## Bắt đầu với Git

### Tạo kho lưu trữ đầu tiên của bạn, sau đó thêm và cam kết các tập tin

Tại dòng lệnh, trước tiên hãy xác minh rằng bạn đã cài đặt Git:

* Trên tất cả các hệ điều hành:

bash

Sao chép mã

git --version

* Trên các hệ điều hành tương tự UNIX:

bash

Sao chép mã

which git

Nếu không có gì được trả về, hoặc lệnh không được nhận diện, bạn có thể cần cài đặt Git trên hệ thống của mình bằng cách tải xuống và chạy trình cài đặt. Xem trang chủ Git để có hướng dẫn cài đặt rõ ràng và dễ hiểu.

Sau khi cài đặt Git, hãy cấu hình tên người dùng và địa chỉ email của bạn. Hãy thực hiện việc này trước khi thực hiện bất kỳ cam kết nào.

Khi Git đã được cài đặt, điều hướng đến thư mục mà bạn muốn đặt dưới sự kiểm soát phiên bản và tạo một kho lưu trữ Git trống:

bash

Sao chép mã

git init

Điều này sẽ tạo một thư mục ẩn, .git, chứa các cấu trúc cần thiết để Git hoạt động.

Tiếp theo, kiểm tra các tập tin mà Git sẽ thêm vào kho lưu trữ mới của bạn; bước này cần được chú ý đặc biệt:

bash

Sao chép mã

git status

Xem xét danh sách các tập tin kết quả; bạn có thể cho Git biết tập tin nào sẽ đưa vào kiểm soát phiên bản (tránh thêm các tập tin chứa thông tin bí mật như mật khẩu, hoặc các tập tin chỉ làm rối kho lưu trữ):

bash

Sao chép mã

git add <tên\_tập\_tin/thư\_mục #1> <tên\_tập\_tin/thư\_mục #2> < ... >

Nếu tất cả các tập tin trong danh sách nên được chia sẻ với tất cả những ai có quyền truy cập vào kho lưu trữ, một lệnh duy nhất sẽ thêm tất cả các tập tin trong thư mục hiện tại và các thư mục con của nó:

bash

Sao chép mã

git add .

Lệnh này sẽ "chuẩn bị" tất cả các tập tin để được đưa vào kiểm soát phiên bản, chuẩn bị cho chúng được cam kết trong cam kết đầu tiên của bạn.

Đối với các tập tin mà bạn không muốn đưa vào kiểm soát phiên bản, hãy tạo và điền nội dung cho một tập tin có tên là .gitignore trước khi thực hiện lệnh add.

Cam kết tất cả các tập tin đã được thêm, cùng với một thông điệp cam kết:

bash

Sao chép mã

git commit -m "Initial commit"

Điều này tạo ra một cam kết mới với thông điệp đã cho. Một cam kết giống như một lần lưu hoặc một ảnh chụp của toàn bộ dự án của bạn. Bạn có thể đẩy, hoặc tải lên, nó vào kho lưu trữ từ xa, và sau đó bạn có thể quay lại nếu cần.

Nếu bạn bỏ qua tham số -m, trình soạn thảo mặc định của bạn sẽ mở ra và bạn có thể chỉnh sửa và lưu thông điệp cam kết ở đó.

#### Thêm một kho lưu trữ từ xa

Để thêm một kho lưu trữ từ xa mới, hãy sử dụng lệnh git remote add trong terminal, trong thư mục nơi kho lưu trữ của bạn được lưu trữ. Lệnh git remote add nhận hai đối số:

1. Tên từ xa, ví dụ: origin
2. URL từ xa, ví dụ: https://<địa\_chỉ\_git\_service>/user/repo.git

bash

Sao chép mã

git remote add origin https://<địa\_chỉ\_git\_service>/owner/repository.git

LƯU Ý: Trước khi thêm kho lưu trữ từ xa, bạn phải tạo kho lưu trữ cần thiết trong dịch vụ git của bạn. Bạn sẽ có thể đẩy/kéo cam kết sau khi thêm kho lưu trữ từ xa.

#### Mục 1.2: Nhân bản (clone) một kho lưu trữ

Lệnh git clone được sử dụng để sao chép một kho lưu trữ Git hiện có từ một máy chủ về máy tính cục bộ. Ví dụ, để nhân bản một dự án trên GitHub:

bash

Sao chép mã

cd <đường\_dẫn\_nơi\_bạn\_muốn\_tạo\_thư\_mục>

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

Để nhân bản một dự án trên BitBucket:

bash

Sao chép mã

cd <đường\_dẫn\_nơi\_bạn\_muốn\_tạo\_thư\_mục>

git clone https://yourusername@bitbucket.org/username/projectname.git

Điều này tạo ra một thư mục có tên là projectname trên máy tính cục bộ, chứa tất cả các tập tin trong kho lưu trữ Git từ xa. Điều này bao gồm các tập tin nguồn cho dự án, cũng như một thư mục con .git chứa toàn bộ lịch sử và cấu hình cho dự án.

Để chỉ định một tên khác cho thư mục, ví dụ: MyFolder:

bash

Sao chép mã

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

Hoặc để nhân bản vào thư mục hiện tại:

bash

Sao chép mã

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

Lưu ý:

1. Khi nhân bản vào một thư mục được chỉ định, thư mục phải trống hoặc không tồn tại.
2. Bạn cũng có thể sử dụng phiên bản ssh của lệnh:

bash

Sao chép mã

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

Phiên bản https và phiên bản ssh là tương đương. Tuy nhiên, một số dịch vụ lưu trữ như GitHub khuyến nghị bạn sử dụng https thay vì ssh.

#### Mục 1.3: Chia sẻ mã nguồn

Để chia sẻ mã nguồn của bạn, bạn tạo một kho lưu trữ trên một máy chủ từ xa mà bạn sẽ sao chép kho lưu trữ cục bộ của bạn vào đó.

Để giảm thiểu việc sử dụng không gian trên máy chủ từ xa, bạn tạo một kho lưu trữ trần (bare repository): một kho lưu trữ chỉ có các đối tượng .git và không tạo một bản sao làm việc trong hệ thống tập tin. Thêm vào đó, bạn đặt máy chủ từ xa này làm máy chủ upstream để dễ dàng chia sẻ các cập nhật với các lập trình viên khác.

Trên máy chủ từ xa:

bash

Sao chép mã

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

Trên máy tính cục bộ:

bash

Sao chép mã

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

(Lưu ý rằng ssh: chỉ là một cách có thể để truy cập kho lưu trữ từ xa.)

Bây giờ sao chép kho lưu trữ cục bộ của bạn lên máy chủ từ xa:

bash

Sao chép mã

git push --set-upstream origin master

Việc thêm --set-upstream (hoặc -u) tạo ra một tham chiếu upstream (theo dõi) được sử dụng bởi các lệnh Git không có tham số, ví dụ: git pull.

#### Mục 1.4: Cài đặt tên người dùng và email của bạn

Bạn cần thiết lập danh tính của mình trước khi tạo bất kỳ cam kết nào. Điều này sẽ cho phép các cam kết có tên và email của tác giả được liên kết đúng. Việc này không liên quan đến xác thực khi đẩy đến một kho lưu trữ từ xa (ví dụ: khi đẩy đến kho lưu trữ từ xa bằng tài khoản GitHub, BitBucket hoặc GitLab của bạn).

Để khai báo danh tính đó cho tất cả các kho lưu trữ, sử dụng git config --global. Điều này sẽ lưu thiết lập trong tệp .gitconfig của người dùng: ví dụ $HOME/.gitconfig hoặc đối với Windows, %USERPROFILE%\.gitconfig.

bash

Sao chép mã

git config --global user.name "Tên của bạn"

git config --global user.email mail@example.com

Để khai báo danh tính cho một kho lưu trữ đơn lẻ, sử dụng git config bên trong một kho lưu trữ. Điều này sẽ lưu thiết lập bên trong kho lưu trữ cá nhân, trong tệp $GIT\_DIR/config, ví dụ /path/to/your/repo/.git/config.

bash

Sao chép mã

cd /path/to/my/repo

git config user.name "Tên đăng nhập của bạn tại công việc"

git config user.email mail\_at\_work@example.com

Các thiết lập lưu trong tệp config của kho lưu trữ sẽ ưu tiên hơn so với cấu hình toàn cục khi bạn sử dụng kho lưu trữ đó.

Mẹo: nếu bạn có các danh tính khác nhau (một cho dự án mã nguồn mở, một cho công việc, một cho kho lưu trữ cá nhân, ...), và bạn không muốn quên thiết lập danh tính đúng cho mỗi kho lưu trữ khác nhau mà bạn đang làm việc:

* Xóa một danh tính toàn cục

bash

Sao chép mã

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

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

Phiên bản ≥ 2.8

Để buộc git chỉ tìm danh tính của bạn trong các thiết lập của kho lưu trữ, không trong cấu hình toàn cục:

bash

Sao chép mã

git config --global user.useConfigOnly true

Bằng cách này, nếu bạn quên thiết lập user.name và user.email cho một kho lưu trữ cụ thể và cố gắng thực hiện cam kết, bạn sẽ thấy thông báo:

perl

Sao chép mã

no name was given and auto-detection is disabled

no email was given and auto-detection is disabled

#### Mục 1.5: Thiết lập upstream remote

Nếu bạn đã nhân bản một nhánh (ví dụ: một dự án mã nguồn mở trên Github), bạn có thể không có quyền đẩy vào kho lưu trữ upstream, vì vậy bạn cần cả nhánh của mình và có thể lấy dữ liệu từ kho lưu trữ upstream.

Trước tiên kiểm tra tên từ xa:

bash

Sao chép mã

git remote -v

perl

Sao chép mã

origin https://github.com/myusername/repo.git (fetch)

origin https://github.com/myusername/repo.git (push)

upstream # dòng này có thể hoặc không có ở đây

Nếu upstream đã có sẵn (nó có trong một số phiên bản Git), bạn cần thiết lập URL (hiện tại nó trống):

bash

Sao chép mã

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

Nếu upstream không có ở đó, hoặc nếu bạn cũng muốn thêm nhánh của một người bạn/đồng nghiệp (hiện tại chúng không tồn tại):

bash

Sao chép mã

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

git remote add dave https://github.com/dave/repo.git

#### Mục 1.6: Tìm hiểu về một lệnh

Để có thêm thông tin về bất kỳ lệnh git nào – tức là chi tiết về những gì lệnh làm, các tùy chọn có sẵn và tài liệu khác – sử dụng tùy chọn --help hoặc lệnh help.

Ví dụ, để có tất cả thông tin có sẵn về lệnh git diff, sử dụng:

bash

Sao chép mã

git diff --help

git help diff

Tương tự, để có tất cả thông tin có sẵn về lệnh status, sử dụng:

bash

Sao chép mã

git status --help

git help status

Nếu bạn chỉ muốn một trợ giúp nhanh hiển thị ý nghĩa của các cờ dòng lệnh được sử dụng nhiều nhất, sử dụng -h:

bash

Sao chép mã

git checkout -h

#### Mục 1.7: Thiết lập SSH cho Git

Nếu bạn đang sử dụng Windows, mở Git Bash. Nếu bạn đang sử dụng Mac hoặc Linux, mở Terminal của bạn.

Trước khi bạn tạo khóa SSH, bạn có thể kiểm tra xem bạn có bất kỳ khóa SSH hiện có nào không. Liệt kê nội dung của thư mục ~/.ssh của bạn:

bash

Sao chép mã

ls -al ~/.ssh

Danh sách tất cả các tập tin trong thư mục ~/.ssh của bạn.

Kiểm tra danh sách thư mục để xem bạn đã có một khóa SSH công khai chưa. Theo mặc định, tên tập tin của các khóa công khai là một trong những tên sau:

* id\_dsa.pub
* id\_ecdsa.pub
* id\_ed25519.pub
* id\_rsa.pub

Nếu bạn thấy một cặp khóa công khai và khóa riêng hiện có mà bạn muốn sử dụng trên Bitbucket, GitHub (hoặc tương tự), bạn có thể sao chép nội dung của tập tin id\_\*.pub.

Nếu không, bạn có thể tạo một cặp khóa công khai và khóa riêng mới với lệnh sau:

bash

Sao chép mã

ssh-keygen

Nhấn Enter hoặc Return để chấp nhận vị trí mặc định. Nhập và nhập lại cụm mật khẩu khi được nhắc, hoặc để trống.

Đảm bảo rằng khóa SSH của bạn được thêm vào ssh-agent. Khởi động ssh-agent trong nền nếu nó chưa chạy:

bash

Sao chép mã

eval "$(ssh-agent -s)"

Thêm khóa SSH của bạn vào ssh-agent. Lưu ý rằng bạn cần thay thế id\_rsa trong lệnh bằng tên của tập tin khóa riêng của bạn:

bash

Sao chép mã

ssh-add ~/.ssh/id\_rsa

Nếu bạn muốn thay đổi upstream của một kho lưu trữ hiện có từ HTTPS sang SSH, bạn có thể chạy lệnh sau:

bash

Sao chép mã

git remote set-url origin ssh://git@bitbucket.server.com:7999/projects/your\_project.git

Để nhân bản một kho lưu trữ mới qua SSH, bạn có thể chạy lệnh sau:

bash

Sao chép mã

git clone ssh://git@bitbucket.server.com:7999/projects/your\_project.git

#### Mục 1.8: Cài đặt Git

Bây giờ chúng ta hãy bắt đầu sử dụng Git. Trước tiên, bạn phải cài đặt nó. Bạn có thể cài đặt Git theo nhiều cách; hai cách chính là cài đặt từ mã nguồn hoặc cài đặt gói có sẵn cho nền tảng của bạn.

**Cài đặt từ mã nguồn**

Nếu bạn có thể, việc cài đặt Git từ mã nguồn thường hữu ích, vì bạn sẽ có được phiên bản mới nhất. Mỗi phiên bản của Git thường bao gồm các cải tiến giao diện người dùng hữu ích, vì vậy việc có phiên bản mới nhất thường là lựa chọn tốt nhất nếu bạn cảm thấy thoải mái khi biên dịch phần mềm từ mã nguồn. Cũng có trường hợp nhiều bản phân phối Linux chứa các gói rất cũ; do đó, trừ khi bạn đang sử dụng bản phân phối rất cập nhật hoặc sử dụng backports, việc cài đặt từ mã nguồn có thể là lựa chọn tốt nhất.

Để cài đặt Git, bạn cần có các thư viện sau mà Git phụ thuộc vào: curl, zlib, openssl, expat, và libiconv. Ví dụ, nếu bạn đang sử dụng một hệ thống có yum (như Fedora) hoặc apt-get (như một hệ thống dựa trên Debian), bạn có thể sử dụng một trong những lệnh sau để cài đặt tất cả các phụ thuộc:

bash

Sao chép mã

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

Khi bạn đã có tất cả các phụ thuộc cần thiết, bạn có thể tiến hành lấy bản snapshot mới nhất từ trang web Git: <http://git-scm.com/download>

Sau đó, biên dịch và cài đặt:

bash

Sao chép mã

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

Sau khi hoàn tất, bạn cũng có thể lấy Git qua chính Git để cập nhật:

bash

Sao chép mã

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

**Cài đặt trên Linux**

Nếu bạn muốn cài đặt Git trên Linux qua một trình cài đặt nhị phân, bạn thường có thể làm như vậy thông qua công cụ quản lý gói cơ bản đi kèm với bản phân phối của bạn. Nếu bạn đang sử dụng Fedora, bạn có thể sử dụng yum:

bash

Sao chép mã

yum install git

Hoặc nếu bạn đang sử dụng một bản phân phối dựa trên Debian như Ubuntu, hãy thử apt-get:

bash

Sao chép mã

apt-get install git

**Cài đặt trên Mac**

Có ba cách dễ dàng để cài đặt Git trên Mac. Cách dễ nhất là sử dụng trình cài đặt Git đồ họa, bạn có thể tải xuống từ trang SourceForge: <http://sourceforge.net/projects/git-osx-installer/>

Cách khác là cài đặt Git qua MacPorts (<http://www.macports.org>). Nếu bạn đã cài đặt MacPorts, cài đặt Git qua:

bash

Sao chép mã

sudo port install git +svn +doc +bash\_completion +gitweb

Bạn không cần thêm tất cả các tiện ích bổ sung, nhưng có lẽ bạn sẽ muốn bao gồm +svn trong trường hợp bạn cần sử dụng Git với các kho lưu trữ Subversion.

Homebrew (<http://brew.sh/>) là một lựa chọn khác để cài đặt Git. Nếu bạn đã cài đặt Homebrew, cài đặt Git qua:

bash

Sao chép mã

brew install git

**Cài đặt trên Windows**

Cài đặt Git trên Windows rất dễ dàng. Dự án msysGit có một trong những quy trình cài đặt dễ dàng nhất. Chỉ cần tải xuống tệp trình cài đặt exe từ trang GitHub, và chạy nó: <http://msysgit.github.io>

Sau khi cài đặt, bạn có cả phiên bản dòng lệnh (bao gồm một trình khách SSH sẽ hữu ích sau này) và GUI tiêu chuẩn.

Lưu ý về việc sử dụng trên Windows: bạn nên sử dụng Git với shell msysGit được cung cấp (kiểu Unix), nó cho phép sử dụng các dòng lệnh phức tạp được đưa ra trong sách này. Nếu bạn cần, vì lý do nào đó, sử dụng shell Windows gốc/console dòng lệnh, bạn phải sử dụng dấu ngoặc kép thay vì dấu ngoặc đơn (cho các tham số có dấu cách trong đó) và bạn phải trích dẫn các tham số kết thúc bằng dấu mũ (^) nếu chúng là cuối cùng trên dòng, vì nó là ký hiệu tiếp tục trên Windows.

## Chương 2: Duyệt lịch sử

| **Tham số** | **Giải thích** |
| --- | --- |
| -q, --quiet | Yên lặng, ngăn xuất kết quả khác biệt |
| --source | Hiển thị nguồn của cam kết |
| --use-mailmap | Sử dụng tệp mail map (thay đổi thông tin người dùng khi cam kết) |
| --decorate=[...] | Tùy chọn trang trí |
| --L <n,m  > | Hiển thị log cho phạm vi cụ thể của các dòng trong một tệp, bắt đầu từ 1. Bắt đầu từ dòng n, đi đến dòng m. Cũng hiển thị khác biệt. |
| --show-signature | Hiển thị chữ ký của các cam kết đã ký |
| -i, --regexp-ignore-case | Khớp biểu thức chính quy giới hạn mẫu mà không quan tâm đến chữ hoa chữ thường |

#### Mục 2.1: Git Log "Thông thường"

git log sẽ hiển thị tất cả các cam kết của bạn với tác giả và mã băm. Điều này sẽ được hiển thị trên nhiều dòng cho mỗi cam kết. (Nếu bạn muốn hiển thị một dòng duy nhất cho mỗi cam kết, hãy xem phần oneline). Sử dụng phím q để thoát khỏi log.

Mặc định, không có tham số nào, git log liệt kê các cam kết được thực hiện trong kho lưu trữ đó theo thứ tự ngược thời gian - nghĩa là các cam kết mới nhất sẽ xuất hiện trước. Như bạn có thể thấy, lệnh này liệt kê mỗi cam kết với mã băm SHA-1, tên và email của tác giả, ngày viết, và thông điệp cam kết.

Ví dụ (từ kho lưu trữ Free Code Camp):

plaintext

Sao chép mã

commit 87ef97f59e2a2f4dc425982f76f14a57d0900bcf

Merge: e50ff0d eb8b729

Author: Brian

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

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

Date: Thu Mar 24 14:26:04 2016 +0530

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

Remove unnecessary comma from CONTRIBUTING.md

Nếu bạn muốn giới hạn lệnh của mình chỉ hiển thị log của các cam kết gần nhất, bạn có thể đơn giản truyền vào tham số. Ví dụ, nếu bạn muốn liệt kê log của 2 cam kết gần nhất:

bash

Sao chép mã

git log -2

#### Mục 2.2: Log Đẹp Hơn

Để xem log trong cấu trúc dạng đồ thị đẹp hơn, sử dụng:

bash

Sao chép mã

git log --decorate --oneline --graph

Ví dụ đầu ra:

plaintext

Sao chép mã

\* 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

...

Vì đây là một lệnh khá dài, bạn có thể tạo một bí danh (alias):

bash

Sao chép mã

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

Để sử dụng phiên bản alias:

* Lịch sử của nhánh hiện tại:

bash

Sao chép mã

git lol

* Lịch sử kết hợp của nhánh đang hoạt động (HEAD), nhánh develop và nhánh origin/master:

bash

Sao chép mã

git lol HEAD develop origin/master

* Lịch sử kết hợp của tất cả mọi thứ trong kho lưu trữ của bạn:

bash

Sao chép mã

git lol --all

#### Mục 2.3: Tô Màu Log

bash

Sao chép mã

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

Tùy chọn format cho phép bạn chỉ định định dạng đầu ra log của riêng mình:

Thông tin chi tiết:

* %C(color\_name): Tùy chọn màu sắc cho đầu ra tiếp theo
* %h hoặc %H: Rút gọn mã cam kết (sử dụng %H cho mã đầy đủ)
* %Creset: Đặt lại màu về màu mặc định của terminal
* %d: Tên tham chiếu
* %s: Chủ đề [thông điệp cam kết]
* %cr: Ngày cam kết, so với ngày hiện tại
* %an: Tên tác giả

#### Mục 2.4: Log Một Dòng

bash

Sao chép mã

git log --oneline

sẽ hiển thị tất cả các cam kết của bạn với chỉ phần đầu của mã băm và thông điệp cam kết. Mỗi cam kết sẽ nằm trên một dòng duy nhất, như cờ oneline gợi ý.

Tùy chọn oneline in mỗi cam kết trên một dòng, hữu ích nếu bạn đang xem xét nhiều cam kết.

Ví dụ (từ kho lưu trữ Free Code Camp, với cùng một đoạn mã từ ví dụ khác):

plaintext

Sao chép mã

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/

Nếu bạn muốn giới hạn lệnh của mình chỉ hiển thị log của các cam kết gần nhất, bạn có thể đơn giản truyền vào tham số. Ví dụ, nếu bạn muốn liệt kê log của 2 cam kết gần nhất:

bash

Sao chép mã

git log -2 --oneline

#### Mục 2.5: Tìm Kiếm Log

bash

Sao chép mã

git log -S"#define SAMPLES"

Tìm kiếm sự thêm vào hoặc loại bỏ của chuỗi cụ thể hoặc chuỗi khớp với REGEXP đã cung cấp. Trong trường hợp này, chúng ta đang tìm kiếm sự thêm vào/loại bỏ của chuỗi #define SAMPLES.

Ví dụ:

plaintext

Sao chép mã

+#define SAMPLES 100000

hoặc

plaintext

Sao chép mã

-#define SAMPLES 100000

bash

Sao chép mã

git log -G"#define SAMPLES"

Tìm kiếm sự thay đổi trong các dòng chứa chuỗi cụ thể hoặc chuỗi khớp với REGEXP đã cung cấp.

Ví dụ:

plaintext

Sao chép mã

-#define SAMPLES 100000

+#define SAMPLES 100000000

#### Mục 2.6: Liệt Kê Tất Cả Đóng Góp Theo Tên Tác Giả

git shortlog tóm tắt git log và nhóm theo tác giả. Nếu không có tham số nào được đưa ra, danh sách tất cả các cam kết được thực hiện bởi từng người cam kết sẽ được hiển thị theo thứ tự thời gian.

bash

Sao chép mã

git shortlog

Ví dụ:

plaintext

Sao chép mã

Committer 1 (<number\_of\_commits>):

Commit Message 1

Commit Message 2

...

Committer 2 (<number\_of\_commits>):

Commit Message 1

Commit Message 2

...

Để chỉ đơn giản xem số lượng cam kết và bỏ qua mô tả cam kết, sử dụng tùy chọn summary:

bash

Sao chép mã

git shortlog -s

Ví dụ:

plaintext

Sao chép mã

<number\_of\_commits> Committer 1

<number\_of\_commits> Committer 2

Để sắp xếp đầu ra theo số lượng cam kết thay vì theo thứ tự chữ cái của tên người cam kết, sử dụng tùy chọn numbered:

bash

Sao chép mã

git shortlog -n

Để thêm email của người cam kết, sử dụng tùy chọn email:

bash

Sao chép mã

git shortlog -e

Tùy chọn định dạng tùy chỉnh cũng có thể được cung cấp nếu bạn muốn hiển thị thông tin khác ngoài chủ đề cam kết:

bash

Sao chép mã

git shortlog --format

Xem phần Tô Màu Log ở trên để biết thêm thông tin về điều này.

#### Mục 2.7: Tìm Kiếm Chuỗi Cam Kết Trong Git Log

Tìm kiếm git log sử dụng một chuỗi trong log:

bash

Sao chép mã

git log [options] --grep "search\_string"

Ví dụ:

bash

Sao chép mã

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

Sẽ tìm kiếm chuỗi removed file trong tất cả các log trên tất cả các nhánh.

Bắt đầu từ git 2.4+, tìm kiếm có thể bị đảo ngược bằng cách sử dụng tùy chọn --invert-grep. Ví dụ:

bash

Sao chép mã

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

Sẽ hiển thị tất cả các cam kết không chứa add file.

#### Mục 2.8: Log Cho Một Phạm Vi Dòng Trong Một Tệp

bash

Sao chép mã

git log -L 1,20:index.html

Ví dụ:

plaintext

Sao chép mã

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">

#### Mục 2.9: Lọc Log

bash

Sao chép mã

git log --after '3 days ago'

Các ngày cụ thể cũng hoạt động:

bash

Sao chép mã

git log --after 2016-05-01

Giống như các lệnh và cờ khác chấp nhận tham số ngày, định dạng ngày được cho phép là định dạng được hỗ trợ bởi GNU date (rất linh hoạt).

Một bí danh cho --after là --since.

Các cờ tương ứng cho ngược lại cũng tồn tại: --before và --until.

Bạn cũng có thể lọc log theo tác giả. Ví dụ:

bash

Sao chép mã

git log --author=author

#### Mục 2.10: Log Với Thay Đổi Trong Dòng

Để xem log với các thay đổi trong dòng, sử dụng tùy chọn -p hoặc --patch.

bash

Sao chép mã

git log --patch

Ví dụ (từ kho lưu trữ Trello Scientist):

plaintext

Sao chép mã

commit 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).

#### Mục 2.11: Log Hiển Thị Các Tệp Đã Cam Kết

bash

Sao chép mã

git log --stat

Ví dụ:

plaintext

Sao chép mã

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(+)

#### Mục 2.12: Hiển Thị Nội Dung Một Cam Kết

Sử dụng git show chúng ta có thể xem một cam kết duy nhất

bash

Sao chép mã

git show 48c83b3

git show 48c83b3690dfc7b0e622fd220f8f37c26a77c934

Ví dụ:

plaintext

Sao chép mã

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" ));

#### Mục 2.13: Git Log Giữa Hai Nhánh

git log master..foo sẽ hiển thị các cam kết có trên nhánh foo và không có trên nhánh master. Hữu ích để xem những cam kết nào bạn đã thêm kể từ khi tạo nhánh!

#### Mục 2.14: Một Dòng Hiển Thị Tên Người Cam Kết và Thời Gian Kể Từ Cam Kết

plaintext

Sao chép mã

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

Ví dụ:

plaintext

Sao chép mã

\* 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

## Làm việc với các kho lưu trữ từ xa

#### Mục 3.1: Xóa một nhánh từ xa

Để xóa một nhánh từ xa trong Git:

bash

Sao chép mã

git push [tên\_remote] --delete [tên\_nhánh]

hoặc

bash

Sao chép mã

git push [tên\_remote] :[tên\_nhánh]

#### Mục 3.2: Thay đổi URL của Remote Git

Kiểm tra remote hiện có:

bash

Sao chép mã

git remote -v

# origin https://github.com/username/repo.git (fetch)

# origin https://github.com/username/repo.git (push)

Thay đổi URL của kho lưu trữ:

bash

Sao chép mã

git remote set-url origin https://github.com/username/repo2.git

# Thay đổi URL của remote 'origin'

Xác minh URL remote mới:

bash

Sao chép mã

git remote -v

# origin https://github.com/username/repo2.git (fetch)

# origin https://github.com/username/repo2.git (push)

#### Mục 3.3: Liệt kê các Remote hiện có

Liệt kê tất cả các remote hiện có liên kết với kho lưu trữ này:

bash

Sao chép mã

git remote

Liệt kê chi tiết tất cả các remote hiện có liên kết với kho lưu trữ này bao gồm cả URL fetch và push:

bash

Sao chép mã

git remote --verbose

hoặc đơn giản là:

bash

Sao chép mã

git remote -v

#### Mục 3.4: Loại bỏ các bản sao cục bộ của các nhánh từ xa đã bị xóa

Nếu một nhánh từ xa đã bị xóa, kho lưu trữ cục bộ của bạn cần được thông báo để loại bỏ tham chiếu tới nó.

Để loại bỏ các nhánh đã bị xóa từ một remote cụ thể:

bash

Sao chép mã

git fetch [tên\_remote] --prune

Để loại bỏ các nhánh đã bị xóa từ tất cả các remotes:

bash

Sao chép mã

git fetch --all --prune

#### Mục 3.5: Cập nhật từ kho lưu trữ Upstream

Giả sử bạn đã thiết lập upstream (như trong phần "thiết lập kho lưu trữ upstream"):

bash

Sao chép mã

git fetch [tên\_remote]

git merge [tên\_remote]/[tên\_nhánh]

Lệnh pull kết hợp một lệnh fetch và một lệnh merge:

bash

Sao chép mã

git pull

Lệnh pull với cờ --rebase kết hợp một lệnh fetch và một lệnh rebase thay vì merge:

bash

Sao chép mã

git pull --rebase [tên\_remote] [tên\_nhánh]

#### Mục 3.6: ls-remote

git ls-remote là một lệnh đặc biệt cho phép bạn truy vấn một kho lưu trữ từ xa mà không cần phải clone/fetch nó trước. Nó sẽ liệt kê các refs/heads và refs/tags của kho lưu trữ từ xa đó.

Bạn sẽ thấy đôi khi có refs/tags/v0.1.6 và refs/tags/v0.1.6^{}: ký hiệu ^{} để liệt kê tag đã giải thích (tức là cam kết mà tag đang trỏ tới).

Kể từ git 2.8 (tháng 3 năm 2016), bạn có thể tránh mục nhập đôi cho một tag và liệt kê trực tiếp các tag đã giải thích với:

bash

Sao chép mã

git ls-remote --ref

Nó cũng có thể giúp giải quyết URL thực tế được sử dụng bởi một kho lưu trữ từ xa khi bạn có thiết lập cấu hình url.<base>.insteadOf.

Nếu git remote --get-url <tên\_remote> trả về https://server.com/user/repo, và bạn đã thiết lập git config url.ssh://git@server.com:.insteadOf https://server.com/:

bash

Sao chép mã

git ls-remote --get-url <tên\_remote>

ssh://git@server.com:user/repo

#### Mục 3.7: Thêm một kho lưu trữ từ xa mới

bash

Sao chép mã

git remote add upstream [url-kho-luu-tru-git]

Thêm kho lưu trữ git từ xa được biểu thị bởi url-kho-luu-tru-git làm remote mới có tên upstream vào kho lưu trữ git.

#### Mục 3.8: Thiết lập Upstream cho một nhánh mới

Bạn có thể tạo một nhánh mới và chuyển sang nó bằng:

bash

Sao chép mã

git checkout -b AP-57

Sau khi sử dụng git checkout để tạo một nhánh mới, bạn cần thiết lập upstream cho nhánh đó để có thể đẩy lên:

bash

Sao chép mã

git push --set-upstream origin AP-57

Sau đó, bạn có thể sử dụng git push khi đang ở trên nhánh đó.

#### Mục 3.9: Bắt đầu

Cú pháp để đẩy lên một nhánh từ xa:

bash

Sao chép mã

git push <tên\_remote> <tên\_nhánh>

Ví dụ:

bash

Sao chép mã

git push origin master

#### Mục 3.10: Đổi tên một Remote

Để đổi tên remote, sử dụng lệnh git remote rename. Lệnh này nhận hai đối số:

* Tên remote hiện tại, ví dụ: origin
* Tên mới cho remote, ví dụ: destination

Lấy tên remote hiện có:

bash

Sao chép mã

git remote

# origin

Kiểm tra remote hiện có với URL:

bash

Sao chép mã

git remote -v

# origin https://github.com/username/repo.git (fetch)

# origin https://github.com/username/repo.git (push)

Đổi tên remote:

bash

Sao chép mã

git remote rename origin destination

# Đổi tên remote từ 'origin' thành 'destination'

Xác minh tên mới:

bash

Sao chép mã

git remote -v

# destination https://github.com/username/repo.git (fetch)

# destination https://github.com/username/repo.git (push)

#### Các Lỗi Có Thể Xảy Ra

1. Không thể đổi tên phần cấu hình 'remote.[tên\_cũ]' thành 'remote.[tên\_mới]'
   * Lỗi này có nghĩa là remote bạn thử đổi tên không tồn tại.
2. Remote [tên\_mới] đã tồn tại.
   * Thông báo lỗi này tự giải thích.

#### Mục 3.11: Hiển thị thông tin về một Remote cụ thể

Xuất một số thông tin về một remote đã biết: origin

bash

Sao chép mã

git remote show origin

Chỉ in URL của remote:

bash

Sao chép mã

git config --get remote.origin.url

Với phiên bản 2.7+, bạn cũng có thể làm điều này, có thể tốt hơn so với lệnh trên sử dụng lệnh config:

bash

Sao chép mã

git remote get-url origin

#### Mục 3.12: Thiết lập URL cho một Remote cụ thể

Bạn có thể thay đổi URL của một remote hiện có bằng lệnh:

bash

Sao chép mã

git remote set-url <tên\_remote> <url>

#### Mục 3.13: Lấy URL cho một Remote cụ thể

Bạn có thể lấy URL của một remote hiện có bằng cách sử dụng lệnh:

bash

Sao chép mã

git remote get-url <tên>

Mặc định, điều này sẽ là:

bash

Sao chép mã

git remote get-url origin

#### Mục 3.14: Thay đổi một kho lưu trữ từ xa

Để thay đổi URL của kho lưu trữ mà bạn muốn remote của bạn trỏ tới, bạn có thể sử dụng tùy chọn set-url, như sau:

bash

Sao chép mã

git remote set-url <tên\_remote> <url\_kho\_luu\_tru\_từ\_xa>

Ví dụ:

bash

Sao chép mã

git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git

## Chương 4: Staging (Tạm thời)

#### Mục 4.1: Staging tất cả các thay đổi trong tệp

bash

Sao chép mã

git add -A

Phiên bản ≥ 2.0

bash

Sao chép mã

git add .

Trong phiên bản 2.x, git add . sẽ staging tất cả các thay đổi trong thư mục hiện tại và tất cả các thư mục con của nó. Tuy nhiên, trong phiên bản 1.x, nó chỉ staging các tệp mới và tệp đã chỉnh sửa, không bao gồm các tệp bị xóa. Sử dụng git add -A hoặc lệnh tương đương git add --all để staging tất cả các thay đổi trong bất kỳ phiên bản nào của git.

#### Mục 4.2: Bỏ staging một tệp chứa thay đổi

bash

Sao chép mã

git reset <filePath>

#### Mục 4.3: Thêm thay đổi theo từng phần

Bạn có thể xem những "hunks" (phần thay đổi) nào sẽ được staging để commit bằng cách sử dụng cờ patch:

bash

Sao chép mã

git add -p

hoặc

bash

Sao chép mã

git add --patch

Điều này mở ra một giao diện tương tác cho phép bạn xem các khác biệt và quyết định có bao gồm chúng hay không.

Tùy chọn khi staging một hunk:

* y staging hunk này cho lần commit tiếp theo
* n không staging hunk này cho lần commit tiếp theo
* q thoát; không staging hunk này hoặc bất kỳ hunk nào còn lại
* a staging hunk này và tất cả các hunk sau đó trong tệp
* d không staging hunk này hoặc bất kỳ hunk nào sau đó trong tệp
* g chọn một hunk để chuyển đến
* / tìm kiếm một hunk khớp với regex đã cho
* j để hunk này không quyết định, xem hunk chưa quyết định tiếp theo
* J để hunk này không quyết định, xem hunk tiếp theo
* k để hunk này không quyết định, xem hunk chưa quyết định trước đó
* K để hunk này không quyết định, xem hunk trước đó
* s chia hunk hiện tại thành các hunk nhỏ hơn
* e chỉnh sửa thủ công hunk hiện tại
* ? in trợ giúp về hunk

Điều này giúp dễ dàng bắt những thay đổi mà bạn không muốn commit.

Bạn cũng có thể mở thông qua git add --interactive và chọn p.

#### Mục 4.4: Thêm thay đổi tương tác

git add -i (hoặc --interactive) sẽ cung cấp cho bạn một giao diện tương tác để chỉnh sửa index, chuẩn bị những gì bạn muốn có trong lần commit tiếp theo. Bạn có thể thêm và xóa các thay đổi trong toàn bộ tệp, thêm các tệp chưa được theo dõi và xóa các tệp khỏi việc theo dõi, nhưng cũng có thể chọn các phần thay đổi để đưa vào index, bằng cách chọn các phần thay đổi, chia nhỏ các phần đó, hoặc thậm chí chỉnh sửa khác biệt. Nhiều công cụ commit đồ họa cho Git (như git gui) bao gồm tính năng này; điều này có thể dễ sử dụng hơn so với phiên bản dòng lệnh.

Nó rất hữu ích khi:

1. Bạn có các thay đổi lẫn lộn trong thư mục làm việc mà bạn muốn đặt trong các commit riêng biệt, không phải tất cả trong một commit duy nhất.
2. Bạn đang ở giữa một rebase tương tác và muốn chia nhỏ commit quá lớn.

bash

Sao chép mã

$ 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>

Nửa trên của đầu ra này hiển thị trạng thái hiện tại của index được chia thành các cột staged và unstaged:

1. index.js đã thêm 4 dòng và xóa 4 dòng. Hiện tại nó không được staging, vì trạng thái hiện tại báo cáo "unchanged." Khi tệp này được staging, phần +4/-4 sẽ được chuyển đến cột staged và cột unstaged sẽ đọc "nothing."
2. package.json đã thêm một dòng và đã được staging. Không có thay đổi nào thêm kể từ khi nó được staging như đã được chỉ ra bởi dòng "nothing" dưới cột unstaged.

Nửa dưới hiển thị những gì bạn có thể làm. Hoặc nhập một số (1-8) hoặc một chữ cái (s, u, r, a, p, d, q, h).

* status hiển thị đầu ra giống như phần trên của đầu ra trên.
* update cho phép bạn thực hiện các thay đổi bổ sung vào các commit đã staging với cú pháp bổ sung.
* revert sẽ hoàn nguyên thông tin commit đã staging về HEAD.
* add untracked cho phép bạn thêm các đường dẫn tệp chưa được theo dõi trước đó bởi kiểm soát phiên bản.
* patch cho phép chọn một đường dẫn từ đầu ra tương tự như trạng thái để phân tích thêm.
* diff hiển thị những gì sẽ được commit.
* quit thoát lệnh.
* help hiển thị trợ giúp thêm về việc sử dụng lệnh này.

#### Mục 4.5: Hiển thị các thay đổi đã staging

Để hiển thị các phần đã staging cho commit:

bash

Sao chép mã

git diff --cached

#### Mục 4.6: Staging một tệp đơn lẻ

Để staging một tệp để commit, chạy:

bash

Sao chép mã

git add <filename>

#### Mục 4.7: Staging các tệp đã xóa

bash

Sao chép mã

git rm <filename>

Để xóa tệp khỏi git mà không xóa nó khỏi đĩa, sử dụng cờ --cached

bash

Sao chép mã

git rm --cached <filename>

## Chapter 5: Bỏ qua các tệp và thư mục

Chủ đề này minh họa cách tránh thêm các tệp không mong muốn (hoặc thay đổi tệp) vào kho Git. Có nhiều cách (global hoặc local .gitignore, .git/exclude, git update-index --assume-unchanged, và git update-index --skip-tree), nhưng hãy nhớ rằng Git quản lý nội dung, có nghĩa là: bỏ qua thực sự bỏ qua nội dung của thư mục (tức là các tệp). Một thư mục trống sẽ bị bỏ qua theo mặc định, vì nó không thể được thêm vào.

### Section 5.1: Bỏ qua các tệp và thư mục với tệp .gitignore

Bạn có thể khiến Git bỏ qua các tệp và thư mục nhất định — tức là, loại trừ chúng khỏi việc được Git theo dõi — bằng cách tạo một hoặc nhiều tệp .gitignore trong kho của bạn.

Trong các dự án phần mềm, .gitignore thường chứa danh sách các tệp và/hoặc thư mục được tạo ra trong quá trình xây dựng hoặc khi chạy. Các mục trong tệp .gitignore có thể bao gồm tên hoặc đường dẫn trỏ đến:

1. Tài nguyên tạm thời ví dụ như bộ nhớ cache, tệp nhật ký, mã đã biên dịch, v.v.
2. Tệp cấu hình cục bộ không nên chia sẻ với các nhà phát triển khác
3. Các tệp chứa thông tin bí mật, chẳng hạn như mật khẩu đăng nhập, khóa và thông tin xác thực

Khi được tạo trong thư mục cấp cao nhất, các quy tắc sẽ được áp dụng đệ quy cho tất cả các tệp và thư mục con trong toàn bộ kho. Khi được tạo trong một thư mục con, các quy tắc sẽ áp dụng cho thư mục cụ thể đó và các thư mục con của nó.

Khi một tệp hoặc thư mục bị bỏ qua, nó sẽ không:

1. Được Git theo dõi
2. Được báo cáo bởi các lệnh như git status hoặc git diff
3. Được thêm vào bằng các lệnh như git add -A

Trong trường hợp không phổ biến mà bạn cần bỏ qua các tệp đã được theo dõi, hãy cẩn thận đặc biệt. Xem: Bỏ qua các tệp đã được commit vào kho Git.

Ví dụ:

Dưới đây là một số ví dụ chung về các quy tắc trong tệp .gitignore, dựa trên mẫu tệp glob:

less

Sao chép mã

# Các dòng bắt đầu bằng `#` là các chú thích.

# Bỏ qua các tệp có tên 'file.ext'

file.ext

# Chú thích không thể ở cùng dòng với các quy tắc!

# Dòng sau đây bỏ qua các tệp có tên 'file.ext # not a comment'

file.ext # not a comment

# Bỏ qua các tệp với đường dẫn đầy đủ.

# Điều này khớp với các tệp trong thư mục gốc và các thư mục con.

# Tức là otherfile.ext sẽ bị bỏ qua ở bất kỳ đâu trên cây thư mục.

dir/otherdir/file.ext

otherfile.ext

# Bỏ qua các thư mục

# Cả thư mục và nội dung của nó sẽ bị bỏ qua.

bin/

gen/

# Mẫu glob cũng có thể được sử dụng ở đây để bỏ qua các đường dẫn có ký tự nhất định.

# Ví dụ, quy tắc dưới đây sẽ khớp với cả build/ và Build/

[bB]uild/

# Không có dấu gạch chéo cuối, quy tắc sẽ khớp với cả tệp và/hoặc

# một thư mục, vì vậy quy tắc sau sẽ bỏ qua cả tệp có tên `gen`

# và một thư mục có tên `gen`, cũng như bất kỳ nội dung nào của thư mục đó

bin

gen

# Bỏ qua các tệp theo phần mở rộng

# Tất cả các tệp có các phần mở rộng này sẽ bị bỏ qua trong

# thư mục này và tất cả các thư mục con của nó.

\*.apk

\*.class

# Có thể kết hợp cả hai hình thức để bỏ qua các tệp có phần

# mở rộng nhất định trong các thư mục nhất định. Các quy tắc sau

# sẽ thừa với các quy tắc chung đã được định nghĩa ở trên.

java/\*.apk

gen/\*.class

# Để bỏ qua các tệp chỉ ở thư mục cấp cao nhất, nhưng không ở các

# thư mục con, hãy thêm ký tự `/` trước quy tắc

/\*.apk

/\*.class

# Để bỏ qua bất kỳ thư mục nào có tên DirectoryA

# ở bất kỳ độ sâu nào, hãy sử dụng \*\* trước DirectoryA

# Đừng quên dấu gạch chéo cuối,

# Nếu không, nó sẽ bỏ qua tất cả các tệp có tên DirectoryA, thay vì các thư mục

\*\*/DirectoryA/

# Điều này sẽ bỏ qua

# DirectoryA/

# DirectoryB/DirectoryA/

# DirectoryC/DirectoryB/DirectoryA/

# Nó sẽ không bỏ qua một tệp có tên DirectoryA, ở bất kỳ cấp độ nào

# Để bỏ qua bất kỳ thư mục nào có tên DirectoryB trong một

# thư mục có tên DirectoryA với bất kỳ số lượng

# thư mục nào ở giữa, hãy sử dụng \*\* giữa các thư mục

DirectoryA/\*\*/DirectoryB/

# Điều này sẽ bỏ qua

# DirectoryA/DirectoryB/

# DirectoryA/DirectoryQ/DirectoryB/

# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/

# Để bỏ qua một tập hợp các tệp, có thể sử dụng ký tự đại diện, như đã thấy ở trên.

# Một ký tự '\*' duy nhất sẽ bỏ qua tất cả mọi thứ trong thư mục của bạn, bao gồm cả tệp .gitignore của bạn.

# Để loại trừ các tệp cụ thể khi sử dụng ký tự đại diện, hãy phủ định chúng.

# Vì vậy, chúng được loại trừ khỏi danh sách bỏ qua:

!.gitignore

# Sử dụng ký tự backslash làm ký tự thoát để bỏ qua các tệp có dấu thăng (#)

# (hỗ trợ từ phiên bản 1.6.2.1)

\#\*#

Hầu hết các tệp .gitignore là tiêu chuẩn trên các ngôn ngữ khác nhau, vì vậy để bắt đầu, đây là một bộ tệp .gitignore mẫu được liệt kê theo ngôn ngữ mà bạn có thể sao chép hoặc chỉnh sửa cho dự án của mình. Ngoài ra, đối với một dự án mới, bạn có thể cân nhắc tự động tạo một tệp khởi đầu bằng cách sử dụng một công cụ trực tuyến.

Các dạng khác của .gitignore

Tệp .gitignore được dự định được commit như một phần của kho lưu trữ. Nếu bạn muốn bỏ qua một số tệp mà không cần commit các quy tắc bỏ qua, đây là một số tùy chọn:

1. Chỉnh sửa tệp .git/info/exclude (sử dụng cùng cú pháp với .gitignore). Các quy tắc sẽ có phạm vi toàn cầu trong kho lưu trữ;
2. Thiết lập một tệp gitignore toàn cầu sẽ áp dụng các quy tắc bỏ qua cho tất cả các kho lưu trữ cục bộ của bạn:

Ngoài ra, bạn có thể bỏ qua các thay đổi cục bộ đối với các tệp đã được theo dõi mà không cần thay đổi cấu hình git toàn cầu bằng cách:

less

Sao chép mã

git update-index --skip-worktree [<file>...]: đối với các thay đổi cục bộ nhỏ

git update-index --assume-unchanged [<file>...]: đối với các tệp sẵn sàng cho sản xuất, không thay đổi upstream

Xem thêm chi tiết về sự khác biệt giữa các cờ này và tài liệu git update-index để biết thêm các tùy chọn.

Dọn dẹp các tệp bị bỏ qua

Bạn có thể sử dụng git clean -X để dọn dẹp các tệp bị bỏ qua:

css

Sao chép mã

git clean -Xn # hiển thị danh sách các tệp bị bỏ qua

git clean -Xf # xóa các tệp đã hiển thị trước đó

Lưu ý: -X (chữ in hoa) chỉ dọn dẹp các tệp bị bỏ qua. Sử dụng -x (không in hoa) để cũng xóa các tệp chưa được theo dõi.

Xem tài liệu git clean để biết thêm chi tiết.

Xem tài liệu Git để biết thêm chi tiết.

### Section 5.2: Kiểm tra xem một tệp có bị bỏ qua hay không

Lệnh git check-ignore báo cáo về các tệp bị Git bỏ qua. Bạn có thể truyền các tên tệp vào dòng lệnh, và git check-ignore sẽ liệt kê các tên tệp bị bỏ qua. Ví dụ:

bash

Sao chép mã

$ cat .gitignore

\*.o

$ git check-ignore example.o Readme.md

example.o

Ở đây, chỉ có các tệp \*.o được định nghĩa trong .gitignore, vì vậy Readme.md không được liệt kê trong đầu ra của git check-ignore.

Nếu bạn muốn thấy dòng nào trong .gitignore chịu trách nhiệm bỏ qua một tệp, thêm -v vào lệnh git check-ignore:

bash

Sao chép mã

$ git check-ignore -v example.o Readme.md

.gitignore:1:\*.o example.o

Từ Git 1.7.6 trở đi, bạn cũng có thể sử dụng git status --ignored để xem các tệp bị bỏ qua. Bạn có thể tìm thêm thông tin về điều này trong tài liệu chính thức hoặc trong phần Finding files ignored by .gitignore.

### Section 5.3: Ngoại lệ trong tệp .gitignore

Nếu bạn bỏ qua các tệp bằng cách sử dụng một mẫu nhưng có ngoại lệ, hãy thêm dấu chấm than (!) vào ngoại lệ. Ví dụ:

plaintext

Sao chép mã

\*.txt

!important.txt

Ví dụ trên yêu cầu Git bỏ qua tất cả các tệp có phần mở rộng .txt ngoại trừ các tệp có tên important.txt.

Nếu tệp nằm trong một thư mục bị bỏ qua, bạn không thể dễ dàng bao gồm lại nó như vậy:

plaintext

Sao chép mã

folder/

!folder/\*.txt

Trong ví dụ này, tất cả các tệp .txt trong thư mục sẽ vẫn bị bỏ qua.

Cách đúng là bao gồm lại chính thư mục đó trên một dòng riêng biệt, sau đó bỏ qua tất cả các tệp trong thư mục bằng \*, cuối cùng bao gồm lại các tệp .txt trong thư mục, như sau:

plaintext

Sao chép mã

!folder/

folder/\*

!folder/\*.txt

Lưu ý: Đối với các tên tệp bắt đầu bằng dấu chấm than, thêm hai dấu chấm than hoặc thoát bằng ký tự \:

plaintext

Sao chép mã

!!includethis

\!excludethis

### Section 5.4: Tệp .gitignore toàn cầu

Để Git bỏ qua các tệp nhất định trên tất cả các kho lưu trữ, bạn có thể tạo một tệp .gitignore toàn cầu với lệnh sau trong terminal hoặc command prompt của bạn:

bash

Sao chép mã

$ git config --global core.excludesfile <Path\_To\_Global\_gitignore\_file>

Git sẽ sử dụng tệp này bên cạnh mỗi tệp .gitignore của từng kho lưu trữ. Các quy tắc cho điều này là:

* Nếu tệp .gitignore cục bộ rõ ràng bao gồm một tệp trong khi tệp .gitignore toàn cầu bỏ qua nó, thì tệp .gitignore cục bộ sẽ ưu tiên (tệp sẽ được bao gồm).
* Nếu kho lưu trữ được clone trên nhiều máy, thì tệp .gitignore toàn cầu phải được tải trên tất cả các máy hoặc ít nhất bao gồm nó, vì các tệp bị bỏ qua sẽ được đẩy lên kho lưu trữ trong khi máy tính có tệp .gitignore toàn cầu sẽ không cập nhật nó. Đây là lý do tại sao một tệp .gitignore cụ thể của kho lưu trữ là một ý tưởng tốt hơn so với một tệp toàn cầu nếu dự án được làm việc bởi một nhóm.

Tệp này là một nơi tốt để giữ các mục bỏ qua theo nền tảng, máy hoặc người dùng, ví dụ như OSX .DS\_Store, Windows Thumbs.db hoặc bỏ qua các tệp Vim \*.ext~ và \*.ext.swp nếu bạn không muốn giữ chúng trong kho lưu trữ. Vì vậy, một thành viên trong nhóm làm việc trên OS X có thể thêm tất cả .DS\_STORE và \_MACOSX (thực tế là vô ích), trong khi một thành viên khác trong nhóm làm việc trên Windows có thể bỏ qua tất cả thumbs.db.

### Section 5.5: Bỏ qua các tệp đã được commit vào kho Git

Nếu bạn đã thêm một tệp vào kho Git của mình và bây giờ muốn dừng theo dõi nó (để nó không có mặt trong các lần commit tiếp theo), bạn có thể xóa nó khỏi index:

bash

Sao chép mã

git rm --cached <file>

Điều này sẽ xóa tệp khỏi kho lưu trữ và ngăn các thay đổi tiếp theo được Git theo dõi. Tùy chọn --cached sẽ đảm bảo rằng tệp không bị xóa vật lý.

Lưu ý rằng các nội dung đã thêm trước đó của tệp vẫn sẽ có thể xem được thông qua lịch sử Git.

Hãy nhớ rằng nếu ai đó khác pull từ kho lưu trữ sau khi bạn xóa tệp khỏi index, bản sao của họ sẽ bị xóa vật lý.

Bạn có thể làm cho Git giả vờ rằng phiên bản của tệp trong thư mục làm việc là cập nhật và đọc phiên bản index thay vào đó (do đó bỏ qua các thay đổi trong nó) với bit "skip worktree":

bash

Sao chép mã

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

Việc ghi không bị ảnh hưởng bởi bit này, an toàn nội dung vẫn là ưu tiên hàng đầu. Bạn sẽ không bao giờ mất các thay đổi bị bỏ qua quý giá của mình; mặt khác, bit này xung đột với stashing: để loại bỏ bit này, sử dụng:

bash

Sao chép mã

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

Đôi khi được khuyến nghị sai rằng nói dối với Git và làm cho nó giả định rằng tệp không thay đổi mà không kiểm tra nó. Điều này trông giống như bỏ qua bất kỳ thay đổi nào đối với tệp, mà không xóa nó khỏi index:

bash

Sao chép mã

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

Điều này sẽ buộc git bỏ qua bất kỳ thay đổi nào được thực hiện trong tệp (hãy nhớ rằng nếu bạn pull bất kỳ thay đổi nào đối với tệp này, hoặc bạn stash nó, các thay đổi bị bỏ qua của bạn sẽ bị mất).

Nếu bạn muốn git "quan tâm" đến tệp này một lần nữa, hãy chạy lệnh sau:

bash

Sao chép mã

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

Top of Form

Bottom of Form

### Section 5.6: Bỏ qua các tệp cục bộ mà không cần commit các quy tắc bỏ qua

Tệp .gitignore bỏ qua các tệp cục bộ, nhưng nó được dự định sẽ được commit vào kho và chia sẻ với các cộng tác viên và người dùng khác. Bạn có thể thiết lập một tệp .gitignore toàn cầu, nhưng khi đó tất cả các kho lưu trữ của bạn sẽ chia sẻ những thiết lập đó. Nếu bạn muốn bỏ qua các tệp nhất định trong một kho lưu trữ cục bộ và không đưa tệp đó vào bất kỳ kho nào, hãy chỉnh sửa tệp .git/info/exclude bên trong kho của bạn.

Ví dụ:

plaintext

Sao chép mã

# những tệp này chỉ bị bỏ qua trên kho này

# những quy tắc này không được chia sẻ với bất kỳ ai

# vì chúng là cá nhân

gtk\_tests.py

gui/gtk/tests/\*

localhost

pushReports.py

server/

### Section 5.7: Bỏ qua các thay đổi tiếp theo đối với một tệp (mà không xóa nó)

Đôi khi bạn muốn giữ một tệp trong Git nhưng bỏ qua các thay đổi tiếp theo.

Hãy cho Git biết bỏ qua các thay đổi đối với một tệp hoặc thư mục bằng cách sử dụng update-index:

bash

Sao chép mã

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

Lệnh trên yêu cầu Git giả định rằng my-file.txt không thay đổi, và không kiểm tra hoặc báo cáo các thay đổi. Tệp vẫn còn trong kho lưu trữ.

Điều này có thể hữu ích để cung cấp các giá trị mặc định và cho phép ghi đè môi trường cục bộ, ví dụ:

bash

Sao chép mã

# tạo một tệp với một số giá trị

cat <<EOF

MYSQL\_USER=app

MYSQL\_PASSWORD=FIXME\_SECRET\_PASSWORD

EOF > .env

# commit vào Git

git add .env

git commit -m "Adding .env template"

# bỏ qua các thay đổi trong tương lai đối với .env

git update-index --assume-unchanged .env

# cập nhật mật khẩu của bạn

vi .env

# không có thay đổi!

git status

### Section 5.8: Bỏ qua một tệp trong bất kỳ thư mục nào

Để bỏ qua một tệp foo.txt trong bất kỳ thư mục nào, bạn chỉ cần viết tên của nó:

plaintext

Sao chép mã

foo.txt # khớp với tất cả các tệp 'foo.txt' trong bất kỳ thư mục nào

Nếu bạn muốn bỏ qua tệp chỉ trong một phần của cây thư mục, bạn có thể chỉ định các thư mục con của một thư mục cụ thể với mẫu \*\*:

plaintext

Sao chép mã

bar/\*\*/foo.txt # khớp với tất cả các tệp 'foo.txt' trong 'bar' và tất cả các thư mục con

Hoặc bạn có thể tạo một tệp .gitignore trong thư mục bar/. Tương đương với ví dụ trước là tạo tệp bar/.gitignore với các nội dung sau:

plaintext

Sao chép mã

foo.txt # khớp với tất cả các tệp 'foo.txt' trong bất kỳ thư mục nào dưới bar/

### Section 5.9: Các mẫu .gitignore được điền sẵn

Nếu bạn không chắc chắn nên liệt kê những quy tắc nào trong tệp .gitignore của mình, hoặc bạn chỉ muốn thêm các ngoại lệ được chấp nhận chung vào dự án của mình, bạn có thể chọn hoặc tạo tệp .gitignore:

* [gitignore.io](https://www.gitignore.io/)
* [GitHub gitignore](https://github.com/github/gitignore)

Nhiều dịch vụ lưu trữ như GitHub và BitBucket cung cấp khả năng tạo các tệp .gitignore dựa trên các ngôn ngữ lập trình và IDE mà bạn có thể đang sử dụng.

### Section 5.10: Bỏ qua các tệp trong các thư mục con (Nhiều tệp .gitignore)

Giả sử bạn có cấu trúc kho lưu trữ như sau:

plaintext

Sao chép mã

examples/

output.log

src/

<files not shown>

output.log

README.md

Tệp output.log trong thư mục examples là hợp lệ và cần thiết cho dự án để thu thập thông tin trong khi tệp bên dưới src/ được tạo trong quá trình gỡ lỗi và không nên có trong lịch sử hoặc một phần của kho lưu trữ.

Có hai cách để bỏ qua tệp này. Bạn có thể đặt đường dẫn tuyệt đối vào tệp .gitignore ở gốc thư mục làm việc:

plaintext

Sao chép mã

# /.gitignore

src/output.log

Ngoài ra, bạn có thể tạo một tệp .gitignore trong thư mục src/ và bỏ qua tệp tương ứng với .gitignore này:

plaintext

Sao chép mã

# /src/.gitignore

output.log

### Section 5.11: Tạo một thư mục trống

Không thể thêm và commit một thư mục trống trong Git vì Git quản lý các tệp và đính kèm thư mục của chúng vào, giúp giảm kích thước commit và cải thiện tốc độ. Để giải quyết vấn đề này, có hai phương pháp:

**Phương pháp một: .gitkeep**

Một cách để giải quyết vấn đề này là sử dụng tệp .gitkeep để đăng ký thư mục cho Git. Để làm điều này, chỉ cần tạo thư mục cần thiết và thêm một tệp .gitkeep vào thư mục đó. Tệp này trống và không phục vụ mục đích nào khác ngoài việc đăng ký thư mục. Để làm điều này trên Windows (vì có các quy tắc đặt tên tệp không thân thiện), chỉ cần mở Git Bash trong thư mục và chạy lệnh:

bash

Sao chép mã

$ touch .gitkeep

Lệnh này chỉ tạo một tệp .gitkeep trống trong thư mục hiện tại.

**Phương pháp hai: dummy.txt**

Một cách khác rất giống với trên và các bước có thể theo dõi, nhưng thay vì .gitkeep, chỉ cần sử dụng dummy.txt thay thế. Điều này có thêm lợi ích là có thể dễ dàng tạo nó trong Windows bằng cách sử dụng menu ngữ cảnh. Và bạn cũng có thể để lại những thông điệp hài hước trong đó. Bạn cũng có thể sử dụng tệp .gitkeep để theo dõi thư mục trống. .gitkeep thường là một tệp trống được thêm vào để theo dõi thư mục trống.

### Section 5.12: Tìm các tệp bị bỏ qua bởi .gitignore

Bạn có thể liệt kê tất cả các tệp bị Git bỏ qua trong thư mục hiện tại bằng lệnh:

bash

Sao chép mã

git status --ignored

Vì vậy, nếu chúng ta có cấu trúc kho lưu trữ như thế này:

plaintext

Sao chép mã

.git

.gitignore

./example\_1

./dir/example\_2

./example\_2

Và tệp .gitignore chứa:

plaintext

Sao chép mã

example\_2

Kết quả của lệnh sẽ là:

bash

Sao chép mã

$ 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

Nếu bạn muốn liệt kê các tệp bị bỏ qua trong các thư mục một cách đệ quy, bạn phải sử dụng tham số bổ sung --untracked-files=all:

Kết quả sẽ trông như sau:

bash

Sao chép mã

$ 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

### Section 5.13: Bỏ qua một phần của tệp [stub]

Đôi khi bạn có thể muốn có các thay đổi cục bộ trong một tệp mà bạn không muốn commit hoặc công bố. Lý tưởng nhất là các thiết lập cục bộ nên tập trung trong một tệp riêng có thể được đặt vào .gitignore, nhưng đôi khi như một giải pháp ngắn hạn, có thể hữu ích khi có một cái gì đó cục bộ trong một tệp đã được kiểm tra.

Bạn có thể làm cho Git "không thấy" những dòng này bằng cách sử dụng bộ lọc clean. Chúng sẽ không hiển thị ngay cả trong các bản diff.

Giả sử đây là đoạn mã từ tệp file1.c:

c

Sao chép mã

struct settings s;

s.host = "localhost";

s.port = 5653;

s.auth = 1;

s.port = 15653; // NOCOMMIT

s.debug = 1; // NOCOMMIT

s.auth = 0; // NOCOMMIT

Bạn không muốn công bố các dòng NOCOMMIT ở bất kỳ đâu.

Tạo bộ lọc "nocommit" bằng cách thêm điều này vào tệp cấu hình Git như .git/config:

plaintext

Sao chép mã

[filter "nocommit"]

clean=grep -v NOCOMMIT

Thêm (hoặc tạo) điều này vào .git/info/attributes hoặc .gitmodules:

plaintext

Sao chép mã

file1.c filter=nocommit

Và các dòng NOCOMMIT của bạn sẽ bị ẩn khỏi Git.

**Lưu ý:**

* Sử dụng bộ lọc clean làm chậm quá trình xử lý các tệp, đặc biệt là trên Windows.
* Dòng bị bỏ qua có thể biến mất khỏi tệp khi Git cập nhật nó. Điều này có thể được khắc phục bằng bộ lọc smudge, nhưng nó phức tạp hơn.
* Chưa được thử nghiệm trên Windows.

### Section 5.14: Bỏ qua các thay đổi trong các tệp đã được theo dõi [stub]

.gitignore và .git/info/exclude chỉ hoạt động với các tệp chưa được theo dõi.

Để đặt cờ bỏ qua trên một tệp đã được theo dõi, sử dụng lệnh update-index:

bash

Sao chép mã

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

Để hoàn nguyên, sử dụng:

bash

Sao chép mã

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

Bạn có thể thêm đoạn mã này vào cấu hình git toàn cầu của mình để có các lệnh git hide, git unhide và git hidden tiện lợi hơn:

plaintext

Sao chép mã

[alias]

hide = update-index --skip-worktree

unhide = update-index --no-skip-worktree

hidden = "!git ls-files -v | grep ^[hsS] | cut -c 3-"

Bạn cũng có thể sử dụng tùy chọn --assume-unchanged với chức năng update-index:

bash

Sao chép mã

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

Nếu bạn muốn Git theo dõi tệp này lại để xem các thay đổi, sử dụng:

bash

Sao chép mã

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

Khi cờ --assume-unchanged được chỉ định, người dùng hứa sẽ không thay đổi tệp và cho phép Git giả định rằng tệp trong cây làm việc khớp với những gì được ghi lại trong index. Git sẽ thất bại nếu cần sửa đổi tệp này trong index, ví dụ khi gộp một commit; do đó, trong trường hợp tệp được giả định là chưa theo dõi bị thay đổi upstream, bạn sẽ cần xử lý tình huống này theo cách thủ công. Trọng tâm là hiệu suất trong trường hợp này.

Trong khi cờ --skip-worktree hữu ích khi bạn chỉ đạo Git không chạm vào một tệp cụ thể nào đó vì tệp này sẽ được thay đổi cục bộ và bạn không muốn vô tình commit các thay đổi (ví dụ như tệp cấu hình/properties được cấu hình cho một môi trường cụ thể). Skip-worktree sẽ được ưu tiên hơn assume-unchanged khi cả hai được thiết lập.

### Section 5.15: Xóa các tệp đã được commit nhưng nằm trong .gitignore

Đôi khi xảy ra tình huống một tệp đã được Git theo dõi, nhưng sau đó được thêm vào .gitignore để ngừng theo dõi nó. Đây là kịch bản rất phổ biến khi quên dọn dẹp các tệp như vậy trước khi thêm vào .gitignore. Trong trường hợp này, tệp cũ sẽ vẫn tồn tại trong kho lưu trữ.

Để khắc phục vấn đề này, bạn có thể thực hiện việc "dry-run" loại bỏ mọi thứ trong kho lưu trữ, sau đó thêm lại tất cả các tệp. Miễn là bạn không có thay đổi nào đang chờ xử lý và tham số --cached được sử dụng, lệnh này khá an toàn để chạy:

1. **Xóa mọi thứ khỏi index (các tệp sẽ vẫn còn trong hệ thống tệp):**

bash

Sao chép mã

$ git rm -r --cached .

1. **Thêm lại mọi thứ (chúng sẽ được thêm vào trạng thái hiện tại, bao gồm cả các thay đổi):**

bash

Sao chép mã

$ git add .

1. **Commit, nếu có bất kỳ thay đổi nào. Bạn chỉ nên thấy các tệp bị xóa:**

bash

Sao chép mã

$ git commit -m 'Remove all files that are in the .gitignore'

1. **Cập nhật remote:**

bash

Sao chép mã

$ git push origin master

Các bước này sẽ giúp bạn dọn dẹp các tệp đã được commit nhưng hiện tại nằm trong .gitignore và đảm bảo rằng kho lưu trữ của bạn không chứa các tệp không mong muốn.

## Chapter 6: Git Diff

Parameter Details

* -p, -u, --patch: Tạo bản vá (patch).
* -s, --no-patch: Bỏ qua đầu ra diff. Hữu ích cho các lệnh như git show hiển thị bản vá theo mặc định, hoặc để hủy bỏ hiệu ứng của --patch.
* --raw: Tạo diff ở định dạng raw.
* --diff-algorithm=: Chọn thuật toán diff. Các biến thể bao gồm: myers, minimal, patience, histogram.
* --summary: Xuất bản tóm tắt thông tin tiêu đề mở rộng như tạo mới, đổi tên và thay đổi chế độ.
* --name-only: Chỉ hiển thị tên của các tệp đã thay đổi.
* --name-status: Hiển thị tên và trạng thái của các tệp đã thay đổi. Các trạng thái phổ biến nhất là M (Modified), A (Added), và D (Deleted).
* --check: Cảnh báo nếu các thay đổi tạo ra các dấu hiệu xung đột hoặc lỗi khoảng trắng. Lỗi khoảng trắng được kiểm soát bởi cấu hình core.whitespace. Theo mặc định, khoảng trắng ở cuối dòng (bao gồm cả các dòng chỉ chứa khoảng trắng) và ký tự khoảng trắng ngay sau ký tự tab trong phần đầu dòng được coi là lỗi khoảng trắng. Thoát với trạng thái khác 0 nếu phát hiện vấn đề. Không tương thích với --exit-code.
* --full-index: Thay vì chỉ hiển thị một vài ký tự đầu tiên, hiển thị đầy đủ tên đối tượng blob pre- và post-image trên dòng "index" khi tạo đầu ra ở định dạng patch.
* --binary: Ngoài --full-index, tạo một diff nhị phân có thể được áp dụng với git apply.
* -a, --text: Xem tất cả các tệp như văn bản.
* --color: Đặt chế độ màu; ví dụ, sử dụng --color=always nếu bạn muốn truyền một diff tới less và giữ nguyên màu của Git.

### Section 6.1: Hiển thị sự khác biệt trong nhánh làm việc

bash

Sao chép mã

git diff

Điều này sẽ hiển thị các thay đổi chưa được stage trên nhánh hiện tại từ commit trước đó. Nó chỉ hiển thị các thay đổi so với index, nghĩa là nó hiển thị những gì bạn có thể thêm vào commit tiếp theo nhưng chưa làm. Để thêm (stage) các thay đổi này, bạn có thể sử dụng git add.

Nếu một tệp đã được stage, nhưng bị sửa đổi sau khi được stage, git diff sẽ hiển thị sự khác biệt giữa tệp hiện tại và phiên bản đã được stage.

### Section 6.2: Hiển thị các thay đổi giữa hai commit

bash

Sao chép mã

git diff 1234abc..6789def # old new

Ví dụ: Hiển thị các thay đổi được thực hiện trong 3 commit cuối cùng:

bash

Sao chép mã

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

Lưu ý: Hai dấu chấm (..) là tùy chọn, nhưng thêm vào để làm rõ ràng.

Điều này sẽ hiển thị sự khác biệt văn bản giữa các commit, bất kể chúng nằm ở đâu trong cây.

### Section 6.3: Hiển thị sự khác biệt cho các tệp đã được stage

bash

Sao chép mã

git diff --staged

Điều này sẽ hiển thị các thay đổi giữa commit trước và các tệp hiện đang được stage.

LƯU Ý: Bạn cũng có thể sử dụng các lệnh sau để đạt được điều tương tự:

bash

Sao chép mã

git diff --cached

Là một từ đồng nghĩa với --staged hoặc

bash

Sao chép mã

git status -v

Sẽ kích hoạt các thiết lập chi tiết của lệnh trạng thái.

### Section 6.4: So sánh các nhánh

Hiển thị các thay đổi giữa điểm cuối của nhánh new và điểm cuối của nhánh original:

bash

Sao chép mã

git diff original new # tương đương với original..new

Hiển thị tất cả các thay đổi trên nhánh new kể từ khi nó tách ra từ nhánh original:

bash

Sao chép mã

git diff original...new # tương đương với $(git merge-base original new)..new

Sử dụng chỉ một tham số như

bash

Sao chép mã

git diff original

Tương đương với

bash

Sao chép mã

git diff original..HEAD

### Section 6.5: Hiển thị cả các thay đổi đã được stage và chưa được stage

Để hiển thị tất cả các thay đổi đã được stage và chưa được stage, sử dụng:

bash

Sao chép mã

git diff HEAD

LƯU Ý: Bạn cũng có thể sử dụng lệnh sau:

bash

Sao chép mã

git status -vv

Điểm khác biệt là đầu ra của lệnh sau sẽ thực sự cho bạn biết những thay đổi nào đã được stage để commit và những thay đổi nào chưa.

### Section 6.6: Hiển thị sự khác biệt cho một tệp hoặc thư mục cụ thể

bash

Sao chép mã

git diff myfile.txt

Hiển thị các thay đổi giữa commit trước của tệp đã chỉ định (myfile.txt) và phiên bản đã được sửa đổi cục bộ nhưng chưa được stage.

Điều này cũng hoạt động cho các thư mục:

bash

Sao chép mã

git diff documentation

Lệnh trên sẽ hiển thị các thay đổi giữa commit trước của tất cả các tệp trong thư mục đã chỉ định (documentation/) và các phiên bản đã được sửa đổi cục bộ của những tệp này, nhưng chưa được stage.

Để hiển thị sự khác biệt giữa một phiên bản của tệp trong commit đã cho và phiên bản HEAD cục bộ, bạn có thể chỉ định commit mà bạn muốn so sánh:

bash

Sao chép mã

git diff 27fa75e myfile.txt

Hoặc nếu bạn muốn xem sự khác biệt giữa hai commit riêng biệt:

bash

Sao chép mã

git diff 27fa75e ada9b57 myfile.txt

Để hiển thị sự khác biệt giữa phiên bản được chỉ định bởi hash ada9b57 và commit mới nhất trên nhánh my\_branchname cho chỉ thư mục tương đối được gọi là my\_changed\_directory/, bạn có thể làm như sau:

bash

Sao chép mã

git diff ada9b57 my\_branchname my\_changed\_directory/

### Section 6.7: Xem sự khác biệt từng từ cho các dòng dài

bash

Sao chép mã

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

Thay vì hiển thị các dòng đã thay đổi, lệnh này sẽ hiển thị các sự khác biệt trong từng từ. Ví dụ, thay vì:

plaintext

Sao chép mã

-Hello world

+Hello world!

Nơi toàn bộ dòng được đánh dấu là đã thay đổi, word-diff thay đổi đầu ra thành:

plaintext

Sao chép mã

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

Bạn có thể bỏ qua các dấu hiệu [-, -], {+, +} bằng cách chỉ định --word-diff=color hoặc --color-words. Điều này sẽ chỉ sử dụng mã màu để đánh dấu sự khác biệt:

### Section 6.8: Hiển thị sự khác biệt giữa phiên bản hiện tại và phiên bản trước đó

bash

Sao chép mã

git diff HEAD^ HEAD

Điều này sẽ hiển thị các thay đổi giữa commit trước và commit hiện tại.

### Section 6.9: Tạo một diff tương thích với patch

Đôi khi bạn chỉ cần một diff để áp dụng bằng cách sử dụng patch. Diff thông thường của Git không hoạt động. Hãy thử điều này thay thế:

bash

Sao chép mã

git diff --no-prefix > some\_file.patch

Sau đó ở một nơi khác bạn có thể áp dụng ngược lại:

bash

Sao chép mã

### patch -p0 < some\_file.patchSection 6.10: Sự khác biệt giữa hai commit hoặc nhánh

Để xem sự khác biệt giữa hai nhánh:

bash

Sao chép mã

git diff <branch1>..<branch2>

Để xem sự khác biệt giữa hai commit:

bash

Sao chép mã

git diff <commitId1>..<commitId2>

Để xem sự khác biệt với nhánh hiện tại:

bash

Sao chép mã

git diff <branch/commitId>

Để xem tóm tắt các thay đổi:

bash

Sao chép mã

git diff --stat <branch/commitId>

Để xem các tệp đã thay đổi sau một commit nhất định:

bash

Sao chép mã

git diff --name-only <commitId>

Để xem các tệp khác nhau so với một nhánh:

bash

Sao chép mã

git diff --name-only <branchName>

Để xem các tệp đã thay đổi trong một thư mục sau một commit nhất định:

bash

Sao chép mã

git diff --name-only <commitId> <folder\_path>

### Section 6.11: Sử dụng meld để xem tất cả các thay đổi trong thư mục làm việc

bash

Sao chép mã

git difftool -t meld --dir-diff

Lệnh này sẽ hiển thị các thay đổi trong thư mục làm việc. Ngoài ra,

bash

Sao chép mã

git difftool -t meld --dir-diff [COMMIT\_A] [COMMIT\_B]

Lệnh này sẽ hiển thị sự khác biệt giữa hai commit cụ thể.

### Section 6.12: So sánh các tệp mã hóa UTF-16 và các tệp plist nhị phân

Bạn có thể so sánh các tệp mã hóa UTF-16 (ví dụ như tệp chuỗi localization của iOS và macOS) bằng cách chỉ định cách Git nên so sánh các tệp này.

Thêm đoạn mã sau vào tệp ~/.gitconfig của bạn:

plaintext

Sao chép mã

[diff "utf16"]

textconv = "iconv -f utf-16 -t utf-8"

iconv là một chương trình để chuyển đổi các mã hóa khác nhau.

Sau đó chỉnh sửa hoặc tạo một tệp .gitattributes trong thư mục gốc của kho lưu trữ nơi bạn muốn sử dụng nó. Hoặc chỉ cần chỉnh sửa ~/.gitattributes:

plaintext

Sao chép mã

\*.strings diff=utf16

Điều này sẽ chuyển đổi tất cả các tệp có đuôi .strings trước khi Git so sánh.

Bạn có thể làm những điều tương tự cho các tệp khác, có thể được chuyển đổi thành văn bản.

Đối với các tệp plist nhị phân, bạn chỉnh sửa .gitconfig:

plaintext

Sao chép mã

[diff "plist"]

textconv = plutil -convert xml1 -o -

Và chỉnh sửa .gitattributes:

plaintext

Sao chép mã

\*.plist diff=plist

Điều này sẽ chuyển đổi tất cả các tệp có đuôi .plist trước khi Git so sánh.

## Chapter 7: Undoing Changes in Git

### Section 7.1: Quay lại một commit trước đó

Để quay lại một commit trước đó, trước tiên hãy tìm hash của commit bằng cách sử dụng git log.

Để tạm thời quay lại commit đó, tách rời HEAD của bạn bằng:

bash

Sao chép mã

git checkout 789abcd

Điều này đặt bạn vào commit 789abcd. Bạn có thể tạo các commit mới trên commit cũ này mà không ảnh hưởng đến nhánh mà HEAD của bạn đang ở. Bất kỳ thay đổi nào có thể được chuyển thành một nhánh đúng đắn bằng cách sử dụng branch hoặc checkout -b.

Để quay lại một commit trước đó trong khi giữ các thay đổi:

bash

Sao chép mã

git reset --soft 789abcd

Để quay lại commit cuối cùng:

bash

Sao chép mã

git reset --soft HEAD~

Để xóa vĩnh viễn bất kỳ thay đổi nào được thực hiện sau một commit cụ thể, sử dụng:

bash

Sao chép mã

git reset --hard 789abcd

Để xóa vĩnh viễn bất kỳ thay đổi nào được thực hiện sau commit cuối cùng:

bash

Sao chép mã

git reset --hard HEAD~

**Cẩn thận:** Trong khi bạn có thể khôi phục các commit bị loại bỏ bằng cách sử dụng reflog và reset, các thay đổi chưa commit không thể được khôi phục. Sử dụng git stash; git reset thay vì git reset --hard để an toàn.

### Section 7.2: Hoàn tác các thay đổi

Hoàn tác các thay đổi đối với một tệp hoặc thư mục trong bản sao làm việc:

bash

Sao chép mã

git checkout -- file.txt

Sử dụng trên tất cả các đường dẫn tệp, đệ quy từ thư mục hiện tại, nó sẽ hoàn tác tất cả các thay đổi trong bản sao làm việc:

bash

Sao chép mã

git checkout -- .

Để chỉ hoàn tác một phần của các thay đổi, sử dụng --patch. Bạn sẽ được hỏi, cho mỗi thay đổi, nếu nó nên được hoàn tác hay không:

bash

Sao chép mã

git checkout --patch -- dir

Để hoàn tác các thay đổi đã được thêm vào index:

bash

Sao chép mã

git reset --hard

Không có cờ --hard, lệnh này sẽ thực hiện reset mềm.

Với các commit cục bộ mà bạn chưa đẩy lên remote, bạn cũng có thể thực hiện reset mềm. Bạn có thể chỉnh sửa lại các tệp và sau đó commit lại:

bash

Sao chép mã

git reset HEAD~2

Ví dụ trên sẽ hoàn tác hai commit cuối cùng của bạn và trả lại các tệp vào bản sao làm việc của bạn. Bạn có thể thực hiện các thay đổi tiếp theo và tạo các commit mới.

**Cẩn thận:** Tất cả các thao tác này, ngoại trừ reset mềm, sẽ xóa vĩnh viễn các thay đổi của bạn. Để có lựa chọn an toàn hơn, sử dụng git stash -p hoặc git stash, sau đó hoàn tác với stash pop hoặc xóa mãi mãi với stash drop.

### Section 7.3: Sử dụng reflog

Nếu bạn gặp sự cố khi rebase, một tùy chọn để bắt đầu lại là quay lại commit trước khi rebase. Bạn có thể làm điều này bằng cách sử dụng reflog (lịch sử của mọi thứ bạn đã làm trong 90 ngày qua - điều này có thể được cấu hình):

bash

Sao chép mã

$ 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

...

Bạn có thể thấy commit trước khi rebase là HEAD@{3} (bạn cũng có thể checkout hash):

bash

Sao chép mã

git checkout HEAD@{3}

Bây giờ bạn có thể tạo một nhánh mới, xóa nhánh cũ hoặc thử rebase lại.

Bạn cũng có thể reset trực tiếp trở lại một điểm trong reflog của mình, nhưng chỉ làm điều này nếu bạn hoàn toàn chắc chắn đó là điều bạn muốn làm:

bash

Sao chép mã

git reset --hard HEAD@{3}

Điều này sẽ đặt cây git hiện tại của bạn khớp với cách nó đã tại điểm đó (xem phần Hoàn tác các thay đổi).

Điều này có thể được sử dụng nếu bạn tạm thời xem nhánh hoạt động như thế nào khi rebase trên một nhánh khác, nhưng bạn không muốn giữ kết quả.

### Section 7.4: Hoàn tác các lần merge

**Hoàn tác một merge chưa đẩy lên remote**

Nếu bạn chưa đẩy merge của mình lên kho lưu trữ remote thì bạn có thể theo cùng quy trình như trong việc hoàn tác commit mặc dù có một số khác biệt nhỏ.

Reset là lựa chọn đơn giản nhất vì nó sẽ hoàn tác cả commit merge và bất kỳ commit nào được thêm từ nhánh. Tuy nhiên, bạn sẽ cần biết SHA để reset trở lại, điều này có thể khó khăn vì log git của bạn sẽ hiển thị các commit từ cả hai nhánh. Nếu bạn reset đến commit sai (ví dụ, một commit trên nhánh khác) nó có thể phá hủy công việc đã commit.

bash

Sao chép mã

git reset --hard <last commit from the branch you are on>

Hoặc, giả sử merge là commit mới nhất của bạn:

bash

Sao chép mã

git reset HEAD~

Revert an toàn hơn, vì nó sẽ không phá hủy công việc đã commit, nhưng đòi hỏi nhiều công việc hơn vì bạn phải revert revert trước khi bạn có thể merge lại nhánh (xem phần tiếp theo).

**Hoàn tác một merge đã đẩy lên remote**

Giả sử bạn merge vào một tính năng mới (add-gremlins):

bash

Sao chép mã

git merge feature/add-gremlins

...

# Resolve any merge conflicts

git commit # commit the merge

...

git push

...

501b75d..17a51fd master -> master

Sau đó, bạn phát hiện ra rằng tính năng bạn vừa merge vào làm hỏng hệ thống cho các nhà phát triển khác, nó phải được hoàn tác ngay lập tức, và sửa tính năng sẽ mất quá nhiều thời gian nên bạn chỉ muốn hoàn tác merge:

bash

Sao chép mã

git revert -m 1 17a51fd

...

git push

...

17a51fd..e443799 master -> master

Lúc này, các gremlin đã được xóa khỏi hệ thống và các nhà phát triển của bạn đã ngừng la hét vào bạn. Tuy nhiên, chúng ta chưa hoàn tất. Khi bạn sửa vấn đề với tính năng add-gremlins, bạn sẽ cần hoàn tác revert này trước khi bạn có thể merge lại.

bash

Sao chép mã

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

Lúc này, tính năng của bạn đã được thêm thành công. Tuy nhiên, vì các lỗi loại này thường được giới thiệu bởi xung đột merge, một quy trình làm việc hơi khác có thể hữu ích hơn vì nó cho phép bạn sửa xung đột merge trên nhánh của bạn.

bash

Sao chép mã

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)

Sau khi bạn hoàn tất việc sửa lỗi trong nhánh tính năng và đã hoàn tác lần revert trước đó, bạn có thể thực hiện merge lại nhánh tính năng vào nhánh chính:

bash

Sao chép mã

git checkout master

...

# Không cần revert lại lần revert tại điểm này vì đã làm trước đó

git merge feature/add-gremlins

...

# Sửa bất kỳ xung đột merge nào được giới thiệu bởi việc sửa lỗi

git commit # commit lần merge

...

git push

### Section 7.5: Hoàn tác một số commit hiện có

Sử dụng git revert để hoàn tác các commit hiện có, đặc biệt khi các commit đó đã được đẩy lên kho lưu trữ remote. Lệnh này sẽ tạo ra một commit mới để đảo ngược hiệu ứng của một số commit trước đó, bạn có thể đẩy an toàn mà không cần viết lại lịch sử.

Đừng sử dụng git push --force trừ khi bạn muốn gây ra sự phản đối của tất cả người dùng khác của kho lưu trữ đó. Không bao giờ viết lại lịch sử công khai.

Nếu, ví dụ, bạn vừa đẩy một commit có chứa lỗi và bạn cần quay lại, hãy làm như sau:

bash

Sao chép mã

git revert HEAD~1

git push

Bây giờ bạn có thể tự do hoàn tác commit revert cục bộ, sửa mã của bạn, và đẩy mã tốt lên:

bash

Sao chép mã

git revert HEAD~1

work .. work .. work ..

git add -A .

git commit -m "Update error code"

git push

Nếu commit bạn muốn revert đã nằm xa hơn trong lịch sử, bạn chỉ cần truyền hash của commit. Git sẽ tạo ra một commit đảo ngược lại commit gốc của bạn, mà bạn có thể đẩy lên remote an toàn:

bash

Sao chép mã

git revert 912aaf0228338d0c8fb8cca0a064b0161a451fdc

git push

### Section 7.6: Hoàn tác / Làm lại một loạt các commit

Giả sử bạn muốn hoàn tác một tá commit và bạn chỉ muốn giữ lại một số trong đó.

Sử dụng git rebase -i <earlier SHA> để đưa rebase vào chế độ tương tác. Nó sẽ bắt đầu giống như rebase được thảo luận ở trên, nhưng trước khi phát lại bất kỳ commit nào, nó sẽ tạm dừng và cho phép bạn chỉnh sửa nhẹ từng commit khi nó được phát lại. rebase -i sẽ mở trong trình soạn thảo văn bản mặc định của bạn, với một danh sách các commit đang được áp dụng, như sau:

plaintext

Sao chép mã

pick e43a6f6 Commit message 1

pick c9f2e8b Commit message 2

pick f8b1f4c Commit message 3

* pick: Giữ lại commit đó.
* reword: Chỉnh sửa thông điệp commit nhưng giữ nguyên nội dung.
* edit: Tạm dừng tại commit đó để chỉnh sửa nó.
* squash: Kết hợp commit đó với commit trước đó.
* fixup: Giống như squash nhưng bỏ qua thông điệp commit của commit này.
* exec: Chạy một lệnh shell tại điểm đó.
* drop: Bỏ qua commit đó.

Sau khi bạn đã chỉnh sửa file, lưu và thoát. Git sẽ thực hiện rebase theo chỉ dẫn của bạn.

bash

Sao chép mã

git rebase -i HEAD~N

Nơi N là số commit gần đây mà bạn muốn rebase. Ví dụ, nếu bạn muốn chỉnh sửa 5 commit cuối cùng:

bash

Sao chép mã

git rebase -i HEAD~5

Nếu bạn gặp bất kỳ xung đột nào trong quá trình rebase, Git sẽ tạm dừng và cho phép bạn giải quyết xung đột. Sau khi giải quyết xung đột, sử dụng:

bash

Sao chép mã

git rebase --continue

Nếu bạn muốn hủy bỏ quá trình rebase bất cứ lúc nào:

bash

Sao chép mã

git rebase --abort

Sau khi hoàn thành, đẩy các thay đổi lên remote:

bash

Sao chép mã

git push --force-with-lease

Lưu ý rằng --force-with-lease là một cách an toàn hơn để đẩy các thay đổi buộc vì nó đảm bảo rằng không có ai khác đã đẩy các thay đổi trong khi bạn đang rebase.

## Chapter 8: Merging

Parameter Details

* -m: Thêm thông điệp vào commit merge.
* -v: Hiển thị đầu ra chi tiết.
* --abort: Thử khôi phục tất cả các tệp về trạng thái trước khi merge.
* --ff-only: Hủy ngay lập tức khi cần một commit merge.
* --no-ff: Bắt buộc tạo một commit merge, ngay cả khi không cần thiết.
* --no-commit: Giả vờ như merge thất bại để cho phép kiểm tra và điều chỉnh kết quả.
* --stat: Hiển thị diffstat sau khi hoàn thành merge.
* -n, --no-stat: Không hiển thị diffstat.
* --squash: Cho phép tạo một commit duy nhất trên nhánh hiện tại với các thay đổi đã merge.

### Section 8.1: Automatic Merging

Khi các commit trên hai nhánh không xung đột, Git có thể tự động merge chúng:

bash

Sao chép mã

git merge another\_branch

Đầu ra:

plaintext

Sao chép mã

Auto-merging file\_a

Merge made by the 'recursive' strategy.

file\_a | 2 +-

1 file changed, 1 insertion(+), 1 deletion(-)

### Section 8.2: Tìm tất cả các nhánh không có thay đổi được merge

Đôi khi bạn có thể có các nhánh nằm xung quanh đã được merge vào master. Lệnh sau tìm tất cả các nhánh không phải master mà không có commit độc nhất so với master. Điều này rất hữu ích để tìm các nhánh không được xóa sau khi PR được merge vào master.

bash

Sao chép mã

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

### Section 8.3: Hủy bỏ một merge

Sau khi bắt đầu merge, bạn có thể muốn dừng merge và khôi phục mọi thứ về trạng thái trước khi merge. Sử dụng --abort:

bash

Sao chép mã

git merge --abort

### Section 8.4: Merge với một commit

Hành vi mặc định là khi merge được giải quyết là fast-forward, chỉ cập nhật con trỏ nhánh mà không tạo một commit merge. Sử dụng --no-ff để giải quyết.

bash

Sao chép mã

git merge <branch\_name> --no-ff -m "<commit message>"

### Section 8.5: Giữ các thay đổi từ một phía của merge

Trong quá trình merge, bạn có thể truyền --ours hoặc --theirs cho git checkout để lấy tất cả các thay đổi cho một tệp từ một phía của merge.

bash

Sao chép mã

$ git checkout --ours -- file1.txt # Sử dụng phiên bản của chúng ta của file1, xóa tất cả các thay đổi của họ

$ git checkout --theirs -- file2.txt # Sử dụng phiên bản của họ của file2, xóa tất cả các thay đổi của chúng ta

### Section 8.6: Merge một nhánh vào nhánh khác

bash

Sao chép mã

git merge incomingBranch

Điều này merge nhánh incomingBranch vào nhánh bạn đang ở. Ví dụ, nếu bạn đang ở nhánh master, thì incomingBranch sẽ được merge vào master.

Merge có thể tạo ra xung đột trong một số trường hợp. Nếu điều này xảy ra, bạn sẽ thấy thông báo:

plaintext

Sao chép mã

Automatic merge failed; fix conflicts and then commit the result.

Bạn sẽ cần chỉnh sửa thủ công các tệp bị xung đột, hoặc để hoàn tác nỗ lực merge của bạn, chạy:

bash

Sao chép mã

git merge --abort

## Chapter 9: Submodules

### Section 9.1: Cloning a Git Repository Having Submodules

Khi bạn clone một kho lưu trữ sử dụng submodule, bạn sẽ cần khởi tạo và cập nhật chúng.

bash

Sao chép mã

$ git clone --recursive https://github.com/username/repo.git

Lệnh này sẽ clone các submodule được tham chiếu và đặt chúng vào các thư mục thích hợp (bao gồm cả các submodule bên trong submodule). Điều này tương đương với việc chạy git submodule update --init --recursive ngay sau khi clone hoàn thành.

### Section 9.2: Updating a Submodule

Một submodule tham chiếu một commit cụ thể trong một kho lưu trữ khác. Để kiểm tra trạng thái chính xác được tham chiếu cho tất cả các submodule, chạy:

bash

Sao chép mã

git submodule update --recursive

Đôi khi thay vì sử dụng trạng thái được tham chiếu, bạn muốn cập nhật checkout cục bộ của bạn lên trạng thái mới nhất của submodule trên remote. Để kiểm tra tất cả các submodule tới trạng thái mới nhất trên remote với một lệnh duy nhất, bạn có thể sử dụng:

bash

Sao chép mã

git submodule foreach git pull <remote> <branch>

Hoặc sử dụng các tham số mặc định của git pull:

bash

Sao chép mã

git submodule foreach git pull

Lưu ý rằng điều này chỉ cập nhật bản sao làm việc cục bộ của bạn. Chạy git status sẽ liệt kê thư mục submodule như "dirty" nếu nó thay đổi vì lệnh này. Để cập nhật kho lưu trữ của bạn để tham chiếu đến trạng thái mới, bạn phải commit các thay đổi:

bash

Sao chép mã

git add <submodule\_directory>

git commit

Có thể có một số thay đổi bạn có thể có xung đột merge nếu bạn sử dụng git pull vì vậy bạn có thể sử dụng git pull --rebase để tua lại các thay đổi của bạn lên đầu, hầu hết thời gian điều này sẽ giảm khả năng xảy ra xung đột. Nó cũng sẽ kéo tất cả các nhánh về cục bộ.

bash

Sao chép mã

git submodule foreach git pull --rebase

Để checkout trạng thái mới nhất của một submodule cụ thể, bạn có thể sử dụng:

bash

Sao chép mã

git submodule update --remote <submodule\_directory>

### Section 9.3: Adding a Submodule

Bạn có thể bao gồm một kho lưu trữ Git khác như một thư mục trong dự án của bạn, được theo dõi bởi Git:

bash

Sao chép mã

$ git submodule add https://github.com/jquery/jquery.git

Bạn nên thêm và commit tệp .gitmodules mới; tệp này sẽ thông báo cho Git biết submodule nào nên được clone khi git submodule update được chạy.

### Section 9.4: Setting a Submodule to Follow a Branch

Một submodule luôn được checkout tại một commit SHA1 cụ thể (được gọi là "gitlink", mục đặc biệt trong index của kho lưu trữ chính).

Nhưng bạn có thể yêu cầu cập nhật submodule đó đến commit mới nhất của một nhánh của kho lưu trữ remote của submodule. Thay vì vào từng submodule, thực hiện git checkout <branch> --track origin/<branch>, git pull, bạn có thể đơn giản làm (từ kho lưu trữ chính):

bash

Sao chép mã

git submodule update --remote --recursive

Vì SHA1 của submodule sẽ thay đổi, bạn vẫn cần phải theo dõi điều này bằng:

bash

Sao chép mã

git add .

git commit -m "update submodules"

Điều này giả định các submodule đã được:

* Được thêm với một nhánh để theo dõi:

bash

Sao chép mã

git submodule add -b <branch> -- /url/of/submodule/repo

Hoặc được cấu hình (cho một submodule hiện có) để theo dõi một nhánh:

bash

Sao chép mã

cd /path/to/parent/repo

git config -f .gitmodules submodule.<submodule>.branch <branch>

### Section 9.5: Moving a Submodule

**Phiên bản > 1.8**

Chạy:

bash

Sao chép mã

$ git mv /path/to/module new/path/to/module

**Phiên bản ≤ 1.8**

1. Chỉnh sửa .gitmodules và thay đổi đường dẫn của submodule thích hợp, và đặt nó vào index với git add .gitmodules.
2. Nếu cần thiết, tạo thư mục cha của vị trí mới của submodule (mkdir -p /path/to).
3. Di chuyển tất cả nội dung từ thư mục cũ sang thư mục mới (mv -vi /path/to/module new/path/to/submodule).
4. Đảm bảo Git theo dõi thư mục này (git add /path/to).
5. Xóa thư mục cũ với git rm --cached /path/to/module.
6. Di chuyển thư mục .git/modules//path/to/module cùng với tất cả nội dung của nó đến .git/modules//path/to/module.
7. Chỉnh sửa tệp .git/modules//path/to/config, đảm bảo mục worktree trỏ đến vị trí mới, ví dụ: worktree = ../../../../..//path/to/module. Thường có hai dấu chấm thêm so với các thư mục trong đường dẫn trực tiếp ở vị trí đó. Chỉnh sửa tệp /path/to/module/.git, đảm bảo rằng đường dẫn trong đó trỏ đến vị trí mới chính xác trong thư mục .git của dự án chính, ví dụ: gitdir: ../../../.git/modules//path/to/module.

Kết quả git status sẽ trông như sau:

plaintext

Sao chép mã

# 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

#

1. Cuối cùng, commit các thay đổi.

Ví dụ này từ Stack Overflow, bởi Axel Beckert.

### Section 9.6: Removing a Submodule

**Phiên bản > 1.8**

Bạn có thể xóa một submodule (ví dụ: the\_submodule) bằng cách gọi:

bash

Sao chép mã

$ git submodule deinit the\_submodule

$ git rm the\_submodule

git submodule deinit the\_submodule xóa mục của the\_submodule khỏi .git/config. Điều này loại trừ the\_submodule khỏi git submodule update, git submodule sync và git submodule foreach và xóa nội dung cục bộ của nó. Điều này cũng sẽ không được hiển thị như thay đổi trong kho lưu trữ chính của bạn. git submodule init và git submodule update sẽ khôi phục lại submodule, mà không có các thay đổi cần commit trong kho lưu trữ chính của bạn.

git rm the\_submodule sẽ xóa submodule khỏi cây làm việc. Các tệp sẽ bị xóa cùng với mục của submodule trong tệp .gitmodules. Nếu chỉ chạy git rm the\_submodule (không có lệnh git submodule deinit the\_submodule trước đó), mục của submodule trong tệp .git/config của bạn sẽ vẫn còn.

**Phiên bản < 1.8**

Lấy từ đây:

1. Xóa phần liên quan khỏi tệp .gitmodules.
2. Stage các thay đổi của .gitmodules:

bash

Sao chép mã

git add .gitmodules

1. Xóa phần liên quan khỏi .git/config.
2. Chạy git rm --cached path\_to\_submodule (không có dấu gạch chéo ở cuối).
3. Chạy rm -rf .git/modules/path\_to\_submodule.
4. Commit:

bash

Sao chép mã

git commit -m "Removed submodule <name>"

1. Xóa các tệp submodule hiện không được theo dõi:

bash

Sao chép mã

rm -rf path\_to\_submodule

## Chapter 10: Committing

Parameter Details

* --message, -m: Thông điệp để bao gồm trong commit. Tham số này bỏ qua hành vi bình thường của Git là mở một trình soạn thảo.
* --amend: Chỉ định rằng các thay đổi hiện tại đã được stage sẽ được thêm vào commit trước đó. Hãy cẩn thận, điều này có thể viết lại lịch sử!
* --no-edit: Sử dụng thông điệp commit đã chọn mà không khởi động trình soạn thảo. Ví dụ, git commit --amend --no-edit sửa đổi một commit mà không thay đổi thông điệp commit.
* --all, -a: Commit tất cả các thay đổi, bao gồm cả các thay đổi chưa được stage.
* --date: Đặt ngày thủ công sẽ được gắn với commit.
* --only: Chỉ commit các đường dẫn được chỉ định. Điều này sẽ không commit những gì bạn đã stage trừ khi được yêu cầu.
* --patch, -p: Sử dụng giao diện lựa chọn patch tương tác để chọn các thay đổi để commit.
* --help: Hiển thị trang hướng dẫn cho lệnh git commit.
* -S[keyid], -S --gpgsign[=keyid], -S --no-gpg-sign: Ký commit, GPG-ký commit, hủy bỏ cấu hình commit.gpgSign.
* -n, --no-verify: Bỏ qua các hook pre-commit và commit-msg. Xem thêm Hooks.

Commit với Git cung cấp sự chịu trách nhiệm bằng cách gán tác giả với các thay đổi mã. Git cung cấp nhiều tính năng cho tính cụ thể và bảo mật của commit. Chủ đề này giải thích và minh họa các thực hành và quy trình đúng đắn khi commit với Git.

### Section 10.1: Stage và commit các thay đổi

#### Các thao tác cơ bản

Sau khi thực hiện các thay đổi với mã nguồn của bạn, bạn nên stage các thay đổi đó với Git trước khi bạn có thể commit chúng. Ví dụ, nếu bạn thay đổi README.md và program.py:

bash

Sao chép mã

git add README.md program.py

Điều này thông báo cho git rằng bạn muốn thêm các tệp vào commit tiếp theo của bạn.

Sau đó, commit các thay đổi của bạn với:

bash

Sao chép mã

git commit

Lưu ý rằng lệnh này sẽ mở một trình soạn thảo văn bản, thường là vim. Nếu bạn không quen với vim, bạn có thể muốn biết rằng bạn có thể nhấn i để vào chế độ chèn, viết thông điệp commit của bạn, sau đó nhấn Esc và :wq để lưu và thoát. Để tránh mở trình soạn thảo văn bản, đơn giản bao gồm cờ -m với thông điệp của bạn:

bash

Sao chép mã

git commit -m "Commit message here"

Thông điệp commit thường tuân theo một số quy tắc định dạng cụ thể, xem phần Good commit messages để biết thêm thông tin.

#### Các phím tắt

Nếu bạn đã thay đổi nhiều tệp trong thư mục, thay vì liệt kê từng tệp một, bạn có thể sử dụng:

bash

Sao chép mã

git add --all # tương đương với "git add -a"

Hoặc để thêm tất cả các thay đổi, không bao gồm các tệp đã bị xóa, từ thư mục cấp cao nhất và các thư mục con:

bash

Sao chép mã

git add .

Hoặc để chỉ thêm các tệp hiện đang được theo dõi ("update"):

bash

Sao chép mã

git add -u

Nếu muốn, hãy xem lại các thay đổi đã được stage:

bash

Sao chép mã

git status # hiển thị danh sách các tệp đã thay đổi

git diff --cached # hiển thị các thay đổi đã được stage bên trong các tệp đã được stage

Cuối cùng, commit các thay đổi:

bash

Sao chép mã

git commit -m "Commit message here"

Ngoài ra, nếu bạn chỉ chỉnh sửa các tệp hiện có hoặc xóa các tệp, và không tạo ra bất kỳ tệp mới nào, bạn có thể kết hợp các hành động git add và git commit trong một lệnh duy nhất:

bash

Sao chép mã

git commit -am "Commit message here"

Lưu ý rằng lệnh này sẽ stage tất cả các tệp đã chỉnh sửa theo cách tương tự như git add --all.

#### Dữ liệu nhạy cảm

Bạn không bao giờ nên commit bất kỳ dữ liệu nhạy cảm nào, chẳng hạn như mật khẩu hoặc khóa riêng tư. Nếu điều này xảy ra và các thay đổi đã được đẩy lên máy chủ trung tâm, hãy coi bất kỳ dữ liệu nhạy cảm nào là đã bị xâm phạm. Nếu không, có thể xóa dữ liệu như vậy sau đó. Một giải pháp nhanh chóng và dễ dàng là sử dụng "BFG Repo-Cleaner": [BFG Repo-Cleaner](https://rtyley.github.io/bfg-repo-cleaner/).

Lệnh bfg --replace-text passwords.txt my-repo.git sẽ đọc các mật khẩu từ tệp passwords.txt và thay thế chúng bằng \*\*\*REMOVED\*\*\*. Hoạt động này xem xét tất cả các commit trước đó của toàn bộ kho lưu trữ.

### Section 10.2: Good Commit Messages

Điều quan trọng là để ai đó duyệt qua log git có thể dễ dàng hiểu được mỗi commit là gì.

Thông điệp commit tốt thường bao gồm một số nhiệm vụ hoặc một vấn đề trong trình theo dõi và một mô tả ngắn gọn về những gì đã được thực hiện và lý do tại sao, và đôi khi cũng làm thế nào nó đã được thực hiện.

Thông điệp tốt có thể trông như sau:

plaintext

Sao chép mã

TASK-123: Implement login through OAuth

TASK-124: Add auto minification of JS/CSS files

TASK-125: Fix minifier error when name > 200 chars

Trong khi các thông điệp sau sẽ không hữu ích lắm:

plaintext

Sao chép mã

fix // Đã sửa cái gì?

just a bit of a change // Đã thay đổi gì?

TASK-371 // Không có mô tả, người đọc sẽ cần xem trình theo dõi để biết giải thích

Implemented IFoo in IBar // Tại sao cần thiết?

Một cách để kiểm tra xem một thông điệp commit có được viết đúng ngữ điệu hay không là thay thế chỗ trống bằng thông điệp và xem liệu nó có hợp lý không:

plaintext

Sao chép mã

Nếu tôi thêm commit này, tôi sẽ \_\_\_ vào kho lưu trữ của mình.

#### Bảy quy tắc của một thông điệp commit tuyệt vời

1. Tách dòng chủ đề khỏi phần thân bằng một dòng trống.
2. Giới hạn dòng chủ đề trong 50 ký tự.
3. Viết hoa dòng chủ đề.
4. Không kết thúc dòng chủ đề bằng dấu chấm.
5. Sử dụng ngữ điệu mệnh lệnh trong dòng chủ đề.
6. Bọc mỗi dòng của phần thân thủ công ở 72 ký tự.
7. Sử dụng phần thân để giải thích cái gì và tại sao thay vì làm thế nào.

Bảy quy tắc này từ blog của Chris Beam.

### Section 10.3: Amending a Commit

Nếu commit mới nhất của bạn chưa được công khai (chưa đẩy lên một kho lưu trữ upstream) thì bạn có thể sửa đổi commit của mình.

bash

Sao chép mã

git commit --amend

Điều này sẽ đặt các thay đổi hiện đã được stage vào commit trước đó.

**Lưu ý:** Điều này cũng có thể được sử dụng để chỉnh sửa một thông điệp commit sai. Nó sẽ mở trình soạn thảo mặc định (thường là vi / vim / emacs) và cho phép bạn thay đổi thông điệp trước đó.

Để chỉ định thông điệp commit trực tiếp trên dòng lệnh:

bash

Sao chép mã

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

Hoặc để sử dụng thông điệp commit trước đó mà không thay đổi nó:

bash

Sao chép mã

git commit --amend --no-edit

Sửa đổi commit sẽ cập nhật ngày commit nhưng để nguyên ngày của tác giả. Bạn có thể yêu cầu git làm mới thông tin này.

bash

Sao chép mã

git commit --amend --reset-author

Bạn cũng có thể thay đổi tác giả của commit với:

bash

Sao chép mã

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

**Lưu ý:** Hãy nhớ rằng sửa đổi commit mới nhất sẽ thay thế nó hoàn toàn và commit trước đó sẽ bị xóa khỏi lịch sử của nhánh. Điều này nên được lưu ý khi làm việc với các kho lưu trữ công khai và trên các nhánh có các cộng tác viên khác.

Điều này có nghĩa là nếu commit trước đó đã được đẩy lên, sau khi sửa đổi nó bạn sẽ phải đẩy bằng --force.

### Section 10.4: Committing Without Opening an Editor

Git thường sẽ mở một trình soạn thảo (như vim hoặc emacs) khi bạn chạy git commit. Truyền tham số -m để chỉ định thông điệp từ dòng lệnh:

bash

Sao chép mã

git commit -m "Commit message here"

Thông điệp commit của bạn có thể bao gồm nhiều dòng:

bash

Sao chép mã

git commit -m "Commit subject line message here

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

Hoặc bạn có thể truyền nhiều tham số -m:

bash

Sao chép mã

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

### Section 10.5: Committing Changes Directly

Thông thường, bạn phải sử dụng git add hoặc git rm để thêm các thay đổi vào index trước khi bạn có thể git commit chúng. Sử dụng tham số -a hoặc --all để tự động thêm mọi thay đổi (đối với các tệp đã được theo dõi) vào index, bao gồm cả các tệp đã bị xóa:

bash

Sao chép mã

git commit -a

Nếu bạn cũng muốn thêm thông điệp commit, bạn có thể làm:

bash

Sao chép mã

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

Bạn cũng có thể kết hợp hai cờ:

bash

Sao chép mã

git commit -am "your commit message goes here"

Bạn không nhất thiết phải commit tất cả các tệp cùng một lúc. Bỏ qua cờ -a hoặc --all và chỉ định tệp bạn muốn commit trực tiếp:

bash

Sao chép mã

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

Để commit trực tiếp nhiều tệp cụ thể, bạn có thể chỉ định một hoặc nhiều tệp, thư mục và mẫu:

bash

Sao chép mã

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

### Section 10.6: Selecting Which Lines Should Be Staged for Committing

Giả sử bạn có nhiều thay đổi trong một hoặc nhiều tệp nhưng từ mỗi tệp bạn chỉ muốn commit một số thay đổi, bạn có thể chọn các thay đổi mong muốn bằng cách sử dụng:

bash

Sao chép mã

git add -p

Hoặc:

bash

Sao chép mã

git add -p [file]

Mỗi thay đổi của bạn sẽ được hiển thị riêng lẻ, và đối với mỗi thay đổi bạn sẽ được yêu cầu chọn một trong các tùy chọn sau:

* y: Có, thêm hunk này.
* n: Không, không thêm hunk này.
* d: Không, không thêm hunk này, hoặc bất kỳ hunk nào khác còn lại cho tệp này. Hữu ích nếu bạn đã thêm những gì bạn muốn và muốn bỏ qua phần còn lại.
* s: Chia nhỏ hunk thành các hunk nhỏ hơn, nếu có thể.
* e: Chỉnh sửa hunk thủ công. Đây có lẽ là tùy chọn mạnh mẽ nhất. Nó sẽ mở hunk trong trình soạn thảo văn bản và bạn có thể chỉnh sửa nó theo nhu cầu.

Điều này sẽ stage các phần của các tệp bạn chọn. Sau đó, bạn có thể commit tất cả các thay đổi đã được stage như sau:

bash

Sao chép mã

git commit -m 'Commit Message'

Các thay đổi không được stage hoặc commit sẽ vẫn xuất hiện trong các tệp làm việc của bạn và có thể được commit sau nếu cần. Hoặc nếu các thay đổi còn lại không mong muốn, chúng có thể bị loại bỏ với:

bash

Sao chép mã

git reset --hard

Ngoài việc chia một thay đổi lớn thành các commit nhỏ hơn, cách tiếp cận này cũng hữu ích cho việc xem lại những gì bạn sắp commit. Bằng cách xác nhận từng thay đổi riêng lẻ, bạn có cơ hội kiểm tra những gì bạn đã viết và có thể tránh việc vô tình stage mã không mong muốn như các câu lệnh println hoặc logging.

### Section 10.7: Creating an Empty Commit

Nói chung, các commit trống (hoặc các commit có trạng thái giống hệt với commit cha) là một lỗi. Tuy nhiên, khi kiểm tra các hook build, hệ thống CI và các hệ thống khác kích hoạt khi có commit, thì việc có thể dễ dàng tạo các commit mà không cần chỉnh sửa/tạo tệp dummy là tiện lợi.

Tham số --allow-empty sẽ bỏ qua kiểm tra.

bash

Sao chép mã

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

### Section 10.8: Committing on Behalf of Someone Else

Nếu người khác đã viết mã bạn đang commit, bạn có thể ghi nhận họ bằng tham số --author:

bash

Sao chép mã

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

Bạn cũng có thể cung cấp một mẫu, mà Git sẽ sử dụng để tìm kiếm các tác giả trước đó:

bash

Sao chép mã

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

Trong trường hợp này, thông tin tác giả từ commit gần đây nhất có chứa "John" sẽ được sử dụng.

Trên GitHub, các commit được thực hiện theo bất kỳ cách nào trên sẽ hiển thị hình thu nhỏ của tác giả lớn, với hình nhỏ của người commit nhỏ hơn và ở phía trước.

### Section 10.9: GPG Signing Commits

1. Xác định ID khóa của bạn:

bash

Sao chép mã

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

Kết quả:

plaintext

Sao chép mã

/Users/davidcondrey/.gnupg/secring.gpg

--------------------------------------

sec 2048R/YOUR-16-DIGIT-KEY-ID YYYY-MM-DD [expires: YYYY-MM-DD]

ID của bạn là mã alphanumeric 16 chữ số theo sau dấu gạch chéo đầu tiên.

1. Định nghĩa ID khóa của bạn trong cấu hình git của bạn:

bash

Sao chép mã

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

1. Từ phiên bản 1.7.9 trở đi, lệnh git commit chấp nhận tùy chọn -S để đính kèm một chữ ký vào các commit của bạn. Sử dụng tùy chọn này sẽ yêu cầu mật khẩu GPG của bạn và sẽ thêm chữ ký của bạn vào log commit.

bash

Sao chép mã

git commit -S -m "Your commit message"

### Section 10.10: Committing Changes in Specific Files

Bạn có thể commit các thay đổi đã thực hiện đối với các tệp cụ thể và bỏ qua việc stage chúng bằng cách sử dụng git add:

bash

Sao chép mã

git commit file1.c file2.h

Hoặc bạn có thể stage các tệp trước:

bash

Sao chép mã

git add file1.c file2.h

Và commit chúng sau:

bash

Sao chép mã

git commit

### Section 10.11: Committing at a Specific Date

bash

Sao chép mã

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

Tham số --date đặt ngày tác giả. Ngày này sẽ xuất hiện trong đầu ra tiêu chuẩn của git log, ví dụ.

Để buộc ngày commit cũng:

bash

Sao chép mã

GIT\_COMMITTER\_DATE=2016-07-01 git commit -m 'Fix UI bug' --date 2016-07-01

Tham số date chấp nhận các định dạng linh hoạt như được hỗ trợ bởi GNU date, ví dụ:

bash

Sao chép mã

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'

Khi ngày không chỉ định thời gian, thời gian hiện tại sẽ được sử dụng và chỉ có ngày sẽ bị ghi đè.

### Section 10.12: Amending the Time of a Commit

Bạn có thể sửa đổi thời gian của một commit bằng cách sử dụng:

bash

Sao chép mã

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

Hoặc thậm chí:

bash

Sao chép mã

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

### Section 10.13: Amending the Author of a Commit

Nếu bạn commit với tác giả sai, bạn có thể thay đổi và sau đó sửa đổi:

bash

Sao chép mã

git config user.name "Full Name"

git config user.email "email@example.com"

git commit --amend --reset-author

## Chapter 11: Aliases

### Section 11.1: Simple Aliases

Có hai cách để tạo alias trong Git:

1. Sử dụng tệp ~/.gitconfig:

ini

Sao chép mã

[alias]

ci = commit

st = status

co = checkout

1. Sử dụng dòng lệnh:

bash

Sao chép mã

git config --global alias.ci "commit"

git config --global alias.st "status"

git config --global alias.co "checkout"

Sau khi alias được tạo, bạn có thể gõ:

* git ci thay cho git commit,
* git st thay cho git status,
* git co thay cho git checkout.

Cũng giống như các lệnh git thông thường, các alias có thể được sử dụng cùng với các đối số. Ví dụ:

bash

Sao chép mã

git ci -m "Commit message..."

git co -b feature-42

### Section 11.2: Liệt kê / Tìm kiếm các Alias hiện có

Bạn có thể liệt kê các alias git hiện có bằng cách sử dụng --get-regexp:

bash

Sao chép mã

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

#### Tìm kiếm các Alias

Để tìm kiếm các alias, thêm đoạn sau vào tệp .gitconfig của bạn dưới [alias]:

ini

Sao chép mã

[alias]

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

Sau đó, bạn có thể sử dụng:

* git aliases: Hiển thị TẤT CẢ các alias.
* git aliases commit: Chỉ hiển thị các alias chứa từ "commit".

### Section 11.3: Advanced Aliases

Git cho phép bạn sử dụng các lệnh không phải git và cú pháp đầy đủ của shell trong các alias nếu bạn thêm tiền tố !.

Trong tệp ~/.gitconfig của bạn:

ini

Sao chép mã

[alias]

temp = !git add -A && git commit -m "Temp"

Việc cú pháp đầy đủ của shell có sẵn trong các alias có tiền tố này cũng có nghĩa là bạn có thể sử dụng các hàm shell để tạo các alias phức tạp hơn, chẳng hạn như các alias sử dụng đối số dòng lệnh:

ini

Sao chép mã

[alias]

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

Alias trên định nghĩa hàm f, sau đó chạy nó với bất kỳ đối số nào bạn truyền vào alias. Vì vậy, chạy git ignore .tmp/ sẽ thêm .tmp/ vào tệp .gitignore của bạn.

Trên thực tế, mẫu này hữu ích đến mức Git định nghĩa các biến $1, $2, v.v. cho bạn, vì vậy bạn không cần phải định nghĩa một hàm đặc biệt cho nó. (Nhưng hãy nhớ rằng Git cũng sẽ thêm các đối số vào cuối, ngay cả khi bạn truy cập chúng thông qua các biến này, vì vậy bạn có thể muốn thêm một lệnh giả ở cuối.)

Lưu ý rằng các alias có tiền tố ! theo cách này được chạy từ thư mục gốc của checkout git của bạn, ngay cả khi thư mục hiện tại của bạn ở sâu hơn trong cây. Đây có thể là một cách hữu ích để chạy một lệnh từ gốc mà không cần phải cd đến đó rõ ràng.

ini

Sao chép mã

[alias]

ignore = "! echo $1 >> .gitignore"

### Section 11.4: Tạm thời bỏ qua các tệp đã được theo dõi

Để tạm thời đánh dấu một tệp là bị bỏ qua (truyền tệp làm tham số cho alias) - gõ:

ini

Sao chép mã

[alias]

unwatch = update-index --assume-unchanged

Để bắt đầu theo dõi tệp lại - gõ:

ini

Sao chép mã

[alias]

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

Để liệt kê tất cả các tệp đã bị tạm thời bỏ qua - gõ:

ini

Sao chép mã

[alias]

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

Để xóa danh sách các tệp không được theo dõi - gõ:

ini

Sao chép mã

[alias]

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

Ví dụ sử dụng các alias:

bash

Sao chép mã

git unwatch my\_file.txt

git watch my\_file.txt

git unwatched

git watchall

### Section 11.5: Hiển thị Log Đẹp với Đồ Thị Nhánh

Trong tệp ~/.gitconfig của bạn:

ini

Sao chép mã

[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'

Giải thích các tùy chọn và placeholder được sử dụng trong định dạng --pretty:

* --graph: Vẽ cây commit.
* --date-order: Sử dụng thứ tự timestamp commit khi có thể.
* --first-parent: Chỉ theo dõi nhánh cha đầu tiên tại nút merge.
* --branches: Hiển thị tất cả các nhánh cục bộ (theo mặc định, chỉ hiển thị nhánh hiện tại).
* --all: Hiển thị tất cả các nhánh cục bộ và remote.
* %h: Giá trị hash cho commit (rút gọn).
* %ad: Dấu thời gian (tác giả).
* %an: Tên người dùng tác giả.
* %C(auto): Sử dụng màu được định nghĩa trong phần [color].
* %Creset: Đặt lại màu.
* %d: --decorate (tên nhánh và tag).
* %s: Thông điệp commit.
* %an: Tên tác giả (có thể là %cn cho tên người commit).

### Section 11.6: Xem Tệp Bị Bỏ Qua bởi Cấu Hình .gitignore của Bạn

Trong tệp ~/.gitconfig của bạn:

ini

Sao chép mã

[alias]

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

&& git ls-files --others -i --exclude-standard

Hiển thị một dòng cho mỗi tệp, vì vậy bạn có thể grep (chỉ các thư mục):

bash

Sao chép mã

$ git ignored | grep '/$'

.yardoc/

doc/

Hoặc đếm:

bash

Sao chép mã

$ git ignored | wc -l

199811 # oops, thư mục chính của tôi đang trở nên đông đúc

### Section 11.7: Cập Nhật Mã trong Khi Giữ Lịch Sử Tuyến Tính

Đôi khi bạn cần giữ một lịch sử tuyến tính (không phân nhánh) của các commit mã của bạn. Nếu bạn đang làm việc trên một nhánh trong một thời gian, điều này có thể khó khăn nếu bạn phải thực hiện git pull thường xuyên vì điều đó sẽ ghi lại một merge với upstream.

Trong tệp ~/.gitconfig của bạn:

ini

Sao chép mã

[alias]

up = pull --rebase

Điều này sẽ cập nhật với nguồn upstream của bạn, sau đó áp dụng lại bất kỳ công việc nào bạn chưa đẩy lên trên cùng của những gì bạn đã kéo xuống.

Để sử dụng:

bash

Sao chép mã

git up

### Section 11.8: Bỏ Stage Các Tệp Đã Stage

Thông thường, để loại bỏ các tệp đã stage để được commit, bạn sử dụng git reset. reset có nhiều chức năng phụ thuộc vào các đối số được cung cấp cho nó. Để hoàn toàn bỏ stage tất cả các tệp đã stage, chúng ta có thể sử dụng alias git để tạo một alias mới sử dụng reset nhưng bây giờ chúng ta không cần phải nhớ cung cấp các đối số chính xác cho reset.

bash

Sao chép mã

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

Bây giờ, bất kỳ lúc nào bạn muốn bỏ stage các tệp đã stage, gõ git unstage và bạn đã sẵn sàng.

## Chapter 12: Rebasing

Parameter Details

* --continue: Tiếp tục quá trình rebase sau khi đã giải quyết xung đột merge.
* --abort: Hủy bỏ thao tác rebase và đặt lại HEAD về nhánh gốc. Nếu nhánh được cung cấp khi bắt đầu rebase, thì HEAD sẽ được đặt lại về nhánh đó. Nếu không, HEAD sẽ được đặt lại về nơi nó đã ở khi bắt đầu thao tác rebase.
* --keep-empty: Giữ các commit không thay đổi gì từ cha mẹ của chúng trong kết quả.
* --skip: Tiếp tục quá trình rebase bằng cách bỏ qua patch hiện tại.
* -m, --merge: Sử dụng chiến lược merge để rebase. Khi chiến lược merge mặc định (recursive) được sử dụng, điều này cho phép rebase nhận biết về các rename ở phía upstream. Lưu ý rằng một rebase merge hoạt động bằng cách phát lại mỗi commit từ nhánh làm việc lên trên nhánh upstream. Vì lý do này, khi xảy ra xung đột merge, phía được báo cáo là ours là chuỗi commit đã rebase cho đến nay, bắt đầu với upstream, và theirs là nhánh làm việc. Nói cách khác, các phía bị đảo ngược.
* --stat: Hiển thị diffstat của những gì đã thay đổi upstream kể từ lần rebase cuối cùng. Diffstat cũng được kiểm soát bởi tùy chọn cấu hình rebase.stat.
* -x, --exec command: Thực hiện rebase tương tác, dừng giữa mỗi commit và thực thi lệnh.

### Section 12.1: Local Branch Rebasing

Rebasing áp dụng lại một loạt các commit lên trên một commit khác. Để rebase một nhánh, checkout nhánh đó và sau đó rebase nó lên trên một nhánh khác.

bash

Sao chép mã

git checkout topic

git rebase master # rebase nhánh hiện tại lên nhánh master

Điều này sẽ làm cho:

plaintext

Sao chép mã

A---B---C topic

/

D---E---F---G master

Thành:

plaintext

Sao chép mã

A'--B'--C' topic

/

D---E---F---G master

Các thao tác này có thể được kết hợp vào một lệnh duy nhất để checkout nhánh và ngay lập tức rebase nó:

bash

Sao chép mã

git rebase master topic # rebase nhánh topic lên nhánh master

Quan trọng: Sau khi rebase, các commit đã áp dụng sẽ có một hash khác. Bạn không nên rebase các commit mà bạn đã đẩy lên một host từ xa. Hậu quả có thể là không thể git push nhánh local đã rebase của bạn lên một host từ xa, khiến bạn chỉ có tùy chọn git push --force.

### Section 12.2: Rebase: ours and theirs, local and remote

Một rebase thay đổi ý nghĩa của "ours" và "theirs":

bash

Sao chép mã

git checkout topic

git rebase master # rebase nhánh topic lên trên nhánh master

Bất cứ điều gì HEAD đang trỏ tới là "ours".

Điều đầu tiên một rebase làm là đặt lại HEAD về nhánh master; trước khi chọn các commit từ nhánh topic cũ để tạo nhánh mới (mỗi commit trong nhánh topic trước đây sẽ được viết lại và sẽ được nhận diện bằng một hash khác).

Liên quan đến các thuật ngữ được sử dụng bởi các công cụ merge (không bị nhầm lẫn với ref local hoặc ref remote)

* local là master ("ours"),
* remote là topic ("theirs").

Điều này có nghĩa là một công cụ merge/diff sẽ trình bày nhánh upstream là local (master: nhánh trên đó bạn đang rebase), và nhánh làm việc là remote (topic: nhánh đang được rebase).

plaintext

Sao chép mã

+-----------------------------------------+

| LOCAL:master | BASE | REMOTE:topic |

+-----------------------------------------+

| MERGED |

+-----------------------------------------+

#### Inversion illustrated

**On a merge:**

plaintext

Sao chép mã

c--c--x--x--x(\*) <- nhánh hiện tại topic ('\*'=HEAD)

\

\

\--y--y--y <- nhánh khác để merge

Chúng ta không thay đổi nhánh hiện tại topic, vì vậy những gì chúng ta có vẫn là những gì chúng ta đang làm việc trên (và chúng ta merge từ một nhánh khác).

plaintext

Sao chép mã

c--c--x--x--x---------o(\*) MERGE, vẫn trên nhánh topic

\ ^ /

\ ours /

\ /

--y--y--y--/

^

theirs

**On a rebase:**

Nhưng trên một rebase chúng ta đổi bên vì điều đầu tiên mà một rebase làm là checkout nhánh upstream để phát lại các commit hiện tại lên trên nó!

plaintext

Sao chép mã

c--c--x--x--x(\*) <- nhánh hiện tại topic ('\*'=HEAD)

\

\

\--y--y--y <- nhánh upstream

Một git rebase upstream sẽ đầu tiên đặt HEAD về nhánh upstream, do đó sự thay đổi của 'ours' và 'theirs' so với nhánh làm việc "hiện tại".

plaintext

Sao chép mã

c--c--x--x--x <- nhánh "hiện tại" trước đây, mới là "theirs"

\

\

\--y--y--y(\*) <- đặt HEAD vào commit này, để phát lại các x lên đó

^ điều này sẽ là "ours" mới

|

upstream

Rebase sau đó sẽ phát lại các commit "their" trên nhánh "our" mới:

plaintext

Sao chép mã

c--c..x..x..x <- các commit "theirs" cũ, hiện là "ghosts", có sẵn thông qua "reflogs"

\

\

\--y--y--y--x'--x'--x'(\*) <- topic khi tất cả các x được phát lại,

^ trỏ nhánh topic đến commit này

|

nhánh upstream

### Section 12.3: Interactive Rebase

Ví dụ này nhằm mô tả cách sử dụng git rebase ở chế độ tương tác. Dự kiến rằng bạn đã có kiến thức cơ bản về git rebase là gì và nó làm gì.

Rebase tương tác được khởi động bằng cách sử dụng lệnh sau:

bash

Sao chép mã

git rebase -i

Tham số -i đề cập đến chế độ tương tác. Sử dụng rebase tương tác, bạn có thể thay đổi thông điệp commit, cũng như sắp xếp lại, chia nhỏ, và/hoặc gộp (combine) các commit.

Giả sử bạn muốn sắp xếp lại ba commit cuối cùng của mình. Để làm điều này bạn có thể chạy:

bash

Sao chép mã

git rebase -i HEAD~3

Sau khi thực hiện lệnh trên, một tệp sẽ được mở trong trình soạn thảo văn bản của bạn nơi bạn sẽ có thể chọn cách các commit của bạn sẽ được rebase. Cho mục đích của ví dụ này, chỉ cần thay đổi thứ tự của các commit, lưu tệp và đóng trình soạn thảo. Điều này sẽ khởi động một rebase với thứ tự mà bạn đã áp dụng. Nếu bạn kiểm tra git log, bạn sẽ thấy các commit của bạn trong thứ tự mới mà bạn đã chỉ định.

#### Thay đổi thông điệp commit

Bây giờ, bạn đã quyết định rằng một trong các thông điệp commit là mơ hồ và bạn muốn nó chi tiết hơn. Hãy xem xét ba commit cuối cùng bằng cách sử dụng cùng lệnh.

bash

Sao chép mã

git rebase -i HEAD~3

Thay vì sắp xếp lại thứ tự các commit sẽ được rebase, lần này chúng ta sẽ thay đổi pick, mặc định, thành reword trên commit mà bạn muốn thay đổi thông điệp.

Khi bạn đóng trình soạn thảo, rebase sẽ khởi động và nó sẽ dừng lại tại thông điệp commit cụ thể mà bạn muốn thay đổi. Điều này sẽ cho phép bạn thay đổi thông điệp commit thành bất kỳ điều gì bạn muốn. Sau khi bạn đã thay đổi thông điệp, chỉ cần đóng trình soạn thảo để tiếp tục.

#### Thay đổi nội dung của một commit

Ngoài việc thay đổi thông điệp commit, bạn cũng có thể điều chỉnh các thay đổi đã được thực hiện bởi commit. Để làm điều này, chỉ cần thay đổi pick thành edit cho một commit. Git sẽ dừng lại khi nó đến commit đó và cung cấp các thay đổi ban đầu của commit trong khu vực staging. Bây giờ bạn có thể điều chỉnh các thay đổi đó bằng cách bỏ stage chúng hoặc thêm các thay đổi mới.

Ngay khi khu vực staging chứa tất cả các thay đổi bạn muốn trong commit đó, commit các thay đổi. Thông điệp commit cũ sẽ được hiển thị và có thể được điều chỉnh để phản ánh commit mới.

#### Chia nhỏ một commit thành nhiều commit

Giả sử bạn đã thực hiện một commit nhưng sau đó quyết định rằng commit này có thể được chia thành hai hoặc nhiều commit. Sử dụng cùng lệnh như trước, thay thế pick bằng edit và nhấn enter.

Bây giờ, git sẽ dừng lại tại commit bạn đã đánh dấu để chỉnh sửa và đặt tất cả nội dung của nó vào khu vực staging. Từ đó, bạn có thể chạy git reset HEAD^ để đặt commit vào thư mục làm việc của bạn. Sau đó, bạn có thể thêm và commit các tệp theo một trình tự khác - cuối cùng chia một commit thành n commit.

#### Squashing nhiều commit thành một commit

Giả sử bạn đã thực hiện một số công việc và có nhiều commit mà bạn nghĩ rằng có thể là một commit duy nhất. Để làm điều này, bạn có thể thực hiện git rebase -i HEAD~3, thay thế 3 bằng số lượng commit phù hợp.

Lần này thay đổi pick thành squash thay thế. Trong quá trình rebase, commit mà bạn đã chỉ định để squash sẽ được squash lên trên commit trước đó; biến chúng thành một commit duy nhất.

### Section 12.4: Rebase xuống commit đầu tiên

Từ Git 1.7.12, có thể rebase xuống commit gốc. Commit gốc là commit đầu tiên từng được thực hiện trong một kho lưu trữ và thường không thể chỉnh sửa. Sử dụng lệnh sau:

bash

Sao chép mã

git rebase -i --root

### Section 12.5: Cấu hình autostash

Autostash là một tùy chọn cấu hình rất hữu ích khi sử dụng rebase cho các thay đổi local. Thường thì, bạn có thể cần mang vào các commit từ nhánh upstream, nhưng chưa sẵn sàng để commit ngay.

Tuy nhiên, Git không cho phép bắt đầu một rebase nếu thư mục làm việc không sạch. Autostash sẽ giải quyết vấn đề này:

bash

Sao chép mã

git config --global rebase.autostash # cấu hình một lần

git rebase @{u} # ví dụ rebase trên nhánh upstream

Autostash sẽ được áp dụng bất cứ khi nào rebase kết thúc. Không quan trọng liệu rebase có kết thúc thành công hay bị hủy bỏ. Dù bằng cách nào, autostash sẽ được áp dụng. Nếu rebase thành công, và commit cơ sở do đó đã thay đổi, thì có thể sẽ có xung đột giữa autostash và các commit mới. Trong trường hợp này, bạn sẽ phải giải quyết các xung đột trước khi commit. Điều này không khác gì nếu bạn đã thủ công stash và sau đó áp dụng, vì vậy không có nhược điểm gì khi làm tự động.

### Section 12.6: Kiểm tra tất cả các commit trong quá trình rebase

Trước khi thực hiện một pull request, việc đảm bảo rằng biên dịch thành công và các bài kiểm tra đều qua đối với mỗi commit trong nhánh là rất hữu ích. Chúng ta có thể làm điều đó tự động bằng cách sử dụng tham số -x.

Ví dụ:

bash

Sao chép mã

git rebase -i -x make

Sẽ thực hiện rebase tương tác và dừng lại sau mỗi commit để thực hiện lệnh make. Trong trường hợp make thất bại, git sẽ dừng lại để cho bạn cơ hội sửa các vấn đề và chỉnh sửa commit trước khi tiếp tục chọn commit tiếp theo.

### Section 12.7: Rebasing Before a Code Review

#### Summary

Mục tiêu của việc này là sắp xếp lại tất cả các commit của bạn thành các commit có ý nghĩa hơn để dễ dàng kiểm tra mã (code review). Nếu có quá nhiều lớp thay đổi trên quá nhiều tệp cùng một lúc, việc kiểm tra mã sẽ trở nên khó khăn hơn. Nếu bạn có thể sắp xếp lại các commit được tạo theo thứ tự thời gian thành các commit theo chủ đề, thì quy trình kiểm tra mã sẽ dễ dàng hơn (và có thể giảm thiểu lỗi).

Ví dụ đơn giản quá mức này không phải là chiến lược duy nhất để sử dụng git để làm các kiểm tra mã tốt hơn. Đây là cách tôi thực hiện, và nó là một cái gì đó để khuyến khích người khác xem xét cách làm cho việc kiểm tra mã và lịch sử git dễ dàng/tốt hơn. Nó cũng minh họa sức mạnh của rebase một cách tổng quát.

Ví dụ này giả định bạn đã biết về rebase tương tác.

Giả sử:

* Bạn đang làm việc trên một nhánh tính năng từ master.
* Tính năng của bạn có ba lớp chính: front-end, back-end, DB.
* Bạn đã thực hiện nhiều commit khi làm việc trên nhánh tính năng. Mỗi commit chạm đến nhiều lớp cùng một lúc.
* Bạn muốn (cuối cùng) chỉ có ba commit trong nhánh của mình:
  + Một commit chứa tất cả các thay đổi front-end.
  + Một commit chứa tất cả các thay đổi back-end.
  + Một commit chứa tất cả các thay đổi DB.

#### Strategy

* Chúng ta sẽ thay đổi các commit theo thứ tự thời gian thành các commit "theo chủ đề".
* Đầu tiên, chia tất cả các commit thành nhiều commit nhỏ hơn, mỗi commit chỉ chứa một chủ đề tại một thời điểm (trong ví dụ của chúng ta, các chủ đề là các thay đổi front-end, back-end, DB).
* Sau đó, sắp xếp lại các commit theo chủ đề và 'squash' chúng thành các commit theo chủ đề duy nhất.

#### Example

bash

Sao chép mã

$ 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

Điều này sẽ hiển thị trong trình soạn thảo văn bản:

plaintext

Sao chép mã

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

Thay đổi thành:

plaintext

Sao chép mã

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

Sau đó git sẽ áp dụng từng commit một. Sau mỗi commit, nó sẽ hiển thị một lời nhắc, và sau đó bạn có thể làm như sau:

plaintext

Sao chép mã

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

bash

Sao chép mã

$ 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^ # Điều này sẽ 'uncommits' tất cả các thay đổi trong commit này.

$ git status -s

M db.sql

M page.html

$ git add db.sql # bây giờ chúng ta sẽ tạo các commit nhỏ theo chủ đề

$ git commit -m "first draft: db.sql"

$ git add page.html

$ git commit -m "first draft: page.html"

$ git rebase --continue

Sau đó, bạn sẽ lặp lại các bước này cho mỗi commit. Cuối cùng, bạn sẽ có:

bash

Sao chép mã

$ 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

Bây giờ chúng ta sẽ chạy rebase một lần nữa để sắp xếp lại và squash:

bash

Sao chép mã

$ git rebase -i master

Điều này sẽ hiển thị trong trình soạn thảo văn bản:

plaintext

Sao chép mã

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

Thay đổi thành:

plaintext

Sao chép mã

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

LƯU Ý: Đảm bảo rằng bạn bảo git rebase áp dụng/squash các commit nhỏ theo chủ đề theo thứ tự chúng đã được commit theo thứ tự thời gian. Nếu không, bạn có thể gặp phải các xung đột merge không cần thiết và giả mạo để giải quyết.

Khi rebase tương tác này kết thúc, bạn sẽ có:

bash

Sao chép mã

$ git log --oneline master..

74bdd5f adding todos: GUI layer

e8d8f7e adding todos: business logic layer

121c578 adding todos: DB layer

#### Recap

Bây giờ bạn đã rebase các commit theo thứ tự thời gian của mình thành các commit theo chủ đề. Trong thực tế, bạn có thể không cần làm điều này mỗi lần, nhưng khi bạn muốn hoặc cần làm điều này, bây giờ bạn đã có thể. Ngoài ra, hy vọng bạn đã học được nhiều hơn về git rebase.

### Section 12.8: Aborting an Interactive Rebase

Bạn đã bắt đầu một rebase tương tác. Trong trình soạn thảo nơi bạn chọn các commit, bạn quyết định rằng có gì đó sai (ví dụ, một commit bị thiếu hoặc bạn đã chọn nhầm điểm đến rebase), và bạn muốn hủy bỏ rebase.

Để làm điều này, chỉ cần xóa tất cả các commit và hành động (tức là tất cả các dòng không bắt đầu bằng ký hiệu #) và rebase sẽ bị hủy bỏ!

Văn bản trợ giúp trong trình soạn thảo thực sự cung cấp gợi ý này:

plaintext

Sao chép mã

# 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

### Section 12.9: Setup git-pull for Automatically Performing a Rebase Instead of a Merge

Nếu nhóm của bạn đang theo quy trình làm việc dựa trên rebase, thì việc thiết lập git để mỗi nhánh mới được tạo sẽ thực hiện thao tác rebase, thay vì thao tác merge, trong quá trình git pull có thể rất hữu ích.

Để thiết lập mỗi nhánh mới tự động rebase, thêm đoạn sau vào tệp .gitconfig hoặc .git/config của bạn:

ini

Sao chép mã

[branch]

autosetuprebase = always

Lệnh dòng lệnh:

bash

Sao chép mã

git config [--global] branch.autosetuprebase always

Ngoài ra, bạn có thể thiết lập lệnh git pull luôn hoạt động như thể tùy chọn --rebase đã được truyền vào:

ini

Sao chép mã

[pull]

rebase = true

Lệnh dòng lệnh:

bash

Sao chép mã

git config [--global] pull.rebase true

### Section 12.10: Pushing After a Rebase

Đôi khi bạn cần viết lại lịch sử với một rebase, nhưng git push lại phản đối vì bạn đã viết lại lịch sử.

Điều này có thể được giải quyết bằng git push --force, nhưng hãy cân nhắc sử dụng git push --force-with-lease, chỉ ra rằng bạn muốn push thất bại nếu nhánh theo dõi từ xa cục bộ khác với nhánh trên remote, ví dụ: có ai đó đã push lên remote sau lần fetch cuối cùng. Điều này tránh việc vô tình ghi đè lên lần push gần đây của người khác.

Lưu ý: git push --force - và thậm chí --force-with-lease - có thể là một lệnh nguy hiểm vì nó viết lại lịch sử của nhánh. Nếu một người khác đã pull nhánh trước khi push bắt buộc, lệnh git pull hoặc git fetch của họ sẽ gặp lỗi vì lịch sử cục bộ và lịch sử remote bị phân kỳ. Điều này có thể gây ra lỗi không mong muốn cho người đó. Với việc xem xét đủ kỹ các reflog, công việc của người dùng khác có thể được khôi phục, nhưng điều này có thể dẫn đến mất nhiều thời gian. Nếu bạn phải thực hiện một push bắt buộc lên một nhánh có người đóng góp khác, hãy cố gắng phối hợp với họ để họ không phải xử lý các lỗi.

bash

Sao chép mã

git push --force-with-lease

## Chapter 13: Configuration

Parameter Details

* --system: Chỉnh sửa tệp cấu hình toàn hệ thống, được sử dụng cho mọi người dùng (trên Linux, tệp này nằm tại $(prefix)/etc/gitconfig)
* --global: Chỉnh sửa tệp cấu hình toàn cục, được sử dụng cho mọi kho lưu trữ bạn làm việc (trên Linux, tệp này nằm tại ~/.gitconfig)
* --local: Chỉnh sửa tệp cấu hình cụ thể cho kho lưu trữ, nằm tại .git/config trong kho lưu trữ của bạn; đây là thiết lập mặc định

### Section 13.1: Setting Which Editor to Use

Có nhiều cách để thiết lập trình soạn thảo nào sẽ được sử dụng cho việc commit, rebase, v.v.

#### Thay đổi thiết lập cấu hình core.editor:

bash

Sao chép mã

$ git config --global core.editor nano

#### Thiết lập biến môi trường GIT\_EDITOR:

* Cho một lệnh:

bash

Sao chép mã

$ GIT\_EDITOR=nano git commit

* Cho tất cả các lệnh chạy trong một terminal. Lưu ý: Điều này chỉ áp dụng cho đến khi bạn đóng terminal.

bash

Sao chép mã

$ export GIT\_EDITOR=nano

#### Để thay đổi trình soạn thảo cho tất cả các chương trình terminal, không chỉ Git, thiết lập biến môi trường VISUAL hoặc EDITOR:

(Tham khảo VISUAL vs EDITOR)

bash

Sao chép mã

$ export EDITOR=nano

Lưu ý: Như trên, điều này chỉ áp dụng cho terminal hiện tại; shell của bạn thường sẽ có một tệp cấu hình để cho phép bạn thiết lập nó vĩnh viễn. (Trên bash, ví dụ, thêm dòng trên vào tệp ~/.bashrc hoặc ~/.bash\_profile của bạn.)

Một số trình soạn thảo văn bản (chủ yếu là GUI) sẽ chỉ chạy một phiên bản cùng một lúc và thường thoát nếu bạn đã mở một phiên bản của chúng. Nếu đây là trường hợp cho trình soạn thảo văn bản của bạn, Git sẽ in thông báo Aborting commit due to empty commit message mà không cho phép bạn chỉnh sửa thông điệp commit trước. Nếu điều này xảy ra với bạn, hãy tham khảo tài liệu của trình soạn thảo văn bản của bạn để xem liệu nó có cờ --wait (hoặc tương tự) để làm cho nó tạm dừng cho đến khi tài liệu được đóng lại hay không.

### Section 13.2: Auto Correct Typos

bash

Sao chép mã

git config --global help.autocorrect 17

Điều này kích hoạt tự động sửa lỗi trong git và sẽ tha thứ cho bạn những sai lầm nhỏ (ví dụ: git stats thay vì git status). Tham số bạn cung cấp cho help.autocorrect xác định thời gian hệ thống nên chờ, tính bằng phần mười của giây, trước khi tự động áp dụng lệnh đã được sửa lỗi. Trong lệnh trên, 17 có nghĩa là git sẽ chờ 1,7 giây trước khi áp dụng lệnh đã được sửa lỗi.

Tuy nhiên, những sai lầm lớn hơn sẽ được coi là thiếu lệnh, vì vậy gõ một cái gì đó như git testingit sẽ dẫn đến testingit is not a git command.

### Section 13.3: List and Edit the Current Configuration

Git config cho phép bạn tùy chỉnh cách git hoạt động. Nó thường được sử dụng để thiết lập tên và email của bạn hoặc trình soạn thảo yêu thích của bạn hoặc cách các merge nên được thực hiện.

#### Để xem cấu hình hiện tại:

bash

Sao chép mã

$ git config --list

...

core.editor=vim

credential.helper=osxkeychain

...

#### Để chỉnh sửa cấu hình:

bash

Sao chép mã

$ git config <key> <value>

$ git config core.ignorecase true

Nếu bạn muốn thay đổi có hiệu lực cho tất cả các kho lưu trữ của bạn, sử dụng --global:

bash

Sao chép mã

$ git config --global user.name "Your Name"

$ git config --global user.email "Your Email"

$ git config --global core.editor vi

Bạn có thể liệt kê lại để xem các thay đổi của mình.

### Section 13.4: Username and Email Address

Ngay sau khi bạn cài đặt Git, điều đầu tiên bạn nên làm là thiết lập tên người dùng và địa chỉ email của mình. Từ shell, gõ:

bash

Sao chép mã

git config --global user.name "Mr. Bean"

git config --global user.email mrbean@example.com

* git config là lệnh để lấy hoặc thiết lập các tùy chọn.
* --global có nghĩa là tệp cấu hình cụ thể cho tài khoản người dùng của bạn sẽ được chỉnh sửa.
* user.name và user.email là các khóa cho các biến cấu hình; user là phần của tệp cấu hình. name và email là tên của các biến.
* "Mr. Bean" và mrbean@example.com là các giá trị bạn đang lưu trữ trong hai biến này. Lưu ý dấu ngoặc kép quanh "Mr. Bean", điều này là cần thiết vì giá trị bạn đang lưu trữ chứa khoảng trắng.

### Section 13.5: Multiple Usernames and Email Addresses

Từ Git 2.13 trở đi, nhiều tên người dùng và địa chỉ email có thể được cấu hình bằng cách sử dụng bộ lọc thư mục.

#### Ví dụ cho Windows:

.gitconfig

* Chỉnh sửa: git config --global -e
* Thêm:

ini

Sao chép mã

[includeIf "gitdir:D:/work/"]

path = .gitconfig-work.config

[includeIf "gitdir:D:/opensource/"]

path = .gitconfig-opensource.config

#### Lưu ý:

* Thứ tự phụ thuộc, cái nào khớp cuối cùng sẽ được áp dụng ("wins").
* / ở cuối là cần thiết - ví dụ: gitdir:D:/work sẽ không hoạt động.
* Tiền tố gitdir: là bắt buộc.

.gitconfig-work.config

* Tệp trong cùng thư mục với .gitconfig:

ini

Sao chép mã

[user]

name = Money

email = work@somewhere.com

.gitconfig-opensource.config

* Tệp trong cùng thư mục với .gitconfig:

ini

Sao chép mã

[user]

name = Nice

email = cool@opensource.stuff

#### Ví dụ cho Linux:

ini

Sao chép mã

[includeIf "gitdir:~/work/"]

path = .gitconfig-work

[includeIf "gitdir:~/opensource/"]

path = .gitconfig-opensource

Nội dung tệp và lưu ý như phần Windows.

### Section 13.6: Multiple Git Configurations

Bạn có tối đa 5 nguồn cho cấu hình git:

#### 6 tệp:

1. %ALLUSERSPROFILE%\Git\Config (chỉ Windows)
2. (system) <git>/etc/gitconfig với <git> là đường dẫn cài đặt git.
   * (Trên Windows, là <git>\mingw64\etc\gitconfig)
3. (system) $XDG\_CONFIG\_HOME/git/config (chỉ Linux/Mac)
4. (global) ~/.gitconfig (Windows: %USERPROFILE%\.gitconfig)
5. (local) .git/config (trong một git repo $GIT\_DIR)

#### Một tệp chuyên dụng (với git config -f), sử dụng ví dụ để chỉnh sửa cấu hình của submodule:

bash

Sao chép mã

git config -f .gitmodules ...

#### Lệnh dòng lệnh với git -c:

bash

Sao chép mã

git -c core.autocrlf=false fetch

* Thứ tự là quan trọng: bất kỳ cấu hình nào được thiết lập trong một nguồn có thể bị ghi đè bởi một nguồn được liệt kê dưới nó.
* git config --system/global/local là lệnh để liệt kê 3 trong số các nguồn đó, nhưng chỉ git config -l mới liệt kê tất cả các cấu hình đã được giải quyết.

"Giải quyết" nghĩa là nó chỉ liệt kê giá trị cấu hình cuối cùng đã bị ghi đè.

Từ git 2.8, nếu bạn muốn xem cấu hình nào đến từ tệp nào, bạn gõ:

bash

Sao chép mã

git config --list --show-origin

### Section 13.7: Configuring Line Endings

#### Mô tả

Khi làm việc với một nhóm sử dụng các hệ điều hành (OS) khác nhau trên dự án, đôi khi bạn có thể gặp rắc rối khi xử lý line endings.

#### Microsoft Windows

Khi làm việc trên hệ điều hành Microsoft Windows, line endings thường có dạng - carriage return + line feed (CR+LF). Mở một tệp đã được chỉnh sửa bằng máy Unix như Linux hoặc OSX có thể gây rắc rối, làm cho văn bản dường như không có line endings. Điều này là do hệ thống Unix áp dụng các line-endings khác nhau chỉ có line feeds (LF).

Để khắc phục điều này, bạn có thể chạy lệnh sau:

bash

Sao chép mã

git config --global core.autocrlf=true

Khi checkout, lệnh này sẽ đảm bảo line-endings được cấu hình phù hợp với hệ điều hành Microsoft Windows (LF -> CR+LF).

#### Unix Based (Linux/OSX)

Tương tự, có thể có vấn đề khi người dùng trên hệ điều hành Unix cố gắng đọc các tệp đã được chỉnh sửa trên hệ điều hành Microsoft Windows. Để ngăn ngừa bất kỳ vấn đề nào không mong muốn, chạy lệnh:

bash

Sao chép mã

git config --global core.autocrlf=input

Khi commit, lệnh này sẽ thay đổi line-endings từ CR+LF -> LF.

### Section 13.8: Configuration for One Command Only

Bạn có thể sử dụng -c <name>=<value> để thêm cấu hình chỉ cho một lệnh duy nhất.

#### Để commit dưới tên người dùng khác mà không cần thay đổi cài đặt trong .gitconfig:

bash

Sao chép mã

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

Lưu ý: cho ví dụ đó, bạn không cần chỉ định cả user.name và user.email, git sẽ hoàn thành thông tin thiếu từ các commit trước.

### Section 13.9: Setup a Proxy

Nếu bạn đang ở sau một proxy, bạn cần nói cho git về nó:

bash

Sao chép mã

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

Nếu bạn không còn ở sau proxy nữa:

bash

Sao chép mã

git config --global --unset http.proxy

## Chapter 14: Branching

Parameter Details

* -d, --delete: Xóa một nhánh. Nhánh này phải được hợp nhất hoàn toàn vào nhánh upstream của nó, hoặc vào HEAD nếu không có upstream nào được đặt với --track hoặc --set-upstream.
* -D: Phím tắt cho --delete --force.
* -m, --move: Di chuyển/đổi tên một nhánh và tương ứng với reflog.
* -M: Phím tắt cho --move --force.
* -r, --remotes: Liệt kê hoặc xóa (nếu sử dụng với -d) các nhánh theo dõi từ xa.
* -a, --all: Liệt kê cả các nhánh theo dõi từ xa và nhánh local.
* --list: Kích hoạt chế độ danh sách. git branch <pattern> sẽ cố gắng tạo một nhánh, sử dụng git branch --list <pattern> để liệt kê các nhánh phù hợp.
* --set-upstream: Nếu nhánh được chỉ định chưa tồn tại hoặc nếu --force được cung cấp, hoạt động chính xác như --track. Nếu không, thiết lập cấu hình như --track khi tạo nhánh, ngoại trừ việc nơi mà nhánh trỏ đến không thay đổi.

### Section 14.1: Creating and Checking Out New Branches

#### Tạo và Kiểm Tra Nhánh Mới

Để tạo một nhánh mới, trong khi vẫn ở trên nhánh hiện tại, sử dụng:

bash

Sao chép mã

git branch <name>

Thông thường, tên nhánh không được chứa khoảng trắng và tuân theo các quy định khác được liệt kê ở đây. Để chuyển sang một nhánh hiện có:

bash

Sao chép mã

git checkout <name>

Để tạo một nhánh mới và chuyển sang nhánh đó:

bash

Sao chép mã

git checkout -b <name>

Để tạo một nhánh tại một điểm khác ngoài commit cuối cùng của nhánh hiện tại (còn được gọi là HEAD), sử dụng một trong các lệnh sau:

bash

Sao chép mã

git branch <name> [<start-point>]

git checkout -b <name> [<start-point>]

<start-point> có thể là bất kỳ revision nào mà git biết đến (ví dụ: tên nhánh khác, commit SHA, hoặc tham chiếu biểu tượng như HEAD hoặc tên tag):

bash

Sao chép mã

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

Để tạo một nhánh từ một nhánh từ xa (mặc định <remote\_name> là origin):

bash

Sao chép mã

git branch <name> <remote\_name>/<branch\_name>

git checkout -b <name> <remote\_name>/<branch\_name>

Nếu một tên nhánh được tìm thấy chỉ trên một remote, bạn có thể đơn giản sử dụng:

bash

Sao chép mã

git checkout -b <branch\_name>

Điều này tương đương với:

bash

Sao chép mã

git checkout -b <branch\_name> <remote\_name>/<branch\_name>

#### Di Chuyển Nhiều Commit Gần Đây Sang Nhánh Mới

Đôi khi bạn có thể cần di chuyển nhiều commit gần đây sang một nhánh mới. Điều này có thể được thực hiện bằng cách tạo nhánh và "quay lại", như sau:

bash

Sao chép mã

git branch <new\_name>

git reset --hard HEAD~2 # Quay lại 2 commit, bạn sẽ mất công việc chưa được commit.

git checkout <new\_name>

#### Minh Họa Kỹ Thuật Này

Trạng thái ban đầu:

plaintext

Sao chép mã

A-B-C-D-E (HEAD)

↑

master

Sau git branch <new\_name>:

plaintext

Sao chép mã

newBranch

↓

A-B-C-D-E (HEAD)

↑

master

Sau git reset --hard HEAD~2:

plaintext

Sao chép mã

newBranch

↓

A-B-C (HEAD)

↑

master

### Section 14.2: Listing Branches

Git cung cấp nhiều lệnh để liệt kê các nhánh. Tất cả các lệnh này sử dụng chức năng của git branch, sẽ cung cấp danh sách các nhánh nhất định, tùy thuộc vào các tùy chọn được đưa vào dòng lệnh. Git sẽ nếu có thể, chỉ ra nhánh hiện đang được chọn với một dấu sao bên cạnh.

#### Mục Tiêu - Lệnh

* Liệt kê các nhánh local:

bash

Sao chép mã

git branch

* Liệt kê các nhánh local chi tiết:

bash

Sao chép mã

git branch -v

* Liệt kê các nhánh remote và local:

bash

Sao chép mã

git branch -a

hoặc:

bash

Sao chép mã

git branch --all

* Liệt kê các nhánh remote và local (chi tiết):

bash

Sao chép mã

git branch -av

* Liệt kê các nhánh remote:

bash

Sao chép mã

git branch -r

* Liệt kê các nhánh remote với commit mới nhất:

bash

Sao chép mã

git branch -rv

* Liệt kê các nhánh đã hợp nhất:

bash

Sao chép mã

git branch --merged

* Liệt kê các nhánh chưa hợp nhất:

bash

Sao chép mã

git branch --no-merged

* Liệt kê các nhánh chứa commit:

bash

Sao chép mã

git branch --contains [<commit>]

#### Lưu ý:

* Thêm một v vào -v, ví dụ: $ git branch -avv hoặc $ git branch -vv sẽ in tên của nhánh upstream nữa.
* Các nhánh được hiển thị bằng màu đỏ là các nhánh remote.

### Section 14.3: Delete a Remote Branch

Để xóa một nhánh trên kho lưu trữ remote origin, bạn có thể sử dụng cho phiên bản Git 1.5.0 và mới hơn:

bash

Sao chép mã

git push origin :<branchName>

Và từ phiên bản Git 1.7.0, bạn có thể xóa một nhánh remote bằng cách sử dụng:

bash

Sao chép mã

git push origin --delete <branchName>

#### Để xóa một nhánh theo dõi từ xa local:

bash

Sao chép mã

git branch --delete --remotes <remote>/<branch>

git branch -dr <remote>/<branch> # Ngắn gọn hơn

git fetch <remote> --prune # Xóa nhiều nhánh theo dõi lỗi thời

git fetch <remote> -p # Ngắn gọn hơn

Để xóa một nhánh local. Lưu ý rằng điều này sẽ không xóa nhánh nếu nó có bất kỳ thay đổi chưa hợp nhất nào:

bash

Sao chép mã

git branch -d <branchName>

Để xóa một nhánh, ngay cả khi nó có thay đổi chưa hợp nhất:

bash

Sao chép mã

git branch -D <branchName>

### Section 14.4: Quick Switch to the Previous Branch

Bạn có thể nhanh chóng chuyển sang nhánh trước đó bằng cách sử dụng:

bash

Sao chép mã

git checkout -

### Section 14.5: Check Out a New Branch Tracking a Remote Branch

Có ba cách để tạo một nhánh mới feature theo dõi nhánh remote origin/feature:

bash

Sao chép mã

git checkout --track -b feature origin/feature

git checkout -t origin/feature

git checkout feature # giả sử rằng không có nhánh local feature và chỉ có một remote với nhánh feature.

Để đặt upstream để theo dõi nhánh remote - gõ:

bash

Sao chép mã

git branch --set-upstream-to=<remote>/<branch> <branch>

git branch -u <remote>/<branch> <branch>

Trong đó:

* <remote> có thể là: origin, develop hoặc do người dùng tạo.
* <branch> là nhánh của người dùng để theo dõi trên remote.

Để xác minh nhánh remote nào mà các nhánh local của bạn đang theo dõi:

bash

Sao chép mã

git branch -vv

### Section 14.6: Delete a Branch Locally

Xóa nhánh tên dev nếu các thay đổi của nó đã được hợp nhất với một nhánh khác và sẽ không bị mất. Nếu nhánh dev chứa các thay đổi chưa được hợp nhất mà sẽ bị mất, git branch -d sẽ thất bại:

bash

Sao chép mã

$ 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'.

Theo thông báo cảnh báo, bạn có thể buộc xóa nhánh (và mất bất kỳ thay đổi chưa hợp nhất nào trong nhánh đó) bằng cách sử dụng cờ -D:

bash

Sao chép mã

$ git branch -D dev

### Section 14.7: Create an Orphan Branch (i.e., Branch with No Parent Commit)

Tạo một nhánh mới mà không có commit cha:

bash

Sao chép mã

git checkout --orphan new-orphan-branch

Commit đầu tiên được thực hiện trên nhánh mới này sẽ không có cha mẹ và nó sẽ là gốc của một lịch sử mới hoàn toàn tách biệt với tất cả các nhánh và commit khác.

### Section 14.8: Rename a Branch

#### Đổi tên nhánh mà bạn đã checkout:

bash

Sao chép mã

git branch -m new\_branch\_name

#### Đổi tên một nhánh khác:

bash

Sao chép mã

git branch -m branch\_you\_want\_to\_rename new\_branch\_name

### Section 14.9: Searching in Branches

#### Liệt kê các nhánh local chứa một commit hoặc tag cụ thể:

bash

Sao chép mã

git branch --contains <commit>

#### Liệt kê các nhánh local và remote chứa một commit hoặc tag cụ thể:

bash

Sao chép mã

git branch -a --contains <commit>

### Section 14.10: Push Branch to Remote

#### Để đẩy các commit đã thực hiện trên nhánh local của bạn lên một kho lưu trữ remote, sử dụng lệnh git push. Lệnh này nhận hai đối số:

* Tên remote, ví dụ: origin
* Tên nhánh, ví dụ: master

Ví dụ:

bash

Sao chép mã

git push <REMOTENAME> <BRANCHNAME>

Ví dụ thông thường, bạn thường chạy git push origin master để đẩy các thay đổi local của bạn lên kho lưu trữ trực tuyến của bạn.

#### Sử dụng -u (viết tắt của --set-upstream) sẽ thiết lập thông tin theo dõi trong quá trình push:

bash

Sao chép mã

git push -u <REMOTENAME> <BRANCHNAME>

Mặc định, git đẩy nhánh local lên một nhánh remote có cùng tên. Ví dụ, nếu bạn có một nhánh local tên là new-feature, nếu bạn push nhánh local này, nó sẽ tạo ra một nhánh remote cũng tên là new-feature. Nếu bạn muốn sử dụng một tên khác cho nhánh remote, thêm tên remote sau tên nhánh local, ngăn cách bằng dấu ::

bash

Sao chép mã

git push <REMOTENAME> <LOCALBRANCHNAME>:<REMOTEBRANCHNAME>

### Section 14.11: Move Current Branch HEAD to an Arbitrary Commit

Một nhánh chỉ là một con trỏ tới một commit, vì vậy bạn có thể di chuyển nó tự do. Để làm cho nhánh chỉ tới commit aabbcc, thực hiện lệnh:

bash

Sao chép mã

git reset --hard aabbcc

Lưu ý rằng điều này sẽ ghi đè commit hiện tại của nhánh của bạn, và do đó, toàn bộ lịch sử của nó. Bạn có thể mất một số công việc bằng cách thực hiện lệnh này. Nếu điều đó xảy ra, bạn có thể sử dụng reflog để khôi phục các commit đã mất. Có thể khuyên rằng bạn nên thực hiện lệnh này trên một nhánh mới thay vì nhánh hiện tại của bạn.

Tuy nhiên, lệnh này có thể đặc biệt hữu ích khi thực hiện rebase hoặc làm các sửa đổi lịch sử lớn khác.

## Chapter 15: Rev-List

### Parameter Details

* --oneline: Hiển thị các commit dưới dạng một dòng duy nhất với tiêu đề của chúng.

### Section 15.1: List Commits in master but not in origin/master

Sử dụng lệnh git rev-list để liệt kê các commit trong một nhánh nhưng không có trong nhánh khác. Đây là một công cụ tuyệt vời khi bạn đang cố gắng xác định xem mã đã được hợp nhất vào một nhánh hay chưa.

bash

Sao chép mã

git rev-list --oneline master ^origin/master

Lệnh này sẽ liệt kê các commit có trong master nhưng không có trong origin/master. Sử dụng tùy chọn --oneline sẽ hiển thị tiêu đề của mỗi commit.

Toán tử ^ loại trừ các commit trong nhánh được chỉ định khỏi danh sách.

Bạn có thể truyền nhiều hơn hai nhánh nếu muốn. Ví dụ:

bash

Sao chép mã

git rev-list foo bar ^baz

Lệnh này sẽ liệt kê các commit có trong foo và bar, nhưng không có trong baz.

### Tổng kết

Sử dụng git rev-list với các tùy chọn thích hợp giúp bạn dễ dàng kiểm tra sự khác biệt giữa các nhánh và xác định xem các commit đã được hợp nhất hay chưa. Tùy chọn --oneline giúp hiển thị thông tin một cách ngắn gọn và dễ đọc.

## Chapter 16: Squashing

### Section 16.1: Squash Recent Commits Without Rebasing

Nếu bạn muốn gộp x commit trước đó vào một commit duy nhất, bạn có thể sử dụng các lệnh sau:

bash

Sao chép mã

git reset --soft HEAD~x

git commit

Thay x bằng số lượng commit trước đó bạn muốn bao gồm trong commit gộp. Lưu ý rằng điều này sẽ tạo một commit mới, bỏ qua thông tin về các commit trước đó bao gồm tác giả, thông điệp và ngày tháng. Bạn có thể muốn sao chép và dán thông điệp commit hiện có trước.

### Section 16.2: Squashing Commit During Merge

Bạn có thể sử dụng git merge --squash để gộp các thay đổi được giới thiệu bởi một nhánh vào một commit duy nhất. Không có commit thực sự nào được tạo ra.

bash

Sao chép mã

git merge --squash <branch>

git commit

Điều này tương đối tương đương với việc sử dụng git reset, nhưng thuận tiện hơn khi các thay đổi được kết hợp có một tên biểu tượng. So sánh:

bash

Sao chép mã

git checkout <branch>

git reset --soft $(git merge-base master <branch>)

git commit

### Section 16.3: Squashing Commits During a Rebase

Các commit có thể được gộp lại trong quá trình git rebase. Được khuyến nghị rằng bạn nên hiểu về rebase trước khi cố gắng gộp các commit theo cách này.

1. Xác định commit mà bạn muốn rebase từ đó, và ghi lại hash commit của nó.
2. Chạy git rebase -i [commit hash]. Ngoài ra, bạn có thể gõ HEAD~4 thay vì hash commit để xem commit mới nhất và 4 commit trước đó.
3. Trong trình soạn thảo mở ra khi chạy lệnh này, xác định commit nào bạn muốn gộp. Thay pick ở đầu các dòng đó bằng squash để gộp chúng vào commit trước.
4. Sau khi chọn commit nào bạn muốn gộp, bạn sẽ được yêu cầu viết một thông điệp commit.

#### Logging Commits to Determine Where to Rebase

bash

Sao chép mã

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

bash

Sao chép mã

git rebase -i 36d15de

Tại thời điểm này, trình soạn thảo của bạn sẽ mở ra và bạn có thể mô tả những gì bạn muốn làm với các commit. Git cung cấp hướng dẫn trong các bình luận. Nếu bạn để nguyên như vậy, thì không có gì xảy ra vì mọi commit sẽ được giữ lại và thứ tự của chúng sẽ giống như trước khi rebase. Trong ví dụ này, chúng ta áp dụng các lệnh sau:

plaintext

Sao chép mã

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 log after writing commit message

bash

Sao chép mã

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

### Section 16.4: Autosquashing and Fixups

Khi commit các thay đổi, bạn có thể chỉ định rằng commit sẽ được gộp vào commit khác trong tương lai và điều này có thể được thực hiện như sau:

bash

Sao chép mã

git commit --squash=[commit hash of commit to which this commit will be squashed to]

Bạn cũng có thể sử dụng --fixup=[commit hash] thay thế để fixup. Cũng có thể sử dụng các từ từ thông điệp commit thay vì hash commit, như sau:

bash

Sao chép mã

git commit --squash :/things

Trong đó commit gần đây nhất có từ 'things' sẽ được sử dụng. Thông điệp commit của những commit này sẽ bắt đầu với 'fixup!' hoặc 'squash!' tiếp theo là phần còn lại của thông điệp commit mà những commit này sẽ được gộp vào.

#### Khi thực hiện rebase với --autosquash, các commit fixup/squash sẽ được tự động gộp lại. Để tránh phải gõ --autosquash mỗi lần rebase, bạn có thể bật tùy chọn này mặc định:

bash

Sao chép mã

git config --global rebase.autosquash true

### Section 16.5: Autosquash: Committing Code You Want to Squash During a Rebase

Giả sử lịch sử sau đây, bạn thực hiện một thay đổi mà bạn muốn gộp vào commit bbb2222 A second commit:

bash

Sao chép mã

$ git log --oneline --decorate

ccc3333 (HEAD -> master) A third commit

bbb2222 A second commit

aaa1111 A first commit

9999999 Initial commit

Sau khi bạn đã thực hiện các thay đổi của mình, bạn có thể thêm chúng vào chỉ mục như bình thường, sau đó commit chúng sử dụng đối số --fixup với tham chiếu đến commit bạn muốn gộp vào:

bash

Sao chép mã

$ git add .

$ git commit --fixup bbb2222

[my-feature-branch ddd4444] fixup! A second commit

Điều này sẽ tạo một commit mới với thông điệp commit mà Git có thể nhận ra trong quá trình rebase tương tác:

bash

Sao chép mã

$ 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

Tiếp theo, thực hiện rebase tương tác với đối số --autosquash:

bash

Sao chép mã

$ git rebase --autosquash --interactive HEAD~4

Git sẽ đề xuất bạn gộp commit bạn đã thực hiện với --fixup vào đúng vị trí:

plaintext

Sao chép mã

pick aaa1111 A first commit

pick bbb2222 A second commit

fixup ddd4444 fixup! A second commit

pick ccc3333 A third commit

Để tránh phải gõ --autosquash mỗi lần rebase, bạn có thể bật tùy chọn này mặc định:

bash

Sao chép mã

$ git config --global rebase.autosquash true

## Chapter 17: Cherry Picking

Parameters Details

* -e, --edit: Với tùy chọn này, git cherry-pick sẽ cho phép bạn chỉnh sửa thông điệp commit trước khi commit.
* -x: Khi ghi lại commit, thêm một dòng cho biết "(cherry picked from commit …)" vào thông điệp commit gốc để chỉ ra commit này được cherry-pick từ đâu. Điều này chỉ thực hiện cho các cherry-pick không có xung đột.
* --ff: Nếu HEAD hiện tại giống với cha của commit được cherry-pick, thì một fast forward đến commit này sẽ được thực hiện.
* --continue: Tiếp tục thao tác đang tiến hành bằng cách sử dụng thông tin trong .git/sequencer. Có thể được sử dụng để tiếp tục sau khi giải quyết xung đột trong một cherry-pick hoặc revert thất bại.
* --quit: Quên thao tác hiện tại đang tiến hành. Có thể được sử dụng để xóa trạng thái sequencer sau một cherry-pick hoặc revert thất bại.
* --abort: Hủy thao tác và quay lại trạng thái trước khi bắt đầu.

Cherry-pick sẽ lấy bản vá được giới thiệu trong một commit và cố gắng áp dụng nó trên nhánh mà bạn hiện đang ở.

### Section 17.1: Copying a Commit from One Branch to Another

Lệnh git cherry-pick <commit-hash> sẽ áp dụng các thay đổi được thực hiện trong một commit hiện có vào một nhánh khác, trong khi ghi lại một commit mới. Về cơ bản, bạn có thể sao chép các commit từ nhánh này sang nhánh khác.

Giả sử cây sau đây (Source):

css

Sao chép mã

dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [master]

\

76cada - 62ecb3 - b886a0 [feature]

Giả sử chúng ta muốn sao chép b886a0 sang master (trên cùng của 5a6057).

Chúng ta có thể chạy:

bash

Sao chép mã

git checkout master

git cherry-pick b886a0

Bây giờ cây của chúng ta sẽ trông như sau:

css

Sao chép mã

dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 - a66b23 [master]

\

76cada - 62ecb3 - b886a0 [feature]

Nơi commit mới a66b23 có cùng nội dung (diff source, thông điệp commit) như b886a0 (nhưng có cha mẹ khác). Lưu ý rằng cherry-pick chỉ chọn các thay đổi trên commit đó (trong trường hợp này là b886a0) chứ không phải tất cả các thay đổi trong nhánh feature (để làm điều này, bạn sẽ phải sử dụng rebase hoặc merge).

### Section 17.2: Copying a Range of Commits from One Branch to Another

Lệnh git cherry-pick <commit-A>..<commit-B> sẽ đặt mọi commit sau A và bao gồm cả B lên trên nhánh hiện tại được checkout.

bash

Sao chép mã

git cherry-pick <commit-A>..<commit-B>

Lệnh git cherry-pick <commit-A>^..<commit-B> sẽ đặt commit A và mọi commit lên đến và bao gồm cả B lên trên nhánh hiện tại được checkout.

bash

Sao chép mã

git cherry-pick <commit-A>^..<commit-B>

### Section 17.3: Checking if a Cherry-Pick is Required

Trước khi bắt đầu quá trình cherry-pick, bạn có thể kiểm tra xem commit bạn muốn cherry-pick đã tồn tại trong nhánh mục tiêu chưa, trong trường hợp đó bạn không cần phải làm gì cả.

Lệnh git branch --contains <commit> liệt kê các nhánh local chứa commit được chỉ định.

bash

Sao chép mã

git branch --contains <commit>

Lệnh git branch -r --contains <commit> cũng bao gồm các nhánh theo dõi remote trong danh sách.

bash

Sao chép mã

git branch -r --contains <commit>

### Section 17.4: Find Commits Yet to be Applied to Upstream

Lệnh git cherry hiển thị các thay đổi chưa được cherry-pick.

Ví dụ:

bash

Sao chép mã

git checkout master

git cherry development

... và xem kết quả như sau:

diff

Sao chép mã

+ 492508acab7b454eee8b805f8ba906056eede0ff

- 5ceb5a9077ddb9e78b1e8f24bfc70e674c627949

+ b4459544c000f4d51d1ec23f279d9cdb19c1d32b

+ b6ce3b78e938644a293b2dd2a15b2fecb1b54cd9

Các commit bắt đầu bằng + sẽ là những commit chưa được cherry-pick vào development.

Cú pháp:

bash

Sao chép mã

git cherry [-v] [<upstream> [<head> [<limit>]]]

Các tùy chọn:

* -v: Hiển thị các tiêu đề commit bên cạnh các SHA1.
* <upstream>: Nhánh upstream để tìm các commit tương đương. Mặc định là nhánh upstream của HEAD.
* <head>: Nhánh làm việc; mặc định là HEAD.
* <limit>: Không báo cáo các commit đến (và bao gồm) giới hạn.

Tham khảo tài liệu git-cherry để biết thêm thông tin.

## Chapter 18: Recovering

### Section 18.1: Recovering from a Reset

Với Git, bạn có thể (hầu như) luôn luôn quay ngược thời gian lại.

Đừng ngại thử nghiệm các lệnh thay đổi lịch sử. Git không xóa các commit của bạn trong 90 ngày theo mặc định, và trong thời gian đó, bạn có thể dễ dàng khôi phục chúng từ reflog:

bash

Sao chép mã

$ git reset @~3 # quay lại 3 commit trước

$ 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

... các commit trước đó ...

$ git reset 2c52489

... và bạn đã trở lại nơi bạn bắt đầu.

Lưu ý: Hãy cẩn thận với các tùy chọn như --hard và --force — chúng có thể loại bỏ dữ liệu.

Cũng tránh viết lại lịch sử trên các nhánh mà bạn đang cộng tác.

### Section 18.2: Recover from Git Stash

Để lấy stash gần nhất sau khi chạy git stash, sử dụng:

bash

Sao chép mã

git stash apply

Để xem danh sách các stash của bạn, sử dụng:

bash

Sao chép mã

git stash list

Bạn sẽ nhận được một danh sách giống như sau:

plaintext

Sao chép mã

stash@{0}: WIP on master: 67a4e01 Merge tests into develop

stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login

Chọn một stash khác để khôi phục với số xuất hiện cho stash bạn muốn:

bash

Sao chép mã

git stash apply stash@{2}

Bạn cũng có thể sử dụng git stash pop, nó hoạt động giống như git stash apply như:

bash

Sao chép mã

git stash pop

hoặc

bash

Sao chép mã

git stash pop stash@{2}

Sự khác biệt giữa git stash apply và git stash pop:

* git stash pop: Dữ liệu stash sẽ bị xóa khỏi danh sách stash.

Ví dụ:

bash

Sao chép mã

git stash list

Bạn sẽ nhận được một danh sách giống như sau:

plaintext

Sao chép mã

stash@{0}: WIP on master: 67a4e01 Merge tests into develop

stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login

Bây giờ, pop dữ liệu stash bằng lệnh:

bash

Sao chép mã

git stash pop

Kiểm tra lại danh sách stash:

bash

Sao chép mã

git stash list

Bạn sẽ thấy một dữ liệu stash đã bị xóa (popped) khỏi danh sách stash và stash@{1} đã trở thành stash@{0}.

### Section 18.3: Recovering from a Lost Commit

Trong trường hợp bạn đã quay lại một commit cũ và mất một commit mới hơn, bạn có thể khôi phục commit bị mất bằng cách chạy:

bash

Sao chép mã

git reflog

Sau đó, tìm commit bị mất của bạn và reset lại nó bằng cách:

bash

Sao chép mã

git reset HEAD --hard <sha1-of-commit>

### Section 18.4: Restore a Deleted File after a Commit

Trong trường hợp bạn đã vô tình commit xóa một file và sau đó nhận ra rằng bạn cần nó lại.

Đầu tiên, tìm id commit của commit đã xóa file của bạn:

bash

Sao chép mã

git log --diff-filter=D --summary

Sẽ cung cấp cho bạn một bản tóm tắt được sắp xếp của các commit đã xóa các file.

Sau đó, tiến hành khôi phục file bằng:

bash

Sao chép mã

git checkout 81eeccf~1 <your-lost-file-name>

(Thay 81eeccf bằng id commit của bạn)

### Section 18.5: Restore File to a Previous Version

Để khôi phục một file về phiên bản trước đó, bạn có thể sử dụng reset:

bash

Sao chép mã

git reset <sha1-of-commit> <file-name>

Nếu bạn đã thực hiện các thay đổi cục bộ đối với file (mà bạn không cần!), bạn cũng có thể sử dụng tùy chọn --hard:

bash

Sao chép mã

git reset --hard <sha1-of-commit> <file-name>

### Section 18.6: Recover a Deleted Branch

Để khôi phục một nhánh đã bị xóa, bạn cần tìm commit là head của nhánh đã bị xóa bằng cách chạy:

bash

Sao chép mã

git reflog

Sau đó, bạn có thể tạo lại nhánh bằng cách chạy:

bash

Sao chép mã

git checkout -b <branch-name> <sha1-of-commit>

Lưu ý rằng bạn sẽ không thể khôi phục các nhánh đã bị xóa nếu trình thu gom rác của git đã xóa các commit không có tham chiếu (dangling commits). Luôn luôn có một bản sao lưu của kho lưu trữ của bạn, đặc biệt khi bạn làm việc trong một nhóm nhỏ hoặc dự án độc quyền.

## Chapter 19: Git Clean

Parameters Details

* -d: Xóa các thư mục không được theo dõi ngoài các tệp không được theo dõi. Nếu một thư mục không được theo dõi được quản lý bởi một kho lưu trữ Git khác, nó sẽ không bị xóa theo mặc định. Sử dụng tùy chọn -f hai lần nếu bạn thực sự muốn xóa một thư mục như vậy.
* -f, --force: Nếu biến cấu hình Git clean.requireForce không được đặt thành false, git clean sẽ từ chối xóa các tệp hoặc thư mục trừ khi được cung cấp -f, -n hoặc -i. Git sẽ từ chối xóa các thư mục có thư mục con .git hoặc tệp trừ khi được cung cấp -f lần thứ hai.
* -i, --interactive: Tương tác nhắc nhở việc loại bỏ từng tệp.
* -n, --dry-run: Chỉ hiển thị danh sách các tệp sẽ bị xóa, mà không thực sự xóa chúng.
* -q, --quiet: Chỉ hiển thị các lỗi, không phải danh sách các tệp đã xóa thành công.

### Section 19.1: Clean Interactively

Chạy lệnh sau để xóa tệp không theo dõi một cách tương tác:

bash

Sao chép mã

git clean -i

Lệnh này sẽ in ra các mục cần xóa và yêu cầu xác nhận qua các lệnh như sau:

plaintext

Sao chép mã

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>

Tùy chọn tương tác -i có thể được thêm cùng với các tùy chọn khác như -X, -d, v.v.

### Section 19.2: Forcefully Remove Untracked Files

Chạy lệnh sau để xóa tất cả các tệp không được theo dõi:

bash

Sao chép mã

git clean -f

### Section 19.3: Clean Ignored Files

Chạy lệnh sau để xóa tất cả các tệp bị bỏ qua từ thư mục hiện tại và tất cả các thư mục con:

bash

Sao chép mã

git clean -fX

Để xem trước tất cả các tệp sẽ bị xóa, chạy lệnh sau:

bash

Sao chép mã

git clean -Xn

### Section 19.4: Clean All Untracked Directories

Chạy lệnh sau để xóa tất cả các thư mục không được theo dõi và các tệp trong đó. Lệnh này sẽ bắt đầu từ thư mục làm việc hiện tại và sẽ lặp qua tất cả các thư mục con:

bash

Sao chép mã

git clean -fd

Để xem trước tất cả các thư mục sẽ bị xóa, chạy lệnh sau:

bash

Sao chép mã

git clean -dn

Các lệnh trên cung cấp cho bạn các cách để quản lý và xóa các tệp và thư mục không được theo dõi trong kho lưu trữ Git của bạn, đảm bảo rằng bạn có thể duy trì một môi trường làm việc sạch sẽ và có tổ chức.

## Chapter 20: Using a .gitattributes file

### Section 20.1: Automatic Line Ending Normalization

Tạo một tệp .gitattributes trong thư mục gốc của dự án chứa:

plaintext

Sao chép mã

\* text=auto

Điều này sẽ dẫn đến tất cả các tệp văn bản (được Git nhận diện) được commit với LF, nhưng được checkout theo mặc định của hệ điều hành máy chủ.

Điều này tương đương với các giá trị mặc định được khuyến nghị của core.autocrlf là:

* input trên Linux/macOS
* true trên Windows

### Section 20.2: Identify Binary Files

Git khá tốt trong việc nhận diện các tệp nhị phân, nhưng bạn có thể chỉ định rõ ràng các tệp nào là tệp nhị phân. Tạo một tệp .gitattributes trong thư mục gốc của dự án chứa:

plaintext

Sao chép mã

\*.png binary

binary là một thuộc tính macro tích hợp tương đương với -diff -merge -text.

### Section 20.3: Prefilled .gitattribute Templates

Nếu bạn không chắc chắn nên liệt kê những quy tắc nào trong tệp .gitattributes của mình, hoặc bạn chỉ muốn thêm các thuộc tính được chấp nhận chung vào dự án của mình, bạn có thể chọn hoặc tạo một tệp .gitattributes tại:

* [gitattributes.io](https://gitattributes.io/)
* [GitHub - alexkaratarakis/gitattributes](https://github.com/alexkaratarakis/gitattributes)

### Section 20.4: Disable Line Ending Normalization

Tạo một tệp .gitattributes trong thư mục gốc của dự án chứa:

plaintext

Sao chép mã

\* -text

Điều này tương đương với việc đặt core.autocrlf = false.

Sử dụng tệp .gitattributes giúp bạn kiểm soát cách Git xử lý các tệp trong dự án của bạn, từ việc chuẩn hóa dòng kết thúc đến việc xác định rõ các tệp nhị phân. Điều này có thể rất hữu ích trong việc duy trì tính nhất quán và quản lý các tệp trong dự án của bạn.

## Chapter 21: .mailmap File: Associating Contributor and Email Aliases

### Section 21.1: Merge Contributors by Aliases to Show Commit Count in Shortlog

Khi các cộng tác viên thêm vào một dự án từ các máy hoặc hệ điều hành khác nhau, có thể xảy ra việc họ sử dụng các địa chỉ email hoặc tên khác nhau, điều này sẽ làm phân mảnh danh sách và thống kê của các cộng tác viên.

Chạy lệnh git shortlog -sn để lấy danh sách các cộng tác viên và số lượng commit của họ có thể dẫn đến kết quả sau:

plaintext

Sao chép mã

Patrick Rothfuss 871

Elizabeth Moon 762

E. Moon 184

Rothfuss, Patrick 90

Sự phân mảnh/không liên kết này có thể được điều chỉnh bằng cách cung cấp một tệp văn bản thuần .mailmap, chứa các ánh xạ email.

Tất cả các tên và địa chỉ email được liệt kê trên một dòng sẽ được liên kết với thực thể đầu tiên được đề cập tương ứng.

Với ví dụ trên, một ánh xạ có thể trông như thế này:

plaintext

Sao chép mã

Patrick Rothfuss <fussy@kingkiller.com> Rothfuss, Patrick <fussy@kingkiller.com>

Elizabeth Moon <emoon@marines.mil> E. Moon <emoon@scifi.org>

Khi tệp này tồn tại trong thư mục gốc của dự án, chạy lại lệnh git shortlog -sn sẽ cho ra danh sách được tổng hợp:

plaintext

Sao chép mã

Patrick Rothfuss 961

Elizabeth Moon 946

### Tạo tệp .mailmap

1. **Tạo tệp .mailmap**:
   * Tạo một tệp mới tên là .mailmap trong thư mục gốc của dự án.
2. **Thêm ánh xạ email**:
   * Thêm các dòng ánh xạ vào tệp .mailmap. Mỗi dòng ánh xạ sẽ liên kết các tên và địa chỉ email khác nhau của một cộng tác viên vào một tên và địa chỉ email chính.

Ví dụ về nội dung của tệp .mailmap:

plaintext

Sao chép mã

Patrick Rothfuss <fussy@kingkiller.com> Rothfuss, Patrick <fussy@kingkiller.com>

Elizabeth Moon <emoon@marines.mil> E. Moon <emoon@scifi.org>

1. **Chạy lại lệnh git shortlog**:
   * Sau khi thêm tệp .mailmap, chạy lại lệnh git shortlog -sn để thấy kết quả đã được tổng hợp.

bash

Sao chép mã

git shortlog -sn

### Lợi ích của việc sử dụng tệp .mailmap

* **Kết hợp các đóng góp từ cùng một người**: Giúp tổng hợp và hiển thị số lượng commit chính xác từ cùng một người dùng, dù họ có sử dụng các email hoặc tên khác nhau.
* **Thống kê chính xác hơn**: Cải thiện độ chính xác của các số liệu thống kê và danh sách đóng góp của dự án.
* **Quản lý dự án tốt hơn**: Giúp các nhà quản lý dự án dễ dàng theo dõi và quản lý các đóng góp từ các thành viên khác nhau trong nhóm.

Sử dụng tệp .mailmap là một cách hiệu quả để đảm bảo rằng các đóng góp của cùng một người dùng không bị phân mảnh và thống kê được hiển thị chính xác.

## Chương 22: Phân tích các loại quy trình làm việc

### Mục 22.1: Quy trình làm việc tập trung

Với mô hình quy trình làm việc cơ bản này, nhánh master chứa tất cả các phát triển đang hoạt động. Các cộng tác viên sẽ cần phải đặc biệt chắc chắn rằng họ đã pull các thay đổi mới nhất trước khi tiếp tục phát triển, vì nhánh này sẽ thay đổi nhanh chóng. Mọi người đều có quyền truy cập vào kho lưu trữ này và có thể commit các thay đổi trực tiếp vào nhánh master.

Hình ảnh minh họa của mô hình này:

Đây là mô hình kiểm soát phiên bản cổ điển, trên đó các hệ thống cũ như Subversion và CVS được xây dựng. Các phần mềm hoạt động theo cách này được gọi là Hệ thống Kiểm soát Phiên bản Tập trung (CVCS). Mặc dù Git có thể hoạt động theo cách này, nhưng có những nhược điểm đáng chú ý, chẳng hạn như phải thực hiện merge trước mỗi lần pull. Rất có thể một nhóm làm việc theo cách này, nhưng việc giải quyết xung đột merge liên tục có thể tiêu tốn nhiều thời gian quý báu.

Đây là lý do tại sao Linus Torvalds tạo ra Git không phải như một CVCS, mà là một DVCS, hay Hệ thống Kiểm soát Phiên bản Phân tán, tương tự như Mercurial. Lợi thế của cách làm việc mới này là tính linh hoạt được thể hiện trong các ví dụ khác trên trang này.

### Mục 22.2: Quy trình làm việc Gitflow

Ban đầu được đề xuất bởi Vincent Driessen, Gitflow là một quy trình phát triển sử dụng git và một số nhánh được định nghĩa trước. Điều này có thể được coi là một trường hợp đặc biệt của Quy trình làm việc Nhánh Tính năng.

Ý tưởng của mô hình này là có các nhánh riêng biệt dành riêng cho các phần cụ thể trong quá trình phát triển:

* **Nhánh master** luôn là mã sản xuất mới nhất. Mã thử nghiệm không thuộc về đây.
* **Nhánh develop** chứa tất cả các phát triển mới nhất. Những thay đổi phát triển này có thể là bất cứ điều gì, nhưng các tính năng lớn hơn được dành riêng cho các nhánh riêng của chúng. Mã ở đây luôn được làm việc và merge vào release trước khi phát hành/triển khai.
* **Nhánh hotfix** dành cho các sửa lỗi nhỏ, không thể chờ đến bản phát hành tiếp theo. Các nhánh hotfix tách ra từ master và được merge lại vào cả master và develop.
* **Nhánh release** được sử dụng để phát hành phát triển mới từ develop sang master. Bất kỳ thay đổi phút chót nào, chẳng hạn như tăng số phiên bản, đều được thực hiện trong nhánh release, sau đó được merge lại vào master và develop. Khi triển khai một phiên bản mới, master nên được gắn thẻ với số phiên bản hiện tại (ví dụ: sử dụng semantic versioning) để tham chiếu trong tương lai và dễ dàng rollback.
* **Nhánh feature** dành riêng cho các tính năng lớn hơn. Những tính năng này được phát triển cụ thể trong các nhánh được chỉ định và tích hợp với develop khi hoàn thành. Các nhánh feature riêng biệt giúp tách biệt phát triển và có thể triển khai các tính năng đã hoàn thành một cách độc lập.

Hình ảnh minh họa của mô hình này:

### Mục 22.3: Quy trình làm việc Nhánh Tính năng

Ý tưởng cốt lõi của Quy trình làm việc Nhánh Tính năng là tất cả các phát triển tính năng nên diễn ra trong một nhánh dành riêng thay vì nhánh master. Sự đóng gói này giúp nhiều nhà phát triển dễ dàng làm việc trên một tính năng cụ thể mà không làm gián đoạn cơ sở mã chính. Nó cũng có nghĩa là nhánh master sẽ không bao giờ chứa mã bị hỏng, điều này là một lợi thế lớn cho các môi trường tích hợp liên tục.

Việc đóng gói phát triển tính năng cũng cho phép tận dụng các yêu cầu pull, đây là cách để bắt đầu các cuộc thảo luận xung quanh một nhánh. Chúng cho phép các nhà phát triển khác phê duyệt một tính năng trước khi nó được tích hợp vào dự án chính thức. Hoặc, nếu bạn bị mắc kẹt giữa chừng khi phát triển một tính năng, bạn có thể mở một yêu cầu pull để hỏi ý kiến từ các đồng nghiệp. Điểm quan trọng là, các yêu cầu pull giúp nhóm của bạn dễ dàng bình luận về công việc của nhau.

Dựa trên Hướng dẫn của Atlassian.

### Mục 22.4: Quy trình làm việc GitHub Flow

Phổ biến trong nhiều dự án mã nguồn mở nhưng không chỉ giới hạn ở đó.

Nhánh master của một vị trí cụ thể (Github, Gitlab, Bitbucket, máy chủ cục bộ) chứa phiên bản có thể triển khai mới nhất. Đối với mỗi tính năng mới/sửa lỗi/thay đổi kiến trúc, mỗi nhà phát triển tạo một nhánh.

Các thay đổi diễn ra trên nhánh đó và có thể được thảo luận trong một yêu cầu pull, đánh giá mã, v.v. Khi được chấp nhận, chúng sẽ được hợp nhất vào nhánh master.

Quy trình đầy đủ của Scott Chacon:

1. Bất kỳ điều gì trong nhánh master đều có thể triển khai.
2. Để làm việc trên một điều mới, hãy tạo một nhánh có tên mô tả từ nhánh master (ví dụ: new-oauth2-scopes).
3. Commit vào nhánh đó cục bộ và thường xuyên đẩy công việc của bạn lên nhánh cùng tên trên máy chủ.
4. Khi bạn cần phản hồi hoặc trợ giúp, hoặc bạn nghĩ rằng nhánh đã sẵn sàng để hợp nhất, hãy mở một yêu cầu pull.
5. Sau khi người khác đã đánh giá và phê duyệt tính năng, bạn có thể hợp nhất nó vào nhánh master.
6. Một khi nó được hợp nhất và đẩy lên 'master', bạn có thể và nên triển khai ngay lập tức.

Ban đầu được trình bày trên trang web cá nhân của Scott Chacon.

Hình ảnh minh họa của tham chiếu GitHub Flow.

### Mục 22.5: Quy trình làm việc Forking

Loại quy trình làm việc này khác biệt hoàn toàn so với các quy trình khác được đề cập trong chủ đề này. Thay vì có một kho lưu trữ tập trung mà tất cả các nhà phát triển có quyền truy cập, mỗi nhà phát triển có kho lưu trữ riêng của mình được fork từ kho lưu trữ chính. Lợi thế của điều này là các nhà phát triển có thể đăng bài lên kho lưu trữ của riêng mình thay vì kho lưu trữ chia sẻ và một người bảo trì có thể tích hợp các thay đổi từ các kho lưu trữ được fork vào kho lưu trữ gốc khi thích hợp.

Hình ảnh minh họa cho quy trình làm việc này như sau:

## Chương 23: Pulling (Kéo)

Tham số và Chi tiết

* **--quiet**: Không hiển thị đầu ra văn bản.
* **-q**: Viết tắt cho --quiet.
* **--verbose**: Hiển thị đầu ra chi tiết. Được truyền tới các lệnh fetch và merge/rebase tương ứng.
* **-v**: Viết tắt cho --verbose.
* **--[no-]recurse-submodules[=yes|on-demand|no]**: Có nên fetch các commit mới cho các submodule không? (Lưu ý rằng đây không phải là một pull/checkout)

Không giống như push với Git, nơi các thay đổi cục bộ của bạn được gửi đến máy chủ của kho lưu trữ trung tâm, pull với Git lấy mã hiện tại trên máy chủ và 'kéo' nó xuống từ máy chủ kho lưu trữ về máy cục bộ của bạn. Chủ đề này giải thích quá trình kéo mã từ một kho lưu trữ bằng Git cũng như các tình huống có thể gặp phải khi kéo mã khác vào bản sao cục bộ.

### Mục 23.1: Kéo các thay đổi vào kho lưu trữ cục bộ

#### Pull đơn giản

Khi bạn làm việc trên một kho lưu trữ từ xa (ví dụ, GitHub) với người khác, bạn sẽ muốn chia sẻ các thay đổi của mình với họ. Khi họ đã đẩy các thay đổi của mình lên kho lưu trữ từ xa, bạn có thể lấy những thay đổi đó bằng cách kéo từ kho lưu trữ này.

bash

Sao chép mã

git pull

Sẽ làm điều đó, trong hầu hết các trường hợp.

#### Pull từ một remote hoặc branch khác

Bạn có thể kéo các thay đổi từ một remote hoặc branch khác bằng cách chỉ định tên của chúng.

bash

Sao chép mã

git pull origin feature-A

Sẽ kéo branch feature-A từ origin vào branch cục bộ của bạn. Lưu ý rằng bạn có thể cung cấp trực tiếp một URL thay vì tên remote, và một tên đối tượng như một commit SHA thay vì tên branch.

#### Pull thủ công

Để bắt chước hành vi của lệnh git pull, bạn có thể sử dụng git fetch sau đó git merge.

bash

Sao chép mã

git fetch origin # lấy các đối tượng và cập nhật refs từ origin

git merge origin/feature-A # thực hiện merge thực sự

Điều này có thể cho bạn nhiều kiểm soát hơn và cho phép bạn kiểm tra nhánh từ xa trước khi merge nó. Thực tế, sau khi fetch, bạn có thể xem các nhánh từ xa bằng lệnh git branch -a và kiểm tra chúng bằng

bash

Sao chép mã

git checkout -b local-branch-name origin/feature-A # kiểm tra nhánh từ xa

# kiểm tra nhánh, tạo commit, squash, sửa đổi hoặc bất cứ điều gì

git checkout merging-branches # di chuyển đến nhánh đích

git merge local-branch-name # thực hiện merge

Điều này có thể rất hữu ích khi xử lý các yêu cầu pull.

### Mục 23.2: Cập nhật với các thay đổi cục bộ

Khi có các thay đổi cục bộ, lệnh git pull sẽ báo lỗi và không thực hiện:

bash

Sao chép mã

error: Your local changes to the following files would be overwritten by merge

Để cập nhật (giống như lệnh svn update trong subversion), bạn có thể chạy:

bash

Sao chép mã

git stash

git pull --rebase

git stash pop

Một cách thuận tiện là định nghĩa một alias sử dụng:

#### Phiên bản < 2.9

bash

Sao chép mã

git config --global alias.up '!git stash && git pull --rebase && git stash pop'

#### Phiên bản ≥ 2.9

bash

Sao chép mã

git config --global alias.up 'pull --rebase --autostash'

Sau đó, bạn chỉ cần sử dụng:

bash

Sao chép mã

git up

### Mục 23.3: Pull, ghi đè lên cục bộ

bash

Sao chép mã

git fetch

git reset --hard origin/master

Chú ý: Trong khi các commit bị loại bỏ bằng lệnh reset --hard có thể được khôi phục bằng cách sử dụng reflog và reset, các thay đổi chưa commit sẽ bị xóa vĩnh viễn. Thay đổi origin và master thành remote và branch bạn muốn kéo bắt buộc tương ứng, nếu chúng được đặt tên khác.

### Mục 23.4: Kéo mã từ remote

bash

Sao chép mã

git pull

### Mục 23.5: Giữ lịch sử tuyến tính khi kéo

#### Rebase khi kéo

Nếu bạn đang kéo các commit mới từ kho lưu trữ từ xa và bạn có các thay đổi cục bộ trên nhánh hiện tại thì git sẽ tự động merge phiên bản từ xa và phiên bản của bạn. Nếu bạn muốn giảm số lần merge trên nhánh của mình, bạn có thể yêu cầu git rebase các commit của bạn trên phiên bản từ xa của nhánh.

bash

Sao chép mã

git pull --rebase

#### Đặt làm hành vi mặc định

Để làm điều này trở thành hành vi mặc định cho các nhánh mới được tạo, nhập lệnh sau:

bash

Sao chép mã

git config branch.autosetuprebase always

Để thay đổi hành vi của một nhánh hiện có, sử dụng lệnh này:

bash

Sao chép mã

git config branch.BRANCH\_NAME.rebase true

Và

bash

Sao chép mã

git pull --no-rebase

Để thực hiện một lệnh pull bình thường.

#### Kiểm tra xem có thể fast-forward

Để chỉ cho phép fast forwarding nhánh cục bộ, bạn có thể sử dụng:

bash

Sao chép mã

git pull --ff-only

Điều này sẽ hiển thị lỗi khi nhánh cục bộ không thể fast-forward, và cần phải được rebase hoặc merge với upstream.

### Mục 23.6: Pull, "quyền bị từ chối"

Một số vấn đề có thể xảy ra nếu thư mục .git có quyền sai. Sửa vấn đề này bằng cách đặt chủ sở hữu của toàn bộ thư mục .git. Đôi khi nó xảy ra khi người dùng khác pull và thay đổi quyền của thư mục hoặc tệp .git.

Để khắc phục vấn đề:

bash

Sao chép mã

chown -R youruser:yourgroup .git/

## Chương 24: Hooks

### Mục 24.1: Pre-push

Có sẵn trong Git 1.8.2 và các phiên bản mới hơn.

#### Phiên bản ≥ 1.8

Các hooks pre-push có thể được sử dụng để ngăn chặn một đẩy (push) được thực hiện. Điều này có ích khi bạn muốn chặn các đẩy thủ công ngẫu nhiên đến các nhánh cụ thể, hoặc chặn các đẩy nếu một kiểm tra được thiết lập không thành công (kiểm tra đơn vị, kiểm tra cú pháp).

Một hook pre-push được tạo ra bằng cách đơn giản tạo một tệp có tên là pre-push trong thư mục .git/hooks/, và (lưu ý), đảm bảo rằng tệp đó có thể thực thi: chmod +x ./git/hooks/pre-push.

Dưới đây là một ví dụ từ Hannah Wolfe chặn đẩy lên nhánh master:

bash

Sao chép mã

#!/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 sẽ được thực hiện

fi

exit 1 # push sẽ không được thực hiện

else

exit 0 # push sẽ được thực hiện

fi

Dưới đây là một ví dụ từ Volkan Unsal đảm bảo rằng các kiểm tra RSpec đều thành công trước khi cho phép đẩy:

ruby

Sao chép mã

#!/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

# tìm xem có lỗi nào không

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 khi các kiểm tra thành công:

# puts "Xem kết quả kiểm tra tại #{File.expand\_path(html\_path)}"

sleep 1

exit 0

else

puts "\aCOMMIT FAILED!!"

puts "Xem kết quả RSpec của bạn tại #{File.expand\_path(html\_path)}"

puts

puts "#{errors} failed! #{examples} run, #{pending} pending"

# Mở HTML Output khi các kiểm tra thất bại

# `open #{html\_path}`

exit 1

end

Như bạn có thể thấy, có rất nhiều khả năng, nhưng mấu chốt là exit 0 nếu mọi thứ diễn ra tốt đẹp và exit 1 nếu có vấn đề. Bất cứ khi nào bạn exit 1, việc đẩy sẽ bị ngăn chặn và mã của bạn sẽ ở trạng thái như trước khi chạy git push.

Khi sử dụng các hooks phía client, hãy nhớ rằng người dùng có thể bỏ qua tất cả các hooks phía client bằng cách sử dụng tùy chọn "--no-verify" trên một lệnh đẩy. Nếu bạn dựa vào hook để thực thi quy trình, bạn có thể gặp rủi ro.

**Tài liệu**: <https://git-scm.com/docs/githooks#_pre_push>

**Ví dụ chính thức**: <https://github.com/git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample>

### Mục 24.2: Kiểm tra build Maven (hoặc hệ thống build khác) trước khi commit

sh

Sao chép mã

.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

### Mục 24.3: Tự động chuyển tiếp một số đẩy nhất định tới các kho lưu trữ khác

Các hooks post-receive có thể được sử dụng để tự động chuyển tiếp các đẩy đến một kho lưu trữ khác.

sh

Sao chép mã

$ 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

Trong ví dụ này, biểu thức chính quy egrep tìm kiếm một định dạng nhánh cụ thể (ở đây: JIRA-12345 như được sử dụng để đặt tên cho các vấn đề Jira). Bạn có thể bỏ qua phần này nếu bạn muốn chuyển tiếp tất cả các nhánh, tất nhiên.

### Mục 24.4: Commit-msg

Hook này tương tự như hook prepare-commit-msg, nhưng nó được gọi sau khi người dùng nhập tin nhắn commit thay vì trước đó. Hook này thường được sử dụng để cảnh báo các nhà phát triển nếu tin nhắn commit của họ không đúng định dạng.

Đối số duy nhất được truyền vào hook này là tên của tệp chứa tin nhắn. Nếu bạn không thích tin nhắn mà người dùng đã nhập, bạn có thể thay đổi tệp này tại chỗ (giống như prepare-commit-msg) hoặc bạn có thể hủy bỏ commit hoàn toàn bằng cách thoát với trạng thái không bằng không.

Ví dụ sau đây được sử dụng để kiểm tra xem từ "ticket" kèm theo số có xuất hiện trong tin nhắn commit hay không:

sh

Sao chép mã

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

### Mục 24.5: Local hooks

Local hooks chỉ ảnh hưởng đến các kho lưu trữ cục bộ mà chúng cư trú. Mỗi nhà phát triển có thể thay đổi các hook cục bộ của riêng họ, vì vậy chúng không thể được sử dụng một cách đáng tin cậy để thực thi một chính sách commit. Chúng được thiết kế để giúp các nhà phát triển tuân thủ các hướng dẫn nhất định và tránh các vấn đề tiềm ẩn trong tương lai.

Có sáu loại local hooks: pre-commit, prepare-commit-msg, commit-msg, post-commit, post-checkout và pre-rebase.

Bốn hook đầu tiên liên quan đến commit và cho phép bạn kiểm soát một phần nào đó trong chu kỳ sống của commit. Hai hook cuối cùng cho phép bạn thực hiện một số hành động bổ sung hoặc kiểm tra an toàn cho các lệnh git checkout và git rebase.

Tất cả các hook "pre-" cho phép bạn thay đổi hành động sắp xảy ra, trong khi các hook "post-" chủ yếu được sử dụng để thông báo.

### Mục 24.6: Post-checkout

Hook này hoạt động tương tự như hook post-commit, nhưng nó được gọi bất cứ khi nào bạn kiểm tra thành công một tham chiếu với git checkout. Đây có thể là một công cụ hữu ích để xóa thư mục làm việc của bạn khỏi các tệp được tạo tự động mà nếu không sẽ gây nhầm lẫn.

Hook này chấp nhận ba tham số:

1. ref của HEAD trước đó,
2. ref của HEAD mới, và
3. cờ cho biết đó là một nhánh checkout hay một file checkout (1 hoặc 0, tương ứng).

Trạng thái thoát của nó không ảnh hưởng đến lệnh git checkout.

### Mục 24.7: Post-commit

Hook này được gọi ngay sau hook commit-msg. Nó không thể thay đổi kết quả của hoạt động git commit, do đó nó chủ yếu được sử dụng cho mục đích thông báo.

Script này không có tham số và trạng thái thoát của nó không ảnh hưởng đến commit theo bất kỳ cách nào.

### Mục 24.8: Post-receive

Hook này được gọi sau khi một hoạt động đẩy (push) thành công. Nó thường được sử dụng cho mục đích thông báo.

Script này không có tham số, nhưng nhận thông tin giống như pre-receive thông qua đầu vào chuẩn:

php

Sao chép mã

<old-value> <new-value> <ref-name>

### Mục 24.9: Pre-commit

Hook này được thực thi mỗi lần bạn chạy git commit, để xác minh những gì sắp được commit. Bạn có thể sử dụng hook này để kiểm tra snapshot sắp được commit.

Loại hook này hữu ích cho việc chạy các kiểm tra tự động để đảm bảo commit sắp tới không phá vỡ chức năng hiện có của dự án. Loại hook này cũng có thể kiểm tra lỗi khoảng trắng hoặc lỗi EOL.

Không có tham số nào được truyền vào script pre-commit, và việc thoát với trạng thái không bằng không sẽ hủy bỏ toàn bộ commit.

### Mục 24.10: Prepare-commit-msg

Hook này được gọi sau hook pre-commit để điền vào trình soạn thảo văn bản với một tin nhắn commit. Nó thường được sử dụng để thay đổi các tin nhắn commit được tạo tự động cho các commit được squash hoặc merge.

Một đến ba đối số được truyền vào hook này:

1. Tên của tệp tạm thời chứa tin nhắn.
2. Loại commit, có thể là:
   * message (-m hoặc tùy chọn -F),
   * template (-t tùy chọn),
   * merge (nếu là commit merge), hoặc
   * squash (nếu là commit squash).
3. SHA1 hash của commit liên quan. Chỉ được cung cấp nếu -c, -C, hoặc tùy chọn --amend được cung cấp.

Tương tự như pre-commit, thoát với trạng thái không bằng không sẽ hủy bỏ commit.

### Mục 24.11: Pre-rebase

Hook này được gọi trước khi git rebase bắt đầu thay đổi cấu trúc mã. Hook này thường được sử dụng để đảm bảo rằng một thao tác rebase là phù hợp.

Hook này nhận hai tham số:

1. Nhánh upstream mà chuỗi nhánh đã được fork từ đó, và
2. Nhánh đang được rebase (trống khi rebase nhánh hiện tại).

Bạn có thể hủy bỏ thao tác rebase bằng cách thoát với trạng thái không bằng không.

### Mục 24.12: Pre-receive

Hook này được thực thi mỗi lần ai đó sử dụng git push để đẩy các commit đến kho lưu trữ. Nó luôn cư trú trong kho lưu trữ từ xa là đích đến của việc đẩy và không phải trong kho lưu trữ gốc (cục bộ).

Hook này chạy trước khi bất kỳ tham chiếu nào được cập nhật. Nó thường được sử dụng để thực thi bất kỳ loại chính sách phát triển nào.

Script này không có tham số, nhưng mỗi ref đang được đẩy được truyền vào script trên một dòng riêng biệt trên đầu vào chuẩn với định dạng sau:

php

Sao chép mã

<old-value> <new-value> <ref-name>

### Mục 24.13: Update

Hook này được gọi sau pre-receive, và nó hoạt động theo cách tương tự. Nó được gọi trước khi bất cứ điều gì thực sự được cập nhật, nhưng được gọi riêng cho từng ref được đẩy thay vì tất cả các ref cùng một lúc.

Hook này chấp nhận ba tham số:

1. Tên của ref đang được cập nhật,
2. Tên đối tượng cũ được lưu trữ trong ref, và
3. Tên đối tượng mới được lưu trữ trong ref.

Đây là thông tin giống như được truyền vào pre-receive, nhưng vì update được gọi riêng cho từng ref, bạn có thể từ chối một số ref trong khi cho phép những ref khác.

## Chương 25: Nhân Bản Kho Lưu Trữ

### Mục 25.1: Nhân Bản Nông

Nhân bản một kho lưu trữ lớn (như một dự án với nhiều năm lịch sử) có thể mất nhiều thời gian, hoặc thất bại vì lượng dữ liệu cần chuyển. Trong trường hợp bạn không cần toàn bộ lịch sử, bạn có thể thực hiện nhân bản nông:

sh

Sao chép mã

git clone [repo\_url] --depth 1

Lệnh trên sẽ chỉ lấy commit cuối cùng từ kho lưu trữ từ xa.

Lưu ý rằng bạn có thể không thể giải quyết các hợp nhất trong một kho lưu trữ nông. Thường thì tốt hơn nên lấy ít nhất là số lượng commit bạn cần để lùi lại và giải quyết hợp nhất. Ví dụ, để lấy 50 commit gần nhất:

sh

Sao chép mã

git clone [repo\_url] --depth 50

Sau đó, nếu cần thiết, bạn có thể lấy phần còn lại của kho lưu trữ: Phiên bản ≥ 1.8.3

sh

Sao chép mã

git fetch --unshallow # tương đương với git fetch --depth=2147483647

lấy phần còn lại của kho lưu trữ

Phiên bản < 1.8.3

sh

Sao chép mã

git fetch --depth=1000 # lấy 1000 commit gần nhất

### Mục 25.2: Nhân Bản Thông Thường

Để tải về toàn bộ kho lưu trữ bao gồm lịch sử đầy đủ và tất cả các nhánh, gõ:

sh

Sao chép mã

git clone <url>

Ví dụ trên sẽ đặt nó vào một thư mục có cùng tên với tên kho lưu trữ. Để tải về kho lưu trữ và lưu nó vào một thư mục cụ thể, gõ:

sh

Sao chép mã

git clone <url> [directory]

Để biết thêm chi tiết, hãy tham khảo Clone a repository.

### Mục 25.3: Nhân Bản Một Nhánh Cụ Thể

Để nhân bản một nhánh cụ thể của kho lưu trữ, gõ --branch <branch name> trước URL của kho lưu trữ:

sh

Sao chép mã

git clone --branch <branch name> <url> [directory]

Để sử dụng tùy chọn rút gọn cho --branch, gõ -b. Lệnh này tải về toàn bộ kho lưu trữ và kiểm tra <branch name>. Để tiết kiệm dung lượng đĩa, bạn có thể nhân bản lịch sử chỉ dẫn đến một nhánh duy nhất với:

sh

Sao chép mã

git clone --branch <branch\_name> --single-branch <url> [directory]

Nếu --single-branch không được thêm vào lệnh, lịch sử của tất cả các nhánh sẽ được nhân bản vào [directory]. Điều này có thể là vấn đề với các kho lưu trữ lớn. Để sau này hoàn tác cờ --single-branch và lấy phần còn lại của kho lưu trữ, sử dụng lệnh:

sh

Sao chép mã

git config remote.origin.fetch "+refs/heads/\*:refs/remotes/origin/\*"

git fetch origin

### Mục 25.4: Nhân Bản Đệ Quy

Phiên bản ≥ 1.6.5

sh

Sao chép mã

git clone <url> --recursive

Nhân bản kho lưu trữ và cũng nhân bản tất cả các submodules. Nếu các submodules chứa thêm các submodules khác, Git cũng sẽ nhân bản chúng.

### Mục 25.5: Nhân Bản Sử Dụng Proxy

Nếu bạn cần tải về các tệp với git dưới một proxy, thiết lập máy chủ proxy hệ thống có thể không đủ. Bạn cũng có thể thử các lệnh sau:

sh

Sao chép mã

git config --global http.proxy http://<proxy-server>:<port>/

|  |
| --- |
|  |

## Chương 26: Lưu Tạm (Stashing)

Tham Số Chi Tiết

* show: Hiển thị các thay đổi được ghi lại trong stash dưới dạng sự khác biệt giữa trạng thái stash và gốc ban đầu của nó. Khi không có <stash> được cung cấp, nó sẽ hiển thị stash mới nhất.
* list: Liệt kê các stash hiện có. Mỗi stash được liệt kê với tên của nó (ví dụ: stash@{0} là stash mới nhất, stash@{1} là stash trước đó, v.v.), tên của nhánh hiện tại khi stash được tạo, và một mô tả ngắn gọn về commit mà stash được dựa trên.
* pop: Loại bỏ một trạng thái stash duy nhất từ danh sách stash và áp dụng nó lên trạng thái cây làm việc hiện tại.
* apply: Giống như pop, nhưng không loại bỏ trạng thái khỏi danh sách stash.
* clear: Xóa tất cả các trạng thái stash. Lưu ý rằng các trạng thái này sau đó sẽ bị đối tượng quét dọn, và có thể không thể khôi phục được.
* drop: Loại bỏ một trạng thái stash duy nhất từ danh sách stash. Khi không có <stash> được cung cấp, nó sẽ loại bỏ stash mới nhất, tức là stash@{0}, nếu không thì <stash> phải là một tham chiếu log stash hợp lệ dưới dạng stash@{<revision>}.
* create: Tạo một stash (là một đối tượng commit thông thường) và trả về tên đối tượng của nó, mà không lưu trữ nó ở bất kỳ đâu trong namespace ref. Điều này được thiết kế để hữu ích cho các script. Đây có lẽ không phải là lệnh bạn muốn sử dụng; hãy xem "save" ở trên.
* store: Lưu một stash đã được tạo thông qua git stash create (là một commit merge lơ lửng) trong ref stash, cập nhật stash reflog. Điều này được thiết kế để hữu ích cho các script. Đây có lẽ không phải là lệnh bạn muốn sử dụng; hãy xem "save" ở trên.

### Mục 26.1: Lưu Tạm Là Gì?

Khi làm việc trên một dự án, bạn có thể đang làm dở dang một thay đổi trên nhánh tính năng khi một lỗi được báo cáo trên nhánh master. Bạn chưa sẵn sàng commit mã của mình, nhưng cũng không muốn mất các thay đổi của mình. Đây là lúc git stash trở nên hữu ích.

Chạy git status trên một nhánh để hiển thị các thay đổi chưa được commit của bạn:

sh

Sao chép mã

(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")

Sau đó chạy git stash để lưu các thay đổi này vào một stack:

sh

Sao chép mã

(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

Nếu bạn đã thêm tệp vào thư mục làm việc của mình, bạn cũng có thể lưu trữ chúng. Bạn chỉ cần đưa chúng vào giai đoạn đầu tiên.

sh

Sao chép mã

(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) $

Thư mục làm việc của bạn bây giờ đã sạch sẽ khỏi mọi thay đổi mà bạn đã thực hiện. Bạn có thể thấy điều này bằng cách chạy lại git status:

sh

Sao chép mã

(master) $ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

Để áp dụng stash cuối cùng, chạy git stash apply (thêm nữa, bạn có thể áp dụng và loại bỏ các thay đổi được lưu trữ cuối cùng với git stash pop):

sh

Sao chép mã

(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")

Lưu ý rằng việc lưu trữ không ghi nhớ nhánh mà bạn đang làm việc. Trong các ví dụ trên, người dùng đã lưu trữ trên nhánh master. Nếu họ chuyển sang nhánh dev và chạy git stash apply, stash cuối cùng sẽ được đặt trên nhánh dev.

sh

Sao chép mã

(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")

### Mục 26.2: Tạo stash

Lưu trạng thái hiện tại của thư mục làm việc và chỉ mục (còn được gọi là khu vực staging) vào một ngăn xếp stash.

sh

Sao chép mã

git stash

Để bao gồm tất cả các tệp chưa được theo dõi vào stash, sử dụng các cờ --include-untracked hoặc -u.

sh

Sao chép mã

git stash --include-untracked

Để thêm thông báo vào stash của bạn để dễ dàng nhận diện sau này:

sh

Sao chép mã

git stash save "<thông báo bất kỳ>"

Để giữ nguyên trạng thái khu vực staging sau khi stash, sử dụng các cờ --keep-index hoặc -k.

sh

Sao chép mã

git stash --keep-index

### Mục 26.3: Áp dụng và loại bỏ stash

Để áp dụng stash cuối cùng và loại bỏ nó khỏi ngăn xếp:

sh

Sao chép mã

git stash pop

Để áp dụng stash cụ thể và loại bỏ nó khỏi ngăn xếp:

sh

Sao chép mã

git stash pop stash@{n}

### Mục 26.4: Áp dụng stash mà không loại bỏ nó

Áp dụng stash cuối cùng mà không loại bỏ nó khỏi ngăn xếp:

sh

Sao chép mã

git stash apply

Hoặc một stash cụ thể:

sh

Sao chép mã

git stash apply stash@{n}

### Mục 26.5: Hiển thị stash

Hiển thị các thay đổi đã lưu trong stash cuối cùng:

sh

Sao chép mã

git stash show

Hoặc một stash cụ thể:

sh

Sao chép mã

git stash show stash@{n}

Để hiển thị nội dung của các thay đổi đã lưu cho stash cụ thể:

sh

Sao chép mã

git stash show -p stash@{n}

### Mục 26.6: Lưu tạm một phần

Nếu bạn muốn chỉ lưu tạm một số khác biệt trong tập hợp làm việc của mình, bạn có thể sử dụng lưu tạm một phần.

sh

Sao chép mã

git stash -p

Và sau đó tương tác chọn những hunk để lưu tạm.

Từ phiên bản 2.13.0, bạn cũng có thể tránh chế độ tương tác và tạo một stash một phần với một pathspec sử dụng từ khóa push mới.

sh

Sao chép mã

git stash push -m "Lưu tạm một phần của tôi" -- app.config

### Mục 26.7: Liệt kê các stash đã lưu

sh

Sao chép mã

git stash list

Điều này sẽ liệt kê tất cả các stash trong ngăn xếp theo thứ tự thời gian ngược lại.

Bạn sẽ nhận được một danh sách trông giống như sau:

sh

Sao chép mã

stash@{0}: WIP on master: 67a4e01 Merge tests into develop

stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login

Bạn có thể tham chiếu một stash cụ thể bằng tên của nó, ví dụ như stash@{1}.

### Mục 26.8: Di chuyển công việc đang thực hiện sang nhánh khác

Nếu trong khi làm việc bạn nhận ra rằng bạn đang ở nhánh sai và bạn chưa tạo bất kỳ commit nào, bạn có thể dễ dàng di chuyển công việc của mình sang nhánh đúng bằng cách sử dụng lưu tạm:

sh

Sao chép mã

git stash

git checkout correct-branch

git stash pop

Nhớ rằng git stash pop sẽ áp dụng stash cuối cùng và xóa nó khỏi danh sách stash. Để giữ stash trong danh sách và chỉ áp dụng cho một nhánh, bạn có thể sử dụng:

sh

Sao chép mã

git stash apply

### Mục 26.9: Loại bỏ stash

Loại bỏ tất cả các stash:

sh

Sao chép mã

git stash clear

Loại bỏ stash cuối cùng:

sh

Sao chép mã

git stash drop

Hoặc một stash cụ thể:

sh

Sao chép mã

git stash drop stash@{n}

### Mục 26.10: Áp dụng một phần stash bằng checkout

Bạn đã tạo một stash và muốn checkout chỉ một số tệp trong stash đó.

sh

Sao chép mã

git checkout stash@{0} -- myfile.txt

### Mục 26.11: Khôi phục các thay đổi trước đó từ stash

Để áp dụng stash mới nhất sau khi chạy git stash, sử dụng:

sh

Sao chép mã

git stash apply

Để xem danh sách các stash của bạn, sử dụng:

sh

Sao chép mã

git stash list

Bạn sẽ nhận được một danh sách trông giống như sau:

sh

Sao chép mã

stash@{0}: WIP on master: 67a4e01 Merge tests into develop

stash@{1}: WIP on master: 70f0d95 Add user role to localStorage on user login

Chọn một stash khác để khôi phục với số hiển thị cho stash bạn muốn:

sh

Sao chép mã

git stash apply stash@{2}

### Mục 26.12: Lưu tạm tương tác

Lưu tạm (stashing) lấy trạng thái bẩn của thư mục làm việc của bạn - tức là các tệp đã được sửa đổi và các thay đổi đã được staged - và lưu chúng vào một ngăn xếp các thay đổi chưa hoàn thành mà bạn có thể áp dụng lại bất cứ lúc nào.

**Lưu tạm chỉ các tệp đã sửa đổi:** Giả sử bạn không muốn lưu tạm các tệp đã staged và chỉ lưu tạm các tệp đã sửa đổi, bạn có thể sử dụng:

sh

Sao chép mã

git stash --keep-index

Điều này sẽ chỉ lưu tạm các tệp đã sửa đổi.

**Lưu tạm các tệp chưa được theo dõi:** Lưu tạm không bao giờ lưu các tệp chưa được theo dõi, nó chỉ lưu tạm các tệp đã sửa đổi và staged. Vì vậy, giả sử bạn cần lưu tạm các tệp chưa được theo dõi, bạn có thể sử dụng:

sh

Sao chép mã

git stash -u

Điều này sẽ lưu tạm các tệp chưa được theo dõi, staged và đã sửa đổi.

**Lưu tạm một số thay đổi cụ thể:** Giả sử bạn cần lưu tạm chỉ một phần của mã từ tệp hoặc chỉ một số tệp từ tất cả các tệp đã sửa đổi và staged, bạn có thể làm như sau:

sh

Sao chép mã

git stash --patch

Git sẽ không lưu tạm tất cả những gì đã sửa đổi mà thay vào đó sẽ nhắc bạn tương tác xem bạn muốn lưu tạm thay đổi nào và giữ lại thay đổi nào trong thư mục làm việc của bạn.

### Mục 26.13: Khôi phục một stash đã bị xóa

Nếu bạn chỉ vừa mới pop nó và terminal vẫn mở, bạn vẫn sẽ có giá trị hash được in bởi git stash pop trên màn hình:

sh

Sao chép mã

$ git stash pop

[...]

Dropped refs/stash@{0} (2ca03e22256be97f9e40f08e6d6773c7d41dbfd1)

(Lưu ý rằng git stash drop cũng tạo ra dòng tương tự.)

Nếu không, bạn có thể tìm nó bằng cách sử dụng lệnh này:

sh

Sao chép mã

git fsck --no-reflog | awk '/dangling commit/ {print $3}'

Điều này sẽ hiển thị tất cả các commit ở đầu biểu đồ commit của bạn mà không còn được tham chiếu từ bất kỳ nhánh hoặc tag nào - mọi commit bị mất, bao gồm cả mọi commit stash bạn đã từng tạo, sẽ ở đâu đó trong biểu đồ đó.

Cách dễ nhất để tìm commit stash bạn muốn có lẽ là chuyển danh sách đó cho gitk:

sh

Sao chép mã

gitk --all $( git fsck --no-reflog | awk '/dangling commit/ {print $3}' )

Điều này sẽ khởi chạy một trình duyệt kho lưu trữ hiển thị cho bạn mọi commit trong kho lưu trữ từng có, bất kể liệu nó có thể truy cập được hay không.

Bạn có thể thay thế gitk bằng cái gì đó như git log --graph --oneline --decorate nếu bạn thích một biểu đồ đẹp trên console thay vì một ứng dụng GUI riêng biệt.

Để xác định commit stash, hãy tìm các thông báo commit có dạng:

sh

Sao chép mã

WIP on somebranch: commithash Some old commit message

Khi bạn biết hash của commit bạn muốn, bạn có thể áp dụng nó như một stash:

sh

Sao chép mã

git stash apply sh\_hash

Hoặc bạn có thể sử dụng menu ngữ cảnh trong gitk để tạo các nhánh cho bất kỳ commit không thể truy cập nào mà bạn quan tâm. Sau đó, bạn có thể làm bất cứ điều gì bạn muốn với chúng bằng tất cả các công cụ thông thường. Khi bạn hoàn thành, chỉ cần xóa các nhánh đó đi.

## Chương 27: Subtrees

### Mục 27.1: Tạo, Kéo và Chuyển Ngược Subtree

#### Tạo Subtree

Thêm một remote mới có tên là plugin trỏ tới kho lưu trữ của plugin:

sh

Sao chép mã

git remote add plugin https://path.to/remotes/plugin.git

Sau đó tạo một subtree chỉ định tiền tố thư mục mới plugins/demo. plugin là tên remote, và master đề cập đến nhánh master trên kho lưu trữ của subtree:

sh

Sao chép mã

git subtree add --prefix=plugins/demo plugin master

#### Kéo Cập Nhật Subtree

Kéo các commit bình thường được thực hiện trong plugin:

sh

Sao chép mã

git subtree pull --prefix=plugins/demo plugin master

#### Chuyển Ngược Cập Nhật Subtree

1. Chỉ định các commit đã được thực hiện trong superproject để chuyển ngược:

sh

Sao chép mã

git commit -am "new changes to be backported"

1. Checkout nhánh mới để hợp nhất, thiết lập để theo dõi kho lưu trữ của subtree:

sh

Sao chép mã

git checkout -b backport plugin/master

1. Cherry-pick các commit để chuyển ngược:

sh

Sao chép mã

git cherry-pick -x --strategy=subtree master

1. Đẩy các thay đổi trở lại nguồn plugin:

sh

Sao chép mã

git push plugin backport:master

## Chương 28: Đổi Tên

Thông số chi tiết

* -f hoặc --force: Buộc đổi tên hoặc di chuyển tệp ngay cả khi mục tiêu đã tồn tại.

### Mục 28.1: Đổi Tên Thư Mục

Để đổi tên một thư mục từ oldName thành newName, sử dụng lệnh:

sh

Sao chép mã

git mv directoryToFolder/oldName directoryToFolder/newName

Sau đó thực hiện git commit và/hoặc git push.

Nếu gặp lỗi sau:

sh

Sao chép mã

fatal: renaming 'directoryToFolder/oldName' failed: Invalid argument

Hãy sử dụng lệnh sau:

sh

Sao chép mã

git mv directoryToFolder/oldName temp && git mv temp directoryToFolder/newName

### Mục 28.2: Đổi Tên Nhánh Cục Bộ và Nhánh Trên Remote

Cách dễ nhất là kiểm tra nhánh cục bộ đã được checkout:

sh

Sao chép mã

git checkout old\_branch

Sau đó đổi tên nhánh cục bộ, xóa nhánh cũ trên remote và thiết lập nhánh mới được đổi tên là upstream:

sh

Sao chép mã

git branch -m new\_branch

git push origin :old\_branch

git push --set-upstream origin new\_branch

### Mục 28.3: Đổi Tên Nhánh Cục Bộ

Bạn có thể đổi tên nhánh trong kho lưu trữ cục bộ bằng cách sử dụng lệnh sau:

sh

Sao chép mã

git branch -m old\_name new\_name

## Chương 29: Đẩy (Pushing)

Thông số chi tiết

* --force: Ghi đè tham chiếu từ xa để khớp với tham chiếu cục bộ của bạn. Có thể khiến kho lưu trữ từ xa mất các commit, vì vậy hãy cẩn thận khi sử dụng.
* --verbose: Chạy một cách chi tiết.
* <remote>: Kho lưu trữ từ xa là đích đến của thao tác đẩy.
* <refspec>...: Chỉ định tham chiếu từ xa nào cần cập nhật bằng tham chiếu hoặc đối tượng cục bộ nào.

Sau khi thay đổi, staging và commit mã với Git, việc đẩy là cần thiết để làm cho các thay đổi của bạn có sẵn cho người khác và chuyển các thay đổi cục bộ của bạn lên máy chủ kho lưu trữ. Chủ đề này sẽ đề cập đến cách đẩy mã đúng cách sử dụng Git.

### Mục 29.1: Đẩy một đối tượng cụ thể lên nhánh từ xa

**Cú pháp tổng quát**

sh

Sao chép mã

git push <remotename> <object>:<remotebranchname>

**Ví dụ**

sh

Sao chép mã

git push origin master:wip-yourname

Sẽ đẩy nhánh master của bạn lên nhánh wip-yourname của origin (thường là kho lưu trữ bạn đã clone từ).

**Xóa nhánh từ xa** Xóa nhánh từ xa là tương đương với việc đẩy một đối tượng trống lên nó.

sh

Sao chép mã

git push <remotename> :<remotebranchname>

**Ví dụ**

sh

Sao chép mã

git push origin :wip-yourname

Sẽ xóa nhánh từ xa wip-yourname.

Thay vì sử dụng dấu hai chấm, bạn cũng có thể sử dụng cờ --delete, điều này đôi khi dễ đọc hơn. **Ví dụ**

sh

Sao chép mã

git push origin --delete wip-yourname

**Đẩy một commit duy nhất** Nếu bạn có một commit duy nhất trong nhánh của bạn mà bạn muốn đẩy lên từ xa mà không đẩy bất cứ gì khác, bạn có thể sử dụng lệnh sau:

sh

Sao chép mã

git push <remotename> <commit SHA>:<remotebranchname>

**Ví dụ** Giả sử lịch sử git như sau:

r

Sao chép mã

eeb32bc Commit 1 - đã đẩy

347d700 Commit 2 - muốn đẩy

e539af8 Commit 3 - chỉ có cục bộ

5d339db Commit 4 - chỉ có cục bộ

Để đẩy chỉ commit 347d700 lên nhánh từ xa master, sử dụng lệnh sau:

sh

Sao chép mã

git push origin 347d700:master

### Mục 29.2: Đẩy

sh

Sao chép mã

git push

sẽ đẩy mã của bạn lên upstream hiện có. Tùy thuộc vào cấu hình đẩy, nó sẽ đẩy mã từ nhánh hiện tại của bạn (mặc định trong Git 2.x) hoặc từ tất cả các nhánh (mặc định trong Git 1.x).

**Chỉ định kho lưu trữ từ xa** Khi làm việc với git, có thể hữu ích khi có nhiều kho lưu trữ từ xa. Để chỉ định một kho lưu trữ từ xa để đẩy tới, chỉ cần thêm tên của nó vào lệnh.

sh

Sao chép mã

git push origin

**Chỉ định nhánh** Để đẩy tới một nhánh cụ thể, ví dụ feature\_x:

sh

Sao chép mã

git push origin feature\_x

**Thiết lập nhánh theo dõi từ xa** Trừ khi nhánh bạn đang làm việc ban đầu đến từ một kho lưu trữ từ xa, việc chỉ sử dụng git push sẽ không hoạt động lần đầu tiên. Bạn phải thực hiện lệnh sau để nói với git đẩy nhánh hiện tại tới một kết hợp remote/branch cụ thể:

sh

Sao chép mã

git push --set-upstream origin master

Ở đây, master là tên nhánh trên remote origin. Bạn có thể sử dụng -u như một lối tắt cho --set-upstream.

**Đẩy tới một kho lưu trữ mới** Để đẩy tới một kho lưu trữ mà bạn chưa tạo hoặc còn trống:

1. Tạo kho lưu trữ trên GitHub (nếu có thể)
2. Sao chép url được cung cấp cho bạn, dạng https://github.com/USERNAME/REPO\_NAME.git
3. Đi tới kho lưu trữ cục bộ của bạn, và thực hiện git remote add origin URL Để xác minh nó đã được thêm vào, chạy git remote -v
4. Chạy git push origin master Mã của bạn sẽ hiện trên GitHub.

Để biết thêm thông tin, xem Thêm một kho lưu trữ từ xa.

**Giải thích** Đẩy mã nghĩa là git sẽ phân tích sự khác biệt của các commit cục bộ của bạn và từ xa và gửi chúng để được ghi vào upstream. Khi đẩy thành công, kho lưu trữ cục bộ và kho lưu trữ từ xa của bạn sẽ được đồng bộ và người dùng khác có thể xem các commit của bạn.

### Mục 29.3: Đẩy Buộc (Force Pushing)

Đôi khi, khi bạn có các thay đổi cục bộ không tương thích với các thay đổi từ xa (ví dụ, khi bạn không thể fast-forward nhánh từ xa, hoặc nhánh từ xa không phải là tổ tiên trực tiếp của nhánh cục bộ của bạn), cách duy nhất để đẩy các thay đổi của bạn là đẩy buộc.

sh

Sao chép mã

git push -f

hoặc

sh

Sao chép mã

git push --force

**Lưu ý quan trọng** Điều này sẽ ghi đè bất kỳ thay đổi từ xa nào và remote của bạn sẽ khớp với cục bộ của bạn.

**Chú ý**: Sử dụng lệnh này có thể khiến kho lưu trữ từ xa mất các commit. Hơn nữa, việc đẩy buộc không được khuyến khích nếu bạn đang chia sẻ kho lưu trữ từ xa này với người khác, vì lịch sử của họ sẽ giữ lại mọi commit bị ghi đè, do đó làm cho công việc của họ không đồng bộ với kho lưu trữ từ xa.

Theo nguyên tắc chung, chỉ đẩy buộc khi:

* Không có ai khác ngoài bạn đã kéo các thay đổi bạn đang cố gắng ghi đè.
* Bạn có thể buộc mọi người sao chép một bản sao mới sau khi đẩy buộc và làm cho mọi người áp dụng các thay đổi của họ vào đó (mọi người có thể ghét bạn vì điều này).

### Mục 29.4: Đẩy các tag

sh

Sao chép mã

git push --tags

Đẩy tất cả các tag trong kho lưu trữ cục bộ chưa có trong kho lưu trữ từ xa.

### Mục 29.5: Thay đổi hành vi đẩy mặc định

* current cập nhật nhánh trên kho lưu trữ từ xa chia sẻ tên với nhánh làm việc hiện tại.

sh

Sao chép mã

git config push.default current

* simple đẩy tới nhánh upstream, nhưng sẽ không hoạt động nếu nhánh upstream có tên khác.

sh

Sao chép mã

git config push.default simple

* upstream đẩy tới nhánh upstream, bất kể nó có tên gì.

sh

Sao chép mã

git config push.default upstream

* matching đẩy tất cả các nhánh khớp trên cục bộ và từ xa.

sh

Sao chép mã

git config push.default matching

Sau khi bạn đã thiết lập kiểu ưu tiên, sử dụng:

sh

Sao chép mã

git push

để cập nhật kho lưu trữ từ xa.

Top of Form

Bottom of Form

## Chương 30: Nội bộ

### Mục 30.1: Kho lưu trữ

Một kho lưu trữ Git là một cấu trúc dữ liệu trên đĩa lưu trữ metadata cho một tập hợp các tệp và thư mục. Nó nằm trong thư mục .git/ của dự án của bạn. Mỗi lần bạn commit dữ liệu vào Git, nó được lưu trữ ở đây. Ngược lại, .git/ chứa mọi commit đơn lẻ. Cấu trúc cơ bản của nó như thế này:

Sao chép mã

.git/

objects/

refs/

### Mục 30.2: Đối tượng

Git về cơ bản là một kho lưu trữ key-value. Khi bạn thêm dữ liệu vào Git, nó xây dựng một đối tượng và sử dụng hàm băm SHA-1 của nội dung đối tượng làm khóa. Do đó, bất kỳ nội dung nào trong Git cũng có thể được tra cứu bằng hàm băm của nó:

sh

Sao chép mã

git cat-file -p 4bb6f98

Có 4 loại đối tượng:

* blob
* tree
* commit
* tag

### Mục 30.3: Tham chiếu HEAD

HEAD là một tham chiếu đặc biệt. Nó luôn trỏ đến đối tượng hiện tại. Bạn có thể xem nó đang trỏ vào đâu bằng cách kiểm tra tệp .git/HEAD. Thông thường, HEAD trỏ đến một tham chiếu khác:

sh

Sao chép mã

$ cat .git/HEAD

ref: refs/heads/mainline

Nhưng nó cũng có thể trỏ trực tiếp đến một đối tượng:

sh

Sao chép mã

$ cat .git/HEAD

4bb6f98a223abc9345a0cef9200562333

Đây là cái được gọi là "detached head" - vì HEAD không gắn vào (trỏ vào) bất kỳ tham chiếu nào, mà thay vào đó trỏ trực tiếp đến một đối tượng.

### Mục 30.4: Tham chiếu

Một tham chiếu về cơ bản là một con trỏ. Đó là một tên trỏ đến một đối tượng. Ví dụ,

lua

Sao chép mã

"master" --> 1a410e...

Chúng được lưu trữ trong .git/refs/heads/ dưới dạng các tệp văn bản thuần túy.

sh

Sao chép mã

$ cat .git/refs/heads/mainline

4bb6f98a223abc9345a0cef9200562333

Đây là những gì thường được gọi là nhánh. Tuy nhiên, bạn sẽ thấy rằng trong Git không có khái niệm về nhánh - chỉ có tham chiếu. Bây giờ, có thể điều hướng Git hoàn toàn bằng cách nhảy đến các đối tượng khác nhau trực tiếp bằng hàm băm của chúng. Nhưng điều này sẽ rất bất tiện. Một tham chiếu cung cấp cho bạn một tên tiện lợi để tham chiếu đến các đối tượng. Sẽ dễ dàng hơn nhiều để yêu cầu Git đi đến một vị trí cụ thể bằng tên hơn là bằng hàm băm.

### Mục 30.5: Đối tượng Commit

Một commit có lẽ là loại đối tượng quen thuộc nhất với người dùng Git, vì đó là những gì họ thường tạo ra với các lệnh git commit. Tuy nhiên, commit không chứa trực tiếp bất kỳ tệp hoặc dữ liệu thay đổi nào. Thay vào đó, nó chứa chủ yếu là metadata và các con trỏ đến các đối tượng khác chứa nội dung thực sự của commit. Một commit chứa một số thứ:

* hàm băm của một tree
* hàm băm của commit cha
* tên/email của tác giả, tên/email của người commit
* thông điệp commit Bạn có thể xem nội dung của bất kỳ commit nào như thế này:

sh

Sao chép mã

$ git cat-file commit 5bac93

tree 04d1daef...

parent b7850ef5...

author Geddy Lee <glee@rush.com>

commiter Neil Peart <npeart@rush.com>

First commit!

**Tree** Một ghi chú rất quan trọng là các đối tượng tree lưu trữ MỌI tệp trong dự án của bạn, và nó lưu trữ toàn bộ tệp không phải là các bản diffs. Điều này có nghĩa là mỗi commit chứa một snapshot của toàn bộ dự án\*.

\*Về mặt kỹ thuật, chỉ các tệp thay đổi mới được lưu trữ. Nhưng đây chỉ là một chi tiết triển khai để tăng hiệu quả. Từ quan điểm thiết kế, một commit nên được coi là chứa một bản sao hoàn chỉnh của dự án.

**Parent** Dòng parent chứa hàm băm của một đối tượng commit khác, và có thể được coi là một "con trỏ cha" trỏ đến "commit trước đó". Điều này ngầm tạo thành một đồ thị các commit được gọi là đồ thị commit. Cụ thể, nó là một đồ thị có hướng không tuần hoàn (DAG).

### Mục 30.6: Đối tượng Tree

Một tree cơ bản đại diện cho một thư mục trong hệ thống tệp truyền thống: các container lồng nhau chứa tệp hoặc thư mục khác. Một tree chứa:

* 0 hoặc nhiều đối tượng blob
* 0 hoặc nhiều đối tượng tree

Giống như bạn có thể sử dụng ls hoặc dir để liệt kê nội dung của một thư mục, bạn có thể liệt kê nội dung của một đối tượng tree.

sh

Sao chép mã

$ git cat-file -p 07b1a631

100644 blob b91bba1b .gitignore

100644 blob cc0956f1 Makefile

040000 tree 92e1ca7e src

...

Bạn có thể tra cứu các tệp trong một commit bằng cách tìm hàm băm của tree trong commit đó, sau đó xem tree đó:

sh

Sao chép mã

$ 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

...

### Mục 30.7: Đối tượng Blob

Một blob chứa nội dung tệp nhị phân bất kỳ. Thường thì đó sẽ là văn bản thô như mã nguồn hoặc bài viết blog. Nhưng nó cũng có thể dễ dàng là các byte của một tệp PNG hoặc bất kỳ thứ gì khác.

Nếu bạn có hàm băm của một blob, bạn có thể xem nội dung của nó.

sh

Sao chép mã

$ git cat-file -p d429810

package com.example.project

class Foo {

...

}

...

Ví dụ, bạn có thể duyệt qua một tree như trên, sau đó xem một trong các blob trong đó.

sh

Sao chép mã

$ 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

...

### Mục 30.8: Tạo commit mới

Lệnh git commit thực hiện một số việc:

1. Tạo các blob và tree để đại diện cho thư mục dự án của bạn - được lưu trữ trong .git/objects.
2. Tạo một đối tượng commit mới với thông tin tác giả, thông điệp commit và tree gốc từ bước 1 - cũng được lưu trữ trong .git/objects.
3. Cập nhật tham chiếu HEAD trong .git/HEAD tới hàm băm của commit mới tạo.

Điều này dẫn đến một snapshot mới của dự án của bạn được thêm vào Git, kết nối với trạng thái trước đó.

### Mục 30.9: Di chuyển HEAD

Khi bạn chạy git checkout trên một commit (được chỉ định bằng hàm băm hoặc tham chiếu), bạn đang yêu cầu Git làm cho thư mục làm việc của bạn trông giống như khi snapshot được chụp.

1. Cập nhật các tệp trong thư mục làm việc để khớp với tree bên trong commit.
2. Cập nhật HEAD để trỏ đến hàm băm hoặc tham chiếu được chỉ định.

### Mục 30.10: Di chuyển các tham chiếu

Chạy git reset --hard sẽ di chuyển các tham chiếu đến hàm băm/ tham chiếu được chỉ định. Di chuyển MyBranch tới b8dc53:

sh

Sao chép mã

$ git checkout MyBranch # di chuyển HEAD đến MyBranch

$ git reset --hard b8dc53 # làm cho MyBranch trỏ đến b8dc53

### Mục 30.11: Tạo tham chiếu mới

Chạy git checkout -b <refname> sẽ tạo một tham chiếu mới trỏ đến commit hiện tại.

sh

Sao chép mã

$ cat .git/HEAD

1f324a

$ git checkout -b TestBranch

$ cat .git/refs/heads/TestBranch

1f324a

Top of Form

Bottom of Form

## Chương 31: git-tfs

### Mục 31.1: git-tfs clone

Lệnh này sẽ tạo một thư mục cùng tên với dự án, ví dụ: /My.Project.Name

sh

Sao chép mã

$ git tfs clone http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name

### Mục 31.2: git-tfs clone từ kho git bare

Cloning từ kho git nhanh hơn gấp mười lần so với cloning trực tiếp từ TFVS và hoạt động tốt trong môi trường nhóm. Ít nhất một thành viên trong nhóm sẽ phải tạo kho git bare bằng cách thực hiện git-tfs clone trước. Sau đó, kho mới có thể được khởi tạo để làm việc với TFVS.

sh

Sao chép mã

$ git clone x:/fileshare/git/My.Project.Name.git

$ cd My.Project.Name

$ git tfs bootstrap

$ git tfs pull

### Mục 31.3: git-tfs cài đặt qua Chocolatey

Các bước sau giả định bạn sẽ sử dụng kdiff3 để so sánh tệp và mặc dù không bắt buộc nhưng đó là một ý tưởng tốt.

sh

Sao chép mã

C:\> choco install kdiff3

Git có thể được cài đặt trước để bạn có thể chỉ định bất kỳ tham số nào bạn muốn. Ở đây, tất cả các công cụ Unix cũng được cài đặt và 'NoAutoCrlf' có nghĩa là kiểm tra như hiện tại, cam kết như hiện tại.

sh

Sao chép mã

C:\> choco install git -params '"/GitAndUnixToolsOnPath /NoAutoCrlf"'

Đây là tất cả những gì bạn thực sự cần để cài đặt git-tfs qua chocolatey.

sh

Sao chép mã

C:\> choco install git-tfs

### Mục 31.4: git-tfs Check In

Khởi chạy hộp thoại Check In cho TFVS.

sh

Sao chép mã

$ git tfs checkintool

Lệnh này sẽ lấy tất cả các cam kết cục bộ của bạn và tạo một check-in duy nhất.

### Mục 31.5: git-tfs push

Đẩy tất cả các cam kết cục bộ đến TFVS remote.

sh

Sao chép mã

$ git tfs rcheckin

Lưu ý: lệnh này sẽ thất bại nếu cần có Check-in Notes. Bạn có thể bỏ qua yêu cầu này bằng cách thêm git-tfs-force: rcheckin vào thông điệp cam kết.

## Chương 32: Thư mục trống trong Git

### Mục 32.1: Git không theo dõi thư mục

Giả sử bạn đã khởi tạo một dự án với cấu trúc thư mục như sau:

bash

Sao chép mã

/build

app.js

Sau đó bạn thêm tất cả những gì đã tạo và cam kết:

sh

Sao chép mã

git init

git add .

git commit -m "Initial commit"

Git sẽ chỉ theo dõi tệp app.js.

Giả sử bạn đã thêm một bước build vào ứng dụng của mình và dựa vào thư mục "build" để có nó làm thư mục đầu ra (và bạn không muốn biến nó thành một hướng dẫn thiết lập mà mỗi nhà phát triển phải tuân theo), một quy ước là bao gồm một tệp ".gitkeep" bên trong thư mục và để Git theo dõi tệp đó.

bash

Sao chép mã

/build

.gitkeep

app.js

Sau đó thêm tệp mới này:

sh

Sao chép mã

git add build/.gitkeep

git commit -m "Keep the build directory around"

Git sẽ theo dõi tệp build/.gitkeep và do đó thư mục build sẽ được giữ lại khi checkout. Lưu ý rằng đây chỉ là một quy ước và không phải là một tính năng của Git.

## Chương 33: git-svn

### Mục 33.1: Cloning kho SVN

Bạn cần tạo một bản sao cục bộ mới của kho lưu trữ bằng lệnh:

sh

Sao chép mã

git svn clone SVN\_REPO\_ROOT\_URL [DEST\_FOLDER\_PATH] -T TRUNK\_REPO\_PATH -t TAGS\_REPO\_PATH -b BRANCHES\_REPO\_PATH

Nếu kho SVN của bạn theo cấu trúc tiêu chuẩn (các thư mục trunk, branches, tags) bạn có thể tiết kiệm gõ phím:

sh

Sao chép mã

git svn clone -s SVN\_REPO\_ROOT\_URL [DEST\_FOLDER\_PATH]

Lệnh git svn clone sẽ kiểm tra từng revision của SVN, từng cái một, và tạo một commit git trong kho lưu trữ cục bộ của bạn để tái tạo lịch sử. Nếu kho SVN có nhiều commit, việc này sẽ mất một thời gian.

Khi lệnh hoàn thành, bạn sẽ có một kho git đầy đủ với một nhánh cục bộ gọi là master theo dõi nhánh trunk trong kho SVN.

### Mục 33.2: Đẩy các thay đổi cục bộ lên SVN

Lệnh:

sh

Sao chép mã

git svn dcommit

sẽ tạo một revision SVN cho mỗi commit git cục bộ của bạn. Như với SVN, lịch sử git cục bộ của bạn phải đồng bộ với các thay đổi mới nhất trong kho SVN, vì vậy nếu lệnh thất bại, hãy thử thực hiện git svn rebase trước.

### Mục 33.3: Làm việc cục bộ

Chỉ cần sử dụng kho git cục bộ của bạn như một kho git bình thường, với các lệnh git bình thường:

* git add FILE và git checkout -- FILE để đưa/lấy file ra khỏi khu vực chuẩn bị
* git commit để lưu các thay đổi của bạn. Các commit này sẽ là cục bộ và sẽ không được "đẩy" lên kho SVN, giống như trong một kho git bình thường
* git stash và git stash pop cho phép sử dụng stashes
* git reset HEAD --hard để hoàn nguyên tất cả các thay đổi cục bộ của bạn
* git log để truy cập toàn bộ lịch sử trong kho lưu trữ
* git rebase -i để bạn có thể tự do viết lại lịch sử cục bộ của mình
* git branch và git checkout để tạo các nhánh cục bộ

Như tài liệu git-svn đã nêu "Subversion là một hệ thống kém tinh vi hơn nhiều so với Git" vì vậy bạn không thể sử dụng toàn bộ sức mạnh của git mà không làm rối lịch sử trong máy chủ Subversion. May mắn thay, các quy tắc rất đơn giản: Giữ lịch sử tuyến tính.

Điều này có nghĩa là bạn có thể thực hiện hầu hết các thao tác git: tạo nhánh, xóa/sắp xếp lại/gộp commit, di chuyển lịch sử xung quanh, xóa commit, v.v. Bất cứ điều gì ngoại trừ merge. Nếu bạn cần tích hợp lịch sử của các nhánh cục bộ, hãy sử dụng git rebase.

Khi bạn thực hiện một merge, một merge commit được tạo ra. Đặc điểm của merge commit là nó có hai parent, và điều đó làm cho lịch sử không tuyến tính. Lịch sử không tuyến tính sẽ gây nhầm lẫn cho SVN trong trường hợp bạn "đẩy" một merge commit lên kho lưu trữ.

Tuy nhiên, đừng lo lắng: bạn sẽ không phá vỡ bất cứ điều gì nếu bạn "đẩy" một merge commit git lên SVN. Nếu bạn làm như vậy, khi merge commit git được gửi lên máy chủ svn, nó sẽ chứa tất cả các thay đổi của tất cả các commit cho merge đó, vì vậy bạn sẽ mất lịch sử của những commit đó, nhưng không mất các thay đổi trong mã của bạn.

### Mục 33.4: Nhận các thay đổi mới nhất từ SVN

Tương đương với git pull là lệnh:

sh

Sao chép mã

git svn rebase

Lệnh này truy xuất tất cả các thay đổi từ kho SVN và áp dụng chúng trên các commit cục bộ của bạn trong nhánh hiện tại.

Bạn cũng có thể sử dụng lệnh:

sh

Sao chép mã

git svn fetch

để truy xuất các thay đổi từ kho SVN và đưa chúng về máy cục bộ của bạn mà không áp dụng chúng vào nhánh cục bộ của bạn.

### Mục 33.5: Xử lý các thư mục trống

Git không nhận ra khái niệm về thư mục, nó chỉ làm việc với các tệp và đường dẫn của chúng. Điều này có nghĩa là git không theo dõi các thư mục trống. Tuy nhiên, SVN thì có. Sử dụng git-svn có nghĩa là, theo mặc định, bất kỳ thay đổi nào bạn thực hiện liên quan đến các thư mục trống bằng git sẽ không được chuyển đến SVN.

Sử dụng flag --rmdir khi đưa ra lệnh sẽ sửa vấn đề này, và xóa một thư mục trống trong SVN nếu bạn xóa tệp cuối cùng bên trong nó cục bộ:

sh

Sao chép mã

git svn dcommit --rmdir

Đáng tiếc là lệnh này không xóa các thư mục trống hiện có: bạn cần phải làm điều đó thủ công.

Để tránh việc thêm flag này mỗi khi bạn thực hiện dcommit, hoặc để an toàn hơn nếu bạn đang sử dụng công cụ git GUI (như SourceTree), bạn có thể đặt hành vi này làm mặc định với lệnh:

sh

Sao chép mã

git config --global svn.rmdir true

Lệnh này sẽ thay đổi tệp .gitconfig của bạn và thêm các dòng sau:

sh

Sao chép mã

[svn]

rmdir = true

Để xóa tất cả các tệp và thư mục không được theo dõi mà nên được giữ trống cho SVN, sử dụng lệnh git:

sh

Sao chép mã

git clean -fd

Lưu ý: lệnh trước sẽ xóa tất cả các tệp và thư mục không được theo dõi, kể cả những tệp và thư mục nên được SVN theo dõi! Nếu bạn cần tạo lại các thư mục trống được SVN theo dõi, hãy sử dụng lệnh:

sh

Sao chép mã

git svn mkdirs

Trên thực tế, điều này có nghĩa là nếu bạn muốn dọn dẹp không gian làm việc của mình khỏi các tệp và thư mục không được theo dõi, bạn nên luôn sử dụng cả hai lệnh để tạo lại các thư mục trống được SVN theo dõi:

sh

Sao chép mã

git clean -fd && git svn mkdirs

## Chương 34: Tạo Bản Lưu Trữ

Thông số

* **--format=<fmt>**: Định dạng của bản lưu trữ: tar hoặc zip. Nếu không có tùy chọn này và tệp đầu ra được chỉ định, định dạng sẽ được suy ra từ tên tệp nếu có thể. Nếu không, mặc định là tar.
* **-l, --list**: Hiển thị tất cả các định dạng có sẵn.
* **-v, --verbose**: Báo cáo tiến trình tới stderr.
* **--prefix=<prefix>/**: Thêm <prefix>/ vào mỗi tên tệp trong bản lưu trữ.
* **-o <file>, --output=<file>**: Ghi bản lưu trữ vào <file> thay vì stdout.
* **--worktree-attributes**: Tìm các thuộc tính trong các tệp .gitattributes trong cây làm việc.
* **<extra>**: Đây có thể là bất kỳ tùy chọn nào mà backend của archiver hiểu. Đối với backend zip, sử dụng -0 sẽ lưu trữ các tệp mà không nén chúng, trong khi -1 đến -9 có thể được sử dụng để điều chỉnh tốc độ và tỷ lệ nén.
* **--remote=<repo>**: Lấy một bản lưu trữ tar từ một kho lưu trữ từ xa <repo> thay vì kho lưu trữ cục bộ.
* **--exec=<git-upload-archive>**: Sử dụng với --remote để chỉ định đường dẫn đến <git-upload-archive> trên máy từ xa.
* **<tree-ish>**: Cây hoặc commit để tạo một bản lưu trữ.
* **<path>**: Nếu không có tham số tùy chọn, tất cả các tệp và thư mục trong thư mục làm việc hiện tại được bao gồm trong bản lưu trữ. Nếu một hoặc nhiều đường dẫn được chỉ định, chỉ những đường dẫn này được bao gồm.

### Mục 34.1: Tạo một bản lưu trữ của kho git

Với git archive, có thể tạo các bản lưu trữ nén của một kho lưu trữ, ví dụ để phân phối các phiên bản phát hành.

Tạo một bản lưu trữ tar của revision HEAD hiện tại:

sh

Sao chép mã

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

Tạo một bản lưu trữ tar của revision HEAD hiện tại với nén gzip:

sh

Sao chép mã

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

Điều này cũng có thể thực hiện với (sử dụng xử lý tar.gz tích hợp):

sh

Sao chép mã

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

Tạo một bản lưu trữ zip của revision HEAD hiện tại:

sh

Sao chép mã

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

Ngoài ra, có thể chỉ định một tệp đầu ra với phần mở rộng hợp lệ và định dạng và kiểu nén sẽ được suy ra từ nó:

sh

Sao chép mã

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

### Mục 34.2: Tạo một bản lưu trữ của kho git với tiền tố thư mục

Được coi là thực hành tốt để sử dụng tiền tố khi tạo các bản lưu trữ git, để việc giải nén sẽ đặt tất cả các tệp vào một thư mục. Để tạo một bản lưu trữ của HEAD với tiền tố thư mục:

sh

Sao chép mã

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

Khi được giải nén, tất cả các tệp sẽ được giải nén vào trong một thư mục tên là src-directory-name trong thư mục hiện tại.

### Mục 34.3: Tạo bản lưu trữ của kho git dựa trên nhánh, revision, tag hoặc thư mục cụ thể

Cũng có thể tạo các bản lưu trữ của các mục khác ngoài HEAD, chẳng hạn như các nhánh, commit, tag và thư mục.

Để tạo một bản lưu trữ của một nhánh cục bộ dev:

sh

Sao chép mã

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

Để tạo một bản lưu trữ của một nhánh từ xa origin/dev:

sh

Sao chép mã

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

Để tạo một bản lưu trữ của một tag v.01:

sh

Sao chép mã

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

Tạo một bản lưu trữ của các tệp trong một thư mục con cụ thể (sub-dir) của revision HEAD:

sh

Sao chép mã

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

## Chương 35: Viết lại lịch sử với filterbranch

### Mục 35.1: Thay đổi tác giả của các commit

Bạn có thể sử dụng một bộ lọc môi trường để thay đổi tác giả của các commit. Chỉ cần sửa đổi và xuất $GIT\_AUTHOR\_NAME trong script để thay đổi ai là tác giả của commit.

Tạo một file filter.sh với nội dung như sau:

sh

Sao chép mã

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

Sau đó chạy filter-branch từ dòng lệnh:

sh

Sao chép mã

chmod +x ./filter.sh

git filter-branch --env-filter ./filter.sh

### Mục 35.2: Đặt git committer bằng với commit author

Lệnh này, với một phạm vi commit từ commit1..commit2, viết lại lịch sử sao cho tác giả commit của git cũng trở thành người commit của git:

sh

Sao chép mã

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

## Chương 36: Di chuyển sang Git

### Mục 36.1: SubGit

SubGit có thể được sử dụng để thực hiện một lần nhập từ kho lưu trữ SVN sang git.

sh

Sao chép mã

$ subgit import --non-interactive --svn-url http://svn.my.co/repos/myproject myproject.git

### Mục 36.2: Di chuyển từ SVN sang Git bằng tiện ích chuyển đổi của Atlassian

Tải xuống tiện ích chuyển đổi của Atlassian tại đây. Tiện ích này yêu cầu Java, vì vậy hãy đảm bảo rằng bạn đã cài đặt Java Runtime Environment (JRE) trên máy bạn dự định thực hiện chuyển đổi.

Sử dụng lệnh java -jar svn-migration-scripts.jar verify để kiểm tra xem máy của bạn có thiếu bất kỳ chương trình nào cần thiết để hoàn thành việc chuyển đổi không. Lệnh này kiểm tra các tiện ích Git, subversion và git-svn. Nó cũng xác minh rằng bạn đang thực hiện di chuyển trên một hệ thống tệp có phân biệt chữ hoa chữ thường. Việc di chuyển sang Git nên được thực hiện trên một hệ thống tệp có phân biệt chữ hoa chữ thường để tránh làm hỏng kho lưu trữ.

Tiếp theo, bạn cần tạo một tệp tác giả. Subversion theo dõi các thay đổi bằng tên người dùng của người commit. Git, tuy nhiên, sử dụng hai thông tin để phân biệt một người dùng: tên thật và địa chỉ email. Lệnh sau sẽ tạo một tệp văn bản ánh xạ tên người dùng subversion thành tương đương của họ trong Git:

sh

Sao chép mã

java -jar svn-migration-scripts.jar authors <svn-repo> authors.txt

Trong đó <svn-repo> là URL của kho lưu trữ subversion mà bạn muốn chuyển đổi. Sau khi chạy lệnh này, thông tin nhận dạng của những người đóng góp sẽ được ánh xạ trong tệp authors.txt. Địa chỉ email sẽ có dạng <username>@mycompany.com. Trong tệp tác giả, bạn sẽ cần phải thay đổi tên mặc định của mỗi người (mà theo mặc định đã trở thành tên người dùng của họ) thành tên thực của họ. Hãy chắc chắn kiểm tra tất cả các địa chỉ email để đảm bảo chúng đúng trước khi tiến hành.

Lệnh sau sẽ clone một kho lưu trữ SVN thành một kho lưu trữ Git:

sh

Sao chép mã

git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>

Trong đó <svn-repo> là URL của kho lưu trữ được sử dụng ở trên và <git-repo-name> là tên thư mục trong thư mục hiện tại để clone kho lưu trữ vào. Có một số điều cần xem xét trước khi sử dụng lệnh này:

* Cờ --stdlayout từ trên cho Git biết rằng bạn đang sử dụng một bố cục tiêu chuẩn với các thư mục trunk, branches, và tags. Các kho lưu trữ subversion với bố cục không tiêu chuẩn yêu cầu bạn phải chỉ định vị trí của thư mục trunk, bất kỳ/all các thư mục branch, và thư mục tags. Điều này có thể được thực hiện bằng cách làm theo ví dụ này: git svn clone --trunk=/trunk --branches=/branches --branches=/bugfixes --tags=/tags --authorsfile=authors.txt <svn-repo> <git-repo-name>.

Lệnh git svn clone nhập các nhánh subversion (và trunk) dưới dạng các nhánh từ xa bao gồm cả các tag subversion (các nhánh từ xa có tiền tố tags/). Để chuyển chúng thành các nhánh và tag thực tế, hãy chạy các lệnh sau trên máy Linux theo thứ tự chúng được cung cấp. Sau khi chạy chúng, git branch -a nên hiển thị các tên nhánh chính xác và git tag -l nên hiển thị các tag của kho lưu trữ.

sh

Sao chép mã

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

Quá trình chuyển đổi từ svn sang Git đã hoàn tất! Chỉ cần đẩy kho lưu trữ cục bộ của bạn lên một máy chủ và bạn có thể tiếp tục đóng góp sử dụng Git cũng như có một lịch sử phiên bản hoàn chỉnh từ svn.

### Mục 36.3: Di chuyển từ Mercurial sang Git

Có thể sử dụng các phương pháp sau để nhập một kho lưu trữ Mercurial vào Git:

1. Sử dụng fast export:

sh

Sao chép mã

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

1. Sử dụng Hg-Git: Một câu trả lời rất chi tiết ở đây: [Stack Overflow](https://stackoverflow.com/a/31827990/5283213)
2. Sử dụng Trình nhập của GitHub: Thực hiện theo các hướng dẫn chi tiết tại GitHub.

### Mục 36.4: Di chuyển từ Team Foundation Version Control (TFVC) sang Git

Bạn có thể di chuyển từ kiểm soát phiên bản của Team Foundation sang git bằng cách sử dụng một công cụ nguồn mở gọi là Git-TF. Quá trình di chuyển cũng sẽ chuyển lịch sử hiện có của bạn bằng cách chuyển đổi các lần checkin của tfs thành các commit của git.

Để đưa giải pháp của bạn vào Git bằng cách sử dụng Git-TF, hãy thực hiện các bước sau:

* Tải xuống Git-TF
* Bạn có thể tải xuống (và cài đặt) Git-TF từ Codeplex: Git-TF @ Codeplex
* Clone giải pháp TFVC của bạn
* Khởi chạy powershell (win) và gõ lệnh:

sh

Sao chép mã

git-tf clone http://my.tfs.server.address:port/tfs/mycollection '$/myproject/mybranch/mysolution' --deep

Cờ --deep là từ khóa cần lưu ý vì nó cho Git-Tf biết để sao chép lịch sử checkin của bạn. Bây giờ bạn đã có một kho lưu trữ git cục bộ trong thư mục từ đó bạn gọi lệnh clone của bạn.

Cleanup

* Thêm một tệp .gitignore. Nếu bạn đang sử dụng Visual Studio, trình soạn thảo có thể làm điều này cho bạn, nếu không bạn có thể làm điều này thủ công bằng cách tải xuống một tệp hoàn chỉnh từ github/gitignore.
* Xóa bỏ các ràng buộc kiểm soát nguồn TFS khỏi giải pháp (xóa tất cả các tệp \*.vssscc). Bạn cũng có thể sửa đổi tệp giải pháp của mình bằng cách xóa phần GlobalSection(TeamFoundationVersionControl)......EndGlobalSection.
* Commit & Push

Hoàn thành chuyển đổi của bạn bằng cách commit và đẩy kho lưu trữ cục bộ của bạn lên kho lưu trữ từ xa.

sh

Sao chép mã

git add .

git commit -a -m "Converted solution source control from TFVC to Git"

git remote add origin https://my.remote/project/repo.git

git push origin master

### Mục 36.5: Di chuyển từ SVN sang Git bằng cách sử dụng svn2git

svn2git là một công cụ bọc Ruby xung quanh hỗ trợ SVN gốc của git thông qua git-svn, giúp bạn di chuyển các dự án từ Subversion sang Git, giữ lại lịch sử (bao gồm lịch sử trunk, tags và branches).

Ví dụ: Để di chuyển một kho lưu trữ svn với bố cục tiêu chuẩn (tức là branches, tags và trunk ở cấp gốc của kho lưu trữ):

sh

Sao chép mã

$ svn2git http://svn.example.com/path/to/repo

Để di chuyển một kho lưu trữ svn không nằm trong bố cục tiêu chuẩn:

sh

Sao chép mã

$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches branches-dir

Trong trường hợp bạn không muốn di chuyển (hoặc không có) branches, tags hoặc trunk, bạn có thể sử dụng các tùy chọn --notrunk, --nobranches, và --notags. Ví dụ:

sh

Sao chép mã

$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches

sẽ chỉ di chuyển lịch sử của trunk.

Để giảm không gian cần thiết cho kho lưu trữ mới của bạn, bạn có thể muốn loại trừ bất kỳ thư mục hoặc tệp nào bạn đã thêm vào mà không nên (ví dụ: thư mục build hoặc các tệp lưu trữ):

sh

Sao chép mã

$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.\*\.zip$'

Tối ưu hóa sau di chuyển

Nếu bạn đã có vài nghìn commit (hoặc nhiều hơn) trong kho lưu trữ git mới tạo của mình, bạn có thể muốn giảm không gian sử dụng trước khi đẩy kho lưu trữ của mình lên từ xa. Điều này có thể được thực hiện bằng cách sử dụng lệnh sau:

sh

Sao chép mã

$ git gc --aggressive

Lưu ý: Lệnh trước có thể mất đến vài giờ trên các kho lưu trữ lớn (hàng chục nghìn commit và/hoặc hàng trăm megabyte lịch sử).

## Chương 37: Hiển thị

### Mục 37.1: Tổng quan

git show hiển thị các đối tượng Git khác nhau.

Đối với các commit: Hiển thị thông báo commit và diff của các thay đổi được giới thiệu.

| **Lệnh** | **Mô tả** |
| --- | --- |
| git show | hiển thị commit trước đó |
| git show @~3 | hiển thị commit thứ 3 từ cuối cùng |

Đối với cây và blobs: Hiển thị cây hoặc blob.

| **Lệnh** | **Mô tả** |
| --- | --- |
| git show @~3: | hiển thị thư mục gốc của dự án như nó đã tồn tại 3 commit trước (một cây) |
| git show @~3:src/program.js | hiển thị src/program.js như nó đã tồn tại 3 commit trước (một blob) |
| git show @:a.txt @:b.txt | hiển thị a.txt được ghép với b.txt từ commit hiện tại |

Đối với thẻ (tags): Hiển thị thông báo thẻ và đối tượng được tham chiếu.

## Chương 38: Giải quyết xung đột merge

#### Mục 38.1: Giải quyết thủ công

Trong quá trình thực hiện git merge, bạn có thể thấy git báo lỗi "merge conflict". Nó sẽ báo cho bạn biết những tệp nào có xung đột và bạn sẽ cần phải giải quyết xung đột.

Một lệnh git status tại bất kỳ thời điểm nào sẽ giúp bạn thấy những gì vẫn cần chỉnh sửa với một thông báo hữu ích như:

sh

Sao chép mã

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 để lại các đánh dấu trong các tệp để cho bạn biết nơi xung đột xảy ra:

html

Sao chép mã

<<<<<<<<< HEAD: index.html <!-- chỉ trạng thái của nhánh hiện tại -->

<div id="footer">contact : email@somedomain.com</div>

========= <!-- chỉ ngăn cách giữa các xung đột -->

<div id="footer">

please contact us at email@somedomain.com

</div>

>>>>>>>>> iss2: index.html <!-- chỉ trạng thái của nhánh khác (iss2) -->

Để giải quyết xung đột, bạn phải chỉnh sửa khu vực giữa các đánh dấu <<<<<< và >>>>>> một cách phù hợp, xóa hoàn toàn các dòng trạng thái (<<<<<<, >>>>>>, và ========). Sau đó git add index.html để đánh dấu đã giải quyết và git commit để hoàn tất quá trình merge.

## Chương 39: Gói

#### Mục 39.1: Tạo một gói git trên máy cục bộ và sử dụng nó trên một máy khác

Đôi khi bạn có thể muốn duy trì các phiên bản của một kho lưu trữ git trên các máy không có kết nối mạng. Gói cho phép bạn đóng gói các đối tượng và tham chiếu git trong một kho lưu trữ trên một máy và nhập chúng vào một kho lưu trữ trên máy khác.

sh

Sao chép mã

git tag 2016\_07\_24

git bundle create changes\_between\_tags.bundle [some\_previous\_tag]..2016\_07\_24

Chuyển file changes\_between\_tags.bundle đến máy từ xa bằng cách nào đó, ví dụ: qua ổ đĩa USB. Một khi bạn đã có nó ở đó:

sh

Sao chép mã

git bundle verify changes\_between\_tags.bundle # kiểm tra xem gói có bị hỏng không

git checkout [some branch] # trong repo trên máy từ xa

git bundle list-heads changes\_between\_tags.bundle # liệt kê các tham chiếu trong gói

git pull changes\_between\_tags.bundle [reference from the bundle, e.g. last field from the previous output]

Quá trình ngược lại cũng có thể. Một khi bạn đã thực hiện thay đổi trên kho lưu trữ từ xa, bạn có thể đóng gói các delta; đặt các thay đổi lên, ví dụ, một ổ đĩa USB, và gộp chúng lại vào kho lưu trữ cục bộ để hai kho lưu trữ có thể duy trì đồng bộ mà không cần yêu cầu truy cập trực tiếp bằng git, ssh, rsync hoặc http giữa các máy.

## Chương 40: Hiển thị lịch sử commit bằng đồ họa với Gitk

#### Mục 40.1: Hiển thị lịch sử commit cho một tệp tin

sh

Sao chép mã

gitk path/to/myfile

#### Mục 40.2: Hiển thị tất cả các commit giữa hai commit

Giả sử bạn có hai commit d9e1db9 và 5651067 và muốn xem những gì đã xảy ra giữa chúng. d9e1db9 là tổ tiên cũ nhất và 5651067 là hậu duệ cuối cùng trong chuỗi các commit.

sh

Sao chép mã

gitk --ancestry-path d9e1db9 5651067

#### Mục 40.3: Hiển thị các commit kể từ thẻ phiên bản

Nếu bạn có thẻ phiên bản v2.3 bạn có thể hiển thị tất cả các commit kể từ thẻ đó.

sh

Sao chép mã

gitk v2.3..

## Chương 41: Phân tích/Bisecting để tìm commit lỗi

#### Mục 41.1: Tìm kiếm nhị phân (git bisect)

git bisect cho phép bạn tìm commit nào đã giới thiệu một lỗi bằng cách sử dụng tìm kiếm nhị phân.

Bắt đầu bằng cách phân đoạn một phiên bằng cách cung cấp hai tham chiếu commit: một commit tốt trước khi lỗi xuất hiện, và một commit xấu sau khi lỗi xuất hiện. Thông thường, commit xấu là HEAD.

sh

Sao chép mã

# bắt đầu phiên git bisect

$ git bisect start

# cung cấp một commit mà lỗi không tồn tại

$ git bisect good 49c747d

# cung cấp một commit mà lỗi tồn tại

$ git bisect bad HEAD

Git sẽ bắt đầu một tìm kiếm nhị phân: Nó chia phiên bản thành hai phần và chuyển kho lưu trữ đến phiên bản trung gian. Kiểm tra mã để xác định xem phiên bản đó là tốt hay xấu:

sh

Sao chép mã

# nói cho git biết phiên bản là tốt,

# nghĩa là nó không chứa lỗi

$ git bisect good

# nếu phiên bản chứa lỗi,

# thì nói cho git biết nó là xấu

$ git bisect bad

Git sẽ tiếp tục thực hiện tìm kiếm nhị phân trên từng tập hợp con xấu còn lại tùy theo hướng dẫn của bạn. Git sẽ trình bày một phiên bản duy nhất mà, trừ khi các cờ của bạn không chính xác, sẽ đại diện chính xác cho phiên bản nơi lỗi đã được giới thiệu.

Sau đó nhớ chạy git bisect reset để kết thúc phiên bisect và trở về HEAD.

sh

Sao chép mã

$ git bisect reset

Nếu bạn có một script có thể kiểm tra lỗi, bạn có thể tự động hóa quá trình này với:

sh

Sao chép mã

$ git bisect run [script] [arguments]

Trong đó [script] là đường dẫn đến script của bạn và [arguments] là bất kỳ tham số nào nên được truyền cho script của bạn. Chạy lệnh này sẽ tự động chạy qua tìm kiếm nhị phân, thực hiện git bisect good hoặc git bisect bad tại mỗi bước tùy theo mã thoát của script của bạn. Thoát với 0 chỉ ra tốt, trong khi thoát với 1-124, 126 hoặc 127 chỉ ra xấu. 125 chỉ ra rằng script không thể kiểm tra phiên bản đó (điều này sẽ kích hoạt git bisect skip).

#### Mục 41.2: Tìm kiếm commit lỗi bán tự động

Giả sử bạn đang ở trên nhánh master và có một cái gì đó không hoạt động như mong đợi (một hồi quy đã được giới thiệu), nhưng bạn không biết ở đâu. Tất cả những gì bạn biết là, nó đã hoạt động trong lần phát hành cuối cùng (được đánh dấu hoặc bạn biết mã hash của commit, giả sử là old-rel ở đây).

Git có thể giúp bạn tìm commit lỗi đã giới thiệu hồi quy với một số bước rất ít (tìm kiếm nhị phân).

Trước tiên, bắt đầu phân đoạn:

sh

Sao chép mã

git bisect start master old-rel

Điều này sẽ báo cho git rằng master là một phiên bản bị hỏng (hoặc phiên bản bị hỏng đầu tiên) và old-rel là phiên bản cuối cùng đã biết. Git sẽ kiểm tra một đầu tách giữa cả hai commit. Bây giờ, bạn có thể thực hiện kiểm tra của mình. Tùy thuộc vào việc nó hoạt động hay không, phát hành

sh

Sao chép mã

git bisect good

hoặc

sh

Sao chép mã

git bisect bad

Trong trường hợp commit này không thể kiểm tra, bạn có thể dễ dàng reset và kiểm tra commit đó, git sẽ lo liệu điều này.

Sau một vài bước, git sẽ xuất ra mã hash của commit lỗi.

Để hủy bỏ quá trình bisect chỉ cần phát hành

sh

Sao chép mã

git bisect reset

và git sẽ khôi phục trạng thái trước đó.

## Chương 42: Blaming

| **Tham số** | **Chi tiết** |
| --- | --- |
| filename | Tên của tệp tin cần kiểm tra chi tiết |
| -f | Hiển thị tên tệp tin trong commit gốc |
| -e | Hiển thị email tác giả thay vì tên tác giả |
| -w | Bỏ qua khoảng trắng khi so sánh giữa phiên bản con và cha |
| -L start,end | Chỉ hiển thị phạm vi dòng cho trước. Ví dụ: git blame -L 1,2 [filename] |
| --show-stats | Hiển thị thống kê bổ sung ở cuối đầu ra blame |
| -l | Hiển thị rev dài (Mặc định: tắt) |
| -t | Hiển thị dấu thời gian thô (Mặc định: tắt) |
| --reverse | Đi ngược dòng lịch sử thay vì đi ngược lại |
| -p, --porcelain | Đầu ra cho máy móc tiêu thụ |
| -M | Phát hiện các dòng đã di chuyển hoặc sao chép trong một tệp |
| -C | Ngoài -M, phát hiện các dòng đã di chuyển hoặc sao chép từ các tệp khác đã được sửa đổi trong cùng một commit |
| -h | Hiển thị thông báo trợ giúp |
| -c | Sử dụng chế độ đầu ra giống như git-annotate (Mặc định: tắt) |
| -n | Hiển thị số dòng trong commit gốc (Mặc định: tắt) |

#### Mục 42.1: Chỉ hiển thị các dòng nhất định

Đầu ra có thể bị giới hạn bằng cách chỉ định phạm vi dòng như:

sh

Sao chép mã

git blame -L <start>,<end>

Trong đó <start> và <end> có thể là:

* Số dòng

sh

Sao chép mã

git blame -L 10,30

* Biểu thức chính quy (regex)

sh

Sao chép mã

git blame -L /void main/, git blame -L 46,/void foo/

* Offset dương hoặc âm (chỉ dành cho <end>)

sh

Sao chép mã

git blame -L 108,+30, git blame -L 215,-15

Nhiều phạm vi dòng có thể được chỉ định, và các phạm vi chồng chéo được cho phép.

sh

Sao chép mã

git blame -L 10,30 -L 12,80 -L 120,+10 -L ^/void main/,+40

#### Mục 42.2: Để tìm hiểu ai đã thay đổi một tệp

sh

Sao chép mã

// Hiển thị tác giả và commit cho mỗi dòng của tệp tin được chỉ định

git blame test.c

// Hiển thị email tác giả và commit cho mỗi dòng của tệp tin được chỉ định

git blame -e test.c

// Giới hạn lựa chọn các dòng bằng phạm vi chỉ định

git blame -L 1,10 test.c

#### Mục 42.3: Hiển thị commit đã sửa đổi dòng cuối cùng

sh

Sao chép mã

git blame <file>

sẽ hiển thị tệp tin với mỗi dòng được chú thích với commit đã sửa đổi nó lần cuối.

#### Mục 42.4: Bỏ qua các thay đổi chỉ có khoảng trắng

Đôi khi các repo sẽ có commit chỉ điều chỉnh khoảng trắng, ví dụ sửa lỗi thụt đầu dòng hoặc chuyển đổi giữa tab và dấu cách. Điều này làm cho việc tìm commit nơi mã thực sự được viết trở nên khó khăn.

sh

Sao chép mã

git blame -w

sẽ bỏ qua các thay đổi chỉ có khoảng trắng để tìm nơi dòng thực sự đến từ đâu.

## Chương 43: Cú pháp phiên bản Git

#### Mục 43.1: Xác định phiên bản bằng tên đối tượng

sh

Sao chép mã

$ git show dae86e1950b1277e545cee180551750029cfe735

$ git show dae86e19

Bạn có thể chỉ định phiên bản (hoặc bất kỳ đối tượng nào: tag, tree i.e. nội dung thư mục, blob i.e. nội dung tệp tin) bằng cách sử dụng tên đối tượng SHA-1, có thể là chuỗi thập lục phân đầy đủ 40 byte hoặc một chuỗi con duy nhất trong kho lưu trữ.

#### Mục 43.2: Tên ref tượng trưng: nhánh, thẻ, nhánh theo dõi từ xa

sh

Sao chép mã

$ git log master # chỉ định nhánh

$ git show v1.0 # chỉ định thẻ

$ git show HEAD # chỉ định nhánh hiện tại

$ git show origin # chỉ định nhánh theo dõi từ xa mặc định cho remote 'origin'

Bạn có thể chỉ định phiên bản bằng cách sử dụng tên ref tượng trưng, bao gồm các nhánh (ví dụ: 'master', 'next', 'maint'), thẻ (ví dụ: 'v1.0', 'v0.6.3-rc2'), các nhánh theo dõi từ xa (ví dụ: 'origin', 'origin/master') và các ref đặc biệt như 'HEAD' cho nhánh hiện tại. Nếu tên ref tượng trưng bị mơ hồ, ví dụ như bạn có cả nhánh và thẻ tên là 'fix' (không khuyến khích việc có nhánh và thẻ có cùng tên), bạn cần chỉ định loại ref bạn muốn sử dụng:

sh

Sao chép mã

$ git show heads/fix # hoặc 'refs/heads/fix', để chỉ định nhánh

$ git show tags/fix # hoặc 'refs/tags/fix', để chỉ định thẻ

#### Mục 43.3: Phiên bản mặc định: HEAD

sh

Sao chép mã

$ git show # tương đương với 'git show HEAD'

'HEAD' là tên commit mà bạn đã dựa vào để thực hiện các thay đổi trong cây làm việc, và thường là tên tượng trưng cho nhánh hiện tại. Nhiều lệnh (nhưng không phải tất cả) nhận tham số phiên bản mặc định là 'HEAD' nếu nó bị thiếu.

#### Mục 43.4: Tham chiếu Reflog: <refname>@{<n>}

sh

Sao chép mã

$ git show @{1} # sử dụng reflog cho nhánh hiện tại

$ git show master@{1} # sử dụng reflog cho nhánh 'master'

$ git show HEAD@{1} # sử dụng 'HEAD' reflog

Một ref, thường là một nhánh hoặc HEAD, theo sau bởi hậu tố @ với một tham số thứ tự được bao quanh bởi cặp ngoặc nhọn (ví dụ: {1}, {15}) chỉ định giá trị thứ n trước đó của ref trong kho lưu trữ cục bộ của bạn. Bạn có thể kiểm tra các mục nhập reflog gần đây bằng lệnh git reflog, hoặc tùy chọn --walk-reflogs / -g cho lệnh git log.

sh

Sao chép mã

$ 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

Lưu ý: sử dụng reflogs thực tế đã thay thế cơ chế cũ hơn của việc sử dụng ref ORIG\_HEAD (tương đương với HEAD@{1}).

#### Mục 43.5: Tham chiếu Reflog: <refname>@{<date>}

sh

Sao chép mã

$ git show master@{yesterday}

$ git show HEAD@{5 minutes ago} # hoặc HEAD@{5.minutes.ago}

Một ref theo sau bởi hậu tố @ với một tham số ngày được bao quanh bởi cặp ngoặc nhọn (ví dụ: {yesterday}, {1 month 2 weeks 3 days 1 hour 1 second ago} hoặc {1979-02-26 18:30:00}) chỉ định giá trị của ref tại một thời điểm trước đó (hoặc gần nhất với nó). Lưu ý rằng điều này tra cứu trạng thái của ref cục bộ của bạn tại một thời điểm nhất định; ví dụ, những gì đã có trong nhánh 'master' cục bộ của bạn vào tuần trước. Bạn có thể sử dụng git reflog với một tham số ngày để tra cứu thời gian chính xác mà bạn đã làm điều gì đó cho ref cụ thể trong kho lưu trữ cục bộ.

sh

Sao chép mã

$ 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

#### Mục 43.6: Nhánh theo dõi: <branchname>@{upstream}

sh

Sao chép mã

$ git log @{upstream}.. # những gì đã được thực hiện cục bộ và chưa được xuất bản, nhánh hiện tại

$ git show master@{upstream} # hiển thị upstream của nhánh 'master'

Hậu tố @{upstream} được thêm vào một tên nhánh (dạng ngắn <branchname>@{u}) đề cập đến nhánh mà nhánh được chỉ định bởi branchname được đặt để xây dựng trên (được cấu hình với branch.<name>.remote và branch.<name>.merge, hoặc với lệnh git branch --set-upstream-to=<branch>). Một tên nhánh bị thiếu sẽ mặc định là tên hiện tại. Cùng với cú pháp cho phạm vi phiên bản, nó rất hữu ích để xem các commit mà nhánh của bạn đang vượt trước upstream (các commit trong kho lưu trữ cục bộ của bạn chưa có mặt trên upstream), và các commit mà bạn đang chậm hơn (các commit trong upstream chưa được hợp nhất vào nhánh cục bộ), hoặc cả hai:

sh

Sao chép mã

$ git log --oneline @{u}..

$ git log --oneline ..@{u}

$ git log --oneline --left-right @{u}... # tương tự như ...@{u}

#### Mục 43.7: Chuỗi kế thừa commit: <rev>^, <rev>~<n>, v.v.

sh

Sao chép mã

$ git reset --hard HEAD^ # bỏ commit cuối cùng

$ git rebase --interactive HEAD~5 # rebase 4 commit cuối cùng

Một hậu tố ^ cho một tham số phiên bản có nghĩa là cha đầu tiên của đối tượng commit đó. ^<n> có nghĩa là cha thứ <n> (tức là <rev>^ tương đương với <rev>^1). Một hậu tố ~<n> cho một tham số phiên bản có nghĩa là đối tượng commit là tổ tiên thế hệ thứ <n> của đối tượng commit được đặt tên, chỉ theo các cha đầu tiên. Điều này có nghĩa là ví dụ <rev> có nghĩa là <rev>~1, và tương đương với <rev>^1, hoặc <rev>^ ngắn gọn. Cú pháp này có thể kết hợp. Để tìm các tên tượng trưng như vậy, bạn có thể sử dụng lệnh git name-rev:

sh

Sao chép mã

$ git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a

33db5f4d9027a10e477ccf054b2c1ab94f74c85a tags/v0.99~940

Lưu ý rằng --pretty=oneline và không phải --oneline phải được sử dụng trong ví dụ sau

sh

Sao chép mã

$ 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

#### Mục 43.8: Giải tham chiếu nhánh và thẻ: <rev>^0, <rev>^{<type>}

Trong một số trường hợp hành vi của một lệnh phụ thuộc vào việc nó được cung cấp tên nhánh, tên thẻ hoặc một phiên bản tùy ý. Bạn có thể sử dụng cú pháp "giải tham chiếu" nếu bạn cần cái sau. Một hậu tố ^ theo sau bởi tên loại đối tượng (tag, commit, tree, blob) được bao quanh bởi cặp ngoặc nhọn (ví dụ v0.99.8^{commit}) có nghĩa là giải tham chiếu đối tượng tại <rev> đệ quy cho đến khi tìm thấy một đối tượng của loại <type> hoặc đối tượng không thể giải tham chiếu nữa. <rev>^0 là một cách viết tắt cho <rev>^{commit}.

sh

Sao chép mã

$ git checkout HEAD^0 # tương đương với 'git checkout --detach' trong Git hiện đại

Một hậu tố ^ theo sau bởi một cặp ngoặc nhọn trống (ví dụ v0.99.8^{}) có nghĩa là giải tham chiếu thẻ đệ quy cho đến khi tìm thấy một đối tượng không phải thẻ. So sánh

sh

Sao chép mã

$ git show v1.0

$ git cat-file -p v1.0

$ git replace --edit v1.0

với

sh

Sao chép mã

$ git show v1.0^{}

$ git cat-file -p v1.0^{}

$ git replace --edit v1.0^{}

### Mục 43.9: Commit khớp gần nhất: <rev>^{/<text>}, :/<text>

sh

Sao chép mã

$ git show HEAD^{/fix nasty bug} # tìm từ HEAD

$ git show ':/fix nasty bug' # tìm từ bất kỳ nhánh nào

Một dấu hai chấm (':'), theo sau bởi một dấu gạch chéo ('/'), theo sau bởi một đoạn văn bản, sẽ đặt tên cho commit có thông điệp commit khớp với biểu thức chính quy được chỉ định. Tên này trả về commit khớp gần nhất mà có thể truy cập từ bất kỳ ref nào. Biểu thức chính quy có thể khớp với bất kỳ phần nào của thông điệp commit. Để khớp với thông điệp bắt đầu bằng một chuỗi, có thể sử dụng ví dụ: :/^foo. Chuỗi đặc biệt :/! được dành riêng cho các điều chỉnh đối với những gì được khớp. :/!-foo thực hiện khớp phủ định, trong khi :/!!foo khớp với ký tự !, theo sau bởi foo.

Một hậu tố ^ cho một tham số phiên bản, theo sau bởi một cặp ngoặc nhọn chứa văn bản dẫn đầu bởi một dấu gạch chéo, tương đương với cú pháp :/<text> bên dưới mà nó trả về commit khớp gần nhất có thể truy cập từ <rev> trước ^.

Ví dụ minh họa:

sh

Sao chép mã

$ git show HEAD^{/initial commit} # tìm từ HEAD

$ git show ':/added README' # tìm từ bất kỳ nhánh nào

Trong ví dụ này, git show HEAD^{/initial commit} sẽ tìm và hiển thị commit gần nhất từ HEAD có thông điệp commit chứa chuỗi "initial commit". Tương tự, git show ':/added README' sẽ tìm và hiển thị commit gần nhất từ bất kỳ nhánh nào có thông điệp commit chứa chuỗi "added README".

## Chương 44: Worktrees

Thông số chi tiết

* -f, --force: Theo mặc định, add từ chối tạo một worktree mới khi <branch> đã được checkout bởi một worktree khác. Tùy chọn này sẽ bỏ qua biện pháp bảo vệ đó.
* -b <new-branch>, -B <new-branch>: Với add, tạo một nhánh mới tên là <new-branch> bắt đầu từ <branch>, và checkout <new-branch> vào worktree mới. Nếu <branch> bị bỏ qua, nó mặc định là HEAD. Mặc định, -b từ chối tạo một nhánh mới nếu nó đã tồn tại. -B bỏ qua biện pháp bảo vệ này, đặt lại <new-branch> về <branch>.
* --detach: Với add, tách rời HEAD trong worktree mới.
* --[no-]checkout: Theo mặc định, add sẽ checkout <branch>, tuy nhiên, --no-checkout có thể được sử dụng để ngăn chặn checkout nhằm thực hiện các tùy chỉnh, chẳng hạn như cấu hình sparse-checkout.
* -n, --dry-run: Với prune, không xóa bất cứ thứ gì; chỉ báo cáo những gì nó sẽ xóa.
* --porcelain: Với list, xuất ra định dạng dễ phân tích cho các script. Định dạng này sẽ ổn định qua các phiên bản Git và không phụ thuộc vào cấu hình người dùng.
* -v, --verbose: Với prune, báo cáo tất cả các mục bị xóa.
* --expire <time>: Với prune, chỉ hết hạn các worktree không sử dụng cũ hơn <time>.

#### Mục 44.1: Sử dụng một worktree

Bạn đang làm việc giữa chừng trên một tính năng mới, và sếp của bạn yêu cầu bạn sửa ngay một lỗi nào đó. Bạn có thể muốn sử dụng git stash để lưu trữ tạm thời các thay đổi của mình. Tuy nhiên, vào thời điểm này thư mục làm việc của bạn đang ở trạng thái lộn xộn (với các tệp mới, di chuyển và bị xóa, và các phần khác rải rác) và bạn không muốn làm gián đoạn tiến trình của mình.

Bằng cách thêm một worktree, bạn tạo ra một thư mục làm việc liên kết tạm thời để thực hiện sửa lỗi khẩn cấp, xóa nó khi xong việc, và sau đó tiếp tục phiên làm việc trước đó của bạn:

sh

Sao chép mã

$ git worktree add -b emergency-fix ../temp master

$ pushd ../temp

# ... làm việc ...

$ git commit -a -m 'sửa lỗi khẩn cấp cho sếp'

$ popd

$ rm -rf ../temp

$ git worktree prune

LƯU Ý: Trong ví dụ này, bản sửa lỗi vẫn nằm trong nhánh emergency-fix. Tại thời điểm này, bạn có thể muốn git merge hoặc git format-patch và sau đó xóa nhánh emergency-fix.

#### Mục 44.2: Di chuyển một worktree

Hiện tại (phiên bản 2.11.0), Git không có chức năng tích hợp để di chuyển một worktree đã tồn tại. Đây là một lỗi chính thức (xem <https://git-scm.com/docs/git-worktree#_bugs>).

Để vượt qua hạn chế này, bạn có thể thực hiện các thao tác thủ công trực tiếp trong các tệp tham chiếu .git.

Trong ví dụ này, bản sao chính của repo nằm ở /home/user/project-main và worktree thứ cấp nằm ở /home/user/project-1, và chúng ta muốn di chuyển nó đến /home/user/project-2.

Không thực hiện bất kỳ lệnh git nào giữa các bước này, nếu không trình thu gom rác có thể được kích hoạt và các tham chiếu đến cây làm việc thứ cấp có thể bị mất. Thực hiện các bước này từ đầu đến cuối mà không bị gián đoạn:

1. Thay đổi tệp .git của worktree để trỏ đến vị trí mới bên trong cây chính. Tệp /home/user/project-1/.git bây giờ sẽ chứa nội dung sau:

plaintext

Sao chép mã

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

1. Đổi tên worktree bên trong thư mục .git của dự án chính bằng cách di chuyển thư mục của worktree trong đó:

sh

Sao chép mã

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

1. Thay đổi tham chiếu bên trong /home/user/project-main/.git/worktrees/project-2/gitdir để trỏ đến vị trí mới. Trong ví dụ này, tệp sẽ có nội dung sau:

plaintext

Sao chép mã

/home/user/project-2/.git

1. Cuối cùng, di chuyển worktree của bạn đến vị trí mới:

sh

Sao chép mã

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

Nếu bạn đã làm mọi thứ đúng, việc liệt kê các worktree hiện có sẽ chỉ đến vị trí mới: sh $ git worktree list /home/user/project-main 23f78ad [master] /home/user/project-2 78ac3f3 [branch-name]

Bây giờ cũng nên an toàn để chạy git worktree prune.

## Chương 45: Git Remote

#### Mục 45.1: Hiển thị các kho lưu trữ từ xa

Để liệt kê tất cả các kho lưu trữ từ xa được cấu hình, sử dụng git remote. Nó hiển thị tên ngắn (bí danh) của mỗi tay cầm từ xa mà bạn đã cấu hình.

sh

Sao chép mã

$ git remote

premium

premiumPro

origin

Để hiển thị thông tin chi tiết hơn, có thể sử dụng cờ --verbose hoặc -v. Đầu ra sẽ bao gồm URL và loại của remote (push hoặc pull):

sh

Sao chép mã

$ 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)

#### Mục 45.2: Thay đổi URL từ xa của kho lưu trữ Git của bạn

Bạn có thể muốn làm điều này nếu kho lưu trữ từ xa được di chuyển. Lệnh để thay đổi URL từ xa là:

sh

Sao chép mã

git remote set-url

Nó nhận 2 đối số: tên remote hiện có (origin, upstream) và URL.

Kiểm tra URL từ xa hiện tại của bạn:

sh

Sao chép mã

git remote -v

origin https://bitbucket.com/develop/myrepo.git (fetch)

origin https://bitbucket.com/develop/myrepo.git (push)

Thay đổi URL từ xa của bạn:

sh

Sao chép mã

git remote set-url origin https://localserver/develop/myrepo.git

Kiểm tra lại URL từ xa của bạn:

sh

Sao chép mã

git remote -v

origin https://localserver/develop/myrepo.git (fetch)

origin https://localserver/develop/myrepo.git (push)

#### Mục 45.3: Xóa một kho lưu trữ từ xa

Xóa remote có tên <name>. Tất cả các nhánh remote-tracking và cấu hình cài đặt cho remote sẽ bị xóa.

Để xóa một kho lưu trữ từ xa dev:

sh

Sao chép mã

git remote rm dev

#### Mục 45.4: Thêm một kho lưu trữ từ xa

Để thêm một remote, sử dụng git remote add trong thư mục gốc của kho lưu trữ cục bộ của bạn.

Để thêm một kho lưu trữ Git từ xa <url> dưới một tên ngắn dễ dàng <name> sử dụng:

sh

Sao chép mã

git remote add <name> <url>

Lệnh git fetch <name> sau đó có thể được sử dụng để tạo và cập nhật các nhánh remote-tracking <name>/<branch>.

#### Mục 45.5: Hiển thị thông tin chi tiết về kho lưu trữ từ xa

Bạn có thể xem thông tin chi tiết về một kho lưu trữ từ xa bằng cách sử dụng git remote show <remote repository alias>

sh

Sao chép mã

git remote show origin

Kết quả:

plaintext

Sao chép mã

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)

#### Mục 45.6: Đổi tên một kho lưu trữ từ xa

Đổi tên remote từ <old> thành <new>. Tất cả các nhánh remote-tracking và cấu hình cài đặt cho remote sẽ được cập nhật.

Để đổi tên một nhánh remote từ dev thành dev1:

sh

Sao chép mã

git remote rename dev dev1

## Chương 46: Git Large File Storage (LFS)

#### Mục 46.1: Khai báo các loại tệp nhất định để lưu trữ bên ngoài

Một quy trình làm việc phổ biến khi sử dụng Git LFS là khai báo những tệp nào được chặn thông qua hệ thống quy tắc, giống như các tệp .gitignore. Phần lớn thời gian, các ký tự đại diện được sử dụng để chọn các loại tệp nhất định để theo dõi toàn diện. Ví dụ: git lfs track "\*.psd" Khi một tệp khớp với mẫu trên được thêm vào và cam kết, khi nó được đẩy lên từ xa, nó sẽ được tải lên riêng biệt, với một con trỏ thay thế tệp trong kho lưu trữ từ xa. Sau khi một tệp đã được theo dõi với lfs, tệp .gitattributes của bạn sẽ được cập nhật tương ứng. GitHub khuyến nghị cam kết tệp .gitattributes cục bộ của bạn, thay vì làm việc với tệp .gitattributes toàn cầu, để đảm bảo rằng bạn không gặp phải bất kỳ vấn đề nào khi làm việc với các dự án khác nhau.

#### Mục 46.2: Đặt cấu hình LFS cho tất cả các bản sao

Để đặt các tùy chọn LFS áp dụng cho tất cả các bản sao, tạo và cam kết một tệp có tên .lfsconfig tại gốc kho lưu trữ. Tệp này có thể chỉ định các tùy chọn LFS theo cùng cách như được cho phép trong .git/config. Ví dụ, để loại trừ một tệp nhất định khỏi các lần tải xuống LFS theo mặc định, tạo và cam kết .lfsconfig với nội dung sau:

plaintext

Sao chép mã

[lfs]

fetchexclude = ReallyBigFile.wav

#### Mục 46.3: Cài đặt LFS

Tải xuống và cài đặt, thông qua Homebrew hoặc từ trang web. Đối với Brew:

sh

Sao chép mã

brew install git-lfs

git lfs install

Thường bạn cũng cần thực hiện một số cài đặt trên dịch vụ lưu trữ từ xa của bạn để cho phép nó làm việc với lfs. Điều này sẽ khác nhau cho từng dịch vụ lưu trữ, nhưng có thể chỉ cần kiểm tra một hộp đánh dấu rằng bạn muốn sử dụng git lfs.

## Chương 47: Git Patch

Tham số Chi tiết

less

Sao chép mã

(<mbox>|<Maildir>)...

Danh sách các tệp mailbox để đọc các bản vá. Nếu bạn không cung cấp đối số này, lệnh sẽ đọc từ đầu vào chuẩn. Nếu bạn cung cấp thư mục, chúng sẽ được xử lý như Maildirs.

-s, --signoff Thêm một dòng Signed-off-by: vào thông báo commit, sử dụng danh tính của người commit là bạn.

-q, --quiet Giữ im lặng. Chỉ in các thông báo lỗi.

-u, --utf8 Truyền cờ -u cho git mailinfo. Thông báo commit log được đề xuất từ email sẽ được mã hóa lại thành mã hóa UTF-8 (biến cấu hình i18n.commitencoding có thể được sử dụng để chỉ định mã hóa ưa thích của dự án nếu không phải là UTF-8). Bạn có thể sử dụng --no-utf8 để ghi đè điều này.

--no-utf8 Truyền cờ -n cho git mailinfo.

-3, --3way Khi bản vá không áp dụng một cách sạch sẽ, chuyển sang hợp nhất 3 chiều nếu bản vá ghi lại danh tính của các blob mà nó dự định áp dụng và chúng ta có sẵn các blob đó trong cục bộ.

--ignore-date, --ignore-space-change, --ignore-whitespace, --whitespace=<option>, -C<n>, -p<n>, --directory=<dir>, --exclude=<path>, --include=<path>, --reject Các cờ này được truyền cho chương trình git apply để áp dụng bản vá.

--patch-format Mặc định lệnh sẽ cố gắng phát hiện định dạng bản vá tự động. Tùy chọn này cho phép người dùng bỏ qua phát hiện tự động và chỉ định định dạng bản vá mà bản vá (các) nên được hiểu là. Các định dạng hợp lệ là mbox, stgit, stgit-series, và hg.

-i, --interactive Chạy tương tác.

--committer-date-is-author-date Mặc định lệnh sẽ ghi lại ngày từ thông báo email là ngày tác giả commit và sử dụng thời gian tạo commit là ngày của người commit. Điều này cho phép người dùng nói dối về ngày của người commit bằng cách sử dụng cùng một giá trị với ngày của tác giả.

--skip Bỏ qua bản vá hiện tại. Điều này chỉ có ý nghĩa khi khởi động lại một bản vá bị hủy.

-S[<keyid>], --gpg-sign[=<keyid>] Ký GPG các commit.

--continue, -r, --resolved Sau khi thất bại bản vá (ví dụ: cố gắng áp dụng bản vá xung đột), người dùng đã áp dụng nó bằng tay và tệp index lưu trữ kết quả của việc áp dụng. Tạo một commit sử dụng quyền tác giả và thông báo commit được trích xuất từ email và tệp index hiện tại, và tiếp tục.

--resolvemsg=<msg> Khi một bản vá thất bại xảy ra, <msg> sẽ được in ra màn hình trước khi thoát. Điều này ghi đè thông báo chuẩn thông báo bạn sử dụng --continue hoặc --skip để xử lý thất bại. Điều này chỉ dành cho sử dụng nội bộ giữa git rebase và git am.

--abort Khôi phục nhánh gốc và hủy bỏ thao tác vá.

#### Mục 47.1: Tạo bản vá

Để tạo một bản vá, có hai bước:

1. Thực hiện các thay đổi của bạn và commit chúng.
2. Chạy git format-patch <commit-reference> để chuyển đổi tất cả các commit kể từ commit <commit-reference> (không bao gồm nó) thành các tệp bản vá.

Ví dụ, nếu các bản vá nên được tạo từ hai commit gần nhất:

sh

Sao chép mã

git format-patch HEAD~~

Điều này sẽ tạo ra 2 tệp, một cho mỗi commit kể từ HEAD~~, như sau:

plaintext

Sao chép mã

0001-hello\_world.patch

0002-beginning.patch

#### Mục 47.2: Áp dụng bản vá

Chúng ta có thể sử dụng git apply some.patch để áp dụng các thay đổi từ tệp .patch vào thư mục làm việc hiện tại của bạn. Chúng sẽ không được staged và cần phải được commit.

Để áp dụng một bản vá như một commit (với thông báo commit của nó), sử dụng:

sh

Sao chép mã

git am some.patch

Để áp dụng tất cả các tệp bản vá vào cây:

sh

Sao chép mã

git am \*.patch

Top of Form

Bottom of Form

## Chương 48: Thống kê Git

#### Tham số Chi tiết

less

Sao chép mã

-n, --numbered: Sắp xếp đầu ra theo số lượng commit mỗi tác giả thay vì theo thứ tự bảng chữ cái.

-s, --summary: Chỉ cung cấp tóm tắt số lượng commit.

-e, --email: Hiển thị địa chỉ email của mỗi tác giả.

--format[=<format>]: Thay vì tiêu đề commit, sử dụng thông tin khác để mô tả mỗi commit. <format> có thể là bất kỳ chuỗi nào được chấp nhận bởi tùy chọn --format của git log.

-w[<width>[,<indent1>[,<indent2>]]]: Ngắt dòng đầu ra bằng cách ngắt mỗi dòng tại width. Dòng đầu tiên của mỗi mục được thụt vào indent1 số khoảng trắng và các dòng tiếp theo được thụt vào indent2 số khoảng trắng.

<revision range>: Chỉ hiển thị các commit trong phạm vi revision được chỉ định. Mặc định là toàn bộ lịch sử cho đến commit hiện tại.

[--] <path>: Chỉ hiển thị các commit giải thích cách các tệp khớp với path được tạo ra. Các đường dẫn có thể cần được thêm tiền tố "-- " để tách chúng khỏi các tùy chọn hoặc phạm vi revision.

#### Mục 48.1: Dòng mã theo từng nhà phát triển

sh

Sao chép mã

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

#### Mục 48.2: Liệt kê mỗi nhánh và ngày sửa đổi cuối cùng của nó

sh

Sao chép mã

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

#### Mục 48.3: Commits theo từng nhà phát triển

Git shortlog được sử dụng để tóm tắt các đầu ra log của git và nhóm các commit theo tác giả.

sh

Sao chép mã

git shortlog -sn # Tên và Số lượng commit

git shortlog -sne # Tên cùng với địa chỉ email và Số lượng commit

Hoặc

sh

Sao chép mã

git log --pretty=format:%ae | gawk -- '{ ++c[$0]; } END { for(cc in c) printf "%5d %s\n",c[cc],cc; }'

Lưu ý: Các commit của cùng một người có thể không được nhóm lại với nhau nếu tên và/hoặc địa chỉ email của họ được đánh vần khác nhau. Để giải quyết vấn đề này, hãy tham khảo tính năng .mailmap.

#### Mục 48.4: Commits theo ngày

sh

Sao chép mã

git log --pretty=format:"%ai" | awk '{print " : "$1}' | sort -r | uniq -c

#### Mục 48.5: Tổng số commit trong một nhánh

sh

Sao chép mã

git log --pretty=oneline |wc -l

#### Mục 48.6: Liệt kê tất cả các commit ở định dạng đẹp

sh

Sao chép mã

git log --pretty=format:"%Cgreen%ci %Cblue%cn %Cgreen%cr%Creset %s"

Điều này sẽ cung cấp một cái nhìn tổng quan đẹp về tất cả các commit (1 dòng mỗi commit) với ngày, người dùng và thông báo commit. Tùy chọn --pretty có nhiều trình giữ chỗ, mỗi cái bắt đầu bằng %. Tất cả các tùy chọn có thể được tìm thấy [tại đây](https://git-scm.com/docs/git-log#_pretty_formats).

#### Mục 48.7: Tìm tất cả các kho Git cục bộ trên máy tính

Để liệt kê tất cả các vị trí kho git trên máy tính của bạn, bạn có thể chạy lệnh sau:

sh

Sao chép mã

find $HOME -type d -name ".git"

Giả sử bạn có locate, điều này sẽ nhanh hơn nhiều:

sh

Sao chép mã

locate .git |grep git$

Nếu bạn có gnu locate hoặc mlocate, điều này sẽ chỉ chọn các thư mục git:

sh

Sao chép mã

locate -ber \\.git$

#### Mục 48.8: Hiển thị tổng số commit mỗi tác giả

Để có được tổng số commit mà mỗi nhà phát triển hoặc cộng tác viên đã thực hiện trên một kho lưu trữ, bạn có thể đơn giản sử dụng git shortlog:

sh

Sao chép mã

git shortlog -s

Cung cấp tên tác giả và số lượng commit của mỗi người. Ngoài ra, nếu bạn muốn kết quả được tính trên tất cả các nhánh, hãy thêm cờ --all vào lệnh:

sh

Sao chép mã

git shortlog -s --all

## Chương 49: git send-email

#### Mục 49.1: Sử dụng git send-email với Gmail

Nếu bạn làm việc trên một dự án như nhân Linux, thay vì tạo một pull request, bạn sẽ cần gửi các commit của mình đến một listserv để xem xét. Mục này chi tiết cách sử dụng git-send-email với Gmail.

Thêm những dòng sau vào tệp .gitconfig của bạn:

ini

Sao chép mã

[sendemail]

smtpserver = smtp.googlemail.com

smtpencryption = tls

smtpserverport = 587

smtpuser = name@gmail.com

Sau đó trên web: Đi tới Google -> My Account -> Connected Apps & Sites -> Allow less secure apps -> Bật chế độ ON

Để tạo một bộ patch:

sh

Sao chép mã

git format-patch HEAD~~~~ --subject-prefix="PATCH <project-name>"

Sau đó gửi các patch đến listserv:

sh

Sao chép mã

git send-email --annotate --to project-developers-list@listserve.example.com 00\*.patch

Để tạo và gửi phiên bản cập nhật (ví dụ: phiên bản 2) của patch:

sh

Sao chép mã

git format-patch -v 2 HEAD~~~~

git send-email --to project-developers-list@listserve.example.com v2-00\*.patch

#### Mục 49.2: Soạn thảo

Các tham số khi soạn thảo email:

less

Sao chép mã

--from \* Email From:

--[no-]to \* Email To:

--[no-]cc \* Email Cc:

--[no-]bcc \* Email Bcc:

--subject \* Email "Subject:"

--in-reply-to \* Email "In-Reply-To:"

--[no-]xmailer \* Thêm tiêu đề "X-Mailer:" (mặc định là có).

--[no-]annotate \* Xem lại từng patch sẽ được gửi trong trình soạn thảo.

--compose \* Mở trình soạn thảo để viết phần giới thiệu.

--compose-encoding \* Mã hóa giả định cho phần giới thiệu.

--8bit-encoding \* Mã hóa giả định cho các email 8 bit nếu không được khai báo.

--transfer-encoding \* Mã hóa truyền tải để sử dụng (quoted-printable, 8bit, base64).

#### Mục 49.3: Gửi các patch qua email

Giả sử bạn có nhiều commit cho một dự án (ở đây là ulogd2, nhánh chính là git-svn) và bạn muốn gửi bộ patch của mình đến danh sách gửi thư devel@netfilter.org. Để làm điều này, chỉ cần mở một shell tại thư mục gốc của thư mục git và sử dụng:

sh

Sao chép mã

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/

Lệnh đầu tiên sẽ tạo một loạt email từ các patch trong /tmp/ulogd2/ với báo cáo thống kê và lệnh thứ hai sẽ bắt đầu trình soạn thảo của bạn để soạn thảo một email giới thiệu cho bộ patch. Để tránh các chuỗi email luồng không mong muốn, bạn có thể sử dụng:

sh

Sao chép mã

git config sendemail.chainreplyto false

## Chương 50: Ứng dụng GUI Git

#### Mục 50.1: gitk và git-gui

Khi bạn cài đặt Git, bạn cũng sẽ nhận được các công cụ giao diện đồ họa của nó, gitk và git-gui. gitk là một công cụ xem lịch sử đồ họa. Nghĩ về nó như là một GUI mạnh mẽ cho git log và git grep. Đây là công cụ để sử dụng khi bạn muốn tìm kiếm điều gì đó đã xảy ra trong quá khứ hoặc hình dung lịch sử dự án của bạn. Gitk dễ dàng được gọi từ dòng lệnh. Chỉ cần cd vào một kho Git, và gõ:

sh

Sao chép mã

$ gitk [tùy chọn git log]

Gitk chấp nhận nhiều tùy chọn dòng lệnh, hầu hết trong số đó được truyền qua hành động git log cơ bản. Một trong những tùy chọn hữu ích nhất có lẽ là cờ --all, cho gitk hiển thị các commit có thể đạt được từ bất kỳ ref nào, không chỉ HEAD. Giao diện của Gitk trông như thế này:

**Hình 1-1. Trình xem lịch sử gitk** Trên đầu là thứ gì đó trông giống như đầu ra của git log --graph; mỗi chấm đại diện cho một commit, các dòng đại diện cho các mối quan hệ cha con, và các ref được hiển thị như các hộp màu. Chấm màu vàng đại diện cho HEAD, và chấm màu đỏ đại diện cho các thay đổi chưa trở thành commit. Ở dưới cùng là một cái nhìn của commit đã chọn; các nhận xét và bản vá ở bên trái, và một cái nhìn tổng quan bên phải. Ở giữa là một tập hợp các điều khiển được sử dụng để tìm kiếm lịch sử.

Bạn có thể truy cập nhiều chức năng liên quan đến git qua việc nhấp chuột phải vào tên nhánh hoặc tin nhắn commit. Ví dụ như việc kiểm tra một nhánh khác hoặc cherry-pick một commit có thể dễ dàng thực hiện chỉ với một cú nhấp chuột.

git-gui, mặt khác, chủ yếu là một công cụ để tạo ra các commit. Nó cũng dễ dàng được gọi từ dòng lệnh:

sh

Sao chép mã

$ git gui

Và nó trông như thế này:

**Hình 1-2. Công cụ commit git-gui** Bên trái là chỉ mục; các thay đổi chưa được staging ở trên, các thay đổi đã staging ở dưới. Bạn có thể di chuyển toàn bộ tệp giữa hai trạng thái bằng cách nhấp vào biểu tượng của chúng, hoặc bạn có thể chọn một tệp để xem bằng cách nhấp vào tên của nó.

Ở trên cùng bên phải là chế độ xem diff, hiển thị các thay đổi cho tệp được chọn hiện tại. Bạn có thể stage các hunks (hoặc các dòng) riêng lẻ bằng cách nhấp chuột phải trong khu vực này.

Ở dưới cùng bên phải là khu vực tin nhắn và hành động. Gõ tin nhắn của bạn vào hộp văn bản và nhấp vào “Commit” để làm điều tương tự như git commit. Bạn cũng có thể chọn sửa đổi commit cuối cùng bằng cách chọn nút radio “Amend”, điều này sẽ cập nhật khu vực “Staged Changes” với nội dung của commit cuối cùng. Sau đó, bạn có thể đơn giản staging hoặc unstaging một số thay đổi, thay đổi tin nhắn commit, và nhấp vào “Commit” một lần nữa để thay thế commit cũ bằng một commit mới.

gitk và git-gui là các ví dụ về các công cụ hướng nhiệm vụ. Mỗi công cụ được thiết kế cho một mục đích cụ thể (xem lịch sử và tạo commit, tương ứng), và bỏ qua các tính năng không cần thiết cho nhiệm vụ đó.

Nguồn: <https://git-scm.com/book/en/v2/Git-in-Other-Environments-Graphical-Interfaces>

#### Mục 50.2: GitHub Desktop

* Website: <https://desktop.github.com>
* Giá: miễn phí
* Nền tảng: OS X và Windows
* Phát triển bởi: GitHub

#### Mục 50.3: Git Kraken

* Website: <https://www.gitkraken.com>
* Giá: 60 USD/năm (miễn phí cho mã nguồn mở, giáo dục, phi lợi nhuận, startups hoặc sử dụng cá nhân)
* Nền tảng: Linux, OS X, Windows
* Phát triển bởi: Axosoft

#### Mục 50.4: SourceTree

* Website: <https://www.sourcetreeapp.com>
* Giá: miễn phí (cần tài khoản)
* Nền tảng: OS X và Windows
* Phát triển bởi: Atlassian

#### Mục 50.5: Git Extensions

* Website: <https://gitextensions.github.io>
* Giá: miễn phí
* Nền tảng: Windows

#### Mục 50.6: SmartGit

* Website: <http://www.syntevo.com/smartgit/>
* Giá: Miễn phí cho sử dụng phi thương mại. Giấy phép vĩnh viễn có giá 99 USD
* Nền tảng: Linux, OS X, Windows
* Phát triển bởi: syntevo

## Chương 51: Reflog - Khôi phục các commit không hiển thị trong git log

#### Mục 51.1: Khôi phục từ một lần rebase tồi

Giả sử bạn đã bắt đầu một rebase tương tác:

sh

Sao chép mã

git rebase --interactive HEAD~20

và do nhầm lẫn, bạn đã gộp hoặc loại bỏ một số commit mà bạn không muốn mất, nhưng sau đó đã hoàn thành rebase. Để khôi phục, hãy thực hiện git reflog, và bạn có thể thấy một số đầu ra như sau:

css

Sao chép mã

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} ...

...

Trong trường hợp này, commit cuối cùng, ddddddd (hoặc HEAD@{n+1}) là đầu của nhánh trước khi rebase. Do đó, để khôi phục commit đó (và tất cả các commit cha, bao gồm cả những commit đã bị gộp hoặc loại bỏ nhầm), hãy thực hiện:

sh

Sao chép mã

$ git checkout HEAD@{n+1}

Bạn có thể tạo một nhánh mới tại commit đó với git checkout -b [branch]. Xem thêm thông tin về Branching để biết thêm chi tiết.

## Chương 52: TortoiseGit

#### Mục 52.1: Gộp các commit

**Cách dễ dàng** Điều này sẽ không hoạt động nếu có commit hợp nhất trong lựa chọn của bạn.

**Cách nâng cao** Bắt đầu hộp thoại rebase

#### Mục 52.2: Giả định không thay đổi

Nếu một tệp đã thay đổi, nhưng bạn không muốn commit nó, hãy đặt tệp đó là "Giả định không thay đổi"

#### Mục 52.4: Tạo nhánh

Đối với những người sử dụng giao diện người dùng để tạo nhánh, nhấp chuột phải vào kho lưu trữ sau đó chọn Tortoise Git -> Create Branch...

## Chương 53: Công cụ so sánh và hợp nhất bên ngoài

#### Mục 53.1: Thiết lập KDiff3 làm công cụ hợp nhất

Thêm các dòng sau vào tệp .gitconfig toàn cầu của bạn

ini

Sao chép mã

[merge]

tool = kdiff3

[mergetool "kdiff3"]

path = D:/Program Files (x86)/KDiff3/kdiff3.exe

keepBackup = false

keepbackup = false

trustExitCode = false

Hãy nhớ đặt thuộc tính path trỏ tới thư mục nơi bạn đã cài đặt KDiff3.

#### Mục 53.2: Thiết lập KDiff3 làm công cụ so sánh

ini

Sao chép mã

[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\"

#### Mục 53.3: Thiết lập IntelliJ IDE làm công cụ hợp nhất (Windows)

ini

Sao chép mã

[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

Lưu ý: Thuộc tính cmd này không chấp nhận các ký tự lạ trong đường dẫn. Nếu vị trí cài đặt IDE của bạn có các ký tự lạ (ví dụ: nó được cài đặt trong Program Files (x86)), bạn sẽ phải tạo một symlink.

#### Mục 53.4: Thiết lập IntelliJ IDE làm công cụ so sánh (Windows)

ini

Sao chép mã

[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")\"

Lưu ý: Thuộc tính cmd này không chấp nhận các ký tự lạ trong đường dẫn. Nếu vị trí cài đặt IDE của bạn có các ký tự lạ (ví dụ: nó được cài đặt trong Program Files (x86)), bạn sẽ phải tạo một symlink.

#### Mục 53.5: Thiết lập Beyond Compare

Bạn có thể đặt đường dẫn tới bcomp.exe

sh

Sao chép mã

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

và cấu hình bc3 làm mặc định

sh

Sao chép mã

git config --global diff.tool bc3

## Chương 54: Cập nhật tên đối tượng trong tham chiếu

#### Mục 54.1: Cập nhật tên đối tượng trong tham chiếu

#### Sử dụng

Cập nhật tên đối tượng được lưu trữ trong tham chiếu.

#### CÚ PHÁP

sh

Sao chép mã

git update-ref [-m <lý\_do>] (-d <ref> [<giá\_trị\_cũ>] | [--no-deref] [--create-reflog] <ref> <giá\_trị\_mới> [<giá\_trị\_cũ>] | --stdin [-z])

#### Cú pháp chung

1. Tham chiếu biểu tượng, cập nhật nhánh hiện tại đến đối tượng mới.

sh

Sao chép mã

git update-ref HEAD <giá\_trị\_mới>

1. Lưu giá\_trị\_mới trong ref, sau khi xác nhận rằng giá trị hiện tại của ref khớp với giá\_trị\_cũ.

sh

Sao chép mã

git update-ref refs/head/master <giá\_trị\_mới> <giá\_trị\_cũ>

Cú pháp trên cập nhật đầu nhánh master tới giá\_trị\_mới chỉ khi giá trị hiện tại của nó là giá\_trị\_cũ.

* Sử dụng cờ -d để xóa ref được đặt tên sau khi xác minh nó vẫn chứa giá\_trị\_cũ.
* Sử dụng --create-reflog, update-ref sẽ tạo một reflog cho mỗi ref ngay cả khi thông thường sẽ không được tạo.
* Sử dụng cờ -z để chỉ định định dạng kết thúc bằng NUL, bao gồm các giá trị như cập nhật, tạo, xóa, xác minh.

#### Các thao tác

* **Cập nhật**: Đặt ref tới giá\_trị\_mới sau khi xác minh giá\_trị\_cũ, nếu có. Chỉ định giá\_trị\_mới bằng không để đảm bảo ref không tồn tại sau khi cập nhật và/hoặc giá\_trị\_cũ bằng không để đảm bảo ref không tồn tại trước khi cập nhật.
* **Tạo**: Tạo ref với giá\_trị\_mới sau khi xác minh nó không tồn tại. giá\_trị\_mới không thể là không.
* **Xóa**: Xóa ref sau khi xác minh nó tồn tại với giá\_trị\_cũ, nếu có. Nếu có, giá\_trị\_cũ không thể là không.
* **Xác minh**: Xác minh ref so với giá\_trị\_cũ nhưng không thay đổi nó. Nếu giá\_trị\_cũ là không hoặc không có, ref không được tồn tại.

## Chương 55: Tên nhánh Git trên Bash Ubuntu

#### Mục 55.1: Tên nhánh trong terminal

#### PS1 là gì

PS1 là Prompt String 1. Đây là một trong những prompt có sẵn trong shell của Linux/UNIX. Khi bạn mở terminal, nó sẽ hiển thị nội dung được định nghĩa trong biến PS1 trên prompt bash của bạn. Để thêm tên nhánh vào prompt bash, chúng ta phải chỉnh sửa biến PS1 (đặt giá trị của PS1 trong ~/.bash\_profile).

#### Hiển thị tên nhánh git

Thêm các dòng sau vào ~/.bash\_profile của bạn:

sh

Sao chép mã

git\_branch() {

git branch 2> /dev/null | sed -e '/^[^\*]/d' -e 's/\* \(.\*\)/ (\1)/'

}

export PS1="\u@\h \[\033[32m\]\w\[\033[33m\]\$(git\_branch)\[\033[00m\] $ "

Hàm git\_branch này sẽ tìm tên nhánh mà bạn đang làm việc. Sau khi thực hiện các thay đổi này, bạn có thể di chuyển đến repo git trên terminal và sẽ có thể thấy tên nhánh.

## hương 56: Hook phía client của Git

Giống như nhiều Hệ thống Kiểm soát Phiên bản khác, Git có cách để kích hoạt các script tùy chỉnh khi các hành động quan trọng xảy ra. Có hai nhóm hook này: hook phía client và hook phía server. Hook phía client được kích hoạt bởi các thao tác như commit và merge, trong khi hook phía server chạy trên các thao tác mạng như nhận các commit được push. Bạn có thể sử dụng các hook này vì nhiều lý do khác nhau.

#### Mục 56.1: Git pre-push hook

Script pre-push được gọi bởi git push sau khi nó đã kiểm tra trạng thái remote, nhưng trước khi bất cứ điều gì được push. Nếu script này thoát với trạng thái khác 0 thì không có gì sẽ được push.

Hook này được gọi với các tham số sau:

* $1 -- Tên của remote mà push đang được thực hiện (Ví dụ: origin)
* $2 -- URL mà push đang được thực hiện (Ví dụ: https://<url>/.git)

Thông tin về các commit đang được push được cung cấp dưới dạng các dòng vào đầu vào chuẩn theo dạng:

plaintext

Sao chép mã

<local\_ref> <local\_sha1> <remote\_ref> <remote\_sha1>

Các giá trị mẫu:

* local\_ref = refs/heads/master
* local\_sha1 = 68a07ee4f6af8271dc40caae6cc23f283122ed11
* remote\_ref = refs/heads/master
* remote\_sha1 = efd4d512f34b11e3cf5c12433bbedd4b1532716f

Dưới đây là ví dụ về script pre-push được lấy từ pre-push.sample mặc định được tạo tự động khi một repository mới được khởi tạo với git init:

sh

Sao chép mã

# 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

#### Mục 56.2: Cài đặt Hook

Các hook đều được lưu trong thư mục con hooks của thư mục Git. Trong hầu hết các dự án, đó là .git/hooks. Để kích hoạt một script hook, hãy đặt một file trong thư mục hooks của thư mục .git của bạn với tên phù hợp (không có bất kỳ phần mở rộng nào) và có thể thực thi được.

## Chương 57: Git rerere

rerere (reuse recorded resolution) cho phép bạn yêu cầu Git ghi nhớ cách bạn đã giải quyết một xung đột đoạn mã. Điều này cho phép nó tự động được giải quyết lần sau khi Git gặp phải xung đột tương tự.

#### Mục 57.1: Kích hoạt rerere

Để kích hoạt rerere, chạy lệnh sau:

sh

Sao chép mã

$ git config --global rerere.enabled true

Điều này có thể được thực hiện trong một repository cụ thể cũng như toàn cầu.

## Chương 58: Thay đổi tên repository Git

Nếu bạn thay đổi tên repository ở phía remote, chẳng hạn như trên GitHub hoặc Bitbucket, khi bạn push code hiện tại của mình, bạn sẽ thấy lỗi: **Fatal error, repository not found**.

#### Mục 58.1: Thay đổi cài đặt local

Đi đến terminal,

sh

Sao chép mã

cd projectFolder

git remote -v # (nó sẽ hiển thị URL git trước đó)

git remote set-url origin https://username@bitbucket.org/username/newName.git

git remote -v # (kiểm tra lại, nó sẽ hiển thị URL git mới)

git push # thực hiện bất cứ điều gì bạn muốn.

## Chương 59: Tagging trong Git

Giống như hầu hết các Hệ thống Kiểm soát Phiên bản (VCS), Git có khả năng gắn thẻ các điểm cụ thể trong lịch sử để đánh dấu là quan trọng. Thông thường, mọi người sử dụng chức năng này để đánh dấu các điểm phát hành (v1.0, v.v.).

#### Mục 59.1: Liệt kê tất cả các tag có sẵn

Sử dụng lệnh git tag để liệt kê tất cả các tag có sẵn:

sh

Sao chép mã

$ git tag

<output follows> ``` v0.1 v1.3 ``` Lưu ý: các tag được hiển thị theo thứ tự bảng chữ cái.

Bạn cũng có thể tìm kiếm các tag có sẵn:

sh

Sao chép mã

$ 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 ```

#### Mục 59.2: Tạo và đẩy tag(s) trong GIT

Tạo một tag: Để tạo một tag trên nhánh hiện tại của bạn:

sh

Sao chép mã

git tag <tagname>

Điều này sẽ tạo một tag local với trạng thái hiện tại của nhánh bạn đang ở.

Để tạo một tag với một commit cụ thể:

sh

Sao chép mã

git tag tag-name commit-identifier

Điều này sẽ tạo một tag local với commit-identifier của nhánh bạn đang ở.

Đẩy một commit trong GIT:

Đẩy một tag riêng lẻ:

sh

Sao chép mã

git push origin tag-name

Đẩy tất cả các tag cùng một lúc:

sh

Sao chép mã

git push origin --tags

## Chương 60: Dọn dẹp repository local và remote của bạn

#### Mục 60.1: Xóa các nhánh local đã bị xóa trên remote

Để theo dõi giữa các nhánh local và các nhánh remote đã bị xóa, sử dụng

sh

Sao chép mã

git fetch -p

sau đó bạn có thể sử dụng

sh

Sao chép mã

git branch -vv

để xem các nhánh nào không còn được theo dõi nữa.

Các nhánh không còn được theo dõi sẽ có dạng dưới đây, chứa từ 'gone'

plaintext

Sao chép mã

branch 12345e6 [origin/branch: gone] Fixed bug

sau đó bạn có thể sử dụng kết hợp các lệnh trên, tìm kiếm nơi 'git branch -vv' trả về 'gone' rồi sử dụng '-d' để xóa các nhánh

sh

Sao chép mã

git fetch -p && git branch -vv | awk '/: gone]/{print $1}' | xargs git branch -d

## Chương 61: diff-tree

So sánh nội dung và chế độ của các blob được tìm thấy qua hai đối tượng cây.

#### Mục 61.1: Xem các tệp đã thay đổi trong một commit cụ thể

sh

Sao chép mã

git diff-tree --no-commit-id --name-only -r COMMIT\_ID

#### Mục 61.2: Cách sử dụng

sh

Sao chép mã

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

**Giải thích các tùy chọn:**

* -r: so sánh đệ quy
* --root: bao gồm commit ban đầu như một sự khác biệt so với /dev/null

#### Mục 61.3: Các tùy chọn diff thông thường

**Giải thích các tùy chọn:**

* -z: xuất diff-raw với các dòng kết thúc bằng NUL.
* -p: xuất định dạng patch.
* -u: đồng nghĩa với -p.
* --patch-with-raw: xuất cả patch và định dạng diff-raw.
* --stat: hiển thị diffstat thay vì patch.
* --numstat: hiển thị diffstat số thay vì patch.
* --patch-with-stat: xuất một patch và đính kèm diffstat của nó.
* --name-only: chỉ hiển thị tên của các tệp đã thay đổi.
* --name-status: hiển thị tên và trạng thái của các tệp đã thay đổi.
* --full-index: hiển thị tên đối tượng đầy đủ trên các dòng chỉ mục.
* --abbrev=<n>: viết tắt tên đối tượng trong tiêu đề diff-tree và diff-raw.
* -R: hoán đổi các cặp tệp đầu vào.
* -B: phát hiện viết lại hoàn toàn.
* -M: phát hiện đổi tên.
* -C: phát hiện sao chép.
* --find-copies-harder: thử các tệp không thay đổi làm ứng viên cho phát hiện sao chép.
* -l<n>: giới hạn các nỗ lực đổi tên đến các đường dẫn.
* -O: sắp xếp lại các khác biệt theo .
* -S: tìm cặp tệp mà chỉ một bên chứa chuỗi.
* --pickaxe-all: hiển thị tất cả các tệp khác biệt khi -S được sử dụng và có kết quả.
* -a, --text: xử lý tất cả các tệp như văn bản.

Top of Form

Bottom of Form