Git Magic

Lynn, Ben

Git Magic

Lynn, Ben

Mục lục

Lời nói đầu

Git [http://git-scm.com/] là công cụ quản lý mã nguồn vạn năng. Đây là một công cụ quản lý mã nguồn tin cậy, ổn định, đa dụng và cực kỳ mềm dẻo và chính sự mềm dẻo của Git làm cho việc học nó trở nên khó khăn, tất nhiên là không nói đến những người đã tạo ra nó.

Theo quan sát của Arthur C. Clarke, bất kể công nghệ tiên tiến nào cũng không thể phân biệt rạch ròi là nó có kỳ diệu hay không. Đây cũng là cách hay để đề cập đến Git: những người mới sử dụng không cần quan tâm đến bên trong Git làm việc như thế nào mà hãy xem khả năng thần kỳ của nó như là một điệu gizmo có thể làm những người coi nó là bạn sửng sốt và làm điên đầu những người đối lập.

Thay vì đi sâu vào chi tiết, chúng tôi đưa ra phác thảo cách làm việc của các hiệu ứng chuyên biệt. Sau khi sử dụng lặp lại nhiều lần, từ từ bạn sẽ hiểu từng mẹo một và thực hiện được những việc mà mình muốn làm.

Bản dịch

- Tiếng Trung Giản thể [/~blynn/gitmagic/intl/zh_cn/]: dịch bởi JunJie, Meng và JiangWei. Đã chuyển đổi sang: Tiếng Trung Phồn thể [/~blynn/gitmagic/intl/zh_tw/] thông qua lệnh `cconv -f UTF8-CN -t UTF8-TW`.
- Tiếng Pháp [/~blynn/gitmagic/intl/fr/]: dịch bởi Alexandre Garel; và đồng thời được xuất bản tại itaapy [http://tutoriels.itaapy.com/].
- Tiếng Đức [/~blynn/gitmagic/intl/de/]: dịch bởi Benjamin Bellee và Armin Stebich; và đồng thời xuất bản trên website của Armin [http://gitmagic.lordofbikes.de/].
- Tiếng Ý [/~blynn/gitmagic/intl/it/]: dịch bởi Mattia Rigotti.
- Tiếng Hàn Quốc [/~blynn/gitmagic/intl/ko/]: dịch bởi Jung-Ho (John) Han; đồng thời xuất bản trên trang web của John [https://sites.google.com/site/drinkhanjohn/useful-links/].
- Tiếng Ba Lan [/~blynn/gitmagic/intl/pl/]: dịch bởi Damian Michna.
- Tiếng Bồ Đào Nha Bra-xin [/~blynn/gitmagic/intl/pt_br/]: dịch bởi José Inácio Serafini và Leonardo Siqueira Rodrigues.
- Tiếng Nga [/~blynn/gitmagic/intl/ru/]: dịch bởi Tikhon Tarnavsky, Mikhail Dymskov và một số người khác.
- Tiếng Tây Ban Nha [/~blynn/gitmagic/intl/es/]: dịch bởi Rodrigo Toledo và Ariset Llerena Tapia.
- Ukrainian [/~blynn/gitmagic/intl/uk/]: dịch bởi Volodymyr Bodenchuk.
- Tiếng Việt [/~blynn/gitmagic/intl/vi/]: dịch bởi Trần Ngọc Quân và đồng thời xuất bản bản dịch này trên trang Web cá nhân của mình [http://vnwildman.users.sourceforge.net/gitmagic/].

Các định dạng khác

- Trang web đơn [book.html]: dạng HTML đơn giản, không được định dạng bằng CSS.

- Định dạng PDF [book.pdf]: thuận tiện cho việc in ấn.
- Gói dành cho Debian [http://packages.debian.org/gitmagic], gói dành cho Ubuntu [http://packages.ubuntu.com/gitmagic]: tải về đĩa cứng từ các địa chỉ này. Tiện lợi khi máy chủ này không kết nối mạng hay không hoạt động [http://csdcf.stanford.edu/status/].
- Sách giấy [Amazon.com [http://www.amazon.com/Git-Magic-Ben-Lynn/dp/1451523343/]]: 64 trang, 15.24cm x 22.86cm, đen trắng. Rất tiện sử dụng vì chẳng cần đến điện.

Lời cảm ơn!

Tôi gửi lời cảm ơn đến những người đã dịch quyển sách này. Tôi rất cảm kích vì có được số lượng độc giả rộng lớn có được bởi những người đã được nêu tên ở trên.

Dustin Sallings, Alberto Bertogli, James Cameron, Douglas Livingstone, Michael Budde, Richard Albury, Tarmigan, Derek Mahar, Frode Aannevik, Keith Rarick, Andy Somerville, Ralf Recker, Øyvind A. Holm, Miklos Vajna, Sébastien Hinderer, Thomas Miedema, Joe Malin, Tyler Breisacher, Sonia Hamilton, Julian Haagsma, Romain Lespinasse, Sergey Litvinov, Oliver Ferrigni, David Toca, Сергей Сергеев, Joël Thieffry và Baiju Muthukadan đã đóng góp trong việc sửa chữa và cải tiến nội dung.

François Marier đã bảo trì gói Debian do Daniel Baumann khởi xướng.

Tôi cũng gửi lời cảm ơn tới sự giúp đỡ và sự tán dương của các bạn. Tôi muốn trích dẫn những lời đó ra đây, nhưng làm như thế có vẻ hơi lố bịch, tự cao tự đại.

Nếu tôi có sai sót gì, xin hãy thông tin hay gửi bản vá cho tôi!

Giấy phép sử dụng

Hướng dẫn này được phát hành dựa trên Giấy Ghép Công phiên bản 3 [http://www.gnu.org/licenses/gpl-3.0.html]. Đương nhiên, nội dung của quyển sách được quản lý bằng Git, và bạn có thể dễ dàng có được nó bằng cách gõ:

```
$ git clone git://repo.or.cz/gitmagic.git  # Tạo ra thư mục "gitmagic".
```

hay từ các máy chủ khác:

```
$ git clone git://github.com/blynn/gitmagic.git
$ git clone git://gitorious.org/gitmagic/mainline.git
$ git clone https://code.google.com/p/gitmagic/
$ git clone git://git.assembla.com/gitmagic.git
$ git clone git@bitbucket.org:blynn/gitmagic.git
```

GitHub, Assembla, và Bitbucket có hỗ trợ các kho có tính riêng tư, hai địa sau là miễn phí.

Chương 1. Giới thiệu

Tôi sử dụng cách ví von để giới thiệu về hệ thống quản lý mã nguồn. Xem bài viết về quản lý mã nguồn trên Wikipedia [http://en.wikipedia.org/wiki/Revision_control] để có được sự giải thích thỏa đáng.

Công Việc giống như Trò Chơi

Tôi đã chơi điện tử trên máy tính suốt từ bé đến giờ. Nhưng tôi chỉ bắt đầu sử dụng hệ thống quản lý mã nguồn khi đã trưởng thành. Tôi tin rằng không chỉ có mình tôi như thế, và việc so sánh giữa hai điều đó sẽ làm cho các khái niệm trở nên dễ hiểu, dễ giải thích hơn.

Hãy nghĩ việc biên soạn mã nguồn, tài liệu cũng giống như việc chúng ta đang chơi trò chơi điện tử trên máy tính. Một khi bạn đã làm được kha khá, bạn sẽ muốn ghi lại thành quả công việc của mình. Để làm điều đó, bạn chỉ việc bấm vào nút *Save* trong chương trình biên soạn của mình.

Nhưng việc làm này sẽ ghi đè lên dữ liệu của bản cũ. Điều này cũng giống như các trò chơi đã cũ chỉ cho phép ghi trên một tập tin: bạn phải chắc chắn là mình muốn ghi lại, nếu không thì bạn không bao giờ có thể quay lại trạng thái cũ nữa. Và thật không may vì lần lưu trước đó có thể là đúng tại một điểm rất hay trong lượt chơi và bạn muốn quay lại đó sau này. Tệ hơn nữa là khi bản ghi lại hiện tại lại không đúng, và thế là bạn sẽ phải bắt đầu lại từ đầu.

Quản Lý Mã Nguồn

Khi biên soạn, bạn có thể chọn *Save As...* để ghi lại tập tin hiện tại nhưng với một cái tên khác, hay là sao chép tập tin ra một chỗ khác trước khi bạn ghi lại, nếu như bạn muốn dùng cả các bản cũ. Bạn có thể nén chúng lại để tiết kiệm dung lượng lưu trữ. Đây là dạng thức nguyên thủy và tốn nhiều công sức cho việc quản lý dữ liệu. Trò chơi trên máy tính đã cải tiến cách trên từ rất lâu rồi, rất nhiều trong số chúng cung cấp cho bạn khả năng tự động ghi lại sau từng khoảng thời gian nhất định.

Chúng ta sẽ giải quyết một vấn đề hơi hóc búa một chút nhé! Bạn nói rằng bạn có nhiều tập tin có liên quan mật thiết với nhau, như mã nguồn cho một dự án chẳng hạn, hay các tập tin cho một website. Bây giờ nếu bạn muốn giữ một phiên bản cũ bạn phải lưu giữ toàn bộ thư mục. Giữ nhiều phiên bản như thế bằng cách thủ công thật bất tiện, và sẽ nhanh chóng trở nên tốn kém.

Đối với một số trò chơi, ghi lại một trò chơi thực tế là bao gồm toàn bộ thư mục. Những trò chơi này thực thi việc này tự động và chỉ đưa ra một giao diện thích hợp cho người chơi để quản lý các phiên bản của thư mục này.

Các hệ thống quản lý mã nguồn cũng hoạt động theo cách ấy. Chúng có một giao diện tinh tế để quản lý một nhóm các thứ trong thư mục. Bạn có thể ghi lại trạng thái của thư mục một cách thường xuyên, và bạn có thể tải lên bất kỳ một trạng thái nào đã được ghi lại trước đó. Không giống như các trò chơi trên máy tính, chúng thường khôn khéo hơn về việc tiết kiệm không gian lưu trữ. Thông thường, chỉ có một số ít tài liệu được sửa đổi giữa các phiên bản, và cũng không nhiều. Nó chỉ lưu giữ những cái có thay đổi thay vì toàn bộ tất cả.

Hệ Thống Phân Tán

Bây giờ hãy tưởng tượng có một trò chơi rất khó. Khó để hoàn thành đến mức là có nhiều game thủ lão luyện trên toàn thế giới quyết định lập thành đội và chia sẻ những trò chơi mà họ đã lưu lại với mục đích là để tất cả mọi người có thể theo dõi được nhau. **Speedruns** là những ví dụ trong đời sống thực: các đấu thủ được phân hóa theo các mức của cùng một trò chơi hợp tác với nhau để đạt được các kết quả đáng kinh ngạc.

Làm thế nào bạn có thể cài đặt một hệ thống mà chúng có thể lấy được từng bản ghi của mỗi người một cách dễ dàng? Và tải lên cái mới hơn?

Ngày xưa, mọi dự án đều sử dụng hệ thống quản lý tập trung. Máy chủ ở một chỗ đâu đó và giữ tất cả các trò chơi đã được ghi lại. Không còn ai khác làm điều đó nữa. Mọi người giữ phần lớn các trò chơi được ghi lại trong máy của họ. Khi một đấu thủ muốn chơi, họ có thể tải về bản ghi lại cuối cùng đã lưu lại ở máy chủ, chơi một lúc, ghi lại và tải trở lại máy chủ để những người khác có thể sử dụng.

Điều gì xảy ra khi một người chơi, vì một lý do nào đó, muốn có được lần chơi cũ hơn? Lý do có thể là lần chơi hiện tại không ổn định hay có sai sót bởi vì một người chơi nào đó quên không chỉnh lại trò chơi về mức 3, và họ muốn tìm lần chơi đã ghi lại cuối cùng mà họ vẫn chưa hoàn thành. Hay có thể là họ muốn so sánh sự khác nhau giữa các lần chơi để thấy được thành quả của từng người chơi.

Có rất nhiều lý do vì sao cần đến bản cũ hơn, nhưng kết cục là giống nhau. Họ phải hỏi máy chủ trung tâm để lấy về trò chơi cũ đã được lưu lại. Càng ghi lại nhiều trò chơi, họ càng cần phải liên lạc nhiều với nhau.

Những hệ thống quản lý mã nguồn thế hệ mới, Git là một trong số đó, được biết đến như một hệ thống phân tán, và có thể coi nó là một hệ thống tập trung có mở rộng. Khi người chơi tải về từ máy chủ chính, họ lấy toàn bộ tất cả các lần đã ghi lại, không chỉ mỗi bản cuối cùng. Điều đó có nghĩa là họ trở thành bản sao của máy chủ trung tâm.

Việc khởi tạo bản sao như thế có vẻ hơi xa hoa, đặc biệt là nếu nó có lịch sử phát triển lâu dài, nhưng cái giá phải trả cũng chỉ là việc cần nhiều thời gian để lấy về. Một lợi ích trực tiếp của việc này là khi các tài liệu cũ cần đến, việc liên lạc với máy chủ trung tâm là không cần thiết nữa.

Quan Niệm Cổ Hủ

Một quan niệm phổ biến là hệ thống phân tán không thích hợp với các dự án có yêu cầu một kho chứa trung tâm chính thức. Không điều gì có thể chà đạp lên sự thật. Chụp ảnh ai đó không có nghĩa là lấy đi linh hồn họ. Cũng như thế, nhân bản kho chính cũng không làm giảm đi sự quan trọng của nó.

Tóm lại, một hệ thống phân tán đã thiết kế tốt thì làm bất cứ công việc nào cũng khá hơn một hệ thống quản lý mã nguồn tập trung. Tài nguyên mạng thường thì tốn kém hơn các tài nguyên nội bộ. Chúng ta sẽ nói đến các hạn chế của hệ thống phân tán sau, sự so sánh như sau thường đúng: hệ thống phân tán thường tốt hơn.

Một dự án nhỏ có thể chỉ cần dùng một phần nhỏ các đặc tính được đưa ra bởi một hệ thống như thế, nhưng việc sử dụng một hệ thống không có khả năng mở rộng

cho một dự án nhỏ thì cũng giống như việc sử dụng hệ thống số La Mã để tính toán các số nhỏ.

Hơn thế nữa, dự án của bạn có thể lớn vượt ra ngoài dự kiến ban đầu. Việc sử dụng Git từ lúc khởi sự thì cũng giống như việc sử dụng một bộ dao vạn năng chỉ để phục vụ cho mỗi việc mở nút chai. Đến một ngày nào đó bạn cần đến một cái chìa vít bạn sẽ vui sướng vì mình không chỉ có mỗi cái mở nút chai.

Xung Đột Khi Trộn

Với chủ đề này, dùng cách ví von nó với một trò chơi trên máy tính là hơi khó. Thay vì thế, để chúng tôi dùng việc biên soạn một tài liệu để giải thích cho bạn.

Giả sử Alice chèn thêm một dòng vào đầu một tập tin, và Bob nối một dòng vào cuối của bản sao của mình. Cả hai đều tải lên các thay đổi của mình. Phần lớn các hệ thống sẽ tự động tìm ra hành động hợp lý: chấp nhận và trộn các sự thay đổi của họ, do đó cả hai sự thay đổi mà Alice và Bob tạo ra đều được dùng.

Bây giờ giả sử cả Alice và Bob cùng sửa một dòng. Thế thì mâu thuẫn này không thể sử lý được mà không có sự can thiệp của con người. Người thứ hai tải lên sẽ được thông báo có xung đột xảy ra, *merge conflict*, và phải chọn một là sửa thêm nữa, hay sửa lại toàn bộ dòng đó.

Nhiều tình huống phức tạp có thể nảy sinh. Hệ thống quản lý mã nguồn giữ phần dễ dàng cho chúng, và để lại những tình huống khó khăn cho chúng ta. Nhưng thông thường cách ứng xử của chúng có thể điều chỉnh được.

Chương 2. Các Thủ Thuật Cơ Bản

Thay vì lao vào cả một biển lệnh với Git, bạn hãy sử dụng các ví dụ cơ bản để bắt đầu. Mặc dù chúng rất đơn giản, nhưng tất cả chúng đều rất hữu dụng. Quả thực là vậy, trong tháng đầu tiên sử dụng Git, tôi chưa bao giờ vượt quá những gì nói trong chương này.

Ghi lại Trạng thái

Bạn muốn thử thực hiện một số lệnh gì đó với Git? Trước khi làm điều đó, thực hiện các lệnh sau trong thư mục hiện hành chứa các mã nguồn hay văn bản mà bạn muốn quản lý:

```
$ git init
$ git add .
$ git commit -m "Bản sao lưu đầu tiên"
```

Bây giờ nếu như các sửa đổi của bạn vừa làm không được như mong đợi, hãy phục hồi lại bản cũ:

```
$ git reset --hard # Đặt lại trạng thái và dữ liệu như lần chuyển giao cuối
```

Sau đó sửa nội dung cho đúng ý mình rồi ghi lại thành một trạng thái mới:

```
$ git commit -a -m "Bản sao lưu khác"
```

Thêm, Xóa, Đổi Tên

Lệnh ở trên chỉ giữ dấu vết các tập tin hiện diện tại thời điểm bạn chạy lệnh **git add**. Nếu bạn thêm các tập tin hay thư mục, thì bạn sẽ phải thông báo với Git:

```
$ git add readme.txt Documentation
```

Tương tự như vậy, nếu bạn muốn Git bỏ đi các tập tin nào đó:

```
$ git rm kludge.h obsolete.c
$ git rm -r incriminating/evidence/ #gỡ bỏ một cách đệ qui
```

Git xóa bỏ những tập tin nếu như bạn chưa làm.

Đổi tên tập tin thì cũng giống như là việc bạn gỡ bỏ tên cũ và đặt vào nó cái tên mới. Lệnh **git mv** có cú pháp rất giống lệnh **mv** của hệ thống Linux. Ví dụ:

```
$ git mv bug.c feature.c
```

Chức Năng Undo/Redo

Đôi khi bạn chỉ muốn quay trở lại và bỏ đi những thay đổi trong quá khứ tại một thời điểm nào đó bởi vì chúng tất cả đã sai. Thì lệnh:

```
$ git log
```

sẽ hiển thị cho bạn danh sách các lần chuyển giao gần đây cùng với giá trị băm SHA1:

```
commit 766f9881690d240ba334153047649b8b8f11c664
Author: Bob <bob@example.com>
Date:   Tue Mar 14 01:59:26 2000 -0800

    Replace printf() with write().

commit 82f5ea346a2e651544956a8653c0f58dc151275c
Author: Alice <alice@example.com>
Date:   Thu Jan 1 00:00:00 1970 +0000

    Initial commit.
```

Chỉ vài ký tự của giá trị băm là đủ để chỉ ra một chuyển giao cụ thể; một cách khác là chép và dán giá trị băm. Gõ:

```
$ git reset --hard 766f
```

để phục hồi lại trạng thái đã được chỉ ra và xóa bỏ tất cả các lần chuyển giao mới hơn kể từ đó.

Một lúc nào đó bạn lại muốn nhảy tới một bản cũ hơn. Trong trường hợp này thì gõ:

```
$ git checkout 82f5
```

Nó giúp bạn quay lại đúng thời điểm đó, trong khi vẫn giữ lại những lần chuyển giao mới hơn. Tuy nhiên, giống như cỗ máy thời gian trong các bộ phim khoa học viễn tưởng, nếu bây giờ bạn sửa sau đó chuyển giao, bạn sẽ ở trong một thực tại khác, bởi vì hành động của bạn bây giờ đã khác với khi chúng ta lần đầu tiên ở tại đây.

Có cách thực tế hơn là sử dụng *branch*, và chúng ta có nhiều điều để nói về nó sau này. Bây giờ, chỉ cần nhớ là:

```
$ git checkout master
```

sẽ mang chúng ta trở về hiện tại. Ngoài ra, để tránh rủi ro khi sử dụng Git, thì luôn luôn chuyển giao hay reset các thay đổi của bạn trước khi chạy lệnh checkout.

Sự tương đồng với **ván đấu** trên máy tính:

- `git reset --hard`: lấy cái cũ đã được lưu lại và xóa tất cả các **ván đấu** mới hơn cái vừa lấy.
- `git checkout`: lấy một cái cũ, nhưng chỉ chơi với nó, trạng thái của **ván đấu** sẽ tách riêng về phía mới hơn chỗ mà bạn đã ghi lại lần đầu tiên. Bất kỳ **ván đấu** nào bạn tạo từ bây giờ sẽ là bản cuối cùng trong nhánh riêng rẽ tương ứng với một thực tại khác mà bạn đã gia nhập vào. chúng tôi sẽ nói sau.

Bạn có thể chọn chỉ phục hồi lại các tập tin hay thư mục bạn muốn bằng cách thêm vào chúng vào phần sau của câu lệnh:

```
$ git checkout 82f5 some.file another.file
```

Bạn phải cẩn thận khi sử dụng các lệnh, như là lệnh **checkout** có thể âm thầm ghi đè lên các tập tin. Để ngăn ngừa rủi ro như thế, hãy chuyển giao trước khi chạy lệnh checkout, nhất là khi mới học sử dụng Git. Tóm lại, bất kỳ khi nào bạn không chắc chắn về một lệnh nào đó, dù có là lệnh của Git hay không, đầu tiên hãy chạy lệnh **git commit -a**.

Bạn không thích việc cắt dán ư? Hãy sử dụng:

```
$ git checkout :/"My first b"
```

để nhảy tới lần chuyển giao mà phần chú thích của nó bắt đầu với chuỗi bạn đã cho. Bạn cũng có thể yêu cầu trạng thái thứ 5 kể từ cái cuối cùng:

```
$ git checkout master~5
```

Sự quay lại

Trong một phiên tòa, mỗi sự kiện được gắn với một bản ghi. Cũng giống thế, bạn có thể chọn lệnh chuyển giao để undo.

```
$ git commit -a
$ git revert 1b6d
```

sẽ chỉ undo lần chuyển giao với giá trị băm đã chỉ ra. Sự quay trở lại được ghi nhận như là một lần chuyển giao mới, bạn có thể xác nhận lại điều này bằng lệnh **git log**.

Tạo Nhật Ký các thay đổi

Một số dự án yêu cầu phải có một changelog [http://en.wikipedia.org/wiki/Changelog]. Bạn có thể tạo một cái bằng cách gõ:

```
$ git log > Changelog # ThayĐổi
```

Tải về các tập tin

Lấy về một bản sao của một dự án quản lý bằng Git bằng cách gõ:

```
$ git clone git://server/path/to/files
```

Ví dụ, để lấy tất cả các tập tin mà tôi đã dùng để tạo ra cho quyển sách này là:

```
$ git clone git://git.or.cz/gitmagic.git
```

Chúng ta sẽ có nhiều điều để nói về lệnh **clone** sớm thôi.

Thử Nghiệm

Nếu bạn đã tải về một bản sao của một dự án bằng lệnh **git clone**, bạn có thể lấy về phiên bản cuối cùng với lệnh:

```
$ git pull
```

Xuất Bản

Giả sử bạn đã tạo được một kho Git và bạn muốn chia sẻ nó với người khác. Bạn có thể bảo họ tải về từ máy tính của mình, nhưng nếu họ làm như thế trong khi bạn đang cải tiến nó hay có những thay đổi mang tính thử nghiệm, họ có thể gặp trục trặc. Dĩ nhiên, đây là lý do tại sao mà chu kỳ phát hành phần mềm lại tồn tại phải

không nào. Những người phát triển có thể làm việc thường xuyên trên một dự án, nhưng họ chỉ xuất bản những đoạn mã mà họ cảm thấy nó có thể dùng được để tránh ảnh hưởng đến người khác.

Thực hiện điều này với Git, trong thư mục làm việc của Git:

```
$ git init
$ git add .
$ git commit -m "Bản phát hành đầu tiên"
```

Sau đó nói với những người cùng sử dụng hãy chạy:

```
$ git clone máy.tính.của.bạn:/đường/dẫn/tới/script
```

để tải dữ liệu về. Giả định là họ truy cập thông qua ssh. Nếu không, chạy **git daemon** và nói với người sử dụng là chạy lệnh sau để thay thế:

```
$ git clone git://máy.tính.của.bạn:/đường/dẫn/tới/script
```

Kể từ lúc này, bất cứ khi nào mã nguồn của bạn đã có thể sử dụng được, chỉ việc thực hiện:

```
$ git commit -a -m "Bản phát hành tiếp"
```

và những người sử dụng có thể cập nhật dữ liệu của họ bằng cách chuyển tới thư mục làm việc tương ứng và gõ:

```
$ git pull
```

Những người sử dụng sẽ không bao giờ thấy được dữ liệu cuối cùng của bạn mà bạn không muốn họ thấy.

Tôi Đã Làm Được Gì?

Tìm tất cả các thay đổi kể từ lần bạn chuyển giao lần cuối bằng lệnh:

```
$ git diff
```

Hay từ hôm qua:

```
$ git diff "@{yesterday}"
```

Hay giữa một bản nào đó và bản trước đây 2 bản:

```
$ git diff 1b6d "master~2"
```

Trong từng trường hợp, đầu ra là một miếng vá mà nó có thể được sử dụng với lệnh **git apply**. Cũng có thể dùng lệnh:

```
$ git whatchanged --since="2 weeks ago"
```

Thường thường, tôi duyệt lịch sử bằng qgit [http://sourceforge.net/projects/qgit] để thay thế cách ở trên, bởi vì nó có giao diện đồ họa bóng bẩy, hay tig [http://jonas.nitro.dk/tig/], có giao diện dòng lệnh làm việc rất tốt với các máy có kết nối mạng chậm. Một lựa chọn khác là cài đặt máy chủ web, chạy lệnh **git instaweb** và sử dụng bất kỳ trình duyệt web nào.

Bài Tập

Giả sử A, B, C, D là 4 lần chuyển giao thành công, với B giống A ngoại trừ một số tập tin bị xóa bỏ. Chúng ta muốn thêm các tập tin đó trở lại D. Chúng ta thực hiện điều này bằng cách nào?

Ở đây chúng ta có ít nhất 3 giải pháp. Giả thiết chúng ta đang ở D:

1. Sự khác nhau giữa A và B là việc các tập tin đã bị gỡ bỏ. Chúng ta có thể tạo miếng vá tương ứng với sự khác biệt này và áp dụng miếng vá đó:

   ```
   $ git diff B A | git apply
   ```

2. Kể từ sau khi chúng ta ghi lại các tập tin tại A trở đi, chúng ta có thể lấy lại:

   ```
   $ git checkout A foo.c bar.h
   ```

3. Chúng ta có thể xem sự di chuyển từ A tới B giống như là một thay đổi mà chúng ta muốn undo:

   ```
   $ git revert B
   ```

Lựa chọn nào là tốt nhất? Cách nào bạn thích nhất. Thật dễ dàng để có được thứ mà bạn muốn với Git, và thường là có nhiều hơn một cách để thực hiện được một thứ bạn muốn.

Chương 3. Nhân Bản

Trong các hệ thống quản lý mã nguồn trước đây, checkout là tác vụ cơ bản để lấy các tập tin về. Bạn lấy về toàn bộ các tập tin được lưu giữ trong từng phiên bản riêng biệt.

Với Git và các hệ thống quản lý mã nguồn phân tán, clone là tác vụ cơ bản. Để lấy các tập tin, bạn lấy về toàn bộ kho chứa. Nói cách khác, bạn thực tế là một bản sao của máy chủ trung tâm. Bất kỳ cái gì bạn thể làm được với kho chứa chính thì cũng làm được ở đây.

Đồng bộ hóa Các Máy tính

Tôi có thể tạo gói **tarball** hay sử dụng lệnh **rsync** để sao lưu dự phòng và đồng bộ hóa dữ liệu. Nhưng thỉnh thoảng, tôi biên tập trên máy tính xách tay của mình, nhưng lúc khác lại trên máy tính để bàn, và chúng có thể không có sự trao đổi được với nhau.

Khởi tạo kho chứa Git và commit các tập tin trên một máy tính. Sau đó trên máy tính kia chạy lệnh:

```
$ git clone other.computer:/path/to/files
```

để tạo một bản sao thứ hai cho các tập tin và kho chứa. Từ giờ trở đi,

```
$ git commit -a
$ git pull other.computer:/path/to/files HEAD
```

sẽ lấy về một trạng thái của các tập tin trên máy tính khác về máy bạn đang làm việc. Nếu bạn vừa tạo ra một sự chỉnh sửa xung đột trong cùng một tập tin, Git sẽ cho bạn biết và bạn có thể chuyển giao sau khi đã sửa chữa chúng.

Quản lý theo cách Cũ

Khởi tạo kho Git cho các tập tin của bạn:

```
$ git init
$ git add .
$ git commit -m "Lần commit khởi tạo"
```

Trên máy chủ trung tâm, khởi tạo kho *thuần ở* một thư mục nào đó:

```
$ mkdir proj.git
$ cd proj.git
$ git --bare init
$ touch proj.git/git-daemon-export-ok
```

Khởi động dịch vụ Git nếu cần:

```
$ git daemon --detach  # nó có thể đã hoạt động rồi
```

Để làm một máy chủ chạy dịch vụ Git, làm theo các chỉ dẫn sau để cài đặt và khởi tạo kho Git. Cách thường thấy nhất là điền vào mẫu có sẵn trên trang web.

Push dự án của bạn lên máy chủ trung tâm bằng lệnh:

```
$ git push central.server/path/to/proj.git HEAD
```

Để lấy về mã nguồn, các nhà phát triển phần mềm chỉ cần gõ:

```
$ git clone central.server/path/to/proj.git
```

Sau khi thay đổi, các nhà phát triển phần mềm sẽ lưu lại các thay đổi trên máy tính của mình:

```
$ git commit -a
```

Để cập nhật lên bản mới nhất:

```
$ git pull
```

Mọi xung đột phải được xử lý trước, sau đó mới chuyển giao:

```
$ git commit -a
```

Gửi thay đổi của mình lên máy chủ trung tâm:

```
$ git push
```

Nếu máy chủ trung tâm có thay đổi bởi hành động của một người phát triển phần mềm khác, quá trình push sẽ bị lỗi, và anh ta phải pull về bản mới nhất, xử lý các xung đột khi trộn, sau đó thử lại.

Người dùng phải có quyền truy cập SSH mới có thể thực hiện được lệnh pull và push ở trên. Tuy nhiên, ai cũng có thể lấy mã nguồn về bằng lệnh:

```
$ git clone git://central.server/path/to/proj.git
```

Giao thức git nguyên bản thì cũng giống như là HTTP: ở đây không cần xác thực, do vậy ai cũng có thể lấy về dự án. Do vậy, theo mặc định, việc push thông qua giao thức git là không được phép.

Mã nguồn riêng tư

Với một dự án nguồn-đóng, bỏ quên lệnh touch, và chắc chắn là chưa từng tạo ra file nào có tên `git-daemon-export-ok`. Kho chứa từ giờ trở đi không thể lấy về thông qua giao thức git; chỉ những người có khả năng truy cập bằng SSH mới có thể thấy nó. Nếu tất cả các kho chứa đều đóng, việc chạy git daemon là không cần thiết nữa bởi vì tất cả việc truyền thông bây giờ đều thông qua SSH.

Kho thuần

Kho thuần (bare) được đặt tên như vậy vì nó không chứa thư mục làm việc; nó *thuần túy* chứa các tập tin thường là ẩn trong thư mục con `.git`. Hay nói cách khác, nó chứa lịch sử mã nguồn của một dự án, và không bao giờ giữ các tập tin bất kỳ phiên bản nào.

Kho thuần có vai trò hoạt động giống như máy chủ trung tâm trong các hệ thống quản lý mã nguồn tập trung: thư mục chủ dự án của bạn. Các nhà phát triển phần mềm nhân bản dữ liệu dự án của bạn ở đây, và *push* các thay đổi chính thức lên đó. Thông thường nó đặt tại máy chủ mà máy này đôi khi còn làm một số việc khác nữa.

Sự biên soạn mã nguồn xảy ra trên bản sao, vì vậy thư mục chủ của kho chứa có thể hoạt động mà không cần thư mục làm việc.

Nhiều lệnh Git gặp lỗi trên kho thuần trừ phi biến môi trường GIT_DIR được đặt với giá trị là đường dẫn đến kho chứa, hay tùy chọn --bare được áp dụng.

Push ngược với pull

Tại sao chúng tôi lại giới thiệu lệnh *push*, thay vì trông cậy vào lệnh *pull* quen thuộc? Trước hết, việc pull gặp lỗi trên kho thuần: thay vào đó bạn phải dùng lệnh *fetch*, lệnh này chúng ta sẽ nói sau. Nhưng dù là chúng ta giữ kho chứa thông thường trên máy chủ trung tâm, việc *pull* lên nó hơi cồng kềnh. Chúng ta phải đăng nhập vào máy chủ trước, và cung cấp cho lệnh *pull* địa chỉ mạng của máy chúng ta đang *pull* từ đó. Chương trình tường lửa có thể gây trở ngại, và điều gì xảy ra khi chúng ta không có khả năng truy cập *hệ vỏ* máy chủ trong chỗ đầu tiên đó?

Tuy nhiên, ngoài trường hợp này ra, chúng ta còn nản lòng với việc push lên kho chứa, bởi vì tình trạng hỗn loạn có thể xảy khi thư mục đích có chứa thư mục làm việc.

Tóm lại, khi bạn học Git, chỉ *push* khi đích là kho thuần; nếu không thì dùng *pull*.

Rẽ nhánh một dự án

Bạn chán ngấy cách mà dự án mà bạn đang làm việc chạy? Bạn nghĩ mình có thể làm tốt hơn thế? Thế thì trên máy chủ của mình:

```
$ git clone git://main.server/path/to/files
```

Tiếp theo, nói với mọi người về việc nhánh rẽ từ dự án tại máy chủ của bạn.

Từ bây giờ trở đi, bạn có thể trộn các sự thay đổi từ dự án gốc với lệnh:

```
$ git pull
```

Sao lưu không giới hạn

Bạn muốn lưu trữ dự án của mình ở nhiều nơi khác nhau ư? Nếu dự án của bạn có nhiều người cùng phát triển, bạn không cần phải làm gì cả! Mỗi một bản sao đều đồng thời có tác dụng như một bản sao lưu dự phòng. Mỗi bản sao không chỉ lưu trạng thái hiện hành mà toàn bộ lịch sử trong dự án. Nhờ có giá trị băm bằng mật mã, nếu bản sao của người nào đó bị hỏng, nó sẽ được phát hiện ngay sau khi họ liên lạc với những người khác.

Nếu dự án của bạn không phổ biến, hãy tìm máy chủ lưu giữ bản sao của mình càng nhiều càng tốt.

Người mắc bệnh hoang tưởng sẽ luôn luôn ghi ra 20-byte giá trị băm SHA1 cuối cùng của HEAD ở đâu đó an toàn. Nó phải an toàn, không riêng tư. Ví dụ, xuất bản nó lên báo giấy cũng tốt, bởi vì rất khó để thay đổi tất cả các bản sao của nó.

Làm nhiều việc cùng lúc

Bạn muốn làm nhiều việc cùng một lúc trên một dự án. Thế thì hãy commit dự án của bạn và chạy:

```
$ git clone . /some/new/directory
```

Nhờ có liên kết cứng [http://en.wikipedia.org/wiki/Hard_link], việc nhân bản nội bộ yêu cầu ít không gian và thời gian hơn so với việc sao lưu thông thường.

Bây giờ bạn có thể làm hai công việc độc lập nhau cùng một lúc. Ví dụ như, bạn có thể biên soạn trên bản sao này trong khi bản sao kia đang được biên dịch. Tại bất kỳ thời điểm nào, bạn cũng có thể commit và pull các thay đổi từ một bản sao khác:

```
$ git pull /the/other/clone HEAD
```

Song hành cùng các hệ thống SCM khác

Bạn đang làm việc cho một dự án có sử dụng hệ thống quản lý mã nguồn cũ, và bạn lại muốn sử dụng Git? Thế thì hãy khởi tạo kho chứa Git trong thư mục bạn đang làm việc:

```
$ git init
$ git add .
$ git commit -m "Lần commit khởi tạo"
```

sau đó nhân bản nó:

```
$ git clone . /some/new/directory
```

Bây giờ hãy chuyển đến thư mục mới đó và làm việc ở đây thay vì chỗ cũ, sử dụng Git để thỏa mãn tình yêu của mình. Đôi khi, bạn sẽ muốn đồng bộ hóa với những người khác nữa, trong trường hợp đó hãy chuyển tới thư mục nguyên gốc, đồng bộ hóa bằng cách sử dụng một hệ thống quản lý mã nguồn khác, và gõ:

```
$ git add .
$ git commit -m "Đồng bộ hóa với những người khác"
```

Sau đó chuyển tới thư mục mới và chạy:

```
$ git commit -a -m "Mô tả về các thay đổi của mình"
$ git pull
```

Thủ tục đem lại các thay đổi của bạn tới những người khác nữa trên hệ thống quản lý mã nguồn khác. Thư mục mới có chứa các tập tin mà bạn thay đổi. Chạy các lệnh mà hệ thống quản lý mã nguồn khác cần để tải chúng lên kho chứa trung tâm.

Subversion, có lẽ là hệ thống quản lý mã nguồn tập trung tốt nhất, được sử dụng bởi vô số các dự án. Lệnh **git svn** sẽ tự động hóa những việc đã nói ở trên dành cho Subversion, bạn cũng có thể làm như thế để xuất dự án Git thành Subversion [http://google-opensource.blogspot.com/2008/05/export-git-project-to-google-code.html].

Mercurial

Mercurial là hệ thống tương tự như hệ thống quản lý mã nguồn có thể làm việc gần giống như Git. Với plugin hg-git, người sử dụng Mercurial có thể push và pull từ kho Git mà không mất mát gì.

Lấy plugin hg-git dành cho Git bằng cách:

```
$ git clone git://github.com/schacon/hg-git.git
```

hay là sử dụng Mercurial:

```
$ hg clone http://bitbucket.org/durin42/hg-git/
```

Thật buồn là tôi không biết rằng có một plugin tương tự dành cho Git. Vì vậy, tôi ủng hộ Git hơn Mercurial khi dùng cho kho chứa chính, dù là bạn thích Mercurial hơn. Với một dự án Mercurial, thường thường một người tình nguyện duy trì kho Git song song cho tương thích với người sử dụng Git, cũng phải cảm ơn plugin hg-git, một dự án Git tự động tiếp nhận người sử dụng Mercurial.

Mặc dù plugin có thể chuyển đổi Mercurial sang Git bằng cách push tới một kho rỗng, công việc này là dễ dàng với script hg-fast-export.sh, đã sẵn dùng từ:

```
$ git clone git://repo.or.cz/fast-export.git
```

Để chuyển đổi, trong một thư mục rỗng hãy chạy:

```
$ git init
$ hg-fast-export.sh -r /hg/repo
```

sau khi đặt script vào trong biến môi trường $PATH.

Bazaar

Chúng tôi đề cập vắn tắt về Bazaar bởi vì nó là hệ thống quản lý mã nguồn phân tán miễn phí và phổ biến nhất chỉ sau Git và Mercurial.

Bazaar có lợi thế vì phát triển sau, có tuổi tương đối trẻ; những người thiết kế ra nó có thể học hỏi được nhiều từ các sai lầm trong quá khứ, và tránh được vết xe đổ. Ngoài ra, các nhà phát triển còn lưu tâm đến khả năng chuyển đổi và tương tác với các hệ thống quản lý mã nguồn khác.

Plugin bzr-git giúp người dùng Bazaar làm việc với kho Git trong chừng mực nào đó. Chương trình chuyển đổi Bazaar thành Git, và có thể làm hơn thế nữa, trong khi bzr-fast-export thích hợp nhất cho việc chuyển đổi một lần duy nhất.

Tại sao Tôi dùng Git?

Trước tiên, tôi chọn Git bởi tôi nghe nói nó làm được việc phi thường là có thể quản lý mã nguồn cho một thứ khó quản lý như hạt nhân của hệ điều hành **Linux**. Tôi chưa bao giờ nghĩ đến việc chuyển sang dùng một phần mềm quản lý mã nguồn khác. Git làm được những việc thật đáng ngưỡng mộ, và tôi chưa từng bao giờ gặp các vấn đề với sai sót của nó. Do tôi hoạt động chủ yếu trên Linux, phát hành trên các nền tảng khác không phải là điều mà tôi quan tâm.

Ngoài ra, tôi thích lập trình bằng ngôn ngữ C và bash scripts để thực thi như là ngôn ngữ kịch bản Python: ở đây có rất ít sự phụ thuộc, và tôi đam mê với những hệ thống thi hành nhanh chóng.

Tôi đã nghĩ về việc làm thế nào để Git có thể phát triển, xa hơn nữa là tự mình viết một công cụ tương tự như Git, nhưng đây không phải là bài tập có tính thực tế. Khi

tôi hoàn thành dự án của mình, dù sao đi nữa tôi vẫn sẽ dùng Git, với lợi thế là có thể chuyển đổi từ hệ thống cũ sang một cách nhanh chóng.

Theo lẽ tự nhiên, những thứ cần thiết cho bạn và những thứ bạn mong muốn có lẽ khác nhau, và bạn có thể tốt hơn nếu mình làm việc với hệ thống khác. Dù sao đi nữa, bạn sẽ không bao giờ phải hối tiếc vì đã chọn Git.

Chương 4. Thủ Thuật Tạo Nhánh

Tạo Nhánh và Trộn là các đặc tính sát thủ của Git.

Vấn đề đặt ra: Những nhân tố bên ngoài chắc hẳn đòi hỏi cần hoán chuyển văn cảnh. Một lỗi tồi tệ xuất hiện trong phiên bản đã được phát hành mà không được cảnh báo gì. Điều tồi tệ nhất có thể xảy ra là phải xóa bỏ hẳn đặc tính kỹ thuật đó. Người phát triển phần mềm, người mà đã giúp bạn viết nó, cần biết lý do về việc bãi bỏ. Trong tất cả các trường hợp trên, bạn buộc phải xóa bỏ cái mà bạn đang làm và làm một cái hoàn toàn mới.

Việc gán đoạn suy nghĩ có thể làm giảm hiệu suất làm việc, và việc hoán chuyển nội dung càng cồng kềnh, vướng víu càng gây hậu quả nặng nề. Đối với các hệ thống quản lý mã nguồn tập trung chúng ta phải tải về một bản sao các tập tin mới từ máy chủ trung tâm. Các hệ thống phân tán hoạt động hiệu quả hơn, như là chúng ta có thể nhân bản một cách cục bộ.

Nhưng việc nhân bản bắt buộc phải sao chép toàn bộ thư mục làm việc cũng như là toàn bộ các mục trong lịch sử cho đến thời điểm đã được chỉ ra. Dù là Git giảm bớt sự lãng phí cho việc này bằng cách chia sẻ và tạo ra các liên kết tập tin cứng, chính bản thân các tập tin dự án cũng phải được tạo ra trong các đề mục của chúng trong thư mục làm việc.

Giải pháp: Git có một công cụ tốt hơn để sử lý tình huống này, nó nhanh và tiết kiệm không gian lưu trữ hơn lệnh nhân bản đó chính là: **git branch**.

Với vài câu thần chú, các tập tin trong thư mục của bạn dễ dàng biến đổi từ phiên bản này sang phiên bản khác. Sự chuyển đổi này có thể làm nhiều hơn việc di chuyển trong trong lịch sử một các đơn thuần. Các tập tin của bạn có thể chuyển hình thái từ bản phát hành cuối thành phiên bản thử nghiệm, thành phiên bản phát triển hiện nay, thành phiên bản của người bạn của bạn, và cứ như thế.

Nút Điều Khiển

Mỗi khi chơi điện tử, bạn bấm vào nút (“nút điều khiển”), màn hình có lẽ hiển thị ngay ra một cái bảng hay một thứ gì đó? Thế thì nhỡ ông chủ của bạn đang đi lại trong văn phòng nơi bạn đang chơi điện tử thì làm cách nào để nhanh chóng giấu chúng đi?

Ở thư mục nào đó:

```
$ echo "Tôi thông minh hơn xếp của mình" > myfile.txt
$ git init
$ git add .
$ git commit -m "Lần commit đầu tiên"
```

Chúng ta đã tạo ra kho chứa Git mà nó theo dõi một tập tin văn bản có chứa một thông điệp đã biết trước. Giờ hãy gõ:

```
$ git checkout -b boss  # dường như chẳng có gì thay đổi sau lệnh này
$ echo "Xếp thông minh hơn tôi" > myfile.txt
$ git commit -a -m "Lần chuyển giao khác"
```

Điều này cũng giống như việc chúng ta ghi đè lên tập tin của mình sau đó commit nó. Nhưng không, đó chỉ là ảo tưởng. Gõ:

```
$ git checkout master  # quay trở lại phiên bản nguyên gốc của tập tin
```

Ối trời ơi! Tập tin văn bản lại trở về như cũ mất rồi. Và nếu ông chủ có ý định ngó qua thư mục của bạn thì hãy gõ:

```
$ git checkout boss  # chuyển trở lại phiên bản vừa mắt ông chủ
```

Bạn có thể hoán chuyển giữa hai phiên bản của tập tin tùy thích, và commit từng cái trong số chúng một cách độc lập.

Bản Nháp

Bạn nói rằng mình đang làm việc với một số đặc tính kỹ thuật, và vì lý do nào đó, bạn muốn quay trở lại bản cách đây ba bản và tạm thời đặt thêm vài dòng lệnh in ra màn hình để có thể thấy được một số hàm hoạt động như thế nào. Thế thì:

```
$ git commit -a
$ git checkout HEAD~3
```

Giờ thì bạn có thể thêm những dòng mã lệnh tạm thời ở đâu mình muốn. Bạn còn có thể commit những thay đổi đó. Khi bạn đã làm xong, hãy thực hiện lệnh

```
$ git checkout master
```

để quay lại công việc chính. Chú ý là bất kỳ các thay đổi không được commit sẽ đổ xuống sông xuống biển.

Nhưng bạn lại muốn ghi lại các thay đổi tạm thời đó sau khi đã làm xong? Rất dễ:

```
$ git checkout -b dirty
```

và commit trước khi quay trở lại nhánh master. Khi nào đó bạn muốn quay trở lại các thay đổi ở trên, đơn giản, chỉ cần gõ:

```
$ git checkout dirty
```

Chúng ta đã đụng chạm đến lệnh như trên ở những chương trước rồi, khi thảo luận về việc tải về một trạng thái cũ. Cuối cùng chúng ta có thể thuật lại toàn bộ câu chuyện: các tập tin đã thay đổi theo trạng thái đã được yêu cầu, nhưng chúng ta phải rời bỏ nhánh master. Tất cả những lần commit được tạo ra từ đây sẽ dẫn bạn đi trên một nhánh khác, nhánh này có thể được đặt tên sau.

Mặt khác, sau khi *check out* một trạng thái cũ, Git tự động đặt bạn vào một trạng thái mới, một nhánh chưa có tên, và nhánh này có thể đặt tên và ghi lại với lệnh **git checkout -b**.

Sửa Nhanh

Bạn đang phân vân giữa ngã ba đường khi bạn phải quyết định là xóa tất cả mọi thứ hoặc là sửa chữa các lỗi mới phát hiện ra trong lần commit `1b6d...`:

```
$ git commit -a
$ git checkout -b fixes 1b6d # checkout và đặt tên là nhánh fixes
```

Sau khi hoàn tất việc sửa chữa:

```
$ git commit -a -m "Đã sửa"
$ git checkout master
```

và sau đó quay lại công việc theo phận sự của mình. Bạn thậm chí có thể trộn với lần commit đã sửa để sửa lỗi:

```
$ git merge fixes
```

Trộn

Với một số hệ thống quản lý mã nguồn, việc tạo các nhánh rất dễ dàng nhưng trộn chúng trở lại là một bài toán hóc búa. Với Git, việc trộn là dễ dàng và bạn có thể không hay biết nó hoạt động như thế nào.

Chúng ta đã sử dụng việc trộn từ lâu rồi. Lệnh **pull** trên thực tế đã *fetch* (lấy về) các lần commit và sau đó trộn chúng vào trong nhánh hiện hành của bạn. Nếu trên máy của mình bạn không có thay đổi gì cả, thế thì việc trộn sẽ là một *fast forward* (chuyển tiếp nhanh), trường hợp này cũng na ná như việc lấy về phiên bản cuối cùng trong hệ thống quản lý mã nguồn tập trung. Nhưng nếu bạn đã có thay đổi trên máy của mình, Git sẽ tự động trộn, và báo lỗi cho bạn nếu có xung đột xảy ra.

Thông thường, mỗi lần commit có một *commit cha*, tạm gọi thế, chính là lần commit trước. Việc trộn các nhánh với nhau phát sinh ra một lần commit với ít nhất hai *cha*. Điều này đặt ra câu hỏi: lần commit mà `HEAD~10` thực sự ám chỉ đến là lần nào? Một lần commit có thể có nhiều cha, thế thì chúng ta phải theo cái nào?

Nó sẽ gọi ra *cha* đầu tiên. Đây là điều ta mong muốn bởi vì nhánh hiện hành trở thành cha đầu tiên trong suốt quá trình trộn; thường, bạn chỉ liên quan đến những thay đổi mình tạo ra trong nhánh hiện hành, cốt để mà đối lập với việc trộn thay đổi từ các nhánh khác.

Bạn hãy nhớ Git quy một cha nào đó với một dấu mũ. Ví dụ, để hiển thị nhật ký tính từ *cha* thứ hai:

```
$ git log HEAD^2
```

Bạn có thể bỏ qua số dành cho cha đầu tiên. Ví dụ, để hiển thị sự khác nhau với cha đầu tiên:

```
$ git diff HEAD^
```

Bạn có thể tổ hợp các dấu mũ này với các kiểu khác nhau. Ví dụ:

```
$ git checkout 1b6d^^2~10 -b ancient
```

bắt đầu một nhánh mới “ancient” tương ứng với trạng thái lần commit thứ 10 trở về trước từ cha thứ hai của cha thứ nhất của lần commit bắt đầu với 1b6d.

Làm Việc Liên Tục

Thường trong các dự án phần cứng, bước thứ hai của kế hoạch phải chờ bước thứ nhất hoàn thành. Một chiếc xe hơi cần sửa chữa có thể phải nằm chờ trong xưởng sửa chữa cho đến khi các chi tiết phụ tùng đặc biệt được chuyển đến từ nhà máy. Một mẫu có thể phải chờ một con chip được làm ra trước khi quá trình chế tác có thể tiếp tục.

Dự án phần mềm cũng có thể tương tự như thế. Bộ phận thứ hai có một số tính năng có thể phải chờ cho đến khi phần thứ nhất đã được phát hành và kiểm tra. Một số dự án yêu cầu mã nguồn của bạn phải được xem xét lại trước khi chấp nhận nó, vì vậy bạn có thể phải chờ cho đến khi bộ phận thứ nhất đã được chấp thuận trước khi bắt đầu phần thứ hai.

Nhờ có việc tạo nhánh và trộn dễ dàng và cũng chẳng mất mát gì, chúng ta có thể phá vỡ quy tắc và làm việc trên Part II trước khi Part I chính thức sẵn sàng. Giả sử bạn đã commit Part I và gửi nó đi để xem xét. Giả sử bạn đang ở nhánh master. Thế thì hãy phân nhánh ra:

```
$ git checkout -b part2
```

Tiếp đến, làm việc trên Part II, commit những thay đổi của bạn bao nhiêu tùy thích. Lỗi là ở con người, và bạn sẽ phải thường xuyên quay trở lại để sửa lỗi nào đó trong Part I. Nếu may mắn, hay trong trường hợp tốt, bạn có thể bỏ qua những dòng này.

```
$ git checkout master  # Quay trở lại Part I.
$ sửa_lỗi
$ git commit -a        # Commit sau khi sửa lỗi.
$ git checkout part2   # Quay trở lại Part II.
$ git merge master     # Trộn các thay đổi.
```

Cuối cùng, Part I được chấp thuận:

```
$ git checkout master  # Quay trở lại Part I.
$ submit files         # Xuất bản ra!
$ git merge part2      # Trộn vào Part II.
$ git branch -d part2  # Xóa nhánh "part2".
```

Bây giờ chúng ta lại ở trong nhánh master, với Part II trong thư mục làm việc.

Thủ thuật này rất dễ dàng để mở rộng ra dành cho nhiều phần hơn. Nó cũng để dàng để phân nhánh ra từ quá khứ: giả sử muộn bạn mới nhận ra là mình phải tạo một nhánh từ trước đây 7 lần commit. Thế thì gõ:

```
$ git branch -m master part2  # Đổi tên nhánh "master" thành "part2".
$ git checkout HEAD~7 -b master # Tạo nhánh "master" mới 7 lần commit ngược từ trước.
```

Nhánh master bây giờ chỉ chứa Part I, và nhánh part2 trở thành nhánh chứa. Chúng ta đang ở nhánh sau; chúng ta đã tạo nhánh master mà không chuyển đến nó, bởi vì chúng ta muốn tiếp tục làm việc trên part2. Điều này hơi bất thường. Cho đến lúc này, Chúng ta chuyển đến các nhánh ngay sau khi chúng được tạo ra, như là trong:

```
$ git checkout HEAD~7 -b master  # Tạo một nhánh và chuyển tới nó.
```

Cải Tổ Lại Sự Pha Trộn

Có lẽ bạn thích làm việc trên mọi khía cạnh của một dự án trên cùng một nhánh. Bạn muốn giữ riêng các thay đổi mình đang làm cho riêng mình và muốn những người khác chỉ thấy được các lần commit của bạn sau khi đã được tổ chức lại. Hãy chuẩn bị một cặp nhánh:

```
$ git branch -b sanitized # Tạo một nhánh dùng cho việc dọn.
```

```
$ git checkout -b medley # Tạo và chuyển nhánh thành nơi làm việc.
```

Tiếp theo, làm việc gì đó: sửa lỗi, thêm các đặc tính kỹ thuật, thêm mã lệnh tạm thời, vân vân, commit thường xuyên. Sau đó:

```
$ git checkout sanitized # tạm dịch: đã được vệ sinh
$ git cherry-pick medley^^ # tạm dịch: hỗn độn; ^^: ông bà
```

áp dụng nhánh ông-bà của lần commit head của nhánh “medley” thành nhánh “sanitized”. Với lệnh thích hợp là cherry-picks bạn có thể cấu trúc một nhánh mà nó chỉ chứa mã nguồn không thay đổi, và những lần commit có liên quan sẽ được nhóm lại với nhau.

Quản Lý Các Nhánh

Liệt kê tất cả các nhánh bằng cách gõ:

```
$ git branch
```

Theo mặc định, bạn bắt đầu tại nhánh có tên “master”. Một số người chủ trương rời bỏ nhánh “master” mà không động chạm gì đến nó và tạo các nhánh mới dành cho các chỉnh sửa của riêng mình.

Các tùy chọn **-d** and **-m** cho phép bạn xóa hay di chuyển (đổi tên) các nhánh. Xem thêm **git help branch**.

Nhánh “master” thông thường rất hữu dụng. Những người khác sẽ nghĩ rằng kho chứa của bạn có nhánh mang tên này, và nhánh đó chứa phiên bản chính thức của dự án của bạn. Mặc dù bạn có thể đổi tên hay xóa bỏ nhánh “master”, nhưng bạn không nên làm như thế mà hãy tôn trọng thỏa thuận ngầm này.

Nhánh Tạm

Một lát sau bạn có lẽ nhận thức được rằng mình cần có các nhánh tạm thời vì các lý do như: mọi nhánh khác đơn thuần phục vụ cho việc ghi lại trạng thái hiện tại do vậy bạn có thể nhảy trở lại các trạng thái cũ hơn để mà sửa chữa các lỗi nghiêm trọng hay làm một cái gì đó.

Điều này cũng tương tự như việc chuyển kênh trên TV một cách tạm thời để thấy chương trình khác đang chiếu cái gì. Nhưng thay vì chỉ cần nhấn vài cái nút, bạn phải tạo, “checkout”, trộn và xóa nhánh tạm đó. May mắn thay, Git có cách ngắn gọn tiện lợi chẳng thua kém gì chiếc điều khiển từ xa của một chiếc TV:

```
$ git stash
```

Lệnh này ghi lại trạng thái hiện hành vào một vị trí tạm thời (một **stash**) và phục hồi lại trạng thái trước đó. Thư mục bạn đang làm việc xuất hiện chính xác như trước khi bạn chỉnh sửa, và bạn có thể sửa lỗi, pull về các thay đổi ngược dòng, và cứ như thế. Khi bạn muốn qua trở lại trạng thái đã được tạm giấu đi đó, hãy gõ:

```
$ git stash apply  # Bạn có thể sẽ phải giải quyết các xung đột có thể nảy sinh.
```

Bạn có thể có nhiều trạng thái được tạm giấu đi, và vận dụng chúng theo nhiều cách khác nhau. Xem **git help stash** để biết thêm chi tiết. Bạn cũng có thể đoán được, Git duy trì các nhánh ở hậu trường để thực hiện việc này.

Làm Theo Cách Của Mình

Bạn có thể sẽ cảm thấy việc sử dụng nhánh phiền hà quá. Cuối cùng, **clone** có lẽ là lệnh nhanh nhất, và bạn có thể hoán chuyển giữa chúng với lệnh **cd** thay vì sử dụng lệnh riêng của Git.

Ta thử xét đến các trình duyệt web. Tại sao việc hỗ trợ mở nhiều tab thì cũng tốt như mở trên nhiều cửa sổ khác nhau? Bởi vì cả hai điều này thể hiện tính đa dạng của quan điểm, phong cách sống. Một số người sử dụng lại thích chỉ giữ một cửa sổ trình duyệt được mở, và sử dụng các tab để hiển thị nhiều trang web một lúc. Những người khác có lẽ lại khăng khăng cực đoan cho rằng: mở trên nhiều cửa sổ khác nhau và chẳng cần tab nữa. Một nhóm khác lại thích cả hai cách trên.

Việc tạo nhánh thì cũng giống như tạo các tab cho thư mục làm việc của bạn, còn việc nhân bản thì lại giống như việc mở một cửa sổ duyệt mới. Những việc này nhanh chóng và nội bộ, thế thì sao lại không thử nghiệm để tìm thấy cách nào thích hợp nhất cho mình? Git giúp bạn làm việc chính xác như bạn muốn.

Chương 5. Bài Học về Lịch Sử

Một hệ quả tất yếu của đặc tính phân tán của Git là việc lịch sử có thể biên soạn lại một cách dễ dàng. Nhưng nếu bạn xáo trộn quá khứ, hãy cẩn thận: chỉ biên soạn lại các phần trong lịch sử chỉ khi bạn sở hữu nó một mình. Cũng giống như việc các quốc gia tranh cãi không kết thúc xem ai là người tận tâm, hành động nào là tàn ác, nếu một người khác có một bản sao mà lịch sử của nó lại khác với cái của bạn, bạn sẽ gặp rắc rối ngay khi cần tương tác với họ.

Một số nhà phát triển phần mềm quả quyết rằng lịch sử không thể thay đổi, tất cả mọi thứ. Một số khác lại cho rằng chỉnh sửa lại cấu trúc trước khi phát hành nó ra đại chúng. Git chấp nhận cả hai quan điểm. Giống như việc nhân bản, tạo nhánh và hòa trộn, viết lại lịch sử đơn giản chỉ là một quyền lực mà Git trao cho bạn. Bạn có thể làm thế nếu muốn.

Dừng Lại Sửa Chữa

Bạn vừa mới commit, nhưng lại ước rằng mình đã gõ những dòng chú thích có nội dung khác phải không? Thế thì hãy chạy:

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

để thay đổi chú thích cuối cùng. Bạn giật mình vì quên thêm các tập tin vào? Chạy lệnh **git add** để thêm nó vào, và sau đó lại chạy lệnh ở trên.

Bạn muốn thêm vài chỉnh sửa vào lần cuối mình đã commit ư? Thế thì cứ sửa chúng đi và sau đó chạy lệnh:

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

... Và Sau đó là Nhiều Lần

Giả sử vấn đề trục trặc ở lần commit cách đây mười lần. Sau một buổi làm việc dài, bạn đã tạo ra hàng tá các lần commit. Nhưng bạn không hoàn toàn hài lòng với cách mà chúng được tổ chức, và một số lần commit cần được soạn lại phần mô tả. Thế thì hãy gõ:

```
$ git rebase -i HEAD~10
```

và 10 lần commit cuối sẽ xuất hiện trong $EDITOR yêu thích của bạn. Dưới đây là một đoạn trích ví dụ:

```
pick 5c6eb73 Added repo.or.cz link
pick a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

Lần commit cũ đứng trước lần mới hơn trong danh sách, không giống như kết quả khi chạy lệnh `log`. Ở đây, 5c6eb73 là lần commit cũ nhất, và 100834f là mới nhất. Thế thì:

- Xóa bỏ các lần commit bằng cách xóa các dòng tương ứng. Giống như lệnh revert, nhưng không ghi biên bản: nó sẽ coi như là lần commit đó chưa từng bao giờ tồn tại.

- Đặt lại thứ tự các lần commit bằng cách thay đổi thứ tự các dòng.
- Thay thế pick bằng:
  - edit để đánh dấu lần commit đó là dành cho việc tu bổ.
  - reword để thay đổi phần chú giải.
  - squash để hòa trộn với lần commit trước.
  - fixup để hòa trộn với lần commit trước và bỏ qua việc ghi lại phần chú giải.

Ví dụ, chúng ta chẳng hạn thay thế pick ở dòng thứ hai bằng squash:

```
pick 5c6eb73 Added repo.or.cz link
squash a311a64 Reordered analogies in "Work How You Want"
pick 100834f Added push target to Makefile
```

Sau đó chúng ta ghi lại thay đổi và thoát ra. Git trộn lần a311a64 vào 5c6eb73. Vì vậy **squash** trộn với lần kế trước nó: có thể nghĩ đây là quá trình "nén dữ liệu".

Hơn thế nữa, Git sau đó tổ hợp nhật ký của chúng và hiện tại và chỉnh sửa lại. Lệnh **fixup** bỏ qua bước này; việc sửa nhật ký đơn giản là bỏ qua.

Nếu bạn đánh dấu một lần commit bằng **edit**, Git đưa bạn trở lại quá khứ, tới lần commit lâu nhất đó. Bạn có thể tu bổ một lần commit cũ như đã mô tả ở phần trên, và thậm chí tạo ra các lần commit mới ở chỗ này. Một khi bạn đã hài lòng với việc "retcon", hãy chạy *cỗ máy thời gian* bằng cách chạy lệnh:

```
$ git rebase --continue
```

Git sửa commits cho tới **edit** kế tiếp, hoặc tới hiện tại nếu không còn việc gì cần phải làm.

Bạn còn có thể bãi bỏ việc rebase bằng lệnh:

```
$ git rebase --abort
```

Do vậy cứ commit thoải mái và thường xuyên bởi vì bạn có thể dọn dẹp cho gọn gàng sau này bằng lệnh rebase.

Thay Đổi Riêng Sắp Xếp Sau

Bạn đang làm việc trên một dự án đang hoạt động. Bạn đã tạo ra một số lần commit tại máy tính của mình, và sau đó bạn đồng bộ hóa với cây chính thức bằng cách hòa trộn. Chu kỳ này tự lặp chính nó một số lần trước khi bạn thực sự push tới cây trên máy chủ trung tâm.

Nhưng hiện tại lịch sử bản sao Git trên máy tính của bạn là một mớ hỗn độn của những lần thay đổi trên máy tính riêng và máy chính thức. Bạn muốn thấy tất cả các thay đổi của riêng mình trong một đoạn liên tục không ngắt quãng, và sau tất cả các thay đổi từ kho chính thức.

Đây chính là công việc dành cho lệnh **git rebase** đã được miêu tả ở trên. Trong nhiều trường hợp bạn có thể sử dụng cờ **--onto** và tránh xa sự tương tác với các máy tính khác.

Xem thêm trong **git help rebase** để thấy được chi tiết các ví dụ dành cho lệnh đáng kinh ngạc này. Bạn có thể chia cắt các lần commit. Bạn còn có thể xắp xếp lại các nhánh của một cấu trúc cây.

Hãy cẩn thận: rebase là một lệnh mạnh mẽ. Với những lần rebases phức tạp, trước hết hãy tạo ra một bản sao lưu dự phòng bằng lệnh **git clone**.

Viết Lại Lịch Sử

Thỉnh thoảng, bạn muốn việc quản lý mã nguồn giống việc người ta sơn vẽ chân dung một con người, tẩy xóa chúng từ lịch sử như theo ý của Stalinesque. Ví dụ, giả sử chúng ta có ý định phát hành dự án, nhưng lại liên đới đến một tập tin mà nó phải được giữ bí mật vì lý do nào đó. Chẳng hạn như tôi đã quên khi ghi lại số thẻ tín dụng vào trong một tập tin văn bản và ngẫu nhiên nó được thêm vào trong dự án. Việc xóa tập tin này là chưa đủ, bởi vì ta có thể đọc nó từ lần commit cũ. Chúng ta phải gỡ bỏ tập tin này từ tất cả các lần đã commit:

```
$ git filter-branch --tree-filter 'rm top/secret/file' HEAD
```

Xem **git help filter-branch**, nội dung của nó sẽ thảo luận về ví dụ này và đưa ra một cách thức nhanh hơn. Đại thể, lệnh **filter-branch** giúp bạn thay đổi cả một chương lớn của lịch sử chỉ chỉ bằng một lệnh đơn.

Sau này, thư mục .git/refs/original mô tả trạng thái của công việc trước khi thực hiện. Kiểm tra lệnh filter-branch đã làm những thứ bạn muốn chưa, sau đó xóa thư mục này đi nếu bạn muốn chạy lệnh filter-branch lần nữa.

Cuối cùng, thay thế bản sao của dự án của bạn bằng phiên bản bạn đã sửa lại nếu bạn muốn tương thích với chúng sau này.

Tự Tạo Lịch Sử

Bạn muốn chuyển đổi dự án của mình sang sử dụng Git? Nếu nó được quản lý bởi một hệ thống nổi tiếng hơn, thế thì nếu may mắn sẽ có người nào đó đã viết sẵn một đoạn kịch bản để xuất toàn bộ lịch sử ra thành Git.

Nếu không, thì nên xem xét đến việc sử dụng lệnh **git fast-import**, lệnh này đọc văn bản đầu vào ở một định dạng riêng để mà tạo ra một lịch sử cho Git từ ban đầu. Thông thường một script sử dụng lệnh này là một nhóm lệnh tổ hợp với nhau và chỉ chạy một lần, di chuyển một dự án chỉ bằng một lệnh đơn.

Dưới đây là một ví dụ, dán danh sách theo sau đâu vào một tập tin tạm thời nào đó, chẳng hạn như là /tmp/history:

```
commit refs/heads/master
committer Alice <alice@example.com> Thu, 01 Jan 1970 00:00:00 +0000
data <<EOT
Initial commit.
EOT

M 100644 inline hello.c
data <<EOT
#include <stdio.h>
```

```
int main() {
  printf("Hello, world!\n");
  return 0;
}
EOT


commit refs/heads/master
committer Bob <bob@example.com> Tue, 14 Mar 2000 01:59:26 -0800
data <<EOT
Replace printf() with write().
EOT

M 100644 inline hello.c
data <<EOT
#include <unistd.h>

int main() {
  write(1, "Hello, world!\n", 14);
  return 0;
}
EOT
```

Thế thì tạo một kho Git từ thư mục chứa các tập tin tạm thời này bằng cách gõ:

```
$ mkdir project; cd project; git init
$ git fast-import --date-format=rfc2822 < /tmp/history
```

Bạn có thể checkout phiên bản cuối của dự án với:

```
$ git checkout master .
```

Lệnh **git fast-export** chuyển đổi bất kỳ một kho chứa nào thành định dạng phù hợp với lệnh **git fast-import**, và bạn có thể nghiên cứu nó để tạo riêng cho mình một chương trình xuất, và cũng làm như thế để tạo thành kho chuyên chở ở định dạng con người có thể đọc được. Thực vậy, những lệnh này có thể gửi một kho chứa ở dạng văn bản thông qua một kênh chỉ cho phép văn bản truyền đi.

Vị Trí Nào Phát Sinh Lỗi?

Bạn vừa mới phát hiện ra một đặc tính không hoạt động trong chương trình mà bạn chắc chắn là nó đã hoạt động vài tháng trước. Tệ quá! Bạn tự hỏi là lỗi bắt đầu từ chỗ nào nhỉ? Nếu như chỉ có mình bạn kiểm tra cũng như phát triển đặc tính này.

Lúc này thì đã quá muộn rồi. Tuy nhiên, chỉ cần bạn commit thường xuyên, Git có thể xác định vị trí của trục trặc:

```
$ git bisect start
$ git bisect bad HEAD
$ git bisect good 1b6d
```

Git checks out một trạng thái nằm giữa chúng. Kiểm tra đặc tính kỹ thuật, và nếu nó vẫn hỏng:

```
$ git bisect bad
```

Nếu không thì thay "bad" bằng "good". Git sẽ chuyên chở bạn qua lại nửa bước giữa hai trạng thái là phiên bản "tốt" và "xấu", thu hẹp khả năng lại. Sau khi lặp đi lặp lại một số lần, việc tìm kiếm nhị phân này sẽ dẫn bạn đến lần commit mà nó làm nguyên nhân dẫn đễ hỏng hóc. Một khi bạn đã hoàn tất việc điều tra, trở lại trạng thái nguyên bản của bạn bằng cách gõ:

```
$ git bisect reset
```

Thay vì thử nghiệm mọi thay đổi một cách thủ công, hãy tự động hóa sự tìm kiếm này bằng cách chạy:

```
$ git bisect run my_script
```

Git sử dụng giá trị trả về của lệnh đã cho, thông thường là từ các đoạn kịch bản, để quyết định là lệnh đã thực hiện tốt hay không: lệnh sẽ trả về giá trị 0 khi tốt, 125 khi sự thay đổi bị bỏ qua, và bất kỳ giá trị nào khác nằm giữa 1 và 127 nếu gặp lỗi. Một giá trị âm sẽ bãi bỏ lệnh bisect.

Bạn có thể làm nhiều hơn thế: trang trợ giúp giảng giải cho bạn làm thế nào để hiểu được lệnh bisect làm việc như thế nào, xem xét hay xem lại nhật ký lệnh bisect, và loại trừ các thay đổi ngớ ngẩn để tăng tốc độ tìm kiếm.

Ai Đã Làm Nó Sai?

Giống như các hệ thống quản lý mã nguồn khác, Git cũng có lệnh blame:

```
$ git blame bug.c
```

lệnh này chú thích tất cả các dòng có trong tập tin được chỉ ra cho thấy ai là người cuối cùng sửa nó, và là khi nào. Không giống các hệ thống quản lý mã nguồn khác, hành động này hoạt động không cần có mạng, việc đọc chỉ đơn thuần từ ổ đĩa của máy tính cá nhân.

Kinh Nghiệm Riêng

Trong một hệ thống quản lý mã nguồn tập trung, thay đổi lịch sử là một việc làm khó khăn, và chỉ làm được thế nếu đó là người quản trị. Việc nhân bản, tạo nhánh và trộn không thể thiếu việc truyền thông qua mạng. Cũng như thế với các tác vụ cơ bản khác như là duyệt lịch sử, hay là commit một thay đổi. Trong một số hệ thống khác, người dùng yêu cầu có kết nối mạng chỉ để xem các thay đổi của họ hay mở một tập tin để biên tập.

Hệ thống tập trung không cho phép làm việc mà không có mạng, và đòi hỏi cơ sở hạ tầng mạng máy tính đắt đỏ tốn kém, đặc biệt là khi số nhà phát triển phần mềm tăng lên. Điều quan trọng, tất cả mọi tác vụ sẽ chậm hơn ở mức độ nào đó, thường thường những người sử dụng sẽ lảng tránh việc sử dụng các lệnh cao cấp trừ phi nó thực sự cần thiết. Trừ những trường hợp đặc biệt là các lệnh cơ bản. Khi những người dùng phải chạy các lệnh chạy chậm, hiệu suất bị giảm bởi vì nó làm gián đoạn công việc của cả một dây truyền.

Tôi trực tiếp đã trải qua những hiện tượng đó. Git là hệ thống quản lý mã nguồn đầu tiên tôi sử dụng. Tôi nhanh chóng làm quen với nó, bị quyến rũ bởi những đặc tính kỹ thuật mà nó đã cung cấp. Tôi đơn giản cho rằng các hệ thống khác thì cũng tương tự: việc chọn lựa một hệ thống quản lý mã nguồn thì cũng chẳng khác việc chọn một trình biên soạn hay một trình duyệt web.

Tôi sẽ sốc nếu như sau này bị bắt buộc sử dụng hệ thống quản lý mã nguồn tập trung. Một kết nối Internet chậm chạp cũng chẳng phải là vấn đề lớn đối với Git, nhưng nó sẽ làm cho các nhà phát triển phần mềm không thể chịu nổi khi nó cần sự tin cậy như ổ đĩa nội bộ. Thêm nữa, tôi đã gặp một số mắc mớ với một số lệnh, mà chính nó đã ngăn cản tôi làm việc một cách trôi chảy.

Khi tôi phải chạy những lệnh cần nhiều thời gian, việc làm ngắt quãng việc suy nghĩ sẽ gây nên thiệt hại rất to lớn. Trong khi chờ cho việc truyền thông với máy chủ hoàn tất, tôi sẽ phải làm một vài thứ gì đó khác để lấp chỗ trống, chẳng hạn như lấy thư điện tử hay viết tài liệu. Sau một khoảng thời gian tôi quay trở lại nhiệm vụ chính của mình, lệnh đã hoàn tất từ lâu rồi, và tôi phải lãng phí thêm nhiều thời gian nữa để nhớ lại xem mình đang làm gì. Con người thường dở khi phải thay đổi mạch văn.

Ở đây còn có một hậu quả rất đáng quan tâm nữa: đoán trước được việc tắc nghẽn của mạng máy tính, nhiều cá nhân riêng lẻ có thể chiếm dụng nhiều lưu lượng mạng hơn cần thiết trên các tác vụ khác nhau để cố gắng giảm thiểu sự chậm trễ có thể xảy ra trong tương lai. Hậu quả cuối cùng là sự quá tải quá mức, chính việc vô tình ủng hộ việc tiêu dùng cá nhân như thế đã tiêu tốn nhiều lưu lượng mạng hơn và sau đó nó làm cho việc tắc nghẽn càng lúc càng trở nên tồi tệ hơn.

Chương 6. Đa Người Dùng

Lúc đầu tôi sử dụng Git cho dự án riêng của mình mà tôi cũng là người duy nhất phát triển nó. Trong số những lệnh liên quan đến bản chất phân tán của Git, tôi chỉ cần lệnh **pull** và **clone** để giữ cùng một dự án nhưng ở những chỗ khác nhau.

Sau đó tôi muốn mã nguồn của mình được phổ biến trên mạng bằng việc sử dụng Git, và bao gồm cả những thay đổi từ những người đóng góp. Tôi đã phải học cách làm thế nào để quản lý các dự án có nhiều người phát triển phần mềm ở khắp nơi trên toàn thế giới. May mắn thay, đây là sở trường của Git, và người ta có thể nói đây là điều sống còn của một hệ thống quản lý mã nguồn.

Tôi Là Ai?

Mỗi lần commit sẽ lưu giữ tên và địa chỉ thư điện tử, điều này có thể nhìn thấy bằng lệnh **git log**. Theo mặc định, Git sử dụng các trường để lưu giữ các cài đặt trong hệ thống của mình. Để cài đặt các thông tin cá nhân của mình vào, hãy gõ:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Bỏ qua cờ global để đặt những thông tin này chỉ sử dụng cho kho chứa hiện tại.

Git Thông Qua SSH, HTTP

Giả sử bạn có thể truy cập vào một máy chủ web qua SSH, nhưng Git lại chưa được cài đặt ở đây. Mặc dù không hiệu quả như giao thức nguyên bản của nó, nhưng Git vẫn có thể truyền thông thông qua HTTP.

Tải về, dịch và cài Git bằng tài khoản của bạn, và tạo kho chứa tại thư mục chứa trang web của bạn:

```
$ GIT_DIR=proj.git git init
$ cd proj.git
$ git --bare update-server-info
$ cp hooks/post-update.sample hooks/post-update
```

Với các phiên bản Git cũ, lệnh copy không thực hiện được và bạn phải chạy:

```
$ chmod a+x hooks/post-update
```

Từ giờ bạn có thể xuất bản mới nhất của mình thông qua SSH từ một bản sao bất kỳ:

```
$ git push máy.chủ.web:/đường/dẫn/đến/proj.git master
```

và mọi người có thể lấy dự án của bạn với lệnh:

```
$ git clone http://máy.chủ.web/proj.git
```

Git Thông Qua Mọi Thứ

Bạn muốn đồng bộ hóa kho chứa nhưng lại không có máy chủ và cũng không có mạng? Bạn cần trong những trường hợp khẩn cấp? Chúng ta đã biết lệnh **git fast-**

export và **git fast-import** có thể chuyển đổi một kho chứa thành một tập tin đơn và ngược lại. Chúng ta có thể chuyển qua chuyển lại như vậy để truyền kho Git đi thông qua bất kỳ phương tiện nào, nhưng có một công cụ hiệu quả hơn đó chính là **git bundle**.

Người gửi tạo một *bundle*:

```
$ git bundle create somefile HEAD
```

sau đó gửi bundle, somefile, tới người cần bằng cách nào đó: thư điện tử, ổ đĩa USB, và bản in **xxd** và một bộ quét nhận dạng chữ OCR, đọc các bit thông qua điện thoại, tín hiệu khói, v.v.. Người nhận khôi phục lại các lần commit từ bundle nhận được bằng cách gõ:

```
$ git pull somefile
```

Bộ nhận thậm chí có thể làm được việc này từ một kho chứa rỗng. Mặc dù kích thước của nó, somefile chứa các mục kho Git nguyên thủy.

Trong các dự án lớn hơn, việc triệt tiêu lãng phí bằng cách chỉ bundle những thay đổi còn thiếu kho chứa khác. Ví dụ, giả sử lần commit “1b6d...” là lần commit gần nhất đã được chia sẻ giữa cả hai thành viên:

```
$ git bundle create somefile HEAD ^1b6d
```

Nếu phải làm việc này thường xuyên, một khó khăn là bạn không thể nhớ được chính xác lần commit tương ứng với lần gửi cuối. Trang trợ giúp sẽ gợi ý cho bạn cách sử dụng các thẻ (tag) để giải quyết vấn đề này. Ấy là, sau khi bạn gửi một bundle, thì hãy gõ:

```
$ git tag -f lastbundle HEAD
```

và tạo một bản bundles mới với:

```
$ git bundle create newbundle HEAD ^lastbundle
```

Vá: Sự Thịnh Hành Toàn Cầu

Miếng vá được trình bày ở dạng văn bản để thể hiện các thay đổi của bạn, nó dễ dàng được đọc hiểu bởi con người cũng như là máy tính. Điều này mang lại cho chúng sức lôi cuốn toàn cầu. Bạn có thể gửi miếng vá qua thư điện tử cho những nhà phát triển phần mềm khác mà chẳng cần lo họ đang sử dụng hệ thống quản lý mã nguồn nào. Chừng nào mà độc giả của bạn có thể đọc được thư điện tử của mình thì họ còn có thể thấy được phần chỉnh sửa của bạn. Tương tự thế, về phía mình, những thứ bạn cần là có một địa chỉ thư điện tử: ở đây chẳng cần cài đặt kho chứa Git nào trên mạng.

Sử dụng lại ví dụ từ chương đầu tiên:

```
$ git diff 1b6d > my.patch
```

đầu ra là một miếng vá mà bạn có thể dán vào một thư điện tử để trao đổi với người khác. Ở kho Git, gõ:

```
$ git apply < my.patch
```

để áp dụng miếng vá.

Còn một hình thức định dạng khác nữa, tên và có lẽ cả chữ ký của tác giả cũng được ghi lại, tạo miếng vá tương ứng với một thời điểm chính xác trong quá khứ bằng cách gõ:

```
$ git format-patch 1b6d
```

Tệp tin lưu kết quả có thể chuyển cho lệnh **git-send-email**, hay có thể làm thủ công. Bạn cũng có thể chỉ định rõ một vùng commit:

```
$ git format-patch 1b6d..HEAD^^
```

Ở phía người nhận cuối, ghi lại thư điện tử thành tập tin, sau đó chạy lệnh:

```
$ git am < email.txt
```

Lệnh này sẽ áp dụng cho miếng vá nhận được, đồng thời tạo ra một lần commit, bao gồm các thông tin như là tác giả.

Với một chương trình đọc thư điện tử, bạn có thể sử dụng con chuột để chuyển định dạng thư về dạng văn bản thuần trước khi ghi miếng vá thành một tập tin.

Có một số khác biệt nhỏ giữa các trình đọc đọc thư điện tử, nhưng nếu bạn sử dụng một trong số chúng, bạn hầu như chắc chắn là người mà có thể cấu hình chúng một cách dễ dàng mà chẳng cần phải đọc hướng dẫn sử dụng!

Rất tiếc! Tôi đã chuyển đi

Sau khi nhân bản kho chứa, việc chạy lệnh **git push** hay **git pull** sẽ tự động push tới hay pull từ URL gốc. Git đã làm điều này như thế nào? Bí mật nằm ở chỗ các tùy chọn config đã được tạo ra cùng với bản sao. Hãy xem thử:

```
$ git config --list
```

Tùy chọn `remote.origin.url` sẽ lưu giữ URL; “origin” là cái tên được đặt cho kho nguồn. Với nhánh “master” theo như thường lệ, chúng ta có thể thay đổi hay xóa các tên này nhưng chẳng có lý do gì để phải làm như thế cả.

Nếu kho chứa nguyên bản đã chuyển chỗ, chúng ta có thể cập nhật URL thông qua:

```
$ git config remote.origin.url git://url.mới/proj.git
```

Tùy chọn `branch.master.merge` chỉ ra nhánh remote mặc định trong lệnh **git pull**. Trong suốt quá trình nhân bản, nó được đặt cho nhánh hiện hành của kho chứa mã nguồn, như thế cho dù HEAD của kho nguồn về sau có di chuyển đến một nhánh khác, lệnh pull sau này sẽ trung thành với nhánh nguyên gốc.

Tùy chọn này chỉ áp dụng cho kho chứa chúng ta lần đầu tiên nhân bản từ, nó được ghi trong tùy chọn `branch.master.remote`. Nếu chúng ta pull từ kho chứa khác chúng ta phải chỉ đích xác tên nhánh mà chúng ta muốn:

```
$ git pull git://example.com/other.git master
```

Phần phía trên giải thích tại sao một số lệnh push và pull ví dụ của chúng ta lại không có tham số.

Nhánh Trên Mạng

Khi bạn nhân bản một kho chứa, bạn cũng đồng thời nhân bản tất cả các nhánh của nó. Bạn sẽ không nhận được cảnh báo này bởi vì Git không thông báo cho bạn: bạn phải hỏi mới có thể biết chính xác. Việc làm này giúp ngăn ngừa phiền phức do nhánh mạng gây ra cho các nhánh của bạn, và cũng làm cho Git dễ dàng hơn với người mới dùng.

Ta liệt kê các nhánh bằng lệnh:

```
$ git branch -r
```

Bạn nhận được kết quả trông giống như thế này:

```
origin/HEAD
origin/master
origin/experimental
```

Những nhánh tương ứng và HEAD của kho chứa remote, và bạn có thể sử dụng trong các lệnh Git thông thường. Ví dụ như là, giả sử bạn làm nhiều lần commit, và muốn so sánh chúng với bản đã fetch cuối cùng. Bạn cũng có thể tìm kiếm trong tập tin log để có được giá trị băm SHA1 thích hợp, nhưng dễ dàng hơn việc gõ:

```
$ git diff origin/HEAD
```

Hay bạn có thể xem nhánh “experimental” đang làm gì:

```
$ git log origin/experimental
```

Đa Máy chủ

Giả sử hai người cùng phát triển trên một sự án của chúng ta, và họ muốn giữ lại sự khác biệt trên cả hai. Chúng ta theo dõi hơn một kho chứa tại một thời điểm với lệnh:

```
$ git remote add other git://example.com/some_repo.git
$ git pull other some_branch
```

Bây giờ chúng ta có thể trộn với nhánh của kho chứa thứ hai, và chúng ta dễ dàng truy cập tất cả các nhánh của tất cả các kho chứa:

```
$ git diff origin/experimental^ other/some_branch~5
```

Nhưng chúng ta chỉ muốn so sánh sự khác nhau giữ chúng nhưng không áp dụng các thay đổi này với chúng ta? Nói cách khác, chúng ta khảo sát các nhánh của họ nhưng không thay đổi những gì đang có trong thư mục làm việc của mình. Thế thì thay vì pull, ta dùng lệnh:

```
$ git fetch       # Lấy về từ nguyên gốc, theo mặc định.
$ git fetch other  # Lấy về từ lập trình viên thứ hai.
```

Lệnh này chỉ mang về phần lịch sử. Mặc dù thư mục làm việc vẫn còn nguyên chưa bị động đến, chúng ta có thể xét bất kỳ nhánh nào của bất kỳ kho chứa nào trong một lệnh Git bởi vì chúng ta bây giờ đang làm việc trên bản sao trên máy của mình.

Giờ ta xét đến phần hậu trường, lệnh pull đơn giản là **fetch** sau đó **merge**. Thông thường chúng ta **pull** bởi vì chúng ta muốn trộn với lần commit cuối cùng sau khi fetch; việc làm này là một ngoại lệ đáng chú ý.

Xem **git help remote** để biết cách làm thế nào để gỡ bỏ kho chứa trên mạng, bỏ qua các nhánh xác định, và những thứ khác nữa.

Sở Thích Riêng Của Tôi

Với dự án của mình, tôi thích những người cộng tác tạo các kho chứa ở nơi mà tôi có thể pull. Một số dịch vụ Git cho phép bạn đặt nhánh riêng của mình từ một dự án trên đó chỉ cần sử dụng chuột.

Sau khi tôi fetch một cây (tree), tôi chạy lệnh Git để di chuyển và xem xét các thay đổi, với ý tưởng là để tổ chức và mô tả tốt hơn. Tôi trộn với các thay đổi của chính mình, và có thể sẽ sửa thêm chút ít. Sau khi đã hài lòng, tôi push nó lên kho chứa chính.

Mặc dù tôi ít nhận được sự cộng tác, nhưng tôi tin rằng việc này sẽ thay đổi theo chiều hướng tốt lên. Hãy đọc blog của Linus Torvalds [http://torvalds-family.blogspot.com/2009/06/happiness-is-warm-scm.html].

Git thuận lợi trong việc tạo các miếng vá, cũng như là nó tiết kiệm công sức cho chúng ta trong việc chuyển đổi chúng thành những lần commit dành cho Git. Hơn thế nữa, Git lưu giữ các thông tin rất chi tiết như là ghi lại tên và địa chỉ thư điện tử của tác giả, cũng như là ngày tháng và thời gian, và nó cũng đòi hỏi tác giả phải mô tả về những thay đổi mà họ đã tạo ra.

Chương 7. Trở Thành Kiện Tướng

Bây giờ, bạn có thể thông qua lệnh **git help** để bật trang trợ giúp lên và có thể hiểu gần như tất cả mọi thứ. Tuy nhiên, việc xác định chính xác lệnh cần sử dụng để giải quyết các vấn đề thực tế đặt ra có lẽ chẳng dễ dàng gì. Có thể tôi có thể giúp bạn tiết kiệm được thời gian: bên dưới là một vài cách giải quyết các vấn đề thực tế đặt ra mà tôi đã từng sử dụng trong quá khứ.

Phát hành Mã Nguồn

Với dự án của tôi, Git giữ theo dõi chính xác các tập tin tôi muốn lưu trữ và phát hành tới người dùng. Để tạo gói tarball cho mã nguồn, tôi chạy:

```
$ git archive --format=tar --prefix=proj-1.2.3/ HEAD
```

Chỉ Commit Những Gì Thay Đổi

Việc phải thông báo với Git khi bạn thêm, xóa hay đổi tên các tập tin là việc rầy rà với các dự án nào đó. Thay vào đó, bạn có thể gõ:

```
$ git add .
$ git add -u
```

Git sẽ xem tất cả các tập tin trong thư mục hiện tại và làm công việc mà nó phải làm. Thay vì chạy lệnh add thứ hai, hãy chạy `git commit -a` nếu bạn cũng có ý định commit vào lúc này. Xem **git help ignore** để biết làm cách nào để chỉ ra các tập tin bỏ qua.

Bạn có thể thi hành những điều trên chỉ cần một dòng lệnh:

```
$ git ls-files -d -m -o -z | xargs -0 git update-index --add --remove
```

Tùy chọn **-z** và **-0** dùng để ngăn ngừa sai hỏng không mong muốn từ những tập tin có chứa các ký tự đặc biệt. Bởi vì lệnh này bổ xung những tập tin đã bị bỏ qua, bạn có thể muốn sử dụng tùy chọn `-x` hay `-X`.

Lần commit này Nhiều Quá!

Bạn quên việc commit quá lâu? Bạn quá mải mê với việc viết mã nguồn mà quên đi việc quản lý nó? Bạn muốn những thay đổi liên quan đến nhau phải được commit riêng từng lần và nối tiếp nhau, bởi vì đây là phong cách của bạn?

Đừng lo lắng. Chạy:

```
$ git add -p
```

Với mỗi lần thay đổi mà bạn tạo ra, Git sẽ hiện cho bạn biết từng đoạn mã đã bị thay đổi, và hỏi nó có phải là một bộ phận của lần commit tiếp theo. Trả lời là "y" hay "n". Bạn có các sự lựa chọn khác, như là hoãn lại; gõ "?" để biết thêm chi tiết.

Khi nào bạn thỏa mãn thì gõ

```
$ git commit
```

để commit chính xác các thay đổi mà bạn đã chọn lựa (các thay đổi về *staged*). Hãy chắc chắn là bạn đã không dùng tùy chọn **-a**, nếu không thì Git sẽ commit tất cả.

Nhưng bạn lại có những tài liệu đã được chỉnh sửa đặt tại nhiều chỗ khác nhau? Việc duyệt từng cái một sẽ làm bạn nản lòng. Trong trường hợp này, sử dụng lệnh **git add -i**, với giao diện không ít rắc rối hơn, nhưng uyển chuyển hơn. Chỉ cần gõ vài cái, bạn có thể đưa vào hay gỡ bỏ nhiều tập tin vào một trạng thái cùng một lúc, hay xem xét và chọn các thay đổi chỉ trong các tập tin riêng biệt. Có một sự lựa chọn khác, chạy lệnh **git commit --interactive** mà nó sẽ tự động commit sau khi bạn làm xong.

Mục Lục: Vùng trạng thái của Git

Chúng ta trốn tránh chưa muốn nói đến một thứ nổi tiếng của Git đó là *index* (mục lục), nhưng chúng ta phải đối mặt với nó để mà giảng giải những điều ở trên. Chỉ mục là vùng trạng thái tạm thời. Git ít khi di chuyển dữ liệu qua lại một cách trực tiếp giữa dự án của bạn và lịch sử của nó. Đúng hơn là Git đầu tiên ghi dữ liệu vào mục lục, và sau đó sao chép dữ liệu trong chỉ mục vào chỗ cần ghi cuối.

Ví dụ, lệnh **commit -a** thực sự bao gồm hai quá trình. Bước thứ nhất là đặt toàn bộ dữ liệu hiện tại của mọi tập tin cần theo dõi vào bảng mục lục. Bước thứ hai là ghi vào bảng mục lục. Việc commit không sử dụng tùy chọn **-a** chỉ thi hành bước thứ hai, và nó chỉ có ý nghĩa sau khi chạy lệnh mà lệnh này bằng cách này hay cách khác thay đổi bảng chỉ mục, như là lệnh **git add** chẳng hạn.

Thường thường chúng ta bỏ qua mục lục và lấy cớ là chúng ta đang đọc trực tiếp và ghi thẳng vào trong lịch sử. Vì lý do này, chúng ta muốn việc điều khiển chính xác, như vậy chúng ta chỉnh sửa mục lục bằng cách thủ công. Chúng ta đặt một dữ liệu hiện hành của một số, không phải tất cả, các thay đổi của chúng ta vào bảng mục lục, và sau đó ghi những cái này vào lịch sử.

Đừng Quên HEAD Của Mình

Thẻ HEAD giống như một con trỏ, nó trỏ đến lần commit cuối cùng, tự động di chuyển theo mỗi lần commit mới. Một số lệnh của Git giúp bạn di chuyển nó. Ví dụ như:

```
$ git reset HEAD~3
```

sẽ chuyển HEAD lên vị trí lần commit cách đây ba lần. Thế thì tất cả các lệnh Git thi hành như khi bạn ở vị trí commit này, trong khi các tập tin của bạn vẫn nằm ở hiện tại. Xem thêm phần trợ giúp cho một số ứng dụng.

Nhưng ta lại muốn quay trở lại phần sau này? Lần commit cũ không biết gì về phần sau này cả.

Nếu bạn có giá trị băm SHA1 của HEAD gốc thì:

```
$ git reset 1b6d
```

Nhưng giả sử bạn không có được nó? Đừng lo: với những lệnh như thế, Git ghi lại HEAD gốc với thẻ có tên là ORIG_HEAD, và bạn có thể trở về ngon lành và an toàn với lệnh:

```
$ git reset ORIG_HEAD
```

Tìm HEAD

Có thể ORIG_HEAD là chưa đủ. Có lẽ bạn vừa nhận thấy mình vừa tạo ra một sai sót có quy mô lớn và bạn cần phải quay lại một lần commit cách đây lâu lắm rồi trong một nhánh mà bạn đã quên rồi vì nó đã quá lâu.

Theo mặc định, Git giữ một lần commit ít nhất là hai tuần lễ, ngay cả khi bạn đã ra lệnh cho Git phá hủy nhánh chứa nó. Sự khó khăn là ở chỗ làm thế nào để tìm được giá trị băm thích hợp. Bạn có thể tìm kiếm tất cả các giá trị băm trong .git/objects và sử dụng phương pháp thử sai tất cả các giá trị để có được thứ mình muốn. Nhưng còn có cách dễ dàng hơn.

Git ghi lại mọi giá trị băm của mọi lần commit trong máy tính tại thư mục .git/logs. Thư mục con refs chứa lịch sử của tất cả các hoạt động trên tất cả cách nhánh, trong khi tập tin HEAD giữ tất cả các giá trị băm mà nó từng có được. Phần sau có thể được sử dụng để tìm kiếm giá trị băm của các lần commits trên các nhánh cái mà đã bị cắt đi một cách tình cờ.

Lệnh reflog cung cấp cho chúng ta một giao diện thân thiện dành cho các tập tin log. Bạn có thể thử bằng lệnh:

```
$ git reflog
```

Thay vì phải cắt và dán giá trị băm từ reflog, hãy thử:

```
$ git checkout "@{10 minutes ago}"
```

Hay checkout lần thứ 5 kể từ lần commit cuối viếng thăm thông qua lệnh:

```
$ git checkout "@{5}"
```

Xem chương “Specifying Revisions” (tạm dịch: chỉ định các điểm xét duyệt) từ lệnh **git help rev-parse** để biết thêm chi tiết.

Bạn muốn cấu hình thời gian gia hạn lâu hơn việc xóa bỏ những lần commit. Ví dụ:

```
$ git config gc.pruneexpire "30 days"
```

có nghĩa là việc xóa một lần commit sẽ chỉ thực sự xảy ra khi 30 ngày đã qua và lệnh **git gc** được triệu gọi.

Bạn cũng có thể không cho phép chạy lệnh **git gc** một cách tự động:

```
$ git config gc.auto 0
```

trong trường hợp này những lần commit sẽ chỉ bị xóa bỏ khi bạn chạy lệnh **git gc**.

Xây Dựng trên Git

Tuân thủ theo phong thái UNIX, Git được thiết kế cho phép nó dễ dàng được sử dụng như là một thành phần thực thi bên dưới của các chương trình khác, như là cho giao diện đồ họa GUI và giao diện Web để thay thế cho giao diện dòng lệnh, công cụ quản lý các miếng vá, các công cụ xuất/nhập và chuyển đổi, và những thứ tương tự như thế. Trên thực tế, một số lệnh Git bản chất nó cũng là các kịch bản

đứng trên vai của những người khổng lồ, chính là hệ điều hành. Chỉ cần sửa đổi một chút, bạn có thể bắt Git làm việc phù hợp với sở thích của mình.

Một mẹo nhỏ là sử dụng một tính năng sẵn có trong Git bằng cách gán bí danh cho các lệnh để nó trở nên ngắn gọn hơn như sau:

```
$ git config --global alias.co checkout   # gán bí danh cho lệnh checkout là co
$ git config --global --get-regexp alias  # hiển thị bí danh hiện hành
alias.co checkout
$ git co foo                              # có kết quả giống như chạy lệnh 'git checkout foo'
```

Một thứ khác là hiển thị nhánh hiện hành lên màn hình hay thanh tiêu đề của cửa sổ. Gọi lệnh:

```
$ git symbolic-ref HEAD
```

sẽ hiển thị tên của nhánh hiện hành. Trong thực tiễn, bạn thích hợp nhất muốn gỡ bỏ "refs/heads/" và tránh các lỗi:

```
$ git symbolic-ref HEAD 2> /dev/null | cut -b 12-
```

Thư mục con contrib là một kho báu được tìm thấy trong số các công cụ được xây dựng dành cho Git. Đúng lúc, một số trong số chúng có thể được xúc tiến thành lệnh chính thức. Trên hệ thống Debian và Ubuntu, thư mục này ở tại /usr/share/doc/git-core/contrib.

Một thư mục phổ biến khác là workdir/git-new-workdir. Thông qua các liên kết tài tình, đoạn kịch bản này tạo ra một thư mục làm việc mới trong khi phần lịch sử thì chia sẻ với kho chứa nguyên gốc:

```
$ git-new-workdir an/existing/repo new/directory
```

Thư mục mới và các tập tin trong nó có thể được coi như một bản sao, ngoại trừ phần lịch sử được chia sẻ dùng chung, hai cây được tự động đồng bộ hóa. Ở đây không cần có sự trộn, push, hay pull.

Cứ Phiêu Lưu

Ngày nay, người sử dụng Git rất khó phá hủy dữ liệu. Nhưng nếu như bạn biết mình đang làm gì, bạn có thể vượt qua sự bảo vệ dành cho các lệnh thông thường đó.

Checkout: Không commit các thay đổi là nguyên nhân của việc checkout gặp lỗi. Để phá hủy sự thay đổi của mình, và dẫu sao cũng checkout commit đã cho, sử dụng cờ bắt buộc (force):

```
$ git checkout -f HEAD^
```

Mặt khác, nếu bạn chỉ định rõ một đường dẫn chi tiết cho lệnh, thế thì ở đây không có sự kiểm tra an toàn nào cả. Đường dẫn được áp dụng sẽ bị âm thầm ghi đè lên. Hãy cẩn thận nếu bạn sử dụng lệnh checkout theo cách này.

Reset: Lệnh reset cũng xảy ra lỗi khi không commit các thay đổi. Để bắt buộc nó, chạy:

```
$ git reset --hard 1b6d
```

Branch: Việc xóa các nhánh cũng gặp lỗi nếu đó là nguyên nhân khiến các thay đổi bị mất. Để ép buộc việc này, hãy gõ:

```
$ git branch -D dead_branch  #thay vì sử dụng tùy chọn -d
```

Cũng tương tự như thế, việc cố gắng ghi đè lên một nhánh bằng cách di chuyển nhánh khác đến nó cũng gây ra lỗi. Để ép buộc sự di chuyển nhánh, gõ:

```
$ git branch -M source target  # thay vì sử dụng tùy chọn -m
```

Không giống như checkout và reset, hai lệnh trên trì hoãn việc phá hủy dữ liệu. Các thay đổi vẫn còn lưu giữ trong thư mục con .git, và có thể lấy lại được bằng cách lấy giá trị băm .git/logs thích hợp (xem phần "Tìm - HEAD" ở phía trên). Theo mặc định, chúng sẽ giữ ít nhất là hai tuần lễ.

Clean: Một số lệnh Git từ chối thi hành bởi vì chúng lo lắng về việc làm như thế làm mất dấu hoàn toàn các tập tin. Nếu bạn chắc chắn về tất cả các tập tin và thư mục không cần Git theo dõi nữa và muốn phá hủy chúng đi, thế thì xóa chúng triệt để với lệnh:

```
$ git clean -f -d
```

Sau này, lệnh rầy rà đó sẽ hoạt động!

Ngăn Ngừa Commit Sai

Có một số lỗi ngớ ngẩn đã xảy ra với tôi. Điều tồi tệ nhất là để sót các tập tin bởi vì quên lệnh **git add**. Ít tệ hại hơn là các ký tự khoảng trắng và những xung đột không cần phải trộn: mặc dù cũng chẳng tệ hại lắm, tôi mong rằng những điều này sẽ không xảy ra với mọi người.

Tôi đã tránh được các lỗi ngu ngốc đó bằng cách sử dụng một *hook* để nó cảnh báo người dùng khi có những vấn đề:

```
$ cd .git/hooks
$ cp pre-commit.sample pre-commit  # Với phiên bản Git cũ cần chạy lệnh: chmod +x pre-commi
```

Ngày nay Git sẽ không commit nếu khi nó trộn nó chỉ tìm thấy những khoảng trắng vô ích hay những xung đột không cần giải trộn.

Với bản hướng dẫn này, tôi cuối cùng đã thêm vào dòng đầu của hook **pre-commit** để đề phòng khi ta lơ đãng:

```
if git ls-files -o | grep '\.txt$'; then
  echo FAIL! Untracked .txt files.
  exit 1
fi
```

Nhiều hoạt động của Git hỗ trợ hook; hãy xem **git help hooks**. Chúng tôi đã kích hoạt một hook mẫu là **post-update** trước khi nói đến Git thông qua HTTP. Cái này chạy mỗi khi head di chuyển. Đoạn script ví dụ post-update cập nhật các tập tin Git cần cho việc truyền thông thông qua Git-agnostic chuyên chở bằng giao thức giống như là HTTP.

Chương 8. Bí Quyết của Git

Chúng ta mổ xẻ để hiểu được làm thế nào mà Git có thể thi hành kỳ diệu như vậy. Tôi sẽ không thể nói quá chi tiết được. Nếu bạn muốn có được sự mô tả chỉ tiết thì hãy đọc sổ tay hướng dẫn sử dụng Git [http://schacon.github.com/git/user-manual.html].

Tính Ẩn

Git làm việc có vẻ kín đáo? Chỉ cần nói riêng về việc sử dụng lệnh **commit** và **merge**, bạn có thể làm việc mà không cần biết đến sự tồn tại của hệ thống quản lý mã nguồn. Cho đến khi bạn cần nó, và cho đến khi bạn vui sướng vì Git đã trông coi mã nguồn cho bạn trong suốt thời gian qua.

Các hệ thống quản lý mã nguồn khác ép buộc bạn luôn luôn phải tranh đấu với thói quan liêu. Quyền truy cập của các tập tin có thể là chỉ cho phép đọc trừ phi bạn nói rõ với máy chủ trung tâm là các tập tin nào bạn muốn chỉnh sửa. Tốc độ làm việc của phần lớn các lệnh sẽ tỷ lệ nghịch với số lượng người sử dụng. Mọi công việc sẽ đình trệ khi mạng máy tính hay máy chủ ngừng hoạt động.

Đối lập với hạn chế trên, Git đơn giản giữ lịch sử của dự án của bạn tại thư mục `.git` trong thư mục làm việc của bạn. Đây là bản sao lịch sử của riêng bạn, do vậy bạn có thể làm việc không cần mạng cho đến khi cần truyền thông với những người khác. Bạn có toàn quyền quyết định với các tập tin của mình bởi vì Git có thể tạo lại trạng thái đã ghi lại từ `.git` bất kỳ lúc nào một cách dễ dàng.

Toàn Vẹn Dữ Liệu

Phần lớn mọi người sử dụng phương pháp mã hóa để giữ cho thông tin của mình không bị nhòm ngó, nhưng có thứ quan trọng không kém đó là giữ cho thông tin của mình được toàn vẹn. Chính việc sử dụng hàm băm mã hóa đã làm ngăn ngừa sự sai hỏng dữ liệu do rủi ro hay ác ý.

Giá trị SHA1 có thể coi như là một số định danh 160-bit không trùng lắp cho mỗi chuỗi ký tự bạn dùng trong đời sống của mình. Trên thực tế nó còn làm được nhiều hơn thế: nó có thể thực hiện điều trên với mọi chuỗi ký tự mà mọi người có thể sử dụng trong suốt cuộc đời của mình.

Bản thân giá trị SHA1 cũng là một chuỗi ký tự, chúng ta có thể băm chuỗi có chứa giá trị băm khác. Khả năng quan sát đơn giản này cực kỳ hữu dụng: tra cứu *hash chains* (tra cứu theo các chuỗi móc xích với nhau bằng giá trị băm). Sau này chúng ta sẽ thấy làm cách nào Git sử dụng nó để mà đảm bảo tính toàn vẹn của dữ liệu.

Tóm lại, Git lưu giữ dữ liệu của bạn trong thư mục con `.git/objects`, thay vì sử dụng tên tập tin như thông thường, bạn sẽ chỉ nhìn thấy ID của chúng. Bằng cách sử dụng ID để làm tên tập tin, cũng tốt như là cách sử dụng kỹ thuật lockfiles (khóa tập tin) và timestamp (theo dõi thời gian của tập tin), Git biến hệ thống tập tin thông thường bất kỳ nào trở thành một cơ sở dữ liệu hiệu quả và mạnh mẽ.

Thông Minh

Làm thể nào mà Git biết bạn đã đổi tên một tập tin, dù là bạn chẳng bao giờ đề cập đến điều này một cách rõ ràng? Chắc chắn rồi, bạn có lẽ đã chạy lệnh **git mv**, nhưng nó chính xác giống hệt như việc chạy lệnh **git rm** sau đó là lệnh **git add**.

Git khám phá ra cách truy tìm các tập tin đã được đổi tên hay sao chép giữa các phiên bản liên tiếp. Trên thực tế, nó có thể tìm ra từng đoạn mã nguồn đã bị di chuyển hay sao chép giữa các tập tin! Dẫu cho nó không thể xử lý được mọi trường hợp, nó làm khá tốt, và đặc tính này luôn luôn được phát triển. Nếu nó không làm việc với bạn, hãy thử bật các tùy chọn dành cho việc phát hiện sự sao chép, và nên cất nhắc đến việc cập nhật.

Mục Lục

Với mọi tập tin được theo dõi, Git ghi lại các thông tin như là kích thước, thời gian tạo và lần cuối sửa đổi trong một tập tin được biết đến là một mục lục *index*. Để xác định rõ một tập tin có bị thay đổi hay không, Git so sánh nó ở thời điểm hiện tại với phần lưu giữ trong bộ nhớ. Nếu chúng giống nhau, thế thì Git có thể bỏ qua việc đọc tập tin đó lại lần nữa.

Bởi vì gọi lệnh “stat” nhanh hơn đáng kể so với đọc tập tin, nếu bạn chỉ chỉnh sửa vài tập tin, Git có thể cập nhật trạng thái của nó cực kỳ nhanh chóng.

Chúng ta đã nói trước rằng mục lục (index) là vùng làm việc của trạng thái. Tại sao lại là một chùm tập tin stat vùng stage? Bởi vì lệnh add đặt các tập tin vào trong cơ sở dữ liệu của Git và cập nhật những stat này, trong lúc lệnh commit được thực hiện, mà không có tùy chọn, tạo ra một commit trên cơ sở chỉ trên các stat và các tập tin đã sẵn có trong cơ sở dữ liệu.

Nguồn Gốc của Git

Linux Kernel Mailing List post [http://lkml.org/lkml/2005/4/6/121] này miêu tả các sự kiện nối tiếp nhau về Git. Toàn bộ tuyến này chỉ dành cho các sử gia đam mê Git.

Đối tượng Cơ Sở Dữ Liệu

Mỗi một phiên bản của dữ liệu của bạn được giữ trong *đối tượng cơ sở dữ liệu* (object database), mà nó nằm trong thư mục con .git/objects; cái khác nằm trong thư mục .git/ lưu giữ ít dữ liệu hơn: mục lục, tên các nhánh, các thẻ tag, các tùy chọn cấu hình, nhật ký, vị trí hiện tại của head của lần commit, và những thứ tương tự như thế. Đối tượng cơ sở dữ liệu cho đến bây giờ vẫn là phần tử cơ bản xuất sắc nhất, và là cội nguồn sức mạnh của Git.

Mỗi tập tin trong .git/objects là một *đối tượng*. Ở đây có 3 loại đối tượng liên quan đến chúng ta: đối tượng nội dung tập tin *blob*, đối tượng cây *tree*, và đối tượng lần chuyển giao *commit*.

Đối Tượng Blob

Đầu tiên, hãy tạo một tập tin bất kỳ. Đặt cho nó một cái tên, tên gì cũng được. Trong một thư mục rỗng:

```
$ echo sweet > YOUR_FILENAME
$ git init
$ git add .
$ find .git/objects -type f
```

Bạn sẽ thấy .git/objects/aa/823728ea7d592acc69b36875a482cdf3fd5c8d.

Làm sao mà tôi biết được tập tin khi không thấy tên của nó? Đó là bởi vì đó là giá trị băm SHA1 của:

"blob" SP "6" NUL "sweet" LF

là aa823728ea7d592acc69b36875a482cdf3fd5c8d, với SP là khoảng trắng, NUL là byte có giá trị bằng 0 và LF ký tự xuống dòng. Bạn có thể xác minh lại bằng lệnh sau:

$ printf "blob 6\000sweet\n" | sha1sum

Git sử dụng cách *lấy nội dung để làm tên cho tập tin*: tập tin không được lưu trữ như theo tên của chúng, mà bằng giá trị băm dữ liệu mà chúng chứa, trong tập tin chúng ta gọi là một *đối tượng blob*. Chúng ta có thể nghĩ giá trị băm như là một định danh duy nhất cho nội dung của tập tin, do vậy ta có tên tập tin được định danh bởi nội dung của nó. Phần khởi đầu blob 6 đơn thuần chỉ là phần đầu để thể hiện kiểu của đối tượng và độ dài của nó tính bằng byte; việc làm này làm đơn giản hóa việc vận hành bên trong Git.

Đến đây tôi có thể dễ dàng đoán được bạn nghĩ gì. Tên của tập tin là không thích hợp: chỉ khi có dữ liệu bên trong được sử dụng để xây dựng nên đối tượng blob.

Bạn có lẽ sẽ kinh ngạc với những gì xảy ra với các tập tin có cùng nội dung. Hãy thử thêm một bản sao một tập tin nào đó của bạn, với bất kỳ một cái tên nào cũng được. Nội dung của .git/objects ở tại cùng một chỗ cho dù bạn thêm vào bao nhiêu lần đi chăng nữa. Git chỉ lưu giữ dữ liệu một lần duy nhất.

Nhưng dẫu sao, các tập tin nằm trong .git/objects đã bị nén lại theo chuẩn zlib do vậy bạn không thể xem chúng một cách trực tiếp được. Đọc chúng thông qua zpipe -d [http://www.zlib.net/zpipe.c], hay gõ:

$ git cat-file -p aa823728ea7d592acc69b36875a482cdf3fd5c8d

lệnh này trình bày đối tượng được cho ở dạng dễ đọc trên màn hình.

Đối Tượng Tree

Nhưng mà tên tập tin ở đâu? Chúng phải được lưu giữ ở đâu đó chứ. Git lấy tên tập tin trong quá trình commit:

$ git commit # Gõ chú thích.
$ find .git/objects -type f

Bạn sẽ thấy ba đối tượng. Ở thời điểm này tôi không thể nói hai tập tin mới này là cái gì, hãy nghĩ nó là một phần của tên tập tin bạn đang xét. Chúng ta sẽ xuất phát từ giả định bạn chọn “rose”. Nếu bạn không làm thế, bạn có thể viết lại lịch sử để làm cho nó giống như bạn đã làm thế:

$ git filter-branch --tree-filter 'mv YOUR_FILENAME rose'
$ find .git/objects -type f

Bây giờ bạn có lẽ nhìn thấy tập tin .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9, bởi vì đây là giá trị băm SHA1 của nội dung của nó:

"tree" SP "32" NUL "100644 rose" NUL 0xaa823728ea7d592acc69b36875a482cdf3fd5c8d

Xác thực tập tin này chứa thông tin như trên bằng cách gõ:

$ echo 05b217bb859794d08bb9e4f7f04cbda4b207fbe9 | git cat-file --batch

Với lệnh zpipe, ta có thể dễ dàng xác thực một giá trị băm:

$ zpipe -d < .git/objects/05/b217bb859794d08bb9e4f7f04cbda4b207fbe9 | sha1sum

Sự thẩm tra giá trị băm thì rắc rối hơn thông qua lệnh cat-file bởi vì phần kết xuất của nó chứa nhiều thông tin hơn đối tượng tập tin thường không bị nén.

Tập tin này là một đối tượng cây *tree*: một danh sách các hàng bao gồm kiểu tập tin, tên tập tin, và giá trị băm. Trong ví dụ của chúng ta, kiểu tập tin là 100644, điều này có nghĩa ‘rose` là tập tin bình thường, và giá trị băm là một đối tượng blob mà nó chứa nội dung của `rose’. Dạng tập tin khác có thể là tập tin thực thi, liên kết mềm hay các thư mục. Trong trường hợp cuối, giá trị băm sẽ chỉ đến đối tượng cây *tree*.

Nếu bạn đã chạy lệnh filter-branch, bạn sẽ có các đối tượng cũ bạn không cần đến sau này nữa. Mặc dù chúng sẽ bị loại bỏ một cách tự động một khi thời hạn chấm dứt đã đến, nhưng chúng ta sẽ xóa chúng ngay bây giờ theo cách dưới đây:

```
$ rm -r .git/refs/original
$ git reflog expire --expire=now --all
$ git prune
```

Với các dự án thật bạn nên tránh việc sử dụng lệnh như trên, làm như thế bạn đã phá hủy dữ liệu sao lưu dự phòng. Nếu bạn muốn làm sạch kho chứa, cách hay nhất là tạo một bản sao mới. Cũng thế, hãy cẩn thận khi thao tác trực tiếp với thư mục .git: điều gì xảy ra nếu một lệnh Git khác cũng đang thực thi cùng lúc, hay là mất điện đột ngột? Đại khái, refs có thể được xóa bằng lệnh **git update-ref -d**, mặc dù thường thường nó an toàn hơn xóa refs/original bằng tay.

Commit

Chúng tôi đã giảng giải cho bạn 2 trong số 3 đối tượng của Git. Cái thứ 3 chính là **commit**. Nội dung của nó buộc chặt vào phần chú thích của lần commit cũng như thời gian, ngày tháng chúng được tạo ra. Để cho khớp với những thứ chúng ta có ở đây, chúng ta sẽ phải chỉnh nó một chút:

```
$ git commit --amend -m Shakespeare  # Thay đổi phần chú thích.
$ git filter-branch --env-filter 'export
   GIT_AUTHOR_DATE="Fri 13 Feb 2009 15:31:30 -0800"
   GIT_AUTHOR_NAME="Alice"
   GIT_AUTHOR_EMAIL="alice@example.com"
   GIT_COMMITTER_DATE="Fri, 13 Feb 2009 15:31:30 -0800"
   GIT_COMMITTER_NAME="Bob"
   GIT_COMMITTER_EMAIL="bob@example.com"'  # dấu vết thời gian và tác giả đã bị gian lận.
$ find .git/objects -type f
```

Bạn có thể thấy .git/objects/49/993fe130c4b3bf24857a15d7969c396b7bc187 là giá trị băm SHA1 của nội dung của nó:

"commit 158" NUL

```
"tree 05b217bb859794d08bb9e4f7f04cbda4b207fbe9" LF
"author Alice <alice@example.com> 1234567890 -0800" LF
"committer Bob <bob@example.com> 1234567890 -0800" LF
LF
"Shakespeare" LF
```

Như ở phần trước, bạn có thể chạy lệnh zpipe hay cat-file để tự mình trông thấy.

Đây là lần commit đầu tiên, do vậy lần commit này không có cha, nhưng những lần commit sau sẽ luôn luôn chứa it nhất là một dòng chỉ định commit cha.

Khó Phân Biệt Được sự Thần Kỳ

Bí mật của Git dường như có vẻ đơn giản. Nó giống như bạn có thể trộn lẫn cùng nhau một ít kịch bản và một ít mã C mà đun trong vài giờ: nhào trộn của quá trình hoạt động của hệ thống tập tin và mã băm SHA1, bày biện thêm với các khóa và đồng bộ hóa tập tin để tăng vị ngon. Trên thực tế, những mô tả như thế với Git các phiên bản trước kia là chính xác. Tuy nhiên, ngoài chiêu bài đóng gói để tiết kiệm không gian lưu trữ, sử dụng mục lục để tiết kiệm thời gian ra, giờ chúng ta còn biết thêm làm cách nào Git khéo léo thay đổi hệ thống tập tin thành một cơ sở dữ liệu hoàn hảo cho việc quản lý mã nguồn.

Ví dụ, nếu một tập tin bất kỳ nào đó trong đối tượng cơ sở dữ liệu bị sai hỏng bởi lỗi do ổ đĩa, thế thì giá trị băm tương ứng của nó sẽ không đúng nữa, điều này sẽ mang lại rắc rối cho chúng ta. Bằng việc băm giá trị băm của đối tượng khác, chúng ta có thể duy trì tính toàn vẹn ở tất cả các mức. Commit là hạt nhân, thật vậy đấy, mỗi lần commit không bao giờ ghi lại nửa vời: chúng ta có thể chỉ tính toán mã băm của lần commit và lưu giữ giá trị của nó trong cơ sở dữ liệu sau khi chúng ta đã sẵn sàng lưu tất cả các đối tượng là trees, blobs và cha của các lần commit thích hợp. Đối tượng cơ sở dữ liệu không bị ảnh hưởng bởi các sự cố ngắt quãng bất ngờ như là mất điện đột ngột chẳng hạn.

Chúng ta có thể làm thất bại ngay cả những kẻ phá hoại ranh mãnh. Giả sử người nào đó lén lút sửa chữa nội dung của một tập tin trong một phiên bản cũ của dự án. Để giữ đối tượng cơ sở dữ liệu vẫn hoạt động tốt, họ đồng thời cũng phải thay đổi giá trị băm của đối tượng blob tương ứng vì lẽ rằng nó bây giờ đã khác trước. Điều đó có nghĩa là họ sẽ phải thay đổi giá trị băm của một đối tượng tree có liên quan đến tập tin, và việc chỉnh sửa giá trị băm của tất cả các đối tượng commit kéo theo như là tree, thêm nữa là các giá trị băm của toàn bộ các lần commit con cháu của nó. Cái này kéo theo giá trị băm của head tại kho trung tâm không giống với thứ đó tại kho chứa sai hỏng. Bằng cách theo dõi sự tương khớp giá trị băm chúng ta có thể xác định được tập tin bị sai hỏng, cũng như là lần commit nơi mà nó lần đầu bị hư hỏng.

Nói ngắn gọn, dùng 20 byte để đại diện cho lần commit cuối là an toàn, việc cố tình sửa đổi nội dung một kho chứa Git là điều không thể thực hiện được.

Đặc tính nào của Git là trứ danh nhất? Nhánh? Trộn? Hay Tags? Chỉ là chi tiết. Head hiện hành được giữ trong tập tin .git/HEAD, mà nó có chứa giá trị băm của một đối tượng commit. Giá trị băm được cập nhật trong quá trình commit cũng như là một số lệnh khác. Nhánh đại thể cũng tương tự: chúng là các tập tin trong thư mục .git/refs/heads. Tags cũng thế: chúng ở tại .git/refs/tags nhưng chúng được cập nhật bởi các lệnh khác nhau.

Chương 9. Phụ lục A: Hạn chế của Git

Git bộc lộ một số nhược điểm mà tôi đã gặp qua. Một số có thể xử lý thủ công một cách dễ dàng bằng các đoạn kịch bản và hook, một số yêu cầu phải tổ chức lại hay xác lập lại dự án, một số ít rắc rối còn lại, chỉ còn cách là ngồi đợi. Hay tốt hơn cả là bắt tay vào và giúp đỡ họ viết!

Điểm Yếu SHA1

Thời gian trôi đi, những nhà mật mã đã phát hiện ra ngày càng nhiều điểm yếu của thuật toán SHA1. Thực tế người ta đã đã phát hiện thấy sự va chạm giá trị băm. Trong khoảng vài năm, có lẽ những chiếc PC thông thường cũng đủ sức để âm thầm làm hư hỏng một kho Git.

Hy vọng là Git sẽ chuyển sang sử dụng hàm băm tốt hơn trước khi có người tìm ra cách phá mã SHA1.

Microsoft Windows

Sử dụng Git trên hệ điều hành Microsoft Windows có vẻ hơi cồng kềnh một chút:

- Cygwin [http://cygwin.com/], mô phỏng Linux dành cho Windows, có chứa Git đã chuyển đổi để chạy trên Windows [http://cygwin.com/packages/git/].
- Git chạy trên MSys [http://code.google.com/p/msysgit/] là một thay thế với các hỗ trợ tối thiểu nhất, bởi vì chỉ cần một ít lệnh để thực hiện một số việc mà thôi.

Các Tập tin Không liên quan

Nếu dự án của bạn rất lớn và chứa rất nhiều tập tin không có liên quan mà luôn luôn bị thay đổi, Git có thể chịu thiệt thòi hơn các hệ thống khác bởi vì các tập tin không được giữ dấu viết từng cái riêng lẻ. Git giữ các dấu vết thay đổi cho toàn bộ dự án, điều này thường là có lợi.

Giải pháp là chia nhỏ dự án của bạn ra, mỗi một phần bao gồm các tập tin liên quan đến nhau. Hãy sử dụng **git submodule** nếu bạn vẫn muốn giữ mọi thứ trong một kho chung.

Ai Sửa và Sửa gì?

Một số hệ thống quản lý mã nguồn bắt buộc bạn đánh dấu rõ ràng vào tập tin theo một cách nào đó trước khi biên soạn. Trong khi mà điều này đặc biệt phiền toái vì nó lại dính líu đến việc phải liên lạc với máy chủ trung tâm, việc làm này có hai lợi ích:

1. Thực hiện lệnh *diff* diễn ra nhanh bởi vì nó chỉ kiểm tra các tập tin đã đánh dấu.
2. Một người có thể biết được khác đang làm việc trên một tập tin bằng cách hỏi máy chủ trung tâm ai đã đánh dấu là đang sửa.

Với một đoạn kịch bản thích hợp, bạn có thể lưu giữ theo cách này với. Điều này yêu cầu sự hợp tác từ người lập trình, người có thể chạy các kịch bản chuyên biệt khi biên soạn một tập tin.

Lịch Sử Tập Tin

Sau khi Git ghi lại các thay đổi cho các dự án lớn, việc cấu trúc lại lịch sử của một tập tin đơn lẻ yêu cầu phải làm việc nhiều hơn các chương trình quản lý mã nguồn giữ dấu vết theo các tập tin riêng lẻ.

Thiệt hại thường là không đáng kể, và thứ đáng giá mà nó nhận được là các tác vụ khác hoạt động hiệu quả đến không ngờ. Ví dụ, `git checkout` nhanh hơn `cp -a`, và dữ liệu trong dự án lớn nén tốt hơn việc gom lại từ tập tin cơ bản.

Khởi tạo Bản Sao

Việc tạo một bản sao có vẻ hơi xa xỉ hơn là việc *checkout* trong các hệ thống quản lý mã nguồn khác khi phần mềm có lịch sử phát triển lâu dài.

Cái giá phải trả ban đầu là cần nhiều thời gian để lấy về, nhưng nếu đã làm như thế, các tác vụ cần làm sau này sẽ nhanh chóng và không cần có mạng. Tuy nhiên, trong một số hoàn cảnh, cách làm phù hợp hơn là tạo một bản sao không đầy đủ bằng tùy chọn `--depth`. Điều này giúp ta tạo bản sao nhanh hơn, nhưng bản sao nhận được sẽ thiếu đi một số chức năng do đó bạn sẽ không thể thực thi được một số lệnh.

Các Dự Án Hay Thay Đổi

Git được viết ra với mục đích chú tâm đến kích thước tạo ra bởi các thay đổi. Con người chỉ tạo ra sự thay đổi rất nhỏ giữa các phiên bản. Như là bổ xung lời nhận xét là có sửa lỗi ở đây, có đặc tính mới ở đây, sửa lỗi chú thích, v.v.. Nhưng nếu các tập tin của bạn căn bản khác nhau, thì trong mỗi lần commit, nó sẽ ghi lại toàn bộ các thay đổi vào lịch sử và làm cho dự án của bạn tất yếu sẽ tăng kích cỡ.

Không có bất kỳ một hệ thống quản lý mã nguồn nào có thể làm được điều này, nhưng những người sử dụng Git theo tiêu chuẩn sẽ còn phải chịu tổn thất hơn khi lịch sử của nó được nhân bản.

Đây là lý do tại sao các thay đổi quá lớn cần được xem xét. Định dạng các tập tin có thể bị thay đổi. Các thay đổi nhỏ chỉ xảy ra phần lớn tại một số ít tập tin.

Việc xét đến việc sử dụng cơ sở dữ liệu hay các giải pháp sao-lưu/lưu-trữ có lẽ là thứ có vẻ thực tế hơn, không nên dùng hệ thống quản lý mã nguồn. Ví dụ, quản lý mã nguồn không thích hợp cho việc quản lý các ảnh được chụp một cách định kỳ từ webcam.

Nếu các tập tin thực sự thay đổi thường xuyên và chúng cần phải quản lý, việc xem xét khả năng sử dụng Git hoạt động như một hệ thống quản lý tập trung là có thể chấp nhận được. Một người có thể tải về một bản sao không đầy đủ, nó chỉ lấy về một ít hay không lấy về lịch sử của dự án. Dĩ nhiên, nhiều công cụ dành cho Git sẽ không thể hoạt động được, và sự sửa chữa phải được chuyển lên như là các miếng vá. Điều này chắc chắn là tốt và nó giống như là ta không thể hiểu nổi tại sao một số người lại muốn có được lịch sử của rất nhiều các tập tin chẳng hoạt động ổn định.

Một ví dụ khác là dự án phụ thuộc vào firmware, cái này có dạng thức là tập tin nhị phân có kích thước rất lớn. Người sử dụng không quan tâm tới lịch sử của firmware, vả lại khả năng nén của nó lại cũng rất ít, vì vậy quản lý firmware có lẽ là không cần thiết vì nó làm phình to kích thước kho chứa.

Trong trường hợp này, mã nguồn có thể lưu giữ trong kho Git, và tập tin nhị phân được giữ ở nơi khác. Để cho công việc trở nên dễ dàng hơn, một người có thể tạo ra một đoạn kịch bản mà nó sử dụng Git để nhân bản mã nguồn, và dùng lệnh rsync hay Git lấy về firmware.

Bộ Đếm

Một số hệ quản trị mã nguồn tập trung duy trì một số nguyên dương tự động tăng lên khi có lần commit mới được chấp nhận. Git quy các thay đổi này bởi giá trị băm của chúng, điều này là tốt trong phần lớn hoàn cảnh.

Nhưng một số người thích có nó ở dạng số nguyên. May mắn thay, rất dễ dàng để viết các đoạn kịch bản làm được như thế với mỗi lần cập nhật, kho Git trung tâm Git gia một số nguyên, có thể là trong một thẻ (tag), và kết hợp nó với giá trị băm của lần commit cuối.

Mỗi bản sao có thể có một bộ đếm riêng, nhưng điều này chẳng ích lợi gì, chỉ có kho chứa trung tâm và bộ đếm của nó là có ý nghĩa với mọi người.

Với Thư Mục Rỗng

Các thư mục rỗng không được theo dõi. Tạo ra các thư mục giả để thử trục trặc này.

Xét về mặt thi hành của Git, thay vì thiết kế của nó, điều hạn chế này này là đáng trách. Với một chút may mắn, một khi Git thấy có thêm lợi ích từ việc này, thêm nhiều người đòi hỏi tính năng này và nó sẽ được thực hiện.

Lần Commit Khởi tạo

Hệ thống số đếm khoa học của máy tính đếm từ 0, thay vì 1. Thật không may, có liên quan đến các lần commit, Git không tôn trọng quy ước này. Rất nhiều lệnh bất lợi trước lần commit khởi tạo. Thêm nữa, các trường hợp ngoại lệ phải được xử lý theo một cách đặc biệt, như là việc rebasing một nhánh với lần commit khởi tạo khác.

Git có thể có được lợi ích từ việc định nghĩa lần commit zero: ngay khi kho chứa được tạo ra, HEAD được đặt cho một chuỗi ký tự bao gồm 20 byte rỗng. Lần commit đặc biệt này tương ứng với một cây (tree) rỗng, không có gốc, tại một thời điểm được đề lùi về trước.

Sau đó chạy lệnh git log, ví dụ thế, thì Git nên báo cho người dùng biết chưa có lần commit nào, thay vì phát ra một lỗi nghiêm trọng. Điều tương tự xảy ra với các công cụ khác.

Tất cả các bản commit đầu tiên hoàn toàn là con cháu của bản 0 (zero).

Tuy nhiên, ở đây có một số vấn đề xảy ra trong một số trường hợp đặc biệt. Nếu nhiều nhánh với các lần khởi tạo commit khác nhau được trộn với nhau, sau đó rebase kết quả đòi hỏi thực chất có sự can thiệp bằng tay.

Giao diện chưa rõ ràng

Để commit A và B, nghĩa của biểu thức "A..B" và "A...B" tùy thuộc vào việc lệnh mong đó là hai đầu mút hay là một vùng. Xem **git help diff** và **git help rev-parse**.

Chương 10. Phụ lục B: Dịch cuốn sách này

Tôi khuyến nghị các bạn làm theo các bước sau để thực hiện việc dịch thuật, làm như vậy thì các đoạn kịch bản của tôi có thể nhanh chóng tạo ra các tài liệu ở định dạng HTML và PDF, và tất cả các bản dịch có thể ở trong cùng một kho chứa.

Lấy một bản sao của mã nguồn, sau đó tạo một thư mục tương ứng với ngôn ngữ bạn dịch language's IETF tag [http://www.iana.org/assignments/language-subtag-registry]: xem tại the W3C article on internationalization [http://www.w3.org/International/articles/language-tags/Overview.en.php]. Ví dụ, tiếng Anh là "en", Nhật là "ja", tiếng Trung Quốc Phồn thể là "zh-Hant". Trong thư mục mới đó, và dịch tập tin txt từ thư mục con "en".

Một ví dụ cụ thể là để dịch phần hướng dẫn này thành ngôn ngữ Klingon [http://en.wikipedia.org/wiki/Klingon_language], bạn hãy gõ vào:

```
$ git clone git://repo.or.cz/gitmagic.git
$ cd gitmagic
$ mkdir tlh  # "tlh" là mã ngôn ngữ IETF dành cho Klingon.
$ cd tlh
$ cp ../en/intro.txt .
$ edit intro.txt  # Dịch tập tin này.
```

và cứ như thế cho những tập tin còn lại.

Chỉnh sửa lại Makefile và thêm mã ngôn ngữ cho biến TRANSLATIONS. Bạn có thể xem thử kết quả công việc của mình như sau:

```
$ make tlh
$ firefox book-tlh/index.html
```

Hãy chuyển giao công việc của bạn thường xuyên, và cho tôi biết khi nào thì chúng sẵn sàng để sử dụng. GitHub.com có giao diện thuận lợi như sau: rẽ nhánh dự án "gitmagic", *push* các thay đổi của bạn lên, sau đó thông báo với tôi để tôi trộn.