Kostenloses eBook

LERNEN Git

Free unaffiliated eBook created from
Stack Overflow contributors.

#git

Inhaltsverzeichnis

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: git

It is an unofficial and free Git ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Git.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit Git

Bemerkungen

Git ist ein kostenloses, verteiltes Versionskontrollsystem, mit dem Programmierer Codeänderungen über "Momentaufnahmen" (Commits) im aktuellen Status verfolgen können. Durch die Verwendung von Commits können Programmierer gemeinsam Funktionen testen, debuggen und neue Funktionen erstellen. Alle Commits werden in einem so genannten "Git-Repository" aufbewahrt, das auf Ihrem Computer, privaten Servern oder Open-Source-Websites wie Github gehostet werden kann.

Mit Git können Benutzer außerdem neue "Verzweigungen" des Codes erstellen, wodurch verschiedene Versionen des Codes nebeneinander leben können. Dies ermöglicht Szenarien, in denen ein Zweig die aktuellste stabile Version enthält, ein anderer Zweig eine Reihe neuer Features enthält, die entwickelt werden, und ein weiterer Zweig enthält einen anderen Satz von Funktionen. Git macht den Prozess, diese Zweige zu erstellen und sie anschließend wieder zusammenzufügen, fast schmerzlos.

Git hat 3 verschiedene "Bereiche" für Ihren Code:

- **Arbeitsverzeichnis** : Der Bereich, in dem Sie Ihre gesamte Arbeit erledigen werden (Erstellen, Bearbeiten, Löschen und Organisieren von Dateien).
- **Bereitstellungsbereich** : Der Bereich, in dem Sie die Änderungen auflisten, die Sie am Arbeitsverzeichnis vorgenommen haben
- **Repository** : Hier speichert Git die von Ihnen vorgenommenen Änderungen dauerhaft in verschiedenen Versionen des Projekts

Git wurde ursprünglich zur Verwaltung der Linux-Kernel-Quelle erstellt. Durch die Vereinfachung ermutigen sie kleine Commits, fälschen von Projekten und verschmelzen zwischen den Gabeln und haben viele kurzlebige Zweige.

Die größte Änderung für Menschen, die an CVS oder Subversion gewöhnt sind, ist, dass jeder Checkout nicht nur den Quellbaum, sondern auch den gesamten Verlauf des Projekts enthält. Übliche Vorgänge wie das Vergleichen von Revisionen, das Auschecken älterer Revisionen, das Festschreiben (an Ihre lokale Historie), das Erstellen einer Verzweigung, das Auschecken einer anderen Verzweigung, das Zusammenführen von Verzweigungen oder Patchdateien können lokal erfolgen, ohne dass Sie mit einem zentralen Server kommunizieren müssen. Damit ist die größte Quelle für Latenz und Unzuverlässigkeit beseitigt. Die Kommunikation mit dem "Upstream" - Repository ist nur erforderlich, um die neuesten Änderungen zu erhalten und die lokalen Änderungen für andere Entwickler zu veröffentlichen. Dies macht aus einer technischen Einschränkung (wer auch immer das Projekt das Projekt besitzt) eine organisatorische Entscheidung (Ihr "Upstream" ist derjenige, mit dem Sie synchronisieren möchten).

Versionen

Ausführung	Veröffentlichungsdatum
2.13	2017-05-10
2.12	2017-02-24
2.11.1	2017-02-02
2.11	2016-11-29
2.10.2	2016-10-28
2.10	2016-09-02
2,9	2016-06-13
2.8	2016-03-28
2,7	2015-10-04
2.6	2015-09-28
2,5	2015-07-27
2.4	2015-04-30
2.3	2015-02-05
2.2	2014-11-26
2.1	2014-08-16
2,0	2014-05-28
1,9	2014-02-14
1.8.3	2013-05-24
1.8	2012-10-21
1.7.10	2012-04-06
1.7	2010-02-13
1.6.5	2009-10-10
1.6.3	2009-05-07
1.6	2008-08-17
1.5.3	2007-09-02

Ausführung	Veröffentlichungsdatum
1,5	2007-02-14
1.4	2006-06-10
1.3	2006-04-18
1.2	2006-02-12
1.1	2006-01-08
1,0	2005-12-21
0,99	2005-07-11

Examples

Erstellen Sie Ihr erstes Repository, fügen Sie dann Dateien hinzu und legen Sie sie fest

Überprüfen Sie zunächst in der Befehlszeile, ob Sie Git installiert haben:

Auf allen Betriebssystemen:

```
git --version
```

Auf UNIX-ähnlichen Betriebssystemen:

```
which git
```

Wenn nichts zurückgegeben wird oder der Befehl nicht erkannt wird, müssen Sie möglicherweise Git auf Ihrem System installieren, indem Sie das Installationsprogramm herunterladen und ausführen. Auf der Git-Homepage finden Sie außergewöhnlich klare und einfache Installationsanweisungen.

Konfigurieren Sie nach der Installation von Git Ihren Benutzernamen und Ihre E-Mail-Adresse . Tun Sie dies, *bevor Sie* ein Commit ausführen.

Navigieren Sie nach der Installation von Git zu dem Verzeichnis, das Sie der Versionskontrolle zuordnen möchten, und erstellen Sie ein leeres Git-Repository:

```
git init
```

Dadurch wird ein versteckter Ordner, `.git` , erstellt, der die für Git funktionierenden Rohrleitungen enthält.

Prüfen Sie anschließend, welche Dateien Git Ihrem neuen Repository hinzufügen wird. Dieser

Schritt ist besondere Sorgfalt wert:

```
git status
```

Überprüfen Sie die Ergebnisliste der Dateien. Sie können Git mitteilen, welche der Dateien in die Versionskontrolle eingefügt werden soll (vermeiden Sie das Hinzufügen von Dateien mit vertraulichen Informationen wie Kennwörter oder Dateien, die das Repo nur durcheinanderbringen)

```
git add <file/directory name #1> <file/directory name #2> < ... >
```

Wenn alle Dateien in der Liste für alle Benutzer freigegeben werden sollen, die Zugriff auf das Repository haben, wird ein einzelner Befehl alles in Ihr aktuelles Verzeichnis und dessen Unterverzeichnisse hinzufügen:

```
git add .
```

Dadurch werden alle Dateien "in Stufe" versetzt , die der Versionskontrolle hinzugefügt werden, um sie für den ersten Commit vorzubereiten.

Erstellen Sie für Dateien, die Sie niemals unter Versionskontrolle haben möchten, eine Datei mit dem Namen `.gitignore` bevor Sie den Befehl `add .gitignore` .

Übertragen Sie alle hinzugefügten Dateien zusammen mit einer Commit-Nachricht:

```
git commit -m "Initial commit"
```

Dadurch wird ein neues Commit mit der angegebenen Nachricht erstellt. Ein Commit ist wie eine Sicherung oder Momentaufnahme Ihres gesamten Projekts. Sie können es jetzt in ein Remote-Repository pushen oder hochladen, und später können Sie bei Bedarf darauf zurückspringen. Wenn Sie den Parameter `-m` angeben, wird Ihr Standardeditor geöffnet, und Sie können die Festschreibungsnachricht dort bearbeiten und speichern.

Fernbedienung hinzufügen

Um eine neue Fernbedienung hinzuzufügen, verwenden Sie den Befehl `git remote add` im Terminal, in dem sich Ihr Repository befindet.

Der Befehl `git remote add` benötigt zwei Argumente:

1. Ein entfernter Name, z. B. `origin`
2. Eine entfernte URL, zum Beispiel `https://<your-git-service-address>/user/repo.git`

```
    git remote add origin https://<your-git-service-address>/owner/repository.git
```

HINWEIS: Bevor Sie die Fernbedienung hinzufügen, müssen Sie das erforderliche Repository in Ihrem git-Dienst erstellen. Nach dem Hinzufügen der Fernbedienung können Sie Commits per Push / Pull übertragen.

Klonen Sie ein Repository

Der Befehl `git clone` wird verwendet, um ein vorhandenes Git-Repository von einem Server auf den lokalen Computer zu kopieren.

So kopieren Sie beispielsweise ein GitHub-Projekt:

```
cd <path where you'd like the clone to create a directory>
git clone https://github.com/username/projectname.git
```

So klonen Sie ein BitBucket-Projekt:

```
cd <path where you'd like the clone to create a directory>
git clone https://yourusername@bitbucket.org/username/projectname.git
```

Dadurch wird auf dem lokalen Computer ein Verzeichnis mit dem Namen `projectname` erstellt, das alle Dateien im fernen Git-Repository enthält. Dazu gehören Quelldateien für das Projekt sowie ein `.git` Unterverzeichnis, das die gesamte Historie und Konfiguration für das Projekt enthält.

Um einen anderen Namen des Verzeichnisses anzugeben, z. B. `MyFolder` :

```
git clone https://github.com/username/projectname.git MyFolder
```

Oder im aktuellen Verzeichnis klonen:

```
git clone https://github.com/username/projectname.git .
```

Hinweis:

1. Beim Klonen in ein angegebenes Verzeichnis muss das Verzeichnis leer oder nicht vorhanden sein.
2. Sie können auch die `ssh` Version des Befehls verwenden:

   ```
   git clone git@github.com:username/projectname.git
   ```

Die `https` Version und die `ssh` Version sind gleichwertig. Einige Hosting-Dienste wie GitHub empfehlen jedoch die Verwendung von `https` anstelle von `ssh` .

Upstream-Fernbedienung einrichten

Wenn Sie einen Fork geklont haben (z. B. ein Open-Source-Projekt auf Github), haben Sie möglicherweise keinen Push-Zugriff auf das Upstream-Repository. Sie benötigen also sowohl Ihren Fork als auch das Upstream-Repository.

Überprüfen Sie zuerst die Namen der Fernbedienung:

```
$ git remote -v
```

```
origin    https://github.com/myusername/repo.git (fetch)
origin    https://github.com/myusername/repo.git (push)
upstream  # this line may or may not be here
```

Wenn `upstream` bereits vorhanden ist (es gibt *einige* Git-Versionen), müssen Sie die URL festlegen (derzeit ist sie leer):

```
$ git remote set-url upstream https://github.com/projectusername/repo.git
```

Wenn der Upstream **nicht** vorhanden ist oder wenn Sie auch die Gabel eines Freundes / Kollegen hinzufügen möchten (derzeit sind diese nicht vorhanden):

```
$ git remote add upstream https://github.com/projectusername/repo.git
$ git remote add dave https://github.com/dave/repo.git
```

Code teilen

Um Ihren Code freizugeben, erstellen Sie ein Repository auf einem Remote-Server, auf den Sie Ihr lokales Repository kopieren.

Um den Speicherplatz auf dem Remote-Server zu minimieren, erstellen Sie ein einfaches Repository: eines, das nur die `.git` Objekte enthält und keine Arbeitskopie im Dateisystem erstellt. Als Bonus haben Sie diese Fernbedienung als Upstream-Server festgelegt, um Updates problemlos mit anderen Programmierern gemeinsam zu nutzen.

Auf dem Remote-Server:

```
git init --bare /path/to/repo.git
```

Auf dem lokalen Rechner:

```
git remote add origin ssh://username@server:/path/to/repo.git
```

(Beachten Sie, dass `ssh:` nur eine Möglichkeit ist, auf das Remote-Repository zuzugreifen.)

Kopieren Sie nun Ihr lokales Repository in das Remote:

```
git push --set-upstream origin master
```

Hinzufügen `--set-upstream` (oder `-u`) erstellt eine stromaufwärtige (tracking) Referenz , die durch das Argument verwendet wird losen Git Befehle, zB `git pull` .

Festlegen Ihres Benutzernamens und Ihrer E-Mail

```
You need to set who you are *before* creating any commit.  That will allow commits to have the
right author name and email associated to them.
```

Es hat nichts mit Authentifizierung zu tun, wenn Sie zu einem Remote-Repository wechseln

(z. B. wenn Sie mit Ihrem GitHub-, BitBucket- oder GitLab-Konto zu einem Remote-Repository wechseln).

Um diese Identität für *alle* Repositorys `git config --global` , verwenden Sie `git config --global` Dadurch wird die Einstellung in der `.gitconfig` Datei Ihres Benutzers gespeichert: z. B. `$HOME/.gitconfig` oder für Windows `%USERPROFILE%\.gitconfig` .

```
git config --global user.name "Your Name"
git config --global user.email mail@example.com
```

Um eine Identität für ein einzelnes Repository zu deklarieren, verwenden Sie `git config` innerhalb eines Repos.
Dadurch wird die Einstellung im einzelnen Repository in der Datei `$GIT_DIR/config` . zB `/path/to/your/repo/.git/config` .

```
cd /path/to/my/repo
git config user.name "Your Login At Work"
git config user.email mail_at_work@example.com
```

In der Konfigurationsdatei eines Repositorys gespeicherte Einstellungen haben Vorrang vor der globalen Konfiguration, wenn Sie dieses Repository verwenden.

Tipps: Wenn Sie unterschiedliche Identitäten haben (eine für Open-Source-Projekte, eine für die Arbeit, eine für private Repos, ...), und Sie nicht vergessen möchten, für jedes Repo, an dem Sie arbeiten, die richtige zu wählen :

- **Entfernen Sie eine globale Identität**

  ```
  git config --global --remove-section user.name
  git config --global --remove-section user.email
  ```

2.8

- Um zu erzwingen, dass git nur in den Einstellungen eines Repositorys nach Ihrer Identität sucht, nicht in der globalen Konfiguration:

  ```
    git config --global user.useConfigOnly true
  ```

Wenn Sie also vergessen, `user.name` und `user.email` für ein bestimmtes Repository `user.name` und ein Commit `user.email` `user.name` `user.email` :

```
no name was given and auto-detection is disabled
no email was given and auto-detection is disabled
```

Einen Befehl kennenlernen

Um weitere Informationen zu jedem git-Befehl zu erhalten, z. B. Details zu den Funktionen des

Befehls, verfügbaren Optionen und anderer Dokumentation, verwenden Sie die Option `--help` oder den Befehl `help` .

Um beispielsweise alle verfügbaren Informationen zum Befehl `git diff` , verwenden Sie:

```
git diff --help
git help diff
```

Um alle verfügbaren Informationen zum `status` abzurufen, verwenden Sie Folgendes:

```
git status --help
git help status
```

Wenn Sie nur eine schnelle Hilfe benötigen, um die Bedeutung der am häufigsten verwendeten Befehlszeilenflags anzuzeigen, verwenden Sie `-h` :

```
git checkout -h
```

Richten Sie SSH für Git ein

Wenn Sie **Windows verwenden,** öffnen Sie Git Bash . Wenn Sie **Mac** oder **Linux verwenden,** öffnen Sie Ihr Terminal.

Bevor Sie einen SSH-Schlüssel generieren, können Sie prüfen, ob bereits vorhandene SSH-Schlüssel vorhanden sind.

Listen Sie den Inhalt Ihres `~/.ssh` Verzeichnisses auf:

```
$ ls -al ~/.ssh
# Lists all the files in your ~/.ssh directory
```

Überprüfen Sie die Verzeichnisliste, um festzustellen, ob Sie bereits über einen öffentlichen SSH-Schlüssel verfügen. Standardmäßig sind die Dateinamen der öffentlichen Schlüssel eine der folgenden:

```
id_dsa.pub
id_ecdsa.pub
id_ed25519.pub
id_rsa.pub
```

Wenn ein vorhandenes Paar aus öffentlichen und privaten Schlüsseln angezeigt wird, das Sie in Ihrem Bitbucket-, GitHub-Konto (oder einem ähnlichen Konto) verwenden möchten, können Sie den Inhalt der `id_*.pub` Datei kopieren.

Wenn nicht, können Sie mit dem folgenden Befehl ein neues öffentliches und privates Schlüsselpaar erstellen:

```
$ ssh-keygen
```

Drücken Sie die Eingabetaste, um den Standardspeicherort zu übernehmen. Geben Sie eine Passphrase ein und geben Sie sie erneut ein, wenn Sie dazu aufgefordert werden, oder lassen Sie sie leer.

Stellen Sie sicher, dass Ihr SSH-Schlüssel zum ssh-agent hinzugefügt wird. Starten Sie den ssh-agent im Hintergrund, falls er noch nicht läuft:

```
$ eval "$(ssh-agent -s)"
```

Fügen Sie Ihren SSH-Schlüssel zum ssh-agent hinzu. Beachten Sie, dass Sie `id_rsa` im Befehl durch den Namen Ihrer **privaten Schlüsseldatei** ersetzen `id_rsa` :

```
$ ssh-add ~/.ssh/id_rsa
```

Wenn Sie den Upstream eines vorhandenen Repositorys von HTTPS zu SSH ändern möchten, können Sie den folgenden Befehl ausführen:

```
$ git remote set-url origin ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Um ein neues Repository über SSH zu klonen, können Sie den folgenden Befehl ausführen:

```
$ git clone ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Git Installation

Lass uns etwas Git benutzen. Das Wichtigste zuerst - Sie müssen es installieren. Sie können es auf verschiedene Arten bekommen; Die beiden wichtigsten sind die Installation von der Quelle oder die Installation eines vorhandenen Pakets für Ihre Plattform.

Installation von der Quelle

Wenn Sie können, ist es generell sinnvoll, Git von Source zu installieren, da Sie die aktuellste Version erhalten. Jede Version von Git enthält in der Regel nützliche Verbesserungen der Benutzeroberfläche. Die neueste Version ist daher oft die beste Route, wenn Sie sich beim Erstellen einer Software nach dem Quellcode wohl fühlen. Es ist auch so, dass viele Linux-Distributionen sehr alte Pakete enthalten. Wenn Sie sich also nicht auf einer sehr aktuellen Distribution befinden oder Backports verwenden, ist die Installation von der Quelle möglicherweise die beste Wahl.

Zur Installation von Git benötigen Sie die folgenden Bibliotheken, auf die Git angewiesen ist: curl, zlib, openssl, expat und libiconv. Wenn Sie sich beispielsweise auf einem System mit yum (wie Fedora) oder apt-get (wie einem Debian-basierten System) befinden, können Sie alle diese Abhängigkeiten mit einem der folgenden Befehle installieren:

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

$ apt-get install libcurl4-gnutls-dev libexpat1-dev gettext \
```

```
  libz-dev libssl-dev
```

Wenn Sie alle notwendigen Abhängigkeiten haben, können Sie den neuesten Schnappschuss von der Git-Website abrufen:

http://git-scm.com/download Kompilieren und installieren Sie dann:

```
$ tar -zxf git-1.7.2.2.tar.gz
$ cd git-1.7.2.2
$ make prefix=/usr/local all
$ sudo make prefix=/usr/local install
```

Nachdem dies erledigt ist, können Sie Git über Git selbst für Updates erhalten:

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

Installation unter Linux

Wenn Sie Git über ein binäres Installationsprogramm unter Linux installieren möchten, können Sie dies in der Regel über das grundlegende Paketverwaltungstool Ihrer Distribution tun. Wenn Sie auf Fedora sind, können Sie yum verwenden:

```
$ yum install git
```

Wenn Sie sich in einer Debian-basierten Distribution wie Ubuntu befinden, versuchen Sie apt-get:

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

Installation auf dem Mac

Es gibt drei einfache Möglichkeiten, Git auf einem Mac zu installieren. Am einfachsten ist es, das grafische Git-Installationsprogramm zu verwenden, das Sie von der SourceForge-Seite herunterladen können.

http://sourceforge.net/projects/git-osx-installer/

Abbildung 1-7. Git OS X-Installationsprogramm. Die andere Möglichkeit ist, Git über MacPorts (http://www.macports.org) zu installieren. Wenn Sie MacPorts installiert haben, installieren Sie Git via

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

Sie müssen nicht alle Extras hinzufügen, Sie sollten jedoch + svn hinzufügen, falls Sie Git mit Subversion-Repositorys verwenden müssen (siehe Kapitel 8).

Homebrew (http://brew.sh/) ist eine weitere Alternative zur Installation von Git. Wenn Sie Homebrew installiert haben, installieren Sie Git via

```
$ brew install git
```

Installation unter Windows

Die Installation von Git unter Windows ist sehr einfach. Das msysGit-Projekt hat eine der einfacheren Installationsprozeduren. Laden Sie einfach die Installer-Exe-Datei von der GitHub-Seite herunter und führen Sie sie aus:

```
http://msysgit.github.io
```

Nach der Installation haben Sie sowohl eine Befehlszeilenversion (einschließlich eines SSH-Clients, der sich später als nützlich erweisen wird) als auch die Standard-GUI.

Hinweis zur Verwendung von Windows: Sie sollten Git mit der mitgelieferten msysGit-Shell (Unix-Stil) verwenden. Sie ermöglicht die Verwendung der komplexen Befehlszeilen in diesem Buch. Wenn Sie aus irgendeinem Grund die native Windows-Shell- / Befehlszeilenkonsole verwenden müssen, müssen Sie Anführungszeichen anstelle von einfachen Anführungszeichen verwenden (für Parameter mit Leerzeichen) und Sie müssen die Parameter angeben, die mit dem Akzentflexakzent (^) enden) wenn sie zuletzt in der Zeile stehen, da es ein Fortführungssymbol in Windows ist.

Erste Schritte mit Git online lesen: https://riptutorial.com/de/git/topic/218/erste-schritte-mit-git

Kapitel 2: .mailmap-Datei: Verknüpfen von Mitwirkenden und E-Mail-Aliasnamen

Syntax

- # Ersetzen Sie nur E-Mail-Adressen
  <primary@example.org> <alias@example.org>
- # Name durch E-Mail-Adresse ersetzen
  Mitwirkender <primary@example.org>
- # Mehrere Aliase unter einem Namen und einer E-Mail zusammenführen
  # Beachten Sie, dass dies nicht 'Other <alias2@example.org>' zugeordnet wird.
  Mitwirkender <primary@example.org> <alias1@example.org> Mitwirkender
  <alias2@example.org>

Bemerkungen

Eine `.mailmap` Datei kann in einem beliebigen Texteditor erstellt werden und ist nur eine `.mailmap` Textdatei, die optionale Namen der Bearbeiter, primäre E-Mail-Adressen und deren Aliasnamen enthält. Es muss im Stammverzeichnis des Projekts neben dem Verzeichnis `.git` .

`git shortlog` Sie, dass dies lediglich die visuelle Ausgabe von Befehlen wie `git shortlog` oder `git log --use-mailmap` . Dadurch wird der Commit-Verlauf **nicht** umgeschrieben oder Commits mit unterschiedlichen Namen und / oder E-Mail-Adressen verhindert.

Um das Festschreiben von Informationen wie E-Mail-Adressen zu verhindern, sollten Sie stattdessen Git Hooks verwenden.

Examples

Einbinden von Teilnehmern nach Aliasnamen, um die Commit-Anzahl im Shortlog anzuzeigen.

Wenn Mitwirkende von verschiedenen Maschinen oder Betriebssystemen zu einem Projekt hinzufügen, kann es vorkommen, dass sie unterschiedliche E-Mail-Adressen oder Namen dafür verwenden, wodurch Teilnehmerlisten und Statistiken fragmentiert werden.

`git shortlog -sn` , um eine Liste der Mitwirkenden und die Anzahl der Commits zu erhalten, kann dies zu folgender Ausgabe führen:

```
Patrick Rothfuss 871
Elizabeth Moon 762
E. Moon 184
Rothfuss, Patrick 90
```

Diese Fragmentierung / Trennung kann angepasst werden, indem eine `.mailmap` Text-Datei `.mailmap` , die E-Mail-Zuordnungen enthält.

Alle Namen und E-Mail-Adressen, die in einer Zeile aufgeführt sind, werden jeweils der zuerst genannten Entität zugeordnet.

Für das obige Beispiel könnte ein Mapping folgendermaßen aussehen:

```
Patrick Rothfuss <fussy@kingkiller.com> Rothfuss, Patrick <fussy@kingkiller.com>
Elizabeth Moon <emoon@marines.mil> E. Moon <emoon@scifi.org>
```

Sobald diese Datei im Stammverzeichnis des Projekts vorhanden ist, führt die `git shortlog -sn` Ausführung von `git shortlog -sn` einer komprimierten Liste:

```
Patrick Rothfuss 961
Elizabeth Moon 946
```

.mailmap-Datei: Verknüpfen von Mitwirkenden und E-Mail-Aliasnamen online lesen: https://riptutorial.com/de/git/topic/1270/-mailmap-datei--verknupfen-von-mitwirkenden-und-e-mail-aliasnamen

Kapitel 3: Aktualisieren Sie den Objektnamen in der Referenz

Examples

Aktualisieren Sie den Objektnamen in der Referenz

Benutzen

Aktualisieren Sie den Objektnamen, der in der Referenz gespeichert ist

ZUSAMMENFASSUNG

```
git update-ref [-m <reason>] (-d <ref> [<oldvalue>] | [--no-deref] [--create-reflog] <ref>
<newvalue> [<oldvalue>] | --stdin [-z])
```

Allgemeine Syntax

1. Aktualisieren Sie den aktuellen Zweigkopf auf das neue Objekt, indem Sie die symbolischen Refs referenzieren.

   ```
   git update-ref HEAD <newvalue>
   ```

2. Speichert den `newvalue` in `ref` , nachdem `newvalue` wurde, dass der aktuelle Wert des `ref` mit dem `oldvalue` Wert `oldvalue` .

   ```
   git update-ref refs/head/master <newvalue> <oldvalue>
   ```

   Die obige Syntax aktualisiert den `newvalue` nur dann auf `newvalue` , wenn der aktuelle Wert `oldvalue` .

Verwenden Sie das Flag `-d` um den benannten `<ref>` `<oldvalue>` nachdem Sie überprüft haben, dass er noch `<oldvalue>` enthält.

Verwenden Sie `--create-reflog` , update-ref erstellt einen Reflog für jeden Ref, selbst wenn normalerweise kein Reflog erstellt wird.

Verwenden Sie das Flag `-z` , um im NUL-terminierten Format anzugeben, das Werte wie "Aktualisieren", "Erstellen", "Löschen", "Überprüfen" enthält.

Aktualisieren

Setzen Sie `<ref>` auf `<newvalue>` nachdem Sie `<oldvalue>` überprüft `<oldvalue>` , falls angegeben.

Geben Sie eine Null `<newvalue>` an, um sicherzustellen, dass der Ref nach dem Update nicht vorhanden ist, und / oder einen Null `<oldvalue>` , um sicherzustellen, dass der Ref vor dem Update nicht existiert.

Erstellen

Erstellen Sie `<ref>` mit `<newvalue>` nachdem Sie überprüft haben, dass es nicht existiert. Der angegebene `<newvalue>` darf nicht Null sein.

Löschen

Löschen Sie `<ref>` nachdem Sie `<oldvalue>` , dass es mit `<oldvalue>` , falls vorhanden. Wenn angegeben, darf `<oldvalue>` nicht Null sein.

Überprüfen

Überprüfen Sie `<ref>` gegen `<oldvalue>` , ändern Sie ihn jedoch nicht. Wenn `<oldvalue>` Null ist oder fehlt, darf der Ref nicht existieren.

Aktualisieren Sie den Objektnamen in der Referenz online lesen:
https://riptutorial.com/de/git/topic/7579/aktualisieren-sie-den-objektnamen-in-der-referenz

Kapitel 4: Aliase

Examples

Einfache Aliase

Es gibt zwei Möglichkeiten zum Erstellen von Aliasnamen in Git:

- mit der Datei `~/.gitconfig` :

```
[alias]
    ci = commit
    st = status
    co = checkout
```

- mit der Kommandozeile:

```
git config --global alias.ci "commit"
git config --global alias.st "status"
git config --global alias.co "checkout"
```

Nachdem der Alias erstellt wurde, geben Sie Folgendes ein:

- `git ci` statt `git commit` ,
- `git st` statt `git status` ,
- `git co` statt `git checkout` .

Wie bei normalen git-Befehlen können Aliase neben Argumenten verwendet werden. Zum Beispiel:

```
git ci -m "Commit message..."
git co -b feature-42
```

Vorhandene Aliase auflisten / suchen

Sie können vorhandene Git-Aliasnamen mit `--get-regexp` :

```
$ git config --get-regexp '^alias\.'
```

Aliase werden gesucht

`.gitconfig` Ihrer `.gitconfig` unter `[alias]` Folgendes hinzu, um Aliase zu suchen :

```
aliases = !git config --list | grep ^alias\\. | cut -c 7- | grep -Ei --color \"$1\" "#"
```

Dann kannst du:

- `git aliases` - zeige ALLE Aliase
- `git aliases commit` - nur Aliase, die "commit" enthalten

Erweiterte Aliase

Mit Git können Sie Nicht-Git-Befehle und vollständige `sh` Shell-Syntax in Ihren Aliasnamen verwenden, wenn Sie ihnen ein Präfix voranstellen `!` .

In Ihrer `~/.gitconfig` Datei:

```
[alias]
    temp = !git add -A && git commit -m "Temp"
```

Die Tatsache, dass die vollständige Shell-Syntax in diesen vorangestellten Aliases verfügbar ist, bedeutet auch, dass Sie Shell-Funktionen verwenden können, um komplexere Aliase zu erstellen, z. B. solche, die Befehlszeilenargumente verwenden:

```
[alias]
    ignore = "!f() { echo $1 >> .gitignore; }; f"
```

Der obige Alias definiert die `f` Funktion und führt sie dann mit allen Argumenten aus, die Sie an den Alias übergeben. `.gitignore` `git ignore .tmp/ .gitignore` würde dies zu Ihrer `.gitignore` Datei `.tmp/ .gitignore` .

Tatsächlich ist dieses Muster so nützlich, dass Git `$1` , `$2` usw. für Sie definiert, sodass Sie nicht einmal eine spezielle Funktion dafür definieren müssen. (Beachten Sie jedoch, dass Git die Argumente trotzdem anhängt, auch wenn Sie über diese Variablen darauf zugreifen. Daher möchten Sie am Ende einen Dummy-Befehl hinzufügen.)

Beachten Sie, dass Aliasnamen vorangestellt sind `!` Auf diese Weise werden Sie vom Stammverzeichnis Ihrer Git-Kasse aus ausgeführt, auch wenn sich Ihr aktuelles Verzeichnis tiefer in der Baumstruktur befindet. Dies kann eine nützliche Methode sein, um einen Befehl von der Wurzel aus auszuführen, ohne dort explizit `cd` ausführen zu müssen.

```
[alias]
    ignore = "! echo $1 >> .gitignore"
```

Verfolgte Dateien vorübergehend ignorieren

So markieren Sie eine Datei vorübergehend als ignoriert (übergeben Sie die Datei als Parameter an einen Alias) - Geben Sie Folgendes ein:

```
unwatch = update-index --assume-unchanged
```

Um die Datei erneut zu starten, geben Sie Folgendes ein:

```
watch = update-index --no-assume-unchanged
```

Um alle Dateien aufzulisten, die vorübergehend ignoriert wurden, geben Sie Folgendes ein:

```
unwatched = "!git ls-files -v | grep '^[[:lower:]]'"
```

Um die nicht abgemeldete Liste zu löschen, geben Sie Folgendes ein:

```
watchall = "!git unwatched | xargs -L 1 -I % sh -c 'git watch `echo % | cut -c 2-`'"
```

Beispiel für die Verwendung der Aliase:

```
git unwatch my_file.txt
git watch my_file.txt
git unwatched
git watchall
```

Hübsches Protokoll mit Zweigdiagramm anzeigen

```
[alias]
  logp=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

  lg =  log --graph --date-order --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
  lgb = log --graph --date-order --branches --first-parent \
     --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
 lga = log --graph --date-order --all \
   --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(bold
cyan)<%an>%Creset'
```

Hier eine Erläuterung der Optionen und des Platzhalters, die im `--pretty` Format verwendet werden (vollständige Liste mit `git help log`)

--graph - Zeichne den Commit-Baum

--date-order - Verwenden Sie, wenn möglich, einen Commit-Zeitstempel

- first-parent - Folgen Sie nur dem ersten übergeordneten Knoten im Zusammenführungsknoten.

--branches - Zeigt alle lokalen Zweigstellen an (standardmäßig wird nur der aktuelle Zweig angezeigt)

--all - Zeigt alle lokalen und entfernten Niederlassungen an

% h - Hashwert für Festschreiben (abgekürzt)

% ad - Datumsstempel (Autor)

% an - Autor Benutzername

% an - Commit-Benutzername

% C (automatisch) - Zum Verwenden der im Abschnitt [color] definierten Farben

% Creset - zum Zurücksetzen der Farbe

% d - --decorate (Zweig- und Tag-Namen)

% s - Commit-Nachricht

% ad - Autorendatum (folgt - Direktive -Datum) (und kein Commiter-Datum)

% an - Autorenname (kann% cn für Commitername sein)

Code aktualisieren, während eine lineare Historie beibehalten wird

Manchmal müssen Sie eine lineare (nicht verzweigte) Historie Ihrer Code-Commits führen. Wenn Sie eine Zeit lang an einem Zweig arbeiten, kann dies schwierig sein, wenn Sie einen regulären `git pull` da dies ein Zusammenführen mit dem Upstream aufzeichnet.

```
[alias]
  up = pull --rebase
```

Dies wird mit Ihrer Upstream-Quelle aktualisiert und anschließend alle Arbeiten erneut angewendet, die Sie nicht auf das gesetzt haben, was Sie zuvor heruntergezogen haben.

Benutzen:

```
git up
```

Sehen Sie, welche Dateien von Ihrer .gitignore-Konfiguration ignoriert werden

```
[ alias ]

    ignored = ! git ls-files --others --ignored --exclude-standard --directory \
            && git ls-files --others -i --exclude-standard
```

Zeigt eine Zeile pro Datei an, damit Sie grep (nur Verzeichnisse):

```
$ git ignored | grep '/$'
.yardoc/
doc/
```

Oder zählen:

```
~$ git ignored | wc -l
199811                # oops, my home directory is getting crowded
```

Inszenieren von inszenierten Dateien

Um Dateien zu entfernen, die mithilfe des `git reset` Commits bereitgestellt werden, hat `reset`

normalerweise eine Vielzahl von Funktionen, abhängig von den dafür bereitgestellten Argumenten. Um das Gerät vollständig unstage alle Dateien aufgeführt, können wir die Verwendung von git Aliase machen , um einen neuen Alias zu erstellen , die verwendet `reset` , aber jetzt brauchen wir nicht zu erinnern , die richtigen Argumente zu liefern , um `reset` .

```
git config --global alias.unstage "reset --"
```

Wenn Sie **nun** Dateien in die **Bühne bringen** möchten, **geben** Sie `git unstage` Nun können Sie **loslegen** .

Aliase online lesen: https://riptutorial.com/de/git/topic/337/aliase

Kapitel 5: Analysieren von Arten von Workflows

Bemerkungen

Die Verwendung von Versionskontrollsoftware wie Git mag anfangs etwas beängstigend sein, aber das intuitive Design, das sich auf Verzweigungen spezialisiert, ermöglicht eine Reihe verschiedener Arten von Workflows. Wählen Sie das für Ihr Entwicklungsteam passende aus.

Examples

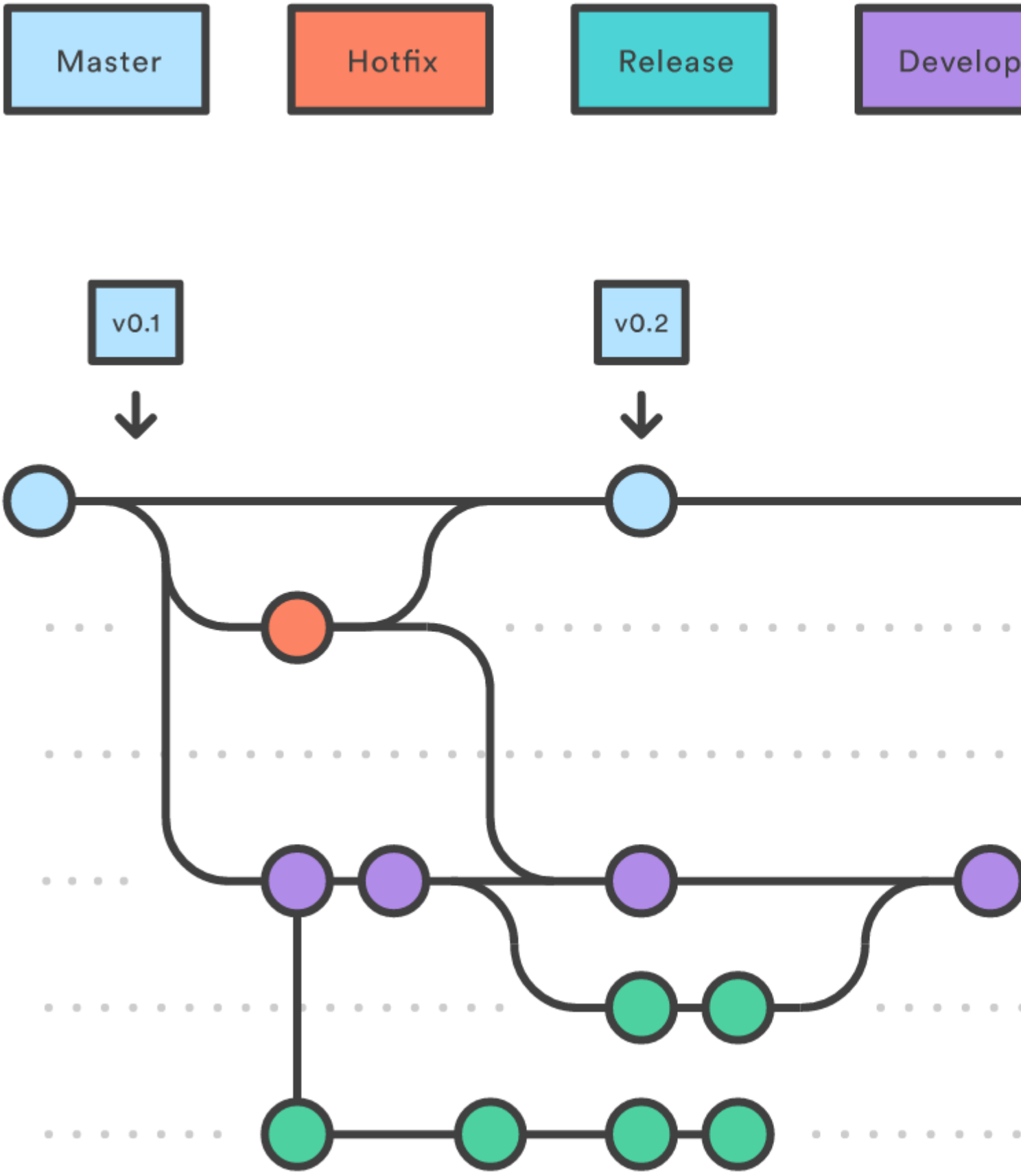
Gitflow Workflow

Ursprünglich von Vincent Driessen vorgeschlagen, ist Gitflow ein Entwicklungsworkflow, der Git und mehrere vordefinierte Zweige verwendet. Dies kann als Sonderfall des Feature Branch Workflow betrachtet werden .

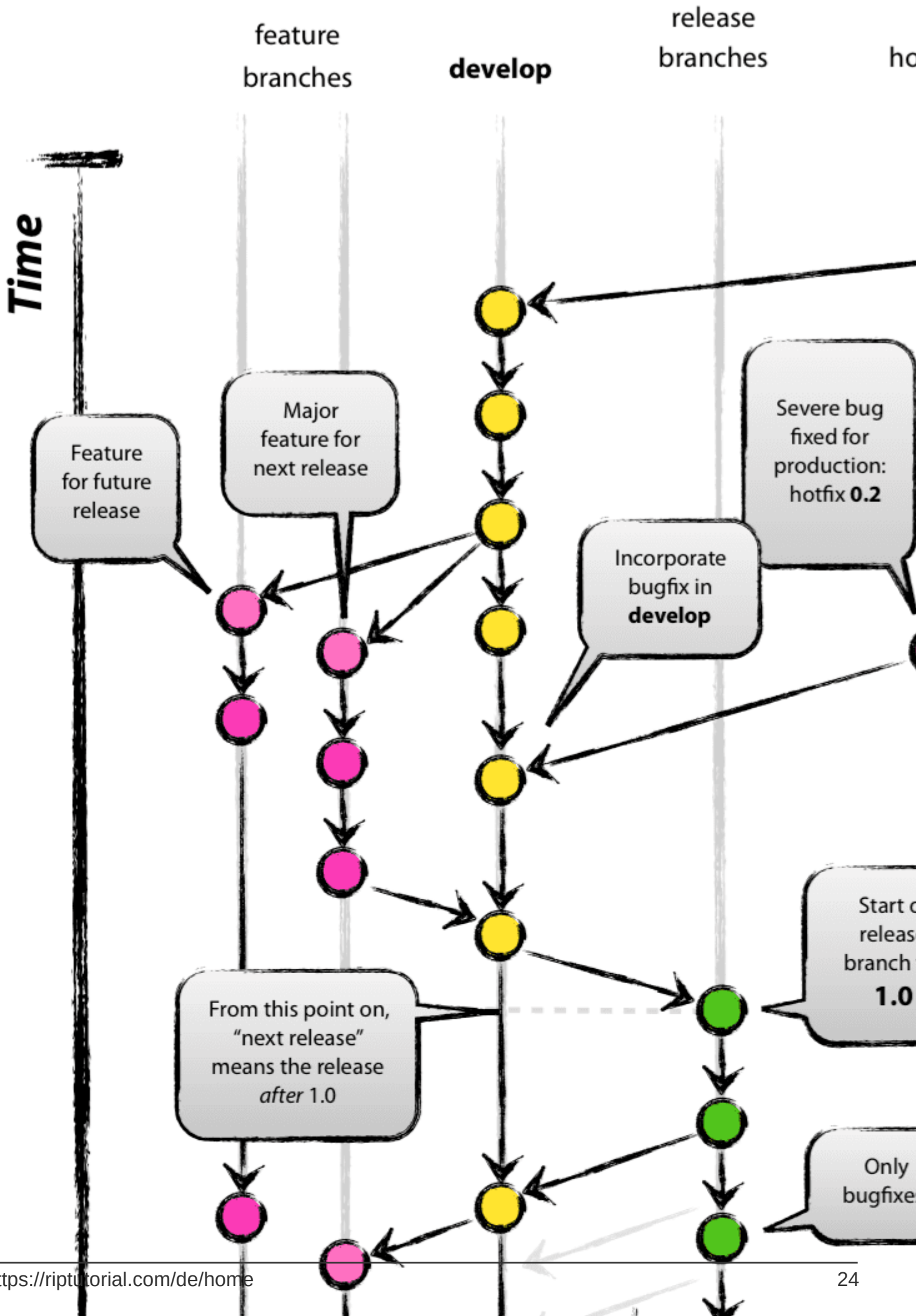
Der Grundgedanke dabei ist, separate Niederlassungen für bestimmte Teile in der Entwicklung zu reservieren:

- `master` ist immer der aktuellste *Produktionscode* . Experimenteller Code gehört nicht hierher.
- `develop` enthält alle neuesten *Entwicklungen* . Diese Entwicklungsänderungen können so ziemlich alles sein, aber größere Funktionen sind ihren eigenen Zweigen vorbehalten. Code wird hier vor der Veröffentlichung / Bereitstellung immer bearbeitet und in das `release` eingebunden.
- `hotfix` Zweige `hotfix` kleinere Fehlerbehebungen, die nicht bis zur nächsten Version warten können. `hotfix` Zweige kommen von `master` und werden wieder in `master` und `develop` .
- `release` Zweigen werden neue Entwicklungen von der `develop` bis zum `master` freigegeben. Alle Änderungen in letzter Minute, z. B. die Versionsnummern zum Bumping, werden im Release-Zweig vorgenommen und dann wieder mit `master` und `develop` . Bei der Bereitstellung einer neuen Version sollte `master` mit der aktuellen Versionsnummer gekennzeichnet sein (z. B. mit semantischer Versionierung), um später darauf zurückgreifen zu können.
- `feature` Zweige sind für größere Features reserviert. Diese sind speziell in bestimmten Zweigen entwickelt und integriert mit der `develop` , wenn Sie fertig. Dedicated `feature` Zweige helfen Entwicklung zu trennen und *done* Funktionen unabhängig voneinander einsetzen zu können.

Eine visuelle Darstellung dieses Modells:

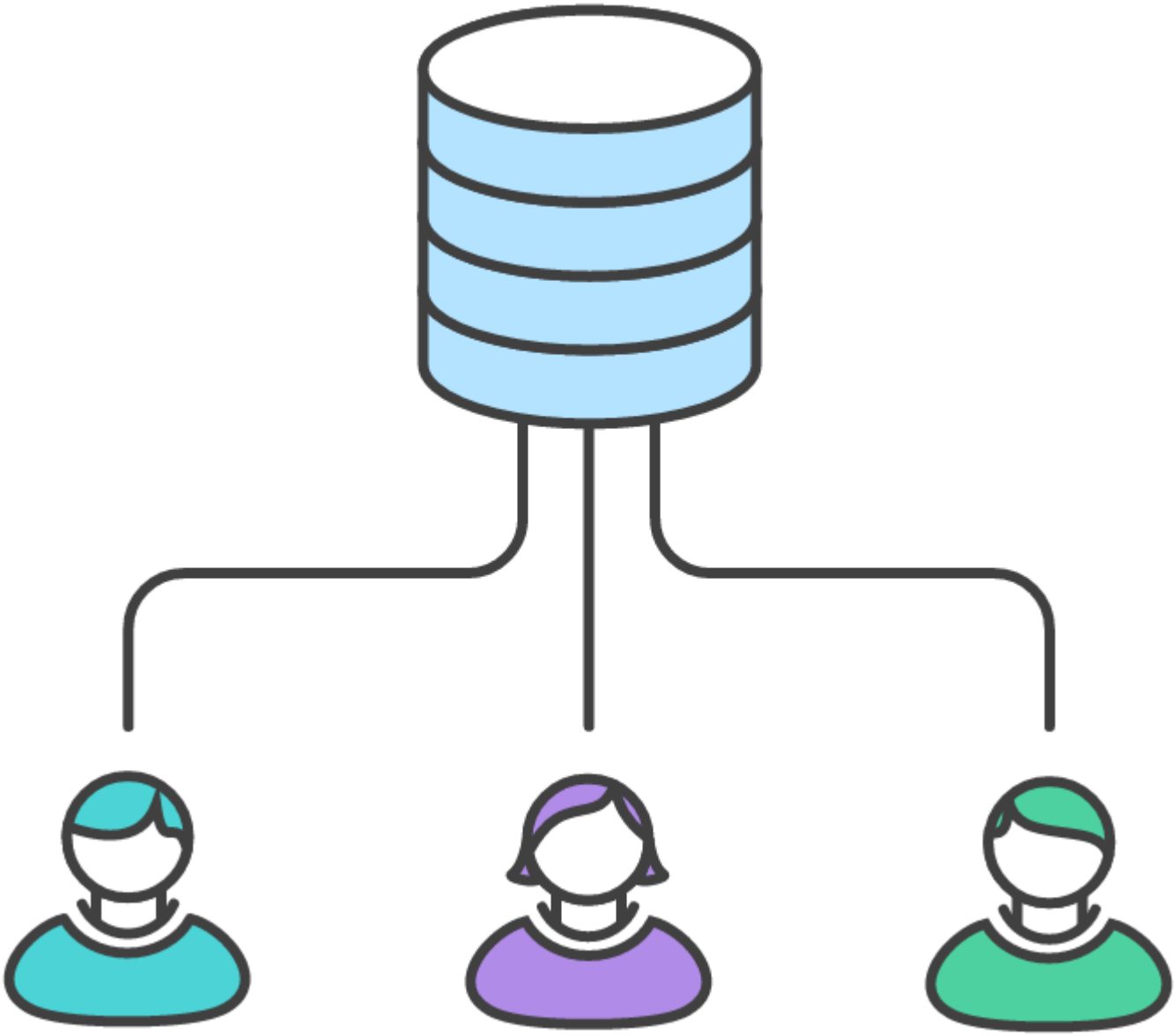


Die ursprüngliche Darstellung dieses Modells:


feature branches
develop
release branches
Time
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
From this point on, "next release" means the release after 1.0

Zweig alle aktiven Entwicklungen. Die Mitwirkenden müssen besonders sicher sein, dass sie die neuesten Änderungen vornehmen, bevor sie sich weiterentwickeln können, denn diese Branche wird sich schnell ändern. Jeder hat Zugriff auf dieses Repo und kann Änderungen direkt an die Hauptniederlassung übergeben.

Visuelle Darstellung dieses Modells:



Dies ist das klassische Versionskontrollparadigma, auf dem ältere Systeme wie Subversion und CVS aufgebaut wurden. Software, die auf diese Weise funktioniert, nennt man zentrale Versionskontrollsysteme (CVCS). Git ist zwar in der Lage, auf diese Weise zu arbeiten, es gibt jedoch erhebliche Nachteile, z. B. dass vor jedem Ziehen eine Zusammenführung erforderlich ist. Es ist durchaus möglich, dass ein Team auf diese Weise arbeitet, aber die ständige Konfliktlösung kann am Ende viel wertvolle Zeit kosten.

Aus diesem Grund hat Linus Torvalds Git nicht als CVCS, sondern als *DVCS* (*Distributed Version Control System) entwickelt* , ähnlich wie Mercurial. Der Vorteil dieser neuen Vorgehensweise liegt in der Flexibilität, die in den anderen Beispielen auf dieser Seite gezeigt wird.

Funktionszweig-Workflow

Der Kerngedanke hinter dem Funktionszweiges Der Workflow ist , dass alle Feature - Entwicklung in einem speziellen Zweig nehmen sollte anstelle des `master` - Zweig. Diese Verkapselung erleichtert es mehreren Entwicklern, an einem bestimmten Feature zu arbeiten, ohne die Hauptcodebase zu stören. Dies bedeutet auch, dass der `master` Zweig niemals fehlerhaften Code enthält, was für kontinuierliche Integrationsumgebungen von großem Vorteil ist.

Durch die Verkapselung der Feature-Entwicklung können auch Pull-Anfragen genutzt werden, mit denen Diskussionen in einem Zweig initiiert werden können. Sie geben anderen Entwicklern die Möglichkeit, sich für ein Feature abzumelden, bevor es in das offizielle Projekt integriert wird. Wenn Sie mitten in einer Funktion stecken bleiben, können Sie eine Pull-Anfrage öffnen, in der Sie nach Vorschlägen Ihrer Kollegen gefragt werden. Der Punkt ist, Pull-Anfragen machen es Ihrem Team unglaublich leicht, die Arbeit des anderen zu kommentieren.

basierend auf Atlassian Tutorials .

GitHub Flow

In vielen Open Source-Projekten beliebt, aber nicht nur.

Der Master- Zweig eines bestimmten Standorts (Github, Gitlab, Bitbucket, lokaler Server) enthält die neueste lieferbare Version. Für jede neue Feature- / Fehlerbehebung / Architekturänderung erstellt jeder Entwickler einen Zweig.

Änderungen finden in diesem Zweig statt und können in einer Pull-Anfrage, Code-Überprüfung usw. besprochen werden. Nach der Annahme werden sie mit dem Hauptzweig zusammengeführt.

Voller Fluss von Scott Chacon:

- Alles in der Master-Niederlassung kann eingesetzt werden
- Um an etwas Neuem zu arbeiten, erstellen Sie einen beschreibend benannten Zweig von master (dh: new-oauth2-scope).
- Übernehmen Sie diese Verzweigung lokal und verschieben Sie Ihre Arbeit regelmäßig an dieselbe Verzweigung auf dem Server
- Wenn Sie Feedback oder Hilfe benötigen oder glauben, dass der Zweig zum Zusammenführen bereit ist, öffnen Sie eine Pull-Anforderung
- Nachdem eine andere Person die Funktion überprüft und abgemeldet hat, können Sie sie mit dem Master zusammenführen
- Sobald es zusammengeführt und auf "Master" verschoben wurde, können und sollten Sie es sofort implementieren

Ursprünglich auf Scott Chacons persönlicher Website präsentiert .

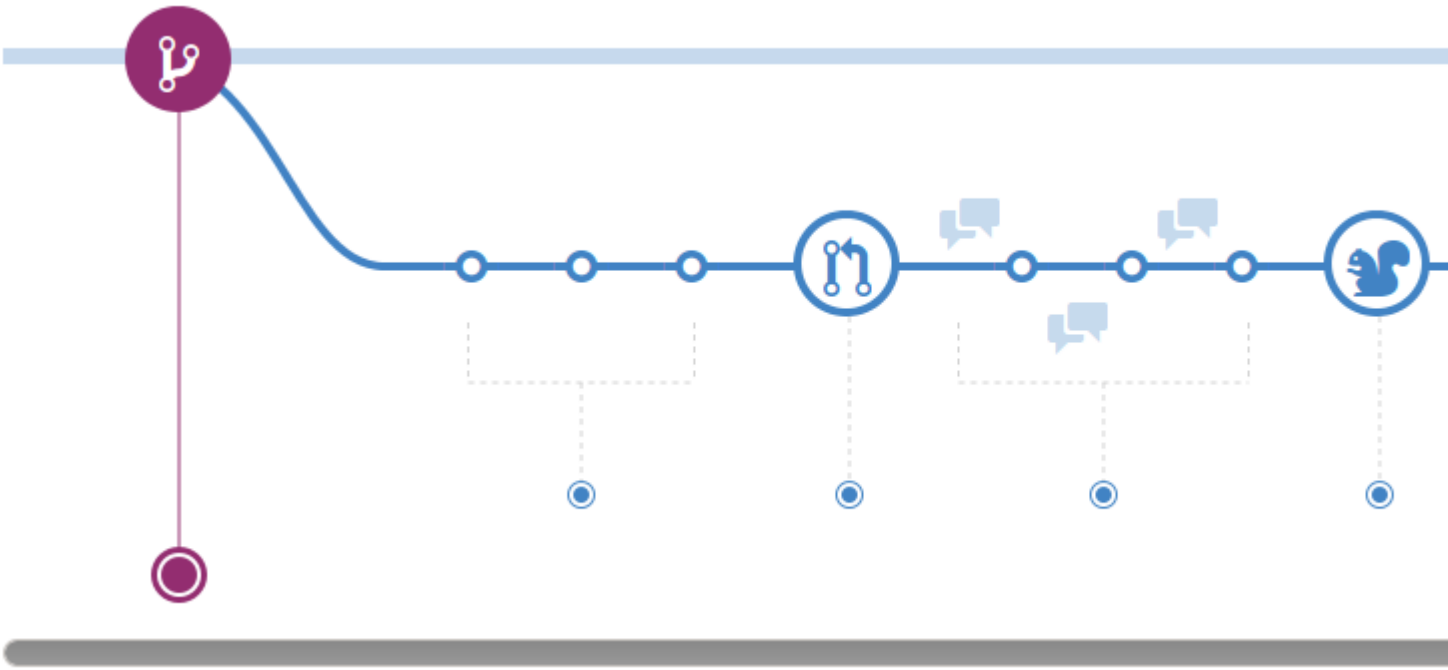


Bild mit freundlicher Genehmigung der GitHub Flow-Referenz

Analysieren von Arten von Workflows online lesen:
https://riptutorial.com/de/git/topic/1276/analysieren-von-arten-von-workflows

Kapitel 6: Ändern Sie den Namen des Git-Repositorys

Einführung

Wenn Sie den Repository-Namen auf der Remote-Seite ändern, z. B. Github oder Bitbucket, wenn Sie den vorhandenen Code pushen, wird der folgende Fehler angezeigt: Schwerwiegender Fehler, Repository nicht gefunden **.

Examples

Lokale Einstellung ändern

Zum Terminal gehen,

```
cd projectFolder
git remote -v (it will show previous git url)
git remote set-url origin https://username@bitbucket.org/username/newName.git
git remote -v (double check, it will show new git url)
git push (do whatever you want.)
```

Ändern Sie den Namen des Git-Repositorys online lesen:
https://riptutorial.com/de/git/topic/9291/andern-sie-den-namen-des-git-repositorys

Kapitel 7: Anzeigen des Commit-Verlaufs grafisch mit Gitk

Examples

Commit-Verlauf für eine Datei anzeigen

```
gitk path/to/myfile
```

Alle Commits zwischen zwei Commits anzeigen

Nehmen wir an, Sie haben zwei Commits `d9e1db9` und `5651067` und möchten sehen, was zwischen ihnen passiert ist. `d9e1db9` ist der älteste Vorfahre und `5651067` der letzte Nachkomme in der `5651067` Kette.

```
gitk --ancestry-path d9e1db9 5651067
```

Commits seit Versions-Tag anzeigen

Wenn Sie das Versions-Tag `v2.3` , können Sie alle Commits seit diesem Tag anzeigen.

```
gitk v2.3..
```

Anzeigen des Commit-Verlaufs grafisch mit Gitk online lesen:
https://riptutorial.com/de/git/topic/3637/anzeigen-des-commit-verlaufs-grafisch-mit-gitk

Kapitel 8: Arbeitsbäume

Syntax

- git worktree add [-f] [--detach] [--checkout] [-b <Neuer Zweig>] <Pfad> [<Zweig>]
- git worktree prune [-n] [-v] [--expire <expire>]
- git worktree list [--porzellan]

Parameter

Parameter	Einzelheiten
-f --force	Fügen Sie standardmäßig keinen neuen Arbeitsbaum hinzu, wenn `<branch>` bereits von einem anderen Arbeitsbaum ausgecheckt ist. Diese Option überschreibt diesen Schutz.
-b `<new-branch>` -B `<new-branch>`	Erstellen Sie mit add einen neuen Zweig mit dem Namen `<new-branch>` , der bei `<branch>` `<new-branch>` beginnt, und checken Sie `<new-branch>` in den neuen Arbeitsbaum aus. Wenn `<branch>` weggelassen wird, wird standardmäßig `HEAD` . Standardmäßig weigert sich `-b` , einen neuen Zweig anzulegen, wenn dieser bereits vorhanden ist. `-B` überschreibt diese Sicherheitsmaßnahme und setzt `<new-branch>` auf `<branch>` .
--ablösen	Mit add trennen Sie `HEAD` im neuen Arbeitsbaum.
- [no-] Kasse	Standardmäßig fügen Sie `--no-checkout` hinzu `<branch>` . Mit `--no-checkout` kann jedoch das Auschecken unterdrückt werden, um Anpassungen vorzunehmen, z.
-n - dry-run	Entfernen Sie nichts mit der Pflaume. einfach berichten, was es entfernen würde.
--Porzellan	Mit liste, Ausgabe in einem einfach zu parse Format für Skripte. Dieses Format bleibt über Git-Versionen und unabhängig von der Benutzerkonfiguration stabil.
-v --verbose	Melden Sie mit Pflaume alle Entfernungen.
--expire `<time>`	Mit Pflaume verfallen nur ungenutzte Arbeitsbäume, die älter als `<time>` .

Bemerkungen

Weitere Informationen finden Sie in der offiziellen Dokumentation: https://git-scm.com/docs/git-worktree .

Examples

Verwenden eines Arbeitsbaums

Sie sind gerade dabei, an einer neuen Funktion zu arbeiten, und Ihr Chef fordert Sie auf, sofort etwas zu beheben. In der Regel möchten Sie mit `git stash` Ihre Änderungen vorübergehend speichern. Zu diesem Zeitpunkt befindet sich Ihr Arbeitsbaum jedoch in einem Zustand der Unordnung (neue, verschobene und entfernte Dateien sowie andere verstreute Teile), und Sie möchten Ihren Fortschritt nicht stören.

Durch das Hinzufügen eines Worktrees erstellen Sie einen temporären verknüpften Arbeitsbaum, um den Notfall zu beheben, entfernen Sie ihn, wenn Sie fertig sind, und setzen Sie dann Ihre vorherige Codierungssitzung fort:

```
$ git worktree add -b emergency-fix ../temp master
$ pushd ../temp
# ... work work work ...
$ git commit -a -m 'emergency fix for boss'
$ popd
$ rm -rf ../temp
$ git worktree prune
```

ANMERKUNG: In diesem Beispiel befindet sich der Fix immer noch im Notfall-Zweig. An dieser Stelle möchten Sie wahrscheinlich `git merge` oder `git format-patch` und anschließend den notfall-notierungszweig entfernen.

Verschieben eines Arbeitsbaums

Derzeit (ab Version 2.11.0) gibt es keine integrierte Funktionalität zum Verschieben eines bereits vorhandenen Arbeitsbaums. Dies ist ein offizieller Fehler (siehe https://git-scm.com/docs/git-worktree#_bugs) .

Um diese Einschränkung zu umgehen, ist es möglich, manuelle Vorgänge direkt in den `.git` Referenzdateien auszuführen.

In diesem Beispiel lebt die Haupt Kopie des Repo bei `/home/user/project-main` und der sekundäre worktree befindet sich unter `/home/user/project-1` und wollen wir es bewegen `/home/user/project-2`

Führen Sie zwischen diesen Schritten keinen git-Befehl aus, da sonst der Garbage Collector ausgelöst wird und die Referenzen auf den sekundären Baum verloren gehen können. Führen Sie diese Schritte ohne Unterbrechung vom Anfang bis zum Ende aus:

1. Ändern Sie die die worktree `.git` Datei an die neue Position innerhalb des Hauptbaum zu zeigen. Die Datei `/home/user/project-1/.git` sollte nun Folgendes enthalten:

   ```
   gitdir: /home/user/project-main/.git/worktrees/project-2
   ```

2. Benennen Sie die worktree innerhalb des `.git` Verzeichnis des Hauptprojekt , das von der

worktree des Verzeichnisses zu bewegen, die dort vorhanden ist :

```
$ mv /home/user/project-main/.git/worktrees/project-1 /home/user/project-
main/.git/worktrees/project-2
```

3. Ändern Sie die Referenz in `/home/user/project-main/.git/worktrees/project-2/gitdir` , dass sie auf den neuen Speicherort `/home/user/project-main/.git/worktrees/project-2/gitdir` . In diesem Beispiel hätte die Datei den folgenden Inhalt:

```
/home/user/project-2/.git
```

4. Verschieben Sie schließlich Ihren Arbeitsbaum an den neuen Speicherort:

```
$ mv /home/user/project-1 /home/user/project-2
```

Wenn Sie alles richtig gemacht haben, sollte das Auflisten der vorhandenen Arbeitsbäume auf den neuen Speicherort verweisen:

```
$ git worktree list
/home/user/project-main  23f78ad [master]
/home/user/project-2     78ac3f3 [branch-name]
```

Es sollte jetzt auch sicher sein, `git worktree prune` .

Arbeitsbäume online lesen: https://riptutorial.com/de/git/topic/3801/arbeitsbaume

Kapitel 9: Archiv

Syntax

- git archive [--format = <fmt>] [--list] [--prefix = <Präfix> /] [<Extra>] [-o <Datei> | --output = <Datei>] [--worktree-attributes] [--remote = <Repo> [--exec = <Git-Upload-Archiv>]] <Baum-ish> [<Pfad> ...]

Parameter

Parameter	Einzelheiten
--format = <fmt>	Format des entstehenden Archivs: `tar` oder `zip` . Wenn diese Option nicht angegeben ist und die Ausgabedatei angegeben ist, wird das Format nach Möglichkeit aus dem Dateinamen abgeleitet. Andernfalls ist der Standardwert `tar` .
-l, --list	Alle verfügbaren Formate anzeigen.
-v, --verbose	Melden Sie den Fortschritt an stderr.
--prefix = <Präfix> /	Stellen Sie jedem Dateinamen im Archiv <Präfix> / vor.
-o <Datei>, --output = <Datei>	Schreiben Sie das Archiv in <file> anstelle von stdout.
--Worktree-Attribute	Suchen Sie in den `.gitattributes` Dateien im Arbeitsbaum nach Attributen.
<extra>	Dies können alle Optionen sein, die das Backend des Archivierers versteht. Bei einem `zip` Backend werden die Dateien mit `-0` gespeichert, ohne sie zu entleeren. Mit `-1` bis `-9` können Sie die Kompressionsgeschwindigkeit und das Verhältnis einstellen.
--remote = <Repo>	Rufen Sie ein tar-Archiv aus einem Remote-Repository `<repo>` und nicht aus dem lokalen Repository ab.
--exec = <git-upload-archive>	Wird mit `--remote` , um den Pfad zum `<git-upload-archive` auf der Remote- `--remote` anzugeben.
<tree-ish>	Der Baum oder das Commit, für das ein Archiv erstellt werden soll.
<Pfad>	Ohne einen optionalen Parameter werden alle Dateien und Verzeichnisse

Parameter	Einzelheiten
	im aktuellen Arbeitsverzeichnis in das Archiv aufgenommen. Wenn ein oder mehrere Pfade angegeben sind, werden nur diese enthalten.

Examples

Erstellen Sie ein Archiv des Git-Repository mit Verzeichnis-Präfix

Es wird empfohlen, beim Erstellen von git-Archiven ein Präfix zu verwenden, damit bei der Extraktion alle Dateien in einem Verzeichnis abgelegt werden. So erstellen Sie ein Archiv von `HEAD` mit einem Verzeichnispräfix:

```
git archive --output=archive-HEAD.zip --prefix=src-directory-name HEAD
```

Beim Extrahieren werden alle Dateien in einem Verzeichnis namens `src-directory-name` im aktuellen Verzeichnis extrahiert.

Erstellen Sie ein Archiv des Git-Repository basierend auf einem bestimmten Zweig, einer Revision, einem Tag oder einem Verzeichnis

Es können auch Archive mit anderen Elementen als `HEAD` , z. B. Verzweigungen, Commits, Tags und Verzeichnisse.

Um ein Archiv von einem lokalen Niederlassung zu erstellen `dev` :

```
git archive --output=archive-dev.zip --prefix=src-directory-name dev
```

So erstellen Sie ein Archiv für einen entfernten Zweig `origin/dev` :

```
git archive --output=archive-dev.zip --prefix=src-directory-name origin/dev
```

So erstellen Sie ein Archiv eines Tags `v.01` :

```
git archive --output=archive-v.01.zip --prefix=src-directory-name v.01
```

Erstellen Sie ein Archiv von Dateien in einem bestimmten Unterverzeichnis (`sub-dir`) der Revision `HEAD` :

```
git archive zip --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir/
```

Erstellen Sie ein Archiv von Git Repository

Mit `git archive` es möglich, komprimierte Archive eines Repositorys zu erstellen, zum Beispiel zur Verteilung von Releases.

Erstellen Sie ein tar-Archiv der aktuellen `HEAD` Version:

```
git archive --format tar HEAD | cat > archive-HEAD.tar
```

Erstellen Sie ein tar-Archiv der aktuellen `HEAD` Version mit gzip-Komprimierung:

```
git archive --format tar HEAD | gzip > archive-HEAD.tar.gz
```

Dies kann auch mit (das die eingebaute tar.gz-Behandlung verwendet) durchgeführt werden:

```
git archive --format tar.gz HEAD > archive-HEAD.tar.gz
```

Erstellen Sie ein ZIP-Archiv der aktuellen `HEAD` Version:

```
git archive --format zip HEAD > archive-HEAD.zip
```

Alternativ können Sie einfach eine Ausgabedatei mit gültiger Erweiterung angeben. Daraus werden Format und Komprimierungstyp abgeleitet:

```
git archive --output=archive-HEAD.tar.gz HEAD
```

Archiv online lesen: https://riptutorial.com/de/git/topic/2815/archiv

Kapitel 10: Aufbau

Syntax

- git config [<Dateioption>] name [Wert] # einer der häufigsten Anwendungsfälle von git config

Parameter

Parameter	Einzelheiten
`--system`	Bearbeitet die systemweite Konfigurationsdatei, die für jeden Benutzer verwendet wird (unter Linux befindet sich diese Datei unter `$(prefix)/etc/gitconfig`).
`--global`	Bearbeitet die globale Konfigurationsdatei, die für jedes Repository verwendet wird, an dem Sie arbeiten (unter Linux befindet sich diese Datei unter `~/.gitconfig`
`--local`	Bearbeitet die respositoryspezifische Konfigurationsdatei, die sich in `.git/config` in Ihrem Repository befindet. Dies ist die Standardeinstellung

Examples

Benutzername und E-Mail-Adresse

Gleich nach der Installation von Git sollten Sie als erstes Ihren Benutzernamen und Ihre E-Mail-Adresse festlegen. Geben Sie aus einer Shell Folgendes ein:

```
git config --global user.name "Mr. Bean"
git config --global user.email mrbean@example.com
```

- `git config` ist der Befehl zum Abrufen oder Einstellen von Optionen
- `--global` bedeutet, dass die für Ihr Benutzerkonto spezifische Konfigurationsdatei bearbeitet wird
- `user.name` und `user.email` sind die Schlüssel für die Konfigurationsvariablen. `user` ist der Abschnitt der Konfigurationsdatei. `name` und `email` sind die Namen der Variablen.
- `"Mr. Bean"` und `mrbean@example.com` sind die Werte, die Sie in den beiden Variablen speichern. Beachten Sie die Anführungszeichen um `"Mr. Bean"` , die erforderlich sind, weil der Wert, den Sie speichern, ein Leerzeichen enthält.

Mehrere Git-Konfigurationen

Sie haben bis zu 5 Quellen für die Git-Konfiguration:

- 6 Dateien:
  - **`%ALLUSERSPROFILE%\Git\Config`** (nur Windows)
  - (System) **`<git>/etc/gitconfig`** , wobei `<git>` der Installationspfad von git ist. (Unter Windows ist es **`<git>\mingw64\etc\gitconfig`**)
  - (System) **`$XDG_CONFIG_HOME/git/config`** (nur Linux / Mac)
  - (global) `~/.gitconfig` (Windows: `%USERPROFILE%\.gitconfig`)
  - (local) `.git/config` (innerhalb eines git repo `$GIT_DIR`)
  - eine **dedizierte Datei** (mit `git config -f`), die zum Beispiel zum Ändern der Konfiguration von Submodulen verwendet wird: `git config -f .gitmodules ...`
- **die Befehlszeile mit `git -c`** : `git -c core.autocrlf=false fetch` würde *jede* andere außer Kraft setzen `core.autocrlf` zu `false` , *nur* für diesen `fetch` Befehl.

Die Reihenfolge ist wichtig: Jede in einer Quelle eingestellte Konfiguration kann von einer darunter aufgelisteten Quelle überschrieben werden.

`git config --system/global/local` ist der Befehl, 3 dieser Quellen aufzulisten, aber nur git config -l würde *alle aufgelösten* Konfigurationen auflisten.
"Gelöst" bedeutet, dass nur der endgültig überschriebene Konfigurationswert aufgeführt wird.

Wenn Sie sehen möchten, welche Config aus welcher Datei stammt, geben Sie seit git 2.8 Folgendes ein:

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

Festlegen, welcher Editor verwendet werden soll

Es gibt verschiedene Möglichkeiten, den Editor festzulegen, der zum Festschreiben, Umbasieren usw. verwendet werden soll.

- Ändern Sie die Konfigurationseinstellung `core.editor` .

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

- `GIT_EDITOR` Umgebungsvariable `GIT_EDITOR` .

  Für einen Befehl:

  ```
  $ GIT_EDITOR=nano git commit
  ```

  Oder für alle Befehle, die in einem Terminal laufen. **Hinweis:** Dies gilt nur bis zum Schließen des Terminals.

  ```
  $ export GIT_EDITOR=nano
  ```

- Um den Editor für *alle* Terminalprogramme und nicht nur für Git zu ändern, legen Sie die Umgebungsvariable `VISUAL` oder `EDITOR` . (Siehe `VISUAL` **VS** `EDITOR` .)

```
$ export EDITOR=nano
```

Hinweis: Wie oben gilt dies nur für das aktuelle Terminal. Ihre Shell verfügt normalerweise über eine Konfigurationsdatei, mit der Sie diese dauerhaft festlegen können. (`~/.bashrc` zum Beispiel bei `bash` Ihre `~/.bashrc` oder `~/.bash_profile` .)

Einige Texteditoren (hauptsächlich GUI-Editoren) führen jeweils nur eine Instanz aus und werden im Allgemeinen beendet, wenn bereits eine Instanz von ihnen geöffnet ist. Wenn dies für Ihren Texteditor der Fall ist, druckt Git die Nachricht `Aborting commit due to empty commit message.` ohne dass Sie die Commit-Nachricht zuerst bearbeiten dürfen. Wenn dies bei Ihnen der `--wait` , konsultieren Sie die Dokumentation Ihres Texteditors, um zu sehen, ob er eine `--wait` (oder ähnliches) hat, durch die das Dokument `--wait` wird, bis das Dokument geschlossen wird.

Zeilenenden konfigurieren

Beschreibung

Wenn Sie mit einem Team arbeiten, das im gesamten Projekt verschiedene Betriebssysteme (Betriebssysteme) verwendet, kann es manchmal zu Problemen beim Umgang mit Leitungsenden kommen.

Microsoft Windows

Wenn Sie mit einem Microsoft Windows-Betriebssystem arbeiten, haben die Zeilenenden normalerweise die Form Carriage Return + Line Feed (CR + LF). Das Öffnen einer Datei, die mit einem Unix-Computer wie Linux oder OSX bearbeitet wurde, kann zu Problemen führen, was den Anschein erweckt, dass Text keine Zeilenenden hat. Dies liegt daran, dass Unix-Systeme nur unterschiedliche Zeilenenden von Formularzeilen (LF) verwenden.

Um dies zu beheben, können Sie die folgenden Anweisungen ausführen

```
git config --global core.autocrlf=true
```

Beim **Auschecken** wird diese Anweisung Zeilenende sicherzustellen , in Übereinstimmung mit Microsoft Windows - Betriebssystem konfiguriert sind (LF -> CR + LF)

Unix-basiert (Linux / OSX)

In ähnlicher Weise können Probleme auftreten, wenn der Benutzer unter Unix-basierten Betriebssystemen versucht, Dateien zu lesen, die unter Microsoft Windows-Betriebssystem bearbeitet wurden. Um zu verhindern, dass unerwartete Probleme auftreten

```
git config --global core.autocrlf=input
```

Beim **Festschreiben** werden dadurch die Zeilenenden von CR + LF -> + LF geändert

Konfiguration nur für einen Befehl

Sie können `-c <name>=<value>` , um eine Konfiguration nur für einen Befehl hinzuzufügen.

Festschreiben als anderer Benutzer, ohne Ihre Einstellungen in .gitconfig ändern zu müssen:

```
git -c user.email = mail@example commit -m "some message"
```

Hinweis: In diesem Beispiel müssen Sie nicht nur `user.name` und `user.email` genau `user.name` `user.email` git wird die fehlenden Informationen aus den vorherigen Commits ergänzen.

Richten Sie einen Proxy ein

Wenn Sie hinter einem Proxy stehen, müssen Sie git davon erzählen:

```
git config --global http.proxy http://my.proxy.com:portnumber
```

Wenn Sie nicht mehr hinter einem Proxy stehen:

```
git config --global --unset http.proxy
```

Auto korrekte Tippfehler

```
git config --global help.autocorrect 17
```

Dies ermöglicht eine Autokorrektur in git und vergibt Ihnen kleinere Fehler (z. B. `git stats` statt `git status`). Der Parameter, den Sie für `help.autocorrect` , `help.autocorrect` fest, wie lange das System in Zehntelsekunden warten soll, bevor der autokorrekte Befehl automatisch `help.autocorrect` wird. Im obigen Befehl 17 bedeutet, dass git 1,7 Sekunden warten muss, bevor der autokorrekte Befehl angewendet wird.

Größere Fehler werden jedoch als fehlende Befehle betrachtet. `git testingit` also etwas wie `git testingit` `testingit is not a git command.`

Auflisten und Bearbeiten der aktuellen Konfiguration

Mit der Git-Konfiguration können Sie die Funktionsweise von Git anpassen. Es wird häufig verwendet, um Ihren Namen und Ihre E-Mail-Adresse oder Ihren bevorzugten Editor festzulegen oder festzulegen, wie die Zusammenführung erfolgen soll.

Um die aktuelle Konfiguration anzuzeigen.

```
$ git config --list
...
core.editor=vim
credential.helper=osxkeychain
...
```

So bearbeiten Sie die Konfiguration:

```
$ git config <key> <value>
$ git config core.ignorecase true
```

Wenn Sie beabsichtigen, dass die Änderung für alle Ihre Repositorys wahr ist, verwenden Sie `--global`

```
$ git config --global user.name "Your Name"
$ git config --global user.email "Your Email"
$ git config --global core.editor vi
```

Sie können erneut auflisten, um Ihre Änderungen zu sehen.

Mehrere Benutzernamen und E-Mail-Adresse

Seit Git 2.13 können mehrere Benutzernamen und E-Mail-Adressen mithilfe eines Ordnerfilters konfiguriert werden.

Beispiel für Windows:

.gitconfig

Edit: `git config --global -e`

Hinzufügen:

```
[includeIf "gitdir:D:/work"]
  path = .gitconfig-work.config

[includeIf "gitdir:D:/opensource/"]
  path = .gitconfig-opensource.config
```

Anmerkungen

- Die Reihenfolge ist abhängig, der letzte, der mit "gewinnt" übereinstimmt.
- das `/` am Ende wird benötigt - zB `"gitdir:D:/work"` funktioniert nicht.
- das `gitdir:` ist erforderlich.

.gitconfig-work.config

Datei im selben Verzeichnis wie *.gitconfig*

```
[user]
  name = Money
  email = work@somewhere.com
```

.gitconfig-opensource.config

Datei im selben Verzeichnis wie *.gitconfig*

```
[user]
  name = Nice
  email = cool@opensource.stuff
```

Beispiel für Linux

```
[includeIf "gitdir:~/work/"]
  path = .gitconfig-work
[includeIf "gitdir:~/opensource/"]
  path = .gitconfig-opensource
```

Der Inhalt der Datei und die Hinweise unter Abschnitt Windows.

Aufbau online lesen: https://riptutorial.com/de/git/topic/397/aufbau

Kapitel 11: Bündel

Bemerkungen

Der Schlüssel zu dieser Arbeit besteht darin, ein Paket zu klonen, das am Anfang der Repogeschichte beginnt:

```
git bundle create initial.bundle master
git tag -f some_previous_tag master  # so the whole repo does not have to go each time
```

dieses anfängliche Bündel auf dem entfernten Rechner erhalten; und

```
git clone -b master initial.bundle remote_repo_name
```

Examples

Erstellen eines Git-Pakets auf dem lokalen Computer und dessen Verwendung auf einem anderen Computer

In manchen Fällen möchten Sie möglicherweise die Versionen eines Git-Repositorys auf Computern verwalten, die keine Netzwerkverbindung haben. Mit Bundles können Sie git-Objekte und Referenzen in einem Repository auf einer Maschine packen und diese in ein Repository auf einem anderen importieren.

```
git tag 2016_07_24
git bundle create changes_between_tags.bundle [some_previous_tag]..2016_07_24
```

Übertragen Sie die Datei **changes_between_tags.bundle** auf die entfernte Maschine. zB über Daumenantrieb. Sobald Sie es dort haben:

```
git bundle verify changes_between_tags.bundle  # make sure bundle arrived intact
git checkout [some branch]       # in the repo on the remote machine
git bundle list-heads changes_between_tags.bundle # list the references in the bundle
git pull changes_between_tags.bundle [reference from the bundle, e.g. last field from the
previous output]
```

Das Gegenteil ist auch möglich. Sobald Sie Änderungen am Remote-Repository vorgenommen haben, können Sie die Deltas bündeln. Legen Sie die Änderungen beispielsweise auf einem USB-Stick ab und führen Sie sie wieder in das lokale Repository ein, sodass die beiden synchron bleiben können, ohne dass ein direkter Zugriff auf `git` , `ssh` , `rsync` oder `http` zwischen den Maschinen erforderlich ist.

Bündel online lesen: https://riptutorial.com/de/git/topic/3612/bundel

Kapitel 12: Dateien und Ordner ignorieren

Einführung

In diesem Thema wird veranschaulicht, wie Sie das Hinzufügen unerwünschter Dateien (oder Dateiänderungen) in einem Git-Repo vermeiden. Es gibt verschiedene Möglichkeiten (global oder lokal `.gitignore` , `.gitignore .git/exclude` , `git update-index --assume-unchanged` und `git update-index --skip-tree`). `git update-index --skip-tree` jedoch, dass Git den *Inhalt verwaltet* . ignorierend ignoriert tatsächlich einen *Ordnerinhalt* (dh Dateien). Ein leerer Ordner wird standardmäßig ignoriert, da er sowieso nicht hinzugefügt werden kann.

Examples

Ignorieren von Dateien und Verzeichnissen mit einer .gitignore-Datei

Sie können Git dazu veranlassen, bestimmte Dateien und Verzeichnisse zu ignorieren, d. `.gitignore` Sie von der Verfolgung durch Git auszuschließen, indem Sie eine oder mehrere `.gitignore` Dateien in Ihrem Repository erstellen.

In Softwareprojekten enthält `.gitignore` normalerweise eine Liste von Dateien und / oder Verzeichnissen, die während des Erstellungsprozesses oder zur Laufzeit generiert werden. Einträge in der `.gitignore` Datei können Namen oder Pfade enthalten, die auf `.gitignore` zeigen:

1. temporäre Ressourcen, z. B. Caches, Protokolldateien, kompilierter Code usw.
2. lokale Konfigurationsdateien, die nicht mit anderen Entwicklern gemeinsam genutzt werden dürfen
3. Dateien, die geheime Informationen enthalten, wie Anmeldekennwörter, Schlüssel und Anmeldeinformationen

Wenn sie im obersten Verzeichnis erstellt werden, gelten die Regeln für alle Dateien und Unterverzeichnisse des gesamten Repositorys rekursiv. Bei der Erstellung in einem Unterverzeichnis gelten die Regeln für dieses bestimmte Verzeichnis und dessen Unterverzeichnisse.

Wenn eine Datei oder ein Verzeichnis ignoriert wird, wird Folgendes nicht angezeigt:

1. von Git verfolgt
2. Wird von Befehlen wie `git status` oder `git diff` gemeldet
3. mit Befehlen wie `git add -A` inszeniert

In dem ungewöhnlichen Fall, dass Sie verfolgte Dateien ignorieren müssen, ist besondere Vorsicht geboten. Siehe: Ignorieren Sie Dateien, die bereits an ein Git-Repository übergeben wurden .

Beispiele

Hier einige allgemeine Beispiele für Regeln in einer `.gitignore` Datei, die auf Glob- `.gitignore` basieren:

```
# Lines starting with `#` are comments.

# Ignore files called 'file.ext'
file.ext

# Comments can't be on the same line as rules!
# The following line ignores files called 'file.ext # not a comment'
file.ext # not a comment

# Ignoring files with full path.
# This matches files in the root directory and subdirectories too.
# i.e. otherfile.ext will be ignored anywhere on the tree.
dir/otherdir/file.ext
otherfile.ext

# Ignoring directories
# Both the directory itself and its contents will be ignored.
bin/
gen/

# Glob pattern can also be used here to ignore paths with certain characters.
# For example, the below rule will match both build/ and Build/
[bB]uild/

# Without the trailing slash, the rule will match a file and/or
# a directory, so the following would ignore both a file named `gen`
# and a directory named `gen`, as well as any contents of that directory
bin
gen

# Ignoring files by extension
# All files with these extensions will be ignored in
# this directory and all its sub-directories.
*.apk
*.class

# It's possible to combine both forms to ignore files with certain
# extensions in certain directories. The following rules would be
# redundant with generic rules defined above.
java/*.apk
gen/*.class

# To ignore files only at the top level directory, but not in its
# subdirectories, prefix the rule with a `/`
/*.apk
/*.class

# To ignore any directories named DirectoryA
# in any depth use ** before DirectoryA
# Do not forget the last /,
# Otherwise it will ignore all files named DirectoryA, rather than directories
**/DirectoryA/
# This would ignore
```

```
# DirectoryA/
# DirectoryB/DirectoryA/
# DirectoryC/DirectoryB/DirectoryA/
# It would not ignore a file named DirectoryA, at any level

# To ignore any directory named DirectoryB within a
# directory named DirectoryA with any number of
# directories in between, use ** between the directories
DirectoryA/**/DirectoryB/
# This would ignore
# DirectoryA/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryB/
# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/

# To ignore a set of files, wildcards can be used, as can be seen above.
# A sole '*' will ignore everything in your folder, including your .gitignore file.
# To exclude specific files when using wildcards, negate them.
# So they are excluded from the ignore list:
!.gitignore

# Use the backslash as escape character to ignore files with a hash (#)
# (supported since 1.6.2.1)
\#*#
```

Die meisten `.gitignore` Dateien sind standardmäßig in verschiedenen Sprachen `.gitignore` Um zu beginnen, werden hier `.gitignore` Beispiel- `.gitignore` Dateien nach Sprache aufgelistet, aus der `.gitignore` oder in Ihr Projekt kopiert / `.gitignore` werden kann . Alternativ können Sie für ein neues Projekt eine Starter-Datei mit einem Online-Tool automatisch generieren.

Andere Formen von .gitignore

`.gitignore` Dateien sollen als Bestandteil des Repositorys `.gitignore` werden. Wenn Sie bestimmte Dateien ignorieren möchten, ohne die Ignorierregeln festzulegen, haben Sie folgende Möglichkeiten:

- Bearbeiten Sie die Datei `.gitignore .git/info/exclude` (verwenden Sie dieselbe Syntax wie `.gitignore`). Die Regeln sind global im Repository.
- Richten Sie eine globale Gitignore-Datei ein , die die Ignorierregeln auf alle Ihre lokalen Repositorys anwendet:

Außerdem können Sie lokale Änderungen an verfolgten Dateien ignorieren, ohne die globale Git-Konfiguration zu ändern:

- `git update-index --skip-worktree [<file>...]` : für geringfügige lokale Änderungen
- `git update-index --assume-unchanged [<file>...]` : für produktionsbereite, nicht veränderte Dateien im Upstream

Weitere Informationen zu den Unterschieden zwischen den letzteren Flags und der Dokumentation zum `git update-index` Weitere Optionen.

Ignorierte Dateien bereinigen

Sie können `git clean -X` um ignorierte Dateien zu bereinigen:

```
git clean -Xn #display a list of ignored files
git clean -Xf #remove the previously displayed files
```

Hinweis: `-X` (Caps) bereinigt *nur* ignorierte Dateien. Verwenden Sie `-x` (keine Begrenzungen), um auch nicht protokollierte Dateien zu entfernen.

Weitere Informationen finden Sie in der Dokumentation zu `git clean` .

Weitere Informationen finden Sie im Git-Handbuch .

Ausnahmen in einer .gitignore-Datei

Wenn Sie Dateien mit einem Muster ignorieren, aber Ausnahmen haben, setzen Sie der Ausnahme ein Ausrufezeichen (!) Voran. Zum Beispiel:

```
*.txt
!important.txt
```

Das obige Beispiel weist Git an, alle Dateien mit der Erweiterung `.txt` außer Dateien mit der Bezeichnung `important.txt` zu ignorieren.

Wenn sich die Datei in einem ignorierten Ordner befindet, können Sie sie **NICHT** so einfach erneut einschließen:

```
folder/
!folder/*.txt
```

In diesem Beispiel würden alle TXT-Dateien im Ordner ignoriert.

Der richtige Weg ist, den Ordner selbst in eine separate Zeile einzufügen, dann alle Dateien im `folder` mit `*` ignorieren und schließlich den `folder *.txt` im `folder` wie folgt neu aufzunehmen:

```
!folder/
folder/*
!folder/*.txt
```

Hinweis : **Fügen** Sie bei Dateinamen, die mit einem Ausrufezeichen beginnen, zwei Ausrufezeichen hinzu oder setzen Sie ein Escapezeichen mit dem Zeichen `\` :

```
!!includethis
\!excludethis
```

Eine globale .gitignore-Datei

Damit Git bestimmte Dateien in allen Repositorys ignoriert, können Sie mit dem folgenden Befehl in Ihrem Terminal oder der Eingabeaufforderung einen globalen .gitignore erstellen :

```
$ git config --global core.excludesfile <Path_To_Global_gitignore_file>
```

Git wird dies nun zusätzlich zu der eigenen .gitignore- Datei jedes Repositorys verwenden. Regeln dafür sind:

- Wenn die lokale `.gitignore` Datei explizit eine Datei enthält, während der globale `.gitignore` ignoriert, hat der lokale `.gitignore` Priorität (die Datei wird eingeschlossen).
- Wenn das Repository auf mehreren Rechnern `.gigignore` muss der globale `.gigignore` auf allen Rechnern geladen oder zumindest eingeschlossen sein, da die ignorierten Dateien in den Repo `.gitignore` während der PC mit dem globalen `.gitignore` ihn nicht aktualisiert . Aus diesem Grund ist ein Repo-spezifischer `.gitignore` eine bessere Idee als eine globale, wenn das Projekt von einem Team bearbeitet wird

Diese Datei ist ein guter Ort, um plattform-, maschinen- oder benutzerspezifische Ignorierungen `.DS_Store` , z. B. OSX `.DS_Store` , Windows `Thumbs.db` oder Vim `*.ext~` und `*.ext.swp` ignoriert, wenn Sie diese nicht im Repository behalten möchten . Ein unter OS X arbeitendes Teammitglied kann also alle `.DS_STORE` und `_MACOSX` (was eigentlich nutzlos ist), während ein anderes Teammitglied unter Windows alle `thumbs.bd` ignorieren `thumbs.bd`

Ignorieren Sie Dateien, die bereits an ein Git-Repository übergeben wurden

Wenn Sie bereits eine Datei zu Ihrem Git-Repository hinzugefügt haben und nun die **Verfolgung beenden** möchten (damit sie in zukünftigen Commits nicht vorhanden ist), können Sie sie aus dem Index entfernen:

```
git rm --cached <file>
```

Dies entfernt die Datei aus dem Repository und verhindert, dass weitere Änderungen von Git verfolgt werden. Die Option `--cached` sicher, dass die Datei nicht physisch gelöscht wird.

Beachten Sie, dass der zuvor hinzugefügte Inhalt der Datei weiterhin über den Git-Verlauf sichtbar ist.

Beachten Sie, dass, wenn jemand anderes aus dem Repository zieht, nachdem Sie die Datei aus dem Index entfernt haben, **deren Kopie physisch gelöscht wird** .

Sie können Git so tun, als ob die Version des Arbeitsverzeichnisses der Datei auf dem neuesten Stand ist, und stattdessen die Indexversion lesen (um Änderungen daran zu ignorieren) mit dem Bit " Skip Worktree ":

```
git update-index --skip-worktree <file>
```

Das Schreiben wird durch dieses Bit nicht beeinflusst, die Inhaltssicherheit hat nach wie vor höchste Priorität. Sie werden niemals Ihre wertvollen, ignorierten Änderungen verlieren. Andererseits steht dieses Bit im Konflikt mit dem Einlagern: Um dieses Bit zu entfernen, verwenden Sie

```
git update-index --no-skip-worktree <file>
```

Es wird manchmal ***fälschlicherweise*** empfohlen, Git anzulügen und davon auszugehen, dass die Datei unverändert bleibt, ohne sie zu untersuchen. Auf den ersten Blick sieht es so aus, als würden weitere Änderungen an der Datei ignoriert, ohne sie aus dem Index zu entfernen:

```
git update-index --assume-unchanged <file>
```

Dadurch wird git gezwungen, alle in der Datei vorgenommenen Änderungen zu ignorieren (bedenken Sie, dass **Ihre ignorierten Änderungen verloren gehen** , wenn Sie Änderungen an dieser Datei vornehmen oder sie speichern

Wenn Sie möchten, dass git sich erneut um diese Datei kümmert, führen Sie den folgenden Befehl aus:

```
git update-index --no-assume-unchanged <file>
```

Prüfen, ob eine Datei ignoriert wird

Der Befehl `git check-ignore` meldet Dateien, die von Git ignoriert werden.

Sie können Dateinamen über die Befehlszeile übergeben, und `git check-ignore` listet die ignorierten Dateinamen auf. Zum Beispiel:

```
$ cat .gitignore
*.o
$ git check-ignore example.o Readme.md
example.o
```

Hier sind nur * .o-Dateien in .gitignore definiert, sodass Readme.md nicht in der Ausgabe von `git check-ignore` .

Wenn Sie sehen möchten, in welcher Zeile .gitignore eine Datei ignoriert, fügen Sie dem Befehl git check-ignore -v hinzu:

```
$ git check-ignore -v example.o Readme.md
.gitignore:1:*.o        example.o
```

Ab Git 1.7.6 können Sie auch `git status --ignored` verwenden, um ignorierte Dateien `git status --ignored` . Weitere Informationen hierzu finden Sie in der offiziellen Dokumentation oder in Dateien suchen, die von .gitignore ignoriert werden .

Ignorieren von Dateien in Unterordnern (mehrere Gititore-Dateien)

Angenommen, Sie haben eine Repository-Struktur wie folgt:

```
examples/
    output.log
src/
    <files not shown>
    output.log
README.md
```

`output.log` im Beispielverzeichnis ist gültig und erforderlich, damit das Projekt Verständnis `output.log` , während die `output.log` unterhalb von `src/` während des Debuggens erstellt wird und nicht im Verlauf oder Teil des Repositorys enthalten sein sollte.

Es gibt zwei Möglichkeiten, diese Datei zu ignorieren. Sie können einen absoluten Pfad in die `.gitignore` Datei im Stammverzeichnis des Arbeitsverzeichnisses `.gitignore` :

```
# /.gitignore
src/output.log
```

Alternativ können Sie eine `.gitignore` Datei im Verzeichnis `src/` erstellen und die Datei ignorieren, die relativ zu diesem `.gitignore` :

```
# /src/.gitignore
output.log
```

Eine Datei in einem beliebigen Verzeichnis ignorieren

Um eine Datei `foo.txt` in einem **beliebigen** Verzeichnis zu ignorieren, schreiben Sie einfach ihren Namen:

```
foo.txt # matches all files 'foo.txt' in any directory
```

Wenn Sie die Datei nur in einem Teil der Baumstruktur ignorieren möchten, können Sie die Unterverzeichnisse eines bestimmten Verzeichnisses mit `**` pattern angeben:

```
bar/**/foo.txt # matches all files 'foo.txt' in 'bar' and all subdirectories
```

Sie können auch eine `.gitignore` Datei im Verzeichnis `bar/` erstellen. Entspricht dem vorherigen Beispiel das Erstellen einer Datei `bar/.gitignore` mit folgendem Inhalt:

```
foo.txt # matches all files 'foo.txt' in any directory under bar/
```

Dateien lokal ignorieren, ohne Regeln zu ignorieren

`.gitignore` ignoriert Dateien lokal, es soll jedoch an das Repository übergeben und mit anderen Mitwirkenden und Benutzern geteilt werden. Sie können einen globalen `.gitignore` , aber dann

würden alle Ihre Repositorys diese Einstellungen gemeinsam nutzen.

Wenn Sie bestimmte Dateien in einem Repository lokal ignorieren und die Datei nicht zu einem Repository machen möchten, bearbeiten Sie `.git/info/exclude` in Ihrem Repository.

Zum Beispiel:

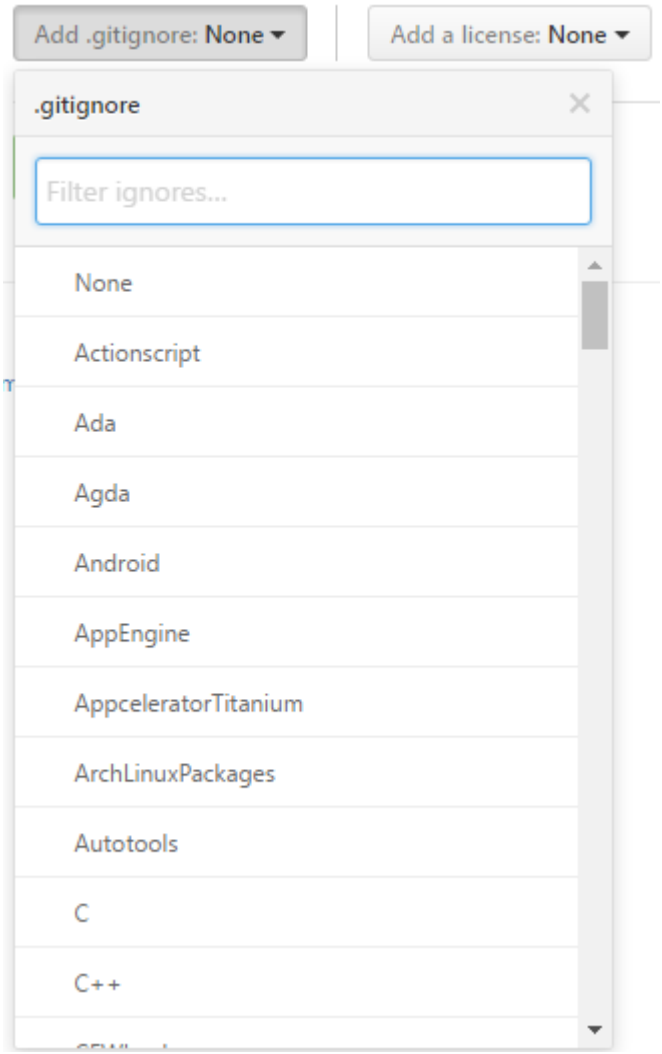
```
# these files are only ignored on this repo
# these rules are not shared with anyone
# as they are personal
gtk_tests.py
gui/gtk/tests/*
localhost
pushReports.py
server/
```

Vorgefüllte .gitignore-Vorlagen

Wenn Sie nicht sicher sind, welche Regeln in Ihrer `.gitignore` Datei aufgeführt werden sollen, oder Sie nur allgemein akzeptierte Ausnahmen zu Ihrem Projekt hinzufügen möchten, können Sie eine `.gitignore` Datei auswählen oder generieren:

- https://www.gitignore.io/
- https://github.com/github/gitignore

Viele Hosting-Services wie GitHub und BitBucket bieten die Möglichkeit, `.gitignore` Dateien basierend auf den verwendeten Programmiersprachen und IDEs zu generieren:



Nachfolgende Änderungen an einer Datei ignorieren (ohne sie zu entfernen)

Manchmal möchten Sie eine Datei in Git halten, spätere Änderungen jedoch ignorieren.

Weisen Sie Git an, Änderungen an einer Datei oder einem Verzeichnis mithilfe von `update-index` zu ignorieren:

```
git update-index --assume-unchanged my-file.txt
```

Der obige Befehl weist Git an, davon auszugehen, dass `my-file.txt` nicht geändert wurde, und Änderungen nicht zu überprüfen oder zu melden. Die Datei ist noch im Repository vorhanden.

Dies kann nützlich sein, um Standardwerte bereitzustellen und Überschreibungen in der Umgebung zuzulassen, z.

```
# create a file with some values in
cat <<EOF
MYSQL_USER=app
MYSQL_PASSWORD=FIXME_SECRET_PASSWORD
EOF > .env

# commit to Git
```

```
git add .env
git commit -m "Adding .env template"

# ignore future changes to .env
git update-index --assume-unchanged .env

# update your password
vi .env

# no changes!
git status
```

Nur einen Teil einer Datei ignorieren [Stub]

Manchmal möchten Sie möglicherweise lokale Änderungen in einer Datei haben, die Sie nicht festlegen oder veröffentlichen möchten. Idealerweise sollten die lokalen Einstellungen in einer separaten Datei konzentriert sein, die in `.gitignore` platziert werden kann. `.gitignore` kann es jedoch kurzfristig hilfreich sein, etwas Lokales in einer eingecheckten Datei zu haben.

Sie können Git dazu bringen, diese Zeilen mit Hilfe des Clean-Filters "unsee" zu machen. Sie werden nicht einmal in Differenzen auftauchen.

Angenommen, hier ist ein Ausschnitt aus der Datei `file1.c` :

```
struct settings s;
s.host = "localhost";
s.port = 5653;
s.auth = 1;
s.port = 15653; // NOCOMMIT
s.debug = 1; // NOCOMMIT
s.auth = 0; // NOCOMMIT
```

Sie möchten `NOCOMMIT` Zeilen nicht überall veröffentlichen.

Erstellen Sie einen "nocommit" -Filter, indem Sie diesen zu der Git-Konfigurationsdatei wie `.git/config` hinzufügen:

```
[filter "nocommit"]
    clean=grep -v NOCOMMIT
```

Fügen Sie dies hinzu (oder erstellen Sie es) zu `.git/info/attributes` oder `.gitmodules` :

```
file1.c filter=nocommit
```

Und Ihre NOCOMMIT-Zeilen sind vor Git verborgen.

Vorsichtsmaßnahmen:

- Die Verwendung eines sauberen Filters verlangsamt die Verarbeitung von Dateien, insbesondere unter Windows.
- Die ignorierte Zeile verschwindet möglicherweise aus der Datei, wenn Git sie aktualisiert.

Dem kann mit einem Wischfilter entgegengewirkt werden, er ist jedoch komplizierter.
- Nicht unter Windows getestet

Änderungen in verfolgten Dateien ignorieren. [Stub]

.gitignore und .git `.git/info/exclude` funktionieren nur für nicht protokollierte Dateien.

Verwenden Sie den Befehl update-index , um das Ignorierungsflag für eine verfolgte Datei festzulegen :

```
git update-index --skip-worktree myfile.c
```

Um dies rückgängig zu machen, verwenden Sie:

```
git update-index --no-skip-worktree myfile.c
```

Sie können diese Schnipsel zu Ihrem globalen hinzufügen git config - bequemer haben `git hide` , `git unhide` und `git hidden` Befehle ein :

```
[alias]
    hide   = update-index --skip-worktree
    unhide = update-index --no-skip-worktree
    hidden  = "!git ls-files -v | grep ^[hsS] | cut -c 3-"
```

Sie können auch die Option --assume-unverändert mit der Update-Index-Funktion verwenden

```
git update-index --assume-unchanged <file>
```

Wenn Sie diese Datei erneut auf die Änderungen überprüfen möchten, verwenden Sie

```
git update-index --no-assume-unchanged <file>
```

Wenn das Flag --assume-unverändert angegeben ist, verspricht der Benutzer, die Datei nicht zu ändern, und lässt Git von der Annahme ausgehen, dass die Arbeitsbaumdatei mit der im Index aufgezeichneten Datei übereinstimmt. Git schlägt fehl, falls diese Datei im Index geändert werden muss zB beim Zusammenführen in einem Commit; Wenn also die angenommene Datei nicht geändert wird, müssen Sie die Situation manuell behandeln. In diesem Fall liegt der Fokus auf der Leistung.

Das Flag --skip-worktree ist nützlich, wenn Sie git anweisen, eine bestimmte Datei nicht zu berühren, da die Datei lokal geändert wird und Sie nicht versehentlich die Änderungen festschreiben möchten (dh Konfigurations- / Eigenschaftendatei, die für eine bestimmte Datei konfiguriert ist Umgebung). Skip-Worktree hat Vorrang vor Annahmen, wenn beide gesetzt sind.

Löschen Sie bereits festgeschriebene Dateien, die jedoch in .gitignore enthalten sind

Manchmal kommt es vor, dass eine Datei von git verfolgt wurde, zu einem späteren Zeitpunkt jedoch zu .gitignore hinzugefügt wurde, um die Verfolgung zu beenden. Es ist ein sehr häufiges Szenario, dass Sie vergessen, solche Dateien vor dem Hinzufügen zu .gitignore zu bereinigen. In diesem Fall hängt die alte Datei immer noch im Repository.

Um dieses Problem zu beheben, könnte man im Repository "im Trockenlauf" entfernen und anschließend alle Dateien wieder hinzufügen. Solange Sie keine ausstehenden Änderungen haben und der Parameter `--cached` wird, ist die `--cached` dieses Befehls ziemlich sicher:

```
# Remove everything from the index (the files will stay in the file system)
$ git rm -r --cached .

# Re-add everything (they'll be added in the current state, changes included)
$ git add .

# Commit, if anything changed. You should see only deletions
$ git commit -m 'Remove all files that are in the .gitignore'

# Update the remote
$ git push origin master
```

Erstellen Sie einen leeren Ordner

Es ist nicht möglich, einen leeren Ordner in Git hinzuzufügen und festzuschreiben, da Git *Dateien* verwaltet und ihr Verzeichnis an sie anfügt. Dies verringert die Festschreibung und erhöht die Geschwindigkeit. Um dies zu umgehen, gibt es zwei Methoden:

Methode eins: `.gitkeep`

Um dies zu `.gitkeep` , müssen Sie den Ordner für Git mit einer `.gitkeep` Datei registrieren. Erstellen Sie dazu einfach das erforderliche Verzeichnis und fügen `.gitkeep` dem Ordner eine `.gitkeep` Datei hinzu. Diese Datei ist leer und dient nur der Registrierung des Ordners. Öffnen Sie dazu in Windows (das unbequeme Dateinamenskonventionen hat) git bash im Verzeichnis und führen Sie den Befehl aus:

```
$ touch .gitkeep
```

Dieser Befehl erstellt eine leere `.gitkeep` Datei im aktuellen Verzeichnis

Methode zwei: `dummy.txt`

Ein weiterer Hack dafür ist dem obigen sehr ähnlich und die gleichen Schritte können befolgt werden, aber statt `.gitkeep` verwenden `dummy.txt` stattdessen eine `dummy.txt` . Dies hat den zusätzlichen Vorteil, dass es einfach in Windows mithilfe des Kontextmenüs erstellt werden kann. Und Sie können auch lustige Nachrichten hinterlassen. Sie können auch die `.gitkeep` Datei verwenden, um das leere Verzeichnis zu verfolgen. `.gitkeep` normalerweise eine leere Datei, die hinzugefügt wird, um das leere Verzeichnis zu verfolgen.

Dateien werden von .gitignore ignoriert

Sie können alle von git ignorierten Dateien im aktuellen Verzeichnis mit folgendem Befehl auflisten:

```
git status --ignored
```

Wenn wir also eine Repository-Struktur haben:

```
.git
.gitignore
./example_1
./dir/example_2
./example_2
```

... und .gitignore-Datei mit:

```
example_2
```

... als Ergebnis des Befehls wird sein:

```
$ git status --ignored

On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

.gitignore
.example_1

Ignored files:
  (use "git add -f <file>..." to include in what will be committed)

dir/
example_2
```

Wenn Sie rekursiv ignorierte Dateien in Verzeichnissen `--untracked-files=all` möchten, müssen Sie den zusätzlichen Parameter - `--untracked-files=all`

Ergebnis sieht so aus:

```
$ git status --ignored --untracked-files=all
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

.gitignore
example_1

Ignored files:
```

```
  (use "git add -f <file>..." to include in what will be committed)

dir/example_2
example_2
```

Dateien und Ordner ignorieren online lesen: https://riptutorial.com/de/git/topic/245/dateien-und-ordner-ignorieren

Kapitel 13: Diff-Baum

Einführung

Vergleicht den Inhalt und den Modus von Blobs, die über zwei Baumobjekte gefunden wurden.

Examples

Siehe die Dateien, die in einem bestimmten Commit geändert wurden

```
git diff-tree --no-commit-id --name-only -r COMMIT_ID
```

Verwendungszweck

```
git diff-tree [--stdin] [-m] [-c] [--cc] [-s] [-v] [--pretty] [-t] [-r] [--root] [<common-diff-options>] <tree-ish> [<tree-ish>] [<path>...]
```

Möglichkeit	Erläuterung
-r	diff rekursiv
--Wurzel	Füge das ursprüngliche Commit als diff gegen / dev / null ein

Allgemeine Diff-Optionen

Möglichkeit	Erläuterung
-z	Ausgabe diff-raw mit Zeilen, die mit NUL abgeschlossen sind.
-p	Ausgabe-Patch-Format.
-u	synonym für -p.
--Patch-with-Raw	Ausgabe eines Patches und des Diff-Raw-Formats.
--stat	zeige diffstat anstelle von patch.
--numstat	numerisches diffstat anstelle von patch anzeigen.
--Patch-with-stat	einen Patch ausgeben und das Diffstat voranstellen.
--name-only	Nur Namen der geänderten Dateien anzeigen.
--Name-Status	Namen und Status der geänderten Dateien anzeigen.

Möglichkeit	Erläuterung
- full-index	vollständigen Objektnamen in Indexzeilen anzeigen.
--abbrev = <n>	Abkürzung der Objektnamen im Diff-Tree-Header und im Diff-Raw.
-R	tauschen Sie Eingabedateipaare aus.
-B	vollständige Umschreibungen erkennen.
-M	Umbenennungen erkennen.
-C	Kopien erkennen.
- Finde-Kopien-härter	Versuchen Sie, unveränderte Dateien als Kandidaten für die Erkennung von Kopien zu verwenden.
-l <n>	Umbenennungsversuche auf Pfade beschränken.
-O	Nachbestelldifferenzen nach dem.
-S	find filepair, dessen einzige Seite die Zeichenfolge enthält.
- Pickaxe-All	Alle Dateien anzeigen, wenn -S verwendet wird und Treffer gefunden wird.
-ein Text	behandeln Sie alle Dateien als Text.

Diff-Baum online lesen: https://riptutorial.com/de/git/topic/10937/diff-baum

Kapitel 14: Durchsuchen der Geschichte

Syntax

- git log [Optionen] [Revisionsbereich] [[-] Pfad ...]

Parameter

Parameter	Erläuterung
-q, --quiet	Leise, unterdrückt die diff-Ausgabe
--Quelle	Zeigt die Quelle des Commits an
--use-mailmap	E-Mail-Map-Datei verwenden (Benutzerinformationen ändern, um Benutzer zu verpflichten)
--decorate [= ...]	Optionen gestalten
--L <n, m: file>	Protokoll für einen bestimmten Zeilenbereich in einer Datei anzeigen, beginnend von 1. Startet von Zeile n bis Zeile m. Zeigt auch diff.
--show-Unterschrift	Signaturen signierter Commits anzeigen
-i, --regexp-ignore-case	Passen Sie die Muster für die Begrenzung regulärer Ausdrücke an, ohne den Buchstaben zu berücksichtigen

Bemerkungen

Referenzen und aktuelle **Dokumentation** : Offizielle Dokumentation von git-log

Examples

"Normales" Git Log

```
git log
```

zeigt alle Ihre Commits mit dem Autor und Hash an. Dies wird in mehreren Zeilen pro Commit angezeigt. (Wenn Sie pro Commit eine einzelne Zeile anzeigen möchten, lesen Sie Onelineing.) Verwenden Sie die Taste `q` , um das Protokoll zu verlassen.

> Standardmäßig, ohne Argumente, listet git log die Commits in diesem Repository in umgekehrter chronologischer Reihenfolge auf, dh die neuesten Commits werden

> zuerst angezeigt. Wie Sie sehen, listet dieser Befehl jedes Commit mit seiner SHA-1-Prüfsumme, dem Namen und der E-Mail-Adresse des Autors, dem Datum und der Commit-Nachricht auf. - **Quelle**

Beispiel (aus dem **Free Code Camp-** Repository):

```
commit 87ef97f59e2a2f4dc425982f76f14a57d0900bcf
Merge: e50ff0d eb8b729
Author: Brian <sludge256@users.noreply.github.com>
Date:   Thu Mar 24 15:52:07 2016 -0700

    Merge pull request #7724 from BKinahan/fix/where-art-thou

    Fix 'its' typo in Where Art Thou description

commit eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5
Author: BKinahan <b.kinahan@gmail.com>
Date:   Thu Mar 24 21:11:36 2016 +0000

    Fix 'its' typo in Where Art Thou description

commit e50ff0d249705f41f55cd435f317dcfd02590ee7
Merge: 6b01875 2652d04
Author: Mrugesh Mohapatra <raisedadead@users.noreply.github.com>
Date:   Thu Mar 24 14:26:04 2016 +0530

    Merge pull request #7718 from deathsythe47/fix/unnecessary-comma

    Remove unnecessary comma from CONTRIBUTING.md
```

Wenn Sie Ihren Befehl auf das letzte `n` Commit-Protokoll beschränken möchten, können Sie einfach einen Parameter übergeben. Zum Beispiel, wenn Sie die letzten 2 Commits-Protokolle auflisten möchten

```
git log -2
```

Oneline-Protokoll

```
git log --oneline
```

zeigt alle Ihre Commits mit nur dem ersten Teil des Hash und der Commit-Nachricht. Jedes Commit wird in einer einzigen Zeile sein, wie das `oneline` Flag vorschlägt.

> Die Option oneline druckt jedes Commit in einer einzigen Zeile. Dies ist nützlich, wenn Sie viele Commits betrachten. - **Quelle**

Beispiel (aus dem **Free Code Camp-** Repository, mit demselben Code-Abschnitt aus dem anderen Beispiel):

```
87ef97f Merge pull request #7724 from BKinahan/fix/where-art-thou
eb8b729 Fix 'its' typo in Where Art Thou description
e50ff0d Merge pull request #7718 from deathsythe47/fix/unnecessary-comma
2652d04 Remove unnecessary comma from CONTRIBUTING.md
```

```
6b01875 Merge pull request #7667 from zerkms/patch-1
766f088 Fixed assignment operator terminology
d1e2468 Merge pull request #7690 from BKinahan/fix/unsubscribe-crash
bed9de2 Merge pull request #7657 from Rafase282/fix/
```

Wenn Sie den Befehl auf das letzte `n` Commit-Protokoll beschränken möchten, können Sie einfach einen Parameter übergeben. Zum Beispiel, wenn Sie die letzten 2 Commits-Protokolle auflisten möchten

```
git log -2 --oneline
```

Schöneres Protokoll

Um das Protokoll in einer hübscheren grafischen Struktur zu sehen, verwenden Sie:

```
git log --decorate --oneline --graph
```

Beispielausgabe:

```
* e0c1cea (HEAD -> maint, tag: v2.9.3, origin/maint) Git 2.9.3
*   9b601ea Merge branch 'jk/difftool-in-subdir' into maint
|\
| * 32b8c58 difftool: use Git::* functions instead of passing around state
| * 98f917e difftool: avoid $GIT_DIR and $GIT_WORK_TREE
| * 9ec26e7 difftool: fix argument handling in subdirs
* |   f4fd627 Merge branch 'jk/reset-ident-time-per-commit' into maint
...
```

Da es sich um einen ziemlich großen Befehl handelt, können Sie einen Alias vergeben:

```
git config --global alias.lol "log --decorate --oneline --graph"
```

So verwenden Sie die Alias-Version:

```
# history of current branch :
git lol

# combined history of active branch (HEAD), develop and origin/master branches :
git lol HEAD develop origin/master

# combined history of everything in your repo :
git lol --all
```

Protokoll mit Änderungen inline

Verwenden Sie die Optionen `-p` oder `--patch` , um das Protokoll mit Inline-Änderungen `--patch` .

```
git log --patch
```

Beispiel (aus Trello Scientist Repository)

```
ommit 8ea1452aca481a837d9504f1b2c77ad013367d25
Author: Raymond Chou <info@raychou.io>
Date:   Wed Mar 2 10:35:25 2016 -0800

    fix readme error link

diff --git a/README.md b/README.md
index 1120a00..9bef0ce 100644
--- a/README.md
+++ b/README.md
@@ -134,7 +134,7 @@ the control function threw, but *after* testing the other functions and
readying
 the logging. The criteria for matching errors is based on the constructor and
 message.

-You can find this full example at [examples/errors.js](examples/error.js).
+You can find this full example at [examples/errors.js](examples/errors.js).

 ## Asynchronous behaviors


commit d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa
:
```

Protokollsuche

```
git log -S"#define SAMPLES"
```

Sucht nach **Hinzufügen** oder **Entfernen** einer bestimmten Zeichenfolge oder der Zeichenfolge, **die** mit REGEXP **übereinstimmt** . In diesem Fall suchen wir nach Hinzufügen / Entfernen der Zeichenfolge `#define SAMPLES` . Zum Beispiel:

```
+#define SAMPLES  100000
```

oder

```
-#define SAMPLES  100000
```

```
git log -G"#define SAMPLES"
```

Sucht nach **Änderungen** in **Zeilen, die eine** bestimmte Zeichenfolge oder die entsprechende Zeichenfolge enthalten **,** die von REGEXP bereitgestellt wird. Zum Beispiel:

```
-#define SAMPLES  100000
+#define SAMPLES  100000000
```

Alle Beiträge nach Autorennamen gruppieren

`git shortlog` fasst `git log` und gruppen nach `git shortlog` zusammen

Wenn keine Parameter angegeben werden, wird eine Liste aller Commits pro Committer in

chronologischer Reihenfolge angezeigt.

```
$ git shortlog
Committer 1 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
    ...
Committer 2 (<number_of_commits>):
    Commit Message 1
    Commit Message 2
    ...
```

Um einfach die Anzahl der Commits zu sehen und die Commit-Beschreibung zu unterdrücken, übergeben Sie die Zusammenfassungsoption:

```
-s

--summary
```

```
$ git shortlog -s
<number_of_commits> Committer 1
<number_of_commits> Committer 2
```

Übergeben Sie die nummerierte Option, um die Ausgabe nach Anzahl der Commits statt alphabetisch nach Committer-Namen zu sortieren:

```
-n

--numbered
```

Um die E-Mail eines Committers hinzuzufügen, fügen Sie die E-Mail-Option hinzu:

```
-e

--email
```

Eine benutzerdefinierte Formatoption kann auch bereitgestellt werden, wenn Sie andere Informationen als den Betreff anzeigen möchten:

```
--format
```

Dies kann eine beliebige Zeichenfolge sein, die von der Option `--format` von `git log` akzeptiert wird.

Weitere Informationen hierzu finden **Sie** oben unter **Farbprotokolle** .

Protokolle filtern

```
git log --after '3 days ago'
```

Bestimmte Termine funktionieren auch:

```
git log --after 2016-05-01
```

Wie bei anderen Befehlen und Flags, die einen Datumsparameter akzeptieren, wird das zulässige Datumsformat von GNU date unterstützt (sehr flexibel).

Ein Alias für - `--after` ist `--since` .

Flaggen gibt es auch für das Gegenteil: `--before` und - `--until` .

Sie können Protokolle auch nach `author` filtern. z.B

```
git log --author=author
```

Protokoll für einen Zeilenbereich in einer Datei

```
$ git log -L 1,20:index.html
commit 6a57fde739de66293231f6204cbd8b2feca3a869
Author: John Doe <john@doe.com>
Date:   Tue Mar 22 16:33:42 2016 -0500

    commit message

diff --git a/index.html b/index.html
--- a/index.html
+++ b/index.html
@@ -1,17 +1,20 @@
 <!DOCTYPE HTML>
 <html>
-       <head>
-        <meta charset="utf-8">
+
+<head>
+    <meta charset="utf-8">
     <meta http-equiv="X-UA-Compatible" content="IE=edge">
     <meta name="viewport" content="width=device-width, initial-scale=1">
```

Protokolle kolorieren

```
git log --graph --pretty=format:'%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green)(%cr)
%C(yellow)<%an>%Creset'
```

Mit der `format` können Sie Ihr eigenes Protokollausgabeformat angeben:

Parameter	Einzelheiten
`%C(color_name)`	Option färbt die Ausgabe, die danach kommt
`%h` oder% H	kürzt Commit-Hash (verwenden Sie% H für vollständigen Hash)
`%Creset`	Setzt die Farbe auf die Standard-Terminalfarbe zurück

Parameter	Einzelheiten
`%d`	Referenznamen
`%s`	Betreff [Commit-Nachricht]
`%cr`	Committer-Datum, relativ zum aktuellen Datum
`%an`	Autorenname

Eine Zeile mit dem Namen und der Uhrzeit des Commiters seit dem Festschreiben

```
tree = log --oneline --decorate --source --pretty=format:'"%Cblue %h %Cgreen %ar %Cblue %an
%C(yellow) %d %Creset %s"'  --all --graph
```

Beispiel

```
*    40554ac  3 months ago  Alexander Zolotov    Merge pull request #95 from
gmandnepr/external_plugins
|\
| *  e509f61  3 months ago  Ievgen Degtiarenko    Documenting new property
| *  46d4cb6  3 months ago  Ievgen Degtiarenko    Running idea with external plugins
| *  6253da4  3 months ago  Ievgen Degtiarenko    Resolve external plugin classes
| *  9fdb4e7  3 months ago  Ievgen Degtiarenko    Keep original artifact name as this may be
important for intellij
| *  22e82e4  3 months ago  Ievgen Degtiarenko    Declaring external plugin in intellij
section
|/
*  bc3d2cb  3 months ago  Alexander Zolotov    Ignore DTD in plugin.xml
```

Git Log zwischen zwei Zweigen

`git log master..foo` zeigt die Commits an, die auf `foo` und nicht auf `master` . Hilfreich, um zu sehen, welche Commits Sie seit dem Verzweigen hinzugefügt haben!

Protokoll mit übertragenen Dateien

```
git log --stat
```

Beispiel:

```
commit 4ded994d7fc501451fa6e233361887a2365b91d1
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Mon Jun 6 21:32:30 2016 -0300

    MercadoLibre java-sdk dependency

 mltracking-poc/.gitignore |  1 +
 mltracking-poc/pom.xml    | 14 ++++++++++++--
 2 files changed, 13 insertions(+), 2 deletions(-)
```

```
commit 506fff56190f75bc051248770fb0bcd976e3f9a5
Author: Manassés Souza <manasses.inatel@gmail.com>
Date:   Sat Jun 4 12:35:16 2016 -0300

    [manasses] generated by SpringBoot initializr

 .gitignore                                                          |  42
++++++++++++
 mltracking-poc/mvnw                                                 | 233
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 mltracking-poc/mvnw.cmd                                             | 145
++++++++++++++++++++++++++++++++++++
 mltracking-poc/pom.xml                                              |  74
++++++++++++++++++++
 mltracking-poc/src/main/java/br/com/mls/mltracking/MltrackingPocApplication.java    |  12
++++
 mltracking-poc/src/main/resources/application.properties            |   0
 mltracking-poc/src/test/java/br/com/mls/mltracking/MltrackingPocApplicationTests.java |  18
+++++
 7 files changed, 524 insertions(+)
```

Zeigt den Inhalt eines einzelnen Commits

Mit `git show` wir ein einzelnes Commit anzeigen

```
git show 48c83b3
git show 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Beispiel

```
commit 48c83b3690dfc7b0e622fd220f8f37c26a77c934
Author: Matt Clark <mrclark32493@gmail.com>
Date:   Wed May 4 18:26:40 2016 -0400

    The commit message will be shown here.

diff --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
index 0b57e4a..fa8e6a5 100755
--- a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
+++ b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
@@ -50,7 +50,7 @@ public enum BuildStatus {

                        colorMap.put(BuildStatus.UNSTABLE, Color.decode( "#FFFF55" ));
-                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#55FF55" ));
+                       colorMap.put(BuildStatus.SUCCESS,  Color.decode( "#33CC33" ));
                        colorMap.put(BuildStatus.BUILDING, Color.decode( "#5555FF" ));
```

Suche nach einer Commit-Zeichenfolge im Git-Protokoll

Durchsuchen des Git-Protokolls mit einer Zeichenfolge im Protokoll:

```
git log [options] --grep "search_string"
```

Beispiel:

```
git log --all --grep "removed file"
```

Sucht nach `removed file` in **allen Protokollen** in **allen Zweigen** .

Ab git 2.4+ kann die Suche mit der Option `--invert-grep` umgekehrt werden.

Beispiel:

```
git log --grep="add file" --invert-grep
```

Zeigt alle Commits an, die keine `add file` enthalten.

Durchsuchen der Geschichte online lesen: https://riptutorial.com/de/git/topic/240/durchsuchen-der-geschichte

Kapitel 15: Externe Zusammenführung und Difftools

Examples

Beyond Compare einrichten

Sie können den Pfad auf `bcomp.exe`

```
git config --global difftool.bc3.path 'c:\Program Files (x86)\Beyond Compare 3\bcomp.exe'
```

und konfigurieren Sie `bc3` als Standard

```
git config --global diff.tool bc3
```

Einrichten von KDiff3 als Zusammenführungswerkzeug

Folgendes sollte zu Ihrer globalen `.gitconfig` Datei `.gitconfig` werden

```
[merge]
    tool = kdiff3
[mergetool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    keepBackup = false
    keepbackup = false
    trustExitCode = false
```

Denken Sie daran, die `path` zu setzen, dass sie auf das Verzeichnis verweist, in dem Sie KDiff3 installiert haben

KDiff3 als Diff-Tool einrichten

```
[diff]
    tool = kdiff3
    guitool = kdiff3
[difftool "kdiff3"]
    path = D:/Program Files (x86)/KDiff3/kdiff3.exe
    cmd = \"D:/Program Files (x86)/KDiff3/kdiff3.exe\" \"$LOCAL\" \"$REMOTE\"
```

Einrichten einer IntelliJ-IDE als Merge-Tool (Windows)

```
[merge]
    tool = intellij
[mergetool "intellij"]
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat merge $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename "$REMOTE")
$(cd $(dirname "$BASE") && pwd)/$(basename "$BASE") $(cd $(dirname "$MERGED") &&
```

```
pwd)/$(basename "$MERGED")\"
    keepBackup = false
    keepbackup = false
    trustExitCode = true
```

Die einzige Frage ist, dass diese `cmd` Eigenschaft keine seltsamen Zeichen im Pfad akzeptiert. Wenn sich im Installationsverzeichnis der IDE seltsame Zeichen befinden (z. B. in `Program Files (x86)` installiert `Program Files (x86)` , müssen Sie einen Symlink erstellen

Einrichten einer IntelliJ-IDE als Vergleichstool (Windows)

```
[diff]
    tool = intellij
    guitool = intellij
[difftool "intellij"]
    path = D:/Program Files (x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat
    cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname
"$LOCAL") && pwd)/$(basename "$LOCAL") $(cd $(dirname "$REMOTE") && pwd)/$(basename
"$REMOTE")\"
```

Die einzige Frage ist, dass diese `cmd` Eigenschaft keine seltsamen Zeichen im Pfad akzeptiert. Wenn sich im Installationsverzeichnis der IDE seltsame Zeichen befinden (z. B. in `Program Files (x86)` installiert `Program Files (x86)` , müssen Sie einen Symlink erstellen

Externe Zusammenführung und Difftools online lesen:
https://riptutorial.com/de/git/topic/5972/externe-zusammenfuhrung-und-difftools

Kapitel 16: Festlegen

Einführung

Commits with Git sorgen für Verantwortlichkeit, indem Autoren mit Änderungen am Code zugeordnet werden. Git bietet mehrere Funktionen für die Spezifität und Sicherheit von Commits. In diesem Thema werden die richtigen Praktiken und Verfahren für das Commit mit Git beschrieben und veranschaulicht.

Syntax

- git commit [Flags]

Parameter

Parameter	Einzelheiten
--message -m	Nachricht, die in das Commit aufgenommen werden soll. Durch die Angabe dieses Parameters wird Gits normales Verhalten beim Öffnen eines Editors umgangen.
--ändern	Geben Sie an, dass die aktuell durchgeführten Änderungen zum *vorherigen* Commit hinzugefügt (geändert) werden sollen. Seien Sie vorsichtig, dies kann die Geschichte umschreiben!
--keine Bearbeitung	Verwenden Sie die ausgewählte Commit-Nachricht, ohne einen Editor zu starten. Zum Beispiel ändert `git commit --amend --no-edit` ein Commit, ohne seine Commit-Nachricht zu ändern.
-all, -a	Übernehmen Sie alle Änderungen, einschließlich der noch nicht bereitgestellten Änderungen.
--Datum	Legen Sie das Datum fest, das dem Commit zugeordnet werden soll.
--nur	Commit nur die angegebenen Pfade. Dies wird nicht das, was Sie derzeit durchgeführt haben, begehen, wenn Sie nicht dazu aufgefordert werden.
--patch, -p	Verwenden Sie die interaktive Patch-Auswahloberfläche, um die zu übergebenden Änderungen auszuwählen.
--Hilfe	Zeigt die Manpage für `git commit`
-S [keyid], -S --gpg-	Sign-Commit, GPG-Sign-Commit, `commit.gpgSign`

Parameter	Einzelheiten
sign [= keyid], -S - no-gpg-sign	Konfigurationsvariable
-n, --no-verify	Diese Option umgeht die Pre-Commit- und Commit-msg-Hooks. Siehe auch Haken

Examples

Bestätigen ohne einen Editor zu öffnen

Git öffnet normalerweise einen Editor (wie `vim` oder `emacs`), wenn Sie `git commit` ausführen. Übergeben Sie die Option `-m` , um eine Nachricht über die Befehlszeile anzugeben:

```
git commit -m "Commit message here"
```

Ihre Commit-Nachricht kann mehrere Zeilen umfassen:

```
git commit -m "Commit 'subject line' message here

More detailed description follows here (after a blank line)."
```

Alternativ können Sie mehrere `-m` Argumente übergeben:

```
git commit -m "Commit summary" -m "More detailed description follows here"
```

Siehe So schreiben Sie eine Git-Commit-Nachricht .

Udacity Git Commit Message Style Guide

Änderung eines Commits

Wenn Ihr **letztes Commit noch nicht veröffentlicht ist** (nicht in ein Upstream-Repository gepusht wird), können Sie Ihr Commit ändern.

```
git commit --amend
```

Dadurch werden die aktuell bereitgestellten Änderungen auf den vorherigen Commit angewendet.

Hinweis: Dies kann auch zum Bearbeiten einer falschen Bestätigungsmeldung verwendet werden. Der Standardeditor (normalerweise `vi` / `vim` / `emacs`) wird `emacs` und Sie können die vorherige Nachricht ändern.

So legen Sie die Festschreibungsnachricht inline fest:

```
git commit --amend -m "New commit message"
```

Oder um die vorherige Commit-Nachricht zu verwenden, ohne sie zu ändern:

```
git commit --amend --no-edit
```

Durch die Änderung wird das Commit-Datum aktualisiert, das Datum des Autors bleibt jedoch erhalten. Sie können git anweisen, die Informationen zu aktualisieren.

```
git commit --amend --reset-author
```

Sie können den Autor des Commits auch ändern mit:

```
git commit --amend --author "New Author <email@address.com>"
```

Hinweis: Beachten Sie, dass die Änderung des letzten Commits vollständig ersetzt wird und das vorherige Commit aus dem Verlauf der Zweigstelle entfernt wird. Dies sollte bei der Arbeit mit öffentlichen Repositories und in Niederlassungen mit anderen Mitarbeitern berücksichtigt werden.

Das bedeutet, wenn das frühere Commit bereits gepusht wurde, müssen Sie nach einer Änderung - `push --force` .

Änderungen direkt übernehmen

Normalerweise müssen Sie `git add` oder `git rm` , um Änderungen zum Index hinzuzufügen, bevor Sie sie `git commit` können. `--all` Option `-a` oder `--all` , um alle Änderungen (an nachverfolgten Dateien) automatisch zum Index hinzuzufügen, einschließlich Entfernungen:

```
git commit -a
```

Wenn Sie auch eine Commit-Nachricht hinzufügen möchten, würden Sie Folgendes tun:

```
git commit -a -m "your commit message goes here"
```

Sie können auch zwei Flaggen verbinden:

```
git commit -am "your commit message goes here"
```

Sie müssen nicht unbedingt alle Dateien gleichzeitig festschreiben. `--all` Flag `-a` oder `--all` und geben Sie an, welche Datei Sie direkt `--all` möchten:

```
git commit path/to/a/file -m "your commit message goes here"
```

Um direkt mehr als eine bestimmte Datei zu übergeben, können Sie auch eine oder mehrere Dateien, Verzeichnisse und Muster angeben:

```
git commit path/to/a/file path/to/a/folder/* path/to/b/file -m "your commit message goes here"
```

Leeres Commit erstellen

Im Allgemeinen sind leere Commits (oder Commits mit einem Zustand, der mit dem übergeordneten Element identisch ist) ein Fehler.

Beim Testen von Build-Hooks, CI-Systemen und anderen Systemen, die ein Commit auslösen, ist es jedoch praktisch, Commits einfach erstellen zu können, ohne eine Dummy-Datei bearbeiten / berühren zu müssen.

Das `--allow-empty` Commit `--allow-empty` die Prüfung.

```
git commit -m "This is a blank commit" --allow-empty
```

Änderungen vornehmen und festschreiben

Die Grundlagen

Nach Änderungen am Quellcode vornehmen, sollten Sie diese Änderungen mit Git **Bühne** , bevor Sie sie begehen kann.

Zum Beispiel, wenn Sie ändern `README.md` und `program.py` :

```
git add README.md program.py
```

Dies teilt git mit, dass Sie die Dateien zum nächsten Commit hinzufügen möchten.

Übernehmen Sie anschließend Ihre Änderungen mit

```
git commit
```

Beachten Sie, dass dadurch ein Texteditor geöffnet wird, der häufig vim ist . Wenn Sie mit vim nicht vertraut sind, möchten Sie vielleicht wissen, dass Sie `i` drücken können, um in den *Einfügemodus zu* wechseln, Ihre `:wq` schreiben und dann `Esc` und `:wq` , um zu speichern und zu beenden. Um zu vermeiden, dass der Texteditor geöffnet wird, `-m` Sie einfach die `-m` in Ihre Nachricht ein

```
git commit -m "Commit message here"
```

Commit-Nachrichten folgen häufig bestimmten Formatierungsregeln. Weitere Informationen finden Sie unter Good-Commit-Nachrichten .

Tastenkombinationen

Wenn Sie viele Dateien im Verzeichnis geändert haben, anstatt jede einzelne aufzulisten, können Sie Folgendes verwenden:

```
git add --all        # equivalent to "git add -a"
```

Oder fügen Sie alle Änderungen (*ausgenommen gelöschte Dateien)* aus dem Verzeichnis und den Unterverzeichnissen der obersten Ebene hinzu:

```
git add .
```

Oder nur Dateien hinzufügen, die aktuell verfolgt werden ("Update"):

```
git add -u
```

Falls gewünscht, überprüfen Sie die inszenierten Änderungen:

```
git status           # display a list of changed files
git diff --cached    # shows staged changes inside staged files
```

Bestätigen Sie abschließend die Änderungen:

```
git commit -m "Commit message here"
```

Wenn Sie nur vorhandene Dateien geändert oder Dateien gelöscht und keine neuen Dateien erstellt haben, können Sie die Aktionen von `git add` und `git commit` in einem einzigen Befehl kombinieren:

```
git commit -am "Commit message here"
```

Beachten Sie, dass dies **alle** modifizierten Dateien auf dieselbe Weise wie `git add --all` .

Sensible Daten

Sie sollten niemals vertrauliche Daten wie Kennwörter oder sogar private Schlüssel festlegen. Wenn dieser Fall eintritt und die Änderungen bereits an einen zentralen Server übertragen werden, betrachten Sie alle sensiblen Daten als gefährdet. Ansonsten ist es möglich, solche Daten nachträglich zu entfernen. Eine schnelle und einfache Lösung ist die Verwendung des "BFG Repo-Cleaner": https://rtyley.github.io/bfg-repo-cleaner/ .

Der Befehl `bfg --replace-text passwords.txt my-repo.git` liest die Passwörter aus der Datei `passwords.txt` und ersetzt diese durch `***REMOVED***` . Diese Operation berücksichtigt alle vorherigen Commits des gesamten Repositorys.

Engagement für jemand anderen

Wenn jemand anderes den Code geschrieben hat, den Sie begehen, können Sie ihm mit der Option `--author` Gutschrift `--author` :

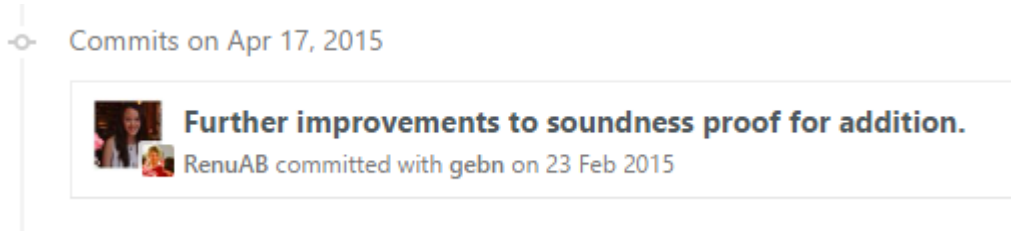
```
git commit -m "msg" --author "John Smith <johnsmith@example.com>"
```

Sie können auch ein Muster angeben, mit dem Git nach vorherigen Autoren sucht:

```
git commit -m "msg" --author "John"
```

In diesem Fall werden die Autoreninformationen aus dem letzten Commit mit einem Autor, der "John" enthält, verwendet.

Auf GitHub zeigen Commits, die auf eine der oben genannten Arten gemacht wurden, die Miniaturansicht eines großen Autors, wobei der Committer kleiner ist und vorne steht:



Änderungen in bestimmten Dateien festlegen

Sie können Änderungen an bestimmten Dateien festschreiben und mit `git add` Staging überspringen:

```
git commit file1.c file2.h
```

Oder Sie können die Dateien zunächst in Stufe bringen:

```
git add file1.c file2.h
```

und begehen sie später:

```
git commit
```

Gute Commit-Nachrichten

Für jemanden, der das `git log` durchläuft, ist es wichtig, leicht zu verstehen, worum es bei jedem Commit ging. Gute Commit-Nachrichten enthalten normalerweise eine Reihe von Aufgaben oder Problemen in einem Tracker und eine kurze Beschreibung, was und warum getan wurde, und manchmal auch, wie es gemacht wurde.

Bessere Nachrichten können wie folgt aussehen:

```
TASK-123: Implement login through OAuth
TASK-124: Add auto minification of JS/CSS files
TASK-125: Fix minifier error when name > 200 chars
```

Die folgenden Meldungen wären jedoch nicht ganz so nützlich:

```
fix                         // What has been fixed?
just a bit of a change      // What has changed?
TASK-371                    // No description at all, reader will need to look at the tracker
themselves for an explanation
Implemented IFoo in IBar    // Why it was needed?
```

Um zu testen, ob eine Commit-Nachricht in der richtigen Stimmung geschrieben wurde, können Sie das Leerzeichen durch die Nachricht ersetzen und prüfen, ob es sinnvoll ist:

Wenn ich dieses Commit hinzufüge, werde ich ___ zu meinem Repository.

Die sieben Regeln einer großen Git-Commit-Nachricht

1. Trennen Sie die Betreffzeile durch eine Leerzeile vom Hauptteil
2. Begrenzen Sie die Betreffzeile auf 50 Zeichen
3. Machen Sie die Betreffzeile groß
4. Beenden Sie die Betreffzeile nicht mit einem Punkt
5. Verwenden Sie die imperative Stimmung in der Betreffzeile
6. Umbrechen Sie jede Körperzeile manuell mit 72 Zeichen
7. Verwenden Sie den Körper, um zu erklären, *was* und *warum* anstatt *wie*

7 Regeln aus dem Blog von Chris Beam .

Festschreiben an einem bestimmten Datum

```
git commit -m 'Fix UI bug' --date 2016-07-01
```

Der `--date` Parameter legt den *Autor Datum.* Dieses Datum wird beispielsweise in der Standardausgabe von `git log` .

So erzwingen Sie auch das *Commit-Datum* :

```
GIT_COMMITTER_DATE=2016-07-01 git commit -m 'Fix UI bug' --date 2016-07-01
```

Der Parameter date akzeptiert die flexiblen Formate, die von GNU date unterstützt werden, beispielsweise:

```
git commit -m 'Fix UI bug' --date yesterday
git commit -m 'Fix UI bug' --date '3 days ago'
git commit -m 'Fix UI bug' --date '3 hours ago'
```

Wenn das Datum keine Uhrzeit angibt, wird die aktuelle Uhrzeit verwendet und nur das Datum wird überschrieben.

Auswählen, welche Zeilen zum Festlegen bereitgestellt werden sollen

Angenommen, Sie haben viele Änderungen in einer oder mehreren Dateien, aber von jeder Datei, für die Sie nur einige der Änderungen festschreiben möchten, können Sie die gewünschten Änderungen wie folgt auswählen:

```
git add -p
```

oder

```
git add -p [file]
```

Jede Ihrer Änderungen wird einzeln angezeigt. Bei jeder Änderung werden Sie aufgefordert, eine der folgenden Optionen auszuwählen:

```
y - Yes, add this hunk

n - No, don't add this hunk

d - No, don't add this hunk, or any other remaining hunks for this file.
    Useful if you've already added what you want to, and want to skip over the rest.

s - Split the hunk into smaller hunks, if possible

e - Manually edit the hunk.  This is probably the most powerful option.
    It will open the hunk in a text editor and you can edit it as needed.
```

Dadurch werden die ausgewählten Teile der Dateien in Szene gesetzt. Dann können Sie alle inszenierten Änderungen wie folgt festlegen:

```
git commit -m 'Commit Message'
```

Die nicht bereitgestellten oder festgeschriebenen Änderungen werden weiterhin in Ihren Arbeitsdateien angezeigt und können bei Bedarf später festgeschrieben werden. Wenn die verbleibenden Änderungen unerwünscht sind, können sie verworfen werden mit:

```
git reset --hard
```

Abgesehen davon, dass eine große Änderung in kleinere Commits zerlegt wird, ist dieser Ansatz auch hilfreich, um zu *überprüfen,* was Sie gerade festlegen. Indem Sie jede Änderung einzeln bestätigen, haben Sie die Möglichkeit zu überprüfen, was Sie geschrieben haben, und Sie können das versehentliche Bereitstellen von unerwünschtem Code wie println / logging-Anweisungen vermeiden.

Ändern der Zeit eines Commits

Sie können den Zeitpunkt eines Commits mit ändern

```
git commit --amend --date="Thu Jul 28 11:30 2016 -0400"
```

oder auch

```
git commit --amend --date="now"
```

Änderung des Autors eines Commits

Wenn Sie ein Commit als falschen Autor festlegen, können Sie es ändern und dann ändern

```
git config user.name "Full Name"
git config user.email "email@example.com"

git commit --amend --reset-author
```

Die GPG-Signatur wird ausgeführt

1. Bestimmen Sie Ihre Schlüssel-ID

   ```
   gpg --list-secret-keys --keyid-format LONG

   /Users/davidcondrey/.gnupg/secring.gpg
   ------------------------------------
   sec   2048R/YOUR-16-DIGIT-KEY-ID YYYY-MM-DD [expires: YYYY-MM-DD]
   ```

   Ihre ID ist ein alphanumerischer 16-stelliger Code, der auf den ersten Schrägstrich folgt.

2. Definieren Sie Ihre Schlüssel-ID in Ihrer git config

   ```
   git config --global user.signingkey YOUR-16-DIGIT-KEY-ID
   ```

3. Ab der Version 1.7.9 akzeptiert git commit die Option -S, um Ihren Commits eine Signatur zuzuordnen. Bei Verwendung dieser Option werden Sie zur Eingabe Ihrer GPG-Passphrase aufgefordert und Ihre Signatur zum Festschreibungsprotokoll hinzugefügt.

   ```
   git commit -S -m "Your commit message"
   ```

Festlegen online lesen: https://riptutorial.com/de/git/topic/323/festlegen

Kapitel 17: Git Branch Name auf Bash Ubuntu

Einführung

Diese Dokumentation behandelt den **Filialnamen** des Git auf dem **Bash-** Terminal. Wir Entwickler müssen den git-Zweignamen sehr häufig finden. Wir können den Zweignamen zusammen mit dem Pfad zum aktuellen Verzeichnis hinzufügen.

Examples

Filialname im Terminal

Was ist PS1?

PS1 steht für Prompt String 1. Der Prompt-String steht in der Linux / UNIX-Shell zur Verfügung. Wenn Sie Ihr Terminal öffnen, wird der in der PS1-Variablen definierte Inhalt in der Bash-Eingabeaufforderung angezeigt. Um der Bash-Eingabeaufforderung einen Zweignamen hinzuzufügen, müssen Sie die PS1-Variable bearbeiten (setzen Sie den Wert von PS1 in ~ / .bash_profile).

Git-Zweigname anzeigen

Fügen Sie Ihrem ~ / .bash_profile folgende Zeilen hinzu

```
git_branch() {
 git branch 2> /dev/null | sed -e '/^[^*]/d' -e 's/* \(.*\)/ (\1)/'
}
export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\$(git_branch)\[\033[00m\] $ "
```

Diese git_branch-Funktion findet den Zweignamen, in dem wir uns befinden. Sobald wir mit diesen Änderungen fertig sind, können wir das git-Repo auf dem Terminal ignorieren und den Zweignamen sehen.

Git Branch Name auf Bash Ubuntu online lesen: https://riptutorial.com/de/git/topic/8320/git-branch-name-auf-bash-ubuntu

Kapitel 18: Git Clean

Syntax

- `git clean [-d] [-f] [-i] [-n] [-q] [-e <pattern>] [-x | -X] [--] <path>`

Parameter

Parameter	Einzelheiten
-d	Entfernen Sie nicht protokollierte Verzeichnisse zusätzlich zu nicht protokollierten Dateien. Wenn ein nicht protokolliertes Verzeichnis von einem anderen Git-Repository verwaltet wird, wird es standardmäßig nicht entfernt. Verwenden Sie die Option -f zweimal, wenn Sie ein solches Verzeichnis wirklich entfernen möchten.
-f, --force	Wenn die Git-Konfigurationsvariable clean ist. `requireForce` ist nicht auf false gesetzt, git clean `requireForce` das Löschen von Dateien oder Verzeichnissen ab, es sei denn -f, -n oder -i. Git lehnt es ab, Verzeichnisse mit dem .git-Unterverzeichnis oder der Datei zu löschen, sofern nicht ein zweites -f angegeben wird.
-i, --interaktiv	Interaktiv fordert das Entfernen jeder Datei auf.
-n, - dry-run	Zeigt nur eine Liste der zu entfernenden Dateien an, ohne sie tatsächlich zu entfernen.
-q, - leise	Nur Fehler anzeigen, nicht die Liste der erfolgreich entfernten Dateien.

Examples

Reinigen Sie ignorierte Dateien

```
git clean -fX
```

Entfernt alle ignorierten Dateien aus dem aktuellen Verzeichnis und allen Unterverzeichnissen.

```
git clean -Xn
```

Zeigt eine Vorschau aller Dateien an, die gesäubert werden sollen.

Reinigen Sie alle nicht verfolgten Verzeichnisse

```
git clean -fd
```

Entfernt alle nicht erfassten Verzeichnisse und die darin enthaltenen Dateien. Es wird im aktuellen Arbeitsverzeichnis gestartet und durchläuft alle Unterverzeichnisse.

```
git clean -dn
```

Zeigt eine Vorschau aller Verzeichnisse an, die gesäubert werden sollen.

Entfernen Sie nicht zurückverfolgte Dateien mit Gewalt

```
git clean -f
```

Entfernt alle nicht protokollierten Dateien.

Interaktiv reinigen

```
git clean -i
```

Druckt Elemente aus, die entfernt werden sollen, und fordert Sie mit den folgenden Befehlen zur Bestätigung auf:

```
Would remove the following items:
  folder/file1.py
  folder/file2.py
*** Commands ***
    1: clean        2: filter by pattern        3: select by numbers        4: ask each
    5: quit        6: help
What now>
```

Interaktive Option `i` kann zusammen mit anderen Optionen wie `x` , `d` usw. hinzugefügt werden.

Git Clean online lesen: https://riptutorial.com/de/git/topic/1254/git-clean

Kapitel 19: Git Diff

Syntax

- `git diff [options] [<commit>] [--] [<path>…]`
- `git diff [options] --cached [<commit>] [--] [<path>…]`
- `git diff [options] <commit> <commit> [--] [<path>…]`
- `git diff [options] <blob> <blob>`
- `git diff [options] [--no-index] [--] <path> <path>`

Parameter

Parameter	Einzelheiten
-p, -u, --patch	Patch generieren
-s, --no-patch	Differenzausgabe unterdrücken. Nützlich für Befehle wie `git show` , die den Patch standardmäßig `--patch` oder um den Effekt von `--patch`
--roh	Generiere den Unterschied im Rohformat
--diff-algorithmus =	Wählen Sie einen Diff-Algorithmus. Die Varianten sind wie folgt: `myers` , `minimal` , `patience` , `histogram`
--Zusammenfassung	Ausgabe einer komprimierten Zusammenfassung erweiterter Header-Informationen wie Erstellung, Umbenennung und Modusänderung
--name-only	Nur Namen der geänderten Dateien anzeigen
--Name-Status	Namen und Status geänderter Dateien anzeigen Die häufigsten Status sind M (Modified), A (Hinzugefügt) und D (Gelöscht).
--prüfen	Warnen, wenn Änderungen Konfliktmarken oder Whitespace-Fehler verursachen. Was als Whitespace-Fehler betrachtet wird, wird durch `core.whitespace` Konfiguration von `core.whitespace` gesteuert. Nachgestellte Leerzeichen (einschließlich Zeilen, die nur aus Leerzeichen bestehen) und ein Leerzeichen, auf das unmittelbar ein Tabulatorzeichen innerhalb des ersten Einzugs der Zeile folgt, werden standardmäßig als Leerzeichenfehler betrachtet. Wird mit einem Status ungleich Null beendet, wenn Probleme gefunden werden. Nicht kompatibel mit --exit-code
- full-index	Zeigen Sie anstelle der ersten Handvoll Zeichen die vollständigen Pre- und Post-Image-Blob-Objektnamen in der "Index" -Zeile an, wenn Sie eine Patch-Format-Ausgabe generieren
--binär	Zusätzlich zu `--full-index` kann ein binärer Diff ausgegeben werden,

Parameter	Einzelheiten
	der mit `git apply` angewendet werden kann
-ein Text	Alle Dateien als Text behandeln.
--Farbe	Stellen Sie den Farbmodus ein. Verwenden `--color=always` also `--color=always` wenn Sie einen Unterschied auf weniger `--color=always` und die Farbe von `--color=always` möchten

Examples

Zeigen Sie Unterschiede im Arbeitszweig

```
git diff
```

Daraufhin werden die nicht *bereitgestellten* Änderungen im aktuellen Zweig aus dem Commit davor *angezeigt* . Es wird nur die Änderungen gegenüber dem Index zeigen, was bedeutet , es zeigt , was Sie zum nächsten hinzufügen *könnten* begehen, haben aber nicht. Um diese Änderungen `git add` , können Sie `git add` .

Wenn eine Datei bereitgestellt wird, aber nach ihrer `git diff` geändert wurde, zeigt `git diff` die Unterschiede zwischen der aktuellen Datei und der bereitgestellten Version an.

Zeigt Unterschiede für bereitgestellte Dateien

```
git diff --staged
```

Dadurch werden die Änderungen zwischen dem vorherigen Commit und den aktuell bereitgestellten Dateien angezeigt.

HINWEIS: Sie können auch die folgenden Befehle verwenden, um dasselbe zu erreichen:

```
git diff --cached
```

Welches ist nur ein Synonym für - `--staged` oder

```
git status -v
```

Dadurch werden die ausführlichen Einstellungen des `status` ausgelöst.

Zeigen Sie sowohl gestaffelte als auch nicht bereitgestellte Änderungen an

Um alle inszenierten *und nicht* inszenierten Änderungen anzuzeigen, verwenden Sie:

```
git diff HEAD
```

HINWEIS: Sie können auch den folgenden Befehl verwenden:

```
git status -vv
```

Der Unterschied besteht darin, dass die Ausgabe des letzteren Ihnen tatsächlich anzeigt, welche Änderungen für das Commit bereitstehen und welche nicht.

Zeige Änderungen zwischen zwei Commits

```
git diff 1234abc..6789def    # old   new
```

Beispiel: Zeigen Sie die Änderungen der letzten 3 Commits an:

```
git diff @~3..@    # HEAD -3   HEAD
```

Hinweis: Die beiden Punkte (..) sind optional, sorgen jedoch für Klarheit.

Dadurch wird der Textunterschied zwischen den Commits angezeigt, unabhängig davon, wo sie sich im Baum befinden.

Verwenden von meld, um alle Änderungen im Arbeitsverzeichnis anzuzeigen

```
git difftool -t meld --dir-diff
```

zeigt die Änderungen des Arbeitsverzeichnisses an. Alternative,

```
git difftool -t meld --dir-diff  [COMMIT_A] [COMMIT_B]
```

zeigt die Unterschiede zwischen zwei spezifischen Commits.

Zeigt Unterschiede für eine bestimmte Datei oder ein bestimmtes Verzeichnis

```
git diff myfile.txt
```

Zeigt die Änderungen zwischen dem vorherigen Commit der angegebenen Datei (`myfile.txt`) und der lokal geänderten Version an, die noch nicht bereitgestellt wurde.

Dies funktioniert auch für Verzeichnisse:

```
git diff documentation
```

Das obige zeigt die Änderungen zwischen dem vorherigen Commit aller Dateien im angegebenen Verzeichnis (`documentation/`) und den lokal modifizierten Versionen dieser Dateien, die noch nicht bereitgestellt wurden.

Um den Unterschied zwischen einer Version einer Datei in einem bestimmten Commit und der

lokalen `HEAD` Version zu zeigen, können Sie das Commit angeben, mit dem Sie vergleichen möchten:

```
git diff 27fa75e myfile.txt
```

Oder wenn Sie die Version zwischen zwei separaten Commits sehen möchten:

```
git diff 27fa75e ada9b57 myfile.txt
```

Um den Unterschied zwischen der durch den Hash `ada9b57` angegebenen Version und dem letzten Commit für den Zweig `my_branchname` für nur das relative Verzeichnis `my_changed_directory/` , können Sie `my_changed_directory/` tun:

```
git diff ada9b57 my_branchname my_changed_directory/
```

Anzeige eines Wortunterschieds für lange Zeilen

```
git diff [HEAD|--staged...] --word-diff
```

Anstatt die geänderten Zeilen anzuzeigen, werden Unterschiede innerhalb der Zeilen angezeigt. Zum Beispiel anstatt:

```
-Hello world
+Hello world!
```

Wenn die gesamte Zeile als geändert markiert ist, ändert `word-diff` die Ausgabe in:

```
Hello [-world-]{+world!+}
```

Sie können die Marken `[-` , `-]` , `{+` , `+}` `--word-diff=color` indem Sie `--word-diff=color` oder `--color-words` . Dies wird nur die Farbkodierung verwenden, um den Unterschied zu markieren:

Anzeigen einer dreifachen Zusammenführung einschließlich des gemeinsamen Vorfahren

```
git config --global merge.conflictstyle diff3
```

Legt den `diff3` Stil als Standard fest: Anstelle des üblichen Formats in Konfliktbereichen werden die beiden Dateien angezeigt:

```
<<<<<<< HEAD
left
=======
right
```

```
>>>>>>> master
```

es wird einen zusätzlichen Abschnitt enthalten, der den ursprünglichen Text enthält (kommt vom gemeinsamen Vorfahren):

```
<<<<<<< HEAD
first
second
|||||||
first
=======
last
>>>>>>> master
```

Dieses Format erleichtert das Verständnis von Zusammenführungskonflikten, z. In diesem Fall wurde lokal `second` hinzugefügt, während Remote `first` und `last` geändert wurde.

```
last
second
```

Die gleiche Auflösung wäre mit der Standardeinstellung viel schwieriger gewesen:

```
<<<<<<< HEAD
first
second
=======
last
>>>>>>> master
```

Zeigt Unterschiede zwischen der aktuellen Version und der letzten Version

```
git diff HEAD^ HEAD
```

Dies zeigt die Änderungen zwischen dem vorherigen Commit und dem aktuellen Commit.

Diff UTF-16-kodierte Text- und Binärplistendateien

Sie können UTF-16-kodierte Dateien unterscheiden (Beispiele für Lokalisierungszeichenfolgen für iOS und macOS), indem Sie angeben, wie git diese Dateien unterscheiden soll.

Fügen Sie Ihrer `~/.gitconfig` Datei Folgendes `~/.gitconfig` .

```
[diff "utf16"]
textconv = "iconv -f utf-16 -t utf-8"
```

`iconv` ist ein Programm zum Konvertieren verschiedener Kodierungen .

Bearbeiten oder erstellen Sie dann eine `.gitattributes` Datei im Stammverzeichnis des Repositorys, in dem Sie sie verwenden möchten. Oder bearbeiten Sie einfach `~/.gitattributes` .

```
*.strings diff=utf16
```

Dadurch werden alle Dateien, die auf `.strings` vor git diffs konvertiert.

Sie können ähnliche Aktionen für andere Dateien ausführen, die in Text konvertiert werden können.

Bei binären Plist-Dateien bearbeiten Sie `.gitconfig`

```
[diff "plist"]
textconv = plutil –convert xml1 –o –
```

und `.gitattributes`

```
*.plist diff=plist
```

Zweige vergleichen

Zeigen Sie die Änderungen zwischen der Spitze des **`new`** und der Spitze des **`original`** :

```
git diff original new     # equivalent to original..new
```

Zeige alle Änderungen auf **`new`** da es vom **`original`** abgezweigt wurde:

```
git diff original...new     # equivalent to $(git merge-base original new)..new
```

Verwenden Sie nur einen Parameter wie

git diff original

ist äquivalent zu

git diff original..KOPF

Zeige Änderungen zwischen zwei Zweigen

```
git diff branch1..branch2
```

Produzieren Sie einen Patch-kompatiblen Diff

Manchmal benötigen Sie nur einen Diff, um den Patch anzuwenden. Der reguläre `git --diff` funktioniert nicht. Versuchen Sie es stattdessen:

```
git diff --no-prefix > some_file.patch
```

Dann können Sie es woanders umkehren:

```
patch -p0 < some_file.patch
```

Unterschied zwischen zwei Festschreibungen oder Zweigen

Unterschied zwischen zwei Zweigen anzeigen

```
git diff <branch1>..<branch2>
```

Unterschied zwischen zwei Zweigen anzeigen

```
git diff <commitId1>..<commitId2>
```

Diff mit aktuellem Zweig anzeigen

```
git diff <branch/commitId>
```

Um die Zusammenfassung der Änderungen anzuzeigen

```
git diff --stat <branch/commitId>
```

Um Dateien anzuzeigen, die sich nach einem bestimmten Commit geändert haben

```
git diff --name-only <commitId>
```

So zeigen Sie Dateien an, die sich von einem Zweig unterscheiden

```
git diff --name-only <branchName>
```

Um Dateien anzuzeigen, die sich nach einem bestimmten Commit in einem Ordner geändert haben

```
git diff --name-only <commitId> <folder_path>
```

Git Diff online lesen: https://riptutorial.com/de/git/topic/273/git-diff

Kapitel 20: Git GUI Clients

Examples

GitHub Desktop

Website: https://desktop.github.com
Kostenlos
Plattformen: OS X und Windows
Entwickelt von: GitHub

Git Kraken

Website: https://www.gitkraken.com
Preis: 60 USD / Jahr (kostenlos für Open Source, Bildung, Non-Profit, Startups oder den persönlichen Gebrauch)
Plattformen: Linux, OS X, Windows
Entwickelt von: Axosoft

SourceTree

Website: https://www.sourcetreeapp.com
Preis: kostenlos (Konto ist erforderlich)
Plattformen: OS X und Windows
Entwickler: Atlassian

gitk und git-gui

> Wenn Sie Git installieren, erhalten Sie auch seine visuellen Werkzeuge, Gitk und Git-Gui.
>
> `gitk` ist eine grafische Verlaufsanzeige. Stellen Sie sich das wie eine leistungsstarke GUI-Shell über Git Log und Git Grep vor. Dies ist das Werkzeug, das Sie verwenden können, wenn Sie versuchen, etwas zu finden, das in der Vergangenheit passiert ist, oder den Verlauf Ihres Projekts zu visualisieren.
>
> Gitk lässt sich am einfachsten über die Befehlszeile aufrufen. Einfach cd in ein Git-Repository eingeben und Folgendes eingeben:
>
> ```
> $ gitk [git log options]
> ```
>
> Gitk akzeptiert viele Befehlszeilenoptionen, von denen die meisten an die zugrunde liegende git-Protokollaktion übergeben werden. Wahrscheinlich eine der nützlichsten ist das Flag `--all` , das gitk anweist, Commits `--all` , die von jedem Ref, nicht nur von HEAD, erreichbar sind. Die Oberfläche von Gitk sieht so aus:

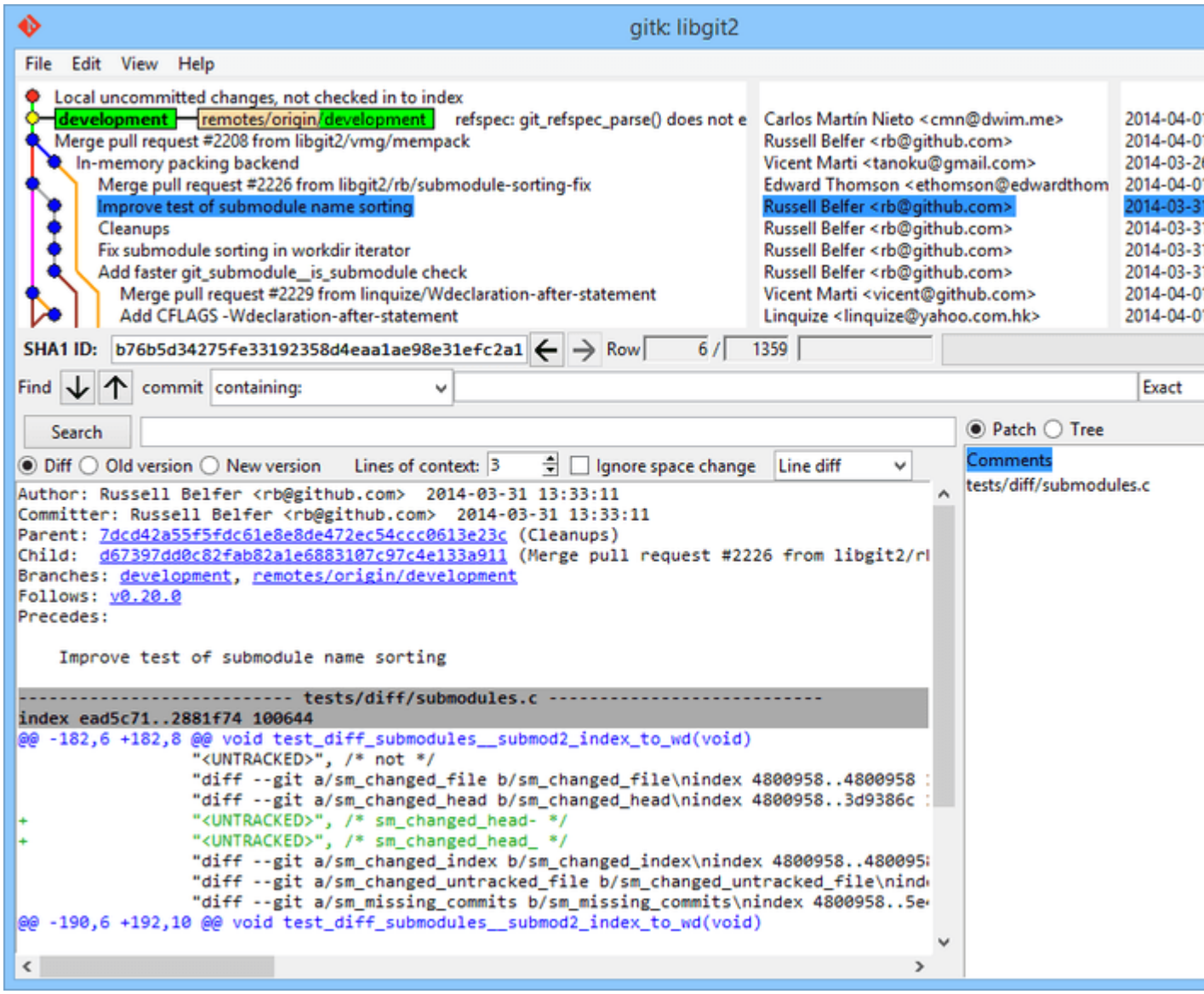


Abbildung 1-1. Der Gitk History Viewer.

Oben ist etwas, das ein bisschen wie die Ausgabe von git log --graph aussieht. Jeder Punkt steht für ein Commit, die Linien für übergeordnete Beziehungen und Refs werden als farbige Kästchen angezeigt. Der gelbe Punkt steht für HEAD und der rote Punkt für Änderungen, die noch zu einem Commit werden sollen. Unten sehen Sie eine Ansicht des ausgewählten Commits. die Kommentare und der Patch links und eine Zusammenfassungsansicht rechts. Dazwischen befindet sich eine Sammlung von Steuerelementen, die zum Durchsuchen des Verlaufs verwendet werden.

Sie können auf viele git-bezogene Funktionen zugreifen, indem Sie mit der rechten Maustaste auf einen Zweignamen oder eine Commit-Nachricht klicken. Zum Beispiel das Auschecken eines anderen Zweigs oder die Auswahl eines Commits mit Kirsche können Sie mit einem Klick erledigen.

`git-gui` hingegen ist in erster Linie ein Werkzeug für die Erstellung von Commits. Es ist auch am einfachsten von der Befehlszeile aus aufzurufen:

```
$ git gui
```

Und es sieht ungefähr so aus:

Das `git-gui` Commit-Tool.

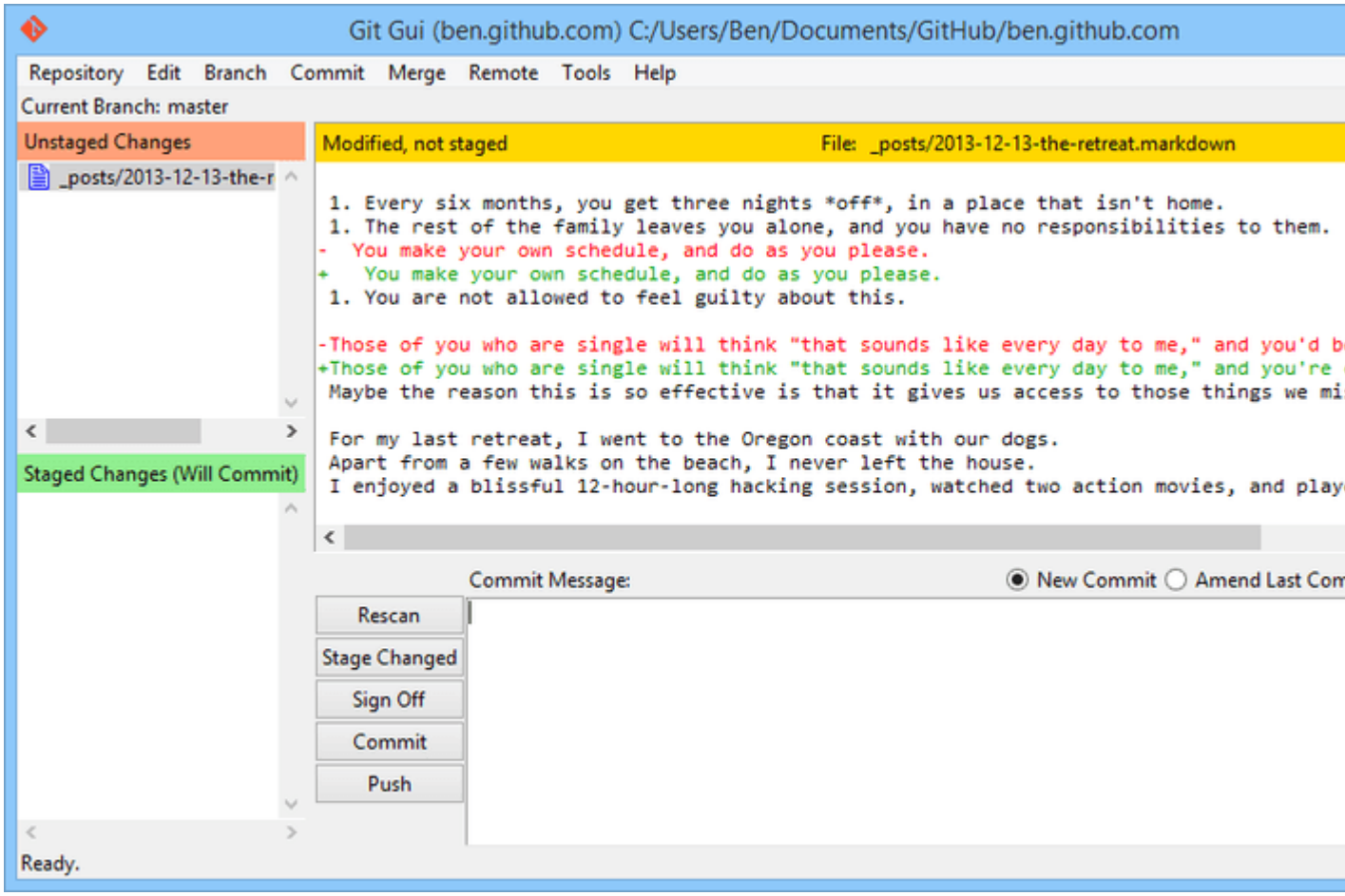


Abbildung 1-2. Das Git-Gui-Commit-Tool.

Auf der linken Seite befindet sich der Index. Ungestufte Änderungen befinden sich oben, gestaffelte Änderungen unten. Sie können ganze Dateien zwischen den beiden Status verschieben, indem Sie auf die entsprechenden Symbole klicken, oder Sie können eine Datei zum Anzeigen auswählen, indem Sie auf ihren Namen klicken.

Oben rechts befindet sich die Diff-Ansicht, in der die Änderungen für die aktuell ausgewählte Datei angezeigt werden. Sie können einzelne Knoten (oder einzelne Linien) in Szene setzen, indem Sie mit der rechten Maustaste in diesen Bereich klicken.

Unten rechts befindet sich der Nachrichten- und Aktionsbereich. Geben Sie Ihre Nachricht in das Textfeld ein und klicken Sie auf "Übernehmen", um etwas Ähnliches wie "git commit" zu tun. Sie können das letzte Commit auch ändern, indem Sie das Optionsfeld "Amend" wählen, wodurch der Bereich "Staged Changes" mit dem Inhalt des letzten Commits aktualisiert wird. Dann können Sie einige Änderungen einfach in die Bühne bringen oder entfernen, die Commit-Nachricht ändern und erneut auf "Commit" klicken, um das alte Commit durch ein neues zu ersetzen.

gitk und git-gui sind beispiele für aufgabenorientierte tools. Jeder von ihnen ist auf

einen bestimmten Zweck zugeschnitten (Historie anzeigen bzw. Commits erstellen) und lässt die für diese Aufgabe nicht erforderlichen Funktionen aus.

Quelle: https://git-scm.com/book/de/v2/Git-in-Other-Environments-Graphical-Interfaces

SmartGit

Website: http://www.syntevo.com/smartgit/
Preis: Nur für nichtkommerziellen Gebrauch. Eine unbefristete Lizenz kostet 99 USD
Plattformen: Linux, OS X, Windows
Entwickelt von: syntevo

Git-Erweiterungen

Website: https://gitextensions.github.io
Kostenlos
Plattform: Windows

Git GUI Clients online lesen: https://riptutorial.com/de/git/topic/5148/git-gui-clients

Kapitel 21: Git Large File Storage (LFS)

Bemerkungen

Git Large File Storage (LFS) zielt darauf ab, eine Einschränkung des Git-Versionskontrollsystems zu vermeiden, da es bei der Versionierung großer Dateien, insbesondere Binärdateien, schlecht funktioniert. LFS löst dieses Problem, indem der Inhalt solcher Dateien auf einem externen Server gespeichert wird und stattdessen nur ein Textzeiger auf den Pfad dieser Assets in der git-Objektdatenbank festgelegt wird.

Übliche Dateitypen, die über LFS gespeichert werden, sind in der Regel kompilierte Quellen. grafische Assets wie PSDs und JPEGs; oder 3D-Assets. Auf diese Weise können von Projekten verwendete Ressourcen in demselben Repository verwaltet werden, anstatt ein separates Verwaltungssystem extern verwalten zu müssen.

LFS wurde ursprünglich von GitHub entwickelt (https://github.com/blog/1986-announcing-git-large-file-storage-lfs) . Atlasssian hatte jedoch fast zeitgleich an einem ähnlichen Projekt gearbeitet, dem sogenannten Git-Lob . Bald wurden diese Bemühungen zusammengelegt, um eine Fragmentierung in der Branche zu vermeiden.

Examples

LFS installieren

Laden Sie herunter und installieren Sie sie entweder über Homebrew oder von der Website .

Für brauen,

```
brew install git-lfs
git lfs install
```

Oft müssen Sie auch den Dienst einrichten, der Ihre Fernbedienung hostet, damit er mit lfs arbeiten kann. Dies ist für jeden Host anders, aber es wird wahrscheinlich nur ein Kästchen angekreuzt, in dem Sie angeben, dass Sie git lfs verwenden möchten.

Deklarieren Sie bestimmte Dateitypen für die externe Speicherung

Ein üblicher Arbeitsablauf für die Verwendung von Git LFS besteht darin, zu deklarieren, welche Dateien über ein `.gitignore` System abgehört werden, genau wie `.gitignore` Dateien.

In der Regel werden Platzhalter verwendet, um bestimmte Dateitypen für das Tracking auszuwählen.

zB `git lfs track "*.psd"`

Wenn eine Datei hinzugefügt wird, die mit dem obigen Muster übereinstimmt, werden sie festgeschrieben. Wenn sie dann an die Remote gesendet wird, wird sie separat hochgeladen,

wobei ein Zeiger die Datei im Remote-Repository ersetzt.

Nachdem eine Datei mit lfs verfolgt wurde, wird Ihre `.gitattributes` Datei entsprechend aktualisiert. Github empfiehlt, Ihre lokale `.gitattributes` Datei zu begehen, anstatt mit einer globalen `.gitattributes` Datei zu arbeiten, um sicherzustellen, dass bei der Arbeit mit verschiedenen Projekten keine Probleme auftreten.

Legen Sie die LFS-Konfiguration für alle Klone fest

Erstellen Sie zum `.lfsconfig` LFS-Optionen für alle Klone eine Datei mit dem Namen `.lfsconfig` im Repository-Stammverzeichnis. Diese Datei kann LFS-Optionen auf dieselbe Weise angeben wie in `.git/config` zulässig.

Um beispielsweise eine bestimmte Datei von LFS- `.lfsconfig` auszuschließen, müssen Sie `.lfsconfig` mit folgendem Inhalt erstellen und `.lfsconfig` :

```
[lfs]
    fetchexclude = ReallyBigFile.wav
```

Git Large File Storage (LFS) online lesen: https://riptutorial.com/de/git/topic/4136/git-large-file-storage--lfs-

Kapitel 22: Git Patch

Syntax

- git am [--signoff] [--keep] [- [no-] keep-cr] [- [no-] utf8] [--3way] [--interactive] [--committer-date-is -author-date] [--ignore-date] [--ignore-space-change | --ignore-whitespace] [--whitespace = <option>] [-C <n>] [-p <n>] [--directory = <dir>] [--exclude = <Pfad>] [- include = <Pfad>] [--reject] [-q | --quiet] [- [keine-] Schere] [-S [<keyid>]] [--patch-format = <format>] [(<mbox> | <Maildir>) ...]
- git am (--continue | --skip | --abort)

Parameter

Parameter	Einzelheiten
(<mbox> | <Maildir>) ...	Die Liste der Postfachdateien, aus denen Patches gelesen werden sollen. Wenn Sie dieses Argument nicht angeben, liest der Befehl die Standardeingabe. Wenn Sie Verzeichnisse angeben, werden diese als Maildirs behandelt.
-s, --signoff	Fügen Sie der Commit-Nachricht eine Signed-off-by-Zeile hinzu, wobei Sie die Committer-Identität Ihrer Person verwenden.
-q, --quiet	Ruhe. Nur Fehlermeldungen drucken.
-u, --utf8	`git mailinfo -u` an `git mailinfo` . Die vorgeschlagene Commit-Protokollnachricht aus der E-Mail wird in UTF-8-Codierung `i18n.commitencoding` (die Konfigurationsvariable `i18n.commitencoding` kann verwendet werden, um die bevorzugte Codierung des Projekts anzugeben, wenn sie nicht UTF-8 ist). Sie können `--no-utf8` , um dies zu überschreiben.
--no-utf8	Übergeben Sie die Option -n an git mailinfo.
-3, - 3-Wege	Wenn der Patch nicht sauber angewendet wird, greifen Sie auf die 3-Wege-Zusammenführung zurück, wenn der Patch die Identität der Blobs aufzeichnet, für die er gelten soll, und wir diese lokal verfügbar haben.
--ignore-date, --ignore-space-change, --ignore-whitespace, --whitespace = <option>, -C	Diese Flags werden an das Programm git apply übergeben, das den Patch anwendet.

Parameter	Einzelheiten
<n>, -p <n>, --directory = <dir>, - exclude = <Pfad>, --include = <Pfad>, --reject	
--Patch-Format	Standardmäßig versucht der Befehl, das Patch-Format automatisch zu erkennen. Mit dieser Option kann der Benutzer die automatische Erkennung umgehen und das Patch-Format angeben, als das / die Patch (e) interpretiert werden sollen. Gültige Formate sind `mbox` , `stgit` , `stgit-series` und `hg` .
-i, --interaktiv	Führen Sie interaktiv aus.
--committer-date-is-author-date	Standardmäßig zeichnet der Befehl das Datum aus der E-Mail-Nachricht als Commit-Autor-Datum auf und verwendet den Zeitpunkt der Commit-Erstellung als Committer-Datum. Auf diese Weise kann der Benutzer das Committer-Datum lügen, indem er denselben Wert wie das Autor-Datum verwendet.
--ignore-date	Standardmäßig zeichnet der Befehl das Datum aus der E-Mail-Nachricht als Commit-Autor-Datum auf und verwendet den Zeitpunkt der Commit-Erstellung als Committer-Datum. Auf diese Weise kann der Benutzer über das Autorendatum lügen, indem er denselben Wert wie das Committerdatum verwendet.
--überspringen	Überspringe den aktuellen Patch. Dies ist nur beim Neustart eines abgebrochenen Patches von Bedeutung.
-S [<keyid>], --gpg-sign [= <keyid>]	GPG-Zeichen wird festgelegt.
--continue, -r, --gelöst	Nach einem Patch-Fehler (z. B. beim Versuch, einen in Konflikt stehenden Patch anzuwenden), hat der Benutzer diesen manuell angewendet und die Indexdatei speichert das Ergebnis der Anwendung. Machen Sie ein Commit mit dem Urheberschafts- und Commit-Protokoll, das aus der E-Mail-Nachricht und der aktuellen Indexdatei extrahiert wurde, und fahren Sie fort.
--resolvemsg = <msg>	Wenn ein Patch-Fehler auftritt, wird `<msg>` vor dem Beenden auf dem Bildschirm angezeigt. Dies überschreibt die Standardnachricht, in der Sie darüber informiert werden, dass Sie `--continue` oder `--skip` , um den Fehler zu `--continue` . Dies ist ausschließlich für den internen Gebrauch zwischen `git rebase` und `git am` .

Parameter	Einzelheiten
--abbrechen	Stellen Sie den ursprünglichen Zweig wieder her und brechen Sie den Patch-Vorgang ab.

Examples

Patch erstellen

Um einen Patch zu erstellen, gibt es zwei Schritte.

1. Nehmen Sie Ihre Änderungen vor und legen Sie sie fest.
2. Führen Sie `git format-patch <commit-reference>` , um alle Commits seit dem commit <commit-reference> (nicht eingeschlossen) in Patch-Dateien zu konvertieren.

Wenn zum Beispiel Patches aus den letzten beiden Commits generiert werden sollen:

```
git format-patch HEAD~~
```

Dadurch werden 2 Dateien erstellt, eine für jedes Commit seit `HEAD~~` wie folgt:

```
0001-hello_world.patch
0002-beginning.patch
```

Patches anwenden

Wir können `git apply some.patch` , um die Änderungen der `.patch` Datei auf Ihr aktuelles Arbeitsverzeichnis anzuwenden. Sie werden nicht inszeniert und müssen begangen werden.

Um einen Patch als Commit (mit der Commit-Nachricht) anzuwenden, verwenden Sie

```
git am some.patch
```

So wenden Sie alle Patch-Dateien auf die Baumstruktur an:

```
git am *.patch
```

Git Patch online lesen: https://riptutorial.com/de/git/topic/4603/git-patch

Kapitel 23: Git Remote

Syntax

- `git remote [-v | --verbose]`
- `git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>]<name> <url>`
- `git remote rename <old> <new>`
- `git remote remove <name>`
- `git remote set-head <name> (-a | --auto | -d | --delete | <branch>)`
- `git remote set-branches [--add] <name> <branch>...`
- `git remote set-url [--push] <name> <newurl> [<oldurl>]`
- `git remote set-url --add [--push] <name> <newurl>`
- `git remote set-url --delete [--push] <name> <url>`
- `git remote [-v | --verbose] show [-n] <name>...`
- `git remote prune [-n | --dry-run] <name>...`
- `git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)...]`
- `git remote show <name>`

Parameter

Parameter	Einzelheiten
-v, --verbose	Laufen Sie wortreich.
-m <Master>	Setzt den Kopf auf den <master> -Zweig der Fernbedienung
--mirror = holen	Refs werden nicht im Namespace refs / remotes gespeichert, sondern im lokalen Repo gespiegelt
--mirror = drücken	`git push` verhält sich so, als ob --mirror übergeben wurde
--keine Tags	`git fetch <name>` importiert keine Tags aus dem Remote-Repo
-t <Zweig>	Gibt die entfernte Stelle an, an der *nur* <Zweig> aufgezeichnet wird
-f	`git fetch <name>` wird unmittelbar nach dem Einrichten der Remote ausgeführt
--Stichworte	`git fetch <name>` importiert jedes Tag aus dem Remote-Repo
-a, --auto	Der HEAD des Symbol-Refs ist auf den gleichen Zweig wie der HEAD der Fernbedienung eingestellt
-d, --delete	Alle aufgelisteten Referenzen werden aus dem Remote-Repository gelöscht
--hinzufügen	Fügt der Liste der aktuell verfolgten Zweige (Satzzweige) <Name> hinzu.

Parameter	Einzelheiten
--hinzufügen	Anstatt eine URL zu ändern, wird eine neue URL hinzugefügt (set-url)
--alles	Schieben Sie alle Äste.
--löschen	Alle URLs, die mit <URL> übereinstimmen, werden gelöscht. (set-url)
--drücken	Push-URLs werden manipuliert, anstatt URLs abzurufen
-n	Die entfernten Köpfe werden nicht zuerst mit `git ls-remote <name>` abgefragt, sondern die zwischengespeicherten Informationen werden verwendet
--Probelauf	Bericht, welche Zweige beschnitten werden, aber nicht wirklich beschneiden
--Pflaume	Entferne entfernte Zweige ohne lokales Gegenstück

Examples

Fügen Sie ein Remote-Repository hinzu

Um eine Remote hinzuzufügen, verwenden Sie `git remote add` im Stammverzeichnis Ihres lokalen Repositorys.

Zum Hinzufügen eines fernen Git-Repositorys <url> als Kurznamen <Name> verwenden

```
git remote add <name> <url>
```

Der Befehl `git fetch <name>` kann dann zum Erstellen und Aktualisieren von Remote-Tracking-Zweigen `<name>/<branch>` .

Benennen Sie ein Remote-Repository um

Benennen Sie die Fernbedienung mit dem Namen `<old>` in `<new>` . Alle Remote-Tracking-Zweige und Konfigurationseinstellungen für die Remote-Umgebung werden aktualisiert.

Um einen entfernten `dev1 dev` in `dev1` :

```
git remote rename dev dev1
```

Entfernen Sie ein Remote-Repository

Entfernen Sie den Remote-Namen `<name>` . Alle Remote-Tracking-Zweige und Konfigurationseinstellungen für die Remote werden entfernt.

So entfernen Sie einen Remote-Repository- `dev` :

```
git remote rm dev
```

Remote-Repositorys anzeigen

Verwenden Sie `git remote` , um alle konfigurierten Remote-Repositorys `git remote` .

Es zeigt den Kurznamen (Aliase) jedes von Ihnen konfigurierten Remote-Handles.

```
$ git remote
premium
premiumPro
origin
```

Um detailliertere Informationen `--verbose` , kann das `--verbose` oder `-v` verwendet werden. Die Ausgabe enthält die URL und den Typ der Fernbedienung (`push` oder `pull`):

```
$ git remote -v
premiumPro    https://github.com/user/CatClickerPro.git (fetch)
premiumPro    https://github.com/user/CatClickerPro.git (push)
premium    https://github.com/user/CatClicker.git (fetch)
premium    https://github.com/user/CatClicker.git (push)
origin    https://github.com/ud/starter.git (fetch)
origin    https://github.com/ud/starter.git (push)
```

Ändern Sie die Remote-URL Ihres Git-Repositorys

Möglicherweise möchten Sie dies tun, wenn das Remote-Repository migriert wird. Der Befehl zum Ändern der Remote-URL lautet:

```
git remote set-url
```

Es sind 2 Argumente erforderlich: ein vorhandener entfernter Name (Ursprung, Upstream) und die URL.

Überprüfen Sie Ihre aktuelle Remote-URL:

```
git remote -v
origin    https://bitbucket.com/develop/myrepo.git (fetch)
origin    https://bitbucket.com/develop/myrepo.git (push)
```

Ändern Sie Ihre Remote-URL:

```
git remote set-url origin https://localserver/develop/myrepo.git
```

Überprüfen Sie erneut Ihre Remote-URL:

```
git remote -v
origin    https://localserver/develop/myrepo.git (fetch)
origin    https://localserver/develop/myrepo.git (push)
```

Weitere Informationen zum Remote-Repository anzeigen

Sie können weitere Informationen zu einem Remote-Repository anzeigen, indem Sie `git remote show <remote repository alias>` anzeigen

```
git remote show origin
```

Ergebnis:

```
remote origin
Fetch URL:  https://localserver/develop/myrepo.git
Push  URL:  https://localserver/develop/myrepo.git
HEAD branch: master
Remote branches:
  master      tracked
Local branches configured for 'git pull':
  master      merges with remote master
Local refs configured for 'git push':
  master      pushes to master      (up to date)
```

Git Remote online lesen: https://riptutorial.com/de/git/topic/4071/git-remote

Kapitel 24: Git rerere

Einführung

`rerere` (`rerere` Auflösung wiederverwenden) können Sie git mitteilen, wie Sie einen Hunk-Konflikt gelöst haben. Dadurch kann es automatisch gelöst werden, wenn git das nächste Mal auf denselben Konflikt trifft.

Examples

Aktivieren von Reerere

Um die `rerere` zu aktivieren, führen Sie den folgenden Befehl aus:

```
$ git config --global rerere.enabled true
```

Dies kann sowohl in einem bestimmten Repository als auch global erfolgen.

Git rerere online lesen: https://riptutorial.com/de/git/topic/9156/git-rerere

Kapitel 25: Git Revisions-Syntax

Bemerkungen

Viele Git-Befehle verwenden Revisionsparameter als Argumente. Je nach Befehl bezeichnen sie ein bestimmtes Commit oder für Befehle, die den Revisionsgraphen durchlaufen (z. B. **git-log (1)**), alle Commits, die von diesem Commit aus erreichbar sind. Sie werden in der Syntaxbeschreibung normalerweise als `<commit>` , `<rev>` oder `<revision>` .

Die Referenzdokumentation für die Git-Revisions-Syntax ist die Manpage **gitrevisions (7)** .

Noch fehlt auf dieser Seite:

- [_] Ausgabe von `git describe` , z. B. `v1.7.4.2-679-g3bee7fb`
- [_] `@` als Abkürzung für `HEAD`
- [_] `@{-<n>}` , z. B. `@{-1}` und `-` Bedeutung `@{-1}`
- [_] `<branchname>@{push}`
- [_] `<rev>^@` für alle Eltern von `<rev>`

Benötigt separate Dokumentation:

- [_] Bezieht sich auf Blobs und Bäume im Repository und im Index: `<rev>:<path>` und `:<n>:<path>` Syntax
- [_] Revisionsbereiche wie `A..B` , `A...B` , `B ^A` , `A^1` und Revisionslimitierung wie `-<n>` , `--since`

Examples

Revision nach Objektname angeben

```
$ git show dae86e1950b1277e545cee180551750029cfe735
$ git show dae86e19
```

Sie können eine Revision (oder in Wahrheit ein beliebiges Objekt: Tag, Baum, dh Verzeichnisinhalt, Blob, dh Dateiinhalt) mithilfe des SHA-1-Objektnamens angeben, entweder einer vollständigen 40-Byte-Hexadezimalzeichenfolge oder einer für das Repository eindeutigen Teilzeichenfolge.

Symbolische Referenznamen: Zweige, Tags, Fernverfolgungszweige

```
$ git log master    # specify branch
$ git show v1.0     # specify tag
$ git show HEAD     # specify current branch
$ git show origin   # specify default remote-tracking branch for remote 'origin'
```

Sie können die Revision mithilfe eines symbolischen Referenznamens angeben, der Verzweigungen (z. B. 'master', 'next', 'maint'), Tags (z. B. 'v1.0', 'v0.6.3-rc2'), remote- enthält.

Verfolgung von Zweigen (z. B. 'Ursprung', 'Ursprung / Master') und Sonderreferenzen wie 'HEAD' für den aktuellen Zweig.

Wenn der symbolische Referenzname mehrdeutig ist, z. B. wenn Sie sowohl Zweig als auch Tag mit dem Namen 'fix' haben (Zweig und Tag mit demselben Namen werden nicht empfohlen), müssen Sie die Art der Referenz angeben, die Sie verwenden möchten:

```
$ git show heads/fix      # or 'refs/heads/fix', to specify branch
$ git show tags/fix       # or 'refs/tags/fix', to specify tag
```

Die Standardversion: HEAD

```
$ git show         # equivalent to 'git show HEAD'
```

'HEAD' benennt das Commit, auf dem Sie die Änderungen im Arbeitsbaum vorgenommen haben, und ist normalerweise der symbolische Name für den aktuellen Zweig. Viele (aber nicht alle) Befehle, die den Revisionsparameter übernehmen, sind standardmäßig auf 'HEAD' gesetzt, wenn sie fehlen.

Reflog-Referenzen: @ { }

```
$ git show @{1}            # uses reflog for current branch
$ git show master@{1}      # uses reflog for branch 'master'
$ git show HEAD@{1}        # uses 'HEAD' reflog
```

Ein ref, normalerweise ein Zweig oder ein HEAD, gefolgt vom Suffix `@` mit einer in einem geschweiften Klammerpaar (z. B. `{1}` , `{15}`) eingeschlossenen Ordinalzahl, gibt den n-ten vorherigen Wert dieses Ref *in Ihrem **lokalen** Repository an* . Sie können die letzten reflog-Einträge mit `git reflog` Befehl `git reflog` oder mit der Option `--walk-reflogs` / `-g --walk-reflogs` um das `git log` zu `git log` .

```
$ git reflog
08bb350 HEAD@{0}: reset: moving to HEAD^
4ebf58d HEAD@{1}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{2}: pull: Fast-forward
f34be46 HEAD@{3}: checkout: moving from af40944bda352190f05d22b7cb8fe88beb17f3a7 to master
af40944 HEAD@{4}: checkout: moving from master to v2.6.3

$ git reflog gitweb-docs
4ebf58d gitweb-docs@{0}: branch: Created from master
```

Hinweis : Die Verwendung von Reflogs ersetzt praktisch den älteren Mechanismus der Verwendung von `ORIG_HEAD` ref (entspricht ungefähr `HEAD@{1}`).

Reflog-Referenzen: @ { }

```
$ git show master@{yesterday}
$ git show HEAD@{5 minutes ago}   # or HEAD@{5.minutes.ago}
```

Ein Ref, gefolgt von dem Suffix `@` mit einer Datumsangabe in einer Klammer (z. B. `{yesterday}` , `{1 month 2 weeks 3 days 1 hour 1 second ago}` oder `{1979-02-26 18:30:00}`), wird angegeben der Wert des Ref zu einem früheren Zeitpunkt (oder dem nächstgelegenen Punkt). Beachten Sie, dass dies den Status Ihrer **lokalen** Referenz zu einem bestimmten Zeitpunkt anzeigt. Zum Beispiel, was letzte Woche in Ihrer lokalen *'Master'*- Filiale war.

Sie können `git reflog` mit einem Datumsbezeichner verwenden, um die genaue Zeit `git reflog` zu der Sie im lokalen Repository etwas `git reflog` um eine Angabe zu machen.

```
$ git reflog HEAD@{now}
08bb350 HEAD@{Sat Jul 23 19:48:13 2016 +0200}: reset: moving to HEAD^
4ebf58d HEAD@{Sat Jul 23 19:39:20 2016 +0200}: commit: gitweb(1): Document query parameters
08bb350 HEAD@{Sat Jul 23 19:26:43 2016 +0200}: pull: Fast-forward
```

Tracked / Upstream-Zweig: @ {Upstream}

```
$ git log @{upstream}..       # what was done locally and not yet published, current branch
$ git show master@{upstream}  # show upstream of branch 'master'
```

Das Suffix `@{upstream}` an einen Verzweigungsnamen angehängt wird (Kurzform `<branchname>@{u}`), bezieht sich auf die Verzweigung, auf die die durch Verzweigungsname angegebene Verzweigung (über `branch.<name>.remote` und `branch.<name>.merge` oder mit `git branch --set-upstream-to=<branch>`). Ein fehlender Zweigname ist standardmäßig der aktuelle.

Zusammen mit der Syntax für Revisionsbereiche ist es sehr nützlich zu sehen, welche Commits Ihrer Filiale vor dem Upstream vorausgeht (Commits in Ihrem lokalen Repository sind noch nicht im Upstream vorhanden) und welche Commits Sie hinter sich haben (Commits im Upstream, die nicht in die lokale Filiale eingebunden sind) oder beide:

```
$ git log --oneline @{u}..
$ git log --oneline ..@{u}
$ git log --oneline --left-right @{u}...  # same as ...@{u}
```

Commit-Ancestry-Kette: ^, ~ , usw.

```
$ git reset --hard HEAD^             # discard last commit
$ git rebase --interactive HEAD~5    # rebase last 4 commits
```

Ein Suffix `^` zu einem Revisionsparameter bedeutet das erste übergeordnete Element dieses Commit-Objekts. `^<n>` bedeutet das <n> -te Elternteil (dh `<rev>^` entspricht `<rev>^1`).

Ein Suffix `~<n>` zu einem Revisionsparameter bedeutet das Commit-Objekt, das der <n> -te Generationsvorfahr des benannten Commit-Objekts ist und nur den ersten Elternteilen folgt. Dies bedeutet, dass zum Beispiel `<rev>~3` `<rev>^^^` . Als Abkürzung `<rev>~` Mittel `<rev>~1` , und ist äquivalent zu `<rev>^1` oder `<rev>^` in kurz.

Diese Syntax ist komponierbar.

Um solche symbolischen Namen zu finden, können Sie den Befehl `git name-rev` :

```
$ git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a
33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags/v0.99~940
```

Beachten Sie, dass im folgenden Beispiel `--pretty=oneline` und nicht `--oneline` verwendet werden muss

```
$ git log --pretty=oneline | git name-rev --stdin --name-only
master Sixth batch of topics for 2.10
master~1 Merge branch 'ls/p4-tmp-refs'
master~2 Merge branch 'js/am-call-theirs-theirs-in-fallback-3way'
[...]
master~14^2 sideband.c: small optimization of strbuf usage
master~16^2 connect: read $GIT_SSH_COMMAND from config file
[...]
master~22^2~1 t7810-grep.sh: fix a whitespace inconsistency
master~22^2~2 t7810-grep.sh: fix duplicated test name
```

Dereferenzieren von Zweigen und Tags: ^ 0, ^ { }

In einigen Fällen hängt das Verhalten eines Befehls davon ab, ob ihm ein Verzweigungsname, ein Variablenname oder eine beliebige Revision gegeben wird. Sie können die "De-Referenzierung" - Syntax verwenden, wenn Sie die letztere benötigen.

Ein Suffix `^` gefolgt von einem Objekttypnamen (`tag` , `commit` , `tree` , `blob`) in `v0.99.8^{commit}` (z. B. `v0.99.8^{commit}`) bedeutet, dass das Objekt bei `<rev>` rekursiv dereferenziert wird, bis ein Objekt vom Typ `<type>` gefunden oder das Objekt kann nicht mehr dereferenziert werden. `<rev>^0` ist eine Abkürzung für `<rev>^{commit}` .

```
$ git checkout HEAD^0    # equivalent to 'git checkout --detach' in modern Git
```

Ein Suffix `^` gefolgt von einem leeren `v0.99.8^{}` (z. B. `v0.99.8^{}`) bedeutet, dass das Tag rekursiv dereferenziert wird, bis ein Nicht-Tag-Objekt gefunden wird.

Vergleichen Sie

```
$ git show v1.0
$ git cat-file -p v1.0
$ git replace --edit v1.0
```

mit

```
$ git show v1.0^{}
$ git cat-file -p v1.0^{}
$ git replace --edit v1.0^{}
```

Jüngster passender Commit: ^ {/ },: /

```
$ git show HEAD^{/fix nasty bug}   # find starting from HEAD
```

```
$ git show ':/fix nasty bug'       # find starting from any branch
```

Ein Doppelpunkt (' `:` '), gefolgt von einem Schrägstrich (' `/` '), gefolgt von einem Text, benennt ein Commit, dessen Commit-Nachricht mit dem angegebenen regulären Ausdruck übereinstimmt. Dieser Name gibt das jüngste übereinstimmende Commit zurück, das von *jedem* Ref. Erreichbar ist. Der reguläre Ausdruck kann mit jedem Teil der Commit-Nachricht übereinstimmen. Zum Abgleichen von Nachrichten, die mit einem String beginnen, können Sie beispielsweise `:/^foo` verwenden `:/^foo` . Die spezielle Reihenfolge `:/!` ist reserviert für Modifikatoren für das, was abgeglichen wird. `:/!-foo` führt ein negatives Match aus, während `:/!!foo` ein Literal enthält! Charakter, gefolgt von `foo` .

Ein Suffix `^` zu einem Revisionsparameter, gefolgt von einem geschweiften Klammerpaar, das einen von einem Schrägstrich geführten Text enthält, ist mit der folgenden Syntax `:/<text>` identisch und gibt das jüngste übereinstimmende Commit zurück, das vom `<rev>` erreichbar ist `^` .

Git Revisions-Syntax online lesen: https://riptutorial.com/de/git/topic/3735/git-revisions-syntax

Kapitel 26: git send-email

Syntax

- git send-email [Optionen] <Datei | Verzeichnis | Rev-Liste Optionen>…
- git send-email - dump-Aliase

Bemerkungen

https://git-scm.com/docs/git-send-email

Examples

Verwenden Sie git send-email mit Google Mail

Hintergrund: Wenn Sie an einem Projekt wie dem Linux-Kernel arbeiten, müssen Sie statt einer Pull-Anforderung Ihre Commits zur Überprüfung an einen Listserv senden. In diesem Eintrag wird beschrieben, wie Sie git-send-E-Mail mit Google Mail verwenden.

Fügen Sie Ihrer .gitconfig-Datei Folgendes hinzu:

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

Dann im Web: Gehen Sie zu Google -> Mein Konto -> Verbundene Apps & Sites -> Weniger sichere Apps zulassen -> Einschalten

So erstellen Sie ein Patch-Set:

```
git format-patch HEAD~~~~ --subject-prefix="PATCH <project-name>"
```

Senden Sie dann die Patches an einen Listserv:

```
git send-email --annotate --to project-developers-list@listserve.example.com 00*.patch
```

So erstellen und senden Sie eine aktualisierte Version (in diesem Beispiel Version 2) des Patches:

```
git format-patch -v 2 HEAD~~~~  ......
git send-email --to project-developers-list@listserve.example.com v2-00*.patch
```

Komponieren

--von * E-Mail Von: - [Nein-] bis * E-Mail An: - [Nein-] CC * E-Mail-Adresse: - [Nein-] Bcc * E-Mail Bcc: --Subjekt * E-Mail "Betreff:" - -in-reply-to * E-Mail "In-Reply-To:" - [no-] xmailer * Header "X-Mailer:" hinzufügen (Standard). - [no-] annotate * Überprüfen Sie jeden Patch, der in einem Editor gesendet wird. --compose * Öffnet einen Editor zur Einführung. --Compose-Encoding * Encoding zur Einführung. --8bit-Kodierung * Kodierung zur Annahme von 8-Bit-Mails, wenn nicht deklariert ist - Übertragungskodierung * Zu verwendende Übertragungskodierung (quoted-printable, 8bit, base64)

Patches per Post senden

Angenommen, Sie haben viel Engagement gegen ein Projekt (hier ulogd2, offizieller Zweig ist git-svn) und Sie möchten Ihr Patchset an die Mailling-Liste devel@netfilter.org senden. Öffnen Sie dazu einfach eine Shell im Stammverzeichnis des git-Verzeichnisses und verwenden Sie:

```
git format-patch --stat -p --raw --signoff  --subject-prefix="ULOGD PATCH" -o /tmp/ulogd2/ -n
git-svn
git send-email --compose --no-chain-reply-to --to devel@netfilter.org /tmp/ulogd2/
```

Der erste Befehl erstellt eine Serie von E-Mails aus Patches in / tmp / ulogd2 / mit einem Statistikbericht und der zweite startet Ihren Editor, um eine Einführungs-E-Mail für das Patchset zu erstellen. Um furchtbare Thread-Serien zu vermeiden, kann man Folgendes verwenden:

```
git config sendemail.chainreplyto false
```

Quelle

git send-email online lesen: https://riptutorial.com/de/git/topic/4821/git-send-email

Kapitel 27: Git Tagging

Einführung

Wie die meisten Versionskontrollsysteme (VCS) kann `Git` bestimmte Punkte in der Historie als wichtig `tag` . Normalerweise verwenden `v1.0` diese Funktion zum Markieren von Release-Punkten (`v1.0` usw.).

Syntax

- Git-Tag [-a | -s | -u <keyid>] [-f] [-m <msg> | -F <file>] <tagname> [<commit> | <Objekt>]
- git tag -d <tagname>
- git tag [-n [<num>]] -l [--contains <commit>] [--contains <commit>] [--points-at <object>] [--column [= <options>] | --no-column] [--create-reflog] [--sort = <key>] [--format = <format>] [-[no-] merged [<commit>]] [<pattern>…]
- git-Tag -v [--format = <format>] <tagname>…

Examples

Alle verfügbaren Tags auflisten

Mit dem Befehl `git tag` werden alle verfügbaren Tags aufgelistet:

```
$ git tag
<output follows>
v0.1
v1.3
```

> **Hinweis** : Die `tags` werden in **alphabetischer** Reihenfolge ausgegeben.

Man kann auch `search` verfügbaren `tags search` :

```
$ git tag -l "v1.8.5*"
<output follows>
v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

Erstellen Sie Tags in GIT und drücken Sie sie

Erstellen Sie ein Tag:

- So erstellen Sie ein Tag in Ihrem aktuellen Zweig:

  ```
  git tag < tagname >
  ```

  Dadurch wird ein lokales `tag` mit dem aktuellen Status des Zweigs erstellt, in dem Sie sich befinden.

- So erstellen Sie ein Tag mit einem Commit:

  ```
  git tag tag-name commit-identifier
  ```

  Dadurch wird ein lokales `tag` mit dem Commit-Bezeichner des Zweigs erstellt, in dem Sie sich befinden.

Push in ein Commit in GIT:

- Drücken Sie ein einzelnes Tag:

  ```
  git push origin tag-name
  ```

- Drücken Sie alle Tags gleichzeitig

  ```
  git push origin --tags
  ```

Git Tagging online lesen: https://riptutorial.com/de/git/topic/10098/git-tagging

Kapitel 28: Git-Client-Side-Hooks

Einführung

Wie viele andere Versionskontrollsysteme bietet Git die Möglichkeit, bei bestimmten wichtigen Aktionen benutzerdefinierte Skripts auszulösen. Es gibt zwei Gruppen dieser Hooks: clientseitig und serverseitig. Clientseitige Hooks werden durch Vorgänge wie das Festschreiben und Zusammenführen ausgelöst, während serverseitige Hooks bei Netzwerkoperationen wie dem Empfang von Push-Commits ausgeführt werden. Sie können diese Haken aus allen möglichen Gründen verwenden.

Examples

Einen Haken installieren

Die Hooks werden alle im `hooks` Unterverzeichnis des Git-Verzeichnisses gespeichert. In den meisten Projekten ist das `.git/hooks` .

Um ein Hook-Skript zu aktivieren, legen Sie eine Datei in das `hooks` Unterverzeichnis Ihres `.git` Verzeichnisses, das entsprechend benannt ist (ohne Erweiterung) und ausführbar ist.

Git Pre-Push-Haken

Pre-Push- Skript wird von `git push` aufgerufen `git push` nachdem der Remote-Status geprüft wurde, bevor etwas gedrückt wurde. Wenn dieses Skript mit einem Status ungleich Null beendet wird, wird nichts weitergeleitet.

Dieser Hook wird mit den folgenden Parametern aufgerufen:

```
 $1 -- Name of the remote to which the push is being done (Ex: origin)
 $2 -- URL to which the push is being done (Ex:
https://<host>:<port>/<username>/<project_name>.git)
```

Informationen zu den übertragenen Commits werden als Zeilen an die Standardeingabe in der folgenden Form geliefert:

```
<local_ref> <local_sha1> <remote_ref> <remote_sha1>
```

Beispielwerte:

```
local_ref = refs/heads/master
local_sha1 = 68a07ee4f6af8271dc40caae6cc23f283122ed11
remote_ref = refs/heads/master
remote_sha1 = efd4d512f34b11e3cf5c12433bbedd4b1532716f
```

Das folgende Beispiel für das Pre-Push-Skript wurde dem Standard-Pre-Push-Beispiel

entnommen, das automatisch erstellt wurde, wenn ein neues Repository mit `git init` initialisiert wird

```
# This sample shows how to prevent push of commits where the log message starts
# with "WIP" (work in progress).

remote="$1"
url="$2"

z40=0000000000000000000000000000000000000000

while read local_ref local_sha remote_ref remote_sha
do
    if [ "$local_sha" = $z40 ]
    then
        # Handle delete
        :
    else
        if [ "$remote_sha" = $z40 ]
        then
            # New branch, examine all commits
            range="$local_sha"
        else
            # Update to existing branch, examine new commits
            range="$remote_sha..$local_sha"
        fi

        # Check for WIP commit
        commit=`git rev-list -n 1 --grep '^WIP' "$range"`
        if [ -n "$commit" ]
        then
            echo >&2 "Found WIP commit in $local_ref, not pushing"
            exit 1
        fi
    fi
done

exit 0
```

Git-Client-Side-Hooks online lesen: https://riptutorial.com/de/git/topic/8654/git-client-side-hooks

Kapitel 29: Git-Statistiken

Syntax

- Git Log [<Optionen>] [<Revisionsbereich>] [[-] <Pfad>]
- git log --pretty = kurz | git shortlog [<options>]
- git shortlog [<Optionen>] [<Revisionsbereich>] [[-] <Pfad>]

Parameter

Parameter	Einzelheiten
`-n` , `--numbered`	Sortieren Sie die Ausgabe nach der Anzahl der Commits pro Autor anstelle der alphabetischen Reihenfolge
`-s` , `--summary`	Geben Sie nur eine Commit-Count-Zusammenfassung an
`-e` , `--email`	Zeigt die E-Mail-Adresse jedes Autors an
`--format` [= <format>]	Verwenden Sie anstelle des Festschreibungsbetreffs andere Informationen, um jede Festschreibung zu beschreiben. <format> kann eine beliebige Zeichenfolge sein, die von der Option `--format` von `git log` akzeptiert wird.
`-w` [<width> [, <indent1> [, <indent2>]]]	Zeilenumbruch der Ausgabe, indem jede Zeile in `width` umbrochen wird. Die erste Zeile jedes Eintrags wird durch `indent1` Anzahl der Leerzeichen von `indent1` eingerückt, und die nachfolgenden Zeilen werden durch Leerzeichen von `indent2` eingerückt.
<Revisionsbereich>	Nur Commits im angegebenen Revisionsbereich anzeigen. Der gesamte Verlauf wird bis zum aktuellen Commit voreingestellt.
[`--`] <Pfad>	Nur Commits anzeigen, die erklären, wie der `path` der Dateien gefunden wurde. Für Pfade muss möglicherweise ein "-" vorangestellt werden, um sie von Optionen oder dem Revisionsbereich zu trennen.

Examples

Commits pro Entwickler

Git- `shortlog` wird verwendet, um die Ausgaben des Git-Protokolls zusammenzufassen und die Commits nach Autor zu gruppieren.

Standardmäßig werden alle `--summary` angezeigt, mit dem Argument `--summary` oder `-s` die

Nachrichten jedoch `--summary` und eine Liste der Autoren mit ihrer Gesamtzahl der Commits `--summary` .

`--numbered` oder `-n` ändert die Reihenfolge von alphabetisch (aufsteigend nach Autor) in absteigende Anzahl von Commits.

```
git shortlog -sn        #Names and Number of commits

git shortlog -sne       #Names along with their email ids and the Number of commits
```

oder

```
git log --pretty=format:%ae \
| gawk -- '{ ++c[$0]; } END { for(cc in c) printf "%5d %s\n",c[cc],cc; }'
```

Hinweis: Zusagen derselben Person können nicht in Gruppen zusammengefasst werden, deren Name und / oder E-Mail-Adresse anders geschrieben wurde. Zum Beispiel werden `John Doe` und `Johnny Doe` in der Liste separat angezeigt. Um dieses `.mailmap` zu beheben, beziehen Sie sich auf die `.mailmap` Funktion.

Festschreiben pro Datum

```
git log --pretty=format:"%ai" | awk '{print " : "$1}' | sort -r | uniq -c
```

Gesamtanzahl der Commits in einer Zweigstelle

```
git log  --pretty=oneline |wc -l
```

Auflistung jedes Zweigs und des Datums der letzten Revision

```
for k in `git branch -a | sed s/^..//`; do echo -e `git log -1 --pretty=format:"%Cgreen%ci
%Cblue%cr%Creset" $k --`\\t"$k";done | sort
```

Codezeilen pro Entwickler

```
git ls-tree -r HEAD | sed -Ee 's/^.{53}//' | \
while read filename; do file "$filename"; done | \
grep -E ': .*text' | sed -E -e 's/: .*//' | \
while read filename; do git blame --line-porcelain "$filename"; done | \
sed -n 's/^author //p' | \
sort | uniq -c | sort -rn
```

Alle Commits im hübschen Format auflisten

```
git log --pretty=format:"%Cgreen%ci %Cblue%cn  %Cgreen%cr%Creset %s"
```

Dies gibt einen schönen Überblick über alle Commits (1 pro Zeile) mit Datum, Benutzer und

Commit-Nachricht.

Die Option `--pretty` hat viele Platzhalter, die jeweils mit `%` . Alle Optionen finden Sie hier

Alle lokalen Git-Repositories auf dem Computer suchen

Um alle Standorte des git-Repositorys auf Ihrem Computer aufzulisten, können Sie Folgendes ausführen

```
find $HOME -type d -name ".git"
```

Vorausgesetzt, Sie haben `locate` , sollte dies viel schneller sein:

```
locate .git |grep git$
```

Wenn Sie `mlocate gnu locate` oder `mlocate` , werden nur die git-Verzeichnisse ausgewählt:

```
locate -ber \\.git$
```

Zeigt die Gesamtzahl der Commits pro Autor an

Um die Gesamtzahl der Commits zu erhalten, die jeder Entwickler oder Mitwirkende für ein Repository abgegeben hat, können Sie einfach das `git shortlog` :

```
git shortlog -s
```

welche die Namen und die Anzahl der Commits des Autors enthält.

Wenn Sie die Ergebnisse für alle Zweige berechnen möchten, fügen `--all` dem Befehl `--all` hinzu:

```
git shortlog -s --all
```

Git-Statistiken online lesen: https://riptutorial.com/de/git/topic/4609/git-statistiken

Kapitel 30: git-svn

Bemerkungen

Klonen wirklich großer SVN-Repositories

Wenn der SVN-Repo-Verlauf wirklich sehr groß ist, kann dies einige Stunden dauern, da git-svn den gesamten Verlauf des SVN-Repos neu erstellen muss. Glücklicherweise müssen Sie das SVN-Repo nur einmal klonen. Wie bei jedem anderen git-Repository können Sie den Repo-Ordner einfach auf andere Mitbearbeiter kopieren. Das Kopieren des Ordners auf mehrere Computer ist schneller als das Klonen großer SVN-Repos von Grund auf.

Über Commits und SHA1

Ihre lokalen git Commits wird *neu geschrieben* werden , wenn der Befehl `git svn dcommit` . Dieser Befehl fügt der Nachricht von git commit einen Text hinzu, der auf die auf dem SVN-Server erstellte SVN-Version verweist, was sehr nützlich ist. Wenn Sie jedoch einen neuen Text hinzufügen, müssen Sie die Nachricht eines vorhandenen Commits ändern, was eigentlich nicht möglich ist: git-Commits sind nicht veränderbar. Die Lösung besteht darin, ein neues Commit mit demselben Inhalt und der neuen Nachricht zu erstellen, aber technisch gesehen ist es sowieso ein neues Commit (dh die SHA1 des Git-Commits ändert sich).

Da für git-svn erstellte git-Commits lokal sind, unterscheiden sich die SHA1-IDs für git-Commits zwischen den einzelnen git-Repositorys! Das bedeutet, dass Sie ein SHA1 nicht verwenden können, um auf ein Commit einer anderen Person zu verweisen, da das gleiche Commit in jedem lokalen Git-Repository ein anderes SHA1 hat. Sie müssen sich auf die an die Commit-Nachricht angehängte svn-Revisionsnummer verlassen, wenn Sie auf den SVN-Server pushen, wenn Sie ein Commit zwischen verschiedenen Kopien des Repositorys referenzieren möchten.

Sie können den SHA1 jedoch für lokale Operationen verwenden (ein bestimmtes Commit anzeigen, abgleichen, zurücksetzen usw.).

Fehlerbehebung

Der Befehl git svn rebase gibt einen Prüfsummenfehler aus

Der Befehl git svn rebase löst einen Fehler ähnlich dem folgenden aus:

```
Checksum mismatch: <path_to_file> <some_kind_of_sha1>
expected: <checksum_number_1>
  got: <checksum_number_2>
```

Die Lösung für dieses Problem ist, svn auf die Revision zurückzusetzen, wenn die problematische Datei zum letzten Mal geändert wurde. Führen Sie einen git svn-Abruf durch, damit die SVN-Historie wiederhergestellt wird. Die Befehle zum SVN-Reset sind:

- git log -1 - `<path_to_file>` (kopieren Sie die SVN-Revisionsnummer, die in der Festschreibungsnachricht `<path_to_file>`)
- git svn reset `<revision_number>`
- git svn holen

Sie sollten in der Lage sein, Daten erneut aus dem SVN zu pushen / zu ziehen

Datei wurde beim Festschreiben nicht gefunden Wenn Sie versuchen, SVN abzurufen oder von diesem abzurufen, wird eine ähnliche Fehlermeldung angezeigt

```
<file_path> was not found in commit <hash>
```

Dies bedeutet, dass eine Revision in SVN versucht, eine Datei zu ändern, die aus irgendeinem Grund in Ihrer lokalen Kopie nicht vorhanden ist. Der beste Weg, um diesen Fehler zu beseitigen, besteht darin, einen Abruf zu erzwingen, der den Pfad dieser Datei ignoriert, und er wird in der neuesten SVN-Version auf seinen Status aktualisiert:

- `git svn fetch --ignore-paths <file_path>`

Examples

Klonen des SVN-Repository

Sie müssen mit dem Befehl eine neue lokale Kopie des Repository erstellen

```
git svn clone SVN_REPO_ROOT_URL [DEST_FOLDER_PATH] -T TRUNK_REPO_PATH -t TAGS_REPO_PATH -b
BRANCHES_REPO_PATH
```

Wenn Ihr SVN-Repository dem Standardlayout (Stamm, Verzweigungen, Tag-Ordner) entspricht, können Sie einige Eingaben speichern:

```
git svn clone -s SVN_REPO_ROOT_URL [DEST_FOLDER_PATH]
```

`git svn clone` überprüft jede SVN-Revision nacheinander und führt ein git-Commit in Ihrem lokalen Repository durch, um den Verlauf neu zu erstellen. Wenn das SVN-Repository viele Commits enthält, wird dies eine Weile dauern.

Wenn der Befehl abgeschlossen ist, haben Sie ein vollständiges Git-Repository mit einem lokalen Zweig namens Master, der den Trunk-Zweig im SVN-Repository verfolgt.

Die neuesten Änderungen von SVN abrufen

Das Äquivalent zu `git pull` ist der Befehl

```
git svn rebase
```

Dadurch werden alle Änderungen aus dem SVN-Repository abgerufen und *auf* Ihre lokalen Commits in Ihrem aktuellen Zweig angewendet.

Sie können den Befehl auch verwenden

```
git svn fetch
```

Abrufen der Änderungen aus dem SVN-Repository und Übertragen der Änderungen auf den lokalen Computer

Lokale Änderungen in SVN verschieben

Der Befehl

```
git svn dcommit
```

erstellt eine SVN-Revision für jedes Ihrer lokalen Git-Commits. Wie bei SVN muss Ihr lokaler Git-Verlauf mit den neuesten Änderungen im SVN-Repository `git svn rebase` . Versuchen Sie daher, wenn der Befehl fehlschlägt, zuerst ein `git svn rebase` .

Vor Ort arbeiten

Verwenden Sie einfach Ihr lokales Git-Repository als normales Git-Repo mit den normalen Git-Befehlen:

- `git add FILE` und `git checkout -- FILE` Um eine Datei zu inszenieren / entpacken
- `git commit` Zum Speichern Ihrer Änderungen. Diese Commits sind lokal und werden nicht wie beim normalen Git-Repository in das SVN-Repo "verschoben"
- `git stash` und `git stash pop` Ermöglicht die Verwendung von Stashes
- `git reset HEAD --hard` Alle lokalen Änderungen `git reset HEAD --hard`
- `git log` Zugriff auf den gesamten Verlauf im Repository
- `git rebase -i` damit Sie Ihre Ortsgeschichte frei umschreiben können
- `git branch` und `git checkout` , um lokale `git checkout` zu erstellen

Wie in der git-svn-Dokumentation angegeben ist "Subversion ist ein System, das weitaus weniger ausgereift ist als Git", so dass Sie nicht die ganze Leistungsfähigkeit von git nutzen können, ohne den Verlauf des Subversion-Servers zu beeinträchtigen. Zum Glück sind die Regeln sehr einfach: **Halten Sie die Geschichte linear**

Das bedeutet, dass Sie nahezu jede beliebige Git-Operation ausführen können: Verzweigungen erstellen, Commits entfernen / neu sortieren / zerquetschen, Historie verschieben, Commits löschen usw. Alles *außer einer Fusion* . Wenn Sie die Historie lokaler Zweige wieder integrieren `git rebase` stattdessen `git rebase` .

Wenn Sie eine Zusammenführung durchführen, wird ein Zusammenführungs-Commit erstellt. Das Besondere an Merge-Commits ist, dass sie zwei Eltern haben, und das macht die Geschichte nicht linear. Nicht-linearer Verlauf verwirrt SVN in dem Fall, dass Sie eine Zusammenführungsfestschreibung in das Repository "pushen".

Aber keine Sorge: **Sie werden nichts kaputt machen, wenn Sie ein Git-Merge-Commit an SVN "pushen"** . Wenn Sie dies tun, wird der git merge-Commit an den svn-Server gesendet, der alle

Änderungen aller Commits für diese Zusammenführung enthält. Dadurch verlieren Sie den Verlauf dieser Commits, nicht jedoch die Änderungen in Ihrem Code.

Umgang mit leeren Ordnern

git kennt das Konzept von Ordnern nicht, es funktioniert nur mit Dateien und deren Dateipfaden. Dies bedeutet, dass git keine leeren Ordner verfolgt. SVN tut es jedoch. Die Verwendung von git-svn bedeutet, *dass Änderungen, die leere Ordner mit git betreffen* , standardmäßig *nicht an SVN weitergegeben werden* .

Die Verwendung des `--rmdir` bei der Ausgabe eines Kommentars behebt dieses Problem und entfernt einen leeren Ordner in SVN, wenn Sie die letzte darin enthaltene Datei lokal löschen:

```
git svn dcommit --rmdir
```

Leider werden **vorhandene leere Ordner nicht entfernt** : Sie müssen dies manuell tun.

Um zu vermeiden, dass Sie das Flag jedes Mal hinzufügen, wenn Sie einen Commit ausführen, oder wenn Sie ein git-GUI-Tool (wie SourceTree) verwenden, können Sie dieses Verhalten mit dem Befehl als Standard festlegen:

```
git config --global svn.rmdir true
```

Dies ändert Ihre .gitconfig-Datei und fügt folgende Zeilen hinzu:

```
[svn]
rmdir = true
```

Um alle nicht erfassten Dateien und Ordner zu entfernen, die für SVN leer bleiben sollen, verwenden Sie den Befehl git:

```
git clean -fd
```

Bitte beachten Sie: Mit dem vorherigen Befehl werden alle nicht protokollierten Dateien und leeren Ordner entfernt, auch diejenigen, die von SVN protokolliert werden sollten! Wenn Sie die leeren Ordner, die von SVN verfolgt werden, generieren möchten, verwenden Sie den Befehl

```
git svn mkdirs
```

In der Praxis bedeutet dies, dass Sie, wenn Sie Ihren Arbeitsbereich von nicht protokollierten Dateien und Ordnern bereinigen möchten, immer beide Befehle verwenden müssen, um die leeren Ordner wiederherzustellen, die von SVN verfolgt werden:

```
git clean -fd && git svn mkdirs
```

git-svn online lesen: https://riptutorial.com/de/git/topic/2766/git-svn

Kapitel 31: git-tfs

Bemerkungen

Git-tfs ist ein Tool von Drittanbietern, um ein Git-Repository mit einem Team Foundation Server-Repository ("TFS") zu verbinden.

Die meisten Remote-TFVS-Instanzen fordern Ihre Anmeldeinformationen für jede Interaktion an. Die Installation des Git-Credential-Manager für Windows kann möglicherweise nicht helfen. Es kann überwunden werden, indem Sie Ihren *Namen* und Ihr *Kennwort* zu Ihrer `.git/config` hinzufügen

```
[tfs-remote "default"]
  url = http://tfs.mycompany.co.uk:8080/tfs/DefaultCollection/
  repository = $/My.Project.Name/
  username = me.name
  password = My733TPwd
```

Examples

Git-Tfs-Klon

Dadurch wird ein Ordner mit demselben Namen wie das Projekt erstellt, dh /My.Project.Name

```
$ git tfs clone http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name
```

git-tfs-Klon aus nacktem Git-Repository

Das Klonen aus einem Git-Repository ist zehnmal schneller als das Klonen direkt aus TFVS und funktioniert gut in einer Teamumgebung. Mindestens ein Teammitglied muss das Bare-Git-Repository erstellen, indem Sie zunächst den regulären git-tfs-Klon ausführen. Dann kann das neue Repository für die Zusammenarbeit mit TFVS gebootet werden.

```
$ git clone x:/fileshare/git/My.Project.Name.git
$ cd My.Project.Name
$ git tfs bootstrap
$ git tfs pull
```

git-tfs wird über Chocolatey installiert

Im Folgenden wird davon ausgegangen, dass Sie kdiff3 für Dateidiffing verwenden. Auch wenn dies nicht unbedingt erforderlich ist, ist dies eine gute Idee.

```
C:\> choco install kdiff3
```

Git kann zuerst installiert werden, damit Sie beliebige Parameter angeben können. Hier sind auch alle Unix-Tools installiert, und "NoAutoCrlf" bedeutet "checkout", wie es heißt, ein Commit, wie es ist.

```
C:\> choco install git -params '"/GitAndUnixToolsOnPath /NoAutoCrlf"'
```

Das ist alles, was Sie wirklich brauchen, um git-tfs via chocolatey zu installieren.

```
C:\> choco install git-tfs
```

git-tfs Einchecken

Starten Sie den Check-In-Dialog für TFVS.

```
$ git tfs checkintool
```

Dadurch werden alle Ihre lokalen Commits übernommen und ein einzelner Check-In erstellt.

git-tfs schieben

Alle lokalen Commits an die TFVS-Fernbedienung senden.

```
$ git tfs rcheckin
```

Hinweis: Dies schlägt fehl, wenn Check-In-Hinweise erforderlich sind. Diese können umgangen werden, indem der Commit-Nachricht `git-tfs-force: rcheckin` .

git-tfs online lesen: https://riptutorial.com/de/git/topic/2660/git-tfs

Kapitel 32: Haken

Syntax

- .git / hooks / applypatch-msg
- .git / hooks / commit-msg
- .git / hooks / post-update
- .git / hooks / Pre-Applypatch
- .git / hooks / pre-commit
- .git / hooks / prepar-commit-msg
- .git / hooks / pre-push
- .git / hooks / pre-rebase
- .git / hooks / update

Bemerkungen

`--no-verify` oder `-n` , um alle lokalen Hooks des angegebenen git-Befehls zu überspringen. ZB: `git commit -n`

Die Informationen auf dieser Seite wurden den offiziellen Git-Dokumenten und Atlassian entnommen .

Examples

Commit-msg

Dieser Hook ähnelt dem Hook " `prepare-commit-msg` , wird jedoch aufgerufen, nachdem der Benutzer eine Commit-Nachricht eingegeben hat. Dies wird normalerweise verwendet, um Entwickler zu warnen, wenn ihre Commit-Nachricht ein falsches Format hat.

Das einzige an diesen Hook übergebene Argument ist der Name der Datei, die die Nachricht enthält. Wenn Ihnen die vom Benutzer eingegebene Nachricht nicht gefällt, können Sie diese Datei entweder direkt ändern (wie bei `prepare-commit-msg`), oder Sie können den Commit vollständig abbrechen, indem Sie den Status mit einem anderen Status als null beenden.

Das folgende Beispiel wird verwendet, um zu überprüfen, ob das Wort Ticket gefolgt von einer Nummer in der Festschreibungsnachricht vorhanden ist

```
word="ticket [0-9]"
isPresent=$(grep -Eoh "$word" $1)

if [[ -z $isPresent ]]
  then echo "Commit message KO, $word is missing"; exit 1;
  else echo "Commit message OK"; exit 0;
fi
```

Lokale Haken

Lokale Hooks wirken sich nur auf die lokalen Repositorys aus, in denen sie sich befinden. Jeder Entwickler kann seine eigenen lokalen Hooks ändern, sodass er nicht zuverlässig verwendet werden kann, um eine Commit-Richtlinie durchzusetzen. Sie sollen Entwicklern die Einhaltung bestimmter Richtlinien erleichtern und mögliche Probleme auf der Straße vermeiden.

Es gibt sechs Arten von lokalen Hooks: Pre-Commit, Prepare-Commit-Msg, Commit-Msg, Post-Commit, Post-Checkout und Pre-Rebase.

Die ersten vier Haken beziehen sich auf Commits und geben Ihnen die Kontrolle über jeden Teil des Lebenszyklus eines Commits. In den letzten beiden Fällen können Sie einige zusätzliche Aktionen oder Sicherheitsüberprüfungen für die Befehle git checkout und git rebase durchführen.

Bei allen "Pre-" -Hooks können Sie die Aktion ändern, die gerade ausgeführt wird, während die "Post-" - Hooks hauptsächlich für Benachrichtigungen verwendet werden.

Post-Checkout

Dieser Hook funktioniert ähnlich wie der Hook nach dem `post-commit` , er wird jedoch immer dann aufgerufen, wenn Sie eine Referenz mit `git checkout` erfolgreich `git checkout` . Dies kann ein nützliches Werkzeug sein, um das Arbeitsverzeichnis von automatisch generierten Dateien zu löschen, was sonst zu Verwirrung führen würde.

Dieser Haken akzeptiert drei Parameter:

1. der Ref des vorherigen HEAD,
2. der Ref des neuen HEAD und
3. ein Flag, das angibt, ob es sich um eine Zweigkasse oder um eine Dateikasse handelt (`1` oder `0`).

Der Exit-Status hat keinen Einfluss auf den Befehl `git checkout` .

Post-Commit

Dieser Hook wird unmittelbar nach dem `commit-msg` Hook aufgerufen. Es kann das Ergebnis der `git commit` nicht ändern. Daher wird es hauptsächlich für Benachrichtigungszwecke verwendet.

Das Skript nimmt keine Parameter an, und sein Beendigungsstatus wirkt sich in keiner Weise auf das Festschreiben aus.

Post empfangen

Dieser Hook wird nach einer erfolgreichen Push-Operation aufgerufen. Es wird normalerweise zu Benachrichtigungszwecken verwendet.

Das Skript nimmt keine Parameter an, erhält jedoch die gleichen Informationen wie `pre-receive` per Standardeingabe:

```
<old-value> <new-value> <ref-name>
```

Pre-Commit

Dieser Hook wird jedes Mal ausgeführt, wenn Sie `git commit` ausführen, um zu überprüfen, was gerade festgeschrieben wird. Mit diesem Hook können Sie die Momentaufnahme überprüfen, die gerade festgeschrieben wird.

Dieser Hook-Typ ist für die Ausführung automatisierter Tests hilfreich, um sicherzustellen, dass die eingehende Festschreibung die vorhandene Funktionalität Ihres Projekts nicht beeinträchtigt. Dieser Hook-Typ kann auch nach Whitespace- oder EOL-Fehlern suchen.

Es werden keine Argumente an das Pre-Commit-Skript übergeben, und das Beenden mit einem Nicht-Null-Status bricht das gesamte Commit ab.

Prepare-Commit-msg

Dieser Hook wird nach dem `pre-commit` Hook aufgerufen, um den Texteditor mit einer Commit-Nachricht zu füllen. Dies wird normalerweise verwendet, um die automatisch generierten Commit-Nachrichten für komprimierte oder zusammengeführte Commits zu ändern.

Ein bis drei Argumente werden an diesen Hook übergeben:

- Der Name einer temporären Datei, die die Nachricht enthält.
- Die Art des Commits entweder
    - Nachricht (Option `-m` oder `-F`),
    - Vorlage (Option `-t`),
    - merge (wenn es sich um ein Merge-Commit handelt) oder
    - squash (wenn andere Commits gequetscht werden).
- Der SHA1-Hash des entsprechenden Commits. Dies ist nur gegeben, wenn die Option `-c` , `-C` oder `--amend` angegeben wurde.

Ähnlich wie beim `pre-commit` bricht das Beenden mit einem Nicht-Null-Status das Commit ab.

Pre-Rebase

Dieser Hook wird aufgerufen, bevor `git rebase` zu ändern beginnt. Dieser Haken wird normalerweise verwendet, um sicherzustellen, dass eine Rebase-Operation geeignet ist.

Dieser Haken benötigt 2 Parameter:

1. der vorgelagerte Zweig, aus dem die Serie gegliedert wurde, und
2. der Zweig wird umbasiert (leer, wenn der aktuelle Zweig umbasiert wird).

Sie können den Rebase-Vorgang abbrechen, indem Sie den Status ungleich Null beenden.

Vor dem Empfang

Dieser Hook wird jedes Mal ausgeführt, wenn jemand `git push` , um Commits in das Repository zu verschieben. Es befindet sich immer im Remote-Repository, das das Ziel des Push ist, und nicht im ursprünglichen (lokalen) Repository.

Der Hook wird ausgeführt, bevor Referenzen aktualisiert werden. Sie wird normalerweise verwendet, um jegliche Art von Entwicklungsrichtlinien durchzusetzen.

Das Skript nimmt keine Parameter an, aber jeder übergebene Ref wird in einer separaten Zeile bei der Standardeingabe im folgenden Format an das Skript übergeben:

```
<old-value> <new-value> <ref-name>
```

Aktualisieren

Dieser Hook wird nach dem `pre-receive` aufgerufen und funktioniert genauso. Es wird aufgerufen, bevor alles aktualisiert wird, aber es wird separat für jeden Ref aufgerufen, der verschoben wurde, und nicht für alle Refs gleichzeitig.

Dieser Haken akzeptiert die folgenden 3 Argumente:

- Name des Refs, der aktualisiert wird,
- alter Objektname in der Referenz gespeichert, und
- Neuer Objektname in der Referenz gespeichert.

Dies ist die gleiche Information, die an `pre-receive` wird. Da jedoch die `update` für jeden Ref einzeln aufgerufen wird, können Sie einige Refs ablehnen und andere zulassen.

Pre-Push

Verfügbar in Git 1.8.2 und höher.

1.8

Pre-Push-Haken können verwendet werden, um zu verhindern, dass ein Push ausgeführt wird. Dies kann aus folgenden Gründen hilfreich sein: Blockieren versehentlicher manueller Pushs auf bestimmte Zweige oder Blockieren von Pushs, wenn eine festgestellte Prüfung fehlschlägt (Komponententests, Syntax).

Ein Pre-Push-Hook wird erstellt, indem einfach eine Datei namens `pre-push` unter `.git/hooks/` und (**Gotcha-Warnung**) erstellt wird, um sicherzustellen, dass die Datei ausführbar ist: `chmod +x ./git/hooks/pre-push` .

Hier ist ein Beispiel von Hannah Wolfe , das einen Push-to-Master blockiert:

```
#!/bin/bash

protected_branch='master'
current_branch=$(git symbolic-ref HEAD | sed -e 's,.*/\(.*\),\1,')

if [ $protected_branch = $current_branch ]
```

```
then
    read -p "You're about to push master, is that what you intended? [y|n] " -n 1 -r <
/dev/tty
    echo
    if echo $REPLY | grep -E '^[Yy]$' > /dev/null
    then
        exit 0 # push will execute
    fi
    exit 1 # push will not execute
else
    exit 0 # push will execute
fi
```

Hier ist ein Beispiel von Volkan Unsal , das sicherstellt, dass RSpec-Tests erfolgreich sind, bevor der Push zugelassen wird:

```
#!/usr/bin/env ruby
require 'pty'
html_path = "rspec_results.html"
begin
  PTY.spawn( "rspec spec --format h > rspec_results.html" ) do |stdin, stdout, pid|
  begin
    stdin.each { |line| print line }
  rescue Errno::EIO
  end
end
rescue PTY::ChildExited
  puts "Child process exit!"
end

# find out if there were any errors
html = open(html_path).read
examples = html.match(/(\d+) examples/)[0].to_i rescue 0
errors = html.match(/(\d+) errors/)[0].to_i rescue 0
if errors == 0 then
  errors = html.match(/(\d+) failure/)[0].to_i rescue 0
end
pending = html.match(/(\d+) pending/)[0].to_i rescue 0

if errors.zero?
  puts "0 failed! #{examples} run, #{pending} pending"
  # HTML Output when tests ran successfully:
  # puts "View spec results at #{File.expand_path(html_path)}"
  sleep 1
  exit 0
else
  puts "\aCOMMIT FAILED!!"
  puts "View your rspec results at #{File.expand_path(html_path)}"
  puts
  puts "#{errors} failed! #{examples} run, #{pending} pending"
  # Open HTML Ooutput when tests failed
  # `open #{html_path}`
  exit 1
end
```

Wie Sie sehen, gibt es viele Möglichkeiten, aber das Kernstück besteht darin, `exit 0` wenn gute Dinge geschehen sind, und `exit 1` wenn schlechte Dinge passiert sind. Bei jedem `exit 1` der Push verhindert und der Code befindet sich im Zustand vor dem Ausführen von `git push...`

Beachten Sie bei der Verwendung von clientseitigen Hooks, dass Benutzer alle clientseitigen Hooks mit der Option "--no-verify" für einen Push überspringen können. Wenn Sie sich auf den Hook verlassen, um den Prozess durchzusetzen, können Sie sich verbrennen.

Dokumentation: https://git-scm.com/docs/githooks#_pre_push
Offizielles Beispiel: https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample

Stellen Sie sicher, dass das Maven-Build (oder ein anderes Build-System) vor dem Festschreiben ausgeführt wird

`.git/hooks/pre-commit`

```
#!/bin/sh
if [ -s pom.xml ]; then
    echo "Running mvn verify"
    mvn clean verify
    if [ $? -ne 0 ]; then
        echo "Maven build failed"
        exit 1
    fi
fi
```

Bestimmte Push-Vorgänge automatisch an andere Repositorys weiterleiten

`post-receive` Hooks können verwendet werden, um eingehende Pushs automatisch an ein anderes Repository weiterzuleiten.

```
$ cat .git/hooks/post-receive

#!/bin/bash

IFS=' '
while read local_ref local_sha remote_ref remote_sha
do

  echo "$remote_ref" | egrep '^refs\/heads\/[A-Z]+-[0-9]+$' >/dev/null && {
    ref=`echo $remote_ref | sed -e 's/^refs\/heads\///'`
    echo Forwarding feature branch to other repository: $ref
    git push -q --force other_repos $ref
  }

done
```

In diesem Beispiel sucht der `egrep` regexp nach einem bestimmten Zweigformat (hier: JIRA-12345, um Jira-Probleme zu benennen). Sie können diesen Teil ausschalten, wenn Sie natürlich alle Zweige weiterleiten möchten.

Haken online lesen: https://riptutorial.com/de/git/topic/1330/haken

Kapitel 33: Halbieren / Finden fehlerhafter Commits

Syntax

- `git bisect <subcommand> <options>`
- `git bisect start <bad> [<good>...]`
- `git bisect reset`
- `git bisect good`
- `git bisect bad`

Examples

Binäre Suche (git bisect)

`git bisect` können Sie anhand einer binären Suche herausfinden, welches Commit einen Fehler eingeführt hat.

Beginnen Sie mit der Halbierung einer Sitzung, indem Sie zwei Commit-Referenzen angeben: ein gutes Commit vor dem Fehler und ein schlechtes Commit nach dem Fehler. Im Allgemeinen ist das schlechte Commit `HEAD` .

```
# start the git bisect session
$ git bisect start

# give a commit where the bug doesn't exist
$ git bisect good 49c747d

# give a commit where the bug exist
$ git bisect bad HEAD
```

`git` startet eine binäre Suche: Die Revision wird in zwei Hälften geteilt und das Repository auf die Zwischenversion umgestellt. Überprüfen Sie den Code, um festzustellen, ob die Revision gut oder schlecht ist:

```
# tell git the revision is good,
# which means it doesn't contain the bug
$ git bisect good

# if the revision contains the bug,
# then tell git it's bad
$ git bisect bad
```

`git` wird die binäre Suche nach jeder noch verbleibenden Teilmenge der fehlerhaften Revisionen

fortsetzen, je nach Ihren Anweisungen. `git` wird eine einzelne Revision präsentieren, die, sofern Ihre Flags nicht korrekt waren, genau die Revision darstellt, in der der Fehler eingeführt wurde.

`git bisect reset` anschließend daran, `git bisect reset` auszuführen, `git bisect reset` die halbierte Sitzung zu beenden und zu HEAD zurückzukehren.

```
$ git bisect reset
```

Wenn Sie ein Skript haben, das nach dem Fehler suchen kann, können Sie den Prozess mit folgendem automatisieren:

```
$ git bisect run [script] [arguments]
```

Dabei ist `[script]` der Pfad zu Ihrem Skript und `[arguments]` Argumente, die an Ihr Skript übergeben werden sollen.

Wenn Sie diesen Befehl ausführen, wird automatisch die binäre Suche ausgeführt. In jedem Schritt werden `git bisect good` oder `git bisect bad` abhängig vom Exit-Code Ihres Skripts. Das Beenden mit 0 zeigt `good` , das Beenden mit 1-124, 126 oder 127 zeigt schlecht an. 125 gibt an, dass das Skript diese Version nicht testen kann (was einen `git bisect skip` auslöst).

Halbautomatisch finden Sie ein fehlerhaftes Commit

Stellen Sie sich vor, Sie befinden sich im `master` Zweig und etwas funktioniert nicht wie erwartet (eine Regression wurde eingeführt), aber Sie wissen nicht, wo. Alles, was Sie wissen, ist, dass in der letzten Version gearbeitet wurde (was z. B. mit einem Tag versehen wurde oder Sie kennen den Commit-Hash, nehmen wir hier `old-rel`).

Git hat Hilfe für Sie, um das fehlerhafte Commit zu finden, das die Regression mit einer sehr geringen Anzahl von Schritten einleitete (binäre Suche).

Beginnen Sie zunächst mit der Halbierung:

```
git bisect start master old-rel
```

Dies sagt git, dass `master` eine defekte Version ist (oder die erste defekte Version) und `old-rel` die letzte bekannte Version ist.

Git wird nun einen losgelösten Kopf in der Mitte beider Commits auschecken. Jetzt können Sie Ihre Tests durchführen. Abhängig davon, ob es funktioniert oder nicht

```
git bisect good
```

oder

```
git bisect bad
```

. Falls dieses Commit nicht getestet werden kann, können Sie es einfach `git reset` und testen. Git

kümmert sich darum.

Nach wenigen Schritten gibt git den fehlerhaften Commit-Hash aus.

Um den zweigeteilten Vorgang abzubrechen, wird nur noch ausgegeben

```
git bisect reset
```

und git stellt den vorherigen Zustand wieder her.

Halbieren / Finden fehlerhafter Commits online lesen:
https://riptutorial.com/de/git/topic/3645/halbieren---finden-fehlerhafter-commits

Kapitel 34: Historie mit Filterzweig umschreiben

Examples

Ändern Sie den Autor von Commits

Sie können einen Umgebungsfilter verwenden, um den Ersteller der Commits zu ändern. Modifizieren und exportieren Sie `$GIT_AUTHOR_NAME` im Skript, um den Autor des Commits zu ändern.

Erstellen Sie eine Datei `filter.sh` mit folgendem Inhalt:

```
if [ "$GIT_AUTHOR_NAME" = "Author to Change From" ]
then
    export GIT_AUTHOR_NAME="Author to Change To"
    export GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

Dann führe `filter-branch` von der Kommandozeile aus:

```
chmod +x ./filter.sh
git filter-branch --env-filter ./filter.sh
```

Git Committer als Commit-Autor festlegen

Dieser Befehl schreibt den Verlauf mit einem Commit-Bereich `commit1..commit2` , sodass der Git-Commit-Autor auch zum Git-Committer wird:

```
git filter-branch -f --commit-filter \
   'export GIT_COMMITTER_NAME=\"$GIT_AUTHOR_NAME\";
    export GIT_COMMITTER_EMAIL=\"$GIT_AUTHOR_EMAIL\";
    export GIT_COMMITTER_DATE=\"$GIT_AUTHOR_DATE\";
    git commit-tree $@' \
    -- commit1..commit2
```

Historie mit Filterzweig umschreiben online lesen: https://riptutorial.com/de/git/topic/2825/historie-mit-filterzweig-umschreiben

Kapitel 35: Ihr lokales und Remote-Repository aufräumen

Examples

Löschen Sie lokale Zweigstellen, die auf der Fernbedienung gelöscht wurden

Zur Fernverfolgung zwischen lokalen und gelöschten Fernzweigen verwenden

```
git fetch -p
```

Sie können dann verwenden

```
git branch -vv
```

um zu sehen, welche Äste nicht mehr verfolgt werden.

Zweige, die nicht mehr verfolgt werden, befinden sich in der untenstehenden Form und enthalten "gegangen".

```
 branch               12345e6 [origin/branch: gone] Fixed bug
```

Sie können dann eine Kombination der oben genannten Befehle verwenden und suchen, wo 'git branch -vv' den Wert 'gegangen' zurückgibt, und dann mit '-d' die Zweige löschen

```
git fetch -p && git branch -vv | awk '/: gone]/{print $1}' | xargs git branch -d
```

Ihr lokales und Remote-Repository aufräumen online lesen:
https://riptutorial.com/de/git/topic/10934/ihr-lokales-und-remote-repository-aufraumen

Kapitel 36: Inszenierung

Bemerkungen

Es ist erwähnenswert, dass das Staging wenig mit "Dateien" selbst und allem, was mit den Änderungen in jeder Datei zusammenhängt, zu tun hat. Wir inszenieren Dateien, die Änderungen enthalten, und git verfolgt die Änderungen als Commits (auch wenn die Änderungen in einem Commit in mehreren Dateien vorgenommen werden).

Die Unterscheidung zwischen Dateien und Commits mag geringfügig erscheinen, aber das Verständnis dieses Unterschieds ist für das Verständnis grundlegender Funktionen wie Kirschpick und Differenz wesentlich. (Siehe die Frustration in Kommentaren bezüglich der Komplexität einer akzeptierten Antwort, die Cherry-Pick als Dateiverwaltungswerkzeug vorschlägt .)

Was ist ein guter Ort, um Konzepte zu erklären? Ist es in Bemerkungen?

Schlüssel Konzepte:

Eine Datei ist die gebräuchlichste Metapher der beiden in der Informationstechnologie. Best Practice schreibt vor, dass sich ein Dateiname nicht ändert, wenn sich der Inhalt ändert (mit einigen wenigen Ausnahmen).

Ein Commit ist eine Metapher, die für die Quellcodeverwaltung eindeutig ist. Commits sind Änderungen, die sich auf einen bestimmten Aufwand beziehen, wie beispielsweise ein Bugfix. Commits beinhalten oft mehrere Dateien. Ein einzelner kleinerer Bugfix kann Änderungen an Vorlagen und CSS in eindeutigen Dateien beinhalten. Wenn die Änderung beschrieben, entwickelt, dokumentiert, überprüft und bereitgestellt wird, können die Änderungen in den einzelnen Dateien kommentiert und als eine Einheit behandelt werden. Die einzige Einheit ist in diesem Fall das Festschreiben. Ebenso wichtig ist, dass durch die Fokussierung auf das Commit während einer Überprüfung die unveränderten Codezeilen in den verschiedenen betroffenen Dateien sicher ignoriert werden können.

Examples

Eine einzelne Datei bereitstellen

Um eine Datei für das Festschreiben bereitzustellen, führen Sie sie aus

```
git add <filename>
```

Alle Änderungen an Dateien bereitstellen

```
git add -A
```

2,0

```
git add .
```

In Version 2.x `git add .` führt alle Änderungen an den Dateien im aktuellen Verzeichnis und allen seinen Unterverzeichnissen aus. In 1.x werden jedoch nur neue und geänderte Dateien bereitgestellt, keine gelöschten Dateien .

Verwenden Sie `git add -A` oder den entsprechenden Befehl `git add --all` , um alle Änderungen an Dateien in einer beliebigen Version von git `git add --all` .

Bühne gelöschte Dateien

```
git rm filename
```

Um die Datei aus Git zu löschen, ohne sie von der Festplatte zu entfernen, verwenden Sie das Flag `--cached`

```
git rm --cached filename
```

Machen Sie eine Datei bereit, die Änderungen enthält

```
git reset <filePath>
```

Interaktives Hinzufügen

`git add -i` (oder `--interactive`) erhalten Sie eine interaktive Schnittstelle, auf der Sie den Index bearbeiten können, um die `--interactive` für das nächste Commit vorzubereiten. Sie können Änderungen an ganzen Dateien hinzufügen und entfernen, nicht protokollierte Dateien hinzufügen und Dateien aus der Nachverfolgung entfernen, aber auch einen Unterabschnitt der Änderungen auswählen, die in den Index aufgenommen werden sollen, indem Sie die hinzuzufügenden Änderungen auswählen, diese teilen oder sogar den Unterschied bearbeiten . Viele grafische Commit-Tools für Git (wie z. B. `git gui`) enthalten eine solche Funktion. Dies ist möglicherweise einfacher zu verwenden als die Befehlszeilenversion.

Es ist sehr nützlich (1), wenn Sie Änderungen im Arbeitsverzeichnis, die Sie in separate Commits einfügen möchten, verwickelt haben, und nicht alle in einem einzigen Commit (2), wenn Sie sich in einer interaktiven Datenbank befinden und auch teilen möchten großes Engagement.

```
$ git add -i
           staged     unstaged path
  1:    unchanged        +4/-4 index.js
  2:        +1/-0      nothing package.json

*** Commands ***
  1: status       2: update       3: revert       4: add untracked
  5: patch        6: diff         7: quit         8: help
What now>
```

In der oberen Hälfte dieser Ausgabe wird der aktuelle Status des Indexes in gestaffelte und nicht

bereitgestellte Spalten angezeigt:

1. `index.js` 4 Zeilen hinzugefügt und 4 Zeilen entfernt. Es wird derzeit nicht inszeniert, da der aktuelle Status "unverändert" anzeigt. Wenn diese Datei bereitgestellt wird, wird das `+4/-4` Bit in die bereitgestellte Spalte übertragen, und die nicht bereitgestellte Spalte liest "nichts".
2. `package.json` wurde um eine Zeile erweitert und inszeniert. Es gibt keine weiteren Änderungen, da es wie in der Zeile "Nichts" unter der Spalte "Nicht bereitgestellt" angezeigt wurde.

Die untere Hälfte zeigt, was Sie tun können. Geben Sie entweder eine Zahl (1-8) oder einen Buchstaben (`s` , `u` , `r` , `a` , `p` , `d` , `q` , `h`) ein.

`status` zeigt die Ausgabe identisch mit dem oberen Teil der Ausgabe.

`update` können Sie weitere Änderungen an den bereitgestellten Commits mit zusätzlicher Syntax vornehmen.

`revert` die bereitgestellten Commit-Informationen wieder auf HEAD zurück.

`add untracked` untracked `add untracked` können Sie Dateipfade hinzufügen, die zuvor von der Versionskontrolle nicht erfasst wurden.

`patch` kann ein Pfad aus einer Ausgabe ähnlich dem `status` für die weitere Analyse ausgewählt werden.

`diff` zeigt an, was begangen wird.

`quit` den Befehl.

`help` bietet weitere Hilfe zur Verwendung dieses Befehls.

Änderungen nach Stück hinzufügen

Sie können sehen, welche "Hunks" der Arbeit mit dem Patch-Flag zum Festschreiben bereitgestellt werden:

```
git add -p
```

oder

```
git add --patch
```

Daraufhin wird eine interaktive Eingabeaufforderung geöffnet, in der Sie die Unterschiede betrachten und entscheiden können, ob Sie sie einschließen möchten oder nicht.

```
Stage this hunk [y,n,q,a,d,/,s,e,?]?
```

- `y` Bühne dieses Stück für das nächste Commit
- `n` inszenieren Sie diesen Hunk nicht für das nächste Commit

- `q` quittieren; Inszenieren Sie diesen Kerl oder einen der verbleibenden Kerle nicht
- `eine` Bühne dieses Stück und alle späteren Teile in der Datei
- `Stellen` Sie dieses Hunk oder eines der späteren Hunks in der Datei nicht bereit
- `g` Wählen Sie einen Hunk aus, zu dem Sie gehen möchten
- `/` Suche nach einem Hunk, der der angegebenen Regex entspricht
- `Ich` lasse dieses Stück unentschieden, siehe das nächste unentschiedene Stück
- `Ich` lasse dieses Stück unentschieden, siehe nächstes Stück
- `Ich` lasse dieses Stück unentschieden, siehe vorheriges unentschlossenes Stück
- `Ich` lasse dieses Stück unentschieden, siehe vorheriges Stück
- `s` spaltet das aktuelle Hunk in kleinere Hunks auf
- `e` das aktuelle Stück manuell bearbeiten
- `?` Hilfe zum Drucken von Brocken

Dies macht es einfach, Änderungen zu erfassen, die Sie nicht festlegen möchten.

Sie können dies auch über `git add --interactive` und Auswahl von `p` öffnen.

Inszenierte Änderungen anzeigen

So zeigen Sie die Hunks an, die zum Festschreiben bereitgestellt werden

```
git diff --cached
```

Inszenierung online lesen: https://riptutorial.com/de/git/topic/244/inszenierung

Kapitel 37: Interne

Examples

Repo

Ein `git repository` ist eine Datenstruktur auf der Festplatte, in der Metadaten für eine Reihe von Dateien und Verzeichnissen gespeichert werden.

Es befindet sich im `.git/` Ihres Projekts. Jedes Mal, wenn Sie Daten an git übergeben, werden diese hier gespeichert. Umgekehrt enthält `.git/` jeden einzelnen Commit.

Ihre Grundstruktur sieht folgendermaßen aus:

```
.git/
    objects/
    refs/
```

Objekte

`git` ist grundsätzlich ein Schlüsselwertspeicher. Wenn Sie Daten zu `git` hinzufügen, wird ein `object` und der SHA-1-Hash des `object` als Schlüssel verwendet.

Daher kann jeder Inhalt in `git` anhand seines Hashes nachgeschlagen werden:

```
git cat-file -p 4bb6f98
```

Es gibt 4 `Object` :

- `blob`
- `tree`
- `commit`
- `tag`

HEAD ref

`HEAD` ist ein Sonder `ref` . Es zeigt immer auf das aktuelle Objekt.

Sie können sehen, wohin es aktuell zeigt, indem Sie die `.git/HEAD` Datei überprüfen.

Normalerweise zeigt `HEAD` auf einen anderen `ref` :

```
$cat .git/HEAD
ref: refs/heads/mainline
```

Es kann aber auch direkt auf ein `object` :

```
$ cat .git/HEAD
```

```
4bb6f98a223abc9345a0cef9200562333
```

Dies ist, was als "losgelöster Kopf" bekannt ist - weil `HEAD` nicht mit einem `ref` , sondern auf ein `object` .

Refs

Ein `ref` ist im Wesentlichen ein Zeiger. Es ist ein Name, der auf ein `object` verweist. Zum Beispiel,

```
"master" --> 1a410e...
```

Sie werden in .git / refs / heads / in Klartextdateien gespeichert.

```
$ cat .git/refs/heads/mainline
4bb6f98a223abc9345a0cef9200562333
```

Dies wird üblicherweise als `branches` . Sie ist jedoch zu beachten , dass in `git` so etwas wie einen gibt es kein `branch` - nur `ref` .

Nun ist es möglich, `git` rein zu navigieren, indem Sie direkt durch ihre Hashwerte zu verschiedenen `objects` springen. Dies wäre jedoch äußerst unpraktisch. Ein `ref` gibt Ihnen einen bequemen Namen, um auf `objects` zu verweisen. Es ist viel einfacher, `git` zu fragen, ob er an einen bestimmten Ort geht und nicht per Hash.

Commit-Objekt

Ein `commit` ist wahrscheinlich der `object` der für `git` Benutzer am bekanntesten ist, da er üblicherweise mit den `git commit` Befehlen erstellt wird.

Der `commit` enthält jedoch keine geänderten Dateien oder Daten. Sie enthält vielmehr meist Metadaten und Verweise auf andere `objects` , die den eigentlichen Inhalt des `commit` .

Ein `commit` enthält einige Dinge:

- Hasch eines `tree`
- Hash eines übergeordneten `commit`
- Name des Autors / E-Mail, Name des Commiters / E-Mail
- Nachricht übermitteln

Sie können den Inhalt jedes Commits folgendermaßen sehen:

```
$ git cat-file commit 5bac93
tree 04d1daef...
parent b7850ef5...
author Geddy Lee <glee@rush.com>
commiter Neil Peart <npeart@rush.com>

First commit!
```

Baum

Ein sehr wichtiger Hinweis ist , dass der `tree` speichert jede Datei in Ihrem Projekt - Objekte und speichert ganze Dateien nicht diffs. Dies bedeutet, dass jeder `commit` eine Momentaufnahme des gesamten Projekts enthält *.

** Technisch werden nur geänderte Dateien gespeichert. Dies ist jedoch eher ein Implementierungsdetail für die Effizienz. Aus `commit` sollte ein `commit` als vollständige Kopie des Projekts betrachtet werden .*

Elternteil

Die `parent` Zeile enthält einen Hash eines anderen `commit` und kann als "übergeordneter Zeiger" betrachtet werden, der auf das "vorherige Festschreiben" verweist. Dies bildet implizit ein **Commit-Diagramm,** das als **Commit-Diagramm bezeichnet wird** . Insbesondere handelt es sich um eine gerichtete azyklische Grafik (DAG).

Baumobjekt

Eine `tree` stellt im Wesentlichen einen Ordner in einem herkömmlichen Dateisystem dar: verschachtelte Container für Dateien oder andere Ordner.

Ein `tree` enthält:

- 0 oder mehr `blob` Objekte
- 0 oder mehr `tree` - Objekte

Genauso wie Sie mit `ls` oder `dir` den Inhalt eines Ordners auflisten können, können Sie den Inhalt eines `tree` auflisten.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
040000 tree 92e1ca7e   src
...
```

Sie können die Dateien in einem `commit` nachschlagen `commit` indem Sie zuerst den Hash des `tree` im `commit` suchen und dann diesen `tree` :

```
$ git cat-file commit 4bb6f93a
tree 07b1a631
parent ...
author ...
commiter ...

$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
```

```
040000 tree 92e1ca7e   src
...
```

Blob-Objekt

Ein `blob` enthält beliebige binäre Dateiinhalte. Normalerweise handelt es sich dabei um unformatierten Text wie Quellcode oder Blogartikel. Es können aber genauso einfach die Bytes einer PNG-Datei oder etwas anderes sein.

Wenn Sie den Hash eines `blob` , können Sie dessen Inhalt betrachten.

```
$ git cat-file -p d429810
package com.example.project

class Foo {
 ...
}
...
```

Sie können zum Beispiel einen `tree` wie oben durchsuchen und dann einen der `blobs` darin betrachten.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b   .gitignore
100644 blob cc0956f1   Makefile
040000 tree 92e1ca7e   src
100644 blob cae391ff   Readme.txt

$ git cat-file -p cae391ff
Welcome to my project! This is the readmefile
...
```

Neue Commits erstellen

Der Befehl `git commit` führt einige Dinge aus:

1. Erstellen Sie `blobs` und `trees` , um Ihr Projektverzeichnis darzustellen - in `.git/objects` gespeichert
2. Erstellt ein neues `commit` Objekt mit Ihren Autorinformationen, der Commit-Nachricht und dem `tree` aus Schritt 1 - ebenfalls in `.git/objects` gespeichert
3. Aktualisiert den `HEAD` ref in `.git/HEAD` mit dem Hash des neu erstellten `commit`

Dies führt dazu, dass ein neuer Snapshot Ihres Projekts zu `git` hinzugefügt wird, der mit dem vorherigen Status verbunden ist.

KOPF bewegen

Wenn Sie `git checkout` für ein Commit ausführen (angegeben durch Hash oder Ref), teilen Sie `git` mit, dass Ihr Arbeitsverzeichnis so aussehen soll, wie es bei der Momentaufnahme gemacht wurde.

1. Aktualisieren Sie die Dateien im Arbeitsverzeichnis so, dass sie der `tree` im `commit`
2. Aktualisieren Sie `HEAD` , dass es auf den angegebenen Hash oder Ref verweist

Refs bewegen

Das Ausführen von `git reset --hard` verschiebt die Refs in den angegebenen Hash / Ref.

`MyBranch` auf `b8dc53` :

```
$ git checkout MyBranch      # moves HEAD to MyBranch
$ git reset --hard b8dc53    # makes MyBranch point to b8dc53
```

Neue Refs erstellen

`git checkout -b <refname>` wird ein neuer Ref erstellt, der auf das aktuelle `commit` verweist.

```
$ cat .git/head
1f324a

$ git checkout -b TestBranch

$ cat .git/refs/heads/TestBranch
1f324a
```

Interne online lesen: https://riptutorial.com/de/git/topic/2637/interne

Kapitel 38: Leere Verzeichnisse in Git

Examples

Git verfolgt keine Verzeichnisse

Angenommen, Sie haben ein Projekt mit der folgenden Verzeichnisstruktur initialisiert:

```
/build
app.js
```

Dann fügen Sie alles hinzu, was Sie bisher erstellt haben, und legen Sie fest:

```
git init
git add .
git commit -m "Initial commit"
```

Git verfolgt nur die Datei app.js.

Angenommen , Sie einen Build - Schritt auf Ihre Bewerbung und stützen sich auf die „bauen“ Verzeichnis , dort zu sein als das Ausgabeverzeichnis hinzugefügt (und Sie nicht möchten , dass es sich um eine Setup - Anweisung jeder Entwickler machen muss folgen), ist eine *Konvention* ein einzubeziehen ".gitkeep" -Datei im Verzeichnis und lassen Sie Git diese Datei verfolgen.

```
/build
  .gitkeep
app.js
```

Dann füge diese neue Datei hinzu:

```
git add build/.gitkeep
git commit -m "Keep the build directory around"
```

Git verfolgt nun die Datei build / .gitkeep, und der Build-Ordner wird an der Kasse zur Verfügung gestellt.

Auch dies ist nur eine Konvention und keine Git-Funktion.

Leere Verzeichnisse in Git online lesen: https://riptutorial.com/de/git/topic/2680/leere-verzeichnisse-in-git

Kapitel 39: Mischkonflikte lösen

Examples

Manuelle Auflösung

Beim Durchführen einer `git merge` kann es vorkommen, dass git einen Fehler "Zusammenführungskonflikt" meldet. Es wird Ihnen gemeldet, welche Dateien Konflikte aufweisen, und Sie müssen die Konflikte lösen.

Ein `git status` an einem beliebigen Punkt hilft Ihnen zu sehen, was noch bearbeitet werden muss, mit einer hilfreichen Nachricht wie

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")

Unmerged paths:
  (use "git add <file>..." to mark resolution)

    both modified:      index.html

no changes added to commit (use "git add" and/or "git commit -a")
```

Git hinterlässt Markierungen in den Dateien, um Ihnen zu sagen, wo der Konflikt entstanden ist:

```
<<<<<<<<< HEAD: index.html #indicates the state of your current branch
<div id="footer">contact : email@somedomain.com</div>
========= #indicates break between conflicts
<div id="footer">
please contact us at email@somedomain.com
</div>
>>>>>>>>> iss2: index.html #indicates the state of the other branch (iss2)
```

Um die Konflikte aufzulösen, müssen Sie den Bereich zwischen den Markierungen <<<<<< und >>>>>>> entsprechend bearbeiten und die Statuszeilen entfernen (<<<<<<<, >>>>>) >> und ======== Zeilen) vollständig. Dann `git add index.html` , um es als gelöst zu markieren, und `git commit` , um die Zusammenführung zu beenden.

Mischkonflikte lösen online lesen: https://riptutorial.com/de/git/topic/3233/mischkonflikte-losen

Kapitel 40: Mit Remotes arbeiten

Syntax

- `git remote [-v | --verbose]`
- `git remote add [-t <branch>] [-m <master>] [-f] [--[no-]tags] [--mirror=<fetch|push>] <name> <url>`
- `git remote rename <old> <new>`
- `git remote remove <name>`
- `git remote set-head <name> (-a | --auto | -d | --delete | <branch>)`
- `git remote set-branches [--add] <name> <branch>…`
- `git remote get-url [--push] [--all] <name>`
- `git remote set-url [--push] <name> <newurl> [<oldurl>]`
- `git remote set-url --add [--push] <name> <newurl>`
- `git remote set-url --delete [--push] <name> <url>`
- `git remote [-v | --verbose] show [-n] <name>…`
- `git remote prune [-n | --dry-run] <name>…`
- `git remote [-v | --verbose] update [-p | --prune] [(<group> | <remote>)…]`

Examples

Neues Remote-Repository hinzufügen

```
git remote add upstream git-repository-url
```

Fügt dem durch " `git-repository-url` neuen `git-repository-url` Namen "new" mit dem Namen " `upstream` zum git-Repository hinzu

Aktualisierung aus dem Upstream-Repository

Vorausgesetzt, Sie setzen den Upstream (wie im Abschnitt "Ein Upstream-Repository festlegen").

```
git fetch remote-name
git merge remote-name/branch-name
```

Der `pull` Befehl kombiniert einen `fetch` und eine `merge` .

```
git pull
```

Der Befehl zum `pull` mit `--rebase` flag kombiniert einen `fetch` und eine `rebase` anstelle von `merge` .

```
git pull --rebase remote-name branch-name
```

ls-remote

`git ls-remote` ist ein eindeutiger Befehl, mit dem Sie ein Remote-Repo abfragen können, *ohne es vorher zu klonen / abzurufen* .

Es werden Refs / Heads und Refs / Tags des Remote-Repos aufgelistet.

Manchmal sehen Sie `refs/tags/v0.1.6` *und* `refs/tags/v0.1.6^{}` : das `^{}` um das dereferenzierte annotierte Tag `refs/tags/v0.1.6^{}` (dh das Commit, auf das das Tag zeigt)

Seit git 2.8 (März 2016) können Sie diesen doppelten Eintrag für ein Tag vermeiden und diese dereferenced Tags direkt auflisten mit:

```
git ls-remote --ref
```

Es kann auch helfen, die tatsächliche URL, die von einem Remote-Repo verwendet wird, aufzulösen, wenn Sie die `url.<base>.insteadOf` " `url.<base>.insteadOf` " haben.
Wenn `git remote --get-url <aremotename>` https://server.com/user/repo zurückgibt, haben Sie `git config url.ssh://git@server.com:.insteadOf https://server.com/` :

```
git ls-remote --get-url <aremotename>
ssh://git@server.com:user/repo
```

Löschen einer Remote Branch

So löschen Sie einen Remote-Zweig in Git:

```
git push [remote-name] --delete [branch-name]
```

oder

```
git push [remote-name] :[branch-name]
```

Lokale Kopien von gelöschten Remote-Filialen entfernen

Wenn eine entfernte Verzweigung gelöscht wurde, muss Ihr lokales Repository aufgefordert werden, die Referenz darauf zu beschneiden.

Löschen Sie gelöschte Zweige von einer bestimmten Fernbedienung aus:

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

Gelöschte Zweige aus *allen* Fernbedienungen beschneiden:

```
git fetch --all --prune
```

Zeigt Informationen zu einer bestimmten Fernbedienung an

Informationen zu einer bekannten Fernbedienung ausgeben: `origin`

```
git remote show origin
```

Nur die URL der Fernbedienung drucken:

```
git config --get remote.origin.url
```

Mit 2.7+ ist es auch möglich, was besser ist als das oben genannte, das den Befehl `config` .

```
git remote get-url origin
```

Vorhandene Remotes auflisten

Listen Sie alle vorhandenen Remotes auf, die mit diesem Repository verbunden sind:

```
git remote
```

Listen Sie alle vorhandenen Remotes, die diesem Repository zugeordnet sind, detailliert auf, einschließlich der `fetch` und `push` URLs:

```
git remote --verbose
```

oder einfach

```
git remote -v
```

Fertig machen

Syntax für das Pushing an einen entfernten Zweig

```
git push <remote_name> <branch_name>
```

Beispiel

```
git push origin master
```

Setze Upstream in einem neuen Zweig

Sie können einen neuen Zweig erstellen und mit wechseln

```
git checkout -b AP-57
```

Nachdem Sie mit git checkout einen neuen Zweig erstellt haben, müssen Sie diesen Ursprungsursprung für die Verwendung festlegen

```
git push --set-upstream origin AP-57
```

Danach können Sie git push verwenden, während Sie sich in diesem Zweig befinden.

Remote-Repository ändern

Um die URL des Repositorys zu ändern, auf das Ihre Remote `set-url` verweisen soll, können Sie die `set-url` Option wie folgt verwenden:

```
git remote set-url <remote_name> <remote_repository_url>
```

Beispiel:

```
git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git
```

Git Remote URL ändern

Vorhandene Fernbedienung prüfen

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
# origin https://github.com/usernam/repo.git (push)
```

Repository-URL ändern

```
git remote set-url origin https://github.com/username/repo2.git
# Change the 'origin' remote's URL
```

Überprüfen Sie die neue Remote-URL

```
git remote -v
# origin  https://github.com/username/repo2.git (fetch)
# origin  https://github.com/username/repo2.git (push)
```

Umbenennen einer Fernbedienung

Verwenden Sie zum Umbenennen der Fernbedienung den Befehl `git remote rename`

Der Befehl `git remote rename` erfordert zwei Argumente:

- Ein vorhandener Remote-Name, zum Beispiel: **Ursprung**
- Ein neuer Name für die Fernbedienung, zum Beispiel: **destination**

Bestehenden Remote-Namen abrufen

```
git remote
# origin
```

Vorhandene Remote mit URL überprüfen

```
git remote -v
# origin https://github.com/username/repo.git (fetch)
```

```
# origin https://github.com/usernam/repo.git (push)
```

Remote umbenennen

```
 git remote rename origin destination
 # Change remote name from 'origin' to 'destination'
```

Überprüfen Sie den neuen Namen

```
git remote -v
# destination https://github.com/username/repo.git (fetch)
# destination https://github.com/usernam/repo.git (push)
```

=== Mögliche Fehler ===

1. Konfig-Abschnitt 'remote. [Alter Name]' konnte nicht in 'remote. [Neuer Name]'

   Dieser Fehler bedeutet, dass die Fernbedienung, die Sie mit dem alten Namen (**Ursprung**) versucht haben, nicht vorhanden ist.

2. Remote [neuer Name] ist bereits vorhanden.

   Fehlermeldung ist selbsterklärend.

Legen Sie die URL für eine bestimmte Fernbedienung fest

Sie können die URL einer vorhandenen Fernbedienung mit dem Befehl ändern

```
git remote set-url remote-name url
```

Rufen Sie die URL für eine bestimmte Fernbedienung ab

Sie können die URL für eine vorhandene Remote-Einheit mit dem Befehl erhalten

```
git remote get-url <name>
```

Standardmäßig ist dies der Fall

```
git remote get-url origin
```

Mit Remotes arbeiten online lesen: https://riptutorial.com/de/git/topic/243/mit-remotes-arbeiten

Kapitel 41: Neueinstellung

Syntax

- `git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] [<upstream>] [<branch>]`
- `git rebase [-i | --interactive] [options] [--exec <cmd>] [--onto <newbase>] --root [<branch>]`
- `git rebase --continue | --skip | --abort | --edit-todo`

Parameter

Parameter	Einzelheiten
--fortsetzen	Starten Sie den Umbasierungsprozess neu, nachdem Sie einen Zusammenführungskonflikt gelöst haben.
--abbrechen	Brechen Sie den Rebase-Vorgang ab und setzen Sie HEAD auf den ursprünglichen Zweig zurück. Wurde beim Start der Rebase-Operation eine Verzweigung bereitgestellt, wird HEAD auf Verzweigung zurückgesetzt. Andernfalls wird HEAD auf die Position zurückgesetzt, an der es sich befand, als der Rebase-Vorgang gestartet wurde.
--keep-empty	Behalten Sie die Commits, die nichts an ihren Eltern ändern, im Ergebnis.
--überspringen	Starten Sie den Rebasierungsprozess neu, indem Sie den aktuellen Patch überspringen.
-m, --merge	Verwenden Sie zum Zusammenfassen Strategien zum Zusammenführen. Wenn die rekursive (Standard-) Zusammenführungsstrategie verwendet wird, kann rebase auf der Upstream-Seite Umbenennungen erkennen. Beachten Sie, dass eine Rebase-Zusammenführung funktioniert, indem jedes Commit aus dem Arbeitszweig oberhalb des Upstream-Zweigs wiedergegeben wird. Aus diesem Grund wird bei einem Verschmelzungskonflikt die als überlieferte Seite die bisher rebasierte Serie, beginnend mit Upstream, und ihre ist der Arbeitszweig. Mit anderen Worten, die Seiten sind vertauscht.
--stat	Zeigt eine Statistik an, was sich seit der letzten Neubasis geändert hat. Der diffstat wird auch von der Konfigurationsoption rebase.stat gesteuert.
-x, --exec `command`	Führen Sie interaktive rebase, Stoppen zwischen jedem Commit und Ausführen - `command`

Bemerkungen

Bitte beachten Sie, dass rebase den Repository-Verlauf effektiv umschreibt.

Durch das erneuern von Festschreibungen, die im Remote-Repository vorhanden sind, könnten Repository-Knoten, die von anderen Entwicklern als Basisknoten für ihre Entwicklungen verwendet werden, neu geschrieben werden. Wenn Sie nicht wirklich wissen, was Sie tun, ist es eine bewährte Methode, sich zu vergewissern, bevor Sie Ihre Änderungen übernehmen.

Examples

Lokale Niederlassung neu einkaufen

Bei der Neuveränderung werden eine Reihe von Commits auf ein anderes Commit **angewendet** .

Um eine Verzweigung neu zu `rebase` , `rebase` Sie die Verzweigung aus und `rebase` sie dann auf einer anderen Verzweigung fest.

```
git checkout topic
git rebase master  # rebase current branch onto master branch
```

Dies würde verursachen:

```
      A---B---C topic
     /
D---E---F---G master
```

Zu etwas werden:

```
              A'--B'--C' topic
             /
D---E---F---G master
```

Diese Operationen können in einem einzigen Befehl kombiniert werden, der die Verzweigung auscheckt und sie sofort wieder herstellt:

```
git rebase master topic   # rebase topic branch onto master branch
```

Wichtig: Nach der Erneuerung haben die angewendeten Commits einen anderen Hash. Sie sollten Commits nicht zurückweisen, die Sie bereits an einen Remote-Host gesendet haben. Eine Folge davon kann sein, dass Sie nicht in der Lage sind `git push` Ihren lokalen Zweig auf einen Remote-Host zu verschieben, wodurch Ihre einzige Option für `git push --force` .

Rebase: unsere und ihre, lokal und entfernt

Ein Rebase wechselt die Bedeutung von "uns" und "ihrer":

```
git checkout topic
git rebase   master    # rebase topic branch on top of master branch
```

Was auch immer HEAD darauf zeigt, ist "unser"

Das erste, was eine Rebase tut, ist das Zurücksetzen des `HEAD` auf `master` ; vor Rosinenpickerei aus dem alten Zweig begeht `topic` zu einem neuen (Commit jeder im ehemaligen `topic` Zweig neu geschrieben wird und wird von einem anderen Hash identifiziert werden).

In Bezug auf von Zusammenführungswerkzeugen verwendete Terminologien (nicht mit lokalem oder Remote-Ref zu verwechseln)

```
=> local is master ("ours"),
=> remote is topic ("theirs")
```

Das bedeutet, dass ein Zusammenführungs / Diff-Werkzeug den Upstream-Zweig als `local` (`master` : der Zweig, auf dem Sie die Basenliste neu definieren) und den Arbeitszweig als `remote` (`topic` : Die Zweigniederlassung, die neu basiert wird) darstellt.

```
+---------------------------------------+
| LOCAL:master |    BASE    | REMOTE:topic |
+---------------------------------------+
|             MERGED                    |
+---------------------------------------+
```

Inversion dargestellt

Beim Zusammenführen:

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
    \
     \
      \--y--y--y <- other branch to merge
```

Wir ändern das aktuelle `topic` , also haben wir immer noch das, woran wir gearbeitet haben (und wir verschmelzen aus einem anderen Zweig).

```
c--c--x--x--x---------o(*)  MERGE, still on branch topic
    \       ^        /
     \     ours     /
      \             /
       --y--y--y--/
              ^
             theirs
```

Auf einer rebase:

Aber **bei einer Rebase** wechseln wir die Seiten, denn als Erstes überprüft eine Rebase den Upstream-Zweig, um die aktuellen Commits darüber wiederzugeben!

```
c--c--x--x--x(*) <- current branch topic ('*'=HEAD)
    \
```

```
     \
      \--y--y--y <- upstream branch
```

Ein **`git rebase upstream`** setzt `HEAD` auf den Upstream-Zweig, daher wird der Wechsel von "unserem" und "ihnen" im Vergleich zum vorherigen "aktuellen" Arbeitszweig vorgenommen.

```
c--c--x--x--x <- former "current" branch, new "theirs"
    \
     \
      \--y--y--y(*) <- set HEAD to this commit, to replay x's on it
                ^       this will be the new "ours"
                |
           upstream
```

Das Fütterungsmaterial wird dann wiederholen ‚ihr' verpflichtet auf dem neuen ‚unserem' `topic` Zweig:

```
c--c..x..x..x <- old "theirs" commits, now "ghosts", available through "reflogs"
    \
     \
      \--y--y--y--x'--x'--x'(*) <- topic  once all x's are replayed,
                ^                        point branch topic to this commit
                |
        upstream branch
```

Interaktiver Rebase

Dieses Beispiel soll beschreiben, wie man `git rebase` im interaktiven Modus nutzen kann. Es wird erwartet, dass man ein grundlegendes Verständnis davon hat, was `git rebase` ist und was es tut.

Interaktiver Rebase wird mit folgendem Befehl initiiert:

```
git rebase -i
```

Die Option `-i` bezieht sich auf den *interaktiven Modus* . Mit interaktiver Rebase kann der Benutzer Festschreibungsnachrichten ändern sowie Neuanordnungen, Split- und / oder Squash-Commits (zu einer Kombination kombinieren).

Angenommen, Sie möchten Ihre letzten drei Commits neu anordnen. Dazu können Sie Folgendes ausführen:

```
git rebase -i HEAD~3
```

Nachdem Sie die obige Anweisung ausgeführt haben, wird eine Datei in Ihrem Texteditor geöffnet, in der Sie auswählen können, wie Ihre Commits neu festgelegt werden. Ändern Sie in diesem Beispiel einfach die Reihenfolge Ihrer Commits, speichern Sie die Datei und schließen Sie den Editor. Dadurch wird in der von Ihnen angewendeten Reihenfolge eine Neugestaltung eingeleitet. Wenn Sie das `git log` überprüfen, werden Ihre Commits in der von Ihnen angegebenen neuen Reihenfolge angezeigt.

Überarbeitung von Commit-Nachrichten

Nun haben Sie entschieden, dass eine der Commit-Nachrichten vage ist, und Sie möchten, dass sie aussagekräftiger ist. Sehen wir uns die letzten drei Commits mit demselben Befehl an.

```
git rebase -i HEAD~3
```

Anstatt die Reihenfolge neu zu ordnen, werden die Commits umbasiert. Diesmal ändern wir `pick` , den Standard, um ein Commit neu zu `reword` an dem Sie die Nachricht ändern möchten.

Wenn Sie den Editor schließen, wird die Rebase initiiert und bei der spezifischen Commit-Nachricht angehalten, die Sie umformulieren wollten. Auf diese Weise können Sie die Festschreibungsnachricht nach Wunsch ändern. Nachdem Sie die Nachricht geändert haben, schließen Sie einfach den Editor, um fortzufahren.

Den Inhalt eines Commits ändern

Neben dem Ändern der Festschreibungsnachricht können Sie auch die durch das Festschreiben vorgenommenen Änderungen anpassen. Ändern Sie dazu einfach die `pick` , `edit` einen Commit zu `edit` . Git stoppt, wenn es bei diesem Commit ankommt, und stellt die ursprünglichen Änderungen des Commits im Staging-Bereich bereit. Sie können diese Änderungen jetzt anpassen, indem Sie sie deaktivieren oder neue Änderungen hinzufügen.

Sobald der Bereitstellungsbereich alle Änderungen enthält, die Sie in diesem Commit wünschen, bestätigen Sie die Änderungen. Die alte Commit-Nachricht wird angezeigt und kann an das neue Commit angepasst werden.

Aufteilen eines einzelnen Commits in mehrere

Angenommen, Sie haben ein Commit vorgenommen, zu einem späteren Zeitpunkt jedoch entschieden, dass dieses Commit in zwei oder mehr Commits aufgeteilt werden kann. Verwenden Sie den gleichen Befehl wie zuvor, ersetzen Sie `pick` stattdessen mit `edit` und drücken Sie die Eingabetaste.

Nun stoppt git bei dem Commit, das Sie für die Bearbeitung markiert haben, und legt den gesamten Inhalt in den Staging-Bereich. Ab diesem Zeitpunkt können Sie `git reset HEAD^` ausführen, um das Commit in Ihrem Arbeitsverzeichnis abzulegen. Anschließend können Sie Ihre Dateien in einer anderen Reihenfolge hinzufügen und festschreiben - letztendlich teilen Sie ein einzelnes Commit in *n* Commits auf.

Mehrere Commits in einem komprimieren

Angenommen, Sie haben einige Arbeit geleistet und haben mehrere Commits, die Ihrer Meinung nach ein einziges Commit sein könnten. Dafür können Sie `git rebase -i HEAD~3` ausführen, wobei `3` durch eine entsprechende Menge an Commits ersetzt wird.

Ersetzen Sie dieses Mal den `pick` durch einen `squash` . Während der Rebase wird das Commit, das Sie zum Squashing angewiesen haben, zusätzlich zum vorherigen Commit gequetscht. verwandeln sie stattdessen in ein einziges Commit.

Abbruch einer interaktiven Neubasis

Sie haben eine interaktive Datenbank gestartet. In dem Editor, in dem Sie Ihre Commits auswählen, entscheiden Sie, dass etwas nicht funktioniert (beispielsweise fehlt ein Commit oder Sie wählen das falsche Rebase-Ziel), und Sie möchten die Rebase abbrechen.

Löschen Sie dazu einfach alle Commits und Aktionen (dh alle Zeilen, die nicht mit dem `#` -Zeichen beginnen), und die Rebase wird abgebrochen!

Der Hilfetext im Editor enthält eigentlich diesen Hinweis:

```
# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#          ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
# Note that empty commits are commented out
```

Nach einer Rebase drücken

Manchmal müssen Sie den Verlauf mit einer Rebase neu schreiben, aber `git push` beschwert sich darüber, weil Sie den Verlauf neu geschrieben haben.

Dies kann mit `git push --force` gelöst werden. Berücksichtigen Sie jedoch `git push --force-with-lease` , um anzugeben, dass der Push fehlschlagen soll, wenn sich der lokale Remote-Tracking-Zweig von dem Zweig auf der Remote- `git push --force-with-lease` unterscheidet, z. B. jemandem sonst nach dem letzten Abruf auf die Fernbedienung gedrückt. Dadurch wird vermieden, dass der letzte Push einer anderen Person versehentlich überschrieben wird.

Hinweis : `git push --force` - und sogar `--force-with-lease` für diese Angelegenheit - kann ein gefährlicher Befehl sein, da er die Historie des Zweigs neu schreibt. Wenn eine andere Person den Zweig vor dem erzwungenen Schub gezogen hatte, werden Fehler in seinem `git pull` oder `git fetch` , da der lokale Verlauf und der Remote-Verlauf voneinander abweichen. Dies kann zu unerwarteten Fehlern der Person führen. Bei ausreichender Betrachtung der Wiederherstellungsprotokolle kann die Arbeit des anderen Benutzers wiederhergestellt werden, dies kann jedoch zu viel Zeitverschwendung führen. Wenn Sie einen Zweig mit anderen Mitwirkenden zwingen müssen, versuchen Sie, sich mit ihnen zu koordinieren, damit sie sich nicht mit Fehlern befassen müssen.

Zurück zum ursprünglichen Commit

Seit Git 1.7.12 ist es möglich, bis zum Root-Commit zu rebasieren. Das Root-Commit ist das erste Commit, das jemals in einem Repository ausgeführt wurde, und kann normalerweise nicht bearbeitet werden. Verwenden Sie den folgenden Befehl:

```
git rebase -i --root
```

Neuerstellung vor einer Codeüberprüfung

Zusammenfassung

Dieses Ziel besteht darin, alle Ihre verstreuten Commits in aussagekräftigere Commits zu reorganisieren, um Codeüberprüfungen zu vereinfachen. Wenn zu viele Ebenen auf einmal zu viele Änderungen gleichzeitig vorgenommen werden, ist es schwieriger, eine Codeüberprüfung durchzuführen. Wenn Sie Ihre chronologisch erstellten Commits in aktuelle Commits reorganisieren können, ist der Codeüberprüfungsprozess einfacher (und möglicherweise fallen weniger Fehler durch den Codeüberprüfungsprozess).

Dieses übermäßig vereinfachte Beispiel ist nicht die einzige Strategie, mit der git bessere Codeüberprüfungen durchführt. Es ist die Art und Weise, wie ich es mache, und es ist etwas, das andere dazu anregt, darüber nachzudenken, wie man Code-Reviews und Git-Geschichte einfacher / besser macht.

Dies zeigt auch pädagogisch die Kraft der Rebase im Allgemeinen.

In diesem Beispiel wird davon ausgegangen, dass Sie über interaktive Umbasierungen Bescheid wissen.

Vorausgesetzt:

- Sie arbeiten an einem Feature-Zweig von Master
- Ihr Feature verfügt über drei Hauptschichten: Frontend, Backend und DB
- Sie haben eine Menge Commits gemacht, während Sie an einem Funktionszweig gearbeitet

haben. Jedes Commit berührt mehrere Ebenen gleichzeitig
- Sie wollen (am Ende) nur drei Commits in Ihrer Branche
    - eine, die alle Frontend-Änderungen enthält
    - eine, die alle Änderungen am hinteren Ende enthält
    - eine, die alle DB-Änderungen enthält

Strategie:

- Wir werden unsere chronologischen Commits in "aktuelle" Commits umwandeln.
- zunächst alle Commits in mehrere, kleinere Commits aufteilen - jeweils nur ein Thema (in unserem Beispiel sind das Front-End, Back-End und DB-Änderungen)
- Dann ordnen Sie unsere aktuellen Commits zusammen und "zerquetschen" Sie sie in einzelne Commits

Beispiel:

```
$ git log --oneline master..
975430b db adding works: db.sql logic.rb
3702650 trying to allow adding todo items: page.html logic.rb
43b075a first draft: page.html and db.sql
$ git rebase -i master
```

Dies wird im Texteditor angezeigt:

```
pick 43b075a first draft: page.html and db.sql
pick 3702650 trying to allow adding todo items: page.html logic.rb
pick 975430b db adding works: db.sql logic.rb
```

Ändern Sie es zu diesem:

```
e 43b075a first draft: page.html and db.sql
e 3702650 trying to allow adding todo items: page.html logic.rb
e 975430b db adding works: db.sql logic.rb
```

Dann wendet git jeweils ein Commit an. Nach jedem Commit wird eine Eingabeaufforderung angezeigt, und Sie können dann Folgendes tun:

```
Stopped at 43b075a92a952faf999e76c4e4d7fa0f44576579... first draft: page.html and db.sql
You can amend the commit now, with

        git commit --amend

Once you are satisfied with your changes, run

        git rebase --continue

$ git status
rebase in progress; onto 4975ae9
You are currently editing a commit while rebasing branch 'feature' on '4975ae9'.
```

```
  (use "git commit --amend" to amend the current commit)
  (use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working directory clean
$ git reset HEAD^ #This 'uncommits' all the changes in this commit.
$ git status -s
 M db.sql
 M page.html
$ git add db.sql  #now we will create the smaller topical commits
$ git commit -m "first draft: db.sql"
$ git add page.html
$ git commit -m "first draft: page.html"
$ git rebase --continue
```

Dann werden Sie diese Schritte für jedes Commit wiederholen. Am Ende hast du folgendes:

```
$ git log --oneline
0309336 db adding works: logic.rb
06f81c9 db adding works: db.sql
3264de2 adding todo items: page.html
675a02b adding todo items: logic.rb
272c674 first draft: page.html
08c275d first draft: db.sql
```

Jetzt führen wir rebase noch einmal aus, um neu zu ordnen und zu squash:

```
$ git rebase -i master
```

Dies wird im Texteditor angezeigt:

```
pick 08c275d first draft: db.sql
pick 272c674 first draft: page.html
pick 675a02b adding todo items: logic.rb
pick 3264de2 adding todo items: page.html
pick 06f81c9 db adding works: db.sql
pick 0309336 db adding works: logic.rb
```

Ändern Sie es zu diesem:

```
pick 08c275d first draft: db.sql
s 06f81c9 db adding works: db.sql
pick 675a02b adding todo items: logic.rb
s 0309336 db adding works: logic.rb
pick 272c674 first draft: page.html
s  3264de2 adding todo items: page.html
```

HINWEIS: Stellen Sie sicher, dass Sie git rebase anweisen, die kleineren aktuellen Commits *in der Reihenfolge* anzuwenden, *in der sie chronologisch festgelegt wurden* . Andernfalls haben Sie möglicherweise falsche, unnötige Zusammenführungskonflikte, mit denen Sie fertig werden müssen.

Wenn diese interaktive Rebase alle gesagt und getan ist, erhalten Sie Folgendes:

```
$ git log --oneline master..
74bdd5f adding todos: GUI layer
e8d8f7e adding todos: business logic layer
121c578 adding todos: DB layer
```

Rekapitulieren

Sie haben jetzt Ihre chronologischen Commits in aktuelle Commits umgewandelt. Im wirklichen Leben müssen Sie dies möglicherweise nicht jedes Mal tun, aber wenn Sie dies wollen oder müssen, können Sie dies jetzt tun. Außerdem haben Sie hoffentlich mehr über Git Rebase erfahren.

Richten Sie git-pull ein, um automatisch eine Rebase statt einer Zusammenführung durchzuführen

Wenn Ihr Team einen Rebase-basierten Workflow verfolgt, kann es von Vorteil sein, git so einzustellen, dass jeder neu erstellte Zweig während eines `git pull` Vorgangs eine Rebase-Operation anstelle einer Merge-Operation ausführt.

Fügen Sie Ihrer `.gitconfig` oder `.gitconfig .git/config` Folgendes hinzu, um jeden *neuen* Zweig so einzurichten, dass er automatisch neu erstellt wird:

```
[branch]
autosetuprebase = always
```

Befehlszeile: *`git config [--global] branch.autosetuprebase always`*

Alternativ können Sie den Befehl `git pull --rebase` , dass er sich immer so verhält, als ob die Option `--rebase` wurde:

```
[pull]
rebase = true
```

Befehlszeile: *`git config [--global] pull.rebase true`*

Testen aller Commits während der Rebase

Bevor Sie eine Pull-Anforderung erstellen, sollten Sie sicherstellen, dass das Kompilieren erfolgreich ist und die Tests für jedes Commit in der Verzweigung bestanden werden. Dies kann automatisch mit dem Parameter `-x` .

Zum Beispiel:

```
git rebase -i -x make
```

führt die interaktive Rebase durch und stoppt nach jedem Commit, um `make` auszuführen. Falls `make` fehlschlägt, gibt git an, um Ihnen die Möglichkeit zu geben, die Probleme zu korrigieren und das Commit zu ändern, bevor Sie mit der nächsten fortfahren.

Autostash konfigurieren

Autostash ist eine sehr nützliche Konfigurationsoption, wenn Sie rebase für lokale Änderungen verwenden. Oft müssen Sie Commits aus dem Upstream-Zweig einbringen, sind aber noch nicht dazu bereit.

Git lässt jedoch nicht zu, dass eine Rebase gestartet wird, wenn das Arbeitsverzeichnis nicht sauber ist. Autostash zur Rettung:

```
git config --global rebase.autostash    # one time configuration
git rebase @{u}                         # example rebase on upstream branch
```

Der Autostash wird angewendet, sobald die Erneuerung abgeschlossen ist. Es spielt keine Rolle, ob die Basis erfolgreich abgeschlossen wurde oder ob sie abgebrochen wird. In beiden Fällen wird der Autostash angewendet. Wenn die Erneuerung erfolgreich war und sich das Basis-Commit geändert hat, kann es zu einem Konflikt zwischen dem Autostash und dem neuen Commit kommen. In diesem Fall müssen Sie die Konflikte lösen, bevor Sie sich verpflichten. Dies ist nicht anders als wenn Sie manuell verstaut und dann angewendet hätten, so dass es keinen Nachteil gibt, dies automatisch zu tun.

Neueinstellung online lesen: https://riptutorial.com/de/git/topic/355/neueinstellung

Kapitel 42: Reflog - Wiederherstellen von Commits, die nicht im Git-Log angezeigt werden

Bemerkungen

Bei Gits Refflog wird bei jeder Änderung die Position von HEAD (der Referenz für den aktuellen Status des Repositorys) aufgezeichnet. Im Allgemeinen umfasst jede Operation, die möglicherweise destruktiv ist, das Verschieben des HEAD-Zeigers (da sich bei Änderungen, auch in der Vergangenheit, der Hash-Befehl des Commits ändert), ist es immer möglich, vor einer gefährlichen Operation in einen älteren Zustand zurückzukehren , indem Sie die rechte Linie im reflog finden.

Objekte, auf die von keinem Ref verwiesen wird, werden in der Regel innerhalb von 30 Tagen gesammelt. Der Reflog kann also nicht immer helfen.

Examples

Wiederherstellen von einer schlechten Rebase

Angenommen, Sie haben eine interaktive Basis gestartet:

```
git rebase --interactive HEAD~20
```

und versehentlich haben Sie einige Commits, die Sie nicht verlieren wollten, gestaucht oder fallen gelassen, aber dann die Rebase abgeschlossen. Zur Wiederherstellung führen Sie `git reflog` aus. Möglicherweise sehen Sie eine Ausgabe wie diese:

```
aaaaaaa HEAD@{0} rebase -i (finish): returning to refs/head/master
bbbbbbb HEAD@{1} rebase -i (squash): Fix parse error
...
ccccccc HEAD@{n} rebase -i (start): checkout HEAD~20
ddddddd HEAD@{n+1} ...
...
```

In diesem Fall ist das letzte Commit, `ddddddd` (oder `HEAD@{n+1}`), die Spitze Ihres Zweigs, der sich *vor dem Rebase-Modus befindet* . Um dieses Commit (und alle übergeordneten Commits, einschließlich der versehentlich zusammengedrückten oder verworfenen) Commits wiederherzustellen, führen Sie Folgendes aus:

```
$ git checkout HEAD@{n+1}
```

Sie können dann an diesem Commit mit `git checkout -b [branch]` einen neuen Zweig erstellen. Weitere Informationen finden Sie unter Verzweigung .

Reflog - Wiederherstellen von Commits, die nicht im Git-Log angezeigt werden online lesen: https://riptutorial.com/de/git/topic/5149/reflog---wiederherstellen-von-commits--die-nicht-im-git-log-angezeigt-werden

Kapitel 43: Repositorys klonen

Syntax

- git clone [<options>] [-] <repo> [<dir>]
- git clone [--template = <template_directory>] [-l] [-s] [--no-hardlinks] [-q] [-n] [--bare] [--mirror] [-o <Name>] [-b <Name>] [-u <Upload-Pack>] [--reference <Repository>] [--dissociate] [--separate-git-dir <git dir>] [--depth <Tiefe>] [- [no-] single-branch] [--recursive | --recurse-submodules] [- [no-] shallow-submodules] [--jobs <n>] [-] <Repository> [<Verzeichnis>]

Examples

Flacher Klon

Das Klonen eines großen Repositorys (beispielsweise eines Projekts mit mehrjähriger Historie) kann aufgrund der zu übertragenden Datenmenge sehr lange dauern oder fehlschlagen. Wenn Sie nicht den vollständigen Verlauf zur Verfügung haben müssen, können Sie einen flachen Klon erstellen:

```
git clone [repo_url] --depth 1
```

Der obige Befehl holt nur den letzten Commit aus dem Remote-Repository.

Beachten Sie, dass Sie möglicherweise keine Zusammenführungen in einem flachen Repository auflösen können. Es ist oft eine gute Idee, mindestens so viele Commits anzunehmen, dass Sie zurückgehen müssen, um Zusammenführungen aufzulösen. Um beispielsweise die letzten 50 Commits zu erhalten:

```
git clone [repo_url] --depth 50
```

Bei Bedarf können Sie später den Rest des Repositorys abrufen:

1.8.3

```
git fetch --unshallow     # equivalent of git fetch --depth=2147483647
                          # fetches the rest of the repository
```

1.8.3

```
git fetch --depth=1000    # fetch the last 1000 commits
```

Regelmäßiger Klon

Um das gesamte Repository einschließlich des vollständigen Verlaufs und aller Zweige herunterzuladen, geben Sie Folgendes ein:

```
git clone <url>
```

Das obige Beispiel legt es in einem Verzeichnis ab, das den Namen des Repositorys trägt.

Um das Repository herunterzuladen und in einem bestimmten Verzeichnis zu speichern, geben Sie Folgendes ein:

```
git clone <url> [directory]
```

Weitere Informationen finden Sie unter Repository klonen .

Klonen Sie einen bestimmten Zweig

Um einen bestimmten Zweig eines Repositorys zu klonen, geben `--branch <branch name>` vor der Repository-URL `--branch <branch name>` :

```
git clone --branch <branch name> <url> [directory]
```

Um die Abkürzungsoption für `--branch` , geben Sie `-b` . Dieser Befehl lädt das gesamte Repository herunter und checkt `<branch name>` .

Um Speicherplatz zu sparen, können Sie die Historie klonen, die nur zu einem Zweig führt, mit:

```
git clone --branch <branch_name> --single-branch <url> [directory]
```

Wenn `--single-branch` nicht zum Befehl hinzugefügt wird, wird der Verlauf aller Zweige in `[directory]` geklont. Dies kann bei großen Repositorys auftreten.

Um später das Flag `--single-branch` rückgängig zu machen und den Rest des Repositorys abzurufen, verwenden Sie den Befehl:

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

Klonen Sie rekursiv

1.6.5

```
git clone <url> --recursive
```

Klont das Repository und auch alle Submodule. Wenn die Submodule selbst zusätzliche Submodule enthalten, werden diese auch von Git kopiert.

Klonen mit einem Proxy

Wenn Sie Dateien mit git unter einem Proxy herunterladen müssen, kann es nicht ausreichen, den

Proxy-Server systemweit einzustellen. Sie könnten auch folgendes versuchen:

```
git config --global http.proxy http://<proxy-server>:<port>/
```

Repositorys klonen online lesen: https://riptutorial.com/de/git/topic/1405/repositorys-klonen

Kapitel 44: Rev-Liste

Syntax

- git rev-list [Optionen] <Festschreiben> ...

Parameter

Parameter	Einzelheiten
--eine Linie	Commits als einzelne Zeile mit Titel anzeigen.

Examples

Liste Commits im Master, aber nicht im Origin / Master

```
git rev-list --oneline master ^origin/master
```

Git rev-list listet Commits in einem Zweig auf, die sich nicht in einem anderen Zweig befinden. Es ist ein großartiges Werkzeug, wenn Sie herausfinden möchten, ob Code in einen Zweig eingefügt wurde oder nicht.

- Mit der Option `--oneline` wird der Titel jedes Commits angezeigt.
- Der Operator `^` schließt Commits im angegebenen Zweig von der Liste aus.
- Sie können mehr als zwei Zweige übergeben, wenn Sie möchten. Zum Beispiel, `git rev-list foo bar ^baz` listet Commits in foo und bar auf, aber nicht baz.

Rev-Liste online lesen: https://riptutorial.com/de/git/topic/431/rev-liste

Kapitel 45: Rosinenpickerei

Einführung

Ein Kirschpick nimmt den Patch, der in einem Commit eingeführt wurde, und versucht, ihn in dem Zweig, in dem Sie sich gerade befinden, erneut anzuwenden.

Quelle: Git SCM Book

Syntax

- git cherry-pick [--edit] [-n] [-m übergeordnete Nummer] [-s] [-x] [--ff] [-S [Schlüssel-ID]] Commit
- git cherry pick - weiter
- git cherry pick - quit
- git cherry pick --abort

Parameter

Parameter	Einzelheiten
-e, --edit	Mit dieser Option können Sie die Commit-Nachricht mit `git cherry-pick` dem Commit bearbeiten.
-x	Hängen Sie beim Aufzeichnen des Commits eine Zeile mit der Aufschrift "(Kirsche aus Commit…)" an die ursprüngliche Commit-Nachricht an, um anzugeben, von welchem Commit diese Änderung aus dem Kirschbaum ausgewählt wurde. Dies geschieht nur für Kirschpickel ohne Konflikte.
--ff	Wenn der aktuelle HEAD mit dem übergeordneten Element des Cherry-pick'ed Commit identisch ist, wird ein Schnellvorlauf zu diesem Commit ausgeführt.
--fortsetzen	Setzen Sie den Vorgang unter Verwendung der Informationen in .git / Sequenzer fort. Kann verwendet werden, um nach dem Lösen von Konflikten in einem fehlgeschlagenen Cherry-Pick oder Revert fortzufahren.
--Verlassen	Vergessen Sie den laufenden Vorgang. Kann verwendet werden, um den Sequencer-Status nach einem fehlgeschlagenen Cherry-Pick oder Revert zu löschen.
--abbrechen	Brechen Sie den Vorgang ab und kehren Sie in den Zustand vor der Sequenz zurück.

Examples

Kopieren eines Commits von einem Zweig in einen anderen

`git cherry-pick <commit-hash>` wendet die in einem vorhandenen Commit vorgenommenen Änderungen auf einen anderen Zweig an und zeichnet ein neues Commit auf. Im Wesentlichen können Sie Commits von Zweig zu Zweig kopieren.

Gegeben der folgende Baum (Quelle)

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]
           \
            76cada - 62ecb3 - b886a0 [feature]
```

Nehmen wir an, wir wollen `b886a0` nach master kopieren (über `5a6057`).

Wir können rennen

```
git checkout master
git cherry-pick b886a0
```

Nun sieht unser Baum so aus:

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 - a66b23 [master]
           \
            76cada - 62ecb3 - b886a0 [feature]
```

Wobei das neue Commit `a66b23` denselben Inhalt (`a66b23` , Commit-Nachricht) wie `b886a0` (aber ein anderes übergeordnetes `b886a0`). Beachten Sie, dass die `b886a0` nur Änderungen dieses Commits (in diesem Fall `b886a0`) und nicht alle Änderungen im Feature-Zweig übernimmt (dazu müssen Sie entweder Rebasierung oder Zusammenführen verwenden).

Einen Commit-Bereich von einem Zweig in einen anderen kopieren

`git cherry-pick <commit-A>..<commit-B>` jedes Commit *nach* A und bis einschließlich B auf den gerade ausgecheckten Zweig.

`git cherry-pick <commit-A>^..<commit-B>` Commit A und jedes Commit bis einschließlich B auf den derzeit ausgecheckten Zweig.

Überprüfen, ob ein Kirschpickel erforderlich ist

Bevor Sie mit dem Kirschpick-Vorgang beginnen, können Sie überprüfen, ob das Commit, das Sie zum Kirschpicken auswählen möchten, bereits im Zielzweig vorhanden ist. In diesem Fall müssen Sie nichts tun.

`git branch --contains <commit>` listet lokale Zweigstellen auf, die den angegebenen Commit enthalten.

`git branch -r --contains <commit>` enthält auch Remote Tracking-Zweige in der Liste.

Finden Sie Commits, die noch auf Upstream angewendet werden sollen

Der Befehl `git cherry` zeigt die Änderungen, die noch nicht auserlesen wurden.

Beispiel:

```
git checkout master
git cherry development
```

... und sehen die Ausgabe ein bisschen so aus:

```
+ 492508acab7b454eee8b805f8ba906056eede0ff
- 5ceb5a9077ddb9e78b1e8f24bfc70e674c627949
+ b4459544c000f4d51d1ec23f279d9cdb19c1d32b
+ b6ce3b78e938644a293b2dd2a15b2fecb1b54cd9
```

Die Verpflichtung, mit `+` sein, werden diejenigen sein, die noch nicht in die `development` eingepfercht sind.

Syntax:

```
git cherry [-v] [<upstream> [<head> [<limit>]]]
```

Optionen:

-v Zeigt die Commit-Subjekte neben den SHA1s an.

<Upstream> Upstream-Zweig zur Suche nach äquivalenten Commits. Der Standardwert ist der Upstream-Zweig von HEAD.

<head> Arbeitszweig; Der Standardwert ist HEAD.

<limit> Keine Commits bis zum Limit (einschließlich) melden.

Überprüfen Sie die Git-Cherry-Dokumentation für weitere Informationen.

Rosinenpickerei online lesen: https://riptutorial.com/de/git/topic/672/rosinenpickerei

Kapitel 46: Schieben

Einführung

Nach dem Ändern, Staging und Festlegen von Code mit Git ist Push erforderlich, um Ihre Änderungen für andere verfügbar zu machen und Ihre lokalen Änderungen an den Repository-Server zu übertragen. In diesem Thema wird beschrieben, wie Sie den Code mithilfe von Git ordnungsgemäß pushen.

Syntax

- git push [-f | --force] [-v | --verbose] [<remote> [<refspec> ...]]

Parameter

Parameter	Einzelheiten
--Macht	Überschreibt den Remote-Ref entsprechend Ihrem lokalen Ref. *Kann dazu führen, dass das Remote-Repository Commits verliert, verwenden Sie es daher mit Vorsicht* .
--verbose	Laufen Sie wortreich.
<remote>	Das entfernte Repository, das das Ziel der Push-Operation ist.
<refspec> ...	Geben Sie an, welcher Remote-Ref mit welchem lokalen Ref oder Objekt aktualisiert werden soll.

Bemerkungen

Upstream & Downstream

In Bezug auf die Quellcodeverwaltung sind Sie **"Downstream",** wenn Sie aus einem Repository kopieren (Klonen, Auschecken usw.). Informationen flossen "stromabwärts" zu Ihnen.

Wenn Sie Änderungen vornehmen, möchten Sie sie normalerweise **"Upstream"** zurückschicken, damit sie in das Repository gelangen, sodass alle, die aus derselben Quelle ziehen, mit den gleichen Änderungen arbeiten. Dies ist meistens eine soziale Frage, wie jeder seine Arbeit koordinieren kann, und nicht eine technische Anforderung an die Quellcodeverwaltung. Sie möchten, dass Ihre Änderungen in das Hauptprojekt übernommen werden, sodass Sie keine abweichenden Entwicklungslinien verfolgen.

> Manchmal lesen Sie über Paket- oder Release-Manager (die Personen, nicht das Tool), die über das Senden von Änderungen an "Upstream" sprechen. Das bedeutet normalerweise, dass sie die Originalquellen anpassen mussten, um ein Paket für ihr System erstellen zu können. Sie möchten diese Änderungen nicht fortsetzen. Wenn sie also "Upstream" an die ursprüngliche Quelle gesendet werden, sollten sie sich in der nächsten Version nicht mit demselben Problem befassen müssen.

(Quelle)

Examples

drücken

```
git push
```

wird Ihren Code zu Ihrem vorhandenen Upstream pushen. Abhängig von der Push-Konfiguration wird entweder Code von Ihrem aktuellen Zweig (Standard in Git 2.x) oder von allen Zweigen (Standard in Git 1.x) übertragen.

Remote-Repository angeben

Wenn Sie mit git arbeiten, kann es praktisch sein, mehrere Remote-Repositorys zu haben. Um ein Remote-Repository anzugeben, an das Push gesendet werden soll, hängen Sie einfach den Namen an den Befehl an.

```
git push origin
```

Niederlassung angeben

Um zu einem bestimmten Zweig zu gelangen, sagen Sie `feature_x` :

```
git push origin feature_x
```

Legen Sie den Remote-Tracking-Zweig fest

Wenn der Zweig, an dem Sie gerade arbeiten, ursprünglich aus einem Remote-Repository stammt, funktioniert die Verwendung von `git push` nicht beim ersten Mal. Sie müssen den folgenden Befehl ausführen, um git mitzuteilen, dass der aktuelle Zweig auf eine bestimmte Remote- / Zweigkombination verschoben werden soll

```
git push --set-upstream origin master
```

`master` ist hier der Zweigname des entfernten `origin` . Sie können `-u` als Abkürzung für `--set-upstream` .

In ein neues Repository verschieben

Um zu einem Repository zu gelangen, das Sie noch nicht erstellt haben oder leer sind:

1. Erstellen Sie das Repository auf GitHub (falls zutreffend).
2. Kopieren Sie die angegebene URL in der Form `https://github.com/USERNAME/REPO_NAME.git`
3. Wechseln Sie zu Ihrem lokalen Repository und führen Sie `git remote add origin URL`
    - Um zu überprüfen, ob es hinzugefügt wurde, führen Sie `git remote -v`
4. Führen Sie `git push origin master`

Ihr Code sollte sich jetzt auf GitHub befinden

Weitere Informationen finden Sie unter Hinzufügen eines Remote-Repositorys

---

Erläuterung

Push-Code bedeutet, dass git die Unterschiede zwischen Ihren lokalen Commits und Remote analysiert und diese an den Upstream-Server sendet. Wenn Push erfolgreich ist, werden Ihr lokales Repository und Ihr Remote-Repository synchronisiert, und andere Benutzer können Ihre Commits sehen.

Weitere Informationen zu den Konzepten "vorgelagert" und "nachgelagert" finden Sie unter Anmerkungen .

Zwangsschub

Wenn Sie lokale Änderungen haben, die nicht mit Remote-Änderungen kompatibel sind (z. B. wenn Sie den Remote-Zweig nicht schnell vorspulen können oder der Remote-Zweig kein direkter Vorfahre Ihres lokalen Zweigs ist), können Sie die Änderungen nur durch einen Push-Befehl erzwingen .

```
git push -f
```

oder

```
git push --force
```

Wichtige Notizen

Dadurch werden alle Änderungen an der Fernbedienung **überschrieben,** und Ihre Fernbedienung passt zu Ihrer lokalen.

Achtung: Die Verwendung dieses Befehls kann dazu führen, dass das Remote-Repository **Commits verliert** . Darüber hinaus wird dringend davon abgeraten, einen Force-Push durchzuführen, wenn Sie dieses Remote-Repository mit anderen teilen, da deren Historie jedes überschriebene Commit beibehält und somit die Arbeit mit dem Remote-Repository nicht mehr synchron ist.

Als Faustregel gilt: Nur wenn:

- Niemand außer Sie haben die Änderungen übernommen, die Sie überschreiben möchten
- Sie können jeden zwingen, nach dem erzwungenen Druck eine neue Kopie zu kopieren, und alle Änderungen daran vornehmen (die Leute mögen Sie dafür hassen).

Schieben Sie ein bestimmtes Objekt an einen entfernten Zweig

Allgemeine Syntax

```
git push <remotename> <object>:<remotebranchname>
```

Beispiel

```
git push origin master:wip-yourname
```

`wip-yourname` Ihren Master-Zweig zum Ursprungszweig " `wip-yourname` (meistens aus dem Repository, aus dem Sie geklont haben).

Entfernten Zweig löschen

Das Löschen des entfernten Zweigs entspricht einem leeren Objekt.

```
git push <remotename> :<remotebranchname>
```

Beispiel

```
git push origin :wip-yourname
```

Löscht den entfernten Zweig `wip-yourname`

Anstelle des Doppelpunkts können Sie auch das Flag --delete verwenden, das in manchen Fällen besser lesbar ist.

Beispiel

```
git push origin --delete wip-yourname
```

Drücken Sie ein einzelnes Commit

Wenn Sie in Ihrem Zweig ein einziges Commit haben, das Sie an eine Remote verschieben möchten, ohne etwas anderes zu drücken, können Sie Folgendes verwenden

```
git push <remotename> <commit SHA>:<remotebranchname>
```

Beispiel

Angenommen, eine Git-Geschichte wie diese

```
eeb32bc Commit 1 - already pushed
347d700 Commit 2 - want to push
e539af8 Commit 3 - only local
5d339db Commit 4 - only local
```

Verwenden Sie den folgenden Befehl, um nur den Commit *347d700* an den Remote- *Master zu übergeben*

```
git push origin 347d700:master
```

Ändern des Standard-Push-Verhaltens

Current aktualisiert den Zweig des Remote-Repositorys, der einen Namen mit dem aktuellen Arbeitszweig teilt.

```
git config push.default current
```

Einfache Pushs an den Upstream-Zweig, funktionieren jedoch nicht, wenn der Upstream-Zweig anders bezeichnet wird.

```
git config push.default simple
```

Upstream schiebt in den Upstream-Zweig, egal wie er genannt wird.

```
git config push.default upstream
```

Beim Matching werden alle Zweige, die auf der lokalen und der fernen git config push.default übereinstimmen, in den Upstream verschoben

Nachdem Sie den bevorzugten Stil festgelegt haben, verwenden Sie

```
git push
```

um das entfernte Repository zu aktualisieren.

Tags drücken

```
git push --tags
```

Schickt alle `git tags` im lokalen Repository, die sich nicht im fernen befinden.

Schieben online lesen: https://riptutorial.com/de/git/topic/2600/schieben

Kapitel 47: Schuldzuweisungen

Syntax

- git blame [Dateiname]
- git blame [-f] [- e] [- w] [Dateiname]
- git blame [-L Bereich] [Dateiname]

Parameter

Parameter	Einzelheiten
Dateiname	Name der Datei, für die Details geprüft werden müssen
-f	Zeigen Sie den Dateinamen im Ursprungs-Commit an
-e	Zeigt die Autor-E-Mail-Adresse anstelle des Autorennamens an
-w	Ignorieren Sie Leerzeichen, während Sie einen Vergleich zwischen der Version des Kindes und des Elternteils durchführen
-L Start, Ende	Nur den angegebenen Zeilenbereich anzeigen Beispiel: `git blame -L 1,2 [filename]`
- show-stats	Zeigt zusätzliche Statistiken am Ende der Schuldverschreibung an
-l	Lange Drehzahl anzeigen (Standard: aus)
-t	Rohzeitstempel anzeigen (Standard: aus)
-umkehren	Gehe die Geschichte vorwärts statt rückwärts
-p, --porzellan	Ausgabe für den Maschinenverbrauch
-M	Verschobene oder kopierte Zeilen in einer Datei erkennen
-C	Ermitteln Sie neben -M auch Zeilen, die aus anderen Dateien verschoben oder kopiert wurden, die im selben Commit geändert wurden
-h	Zeigen Sie die Hilfemeldung an
-c	Verwenden Sie den gleichen Ausgabemodus wie git-annotate (Standard: aus)
-n	Zeigt die Zeilennummer im ursprünglichen Commit an (Standard: aus)

Bemerkungen

Der Befehl git blame ist sehr nützlich, wenn Sie wissen möchten, wer auf Zeilenbasis Änderungen an einer Datei vorgenommen hat.

Examples

Zeigt das Commit an, das zuletzt eine Zeile geändert hat

```
git blame <file>
```

zeigt die Datei mit jeder Zeile, die mit dem Commit versehen wurde, der sie zuletzt geändert hat

Ignoriere nur Whitespace-Änderungen

Repos enthalten manchmal Commits, die nur den Leerraum anpassen, z. B. die Einrückung korrigieren oder zwischen Tabs und Leerzeichen wechseln. Dies macht es schwierig, das Commit zu finden, an dem der Code tatsächlich geschrieben wurde.

```
git blame -w
```

ignoriert nur Whitespace-Änderungen, um herauszufinden, woher die Linie wirklich kam.

Nur bestimmte Zeilen anzeigen

Die Ausgabe kann durch Angabe von Zeilenbereichen als eingeschränkt werden

```
git blame -L <start>,<end>
```

Wo können `<start>` und `<end>` sein:

- Zeilennummer

  ```
  git blame -L 10,30
  ```

- / Regex /

  ```
  git blame -L /void main/ , git blame -L 46,/void foo/
  ```

- + offset, -offset (nur für `<end>`)

  ```
  git blame -L 108,+30 , git blame -L 215,-15
  ```

Es können mehrere Linienbereiche angegeben werden und überlappende Bereiche sind zulässig.

```
git blame -L 10,30 -L 12,80 -L 120,+10 -L ^/void main/,+40
```

Um herauszufinden, wer eine Datei geändert hat

```
// Shows the author and commit per line of specified file
```

```
git blame test.c

// Shows the author email and commit per line of specified
git blame -e test.c file

// Limits the selection of lines by specified range
git blame -L 1,10 test.c
```

Schuldzuweisungen online lesen: https://riptutorial.com/de/git/topic/3663/schuldzuweisungen

Kapitel 48: Show

Syntax

- git show [Optionen] <Objekt> ...

Bemerkungen

Zeigt verschiedene Git-Objekte.

- Bei Commits wird die Commit-Nachricht und der Diff angezeigt
- Zeigt für Tags die Tag-Nachricht und das referenzierte Objekt an

Examples

Überblick

`git show` zeigt verschiedene Git-Objekte.

Für Commits:

Zeigt die Commit-Nachricht und einen Unterschied der eingeführten Änderungen an.

Befehl	Beschreibung
`git show`	zeigt den vorherigen Commit
`git show @~3`	zeigt das dritte vorletzte Commit

Für Bäume und Kleckse:

Zeigt den Baum oder Fleck.

Befehl	Beschreibung
`git show @~3:`	Zeigt das Projektstammverzeichnis an, wie es vor 3 Commits war (ein Baum)
`git show @~3:src/program.js`	Zeigt `src/program.js` wie es vor 3 Commits war (ein Blob)
`git show @:a.txt @:b.txt`	Zeigt eine mit `a.txt` verkettete `b.txt` aus dem aktuellen Commit

Für Tags:

Zeigt die Tag-Nachricht und das referenzierte Objekt an.

Show online lesen: https://riptutorial.com/de/git/topic/3030/show

Kapitel 49: Squashing

Bemerkungen

Was ist Quetschen?

Squashing ist der Prozess, bei dem mehrere Commits in einem einzigen Commit zusammengefasst werden, der alle Änderungen der ursprünglichen Commits umfasst.

Quetschen und entfernte Äste

Seien Sie besonders vorsichtig, wenn Sie Commits in einem Zweig komprimieren, der einen entfernten Zweig verfolgt. Wenn Sie ein Commit, das bereits an einen entfernten Zweig gepusht wurde, quetschen, werden die beiden Äste auseinandergeführt, und Sie müssen `git push -f` , um diese Änderungen auf den entfernten Zweig zu erzwingen. **Beachten Sie, dass dies zu Problemen für andere Benutzer führen kann, die diesen entfernten Zweig verfolgen. Aus diesem Grund ist** Vorsicht geboten, wenn erzwungene Commits in öffentlichen oder gemeinsam genutzten Repositories erzwungen werden.

Wenn das Projekt auf GitHub gehostet wird, können Sie den "Push-Push-Schutz erzwingen" für einige Zweige wie den `master` aktivieren, indem Sie es zu `Settings` - `Branches` - `Protected Branches` hinzufügen.

Examples

Letzte Commits für Squash ohne erneute Wiederherstellung

Wenn Sie die vorherigen `x` Commits zu einem einzigen komprimieren möchten, können Sie die folgenden Befehle verwenden:

```
git reset --soft HEAD~x
git commit
```

Ersetzen Sie `x` durch die Anzahl der vorherigen Commits, die in das komprimierte Commit aufgenommen werden sollen.

Beachten Sie, dass dadurch ein *neues* Commit erstellt wird, das im Wesentlichen die Informationen zu den vorherigen `x` Commits (einschließlich Autor, Nachricht und Datum) vergisst. Sie möchten wahrscheinlich *zuerst* eine vorhandene Bestätigungsnachricht kopieren und einfügen.

Quetsch-Commits während einer Rebase

Commits können während einer `git rebase` gestaucht werden. Es wird empfohlen, dass Sie die Umbasierung verstehen, bevor Sie versuchen, Commits auf diese Weise zu quetschen .

1. Stellen Sie fest, von welchem Commit Sie zurückgestoßen werden möchten, und notieren Sie den Commit-Hash.

2. Führen Sie `git rebase -i [commit hash]` .

   Alternativ können Sie `HEAD~4` anstelle eines Commit-Hashes eingeben, um das letzte Commit und 4 weitere Commits vor dem letzten anzuzeigen.

3. Bestimmen Sie im Editor, der beim Ausführen dieses Befehls geöffnet wird, welche Commits Sie quetschen möchten. Ersetzen Sie `pick` am Anfang dieser Zeilen durch `squash` um sie beim vorherigen Commit zu quetschen.

4. Nachdem Sie die Commits ausgewählt haben, die Sie quetschen möchten, werden Sie aufgefordert, eine Commit-Nachricht zu schreiben.

Logging-Commits, um zu bestimmen, wo die Datenbank neu erstellt werden soll

```
> git log --oneline
612f2f7 This commit should not be squashed
d84b05d This commit should be squashed
ac60234 Yet another commit
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit

> git rebase -i 36d15de
```

An dieser Stelle wird der Editor Ihrer Wahl angezeigt, in dem Sie beschreiben können, was Sie mit den Commits machen möchten. Git bietet Hilfe in den Kommentaren. Wenn Sie es so belassen, wie es ist, wird nichts passieren, da jedes Commit beibehalten wird und seine Reihenfolge dieselbe ist wie vor der Rebase. In diesem Beispiel wenden wir die folgenden Befehle an:

```
pick ac60234 Yet another commit
squash d84b05d This commit should be squashed
pick 612f2f7 This commit should not be squashed

# Rebase 36d15de..612f2f7 onto 36d15de (3 command(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
```

Git-Protokoll nach dem Schreiben der Commit-Nachricht

```
> git log --oneline
77393eb This commit should not be squashed
e090a8c Yet another commit
36d15de Rebase from here
17692d1 Did some more stuff
e647334 Another Commit
2e30df6 Initial commit
```

Autosquash: Commit-Code, den Sie während einer Rebase quetschen möchten

Stellen Sie sich vor, Sie nehmen eine Änderung vor, die Sie für das Commit `bbb2222 A second commit` möchten:

```
$ git log --oneline --decorate
ccc3333 (HEAD -> master) A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

Nachdem Sie Ihre Änderungen vorgenommen haben, können Sie sie wie üblich zum Index hinzufügen und dann mit dem Argument `--fixup` mit einem Verweis auf das Commit, das Sie `--fixup` möchten, `--fixup` :

```
$ git add .
$ git commit --fixup bbb2222
[my-feature-branch ddd4444] fixup! A second commit
```

Dadurch wird ein neues Commit mit einer Commit-Nachricht erstellt, die Git während einer interaktiven Rebase erkennen kann:

```
$ git log --oneline --decorate
ddd4444 (HEAD -> master) fixup! A second commit
ccc3333 A third commit
bbb2222 A second commit
aaa1111 A first commit
9999999 Initial commit
```

Führen Sie als Nächstes eine interaktive Basis mit dem Argument `--autosquash` :

```
$ git rebase --autosquash --interactive HEAD~4
```

Git wird Ihnen vorschlagen, das Commit, das Sie mit dem `commit --fixup` in die richtige Position zu `commit --fixup` :

```
pick aaa1111 A first commit
pick bbb2222 A second commit
fixup ddd4444 fixup! A second commit
pick ccc3333 A third commit
```

Um zu vermeiden, dass Sie `--autosquash` auf jeder Basisstation eingeben müssen, können Sie diese Option standardmäßig aktivieren:

```
$ git config --global rebase.autosquash true
```

Quetsch-Commit beim Zusammenführen

Sie können `git merge --squash` , um die von einem Zweig eingeführten Änderungen in einem einzigen Commit zu `git merge --squash` . Es wird kein tatsächliches Commit erstellt.

```
git merge --squash <branch>
git commit
```

Dies ist mehr oder weniger gleichbedeutend mit der Verwendung von `git reset` , ist jedoch bequemer, wenn die eingegebenen Änderungen einen symbolischen Namen haben. Vergleichen Sie:

```
git checkout <branch>
git reset --soft $(git merge-base master <branch>)
git commit
```

Autosquashing und Korrekturen

Beim Festschreiben von Änderungen kann festgelegt werden, dass das Commit in Zukunft auf ein anderes Commit komprimiert wird.

```
git commit --squash=[commit hash of commit to which this commit will be squashed to]
```

Man könnte auch `--fixup=[commit hash]` alternativ zu fixup verwenden.

Es ist auch möglich, Wörter aus der Festschreibungsnachricht anstelle des Festschreibungs-Hashwerts zu verwenden.

```
git commit --squash :/things
```

wo das letzte Commit mit dem Wort "Dinge" verwendet würde.

Diese Commits-Nachricht würde mit `'fixup!'` oder `'squash!'` gefolgt von dem Rest der Commit-Nachricht, in die diese Commits zerquetscht werden.

Beim Umbasieren sollte `--autosquash` flag verwendet werden, um die Funktion autosquash / fixup zu verwenden.

Squashing online lesen: https://riptutorial.com/de/git/topic/598/squashing

Kapitel 50: Submodule

Examples

Submodul hinzufügen

Sie können ein anderes Git-Repository als Ordner in Ihr Projekt aufnehmen, das von Git verfolgt wird:

```
$ git submodule add https://github.com/jquery/jquery.git
```

Sie sollten die neue `.gitmodules` Datei hinzufügen und `.gitmodules` . Dies teilt Git mit, welche Submodule geklont werden sollen, wenn das `git submodule update` wird.

Klonen eines Git-Repositorys mit Submodulen

Wenn Sie ein Repository klonen, das Submodule verwendet, müssen Sie diese initialisieren und aktualisieren.

```
$ git clone --recursive https://github.com/username/repo.git
```

Dadurch werden die referenzierten Submodule geklont und in den entsprechenden Ordnern (einschließlich Submodulen innerhalb von Submodulen) abgelegt. Dies ist gleichbedeutend mit der Ausführung von `git submodule update --init --recursive` unmittelbar nachdem der Klon beendet ist.

Aktualisieren eines Submoduls

Ein Submodul verweist auf ein bestimmtes Commit in einem anderen Repository. Führen Sie run aus, um den genauen Status zu überprüfen, der für alle Submodule referenziert wird

```
git submodule update --recursive
```

Anstatt den Status zu verwenden, auf den verwiesen wird, möchten Sie in Ihrem lokalen Checkout den neuesten Status dieses Submoduls auf einer Fernbedienung aktualisieren. Um alle Submodule mit einem einzigen Befehl auf den neuesten Stand auf der Fernbedienung auszuchecken, können Sie verwenden

```
git submodule foreach git pull <remote> <branch>
```

oder verwenden Sie die Standardgit- `git pull` Argumente

```
git submodule foreach git pull
```

Beachten Sie, dass dies nur Ihre lokale Arbeitskopie aktualisiert. Beim Ausführen von `git status`

wird das Submodul-Verzeichnis als fehlerhaft angezeigt, wenn es sich aufgrund dieses Befehls geändert hat. Um stattdessen Ihr Repository zu aktualisieren, um auf den neuen Status zu verweisen, müssen Sie die Änderungen festschreiben:

```
git add <submodule_directory>
git commit
```

Wenn Sie `git pull` , kann es zu einigen Änderungen kommen, bei denen Konflikte auftreten können, wenn Sie `git pull` So können Sie `git pull --rebase` , um Ihre Änderungen nach oben zurückzuspulen. Meist verringert dies die Konfliktwahrscheinlichkeit. Es zieht auch alle Äste nach lokal.

```
git submodule foreach git pull --rebase
```

Um den neuesten Status eines bestimmten Submoduls zu überprüfen, können Sie Folgendes verwenden:

```
git submodule update --remote <submodule_directory>
```

Festlegen eines Submoduls, um einem Zweig zu folgen

Ein Submodul wird immer bei einem bestimmten Commit SHA1 (dem "gitlink", spezieller Eintrag im Index des übergeordneten Repos) ausgecheckt

Man kann jedoch anfordern, dieses Submodul auf das letzte Commit eines Zweigs des Submoduls Remote Repo zu aktualisieren.

Anstatt in jedes Submodul zu gehen, eine `git checkout abranch --track origin/abranch, git pull` , können Sie (aus dem übergeordneten Repo) einfach `git checkout abranch --track origin/abranch, git pull` tun:

```
git submodule update --remote --recursive
```

Da sich der SHA1 des Submoduls ändern würde, müssten Sie dem noch folgen:

```
git add .
git commit -m "update submodules"
```

Das setzt voraus, dass die Submodule waren:

- entweder mit folgendem Zweig hinzugefügt:

  ```
  git submodule -b abranch -- /url/of/submodule/repo
  ```

- oder konfiguriert (für ein vorhandenes Submodul), um einem Zweig zu folgen:

  ```
  cd /path/to/parent/repo
  ```

```
git config -f .gitmodules submodule.asubmodule.branch abranch
```

Entfernen eines Submoduls

1.8

Sie können ein Submodul (z. B. das `the_submodule`) entfernen, indem Sie `the_submodule` aufrufen:

```
$ git submodule deinit the_submodule
$ git rm the_submodule
```

- `git submodule deinit the_submodule` Löschungen `the_submodule` s'Eintrag von .git / config. Dies schließt das Submodul von `git submodule update` , `git submodule sync` und `git submodule foreach` Aufrufe aus und löscht seinen lokalen Inhalt (Quelle) . Dies wird auch nicht als Änderung in Ihrem übergeordneten Repository angezeigt. `git submodule init` und `git submodule update` stellen das Submodul wieder her, ohne dass Änderungen in Ihrem übergeordneten Repository vorgenommen werden müssen.

- `git rm the_submodule` entfernt das Submodul aus dem Arbeitsbaum. Die Dateien werden ebenso wie der Eintrag der Submodule in der `.gitmodules` Datei (Quelle) `.gitmodules` . Wenn nur `git rm the_submodule` (ohne vorheriges `git submodule deinit the_submodule` ausgeführt wird, bleibt der Eintrag der Submodule in Ihrer .git / config -Datei erhalten.

1.8

Von hier genommen :

1. Löschen Sie den relevanten Abschnitt aus der `.gitmodules` Datei.
2. Stufe der `.gitmodules` Änderungen `git add .gitmodules`
3. Löschen Sie den relevanten Abschnitt aus `.git/config` .
4. Führen Sie `git rm --cached path_to_submodule` (kein `git rm --cached path_to_submodule` Schrägstrich).
5. Führen Sie `rm -rf .git/modules/path_to_submodule`
6. Commit `git commit -m "Removed submodule <name>"`
7. Löschen Sie die jetzt nicht protokollierten Submodul-Dateien
8. `rm -rf path_to_submodule`

Verschieben eines Submoduls

1.8

Lauf:

```
$ git mv old/path/to/module new/path/to/module
```

1.8

1. Bearbeiten Sie .gitmodules und ändern Sie den Pfad des Submoduls entsprechend. git add .gitmodules mit git add .gitmodules in den Index git add .gitmodules .

2. Erstellen Sie bei Bedarf das übergeordnete Verzeichnis des neuen Speicherorts des Submoduls (mkdir -p *new/path/to*).
3. Verschieben Sie den gesamten Inhalt vom alten in das neue Verzeichnis (mv -vi *old/path/to/module new/path/to/submodule*).
4. Stellen Sie sicher, dass Git dieses Verzeichnis verfolgt (git add *new/path* /to).
5. Entfernen Sie das alte Verzeichnis mit git rm --cached *old/path/to/module* .
6. Verschieben Sie das Verzeichnis .git/modules/ *old/path/to/module* mit seinem gesamten Inhalt nach .git/modules/ *new/path/to/module* .
7. Bearbeiten Sie die Datei .git .git/modules/ *new/path/to* /config , und stellen Sie sicher, dass das Worktree-Element auf die neuen Speicherorte verweist. In diesem Beispiel sollte es also worktree = ../../../../../ *old/path/to/module* . In der Regel sollten sich an dieser Stelle zwei weitere .. dann Verzeichnisse im direkten Pfad befinden. . Bearbeiten Sie die Datei *new/path/to/module* /.git , stellen Sie sicher , dass der Pfad in ihm auf den richtigen neuen Standort innerhalb des Hauptprojekt .git Ordner, so in diesem Beispiel gitdir: ../../../.git/modules/ *new/path/to/module* .

   git status Ausgabe des git status sieht danach so aus:

   ```
   # On branch master
   # Changes to be committed:
   #   (use "git reset HEAD <file>..." to unstage)
   #
   #       modified:   .gitmodules
   #       renamed:    old/path/to/submodule -> new/path/to/submodule
   #
   ```

8. Bestätigen Sie anschließend die Änderungen.

Dieses Beispiel aus Stack Overflow von Axel Beckert

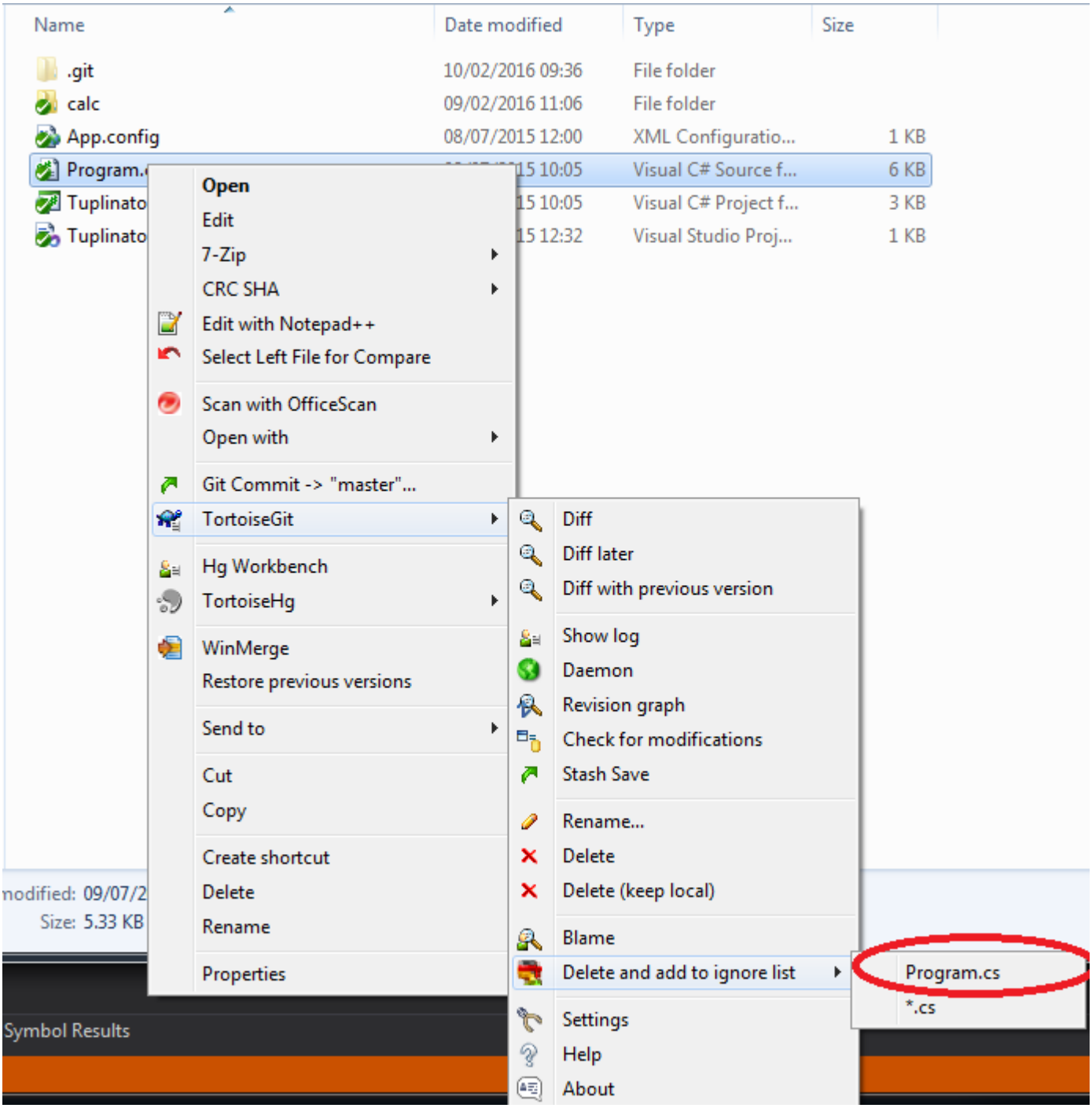
Submodule online lesen: https://riptutorial.com/de/git/topic/306/submodule

Kapitel 51: TortoiseGit

Examples

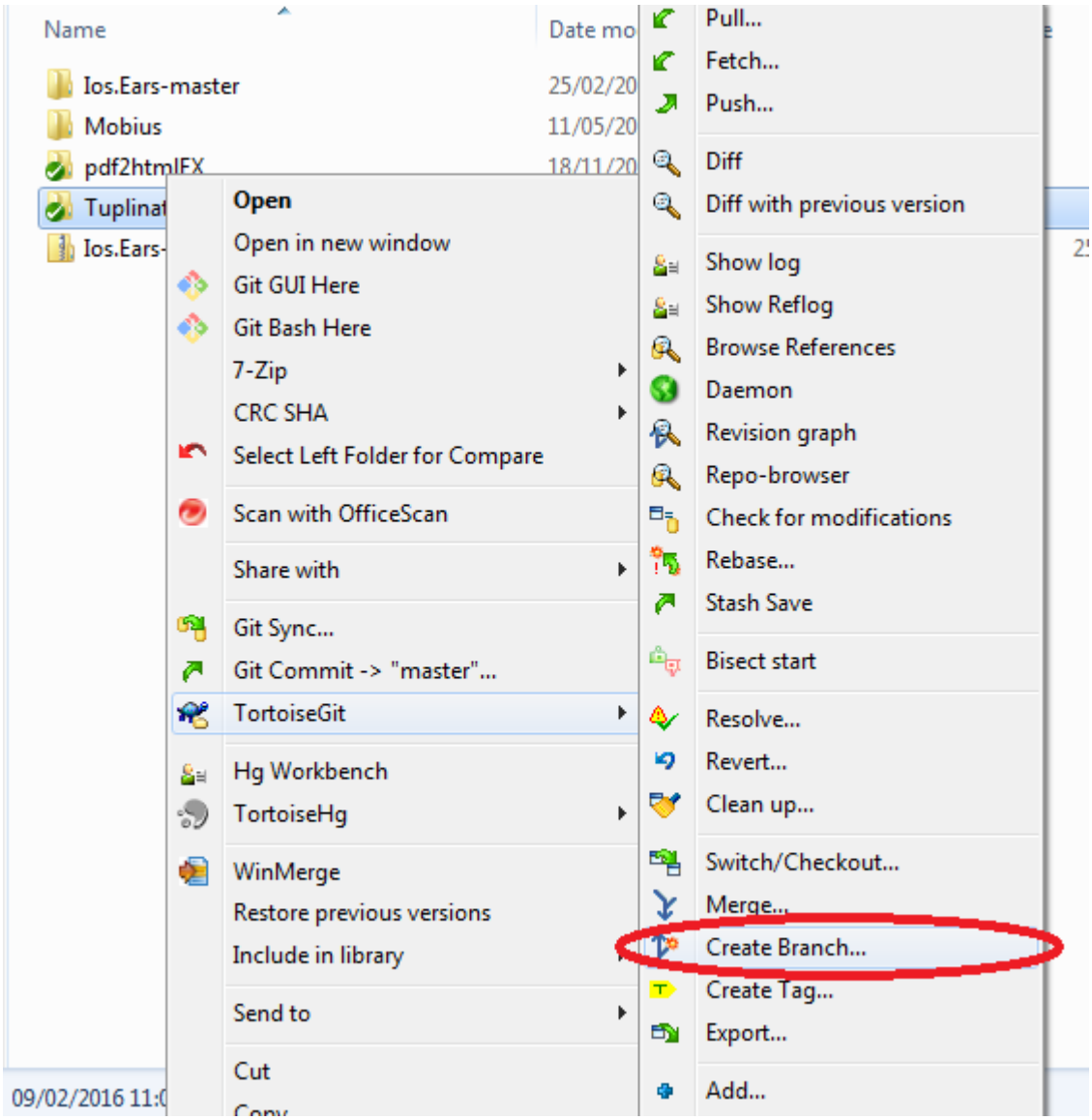
Dateien und Ordner ignorieren

Diejenigen, die die TortioseGit-Benutzeroberfläche verwenden, klicken Sie mit der rechten Maustaste auf die Datei (oder den Ordner), die Sie ignorieren möchten -> TortoiseGit -> Delete and add to ignore list Klicken Sie auf Ok und Sie sollten fertig sein.

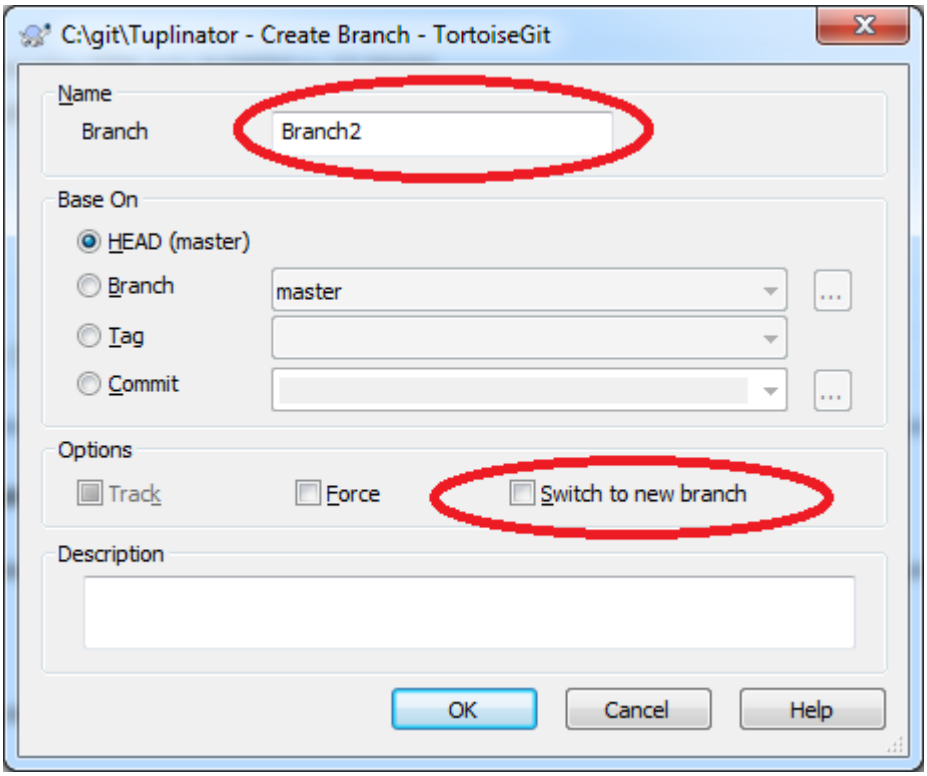


Verzweigung

Für diejenigen, die eine Benutzeroberfläche verwenden, um zu verzweigen, klicken Sie im Repository mit der rechten Maustaste, dann auf Tortoise Git -> Create Branch...

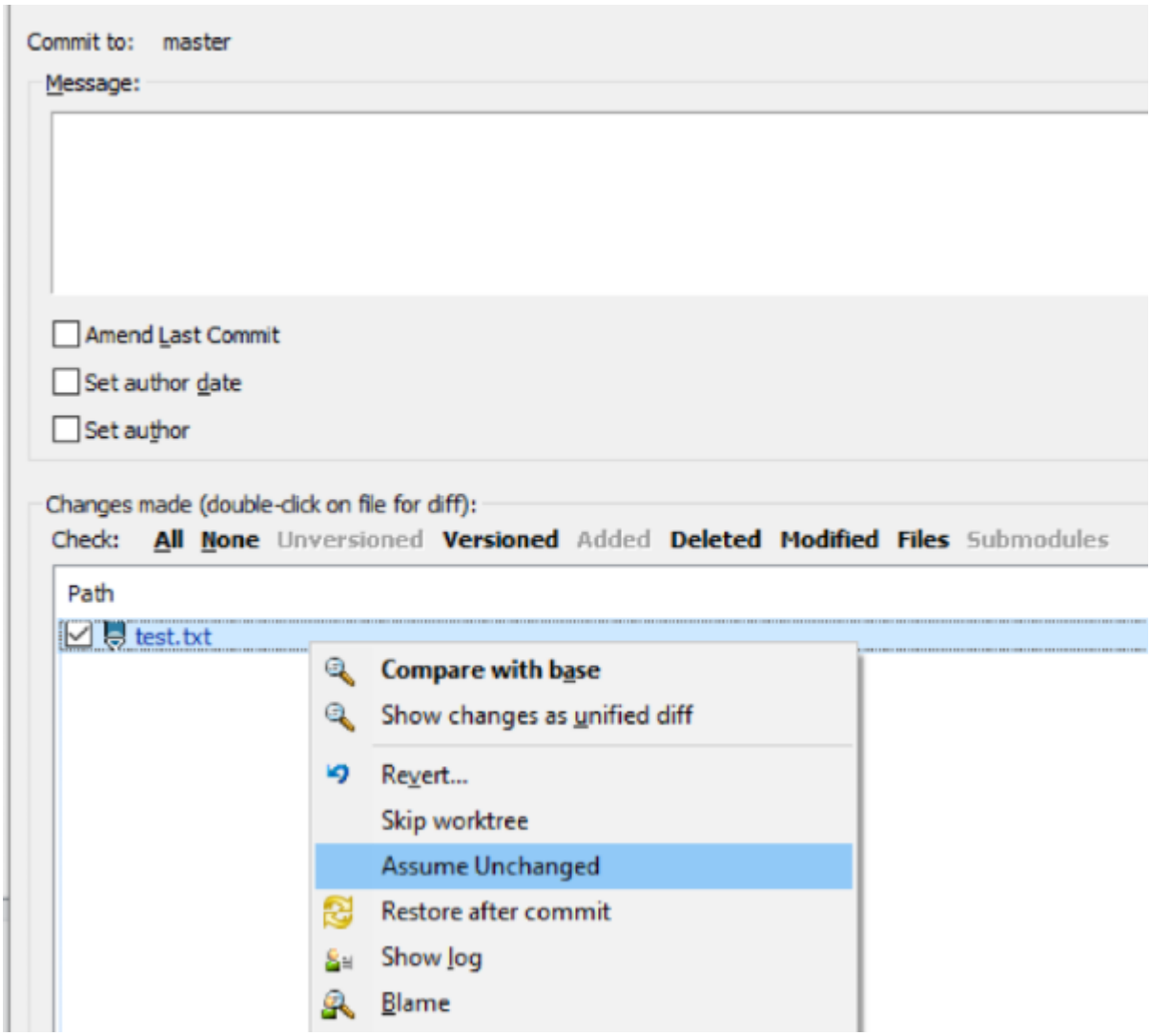


Neues Fenster wird geöffnet -> Give branch a name -> Markieren Sie das Kästchen Switch to new branch wechseln (Wahrscheinlich möchten Sie nach dem Verzweigen mit ihm arbeiten). -> Klicken Sie auf OK und Sie sollten fertig sein.



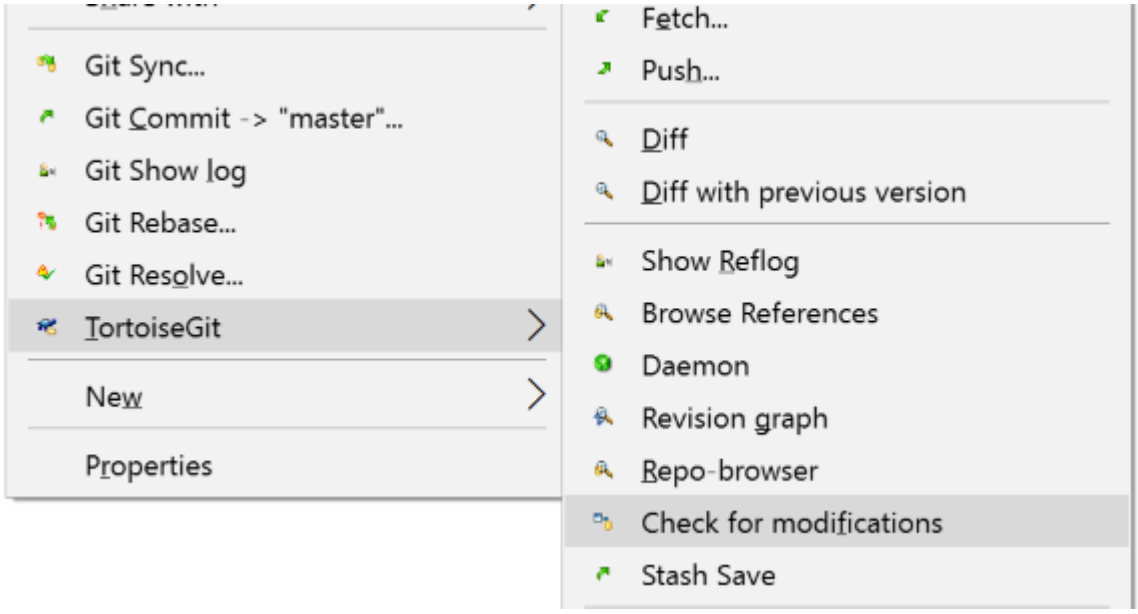
Angenommen unverändert

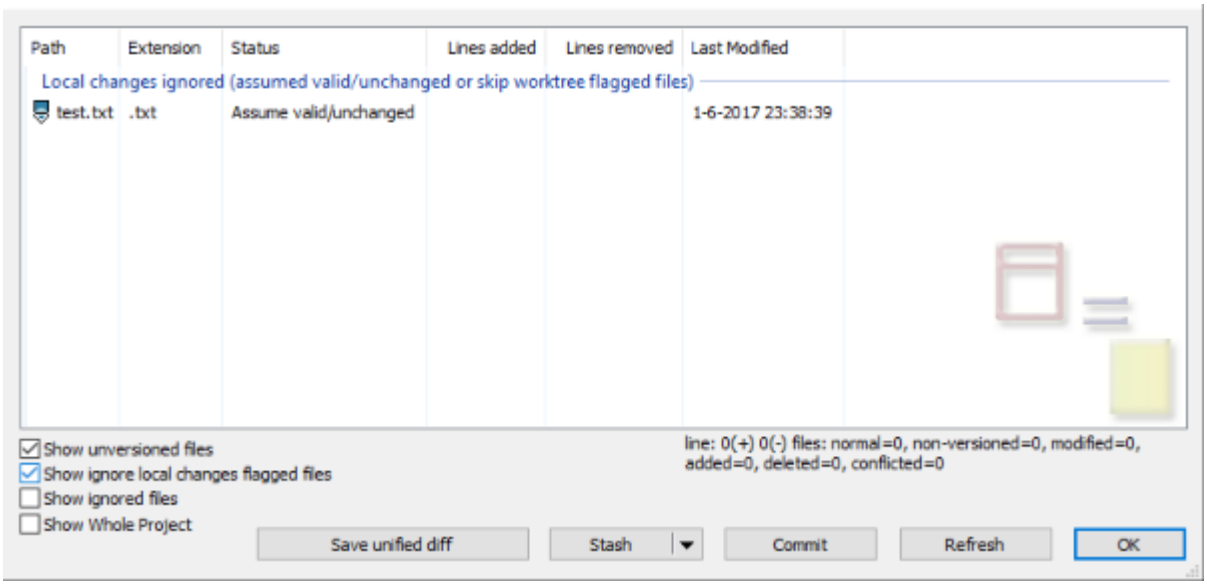
Wenn eine Datei geändert wird, Sie aber nicht festschreiben möchten, legen Sie die Datei als "Angenommen unverändert"

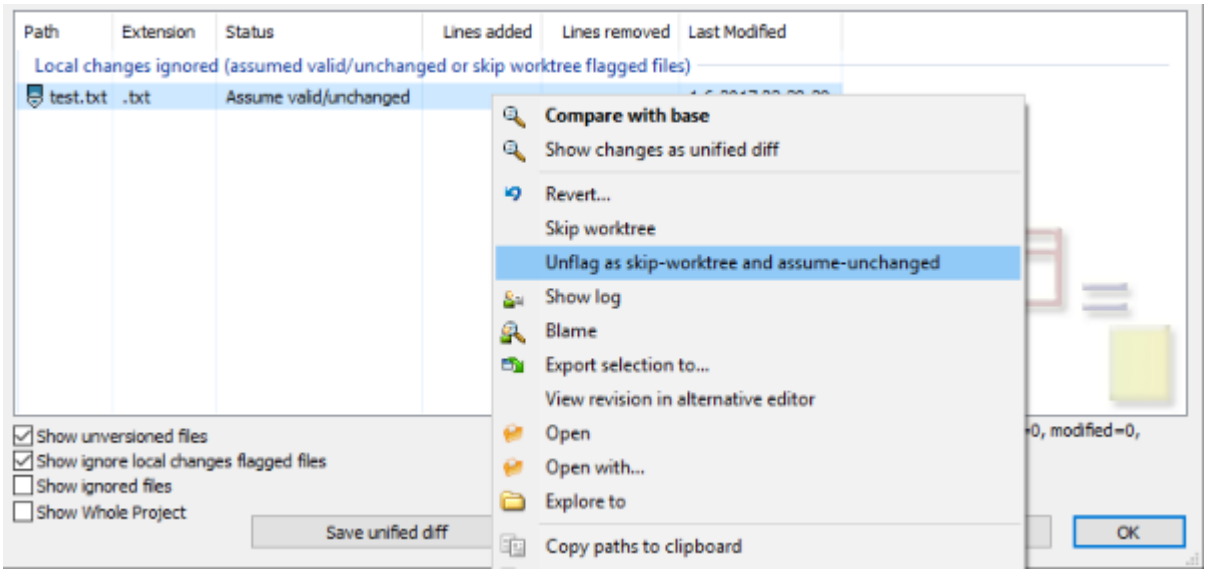


Rückgängig machen "unverändert annehmen"

Benötigen Sie einige Schritte:



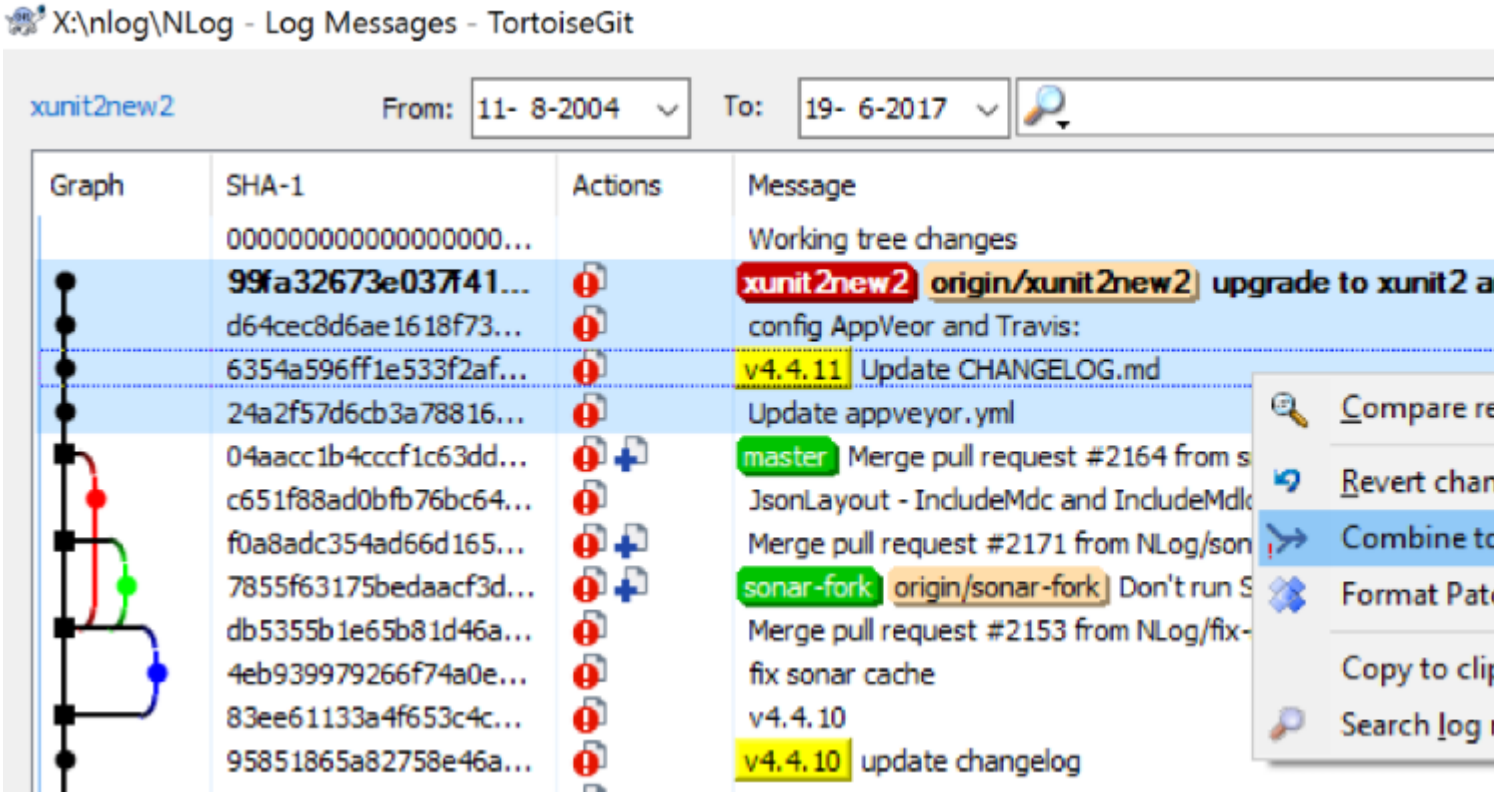




Squash begeht

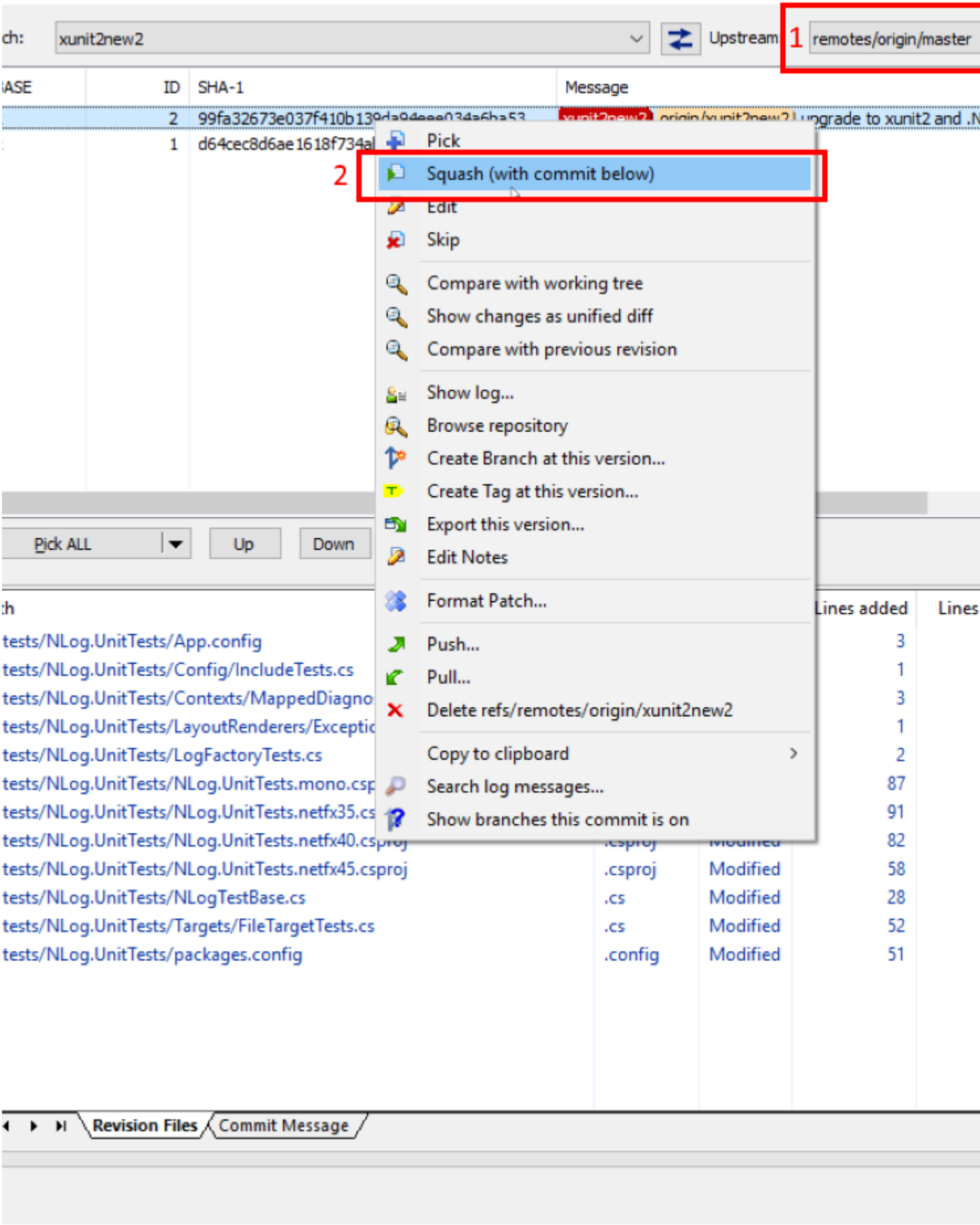
Der einfache Weg

Dies funktioniert nicht, wenn die Auswahl Merge-Commits enthält



Der fortgeschrittene Weg

Starten Sie den Rebase-Dialog:



TortoiseGit online lesen: https://riptutorial.com/de/git/topic/5150/tortoisegit

Kapitel 52: Umbenennung

Syntax

- `git mv <source> <destination>`
  - `git mv -f <source> <destination>`

Parameter

Parameter	Einzelheiten
`-f` oder `--force`	Erzwingen Sie das Umbenennen oder Verschieben einer Datei, auch wenn das Ziel vorhanden ist

Examples

Ordner umbenennen

Um einen Ordner von oldName in newName

```
git mv directoryToFolder/oldName directoryToFolder/newName
```

Gefolgt von git commit und / oder git push

Wenn dieser Fehler auftritt:

```
fatal: Umbenennen von 'directoryToFolder / oldName' fehlgeschlagen: Ungültiges
Argument
```

Verwenden Sie den folgenden Befehl:

```
git mv directoryToFolder/oldName temp && git mv temp directoryToFolder/newName
```

Umbenennen einer lokalen Niederlassung

Sie können die Verzweigung im lokalen Repository mit folgendem Befehl umbenennen:

```
git branch -m old_name new_name
```

benennen Sie einen lokalen und einen entfernten Zweig um

Der einfachste Weg ist, die lokale Filiale auschecken zu lassen:

```
git checkout old_branch
```

Benennen Sie dann den lokalen Zweig um, löschen Sie die alte Fernbedienung und legen Sie den neuen umbenannten Zweig als Upstream fest:

```
git branch -m new_branch
git push origin :old_branch
git push --set-upstream origin new_branch
```

Umbenennung online lesen: https://riptutorial.com/de/git/topic/1814/umbenennung

Kapitel 53: Umstellung auf Git

Examples

Migrieren Sie von SVN zu Git mit dem Atlassian-Konvertierungsprogramm

Laden Sie das Atlassian-Konvertierungsprogramm hier herunter. Für dieses Dienstprogramm ist Java erforderlich. Stellen Sie daher sicher, dass die Java Runtime Environment- JRE auf dem Rechner installiert ist, auf dem Sie die Konvertierung durchführen möchten.

java -jar svn-migration-scripts.jar verify Sie mit dem Befehl java -jar svn-migration-scripts.jar verify ob auf Ihrem Computer eines der Programme fehlt, die zum Abschluss der Konvertierung erforderlich sind. Insbesondere überprüft dieser Befehl die Dienstprogramme Git, Subversion und git-svn . Außerdem wird überprüft, ob Sie die Migration in einem Dateisystem mit Groß- und Kleinschreibung durchführen. Die Migration zu Git sollte in einem Dateisystem durchgeführt werden, bei dem die Groß- und Kleinschreibung beachtet wird, um eine Beschädigung des Repository zu vermeiden.

Als Nächstes müssen Sie eine Autorendatei erstellen. Subversion verfolgt Änderungen nur anhand des Benutzernamens des Committers. Git verwendet jedoch zwei Informationen, um einen Benutzer zu unterscheiden: einen echten Namen und eine E-Mail-Adresse. Der folgende Befehl generiert eine Textdatei, die die Subversion-Benutzernamen ihren Git-Entsprechungen zuordnet:

```
java -jar svn-migration-scripts.jar authors <svn-repo> authors.txt
```

Dabei ist <svn-repo> die URL des Subversion-Repositorys, das Sie konvertieren möchten. Nachdem Sie diesen Befehl ausgeführt haben, werden die Identifikationsinformationen der Mitwirkenden in authors.txt . Die E-Mail-Adressen haben die Form <username>@mycompany.com . In der Autorendatei müssen Sie den Standardnamen jeder Person (der standardmäßig zu ihrem Benutzernamen wurde) manuell in ihren tatsächlichen Namen ändern. Stellen Sie sicher, dass Sie auch alle E-Mail-Adressen auf Korrektheit prüfen, bevor Sie fortfahren.

Mit dem folgenden Befehl wird ein svn-Repo als Git-Befehl geklont:

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

Dabei ist <svn-repo> dieselbe Repository-URL wie oben und <git-repo-name> der Ordnername im aktuellen Verzeichnis, in das das Repository geklont werden soll. Es gibt einige Überlegungen, bevor Sie diesen Befehl verwenden:

- Das Flag --stdlayout von oben teilt Git mit, dass Sie ein Standardlayout mit Ordnern trunk , branches und tags . Subversion - Repositorys mit Nicht-Standard - Layouts benötigen Sie die Standorte der angeben trunk - Ordner, einen / alle branch Ordner und die tags Ordner. Dies kann durch das folgende Beispiel erreicht werden: git svn clone --trunk=/trunk --branches=/branches --branches=/bugfixes --tags=/tags --authors-file=authors.txt <svn-repo> <git-repo-name> .
- Dieser Befehl kann je nach Größe des Repos mehrere Stunden dauern.
- Um die Konvertierungszeit für große Repositorys zu verkürzen, kann die Konvertierung direkt auf dem Server ausgeführt werden, auf dem sich das Subversion-Repository befindet, um den Netzwerk-Overhead zu beseitigen.

git svn clone importiert die Subversion-Zweige (und den Trunk) als Remote Branches einschließlich Subversion-Tags (Remote Branches, denen das tags/ vorangestellt ist). Um diese in tatsächliche Zweige und Tags zu konvertieren, führen Sie die folgenden Befehle auf einem Linux-Computer in der angegebenen Reihenfolge aus. Nach dem Ausführen sollte git branch -a die richtigen git tag -l , und git tag -l sollte die Repository-Tags anzeigen.

```
git for-each-ref refs/remotes/origin/tags | cut -d / -f 5- | grep -v @ | while read tagname;
do git tag $tagname origin/tags/$tagname; git branch -r -d origin/tags/$tagname; done
git for-each-ref refs/remotes | cut -d / -f 4- | grep -v @ | while read branchname; do git
```

```
branch "$branchname" "refs/remotes/origin/$branchname"; git branch -r -d "origin/$branchname";
done
```

Die Konvertierung von SVN zu Git ist jetzt abgeschlossen! push Sie einfach Ihr lokales Repo auf einen Server, und Sie können weiterhin mit Git beitragen und einen vollständig konservierten Versionsverlauf von svn haben.

SubGit

Mit SubGit kann ein SVN-Repository einmalig in git importiert werden.

```
$ subgit import --non-interactive --svn-url http://svn.my.co/repos/myproject myproject.git
```

Migrieren Sie von SVN zu Git mit svn2git

svn2git ist ein Ruby-Wrapper für gits native SVN-Unterstützung durch git-svn . Er hilft Ihnen bei der Migration von Projekten von Subversion nach Git und speichert den Verlauf (einschließlich Stamm, Tags und Zweigverlauf).

Beispiele

So migrieren Sie ein svn-Repository mit dem Standardlayout (dh Verzweigungen, Tags und Trunk auf Stammebene des Repositorys):

```
$ svn2git http://svn.example.com/path/to/repo
```

So migrieren Sie ein svn-Repository, das sich nicht im Standardlayout befindet:

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches
branches-dir
```

--notrunk Sie keine Zweige, Tags oder --notrunk migrieren möchten (oder nicht), können Sie die Optionen --notrunk , --nobranches und --notags .

Beispiel: $ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches nur die Trunk-Historie.

Um den für Ihr neues Repository erforderlichen Speicherplatz zu reduzieren, möchten Sie möglicherweise alle Verzeichnisse oder Dateien ausschließen, die Sie hinzugefügt haben, während Sie nicht vorhanden waren (z. B. Verzeichnis oder Archive erstellen):

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

Post-Migration-Optimierung

Wenn Sie bereits einige tausend Commits (oder mehr) in Ihrem neu erstellten git-Repository haben, möchten Sie möglicherweise den Speicherplatz reduzieren, bevor Sie Ihr Repository auf eine Remote-Station verschieben. Dies kann mit folgendem Befehl erfolgen:

```
$ git gc --aggressive
```

Hinweis: Der vorherige Befehl kann bei großen Repositorys (zehntausende Commits und / oder Hunderte von Megabytes History) bis zu mehreren Stunden dauern.

Migrieren Sie von Team Foundation Version Control (TFVC) zu Git

Sie können von der Team Foundation-Versionskontrolle zu git migrieren, indem Sie ein Open-Source-Tool namens Git-TF verwenden. Durch die Migration wird auch Ihr vorhandener Verlauf

übertragen, indem Sie tfs-checkins in git-Commits konvertieren.

Führen Sie die folgenden Schritte aus, um Ihre Lösung mithilfe von Git-TF in Git zu integrieren:

Laden Sie Git-TF herunter

Sie können Git-TF von Codeplex herunterladen (und installieren): Git-TF @ Codeplex

Klonen Sie Ihre TFVC-Lösung

Starten Sie Powershell (win) und geben Sie den Befehl ein

```
git-tf clone http://my.tfs.server.address:port/tfs/mycollection
'$/myproject/mybranch/mysolution' --deep
```

Der Schalter --deep ist das Schlüsselwort, das zu beachten ist, da Git-Tf angewiesen wird, Ihren Checkin-Verlauf zu kopieren. Sie haben jetzt ein lokales Git-Repository in dem Ordner, von dem aus Sie Ihren Cloe-Befehl aufgerufen haben.

Aufräumen

- Fügen Sie eine .gitignore-Datei hinzu. Wenn Sie Visual Studio verwenden, kann der Editor dies für Sie tun. Andernfalls können Sie dies manuell tun, indem Sie eine vollständige Datei von github / gitignore herunterladen .
- RemoveTFS-Quellcodeverwaltungsbindungen aus der Lösung (alle * .vssscc-Dateien entfernen). Sie können Ihre Lösungsdatei auch ändern, indem Sie GlobalSection (TeamFoundationVersionControl) ... EndClobalSection entfernen

Commit & Push

Schließen Sie Ihre Konvertierung ab, indem Sie Ihr lokales Repository an Ihre Remote-Station übergeben und dies tun.

```
git add .
git commit -a -m "Coverted solution source control from TFVC to Git"

git remote add origin https://my.remote/project/repo.git

git push origin master
```

Mercurial zu Git migrieren

Sie können die folgenden Methoden verwenden, um ein Mercurial Repo in Git zu importieren:

1. Mit schnellem Export :

```
cd
git clone git://repo.or.cz/fast-export.git
git init git_repo
cd git_repo
~/fast-export/hg-fast-export.sh -r /path/to/old/mercurial_repo
git checkout HEAD
```

2. Verwendung von Hg-Git : Eine sehr detaillierte Antwort hier: https://stackoverflow.com/a/31827990/5283213

3. Verwenden des GitHub-Importers : Folgen Sie den (ausführlichen) Anweisungen bei GitHub .

Umstellung auf Git online lesen: https://riptutorial.com/de/git/topic/3026/umstellung-auf-git

Kapitel 54: Unterbäume

Syntax

- git subtree add -P <prefix> <commit>
  - git subtree add -P <prefix> <repository> <ref>
  - git subtree pull -P <prefix> <repository> <ref>
  - git subtree push -P <prefix> <repository> <ref>
  - git subtree merge -P <prefix> <commit>
  - git subtree split -P <prefix> [OPTIONS] [<commit>]

Bemerkungen

Dies ist eine Alternative zur Verwendung eines submodule

Examples

Teilbaum erstellen, ziehen und zurückschicken

Teilbaum erstellen

Fügen Sie ein neues Remote- plugin , das auf das Repository des plugin verweist:

```
git remote add plugin https://path.to/remotes/plugin.git
```

Erstellen Sie dann eine Unterstruktur, in der die neuen Ordnerpräfix- plugins/demo . plugin ist der entfernte Name, und master bezieht sich auf den master-Zweig im Repository des Teilbaums:

```
git subtree add --prefix=plugins/demo plugin master
```

Teilbaum-Updates abrufen

Ziehen Sie normale Commits im Plugin:

```
git subtree pull --prefix=plugins/demo plugin master
```

Backport-Teilbaum-Updates

1. Festlegen von Commits im Superprojekt, die zurückportiert werden sollen:

   ```
   git commit -am "new changes to be backported"
   ```

2. Überprüfen Sie den neuen Zweig zum Zusammenführen, und legen Sie fest, dass das Teilstruktur-Repository verfolgt wird:

   ```
   git checkout -b backport plugin/master
   ```

3. Kirschpick-Backports:

   ```
   git cherry-pick -x --strategy=subtree master
   ```

4. Änderungen zurück zur Plugin-Quelle verschieben:

```
git push plugin backport:master
```

Unterbäume online lesen: https://riptutorial.com/de/git/topic/1634/unterbaume

Kapitel 55: Verhängnis

Examples

Zusammenführen rückgängig machen

Rückgängigmachen einer Zusammenführung noch nicht auf eine Fernbedienung verschoben

Wenn Sie Ihre Zusammenführung noch nicht in das Remote-Repository gepusht haben, können Sie dasselbe Verfahren wie beim Zurücknehmen des Commits ausführen, auch wenn einige geringfügige Unterschiede bestehen.

Ein Reset ist die einfachste Option, da sowohl das Merge-Commit als auch das Commit aus dem Zweig rückgängig gemacht werden. Sie müssen jedoch wissen, auf was SHA zurückgesetzt werden muss. Dies kann schwierig sein, da in Ihrem git log jetzt Commits aus beiden Zweigen git log werden. Wenn Sie auf das falsche Commit zurücksetzen (z. B. ein anderes auf dem anderen Zweig) , **kann dies die festgelegte Arbeit zerstören.**

```
> git reset --hard <last commit from the branch you are on>
```

Oder nehmen Sie an, die Verschmelzung war Ihre letzte Zusage.

```
> git reset HEAD~
```

Ein Revert ist sicherer, da es die festgelegte Arbeit nicht zerstört, sondern mehr Arbeit erfordert, da Sie den Revert rückgängig machen müssen, bevor Sie den Zweig wieder zusammenführen können (siehe nächster Abschnitt).

Rückgängigmachen einer Zusammenführung wurde auf eine Fernbedienung verschoben

Angenommen, Sie führen ein neues Feature zusammen (add-gremlins)

```
> git merge feature/add-gremlins
...
   #Resolve any merge conflicts
> git commit #commit the merge
...
> git push
...
   501b75d..17a51fd  master -> master
```

Anschließend stellen Sie fest, dass das Feature, mit dem Sie gerade zusammengefügt wurden, das System für andere Entwickler brach, es muss sofort rückgängig gemacht werden. Die Korrektur des Features selbst dauert zu lange, sodass Sie die Zusammenführung einfach rückgängig machen möchten.

```
> git revert -m 1 17a51fd
...
> git push
...
   17a51fd..e443799  master -> master
```

Zu diesem Zeitpunkt sind die Gremlins nicht im System und Ihre Entwickler haben aufgehört, Sie anzuschreien. Wir sind jedoch noch nicht fertig. Wenn Sie das Problem mit der Add-Gremlins-Funktion behoben haben, müssen Sie diese Zurücknahme rückgängig machen, bevor Sie sie wieder zusammenführen können.

```
> git checkout feature/add-gremlins
```

```
...
   #Various commits to fix the bug.
> git checkout master
...
> git revert e443799
...
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

Jetzt ist Ihre Funktion erfolgreich hinzugefügt. In Anbetracht der Tatsache, dass Fehler dieses Typs häufig durch Zusammenführungskonflikte verursacht werden, ist ein etwas anderer Arbeitsablauf manchmal hilfreicher, da Sie den Zusammenführungskonflikt in Ihrem Zweig beheben können.

```
> git checkout feature/add-gremlins
...
   #Merge in master and revert the revert right away.  This puts your branch in
   #the same broken state that master was in before.
> git merge master
...
> git revert e443799
...
   #Now go ahead and fix the bug (various commits go here)
> git checkout master
...
   #Don't need to revert the revert at this point since it was done earlier
> git merge feature/add-gremlins
...
   #Fix any merge conflicts introduced by the bug fix
> git commit #commit the merge
...
> git push
```

Reflog verwenden

Wenn Sie eine Rebase vermasseln, besteht die Möglichkeit, erneut zu beginnen, zum Commit (Pre-Rebase). Sie können dies mit reflog tun (das enthält den Verlauf reflog die Sie in den letzten 90 Tagen erstellt haben - dies kann konfiguriert werden):

```
$ git reflog
4a5cbb3 HEAD@{0}: rebase finished: returning to refs/heads/foo
4a5cbb3 HEAD@{1}: rebase: fixed such and such
904f7f0 HEAD@{2}: rebase: checkout upstream/master
3cbe20a HEAD@{3}: commit: fixed such and such
...
```

Sie können das Festschreiben sehen, bevor die Rebase HEAD@{3} (Sie können den Hash auch auschecken):

```
git checkout HEAD@{3}
```

Jetzt erstellen Sie einen neuen Zweig / löschen den alten Zweig / versuchen Sie es erneut.

Sie können auch direkt auf einen Punkt in Ihrem reflog zurücksetzen. reflog Sie reflog jedoch nur, wenn Sie zu 100% sicher sind, dass Sie das tun möchten:

```
git reset --hard HEAD@{3}
```

Dadurch wird der aktuelle Git-Baum so eingestellt, dass er zu dem Zeitpunkt passt, an dem er sich zu diesem Zeitpunkt befand (siehe Änderungen rückgängig machen).

Dies kann verwendet werden, wenn Sie vorübergehend sehen, wie gut ein Zweig funktioniert, wenn er auf einen anderen Zweig umgestaltet wird, die Ergebnisse jedoch nicht beibehalten werden sollen.

Rückkehr zu einem vorherigen Commit

Um zu einem vorherigen Commit zurückzukehren, suchen Sie zunächst den Hash des Commits mithilfe von `git log` .

Um vorübergehend zu diesem Commit zurückzukehren, entfernen Sie Ihren Kopf mit:

```
git checkout 789abcd
```

Dies bringt Sie zur 789abcd . Sie können jetzt zusätzlich zu diesem alten Commit neue Commits vornehmen, ohne den Zweig zu beeinflussen, in dem sich Ihr Kopf befindet. Alle Änderungen können mit `branch oder checkout -b` in einen richtigen Zweig geändert werden.

So gehen Sie zu einem vorherigen Commit zurück und behalten die Änderungen bei:

```
git reset --soft 789abcd
```

So machen Sie den ***letzten*** Commit rückgängig:

```
git reset --soft HEAD~
```

Um nach einem bestimmten Commit vorgenommene Änderungen dauerhaft zu verwerfen, verwenden Sie:

```
git reset --hard 789abcd
```

Änderungen, die nach dem ***letzten*** Commit vorgenommen wurden, dauerhaft verwerfen:

```
git reset --hard HEAD~
```

Achtung: Während Sie die verworfenen Commits mithilfe von reflog und reset wiederherstellen können, können nicht reflog Änderungen nicht wiederhergestellt werden. Verwenden Sie `git stash; git reset` statt `git reset --hard` um sicher zu gehen.

Änderungen rückgängig machen

Machen Sie Änderungen an einer Datei oder einem Verzeichnis in der **Arbeitskopie** rückgängig.

```
git checkout -- file.txt
```

Wird über alle Dateipfade rekursiv aus dem aktuellen Verzeichnis verwendet, werden alle Änderungen in der Arbeitskopie rückgängig gemacht.

```
git checkout -- .
```

Um nur Teile der Änderungen rückgängig zu machen, verwenden Sie `--patch` . Sie werden für jede Änderung gefragt, ob sie rückgängig gemacht werden soll oder nicht.

```
git checkout --patch -- dir
```

Änderungen rückgängig machen, die dem **Index** hinzugefügt wurden.

```
git reset --hard
```

Ohne das Flag --hard wird ein Soft-Reset durchgeführt.

Mit lokalen Commits, die Sie noch auf eine Fernbedienung verschieben müssen, können Sie auch einen Soft-Reset durchführen. Sie können also die Dateien und dann die Commits überarbeiten.

```
git reset HEAD~2
```

Das obige Beispiel würde Ihre letzten beiden Commits abwickeln und die Dateien in Ihre Arbeitskopie zurückbringen. Sie können dann weitere Änderungen und neue Commits vornehmen.

Achtung: Bei allen diesen Vorgängen werden Ihre Änderungen mit Ausnahme von Soft-Resets dauerhaft gelöscht. Verwenden Sie für eine sicherere Option git stash -p bzw. git stash . Sie können später mit stash pop rückgängig machen oder mit stash drop für immer löschen.

Wiederherstellen einiger vorhandener Commits

Verwenden Sie git revert, um vorhandene Commits rückgängig zu machen, insbesondere wenn diese Commits in ein Remote-Repository verschoben wurden. Es zeichnet einige neue Commits auf, um den Effekt einiger früherer Commits umzukehren, die Sie ohne Umschreiben des Verlaufs sicher verschieben können.

Verwenden **Sie keine** git push --force , wenn Sie die opprobrium aller anderen Nutzer dieses Repository bringen möchten. Schreiben Sie niemals öffentliche Geschichte neu.

Wenn Sie beispielsweise gerade einen Commit mit einem Fehler nach oben verschoben haben und ihn zurücksetzen müssen, gehen Sie wie folgt vor:

```
git revert HEAD~1
git push
```

Jetzt können Sie das Zurücksetzungs-Commit lokal rückgängig machen, Ihren Code korrigieren und den guten Code pushen:

```
git revert HEAD~1
work .. work .. work ..
git add -A .
git commit -m "Update error code"
git push
```

Wenn das Commit, das Sie zurücksetzen möchten, bereits weiter hinten in der Historie ist, können Sie einfach den Commit-Hash übergeben. Git erstellt ein Gegen-Commit, um Ihr ursprüngliches Commit rückgängig zu machen, das Sie sicher auf Ihre Fernbedienung verschieben können.

```
git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc
git push
```

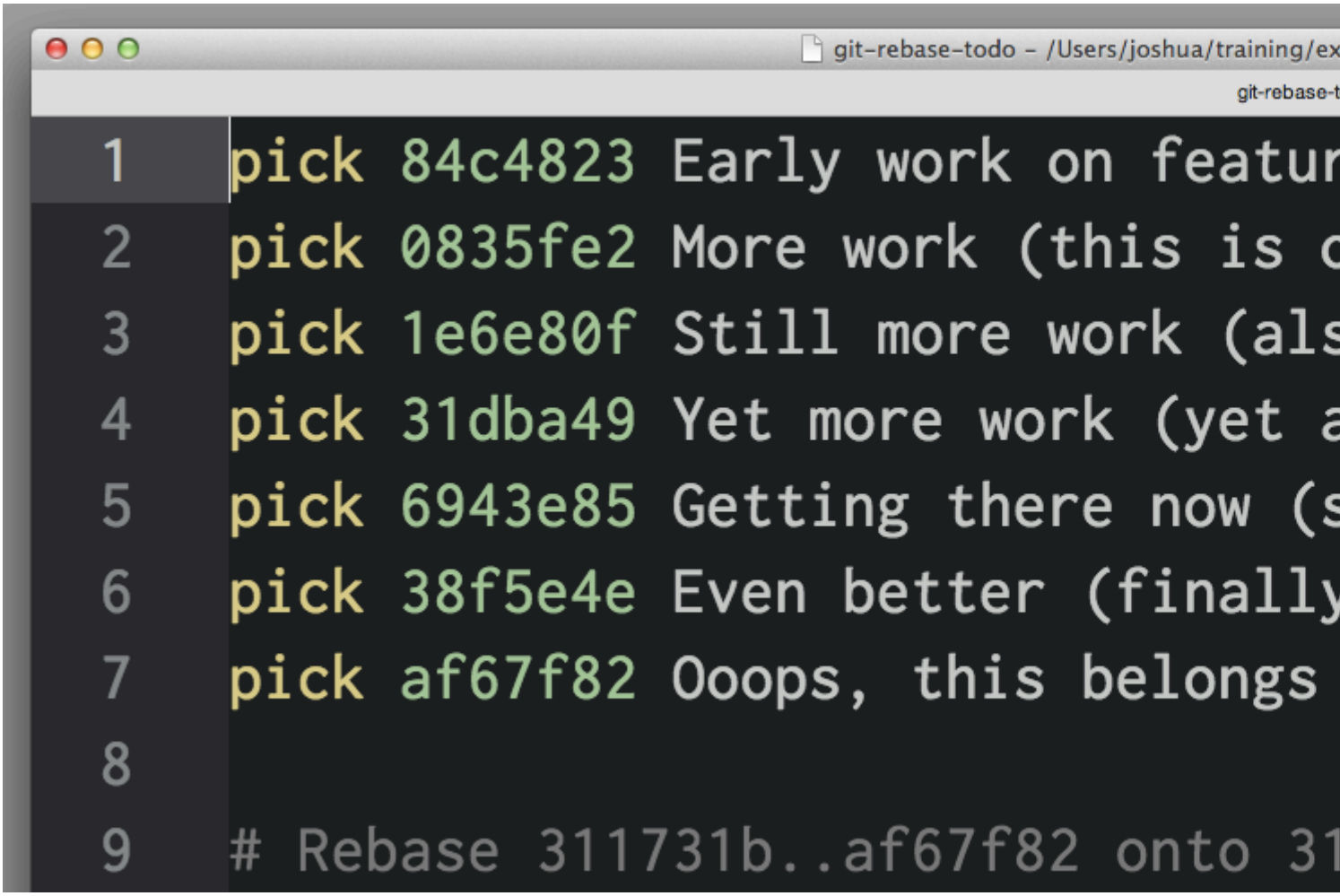
Eine Reihe von Commits rückgängig machen / wiederholen

Angenommen, Sie möchten ein Dutzend Commits rückgängig machen und nur einige davon.

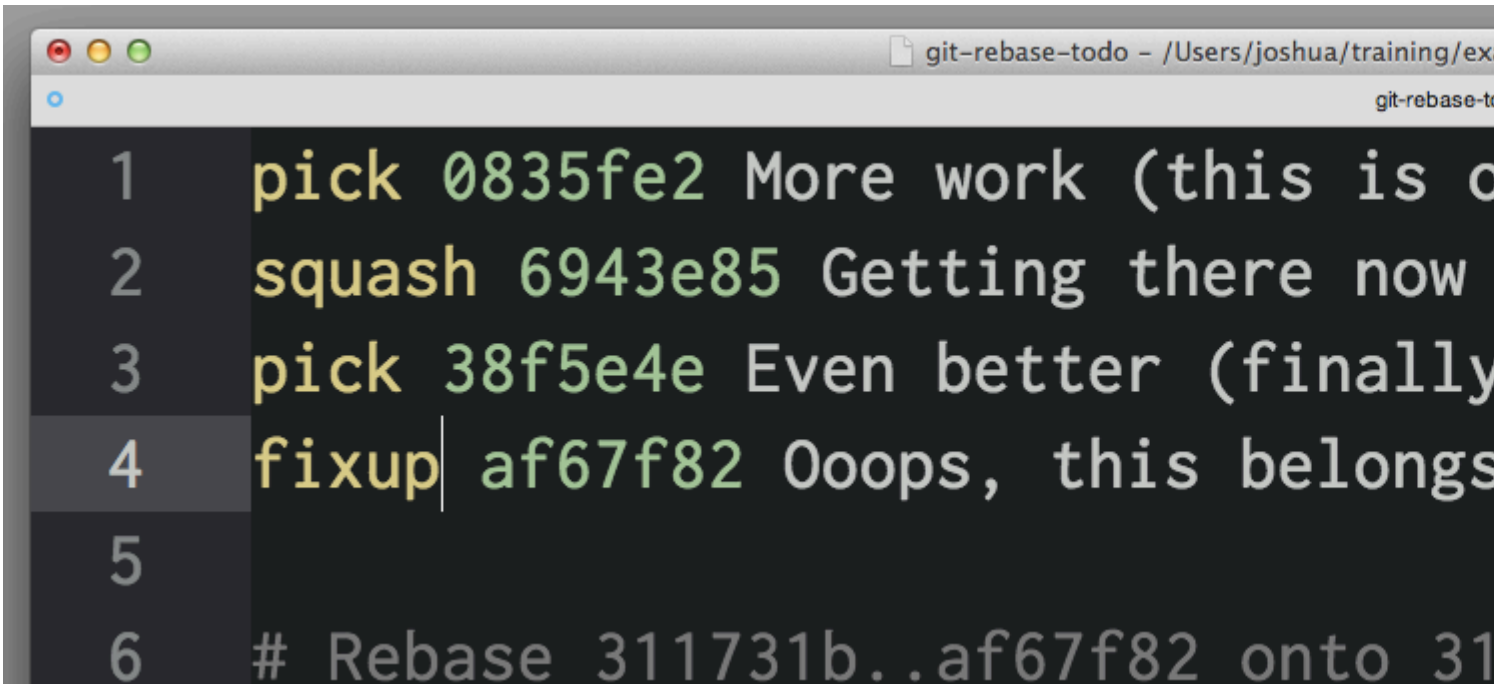
```
git rebase -i <earlier SHA>
```

-i versetzt Rebase in den "interaktiven Modus". Es fängt wie die oben diskutierte Rebase an, aber bevor Commits erneut abgespielt werden, wird sie angehalten und Sie können jedes Commit während der Wiedergabe sanft ändern. *rebase -i* wird in Ihrem Standard-Texteditor geöffnet, und

eine Liste von Commits wird angewendet:



Um ein Commit zu löschen, löschen Sie einfach diese Zeile in Ihrem Editor. Wenn Sie die fehlerhaften Commits nicht mehr in Ihrem Projekt verwenden möchten, können Sie die obigen Zeilen 1 und 3-4 löschen. Wenn Sie zwei Commits miteinander kombinieren möchten, können Sie die Befehle squash oder fixup



Verhängnis online lesen: https://riptutorial.com/de/git/topic/285/verhangnis

Kapitel 56: Verstauen

Syntax

- `git stash list [<options>]`
  - `git stash show [<stash>]`
  - `git stash drop [-q|--quiet] [<stash>]`
  - `git stash ( pop | apply ) [--index] [-q|--quiet] [<stash>]`
  - `git stash branch <branchname> [<stash>]`
  - `git stash [save [-p|--patch] [-k|--[no-]keep-index] [-q|--quiet] [-u|--include-untracked] [-a|--all] [<message>]]`
  - `git stash clear`
  - `git stash create [<message>]`
  - `git stash store [-m|--message <message>] [-q|--quiet] <commit>`

Parameter

Parameter	Einzelheiten
Show	Zeigen Sie die im Stash aufgezeichneten Änderungen als Differenz zwischen dem abgelegten Zustand und dem ursprünglichen übergeordneten Element an. Wenn kein <Stash> angegeben ist, wird der letzte angezeigt.
Liste	Listen Sie Ihre aktuellen Lagerbestände auf. Jeder Stash wird mit seinem Namen aufgelistet (z. B. Stash @ {0} ist der letzte Stash, Stash @ {1} der Vorgänger usw.), der Name des Zweigs, der zum Zeitpunkt des Stash aktuell war, und ein Kurzname Beschreibung des Commits, auf dem der Stash basiert.
Pop	Entfernen Sie einen einzelnen überlagerten Status aus der Stash-Liste und wenden Sie ihn auf den aktuellen Status der Arbeitsstruktur an.
sich bewerben	Wie `pop` , aber entfernen Sie den Status nicht aus der Stash-Liste.
klar	Entfernen Sie alle abgelegten Zustände. Beachten Sie, dass diese Zustände dann beschnitten werden und möglicherweise nicht wiederhergestellt werden können.
fallen	Entfernen Sie einen einzelnen verstauten Zustand aus der Stash-Liste. Wenn kein <Stash> angegeben ist, wird der letzte entfernt. dh stash @ {0}, andernfalls muss <stash> eine gültige stash-Protokollreferenz des Formulars stash @ {<revision>} sein.
erstellen	Erstellen Sie einen Stash (der ein reguläres Commit-Objekt ist) und geben Sie den Objektnamen zurück, ohne ihn irgendwo im ref-Namespace zu speichern. Dies soll für Skripte nützlich sein. Es ist wahrscheinlich nicht der Befehl, den Sie verwenden möchten. Siehe "Speichern" oben.
Geschäft	Speichern Sie einen bestimmten Stash, der mit git stash create erstellt wurde (was ein unbestimmtes Merge-Commit ist) im Stash ref, und aktualisieren Sie den Stash-Refflog. Dies soll für Skripte nützlich sein. Es ist wahrscheinlich

Parameter	Einzelheiten
	nicht der Befehl, den Sie verwenden möchten. Siehe "Speichern" oben.

Bemerkungen

Stashing ermöglicht uns ein sauberes Arbeitsverzeichnis, ohne dass Informationen verloren gehen. Dann ist es möglich, an etwas anderem zu arbeiten und / oder Zweige zu wechseln.

Examples

Was ist Verstauen?

Wenn Sie an einem Projekt arbeiten, befinden Sie sich möglicherweise auf halbem Weg in einer Feature-Zweig-Änderung, wenn ein Fehler beim Master auftritt. Sie sind nicht bereit, Ihren Code festzuschreiben, aber Sie möchten auch Ihre Änderungen nicht verlieren. Hier ist git stash praktisch.

Führen Sie den git status in einem Zweig aus, um Ihre nicht festgeschriebenen Änderungen anzuzeigen:

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

    modified:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Führen Sie dann git stash , um diese Änderungen in einem Stapel zu speichern:

```
(master) $ git stash
Saved working directory and index state WIP on master:
2f2a6e1 Merge pull request #1 from test/test-branch
HEAD is now at 2f2a6e1 Merge pull request #1 from test/test-branch
```

Wenn Sie Dateien zu Ihrem Arbeitsverzeichnis hinzugefügt haben, können diese ebenfalls gespeichert werden. Sie müssen sie nur zuerst inszenieren.

```
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        NewPhoto.c

nothing added to commit but untracked files present (use "git add" to track)
(master) $ git stage NewPhoto.c
(master) $ git stash
Saved working directory and index state WIP on master:
(master) $ git status
On branch master
nothing to commit, working tree clean
```

```
(master) $
```

Ihr Arbeitsverzeichnis enthält jetzt keine Änderungen, die Sie vorgenommen haben. Sie können dies durch erneutes Ausführen des git status :

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

Um den letzten Stash git stash apply , müssen Sie den letzten git stash pop mit git stash pop anwenden *und* entfernen.

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

    modified:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Beachten Sie jedoch, dass das Verstecken sich nicht an den Zweig erinnert, an dem Sie gearbeitet haben. In den obigen Beispielen lag der Benutzer bei **master** . Wenn sie zum **dev-** Zweig wechseln, **dev** und run git stash apply der letzte Stash auf den **dev-** Zweig gelegt.

```
(master) $ git checkout -b dev
Switched to a new branch 'dev'
(dev) $ git stash apply
On branch dev
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   business/com/test/core/actions/Photo.c

no changes added to commit (use "git add" and/or "git commit -a")
```

Versteck schaffen

Speichern Sie den aktuellen Status des Arbeitsverzeichnisses und des Index (auch als Bereitstellungsbereich bezeichnet) in einem Stapel von Stashes.

```
git stash
```

Um alle nicht protokollierten Dateien in den Stash aufzunehmen, verwenden Sie die --include-untracked oder -u .

```
git stash --include-untracked
```

Um eine Nachricht in Ihren Vorrat aufzunehmen, um sie später leichter identifizierbar zu machen

```
git stash save "<whatever message>"
```

Um den Staging-Bereich nach dem --keep-index im aktuellen Status zu --keep-index verwenden Sie die --keep-index oder -k .

```
git stash --keep-index
```

Liste der gespeicherten Stashes

```
git stash list
```

Dadurch werden alle Stapel im Stapel in umgekehrter chronologischer Reihenfolge aufgeführt. Sie erhalten eine Liste, die ungefähr so aussieht:

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Sie können sich auf einen bestimmten Stash beziehen, z. B. auf stash@{1} .

Versteck anzeigen

Zeigt die Änderungen an, die im letzten Speicher gespeichert wurden

```
git stash show
```

Oder ein bestimmtes Versteck

```
git stash show stash@{n}
```

Um den Inhalt der Änderungen anzuzeigen, die für den jeweiligen Bestand gespeichert wurden

```
git stash show -p stash@{n}
```

Versteck entfernen

Entfernen Sie alle Vorräte

```
git stash clear
```

Entfernt den letzten Vorrat

```
git stash drop
```

Oder ein bestimmtes Versteck

```
git stash drop stash@{n}
```

Übernehmen und entfernen

Um den letzten Stash anzuwenden und aus dem Stapel zu entfernen, geben Sie Folgendes ein:

```
git stash pop
```

Um einen bestimmten Stash anzuwenden und ihn aus dem Stack zu entfernen, geben Sie Folgendes ein:

```
git stash pop stash@{n}
```

Übernehmen, ohne es zu entfernen

Wendet den letzten Stapel an, ohne ihn aus dem Stapel zu entfernen

```
git stash apply
```

Oder ein bestimmtes Versteck

```
git stash apply stash@{n}
```

Wiederherstellung früherer Änderungen aus dem Stash

Verwenden Sie, um den neuesten Stand nach dem Ausführen von Git-Stash zu erhalten

```
 git stash apply
```

Um eine Liste Ihrer Vorräte anzuzeigen, verwenden Sie

```
 git stash list
```

Sie erhalten eine Liste, die ungefähr so aussieht

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Wählen Sie einen anderen Git-Stash zum Wiederherstellen mit der Nummer, die für den gewünschten Stash angezeigt wird

```
git stash apply stash@{2}
```

Teilweise Versteck

Wenn Sie nur *einige* Unterschiede in Ihrem Arbeitssatz ablegen möchten, können Sie einen Teileinsatz verwenden.

```
git stash -p
```

Und wählen Sie dann interaktiv aus, welche Teile Sie unterbringen möchten.

Ab Version 2.13.0 können Sie auch den interaktiven Modus vermeiden und mithilfe des neuen **Push-** Schlüsselworts einen Teilspeicher mit einer Pfadspezifikation erstellen.

```
git stash push -m "My partial stash" -- app.config
```

Übernehmen Sie einen Teil des Vorrats mit der Kasse

Sie haben einen Vorrat erstellt und möchten nur einige der Dateien in diesem Bestand prüfen.

```
git checkout stash@{0} -- myfile.txt
```

Interaktives Verstauen

Das Ablegen nimmt den fehlerhaften Zustand Ihres Arbeitsverzeichnisses - also Ihre geänderten, nachverfolgten Dateien und inszenierten Änderungen - und speichert es auf einem Stapel nicht

abgeschlossener Änderungen, die Sie jederzeit erneut anwenden können.

Nur veränderte Dateien ablegen:

Angenommen, Sie möchten die bereitgestellten Dateien nicht stauen und nur die geänderten Dateien stauen, sodass Sie Folgendes verwenden können:

```
git stash --keep-index
```

Dadurch werden nur die geänderten Dateien gespeichert.

Nicht gespeicherte Dateien stauen:

Stash speichert niemals die nicht protokollierten Dateien, sondern nur die geänderten und bereitgestellten Dateien. Angenommen, Sie müssen auch die nicht aufgespürten Dateien stauen, dann können Sie Folgendes verwenden:

```
git stash -u
```

Dadurch werden die nicht protokollierten, bereitgestellten und geänderten Dateien erfasst.

Behalte nur bestimmte Änderungen:

Angenommen, Sie müssen nur einen Teil des Codes aus der Datei oder nur einige Dateien aus allen geänderten und gespeicherten Dateien verstauen. Dann können Sie dies folgendermaßen tun:

```
git stash --patch
```

Git speichert nicht alles, was geändert wurde, sondern fordert Sie interaktiv auf, welche der Änderungen Sie speichern möchten und welche Sie in Ihrem Arbeitsverzeichnis behalten möchten.

Verschieben Sie Ihre laufenden Arbeiten in einen anderen Zweig

Wenn Sie während der Arbeit feststellen, dass Sie sich in einem falschen Zweig befinden und noch keine Commits erstellt haben, können Sie Ihre Arbeit ganz einfach mithilfe von Stashing zur Korrektur des Zweigs verschieben:

```
git stash
git checkout correct-branch
git stash pop
```

Denken Sie daran, dass git stash pop den letzten Stash anwendet und ihn aus der Stash-Liste löscht. Um den Vorrat in der Liste zu halten und nur für einen Zweig zu gelten, können Sie Folgendes verwenden:

```
git stash apply
```

Stelle einen heruntergefallenen Vorrat wieder her

Wenn Sie es gerade erst geknackt haben und das Terminal noch geöffnet ist, wird der Hash-Wert immer noch von git stash pop auf dem Bildschirm gedruckt:

```
$ git stash pop
[...]
Dropped refs/stash@{0} (2ca03e22256be97f9e40f08e6d6773c7d41dbfd1)
```

(Beachten Sie, dass Git-Stash-Drop auch dieselbe Linie erzeugt.)

Ansonsten können Sie es folgendermaßen finden:

```
git fsck --no-reflog | awk '/dangling commit/ {print $3}'
```

Daraufhin werden alle Commits an den Tipps Ihres Commit-Graphen angezeigt, auf die von keinem Zweig oder Tag mehr verwiesen wird. Jedes verlorene Commit, einschließlich jedes von Ihnen erstellten Stash-Commits, befindet sich irgendwo in diesem Diagramm.

Der einfachste Weg, das gewünschte Stash-Commit zu finden, besteht wahrscheinlich darin, diese Liste an gitk zu gitk :

```
gitk --all $( git fsck --no-reflog | awk '/dangling commit/ {print $3}' )
```

Dadurch wird ein Repository-Browser gestartet, der Ihnen *jedes Commit im Repository zeigt* , unabhängig davon, ob es erreichbar ist oder nicht.

Sie können gitk dort durch etwas wie git log --graph --oneline --decorate wenn Sie eine nette Grafik auf der Konsole einer separaten GUI-App vorziehen.

Suchen Sie nach Commit-Nachrichten in diesem Formular:

WIP on *somebranch* : *commithash Eine alte Commit-Nachricht*

Sobald Sie den Hash des gewünschten Commits kennen, können Sie ihn als Vorrat verwenden:

```
git stash apply $stash_hash
```

Oder Sie können das Kontextmenü in gitk , um Verzweigungen für alle nicht erreichbaren Commits zu erstellen, an denen Sie interessiert sind. Danach können Sie mit all den üblichen Werkzeugen machen, was Sie möchten. Wenn Sie fertig sind, blasen Sie diese Äste einfach wieder weg.

Verstauen online lesen: https://riptutorial.com/de/git/topic/1440/verstauen

Kapitel 57: Verwenden einer .gitattributes-Datei

Examples

Zeilenende-Normalisierung deaktivieren

Erstellen Sie eine .gitattributes Datei im Projektstamm, die .gitattributes enthält:

```
* -text
```

Dies entspricht der Einstellung von core.autocrlf = false .

Automatische Normalisierung der Leitungsenden

Erstellen Sie eine .gitattributes Datei im Projektstamm, die .gitattributes enthält:

```
* text=auto
```

Dies führt dazu, dass alle Textdateien (wie von Git angegeben) mit LF festgeschrieben, jedoch gemäß den Standardeinstellungen des Host-Betriebssystems ausgecheckt werden.

Dies entspricht den empfohlenen core.autocrlf von core.autocrlf :

- input unter Linux / macOS
- true unter Windows

Identifizieren Sie binäre Dateien

Git ist ziemlich gut darin, binäre Dateien zu identifizieren, aber Sie können explizit angeben, welche Dateien binär sind. Erstellen Sie eine .gitattributes Datei im Projektstamm, die .gitattributes enthält:

```
*.png binary
```

binary ist ein integriertes -diff -merge -text .

Vorgefüllte .gitattribute-Vorlagen

Wenn Sie nicht sicher sind, welche Regeln in Ihrer .gitattributes Datei aufgeführt werden sollen, oder Sie nur allgemein akzeptierte Attribute zu Ihrem Projekt hinzufügen möchten, können Sie eine .gitattributes Datei unter folgender .gitattributes auswählen oder generieren:

- https://gitattributes.io/
- https://github.com/alexkaratarakis/gitattributes

Verwenden einer .gitattributes-Datei online lesen:
https://riptutorial.com/de/git/topic/1269/verwenden-einer--gitattributes-datei

Kapitel 58: Verzweigung

Syntax

- `git branch [--set-upstream | --track | --no-track] [-l] [-f] <branchname> [<start-point>]`
  - `git branch (--set-upstream-to=<upstream> | -u <upstream>) [<branchname>]`
  - `git branch --unset-upstream [<branchname>]`
  - `git branch (-m | -M) [<oldbranch>] <newbranch>`
  - `git branch (-d | -D) [-r] <branchname>…`
  - `git branch --edit-description [<branchname>]`
  - `git branch [--color[=<when>] | --no-color] [-r | -a] [--list] [-v [--abbrev=<length> | --no-abbrev]] [--column[=<options>] | --no-column] [(--merged | --no-merged | --contains) [<commit>]] [--sort=<key>] [--points-at <object>] [<pattern>…]`

Parameter

Parameter	Einzelheiten
-d, --delete	Einen Zweig löschen. Der Zweig muss in seinem Upstream-Zweig oder in `HEAD` vollständig zusammengeführt werden, wenn kein Upstream mit `--track` oder `--set-upstream`
-D	`--delete --force` für `--delete --force`
-m, -bewegen	Verschieben / Umbenennen eines Zweigs und des entsprechenden Reflogs
-M	`--move --force` für `--move --force`
-r, --remotes	Listen oder löschen Sie (falls mit -d verwendet) die Remote-Tracking-Zweige
-a, --all	Listen Sie sowohl Fernverfolgungszweige als auch lokale Zweigstellen auf
--Liste	Aktivieren Sie den Listenmodus. `git branch <pattern>` versucht, einen Zweig zu erstellen. Verwenden Sie `git branch --list <pattern>` um übereinstimmende Zweige `git branch --list <pattern>`
--set-upstream	Wenn der angegebene Zweig noch nicht existiert oder `--force` angegeben wurde, `--track` genau wie `--track` . Ansonsten wird die Konfiguration wie --track beim Erstellen der Verzweigung festgelegt, außer dass die Verzweigungspunkte nicht geändert werden

Bemerkungen

Jedes Git-Repository hat eine oder mehrere *Niederlassungen* . Eine Verzweigung ist eine benannte Referenz auf den HEAD einer Folge von Commits.

Ein Git-Repo verfügt über einen *aktuellen* Zweig (durch ein * in der Liste der Zweignamen, die vom Befehl git branch gedruckt werden). Wenn Sie mit dem Befehl git commit ein neues Commit erstellen, wird das neue Commit zum HEAD des aktuellen Zweigs Der vorherige HEAD wird das übergeordnete Element des neuen Commits.

Ein neuer Zweig hat den gleichen HEAD wie der Zweig, aus dem er erstellt wurde, bis etwas an den neuen Zweig übergeben wird.

Examples

Auflistung der Niederlassungen

Git bietet mehrere Befehle zum Auflisten von Zweigen. Alle Befehle verwenden die Funktion von `git branch` , die eine Liste bestimmter Verzweigungen bereitstellt, abhängig davon, welche Optionen in der Befehlszeile stehen. Git zeigt, wenn möglich, den aktuell ausgewählten Zweig mit einem Stern daneben an.

Tor	Befehl
Lokale Filialen auflisten	`git branch`
Listen Sie lokale Niederlassungen ausführlich auf	`git branch -v`
Liste entfernter und lokaler Niederlassungen	`git branch -a` ODER `git branch --all`
Liste entfernter und lokaler Niederlassungen (ausführlich)	`git branch -av`
Liste entfernter Zweige	`git branch -r`
Listen Sie entfernte Zweige mit dem neuesten Commit auf	`git branch -rv`
Listen Sie zusammengeführte Zweige auf	`git branch --merged`
Listen Sie nicht zusammengeführte Zweige auf	`git branch --no-merged`
Liste der Zweige mit Commit	`git branch --contains [<commit>]`

Anmerkungen :

- Durch Hinzufügen eines zusätzlichen v zu -v z. B. $ git branch -avv oder $ git branch -vv wird auch der Name des Upstream-Zweigs gedruckt.
- Zweige in roter Farbe sind entfernte Zweige

Neue Filialen anlegen und auschecken

Um einen neuen Zweig zu erstellen, während Sie auf dem aktuellen Zweig bleiben, verwenden Sie:

```
git branch <name>
```

Im Allgemeinen darf der Zweigname keine Leerzeichen enthalten und unterliegt anderen hier aufgeführten Spezifikationen. So wechseln Sie zu einem vorhandenen Zweig:

```
git checkout <name>
```

So erstellen Sie einen neuen Zweig und wechseln zu ihm:

```
git checkout -b <name>
```

Verwenden Sie einen der folgenden Befehle, um einen Zweig an einem anderen Punkt als dem letzten Commit des aktuellen Zweigs (auch als HEAD bezeichnet) zu erstellen:

```
git branch <name> [<start-point>]
git checkout -b <name> [<start-point>]
```

Der <start-point> beliebig seine Revision zu GIT (zB ein anderer Zweigname, commit SHA oder eine symbolische Referenz wie Kopf oder einen Tag - Namen) bekannt:

```
git checkout -b <name> some_other_branch
git checkout -b <name> af295
git checkout -b <name> HEAD~5
git checkout -b <name> v1.0.5
```

So erstellen Sie eine Verzweigung aus einer entfernten Verzweigung (der Standardwert <remote_name> ist Ursprung):

```
git branch <name> <remote_name>/<branch_name>
git checkout -b <name> <remote_name>/<branch_name>
```

Wenn ein bestimmter Zweigname nur auf einer Fernbedienung vorhanden ist, können Sie einfach verwenden

```
git checkout -b <branch_name>
```

das ist äquivalent zu

```
git checkout -b <branch_name> <remote_name>/<branch_name>
```

Manchmal müssen Sie möglicherweise einige Ihrer letzten Commits in einen neuen Zweig verschieben. Dies kann durch Verzweigung und "Zurückrollen" wie folgt erreicht werden:

```
git branch <new_name>
git reset --hard HEAD~2 # Go back 2 commits, you will lose uncommitted work.
git checkout <new_name>
```

Hier ist eine illustrative Erklärung dieser Technik:

```
 Initial state       After git branch <new_name>    After git reset --hard HEAD~2
                             newBranch                      newBranch
                                 ↓                              ↓
A-B-C-D-E (HEAD)         A-B-C-D-E (HEAD)               A-B-C-D-E (HEAD)
        ↑                        ↑                              ↑
      master                   master                         master
```

Einen Zweig lokal löschen

```
$ git branch -d dev
```

Löscht den Zweig mit dem Namen dev *wenn* seine Änderungen mit einem anderen Zweig zusammengeführt werden und nicht verloren gehen. Wenn der dev Zweig Änderungen enthält, die noch nicht zusammengeführt wurden und verloren gingen, schlägt git branch -d fehl:

```
$ git branch -d dev
```

```
error: The branch 'dev' is not fully merged.
If you are sure you want to delete it, run 'git branch -D dev'.
```

Mit der Warnmeldung können Sie das Löschen des Zweigs erzwingen (und alle nicht zusammengeführten Änderungen in diesem Zweig verlieren), indem Sie das Flag -D verwenden:

```
$ git branch -D dev
```

Überprüfen Sie einen neuen Zweig, der einen entfernten Zweig verfolgt

Es gibt drei Möglichkeiten , eine Niederlassung zu schaffen feature , die die Remote - Zweigspuren origin/feature :

- git checkout --track -b feature origin/feature ,
- git checkout -t origin/feature ,
- git checkout feature - vorausgesetzt, es gibt keinen lokalen feature Zweig und es gibt nur einen Remote mit dem feature Zweig.

So richten Sie Upstream ein, um den Remote-Zweig zu verfolgen:

- git branch --set-upstream-to=<remote>/<branch> <branch>
  - git branch -u <remote>/<branch> <branch>

woher:

- <remote> kann sein: origin , develop oder vom Benutzer erstellt,
- <branch> ist der Zweig des Benutzers, der auf der Ferne verfolgt werden soll.

So überprüfen Sie, welche Remote-Zweigstellen Ihre lokalen Zweigstellen verfolgen:

- git branch -vv

Einen Zweig umbenennen

Benennen Sie den ausgecheckten Zweig um:

```
git branch -m new_branch_name
```

Einen anderen Zweig umbenennen:

```
git branch -m branch_you_want_to_rename new_branch_name
```

Überschreiben Sie eine einzelne Datei im aktuellen Arbeitsverzeichnis mit derselben Datei aus einem anderen Zweig

Die ausgecheckte Datei **überschreibt die** noch nicht vorgenommenen Änderungen, die Sie in dieser Datei vorgenommen haben.

Dieser Befehl file.example die Datei file.example (die sich im Verzeichnispfad path/to/) aus und **überschreibt alle Änderungen, die** Sie an dieser Datei vorgenommen haben.

```
git checkout some-branch path/to/file
```

some-branch kann alles sein , tree-ish zu git (siehe bekannt Revision Auswahl und gitrevisions für weitere Details)

Sie müssen -- vor dem Pfad hinzufügen, wenn Ihre Datei mit einer Datei verwechselt werden kann

(ansonsten optional). Nach dem -- können keine weiteren Optionen geliefert werden.

```
git checkout some-branch -- some-file
```

Die zweite some-file ist in diesem Beispiel eine Datei.

Löschen Sie einen entfernten Zweig

Um einen Zweig im origin Remote-Repository zu löschen, können Sie für Git Version 1.5.0 und neuer verwenden

```
git push origin :<branchName>
```

Ab Git Version 1.7.0 können Sie einen Remote-Zweig mit löschen

```
git push origin --delete <branchName>
```

So löschen Sie einen lokalen Remote-Tracking-Zweig:

```
git branch --delete --remotes <remote>/<branch>
git branch -dr <remote>/<branch> # Shorter

git fetch <remote> --prune # Delete multiple obsolete tracking branches
git fetch <remote> -p      # Shorter
```

Einen Zweig lokal löschen. Beachten Sie, dass dadurch die Verzweigung nicht gelöscht wird, wenn sich nicht zusammengeführte Änderungen ergeben:

```
git branch -d <branchName>
```

So löschen Sie einen Zweig, auch wenn Änderungen nicht zusammengeführt wurden:

```
git branch -D <branchName>
```

Erstellen Sie einen verwaisten Zweig (dh Zweig ohne übergeordnetes Commit)

```
git checkout --orphan new-orphan-branch
```

> Das erste Commit für diesen neuen Zweig wird keine Eltern haben, und es wird die Wurzel einer neuen Geschichte sein, die völlig von allen anderen Zweigen und Commits getrennt ist.

Quelle

Zweig zur Fernbedienung drücken

Verwenden Sie diese Option, um in Ihrem lokalen Zweig vorgenommene Commits in ein Remote-Repository zu verschieben.

Der Befehl git push benötigt zwei Argumente:

- Ein entfernter Name, z. B. origin
- Ein Zweigname, zum Beispiel master

Zum Beispiel:

```
git push  <REMOTENAME> <BRANCHNAME>
```

Beispielsweise führen Sie in der Regel git push origin master , um Ihre lokalen Änderungen in Ihr Online-Repository zu übernehmen.

Mit -u (kurz für --set-upstream) werden die Tracking-Informationen während des Push- --set-upstream eingerichtet.

```
git push -u <REMOTENAME> <BRANCHNAME>
```

Standardmäßig schiebt git den lokalen Zweig an einen entfernten Zweig mit demselben Namen. Wenn Sie beispielsweise ein lokales new-feature namens new-feature , wenn Sie auf den lokalen Zweig drücken, wird auch ein new-feature Zweig für einen entfernten Zweig erstellt. Wenn Sie einen anderen Namen für den Remote - Zweig verwenden möchten, fügen Sie den Remote - Namen nach dem lokalen Zweignamen, getrennt durch : :

```
git push <REMOTENAME> <LOCALBRANCHNAME>:<REMOTEBRANCHNAME>
```

Verschieben Sie den aktuellen Zweigkopf in ein beliebiges Commit

Ein Zweig ist nur ein Zeiger auf ein Commit, sodass Sie ihn frei verschieben können. aabbcc den Befehl aus, damit sich der Zweig auf das aabbcc bezieht

```
git reset --hard aabbcc
```

Bitte beachten Sie, dass dadurch der aktuelle Commit Ihrer Zweigstelle und damit der gesamte Verlauf überschrieben wird. Durch diesen Befehl könnten Sie etwas Arbeit verlieren. In diesem Fall können Sie das verlorene Commit mit dem Reflog wiederherstellen. Es kann empfohlen werden, diesen Befehl in einem neuen Zweig anstelle des aktuellen Zweigs auszuführen.

Dieser Befehl kann jedoch besonders nützlich sein, wenn Sie neue Basenänderungen vornehmen oder solche Änderungen vornehmen.

Schneller Wechsel zum vorherigen Zweig

Mit können Sie schnell zum vorherigen Zweig wechseln

```
git checkout -
```

In Filialen suchen

Um lokale Zweige aufzulisten, die ein bestimmtes Commit oder Tag enthalten

```
git branch --contains <commit>
```

Lokale und entfernte Zweige auflisten, die ein bestimmtes Commit oder Tag enthalten

```
git branch -a --contains <commit>
```

Verzweigung online lesen: https://riptutorial.com/de/git/topic/415/verzweigung

Kapitel 59: Wiederherstellen

Examples

Wiederherstellen von einem verlorenen Commit

Falls Sie zu einem früheren Commit zurückgekehrt sind und ein neueres Commit verloren haben, können Sie das verlorene Commit durch Ausführen wiederherstellen

```
git reflog
```

Finden Sie dann Ihr verlorenes Commit und stellen Sie es wieder her

```
git reset HEAD --hard <sha1-of-commit>
```

Stellen Sie eine gelöschte Datei nach einem Commit wieder her

Falls Sie versehentlich einen Löschvorgang für eine Datei festgelegt haben und später feststellen, dass Sie ihn wieder benötigen.

Suchen Sie zuerst die Commit-ID des Commits, das Ihre Datei gelöscht hat.

```
git log --diff-filter=D --summary
```

Gibt Ihnen eine sortierte Zusammenfassung der Commits, die Dateien gelöscht haben.

Fahren Sie dann mit der Wiederherstellung der Datei fort

```
git checkout 81eeccf~1 <your-lost-file-name>
```

(Ersetzen Sie 81eeccf durch Ihre eigene Festschreibungs-ID.)

Wiederherstellen der Datei auf eine frühere Version

Um eine Datei auf eine frühere Version zurückzusetzen, können Sie reset .

```
git reset <sha1-of-commit> <file-name>
```

Wenn Sie bereits lokale Änderungen an der Datei vorgenommen haben (die Sie nicht benötigen!), --hard Sie auch die Option --hard

Einen gelöschten Zweig wiederherstellen

Um einen gelöschten Zweig wiederherzustellen, müssen Sie den Commit finden, der den Kopf Ihres gelöschten Zweigs war, indem Sie ihn ausführen

```
git reflog
```

Sie können den Zweig dann neu erstellen, indem Sie ihn ausführen

```
git checkout -b <branch-name> <sha1-of-commit>
```

Sie können gelöschte Verzweigungen nicht wiederherstellen, wenn gits Garbage Collector unvollständige Commits gelöscht hat - solche ohne Verweise. Halten Sie immer eine Sicherungskopie Ihres Repositorys bereit, insbesondere wenn Sie in einem kleinen Team / einem

proprietären Projekt arbeiten

Wiederherstellen nach einem Reset

Mit Git können Sie die Uhr (fast) immer zurückdrehen

Haben Sie keine Angst davor, mit Befehlen zu experimentieren, die die Geschichte neu schreiben *. Standardmäßig löscht Git Ihre Commits nicht für 90 Tage, und während dieser Zeit können Sie sie problemlos aus dem Reflog wiederherstellen:

```
$ git reset @~3   # go back 3 commits
$ git reflog
c4f708b HEAD@{0}: reset: moving to @~3
2c52489 HEAD@{1}: commit: more changes
4a5246d HEAD@{2}: commit: make important changes
e8571e4 HEAD@{3}: commit: make some changes
... earlier commits ...
$ git reset 2c52489
... and you're back where you started
```

** Achten Sie auf Optionen wie - --hard und --force die jedoch Daten verwerfen können.*
** Vermeiden Sie auch das Umschreiben des Verlaufs in allen Zweigen, in denen Sie zusammenarbeiten.*

Erholen Sie sich von Git-Stash

Verwenden Sie, um den neuesten Stand nach dem Ausführen von Git-Stash zu erhalten

```
git stash apply
```

Um eine Liste Ihrer Vorräte anzuzeigen, verwenden Sie

```
git stash list
```

Sie erhalten eine Liste, die ungefähr so aussieht

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Wählen Sie einen anderen Git-Stash zum Wiederherstellen mit der Nummer, die für den gewünschten Stash angezeigt wird

```
git stash apply stash@{2}
```

Sie können auch "Git Stash Pop" wählen, es funktioniert genauso wie "Git Stash anwenden" wie ..

```
 git stash pop
```

oder

```
 git stash pop stash@{2}
```

Unterschied im Git-Stash anwenden und Git-Stash-Pop ...

Git Stash Pop : - Stash-Daten werden aus dem Stapel der Stash-Liste entfernt.

Ex:-

```
git stash list
```

Sie erhalten eine Liste, die ungefähr so aussieht

```
stash@{0}: WIP on master: 67a4e01 Merge tests into develop
stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Pope jetzt die Daten mit dem Befehl

```
git stash pop
```

Überprüfen Sie erneut, ob eine Liste vorhanden ist

```
git stash list
```

Sie erhalten eine Liste, die ungefähr so aussieht

```
 stash@{0}: WIP on master: 70f0d95 Add user role to localStorage on user login
```

Sie können sehen, dass ein Stash-Datenbestand aus der Stash-Liste entfernt (geknipst) wird und aus @ {1} Stash {@} wurde.

Wiederherstellen online lesen: https://riptutorial.com/de/git/topic/725/wiederherstellen

Kapitel 60: Ziehen

Einführung

Im Gegensatz zum Push-Vorgang mit Git, bei dem Ihre lokalen Änderungen an den Server des zentralen Repositorys gesendet werden, übernimmt das Ziehen mit Git den aktuellen Code auf dem Server und "zieht" ihn vom Server des Repositorys auf Ihren lokalen Computer. In diesem Thema wird der Vorgang des Abrufs von Code aus einem Repository mit Git sowie die Situationen erläutert, die beim Ziehen von anderem Code in die lokale Kopie auftreten können.

Syntax

- `git pull [Optionen [<Repository> [<Refspec> ...]]`

Parameter

Parameter	Einzelheiten
`--quiet`	Keine Textausgabe
`-q`	Abkürzung für `--quiet`
`--verbose`	ausführliche Textausgabe. Wird an Befehle zum Abrufen und Zusammenführen / Zurücksetzen übergeben.
`-v`	Abkürzung für `--verbose`
`--[no-]recurse-submodules[=yes | on-demand | no]`	Neue Commits für Submodule abrufen? (Nicht dass dies kein Pull / Checkout ist)

Bemerkungen

`git pull` führt `git fetch` mit den angegebenen Parametern aus und ruft `git merge` auf, um die abgerufenen Zweigköpfe mit dem aktuellen Zweig zusammenzuführen.

Examples

Aktualisierung mit lokalen Änderungen

Wenn lokale Änderungen vorhanden sind, `git pull` Befehl `git pull` Berichterstellung ab:

> Fehler: Ihre lokalen Änderungen an den folgenden Dateien werden beim Zusammenführen überschrieben

Um zu aktualisieren (wie bei `svn update` mit subversion), können Sie Folgendes ausführen:

```
git stash
git pull --rebase
git stash pop
```

Ein bequemer Weg könnte sein, einen Alias zu definieren mit:

2,9

```
git config --global alias.up '!git stash && git pull --rebase && git stash pop'
```

2,9

```
git config --global alias.up 'pull --rebase --autostash'
```

Als nächstes können Sie einfach verwenden:

```
git up
```

Code von der Fernbedienung abziehen

```
git pull
```

Ziehen, lokal überschreiben

```
git fetch
git reset --hard origin/master
```

Achtung: Bei der Verwendung von begeht verworfen reset --hard gestellt werden kann unter Verwendung von reflog und reset werden immer unbestätigte Änderungen gelöscht.

Ändern Sie origin und master in den Remote- und Zweig, in den Sie zwangsweise ziehen möchten, falls sie anders benannt sind.

Linearer Verlauf beim Ziehen

Neueinstellung beim Ziehen

Wenn Sie neue Commits aus dem Remote-Repository abrufen und lokale Änderungen am aktuellen Zweig vorgenommen haben, führt git automatisch die Remote-Version und Ihre Version zusammen. Wenn Sie die Anzahl der Zusammenführungen in Ihrem Zweig reduzieren möchten, können Sie git anweisen, Ihre Commits in der Remote-Version des Zweigs neu zu definieren.

```
git pull --rebase
```

Machen Sie es zum Standardverhalten

Geben Sie den folgenden Befehl ein, um dies als Standardverhalten für neu erstellte Zweige festzulegen:

```
git config branch.autosetuprebase always
```

So ändern Sie das Verhalten eines vorhandenen Zweigs:

```
git config branch.BRANCH_NAME.rebase true
```

Und

```
git pull --no-rebase
```

So führen Sie einen normalen Zusammenführungszug aus.

Prüfen Sie, ob es schnell vorwärts geht

Um nur das schnelle Weiterleiten des lokalen Zweigs zuzulassen, können Sie Folgendes verwenden:

```
git pull --ff-only
```

Dies zeigt einen Fehler an, wenn der lokale Zweig nicht schnell vorwärts gerollt werden kann und entweder neu basiert oder mit Upstream zusammengeführt werden muss.

Ziehen Sie, "Erlaubnis verweigert"

Es können einige Probleme auftreten, wenn der .git Ordner eine falsche Berechtigung hat. Beheben dieses Problems durch Festlegen des Besitzers des vollständigen .git Ordners. Es kommt manchmal vor, dass ein anderer Benutzer die Rechte des .git Ordners oder der Dateien zieht und ändert.

Um das Problem zu lösen:

```
chown -R youruser:yourgroup .git/
```

Änderungen in ein lokales Repository ziehen

Einfaches Ziehen

Wenn Sie an einem Remote-Repository (z. B. GitHub) mit einer anderen Person arbeiten, möchten Sie Ihre Änderungen irgendwann mit ihnen teilen. Sobald sie gedrückt ihre Änderungen an eine Remote - Repository, können Sie diese Änderungen abrufen , indem Sie aus diesem Repository *ziehen*.

```
git pull
```

Wird es in den meisten Fällen tun.

Ziehen Sie von einer anderen Fernbedienung oder einem Zweig aus

Sie können Änderungen von einer anderen Remote- oder Zweigstelle abrufen, indem Sie deren Namen angeben

```
git pull origin feature-A
```

Die Filiale ziehen feature-A Form origin in Ihre lokale Niederlassung. Beachten Sie, dass Sie direkt einen URL anstelle eines Remote-Namens und einen Objektnamen angeben können, z. B. eine Festschreibungs-SHA anstelle eines Zweignamens.

Manueller Zug

Um das Verhalten eines Git-Pulls zu imitieren, können Sie git fetch dann git merge

```
git fetch origin # retrieve objects and update refs from origin
git merge origin/feature-A # actually perform the merge
```

Dadurch erhalten Sie mehr Kontrolle und können den entfernten Zweig vor dem Zusammenführen prüfen. Tatsächlich können Sie nach dem Abrufen die entfernten Zweige mit git branch -a und sie mit überprüfen

```
git checkout -b local-branch-name origin/feature-A # checkout the remote branch
# inspect the branch, make commits, squash, ammend or whatever
git checkout merging-branches # moving to the destination branch
git merge local-branch-name # performing the merge
```

Dies kann bei der Verarbeitung von Pull-Anfragen sehr praktisch sein.

Ziehen online lesen: https://riptutorial.com/de/git/topic/1308/ziehen

Kapitel 61: Zusammenführen

Syntax

- `git merge` **another_branch** `[Optionen]`
- `git merge` **--abort**

Parameter

Parameter	Einzelheiten
`-m`	Nachricht, die in das Merge-Commit aufgenommen werden soll
`-v`	Ausführliche Ausgabe anzeigen
`--abort`	Versuchen Sie, alle Dateien in ihren Zustand zurückzusetzen
`--ff-only`	Bricht sofort ab, wenn ein Merge-Commit erforderlich ist
`--no-ff`	Erzwingt die Erstellung eines Merge-Commits, auch wenn dies nicht zwingend erforderlich ist
`--no-commit`	Gibt vor, dass die Zusammenführung die Überprüfung und das Optimieren des Ergebnisses nicht zulässt
`--stat`	Zeigt ein diffstat nach dem Abschluss der Zusammenführung an
`-n` / `--no-stat`	Das diffstat nicht anzeigen
`--squash`	Erlaubt ein einmaliges Commit für den aktuellen Zweig mit den zusammengeführten Änderungen

Examples

Verbinden Sie einen Zweig mit einem anderen

```
git merge incomingBranch
```

Dadurch wird der Zweig " incomingBranch mit dem Zweig verbunden, in dem Sie sich aktuell befinden. Wenn Sie sich zum Beispiel im master , wird " incomingBranch " mit dem master .

Das Zusammenführen kann in manchen Fällen zu Konflikten führen. In diesem Fall wird die Meldung Automatic merge failed; fix conflicts and then commit the result. Sie müssen die in Konflikt stehenden Dateien manuell bearbeiten oder den Zusammenführungsversuch rückgängig machen, indem Sie Folgendes ausführen:

```
git merge --abort
```

Automatisches Zusammenführen

Wenn die Commits in zwei Zweigen nicht miteinander in Konflikt stehen, kann Git sie automatisch zusammenführen:

```
~/Stack Overflow(branch:master) » git merge another_branch
Auto-merging file_a
Merge made by the 'recursive' strategy.
 file_a | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Abbrechen einer Zusammenführung

Nach dem Starten einer Zusammenführung möchten Sie möglicherweise die Zusammenführung beenden und alles in den Zustand vor der Zusammenführung zurücksetzen. Verwenden Sie --abort :

```
git merge --abort
```

Behalten Sie Änderungen nur auf einer Seite der Zusammenführung bei

Während einer Zusammenführung können Sie --ours oder --theirs an der git checkout --theirs , um alle Änderungen für eine Datei von der einen oder der anderen Seite einer Zusammenführung zu übernehmen.

```
$ git checkout --ours   -- file1.txt # Use our version of file1, delete all their changes
$ git checkout --theirs -- file2.txt # Use their version of file2, delete all our changes
```

Mit einem Commit zusammenführen

Das Standardverhalten ist, wenn die Zusammenführung als schneller Vorlauf aufgelöst wird und nur der Verzweigungszeiger aktualisiert wird, ohne eine Zusammenführungsfestschreibung zu erstellen. Verwenden Sie --no-ff zum Auflösen.

```
git merge <branch_name> --no-ff -m "<commit message>"
```

Finden aller Zweige ohne zusammengeführte Änderungen

Manchmal liegen Zweige herum, deren Änderungen bereits in Master eingearbeitet wurden. Dadurch werden alle Zweige gefunden, die kein master sind und im Vergleich zu master keine eindeutigen Commits haben. Dies ist sehr nützlich, um Zweige zu finden, die nicht gelöscht wurden, nachdem der PR in den Master eingefügt wurde.

```
 for branch in $(git branch -r) ; do
   [ "${branch}" != "origin/master" ] && [ $(git diff master...${branch} | wc -l) -eq 0 ] &&
echo -e `git show --pretty=format:"%ci %cr" $branch | head -n 1`\\t$branch
 done | sort -r
```

Zusammenführen online lesen: https://riptutorial.com/de/git/topic/291/zusammenfuhren

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Git	Ajedi32, Ala Eddine JEBALI, Allan Burleson, Amitay Stern, Andy Hayden, AnimiVulpis, ArtOfWarfare, bahrep, Boggin, Brian, Community, Craig Brett, Dan Hulme, ericdwang, eykanal, Fernando Hoces De La Guardia, Fred Barclay, Henrique Barcelos, intboolstring, Irfan, Jackson Blankenship, janos, Jav_Rock, jeffdill2, JonasCz, JonyD, Joseph Dasenbrock, Kageetai, Karthik , KartikKannapur, Kayvan N, Knu, Lambda Ninja, maccard, Marek Skiba, Mateusz Piotrowski, Mingle Li, mouche, Nathan Arthur, Neui, NRKirby, ob1, ownsourcing dev training, Pod, Prince J, RamenChef, Rick, Roald Nefs, ronnyfm, Sazzad Hissain Khan, Scott Weldon, Sibi Raj, TheDarkKnight, theheadofabroom, ʇolɐəz əɥʇ qoq, Tot Zam, Tyler Zika, tymspy, Undo, VonC
2	.mailmap-Datei: Verknüpfen von Mitwirkenden und E-Mail-Aliasnamen	Mario, Michael Plotke
3	Aktualisieren Sie den Objektnamen in der Referenz	Keyur Ramoliya, RamenChef
4	Aliase	AesSedai101, Ajedi32, Andy, Anthony Staunton, Asenar, bstpierre , erewok, eush77, fracz, Gaelan, jrf, jtbandes, madhead, Michael Deardeuff, mickeyandkaka, nus, penguincoder, riyadhalnur, thanksd, Tom Hale, Wojciech Kazior, zinking
5	Analysieren von Arten von Workflows	Boggin, Confiqure, Daniel Käfer, Dimitrios Mistriotis, forresthopkinsa, hardmooth, Horen, Kissaki, Majid, Sardathrion, Scott Weldon
6	Ändern Sie den Namen des Git-Repositorys	xiaoyaoworm
7	Anzeigen des Commit-Verlaufs grafisch mit Gitk	orkoden
8	Arbeitsbäume	andipla, Confiqure, Victor Schröder
9	Archiv	Dartmouth, forevergenin, Neto Buenrostro, RamenChef
10	Aufbau	APerson, Asenar, Cache Staheli, Chris Rasys, e.doroskevic, Julian, Liyan Chang, Majid, Micah Smith, Ortomala Lokni, Peter Mitrano, Priyanshu Shekhar, Scott Weldon, VonC, Wolfgang
11	Bündel	jwd630
12	Dateien und Ordner ignorieren	AER, AesSedai101, agilob, Alex, Amitay Stern, AnimiVulpis, Ates Goral, Aukhan, Avamander, Ben, bpoiss, Braiam, bwegs, Cache Staheli, Collin M, Community, Dartmouth, David Grayson, Devesh Saini, Dheeraj vats, eckes, Ed Cottrell, enrico.bacis, Everettss, Fabio, fracz, Franck Dernoncourt, Fred Barclay, Functino, geek1011, Guillaume Pascal, HerrSerker, intboolstring , Irfan, Jakub Narębski, Jeff Puckett, Jens, joaquinlpereyra,

		John Slegers, JonasCz, Jörn Hees, joshng, Kačer, Kapep, Kissaki , knut, LeftRight92, Mackattack, Marvin, Matt, MayeulC, Mitch Talmadge, Narayan Acharya, Nathan Arthur, Neui, noqɟʎdʎzɐɔ, Nuri Tasdemir, Ortomala Lokni, PaladiN, Panda, peci1, pktangyue , poke, pylang, RhysO, Rick, rokonoid, Sascha, Scott Weldon, Sebastianb, SeeuD1, sjas, Slayther, SnoringFrog, spikeheap, theJollySin, Toby, ɟolɐəz əɥɟ qoq, Tom Gijselinck, Tomasz Bąk, Vi., Victor Schröder, VonC, Wilfred Hughes, Wolfgang, ydaetskcoR, Yosvel Quintero, Yury Fedorov, Zaz, Zeeker
13	Diff-Baum	fybw id
14	Durchsuchen der Geschichte	Ahmed Metwally, Andy Hayden, Aratz, Atif Hussain, Boggin, Brett , Confiqure, davidcondrey, Fabio, Flows, fracz, Fred Barclay, guleria, intboolstring, janos, jaredr, Kamiccolo, KraigH, LeGEC , manasouza, Matt Clark, Matthew Hallatt, MByD, mpromonet, Muhammad Abdullah, Noah, Oleander, Pedro Pinheiro, RedGreenCode , Toby Allen, Vogel612, ydaetskcoR
15	Externe Zusammenführung und Difftools	AesSedai101, Micha Wiedenmann
16	Festlegen	Aaron Critchley, AER, Alan, Allan Burleson, Amitay Stern, Andrew Sklyarevsky, Andy Hayden, Anonymous Entity, APerson, bandi, Cache Staheli, Chris Forrence, Cody Guldner, cormacrelf, davidcondrey, Deep, depperm, ericdwang, Ethunxxx, Fred Barclay, George Brighton, Igor Ivancha, intboolstring, JacobLeach, James Taylor, janos, joeytwiddle, Jordan Knott, KartikKannapur, kisanme, Majid, Matt Clark, Matthew Hallatt, MayeulC, Micah Smith, Pod, Rick, Scott Weldon, SommerEngineering, Sonny Kim, Thomas Gerot, Undo, user1990366, vguzmanp, Vladimir F, Zaz
17	Git Branch Name auf Bash Ubuntu	Manishh
18	Git Clean	gnis, MayeulC, n0shadow, pktangyue, Priyanshu Shekhar, Ralf Rafael Frix
19	Git Diff	Aaron Critchley, Abhijeet Kasurde, Adi Lester, anderas, apidae, Brett, Charlie Egan, eush77, □□□□□, intboolstring, Jack Ryan, JakeD, Jakub Narębski, jeffdill2, Joseph K. Strauss, khanmizan, Luke Taylor, Majid, mnoronha, Nathaniel Ford, Ogre Psalm33, orkoden, Ortomala Lokni, penguincoder, pylang, SurDin, Will, ydaetskcoR, Zaz
20	Git GUI Clients	Alu, Daniel Käfer, Greg Bray, Nemanja Trifunovic, Pedro Pinheiro
21	Git Large File Storage (LFS)	Alex Stuckey, Matthew Hallatt, shoelzer
22	Git Patch	Dartmouth, Liju Thomas
23	Git Remote	AER, ambes, Dániel Kis, Dartmouth, Elizabeth, Jav_Rock, Kalpit, RamenChef, sonali, sunkuet02
24	Git rerere	Isak Combrinck
25	Git Revisions-Syntax	Dartmouth, Jakub Narębski
26	git send-email	Aaron Skomra, Dong Thang, fybw id, Jav_Rock, kofemann

27	Git Tagging	Atul Khanduri, demonplus, TheDarkKnight
28	Git-Client-Side-Hooks	Kelum Senanayake, kiamlaluno
29	Git-Statistiken	Dartmouth, Farhad Faghihi, Hugo Buff, KartikKannapur, lxer, penguincoder, RamenChef, SashaZd, Tyler Hyndman, vkluge
30	git-svn	Bryan, Randy, Ricardo Amores, RobPethi
31	git-tfs	Boggin, Kissaki
32	Haken	AesSedai101, AnoE, Christiaan Maks, Confiqure, Eidolon, Flows, fracz, kaartic, lostphilosopher, mwarsco
33	Halbieren / Finden fehlerhafter Commits	4444, Hannoun Yassir, jornh, Kissaki, MrTux, Scott Weldon, Simone Carletti, zebediah49
34	Historie mit Filterzweig umschreiben	gavinbeatty, gavv, Glenn Smith
35	Ihr lokales und Remote-Repository aufräumen	Thomas Crowley
36	Inszenierung	AesSedai101, Andy Hayden, Asaph, Confiqure, intboolstring, Jakub Narębski, jkdev, Muhammad Abdullah, Nathan Arthur, ownsourcing dev training, Richard Dally, Wolfgang
37	Interne	nighthawk454
38	Leere Verzeichnisse in Git	Ates Goral
39	Mischkonflikte lösen	Braiam, Dartmouth, David Ben Knoble, Fabio, nus, Vivin George, Yury Fedorov
40	Mit Remotes arbeiten	Boggin, Caleb Brinkman, forevergenin, heitortsergent, intboolstring, jeffdill2, Julie David, Kalpit, Matt Clark, MByD , mnoronha, mpromonet, mystarrocks, Pascalz, Raghav, Ralf Rafael Frix, Salah Eddine Lahniche, Sam, Scott Weldon, Stony, Thamilan, Vivin George, VonC, Zaz
41	Neueinstellung	AER, Alexander Bird, anderas, Ashwin Ramaswami, Braiam, BusyAnt , Confiqure, Daniel Käfer, Derek Liu, Dunno, e.doroskevic, Enrico Campidoglio, eskwayrd, □□□□□, Hugo Ferreira, intboolstring, Jeffrey Lin, Joel Cornett, Joseph K. Strauss, jtbandes, Julian, Kissaki, LeGEC, Libin Varghese, Luca Putzu, lucash, madhukar93, Majid, Matt, Matthew Hallatt, Menasheh, Michael Mrozek, Nemanja Boric, Ortomala Lokni, Peter Mitrano, pylang, Richard, takteek, Travis, Victor Schröder, VonC, Wasabi Fan, yarons, Zaz
42	Reflog - Wiederherstellen von Commits, die nicht im Git-Log angezeigt werden	Braiam, Peter Amidon, Scott Weldon
43	Repositorys klonen	AER, Andrea Romagnoli, Andy Hayden, Blundering Philosopher, Dartmouth, Ezra Free, ganesshkumar, □□□□□, kartik, KartikKannapur, mnoronha, Peter Mitrano, pkowalczyk, Rick, Undo

		, Wojciech Kazior
44	Rev-Liste	mkasberg
45	Rosinenpickerei	Atul Khanduri, Braiam, bud-e, dubek, Florian Hämmerle, intboolstring, Julian, kisanme, Lochlan, mpromonet, RedGreenCode
46	Schieben	AER, Cody Guldner, cringe, frlan, Guillaume, intboolstring, Mário Meyrelles, Marvin, Matt S, MayeulC, pcm, pogosama, Thomas Gerot, Tomás Cañibano
47	Schuldzuweisungen	fracz, Matthew Hallatt, nighthawk454, Priyanshu Shekhar, WPrecht
48	Show	Zaz
49	Squashing	adarsh, ams, AndiDog, bandi, Braiam, Caleb Brinkman, eush77, georgebrock, jpkrohling, Julian, Mateusz Piotrowski, Ortomala Lokni, RamenChef, Tall Sam, WMios
50	Submodule	321hendrik, Chin Huang, ComicSansMS, foraidt, intboolstring, J F, kowsky, mpromonet, PaladiN, tinlyx, Undo, VonC
51	TortoiseGit	Julian, Matas Vaitkevicius
52	Umbenennung	bud-e, Karan Desai, P.J.Meisch, PhotometricStereo
53	Umstellung auf Git	AesSedai101, Boggin, Confiqure, Guillaume Pascal, Indregaard, Rick, TheDarkKnight
54	Unterbäume	4444, Jeff Puckett
55	Verhängnis	Adi Lester, AesSedai101, Alexander Bird, Andy Hayden, Boggin, brentonstrine, Brian, Colin D Bennett, ericdwang, Karan Desai, Matthew Hallatt, Nathan Arthur, Nathaniel Ford, Nithin K Anil, Pace, Rick, textshell, Undo, Zaz
56	Verstauen	aavrug, AesSedai101, Asaph, Brian Hinchey, bud-e, Cache Staheli , Deep, e.doroskevic, fracz, GingerPlusPlus, Guillaume, inkista , Jakub Narębski, Jarede, jeffdill2, joeytwiddle, Julie David, Kara, Koraktor, Majid, manasouza, Ortomala Lokni, Patrick, Peter Mitrano, Ralf Rafael Frix, Sebastianb, Tomás Cañibano, Wojciech Kazior
57	Verwenden einer .gitattributes-Datei	Chin Huang, dahlbyk, Toby
58	Verzweigung	Amitay Stern, Andrew Kay, AnimiVulpis, Bad, BobTuckerman, Community, dan, Daniel Käfer, Daniel Stradowski, Deepak Bansal, djb, Don Kirkby, Duncan X Simpson, Eric Bouchut, forevergenin, fracz, Franck Dernoncourt, Fred Barclay, Frodon, gavv, Irfan, james large, janos, Jason, Joel Cornett, Jon Schneider, Jonathan, Joseph Dasenbrock, jrf, kartik, KartikKannapur, khanmizan, kirrmann, kisanme, Majid, Martin, MayeulC, Michael Richardson, Mihai, Mitch Talmadge, mkasberg, nepda, Noah, Noushad PP, Nowhere man, olegtaranenko, Ortomala Lokni, Ozair Kafray, PaladiN, □ANAYI□TIS, Priyanshu Shekhar, Ralf Rafael Frix, Richard Hamilton, Robin, RudolphEst, Siavas, Simone Carletti, the12, Uwe, Vlad, wintersolider, Wojciech Kazior, Wolfgang, Yerko Palma, Yury Fedorov, zygimantus

59	Wiederherstellen	Creative John, Hardik Kanjariya ツ, Julie David, kisanme, ANAYITIS, Scott Weldon, strangeqargo, Zaz
60	Ziehen	Kissaki, MayeulC, mpromonet, rene, Ryan, Scott Weldon, Shog9, Stony, Thamilan, Thomas Gerot, Zaz
61	Zusammenführen	brentonstrine, Liam Ferris, Noah, penguincoder, Undo, Vogel612, Wolfgang