Bernd Öggl
Michael Kofler

Git

Projektverwaltung für Entwickler und DevOps-Teams

- Git effektiv nutzen und sicher administrieren
- Features von GitHub und GitLab einsetzen
- Best Practices & Workflows für eigene Repositorys

Beispiele zum Download und Klonen

Rheinwerk Computing

Vorwort

Immer, wenn mehrere Personen gemeinsam an einem Softwareprojekt arbeiten, braucht es ein System, um alle durchgeführten Änderungen nachvollziehbar zu speichern. Gleichzeitig gibt ein derartiges Versionsverwaltungssystem allen Entwicklern Zugriff auf das gesamte Projekt. Jeder Programmierer weiß, was die anderen zuletzt gemacht haben, jede Entwicklerin kann den Code der anderen ausprobieren und das Zusammenspiel mit ihren eigenen Änderungen testen.

In der Vergangenheit gab es viele Versionsverwaltungssysteme, z. B. CVS, Subversion (SVN) oder Visual SourceSafe. Im vergangenen Jahrzehnt hat sich aber Git zum De-facto-Standard entwickelt.

Einen wesentlichen Anteil an diesem Erfolg hatte die Webplattform GitHub, die den Einstieg und die Nutzung von Git wesentlich vereinfachte. Unzählige Open-Source-Projekte nutzen das kostenlose Angebot GitHubs zum Projekt-Hosting. Kommerzielle Kunden, die den Quellcode nicht veröffentlichen wollten, zahlen für diesen Service. GitHub ist natürlich nicht die einzige Git-Plattform: Wichtige Konkurrenten sind z. B. GitLab, Azure DevOps und Bitbucket. Dessen ungeachtet kaufte Microsoft 2018 GitHub für unglaubliche 7,5 Milliarden US$. Im Gegensatz zu anderen Übernahmen hat dies bisher der Popularität von GitHub nicht geschadet.

Die Geschichte von Git

Git entstand, weil Linus Torvalds für die Weiterentwicklung des Linux-Kernels ein neues Versionsverwaltungssystem brauchte. Die Entwicklergemeinde hatte zuvor das Programm BitKeeper verwendet. Linux Torvalds war mit dem Programm grundsätzlich zufrieden, eine Lizenzänderung machte aber einen Wechsel erforderlich. Von den damals verfügbaren Open-Source-Programmen genügte keines seinen hohen Ansprüchen.

So stoppte der Linux-Chefentwickler kurzzeitig seine Hauptarbeit und schuf in nur zwei Wochen das Grundgerüst von Git. Der Name *Git* steht sinngemäß für *Blödmann* oder *Depp*, und auch die Hilfeseite `man git` bezeichnet das Programm als *the stupid content tracker*.

Was für ein Understatement das ist, wurde erst nach und nach klar, als Linus Torvalds die Weiterentwicklung von Git längst wieder aus der Hand gegeben hatte: Nicht nur die Kernel-Entwickler stellten ihre Arbeit rasch und problemlos auf Git um, in den folgenden Jahren wechselten immer mehr Softwareprojekte auch außerhalb der Open-Source-Welt zu Git.

Den endgültigen Durchbruch schaffte Git, als sich Webplattformen wie GitHub und GitLab etablierten. Diese Websites vereinfachen das Hosting von Git-Projekten enorm und sind heute aus dem Git-Alltag nicht mehr wegzudenken. (Selbst der Linux-Kernel befindet sich mittlerweile auf GitHub!)

Ein bisschen ist das eine Ironie des Schicksals: Linus Torvalds wichtigstes Ziel beim Design von Git war es, ein dezentrales Versionsverwaltungssystem zu schaffen. Aber erst der zentralistische Ansatz von GitHub und Co. machte Git für Entwickler abseits der Guru-Liga richtig attraktiv.

Es gibt heute Stimmen, die die Bedeutung von Git ebenso hoch einschätzen wie die von Linux. Damit ist es Linus Torvalds gleich zwei Mal gelungen, einen Bereich des Software-Universums vollständig auf den Kopf zu stellen.

Jeder verwendet es, keiner versteht es

Bei aller Begeisterung für Git: Es ist unübersehbar, dass Git von Profis für Profis konzipiert wurde. Wir wollen in diesem Buch gar nicht erst den Eindruck erwecken, Git wäre einfach. Das ist es nicht:

- Häufig führt nicht ein Weg zum Ziel, vielmehr gibt es mehrere Wege. Für die, die Git schon beherrschen, ist das nützlich; aber wenn Sie Git gerade lernen, verwirrt diese Vielfalt.
- Vielen Open-Source-Projekten wird der Vorwurf gemacht, sie seien schlecht dokumentiert. Das kann man bei Git wirklich nicht sagen. Im Gegenteil! Jedes Git-Kommando, jede Anwendungsmöglichkeit wird in `man`-Seiten sowie auf der Webseite *https://git-scm.com/docs* so ausführlich und mit allen erdenklichen Sonderfällen erläutert, dass man sich in den Details geradezu verliert.
- Erschwerend kommt hinzu, dass es ähnliche Begriffe mit unterschiedlichen Bedeutungen gibt, leicht zu verwechselnde Subkommandos, die stark voneinander abweichende Aufgaben erfüllen. Manche Begriffe haben je nach Kontext unterschiedliche Bedeutungen oder werden in der Dokumentation uneinheitlich verwendet.

Wir geben es ganz offen zu: Trotz jahrelanger Git-Praxis haben wir beim Schreiben dieses Buchs noch eine Menge Details dazugelernt!

Über dieses Buch

Natürlich ist es möglich, Git sehr minimalistisch zu verwenden. Allerdings können kleine Abweichungen von der täglichen Routine dann zu überraschenden und oft unverständlichen Nebenwirkungen oder Fehlern führen.

Jeder Git-Einsteiger kennt das Gefühl, wenn ein Git-Kommando eine unverständliche Fehlermeldung liefert: Mit kaltem Schweiß überlegt man, ob man gerade das Repository für alle Entwickler nachhaltig zerstört hat und wen man bitten könnte, Git mit den richtigen Kommandos doch zur Weiterarbeit zu überreden.

Deswegen ist es nicht zielführend, Git zu beschreiben, ohne dabei in die Tiefe zu gehen. Erst ein gutes Verständnis für die Funktionsweise von Git gibt die notwendige Sicherheit, Merge-Konflikte oder andere Probleme sauber beheben zu können.

Gleichzeitig war uns aber klar, dass dieses Buch nur funktionieren kann, wenn wir den wesentlichen Funktionen den Vorrang geben. Trotz 400 Seiten ist dieses Buch *nicht* die allumfassende Anleitung zu Git, die auch den letzten Sonderfall berücksichtigt und jedes noch so exotische Git-Subkommando vorstellt. Wir haben daher in diesem Buch die Spreu vom Weizen getrennt.

Dieses Buch ist in überschaubare Kapitel gegliedert, die Sie wie bei einem Bausteinsystem nach Bedarf lesen können:

- Nach einer kurzen Einführung (»Git in zehn Minuten«) führen wir in den Kapiteln »Learning by Doing«, »Git-Grundlagen« und schließlich »Datenanalyse im Git-Repository« in den Umgang mit Git ein. Dabei konzentrieren wir uns auf die Nutzung von Git auf Kommandoebene und gehen nur am Rande auf Plattformen wie GitHub bzw. auf andere Benutzeroberflächen ein.

  Git-Einsteigern empfehlen wir, mit diesen vier Kapiteln zu starten. Selbst wenn Sie schon etwas Git-Erfahrung haben, sollten Sie sich unbedingt ein paar Stunden Zeit nehmen, um »Git-Grundlagen« zu lesen und einige der dort vorgestellten Techniken (Merging, Rebasing etc.) in einem Test-Repository auszuprobieren.
- Die folgenden drei Kapitel – »GitHub«, »GitLab« sowie »Azure DevOps, Bitbucket, Gitea und Gitolite« – stellen die wichtigsten Git-Plattformen vor. Gerade für komplexe Projekte bieten diese Plattformen nützliche Zusatzfunktionen, z. B. um automatische Tests durchzuführen oder um Continuous Integration zu implementieren.

  Selbstverständlich berücksichtigen wir auch den Fall, dass Sie Ihr Git-Repository selbst hosten möchten. Mit GitLab, Gitea oder Gitolite lässt sich dieser Wunsch relativ leicht realisieren.
- Damit wenden wir uns von den Grundlagen der Praxis zu: Im Kapitel »Workflows« zeigen wir populäre Muster, wie Sie die Arbeit vieler Entwickler mit Git in geordnete Bahnen (*Branches*) leiten.

  Im Kapitel »Arbeitstechniken« stehen fortgeschrittene Git-Funktionen im Vordergrund, z. B. Hooks, Submodule, Subtrees sowie die Zwei-Faktor-Authentifizierung, die alle größeren Git-Plattformen unterstützen.

»Git in der Praxis« zeigt, wie Sie auf Linux-Systemen Konfigurationsdateien (*Dotfiles*) bzw. das ganze `/etc`-Verzeichnis mit Git versionieren, wie Sie ein Projekt von SVN auf Git umstellen oder wie Sie eine simple Website schnell und einfach mit Git und Hugo realisieren.

»Gängige Probleme und ihre Lösungen« helfen Ihnen bei schwer verständlichen Fehlermeldungen aus der Sackgasse. Hier finden Sie auch Anleitungen, wie Sie Sonderwünsche realisieren – z. B. wie Sie große Dateien aus dem Git-Repository entfernen oder wie Sie einen Merge-Vorgang nur für eine ausgewählte Datei durchführen.

- Die »Kommandoreferenz« fasst in aller Kürze die wichtigsten Git-Kommandos und deren Optionen zusammen. Dabei haben wir uns vom Motto »weniger ist mehr« leiten lassen. Unser Ziel war nicht eine vollständige Referenz, sondern eine Art »Essenz von Git«.

Beispiel-Repositorys

Einige Beispiele aus dem Buch stellen wir Ihnen auf GitHub zur Verfügung. Werfen Sie einen Blick auf die Begleitwebsite zum Buch bzw. direkt auf GitHub!

https://gitbuch.info
https://github.com/git-buch

Lieber Leser, liebe Leserin!

Uns ist bewusst, dass Sie vielleicht nicht mit großer Freude die Lektüre dieses Buchs beginnen: Sie wollen oder müssen für ein Projekt Git verwenden. Aber Ihr Ziel ist nicht Git an sich, vielmehr wollen Sie Code produzieren, Ihr Projekt vorantreiben. Sie haben eigentlich weder Zeit noch Lust, sich mit Git zu beschäftigen – Sie wollen gerade so viel wissen, dass Sie Git fehlerfrei anwenden können.

Wir haben dafür Verständnis. Trotzdem empfehlen wir Ihnen dringend, ein paar Stunden mehr als geplant zu investieren, um Git systematisch kennenzulernen.

Wir versprechen Ihnen: Sie gewinnen diese Zeit später zurück! Zu wenig Git-Verständnis bedeutet zwangsläufig, dass Sie immer wieder im Internet nach der Lösung für ein gerade aufgetretenes Problem suchen müssen (oft unter Zeitdruck).

Auch wenn Sie aktuell primär Ihr Projekt im Fokus haben: Git-Kenntnisse sind langfristig eine Kernkompetenz, die Sie als Entwickler(in) in vielen zukünftigen Projekten brauchen werden! In diesem Sinne wünschen wir Ihnen viel Erfolg mit Git!

Michael Kofler (*https://kofler.info*)
Bernd Öggl (*https://webman.at*)

Kapitel 4
Datenanalyse im Git-Repository

In diesem Kapitel geht es darum, ein Repository gezielt nach Daten zu durchsuchen: Welche Dateien sind unter Versionskontrolle? In welchen Commits wurde eine Datei zuletzt geändert? Welche Änderungen wurden dabei durchgeführt? In welchen Commits kommt ein bestimmter Begriff in der Commit-Message vor?

Wie im vorigen Kapitel konzentrieren wir uns dabei auf den Einsatz des Kommandos `git` und gehen nur am Rande auf andere Tools ein:

- Commits durchsuchen: `git log`, `git reflog`, `git tag`, `git shortlog`
- Dateien durchsuchen: `git show`, `git diff`, `git grep`, `git blame`
- Fehler suchen: `git bisect`
- Statistik und Visualisierung: `git shortlog`, `gitstats`, Gitgraph.js

Wir wollen aber nicht unerwähnt lassen, dass Entwicklungsumgebungen, Editoren, Weboberflächen von Git-Plattformen oder spezielle (oft kommerzielle) Programme wie GitKraken beim Durchsuchen des Git-Repositorys mehr Komfort bieten. Auf unseren persönlichen Favoriten, das Programm VSCode in Kombination mit der Erweiterung GitLens, haben wir ja schon mehrfach hingewiesen.

Aber wie so oft gilt auch hier: Wenn Sie einmal verstanden haben, wie Git intern funktioniert und welche Funktionen es auf Kommandoebene gibt, fällt Ihnen die Anwendung solcher Tools umso leichter. Außerdem stößt jedes Tool früher an die Grenzen als das Kommando `git`!

4.1 Commits durchsuchen (git log)

Das Kommando `git log` zeigt, ausgehend vom aktuellen Commit, die vorangegangenen Commits an. Das ist möglich, weil zusammen mit jedem Commit auch eine Referenz auf den Parent-Commit gespeichert wird. (Bei Merge-Commits gibt es entsprechend mindestens zwei Parents.)

Standardmäßig zeigt `git log` zu jedem Commit alle Metadaten (Datum, Autor, Zweig etc.) sowie die jeweilige Commit-Message an (siehe Abbildung 4.1). Wenn es mehr

Commits gibt, als im Terminalfenster Platz finden, können Sie mit den Cursortasten durch die Commit-Abfolge scrollen. Q beendet die Anzeige.

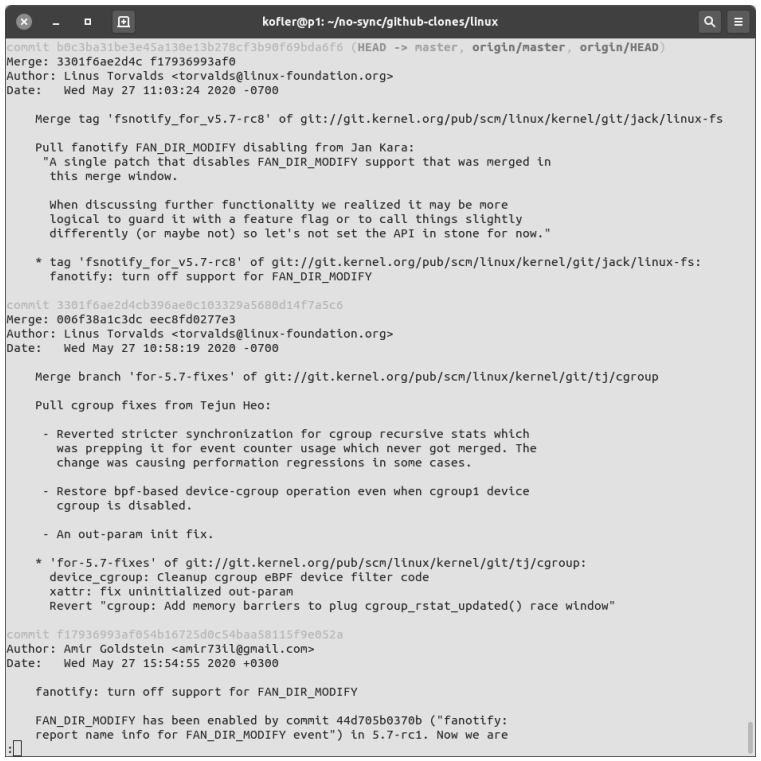

Abbildung 4.1 Commits des Linux-Kernels in einem Terminalfenster

Spielwiese Linux-Kernel

Wenn Sie sich gerade in Git einarbeiten, haben Sie vielleicht noch kein großes eigenes Git-Repository. Verwenden Sie einfach den Linux-Kernel! Mit fast einer Million Commits von unzähligen Entwicklern und über 600 mit Tags gekennzeichneten Releases (Stand Mitte 2020) finden Sie eine wunderbare Spielwiese vor – und sehen außerdem, wie schnell Git selbst bei riesigen Repositorys funktioniert. Der einzige Nachteil: Mit mehr als 4 GByte ist der Platzbedarf auf Ihrer Festplatte/SSD nicht unerheblich.

```
git clone https://github.com/torvalds/linux.git
```

Intern wird die Ausgabe von `git log` durch einen sogenannte *Pager* geleitet, wobei üblicherweise das Kommando `less` zum Einsatz kommt. Dementsprechend gelten die bei `less` üblichen Tastenkürzel (siehe Tabelle 4.1). Besonders praktisch ist die Suchfunktion, die Sie mit / starten.

Tastenkürzel	Funktion
Cursortasten	Scrollen durch den Text.
G	Springt an den Beginn des Texts.
⇧+G	Springt an das Ende des Texts.
/ abc ↵	Sucht vorwärts.
? abc ↵	Sucht rückwärts.
N	Wiederholt die letzte Suche (vorwärts).
⇧+N	Wiederholt die letzte Suche (rückwärts).
Q	Beendet `less`.
H	Zeigt die Onlinehilfe an.

Tabelle 4.1 »less«-Tastenkürzel

Wenn `git` die Buchstaben ä, ö, ü und ß, andere internationale Zeichen oder Emojis fehlerhaft anzeigt, liegt dies oft am fehlerhaften Zusammenspiel zwischen `git` und dem Textanzeigekommando `less`. Abhilfe schafft vorübergehend die Option `--no-pager`, dauerhaft das folgende Kommando:

```
git config --global core.pager 'less --raw-control-chars'
```

Übersichtliches Logging

Häufig zeigt `git log` mehr Details an, als Sie eigentlich brauchen. Dafür fehlen vielleicht andere Informationen. Abhilfe schaffen die folgenden zwei Optionen:

- `--graph`: visualisiert Zweige (ASCII-Art)
- `--oneline`: fasst Metadaten und Commit-Message in einer Zeile zusammen

Umgekehrt fehlt im Logging vielleicht genau die Information, nach der Sie suchen:

- `--all`: zeigt auch Commits anderer Zweige an
- `--decorate`: zeigt auch Tags an
- `--name-only`: listet die veränderten Dateien auf
- `--name-status`: listet die Art der Änderungen pro Datei auf (z. B. `M` für *modified*, `D` für *deleted*, `A` für *added*)
- `--pretty=online|short|medium|full|fuller|...`: vordefinierte Ausgabeformate für die Metadaten und die Commit-Message
- `--numstat`: listet die Anzahl der geänderten Zeilen pro Datei auf
- `--stat`: listet den Umfang der Änderungen pro Datei als Balkendiagramm auf

Es ist eine gute Idee, die Wirkung der Optionen einfach einmal auszuprobieren. Die meisten Optionen können miteinander kombiniert werden. Abbildung 4.2 zeigt nochmals die Commits des Linux-Kernels, diesmal mit den Optionen `--graph --oneline`. Eine detailliertere Beschreibung der Syntax von `git log` folgt in Kapitel 12, »Kommandoreferenz«.

Abbildung 4.2 Kompakte Commit-Darstellung mit Zweigvisualisierung

Eigene Formatierung (Pretty-Syntax)

Wenn Sie mit den vorgegebenen Formaten nicht zufrieden sind, können Sie die Ausgabe der Commits durch die Option `--pretty=format'<fmt>'` selbst formatieren. Dabei setzt sich `<fmt>` aus `printf`-ähnlichen Codes zusammen. Unzählige weitere Codes dokumentiert `man git-log`. Das Format für die Ausgabe von Datum und Uhrzeit kann zusätzlich durch die Option `--date=iso|local|short|...` beeinflusst werden.

Im folgenden Beispiel sollen nur der siebenstellige Commit-Code, die ersten 20 Zeichen des Entwicklernamens sowie die erste Zeile der Commit-Message angezeigt werden:

```
git log --pretty=format:'%h %<(20)%an %s'
  35870e2  Michael Kofler      bugfix y
  ebdb53f  Bernd Öggl          added validation
  9ae3fb8  Michael Kofler      feature x
```

Um den Autorennamen rot anzuzeigen, muss die Formatzeichenkette wie folgt umgebaut werden:

```
git log --pretty=format:'%h %>(20)%Cred%an%Creset %s'
```

Die wichtigsten Codes listet Tabelle 4.2 für Sie auf.

Code	Bedeutung
%H	vollständiger Hashcode
%h	siebenstelliger Hashcode
%ad	Author Date
%cd	Commit Date
%an	Name des Entwicklers (*Author*)
%ae	E-Mail-Adresse des Entwicklers
%s	erste Zeile der Commit-Message (*Subject*)
%b	Rest der Commit-Message (*Body*)
%n	neue Zeile
%<(20)	nächste Spalte 20 Zeichen linksbündig
%>(20)	nächste Spalte 20 Zeichen rechtsbündig
%Cred	ab hier Ausdruck rot darstellen
%Cgreen	ab hier Ausdruck grün darstellen
%C...	diverse weitere Farben
%Creset	Farbe zurücksetzen

Tabelle 4.2 Pretty-Format-Codes

Commit-Messages durchsuchen

Mit der Option `--grep 'pattern'` zeigt `git log` nur die Commits an, in deren Message der Suchbegriff vorkommt. Dabei wird auch die Groß- und Kleinschreibung berücksichtigt. Wenn Sie das nicht wollen, geben Sie zusätzlich die Option `-i` an.

Das folgende Kommando sucht in *allen* Commits (nicht nur in denen des aktuellen Zweigs) nach dem Suchbegriff »CVE« in beliebiger Groß- und Kleinschreibung:

```
git log --all -i --grep CVE
```

Leider werden die gefundenen Suchbegriffe nicht farblich hervorgehoben. Das können Sie erreichen, indem Sie zuerst `git log` ohne die Option `--grep` ausführen, den resultierenden Ergebnistext dann mit dem Kommando `grep` filtern und schließlich durch `less` leiten. Diese Vorgehensweise ist allerdings nicht besonders effizient und bietet weniger Optionen bei der Darstellung der Commits. (Die `grep`-Option `-5` bewirkt, dass außer der gefundenen Zeile jeweils die fünf Zeilen oberhalb und

unterhalb dargestellt werden. Die `less`-Option `-R` ist notwendig, damit die von `grep` weitergeleiteten Farbcodes korrekt verarbeitet werden.)

```
git log --all | grep -i -5 --color=always CVE | less -R
```

Commits suchen, die bestimmte Dateien verändern

Oft sind Sie nicht an *allen* Commits interessiert, sondern nur an Commits, in denen eine bestimmte Datei oder irgendeine Datei aus einem bestimmten Verzeichnis verändert wird. Dazu übergeben Sie an `git log` den Datei- oder Verzeichnisnamen. Falls es eine Namensgleichheit mit Tags, Branches etc. gibt, müssen Sie `--` voranstellen.

Das folgende Kommando filtert die Commits des Linux-Kernels heraus, in denen Dateien des ext4-Treibers (im Verzeichnis `fs/ext4`) verändert wurden. Dank der Option `--stat` werden auch gleich die Namen der geänderten Dateien und der Umfang der Änderungen angezeigt.

```
git log --oneline --stat -- fs/ext4
  959f75845129 ext4: fix fiemap size checks for bitmap files
   fs/ext4/extents.c | 31 +++++++++++++++++++++++++++++++
   fs/ext4/ioctl.c   | 33 ++-------------------------------
   2 files changed, 33 insertions(+), 31 deletions(-)
   ...
  54d3adbc29f0 ext4: save all error info in save_error_info()
                     and drop ext4_set_errno()
   fs/ext4/balloc.c         |   7 +++----
   fs/ext4/block_validity.c |  18 ++++++++-----------
   fs/ext4/ext4.h           |  54 ++++++++++++++++++++++++++++++++++++++++++++++++
                               ++++++--------------------
   fs/ext4/ext4_jbd2.c      |  13 ++++++---------
   ...
```

Umbenannte Dateien verfolgen

`git log -- <file>` kommt nicht mit dem Fall zurecht, dass sich der Name einer Datei ändert. In solchen Fällen müssen Sie die zusätzliche Option `--follow` verwenden, also `git log --follow -- <file>`.

Commits eines bestimmten Entwicklers suchen

Mit der Option `--author <name>` oder `--author <email>` filtern Sie die Commits eines bestimmten Entwicklers heraus. Wie bei `--grep` werden `<name>` bzw. `<email>` als Muster interpretiert.

Mit dem folgenden Beispiel bleiben wir beim Dateisystem-Code des Linux-Kernels und suchen nach Commits von *Theodore Ts'o*. Der Apostroph im Namen macht die Suche nicht einfacher. Geben Sie stattdessen einfach einen Punkt an. (Der Punkt wird gemäß der Syntax für reguläre Ausdrücke als Platzhalter für ein beliebiges Zeichen interpretiert.)

```
git log --oneline  --author 'Theodore Ts.o'
```

Das zweite Beispiel sucht nach E-Mail-Adressen, in denen *ibm.com* vorkommt:

```
git log   --author 'ibm.com'
```

Commit-Bereich einschränken (Range-Syntax)

Normalerweise liefert `git log [<branch>]` alle Commits des aktuellen bzw. des angegebenen Zweigs bis zurück zum Anfang der Commit-Abfolge, also in der Regel bis hin zum ersten Commit des Repositorys. Das ist nicht immer sinnvoll. Oft sind Sie nur an Commits interessiert, die spezifisch für einen Branch oder mehrere Branches gelten, nicht aber an der gemeinsamen Basis. In solchen Fällen können Sie die Range-Syntax `<branch1>..<branch2>` bzw. `<branch1>...<branch2>` verwenden. Anstelle von Zweignamen können Sie auch Hashcodes oder andere Revisionsangaben verwenden (siehe auch Abschnitt 3.12, »Referenzen auf Commits«).

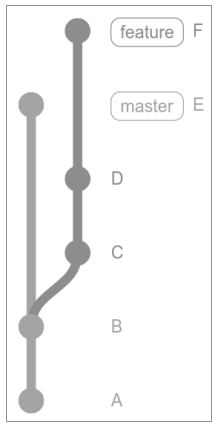

Abbildung 4.3 Commits in zwei Zweigen

Als Ausgangspunkt für die folgenden Beispiele gilt die in Abbildung 4.3 dargestellte Commit-Abfolge, wobei die Commit-Messages einfach `A`, `B`, `C` usw. lauten. Momentan ist der Zweig `master` aktiv. Ohne Range-Syntax werden jeweils alle Commits bis zurück zum initialen Commit `A` angezeigt:

```
git checkout master
git log --oneline
```

```
  ebdb53f (HEAD -> master) E
  c9bb505 B
  45c6cd4 A

git log --oneline feature
  35870e2 (feature) F
  9ae3fb8 D
  b115d39 C
  c9bb505 B
  45c6cd4 A
```

`git log master..feature` zeigt nur die nicht mit `master` zusammengeführten Commits des Feature-Zweigs. Die gemeinsame Basis fällt weg (hier also die Commits `A` und `B`). Anstelle von `master..feature` gibt es zwei alternative Schreibweisen, die eigentlich syntaktisch klarer sind, in der Praxis aber selten vorkommen:

```
git log --oneline master..feature
git log --oneline feature --not master  (gleichwertig)
git log --oneline feature ^master       (auch gleichwertig)
  35870e2 (feature) F
  9ae3fb8 D
  b115d39 C
```

`git log master...feature` mit drei Punkten funktioniert wie das obige Kommando, berücksichtigt aber zusätzlich die seit der Trennung der Zweige in `master` durchgeführten Commits. (Zum selben Ergebnis kommen Sie übrigens auch, wenn Sie die Branch-Namen vertauschen.)

```
git log --oneline master...feature
git log --oneline feature...master      (gleichwertig)
  35870e2 (feature) F
  ebdb53f (HEAD -> master) E
  9ae3fb8 D
  b115d39 C
```

Commits zeitlich eingrenzen

Anstelle der im vorigen Abschnitt vorgestellten Range-Syntax, die den Commit-Bereich anhand logischer Kriterien einschränkt, können Sie die von `git log` gelieferten Commits mit Optionen auch zeitlich eingrenzen:

- `--since <date>` bzw. `--after <date>` zeigt nur Commits, die nach `<date>` entstanden sind.
- `--until <date>` bzw. `--before <date>` zeigt nur Commits, die vor/bis `<date>` durchgeführt wurden.

Wenn Sie die im Mai 2020 entstandenen Commits ansehen möchten, führen Sie das folgende Kommando aus:

```
git log  --after 2020-05-01 --until 2020-05-31
```


Commits sortieren

Standardmäßig werden Commits durch `git log` zeitlich sortiert, der neueste Commit zuerst. Das ändert sich allerdings, sobald Sie die Option `--graph` hinzufügen. `git log` bündelt jetzt zusammengehörende Commits. Wenn Sie die Commits trotz `--graph` im zeitlichen Ablauf ordnen wollen, verwenden Sie die Zusatzoption `--date-order`. Umgekehrt können Sie die Gruppierung der Commits nach Zweigen auch ohne `--graph` mit `--topo-order` erreichen.

Die folgenden Beispiele beziehen sich wieder auf Abbildung 4.3. Allerdings wurden die Zweige mit Merge verbunden:

```
git checkout master
git merge feature
```

Normalerweise ordnet `git log` die Commits streng chronologisch. (Die Option `--pretty` ermöglicht hier eine einzeilige Darstellung samt Commit-Datum. Zur Verbesserung der Übersicht haben wir die Originalausgaben ein wenig umformatiert und Wochentage und Jahreszahlen entfernt.)

```
git log --pretty=format:"%h %cd %s" --date=local

  52003e9  Jun 13 07:06:25     Merge branch 'feature'
  35870e2  Jun 10 10:32:56     F
  ebdb53f  Jun 10 10:32:38     E
  9ae3fb8  Jun 10 10:32:04     D
  b115d39  Jun 10 10:30:36     C
  c9bb505  Jun 10 10:29:24     B
  45c6cd4  Jun 10 10:29:16     A
```

Mit der Option `--graph` werden die Commits `C`, `D` und `F` gruppiert:

```
git log --pretty=format:"%h %cd %s" --date=local --graph

  *    52003e9  Jun 13 07:06:25    Merge branch 'feature'
  |\
  | *  35870e2  Jun 10 10:32:56    F
  | *  9ae3fb8  Jun 10 10:32:04    D
  | *  b115d39  Jun 10 10:30:36    C
  * |  ebdb53f  Jun 10 10:32:38    E
  |/
  *    c9bb505  Jun 10 10:29:24    B
  *    45c6cd4  Jun 10 10:29:16    A
```

Die Option `--date-order` stellt trotz Zweigdarstellung die ursprüngliche Ordnung wieder her:

```
git log --pretty=format:"%h %cd %s" --date=local --graph \
        --date-order

  *    52003e9  Jun 13 07:06:25    Merge branch 'feature'
  |\
  | *  35870e2  Jun 10 10:32:56    F
  * |  ebdb53f  Jun 10 10:32:38    E
  | *  9ae3fb8  Jun 10 10:32:04    D
  | *  b115d39  Jun 10 10:30:36    C
  |/
  *    c9bb505  Jun 10 10:29:24    B
  *    45c6cd4  Jun 10 10:29:16    A
```

Author Date versus Commit Date

Zusammen mit jedem Commit werden *zwei* Zeitangaben gespeichert, das *Author Date* und das *Commit Date*. Normalerweise stimmen beide Zeitangaben überein. Bei Commits, die durch Rebasing verändert wurden, ist das aber nicht der Fall: Dann gibt das Author Date den Zeitpunkt an, zu dem der ursprüngliche Commit entstanden ist. Das Commit Date verweist auf den Zeitpunkt des Rebasing.

Wenn Sie beim Sortieren der Commits das Author Date berücksichtigen wollen, verwenden Sie die Option `--author-date-order`. Die Commits werden nun wie bei `--topo-order` gruppiert, innerhalb der Zweige (von denen es dank Rebasing üblicherweise weniger oder gar keine gibt) wird aber das Author Date als Sortierkriterium verwenden.

Markierte Commits (git tag)

`git tag` liefert eine Liste aller Tags. `git tag <pattern>` schränkt das Ergebnis auf Tags ein, die dem Suchmuster entsprechen. Sobald Sie das gewünschte Tag ermittelt haben, können Sie sich mit `git log <tagname>` die Commits ansehen, die zu diesem Release geführt haben.

Alternativ können Sie mit `git log --simplify-by-decoration` nur solche Commits anzeigen, die Tags enthalten oder auf die ein Zweig verweist. In großen Repositorys ist das aber vergleichsweise langsam.

`git log` zeigt normalerweise keine Tags an. Wenn Sie diese Zusatzinformation wünschen, übergeben Sie an `git log` die Option `--decorate`. Wenn Sie dennoch eine kompakte Anzeige wünschen, können Sie `--decorate` wie bisher mit `--oneline` kombinieren.

Referenzlog (git reflog)

Immer wenn von der Commit-Abfolge (also dem *Commit Log*) die Rede ist, müssen wir auch auf das Referenz-Log hinweisen: Es enthält alle lokal durchgeführten Kommandos, die den globalen HEAD oder den Head eines Zweiges verändert haben. Das Kommando `git reflog` listet diese Aktionen samt den Hashcodes der Commits auf:

```
git reflog

  ebdb53f (HEAD -> master) HEAD@{0}: checkout: moving from
                                     feature to master
  35870e2 (feature) HEAD@{1}: commit: F
  9ae3fb8 HEAD@{2}: checkout: moving from master to feature
  ebdb53f (HEAD -> master) HEAD@{3}: commit: E
  c9bb505 HEAD@{4}: checkout: moving from feature to master
  9ae3fb8 HEAD@{5}: commit: D
```

Wenn Sie die detaillierte Ausgabe von `git log` wünschen, aber gleichzeitig genau die Commits sehen möchten, die `git reflog` liefert, führen Sie `git log` mit der Option `--walk-reflog` aus:

```
git log --walk-reflogs

  commit ebdb53f0db624c6dd4d754940903c3be905a9be (HEAD -> master)
  Reflog: HEAD@{0} (Michael Kofler <...>)
  Reflog message: checkout: moving from feature to master
  Author: Michael Kofler <...>
  Date:   Wed Jun 10 10:32:38 2020 +0200

      E

  commit 35870e24fb49bb77622e17f5844cfaeb515c0a00 (feature)
  Reflog: HEAD@{1} (Michael Kofler <...>)
  Reflog message: commit: F
  Author: Michael Kofler <...>
  Date:   Wed Jun 10 10:32:56 2020 +0200

      F
```

Anstelle von `--walk-reflog` können Sie auch die Option `--reflog` verwenden. Damit wird jeder Commit nur einmal angezeigt. (Bei `--walk-reflog` kann der gleiche Commit mehrfach auftauchen, z. B. immer dann, wenn Sie zuvor mit `git checkout` den Zweig gewechselt haben.)

4.2 Dateien durchsuchen

Während wir uns Abschnitt 4.1, »Commits durchsuchen (git log)«, darauf konzentriert haben, die Metadaten eines Repositorys zu durchsuchen, ist nun der Inhalt an der Reihe: Welchen Inhalt hatte eine bestimmte Datei zu einem früheren Zeitpunkt? Was hat sich seither geändert? Und wer ist dafür verantwortlich? Bei der Beantwortung dieser und weiterer Fragen helfen ein ganzes Bündel von Kommandos, unter anderem `git show`, `git diff` und `git blame`.

Alte Versionen einer Datei ansehen (git show)

Das Kommando `git show <revision>:<file>` haben wir in Abschnitt 3.4, »Commit-Undo«, schon vorgestellt: Es gibt die Datei `<file>` in dem Zustand aus, den sie hatte, als der Commit `<revision>` aktuell war. Wenn Sie also Version 2.0 Ihres Programms mit dem Tag `v2.0` gekennzeichnet haben und wissen wollen, wie die Datei `index.php` damals aussah, führen Sie das folgende Kommando aus:

```
git show v2.0:index.php
```

Natürlich können Sie die Ausgabe auch in eine andere Datei umleiten, damit Sie beide Versionen (die aktuelle und die alte) parallel zur Verfügung haben:

```
git show v2.0:index.php > old_index.php
```

Unterschiede zwischen Dateien ansehen (git diff)

Wenn Sie wissen möchten, was sich zwischen der aktuellen Version und einer alten Version einer Datei geändert hat, verwenden Sie `git diff`. Das folgende Programm zeigt an, wie sich die Datei `index.php` seit der Version 2.0 geändert hat. Die Ausgabe besteht aus mehreren Blöcken, die mit `@@` eingeleitet werden und die Position angeben. Zur Orientierung helfen einige Zeilen Code, den Kontext herzustellen. Anschließend folgen die geänderten Zeilen, denen - oder + vorangestellt ist, je nachdem, ob sie gelöscht oder hinzugefügt wurden. (Im Terminal sind die gelöschten Zeilen rot und die hinzugefügten Zeilen grün hervorgehoben, was in diesem Buch leider nicht dargestellt werden kann.)

```
git diff v2.0 index.php

  diff --git a/index.php b/index.php
  index a41783c..d1e3af2 100644
  --- a/index.php
  +++ b/index.php
  @@ -10,9 +10,9 @@ try {
     exit();
   }
```



```
  -try {
  -  $ctl->checkAccess();
  -} catch (Exception $e) {
  +if ($ctl->checkAccess() === TRUE) {
  +  $ctl->showRequestedPage();
  +} else {
     if ($ctl->isJSONRequest()) {
       $data = new stdClass();
       $data->error = true;
  @@ -29,4 +29,3 @@ try {
       exit();
     }
   }
  -$ctl->showRequestedPage();
```

Wenn Sie nur am Umfang der Änderungen interessiert sind, übergeben Sie zusätzlich die Option `--compact-summary`:

```
git diff --compact-summary v2.0 index.php
 index.php | 7 +++----
 1 file changed, 3 insertions(+), 4 deletions(-)
```

Der Befehl `git diff <revision1>..<revision2> <file>` zeigt die Änderungen zwischen zwei alten Versionen an:

```
git diff --compact-summary v1.0..v2.0 index.php
```

Natürlich können Sie an `git diff` anstelle von Tags bzw. Versionen auch die Hashcodes von Commits, die Namen von Zweigen oder sonstige Referenzen übergeben (siehe Abschnitt 3.12, »Referenzen auf Commits«). Beachten Sie, dass die ausgesprochen praktische Schreibweise `HEAD@{2.weeks.ago}` zur zeitlichen Einordnung nur für lokal durchgeführte Commits funktioniert, also nur für Aktionen, die im Reflog gespeichert sind. Davon abgesehen gibt es keine Möglichkeiten, den Vergleichs-Commit zeitlich festzulegen. Gegebenenfalls müssen Sie zuerst mit `git log` einen zeitlich passenden Commit suchen und dessen Hashcode dann an `git diff` übergeben.

Range-Syntax mit drei Punkten

Die Variante `git diff <rev1>...<rev2>` ist vor allem dann zweckmäßig, wenn es sich bei den Revisionen um Zweige handelt. In diesem Fall ermittelt `git diff` zuerst die letzte gemeinsame Basis beider Zweige und zeigt dann an, was sich in `<rev2>` im Vergleich zum letzten gemeinsamen Commit verändert hat. Anders als bei `<rev1>..<rev2>` werden aber alle Änderungen ignoriert, die seither in `<rev1>` passiert sind.

Unterschiede zwischen Commits ansehen

Wenn Sie bei `git diff` auf die Angabe einer Datei verzichten, zeigt es *alle* geänderten Dateien seit der angegebenen Version bzw. zwischen zwei Versionen/Commits an. Wiederum ist die Option `--compact-summary` hilfreich, wenn Sie vorerst nur einen Überblick gewinnen möchten.

Bei umfangreichen Änderungen fehlt der Platz, um für jede geänderte Zeile ein + oder ein - auszugeben. Stattdessen wird nach | die Gesamtanzahl der geänderten Zeilen angegeben. Die Anzahl der Plus- und Minus-Zeichen ist relativ zu der Datei mit den größten Änderungen. Je länger der Balken aus den Zeichen ist, desto umfangreicher sind die Änderungen ausgefallen.

```
git diff --compact-summary v1.0..v2.0 index.php

  css/autocompleteList.css                  |   225 +-
  css/editproject.css (new)                 |    13 +
  css/edituser.css                          |    99 +-
  css/iprot.css                             |   648 ++++-
  css/iprot/jquery-ui-1.8.13.custom.css     |     2 +-
  css/mobile.css (new)                      |    17 +
  ...
  269 files changed, 22819 insertions(+), 12792 deletions(-)
```

Selten sind Sie einfach an allen Änderungen interessiert. Zwei Optionen helfen dabei, das Ergebnis gezielt einzuschränken:

- Mit `-G <pattern>` geben Sie ein Suchmuster (einen regulären Ausdruck) an. `git diff` liefert dann nur die Textdateien, deren Änderungen den Suchausdruck enthalten, wobei die Groß- und Kleinschreibung exakt übereinstimmen muss.
- `--diff-filter=A|C|D|M|R` filtert jene Dateien heraus, die hinzugefügt (*added*), kopiert (*copied*), gelöscht (*deleted*), verändert (*modified*) oder umbenannt (*renamed*) wurden.

Das folgende Kommando liefert die Dateien, die zwischen Version 1.0 und 2.0 verändert wurden und in deren Code der Suchtext *PDF* vorkommt.

```
git diff  -G PDF --diff-filter=M --compact-summary v1.0..v2.0
```

Änderungen seit dem letzten Commit

Bevor Sie `git commit` ausführen, ist es oft eine gute Idee, sich einen Überblick über die Änderungen in allen für den Commit vorgemerkten Dateien zu verschaffen. Genau das macht `git diff --staged`:

Sollten Sie `git add` noch nicht ausgeführt haben bzw. vorhaben, `git commit -a` zu verwenden, zeigt `git diff` ohne irgendwelche weiteren Parameter alle zuletzt durchgeführten Änderungen an. (Nicht berücksichtigt werden neue Dateien, die noch nicht unter Versionskontrolle stehen.)

Dateien durchsuchen (git grep)

An welchen Stellen in den zahlreichen Dateien aus Ihrem riesigen Projekt wird die Funktion *X* aufgerufen oder ein Objekt der Klasse *Y* erzeugt? Antwort auf derartige Fragen gibt `git grep <pattern>`. Standardmäßig berücksichtigt das Kommando alle Dateien im Projektverzeichnis und listet die Zeilen auf, in denen der Suchausdruck in exakter Groß- und Kleinschreibung auftritt. (Wenn Sie nicht zwischen Groß- und Kleinschreibung differenzieren wollen, geben Sie zusätzlich die Option `-i` an.)

```
git grep SKAction
  ios-pacman/Maze.swift:   let setGlitter = SKAction.setTextur...
  ios-pacman/Maze.swift:   let setStandard  = SKAction.setText...
  ios-pacman/Maze.swift:   let waitShort = SKAction.wait(forDu...
  ...
```

Ein kompakteres Suchergebnis erhalten Sie mit `--count`. In diesem Fall zeigt `git grep` nur an, wie oft der Suchausdruck in den jeweiligen Dateien vorkommt:

```
git grep --count CGSize
  ios-pacman/CGOperators.swift:6
  ios-pacman/Global.swift:1
  ios-pacman/Maze.swift:4
  ...
```

Durch die Angabe von Dateien oder Verzeichnisse können Sie die Suche einschränken. Das folgende Kommando durchsucht die Dateien im Verzeichnis `css` nach dem Schlüsselwort `margin`. Wegen der Option `-n` wird zu jeder Fundstelle auch die Zeilennummer angegeben.

```
git grep -n margin css/
  css/config.json:100:    "@form-group-margin-bottom": "15px",
  css/config.json:144:    "@navbar-margin-bottom": "@line-heig...
  css/editglobal.css:25:  margin-top: 1px;
  css/editglobal.css:29:  margin-top: 0px;
  ...
```

Natürlich können Sie auch alte Versionen Ihres Codes durchsuchen, indem Sie die gewünschte Revision vor den Dateinamen oder Verzeichnissen angeben. Wenn der Suchausdruck wie im folgenden Beispiel Sonder- oder Leerzeichen enthält, müssen

Sie ihn zwischen Apostrophe stellen. Das folgende Beispiel sucht in Version 2.0 des Programms nach UPDATE-Kommandos, die die Tabelle person verändern:

```
git grep 'UPDATE person' v2.0
  v2.0:lib/delete.php:          $sql = "UPDATE person SET sta...
  v2.0:lib/person.php:          $sql = sprintf("UPDATE person...
  v2.0:lib/personengruppe.php: $sql = sprintf("UPDATE person...
  ...
```

Schwierig ist die Anwendung von git grep, wenn Sie nicht wissen, in welchem Commit Sie suchen sollen bzw. wenn es sich um Änderungen handelt, die nur vorübergehend durchgeführt und später wieder aus der Codebasis entfernt wurden. In solchen Fällen können Sie mit git rev-list v1.0..v2.0 eine Liste mit den Hashcodes aller Commits für den fraglichen Zeitraum erstellen. Diese Liste verarbeiten Sie dann mit git grep.

Beispielsweise zählt das folgende Kommando, wie oft das SQL-Schlüsselwort UPDATE in diversen Versionen der Datei lib/kapitel.php vorkommt. Wie bei git log wird der neueste Commit zuerst berücksichtigt. Die Zeichen -- trennen die durch git rev-list erzeugte Hashcode-Liste vom Dateinamen.

```
git grep -c 'UPDATE'  $(git rev-list v1.0..v2.0) -- user.php
  262d67fed686cda939092e7b0cb337bbc1e2dbe9:user.php:5
  96d0a06d389784ec93f252a097185ee3678a2c1c:user.php:5
  c07c2f0ce5682bea898ba3a65a15bf5230dd23dc:user.php:4
  ...
```

Urheberschaft von Code herausfinden (git blame)

Wenn Sie mit den hier beschriebenen Kommandos die Datei gefunden haben, die Sie eigentlich interessiert, ist die nächste Frage natürlich: Wer ist für den dort enthaltenen Code verantwortlich? Ein großartiges Hilfsmittel ist in diesem Fall git blame <file>. Ohne weitere Optionen zeigt es die betreffende Datei zeilenweise an und gibt bei jeder Zeile an, in welchem Commit von welchem Autor zu welchem Datum diese Zeile verändert wurde (siehe Abbildung 4.4).

Mit der Option -L 100,200 berücksichtigen Sie nur die Zeilennummern 100 bis 200. Eine große Hilfe beim Lesen der Ausgaben sind die beiden folgenden Optionen:

- --color-lines stellt Fortsetzungszeilen aus dem gleichen Commit in blauer Farbe dar.
- --color-by-age kennzeichnet frisch geänderten Code rot (Änderungen im letzten Monat) und mäßig neuen Code weiß (Änderungen im letzten Jahr).

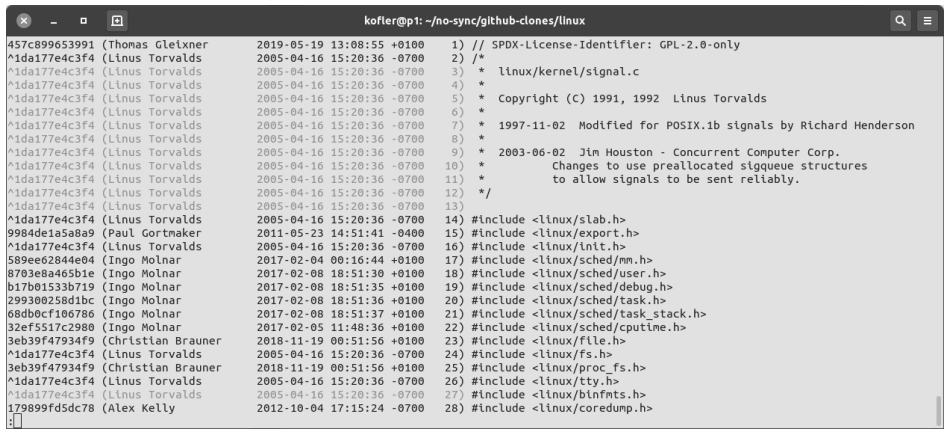

Abbildung 4.4 Urheberschaft der Datei »signal.c« des Linux-Kernels

Eine noch übersichtlichere Darstellung der Blame-Ergebnisse bieten die Websites von GitLab, GitHub und Co. Außerdem können Sie dort sich per Mausklick direkt den betreffenden Commit ansehen.

Boundary Commits

Wenn im lokalen Repository nicht alle Commits enthalten sind, kommt es vor, dass einzelnen Hashcodes das Zeichen ^ (*Caret* bzw. *Circumflex*) vorangestellt wird, z. B. ^1da177e4c3f4. Es weist auf einen *Boundary Commit* hin, also auf den letzten im Repository verfügbaren Commit.

4.3 Fehler suchen (git bisect)

Stellen Sie sich vor, Sie bemerken, dass in einem Feature Ihres Programms ein Fehler auftritt, aber es gelingt Ihnen nicht, dessen Ursache zu finden oder auch nur einzugrenzen. Vermutlich handelt es sich um eine Wechselwirkung, die erst durch Änderungen in mehreren Dateien entstanden ist.

Sie sind sich sicher, dass der Fehler früher nicht aufgetreten ist. Mit git checkout v1.5 sind Sie vorübergehend zur Version 1.5 zurückgekehrt und haben diese nochmals getestet. Dort ist die Welt noch in Ordnung. Seither gab es 357 Commits. (Das Kommando git rev-list ist eine einfachere Variante zu git log, das normalerweise anstelle von Commit-Messages einfach nur die Hashcodes der betreffenden Commits liefert. Mit der Option --count zählt es die Commits zwischen zwei Punkten eines Zweigs.)

```
git rev-list v1.5..HEAD --count
  357
```

Um herauszufinden, was den Fehler verursacht, müssen Sie den ersten Commit finden, in dem der Fehler auftritt. Das klingt nach der sprichwörtlichen Suche nach einer Nadel im Heuhaufen.

Glücklicherweise unterstützt Sie `git bisect` bei dem Unterfangen. Die Idee von `git bisect` besteht darin, dass Sie zuerst den letzten bekannten »guten« und »schlechten« Commit angeben – in diesem Beispiel den Commit mit dem Tag `v1.5` sowie den aktuellen Commit (also HEAD). `git bisect` führt nun einen Checkout in der Mitte des Commit-Bereichs aus, halbiert also den Suchbereich. (Damit liegt der Fall eines *Detached HEADS* vor, d. h. HEAD verweist nicht auf das Ende eines Zweigs, sondern auf irgendeinen Commit in der Vergangenheit.)

```
git bisect start
git bisect bad HEAD
git bisect good v1.5
  Bisecting: 178 revisions left to test after this
    (roughly 8 steps)
  [e84fd83319c1280bcef38400299fd55925ea25e6] Merge branch ...
```

Jetzt liegt es an Ihnen zu testen, ob der Fehler bei diesem Commit noch immer auftritt. *Wie* Sie diesen Test durchführen, hängt ganz von der Art des Codes ab. Eventuell müssen Sie Ihr Programm kompilieren, um es zu testen. Bei einer Webapplikation reicht dagegen ein Test im Browser. Je nachdem, wie der Test ausfällt, melden Sie das Ergebnis mit `git bisect bad` oder mit `git bisect good`:

```
git bisect bad
  Bisecting: 89 revisions left to test after this
    (roughly 7 steps)
  [cea22541893ded6e6e9f6a9d40bf6d0c2ec806d8] bugfix xy ...
```

Abhängig von Ihrer Antwort weiß `git bisect` nun, ob es in der oberen oder unteren Hälfte des Commit-Bereichs weitersuchen soll. Das Kommando führt einen weiteren Checkout in der Mitte des verbleibenden Suchbereichs aus. Der Suchbereich wurde damit auf ca. ein Viertel reduziert.

Abermals müssen Sie nun den Test wiederholen, ob der Fehler noch auftritt oder nicht, und diese Information an `git` weiterleiten. Auf diese Weise fahren Sie fort, bis `git bisect` schließlich meldet:

```
git bisect good
  4127d9d06ecbae0d4d9babaaa8aacebc0c8853cb is the first bad
  commit ...
```


Damit wissen Sie, zu welchem Zeitpunkt in der Vergangenheit der Fehler erstmals aufgetreten ist. Die Suche nach der Ursache des Fehlers steht jetzt noch aus – aber eigentlich sollte `git diff HEAD^`, also die Zusammenfassung der Änderungen im Vergleich zu vorigen Commit, Sie auf die richtige Spur bringen.

Mit `git bisect reset` beenden Sie schließlich `git bisect` und kehren zurück zum Head des Zweiges, in dem Sie sich zu Beginn der Suche befanden. Dort versuchen Sie nun, den jetzt eingegrenzten Fehler endgültig zu beheben.

```
git bisect reset
  Previous HEAD position was ef81d5c fix: getLink for csv ...
  Switched to branch 'develop'
```

4.4 Statistik und Visualisierung

Bei großen Repositorys sieht man oft den Wald vor lauter Bäumen (in unserem Fall eigentlich: vor lauter Zweigen) nicht mehr. In diesem Abschnitt stellen wir Ihnen `git`-Kommandos sowie diverse Werkzeuge vor, mit denen Sie wieder den Durchblick erlangen.

Einfache Zahlenspiele (git shortlog)

Ein praktisches Kommando, um einen ersten Überblick zu erhalten, ist `git shortlog`. In seiner einfachsten Form liefert es eine alphabetisch geordnete Liste aller Commit-Autoren, wobei zu jedem Autor die Anzahl der Commits sowie jeweils die erste Zeile jeder Commit-Message angegeben wird.

Durch diverse Optionen können Sie die Ausgabe weiter verkürzen. Das folgende Kommando liefert eine Liste der Entwickler und Entwicklerinnen des Linux-Kernels, die seit Anfang 2019 die meisten Commits aufzuweisen haben, wobei Merge-Commits nicht gerechnet werden:

```
git shortlog --summary --numbered --email --no-merges \
             --since 2019-01-01

  1488  Chris Wilson <chris@chris-wilson.co.uk>
  1104  Christoph Hellwig <hch@lst.de>
  1065  YueHaibing <yuehaibing@huawei.com>
   875  Thomas Gleixner <tglx@linutronix.de>
   852  Takashi Iwai <tiwai@suse.de>
   799  Colin Ian King <colin.king@canonical.com>
   ...
```

Die Gesamtanzahl aller Commits (über alle Zweige) ermitteln Sie mit `git rev-list`:

```
git rev-list --all --count
  917418
```

Die Anzahl der Dateien im aktuellen Zweig ermitteln Sie, indem Sie die Ausgabe von `git ls-files` an `wc` (*word count*) weiterleiten:

```
git ls-files | wc -l
  67975
```

Analog können Sie auch die Anzahl der Branches und Tags herausfinden:

```
git branch -a | wc -l
  3

git tag | wc -l
  652
```

Den Umfang der Änderungen zwischen zwei Versionen/Zweigen/Revisionen Ihres Projekts können Sie mit `git diff --shortstat` ermitteln:

```
git diff --shortstat v5.5..v5.6
   11533 files changed
  600555 insertions(+)
  285511 deletions(-)
```

Statistik-Tools und -Scripts

Das Internet ist voll von Scripts und Programmen, die aus einem Git-Repository mehr Details als die obigen Kommandos herausholen können. Einen guten Startpunkt bietet der folgende Stack-Overflow-Artikel:

https://stackoverflow.com/questions/1828874

Beliebt und unter Linux einfach anzuwenden ist das Python-Script `gitstats`. Nach der Installation übergeben Sie an das Script den Pfad zum Repository sowie ein Verzeichnis, in dem die Ergebnisdateien gespeichert werden sollen. Ausgehend von der Datei `index.html` können Sie sich in einem Webbrowser dann diverse statistische Auswertungen ansehen. Das Erscheinungsbild der dazugehörenden Grafiken ist allerdings ein wenig minimalistisch.

```
sudo apt install gnuplot
git clone git://repo.or.cz/gitstats.git
mkdir result
gitstats/gitstats <path/to/repo> results/
google-chrome results/index.html
```


Zweige visualisieren

Gerade in Git-Schulungen oder bei dem Versuch, Kollegen die Funktionsweise von Git zu verdeutlichen, besteht der Wunsch, die über mehrere Zweige verteilten Commits »ordentlich« zu visualisieren. Die Ergebnisse von `git log --graph` sind dazu aber ungeeignet.

Schon ein wenig besser ist die Darstellung durch das auf vielen Rechnern installierte Programm `gitk` (siehe Abbildung 4.5). Es wird üblicherweise aus dem Terminal heraus gestartet und zeigt die Commit-Abfolge für den gerade aktuellen Zweig.

Abbildung 4.5 Visualisierung von Zweigen durch »gitk«

Falls Sie Wert auf eine übersichtlichere Darstellung von Zweigen legen, haben wir ein paar Vorschläge für Sie:

- Das kommerzielle Programm *GitKraken* zeigt nicht nur die Commit-Abfolge in einer ansprechenden Form an (siehe Abbildung 4.6), sondern bietet eine Menge weiterer Funktionen, die bei der Administration von Git-Repositorys helfen. Die kostenlose Version kann nur für öffentliche Repositorys verwendet werden.

- Auch manche Git-Plattformen enthalten Visualisierungsfunktionen. Beispielsweise zeigt GitLab auf der Teilseite REPOSITORY • GRAPH eine übersichtliche Darstellung des Commit-Verlaufs (siehe Abbildung 4.7).
- GitHub-Anwender, die diesbezüglich weniger verwöhnt sind, sollten sich das kommerzielle Projekt GFC (*Git Flow Chart*) ansehen: Die Website *https://gfc.io* kann die Commit-Abfolge von Repositorys auf GitHub und Bitbucket visualisieren. Die Grundfunktionen stehen für öffentliche Repositorys kostenlos zur Verfügung. Wenn Sie GFC für private Repositorys bzw. in Kombination mit den GitHub-Teamfunktionen einsetzen wollen, müssen Sie einen monatlichen Obolus leisten.

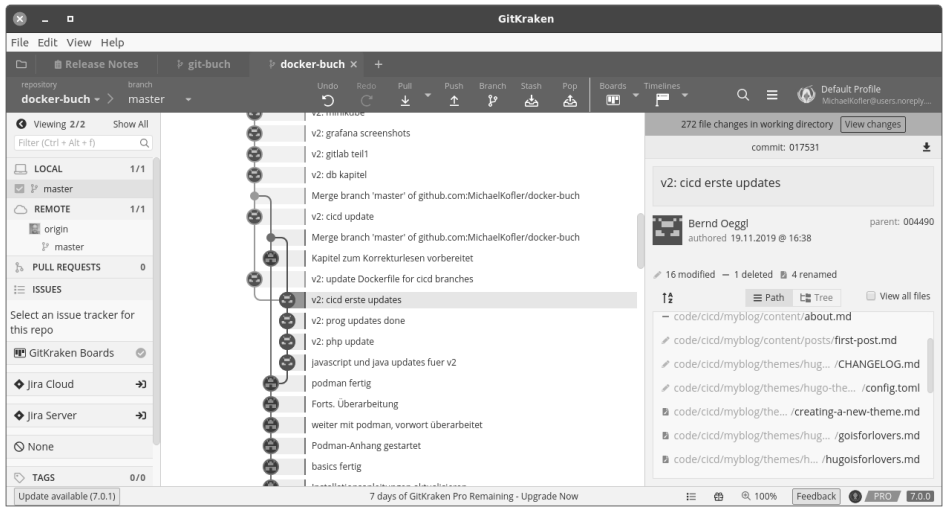

Abbildung 4.6 Das Programm »GitKraken«

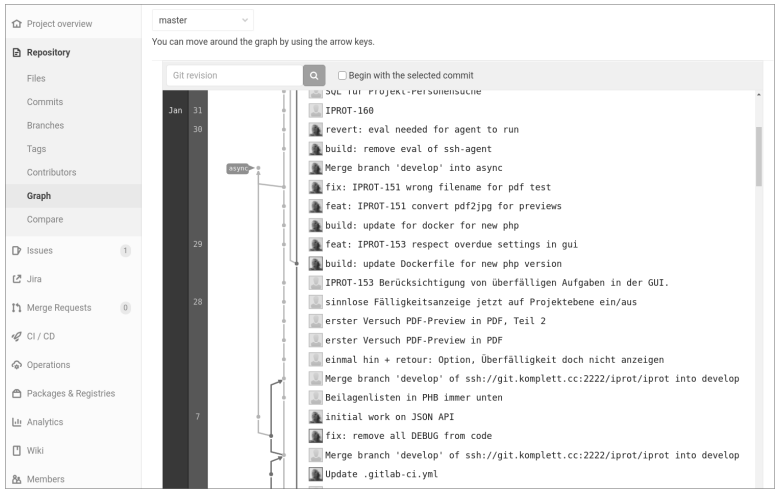

Abbildung 4.7 Darstellung von Zweigen in GitLab

GitGraph.js

Vielleicht ist es Ihnen aufgefallen: Alle Abbildungen in diesem Buch, die die Commit-Abfolge mehrerer Zweige zeigen, haben ein einheitliches Aussehen. Das ist natürlich kein Zufall. Mit der Open-Source-Bibliothek *GitGraph.js* und wenigen Zeilen eigenem JavaScript-Code lassen sich viele Visualisierungswünsche erfüllen. Das Ergebnis ist dann im Webbrowser zu bewundern. Werfen Sie einen Blick auf die Projektwebsite *https://gitgraphjs.com*, die elementare Arbeitstechniken in Form einer Präsentation zusammenfasst!

Leider ist GitGraph.js nicht in der Lage, echte Commits aus einem Repository zu zeichnen. Sie müssen die Commit-Struktur also durch entsprechende JavaScript-Anweisungen zusammensetzen, was mit einigem Aufwand verbunden ist (den wir für dieses Buch natürlich nicht gescheut haben).

Die folgenden Zeilen zeigen den Code für Abbildung 4.3. `graphContainer` verweist auf die Stelle im HTML-Code, wo das Diagramm dargestellt werden soll. `mytemplate` enthält einige Optionen, um Commits ohne Autoren und Hashcodes, Zweige aber mit ihren Namen anzuzeigen. `createGitGraph` erzeugt die vorerst leere Commit-Abfolge. Mit `branch` und `commit` werden dann Commits und Zweige hinzugefügt.

```
<!doctype html>
<html><head>
<script src="https://cdn.jsdelivr.net/npm/@gitgraph/js">
</script>
</head>
<body>
<div id="graph"></div>
<script>
const graphContainer = document.getElementById("graph");
const mytemplate =  GitgraphJS.templateExtend(
   GitgraphJS.TemplateName.Metro, {
        commit: {  message: { displayAuthor: false,
                              displayHash: false    } },
        branch: {  label:   { display: true } }
      });
const gitgraph = GitgraphJS.createGitgraph(
  graphContainer,
  { author: " ", template: mytemplate } );
const master  = gitgraph.branch("master").commit("A").commit("B")
const develop = master.branch("feature").commit("C").commit("D")
master.commit("E")
develop.commit("F")
</script>
</body></html>
```

10.5 Ein Blog mit Git und Hugo

Von Git zum Blogsystem – das heißt, den inhaltlichen Bogen dieses Buchs schon beinahe zu überspannen. Keine Angst, wir werden Ihnen gleich zeigen, dass sich Git in Kombination mit bestimmten Blogsystemen sehr gewinnbringend einsetzen lässt und den Blog-Arbeitsablauf dramatisch vereinfachen kann. Das gilt insbesondere dann, wenn Sie mit Markdown vertraut sind, beim Schreiben einen eher technischen Ansatz vorziehen und keinen Bedarf an überladenen Weboberflächen zur CMS-Administration haben. Außerdem gibt uns dieser Abschnitt die Möglichkeit, die interessante Git-Erweiterung *Git LFS* vorzustellen. LFS steht für *Large File Storage*.

Von WordPress zu Hugo

Wenn man heute über Software für Blogs oder *Content Management Systems* (CMS) spricht, kommt schnell *WordPress* ins Spiel: Diese PHP/MySQL-Software hat einen wahren Siegeszug hingelegt und ist aktuell das am weitesten verbreitete CMS.

Doch die Webtechnologie hat sich weiterentwickelt, und das serverseitige Erzeugen von Webseiten, wie es WordPress mit PHP und MySQL macht, ist nicht mehr in allen Bereichen State of the Art. Zunehmend beliebt sind *Single-Page Applications*. Sie entlasten den Server und verlagern einen Teil der Rechenleistung mittels JavaScript auf den Client. REST-APIs liefern die Daten im JSON-Format an das Frontend.

In diesem Abschnitt lassen wir PHP und JavaScript freilich links liegen und stellen Ihnen eine weitere Webtechnologie vor, die in den letzten Jahren viel Beachtung fand: Mit einem *Static Site Generator* lässt sich Text im Markdown-Format mit der Hilfe von HTML-Vorlagen in eine vollständige Website umwandeln. Navigationselemente, RSS-Feeds, Verlinkungen zu Kategorien und Tags werden alle beim Programmaufruf erzeugt und in statischen Dateien gespeichert.

Was auf den ersten Blick etwas altbacken klingt, bringt große Vorteile mit sich: Die Inhalte können sehr schnell ausgeliefert werden, ohne den Server mit Datenbankabfragen zu belasten. Der wohl größte Vorteil ist aber der enorme Sicherheitsgewinn: Auf dem Server selbst läuft keine Programmiersprache mehr, die Angreifer gerne als Einfallstor verwenden.

Ein prominenter Vertreter dieser Software ist Jekyll, das von Tom Preston-Werner, einem der GitHub-Gründer, bereits 2008 entwickelt wurde. Die Open-Source-Software ist heute noch bei GitHub im Einsatz und kann für GitHub Pages verwendet werden. Andere prominente Vertreter dieser Zunft sind Next.js, Nuxt.js oder Hugo. Während Next.js oder Nuxt.js eigentlich JavaScript-Frameworks für Single-Page Applications sind, können Sie Hugo ganz ohne JavaScript-Kenntnisse verwenden. Das Programm konvertiert blitzschnell Markdown-Dateien in HTML und kann einfach mit Templates und Themes gesteuert werden.

Hugo

Wir haben uns für Hugo entschieden, da wir schon in einem anderen Projekt positive Erfahrungen damit gemacht haben. Es lässt sich rasch installieren und ist sehr effizient im Betrieb. Sie laden einfach das Binary für die Plattform von der GitHub-Projekt-Website:

https://github.com/gohugoio/hugo/releases

Das Kommandozeilenprogramm hat eine Option, die zur Einrichtung eines neuen Blogs dient. Damit starten wir unser kleines Projekt:

```
hugo new site my-blog

  Congratulations! Your new Hugo site is created in /src/my-blog.
  ...
```

Hugo hat eine Verzeichnisstruktur angelegt, in der sich nur zwei Dateien befinden. Der Ordner `my-blog` sieht so aus:

```
|-- archetypes
|   `-- default.md
|-- config.toml
|-- content
|-- data
|-- layouts
|-- static
`-- themes
```

Wir initialisieren ein neues Git-Repository darin, denn wir wollen alle Schritte unseres Blogs dokumentieren:

```
git init
git add .
git status

On branch master
No commits yet
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   archetypes/default.md
    new file:   config.toml
```

Dabei stoßen wir gleich auf eine Eigenheit von Git: Obwohl wir mit `git add .` alle Einträge im aktuellen Verzeichnis zum Index hinzugefügt haben, werden nur die beiden Dateien `default.md` und `config.toml` für den Commit vorgesehen. Das liegt daran, dass Git nur den Inhalt von Dateien verfolgt; leere Verzeichnisse gehören nicht dazu.

In unserem Fall ist das kein Problem. Wir werden in der lokalen Arbeitskopie weiterarbeiten, und sobald sich die Verzeichnisse mit Inhalt füllen, werden sie automatisch in das Repository aufgenommen. Manchmal möchte man aber explizit ein leeres Verzeichnis in das Repository inkludieren. Zum Beispiel könnte ein Programm im laufenden Betrieb dort Daten hineinschreiben, ohne das Verzeichnis zuvor zu erstellen. Die einzige Lösung für das Problem besteht darin, in den leeren Verzeichnissen Dateien anzulegen. Sie können dazu `.gitignore`-Dateien verwenden, wie es die Git-FAQ empfiehlt (*https://links.gitbuch.info/empty-dir*), aber es eignet sich auch jede andere Datei.

Hugo Themes als Git Submodule

Wie Hugo die Inhalte in HTML und CSS konvertiert, wird durch das verwendete Theme gesteuert. Wir haben uns für das Theme *Beautiful Hugo* entschieden, das sowohl am Desktop als auch auf mobilen Geräten gut und *responsive* funktioniert und unter der freien MIT-Lizenz auf GitHub zu finden ist:

https://themes.gohugo.io/beautifulhugo/

Um das Theme zu verwenden, fügen wir sein Repository als Submodul (siehe Abschnitt 9.3) dem Unterordner `themes` hinzu:

```
git submodule add \
    https://github.com/halogenica/beautifulhugo.git \
    themes/beautifulhugo

  Cloning into '/src/my-blog/themes/beautifulhugo'...
  ...
```

Sollte der Autor das Theme weiter verbessern, haben wir durch die Submodul-Technik die Möglichkeit, das Update einfach auszuprobieren. Mit dem `submodule add`-Aufruf wurde das Theme geklont, und die Änderungen wurden gleich zum Index hinzugefügt. Schließlich stellen wir das Theme in der Konfigurationsdatei ein und probieren es mit dem in Hugo integrierten Webserver aus:

```
echo 'theme = "beautifulhugo"' >> config.toml
hugo serve

  ...
  Web Server is available at http://localhost:1313/ ...
  Press Ctrl+C to stop
```

Wir öffnen die angegebene URL *http://localhost:1313* und sehen das durchaus noch verbesserungsfähige Ergebnis (siehe Abbildung 10.3). Damit ist es Zeit für den ersten Commit. Das Gerüst unseres Blogs ist bereits fertig.

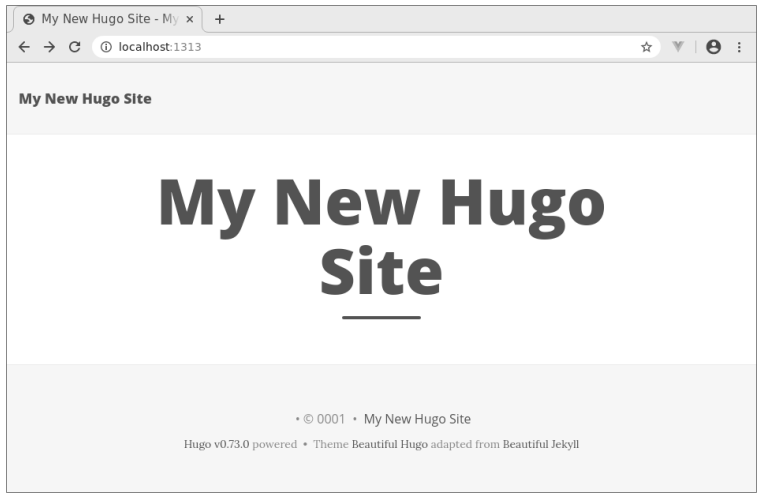

Abbildung 10.3 Das Hugo Theme »Beautiful Hugo« ohne Anpassungen

Wie wir der dem Theme mitgelieferten Beispielseite entnehmen, können wir noch einige Änderungen in der Konfigurationsdatei `config.toml` vornehmen. Wir ergänzen hier unter anderem einen Untertitel, das Datumsformat und Informationen zum Autor. Außerdem stellen wir das Hauptmenü in den Abschnitten `[[menu.main]]` ein.

```
# Datei config.toml
...
theme = "beautifulhugo"

[Params]
  subtitle = "Reisenotizen"
  dateFormat = "2. January 2006"
  ...
[Author]
  name = "bernd"
  github = "git-buch"
  gitlab = "gitbuch"
  ...
[[menu.main]]
  name = "Blog"
  url = ""
  weight = 1
[[menu.main]]
  name = "About"
  url = "pages/about/"
  weight = 2
  ...
```


Blog mit Inhalt füllen

Nun müssen wir uns um den Content kümmern. Unser erster Eintrag dokumentiert z. B. die Reise zur Messe *Intergeo* in Stuttgart im September 2019. Wir verwenden Hugo, um die Struktur für den neuen Eintrag zu erzeugen. Der Eintrag soll im Ordner `posts/2019-09-19` unterhalb des `content`-Ordners liegen.

```
hugo new posts/2019-09-19/index.md

  /src/my-blog/content/posts/2019-09-19/index.md created
```

Obwohl der noch geöffnete Webbrowser die Webseite bei jeder Änderung an Dateien neu lädt und das auch gerade gemacht hat, sehen wir nichts von unserem neuen Eintrag. Schuld ist die Meta-Anweisung `draft: true` im Kopfteil der neu erstellten `index.md`-Datei. Sobald wir diese Zeile löschen oder den Wert von `true` in `false` ändern, erscheint der Eintrag auf der Startseite des Blogs.

Wir kopieren ein Handyfoto von der Messe in den Ordner und ergänzen die Markdown-Datei um ein paar Anekdoten dieser Reise. Bevor wir diese Änderungen per Commit speichern, wenden wir uns dem eingangs erwähnten Git-LFS-Modul zu.

Git LFS

Die Erweiterung *Git Large File Storage (LFS)* entstand aufgrund der Problematik, dass Git mit binären Dateien nicht besonders gut umgehen kann. Speziell wenn es sich um große binäre Dateien handelt, die womöglich schlecht komprimierbar sind und sich häufig ändern, wächst die Größe des Repositorys stark an.

Jetzt kann man natürlich argumentieren, dass große binäre Dateien eben nichts in einem Git-Repository verloren haben. Aber nehmen wir unser Beispiel mit den Fotos und den Blogeinträgen: Würde man Text und Bilder getrennt verwalten und vielleicht auch getrennt sichern, stiege die Gefahr, dass man irgendwann Daten verliert (wir sprechen hier leider aus persönlicher Erfahrung).

Git LFS löst das Problem der zu groß werdenden Repositorys, indem per LFS verwaltete Dateien nicht im Repository selbst, sondern an einem anderen Speicherort abgelegt werden. Die Datei selbst enthält nur einen Verweis auf den Hashcode der Datei (einen *Pointer* in der LFS-Nomenklatur). LFS verwendet dabei den Hash-Algorithmus SHA-256, der wesentlich sicherer ist als das aktuell von Git eingesetzte Verfahren SHA-1 (siehe Abschnitt 3.13, »Git-Interna«).

Als Anwender von `git lfs` bekommen wir von den LFS-Pointern nie etwas zu sehen. Grund dafür ist der ausgeklügelte Filtermechanismus, der die Textdateien durch binären Originalinhalte ersetzt. Damit die Filter in Kraft treten können, müssen wir zuerst Git LFS installieren und aktivieren. Unter Debian oder Ubuntu reicht dazu der Aufruf von `sudo apt install git-lfs`. Installationspakete für alle gängigen Plattfor-

men finden Sie unter *https://github.com/git-lfs/git-lfs/releases*. Um Git LFS für unser Repository zu aktivieren, verwenden wir folgendes Kommando:

```
git lfs install

    Updated git hooks.
    Git LFS initialized.
```

Dabei werden mehrere Schritte ausgeführt. Wird das Kommando zum ersten Mal auf diesem Computer ausgeführt, fügt LFS einen neuen Abschnitt in unsere persönliche Git-Konfigurationsdatei ein:

```
[filter "lfs"]
    clean = git-lfs clean -- %f
    smudge = git-lfs smudge -- %f
    process = git-lfs filter-process
    required = true
```

Der `clean`-Filter speichert den binären Inhalt der Datei in einem Unterordner von `.git/lfs` ab und ersetzt die Originaldatei durch den oben beschriebenen LFS-Pointer. Dieser Vorgang geschieht bei `git add`, also wenn die Datei auf dem Git-Index hinzugefügt wird. Umgekehrt holt der `smudge`-Filter den binären Inhalt aus dem `.git/lfs`-Ordner und ersetzt den Pointer durch die korrekten Inhalte.

Zu den Filtern werden noch Git-Hooks installiert, die sich unter anderem um den Upload und Download der Binärdateien vom LFS-Speicherplatz kümmern. Doch damit genug der Theorie; wir fügen jetzt das Foto zum LFS-Speicher hinzu. Damit das funktioniert, müssen wir Git anweisen, welche Dateitypen mit LFS verwaltet werden sollen.

```
git lfs track '*.jpg'
git add .
git status

  On branch master
  Changes to be committed:
    (use "git restore --staged <file>..." to unstage)
      new file:   .gitattributes
      new file:   content/posts/2019-09-19/index.md
      new file:   content/posts/2019-09-19/intergeo.jpg
```

Wir lassen also Dateien, die auf `.jpg` enden, von LFS verwalten. Bei dem anschließenden `add` und `status` sehen wir keine Veränderung. Das ist auch das besonders Angenehme an Git LFS: Ist LFS einmal eingerichtet, brauchen wir uns um nichts mehr zu kümmern, wir merken gar nicht, dass es aktiv ist.

Für unser lokales Repository bringt LFS noch keinen entscheidenden Vorteil: Alle Veränderungen – auch an den von LFS verwalteten Bildern – bleiben im lokalen Ordner `.git/lfs`. Wir legen jetzt unser Remote Repository bei GitHub an und übertragen den aktuellen Stand dorthin:

```
git remote add origin git@github.com:git-buch/my-blog.git
git push -u origin master

  Uploading LFS objects: 100% (1/1), 1.1 MB | 0 B/s, done.
  Enumerating objects: 21, done.
  ...
   * [new branch]      master -> master
  Branch 'master' set up to track remote branch 'master' from
  'origin'.
```

Wir sehen einen neuen Eintrag in der sonst schon bekannten Ausgabe von `git push`: Mit *Uploading LFS objects* teilt uns Git mit, dass die von LFS verwalteten Objekte getrennt vom restlichen Repository hochgeladen werden. Wie bereits erwähnt, bleibt der Vorgang völlig transparent, und wir merken gar nicht, dass die Bilder in irgendeiner Weise anders verwaltet werden. Einzig der Hinweis in der GitHub-Oberfläche *Stored with Git LFS* zeigt uns, dass das Bild von LFS verwaltet wird (siehe Abbildung 10.4).

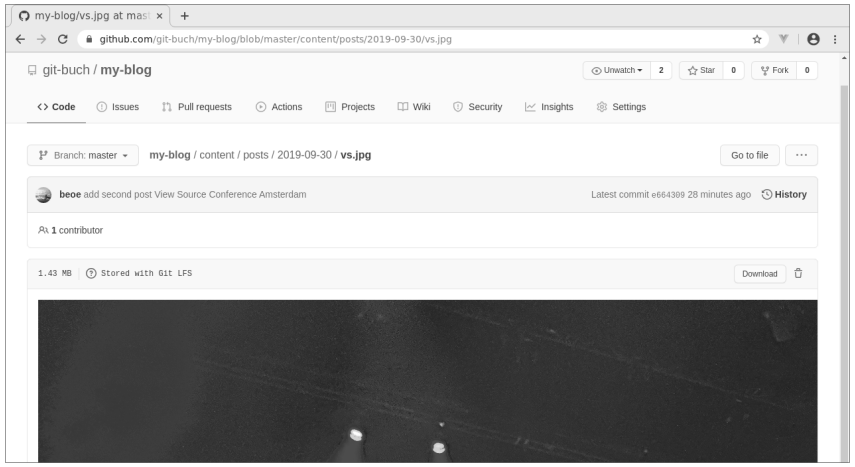

Abbildung 10.4 Ein mit LFS verwaltetes Foto in der GitHub-Oberfläche

Wir haben inzwischen einen zweiten Blogeintrag hinzugefügt, und uns fällt auf, dass wir mit der Qualität der Bilder gar nicht zufrieden sind. Daher bearbeiten wir beide Bilder, committen und pushen die Änderungen. In unserem lokalen `.git/lfs`-Ordner sind jetzt jeweils zwei Versionen der Bilder gespeichert, und sie belegen insgesamt 3,6 MByte an Platz (das Programm `du` errechnet die *Disk Usage* eines Ordners).

```
du -h .git/lfs

  480K  .git/lfs/objects/ca/99
  ...
  3,6M  .git/lfs
```

Spannend wird die Sache, wenn wir einen neuen Klon von unserem Remote Repository anlegen und darin die Größe des `.git/lfs`-Ordners untersuchen. Im letzten Schritt von `git clone` werden die oben angesprochenen Filter aktiv: Git holt nurmehr genau die Version der Bilder vom LFS-Speicherplatz, die für den aktuellen `HEAD` gebraucht werden. Die Ausgabe von `du -h` ergibt dann folglich nur mehr 1,1 MByte, was der Summe der beiden geänderten Bilder entspricht.

```
git clone https://github.com/git-buch/my-blog.git

  Cloning into 'my-blog'...
  ...
  Filtering content: 100% (2/2), 1.01 MiB | 576.00 KiB/s, done.

du -h my-blog/.git/lfs

  ...
  1,1M  my-blog/.git/lfs
```

Mit Ausnahme von Gitolite unterstützen alle in diesem Buch vorgestellten Git-Hosting-Provider Git LFS. Allerdings können Sie bei Azure Repos SSH nicht verwenden, wenn Sie LFS in Ihrem Repository aktiviert haben. Bei Gitea muss Git LFS in der Konfigurationsdatei explizit aktiviert werden.

Als Abschluss dieses Beispiels wollen wir unseren Blog natürlich noch veröffentlichen. Dazu werden wir Ihnen gleich zwei verschiedene Möglichkeiten zeigen. Die erste Variante kann über ein paar Mausklicks in Ihrem Webbrowser aktiviert werden und verwendet den Dienst von *Netlify*, die zweite Variante besteht aus einer GitHub Action und verwendet GitHub Pages.

Deploy mit Netlify

Netlify hat sich genau auf diesen Use Case spezialisiert. Der Dienst verbindet sich mit GitHub (oder auch GitLab oder Bitbucket) und konvertiert automatisch Ihren Quellcode mit einem Static Site Generator Ihrer Wahl und liefert die fertige Webseite auf dem eigenen *Content Delivery Network* (CDN) aus.

Zum Einstieg bietet Netlify einen kostenlosen Zugang an, auf dem immerhin bis zu 500 Projekte gehostet werden können.

Um unser Projekt auf Netlify online zu bringen, beginnen wir auf der Website von Netlify *https://www.netlify.com*. Unter Sign Up erlauben wir den Zugriff auf unseren GitHub-Account. Im folgenden Assistenten werden wir durch drei Schritte geführt, in denen neben dem GitHub-Repository auch das Build-Kommando angegeben werden muss. Da Netlify erkennt, dass es sich bei unserem Repository um eine Hugo-Seite handelt, ist das Feld schon korrekt ausgefüllt (siehe Abbildung 10.5).

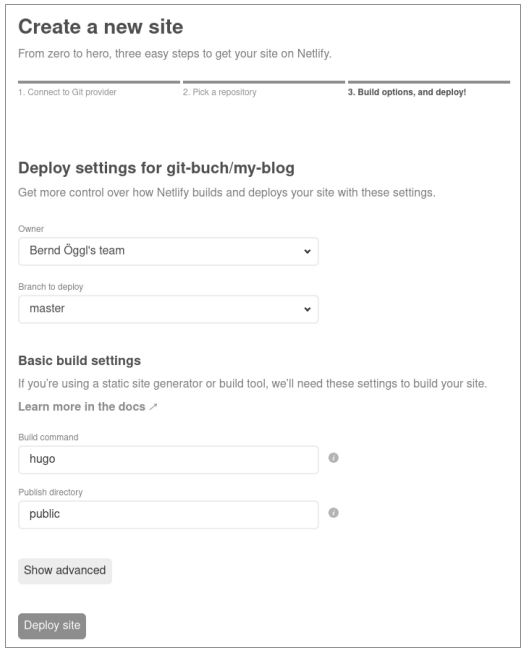

Abbildung 10.5 Der Import unseres GitHub-Projekts in Netlify

Netlify stellt uns automatisch einen Domainnamen zur Verfügung (in unserem Fall ist das `nervous-ardinghelli-90e87e.netlify.app`), und wir können optional einen eigenen DNS-Namen für die Seite angeben (wir verwenden `my-blog.gitbuch.info`). In der eigenen DNS-Verwaltung müssen wir dazu einen `CNAME`-Eintrag für den zufällig generierten Netlify-Hostname und den eigenen Domainnamen erstellen. Beim ersten Deployment erstellt Netlify automatisch SSL-Zertifikate für beide Namen, und unser Blog ist binnen weniger Minuten mit HTTPS online. Das war einfach! Sobald wir eine Änderung auf GitHub hochladen, startet Netlify einen neuen Build- und Deploy-Vorgang, und die Updates sind online.

Deploy mit GitHub Action und GitHub Pages

Wenn Sie den gerade vorgestellten Workflow mit Netlify nicht verwenden wollen, können Sie automatische Builds natürlich auch auf GitHub laufen lassen. Die GitHub

Action dazu müssen Sie gar nicht selbst schreiben, denn wenig überraschend hat das schon jemand gemacht.

Im Zusammenhang mit Git LFS und Submodulen wollen wir aber noch auf ein paar Details aufmerksam machen. Die schon aus Abschnitt 5.2 bekannte GitHub Action `checkout` wird um zwei Parameter erweitert, damit Submodule korrekt geklont und die Filter für LFS aktiviert werden:

```
on:
  push:
    branches: [ master ]
  pull_request:
    branches: [ master ]
jobs:
  build:
    runs-on: ubuntu-latest
    - uses: actions/checkout@v2
      with:
        lfs: true
        submodules: true
```

In drei Schritten verwandeln wir unseren Quellcode in HTML und laden das Ergebnis mit Push in einen Branch in unserem Repository hoch. Wir verwenden dazu zwei Actions vom GitHub-User `peaceiris`, die im GitHub Marketplace zum Download zur Verfügung stehen.

- Der erste Schritt, den wir `Hugo setup` benannt haben, installiert mit Hilfe der Action `peaceiris/actions-hugo` das Programm Hugo in unserer Umgebung. Der Zusatz `extended: true` lädt die erweiterte Hugo-Version, die auch Sass-Stylesheets umwandeln kann.
- Im `Build`-Schritt starten wir Hugo ohne weitere Parameter, wodurch die fertige Webseite im Ordner `public` gespeichert wird.
- Im dritten und letzten Schritt wird der Inhalt des `public`-Ordners mit der Action `peaceiris/actions-gh-pages` in den Branch `gh-pages` hochgeladen (Commit und Push). Das geheime `GITHUB_TOKEN`, das für die Push-Aktion benötigt wird, ist als Variable in allen GitHub Actions automatisch verfügbar.

```
  - name: Hugo setup
    uses: peaceiris/actions-hugo@v2.4.12
    with:
      extended: true
  - name: Build
    run: hugo
  - name: Deploy
    uses: peaceiris/actions-gh-pages@v3
    with:
      github_token: ${{ secrets.GITHUB_TOKEN }}
      publish_dir: ./public
```


Beachten Sie, dass der `gh-pages`-Branch nur die fertige Webseite enthält und nicht den Quellcode aus dem Master-Branch. Die Ordnerstruktur zwischen `master` und `gh-pages` ist völlig unterschiedlich, was für unsere bisherige Verwendung von Branches sehr ungewöhnlich ist. Um dieses Verhalten zu erreichen, sieht die GitHub-Weboberfläche die Option vor, den `gh-pages`-Branch öffentlich zugänglich zu machen (siehe Abbildung 10.6).

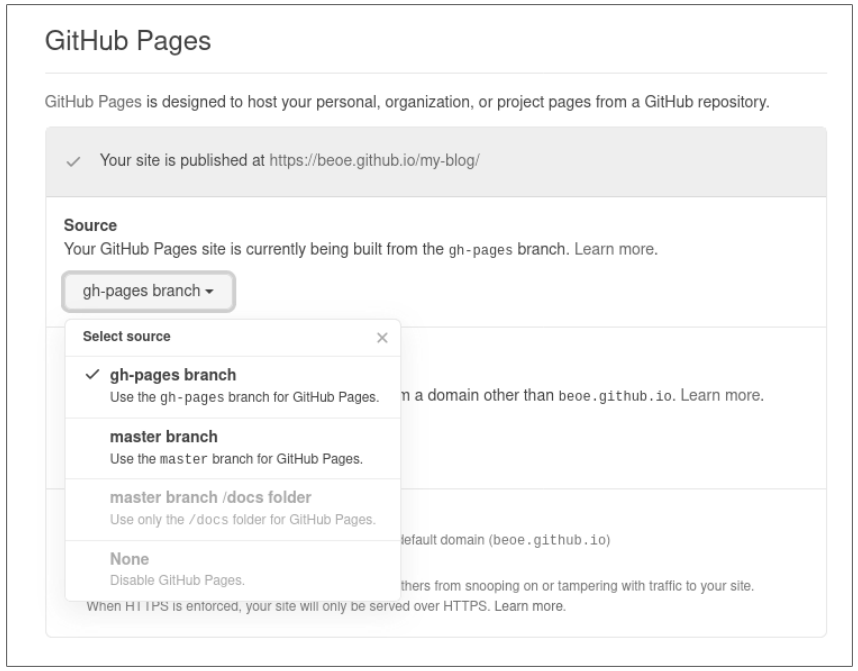

Abbildung 10.6 Die Einstellung für GitHub Pages am Branch gh-pages in GitHub

Sobald wir diese Einstellung aktivieren (Settings • Options • GitHub Pages) und unsere GitHub Action fehlerfrei läuft, ist der Blog bei GitHub online.

Wir haben das Layout der Startseite (eigentlich der Komponente, die Listen in Hugo erzeugt) noch etwas angepasst, damit die Bilder nicht so viel Platz einnehmen. Den Quellcode für das gesamte Beispiel finden Sie natürlich auf unserem GitHub-Account:

https://github.com/git-buch/my-blog

Auf einen Blick

1 Git in zehn Minuten 13
2 Learning by Doing 21
3 Git-Grundlagen 75
4 Datenanalyse im Git-Repository 153
5 GitHub 177
6 GitLab 209
7 Azure DevOps, Bitbucket, Gitea und Gitolite 235
8 Workflows 257
9 Arbeitstechniken 281
10 Git in der Praxis 315
11 Git-Probleme und ihre Lösung 347
12 Kommandoreferenz 369

Inhalt

Vorwort 9

1 Git in zehn Minuten 13

1.1 Was ist Git? 13
1.2 Software von GitHub herunterladen 16
1.3 Programmieren lernen mit Backup und Undo 18

2 Learning by Doing 21

2.1 git-Kommando installieren 21
2.2 GitHub-Account und -Repositorys einrichten 28
2.3 Mit dem Kommando »git« arbeiten 32
2.4 Authentifizierung 45
2.5 Git spielerisch lernen (Githug) 55
2.6 Entwicklungsumgebungen und Editoren 56
2.7 An einem fremden GitHub-Projekt mitarbeiten 70
2.8 Synchronisation und Backups 72

3 Git-Grundlagen 75

3.1 Nomenklatur 75
3.2 Die Git-Datenbank 80
3.3 Commits 84
3.4 Commit-Undo 92
3.5 Branches 100
3.6 Merge 105
3.7 Stashing 113
3.8 Remote Repositorys 115
3.9 Merge-Konflikte lösen 126
3.10 Rebasing 133
3.11 Tags 139
3.12 Referenzen auf Commits 144
3.13 Git-Interna 149

4 Datenanalyse im Git-Repository ... 153

4.1 Commits durchsuchen (git log) ... 153
4.2 Dateien durchsuchen ... 164
4.3 Fehler suchen (git bisect) ... 169
4.4 Statistik und Visualisierung ... 171

5 GitHub ... 177

5.1 Pull-Requests ... 178
5.2 Actions ... 183
5.3 Paketmanager (GitHub Packages) ... 191
5.4 Automatische Sicherheits-Scans ... 194
5.5 Weitere GitHub-Funktionen ... 198
5.6 GitHub CLI ... 204

6 GitLab ... 209

6.1 On Premises versus Cloud ... 210
6.2 Installation ... 211
6.3 Das erste Projekt ... 218
6.4 Pipelines ... 220
6.5 Merge-Requests ... 231
6.6 Web-IDE ... 233

7 Azure DevOps, Bitbucket, Gitea und Gitolite ... 235

7.1 Azure DevOps ... 235
7.2 Bitbucket ... 240
7.3 Gitea ... 242
7.4 Gitolite ... 252

8 Workflows ... 257

8.1 Anweisungen für das Team ... 257
8.2 Solo-Entwicklung ... 258
8.3 Feature-Branches für Teams ... 260
8.4 Merge/Pull-Requests ... 267
8.5 Long-Running Branches – Gitflow ... 271

8.6 Trunk-based Development ... 276
8.7 Welcher Workflow ist der Richtige? ... 279

9 Arbeitstechniken ... 281

9.1 Hooks ... 281
9.2 Prägnante Commit-Messages ... 287
9.3 Submodule und Subtrees ... 294
9.4 Mehr Komfort in Bash und Zsh ... 304
9.5 Zwei-Faktor-Authentifizierung ... 307

10 Git in der Praxis ... 315

10.1 Etckeeper ... 316
10.2 Dotfiles mit Git verwalten ... 319
10.3 Zugriff auf Subversion mit git-svn ... 326
10.4 Von SVN zu Git migrieren ... 330
10.5 Ein Blog mit Git und Hugo ... 335

11 Git-Probleme und ihre Lösung ... 347

11.1 Git-Fehlermeldungen (Ursache und Lösung) ... 347
11.2 Merge für eine einzelne Datei ... 354
11.3 Dateien permanent aus Git löschen ... 355
11.4 Ein Projekt aufteilen ... 363
11.5 Commits in einen anderen Branch verschieben ... 364

12 Kommandoreferenz ... 369

12.1 git-Kommando ... 369
12.2 Revisionssyntax ... 401
12.3 git-Konfiguration ... 402

Index ... 409