Die vielfältigen Fähigkeiten von Git

Allerlei Süßes

Andreas Krüger

Im normalen Entwickleralltag nutzt man die Standardmöglichkeiten von Git. Aber gelegentlich können selten genutzte Fähigkeiten das Leben angenehmer machen – wenn man sie denn kennt.

Wie ein roter Faden durchzieht „Vielfachheit" die Git-Fähigkeiten, die der folgende Artikel vorstellt. Damit ist gemeint, dass von einem Repository aus mehrere verwalten werden können, wo im normalen Alltag meistens „eins" genügt. Beispiele gefällig? Man nutzt im Normalfall nur einen Worktree (Verzeichnisbaum) auf der eigenen Festplatte, ein Remote-Repository „origin" und einen Versionsgraphen im lokalen Repository (s. Abb. 1).

Im Folgenden kommen zunächst Situationen zur Sprache, in denen mehrere Arbeitsverzeichnisse, mehrere Remotes und mehrere Versionsbäume nützlich sind.

Mehrere Arbeitsverzeichnisse

Eine solche Situation ist die Unterbrechung. Irgendetwas schiebt sich in der Dringlichkeit nach vorne. Was man gerade getan hat, muss unterbrochen werden und warten. Ein klassisches Beispiel aus der DevOps-Praxis: Man entwickelt an einem Feature-Branch. Plötzlich hängt etwas in der CI/CD-Pipeline und ist im master zu fixen, und zwar schnell.

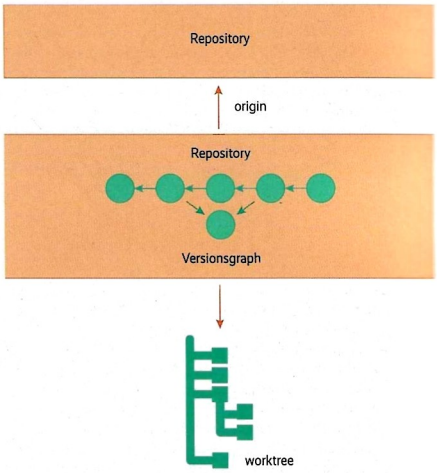
Es gibt mehrere Möglichkeiten, mit einer solchen Situation umzugehen. Beispielsweise kann man `git stash` benutzen, um den Feature-Branch-Zustand zu sichern. Dann wechselt man vom Feature-Branch zum master und nimmt die Arbeit am CI/CD-Problem auf. Eine andere, probate Möglichkeit bietet sich dadurch, dass ein Git-Repository auf der lokalen Festplatte mehr als ein Arbeitsverzeichnis („Worktree") verwalten kann (s. Abb. 2).

Man erzeugt so ein zweites Arbeitsverzeichnis aus der Wurzel des ersten heraus, zum Beispiel mit

```
git worktree add ../worktree2 master
```

Nun kann man mit einem schlichten `cd ../worktree2` in das erzeugte Arbeitsverzeichnis wechseln, in dem master bereits ausgecheckt ist, und mit `cd` wieder zurück ins ursprüngliche mit dem Feature-Branch. Überhaupt ermöglichen multiple Arbeitsverzeichnisse bequeme und schnelle Kontextwechsel. Zwei IDE-Instanzen können parallel laufen, man braucht bei Bedarf nur zwischen den Fenstern zu wechseln.

In der geschilderten Beispielsituation, dass an der CI/CD-Pipeline etwas repariert werden muss, kann das sehr angenehm sein: Während der CI/CD-Job erst losläuft, hat man schon wieder die Arbeit am Feature-Branch aufgenommen. Intern hat „worktree2" in seinem Verwaltungsverzeichnis `.git` nur eine Art Verweis auf das eigentliche Repository. Dadurch stehen alle Commits, Branches und Remotes in beiden gleichartig zur Verfügung. Man kann zum Beispiel in einem Arbeitsverzeichnis einen Commit zu einem Branch hinzufügen und



Der Normalfall mit einem Worktree (Abb. 1)

ihn in einem zweiten Arbeitsverzeichnis in einen anderen Branch hineinmergen.

Den Index gibt es naturgemäß für jeden Worktree einzeln. So bleiben die verschiedenen `git add`-Befehle unabhängig voneinander. Übrigens wehrt sich Git dagegen, den selben Branch in zwei verschiedenen Arbeitsverzeichnissen auszuchecken. Wer parallele Arbeitsverzeichnisse für sich entdeckt hat, mag vielleicht gleich beim Anlegen eines neuen Feature-Branches für diesen ein eigenes Arbeitsverzeichnis vorsehen. Dafür bietet `git add worktree` eine ähnliche Funktionsweise `-b new_branch` wie `git checkout`. Man nutzt sie zum Beispiel wie folgt:

```
git fetch origin
git worktree add -b f_branch ../f_branch_worktree origin/master
```

Ein solches Arbeitsverzeichnis lässt sich wieder abräumen, zum Beispiel mit folgender Zeile:

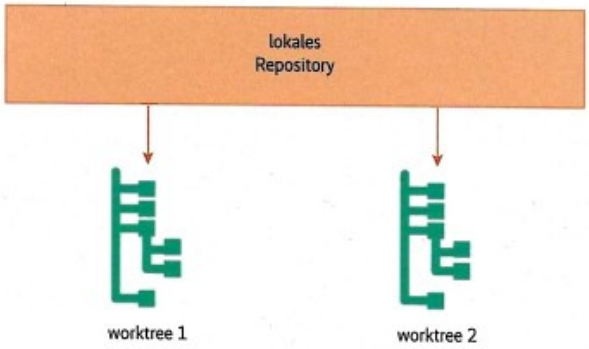
```
git worktree remove ../f_branch_worktree
```

Die Arbeit mit mehreren Worktrees hat sich in der alltäglichen Praxis des Autors bewährt. Allerdings warnt die Dokumentation derzeit, dass dieses Feature von Git noch experimentell ist, und rät insbesondere davon ab, von Repositories mit Submodulen mehrere Arbeitsverzeichnisse zu generieren.

Mehrere Remotes

Das Git-Repository auf der eigenen Festplatte kann mit mehreren Remotes kommunizieren, also fernen Git-Repositories, die (normalerweise) auf irgendwelchen Git-Servern liegen.

Mehrere Remotes können zum Beispiel nützlich sein, wenn man temporär mit einer Person zusammenarbeitet, die auf den offiziellen Git-Server keinen Zugriff hat. Dem Autor ist das mehrfach passiert, zuletzt mit einer Praktikantin. Einerseits wünschte das betreffende Projekt ihre Mitarbeit. Andererseits wollte das Unternehmen für die (kurzzeitige) Mitarbeit nicht den Aufwand treiben, der Praktikantin einen vollen Account im Firmen-LDAP einzurichten. Vielmehr baten die Firmenadmins des damaligen Arbeitgebers den Autoren, eine pragmatische Einzelfalllösung abseits der offiziellen Infrastruktur zu finden. Was nun? Wie bekommt man die Zusammenarbeit unter diesen Rahmenbedingungen hin? Das geht, sogar recht problemlos, indem man ein zweites, temporäres Remote-Repo anlegt, auf das beide Personen zugreifen können (s. Abb. 3).



Beispielhaft hängen zwei Arbeitsverzeichnisse am lokalen Repository (Abb. 2)

lässt sich ebenfalls prima einsetzen, Vertrauen zwischen A und B vorausgesetzt. Ein Docker-Container ist ebenfalls nutzbar.

Für so einen temporären Repository-Server braucht man sich normalerweise keine Gedanken über Backups zu machen (was die Sache weiter vereinfacht), denn alle Informationen sind auch in den lokalen Repositories vorhanden, und was wichtig ist, wandert über kurz oder lang ohnehin ins offizielle Repository. Nachdem das Ziel der Kooperation erreicht ist, kann das temporäre Repository einfach ohne Weiteres gelöscht werden.

Ist ein Server gefunden, lässt sich das temporäre Repository mit wenigen Handgriffen einrichten. Im folgenden Beispiel haben beide Personen auf dem Host *server.example.org* Zugriff auf den gemeinsam genutzten User „user". Einer von beiden bringt die Sache in Gang:

```
ssh user@server.example.org mkdir -p collab-repo
ssh user@server.example.org cd collab-repo '&&' git init -bare
```

Anschließend richtet A für sein lokales Repository ein neue Verbindung zu einem „Remote" mit dem Namen „collab" ein:

```
git remote add collab user@server.example.org
```

und füllt das neue temporäre Repo:

```
git push collab origin/master:master
```

B holt sich das Material mit dem Befehl:

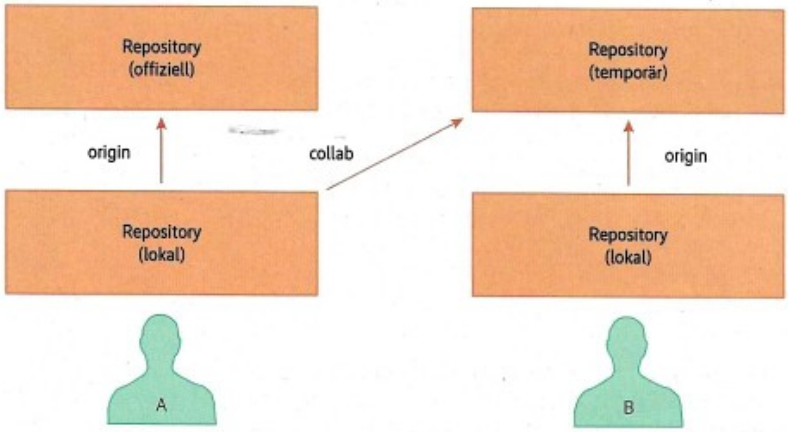
```
git clone user@server.example.org
```

Nutzen A und B für den ssh-Zugriff auf *server.example.org* unterschiedliche Nutzer *auser* und *buser*, wird es geringfügig komplizierter. Gewöhnlich wird man eine gemeinsame Gruppe *ggroup* auf *server.example.org* finden oder erstellen, der *auser* und *buser* beide angehören.

Schnell mal eben ein temporäres Remote

Es gibt verschiedene Möglichkeiten, wie man ein temporäres Repository aufsetzen kann, auf das zwei Personen A und B zugreifen können.

Besonders bequem hat man es, wenn es irgendwo einen Rechner gibt, auf dem Git installiert ist und auf den sowohl A als auch B per ssh zugreifen können. Das könnte ein ohnehin vorhandener Server sein. Auch eine kurzfristig in einer Cloud oder bei einem Provider angemietete virtuelle oder reale Maschine ist brauchbar. Selbst ein Winzling (ein Raspberry Pi oder Ähnliches) reicht dafür aus. Einer der beiden Arbeitsplatzrechner (der von A oder der von B)



Temporäres zweites Git-Remote (Abb. 3)

Dann kann A das Repository wie in Listing 1 initialisieren: Anschließend nutzt A `auser@server.example.org:collab-repo` und B `buser@server.example.org:/home/auser/collab-repo` für den Zugriff.

Eventuell ist es noch nötig, `/home/auser` mit `chmod a+rx /home/auser` für lesende Zugriffe zu öffnen. Das ermöglicht unter Umständen lesenden Zugriff durch andere User von *server.example.org* auch auf andere Dateien von *auser*. Will man das vermeiden, nutzt man alternativ ein neutrales Verzeichnis, zum Beispiel `/var/lib/collab-repo`.

Arbeiten mit einem temporären Kollaborations-Repository

Wenn man ein temporäres Repository erst einmal hat, können die beiden beteiligten Personen bequem damit arbeiten. B möchte zum Beispiel in einem Feature-Branch arbeiten:

```
git fetch
git checkout -b b_contrib origin/master
```

A kann neues Material aus dem offiziellen Repository jederzeit zur Verfügung stellen:

```
git fetch origin
git push collab origin/master:master
```

B nimmt das Material entgegen, wie bei Feature-Branches üblich:

```
git fetch origin
git rebase origin/master
```

B kann Material zur Verfügung stellen:

```
git push origin -u b_contrib
```

und A schaut es sich an:

```
git fetch collab
git checkout -b b_contrib collab/b_contrib
```

Alternativ richtet sich A dafür wie oben einen eigenen Worktree ein. Soll das Material im offiziellen Repository als Feature-Branch auftauchen, kann A es dort zugänglich machen:

```
git push origin collab/b_contrib:b_contrib
```

Ross und beide Reiter nennen

Bei dieser Arbeitsweise bleiben die Commits von B erhalten. Sie finden sich später komplett mit Checkin-Kommentar, Zeitstempel und Autorenangabe „B" im offiziellen Repo. Dass sie durch Vermittlung von A dort gelandet sind, geht aus dem Repository-Inhalt nicht mehr hervor. Das kann im Einzelfall erwünscht oder unerwünscht sein. Möglicherweise möchte man die Mitwirkung von A langfristig nachvollziehen können. Dafür bietet sich eine andere selten genutzte Vielfalt an, die Git bietet.

Jedes Versionsmanagementsystem kann selbstverständlich für jede Änderung die Frage beantworten: Wer war das? Git bietet das natürlich auch, aber mit zusätzlichem Mehrwert: Für jeden Commit werden nicht nur ein, sondern zwei Verantwortliche ins Repository eingetragen (s. Abb. 4). Als Author wird in der Git-Terminologie die Person bezeichnet, die die Änderung inhaltlich entwickelte, als Committer, wer sie ins Repository eintrug. Die beiden sind häufig identisch, müssen es aber nicht sein.

Listing 1: Initialisierung des Repository

```
ssh auser@server.example.org mkdir -p collab-repo
ssh auser@server.example.org chgrp ggroup collab-repo
ssh auser@server.example.org chmod g+sw collab-repo
ssh auser@server.example.org cd collab-repo '&&' git init --bare --share=group
```

Einen neuen Commit kennzeichnet die eintragende Person als den inhaltlichen Beitrag von zum Beispiel „A. U. Thor" mit einem Befehl wie

```
git commit --author='A U Thor <a.u.thor@example.org>'
```

Die entstehenden kompletten Angaben kann man sich mit `git log` anschauen, für den letzten Commit zum Beispiel mit

```
git log --pretty=fuller HEAD^..HEAD
```

Ein beispielhafter Output sieht wie in Listing 2 aus.

Es ist auch möglich, sich nachträglich als Committer einzutragen. Wer das möchte, erstellt einfach eine Kopie des Materials. In der oben beschriebenen Situation könnte A dazu folgende Befehle ausführen:

```
git checkout b_contrib
git rebase --no-ff origin/master
```

Ein Repository mit nur einem Commit

Die bisher vorgeschlagene Methode der Zusammenarbeit von A und B basiert auf vollwertigen Repositories, die jeweils die gesamte Vergangenheit des Projekts mit an Bord haben. Die historische Vollständigkeit ist manchmal nützlich, aber gelegentlich auch überflüssiger Ballast. Das Hauptthema dieses Artikels, „Vielzahl statt Einzahl", wird deshalb hier kurzfristig umgekehrt: Es soll im Folgenden von Repositories die Rede sein, die statt der sonst üblichen Gesamthistorie nur einen Commit enthalten, oder nur einige wenige.

Eine (teilweise) Kopie eines Repositories mit unvollständiger Historie nennt die Git-Dokumentation „shallow". Man erzeugt sie mit einer entsprechenden `--depth`-Option von `git clone` (s. Abb 5).

Zum Beispiel soll wieder eine Zusammenarbeit zwischen zwei Personen A und B auf dem Feature-Branch `f_branch` erfolgen. Den hat A bereits angelegt. Dann kann A ein Repository nur mit dem letzten Commit von `f_branch` erzeugen und an B weitergeben. Das geht notfalls sogar ganz ohne Git-Server.

Angenommen, das lokale Repo von A liegt in *$HOME/lokal-repo*. Dann erzeugt A die unvollständige Kopie (in einem Verzeichnis, das vorzugsweise gerade *nicht* in einem Git-Worktree liegt) mit:

```
git clone --bare --single-branch -b f_branch --depth 1 file://$HOME/lokal-repo
```

Die erzeugte teilweise Repository-Kopie enthält zunächst noch einen Verweis auf das eigene lokale *$HOME/lokal-repo*. Den entfernt A (nach Wechsel in das entsprechende Verzeichnis) mit:

```
git remote remove origin
```

Das so erzeugte Repository kann A nun in ein ZIP-Archiv oder Ähnliches einpacken und auf passendem Weg B zur Verfügung stellen. Das empfiehlt sich vor allem, wenn ein Git-Server noch nicht gefunden ist, die Netzwerkverbindung zwischen A und B lahmt und B möglichst schnell loslegen soll.

B packt das Archiv wieder aus und nutzt das entstehende Repository als Basis für eine `git clone`-Operation. Mit dem daraus entstehenden Arbeits-Repository arbeitet B lokal wie gewohnt weiter. Später soll B Ergebnisse zur Verfügung stellen. Man kann dann problemlos nachträglich auf einen gemeinsam erreichbaren Klon aufrüsten, auf den B Schreibzugriff hat, zum Beispiel ssh-basiert wie oben.

Haben sich nur Textdateien geändert, kann B auch eine Patch-Datei an A schicken, also den Output von `git diff`. Das geht bequem via E-Mail. Mit `git apply` und anschließendem `git commit --author ...` kann A die Ergebnisse von B lokal integrieren.

Teilweise Kopieren: Auch sonst nützlich!

Die Idee, sich beim Kopieren eines Repositories auf den letzten Commit zu beschränken, ist auch sonst gelegentlich nützlich. Viele Builds, CI-Pipelines und ähnliche automatisierte Prozesse können auf einfache Weise erheblich beschleunigt werden, wenn einem ohnehin vorhandenen `git clone` die Option `--depth 1` mitgegeben wird.

Derselbe Trick kann ebenfalls hilfreich sein, wenn man auf den Inhalt eines Open-Source-Repositories neugierig ist. Man will gerne so schnell wie möglich einen Worktree auf der eigenen Festplatte begutachten können. Die Historie des Repositories interessiert erst später, in zweiter Linie.

In diesem Fall besorgt man sich das Repository zunächst mit `git clone --depth 1`. Das geht schnell, entsprechend bald kann man anfangen, sich im Worktree umzusehen. Währenddessen lässt man im Hintergrund den langen Download der Gesamthistorie laufen:

```
git fetch --unshallow
```

Wenn man das auf einem Repository mit vollständiger Historie aufruft, es also nicht „shallow" ist, erscheint natürlich eine Fehlermeldung. Bei einigen Git-Versionen ist die Fehlermeldung verwirrend falsch ins Deutsche übersetzt:

Die Option --unshallow kann nicht in einem Repository mit unvollständiger Historie verwendet werden.

Das sollte eigentlich „... mit vollständiger Historie ..." heißen.

Wenn man von einem Repository beim initialen Klonen nur den letzten Commit haben wollte, holt Git auch bei späteren `git fetch`-Operationen zunächst nur den betreffenden Branch. Bei von Anfang an komplett geklonten Repositories holt `git fetch` dagegen normalerweise alle remote vorhandenen Branches. Man kann nachträglich mit folgender Zeile konfigurieren, dass alle Branches geholt werden sollen:

```
git config remote.origin.fetch '+refs/heads/*:refs/remotes/origin/*'
```

Open Source Contributions mit zwei Remotes

Es arbeitet sich auch gut mit zwei Remotes, wenn man zu einem Open-Source-Projekt beitragen will. Eine Herangehensweise ist es, das offizielle Repo auf die eigene Festplatte zu klonen und dort einen Feature-Branch zu erstellen. Dann kann man experimentieren. Erst, wenn ein nützliches Ergebnis dabei herauskommt, wird es zunächst (immer noch lokal) in präsentable Form gebracht. Es genügt völlig, erst danach den sichtbaren Fork auf der Plattform des Git-Providers zu erstellen. Der wird zum zweiten Remote des lokalen Repos auf der eigenen Festplatte; bewährt haben sich die Remote-Namen „origin" und „mine". Auf `git push` der eigenen Arbeit folgt der übliche Pull Request an das Projekt.

Open Source und mehrere Identitäten

In Zusammenhang mit Open Source tritt manchmal auch ein Problem auf, das ebenfalls mit Vielfachheit zu tun hat, und zwar mit verschiedenen Identitäten derselben Person. Es betrifft Aktive, die für Lohn und Brot in kommerziellen Projekten arbeiten und sich zusätzlich in ihrer Freizeit für Open-Source-Projekte engagieren. Auf den ersten Blick sieht es so aus, als müssten sie sich entscheiden: Welche E-Mail-Anschrift sollen sie für die eigenen Git-Aktivitäten benutzen: die berufliche oder die private?

Zwei Verantwortliche in jedem Commit (Abb. 4)

Viele entscheiden sich für die private. Es kann ganz amüsant sein, sich für ein internes kommerzielles Repository von example.com eine Gesamtliste der im Projekt benutzten E-Mail-Anschriften zu besorgen, die nicht zu example.com gehören:

```
git log --pretty='%ae' | sort -u | grep -v example.com
```

Obwohl solche Kommandozeilen oft viel Output erzeugen, besteht das Problem eigentlich nur scheinbar. Denn Git erlaubt durchaus mehrere Identitäten derselben Person: Man kann die eigene Identität für jedes Repository einzeln konfigurieren.

Konkret wechselt man in ein Repository und konfiguriert dort spezifisch die in die Commits dieses Repositories einzutragende eigene E-Mail-Anschrift, sowie, wenn man für den Namen eine andere Variante benutzen möchte als sonst, auch diesen. Dafür genügt es, `git config` wie bei der Ersteinrichtung von git aufzurufen, aber jetzt die Option `--global` wegzulassen. Dadurch wird die Konfiguration Repository-spezifisch:

```
git config user.email 'a.u.thor@example.org'
git config user.name 'A. U. Thor'
```

Mehrere Bäume

Eine weitere Mehrfachmöglichkeit von Git wird nicht oft gebraucht, aber wenn, ist sie da: Git kann im selben Repository komplett getrennte Branch-Bäume verwalten.

In einem üblichen Repository gibt es nur einen Commit ohne Vorgänger: den ersten. Aber es kann durchaus mehrere solcher Commits geben (s. Abb 6). Graphentheoretisch ausgedrückt: Der Commit-Graph darf unzusammenhängend sein.

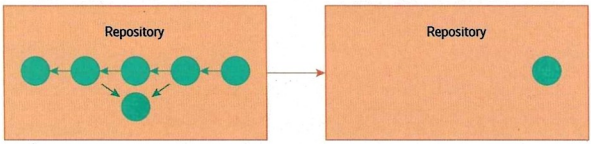
Man kann in einem existierenden Repository einen neuen leeren Branch ohne jede Commits und Vorfahren erzeugen, zum Beispiel mit:

```
git checkout --orphan isolated-branch
```

Die Situation ist ähnlich der nach `git init`: Ein Commit, den man nun erzeugt, ist ein „erster" Commit, also einer ohne Vorgänger. Man fängt sozusagen ganz von vorne an. Dies kann unmittelbar vor der Erstveröffentlichung bisher interner Arbeit nützlich sein. Etwas hat klein angefangen, als privates Feierabendprojekt einer

Listing 2: Output von git log

```
commit 84601e93dff652b0c8c2cbc1ec9e476366b06888 (HEAD -> git-vielfalt-artikel)
Author: A U Thor <a.u.thor@example.org>
AuthorDate: Mon Mar 4 14:23:18 2019 +0100
Commit: Andreas Krüger <andreas.krueger@innoq.com>
CommitDate: Mon Mar 4 14:23:18 2019 +0100
Sample commit with different author.
```



Shallow Clone (Abb. 5)

einzelnen Person, hat einige Irrungen und Wirrungen durchlebt, sich dann ganz manierlich entwickelt und soll nun als Open-Source-Repository debütieren.

In der Situation ist es eine offene Frage, ob man die privaten Irrungen und Wirrungen mit veröffentlichen will. Schon die genauen Zeitstempel, wann man seinerzeit aktiv war, sind verhältnismäßig intime personenbezogene Daten, die man der Welt vielleicht nicht zur Verfügung stellen will. An die Stelle der detaillierten Gesamthistorie kann eine Art „Paukenschlag-Commit" treten, der das gesamte bisher Erreichte sozusagen „aus dem Nichts" entstehen lässt. Ein beispielhaftes Vorgehen ist:

```
git branch -m master old-master
git checkout --orphan master old-master
git commit -m 'Results thus far, without convoluted history.'
```

Der Code benennt zunächst den bisherigen master in old-master um. Es gibt also kurzfristig keinen Branch master mehr im Repository. In der nächsten Zeile wird master neu hergestellt als leerer Branch, ohne Commit, Vorgänger oder Historie. Dabei werden Worktree und Index vorbelegt mit dem Inhalt aus old-master. Der nun folgende Befehl erzeugt den gewünschten „Paukenschlag-Commit".

Man kann sich anschließend mit `gitk` oder `git log --pretty=raw` davon überzeugen, dass der neue Commit tatsächlich keine Vorgänger hat.

Copy and Paste mit Git

Eine Situation mit mehreren getrennten Zusammenhangskomponenten des Commit-Graphen lässt sich auch anders herstellen: Man fügt dazu dem lokalen Repository ein zweites Remote hinzu, das inhaltlich mit dem bisherigen nichts zu tun hat.

```
git remote add other-remote user@server.example.org
git fetch other-remote
```

Das kann gelegentlich nützlich sein, um einzelne Dateien oder Verzeichnisbäume von einem anderen Repository zu übernehmen, ohne dass Commits des anderen Repositories deshalb in die eigene Versionsgeschichte integriert werden. Dazu benutzt man eine der Varianten von `git reset`. Der Inhalt des Verzeichnisses src/ other wird Bestandteil des eigenen Verzeichnisbaums durch:

```
git reset other-remote/master -- src/other
git commit -m 'Copy src/other from other-remote.'
git checkout -- .
```

Ungewöhnlich dabei ist, dass `git reset` den Verzeichnisbaum direkt in den Index kopiert, aber nicht in den Worktree. Wer vor (oder unmittelbar nach) dem `git commit` ein `git status` aufruft, dem wird das so angezeigt, als sei im Worktree alles unterhalb vonsrc/other gelöscht. Das abschließende `git checkout` bringt Worktree wieder in Übereinstimmung mit dem letzten Commit und Index.

Diese Variante von `git reset` erlaubt es ganz allgemein, bequem Material (Dateien und Verzeichnisse) von einem beliebigen Commit direkt in den Index zu kopieren, ohne Umweg über Kopien im Arbeitsverzeichnis. Der Ursprungs-Commit des Materials erscheint dabei nicht als Vorgänger. So kann man kurzfristig ein zweites Repository als ein neues Remote lokal einbinden, Material mit `git reset` kopieren und das Remote wieder entfernen. Dadurch entsteht keine dauerhafte Beziehung zum zweiten Repository.

Mehrere Repositories synchronisieren: „Die Pumpe"

Wie schon dargelegt kann eine kurzfristige Kopie eines Repositories auf einem zweiten Server für temporäre Kooperation nützlich sein. Aber auch für einen langfristigen Zweitwohnsitz eines Repositories gibt es oft gute Gründe. Im einfachsten Fall ist dabei die Version auf einem der Server die führende Instanz. In solchen Fällen nutzt man gerne eine „Pumpe". Damit ist eine Automatik gemeint, die regelmäßig alle Commits und Branches vom führenden Repository in das andere kopiert.

Wann kann man so etwas gebrauchen? Zum Beispiel hat man ein intern benutztes Repository veröffentlicht, will aber nicht alle Teammitglieder und CI/CD/Build-Pipelines zwingen, sich Credentials für das öffentliche Angebot zu besorgen. Stattdessen bleibt das alte, intern benutzte Repository wie bisher in Gebrauch. Eine Pumpe hält die öffentlich sichtbare Instanz automatisch aktuell.

Ein anderer Fall aus der Praxis: Das Entwicklungsteam ist gewohnt, auf den Server eines Git-as-a-Service-Anbieters zuzugreifen. Aber ein alternatives Angebot bietet eine angenehme CI-Lösung (für dort gehostete Repositories), die das Team nutzen möchte. Hier hilft ebenfalls eine Pumpe.

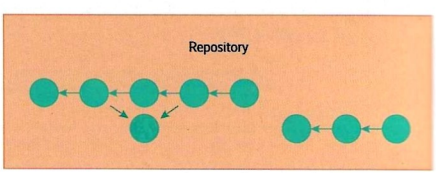
Eine solche Pumpe soll wartungsfrei laufen, gehört also automatisiert und in einem CI/CD-Job untergebracht. Die nötigen Funktionen sind in jeder passablen Skriptsprache schnell geschrieben. Es folgt eine kurze Übersicht, was dafür zu tun ist. Die hier beschriebene Beispielpumpe nutzt ein lokales Repository. Sie soll von `remote_from` nach `remote_to` kopieren. Den folgenden Befehl ruft die Pumpe zunächst auf, um das lokale Repository beiderseits auf den neuesten Stand zu bringen:

```
git fetch -p remote_from
git fetch -p remote_to
```

Dann wird der Stand extrahiert:

```
git for-each-ref --format '%(refname) %(objectname)' refs/remotes/remote_from
git for-each-ref --format '%(refname) %(objectname)' refs/remotes/remote_to
```

Der jeweilige Output (einzeln) aufgefangen. Das ergibt je eine Liste von Branch-Namen (HEAD ignoriert man) mit SHAs. Bei aktuellen Versionen von git kann man sich das Leben übrigens noch einfacher machen, indem man `%(refname:lstrip=3)` benutzt. Beide



Repo mit Branches ohne gemeinsame Vorfahren (Abb. 6)

Listen kann man nun mit ein paar Zeilen Code der genutzten Skriptsprache vergleichen. Auf Unterschiede reagiert die Pumpe wie folgt.

Einen Branch `branch_n` von `remote_from`, den es auf `remote_to` noch nicht gibt oder der dort auf einen anderen SHA hinausläuft als auf remote_from, kopiert man mit

```
git push -q --force remote_to refs/remote/remote_from/branch_n: ↵
                                                refs/heads/branch_n
```

Einen `old_branch`, den es auf `remote_from` nicht mehr gibt, aber noch auf `remote_to`, löscht man mit

```
git push -q remote_to :old_branch
```

Regelmäßig aufgerufen, hält die so programmierte Pumpe alle Branches von `remote_to` stets auf demselben Stand wie die von `remote_from`. Wer außerdem Tags nutzt und übertragen will, kann die Pumpe passend erweitern. Im einfachsten Fall genügt es, `git fetch --tags remote_from` mit `git push --tags remote_to` zu kombinieren. In komplizierteren Fällen hilft eine passende projektspezifische Erweiterung der grundlegenden Pumpenfunktion.

Mehrere Credentials im Zugriff

Innerhalb der CI/CD-Umgebung braucht die Pumpe zwei Credentials, um auf beide Repositories zuzugreifen. Ein gängiger Weg ist, Credentials in CI/CD-Umgebungen mit Umgebungsvariablen zur Verfügung zu stellen. Wie kann man das praktisch organisieren? Das kommt ganz auf den Zugriff an und es gibt verschiedene Möglichkeiten. Die üblichen Kandidaten sind SSH und HTTPS.

SSH Credentials

Auf ein entferntes (remote) Repository *user@host.example.org* (*oft git@host.example.org*) wird mit SSH zugegriffen. Damit der Zugriff funktioniert, sind normalerweise zwei Bedingungen zu erfüllen: Der SSH-Host-Key des Servers ist lokal bekannt und der SSH-Key des lokalen Users ist auf dem Server eingetragen und mit den jeweiligen Bedingungen versehen.

Um die erste Bedingung zu erfüllen, fängt man den Host-Key einmalig manuell mit folgender Zeile auf:

```
ssh -o UserKnownHostsFile=known_hosts user@host.example.org
```

Hat der eigene Rechner mit diesem Host schon kommuniziert, überprüft man als nächstes, ob der aufgefangene Host-Key in der neuen Datei known_hosts im derzeitigen Verzeichnis mit dem in der zentralen $HOME/.ssh/known_hosts übereinstimmt. Dazu vergleicht man den Output:

```
ssh-keygen -F host.example.org -l -f $HOME/.ssh/known_hosts
ssh-keygen -F host.example.org -l -f known_hosts
```

Damit erhält man den öffentlichen Host-Key in der Datei known_hosts im derzeitigen Verzeichnis. An der Datei ist nichts Geheimes. Um sie zu nutzen, kopiert man sie ins Verzeichnis $HOME/.ssh des betreffenden Benutzers, der in der CI/CD-Umgebung die Pumpe ausführt.

Will man auf mehrere Server via SSH zugreifen, ist ein Kopieren der Angaben hintereinander in eine gemeinsame known_hosts notwendig. Das eigentliche Credential für die Pumpe ist der private SSH-Schlüssel. Den erzeugt man zum Beispiel mit

```
ssh-keygen -N '' -f id_rsa
```

Der Befehl erzeugt zwei Dateien, id_rsa.pub und id_rsa. Die erste ist der öffentliche Schlüssel, den man einmalig in die UI des Git-Servers kopiert. Diese Datei braucht weder Schutz noch Geheimhaltung. Die Pumpe selbst benötigt diese Datei übrigens nicht.

Der Inhalt von id_rsa ist dagegen zu schützen. Wer den Inhalt kennt, der kann auf die entsprechenden Repos auf dem Git-Server zugreifen (also genau das, was die Pumpe tun soll). Den Inhalt konfiguriert man in die UI des CI/CD-Umgebung als „Geheimnis", das während eines Laufes der Pumpe als Umgebungsvariable zur Verfügung steht, zum Beispiel GIT_PRIVATE_KEY.

Damit die Pumpe den Key nutzen kann, läuft am Anfang des Jobs:

```
mkdir -p "$HOME/.ssh"
touch "$HOME/.ssh/id_rsa"
chmod go-rwx "$HOME/.ssh" "$HOME/.ssh/id_rsa"
echo "$GIT_PRIVATE_KEY" > "$HOME/.ssh/id_rsa"
```

Die Datei known_hosts gehört ins selbe Verzeichnis $HOME/.ssh. Leider entsteht durch dieses Verfahren auf dem Buildserver die Datei $HOME/.ssh/id_rsa, die vor Missbrauch zu schützen ist.

Listing 3: Ein Skript zum Erzeugen des Outputs

```
#!/bin/bash

if test "$1" = get
then
 echo username="${GIT_USER}"
 echo password="${GIT_PASSWD}"
fi
```

HTTPS Credentials

Beim Zugriff auf ein Git-Repository via HTTPS kommt man normalerweise mit klassischen User- und Passwort-Angaben weiter. Die übernimmt git von einem Skript (Listing 3), das auf den Aufruf mit dem Parameter `get` wartet und dann zwei Zeilen Output erzeugt: eine mit der User-Angabe, eine für das Passwort.

Die nötigen Angaben konfiguriert man in der jeweiligen CI/CD-Umgebung als Geheimnisse, die in den Umgebungsvariablen GIT_USER und GIT_PASSWD zur Verfügung stehen. Das Skript stellt man für die Pumpe zur Verfügung. Es wird für Zugriffe auf einen Git-Server https://git.example.org durch folgenden Befehl aktiviert:

```
git config --global credential.https://git.example.org.helper /usr/local/ ↵
                                                        bin/gitcreds
```

Wenn man Credentials für mehrere Server braucht, arbeitet man möglicherweise mit mehreren Instanzen wie gitcreds1 und gitcreds2. Alternativ genügt ein einzelnes Skript. Indem es STDIN auswertet, kann es erfahren, welche Credentials gerade notwendig sind.

Fazit

Git bietet vielfache Möglichkeiten abseits des Üblichen. Wenn man sie kennt, kann man sich immer mal wieder das Leben erleichtern. Obendrein hilft die Kenntnis der Möglichkeiten, besser zu durchschauen, was man normalerweise eigentlich tut. Das verschafft das angenehme Gefühl, bei der Nutzung von Git mehr „Wasser unter dem Kiel" zu haben. (bbo@ix.de)

Dr. Andreas Krüger

arbeitet als Seniorspezialist bei DB Systel GmbH. Er übernimmt Verantwortung für Infrastruktur wie Cloud-Umgebungen oder Build- und Deployment Pipelines, plant, verhandelt, dokumentiert und implementiert Schnittstellen, Algorithmen und Anwendungen sowie entwirft tragfähige Softwarearchitektur.