest, wird Git sie als modifiziert erkennen, weil Du sie seit dem letzten Commit geändert hast. Du merkst diese geänderten Dateien für den nächsten Commit vor (d.h. Du fügst sie zur Staging Area hinzu bzw. Du stagest sie), legst aus allen markierten Änderungen einen Commit an und der Vorgang beginnt von vorn. Bild 2-1 stellt diesen Zyklus dar:



Den Zustand Deiner Dateien prüfen

Das wichtigste Hilfsmittel, um den Zustand zu überprüfen, in dem sich die Dateien in Deinem Repository gerade befinden, ist der Befehl `git status`. Wenn Du diesen Befehl unmittelbar nach dem Klonen eines Repositories ausführst, sollte er folgende Ausgabe liefern:

```
$ git status
```

```
# On branch 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 Versionskontrolle stehen und seit dem letzten Commit geändert wurden – andernfalls würden sie hier aufgelistet werden. Außerdem teilt Dir der Befehl mit, in welchem Branch Du Dich gerade befindest. In diesem Beispiel ist dies der Branch `master`. Mach Dir darüber im Moment keine Gedanken, wir werden im nächsten Kapitel auf Branches detailliert eingehen.

Sagen wir Du fügst eine neue `README` Datei zu Deinem Projekt hinzu. Wenn die Datei zuvor nicht existiert hat und Du jetzt `git status` ausführst, zeigt Git die bisher nicht versionierte Datei wie folgt an:

```
$ vim README
$ git status
# On branch 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 in der Sektion „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 Snapshot (Commit) von Git nicht enthalten ist. Git nimmt eine solche Datei nicht automatisch in die Versionskontrolle auf, sondern man muss Git dazu ausdrücklich auffordern. Ansonsten würden generierte Binärdateien oder andere Dateien, die Du nicht in Deinem Repository haben willst, 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 Versionskontrolle hinzu.

Neue Dateien zur Versionskontrolle hinzufügen

Um eine neue Datei zur Versionskontrolle hinzuzufügen, verwendest Du den Befehl `git add`. Für Deine neue `README` Datei kannst Du ihn wie folgt ausführen:

```
$ git add README
```

Wenn Du den `git status` Befehl erneut ausführst, siehst Du, dass sich Deine `README` Datei jetzt unter Versionskontrolle befindet und für den nächsten Commit vorgemerkt ist (gestaged ist):

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


Dass die Datei für den nächsten Commit vorgemerkt ist, siehst Du daran, dass sie in der Sektion „Changes to be committed“ aufgelistet ist. Wenn Du jetzt einen Commit anlegst, wird der Snapshot den Zustand der Datei beinhalten, den sie zum Zeitpunkt des Befehls `git add` hatte. Du erinnerst Dich daran, dass Du, als Du vorhin `git init` ausgeführt hast, anschließend `git add` ausgeführt hast: an dieser Stelle hast Du die Dateien in Deinem Verzeichnis der Versionskontrolle hinzugefügt. Der `git add` Befehl akzeptiert einen Pfadnamen einer Datei oder eines Verzeichnisses. Wenn Du ein Verzeichnis angibst, fügt `git add` alle Dateien in diesem Verzeichnis und allen Unterverzeichnissen rekursiv hinzu.

Geänderte Dateien stagen

Wenn Du eine bereits versionierte Datei `benchmarks.rb` änderst und den `git status` Befehl ausführst, erhältst Du folgendes:

```
$ git status
# On branch 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)
#
#   modified:   benchmarks.rb
#
```

Die Datei `benchmarks.rb` erscheint in der Sektion „Changes not staged for commit“ – d.h., dass eine versionierte Datei im Arbeitsverzeichnis verändert worden ist, aber noch nicht für den Commit vorgemerkt wurde. Um sie vorzumerken, führst Du den Befehl `git add` aus. (`git add` wird zu verschiedenen Zwecken eingesetzt. Man verwendet ihn, um neue Dateien zur Versionskontrolle hinzuzufügen, Dateien für einen Commit zu markieren und verschiedene andere Dinge – beispielsweise, einen Konflikt aus einem Merge als aufgelöst zu kennzeichnen.)

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

Beide Dateien sind nun für den nächsten Commit vorgemerkt. Nehmen wir an, Du willst jetzt aber noch eine weitere Änderung an der Datei `benchmarks.rb` vornehmen, bevor Du den Commit tatsächlich anlegst. Du öffnest die Datei und änderst sie. Jetzt könntest Du den Commit anlegen. Aber zuvor führen wir noch mal `git status` aus:

```
$ vim benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#   modified:   benchmarks.rb
#
```

Huch, was ist das? Jetzt wird `benchmarks.rb` sowohl in der Staging Area als auch als geändert aufgelistet. Die Erklärung dafür ist, dass Git eine Datei in exakt dem Zustand für den Commit vormerkt, in dem sie sich befindet, wenn Du den Befehl `git add` ausführst. Wenn Du den Commit jetzt anlegst, wird die Version der Datei `benchmarks.rb` diejenigen Inhalte haben, die sie hatte, als Du `git add` zuletzt ausgeführt hast – nicht diejenigen, die sie in dem Moment hat, wenn Du den Commit anlegst. Wenn Du stattdessen die gegenwärtige Version im Commit haben willst, kannst Du einfach erneut `git add` ausführen:

```
$ git add benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   new file:   README
#   modified:   benchmarks.rb
#
```

Dateien ignorieren

Du wirst in der Regel eine Reihe von Dateien in Deinem Projektverzeichnis haben, die Du nicht versionieren bzw. im Repository haben willst, wie z.B. automatisch generierte Dateien, wie Logdateien oder Dateien, die Dein Build-System erzeugt. In solchen Fällen kannst Du in der Datei `.gitignore` alle Dateien oder Dateimuster angeben, die Du ignorieren willst.

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

Die erste Zeile weist Git an, alle Dateien zu ignorieren, die mit einem `.o` oder `.a` enden (also Objekt- und Archiv-Dateien, die von Deinem Build-System erzeugt werden). Die zweite Zeile bewirkt, dass alle Dateien ignoriert werden, die mit einer Tilde (`~`) enden. Viele Texteditoren speichern ihre temporären Dateien auf diese Weise, wie beispielsweise Emacs. Du kannst außerdem Verzeichnisse wie `log`, `tmp` oder `pid` hinzufügen, automatisch erzeugte Dokumentation, und so weiter. Es ist normalerweise empfehlenswert, eine `.gitignore` Datei anzulegen, bevor

man mit der eigentlichen Arbeit anfängt, damit man nicht versehentlich Dateien ins Repository hinzufügt, die man dort nicht wirklich haben will.

Folgende Regeln gelten in einer `.gitignore` Datei:

- Leere Zeilen oder Zeilen, die mit `#` beginnen, werden ignoriert.
- Standard glob Muster funktionieren.
- Du kannst ein Muster mit einem Schrägstrich (`/`) abschließen, um ein Verzeichnis zu deklarieren.
- Du kannst ein Muster negieren, indem Du ein Ausrufezeichen (`!`) voranstellst.

Glob Muster sind vereinfachte reguläre Ausdrücke, die von der Shell verwendet werden. Ein Stern (`*`) bezeichnet „kein oder mehrere Zeichen“; `[abc]` bezeichnet eines der in den eckigen Klammern angegebenen Zeichen (in diesem Fall also `a`, `b` oder `c`); ein Fragezeichen (`?`) bezeichnet ein beliebiges, einzelnes Zeichen; und eckige Klammern mit Zeichen, die von einem Bindestrich getrennt werden (`[0-9]`) bezeichnen ein Zeichen aus der jeweiligen Menge von Zeichen (in diesem Fall also aus der Menge der Zeichen von 0 bis 9).

Hier ist ein weiteres Beispiel für eine `.gitignore` Datei:

```
# ein Kommentar - dieser wird ignoriert
# ignoriert alle Dateien, die mit .a enden
*.a
# nicht aber lib.a Dateien (obwohl obige Zeile *.a ignoriert)
!lib.a
# ignoriert eine TODO Datei nur im Wurzelverzeichnis, nicht aber
/TODO
# ignoriert alle Dateien im build/ Verzeichnis
build/
# ignoriert doc/notes.txt, aber nicht doc/server/arch.txt
doc/*.txt
# ignoriert alle .txt Dateien unterhalb des doc/ Verzeichnis
doc/**/*.txt
```

Die Kombination `**/` wurde in der Git Version 1.8.2 eingeführt.

Die Änderungen in der Staging Area durchsehen

Wenn Dir die Ausgabe des Befehl `git status` nicht aussagekräftig genug ist, weil Du exakt wissen willst, was sich geändert hat – und nicht lediglich, welche Dateien geändert wurden – kannst Du den `git diff` Befehl verwenden. Wir werden `git diff` später noch einmal im Detail besprechen, aber Du wirst diesen Befehl in der Regel verwenden wollen, um eine der folgenden, zwei Fragen zu beantworten: Was hast Du geändert, aber noch nicht für einen Commit vorgemerkt? Und welche Änderungen hast Du für einen Commit bereits vorgemerkt? Während `git status` diese Fragen nur mit Dateinamen beantwortet, zeigt Dir `git diff` exakt an, welche Zeilen hinzugefügt, geändert und entfernt wurden. Dies entspricht gewissermaßen einem Patch.

Nehmen wir an, Du hast die Datei `README` geändert und für einen Commit in der Staging Area

vorgemerkt. Dann änderst Du außerdem die Datei `benchmarks.rb`, fügst sie aber noch nicht zur Staging Area hinzu. Wenn Du den `git status` Befehl dann ausführst, zeigt er Dir in etwa Folgendes an:

```
$ git status
# On branch 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)
#
#   modified:   benchmarks.rb
#
```

Um festzustellen, welche Änderungen Du bisher nicht gestaged hast, führe `git diff` ohne irgendwelche weiteren Argumente aus:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..da65585 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
    @commit.parents[0].parents[0].parents[0]
  end

+  run_code(x, 'commits 1') do
+    git.commits.size
+  end
+
  run_code(x, 'commits 2') do
    log = git.commits('master', 15)
    log.size
```

Dieser Befehl vergleicht die Inhalte Deines Arbeitsverzeichnisses mit den Inhalten Deiner Staging Area. Das Ergebnis zeigt Dir die Änderungen, die Du an Dateien im Arbeitsverzeichnis vorgenommen, aber noch nicht für den nächsten Commit vorgemerkt hast.

Wenn Du sehen willst, welche Änderungen in der Staging Area und somit für den nächsten Commit vorgesehen sind, kannst Du `git diff --cached` verwenden. (Ab der Version Git 1.6.1 kannst Du außerdem `git diff --staged` verwenden, was vielleicht leichter zu merken ist.) Dieser Befehl vergleicht die Inhalte der Staging Area mit dem letzten Commit:

```
$ git diff --cached
diff --git a/README b/README
new file mode 100644
index 0000000..03902a1
--- /dev/null
+++ b/README2
```

```
@@ -0,0 +1,5 @@
+grit
+ by Tom Preston-Werner, Chris Wanstrath
+ http://github.com/mojombo/grit
+
+Grit is a Ruby library for extracting information from a Git repository
```

Es ist wichtig, im Kopf zu behalten, dass `git diff` nicht alle Änderungen seit dem letzten Commit anzeigt – er zeigt lediglich diejenigen Änderungen an, die noch nicht in der Staging Area sind. Das kann verwirrend sein. Wenn Du all Deine Änderungen bereits für einen Commit vorgemerkt hast, zeigt `git diff` überhaupt nichts an.

Ein anderes Beispiel: Wenn Du Änderungen an der Datei `benchmarks.rb` bereits zur Staging Area hinzugefügt hast und sie dann anschließend noch mal änderst, kannst Du `git diff` verwenden, um diese letzten Änderungen anzuzeigen, die noch nicht in der Staging Area sind:

```
$ git add benchmarks.rb
$ echo '# test line' >> benchmarks.rb
$ git status
# On branch master
#
# Changes to be committed:
#
#   modified:   benchmarks.rb
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
```

Jetzt kannst Du `git diff` verwenden, um zu sehen, was noch nicht für den nächsten Commit vorgemerkt ist:

```
$ git diff
diff --git a/benchmarks.rb b/benchmarks.rb
index e445e28..86b2f7c 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -127,3 +127,4 @@ end
   main()

  ##pp Grit::GitRuby.cache_client.stats
+# test line
```

und `git diff --cached`, um zu sehen, was für den nächsten Commit vorgesehen ist:

```
$ git diff --cached
diff --git a/benchmarks.rb b/benchmarks.rb
index 3cb747f..e445e28 100644
--- a/benchmarks.rb
+++ b/benchmarks.rb
@@ -36,6 +36,10 @@ def main
```

```

    @commit.parents[0].parents[0].parents[0]
end

+   run_code(x, 'commits 1') do
+       git.commits.size
+   end
+
+   run_code(x, 'commits 2') do
+       log = git.commits('master', 15)
+       log.size
+   end

```

Einen Commit erzeugen

Nachdem Du jetzt alle Änderungen, die Du im nächsten Commit haben willst, in Deiner Staging Area gesammelt hast, kannst Du den Commit anlegen. Denke daran, dass Änderungen, die nicht in der Staging Area sind (also alle Änderungen, die Du vorgenommen hast, seit Du zuletzt `git add` ausgeführt hast), auch nicht in den Commit aufgenommen werden. Sie werden ganz einfach weiterhin als geänderte Dateien im Arbeitsverzeichnis verbleiben. In unserem Beispiel haben wir gesehen, dass alle Änderungen vorgemerkt waren, als wir zuletzt `git status` ausgeführt haben, also können wir den Commit jetzt anlegen. Das geht am einfachsten mit dem Befehl:

```
$ git commit
```

Wenn Du diesen Befehl ausführst, wird Git den Texteditor Deiner Wahl starten. (D.h. denjenigen Texteditor, der durch die `$EDITOR` Variable Deiner Shell angegeben wird – normalerweise ist das `vim` oder `emacs`, aber Du kannst jeden Editor Deiner Wahl angeben. Wie in Kapitel 1 besprochen, kannst Du dazu `git config --global core.editor` verwenden.)

Der Editor zeigt in etwa folgenden Text an (dies ist ein Beispiel mit `vim`):

```

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   README
#       modified:  benchmarks.rb
~
~
~
".git/COMMIT_EDITMSG" 10L, 283C

```

Du siehst, dass die vorausgefüllte Commit Meldung die Ausgabe des letzten `git status` Befehls als einen Kommentar und darüber eine leere Zeile enthält. Du kannst die Kommentare entfernen und Deine eigene Meldung einfügen. Oder Du kannst sie stehen lassen, damit Du siehst, was im Commit enthalten sein wird. (Um die Änderungen noch detaillierter sehen zu können, kannst Du den Befehl `git commit` mit der Option `-v` verwenden. Das fügt zusätzlich das Diff Deiner Änderungen im Editor ein, sodass Du exakt sehen kannst, was sich im Commit befindet.) Wenn

Du den Texteditor beendest, erzeugt Git den Commit mit der gegebenen Meldung (d.h., ohne den Kommentar und das Diff).

Alternativ kannst Du die Commit Meldung direkt mit dem Befehl `git commit` angeben, indem Du die Option `-m` wie folgt verwendest:

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[master]: created 463dc4f: "Fix benchmarks for speed"
 2 files changed, 3 insertions(+), 0 deletions(-)
 create mode 100644 README
```

Du hast jetzt Deinen ersten Commit angelegt! Git zeigt Dir als Rückmeldung einige Details über den neu angelegten Commit an: in welchem Branch er sich befindet (master), welche SHA-1 Checksumme er hat (463dc4f, in diesem Fall nur die Kurzform), wie viele Dateien geändert wurden und eine Zusammenfassung über die insgesamt neu hinzugefügten und entfernten Zeilen in diesem Commit.

Denke daran, dass jeder neue Commit denjenigen Snapshot aufzeichnet, den Du in der Staging Area vorbereitet hast. Änderungen, die nicht in der Staging Area waren, werden weiterhin als modifizierte Dateien im Arbeitsverzeichnis vorliegen. Jedes Mal wenn Du einen Commit anlegst, zeichnest Du einen Snapshot Deines Projektes auf, zu dem Du zurückkehren oder mit dem Du spätere Änderungen vergleichen kannst.

Die Staging Area überspringen

Obwohl die Staging Area unglaublich nützlich ist, um genau diejenigen Commits anzulegen, die Du in Deiner Projekt Historie haben willst, ist sie manchmal auch ein bisschen umständlich. Git stellt Dir deshalb eine Alternative zur Verfügung, mit der Du die Staging Area überspringen kannst. Wenn Du den Befehl `git commit` mit der Option `-a` ausführst, übernimmt Git automatisch alle Änderungen an diejenigen Dateien, die sich bereits unter Versionskontrolle befinden, in den Commit – sodass Du auf diese Weise den Schritt `git add` weglassen kannst:

```
$ git status
# On branch master
#
# Changes not staged for commit:
#
#   modified:   benchmarks.rb
#
$ git commit -a -m 'added new benchmarks'
[master 83e38c7] added new benchmarks
 1 files changed, 5 insertions(+), 0 deletions(-)
```

Beachte, dass Du in diesem Fall `git add` zuvor noch nicht ausgeführt hast, die Änderungen an `benchmarks.rb` aber dennoch in den Commit übernommen werden.

Dateien entfernen

Um eine Datei aus der Git Versionskontrolle zu entfernen, muss diese von den verfolgten Dateien (genauer, aus der Staging Area) entfernt werden und dann mit einem Commit bestätigt werden. Der Befehl `git rm` tut genau das – und löscht die Datei außerdem aus dem Arbeitsverzeichnis, sodass sie dort nicht unbeabsichtigt (als eine nun unversionierte Datei) liegen bleibt.

Wenn Du einfach nur eine Datei aus dem Arbeitsverzeichnis löschst, wird sie in der Sektion „Changes not staged for commit“ angezeigt, wenn Du `git status` ausführst:

```
$ rm grit.gemspec
$ git status
# On branch master
#
# Changes not staged for commit:
#   (use "git add/rm <file>..." to update what will be committed)
#
#       deleted:    grit.gemspec
#
```

Wenn Du jetzt `git rm` ausführst, wird diese Änderung für den nächsten Commit in der Staging Area vorgemerkt:

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

Nach dem nächsten Anlegen eines Commits, wird die Datei nicht mehr im Arbeitsverzeichnis liegen und sich nicht länger unter Versionskontrolle befinden. Wenn Du die Datei zuvor geändert und diese Änderung bereits zur Staging Area hinzugefügt hattest, musst Du die Option `-f` verwenden, um zu erzwingen, dass sie gelöscht wird. Dies ist eine Sicherheitsmaßnahme, um zu vermeiden, dass Du versehentlich Daten löschst, die sich bisher noch nicht als Commit Snapshot in der Historie Deines Projektes befinden – und deshalb auch nicht wiederhergestellt werden können.

Ein anderer Anwendungsfall für `git rm` ist, dass Du eine Datei in Deinem Arbeitsverzeichnis behalten, aber aus der Staging Area nehmen willst. In anderen Worten, Du willst die Datei nicht löschen, sondern aus der Versionskontrolle nehmen. Das könnte zum Beispiel der Fall sein, wenn Du vergessen hattest, eine Datei in `.gitignore` anzugeben und sie versehentlich zur Versionskontrolle hinzugefügt hast, beispielsweise eine große Logdatei oder eine Reihe kompilierter `.a` Dateien. Hierzu kannst Du dann die `--cached` Option verwenden:

```
$ git rm --cached readme.txt
```


Der `git rm` Befehl akzeptiert Dateien, Verzeichnisse und `glob` Dateimuster. D.h., Du kannst z.B. folgendes tun:

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

Beachte den Backslash (`\`) vor dem Stern (`*`). Er ist nötig, weil Git Dateinamen zusätzlich zur Dateinamen-Expansion Deiner Shell selbst vervollständigt. D.h., dieser Befehl entfernt alle Dateien, die die Erweiterung `.log` haben und sich im `/log` Verzeichnis befinden. Ein anderes Beispiel ist:

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

Dieser Befehl entfernt alle Dateien, die mit einer Tilde (`~`) aufhören.

Dateien verschieben

Anders als andere VCS Systeme verfolgt Git nicht explizit, ob Dateien verschoben werden. Wenn Du eine Datei umbenennst, werden darüber keine Metadaten in der Historie gespeichert. Stattdessen ist Git schlau genug, solche Dinge im Nachhinein zu erkennen. Wir werden uns damit später noch befassen.

Es ist allerdings ein bisschen verwirrend, dass Git trotzdem einen `git mv` Befehl kennt. Wenn Du eine Datei umbenennen willst, kannst Du folgendes tun:

```
$ git mv file_from file_to
```

Das funktioniert einwandfrei. Wenn Du diesen Befehl ausführst und danach den `git status` ausführst, zeigt Git an, dass die Datei umbenannt wurde:

```
$ git mv README.txt README
$ git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       renamed:    README.txt -> README
#
```

Allerdings kannst Du genauso folgendes tun:

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

Git ist clever genug, selbst herauszufinden, dass Du die Datei umbenannt hast. Du brauchst dies also nicht explizit mit dem `git mv` Befehl zu tun. Der einzige Unterschied ist, dass Du mit `git mv` nur einen Befehl, nicht drei, ausführen musst – das ist natürlich etwas bequemer. Darüberhinaus

kannst Du aber Dateien auf jede beliebige Art und Weise extern umbenennen und dann später `git add` bzw. `git rm` verwenden, wenn Du einen Commit zusammenstellst.

Die Commit Historie anzeigen

Nachdem Du einige Commits angelegt oder ein bestehendes Repository geklont hast, willst Du vielleicht wissen, welche Änderungen zuletzt vorgenommen wurden. Der grundlegende und mächtige Befehl, mit dem Du das tun kannst, ist `git log`.

Die folgende Beispiele beziehen sich auf ein sehr simples Repository mit dem Namen „simplegit“, das ich oft für Demonstationszwecke verwende:

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

Wenn Du in diesem Projekt `git log` ausführst, solltest Du eine Ausgabe wie die folgende sehen:

```
$ 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 code

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

first commit

Der Befehl `git log` listet die Historie der Commits eines Projekts in umgekehrter chronologischer Reihenfolge auf, wenn man ihn ohne weitere Argumente ausführt, d.h. die letzten Commits stehen oben. Wie Du sehen kannst wird jeder Commit mit seiner SHA-1 Checksumme, Namen und E-Mail Adresse des Autors, dem Datum und der Commit Meldung aufgelistet.

Für den Befehl `git log` gibt es eine riesige Anzahl von Optionen, mit denen man sehr genau eingrenzen kann, wonach man in einer Historie sucht. Schauen wir uns also einige der am häufigsten verwendeten Optionen an.

Eine sehr nützliche Option ist `-p`. Sie zeigt die Änderungen an, die in einem Commit enthalten sind. Du kannst außerdem `-2` angeben, wodurch nur die letzten beiden Einträge angezeigt werden:

```
$ 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
@@ -5,5 +5,5 @@ require 'rake/gempackagetask'
  spec = Gem::Specification.new do |s|
    s.name       = "simplegit"
-   s.version    = "0.1.0"
+   s.version    = "0.1.1"
    s.author     = "Scott Chacon"
    s.email      = "schacon@gee-mail.com"

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

removed unnecessary test code

```
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
\ No newline at end of file
```

Diese Option zeigt also im Prinzip die gleiche Information wie zuvor, aber zusätzlich zu jedem Eintrag ein Diff. Das ist nützlich, um einen Code Review zu machen oder eben mal eine Reihe von Commits durchzuschauen, die ein Mitarbeiter angelegt hat.

Manchmal ist es einfacher Änderungen an Hand der Wörter anstatt zeilenbasiert zu überprüfen. Git bietet dafür die Option `--word-diff`, welche man an den Befehl `git log -p` anhängen kann. Man weist Git damit an, einen Vergleich auf Basis der Wörter anstatt Zeile für Zeile durchzuführen. Dieser Vergleich ist ziemlich nutzlos wenn man Änderungen innerhalb von Quellcode vergleicht. Beim Vergleich von langen Textdateien zeigt er aber seine Stärke. Er bietet sich zum Beispiel für Bücher oder wissenschaftliche Texte an. Hierzu ein Beispiel:

```
$ git log -U1 --word-diff
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
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
  s.name       = "simplegit"
  s.version    = ["0.1.0"]{+"0.1.1"+}
  s.author     = "Scott Chacon"
```

Wie man in der Ausgabe sehen kann, zeigt dieser Vergleich nicht an, welche Zeilen hinzugekommen und welche entfallen sind. Stattdessen werden Änderungen innerhalb der Zeilen dargestellt. Mit der Sequenz {+ +} wird ein neu hinzugekommenes Wort gekennzeichnet, mit [- -] ein Wort, welches entfernt wurde. Normalerweise zeigt Git bei einem Vergleich drei zusätzliche Zeilen ober- und unterhalb der eigentlichen Änderung an. Bei einem Textvergleich reicht meist eine zusätzliche Zeile. Man kann dies mit der Option -u1 erreichen, so wie in dem oben gezeigten Beispiel.

Außerdem gibt es verschiedene Optionen, die nützlich sind, um Dinge zusammenzufassen. Beispielsweise kannst Du eine kurze Statistik über jeden Commit mit der Option --stat anzeigen lassen:

```
$ 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 files changed, 1 insertions(+), 1 deletions(-)

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

    removed unnecessary test code

lib/simplegit.rb |      5 -----
1 files changed, 0 insertions(+), 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(+), 0 deletions(-)
```

Die --stat Option zeigt unterhalb jedes Commits eine kurze Statistik über die jeweiligen

Änderungen an: welche Dateien geändert wurden und wieviele Zeilen insgesamt hinzugefügt oder entfernt wurden. Eine weitere nützliche Option ist `--pretty`. Diese Option ändert das Format der Ausgabe und es gibt eine Anzahl mitgelieferter Formate. Das `oneline` Format listet jeden Commit in einer einzigen Zeile, was nützlich ist, wenn Du eine große Anzahl von Commits durchsuchen willst. Die `short`, `full` und `fuller` Formate zeigen die Commits in ähnlicher Form an, aber mit jeweils mehr oder weniger Informationen.

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

Eines der interessantesten Formate ist `format`, das Dir erlaubt, Dein eigenes Format zu verwenden. Dies ist insbesondere nützlich, wenn Du die Ausgabe in ein anderes Programm einlesen willst (da Du das Format explizit angibst, kannst Du sicher sein, dass es sich nicht ändert, wenn Du Git auf eine neuere Version aktualisierst):

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

Tabelle 2-1 zeigt einige nützliche Optionen, die von `format` akzeptiert werden:

Option	Beschreibung
<code>%H</code>	Commit Hash
<code>%h</code>	Abgekürzter Commit Hash
<code>%T</code>	Baum Hash
<code>%t</code>	Abgekürzter Baum Hash
<code>%P</code>	Eltern Hashs
<code>%p</code>	Abgekürzte Eltern Hashs
<code>%an</code>	Autor Name
<code>%ae</code>	Autor E-Mail
<code>%ad</code>	Autor Datum (format akzeptiert eine <code>--date=</code> Option)
<code>%ar</code>	Autor Datum, relativ
<code>%cn</code>	Committer Name
<code>%ce</code>	Committer E-Mail
<code>%cd</code>	Committer Datum
<code>%cr</code>	Committer Datum, relativ
<code>%s</code>	Betreff

Du fragst Dich vielleicht, was der Unterschied zwischen Autor und Committer ist. Der Autor ist diejenige Person, die eine Änderung ursprünglich vorgenommen hat. Der Committer dagegen ist diejenige Person, die den Commit angelegt hat. D.h., wenn Du einen Patch an ein Projekt Team schickst und eines der Team Mitglieder den Patch akzeptiert und verwendet, wird beiden Anerkennung gezollt – sowohl Dir als Autor als auch dem Teammitglied als Comitter. Wir werden auf diese Unterschiedung in Kapitel 5 noch einmal genauer eingehen.

Die `oneline` und `format` Optionen können außerdem zusammen mit einer weiteren Option `--graph` verwendet werden. Diese Option fügt einen netten kleinen ASCII Graphen hinzu, der die

Branch- und Merge-Historie des Projektes anzeigt. Das kannst Du z.B. in Deinem Klon des Grit Projekt Repositorys sehen:

```
$ 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
```

Das sind nur einige eher simple Format Optionen für die Ausgabe von `git log` – es gibt sehr viel mehr davon. Tabelle 2-2 listet diejenigen Optionen auf, die wir bisher besprochen haben, und einige weitere, die besonders nützlich sind:

Option	Beschreibung
<code>-p</code>	Zeigt den Patch, der einem Commit entspricht.
<code>--word-diff</code>	Führt den Vergleich Wort für Wort, anstatt Zeile für Zeile aus.
<code>--stat</code>	Zeigt Statistiken über die in einem Commit geänderten Dateien und eingefügt/entfernte Zeilen.
<code>--shortstat</code>	Zeigt nur die Kurzstatistik über eingefügte/entfernte Zeilen aus der Commit Information.
<code>--name-only</code>	Zeigt die Liste der geänderte Dateien nach der Commit Information.
<code>--name-status</code>	Zeigt die Liste der Dateien mit der hinzugefügt/geändert/entfernt.
<code>--abbrev-commit</code>	Zeigt nur die ersten Zeichen einer SHA-1 Checksumme, nicht alle 40 Zeichen.
<code>--relative-date</code>	Zeigt das Datum in relativem Format (z.B. „2 weeks ago“), nicht als vollständiges Datum.
<code>--graph</code>	Zeigt einen ASCII Graphen der Branch- und Merge-Historie neben der Ausgabe.
<code>--pretty</code>	Zeigt Commits in einem alternativen Format. Gültige Optionen sind: <code>oneline</code> , <code>full</code> , <code>fuller</code> , <code>fuller2</code> , <code>fuller3</code> , <code>fuller4</code> , <code>fuller5</code> , <code>fuller6</code> , <code>fuller7</code> , <code>fuller8</code> , <code>fuller9</code> , <code>fuller10</code> , <code>fuller11</code> , <code>fuller12</code> , <code>fuller13</code> , <code>fuller14</code> , <code>fuller15</code> , <code>fuller16</code> , <code>fuller17</code> , <code>fuller18</code> , <code>fuller19</code> , <code>fuller20</code> , <code>fuller21</code> , <code>fuller22</code> , <code>fuller23</code> , <code>fuller24</code> , <code>fuller25</code> , <code>fuller26</code> , <code>fuller27</code> , <code>fuller28</code> , <code>fuller29</code> , <code>fuller30</code> , <code>fuller31</code> , <code>fuller32</code> , <code>fuller33</code> , <code>fuller34</code> , <code>fuller35</code> , <code>fuller36</code> , <code>fuller37</code> , <code>fuller38</code> , <code>fuller39</code> , <code>fuller40</code> , <code>fuller41</code> , <code>fuller42</code> , <code>fuller43</code> , <code>fuller44</code> , <code>fuller45</code> , <code>fuller46</code> , <code>fuller47</code> , <code>fuller48</code> , <code>fuller49</code> , <code>fuller50</code> , <code>fuller51</code> , <code>fuller52</code> , <code>fuller53</code> , <code>fuller54</code> , <code>fuller55</code> , <code>fuller56</code> , <code>fuller57</code> , <code>fuller58</code> , <code>fuller59</code> , <code>fuller60</code> , <code>fuller61</code> , <code>fuller62</code> , <code>fuller63</code> , <code>fuller64</code> , <code>fuller65</code> , <code>fuller66</code> , <code>fuller67</code> , <code>fuller68</code> , <code>fuller69</code> , <code>fuller70</code> , <code>fuller71</code> , <code>fuller72</code> , <code>fuller73</code> , <code>fuller74</code> , <code>fuller75</code> , <code>fuller76</code> , <code>fuller77</code> , <code>fuller78</code> , <code>fuller79</code> , <code>fuller80</code> , <code>fuller81</code> , <code>fuller82</code> , <code>fuller83</code> , <code>fuller84</code> , <code>fuller85</code> , <code>fuller86</code> , <code>fuller87</code> , <code>fuller88</code> , <code>fuller89</code> , <code>fuller90</code> , <code>fuller91</code> , <code>fuller92</code> , <code>fuller93</code> , <code>fuller94</code> , <code>fuller95</code> , <code>fuller96</code> , <code>fuller97</code> , <code>fuller98</code> , <code>fuller99</code> , <code>fuller100</code> , <code>fuller101</code> , <code>fuller102</code> , <code>fuller103</code> , <code>fuller104</code> , <code>fuller105</code> , <code>fuller106</code> , <code>fuller107</code> , <code>fuller108</code> , <code>fuller109</code> , <code>fuller110</code> , <code>fuller111</code> , <code>fuller112</code> , <code>fuller113</code> , <code>fuller114</code> , <code>fuller115</code> , <code>fuller116</code> , <code>fuller117</code> , <code>fuller118</code> , <code>fuller119</code> , <code>fuller120</code> , <code>fuller121</code> , <code>fuller122</code> , <code>fuller123</code> , <code>fuller124</code> , <code>fuller125</code> , <code>fuller126</code> , <code>fuller127</code> , <code>fuller128</code> , <code>fuller129</code> , <code>fuller130</code> , <code>fuller131</code> , <code>fuller132</code> , <code>fuller133</code> , <code>fuller134</code> , <code>fuller135</code> , <code>fuller136</code> , <code>fuller137</code> , <code>fuller138</code> , <code>fuller139</code> , <code>fuller140</code> , <code>fuller141</code> , <code>fuller142</code> , <code>fuller143</code> , <code>fuller144</code> , <code>fuller145</code> , <code>fuller146</code> , <code>fuller147</code> , <code>fuller148</code> , <code>fuller149</code> , <code>fuller150</code> , <code>fuller151</code> , <code>fuller152</code> , <code>fuller153</code> , <code>fuller154</code> , <code>fuller155</code> , <code>fuller156</code> , <code>fuller157</code> , <code>fuller158</code> , <code>fuller159</code> , <code>fuller160</code> , <code>fuller161</code> , <code>fuller162</code> , <code>fuller163</code> , <code>fuller164</code> , <code>fuller165</code> , <code>fuller166</code> , <code>fuller167</code> , <code>fuller168</code> , <code>fuller169</code> , <code>fuller170</code> , <code>fuller171</code> , <code>fuller172</code> , <code>fuller173</code> , <code>fuller174</code> , <code>fuller175</code> , <code>fuller176</code> , <code>fuller177</code> , <code>fuller178</code> , <code>fuller179</code> , <code>fuller180</code> , <code>fuller181</code> , <code>fuller182</code> , <code>fuller183</code> , <code>fuller184</code> , <code>fuller185</code> , <code>fuller186</code> , <code>fuller187</code> , <code>fuller188</code> , <code>fuller189</code> , <code>fuller190</code> , <code>fuller191</code> , <code>fuller192</code> , <code>fuller193</code> , <code>fuller194</code> , <code>fuller195</code> , <code>fuller196</code> , <code>fuller197</code> , <code>fuller198</code> , <code>fuller199</code> , <code>fuller200</code> , <code>fuller201</code> , <code>fuller202</code> , <code>fuller203</code> , <code>fuller204</code> , <code>fuller205</code> , <code>fuller206</code> , <code>fuller207</code> , <code>fuller208</code> , <code>fuller209</code> , <code>fuller210</code> , <code>fuller211</code> , <code>fuller212</code> , <code>fuller213</code> , <code>fuller214</code> , <code>fuller215</code> , <code>fuller216</code> , <code>fuller217</code> , <code>fuller218</code> , <code>fuller219</code> , <code>fuller220</code> , <code>fuller221</code> , <code>fuller222</code> , <code>fuller223</code> , <code>fuller224</code> , <code>fuller225</code> , <code>fuller226</code> , <code>fuller227</code> , <code>fuller228</code> , <code>fuller229</code> , <code>fuller230</code> , <code>fuller231</code> , <code>fuller232</code> , <code>fuller233</code> , <code>fuller234</code> , <code>fuller235</code> , <code>fuller236</code> , <code>fuller237</code> , <code>fuller238</code> , <code>fuller239</code> , <code>fuller240</code> , <code>fuller241</code> , <code>fuller242</code> , <code>fuller243</code> , <code>fuller244</code> , <code>fuller245</code> , <code>fuller246</code> , <code>fuller247</code> , <code>fuller248</code> , <code>fuller249</code> , <code>fuller250</code> , <code>fuller251</code> , <code>fuller252</code> , <code>fuller253</code> , <code>fuller254</code> , <code>fuller255</code> , <code>fuller256</code> , <code>fuller257</code> , <code>fuller258</code> , <code>fuller259</code> , <code>fuller260</code> , <code>fuller261</code> , <code>fuller262</code> , <code>fuller263</code> , <code>fuller264</code> , <code>fuller265</code> , <code>fuller266</code> , <code>fuller267</code> , <code>fuller268</code> , <code>fuller269</code> , <code>fuller270</code> , <code>fuller271</code> , <code>fuller272</code> , <code>fuller273</code> , <code>fuller274</code> , <code>fuller275</code> , <code>fuller276</code> , <code>fuller277</code> , <code>fuller278</code> , <code>fuller279</code> , <code>fuller280</code> , <code>fuller281</code> , <code>fuller282</code> , <code>fuller283</code> , <code>fuller284</code> , <code>fuller285</code> , <code>fuller286</code> , <code>fuller287</code> , <code>fuller288</code> , <code>fuller289</code> , <code>fuller290</code> , <code>fuller291</code> , <code>fuller292</code> , <code>fuller293</code> , <code>fuller294</code> , <code>fuller295</code> , <code>fuller296</code> , <code>fuller297</code> , <code>fuller298</code> , <code>fuller299</code> , <code>fuller300</code> , <code>fuller301</code> , <code>fuller302</code> , <code>fuller303</code> , <code>fuller304</code> , <code>fuller305</code> , <code>fuller306</code> , <code>fuller307</code> , <code>fuller308</code> , <code>fuller309</code> , <code>fuller310</code> , <code>fuller311</code> , <code>fuller312</code> , <code>fuller313</code> , <code>fuller314</code> , <code>fuller315</code> , <code>fuller316</code> , <code>fuller317</code> , <code>fuller318</code> , <code>fuller319</code> , <code>fuller320</code> , <code>fuller321</code> , <code>fuller322</code> , <code>fuller323</code> , <code>fuller324</code> , <code>fuller325</code> , <code>fuller326</code> , <code>fuller327</code> , <code>fuller328</code> , <code>fuller329</code> , <code>fuller330</code> , <code>fuller331</code> , <code>fuller332</code> , <code>fuller333</code> , <code>fuller334</code> , <code>fuller335</code> , <code>fuller336</code> , <code>fuller337</code> , <code>fuller338</code> , <code>fuller339</code> , <code>fuller340</code> , <code>fuller341</code> , <code>fuller342</code> , <code>fuller343</code> , <code>fuller344</code> , <code>fuller345</code> , <code>fuller346</code> , <code>fuller347</code> , <code>fuller348</code> , <code>fuller349</code> , <code>fuller350</code> , <code>fuller351</code> , <code>fuller352</code> , <code>fuller353</code> , <code>fuller354</code> , <code>fuller355</code> , <code>fuller356</code> , <code>fuller357</code> , <code>fuller358</code> , <code>fuller359</code> , <code>fuller360</code> , <code>fuller361</code> , <code>fuller362</code> , <code>fuller363</code> , <code>fuller364</code> , <code>fuller365</code> , <code>fuller366</code> , <code>fuller367</code> , <code>fuller368</code> , <code>fuller369</code> , <code>fuller370</code> , <code>fuller371</code> , <code>fuller372</code> , <code>fuller373</code> , <code>fuller374</code> , <code>fuller375</code> , <code>fuller376</code> , <code>fuller377</code> , <code>fuller378</code> , <code>fuller379</code> , <code>fuller380</code> , <code>fuller381</code> , <code>fuller382</code> , <code>fuller383</code> , <code>fuller384</code> , <code>fuller385</code> , <code>fuller386</code> , <code>fuller387</code> , <code>fuller388</code> , <code>fuller389</code> , <code>fuller390</code> , <code>fuller391</code> , <code>fuller392</code> , <code>fuller393</code> , <code>fuller394</code> , <code>fuller395</code> , <code>fuller396</code> , <code>fuller397</code> , <code>fuller398</code> , <code>fuller399</code> , <code>fuller400</code> , <code>fuller401</code> , <code>fuller402</code> , <code>fuller403</code> , <code>fuller404</code> , <code>fuller405</code> , <code>fuller406</code> , <code>fuller407</code> , <code>fuller408</code> , <code>fuller409</code> , <code>fuller410</code> , <code>fuller411</code> , <code>fuller412</code> , <code>fuller413</code> , <code>fuller414</code> , <code>fuller415</code> , <code>fuller416</code> , <code>fuller417</code> , <code>fuller418</code> , <code>fuller419</code> , <code>fuller420</code> , <code>fuller421</code> , <code>fuller422</code> , <code>fuller423</code> , <code>fuller424</code> , <code>fuller425</code> , <code>fuller426</code> , <code>fuller427</code> , <code>fuller428</code> , <code>fuller429</code> , <code>fuller430</code> , <code>fuller431</code> , <code>fuller432</code> , <code>fuller433</code> , <code>fuller434</code> , <code>fuller435</code> , <code>fuller436</code> , <code>fuller437</code> , <code>fuller438</code> , <code>fuller439</code> , <code>fuller440</code> , <code>fuller441</code> , <code>fuller442</code> , <code>fuller443</code> , <code>fuller444</code> , <code>fuller445</code> , <code>fuller446</code> , <code>fuller447</code> , <code>fuller448</code> , <code>fuller449</code> , <code>fuller450</code> , <code>fuller451</code> , <code>fuller452</code> , <code>fuller453</code> , <code>fuller454</code> , <code>fuller455</code> , <code>fuller456</code> , <code>fuller457</code> , <code>fuller458</code> , <code>fuller459</code> , <code>fuller460</code> , <code>fuller461</code> , <code>fuller462</code> , <code>fuller463</code> , <code>fuller464</code> , <code>fuller465</code> , <code>fuller466</code> , <code>fuller467</code> , <code>fuller468</code> , <code>fuller469</code> , <code>fuller470</code> , <code>fuller471</code> , <code>fuller472</code> , <code>fuller473</code> , <code>fuller474</code> , <code>fuller475</code> , <code>fuller476</code> , <code>fuller477</code> , <code>fuller478</code> , <code>fuller479</code> , <code>fuller480</code> , <code>fuller481</code> , <code>fuller482</code> , <code>fuller483</code> , <code>fuller484</code> , <code>fuller485</code> , <code>fuller486</code> , <code>fuller487</code> , <code>fuller488</code> , <code>fuller489</code> , <code>fuller490</code> , <code>fuller491</code> , <code>fuller492</code> , <code>fuller493</code> , <code>fuller494</code> , <code>fuller495</code> , <code>fuller496</code> , <code>fuller497</code> , <code>fuller498</code> , <code>fuller499</code> , <code>fuller500</code> , <code>fuller501</code> , <code>fuller502</code> , <code>fuller503</code> , <code>fuller504</code> , <code>fuller505</code> , <code>fuller506</code> , <code>fuller507</code> , <code>fuller508</code> , <code>fuller509</code> , <code>fuller510</code> , <code>fuller511</code> , <code>fuller512</code> , <code>fuller513</code> , <code>fuller514</code> , <code>fuller515</code> , <code>fuller516</code> , <code>fuller517</code> , <code>fuller518</code> , <code>fuller519</code> , <code>fuller520</code> , <code>fuller521</code> , <code>fuller522</code> , <code>fuller523</code> , <code>fuller524</code> , <code>fuller525</code> , <code>fuller526</code> , <code>fuller527</code> , <code>fuller528</code> , <code>fuller529</code> , <code>fuller530</code> , <code>fuller531</code> , <code>fuller532</code> , <code>fuller533</code> , <code>fuller534</code> , <code>fuller535</code> , <code>fuller536</code> , <code>fuller537</code> , <code>fuller538</code> , <code>fuller539</code> , <code>fuller540</code> , <code>fuller541</code> , <code>fuller542</code> , <code>fuller543</code> , <code>fuller544</code> , <code>fuller545</code> , <code>fuller546</code> , <code>fuller547</code> , <code>fuller548</code> , <code>fuller549</code> , <code>fuller550</code> , <code>fuller551</code> , <code>fuller552</code> , <code>fuller553</code> , <code>fuller554</code> , <code>fuller555</code> , <code>fuller556</code> , <code>fuller557</code> , <code>fuller558</code> , <code>fuller559</code> , <code>fuller560</code> , <code>fuller561</code> , <code>fuller562</code> , <code>fuller563</code> , <code>fuller564</code> , <code>fuller565</code> , <code>fuller566</code> , <code>fuller567</code> , <code>fuller568</code> , <code>fuller569</code> , <code>fuller570</code> , <code>fuller571</code> , <code>fuller572</code> , <code>fuller573</code> , <code>fuller574</code> , <code>fuller575</code> , <code>fuller576</code> , <code>fuller577</code> , <code>fuller578</code> , <code>fuller579</code> , <code>fuller580</code> , <code>fuller581</code> , <code>fuller582</code> , <code>fuller583</code> , <code>fuller584</code> , <code>fuller585</code> , <code>fuller586</code> , <code>fuller587</code> , <code>fuller588</code> , <code>fuller589</code> , <code>fuller590</code> , <code>fuller591</code> , <code>fuller592</code> , <code>fuller593</code> , <code>fuller594</code> , <code>fuller595</code> , <code>fuller596</code> , <code>fuller597</code> , <code>fuller598</code> , <code>fuller599</code> , <code>fuller600</code> , <code>fuller601</code> , <code>fuller602</code> , <code>fuller603</code> , <code>fuller604</code> , <code>fuller605</code> , <code>fuller606</code> , <code>fuller607</code> , <code>fuller608</code> , <code>fuller609</code> , <code>fuller610</code> , <code>fuller611</code> , <code>fuller612</code> , <code>fuller613</code> , <code>fuller614</code> , <code>fuller615</code> , <code>fuller616</code> , <code>fuller617</code> , <code>fuller618</code> , <code>fuller619</code> , <code>fuller620</code> , <code>fuller621</code> , <code>fuller622</code> , <code>fuller623</code> , <code>fuller624</code> , <code>fuller625</code> , <code>fuller626</code> , <code>fuller627</code> , <code>fuller628</code> , <code>fuller629</code> , <code>fuller630</code> , <code>fuller631</code> , <code>fuller632</code> , <code>fuller633</code> , <code>fuller634</code> , <code>fuller635</code> , <code>fuller636</code> , <code>fuller637</code> , <code>fuller638</code> , <code>fuller639</code> , <code>fuller640</code> , <code>fuller641</code> , <code>fuller642</code> , <code>fuller643</code> , <code>fuller644</code> , <code>fuller645</code> , <code>fuller646</code> , <code>fuller647</code> , <code>fuller648</code> , <code>fuller649</code> , <code>fuller650</code> , <code>fuller651</code> , <code>fuller652</code> , <code>fuller653</code> , <code>fuller654</code> , <code>fuller655</code> , <code>fuller656</code> , <code>fuller657</code> , <code>fuller658</code> , <code>fuller659</code> , <code>fuller660</code> , <code>fuller661</code> , <code>fuller662</code> , <code>fuller663</code> , <code>fuller664</code> , <code>fuller665</code> , <code>fuller666</code> , <code>fuller667</code> , <code>fuller668</code> , <code>fuller669</code> , <code>fuller670</code> , <code>fuller671</code> , <code>fuller672</code> , <code>fuller673</code> , <code>fuller674</code> , <code>fuller675</code> , <code>fuller676</code> , <code>fuller677</code> , <code>fuller678</code> , <code>fuller679</code> , <code>fuller680</code> , <code>fuller681</code> , <code>fuller682</code> , <code>fuller683</code> , <code>fuller684</code> , <code>fuller685</code> , <code>fuller686</code> , <code>fuller687</code> , <code>fuller688</code> , <code>fuller689</code> , <code>fuller690</code> , <code>fuller691</code> , <code>fuller692</code> , <code>fuller693</code> , <code>fuller694</code> , <code>fuller695</code> , <code>fuller696</code> , <code>fuller697</code> , <code>fuller698</code> , <code>fuller699</code> , <code>fuller700</code> , <code>fuller701</code> , <code>fuller702</code> , <code>fuller703</code> , <code>fuller704</code> , <code>fuller705</code> , <code>fuller706</code> , <code>fuller707</code> , <code>fuller708</code> , <code>fuller709</code> , <code>fuller710</code> , <code>fuller711</code> , <code>fuller712</code> , <code>fuller713</code> , <code>fuller714</code> , <code>fuller715</code> , <code>fuller716</code> , <code>fuller717</code> , <code>fuller718</code> , <code>fuller719</code> , <code>fuller720</code> , <code>fuller721</code> , <code>fuller722</code> , <code>fuller723</code> , <code>fuller724</code> , <code>fuller725</code> , <code>fuller726</code> , <code>fuller727</code> , <code>fuller728</code> , <code>fuller729</code> , <code>fuller730</code> , <code>fuller731</code> , <code>fuller732</code> , <code>fuller733</code> , <code>fuller734</code> , <code>fuller735</code> , <code>fuller736</code> , <code>fuller737</code> , <code>fuller738</code> , <code>fuller739</code> , <code>fuller740</code> , <code>fuller741</code> , <code>fuller742</code> , <code>fuller743</code> , <code>fuller744</code> , <code>fuller745</code> , <code>fuller746</code> , <code>fuller747</code> , <code>fuller748</code> , <code>fuller749</code> , <code>fuller750</code> , <code>fuller751</code> , <code>fuller752</code> , <code>fuller753</code> , <code>fuller754</code> , <code>fuller755</code> , <code>fuller756</code> , <code>fuller757</code> , <code>fuller758</code> , <code>fuller759</code> , <code>fuller760</code> , <code>fuller761</code> , <code>fuller762</code> , <code>fuller763</code> , <code>fuller764</code> , <code>fuller765</code> , <code>fuller766</code> , <code>fuller767</code> , <code>fuller768</code> , <code>fuller769</code> , <code>fuller770</code> , <code>fuller771</code> , <code>fuller772</code> , <code>fuller773</code> , <code>fuller774</code> , <code>fuller775</code> , <code>fuller776</code> , <code>fuller777</code> , <code>fuller778</code> , <code>fuller779</code> , <code>fuller780</code> , <code>fuller781</code> , <code>fuller782</code> , <code>fuller783</code> , <code>fuller784</code> , <code>fuller785</code> , <code>fuller786</code> , <code>fuller787</code> , <code>fuller788</code> , <code>fuller789</code> , <code>fuller790</code> , <code>fuller791</code> , <code>fuller792</code> , <code>fuller793</code> , <code>fuller794</code> , <code>fuller795</code> , <code>fuller796</code> , <code>fuller797</code> , <code>fuller798</code> , <code>fuller799</code> , <code>fuller800</code> , <code>fuller801</code> , <code>fuller802</code> , <code>fuller803</code> , <code>fuller804</code> , <code>fuller805</code> , <code>fuller806</code> , <code>fuller807</code> , <code>fuller808</code> , <code>fuller809</code> , <code>fuller810</code> , <code>fuller811</code> , <code>fuller812</code> , <code>fuller813</code> , <code>fuller814</code> , <code>fuller815</code> , <code>fuller816</code> , <code>fuller817</code> , <code>fuller818</code> , <code>fuller819</code> , <code>fuller820</code> , <code>fuller821</code> , <code>fuller822</code> , <code>fuller823</code> , <code>fuller824</code> , <code>fuller825</code> , <code>fuller826</code> , <code>fuller827</code> , <code>fuller828</code> , <code>fuller829</code> , <code>fuller830</code> , <code>fuller831</code> , <code>fuller832</code> , <code>fuller833</code> , <code>fuller834</code> , <code>fuller835</code> , <code>fuller836</code> , <code>fuller837</code> , <code>fuller838</code> , <code>fuller839</code> , <code>fuller840</code> , <code>fuller841</code> , <code>fuller842</code> , <code>fuller843</code> , <code>fuller844</code> , <code>fuller845</code> , <code>fuller846</code> , <code>fuller847</code> , <code>fuller848</code> , <code>fuller849</code> , <code>fuller850</code> , <code>fuller851</code> , <code>fuller852</code> , <code>fuller853</code> , <code>fuller854</code> , <code>fuller855</code> , <code>fuller856</code> , <code>fuller857</code> , <code>fuller858</code> , <code>fuller859</code> , <code>fuller860</code> , <code>fuller861</code> , <code>fuller862</code> , <code>fuller863</code> , <code>fuller864</code> , <code>fuller865</code> , <code>fuller866</code> , <code>fuller867</code> , <code>fuller868</code> , <code>fuller869</code> , <code>fuller870</code> , <code>fuller871</code> , <code>fuller872</code> , <code>fuller873</code> , <code>fuller874</code> , <code>fuller875</code> , <code>fuller876</code> , <code>fuller877</code> , <code>fuller878</code> , <code>fuller879</code> , <code>fuller880</code> , <code>fuller881</code> , <code>fuller882</code> , <code>fuller883</code> , <code>fuller884</code> , <code>fuller885</code> , <code>fuller886</code> , <code>fuller887</code> , <code>fuller888</code> , <code>fuller889</code> , <code>fuller890</code> , <code>fuller891</code> , <code>fuller892</code> , <code>fuller893</code> , <code>fuller894</code> , <code>fuller895</code> , <code>fuller896</code> , <code>fuller897</code> , <code>fuller898</code> , <code>fuller899</code> , <code>fuller900</code> , <code>fuller901</code> , <code>fuller902</code> , <code>fuller903</code> , <code>fuller904</code> , <code>fuller905</code> , <code>fuller906</code> , <code>fuller907</code> , <code>fuller908</code> , <code>fuller909</code> , <code>fuller910</code> , <code>fuller911</code> , <code>fuller912</code> , <code>fuller913</code> , <code>fuller914</code> , <code>fuller915</code> , <code>fuller916</code> , <code>fuller917</code> , <code>fuller918</code> , <code>fuller919</code> , <code>fuller920</code> , <code>fuller921</code> , <code>fuller922</code> , <code>fuller923</code> , <code>fuller924</code> , <code>fuller925</code> , <code>fuller926</code> , <code>fuller927</code> , <code>fuller928</code> , <code>fuller929</code> , <code>fuller930</code> , <code>fuller931</code> , <code>fuller932</code> , <code>fuller933</code> , <code>fuller934</code> , <code>fuller935</code> , <code>fuller936</code> , <code>fuller937</code> , <code>fuller938</code> , <code>fuller939</code> , <code>fuller940</code> , <code>fuller941</code> , <code>fuller942</code> , <code>fuller943</code> , <code>fuller944</code> , <code>fuller945</code> , <code>fuller946</code> , <code>fuller947</code> , <code>fuller948</code> , <code>fuller949</code> , <code>fuller950</code> , <code>fuller951</code> , <code>fuller952</code> , <code>fuller953</code> , <code>fuller954</code> , <code>fuller955</code> , <code>fuller956</code> , <code>fuller957</code> , <code>fuller958</code> , <code>fuller959</code> , <code>fuller960</code> , <code>fuller961</code> , <code>fuller962</code> , <code>fuller963</code> , <code>fuller964</code> , <code>fuller965</code> , <code>fuller966</code> , <code>fuller967</code> , <code>fuller968</code> , <code>fuller969</code> , <code>fuller970</code> , <code>fuller971</code> , <code>fuller972</code> , <code>fuller973</code> , <code>fuller974</code> , <code>fuller975</code> , <code>fuller976</code> , <code>fuller977</code> , <code>fuller978</code> , <code>fuller979</code> , <code>fuller980</code> , <code>fuller981</code> , <code>fuller982</code> , <code>fuller983</code> , <code>fuller984</code> , <code>fuller985</code> , <code>fuller986</code> , <code>fuller987</code> , <code>fuller988</code> , <code>fuller989</code> , <code>fuller990</code> , <code>fuller991</code> , <code>fuller992</code> , <code>fuller993</code> , <code>fuller994</code> , <code>fuller995</code> , <code>fuller996</code> , <code>fuller997</code> , <code>fuller998</code> , <code>fuller999</code> , <code>fuller1000</code> , <code>fuller1001</code> , <code>fuller1002</code> , <code>fuller1003</code> , <code>fuller1004</code> , <code>fuller1005</code> , <code>fuller1006</code> , <code>fuller1007</code> , <code>fuller1008</code> , <code>fuller1009</code> , <code>fuller1010</code> , <code>fuller1011</code> , <code>fuller1012</code> , <code>fuller1013</code> , <code>fuller1014</code> , <code>fuller1015</code> , <code>fuller1016</code> , <code>fuller1017</code> , <code>fuller1018</code> , <code>fuller1019</code> , <code>fuller1020</code> , <code>fuller1021</code> , <code>fuller1022</code> , <code>fuller1023</code> , <code>fuller1024</code> , <code>fuller1025</code> , <code>fuller1026</code> , <code>fuller1027</code> , <code>fuller1028</code> , <code>fuller1029</code> , <code>fuller1030</code> , <code>fuller1031</code> , <code>fuller1032</code> , <code>fuller1033</code> , <code>fuller1034</code> , <code>fuller1035</code> , <code>fuller1036</code> , <code>fuller1037</code> , <code>fuller1038</code> , <code>fuller1039</code> , <code>fuller1040</code> , <code>fuller1041</code> , <code>fuller1042</code> , <code>fuller1043</code> , <code>fuller1044</code> , <code>fuller1045</code> , <code>fuller1046</code> , <code>fuller1047</code> , <code>fuller1048</code> , <code>fuller1049</code> , <code>fuller1050</code> , <code>fuller1051</code> , <code>fuller1052</code> , <code>fuller1053</code> , <code>fuller1054</code> , <code>fuller1055</code> , <code>fuller1056</code> , <code>fuller1057</code> , <code>fuller1058</code> , <code>fuller1059</code> , <code>fuller1060</code> , <code>fuller1061</</code>

erlaubt, nach einem bestimmten Autor zu suchen, und die `--grep` Option nach Stichworten in den Commit Meldungen. (Wenn Du sowohl nach dem Autor als auch nach Stichworten suchen willst, musst Du zusätzlich `--all-match` angeben – andernfalls zeigt der Befehl alle Commits, die entweder das eine oder das andere Kriterium erfüllen.)

Eine letzte sehr nützliche Option, die von `git log` akzeptiert wird, ist ein Pfad. Wenn Du einen Verzeichnis- oder Dateinamen angibst, kannst Du die Ausgabe auf Commits einschränken, die sich auf die jeweiligen Verzeichnisse oder Dateien beziehen. Der Pfad muss als letztes angegeben und mit einem doppelten Bindestrich (`--`) von den Optionen getrennt werden.

Tabelle 2-3 zeigt die besprochenen und einige weitere, übliche Optionen:

Option	Beschreibung
<code>-(n)</code>	Begrenzt die Ausgabe auf die letzten <code>n</code> commits
<code>--since</code> , <code>--after</code>	Zeigt nur Commits, die nach dem angegebenen Datum angelegt wurden
<code>--until</code> , <code>--before</code>	Zeigt nur Commits, die vor dem angegebenen Datum angelegt wurden
<code>--author</code>	Zeigt nur Commits, die von dem angegebenen Autor vorgenommen wurden.
<code>--committer</code>	Zeigt nur Commits, die von dem angegebenen Committer angelegt wurden

Um beispielweise alle Commits aus der Git Quelltext Historie anzuzeigen, die alle der folgende Bedingungen erfüllen:

- Autor des Commits ist Junio Hamano
- Commit Datum Oktober 2008
- Commits, welche Änderungen im Testverzeichnis beinhalten
- Commits, welche keine Merges sind

kannst Du folgenden Befehl verwenden:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
  --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

Aus etwa 20.000 Commits in der Git Quellcode Historie, filtert dieser Befehl gerade einmal 6 Commits heraus, die diesen Kriterien entsprechen.

Grafische Darstellung der Historie

Wenn Dir eine grafische Anzeige der Commit Historie lieber ist, kannst Du das Tcl/Tk Programm `gitk`, welches mit Git ausgeliefert wird, ausprobieren. `gitk` ist im wesentlichen eine grafische Version von `git log` und akzeptiert fast alle Filteroptionen, die `git log` auch akzeptiert. Wenn Du `gitk` in einem Projekt ausführst, siehst Du etwa folgende Ausgabe:



Die Commit Historie wird in der oberen Hälfte des Fensters dargestellt. Daneben ein Graph, der die Branches und Merges zeigt. Nach Auswahl eines Commits, zeigt die Vergleichsanzeige in der unteren Hälfte des Fensters die jeweiligen Änderungen in diesem Commit.

Änderungen rückgängig machen

Es kommt immer wieder mal vor, dass Du Änderungen rückgängig machen willst. Im Folgenden gehen wir auf einige grundlegende Möglichkeiten dazu ein. Sei allerdings vorsichtig damit, denn Du kannst nicht immer alles wieder herstellen, was Du rückgängig gemacht hast. Dies ist eine der wenigen Situationen in Git, in denen man Daten verlieren kann, wenn man etwas falsch macht.

Den letzten Commit ändern

Manchmal hat man einen Commit zu früh angelegt und möglicherweise vergessen, einige Dateien hinzuzufügen, oder eine falsche Commit Meldung verwendet. Wenn Du den letzten Commit korrigieren willst, kannst Du dazu `git commit` zusammen mit der `--amend` Option verwenden:

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

Dieser Befehl verwendet Deine Staging Area für den Commit. Wenn Du seit dem letzten Commit keine Änderungen vorgenommen hast (z.B. wenn Du den Befehl unmittelbar nach einem Commit ausführst), wird der Snapshot exakt genauso aussehen wie der vorherige – alles, was Du dann änderst, ist die Commit Meldung.

Der Texteditor startet wie üblich, aber diesmal enthält er bereits die Meldung aus dem vorherigen Commit. Du kannst diese Meldung wie gewohnt bearbeiten, speichern und die vorherige Meldung dadurch überschreiben.

Wenn Du beispielsweise einen Commit angelegt hast und dann feststellst, dass Du zuvor vergessen hast, die Änderungen in einer bestimmten Datei zur Staging Area hinzuzufügen, kannst Du folgendes tun:

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

Diese drei Befehle legen einen einzigen neuen Commit an – der letzte Befehl ersetzt dabei das Ergebnis des ersten Befehls.

Änderungen aus der Staging Area entfernen

Die nächsten zwei Abschnitte gehen darauf ein, wie Du Änderungen in der Staging Area und dem Arbeitsverzeichnis verwalten kannst. Praktischerweise liefert Dir der Befehl `git status`, den Du verwendest, um den Status dieser beiden Bereiche zu überprüfen, zugleich auch einen Hinweis dafür, wie Du Änderungen rückgängig machen kannst. Nehmen wir beispielsweise an, Du hast zwei Dateien geändert und willst sie als zwei separate Commits anlegen, Du hast aber versehentlich `git add *` ausgeführt und damit beide zur Staging Area hinzugefügt. Wie kannst Du jetzt eine der beiden Änderungen wieder aus der Staging Area nehmen? `git status` gibt Dir einen Hinweis:

```
$ git add .
```

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#       modified:   benchmarks.rb
#
```

Direkt unter der Zeile „Changes to be committed“ findest Du den Hinweis „use git reset HEAD <file>... to unstage“, d.h. „aus der Staging Area zu entfernen“. Wir verwenden nun also diesen Befehl, um die Änderungen an der Datei `benchmarks.rb` aus der Staging Area zu nehmen:

```
$ git reset HEAD benchmarks.rb
benchmarks.rb: locally modified
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
# 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:   benchmarks.rb
#
```

Der Befehl liest sich zunächst vielleicht etwas merkwürdig, aber wie Du siehst, funktioniert er. Die Datei `benchmarks.rb` ist weiterhin modifiziert, befindet sich aber nicht mehr in der Staging Area.

Eine Änderung an einer Datei rückgängig machen

Was aber, wenn Du die Änderungen an der Datei `benchmarks.rb` überhaupt nicht beibehalten willst? D.h., wenn Du sie in den Zustand zurückversetzen willst, in dem sie sich befand, als Du den letzten Commit angelegt hast (oder das Repository geklont hast). Das ist einfach, und glücklicherweise zeigt der `git status` Befehl ebenfalls bereits einen Hinweis dafür an. Die obige Ausgabe enthält den folgenden Text:

```
# 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:   benchmarks.rb
#
```

Das sagt ziemlich klar, was wir zu tun haben um die Änderungen an der Datei zu verwerfen (genauer gesagt, Git tut dies seit der Version 1.6.1 – wenn Du eine ältere Version hast, empfehlen wir dir, sie zu aktualisieren). Wir führen den vorgeschlagenen Befehl also aus:

```
$ git checkout -- benchmarks.rb
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       modified:   README.txt
#
```

Die Änderung wurde also rückgängig gemacht: sie taucht nicht mehr in der Liste der geänderten Dateien auf. Sei Dir bewusst, dass dieser Befehl potentiell gefährlich ist, da er Änderungen an einer Datei vollständig verwirft. Es ist also ratsam, ihn nur dann zu verwenden, wenn Du Dir absolut sicher bist, dass Du die Änderungen nicht mehr brauchst. Für Situationen, in denen Du eine Änderung lediglich vorläufig aus dem Weg räumen willst, werden wir im nächsten Kapitel noch auf Stashing und Branching eingehen – die dazu besser geeignet sind.

Beachte, dass alles was jemals in einem Commit in Git enthalten war, fast immer wieder hergestellt werden kann. Selbst Commits, die sich in gelöschten Branches befanden, oder Commits, die mit einem --amend Commit überschrieben wurden, können wieder hergestellt werden (siehe Kapitel 9 für Datenrettung). Allerdings wirst Du Änderungen, die es nie in einen Commit geschafft haben, wahrscheinlich auch nie wieder restaurieren können.

Mit externen Repositorys arbeiten

Um mit anderen via Git zusammenzuarbeiten, musst Du wissen, wie Du auf externe (engl. „remote“) Repositorys zugreifen kannst. Remote Repositorys sind Versionen Deines Projektes, die im Internet oder irgendwo in einem anderen Netzwerk gespeichert sind. Du kannst mehrere solcher Repositorys haben und Du kannst jedes davon entweder nur lesen oder lesen und schreiben. Mit anderen via Git zusammenzuarbeiten impliziert, solche Repositorys zu verwalten und Daten aus ihnen herunter- oder heraufzuladen, um Deine Arbeit für andere verfügbar zu machen. Um Remote Repositorys zu verwalten, muss man wissen, wie man sie anlegt und wieder entfernt, wenn sie nicht mehr verwendet werden, wie man externe Branches verwalten und nachverfolgen kann, und mehr. In diesem Kapitel werden wir auf diese Aufgaben eingehen.

Remote Repositorys anzeigen

Der `git remote` Befehl zeigt Dir an, welche externen Server Du für Dein Projekt lokal konfiguriert hast, und listet die Kurzbezeichnungen für diese Remote Repository auf. Wenn Du ein Repository geklont hast, solltest Du mindestens `origin` sehen – welches der Standardname ist, den Git für denjenigen Server vergibt, von dem Du geklont hast:

```
$ git clone git://github.com/schacon/ticgit.git
Initialized empty Git repository in /private/tmp/ticgit/.git/
remote: Counting objects: 595, done.
remote: Compressing objects: 100% (269/269), done.
remote: Total 595 (delta 255), reused 589 (delta 253)
Receiving objects: 100% (595/595), 73.31 KiB | 1 KiB/s, done.
Resolving deltas: 100% (255/255), done.
$ cd ticgit
$ git remote
origin
```

Du kannst außerdem die Option `-v` verwenden, welche für jeden Kurznamen auch die jeweilige URL anzeigt, die Git gespeichert hat:

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

Wenn Du mehr als ein Remote Repository konfiguriert hast, zeigt der Befehl alle an. Für mein eigenes Grit Repository sieht das beispielsweise wie folgt aus:

```
$ cd grit
$ git remote -v
bakdoor  git://github.com/bakdoor/grit.git
cho45    git://github.com/cho45/grit.git
defunkt  git://github.com/defunkt/grit.git
koke     git://github.com/koke/grit.git
origin   git@github.com:mojombo/grit.git
```

D.h., mein lokales Repository kennt die Repositorys von all diesen Leuten und ich kann ihre

Beiträge zu meinem Projekt ganz einfach herunterladen und zum Projekt hinzufügen.

Remote Repositorys hinzufügen

Ich habe in vorangegangenen Kapiteln schon Beispiele dafür aufgezeigt, wie man ein Remote Repository hinzufügen kann, aber ich will noch einmal darauf eingehen. Um ein neues Remote Repository mit einem Kurznamen hinzuzufügen, den Du Dir leicht merken kannst, führst Du den Befehl `git remote add [shortname] [url]` aus:

```
$ git remote
origin
$ git remote add pb git://github.com/paulboone/ticgit.git
$ git remote -v
origin  git://github.com/schacon/ticgit.git
pb      git://github.com/paulboone/ticgit.git
```

Jetzt kannst Du den Namen `pb` anstelle der vollständigen URL in verschiedenen Befehlen verwenden. Wenn Du beispielsweise alle Informationen, die in Pauls, aber noch nicht in Deinem eigenen Repository verfügbar sind, herunterladen willst, kannst Du den Befehl `git fetch pb` verwenden:

```
$ git fetch pb
remote: Counting objects: 58, done.
remote: Compressing objects: 100% (41/41), done.
remote: Total 44 (delta 24), reused 1 (delta 0)
Unpacking objects: 100% (44/44), done.
From git://github.com/paulboone/ticgit
 * [new branch]      master      -> pb/master
 * [new branch]      ticgit      -> pb/ticgit
```

Pauls master Branch ist jetzt lokal auf Deinem Rechner als `pb/master` verfügbar – Du kannst ihn mit einem Deiner eigenen Branches zusammenführen oder auf einen lokalen Branch wechseln, um damit zu arbeiten.

Änderungen aus Remote Repositorys herunterladen und herunterladen inkl. zusammenführen

Wie Du gerade gesehen hast, kannst Du Daten aus Remote Repositorys herunterladen, indem Du den folgenden Befehl verwendest:

```
$ git fetch [remote-name]
```

Dieser Befehl lädt alle Daten aus dem Remote Repository herunter, die noch nicht auf Deinem Rechner verfügbar sind. Danach kennt Dein eigenes Repository Verweise auf alle Branches in dem Remote Repository, die Du jederzeit mit Deinen eigenen Branches zusammenführen oder durchschauen kannst. (Wir werden in Kapitel 3 detaillierter darauf eingehen, was genau Branches sind.)

Wenn Du ein Repository geklont hast, legt der Befehl automatisch einen Verweis auf dieses Repository unter dem Namen `origin` an. D.h. `git fetch origin` lädt alle Neuigkeiten herunter, die in dem Remote Repository von anderen hinzugefügt wurden, seit Du es geklont hast (oder zuletzt `git fetch` ausgeführt hast). Es ist wichtig, zu verstehen, dass der `git fetch` Befehl Daten lediglich in Dein lokales Repository lädt. Er führt sich mit Deinen eigenen Commits in keiner Weise zusammen (mergt) oder modifiziert, woran Du gerade arbeitest. D.h. Du musst die heruntergeladenen Änderungen anschließend selbst manuell mit Deinen eigenen zusammenführen, wenn Du das willst.

Wenn Du allerdings einen Branch so aufgesetzt hast, dass er einem Remote Branch „folgt“ (also einen „Tracking Branch“, wir werden im nächsten Abschnitt und in Kapitel 3 noch genauer darauf eingehen), dann kannst Du den Befehl `git pull` verwenden, um automatisch neue Daten herunterzuladen und den externen Branch gleichzeitig mit dem aktuellen, lokalen Branch zusammenzuführen. Das ist oft die bequemere Arbeitsweise. `git clone` setzt Deinen lokalen master Branch deshalb standardmäßig so auf, dass er dem Remote master Branch des geklonten Repositorys folgt (sofern das Remote Repository einen master Branch hat). Wenn Du dann `git pull` ausführst, wird Git die neuen Commits aus dem externen Repository holen und versuchen, sie automatisch mit dem Code zusammenzuführen, an dem Du gerade arbeitest.

Änderungen in ein Remote Repository hochladen

Wenn Du mit Deinem Projekt an einen Punkt gekommen bist, an dem Du es anderen zur Verfügung stellen willst, kannst Du Deine Änderungen in ein gemeinsam genutztes Repository hochladen (engl. „push“). Der Befehl dafür ist einfach: `git push [remote-name] [branch-name]`. Wenn Du Deinen master Branch auf den `origin` Server hochladen willst (noch einmal, wenn Du ein Repository klonst, setzt Git diesen Namen automatisch für dich), dann kannst Du diesen Befehl verwenden:

```
$ git push origin master
```

Das funktioniert nur dann, wenn Du Schreibrechte für das jeweilige Repository besitzt und niemand anders in der Zwischenzeit irgendwelche Änderungen hochgeladen hat. Wenn zwei Leute ein Repository zur gleichen Zeit klonen, dann zuerst der eine seine Änderungen hochlädt und der zweite anschließend versucht, das gleiche zu tun, dann wird sein Versuch korrekterweise abgewiesen. In dieser Situation muss man neue Änderungen zunächst herunterladen und mit seinen eigenen zusammenführen, um sie dann erst hochzuladen. In Kapitel 3 gehen wir noch einmal ausführlicher darauf ein.

Ein Remote Repository durchstöbern

Wenn Du etwas über ein bestimmtes Remote Repository wissen willst, kannst Du den Befehl `git remote show [remote-name]` verwenden. Wenn Du diesen Befehl mit dem entsprechenden Kurznamen, z.B. `origin` verwendest, erhältst Du etwa folgende Ausgabe:

```
$ git remote show origin
* remote origin
```

```
URL: git://github.com/schacon/ticgit.git
Remote branch merged with 'git pull' while on branch master
master
Tracked remote branches
master
ticgit
```

Das zeigt Dir die URL für das Remote Repository, die Information welche Branches verfolgt werden und welcher Branch aus dem Remote Repository mit Deinem eigenen Master zusammengeführt wird, wenn Du `git pull` ausführst.

Dies ist ein eher einfaches Beispiel, das Dir früher oder später so ähnlich über den Weg laufen wird. Wenn Du Git aber täglich verwendest, erhältst Du mit `git remote show` sehr viel mehr Informationen:

```
$ git remote show origin
* remote origin
URL: git@github.com:defunkt/github.git
Remote branch merged with 'git pull' while on branch issues
issues
Remote branch merged with 'git pull' while on branch master
master
New remote branches (next fetch will store in remotes/origin)
caching
Stale tracking branches (use 'git remote prune')
libwalker
walker2
Tracked remote branches
acl
apiv2
dashboard2
issues
master
postgres
Local branch pushed with 'git push'
master:master
```

Dieser Befehl zeigt, welcher Branch automatisch hochgeladen werden wird, wenn Du `git push` auf bestimmten Branches ausführst. Er zeigt außerdem, welche Branches es im Remote Repository gibt, die Du selbst noch nicht hast, welche Branches dort gelöscht wurden, und Branches, die automatisch mit lokalen Branches zusammengeführt werden, wenn Du `git pull` ausführst.

Verweise auf externe Repositories löschen und umbenennen

Wenn Du eine Referenz auf ein Remote Repository umbenennen willst, kannst Du in neueren Git Versionen den Befehl `git remote rename` verwenden, um den Kurznamen zu ändern. Wenn Du beispielsweise `pb` in `paul` umbenennen willst, lautet der Befehl:

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


origin
paul

Beachte dabei, dass dies Deine Branch Namen für Remote Branches ebenfalls ändert. Der Branch, der zuvor mit `pb/master` referenziert werden konnte, heißt jetzt `paul/master`.

Wenn Du eine Referenz aus irgendeinem Grund entfernen willst (z.B. weil Du den Server umgezogen hast oder einen bestimmten Mirror nicht länger verwendest, oder weil jemand vielleicht nicht länger mitarbeitet), kannst Du `git remote rm` verwenden:

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

Tags

Wie die meisten anderen VCS kann Git bestimmte Punkte in der Historie als besonders wichtig markieren, also taggen. Normalerweise verwendet man diese Funktionalität, um Release Versionen zu markieren (z.B. v1.0). In diesem Abschnitt gehen wir darauf ein, wie Du vorhandene Tags anzeigen und neue Tags erstellen kannst, und worin die Unterschiede zwischen verschiedenen Typen von Tags bestehen.

Vorhandene Tags anzeigen

Um die in einem Repository vorhandenen Tags anzuzeigen, kannst Du den Befehl `git tag` ohne irgendwelche weiteren Optionen verwenden:

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

Dieser Befehl listet die Tags in alphabetischer Reihenfolge auf. Die Reihenfolge ist aber eigentlich nicht so wichtig.

Du kannst auch nach Tags mit einem bestimmten Muster suchen. Das Git Quellcode Repository enthält beispielsweise mehr als 240 Tags. Wenn Du nur an denjenigen interessiert bist, die zur Version 1.4.2 gehören, kannst Du folgendes tun:

```
$ git tag -l 'v1.4.2.*'
v1.4.2.1
v1.4.2.2
v1.4.2.3
v1.4.2.4
```

Neue Tags anlegen

Git kennt im wesentlichen zwei Typen von Tags: einfache (engl. lightweight) und kommentierte (engl. annotated) Tags. Ein einfacher Tag ist wie ein Branch, der sich niemals ändert – es ist lediglich ein Zeiger auf einen bestimmten Commit. Kommentierte Tags dagegen werden als vollwertige Objekte in der Git Datenbank gespeichert. Sie haben eine Checksumme, beinhalten Namen und E-Mail Adresse desjenigen, der den Tag angelegt hat, das jeweilige Datum sowie eine Meldung. Sie können überdies mit GNU Privacy Guard (GPG) signiert und verifiziert werden. Generell empfiehlt sich deshalb, kommentierte Tags anzulegen. Wenn man aber aus irgendeinem Grund einen temporären Tag anlegen will, für den all diese zusätzlichen Informationen nicht nötig sind, dann kann man auf einfache Tags zurückgreifen.

Kommentierte Tags

Einen kommentierten Tag legst Du an, indem Du dem `git tag` Befehl die Option `-a` übergibst:

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

Die Option -m gibt dabei wiederum die Meldung an, die zum Tag hinzugefügt wird. Wenn Du keine Meldung angibst, startet Git wie üblich Deinen Editor, sodass Du eine Meldung eingeben kannst.

git show zeigt Dir dann folgenden Tag zusammen mit dem jeweiligen Commit, auf den der Tag verweist, an:

```
$ git show v1.4
tag v1.4
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 14:45:11 2009 -0800

my version 1.4
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sun Feb 8 19:02:46 2009 -0800

    Merge branch 'experiment'
```

Die Ausgabe listet also zunächst die Informationen über denjenigen auf, der den Tag angelegt hat, sowie die Tag Meldung und dann die Commit Informationen selbst.

Signierte Tags

Wenn Du einen privaten GPG Schlüssel hast, kannst Du Deine Tags zusätzlich mit GPG signieren. Dazu verwendest Du einfach die Option -s anstelle von -a:

```
$ 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@gee-mail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
```

Wenn Du jetzt git show auf diesen Tag anwendest, siehst Du, dass der Tag Deine GPG Signatur hinterlegt hat:

```
$ git show v1.5
tag v1.5
Tagger: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Feb 9 15:22:20 2009 -0800

my signed 1.5 tag
-----BEGIN PGP SIGNATURE-----
Version: GnuPG v1.4.8 (Darwin)
```

```
iEYEA BECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN
Ki0An2JeAVUCAiJ70x6ZEtK+NvZAJ82/
=WryJ
-----END PGP SIGNATURE-----
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800
```

Merge branch 'experiment'

Darauf, wie Du signierte Tags verifizieren kannst, werden wir gleich noch eingehen.

Einfache Tags

Einfache Tags sind die zweite Form von Tags, die Git kennt. Für einen einfachen Tag wird im wesentlichen die jeweilige Commit Prüfsumme, und sonst keine andere Information, in einer Datei gespeichert. Um einen einfachen Tag anzulegen, verwendest Du einfach keine der drei Optionen -a, -s und -m:

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

Wenn Du jetzt `git show` auf den Tag ausführst, siehst Du keine der zusätzlichen Tag Informationen. Der Befehl zeigt einfach den jeweiligen Commit:

```
$ git show v1.4-lw
commit 15027957951b64cf874c3557a0f3547bd83b3ff6
Merge: 4a447f7... a6b4c97...
Author: Scott Chacon <schacon@gee-mail.com>
Date: Sun Feb 8 19:02:46 2009 -0800
```

Merge branch 'experiment'

Tags verifizieren

Um einen signierten Tag zu verifizieren, kannst Du `git tag -v [Tag Name]` verwenden. Dieser Befehl verwendet GPG, um die Signatur mit Hilfe des öffentlichen Schlüssels des Signierenden zu verifizieren – weshalb Du diesen Schlüssel in Deinem Schlüsselbund haben musst:

```
$ git tag -v v1.4.2.1
object 883653babd8ee7ea23e6a5c392bb739348b1eb61
type commit
tag v1.4.2.1
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700
```

GIT 1.4.2.1

Minor fixes since 1.4.2, including git-mv and git-http with alternates.

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Good signature from "Junio C Hamano <junkio@cox.net>"

gpg: aka "[jpeg image of size 1513]"

Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A

Wenn Du den öffentlichen Schlüssel des Signierenden nicht in Deinem Schlüsselbund hast, wirst Du statt dessen eine Meldung sehen wie:

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'

Nachträglich taggen

Du kannst Commits jederzeit taggen, auch lange Zeit nachdem sie angelegt wurden. Nehmen wir an, Deine Commit Historie sieht wie folgt aus:

```
$ git log --pretty=oneline
15027957951b64cf874c3557a0f3547bd83b3ff6 Merge branch 'experiment'
a6b4c97498bd301d84096da251c98a07c7723e65 beginning write support
0d52aaab4479697da7686c15f77a3d64d9165190 one more thing
6d52a271eda8725415634dd79daabbc4d9b6008e Merge branch 'experiment'
0b7434d86859cc7b8c3d5e1dddfed66ff742fcbc added a commit function
4682c3261057305bdd616e23b64b0857d832627b added a todo file
166ae0c4d3f420721acbb115cc33848dfcc2121a started write support
9fceb02d0ae598e95dc970b74767f19372d61af8 updated rakefile
964f16d36dfccde844893cac5b347e7b3d44abbc commit the todo
8a5cbc430f1a9c3d00faaeffd07798508422908a updated readme
```

Nehmen wir an, dass Du vergessen hast, Version v1.2 des Projekts zu taggen und dass dies der Commit „updated rakefile“ gewesen ist. Du kannst diesen jetzt im Nachhinein taggen, indem Du die Checksumme des Commits (oder einen Teil davon) am Ende des Befehls angibst:

```
$ git tag -a v1.2 -m 'version 1.2' 9fceb02
```

Du siehst jetzt, dass Du einen Tag für den Commit angelegt hast:

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

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

```
version 1.2
commit 9fceb02d0ae598e95dc970b74767f19372d61af8
Author: Magnus Chacon <mchacon@gee-mail.com>
Date: Sun Apr 27 20:43:35 2008 -0700
```

```
updated rakefile
```

```
...
```

Tags veröffentlichen

Der `git push` Befehl lädt Tags nicht von sich aus auf externe Server. Stattdessen muss Du Tags explizit auf einen externen Server hochladen, nachdem Du sie angelegt hast. Der Vorgang entspricht dem bei Branches: Du kannst den Befehl `git push origin [tagname]` verwenden.

```
$ git push origin v1.5
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag] v1.5 -> v1.5
```

Wenn Du viele Tags auf einmal hochladen willst, kannst Du dem `git push` Befehl außerdem die `--tags` Option übergeben und auf diese Weise sämtliche Tags auf dem Remote Server veröffentlichen, die dort noch nicht bekannt sind.

```
$ git push origin --tags
Counting objects: 50, done.
Compressing objects: 100% (38/38), done.
Writing objects: 100% (44/44), 4.56 KiB, done.
Total 44 (delta 18), reused 8 (delta 1)
To git@github.com:schacon/simplegit.git
* [new tag] v0.1 -> v0.1
* [new tag] v1.2 -> v1.2
* [new tag] v1.4 -> v1.4
* [new tag] v1.4-lw -> v1.4-lw
* [new tag] v1.5 -> v1.5
```

Wenn jetzt jemand anderes das Repository klonet oder von dort aktualisiert, wird er all diese Tags ebenfalls erhalten.

Tipps und Tricks

Bevor wir zum Ende dieses Grundlagenkapitels kommen, möchten wir noch einige Tipps und Tricks vorstellen, die Dir den Umgang mit Git ein bisschen vereinfachen können. Du kannst Git natürlich einsetzen, ohne diese Tipps anzuwenden, und wir werden später in diesem Buch auch nicht darauf Bezug nehmen oder sie voraussetzen. Aber wir finden, Du solltest sie kennen, weil sie einfach nützlich sind.

Auto-Vervollständigung

Wenn Du die Bash Shell verwendest, dann kannst Du ein Skript für die Git Auto-Vervollständigung einbinden. Du kannst dieses Skript direkt aus den Git Quellen von <https://github.com/git/git/blob/master/contrib/completion/git-completion.bash> herunterladen. Kopiere diese Datei in Dein Home Verzeichnis und füge die folgende Zeile in Deine `.bashrc` Datei hinzu:

```
source ~/git-completion.bash
```

Wenn Du Git Auto-Vervollständigung für alle Benutzer Deines Rechners aufsetzen willst, kopiere das Skript in das Verzeichnis `/opt/local/etc/bash_completion.d` (auf Mac OS X Systemen) bzw. `/etc/bash_completion.d/` (auf Linux Systemen). Bash sucht in diesem Verzeichnis nach Erweiterungen für die Autovervollständigung und lädt sie automatisch.

Auf Windows Systemen sollte die Autovervollständigung bereits aktiv sein, wenn Du die Git Bash aus dem msysGit Paket verwendest.

Während Du einen Git Befehl eintippst, kannst Du die Tab Taste drücken und Du erhältst eine Auswahl von Vorschlägen, aus denen Du auswählen kannst:

```
$ git co<tab><tab>
commit config
```

D.h., wenn Du `git co` schreibst und dann die Tab Taste zwei Mal drückst, erhältst Du die Vorschläge `commit` und `config`. Wenn Du Tab nur ein Mal drückst, vervollständigt den Befehl Deine Eingabe direkt zu `git commit`.

Das funktioniert auch mit Optionen – was oftmals noch hilfreicher ist. Wenn Du beispielsweise `git log` verwenden willst und Dich nicht an eine bestimmte Option erinnern kannst, schreibst Du einfach den Befehl und drückst die Tab Taste, um die Optionen anzuzeigen:

```
$ git log --s<tab>
--shortstat  --since=  --src-prefix=  --stat  --summary
```

Du musst also nicht dauernd die Dokumentation zu Rate ziehen und erspart Dir somit etwas Zeit. Ein toller Trick, nicht wahr?

Git Aliase

Git versucht nicht zu erraten, welchen Befehl Du verwenden willst, wenn Du ihn nur teilweise eingibst. Wenn Du lange Befehle nicht immer wieder eintippen willst, kannst Du mit `git config` auf einfache Weise Aliase definieren. Hier einige Beispiele, die Du vielleicht nützlich findest:

```
$ 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 heißt, dass Du z.B. einfach `git ci` anstelle von `git commit` schreiben kannst. Wenn Du Git oft verwendest, werden Dir sicher weitere Befehle begegnen, die Du sehr oft nutzt. In diesem Fall zögere nicht, weitere Aliase zu definieren.

Diese Technik kann auch dabei helfen, Git Befehle zu definieren, von denen Du denkst, es sollte sie geben:

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

Das bewirkt, dass die beiden folgenden Befehle äquivalent sind:

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

Unser neuer Alias ist wahrscheinlich aussagekräftiger, oder? Ein weiterer, typischer Alias ist der `last` Befehl:

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

Auf diese Weise kannst Du leicht den letzten Commit nachschlagen:

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800
```

```
    test for current head
```

```
    Signed-off-by: Scott Chacon <schacon@example.com>
```

Wie Du Dir denken kannst, ersetzt Git ganz einfach den Alias mit dem jeweiligen Befehl, für den er definiert ist. Wenn Du allerdings einen externen Befehl anstelle eines Git Befehls ausführen willst, kannst Du den Befehl mit einem Ausführungszeichen (!) am Anfang kennzeichnen. Das ist in der Regel nützlich, wenn Du Deine eigenen Hilfsmittel schreibst, um Git zu erweitern. Wir können das demonstrieren, indem wir `git visual` als `gitk` definieren:

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


Zusammenfassung

Du solltest jetzt in der Lage sein, die wichtigsten Git Befehle einzusetzen und Repositorys neu zu erzeugen und zu klonen, Änderungen vorzunehmen und zur Staging Area hinzuzufügen, Commits anzulegen und die Historie aller Commits in einem Repository zu durchsuchen. Als nächstes werden wir auf ein herausragendes Feature von Git eingehen: das Branch Konzept.

Git Branching

Nahezu jedes VCS unterstützt eine Form von Branching. Branching bedeutet, dass Du von der Hauptentwicklungslinie abzweigst und unabhängig von dem Hauptzweig weiterarbeitest. Bei vielen VCS ist das ein umständlicher und komplizierter Prozess. Nicht selten ist es notwendig, dass eine Kopie des kompletten Arbeitsverzeichnisses erstellt werden muss, was bei großen Projekten eine ganze Weile dauern kann.

Manche Leute bezeichnen Gits Branching-Modell als dessen „Killer-Feature“, was Git zweifellos von dem Rest der VCS-Community abhebt. Aber was macht es so besonders? Die Art und Weise, wie Git Branches behandelt, ist unglaublich leichtfüßig. Dies führt dazu, dass nahezu jede Branch-Operation verzögerungsfrei ausgeführt wird. Auch das Hin- und Herschalten zwischen einzelnen Entwicklungszweigen läuft genauso schnell ab. Im Gegensatz zu anderen VCS ermutigt Git zu einer Arbeitsweise mit häufigem Branching und Merging – oft mehrmals am Tag. Die Branching-Funktion zu verstehen und zu meistern gibt Dir ein mächtiges und einmaliges Werkzeug an die Hand und kann Deine Art zu entwickeln buchstäblich verändern.

Was ist ein Branch?

Um wirklich zu verstehen wie Git Branching durchführt, müssen wir einen Schritt zurück gehen und untersuchen wie Git die Daten speichert. Wie Du Dich vielleicht noch an Kapitel 1 erinnerst, speichert Git seine Daten nicht als Serie von Änderungen oder Unterschieden, sondern als Serie von Schnappschüssen.

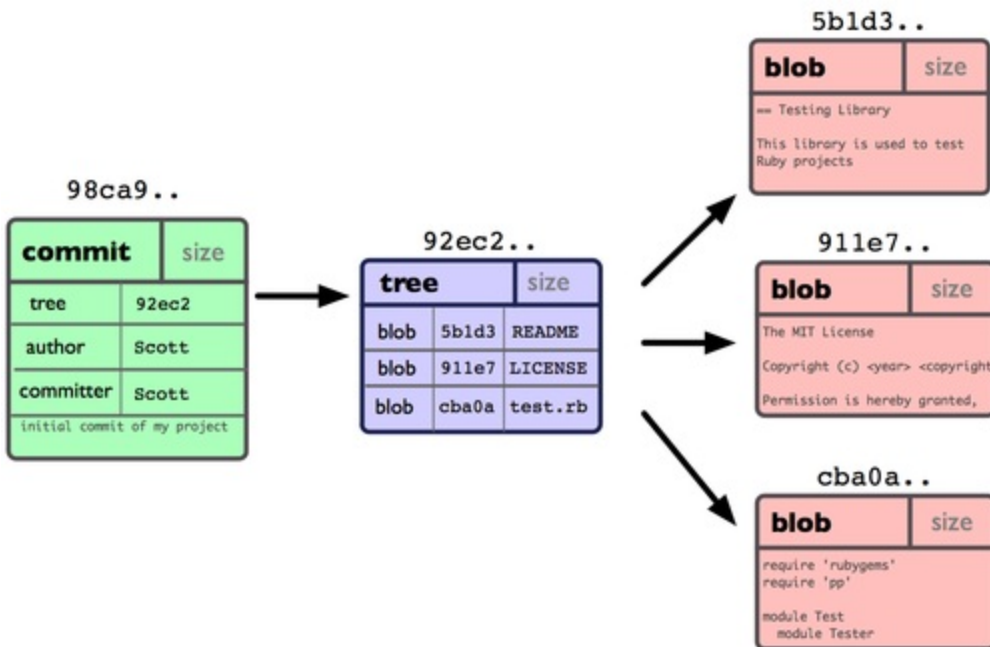
Wenn Du in Git committest, speichert Git ein sogenanntes Commit-Objekt. Dieses enthält einen Zeiger zu dem Schnappschuss mit den Objekten der Staging-Area, dem Autor, den Commit-Metadaten und einem Zeiger zu den direkten Eltern des Commits. Ein initialer Commit hat keine Eltern-Commits, ein normaler Commit stammt von einem Eltern-Commit ab und ein Merge-Commit, welcher aus einer Zusammenführung von zwei oder mehr Branches resultiert, besitzt ebenso viele Eltern-Commits.

Um das zu verdeutlichen, lass uns annehmen, Du hast ein Verzeichnis mit drei Dateien, die Du alle zu der Staging-Area hinzufügst und in einem Commit verpackst. Durch das Stagen der Dateien erzeugt Git für jede Datei eine Prüfsumme (der SHA-1 Hash, den wir in Kapitel 1 erwähnt haben), speichert diese Version der Datei im Git-Repository (Git referenziert auf diese als Blobs) und fügt die Prüfsumme der Staging-Area hinzu:

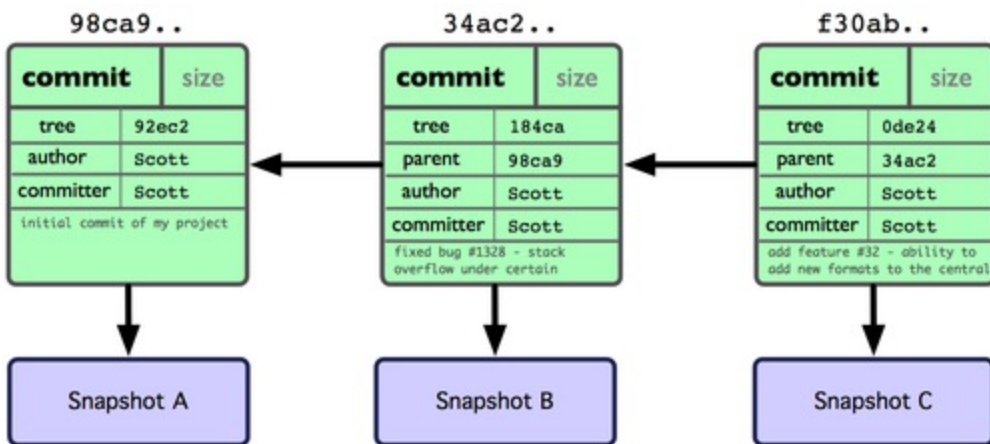
```
$ git add README test.rb LICENSE
$ git commit -m 'initial commit of my project'
```

Wenn Du einen Commit mit dem Kommando `git commit` erstellst, erzeugt Git für jedes Projektverzeichnis eine Prüfsumme und speichert diese als sogenanntes tree-Objekt im Git Repository. Git erzeugt dann ein Commit Objekt, das die Metadaten und den Zeiger zum tree-Objekt des Wurzelverzeichnis enthält, um bei Bedarf den Snapshot erneut erzeugen zu können.

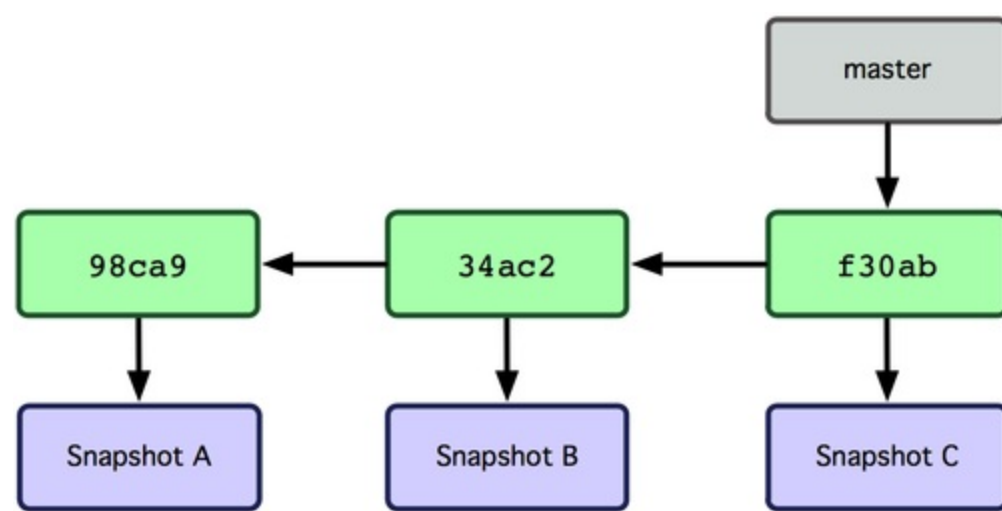
Dein Git-Repository enthält nun fünf Objekte: einen Blob für den Inhalt jeder der drei Dateien, einen Baum, der den Inhalt des Verzeichnisses auflistet und spezifiziert welcher Dateiname zu welchem Blob gehört, sowie einen Zeiger, der auf die Wurzel des Projektbaumes und die Metadaten des Commits verweist. Prinzipiell sehen Deine Daten im Git Repository wie in Abbildung 3-1 aus.



Wenn Du erneut etwas änderst und wieder einen Commit machst, wird dieser einen Zeiger enthalten, der auf den Vorhergehenden verweist. Nach zwei weiteren Commits könnte die Historie wie in Abbildung 3-2 aussehen.



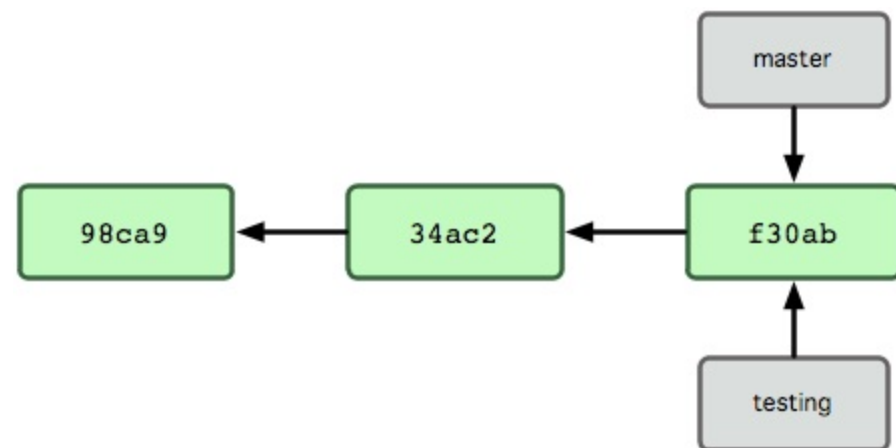
Ein Branch in Git ist nichts anderes als ein simpler Zeiger auf einen dieser Commits. Der Standardname eines Git-Branches lautet `master`. Mit dem initialen Commit erhältst Du einen `master`-Branch, der auf Deinen letzten Commit zeigt. Mit jedem Commit bewegt er sich automatisch vorwärts.



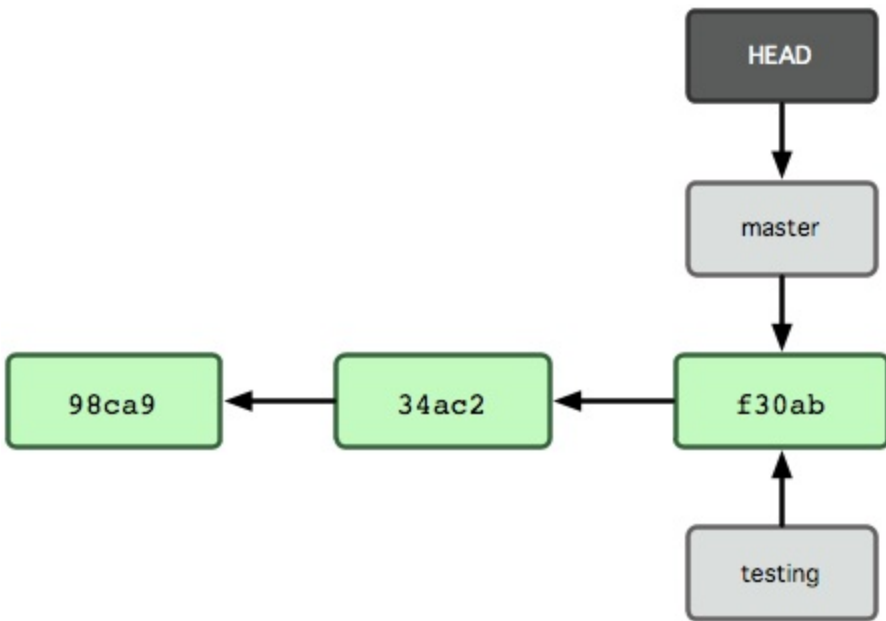
Was passiert, wenn Du einen neuen Branch erstellst? Nun, zunächst wird ein neuer Zeiger erstellt. Sagen wir, Du erstellst einen neuen Branch mit dem Namen `testing`. Das machst Du mit dem `git branch` Befehl:

```
$ git branch testing
```

Dies erzeugt einen neuen Zeiger, der auf den gleichen Commit zeigt, auf dem Du gerade arbeitest (siehe Abbildung 3-4).



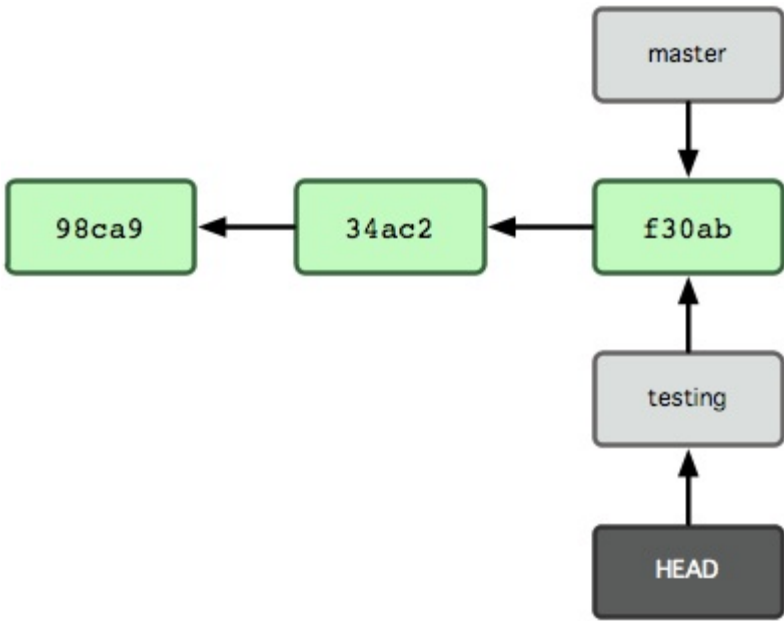
Woher weiß Git, welchen Branch Du momentan verwendest? Dafür gibt es einen speziellen Zeiger mit dem Namen `HEAD`. Berücksichtige, dass dieses Konzept sich grundsätzlich von anderen `HEAD`-Konzepten anderer VCS, wie Subversion oder CVS, unterscheidet. Bei Git handelt es sich bei `HEAD` um einen Zeiger, der auf Deinen aktuellen lokalen Branch zeigt. In dem Fall bist Du aber immer noch auf dem `master`-Branch. Das `git branch` Kommando hat nur einen neuen Branch erstellt, aber nicht zu diesem gewechselt (siehe Abbildung 3-5).



Um zu einem anderen Branch zu wechseln, benutze das Kommando `git checkout`. Lass uns nun zu unserem neuen Branch `testing` wechseln:

```
$ git checkout testing
```

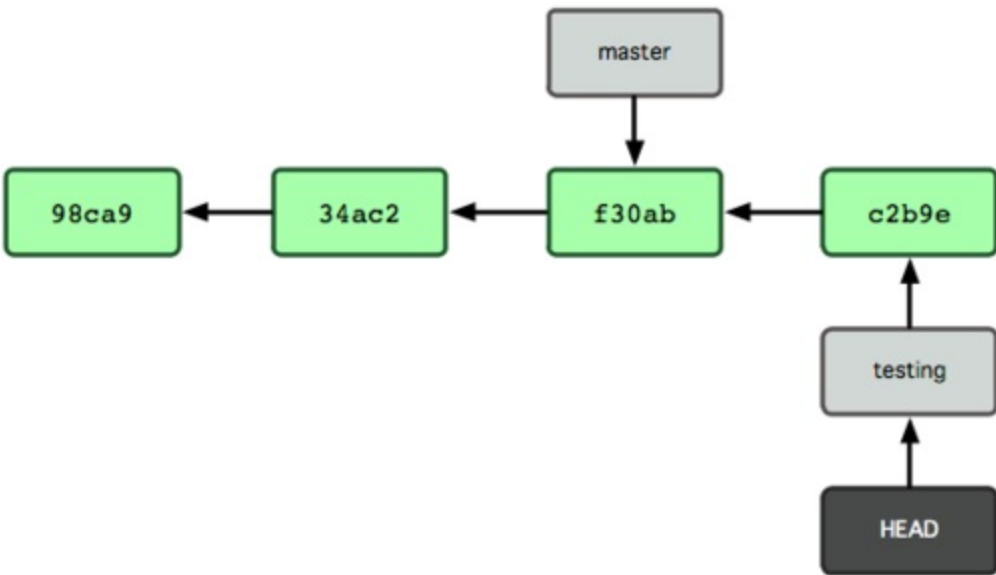
Das lässt `HEAD` neuerdings auf den Branch „testing“ verweisen (siehe Abbildung 3-6).



Und was bedeutet das? Ok, lass uns noch einen weiteren Commit machen:

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

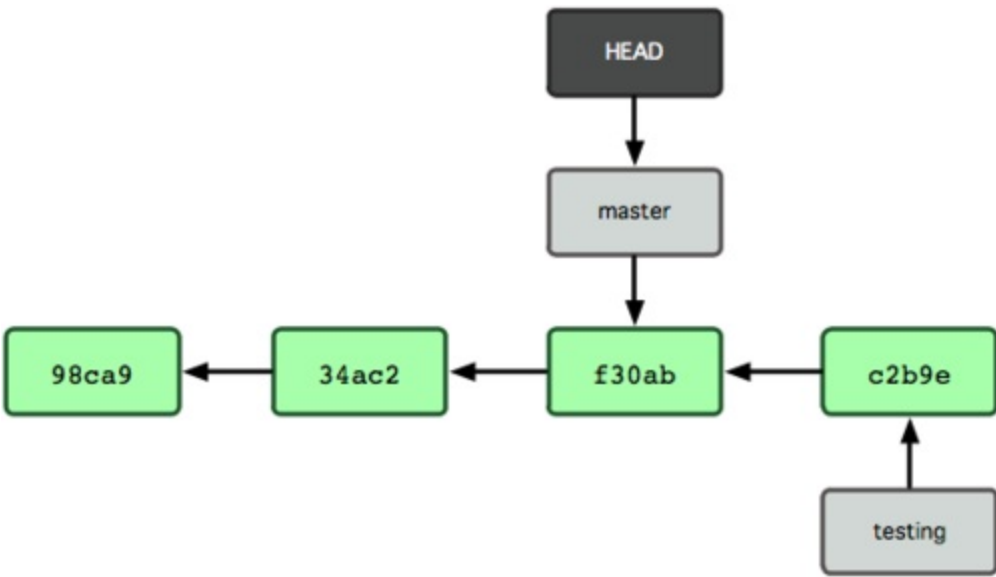
Abbildung 3-7 verdeutlicht das Ergebnis.



Das ist interessant, denn Dein Branch `testing` hat sich jetzt voranbewegt und Dein `master`-Branch zeigt immer noch auf seinen letzten Commit. Den Commit, den Du zuletzt bearbeitet hattest, bevor Du mit `git checkout` den aktuellen Zweig gewechselt hast. Lass uns zurück zu dem `master`-Branch wechseln:

```
$ git checkout master
```

Abbildung 3-8 zeigt das Ergebnis.



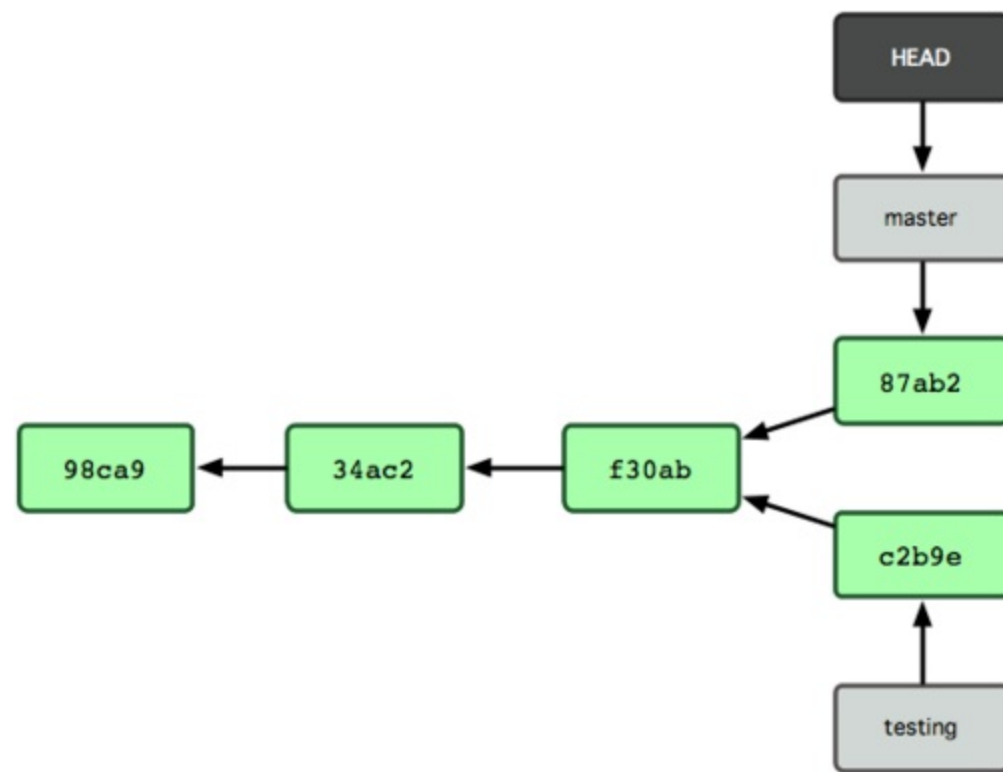
Das Kommando hat zwei Dinge veranlasst. Zum einen bewegt es den `HEAD`-Zeiger zurück zum `master`-Branch, zum anderen setzt es alle Dateien im Arbeitsverzeichnis auf den Bearbeitungsstand des letzte Commits in diesem Zweig zurück. Das bedeutet aber auch, dass nun alle Änderungen am Projekt vollkommen unabhängig von älteren Projektversionen erfolgen. Kurz gesagt, werden alle Änderungen aus dem `testing`-Zweig vorübergehend rückgängig gemacht und Du hast die Möglichkeit einen vollkommen neuen Weg in der Entwicklung einzuschlagen.

Lass uns ein paar Änderungen machen und mit einem Commit festhalten:

```
$ vim test.rb
```

```
$ git commit -a -m 'made other changes'
```

Nun verzweigen sich die Projektverläufe (siehe Abbildung 3-9). Du hast einen Branch erstellt und zu ihm gewechselt, hast ein bisschen gearbeitet, bist zu Deinem Haupt-Zweig zurückgekehrt und hast da was ganz anderes gemacht. Beide Arbeiten existieren vollständig unabhängig voneinander in zwei unterschiedlichen Branches. Du kannst beliebig zwischen den beiden Zweigen wechseln und sie zusammenführen, wenn Du meinst es wäre soweit. Und das alles hast Du mit simplen `branch` und `checkout`-Befehlen vollbracht.



Branches können in Git spielend erstellt und entfernt werden, da sie nur kleine Dateien sind, die eine 40 Zeichen lange SHA-1 Prüfsumme der Commits enthalten, auf die sie verweisen. Einen neuen Zweig zu erstellen erzeugt ebenso viel Aufwand wie das Schreiben einer 41 Byte großen Datei (40 Zeichen und einen Zeilenumbruch).

Das steht im krassen Gegensatz zu dem Weg, den die meisten andere VCS Tools beim Thema Branching einschlagen. Diese kopieren oftmals jeden neuen Entwicklungszweig in ein weiteres Verzeichnis, was – je nach Projektgröße – mehrere Minuten in Anspruch nehmen kann, wohingegen Git diese Aufgabe sofort erledigt. Da wir außerdem immer den Ursprungs-Commit festhalten, lässt sich problemlos eine gemeinsame Basis für eine Zusammenführung finden und umsetzen. Diese Eigenschaft soll Entwickler ermutigen Entwicklungszweige häufig zu erstellen und zu nutzen.

Lass uns mal sehen, warum Du das machen solltest.

Einfaches Branching und Merging

Lass uns das Ganze an einem Beispiel durchgehen, dessen Workflow zum Thema Branching und Zusammenführen Du im echten Leben verwenden kannst. Folge einfach diesen Schritten:

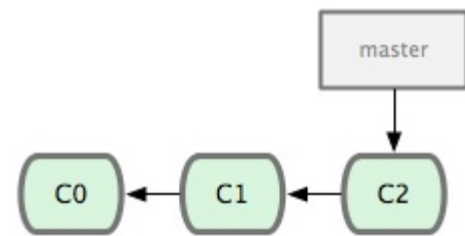
1. Arbeite an einer Webseite.
2. Erstell einen Branch für irgendeine neue Geschichte, an der Du arbeitest.
3. Arbeite in dem Branch.

In diesem Augenblick kommt ein Anruf, dass ein kritisches Problem aufgetreten ist und sofort gelöst werden muss. Du machst folgendes:

1. Geh zurück zu Deinem „Produktiv“-Zweig.
2. Erstelle eine Branch für den Hotfix.
3. Nach dem Testen führst Du den Hotfix-Branch mit dem „Produktiv“-Branch zusammen.
4. Schalte wieder auf Deine alte Arbeit zurück und werkel weiter.

Branching Grundlagen

Sagen wir, Du arbeitest an Deinem Projekt und hast bereits einige Commits durchgeführt (siehe Abbildung 3-10).



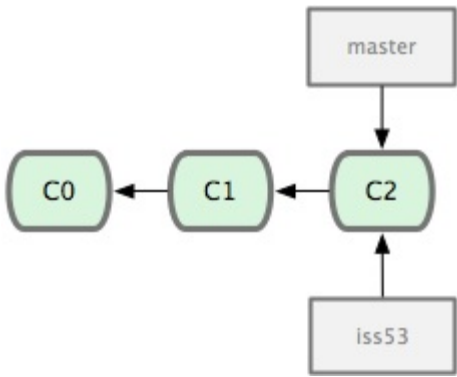
Du hast Dich dafür entschieden an dem Issue #53, des Issue-Trackers XY, zu arbeiten. Um eines klarzustellen, Git ist an kein Issue-Tracking-System gebunden. Da der Issue #53 allerdings ein Schwerpunktthema betrifft, wirst Du einen neuen Branch erstellen um daran zu arbeiten. Um in einem Arbeitsschritt einen neuen Branch zu erstellen und zu aktivieren kannst Du das Kommando `git checkout` mit der Option `-b` verwenden:

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

Das ist die Kurzform von

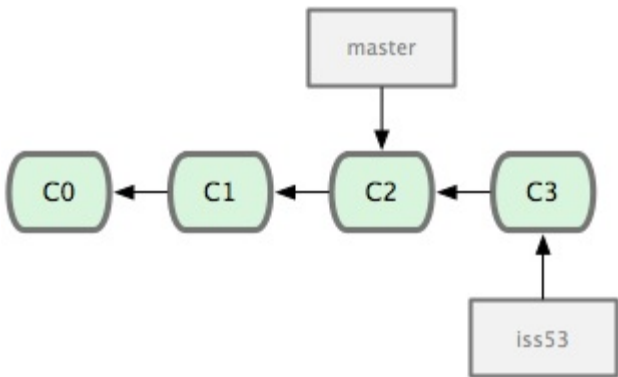
```
$ git branch iss53
$ git checkout iss53
```

Abbildung 3-11 verdeutlicht das Ergebnis.



Du arbeitest an Deiner Web-Seite und machst ein paar Commits. Das bewegt den `iss53`-Branch vorwärts, da Du ihn ausgebucht hast (das heißt, dass Dein HEAD-Zeiger darauf verweist; siehe Abbildung 3-12):

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



Nun bekommst Du einen Anruf, in dem Dir mitgeteilt wird, dass es ein Problem mit der Internet-Seite gibt, das Du umgehend beheben sollst. Mit Git musst Du Deine Fehlerkorrektur nicht zusammen mit den `iss53`-Änderungen einbringen. Und Du musst keine Zeit damit verschwenden Deine bisherigen Änderungen rückgängig zu machen, bevor Du mit der Fehlerbehebung an der Produktionsumgebung beginnen kannst. Alles was Du tun musst, ist zu Deinem MASTER-Branch wechseln.

Beachte jedoch, dass Dich Git den Branch nur wechseln lässt, wenn bisherige Änderungen in Deinem Arbeitsverzeichnis oder Deiner Staging-Area nicht in Konflikt mit dem Zweig stehen, zu dem Du nun wechseln möchtest. Am besten es liegt ein sauberer Status vor wenn man den Branch wechselt. Wir werden uns später mit Wegen befassen, dieses Verhalten zu umgehen (namentlich „Stashing“ und „Commit Amending“). Vorerst aber hast Du Deine Änderungen bereits comitted, sodass Du zu Deinem MASTER-Branch zurückwechseln kannst.

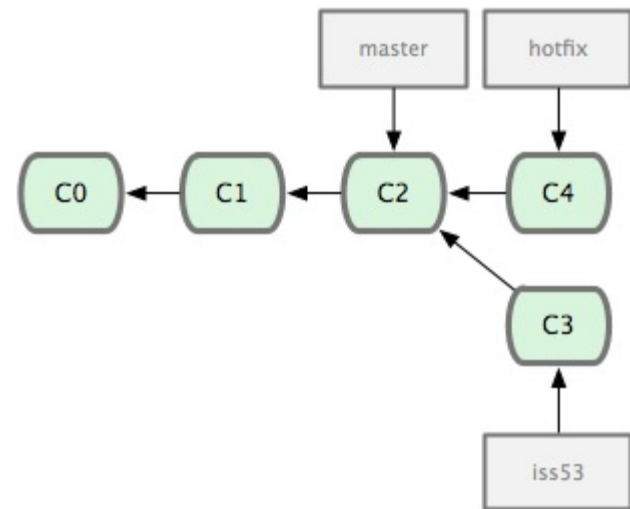
```
$ git checkout master
Switched to branch "master"
```

Zu diesem Zeitpunkt befindet sich das Arbeitsverzeichnis des Projektes in exakt dem gleichen Zustand, in dem es sich befand, als Du mit der Arbeit an Issue #53 begonnen hast und Du kannst Dich direkt auf Deinen Hotfix konzentrieren. Dies ist ein wichtiger Moment um sich vor Augen zu halten, dass Git Dein Arbeitsverzeichnis auf den Zustand des Commits, auf den dieser Branch

zeigt, zurücksetzt. Es erstellt, entfernt und verändert Dateien automatisch, um sicherzustellen, dass Deine Arbeitskopie haargenau so aussieht wie der Zweig nach Deinem letzten Commit.

Nun hast Du einen Hotfix zu erstellen. Lass uns dazu einen Hotfix-Branch erstellen, an dem Du bis zu dessen Fertigstellung arbeitest (siehe Abbildung 3-13):

```
$ git checkout -b hotfix
Switched to a new branch "hotfix"
$ vim index.html
$ git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
1 files changed, 0 insertions(+), 1 deletions(-)
```

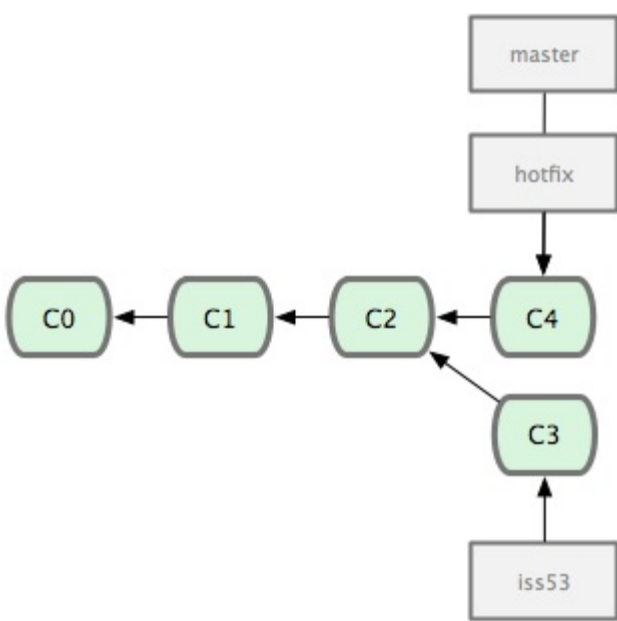


Mach Deine Tests, stell sicher das sich der Hotfix verhält wie erwartet und führe ihn mit dem Master-Branch zusammen, um ihn in die Produktionsumgebung zu integrieren. Das machst Du mit dem `git merge`-Kommando:

```
$ git checkout master
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 README |      1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Du wirst die Mitteilung „Fast Forward“ während des Zusammenführens bemerken. Da der neue Commit direkt von dem ursprünglichen Commit, auf den sich der nun eingebrachte Zweig bezieht, abstammt, bewegt Git einfach den Zeiger weiter. Mit anderen Worten kann Git den neuen Commit, durch verfolgen der Commitabfolge, direkt erreichen, dann bewegt es ausschließlich den Branch-Zeiger. Zu einer tatsächlichen Kombination der Commits besteht ja kein Anlass. Dieses Vorgehen wird „Fast Forward“ genannt.

Deine Modifikationen befinden sich nun als Schnappschuss in dem Commit, auf den der master-Branch zeigt, diese lassen sich nun veröffentlichen (siehe Abbildung 3-14).



Nachdem Dein superwichtiger Hotfix veröffentlicht wurde, kannst Du Dich wieder Deiner ursprünglichen Arbeit zuwenden. Vorher wird sich allerdings des nun nutzlosen Hotfix-Zweiges entledigt, schließlich zeigt der Master-Branch ebenfalls auf die aktuelle Version. Du kannst ihn mit der -d-Option von `git branch` entfernen:

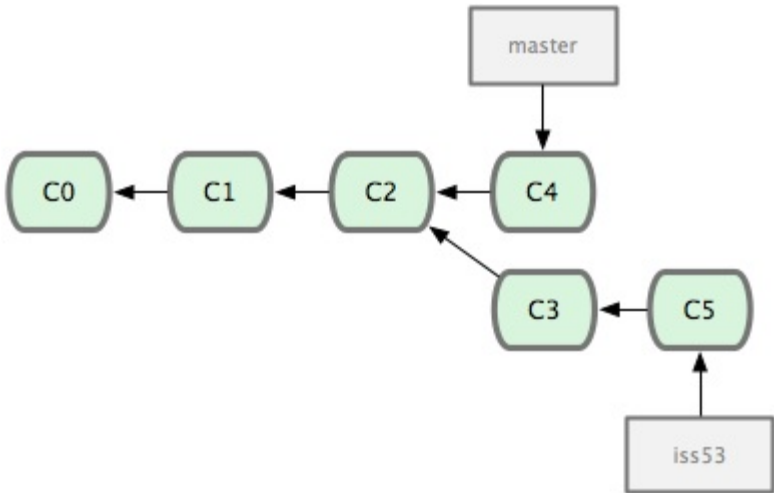
```

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

Nun kannst Du zu Deinem Issue #53-Branch zurückwechseln und mit Deiner Arbeit fortfahren (Abbildung 3-15):

```

$ git checkout iss53
Switched to branch "iss53"
$ vim index.html
$ git commit -a -m 'finished the new footer [issue 53]'
[iss53]: created ad82d7a: "finished the new footer [issue 53]"
1 files changed, 1 insertions(+), 0 deletions(-)
  
```



An dieser Stelle ist anzumerken, dass die Änderungen an dem `hotfix`-Branch nicht in Deinen `iss53`-Zweig eingeflossen sind. Falls nötig kannst Du den `master`-Branch allerdings mit dem Kommando `git merge master` mit Deinem Zweig kombinieren. Oder Du wartest, bis Du den

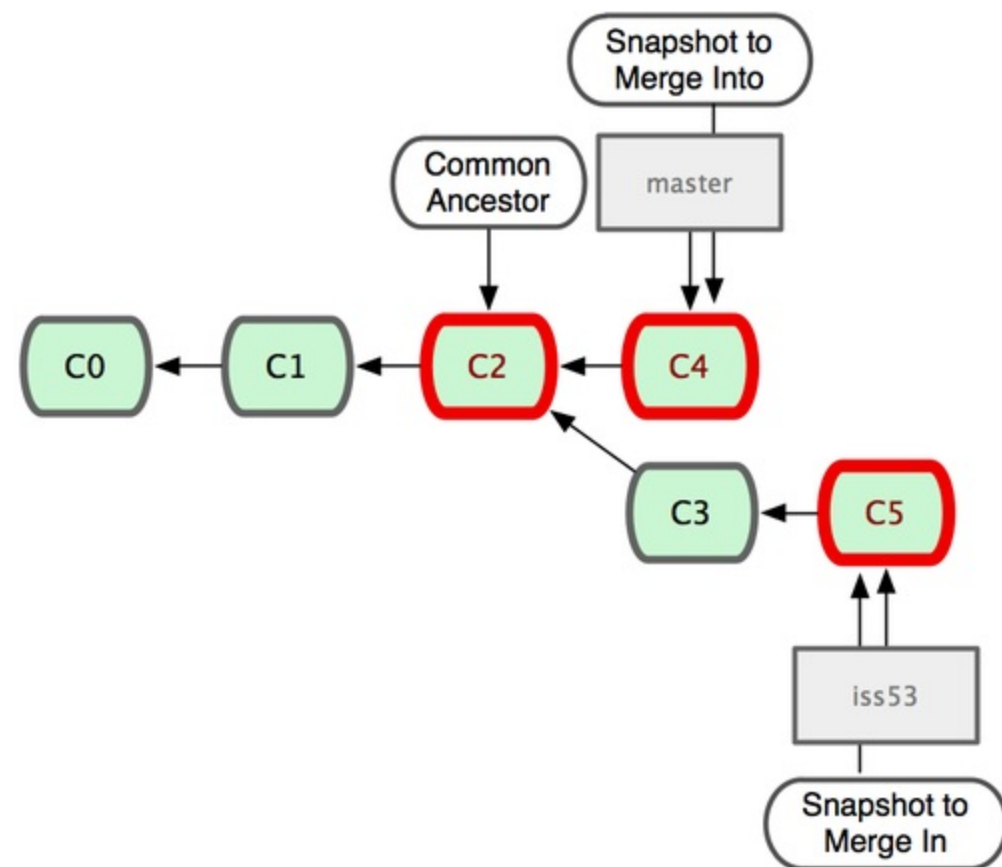
iss53-Branch später in den Master-Zweig zurückführst.

Die Grundlagen des Zusammenführens (Mergen)

Angenommen Du entscheidest dich, dass Deine Arbeit an issue #53 getan ist und Du diese mit der master Branch zusammenführen möchtest. Das passiert, indem Du ein merge in die iss53 Branch machst, ähnlich dem merge mit der hotfix Branch von vorhin. Alles was Du machen musst, ist ein checkout der Branch, in die Du das merge machen willst und das Ausführen des Kommandos `git merge`:

```
$ git checkout master
$ git merge iss53
Merge made by recursive.
 README |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

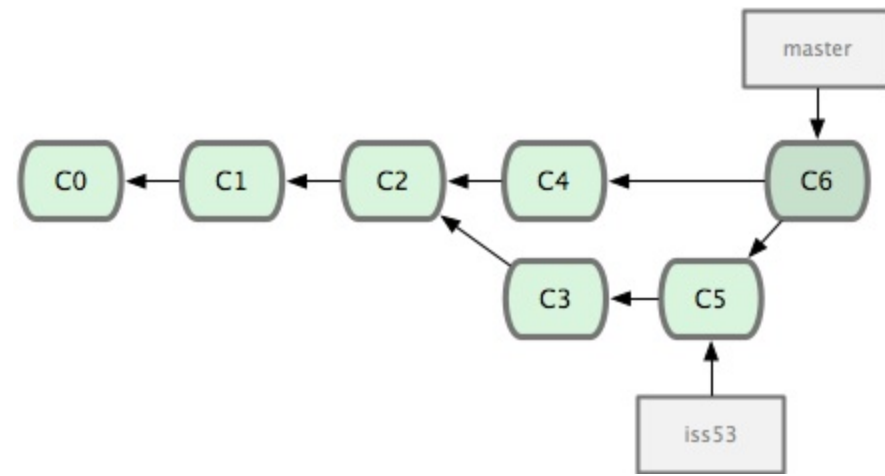
Das sieht ein bisschen anders aus als das hotfix merge von vorhin. Hier läuft Deine Entwicklungshistorie auseinander. Ein commit auf Deine Arbeits-Branch ist kein direkter Nachfolger der Branch in die Du das merge gemacht hast, Git hat da einiges zu tun, es macht einen 3-Wege merge: es geht von den beiden snapshots der Branches und dem allgemeinen Nachfolger der beiden aus. Abbildung 3-16 zeigt die drei snapshots, die Git in diesem Fall für das merge verwendet.



Anstatt einfach den 'pointer' weiterzubewegen, erstellt Git einen neuen snapshot, der aus dem 3-Wege 'merge' resultiert und erzeugt einen neuen 'commit', der darauf verweist (siehe Abbildung 3-17). Dies wird auch als 'merge commit' bezeichnet und ist ein Spezialfall, weil es mehr als nur ein

Elternteil hat.

Es ist wichtig herauszustellen, dass Git den besten Nachfolger für die 'merge' Basis ermittelt, denn hierin unterscheidet es sich von CVS und Subversion (vor Version 1.5), wo der Entwickler die 'merge' Basis selbst ermitteln muss. Damit wird das Zusammenführen in Git um einiges leichter, als in anderen Systemen.



Jetzt da wir die Arbeit zusammengeführt haben, ist der `iss53`-Branch nicht mehr notwendig. Du kannst ihn löschen und das Ticket im Ticket-Tracking-System schliessen.

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

Grundlegende Merge-Konflikte

Gelegentlich verläuft der Prozess nicht ganz so glatt. Wenn Du an den selben Stellen in den selben Dateien unterschiedlicher Branches etwas geändert hast, kann Git diese nicht sauber zusammenführen. Wenn Dein Fix an 'issue #53' die selbe Stelle in einer Datei verändert hat, die Du auch mit `hotfix` angefasst hast, wirst Du einen 'merge'-Konflikt erhalten, der ungefähr so aussehen könnte:

```
$ 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 hat hier keinen 'merge commit' erstellt. Es hat den Prozess gestoppt, damit Du den Konflikt beseitigen kannst. Wenn Du sehen willst, welche Dateien 'unmerged' aufgrund eines 'merge' Konflikts sind, benutze einfach `git status`:

```
[master*]$ git status
index.html: needs merge
# 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)
#
# unmerged:   index.html
```

#

Alles, was einen 'merge' Konflikt aufweist und nicht gelöst werden konnte, wird als 'unmerged' aufgeführt. Git fügt den betroffenen Dateien Standard-Konfliktlösungsmarker hinzu, sodass Du diese öffnen und den Konflikt manuell lösen kannst. Deine Datei enthält einen Bereich, der so aussehen könnte:

```
<<<<<< 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 heisst, die Version in HEAD (Deines 'master'-Branches, denn der wurde per 'checkout' aktiviert als Du das 'merge' gemacht hast) 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, musst Du Dich entweder für einen der beiden Teile entscheiden oder Du ersetzt den Teil komplett:

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

Diese Lösung hat von beiden Teilen etwas und ich habe die Zeilen mit <<<<<<, =====, und >>>>>> komplett gelöscht. Nachdem Du alle problematischen Bereiche, in allen durch den Konflikt betroffenen Dateien, beseitigt hast, führe einfach `git add` für alle betroffenen Dateien aus und markieren sie damit als bereinigt. Dieses 'staging' der Dateien markiert sie für Git als bereinigt. Wenn Du ein grafischen Tool zur Bereinigung benutzen willst, dann verwende `git mergetool`. Das welches ein passendes grafisches 'merge'-Tool startet und Dich durch die Konfliktbereiche führt:

```
$ git mergetool
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff
Merging the files: index.html
```

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

Wenn Du ein anderes Tool anstelle des Standardwerkzeug für ein 'merge' verwenden möchtest (Git verwendet in meinem Fall `opendiff`, da ich auf einem Mac arbeite), dann kannst Du alle unterstützten Werkzeuge oben – neben „merge tool candidates“ – aufgelistet sehen. Tippe einfach den Namen Deines gewünschten Werkzeugs ein. In Kapitel 7 besprechen wir, wie Du diesen Standardwert in Deiner Umgebung dauerhaft ändern kannst.

Wenn Du das 'merge' Werkzeug beendest, fragt Dich Git, ob das Zusammenführen erfolgreich

war. Wenn Du mit 'Ja' antwortest, wird das Skript diese Dateien als gelöst markieren.

Du kannst `git status` erneut ausführen, um zu sehen, ob alle Konflikte gelöst sind:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#   modified:   index.html
#
```

Wenn Du zufrieden bist und Du geprüft hast, dass alle Konflikte beseitigt wurden, kannst Du `git commit` ausführen um den 'merge commit' abzuschliessen. Die Standardbeschreibung für diese Art 'commit' sieht wie folgt 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.
#
```

Wenn Du glaubst für zukünftige Betrachter des Commits könnte interessant sein warum Du getan hast, was Du getan hast, dann kannst Du der Commit-Beschreibung noch zusätzliche Informationen hinzufügen – sofern das nicht trivial erscheint.

Branch Management

Du weißt jetzt, wie Du Branches erstellen, mergen und löschen kannst. Wir schauen uns jetzt noch ein paar recht nützliche Tools für die Arbeit mit Branches an.

Das Kommando `git branch` kann mehr, als nur Branches zu erstellen oder zu löschen. Wenn Du es ohne weitere Argumente ausführst, wird es Dir eine Liste mit Deinen aktuellen Branches anzeigen:

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

Das `*` vor dem `master`-Branch bedeutet, dass dies der gerade ausgecheckte Branch ist. Wenn Du also jetzt einen Commit erzeugst, wird dieser in den `master`-Branch gehen. Du kannst Dir mit `git branch -v` ganz schnell für jeden Branch den letzten Commit anzeigen lassen:

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

Mit einer weiteren nützlichen Option kannst Du herausfinden, in welchem Zustand Deine Branches sind: welche der Branches wurden bereits in den aktuellen Branch gemergt und welche wurden es nicht. Für diesen Zweck gibt es in Git die Optionen `--merged` und `--no-merged`. Um herauszufinden, welche Branches schon in den aktuell ausgecheckten gemergt wurden, kannst Du einfach `git branch --merged` aufrufen:

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

Da Du den Branch `iss53` schon gemergt hast, siehst Du ihn in dieser Liste. Alle Branches in dieser Liste, welchen kein `*` voransteht, können ohne Datenverlust mit `git branch -d` gelöscht werden, da sie ja bereits gemergt wurden.

Um alle Branches zu sehen, welche noch nicht gemergte Änderungen enthalten, kannst Du `git branch --no-merged` aufrufen:

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

Die Liste zeigt Dir den anderen Branch. Er enthält Arbeit, die noch nicht gemergt wurde. Der Versuch, den Branch mit `git branch -d` zu löschen schlägt fehl:

```
$ git branch -d testing
error: The branch 'testing' is not an ancestor of your current HEAD.
If you are sure you want to delete it, run 'git branch -D testing'.
```

Wenn Du den Branch trotzdem löschen willst – und damit alle Änderungen dieses Branches verlieren – kannst Du das mit `git branch -D testing` machen.

Branching Workflows

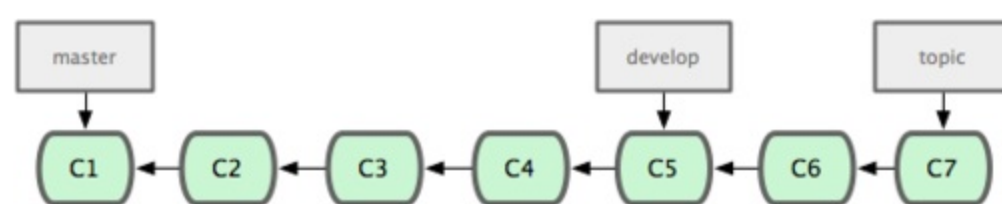
Jetzt da Du die Grundlagen von 'branching' und 'merging' kennst, fragst Du Dich sicher, was Du damit anfangen kannst. In diesem Abschnitt werden wir uns typische Workflows anschauen, die dieses leichtgewichtige 'branching' möglich macht. Und Du kannst dann entscheiden, ob Du es in Deinem eigenen Entwicklungszyklus verwenden willst.

Langfristige Branches

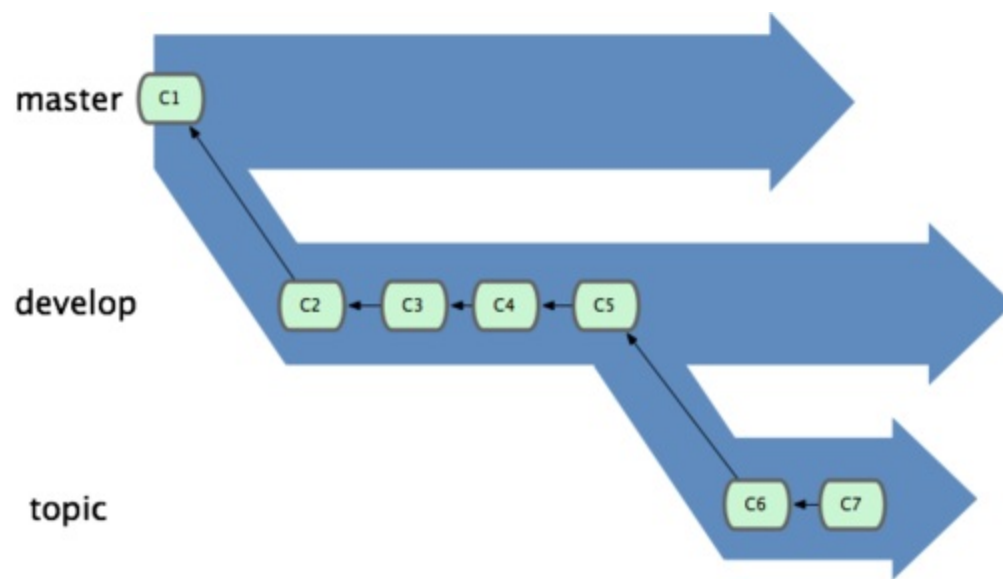
Da Git das einfache 3-Wege-'merge' verwendet, ist häufiges Zusammenführen von einer Branch in eine andere über einen langen Zeitraum generell einfach zu bewerkstelligen. Das heißt, Du kannst mehrere Branches haben, die alle offen sind und auf unterschiedlichen Ebenen Deines Entwicklungszyklus Verwendung finden, und diese regelmäßig ineinander zusammenführen.

Viele Git-Entwickler verfolgen mit ihrem Workflow den Ansatz, nur den stabilen Code in dem master-Branch zu halten – möglicherweise auch nur Code, der released wurde oder werden kann. Sie betreiben parallel einen anderen Branch zum Arbeiten oder Testen. Wenn dieser parallele Zweig einen stabilen Status erreicht, kann er mit dem master-Branch zusammengeführt werden. Dies findet bei themenbezogenen Branches (kurzfristigen Branches, wie der zuvor genannte `iss53`-Branch) Anwendung, um sicherzustellen, dass dieser die Tests besteht und keine Fehler verursacht.

In Realität reden wir über sich bewegende Zeiger, die den Commit-Verlauf weiterwandern. Die stabilen Branches liegen unten und die bleeding-edge Branches weiter oben in der Zeitlinie (siehe Abbildung 3-18).



Es ist leichter sich die verschiedenen Branches als Arbeitsdepots vorzustellen, in denen Sätze von Commits in stabilere Depots aufsteigen, sobald sie ausreichend getestet wurden (siehe Abbildung 3-19).



Das lässt sich für beliebig viele Stabilitätsabstufungen umsetzen. Manche größeren Projekte haben auch einen `proposed` (Vorgeschlagen) oder `pu` (proposed updates – vorgeschlagene Updates) Zweig mit Branches die vielleicht noch nicht bereit sind in den `next`- oder `master`-Branch integriert zu werden. Die Idee dahinter ist, dass Deine Branches verschiedene Stabilitätsabstufungen repräsentieren. Sobald sie eine stabilere Stufe erreichen, werden sie in den nächsthöheren Branch vereinigt.

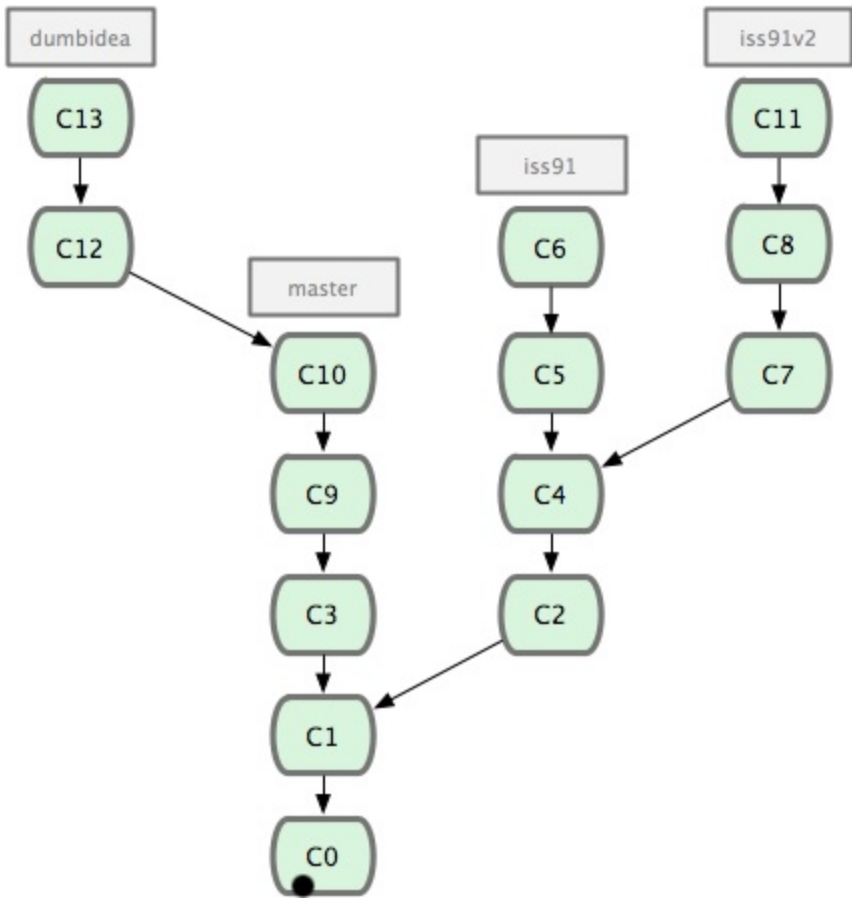
Nochmal, langfristig verschiedene Branches parallel laufen zu lassen ist nicht notwendig, aber oft hilfreich. Insbesondere wenn man es mit sehr großen oder komplexen Projekten zu tun hat.

Themen-Branches

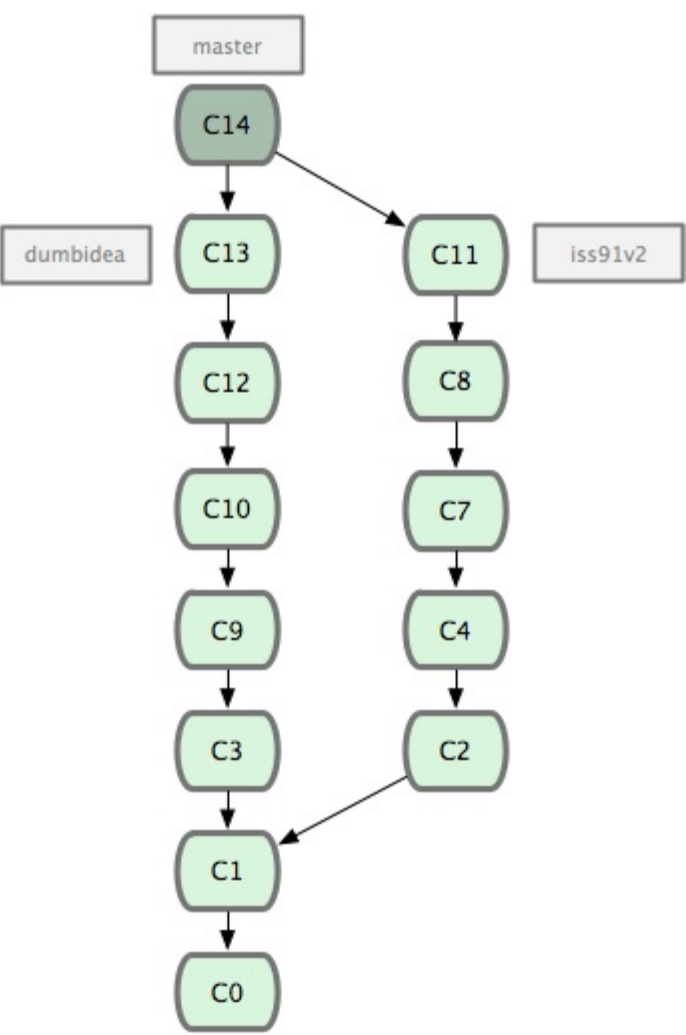
Themen-Branches sind in jedem Projekt nützlich, egal bei welcher Größe. Ein Themen-Branch ist ein kurzlebiger Zweig der für eine spezielle Aufgabe oder ähnliche Arbeiten erstellt und benutzt wird. Das ist vielleicht etwas was Du noch nie zuvor mit einem Versionierungssystem gemacht hast, weil es normalerweise zu aufwändig und mühsam ist Branches zu erstellen und zusammenzuführen. Mit Git ist es allerdings vollkommen geläufig mehrmals am Tag Branches zu erstellen, an ihnen zu arbeiten, sie zusammenzuführen und sie anschließend wieder zu löschen.

Du hast das im letzten Abschnitt an den von Dir erstellten `iss53`- und `hotfix`-Branches gesehen. Du hast mehrere Commits auf sie angewendet und sie unmittelbar nach Zusammenführung mit Deinem Hauptzweig gelöscht. Diese Technik erlaubt es Dir schnell und vollständig den Kontext zu wechseln. Da Deine Arbeit in verschiedene Depots aufgeteilt ist, in denen alle Änderungen unter die Thematik dieses Branches fallen, ist es leichter nachzuvollziehen was bei Code-Überprüfungen und ähnlichem geschehen ist.

Stell Dir vor, Du arbeitest ein bisschen (in `master`), erstellst mal eben einen Branch für einen Fehler (`iss91`), arbeitest an dem für eine Weile, erstellst einen zweiten Branch um eine andere Problemlösung für den selben Fehler auszuprobieren (`iss91v2`), wechselst zurück zu Deinem `MASTER`-Branch, arbeitest dort ein bisschen und machst dann einen neuen Branch für etwas, wovon Du nicht weißt ob's eine gute Idee ist (`dumbidea`-Branch). Dein Commit-Verlauf wird wie in Abbildung 3-20 aussehen.



Nun, sagen wir Du hast Dich entschieden die zweite Lösung des Fehlers (iss91v2) zu bevorzugen, außerdem hast den dumbidea-Branch Deinen Mitarbeitern gezeigt und es hat sich herausgestellt das er genial ist. Du kannst also den ursprünglichen iss91-Branch (unter Verlust der Commits C5 und C6) wegschmeißen und die anderen Beiden vereinen. Dein Verlauf sieht dann aus wie in Abbildung 3-21.



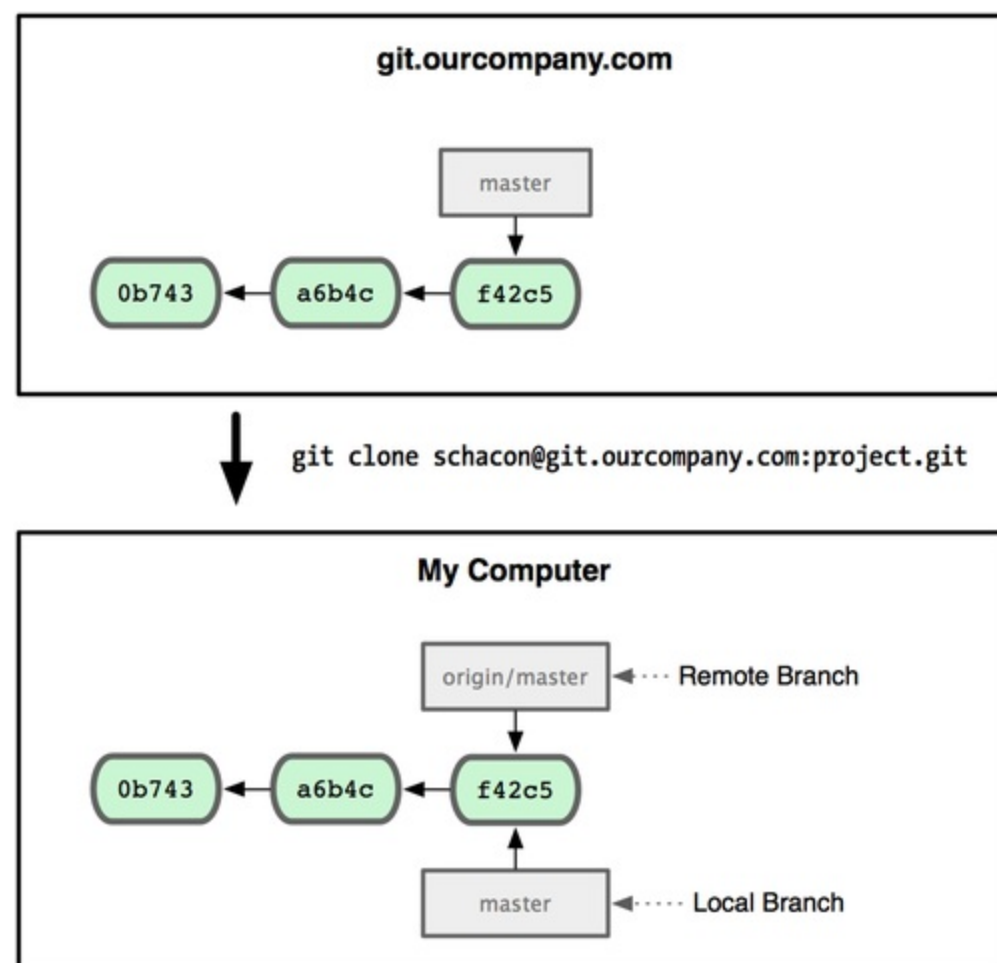
Es ist wichtig sich daran zu erinnern, dass alle diese Branches nur lokal existieren. Wenn Du Verzweigungen schaffst (branchst) und wieder zusammenführst (mergest), findet dies nur in Deinem Git-Repository statt – es findet keine Server-Kommunikation statt.

Externe Branches

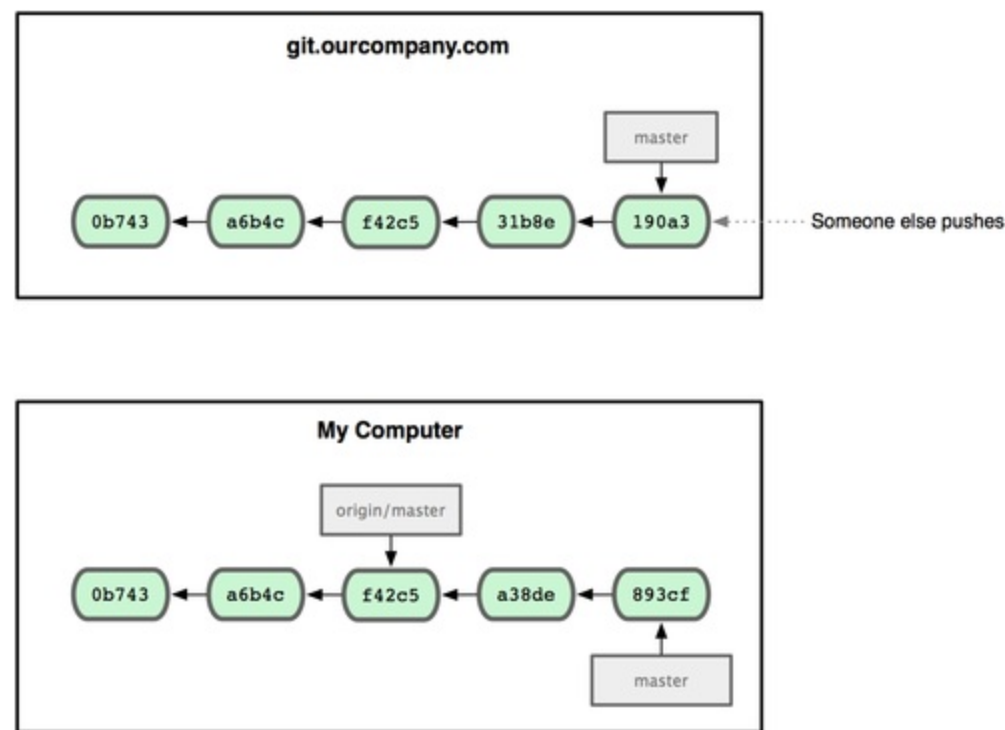
Externe (Remote) Branches sind Referenzen auf den Zustand der Branches in Deinen externen Repositories. Es sind lokale Branches die Du nicht verändern kannst, sie werden automatisch verändert wann immer Du eine Netzwerkoperation durchführst. Externe Branches verhalten sich wie Lesezeichen, um Dich daran zu erinnern an welcher Position sich die Branches in Deinen externen Repositories befanden, als Du Dich zuletzt mit ihnen verbunden hattest.

Externe Branches besitzen die Schreibweise (Repository)/(Branch). Wenn Du beispielsweise wissen möchtest wie der master-Branch in Deinem origin-Repository ausgesehen hat, als Du zuletzt Kontakt mit ihm hattest, dann würdest Du den origin/master-Branch überprüfen. Wenn Du mit einem Mitarbeiter an einer Fehlerbehebung gearbeitet hast, und dieser bereits einen iss53-Branch hochgeladen hat, besitzt Du möglicherweise Deinen eigenen lokalen iss53-Branch. Der Branch auf dem Server würde allerdings auf den Commit von origin/iss53 zeigen.

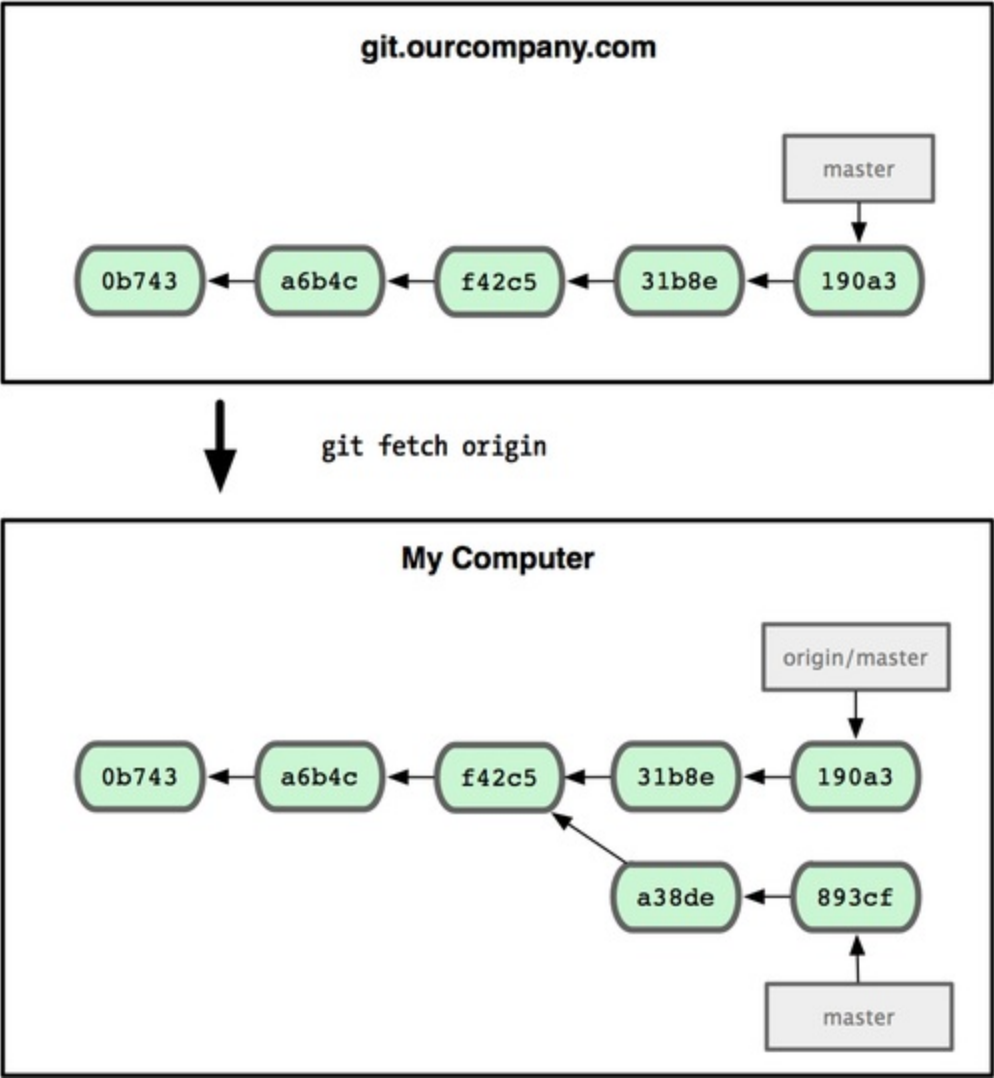
Das kann ein wenig verwirrend sein, lass uns also ein Beispiel betrachten. Nehmen wir an Du hättest in Deinem Netzwerk einen Git-Server mit der Adresse `git.ourcompany.com`. Wenn Du von ihm klonst, nennt Git ihn automatisch `origin` für dich, lädt all seine Daten herunter, erstellt einen Zeiger an die Stelle wo sein master-Branch ist und benennt es lokal `origin/master`; und er ist unveränderbar für dich. Git gibt Dir auch einen eigenen master-Branch mit der gleichen Ausgangsposition wie origins master-Branch, damit Du einen Punkt für den Beginn Deiner Arbeiten hast (siehe Abbildung 3-22).



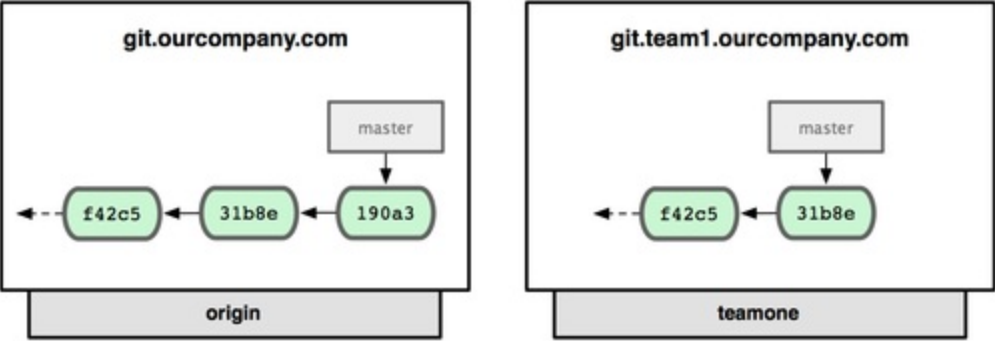
Wenn Du ein wenig an Deinem lokalen master-Branch arbeitest und unterdessen jemand etwas zu `git.ourcompany.com` herauflädt, verändert er damit dessen master-Branch und eure Arbeitsverläufe entwickeln sich unterschiedlich. Indes bewegt sich Dein `origin/master`-Zeiger nicht, solange Du keinen Kontakt mit Deinem `origin`-Server aufnimmst (siehe Abbildung 3-23).



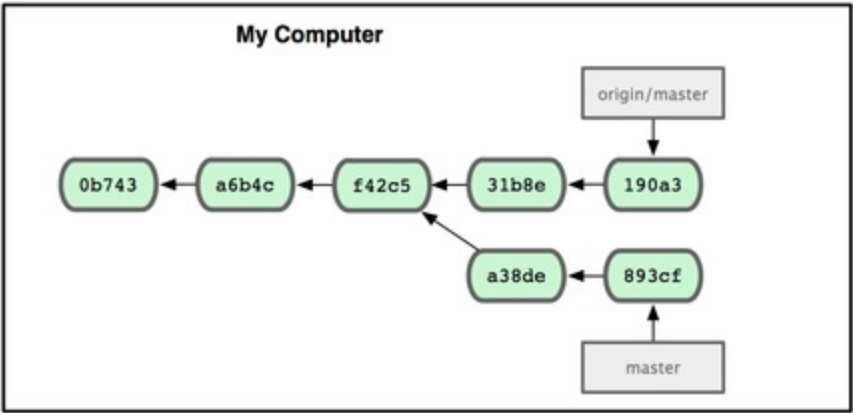
Um Deine Arbeit abzugleichen, führe ein `git fetch origin`-Kommando aus. Das Kommando schlägt nach welcher Server `origin` ist (in diesem Fall `git.ourcompany.com`), holt alle Daten die Dir bisher fehlen und aktualisiert Deine lokale Datenbank, indem es Deinen `origin/master`-Zeiger auf seine neue aktuellere Position bewegt (siehe Abbildung 3-24).



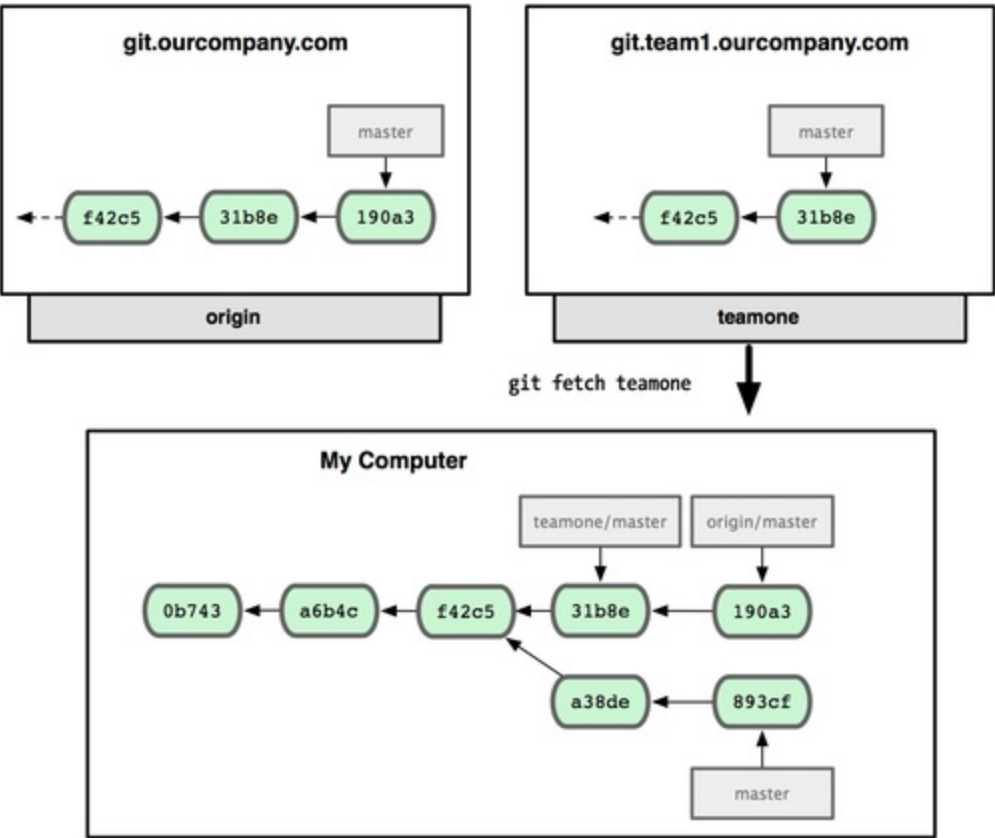
Um zu demonstrieren wie Branches auf verschiedenen Remote-Servern aussehen, stellen wir uns vor, dass Du einen weiteren internen Git-Server besitzt, welcher nur von einem Deiner Sprint-Teams zur Entwicklung genutzt wird. Diesen Server erreichen wir unter `git.team1.ourcompany.com`. Du kannst ihn, mit dem Git-Kommando `git remote add` – wie in Kapitel 2 beschrieben, Deinem derzeitigen Arbeitsprojekt als weiteren Quell-Server hinzufügen. Gib dem Remote-Server den Namen `teamone`, welcher nun als Synonym für die ausgeschriebene Internetadresse dient (siehe Abbildung 3-25).



`git remote add teamone git://git.team1.ourcompany.com`



Nun kannst Du einfach `git fetch teamone` ausführen um alles vom Server zu holen was Du noch nicht hast. Da der Datenbestand auf dem Teamserver ein Teil der Informationen auf Deinem origin-Server ist, holt Git keine Daten, erstellt allerdings einen Remote-Branch namens `teamone/master`, der auf den gleichen Commit wie `teamones master-Branch` zeigt (siehe Abbildung 3-26).



Wenn Du einen Branch mit der Welt teilen möchtest, musst Du ihn auf einen externen Server laden, auf dem Du Schreibrechte besitzt. Deine lokalen Zweige werden nicht automatisch mit den Remote-Servern synchronisiert wenn Du etwas änderst – Du musst die zu veröffentlichenden Branches explizit hochladen (pushen). Auf diesem Weg kannst Du an privaten Zweigen arbeiten die Du nicht veröffentlichen möchtest, und nur die Themen-Branches replizieren an denen Du gemeinsam mit anderen entwickeln möchtest.

Wenn Du einen Zweig namens `serverfix` besitzt, an dem Du mit anderen arbeiten möchtest, dann kannst Du diesen auf dieselbe Weise hochladen wie Deinen ersten Branch. Führe `git push (remote) (branch)` aus:

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

Hierbei handelt es sich um eine Abkürzung. Git erweitert die `serverfix`-Branchbezeichnung 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 externen Server”. Wir werden den `refs/heads/-`Teil in Kapitel 9 noch näher beleuchten, Du kannst ihn aber in der Regel weglassen. Du kannst auch `git push origin serverfix:serverfix` ausführen, was das gleiche bewirkt – es bedeutet “Nimm meinen `serverfix` und mach ihn zum externen `serverfix`”. Du kannst dieses Format auch benutzen um einen lokalen Zweig in einen externen Branch mit anderem Namen zu laden. Wenn Du ihn auf dem externen Server nicht `serverfix` nennen möchtest, könntest Du stattdessen `git push origin serverfix:awesomebranch` ausführen um Deinen lokalen `serverfix`-Branch in den `awesomebranch`-Zweig in Deinem externen Projekt zu laden.

Das nächste Mal wenn einer Deiner Mitarbeiter den aktuellen Status des Git-Projektes vom Server abrufen wird er eine Referenz, auf den externen Branch `origin/serverfix`, unter dem Namen `serverfix` erhalten:

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix      -> origin/serverfix
```

Es ist wichtig festzuhalten, dass Du mit Abrufen eines neuen externen Branches nicht automatisch eine lokale, bearbeitbare Kopie derselben erhältst. Mit anderen Worten, in diesem Fall bekommst Du keinen neuen `serverfix`-Branch – sondern nur einen `origin/serverfix`-Zeiger den Du nicht verändern kannst.

Um diese referenzierte Arbeit mit Deinem derzeitigen Arbeitsbranch zusammenzuführen kannst Du `git merge origin/serverfix` ausführen. Wenn Du allerdings Deine eigene Arbeitskopie des `serverfix`-Branches erstellen möchtest, dann kannst Du diesen auf Grundlage des externen Zweiges erstellen:

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

Dies erstellt Dir einen lokalen bearbeitbaren Branch mit der Grundlage des `origin/serverfix`-Zweiges.

Tracking Branches

Das Auschecken eines lokalen Branches von einem Remote-Branch erzeugt automatisch einen sogenannten *Tracking-Branch*. Tracking Branches sind lokale Branches mit einer direkten Beziehung zu dem Remote-Zweig. Wenn Du Dich in einem Tracking-Branch befindest und `git push` eingibst, weiß Git automatisch zu welchem Server und Repository es pushen soll. Ebenso führt `git pull` in einem dieser Branches dazu, dass alle entfernten Referenzen gefetched und automatisch in den Zweig gemerged werden.

Wenn Du ein Repository klonst, wird automatisch ein `master`-Branch erzeugt, welcher `origin/master` verfolgt. Deshalb können `git push` und `git pull` ohne weitere Argumente aufgerufen werden. Du kannst natürlich auch eigene Tracking-Branches erzeugen – welche die nicht Zweige auf `origin` und dessen `master`-Branch verfolgen. Der einfachste Fall ist das bereits gesehene Beispiel in welchem Du `git checkout -b [branch] [remotename]/[branch]` ausführst. Mit der Git-Version 1.6.2 oder später kannst Du auch die `--track`-Kurzvariante nutzen:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

Um einen lokalen Branch mit einem anderem Namen als der Remote-Branch, kannst Du einfach die erste Varianten mit einem neuen lokalen Branch-Namen:

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

Nun wird Dein lokaler Branch `sf` automatisch push und pull auf `origin/serverfix` durchführen.

Löschen entfernter Branches

Stellen wir uns Du bist fertig mit Deinem Remote-Branch – sagen wir Deine Mitarbeiter und Du, Ihr seid fertig mit einer neuen Funktion und habt sie in den entfernten `master`-Branch (oder in welchem Zweig Ihr sonst den stabilen Code ablegt) gemerged. Du kannst einen Remote-Branch mit der unlogischen Syntax `git push [remotename] :[branch]` löschen. Wenn Du Deinen

serverfix-Branch vom Server löschen möchtest, führe folgendes aus:

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

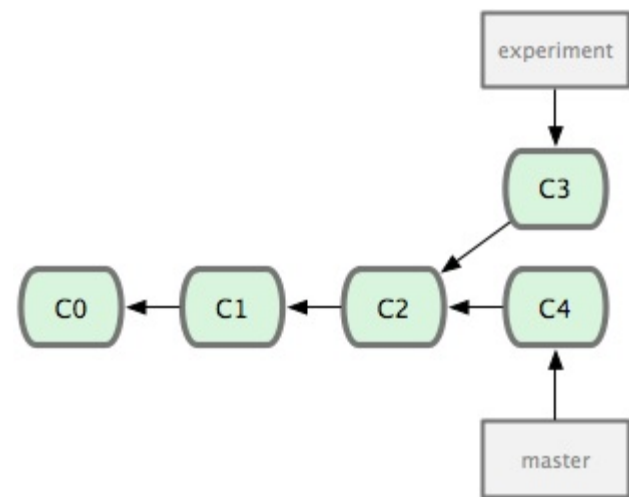
Boom. Kein Zweig mehr auf Deinem Server. Du möchtest Dir diese Seite vielleicht markieren, weil Du dieses Kommando noch benötigen wirst und man leicht dessen Syntax vergisst. Ein Weg sich an dieses Kommando zu erinnern führt über die `git push [remotename] [localbranch]:[remotebranch]`-Syntax, welche wir bereits behandelt haben. Wenn Du den `[localbranch]`-Teil weglässt, dann sagst Du einfach „Nimm nichts von meiner Seite und mach es zu `[remotebranch]`“.

Rebasing

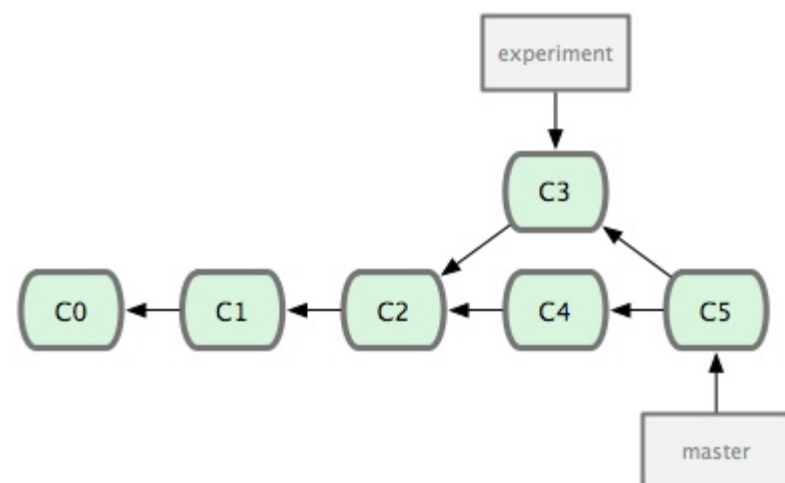
Es gibt in Git zwei Wege um Änderungen von einem Branch in einen anderen zu überführen: das merge und das rebase-Kommando. In diesem Abschnitt wirst Du kennenlernen was Rebasing ist, wie Du es anwendest, warum es ein verdammt abgefahrenes Werkzeug ist und wann Du es lieber nicht einsetzen möchtest.

Der einfache Rebase

Wenn Du zu einem früheren Beispiel aus dem Merge-Kapitel zurückkehrst (siehe Abbildung 3-27), wirst Du sehen, dass Du Deine Arbeit auf zwei unterschiedliche Branches aufgeteilt hast.



Der einfachste Weg um Zweige zusammenzuführen ist, wie bereits behandelt, das merge-Kommando. Es produziert einen Drei-Wege-Merge zwischen den beiden letzten Branch-Zuständen (C3 und C4) und ihrem wahrscheinlichsten Vorgänger (C2). Es produziert seinerseits einen Schnappschuss des Projektes (und einen Commit), wie in Abbildung 3-28 dargestellt.

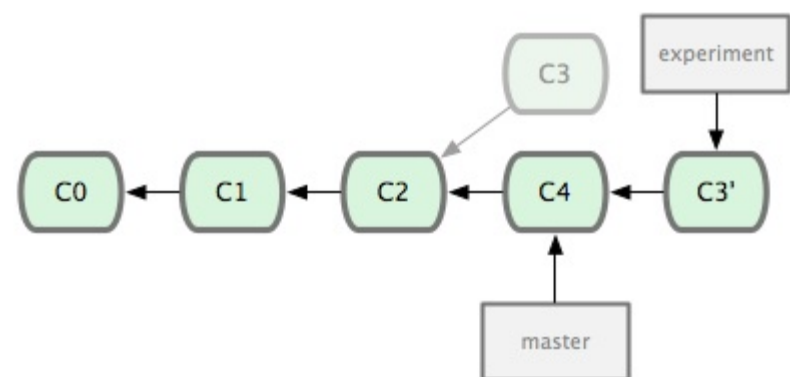


Wie auch immer, es gibt noch einen anderen Weg: Du kannst den Patch der Änderungen – den wir in C3 eingeführt haben – über C4 anwenden. Dieses Vorgehen nennt man in Git *rebasing*. Mit dem rebase-Kommando kannst Du alle Änderungen die auf einem Branch angewendet wurden auf einen anderen Branch erneut anwenden.

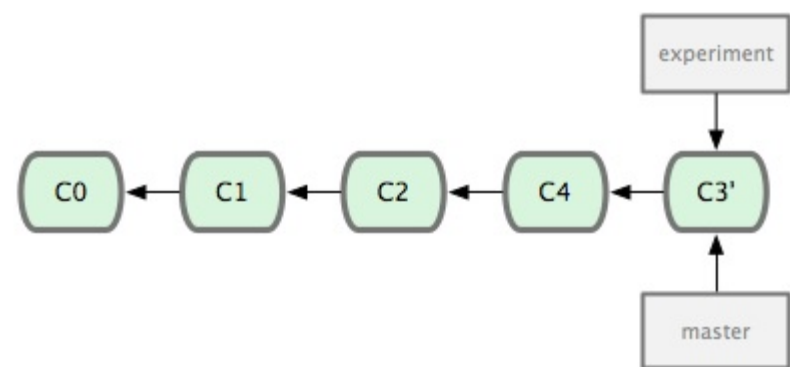
In unserem Beispiel würdest Du folgendes ausführen:

```
$ 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 zu dem gemeinsamen/allgemeinen Vorfahren [gemeinsamer Vorfahr oder der Ursprung der beiden Branches?] der beiden Branches (des Zweiges auf dem Du arbeitest und des Zweiges auf den Du *rebasen* möchtest) geht, die Differenzen jedes Commits des aktuellen Branches ermittelt und temporär in einer Datei ablegt. Danach wird der aktuelle Branch auf den Schnittpunkt der beiden Zweige zurückgesetzt und alle zwischengespeicherte Commits nacheinander auf den Zielbranch angewendet. Die Abbildung 3-29 bildet diesen Prozess ab.



An diesem Punkt kannst Du zurück zum Master-Branch wechseln und einen fast-forward Merge durchführen (siehe Abbildung 3-30).



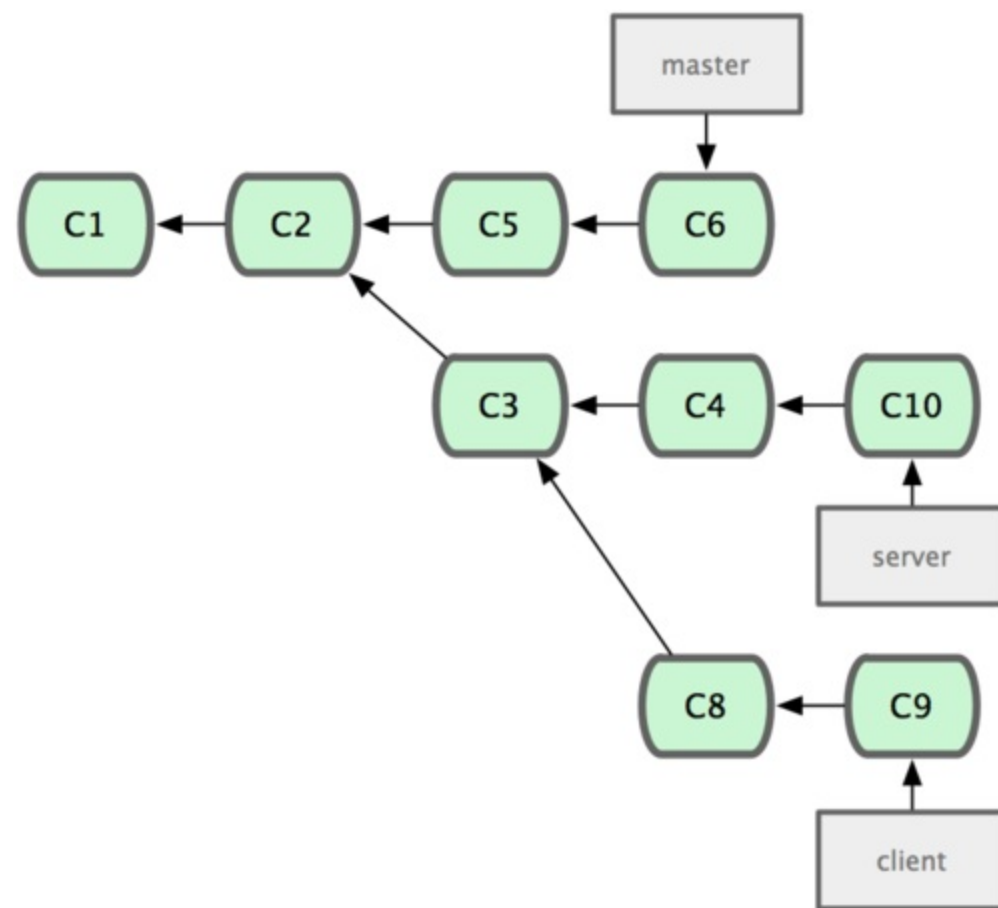
Nun ist der Schnappschuss, auf den C3' zeigt, exakt der gleiche, wie der auf den C5 in dem Merge-Beispiel gezeigt hat. Bei dieser Zusammenführung entsteht kein unterschiedliches Produkt, durch Rebasing entsteht allerdings ein sauberer Verlauf. Bei genauerer Betrachtung der Historie entpuppt sich der Rebased-Branch als linearer Verlauf – es scheint als sei die ganze Arbeit in einer Serie entstanden, auch wenn sie in Wirklichkeit parallel stattfand.

Du wirst das häufig anwenden um sicherzustellen, dass sich Deine Commits sauber in einen Remote-Branch integrieren – möglicherweise in einem Projekt bei dem Du Dich beteiligen möchtest, Du jedoch nicht der Verantwortliche bist. In diesem Fall würdest Du Deine Arbeiten in einem eigenen Branch erledigen und im Anschluss Deine Änderungen auf `origin/master` rebasen. Dann hätte der Verantwortliche nämlich keinen Aufwand mit der Integration – nur einen Fast-Forward oder eine saubere Integration (= Rebase?).

Beachte, dass der Schnappschuss nach dem letzten Commit, ob es der letzte der Rebase-Commits nach einem Rebase oder der finale Merge-Commit nach einem Merge ist, exakt gleich ist. Sie unterscheiden sich nur in ihrem Verlauf. Rebasing wiederholt einfach die Änderungen einer Arbeitslinie auf einer anderen, in der Reihenfolge in der sie entstanden sind. Im Gegensatz hierzu nimmt Merging die beiden Endpunkte der Arbeitslinien und führt diese zusammen.

Mehr interessante Rebases

Du kannst Deinen Rebase auch auf einem anderen Branch als dem Rebase-Branch anwenden lassen. Nimm zum Beispiel den Verlauf in Abbildung 3-31. Du hattest einen Themen-Branch (`server`) eröffnet um ein paar serverseitige Funktionalitäten zu Deinem Projekt hinzuzufügen und einen Commit gemacht. Dann hast Du einen weiteren Branch abgezweigt um clientseitige Änderungen (`client`) vorzunehmen und dort ein paarmal committed. Zum Schluss hast Du wieder zu Deinem Server-Branch gewechselt und ein paar weitere Commits gebaut.

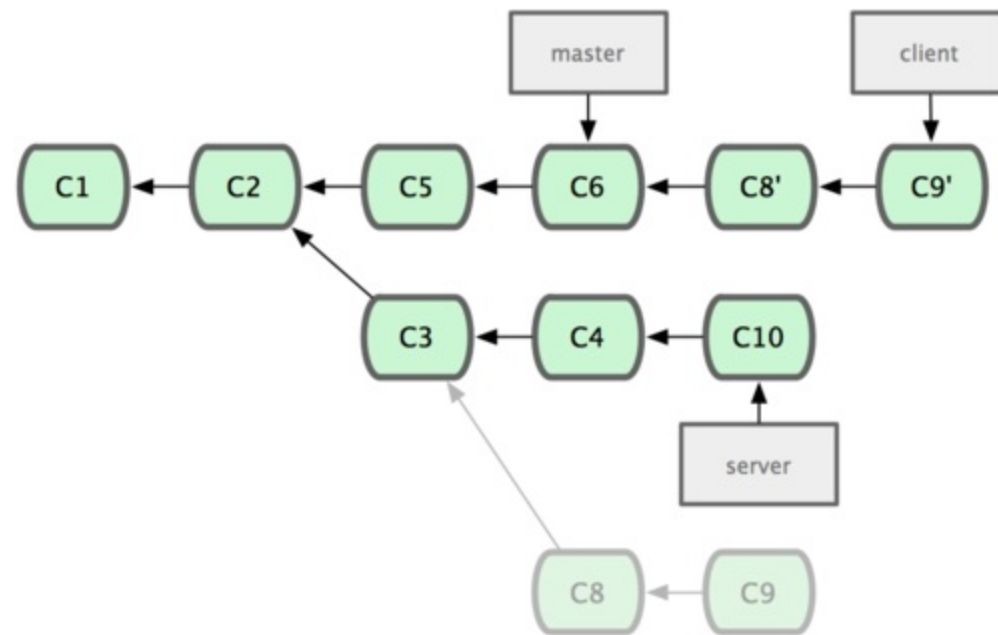


Stell Dir vor, Du entscheidest Dich Deine clientseitigen Änderungen für einen Release in die Hauptlinie zu mergen, die serverseitigen Änderungen möchtest Du aber noch zurückhalten bis sie besser getestet wurden. Du kannst einfach die Änderungen am Client, die den Server nicht betreffen, (C8 und C9) mit der `--onto`-Option von `git rebase` erneut auf den Master-Branch anwenden:

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

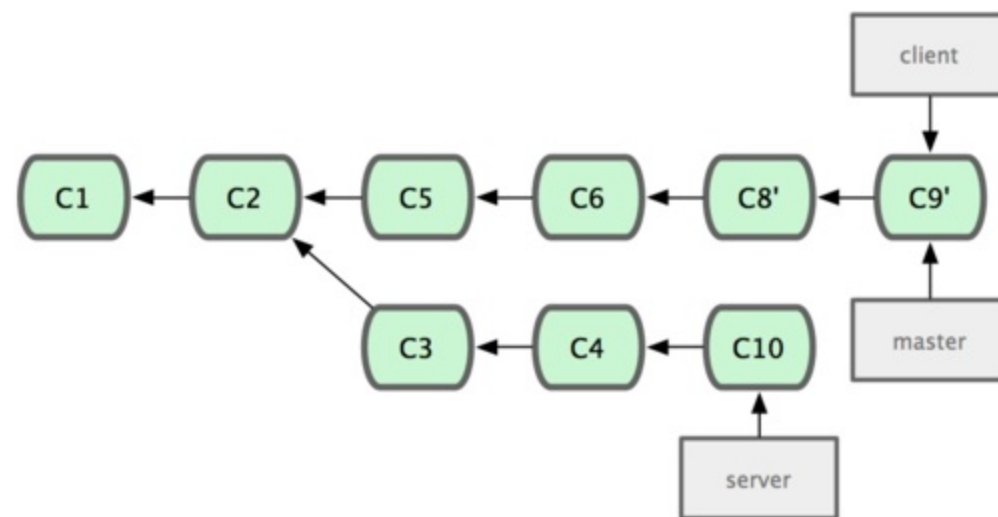
Das bedeutet einfach "Checke den Client-Branch aus, finde die Patches heraus die auf dem gemeinsamen Vorfahr der `client`- und `server`-Branches basieren und wende sie erneut auf dem

master-Branch an.” Das ist ein bisschen komplex, aber das Ergebnis – wie in Abbildung 3-32 – ist richtig cool.



Jetzt kannst Du Deinen Master-Branch fast-forwarden (siehe Abbildung 3-33):

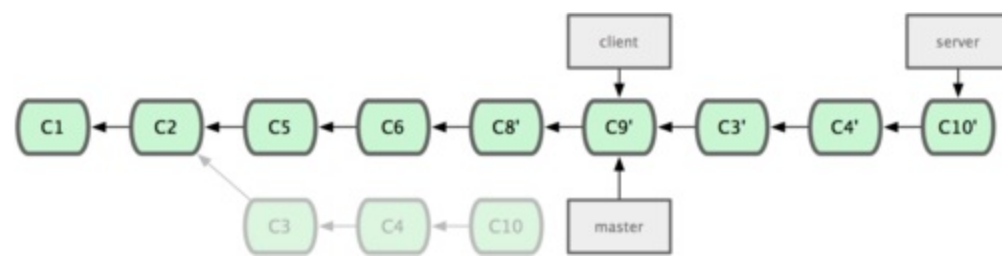
```
$ git checkout master  
$ git merge client
```



Lass uns annehmen, Du entscheidest Dich Deinen Server-Branch ebenfalls einzupflegen. Du kannst den Server-Branch auf den Master-Branch rebasen ohne diesen vorher auschecken zu müssen, indem Du das Kommando `git rebase [Basis-Branch] [Themen-Branch]` ausführst. Es macht für Dich den Checkout des Themen-Branche (in diesem Fall `server`) und wiederholt ihn auf dem Basis-Branch (`master`):

```
$ git rebase master server
```

Das wiederholt Deine `server`-Arbeit auf der Basis der `server`-Arbeit, wie in Abbildung 3-34 ersichtlich.



Dann kannst Du den Basis-Branch (master) fast-forwarden:

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

Du kannst den client- und server-Branch nun entfernen, da Du die ganze Arbeit bereits integriert wurde und Sie nicht mehr benötigst. Du hinterlässt den Verlauf für den ganzen Prozess wie in Abbildung 3-35:

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

Insert 18333fig0335.png Abbildung 3-35: 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:

Rebase keine Commits die Du in ein öffentliches Repository hochgeladen hast.

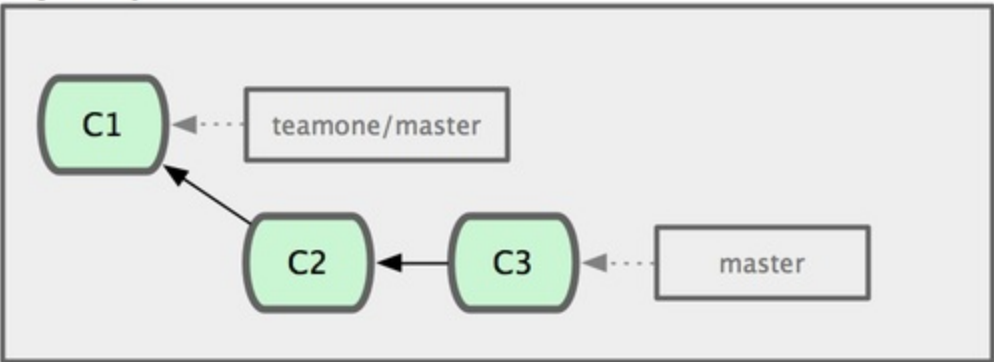
Wenn Du diesem Ratschlag folgst ist alles in Ordnung. Falls nicht, werden die Leute Dich hassen und Du wirst von Deinen Freunden und Deiner Familie verachtet.

Wenn Du Zeug rebased, hebst Du bestehende Commits auf und erstellst stattdessen welche, die zwar ähnlich aber unterschiedlich sind. Wenn Du Commits irgendwohin hochlädst und andere ziehen sich diese herunter und nehmen sie als Grundlage für ihre Arbeit, dann müssen Deine Mitwirkenden ihre Arbeit jedesmal re-mergen, sobald Du Deine Commits mit einem `git rebase` überschreibst und verteilst. Und richtig chaotisch wird's wenn Du versuchst deren Arbeit in Deine Commits zu integrieren.

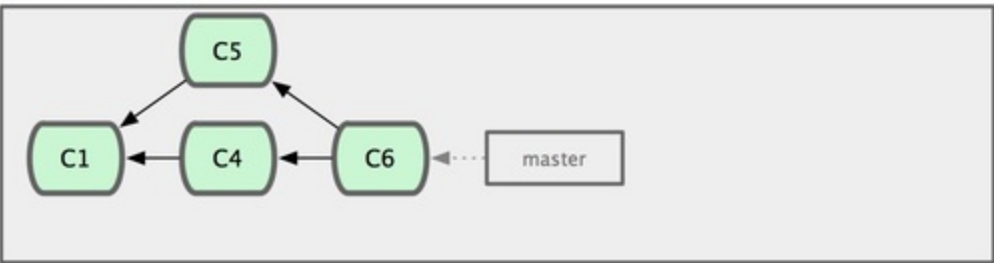
Lass uns mal ein Beispiel betrachten wie das Rebasen veröffentlichter Arbeit Probleme verursachen kann. Angenommen Du klonst von einem zentralen Server und werkelst ein bisschen daran rum. Dein Commit-Verlauf sieht wie in Abbildung 3-36 aus.



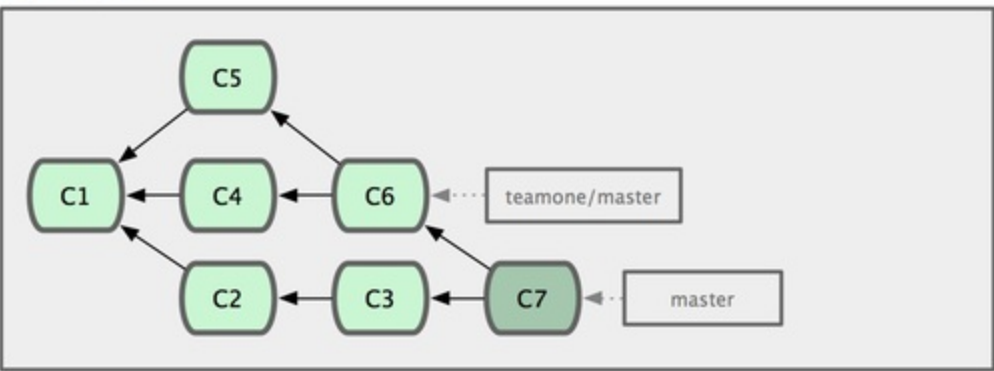
My Computer



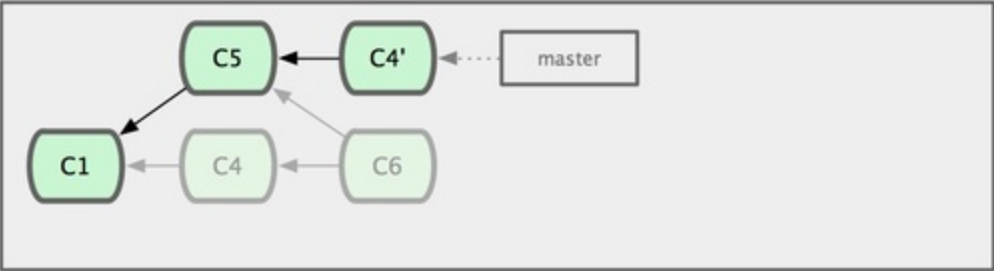
Ein anderer arbeitet unterdessen weiter, macht einen Merge und lädt seine Arbeit auf den zentralen Server. Du fetchst die Änderungen und mergest den neuen Remote-Branch in Deine Arbeit, sodass Dein Verlauf wie in Abbildung 3-37 aussieht.



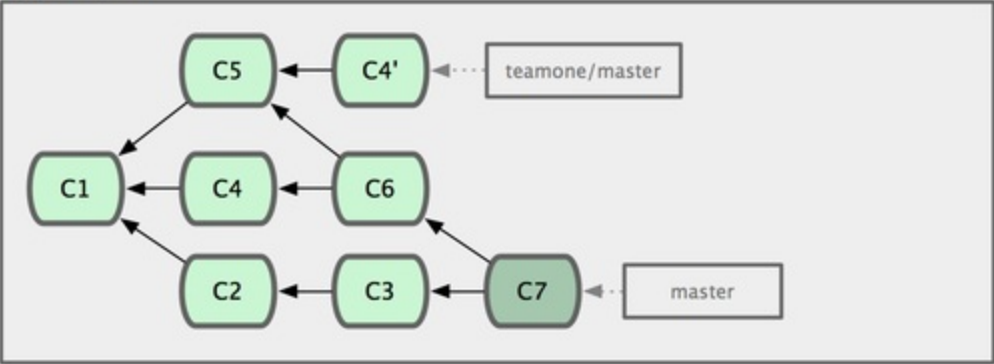
My Computer



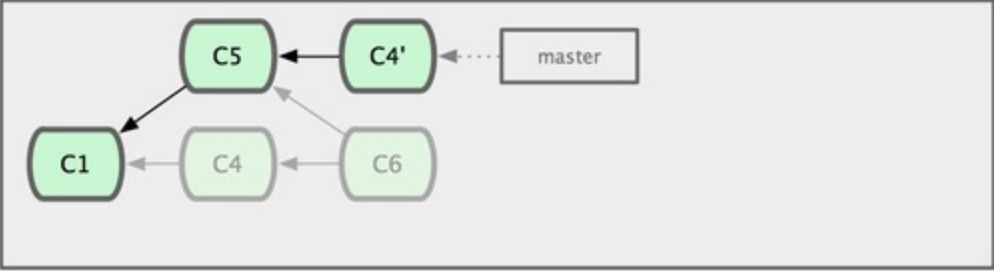
Als nächstes entscheidet sich die Person, welche den Merge hochgeladen hat diesen rückgängig zu machen und stattdessen die Commits zu rebasen. Sie macht einen `git push --force` um den Verlauf auf dem Server zu überschreiben. Du lädst Dir das Ganze dann mit den neuen Commits herunter.



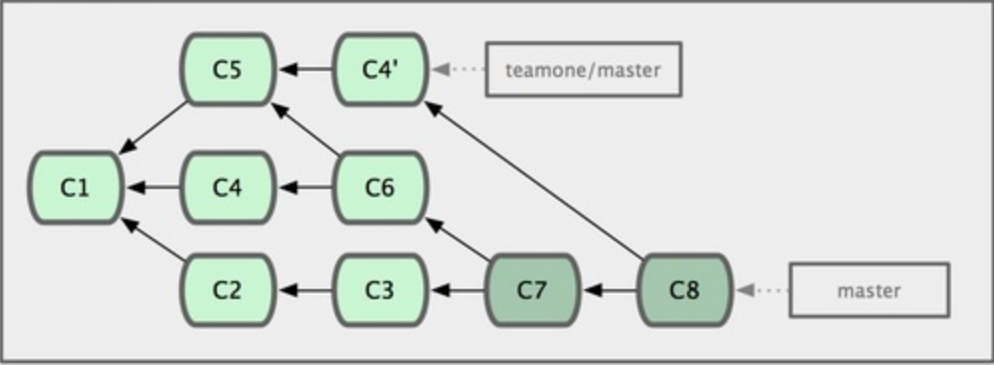
My Computer



Nun musst Du seine Arbeit erneut in Deine Arbeitslinie mergen, obwohl Du das bereits einmal gemacht hast. Rebasing ändert die SHA-1-Hashes der Commits, weshalb sie für Git wie neue Commits aussehen. In Wirklichkeit hast Du die C4-Arbeit bereits in Deinem Verlauf (siehe Abbildung 3-39).



My Computer



Irgendwann musst Du seine Arbeit einmergen, damit Du auch zukünftig mit dem anderen Entwickler zusammenarbeiten kannst. Danach wird Dein Commit-Verlauf sowohl den C4 als auch den C4'-Commit enthalten, welche zwar verschiedene SHA-1-Hashes besitzen aber die gleichen Änderungen und die gleiche Commit-Beschreibung enthalten. Wenn Du so einen Verlauf mit `git log` betrachtest, wirst Du immer zwei Commits des gleichen Autors, zur gleichen Zeit und mit der gleichen Commit-Nachricht sehen. Was ganz schön verwirrend ist. Wenn Du diesen Verlauf

außerdem auf den Server hochlädst, wirst Du dort alle rebasierten Commits einführen, was auch noch andere verwirren kann.

Wenn Du rebasing als Weg behandelst um aufzuräumen und mit Commits zu arbeiten, bevor Du sie hochlädst und wenn Du nur Commits rebased, die noch nie publiziert wurden, dann fährst Du goldrichtig. Wenn Du Commits rebased die bereits veröffentlicht wurden und Leute vielleicht schon ihre Arbeit darauf aufgebaut haben, dann bist Du vielleicht für frustrierenden Ärger verantwortlich.

Zusammenfassung

Wir haben einfaches Branching und Merging mit Git behandelt. Du solltest nun gut damit zurecht kommen Branches zu erstellen, zwischen Branches zu wechseln und lokale Branches mit einem Merge zusammenzuführen. Ausserdem solltest Du in der Lage sein Deine Branches zu veröffentlichen indem Du sie auf einen zentralen Server lädst, mit anderen auf öffentlichen Branches zusammenzuarbeiten und Deine Branches zu rebasen bevor sie veröffentlicht werden.

Git auf dem Server

Zum jetzigen Zeitpunkt solltest Du in der Lage sein, die am häufigsten wiederkehrenden Aufgaben mit Git zu lösen. Um eine Zusammenarbeit zu ermöglichen solltest Du jedoch darüber nachdenken, ein externes Repository zur Verfügung stellen. Wenngleich es technisch möglich ist, direkt mit Repositories Anderer zu arbeiten und Änderungen dorthin zu pushen oder von dort zu holen, ist dieses Vorgehen verpöht, da es sehr leicht die Arbeit Anderer durcheinander zu bringen. Wenn man nicht vorsichtig ist, verliert man schnell den Überblick darüber wer woran arbeitet. Des weiteren erfordert der direkte Umgang mit Git Repositories, dass diese permanent verfügbar sind. Ein Repository auf dem eigenen Computer ist nur dann für alle Mitentwickler erreichbar, wenn der Computer läuft. Es macht deshalb Sinn, ein zentrales und zuverlässig erreichbares Repository einzurichten. Wir werden dieses Repository im Folgenden als „Git Server“ bezeichnen. Es sollte jedoch schnell klar werden, dass nur minimale Ressourcen notwendig sind, um Git Repositories zu hosten. In den seltensten Fällen ist ein dedizierter Server dafür notwendig.

Einen Git Server zu betreiben ist einfach. Die erste Entscheidung, die zu treffen ist, ist die des zu verwendenden Protokolls zur Kommunikation mit dem Server. Der erste Teil dieses Kapitels wird deshalb die zur Verfügung stehenden Protokolle und ihre Vor- und Nachteile beschreiben. Darüber hinaus werden einige typische Konfigurationen zum Betreiben eines Git Servers vorgestellt. Falls keine Bedenken bestehen, Code von externen Anbietern hosten zu lassen, werden zuletzt ein paar Optionen für gehostete Git Repositories aufgezeigt. Dies erspart die Mehrarbeit der Einrichtung und Wartung eines eigenen Git Servers.

Wenn Du kein Interesse am Betreiben eines eigenen Servers hast, kannst Du zum letzten Absatz des Kapitels springen, um ein paar Möglichkeiten zum Einrichten eines gehosteten Accounts zu sehen. Im nächsten Kapitel diskutieren wir verschiedene Vor- und Nachteile vom Arbeiten in einer verteilten Quellcode-Kontroll-Umgebung.

Ein externes Repository ist im Allgemeinen ein *einfaches Repository* – ein Git Repository ohne Arbeitsverzeichnis. Weil das Repository nur als Zusammenarbeitspunkt genutzt wird, gibt es keinen Grund, einen Schnappschuss ausgecheckt auf der Festplatte zu haben; es sind nur die Git Daten. Mit einfachen Begriffen, ein einfaches Repository ist der Inhalt von Deinem `.git` Verzeichnis in Deinem Projekt und nichts anderes.

Die Protokolle

Git kann vier wichtige Netzwerk Protokolle zum Datentransfer benutzen: Lokal, Secure Shell (SSH), Git und HTTP. Hier wollen wir diskutieren, was diese Protokolle sind und unter welchen grundlegenden Gegebenheiten Du sie benutzen möchtest (oder auch nicht).

Es ist wichtig zu beachten, dass alle Protokolle mit Ausnahme von HTTP eine funktionierende Git Installation auf dem Server benötigen.

Lokales Protokoll

Am einfachsten ist das *Lokale Protokoll*, wobei das externe Repository in einem anderen Verzeichnis auf der Festplatte ist. Das wird oft genutzt, wenn jeder aus Deinem Team Zugriff zu einem gemeinsamen Dateisystem hat, zum Beispiel ein eingebundenes NFS, oder im unwahrscheinlicheren Fall jeder loggt sich auf bei dem gleichen Computer ein. Das letztere ist nicht ideal, weil alle Code Repository Instanzen auf dem selben Computer wären, ein katastrophaler Verlust wäre wahrscheinlicher.

Wenn Du ein gemeinsames Dateisystem eingebunden hast, kannst Du von einem lokalen Datei-basiertem Repository klonen, pushen und pullen. Um ein Repository wie dieses zu klonen, oder ein externes zu einem bestehenden Projekt hinzuzufügen, benutze den Pfad zu dem Repository als URL. Um zum Beispiel ein lokales Repository zu klonen kannst Du einen Befehl wie diesen nutzen:

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

Oder Du kannst das machen:

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

Git arbeitet etwas anders, wenn Du am Anfang der URL ausdrücklich `file://` angibst. Wenn Du nur den Pfad angibst, und sowohl die Quelle, als auch das Ziel sich auf dem selben Dateisystem befinden, versucht Git harte Links zu benutzen. Wenn sie sich nicht auf dem selben Dateisystem befinden, kopiert Git die benötigten Objekte mit Hilfe der Standardkopierfunktion des jeweiligen Betriebssystems. Wenn Du `file://` angibst, startet Git den Prozess, den es normalerweise zum Übertragen von Daten über ein Netzwerk verwendet, dass ist gewöhnlich eine wesentlich ineffizientere Methode zum Übertragen der Daten. Der Hauptgrund das `file://`-Präfix anzugeben ist eine saubere Kopie von dem Repository mit fremden Referenzen oder fehlenden Objekten – generell nach einem Import von einem anderen Versionskontrollsystem oder etwas ähnliches (siehe Kapitel 9 für Wartungsarbeiten). Wir benutzen hier den normalen Pfad, weil das fast immer schneller ist.

Um ein lokales Repository zu einem existierenden Git Projekt hinzuzufügen, kannst Du einen Befehl wie diesen ausführen:

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


Dann kannst Du zu diesem externen Repository pushen und davon pullen, als würdest Du das über ein Netzwerk machen.

Die Vorteile

Die Vorteile von Datei-basierten Repositories sind die Einfachheit und das Nutzen bereits bestehender Datei-Berechtigungen und bestehendem Netzwerk-Zugriff. Wenn Du bereits ein gemeinsames Dateisystem hast, zu dem das gesamte Team Zugriff hat, ist das Einrichten eines Repositories sehr einfach. Du exportierst eine Kopie des einfachen Repositories dahin, wo jeder gemeinsamen Zugriff hat und stellst die Lese- und Schreibberechtigungen wie bei jedem anderem gemeinsamen Verzeichnis ein. Wir werden im nächsten Abschnitt „Git auf einen Server bekommen“ diskutieren, wie man die Kopie eines einfachen Repositories für diesen Zweck exportiert.

Dies ist auch eine nette Möglichkeit zum schnellen Abholen von Änderungen aus dem Arbeitsverzeichnis von jemand anderem. Wenn Du und ein Kollege an dem gleichen Projekt arbeitet und ihr wollt etwas auschecken, dann ein Befehl wie `git pull /home/john/project` ist oft einfacher als das pushen zu einem externen Server und das pullen zurück.

Die Nachteile

Die Nachteile von dieser Methode sind, dass ein gemeinsamer Zugriff im allgemeinen schwieriger einrichten ist und der Zugriff von mehreren Orten ist schwieriger als einfacher Netzwerk Zugriff. Wenn Du von Deinem Laptop zuhause pushen möchtest, musst Du eine entfernte Festplatte einbinden. Das kann schwierig und langsam sein, verglichen mit netzwerk-basiertem Zugriff.

Es ist auch wichtig zu erwähnen, dass dies nicht unbedingt die schnellste Möglichkeit ist, wenn Du ein gemeinsames Dateisystem oder ähnliches hast. Ein lokales Repository ist nur dann schnell, wenn Du schnellen Zugriff auf die Daten hast. Ein NFS-basiertes Repository ist oftmals langsamer als ein Repository über SSH auf dem gleichen Server, weil Git über SSH auf jedem System auf den lokalen Festplatten arbeitet.

Das SSH Protokoll

Das vermutlich gebräuchlichste Transport-Protokoll für Git ist SSH. Das hat den Grund, dass der SSH-Zugriff an den meisten Orten bereits eingerichtet ist – und wenn das nicht der Fall ist, einfach zu machen ist. SSH ist außerdem das einzige netzwerk-basierte Protokoll von dem man einfach lesen und darauf schreiben kann. Die beiden anderen Netzwerk-Protokolle (HTTP und Git) sind nur lesbar. Auch wenn sie für die breite Masse sind, brauchst Du trotzdem SSH für Deine Schreib-Befehle. SSH ist auch ein authentifiziertes Netzwerk-Protokoll, und weil es universell ist, ist es im Allgemeinen einfach einzurichten und zu benutzen.

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

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

Oder Du kannst auch kein Protokoll angeben – Git nimmt SSH an, wenn Du nicht eindeutig bist:

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

Du kannst auch keinen Benutzer angeben, und Git nimmt den Benutzer an, als der Du gerade eingeloggt bist.

Die Vorteile

Die Vorteile von SSH sind vielseitig. Erstens, grundlegend musst Du es benutzen, wenn Du authentifizierten Schreib-Zugriff auf Dein Repository über ein Netzwerk haben möchtest. Zweitens, SSH ist relativ einfach einzurichten – SSH-Dämonen sind alltäglich, viele Netzwerk-Administratoren haben Erfahrungen mit ihnen und viele Betriebssysteme sind mit ihnen eingerichtet oder haben Werkzeuge um sie zu verwalten. Als nächstes, Zugriff über SSH ist sicher – der gesamte Daten-Transfer ist verschlüsselt und authentifiziert. Als letztes, wie Git und die lokalen Protokolle, SSH ist effizient, es macht die Daten so kompakt wie möglich bevor es die Daten überträgt.

Die Nachteile

Die negative Seite von SSH ist, dass Du Deine Repositories nicht anonym darüber anbieten kannst. Die Leute müssen Zugriff auf Deine Maschine über SSH haben um zuzugreifen, auch mit einem Nur-Lese-Zugriff, was SSH nicht zuträglich zu Open-Source-Projekten macht. Wenn Du es nur innerhalb von Deinem Firmen-Netzwerk benutzt, SSH ist vielleicht das einzige Protokoll mit dem Du arbeiten musst. Wenn Du anonymen Nur-Lese-Zugriff zu Deinen Projekten erlauben willst, musst Du SSH für Dich einsetzen um zu pushen, aber ein anderes Protokoll für andere um zu pullen.

Das Git Protokoll

Als nächstes kommt das Git Protokoll. Das ist ein spezieller Dämon, der zusammen mit Git kommt. Er horcht auf einem bestimmten Port (9418), dieser Service ist vergleichbar mit dem SSH-Protokoll, aber ohne jegliche Authentifizierung. Um ein Repository über das Git Protokoll, musst Du die `git-daemon-export-ok` Datei erstellen – der Dämon bietet kein Repository ohne die Datei darin an – außer dieser Datei gibt es keine Sicherheit. Entweder das Git Repository ist für jeden zum Clonen verfügbar oder halt nicht. Das bedeutet, dass dieses Protokoll generell kein push anbietet. Du kannst push-Zugriff aktivieren; aber ohne Authentifizierung, wenn Du den push-Zugriff aktivierst, kann jeder im Internet, der Deine Projekt-URL findet, zu Deinem Projekt pushen. Ausreichend zu sagen, dass das selten ist.

Die Vorteile

Das Git Protokoll ist das schnellste verfügbare Transfer Protokoll. Wenn Du viel Traffic für ein öffentliches Projekt hast oder ein sehr großes Projekt hast, dass keine Benutzer-Authentifizierung

für den Lese-Zugriff voraussetzt, es ist üblich einen Git Dämon einzurichten, der Dein Projekt serviert. Er benutzt den selben Daten-Transfer Mechanismus wie das SSH-Protokoll, aber ohne den Entschlüsselungs- und Authentifizierungs-Overhead.

Die Nachteile

Die Negativseite des Git Protokoll ist das Fehlen der Authentifizierung. Es ist generell unerwünscht, dass das Git Protokoll der einzige Zugang zu dem Projekt ist. Im Allgemeinen willst Du es mit SSH-Zugriff für die Entwickler paaren, die push (Schreib) Zugriff haben und jeder andere benutzt `git://` für Nur-Lese-Zugriff. Es ist vielleicht auch das das schwierigste Protokoll beim Einrichten. Es muss ein eigener Dämon laufen, welcher Git-spezifisch ist – wir wollen im „Gitosis“-Abschnitt in diesem Kapitel schauen, wie man einen einrichtet – es setzt eine `xinetd`-Konfiguration oder ähnliches voraus, das ist nicht immer ein Spaziergang. Es setzt auch einen Firewall-Zugriff auf den Port 9418 voraus, das ist kein Standard-Port, den Firmen-Firewalls immer erlauben. Hinter einer großen Firmen-Firewall ist dieser unklare Port häufig gesperrt.

Das HTTP/S Protokoll

Als letztes haben wir das HTTP Protokoll. Das Schöne am HTTP bzw. HTTPS Protokoll ist die Einfachheit des Einrichtens. Im Grunde musst Du nur das einfach Git Repository in Dein HTTP Hauptverzeichnis legen und einen speziellen `post-update` hook (xxx) einrichten und schon bist Du fertig (siehe Kapitel 7 für Details zu Git hooks (xxx)). Jetzt kann jeder, der auf den Web-Server mit dem Repository zugreifen kann, das Repository klonen. Um Lese-Zugriff auf das Repository über HTTP zu erlauben, führe die folgenden Befehle aus:

```
$ 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 ist alles. Der `post-update` hook (xxx) der standardmäßig zusammen mit Git kommt führt den richtigen Befehl aus (`git update-server-info`), damit der HTTP-Server das Repository richtig abrufen und klonen kann. Dieser Befehl läuft, wenn Du zu diesem Repository per SSH pusht, andere Leute können dann klonen mit dem Befehl

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

In diesem besonderen Fall benutzen Dir den `/var/www/htdocs`-Pfad, der typisch für Apache-Setups ist, aber Du kannst jeden statischen Web-Server benutzen – nur das einfache Repository in den richtigen Ordner legen. Die Git-Daten werden als einfache statische Dateien serviert (siehe Kapitel 9 für Details, wie es genau serviert wird).

Es ist möglich, Git-Daten auch über HTTP zu pushen, trotzdem wird diese Technik nicht oft eingesetzt und es setzt komplexe WebDAV-Anforderungen voraus. Weil es selten genutzt wird, werden wir das nicht in diesem Buch behandeln. Wenn Du Interesse am HTTP-Push-Protokoll

hast, kannst Du das Einrichten unter <http://www.kernel.org/pub/software/scm/git/docs/howto/setup-git-server-over-http.txt> nachlesen. Eine Schöne Sache über Git-Push über HTTP ist, dass Du jeden WebDAV-Server benutzen kannst, ohne spezifische Git-Features; also kannst Du diese Funktionalität nutzen, wenn Dein Web-Hosting-Provider WebDAV unterstützt, um Änderungen auf Deine Webseite zu schreiben.

Die Vorteile

Die Positivseite beim Benutzen des HTTP-Protokolls ist, dass es einfach einzurichten ist. Das Ausführen von einer Handvoll notwendiger Befehle ist ein einfacher Weg, um der Welt Lese-Zugriff auf Dein Git-Repository zu geben. Es braucht nur ein paar Minuten. Das HTTP Protokoll benötigt auch nicht viele Ressourcen auf Deinem Server. Es braucht generell nur einen statischen Server um die Daten zu auszuliefern. Ein normaler Apache-Server kann Tausende von Dateien pro Sekunde servieren – es ist schwierig selbst einen kleinen Server zu überlasten.

Du kannst Deine Repositories auch als Nur-Lese-Repositories über HTTPS servieren, Du kannst also den Daten-Transfer verschlüsseln. Oder Du kannst so weit gehen, dass die Clients spezifische signierte SSL-Zertifikate benutzen. Im Allgemeinen, wenn Du soweit gehst, ist es einfacher öffentliche SSH-Schlüssel zu benutzen; aber es ist vielleicht für Deinen Fall eine bessere Lösung, signierte SSL-Zertifikate zu benutzen oder andere HTTP-basierte Authentifizierungs-Methoden für Nur-Lese-Zugriff über HTTPS zu benutzen.

Eine andere schöne Sache ist, dass HTTP so oft genutzt wird, dass Firmen-Firewalls oft Traffic über den HTTP-Port erlauben.

Die Nachteile

Die Unterseite vom Servieren von Deinem Repository über HTTP ist, dass es recht ineffizient für den Client ist. Es braucht im Allgemeinen länger zu clonen oder Daten vom Repository zu holen und Du hast oft wesentlich mehr Netzwerk-Overhead und Transfer-Volumen als mit jedem anderen Netzwerk Protokoll. Weil es nicht so intelligent beim Daten-Transfer ist, um nur die benötigten Daten zu übertragen – es gibt keine dynamische Arbeit auf dem Server bei diesen Aktionen – das HTTP Protokoll wird oft als *dummes* Protokoll bezeichnet. Für mehr Informationen über die Unterschiede bei der Effizienz zwischen dem HTTP Protokoll und den anderen Protokollen: siehe Kapitel 9.

Git auf einen Server bekommen

Um zunächst einen beliebigen Git Server einzurichten, musst Du ein existierendes Repository in ein neues einfaches Repository exportieren – ein Repository, dass kein Arbeitsverzeichnis enthält. Das ist im Allgemeinen einfach zu erledigen. Um zunächst Dein Repository zu klonen, um ein neues einfaches Repository anzulegen, führst Du den Klonen-Befehl mit der `--bare` Option aus. Per Konvention haben einfache Repository Verzeichnisse die Endung `.git`, wie hier:

```
$ git clone --bare my_project my_project.git
Initialized empty Git repository in /opt/projects/my_project.git/
```

Die Ausgabe für diesen Befehl ist etwas verwirrend. Weil `clone` im Grunde ein `git init` und ein `git fetch` ist, sehen wir eine Ausgabe vom `git init`-Teil, der ein leeres Verzeichnis anlegt. Die eigentliche Objekt-Übertragung erzeugt keine Ausgabe, aber sie findet statt. Du solltest jetzt eine Kopie von den Git-Verzeichnis Daten in Deinem `my_project.git` Verzeichnis haben.

Dies ist entsprechend zu etwas wie

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

Es gibt ein paar kleine Unterschiede in der Konfigurationsdatei, aber für Deine Zwecke ist es nahezu dasselbe. Es nimmt das Git-Repository selbst, ohne ein Arbeitsverzeichnis, und erzeugt ein Verzeichnis speziell für es allein.

Inbetriebnahme des einfachen Repository auf einem Server

Jetzt da Du eine einfache Kopie Deines Repository hast, ist alles was Du tun musst das Aufsetzen auf einem Server und das Einrichten Deiner Protokolle. Angenommen, Du hast einen Server mit dem Namen `git.example.com`, zu dem Du SSH-Zugang hast, und Du möchtest alle Deine Git Repositories im Verzeichnis `/opt/git` speichern. Du kannst Dein neues Repository einrichten, indem Du Dein einfaches Repository dorthin kopierst:

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

An diesem Punkt können andere Benutzer, die SSH-Zugang zu dem gleichen Server und Lesezugriff auf das `/opt/git` Verzeichnis haben, Dein Repository klonen:

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

Wenn sich ein Benutzer per SSH mit dem Server verbindet und Schreibzugriff zu dem Verzeichnis `/opt/git/my_project.git` hat, wird er automatisch auch Push-Zugriff haben. Git wird automatisch die richtigen Gruppenschreibrechte zu einem Repository hinzufügen, wenn Du den `git init` Befehl mit der `--shared` Option ausführst:

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

Du siehst wie einfach es ist ein Git Repository zu nehmen, eine einfache Version zu erzeugen, es auf einen Server zu platzieren zu dem Du und Deine Mitarbeiter SSH-Zugriff haben.

Es ist wichtig zu beachten, dass dies wortwörtlich alles ist, was Du tun musst, um einen nützlichen Git-Server laufen zu lassen, zu dem mehrere Personen Zugriff haben – füge auf dem Server einfach SSH-fähige Konten und irgendwo ein einfaches Repository hinzu, zu dem alle Benutzer Schreib- und Lesezugriff haben.

In den nächsten Abschnitten wirst Du sehen, wie man auf anspruchsvollere Konfigurationen erweitert. Diese Diskussion wird beinhalten nicht Benutzerkonten für jeden Benutzer hinzufügen zu müssen, öffentlichen Lese-Zugriff auf Repositories hinzuzufügen, die Einrichtung von Web-UIs, die Benutzung des Gitis-Tools und weiteres. Aber denke daran, zur Zusammenarbeit mit ein paar Personen an einem privaten Projekt, ist alles was Du *brauchst* ein SSH-Server und ein einfaches Repository.

Kleine Konfigurationen

Wenn Du eine kleine Ausrüstung hast oder Git gerade in Deinem Unternehmen ausprobierst und nur ein paar Entwickler hast, sind die Dinge einfach für dich. Einer der kompliziertesten Aspekte der Einrichtung eines Git-Servers ist die Benutzerverwaltung. Wenn einige Repositories für bestimmte Benutzer nur lesend zugänglich sein sollen und andere Benutzer Lese/Schreib-Zugriff haben sollen, können Zugriff und Berechtigungen ein bisschen schwierig zu organisieren sein.

SSH-Zugriff

Wenn Du bereits einen Server hast, zu dem alle Entwickler SSH-Zugriff haben, ist es generell einfach, Dein erstes Repository einzurichten, weil Du fast keine Arbeit zu erledigen hast (wie wir im letzten Abschnitt abgedeckt haben). Wenn Du komplexere Zugriffskontroll-Berechtigungen auf Deine Repositories willst, kannst Du diese mit normalen Dateisystem-Berechtigungen des Betriebssystems Deines Servers bewältigen.

Wenn Du Deine Repositories auf einem Server platzieren möchtest, der keine Accounts für jeden aus Deinem Team hat, der Schreibzugriff haben soll, dann musst Du SSH-Zugriff für diese Personen einrichten. Wir nehmen an, wenn Du einen Server hast, mit welchem Du dies tun möchtest, Du bereits einen SSH-Server installiert hast und so greifst Du auf den Server zu.

Es gibt ein paar Wege allen Mitgliedern Deines Teams Zugriff zu geben. Der erste ist einen Account für jeden einzurichten, was unkompliziert aber mühsam sein kann. Du möchtest vielleicht nicht für jeden Benutzer `adduser` ausführen und ein temporäres Passwort setzen.

Eine zweite Methode ist, einen einzigen ‚git‘-Benutzer auf der Maschine zu erstellen und jeden Benutzer, der Schreibzugriff haben soll, nach einem öffentlichen SSH-Schlüssel zu fragen und diesen Schlüssel zu der `~/.ssh/authorized_keys`-Datei Deines neuen ‚git‘ Benutzers hinzuzufügen. Das beeinflusst die Commit-Daten in keiner Weise – der SSH-Benutzer, mit dem Du Dich verbindest, beeinflusst die von Dir aufgezeichneten Commits nicht.

Ein anderer Weg ist, einen LDAP-Server zur Authentifizierung zu benutzen oder eine andere zentrale Authentifizierungsquelle, die Du vielleicht bereits eingerichtet hast. Solange jeder Benutzer Shell-Zugriff zu der Maschine hat, sollte jede SSH-Authentifizierungsmethode funktionieren, die Du Dir vorstellen kannst.

Generiere Deinen öffentlichen SSH-Schlüssel

Darüber hinaus benutzen viele Git-Server öffentliche SSH-Schlüssel zur Authentifizierung. Um einen öffentlichen Schlüssel bereitzustellen muss jeder Benutzer Deines Systems einen solchen Schlüssel generieren, falls sie noch keinen haben. Dieser Prozess ist bei allen Betriebssystemen ähnlich. Als erstes solltest Du überprüfen, ob Du nicht schon einen Schlüssel hast. Standardmäßig werden die SSH-Schlüssel der Benutzer in ihrem `~/.ssh`-Verzeichnis gespeichert. Du kannst einfach überprüfen, ob Du einen Schlüssel hast, indem Du in das Verzeichnis gehst und den Inhalt auflistest:

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

Du suchst nach Paar Dateien namens `irgendetwas` und `irgendetwas.pub`, die Datei `irgendetwas` heißt normalerweise `id_dsa` oder `id_rsa`. Die `.pub`-Datei ist Dein öffentlicher Schlüssel und die andere Datei ist Dein privater Schlüssel. Wenn Du diese Dateien nicht hast (oder gar kein `.ssh`-Verzeichnis hast), kannst Du sie mit dem Ausführen des Programms `ssh-keygen` erzeugen. Das Programm wird mit dem SSH-Paket auf Linux/Mac-Systemen mitgeliefert und kommt mit dem MSysGit-Paket unter Windows:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/schacon/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /Users/schacon/.ssh/id_rsa.
Your public key has been saved in /Users/schacon/.ssh/id_rsa.pub.
The key fingerprint is:
43:c5:5b:5f:b1:f1:50:43:ad:20:a6:92:6a:1f:9a:3a schacon@agadorlaptop.local
```

Zunächst wird bestätigt, wo Du den Schlüssel speichern möchtest (`.ssh/id_rsa`) und dann wird zweimal nach der Passphrase gefragt, die Du leer lassen kannst, wenn Du kein Passwort bei der Benutzung des Schlüssels eintippen möchtest.

Jeder Benutzer der dies macht, muss seinen öffentlichen Schlüssel an sich senden oder wer auch immer den Git-Server administriert (angenommen Du benutzt eine SSH-Server Konfiguration, die öffentliche Schlüssel benötigt). Alles was die Benutzer tun müssen ist, den Inhalt der `.pub`-Datei zu kopieren und an Dich per E-Mail zu schicken. Der öffentliche Schlüssel sieht etwa wie folgt aus:

```
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAABIWAAAQEAKlOUpkDHRfHY17SbrmTIpNLTKG9Tjom/BWDSU
GPl+nafz1HDTYW7hdI4yZ5ew18JH4JW9jbhUFrviQzM7x1ELEVf4h9lFX5QVkbPppSwg0cda3
Pbv7k0dJ/MTyBlWXFCR+HAo3FXRitBqxiX1nKhXpHAZsMciLq8V6RjsNAQwdsdMFvSlVK/7XA
t3FaoJoAsncM1Q9x5+3V0Ww68/eIFmb1zuUFljQJKprX88XypNDvjYNby6vw/Pb0rwert/En
mZ+AW40ZPnTPI89ZPmVMLuayrD2cE86Z/il8b+gw3r3+1nKatmIkjn2so1d01QraTlMqVSsbx
NrRFi9wrf+M7Q== schacon@agadorlaptop.local
```


Eine detailliertere Anleitung zur Erstellung eines SSH-Schlüssels unter den verschiedenen Betriebssystemen ist der GitHub-Leitfaden für SSH-Schlüssel unter <http://github.com/guides/providing-your-ssh-key>.

Einrichten des Servers

Nun kommen wir zur Einrichtung des SSH-Zugangs auf der Server-Seite. In diesem Beispiel verwendest Du die `authorized_keys`-Methode zur Authentifizierung der Benutzer. Wir nehmen auch an, dass Du eine gebräuchliche Linux-Distribution wie Ubuntu verwendest. Zuerst erstellst Du den Benutzer `git` und ein `.ssh`-Verzeichnis für diesen Benutzer.

```
$ sudo adduser git
$ su git
$ cd
$ mkdir .ssh
```

Als nächstes ist es nötig, einige öffentliche SSH-Schlüssel der Entwickler zu der `authorized_keys`-Datei des Benutzers hinzuzufügen. Nehmen wir an, dass Du ein paar Schlüssel per E-Mail empfangen hast und diese in temporären Dateien gespeichert hast. Die öffentlichen Schlüssel sehen wieder etwa wie folgt aus:

```
$ cat /tmp/id_rsa.john.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQCB007n/ww+ouN4gSLKssMxXnB0vf9LGt4L
ojG6rs6hPB09j9R/T17/x4lhJA0F3FR1rP6kYBRswj2aThGw6HXLm9/5zytK6Ztg3RPKK+4k
Yjh6541NYSnEAZuXz0jTTyAUfrtU3Z5E003C4ox0j6H0rfIF1kKI9MAQLMdpGW1GYEIgS9Ez
Sdfd8AcCIicTDWbqLAcU4UpkaX8KyG1LwsNuuGztobF8m72ALC/nLF6JLtPofwFBlgc+myiv
07TCUSBDLQlgMV0Fq1I2uPWQ0k0WQAHuke0mfjy2jctxSDBQ220ymjaNsHT4kgtZg2AYYgPq
dAv8JggJICUvax2T9va5 gsg-keypair
```

Du hängst sie einfach an Deine `authorized_keys`-Datei 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
```

Jetzt kannst Du einen leeren Ordner für sie anlegen, indem Du den Befehl `git init` mit der Option `--bare` ausführst. Damit wird ein Repository ohne ein Arbeitsverzeichnis erzeugt.

```
$ cd /opt/git
$ mkdir project.git
$ cd project.git
$ git --bare init
```

Dann können John, Josie oder Jessica die erste Version ihres Projektes in das Repository hochladen, indem sie es als externes Repository hinzufügen und einen Branch hochladen. Beachte, dass sich bei jeder Projekterstellung jemand mit der Maschine auf eine Shell verbinden muss, um ein einfaches Repository zu erzeugen. Lass uns `gitserver` als Hostnamen des Servers verwenden, auf dem Du den Benutzer `git` und das Repository eingerichtet hast. Wenn Du den Server intern betreibst und das DNS so eingerichtet hast, dass `gitserver` auf den Server zeigt, dann kannst Du die Befehle ziemlich wie hier benutzen:

```
# on Johns computer
$ cd myproject
```

```
$ git init
$ git add .
$ git commit -m 'initial commit'
$ git remote add origin git@gitserver:/opt/git/project.git
$ git push origin master
```

An diesem Punkt können die anderen das Repository klonen und Änderungen ebenso leicht hochladen:

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

Mit dieser Methode kannst Du schnell für eine Handvoll Entwickler einen Lese/Schreib Git-Server zum Laufen bekommen.

Als zusätzliche Vorsichtsmaßnahme kannst Du den Benutzer ‚git‘ so beschränken, dass er nur Git-Aktivitäten mit einem limitierten Shell-Tool namens `git-shell` ausführen kann, dass mit Git kommt. Wenn Du das als Login-Shell des ‚git‘-Benutzers einrichtest, dann hat der Benutzer ‚git‘ keinen normalen Shell-Zugriff auf den Server. Zur Benutzung bestimme `git-shell` anstatt von `bash` oder `csh` als Login-Shell Deines Benutzers. Um das zu tun wirst Du wahrscheinlich Deine `/etc/passwd` editieren:

```
$ sudo vim /etc/passwd
```

Am Ende solltest Du eine Zeile finden, die in etwa so aussieht:

```
git:x:1000:1000::/home/git:/bin/sh
```

Ändere `/bin/sh` zu `/usr/bin/git-shell` (oder führe `which git-shell` aus, um zu sehen, wo es installiert ist). Die Zeile sollte in etwa so aussehen:

```
git:x:1000:1000::/home/git:/usr/bin/git-shell
```

Jetzt kann der ‚git‘-Benutzer die SSH-Verbindung nur noch verwenden, um Git-Repositories hochzuladen und herunterzuladen. Der Benutzer kann sich nicht mehr per Shell zur Maschine verbinden. Wenn Du es versuchst, siehst Du eine Login-Ablehnung wie diese:

```
$ ssh git@gitserver
fatal: What do you think I am? A shell?
Connection to gitserver closed.
```

Öffentlicher Zugang

Was ist, wenn Du anonymen Lese-Zugriff zu Deinem Projekt ermöglichen möchtest? Vielleicht möchtest Du ein Open-Source Projekt, anstatt einem privaten, nicht öffentlichen Projekt hosten. Oder Du hast ein paar automatisierte Build-Server oder Continuous Integration Server, die ständig wechseln, und Du möchtest für diese nicht dauernd neue SSH-Schlüssel generieren. Dann wäre es doch schön, wenn ein anonymer Lese-Zugriff zu Deinem Projekt möglich wäre.

Der wahrscheinlich einfachste Weg für kleinere Konfigurationen ist, einen Webserver, in dessen Basisverzeichnis die Git Repositorys liegen, laufen zu lassen und den `post-update` Hook, den wir im ersten Abschnitt dieses Kapitels erwähnt haben, zu aktivieren. Gehen wir vom vorherigen Beispiel aus. Sagen wir, Du hast Deine Repositorys im Verzeichnis `/opt/git` und ein Apache-Server läuft auf Deiner Maschine. Du kannst dafür jeden beliebigen Webserver benutzen, aber in diesem Beispiel demonstrieren wir das Ganze an Hand einer Apache Basis-Konfiguration. Dies sollte Dir eine Vorstellung geben, wie Du es mit dem Webserver Deiner Wahl umsetzen kannst.

Zuerst musst Du den Hook aktivieren:

```
$ cd project.git
$ mv hooks/post-update.sample hooks/post-update
$ chmod a+x hooks/post-update
```

Wenn Du eine ältere Git Version als 1.6 benutzt, brauchst Du den `mv`-Befehl nicht auszuführen. Das Namensschema mit der `.sample` Endung wurde erst bei den neueren Git Versionen eingeführt.

Welche Aufgabe hat der `post-update` Hook? Er enthält in etwa folgendes:

```
$ cat .git/hooks/post-update
#!/bin/sh
exec git-update-server-info
```

Wenn Du via SSH etwas auf den Server hochlädst, wird Git den Befehl `git-update-server-info` ausführen. Dieser Befehl aktualisiert alle Dateien, die benötigt werden, damit das Repository über HTTP geholt (`fetch`) beziehungsweise geklont werden kann.

Als nächstes musst Du einen VirtualHost Eintrag zu Deiner Apache-Konfiguration hinzufügen. Das dort angegebene DocumentRoot Verzeichnis muss mit dem Basisverzeichnis Deiner Git Projekte übereinstimmen. In diesem Beispiel gehen wir davon aus, dass ein Wildcard-DNS Eintrag besteht, der dafür sorgt, dass `*.gitserver` auf den Server zeigt, auf dem das Ganze hier läuft:

```
<VirtualHost *:80>
    ServerName git.gitserver
    DocumentRoot /opt/git
    <Directory /opt/git/>
        Order allow, deny
        allow from all
    </Directory>
```

</VirtualHost>

Da die Apache-Instanz, die das CGI-Skript ausführt, standardmäßig unter dem Benutzer `www-data` läuft, musst Du auch die Unix Eigentümer-Gruppe des Verzeichnisses `/opt/git` auf `www-data` setzen. Ansonsten kann Dein Webserver die Repositorys nicht lesen:

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

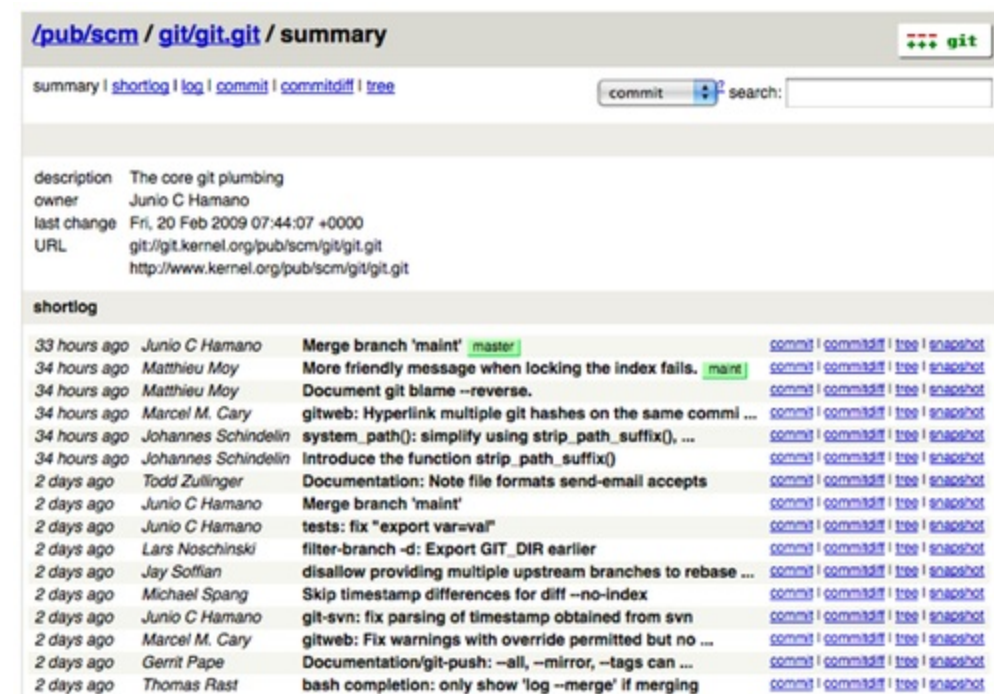
Nach einem Neustart des Apache, solltest Du in der Lage sein, Deine Repositorys innerhalb diesem Verzeichnis zu klonen, indem Du die URL für das jeweilige Projekt angibst.

```
$ git clone http://git.gitserver/project.git
```

Auf diese Art und Weise kannst Du in wenigen Minuten einen HTTP-basierten Lese-Zugriff auf all Deine Projekte für eine große Anzahl von Benutzern ermöglichen. Ein Git Daemon ist eine andere einfache Möglichkeit für einen öffentlichen Zugang. Wenn Du diese Methode bevorzugst, solltest Du Dir den nächsten Abschnitt unbedingt anschauen.

GitWeb

Da Du jetzt sowohl einen einfachen Lese- und Schreibzugriff, als auch einen schreibgeschützten Zugang auf Dein Projekt hast, wird es jetzt Zeit eine simple webbasierte Visualisierung dafür einzurichten. Git wird mit einem CGI-Skript namens GitWeb ausgeliefert, welches für diese Aufgabe häufig verwendet wird. Auf Seiten, wie zum Beispiel <http://git.kernel.org>, kannst Du Dir GitWeb in Aktion anschauen (siehe Abbildung 4-1).



Wenn Du Dir anschauen möchtest, wie GitWeb bei Deinem Projekt ausschauen würde, gibt es dafür eine einfache Möglichkeit. Wenn Du einen einfachen Webserver wie `lighttpd` or `webrick` auf Deinem Server installiert hast, kannst Du mit einem in Git integrierten Kommando eine temporäre Instanz von GitWeb starten. Da `lighttpd` auf vielen Linux Rechnern bereits installiert ist, kannst Du versuchen ihn zum Laufen zu bringen, indem Du das Kommando `git instaweb` in Deinem Projektverzeichnis ausführst. Bei Mac OS X 10.5 alias Leopard ist Ruby bereits vorinstalliert. Falls Du also einen Mac verwendest, solltest Du es mal mit `webrick` versuchen. Um `instaweb` mit einem anderen Webserver als `lighttpd` zu starten, kannst Du an den Befehl die Option `--httpd` anhängen.

```
$ 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]
```

Dadurch wird auf dem Port 1234 ein HTTPD Server gestartet. Gleichzeitig wird automatisch Dein Webbrowser mit der entsprechenden Seite geöffnet. Das Ganze gestaltet sich also ziemlich einfach. Wenn Du dann fertig bist und den Server wieder beenden willst, kannst Du das gleiche Kommando mit der Option `--stop` ausführen:

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

Wenn Du das Web Interface für Dein Team oder ein von Dir gehostetes Open-Source Projekt

dauerhaft zur Verfügung stellen willst, musst Du das CGI-Skript so einrichten, dass es von Deinem normalen Webserver zur Verfügung gestellt werden kann. Bei manchen Linux Distributionen ist ein gitweb Paket enthalten, welches Du via apt oder yum installieren kannst. Vielleicht probierst Du das einfach mal zuerst aus. Wir werden hier nämlich die manuelle Installation von GitWeb nur kurz überfliegen. Zum Starten benötigst Du den Git Quellcode. Dort ist GitWeb enthalten und Du kannst damit Dein angepasstes CGI-Skript erstellen:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git
$ cd git/
$ make GITWEB_PROJECTROOT="/opt/git" \
    prefix=/usr gitweb
$ sudo cp -Rf gitweb /var/www/
```

Bitte beachte, dass Du dem Kommando angeben musst, wo sich Deine Git Repositorys befinden. Dazu wird die GITWEB_PROJECTROOT Variable verwendet. Jetzt musst Du den Apache noch so konfigurieren, dass er CGI für das Skript verwendet. Dazu kannst Du einen VirtualHost einrichten:

```
<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>
```

Ich möchte Dich noch einmal darauf hinweisen, dass GitWeb prinzipiell mit jedem CGI-fähigen Webserver funktioniert. Wenn Du einen anderen Webserver bevorzugst, sollte es also kein Problem sein, GitWeb dafür einzurichten. Nach einem Neustart Deines Apache solltest Du jetzt in der Lage sein, Deine Repositorys über die Adresse `http://gitserver/` in GitWeb anzuschauen. Gleichzeitig kannst Du über `http://git.gitserver` Deine Repositorys per HTTP klonen und abholen (im Sinne eines Fetch).

Gitosis

Das manuelle Verwalten der öffentlichen Benutzerschlüssel in der Datei `authorized_keys` ist auf Dauer nicht sinnvoll. Wenn Du hunderte von Benutzer verwalten musst, wird dieser Prozess noch viel schwieriger und macht keinen Spaß mehr. Du musst jedes mal über die Shell auf Deinen Server zugreifen und es gibt auch keine Zugriffsberechtigungen. Jeder der in der `authorized_keys` Datei eingetragen ist, hat auf jedes Projekt Lese- und Schreibzugriff.

Vielleicht ist es deshalb sinnvoll, dass Du Dich mit dem weit verbreiteten Projekt Gitosis beschäftigst. Gitosis ist im Grunde eine Sammlung von Skripts, die Dir dabei helfen, sowohl die Datei `authorized_keys` zu verwalten, als auch ein paar einfache Zugriffsberechtigungen zu setzen. Das wirklich interessante an diesem Werkzeug ist es, dass die Benutzeroberfläche zum Hinzufügen von Benutzern oder Setzen von Berechtigungen, kein Web Interface ist, sondern ein spezielles Git Repository. Die ganzen Informationen werden in diesem Projekt verwaltet und sobald dieses gepusht wird, konfiguriert Gitosis den Server auf Basis dieser Daten entsprechend um. Das ist ziemlich cool, oder?

Die Installation von Gitosis ist nicht einfach, aber auf jeden Fall machbar. Am einfachsten gestaltet sich die Installation auf einem Linux Server. In unserem Beispiel verwenden wir dafür einen Standard Ubuntu Server in der Version 8.10.

Gitosis setzt einige Python Werkzeuge voraus. Deshalb solltest Du zuerst das Python Setuptools Paket installieren. Unter Ubuntu wird es als `python-setuptools` zur Verfügung gestellt:

```
$ apt-get install python-setuptools
```

Danach klonst Du Gitosis von der offiziellen Projektseite und installierst es:

```
$ git clone git://eagain.net/gitosis.git
$ cd gitosis
$ sudo python setup.py install
```

Einige ausführbare Dateien, welche von Gitosis benötigt werden, werden mit dem Skript installiert. Außerdem will Gitosis seine Repositories im Verzeichnis `/home/git` ablegen. Das ist auch ok so. Allerdings liegen unsere Repositories bereits im Verzeichnis `/opt/git`, aber anstatt alles umzukonfigurieren, erstellst Du einfach eine symbolische Verknüpfung (symlink):

```
$ ln -s /opt/git /home/git/repositories
```

Gitosis wird Deine Schlüssel für Dich verwalten. Deshalb musst Du die bereits vorhandene Datei `authorized_keys` entfernen, die Schlüssel später wieder hinzufügen und Gitosis die automatisierte Verarbeitung dieser Datei überlassen. Zuerst musst Du also die Datei `authorized_keys` aus dem Weg räumen:

```
$ mv /home/git/.ssh/authorized_keys /home/git/.ssh/ak.bak
```

Wenn Du für den Benutzer `,git‘` die Shell auf die `git-shell` gesetzt hast (siehe Kapitel 4.4),

musst Du die normale Shell für diesen Benutzer wieder aktivieren. Den Benutzern wird es danach immer noch nicht möglich sein sich einzuloggen, dafür sorgt Gitosis. Ändere also in Deiner `/etc/passwd` die Zeile

```
git:x:1000:1000:~/home/git:/usr/bin/git-shell
```

zurück in folgendes:

```
git:x:1000:1000:~/home/git:/bin/sh
```

Jetzt wird es Zeit Gitosis zu initialisieren. Dafür musst Du das Kommando `gitosis-init` mit Deinem öffentlichen Schlüssel ausführen. Wenn sich Dein öffentlicher Schlüssel nicht auf dem Server befindet, musst Du ihn vorher dorthin kopieren:

```
$ sudo -H -u git gitosis-init < /tmp/id_dsa.pub
Initialized empty Git repository in /opt/git/gitosis-admin.git/
Reinitialized existing Git repository in /opt/git/gitosis-admin.git/
```

Dem Benutzer, mit dem hier angegeben Schlüssel, ist es jetzt möglich, das Git Repository, mit dem Gitosis konfiguriert wird, zu modifizieren. Als nächstes musst Du manuell das Execute-Bit für das Skript `post-update` in Deinem neuen „Verwaltungs“-Repository setzen.

```
$ sudo chmod 755 /opt/git/gitosis-admin.git/hooks/post-update
```

Jetzt kann es losgehen. Wenn Du alles richtig eingerichtet hast, kannst Du jetzt versuchen Dich über SSH einzuloggen. Du musst dafür den Benutzer verwenden, dem der öffentliche Schlüssel gehört, den Du in den vorherigen Schritten hinzugefügt hast. Du solltest dann in etwa folgende Ausgabe erhalten:

```
$ ssh git@gitserver
PTY allocation request failed on channel 0
fatal: unrecognized command 'gitosis-serve schacon@quaternion'
Connection to gitserver closed.
```

Das bedeutet das Gitosis Dich als Benutzer kennt, aber Dich ausschließt, weil Du nicht versuchst ein Git Kommando auszuführen. Lass uns also ein Git Kommando ausprobieren. Wir klonen dazu das Verwaltungsrepository von Gitosis:

```
# on your local computer
$ git clone git@gitserver:gitosis-admin.git
```

Jetzt hast Du auf Deinem Rechner ein Verzeichnis mit dem Namen `gitosis-admin`. Dieses besteht aus hauptsächlich zwei Teilen:

```
$ cd gitosis-admin
$ find .
./gitosis.conf
./keydir
./keydir/scott.pub
```

Mit Hilfe der Datei `gitosis.conf` kannst Du die Benutzer, Repositorys und die Zugriffsrechte festlegen. Im Verzeichnis `keydir` werden alle öffentlichen Schlüssel der Benutzer abgelegt, die einen Zugriff auf Deine Repositorys haben sollen und zwar für jeden Benutzer eine einzelne Datei. Der Name der Datei, die sich jetzt bereits im `keydir` Verzeichnis befindet (in diesem Beispiel ist es `scott.pub`), wird bei Dir anders lauten. Die Beschreibung, die sich am Ende des öffentlichen Schlüssels befindet, der beim initialen Import mit dem Skript `gitosis-init` angegeben wurde, wird von Gitosis als Dateiname verwendet.

Wenn Du Dir die Datei `gitosis.conf` anschaust, sollten lediglich Daten für das Projekt `gitosis-admin` enthalten sein. `gitosis-admin` ist das Projekt, welches Du gerade geklont hast.

```
$ cat gitosis.conf
[gitosis]

[group gitosis-admin]
writable = gitosis-admin
members = scott
```

In dieser Datei wird Dir angezeigt, dass nur der Benutzer ‚scott‘ — das ist der Anwender mit dessen öffentlichen Schlüssel Gitosis initialisiert wurde — Zugriff auf das Projekt `gitosis-admin` hat.

Lass uns jetzt für Dich ein neues Projekt hinzufügen. Dazu fügen wir eine neue Sektion mit dem Namen `mobile` hinzu, in der wir alle Teammitglieder aus dem „Mobile“-Team und alle Repositorys, die von Ihnen benötigt werden, auflisten. Da ‚scott‘ bisher der einzige Benutzer im System ist, werden wir ihn als einziges Teammitglied hinzufügen. Das erste von uns erzeugte Projekt mit dem wir anfangen, nennt sich `iphone_project`:

```
[group mobile]
writable = iphone_project
members = scott
```

Immer wenn Du Änderungen im Projekt `gitosis-admin` durchführst, musst Du diese Änderungen auch einchecken und auf den Server pushen, damit diese auch eine Wirkung zeigen:

```
$ git commit -am 'add iphone_project and mobile group'
[master]: created 8962da8: "changed name"
 1 files changed, 4 insertions(+), 0 deletions(-)
$ git push
Counting objects: 5, done.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 272 bytes, done.
Total 3 (delta 1), reused 0 (delta 0)
To git@gitserver:/opt/git/gitosis-admin.git
 fb27aec..8962da8  master -> master
```

Du kannst Deinen ersten Push für das neue Projekt `iphone_project` ausführen, indem Du Deinen Server als Remote zu Deinem lokalen Repository hinzufügst und dann auf diesen pushst. Ab jetzt musst Du auf Deinem Server nicht mehr manuell ein Bare Repository für neue Projekte erstellen.

Gitosis übernimmt diese Aufgabe für Dich, sobald es den ersten Push erhält:

```
$ git remote add origin git@gitserver:iphone_project.git
$ git push origin master
Initialized empty Git repository in /opt/git/iphone_project.git/
Counting objects: 3, done.
Writing objects: 100% (3/3), 230 bytes, done.
Total 3 (delta 0), reused 0 (delta 0)
To git@gitserver:iphone_project.git
 * [new branch]      master -> master
```

Beachte, dass Du den Pfad nicht angeben musst (es ist sogar so, wenn Du ihn angibst, wird es nicht funktionieren). Gib lediglich ein Doppelpunkt gefolgt vom Namen des Projekts an. Das reicht Gitosis aus, um den richtigen Pfad für Dich zu finden.

Da Du an diesem Projekt nicht alleine, sondern mit Deinen Freunden, arbeiten willst, musst Du deren öffentliche Schlüssel wieder hinzufügen. Aber anstatt diese manuell in der Datei `~/.ssh/authorized_keys` auf Deinem Server einzutragen, fügst Du jeden Schlüssel als einzelne Datei in das Verzeichnis `keydir` hinzu. Der Name der Dateien entspricht den gleichen Namen, auf die Du in der Datei `gitosis.conf` referenzierst. Lass uns für John, Josie und Jessica die öffentlichen Schlüssel hinzufügen:

```
$ cp /tmp/id_rsa.john.pub keydir/john.pub
$ cp /tmp/id_rsa.josie.pub keydir/josie.pub
$ cp /tmp/id_rsa.jessica.pub keydir/jessica.pub
```

Damit diese Personen Lese- und Schreibzugriff auf das Projekt `iphone_project` haben, musst Du sie dem ‚mobile‘-Team hinzufügen:

```
[group mobile]
writable = iphone_project
members = scott john josie jessica
```

Nachdem Du diese Änderung commitet und gepusht hast, können alle vier Benutzer dieses Projekt lesen und schreiben.

Mit Gitosis kann man auch einfache Zugriffsberechtigungen setzen. Wenn John nur Lesezugriff zum Projekt haben soll, dann kannst Du stattdessen folgendes angeben:

```
[group mobile]
writable = iphone_project
members = scott josie jessica
```

```
[group mobile_ro]
readonly = iphone_project
members = john
```

Mit dieser Konfiguration kann John das Projekt klonen und kann neue Stände herunterladen, aber Gitosis wird jeden Push von ihm ablehnen. Du kannst beliebig viele Gruppen angeben, die jeweils unterschiedliche Benutzer und Projekte enthalten. Ebenso ist es möglich eine andere Gruppe als

Mitglied hinzuzufügen damit deren Teammitglieder automatisch miteinbezogen werden. Dazu musst Du bei der Gruppe @ als Präfix angeben:

```
[group mobile_committers]
members = scott josie jessica
```

```
[group mobile]
writable  = iphone_project
members   = @mobile_committers
```

```
[group mobile_2]
writable  = another_iphone_project
members   = @mobile_committers john
```

Solltest Du Probleme mit Gitis haben, hilft es Dir vielleicht, wenn Du den Eintrag `loglevel=DEBUG` in der Sektion `[gitis]` hinzufügst. Wenn Du keinen Push auf das Verwaltungsrepository ausführen kannst und Du Dich damit ausgeschlossen hast, kannst Du die Konfiguration unter `/home/git/.gitis.conf` manuell bearbeiten. Diese Datei verwendet Gitis als Konfigurationsdatei. Bei einem Push wird diese durch die im Verwaltungsrepository enthaltene Datei ersetzt. Wenn Du diese Datei also manuell änderst, bleibt diese bis zum nächsten erfolgreichen Push auf das Projekt `gitis-admin` bestehen.

Gitolite

In diesem Abschnitt werde ich einen kurzen Einblick in Gitolite geben und die Basisinstallation und Konfiguration besprechen. Jedoch kann meine kurze Einführung nicht die ausführliche [Dokumentation](#), die Gitolite bietet, ersetzen. Es könnte sein, dass es gelegentlich Änderungen an diesem Abschnitt gibt, deshalb solltest Du auch einen Blick auf die [aktuellste Version](#) wagen.

Gitolite ist als Schicht für die Zugriffsberechtigung oberhalb von Git angeordnet und verwendet sshd oder httpd zur Authentifikation. (Kurze Wiederholung: Bei der Authentifizierung wird der Benutzer identifiziert und die Zugriffsberechtigung entscheidet, ob der Benutzer die gewünschte Operation ausführen darf oder nicht).

Gitolite ermöglicht es Dir die Berechtigungen auf Repository Ebene festzulegen, erlaubt aber zusätzlich auch die Berechtigung auf Ebene von Branches oder Tags innerhalb eines Repositorys zu definieren. Das bedeutet also, dass Du festlegen kannst, dass bestimmte Leute (oder eine Gruppe von Leuten) nur bestimmte „refs“ (Branches oder Tags) pushen können, andere Personen sollen das wiederum nicht können.

Installation

Auch ohne Studium der ausführlichen Dokumentation, die Gitolite beiliegt, gestaltet sich die Installtion sehr einfach. Du benötigst dazu einen Account auf irgendeiner Art von Unix Server. Du brauchst keine Root-Rechte, vorausgesetzt Git, Perl und ein OpenSSH kompatibler SSH Server sind bereits installiert. In unserem Beispiel verwenden wir den Benutzer git auf einem Host mit dem Namen gitserver.

Gitolite ist in Bezug auf „Server“-Software ein wenig ungewöhnlich — der Zugriff erfolgt per SSH und somit ist jede auf dem Server vorhandene User-Id ein potentieller „gitolite host“. In unserem Beispiel werden wir die einfachste Methode der Installation beschreiben. Die anderen Möglichkeiten können der Dokumentation entnommen werden.

Zu Beginn legst Du auf Deinem Server einen Benutzer mit dem Namen git an und loggst Dich mit diesem ein. Danach kopierst Du Deinen öffentlichen SSH Schlüssel (die Datei lautet ~/.ssh/id_rsa.pub, falls Du ssh-keygen mit den Standardoptionen ausgeführt hast) von Deiner Workstation auf den Server und nennst ihn entsprechend dem Schema <yourname>.pub um (in unserem Beispiel verwenden wir die Datei scott.pub). Danach führst Du die folgenden Kommandos aus:

```
$ git clone git://github.com/sitaramc/gitolite
$ gitolite/install -ln
# assumes $HOME/bin exists and is in your $PATH
$ gitolite setup -pk $HOME/scott.pub
```

Der letzte Befehl erzeugt ein neues Git Repository mit dem Namen gitolite-admin auf Deinem Server.

Zum Abschluss musst Du auf Deiner Workstation den Befehl `git clone git@gitserver:gitolite-admin` ausführen. Jetzt bist Du im Prinzip fertig. Gitolite ist nun auf Deinem Server installiert und auf Deiner Workstation liegt das neue Repository `gitolite-admin` vor. Du kannst Gitolite nun administrieren, indem Du Änderungen an diesem Repository ausführst und zurück auf den Server pushst.

Benutzerdefinierte Installation

Obwohl die schnelle Standardinstallation für die meisten Leute ausreicht, gibt es ein paar Möglichkeiten die Installation an Deine Gegebenheiten anzupassen, falls Du dies für nötig hältst. Teilweise reicht es, die `rc` Datei zu bearbeiten. Sollte das nicht ausreichen, gibt es genügend Dokumentation, die beschreibt, wie Gitolite angepasst werden kann.

Konfigurationsdateien und Regeln für die Zugangskontrolle

Nachdem die Installation abgeschlossen ist, wechselst Du in den `gitolite-admin` Klon auf Deiner Workstation und stöberst dort am besten ein wenig herum:

```
$ cd ~/gitolite-admin/
$ ls
conf/  keydir/
$ find conf keydir -type f
conf/gitolite.conf
keydir/scott.pub
$ cat conf/gitolite.conf

repo gitolite-admin
    RW+                = scott

repo testing
    RW+                = @all
```

Es ist wichtig anzumerken, dass „scott“ (das entspricht dem Name des öffentlichen Schlüssel, den Du beim Ausführen des `gitolite setup` Kommandos angegeben hast) Lese- und Schreibzugriff auf das `gitolite-admin` Repository hat. Zusätzlich existiert eine Datei mit dem gleichen Namen. Diese beinhaltet den öffentlichen Schlüssel.

Neue Benutzer hinzuzufügen gestaltet sich einfach. Um einen neuen Anwender mit dem Namen „alice“ hinzuzufügen, benötigst Du ihren öffentlichen Schlüssel. Diesen nennst Du in `alice.pub` um und legst ihn im Verzeichnis `keydir` Deines geklonten `gitolite-admin` Repositorys auf Deiner Workstation ab. Danach stagst Du die Änderungen, commitest diese und pushst sie auf den Server und voilà, der Benutzer wurde hinzugefügt.

Die Syntax der Gitolite Konfigurationsdateien ist gut dokumentiert, deshalb gehen wir hier nur auf die wichtigsten Details ein.

Einzelne Benutzer oder Repositorys können zur besseren Verwaltung zu Gruppen zusammengefasst werden. Die Gruppennamen verhalten sich wie Makros. Beim Anlegen ist es

unabhängig, ob es sich um Projekte oder Benutzer handelt. Diese Festlegung wird erst getroffen, wenn diese „Makros“ verwendet werden.

```
@oss_repos      = linux perl rakudo git gitolite
@secret_repos   = fenestra pear

@admins         = scott
@interns        = ashok
@engineers      = sitaram dilbert wally alice
@staff          = @admins @engineers @interns
```

Du kannst die Berechtigungen auf „ref“-Ebene (Branches und Tags) festlegen. Im folgenden Beispiel darf die Gruppe „interns“ nur auf den „int“ Branch pushen. Die Benutzer der Gruppe „engineers“ können jeden Branch pushen, der mit dem Prefix „eng-“ beginnt. Zusätzlich kann diese Gruppe jeden Tag, mit dem Namen „rc“, gefolgt von einer einzelnen Zahl, pushen. Die Benutzer der Gruppe „admins“ können jede Operation für jeden „ref“ durchführen (inklusive Rewind-Operationen).

```
repo @oss_repos
  RW int$          = @interns
  RW eng-          = @engineers
  RW refs/tags/rc[0-9] = @engineers
  RW+              = @admins
```

Der Ausdruck hinter RW oder RW+ ist ein regulärer Ausdruck (Regex), gegen den die Referenzen (ref), die gepusht werden, verglichen werden. Wir nennen das auch „Refex“. Mit einem solchen Refex hat man ein sehr mächtiges Werkzeug an der Hand und kann noch viel mehr machen, als hier aufgezeigt ist. Aus diesem Grund solltest Du es aber damit auch nicht übertreiben, wenn Du mit den regulären Ausdrücken aus Perl nicht vertraut bist.

Wie Du vielleicht bereits vermutest hast, stellt Gitolite den Ausdruck refs/heads/ den Refex voran, wenn diese nicht mit refs/ beginnen.

Ein wichtige Eigenschaft der Syntax der Konfigurationsdatei ist, dass nicht alle Regeln für ein Repository an einer gemeinsamen Stelle festgehalten werden müssen. Du kannst die ganzen allgemeingültigen Dinge, wie zum Beispiel die oben gezeigten Regeln für alle oss_repos, an einer Stelle zusammenfassen und später dann spezifische Regeln für die einzelnen Fälle festlegen. Zum Beispiel folgendermaßen:

```
repo gitolite
  RW+              = sitaram
```

Diese Regel gehört dann zum Regelsatz des gitolite Repository.

An dieser Stelle fragst Du Dich vielleicht, wie die Zugriffsregeln eigentlich angewandt werden. Lass uns das kurz anschauen.

Es gibt zwei Ebenen für die Zugriffsberechtigung in Gitolite. Die erste befindet sich auf Repository Ebene. Wenn Du Lese- oder Schreibzugriff auf jede Ref in einem Repository hast, dann

kannst Du damit das ganze Repository sowohl lesen, als auch schreiben. Gitosis kennt nur diese Art der Zugriffsberechtigung.

Die zweite Ebene bezieht sich auf Branches oder Tags innerhalb eines Repositorys. Auf dieser Ebene kann allerdings nur der Schreibzugriff beschränkt werden. Der Benutzername, die Art des Zugriffs (w oder +) und der Refname, der aktualisiert wird, sind bekannt. Gitolite prüft, ob einer dieser Regeln auf diese Kombination zutrifft (hierbei ist allerdings zu beachten, dass der Refname mit dem regulären Ausdruck verglichen und kein eins zu eins String-Vergleich durchgeführt wird). Die Zugriffsregeln werden entsprechend der Reihenfolge innerhalb der Konfigurationsdatei abgearbeitet. Wenn eine Kombination zutrifft, kann der Push durchgeführt werden. Trifft keine zu, dann wird der Push verweigert.

Erweiterte Zugriffsberechtigungen mit „deny“ Regeln

Bis jetzt waren alle vorgestellten Berechtigungen entweder R, RW, oder RW+. Gitolite kennt aber noch eine weitere Berechtigung: -, welche für „deny“, also ablehnen steht. Dies gibt Dir noch viel mehr Möglichkeiten, allerdings auf Kosten der Komplexität, denn ab jetzt ist ein Fallthrough nicht die einzige Möglichkeit, wie ein Zugriff auf das Repository abgelehnt wird. Das heißt, die Reihenfolge der aufgestellten Regeln, hat auch eine Bedeutung.

Nehmen wir mal an, dass bei unserem bekannten Beispiel, die Gruppe „engineers“ auf alle Branches, außer master und integ, Rewind-Rechte haben soll. Das können wir folgendermaßen erreichen:

RW	master integ	=	@engineers
-	master integ	=	@engineers
RW+		=	@engineers

Noch einmal zur Wiederholung, man muss jede einzelne Regel von oben nach unten durchgehen und überprüfen, ob eine Regel auf den aktuellen Zugriffsmodus zutrifft oder ob eine Deny-Regel den Zugriff verhindert. Ein Push auf den Branch master oder integ, welcher nicht einem Rewind Push entspricht, wird durch die erste Regel erlaubt. Ein Rewind Push auf diese Refs trifft also auf die erste Regel nicht zu. Deshalb wird die zweite Regel geprüft und auf Grund der Deny-Regel wird der Push verweigert. Jeder Push (unabhängig, ob es sich um einen Rewind Push oder einen normalen Push handelt) auf eine Ref, welche nicht master oder integ entspricht, trifft nicht auf einer der beiden ersten Regeln zu, und wird damit auf Grund der dritten Regel erlaubt.

Ein Push auf Basis von Dateiänderungen einschränken

Neben der Zugriffsbeschränkung auf Basis von Branches, kannst Du genauso verhindern, dass eine Änderung an einer bestimmten Datei gepusht wird. Beispielsweise ist ein Makefile (oder auch andere Programme) nicht dafür geeignet, dass es von jeder x-beliebigen Person geändert wird. Meist hängen von so einem Makefile viele Dinge ab oder vieles könnte schief laufen, wenn die Änderungen an der Datei nicht korrekt durchgeführt werden würden. Du kannst deshalb Gitolite folgendermaßen konfigurieren:


```
repo foo
  RW          =    @junior_devs @senior_devs

- VREF/NAME/Makefile =    @junior_devs
```

Alle Anwender von älteren Gitolite Versionen, die auf eine neue Gitolite Version wechseln, sollten darauf achten, dass sich die neue Version signifikant anders im Bezug auf dieses Feature verhält. Die Umstellungsanleitung (migration guide) weist hier auf weitere Details hin.

Personenbezogene Branches

Gitolite bietet mit den „personal branches“ (genauer „personal branch namespace“) eine weitere Eigenschaft, die im Unternehmensumfeld sehr hilfreich sein kann.

Jede Menge Codeänderungen in der Welt von Git passieren, weil jemand einen Pull-Request durchführt. Im Unternehmensumfeld ist ein anonymer Zugriff ein absolutes No-Go und oft kann eine Entwickler Workstation keine Authentifizierung bieten. Deshalb müssen die Änderungen an den zentralen Server gepusht werden und jemand anders muss diese von dort abholen.

Dies würde normalerweise zu dem gleichen Branchnamen-Wirrwarr führen, wie es in zentralisierten Versionskontrollsystemen anzufinden ist. Außerdem wäre es für den Administrator äußerst lästig, die ganzen Berechtigungen dafür zu setzen.

Gitolite lässt die Definition eines „personal“ oder „scratch“ Namensraum für jeden einzelnen Entwickler zu (zum Beispiel: `refs/personal/<devname>/*`). Die Dokumentation enthält dazu weitere Details.

„Wildcard“ Repositorys

Mit Platzhaltern (eigentlich Perl reguläre Ausdrücke), wie zum Beispiel `assignments/s[0-9][0-9]/a[0-9][0-9]`, kannst Du in Gitolite auch Repositorys definieren. Außerdem bietet Gitolite einen neuen Berechtigungsmodus (c), welcher es den Benutzern ermöglicht, auf Basis dieser Platzhalter, Repositorys zu erzeugen. Dem Benutzer, der das Repository erzeugt hat, wird dieses automatisch zugewiesen, was es ihm oder ihr ermöglicht anderen Benutzern Lese- oder Schreibrechte (R und RW) zuzuweisen, damit diese zum Projekt beitragen können. Wieder möchte ich Dich darauf hinweisen, dass die Dokumentation weitere Details enthält.

Weitere Besonderheiten

Ich möchte das Thema Gitolite abschließen, indem ich noch ein paar weitere Besonderheiten kurz anspreche. Diese und viele weitere Features von Gitolite werden ausführlich in der Dokumentation beschrieben.

Protokollierung: Gitolite protokolliert alle Zugriffe, die erfolgreich waren. Wenn Du ein bisschen nachlässig bei der Vergabe von Rewind-Rechten (RW+) warst und irgendeiner der Personen mit Rewinde-Rechte dann den master zerstört, dann kann Dir die Protokolldatei eine Menge Arbeit

ersparen, weil sie Dir hilft, leicht und schnell die SHA Prüfsumme zu finden, die dem Erdboden gleich gemacht wurde.

Zugriffsrechte herausfinden: Ein anderes praktisches Merkmal von Gitolite lernst Du kennen, wenn Du versuchst Dich über SSH auf dem Server einzuloggen. Gitolite zeigt Dir dann alle Repositorys an, auf die Du Zugriff hast und welche Berechtigung Du für diese hast. Hierzu ein Beispiel:

```
hello scott, this is git@git running gitolite3 v3.01-18-g9609868 on git 1.7.4
```

```
R      anu-wsd
R      entrans
R  W   git-notes
R  W   gitolite
R  W   gitolite-admin
R      indic_web_input
R      shreelipi_converter
```

Administration aufteilen: Bei richtig großen Installationen kannst Du die Verantwortlichkeit für verschiedene Gruppen von Repositorys an verschiedene Leute verteilen, damit diese die Repositorys unabhängig verwalten können. Das macht das Leben des Haupt-Administrator leichter und verhindert, dass er der Flaschenhals im System ist.

Spiegelung: Gitolite kann Dir helfen verschiedene Mirrors (Spiegelserver) zu verwalten. Außerdem ist es damit einfach zwischen verschiedenen Mirrors zu wechseln, wenn der primäre Server offline ist.

Git Daemon

Wenn Du anonymen, öffentlichen Lesezugriff für Deine Repositorys zur Verfügung stellen willst, solltest Du Dir mal das Git Protokoll, als Ersatz für das HTTP Protokoll anschauen. Der Hauptgrund dafür ist die Geschwindigkeit. Das Git Protokoll ist weitaus effizienter und deshalb viel schneller als das HTTP Protokoll. Wenn Du Deinen Anwendern Zeit ersparen willst, solltest Du es zur Verfügung stellen.

Ich möchte Dich noch mal darauf hinweisen, dass das Git Protokoll nur für anonymen (also ohne Authentifizierung) Lesezugriff geeignet ist. Wenn Du einen öffentlichen Server betreibst, sollte das Git Protokoll nur für Projekte eingesetzt werden, die öffentlich für den Rest einsehbar sein sollen. Innerhalb Deines eigenen Netzwerks, welches mit einer Firewall abgeschottet ist, ist es sinnvoll, da Du mit dem Git Protokoll einer großen Anzahl von Benutzern und Computern (Continuous Integration oder Build-Server), Lesezugriff zur Verfügung stellen kannst, ohne dass Du für jeden einzelnen Nutzer ein SSH Schlüssel verwalten musst.

Auf jeden Fall ist es sehr einfach das Git Protokoll einzurichten. Im Prinzip musst Du nur folgendes tun:

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

Mit `--reuseaddr` teilst Du dem Server mit, dass ein Neustart sofort durchgeführt werden kann, ohne darauf zu warten, dass alte, offene Verbindungen mit einem Timeout abbrechen. Die Option `--base-path` erlaubt es den Nutzern, Projekte zu klonen, ohne den gesamten Pfad angeben zu müssen. Die Pfadangabe als letztes Argument gibt dem Daemon an, wo sich die zu exportierenden Repositorys befinden. Wenn Du eine Firewall eingerichtet hast, musst Du zusätzlich den Port 9418 freischalten.

Du kannst den Hintergrunddienst für diesen Prozess auf verschiedene Art und Weise einrichten. Das ist natürlich abhängig vom verwendeten Betriebssystem. Auf einem Ubuntu System kannst Du dazu ein Upstart Skript verwenden. Zum Beispiel fügst Du in der folgenden Datei

```
/etc/event.d/local-git-daemon
```

das folgende Skript ein (Achtung: In neueren Ubuntu-Versionen lautet der Pfad `/etc/init`):

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

Aus Sicherheitsgründen wird dringend empfohlen, dass dieser Daemon als Benutzer ausgeführt wird, welcher nur Lesezugriff auf die betreffenden Repositorys hat. Du stellst das auf einfache Art

und Weise sicher, indem Du einen neuen Benutzer ‚git-ro‘ erstellst und den Daemon mit diesem ausführst. Einfachheitshalber verwenden wir hier den Benutzer ‚git‘, den wir auch schon für Gitosis verwendet haben.

Nach einem Neustart des System wird der Git Daemon automatisch starten. Er startet ebenso neu, wenn er unerwartet beendet wird. Der Daemon kann auch ohne einen Neustart gestartet werden:

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

Auf anderen Systemen kannst Du `xinetd`, ein Skript in der `sysvinit`-Umgebung oder irgendetwas anderes verwenden. Du musst nur sicherstellen, dass der Befehl als Hintergrunddienst ausgeführt wird.

Als nächstes musst Du dem Gitosis Server mitteilen, für welche Repositorys ein anonymer Zugriff über das Git Protokoll möglich sein soll. Für jedes einzelne Repository kannst Du individuell festlegen, ob der Git Daemon auf dieses Zugriff haben soll. Wenn Du beispielsweise das Git Protokoll für das Projekt `iphone_project` erlauben willst, kannst Du folgendes am Ende der Konfigurationsdatei `gitosis.conf` einfügen:

```
[repo iphone_project]
daemon = yes
```

Nachdem Du dies eingecheckt und gepusht hast, sollte Dein im Hintergrund laufender Git Daemon die Anfragen aller Benutzer, die Zugriff auf den Port 9418 haben, bearbeiten.

Wenn Du Dich gegen Gitosis entschieden hast, aber trotzdem dem Git Daemon verwenden willst, musst Du auf dem Server für jedes Projekt, welches der Git Daemon zur Verfügung stellen soll, folgendes ausführen:

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

Wenn diese Datei existiert, erlaubt Git einen anonymen Lesezugriff auf dieses Projekt.

In Gitosis kann man ebenso einstellen, welche Projekte in GitWeb dargestellt werden sollen. Dazu musst Du als erstes in etwa folgendes in die Konfigurationsdatei `/etc/gitweb.conf` einfügen:

```
$projects_list = "/home/git/gitosis/projects.list";
$projectroot = "/home/git/repositories";
$export_ok = "git-daemon-export-ok";
@git_base_url_list = ('git://gitserver');
```

Um für die einzelnen Projekte festzulegen, dass diese in GitWeb auftauchen, musst Du die Einstellung `gitweb` in der Gitosis Konfigurationsdatei festlegen. Wenn Du beispielsweise willst, dass das Repository `iphone_project` in GitWeb erscheint, muss die Einstellung `repo` in etwa folgendermaßen aussehen:

```
[repo iphone_project]
daemon = yes
```

gitweb = yes

Das Projekt `iphone_projekt` wird automatisch in GitWeb angezeigt, sobald Du die Änderungen eingchecked und auf den Server gepusht hast.

Git Hosting

Wenn Du Dir die ganze Arbeit sparen willst, die beim Einrichten eines eigenen Git Servers so anfällt, kannst Du auch einen der vielen Hosting-Anbieter benutzen, um Deine Git Projekte zu verwalten. Wenn Du Dich für diese Option entscheidest, hat das gewisse Vorteile: Die Konfiguration bei den Hosting-Anbietern ist meist sehr schnell durchgeführt und Du kannst sofort mit Deinen Projekten loslegen. Zusätzlich ersparst Du Dir die Wartung und Überwachung Deines eigenen Servers. Auch wenn Du für Deine privaten oder firmeninternen Projekte einen eigenen Server betreibst, sind solche Hosting-Anbieter nützlich, da Du diese dann für Deine Open-Source Projekte verwenden kannst. Dadurch wirst Du innerhalb der Open-Source Community leichter gefunden und es ist einfacher Dir bei Deinen Projekten zu helfen.

Heutzutage stehen Dir viele Anbieter zur Auswahl. Jeder hat seine Vor- und Nachteile. Eine aktuelle Liste von Anbietern findest Du auf der folgenden Seite:

<https://git.wiki.kernel.org/index.php/GitHosting>

Da wir nicht alle Anbieter vorstellen können und ich zufälligerweise bei einem der Anbieter, nämlich GitHub, arbeite, werde ich in diesem Kapitel auf diese Plattform näher eingehen. Wir werden die Erstellung eines Accounts und die Erzeugung eines Projekts auf GitHub besprechen. Das sollte Dir einen leichten Einstieg in die Welt von GitHub ermöglichen.

GitHub ist die mit Abstand größte Open-Source Git Hosting Plattform und bietet als einer der wenigen, sowohl öffentliche und private Hosting-Optionen. Das erlaubt es Dir, Deine Open-Source Projekte und proprietären Code in einer einzelnen Plattform zu verwalten. Sogar für dieses Buch haben wir GitHub benutzt, um gemeinsam unter Ausschluss der Öffentlichkeit daran zu arbeiten.

GitHub

GitHub verwaltet und gruppiert Projekte ein wenig anders ein, als andere Code-Hosting Webseiten. GitHub fokussiert sich dabei nicht speziell auf die Projekte, sondern eher auf den Anwender. Dazu ein Beispiel. Wenn ich mein Projekt `grit` auf GitHub einstelle, dann findet man dieses nicht unter `github.com/grit`, sondern unter `github.com/schacon/grit`. Es gibt in diesem Sinne, keine in Stein gemeißelte Stelle an dem das Projekt verwaltet wird. Das bedeutet, man kann ein Projekt nahtlos von einem Benutzer zu einem anderen Benutzer übertragen, wenn dieser zum Beispiel das Projekt aufgibt.

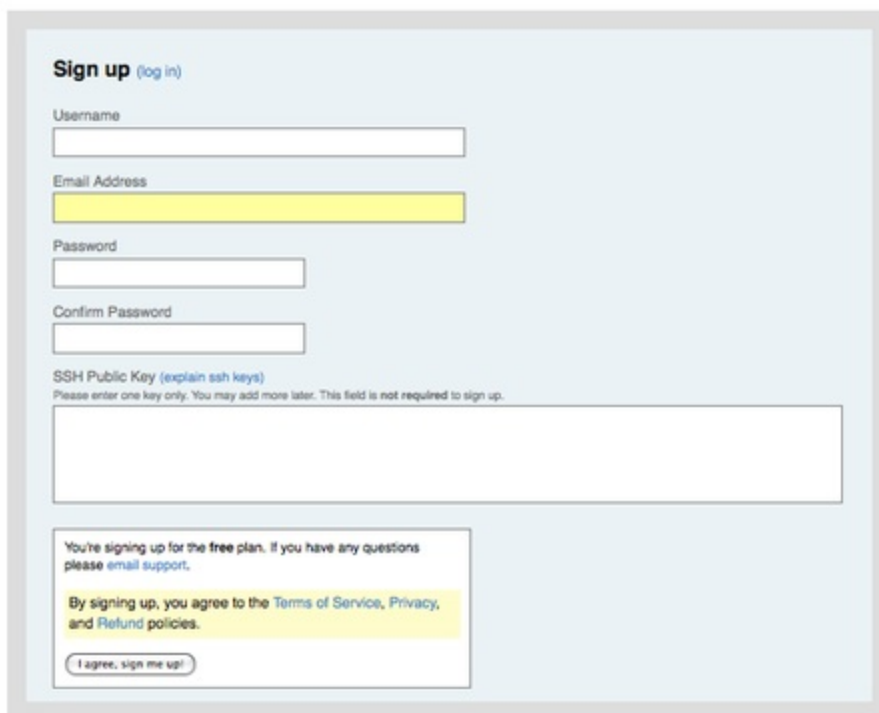
Wenn ein Anwender ein geschütztes, nicht öffentliches Repository auf GitHub verwalten will, so muss er dafür bezahlen. Damit verdient die Firma GitHub ihr Geld. Aber die Verwaltung und Nutzung für Open-Source Projekte ist kostenlos. Die Anzahl der Projekte ist dabei nicht beschränkt. Der Nutzer muss dazu lediglich einen Account erstellen. Wie das geht, möchte ich hier kurz vorstellen.

Einrichten eines Benutzeraccounts

Um loslegen zu können, musst Du Dir einen Benutzeraccount erstellen. Gib dazu die Adresse `http://github.com/plans` in Deinem Browser ein und wähle den Button „Sign Up“ unter dem „Free account“-Bereich aus (siehe Abbildung 4-2). Danach wirst Du auf die Anmeldeseite weitergeleitet.



Auf dieser Seite musst Du einen Nutzernamen auswählen, der bisher im System nicht vorhanden ist. Zusätzlich musst Du Deine E-Mail Adresse angeben, die mit Deinem Account verknüpft wird, und ein Passwort angeben (siehe Abbildung 4-3).



Wenn Du Deinen öffentlichen SSH Schlüssel zur Hand hast, kannst Du diesen auch gleich bei der Registrierung angeben. Die Vorgehensweise zum Generieren eines Schlüssels haben wir bereits im Kapitel 4.3 besprochen. Du musst den Inhalt der öffentlichen Schlüsseldatei kopieren und in das „SSH Public Key“ Formularfeld einfügen. Wenn Du auf den „explain ssh keys“ Link klickst, erhältst Du detaillierte Anweisungen zum Ausführen dieses Vorgangs auf verschiedenen Betriebssystemen. Wenn Du auf den „I agree, sign me up“ Button drückst, landest Du in Deinem neuen Benutzer-Dashboard (siehe Abbildung 4-4).



Im nächsten Schritt kannst Du ein neues Repository erzeugen.

Erzeugen eines neuen Repository

Um ein neues Repository anzulegen, musst Du auf den „create a new one“-Link, welcher neben Deinen Repositories auf dem Benutzer-Dashboard angezeigt wird, klicken. Du landest daraufhin im Formular zum Erzeugen eines neuen Repositories (siehe Abbildung 4-5).

The screenshot shows the 'Create a New Repository' form. The title is 'Create a New Repository'. Below the title is a description: 'Create a new empty repository into which you can push your local git repo.' A note in red text says: 'NOTE: If you intend to push a copy of a repository that is already hosted on GitHub, then you should fork it instead.' The form has three input fields: 'Project Name' (containing 'iphone_project'), 'Description' (containing 'iphone project for our mobile group'), and 'Homepage URL' (empty). Below the input fields is a section for 'Who has access to this repository? (You can change this later)'. It has two radio buttons: 'Anyone (learn more about public repos)' (selected) and 'Upgrade your plan to create more private repositories!'. At the bottom is a 'Create Repository' button.

Im Prinzip musst Du nur einen Projektnamen und wenn Du es für nötig erachtest eine Beschreibung Deines Projekts angeben. Wenn das erledigt ist, kannst Du auf den „Create Repository“ Button klicken. Du hast soeben Dein erstes Repository auf GitHub erzeugt (siehe Abbildung 4-6).



Da in dem Repository noch kein Code enthalten ist, gibt Dir GitHub ein paar Hinweise, wie Du ein neues Projekt anlegst, wie Du ein bereits vorhandenes Git Projekt auf GitHub pusht oder wie man ein bestehendes Subversion Repository in GitHub importieren kann (siehe Abbildung 4-7).

Global setup:

Download and install [Git](#)
`git config --global user.email test@github.com`

Next steps:

```
mkdir iphone_project
cd iphone_project
git init
touch README
git add README
git commit -m 'first commit'
git remote add origin git@github.com:testinguser/iphone_project.git
git push origin master
```

Existing Git Repo?

```
cd existing_git_repo
git remote add origin git@github.com:testinguser/iphone_project.git
git push origin master
```

Importing a SVN Repo?

[Click here](#)

When you're done:

[Continue](#)

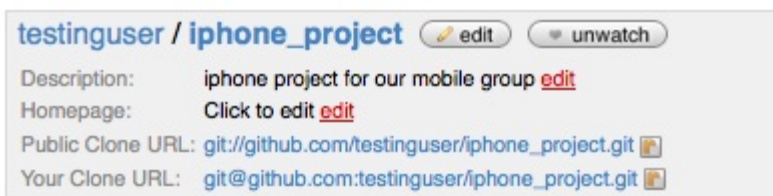
Die Hilfestellung entspricht der Vorgehensweise, die ich bereits in diesem Buch vorgestellt habe. Um ein neues Git Projekt in einem vorhandenen Verzeichnis anzulegen, kannst Du die folgenden Befehle verwenden:

```
$ git init
$ git add .
$ git commit -m 'initial commit'
```

Wenn Du bereits ein lokales Git Repository auf Deinem PC hast, kannst Du GitHub als zusätzlichen Remote hinzufügen und den master Branch pushen:

```
$ git remote add origin git@github.com:testinguser/iphone_project.git
$ git push origin master
```

Das war es schon. Dein Projekt ist nun auf GitHub erreichbar. Du kannst jetzt die zugehörige URL an jeden weitergeben, den Du am Projekt teilhaben lassen willst. In unserem Beispiel lautet der Link hierfür http://github.com/testinguser/iphone_project. Im oberen Header-Bereich jeder Projektseite werden zwei verschiedene Git URLs angezeigt (siehe Abbildung 4-8).

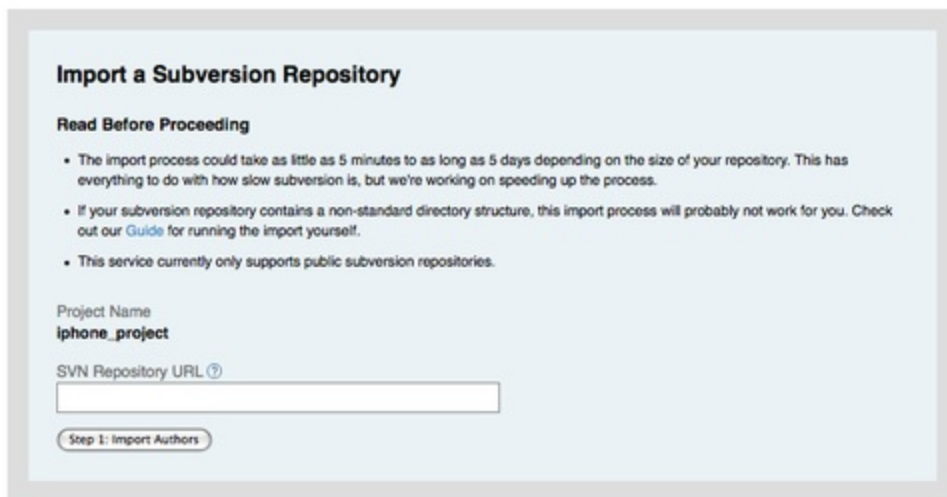


Die „Public Clone URL“ ist eine öffentliche Git URL, die man zum Klonen des Projekts verwenden kann. Über diese URL kann lediglich lesend zugegriffen werden. Ein Schreibzugriff ist nicht möglich. Du kannst diese URL beliebig weiter verteilen und zum Beispiel auch auf Deiner Homepage oder einem anderen Medium veröffentlichen.

Die „Your Clone URL“ ist eine auf SSH basierte URL, mit Hilfe derer, vom Projekt gelesen, als auch geschrieben werden kann. Diese URL kann aber nur der Anwender nutzen, der im Besitz des privaten Schlüssels ist, welcher zu dem öffentlichen Schlüssel gehört, der bei dem GitHub Benutzer für dieses Projekt hinterlegt ist (Du hast den öffentlichen Schlüssel bei der Registrierung Deines Accounts angegeben).. Dieser Link wird allerdings nur Dir angezeigt. Andere Benutzer, die Deine Projekte besuchen, können nur die „Public Clone URL“ sehen.

Import von Subversion

Wenn Du bereits ein Subversion Projekt hast und dieses in Git importieren möchtest, kann GitHub Dir diese Aufgabe in vielen Fällen übernehmen. Am Ende der Seite mit den Hilfestellungen ist ein Link, der Dich auf die Seite mit dem Formular zum Importieren eines Subversion Projekts weiterleitet. Du musst in diesem Formular nur die URL des Subversion Projekts angeben (siehe Abbildung 4-9).



The screenshot shows the 'Import a Subversion Repository' page on GitHub. It has a light blue header with the title 'Import a Subversion Repository'. Below the title is a section 'Read Before Proceeding' with three bullet points: 'The import process could take as little as 5 minutes to as long as 5 days depending on the size of your repository. This has everything to do with how slow subversion is, but we're working on speeding up the process.', 'If your subversion repository contains a non-standard directory structure, this import process will probably not work for you. Check out our [Guide](#) for running the import yourself.', and 'This service currently only supports public subversion repositories.' Below this is a form with two fields: 'Project Name' with the value 'iphone_project' and 'SVN Repository URL' with a help icon. At the bottom is a button labeled 'Step 1: Import Authors'.

Wenn Dein Projekt sehr groß, nicht standardkonform oder nicht öffentlich einsehbar ist, kann es passieren dass dieser Vorgang fehlschlägt. In Kapitel 7 liefere ich die Antwort, wie ein solcher Import, manuell durchgeführt werden kann.

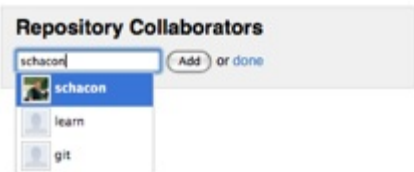
Mitarbeiter hinzufügen

Um gemeinsam an einem Projekt zu arbeiten, kannst Du auch andere Personen für das Projekt freischalten. Wenn Du willst das John, Josie und Jessica ebenso auf das Projekt pushen können, musst Du sie als Mitarbeiter für Dein Projekt freischalten. Voraussetzung hierfür ist natürlich, dass Sie alle einen GitHub Account besitzen. Nachdem Du sie zum Projekt hinzugefügt hast, können sie unter Verwendung ihrer öffentlichen Schlüssel auf das Projekt pushen.

Klicke auf die „edit“-Schaltfläche in der Projektübersicht oder wähle den Admin-Tab im oberen Bereich Deines Projekts um zur Administrationsoberfläche zu gelangen (siehe Abbildung 4-10).



Um einem anderen Benutzer Schreibrechte zu Deinem Projekt zu gewähren, kannst Du auf den „Add another collaborator“-Link klicken. Daraufhin erscheint ein Eingabefeld, in welches Du die Benutzer eingeben kannst. Während des Tippens, erscheint ein kleines Popup, welches Benutzernamen anzeigt, die Deiner Eingabe entsprechen. Wenn Du den gewünschten Benutzer gefunden hast, kannst Du die „Add“-Schaltfläche betätigen, um diesen Benutzer als Mitarbeiter zu Deinem Projekt hinzuzufügen (siehe Abbildung 4-11).



Wenn Du Dein Team fertig zusammengestellt hast, solltest Du eine Liste aller Mitarbeiter im „Repository Collaborators“-Bereich sehen (siehe Abbildung 4.12).



Wenn Du einer Person den Zugriff auf Dein Repository entziehen willst, kannst Du auf den „revoke“-Link klicken. Dadurch kann diese Person nicht mehr auf Dein Repository pushen. Für zukünftige Projekte kannst Du die Liste der Benutzer auch für andere Projekte übernehmen.

Dein Projekt

Nachdem Du das erste mal auf das GitHub Repository gepusht hast oder es von Subversion importiert hast, sieht die Hauptseite Deines GitHub-Projekts entsprechend Abbildung 4-13 aus.

The screenshot shows the GitHub interface for a repository named 'testinguser / iphone_project'. At the top, there's a navigation bar with the GitHub logo, a search bar, and links for Home, Pricing and Signup, Repositories, Blog, and Login. Below this, there are tabs for Source, Commits, Network (1), Downloads (0), Wiki (1), and Graphs. The 'Source' tab is active, showing the repository's file structure. The repository description is 'iphone project for our mobile group' and the clone URL is 'git://github.com/testinguser/iphone_project.git'. The 'initial commit' section shows a commit by 'schocon' (author) 5 days ago. Below this, a table lists the files in the repository, all of which were added in the initial commit.

name	age	message	history
Classes/	5 days ago	initial commit [schocon]	
Info.plist	5 days ago	initial commit [schocon]	
MainWindow.xib	5 days ago	initial commit [schocon]	
build/	5 days ago	initial commit [schocon]	
iGit.xcodeproj/	5 days ago	initial commit [schocon]	
iGitViewController.xib	5 days ago	initial commit [schocon]	
iGit_Prefix.pch	5 days ago	initial commit [schocon]	
main.m	5 days ago	initial commit [schocon]	

Wenn andere Dein Projekt in GitHub aufrufen, sehen sie als erstes diese Hauptseite. Die verschiedenen Funktionen, die GitHub für ein Projekt unterstützt, sind in verschiedene Tabs aufgeteilt. Der „Commit“-Tab enthält eine Liste aller Commits in chronologischer Reihenfolge. Dabei steht der neueste Commit ganz oben. Eine ähnliche Liste erhältst Du, wenn Du den `git log` Befehl ausführst. Der „Network“-Tab zeigt alle Benutzer an, die ein Fork von Deinem Projekt erstellt haben und ihren Teil zum Projekt beigetragen haben. Im „Download“-Tab kannst Du Binärdateien hochladen oder Zip-Archive beziehungsweise Tarballs zur Verfügung stellen, die einem bestimmten Tag Deines Projekts entsprechen. Im „Wiki“-Tab findest Du ein Wiki, welches Du zu Dokumentationszwecken verwenden kannst. Außerdem kannst Du dort andere Informationen über Dein Projekt der Welt mitteilen. Der „Graphs“-Tab stellt Dir eine grafische Übersicht zur Verfügung, die Dir anzeigt, wer und wann jemand etwas zu Deinem Projekt beigetragen hat. Außerdem enthält dieser Tab eine Projekt-Statistik. Der „Source“-Tab, welcher standardmäßig beim Aufruf Deines Projekts angezeigt wird, zeigt das oberste Verzeichnis Deines Repositorys an. Wenn Dein Projekt eine README-Datei enthält, wird der Inhalt dieser Datei unterhalb der Verzeichnisstruktur angezeigt. Zusätzlich zeigt dieser Tab eine Übersicht mit den letzten Commit-Informationen an.

Fork von einem Projekt erstellen

Wenn Du an einem bereits vorhandenen Projekt mitarbeiten willst und Du zu diesem keine Schreibrechte hast, bietet Dir GitHub die Möglichkeit einen Fork von diesem Projekt zu erstellen. Wenn Du beim Stöbern durch GitHub bei einem interessanten Projekt landest und Du damit ein bisschen spielen willst, kannst Du die „Fork“-Schaltfläche im Projekt-Header auswählen. GitHub kopiert das gesamte Projekt in Deinen Benutzerbereich. Auf dieses kannst Du jetzt auch pushen.

Dadurch dass jeder, jedes Projekt kopieren kann, muss sich der Verwalter eines Projekts nicht darum kümmern, dass jedem Mitarbeiter die entsprechenden Schreibrechte erhält. Man kann einfach einen Fork erstellen und auf diesen pushen. Der Verwalter des geforkten Projekts kann dieses neue Projekt als neuen Remote hinzufügen und die Änderungen davon holen. Dann kann er

diese Änderungen in das Hauptprojekt mergen.

Um einen Fork von einem Projekt zu erstellen, kannst Du die jeweilige Projektseite besuchen (in diesem Fall `mojombo/chronic`) und die „Fork“-Schaltfläche im oberen Bereich auswählen (siehe Abbildung 4-14).



Nach ein paar Sekunden wirst Du zu der neuen Projektseite weitergeleitet. Dort siehst Du auch, dass dieses Projekt ein Fork eines anderen Projekts ist (siehe Abbildung 4-15).



GitHub Zusammenfassung

Das war alles was ich über GitHub berichten möchte. Herausheben möchte ich aber, dass sich die Arbeit mit GitHub sehr einfach gestaltet. Innerhalb kürzester Zeit kannst Du Dir einen Account einrichten, ein neues Projekt hinzufügen und auf dieses pushen. Wenn Dein Projekt ein Open-Source Projekt ist und damit öffentlich einsehbar ist, steht Dir mit GitHub eine riesige Community mit zahlreichen Entwicklern zur Seite, die sich an Deinem Projekt beteiligen können, indem sie einen Fork erstellen. Vielleicht gelingt Dir mit GitHub der Einstieg in die Welt von Git noch einfacher.

Zusammenfassung

Es stehen Dir viele Möglichkeiten offen, ein Remote-Repository einzurichten, sodass Du mit anderen Entwicklern zusammenarbeiten kannst und Deine Arbeit anderen zur Verfügung stellen kannst.

Wenn Du Deinen eigenen Server betreibst, stehen Dir natürlich alle Möglichkeiten offen und Du kannst ihn entsprechend Deiner Anforderungen konfigurieren. Dennoch darf man den Zeitaufwand zum Aufsetzen und Warten des Servers nicht unterschätzen. Wenn Du Deine Repositorys bei einem Hosting-Anbieter verwaltest, hält sich der Aufwand für die Einrichtung und Wartung in Grenzen, allerdings vertraust Du Deine Daten auch einem fremden Anbieter an, was manche Firmen oder Organisationen untersagen.

Es sollte Dir nicht schwerfallen eine passende Lösung für Dich, Deine Firma oder Deine Organisation zu finden.

Distribuierte Arbeit mit Git (xxx)

Du hast jetzt ein externes Repository aufgesetzt, sodass alle Mitglieder des Teams ihren Code zur Verfügung stellen können, und Du hast Dich mit den wesentlichen Git Befehlen für die Arbeit in einem lokalen Repository vertraut gemacht. Als nächstes werden wir uns einige Arbeitsabläufe für distribuierte Repositories ansehen, die Git Dir ermöglicht.

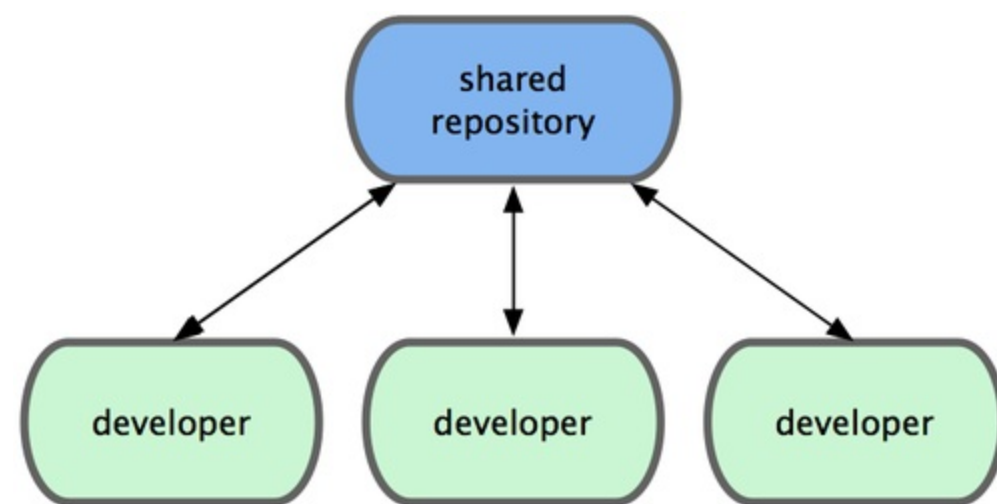
In diesem Kapitel wirst Du lernen, wie Du in einer distribuierten Umgebung als Entwickler Code zu einem Projekt beisteuern, als für Verantwortlicher Code von anderen ins Projekt übernehmen kannst und wie Du das so gestalten kannst, dass es in einem Projekt mit einer großen Anzahl von Entwicklern für alle Beteiligten möglichst einfach ist.

Distribuierte Workflows

Anders als in zentralisierten Versionskontrollsystemen (CVCS) ermöglicht die Distribuietheit von Git eine sehr viel flexiblere Zusammenarbeit von Entwicklern. In zentralisierten Systemen fungieren alle Beteiligten als gleichwertige Netzknoten, die in mehr oder weniger der gleichen Weise am zentralen Knotenpunkt (dem zentralen Repository) arbeiten. In Git dagegen ist jeder Beteiligte selbst potentiell zentraler Knotenpunkt. D.h. jeder Entwickler kann sowohl Code zu anderen Repositories beitragen und ein öffentliches Repository zur Verfügung stellen, an dem wiederum andere mitarbeiten. Das ermöglicht eine riesige Anzahl von Möglichkeiten, Arbeitsabläufe zu gestalten, die auf das jeweilige Projekt und/oder Team perfekt zugeschnitten sind. Wir werden auf einige übliche Paradigmen, die diese Flexibilität nutzen, und deren Vor- und Nachteile eingehen. Du kannst daraus ein Modell auswählen, oder Du kannst sie miteinander kombinieren, um sie an Deine eigenen Erfordernisse anzupassen.

Zentralisierter Workflow

In einem zentralisierten System gibt es grob gesagt ein einziges Modell der Zusammenarbeit. Ein zentraler Knotenpunkt (oder Repository) kann Code von anderen akzeptieren und übernehmen, und alle Beteiligten synchronisieren ihre Arbeit damit. Entwickler fungieren als Knoten, die ihre Arbeit an diesem einen, zentralen Punkt synchronisieren (siehe Bild 5-1).



Das heißt: wenn zwei Entwickler Code aus dem zentralen Repository abholen und beide Änderungen vornehmen, dann kann der erste Entwickler seine Änderungen ohne Probleme im zentralen Repository abliefern. Der zweite Entwickler muss sie zunächst mit den Änderungen des ersten Entwicklers zusammenführen, damit er dessen Arbeit nicht überschreibt. Dieses Konzept trifft sowohl auf Git als auch auf Subversion (und jedes andere CVCS) zu, und es funktioniert in Git perfekt.

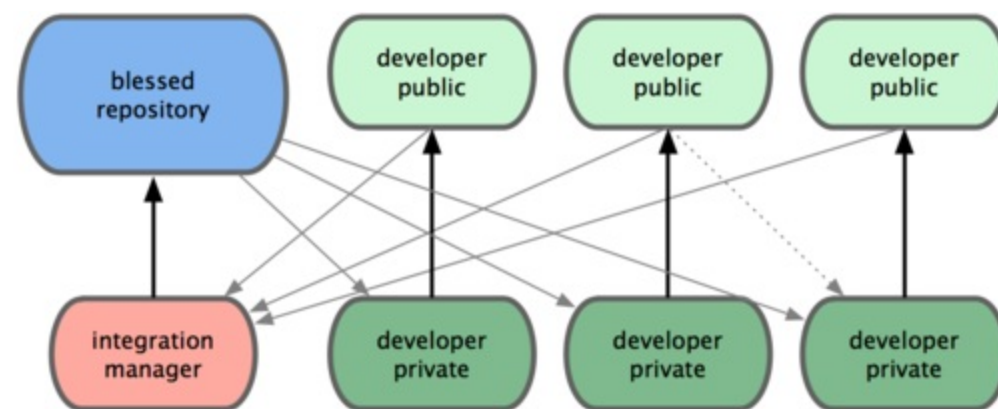
In einem kleinen Team oder einem Team, das mit einem zentralisierten Workflow zufrieden ist, kann man diesen Workflow ohne weiteres mit Git realisieren. Man setzt einfach ein einziges Repository auf und gibt jedem im Team Schreibzugriff („push access“). Git sorgt dann dafür, dass niemand die Arbeit von anderen überschreiben kann. Wenn ein Entwickler das Repository klonet, Änderungen vornimmt und dann versucht ins zentrale Repository zu pushen, obwohl jemand

anders in der Zwischenzeit Änderungen gepusht hat, dann wird der Server das zurückweisen. Dem Entwickler wird dann mitgeteilt, dass er versucht hat, sogenannte „non-fast-forward“ Änderungen hochzuladen und dass er zuvor die Änderungen des anderen Entwicklers herunterladen und mit seinen zusammenführen muss. Viele Leute mögen diesen Arbeitsablauf, weil sie mit dem Paradigma bereits vertraut sind und sich damit wohl fühlen.

Integration-Manager Workflow

Weil Git ermöglicht, eine Vielzahl von externen Repositories zu betreiben, ist es außerdem möglich, einen Arbeitsprozess zu gestalten, in dem jeder Entwickler Schreibzugriff auf sein eigenes öffentliches Repository hat, aber nur Lesezugriff auf die Repositories von allen anderen Beteiligten. In diesem Szenario stellt jedes Repository ein eigenes „offizielles“ Projekt dar. Um zu einem solchen distribuierten Projekt Änderungen beizusteuern, kannst Du einen eigenen, öffentlichen Klon des Projektes anlegen und Deine Änderungen dort publizieren. Anschließend kannst Du den Betreiber des Haupt-Repositories bitten, Deine Änderungen in sein Repository zu übernehmen. Er kann dann Dein Repository als ein externes Repository auf seinem Rechner einrichten, Deine Änderungen lokal testen, sie in einen seiner Branches (z.B. master) mergen und dann in sein öffentliches Repository pushen. Dieser Prozess läuft wie folgt ab (siehe Bild 5-2):

1. Der Projekt Betreiber pusht in ein öffentliches Repository.
2. Ein Mitarbeiter klonst das Repository und nimmt Änderungen daran vor.
3. Der Mitarbeiter pusht diese in sein eigenes öffentliches Repository.
4. Der Mitarbeiter schickt dem Betreiber eine E-Mail und bittet darum, die Änderungen zu übernehmen.
5. Der Betreiber richtet das Repository des Mitarbeiters als ein externes Repository ein und führt die Änderungen mit einem seiner eigenen Branches zusammen.
6. Der Betreiber pusht die zusammengeführten Änderungen in sein öffentliches Repository.

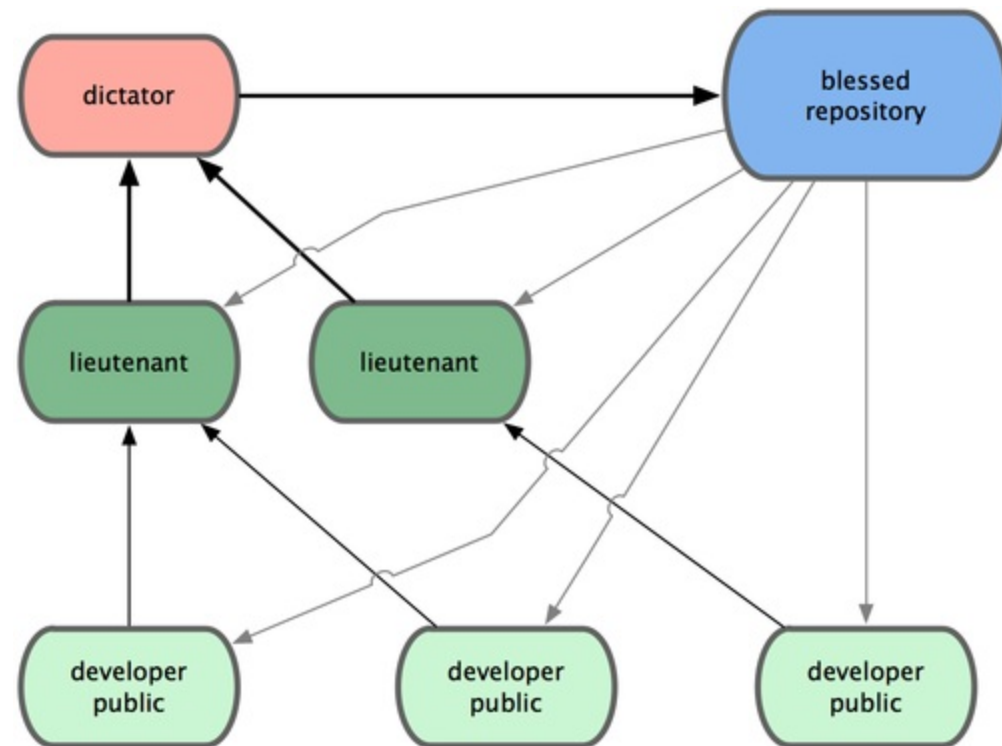


Dies ist ein weit verbreiteter Arbeitsablauf wie ihn z.B. auch GitHub ermöglicht, wo man ein Projekt auf sehr einfache Weise forken und seine Änderungen in seinen eigenen Fork pushen kann, um sie anderen zur Verfügung zu stellen. Einer der Hauptvorteile dieser Vorgehensweise ist, dass man an seinem Fork jederzeit weiterarbeiten, der Betreiber des Projektes Änderungen aber auch jederzeit übernehmen kann. Mitarbeiter müssen nicht darauf warten, dass der Betreiber Änderungen übernimmt – und jeder Beteiligte kann in seinem eigenen Rhythmus und Tempo arbeiten.

Diktator und Leutnants Workflow

Dies ist Variante eines Workflows mit zahlreichen Repositories, die normalerweise von sehr großen Projekten mit hunderten von Mitarbeitern verwendet wird. Das bekannteste Beispiel ist wahrscheinlich der Linux Kernel. In diesem Projekt sind zahlreiche Integration Manager, die „Leutnants“, für verschiedene Bereiche des Repositories zuständig. Für sämtliche Leutnants gibt es wiederum einen Integration Manager, der als der „wohlwollende Diktator“ („benevolent dictator“) bezeichnet wird. Das Repository des wohlwollenden Diktators fungiert als das Referenz-Repository aus dem alle Beteiligten ihre eigenen Repositories aktualisieren müssen. Dieser Prozess funktioniert also wie folgt (siehe Bild 5-3)

1. Normale Entwickler arbeiten in ihren Arbeitsbranches (xxx) und rebasen (xxx) ihre Änderungen auf der Basis des Master Branches. Der Master Branch ist derjenige des Diktators.
2. Die Leutnants mergen die Arbeitsbranches der Entwickler in ihre Master Branches.
3. Der Diktator merged die Master Branches der Leutnants mit seinem eigenen Master Branch zusammen.
4. Der Diktator pusht seinen Master Branch ins Referenz-Repository, sodass alle ihre Arbeit wiederum damit synchronisieren (rebasen) können.



Diese Art Workflow ist nicht unbedingt weit verbreitet, aber für große Projekte oder Projekte mit strikten hierarchischen Rollen sehr nützlich, weil der Projektleiter (der Diktator) Arbeit in großem Umfang delegieren und ganze Teilbereiche von Code von verschiedenen Endpunkten zusammensammeln und integrieren kann.

Wir haben jetzt einige übliche Workflows besprochen, die in einem distribuierten System wie Git möglich sind. Natürlich kann man sie mannigfaltig abwandeln und miteinander kombinieren, um sie an ein spezielles reales Projekt und Team anzupassen. Nachdem Du jetzt hoffentlich in der

Lage bist, Dir einen Workflow vorzustellen, der für Dich selbst Sinn macht, gehen wir auf einige etwas spezifischere Beispiele ein und darauf, wie man die verschiedenen Rollen umsetzen kann, die die Workflows ausmachen.

An einem Projekt mitarbeiten

Du kennst jetzt einige grundlegende Workflow Varianten, und Du solltest ein gutes Verständnis im Umgang mit grundlegenden Git Befehlen haben. In diesem Abschnitt lernst Du wie Du an einem Projekt mitzuarbeiten kannst.

Diesen Prozess zu beschreiben ist nicht leicht, weil es so viele Variationen gibt. Git ist so unheimlich flexibel, dass Leute auf vielen unterschiedlichen Wegen zusammenarbeiten können, und es ist problematisch, zu erklären, wie Du arbeiten *solltest*, weil jedes Projekt ein bisschen anders ist. Zu den Variablen gehören: die Anzahl der aktiven Mitarbeiter, der Workflow des Projektes, Deine Commit Rechte und möglicherweise eine vorgeschriebene, externe Methode, Änderungen einzureichen.

Wie viele Mitarbeiter tragen aktive Code zum Projekt bei? Und wie oft? In vielen Fällen findest Du zwei oder drei Entwickler, die täglich einige Commits anlegen, möglicherweise weniger in eher (xxx dormant xxx) Projekten. In wirklich großen Unternehmen oder Projekten können tausende Entwickler involviert sein und täglich dutzende oder sogar hunderte von Patches produzieren. Mit so vielen Mitarbeitern ist es aufwendiger, sicher zu stellen, dass Änderungen sauber mit der Codebase zusammengeführt werden können. Deine Änderungen könnten sich als überflüssig oder dysfunktional (xxx) erweisen, nachdem andere Änderungen übernommen wurden, seit Du angefangen hast, an Deinen eigenen zu arbeiten oder während sie darauf warteten, geprüft und eingefügt (xxx) zu werden.

Die nächste Variable ist der Workflow, der in diesem Projekt besteht. Ist es ein zentralisierter Workflow, in dem jeder Entwickler Schreibzugriff auf die Hauptentwicklungslinie hat? Hat das Projekt einen Leiter oder Integration Manager, der alle Patches prüft? Werden die Patches durch eine bestimmte Gruppe oder öffentlich, z.B. durch die Community, geprüft? Nimmst Du selbst an diesem Prozess teil? Gibt es ein Leutnant System, in dem Du Deine Arbeit zunächst an einen Leutnant übergibst?

Eine weitere Frage ist, welche Commit Rechte Du hast. Wenn Du Schreibrechte hast, sieht der Arbeitsablauf, mit dem Du Änderungen beisteuern kannst, natürlich völlig anders aus, als wenn Du nur Leserechte hast. Und in letzterem Fall: in welcher Form werden Änderungen in diesem Projekt akzeptiert? Gibt es dafür überhaupt eine Richtlinie? Wie umfangreich sind die Änderungen, die Du jeweils besteuerst? Und wie oft tust Du das?

Die Antworten auf diese Fragen können maßgeblich beeinflussen, wie Du effektiv an einem Projekt mitarbeiten kannst und welche Workflows Du zur Auswahl hast. Wir werden verschiedene Aspekte davon in einer Reihe von Fallbeispielen besprechen, wobei wir mit simplen Beispielen anfangen und später komplexere Szenarios besprechen. Du wirst hoffentlich in der Lage sein, aus diesen Beispielen einen eigenen Workflow zu konstruieren, der Deinen Anforderungen entspricht.

Commit Richtlinien

Bevor wir uns verschiedene konkrete Fallbeispiele ansehen, einige kurze Anmerkungen über

Commit Meldungen. Gute Richtlinien für Commit Meldungen zu haben und sich danach zu richten, macht die Zusammenarbeit mit anderen und die Arbeit mit Git selbst sehr viel einfacher. Im Git Projekt gibt es ein Dokument mit einer Reihe nützlicher Tipps für das Anlegen von Commits, aus denen man Patches erzeugen will. Schau im Git Quellcode nach der Datei `Documentation/SubmittingPatches`.

Zunächst einmal solltest Du keine Whitespace Fehler (xxx) comitten:

```
$ git diff --check
lib/simplegit.rb:5: trailing whitespace.
+   @git_dir = File.expand_path(git_dir)XX
lib/simplegit.rb:7: trailing whitespace.
+ XXXXXXXXXXXXX
lib/simplegit.rb:26: trailing whitespace.
+   def command(git_cmd)XXXX
```

Wenn Du diesen Befehl ausführst, bevor Du einen Commit anlegst, warnt er dich, falls in Deinen Änderungen Whitespace Probleme vorliegen, die andere Entwickler ärgern könnten.

Versuche außerdem, Deine Änderungen in logisch zusammenhängende Einheiten zu gruppieren. Wenn möglich, versuche Commits möglichst leichtverständlich (xxx) zu gestalten: arbeite nicht ein ganzes Wochenende lang an fünf verschiedenen Problemen und committe sie dann am Montag als einen einzigen, riesigen Commit. Selbst wenn Du am Wochenende keine Commits angelegt hast, verwende die Staging Area, um Deine Änderungen auf mehrere Commits aufzuteilen, jeweils mit einer verständlichen Meldung. Wenn einige Änderungen dieselbe Datei betreffen, probiere sie mit `git add --patch` nur teilweise zur Staging Area hinzuzufügen (das werden wir in Kapitel 6 noch im Detail besprechen). Der Projekt Snapshot wird am Ende derselbe sein, ob Du nun einen einzigen großen oder mehrere kleine Commits anlegst, daher versuche, es anderen Entwickler zu erleichtern machen, Deine Änderungen zu verstehen. Auf diese Weise machst Du es auch einfacher, einzelne Änderungen später herauszunehmen oder rückgängig zu machen. Kapitel 6 beschreibt eine Reihe nützlicher Git Tricks, die hilfreich sind, um die Historie umzuschreiben oder interaktiv Dateien zur Staging Area hinzuzufügen. Verwende diese Hilfsmittel, um eine sauber und leicht verständliche Historie von Änderungen aufzubauen.

Ein weitere Sache, der Du ein bisschen Aufmerksamkeit schenken solltest, ist die Commit Meldung selbst. Wenn man sich angewöhnt, aussagekräftige und hochwertige Commit Meldungen zu schreiben, macht man sich selbst und anderen das Leben erheblich einfacher. Im allgemeinen sollte eine Commit Meldung mit einer einzelnen Zeile anfangen, die nicht länger als 50 Zeichen sein sollte. Dann sollte eine leere Zeile folgen und schließlich eine ausführlichere Beschreibung der Änderungen.

Short (50 chars or less) summary of changes

More detailed explanatory text, if necessary. Wrap it to about 72 characters or so. In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the

two together.

Further paragraphs come after blank lines.

- Bullet points are okay, too
- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here

Wenn Du Deine Commit Meldungen in dieser Weise formatierst, kannst Du Dir und anderen eine Menge Ärger ersparen. Das Git Projekt selbst hat wohl-formatierte Commit Meldungen. Wir empfehlen, einmal `git log --no-merges` in diesem Repository auszuführen, um einen Eindruck zu erhalten, wie eine gute Commit History eines Projektes aussehen kann.

In den folgenden Beispielen hier und fast überall in diesem Buch verwende ich keine derartigen, schön formatierten Meldungen. Stattdessen verwende ich die `-m` Option zusammen mit `git commit`. Also folge meinen Worten, nicht meinem Beispiel.

Kleine Teams

Das einfachste Setup, mit dem Du zu tun haben wirst, ist ein privates Projekt mit ein oder zwei Entwicklern. Mit „privat“ meine ich, dass es „closed source“, d.h. nicht lesbar für Dritte ist. Alle beteiligten Entwickler haben Schreibzugriff auf das Repository.

In einer solchen Umgebung kann man einen ähnlichen Workflow verwenden, wie für Subversion oder ein anderes zentralisiertes System. Du hast dann immer noch Vorteile wie, dass Du offline committen kannst und dass Branching und Merging so unglaublich einfach ist. Der Hauptunterschied ist, dass Merges auf der Client Seite stattfinden und nicht, wenn man committet, auf dem Server. Schauen wir uns an, wie die Arbeit von zwei Entwicklern in einem gemeinsamen Repository abläuft. Der erste Entwickler, John, klonst das Repository, nimmt eine Änderung vor und comittet auf seinem Rechner. (Wir kürzen die Beispiele etwas ab und ersetzen die hierfür irrelevanten Protokoll Meldungen mit xxx.)

```
# John's Machine
$ git clone john@github:simplegit.git
Initialized empty Git repository in /home/john/simplegit/.git/
...
$ cd simplegit/
$ vim lib/simplegit.rb
$ git commit -am 'removed invalid default value'
[master 738ee87] removed invalid default value
 1 files changed, 1 insertions(+), 1 deletions(-)
```

Der zweite Entwickler, Jessica, tut das gleiche. Sie klonst das Repository und committet eine Änderung:

```
# Jessica's Machine
$ git clone jessica@github:simplegit.git
Initialized empty Git repository in /home/jessica/simplegit/.git/
```

```
...
$ cd simplegit/
$ vim TODO
$ git commit -am 'add reset task'
[master fbff5bc] add reset task
1 files changed, 1 insertions(+), 0 deletions(-)
```

Jetzt lädt Jessica ihre Arbeit mit `git push` auf den Server:

```
# Jessica's Machine
$ git push origin master
...
To jessica@github:simplegit.git
1edee6b..fbff5bc master -> master
```

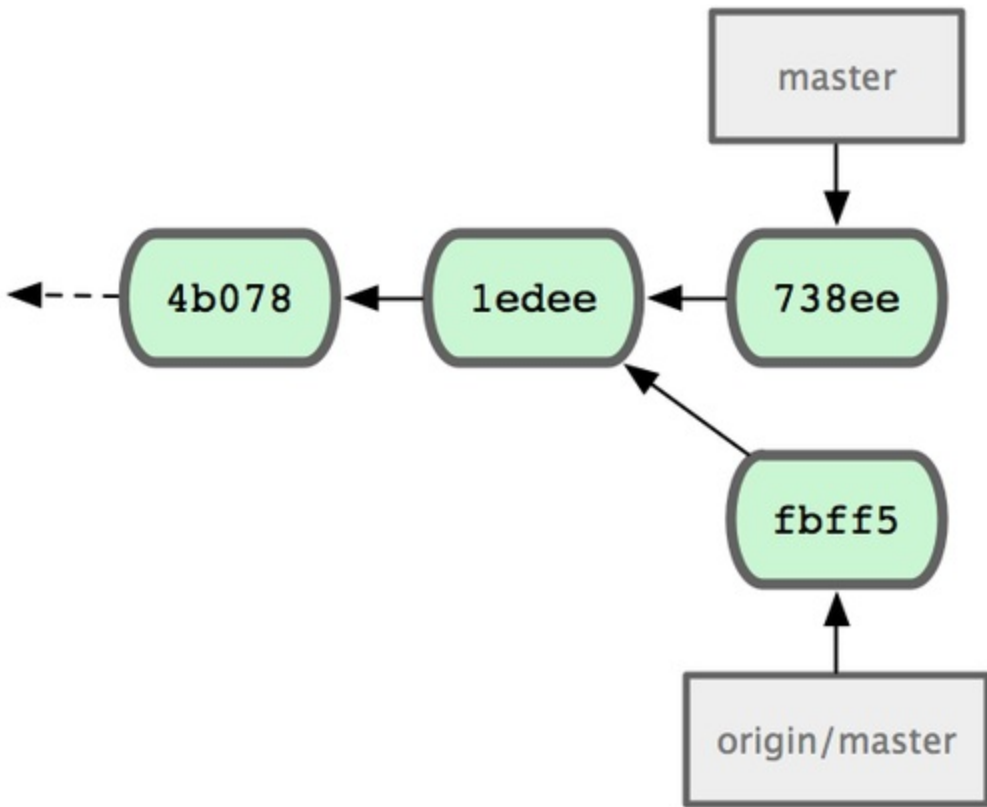
John versucht, das selbe zu tun:

```
# John's Machine
$ git push origin master
To john@github:simplegit.git
! [rejected] master -> master (non-fast forward)
error: failed to push some refs to 'john@github:simplegit.git'
```

John darf seine Änderung nicht pushen, weil Jessica in der Zwischenzeit gepusht hat. Dies ist ein Unterschied zu Subversion: wie Du siehst, haben die beiden Entwickler nicht dieselbe Datei bearbeitet. Während Subversion automatisch merged, wenn lediglich verschiedene Dateien bearbeitet wurden, muss man Commits in Git lokal mergen. John muss also Jessicas Änderungen herunterladen und mergen, bevor er dann selbst pushen darf:

```
$ git fetch origin
...
From john@github:simplegit
+ 049d078...fbff5bc master -> origin/master
```

Zu diesem Zeitpunkt sieht Johns lokales Repository jetzt aus wie in Bild 5-4.



John hat eine Referenz auf Jessicas Änderungen, aber er muss sie mit seinen eigenen Änderungen mergen, bevor er auf den Server pushen darf:

```
$ git merge origin/master
Merge made by recursive.
TODO |      1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

Der Merge verläuft glatt: Johns Commit Historie sieht jetzt aus wie in Bild 5-5.

Insert 18333fig0505.png Johns Repository nach dem Merge mit origin/master

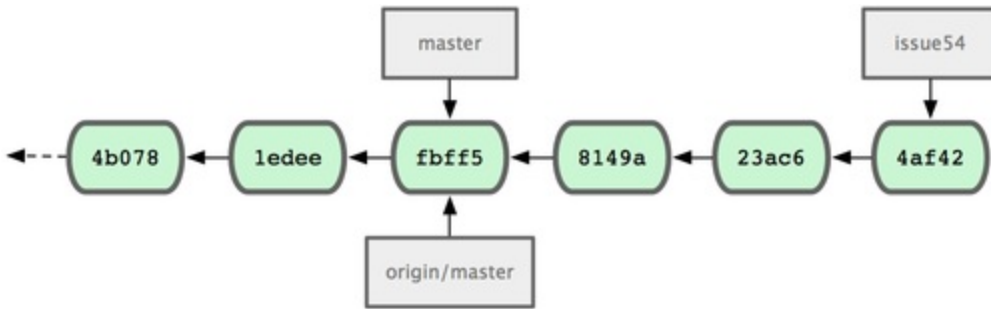
John sollte seinen Code jetzt testen, um sicher zu stellen, dass alles weiterhin funktioniert. Dann kann er seine Arbeit auf den Server pushen:

```
$ git push origin master
...
To john@githost:simplegit.git
 fbff5bc..72bbc59  master -> master
```

Johns Commit Historie sieht schließlich aus wie in Bild 5-6.

Insert 18333fig0506.png Johns Commit Historie nach dem pushen auf den origin Server

In der Zwischenzeit hat Jessica auf einem Topic Branch (xxx) gearbeitet. Sie hat einen Topic Branch mit dem Namen issue54 und darin drei Commits angelegt. Sie hat Johns Änderungen bisher noch nicht herunter geladen, sodass ihre Commit Historie jetzt so aussieht wie in Bild 5-7.

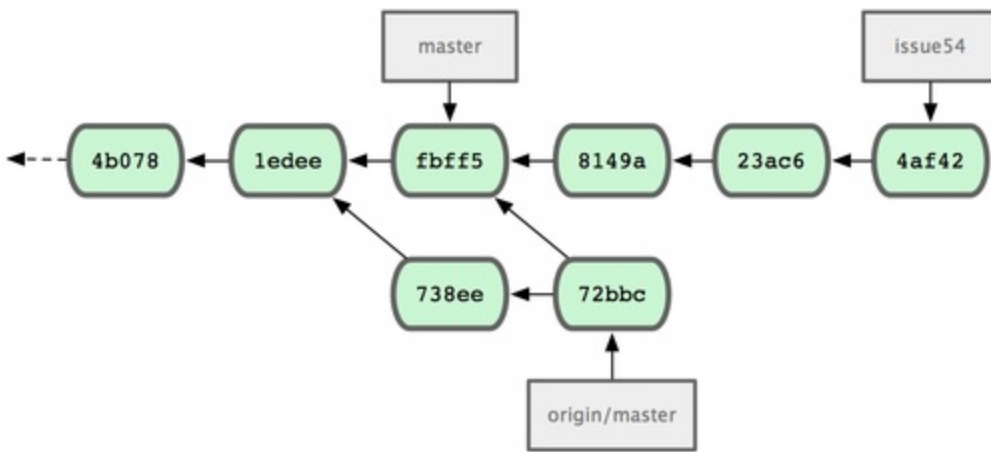


Jessica will ihre Arbeit jetzt mit John synchronisieren. Also lädt sie seine Änderungen herunter:

```

# Jessica's Machine
$ git fetch origin
...
From jessica@github:simplegit
 fbff5bc..72bbc59  master    -> origin/master
  
```

Das lädt die Änderungen, die John in der Zwischenzeit hochgeladen hat. Jessicas Historie entspricht jetzt Bild 5-8.



Jessica hat die Arbeit in ihrem Topic Branch abgeschlossen, aber sie will wissen, welche neuen Änderungen es gibt, mit denen sie ihre eigenen mergen muss.

```

$ git log --no-merges origin/master ^issue54
commit 738ee872852dfaa9d6634e0dea7a324040193016
Author: John Smith <jsmith@example.com>
Date: Fri May 29 16:01:27 2009 -0700
  
```

removed invalid default value

Jetzt kann Jessica zunächst ihren Topic Branch issue54 in ihren master Branch mergen, dann Johns Änderungen aus origin/master in ihren master Branch mergen und schließlich das Resultat auf den origin Server pushen. Als erstes wechselt sie zurück auf ihren master Branch:

```

$ git checkout master
Switched to branch "master"
Your branch is behind 'origin/master' by 2 commits, and can be fast-forwarded.
  
```

Sie kann jetzt entweder origin/master oder issue54 zuerst mergen – sie sind beide „upstream“

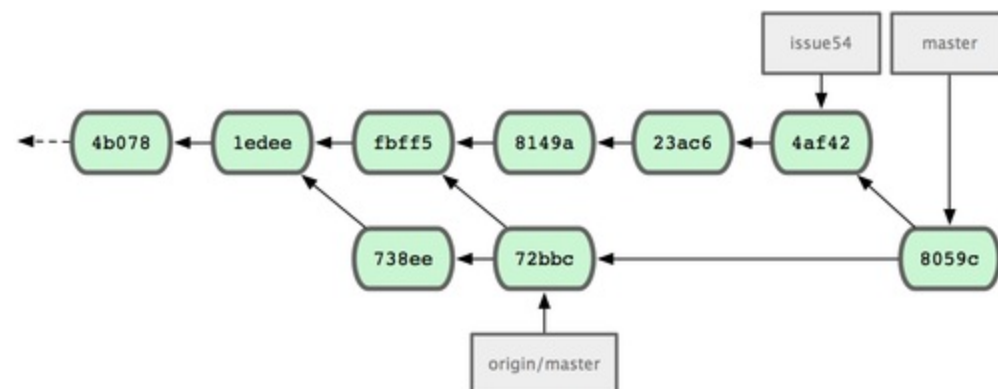
(xxx). Der resultierende Snapshot wäre identisch, egal in welcher Reihenfolge sie beide Branches in ihren master Branch merged, lediglich die Historie würde natürlich minimal anders aussehen. Jessica entscheidet sich, issue54 zuerst zu mergen:

```
$ git merge issue54
Updating fbff5bc..4af4298
Fast forward
 README      |      1 +
 lib/simplegit.rb |      6 +++++-
 2 files changed, 6 insertions(+), 1 deletions(-)
```

Das ging glatt, wie Du siehst, war es ein einfacher „fast-forward“ Merge. Als nächstes merged Jessica Johns Änderungen aus origin/master:

```
$ git merge origin/master
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |      2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
```

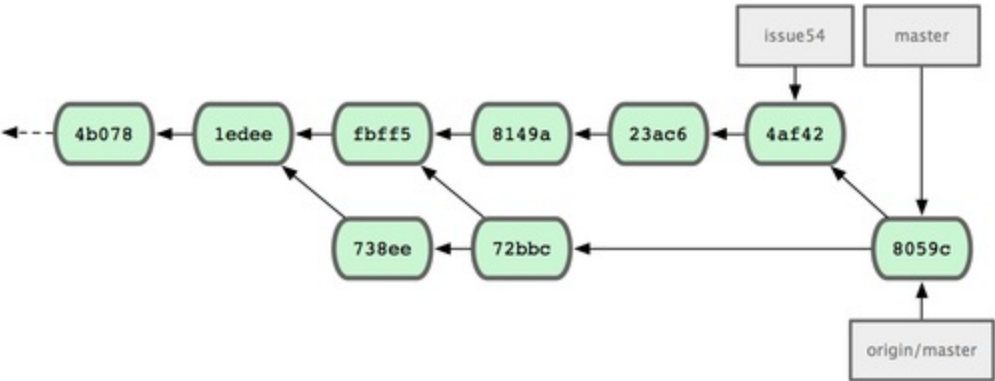
Auch hier treten keine Konflikte auf. Jessicas Historie sieht jetzt wie folgt aus (Bild 5-9).



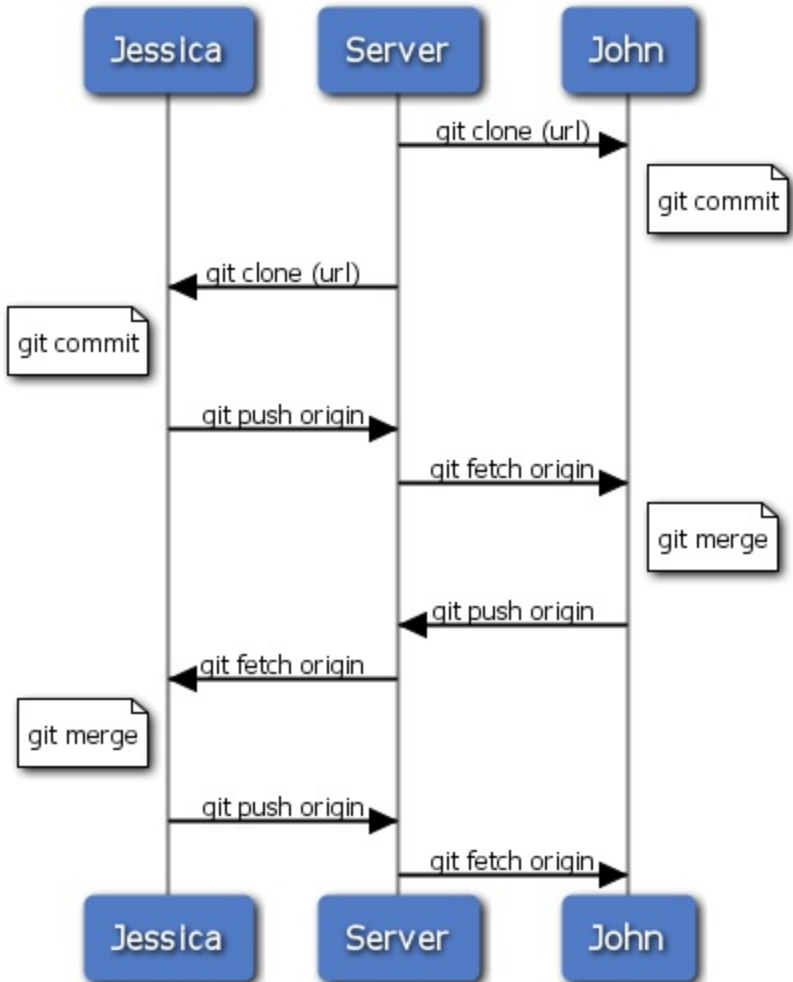
origin/master ist jetzt in Jessicas master Branch enthalten (xxx reachable xxx), sodass sie in der Lage sein sollte, auf den origin Server zu pushen (vorausgesetzt, John hat zwischenzeitlich nicht gepusht):

```
$ git push origin master
...
To jessica@github:simplegit.git
 72bbc59..8059c15 master -> master
```

Beide Entwickler haben jetzt einige Male committed und die Arbeit des jeweils anderen erfolgreich mit ihrer eigenen zusammengeführt.



Dies ist eine der simpelsten Workflow Varianten. Du arbeitest eine Weile, normalerweise in einem Topic Branch, und mergst in Deinen master Branch, wenn Du fertig bist. Wenn Du Deine Änderungen anderen zur Verfügung stellen willst, holst Du den aktuellen origin/master Branch, mergst Deinen master Branch damit und pushst das ganze zurück auf den origin Server. Der Ablauf sieht in etwa wie folgt aus (Bild 5-11).



Teil-Teams mit Integration Manager

Im folgenden Szenario sehen wir uns die Rollen von Mitarbeitern in einem größeren, nicht öffentlich arbeitenden Team an. Du wirst sehen, wie man in einer Umgebung arbeiten kann, in der kleine Gruppen (z.B. an einzelnen Features) zusammenarbeiten und ihre Ergebnisse dann von einer weiteren Gruppe in die Hauptentwicklungslinie integriert werden.

Sagen wir John und Jessica arbeiten gemeinsam an einem Feature, während Jessica und Josie an einem anderen arbeiten. Das Unternehmen verwendet einen Integration-Manager Workflow, in dem die Arbeit der verschiedenen Gruppen von anderen Mitarbeitern zentral integriert werden – und der master Branch nur von diesen letzteren geschrieben werden kann. In diesem Szenario wird sämtliche Arbeit von den Teams in Branches erledigt und dann von den Integration-Managern zusammengeführt.

Schauen wir uns Jessicas Workflow an, während sie mit jeweils verschiedenen Entwicklern parallel an zwei Features arbeitet. Nehmen wir an, sie hat das Repository bereits geklont und will zuerst an featureA arbeiten. Sie legt einen neuen Branch für das Feature an und fängt an, daran zu arbeiten:

```
# 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(-)
```

Jetzt will sie ihre Arbeit John zur Verfügung stellen, der am gleichen Feature arbeiten will, und pusht dazu ihre Commits in ihrem featureA Branch auf den Server. Jessica hat keinen Schreibzugriff auf den master Branch – den haben nur die Integration Manager – also pusht sie ihren Feature Branch, der nur der Zusammenarbeit mit John dient:

```
$ git push origin featureA
...
To jessica@github:simplegit.git
 * [new branch]      featureA -> featureA
```

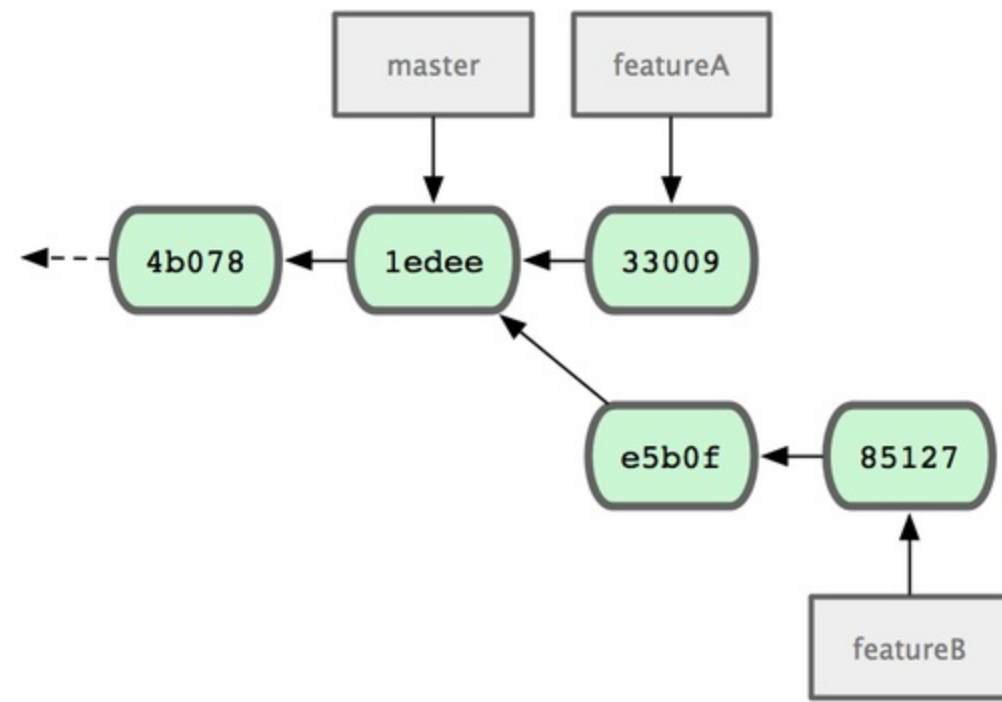
Jessica schickt John eine E-Mail und lässt ihn wissen, dass sie ihre Arbeit in einen Branch featureA hochgeladen hat. Während sie jetzt auf Feedback von John wartet, kann Jessica anfangen, an featureB zu arbeiten – diesmal gemeinsam mit Josie. Also legt sie einen neuen Feature Branch an, der auf dem gegenwärtigen master Branch des origin Servers basiert:

```
# Jessica's Machine
$ git fetch origin
$ git checkout -b featureB origin/master
Switched to a new branch "featureB"
```

Jetzt legt Jessica eine Reihe von Commits im featureB Branch an:

```
$ 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 entspricht jetzt Bild 5-12.



Jessica könnte ihre Arbeit jetzt hochladen, aber sie hat eine E-Mail von Josie erhalten, dass sie bereits einen Feature Branch `featureBee` für dasselbe Feature auf dem Server angelegt hat. Jessica muss also erst ihre eigenen Änderungen mit diesem Branch mergen und dann dorthin pushen. Sie lädt also Josies Änderungen mit `git fetch` herunter:

```
$ git fetch origin
...
From jessica@github:simplegit
* [new branch]      featureBee -> origin/featureBee
```

Jessica kann ihre eigene Arbeit jetzt mit diesen Änderungen zusammenführen:

```
$ git merge origin/featureBee
Auto-merging lib/simplegit.rb
Merge made by recursive.
 lib/simplegit.rb |    4 ++++
 1 files changed, 4 insertions(+), 0 deletions(-)
```

Es gibt jetzt ein kleines Problem. Jessica muss die zusammengeführten Änderungen in ihrem `featureB` Branch in den `featureBee` Branch auf dem Server pushen. Das kann sie tun, indem sie sowohl den Namen ihres lokalen Branches als auch des externen Branches angibt, und zwar mit einem Doppelpunkt getrennt:

```
$ git push origin featureB:featureBee
...
To jessica@github:simplegit.git
 fba9af8..cd685d1 featureB -> featureBee
```

Das nennt man eine *Refspec*. In Kapitel 9 gehen wir detailliert auf Git Refspecs ein und darauf, was man noch mit ihnen machen kann.

Als nächstes schickt John Jessica eine E-Mail. Er schreibt, dass er einige Änderungen in den featureA Branch gepusht hat, und bittet sie, diese zu prüfen. Sie führt also `git fetch` aus, um die Änderungen herunter zu laden:

```
$ git fetch origin
...
From jessica@github:simplegit
 3300904..aad881d  featureA  -> origin/featureA
```

Danach kann sie die neuen Änderungen mit `git log` auflisten:

```
$ git log origin/featureA ^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

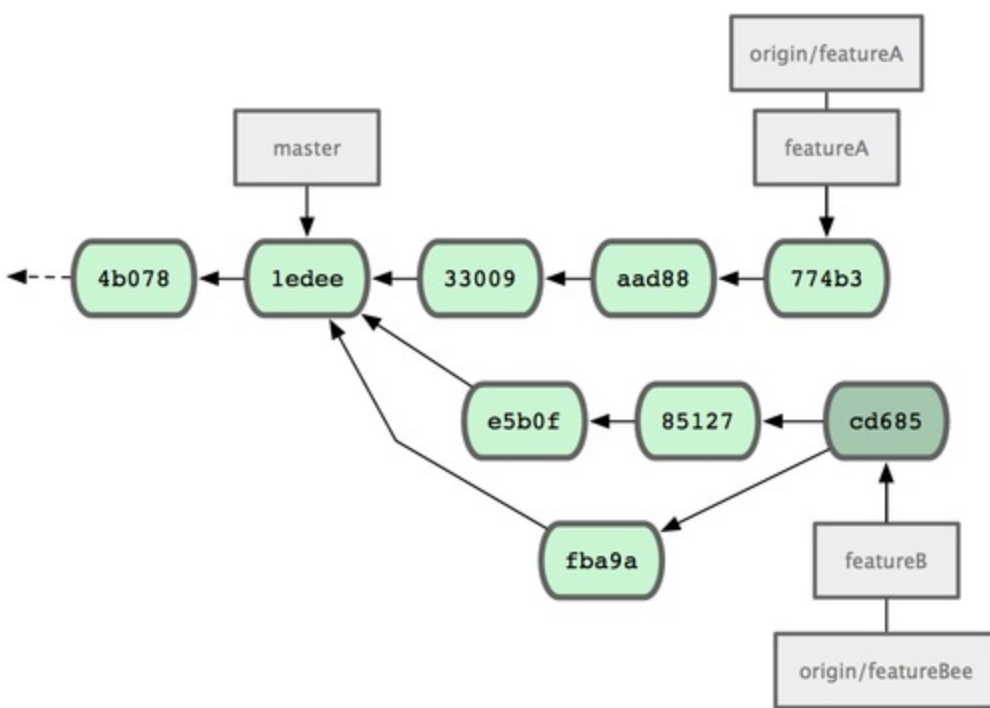
Schließlich aktualisiert sie ihren eigenen featureA Branch mit Johns Änderungen:

```
$ git checkout featureA
Switched to branch "featureA"
$ git merge origin/featureA
Updating 3300904..aad881d
Fast forward
 lib/simplegit.rb | 10 ++++++++-
1 files changed, 9 insertions(+), 1 deletions(-)
```

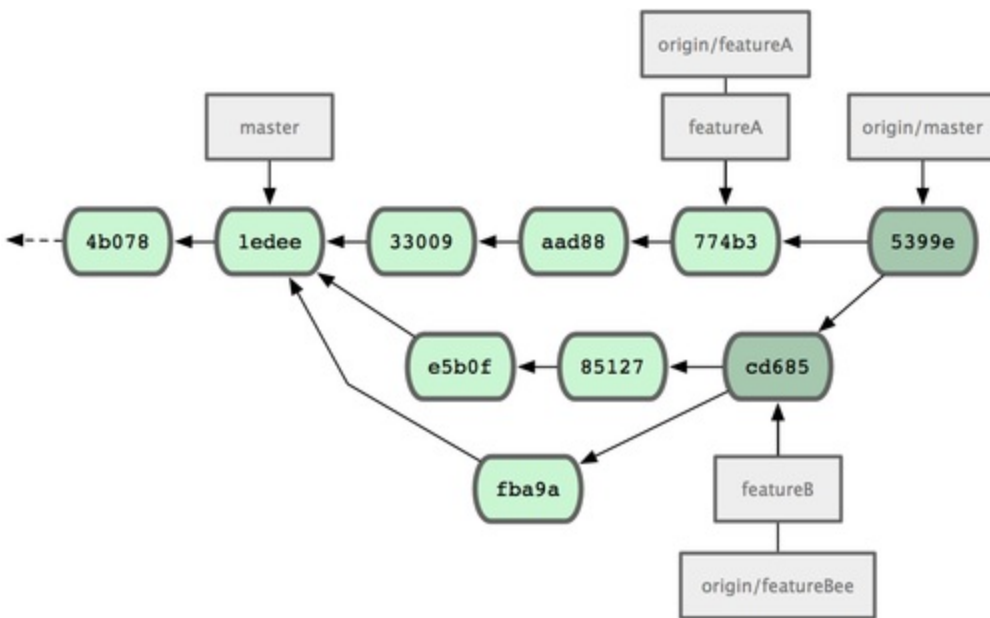
Jessica will eine kleine Änderung vornehmen. Also comittet sie und pusht den neuen Commit auf den Server:

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

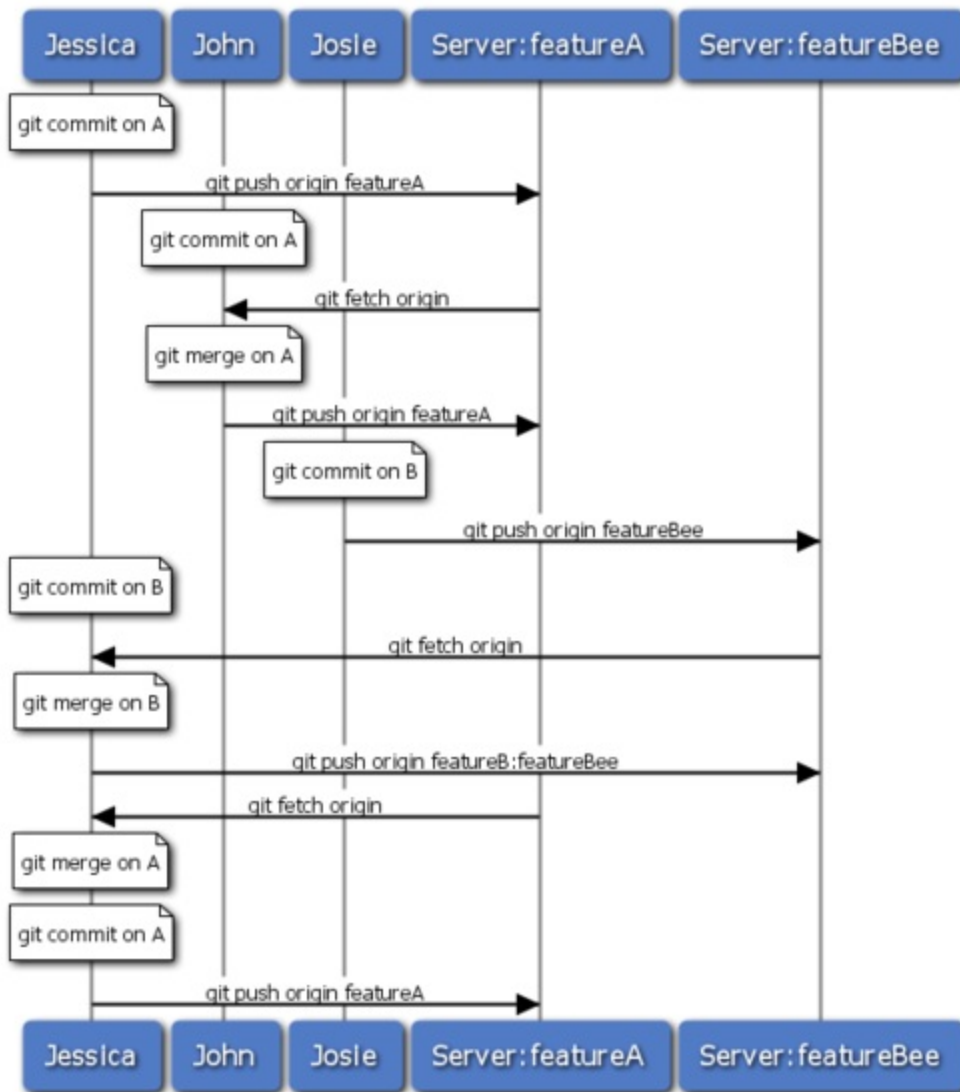
Jessicas Commit Historie sieht jetzt wie folgt aus (Bild 5-13).



Jessica, Josie und John informieren jetzt ihre Integration Manager, dass die Änderungen in den Branches featureA und featureBee fertig sind und in die Hauptlinie in master übernommen werden können. Nachdem das geschehen ist, wird `git fetch` die neuen Merge Commits herunterladen und die Commit Historie in etwa wie folgt aussehen (Bild 5-14):



Viele Teams wechseln zu Git, weil es auf einfache Weise ermöglicht, verschiedene Teams parallel an verschiedenen Entwicklungslinien zu arbeiten, die erst später im Prozess integriert werden. Ein riesiger Vorteil von Git besteht darin, dass man in kleinen Teilgruppen über externe Branches zusammenarbeiten kann, ohne dass dazu notwendig wäre, das gesamte Team zu involvieren und möglicherweise aufzuhalten. Der Ablauf dieser Art von Workflow kann wie folgt dargestellt werden (Bild 5-15).



Kleine, öffentliche Projekte

An öffentlichen Projekten mitzuarbeiten funktioniert ein bisschen anders. Weil man normalerweise keinen Schreibzugriff auf das öffentliche Repository des Projektes hat, muss man mit den Betreibern in anderer Form zusammenarbeiten. Unser erstes Beispiel beschreibt, wie man zu Projekten auf Git Hosts beitragen kann, die es erlauben Forks eines Projektes anzulegen. Z.B. unterstützen die Git Hosting Seiten repo.or.cz und [GitHub](https://github.com) dieses Feature – und viele Projekt Betreiber akzeptieren Änderungen in dieser Form. Das nächste Beispiel geht dann darauf ein, wie man mit Projekten arbeiten kann, die es bevorzugen, Patches per E-Mail zu erhalten (xxx oder Ticket Tracker, wie z.B. [Rails xxx](https://rails.rubyonrails.org)).

Zunächst wirst vermutlich das Hauptrepository klonen, einen Topic Branch für Deinen Patch anlegen und dann darin arbeiten. Der Prozess sieht dann in etwa so aus:

```

$ git clone (url)
$ cd project
$ git checkout -b featureA
$ (work)
$ git commit
$ (work)
$ git commit
  
```


Es ist wahrscheinlich sinnvoll, `git rebase -i` zu verwenden, um die verschiedenen Commits zu einem einzigen zusammen zu packen („squash“, quetschen) oder um sie in anderer Weise neu zu arrangieren, sodass es für die Projekt Betreiber leichter ist, die Änderungen nach zu vollziehen. In Kapitel 6 gehen wir ausführlicher auf das interaktive `rebase -i` ein.

Wenn Dein Branch fertig ist und Du Deine Arbeit den Projekt Betreibern zur Verfügung stellen willst, gehst Du auf die Projekt Seite und klickst auf den „Fork“ Button. Dadurch legst Du Deinen eigenen Fork des Projektes an, in den Du dann schreiben kannst. Die Repository URL dieses Forks musst Du dann als ein zweites, externes Repository („remote“) einrichten. In unserem Beispiel verwenden wir den Namen `myfork`:

```
$ git remote add myfork (url)
```

Jetzt kannst Du Deine Änderungen dorthin hochladen. Am besten tust Du das, indem Du Deinen Topic Branch hochlädst (statt ihn in Deinen master Branch zu mergen und den dann hochzuladen). Dies deshalb, weil Du, wenn Deine Änderungen nicht akzeptiert werden, Deinen eigenen master Branch nicht zurücksetzen musst. Wenn die Projekt Betreiber Deine Änderungen mergen, rebasen oder cherry-picken, landen sie schließlich ohnehin in Deinem master Branch.

```
$ git push myfork featureA
```

Nachdem Du Deine Arbeit in Deinen Fork hochgeladen hast, musst Du die Projekt Betreiber benachrichtigen. Dies wird oft als „pull request“ bezeichnet. Du kannst ihn entweder direkt über die Webseite schicken (GitHub hat dazu einen „pull request“ Button) oder den Git Befehl `git request-pull` verwenden und manuell eine E-Mail an die Projekt Betreiber schicken.

Der `request-pull` Befehl vergleicht denjenigen Branch, für den Deine Änderungen gedacht sind, mit Deinem Topic Branch und gibt eine Übersicht der Änderungen aus. Wenn Jessica zwei Änderungen in einem Topic Branch hat und nun John einen pull request schicken will, kann sie folgendes tun:

```
$ git request-pull origin/master myfork
The following changes since commit 1edee6b1d61823a2de3b09c160d7080b8d1b3a40:
  John 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(-)
```

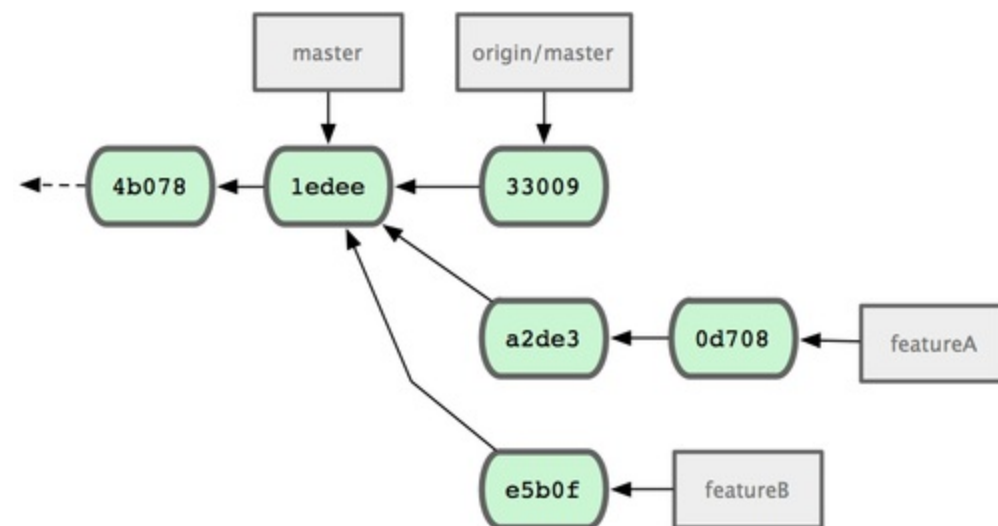
Die Ausgabe kann an den Projekt Betreiber geschickt werden – sie sagt klar, auf welchem Branch die Arbeit basiert, gibt eine Zusammenfassung der Änderungen und gibt an, aus welchem Fork

oder Repository man die Änderungen herunterladen kann.

Wenn Du nicht selbst Betreiber eines bestimmten Projektes bist, ist es im Allgemeinen einfacher, einen Branch master immer dem origin/master Branch tracken (xxx folgen) zu lassen und eigene Änderungen in Topic Branches vorzunehmen, die man leicht wieder löschen kann, wenn sie nicht akzeptiert werden. Wenn Du Aspekte Deiner Arbeit in Topic Branches isolierst, kannst Du sie außerdem recht leicht auf den letzten Stand des Hauptrepositories rebasen, falls das Hauptrepository in der Zwischenzeit weiter entwickelt wurde und Deine Commits nicht mehr sauber passen. Wenn Du beispielsweise an einem anderen Patch für das Projekt arbeiten willst, verwende dazu nicht weiter den gleichen Topic Branch, den Du gerade in Deinen Fork hochgeladen hast. Lege statt dessen einen neuen Topic Branch an, der wiederum auf dem master Branch des Hauptrepositories basiert.

```
$ git checkout -b featureB origin/master
$ (work)
$ git commit
$ git push myfork featureB
$ (email maintainer)
$ git fetch origin
```

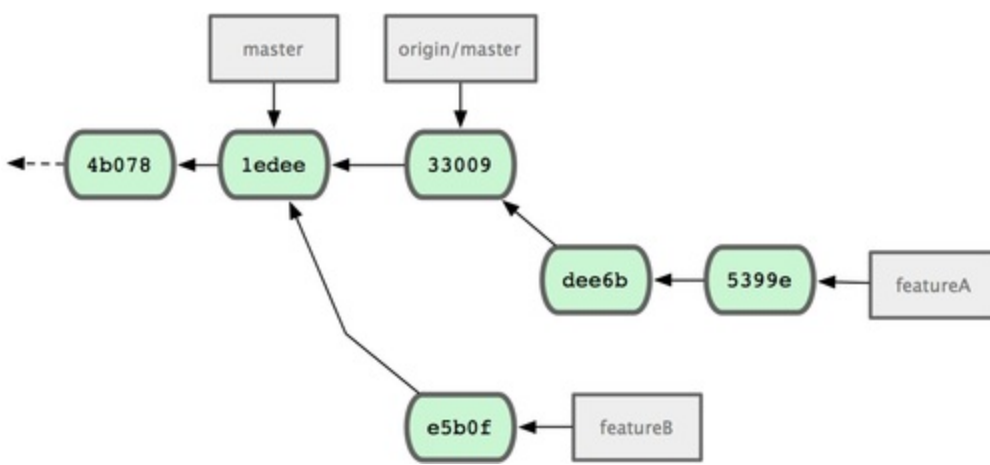
Deine Arbeit an den verschiedenen Patches sind jetzt in Deine Topic Branches isoliert – ähnlich wie in einer Patch Queue – sodass Du die einzelnen Topic Branches neu schreiben, rebasen und ändern kannst, ohne dass sie mit einander in Konflikt geraten (siehe Bild 5-16).



Sagen wir, der Projekt Betreiber hat eine Reihe von Änderungen Dritter in das Projekt übernommen und Deine eigenen Änderungen lassen sich jetzt nicht mehr sauber mergen. In diesem Fall kannst Du Deine Änderungen auf dem neuen Stand des origin/master Branches rebasen, Konflikte beheben und Deine Arbeit erneut einreichen:

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

Das schreibt Deine Commit Historie neu, sodass sie jetzt so aussieht (Bild 5-17):



Weil Du den Branch rebased hast, musst Du die Option `-f` verwenden, um den `featureA` Branch auf dem Server zu ersetzen, denn Du hast die Commit Historie umgeschrieben und nun ist ein Commit enthalten, von dem der gegenwärtig letzte Commit des externen Branches nicht abstammt. Eine Alternative dazu wäre, den Branch jetzt in einen neuen externen Branch zu pushen, z.B. `featureAv2`.

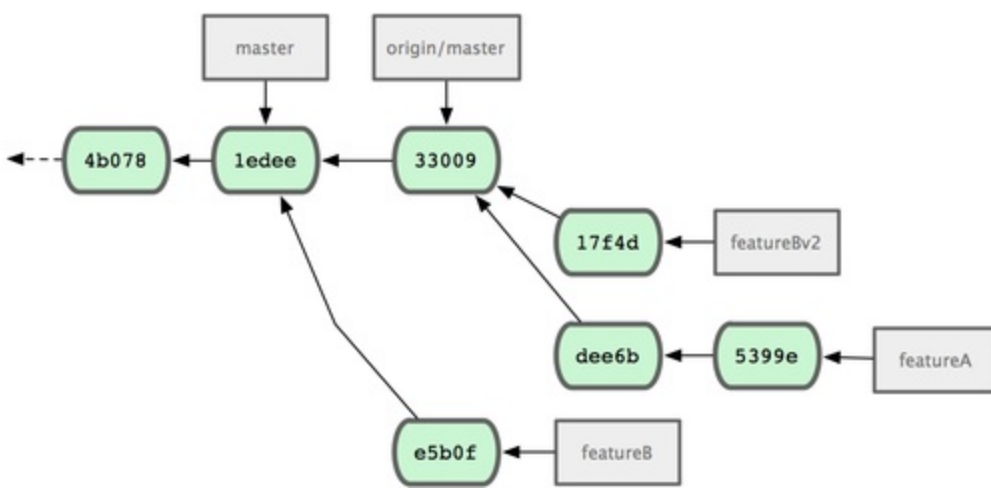
Schauen wir uns noch ein anderes Szenario an: der Projekt Betreiber hat sich Deine Arbeit angesehen und will die Änderungen übernehmen, aber er bittet dich, noch eine Kleinigkeit an der Implementierung zu ändern. Du willst die Gelegenheit außerdem nutzen, um Deine Änderungen neu auf den gegenwärtigen `master` Branch des Projektes zu basieren. Du legst dazu einen neuen Branch von `origin/master` an, übernimmst Deine Änderungen dahin, löst ggf. Konflikte auf, nimmst die angeforderte Änderung an der Implementierung vor und lädst die Änderungen auf den Server:

```

$ git checkout -b featureBv2 origin/master
$ git merge --no-commit --squash featureB
$ (change implementation)
$ git commit
$ git push myfork featureBv2
  
```

Die `--squash` Option bewirkt, dass alle Änderungen des Merge Branches (`featureB`) übernommen werden, ohne aber dass zusätzlich ein Merge Commit angelegt wird. Die `--no-commit` Option instruiert Git außerdem, nicht automatisch einen Commit anzulegen. Das erlaubt dir, sämtliche Änderungen aus dem anderen Branch zu übernehmen und dann weitere Änderungen vorzunehmen, bevor Du das Ganze dann in einem neuen Commit speicherst.

Jetzt kannst Du dem Projekt Betreiber eine Nachricht schicken, dass Du die angeforderte Änderung vorgenommen hast und dass er Deine Arbeit in Deinem `featureBv2` Branch finden kann (siehe Bild 5-18).



Große öffentliche Projekte

Viele große Projekte haben einen etablierten Prozess, nach dem sie vorgehen, wenn es darum geht, Patches zu akzeptieren. Du musst Dich mit den jeweiligen Regeln vertraut machen, die in jedem Projekt ein bisschen anders sind. Allerdings akzeptieren viele große Projekte Patches per E-Mail über eine Entwickler Mailingliste (xxx oder einen Bugtracker, wie Rails xxx). Deshalb gehen wir auf dieses Beispiel als nächstes ein.

Der Workflow ist ähnlich wie im vorherigen Szenario. Du legst für jeden Patch oder jede Patch Serie einen Topic Branch an, in dem Du arbeitest. Der Unterschied besteht dann darin, auf welchem Wege Du die Änderungen an das Projekt schickst. Statt das Projekt zu forken und Änderungen in Deinen Fork hochzuladen, erzeugst Du eine E-Mail Version Deiner Commits und schickst sie als Patch an die Entwickler Mailingliste.

```

$ git checkout -b topicA
$ (work)
$ git commit
$ (work)
$ git commit
  
```

Jetzt hast Du zwei Commits, die Du an die Mailingliste schicken willst. Du kannst den Befehl `git format-patch` verwenden, um aus diesen Commits Dateien zu erzeugen, die im mbox-Format formatiert sind und die Du per E-Mail verschicken kannst. Dieser Befehl macht aus jedem Commit eine E-Mail Datei. Die erste Zeile der Commit Meldung wird zum Betreff der E-Mail und der Rest der Commit Meldung sowie der Patch des Commits selbst wird zum Text der E-Mail. Das schöne daran ist, dass wenn man einen auf diese Weise erzeugten Patch benutzt, dann bleiben alle Commit Informationen erhalten. Du kannst das in den nächsten Beispielen sehen:

```

$ git format-patch -M origin/master
0001-add-limit-to-log-function.patch
0002-changed-log-output-to-30-from-25.patch
  
```

Der Befehl `git format-patch` zeigt Dir die Namen der Patch Dateien an, die er erzeugt hat. (Die `-M` option weist Git an, nach umbenannten Dateien Ausschau zu halten.) Die Dateien sehen dann so 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')
--
1.6.2.rc1.20.g8c5b.dirty
```

Du kannst diese Patch Dateien anschließend bearbeiten, z.B. um weitere Informationen für die Mailingliste hinzuzufügen, die Du nicht in der Commit Meldung haben willst. Wenn Du zusätzlichen Text zwischen der --- Zeile und dem Anfang des Patches (der Zeile `lib/simplegit.rb` in diesem Fall), dann ist er für den Leser sichtbar, aber Git wird ihn ignorieren, wenn man den Patch verwendet.

Um das jetzt an die Mailingliste zu schicken, kannst Du entweder die Datei per copy-and-paste in Dein E-Mail Programm kopieren, als Anhang an eine E-Mail anhängen oder Du kannst die Dateien mit einem Befehlszeilen Programm direkt verschicken. Patches zu kopieren verursacht oft Formatierungsprobleme – insbesondere mit „smarten“ E-Mail Clients, die die Dinge umformatieren, die man einfügt. Zum Glück bringt Git aber ein Tool mit, mit dem man Patches in ihrer korrekten Formatierung über IMAP verschicken kann. In folgendem Beispiel zeige ich, wie man Patches über Gmail verschicken kann, welches der E-Mail Client ist, den ich selbst verwende. Darüberhinaus findest Du ausführliche Beschreibungen für zahlreiche E-Mail Programme am Ende der schon erwähnten Datei `Documentation/SubmittingPatches` im Git Quellcode.

Zunächst musst Du die IMAP Sektion in Deiner `~/.gitconfig` Datei ausfüllen. Du kannst jeden Wert separat mit dem Befehl `git config` eingeben oder Du kannst die Datei öffnen und sie manuell eingeben. Im Endeffekt sollte Deine `~/.gitconfig` Datei in etwa so aussehen:

```
[imap]
  folder = "[Gmail]/Drafts"
  host = imaps://imap.gmail.com
```

```
user = user@gmail.com
pass = p4ssw0rd
port = 993
sslverify = false
```

Wenn Dein IMAP Server kein SSL verwendet, kannst Du die letzten beiden Zeilen wahrscheinlich weglassen und der host dürfte mit `imap://` und nicht `imaps://` beginnen. Wenn Du diese Einstellungen konfiguriert hast, kannst Du `git imap-send` verwenden, um Deine Patches in den Entwurfsordner des angegebenen IMAP Servers zu kopieren:

```
$ 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
```

Jetzt kannst Du in Deinen Entwürfe-Ordner wechseln, die Mailingliste an die Du den Patch senden möchtest im An-Feld setzen, vielleicht noch den Maintainer oder die verantwortliche Person in das CC-Feld einfügen und dann das Ganze losschicken.

Man kann Patches auch über einen SMTP-Server schicken. Wie im letzten Beispiel, kann man auch hier jeden einzelnen Wert mit einer Reihe von `git config` Kommandos setzen. Oder aber Du änderst die Sektion `sendemail` in Deiner `~/.gitconfig` Datei manuell:

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

Nach der Änderungen kannst Du mit `git send-email` die Patches abschicken:

```
$ 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 dann für jeden Patch, den Du verschickst, ein paar Log Informationen 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

Jetzt solltest Du in den Entwurfsordner Deines E-Mail Clients gehen, als Empfänger Adresse die jeweilige Mailingliste angeben, möglicherweise ein CC an den Projektbetreiber oder einen anderen Verantwortlichen setzen und die E-Mail dann verschicken können.

Zusammenfassung

Wir haben jetzt eine Reihe von Workflows besprochen, die für jeweils sehr verschiedene Arten von Projekten üblich sind und denen Du vermutlich begegnen wirst. Wir haben außerdem einige neue Tools besprochen, die dabei hilfreich sind, diese Workflows umzusetzen. Als nächstes werden wir auf die andere Seite dieser Medaille eingehen: wie Du selbst ein Git Projekt betreiben kannst. Du wirst lernen, wie Du als „wohlwollender Diktator“ oder als Integration Manager arbeiten kannst.

Ein Projekt betreiben

Neben dem Wissen, das Du brauchst, um zu einem bestehenden Projekt Änderungen beizutragen, wirst Du vermutlich wissen wollen, wie Du selbst ein Projekt betreiben kannst. Dazu willst Du Patches akzeptieren und anwenden, die per `git format-patch` erzeugt und Dir per E-Mail geschickt wurden. Oder Du willst Änderungen aus externen Branches übernehmen, die Du zu Deinem Projekt hinzugefügt hast. Ob Du nun für das Hauptrepository verantwortlich bist oder ob Du dabei helfen willst, Patches zu verifizieren und zu bestätigen – in beiden Fällen musst Du wissen, wie Du Änderungen in einer Weise übernehmen kannst, die für andere Mitarbeiter nachvollziehbar und für Dich selbst tragbar ist.

In Topic Branches arbeiten

Wenn Du Änderungen von anderen übernehmen willst, ist normalerweise eine gute Idee, sie in einem Topic Branch auszuprobieren – d.h., einem temporären Branch, dessen Zweck nur darin besteht, die jeweiligen Änderungen auszuprobieren. Auf diese Weise ist es einfach, Patches ggf. anzupassen oder sie im Zweifelsfall im Topic Branch liegen zu lassen, wenn sie nicht funktionieren und Du im Moment nicht die Zeit hast, Dich weiter damit zu befassen. Es ist empfehlenswert, Topic Branches Namen zu geben, die gut kommunizieren, worum es sich bei den jeweiligen Änderungen dreht, wie z.B. `ruby_client` oder etwas ähnlich aussagekräftiges, das Dir hilft, Dich daran zu erinnern. Der Projekt Betreiber des Git Projektes selbst vergibt Namensräume für solche Branches – wie z.B. `sc/ruby_client`, wobei `sc` ein Kürzel für den jeweiligen Autor des Patches ist. Wie Du inzwischen weißt, kannst Du einen neuen Branch, der auf dem gegenwärtigen master Branch basiert, wie folgt erzeugen (xxx falsch, das stimmt nur, wenn master der aktuelle Branch ist xxx):

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

Oder, wenn außerdem direkt zu dem neuen Branch wechseln willst, kannst Du die `-b` für den `git checkout` Befehl verwenden:

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

Nachdem Du jetzt einen Topic Branch angelegt hast, kannst Du die Änderungen zu diesem Branch hinzufügen, um herauszufinden, ob Du sie dauerhaft in einen offiziellen Branch übernehmen willst.

Patches aus E-Mails verwenden

Wenn Du einen Patch, den Du auf Dein Projekt anwenden willst, per E-Mail erhältst, gibt es zwei Möglichkeiten, das zu tun: `git apply` und `git am`.

Einen Patch verwenden: `git apply`

Wenn der Patch mit `git diff` oder dem Unix Befehl `diff` erzeugt wurde, dann kannst Du ihn mit

dem Befehl `git apply` anwenden. Nehmen wir an, Du hast den Patch nach `/tmp/patch-ruby-client.patch` gespeichert. Dann kannst Du ihn wie folgt verwenden:

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

Das ändert die Dateien in Deinem Git Arbeitsverzeichnis. Das ist fast das selbe wie wenn Du den Unix Befehl `patch -p1` verwendest. Der Git Befehl ist aber paranoider und akzeptiert nicht so viele unklare Übereinstimmungen. Außerdem kann er mit neu hinzugefügten, gelöschten und umbenannten Dateien umgehen, was der Unix Befehl `patch` nicht kann. Schließlich ist `git apply` ein „alles oder nichts“ Befehl, der entweder alle Änderungen übernimmt oder gar keine (wenn bei einem etwas schief geht), während `patch` Änderungen auch teilweise übernimmt, sodass er Dein Arbeitsverzeichnis gegebenenfalls in einem unbrauchbaren Zustand hinterlässt. `git apply` ist also insgesamt strenger als `patch`. Es legt im übrigen keinen Commit für Dich an. Nachdem Du `git apply` ausgeführt hast, musst Du die Änderungen manuell zur Staging Area hinzufügen und comitten.

Du kannst `git apply --check` verwenden, um zu testen, ob der Patch sauber anwendbar wäre:

```
$ 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 dieser Befehl nichts ausgibt, sollte der Befehl sauber anwendbar sein.

Einen Patch verwenden: `git am`

Wenn der Autor des Patches selbst mit Git arbeitet, kann er Dir das Leben leichter machen, indem er `git format-patch` verwendet, um seinen Patch zu erzeugen: der Patch wird dann die Commit Informationen über den Autor sowie die Commit Meldung enthalten. Es ist also empfehlenswert, Entwickler darum zu bitten und zu ermutigen, `git format-patch` statt `git diff` zu verwenden. Du wirst dann `git apply` nur sehr selten anwenden müssen (xxx legacy patches ??? xxx)

Um einen Patch zu verwenden, der mit `git format-patch` erzeugt wurde, benutzt Du den Befehl `git am`. Technisch gesehen ist `git am` ein Befehl, der eine mbox Datei lesen kann, d.h. eine einfache Nur-Text-Datei, die eine oder mehrere E-Mails enthalten kann. Eine solche Datei sieht in etwa wie folgt 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
```

Das ist der Anfang der Ausgabe des `git format-patch` Befehls, die Du im vorherigen Abschnitt gesehen hast – und außerdem valides mbox E-Mail Format. Wenn Du eine solche Datei im mbox Format erhalten hast und der Absender `git send-email` korrekt verwendet hat, kannst Du `git am`

mit der Datei verwenden und alle darin enthaltenen Patches werden auf Dein Projekt angewendet. Wenn Du einen E-Mail Client verwendest, der mehrere E-Mails im mbox Format in einer Datei speichern oder exportieren kann, kannst Du auch eine ganze Reihe von Patches in eine einzige Datei speichern und dann `git am` verwenden, um sie nacheinander anzuwenden.

Wenn jemand einen Patch, der mit `git format-patch` erzeugt wurde, in einem Ticketsystem oder ähnlichem abgelegt hat, kannst Du die Datei lokal speichern und dann ebenfalls `git am` ausführen, um den Patch anzuwenden:

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

Der Patch passte sauber auf die Codebase und hat automatisch einen neuen Commit angelegt. Die Autor Information wurde aus den From und Date Headern der E-Mail übernommen und die Commit Meldung aus dem Subject und Body der E-Mail. Wenn der Patch z.B. aus dem mbox Beispiel von oben stammt, sieht der resultierende Commit wie folgt aus:

```
$ 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
```

Das Commit Feld zeigt den Namen desjenigen, der den Patch angewendet hat und CommitDate das jeweilige Datum und Uhrzeit. Die Felder Author und AuthorDate geben an, wer den Commit wann angelegt hat.

Es ist allerdings möglich, dass der Patch nicht sauber auf den gegenwärtigen Code passt. Möglicherweise unterscheidet sich der jeweilige Branch inzwischen erheblich von dem Zustand, in dem er sich befand, als die in dem Patch enthaltenen Änderungen geschrieben wurden. In dem Fall wird `git am` fehlschlagen und Dir mitteilen, was zu tun ist:

```
$ 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".
```

Der Befehl fügt Konfliktmarkierungen in allen problematischen Dateien ein, so wie bei einem konfligierenden Merge oder Rebase. Und Du kannst den Konflikt in der selben Weise beheben: die Datei bearbeiten, die Änderungen zur Staging Area hinzufügen und dann `git am --resolved`

ausführen, um mit dem jeweils nächsten Patch (falls vorhanden) fortzufahren.

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

Wenn Du willst, dass Git versucht, einen Konflikt etwas intelligenter zu lösen, kannst Du die -3 Option angeben, sodass Git einen 3-Wege-Merge versucht. Dies ist deshalb nicht der Standard, weil ein 3-Wege-Merge nicht funktioniert, wenn der Commit, auf dem der Patch basiert, nicht Teil Deines Repositories ist. Wenn Du den Commit allerdings in Deiner Historie hast, d.h. wenn der Patch auf einem öffentlichen Commit basiert, dann ist die -3 Option oft die bessere Variante, um einen konfligierenden Patch anzuwenden:

```
$ 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 habe ich versucht, einen Patch anzuwenden, der ich bereits zuvor angewendet hatte. Ohne die -3 Option würde ich einen Konflikt erhalten.

Wenn Du eine Reihe von Patches aus einer Datei im mbox Format anwendest, kannst Du außerdem den git am Befehl in einem interaktiven Modus ausführen. In diesem Modus hält Git bei jedem Patch an und fragt Dich jeweils, ob Du den Patch anwenden willst:

```
$ 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
```

Das ist praktisch, wenn Du eine ganze Reihe von Patches in einer Datei hast. Du kannst jeweils Patches anzeigen, an die Du Dich nicht erinnern kannst, oder Patches auslassen, z.B. weil Du sie schon zuvor angewendet hattest.

Nachdem Du alle Patches in Deinem Topic Branch angewendet hast, kannst Du die Änderungen durchsehen, testen und entscheiden, ob Du sie in einen dauerhaft bestehenden Branch übernehmen willst.

Checking Out Remote Branches

Möglicherweise kommen die Änderungen aber nicht als Patch sondern von einem Git Anwender, der sein eigenes Repository aufgesetzt hat, seine Änderungen dorthin hochgeladen und Dir dann die URL des Repositories und den Namen des Branches geschickt hat. In diesem Fall kannst Du

das Repository als ein „remote“ (externes Repository) hinzufügen und die Änderungen lokal mergen.

Wenn Dir z.B. Jessica eine E-Mail schickt und mitteilt, dass sie ein großartiges, neues Feature im `ruby-client` Branch ihres Repositories hat, dann kannst Du das Feature testen, indem Du das Repository als externes Repository Deines Projektes konfigurieren und den Branch lokal auscheckst:

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

Wenn sie Dir später erneut eine E-Mail mit einem anderen Branch schickt, der ein anderes, großartiges Feature enthält, dann kannst Du diesen Branch direkt herunterladen und auschecken, weil Du das externe Repository noch konfiguriert hast.

Dies ist insbesondere nützlich, wenn Du mit jemandem regelmäßig zusammen arbeitest. Wenn jemand lediglich gelegentlich einen einzelnen Patch beiträgt, dann ist es wahrscheinlich weniger aufwendig, ihn per E-Mail zu akzeptieren, als von jedem zu erwarten, einen eigenen Server zu betreiben, und selbst ständig externe Repositories hinzuzufügen und zu entfernen. Du wirst kaum hunderte von externen Repositories verwalten wollen, nur um von jedem ein paar Änderungen zu erhalten. Auf der anderen Seite erleichtern Dir Scripts und Hosted Services diesen Prozess. Es hängt also alles davon ab, wie Du selbst und wie Deine Mitarbeiter entwickeln.

Wenn Du mit jemandem nicht regelmäßig zusammen arbeitest, aber trotzdem aus ihrem Repository mergen willst, dann kannst Du dem `git pull` Befehl die URL ihres externen Repositories übergeben. Das lädt den entsprechenden Branch einmalig herunter und merged ihn in Deinen aktuellen Branch, ohne aber die externe URL als eine Referenz zu speichern:

```
$ git pull git://github.com/onetimeguy/project.git
From git://github.com/onetimeguy/project
 * branch                HEAD          -> FETCH_HEAD
Merge made by recursive.
```

Neuigkeiten durchsehen

Du hast jetzt einen Topic Branch, der die neuen Änderungen enthält, und kannst jetzt herausfinden, was Du damit anfangen willst. In diesem Abschnitt gehen wir noch mal auf einige Befehle ein, die nützlich sind, um herauszufinden, welche Änderungen Du übernehmen würdest, wenn Du den Topic Branch in Deinen Hauptbranch mergest.

Es ist in der Regel hilfreich, sich Commits anzusehen, die sich in diesem Branch, nicht aber im master Branch befinden. Du kannst Commits aus dem master Branch ausschließen, indem Du die `--not` Option verwendest. Wenn Du beispielsweise zwei neue Commits erhalten und sie in einen Topic Branch `contrib` übernommen hast, kannst Du folgendes tun:

```
$ 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

Wie Du schon gelernt hast, kannst Du außerdem die Option `-p` verwenden, um zu sehen, welche Diffs die Commits enthalten.

Wenn Du ein vollständiges Diff aller Änderungen sehen willst, die Dein Topic Branch gegenüber z.B. dem master Branch enthält, brauchst Du einen Trick. Möglicherweise würdest Du zuerst das hier ausprobieren:

```
$ git diff master
```

Der Befehl gibt Dir ein Diff aus, kann aber irreführend sein. Wenn im master Branch Änderungen committed wurden, seit der Branch angelegt wurde, erhältst Du scheinbar merkwürdige Ergebnisse. Das liegt daran, dass Git den Snapshot des letzten Commits des Topic Branches, in dem Du Dich momentan befindest, mit dem letzten Commit des master Branches vergleicht. Wenn Du beispielsweise eine Zeile in einer Datei im master branch hinzugefügt hast, scheint der direkte Vergleich auszusagen, dass diese Zeile im Topic Branch entfernt wurde.

Wenn master ein direkter Vorfahr Deines Topic Branches ist, ist das kein Problem. Aber wenn sich die beiden Historien auseinander bewegt (xxx) haben, dann scheint das Diff auszusagen, dass Du alle Neuigkeiten im Topic Branch hinzufügst und alle Neuigkeiten im master Branch entfernst.

Was Du aber eigentlich wissen willst, ist welche Änderungen der Topic Branch hinzugefügt hat, d.h. die Änderungen, die Du in den master Branch neu einführen würdest, wenn Du den Topic Branch mergest. Dieses Ergebnis erhältst Du, wenn Du den letzten Commit im Topic Branch mit dem letzten Commit vergleichst, den der Topic Branch mit master gemeinsam hat.

Technisch gesehen könntest Du den letzten gemeinsamen Commit explizit erfragen und dann ein Diff darauf ausführen:

```
$ git merge-base contrib master
36c7dba2c95e6bbb78dfa822519ecfec6e1ca649
$ git diff 36c7db
```

Das ist natürlich nicht sonderlich bequem, weshalb Git eine Kurzform dafür definiert: die triple-dot Syntax (xxx). Im Context des `git diff` Befehls bewirkt dies, dass Du ein Diff erhältst, das den letzten gemeinsamen Commit der Histories beider angegebenen Branches mit dem letzten Commit des zuletzt angegebenen Branches vergleicht:

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

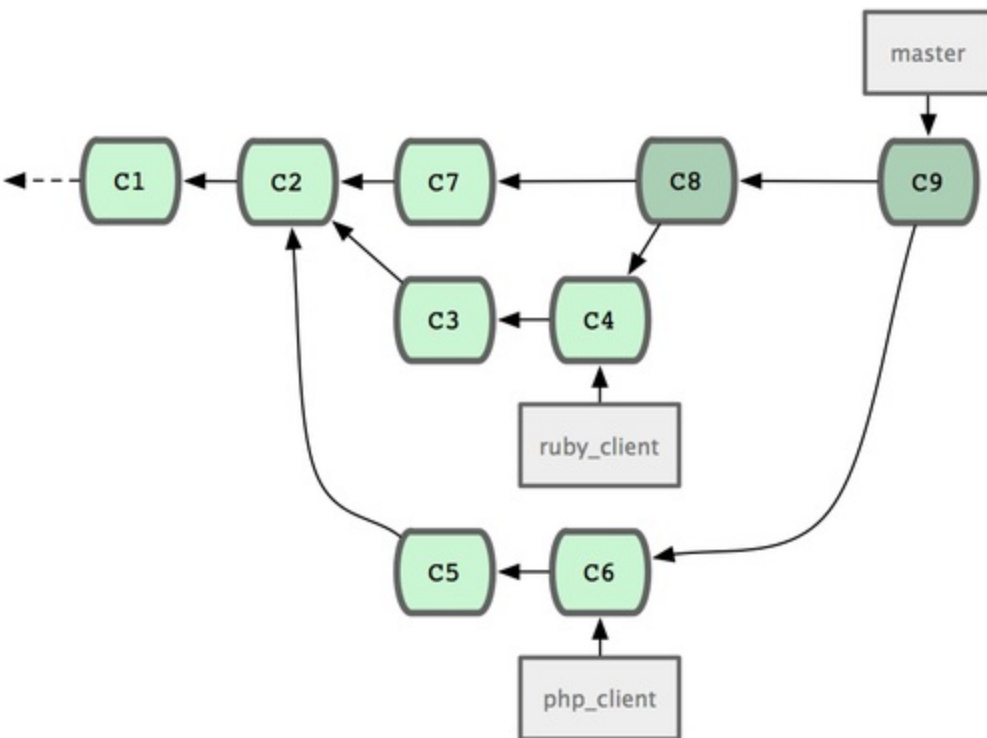
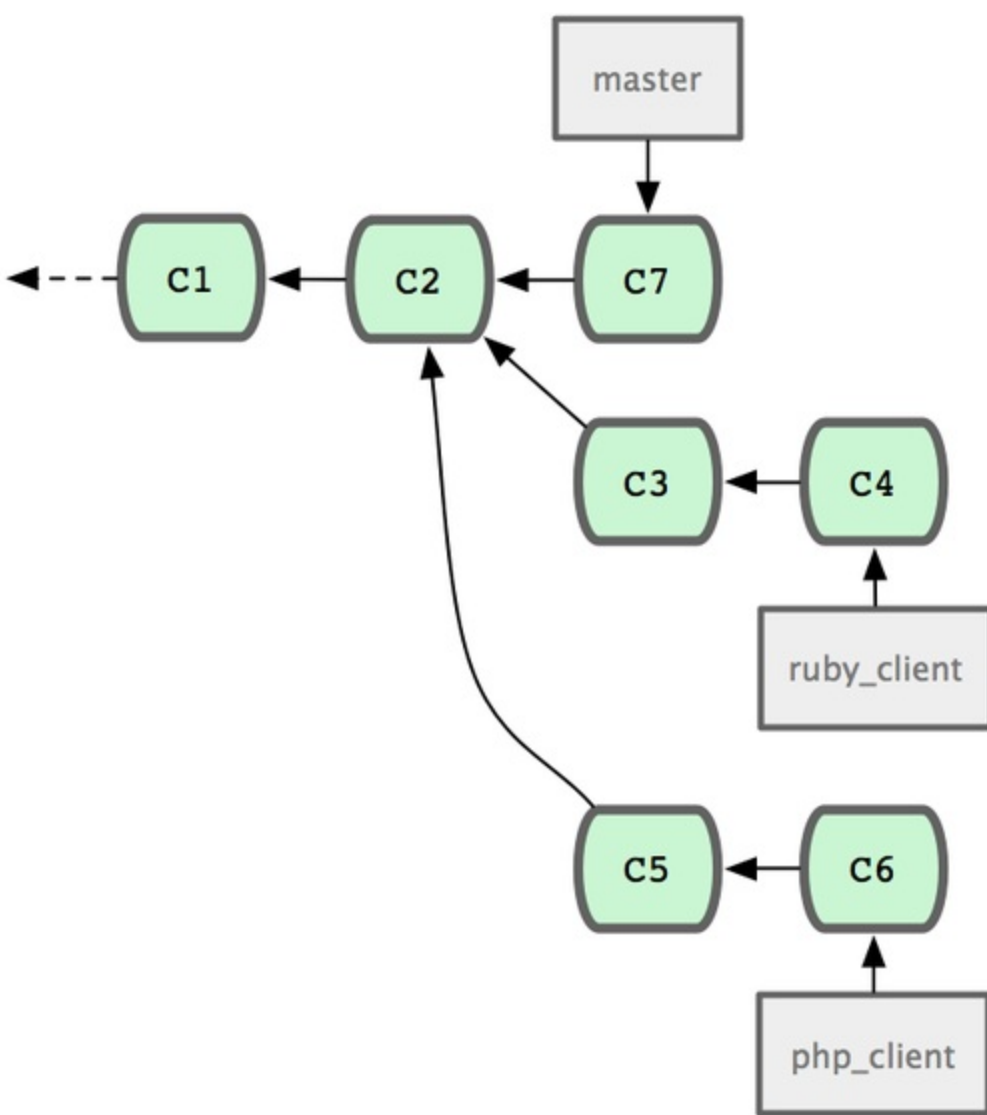
Dieser Befehl zeigt Dir diejenigen Änderungen, die im Topic Branch eingeführt wurden, die aber noch nicht in master enthalten sind.

Beiträge anderer integrieren

Sobald Du die Änderungen in Deinem Topic Branch in einen dauerhafteren Branch übernehmen willst, fragt sich, wie Du das anstellen kannst. Und welchen generellen Workflow willst Du verwenden, um das Projekt zu pflegen? Wir werden eine Reihe von Möglichkeiten besprechen, die Dir zur Verfügung stehen.

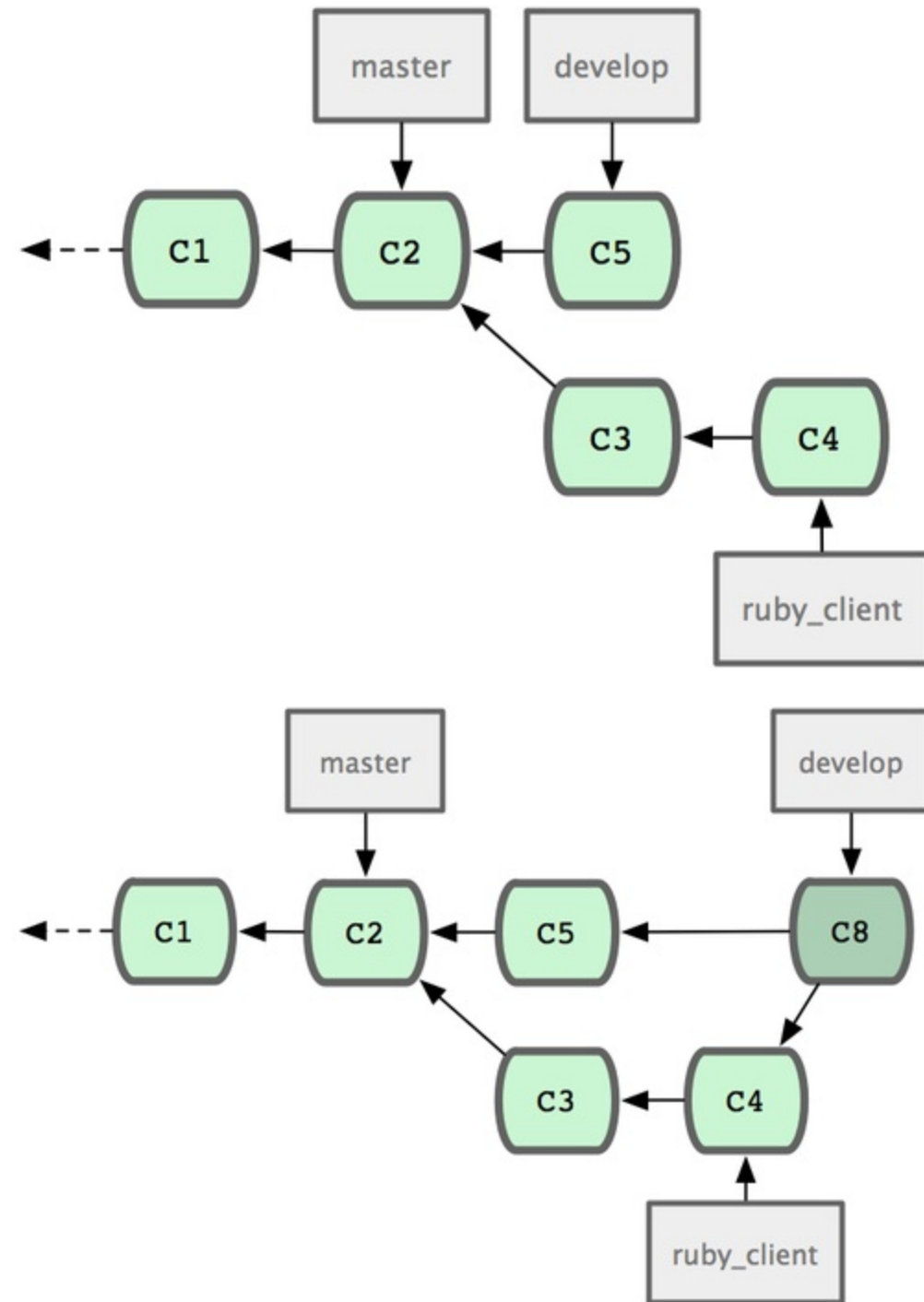
Merge Workflows

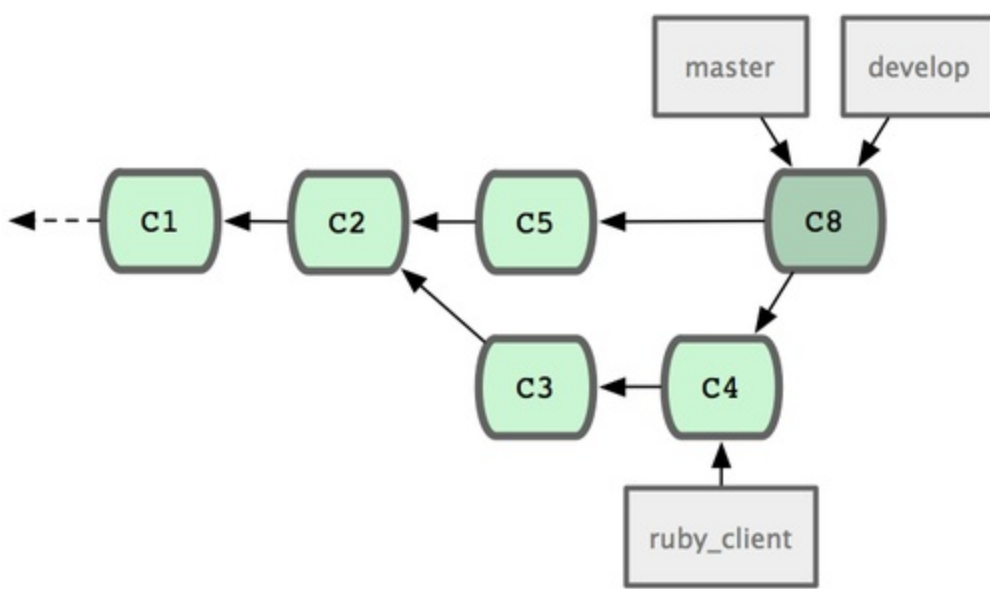
Eine einfache Möglichkeit besteht darin, Deine Arbeit einfach in den master Branch zu mergen. In diesem Workflow hast Du einen master Branch, der eine stabilen Code beinhaltet. Wenn Du Änderungen in einem Topic Branch hast, die von Dir selbst oder jemand anderem geschrieben und die verifiziert sind, dann mergest Du diesen Topic Branch in den master Branch, löschst den Topic Branch und fährst mit diesem Prozess so fort. Wenn es ein Repository mit zwei Branches gibt, die `ruby_client` und `php_client` heißen (wie in Bild 5-19), und Du mergest `ruby_client` zuerst und `php_client` danach, dann wird die Historie danach aussehen wie im Bild 5-20.



Dies ist vermutlich der einfachste, mögliche Workflow. Er ist allerdings für große Repositories oder Projekte manchmal problematisch.

Wenn Du mehr Entwickler oder ein größeres Projekt hast, wirst Du in der Regel einen Merge Zyklus mit mindestens zwei Phasen verwenden wollen. In einem solchen Workflow hast Du dann zwei dauerhafte Branches, z.B. master und develop, wobei master ausschließlich sehr stabile Releases enthält und neuer Code im develop Branch integriert wird. Jedes Mal, wenn Du einen neuen Topic Branch mergen willst, mergest Du ihn nach develop (Bild 5-22). Und nur dann, wenn Du einen Release taggen willst, führst Du ein fast-forward (xxx) it master bis zum gegenwärtigen, nun stabilen Status des develop Branch durch.

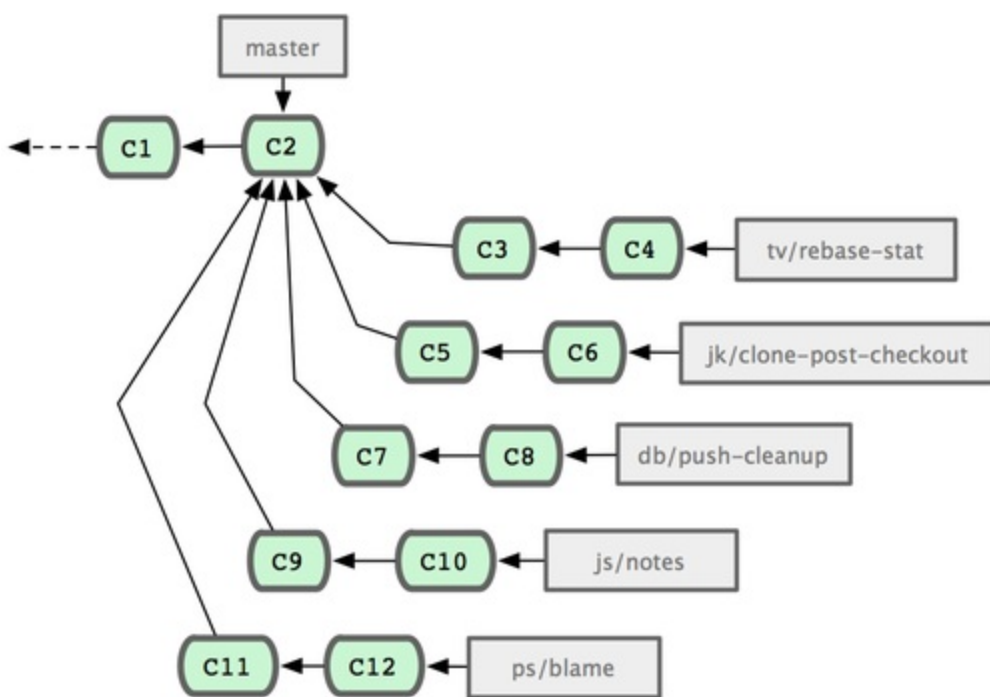




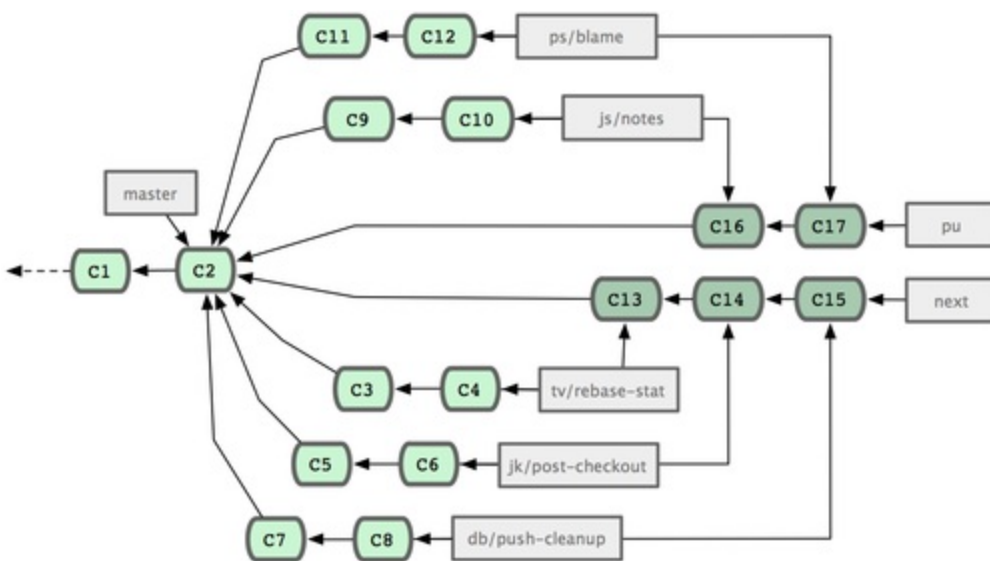
Auf diese Weise kann jeder, der Dein Repository klonet, auf einfache Weise Deinen aktuellen master Branch verwenden und ihn auf neue Releases aktualisieren. Oder er kann den develop Branch ausprobieren, in dem sich die jeweils letzten, brandneuen Änderungen befinden. Du kannst dieses Konzept noch weiterführen, indem Du einen integrate Branch pflegst, in den neue Änderungen jeweils integriert werden. Sobald der Code in diesem Branch stabil zu sein scheint und alle Tests durchlaufen (xxx), übernimmst Du die Änderungen in den develop Branch. Und wenn sie sich für eine Weile in der Praxis als stabil erwiesen haben, fast-forwardest (xxx) Du den master Branch.

Workflows für umfassende Merges

Das Git Projekt selbst hat view dauerhafte Branches: master, next, pu („proposed updates“, d.h. vorgeschlagene Änderungen) und maint („maintenance backports“, d.h. xxx Rückportierungen). Wenn neue Änderungen herein kommen, werden sie in Topic Branches im Projekt Repository gesammelt, ganz ähnlich wie wir gerade besprochen haben (siehe Bild 5-24). Dann wird evaluiert, ob die Änderungen sicher sind und übernommen werden sollen oder ob sie noch weiter bearbeitet werden müssen. Wenn sie übernommen werden sollen, werden sie in den Branch next gemerged und dieser Branch wird hochgeladen, sodass jeder ausprobieren kann, wie die neue Codebase funktioniert, nachdem die Änderungen miteinander integriert wurden.



Wenn die Topic Branches noch weiter bearbeitet werden müssen, werden sie statt dessen in den pu Branch gemerged. Wenn sie dann stabil sind, werden sie erneut in master gemerged und aus denjenigen Änderungen neu aufgebaut, die sich in next befanden, es aber noch nicht bis in den master Branch geschafft haben (xxx wie jetzt?? xxx). D.h., master bewegt sich fast ständig, next wird gelegentlich rebased, und pu wird noch sehr viel häufiger rebased (siehe Bild 5-25).

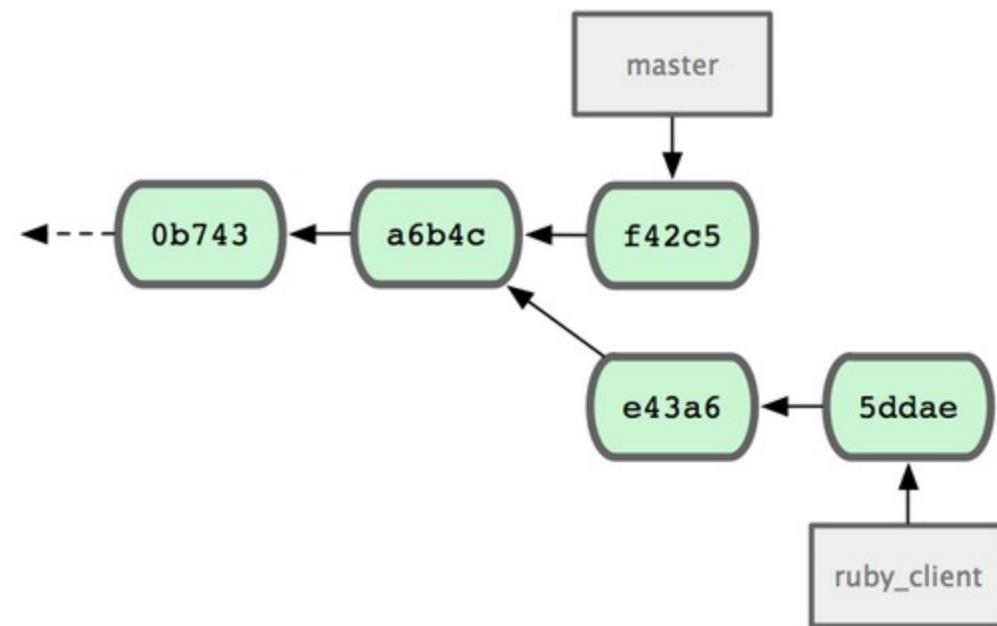


Wenn ein Topic Branch schließlich in master gemerged wird, wird er aus dem Repository gelöscht. Das Git Projekt hat außerdem einen maint Branch, der jeweils vom letzten Release verzweigt. In diesem Branch werden rückportierte Patches für den Fall gesammelt, dass ein Maintenance Release nötig ist. D.h., wenn Du das Git Projekt Repository klonst, findest Du vier Branches des Projektes in verschiedenen Stadien, die Du jeweils ausprobieren kannst, je nachdem wie hochaktuellen Code Du testen oder wie Du zu dem Projekt beitragen willst. Und der Projekt Betreiber hat auf diese Weise einen klar strukturierten Workflow, der es einfacher macht, neue Beiträge zu prüfen und zu verarbeiten.

Rebase und Cherry Picking Workflows

Andere Betreiber bevorzugen, neue Änderungen auf der Basis ihres master Branches zu rebasen oder zu cherry-picken statt sie zu mergen, um auf diese Weise eine eher lineare Historie zu erhalten. Wenn Du Änderungen in einem Topic Branch hast, die Du integrieren willst, dann gehst Du in diesen Branch und führst den rebase Befehl aus, um diese Änderungen auf der Basis des gegenwärtigen master Branches (oder irgendeines anderen, stabileren Branches) neu zu schreiben. Wenn das glatt läuft, kannst Du den master Branch fast-forwarden (xxx) und erhältst so eine lineare Projekt Historie.

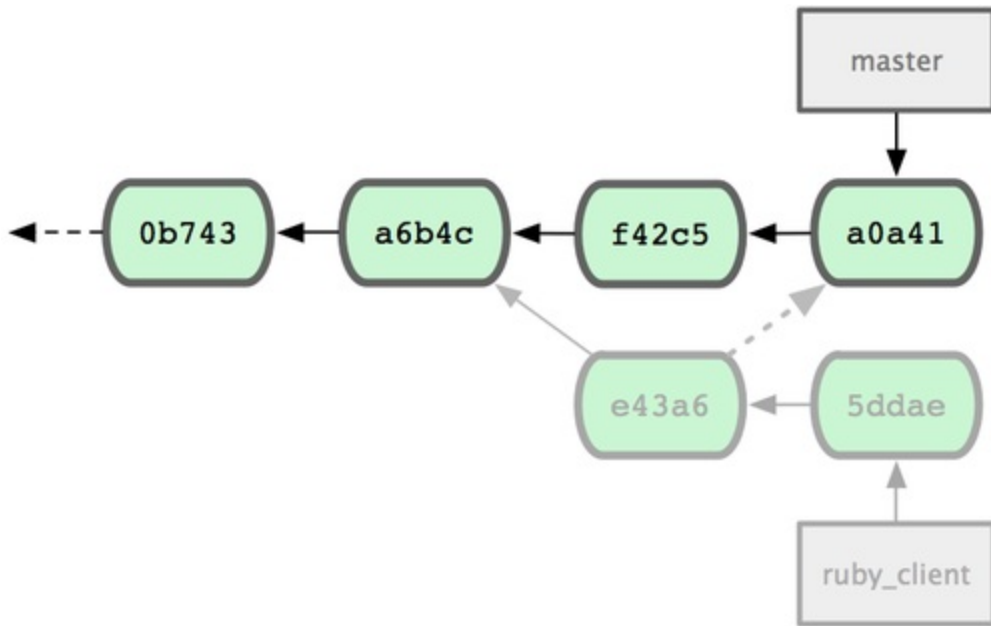
Eine andere Möglichkeit, Commits aus einem Branch in einen anderen zu übernehmen ist der cherry-pick Befehl. In Git ist dieser Befehl quasi ein rebase für einen einzelnen Commit. Er nimmt den Patch, der mit dem Commit eingeführt wurde, und versucht, diesen auf den Branch anzuwenden, in dem Du Dich gerade befindest. Das ist nützlich, wenn Du in einem Topic Branch eine Anzahl von Commits hast, aber lediglich einen davon übernehmen willst. Oder wenn Du überhaupt nur einen Commit im Topic Branch hast, diesen aber lieber cherry-picken willst, statt den ganzen Branch zu rebasen. Nehmen wir z.B. an, Du hast ein Projekt, das so aussieht wie in Bild 5-26.



Wenn Du den Commit e43a6 in Deinen master Branch übernehmen willst, kannst Du folgendes ausführen:

```
$ git cherry-pick e43a6fd3e94888d76779ad79fb568ed180e5fcdf
Finished one cherry-pick.
[master]: created a0a41a9: "More friendly message when locking the index fails."
3 files changed, 17 insertions(+), 3 deletions(-)
```

Das wendet dieselben Änderungen, die in e43a6 eingeführt wurden, auf den master Branch an, aber Du erhältst einen neuen Commit SHA-1 Hash, weil auch das Datum ein anderes ist. Jetzt sieht Deine Historie so aus:



Jetzt kannst Du den Topic Branch inklusive der ggf. darin enthaltenen Commits löschen, falls Du sie nicht noch übernehmen willst.

Releases taggen

Wenn Du einen Release herausgeben willst, ist es empfehlenswert, einen Tag dafür anzulegen, sodass man den jeweiligen Zustand der Historie jederzeit leicht wiederherstellen kann. Wir sind bereits in Kapitel 2 auf Git Tags eingegangen. Wenn Du als Betreiber den neuen Tag signieren willst, könnte das wie folgt aussehen:

```

$ git tag -s v1.5 -m 'my signed 1.5 tag'
You need a passphrase to unlock the secret key for
user: "Scott Chacon <schacon@gmail.com>"
1024-bit DSA key, ID F721C45A, created 2009-02-09
  
```

Wenn Du Deine Tags signierst, könnte das Problem bestehen, dass Du den jeweiligen öffentlichen PGP key zur Verfügung stellen musst. Der Betreiber des Git Projektes löst das, in dem er den öffentlichen Schlüssel als Inhalt im Repository selbst zur Verfügung stellt und einen Tag hat, der direkt auf diesen Inhalt zeigt. Um das zu tun, musst Du zunächst herausfinden, welchen Schlüssel Du verwenden willst:

```

$ 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]
  
```

Dann kannst Du den Schlüssel direkt in die Git Datenbank importieren, indem Du ihn aus GPG exportierst und die Ausgabe nach `git hash-object` weiterreichst. Das schreibt ein neues Objekt mit dem Schlüssel in die Git Datenbank und gibt Dir einen SHA-1 Hash zurück, der dieses Objekt referenziert:

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

Nachdem Du jetzt den Schlüssel im Repository hast, kannst Du einen Tag für den SHA-1 Hash anlegen, den `git hash-object` zurückgegeben hat:

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

Wenn Du `git push --tags` ausführst, wird jetzt der `maintainer-pgp-pub` Tag auf den Server geladen, sodass jeder darauf zugreifen kann. Wenn jemand jetzt einen signierten Tag verifizieren will, kann er Deinen öffentlichen PGP Schlüssel direkt aus der Datenbank holen und in seinen Schlüsselbund importieren:

```
$ git show maintainer-pgp-pub | gpg --import
```

Dieser Schlüssel kann anschließend für alle signierten Tags verwendet werden. Zusätzlich kannst Du Deinen Anwendern in der Tag Meldung erklären, wie sie signierte Tags mit diesem Schlüssel verifizieren können.

Eine Build Nummer generieren

Weil Git keine globalen Nummern wie `v123` kennt, die mit jedem Commit monoton hochgezählt werden, kannst Du, um einen leicht lesbaren Bezeichner für einen bestimmten Commit zu erhalten, den Befehl `git describe` auf diesen Befehl ausführen. Git erzeugt dann einen Bezeichner zurück, der den Namen des nächsten Tags enthält, der Anzahl der Commits seit diesem Tag und die ersten Zeichen des SHA-1 Hashs des Commits:

```
$ git describe master  
v1.6.2-rc1-20-g8c5b85c
```

Auf diese Weise kannst Du in einer Weise auf einen Commit oder Build verweisen, der für Andere leichter verständlich ist. Wenn Du z.B. Git selbst aus dem Quellcode kompilierst, der sich im Git Projekt Repository befindet, dann gibt `git --version` einen ähnlichen Bezeichner zurück. Wenn Du übrigens `git describe` auf einen Commit ausführst, den Du direkt getagged hast, dann erhältst Du statt dessen den Tag Namen.

Der `git describe` Befehl funktioniert mit kommentierten Tags besser (d.h. Tags, die mit dem `-a` oder `-s` Flag erzeugt wurden), sodass es sich empfiehlt, Release Tags auf diese Weise anzulegen, wenn man `git describe` verwenden will. Du kannst diese Bezeichner auch als Parameter für andere Git Befehle, z.B. `git checkout` oder `git show`, wobei Git allerdings lediglich auf den abgekürzten SHA-1 Hash am Ende achtet, sodass er möglicherweise nicht ewig gültig ist. Das Linux Kernel Projekt beispielsweise erhöhte die Anzahl der Zeichen in abgekürzten Hashes kürzlich von 8 auf 10, um die Eindeutigkeit von SHA-1 Hashes sicherzustellen. Ältere `git describe` Ausgaben wurden damit ungültig.

Ein Release vorbereiten

Du willst jetzt ein Release herausgeben. Dazu willst Du u.a. ein Archiv mit dem letzten Snapshot Deines Codes erzeugen, damit ihn auch arme Seelen herunterladen können, die Git nicht verwenden. Der folgende Befehl hilft Dir dabei:

```
$ git archive master --prefix='project/' | gzip > `git describe master`.tar.gz
$ ls *.tar.gz
v1.6.2-rc1-20-g8c5b85c.tar.gz
```

Das erzeugt einen Tarball, der den aktuellen Snapshot in Deinem Arbeitsverzeichnis enthält. Du kannst auf die gleiche Weise ein Zip Archiv erzeugen, indem Du `git archive` die `--format=zip` Option übergibst.

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

Du hast jetzt sowohl einen Tarball als auch ein Zip Archiv Deines Releases. Diese kannst Du z.B. auf Deiner Webseite publizieren oder auch per E-Mail verschicken.

Das Shortlog

Es wird Zeit, den Lesern der Mailingliste zu erklären, was es im Projekt Neues gibt. Der `git shortlog` Befehl ist eine Möglichkeit, schnell eine Art Changelog der Änderungen seit dem letzten Release auszugeben. Er fasst alle Commits in der angegebenen Zeitspanne zusammen. Der folgende Befehl z.B. erzeugt eine Zusammenfassung der Commits seit dem letzten Release, der als `v1.0.1` getagged wurde:

```
$ git shortlog --no-merges master --not v1.0.1
Chris Wanstrath (8):
    Add support for annotated tags to Grit::Tag
    Add packed-refs annotated tag support.
    Add Grit::Commit#to_patch
    Update version and History.txt
    Remove stray `puts`
    Make ls_tree ignore nils
```

```
Tom Preston-Werner (4):
    fix dates in history
    dynamic version method
    Version bump to 1.0.2
    Regenerated gems spec for version 1.0.2
```

Du erhältst eine saubere Auflistung aller Commits seit `v1.0.1`, gruppiert nach Autor. Diese kannst Du z.B. an die Mailingliste schicken oder irgendwie anders publizieren.

Zusammenfassung

Du solltest Dich jetzt einigermaßen vertraut damit fühlen, sowohl Beiträge bei einem bestehenden Projekt einzureichen als auch selbst ein eigenes Projekt zu betreiben und Beiträge anderer zu integrieren. Herzlichen Glückwunsch, Du bist jetzt ein erfolgreicher Git Entwickler! (xxx hä? xxx) Im nächsten Kapitel wirst Du weitere mächtige Git Werkzeuge und Tipps dafür kennenlernen, mit komplexen Situationen umzugehen – die einen wahren Git Meister aus Dir werden. (xxx aha? xxx)

Git Tools

Bis hierher hast Du die meisten täglichen Kommandos und Arbeitsweisen gelernt, die Du brauchst um ein Git-Repository für Deine Quellcode-Kontrolle, zu benutzen. Du hast die grundlegenden Aufgaben des Nachverfolgens und Eincheckens von Dateien gemeistert und Du hast von der Macht der Staging-Area, des Branching und des Mergens Gebrauch gemacht.

Als Nächstes werden wir einige sehr mächtige Werkzeuge besprechen, die Dir Git zur Verfügung stellt. Du wirst zwar nicht unbedingt jeden Tag verwenden, aber mit Sicherheit an einem bestimmten Punkt gut brauchen können.

Revision Auswahl

Git erlaubt Dir, Commits auf verschiedenste Art und Weise auszuwählen. Diese sind nicht immer offensichtlich, aber es ist hilfreich diese zu kennen.

Einzelne Revisionen

Du kannst offensichtlich mithilfe des SHA-1-Hashes einen Commit auswählen, aber es gibt auch menschenfreundlichere Methoden, auf einen Commit zu verweisen. Dieser Bereich skizziert die verschiedenen Wege, die man gehen kann, um sich auf ein einzelnen Commit zu beziehen.

Abgekürztes SHA

Git ist intelligent genug, den richtigen Commit herauszufinden, wenn man nur die ersten paar Zeichen angibt, aber nur unter der Bedingung, dass der SHA-1-Hash mindestens vier Zeichen lang und einzigartig ist — das bedeutet, dass es nur ein Objekt im derzeitigen Repository gibt, das mit diesem bestimmten SHA-1-Hash beginnt.

Um zum Beispiel einen bestimmten Commit zu sehen, kann man das `git log` Kommando ausführen und den Commit identifizieren, indem man eine bestimmte Funktionalität hinzugefügt hat:

```
$ 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
```

In diesem Fall wählt man `1c002dd...`, wenn man diesen Commit mit `git show` anzeigen lassen will, die folgenden Kommandos sind äquivalent (vorausgesetzt die verkürzte Version ist einzigartig):

```
$ git show 1c002dd4b536e7479fe34593e72e6c6c1819e53b
$ git show 1c002dd4b536e7479f
$ git show 1c002d
```

Git kann auch selber eine Kurzform für Deine einzigartigen SHA-1-Werte erzeugen. Wenn Du `--abbrev-commit` dem `git log` Kommando übergibst, wird es den kürzeren Wert benutzen, diesen aber einzigartig halten; die Standardeinstellung sind sieben Zeichen, aber es werden automatisch mehr benutzt, wenn dies nötig ist, um den SHA-1-Hash eindeutig zu bezeichnen.

```
$ git log --abbrev-commit --pretty=oneline
ca82a6d changed the version number
085bb3b removed unnecessary test code
a11bef0 first commit
```

Generell kann man sagen, dass acht bis zehn Zeichen mehr als ausreichend in einem Projekt sind, um eindeutig zu bleiben. Eines der größten Git-Projekte, der Linux-Kernel, fängt langsam an 12 von maximal 40 Zeichen zu nutzen, um eindeutig zu bleiben.

Ein kurzer Hinweis zu SHA-1

Eine Menge Leute machen sich Sorgen, dass ab einem zufälligen Punkt zwei Objekte im Repository vorhanden sind, die den gleichen SHA-1-Hashwert haben. Was dann?

Wenn es passiert, dass bei einem Commit, ein Objekt mit dem gleichen SHA-1-Hashwert im Repository vorhanden ist, wird Git sehen, dass das vorherige Objekt bereits in der Datenbank vorhanden ist und davon ausgehen, dass es bereits geschrieben wurde. Wenn Du versuchst, das Objekt später wieder auszuchecken, wirst Du immer die Daten des ersten Objekts bekommen.

Allerdings solltest Du Dir bewusst machen, wie unglaublich unwahrscheinlich dieses Szenario ist. Die Länge des SHA-1-Hashs beträgt 20 Bytes oder 160 Bits. Die Anzahl der Objekte, die benötigt werden, um eine 50% Chance einer Kollision zu haben, beträgt ungefähr 2^{80} (die Formel zum Berechnen der Kollisionswahrscheinlichkeit lautet $p = (n(n-1)/2) * (1/2^{160})$). 2^{80} ist somit 1.2×10^{24} oder eine Trilliarde. Das ist 1200 Mal so viel, wie es Sandkörner auf der Erde gibt.

Hier ist ein Beispiel, das Dir eine Vorstellung davon gibt, was nötig ist, um in SHA-1 eine Kollision zu bekommen. Wenn alle 6,5 Milliarden Menschen auf der Erde programmieren würden und jeder jede Sekunde Code schreiben würde, der der gesamten Geschichte des Linux-Kernels (1 Million Git-Objekte) entspricht, und diesen dann in ein gigantisches Git-Repository übertragen würde, würde es fünf Jahre dauern, bis das Repository genügend Objekte hätte, um eine 50% Wahrscheinlichkeit für eine einzige SHA-1-Kollision aufzuweisen. Es ist wahrscheinlicher, dass jedes Mitglied Deines Programmierer-Teams, unabhängig voneinander, in einer Nacht von Wölfen angegriffen und getötet wird.

Branch-Referenzen

Am direktesten kannst Du einen Commit spezifizieren, wenn eine Branch-Referenz direkt auf ihn zeigt. In dem Fall kannst Du in allen Git-Befehlen, die ein Commit-Objekt oder einen SHA-1-Wert erwarten, stattdessen den Branch-Namen verwenden. Wenn Du z.B. den letzten Commit in einem Branch sehen willst, sind die folgenden Befehle äquivalent (vorausgesetzt der `topic1` Branch zeigt auf den Commit `ca82a6d`):

```
$ git show ca82a6dff817ec66f44342007202690a93763949
$ git show topic1
```

Wenn Du sehen willst, auf welchen SHA-1-Wert ein Branch zeigt, oder wie unsere Beispiele intern in SHA-1-Werte übersetzt aussähen, kannst Du den Git-Plumbing-Befehl `rev-parse` verwenden. In Kapitel 9 werden wir genauer auf Plumbing-Befehle eingehen. Kurz gesagt ist `rev-parse` als eine Low-Level-Operation gedacht und nicht dafür, im tagtäglichen Gebrauch eingesetzt zu werden. Aber es kann manchmal hilfreich sein, wenn man wissen muss, was unter der Haube vor sich geht:

```
$ git rev-parse topic1
ca82a6dff817ec66f44342007202690a93763949
```

RefLog Kurznamen

Eine andere Sache, die während Deiner täglichen Arbeit im Hintergrund passiert ist, dass Git ein sogenanntes Reflog führt, d.h. ein Log darüber, wohin Deine HEAD- und Branch-Referenzen in den letzten Monaten jeweils gezeigt haben.

Du kannst das Reflog mit `git reflog` anzeigen:

```
$ git reflog
734713b... HEAD@{0}: commit: fixed refs handling, added gc auto, updated
d921970... HEAD@{1}: merge phedders/rdocs: Merge made by recursive.
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
```

Immer dann, wenn ein Branch in irgendeiner Weise aktualisiert wird oder Du den aktuellen Branch wechselst, speichert Git diese Information ebenso im Reflog wie Commits und anderen Informationen. Wenn Du wissen willst, welches der fünfte Wert vor dem HEAD war, kannst Du die `@{n}` Referenz angeben, die Du in Reflog-Ausgabe sehen kannst:

```
$ git show HEAD@{5}
```

Außerdem kannst Du dieselbe Syntax verwenden, um eine Zeitspanne anzugeben. Um z.B. zu sehen, wo Dein master Branch gestern war, kannst Du eingeben:

```
$ git show master@{yesterday}
```

Das zeigt Dir, wo der master Branch gestern war. Diese Technik funktioniert nur mit Daten, die noch im Reflog sind, d.h. man kann sie nicht für Commits verwenden, die ein älter sind als ein paar Monate.

Um Reflog Informationen in einem Format wie in `git log` anzeigen, kannst Du `git log -g` verwenden:

```
$ 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 wichtig zu verstehen, dass das Reflog ausschließlich lokale Daten enthält. Es ist ein Log darüber, was Du in Deinem Repository getan hast, und es ist nie dasselbe wie in einem anderen Klon des selben Repositories. Direkt nachdem Du ein Repository geklont hast, ist das Reflog leer, weil noch keine weitere Aktivität stattgefunden hat. `git show HEAD@{2.months.ago}` funktioniert nur dann, wenn das Projekt mindestens zwei Monate alt ist – wenn Du es vor fünf Minuten erst geklont hast, erhältst Du keine Ergebnisse.

Vorfahren Referenzen

Außerdem kann man Commits über ihre Vorfahren spezifizieren. Wenn Du ein `^` ans Ende einer Referenz setzt, schlägt Git den direkten Vorfahren dieses Commits nach. Nehmen wir an, Deine Historie sieht so aus:

```
$ 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
```

Du kannst jetzt den vorletzten Commit mit `HEAD^` referenzieren, d.h. „den direkten Vorfahren von `HEAD`“.

```
$ git show HEAD^
commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800
```

Merge commit 'phedders/rdocs'

Außerdem kannst Du nach dem ^ eine Zahl angeben. Beispielsweise heißt d921970^2: „der zweite Vorfahr von d921970“. Diese Syntax ist allerdings nur für Merge Commits nützlich, die mehr als einen Vorfahren haben. Der erste Vorfahr ist der Branch, auf dem Du Dich beim Merge befandest, der zweite ist der Commit auf dem Branch, den Du gemergt hast.

```
$ 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

Eine andere Vorfahren-Spezifikation ist ~. Dies bezieht sich ebenfalls auf den ersten Vorfahren, d.h. HEAD~ und HEAD^ sind äquivalent. Einen Unterschied macht es allerdings, wenn Du außerdem eine Zahl angibst. HEAD~2 bedeutet z.B. „der Vorfahr des Vorfahren von HEAD“ oder „der n-te Vorfahr von HEAD“. Beispielsweise würde HEAD~3 in der obigen Historie auf den folgenden Commit zeigen:

```
$ git show HEAD~3
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500
```

ignore *.gem

Dasselbe kannst Du mit HEAD^^^ angeben, was wiederum den „Vorfahren des Vorfahren des Vorfahren“ referenziert:

```
$ git show HEAD^^^
commit 1c3618887afb5fbcbea25b7c013f4e2114448b8d
Author: Tom Preston-Werner <tom@mojombo.com>
Date: Fri Nov 7 13:47:59 2008 -0500
```

ignore *.gem

Du kannst diese Schreibweisen auch kombinieren und z.B. auf den zweiten Vorfahren der obigen Referenz mit HEAD~3^2 zugreifen.

Commit Reihen

Nachdem Du jetzt einzelne Commits spezifizieren kannst, schauen wir uns an, wie man auf Commit-Reihen zugreift. Dies ist vor allem nützlich, um Branches zu verwalten, z.B. wenn man viele Branches hat und solche Fragen beantworten will wie „Welche Änderungen befinden sich in

diesem Branch, die ich noch nicht in meinen Hauptbranch gemergt habe?“.

Zwei-Punkte-Syntax

Die gängigste Weise, Commit-Reihen anzugeben, ist die Zwei-Punkte-Notation. Allgemein gesagt liefert Git damit eine Reihe von Commits, die von dem einem Commit aus erreichbar sind, allerdings nicht von dem anderen aus. Nehmen wir z.B. an, Du hättest eine Commit-Historie wie die folgende (Bild 6-1).

Insert 18333fig0601.png

Du willst jetzt herausfinden, welche Änderungen in Deinem `experiment`-Branch sind, die noch nicht in den `master`-Branch gemergt wurden. Dann kannst Du ein Log dieser Commits mit `master..experiment` anzeigen, d.h. „alle Commits, die von `experiment` aus erreichbar sind, aber nicht von `master`“. Um die folgenden Beispiele ein bisschen abzukürzen und deutlicher zu machen, verwende ich für die Commit-Objekte die Buchstaben aus dem Diagramm anstelle der tatsächlichen Log Ausgabe:

```
$ git log master..experiment
D
C
```

Wenn Du allerdings – anders herum – diejenigen Commits anzeigen willst, die in `master`, aber noch nicht in `experiment` sind, kannst Du die Branch-Namen umdrehen: `experiment..master` zeigt „alles in `master`, das nicht in `experiment` enthalten ist“.

```
$ git log experiment..master
F
E
```

Dies ist nützlich, wenn Du vorhast, den `experiment`-Branch zu aktualisieren, und anzeigen willst, was Du dazu mergen wirst. Oder wenn Du vorhast, in ein Remote-Repository zu pushen und sehen willst, welche Commits betroffen sind:

```
$ git log origin/master..HEAD
```

Dieser Befehl zeigt Dir alle Commits im gegenwärtigen, lokalen Branch, die noch nicht im `master`-Branch des `origin` Repositorys sind. D.h., der Befehl listet diejenigen Commits auf, die auf den Server transferiert würden, wenn Du `git push` benutzt und der aktuelle Branch `origin/master` trackt. Du kannst mit dieser Syntax außerdem eine Seite der beiden Punkte leer lassen. Git nimmt dann an, Du meinst an dieser Stelle `HEAD`. Z.B. kannst Du dieselben Commits wie im vorherigen Beispiel auch mit `git log origin/master..` anzeigen lassen. Git fügt dann `HEAD` auf der rechten Seite ein.

Mehrere Bezugspunkte

Die Zwei-Punkte-Syntax ist eine nützliche Abkürzung, aber möglicherweise willst Du mehr als

nur zwei Branches angeben, um z.B. herauszufinden, welche Commits in einem beliebigen anderen Branch enthalten sind, ausgenommen in demjenigen, auf dem Du Dich gerade befindest. Dazu kannst Du in Git das ^ Zeichen oder --not verwenden, um Commits auszuschließen, die von den angegebenen Referenzen aus erreichbar sind. D.h., die folgenden drei Befehle sind äquivalent:

```
$ git log refA..refB
$ git log ^refA refB
$ git log refB --not refA
```

Das ist praktisch, weil Du auf diese Weise mehr als nur zwei Referenzen angeben kannst, was mit der Zwei-Punkte-Notation nicht geht. Wenn Du beispielsweise alle Commits sehen willst, die von refA oder refB erreichbar sind, nicht aber von refC, dann kannst Du folgende (äquivalente) Befehle benutzen:

```
$ git log refA refB ^refC
$ git log refA refB --not refC
```

Damit hast Du ein sehr mächtiges System von Abfragen zur Verfügung, mit denen Du herausfinden kannst, was in welchen Deiner Branches enthalten ist.

Drei-Punkte-Syntax

Die letzte wichtige Syntax, mit der man Commit-Reihen spezifizieren kann, ist die Drei-Punkte-Syntax, die alle Commits anzeigt, die in einer der beiden Referenzen enthalten sind, aber nicht in beiden. Schau Dir noch mal die Commit Historie in Bild 6-1 an. Wenn Du diejenigen Commits anzeigen willst, die in den master- und experiment-Banches, nicht aber in beiden Branches gleichzeitig enthalten sind, dann kannst Du folgendes tun:

```
$ git log master...experiment
F
E
D
C
```

Dies gibt Dir wiederum ein normale log Ausgabe, aber zeigt nur die Informationen dieser vier Commits – wie üblich sortiert nach dem Commit-Datum.

Eine nützliche Option für den log Befehl ist in diesem Fall --left-right. Sie zeigt Dir an, in welchem der beiden Branches der jeweilige Commit enthalten ist, sodass die Ausgabe noch nützlicher ist:

```
$ git log --left-right master...experiment
< F
< E
> D
> C
```

Mit diesen Hilfsmitteln kannst Du noch einfacher und genauer angeben, welche Commits Du

nachschlagen willst.

Interaktives Stagen

Git umfasst eine Reihe von Skripten, die so manche Aufgabe auf der Kommandozeile leichter machen. Im Folgenden schauen wir uns einige interaktive Befehle an, die dabei hilfreich sein können, wenn man Änderungen in vielen Dateien vorgenommen hat, aber nur einige Änderungen gezielt committen will – nicht alles auf einmal in einem riesigen Commit. Auf diese Weise kann man Commits logisch gruppieren und macht es anderen Entwicklern damit leichter, sie zu verstehen. Wenn Du `git add` mit der `-i` oder `--interactive` Option verwendest, geht Git in einen interaktiven Shell-Modus, der in etwa wie folgt aussieht:

```
$ git add -i
      staged      unstaged path
1:    unchanged    +0/-1  TODO
2:    unchanged    +1/-1  index.html
3:    unchanged    +5/-1  lib/simplegit.rb

*** Commands ***
1: status      2: update      3: revert      4: add untracked
5: patch       6: diff        7: quit        8: help
What now>
```

Wie Du siehst, zeigt dieser Befehl eine andere Ansicht der Staging-Area an – im Wesentlichen also die Information, die Du auch mit `git status` erhältst, aber anders formatiert, kurz und knapp, und informativer. Sie listet alle Änderungen, die in der Staging-Area enthalten sind, auf der linken Seite, und alle anderen Änderungen auf der rechten Seite.

Danach folgt eine Liste von Befehlen wie, u.a., Dateien ganz oder teilweise stagen und unstagen, nicht versionskontrollierte Dateien hinzufügen, Diffs der gestageten Änderungen anzeigen etc.

Hinzufügen und Entfernen von Dateien aus der Staging-Area

Wenn Du am `what now>` Prompt 2 oder u eingibst, wirst Du als Nächstes gefragt, welche Dateien Du stagen willst:

```
What now> 2
      staged      unstaged path
1:    unchanged    +0/-1  TODO
2:    unchanged    +1/-1  index.html
3:    unchanged    +5/-1  lib/simplegit.rb
Update>>
```

Um z.B. die `TODO` und `index.html` Dateien zu stagen, gibst Du die jeweiligen Zahlen ein:

```
Update>> 1,2
      staged      unstaged path
* 1:    unchanged    +0/-1  TODO
* 2:    unchanged    +1/-1  index.html
3:    unchanged    +5/-1  lib/simplegit.rb
Update>>
```

Das * neben den Dateinamen bedeutet, dass die Datei ausgewählt ist und zur Staging-Area hinzugefügt werden wird, sobald Du (bei einem sonst leeren update>> Prompt) Enter drückst:

```
Update>>
updated 2 paths

*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit        8: help
What now> 1

      staged      unstaged path
 1:      +0/-1      nothing TODO
 2:      +1/-1      nothing index.html
 3:      unchanged      +5/-1 lib/simplegit.rb
```

Du kannst sehen, dass die TODO und index.html Dateien jetzt gestaget sind, während simplegit.rb immer noch ungestaget ist. Wenn Du die TODO unstagen willst, kannst Du die Option 3 oder r (für revert) nutzen:

```
*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit        8: help
What now> 3

      staged      unstaged path
 1:      +0/-1      nothing TODO
 2:      +1/-1      nothing index.html
 3:      unchanged      +5/-1 lib/simplegit.rb
Revert>> 1

      staged      unstaged path
* 1:      +0/-1      nothing TODO
 2:      +1/-1      nothing index.html
 3:      unchanged      +5/-1 lib/simplegit.rb
Revert>> [enter]
reverted one path
```

Wenn Du wiederum Deinen Git-Status ansiehst, kannst Du sehen, dass Du die TODO ungestaget hast.

```
*** Commands ***
 1: status      2: update      3: revert      4: add untracked
 5: patch       6: diff        7: quit        8: help
What now> 1

      staged      unstaged path
 1:      unchanged      +0/-1 TODO
 2:      +1/-1      nothing index.html
 3:      unchanged      +5/-1 lib/simplegit.rb
```

Um einen Diff dessen zu sehen, das Du gestaget hast, kannst Du den Befehl 6 oder d (für diff) nutzen. Dieser zeigt Dir eine Liste der gestageten Dateien, und Du kannst diejenigen auswählen, von denen Du den gestageten Diff sehen willst. Dies ähnelt sehr dem Befehl `git diff --cached` auf der Kommandozeile.

```

*** Commands ***
  1: status      2: update      3: revert      4: add untracked
  5: patch       6: diff       7: quit        8: help
What now> 6
      staged      unstaged path
  1:      +1/-1      nothing index.html
Review diff>> 1
diff --git a/index.html b/index.html
index 4d07108..4335f49 100644
--- a/index.html
+++ b/index.html
@@ -16,7 +16,7 @@ Date Finder

```

<p id="out">...</p>

```

-<div id="footer">contact : support@github.com</div>
+<div id="footer">contact : email.support@github.com</div>

<script type="text/javascript">

```

Mit diesen grundlegenden Befehlen kannst Du den interaktiven Hinzufüge-Modus nutzen, um Dir den Umgang mit Deiner Staging-Area etwas zu erleichtern.

Patches stagen

Es ist für Git auch möglich, bestimmte Teile einer Datei zu stagen und nicht den Rest. Wenn Du z.B. zwei Veränderungen an der simplegit.rb machst und eine davon stagen willst und die andere nicht, ist dies sehr einfach in Git möglich. Wähle 5 oder p (für patch) auf dem interaktiven Prompt. Git wird Dich fragen, welche Dateien Du teilweise stagen willst; dann wird es für jeden Abschnitt der gewählten Dateien Diff-Ausschnitte ausgeben und Dich jeweils einzeln fragen, ob Du sie stagen willst.

```

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index dd5ecc4..57399e0 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -22,7 +22,7 @@ class SimpleGit
  end

  def log(treeish = 'master')
-    command("git log -n 25 #{treeish}")
+    command("git log -n 30 #{treeish}")
  end

  def blame(path)
Stage this hunk [y,n,a,d,/,j,J,g,e,]?

```

Du hast an diesem Punkt viele Optionen. Tippe ? ein, um eine Liste der Möglichkeiten zu bekommen:

```

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

Im Allgemeinen, wirst Du y oder n nutzen, wenn Du jeden Ausschnitt stagen willst, aber alle Ausschnitte in bestimmten Dateien zu stagen oder die Entscheidung für einen Ausschnitt auf später zu verschieben kann auch sehr hilfreich sein. Wenn Du nur einen Teil der Datei stagest und den anderen ungestaget lässt, sieht Deine Status-Ausgabe in etwa so aus:

```
What now> 1
      staged      unstaged path
1:    unchanged    +0/-1  TODO
2:      +1/-1      nothing index.html
3:      +1/-1      +4/-0  lib/simplegit.rb
```

Der Status der simplegit.rb ist interessant. Er zeigt Dir, dass ein paar Zeilen gestaget und ein paar ungestaget sind. Du hast diese Datei teilweise gestaget. An dieser Stelle kannst Du das interaktive Hinzufüge-Skript verlassen und `git commit` ausführen, um die teilweise gestageten Dateien zu commiten.

Letztendlich musst Du nicht den interaktiven Hinzufüge-Modus nutzen, um Dateien teilweise zu stagen – Du kannst das gleiche Skript starten, indem Du `git add -p` oder `git add --patch` auf der Kommandozeile eingibst.

Stashen

Während man an einer bestimmten Funktion eines Projekts arbeitet, ist es oft so, dass man den Branch wechseln will, weil man an etwas anderem weiterarbeiten will. Meist ist dann auch das Arbeitsverzeichnis in einem chaotischen Zustand, da die Funktion noch nicht fertiggestellt ist. Das Problem dabei ist, dass Du Deine halbfertige Arbeit dann auch nicht committen möchtest, um später daran weiter arbeiten zu können. Die Lösung dieses Problems bietet der `git stash` Befehl.

Beim Stashen werden die aus Deinem Arbeitsverzeichnis noch nicht committeten Änderungen – also Deine geänderten beobachteten Dateien und die in der Staging-Area enthaltenen Dateien – in einem Stack voller unfertiger Änderungen gespeichert. Diese kannst Du dann jederzeit wieder vom Stack holen und auf Dein Arbeitsverzeichnis anwenden.

Stash verwenden

Um dies zu demonstrieren, gehst Du in Dein Projekt und beginnst an ein paar Dateien zu arbeiten und merkst ein paar dieser Änderungen in der Staging-Area vor. Wenn Du den Befehl `git status` ausführst, siehst Du, dass sich einige Dateien seit dem letzten Commit verändert haben.

```
$ git status
# 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)
#
#       modified:   lib/simplegit.rb
#
```

Jetzt kommt der Zeitpunkt, an dem Du den aktuellen Branch wechseln möchtest. Allerdings willst Du den aktuellen Zustand auch nicht committen, da Deine Arbeit noch nicht ganz fertiggestellt ist. Darum legst Du Deine Änderungen jetzt in einem Stash ab. Um diesen neuen Stash auf dem Stack abzulegen, verwendest Du den Befehl `git stash`:

```
$ 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")
```

In Deinem Arbeitsverzeichnis befinden sich jetzt keine geänderten Dateien mehr und die Staging-Area ist auch leer:

```
$ git status
# On branch master
nothing to commit (working directory clean)
```

In diesem Zustand, kannst Du in beliebig andere Branches wechseln und an etwas anderem weiterarbeiten. Deine Änderungen sind alle in einem Stack gesichert. Um eine Übersicht, der bereits gestashten Änderungen anzusehen, kannst Du den Befehl `git stash list` verwenden:

```
$ 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 diesem Beispiel waren bereits zwei Stashes auf dem Stack vorhanden. Sie wurden zu einem früheren Zeitpunkt gespeichert. Dir stehen jetzt also drei verschiedene Stashes auf dem Stack zur Verfügung. Mit dem Befehl `git stash apply` kannst Du die zuletzt gestashten Änderungen in Deinem Arbeitsverzeichnis wiederherstellen. Git zeigt diesen Befehlsaufruf auch bei Ausführen des Befehls `git stash` als Hilfestellung an. Wenn Du einen der älteren Stashes auf Dein Arbeitsverzeichnis anwenden willst, kannst Du den entsprechenden Stashnamen an den Befehl anhängen: `git stash apply stash@{2}`. Wie Du bereits gesehen hast, verwendet Git die zuletzt gestashten Änderungen und versucht diese im Arbeitsverzeichnis wiederherzustellen, wenn der Stashname nicht angegeben wird:

```
$ git stash apply
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#
#       modified:   index.html
#       modified:   lib/simplegit.rb
#
```

Wie Du sehen kannst, stellt Git die Dateien wieder her, die Du in einem Stash gespeichert hast. In dem Beispiel war Dein Arbeitsverzeichnis in einem sauberen Zustand, als Du versucht hast, den Stash zurückzuladen. Ebenso wurde der Stash auf dem gleichen Branch angewandt, der auch beim Stashen der Änderungen ausgecheckt war. Aber es ist nicht zwingend notwendig, dass der gleiche Branch verwendet wird und dass das Arbeitsverzeichnis in einem sauberen Zustand ist, wenn ein Stash zurückgeladen wird. Du kannst die Änderungen in einem Stash ablegen, zu einem anderen Branch wechseln und die Änderungen in diesem neuen Branch wiederherstellen. Es können auch geänderte oder gestagte Dateien im Arbeitsverzeichnis vorhanden sein, während ein Stash zurückgeladen wird. Können die Änderungen nicht ordnungsgemäß zurückgeladen werden, zeigt Git einen entsprechenden Merge-Konflikt an.

Die Änderungen an den Dateien wurden in Deinem Arbeitsverzeichnis wiederhergestellt. Allerdings ist die Datei, die beim Stashen in der Staging-Area vorhanden war, nicht automatisch wieder in die Staging-Area gewandert. Wenn Du die Option `--index` an den Befehl `git stash apply` anhängst, wird Git versuchen, die Dateien wieder zu stagen. Wenn Du diesen Befehl angewandt hättest, wäre Dein Arbeitsverzeichnis und Deine Staging-Area im exakt gleichen Zustand, wie vor dem Stashen:

```
$ 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)
#
#       modified:   lib/simplegit.rb
#
```

Mit der Option `apply` wird nur versucht die Änderungen wiederherzustellen. Der Stash an sich bleibt weiterhin auf dem Stack vorhanden. Um diesen zu entfernen, kannst Du den Befehl `git stash drop` zusammen mit dem Namen des Stashes anwenden:

```
$ 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)
```

Um den Stash zurückzuführen und gleichzeitig vom Stack zu entfernen, kann der Befehl `git stash pop` verwendet werden.

Zurückgeführten Stash wieder rückgängig machen

Stellen wir uns folgendes Szenario vor: Du wendest die Änderungen eines Stashes wieder auf das Arbeitsverzeichnis an und änderst danach noch ein paar Dateien von Hand. Jetzt möchtest Du die Änderungen, die vom Stash her rühren, aber wieder rückgängig machen. Git besitzt kein Feature, welches dies möglich macht. Allerdings kann man den gleichen Effekt erzeugen, indem man vom betreffenden Stash einen Patch erzeugt und diesen mit der Option `-R` wieder anwendet (Patch rückwärts anwenden).

```
$ git stash show -p stash@{0} | git apply -R
```

An dieser Stelle noch einmal der Hinweis, dass Git den zuletzt erstellten Stash verwendet, wenn kein Stashname angegeben wird:

```
$ git stash show -p | git apply -R
```

Wenn Du dieses Feature öfters benötigst, ist es wahrscheinlich sinnvoll, einen Alias `stash-unapply` in Git dafür anzulegen:

```
$ git config --global alias.stash-unapply '!git stash show -p | git apply -R'
$ git stash
$ #... work work work
$ git stash-unapply
```

Branch auf Basis eines Stashes erzeugen

Wenn Du einen Teil Deiner Arbeit in einem Stash ablegst, dort eine Weile liegen lässt und danach Deine Arbeit an dem Branch fortsetzt, der auch für den Stash verwendet wurde, hast Du vielleicht später Probleme beim Zurückführen des Stashes. Wenn beim Anwenden des Stashes Dateien modifiziert werden sollen, die Du im bisherigen Verlauf bereits geändert hast, werden Merge-Konflikte auftreten, die Du manuell auflösen musst. Wenn Du nach einer einfachen Möglichkeit suchst, die gestashten Änderungen separat zu testen, kannst Du den Befehl `git stash branch` verwenden. Dieser Befehl erzeugt einen neuen Branch, checkt den Commit aus, auf dessen Basis der Stash erstellt wurde, und führt den Stash wieder in das Arbeitsverzeichnis zurück. Wenn dabei kein Fehler auftritt, wird der Stash automatisch gelöscht:

```
$ git stash branch testchanges
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)
#
#       modified:   lib/simplegit.rb
#
Dropped refs/stash@{0} (f0dfc4d5dc332d1cee34a634182e168c4efc3359)
```

Damit ist es auf einfache Art und Weise möglich, die gestashten Änderungen in einem neuen Branch wiederherzustellen und daran weiterzuarbeiten.

Änderungshistorie verändern

Beim Arbeiten mit Git kommt es häufig vor, dass man seine Commit-Historie aus irgendeinem Grund noch einmal ändern möchte. Und das Tolle an Git ist, dass es Dir die Möglichkeit bietet, Entscheidungen erst im allerletzten Moment zu treffen. Zum Beispiel bietet Dir Git mit Hilfe der Staging-Area die Möglichkeit, alle Dateien zu sammeln und kurz vor einem Commit zu entscheiden, welche Daten alle in einen Commit wandern sollen. Du kannst auch Deine Dateien, die sich geändert haben, aber noch nicht ins Repository eingepflegt werden sollen, mit dem Stash-Kommando in einem Zwischenspeicher ablegen. Außerdem kannst Du bereits verfasste Commits nachträglich noch einmal ändern, sodass sich die Historie so ändert, als wäre sie ganz anders vorangeschritten. Das kann man zum Beispiel durch Änderung der Reihenfolge der Commits, durch Ändern von Commit-Nachrichten, durch Modifikationen an Dateien innerhalb eines Commits, durch Zusammenfügen zweier Commits zu einem Commit oder durch Löschen eines Commits erreichen. Und das Besondere daran: Das alles, bevor Du Deine Arbeit mit anderen teilst und veröffentlichst.

In diesem Kapitel werden wir die nützlichen Arbeitsschritte besprechen, die Dir helfen, Deine Commit-Historie Deinen Wünschen entsprechend zu gestalten, sodass Du Dein Ergebnis danach mit anderen teilen kannst und es damit Deinem gewünschten Ergebnis entspricht.

Ändern des letzten Commits

Am häufigsten möchte man wahrscheinlich seinen letzten durchgeführten Commit noch einmal nachträglich ändern. Meist sind es zwei Dinge, die man verändern möchte: Änderung der eingegebenen Commit-Nachricht oder den eigentlichen Inhalt des Schnappschusses durch Hinzufügen, Ändern oder Löschen von Dateien.

Die letzte Commit-Nachricht noch einmal zu ändern ist sehr einfach:

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

Nach Eingabe dieses Befehls wird der Texteditor mit dem Inhalt der letzten Commit-Nachricht geöffnet. Jetzt hat man Gelegenheit diesen Text zu ändern. Nach dem Speichern und Schließen des Editors, wird die Commit-Nachricht des letzten Commits entsprechend angepasst. Der alte Commit ist dadurch nicht mehr vorhanden und Du erhältst einen neuen Commit mit dem gleichen Inhalt und Deiner neuen Commit-Nachricht.

Wenn Du Deine Änderungen bereits eingchecked hast und den Schnappschuss nachträglich durch Hinzufügen oder Ändern von Dateien noch einmal ändern möchtest, läuft das im Prinzip auf die gleiche Art und Weise ab. Meist kommt so etwas vor, weil man vergessen hat, eine neu erstellte Datei zu stagen. Wenn so etwas passiert, kannst Du Folgendes machen: Führe Deine gewünschte Änderungen durch Ändern oder Hinzufügen einer Datei aus und stage dieses Ergebnis mit dem Befehl `git add`. Alternativ kannst Du auch mit dem Befehl `git rm` eine Datei aus dem Repository entfernen. Wenn die Staging-Area Dein gewünschtes Ergebnis enthält, führst Du einfach den Befehl `git commit --amend` aus. Der neue Commit enthält nun die Änderungen aus dem alten

Commit plus die Änderungen aus Deiner Staging-Area.

Mit dem Befehl `--amend` sollte man vorsichtig umgehen, weil sich mit jeder nachträglichen Modifikation eines Commits auch die SHA-1-Prüfsumme ändert. Das Ändern des letzten Commits hat ein ähnliches Verhalten wie das Durchführen eines Rebase-Befehls. Deshalb sollte man einen Commit niemals nachträglich anpassen, wenn dieser bereits veröffentlicht wurde.

Änderung von mehreren Commit-Nachrichten

Um einen Commit, der etwas weiter in der Historie zurückliegt, zu ändern, hilft einem der Befehl `--amend` nicht weiter. Man benötigt dazu ein etwas mächtigeres und komplexeres Werkzeug. Für diese Aufgabe kann man den Rebase-Befehl, den wir bereits kennengelernt haben, auf eine etwas andere Art und Weise nutzen. Anstatt den Rebase auf einen HEAD eines anderen Commits auszuführen, führt man den Rebase auf genau dem gleichen Commit aus, auf dem er bereits basiert. Dazu müssen wir nur den interaktiven Modus des Rebase-Befehls nutzen. Dieser bietet einem die Möglichkeit bei jedem Commit, der geändert werden soll, zu stoppen. Dann kann man seine Änderungen an den Dateien oder an der Commit-Nachricht entsprechend einpflegen und mit dem nächsten Commit fortfahren. Um einen interaktiven Rebase durchzuführen, muss man die Option `-i` an den Befehl `git rebase` anhängen. Außerdem musst Du natürlich bestimmen, wie viele Commits Du ändern möchtest. Dazu musst Du den Commit angeben, auf welchem der Rebase basieren soll.

Wenn Du zum Beispiel die letzten drei, oder eine oder mehrere der letzten drei Commit-Nachrichten ändern möchtest, musst Du zusätzlich zu dem Befehl `git rebase -i` den übergeordneten Commit (also dem Commit, der in der Historie genau ein Commit zurückliegt) des letzten Commits, den Du ändern möchtest, angeben. Bei drei Commit-Nachrichten müsste das Argument also `HEAD~2` beziehungsweise `HEAD~3` lauten. Wahrscheinlich fällt es Dir leichter das Argument `~3` zu merken, weil Du ja schließlich auf die letzten drei Einträge verweisen möchtest. Du solltest Dir aber bewusst sein, dass Du auf den viertältesten Commit verweisen musst, also den übergeordneten Commit, den Du ändern möchtest.

```
$ git rebase -i HEAD~3
```

Es ist wichtig, dass Du Dir bewusst bist, dass mit diesem Rebase-Befehl jeder Commit im Bereich `HEAD~3..HEAD` geändert wird, unabhängig davon, ob Du die Commit-Nachricht beziehungsweise den Schnappschuss änderst oder nicht. Der Rebase-Befehl sollte nie einen Commit beinhalten, der bereits an einen zentralen Server gepusht worden ist. Hältst Du Dich nicht daran, werden sich andere Entwickler über Dich ärgern oder wundern, weil es jetzt eine alternative Version der gleichen Änderung gibt.

Wenn Du den Befehl ausführst, erhältst Du eine Reihe von Commits in Deinem Texteditor. Das könnte in etwa folgendermaßen aussehen:

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

```
# Rebase 710f0f8..a5f4a0d onto 710f0f8
#
# Commands:
# p, pick = use commit
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
#
```

Vielleicht ist es Dir schon aufgefallen, die Commits werden genau in der umgekehrten Reihenfolge dargestellt, wie sie der `log` Befehl ausgegeben hätte. Wenn Du also den Befehl `log` ausführst, erhält man in etwa die folgende Ausgabe:

```
$ 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
```

Siehst Du den Unterschied? Es ist genau die umgekehrte Reihenfolge. Ein interaktiver Rebase wird nach einem festen Schema, einer Art Skript, durchgeführt und der Texteditor zeigt Dir an, wie dieses Skript genau ablaufen wird. Der Rebase startet bei dem Commit, der in der Kommandozeile angegeben wird (`HEAD~3`) und führt die Änderungen, die durch jeden Commit hinzukommen, von oben nach unten aus. Das bedeutet, dass anstatt des neuesten, der älteste Commit ganz oben steht, weil dieser der erste Commit ist, der bearbeitet wird.

Du musst das Skript so anpassen, dass es an jedem Commit anhält, den Du ändern möchtest. Dazu musst Du bei jedem Commit, an dem das Skript anhalten soll, das Wort „pick“ mit dem Wort „edit“ ersetzen. Um zum Beispiel die drittälteste Commit-Nachricht zu ändern, müssen die Änderungen am Skript in etwa folgendermaßen aussehen:

```
edit f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Nachdem Du das Skript gespeichert und den Editor beendet hast, setzt Git nun alle Änderungen bis zum letzten Commit der Liste zurück und zeigt danach in der Kommandozeile in etwa Folgendes an:

```
$ git rebase -i HEAD~3
Stopped at 7482e0d... updated the gemspec to hopefully work better
You can amend the commit now, with
```

```
git commit --amend
```

Once you're satisfied with your changes, run

```
git rebase --continue
```

Diese Anweisungen zeigen Dir sehr genau, was Du zu tun hast. Gib also den folgenden Befehl ein:

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

Im sich öffnenden Texteditor kannst Du jetzt die Commit-Nachricht ändern und danach wieder schließen. Danach führst Du folgenden Befehl aus:

```
$ git rebase --continue
```

Der letzte Befehl speichert die letzten beiden Commits automatisch im Repository und der Rebase ist danach abgeschlossen. Wenn Du in einer weiteren Zeile „pick“ mit „edit“ ersetzt hast, kannst Du die oben dargestellten Schritte entsprechend noch einmal ausführen. Git wird nach jedem Commit anhalten und Dir die Möglichkeit bieten, den Commit anzupassen. Danach kannst Du Git auffordern, den Rebase fortzusetzen (`git rebase --continue`).

Reihenfolge von Commits verändern

Mit einem interaktiven Rebase kannst Du ebenso die Reihenfolge von Commits ändern oder sogar komplette Commits löschen. Um den Commit „added cat-file“ zu löschen und die Reihenfolge der beiden anderen Commits zu ändern, kannst Du das vorhandene Skript

```
pick f7f3f6d changed my name a bit
pick 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

folgendermaßen ändern:

```
pick 310154e updated README formatting and added blame
pick f7f3f6d changed my name a bit
```

Nach dem Speichern und Verlassen des Editors, setzt Git nun alle Änderungen bis zum letzten Commit der Liste zurück, speichert den Commit 310154e, danach den Commit 310154e und beendet danach den Rebase. Das Ergebnis: Der Commit „added cat-file“ ist aus der Historie verschwunden und die Reihenfolge der beiden restlichen Commits ist getauscht.

Mehrere Commits zusammenfassen

Man kann mit einem interaktiven Rebase auch mehrere Commits zu einem einzelnen Commit zusammenfassen. Im Skript der Rebase-Nachricht steht eine Anleitung, wie Du dazu vorgehen musst:

```
#
# Commands:
#   p, pick = use commit
#   e, edit = use commit, but stop for amending
#   s, squash = use commit, but meld into previous commit
#
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will be aborted.
```

#

Wenn Du statt „pick“ oder „edit“, den Befehl „squash“ angibst, führt Git beide Commits zu einem gemeinsamen Commit zusammen und bietet Dir die Möglichkeit, die Commit-Nachricht ebenso entsprechend zu verheiraten. Wenn Du also aus den drei Commits einen einzelnen Commit machen willst, muss Dein Skript folgendermaßen aufgebaut sein:

```
pick f7f3f6d changed my name a bit
squash 310154e updated README formatting and added blame
squash a5f4a0d added cat-file
```

Nach dem Speichern und Beenden des Editors, führt Git alle drei Änderungen zu einem einzelnen Commit zusammen und öffnet einen Texteditor, der alle drei Commit-Nachrichten enthält:

```
# 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
```

Du kannst nun die Commit-Nachricht entsprechend anpassen oder auch entsprechend dem vorgeschlagenen Ergebnis belassen. Wenn Du die Commit-Nachricht speicherst, hast Du danach nur noch einen einzelnen Commit, der die letzten drei Commits beinhaltet, in Deiner Historie.

Aufsplitten eines einzelnen Commits

Man kann mit Git einen einzelnen Commit auch aufsplitten. Das bedeutet, man setzt den ursprünglichen Commit zurück, fügt dann einen Teil der Änderungen zur Staging-Area hinzu und checkt das Ergebnis ein. Dies kann man unbegrenzt oft wiederholen und so einen einzelnen Commit in mehrere Commits aufteilen. Nehmen wir an, wir möchten den mittleren der beiden Commits aufteilen. Anstatt „updated README formatting and added blame“, möchten wir den Commit in folgende beiden Commits aufteilen: „updated README formatting“ soll das Thema des ersten Commits und „added blame“ soll das Thema des zweiten Commits sein. Dazu kannst Du das angezeigte Skript, welches Dir der Befehl `rebase -i` erzeugt, folgendermaßen anpassen:

```
pick f7f3f6d changed my name a bit
edit 310154e updated README formatting and added blame
pick a5f4a0d added cat-file
```

Nach dem Speichern und Schließen des Editors, setzt Git die Änderungen entsprechend zurück und wendet den ersten (f7f3f6d) und zweiten (310154e) Commit an und wechselt danach zurück zur Kommandozeile. Jetzt hast Du die Möglichkeit den letzten Commit zurückzusetzen, ohne dass die Änderungen im Arbeitsverzeichnis zurückgesetzt werden. Das heißt, der Commit im

Repository wird gelöscht, aber Deine Änderungen im Arbeitsverzeichnis gehen nicht verloren. Um dies durchzuführen, kannst Du den Befehl `git reset HEAD^` verwenden. Jetzt kannst Du die gewünschten Änderungen für den ersten Commit zur Staging-Area hinzufügen und danach einchecken. Diesen Vorgang kannst Du beliebig wiederholen, bis alle Änderungen eingchecked sind. Wenn Du fertig bist, kannst Du den Rebase mit `git rebase --continue` fortsetzen beziehungsweise abschließen:

```
$ 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 speichert dazu den letzten Commit (a5f4a0d) aus dem Skript im Repository. Das Resultat sieht in etwa folgendermaßen aus:

```
$ git log -4 --pretty=format:"%h %s"
1c002dd added cat-file
9b29157 added blame
35cfb2b updated README formatting
f3cc40e changed my name a bit
```

Ich möchte Dich noch einmal darauf hinweisen, dass jede SHA-Prüfsumme von allen Commits aus der Liste geändert werden. Bitte stell also sicher, dass diese Commits in keinem öffentlichen Repository verfügbar sind.

Hol den Vorschlaghammer raus: filter-branch

Es gibt noch eine weitere Möglichkeit, wie man die Historie nach seinen Wünschen anpassen kann. Diese wird oft angewandt, wenn man eine große Zahl von Commits automatisiert mit Hilfe eines Skripts anpassen will. Zum Beispiel kann man damit die E-Mail-Adresse in jedem Commit ändern oder auch eine Datei aus jedem Commit entfernen. Das Werkzeug dazu heißt `filter-branch`. Damit kann man einen riesigen Teil der Historie ändern. Man sollte diesen Befehl also nur verwenden, wenn das Projekt noch nicht weit verbreitet ist, oder andere Personen noch nicht damit begonnen haben an dem Projekt zu arbeiten (also auf Basis der bisherigen Historie neue Branches mit Commits erstellt wurden). Trotzdem kann dieses Werkzeug sehr nützlich sein. Ich möchte hier ein paar der Möglichkeiten dieses Werkzeugs vorstellen.

Löschen einer Datei aus jedem Commit

Dieses Szenario tritt sogar relativ häufig auf. Nehmen wir einmal an, jemand fügt gedankenlos eine große binäre Datei mit `git add .` zum Repository dazu und diese soll aber in keinem der Commits enthalten sein. Oder Du hast aus Versehen eine Datei, welche ein Passwort enthält, zum Repository hinzugefügt und möchtest dieses Repository nun veröffentlichen. `filter-branch` ist dann das Werkzeug Deiner Wahl, um die komplette Historie umzukrempeln. Um eine Datei mit dem Namen „passwords.txt“ aus der kompletten Historie zu löschen, kannst Du die Option - -

tree-filter verwenden:

```
$ git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
Rewrite 6b9b3cf04e7c5686a9cb838c3f36a8cb6a0fc2bd (21/21)
Ref 'refs/heads/master' was rewritten
```

Die Option `--tree-filter` führt den nachfolgenden Befehl nach jedem Auschecken eines Commits des Projekts aus und checkt danach das Ergebnis wieder ein. In diesem Beispiel wird die Datei „passwords.txt“ aus jedem Schnappschuss entfernt, unabhängig davon, ob sie existiert oder nicht. Ein anderes Beispiel wäre es, alle Backup-Dateien eines Texteditors aus dem Repository zu löschen. Dazu kann man in etwa den Befehl `git filter-branch -\-tree-filter "rm -f *~"` HEAD ausführen.

Git informiert Dich über den Fortschritt dieses Vorgangs und Du siehst, wie jeder Commit angepasst wird und der Zeiger auf den Branch auf den letzten Commit gesetzt wird. Es ist empfehlenswert, diesen Befehl in einem Testzweig durchzuführen. Wenn das Ergebnis, wie gewünscht ausfällt, kann man danach den Branch master auf diesen Testzweig setzen. Wenn man an den Befehl `filter-branch` die Option `--all` anfügt, führt Git diesen Vorgang für jeden vorhandenen Zweig aus.

Aus einem Unterverzeichnis das neue Wurzelverzeichnis machen

Wenn man zum Beispiel ein Projekt aus einem anderen Versionskontrollwerkzeug in Git importiert, gibt es dort oft Verzeichnisse, die in Git nicht relevant sind, zum Beispiel `trunk`, `tags`, usw.. Wenn man also das Unterverzeichnis `trunk` das neue Wurzelverzeichnis machen will, kann man dies mit Hilfe von `filter-branch` umsetzen:

```
$ git filter-branch --subdirectory-filter trunk HEAD
Rewrite 856f0bf61e41a27326cdae8f09fe708d679f596f (12/12)
Ref 'refs/heads/master' was rewritten
```

Nach der Ausführung dieses Befehls ist das „trunk“ Verzeichnis das neue Arbeitsverzeichnis. Bei diesem Vorgang entfernt Git außerdem alle Commits, die nicht eine Änderung des „trunk“-Verzeichnisses beinhalten.

E-Mail Adresse in jedem Commit ändern

Verflixt, es ist schon wieder passiert. Du hast vergessen, den Befehl `git config` auszuführen und Deinen Namen und E-Mail-Adresse zu setzen, bevor Du mit der Arbeit begonnen hast. Mit `filter-branch` kann man diesen Fehler einfach beheben. Man sollte nur darauf achten, dass man nur seine eigene E-Mail-Adresse ändert. Deshalb verwenden wir die Option `--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
```

Dieser Befehl durchforstet das Repository und ersetzt in jedem Commit, dessen E-Mail-Adresse des Autors „schacon@localhost“ lautet, mit der neuen E-Mail-Adresse „schacon@example.com“. Zusätzlich wird der Name des Autors geändert, falls dieser nicht vorher schon „Scott Chacon“ war. Auf Grund der Architektur, dass in Git in jedem Commit die SHA1-Prüfsumme des Vorgänger-Commits enthalten ist, ändert dieser Befehl jeden Commit in Deiner Historie. Die SHA1-Prüfsumme wird sich auch in allen Commits, die nicht die angegebene E-Mail-Adresse enthalten, verändern.

Mit Hilfe von Git debuggen

Git bietet auch ein paar Werkzeuge, die den Debug-Vorgang bei einem Projekt unterstützen. Da Git so aufgebaut ist, dass es für nahezu jedes Projekt eingesetzt werden kann, sind diese Werkzeuge sehr generisch gehalten. Wenn gewisse Dinge schief laufen, können Dir aber diese Tools oft helfen, den Bug oder den Übeltäter zu finden.

Datei Annotation

Wenn Du nach einem Bug in Deinem Code suchst und gerne wissen willst, wann und warum dieser zum ersten Mal auftrat, dann kann Dir das Werkzeug Datei-Annotation (engl. File Annotation) sicher weiterhelfen. Es kann Dir anzeigen, in welchem Commit die jeweilige Zeile einer Datei zuletzt geändert wurde. Wenn Du also feststellst, dass eine Methode beziehungsweise eine Funktion in Deinem Code nicht mehr das gewünschte Resultat liefert, kannst Du Dir die Datei mit `git blame` genauer ansehen. Nach Aufruf des Befehls zeigt Git Dir an, welche Zeile von welcher Person als letztes geändert wurde, inklusive Datum. Das folgende Beispiel verwendet die Option `-L`, um die Ausgabe auf die Zeilen 12 bis 22 einzuschränken:

```
$ git blame -L 12,22 simplegit.rb
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 12) def show(tree = 'master')
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 13)   command("git show #{tree}")
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 14) end
^4832fe2 (Scott Chacon 2008-03-15 10:31:28 -0700 15)
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 16) def log(tree = 'master')
79eaf55d (Scott Chacon 2008-04-06 10:15:08 -0700 17)   command("git log #{tree}")
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 18) end
9f6560e4 (Scott Chacon 2008-03-17 21:52:20 -0700 19)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 20) def blame(path)
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 21)   command("git blame #{path}")
42cf2861 (Magnus Chacon 2008-04-13 10:45:01 -0700 22) end
```

In der ersten Spalte wird die Kurzform der SHA1-Prüfsumme des Commits angezeigt, in welchem diese Zeile zuletzt verändert wurde. Die nächsten beiden Spalten weisen auf den Autor des Commits und wann dieser verfasst wurde, hin. Auf diese Weise kannst Du leicht bestimmen, wer die jeweilige Zeile geändert hat und wann dies durchgeführt wurde. In den nächsten Spalten wird die Zeilennummer und der Inhalt der Zeile angezeigt. Die Zeilen mit der SHA-1-Prüfsumme `^4832fe2` weisen darauf hin, dass diese bereits im ersten Commit vorhanden waren. Das ist also der Commit, in dem die Datei „simplegit.rb“ zum Repository hinzugefügt wurde und die Zeilen deuten damit darauf hin, dass diese bisher nie geändert wurden. Das ist für Dich wahrscheinlich ein bisschen verwirrend, denn nun kennst Du bereits drei Möglichkeiten, wie Git mit dem Zeichen `^` einer SHA-Prüfsumme eine neue Bedeutung gibt. Aber in Zusammenhang mit `git blame` weist das Zeichen auf den eben geschilderten Sachverhalt hin.

Eine weitere herausragende Eigenschaft von Git ist die Tatsache, dass es nicht per se das Umbenennen von Dateien verfolgt. Git speichert immer den jeweiligen Schnappschuss des Dateisystems und versucht erst danach zu bestimmen, welche Dateien umbenannt wurden. Das bietet Dir zum Beispiel die Möglichkeit herauszufinden, wie Code innerhalb des Repositories hin

und her verschoben wurde. Wenn Du also die Option `-C` an `git blame` anfügst, analysiert Git die angegebene Datei und versucht herauszufinden, ob und von wo bestimmte Codezeilen herkopiert wurden. Vor kurzem habe ich ein Refactoring an einer Datei mit dem Namen `GITServerHandler.m` durchgeführt. Dabei habe ich diese Datei in mehrere Dateien aufgeteilt, eine davon war `GITPackUpload.m`. Wenn ich jetzt `git blame` mit der Option `-C` auf die Datei `GITPackUpload.m` ausführe, erhalte ich eine Ausgabe mit den Codezeilen, von denen das Ergebnis ursprünglich stammt:

```
$ 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) gatherObjectShasFrom
f344f58d GITServerHandler.m (Scott 2009-01-04 143) {
70befddd GITServerHandler.m (Scott 2009-03-22 144) //NSLog(@"GATHER COMM:
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 = [
ad11ac80 GITPackUpload.m (Scott 2009-03-24 148)
ad11ac80 GITPackUpload.m (Scott 2009-03-24 149) //NSLog(@"GATHER COMM:
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 setOl
56ef2caf GITServerHandler.m (Scott 2009-01-05 153)
```

Das ist enorm hilfreich. Für gewöhnlich erhältst Du damit als ursprünglichen Commit, den Commit, von welchem der Code kopiert wurde, da dies der Zeitpunkt war, bei dem diese Zeilen zum ersten Mal angefasst wurden. Git zeigt Dir den ursprünglichen Commit, in dem Du die Zeilen verfasst hast, sogar an, wenn es sich dabei um eine andere Datei handelt.

Das Bisect Werkzeug – Binäre Suche

`git blame` kann Dir sehr weiterhelfen, wenn Du bereits weißt, an welcher Stelle das Problem liegt. Wenn Du aber nicht weißt, warum gewisse Dinge schief laufen, und es gibt inzwischen Dutzende oder Hunderte von Commits seit dem letzten funktionierenden Stand, dann solltest Du `git bisect` als Hilfestellung verwenden. Der `bisect` Befehl führt eine binäre Suche durch die Commit-Historie durch und hilft Dir auf schnelle Art und Weise die Commits zu bestimmen, die eventuell für das Problem verantwortlich sind.

Nehmen wir zum Beispiel an, dass Du gerade eben Deinen Code in einer Produktivumgebung veröffentlicht hast und auf einmal bekommst Du zahlreiche Fehlerberichte über Probleme, die in Deiner Entwicklungsumgebung nicht aufgetreten sind. Du kannst Dir auch keinen Reim darauf bilden, warum der Code so reagiert. Nachdem Du Dich noch einmal näher mit Deinem Code beschäftigt hast, stellst Du fest, dass Du die Fehlerwirkung reproduzieren kannst, aber Dir ist es immer noch ein Rätsel, was genau schief läuft. Wenn Du vor einem solchen Problem stehst, hilft Dir es bestimmt, wenn Du die Historie in mehrere Teile aufspaltest (engl. `bisect`: halbieren, zweiteilen). Als erstes startest Du mit dem Befehl `git bisect start`. Danach gibst Du mit dem Befehl `git bisect bad` an, dass der derzeit ausgecheckte Commit den Fehler aufweist. Jetzt braucht Git noch die Information, in welchem Commit das Problem noch nicht aufgetreten ist. Dazu verwendest Du den Befehl `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
```

Nach Ausführen des letzten Befehls, zeigt Git Dir als Erstes an, dass in etwa 12 Commits zwischen der letzten guten Revision (v1.0) und der aktuellen, fehlerhaften Revision liegen. Auf Basis dieser Information hat Git Dir den mittleren Commit ausgecheckt. Jetzt hast Du die Möglichkeit Deine Tests auf Basis des ausgecheckten Stands durchzuführen, um herauszufinden, ob in diesem Commit der Fehler bereits bestand. Wenn der Fehler hier bereits auftritt, dann wurde er in diesem oder in einem der früheren Commits eingefügt. Wenn der Fehler hier noch nicht auftritt, dann wurde er in einem der späteren Commits eingeschleppt. In unserem Beispiel nehmen wir an, dass in diesem Commit der Fehler noch nicht bestand. Das geben wir mit dem Befehl `git bisect good` an und fahren fort:

```
$ git bisect good
Bisecting: 3 revisions left to test after this
[b047b02ea83310a70fd603dc8cd7a6cd13d15c04] secure this thing
```

Git hat Dir jetzt einen weiteren Commit ausgecheckt, und zwar wieder den mittleren Commit zwischen dem letzten Stand im Repository und dem mittleren Commit aus der letzten Runde. Hier nehmen wir an, dass Du nach Deinen durchgeführten Tests feststellst, dass in diesem Commit der Fehler bereits vorhanden ist. Das müssen wir Git über `git bisect bad` mitteilen:

```
$ git bisect bad
Bisecting: 1 revisions left to test after this
[f71ce38690acf49c1f3c9bea38e09d82a5ce6014] drop exceptions table
```

Git checkt wieder den nächsten mittleren Commit aus und wir stellen fest, dass dieser in Ordnung ist. Ab jetzt hat Git alle notwendigen Informationen um festzustellen, in welchem Commit der Fehler eingebaut wurde. Git zeigt Dir dazu die SHA-1-Prüfsumme des ersten fehlerhaften Commits an. Zusätzlich gibt es noch weitere Commit-Informationen und welche Dateien in diesem Commit geändert wurden, an. Das sollte Dir nun helfen, den Fehler näher zu bestimmen:

```
$ 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
f24d3c6ebcf6c639b1a3814550e62d60b8e68a8e4 M  config
```

Wenn Du fertig mit der Fehlersuche bist, solltest Du den Befehl `git bisect reset` ausführen. Dies checkt den ursprünglichen Stand aus, den Du ausgecheckt hattest, bevor Du mit der Fehlersuche begonnen hast:

```
$ git bisect reset
```

Wie Du vielleicht gesehen hast, ist dieser Befehl ein mächtiges Werkzeug, um Hunderte von Commits auf schnelle Art und Weise nach einem bestimmten Fehler zu durchsuchen. Besonders nützlich ist es, wenn Du ein Skript hast, welches mit dem Fehlercode Null beendet, wenn das Projekt in Ordnung ist und mit einem Fehlercode größer Null, wenn das Projekt Fehler enthält. Wenn Dir ein solches Skript zur Verfügung steht, kannst Du den bisher manuell durchgeführten Vorgang auch automatisieren. Wie im vorigen Beispiel musst Du Git den zuletzt fehlerfreien Commit und den fehlerhaften Commit angeben. Als verkürzte Schreibweise kannst Du an den Befehl `bisect start` den fehlerhaften und den fehlerfreien Commit angeben:

```
$ git bisect start HEAD v1.0  
$ git bisect run test-error.sh
```

Wenn Du die untere, genannte Zeile ausführst, führt Git automatisch nach jedem Auscheckvorgang das Skript `test-error.sh` aus und zwar solange bis es den Commit findet, der als erstes ein fehlerhaftes Ergebnis liefert. Statt eines Skripts kannst Du natürlich auch `make` oder `make tests` oder eine beliebige, andere Testumgebung starten.

Submodule

Oft möchte man während der Arbeit an einem Projekt ein weiteres Projekt darin einbinden und verwenden. Das kann zum Beispiel eine Bibliothek von einer anderen Firma oder vielleicht auch eine selbstentwickelte Bibliothek sein. In einem solchen Szenario tritt dann meistens folgendes Problem auf: Die zwei Projekte sollen unabhängig voneinander entwickelt werden können, aber trotzdem soll es möglich sein, dass eine Projekt im anderen zu verwenden.

Dazu ein Beispiel. Nehmen wir an, Du entwickelst gerade eine Webseite, die unter anderem einen Atom-Feed zur Verfügung stellen soll. Anstatt den notwendigen Code zur Auslieferung des Atom-Feeds selber zu schreiben, entscheidest Du Dich für eine geeignete Bibliothek. Dann wirst Du wahrscheinlich den Code in Dein Projekt einbinden müssen, zum Beispiel durch eine CPAN-Installation oder ein Ruby-Gem oder durch Kopieren des Quellcodes in das Arbeitsverzeichnis Deines Projekts. Das Problem beim Einbinden einer Bibliothek ist, dass es schwierig ist, diese an die eigene Bedürfnisse anzupassen. Noch schwieriger gestaltet sich dann die Veröffentlichung des Projekts, da man sicherstellen muss, dass jeder der die Software verwendet, auf die Bibliothek zugreifen kann. Wenn man die Bibliothek im eigenen Projekt projektspezifisch anpasst, hat man meist ein Problem, wenn man eine neue Version der Bibliothek einspielen will.

Git löst dieses Problem mit Hilfe von Submodulen. Mit Hilfe von Submodulen kann man innerhalb eines Git-Repositorys ein weiteres Git-Repository in einem Unterverzeichnis verwalten. Daraus entsteht der Vorteil, dass man ein anderes Repository in das eigene Projekt klonen kann und die Commits der jeweiligen Projekte trennen kann.

Die ersten Schritte mit Submodulen

Nehmen wir einmal an, dass Du die Rack-Bibliothek (eine Ruby-Gateway-Schnittstelle für Webserver) zu Deinem Projekt hinzufügen willst. Dabei möchtest Du Deine eigenen Änderungen an dieser Bibliothek nachverfolgen, aber auch weiterhin Änderungen von den Rack-Bibliothek-Entwicklern verwalten und zusammenmergen. Das erste was Du dazu tun musst, ist das Repository der Rack Bibliothek in ein Unterverzeichnis Deines Projekts zu klonen. Diesen Vorgang kannst Du mit Hilfe des Befehls `git submodule add` ausführen:

```
$ git submodule add git://github.com/chneukirchen/rack.git rack
Initialized empty Git repository in /opt/subtest/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 422 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
```

Innerhalb Deines Projekts befindet sich nun im Unterverzeichnis `rack` das komplette Rack-Projekt. Man kann jetzt in diesem Verzeichnis Änderungen vornehmen und ein eigenes Remote-Repository mit Schreibrechten, zu welchem man pushen kann, hinzufügen. Ebenso ist es möglich, Änderungen von den Rack-Entwicklern in sein Repository zu laden und diese mit den eigenen Ergebnissen zu mergen. Im Prinzip kann man innerhalb eines Submoduls die gleichen Vorgänge,

wie in einem normalen Repository ausführen. Vorher müssen wir aber noch ein paar weitere Dinge zu Submodulen besprechen. Wenn Du gleich nach dem Hinzufügen des Submoduls, den Befehl `git status` ausführst, wirst Du gleich zwei Dinge bemerken:

```
$ git status
# On branch master
# Changes to be committed:
#   (use "git reset HEAD <file>..." to unstage)
#
#       new file:   .gitmodules
#       new file:   rack
#
```

Als erstes wird Dir die neue Datei `.gitmodules` auffallen. Das ist eine Konfigurationsdatei, welche die Zuordnung der URL des geklonten Projekts und dem lokalen Unterverzeichnis, in welches das Projekt geklont wurde, festlegt:

```
$ cat .gitmodules
[submodule "rack"]
    path = rack
    url = git://github.com/chneukirchen/rack.git
```

Wenn Du mehrere Submodule in einem Projekt verwaltest, werden auch mehrere Einträge in dieser Datei auftauchen. Dabei ist es wichtig zu wissen, dass diese Datei zusammen mit all den anderen Dateien aus Deinem Projekt, ebenso in die Versionskontrolle aufgenommen wird, ähnlich wie die Datei `.gitignore`. Die Datei wird so wie der Rest Deines Projekts gepusht und gepullt. Dadurch wissen andere Personen, die Dein Projekt klonen, von welchem Ort sie die Submodule erhalten können.

Die zweite Auffälligkeit bei der Ausgabe von `git status` ist der Eintrag `rack`. Wenn Du auf diesen Eintrag ein `git diff` durchführst, erhält man in etwa die folgende, interessante Ausgabe:

```
$ git diff --cached rack
diff --git a/rack b/rack
new file mode 160000
index 00000000..08d709f
--- /dev/null
+++ b/rack
@@ -0,0 +1 @@
+Subproject commit 08d709f78b8c5b0fbef7821e37fa53e69afcf433
```

Obwohl `rack` ein Unterverzeichnis in Deinem Arbeitsverzeichnis ist, erkennt Git dieses Verzeichnis als Submodul und verfolgt die Änderungen innerhalb dieses Verzeichnisses nicht, wenn Git nicht innerhalb dieses Verzeichnisses aufgerufen wird. Stattdessen erfasst Git, welcher Commit in diesem Repository ausgecheckt ist. Wenn Du also Änderungen in diesem Unterverzeichnis durchführst und eincheckst, kann das Superprojekt erkennen, dass sich der aktuelle HEAD von diesem Projekt geändert hat. Das Superprojekt kann sich jetzt diesen Commit merken. Auf diese Art und Weise ist es möglich, den kompletten Zustand des Projekts mit allen Projekten, die als Submodul hinzugefügt wurden, zu reproduzieren. In der Git-Terminologie wird

das Projekt, welches eines oder mehrere Submodule enthält, als Superprojekt bezeichnet.

Dabei muss man sich folgender Eigenschaft bewusst sein: Git verwaltet den exakten Commit, der gerade ausgecheckt ist und nicht etwa den Branch oder eine andere Referenz. Git kann also zum Beispiel nicht speichern, dass der aktuelle Stand im Branch master enthalten ist.

Wenn Du Dein Projekt das erste Mal eincheckst, erhältst Du in etwa folgende Ausgabe:

```
$ git commit -m 'first commit with submodule rack'
[master 0550271] first commit with submodule rack
2 files changed, 4 insertions(+), 0 deletions(-)
create mode 100644 .gitmodules
create mode 160000 rack
```

Der Mode 160000, der für den rack-Eintrag gilt, ist ein spezieller Mode in Git. Er bedeutet in etwa, dass man einen Commit als Verzeichnis-Eintrag in Git verwaltet und damit nicht wie normalerweise ein Verzeichnis oder eine Datei.

Das Verzeichnis rack kann man wie ein separates Projekt behandeln und verwenden. Und von Zeit zu Zeit aktualisiert man auch das Superprojekt und speichert damit die letzte Commit-ID des Unterprojekts. Jedes Git-Kommando arbeitet unabhängig in einem der beiden Unterverzeichnisse:

```
$ git log -1
commit 0550271328a0038865aad6331e620cd7238601bb
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:03:56 2009 -0700

    first commit with submodule rack

$ cd rack/
$ git log -1
commit 08d709f78b8c5b0fbbeb7821e37fa53e69afcf433
Author: Christian Neukirchen <chneukirchen@gmail.com>
Date: Wed Mar 25 14:49:04 2009 +0100
```

Document version change

Klonen eines Projekts mit den dazugehörigen Submodulen

Als nächstes klonen wir ein Projekt, welches ein Submodul verwendet. Wenn man ein solches Projekt klonet, werden die entsprechenden Verzeichnisse, welche ein Submodul enthalten, erstellt. Allerdings enthalten diese Verzeichnisse noch keinen Inhalt:

```
$ git clone git://github.com/schacon/myproject.git
Initialized empty Git repository in /opt/myproject/.git/
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 6 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (6/6), done.
$ cd myproject
$ ls -l
```

```
total 8
-rw-r--r--  1 schacon  admin   3 Apr  9 09:11 README
drwxr-xr-x  2 schacon  admin  68 Apr  9 09:11 rack
$ ls rack/
$
```

Das Verzeichnis `rack` wurde zwar erzeugt, aber es ist leer. Deshalb musst Du zwei Dinge ausführen: `git submodule init`, damit wird die Datei für die lokale Konfiguration initialisiert. Und `git submodule update`, welches die gesamten Daten des Projekts von der im `.gitmodules` angegebenen Quelle holt und den entsprechenden Commit, welcher im Superprojekt hinterlegt ist, auscheckt:

```
$ git submodule init
Submodule 'rack' (git://github.com/chneukirchen/rack.git) registered for path 'rack'
$ git submodule update
Initialized empty Git repository in /opt/myproject/rack/.git/
remote: Counting objects: 3181, done.
remote: Compressing objects: 100% (1534/1534), done.
remote: Total 3181 (delta 1951), reused 2623 (delta 1603)
Receiving objects: 100% (3181/3181), 675.42 KiB | 173 KiB/s, done.
Resolving deltas: 100% (1951/1951), done.
Submodule path 'rack': checked out '08d709f78b8c5b0fbef7821e37fa53e69afcf433'
```

Nach Ausführung der beiden Befehle befindet sich das Verzeichnis `rack` in genau dem gleichen Zustand, wie wir es ursprünglich eingchecked haben. Wenn ein anderer Entwickler Änderungen am Rack-Code durchführt, diese eincheckt und Du diese dann pullst und mergst, erhält man einen etwas seltsamen Zustand:

```
$ git merge origin/master
Updating 0550271..85a3eee
Fast forward
 rack |      2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)
[master*]$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   rack
#
```

Der Merge, der gerade durchgeführt worden ist, ist eigentlich nur eine Aktualisierung des Zeigers auf einen neuen Commit des Submoduls. Der eigentliche Inhalt des Submodul-Verzeichnis wurde allerdings nicht aktualisiert. Das sieht dann so aus, als gäbe es noch nicht eingchecked Dateien innerhalb Deines Arbeitsverzeichnis:

```
$ git diff
diff --git a/rack b/rack
index 6c5e70b..08d709f 160000
--- a/rack
```



```
+++ b/rack
@@ -1 +1 @@
-Subproject commit 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
+Subproject commit 08d709f78b8c5b0fbef7821e37fa53e69afcf433
```

Dieser Zustand tritt auf, weil der Zeiger auf den Commit im Submodul derzeit nicht der Commit ist, welcher im Submodul ausgecheckt ist. Um dies zu beheben, muss man den Befehl `git submodule update` erneut ausführen:

```
$ git submodule update
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 2 (delta 0)
Unpacking objects: 100% (3/3), done.
From git@github.com:schacon/rack
    08d709f..6c5e70b  master    -> origin/master
Submodule path 'rack': checked out '6c5e70b984a60b3cecd395edd5b48a7575bf58e0'
```

Dieser Update muss jedes Mal ausgeführt werden, wenn man das Superprojekt pullt und dort eine Änderung in einem Submodul enthalten ist. Es ist vielleicht ein wenig merkwürdig, aber es funktioniert.

Häufig tritt beim Arbeiten mit Submodulen ein Problem bei folgendem Szenario auf: Ein Entwickler führt Änderungen in einem Submodul durch, checkt diese ein, vergisst aber diese Änderungen zum zentralen Server zu pushen. Wenn dann im Superprojekt die Änderung des Submoduls ebenso eingechekkt wird und dieses dann gepusht wird, tritt ein Problem auf. Wenn jetzt andere Entwickler den neuen Stand des Superprojekts holen und den Befehl `git submodule update` ausführen, erhalten sie eine Fehlermeldung, dass der entsprechend referenzierte Commit von dem Submodul nicht gefunden werden konnte. Das passiert weil dieser Commit bei dem zweiten Entwickler noch gar nicht existiert. Wenn ein solcher Fall auftritt, erhält man in etwa folgende Fehlermeldung:

```
$ git submodule update
fatal: reference isn't a tree: 6c5e70b984a60b3cecd395edd5b48a7575bf58e0
Unable to checkout '6c5e70b984a60b3cecd395edd5ba7575bf58e0' in submodule path 'r'
```

Dann kann man allerdings herausfinden, wer zum letzten Mal eine Änderung eingechekkt hat:

```
$ git log -1 rack
commit 85a3eee996800fcfa91e2119372dd4172bf76678
Author: Scott Chacon <schacon@gmail.com>
Date: Thu Apr 9 09:19:14 2009 -0700
```

added a submodule reference I will never make public. hahahahaha!

Dann kannst Du diesem Entwickler eine E-Mail schreiben und ihn auf seinen Fehler aufmerksam machen.

Superprojekte

In manchen großen Projekten möchten die Entwickler die Arbeit in verschiedenen Verzeichnissen aufteilen, sodass das jeweilige Team in diesen Verzeichnissen arbeiten kann. Man trifft diese Vorgehensweise häufig an, wenn ein Team gerade von CVS oder Subversion nach Git gewechselt hat, im alten System ein Modul oder eine Sammlung von solchen Unterverzeichnissen gebildet hat und diesen Arbeitsablauf weiterhin verwenden möchte.

In Git kann man diese Vorgehensweise gut abbilden, indem man für jedes Unterverzeichnis ein neues Git-Repository erzeugt. Zusätzlich kann man dann ein Superprojekt erzeugen und die ganzen Git-Repositorys als Submodul hinzufügen. Ein Vorteil dabei ist, dass man mit Hilfe von Tags und Branches im Superprojekt das Verhältnis der einzelnen Submodule zueinander festhalten kann.

Häufige Probleme mit Submodulen

Die Arbeit mit Submodulen verläuft jedoch nicht immer reibungslos. Man muss verhältnismäßig gut aufpassen, wenn man in einem Submodul-Verzeichnis arbeitet. Wenn man nämlich den Befehl `git submodule update` ausführt, checkt Git den entsprechenden Zustand des Commits aus, aber checkt dabei keinen Branch aus. Diesen Zustand nennt man auch `detached HEAD`. Das bedeutet, dass die Datei `HEAD` direkt auf einen Commit zeigt und nicht, wie sonst üblich, auf eine symbolische Referenz, also zum Beispiel auf einen Branch. Das Problem dabei ist, dass man normalerweise in einem solchen Zustand nicht weiterarbeiten möchte, weil es sehr leicht vorkommen kann, dass Änderungen verloren gehen. Wenn man also den Befehl `git submodule update` aufruft, dann einen Commit in dem entsprechenden Submodul-Verzeichnis ausführt, ohne davor einen Branch auszuchecken, und dann noch einmal `git submodule update` im Superprojekt aufruft, ohne dass man die Änderungen im Submodul im Superprojekt eingingecheckt hat, verliert man die ganzen Änderungen, ohne dass Git einen darauf vorher hinweist. Tatsächlich ist es so, dass die Änderungen nicht verloren gehen, aber es gibt keinen Branch, der auf die entsprechenden Commits hinzeigt, und damit kann es schwierig werden, die entsprechenden Commits wiederherzustellen beziehungsweise sichtbar zu machen.

Um dieses Problem zu vermeiden, sollte man also immer in dem Submodul-Verzeichnis einen neuen Branch mit `git checkout -b work` oder auf eine andere Art und Weise erzeugen. Wenn man dann wieder das Aktualisieren des Submoduls ausführt, wird Git wieder den ursprünglichen Commit auschecken, allerdings hat man jetzt mit dem Branch einen Zeiger auf die neuen Commits und man kann sie leicht wieder auschecken.

Wenn ein Projekt ein Submodul enthält und man im Superprojekt zwischen einzelnen Branches hin und her wechseln möchte, kann sich das manchmal auch schwierig gestalten. Wenn man zum Beispiel einen neuen Branch erzeugt, in diesem dann ein Submodul hinzufügt und dann wieder in den ursprünglichen Branch, welcher das Submodul noch nicht enthält, zurückwechselt, hat man im Arbeitsverzeichnis immer noch das Submodul-Verzeichnis, welches auch so dargestellt wird, als ob es von Git noch nicht verfolgt wird:

```
$ git checkout -b rack
Switched to a new branch "rack"
$ git submodule add git@github.com:schacon/rack.git rack
```

```

Initialized empty Git repository in /opt/myproj/rack/.git/
...
Receiving objects: 100% (3184/3184), 677.42 KiB | 34 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.
$ git commit -am 'added rack submodule'
[rack cc49a69] added rack submodule
 2 files changed, 4 insertions(+), 0 deletions(-)
 create mode 100644 .gitmodules
 create mode 160000 rack
$ git checkout master
Switched to branch "master"
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       rack/

```

In diesem Fall muss man das Verzeichnis entweder an einen anderen Ort verschieben oder löschen. Im letzteren Fall muss man aber wieder das Submodul komplett klonen, wenn man in den anderen Zweig zurückwechselt. Außerdem kann man dabei lokale Änderungen zunichte machen oder Zweige, welche man noch nicht gepusht hat, verlieren.

Die letzte Falle, in die viele Leute tappen, tritt auf, wenn man bereits vorhandene Verzeichnisse in Submodule umwandeln will. Wenn man also Dateien, die bereits von Git verwaltet werden, entfernen und in ein entsprechendes Submodul verschieben möchte, muss man vorsichtig sein. Ansonsten können schwer zu behebende Probleme mit Git auftreten. Nehmen wir zum Beispiel an, dass Du die Dateien vom Rack-Projekt in ein Unterverzeichnis Deines Projekts abgelegt hast und diese jetzt aber in ein Submodul verschieben möchtest. Wenn Du das Unterverzeichnis einfach löschst und dann den Befehl `submodule add` ausführst, zeigt Dir Git folgende Fehlermeldung an:

```

$ rm -Rf rack/
$ git submodule add git@github.com:schacon/rack.git rack
'rack' already exists in the index

```

Man muss dann das Verzeichnis `rack` erst aus der Staging-Area entfernen. Danach kann man dann das Submodul erzeugen:

```

$ git rm -r rack
$ git submodule add git@github.com:schacon/rack.git rack
Initialized empty Git repository in /opt/testsub/rack/.git/
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 | 88 KiB/s, done.
Resolving deltas: 100% (1952/1952), done.

```

Wenn wir jetzt annehmen, dass Du diesen Vorgang innerhalb eines Zweigs durchgeführt hast und jetzt auf einen anderen Zweig, in dem das Submodul noch nicht existiert hat und damit die Dateien noch ganz normal im Repository enthalten waren, wechselst, erhält Du folgenden Fehler:

```
$ git checkout master
error: Untracked working tree file 'rack/AUTHORS' would be overwritten by merge.
```

Dann musst Du das Submodul-Verzeichnis rack an einen anderen Ort verschieben, bevor Du diesen Branch auschecken kannst:

```
$ mv rack /tmp/
$ git checkout master
Switched to branch "master"
$ ls
README  rack
```

Wenn man dann wieder in den Zweig mit dem Submodul zurückwechseln will, erhält man ein leeres Verzeichnis rack. Um dieses zu befüllen, kannst Du entweder `git submodule update` ausführen oder Du kannst Deine Kopie von `/tmp/rack` wieder an den ursprünglichen Ort wiederherstellen.

Subtree Merging

Nachdem wir neben den Vor- und Nachteilen beim Arbeiten mit Submodulen kennengelernt haben, möchte ich jetzt noch eine alternative Lösung zeigen, wie man ähnliche Probleme lösen kann. Wenn Git etwas zusammenführt, also mergt, analysiert es die Teile, die es mergen muss. Auf Basis dieser Analyse entscheidet Git sich für eine geeignete Merging-Methode. Wenn man zwei Branches mergt, dann verwendet Git automatisch die sogenannte Recursive-Strategie. Wenn man mehr als zwei Branches mergt, verwendet Git die sogenannte Octopus-Strategie. Diese Strategien werden automatisch für Dich gewählt, weil die Recursive-Strategie normalerweise sehr gut geeignet ist, um einen Drei-Wege-Merge (engl. three-way merge) durchzuführen — zum Beispiel, wenn es mehr als einen gemeinsamen Vorgänger-Commit gibt — aber der Drei-Wege-Merge ist nur für das Mergen von zwei Branches geeignet. Die Octopus-Merge-Strategie kann mehrere Branches zusammenführen, aber es wird dabei vorsichtiger vorgegangen, um schwierig aufzulösende Konflikte zu vermeiden. Aus diesem Grund wird diese Strategie standardmäßig verwendet, wenn man mehr als zwei Branches zusammenführen möchte.

Es gibt jedoch noch weitere Strategien, die man verwenden kann. Einer dieser Strategien ist der sogenannte Subtree-Merge. Dies kann verwendet werden, um unser Problem mit Unterprojekten zu lösen. Ich möchte Dir im Folgenden aufzeigen, wie man das Rack-Projekt aus dem letzten Kapitel einbindet und dabei den Subtree-Merge anstatt der Submodule verwendet.

Das Prinzip, das hinter einem Subtree-Merge steckt, ist, dass man zwei Projekte hat und eines der Projekte wird in ein Unterverzeichnis des anderen Projekts abgebildet. Wenn man ein Subtree-Merge ausführt, ist Git schlau genug, um zu erkennen, dass ein Projekt ein Abbild von einem anderen Projekt ist und es kann den Merge in geeigneter Weise durchführen — das ist wirklich sehr erstaunlich.

Als erstes musst Du dazu die Rack-Applikation zu Deinem Projekt hinzufügen. Dazu fügst Du das Rack-Projekt als neues Remote-Repository in Deinem Projekt hinzu und checkst dieses in einem separaten Branch aus:

```
$ git remote add rack_remote git@github.com:schacon/rack.git
$ git fetch rack_remote
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 git@github.com:schacon/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"
```

Nach der Ausführung der drei Befehle befindet sich das Rack-Projekt in Deinem Branch `rack_branch` und Dein eigenes Projekt liegt weiterhin im Branch `master`. Wenn man jetzt den jeweiligen Zweig auscheckt, sieht man die unterschiedlichen Inhalte im Wurzelverzeichnis:

```
$ ls
AUTHORS          KNOWN-ISSUES  Rakefile      contrib      lib
COPYING          README        bin           example      test
$ git checkout master
Switched to branch "master"
$ ls
README
```

Jetzt möchten wir das Rack-Projekt in Deinen Branch `master` als Unterverzeichnis hinzufügen. Dies kann man in Git mit dem Befehl `git read-tree` durchführen. In Kapitel 9 werde ich den Befehl `read-tree` und dessen verwandte Befehle näher erläutern. Hier möchte ich nur erklären, dass der Befehl das Wurzelverzeichnis eines Branches in die aktuelle Staging-Area und in das Arbeitsverzeichnis packt. Damit hast Du jetzt zu Deinem Branch `master` zurückgewechselt, den Inhalt des Branches `rack` in das Unterverzeichnis `rack` im Branch `master` Deines Projekts hinterlegt:

```
$ git read-tree --prefix=rack/ -u rack_branch
```

Wenn Du jetzt einen Commit ausführst, erscheint es einem so, als ob die ganzen Dateien aus dem Rack-Projekt in diesem Unterverzeichnis liegen — als ob man das Projekt aus einem Tarball-Container hineinkopiert hätte. Das Besondere ist jetzt aber, dass man Änderungen zwischen den verschiedenen Branches jetzt einfach zusammenführen kann. Das bedeutet, wenn das Rack-Projekt aktualisiert wird, kann man sich diese Änderungen einfach holen, indem man in diesen Branch wechselt und dort einen Pull durchführt:

```
$ git checkout rack_branch
$ git pull
```

Danach kann man diese Änderungen wieder in den Branch `master` mergen. Wenn man den Befehl `git merge -s subtree` verwendet, sollte dies einwandfrei funktionieren. Allerdings wird Git bei Ausführen dieses Befehls auch die jeweilige Historie mergen, was Du wahrscheinlich nicht haben möchtest. Um nun die Änderungen zu holen und eine entsprechende Commit-Nachricht vorzubereiten, hängt man einfach `--squash`, `--no-commit` und natürlich `-s subtree` als Option an:

```
$ git checkout master
$ git merge --squash -s subtree --no-commit rack_branch
Squash commit -- not updating HEAD
Automatic merge went well; stopped before committing as requested
```

Die ganzen Änderungen des Rack-Projekts wurden nun zusammengeführt, Du musst jetzt nur noch einen entsprechenden Commit durchführen. Man kann aber auch genau das Gegenteil machen: Man führt Änderungen im Unterverzeichnis `rack` des Branches `master` aus und mergt diese dann später in den Zweig `rack_branch`. Diesen kann man dann den Entwicklern des Rack-Projekts zur

Verfügung stellen.

Um die Unterschiede zwischen dem Inhalt in Deinem Verzeichnis `rack` und dem Code im Zweig `rack_branch` anzuzeigen, kann man keinen normalen Vergleich mit `diff` durchführen. Man muss stattdessen den Befehl `git diff-tree` verwenden und als Argument den zu vergleichenden Branch angeben:

```
$ git diff-tree -p rack_branch
```

Um Dein Verzeichnis `rack` mit dem letzten Stand des Branches `master` auf dem Server zu vergleichen, kannst Du folgenden Befehl verwenden:

```
$ git diff-tree -p rack_remote/master
```

Zusammenfassung

In diesem Kapitel hast Du viele ausgeklügelte Werkzeuge kennengelernt, die es Dir ermöglichen, Commits und die Staging-Area nach Deinen Vorstellungen zu beeinflussen. Wenn ein Problem in Deinem Projekt auftaucht, solltest Du jetzt leicht bestimmen können, welcher Commit den Fehler verursacht hat, sowie wann und von wem der Fehler begangen wurde. Wenn Du andere Projekte in Deinem Projekt verwenden möchtest, hast Du jetzt mehrere Möglichkeiten kennengelernt, wie Du dies handhaben kannst. An dieser Stelle solltest Du jetzt in der Lage sein, die meisten Dinge, die Du bei der täglichen Arbeit benötigst, in der Kommandozeile durchzuführen, ohne dass Dir dabei Schweißperlen auf der Stirn stehen.

Git individuell einrichten

Ich habe nun die grundlegende Funktionsweise und die Benutzung von Git besprochen. Weiterhin habe ich einige Werkzeuge von Git präsentiert, die dem Benutzer ein einfaches und effizientes Arbeiten ermöglichen. In diesem Kapitel werde ich nun auf einige Operationen eingehen, die Du benutzen kannst um die Funktionsweise von Git Deinen persönlichen Bedürfnissen anzupassen. Dazu führe ich einige wichtige Konfigurationseinstellungen ein, sowie verschiedene Einschubmethoden, auch Hooks genannt. Mit diesen Mitteln kann man Git leicht anpassen, sodass es genau Deinen Ansprüchen, des Unternehmens oder des Teams entspricht.

Git Konfiguration

Wie in Kapitel 1 schon kurz beschrieben, kann man die Konfiguration von Git mit Hilfe des Befehls `git config` steuern. Einer Deiner ersten Aktionen war es, Deinen Namen und E-Mail Adresse anzugeben:

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

Jetzt wirst Du einige weitere, interessantere Optionen kennenlernen, die Du auf gleiche Art und Weise einsetzen kannst, um Git Deiner Arbeitsumgebung anzupassen.

In Kapitel 1 hast Du bereits Deine ersten Erfahrungen mit einigen einfachen Einstellparametern von Git gemacht, aber ich möchte sie hier noch einmal kurz wiederholen. Git verwendet eine Reihe von Konfigurationsdateien, um Deine persönliche Einstellungen, welche von den Standard-Einstellungen abweichen, festzuhalten. Zu aller erst prüft Git die Einstellungen in der Datei `/etc/gitconfig`. Diese Datei enthält Werte, welche für alle Benutzer des Systems und deren Repositorys gelten. Wenn Du `git config` mit der Option `--system` benutzt, liest und schreibt Git von genau dieser Datei.

Als nächstes prüft Git die Datei `~/.gitconfig`, welche nur für den jeweiligen Benutzer gilt. Damit Git diese Datei zum Lesen und Schreiben nutzt, kannst Du die Option `--global` angeben.

Als Letztes sucht Git in der Konfigurationsdatei im Git Verzeichnis des gerade verwendeten Repositorys (`.git/config`). Die dort enthaltenen Parameter sind nur für dieses einzelne Repository gültig. Jede der erwähnten Ebenen überschreibt die vorhergehende. Das bedeutet, dass z.B. die Einstellungen in der Datei `/etc/gitconfig` von den Einstellungen in der Datei `.git/config` überschrieben werden. Du kannst alle Parameter auch durch manuelles Editieren der jeweiligen Datei setzen bzw. verändern (vorausgesetzt Du verwendest die richtige Syntax). In der Regel ist es aber einfacher den Befehl `git config` zu verwenden.

Grundlegende Client Konfiguration

Einstellparameter in Git lassen sich in zwei Kategorien aufteilen: Parameter für die Client-Konfiguration und für die Server-Konfiguration. Der Großteil der Einstellungen bezieht sich auf den Client – zur Konfiguration Deines persönlichen Arbeitsablaufs. Auch wenn es eine große Anzahl an Einstellmöglichkeiten gibt, werde ich nur die wenigen besprechen, die sehr gebräuchlich sind oder Deine Arbeitsweise bedeutend beeinflussen können. Viele Optionen sind nur für Spezialfälle interessant, auf die ich hier aber nicht weiter eingehen möchte. Falls Du eine Liste aller Optionen haben willst, kannst Du folgenden Befehl ausführen:

```
$ git config --help
```

Die Hilfeseite zu `git config` listet alle verfügbaren Optionen sehr detailliert auf.

core.editor

In der Grundeinstellung benutzt Git Deinen Standard Texteditor oder greift auf den Vi Editor zurück, um Deine Commit und Tag Nachrichten zu erstellen und zu bearbeiten. Um einen andern Editor als Standard einzurichten kannst Du die Option `core.editor` nutzen:

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

Ab jetzt wird Git immer Emacs starten um Nachrichten zu editieren, unabhängig davon welcher Standard Shell-Editor gesetzt ist.

commit.template

Wenn Du diese Einstellung auf einen Pfad zu einer Datei auf Deinem System einstellst, wird Git den Inhalt dieser Datei als Standard Commit Nachricht verwenden. Nehmen wir zum Beispiel an, Du erstellst eine Vorlage unter dem Namen `$HOME/.gitmessage.txt`, die den folgenden Inhalt hat:

```
subject line
```

```
what happened
```

```
[ticket: X]
```

Damit Git diese Datei als Standard Nachricht benutzt, die in Deinem Editor erscheint, wenn Du `git commit` aufrufst, richte die Option `commit.template` ein:

```
$ git config --global commit.template $HOME/.gitmessage.txt
$ git commit
```

Wenn Du dann das nächste Mal einen Commit durchführst, wird Dein Editor mit etwa der folgenden Nachricht starten:

```
subject line
```

```
what happened
```

```
[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
```

Falls eine Richtlinie für Commit Nachrichten existiert, solltest Du Git so konfigurieren, dass eine Vorlage davon bei einem Commit geladen wird. Dies erhöht die Chance, dass diese Richtlinie

auch eingehalten wird.

core.pager

Die Einstellung `core.pager` legt fest, welche Anwendung zur Seitenanzeige benutzt wird, wenn Git Text ausgibt, wie zum Beispiel bei `log` und `diff`. Du kannst es auch auf `more` oder eine andere Seitenanzeige Deiner Wahl (der Standard ist `less`) einstellen, oder Du kannst es mittels eines leeren Strings ganz ausschalten:

```
$ git config --global core.pager ''
```

Wenn Du dies ausführst, wird Git immer die komplette Ausgabe aller Befehle anzeigen, egal wie lange sie ist.

user.signingkey

Falls Du signierte kommentierte Tags erstellst (wie in Kapitel 2 beschrieben), so macht es die Arbeit leichter, wenn Du Deinen GPG Signierschlüssel in Git festlegst. Du kannst Deine Schlüssel ID wie folgt festlegen:

```
$ git config --global user.signingkey <gpg-key-id>
```

Beim Signieren von Tags mit Hilfe von `git tag` musst Du Deinen Schlüssel jetzt nicht mehr angeben. Es reicht folgendes auszuführen:

```
$ git tag -s <tag-name>
```

core.excludesfile

In Kapitel 2 habe ich bereits beschrieben, wie Du mit Hilfe der projektspezifischen `.gitignore` Datei Git dazu bringst, bestimmte Dateien nicht weiter zu verfolgen beziehungsweise zu stagen, wenn Du den Befehl `git add` verwendest. Falls Du jedoch eine weitere Datei außerhalb Deines Projekts verwenden willst, die diese Werte enthält oder zusätzliche Muster definiert, dann kannst Du Git mit der Option `core.excludesfile` mitteilen, wo sich diese Datei befindet. Trage hier einfach den Pfad zu einer Datei ein, welche entsprechend einer `.gitignore` Datei aufgebaut ist.

help.autocorrect

Diese Option ist in Git ab Version 1.6.1 verfügbar. Wenn Du in Git einen Befehl falsch schreibst, bekommst Du eine Meldung wie diese:

```
$ git com
git: 'com' is not a git-command. See 'git --help'.
```

```
Did you mean this?
    commit
```

Wenn Du die Option `help.autocorrect` auf 1 setzt, wird Git automatisch den entsprechenden Befehl ausführen, falls es in dieser Situation die einzige passende Alternative ist.

Farben in Git

Git kann für die Textanzeige im Terminal Farben benutzen, die Dir helfen können, die Ausgabe schnell und einfach zu begreifen. Mit einer Vielzahl von Optionen kannst Du die Farben an Deine Vorlieben anpassen.

`color.ui`

Wenn Du Git entsprechend konfigurierst, wird es den Großteil der Ausgaben automatisch farblich darstellen. Du kannst sehr detailliert einstellen, wie und welche Farben verwendet werden sollen, aber um die Standard-Terminalfarben zu aktivieren musst Du `color.ui` auf `,true‘` setzen:

```
$ git config --global color.ui true
```

Wenn dieser Wert gesetzt wurde, benutzt Git für seine Ausgaben Farben, sofern diese zu einem Terminal geleitet werden. Weitere mögliche Einstellungen sind `,false‘`, wodurch alle Farben deaktiviert werden, sowie `,always‘`, wodurch Farben immer aktiviert sind, selbst wenn Du Git Befehle in eine Datei oder über eine Pipe zu einem anderen Befehl umleitest.

Du wirst selten die Einstellung `color.ui = always` benötigen. In den meisten Fällen in denen Du in Deiner umgeleiteten Ausgabe Farben haben willst, kannst Du stattdessen die Option `--color` in der Kommandozeile benutzen. Damit weist Du Git an, die Farbkodierung für die Ausgabe zu verwenden. Die Einstellung `color.ui = true` sollte aber in den meisten Fällen Deinen Anforderungen genügen.

`color.*`

Falls Du im Detail einstellen willst, welche Befehle wie gefärbt werden, dann stellt Git Verb-spezifische Farbeinstellungen zur Verfügung. Jede dieser Optionen kann auf `true`, `false`, oder `always` eingestellt werden:

```
color.branch
color.diff
color.interactive
color.status
```

Zusätzlich hat jede dieser Einstellungen Unteroptionen, die Du benutzen kannst, um die Farbe für einzelne Teile der Ausgabe festzulegen. Um zum Beispiel die Meta Informationen in Deiner Diff Ausgabe mit blauem, fettem Text auf schwarzem Hintergrund darzustellen, kannst Du folgenden Befehl verwenden:

```
$ git config --global color.diff.meta "blue black bold"
```

Du kannst als Farben jeden der folgenden Werte verwenden: normal, black, red, green, yellow, blue, magenta, cyan, oder white. Falls Du ein Attribut wie z.B. die Fettschrift aus dem vorigen Beispiel verwenden willst, stehen Dir folgende Werte zur Auswahl: bold, dim, ul, blink, und reverse.

Auf der Manpage zu `git config` findest Du eine Liste aller Unteroptionen, die Du konfigurieren kannst.

Externe Merge- und Diff-Werzeuge

Bisher hast Du die in Git integrierte Implementierung von diff benutzt, aber Du kannst stattdessen auch eine externe Anwendung verwenden. Du kannst ebenso ein grafisches Merge-Werkzeug zur Auflösung von Konflikten einsetzen, statt diese manuell zu lösen. Ich werde demonstrieren, wie man das grafische Merge-Werkzeug von Perforce (P4Merge) konfiguriert, um Diffs und Merges zu bearbeiten. Ich habe P4Merge gewählt, da es ein freies und gutes grafisches Werkzeug ist.

Da P4Merge für die üblichen Plattformen verfügbar ist, sollte es kein Problem sein, es einmal auszuprobieren. In den Beispielen werde ich Pfadnamen nutzen, die auf Mac- und Linux-System funktionieren. Die Windows Benutzer müssen `/usr/local/bin` durch einen Pfad ersetzen, der in der Umgebungsvariable `PATH` gelistet ist.

Du kannst P4Merge hier herunterladen:

<http://www.perforce.com/perforce/downloads/component.html>

Als erstes solltest Du einige Wrapper Skripte erstellen um Deine Befehle auszuführen. Ich verwende hier die Pfade, die für einen Mac gelten. Auf anderen Systemen muss der Pfad zur ausführbaren Datei von P4Merge entsprechend angepasst werden. Mit den folgenden Befehlen erzeugen wir ein Skript mit dem Namen `extMerge`, welches die Anwendung mit allen angegebenen Argumenten aufruft:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/p4merge.app/Contents/MacOS/p4merge $*
```

Das Wrapper Skript für den Diff Befehl stellt sicher, dass es mit sieben Parametern aufgerufen wird und leitet zwei von diesen an das Merge Skript weiter. Standardmäßig übergibt Git die folgenden Argumente an das Diff-Werkzeug:

```
path old-file old-hex old-mode new-file new-hex new-mode
```

Da nur die Parameter `old-file` und `new-file` benötigt werden, verwenden wir das Wrapper Skript um nur die notwendigen Parameter weiterzugeben.

```
$ cat /usr/local/bin/extDiff
#!/bin/sh
[ $# -eq 7 ] && /usr/local/bin/extMerge "$2" "$5"
```

Außerdem muss sichergestellt werden, dass die Skripte ausführbar sind::

```
$ sudo chmod +x /usr/local/bin/extMerge
$ sudo chmod +x /usr/local/bin/extDiff
```

Jetzt kannst Du Git so konfigurieren, dass es Deine persönlichen Merge- und Diff-Werkzeuge benutzt. Dazu sind einige weitere Einstellungen nötig: `merge.tool`, um die von Git verwendete Merge Strategie festzulegen, `mergetool.*.cmd`, um festzulegen, wie der Befehl auszuführen ist, `mergetool.trustExitCode`, damit Git weiß, ob der Exit-Code des Programms eine erfolgreiche Merge Auflösung anzeigt oder nicht, und `diff.external`, um einzustellen welches Diff Kommando Git benutzen soll. Du kannst also entweder die vier folgenden Befehle ausführen

```
$ git config --global merge.tool extMerge
$ git config --global mergetool.extMerge.cmd \
    'extMerge "$BASE" "$LOCAL" "$REMOTE" "$MERGED"'
$ git config --global mergetool.trustExitCode false
$ git config --global diff.external extDiff
```

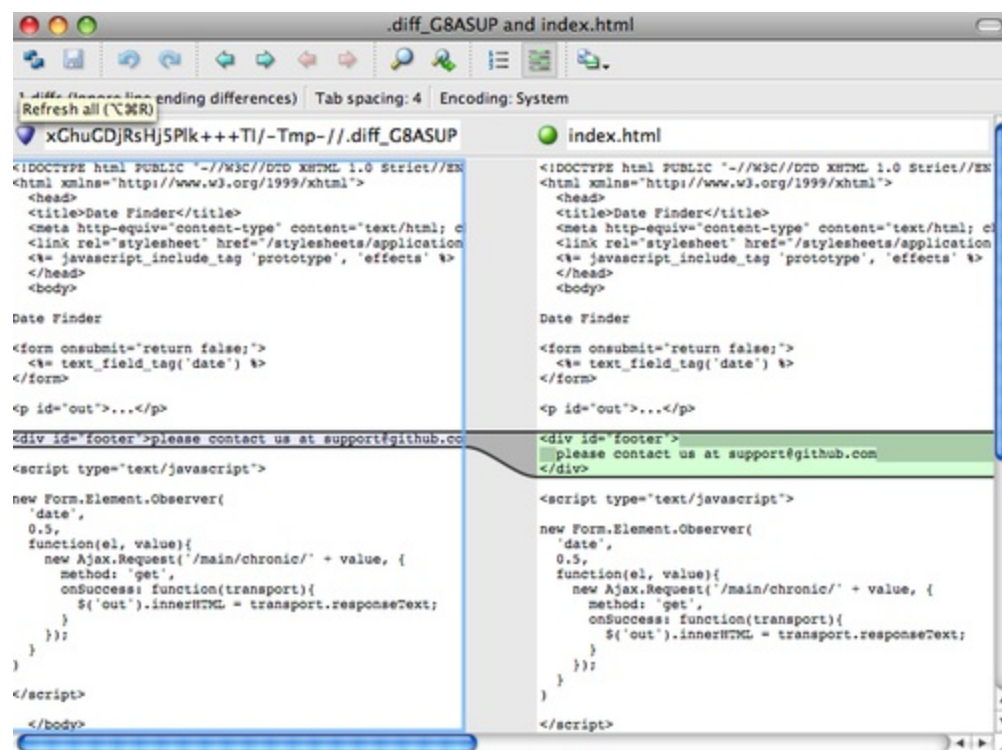
oder Du bearbeitest Deine `~/.gitconfig` Datei und fügst dort folgende Zeilen hinzu:

```
[merge]
  tool = extMerge
[mergetool "extMerge"]
  cmd = extMerge \"$BASE\" \"$LOCAL\" \"$REMOTE\" \"$MERGED\"
  trustExitCode = false
[diff]
  external = extDiff
```

Nach Setzen dieser Einstellungen und beim Ausführen eines Diff Befehls wie den folgenden:

```
$ git diff 32d1776b1^ 32d1776b1
```

wird Git P4Merge starten, anstatt den Vergleich in der Kommandozeile auszugeben. Abbildung 7-1 zeigt hierzu ein Beispiel.



Wenn Du versuchst zwei Branches zu mergen und dabei Merge Konflikte auftreten, kannst Du den Befehl `git mergetool` ausführen. Das Kommando startet P4Merge und erlaubt es Dir, die Konflikte mit Hilfe des grafischen Werkzeugs aufzulösen.

Das Tolle an dem Wrapper Ansatz ist, dass Du Deine Diff- und Merge-Werkzeuge sehr leicht wechseln kannst. Wenn Du zum Beispiel für `extDiff` und `extMerge` statt P4Merge, `KDiff3` verwenden willst, musst Du lediglich Dein Wrapper Skript `extMerge` anpassen:

```
$ cat /usr/local/bin/extMerge
#!/bin/sh
/Applications/kdiff3.app/Contents/MacOS/kdiff3 $*
```

Ab jetzt verwendet Git `KDiff3` zur Anzeige von Diffs und zur Auflösung von Merge Konflikten.

Git wird bereits mit Standard-Einstellungen für verschiedene Merge-Auflösungswerkzeuge ausgeliefert, sodass Du diese nicht extra konfigurieren musst. Als Merge-Werkzeug kann Du `kdiff3`, `opendiff`, `tkdiff`, `meld`, `xxdiff`, `emerge`, `vimdiff` oder `gvimdiff` einstellen. Wenn Du `KDiff3` nur zum Auflösen von Konflikten und nicht für einen Diff verwenden willst, kannst Du den folgenden Befehl ausführen (vorausgesetzt `KDiff3` befindet sich im Standard-Pfad):

```
$ git config --global merge.tool kdiff3
```

Wenn Du diesen Befehl ausführst, anstatt die `extMerge` und `extDiff` Skripte zu erstellen, dann wird Git `KDiff3` zum Auflösen von Merge Konflikten verwenden. Für einen Vergleich verwendet Git weiterhin das integrierte Diff-Werkzeug.

Formatierungen und Leerzeichen

Bei der Zusammenarbeit mit anderen Entwicklern sind Probleme mit Formatierungen und

Leerzeichen einige der frustrierendsten und heikelsten Themen denen viele Entwickler begegnen, vor allem bei plattformübergreifenden Projekten. Es kann sehr leicht passieren, dass durch Patches oder andere gemeinsame Arbeit fast unmerklich Leerzeichen Änderungen eingeführt werden, z.B. weil ein Editor sie stillschweigend einfügt. Beim Programmieren unter Windows können durch Änderungen an einer Zeile auch leicht Wagenrückläufe (CR) am Zeilenende eingefügt werden (relevant bei plattformübergreifenden Projekten). Git kann mit ein paar Einstellungen hierbei unterstützend eingreifen.

core.autocrlf

Falls Du unter Windows programmierst oder ein anderes System benutzt und mit anderen zusammenarbeitest, die unter Windows programmieren, wirst Du sehr wahrscheinlich irgendwann Problemen mit Zeilenenden begegnen. Dies liegt daran, dass Windows sowohl ein CR Zeichen, als auch ein LF Zeichen zum Signalisieren einer neuen Zeile in Dateien verwendet. Mac und Linux nutzen stattdessen nur ein LF Zeichen (Mac OS bis Version 9 verwendet ein einzelnes CR Zeichen). Dies ist eine kleine, aber extrem störende Tatsache beim Arbeiten über Plattformgrenzen hinweg.

Git kann dies vermeiden, indem es CRLF am Zeilenende automatisch zu LF konvertiert, wenn Du ein Commit durchführst, und umgekehrt wenn es Code in Dein lokales Dateisystem auscheckst. Du kannst diese Funktionalität mittels der Option `core.autocrlf` aktivieren. Falls Du auf einem Windows System arbeitest, setze sie auf `true` — dies konvertiert LF zu CRLF, wenn Du Code auscheckst:

```
$ git config --global core.autocrlf true
```

Falls Du auf einem Linux oder Mac System arbeitest, welches LF Zeilenenden verwendet, dann soll Git keine Datei automatisch konvertieren, wenn sie ausgecheckt wird. Wenn allerdings versehentlich eine Datei mit CRLF in das Repository eingeführt wurde, dann möchtest Du vielleicht, dass Git dies automatisch für Dich repariert. Wenn Du den Parameter `core.autocrlf` auf `input` setzt, wird Git bei einem Commit automatisch CRLF in LF umwandeln. Allerdings nicht in die andere Richtung bei einem Checkout:

```
$ git config --global core.autocrlf input
```

Mit dieser Einstellung solltest Du CRLF Zeilenenden in unter Windows ausgecheckten Dateien haben und LF Zeilenenden auf Mac und Linux Systemen und im Repository.

Falls Du ein Windows Programmierer bist und an einem Projekt arbeitest, welches nur unter Windows entwickelt wird, dann kannst Du diese Funktionalität auch deaktivieren. In diesem Fall werden Zeilenenden mit CRLF im Repository gespeichert. Dazu setzt Du die Option auf `false`:

```
$ git config --global core.autocrlf false
```

core.whitespace

Git ist so voreingestellt, dass es einige Leerzeichen Probleme erkennen und beheben kann. Es kann nach vier grundlegenden Problemen mit Leerzeichen suchen — Zwei davon sind standardmässig aktiviert und können deaktiviert werden. Die anderen beiden sind inaktiv, können aber aktiviert werden.

Die zwei standardmäßig aktiven Optionen sind `trailing-space`, das nach Leerzeichen am Ende einer Zeile sucht, und `space-before-tab`, das nach Leerzeichen vor Tabulatoren am Anfang einer Zeile sucht.

Die beiden aktivierbaren, aber normalerweise deaktivierten Optionen sind `indent-with-non-tab`, welches nach Zeilen sucht, die mit acht oder mehr Leerzeichen anstelle von Tabulatoren beginnen, und `cr-at-eol`, wodurch Git angewiesen wird, dass CR Zeichen am Zeilenende in Ordnung sind.

Du kannst Git mitteilen, welche dieser Optionen es aktivieren soll, indem Du `core.whitespace` auf die Werte setzt, die Du an- oder abgeschaltet haben möchtest. Die jeweiligen Werte werden mit einem Komma getrennt. Du kannst Optionen deaktivieren, indem Du sie entweder aus der Parameterliste entfernst, oder ihnen ein `-` Zeichen voranstellst. Wenn Du zum Beispiel alle Optionen außer `cr-at-eol` aktivieren willst, kannst Du folgenden Befehl ausführen:

```
$ git config --global core.whitespace \
    trailing-space,space-before-tab,indent-with-non-tab
```

Git wird die möglichen Problemstellen erkennen, wenn Du den `git diff` Befehl ausführst, und es wird versuchen, sie farblich hervorzuheben, damit Du sie vor einem Commit beheben kannst. Git wird diese Einstellungen auch benutzen, um Dir zu helfen, wenn Du mit `git apply Patches` anwendest. Wenn Du Patches anwendest, kannst Du Git anweisen eine Warnung auszugeben, falls es beim Patchen die spezifizierten Leerzeichenprobleme erkennt:

```
$ git apply --whitespace=warn <patch>
```

Oder Du kannst Git versuchen lassen, diese Probleme automatisch zu beheben, bevor es den Patch anwendet:

```
$ git apply --whitespace=fix <patch>
```

Diese Optionen gelten auch für den Rebase Befehl. Falls Du einen Commit gemacht hast, der problematische Leerzeichen enthält, aber Du die Änderungen noch nicht auf den Server gepusht hast, kannst Du ein rebase mit dem Parameter `--whitespace=fix` ausführen. Damit behebt Git automatisch die Leerzeichenfehler während des Rebase-Vorgangs.

Server Konfiguration

Es gibt nicht annähernd so viele Konfigurationsmöglichkeiten für die Serverfunktionalitäten von Git, aber es gibt dabei einige interessante Parameter, die Du Dir anschauen solltest.

receive.fsckObjects

Die Objekte, die Git durch einen Push empfängt, werden von Haus aus nicht auf Konsistenz geprüft. Auch wenn Git sicherstellen kann, dass jedes Objekt mit dessen SHA-1 Checksumme übereinstimmt und auf gültige Objekte verweist, so wird dies standardmäßig nicht bei jedem Push durchgeführt. Das ist eine aufwändige Operation und kann abhängig von der Größe des Repositorys oder dem Push eine Menge Zeit kosten. Wenn Du die Objektkonsistenz bei jedem Push durch Git prüfen lassen willst, so kannst Du das erzwingen, indem Du `receive.fsckObjects` auf `,true‘` setzt:

```
$ git config --system receive.fsckObjects true
```

Ab jetzt prüft Git die Integrität des Repositorys bevor der Push akzeptiert wird. Damit ist sichergestellt, dass kein Client korrupte Daten einspeist.

receive.denyNonFastForwards

Falls Du auf Commits, die bereits gepusht sind, einen Rebase anwendest, und diese dann versuchst zu pushen, wird Git dies mit einer Fehlermeldung zurückweisen. Wenn der Remote Branch auf einen Commit zeigt, welcher nicht in Deinem lokalen Branch enthalten ist und Du versuchst diesen Branch zu pushen, wird sich Git genau gleich verhalten und den Push verweigern. Das ist in den meisten Fällen eine gute Richtlinie, aber im Falle eines Rebase ist eventuell ein anderes Verhalten gewünscht (vorausgesetzt Du weißt was Du tust). Dann kannst Du den Push auch erzwingen, indem Du den Parameter `-f` zu dem Push Kommando hinzufügst.

Aktualisierungen auf dem Remote Branch, welche nicht einem Fast-Forward entsprechen können durch Setzen des Parameters `receive.denyNonFastForward` auf den Wert `,true‘` deaktiviert werden:

```
$ git config --system receive.denyNonFastForwards true
```

Eine andere Möglichkeit ist die Einrichtung von serverseitigen Hooks, die ich etwas später noch beschreiben werde. Dieser Ansatz erlaubt noch komplexere Szenarien. Man kann z.B. die Pushes, welche nicht einem Fast-Forward entsprechen nur für bestimmte Benutzergruppen verweigern.

receive.denyDeletes

Es ist möglich die Option `denyNonFastForwards` zu umgehen, indem man den Remote Branch zuerst löscht und dann mit einer neuen Referenz pusht. In neueren Versionen von Git (ab Version 1.6.1) kann man den Parameter `receive.denyDeletes` auf `,true‘` setzen:

```
$ git config --system receive.denyDeletes true
```

Dies verbietet grundsätzlich jedem Benutzer das Löschen eines Branches oder Tags. Um einen Remote Branch zu löschen müssen die ref Dateien manuell vom Server entfernt werden. Es gibt aber auch noch andere interessantere Wege dies auf Benutzerbasis über Zugriffssteuerungslisten (ACL) durchzuführen. Ich werde dies am Ende dieses Kapitel noch vorstellen.

Git Attribute

Einige dieser Einstellungen können auch auf einen Pfad beschränkt werden, sodass sie nur für bestimmte Unterverzeichnisse oder eine Gruppe von Dateien gültig sind. Diese Einstellungen werden Git Attribute genannt und werden in der Datei `.gitattributes` in einem der Projektverzeichnisse verwaltet (üblicherweise im Root-Verzeichnis Deines Projekts). Alternativ kannst Du diese auch unter `.git/info/attributes` ablegen. In diesem Fall werden die Attribute nicht in das Repository eingecheckt und gelten nur für dieses einzelne, lokale Repository.

Mittels den Attributen ist es zum Beispiel möglich, verschiedene Merge Strategien für einzelne Dateien oder Verzeichnisse innerhalb Deines Projekts vorzugeben. Ebenso kannst Du Git anweisen, wie ein Vergleich von Binärdateien durchzuführen ist. Oder Du konfigurierst Git so, dass der Inhalt von Dateien vorgefiltert wird, wenn Du ein Commit oder Checkout durchführst. In diesem Abschnitt wirst Du einiger der Attribute kennenlernen, die Du für die einzelnen Verzeichnisse in Deinem Git Projekt vorgeben kannst. Außerdem werde ich einige Beispiele aus der Praxis näher erläutern.

Binärdateien

Mit Hilfe der Git Attribute ist es Dir möglich, Git mitzuteilen, welche Dateien binär sind (für den Fall, dass Git nicht in der Lage ist, dies selbst festzustellen) und wie Git diese behandeln soll. Es kann zum Beispiel sein, dass automatisiert, erstellte Textdateien nicht einfach verglichen werden können. Oder umgekehrt können manche Binärdateien leicht von einem Menschen verglichen werden. Ich werde jetzt aufzeigen, wie Du Git konfigurierst damit es solche Dateien unterscheiden kann.

Binärdateien erkennen

Manche Dateien sehen zwar wie Textdateien aus, sollten aber streng genommen als Binärdateien behandelt werden. So enthalten zum Beispiel Xcode Projekte auf dem Mac eine Datei mit der Endung `.pbxproj`. Die Datei ist eigentlich nur ein JSON-Datensatz (ein Klartext Javascript Datenformat), der von der IDE gespeichert wird und unter anderem die Build Einstellungen enthält. Obwohl sie nur ASCII Zeichen enthält und damit technisch gesehen eine Textdatei ist, sollte man diese nicht als solche behandeln. In Wirklichkeit ist diese Datei eine kleine Datenbank, deren Inhalt nicht zusammengeführt werden kann, wenn zwei Leute sie geändert haben. Das Vergleichen der Datei ist ebenso selten hilfreich. Die Datei ist für die Verarbeitung durch einen Computer gedacht. Kurz gesagt, Du willst, dass man sie als Binärdatei behandelt.

Um Git anzuweisen alle `pbxproj` Dateien als Binärdateien zu behandeln, kannst Du die folgende Zeile zu Deiner `.gitattributes` Datei hinzufügen:

```
*.pbxproj -crlf -diff
```

Ab jetzt wird Git nicht mehr versuchen CRLF Probleme zu lösen oder die Datei beim Commit oder Checkout zu ändern. Außerdem ermittelt Git keine Dateiunterschiede mehr und gibt diese

auch nicht aus, wenn Du den Befehl `git show` oder `git diff` ausführst. Alternativ gibt es auch ein integriertes Makro `binary`, welches den Parametern `-crlf -diff` entspricht:

```
*.pbxproj binary
```

Diff bei Binärdateien

Mit Hilfe der Git Attribute können Unterschiede in binären Dateien effektiv und leicht angezeigt werden. Du kannst Git so konfigurieren, dass es automatisch Binärdateien in Textdateien umwandelt, damit sie mit einem normalen Diff verglichen werden können. Meist stellt sich aber die Frage, wie man binäre Daten in Text konvertieren soll. Wenn man ein Werkzeug findet, welches einem diese Konvertierung abnimmt und die binäre Daten in ein Textformat umwandelt, ist dies meist die beste Lösung. Leider gibt es nur sehr wenige binäre Formate, die sich dafür eignen, dass man sie in lesbare Textformate umwandelt (Ich denke dabei zum Beispiel an Audio-Daten). Wenn dies der Fall ist und Du keine geeignete Möglichkeit gefunden hast, die Daten in lesbare Form zu wandeln, dann ist es oft relativ einfach eine entsprechende Beschreibung des eigentlichen Inhalts zu erhalten. Alternativ gibt es noch Metadaten, wobei Metadaten einem nicht ein vollständiges Abbild vom Dateiinhalt liefern können, aber in diesem Fall ist das besser als gar nichts.

Im folgenden Abschnitt werden wir beide Möglichkeiten besprechen, wie man für weit verbreitete binäre Formate eine lesbare Form für einen Vergleich erhält.

Anmerkung: Es gibt verschiedene Arten von binären Formaten, welche Text beinhalten, und es ist meist sehr schwierig einen passenden Konverter zu finden. In solchen Fällen kann man sein Glück mit dem `strings`-Programm versuchen. Manche der Formate verwenden ein UTF-16 Encoding oder andere Zeichentabellen. In diesem Fall wird es mit dem Programm `strings` meist nicht funktionieren. Da `strings` jedoch auf den meisten Mac- und Linux-Systemen verfügbar ist, sollte man es durchaus auf einen Versuch ankommen lassen.

MS Word files

Als erstes werden wir die beschriebene Technik benutzen um eines der lästigsten Probleme der Menschheit zu lösen: Versionskontrolle von Word Dokumenten. Jeder weiß, dass Word der schrecklichste Editor der Welt ist, aber trotzdem benutzt ihn jeder. Wenn Du Word Dokumente versionieren willst, kannst Du sie in Dein Repository packen und ab und zu einen Commit durchführen. Aber wozu ist das nützlich? Wenn Du einen Vergleich mit `git diff` ausführst, erhältst Du ähnliche Ausgabe wie diese:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index 88839c4..4afcb7c 100644
Binary files a/chapter1.doc and b/chapter1.doc differ
```

Du kannst zwei Versionen nicht direkt vergleichen, außer Du checkst sie aus und prüfst sie manuell, richtig? Es stellt sich heraus, dass dies recht gut mittels Git Attributen möglich ist. Füge

dazu die folgende Zeile in Deine `.gitattributes` Datei ein:

```
*.doc diff=word
```

Dies weist Git an, dass auf jede Datei, die diesem Dateimuster (`.doc`) entspricht, der „word“ Filter angewandt werden soll, wenn Du versuchst, einen Diff mit Dateiunterschieden anzusehen. Was ist nun der „word“ Filter? Dieser muss von Dir noch konfiguriert werden. Du kannst Git so konfigurieren, dass es das `catdoc` Programm verwendet um Word Dokumente in lesbare Textdateien zu konvertieren. `catdoc` wurde speziell dafür entwickelt um lesbaren Text aus binären MS Word Dokumenten zu extrahieren (du erhältst es unter <http://www.wagner.pp.ru/~vitus/software/catdoc/>). Bei jedem Diff wird Git diese Konvertierung durchführen:

```
$ git config diff.word.textconv catdoc
```

Dieser Befehl fügt in der Datei `.git/config` eine Sektion mit folgendem Aufbau hinzu:

```
[diff "word"]
    textconv = catdoc
```

Bei jedem Vergleich von zwei Schnappschüssen wird Git Dateien mit der Dateiendung `.doc` durch den „word“ Filter jagen, welcher durch das `catdoc` Programm definiert ist. Das erzeugt gut lesbare Textversionen Deiner Word Dateien, die für den Vergleich herangezogen werden.

Dazu ein Beispiel. Ich habe Kapitel 1 des Buches in ein Word-Dokument eingefügt und in Git gespeichert. Danach habe ich etwas Text in einem Absatz geändert, die Datei gespeichert und den Befehl `git diff` ausgeführt um zu prüfen, was sich geändert hat:

```
$ git diff
diff --git a/chapter1.doc b/chapter1.doc
index c1c8a0a..b93c9e4 100644
--- a/chapter1.doc
+++ b/chapter1.doc
@@ -128,7 +128,7 @@ and data size)
    Since its birth in 2005, Git has evolved and matured to be easy to use
    and yet retain these initial qualities. It's incredibly fast, it's
    very efficient with large projects, and it has an incredible branching
    -system for non-linear development.
+system for non-linear development (See Chapter 3).
```

Git war erfolgreich und zeigt nun kurz und bündig an, dass ich den Text „(See Chapter 3)“ hinzugefügt habe, was korrekt ist. Wie du siehst, funktioniert perfekt.

OpenDocument Textdateien

Bei OpenDocument Textdateien (`*.odt`), die mit OpenOffice erstellt wurden, können wir die gleiche Herangehensweise wie bei MS Word Dateien (`*.doc`) anwenden.

Füge die folgende Zeile zu der `.gitattributes` Datei hinzu:

```
*.odt diff=odt
```

Jetzt müssen wir noch den odt Diff Filter in der .git/config hinzufügen:

```
[diff "odt"]
    binary = true
    textconv = /usr/local/bin/odt-to-txt
```

OpenDocument Dateien sind eigentlich komprimierte Zip Verzeichnisse, die mehrere Dateien enthalten (der Inhalt: XML-Dateien, Stylesheets, Bilder, usw.). Wir müssen ein Skript schreiben um den Inhalt zu extrahieren und das Ergebnis als reinen Text zurückliefern. Erzeuge dazu eine Datei /usr/local/bin/odt-to-txt (die Datei kann in einem beliebigen Verzeichnis abgelegt werden) mit dem folgenden Inhalt:

```
#!/usr/bin/env perl
# Simplistic OpenDocument Text (.odt) to plain text converter.
# Author: Philipp Kempgen

if (! defined($ARGV[0])) {
    print STDERR "No filename given!\n";
    print STDERR "Usage: $0 filename\n";
    exit 1;
}

my $content = '';
open my $fh, '-|', 'unzip', '-qq', '-p', $ARGV[0], 'content.xml' or die $!;
{
    local $/ = undef; # slurp mode
    $content = <$fh>;
}
close $fh;
$_ = $content;
s/<text:span\b[^>]*>\/g; # remove spans
s/<text:h\b[^>]*>\/\n\n**** /g; # headers
s/<text:list-item\b[^>]*>\s*<text:p\b[^>]*>\/\n -- /g; # list items
s/<text:list\b[^>]*>\/\n\n/g; # lists
s/<text:p\b[^>]*>\/\n /g; # paragraphs
s/<[^>]+>\/g; # remove all XML tags
s\/\n{2,}\/\n\n/g; # remove multiple blank lines
s\/\A\n+\/g; # remove leading blank lines
print "\n", $_, "\n\n";
```

Nun musst Du diese Datei noch ausführbar machen:

```
chmod +x /usr/local/bin/odt-to-txt
```

Jetzt kann Dir git diff aufzeigen, was sich in .odt Dateien geändert hat.

Bilddateien

Auf diese Art und Weise kann man ein weiteres, interessantes Problem lösen. Das Vergleichen

von Bilddateien. Eine Möglichkeit dies zu tun, ist es, JPEG Dateien durch einen Filter zu schicken, der ihre EXIF Bildinformationen extrahiert. EXIF Bildinformationen sind Metadaten, die den meisten Bilddateien beigelegt werden. Wenn Du das Programm `exiftool` herunterlädst und installierst, kannst Du es benutzen um Deine Bilder in einen Text mit diesen Metainformationen umzuwandeln. Damit kann Dir ein Diff zumindest eine textuelle Repräsentation aller Veränderungen an der Datei anzeigen:

```
$ echo '*.png diff=exif' >> .gitattributes
$ git config diff.exif.textconv exiftool
```

Wenn Du nun ein Bild in Deinem Projekt ersetzt und `git diff` ausführst, erhältst Du in etwa folgende Ausgabe:

```
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  : 2009:04:17 10:12:35-07:00
+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
```

Man sieht auf einen Blick, dass sowohl Dateigröße als auch die Bildabmessungen verändert wurden.

Schlüsselwörterweiterung

Entwickler, die an SVN- oder CVS-ähnliche Systeme gewöhnt sind, fragen oft nach der Möglichkeit Schlüsselwörter zu erweitern oder zu ersetzen. Mit Git ist dies nicht so einfach möglich, da eine Datei nach einem durchgeführten Commit nicht mehr verändert werden kann. Die Information über den Commit kann also nicht zur Datei hinzugefügt werden, da Git bereits vor dem Commit die Prüfsumme berechnet. Jedoch hast Du die Möglichkeit Text einzufügen, wenn die Datei ausgecheckt wird und diesen dann wieder entfernen, wenn die Datei zu einem Commit hinzugefügt wird. Die Git Attribute bieten hierfür zwei Möglichkeiten an.

Zunächst kannst Du die SHA-1 Prüfsumme eines Blobs automatisch in ein `Id` Feld einer Datei einfügen. Wenn Du das folgende Attribut für eine oder eine Gruppe von Dateien einstellst, wird Git dieses Feld beim nächsten Checkout mit dem SHA-1 Wert dessen Blobs ersetzen. Hierbei ist es wichtig zu beachten, dass es die Prüfsumme des Blobs selbst ist, und nicht die des Commits:

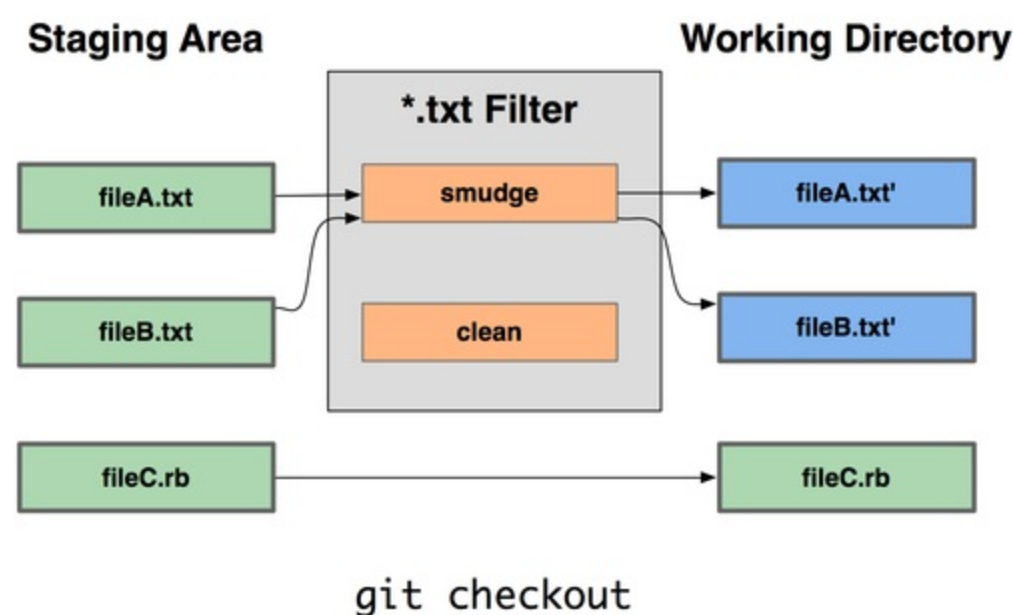

```
$ echo '*.txt ident' >> .gitattributes
$ echo '$Id$' > test.txt
```

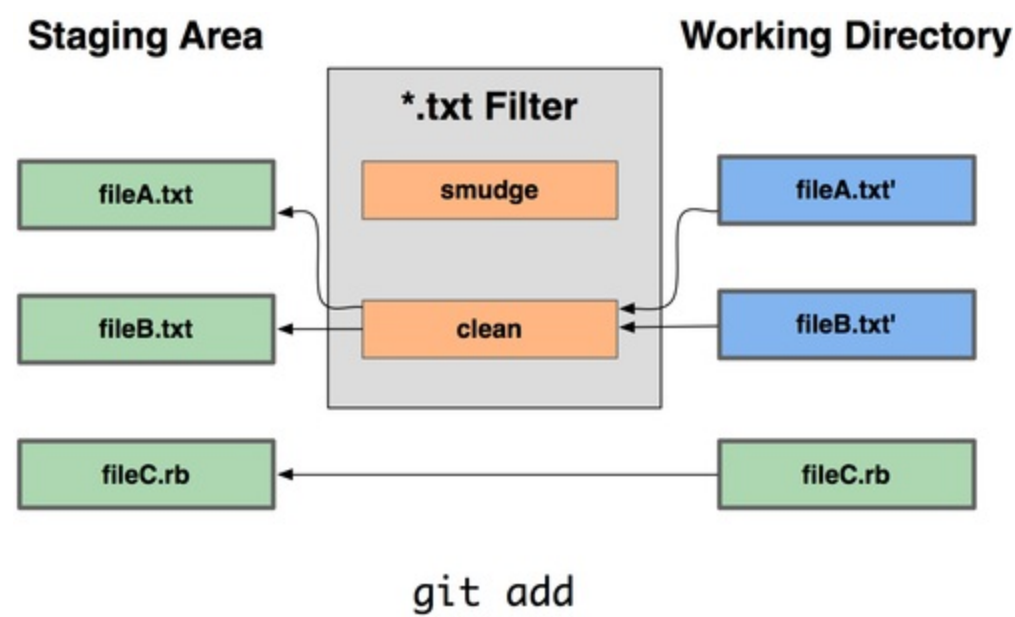
Wenn Du diese Datei das nächste Mal auscheckst, wird Git den SHA Wert des Blobs einfügen:

```
$ rm test.txt
$ git checkout -- test.txt
$ cat test.txt
$Id: 42812b7653c7b88933f8a9d6cad0ca16714b9bb3 $
```

Allerdings ist das Ergebnis nur beschränkt verwertbar. Die SHA Werte als solches sind nicht sehr hilfreich, da sie recht zufällig sind und nicht festgestellt werden kann ob ein SHA Wert älter oder neuer ist, als der andere. In anderen Systemen, wie CVS oder Subversion kann man mit Hilfe der Keyword Expansion Datum- und Zeitstempel einfügen.

Wie sich herausstellt, kann man aber seine eigenen Filter schreiben, um bei Commits oder Checkouts Schlüsselwörter in Dateien zu ersetzen. In der `.gitattributes` Datei kann man einen Filter für bestimmte Pfade angeben und dann Skripte einrichten, die Dateien kurz vor einem Checkout („smudge“, siehe Abbildung 7-2) und kurz vor einem Commit („clean“, siehe Abbildung 7-3) modifizieren. Diese Filter können eingerichtet werden, um alle möglichen witzigen Dinge zu machen.





Die Beschreibung des ersten Commits dieser Funktionalität enthält ein einfaches Beispiel, wie man all seinen C Quellcode durch das `indent` Programm leiten lassen kann, bevor ein Commit gemacht wird. Du kannst dies einrichten, indem Du das entsprechende Filterattribut in der `.gitattributes` Datei auflistest, damit `*.c` Dateien mit dem „indent“ Programm gefiltert werden:

```
*.c      filter=indent
```

Dann muss Git noch gesagt werden, was der „indent“ Filter bei „smudge“ und „clean“ zu tun hat:

```
$ git config --global filter.indent.clean indent
$ git config --global filter.indent.smudge cat
```

Wenn ein Commit Dateien umfasst, die dem Muster `*.c` entsprechen, wird Git diese Dateien vor Ausführung des Commits durch das `indent` Programm leiten. Werden sie wieder ausgecheckt, so schickt Git sie durch das `cat` Programm. `cat` ist im Grunde genommen eine Null-Operation: es gibt genau die Daten wieder aus, die hereinkommen. Diese Einstellung bewirkt also tatsächlich nur, dass alle C Quellcode Dateien vor einem Commit durch den `indent` Filter bearbeitet werden.

Ein weiteres interessantes Beispiel ermöglicht im Stile von RCS die Schlüsselworterweiterung `$Date$`. Damit dies vernünftig funktioniert, brauchst Du ein kleines Skript, welches mit Hilfe des Dateinamen das letzte Commitdatum in diesem Projekt herausfindet und dieses Datum in die Datei einfügt. Hierzu ein kleines Beispiel als Ruby Skript:

```
#!/usr/bin/env ruby
data = STDIN.read
last_date = `git log --pretty=format:"%ad" -1`
puts data.gsub('$Date$', '$Date: ' + last_date.to_s + '$')
```

Das Skript ermittelt das letzte Commitdatum mittels des Befehls `git log`, ersetzt jede Zeichenfolge von `$Date` im Stream `stdin` mit dem Commitdatum und gibt das Ergebnis wieder aus. Dieses Skript sollte auch in der Skriptsprache Deiner Wahl leicht umzusetzen sein. Am besten nennst Du dieses Skript `expand_date` und legst es in Deinem Standard Suchpfad ab. Nun

musst Du noch einen Filter (nennen wir ihn `dater`) in Git einrichten, der Dein `expand_date` Skript benutzt, um die Textdateien beim Checkout zu modifizieren. Zum Säubern der Dateien wird beim Commit ein Perl Ausdruck verwendet:

```
$ git config filter.dater.smudge expand_date
$ git config filter.dater.clean 'perl -pe "s/\\\$Date[^\$]*\\\$/\\\$Date\\\$/"
```

Um wieder zum Ursprungszustand zurückzukehren entfernt dieses kurze Perl Schnipsel alles was es in einer `$Date$` Zeichenfolge findet. Jetzt da Dein Filter fertig ist, kannst Du ihn testen indem Du eine Datei mit dem `$Date$` Schlüsselwort erstellst und das entsprechende Git Attribut für diese Datei einrichtest:

```
$ echo '# $Date$' > date_test.txt
$ echo 'date*.txt filter=dater' >> .gitattributes
```

Wenn Du diese Änderungen eincheckst und wieder erneut auscheckst, sollte Dein Schlüsselwort korrekt ersetzt worden sein:

```
$ 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
# $Date: Tue Apr 21 07:26:52 2009 -0700$
```

Man kann sehen wie mächtig diese Technik für Deinen Entwickleralltag sein kann. Da die `.gitattributes` Datei ebenfalls im Git Repository verwaltet wird und damit an alle Benutzer weitergeben wird, solltest Du vorsichtig mit Filtern umgehen. Denn Dein Filterskript (in diesem Fall das Skript `dater`) liegt nicht unter Versionskontrolle. Deshalb kann es passieren, dass die Schlüsselwortersetzung beziehungsweise das Arbeiten mit dem Repository nicht bei jedem funktioniert. Beim Entwickeln von Filtern solltest Du deshalb darauf achten, dass das Projekt weiterhin benutzt werden kann, auch wenn ein Filter einmal fehlschlägt.

Exportieren von Repositories

Git Attribute erlauben auch einige interessante Dinge, wenn Du Dein Projekt in ein Archiv exportierst.

export-ignore

Du kannst Git anweisen gewisse Dateien oder Verzeichnisse nicht zu exportieren, wenn es ein Archiv erzeugt. Falls es Unterverzeichnisse oder Dateien gibt, die Du nicht in Deiner Archivdatei haben willst, aber in Deinem Projektrepository, so kannst Du diese Dateien mit Hilfe des `export-ignore` Attributes festlegen.

Nehmen wir zum Beispiel an, Du hast einige Testdateien in einem `test/` Unterverzeichnis und es macht keinen Sinn, dass diese in einem Tarball Export Deines Projekts enthalten sind. In diesem

Fall kannst Du die folgende Zeile in Deine Git Attribute aufnehmen:

```
test/ export-ignore
```

Wenn Du jetzt `git archive` ausführst, um einen Tarball Deines Projekts zu erstellen, wird das Verzeichnis nicht mit in das Archiv aufgenommen.

export-subst

Auch das einfache Ersetzen von Schlüsselwörtern ist bei einem Archivierungsvorgang möglich. Git erlaubt die Zeichenfolge `$Format:$` mit allen Formatierungsoptionen des Parameters `--pretty=format` in jeglichen Dateien. Viele der Optionen hast Du bereits in Kapitel 2 kennengelernt. Wenn Du zum Beispiel eine Datei namens `LAST_COMMIT` zu Deinem Projekt hinzufügen willst, welche das Datum des letzten Commits enthalten soll, dann kannst Du die folgenden Befehle ausführen:

```
$ echo 'Last commit date: $Format:%cd$' > LAST_COMMIT
$ echo "LAST_COMMIT export-subst" >> .gitattributes
$ git add LAST_COMMIT .gitattributes
$ git commit -am 'adding LAST_COMMIT file for archives'
```

Nach Ausführung des Befehls `git archive`, wird die Datei `LAST_COMMIT` in Deinem Archiv in etwa folgendermaßen aussehen:

```
$ cat LAST_COMMIT
Last commit date: $Format:Tue Apr 21 08:38:48 2009 -0700$
```

Merge Strategien

Die Git Attribute ermöglichen es ebenso verschiedene Regeln für das Zusammenführen bestimmter Dateien innerhalb Deines Projekts festzulegen. Eine besonders nützliche Option ist es, Git so einzustellen, dass es bei bestimmten Dateien kein Zusammenführen von Konfliktstellen versucht, sondern einfach Deine Version übernimmt und die des anderen verwirft.

Dies ist hilfreich, falls ein Zweig Deines Projekts sehr weit vom Hauptzweig abgewichen oder sehr speziell ist, aber Du weiterhin in der Lage sein willst, Änderungen daran zurückzuführen und dabei gewisse Dateien zu ignorieren. Nehmen wir an Du hast eine Konfigurationsdatei einer Datenbank namens `database.xml`, welche sich in zwei Zweigen unterscheidet. Wenn Du jetzt einen Merge von dem anderen Zweig machen möchtest ohne Deine Datenbankdatei unbrauchbar zu machen, dann kannst Du folgendes Attribut einrichten:

```
database.xml merge=ours
```

Wenn Du ein Merge des anderen Zweiges machst, werden für die Datei `database.xml` keine Merge-Konflikte auftreten, sondern es wird folgendes ausgegeben:

```
$ git merge topic
```

Auto-merging database.xml
Merge made by recursive.

In diesem Fall wird die Datei database.xml aus dem anderen Zweig ignoriert und in Deinem Zweig bleibt die Datei im gleichen Zustand wie vor dem Merge.

Git Hooks

Genau wie bei vielen anderen Versionskontrollsystemen gibt es auch bei Git die Möglichkeit eigene Skripte zu starten, wenn bestimmte, wichtige Ereignisse auftreten. Es gibt zwei Gruppen dieser Einschubmethoden: Hooks für den Client und Hooks für den Server. Die Hooks für den Client können bei Ereignissen, wie zum Beispiel einem Commit oder Merge, eingerichtet werden. Die Hooks für den Server können bei Operationen wie den Empfang von hochgeladenen Commits, ausgeführt werden. Es gibt viele Möglichkeiten diese Hooks sinnvoll einzusetzen. Einige davon werde ich hier vorstellen.

Installieren eines Hooks

Sämtliche Hooks werden im `hooks` Unterverzeichnis des Git Verzeichnisses gespeichert. In den meisten Projekten wird das `.git/hooks` sein. Git installiert in dieses Verzeichnis standardmäßig Beispielskripte. Einige davon sind auch ohne Änderung nützlich und sofort einsetzbar. Zusätzlich dokumentieren diese Beispiele die Eingabewerte des jeweiligen Skripts. Alle Beispiele sind Shellskripte, die hier und da ein Paar Zeilen Perl Code enthalten. Prinzipiell sollte aber jedes ausführbare Skript funktionieren, wenn es korrekt benannt wird. Du kannst also die Skriptsprache Deiner Wahl verwenden, z.B. Ruby oder Python. Die Beispieldateien haben die Endung `.sample`, sie müssen also nur noch umbenannt werden.

Um ein Hook-Skript zu aktivieren, speichere eine entsprechend benannte und ausführbare Datei im `hooks` Unterverzeichnis Deines Git Verzeichnisses. Von diesem Augenblick an sollte es ausgeführt werden. Ich werde hier die meisten der wichtigen Hook Dateinamen besprechen.

Hooks für den Client

Es gibt eine Menge Hooks auf Seiten des Clients. Der folgende Abschnitt teilt die Hooks in drei Gruppen auf: Skripte für den Commit Vorgang, Skripte für den Arbeitsablauf mit E-Mails und den Rest der Client Skripte.

Hooks für den Commit Vorgang

Die ersten vier Hooks hängen mit dem Commit Prozess zusammen. Der `pre-commit` Hook wird zuerst ausgeführt, schon bevor Du die Commit Nachricht eingegeben hast. Der Hook wird oft benutzt, um den zu versionierenden Zustand des Arbeitsverzeichnisses zu prüfen, um festzustellen ob etwas vergessen wurde, um sicherzustellen das Tests ausgeführt wurden oder aus irgendeinem anderen Grund, der es nötig macht, den Code vor dem Commit zu inspizieren. Wenn das entsprechende Skript einen Wert ungleich Null zurückgibt, wird der Commit abgebrochen. Auch für die Prüfung, ob Kodierrichtlinien eingehalten wurden oder für eine statische Codeanalyse (z.B. mit `lint` oder einem entsprechenden Programm) kann dieses Skript verwendet werden. Das von Git installierte Beispielskript prüft zum Beispiel, ob am Zeilenende Leerzeichen vorhanden sind. Der Hook kann mit `git commit --no-verify` auch umgangen werden.

Der `prepare-commit-msg` Hook wird ausgeführt, bevor der Editor für die Commit Nachricht geöffnet wird, aber nachdem die Standardnachricht erstellt wurde. Er erlaubt es die Standardnachricht zu modifizieren, bevor der Autor des Commits sie sieht. Dieser Hook akzeptiert diverse Optionen: den Pfad der Datei, die die bisherige Commit Nachricht enthält, den Typ des Commit und den SHA-1 Hash des Commit, falls es sich um ein Korrektur-Commit handelt. Dieser Hook ist üblicherweise nicht sehr nützlich bei normalen Commits; er ist eher für solche Commits gedacht, bei denen die Standardnachricht automatisch generiert wird, wie zum Beispiel vorlagenbasierte Commit Nachrichten, Commits nach einem Merge, Commits, die zusammengeführt werden und Korrektur-Commits. Du kannst diesen Hook mit einer Commit Vorlage kombinieren, um automatisiert Informationen einzufügen.

Der `commit-msg` Hook akzeptiert einen Parameter, der wiederum der Pfad zu der temporären Datei ist, die die momentane Commit Nachricht enthält. Falls dieses Skript nicht Null zurückgibt, so wird der Commit abgebrochen. Damit kannst Du die Gültigkeit des Projektstatus oder die Commit Nachricht prüfen, bevor ein Commit akzeptiert wird. Im letzten Abschnitt dieses Kapitels werde ich beschreiben, wie man diesen Hook benutzt, um sicherzustellen, dass Commit Nachrichten einem bestimmten Muster entsprechen.

Wenn ein Commit komplett abgeschlossen wurde, wird der `post-commit` Hook ausgeführt. Er akzeptiert keine Parameter, aber Du kannst den letzten Commit einfach mit dem Befehl `git log -1 HEAD` abfragen. Dieses Skript wird üblicherweise für das Senden von Benachrichtigungen oder ähnlichem benutzt.

Diese Skripte für den Commit Prozess können für jeden anderen Arbeitsablauf entsprechend angepasst werden. Oft werden sie benutzt um bestimmte Regeln zu erzwingen. Dabei ist es wichtig zu wissen, dass diese Skripte beim Klonen eines Repositorys nicht mit übertragen werden. Du kannst auf Seiten des Servers die Einhaltung von bestimmten Regeln erzwingen indem die hochgeladenen Commits abgelehnt werden, wenn sie diesen Prinzipien nicht entsprechen. Auf dem Client entscheidet aber der Anwender selber, ob er diese Skripte verwendet oder nicht. Dies sind also Skripte, die den Entwicklern helfen sollen, und sie müssen von ihnen erstellt und gepflegt werden. Aber sie können auch von ihnen jederzeit verändert oder umgangen werden.

Hooks für den Arbeitsablauf mit E-Mails

Für einen E-Mail basierten Arbeitsablauf kannst Du drei Hooks auf dem Client einrichten. Sie werden alle bei Ausführung des Befehls `git am` aufgerufen. Wenn Du also diesen Befehl in Deinem normalen Arbeitsablauf nicht verwendest, kann Du guten Gewissens zum nächsten Abschnitt springen. Falls Du aber Patches per E-Mail erhältst, die mit `git format-patch` erstellt wurden, könnten trotzdem einige dieser Skripte nützlich für Dich sein.

Der erste Hook, der ausgeführt wird, ist `applypatch-msg`. Er akzeptiert genau einen Parameter: den Namen der temporären Datei, die die vorgegebene Commit Nachricht enthält. Git bricht den Patch ab, falls dieses Skript nicht Null zurückgibt. Du kannst dies benutzen um sicherzustellen, dass die Commit Nachricht richtig formatiert ist, oder um die Nachricht zu standardisieren, indem das Skript sie direkt editiert.

Der nächste Hook, der beim Anwenden von Patches via `git am` ausgeführt wird, ist `pre-applypatch`. Er benötigt keine Parameter und wird direkt nach Anwendung des Patches ausgeführt. Damit kannst Du den Zustand Deines Projektes noch vor dem eigentlich Commit inspizieren. Du kannst mit diesem Skript Tests ablaufen lassen oder das Arbeitsverzeichnis anderweitig untersuchen. Falls etwas fehlt oder ein Test fehlschlägt, sorgt eine Beenden des Skripts mit einem Wert ungleich Null ebenfalls für das Abbrechen des `git am` Skripts. Es wird also auch kein Commit ausgeführt.

Der letzte Hook, der während der `git am` Operation ausgeführt wird, ist `post-applypatch`. Du kannst dies verwenden, um eine Benutzergruppe oder den Autoren des Patches darüber zu informieren, dass der Patch übernommen wurde. Der eigentliche Patch Vorgang kann mit diesem Skript aber nicht mehr abgebrochen werden.

Weitere Hooks für den Client

Der `pre-rebase` Hook wird ausgeführt, bevor ein Rebase gestartet wird. Durch einen Rückgabewert ungleich Null kann der Rebase Vorgang abgebrochen werden. Du kannst diesen Hook dazu verwenden um beispielsweise zu verhindern, dass auf bereits gepushte Commits ein Rebase durchgeführt wird. Der von Git installierte Beispiel-Hook für `pre-rebase` macht genau das. Allerdings nimmt dieser an, dass der Name des veröffentlichten Branches `,next'` ist. Du musst wahrscheinlich den Namen durch den Deinen stabilen, öffentlichen Branches ersetzen.

Nach jedem erfolgreichen `git-checkout` wird der `post-checkout` Hook ausgeführt. Du kannst ihn verwenden, um Dein Arbeitsverzeichnis für Deine Arbeitsumgebung einzurichten. Das kann das Hinzukopieren großer Binärdateien bedeuten, die Du nicht unter Versionskontrolle stellen möchtest, das automatisierte Generieren von Dokumentation, oder entsprechend ähnliche Aktionen.

Der letzte Hook, den ich vorstellen möchte, ist der `post-merge` Hook. Er wird nach jedem erfolgreichen Aufruf von `merge` ausgeführt. Du kannst diesen benutzen, um Daten in Deinem Arbeitsverzeichnis wiederherzustellen, die Git nicht unter Versionskontrolle stellen kann. Das sind zum Beispiel Berechtigungsdaten. Dieser Hook kann genauso überprüfen, ob Dateien, die nicht unter Versionskontrolle stehen, entsprechend in das Arbeitsverzeichnis kopiert worden sind, wenn sich dieses ändert.

Serverseitige Hooks

Neben den Hooks für den Client, kannst Du als Systemadministrator auch einige wichtige Hooks auf Seiten des Servers installieren. Damit kannst Du nahezu jede Art von Richtlinie für Dein Projekt erzwingen. Die Skripte werden ausgeführt bevor und nachdem ein Push auf den Server durchgeführt wurde. Das Skript für den vorgelagerten Hook kann den Push jederzeit abbrechen indem es einen Wert ungleich Null zurückgibt. Zusätzlich kann dem Client eine Fehlermeldung zurückgeliefert werden. Mit diesen Hooks kannst Du eine beliebig komplexe Push Richtlinie umsetzen.

pre-receive und post-receive

Das erste Skript, das ausgeführt wird, wenn ein Push von einem Client empfangen wird, ist `pre-receive`. Es akzeptiert eine Liste von Referenzen, die über `,stdin‘` hochgeladen werden. Wird es mit einem Wert ungleich Null beendet, so wird keine von ihnen akzeptiert. Du kannst diesen Hook benutzen, um sicherzustellen, dass keine Pushes durchgeführt werden können, welche nicht einem Fast-Forward entsprechen. Ebenso ist es möglich zu Prüfen, ob der Client, die entsprechende Berechtigung zum Erstellen, Löschen oder Aktualisieren eines Branches hat oder ob er die Berechtigung hat, die jeweiligen Dateien zu ändern, die mit dem Push hochgeladen werden.

Der `post-receive` Hook wird aufgerufen, nachdem der komplette Prozess abgeschlossen ist und kann zum Aktualisieren anderer Dienste oder zum Benachrichtigen von Benutzern verwendet werden. Er erwartet die gleichen `,stdin‘` Daten wie `pre-receive`. Beispielsweise können folgende Aktionen ausgeführt werden: Versand von E-Mails an eine vorgefertigte Liste von Personen, Benachrichtigen eines Continuous Integration Servers oder Aktualisieren eines Issue-Tracking-Werkzeugs (Du kannst sogar die Commit Nachrichten parsen um zu prüfen, ob bestimmte Tickets geöffnet, aktualisiert oder geschlossen werden müssen). Das Skript kann allerdings den Push Prozess nicht abbrechen und der Client bleibt bis zum Abschluss des Skripts mit dem Server verbunden. Du solltest deshalb darauf achten, dass Du keinen Vorgang ausführst, der zu viel Zeit in Anspruch nimmt.

update

Das Update Skript ist dem `pre-receive` Skript sehr ähnlich, außer dass es für jeden Branch, den der Client aktualisieren will, ausgeführt wird. Wenn der Benutzer des Clients versucht mehrere Branches zu pushen, wird `pre-receive` nur einmalig aufgerufen, wohingegen das Update Skript für jeden einzelnen Branch ausgeführt wird. Anstatt von dem Stream `stdin` zu lesen, akzeptiert dieses Skript drei Argumente: der Name der Referenz (Branch), die SHA-1 Prüfsumme auf die die Referenz vor dem Push zeigt und die SHA-1 Prüfsumme, die der Anwender versucht zu pushen. Wenn das Update Skript einen Wert ungleich Null zurückgibt, wird der Vorgang nur für diese Referenz abgebrochen, die anderen Referenzen werden weiterhin aktualisiert.

Beispiel für die Durchsetzung von Richtlinien mit Hilfe von Git

In diesem Abschnitt werden wir die gelernten Dinge verwenden um einen Git Arbeitsablauf umzusetzen, der das Format der Commit Nachrichten prüft, nur Pushes zulässt, die einem Fast-Forward entsprechen und der es nur einem beschränkten Kreis von Nutzern ermöglicht einzelne Unterverzeichnisse innerhalb eines Projekts zu modifizieren. Wir werden Client Skripte erstellen, die für den Entwickler prüfen, ob seine Pushes abgelehnt werden würden und wir werden Server Skripte erstellen, die diese Richtlinien um- bzw. durchsetzen.

Ich habe für diese Hooks Ruby verwendet, weil es einerseits meine bevorzugte Skriptsprache ist und andererseits weil der resultierende Code nahezu einem leicht zu lesenden Pseudo-Code entspricht. Auch wenn Du Ruby normalerweise nicht einsetzt, solltest Du deshalb in der Lage sein, meinen Ausführungen zu folgen. Jede andere Sprache sollte aber genauso funktionieren. Alle Beispielskripte, die standardmäßig in Git enthalten sind, sind entweder Perl oder Bash Skripte. Für diese Sprache findest Du also auch genügend Beispiele.

Server Hooks

Die gesamten Skripte für den Server gehören in die Update Datei in Deinem Hooks Verzeichnis. Die Update Datei wird für jeden Branch, der gepusht wird, gestartet und erhält als Parameter die Referenz, die gepusht wird, die alte Revision auf der der Branch stand und die neue Revision, die gepusht wird. Wenn der Push über SSH ausgeführt wird, hat es auch Zugriff auf den Benutzer mit dem der Push durchgeführt wird. Wenn Du den Server so konfiguriert hast, dass jeder über einen einzelnen Benutzer (zum Beispiel „git“) über das Public-Key Verfahren zugreifen kann, dann wäre es sinnvoll diesem Benutzer einen Shell Wrapper einzurichten, der über den öffentlichen Schlüssel die Identität feststellt und damit die Umgebungsvariablen für den jeweiligen Benutzer setzen kann. In dem Beispiel setze ich voraus, dass der Benutzer, der sich verbinden will, in der Umgebungsvariable \$USER enthalten ist. Deshalb sammelt das Update Skript erstmal alle benötigten Informationen:

```
#!/usr/bin/env ruby

$refname = ARGV[0]
$oldrev  = ARGV[1]
$newrev  = ARGV[2]
$user    = ENV['USER']

puts "Enforcing Policies... \n(#{ $refname }) (#{ $oldrev[0,6] }) (#{ $newrev[0,6] })"
```

Ja, ich verwende globale Variablen. Bitte steinigt mich dafür nicht. Auf diese Art und Weise ist es für mich einfacher das Ganze zu demonstrieren.

Format der Commit Nachricht erzwingen

Deine erste Herausforderung wird es sein, sicherzustellen, dass jede Commit Nachricht einem bestimmten Format entspricht. Nehmen wir zum Beispiel an, dass jeder Commit mit einem Ticket

in Deinem Issue-Tracking-System verknüpft sein soll. Deshalb soll jede Commit Nachricht diese Referenz in etwa dem Format „ref: 1234“ enthalten. Dazu musst Du jeden Commit, der gepusht werden soll, prüfen, ob der entsprechende Text enthalten ist. Ist er es nicht, so musst Du das entsprechende Skripte mit einem Rückgabewert ungleich Null beenden, damit der Push abgelehnt beziehungsweise abgebrochen wird.

Eine Liste aller SHA-1 Prüfsummen, die gepusht werden sollen, erhältst Du, indem Du die Werte \$newrev und \$oldrev an das Git Kommando `git rev-list` übergibst (Dieser Befehl gehört zu den Low-Level Funktionen von Git. Im Englischen werden diese auch als „plumbing“ Befehle bezeichnet). Der Befehl entspricht dem `git log` Kommando, gibt aber im Gegensatz zu diesem nur die SHA-1 Prüfsummen und keine weitere Informationen aus. Um eine Liste aller SHA-1 Prüfsummen zwischen zwei Commits zu erhalten, musst Du in etwa folgendes eingeben:

```
$ git rev-list 538c33..d14fc7
d14fc7c847ab946ec39590d87783c69b031bdfb7
9f585da4401b0a3999e84113824d15245c13f0be
234071a1be950e2a8d078e6141f5cd20c1e61ad3
dfa04c9ef3d5197182f13fb5b9b1fb7717d2222a
17716ec0f1ff5c77eff40b7fe912f9f6cfd0e475
```

Du kannst nun durch diese Liste iterieren und für jeden SHA-1 Commit die entsprechende Commit Nachricht anfordern und diese mit Hilfe eines regulären Ausdrucks auf das jeweilige Format prüfen.

Um dies durchführen zu können, benötigst Du das Wissen, wie man an die Commit Nachricht eines einzelnen Commits herankommt. Um die Rohdaten eines Commits zu erhalten, kannst Du eine andere Low-Level Funktion von Git verwenden, nämlich `git cat-file`. Weitere Low-Level Funktionen werde ich in Kapitel 9 näher erläutern, aber hier reicht es erst einmal, wenn Du das Kommando einfach mal ausprobierst:

```
$ 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
```

Um die Commit Nachricht auf Basis der SHA-1 Prüfsumme zu extrahieren, gibt es eine einfache Möglichkeit. Dazu musst Du die Position der ersten leeren Zeile bestimmen. Der gesamte Text nach dieser leeren Zeile entspricht der Commit Nachricht. Mit dem `sed` Befehl funktioniert das unter Unix Systemen ganz einfach:

```
$ git cat-file commit ca82a6 | sed '1,/^\$/d'
changed the version number
```

Damit sollte es Dir auf einfache Art und Weise möglich sein, jede einzelne Commit Nachricht eines Commits, welcher gepusht werden soll, zu prüfen. Du kannst den Push abbrechen, sollte einer der Nachrichten nicht dem gewünschten Format entsprechen. Um ihn abubrechen reicht es,

wenn der Rückgabewert des Skripts ungleich Null ist. Zusammengefasst ergibt sich die folgende Methode:

```
$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
end
check_message_format
```

Wenn Du diesen Auszug in Dein update Skript einbaust, wird jeder Push abgelehnt, der eine Commit Nachricht enthält, die nicht Deinen Regeln entspricht.

Einrichten eines benutzerspezifischen ACL-Systems

Nehmen wir einmal an, dass Du für Deine Projekte ein Mechanismus einrichten willst, der festlegt, wer auf welche Teile Deines Projekts pushen kann. Mit Hilfe einer Zugriffssteuerungsliste (ACL – Access Control List) ist so etwas möglich. Manche Benutzer sollen vollen Zugriff auf das gesamte Repository haben, andere wiederum dürfen nur auf bestimmte Unterverzeichnisse oder spezielle Dateien pushen. Um diese Regeln durchzusetzen werden wir eine Datei mit dem Namen `acl` erstellen und diese im Bare Repository auf Deinem Git Server ablegen. Außerdem werden wir den update Hook so anpassen, dass dieser die erstellten Regeln prüft und bestimmt, ob die jeweilige Aktion vom jeweiligen Benutzer ausgeführt werden darf. Dazu muss der Hook alle Commits, die gepusht werden, prüfen.

Der erste Schritt ist das Erstellen einer ACL. In unserem Beispiel verwenden wir ein Format, welches der CVS ACL sehr ähnlich ist. Jede Zeile ist nach dem selben Format aufgebaut. Das erste Feld einer Zeile enthält entweder `avail` oder `unavail`. Das nächste Feld ist ein kommaseparierte Liste aller User, auf die die Regel zutrifft. Das letzte Feld enthält den Pfad auf welche die Regel zutrifft (ein leeres Feld bedeutet in diesem Fall freien Zugriff). Alle Felder werden durch einen senkrechten Strich (`|`, auch Pipe genannt) getrennt.

In unserem Beispiel gibt es ein paar Administratoren, ein paar Leute, die sich um die Dokumentation im Verzeichnis `doc` kümmern, und einen Entwickler, der nur auf das `lib` und das `test` Verzeichnis zugreifen darf. In diesem Fall sollte die ACL Datei etwa folgendermaßen aussehen:

```
avail|nickh,pjhyett,defunkt,tpw
avail|usinclair,cdickens,ebronte|doc
avail|schacon|lib
```

```
avail|schacon|tests
```

Als erstes müssen wir die Daten in eine Struktur bringen, die wir einfach weiterverwenden können. Um das ganze Beispiel einfach zu halten, erzwingen wir hier nur die `avail` Direktive. Die folgende Funktion erzeugt ein assoziatives Array, in dem der Benutzername als Schlüssel verwendet wird. Der jeweilige Wert ist ein Array von Dateipfaden, auf die der Benutzer Zugriffsrechte besitzt.

```
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
```

Übergibt man der Funktion `get_acl_access_data` die oben overgestellte ACL wird eine Datenstruktur zurückgegeben, die etwa folgendermaßen aussieht:

```
{"defunkt"=>[nil],
 "tpw"=>[nil],
 "nickh"=>[nil],
 "pjhyett"=>[nil],
 "schacon"=>["lib", "tests"],
 "cdickens"=>["doc"],
 "usinclair"=>["doc"],
 "ebronte"=>["doc"]}
```

Nachdem wir auf diese Weise die jeweiligen Zugriffsrechte bestimmt haben, müssen wir noch rausfinden, welche Verzeichnisse bei den gepushten Commits geändert werden. Nur so können wir sicherstellen, dass ein Benutzer die entsprechenden Zugriffsrechte für das jeweilige Verzeichnis hat.

Mit Hilfe des `git log` Befehls und der Option `--name-only` findet man sehr leicht heraus, welche Dateien in einem einzelnen Commit geändert wurden (dies haben wir bereits im Kapitel 2 vorgestellt):

```
$ git log -1 --name-only --pretty=format:'' 9f585d
```

```
README
lib/test.rb
```

Wenn wir nun die Liste der geänderten Dateien, mit der ACL Struktur, die `get_acl_access_data` zurückliefert, vergleichen, kann man ganz einfach herausfinden, ob der Benutzer das Recht hat,

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.index(access_path) == 0) # 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
```

Ich hoffe Du kannst dem Skript leicht folgen. Mit dem Befehl `git rev-list` erhältst Du eine Liste aller Dateien, die gepusht werden. Danach bestimmen wir für jeden Commit, welche Dateien geändert wurden und prüfen, ob der Benutzer auf diese Pfade zugreifen darf. Die Ruby-Zeile `path.index(access_path) == 0`, die vielleicht nicht so einfach zu verstehen ist, liefert `true` zurück, wenn `path` mit der gleichen Zeichenfolge beginnt, wie `access_path`. Das stellt sicher, dass `access_path` nicht nur innerhalb eines erlaubten Pfads als Zeichenfolge enthalten ist, sondern das wirklich der Anfang der Zeichenketten verglichen wird.

Ab jetzt haben alle Benutzer nur für die jeweils freigegebenen Verzeichnisse Zugriffsrechte und es ist sichergestellt, dass keine falsch formatierten Commit-Nachrichten gepusht werden können.

Verweigern von Pushes, welche nicht einem Fast-Forward entsprechen

Nun müssen wir unser System nur noch so einrichten, dass es nur Fast-Forward Push-Operationen zulässt. Man verwendet dafür die `receive.denyDeletes` und `receive.denyNonFastForwards` Konfigurationsparameter. Das gleiche Ergebnis kann man aber auch über einen Hook erreichen und diesen kann man dann so konfigurieren, dass die Regeln nur für bestimmte Benutzer gelten.

Um herauszufinden, ob es sich um einen Fast-Forward handelt, müssen wir prüfen, ob alle Commits, die ausgehend von der letzten Revision erreichbar sind, auch von der neuen Revision aus erreichbar sind. Gibt es einen Commit auf den das nicht zutrifft, so war der Push kein Fast-

Forward und wir verweigern ihn:

```
# enforces fast-forward only pushes
def check_fast_forward
  missed_refs = `git rev-list #{newrev}..#{oldrev}`
  missed_ref_count = missed_refs.split("\n").size
  if missed_ref_count > 0
    puts "[POLICY] Cannot push a non fast-forward reference"
    exit 1
  end
end

check_fast_forward
```

Das war es. Jetzt sollte alles eingerichtet sein. Wenn Du jetzt noch den Befehl `chmod u+x .git/hooks/update` für die Datei ausführst, in die Du den obigen Code eingefügt hast, und dann einen Push ausführst, welcher keinem Fast-Forward entspricht, erhältst Du in etwa folgende Ausgabe:

```
$ 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] Cannot push a non fast-forward reference
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'
```

Lass uns die Ausgabe etwas genauer anschauen, denn sie enthält ein paar interessante Dinge. An Hand der folgenden Zeile erkennst Du, wenn der Hook gestartet wird.

```
Enforcing Policies...
(refs/heads/master) (8338c5) (c5b616)
```

Bitte beachte, dass wir diesen Text beim Start des update-Skripts auf stdout ausgegeben haben. Es ist wichtig zu wissen, dass alles was Dein Skript auf stdout ausgibt, auf den Client übertragen wird und dort ausgegeben wird.

Als nächstes haben wir da noch die folgende Fehlermeldung.

```
[POLICY] Cannot push a non fast-forward reference
error: hooks/update exited with error code 1
error: hook declined to update refs/heads/master
```

Die erste Zeile hast Du innerhalb des Skripts ausgegeben. Die anderen zwei stammen von Git und

teilen Dir mit, dass Dein update-Skript einen Rückgabewert ungleich Null zurückgegeben hat und das der Push verweigert wird. Als Letztes schauen wir uns noch die folgenden Zeilen an:

```
To git@gitserver:project.git
! [remote rejected] master -> master (hook declined)
error: failed to push some refs to 'git@gitserver:project.git'
```

Du siehst dort eine „remote rejected“ Nachricht für jede Referenz, die Dein Hook verweigert hat. Zusätzlich wird dort angegeben, aus welchem Grund der Push verweigert wurde. In diesem Fall hat der Hook den Push verweigert.

Wenn in einem Deiner Commits die Referenz zu dem Issue-Tracking-System fehlt, wird die folgende von Dir festgelegte Fehlermeldung ausgegeben.

```
[POLICY] Your message is not formatted correctly
```

Auch wenn jemand in einem Commit eine Datei geändert hat, die er eigentlich nicht ändern hätte dürfen, und dann versucht diesen Commit zu pushen, wird eine ähnliche Fehlermeldung ausgegeben. Wenn zum Beispiel einer der Jungs und Mädels aus dem Dokumentationsteam versucht einen Commit zu pushen, der irgendeine Änderung im Verzeichnis `lib` enthält, wird diesen die folgende Meldung angezeigt:

```
[POLICY] You do not have access to push to lib/test.rb
```

Von nun an wird Dein Repository immer in einem ordentlichen Zustand sein. Niemand kann Dein Repository durcheinanderbringen oder eine Commit-Nachricht einbringen, die nicht Deinen Vorgaben entspricht. Vorausgesetzt das update-Skript ist vorhanden und ausführbar.

Client Hooks

Allerdings hat unser strenger update-Hook auch einen Nachteil. Du kannst Dich schon mal auf das unvermeidliche Jammern Deiner Mitarbeiter einstellen, wenn diese ihre Commits nicht pushen können, weil sie verweigert werden. Wenn Du deren mit viel Mühe erstellte Arbeit in letzter Minute ablehnst, kann das für die Benutzer extrem frustrierend und verwirrend sein. Dazu kommt noch, dass diese ihre Historie ändern müssen um das ganze zu korrigieren. Und das ist nicht immer etwas für schwache Nerven.

Um dieses Dilemma zu vermeiden, ist es sinnvoll Deinen Mitarbeiter eine Handvoll Client Hooks zur Verfügung zu stellen, die darauf hinweisen, dass der gerade durchgeführte Commit wahrscheinlich vom Server verweigert wird. Auf diese Art und Weise können Deine Mitarbeiter ihre Arbeit noch korrigieren bevor sie sie einchecken. Zu diesem Zeitpunkt sind die Probleme meistens noch einfacher zu lösen. Da die Hooks während des Klonvorgangs nicht mitübertragen werden, musst Du diese auf andere Weise zur Verfügung stellen. Die Benutzer müssen diese Hooks dann auch noch in ihr `.git/hooks`-Verzeichnis kopieren und ausführbar machen. Du kannst die Hooks auch in Deinem Projekt oder in einem separaten Projekt verwalten und verteilen. Allerdings gibt es keine Möglichkeit, dass diese automatisch eingerichtet werden. Dies muss vom Nutzer selber durchgeführt werden.

Als erstes fangen wir damit an, die Commit-Nachrichten beim Einchecken zu prüfen. Damit ist sichergestellt, dass Dein Server die Commits und damit die Änderungen nicht ablehnt, weil sie eine falsch formatierte Commit-Nachricht enthalten. Um dies sicherzustellen, kannst Du den commit-msg-Hook einrichten. Wenn Du in diesem die Nachricht aus der im ersten Argument übergebenen Datei ausliest und mit Deinem Muster vergleichst, kannst Du Git dazu bringen, dass der Commit abgebrochen wird, wenn das Muster nicht passt:

```
#!/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 an der richtigen Stelle (.git/hooks/commit-msg) liegt und ausführbar ist und ein Commit durchgeführt wird, welcher nicht korrekt formatiert ist, wirst Du folgende Ausgabe sehen:

```
$ git commit -am 'test'
[POLICY] Your message is not formatted correctly
```

In diesem Fall wurde der Commit nicht durchgeführt. Wenn die Commit-Nachricht allerdings richtig formatiert ist, erlaubt Git den Commit:

```
$ git commit -am 'test [ref: 132]'
```

[master e05c914] test [ref: 132]
1 files changed, 1 insertions(+), 0 deletions(-)

Als nächstes möchten wir sicherstellen, dass Dateien nur von den Personen geändert werden, die diese auch ändern dürfen. Dazu verwenden wir wieder die Zugriffssteuerungsliste. Wenn Dein lokales .git-Verzeichnis eine Kopie der ACL Datei enthält, die wir vorher erstellt haben, kann das folgende pre-commit-Skript dafür sorgen, dass die Regeln eingehalten werden.

```
#!/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 vorgestellte Skript entspricht nahezu dem Skript, welches wir für den Server erstellt haben. Bis auf zwei wichtige Ausnahmen. Erstens, die ACL Datei befindet sich an einem anderen Speicherort, da das Skript ausgehend von Deinem Arbeitsverzeichnis und nicht ausgehend von Deinem Git-Verzeichnis ausgeführt wird. Aus diesem Grund muss der Pfad zu der ACL Datei von

```
access = get_acl_access_data('acl')
```

nach

```
access = get_acl_access_data('.git/acl')
```

geändert werden.

Der andere wichtige Unterschied besteht darin, auf welche Art und Weise Du eine Liste der geänderten Dateien erhältst. Auf dem Server haben wir die Möglichkeit die Commits zu durchsuchen. Diese Möglichkeit haben wir beim Client nicht, da der Commit noch gar nicht ausgeführt wurde. Deswegen müssen wir die Dateien aus der Staging Area prüfen. Statt

```
files_modified = `git log -1 --name-only --pretty=format:'' #{ref}`
```

musst Du folgende Zeile verwenden:

```
files_modified = `git diff-index --cached --name-only HEAD`
```

Das sind die einzigen Unterschiede, ansonsten funktioniert das Skript auf die gleiche Art und Weise. Ein Nachteil besteht darin, dass davon ausgegangen wird, dass das Skript mit dem gleichen Benutzer ausgeführt wird, wie die Commits auf den Remote gepusht werden. Wenn sich diese unterscheiden, muss die \$user-Variable manuell angepasst werden.

Im letzten Schritt müssen wir noch prüfen, ob versucht wird einen Push durchzuführen, der keinem Fast-Forward entspricht. Das kommt normalerweise aber nicht so oft vor. Dazu muss entweder ein Rebase für Commits durchgeführt werden, die bereits gepusht wurden oder es muss ein lokaler Branch gepusht werden, dessen Name bereits auf dem Remote vorhanden ist und eine andere Historie aufweist.

Da der Server bereits jeden Push ablehnt, der nicht einem Fast-Forward entspricht und alle Push

verweigert werden, die die Historie ändern würden, kann man jetzt nur noch prüfen, ob der Benutzer einen Rebase für bereits gepushte Commits durchführt.

Hier möchte ich ein Beispiel pre-rebase-Skript vorstellen, welches diese Prüfung vornimmt. Es bestimmt eine Liste aller Commits, die neu geschrieben werden und prüft, ob diese bereits auf irgendeinem Remote vorhanden sind. Wenn dies der Fall ist, wird der Rebase abgebrochen:

```
#!/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
end
```

Das Skript verwendet eine Syntax, die wir bereits im Kapitel 6.1 verwendet haben. Man erhält eine Liste aller Commits, die bereits gepusht wurden, wenn folgender Befehl ausgeführt wird:

```
git rev-list ^#{sha}^@ refs/remotes/#{remote_ref}
```

Die SHA^@-Syntax gibt an, dass alle Eltern-Commits miteinbezogen werden sollen. Man sucht auf diese Art und Weise nach allen Commits, die ausgehend vom letzten auf dem Server vorhandenen Commit, erreichbar sind und nach allen Commits, die ausgehend von dem letzten zu pushenden Commit, nicht erreichbar sind.

Diese Methode ist allerdings auch sehr langsam und meistens auch unnötig. Wenn ein Push ohne die Option -f ausgeführt wird und es sich um einen Push handelt, der keinem Fast-Forward entspricht, wird der Server eine Warnung ausgeben und den Push nicht akzeptieren. Allerdings ist diese Methode eine interessante Übung und kann zumindest in der Theorie verhindern, dass ein Rebase durchgeführt wird, der später wieder rückgängig gemacht werden müsste.

Zusammenfassung

In diesem Kapitel hast Du die wichtigsten Möglichkeiten kennengelernt, wie Du Deinen Git Client und Git Server an Deine gewohnte Arbeitsweise und Projekte anpassen kannst. Wir haben eine große Auswahl an Konfigurationsparametern, dateibasierten Attributen und Hooks vorgestellt. Außerdem haben wir einen Server eingerichtet, der dafür sorgt, dass Deine vorgegebenen Richtlinien eingehalten werden. Du solltest jetzt in der Lage sein, Git an nahezu jeden Workflow anzupassen, den Du Dir vorstellen kannst.

Git und andere Versionsverwaltungen

Leider ist die Welt nicht perfekt. Normalerweise kannst Du nicht bei jedem Deiner Projekte sofort auf Git umsteigen. Manchmal musst Du in einem Deiner Projekte irgendeine andere Versionsverwaltung nutzen, ziemlich oft ist das Subversion. Im ersten Teil dieses Kapitels werden wir das bidirektionale Gateway zwischen Git und Subversion kennenlernen: `git svn`.

Manchmal kommst Du an den Zeitpunkt, zu dem Du ein bestehendes Projekt zu Git konvertieren willst. Der zweite Teil dieses Kapitels zeigt Dir, wie Du Dein Projekt zu Git migrieren kannst. Zunächst behandeln wir Subversion, dann Perforce und zum Schluss verwenden wir ein angepasstes Import-Skript, um einen nicht standard-mäßigen Import abzudecken.

Git und Subversion

Gegenwärtig verwenden die meisten Open-Source-Entwicklungsprojekte und eine große Anzahl von Projekten in Unternehmen Subversion, um ihren Quellcode zu verwalten. Es ist die populärste Open-Source-Versionsverwaltung und wird seit fast einem Jahrzehnt eingesetzt. In vielen Bereichen ähnelt es CVS, das vorher der König der Versionsverwaltungen war.

Eines der großartigen Features von Git ist die bi-direktionale Brücke zu Subversion: `git svn`. Dieses Tool ermöglicht es, Git als ganz normalen Client für einen Subversion-Server zu benutzen, sodass Du alle lokalen Features von Git nutzen kannst und Deine Änderungen dann auf einen Subversion-Server pushen kannst, so als ob Du Subversion lokal nutzen würdest. Das bedeutet, dass Du lokale Branches anlegen kannst, mergen, die staging area, rebasing, cherry-picking etc. verwenden, während Deine Kollegen weiterhin auf ihre angestaubte Art und Weise arbeiten. Das ist eine gute Gelegenheit, um Git in einem Unternehmen einzuführen und Deinen Entwickler-Kollegen dabei zu helfen, effizienter zu werden während Du an der Unterstützung arbeitest, die Infrastruktur so umzubauen, dass Git voll unterstützt wird. Die Subversion-Bridge von Git ist quasi die Einstiegsdroge in die Welt der verteilten Versionsverwaltungssysteme (distributed version control systems, DVCS).

git svn

Das Haupt-Kommando in Git für alle Kommandos der Subversion Bridge ist `git svn`. Dieser Befehl wird allen anderen vorangestellt. Er kennt zahlreiche Optionen, daher werden wir jetzt die gebräuchlichsten zusammen anhand von ein paar Beispielen durchspielen.

Es ist wichtig, dass Du im Hinterkopf behältst, dass Du mit dem Befehl `git svn` mit Subversion interagierst, einem System, das nicht ganz so fortschrittlich ist wie Git. Obwohl Du auch dort ganz einfach Branches erstellen und wieder zusammenführen kannst, ist es üblicherweise am einfachsten, wenn Du die History so geradlinig wie möglich gestaltest, indem Du ein rebase für Deine Arbeit durchführst und es vermeidest, zum Beispiel mit einem entfernten Git-Repository zu interagieren.

Du solltest auf keinen Fall Deine History neu schreiben und dann versuchen, die Änderungen zu publizieren. Bitte schicke Deine Änderungen auch nicht zeitgleich dazu an ein anderes Git-Repository, in dem Du mit Deinen Kollegen zusammenarbeitest, die bereits Git nutzen. Subversion kennt nur eine einzige lineare History für das gesamte Repository und da kommt man schnell mal durcheinander. Wenn Du in einem Team arbeitest, in dem manche Deiner Kollegen SVN und andere schon Git nutzen, dann solltest Du sicherstellen, dass Ihr alle einen SVN-Server zur Zusammenarbeit nutzt, das macht Dein Leben deutlich einfacher.

Installation

Um dieses Feature zu demonstrieren, brauchst Du zunächst ein typisches SVN-Repository, in dem Du Schreibzugriff hast. Wenn Du die folgenden Beispiele selbst ausprobieren willst, brauchst Du eine beschreibbare Kopie meines Test-Repositories. Das geht ganz einfach mit einem kleinen Tool

namens svnsync, das mit den letzten Subversion-Versionen (ab Version 1.4) mitgeliefert wird. Für diese Test habe ich ein neues Subversion Repository auf Google Code angelegt, das einen Teil aus dem protobuf-Projekts kopiert, einem Tool, das Datenstrukturen für die Übertragung über ein Netzwerk umwandelt.

Zunächst einmal musst Du ein neues lokales Subversion-Repository anlegen:

```
$ mkdir /tmp/test-svn
$ svnadmin create /tmp/test-svn
```

Anschließend gibst Du allen Usern die Möglichkeit, die revprops zu ändern. Am einfachsten geht das, indem wir ein pre-revprop-change Skript erstellen, das immer 0 zurückgibt:

```
$ cat /tmp/test-svn/hooks/pre-revprop-change
#!/bin/sh
exit 0;
$ chmod +x /tmp/test-svn/hooks/pre-revprop-change
```

Jetzt kannst Du dieses Projekt auf Deinen lokalen Rechner mit einem Aufruf von `svnsync init` synchronisieren. Als Optionen gibst Du das Ziel- und das Quell-Repository an.

```
$ svnsync init file:///tmp/test-svn http://progit-example.googlecode.com/svn/
```

Das richtet die Properties ein, um die Synchronisierung laufen zu lassen. Nun kannst Du den Code klonen:

```
$ svnsync sync file:///tmp/test-svn
Committed revision 1.
Copied properties for revision 1.
Committed revision 2.
Copied properties for revision 2.
Committed revision 3.
...
```

Obwohl diese Operation möglicherweise nur ein paar wenige Minuten in Anspruch nimmt, wird das Kopieren des Quell-Repositories von einem entfernten Repository in ein lokales fast eine Stunde dauern, auch wenn weniger als 100 Commits getätigt wurden. Subversion muss jede Revision einzeln klonen und sie dann in ein anderes Repository schieben – das ist zwar ziemlich ineffizient, aber für uns der einfachste Weg.

Die ersten Schritte (Getting Started)

Jetzt, da wir ein beschreibbares Subversion Repository haben, können wir mit einem typischen Workflow loslegen. Du beginnst mit dem `git svn clone` Kommando, das ein komplettes Subversion-Repository in ein lokales Git-Repository importiert. Denk daran, dass Du `file:///tmp/test-svn` im folgenden Beispiel mit der URL Deines eigenen Subversion-Repositorys ersetzt, wenn Du den Import für ein real existierendes Subversion-Repository durchführen willst.

```

$ git svn clone file:///tmp/test-svn -T trunk -b branches -t tags
Initialized empty Git repository in /Users/schacon/projects/testsvnsync/svn/.git,
r1 = b4e387bc68740b5af56c2a5faf4003ae42bd135c (trunk)
    A      m4/acx_pthread.m4
    A      m4/stl_hash.m4
...
r75 = d1957f3b307922124eec6314e15bcda59e3d9610 (trunk)
Found possible branch point: file:///tmp/test-svn/trunk => \
    file:///tmp/test-svn /branches/my-calc-branch, 75
Found branch parent: (my-calc-branch) d1957f3b307922124eec6314e15bcda59e3d9610
Following parent with do_switch
Successfully followed parent
r76 = 8624824ecc0badd73f40ea2f01fce51894189b01 (my-calc-branch)
Checked out HEAD:
    file:///tmp/test-svn/branches/my-calc-branch r76

```

Hier werden für die angegebene URL eigentlich zwei Befehle ausgeführt, `git svn init` und anschließend `git svn fetch`. Das kann auch eine Weile dauern. Das Testprojekt hat nur etwa 75 Commits und die Codebase ist nicht so groß, daher benötigen wir nur ein paar Minuten. Da Git aber jede Version einzeln auschecken muss, kann es unter Umständen Stunden oder gar Tage dauern, bis die Ausführung des Befehls fertig ist.

Die Parameter `-T trunk -b branches -t tags` teilen Git mit, dass das Subversion-Repository den normalen Konventionen bezüglich Branching und Tagging folgt. Wenn Du Deinen Trunk, Deine Branches oder Deine Tags anders benannt hast, kannst Du diese hier anpassen. Da die Angabe aus dem Beispiel für die meisten Repositories gängig ist, kannst Du das ganze auch mit `-s` abkürzen. Diese Option steht für das Standard-Repository-Layount und umfasst die oben genannten Parameter. Der folgende Befehl ist äquivalent zum zuvor genannten:

```

$ git svn clone file:///tmp/test-svn -s

```

Jetzt solltest Du ein Git-Repository erzeugt haben, in das Deine Branches und Tags übernommen wurden:

```

$ git branch -a
* master
  my-calc-branch
  tags/2.0.2
  tags/release-2.0.1
  tags/release-2.0.2
  tags/release-2.0.2rc1
  trunk

```

An dieser Stelle soll die wichtige Anmerkung nicht fehlen, dass dieses Tool die Namespaces Deiner entfernten Referenzen unterschiedlich behandelt. Wenn Du ein normales Git-Repository klonst, werden alle Branches auf jenem entfernten Server für Dich lokal verfügbar gemacht, zum Beispiel als `origin/[branch]`, der Namespace entspricht dem Namen des entfernten Branches. `git svn` get allerdings davon aus, dass es nicht mehrere entfernte Repositorys gibt und speichert daher seine Referenzen auf die Bereiche entfernter Server ohne Namespaces. Du kannst das Git-Kommando `show-ref` verwenden, um Dir die vollständigen Namen aller Referenzen anzeigen zu

lassen:

```
$ git show-ref
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/heads/master
aee1ecc26318164f355a883f5d99cff0c852d3c4 refs/remotes/my-calc-branch
03d09b0e2aad427e34a6d50ff147128e76c0e0f5 refs/remotes/tags/2.0.2
50d02cc0adc9da4319eeba0900430ba219b9c376 refs/remotes/tags/release-2.0.1
4caaa711a50c77879a91b8b90380060f672745cb refs/remotes/tags/release-2.0.2
1c4cb508144c513ff1214c3488abe66dcb92916f refs/remotes/tags/release-2.0.2rc1
1cbd4904d9982f386d87f88fce1c24ad7c0f0471 refs/remotes/trunk
```

Ein normales Git-Repository sieht dagegen eher so aus:

```
$ git show-ref
83e38c7a0af325a9722f2fdc56b10188806d83a1 refs/heads/master
3e15e38c198baac84223acfc6224bb8b99ff2281 refs/remotes/gitserver/master
0a30dd3b0c795b80212ae723640d4e5d48cabdff refs/remotes/origin/master
25812380387fdd55f916652be4881c6f11600d6f refs/remotes/origin/testing
```

Du hast zwei entfernte Server: einen, der gitserver heißt und einen master-Branch beinhaltet, und einen weiteren, der origin heißt und zwei Branches (master und testing) enthält.

Hast Du bemerkt, dass die entfernten Referenzen, die von `git svn` im Beispiel importiert wurden, nicht als echte Git-Tags importiert wurden, sondern als entfernte Branches? Dein Subversion-Import sieht aus als besäße er einen eigenen Remote-Bereich namens tags und unterhalb davon einzelne Branches.

Änderungen ins Subversion-Repository committen

Mit unserem funktionierenden Repository können wir nun am Projekt arbeiten und unsere Änderungen committen; dabei nutzen wir Git als SVN-Client. Wenn Du eine der Dateien bearbeitest und sie committest, hast Du lokal einen Commit in Git, der auf dem Subversion-Server (noch) nicht vorhanden ist:

```
$ git commit -am 'Adding git-svn instructions to the README'
[master 97031e5] Adding git-svn instructions to the README
1 files changed, 1 insertions(+), 1 deletions(-)
```

Als nächsten Schritt wirst Du Deine Änderungen einchecken wollen. Dein Umgang mit Subversion wird sich dabei verändern -- Du kannst eine Vielzahl an Commits lokal durchführen und dann alle zusammen an den Subversion-Server schicken. Um Deine Änderungen auf den Subversion-Server zu pushen, verwendest Du das `git svn dcommit` Kommando:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M README.txt
Committed r79
M README.txt
r79 = 938b1a547c2cc92033b74d32030e86468294a5c8 (trunk)
No changes between current HEAD and refs/remotes/trunk
```

Resetting to the latest refs/remotes/trunk

Das bündelt alle Commits, die Du auf Basis des Codes im Subversion-Server durchgeführt hast und führt für jede Änderung ein Subversion-Commit durch. Anschließend werden Deine lokalen Git-Commits angepasst und jeder von ihnen bekommt einen eindeutigen Identifier. Das bedeutet, dass alle SHA-1 Checksums Deiner Commits verändert werden. Dies ist einer der Gründe, warum das Arbeiten mit Git-basierten entfernten Versionen Deines Projekts und zeitgleich mit einem Subversion-Server keine gute Idee ist. Wenn Du Dir den letzten Commit ansiehst, wirst Du feststellen, dass eine neue `git-svn-id` hinzugefügt wurde.

```
$ git log -1
commit 938b1a547c2cc92033b74d32030e86468294a5c8
Author: schacon <schacon@4c93b258-373f-11de-be05-5f7a86268029>
Date: Sat May 2 22:06:44 2009 +0000
```

```
Adding git-svn instructions to the README
```

```
git-svn-id: file:///tmp/test-svn/trunk@79 4c93b258-373f-11de-be05-5f7a86268029
```

Die SHA-Checksum Deines ursprünglichen Commits begann mit `97031e5`, jetzt fängt sie mit `938b1a5` an. Wenn Du zugleich auf einen Git- und einen Subversion-Server pushen willst, solltest Du zunächst an den Subversion-Server pushen (`dcommit`), da diese Aktion Deine Commit-Daten verändert.

Änderungen ins lokale Repository übernehmen

Wenn Du mit anderen Entwicklern zusammenarbeitest, wirst Du irgendwann an den Punkt gelangen an dem einer von Euch Änderungen ins Repository pusht und jemand anderes versuchen wird, ebenfalls seine Änderungen zu pushen und damit einen Konflikt erzeugt. Diese Änderung wird solange zurückgewiesen bis Du die Arbeit des anderen Entwicklers mergst. Mit `git svn` sieht das so aus:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
Merge conflict during commit: Your file or directory 'README.txt' is probably \
out-of-date: resource out of date; try updating at /Users/schacon/libexec/git-\
core/git-svn line 482
```

Um diese Situation zu lösen, kannst Du `git svn rebase` laufen lassen. Das zieht alle Änderungen vom Server, die Dir noch fehlen und führt ein rebase Deiner lokalen Kopie durch (auf Basis dessen, was auf dem Server vorhanden ist).

```
$ git svn rebase
M README.txt
r80 = ff829ab914e8775c7c025d741beb3d523ee30bc4 (trunk)
First, rewinding head to replay your work on top of it...
Applying: first user change
```

Jetzt sind alle Deine Arbeiten auf der gleichen Ebene wie die auf dem Subversion-Server und nun

kannst Du erfolgreich ein dcommit absetzen:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M      README.txt
Committed r81
M      README.txt
r81 = 456cbe6337abe49154db70106d1836bc1332deed (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Es ist wichtig, im Hinterkopf zu behalten, dass git svn sich an dieser Stelle anders als Git verhält. Git erwartet von Dir, dass Du upstream-Arbeiten, die Du lokal noch nicht hast, zunächst mergst, bevor Du pushen kannst. Dieses Vorgehen ist bei git svn nur nötig, wenn es Konflikte bei den Änderungen gibt. Wenn jemand anderes eine geänderte Datei gepusht hat und Du eine andere geänderte Datei pushst, wird Dein dcommit problemlos funktionieren:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
M      configure.ac
Committed r84
M      autogen.sh
r83 = 8aa54a74d452f82eee10076ab2584c1fc424853b (trunk)
M      configure.ac
r84 = cdbac939211ccb18aa744e581e46563af5d962d0 (trunk)
W: d2f23b80f67aaaa1f6f5aaef48f3e3263ac71a92 and refs/remotes/trunk differ, \
  using rebase:
:100755 100755 efa5a59965fbbb5b2b0a12890f1b351bb5493c18 \
  015e4c98c482f0fa71e4d5434338014530b37fa6 M      autogen.sh
First, rewinding head to replay your work on top of it...
Nothing to do.
```

Das ist darum wichtig, weil der daraus resultierende Projekt-Status auf keinem der Computer existierte als Du die Änderungen gepusht hast. Wenn die Änderungen nicht zueinander kompatibel sind aber keinen Konflikt ergeben, wirst Du Probleme bekommen, die schwer zu diagnostizieren sind. Das ist der Unterschied zu einem Git-Server — mit Git kannst Du den Zustand Deines Client-Systems komplett testen bevor Du ihn veröffentlichst, während Du bei Subversion nie sicher sein kannst, dass der Zustand direkt vor und direkt nach dem Commit identisch sind.

Du solltest Dieses Kommando auch ausführen um Änderungen vom Subversion-Server zu ziehen, selbst wenn Du noch nicht so weit bist, einen Commit durchzuführen. Du kannst git svn fetch ausführen um die neuen Daten zu besorgen aber git svn rebase zieht die Daten ebenfalls und aktualisiert Deine lokalen Commits.

```
$ git svn rebase
M      generate_descriptor_proto.sh
r82 = bd16df9173e424c6f52c337ab6efa7f7643282f1 (trunk)
First, rewinding head to replay your work on top of it...
Fast-forwarded master to refs/remotes/trunk.
```

Wenn Du git svn rebase ausführt, stellst Du sicher, dass Dein Code immer up-to-

date ist. Du musst Dir aber sicher sein, dass Dein Arbeitsverzeichnis „sauber“ ist, bevor Du den Befehl ausführst. Wenn Du lokale Änderungen hast, musst Du Deine Arbeit vor dem `git svn rebase` entweder stashen oder temporär commiten. Anderenfalls wird die Ausführung des Befehls angehalten, wenn das Rebase in einem Merge-Konflikt enden würde.

Probleme beim Benutzen von Branches

Wenn Du Dich an den Git-Workflow gewöhnt hast, wirst Du höchstwahrscheinlich Zweige für Deine Arbeitspakete anlegen, mit ihnen arbeiten und sie anschließend wieder mergen. Wenn Du mit `git svn` auf einen Subversion-Server pushst, führst Du am besten eine Rebase-Operation in einen einzigen Zweig durch anstatt alle Zweige zusammenzufügen. Der Grund dafür, das Rebase zu bevorzugen, liegt darin, dass Subversion eine lineare Historie hat und mit dem Merge-Operationen nicht wie Git es tut. Daher folgt `git svn` nur dem ersten Elternelement, wenn es Snapshots in Subversion-Commits umwandelt.

Angenommen, Deine Historie sieht wie folgt aus: Du hast einen Zweig mit dem Namen `experiment` angelegt, zwei Commits durchgeführt und diese dann anschließen mit `master` zusammengeführt. Führst Du nun ein `dcommit` durch, sieht die Ausgabe wie folgt aus:

```
$ git svn dcommit
Committing to file:///tmp/test-svn/trunk ...
    M      CHANGES.txt
Committed r85
    M      CHANGES.txt
r85 = 4bfebeec434d156c36f2bcd18f4e3d97dc3269a2 (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
COPYING.txt: locally modified
INSTALL.txt: locally modified
    M      COPYING.txt
    M      INSTALL.txt
Committed r86
    M      INSTALL.txt
    M      COPYING.txt
r86 = 2647f6b86ccfcaad4ec58c520e369ec81f7c283c (trunk)
No changes between current HEAD and refs/remotes/trunk
Resetting to the latest refs/remotes/trunk
```

Das Ausführen von `dcommit` in einem Zweig mit zusammengeführter Historie funktioniert wunderbar, mit der folgenden Ausnahme: wenn Du Deinen Git-Projekthistorie anschaust, wurden die Commits, die Du im `experiment`-Zweig gemacht hast, nicht neu geschrieben — stattdessen tauchen alle diese Änderungen in der SVN-Version des einzelnen Merge-Commits auf.

Wenn nun jemand anderes Deine Arbeit klont, ist alles, was er oder sie sieht, der Merge-Commit mit all Deinen Änderungen insgesamt; die Daten zu den einzelnen Commits (den Ursprung und die Zeit, wann die Commits stattfanden) sehen sie nicht.

Subversion-Zweige

Das Arbeiten mit Zweigen in Subversion ist nicht das gleiche wie mit Zweigen in Git arbeiten; es ist wohl das beste, wenn Du es vermeiden kannst, viel damit zu arbeiten. Dennoch kannst Du mit `git svn` Zweige in Subversion anlegen und Commits darin durchführen.

Einen neuen SVN-Zweig anlegen

Um einen neuen Zweig in Subversion anzulegen, führst Du `git svn branch [branchname]` aus:

```
$ git svn branch opera
Copying file:///tmp/test-svn/trunk at r87 to file:///tmp/test-svn/branches/opera
Found possible branch point: file:///tmp/test-svn/trunk => \
  file:///tmp/test-svn/branches/opera, 87
Found branch parent: (opera) 1f6bfe471083cbca06ac8d4176f7ad4de0d62e5f
Following parent with do_switch
Successfully followed parent
r89 = 9b6fe0b90c5c9adf9165f700897518dbc54a7cbf (opera)
```

Dieser Befehl macht genau das gleiche wie das `svn copy trunk branches/opera`-Kommando in Subversion und arbeitet auf dem Subversion-Server. Wichtig hierbei ist, dass Du mit Deiner Arbeit nicht automatisch in diesen Zweig wechselst: wenn Du zu diesem Zeitpunkt einen Commit durchführst, wird dieser in `trunk` auf dem Server landen, nicht im `opera`-Zweig.

Den aktiven Zweig wechseln

Git findet heraus, in welchem Zweig Deine `dcommits` abgelegt werden, indem es den `tip` jedes Subversion-Zweiges in Deiner Historie untersucht — Du solltest nur einen einzigen haben und es sollte der letzte mit einer `git-svn-id` in der aktuellen Historie des Zweiges sein.

Wenn Du gleichzeitig mit mehr als einem Zweig arbeiten willst, kannst Du lokale Zweige derart anlegen, dass ein `dcommit` für sie in bestimmte Subversion-Zweige durchgeführt wird. Dazu startest Du diese beim importierten Subversion-Commit für diesen Zweig. Wenn Du einen `opera`-Zweig haben willst, an dem Du separat arbeiten kannst, führst Du

```
$ git branch opera remotes/opera
```

aus. Wenn Du jetzt Deinen `opera`-Zweig in `trunk` (Deinen `master`-Zweig) zusammenführen willst, kannst Du das mit einem normalen `git merge` machen. Du solltest allerdings eine aussagekräftige Commit-Beschreibung angeben (mit `-m`) oder sie wird statt etwas Sinnvollem „Merge branch `opera`“ lauten.

Behalte im Hinterkopf, dass dieses Vorgehen kein normaler Git-Merge-Commit ist, auch wenn Du den `git merge`-Befehl verwendest und das Zusammenführen wahrscheinlich wesentlich einfacher ist als es in Subversion gewesen wäre (weil Git automatisch die passende Basis für das Zusammenführen der Zweige für Dich herausfindet). Du musst diese Daten zurück auf den Subversion-Server schieben, der nicht mit Commits umgehen kann, die mehr als ein Elternelement haben; all Deine Änderungen aus einem anderen Zweig werden in diesem einen Commit zusammengepresst, wenn Du die Änderungen hochschiebst. Nachdem Du einen Zweig

mit einem anderen zusammengefügt hast, kannst Du nicht einfach zurückgehen und mit der Arbeit an diesem Zweig weitermachen, wie Du das normalerweise in Git machen würdest. Wenn Du das `dcommit`-Kommando ausführst, löscht es jegliche Information darüber, welcher Zweig hier hineingefügt wurde und als Folge dessen werden künftige `merge-base`-Berechnungen falsche sein — die `dcommit`-Operation lässt Dein `git merge`-Ergebnis so aussehen als ob Du `git merge --squash` verwendet hättest. Unglücklicherweise gibt es kein ideales Mittel, diese Situation zu vermeiden — Subversion kann diese Information einfach nicht speichern, daher wirst Du immer unter seinen beschränkten Möglichkeiten zu leiden haben, so lange Du es als Server verwendest. Um diese Probleme zu vermeiden, solltest Du den lokalen Zweig (in unserem Beispiel `opera`) löschen, nachdem Du ihn mit dem `trunk` zusammengeführt hast.

Subversion Befehle

Das `git svn`-Werkzeug bietet eine ganze Reihe von Befehlen an, die Dir helfen, den Übergang zu Git zu vereinfachen, indem sie einige Funktionen bereitstellen, die jeden ähneln, die Du bereits in Subversion kanntest. Hier sind ein paar Befehle, die Dir solche Funktionen bereitstellen wie das Subversion früher für Dich tat:

Historie im SVN-Stil

Wenn Du an Subversion gewöhnt bist und Deine Historie so sehen möchtest, wie SVN sie ausgeben würde, kannst Du `git svn log` ausführen, um Deine Commit-Historie in der SVN-Formatierung anzusehen:

```
$ git svn log
-----
r87 | schacon | 2009-05-02 16:07:37 -0700 (Sat, 02 May 2009) | 2 lines
autogen change
-----
r86 | schacon | 2009-05-02 16:00:21 -0700 (Sat, 02 May 2009) | 2 lines
Merge branch 'experiment'
-----
r85 | schacon | 2009-05-02 16:00:09 -0700 (Sat, 02 May 2009) | 2 lines
updated the changelog
```

Du solltest zwei wichtige Dinge über `git svn log` wissen. Erstens: es arbeitet offline, im Gegensatz zum echten `svn log`-Befehl, der den Subversion-Server nach den Daten fragt. Zweitens: es zeigt Dir nur Commits an, die auf den Subversion-Server committet wurden. Lokale Git-Commits die Du nicht mit `dcommit` bestätigt hast, werden nicht aufgeführt, genausowenig wie Commits, die andere in der Zwischenzeit auf dem Subversion-Server gemacht haben. Die Ausgabe zeigt Dir eher den letzten bekannten Zustand der Commits auf dem Subversion-Server.

SVN Vermerke (SVN Annotation)

Genauso wie das Kommando `git svn log` den `svn log`-Befehl offline simuliert kannst Du das Pendant zu `svn annotate` mit dem Befehl `git svn blame [FILE]` ausführen. Die Ausgabe sieht so 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 Protoc
2    temporal Buffer compiler (protoc) execute the following:
2    temporal
```

Auch dieser Befehl zeigt die Commits nicht an, die Du lokal getätigt hast, genauso wenig wie jene, die in der Zwischenzeit zum Subversion-Server übertragen wurden.

SVN-Server-Informationen

Die selben Informationen wie bei `svn info` bekommst Du, wenn Du `git svn info` ausführst:

```
$ 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)
```

Genauso wie `blame` und `log` läuft die Ausführung dieses Befehls offline ab und ist nur so aktuell wie zu dem Zeitpunkt, zu dem Du das letzte Mal mit dem Subversion-Server verbunden warst.

Ignorieren, was Subversion ignoriert

Wenn Du ein Subversion-Repository klonst, das irgendwo `svn:ignore` Eigenschaften definiert hat, wirst Du die `.gitignore`-Dateien wahrscheinlich entsprechend setzen, damit Du nicht aus Versehen Dateien committest, bei denen Du das besser nicht tun solltest. `git svn` kennt zwei Befehle, um Dir bei diesem Problem zu helfen. Der erste ist `git svn create-ignore`, der automatisch entsprechende `.gitignore`-Dateien für Dich anlegt, sodass Dein nächster Commit

diese beinhalten kann.

Der zweite Befehl ist `git svn show-ignore`, der Dir diejenigen Zeilen auf stdout ausgibt, die Du in eine `.gitignore`-Datei einfügen musst. So kannst Du die Ausgabe des Befehls direkt in die Ausnahmedatei umleiten:

```
$ git svn show-ignore > .git/info/exclude
```

Auf diese Weise müllst Du Dein Projekt nicht mit `.gitignore`-Dateien zu. Das ist eine gute Wahl wenn Du der einzige Git-Benutzer in Deinem Team bist (alle anderen benutzen Subversion) und Deine Kollegen keine `.gitignore`-Dateien im Projekt haben wollen.

Zusammenfassung von Git-Svn

Die `git svn`-Werkzeuge sind sehr nützlich, wenn Du derzeit (noch) an einen Subversion-Server gebunden bist oder Dich anderweitig in einer Entwicklungsumgebung befindest, die nicht auf einen Subversion-Server verzichten kann. Wie auch immer: Du solltest es als eine Art gestutztes Git ansehen. Anderenfalls läufst Du Gefahr, Dich und Deine Kollegen durcheinander zu bringen. Um dieses Kliff zu umschiffen, solltest Du folgende Richtlinien befolgen:

- Versuch, eine „geradlinige“ Git-Historie zu führen, die keine von `git merge` durchgeführten Merges enthält. Alle Arbeiten, die Du außerhalb des Hauptzweiges durchführst, solltest Du mit `rebase` in ihn aufnehmen anstatt sie zu mit `merge` zusammenzuführen.
- Setz keinen zusätzlichen, externen Git-Server auf, mit dem Du arbeiten möchtest. Du kannst einen aufsetzen um die Klone für neue Entwickler zu beschleunigen, aber Du solltest keine Änderungen dorthin pushen, die keine `git-svn-id`-Einträge haben. Du solltest vielleicht sogar darüber nachdenken, einen `pre-receive`-Hook einzusetzen, der jede Commit-Nachricht auf eine `git-svn-id` prüft und bestimmte Pushes ablehnt, bei denen diese IDs fehlt.

Wenn Du diese Ratschläge befolgst, werden sie die Arbeit mit dem Subversion-Server erträglich machen. Wenn es Dir irgendwie möglich ist, solltest Du trotzdem zu einem echten Git-Server umziehen, denn davon profitiert Dein Team wesentlich deutlicher.

Zu Git umziehen

Wenn Du bereits Quellcode in einer anderen Versionsverwaltung abgelegt hast, aber Dich nun entschieden hast, von nun an Git zu benutzen, musst Du Dein Projekt so oder so umziehen. Für geläufige Systeme bringt Git einige Importer mit. Anschließend lernen wir, wie Du Deinen eigenen, angepassten Importer entwickeln kann. All das wird im folgenden Abschnitt behandelt.

Import

Jetzt ist es an der Zeit zu lernen, wie Du Daten aus zwei der am meisten benutzten (professionellen) SCM-Systeme importieren kannst: Subversion und Perforce. Ein Großteil der Benutzer, die gegenwärtig zu Git umziehen, arbeiten mit einem von diesen beiden Systemen. Außerdem liefert Git für beide jeweils hochprofessionelle Werkzeuge für den Import mit.

Subversion

Wenn Du die letzten Abschnitte über `git svn` gelesen hast, kannst Du diese Anleitungen ganz einfach benutzen um mit `git svn clone` ein Repository zu klonen. Anschließend stoppst den Subversion-Server, führst einen Push auf den neuen Git-Server durch und beginnst ihn zu benutzen. Wenn Du an die Historie ran willst, kannst Du das genauso schnell erreichen als ob Du die Daten aus dem Subversion-Server beziehen würdest (was eine Weile dauern könnte).

Trotzdem ist der Import nicht perfekt. Und weil das ziemlich lange dauern wird, kannst Du es auch gleich richtig machen. Das erste Problem sind die Informationen über die Autoren. In Subversion besitzt jede Person, die mit dem System arbeitet, einen eigenen User-Account, der in den Commit-Informationen aufgezeichnet wird. Die Beispiele in den vorherigen Abschnitten zeigten dafür manchmal schacon an, wie beispielsweise bei der Ausgabe von `blame` und bei `git svn log`. Wenn Du dies näher an die Autoren-Daten von Git binden willst, musst Du Mapping-Informationen für die Subversion-Benutzer und die Git-Autoren anlegen. Erstelle eine Datei mit dem Namen `users.txt`, die folgendes Mapping-Format verwendet:

```
schacon = Scott Chacon <schacon@geemail.com>
selse = Someo Nelse <selse@geemail.com>
```

Um eine Liste der Namen der Autoren bekommen, die SVN benutzen, kannst Du folgendes Kommando ausführen:

```
$ svn log ^/ --xml | grep -P "^<author" | sort -u | \
    perl -pe 's/<author>(.*?)</author>/$1 = /' > users.txt
```

Dies erzeugt Dir die Log-Ausgabe im XML-Format — Du suchst damit nach den Autoren, erzeugst eine Liste ohne doppelte Einträge und wirfst anschließend das überflüssige XML weg. Leite anschließend die Ausgabe in die Datei `users.txt` um, sodass Du jedem Eintrag den entsprechenden Git-Benutzer zuordnen kannst.

Du kannst diese Datei dann `git svn` zur Verfügung stellen um das Tool dabei zu unterstützen, die

Autoreninformationen besser zu mappen. Du kannst `git svn` ebenfalls mitteilen, dass es die Metadaten nicht einbeziehen soll, die Subversion normalerweise importiert, indem Du dem `clone` oder `init` Kommando die `--no-metadata`-Option mitgibst.

```
$ git svn clone http://my-project.googlecode.com/svn/ \
  --authors-file=users.txt --no-metadata -s my_project
```

Jetzt solltest Du einen hübscheren Subversion-Import in Deinem `my_project`-Verzeichnis haben. Statt eines Commit, 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 sie 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 Autoren-Feld sieht jetzt wesentlich besser aus. Auch die `git-svn-id` wird jetzt nicht mehr gebraucht.

Nach dem Import musst Du noch ein wenig aufräumen. Dafür solltest Du all die merkwürdigen Referenzen säubern, die `git svn` angelegt hat. Zuerst verschiebst Du die Tags, damit sie tatsächliche Git-Tags sind statt merkwürdigen Remote-Zweigen. Anschließend verschieben wir den Rest der Zweige, sodass sie lokale Zweige werden.

Um die Tags so zu verschieben, dass sie echte Git-Tags werden, führst Du folgenden Befehl aus:

```
$ git for-each-ref refs/remotes/tags | cut -d / -f 4- | grep -v @ | while read tag
```

Das nimmt die Referenzen, die vorher Remote-Zweige waren, die mit `tag/` begonnen haben und macht aus ihnen echte (leichtgewichtige) Tags.

Als nächstes verschieben wir den Rest der Referenzen aus `refs/remotes` und machen lokale Zweige daraus:

```
$ git for-each-ref refs/remotes | cut -d / -f 3- | grep -v @ | while read branch
```

Jetzt sind alle alten Zweige richtige Git-Zweige geworden und alle alten Tags sind echte Git-Tags. Als letztes müssen wir den neuen Git-Server noch als entfernten Server einrichten und unsere Änderungen zu ihm pushen. Da wir alle Zweige und Tags einbeziehen wollen, kannst Du diesen

Befehl verwenden:

```
$ git remote add origin git@my-git-server:myrepository.git  
$ git push origin --all  
$ git push origin --tags
```

All Deine Zweige und Tags sollten jetzt in Deinem neuen Git-Server in einem schicken, sauberen Import vorhanden sein.

Perforce

Das nächste System, dass wir zum Importieren anschauen werden, ist Perforce. Ein Import-Werkzeug für Perforce wird ebenfalls mit Git mitgeliefert, allerdings nur im `contrib`-Bereich des Quellcodes — es ist nicht wie `git svn` standardmäßig verfügbar. Um es auszuführen, musst Du den Git-Quellcode von `git.kernel.org` herunterladen:

```
$ git clone git://git.kernel.org/pub/scm/git/git.git  
$ cd git/contrib/fast-import
```

In diesem `fast-import` Verzeichnis wirst Du ein ausführbares Python-Skript mit dem Namen `git-p4` finden. Du musst Python sowie das `p4`-Werkzeug auf Deiner Maschine installiert haben, damit der Import klappt. Als Beispiel werden wir das Jam-Projekt aus dem Perforce Public Depot verwenden. Um den Client einzurichten, musst Du die `P4PORT`-Umgebungsvariable exportieren und sie auf das Perforce-Depot einstellen:

```
$ export P4PORT=public.perforce.com:1666
```

Führe den `git-p4 clone`-Befehl aus, um das Jam-Projekt aus dem Perforce-Server zu importieren. Dazu gibst Du den Depot- und Projekt-Pfad sowie den Pfad an, in den Du das Projekt importieren willst:

```
$ git-p4 clone //public/jam/src@all /opt/p4import  
Importing from //public/jam/src@all into /opt/p4import  
Reinitialized existing Git repository in /opt/p4import/.git/  
Import destination: refs/remotes/p4/master  
Importing revision 4409 (100%)
```

Wenn Du zum `/opt/p4import`-Verzeichnis wechselst und dann `git log` ausführst, kannst Du sehen, dass Dein Import funktioniert hat:

```
$ git log -2  
commit 1fd4ec126171790efd2db83548b85b1bbbc07dc2  
Author: Perforce staff <support@perforce.com>  
Date: Thu Aug 19 10:18:45 2004 -0800
```

```
Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into  
the main part of the document.  Built new tar/zip balls.
```

```
Only 16 months later.
```

```
[git-p4: depot-paths = "//public/jam/src/": change = 4409]
```

```
commit ca8870db541a23ed867f38847eda65bf4363371d
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800
```

```
Update derived jamgram.c
```

```
[git-p4: depot-paths = "//public/jam/src/": change = 3108]
```

Du kannst die git-p4-ID bei jedem Commit sehen. Es ist OK, diese ID hier zu behalten, falls Du später noch mal einen Bezug zu der Perforce-Änderung herstellen musst. Falls Du die ID entfernen willst, ist jetzt Zeit dazu — bevor Du mit der Arbeit an dem neuen Repository beginnst. Du kannst `git filter-branch` benutzen um all die IDs zu entfernen:

```
$ git filter-branch --msg-filter '
    sed -e "/^\[git-p4:/d"
Rewrite 1fd4ec126171790efd2db83548b85b1bbbc07dc2 (123/123)
Ref 'refs/heads/master' was rewritten
```

Wenn Du `git log` ausführst, kannst Du alle SHA1-Prüfsummen für jene Commits sehen, die sich geändert haben, aber die git-p4-Zeichenketten sind nicht mehr in den Commit-Nachrichten vorhanden.

```
$ git log -2
commit 10a16d60cffca14d454a15c6164378f4082bc5b0
Author: Perforce staff <support@perforce.com>
Date: Thu Aug 19 10:18:45 2004 -0800
```

```
Drop 'rc3' moniker of jam-2.5.  Folded rc2 and rc3 RELNOTES into
the main part of the document.  Built new tar/zip balls.
```

```
Only 16 months later.
```

```
commit 2b6c6db311dd76c34c66ec1c40a49405e6b527b2
Author: Richard Geiger <rmg@perforce.com>
Date: Tue Apr 22 20:51:34 2003 -0800
```

```
Update derived jamgram.c
```

Dein Import ist jetzt so weit, dass Du ihn auf den neuen Git-Server pushen kannst.

Ein Import-Tool im Eigenbau

Wenn die Versionsverwaltung, die Du verwendest, nicht Subversion oder Perforce ist, solltest Du zunächst einmal online nach einem Import-Tool suchen — gute Import-Tools sind für CVS, Clear Case, Visual Source Sage und sogar für ein Verzeichnis mit Archiven verfügbar. Wenn für Deinen Anwendungsfall keines dieser Werkzeuge passt, Du eine fast schon ausgestorbene Versionsverwaltung verwendest oder Du aus irgendeinem anderen Grund ein angepassteres

Vorgehen brauchst, dann solltest Du `git fast-import` verwenden. Dieser Befehl nimmt einfache Anweisungen von `stdin` entgegen um entsprechende Git-Daten zu schreiben. Es ist viel einfacher Git-Objekte auf diese Art zu erzeugen als die blanken Git-Kommandos zu verwenden oder zu versuchen, die Roh-Objekte zu schreiben (weitere Informationen findest Du in Kapitel 9).

Um das kurz zu zeigen, schreiben wir einen einfachen Importer. Nehmen wir an, Du arbeitest im Verzeichnis `current` und führst ab und an ein Backup durch, indem Du dieses Verzeichnis in ein Backup-Verzeichnis kopierst und ihm einen anderen Namen mit einem Zeitstempel, z. B. `back_YYYY_MM_DD`, verpasst. Diese Struktur wollen wir jetzt in Git importieren. Dein Verzeichnis sieht also so aus:

```
$ ls /opt/import_from
back_2009_01_02
back_2009_01_04
back_2009_01_14
back_2009_02_03
current
```

Damit wir ein Git-Verzeichnis importieren können, müssen wir uns zunächst noch einmal anschauen, wie Git seine Daten speichert. Wie Du Dich vielleicht erinnerst, ist Git im Grundsatz eine verlinkte Liste von Commit-Objekten die auf eine Momentaufnahme (Snapshots) des Inhalts zeigt. Jetzt musst Du nur noch `fast-import` mitteilen, was diese Snapshots sind, welche Commit-Daten zu ihnen zeigen und die Reihenfolge, in die sie gehören. Deine Strategie wird es sein, einen nach dem anderen durch die Snapshots zu gehen und Commits mit dem Inhalt eines jeden Verzeichnisses zu erzeugen und jeden dieser Commits anschließend mit dem vorherigen zu verknüpfen.

Wie wir das schon im Abschnitt „(...)“ in Kapitel 7 getan haben, programmieren wir diese Lösung in Ruby, weil es die Sprache ist, mit der ich normalerweise arbeite und weil sie recht einfach zu lesen ist. Du kannst das Beispiel in so ziemlich jeder Sprache schreiben, mit der Du vertraut bist — es muss nur die passenden Informationen nach `stdout` schreiben.

Zu Beginn musst Du in das Zielverzeichnis wechseln und jedes Unterverzeichnis identifizieren, das ein Snapshot ist, den Du als Commit importieren willst. Du wirst in jedes dieser Unterverzeichnisse wechseln und den entsprechenden Befehl auszugeben um es zu exportieren. Deine Schleife wird etwa so aussehen:

```
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
```

```
end
```

Du führst `print_export` für jedes Verzeichnis aus. Das nimmt das Manifest und die Markierung des letzten Snapshots entgegen und gibt das Manifest und die Markierung des aktuellen zurück; auf diese Weise kannst Du sie passend verlinken. „Mark“ ist der `fast-import`-Begriff für eine ID, die Du einem Commit gibst. Während Du Commits anlegst, verpasst Du jedem einzelnen eine Markierung, die Du benutzen kannst, um von anderen Commits zu ihm zu linken. Daher ist das erste, was Deine `print_export`-Methode macht, eine Markierung aus dem Verzeichnisnamen zu erstellen:

```
mark = convert_dir_to_mark(dir)
```

Das erreichst Du, indem Du ein Array von Verzeichnissen anlegst und den Index-Wert als Markierung verwendest (eine Markierung muss vom Typ `integer` sein). Deine Methode sieht so aus:

```
$marks = []
def convert_dir_to_mark(dir)
  if !$marks.include?(dir)
    $marks << dir
  end
  ($marks.index(dir) + 1).to_s
end
```

Jetzt hast Du eine `integer`-Repräsentation Deines Commits und brauchst nur noch ein Datum für die Commit-Metadaten. Da das Datum im Verzeichnisnamen enthalten ist, parsen wir es einfach daraus. Die nächste Zeile in Deiner `print_export`-Datei lautet

```
date = convert_dir_to_date(dir)
```

wobei `convert_dir_to_date` so definiert ist:

```
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
```

Die Funktion gibt einen Integer-Wert für das Datum eines jeden Verzeichnisses zurück. Das letzte Stück Meta-Information, das wir noch für jeden Commit brauchen, sind die Commit-Daten des Autors, die wir in eine globale Variable packen:

```
$author = 'Scott Chacon <schacon@example.com>'
```

Jetzt können wir damit beginnen, die Commit-Daten für den Importer auszugeben. Die ursprüngliche Information gibt an, dass Du ein Commit-Objekt definierst und zu welchem Branch

es gehört, gefolgt von der Markierung, die Du angelegt hast, der Committer-Information und der Commit-Nachricht und schließlich der ID des vorhergehenden Commits, falls dieser existiert. Der Code sieht wie folgt 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
```

You hardcode the time zone (-0700) because doing so is easy. If you're importing from another system, you must specify the time zone as an offset. The commit message must be expressed in a special format:

```
data (size)\n(contents)
```

The format consists of the word data, the size of the data to be read, a newline, and finally the data. Because you need to use the same format to specify the file contents later, you create a helper method, export_data:

```
def export_data(string)
  print "data #{string.size}\n#{string}"
end
```

Alles, was jetzt noch übrig bleibt, ist das Feststellen des Dateiinhalts eines jeden Snapshots. Das ist einfach, weil Du jeden davon in einem Verzeichnis hast — Du kannst das deleteall-Kommando ausgeben, gefolgt von den Inhalten einer jeden Datei in dem Verzeichnis. Git wird dann jeden Snapshot entsprechend aufzeichnen:

```
puts 'deleteall'
Dir.glob("**/*").each do |file|
  next if !File.file?(file)
  inline_data(file)
end
```

Anmerkung: Da viele Systeme ihre Revisionen als Änderungen von einem Commit zu einem anderen betrachten, kann fast-import auch mit jedem Commit Kommandos entgegennehmen, die angeben, welche Dateien geändert, entfernt oder verändert wurden und was der neue Inhalt ist. Wir könnten die Unterschiede zwischen den Snapshots berechnen und nur diese Daten bereitstellen, aber das zu tun ist komplizierter — Du kannst Git auch einfach alle Daten füttern und es kümmert sich dann darum. Wenn dieses Vorgehen eher zu Deinen Daten passt, schau Dir die fast-import man-Seite an, um Details darüber zu erfahren, wie diese Daten dafür bereitgestellt werden müssen.

Das Format, in dem die Inhalte einer neuen Datei oder in eine geänderte Datei mit ihrem neuen Inhalt angegeben werden, sieht wie folgt aus:

```
M 644 inline path/to/file
```

```
data (size)
(file contents)
```

In diesem Beispiel ist 644 der Datei-Modus (wenn Du ausführbare Dateien hast, wirst Du möglicherweise 755 sehen bzw. einstellen), und inline gibt an, dass Du die Inhalte direkt im Anschluss an diese Zeile aufführen wirst. Deine `inline_data`-Methode sieht so aus:

```
def inline_data(file, code = 'M', mode = '644')
  content = File.read(file)
  puts "#{code} #{mode} inline #{file}"
  export_data(content)
end
```

Du kannst die `export_data`-Methode, die Du vorher definiert hast, wiederverwenden, da wir das auf die gleiche Weise lösen, wie wir die Daten für die Commit-Nachrichten aufbereitet haben.

Das letzte, das wir jetzt noch machen müssen, ist, die gegenwärtige Marke zurückzugeben, damit sie an den nächsten Durchlauf übergeben werden kann.

```
return mark

$stdout.binmode
```

Das ist alles. Wenn Du diese Skript laufen lässt, wirst Du eine Ausgabe erhalten, die etwa wie folgt aussieht:

```
$ ruby import.rb /opt/import_from
commit refs/heads/master
mark :1
committer Scott Chacon <schacon@geemail.com> 1230883200 -0700
data 29
imported from back_2009_01_02deleteall
M 644 inline file.rb
data 12
version two
commit refs/heads/master
mark :2
committer Scott Chacon <schacon@geemail.com> 1231056000 -0700
data 29
imported from back_2009_01_04from :1
deleteall
M 644 inline file.rb
data 14
version three
M 644 inline new.rb
data 16
new version one
(...)
```

Um den Importer zu starten, leite die Ausgabe durch eine Pipe zu `git fast-import` weiter — während Du im Git Verzeichnis befindest, in das Du importieren willst. Zum Anfang kannst Du ein neues Verzeichnis anlegen, darin `git init` laufen lassen und anschließend das Skript starten:


```

$ 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:           18 (          1 duplicates          )
  blobs   :              7 (          1 duplicates          0 deltas)
  trees   :              6 (          0 duplicates          1 deltas)
  commits :              5 (          0 duplicates          0 deltas)
  tags    :              0 (          0 duplicates          0 deltas)
Total branches:           1 (          1 loads          )
  marks:         1024 (          5 unique          )
  atoms:              3
Memory total:           2255 KiB
  pools:           2098 KiB
  objects:           156 KiB
-----
pack_report: getpagesize() =          4096
pack_report: core.packedGitWindowSize =    33554432
pack_report: core.packedGitLimit =    268435456
pack_report: pack_used_ctr =              9
pack_report: pack_mmap_calls =              5
pack_report: pack_open_windows =            1 /            1
pack_report: pack_mapped =          1356 /          1356
-----

```

Wie Du sehen kannst, werden Dir eine Menge Statistiken über den erreichten Erfolg angezeigt. In diesem Beispiel haben wir insgesamt 18 Objekte für 5 Commits in einen Zweig importiert. Jetzt kannst Du `git log` ausführen, um die neue History einzusehen:

```

$ git log -2
commit 10bfe7d22ce15ee25b60a824c8982157ca593d41
Author: Scott Chacon <schacon@example.com>
Date:   Sun May 3 12:57:39 2009 -0700

    imported from current

commit 7e519590de754d079dd73b44d695a42c9d2df452
Author: Scott Chacon <schacon@example.com>
Date:   Tue Feb 3 01:00:00 2009 -0700

    imported from back_2009_02_03

```

Na also — jetzt haben wir ein schickes, sauberes Git-Repository. Dabei ist es wichtig, dass noch nichts ausgecheckt ist — zu Beginn hast Du keinerlei Dateien in Deinem Arbeitsverzeichnis. Um an sie heran zu kommen, musst Du Deinen Zweig dahin zurücksetzen, wo sich `master` gerade befindet:

```

$ ls
$ git reset --hard master
HEAD is now at 10bfe7d imported from current
$ ls

```

`file.rb` `lib`

Du kannst noch eine ganze Menge mehr mit dem `fast-import`-Tool anstellen — es kann verschiedene Modi behandeln, Binärdaten, mehrere Zweige sowie Merges, Tags, Fortschrittsbalken und mehr. Eine Reihe von Beispielen komplexerer Szenarios werden im `contrib/fast-import`-Verzeichnis des Git-Quellcodes bereitgestellt; eines der besseren ist das `git-p4`-Skript, das ich gerade behandelt habe.

Zusammenfassung

Du solltest Dich jetzt ausreichend sicher fühlen im Umgang mit Git und Subversion bzw. mit dem Import von nahezu jedem existierenden Repository in ein neues Git-Repository, ohne Daten zu verlieren. Das nächste Kapitel wird die Interna von Git behandeln, damit Du jedes einzelne Byte bearbeiten kannst, falls es nötig sein sollte.

Git Interna

Möglicherweise hast Du dieses Kapitel hier direkt aufgeschlagen, vorherige Kapitel ausgelassen, oder bist hierher gelangt, nachdem Du alle vorherigen Kapitel gelesen hast. Wie auch immer, in diesem Kapitel werden wir uns mit der internen Funktionsweise und der Implementierung von Git befassen. Meine eigene Erfahrung ist, dass das Lernen dieser Dinge unerlässlich ist, um zu verstehen, wie unheimlich mächtig und flexibel Git ist. Allerdings gibt es Leute, die der Ansicht sind, dass sie auch verwirrend und unnötig komplex für Anfänger sein können. Deshalb habe ich mich entschieden, dieses Kapitel an das Ende des Buches zu verlegen. Du kannst es, ganz wie es Dir beliebt, früher oder später einschieben.

Lass uns also loslegen. Zunächst will ich betonen, falls das bisher noch nicht klargeworden ist, dass Git im seinen Grundzügen ein Dateisystem ist, dessen Inhalte adressierbar sind und auf dem eine VCS-Schnittstelle aufgesetzt ist. Wir werden gleich genauer darauf eingehen, was das heißt.

In den frühen Tagen von Git (d.h. vor Version 1.5) war die Benutzerschnittstelle sehr viel komplexer, weil es die Dateisystem-Eigenschaften stark betonte – im Gegensatz zu einem herkömmlichen VCS. In den letzten Jahren wurde die Benutzerschnittstelle dann stückweise verbessert und verfeinert, sodass es heute so einfach verständlich und einfach zu verwenden ist, wie andere vergleichbare Systeme, die auf dem Markt erhältlich sind. Allerdings besteht scheinbar weiterhin das Vorurteil, das Git-Interface sei komplex und schwer zu erlernen.

Die inhaltsbasiert adressierbare Dateisystem-Ebene ist erstaunlich cool, weshalb ich in diesem Kapitel zuerst darauf eingehen werde. Als Nächstes lernst Du etwas über die Transport-Mechanismen und Repository-Wartungsaufgaben, mit denen Du möglicherweise irgendwann zu tun bekommen wirst.

Plumbing und Porcelain

In diesem Buch haben wir Git besprochen, indem wir vielleicht 30 Befehle wie `checkout`, `branch`, `remote` und so weiter verwendet haben. Weil Git aber ursprünglich als ein Werkzeugkasten konzipiert war und nicht so sehr als ein komplettes, anwenderfreundliches VCS, gibt es auch eine Reihe von Befehlen, die ihre Arbeit auf einer sehr viel grundlegenden Ebene verrichten. Viele davon sind ursprünglich entwickelt worden, um als UNIX-Befehle miteinander verkettet zu werden oder aus Skripten heraus aufgerufen zu werden. Diese Befehle werden oft als „plumbing“-Befehle (Klempner-Befehle) zusammengefasst, während die eher anwenderfreundlichen Befehle „porcelain“ (d.h. Porzellan) genannt werden.

Die ersten acht Kapitel dieses Buches haben sich fast ausschließlich mit „Porcelain“-Befehlen befasst. In diesem Kapitel gehen wir dagegen auf die zugrundeliegenden „Plumbing“-Befehle ein, u.a. weil sie Dir den Zugriff auf die inneren Abläufe von Git ermöglichen, und weil sie dabei helfen, zu verstehen, warum Git tut, was es tut. Diese Befehle sind nicht dazu gedacht, manuell in der Eingabeaufforderung ausgeführt zu werden, sondern sind als Bausteine für Werkzeuge und Skripts gemeint.

Wenn Du `git init` in einem neuen oder bereits bestehenden Verzeichnis ausführst, erzeugt Git das `.git`-Verzeichnis, das fast alle Dateien enthält, die Git intern speichert und ändert. Wenn Du eine Sicherheitskopie Deines Repositorys anlegen oder es duplizieren willst, dann reicht es aus, dieses Verzeichnis zu kopieren. Dieses ganze Kapitel handelt praktisch nur von den Inhalten dieses Verzeichnisses. Schauen wir es uns einmal an:

```
$ ls
HEAD
branches/
config
description
hooks/
index
info/
objects/
refs/
```

Möglicherweise findest Du darin weitere Dateien. Obiges stammt aus einem mit `git init` neu angelegten Repository – das sind also die Standardinhalte. Der Ordner `branches` wird von neueren Git-Versionen nicht mehr verwendet, und die Datei `descriptions` wird nur vom Programm GitWeb benötigt. Du kannst sie also ignorieren. Die Datei `config` enthält Deine projektspezifischen Konfigurationsoptionen, und im Ordner `info` befindet sich eine Datei, die globale Dateiausschlussmuster enthält, die Du nicht in jeder `.gitignore`-Datei neu spezifizieren willst. Das `hooks`-Verzeichnis enthält die client- oder serverseitigen Hook-Skripte, die wir in Kapitel 7 besprochen haben.

Damit bleiben vier wichtige Einträge übrig: die Dateien `HEAD` und `index` und die Verzeichnisse `objects` und `refs`. Dies sind die Kernkomponenten eines Git-Repositorys. Im `objects`-Verzeichnis befinden sich die Inhalte der Datenbank. Das `refs`-Verzeichnis enthält Referenzen

auf Commit-Objekte (Branches) in dieser Datenbank. Die Datei `HEAD` zeigt auf denjenigen Branch, den Du gegenwärtig ausgecheckt hast, und in der Datei `index` verwaltet Git die Informationen der Staging-Area. Wir werden auf diese Elemente jetzt im einzelnen darauf eingehen, sodass Du nachvollziehen kannst, wie Git intern arbeitet.

Git Objekte

Git ist ein Dateisystem, das Inhalte adressieren kann. Prima. Aber was heißt das? Es bedeutet, dass Git im Kern nichts anderes ist als ein einfacher Key-Value-Store („Schlüssel-Wert-Speicher“). Du kannst darin jede Art von Inhalt ablegen und Git wird einen Schlüssel dafür zurückgeben, den Du dann verwenden kannst, um diesen Inhalt jederzeit nachzuschlagen. Um das auszuprobieren, kannst Du den Plumbing-Befehl `hash-object` verwenden. Dieser nimmt Daten entgegen, speichert diese in Deinem `.git`-Verzeichnis und gibt Dir den Schlüssel zurück, unter dem der Inhalt gespeichert wurde. Dazu initialisierst Du als erstes ein neues Git-Repository und verifizierst, dass das `objects`-Verzeichnis leer ist:

```
$ mkdir test
$ cd test
$ git init
Initialized empty Git repository in /tmp/test/.git/
$ find .git/objects
.git/objects
.git/objects/info
.git/objects/pack
$ find .git/objects -type f
$
```

Git hat also ein Verzeichnis `objects` und darin die Unterverzeichnisse `pack` und `info` angelegt, bisher aber keine weiteren Dateien. Als nächsten speichern wir einen Text in dieser Git-Datenbank:

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Die Option `-w` weist `git hash-object` an, das Objekt zu speichern. Andernfalls würde Dir der Befehl lediglich den Schlüssel mitteilen. `--stdin` weist den Befehl an, den Inhalt von der Standardeingabe einzulesen. Wenn Du diese Option weglässt, erwartet der Befehl zusätzlich einen Dateipfad. Die Ausgabe ist ein 40 Zeichen langer SHA-1-Hash, der eine Prüfsumme des gespeicherten Inhaltes darstellt (wir gehen auf diese Hashes gleich noch genauer ein). Git hat jetzt außerdem eine neue Datei in der Datenbank angelegt:

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Jetzt liegt im Verzeichnis `objects` genau eine Datei. Anfangs speichert Git den Inhalt auf diese Art und Weise: In jeweils einer einzelnen Datei werden die Daten gespeichert, referenziert durch den SHA-1-Hash des Inhaltes und seines Headers. Der Name des Unterverzeichnisses `d6` entspricht den ersten zwei Zeichen des SHA-1-Hashes. Die verbleibenden 38 Zeichen werden als Dateiname verwendet.

Mit dem Befehl `git cat-file` kannst Du den jeweiligen Inhalt nachschlagen. Dieser Befehl ist so etwas wie ein Schweizer Taschenmesser, wenn es um Objekte in der Git-Datenbank geht. Wenn Du die Option `-p` übergibst, versucht `git cat-file`, die Art des Inhaltes herauszufinden und

lesbar darzustellen:

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

Auf diese Weise kannst Du also Inhalte zu Git hinzufügen und von dort wieder auslesen. Das klappt auch mit Dateiinhalten. Wenn Du beispielsweise eine einzelne Datei versionieren willst, legst Du dazu die Datei zunächst an und speicherst ihren Inhalt in der Datenbank:

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

Dann kannst Du Änderungen vornehmen und die Datei erneut speichern:

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Die Datenbank enthält jetzt zwei weitere Versionen der Datei neben dem ursprünglich gespeicherten Inhalt:

```
$ find .git/objects -type f
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Jetzt kannst Du die erste Version der Datei mit folgendem Befehl wieder herstellen:

```
$ 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
```

Sich den SHA-1-Hash für jede Version merken zu müssen, ist allerdings nicht sonderlich praktisch. Außerdem speicherst Du nicht den Dateinamen in der Datenbank, sondern lediglich den Inhalt der Datei. Ein solcher Objekttyp wird als „Blob“ bezeichnet. Mit `git cat-file -t` kannst Du Git nach dem Typ eines Objektes in der Datenbank fragen:

```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Baum-Objekte

Als Nächstes schauen wir uns den Objekttyp „Tree“ (Baum) an, der es ermöglicht, Dateinamen zu

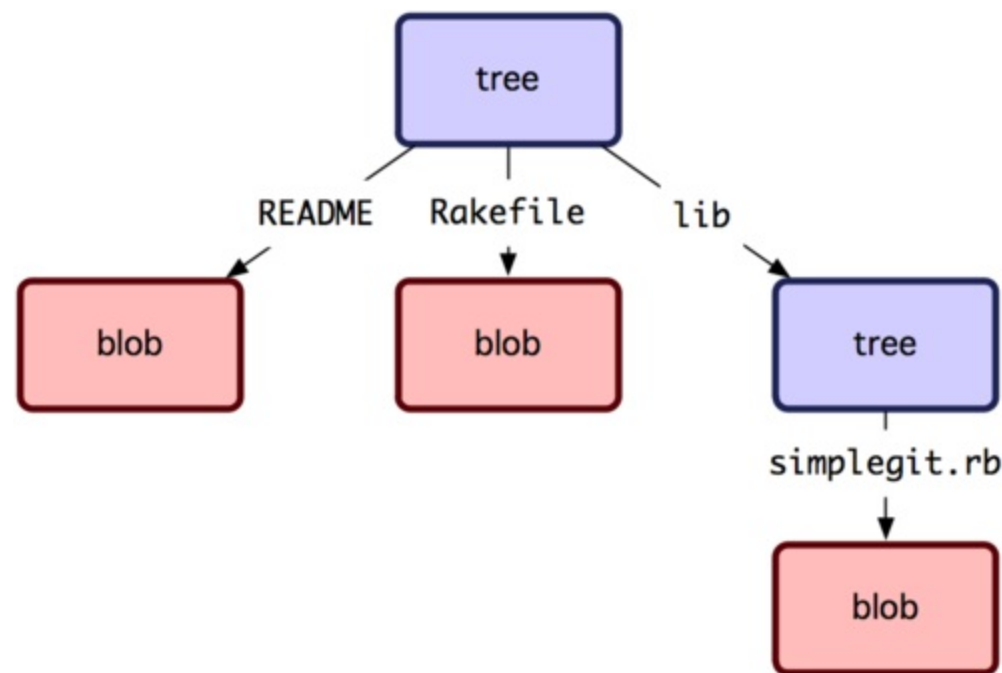
speichern und Dateien zu gruppieren. Git speichert Inhalte in einer ähnlichen Weise wie das UNIX-Dateisystem, allerdings ein bisschen vereinfacht. Sie werden als Tree- und Blob-Objekte abgelegt, wobei die Trees mit UNIX-Verzeichnis-Einträgen korrespondieren und Blobs mehr oder weniger mit den Inode-Einträgen bzw. Datei-Inhalten. Ein einzelnes Baum-Objekt enthält einen oder mehrere Einträge, von denen jeder ein SHA-1-Hash ist, der wiederum einen Blob oder einen Untertree referenziert. Jeder dieser Einträge verfügt außerdem über einen Modus, Typ und Dateinamen. Beispielsweise sieht das aktuelle Tree-Objekt im simplegit-Projekt möglicherweise so aus:

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

Die `master^{tree}`-Syntax spezifiziert, dass wir an dem Tree-Objekt interessiert sind, auf das der letzte Commit des Branches `master` zeigt. Beachte, dass das Unterverzeichnis `lib` nicht auf ein Blob, sondern wiederum auf einen weiteren Baum zeigt.

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```

Git speichert Daten also, konzeptuell gesehen, in etwa wie in Bild 9-1 dargestellt.



Du kannst auch Deine eigenen Tree-Objekte anlegen. Git erzeugt Trees normalerweise, indem es die Inhalte der Staging-Area nimmt und als ein Tree-Objekt speichert. D.h., um ein Tree-Objekt anzulegen, musst Du zunächst einen Index (d.h. eine Staging-Area) aufbauen, indem Du einige Dateien hinzufügst. Um einen einzelnen Eintrag in den Index zu schreiben – z.B. die erste Version der Datei `test.txt` – kannst Du den Plumbing-Befehl `git update-index` verwenden, der eine frühere Version dieser Datei künstlich zu einer neuen Staging-Area hinzufügt. Du musst ihm die Option `--add` übergeben, weil die Datei bisher noch nicht in der Staging-Area enthalten ist (Du hast ja bisher noch überhaupt keine Staging-Area aufgesetzt), und die Option `--cacheinfo`, weil

Du eine Datei hinzufügst, die sich nicht in Deinem Verzeichnis befindet, sondern in der Datenbank. Du gibst außerdem den Modus, SHA-1-Hash und den Dateinamen an:

```
$ git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

In diesem Fall gibst Du als Modus 100644 an, was bedeutet, dass es sich um eine normale Datei handelt. Eine ausführbare Datei wäre dagegen 100755 und ein symbolischer Link 120000. Der Modus entspricht normalen UNIX-Datei-Modi, ist aber weniger flexibel. Die drei genannten Modi sind die einzigen, die in Git für Dateien (Blobs) verwendet werden (es gibt allerdings noch weitere Modi für Verzeichnisse und Submodule).

Jetzt kannst Du den Befehl `git write-tree` verwenden, um die Staging-Area als Tree-Objekt zu schreiben. Dazu brauchst Du die `-w` Option nicht angeben – `git write-tree` schreibt automatisch ein Tree-Objekt für Einträge der Staging-Area, für die es noch keinen Tree gibt:

```
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

Um zu überprüfen, ob es sich wirklich um ein Tree-Objekt handelt:

```
$ git cat-file -t d8329fc1cc938780ffdd9f94e0d364e0ea74f579
tree
```

Jetzt erzeugen wir einen neuen Tree mit der zweiten Version der Datei `test.txt` sowie einer neuen Datei:

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
```

Die Staging-Area enthält jetzt eine neue Version der Datei `test.txt` sowie die neue Datei `new.txt`. Speichern wir diesen Tree (d.h. den gegenwärtigen Status der Staging-Area bzw. des Index als Tree-Objekt) und schauen ihn uns an:

```
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
$ git cat-file -p 0155eb4229851634a0f03eb265b69f5a2d56f341
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```

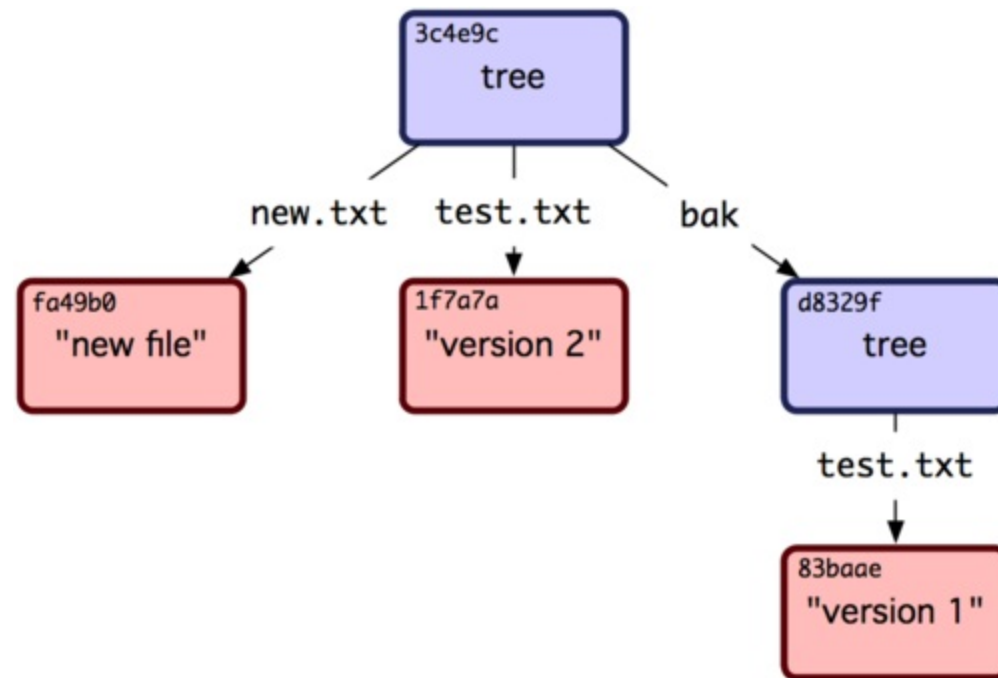
Beachte, dass das Tree-Objekt beide Datei-Einträge hat und dass der SHA-1-Hash der Datei `test.txt` noch derselbe „Version 2“-Hash ist wie zuvor (1f7a7a). Fügen wir jetzt den ersten Tree als ein Unterverzeichnis in diesem hier ein. Du kannst einen Tree mit `git read-tree` in die Staging-Area einlesen. In diesem Fall können wir einen bereits existierenden Tree als einen Untertree zur Staging-Area hinzufügen, indem wir die Option `--prefix` 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 Du ein Arbeitsverzeichnis aus diesem neuen Tree-Objekt auschecken würdest, würdest Du zwei Dateien im Hauptverzeichnis und ein Unterverzeichnis mit dem Namen bak erhalten, in dem sich die erste Version der Datei test.txt befindet. Du kannst Dir die Daten, die Git für diese Strukturen speichert, in etwa wie in Bild 9-2 vorstellen.



Objekte committen

Du hast jetzt drei Trees, die verschiedene Snapshots Deines Projektes spezifizieren, die Du nachverfolgen willst. Das ursprüngliche Problem besteht aber weiterhin: Du musst Dir alle drei SHA-1-Hashwerte merken, um wieder an die Snapshots zu kommen. Ebenso fehlen Dir die Informationen darüber, wer die Snapshots gespeichert hat, wann sie gespeichert wurden und warum. Dies sind die drei Hauptinformationen, die ein Commit-Objekt für uns speichert.

Um ein Commit-Objekt anzulegen, verwendest Du den Befehl `git commit-tree`, spezifizierst den SHA-1-Hash eines einzelnen Trees und welche Commit-Objekte (sofern vorhanden) die direkten Vorgänger sind. Fangen wir damit an, den ersten Tree, den Du angelegt hast, einzuchecken:

```

$ echo 'first commit' | git commit-tree d8329f
fdf4fc3344e67ab068f836878b6c4951e3b15f3d

```

Du kannst Dir dann dieses neue Commit-Objekt mit `cat -file` anschauen:

```

$ 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 besteht aus dem obersten Tree für den Snapshot des Projektes zum gegebenen Zeitpunkt, die Autoren- und ggf. Committer-Information (jeweils entsprechend Deiner `user.name`- und `user.email`-Konfiguration) und dem aktuellen Zeitstempel. Dann folgen eine leere Zeile und die Commit-Nachricht.

Als Nächstes speichern wir die beiden anderen Commit-Objekte und referenzieren jeweils den vorhergehenden Commit:

```
$ 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 zeigt auf einen der drei Snapshot-Trees, die Du zuvor gespeichert hattest. Es mag Dich überraschen, aber Du hast jetzt bereits eine vollständige Git-Historie, die Du mit dem Befehl `git log` inspizieren kannst, indem Du den SHA-1-Hash des letzten Commits angibst:

```
$ git log --stat 1a410e
commit 1a410efbd13591db07496601ebc7a059dd55cfe9
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:15:24 2009 -0700
```

third commit

```
    bak/test.txt |      1 +
    1 files changed, 1 insertions(+), 0 deletions(-)
```

```
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 deletions(-)
```

```
commit fdf4fc3344e67ab068f836878b6c4951e3b15f3d
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri May 22 18:09:34 2009 -0700
```

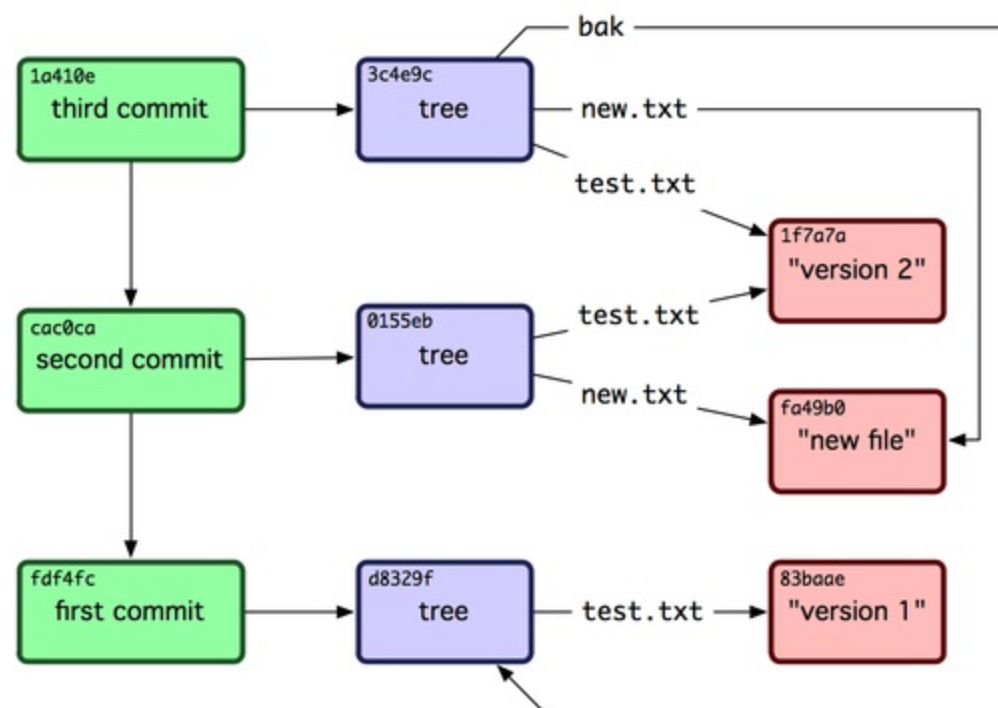
first commit

```
    test.txt |      1 +
    1 files changed, 1 insertions(+), 0 deletions(-)
```

Fantastisch, oder? Du hast jetzt sämtliche Low-Level-Operationen durchgeführt, die eine vollständige Git-Historie aufbauen, ohne aber irgendwelche Frontend-Befehle von Git zu verwenden. Im Wesentlichen ist das derselbe Prozess, der im Hintergrund stattfindet, wenn Du die Befehle `git add` und `git commit` ausführst. Sie speichern Blobs für die Dateien, die Du hinzugefügt oder geändert hast, aktualisieren den Index (d.h. die Staging-Area), speichern Trees und legen Commit Objekte an, die die obersten Trees sowie die Commits referenzieren, die ihnen unmittelbar vorhergingen. Diese drei Hauptobjekte – Blob, Tree und Commit – werden zunächst als separate Dateien im Verzeichnis `.git/objects` gespeichert. Hier ist eine Liste aller Objekte, die sich in unserem Beispiel-Repository jetzt in der Datenbank befinden – jeweils mit einem Kommentar darüber, was sie speichern:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Wenn man all diese internen Zeiger nachverfolgt, erhält man einen Objekt-Graphen wie den folgenden (Bild 9-3).



Objekt-Speicher

Ich habe bereits erwähnt, dass zusammen mit dem jeweiligen Inhalt ein Header gespeichert wird. Schauen wir uns also genauer an, wie genau Git Objekte speichert. Du wirst sehen, wie ein Blob-Objekt – in diesem Fall die Zeichenkette „what is up, doc?“ gespeichert wird. Dazu nutzen wir den

interaktiven Ruby-Modus, den Du mit dem Befehl `irb` starten kannst:

```
$ irb
>> content = "what is up, doc?"
=> "what is up, doc?"
```

Git erzeugt einen Header, der mit dem Objekttyp beginnt, in diesem Fall ist das ein Blob. Dann folgt ein Leerzeichen, die Anzahl der Zeichen des Inhalts und schließlich ein Nullbyte.

```
>> header = "blob #{content.length}\0"
=> "blob 16\000"
```

Git fügt diesen Header mit dem ursprünglichen Inhalt zusammen und kalkuliert aus dem Ergebnis die SHA-1-Prüfsumme. Du kannst einen SHA-1-Hash in Ruby berechnen, indem Du die SHA1-Digest-Bibliothek mit `require` einbindest und dann `Digest::SHA1.hexdigest()` mit der Zeichenkette ausführst:

```
>> store = header + content
=> "blob 16\000what is up, doc?"
>> require 'digest/sha1'
=> true
>> sha1 = Digest::SHA1.hexdigest(store)
=> "bd9dbf5aae1a3862dd1526723246b20206e5fc37"
```

Git komprimiert den neuen Inhalt (d.h. inklusive des Headers) mit `zlib`. In Ruby kannst Du dazu die `zlib`-Bibliothek verwenden, indem Du wiederum zuerst die Bibliothek mit `require` einbindest und dann `Zlib::Deflate.deflate()` mit dem Inhalt aufrufst:

```
>> require 'zlib'
=> true
>> zlib_content = Zlib::Deflate.deflate(store)
=> "\x\234K\312\3110R04c(\317H,Q\310,V(-\320QH\3110\266\a\000_\034\a\235"
```

Schließlich schreibst Du den `zlib`-komprimierten Inhalt in eine Datei auf der Festplatte. Dazu bestimmst Du den Pfad, an den die Datei gespeichert wird (die ersten beiden Zeichen für das Unterverzeichnis und die verbleibenden 38 Zeichen für den Dateinamen). In Ruby kannst Du die Funktion `FileUtils.mkdir_p()` verwenden, um Unterverzeichnisse anzulegen, die noch nicht existieren. Dann öffnest Du die Datei mit `File.open()` und schreibst den komprimierten Inhalt mit `write()` 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
```

Das ist alles – Du hast jetzt ein gültiges Blob-Objekt geschrieben. Git-Objekte werden immer in

dieser Weise gespeichert, lediglich mit verschiedenen Typen, d.h. anstelle der Zeichenkette „blob“ wird der Header mit „commit“ oder „tree“ anfangen. Außerdem sind Commit- und Tree-Inhalte auf eine sehr spezifische Weise formatiert, während Blobs beliebige Inhalte sein können.

Git-Referenzen

Du kannst Befehle wie `git log 1a410e` ausführen, um die Commit-Historie zu inspizieren, aber dazu musst Du Dir jeweils merken, dass `1a410e` der jeweils letzte Commit ist. Um diese SHA-1-Hashes mit einfacheren, verständlichen Namen zu referenzieren, verwendet Git weitere Dateien, in denen die Namen für Hashes gespeichert sind.

Diese Namen werden in Git intern als „references“ oder „refs“ (also Referenz bzw. Verweis) bezeichnet. Du kannst diese Dateien, die SHA-1-Hashes enthalten, im Verzeichnis `.git/refs` finden. In unserem gegenwärtigen Projekt enthält dieses Verzeichnis noch keine Dateien, aber eine simple Verzeichnisstruktur:

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
$
```

Um jetzt eine neue Referenz anzulegen, die Dir dabei hilft, Dich zu erinnern, wo sich Dein letzter Commit befindet, könntest Du, technisch gesehen, Folgendes tun:

```
$ echo "1a410efbd13591db07496601ebc7a059dd55cfe9" > .git/refs/heads/master
```

Jetzt kannst Du diese „head“-Referenz anstelle des SHA-1-Wertes in allen möglichen Git-Befehlen verwenden:

```
$ git log --pretty=oneline master
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Allerdings ist es nicht empfehlenswert, die Referenz-Dateien direkt zu bearbeiten. Git stellt einen sichereren Befehl dafür zur Verfügung, den Befehl `git update-ref`:

```
$ git update-ref refs/heads/master 1a410efbd13591db07496601ebc7a059dd55cfe9
```

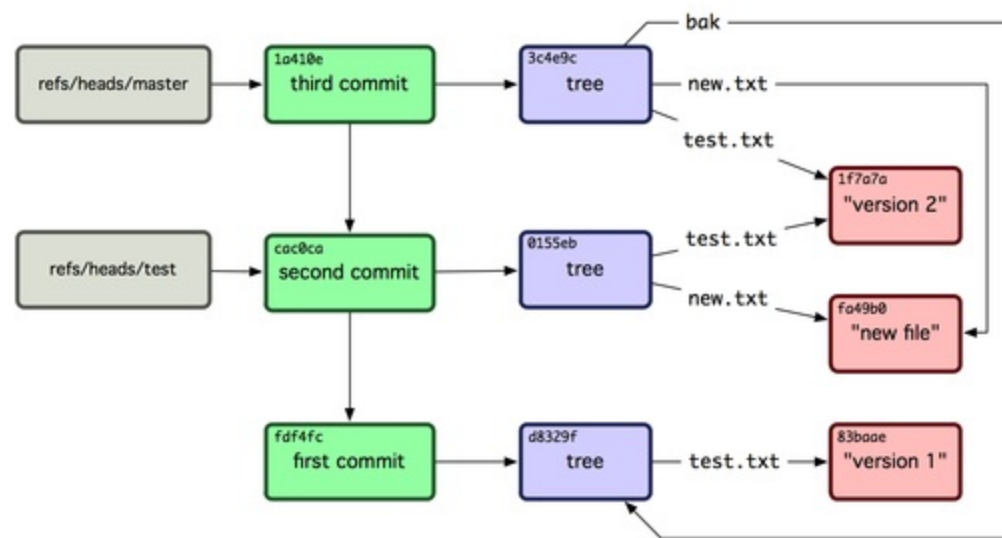
Im Prinzip ist das alles, was einen Branch in Git ausmacht: ein simpler Zeiger oder eine Referenz auf den jeweiligen Head einer Arbeitsreihe. Um einen neuen Branch anzulegen, der vom zweiten Commit aus verzweigt, kannst Du Folgendes tun:

```
$ git update-ref refs/heads/test cac0ca
```

Dein Branch beginnt jetzt beim zweiten Commit:

```
$ git log --pretty=oneline test
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```


Die Git-Datenbank unseres Beispiel-Repositorys ist jetzt wie folgt strukturiert:



Wenn Du Befehle wie `git branch (Branch-Name)` verwendest, führt Git intern im Wesentlichen den Befehl `update-ref` aus, um den SHA-1-Hash des letzten Commits des jeweils gegenwärtigen Branches mit dem gegebenen Namen zu referenzieren.

Der HEAD

Die Frage ist jetzt: Wenn Du `git branch (Branch-Name)` ausführst, woher weiß Git den SHA-1 des letzten Commits? Die Antwort ist: aus der Datei `HEAD`. Diese Datei ist eine symbolische Referenz auf den jeweiligen Branch, auf dem Du Dich gerade befindest. Mit „symbolischer Referenz“ meine ich, dass sie (anders als eine „normale“ Referenz) keinen SHA-1-Hash enthält, sondern stattdessen auf eine andere Referenz zeigt. Wenn Du Dir die Datei ansiehst, findest Du normalerweise etwas wie:

```
$ cat .git/HEAD
ref: refs/heads/master
```

Wenn Du jetzt `git checkout test` ausführst, wird Git die Datei aktualisieren, sodass sie so aussieht:

```
$ cat .git/HEAD
ref: refs/heads/test
```

Wenn Du `git commit` ausführst, erzeugt Git das Commit-Objekt und verwendet als Parent des Commit-Objektes den jeweiligen Wert der Referenz, auf die `HEAD` zeigt.

Du kannst diese Datei manuell bearbeiten, aber wiederum verfügt Git über einen sichereren Befehl, um das zu tun: `git symbolic-ref`. Du kannst den Wert des `HEAD` mit Hilfe des folgenden Befehls lesen:

```
$ git symbolic-ref HEAD
refs/heads/master
```

Und so kannst Du ihn setzen:

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

Du kannst den Befehl allerdings nicht verwenden, um eine Referenz außerhalb von `refs` zu setzen:

```
$ git symbolic-ref HEAD test
fatal: Refusing to point HEAD outside of refs/
```

Tags

Wir haben jetzt Gits drei Haupt-Objekttypen besprochen, aber es gibt noch einen vierten. Das Tag-Objekt ist dem Commit-Objekt sehr ähnlich: es enthält den Autor des Tags, ein Datum, eine Meldung und eine Referenz auf ein anderes Objekt. Der Hauptunterschied besteht darin, dass ein Tag-Objekt auf einen Commit zeigt und nicht auf einen Tree. Ein Tag ist in dieser Hinsicht also ähnlich einem Branch, aber er bewegt sich nie, sondern zeigt immer auf denselben Commit und gibt ihm damit einen netteren Namen.

Wie wir schon in Kapitel 2 besprochen haben, gibt es zwei Typen von Tags: „annotierte“ und „einfache“. Du kannst einen einfachen Tag wie folgt anlegen:

```
$ git update-ref refs/tags/v1.0 cac0cab538b970a37ea1e769cbbde608743bc96d
```

Das ist alles, woraus ein einfacher Tag besteht: einem Branch, der sich nie bewegt. Ein annotierter Tag ist komplexer. Wenn Du einen annotierten Tag anlegst, erzeugt Git ein Tag-Objekt und speichert eine Referenz, die darauf zeigt, statt direkt auf den Commit zu zeigen. Du kannst das sehen, wenn Du einen annotierten Tag anlegst (-a bewirkt, dass wir einen annotierten Tag erhalten):

```
$ git tag -a v1.1 1a410efbd13591db07496601ebc7a059dd55cfe9 -m 'test tag'
```

Das erzeugt den folgenden Objekt SHA-1-Hash:

```
$ cat .git/refs/tags/v1.1
9585191f37f7b0fb9444f35a9bf50de191beadc2
```

Jetzt wendest Du den Befehl `git cat-file` auf diesen SHA-1-Hash an:

```
$ 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
```

Beachte, dass der Wert `object` auf den SHA-1 des Commits zeigt, den Du getaggt hast. Weiterhin muss der Eintrag nicht auf einen Commit zeigen. In Git kann man jedes beliebige Objekt taggen. Im Git-Quellcode befindet sich beispielsweise der öffentliche GPG-Schlüssel des Projektbetreibers als ein Blob-Objekt, sowie ein Tag, der darauf zeigt. Du kannst Dir den öffentlichen Schlüssel anzeigen lassen, indem du den folgenden Befehl im Git-Quellcode-Repository ausführst:

```
$ git cat-file blob junio-gpg-pub
```

Der Linux-Kernel hat auch ein Tag-Objekt, das nicht auf einen Commit zeigt – der erste erstellte Tag zeigt auf den anfänglichen Tree des Quelltext-Imports.

Externe Referenzen

Der dritte Referenztyp ist die externe Referenz („remote reference“). Wenn Du einen externen Server („remote“) definierst und dorthin pushst, merkt sich Git den zuletzt gepushten Commit für jeden Branch im `refs/remotes` Verzeichnis. Beispielsweise fügst Du einen externen Server `origin` hinzu und pushst Deinen Branch `master` dorthin:

```
$ 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
```

Dann kannst Du herausfinden, in welchem Zustand sich der Branch `master` auf dem Server `origin` zuletzt befand (d.h. als Du das letzte Mal mit ihm kommuniziert hast), indem Du Dir die Datei `refs/remotes/origin/master` anschaust:

```
$ cat .git/refs/remotes/origin/master
ca82a6dff817ec66f44342007202690a93763949
```

Externe Referenzen unterscheiden sich von Branches (`refs/heads`) hauptsächlich dadurch, dass man sie nicht auschecken kann. Git verwendet sie quasi als Lesezeichen für den zuletzt bekannten Status, in dem sich die Branches auf externen Servern jeweils befanden.

Pack-Dateien

Kommen wir noch einmal auf die Objekt-Datenbank zurück, die Du für Dein Test-Repository angelegt hast. Im Moment müsstest Du 11 Objekte haben: 4 Blobs, 3 Trees, 3 Commits und 1 Tag:

```
$ find .git/objects -type f
.git/objects/01/55eb4229851634a0f03eb265b69f5a2d56f341 # tree 2
.git/objects/1a/410efbd13591db07496601ebc7a059dd55cfe9 # commit 3
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a # test.txt v2
.git/objects/3c/4e9cd789d88d8d89c1073707c3585e41b0e614 # tree 3
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30 # test.txt v1
.git/objects/95/85191f37f7b0fb9444f35a9bf50de191beadc2 # tag
.git/objects/ca/c0cab538b970a37ea1e769cbbde608743bc96d # commit 2
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4 # 'test content'
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579 # tree 1
.git/objects/fa/49b077972391ad58037050f2a75f74e3671e92 # new.txt
.git/objects/fd/f4fc3344e67ab068f836878b6c4951e3b15f3d # commit 1
```

Git komprimiert die Inhalte dieser Dateien mit zlib und Du hast nicht sonderlich viele davon, sodass die Gesamtgröße der Dateien gerade mal 925 Bytes beträgt. Wir wollen ein anderes interessantes Feature von Git demonstrieren, und dazu müssen wir eine größere Datei hinzufügen, z.B. die `repo.rb` Datei aus der Grit-Bibliothek, die Du schon verwendet hast. Diese Datei ist eine etwa 12K große Quelltext-Datei:

```
$ curl https://raw.githubusercontent.com/mojombo/grit/master/lib/grit/repo.rb > repo.rb
$ git add repo.rb
$ git commit -m 'added repo.rb'
[master 484a592] added repo.rb
 3 files changed, 459 insertions(+), 2 deletions(-)
 delete mode 100644 bak/test.txt
 create mode 100644 repo.rb
 rewrite test.txt (100%)
```

Wenn Du Dir den resultierenden Tree anschaust, findest Du den SHA-1-Hash, den die Datei `repo.rb` für das Blob-Objekt erhalten hat:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e     repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b     test.txt
```

Jetzt kannst Du mit `git cat-file` sehen, wie groß das Objekt ist:

```
$ du -b .git/objects/9b/c1dc421dcd51b4ac296e3e5b6e2a99cf44391e
4102      .git/objects/9b/c1dc421dcd51b4ac296e3e5b6e2a99cf44391e
```

Als Nächstes ändern wir die Datei ein bisschen, um zu sehen, was passiert:

```
$ echo '# testing' >> repo.rb
$ git commit -am 'modified repo a bit'
[master ab1afef] modified repo a bit
```

```
1 files changed, 1 insertions(+), 0 deletions(-)
```

Wenn Du jetzt den Tree anschaust, der durch den Commit angelegt wurde, findest Du etwas Interessantes:

```
$ git cat-file -p master^{tree}
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 05408d195263d853f09dca71d55116663690c27c      repo.rb
100644 blob e3f094f522629ae358806b17daf78246c27c007b      test.txt
```

Das Blob ist ein anderes, d.h. obwohl Du lediglich eine einzige Zeile an das Ende einer 400 Zeilen langen Datei angehängt hast, speichert Git den Inhalt jetzt als ein ganz neues Objekt:

```
$ du -b .git/objects/05/408d195263d853f09dca71d55116663690c27c
4109      .git/objects/05/408d195263d853f09dca71d55116663690c27c
```

Du hast jetzt zwei fast identische 12K große Objekte auf Deiner Festplatte. Wäre es nicht besser, wenn Git nur das erste vollständig und das zweite lediglich als ein Delta zwischen dem ersten und dem zweiten speichern würde?

Tatsächlich kann Git das. Das ursprüngliche Format, in dem Git Objekte in der Datenbank speichert, wird als „freies Objekt-Format“ („loose object format“) bezeichnet. Hin und wieder packt Git allerdings eine Reihe solcher Objekte in eine einzige binäre Datei zusammen, um Platz zu sparen und effizienter zu arbeiten. Eine solche Datei wird als „packfile“ bezeichnet. Git tut das immer dann, wenn zu viele freie Objekte vorhanden sind, wenn Du den Befehl `git gc` manuell ausführst oder wenn Du auf einen externen Server pushst. Schauen wir uns also an, was passiert, wenn wir manuell `git gc` ausführen:

```
$ git gc
Counting objects: 17, done.
Delta compression using 2 threads.
Compressing objects: 100% (13/13), done.
Writing objects: 100% (17/17), done.
Total 17 (delta 1), reused 10 (delta 0)
```

Wenn Du das Objekt-Verzeichnis anschaust, siehst Du, dass die meisten Objekte jetzt fehlen und dass stattdessen zwei neue Objekte aufgetaucht sind:

```
$ find .git/objects -type f
.git/objects/71/08f7ecb345ee9d0084193f147cdad4d2998293
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/info/packs
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack
```

Die verbleibenden Objekte sind diejenigen Blobs, die nicht von irgendeinem Commit referenziert werden – in diesem Fall sind das die Beispielblobs „what is up, doc?“ und „test content“, die wir zuvor gespeichert hatten. Weil wir sie nie zu irgendeinem Commit hinzugefügt haben, werden sie als „dangling“ (wörtlich: herumbaumelnd) betrachtet und nicht im Packfile zusammengepackt.

Die beiden neuen Dateien sind das Packfile und ein Index. Das Packfile ist eine einzelne Datei, die die Inhalte all der Dateien umfasst, die jetzt aus dem Dateisystem entfernt worden sind. Der Index ist eine Datei, die auf Positionen von Objekten im Packfile zeigt, sodass Git schneller nach einem bestimmten Objekt suchen kann. Obwohl diese Objekte auf der Festplatte insgesamt 12K groß waren, bevor Du `git gc` ausgeführt hast, ist das Packfile jetzt nur 6K groß. D.h., Du hast den Platzverbrauch dadurch um die Hälfte reduziert. Toll, oder?

Wie stellt Git das genau an? Wenn Git Objekte zusammen packt, sucht es nach Dateien, die ähnlich benannt und ähnlich groß sind, und speichert dann lediglich Deltas von einer Version zur nächsten. Du kannst ein Packfile inspizieren, um zu sehen, wie Git die Objekte gepackt hat. Der Plumbing Befehl `git verify-pack` macht das möglich:

```
$ git verify-pack -v \
.git/objects/pack/pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.idx
0155eb4229851634a0f03eb265b69f5a2d56f341 tree      71 76 5400
05408d195263d853f09dca71d55116663690c27c blob      12908 3478 874
09f01cea547666f58d6a8d809583841a7c6f0130 tree      106 107 5086
1a410efbd13591db07496601ebc7a059dd55cfe9 commit    225 151 322
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a blob      10 19 5381
3c4e9cd789d88d8d89c1073707c3585e41b0e614 tree      101 105 5211
484a59275031909e19aadb7c92262719cfcdf19a commit    226 153 169
83baae61804e65cc73a7201a7252750c76066a30 blob      10 19 5362
9585191f37f7b0fb9444f35a9bf50de191beadc2 tag        136 127 5476
9bc1dc421dcd51b4ac296e3e5b6e2a99cf44391e blob       7 18 5193 1 \
05408d195263d853f09dca71d55116663690c27c
ab1afef80fac8e34258ff41fc1b867c702daa24b commit    232 157 12
cac0cab538b970a37ea1e769cbbde608743bc96d commit    226 154 473
d8329fc1cc938780ffdd9f94e0d364e0ea74f579 tree      36 46 5316
e3f094f522629ae358806b17daf78246c27c007b blob      1486 734 4352
f8f51d7d8a1760462eca26eebafde32087499533 tree      106 107 749
fa49b077972391ad58037050f2a75f74e3671e92 blob       9 18 856
fdf4fc3344e67ab068f836878b6c4951e3b15f3d commit    177 122 627
chain length = 1: 1 object
pack-7a16e4488ae40c7d2bc56ea2bd43e25212a66c45.pack: ok
```

Du erinnerst dich, dass der Blob `9bc1d` die erste Version der `repo.rb`-Datei ist. Dieser Blob referenziert jetzt den Blob `05408`, der die zweite Version der Datei ist. Die dritte Spalte der Ausgabe ist die Größe des Objektes im Packfile. Wir können also sehen, dass `05408` 12K in Anspruch nimmt, `9bc1d` aber nur 7 Bytes. Das bedeutet also, dass die zweite Version diejenige ist, die vollständig, während die ursprüngliche, erste Version als Delta gespeichert wird! Der Grund dafür ist, dass Du höchstwahrscheinlich einen schnelleren Zugriff auf die jeweils neuesten Dateien brauchst.

Außerdem ist toll, dass ein Repository jederzeit neu gepackt werden kann. Git macht das gelegentlich automatisch, um weniger Platz für die Datenbank zu verbrauchen. Du kannst sie aber auch jederzeit manuell mit `git gc` packen.

Die Refspec

In diesem Buch haben wir bisher einfache Zuweisungen von externen Branches auf lokale Referenzen verwendet. Sie können aber auch durchaus komplex sein. Nehmen wir an, Du hast ein Remote-Repository wie folgt definiert:

```
$ git remote add origin git@github.com:schacon/simplegit-progit.git
```

Das fügt eine Sektion in Deine `.git/config`-Datei hinzu, die Deinen lokalen Namen des externen Repositorys (`origin`), dessen URL und die Refspec spezifiziert, mit der neue Daten heruntergeladen werden.

```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/*:refs/remotes/origin/*
```

Das Format der Refspec besteht aus einem optionalen `+` gefolgt von `<Quelle>:<Ziel>`, wobei `<Quelle>` ein Muster für Referenzen auf der Remote-Seite ist, und `<Ziel>` angibt, wohin diese Referenzen lokal geschrieben werden. Das `+` weist Git an, die Referenz zu mergen, wenn sie nicht mit einem Fast-forward aktualisiert werden kann.

Der Standard, der von `git remote add` automatisch eingerichtet wird, besteht darin, dass Git automatisch alle Referenzen unter `refs/heads/` vom Server holt und sie lokal nach `refs/remotes/origin` speichert. D.h., wenn es auf dem Server einen Branch `master` gibt, kannst Du auf das Log dieses Branches wie folgt zugreifen:

```
$ git log origin/master
$ git log remotes/origin/master
$ git log refs/remotes/origin/master
```

Diese Varianten sind allesamt äquivalent, weil Git sie jeweils zu `refs/remotes/origin/master` vervollständigt.

Wenn Du stattdessen willst, dass Git jeweils nur den Branch `master` herunterlädt und andere Branches auf dem Server ignoriert, kannst Du die `fetch`-Zeile wie folgt ändern:

```
fetch = +refs/heads/master:refs/remotes/origin/master
```

Dies ist allerdings lediglich der Standardwert der Refspec und Du kannst ihn auf der Kommandozeile jederzeit überschreiben. Um zum Beispiel nur den Branch `master` vom Server lokal als `origin/mymaster` zu speichern, kannst Du Folgendes ausführen:

```
$ git fetch origin master:refs/remotes/origin/mymaster
```

Du kannst auch mehrere Refspecs gleichzeitig spezifizieren. Um mehrere Branches zu holen kannst du folgenden Befehl in die Kommandozeile eingeben:

```
$ 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 ein Pull zurückgewiesen, weil der Branch nicht mit einem simplen Fast-forward aktualisiert werden konnte. Du kannst einen Merge erzwingen, indem Du der Refspec ein + voranstellst.

Du kannst außerdem natürlich auch mehrere Refspecs in Deiner Konfiguration spezifizieren. Wenn Du z.B. immer die Branches master und experiment holen willst, fügst Du die folgenden Zeilen hinzu:

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/experiment:refs/remotes/origin/experiment
```

Du kannst keine partiellen Glob-Muster verwenden, d.h. Folgendes wäre ungültig:

```
fetch = +refs/heads/qa*:refs/remotes/origin/qa*
```

Allerdings kannst Du Namensräume verwenden, um etwas Ähnliches zu erreichen. Nehmen wir an, Du hast ein QA-Team, das regelmäßig verschiedene Branches pusht, und Du willst nun den Branch master und sämtliche Branches des QA-Teams, aber keine anderen Branches haben. Dann kannst Du eine Config-Sektion wie die folgende verwenden:

```
[remote "origin"]
    url = git@github.com:schacon/simplegit-progit.git
    fetch = +refs/heads/master:refs/remotes/origin/master
    fetch = +refs/heads/qa/*:refs/remotes/origin/qa/*
```

In einem großen Team mit einem komplexen Workflow, in dem ein QA-Team, Entwickler und ein Integrations-Team jeweils eigene Branches pushen, kann man auf diese Weise Branches einfach in Namensräume einteilen.

Refspecs pushen

Wie aber legt das QA-Team die Branches im qa/ Namensraum ab? Das geht, indem man mit einer Refspec pusht.

Wenn das QA-Team seinen Branch master in einem externen Repository als qa/master speichern will, kann es das wie folgt tun:

```
$ git push origin master:refs/heads/qa/master
```

Um Git so zu konfigurieren, dass diese Refspec jedes Mal automatisch für git push origin verwendet wird, kann man den push Wert in der Config-Datei setzen:


```
[remote "origin"]
  url = git@github.com:schacon/simplegit-progit.git
  fetch = +refs/heads/*:refs/remotes/origin/*
  push = refs/heads/master:refs/heads/qa/master
```

Auf diese Weise wird `git push origin` den lokalen Branch `master` als `qa/master` auf dem Server `origin` speichern.

Referenzen löschen

Man kann Refspecs außerdem verwenden, um Referenzen aus einem externen Repository zu löschen:

```
$ git push origin :topic
```

Das Refspec Format ist `<Quelle>:<Ziel>`. Wenn man den Teil `<Quelle>` weglässt, dann heißt das im obigen Beispiel, dass man den Branch `topic` auf dem Server `origin` auf „nichts“ setzt, d.h. also löscht.

Transfer-Protokolle

Git kann Daten zwischen zwei Repositorys im Wesentlichen auf zwei Arten transportieren: über HTTP und über sogenannte smarte Protokolle, die mit `file://`, `ssh://` und `git://` verwendet werden. Die folgende Sektion gibt einen kurzen Überblick über diese Protokolle und wie sie funktionieren.

Das dumme Protokoll

Das HTTP-Transfer-Protokoll von Git wird oft auch als „dummes“ Protokoll bezeichnet, weil es auf der Server-Seite keinen Git-spezifischen Code benötigt. Der `fetch`-Prozess besteht aus einer Reihe von GET-Requests, für die der Client Vorannahmen über das Layout des Git-Repositorys auf dem Server machen kann. Schauen wir uns den `http-fetch`-Prozess der Bibliothek `simplegit` an:

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

Der Befehl lädt zunächst die Datei `info/refs` herunter. Diese Datei wird vom Befehl `update-server-info` geschrieben, den man als einen `post-receive`-Hook einrichten muss, damit das HTTP-Protokoll richtig funktionieren kann.

```
=> GET info/refs
ca82a6dff817ec66f44342007202690a93763949      refs/heads/master
```

Jetzt hat man eine Liste aller Referenzen und SHA-Prüfsummen in diesem Repository. Als nächstes schaut man die HEAD-Referenz nach, um zu wissen, was ausgecheckt werden muss:

```
=> GET HEAD
ref: refs/heads/master
```

D.h., wenn wir mit dem Prozess fertig sind, wir müssen den Branch `master` auschecken.

Wir können jetzt loslegen. Weil in der Datei `info/refs` der Commit `ca82a6` angegeben ist, fangen wir damit an, dieses Objekt herunterzuladen:

```
=> GET objects/ca/82a6dff817ec66f44342007202690a93763949
(179 bytes of binary data)
```

Wir erhalten also ein Objekt zurück. Dieses Objekt ist im losen Format auf dem Server gespeichert, und wir haben es über einen statischen HTTP-GET-Request herunter geladen. Jetzt können wir es mit `zlib` dekomprimieren, den Header entfernen und den Inhalt des Commits durchsehen:

```
$ 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 brauchen wir also zwei weitere Objekte: cfda3b, welches der Tree der Inhalte dieses Commits ist, und 085bb3, welches der übergeordnete Commit ist:

```
=> GET objects/08/5bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
(179 bytes of data)
```

Das gibt uns das nächste Commit-Objekt. Versuchen wir, das Tree-Objekt zu holen:

```
=> GET objects/cf/da3bf379e4f8dba8717dee55aab78aef7f4daf
(404 - Not Found)
```

Huch. Es sieht so aus, als ob der Tree nicht im losen Format auf dem Server gespeichert ist, weshalb wir eine 404-Antwort („Not found“) erhalten. Dafür kann es verschiedene Gründe geben. Das Objekt könnte in einem anderen, alternativen Repository liegen, oder es könnte sich in einem Packfile befinden. Git sucht deshalb zunächst nach alternativen Repositories:

```
=> GET objects/info/http-alternates
(empty file)
```

```
=> GET objects/info/http-alternates
(leere Datei)
```

Wenn wir hier eine Liste alternativer URLs erhalten, schaut Git dort nach losen Dateien und Packfiles. Auf diese Weise können Repositorys, die Forks von anderen Repositorys sind, mit diesen Objekte im Dateisystem teilen. In unserem Fall sind allerdings keine Alternativen vorhanden, weshalb sich das gesuchte Objekt in einem Packfile befinden muss. Um die vorhandenen Packfiles nachzuschlagen, holt Git die objects/info/packs Datei, die eine entsprechende Auflistung enthält (und ebenfalls mit update-server-info erzeugt wird)

```
=> GET objects/info/packs
P pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack
```

Es gibt nur ein einziges Packfile auf dem Server, weshalb sich unser Objekt darin befinden muss. Aber wir prüfen die Index-Datei, um sicher zu sein. Gäbe es mehrere Packfiles auf dem Server, könnten wir auf diese Weise herausfinden, welches Packfile das gesuchte Objekt enthält:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k of binary data)
```

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.idx
(4k binäre Daten)
```

Nachdem wir jetzt den Packfile-Index haben, können wir prüfen, ob sich unser Objekt darin befindet: der Index enthält eine Liste der SHA-Hashes der Objekte, die sich im Packfile befinden und die jeweiligen Offsets dieser Objekte. Unser Objekt ist vorhanden, also laden wir das Packfile herunter:

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack  
(13k of binary data)
```

```
=> GET objects/pack/pack-816a9b2334da9953e530f27bcac22082a9f5b835.pack  
(13k binäre Daten)
```

Du hast jetzt das Tree-Objekt, also kannst Du jetzt damit fortfahren, über die Commits zu iterieren. Sie sind in unserem Fall allesamt in dem Packfile enthalten, das Du gerade heruntergeladen hast.

Die Ausgabe des ganzen Vorgangs sieht dann in etwa so aus:

```
$ git clone http://github.com/schacon/simplegit-progit.git  
Initialized empty Git repository in /private/tmp/simplegit-progit/.git/  
got ca82a6dff817ec66f44342007202690a93763949  
walk ca82a6dff817ec66f44342007202690a93763949  
got 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
Getting alternates list for http://github.com/schacon/simplegit-progit.git  
Getting pack list for http://github.com/schacon/simplegit-progit.git  
Getting index for pack 816a9b2334da9953e530f27bcac22082a9f5b835  
Getting pack 816a9b2334da9953e530f27bcac22082a9f5b835  
  which contains cfda3bf379e4f8dba8717dee55aab78aef7f4daf  
walk 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7  
walk a11bef06a3f659402fe7563abf99ad00de2209e6
```

Das schlaue Protokoll

Die HTTP-Methode ist simpel, aber sie ist auch ein bisschen ineffizient. Deshalb ist es üblicher, ein schlaues Protokoll für den Datentransfer zu verwenden. Diese Protokolle umfassen serverseitige Prozesse, die Wissen über Git besitzen. Sie können lokale Daten lesen und herausfinden, was auf dem Client schon vorhanden ist oder fehlt und darauf zugeschnittene Daten generieren. Es gibt zwei Sets von Prozessen für den Datentransfer: ein Paar für den Upload und ein Paar für den Download von Daten.

Daten hochladen

Um Daten an einen serverseitigen Prozess zu schicken, verwendet Git die Prozesse `send-pack` und `receive-pack`. Der Prozess `send-pack` läuft auf dem Client und verbindet sich mit einem `receive-pack`-Prozess auf dem Server.

Nehmen wir z.B. an, Du führst `git push origin master` in Deinem Projekt aus und `origin` ist als eine URL mit SSH-Protokoll definiert. Git startet dann einen `send-pack`-Prozess, der eine SSH-Verbindung zum Server initiiert. Dieser versucht, via SSH auf dem Server einen Befehl wie den folgenden auszuführen:

```
$ ssh -x git@github.com "git-receive-pack 'schacon/simplegit-progit.git'"  
005bca82a6dff817ec66f4437202690a93763949 refs/heads/master report-status delete-  
003e085bb3bcb608e1e84b2432f8ecbe6306e7e7 refs/heads/topic  
0000
```

Der `git-receive-pack`-Befehl antwortet dann mit jeweils einer Zeile pro Referenz, die er kennt – in diesem Fall sind das lediglich der Branch `master` und dessen SHA-Prüfsumme. Die erste Zeile listet außerdem Features, die der Server beherrscht (in unserem Fall `report-status` und `delete-refs`).

Jede Zeile beginnt mit einem 4 Byte Hexadezimalzahl-Wert, der angibt, wie lang der Rest der Zeile ist. Die erste Zeile beginnt mit 005b, d.h. dezimal 91. Also ist der Rest der Zeile 91 Zeichen lang. Die nächste Zeile fängt mit 003e an, also dezimal 62. Die letzte Zeile ist 0000, was das Ende der Liste anzeigt.

Nachdem Dein send-pack-Prozess jetzt den Zustand des Servers kennt, kann er als nächstes evaluieren, welche Commits lokal, aber nicht auf dem Server vorhanden sind. der send-pack-Prozess schickt diese Information für jede Referenz, auf die sich der push-Befehl bezieht, an den receive-pack Prozess. Wenn Du beispielsweise den Branch master aktualisierst und einen Branch experiment hinzufügst, dann könnte die Antwort auf send-pack so aussehen:

```
0085ca82a6dff817ec66f44342007202690a93763949    15027957951b64cf874c3557a0f3547bd8f  
00670000000000000000000000000000000000000000 cdfdb42577e2506715f8cfeacdbabc092bf0  
0000
```

Der SHA-1-Wert, der nur aus Nullen besteht, heißt, dass dort zuvor nichts war: Du fügst die experiment-Referenz ja neu hinzu. Würdest Du eine Referenz löschen, würdest Du das Gegenteil sehen: nur Nullen auf der rechten Seite.

Pro Referenz, die Du aktualisierst, schickt Git eine Zeile mit dem alten SHA, dem neuen SHA und der jeweiligen Referenz, die aktualisiert wird. Die erste Zeile listet zudem die Server-Features auf. Als nächstes lädt der Client ein Packfile aller Objekte hoch, die der Server noch nicht kennt. Abschließend antwortet der Server mit einer Erfolgs- oder Fehlermeldung:

```
000Aunpack ok
```

Daten herunterladen

Wenn Du Daten herunterlädst, sind daran die Prozesse `fetch-pack` und `upload-pack` beteiligt. Der Client startet einen `fetch-pack`-Prozess, der sich mit einem `upload-pack`-Prozess auf dem Server verbindet, um auszuhandeln, welche Daten heruntergeladen werden sollen.

Es gibt verschiedene Möglichkeiten, den `upload-pack`-Prozess auf dem Server zu starten: einerseits via SSH auf die gleiche Weise wie den `receive-pack`-Prozess. Und andererseits über den Git-Daemon, der standardmäßig auf dem Server auf dem Port 9418 läuft. Der `fetch-pack`-Prozess schickt etwa Folgendes an den Daemon:

```
003fgit-upload-pack schacon/simplegit-progit.git\0host=myserver.com\0
```

Diese Zeile beginnt wiederum mit 4 Bytes, die angeben, wieviel Daten folgen. Dann kommt der auszuführende Befehl und ein Null-Byte, und schließlich der Hostname des Servers und ein weiteres Null-Byte. Der Git-Daemon prüft, ob der Befehl ausgeführt werden kann, das Repository

existiert und Schreibzugriff erlaubt. Wenn alles stimmt, startet er den upload-pack-Prozess und gibt den Request dorthin weiter.

Wenn Du den fetch-Befehl über SSH verwendest, führt fetch-pack stattdessen etwas aus wie:

```
$ ssh -x git@github.com "git-upload-pack 'schacon/simplegit-progit.git'"
```

In beiden Fällen wird, nachdem fetch-pack verbunden ist, upload-pack eine Antwort wie die folgende zurückschicken:

```
0088ca82a6dff817ec66f44342007202690a93763949 HEAD\0multi_ack thin-pack \
  side-band side-band-64k ofs-delta shallow no-progress include-tag
003fca82a6dff817ec66f44342007202690a93763949 refs/heads/master
003e085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 refs/heads/topic
0000
```

Die Antwort ähnelt der, mit der receive-pack antwortet, aber die aufgelisteten Features sind andere. Zusätzlich wird die HEAD-Referenz mitgeschickt, sodass der Client weiß, was er auschecken muss, falls es sich um einen Clone handelt.

Der fetch-pack-Prozess inspiziert jetzt die vorhandenen Objekte und antwortet mit einer Liste von Objekten, wobei er das Schlüsselwort „want“ für Objekte verwendet, die benötigt werden, und „have“ für Objekte, die bereits vorhanden sind. Am Ende der Liste folgt das Schlüsselwort „done“. Der upload-pack-Prozess schickt dann ein Packfile mit allen benötigten Objekten:

```
0054want ca82a6dff817ec66f44342007202690a93763949 ofs-delta
0032have 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
0000
0009done
```

Das ist ein sehr einfaches Beispiel. In komplexeren Fällen unterstützt der Client die Features multi_ack oder side-band. Aber obiges Beispiel verdeutlicht den grundlegenden Request-Response Zyklus der Prozesse bei schlaun Protokollen.

Wartung und Datenwiederherstellung

Gelegentlich will man ein bisschen aufräumen – ein Repository komprimieren, ein importiertes Repository aufräumen oder verloren gegangene Daten wiederherstellen. Dieses Kapitel wird sich mit einigen derartigen Szenarien befassen.

Wartung

Git führt den Befehl `auto gc` hin und wieder automatisch aus. In den meisten Fällen tut dieser Befehl nichts. Wenn allerdings zu viele lose Objekte (d.h. Objekte, die nicht in einem Packfile gepackt sind) oder zu viele einzelne Packfiles vorhanden sind, führt Git den Befehl `git gc` aus. `gc` steht für „Garbage Collection“. Dieser Befehl führt eine Reihe von Aufgaben durch: er sammelt die losen Objekte und packt sie in ein Packfile, er führt einzelne Packfiles zu einem einzigen, großen Packfile zusammen, und er entfernt Objekte, die mit keinem Commit erreichbar und einige Monate alt sind.

Du kannst `auto gc` wie folgt manuell ausführen:

```
$ git gc --auto
```

Wie schon erwähnt tut dies normalerweise gar nichts. Es müssen sich etwa 7.000 lose Objekte oder mehr als 50 Packfiles angesammelt haben, bevor Git tatsächlich den echten `gc`-Befehl startet. Du kannst diese Werte mit Hilfe der Konfigurationsvariablen `gc.auto` bzw. `gc.autopacklimit` manuell setzen.

`gc` packt außerdem Referenzen in eine einzige Datei zusammen. Nehmen wir an, Dein Repository enthält die folgenden Branches und Tags:

```
$ find .git/refs -type f
.git/refs/heads/experiment
.git/refs/heads/master
.git/refs/tags/v1.0
.git/refs/tags/v1.1
```

Nachdem Du `git gc` ausgeführt hast, werden diese Dateien um der Effizienz willen aus dem Verzeichnis `refs` entfernt und in eine Datei `.git/packed-refs` verschoben, die dann wie folgt aussieht:

```
$ cat .git/packed-refs
# pack-refs with: peeled
cac0cab538b970a37ea1e769cbbde608743bc96d refs/heads/experiment
ab1afef80fac8e34258ff41fc1b867c702daa24b refs/heads/master
cac0cab538b970a37ea1e769cbbde608743bc96d refs/tags/v1.0
9585191f37f7b0fb9444f35a9bf50de191beadc2 refs/tags/v1.1
^1a410efbd13591db07496601ebc7a059dd55cfe9
```

Wenn Du eine Referenz bearbeitest, lässt Git diese Datei unverändert und schreibt stattdessen eine neue Datei nach `refs/heads`. Um eine SHA-Prüfsumme für eine Referenz nachzuschlagen, schaut

Git zunächst im Verzeichnis `refs` und danach erst in der Datei `packed-refs` nach, falls nötig. Wenn eine Referenz also nicht im Verzeichnis `refs` liegt, befindet sie sich wahrscheinlich in der Datei `packed-refs`.

Beachte, dass die letzte Zeile der Datei mit `^` anfängt. Das bedeutet, dass der Tag darüber ein annotierter Tag ist und diese Zeile zeigt den Commit, auf den der annotierte Tag zeigt.

Daten-Wiederherstellung

Irgendwann wird es vielleicht mal vorkommen, dass Du während der Arbeit mit Git einen Commit verlierst. Normalerweise passiert das, wenn Du versehentlich einen Branch löschst, an dem Du gearbeitet hattest und den Du noch brauchst. Oder Du führst `git reset --hard` aus und stellst fest, dass Du einige der Commits noch brauchst. Nehmen wir an, Du steckst in einer solchen Situation – wie kannst Du Deine Commits dann wiederherstellen?

Das folgende Beispiel setzt zuerst den Branch `master` auf einen älteren Commit zurück und stellt die verlorenen Commits dann wieder her. Zunächst schauen wir uns den gegenwärtigen Zustand des Repositorys an:

```
$ git log --pretty=oneline
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Jetzt setzen wir den Branch `master` auf den mittleren Commit zurück:

```
$ git reset --hard 1a410efbd13591db07496601ebc7a059dd55cfe9
HEAD is now at 1a410ef third commit
$ git log --pretty=oneline
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Du hast damit die oberen beiden Commits verloren, d.h. es gibt keinen Branch mehr, von dem aus diese Commits erreichbar wären. Um sie wiederherzustellen, kannst Du einen neuen Branch anlegen, der auf den SHA-Hash des obersten (letzten) Commits zeigt. Der Trick besteht darin, diesen letzten Commit-Hash herauszufinden. Es ist ja nicht so, dass Du Dir jederzeit all die Hashes merken könntest, oder?

In der Regel ist der schnellste Weg, solche Hashes zu finden, der Befehl `git reflog`. Während Du mit Git arbeitest, macht Git im Stillen fortlaufende Notizen darüber, was HEAD ist. Jedes Mal, wenn Du einen Commit anlegst oder den Branch wechselst, wird das „Reflog“ aktualisiert. Das Reflog wird außerdem vom Befehl `git update-ref` verwendet – ein weiterer guter Grund, nicht stattdessen einfach den SHA-Wert in eine Referenz-Datei zu schreiben (wie wir das in der Sektion „Git-Referenzen“ zuvor in diesem Kapitel besprochen haben). Du kannst also jederzeit nachschlagen, woran Du jeweils gearbeitet hast, indem Du den Befehl `git reflog` verwendest:


```
$ git reflog
1a410ef HEAD@{0}: 1a410efbd13591db07496601ebc7a059dd55cfe9: updating HEAD
ab1afef HEAD@{1}: ab1afef80fac8e34258ff41fc1b867c702daa24b: updating HEAD
```

Das zeigt also die beiden verloren gegangenen Commits an, die wir zuvor ausgecheckt hatten. Allerdings zeigt es auch nicht viel mehr Information. Um das Reflog in einer anderen, etwas nützlicheren Weise anzuzeigen, kannst Du `git log -g` verwenden. Das gibt das Reflog im gewohnten Log-Format aus:

```
$ 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 a bit

Es sieht also so aus, als sei der untere Commit derjenige, den Du verloren hast, aber noch brauchst. Du kannst ihn jetzt wiederherstellen, indem Du einen neuen Branch erstellst, der auf diesen Commit zeigt. Beispielsweise kannst Du einen Branch `recover-branch` anlegen, der auf den Commit `ab1afef` zeigt:

```
$ git branch recover-branch ab1afef
$ git log --pretty=oneline recover-branch
ab1afef80fac8e34258ff41fc1b867c702daa24b modified repo a bit
484a59275031909e19aadb7c92262719cfcdf19a added repo.rb
1a410efbd13591db07496601ebc7a059dd55cfe9 third commit
cac0cab538b970a37ea1e769cbbde608743bc96d second commit
fdf4fc3344e67ab068f836878b6c4951e3b15f3d first commit
```

Sehr gut. Du hast jetzt einen neuen Branch `recover-branch`, der diejenigen Commits enthält, die sich zuvor in Deinem Branch `master` befanden. Damit hast Du wieder Zugriff auf die beiden verloren gegangenen Commits. Als nächstes nehmen wir aber außerdem an, dass diese verlorenen Commits aus irgendeinem Grunde nicht im Reflog enthalten sind – Du kannst das z.B. simulieren, indem Du den Branch `recover-branch` und das Reflog löschst. Damit sind die beiden Commits jetzt von nirgendwo her mehr erreichbar:

```
$ git branch -D recover-branch
$ rm -Rf .git/logs/
```

Das Reflog wird im Verzeichnis `.git/logs/` aufbewahrt, d.h. Du hast nun faktisch kein Reflog mehr. Wie kann man einen Commit jetzt noch wiederherstellen? Eine Möglichkeit dazu ist der

Befehl `git fsck`, der die Integrität der Git-Datenbank prüft. Wenn Du den Befehl mit der Option `--full` ausführst, zeigt er alle Objekte an, auf die nicht von einem anderen Objekt verwiesen wird:

```
$ git fsck --full
dangling blob d670460b4b4aece5915caf5c68d12f560a9fe3e4
dangling commit ab1afef80fac8e34258ff41fc1b867c702daa24b
dangling tree aea790b9a58f6cf6f2804eeac9f0abbe9631e4c9
dangling blob 7108f7ecb345ee9d0084193f147cdad4d2998293
```

In diesem Fall findest Du den verlorenen Commit nach den Worten „dangling commit“. Du kannst ihn dann auf dieselbe Weise wiederherstellen wie zuvor, indem Du einen Branch erstellst, der auf diesen Commit-Hash zeigt.

Objekte entfernen

Git ist in vielerlei Hinsicht unschlagbar, aber es gibt auch Features, die Probleme verursachen können. Ein solches Problem kann darin bestehen, dass `git clone` die vollständige Historie eines Projektes herunterlädt, d.h. jede einzelne Version jeder einzelnen Datei. Das ist eine feine Sache, solange es sich um Quellcode handelt, denn Git ist darauf optimiert, diese Art von Daten effizient zu komprimieren. Wenn allerdings irgendwann einmal eine einzelne, sehr große Datei zur Versionskontrolle hinzugefügt wurde, wird jeder Clone dieses Repositorys diese Datei gezwungenermaßen herunterladen müssen – auch dann, wenn die Datei inzwischen aus dem Repository entfernt würde. Weil sie über die Historie erreichbar ist, muss die Datei vorhanden sein.

Hierin kann ein großes Problem bestehen, wenn Du Subversion- oder Perforce-Repositorys nach Git konvertierst. Weil Du in diesen Systemen nicht die gesamte Historie herunterlädst, kann diese Art von Hinzufügung einige unangenehme Konsequenzen haben. Wenn Du Dein Repository aus einem anderen System importiert hast oder aus irgendeinem anderen Grunde findest, dass es sehr viel größer ist, als es eigentlich sein sollte, kannst Du große Objekte wie folgt suchen und entfernen.

Sei Dir allerdings bewusst, dass diese Technik die Commit-Historie zerstört. Sie schreibt angefangen beim ursprünglichen Tree jedes einzelne Commit-Objekt neu, um die jeweilige, große Datei zu entfernen. Wenn Du das direkt nach einem Import tust, d.h. bevor jemand angefangen hat, auf der Basis eines Commits zu arbeiten, ist das in Ordnung – andernfalls müssen alle Mitarbeiter ihre Arbeit auf Deinen Commit rebasen.

Um das zu demonstrieren, werden wir eine große Datei zu Deinem Test-Repository hinzufügen, sie dann im nächsten Commit löschen, in der Datenbank nachschlagen und sie schließlich dauerhaft aus dem Repository entfernen. Als erstes checke also eine große Datei in Dein Repository ein:

```
$ curl http://kernel.org/pub/software/scm/git/git-1.6.3.1.tar.bz2 > git.tbz2
$ git add git.tbz2
$ git commit -am 'added git tarball'
```

```
[master 6df7640] added git tarball
1 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 git.tbz2
```

Oha. Du wolltest keinen dermaßen großen Tarball in Deinem Projekt. Am besten löschen wir es gleich wieder:

```
$ git rm git.tbz2
rm 'git.tbz2'
$ git commit -m 'oops - removed large tarball'
[master da3f30d] oops - removed large tarball
1 files changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 git.tbz2
```

Jetzt lassen wir die Garbage-Collection über die Datenbank laufen und sehen, wieviel Platz sie braucht:

```
$ git gc
Counting objects: 21, done.
Delta compression using 2 threads.
Compressing objects: 100% (16/16), done.
Writing objects: 100% (21/21), done.
Total 21 (delta 3), reused 15 (delta 1)
```

Du kannst auch den Befehl `count-objects` laufen lassen, um einen schnellen Überblick darüber zu erhalten, wieviel Platz das Repository einnimmt:

```
$ git count-objects -v
count: 4
size: 16
in-pack: 21
packs: 1
size-pack: 2016
prune-packable: 0
garbage: 0
```

Der `size-pack`-Eintrag zeigt die Größe der Packfiles in Kilobytes an, d.h. Dein Repository braucht 2 MB. Vor dem letzten Commit lag dieser Wert eher bei 2 KB. D.h., die Datei wurde im letzten Commit eindeutig nicht aus der History entfernt. Jedes Mal, wenn jemand künftig dieses sehr kleine Repository klonet, wird er die 2 MB herunterladen müssen – nur weil wir versehentlich diese große Datei hinzugefügt hatten. Also versuchen wir, sie endgültig loszuwerden.

Zunächst mal müssen wir sie finden. In diesem Fall wissen wir bereits, um welche Datei es sich handelt. Aber nehmen wir an, wir wüssten es nicht. Wie würdest Du herausfinden, welche Datei (oder welche Dateien) so viel Platz verbrauchen? Wenn Du `git gc` laufen gelassen hast, werden sich alle Objekte in einem Packfile befinden. Du kannst dann große Objekte identifizieren, indem Du einen weiteren Plumbing-Befehl, nämlich `git verify-pack` ausführst und nach dem dritten Feld der Ausgabe sortierst, d.h. der Dateigröße. Du kannst sie außerdem durch den `tail` Befehl leiten, denn Du bist ja nur an den wenigen größten Objekten interessiert:

```
$ git verify-pack -v .git/objects/pack/pack-3f8c0...bb.idx | sort -k 3 -n | tail
e3f094f522629ae358806b17daf78246c27c007b blob      1486 734 4667
05408d195263d853f09dca71d55116663690c27c blob      12908 3478 1189
7a9eb2fba2b1811321254ac360970fc169ba2330 blob      2056716 2056872 5401
```

Das größte Objekt ist ganz klar das letzte: es ist 2 MB groß. Um herauszufinden, welche Datei das ist, kannst Du den `rev-list` Befehl verwenden, den wir in Kapitel 7 schon einmal kurz verwendet haben. Wenn Du die Optionen `--objects` und `rev-list` verwendest, werden alle Commit- und Blob-SHAs mit den jeweiligen Dateipfaden aufgelistet, die mit ihnen assoziiert sind. Auf diese Weise kannst Du den Namen des Blobs finden:

```
$ git rev-list --objects --all | grep 7a9eb2fb
7a9eb2fba2b1811321254ac360970fc169ba2330 git.tbz2
```

Du musst diese Datei jetzt aus allen Trees entfernen, in denen sie sich befindet. Du kannst leicht herausfinden, welche Commits diese Datei verändert haben:

```
$ git log --pretty=oneline --branches -- git.tbz2
da3f30d019005479c99eb4c3406225613985a1db oops - removed large tarball
6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 added git tarball
```

Es geht also darum, alle Commits angefangen bei 6df76 neu zu schreiben, sodass der Tarball nicht mehr in Deiner Git-Historie enthalten ist. Um das zu erreichen, verwendest Du den Befehl `git filter-branch`, den wir schon mal in Kapitel 6 verwendet haben:

```
$ git filter-branch --index-filter \
    'git rm --cached --ignore-unmatch git.tbz2' -- 6df7640^..
Rewrite 6df764092f3e7c8f5f94cbe08ee5cf42e92a0289 (1/2)rm 'git.tbz2'
Rewrite da3f30d019005479c99eb4c3406225613985a1db (2/2)
Ref 'refs/heads/master' was rewritten
```

Die Option `--index-filter` ist ähnlich der Option `--tree-filter` aus Kapitel 6. Allerdings übergibt man in diesem Fall nicht einen Befehl, der die Dateien verändert, die sich jeweils ausgecheckt auf der Festplatte befinden. Stattdessen verändert man jeweils die Staging-Area bzw. den Index. Statt eine bestimmte Datei mit z.B: `rm file` zu entfernen, müssen wir also `git rm --cached` verwenden – denn wir wollen sie aus dem Index, nicht von der Festplatte löschen. Der Grund dafür ist einfach Geschwindigkeit: Git braucht nicht jede einzelne Revision auf die Festplatte auszuchecken, um den Filter anzuwenden. Auf diese Weise läuft der ganze Prozess sehr viel schneller. Man kann dieselbe Aufgabe aber auch mit `--tree-filter` erledigen, wenn man will. Die Option `--ignore-match` weist Git an, nicht mit einer Fehlermeldung abubrechen, wenn die Datei, die wir löschen wollen, nicht vorhanden ist. Außerdem teilen wir `filter-branch` mit, die Historie nur von dem Commit 6df7640 an umzuschreiben. Andernfalls würde der Befehl von ganz vorn beginnen und unnötig länger brauchen.

Deine Historie enthält jetzt nicht länger eine Referenz auf diese Datei. Dein Reflog und ein neues Set an Referenzen, die Git unter `.git/refs/original` angelegt hat, als Du `filter-branch` ausgeführt hast, tun das allerdings immer noch – also entfernen wir sie von dort und packen das Repository erneut. Bevor Du packst, musst Du zuerst alles entfernen, was noch Referenzen auf das

alte Objekt enthält:

```
$ rm -Rf .git/refs/original
$ rm -Rf .git/logs/
$ git gc
Counting objects: 19, done.
Delta compression using 2 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (19/19), done.
Total 19 (delta 3), reused 16 (delta 1)
```

Prüfen wir also, wieviel Platz wir damit eingespart haben:

```
$ git count-objects -v
count: 8
size: 2040
in-pack: 19
packs: 1
size-pack: 7
prune-packable: 0
garbage: 0
```

Das gepackte Repository umfasst jetzt nur noch 7K – sehr viel besser als die vorherigen 2MB. Du kannst an dem Wert `size` erkennen, dass sich die große Datei selbst jetzt immer noch in Deinen losen Objekten befindet. Aber sie wird bei einem `git push` oder anschließenden `git clone` nicht übermittelt werden – und das war unser Ziel. Wenn Du das wirklich willst, kannst Du sie jetzt vollständig und endgültig mit `git prune --expire` aus Deinem Repository löschen.

Zusammenfassung

Du solltest jetzt ein gutes Verständnis davon haben, was Git im Hintergrund tut, und in einem gewissen Maße auch davon, wie es implementiert ist. In diesem Kapitel haben wir eine Reihe von Plumbing-Befehlen besprochen, also Befehlen, die grundlegender und einfacher als die Porcelain-Befehle sind, um die es im restlichen Buch ging. Dieses Verständnis sollte Dir helfen, zu verstehen, warum Git tut, was es tut – und natürlich auch dabei, Deine eigenen Werkzeuge und Hilfsskripte zu schreiben, um einen bestimmten Arbeitsablauf für Dich anzupassen.

Git als ein Dateisystem, das Inhalte adressieren kann, ist ein äußerst mächtiges Werkzeug, das Du leicht für mehr als „nur“ als VCS einsetzen kannst. Ich hoffe, Du kannst Dein neugewonnenes Wissen der Git Interna nutzen, um Deine eigene tolle Anwendung dieser Technologie zu implementieren und Dich wohler damit zu fühlen, Git auch in fortgeschrittener Weise zu benutzen.