**1.1 Bắt Đầu - Về Quản Lý Phiên Bản**

**Về Quản Lý Phiên Bản**

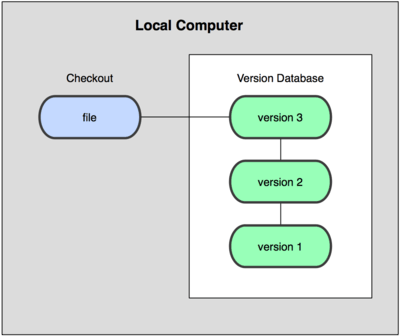
Quản lý phiên bản (mã nguồn) là gì, tại sao bạn nên quan tâm? Quản lý phiên bản là một hệ thống lưu trữ các thay đổi của một tập tin (file) hoặc tập hợp các tập tin theo thời gian, do đó nó giúp bạn có thể quay lại một phiên bản xác định nào đó sau này. Mặc dù các ví dụ trong cuốn sách này sử dụng mã nguồn của phần mềm là đối tượng cho quản lý phiên bản, song trong thực thế bất kỳ loại file nào trên máy tính cũng có thể được sử dụng cho quản lý phiên bản.

Nếu bạn là một nhà thiết kế đồ hoạ hoặc thiết kế website, bạn muốn lưu trữ tất cả các phiên bản của một bức ảnh hoặc bố cục (cái mà chắc chắn bạn cần), thì sử dụng một Hệ Thống Quản Lý phiên bản (Version Control System - VCS) là một cách làm rất khôn ngoan. Một VCS cho phép bạn: khôi phục lại phiên bản cũ của các file, khôi phục lại phiên bản cũ của toàn bộ dự án, xem lại các thay đổi đã được thực hiện theo thời gian, xem ai là người thực hiện thay đổi cuối cùng có thể gây ra sự cố, hay xem ai là người đã gây ra sự cố đó và còn nhiều hơn thế nữa. Sử dụng VCS còn đồng nghĩa với việc khi bạn làm rối tung mọi thứ lên hay vô tình xoá mất các file đi, bạn có khôi phục lại chúng một cách dễ dàng. Hơn nữa, tất cả quá trình này có thể được thực hiện rất nhanh chóng và không hề tốn quá nhiều công sức.

[**Hệ Thống Quản Lý Phiên Bản Cục Bộ**](http://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#H%E1%BB%87-Th%E1%BB%91ng-Qu%E1%BA%A3n-L%C3%BD-Phi%C3%AAn-B%E1%BA%A3n-C%E1%BB%A5c-B%E1%BB%99)

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).

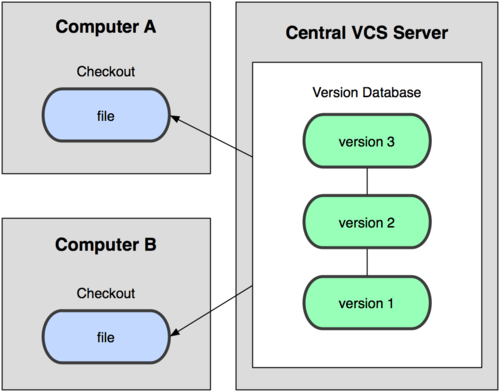


Hình 1-1. Mô hình quản lý phiên bản cục bộ.

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**](http://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#H%E1%BB%87-Th%E1%BB%91ng-Qu%E1%BA%A3n-L%C3%BD-Phi%C3%AAn-B%E1%BA%A3n-T%E1%BA%ADp-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**](http://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#H%E1%BB%87-Th%E1%BB%91ng-Qu%E1%BA%A3n-L%C3%BD-Phi%C3%AAn-B%E1%BA%A3n-Ph%C3%A2n-T%C3%A1n)

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).



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.

# 1.2 Bắt Đầu - Sơ Lược Lịch Sử của Git

## 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" (non-linear development) - (hàng ngàn nhánh song song)
* Phân tán toàn diện
* Có khả năng xử lý các dự án lớn giống như nhân Linux một cách hiệu quả (về mặt tốc độ và khối lượng dữ liệu)

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 (xem Chương 3).

**1.3 Bắt Đầu - Cơ Bản về Git**

**Cơ Bản về Git**

Tóm lại thì, Git là gì? Đây là một phần quan trọng để tiếp thu, bởi vì nếu bạn hiểu được Git là gì và các nguyên tắc cơ bản của việc Git hoạt động như thế nào, thì sử dụng Git một cách hiệu quả sẽ trở nên dễ dàng hơn cho bạn rất nhiều. 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 và Perforce; việc này sẽ giúp bạn tránh được sự hỗn độn, bối rối khi sử dụng nó. Git "nghĩ" về thông tin và lưu trữ nó khá khác biệt so với các hệ thống khác, mặc dù giao diện người dùng tương đối giống nhau; hiểu được những khác biệt đó sẽ giúp bạn tránh được rất nhiều bối rối.

[**Ảnh Chụp, Không Phải Sự Khác Biệt**](http://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#%E1%BA%A2nh-Ch%E1%BB%A5p,-Kh%C3%B4ng-Ph%E1%BA%A3i-S%E1%BB%B1-Kh%C3%A1c-Bi%E1%BB%87t)

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**](http://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#Ph%E1%BA%A7n-L%E1%BB%9Bn-Thao-T%C3%A1c-Di%E1%BB%85n-Ra-C%E1%BB%A5c-B%E1%BB%99)

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**](http://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-T%C3%ADnh-To%C3%A0n-V%E1%BA%B9n)

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**](http://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-Ch%E1%BB%89-Th%C3%AAm-M%E1%BB%9Bi-D%E1%BB%AF-Li%E1%BB%87u)

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**](http://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-Tr%E1%BA%A1ng-Th%C3%A1i)

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).

# 1.4 Bắt Đầu - Cài Đặt Git

## Cài Đặt Git

Hãy bắt đầu một chút vào việc sử dụng Git. Việc đầu tiên bạn cần phải làm là cài đặt nó. Có nhiều cách để thực hiện; hai cách chính đó là cài đặt từ mã nguồn hoặc cài đặt từ một gói có sẵn dựa trên hệ điều hành hiện tại của bạn.

### [Cài Đặt Từ Mã Nguồn](http://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%C3%A0i-%C4%90%E1%BA%B7t-Git#C%C3%A0i-%C4%90%E1%BA%B7t-T%E1%BB%AB-M%C3%A3-Ngu%E1%BB%93n)

Sẽ hữu ích hơn nếu bạn có thể cài đặt Git từ mã nguồn, vì bạn sẽ có được phiên bản mới nhất. Mỗi phiên bản của Git thường bao gồm nhiều cải tiến hữu ích về giao diện người dùng, vì thế cài đặt phiên bản mới nhất luôn là cách tốt nhất nếu như bạn quen thuộc với việc biên dịch phần mềm từ mã nguồn. Đôi khi nhiều phiên bản của Linux sử dụng các gói (package) rất cũ; vì thế trừ khi bạn đang sử dụng phiên bản mới nhất của Linux hoặc thường xuyên cập nhật, cài đặt từ mã nguồn có thể nói là sự lựa chọn tốt nhất.

Để cài đặt được Git, bạn cần có các thư viện mà Git sử dụng như sau: curl, zlib, openssl, expat, và libiconv. Ví dụ như bạn đang sử dụng một hệ điều hành có sử dụng yum (như Fedora) hoặc apt-get (như các hệ điều hành xây dựng trên nền Debian), bạn có thể sử dụng một trong các lệnh sau để cài đặt tất cả các thư viện cần thiết:

$ yum install curl-devel expat-devel gettext-devel \

openssl-devel zlib-devel

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

libz-dev libssl-dev

Khi đã cài đặt xong tất cả các thư viện cần thiết, bước tiếp theo là tải về phiên bản mới nhất của Git từ website của nó:

http://git-scm.com/download

Sau đó, dịch và cài đặt:

$ tar -zxf git-1.7.2.2.tar.gz

$ cd git-1.7.2.2

$ make prefix=/usr/local all

$ sudo make prefix=/usr/local install

Sau khi thực hiện xong các bước trên, bạn cũng có thể tải về các bản cập nhật của Git dùng chính nó như sau:

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

### [Cài Đặt Trên Linux](http://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%C3%A0i-%C4%90%E1%BA%B7t-Git#C%C3%A0i-%C4%90%E1%BA%B7t-Tr%C3%AAn-Linux)

Nếu như bạn muốn cài đặt Git trên Linux thông qua một chương trình cài đặt, bạn có thể làm việc này thông qua phần mềm quản lý các gói cơ bản đi kèm với hệ điều hành của bạn. Nếu bạn đang sử dụng Fedora, bạn có thể dùng yum:

$ yum install git-core

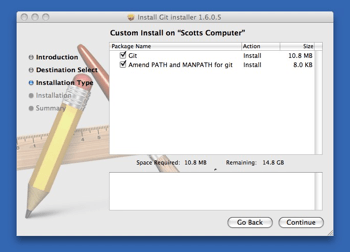
Còn nếu bạn đang sử dụng một hệ điều hành dựa trên nhân Debian như Ubuntu, hãy dùng apt-get:

$ apt-get install git

### [Cài Đặt Trên Mac](http://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%C3%A0i-%C4%90%E1%BA%B7t-Git#C%C3%A0i-%C4%90%E1%BA%B7t-Tr%C3%AAn-Mac)

Có hai cách đơn giản để cài đặt Git trên Mac. Cách đơn giản nhất là sử dụng chương trình cài đặt có hỗ trợ giao diện, bạn có thể tải về từ trang web của SourceForge (xem Hình 1-7):

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



Hình 1-7. Chương trình cài đặt Git cho Mac OS X.

Cách khác để cài đặt Git là thông qua MacPorts (http://www.macports.org). Nếu như bạn đã cài đặt MacPorts, Git có thể được cài đặt sử dụng lệnh sau:

$ sudo port install git-core +svn +doc +bash\_completion +gitweb

Bạn không phải cài đặt các thư viện đi kèm, nhưng có lẽ bạn muốn cài đặt thêm +svn trong trường hợp sử dụng chung Git với Subversion (xem Chương 8).

### [Cài Đặt Trên Windows](http://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%C3%A0i-%C4%90%E1%BA%B7t-Git#C%C3%A0i-%C4%90%E1%BA%B7t-Tr%C3%AAn-Windows)

Cài đặt Git trên Windows rất đơn giản. Dự án msysGit cung cấp một cách cài đặt Git dễ dàng hơn. Đơn giản chỉ tải về tập tin cài đặt định dạng exe từ Github, và chạy:

http://msysgit.github.com/

Sau khi nó được cài đặt, bạn có cả hai phiên bản: command-line (bao gồm SSH) và bản giao diện chuẩn.

Chú ý khi sử dụng trên Windows: bạn nên dùng Git bằng công cụ có sẵn: msysGit shell (kiểu Unix), nó cho phép bạn sử dụng các lệnh phức tạp trong sách này. Vì lý do nào đó, bạn muốn sử dụng cửa sổ dòng lệnh chuẩn của Windows: Windows shell, bạn bản sử dụng nháy kép thay vì nháy đơn (cho các tham số đầu vào có bao gồm dấu cách) và bạn phải dùng dấu mũ (^) cho tham số nếu chúng kéo dài đến cuối dòng, vì nó là ký tự tiếp diễn trong Windows.

**1.5 Bắt Đầu - Cấu Hình Git Lần Đầu**

**Cấu Hình Git Lần Đầu**

Bây giờ Git đã có trên hệ thống, bạn muốn tuỳ biến một số lựa chọn cho môi trường Git của bạn. Bạn chỉ phải thực hiện các bước này một lần duy nhất; chúng sẽ được ghi nhớ qua các lần cập nhật. Bạn cũng có thể thay đổi chúng bất kỳ lúc nào bằng cách chạy lại các lệnh.

Git cung cấp sẵn git config cho phép bạn xem hoặc chỉnh sửa các biến cấu hình để quản lý toàn bộ các khía cạnh của Git như giao diện hay hoạt động. Các biến này có thể được lưu ở ba vị trí khác nhau:

* /etc/gitconfig : Chứa giá trị cho tất cả người dùng và kho chứa trên hệ thống. Nếu bạn sử dụng --system khi chạy git config, thao tác đọc và ghi sẽ được thực hiện trên tập tin này.
* ~/.gitconfig : Riêng biệt cho tài khoản của bạn. Bạn có thể chỉ định Git đọc và ghi trên tập tin này bằng cách sử dụng --global.
* tập tin config trong thư mục git (.git/config) của bất kỳ kho chứa nào mà bạn đang sử dụng: Chỉ áp dụng riêng cho một kho chứa. Mỗi cấp sẽ ghi đè các giá trị của cấp trước nó, vì thế các giá trị trong .git/config sẽ "chiến thắng" các giá trị trong /etc/gitconfig.

Trên Windows, Git sử dụng tập tin .gitconfig trong thư mục $HOME (%USERPROFILE% trên môi trường Windows), cụ thể hơn đó là C:\Documents and Settings\$USER hoặc C:\Users\$USER, tuỳ thuộc vào phiên bản Windows đang sử dụng ($USER là %USERNAME% trên môi trường Windows). Nó cũng tìm kiếm tập tin /etc/gitconfig, mặc dù nó đã được cấu hình sẵn chỉ đến thư mục gốc của MSys, có thể là một thư mục bất kỳ, nơi bạn chọn khi cài đặt.

[**Danh Tính Của Bạn**](http://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%E1%BA%A5u-H%C3%ACnh-Git-L%E1%BA%A7n-%C4%90%E1%BA%A7u#Danh-T%C3%ADnh-C%E1%BB%A7a-B%E1%BA%A1n)

Việc đầu tiên bạn nên làm khi cấu hình Git là chỉ định tên tài khoản và địa chỉ e-mail. Điều này rất quan trọng vì mỗi Git sẽ sử dụng chúng cho mỗi lần commit, những thông tin này được gắn bất di bất dịch vào các commit:

$ git config --global user.name "John Doe"

$ git config --global user.email johndoe@example.com

Tôi xin nhắc lại là bạn chỉ phải làm việc này một lần duy nhất nếu như sử dụng --global, vì Git sẽ sử dụng các thông tin đó cho tất cả những gì bạn làm trên hệ thống. Nếu bạn muốn sử dụng tên và địa chỉ e-mail khác cho một dự án riêng biệt nào đó, bạn có thể chạy lại lệnh trên không sử dụng --global trên dự án đó.

[**Trình Soạn Thảo**](http://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%E1%BA%A5u-H%C3%ACnh-Git-L%E1%BA%A7n-%C4%90%E1%BA%A7u#Tr%C3%ACnh-So%E1%BA%A1n-Th%E1%BA%A3o)

Bây giờ danh tính của bạn đã được cấu hình xong, bạn có thể lựa chọn trình soạn thảo mặc định sử dụng để soạn thảo các dòng lệnh. Mặc định, Git sử dụng trình soạn thảo mặc địch của hệ điều hành, thường là Vi hoặc Vim. Nếu bạn muốn sử dụng một trình soạn thảo khác, như Emacs, bạn có thể sửa như sau:

$ git config --global core.editor emacs

[**Công Cụ So Sánh Thay Đổi**](http://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%E1%BA%A5u-H%C3%ACnh-Git-L%E1%BA%A7n-%C4%90%E1%BA%A7u#C%C3%B4ng-C%E1%BB%A5-So-S%C3%A1nh-Thay-%C4%90%E1%BB%95i)

Một lựa chọn hữu ích khác mà bạn có thể muốn thay đổi đó là chương trình so sánh sự thay đổi để giải quyết các trường hợp xung đột nội dung. Ví dụ bạn muốn sử dụng vimdiff:

$ git config --global merge.tool vimdiff

Git chấp nhận kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, và opendiff là các công cụ trộn/sát nhập (merge) hợp lệ. Bạn cũng có thể sử dụng một công cụ yêu thích khác; xem hướng dẫn ở Chương 7.

[**Kiểm Tra Cấu Hình**](http://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u-C%E1%BA%A5u-H%C3%ACnh-Git-L%E1%BA%A7n-%C4%90%E1%BA%A7u#Ki%E1%BB%83m-Tra-C%E1%BA%A5u-H%C3%ACnh)

Nếu như bạn muốn kiểm tra các cấu hình cài đặt, bạn có thể sử dụng lệnh git config --list để liệt kê tất cả các cài đặt của Git:

$ git config --list

user.name=Scott Chacon

user.email=schacon@gmail.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

Bạn có thể thấy các từ khoá xuất hiện nhiều hơn một lần, bởi vì Git đọc chúng từ các tập tin khác nhau (ví dụ, /etc/gitconfig và ~/.gitconfig). Trong trường hợp này Git sử dụng giá trị xuất hiện cuối cùng cho mỗi từ khoá duy nhất.

Bạn cũng có thể kiểm tra giá trị của một từ khoá riêng biệt nào đó bằng cách sử dụng git config {key}:

$ git config user.name

Scott Chacon

# 1.6 Bắt Đầu - Trợ Giúp

## Trợ Giúp

Nếu bạn cần sự giúp đỡ khi sử dụng Git, có ba cách để hiển thị tài liệu hướng dẫn (manpage) cho bất kỳ câu lệnh Git nào:

$ git help <verb>

$ git <verb> --help

$ man git-<verb>

Ví dụ, bạn có thể hiển thị hướng dẫn cho câu lệnh config bằng cách chạy:

$ git help config

Những lệnh này rất thuận tiện và hữu ích vì bạn có thể sử dụng chúng mọi nơi, ngay cả khi không có kết nối Internet. Nếu các tài liệu hướng dẫn và cuốn sách này chưa đủ, bạn vẫn cần thêm người trợ giúp, hãy thử sử dụng kênh #git hoặc #github trên Freenode IRC server (irc.freenode.net). Những kênh này thường xuyên thu hút hàng trăm người có kiến thức rất tốt về Git và họ luôn sẵn lòng giúp đỡ.

# 1.7 Bắt Đầu - Tóm Tắt

## Tóm Tắt

Bạn đã có kiến thức cơ bạn về Git là gì và chúng khác các CVCS (hệ thống quản lý phiên bản/mã nguồn tập trung) mà bạn đã, đang sử dụng như thế nào. Bạn cũng đã có một phiên bản hoạt động tốt của Git được cấu hình với danh tính cá nhân trên máy tính của bạn. Và đã đến lúc để học một số kiến thức cơ bản về Git.

# Chapter 2

# Cơ Bản Về Git

Đây có thể là chương duy nhất bạn cần đọc để có thể bắt đầu sử dụng Git. Chương này bao hàm từng câu lệnh cơ bản bạn 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. Chúng tôi cũng sẽ hướng dẫn 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.

# 2.1 Cơ Bản Về Git - Tạo Một Kho Chứa Git

## Tạo Một Kho Chứa Git

Bạn có thể tạo một 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](http://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#Kh%E1%BB%9Fi-T%E1%BA%A1o-M%E1%BB%99t-Kho-Ch%E1%BB%A9a-T%E1%BB%AB-Th%C6%B0-M%E1%BB%A5c-C%C5%A9)

Nếu như bạn muốn theo dõi một dự án cũ trong Git:

Bạn hãy di chuyển thư mục hiện hành vào trong thư mục gốc của dự án bằng lệnh:

$ cd [đường dẫn của thư mục]

Sau khi vào được thư mục hiện hành của thư viện, thực thi 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 (track) trong mỗi lần commit, sử dụng câu lệnh git add:

* Track (theo dõi) tất cả các file có phần mở rộng .c

$ git add \*.c

* Track (theo dõi) file có tên README

$ git add README

* Track (theo dõi) tất cả các file trong thư mục chứa gốc (thư mục có chứa .git)

$ git add \*.\*

Sau khi track (đưa các file theo dõi xong) chúng ta mới có thể tiến hành commit những file đã tracked (đã được theo dõi)

$ 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](http://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-Ch%C3%A9p-M%E1%BB%99t-Kho-Ch%E1%BB%A9a-%C4%90%C3%A3-T%E1%BB%93n-T%E1%BA%A1i)

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.git thông 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.

# 2.2 Cơ Bản Về Git - Ghi Lại Thay Đổi vào Kho Chứa

## 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.

### [Kiểm Tra Trạng Thái Của Tập Tin](http://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#Ki%E1%BB%83m-Tra-Tr%E1%BA%A1ng-Th%C3%A1i-C%E1%BB%A7a-T%E1%BA%ADp-Tin)

Công cụ chính để phát hiện trạng thái của tập tin là lệnh git status. Nếu bạn chạy lệnh này trực tiếp sau khi vừa tạo xong một bản sao, bạn sẽ thấy tương tự như sau:

$ git status

# On branch master

nothing to commit, working directory clean

Điều này có nghĩa là bạn có một thư mục làm việc "sạch" - hay nói cách khác, không có tập tin đang theo dõi nào bị thay đổi. Git cũng không phát hiện ra tập tin chưa được theo dõi nào, nếu không thì chúng đã được liệt kê ra đây. Cuối cùng, lệnh này cho bạn biết bạn đang thao tác trên "nhánh" (branch) nào. Hiện tại thì nó sẽ luôn là master, đó là nhánh mặc định; bạn chưa nên quan tâm đến vấn đề này bây giờ. Chương tiếp theo chúng ta sẽ bàn về các Nhánh chi tiết hơn.

Giả sử bạn thêm một tập tin mới vào dự án, một tập tin README đơn giản. Nếu như tập tin này chưa từng tồn tại trước đó, kho bạn chạy git status, bạn sẽ thấy thông báo tập tin chưa được theo dõi như sau:

$ vim README

$ git status

# On branch master

# Untracked files:

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

#

# README

nothing added to commit but untracked files present (use "git add" to track)

Bạn có thể thấy là tập tin README mới chưa được theo dõi, bởi vì nó nằm trong danh sách "Các tập tin chưa được theo dõi:" (Untracked files) trong thông báo trạng thái được hiển thị. Chưa được theo dõi về cơ bản có nghĩa là Git thấy một tập tin chưa tồn tại trong ảnh (lần commit) trước; Git sẽ không tự động thêm nó vào các commit tiếp theo trừ khi bạn chỉ định rõ ràng cho nó làm như vậy. Theo cách này, bạn sẽ không vô tình thêm vào các tập tin nhị phân hoặc các tập tin khác mà bạn không thực sự muốn. Trường hợp này bạn thực sự muốn thêm README, vậy hãy bắt đầu theo dõi nó.

### [Theo Dõi Các Tập Tin Mới](http://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#Theo-D%C3%B5i-C%C3%A1c-T%E1%BA%ADp-Tin-M%E1%BB%9Bi)

Để có thể theo dõi các tập tin mới tạo, bạn sử dụng lệnh git add. Và để bắt đầu theo dõi tập tin README bạn có thể chạy lệnh sau:

$ git add README

Nếu bạn chạy lệnh kiểm tra trạng thái lại một lần nữa, bạn sẽ thấy tập tin README bây giờ đã được theo dõi và tổ chức (staged):

$ git status

# On branch master

# Changes to be committed:

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

#

# new file: README

#

Bạn có thể thấy nó đã được staged vì nó đã nằm trong danh sách "Các thay đổi chuẩn bị commit". Nếu bạn commit tại thời điểm này, phiên bản của tập tin ở thời điểm bạn chạy git add sẽ được thêm vào lịch sử commit. Nhớ lại khi bạn chạy git init lúc trước, sau đó là lệnh git add (files) - đó chính là bắt đầu theo dõi các tập tin trong thư mục của bạn. 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](http://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#Qu%E1%BA%A3n-L%C3%BD-C%C3%A1c-T%E1%BA%ADp-Tin-%C4%90%C3%A3-Thay-%C4%90%E1%BB%95i)

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 câu 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). Chạy git add để "stage" tập tin benchmarks.rb và sau đó chạy lại lệnh git status:

$ git add benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

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

#

# new file: README

# modified: benchmarks.rb

#

Cả hai tập tin đã được tổ chức và sẽ có mặt trong lần commit tới. Bây giờ, giả sử bạn nhớ ra một chi tiết nhỏ nào đó cần thay đổi trong tập tin benchmarks.rb trước khi commit. Bạn lại mở nó ra và sửa, bây giờ thì sẵn sàng để commit rồi. Tuy nhiên, hãy chạy git status lại một lần nữa:

$ vim benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

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

#

# new file: README

# modified: benchmarks.rb

#

# Changes not staged for commit:

# (use "git add <file>..." to update what will be committed)

#

# modified: benchmarks.rb

#

Chuyện gì xảy ra thế này? Bây giờ benchmarks.rb lại nằm trong cả hai danh sách staged và unstaged. Làm sao có thể thế được? Hoá ra là Git tổ chức một tập tin chính lúc bạn chạy lệnh git add. Nếu bạn commit bây giờ, phiên bản của tập tin benchmarks.rb khi bạn chạy git add sẽ được commit chứ không phải như bạn nhìn thấy hiện tại trong thư mục làm việc khi chạy git commit. 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:

$ git add benchmarks.rb

$ git status

# On branch master

# Changes to be committed:

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

#

# new file: README

# modified: benchmarks.rb

#

### [Bỏ Qua Các Tập Tin](http://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#B%E1%BB%8F-Qua-C%C3%A1c-T%E1%BA%ADp-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 ta tự động ví dụ như các tập tin nhật ký (log files) 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. Đây là một 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 archiev có thể được tạo ra khi bạn dịch mã nguồn. Dòng thứ hai yêu cầu Git bỏ qua tất cả tập tin có đuôi là dẫu ngã (~), chúng được sử dụng để lưu các giá trị tạm thời bởi rất nhiều chương trình soạn thảo như Emacs. Bạn có thể thêm vào các thư mục như log, tmp, hay pid; hay các tài liệu được tạo ra 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 thám vào phía trước (!).

Các mẫu toàn cầu giống như các biểu thức chính quy (regular expression) rút gọn được sử dụng trong shell. Dấu sao (\*) khớp với 0 hoặc nhiều ký tự; [abc] khớp với bất kỳ ký tự nào trong dấu ngoặc (trong trường hợp này là a, b, hoặc c); dấu hỏi (?) khớp với một ký tự đơn; và 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).

Đây là một 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

Mẫu \*\*/ có mặt từ Git phiên bản 1.8.2 trở lên.

### [Xem Các Thay Đổi Staged và Unstaged](http://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#Xem-C%C3%A1c-Thay-%C4%90%E1%BB%95i-Staged-v%C3%A0-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. Chúng ta sẽ nói về git diff chi tiết hơn trong phần sau; nhưng chắc chắn 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 status chỉ trả lời những câu hỏi trên một cách chung chung, nhưng git diff chỉ cho bạn chính xác từng dòng đã được thêm hoặc xoá - hay còn được biết đến như là bản vá (patch).

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 bạn chạy lệnh status, bạn sẽ lại nhìn 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

#

Để xem chính xác bạn đã thay đổi nhưng chưa stage những gì, hãy 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 comiit 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:

$ git add benchmarks.rb

$ echo '# test line' >> benchmarks.rb

$ git status

# On branch master

#

# Changes to be committed:

#

# modified: benchmarks.rb

#

# Changes not staged for commit:

#

# modified: benchmarks.rb

#

Bây giờ 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](http://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#Commit-Thay-%C4%90%E1%BB%95i)

Bây giờ, sau khi đã tổ chức các tập tin theo ý muốn, bạn có thể commit chúng. Hãy nhỡ 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. Chúng sẽ vẫn ở trạng thái đã thay đổi trên ổ cứng của bạn. Trong trường hợp này, bạn thấy là từ lần cuối cùng chạy git status, tất cả mọi thứ đã được tổ chức thế nên bạn đã sẵn sàng để commit. Cách đơn giản nhất để commit là gõ vào git commit:

$ git commit

Sau khi chạy lệnh này, chương trình soạn thảo do bạn lựa chọn sẽ được mở lên. (Chương trình được chỉ định bằng biến $EDITOR - thường là vim hoặc emacs, tuy nhiên bạn có thể chọn bất kỳ chương trình nào khác bằng cách sử dụng lệnh git config --global core.editor như bạn đã thấy ở Chương 1).

Nó sẽ hiển thị đoạn thông báo sau (trong ví dụ này là màn hình của 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 và một dòng trống ở trên cùng. Bạn có thể xoá những 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 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

Bây giờ thì bạn đã thực hiện xong commit đầu tiên. Bạn có thể thấy là commit đó hiển thị một số thông tin về chính nó như: nhánh mà bạn commit tới (master), mã băm SHA-1 của commit đó, bao nhiêu tập tin đã thay đổi, và thống kê về số dòng đã thêm cũng như xoá trong commit.

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](http://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#B%E1%BB%8F-Qua-Khu-V%E1%BB%B1c-T%E1%BB%95-Ch%E1%BB%A9c)

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, Git đã cung cấp sẵn cho bạn một "lối tắt". 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:

$ git status

# On branch master

#

# Changes not staged for commit:

#

# modified: benchmarks.rb

#

$ git commit -a -m 'added new benchmarks'

[master 83e38c7] added new benchmarks

1 files changed, 5 insertions(+), 0 deletions(-)

Hãy chú ý tại sao bạn không phải chạy git add với tập tin benchmarks.rb trước khi commit trong trường hợp này.

### [Xoá Tập Tin](http://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#Xo%C3%A1-T%E1%BA%ADp-Tin)

Để xoá một tập tin khỏi Git, bạn phải xoá nó khỏi danh sách được theo dõi (chính xác hơn, xoá nó khỏi khu vực tổ chức) và sau đó commit. Lệnh git rm thực hiện điều đó và cũng xoá 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 xoá tập tin khỏi thư mục làm việc, nó sẽ được hiện 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 xoá 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 đó sẽ bị xoá và không còn được theo dõi nữa. Nếu như bạn đã sửa và thêm tập tin đó vào danh sách, bạn phải ép Git xoá đi bằng cách thêm lựa chọn -f. Đây là một chức năng an toàn nhằm ngăn chặn việc xoá 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 xoá 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 xoá toàn bộ tập tin kết thúc bằng ~.

### [Di Chuyển Tập Tin](http://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#Di-Chuy%E1%BB%83n-T%E1%BA%ADp-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.