**MỤC LỤC**

[**I.** **GIT.** 3](#_Toc467443046)

[**1.** **Quản lý phiên bản.** 3](#_Toc467443047)

[*Hệ Thống Quản Lý Phiên Bản Cục Bộ:* 3](#_Toc467443048)

[*Hệ Thống Quản Lý Phiên Bản Tập Trung:* 4](#_Toc467443049)

[*Hệ Thống Quản Lý Phiên Bản Phân Tán:* 5](#_Toc467443050)

[**2.** **Sơ lược lịch sử của Git.** 6](#_Toc467443051)

[**3.** **Cơ bản về GIT** 7](#_Toc467443052)

[*Ảnh Chụp, Không Phải Sự Khác Biệt:* 7](#_Toc467443053)

[*Phần Lớn Thao Tác Diễn Ra Cục Bộ* 9](#_Toc467443054)

[*Git Mang Tính Toàn Vẹn:* 9](#_Toc467443055)

[*Git Chỉ Thêm Mới Dữ Liệu:* 10](#_Toc467443056)

[*Ba Trạng Thái:* 10](#_Toc467443057)

[**4.** **Tạo một kho chứa GIT.** 12](#_Toc467443058)

[*Khởi Tạo Một Kho Chứa Từ Thư Mục Cũ:* 12](#_Toc467443059)

[*Sao Chép Một Kho Chứa Đã Tồn Tại:* 13](#_Toc467443060)

[**5.** **Ghi lại thay đổi vào kho chứa.** 13](#_Toc467443061)

[**6.** **Xem lịch sử commit.** 14](#_Toc467443062)

[*Hiển Thị Lịch Sử Trên Giao Diện:* 15](#_Toc467443063)

[**7.** **Phục hồi.** 15](#_Toc467443064)

[*Thay Đổi Commit Cuối Cùng:* 16](#_Toc467443065)

[*Loại Bỏ Tập Tin Đã Tổ Chức:* 16](#_Toc467443066)

[*Phục Hồi Tập Tin Đã Thay Đổi:* 18](#_Toc467443067)

[**8.** **Làm việc từ xa.** 19](#_Toc467443068)

[Hiển Thị Máy Chủ 20](#_Toc467443069)

[Thêm Các Kho Chứa Từ Xa 20](#_Toc467443070)

[Truy Cập Và Kéo Về Từ Máy Chủ Trung Tâm 20](#_Toc467443071)

[Đẩy Lên Máy Chủ Trung Tâm 21](#_Toc467443072)

[Kiểm Tra Một Máy Chủ Trung Tâm 21](#_Toc467443073)

[Xóa Và Đổi Tên Từ Xa 22](#_Toc467443074)

[Liệt Kê Tag 22](#_Toc467443075)

[Thêm Tag Mới 23](#_Toc467443076)

[Annotated Tags 23](#_Toc467443077)

[Signed Tags 24](#_Toc467443078)

[Lightweight Tags 26](#_Toc467443079)

[Xác Thực Các Tag 27](#_Toc467443080)

[Tag Muộn 28](#_Toc467443081)

[Chia Sẻ Các Tag 29](#_Toc467443082)

[Nhánh Là Gì? 31](#_Toc467443083)

[Cơ Bản về Phân Nhánh 38](#_Toc467443084)

[Cơ Bản Về Tích Hợp 43](#_Toc467443085)

[Mâu Thuẫn Khi Tích Hợp 45](#_Toc467443086)

[**II.** **SVN.** 50](#_Toc467443087)

[**1.** **SVN là gì?** 50](#_Toc467443088)

[**2.** **SVN giải quyết được vấn đề gì?** 51](#_Toc467443089)

[**3.** **Tạo repository trong SVN** 51](#_Toc467443090)

[**4.** **Tiến trình Checkout trong SVN.** 52](#_Toc467443091)

[**5.** **Tiến trình Update trong SVN** 52](#_Toc467443092)

[**6.** **Thực hiện các thay đổi trong SVN** 53](#_Toc467443093)

[**7.** **Duyệt các thay đổi trong SVN.** 54](#_Toc467443094)

[**8.** **Sửa các lỗi trong SVN** 54](#_Toc467443095)

**GIT VÀ SVN**

1. **GIT**
2. **Quản lý phiên bản**

[*Hệ Thống Quản Lý Phiên Bản Cục Bộ*](https://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-V%E1%BB%81-Qu%E1%BA%A3n-L%C3%BD-Phi%C3%AAn-B%E1%BA%A3n#Hệ-Thống-Quản-Lý-Phiên-Bản-Cục-Bộ)*:*

Nhiều người chọn phương pháp quản lý phiên bản bằng cách copy các file sang một thư mục khác (có thể là các thư mục được đặt tên theo thời gian, nếu họ thông minh). Đây là một phương pháp rất phổ biến bởi vì nó rất đơn giản, tuy nhiên nó cũng rất dễ gây ra lỗi. Bạn sẽ rất dễ quên rằng bạn đang ở trong thư mục nào hay vô tình sửa hoặc sao chép nhầm file mà bạn không muốn.

Để giải quyết vấn đề này, từ lâu các lập trình viên đã phát triển các phiên bản VCS cục bộ có chứa một database đơn giản lưu trữ tất cả các sự thay đổi của các files dưới sự kiểm soát thay đổi (xem Hình 1-1).



*Hình 1-1. Mô hình quản lý phiên bản cục bộ.*

Một trong những hệ thống quản lý phiên bản phổ biến hơn có tên là rcs vẫn còn được sử dụng ở nhiều máy tính cho tới bây giờ. Ngay cả hệ điều hành Mac OS X nổi tiếng cũng đưa vào các lệnh rcs khi bạn cài đặt Developer Tools (Các công cụ dành cho lập trình viên). Phần mềm này cơ bản hoạt động bằng cách lưu giữ các bản vá (những sự thay đổi giữa các file) từ phiên bản này qua phiên bản khác ở một định dạng đặc biệt được lưu trên ổ cứng; nó có thể tái tạo lại bất kỳ file nào ở bất kỳ thời điểm nào bằng cách gộp tất cả các bản vá lại với nhau.

[*Hệ Thống Quản Lý Phiên Bản Tập Trung*](https://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-V%E1%BB%81-Qu%E1%BA%A3n-L%C3%BD-Phi%C3%AAn-B%E1%BA%A3n#Hệ-Thống-Quản-Lý-Phiên-Bản-Tập-Trung)*:*

Vấn đề nghiêm trọng tiếp theo mà mọi người thường mắc phải là họ cần cộng tác với các lập trình viên khác trong hệ thống. Để vượt qua trở ngại này, Hệ Thống Quản Lý Phiên Bản Tập Trung (Centralized Version Control Systems - CVCSs) được phát triển. Các hệ thống này, ví dụ như CVS, Subversion, và Perforce, bao gồm một máy chủ có chứa tất cả các tập tin đã được "phiên bản hoá" (versioned), và danh sách các máy khách có quyền thay đổi các tập tin này trên máy chủ trung tâm đó. Trong vòng nhiều năm, mô hình này đã trở thành tiêu chuẩn cho việc quản lý phiên bản (xem Hình 1-2).



*Hình 1-2. Mô hình quản lý phiên bản tập trung.*

Mô hình này cung cấp rất nhiều lợi thế, đặc biết so với việc quản lý cục bộ. Ví dụ, tất cả người dùng đều biết một phần nào đó những việc mà những người khác trong dự án đang làm. Người quản lý có quyền quản lý ai có thể làm gì theo ý muốn; và việc này dễ dàng hơn nhiều so với việc phải quản lý ở từng cơ sở dử liệu ở từng máy riêng biệt.

Tuy nhiên, mô hình này cũng có những bất cập nghiêm trọng. Dễ nhận thấy nhất đó là "sự cố tập trung" mà máy chủ trung tâm mắc phải. Nếu máy chủ đó không hoạt động trong một giờ, nghĩa là trong khoảng thời gian đó không ai có thể cộng tác với những người còn lại hoặc lưu trữ các thay đổi đã được phiên bản hoá của bất kỳ tập tin nào mà người đó đang thao tác. Nếu ổ cứng lưu trữ cơ sở dữ liệu trung tâm bị hỏng, và các sao lưu dự phòng chưa được tạo ra tính đến thời điểm đó, bạn sẽ mất toàn bộ lịch sử của dự án đó, ngoại trừ những phiên bản cục bộ mà người dùng có được trên máy tính cá nhân. Các hệ thống quản lý phiên bản cục bộ phải đối diện với vấn đề tương tự như thế này mỗi khi toàn bộ lịch sử của dự án được lưu ở một nơi, bạn có nguy cơ mất tất cả.

[*Hệ Thống Quản Lý Phiên Bản Phân Tán*](https://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-V%E1%BB%81-Qu%E1%BA%A3n-L%C3%BD-Phi%C3%AAn-B%E1%BA%A3n#Hệ-Thống-Quản-Lý-Phiên-Bản-Phân-Tán)*:*

Đã tới lúc cần tới các Hệ Thống Quản Lý Phiên Bản Phân Tán - Distributed Version Control Systems (DVCSs). Trong các DVCS (ví dụ như Git, Mercurial, Bazaar hay Darcs), các máy khách không chỉ "check out" (sao chép về máy cục bộ) phiên bản mới nhất của các tập tin: chúng sao chép (mirror) toàn bộ kho chứa (repository). Chính vì vậy nếu như một máy chủ nào mà các hệ thống quản lý phiên bản này (mỗi máy khách là một hệ thống riêng biệt) đang cộng tác ngừng hoạt động, thì kho chứa từ bất kỳ máy khách nào cũng có thể dùng để sao chép ngược trở lại máy chủ để khôi phục lại toàn bộ hệ thống. Mỗi checkout thực sự là một bản sao đầy đủ của tất cả dữ liệu (xem Hình 1-3).

1. **Sơ lược lịch sử của Git.**

Cũng như nhiều thứ tuyệt vời khác trong cuộc sống, Git ra đời từ một chút của sự huỷ diệt/phá sản/kết thúc có tính sáng tạo và sự tranh cãi nảy lửa. Nhân của Linux là một dự án phần mềm mã nguồn mở của một phạm vi khá lớn. Trong phần lớn thời gian bảo trì của nhân Linux (1991-2002), các thay đổi của phần mềm được truyền đi dưới dạng các bản vá và các tập tin lưu trữ. Vào năm 2002, dự án nhân Linux bắt đầu sử dụng một DVCS độc quyền có tên là BitKeeper.

Vào năm 2005, sự hợp tác giữa cộng đồng phát triển nhân Linux và công ty thương mại phát triển BitKeeper bị phá vỡ, và công cụ đó không còn được cung cấp miễn phí nữa. Chính điều này đã thúc đẩy cộng đồng phát triển Linux (chính xác hơn là Linus Torvalds, người sáng lập ra Linux) phát triển công cụ của riêng họ dựa trên những bài học từ việc sử dụng BitKeeper. Một số mục tiêu của hệ thống mới được vạch ra như sau: nhanh, thiết kế đơn giản, hỗ trợ tốt cho phát triển phi tuyến tính, phân tán toàn diện, có khả năng xử lí các dự án lớn giống như nhân Linux một cách hiệu quả.

Kể từ khi ra đời năm 2005, Git đã tiến hoá và phát triển toàn diện để dễ dàng sử dụng hơn, tuy thế các tiêu chí ban đầu vẫn được đảm bảo. Nó nhanh một cách đáng kinh ngạc, vô cùng hiệu quả với các dự án lớn, và một hệ thống phân nhánh không thể tin được cho phát triển phi tuyến tính



*Hình 1-3. Mô hình quản lý phiên bản phân tán.*

Ngoài ra, phần lớn các hệ thống này xử lý rất tốt việc quản lý nhiều kho chứa từ xa, vì thế bạn có thể cộng tác với nhiều nhóm người khác nhau theo những cách khác nhau trong cùng một dự án. Việc này cho phép bạn cài đặt nhiều loại "tiến trình công việc" (workflow) không thể thực hiện được với các hệ thống tập trung, ví dụ như các mô hình phân cấp.

1. **Cơ bản về GIT**

[*Ảnh Chụp, Không Phải Sự Khác Biệt*](https://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%C6%A1-B%E1%BA%A3n-v%E1%BB%81-Git#Ảnh-Chụp,-Không-Phải-Sự-Khác-Biệt)*:*

Sự khác nhau cơ bản giữa Git với bất kỳ VCS nào khác (bao gồm Subversion và tương tự là cách Git "nghĩ" về dữ liệu. Về mặt lý thuyết mà nói, phần lớn hệ thống khác lưu trữ thông tin dưới dạng danh sách các tập tin được thay đổi. Các hệ thống này (CVS, Subversion, Perforce, Bazaar,...) 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, được minh hoạ trong hình 1-4.



*Hình 1-4. Các hệ thống khác hướng tới lưu trữ tập tin dưới dạng các thay đổi so với bản cơ sở của mỗi tập tin.*

Git không nghĩ hoặc xử lý dữ liệu theo cách này. Mà thay vào đó Git coi dữ liệu của nó giống như một tập hợp các "ảnh" (snapshot) của một hệ thống tập tin nhỏ. Mỗi lần bạn "commit", hoặc lưu lại trạng thái hiện tại của dự án trong Git, về cơ bản Git "chụp một bức ảnh" ghi lại nội dung 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. Git thao tác với dữ liệu giống như Hình 1-5.



*Hình 1-5. Git lưu trữ dữ liệu dưới dạng ảnh chụp của dự án theo thời gian.*

Đây là sự khác biệt lớn nhất giữa Git và hầu hết các VCS khác. Nó khiến Git cân nhắc lại hầu hết các khía cạnh của quản lý phiên bản mà phần lớn các hệ thống khác chỉ áp dụng lại từ các thế hệ trước. Chính lý do này làm cho Git giống như một hệ thống quản lý tập tin thu nhỏ với các tính năng, công cụ vô cùng mạnh mẽ được xây dựng dựa trên nó, không chỉ là một hệ thống quản lý phiên bản đơn giản. Chúng ta sẽ khám phá một số lợi ích đạt được từ việc quản lý dữ liệu theo cách này khi bàn luận về Phân nhánh trong Git ở Chương 3.

[*Phần Lớn Thao Tác Diễn Ra Cục Bộ*](https://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%C6%A1-B%E1%BA%A3n-v%E1%BB%81-Git#Phần-Lớn-Thao-Tác-Diễn-Ra-Cục-Bộ)

Phần lớn các thao tác/hoạt động trong Git chỉ cần yêu cầu các tập tin hay tài nguyên cục bộ - thông thường nó sẽ không cần bất cứ thông tin từ máy tính nào khác trong mạng lưới của bạn. Nếu như bạn quen với việc sử dụng các hệ thống quản lý phiên bản tập trung nơi mà đa số hoạt động đều chịu sự ảnh hưởng bởi độ trễ của mạng, thì với Git đó lại là một thế mạnh. Bởi vì toàn bộ dự án hoàn toàn nằm trên ổ cứng của bạn, các thao tác được thực hiện gần như ngay lập tức.

Ví dụ, khi bạn muốn xem lịch sử của dự án, Git không cần phải lấy thông tin đó từ một máy chủ khác để hiển thị, mà đơn giản nó được đọc trực tiếp từ chính cơ sở dữ liệu cục bộ của bạn. Điều này có nghĩa là bạn có thể xem được lịch sử thay đổi của dự án gần như ngay lập tức. Nếu như bạn muốn so sánh sự thay đổi giữa phiên bản hiện tại của một tập tin với phiên bản của một tháng trước, Git có thể tìm kiếm tập tin cũ đó trên máy cục bộ rồi sau đó so sánh sự khác biệt cho bạn. Thay vì việc phải truy vấn từ xa hoặc "kéo về" (pull) phiên bản cũ của tập tin đó từ máy chủ trung tâm rồi mới thực hiện so sánh cục bộ.

Điều này còn đồng nghĩa với có rất ít việc mà bạn không thể làm được khi không có kết nối Internet hoặc VPN bị ngắt. Nếu bạn muốn làm việc ngay cả khi ở trên máy bay hoặc trên tầu, bạn vẫn có thể commit bình thường cho tới khi có kết nối Internet để đồng bộ hoá. Nếu bạn đang ở nhà mà VPN lại không thể kết nối được, bạn cũng vẫn có thể làm việc bình thường. Trong rất nhiều hệ thống khác, việc này gần như là không thể hoặc rất khó khăn. Ví dụ trong Perforce, bạn gần như không thể làm gì nếu như không kết nối được tới máy chủ; trong Subversion và CVS, bạn có thể sửa tập tin nhưng bạn không thể commit các thay đổi đó vào cơ sở dữ liệu (vì cơ sở dữ liệu của bạn không được kết nối). Đây có thể không phải là điều gì đó lớn lao, nhưng bạn sẽ ngạc nhiên về sự thay đổi lớn mà nó có thể làm được.

[*Git Mang Tính Toàn Vẹn*](https://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%C6%A1-B%E1%BA%A3n-v%E1%BB%81-Git#Git-Mang-Tính-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 và về mặt triết học được coi là toàn vẹn. 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. Một mã băm SHA-1 có định dạng như sau:

24b9da6552252987aa493b52f8696cd6d3b00373

Bạn sẽ thấy các mã băm được sử dụng ở mọi nơi trong Git. Thực tế, Git không sử dụng tên của các tập để lưu trữ mà bằng các mã băm từ nội dung của tập tin vào một cơ sở dữ liệu có thể truy vấn được.

[*Git Chỉ Thêm Mới Dữ Liệu*](https://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%C6%A1-B%E1%BA%A3n-v%E1%BB%81-Git#Git-Chỉ-Thêm-Mới-Dữ-Liệu)*:*

Khi bạn thực hiện các hành động trong Git, phần lớn tất cả 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 xoá 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.

Điều này khiến việc sử dụng Git trở nên thích thú bởi vì chúng ta biết rằng chúng ta có thể thử nghiệm mà không lo sợ sẽ phá hỏng mọi thứ. Để có thể hiểu sâu hơn việc Git lưu trữ dữ liệu như thế nào hay làm sao để khôi phục lại dữ liệu có thể đã mất, xem Chương 9.

[*Ba Trạng Thái*](https://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%C6%A1-B%E1%BA%A3n-v%E1%BB%81-Git#Ba-Trạng-Thái)*:*

Bây giờ, hãy chú ý. Đây là điều quan trọng cần ghi nhớ về Git nếu như bạn muốn hiểu được những phần tiếp theo một cách trôi chảy. Mỗi tập tin trong Git được quản lý dựa trên ba trạng thái: committed, modified, 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. Và 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).



*Hình 1-6. Thư mục làm việc, khu vực khán đài, và thư mục Git.*

Thư mục Git là nơi Git lưu trữ các "siêu dữ kiện" (metadata) và cơ sở dữ liệu cho dự án của bạn. Đây là phần quan trọng nhất của Git, nó là phần được sao lưu về khi bạn tạo một bản sao (clone) của một kho chứa từ một máy tính khác.

Thư mục làm việc là bản sao một phiên bản của dự án. Những tập tin này được kéo về (pulled) từ cơ sở dữ liệu được nén lại trong thư mục Git và lưu trên ổ cứng cho bạn sử dụng hoặc chỉnh sửa.

Khu vực khán đài là một tập tin đơn giản được chứa trong thư mục Git, nó chứa thông tin về những gì sẽ được commit trong lần commit sắp tới. Nó còn được biết đến với cái tên "chỉ mục" (index), nhưng khu vực tổ chức (staging area) đang dần được coi là tên tiêu chuẩn.

Tiến trình công việc (workflow) cơ bản của Git:

1. Bạn thay đổi các tập tin trong thư mục làm việc.
2. Bạn tổ chức các tập tin, tạo mới ảnh của các tập tin đó vào khu vực tổ chức.
3. Bạn commit, ảnh của các tập tin trong khu vực tổ chức sẽ được lưu trữ vĩnh viễn vào thư mục Git.

Nếu một phiên bản nào đó của một tập tin ở trong thư mục Git, nó được coi là đã commit. Nếu như nó đã được sửa và thêm vào khu vực tổ chức, nghĩa là nó đã được staged. Và nếu nó được thay đổi từ khi checkout nhưng chưa được staged, nó được coi là đã thay đổi. Trong Chương 2, bạn sẽ được tìm hiểu kỹ hơn về những trạng thái này cũng như làm thế nào để tận dụng lợi thế của chúng hoặc bỏ qua hoàn toàn giai đoạn tổ chức (staged).

1. **Tạo một kho chứa GIT**

### [*Khởi Tạo Một Kho Chứa Từ Thư Mục Cũ*](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-T%E1%BA%A1o-M%E1%BB%99t-Kho-Ch%E1%BB%A9a-Git#Khởi-Tạo-Một-Kho-Chứa-Từ-Thư-Mục-Cũ)*:*

Nếu như 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) hết. (Xem Chương 9 để biết chính xác những tập tin gì có trong thư mục .git bạn vừa tạo.)

Nếu bạn muốn kiếm soát phiên bản cho các tập tin có sẵn (đối lập 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:

$ git add \*.c

$ git add README

$ git commit -m 'phiên bản đầu tiên/khởi tạo của dự án'

Chúng ta sẽ xem những lệnh này thực hiện những gì trong chốc lát nữa. Bâu giờ thì bạn đã có một kho chứ Git với các tập tin đã được theo dõi và một lần commit đầu tiên.

### [*Sao Chép Một Kho Chứa Đã Tồn Tại*](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-T%E1%BA%A1o-M%E1%BB%99t-Kho-Ch%E1%BB%A9a-Git#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. Nếu như bạn đã quen thuộc với các hệ thống VCS khác như là Subversion, bạn sẽ nhận ra rằng câu lệnh này là clone chứ không phải checkout. Đây là một sự khác biệt lớn - Git nhận một bản sao của gần như tất cả dữ liệu mà máy chủ đang có. Mỗi phiên bản của mỗi tập tin sử dụng cho lịch sử của dự án được kéo về khi bạn chạy git clone. Thực tế, nếu ổ cứng máy chủ bị hư hỏng, bạn có thể sử dụng bất kỳ bản sao trên bất kỳ máy khách nào để khôi phục lại trạng thái của máy chủ khi nó được sao chép (bạn có thể mất một số tập tin phía máy chủ, nhưng tất cả phiên bản của dữ liệu vẫn tồn tại ở đó - xem chi tiết ở Chương 4).

Sử dụng lệnh git clone [url] để sao chép một kho chứa. Ví dụ, nếu bạn muốn tạo một bản sao của thư viện Ruby Git có tên Grit, bạn có thể thực hiện như sau:

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

Một thư mục mới có tên grit sẽ được tạo, kèm theo thư mục .git và bản sao mới nhất của tất cả dữ liệu của kho chứa đó bên trong. Nếu bạn xem bên trong thư mục grit, bạn sẽ thấy các tập tin của dự án bên trong, và đã sẵn sàng cho bạn làm việc hoặc sử dụng. Nếu bạn muốn sao chép kho chứa này vào một thư mục có tên khác không phải là grit, bạn có thể chỉ định tên thư mục đó như là một tuỳ chọn tiếp theo khi chạy dòng lệnh:

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

Lệnh này thực thi tương tự như lệnh trước, nhưng thư mục của kho chứa lúc này sẽ có tên là 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.gitthông qua giao thức SSH. Chương 4 sẽ giới thiệu tất cả các tuỳ chọn áp dụng trên máy chủ để nó có thể truy cập vào kho chứa Git của bạn cũng như từng ưu và nhược điểm riêng của chúng.

1. **Ghi lại thay đổi vào kho chứa**

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.

Hãy nhớ là mỗi tập tin trong thư mục làm việc của bạn có thể ở một trong hai trạng thái : *tracked* hoặc*untrachked*. 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 mới 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 trong Hình 2-1.



*Hình 2-1. Vòng đời các trạng thái của tập tin.*

1. **Xem lịch sử 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 liệu lực tốt nhất là sử dụng lệnhgit log.

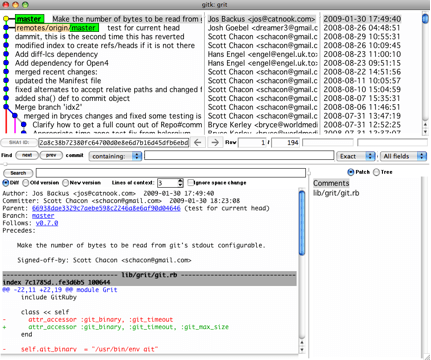
Mặc định, không sử dụng tham số nào, git log liệt kê các commit được thực hiện trong kho chứa đó theo thứ tự thời gian. Đó là, commit mới nhất được hiển thị đầu tiên. Như bạn có thể thấy, lệnh này liệt kê từng commit với mã băm SHA-1, tên người commit, địa chỉ email, ngày lưu, và thông điệp của chúng.

Có rất nhiều tuỳ chọn (tham biến/số) khác nhau cho lệnh git log giúp bạn tìm chỉ hiện thị thứ mà bạn thực sự muốn. Ở đây, chúng ta sẽ cùng xem qua các lựa chọn phổ biến, thường được sử dụng nhiều nhất.

Một trong các tuỳ chọn hữu ích nhất là -p, nó hiện thị diff của từng commit. Bạn cũng có thể dùng -2 để giới hạn chỉ hiển thị hai commit gần nhất

### [*Hiển Thị Lịch Sử Trên Giao Diện*](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#Hiển-Thị-Lịch-Sử-Trên-Giao-Diện)*:*

Nếu bạn muốn sử dụng một công cụ đồ hoạ để trực quan hoá lịch sử commit, bạn có thể thử một chương trình Tcl/Tk có tên gitk được xuất bản kèm với git. Gitk cơ bản là một công cụ git log trực quan, nó chấp nhận hầu hết các lựa chọn để lọc mà git log thường dùng. Nếu bạn gõ gitk trên thư mục của dự án, bạn sẽ thấy giống như Hình 2-2.



*Hình 2-2. Công cụ trực quan hoá lịch sử commit gitk.*

1. **Phục hồi**

Tại 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*](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-Ph%E1%BB%A5c-H%E1%BB%93i#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*](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-Ph%E1%BB%A5c-H%E1%BB%93i#Loại-Bỏ-Tập-Tin-Đã-Tổ-Chức)*:*

Hai phần tiếp theo sẽ minh hoạ cho bạn thấy làm sao để thoả 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 thể phục hồi các thay đổi. Ví dụ như, giả sự 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 thể 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.rbđược thay đổi và một lần nữa lại trở thành chưa tổ chức.

### [*Phục Hồi Tập Tin Đã Thay Đổi*](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-Ph%E1%BB%A5c-H%E1%BB%93i#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 (initialy 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 thể hủy những thay đổi vừa được thực hiện (ít nhất, phiên bản mới nhất của Git, 1.6.1 và mới hơn, hỗ trợ điều này - nếu bạn đang sử dụng phiên bản cũ hơn, chúng tôi khuyên bạn nên nâng cấp để có thể sử dụng được những các chức năng có tính khả dụng cao hơn). 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 chương sau; chúng là các phương pháp thay thế tốt hơn.

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

1. **Làm việc từ xa.**

### [Hiển Thị Máy Chủ](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-L%C3%A0m-Vi%E1%BB%87c-T%E1%BB%AB-Xa#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.

### [Thêm Các Kho Chứa Từ Xa](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-L%C3%A0m-Vi%E1%BB%87c-T%E1%BB%AB-Xa#Thêm-Các-Kho-Chứa-Từ-Xa)

Tôi đã đề cập và đưa một số ví dụ minh họa về việc thêm mới các kho chứa từ xa trong các phần trước, nhưng bây giờ chúng ta sẽ nói sâu hơn về nó. 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]

### [Truy Cập Và Kéo Về Từ Máy Chủ Trung Tâm](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-L%C3%A0m-Vi%E1%BB%87c-T%E1%BB%AB-Xa#Truy-Cập-Và-Kéo-Về-Từ-Máy-Chủ-Trung-Tâm)

Như bạn vừa thấy, để lấy dữ liệu của các dự án từ xa về, bạn có thể chạy:

$ git fetch [remote-name]

Lệnh này sẽ truy cập vào dự án từ xa đó và kéo xuống toàn bộ dữ liệu mà bạn chưa có trong đó cho bạn. 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 mà bạn có thể tích hợp hoặc kiểm tra bất kỳ thời điểm nào. (Chúng ta sẽ đề cập chi tiết hơn về nhánh là gì và sử dụng chúng như thế nào ở Chương 3.)

Nếu bạn tạo bản sao từ một kho chứa nào đó khác, lệnh này sẽ tự động kho chứa từ xa đó vào dưới tênorigin. 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 một điều quan trọng là lệnhfetch 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 không 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 (xem phần tiếp theo và Chương 3để biết thêm chi tiết), 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 nhanh 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](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-L%C3%A0m-Vi%E1%BB%87c-T%E1%BB%AB-Xa#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 orgin 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 muốn đẩy lên, thì hành động của bạn sẽ bị từ chối ngay tức khắc. Trước hết bạn 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. Xem Chương 3 để hiểu chi tiết hơn về làm thế nào để đẩy lên máy chủ trung tâm.

### [Kiểm Tra Một Máy Chủ Trung Tâm](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-L%C3%A0m-Vi%E1%BB%87c-T%E1%BB%AB-Xa#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 làorigin, 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.

### [Xóa Và Đổi Tên Từ Xa](https://git-scm.com/book/vi/v1/C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Git-L%C3%A0m-Vi%E1%BB%87c-T%E1%BB%AB-Xa#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, nếu bạn muốn đổi tên pb thànhpaul, bạn có thể dùng lệnh git remote rename:

$ 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

1. **Đánh dấu.**

### [Liệt Kê Tag](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#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 bảng chứ cái; thứ tự mà nó xuất hiện không thực sự quan trọng lắm.

Bạn cũng có thể tìm kiếm một tag sử dụng mẫu (pattern). Ví dụ, trong kho chứa mã nguồn của Git có chứa hơn 240 tag. Nếu như bạn chỉ quan tâm đến các tag thuộc dải 1.4.2, bạn có thể chạy lệnh sau:

$ 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](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#Thêm-Tag-Mới)

Git sử dụng hai loại tag chính: lightweight và annotated. Một lightweigh tag (hạng nhẹ) giống như một nhánh mà không có sự thay đổi - nó chỉ trỏ đến một commit nào đó. Annotated (chú thích) tag, thì lại 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 có chứa các thông tin trên; tuy nhiên nếu như bạn muốn một tag tạm thời hoặc vì một lý do nào đó bạn không muốn lưu trữ các thông tin trên, lightweight tag là sự lựa chọn hợp lý hơn.

### [Annotated Tags](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#Annotated-Tags)

Tạo một tag chú thích (annnotated) trong Git rất đơn giản. Cách dễ nhất là sử dụng -a khi bạn chạy lệnhtag:

$ 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 nội dung/thông điệp cho tag. Nếu như 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 hiện thị thông tin người tag, ngày commit được tag, và thông báo chú thích trước khi hiện thông tin của commit.

### [Signed Tags](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#Signed-Tags)

Bạn cũng có thể ký các tag của bạn sử dụng GPG, giải 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'

Một lát nữa, bạn sẽ được học làm sao để kiểm tra/xác minh (verify) các tag đã được ký.

### [Lightweight Tags](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#Lightweight-Tags)

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

Lần này, nếu bạn chạy git show trên tag đó, bạn sẽ không thấy các thông tin bổ sung nữa. 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'

### [Xác Thực Các Tag](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#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

Nếu như bạn không có public key của người ký, bạn sẽ thấy thông báo như sau:

gpg: Signature made Wed Sep 13 02:08:25 2006 PDT using DSA key ID F3119B9A

gpg: Can't check signature: public key not found

error: could not verify the tag 'v1.4.2.1'

### [Tag Muộn](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#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 "updated rakefile". Bạn vẫn có thể thêm tag vào lúc này. Để làm được điều bạn 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

Bạn có thể thấy là commit đã được tag:

$ git tag

v0.1

v1.2

v1.3

v1.4

v1.4-lw

v1.5

$ git show v1.2

tag v1.2

Tagger: Scott Chacon <schacon@gee-mail.com>

Date: Mon Feb 9 15:32:16 2009 -0800

version 1.2

commit 9fceb02d0ae598e95dc970b74767f19372d61af8

Author: Magnus Chacon <mchacon@gee-mail.com>

Date: Sun Apr 27 20:43:35 2008 -0700

updated rakefile

...

### [Chia Sẻ Các Tag](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#Chia-Sẻ-Các-Tag)

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

$ git push origin v1.5

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] v1.5 -> v1.5

Nếu bạn có rất nhiều tag muốn đẩy lên cùng một lúc, bạn có thể sử dụng tham số --tags cho lệnhgit 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

1. **Phân nhánh trong GIT**

## Nhánh Là Gì?

Để có thể thực sử hiểu được cách phân nhánh của Git, chúng ta cần nhìn và xem xét lại cách Git lưu trữ dữ liệu. Như bạn đã biết từ Chương 1, 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 (stage), tác giả và thông điệp, hay 0 hoặc 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.

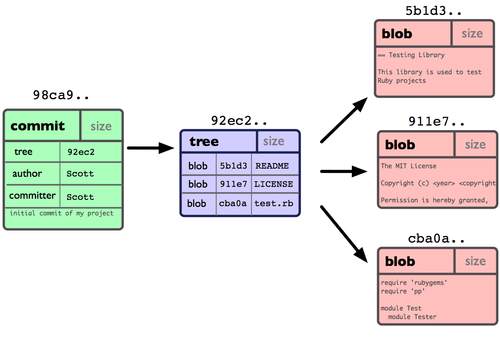
Để hình dung ra vấn đề này, hãy giả sử bạn có một thư mục chứa ba tập tin, và bạn tổ chức tất cả chúng để commit. Quá trình tổ chức các tập tin sẽ thực hiện băm từng tập (sử dụng mã SHA-1 được đề cập ở Chương 1), lưu trữ phiên bản đó của tập tin trong kho chứa Git (Git xem chúng như là các blob), và thêm mã băm đó vào khu vực tổ chức:

$ git add README test.rb LICENSE

$ git commit -m 'initial commit of my project'

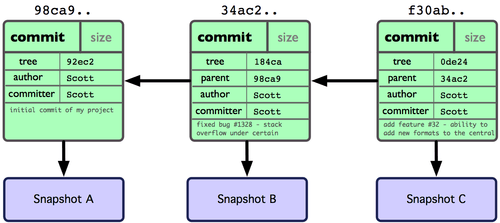
Lệnh git commit khi chạy sẽ băm tất cả các thư mục trong dự án và lưu chúng lại dưới dạng đối tượngtree. Sau đó Git tạo một đối tượng commit có chứa các thông tin mô tả (metadata) và một con trỏ trỏ tới đối tương tree gốc của dự án vì thế nó có thể tạo lại ảnh đó khi cần thiết.

Kho chứa Git của bạn bây giờ có chứa năm đối tượng: một blob cho nội dung của từng tập tin, một "cây" liệt kê nội dung của thư mục và chỉ rõ tên tập tin nào được lưu trữ trong blob nào, và một commit có con trỏ trỏ tới cây gốc và tất cả các thông tin mô tả commit. Về mặt lý thuyết, dữ liệu trong kho chứa Git có hình dạng như trong Hình 3-1.



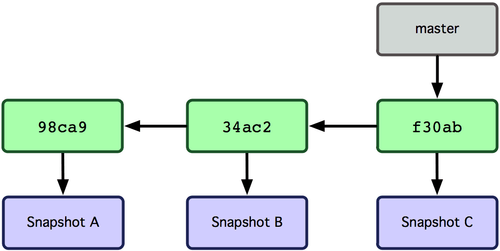
*Hình 3-1. Dữ liệu trong kho chứa với một commit.*

Nếu bạn thực hiện một số thay đổi và commit lại thì commit tiếp theo sẽ lưu một con trỏ tới commit ngay trước nó. Sau hai commit, lịch sử của dự án sẽ tương tự như trong Hình 3-2.



*Hình 3-2. Các đối tượng dữ liệu của Git trong kho chứa nhiều commit.*

Một nhánh trong Git đơn thuần là một con trỏ có khả năng di chuyển được, trỏ đến một trong những commit này. Tên nhánh mặc định của Git là master. Như trong những lần commit đầu tiên, chúng đều được trỏ tới nhánh master. Và mỗi lần bạn thực hiện commit, nó sẽ được tự động ghi vào theo hướng tiến lên. (move forward)

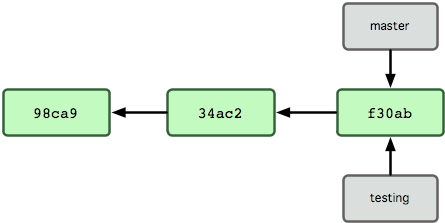


*Hình 3-3. Nhánh trỏ tới dữ liệu commit.*

Chuyện gì xảy ra nếu bạn tạo một nhánh mới? Làm như vậy sẽ tạo ra một con trỏ mới cho phép bạn di chuyển vòng quanh. Ví dụ bạn tạo một nhánh mới có tên testing. Việc này được thực hiện bằng lệnh git branch:

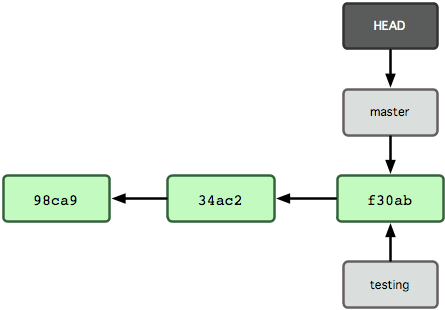
$ git branch testing

Nó sẽ tạo một con trỏ mới, cùng trỏ tới commit hiện tại (mới nhất) của bạn (xem Hình 3-4).



*Hình 304. Nhiều nhánh cùng trỏ vào dữ liệu commit.*

Vậy làm sao Git có thể biết được rằng bạn đang làm việc trên nhánh nào? Git giữ một con trỏ đặc biệt có tên HEAD. Lưu ý khái niệm về HEAD ở đây khác biệt hoàn toàn với các VCS khác mà bạn có thể đã sử dụng qua, như là Subversion hoặc CVS. Trong Git, đây là một con trỏ tới nhánh nội bộ mà bạn đang làm việc. Trong trường hợp này, bạn vẫn đang trên nhánh master. Lệnh git branch chỉ tạo một nhánh mới chứ không tự chuyển sang nhánh đó cho bạn (xem Hình 3-5).

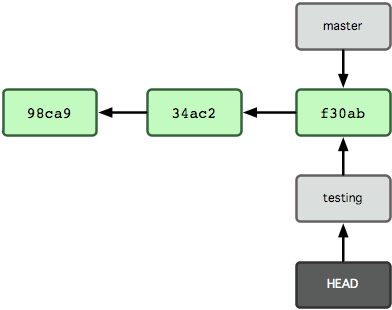


*Hình 3-5. Tập tin HEAD trỏ tới nhánh mà bạn đang làm việc.*

Để chuyển sang một nhánh đang tồn tại, bạn sử dụng lệnh git checkout. Hãy cùng chuyển sang nhánh testing mới:

$ git checkout testing

Lệnh này sẽ chuyển con trỏ HEAD sang nhánh testing (xem Hình 3-6).



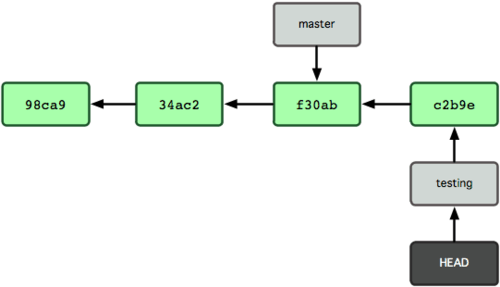
*Hình 3-6. HEAD trỏ tới nhánh khác khi bạn chuyển nhánh.*

Ý nghĩa của việc này là gì? Hãy cùng thực hiện một commit khác:

$ vim test.rb

$ git commit -a -m 'made a change'

Hình 3-7 minh họa kết quả.

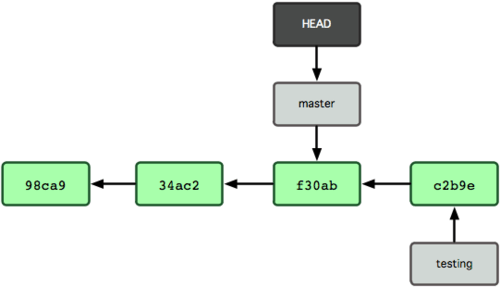


*Hình 3-7. Nhánh mà HEAD trỏ tới di chuyển tiến lên phía trước theo từng commit.*

Điều này thật thú vị, bởi vì nhánh testing của bạn bây giờ đã tiển hẳn lên phía trước, nhưng nhánh masterthì vẫn trỏ tới commit ở thời điểm khi bạn chạy lệnh git checkout để chuyển nhánh. Hãy cùng chuyển trở lại nhánh master:

$ git checkout master

Hình 3-8 hiển thị kết quả.



*Hình 3-8. HEAD chuyển sang nhánh khác khi checkout.*

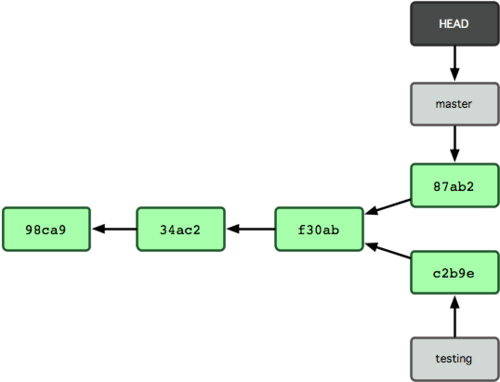
Lệnh này vừa thực hiện hai việc. Nó di chuyển lại con trỏ về nhánh master, và sau đó nó phục hồi lại các tập tin trong thư mục làm việc của bạn trở lại snapshot mà master trỏ tới. Điều này cũng có nghĩa là các thay đổi bạn thực hiện từ thời điểm này trở đi sẽ tách ra so với phiên bản cũ hơn của dự án. Nó "tua lại" các thay đổi cần thiết mà bạn đã thực hiện trên nhánh testing một cách tạm thời để bạn có thể đi theo một hướng khác.

Hãy cùng tạo một vài thay đổi và commit lại một lần nữa:

$ vim test.rb

$ git commit -a -m 'made other changes'

Bây giờ lịch sử của dự án đã bị tách ra (xem Hình 3-9). Bạn tạo mới và chuyển sang một nhánh, thực hiện một số thay đổi trên đó, và rồi chuyển ngược lại nhánh chính và tạo thêm các thay đổi khác. Cả hai sự thay đổi này bị cô lập với nhau ở hai nhánh riêng biệt: bạn có thể chuyển đi hoặc lại giữa cách nhánh và tích hợp chúng lại với nhau khi cần thiết. Và bạn đã thực hiện những việc trên một cách đơn giản với lệnh branch vàcheckout.



*Hình 3-9. Lịch sử các nhánh đã bị phân tách.*

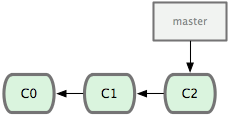
Bởi vì một nhánh trong Git thực tế là một tập tin đơn giản chứa một mã băm SHA-1 có độ dài 40 ký tự của commit mà nó trỏ tới, chính vì thế tạo mới cũng như hủy các nhánh đi rất đơn giản. Tạo mới một nhánh nhanh tương đương với việc ghi 41 bytes vào một tập tin (40 ký tự cộng thêm một dòng mới).

Điều này đối lập rất lớn với cách mà các VCS khác phân nhánh, chính là copy toàn bộ các tập tin hiện có của dự án sang một thư mục thứ hai. Việc này có thể mất khoảng vài giây, thậm chí vài phút, phụ thuộc vào dung lượng của dự án, trong khi đó trong Git thì quá trình này luôn xảy ra ngay lập tức. Thêm một lý do nữa là, chúng ta đang lưu trữ cha của các commit, nên việc tìm kiếm gốc/cơ sở để tích hợp lại được thực hiện một cách tự động và rất dễ dàng. Những tính năng này giúp khuyến khích các lập trình viên tạo và sử dụng nhánh thường xuyên hơn.

1. **Cơ bản về nhánh và tích hợp.**

### [Cơ Bản về Phân Nhánh](https://git-scm.com/book/vi/v1/Ph%C3%A2n-Nh%C3%A1nh-Trong-Git-C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Ph%C3%A2n-Nh%C3%A1nh-v%C3%A0-T%C3%ADch-H%E1%BB%A3p#Cơ-Bản-về-Phân-Nhánh)

Đầu tiên, giả sử bạn đang làm việc trên một dự án đã có một số commit từ trước (xem Hình 3-10).



*Hình 3-10. Một lịch sử commit ngắn và đơn giản.*

Bạn quyết định sẽ giải quyết vấn đề số #53 sử dụng bất kỳ hệ thống giám sát vấn đề (issue-tracking) nào mà công ty bạn đang dùng. Để cho rõ ràng, Git không cung cấp kèm bất kỳ hệ thống giám sát vấn đề nào; nhưng bởi vì vấn đề số #53 là cái mà bạn sẽ tập trung vào nên bạn sẽ tạo một nhánh mới để làm việc trên đó. Để tạo một nhánh và chuyển sang nhánh đó đồng thời, bạn có thể chạy lệnh git checkout với tham số -b:

$ git checkout -b iss53

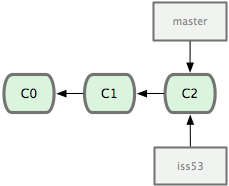
Switched to a new branch "iss53"

Đây là cách sử dụng vắn tắt của:

$ git branch iss53

$ git checkout iss53

Hình 3-11 minh họa kết quả.

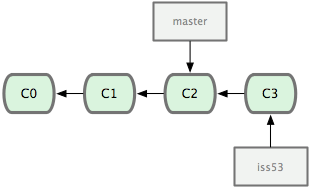


*Hình 3-11. Tạo con trỏ nhánh mới.*

Bạn làm việc trên đó và sau đó thực hiện một số commit. Làm như vậy sẽ khiến nhánh iss53 di chuyển tiến lên, vì bạn đã checkout nó (hay, HEAD đang trỏ đến nó; xem Hình 3-12):

$ vim index.html

$ git commit -a -m 'added a new footer [issue 53]'



*Hình 3-12. Nhánh iss53 đã di chuyển tiến lên cùng với thay đổi của bạn.*

Bây giờ bạn nhận được thông báo rằng có một vấn đề với trang web, và bạn cần khắc phục nó ngay lập tức. Với Git, bạn không phải triển khai bản vá lỗi cùng với các thay đổi bạn đã thực hiện trên nhánh iss53, và bạn không phải tốn quá nhiều công sức để khôi phục lại các thay đổi đó trước khi áp dụng bản vá vào sản xuất. Tất cả những gì bạn cần phải làm là chuyển lại nhánh master.

Tuy nhiên, trước khi làm điều này, bạn nên lưu ý rằng nếu thư mục làm việc hoặc khu vực tổ chức có chứa các thay đổi chưa được commit mà xung đột với nhánh bạn đang làm việc, Git sẽ không cho phép bạn chuyển nhánh. Tốt nhất là bạn nên ở trạng thái làm việc "sạch" (đã commit hết) trước khi chuyển nhánh. Có các cách khác để khắc phục vấn đề này (đó là stashing và sửa commit) mà chúng ta sẽ bàn tới sau. Hiện tại, bạn đã commit hết các thay đổi, vì vậy bạn có thể chuyển lại nhánh master:

$ git checkout master

Switched to branch "master"

Tại thời điểm này, thư mục làm việc của dự án giống hệt như trước khi bạn bắt đầu giải quyết vấn đề #53, và bạn có thể tập trung vào việc sửa lỗi. Điểm quan trọng cần ghi nhớ: Git khôi phục lại thư mục làm việc của bạn để nó giống như snapshot của commit mà nhánh bạn đang làm việc trỏ tới. Nó thêm, xóa, và sửa các tập tin một cách tự động để đảm bảo rằng thư mục làm việc của bạn giống như lần commit cuối cùng.

Tiếp theo, bạn có mỗi lỗi cần phải sửa. Hãy tạo mỗi nhánh để làm việc này cho tới khi nó được hoàn thành (xem Hình 3-13):

$ git checkout -b hotfix

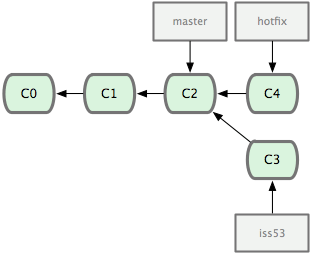
Switched to a new branch "hotfix"

$ vim index.html

$ git commit -a -m 'fixed the broken email address'

[hotfix]: created 3a0874c: "fixed the broken email address"

1 files changed, 0 insertions(+), 1 deletions(-)



*Hình 3-13. Nhánh hotfix dựa trên nhánh master.*

Bạn có thể chạy để kiểm tra, để chắc chắn rằng bản vá lỗi hoạt động đúng theo ý bạn muốn, và sau đó tích hợp nó lại nhánh chính để triển khai. Bạn có thể làm sử dụng lệnh git merge để làm việc này:

$ git checkout master

$ git merge hotfix

Updating f42c576..3a0874c

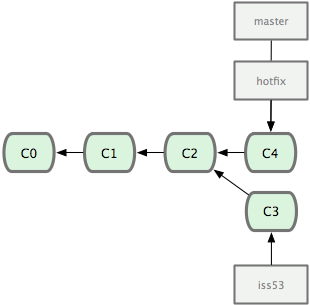
Fast forward

README | 1 -

1 files changed, 0 insertions(+), 1 deletions(-)

Bạn sẽ nhận thấy rằng cụm từ "Fast forward" trong lần tích hợp đó. Bởi vì commit được trở tới bởi nhánh mà bạn tích hợp vào lại trực tiếp là upstream của commit hiện tại, vì vậy Git di chuyển con trỏ về phía trước. Nói cách khác, khi bạn cố gắng tích hợp một commit với một commit khác mà có thể truy cập được từ lịch sử của commit trước thì Git sẽ đơn giản hóa bằng cách di chuyển con trỏ về phía trước vì không có sự rẽ nhánh nào để tích hợp - đây được gọi là "fast forward".

Thay đổi của bạn bây giờ ở trong snapshot của commit được trỏ tới bởi nhánh master, và bạn có thể triển khai thay đổi này (xem Hình 3-14).



*Hình 3-14. Nhánh master và nhánh hotfix cùng trỏ tới một điểm sau khi tích hợp.*

Sau khi triển khai xong bản vá lỗi quan trọng đó, bạn đã sẵn sàng để quay lại với công việc bị gián đoạn trước đó. Tuy nhiên, việc đầu tiên cần làm là xóa nhánh hotfix đi, vì bạn không còn cần tới nó nữa - nhánhmaster trỏ tới cùng một điểm. Bạn có thể xóa nó đi bằng cách sử dụng tham số -d cho lệnh git branch:

$ git branch -d hotfix

Deleted branch hotfix (3a0874c).

Bây giờ bạn đã có thể chuyển lại nhánh mà bạn đang làm việc trước đó về vấn đề #53 và tiếp tục làm việc (xem Hình 3-15):

$ git checkout iss53

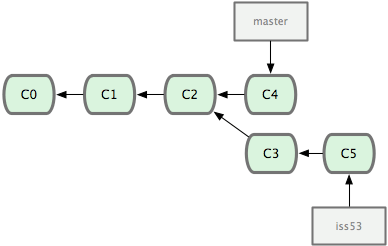
Switched to branch "iss53"

$ vim index.html

$ git commit -a -m 'finished the new footer [issue 53]'

[iss53]: created ad82d7a: "finished the new footer [issue 53]"

1 files changed, 1 insertions(+), 0 deletions(-)



*Hình 3-15. Nhánh iss53 có thể di chuyển về phía trước một cách độc lập.*

Điều đáng chú ý ở đây là những công việc bạn đã thực hiện ở nhánh hotfix không bao gồm trong nhánhiss53. Nếu bạn muốn đưa chúng vào, bạn có thể tích hợp nhánh master vào nhánh iss53 bằng cách chạy lệnh git merge master, hoặc bạn có thể chờ đợi đến khi bạn quyết định tích hợp nhánh iss53ngược trở lại nhánh master về sau.

### [Cơ Bản Về Tích Hợp](https://git-scm.com/book/vi/v1/Ph%C3%A2n-Nh%C3%A1nh-Trong-Git-C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Ph%C3%A2n-Nh%C3%A1nh-v%C3%A0-T%C3%ADch-H%E1%BB%A3p#Cơ-Bản-Về-Tích-Hợp)

Giả sử bạn đã quyết định việc giải quyết vấn đề #53 đã hoàn thành và sẵn sàng để tích hợp vào nhánhmaster. Để làm được điều này, bạn sẽ tích hợp nhánh iss53 lại, giống như bạn đã làm với nhánhhotfix trước đó. Tất cả những gì cần phải làm là chuyển sang (check out) nhánh mà bạn muốn được tích hợp vào và chạy lệnh git merge:

$ git checkout master

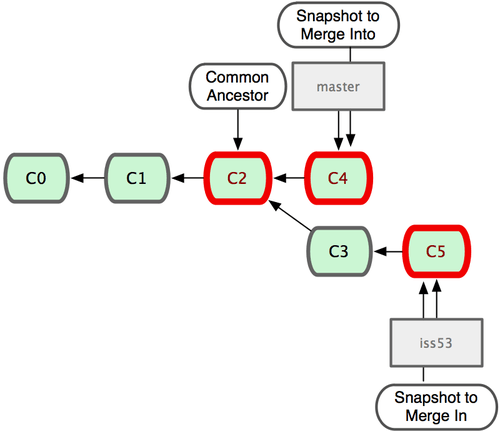
$ git merge iss53

Merge made by recursive.

README | 1 +

1 files changed, 1 insertions(+), 0 deletions(-)

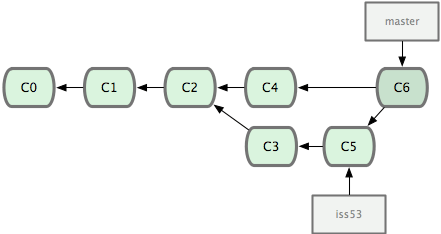
Lần này có hơi khác so với lần tích hợp hotfix trước đó. Trong trường hợp này, lịch sử phát triển của bạn đã bị phân nhánh tại một thời điểm nào đó trước kia. Bởi vì commit trên nhánh mà bạn đang làm việc (master) không phải là "cha" trực tiếp của nhánh mà bạn đang tích hợp vào, Git phải làm một số việc. Trường hợp này, Git thực hiện một tích hợp 3-chiều, sử dụng hai snapshot được trỏ tới bởi các đầu mút của nhánh và "cha chung" của cả hai. Hình 3-16 minh họa ba snapshot mà Git sử dụng để thực hiện phép tích hợp trong trường hợp này.



*Hình 3-16. Git tự động nhận dạng "cha chung" phù hợp nhất để tích hợp các nhánh lại với nhau.*

Thay vì việc chỉ di chuyển con trỏ về phía trước, Git tạo một snapshot mới - được hợp thành từ lần tích hợp 3-chiều này và cũng tự tạo một commit mới trỏ tới nó (xem Hình 3-17). Nó được biết tới như là "commit tích hợp" (merge commit) và nó đặc biệt vì có nhiều hơn một cha.

Đáng để chỉ ra rằng Git tự quyết định cha chung phù hợp nhất để sử dụng làm cơ sở cho việc tích hợp; điểm này khác với CVS hay Subversion (các phiên bản trước 1.5), khi mà các lập trình viên phải tự xác định cơ sở phù hợp nhất để tích hợp. Điều này khiến cho việc tích hợp trong Git trở nên dễ dàng hơn rất nhiều so với các hệ quản trị phiên bản khác.



*Hình 3-17. Git tự động tạo đối tượng commit mới chứa đựng các thay đổi đã tích hợp.*

Bây giờ công việc của bạn đã được tích hợp lại với nhau, bạn không cần thiết phải giữ lại nhánh iss53 nữa. Bạn có thể xóa nó đi và sau đó tự xóa vấn đề này trong hệ thống quản lý vấn đề của bạn:

$ git branch -d iss53

### [Mâu Thuẫn Khi Tích Hợp](https://git-scm.com/book/vi/v1/Ph%C3%A2n-Nh%C3%A1nh-Trong-Git-C%C6%A1-B%E1%BA%A3n-V%E1%BB%81-Ph%C3%A2n-Nh%C3%A1nh-v%C3%A0-T%C3%ADch-H%E1%BB%A3p#Mâu-Thuẫn-Khi-Tích-Hợp)

Đôi khi, quá trình này không diễn ra một cách suôn sẻ. Nếu bạn thay đổi cùng một nội dung của cùng một tập tin ở hai nhánh khác nhau mà bạn đang muốn tích hợp vào, Git không thể tích hợp chúng một cách gọn gàng. Nếu bản vá lỗi cho vấn đề #53 cùng thay đổi một phần của một tập tin giống như nhánh hotfix, bạn sẽ nhận được một sự xung đột khi tiến hành tích hợp như sau:

$ git merge iss53

Auto-merging index.html

CONFLICT (content): Merge conflict in index.html

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

Git chưa tự tạo commit tích hợp mới. Nó tạm dừng quá trình này lại cho đến khi bạn giải quyết xong xung đột. Nếu bạn muốn xem tập tin nào chưa được tích hợp tại bất kỳ thời điểm nào sau khi xung đột xảy ra, bạn có thể sử dụng lệnh git status:

[master\*]$ git status

index.html: needs merge

# On branch 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)

#

# unmerged: index.html

#

Với bất kỳ xung đột nào xảy ra mà chưa được giải quyết, chúng sẽ được liệt kê là unmerged (chưa được tích hợp). Git thêm các dấu hiệu chuẩn riêng để giải quyết xung đột vào các tập tin có xảy ra xung đột, vì thế bạn có thể mở và giải quyết các xung đột đó một cách thủ công. Tập tin của bạn sẽ chứa một phần tương tự như sau:

<<<<<<< HEAD:index.html

<div id="footer">contact : email.support@github.com</div>

=======

<div id="footer">

please contact us at support@github.com

</div>

>>>>>>> iss53:index.html

Điều này có nghĩa là phiên bản trong HEAD (nhánh master, vì nó là nhánh bạn đã check out khi chạy lệnh merge) là phần mới nhất của đoạn đó (mọi thứ phía trên =======), trong khi phiên bản ở nhánh iss53chính là phần phía dưới. Để giải quyết vấn đề này, bạn phải chọn một trong hai phần hoặc tự gộp nội dung của chúng lại. Ví dụ, có thể bạn giải quyết xung đột này bằng cách thay thế toàn bộ đoạn code đó bằng:

<div id="footer">

please contact us at email.support@github.com

</div>

Cách giải quyết này có chứa nội dung của cả hai phần, và tôi đã xóa bỏ hoàn toàn các dòng <<<<<<<,=======, và >>>>>>>. Sau khi giải quyết xong tất cả các phần này trong các tập tin bị xung đột, chạy lệnh git add cho từng tập tin để đánh dấu là chúng đã được giải quyết. Tổ chức chúng cùng đồng nghĩa với việc đánh dấu là đã được giải quyết trong Git. Nếu bạn muốn sử dụng một công cụ có giao diện đồ họa để giải quyết những vấn đề này, bạn có thể sử dụng git mergetool, Git sẽ tự động mở chương trình tương ứng và trợ giúp bạn giải quyết các xung đột:

$ git mergetool

merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff opendiff emerge vimdiff

Merging the files: index.html

Normal merge conflict for 'index.html':

{local}: modified

{remote}: modified

Hit return to start merge resolution tool (opendiff):

Nếu bạn muốn sử dụng một công cụ tích hợp khác thay vì chương trình mặc định (Git sử dụng opendiffcho tôi trong trường hợp này vì tôi đang sử dụng một máy tính Mac), bạn có thể xem danh sách các chương trình tương thích bằng cách chạy lệnh "merge tool candidates". Gõ tên chương trình bạn muốn sử dung. Trong Chương 7, chúng ta sẽ cùng bàn luận về việc làm thế nào để thay đổi giá trị mặc định này.

Sau khi thoát khỏi chương trình hỗ trợ tích hợp, Git sẽ hỏi bạn nếu tích hợp thành công. Nếu bạn trả lời đúng, nó sẽ đánh dấu tập tin đó là đã giải quyết cho bạn.

Bạn có thể chạy git status lại một lần nữa để xác thực rằng tất cả các xung đột đã được giải quyết:

$ git status

# On branch master

# Changes to be committed:

# (use "git reset HEAD <file>..." to unstage)

#

# modified: index.html

#

Nếu bạn hài lòng với điều này, và chắc chắn rằng tất cả các xung đột đã được tổ chức, bạn có thể chạy lệnh git commit để hoàn thành commit tích hợp. Thông điệp mặc định của commit có dạng như sau:

Merge branch 'iss53'

Conflicts:

index.html

#

# It looks like you may be committing a MERGE.

# If this is not correct, please remove the file

# .git/MERGE\_HEAD

# and try again.

#

Bạn có sửa lại nội dung này với các chi tiết về việc bạn đã giải quyết như thế nào nếu bạn cho rằng các thông tin đó sẽ có ích cho các thành viên khác sau này - tại sao bạn lại làm như vậy, nếu như chúng còn chưa rõ ràng.

1. **Quản lý các nhánh**

Bạn đã tạo mới, tích hợp, và xóa một số nhánh, bây giờ hãy cùng xem một số công cụ giúp việc quản lý nhánh trở nên dễ dàng hơn khi tần suất sử dụng nhánh của bạn ngày càng nhiều.

Lệnh git branch thực hiện nhiều việc hơn là chỉ tạo và xóa nhánh. Nếu bạn chạy nó không có tham số, bạn sẽ có danh sách của tất cả các nhánh hiện tại:

$ git branch

iss53

\* master

testing

Lưu ý về ký tự \* đứng trước nhánh master: nó chỉ cho bạn thấy nhánh mà bạn đang làm việc (Checkout). Có nghĩa là nếu bạn commit ở thời điểm hiện tại, thì nhánh master sẽ di chuyển tiến lên phía trước với các thay đổi mới. Để xem commit mới nhất trên từng nhánh, bạn có thể chạy lệnh git branch -v:

$ git branch -v

iss53 93b412c fix javascript issue

\* master 7a98805 Merge branch 'iss53'

testing 782fd34 add scott to the author list in the readmes

Một lựa chọn hữu ích khác để tìm ra trạng thái của các nhánh là lọc qua các nhánh bạn đã hoặc chưa tích hợp vào nhánh hiện tại. Các lựa chọn để sử dụng cho mục đích này gồm --merged và --no-merged. Để biết nhánh nào đã được tích hợp vào nhánh hiện tại, bạn có thể sử dụng git branch --merged:

$ git branch --merged

iss53

\* master

Bởi vì bạn đã tích hợp nhánh iss53 vào trước đó, bạn sẽ thấy nó ở trong danh sách này. Cách nhánh trong danh sách không có dấu \* ở phía trước thường an toàn để xóa bằng cách sử dụng git branch -d; bạn đã tích hợp các thay đổi trong đó vào một nhánh khác, vì thế bạn sẽ không hề bị mất bất cứ dữ liệu gì.

Để xem cách nhánh chứa các công việc/thay đổi chưa được tích hợp vào, bạn có thể chạy lệnh git branch --no-merged:

$ git branch --no-merged

testing

Lệnh này lại hiện thị các nhánh khác. Bởi vì chúng bao gồm các công việc mà bạn chưa tích hợp vào, xóa nó đi bằng lệnh git branch -d sẽ báo lỗi:

$ git branch -d testing

error: The branch 'testing' is not an ancestor of your current HEAD.

If you are sure you want to delete it, run 'git branch -D testing'.

Nếu bạn thực sự muốn xóa nó đi và chấp nhận mất các thay đổi, bạn có thể bắt buộc bằng cách sử dụng tham số -D, như hướng dẫn trong thông báo trên.

1. **SVN**
   1. **SVN là gì?**

Subversion (viết tắt là SVN) là một hệ thông quản lý version (version control system – VCS) được giới thiệu vào năm 2000 bởi công ty CollabNet ([http://subversion.tigris.org](http://subversion.tigris.org/)). Đây là hệ thông hỗ trợ làm việc theo nhóm rất hiệu quả.

Phần mềm:

Cho client: TortoiseSVN

Download: <http://tortoisesvn.net/>

Cho server: VisualSVN – Server

Download: <http://tortoisesvn.net/downloads.html>

Các site cung cấp dịch vụ:

Code.google.com

Sourceforge.net

* 1. **SVN giải quyết được vấn đề gì?**
* Khi một nhóm người làm việc trên cùng một project, việc nhiều người cùng chỉnh sửa nội dung của một file là điều không thể tránh khỏi. SVN cung cấp các chức năng để có thể thực hiện việc này một cách đơn giản và an toàn.
* Subversion được thiết kế với mục đích thay thế hệ thống quản lý phiên bản Concurrent Versioning System (CVS) đã cũ và có nhiều nhược điểm. Subversion có thể được sử dụng để quản lý bất cứ hệ thống phiên bản nào.
* Subversion là hệ thống quản lý source code tập trung (Centralized).
* Subversion sử dụng cho client hiện tại là TortoiseSVN ver 1.9.4 (<http://tortoisesvn.net/downloads.html>)
* Subversion quản lý tập tin và thư mục theo thời gian.
* SVN giống như một hệ thống file server mà các client có thể download và upload file một cách bình thường.
* Điểm đặc biệt của SVN là nó lưu lại tất cả những gì thay đổi trên hệ thống file: file nào đã bị thay đổi lúc nào, thay đổi như thế nào, và ai đã thay đổi nó
* SVN cũng cho phép recover lại những version cũ một cách chính xác. Các chức năng này giúp cho việc làm việc nhóm trở nên hiệu quả và an toàn hơn rất nhiều.

Thông thường, client và server kết nối thông qua mạng LAN hoặc Internet. Client và server có thể cùng chạy trên một may nếu SVN có nhiệm vụ theo vết lcihj sử của dự án do các nhà phát triển phần mềm phát triển trong nội bộ.

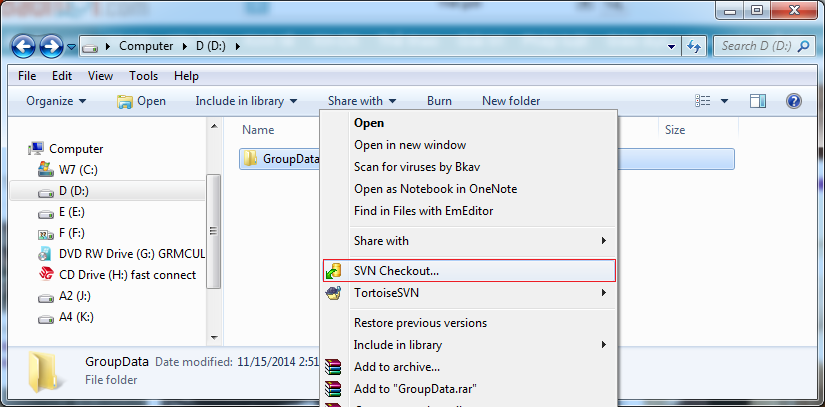
* Subversion hỗ trợ khá nhiều giao thức để kết nối giữa client và server.
* Ví dụ bạn có thể dùng giao thức của úng dụng web như http:// hoặc https://, hay các giao thức cúa SVN như svn:// hoặc svn+ssh://, hoặc nếu phần mềm client – server cài chung trên 1 máy thì có thể dùng file://.
* Việc cho phép server hỗ trợ giao thức nào phụ thuộc vào lúc cấu hình.
  1. **Tạo repository trong SVN**

Kho lưu là địa điểm trung tâm mà các lập trình viên giữ tất cả những gì liên quan đến công việc của họ. Kho lưu khong chỉ giữ file mà còn lưu giữ cả lịch sử về các thay đổi. Điều này có nghĩa là nó duy trì một lịch sử của các sự thay đổi được thực hiện trong các file.

Tiến trình **create** được sử dụng để tạo một repository mới. Hầu hết thời gian tiến trình này được thực hiện chỉ một lần. Khi bạn tạo một repository mới, VCS của bạn sẽ hi vọng bạn nói gì đó để nhận diện nó, như là nơi bạn muốn được tạo hoặc tên mà bạn muốn đặt cho repository.

* 1. **Tiến trình Checkout trong SVN**

Tiến trình **checkout** được sử dụng để tạo ra một bản sao làm việc từ repository. Bản sao làm việc là khu vực làm việc riêng nơi mà các nhà lập trình thực hiện những thay đổi của họ, và sau đó, đệ trình những thay đổi này tới repository.

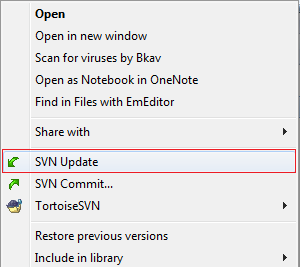


* 1. **Tiến trình Update trong SVN**

Như tên gọi đã đề cập, tiến trình **update** được sử dụng để cập nhật bản sao làm việc. Tiến trình này đồng bộ khu vực làm việc với repository. Khi repository được chia sẻ thì tất cả các thành viên trong team có thể **commit** những thay đổi của họ và bản sao làm việc trở lên cũ đi.

Bây giờ chúng ta giả sử rằng **Tom** và **Jerry** là hai nhà lập trình làm việc trên một dự án. Cả hai kiểm tra phiên bản mới nhất từ repository và bắt đầu làm việc. Tại thời điểm này, bản sao công việc là đồng bộ hoàn toàn với repository. Jerry hoàn thành công việc của anh ta rất hiệu quả và **commit** những thay đổi anh ta thực hiện tới repository.

Bây giờ khu vực làm việc của Tom chưa được cập nhật. Tiến trình **update** sẽ kéo những thay đổi mới nhất của Jerry từ repository và sẽ cập nhật khu vực làm việc của Tom.



* 1. **Thực hiện các thay đổi trong SVN**

Sau khi checkout, ai đó có thể thực hiện các tiến trình đa dạng để thực hiện các thay đổi. Chỉnh sửa là tiến trình thông thường nhất. Người nào đó có thể chỉnh sửa: thêm/gỡ bỏ nội dung file đang hiện hữu.

Một ai đó có thể thêm các file/thư mục. Nhưng ngay lập tức những file/thư mục này không không trở thành một phần của repository, trừ khi chúng đã được thêm trong quá trình diễn ra liệt kê thay đổi và trở thành một phần của repository sau tiến trình commit.

Tương tự, một ai đó có thể xóa file/thư mục. Tiến trình **delete** này ngay lập tức xóa file từ trong khu vực làm việc, nhưng sự xóa bỏ thực sự của file là được thêm vào trong quá trình diễn ra liệt kê thay đổi (pending change-list)và những thay đổi được hoàn thành tới repository sau tiến trình **commit**.

Tiến trình **rename** thay đổi tên của file/thư mục. Tiến trình **move** được sử dụng để di chuyển file/thư mục từ một địa điểm tới địa điểm khác trong cây repository.

* 1. **Duyệt các thay đổi trong SVN**

Khi bạn kiểm tra hoặc cập nhật khu vực làm việc, thì khi đó khu vực làm việc của bạn được hoàn toàn đồng bộ với repository. Nhưng khi bạn thực hiện các thay đổi tới khu vực làm việc, nó trở nên mới hơn repository. Và nó là thói quen tốt để **Review - duyệt lại** những thay đổi của bạn trước khi thực hiện tiến trình commit.

Tiến trình **status** trong SVN liệt kê các chỉnh sửa mà đã thực hiện tới khu vực làm việc. Như chúng ta đã đề cập trước đó, bất cứ khi nào bạn gây ra thay đổi trong khu vực làm việc, tất cả những thay đổi đó trở thành một phần của quá trình diễn ra liệt kê thay đổi. Và tiến trình Status được sử dụng để quan sát quá trình diễn ra liệt kê thay đổi này.

Tiến trình status chỉ cung cấp một danh sách các thay đổi nhưng không đưa ra những chi tiết về chúng. Một ai đó có thể sử dụng tiến trình **diff** để quan sát các chi tiết này của sự chỉnh sửa mà đã được thực hiện tới khu vực làm việc.

* 1. **Sửa các lỗi trong SVN**

Chúng tôi giả sử rằng một ai đó đã gây ra các thay đổi tới khu vực làm việc của anh ta, nhưng bây giờ anh ta muốn dỡ bỏ các thay đổi này. Trong tình huống này, tiến trình **revert**sẽ giúp làm điều đó.

Tiến trình **revert** trả lại các chỉnh sửa mà đã thực hiện tới khu vực làm việc. Nó là có thể để trả lại một hoặc nhiều file/thư mục. Nó cũng có thể trả lại toàn bộ khu vực làm việc. Trong trường hợp này, tiến trình revert sẽ phá hủy quá trình diễn ra liệt kê thay đổi và sẽ mang khu vực làm việc trở lại trạng thái ban đầu.

**TÀI LIỆU THAM KHẢO**

[1] <https://git-scm.com/docs>

[2] <http://tortoisesvn.net/>