COURSE NAME

**VERSION CONTROL SYSTEM**

GIT

horizontal line

# Placeholder image

# Introduction

## GIT - Học nghiêm túc một lần

ref: <https://viblo.asia/p/git-hoc-nghiem-tuc-mot-lan-phan-1-OeVKBo6JZkW>

### Đặt vấn đề

* Khi code ta vô tình làm rối tung code trong các file và chương trình bị lỗi, vậy làm sao để hoàn nguyên code về trạng thái cũ => tạo backup => 100 lần sữa -> có 100 bản backup
* Khi tham gia vào trong một dự án, bạn muốn cập nhật sự thay đổi code của thành viên khác một cách nhanh chóng => dùng usb

==> Hai vấn đề trên ta thấy việc thao tác thủ công như vậy quá bất tiện và dễ gây sai sót => thúc đẩy tạo ra công cụ quản lý phiên bản của source code và Hệ thống quản lý phiên bản (Version Control System - VCS) ra đời.

### Lịch sử hoàn thiện VCS

### Git - History

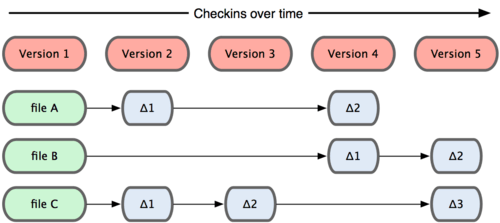
### Git overview

* Nếu như đã từng sử dụng Git thì chắc hẳn lắm lúc chúng ta cũng đã từng phải fix lỗi về Git
* Cho dù là lần đầu học về Git hay đã từng sử dụng, thiết nghĩ rằng bạn nên hiểu rõ hơn bản chất của Git là gì, các nguyên tắc cơ bản và follow làm việc của Git hoạt động như thế nào, thì sẽ sử dụng Git một cách hiệu quả và dễ dàng hơn. Rồi sau đó mới nên đọc tới các lệnh.
* Khi học Git, hãy cố gắng gạt bỏ những kiến thức mà có thể bạn đã biết về các VCS khác, ví dụ như Subversion hay Perforce. Việc này sẽ giúp bạn tránh được sự hỗn độn, bối rối khi sử dụng nó.
* Các lệnh dù rất nhiều nhưng cũng chỉ biến tấu từ những điều cơ bản mà ra. Nắm phần gốc là sẽ làm chủ được phần ngọn, bằng không chúng ta sẽ nhớ câu lệnh khác khổ sở và sau này có gặp bug cũng chẳng biết phải fix kiểu gì.
* Nếu được bạn nên đọc, thực hành và hiểu rõ hết tất cả những điều nói trong các trang About, Document, Community trong trang chủ của Git. Và đọc thêm các blog về Git.

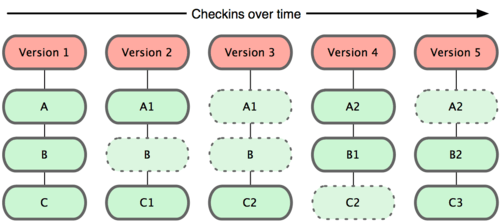
### Git - About

**-> Git Diff**: Điểm khác nhau cơ bản giữa Git với các VCS khác là cách Git "nghĩ" về dữ liệu.

* Phần lớn hệ thống khác lưu thông tin dưới dạng các tập tin được thay đổi, coi thông tin được lưu trữ như là một tập hợp các tập tin và các thay đổi được thực hiện trên mỗi tập tin theo thời gian



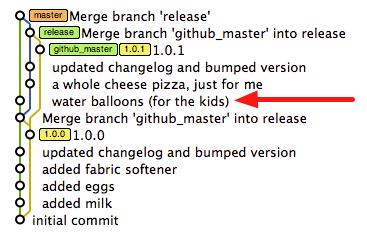
* Còn Git sẽ xem dữ liệu của mình giống như một tập hợp các "ảnh" (snapshot) của một hệ thống tập tin. Mỗi lần bạn lưu lại trạng thái hiện tại của dự án, về cơ bản Git "chụp một bức ảnh" ghi lại thông tin của tất cả các tập tin tại thời điểm đó và tạo ra một tham chiếu tới "ảnh" đó. Để hiệu quả hơn, nếu như tập tin không có sự thay đổi nào, Git không lưu trữ tập tin đó lại một lần nữa mà chỉ tạo một liên kết tới tập tin gốc đã tồn tại trước đó.



Dựa trên thiết kế phần lõi này, Git có những ưu điểm vượt bậc khác

**-> Branching and Merging**

* Thế nào là repository? Repository (hay gọi tắt là repo) được hiểu đơn giản là một kho chứa toàn bộ project bao gồm source code và lịch sử thay đổi cũng như nội dung thay đổi của từng file và từng cá nhân đóng góp cho project đó.
* Git chỉ thêm mới dữ liệu, Dữ liệu của repository sẽ được lưu trong thư mục **.git**. Khi bạn thực hiện các thao tác trong Git, phần lớn các hành động đó đều được thêm vào cơ sở dữ liệu của Git. Rất khó để yêu cầu hệ thống thực hiện một hành động nào đó mà không thể khôi phục lại được hoặc xóa dữ liệu đi dưới mọi hình thức. Giống như trong các VCS khác, bạn có thể mất hoặc làm rối tung dữ liệu mà bạn chưa commit, nhưng khi bạn đã commit thì rất khó để mất các dữ liệu đó, đặc biệt là nếu bạn thường xuyên đẩy (push) cơ sở dữ liệu sang một kho chứa khác. Bạn có thể thoải mái thử nghiệm mà không lo sợ sẽ phá hỏng mọi thứ.
* Repository của Git gồm 2 loại là:
  + Remote repository: Kho này dùng để chia sẻ cho nhiều người và được đặt trên server chuyên dụng.
  + Local repository: Kho này được đặt trên máy của bạn và chỉ dành cho một người mà thôi.
* Branch là nhánh của repository: Các nhánh sẽ độc lập với nhau và phát triển một tính năng, không gây ảnh hưởng đến các nhánh khác. Khi các nhánh hợp nhất lại với nhau thì gọi là merge, nhánh mặc định là master. Branch ở trên local repo thì gọi là local branch. Branch ở trên remote repo thì gọi là remote branch. Một branch trên local có thể liên kết với 1 hoặc nhiều branch trên remote hoặc không branch nào cả.
* Git cho phép và khuyến khích bạn tạo nhiều branch tại máy tính cá nhân của bạn, việc tạo, sáp nhập và xóa bỏ những branch này diễn ra rất nhanh chóng.

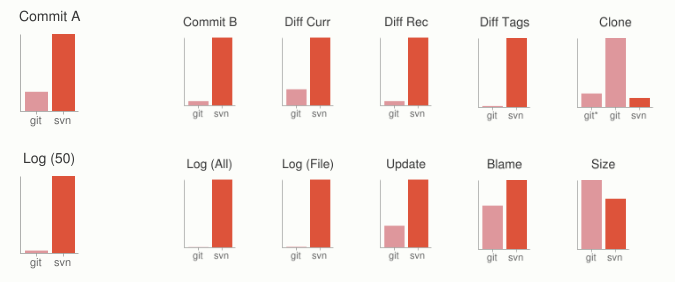


* Thông thường thì branch master chứa phiên bản code chạy ngon nhất để deploy lên production, branch dev để bạn có thể kiểm thử lại chức năng và một vài branch nhỏ để xây dựng các tính năng mới.
* Sau khi viết code để thực hiện chức năng mới ở branch nhỏ xong, bạn hợp nhất (merge) code vào branch dev, kiểm tra lại, nếu ok thì tiếp tục merge code từ dev vào master.
* Việc chuyển đổi trạng thái code từ branch này sang branch khác hoàn toàn dễ dàng, nếu như chức năng mới của bạn bị lỗi, bạn hoàn toàn có thể xóa branch đó đi mà chẳng gây ảnh hưởng những chức năng khác, follow cơ bản là vậy, trong dự án thì có thể tùy biến ít nhiều.

**-> Small and Fast**

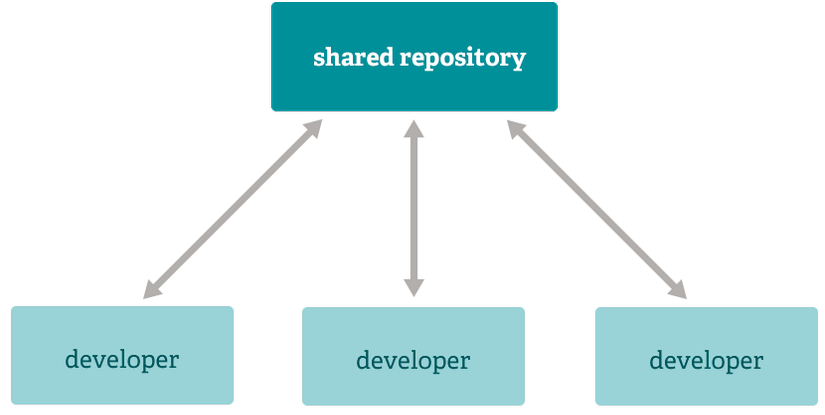
Git chứng tỏ rằng mình rất nhẹ và hoạt động nhanh chóng vì những lý do sau:

* Gần như tất cả các hoạt động được thực hiện tại máy tính cá nhân nên Git có một lợi thế tốc độ rất lớn so với các hệ thống mà liên tục phải giao tiếp với một máy chủ ở đâu đó.
* Git được viết bằng C, làm giảm tổng phí thời gian chạy liên quan đến ngôn ngữ cấp cao hơn.
* Ban đầu Git được xây dựng để làm việc trên hạt nhân Linux, tốc độ và hiệu suất là một mục tiêu thiết kế chính của Git ngay từ đầu.
* Người ta có dẫn ra một vài biểu đồ để so sánh Git với SVN (cột bé hơn thể hiện là nhanh hơn)

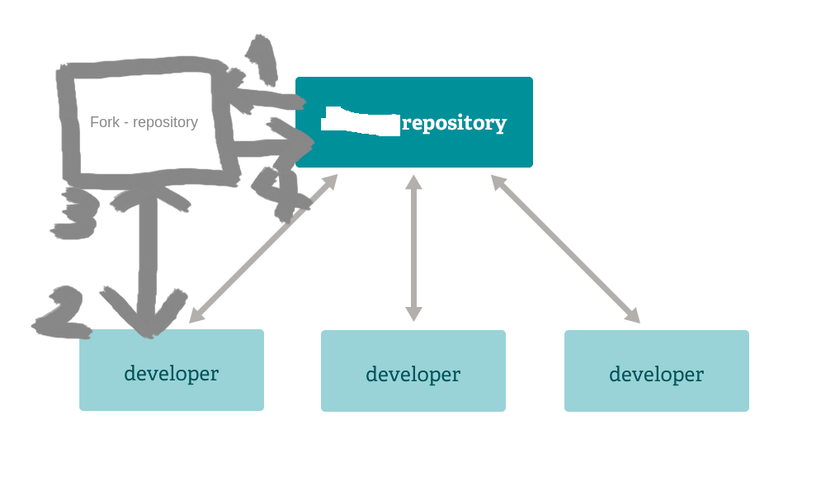


**-> Distributed: Phân phối**

* Backup: Một trong những tính năng hấp dẫn nhất của bất kỳ DSCM nào, trong đó có Git là mỗi người dùng đều có thể tạo một bản sao lưu đầy đủ dữ liệu trên máy chủ chính về máy tính cá nhân của mình, hành động này gọi là clone. Mỗi bản sao này có thể được đẩy lên (push) để thay thế máy chủ chính trong trường hợp rủi ro nào đó hoặc có thể liên tục cập nhật (pull) về những thay đổi từ máy chủ mọi lúc. Đây là 2 kiểu làm việc với Git thông dụng nhất.
* . Subversion-Style: Tạm dịch là kiểu lật đổ, minh họa như hình bên dưới, developer clone shared repository về máy của mình, cập nhật code, sau đó add, commit rồi push lên shared repository. Mỗi một lần commit là một lần lưu lại trạng thái của code dưới local, còn để người khác có thể nhìn thấy được sự thay đổi đó thì cần push lên server.
* Lưu ý là Git sẽ không cho phép bạn push lên máy chủ nếu ai đó đã push trong khoảng thời gian từ lần cuối cùng bạn pull về trở đi. Khi đó bạn cần phải pull về, kiểm tra lại, add, commit rồi mới push lên.



* Integration Manager Tạm dịch là tích hợp quản lý, kiểu này nhìn vào chuyên nghiệp hơn. Nhưng hình ảnh trên trang chủ của Git mô tả hơi khó hiểu, tạm chế bức ảnh trên



1. Sao chép repository về kho của mình, gọi là fork repo
2. Clone fork repo từ kho về máy tính cá nhân
3. Sau khi chỉnh sửa, push trạng thái mới lên fork repo
4. Tạo bản so sánh giữa 2 branch của repo, gọi là pull request, gửi cho người quản lý có quyền merge code mới vào.

**-> Data Assurance**

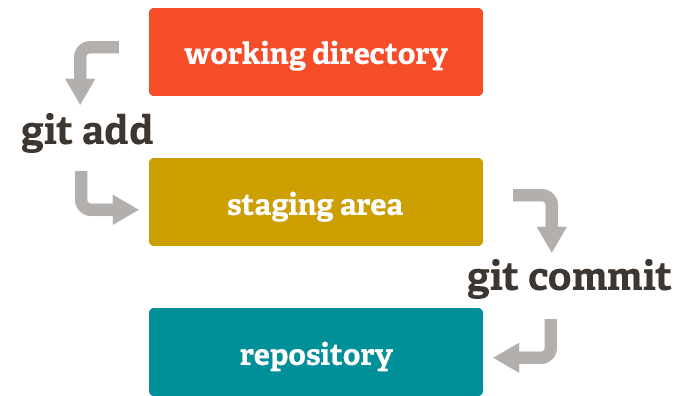
Git đảm bảo tính toàn vẹn của dữ liệu. Mỗi một commit sẽ đính kèm theo một message mô tả sự thay đổi của code cho tới thời điểm bạn commit. Như vậy mỗi khi bạn commit thì Git sẽ nhận nội dung thay đổi code của bạn và nội dung của commit. Git đảm bảo rằng những nội dung này được toàn vẹn.

Mọi thứ trong Git được "băm" (checksum or hash) trước khi lưu trữ và được tham chiếu tới bằng mã băm đó. Có nghĩa là việc thay đổi nội dung của một tập tin hay một thư mục mà Git không biết tới là điều không thể. Chức năng này được xây dựng trong Git ở tầng thấp nhất, bạn không thể mất thông tin/ dữ liệu trong khi truyền tải hoặc nhận về một tập tin bị hỏng mà Git không phát hiện ra.

Cơ chế mà Git sử dụng cho việc băm này được gọi là mã băm SHA-1. Đây là một chuỗi được tạo thành bởi 40 ký tự của hệ cơ số 16 (0-9 và a-f) và được tính toán dựa trên nội dung của tập tin hoặc cấu trúc thư mục trong Git.

**-> Staging Area**

Đây là một khu vực trung gian mà commit có thể được định dạng và xem lại trước khi hoàn thành. Mỗi tập tin trong Git được quản lý dựa trên ba trạng thái: committed, modifed, và staged. Committed có nghĩa là dữ liệu đã được lưu trữ một cách an toàn trong cơ sở dữ liệu. Modified có nghĩa là bạn đã thay đổi tập tin nhưng chưa commit vào cơ sở dữ liệu. Staged là bạn đã đánh dấu sẽ commit phiên bản hiện tại của một tập tin đã chỉnh sửa trong lần commit sắp tới. Điều này tạo ra ba phần riêng biệt của một dự án sử dụng Git: thư mục Git, thư mục làm việc và khu vực tổ chức (staging area).



Các file mà chưa được add vào staging area thì Git sẽ không quan tâm, chỉ khi nào bạn dùng lệnh add, chuyển file vào staging area thì nó mới quản lý, khi bạn commit file nào thì file đó được chuyển từ staging area vào repo.

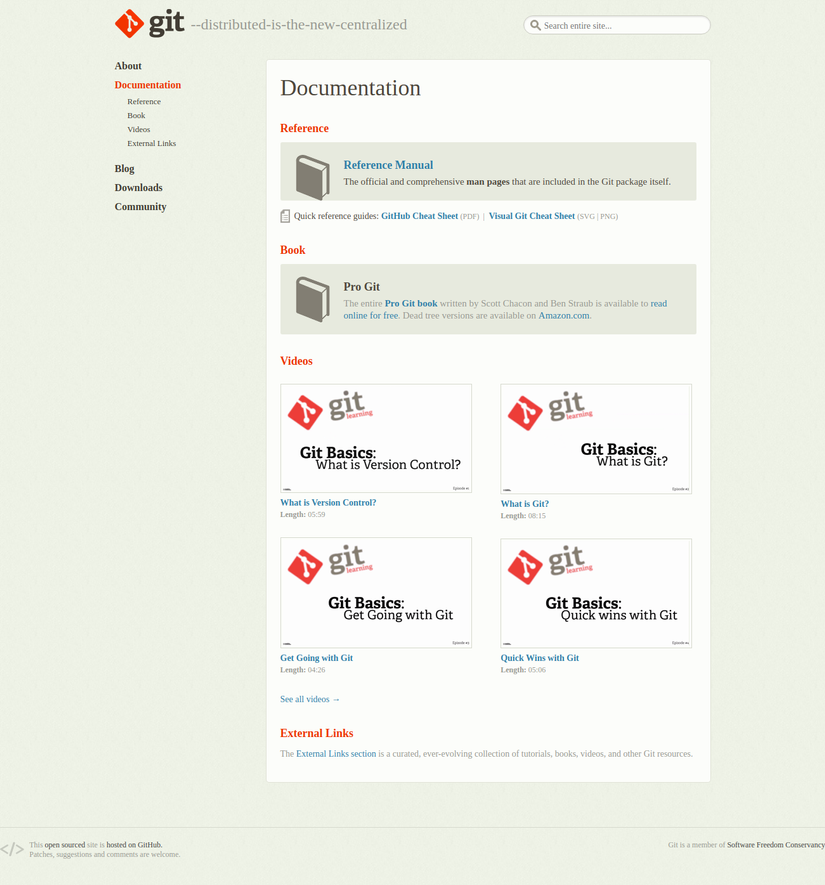
**-> Free and Open source**

**-> Trademark**

### Git - documentation

Hình ảnh bên dưới là trang documentation của Git, gồm 4 phần:

* Reference: Giới thiệu rất chi tiết từng câu lệnh của Git.
* Book: Phần này giới thiệu nội dung cuốn Pro Git, nếu đọc hết cuốn này thì sẽ rất ok.
* Video: Một vài video tham khảm
* External links: Một vài liên kết có liên quan.



#### Install Git

Tham khảo ở mục [Download](https://git-scm.com/downloads) trên trang chủ của Git

#### Tools

* **Tool**: Thuần túy nhất thì các bạn làm việc với command line trên terminal của Linux hoặc Cmd của Windows. Bên cạnh đó thì có các GUI hỗ trợ như SourceTree, GIT Desktop; một số editor như PHP storm hoặc Ruby Mine cũng có tích hợp. Khi dùng command line, cảm thấy khi đó mình sẽ hiểu rõ bản chất vấn đề hơn, tùy biến được nhiều hơn, giải quyết mọi thứ theo ý mình.
* **Server**: Có một số nhà cung cấp như GitHub, GitBucket, Gitlab.
* Sau đây, sẽ hướng dẫn các bạn làm việc với command line trên ubuntu và sử dụng server GitHub. Có một lưu ý là nếu dùng command line trên ubuntu thì bạn nên dùng Zsh, tối ưu nó một chứ. Cài thêm plugin auto\_suggest chứ nếu dùng chay với Bash thì hơi mệt. Tham khảo bài viết của mình tại [đây](https://viblo.asia/p/hieu-ro-hon-toi-uu-va-su-dung-phim-tat-cho-terminal-ORNZqowM50n).

#### Install SSH key

* Trong Git, nếu bạn sử dụng giao thức HTTPs để truyền tải dữ liệu giữa server và client thì mỗi lần bạn pull hay push đều cần phải nhập lại email và mật khẩu Github, vậy hãy dùng giao thức ssh như sau:
* Gõ lệnh sau vào terminal:
  + ssh-keygen -t rsa -C "[abc@gmail.com](mailto:abc@gmail.com)"
* Câu lệnh trên sẽ sinh ra một public key và một private key, private key thì không chia sẻ, còn public key thì share nó với github.
* Đăng nhập vào Github -> Setting -> SSH and GPG keys -> Add new SSH key
* Điều public key vào, từ giờ có thể dùng giao thức SSH được rồi.

## Cơ bản về Git

ref: <https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git>

Session này bao hàm từng lệnh cơ bản cần để thực hiện phần lớn những việc mà bạn sẽ làm với Git. Kết thúc chương này, bạn có thể cấu hình và khởi động được một kho chứa, bắt đầu hay dừng theo dõi các tập tin, và tổ chức/ sắp xếp (stage) cũng như commit các thay đổi. Ở đây cũng sẽ hướng dẫn các bạn làm sao để bỏ qua (ignore) một số tập tin cũng như kiểu tập tin nào đó, làm sao để khôi phục lỗi một cách nhanh chóng và dễ dàng, làm sao để duyệt qua lịch sử của dự án hay xem các thay đổi giữa những lần commit, và làm sao để đẩy lên (push) hay kéo về (pull) từ các kho chứa từ xa.

### Tạo một kho chứa Git

Bạn có thể tạo dự án có sử dụng Git dựa theo hai phương pháp chính.

* Thứ nhất là dùng một dự án hay một thư mục đã có sẵn để nhập (import) vào Git.
* Thứ hai là tạo bản sao của một kho chứa Git đang hoạt động trên một máy chủ khác.

#### Khởi tạo một kho chứa từ thư mục cũ

Nếu bạn muốn theo dõi một dự án cũ trong Git, bạn cần ở trong thư mục của dự án đó và gõ lệnh sau:

$ git init

Lệnh này sẽ tạo một thư mục mới có tên .git, thư mục này chứa tất cả các tập tin cần thiết cho kho chứa - đó chính là bộ khung/xương của kho chứa Git. Cho tới thời điểm hiện tại, vẫn chưa có gì trong dự án của bạn được theo dõi (track).

Nếu bạn muốn kiểm soát phiên bản cho các tập tin có sẵn (đối với một thư mục trống), chắc chắn bạn nên bắt đầu theo dõi các tập tin đó và thực hiện commit đầu tiên/khởi tạo (initial commit). Bạn có thể hoàn thành việc này bằng cách chỉ định tập tin bạn muốn theo dõi trong mỗi lần commit sử dụng câu lệnh git add:

* Example:

$ git add \*.c  
$ git add README  
$ git commit -m 'phiên bản đầu tiên/khởi tạo của dự án'

#### Sao chép một kho chứa đã tồn tại

Nếu như bạn muốn có một bản sao của một kho chứa Git có sẵn - ví dụ như, một dự án mà bạn muốn đóng góp vào - câu lệnh bạn cần là git clone

Sử dụng lệnh git clone [url] để sao chép một kho chứa.

* Example :

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

Bạn có thể sử dụng Git thông qua một số "giao thức truyền tải" (transfer protocol) khác nhau. Ví dụ trước sử dụng giao thức git://, nhưng bạn cũng có thể sử dụng http(s):// hoặc user@server:/path.git thông qua giao thức SSH.

### Ghi lại thay đổi vào kho chứa

ref: <https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-Ghi-L%E1%BA%A1i-Thay-%C4%90%E1%BB%95i-v%C3%A0o-Kho-Ch%E1%BB%A9a>

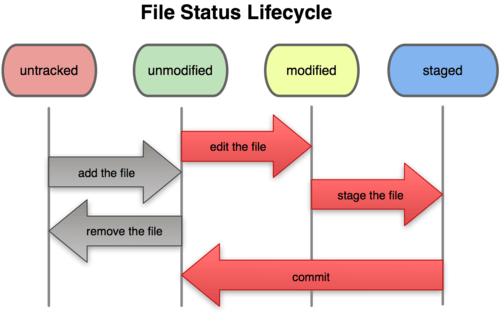
Bây giờ bạn đã có một kho chứa Git thật sự và một bản sao dữ liệu của dự án để làm việc. Bạn cần thực hiện một số thay đổi và commit ảnh của chúng vào kho chứa mỗi lần dự án đạt tới một trạng thái nào đó mà bạn muốn ghi lại.

Mỗi tập tin trong thư mục làm việc có thể ở một trong hai trang thái: tracked hoặc untracked.

Tập tin tracked là các tập tin đã có mặt trong ảnh (snapshot) trước; chúng có thể là unmodified, modified, hoặc staged. Tập tin untracked là các tập tin còn lại - bất kỳ tập tin nào trong thư mục làm việc của bạn mà không có ở ảnh (lần commit) trước hoặc không ở trong khu vực tổ chức (staging area).

Ban đầu, khi bạn tạo bản sao của một kho chứa, tất cả tập tin ở trạng thái "đã được theo dõi" (tracked) và "chưa thay đổi" (unmodified) vì bạn vừa tải chúng về và chưa thực hiện bất kỳ thay đổi nào.

Khi bạn chỉnh sửa các tập tin, Git coi là chúng đã bị thay đổi so với lần commit trước đó. Bạn stage các tập tin bị thay đổi này và sau đó commit tất cả các thay đổi đã được staged (tổ chức) đó, và quá trình này cứ thế lặp đi lặp lại như được miêu tả sau:



#### Kiểm tra trạng thái của tập tin

Công cụ chính để phát hiện trạng thái của tập tin là lệnh git status .

Lệnh này còn cho bạn biết bạn đang thao tác trên "nhánh" (branch) nào.

#### Theo dõi các tập tin mới

Để có thể theo dõi các tập tin mới tạo, bạn sử dụng lệnh git add.

Example: $ git add README

Lệnh git add có thể dùng cho một tập tin hoặc một thư mục; nếu là thư mục, nó sẽ thêm tất cả tập tin trong thư mục đó cũng như các thư mục con.

#### Quản lý các tập tin đã thay đổi

Hãy sửa một tập tin đang được theo dõi. Nếu bạn sửa một tập tin đang được theo dõi như benchmarks.rb sau đó chạy lệnh status, bạn sẽ thấy tương tự như sau:

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

Tập tin benchmarks.rb nằm trong danh sách "các thay đổi chưa được tổ chức/đánh dấu để commit" - có nghĩa là một tập tin đang được theo dõi đã bị thay đổi trong thư mục làm việc nhưng chưa được "staged". Để làm việc này, bạn chạy lệnh git add (đó là một lệnh đa chức năng - bạn có thể dùng nó để bắt đầu theo dõi tập tin, tổ chức tập tin, hoặc các việc khác như đánh dấu đã giải quyết xong các tập tin có nội dung mâu thuẫn nhau khi tích hợp).

Nếu như bạn chỉnh sửa một tập tin sau khi chạy git add, bạn phải chạy git add lại một lần nữa để đưa nó vào phiên bản mới nhất

#### Bỏ qua các tập tin

Thường thì hay có một số loại tập tin mà bạn không muốn Git tự động thêm nó vào hoặc thậm chí hiển thị là không được theo dõi. Những tập tin này thường được tạo tự động ví dụ như các tập tin nhật ký (logfiles) hay các tập được sinh ra khi biên dịch chương trình.

Trong những trường hợp như thế, bạn có thể tạo một tập tin liệt kê các "mẫu" (patterns) để tìm những tập tin này có tên .gitignore.

* Ví dụ của .gitignore:

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

Dòng đầu tiên yêu cầu Git bỏ qua tất cả các tập tin có đuôi là .o hoặc .a - các tập tin object và archive có thể được tạo ra khi bạn dịch mã nguồn. Dòng thứ hai yêu cầu Git bỏ qua tất cả các tập tin có đuôi là dấu ngã (~), chúng được sử dụng để lưu các giá trị tạm thời bởi rất nhiều chương trình soạn thảo. Bạn có thể thêm vào các thư mục như log, tmp, hay pid; hay các tài liệu được tạo tự động, .... Tạo một tập tin .gitignore trước khi bắt đầu làm việc là một ý tưởng tốt, như vậy bạn sẽ không vô tình commit các tập tin mà bạn không muốn.

**Quy tắc cho các mẫu có thể sử dụng trong .gitignore như sau:**

* Dòng trống hoặc bắt đầu với # sẽ được bỏ qua.
* Các mẫu chuẩn toàn cầu hoạt động tốt.
* Mẫu có thể kết thúc bằng dấu gạch chéo (/) để chỉ định một thư mục.
* Bạn có thể có "mẫu phủ định" bằng cách thêm dấu cảm thán vào phía trước (!).

Các mẫu toàn cầu giống như các biểu thức chính quy (regular expression) rút gọn được sử dụng trong shell.

* Dấu (\*) khớp với 0 hoặc nhiều ký tự
* [abc] khớp với bất kỳ ký tự nào trong dấu ngoặc (trong trường hợp là a, b hoặc c)
* Dấu hỏi (?) khớp với một ký tự đơn
* Dấu ngoặc có ký tự được ngăn cách bởi dấu gạch ngang [0-9] khớp bất kỳ ký tự nào trong khoảng đó (ở đây là từ 0 đến 9)

Ví dụ của .gitignore:

# a comment - dòng này được bỏ qua  
# không theo dõi tập tin có đuôi .a   
\*.a  
# nhưng theo dõi tập lib.a, mặc dù bạn đang bỏ qua tất cả tập tin .a ở trên  
!lib.a  
# chỉ bỏ qua tập TODO ở thư mục gốc, chứ không phải ở các thư mục con subdir/TODO  
/TODO  
# bỏ qua tất cả tập tin trong thư mục build/  
build/  
# bỏ qua doc/notes.txt, không phải doc/server/arch.txt  
doc/\*.txt  
# bỏ qua tất cả tập .txt trong thư mục doc/  
doc/\*\*/\*.txt

#### Xem các thay đổi staged và unstaged

Nếu câu lệnh git status quá mơ hồ với bạn - bạn muốn biết chính xác cái đã thay đổi là gì, chứ không chỉ là tập tin nào bị thay đổi - bạn có thể sử dụng lệnh git diff. Bạn sẽ thường xuyên sử dụng nó để trả lời cho hai câu hỏi sau: Cái bạn đã thay đổi nhưng chưa được staged là gì? Và những thứ đã được staged để chuẩn bị commit là gì?. Lệnh git diff chỉ cho bạn chính xác từng dòng đã được thêm hoặc xóa - hay còn được biết đến như là bản vá (patch).

Example:

Giả sử bạn sửa và stage tập tin README lại một lần nữa, sau đó là sửa tập benchmarks.rb mà không stage nó. Nếu dùng lệnh status, bạn sẽ thấy:

$ git status  
# On branch master  
# Changes to be committed:  
# (use "git reset HEAD <file>..." to unstage)  
#  
# new file: README  
#  
# Changes not staged for commit:  
# (use "git add <file>..." to update what will be committed)  
#  
# modified: benchmarks.rb  
#  
Để xem chính xác bạn đã thay đổi nhưng chưa stage những gì, bạn dùng git diff không sử dụng tham số nào khác:

$ git diff  
diff --git a/benchmarks.rb b/benchmarks.rb  
index 3cb747f..da65585 100644  
--- a/benchmarks.rb  
+++ b/benchmarks.rb  
@@ -36,6 +36,10 @@ def main  
 @commit.parents[0].parents[0].parents[0]  
 end  
  
+ run\_code(x, 'commits 1') do  
+ git.commits.size  
+ end  
+  
 run\_code(x, 'commits 2') do  
 log = git.commits('master', 15)  
 Log.size

Câu lệnh này so sánh cái ở trong thư mục làm việc của bạn với cái ở trong khu vực tổ chức (staging). Kết quả cho bạn biết những thứ đã bị thay đổi mà chưa được stage.

Nếu bạn muốn xem những gì bạn đã staged mà chuẩn bị được commit, bạn có thể sử dụng git diff -- cached. (Từ Git 1.6.1 trở đi, bạn có thể sử dụng git diff --staged, có thể sẽ dễ nhớ hơn). Lệnh này so sánh những thay đổi đã được tổ chức với lần commit trước đó:

* $ git diff --cached  
  diff --git a/README b/README  
  new file mode 100644  
  index 0000000..03902a1  
  --- /dev/null  
  +++ b/README2  
  @@ -0,0 +1,5 @@  
  +grit  
  + by Tom Preston-Werner, Chris Wanstrath  
  + http://github.com/mojombo/grit  
  +  
  +Grit is a Ruby library for extracting information from a Git repository

Một điều quan trọng cần ghi nhớ là chỉ chạy git diff không thôi thì nó sẽ không hiển thị cho bạn tất cả thay đổi từ lần commit trước - mà chỉ có các thay đổi chưa được tổ chức. Điều này có thể gây khó hiểu một chút, bởi vì nếu như bạn đã tổ chức tất cả các thay đổi, git diff sẽ không hiện gì cả.

Thêm một ví dụ nữa, nếu như bạn tổ chức tập tin benchmarks.rb rồi sau đó mới sửa nó, bạn có thể sử dụng git diff để xem các thay đổi đã tổ chức cũng như chưa tổ chức.

Bạn có thể sử dụng git diff để xem những gì vẫn chưa được tổ chức

$ git diff  
diff --git a/benchmarks.rb b/benchmarks.rb  
index e445e28..86b2f7c 100644  
--- a/benchmarks.rb  
+++ b/benchmarks.rb  
@@ -127,3 +127,4 @@ end  
 main()  
  
 ##pp Grit::GitRuby.cache\_client.stats  
+# test line

và git diff --cached để xem những gì đã được tổ chức tới thời điểm hiện tại:

$ git diff --cached  
diff --git a/benchmarks.rb b/benchmarks.rb  
index 3cb747f..e445e28 100644  
--- a/benchmarks.rb  
+++ b/benchmarks.rb  
@@ -36,6 +36,10 @@ def main  
 @commit.parents[0].parents[0].parents[0]  
 end  
  
+ run\_code(x, 'commits 1') do  
+ git.commits.size  
+ end  
+  
 run\_code(x, 'commits 2') do  
 log = git.commits('master', 15)  
 Log.size

#### Commit thay đổi

Bây giờ, sau khi đã tổ chức các tập tin theo ý muốn, bạn có thể commit chúng. Hãy nhớ là những gì chưa được tổ chức - bất kỳ tập tin nào được tạo ra hoặc sửa đổi sau khi chạy lệnh git add - sẽ không được commit. Cách đơn giản nhất để commit là lệnh git commit

$ git commit

Sau khi chạy lệnh này, chương trình soạn thảo do bạn lựa chọn sẽ được mở lên. Nó sẽ hiển thị đoạn thông báo sau (màng hình vim)

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

Bạn có thể thấy thông báo mặc định có chứa nội dung của lần chạy git status cuối cùng được dùng làm chú thích. Bạn có thể xóa chú thích này đi và nhập vào nội dung riêng của bạn cho commit đó, hoặc bạn có thể giữ nguyên như vậy để giúp bạn nhớ được những gì đang commit. (Một cách nữa để nhắc nhở bạn rõ ràng hơn những gì bạn đã sửa là truyền vào tham số -v cho git commit. Làm như vậy sẽ đưa tất cả thay đổi như khi thực hiện lệnh git diff vào chương trình soạn thảo, như vậy bạn có thể biết chính xác những gì bạn đã làm.) Khi bạn thoát ra khỏi chương trình soạn thảo, Git tạo commit của bạn với thông báo/điệp đó (các chú thích và diff sẽ bị bỏ đi).

Nói cách khác, bạn có thể gõ trực tiếp thông điệp cùng với lệnh commit bằng cách thêm vào sau cờ -m, như sau:

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

Hãy nhớ là commit lưu lại ảnh các tập tin mà bạn chỉ định trong khu vực tổ chức. Bất kỳ tập tin nào không ở trong đó sẽ vẫn giữ nguyên trạng thái là đã sửa (modified); bạn có thể thực hiện một commit khác để thêm chúng vào lịch sử. Mỗi lần thực hiện commit là bạn đang ghi lại ảnh của dự án mà bạn có thể dựa vào đó để so sánh hoặc khôi phục về sau này.

#### Bỏ qua khu vực tổ chức

Mặc dù tự tổ chức commit theo cách bạn muốn là một cách hay, tuy nhiên đôi khi khu vực tổ chức khiến quy trình làm việc của bạn trở nên phức tạp. Nếu bạn muốn bỏ qua bước này, chỉ cần thêm vào lựa chọn -a khi thực hiện git commit, Git sẽ tự động thêm tất cả các tập tin đã được theo dõi trước khi thực hiện lệnh commit, cho phép bạn bỏ qua bước git add

#### Xóa tập tin

Để xóa một tập tin khỏi Git, bạn phải xóa nó khỏi danh sách được theo dõi (chính xác hơn, xóa nó khỏi khu vực tổ chức) và sau đó commit. Lệnh git rm thực hiện điều đó và cũng xóa tập tin khỏi thư mục làm việc vì thế bạn sẽ không thấy nó như là tập tin không được theo dõi trong những lần tiếp theo.

Nếu bạn chỉ đơn giản xóa tập tin khỏi thư mục làm việc, nó sẽ được hiển thị trong phần "Thay đổi không được tổ chức để commit" (hay unstaged) khi bạn chạy git status:

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

Khi đó, nếu bạn chạy git rm, Git sẽ xóa tập tin đó khỏi khu vực tổ chức:

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

Lần commit tới, tập tin sẽ bị xóa và không được theo dõi nữa. Nếu như bạn đã sửa và thêm tập tin đó vào danh sách, bạn phải ép Git xóa đi bằng cách thêm lựa chọn -f. Đây là một chức năng an toàn nhằm ngăn chặn việc xóa nhầm dữ liệu chưa được lưu vào ảnh và nó sẽ không thể được khôi phục từ Git.

Một chức năng hữu ích khác có thể bạn muốn sử dụng đó là giữ tập tin trong thư mục làm việc nhưng không thêm chúng vào khu vực tổ chức. Hay nói cách khác bạn muốn lưu tập tin trên ổ cứng nhưng không muốn Git theo dõi chúng nữa. Điều này đặc biệt hữu ích nếu như bạn quên thêm nó vào tập .gitignore và vô tình tổ chức (stage) chúng, ví dụ như một tập tin nhật ký lớn hoặc rất nhiều tập tin .a. Để làm được điều này, hãy sử dụng lựa chọn --cached:

$ git rm --cached readme.txt

Bạn có thể truyền vào tập tin, thư mục hay mẫu (patterns) vào lệnh git rm. Nghĩa là bạn có thể thực hiện tương tự như:  
$ git rm log/\\*.log

Chú ý dấu chéo ngược (\) đằng trước \*. Việc này là cần thiết vì ngoài phần mở rộng mặc định Git còn sử dụng thêm phần mở rộng riêng - "This is necessary because Git does its own filename expansion in addition to your shell's filename expansion". Trên windows, dấu gạch ngược (\) phải bỏ đi. Lệnh này xóa toàn bộ tập tin có đuôi .log trong thư mục log/. Hoặc bạn có thể thực hiện tương tự như sau:

$ git rm \\*~

Lệnh này xóa toàn bộ tập tin kết thúc bằng ~.

#### Di chuyển tập tin

Không giống như các hệ thống quản lý phiên bản khác, Git không theo dõi việc di chuyển tập tin một cách rõ ràng. Nếu bạn đổi tên một tập tin trong Git, không có thông tin nào được lưu trữ trong Git có thể cho bạn biết là bạn đã đổi tên một tập tin. Tuy nhiên, Git rất thông minh trong việc tìm ra điều đó - chúng ta sẽ nói về phát hiện việc di chuyển các tập tin sau.

Vì thế nên nó hơi khó hiểu khi Git cung cấp lệnh mv. Nếu bạn muốn đổi tên một tập tin trong Git, bạn có thể dùng:

$ git mv file\_from file\_to

và nó chạy tốt. Thực tế, nếu bạn chạy lệnh tương tự và sau đó kiểm tra trạng thái, bạn sẽ thấy Git coi là nó đã đổi tên một tập tin:

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

Tuy nhiên, việc này lại tương tự việc thực hiện như sau:

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

Git ngầm hiểu đó là đổi tên, vì thế dù bạn đổi tên bằng cách này hoặc dùng lệnh mv cũng không quan trọng. Sự khác biệt duy nhất ở đây là mv là một lệnh duy nhất thay vì ba - sử dụng nó thuận tiện hơn rất nhiều. Quan trọng hơn, bạn có thể dùng bất kỳ cách nào để đổi tên một tập tin, và chạy add/rm sau đó, trước khi commit.

### Xem lịch sử Commit

ref: <https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-Xem-L%E1%BB%8Bch-S%E1%BB%AD-Commit>

Sau khi bạn đã thực hiện rất nhiều commit, hoặc bạn đã sao chép một kho chứa với các commit có sẵn, chắc chắn bạn sẽ muốn xem lại những gì đã xảy ra. Cách đơn giản và có hiệu lực tốt nhất là sử dụng lệnh git log.

Có rất nhiều tùy chọn (tham biến/số) khác nhau cho lệnh git log giúp bạn chỉ hiển thị thứ mà bạn thực sự muốn.

#### Giới hạn thông tin đầu ra

Ngoài các lựa chọn để định dạng đầu ra, git log còn nhận vào một số các lựa chọn khác cho mục đích giới hạn khác - là các lựa chọn cho phép bạn hiển thị một phần các commit. Git hiển thị đầu ra theo trang do vậy bạn chỉ xem được một trang lịch sử tại một thời điểm.

Tuy nhiên, tham số kiểu giới hạn theo thời gian như --since và --until khá hữu ích. Ví dụ, lệnh này hiển thị các commit được thực hiện trong vòng hai tuần gần nhất:

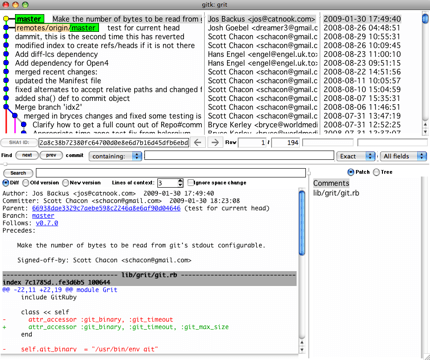
$ git log --since=2.weeks

Lệnh này hoạt động được với rất nhiều định dạng - bạn có thể chỉ định một ngày cụ thể ("2019-01-21") hoặc tương đối như "2 years 1 day 3 minutes ago".

Bạn cũng có thể lọc các commit thỏa mãn một số tiêu chí nhất định. Tham số --author cho phép bạn lọc một tác giả nhất định, và tham số --grep cho phép bạn tìm kiếm các từ khóa trong thông điệp của commit. (Lưu ý là nếu như bạn muốn chỉ định tham số author và grep, bạn phải thêm vào --all-match bằng không lệnh đó sẽ chỉ tìm kiếm các commit thỏa mãn một trong hai.)

#### Hiển thị lịch sử trên giao diện

Nếu bạn muốn sử dụng một công cụ đồ họa để trực quan hóa lịch sử commit, bạn có thể thử một chương trình Tcl/Tk có tên gitk được xuất bản kèm với git. Nếu bạn gõ gitk trên thư mục của dự án, bạn sẽ thấy giống như sau:



Bạn có thể xem lịch sử commit ở phần nửa trên của cửa sổ cùng một biểu đồ "cây" (ancestry) trực quan. Phần xem diff ở nửa dưới của cửa sổ hiển thị các thay đổi trong bất kỳ commit nào bạn click ở trên.

### Phục hồi

Tại một thời điểm bất kỳ, bạn có thể muốn phục hồi (undo) một phần nào đó. Bây giờ, chúng ta sẽ cùng xem xét một số công cụ cơ bản dùng cho việc phục hồi các thay đổi đã thực hiện. Hãy cẩn thận, bởi vì không phải lúc nào bạn cũng có thể làm được điều này. Đây là một trong số ít thuộc thành phần của Git mà bạn có thể mất dữ liệu nếu làm sai.

#### Thay đổi commit cuối cùng

Một trong những cách phục hồi phổ biến thường dùng khi bạn commit quá sớm/vội và có thể quên thêm vào đó một số tập tin hoặc là thông điệp commit không như ý muốn. Nếu như bạn muốn thực hiện lại commit đó, bạn có thể chạy lệnh commit với tham số --amend:

$ git commit --amend

Lệnh này sử dụng khu vực tổ chức để commit. Nếu bạn không thay đổi gì thêm từ lần commit cuối cùng (ví dụ, bạn chạy lệnh này ngay lập tức sau commit trước đó), thì ảnh của dự án sẽ vẫn như vậy và tất cả những gì bạn thay đổi là thông điệp của commit.

Trình soạn thảo văn bản xuất hiện để bạn thay đổi thông điệp của commit, nhưng nó đã chứa nội dung thông điệp của commit trước đó. Bạn có thể sửa nội dung như thường lệ, và nó sẽ được ghi đè lên commit trước đó.

Ví dụ, nếu như bạn thực hiện xong commit và rồi sau đó mới nhận ra rằng đã quên tổ chức các thay đổi trong tập tin bạn muốn để thêm vào commit đó, bạn có thể chạy lệnh sau:

$ git commit -m 'initial commit'  
$ git add forgotten\_file  
$ git commit --amend

Sau khi chạy ba lệnh này, kết quả cuối cùng cũng vẫn chỉ là một commit - commit thứ hai sẽ thay thế các kết quả của commit trước đó.

#### Loại bỏ tập tin đã tổ chức

Hai phần tiếp theo sẽ minh họa cho bạn thấy làm sao để thỏa hiệp các thay đổi giữa khu vực tổ chức và thư mục làm việc. Cái hay ở đây là câu lệnh sử dụng để xác định trạng thái của hai khu vực đồng thời cũng gợi ý cho bạn làm sao có thể phục hồi các thay đổi.

Ví dụ, giả sử bạn sửa nội dung của hai tập tin và muốn commit chúng làm hai lần riêng biệt nhau, nhưng bạn đã vô tình sử dụng git add \* và tổ chức cả hai. Vậy làm thế nào để loại bỏ một trong hai khỏi khu vực tổ chức? Lệnh git status sẽ giúp bạn:

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

Ngay dưới phần "Thay đổi sắp được commit", nó chỉ ra rằng "sử dụng git reset HEAD <file> ... để loại bỏ khỏi khu vực tổ chức". Vậy thì hãy làm theo gợi ý đó để loại bỏ tập tin benchmarks.rb:

$ git reset HEAD benchmarks.rb  
benchmarks.rb: locally modified  
$ git status  
# On branch master  
# Changes to be committed:  
# (use "git reset HEAD <file>..." to unstage)  
#  
# modified: README.txt  
#  
# Changes not staged for commit:  
# (use "git add <file>..." to update what will be committed)  
# (use "git checkout -- <file>..." to discard changes in working directory)  
#  
# modified: benchmarks.rb  
#

Lệnh này hơi khác biệt một chút, nhưng nó hoạt động đúng như chúng ta mong đợi. Tập tin benchmarks.rb được thay đổi và một lần nữa lại trở thành chưa tổ chức.

#### Phục hồi tập tin đã thay đổi

Sẽ như thế nào khi bạn nhận ra rằng bạn không muốn giữ những thay đổi trong tập tin benchmarks.rb?

Làm thế nào để dễ dàng phục hồi lại những thay đổi đó - phục hồi nó lại trạng thái giống như sau khi thực hiện commit cuối cùng (hoặc như sau khi sao chép (initially cloned), hoặc như lúc bạn mới đưa chúng vào thư mục làm việc)? May mắn là git status cũng sẽ cho bạn biết làm sao để thực hiện được việc đó. Trong thông báo đầu ra của ví dụ vừa rồi, khu vực tổ chức của chúng ta như sau:

# Changes not staged for commit:  
# (use "git add <file>..." to update what will be committed)  
# (use "git checkout -- <file>..." to discard changes in working directory)  
#  
# modified: benchmarks.rb  
#

Nó chỉ cho bạn rõ ràng làm sao có thể hủy những thay đổi vừa được thực hiện. Hãy làm theo hướng dẫn:

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

Bạn có thể thấy những thay đổi mà bạn vừa mới phục hồi. Bạn cũng nên nhận ra rằng đây là một câu lệnh nguy hiểm; bất kỳ thay đổi nào được thực hiện trên tập tin đó không còn nữa - bạn vừa mới sao chép một tập tin khác thay thế nó. Đừng nên sử dụng lệnh này trừ khi bạn biết rõ ràng rằng bạn không cần đến tập tin đó. Nếu bạn chỉ không muốn thấy nó nữa, chúng ta sẽ tìm hiểu về phân nhánh và lưu trữ (stashing) trong phần sau; chúng là các phương pháp thay thế tốt hơn.

Hãy nhớ là, bất cứ thứ gì được commit vào Git luôn có thể phục hồi lại. Thậm chí cả các commit ở các nhánh đã bị xóa hoặc bị ghi đè bởi --amend. Tuy nhiên, bất cứ thứ gì bị mất mà chưa được commit thì không có cơ hội phục hồi lại.

### Làm việc từ xa

Để có thể cùng cộng tác với các thành viên khác trên bất kỳ dự án sử dụng Git nào, bạn cần phải biết quản lý các kho chứa của bạn. Các kho chứa từ xa là các phiên bản của dự án của bạn, được lưu trữ trên Internet hoặc một mạng lưới nào đó. Bạn có thể có nhiều kho chứa khác nhau, thường thì bạn có thể chỉ đọc hoặc đọc/ghi.

Cộng tác với các thành viên khác liên quan đến quản lý những kho chứa này và việc kéo, đẩy dữ liệu từ chúng khi bạn cần chia sẻ công việc.

Quản lý các kho chứa từ xa đòi hỏi phải biết cách thêm các kho chứa, xóa kho chứa không hợp lệ, quản lý nhiều nhánh khác nhau và xác định có theo dõi chúng hay không và còn nhiều hơn thế nữa.

#### Hiển thị máy chủ

Để xem bạn đã cấu hình tới máy chủ từ xa nào, bạn có thể chạy lệnh git remote. Nó sẽ liệt kê tên ngắn gọn của mỗi máy chủ từ xa bạn đã chỉ định. Nếu bạn sao chép nó từ một kho chứa có sẵn, ít nhất bạn sẽ thấy bản gốc (origin) - tên mặc định mà Git đặt cho phiên bản trên máy chủ mà bạn đã sao chép từ đó:

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

Bạn cũng có thể sử dụng tham số -v để hiển thị địa chỉ mà Git đã lưu tên rút gọn đó.

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

Nếu bạn có nhiều hơn một máy chủ từ xa, lệnh này sẽ liệt kê hết tất cả. Ví dụ, kho chứa Grit sẽ hiển thị như sau:

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

Điều này có nghĩa là bạn có thể "kéo" những đóng góp từ bất kỳ người dùng nào ở trên một cách dễ dàng. Nhưng chỉ máy chủ nguyên bản từ xa (origin remote) là có địa chỉ SSH, do vậy nó là cái duy nhất mà bạn có thể đẩy lên.

#### Thêm các kho chứa từ xa

Để thêm mới một kho chứa Git từ xa như là một tên rút gọn để bạn có thể tham khảo dễ dàng, hãy chạy lệnh git remote add [shortname] [url]:

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

Bây giờ bạn có thể sử dụng pb trong các câu lệnh, nó có tác dụng tương đương với một địa chỉ hoàn chỉnh. Ví dụ, nếu bạn muốn duyệt qua/truy cập tất cả thông tin mà Paul có mà bạn chưa có trong kho chứa, bạn có thể chạy lệnh git fetch pb:

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

Nhánh chính của Paul có thể truy cập cục bộ như là pb/master - bạn có thể tích hợp nó vào các nhánh của bạn, hoặc sử dụng nó như là một nhánh cục bộ ở thời điểm đó nếu như bạn muốn kiểm tra nó.

#### Truy cập và kéo về từ máy chủ trung tâm

Để lấy dữ liệu của các dự án từ xa về, bạn chạy:

$ git fetch [remote-name]

Sau khi thực hiện xong bước này, bạn đã có các tham chiếu đến toàn bộ các nhánh của dự án từ xa đó, nơi bạn có thể tích hợp hoặc kiểm tra bất kỳ thời điểm nào.

Nếu bạn tạo bản sao từ một kho chứa nào đó khác, lệnh này sẽ tự động kho chứa từ xa đó vào dưới tên origin. Vì thế, git fetch origin sẽ truy xuất (fetch) bất kỳ thay đổi mới nào được đẩy lên trên máy chủ từ sau khi bạn sao chép (hoặc lần truy xuất cuối cùng). Hãy ghi nhớ điều quan trọng là lệnh fetch kéo tất cả dữ liệu về kho chứa trên máy của bạn - nó không tự động tích hợp với bất kỳ thay đổi nào mà bạn đang thực hiện. Bạn phải tích hợp nó một cách thủ công vào kho chứa nội bộ khi đã sẵn sàng.

Nếu bạn có một nhánh được cài đặt để theo dõi một nhánh từ xa khác, bạn có thể sử dụng lệnh git pull để tự động truy xuất và sau đó tích hợp nhánh từ xa vào nhánh nội bộ. Đây có thể là cách dễ dàng và thoải mái hơn cho bạn; và mặc định thì, lệnh git clone tự động cài đặt nhánh chính nội bộ (local master branch) để theo dõi nhánh chính trên máy chủ từ xa (remote master branch) - nơi mà bạn sao chép về (giả sử máy chủ từ xa có một nhánh chính). Thường thì khi chạy lệnh git pull nó sẽ truy xuất dữ liệu từ máy chủ trung tâm nơi lần đầu bạn sao chép và cố gắng tự động tích hợp chúng vào kho chứa hiện thời nơi bạn đang làm việc.

#### Đẩy lên máy chủ trung tâm

Đến một thời điểm nào đó bạn muốn chia sẻ dự án của bạn, bạn phải đẩy ngược nó lên. Câu lệnh để thực hiện rất đơn giản: git push [tên-máy-chủ] [tên-nhánh]. Nếu bạn muốn đẩy nhánh master vào nhánh origin trên máy chủ (nhắc lại, khi sao chép Git thường cài đặt/cấu hình mặc định các tên đó cho bạn), bạn có thể chạy lệnh sau để đẩy các công việc đã hoàn thành ngược lại máy chủ:

$ git push origin master

Lệnh này chỉ hoạt động nếu bạn sao chép từ một máy chủ mà trên đó bạn được cấp phép quyền ghi và chưa có ai khác đẩy dữ liệu lên tại thời điểm đó. Nếu bạn và ai khác cùng sao chép tại cùng một thời điểm; người kia đẩy ngược lên, sau đó bạn cũng đẩy ngược lên, thì hành động của bạn sẽ bị từ chối ngay lập tức. Trước hết bạn cần phải thực hiện kéo các thay đổi mà người đó đã thực hiện và tích hợp/gộp nó vào của bạn, sau đó bạn mới được phép đẩy lên.

#### Kiểm tra một máy chủ trung tâm

Nếu bạn muốn xem chi tiết hơn các thông tin về một kho chứa trung tâm nào đó, bạn có thể sử dụng lệnh git remote show [tên-trung-tâm]. Nếu như bạn chạy lệnh này với một tên rút gọn, như là origin, bạn sẽ thấy tương tự như sau:

$ git remote show origin  
\* remote origin  
 URL: git://github.com/schacon/ticgit.git  
 Remote branch merged with 'git pull' while on branch master  
 master  
 Tracked remote branches  
 master  
 Ticgit

Lệnh này liệt kê địa chỉ của kho chứa trung tâm cũng như thông tin các nhánh đang theo dõi. Nó cho bạn biết rằng nếu như bạn đang ở nhánh master và chạy lệnh git pull, nó sẽ tự động tích hợp nhánh này với nhánh trung tâm sau khi truy xuất toàn bộ các tham chiếu từ xa. Nó cũng liệt kê tất cả các tham chiếu từ xa mà nó đã kéo xuống đó.

Đây là một ví dụ đơn giản mà bạn thường xuyên gặp phải. Khi bạn sử dụng Git thường xuyên hơn, bạn sẽ thường thấy nhiều thông tin hơn từ lệnh git remote show:

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

Lệnh này hiển thị nhánh nào tự động được đẩy lên khi bạn chạy git push trên một nhánh nhất định. Nó cũng cho bạn thấy nhánh nào trên máy chủ trung tâm mà bạn chưa có, nhánh nào bạn có mà đã bị xóa trên máy chủ, và các nhánh nào sẽ tự động được tích hợp khi chạy lệnh git pull.

#### Xóa và đổi tên từ xa

Nếu như bạn muốn đổi tên một tham chiếu, trong những phiên bản gần đây của git bạn có thể chạy git remote rename để đổi tên rút gọn cho một kho chứa từ xa nào đó. Ví dụ:

$ git remote rename pb paul  
$ git remote  
origin  
Paul

Lệnh này đồng thời cũng sẽ thay đổi cả tên các nhánh trung tâm/từ xa của bạn. Các tham chiếu trước đây như pb/master sẽ đổi thành paul/master.

Nếu bạn muốn xóa một tham chiếu đi vì lý do nào đó - bạn đã chuyển máy chủ và không còn sử dụng một bản sao nhất định, hoặc có thể một người dùng nào đó không còn đóng góp vào dự án nữa - bạn có thể sử dụng git remote rm:

$ git remote rm paul  
$ git remote  
Origin

### Đánh dấu

ref: <https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-%C4%90%C3%A1nh-D%E1%BA%A5u>

Cũng giống như đa số các hệ quản trị phiên bản khác, Git có khả năng đánh dấu (tag) các mốc quan trọng trong lịch sử của dự án. Nhìn chung, mọi người sử dụng chức năng này để đánh dấu các thời điểm phát hành (ví dụ như v1.0).

#### Liệt kê Tag

Liệt kê các tag hiện có trong Git khá là đơn giản, bạn chỉ cần gõ git tag:

$ git tag  
v0.1  
V1.3

Lệnh này sẽ liệt kê các tag được sắp xếp theo thứ tự alphabet.

Bạn cũng có thể tìm kiếm một tag sử dụng mẫu (pattern).

$ git tag -l 'v1.4.2.\*'  
v1.4.2.1  
v1.4.2.2  
v1.4.2.3  
V1.4.2.4

#### Thêm tag mới

Git sử dụng hai loại tag chính:

* Lightweight tags

Một lightweight tag (tag hạng nhẹ) giống như một nhánh mà không có sự thay đổi - nó chỉ trỏ đến một commit nào đó. Nói một cách khác để tag các commit là sử dụng lightweight tag. Cơ bản nó là mã băm của một commit được lưu lại vào trong một tập tin - ngoài ra không còn thông tin nào khác. Để tạo một lightweight tag bạn không sử dụng -a, -s, hay -m:

$ git tag v1.4-lw  
$ git tag  
v0.1  
v1.3  
v1.4  
v1.4-lw  
V1.5

Sử dụng lệnh git show lệnh này chỉ show commit mà thôi:

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

* Annotated tags

Annotated (chú thích) tag được lưu trữ như là những đối tượng đầy đủ trong cơ sở dữ liệu của Git. Chúng được băm; chứa tên người tag, địa chỉ email và ngày tháng; có thông điệp kèm theo; và có thể được ký và xác thực bằng GNU Privacy Guard (GPG). Thông thường, annotated tag được khuyến khích sử dụng hơn vì nó chứa các thông tin trên.

Tạo một tag chú thích (annotated) trong Git rất đơn giản, cách dễ nhất là sử dụng -a khi bạn chạy lệnh git tag:

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

Tham số -m được sử dụng để truyền vào một nội dung/thông điệp cho tag. Nếu bạn không chỉ định nội dung cho một annotated tag, Git sẽ mở trình soạn thảo và yêu cầu bạn nhập nội dung vào đó.

Bạn có thể xem được thông tin của tag cùng với commit được tag bằng cách sử dụng lệnh git show:

$ git show v1.4  
tag v1.4  
Tagger: Scott Chacon <schacon@gee-mail.com>  
Date: Mon Feb 9 14:45:11 2009 -0800  
  
my version 1.4  
commit 15027957951b64cf874c3557a0f3547bd83b3ff6  
Merge: 4a447f7... a6b4c97...  
Author: Scott Chacon <schacon@gee-mail.com>  
Date: Sun Feb 8 19:02:46 2009 -0800  
  
 Merge branch 'experiment'

Nó sẽ hiển thị thông tin người tag, ngày commit được tag, và thông báo chú thích trước khi hiện thông tin của commit.

* Signed tags

Bạn cũng có thể ký các tag của bạn sử dụng GPG, giả sử bạn có một private key. Tất cả những gì bạn cần phải làm là sử dụng -s thay vì -a:

$ git tag -s v1.5 -m 'my signed 1.5 tag'  
You need a passphrase to unlock the secret key for  
user: "Scott Chacon <schacon@gee-mail.com>"  
1024-bit DSA key, ID F721C45A, created 2009-02-09

Nếu bạn chạy lệnh git show trên tag đó, bạn có thể thấy được chữ ký GPG của bạn được đính kèm theo nó:

$ git show v1.5  
tag v1.5  
Tagger: Scott Chacon <schacon@gee-mail.com>  
Date: Mon Feb 9 15:22:20 2009 -0800  
  
my signed 1.5 tag  
-----BEGIN PGP SIGNATURE-----  
Version: GnuPG v1.4.8 (Darwin)  
  
iEYEABECAAYFAkmQurIACgkQON3DxfchxFr5cACeIMN+ZxLKggJQf0QYiQBwgySN  
Ki0An2JeAVUCAiJ7Ox6ZEtK+NvZAj82/  
=WryJ  
-----END PGP SIGNATURE-----  
commit 15027957951b64cf874c3557a0f3547bd83b3ff6  
Merge: 4a447f7... a6b4c97...  
Author: Scott Chacon <schacon@gee-mail.com>  
Date: Sun Feb 8 19:02:46 2009 -0800  
  
 Merge branch 'experiment'

#### Xác thực các tag

Để xác thực một tag đã được ký, bạn sử dụng git tag -v [tên-tag]. Lệnh này sử dụng GPG để xác minh chữ ký. Bạn cần phải có public key của người ký để có thể thực hiện được điều này:

$ git tag -v v1.4.2.1  
object 883653babd8ee7ea23e6a5c392bb739348b1eb61  
type commit  
tag v1.4.2.1  
tagger Junio C Hamano <junkio@cox.net> 1158138501 -0700  
  
GIT 1.4.2.1  
  
Minor fixes since 1.4.2, including git-mv and git-http with alternates.  
gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A  
gpg: Good signature from "Junio C Hamano <junkio@cox.net>"  
gpg: aka "[jpeg image of size 1513]"  
Primary key fingerprint: 3565 2A26 2040 E066 C9A7 4A7D C0C6 D9A4 F311 9B9A

#### Tag muộn

Bạn cũng có thể tag các commit mà bạn đã thực hiện trước đó. Giả sử lịch sử commit của bạn giống như sau:

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

Bây giờ, giả sử bạn quên không tag dự án ở phiên bản v1.2, tương đương với commit "update rakefile". Bạn vẫn có thể thêm tag vào lúc này. Để làm được điều này bạn cần chỉ định mã băm của commit (hoặc một phần của nó) ở cuối lệnh:

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

#### Chia sẻ các tag

Mặc định, lệnh git push không truyền (transfer) các tag lên máy chủ trung tâm. Bản phải chỉ định một cách rõ ràng để có thể đẩy các tag lên máy chủ sau khi đã tạo ra chúng. Quá trình này giống như chia sẻ cách nhánh trung tâm - bạn có thể chạy git push origin [tên-tag].

Nếu bạn có rất nhiều tag muốn đẩy lên cùng một lúc, bạn có thể sử dụng tham số --tags cho lệnh git push. Nó sẽ truyền tất cả các tag chưa được đồng bộ lên máy chủ.

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

Bây giờ, nếu ai đó sao chép hoặc kéo dữ liệu từ kho chứa của bạn, họ sẽ cũng có được tất cả các tag.

### Mẹo nhỏ

ref: <https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-M%E1%BA%B9o-Nh%E1%BB%8F>

Trước khi kết thúc session cơ bản về Git này, có một số mẹo nhỏ có thể giúp ích cho việc sử dụng git của bạn trở nên đơn giản và dễ dàng hơn.

#### Gợi ý

Nếu bạn đang sử dụng Bash shell (có thể hiểu là cửa sổ dòng lệnh), Git cung cấp công cụ gợi ý các lệnh rất tốt mà bạn có thể bật nó lên. Nó có thể được tải về trực tiếp từ mã nguồn của Git tại <https://github.com/git/git/blob/master/contrib/completion/git-completion.bash>. Sao chép tập tin này vào thư mục home của bạn và thêm dòng sau vào tập tin .bashrc:

source ~/git-completion.bash

Đó là một mẹo rất hay và đôi khi có thể tiết kiệm thời gian đọc tài liệu của bạn.

#### Bí danh trong git

Git không thể phỏng đoán ra câu lệnh nếu như bạn chỉ gõ một phần của câu lệnh đó. Nếu bạn không muốn gõ toàn bộ từng câu lệnh, bạn có thể dễ dàng cài đặt một bí danh (alias) cho mỗi lệnh sử dụng git config. Sau đây là một số ví dụ có thể hữu ích cho bạn:

$ git config --global alias.co checkout  
$ git config --global alias.br branch  
$ git config --global alias.ci commit  
$ git config --global alias.st status

Có nghĩa là, theo ví dụ thay vì phải gõ git commit, bạn chỉ cần gõ git ci.

Kỹ thuật này cũng có thể rất hữu ích trong việc tạo mới các câu lệnh mà bạn cho rằng sự tồn tại của chúng là cần thiết.

Một bí danh phổ biến là lệnh last, như sau:

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

Với cách này, bạn có thể xem được commit cuối cùng một cách dễ dàng:

$ git last  
commit 66938dae3329c7aebe598c2246a8e6af90d04646  
Author: Josh Goebel <dreamer3@example.com>  
Date: Tue Aug 26 19:48:51 2008 +0800  
  
 test for current head  
  
 Signed-off-by: Scott Chacon <[schacon@example.com](mailto:schacon@example.com)>

Bạn cũng có thể tự nhận thấy rằng, Git thay thế lệnh mới với bất kỳ tên gì bạn đặt cho nó. Tuy nhiên, cũng có thể bạn muốn chạy một lệnh bên ngoài, hơn là bản thân các lệnh trong Git. Trong trường hợp này, bạn bắt đầu lệnh đó với ký tự !. Nó khá hữu ích trong trường hợp bạn viết công cụ riêng của bạn để làm việc với Git. Một ví dụ minh họa là việc tạo bí danh cho git visual để chạy gitk:

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

## Phân nhánh trong Git

ref: <https://git-scm.com/book/vi/v1/Ph%C3%A2n-Nh%C3%A1nh-Trong-Git>

Phân nhánh có nghĩa là bạn phân tách ra từ luồng phát triển chính và tiếp tục làm việc mà không sợ làm ảnh hưởng đến luồng chính.

Nhiều người nhắc đến mô hình phân nhánh của Git như là "chức năng hủy diệt", và chính nó làm cho Git trở nên khác biệt trong cộng đồng VCS. Cách Git phân nhánh nhẹ đến kinh ngạc, các hoạt động tạo nhánh xảy ra gần như ngay lập tức và việc di chuyển đi lại giữa các nhánh cũng thường rất nhanh. Không giống các VCSs khác, Git khuyến khích sử dụng rẽ nhánh và tích hợp thường xuyên cho workflow, thậm chí nhiều lần trong một ngày.

### Nhánh là gì?

Để có thể thực sự hiểu được cách phân nhánh của Git, chúng ta cần nhìn và xem xét lại cách Git lưu trữ dữ liệu. Git không lưu trữ dữ liệu dưới dạng một chuỗi các thay đổi hoặc delta, mà thay vào đó là một chuỗi các ảnh (snapshot).

Khi bạn commit, Git lưu trữ đối tượng commit mà có chứa một con trỏ tới ảnh của nội dung bạn đã tổ chức, tác giả và thông điệp hay 0 hay nhiều con trỏ khác trỏ tới một hoặc nhiều commit cha trực tiếp của commit đó: commit đầu tiên không có cha, commit bình thường có một cha, và nhiều cha cho commit là kết quả được tích hợp lại từ hai hoặc nhiều nhánh.