Contents

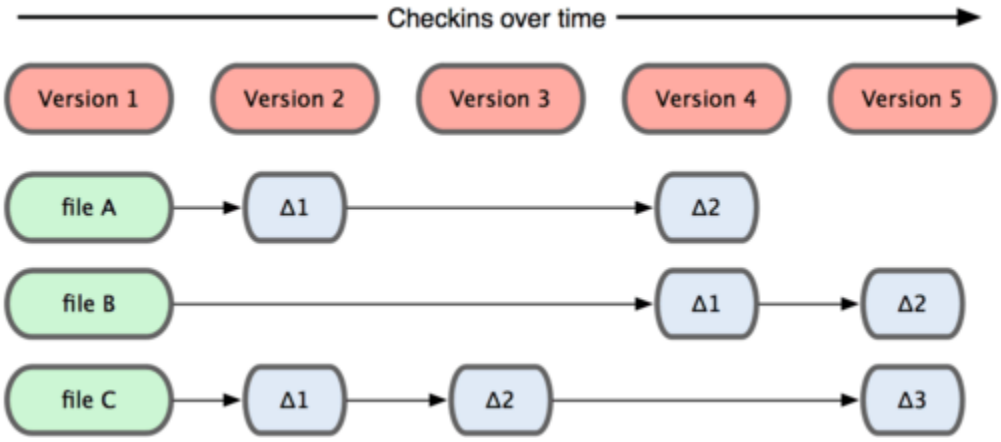
1.3 Bắt Đầu - Cơ Bản về Git

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.

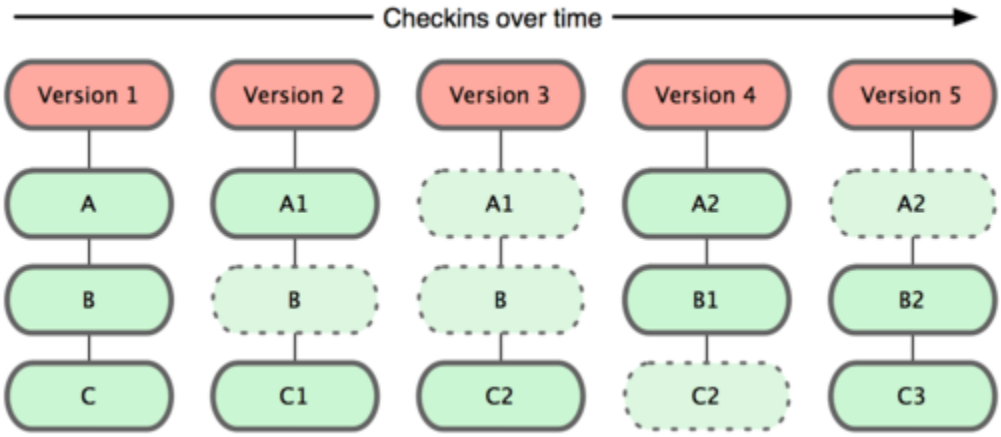
Ảnh Chụp, Không Phải Sự Khác Biệt

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

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

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

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

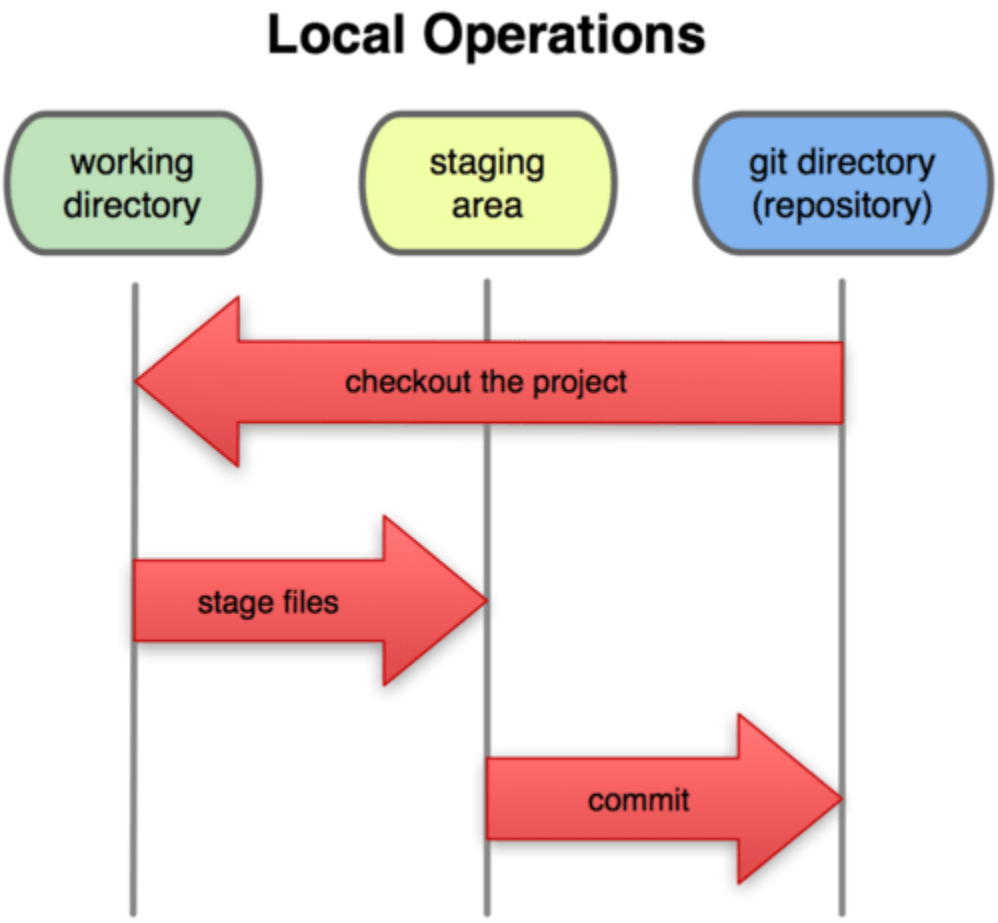
khôi phục lại được hoặc xoá dữ liệu đi dưới mọi hình thức. Giống như trong các VCS khác, bạn có thể mất hoặc làm rối tung dữ liệu mà bạn chưa commit; nhưng khi bạn đã commit thì rất khó để mất các dữ liệu đó, đặc biệt là nếu bạn thường xuyên đẩy (push) cơ sở dữ liệu sang một kho chứa khác.

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

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

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

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
```

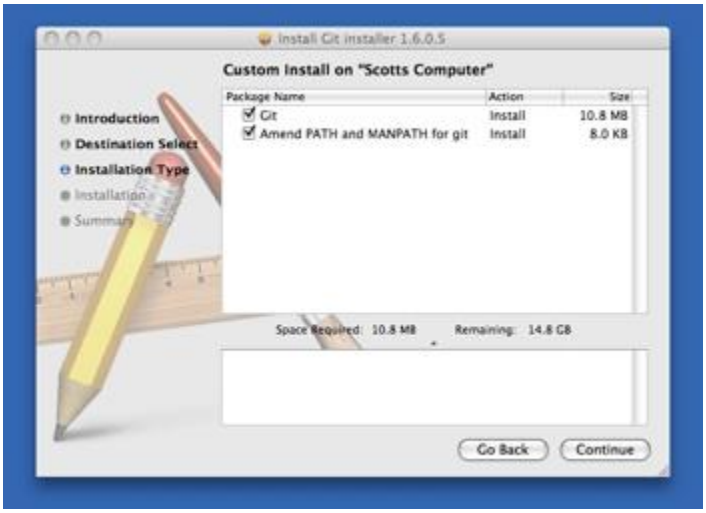
Còn nếu bạn đang sử dụng một hệ điều hành dựa trên nhân Debian như Ubuntu, hãy dùng apt-get:

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

Cài Đặt Trên Mac

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

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

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

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 định 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

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

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

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 kiểm 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

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.

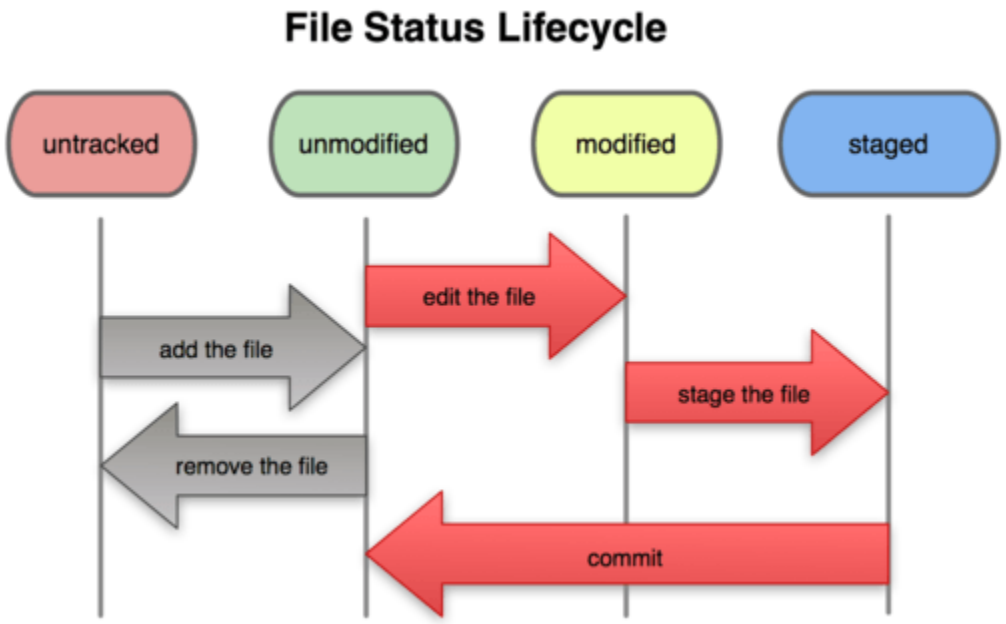
2.2 Cơ Bản Về Git - Ghi Lại Thay Đổi vào Kho Chứa

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

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

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

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

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 dấu 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

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

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

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

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

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.

2.3 Cơ Bản Về Git - Xem Lịch Sử Commit

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 lệnh `git log`.

Các ví dụ sau đây sử dụng một dự án rất đơn giản là `simplegit` tôi thường sử dụng làm ví dụ minh hoạ. Để tải dự án này, bạn hãy chạy lệnh:

```
git clone git://github.com/schacon/simplegit-progit.git
```

Khi bạn chạy `git log` trên dự án này, bạn sẽ thấy tương tự như sau:

```
$ git log
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit
```

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:

```
$ git log -p -2
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -5,5 +5,5 @@ require 'rake/gempackagetask'
 spec = Gem::Specification.new do |s|
     s.name      =   "simplegit"
-    s.version   =   "0.1.0"
+    s.version   =   "0.1.1"
     s.author    =   "Scott Chacon"
     s.email     =   "schacon@gee-mail.com

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700
```

```
    removed unnecessary test code

diff --git a/lib/simplegit.rb b/lib/simplegit.rb
index a0a60ae..47c6340 100644
--- a/lib/simplegit.rb
+++ b/lib/simplegit.rb
@@ -18,8 +18,3 @@ class SimpleGit
     end

 end
-
-if $0 == __FILE__
-  git = SimpleGit.new
-  puts git.show
-end
\ No newline at end of file
```

Lựa chọn này hiển thị thông tin tương tự nhưng thêm vào đó là nội dung diff trực tiếp của từng commit. Điều này rất có ích cho việc xem lại mã nguồn hoặc duyệt qua nhanh chóng những commit mà đồng nghiệp của bạn đã thực hiện.

Đôi khi xem lại cách thay đổi tổng quát (word level) lại dễ dàng hơn việc xem theo dòng. Lựa chọn `--word-diff` được cung cấp trong Git, bạn có thể thêm nó vào sau lệnh `git log -p` để xem diff một cách tổng quát thay vì xem từng dòng theo cách thông thường. Xem diff tổng quát dường như là vô dụng khi sử dụng với mã nguồn, nhưng lại rất hữu ích với các tập tin văn bản lớn như sách hay luận văn. Đây là một ví dụ:

```
$ git log -U1 --word-diff
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

diff --git a/Rakefile b/Rakefile
index a874b73..8f94139 100644
--- a/Rakefile
+++ b/Rakefile
@@ -7,3 +7,3 @@ spec = Gem::Specification.new do |s|
    s.name      =   "simplegit"
    s.version   =   [-"0.1.0"-]{+"0.1.1"+}
    s.author    =   "Scott Chacon"
```

Như bạn có thể thấy, không có dòng nào được thêm hay xoá trong phần thông báo như là với diff thông thường. Thay đổi được hiển thị ngay trên một dòng. Bạn có thể thấy phần thêm mới được bao quanh trong `{+ +}` còn phần xoá đi thì trong `[- -]`. Có thể bạn cũng muốn giảm ba dòng ngữ cảnh trong phần hiển thị diff xuống còn một dòng, vì ngữ cảnh hiện tại là các từ, không phải các dòng nữa. Bạn có thể làm được điều này với tham số `-U1` như ví dụ trên.

Bạn cũng có thể sử dụng một loại lựa chọn thống kê với `git log`. Ví dụ, nếu bạn muốn xem một số thống kê tóm tắt cho mỗi commit, bạn có thể sử dụng tham số `--stat`:

```
$ git log --stat
commit ca82a6dff817ec66f44342007202690a93763949
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Mon Mar 17 21:52:11 2008 -0700

    changed the version number

 Rakefile |    2 +-
 1 files changed, 1 insertions(+), 1 deletions(-)

commit 085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 16:40:33 2008 -0700

    removed unnecessary test code

 lib/simplegit.rb |    5 -----
 1 files changed, 0 insertions(+), 5 deletions(-)

commit a11bef06a3f659402fe7563abf99ad00de2209e6
Author: Scott Chacon <schacon@gee-mail.com>
Date:   Sat Mar 15 10:31:28 2008 -0700

    first commit

 README           |    6 ++++++
 Rakefile         |   23 +++++++++++++++++++++++
 lib/simplegit.rb |   25 +++++++++++++++++++++++++
 3 files changed, 54 insertions(+), 0 deletions(-)
```

Như bạn có thể thấy, lựa chọn `--stat` in ra phía dưới mỗi commit danh sách các tập tin đã chỉnh sửa, bao nhiêu tập tin được sửa, và bao nhiêu dòng trong các tập tin đó được thêm vào hay xoá đi. Nó cũng in ra một phần tóm tắt ở cuối cùng. Một lựa chọn rất hữu ích khác là `--pretty`. Lựa chọn này thay đổi phần hiển thị ra theo các cách khác nhau. Có một số lựa chọn được cung cấp sẵn cho bạn sử dụng. Lựa chọn `oneline` in mỗi commit trên một dòng, có ích khi bạn xem nhiều commit cùng lúc. Ngoài ra các lựa chọn `short`, `full`, và `fuller` hiện thị gần như tương tự nhau với ít hoặc nhiều thông tin hơn theo cùng thứ tự:

```
$ git log --pretty=oneline
ca82a6dff817ec66f44342007202690a93763949 changed the version number
085bb3bcb608e1e8451d4b2432f8ecbe6306e7e7 removed unnecessary test code
a11bef06a3f659402fe7563abf99ad00de2209e6 first commit
```

Lựa chọn thú vị nhất là `format`, cho phép bạn chỉ định định dạng riêng của phần hiện thị. Nó đặc biệt hữu ích khi bạn đang xuất ra cho các máy phân tích thông tin (machine parsing) - vì bạn là người chỉ rõ định dạng, nên bạn sẽ biết được nó không bị thay đổi cùng với các cập nhật sau này của Git.

```
$ git log --pretty=format:"%h - %an, %ar : %s"
ca82a6d - Scott Chacon, 11 months ago : changed the version number
085bb3b - Scott Chacon, 11 months ago : removed unnecessary test code
a11bef0 - Scott Chacon, 11 months ago : first commit
```

Bảng 2-1 liệt kê một vài lựa chọn mà `format` sử dụng.

Lựa chọn	Mô tả thông tin đầu ra
`%H`	Mã băm của commit
`%h`	Mã băm của commit ngắn gọn hơn
`%T`	Băm hiển thị dạng cây
`%t`	Băm hiển thị dạng cây ngắn gọn hơn
`%P`	Các mã băm gốc
`%p`	Mã băm gốc ngắn gọn
`%an`	Tên tác giả
`%ae`	E-mail tác giả
`%ad`	Ngày "tác giả" (định dạng tương tự như lựa chọn --date=)
`%ar`	Ngày tác giả, tương đối
`%cn`	Tên người commit
`%ce`	Email người commit
`%cd`	Ngày commit
`%cr`	Ngày commit, tương đối
`%s`	Chủ để

Có thể bạn băn khoăn về sự khác nhau giữa *tác giả* (author) và *người commit* (committer). *Tác giả* là người đầu tiên viết bản vá (patch), trong khi đó *người commit* là người cuối cùng áp dụng miếng vá đó. Như vậy, nếu bạn gửi một bản vá cho một dự án và một trong các thành viên chính của dự án "áp dụng" (chấp nhận) bản vá đó, cả hai sẽ cùng được ghi nhận công trạng (credit) - bạn với vai trò là tác giả và thành viên của dự án trong vai trò người commit. Chúng ta sẽ bàn kỹ hơn một chút về sự khác nhau này trong *Chương 5*.

Lựa chọn `oneline` và `format` đặc biệt hữu ích khi sử dụng với một tham số khác của `log` là `--graph`. Khi sử dụng, tham số này sẽ thêm một biểu đồ sử dụng dựa trên các ký tự ASCII hiển thị nhánh và lịch sử tích hợp các tập tin của bạn, chúng ta có thể thấy trong dự án Grit như sau:

```
$ git log --pretty=format:"%h %s" --graph
* 2d3acf9 ignore errors from SIGCHLD on trap
*  5e3ee11 Merge branch 'master' of git://github.com/dustin/grit
```

```
|\
| * 420eac9 Added a method for getting the current branch.
* | 30e367c timeout code and tests
* | 5a09431 add timeout protection to grit
* | e1193f8 support for heads with slashes in them
|/
* d6016bc require time for xmlschema
*  11d191e Merge branch 'defunkt' into local
```

Vừa rồi mới chỉ là một số lựa chọn định dạng cơ bản cho `git log` - còn rất nhiều các định dạng khác. Bảng 2-2 liệt kê các lựa chọn chúng ta đã đề cập qua và một số định dạng cơ bản khác có thể hữu ích, cùng với mô tả đầu ra của lệnh `log`.

Tuỳ chọn	Mô tả
`-p`	Hiển thị bản vá với mỗi commit.
`--word-diff`	Hiển thị bản vá ở định dạng tổng quan (word).
`--stat`	Hiển thị thống kê của các tập tin được chỉnh sửa trong mỗi commit.
`--shortstat`	Chỉ hiển thị thay đổi/thêm mới/xoá bằng lệnh --stat.
`--name-only`	Hiển thị danh sách các tập tin đã thay đổi sau thông tin của commit.
`--name-status`	Hiển thị các tập tin bị ảnh hưởng với các thông tin như thêm mới/sửa/xoá.
`--abbrev-commit`	Chỉ hiện thị một số ký tự đầu của mã băm SHA-1 thay vì tất cả 40.
`--relative-date`	Hiển thị ngày ở định dạng tương đối (ví dụ, "2 weeks ago") thay vì định dạng đầy đủ.
`--graph`	Hiển thị biểu đồ ASCII của nhánh và lịch sử tích hợp cùng với thông tin đầu ra khác.
`--pretty`	Hiện thị các commit sử dụng một định dạng khác. Các lựa chọn bao gồm oneline, short, full, fuller và format (cho phép bạn sử dụng định dạng riêng).
`--oneline`	Một lựa chọn ngắn, thuận tiện cho `--pretty=oneline --abbrev-commit`.

Giới Hạn Thông Tin Đầu Ra

Ngoài các lựa chọn để định dạng đầu ra, `git log` còn nhận vào một số các lựa chọn khác cho mục đích giới hạn khác - là các lựa chọn cho phép bạn hiển thị một phần các commit. Bạn đã thấy một trong các tham số đó - đó là `-2`, cái mà dùng để hiện thị hai commit mới nhất. Thực tế bạn có thể dùng `-<n>`, trong đó `n` là số nguyên dương bất kỳ để hiển thị `n` commit mới nhất. Trong thực tế, bạn thường không sử dụng chúng, vì mặc định Git đã hiển thị đầu ra theo trang do vậy bạn chỉ xem được một trang lịch sử tại một thời điểm.

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

```
$ git log --since=2.weeks
```

Lệnh này hoạt động được với rất nhiều định dạng - bạn có thể chỉ định một ngày cụ thể ("2008-01-15") hoặc tương đối như "2 years 1 day 3 minutes ago".

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

Tham số hữu ích cuối cùng sử dụng cho `git log` với vai trò một bộ lọc là đường dẫn. Nếu bạn chỉ định một thư mục hoặc tên một tập tin, bạn có thể giới hạn các commit chỉ được thực hiện trên tập tin đó. Tham số này luôn được sử dụng cuối cùng trong câu lệnh và đứng sau hai gạch ngang (`--`) như thường lệ để phân chia các đường dẫn khác nhau.

Bảng 2-3 liệt kê các lựa chọn trên và một số lựa chọn phổ biến khác cho bạn thao khảo.

Lựa chọn	Mô tả
`-(n)`	Chỉ hiển thị n commit mới nhất
`--since, --after`	Giới hạn các commit được thực hiện sau ngày nhất định.
`--until, --before`	Giới hạn các commit được thực hiện trước ngày nhất định.
`--author`	Chỉ hiện thị các commit mà tên tác giả thoả mãn điều kiện nhất định.
`--committer`	Chỉ hiện thị các commit mà tên người commit thoả mãn điều kiện nhất định.

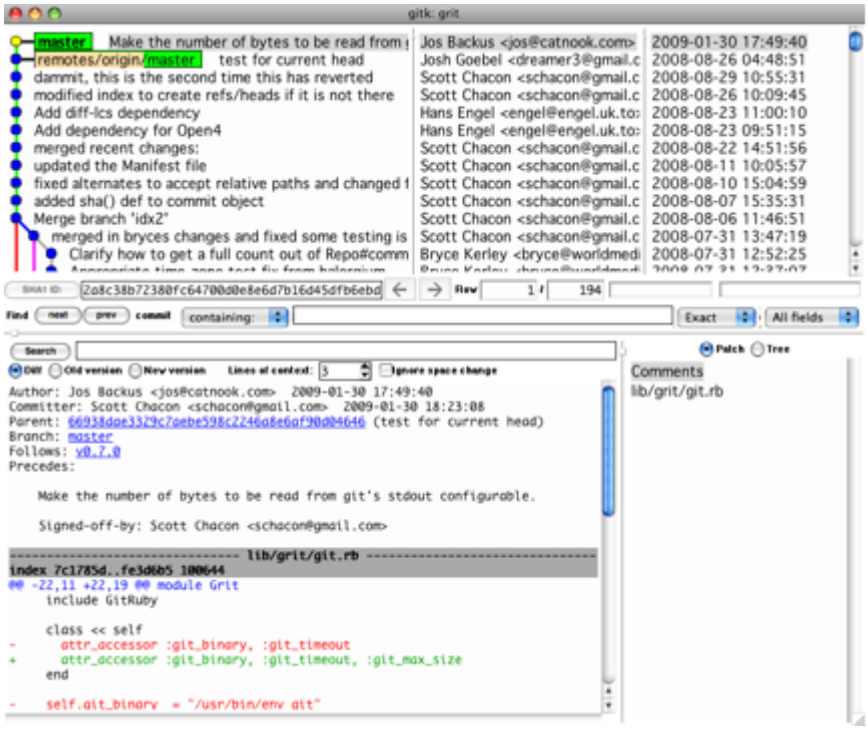
Ví dụ, bạn muốn xem các commit đã thay đổi các tập tin thử nghiệm trong lịch sử mã nguồn của Git, được commit bởi Junio Hâmno trng tháng 10 năm 2008 mà chưa được tích hợp/gộp, bạn có thể chạy lệnh sau:

```
$ git log --pretty="%h - %s" --author=gitster --since="2008-10-01" \
   --before="2008-11-01" --no-merges -- t/
5610e3b - Fix testcase failure when extended attribute
acd3b9e - Enhance hold_lock_file_for_{update,append}()
f563754 - demonstrate breakage of detached checkout wi
d1a43f2 - reset --hard/read-tree --reset -u: remove un
51a94af - Fix "checkout --track -b newbranch" on detac
b0ad11e - pull: allow "git pull origin $something:$cur
```

Có gần 20,000 commit trong lịch sử mã nguồn của Git, lệnh này chỉ hiện thị 6 commit thoả mãn tiêu chí đặt ra.

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.

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

2.4 Cơ Bản Về Git - Phục Hồi

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

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

```
$ git commit --amend
```

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

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

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

```
$ git commit -m 'initial commit'
$ git add forgotten_file
$ git commit --amend
```

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

Loại Bỏ Tập Tin Đã Tổ Chức

Hai phần tiếp theo sẽ minh 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.rb` được thay đổi và một lần nữa lại trở thành chưa tổ chức.

Phục Hồi Tập Tin Đã Thay Đổi

Sẽ như thế nào khi bạn nhận ra rằng bạn không muốn giữ những thay đổi trong tập tin `benchmarks.rb`? Làm thế nào để dễ dàng phục hồi lại những thay đổi đó - phục hồi nó lại trạng thái giống như sau khi thực hiện commit cuối cùng (hoặc như sau khi sao chép (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 được 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` (xem thêm về phục hồi dữ liệu ở *Chương 9*). Tuy nhiên, bất cứ thứ gì bị mất mà chưa được commit thì không có cơ hội phục hồi lại.

2.5 Cơ Bản Về Git - Làm Việc Từ Xa

Làm Việc Từ Xa

Để có thể cùng cộng tác với các thành viên khác trên bất kỳ dự án sử dụng Git nào, bạn cần phải biết quản lý các kho chứa của bạn. Các kho chứa từ xa là các phiên bản của dự án của bạn, được lưu trữ trên Internet hoặc một mạng lưới nào đó. Bạn có thể có nhiều kho chứa khác nhau, thường thì bạn có thể chỉ-đọc hoặc đọc/ghi. Cộng tác với các thành viên khác liên quan đến quản lý những kho chứa này và việc kéo, đẩy dữ liệu từ chúng khi bạn cần chia sẻ công việc. Quản lý các kho chứa từ xa đòi hỏi phải biết cách thêm các kho chứa, xoá kho chứa không hợp lệ, quản lý nhiều nhánh khác nhau và xác định có theo dõi chúng hay không, và còn nhiều hơn thế nữa. Trong phần này chúng ta sẽ đề cập đến các kỹ năng quản lý từ xa này.

Hiển Thị Máy Chủ

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

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

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

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

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

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

Điều này có nghĩa là bạn có thể "kéo" những đóng góp từ bất kỳ người dùng nào ở trên một cách dễ dàng. Nhưng chú ý là chỉ máy chủ nguyên bản từ xa (origin remote) là có địa chỉ SSH, do vậy nó là cái duy nhất mà tôi có thể đẩy lên (chúng ta sẽ tìm hiểu tại sao trong *Chương 4*).

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]`:

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

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

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

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

Truy Cập Và Kéo Về Từ Máy Chủ Trung Tâm

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 tên *origin*. Vì thế, `git fetch origin` sẽ truy xuất (fetch) bất kỳ thay đổi mới nào được đẩy lên trên máy chủ từ sau khi bạn sao chép (hoặc lần truy xuất cuối cùng). Hãy ghi nhớ một điều quan trọng là lệnh `fetch` kéo tất cả dữ liệu về kho chứa trên máy của bạn - nó không tự động tích hợp với bất kỳ thay đổi nào mà bạn đang thực hiện. Bạn phải tích hợp nó một cách thủ 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 3* để 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

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

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

```
$ git remote show origin
* remote origin
  URL: git://github.com/schacon/ticgit.git
```

```
  Remote branch merged with 'git pull' while on branch master
    master
  Tracked remote branches
    master
    ticgit
```

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

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

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

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

Xóa Và Đổi Tên Từ Xa

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

2.6 Cơ Bản Về Git - Đánh Dấu

Đánh Dấu

Cũng giống như đa số các hệ quản trị phiên bản khác, Git có khả năng đánh dấu (tag) các mốc quan trọng trong lịch sử của dự án. Nhìn chung, mọi người sử dụng chức năng này để đánh dấu các thời điểm phát hành (ví dụ như `v1.0`). Trong phần này bạn sẽ được học làm sao để liệt kê các tag hiện có, làm sao để tạo mới tag, và các loại tag khác nhau hiện có.

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 chữ 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

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

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

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

Tham số `-m` được sử dụng để truyền vào 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

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

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

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

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

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 lệnh `git push`. Nó sẽ truyền tất cả các tag chưa được đồng bộ lên máy chủ.

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

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

2.7 Cơ Bản Về Git - Mẹo Nhỏ

Mẹo Nhỏ

Trước khi kết thúc chương cơ bản về Git này, có một vài mẹo nhỏ có thể giúp ích cho việc sử dụng Git của bạn trở nên đơn giản và dễ dàng hơn. Có nhiều người vẫn sử dụng Git mà không biết đến những điều này, chúng ta sẽ không đề cập đến chúng hoặc giả định bạn sẽ sử dụng nó khi kết thúc cuốn sách này; tuy nhiên bạn nên biết cách sử dụng chúng.

Gợi Ý

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

```
source ~/git-completion.bash
```

Nếu như bạn muốn cài đặt công cụ gợi ý này cho tất cả người dùng trên máy tính của bạn, hãy sao chép đoạn mã này vào thư mục `/opt/local/etc/bash_completion.d` trên máy tính Mac hoặc thư mục `/etc/bash_completion.d/` trên các máy tính chạy Linux. Đây là thư mục chứa các đoạn mã mà Bash sẽ tự động chạy để có thể cung cấp chức năng gợi ý cho bạn.

Nếu bạn đang sử dụng Git Bash trên Windows - mặc định khi cài đặt Git trên Windows sử dụng msysGit, chức năng gợi ý đã được cấu hình sẵn.

Ấn phím Tab khi bạn gõ một câu lệnh Git, nó sẽ trả về một tập hợp các gợi ý cho bạn chọn:

```
$ git co<tab><tab>
commit config
```

Trong trường hợp này, gõ `git co` và sau đó gõ Tab hai lần sẽ cho bạn gợi ý commit và config. Gõ thêm `m<tab>` để có được lệnh `git commit` tự động.

Nó cũng hoạt động được với các lựa chọn/tham số, chắc chắn rất hữu ích. Ví dụ như nếu bạn đang chạy lệnh `git log` và không nhớ một trong các lựa chọn, bạn có thể bắt đầu gõ và ấn Tab để xem lệnh nào thỏa mãn:

```
$ git log --s<tab>
--shortstat  --since=  --src-prefix=  --stat   --summary
```

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

Bí Danh Trong Git

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

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

Có nghĩa là, ví dụ, thay vì phải gõ `git commit`, bạn chỉ cần gõ `git ci`. Khi bạn bắt đầu sử dụng Git, chắc chắn bạn sẽ sử dụng cả các câu lệnh khác một cách thường xuyên; trong trường hợp này, đừng ngần ngại tạo thêm các bí danh mới.

Kỹ thuật này cũng có thể rất hữu ích trong việc tạo mới các câu lệnh mà bạn cho rằng sự tồn tại của chúng là cần thiết. Ví dụ như, để làm chính xác các vấn đề liên quan đến tính khả dụng mà bạn gặp phải khi bỏ tổ chức (unstaging) một tập tin, bạn có thể tự tạo bí danh riêng cho việc này:

```
$ git config --global alias.unstage 'reset HEAD --'
```

Lệnh này tương đương với hai câu lệnh sau:

```
$ git unstage fileA
$ git reset HEAD fileA
```

Theo cách này thì nhìn có vẻ rõ ràng hơn. Một bí danh phổ biến khác là lệnh `last`, như sau:

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

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

```
$ git last
commit 66938dae3329c7aebe598c2246a8e6af90d04646
Author: Josh Goebel <dreamer3@example.com>
Date:   Tue Aug 26 19:48:51 2008 +0800

    test for current head

    Signed-off-by: Scott Chacon <schacon@example.com>
```

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

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

2.8 Cơ Bản Về Git - Tổng Kết

Tổng Kết

Đến bây giờ thì bạn đã có thể thực hiện các thao tác cơ bản của Git một cách cục bộ - tạo mới, sao chép kho chứa, tạo thay đổi, tổ chức và commit các thay đổi đó, và xem lịch sử của các thay đổi đã được thực hiện trên kho chứa. Trong phần tiếp theo, chúng ta sẽ đề cập tới chức năng tuyệt vời của Git: mô hình phân nhánh.

Chapter 3

Phân Nhánh Trong Git

Hầu hết mỗi hệ quản trị phiên bản (VCS) đều hỗ trợ một dạng của phân nhánh. Phân nhánh có nghĩa là bạn phân tách ra từ luồng phát triển chính và tiếp tục làm việc mà không sợ làm ảnh hưởng đến luồng chính. Trong nhiều VCS, đây dường như là một quá trình đòi hỏi nhiều công sức và sự cố gắng, thường thì bạn tạo một bản sao mới từ thư mục chứa mã nguồn, nó có thể mất khá nhiều thời gian trên các dự án lớn.

Nhiều người nhắc đến mô hình phân nhánh của Git như là "chức năng hủy diệt", và chính nó làm cho Git trở nên khác biệt trong cộng đồng VCS. Tại sao nó lại đặc biệt đến vậy? Cách Git phân nhánh "nhẹ" một cách đáng kinh ngạc, các hoạt động tạo nhánh xảy ra gần như ngay lập tức và việc di chuyển đi lại giữa các nhánh cũng thường rất nhanh. Không giống các VCSs khác, Git khuyến khích sử dụng rẽ nhánh và tích hợp thường xuyên cho workflow, thậm chí nhiều lần trong một ngày. Hiểu và thành thạo tính năng này cung cấp cho bạn một công cụ mạnh mẽ, độc đáo và có thể thay đổi được cách bạn thường phát triển phần mềm.

3.1 Phân Nhánh Trong Git - Nhánh Là Gì?

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).

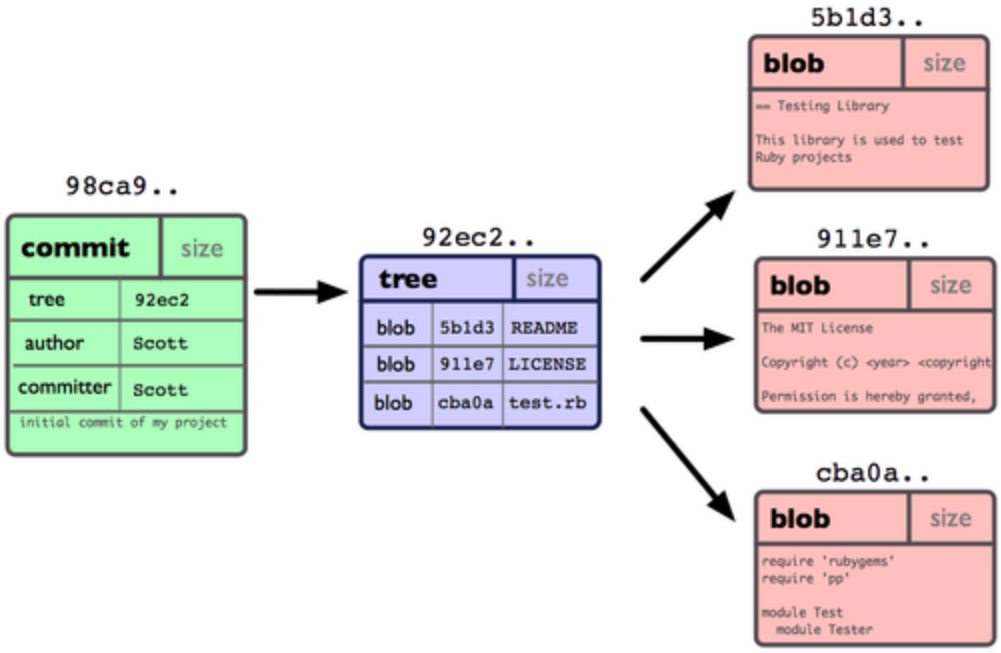
Khi bạn commit, Git lưu trữ đối tượng commit mà có chứa một con trỏ tới ảnh của nội dung bạn đã tổ chức (stage), tác giả và thông điệp, hay 0 hoặc nhiều con trỏ khác trỏ tới một hoặc nhiều commit cha trực tiếp của commit đó: commit đầu tiên không có cha, commit bình thường có một cha, và nhiều cha cho 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'
```

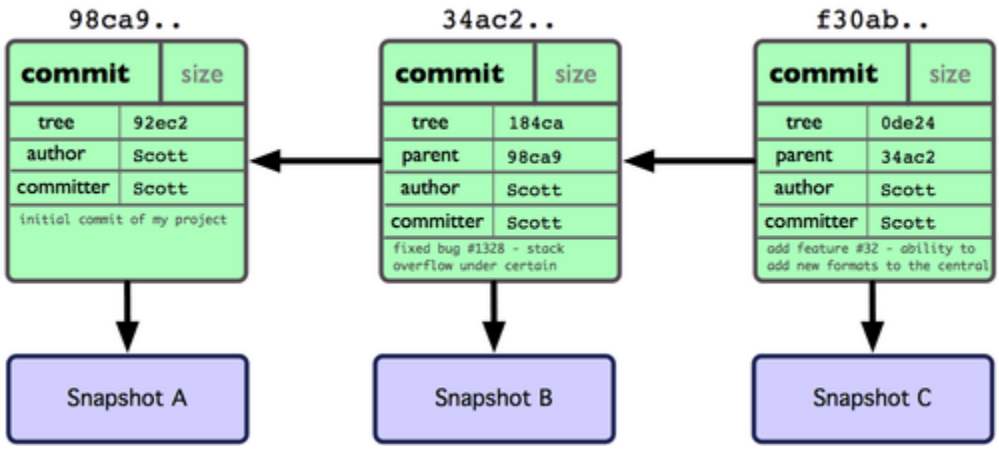
Lệnh `git commit` khi chạy sẽ băm tất cả các thư mục trong dự án và lưu chúng lại dưới dạng đối tượng `tree`. Sau đó Git tạo một đối tượng `commit` có chứa các thông tin mô tả (metadata) và một con trỏ trỏ tới đối tương `tree` gốc của dự án vì thế nó có thể tạo lại ảnh đó khi cần thiết.

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.



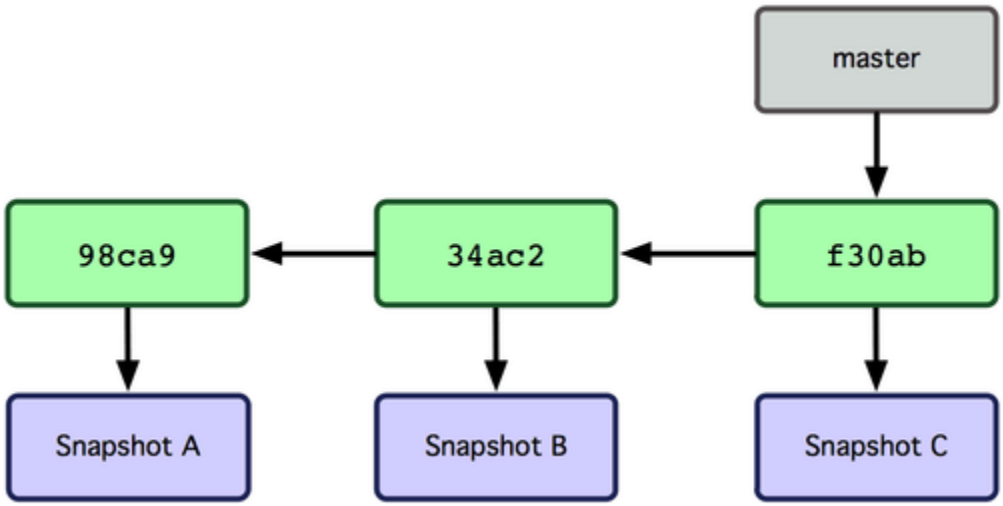
Hình 3-1. Dữ liệu trong kho chứa với một commit.

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)

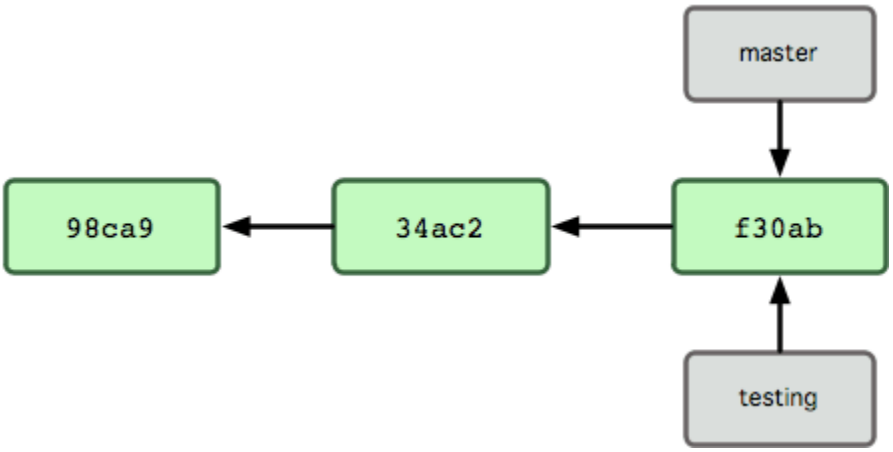


Hình 3-3. Nhánh trỏ tới dữ liệu commit.

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`:

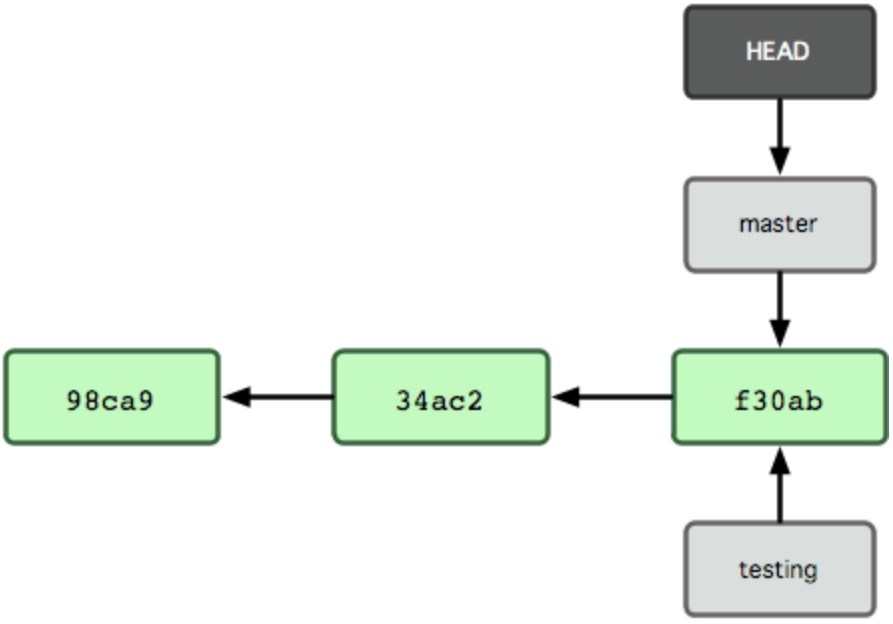
```
$ git branch testing
```

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).

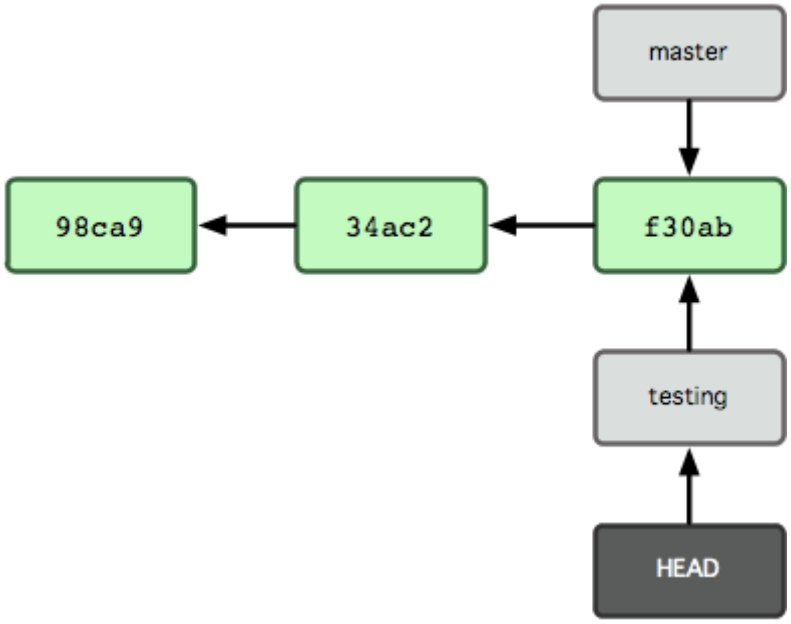


Hình 3-5. Tập tin HEAD trỏ tới nhánh mà bạn đang làm việc.

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).

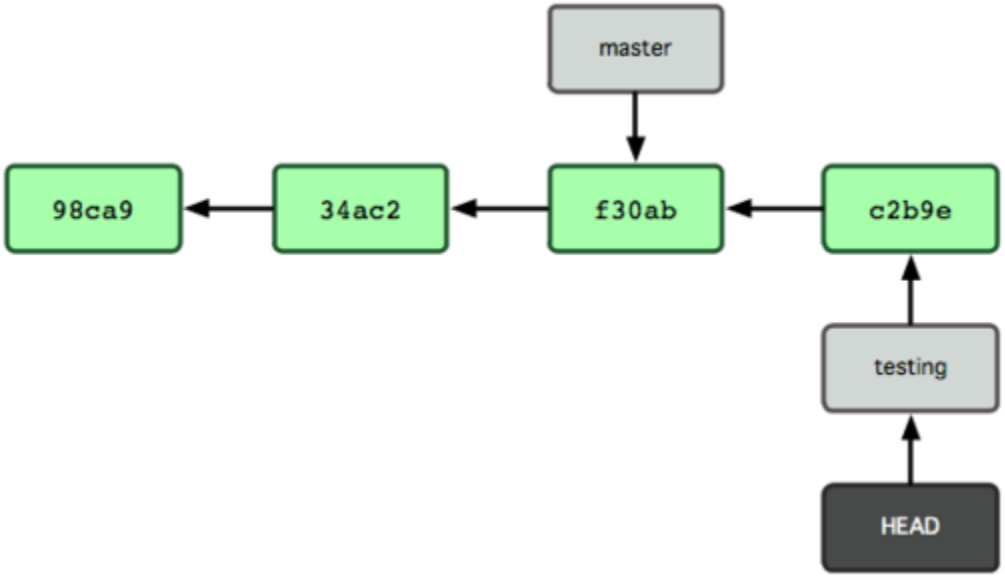


Hình 3-6. HEAD trỏ tới nhánh khác khi bạn chuyển nhánh.

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.

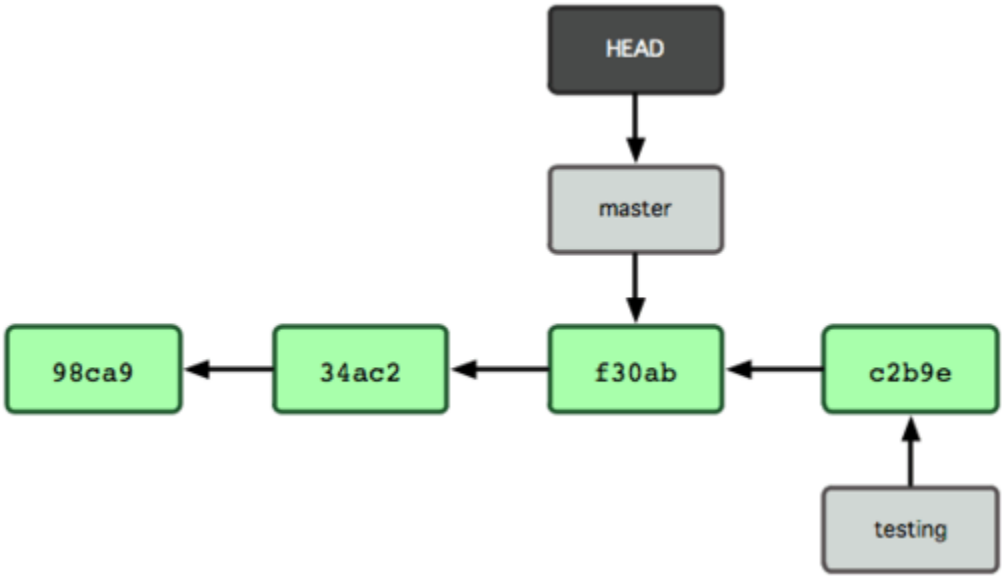


Hình 3-7. Nhánh mà HEAD trỏ tới di chuyển tiến lên phía trước theo từng commit.

Điều này thật thú vị, bởi vì nhánh testing của bạn bây giờ đã tiến hẳn lên phía trước, nhưng nhánh `master` thì 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.



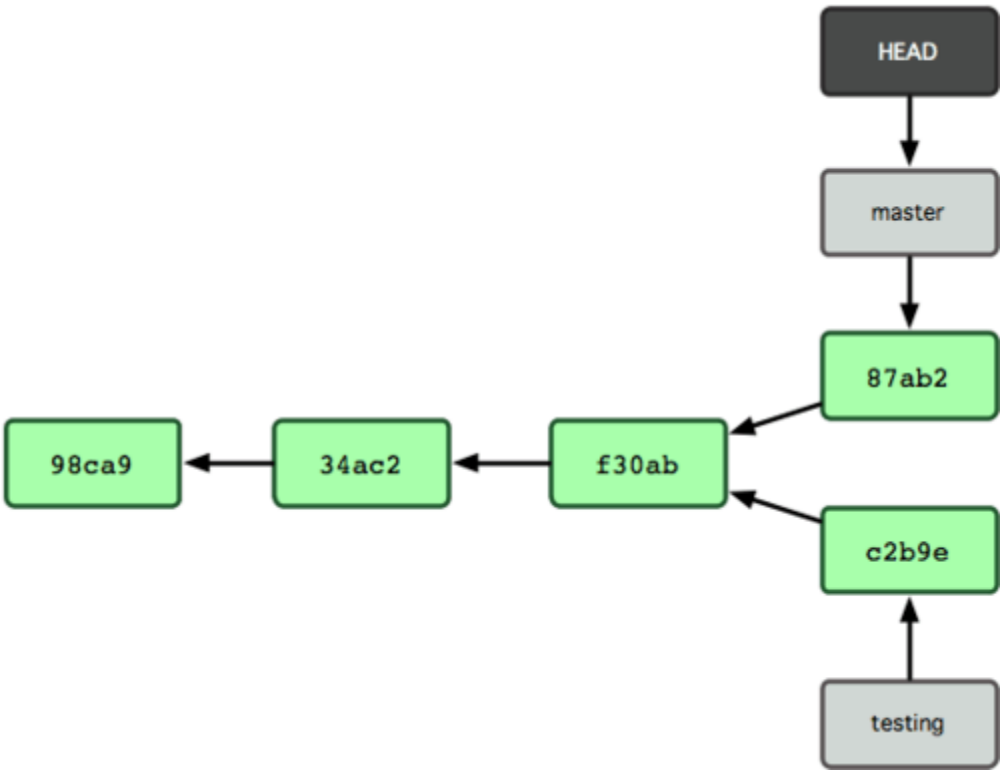
Hình 3-8. HEAD chuyển sang nhánh khác khi checkout.

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` và `checkout`.



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.

Hãy cùng xem tại sao bạn nên làm như vậy.

3.2 Phân Nhánh Trong Git - Cơ Bản Về Phân Nhánh và Tích Hợp

Cơ Bản Về Phân Nhánh và Tích Hợp

Hãy cùng xem qua một ví dụ đơn giản về phân nhánh và tích hợp với một quy trình làm việc mà có thể bạn sẽ sử dụng nó vào thực tế. Bạn sẽ thực hiện theo các bước sau:

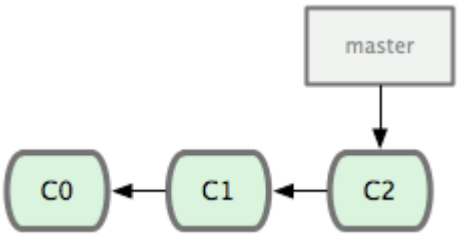
1. Làm việc trên một web site
2. Tạo nhánh cho một câu chuyện mới mà bạn đang làm.
3. Làm việc trên nhánh đó.

Đến lúc này, bạn nhận được thông báo rằng có một vấn đề nghiêm trọng cần được khắc phục ngay. Bạn sẽ làm theo các bước sau:

1. Chuyển lại về nhánh sản xuất (production)
2. Tạo mới một nhánh khác để khắc phục lỗi
3. Sau khi đã kiểm tra ổn định, tích hợp nhánh đó lại và đưa vào hoạt động.
4. Chuyển ngược lại với câu chuyện của bạn và tiếp tục làm việc.

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.

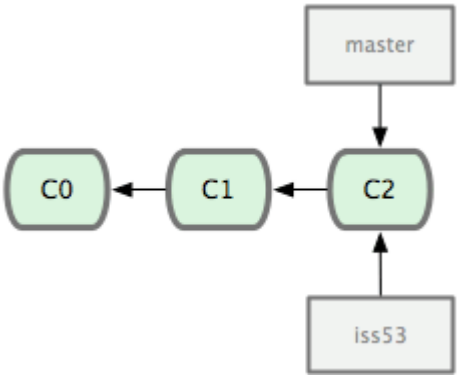
Bạn quyết định sẽ giải quyết vấn đề số #53 sử dụng bất kỳ hệ thống giám sát vấn đề (issue-tracking) nào mà công ty bạn đang dùng. Để cho rõ ràng, Git không cung cấp kèm bất kỳ hệ thống giám sát vấn đề nào; nhưng bởi vì vấn đề số #53 là cái mà bạn sẽ tập trung vào nên bạn sẽ tạo một nhánh mới để làm việc 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
```

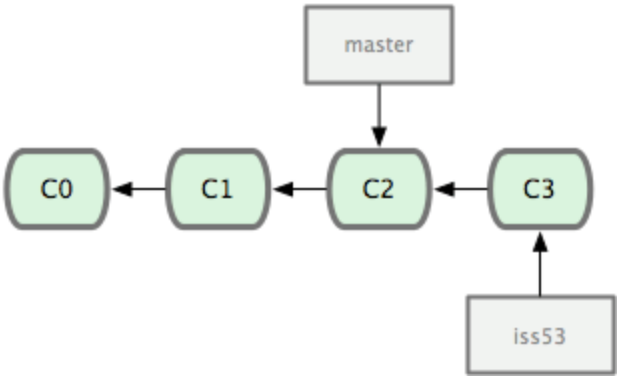
Hình 3-11 minh họa kết quả.



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:

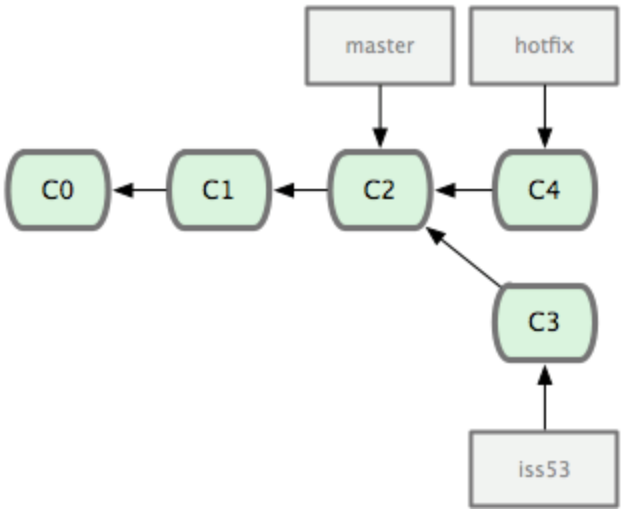
```
$ git checkout master
```

```
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.

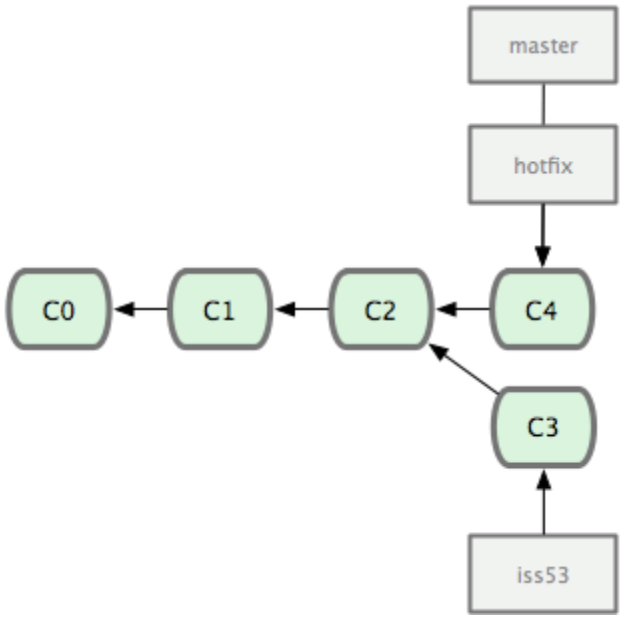
Bạn có thể chạy để kiểm tra, để chắc chắn rằng bản vá lỗi hoạt động đúng theo ý bạn muốn, và sau đó tích hợp nó lại nhánh chính để triển khai. Bạn có thể làm sử dụng lệnh `git merge` để làm việc này:

```
$ 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).



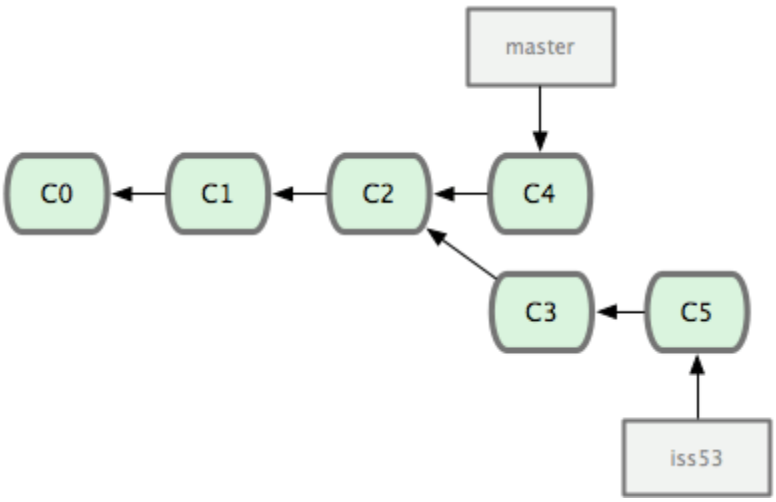
Hình 3-14. Nhánh master và nhánh hotfix cùng trỏ tới một điểm sau khi tích hợp.

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 - nhánh `master` 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.

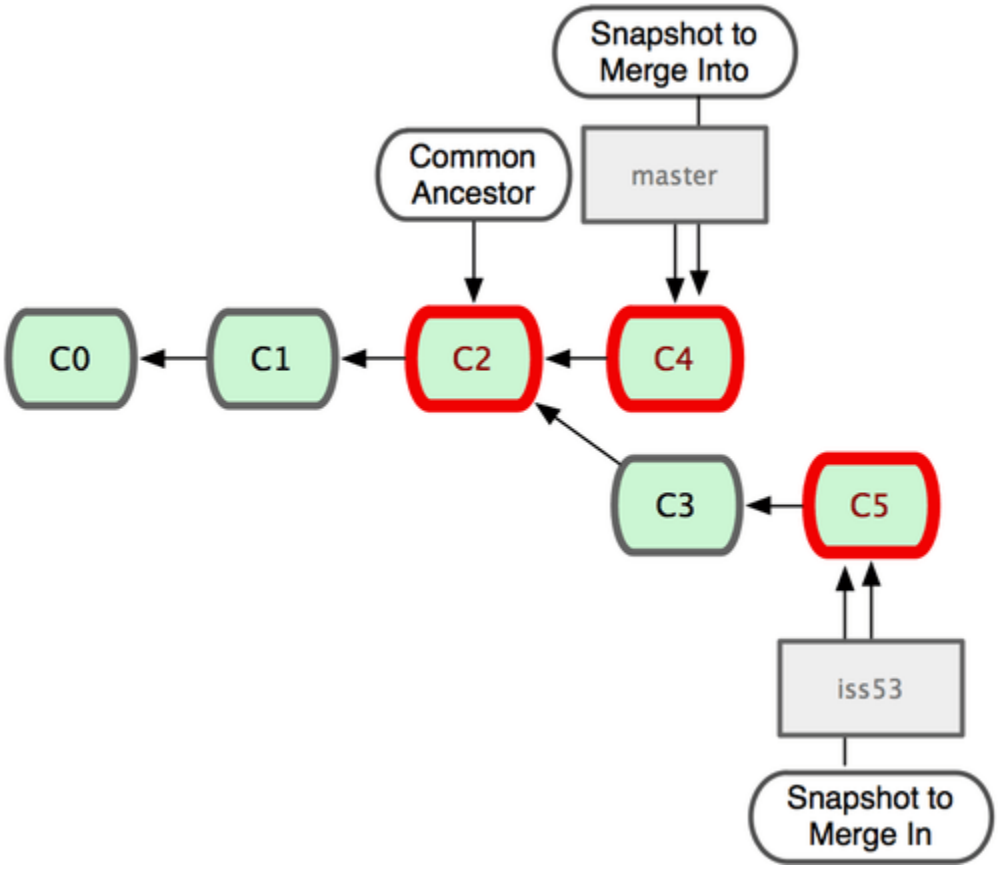
Điều đáng chú ý ở đây là những công việc bạn đã thực hiện ở nhánh `hotfix` không bao gồm trong nhánh `iss53`. Nếu bạn muốn đưa chúng vào, bạn có thể tích hợp nhánh `master` vào nhánh `iss53` bằng cách chạy lệnh `git merge master`, hoặc bạn có thể chờ đợi đến khi bạn quyết định tích hợp nhánh `iss53`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

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 nhánh `master`. Để làm được điều này, bạn sẽ tích hợp nhánh `iss53` lại, giống như bạn đã làm với nhánh `hotfix` 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(-)
```

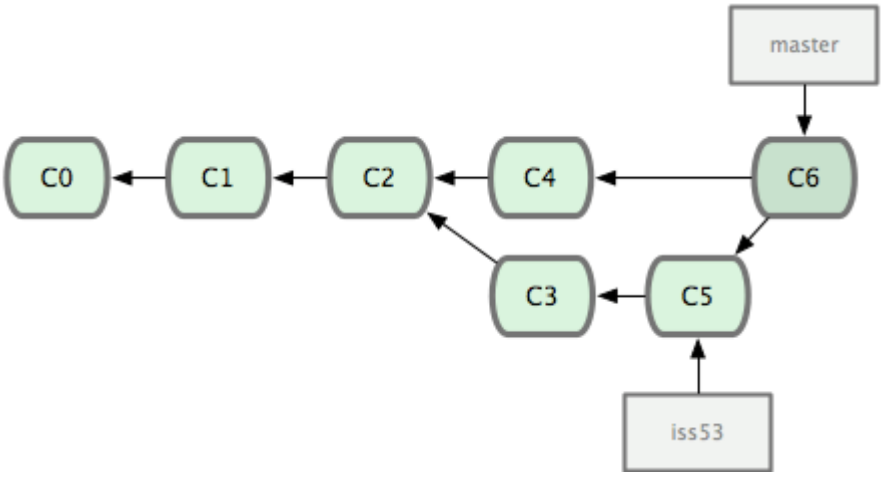
Lần này có hơi khác so với lần tích hợp `hotfix` trước đó. Trong trường hợp này, lịch sử phát triển của bạn đã bị phân nhánh tại một thời điểm nào đó trước kia. Bởi vì commit trên nhánh mà bạn đang làm việc (master) không phải là "cha" trực tiếp của nhánh mà bạn đang tích hợp vào, Git phải làm một số 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

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 `iss53` chính 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 `opendiff` cho 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.

3.3 Phân Nhánh Trong Git - Quản Lý Các Nhánh

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.

3.4 Phân Nhánh Trong Git - Quy Trình Làm Việc Phân Nhánh

Quy Trình Làm Việc Phân Nhánh

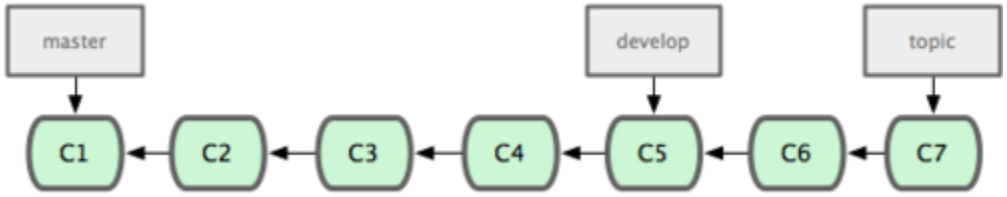
Bây giờ bạn đã có được các kiến thức cơ bản về phân nhánh và tích hợp, vậy bạn có thể hay nên làm gì với chúng. Trong phần này, chúng ta sẽ đề cập tới một số quy trình làm việc phổ biến áp dụng phân nhánh, vì thế bạn có thể tự quyết định có áp dụng chúng vào quy trình làm việc riêng của bạn hay không.

Nhánh Lâu Đời

Bởi vì Git sử dụng tích hợp 3 chiều đơn giản, nên tích hợp từ nhánh này vào nhánh khác nhiều lần trong cùng một giai đoạn thường dễ dàng. Có nghĩa là bạn có thể có nhiều nhánh luôn mở và sử dụng chúng cho các giai đoạn phát triển khác nhau; bạn có thể tích hợp từ một số nhánh nào đó vào các nhánh khác một cách thường xuyên.

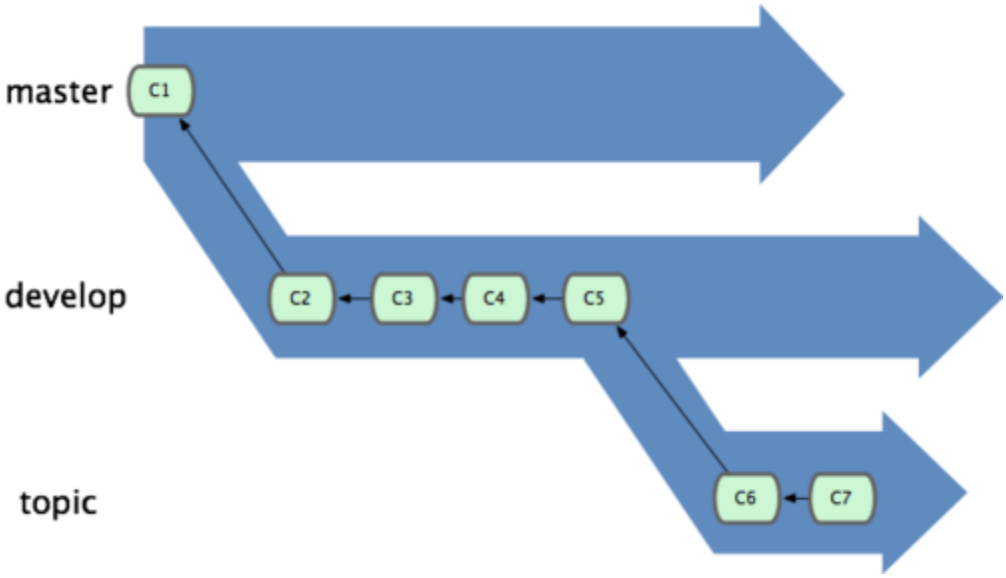
Nhiều lập trình viên Git sử dụng quy trình làm việc dựa theo phương pháp này, chẳng hạn như chỉ chứa mã nguồn ổn định hoàn toàn ở nhánh `master` - hầu như là mã nguồn đã phát hành hoặc chuẩn bị phát hành. Họ có một nhánh song song khác có tên develop hoặc next, nơi mà họ làm việc hoặc sử dụng để kiểm tra độ ổn định - nó không nhất thiết luôn luôn phải ổn định, tuy nhiên mỗi khi nó đạt được trạng thái ổn định, nó sẽ được tích hợp vào nhánh `master`. Chúng được sử dụng với vai trò là các nhánh chủ đề (topic branch) - các nhánh có vòng đời ngắn, giống như nhánh `iss53` trước đó - để đảm bảo chúng qua được các bài kiểm tra và không gây ra lỗi.

Trong thực tế, chúng ta đang nói về các con trỏ di chuyển dọc theo đường thẳng của các commit. Các nhánh ổn định hơn thường ở phía cuối của đường thẳng, còn các nhánh đang phát triển thường ở phía đầu hàng (xem Hình 3-18).



Hình 3-18. Nhánh ổn định hơn thường ở phía cuối hàng trong lịch sử commit.

Sẽ dễ hình dung hơn khi nghĩ về chúng như là các xi-lô, nơi mà tập hợp các commit cô đặc dần thành một xi-lô ổn định hơn khi đã được kiểm tra đầy đủ (xem Hình 3-19).



Hình 3-19. Có lẽ sẽ dễ hiểu hơn khi coi các nhánh là các xi-lô.

Bạn có thể tiếp tục làm theo cách này cho nhiều tầng ổn định khác nhau. Nhiều dự án lớn có nhánh `proposed` hoặc `pu` (proposed updates) được sử dụng cho các nhánh chưa đủ điều kiện để tích hợp vào `next` hoặc `master`. Ý tưởng ở đây là, các nhánh ở các tầng khác nhau của sự ổn định; khi chúng đạt tới một mức ổn định hơn nào đó, chúng sẽ được tích hợp vào tầng trên nó. Tóm lại, có nhiều nhánh tồn lại lâu dài không thật sự cần thiết, nhưng nó thường rất hữu ích, đặc biệt là khi bạn làm việc với các dự án lớn và phức tạp.

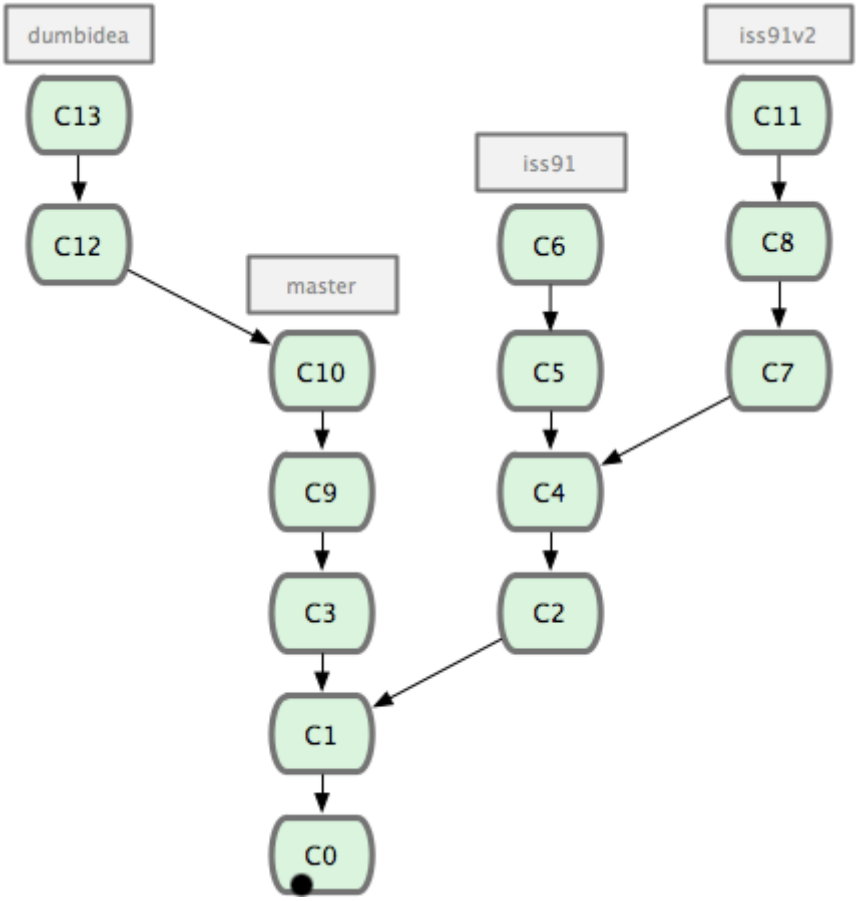
Nhánh Chủ Đề

Nhánh chủ đề (topic branches) thì ngược lại, nó lại khá hữu ích cho các dự án ở bất kỳ cỡ nào. Một nhánh chủ đề là nhánh có vòng đời ngắn mà bạn tạo để phát triển một tính năng nào đó hoặc tương tự. Nó giống như một thứ gì đó mà bạn chưa từng làm với một VCS trước đây bởi vì nhìn chung nó đòi hỏi rất nhiều nỗ lực để tạo mới cũng như tích hợp các nhánh lại với nhau.

Như bạn đã thấy trong phần trước với các nhánh `iss53` và `hotfix` bạn đã tạo ra. Bạn thực hiện một số commit trên đó và xóa chúng đi ngay sau khi tính hợp chúng lại với nhánh chính. Kỹ thuật này cho phép bạn chuyển ngữ cảnh một cách nhanh chóng và toàn diện - vì công việc của bạn tách biệt hoàn toàn ở các xi-lô nơi mà tất cả các thay đổi ở nhánh đó chỉ liên quan đến chủ đề đó, điều này khiến cho việc xem xét lại (review) mã nguồn hoặc tương tự trở nên dễ dàng hơn rất nhiều. Bạn có thể giữ các thay đổi ở đó trong bất kỳ khoảng thời gian nào bạn muốn, có thể tính bằng phút, ngày, hoặc tháng, và sau đó tích hợp lại khi chúng đã sẵn sàng, không quan trọng thứ tự chúng được tạo ra hay làm việc.

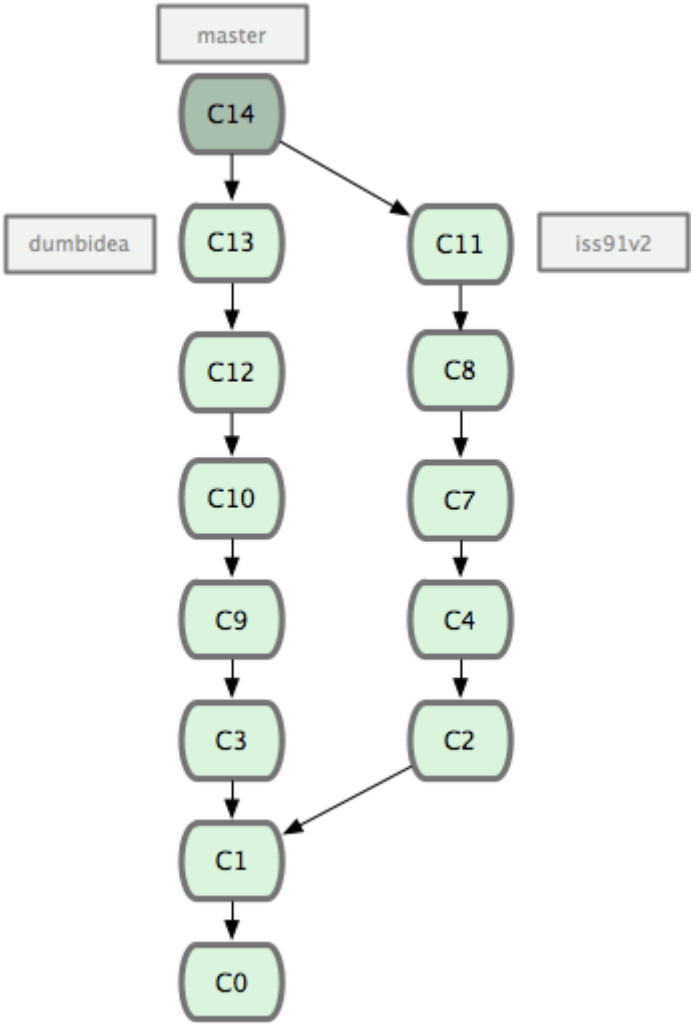
Hãy cùng xét một ví dụ về thực hiện một số công việc (trên nhánh `master`), tạo nhánh cho một vấn đề cần giải quyết (`iss91`), làm việc trên đó một chút, tạo một nhánh thứ hai cùng giải quyết vấn đề đó nhưng theo một cách khác (`iss91v2`), quay trở lại nhánh `master` và làm việc trong một khoảng thời gian nhất định, sau đó tạo một nhánh khác từ đó cho một ý tưởng mà bạn không

chắc chắn là nó có phải là ý hay hay không (nhánh `dumbidea`). Lúc này lịch sử commit của bạn sẽ giống Hình 3-20.



Hình 3-20. Lịch sử commit với nhiều nhánh chủ đề.

Bây giờ, giả sử bạn quyết định lựa chọn cách giải quyết thứ hai (`iss91v2`); và bạn trình bày ý tưởng `dumbidea` cho các đồng nghiệp, điều mà bạn không ngờ tới rằng mọi người lại cho đó là một ý tưởng tuyệt vời. Bạn đã có thể bỏ đi nhánh ban đầu `iss91` (mất commit C5 và C6) và tích hợp hai commit còn lại. Lịch sử của bạn lúc này sẽ giống Hình 3-21.



Hình 3-21. Lịch sử commit sau khi tích hợp dumbidea và iss91v2.

Ghi nhớ một điều quan trọng là khi bạn làm tất cả những việc này, các nhánh hoàn toàn nằm ở máy nội bộ. Khi bạn phân nhánh và tích hợp, tất cả mọi thứ xảy ra trên kho chứa Git của bạn - không có giao tiếp tới máy chủ nào xảy ra.

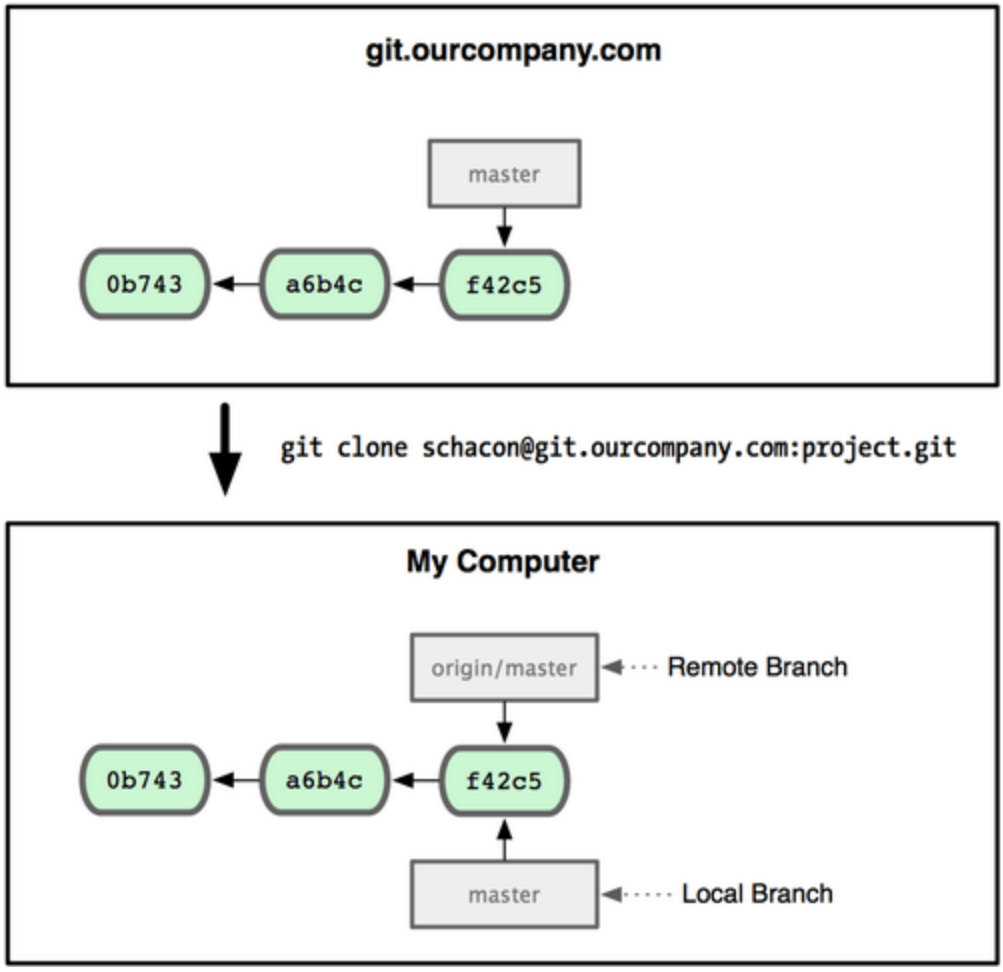
3.5 Phân Nhánh Trong Git - Nhánh Remote

Nhánh Remote

Nhánh từ xa (remote) là các tham chiếu tới trạng thái của các nhánh trên kho chứa trung tâm của bạn. Chúng là các nhánh nội bộ mà bạn không thể di chuyển; chúng chỉ di chuyển một cách tự động mỗi khi bạn thực hiện bất kỳ giao tiếp nào qua mạng lưới. Nhánh remote hoạt động như là các bookmark (dấu) để nhắc nhở bạn các nhánh trên kho chứa trung tâm của bạn ở đâu vào lần cuối cùng bạn kết nối tới.

Chúng có dạng `(remote)/(branch)`. Ví dụ, nếu bạn muốn xem nhánh `master` trên nhánh remote `origin` của bạn như thế nào từ lần giao tiếp cuối cùng, bạn sẽ dùng `origin/master`. Nếu bạn đang giải quyết một vấn đề với đối tác và họ đẩy dữ liệu lên nhánh `iss53`, bạn có thể có riêng nhánh `iss53` trên máy nội bộ; nhưng nhánh trên máy chủ sẽ trỏ tới commit tại `origin/iss53`.

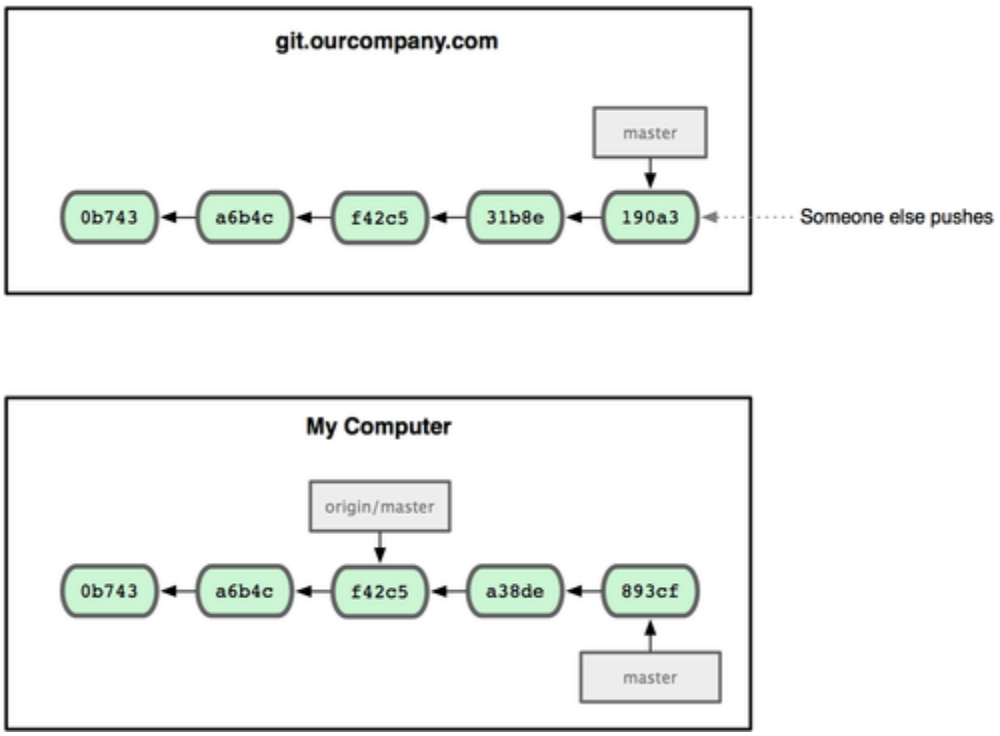
Điều này có thể hơi khó hiểu một chút, vậy hãy cùng xem một ví dụ. Giả sử bạn có một máy chủ Git trên mạng của bạn tại địa chỉ `git.ourcompany.com`. Nếu bạn tạo bản sao từ đây, Git sẽ tự động đặt tên nó là `origin` cho bạn, tải về toàn bộ dữ liệu, tạo một con trỏ tới nhánh `master` và đặt tên nội bộ cho nó là `origin/master`; và bạn không thể di chuyển nó. Git cũng cung cấp cho bạn nhánh `master` riêng, bắt đầu cùng một vị trí với `master` của origin để cho bạn có thể bắt đầu làm việc (xem Hình 3-22).



Hình 3-22. Một bản sao Git cung cấp cho bạn nhánh master riêng và nhánh origin/master trỏ tới nhánh master của origin.

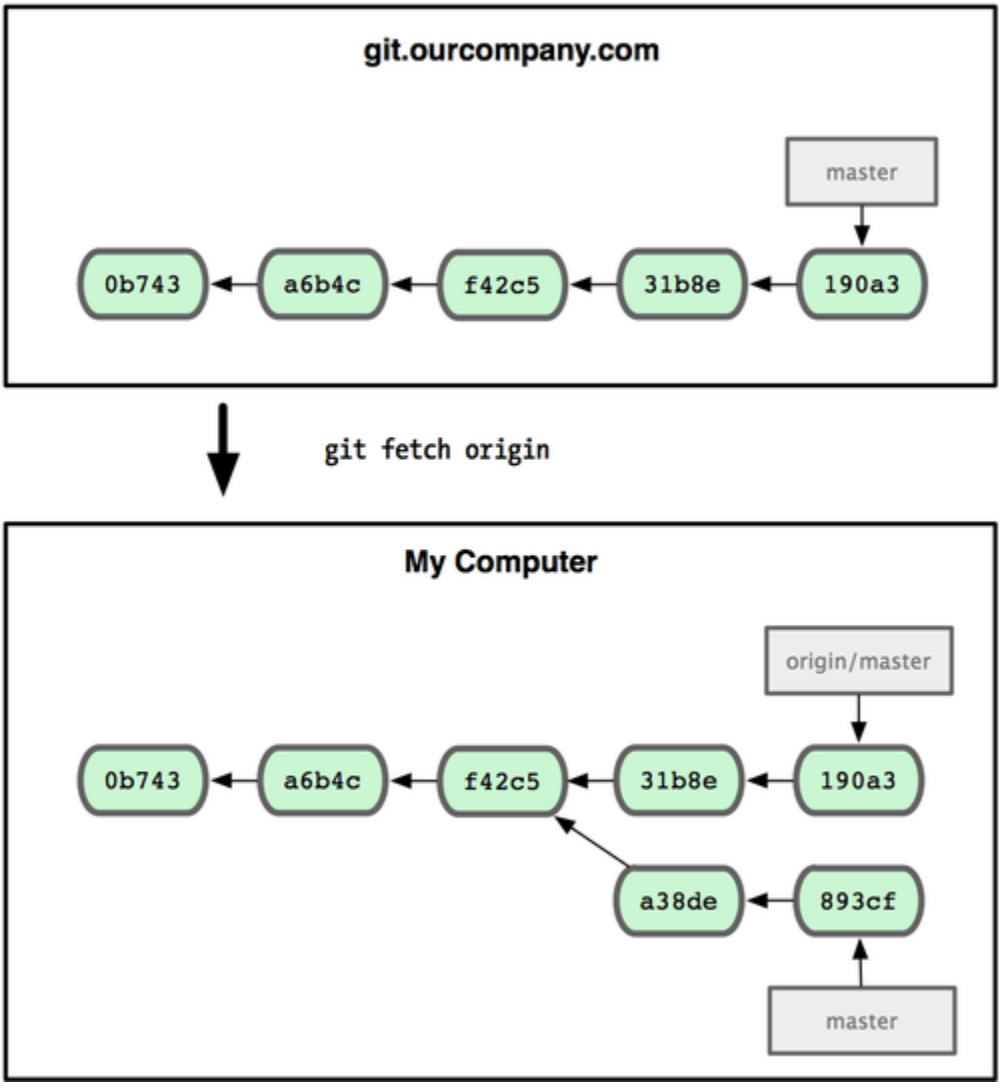
Nếu bạn thực hiện một số thay đổi trên nhánh `master` nội bộ, và cùng thời điểm đó, một người nào đó đẩy lên `git.ourcompany.com` và cập nhật nhánh master của nó, thì lịch sử của bạn sẽ di

chuyển về phía trước khác đi. Miễn là bạn không kết nối tới máy chủ thì con trỏ `origin/master` sẽ vẫn không đổi (xem Hình 3-23).



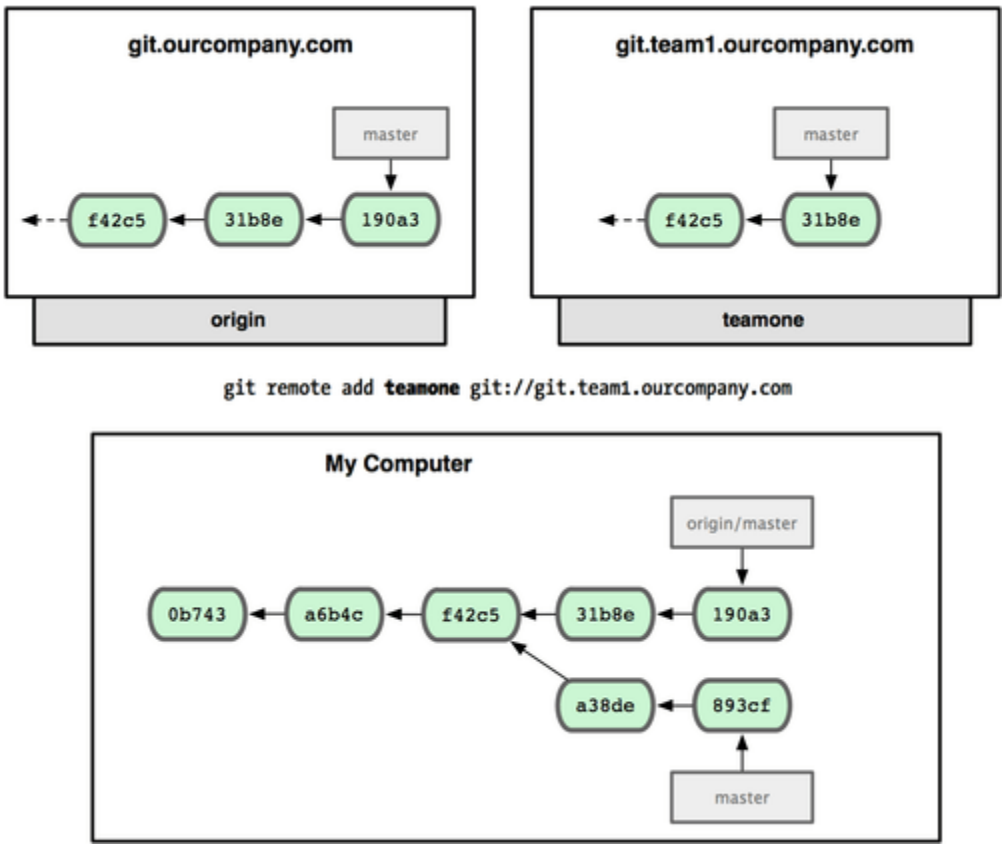
Hình 3-23. Làm việc nội bộ và ai đó đẩy lên máy chủ khiến cho lịch sử thay đổi khác biệt nhau.

Để đồng bộ hóa các thay đổi, bạn chạy lệnh `git fetch origin`. Lệnh này sẽ tìm kiếm máy chủ nào là origin (trong trường hợp này là `git.ourcompany.com`), truy xuất toàn bộ dữ liệu mà bạn chưa có từ đó, và cập nhật cơ sở dữ liệu nội bộ của bạn, di chuyển con trỏ `origin/master` tới vị trí mới được cập nhật (xem Hình 3-24).



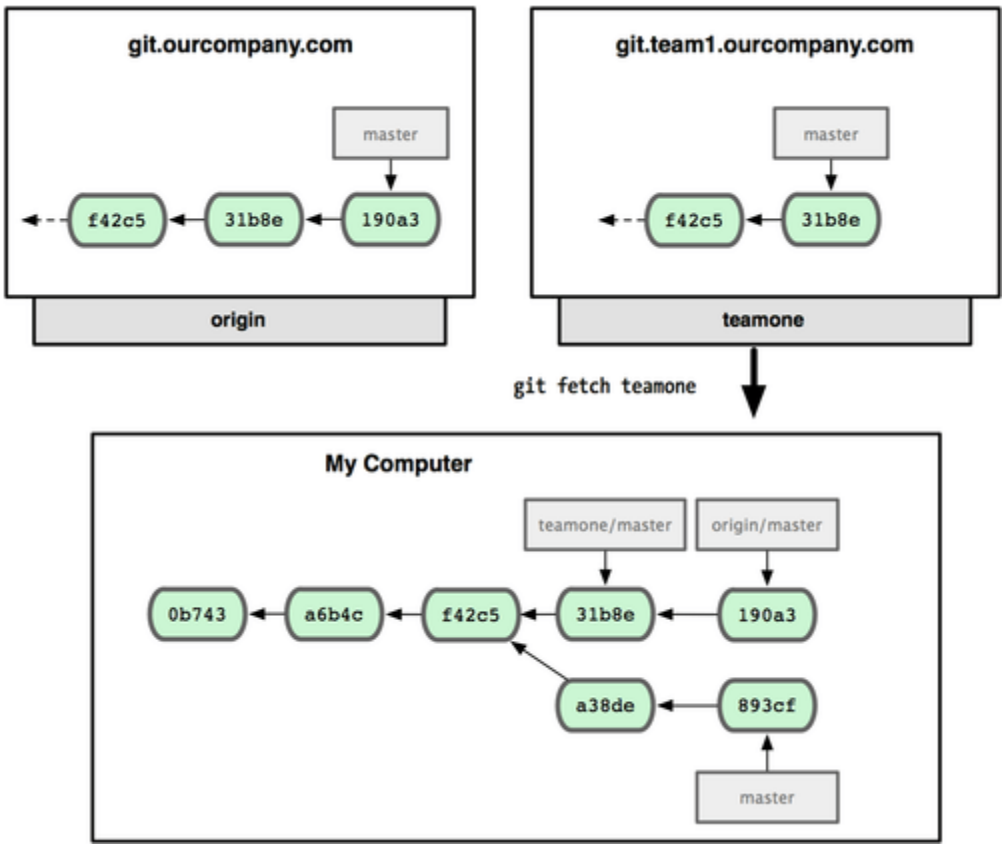
Hình 3-24. Lệnh git fetch cập nhật các tham chiếu từ xa.

Để minh họa cho việc có nhiều máy chủ từ xa và các nhánh từ xa của các dự án thuộc các máy chủ đó, giả sử bạn có một máy chủ Git nội bộ khác sử dụng riêng cho các nhóm "thần tốc". Máy chủ này có địa chỉ là `git.team1.ourcompany.com`. Bạn có thể thêm nó như là một tham chiếu từ xa tới dự án bạn đang làm việc bằng cách chạy lệnh `git remote add` như đã giới thiệu ở Chương 2. Đặt tên cho remote đó là `teamone`, đó sẽ là tên rút gọn thay thế cho địa chỉ đầy đủ kia (xem Hình 3-25).



Hình 3-25. Thêm một máy chủ từ xa khác.

Bây giờ bạn có thể chạy lệnh `git fetch teamone` để truy xất toàn bộ nội dung mà bạn chưa có từ máy chủ `teamone`. Bởi vì máy chủ đó có chứa một tập con dữ liệu từ máy chủ `origin` đang có, Git không truy xuất dữ liệu nào cả mà thiết lập một nhánh từ xa mới là `teamone/master` để trỏ tới commit mà `teamone` đang có như là nhánh `master` (xem Hình 3-26).



Hình 3-26. Bạn sẽ có một tham chiếu tới vị trí nội bộ của nhánh `master` của teamone.

Đẩy Lên

Khi bạn muốn chia sẻ một nhánh với mọi người, bạn cần phải đẩy nó lên một máy chủ mà bạn có quyền ghi trên đó. Nhánh nội bộ của bạn sẽ không tự động thực hiện quá trình đồng bộ hóa - mà bạn phải tự đẩy lên cách nhánh mà bạn muốn chia sẻ. Theo cách này, bạn có thể có các nhánh riêng tư cho những công việc mà bạn không muốn chia sẻ, và chỉ đẩy lên các nhánh chủ đề mà bạn muốn mọi người cùng tham gia đóng góp.

Nếu bạn có một nhánh là `serverfix` mà bạn muốn mọi người cùng cộng tác, bạn có thể đẩy nó lên theo cách mà chúng ta đã làm đối với nhánh đầu tiên. Chạy `git push (remote) (branch)`:

```
$ git push origin serverfix
Counting objects: 20, done.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (15/15), 1.74 KiB, done.
Total 15 (delta 5), reused 0 (delta 0)
To git@github.com:schacon/simplegit.git
 * [new branch]      serverfix -> serverfix
```

Đây là một cách làm tắt. Git tự động mở rộng nhánh `serverfix` thành `refs/heads/serverfix:refs/heads/serverfix`, có nghĩa là, "Hãy sử dụng nhánh nội bộ

serverfix của tôi và đẩy nó lên để cập nhật nhánh serverfix trên máy chủ từ xa." Chúng ta sẽ đi sâu vào phần `refs/heads/` ở Chương 9, nhưng bạn thường có thể bỏ qua nó. Bạn cũng có thể chạy lệnh sau `git push origin serverfix:serverfix`, cách này cũng cho kết quả tương tự - nó có nghĩa là "Hãy sử dụng serverfix của tôi để tạo một serverfix trên máy chủ". Bạn có thể sử dụng định dạng này để đẩy một nhánh nội bộ lên một nhánh từ xa với một tên khác. Nếu bạn không muốn gọi nó là `serverfix` trên máy chủ, bạn có thể chạy lệnh sau `git push origin serverfix:awesomebranch` để đẩy nhánh nội bộ `serverfix` vào nhánh `awesomebranch` trên máy chủ trung tâm.

Lần tới một trong các đồng nghiệp của bạn truy xuất nó từ trên máy chủ, họ sẽ có một tham chiếu tới phiên bản trên máy chủ của `serverfix` dưới tên `origin/serverfix`:

```
$ git fetch origin
remote: Counting objects: 20, done.
remote: Compressing objects: 100% (14/14), done.
remote: Total 15 (delta 5), reused 0 (delta 0)
Unpacking objects: 100% (15/15), done.
From git@github.com:schacon/simplegit
 * [new branch]      serverfix    -> origin/serverfix
```

Điều quan trọng cần chú ý ở đây là khi bạn truy xuất dữ liệu từ máy chủ mà có kèm theo nhánh mới, Git sẽ không tự động tạo phiên bản nội bộ của nhánh đó. Nói cách khác, trong trường hợp này, bạn sẽ không có nhánh `serverfix` mới - bạn chỉ có một con trỏ tới `origin/serverfix` mà bạn không thể chỉnh sửa.

Để tích hợp công việc hiện tại vào nhánh bạn đang làm việc, bạn có thể chạy `git merge origin/serverfix`. Nếu bạn muốn nhánh `serverfix` riêng để có thể làm việc trên đó, bạn có thể tách nó ra khỏi nhánh trung tâm bằng cách:

```
$ git checkout -b serverfix origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

Cách này sẽ tạo cho bạn một nhánh nội bộ mà bạn có thể làm việc, bắt đầu cùng một vị trí với `origin/serverfix`.

Theo Dõi Các Nhánh

Check out một nhánh nội bộ từ một nhánh trung tâm tự động tạo ra một *tracking branch*. Tracking branches là các nhánh nội bộ có liên quan trực tiếp với một nhánh trung tâm. Nếu bạn đang ở trên một tracking branch và chạy `git push`, Git tự động biết nó sẽ phải đẩy lên nhánh nào, máy chủ nào. Ngoài ra, chạy `git pull` khi đang ở trên một trong những nhánh này sẽ truy xuất toàn bộ các tham chiếu từ xa và sau đó tự động tích hợp chúng với các nhánh từ xa tương ứng.

Khi bạn tạo bản sao của một kho chứa, thông thường Git tự động tạp một nhánh `master` để theo dõi `origin/master`. Đó là lý do tại sao `git push` và `git pull` có thể chạy tốt mà không cần bất kỳ tham số nào. Tuy nhiên, bạn có thể cài đặt các tracking branch khác nếu muốn - các nhánh

này không theo dõi nhánh trên `origin` cũng như `master`. Một ví dụ đơn giản giống như bạn vừa thấy: `git checkout -b [branch] [remotename]/[branch]`. Nếu bạn đang sử dụng Git phiên bản 1.6.2 trở lên, bạn có thể sử dụng `--track`:

```
$ git checkout --track origin/serverfix
Branch serverfix set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "serverfix"
```

Để cài đặt một nhánh nội bộ sử dụng tên khác với tên mặc định trên nhánh trung tâm, bạn có thể dễ dàng sử dụng phiên bản đầu tiên với một tên nội bộ khác:

```
$ git checkout -b sf origin/serverfix
Branch sf set up to track remote branch refs/remotes/origin/serverfix.
Switched to a new branch "sf"
```

Bây giờ, nhánh nội bộ sf sẽ tự động "kéo và đẩy" từ origin/serverfix.

Xóa Nhánh Trung Tâm

Giả sử bạn và đồng nghiệp đã hoàn thành một chức năng nào đó và đã tích hợp nó vào nhánh `master` trung tâm (hoặc bất kỳ nhánh nào khác sử dụng cho việc lưu trữ các phiên bản ổn định). Bạn có thể xóa một nhánh trung tâm đi sử dụng cú pháp sau `git push [remotename] :[branch]`. Nếu bạn muốn xóa nhánh `serverfix` trên máy chủ, bạn có thể chạy lệnh sau:

```
$ git push origin :serverfix
To git@github.com:schacon/simplegit.git
 - [deleted]         serverfix
```

Vậy là đã xong, nhánh đó đã bị xóa khỏi máy chủ. Có thể bạn muốn đánh dấu trang này lại, vì bạn sẽ cần đến câu lệnh này và có thể bạn sẽ quên cú pháp của nó. Một cách để nhớ lệnh này là xem lại cú pháp chúng ta đã nhắc tới trước đó `git push [remotename] [localbranch]:[remotebranch]`. Nếu bạn bỏ qua phần `[localbranch]`, thì cơ bản bạn đang thực hiện "Không sử dụng gì từ phía nội bộ để tạo nhánh `[remotebranch]`."

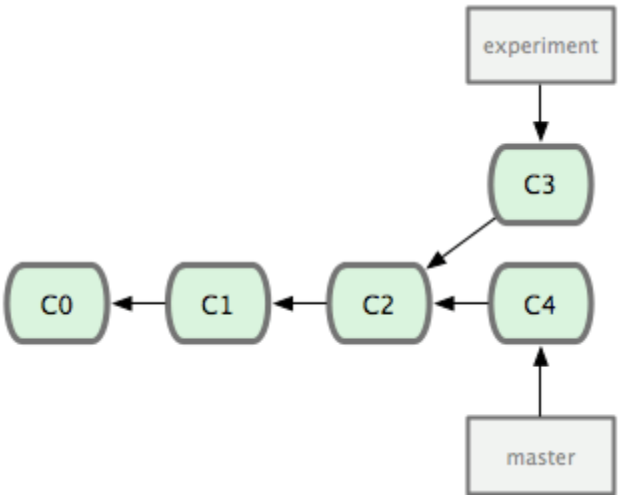
3.6 Phân Nhánh Trong Git - Rebasing

Rebasing

Trong Git, có hai cách chính để tích hợp các thay đổi từ nhánh này vào nhánh khác: đó là `merge` và `rebase`. Trong phần này bạn sẽ được tìm hiểu rebase là gì, sử dụng nó như thế nào, tại sao nó được coi là một công cụ khá tuyệt vời, và trong trường hợp nào thì không nên sử dụng nó.

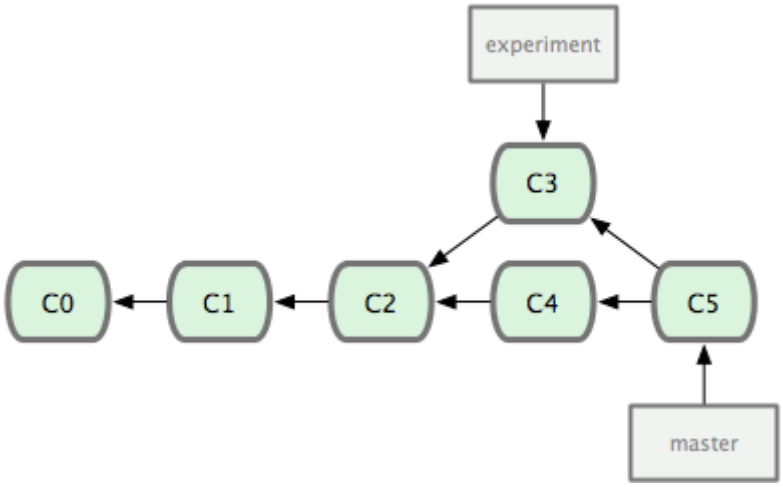
Cơ Bản về Rebase

Nếu bạn xem lại ví dụ trước trong phần Tích Hợp (xem Hình 3-27), bạn có thể thấy rằng bạn đã phân nhánh công việc của bạn và thực hiện commit trên hai nhánh khác nhau.



Hình 3-17. Lần phân nhánh đầu tiên.

Cách đơn giản nhất để tích hợp các nhánh, như chúng ta đã đề cập từ trước, đó là lệnh `merge`. Nó thực hiện tích hợp 3-chiều giữa hai snapshot mới nhất của hai nhánh (C3 và C4) và cha chung gần nhất của cả hai (C2), tạo mới một snapshot khác (và commit), như trong Hình 3-28.



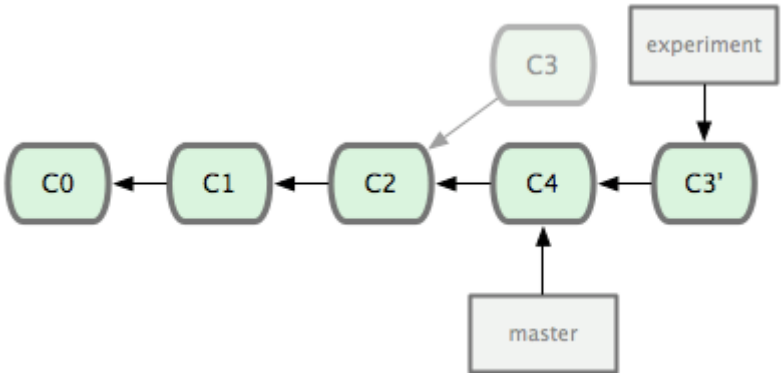
Hình 3-28. Gộp nhánh lại để hợp nhất công việc bị tách ra trước đây.

Tuy nhiên, còn có một cách khác: bạn có thể sử dụng bản vá của thay đổi được đưa ra ở C3 và áp dụng nó lên trên C4. Trong Git, đây được gọi là *rebasing*. Bằng cách sử dụng lệnh `rebase`, bạn có thể sử dụng tất cả các thay đổi được commit ở một nhánh và "chạy lại" (replay) chúng trên một nhánh khác.

Trong ví dụ này, bạn thực hiện như sau:

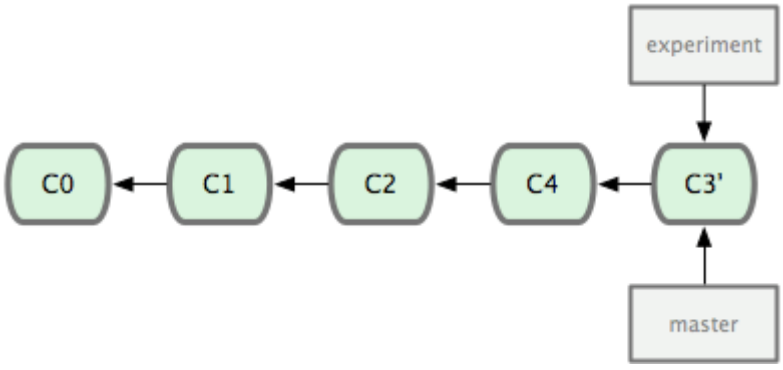
```
$ git checkout experiment
$ git rebase master
First, rewinding head to replay your work on top of it...
Applying: added staged command
```

Nó thực hiện bằng cách đi tới commit cha chung của hai nhánh (nhánh bạn đang làm việc và nhánh bạn đang muốn rebase), tìm sự khác biệt trong mỗi commit của nhánh mà bạn đang làm việc, lưu lại các thay đổi đó vào một tập tin tạm thời, khôi phục lại nhánh hiện tại về cùng một commit với nhánh bạn đang rebase, và cuối cùng áp dụng lần lượt các thay đổi. Hình 3-29 minh họa toàn bộ quá trình này.



Hình 3-29. Quá trình rebase thay đổi ở C3 vào C4.

Đến lúc này, bạn có thể quay lại nhánh `master` và thực hiện fast-forward merge (xem Hình 3-30).



Hình 3-30. Di chuyển nhánh master lên phía trước.

Bây giờ snapshot mà C3' trỏ tới cũng giống như snapshot được trỏ tới bởi C5 trong ví dụ sử dụng merge. Không có sự khác biệt nào khi so sánh kết quả của hai phương pháp này, nhưng sử dụng rebase sẽ cho chúng ta lịch sử rõ ràng hơn. Nếu bạn xem xét lịch sử của nhánh mà chúng ta

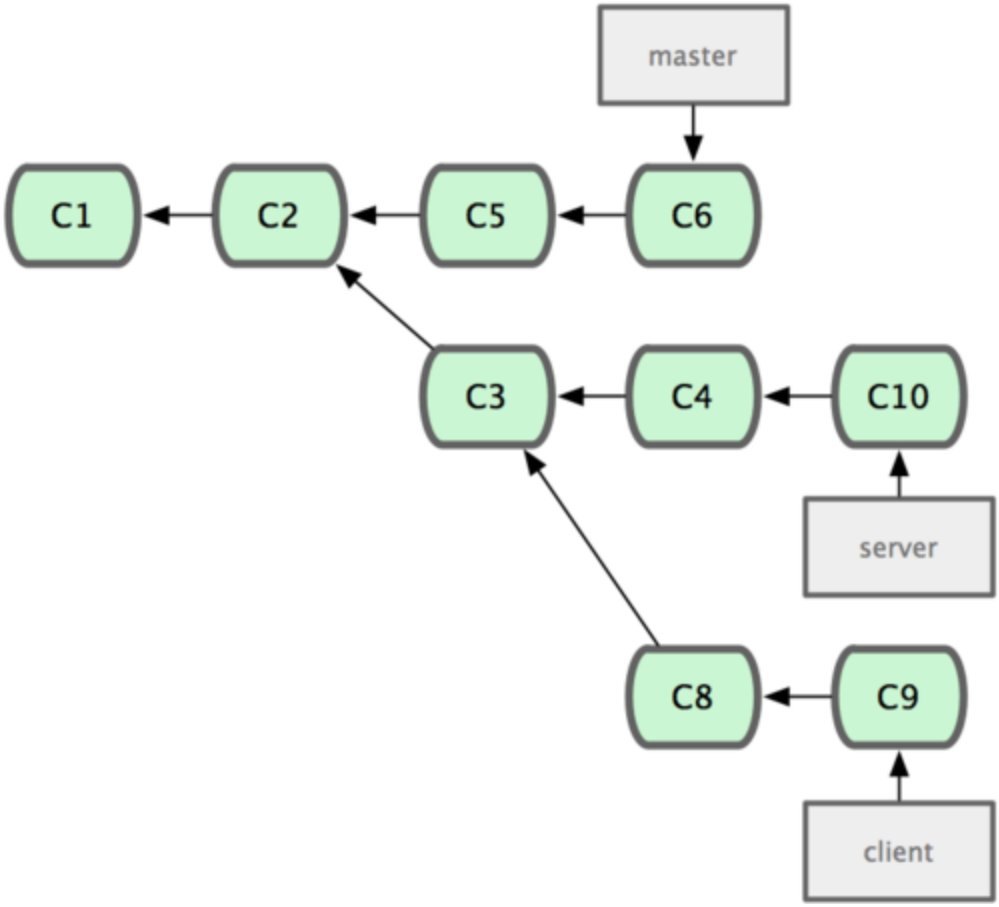
rebase vào, nó giống như một đường thẳng: mọi thứ dường như xảy ra theo trình tự, thậm chí ban đầu nó diễn ra song song.

Bình thường, bạn sử dụng cách này để đảm bảo rằng các commit được áp dụng một cách rõ ràng, rành mạch trên nhánh remote - có lẽ là một dự án mà bạn đang đóng góp chứ không phải duy trì nó. Trong trường hợp này, bạn thực hiện công việc trên một nhánh và sau đó rebase trở lại nhánh `origin/master` khi đã sẵn sàng. Theo cách này thì người duy trì dự án đó không phải thực hiện việc tích hợp - mà chỉ chi chuyển tiến lên phía trước (fast-forwar) hoặc đơn giản là áp dụng chúng vào.

Lưu ý rằng snapshot được trỏ tới bởi commit cuối cùng, cho dù nó là kết quả của việc rebase hay merge, thì nó vẫn giống nhau - chỉ khác nhau về các bước thực hiện mà thôi. Quá trình rebase được thực hiện bằng cách thực hiện lại các thay đổi từ nhánh này qua nhánh khác theo thứ tự chúng đã được thực hiện, trong khi đó merge lại lấy hai điểm kết thúc và gộp chúng lại với nhau.

Rebase Nâng Cao

Bạn cũng có thể thực hiện rebase trên một đối tượng khác mà không phải là nhánh rebase. Xem ví dụ Hình 3-31. Bạn tạo một nhánh chủ để (`server`) để thêm một số tính năng server-side vào dự án, và thực hiện một số commit. Sau đó bạn tạo một nhánh khác để thực hiện một số thay đổi cho phía client (`client`) và cũng commit vài lần. Cuối cùng, bạn quay trở lại nhánh server và thực hiện thêm một số commit nữa.

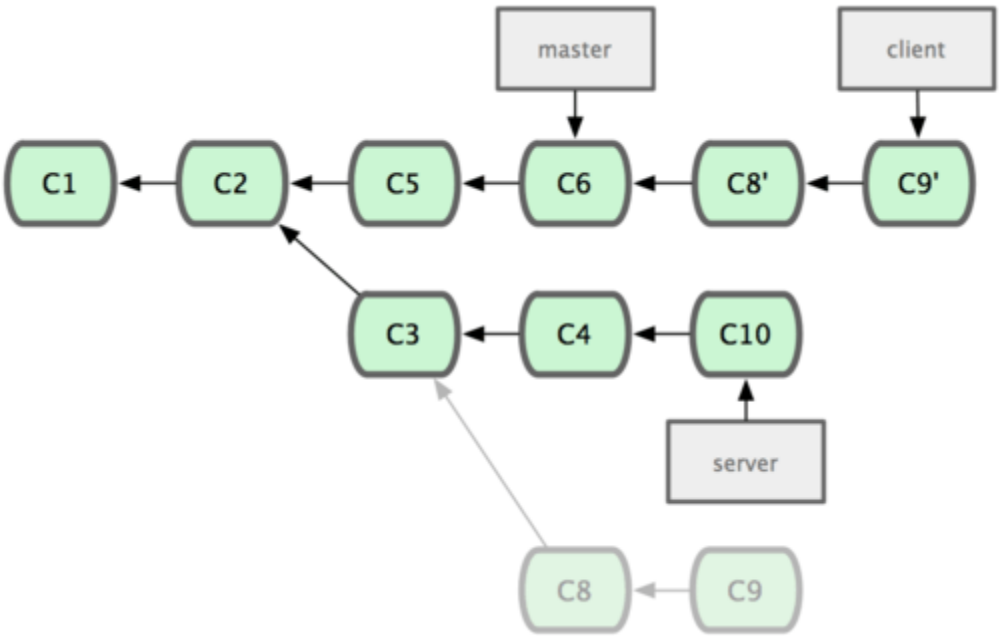


Hình 3-31. Nhánh chủ đề được tạo từ một nhánh chủ đề khác.

Giả sử bạn quyết định tích hợp các thay đổi phía client vào nhánh chính cho bản phát hành sắp tới, nhưng bạn vẫn muốn giữ các thay đổi server-side cho đến khi nó được kiểm tra kỹ lưỡng. Bạn có thể lấy các thay đổi ở client mà không có mặt ở server (C8 và C9) sau đó chạy lại (replay) chúng trên nhánh master bằng cách sử dụng lựa chọn `--onto` cho lệnh `git rebase`:

```
$ git rebase --onto master server client
```

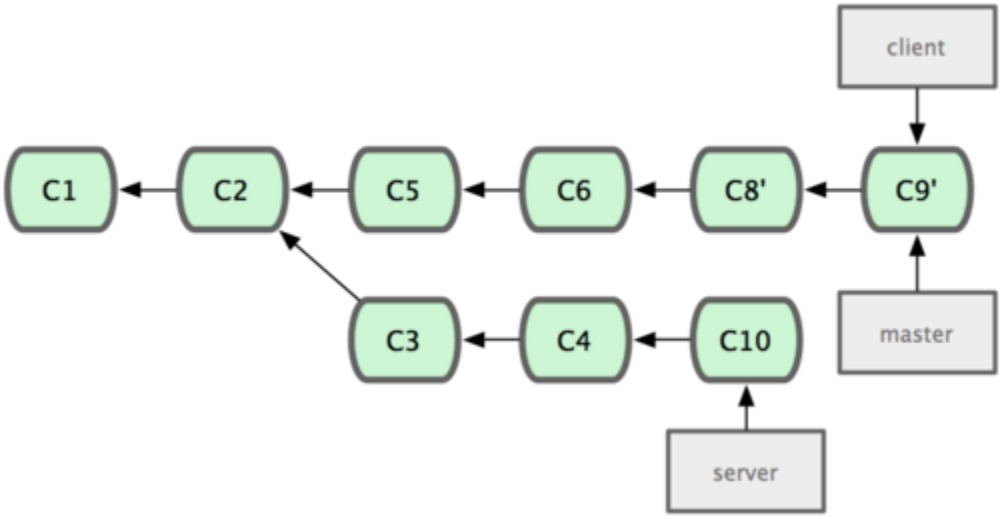
Lệnh này cơ bản nói rằng, "Hãy check out nhánh client, tìm ra các bản vá từ commit chung của nhánh `client` và `server`, sau đó thực thi lại vào nhánh `master`." Nó hơi phức tạp một chút nhưng kết quả như Hình 3-32 thì lại rất tuyệt.



Hình 3-32. Quá trình rebase nhánh chủ đề khỏi một nhánh chủ đề khác.

Bây giờ bạn có thể di chuyển con trỏ của nhánh master tiến lên phía trước (xem Hình 3-33):

```
$ git checkout master
$ git merge client
```

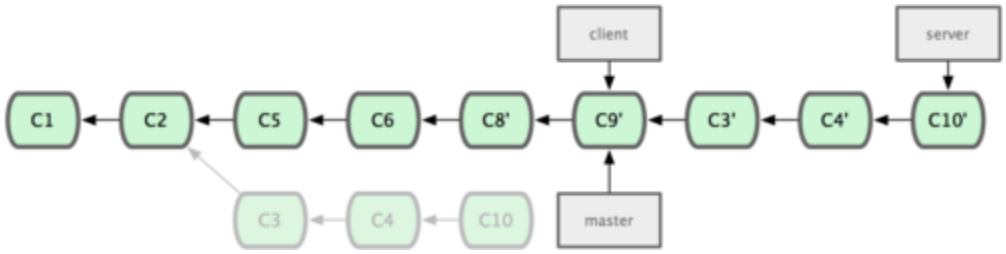


Hình 3-33. Di chuyển nhánh master lên phía trước để bao gồm các thay đổi của nhánh client.

Giả sử rằng bạn quyết định kéo về cả nhánh trên máy chủ. Bạn có thể rebase nhánh trên máy chủ đó vào nhánh master mà không phải checkout trước bằng lệnh `git rebase [basebranch] [topicbranch]` - lệnh này sẽ checkout nhánh chủ đề (trong trường hợp này là `server`) cho bạn và áp dụng lại các thay đổi vào nhánh cơ sở (base) `master`:

```
$ git rebase master server
```

Lệnh này sẽ thực hiện lại các thay đổi trên nhánh `server` chèn vào nhánh `master` như trong Hình 3-34.



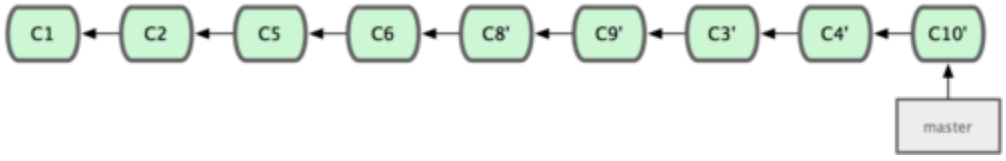
Hình 3-34. Rebase nhánh server chèn lên nhánh master.

Sau đó bạn có thể di chuyển con trỏ nhánh base (`master`):

```
$ git checkout master
$ git merge server
```

Bạn có thể xóa nhánh `client` và `server` vì tất cả công việc đã được tích hợp vào master và bạn không cần đến chúng nữa, lịch sử quả toàn bộ quá trình vừa rồi giống như Hình 3-35:

```
$ git branch -d client
$ git branch -d server
```



Hình 3-35. Lịch sử commit cuối cùng.

Rủi Ro của Rebase

Mặc dù rebase rất hữu ích nhưng nó cũng có không ít những mặt hạn chế, điều này có thể tổng kết bằng câu sau đây:

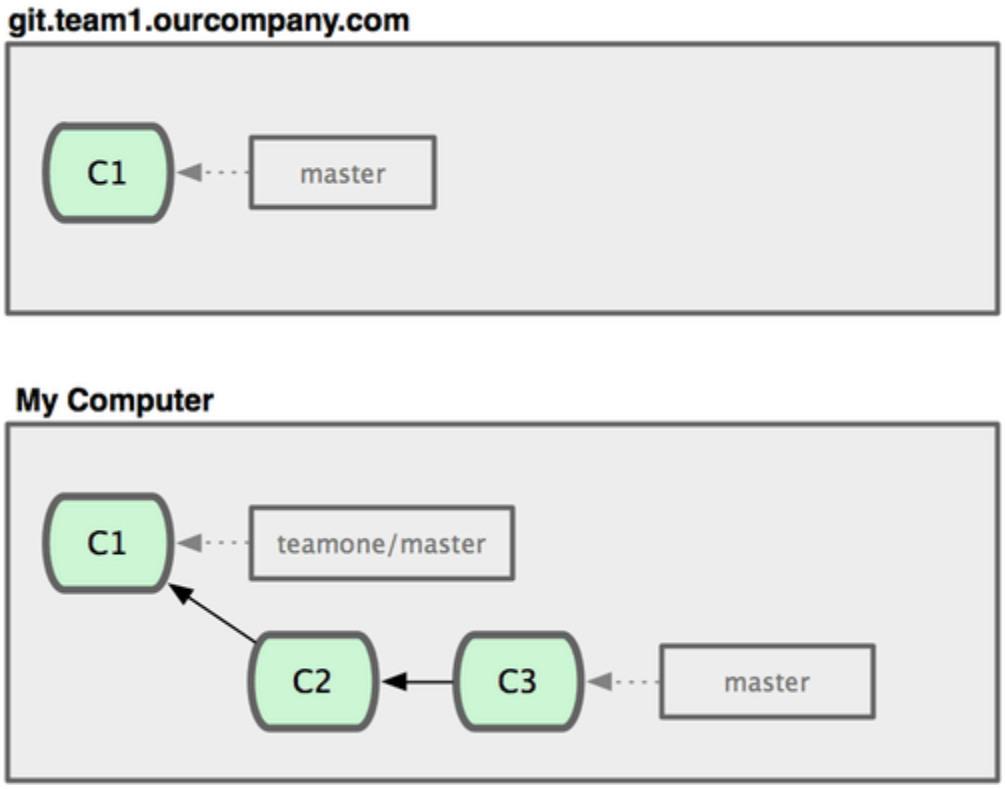
Không được rebase các commit mà bạn đã đẩy lên một kho chứa công khai.

Miễn là bạn làm theo hướng dẫn này, sẽ không có chuyện gì xảy ra. Nếu không, mọi người sẽ ghét bạn, và bạn sẽ bị bạn bè và gia đình coi thường.

Khi bạn thực hiện rebase, bạn đang bỏ đi các commit đã tồn tại và tái tạo lại các commit mới tương tự nhưng thực ra khác biệt. Nếu bạn đẩy commit ở một nơi nào đó và mọi người kéo xuống máy của họ, sau đó bạn sửa lại các commit đó bằng lệnh `git rebase` và đẩy lên một lần

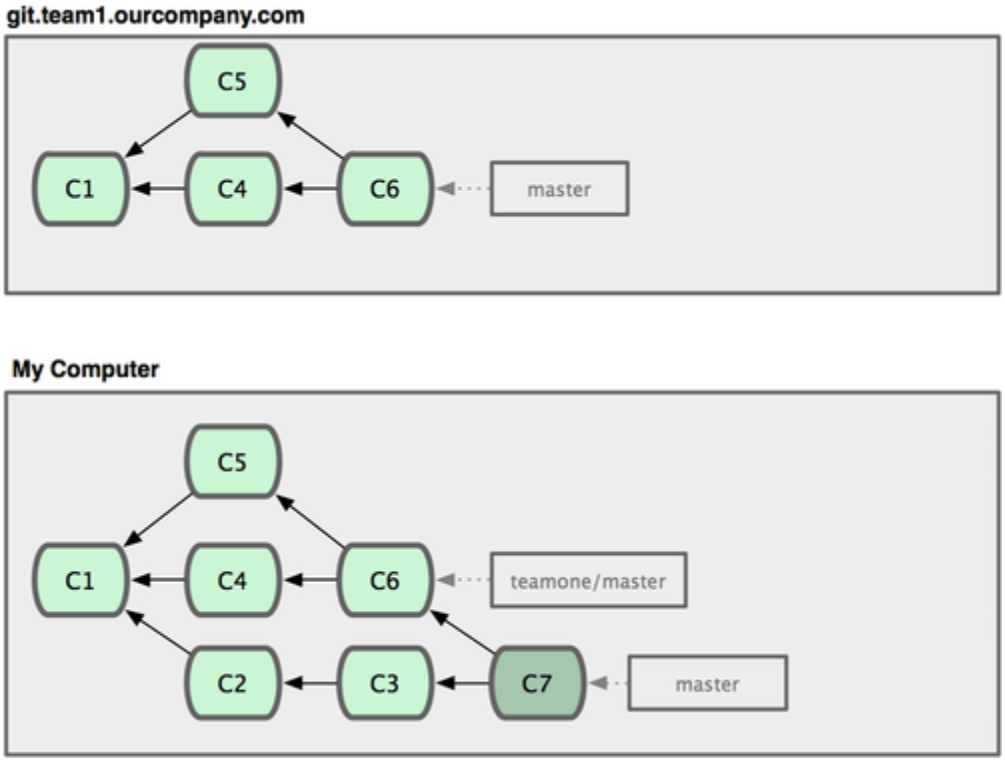
nữa, đồng nghiệp của bạn sẽ phải tích hợp lại công việc của họ và mọi thứ sẽ rối tung lên khi bạn cố gắng kéo các thay đổi của họ ngược lại máy bạn.

Hãy cùng xem một ví dụ làm sao việc rebase công khai có thể gây sự cố. Giả sử bạn tạo bản sao từ một máy chủ trung tâm và thực hiện một số thay đổi từ đó. Lịch sử commit của bạn sẽ giống như Hình 3-36.



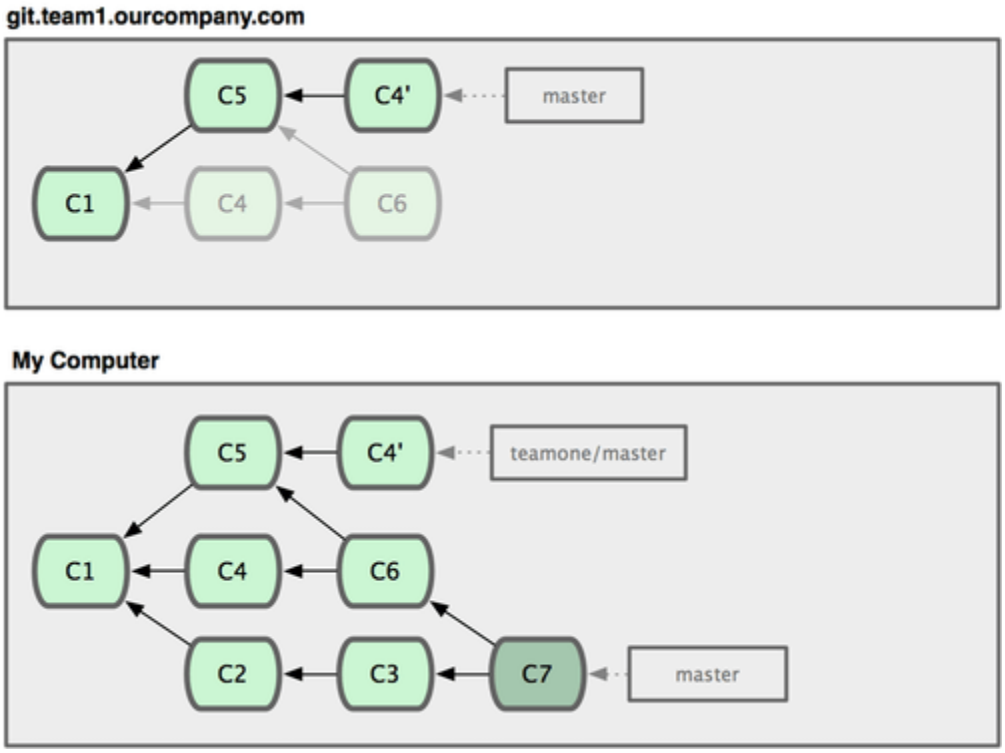
Hình 3-36. Tạo bản sao một kho chứa, và base một số thay đổi vào đó.

Bây giờ, một người khác thực hiện một số thay đổi khác có kèm theo một lần tích hợp (merge), và đẩy lên máy chủ trung tâm. Bạn truy xuất chúng và tích hợp nhánh trung tâm mới đó vào của bạn, lúc này lịch sử của bạn sẽ giống như Hình 3-37.



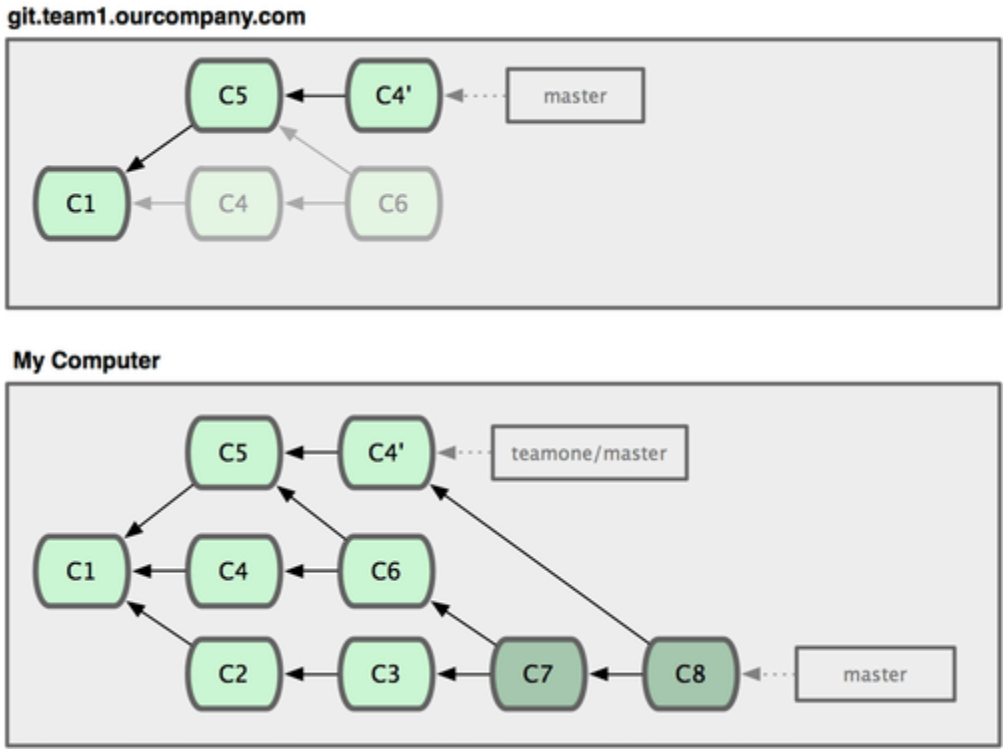
Hình 3-37. Truy xuất thêm các commit và tích hợp lại.

Tiếp theo, người đã đẩy tích hợp đó quyết định lại và rebase lại những thay đổi của họ; họ thực hiện `git push --force` để ghi đè lịch sử trên máy chủ. Sau đó bạn truy xuất lại dữ liệu từ máy chủ, đưa về các commit mới.



Hình 3-38. Một người nào đó đẩy lên các commit rebase, bỏ đi các commit có chứa thay đổi của bạn.

Lúc này, bạn phải tích hợp lại một lần nữa các thay đổi này, mặc dù trước đó bạn đã làm rồi. Quá trình rebase thay đổi mã băm SHA-1 của các commit này vì thế đối với Git chúng giống như các commit mới, mà thực tế thì bạn đã có C4 trong lịch sử của bạn (xem Hình 3-39).



Hình 3-39. Bạn tích hợp các thay đổi tương tự lại một lần nữa vào một commit tích hợp mới.

Bạn phải tích hợp thay đổi đó để có thể theo kịp với các lập trình viên khác về sau này. Sau khi thực hiện việc này, lịch sử commit của bạn sẽ bao gồm cả hai commit C4 và C4' có mã SHA-1 khác nhau nhưng lại có cùng chung nội dung thay đổi cũng như thông điệp commit. Nếu bạn chạy lệnh `git log` trong trường hợp này bạn sẽ thấy hai commit cùng chung ngày commit và thông điệp, điều này sẽ gây khó hiểu cho bạn. Hơn nữa, nếu bạn đẩy chúng ngược lên máy chủ, bạn sẽ đưa vào một lần nữa tất cả các commit đã rebase đó và sẽ gây khó hiểu cho nhiều người khác nữa.

Nếu bạn sử dụng rebase như là cách để dọn dẹp các commit trước khi đẩy chúng lên, và nếu như bạn chỉ rebase commit chưa bao giờ được công khai, thì sẽ không có chuyện gì xảy ra. Nếu bạn rebase các commit đã được công khai và mọi người có thể đã tích hợp (base) nó vào công việc của họ thì bạn có thể gặp phải các vấn đề thực sự khó chịu.

3.7 Phân Nhánh Trong Git - Tổng Kết

Tổng Kết

Chúng ta đã đề cập tới các khái niệm cơ bản về phân nhánh và tích hợp trong Git. Bạn nên nắm vững việc tạo mới, di chuyển giữa các nhánh và tích hợp các nhánh nội bộ lại với nhau. Bạn cũng nên có khả năng chia sẽ các nhánh bằng cách đẩy chúng lên một máy chủ trung tâm, cộng tác với các thành viên khác trên các nhánh dùng chung và rebase chúng trước khi chia sẻ.