В нашей песочнице можно использовать множество команд:

commit

branch

checkout

cherry-pick

reset

revert

rebase

merge

**1.**

Một commit trong một git repository(kho chứa) lưu trữ một ảnh chụp của tất cả các file trong thư mục của bạn. Như kiểu copy và paste cõ bự, thậm chí còn tốt hơn thế!

Thế nhưng Git muốn giữ cho các commit nhẹ hết mức có thể, cho nên git không copy tất cả thư mục một cách mù quáng mỗi khi bạn commit. Nó có thể(khi khả dĩ) nén commit như một tập hợp các thay đổi, hay là một "bản so sánh", giũa một phiên bản kho chứa git với phiên bản tiếp theo.

Đồng thời Git cũng lưu trữ lịch sử commit nào được tạo ra lúc nào. Đó là tại sao hầu hết các commit có commit tổ tiên phía trên nó -- chúng tôi đã hình tượng hóa mối quan hệ này bằng các mũi tên. Trong cộng tác nhóm thì việc gìn giữ lịch sử là rất có ích!

Có vẻ là hơi nhiều lý thuyết rồi, bây giờ bạn chỉ cần hiều các commit là các lát cắt của dự án. Các commit rất nhẹ nên việc chuyển qua lại giữa chúng thì nhanh vô cùng!

git commit

**Nhánh trong Git**

Nhánh trong Git cũng nhẹ đến không ngờ. Chúng chỉ đơn giản là các con trỏ đến commit -- không hơn. Đó là lý do các con chiên Git hay niệm chú:

*rẽ nhánh sớm, rẽ nhánh thường xuyên*

Bởi vì chẳng tốn bao nhiêu bộ nhớ cho việc rẽ nhánh cả, và nó dễ dàng phân chia công việc hơn là có một cái nhánh to tổ chảng.

Khi bạn trộn lẫn commit và nhánh, bạn sẽ thấy chúng kết hợp với nhau thế nào. Còn bây giờ, đơn giản hãy nhớ nhánh cơ bản muốn nói "Tôi muốn thành quả trong commit này và tất cả cha ông của nó"

git branch newImage

git checkout newImage; git commit

Tiện thể, có đường tắt đấy: nếu bạn muốn tạo nhánh mới VÀ đồng thời chuyển sang luôn, bạn chỉ cần gõ

git checkout -b [yourbranchname]

**Nhánh và gộp nhánh**

git merge bugFix

git checkout bugFix; git merge master

***\*hint:***

Để hoàn thành cấp độ này, hãy làm theo các bước bên dưới:

* Tạo một nhánh mới tên là bugFix
* Chuyển sang nhánh bugFix với git checkout bugFix
* Commit một lần
* Trở về master với git checkout
* Commit một lần nữa
* Gộp nhánh bugFix vào master với git merge

Nhớ rằng, bạn luôn luôn có thể bật lại hội thoại này với lệnh "objective"!

**Git Rebase**

Cách thứ 2 để kết hợp thành của của 2 nhánh là rebase. Rebase về căn bản là chọn một loạt các commit, "sao chép" chúng, và ném chúng sang chỗ khác.

Nghe có vẻ phức tạp, lợi thế của rebase là có thể tạo ra cây lịch sử thẳng tuột. Ljch sử commit nhìn sẽ gọn gàng hơn nhiều.

Xem nó hoạt động thế nào nào...

git rebase master

git rebase bugFix

Để hoàn thành cấp độ này, làm theo hướng dẫn sau

* Chuyển sang nhánh mới tên là bugFix
* Commit một lần
* Quay về master và commit lần nữa
* Quay trở lại bugFix và rebase sang master

Chúc may mắn!

**DICH CHUYEN TRONG “GIT”**

**HEAD**

Đầu tiên hãy bàn về 'HEAD'. HEAD là cái tên biểu tượng cho commit iện tại đang được trỏ đến -- về căn bản nó là commit mà bạn đang làm việc.

HEAD luôn luôn trỏ đến commit gần nhất được phản ánh trong cây làm việc. Hầu hết các lệnh git mà làm thay đổi nội dung cây làm việc thường bắt đầu với HEAD.

Bình thường HEAD thì trỏ tới tên nhánh (ví dụ bugFix). Khi bạn commit, thì trạng thái của bugFix được thay đổi và thay đổi này được trông thấy thông qua HEAD.

Để hoàn thành cấp độ này, hãy tháo HEAD khỏi bugFix thay vì đó hãy dán nó vào commit.

Chỉ rõ commit bằng mã băm (hash) của nó. Mã băm của mỗi commit nằm trong trong vòng tròn biểu thị commit đó.

**Tham chiếu tương đối**

Dịch chuyển trong Git bằng cách chỉ định mã băm(hash) của commit cỏ vẻ hơi buồn tẻ. Trong đời thực thì sẽ không có mô tả git trực quan ngay bên cạnh terminal của bạn đâu, nên nếu bạn muốn nhìn mã băm của commit thì phải dùng git log thôi.

Hơn nữa, mã băm thực tế thường dài hơn rất nhiều. Ví dụ, mã băm của commit được giới thiệu trong phần trước là fed2da64c0efc5293610bdd892f82a58e8cbc5d8. Đọc mà xoắn hết cả lưỡi...

Được cái là Git cũng khá thông minh về mã băm. Nó chỉ yêu cầu bạn chỉ định mã băm đủ để xác định commit. Cho nên tôi có thể đơn giản chỉ cần gõ fed2 thay vì cái chuỗi dài ngoằng phía trên.

Toán tử **“~”**

Như tôi đã nói, xác định commit bằng mã băm của nó chẳng hề thuận tiện tẹo nào, đó là tại sao Git có những tham chiếu tương đối. Chúng rất tuyệt vời!

Với tham chiếu tương đối, bạn có thể bắt đầu từ những nơi có thể ghi nhớ được (như là nhánh bugFix hoặc HEAD) và làm việc trên đó.

Những commits tương đối rất mạnh mẽ, nhưng chúng tôi sẽ chỉ giới thiệu 2 loại đơn giản sau:

* Dịch chuyển 1 commit lên trên trong 1 lần với ^

git checkout master^ - *lên một bậc của master*

* Dịch chuyển nhiều commit lên trên trong 1 lần với ~<số>

git checkout HEAD~4 – *lên bốn bậc của master*

**Branch forcing**

Tôi thì hay thường dùng tham chiếu tương đối để dịch chuyển nhánh. Bạn có thể trực tiếp gán lại nhánh cho commit với cú pháp -f. Kiểu như thế này:

git branch -f master HEAD~3 - *dịch chuyển (ép buộc) nhánh master lên 3 commit phía trên HEAD.* ***(trong đó* -f *dùng để ép nhánh (force)).***

**Hoàn tác trong Git**

Có 2 cách nguyên thủy để hủy thay đổi trong Git -- một là dùng git reset và cách khác là dùng git revert. Chúng ta sẽ xem xét từng cái trong hội thoại sau.

**Git Reset**

git reset hoàn tác bằng cách chuyển tham chiếu của nhánh ngược lên commit cũ hơn. Bạn có thể hiểu nó như kiểu "viết lại lịch sử;" git reset sẽ dịch chuyển nhánh lên trên như thể commit chưa bao giờ được tạo ra vậy.

git reset HEAD^

**Git Revert**

Trong khi git reset hoạt động rất tốt tại nhánh cục bộ trên máy cá nhân, cách thức "viết lại lịch sử" chẳng hề có tác dụng lên nhánh ở phương xa mà người khác sử dụng.

Để có thể hoàn tác và chia sẻ hoàn tác đó với người khác, thì ta cần sử dụng git revert. Xem thử cách thức nó hoạt động nào

git revert HEAD

Lạ nhỉ, một commit mới được thả vào bên dưới commit mà ta muốn hoàn tác. Đó là bởi vì commit mới C2' này có chứa thay đổi -- đó là những thay đổi về hoàn tác commit C2.

Dùng revert thì bạn có thể đẩy thay đổi mình lên và chia sẻ với người khác.

**Mang đi kéo về**

Cho đến giờ thì ta đã nắm được cơ bản về git -- commit, rẽ nhánh, và dịch chuyển qua lại trên cây mã nguồn. Chừng ấy khái niệm là đã đủ để tác động đến 90% sức mạnh của kho chứa git và các chức nawmg cần thiết cho nhà phát triển phần mềm.

Tuy nhiên 10% còn lại thì lại khá hữu ích trong các quy trình làm việc phức tạp (hoặc khi bạn vướng phải tình huống khó khăn). The next concept we're going to cover is "chỉnh lý các bản ghi" -- đó là cách để nhà phát triển nói rằng"Tôi muốn lấy bản ghi ở đây và cả bản ghi ở đó" một cách chính xác, linh hoạt và sống động.

Nghe thì có vẻ phức tạp, nhưng thực ra khái niệm này khá đơn giản.

**Git Cherry-pick**

Lệnh đầu tiên của loạt bài này là git cherry-pick, dạng lệnh là::

git cherry-pick <Commit1> <Commit2> <...>

Đó là cách rất trực tiếp để copy một loạt commit xuống dưới vị trí hiện tại của bạn (HEAD). Cá nhân tôi thì rất thích cherry-pick bởi tính tiện dụng và dễ hiểu của nó.

Hãy xem qua một ví dụ!

git cherry-pick C2 C4

**Git Rebase Tương tác**

Khi bạn biết rõ mình muốn những commit nào (và mã băm tương ứng của chúng) thì Git cherry-pick là công cụ tuyệt vời -- nó đơn giản đến bất ngờ.

Nhưng mà lỡ như bạn không biết mình cần commit nào thì sao? May mà git cũng có công cụ cho việc này! Ta có thể dùng rebase tương tác cho việc này -- đó là cách tốt nhất để cân nhắc lựa chọn các commit mà bạn muốn rebase.

Hãy đi sâu vào chi tiết nào...

Rebase tương tác nghĩa là dùng lệnh rebase với tùy chọn -i.

Nếu lệnh của bạn có bao hàm tùy chọn này, git sẽ mở một giao diện người dùng để cho bạn biết những commit nào sẽ được sao chép xuống dưới mục tiêu rebase. Nó cũng đồng thời thể hiện mã băm và thông điệp của commit, điều này là tuyệt vời bởi nhờ đó ta có thể phân biệt được chúng.

Với git "thật", cửa sổ UI (giao diện người dùng) sẽ được mở thông qua một tệp nhờ công cụ chỉnh sửa văn bản như là vim. Vì mục đích học tập, tôi đã xây dựng một cửa sổ hội thoại nhỏ mà nó hành xử cũng tương tự như vậy.

Khi hội thoại tương tác rebase bật lên, bạn có thể làm được 3 điều:

* Bạn có thể sắp xếp lại commit một cách đơn giản thông qua UI (bạn có thể làm điều này bằng cách kéo thả trên công cụ của chúng tôi).
* Bạn có thẻ bỏ qua một vài commit. Điều này được thể hiện qua nút pick -- tắt nút pick nghĩa là bạn bỏ qua commit đó.
* Cuối cùng, bạn có thể nén các commit. Đáng tiếc là trình độ của chúng tôi chưa hỗ trợ được chức năng này do vài vấn đề logic, cho nên tôi sẽ bỏ qua phần này. Nói dơn giản thì -- nó cho phép bạn kết hợp các commit.

Tuyệt! Cùng xem qua một ví dụ nào.

git rebase -i HEAD~4 - *(omit = bỏ)*

**Commit xếp chồng tại địa phương**

Có tình huống thế này thường hay xảy ra trong quá trình phát triển: Tôi dang cố dò lỗi nhưng mà nó lại khá là trúc trắc. Để hỗ trợ cho việc này, thêm vào vài dòng lệnh gỡ lỗi và lệnh in.

Mấy lệnh gỡ lỗi và in này nằm yên trong commit của chúng. Cuối cùng thì tôi cũng tìm ra lỗi, gỡ xong, ngon rồi!

Bây giờ thì lại phải đưa bugFix trở về nhánh master. Nếu mà đơn giản dùng fast-forwarded lên master, thì master lại có tất cả các lệnh gỡ lỗi kia(chẳng muốn chút nào). Phải có cách khác chứ nhỉ...

Ta cần phải bảo git chỉ sao chép 1 commit thôi. Điều này giống với cấp độ trước về điều chỉnh vị trí -- ta có thể dùng các câu lệnh tương tự:

* git rebase -i
* git cherry-pick

Để đạt được mục tiêu này.

**Tung hứng Commit**

Có một tình huống xảy ra khá thường xuyên. Bạn có vài thay đổi trên (newImage) và một vài thay đổi khác trên (caption) và chúng lại liên quan đến nhau, nên chúng nằm chồng lên nhau trong kho của bạn (một lại nối một).

Tréo ngoe là bạn lại phải điều chỉnh một chút ở commit trước. Giả sử như tay thiết kết muốn ta đổi chiều của newImage một chút, mặc dù commit ấy đã xưa lắm rồi!!

Để khắc phục khó khăn này ta có thể làm như sau:

* Ta sẽ dùng git rebase -i sắp xếp lại commit để cái ta cần sửa sẽ nằm trên cùng
* Ta sẽ dùng git commit --amend tạo ra một điều chỉnh nhỏ
* Sau đó ta sẽ lại sắp xếp lại commit như trước bằng cách dùng git rebase -i
* Cuối cùng, ta sẽ chuyển master tới phần đã cập nhật để hoàn thành cấp độ (dùng cách nào tùy bạn)

Có nhiều cách để hoàn thành mục tiêu (Tôi thấy bạn hấp háy sang cherry-pick rồi đấy), rồi ta sẽ thấy chúng nhiều hơn, nhưng giờ hãy cứ tập trung vào kỹ thuật này đã. Sau cùng thì, để ý các dấu nháy đơn(') chứ?-- vì ta đã chuyển commit 2 lần, nên chúng có thêm một dấu nháy đơn. Và một dấu nữa cho commit mà ta đã sửa đổi, thế là ta có trạng thái cuối cùng của cây lịch sử

Nói cách khác, khi tôi so sánh kết quả, tôi chỉ so sánh cấu trúc của cây lịch sử. Sự khác biệt về số lượng ' không được bao gồm trong so sánh. Miễn là cấu trúc nhánh master của bạn giống với cấu trúc đích, tôi sẽ vẫn để bạn qua bài.

**Tung hứng Commit #2**

Nếu bạn vẫn chưa hoàn thành Tung hứng Commit #1 (cấp độ trước), hãy làm nó trước khi tiếp tục

Như bạn đã thấy ở cấp độ trước, ta dùng rebase -i để sắp xếp lại các commit. Một khi commit mà ta muốn sửa đã ở trên cùng, ta có thể dễ dàng --chỉnh sửa(amend) nó và sau đó sắp xếp lại trật tự lúc trước.

Nhưng mà vẫn tồn tại vấn đề khi mà ta sắp xếp quá nhiều, điều này có thể dẫn đến xung đột khi rebase. Thử dùng cách khác với git cherry-pick nào.

**Tag trong Git**

Qua các bài học trước, bạn đã biết được rằng rất dễ dàng để di chuyển qua lại giữa các nhánh và chúng thường tham chiếu đến các commit khác nhau trong quá trình ta làm việc trên chúng. Nhánh thường dễ biết đổi, thường chỉ là tạm thời và chúng luôn như vậy.

Nếu như vậy thì bạn có thể sẽ thắc mắc liệu có cách nào để đánh dấu vĩnh viễn một thời điểm nào đó trong lịch sử làm việc không? Khi mà dự án càng ngày càng mở rộng và merge thì diễn ra liên tục, vậy thì có cách nào để đánh dấu những commit đó một cách cố định hơn nhánh không?

Tất nhiên là có chứ! Git có các tag(thẻ) để hỗ trợ vấn đề này -- chúng (bằng cách nào đó) đánh dấu vĩnh viễn một commit cụ thể nào đó như một "cột mốc" mà sau đó bạn có thể tham chiếu đến như với nhánh.

Quan trọng hơn là, khi ta commit thì chúng không hề bị di chuyển. Bạn không thể "check out" một tag rồi làm việc trên đó -- tag giống như là mỏ neo để chỉ định một điểm cụ thể nào dó trên cây lịch sử.

Cùng xem tag thực tế là như thế nào nào.

git tag v1 c1

Nếu bạn không chỉ định commit, git sẽ gắn thẻ vào commit mà HEAD đang trỏ tới.

**Mô tả của Git**

Bởi gì thẻ đóng vai trò như là "mỏ neo" trên cây lịch sử rất tốt rồi, git cũng có lệnh để mô tả tương quan của bạn dến vị trí "mỏ neo" (thẻ) gần nhất. Và đó là git describe!

git descripe có thể giúp bạn định hướng sau khi dịch chuyển qua lại nhiều lần trên cây lịch sử; đặc biệt là sau khi sử dụng git bisect (công cụ tìm kiếm lỗi của git) hoặc khi sử dụng máy của đồng nghiệp mới đi nghỉ mát về.

Cú pháp sử dụng Git describe như sau:

git describe <thamchiếu>

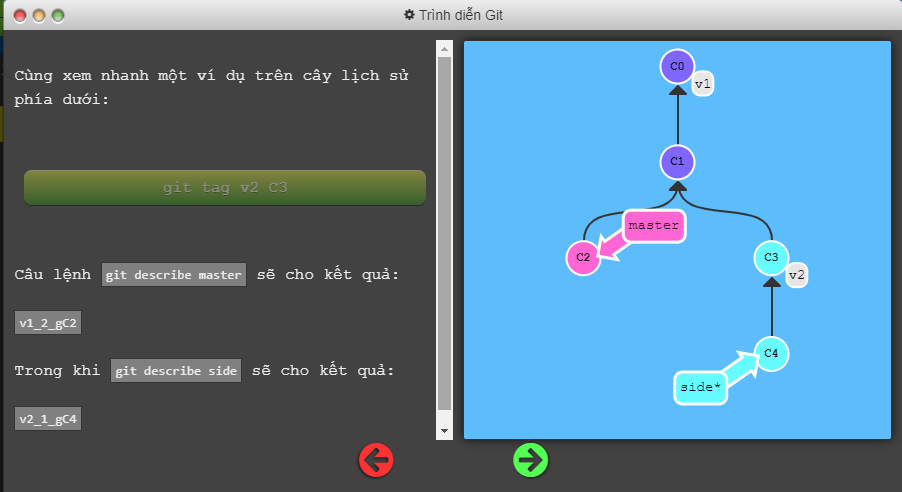
Trong đó **<thamchiếu>** là bất kỳ thứ gì mà git có thể dùng để xác định commit. Nếu bạn không chỉ định tham chiếu, git sẽ dùng vị trí hiện tại của bạn (HEAD).

Đầu ra của câu lệnh sẽ như sau:

<thẻ>\_<sốLượngCommit>\_g<mãBăm>

Trong đó <thẻ> là thẻ tổ tiên gần nhất, <sốLượngCommit> là số lượng commit tính từ tham chiếu đến thẻ, và <mãBăm> là mã băm của commit được mô tả.

git tag v2 c3



**Rebase nhiều nhánh**

Giời ạ, sao mà nhiều nhánh thế khôn biết! Chuyển hết chúng vào master thôi.

Nhưng mà cha quản lý lại muốn oái oăm hơn cơ -- hắn muốn tất cả commit xếp thẳng hàng. Nghĩa là cây lịch sử của ta sẽ có C7' ở cuối, phía trên làC6', và cứ thế theo đúng thứ tự.

Nếu mà nhỡ may bạn làm nhầm thì có thể dùng lệnh reset để bắt đầu lại. Hãy nhớ đối chiếu với đáp án của chúng tôi xem bạn có thể hoàn thành với ít câu lệnh hơn không!

**Tìm về cội nguồn**

Cũng giống ký hiệu ~, ta cũng có thể thêm số theo sau ký hiệu ^.

Nhưng mà không giống như (~) con số theo sau là số lượng thế hệ commit, con số theo sau ^ chỉ định commit cha từ commit merge. Hãy nhớ rằng commit merge có nhiều cha, cho nên chọn cha nào cũng khá là mơ hồ.

Thông thường thì sẽ chọn cha "đầu tiên" từ commit merge, nhưng nếu sau dấu ^ có một con số thì cách hành xử sẽ khác đi.

**Nhánh rối như canh hẹ**

Chààà, anh bạn! Thử thách lần này hơi khoai đây.

Hiện tại thì nhánh master đang có nhiều commit hơn các nhánh one, two và three. Vì một lý do nào đó, ta cần cập nhật những commit trên nhánh master lên các nhánh còn lại kèm với một vài điều chỉnh .

Nhánh one thì cần sắp xếp lại và xóa commit C5, trong khi đó nhánh two thì đơn giản sắp xếp, còn three thì chỉ cần một commit!

Cố gắng động não tìm ra cách đi nhé -- khi bạn làm xong thì nhớ so sánh với đáp án của chúng tôi bằng lệnh show solution.

Bài học về thao tác từ xa

**Push & Pull -- Tác động git từ xa!**

**Git Remotes(Kho lưu trữ từ xa)**

Thực ra thì kho remote cũng không khó hiểu lắm đâu. Giờ đây đâu đâu cũng thấy điện toán đám mây nên ai ai cũng tưởng git remote nó là cái gì đó ma thuật lắm, nhưng chúng thực ra chỉ là những bản sao chép ko chứa của bạn lên máy tính khác. Bạn có thể giao tiếp với chúng qua internet, tù đó mà chuyển giao commit qua lại.

Vì lẽ đó mà kho remote có những đặc tính tuyệt vời:

* Trước hết, có tác dụng như là bản sao dự phòng! Bạn đã biết rằng kho địa phương có thể khôi phục trở lại trạng thái trước đó, nhưng tất cả thông tin đó chỉ được lưu trữ cục bộ. Bằng cách sao chép toàn bộ kho chứa của bạn lên máy tính ở xa, bạn hoàn toàn có thể mất hết dữ liệu tại máy tính mà vẫn tiếp tục làm việc như chưa có gì xảy ra.
* Quạn trọng hơn nữa, kho remote khiến lập trình có tính cộng đồng hơn! Bây giờ bản sao dự án của bạn đã được lưu trũ ở nơi khác, bạn bè của bạn có thể đóng góp (hoặc kéo về cập nhật mới nhất) dự án của bạn rất dễ dàng.

Trực quan hóa các kho lưu trữ từ xa bằng cách sử dụng các trang web đang trở nên phổ biến hơn (như GitHub hoặc Photypeator), nhưng các kho lưu trữ từ xa Luôn luôn \*\* là trụ cột của các công cụ này, vì vậy điều quan trọng là phải hiểu khái niệm này!

**Câu lệnh của chúng tôi để tạo kho lưu trữ từ xa**

Cho đến bây giờ, Learn Git Branching đã tập trung dạy về làm việc trên kho chứa địa phương (rẽ nhánh branch, hợp nhánh merge, dịch chuyển nhánh rebase, vân vân ...). Nhưng bây giờ ta lại muốn học về làm việc trên kho lưu trữ từ xa, ta cần một câu lệnh để cài đặt môi trường cho những bài học này. Đó là git clone

Về mặt kỹ thuật, thì git clone thực tế là câu lệnh dùng để tạo một bản sao chép địa phương của khó chứa từ xa (ví dụ từ github chẳng hạn). Nhưng trong Learn Git Branching ta sẽ dùng câu lệnh này hơi khác biệt một chút -- git clone sẽ tạo ra một bản sao chép lên kho lưu trữu từ xa dựa trên kho chứa cục bộ của bạn. Điều này hoàn toàn ngược lại với câu lệnh thực tế, nhưng nó giúp ta nhân bản và làm việc với kho lưu trữ từ xa, nên ta sẽ dùng nó trong quá trình học.

git clone

**Nhánh Git từ xa**

Giờ bạn đã thấy cách thức hoạt động của git clone, cùng xem xét kỹ hơn những gì đã xảy ra.

Điều đầu tiên mà có thể bạn để ý là một nhánh mới xuất hiện trong kho chứa địa phương của ta là o/master. Loại nhánh này được gọi là nhánh từ xa (remote) ; nhánh từ xa có những thuộc tính đặc biệt vì chúng phục vụ những mục đích duy nhất.

Nhánh từ xa phản ánh trạng thái (state) của kho chứa từ xa (kể từ lần cuối cùng bạn tương tác với kho chứa từ xa). Chúng giúp bạn hiểu về sự khác biệt giữa công tác trên kho chứa cục bộ với kho chứa từ xa -- một bước quan trọng trước khi chia sẻ công việc của bạn với người khác.

Nhánh từ xa có một thuộc tính đặc biệt đó là khi bạn chuyển sang chúng bạn sẽ vào trạng thái tách rời HEAD. Git chủ tâm làm việc này vì bạn không thể công tác trực tiếp trên chúng; bạn phải công tác ở nơi khác và chia sẻ thành quả lên kho chứa từ xa (sau đó nhánh từ xa sẽ được cập nhật).

**o/ là gì vậy?**

Có thể bạn sẽ thắc mắc về ký tự o/ ở đầu tên nhánh từ xa có ý nghĩa gì. Chà, tên nhánh từ xa cũng có (thiết yếu) quy tắc đặt tên -- chúng được hiển thị dưới dạng:

<tên kho từ xa>/<tên nhánh>

Do đó, ở trong o/master thì master là tên nhánh còn o là tên kho chứa từ xa.

Thực tế thì hầu hết các nhà phát triển đặt tên kho chứa từ xa là origin chứ không phải o. Nó trở thành thông lệ đến nỗi Git đặt tên origin cho kho chứa từ xa khi bạn dùng git clone để sao chép một kho chứa.

Đáng tiêc là origin không khớp trong giao diện của chúng tôi, nên chúng tôi đành phải viết tắt là o :( Nhưng hãy nhớ rằng khi dùng git thật sự thì chắc hẳn tên của kho chứa từ xa sẽ là origin!

git checkout o/master; git commit

**Git Fetch**

Lam việc với git remote về căn bản là chuyển giao dữ liệu qua và lại giữa các kho chứa. Miễn là ta có thể chia sẻ qua lại các commit, ta có thể chia sẻ bất kỳ cập nhật gì được theo dõi bở git (và từ đó chia sẻ thành quả, tập tin mới, ý tưởng mới, thư tình, các kiểu con đà điểu.).

Ở bài học này ta sẽ học cách lấy dữ liệu về từ một kho chứa từ xa -- lệnh đó là git fetch.

Bạn sẽ nhận thấy rằng khi ta cập nhật đại điện kho chứa từ xa, Nhánh từ xa của ta cũng được cập nhật để phản ánh đại diện đó. Điều này liên quan đến bài học trước về nhánh từ xa.

git fetch

Đó! Commit C2 và C3 đã được tải xuống kho chứa cục bộ của ta, và nhánh từ xa o/master của ta đã được cập nhật để phản ánh điều này.

**Fetch làm gì**

git fetch thực hiện 2 bước chính, và chỉ 2 bước chính. Nó:

* tải xuống commit có trên kho chứa từ xa mà kho chứa cục bộ của ta không có, và...
* cập nhật nơi mà nhánh từ xa của ta trỏ tới (ví dụ nhánh o/master)

Về căn bản thì git fetch đưa đại diện địa phương của kho chứa từ xa của ta về trạng thái đồng bộ với trạng thái thực tế của kho chứa từ xa (ngay lúc này).

Nếu bạn còn nhớ thì trong bài học trước, chúng tôi có đề cập rằng nhánh từ xa phản ánh trạng thái của kho chứa từ xa kể từ kể từ lần cuối bạn tương tác với các nhánh từ xa đó. git fetch chính là cách để tương tác với cách nhánh từ xa! Hi vọng chúng tôi đã làm sáng tỏ sự liên kết giữa nhánh từ xa và git fetch.

git fetch thường tương tác với kho chứa từ xa bằng Internet (thông qua các giao thức như http:// hoặc git://).

**fetch không làm gì**

Tuy nhiên, git fetch không hề thay đổi trạng thái kho chứa địa phương của bạn. Nó sẽ không cập nhật nhánh master hay thay đổi bất kỳ điều gì trong hệ thống tập tin hiện tại của bạn.

Điều này rất quan trọng, vì nhiều nhà phát triển nghĩ rằng git fetch sẽ làm kho chứa địa phương của họ phản ánh trạng thái của kho chúa từ xa. Nó có thể tải xuống tất cả dữ liệu cần thiết, nhưng nó thực tế không thay đổi bất cứ thứ gì trên tập tin địa phương của bạn. Ta sẽ học lệnh để làm điều đó sau :D

Nói tóm lại, bạn có thể coi git fetch thuần túy như một bước tải xuống.

**Git Pull**

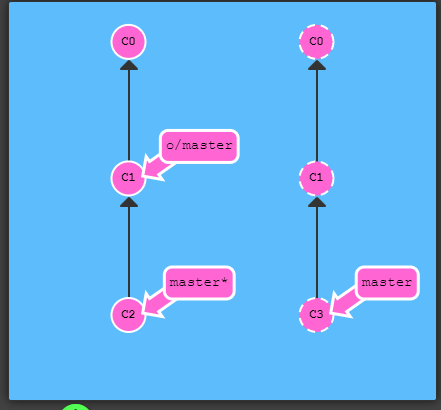
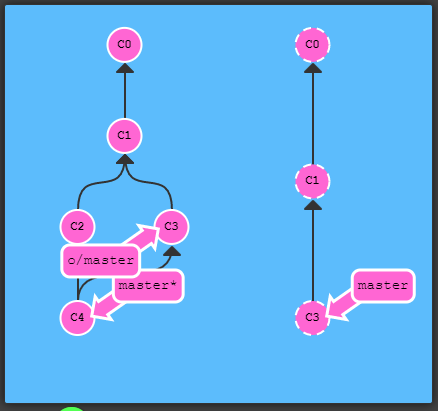
Ta đã biết cách dùng git fetch để nạp các commit từ kho chứa từ xa, giờ hãy cập nhật các commit này vào các nhánh địa phương!

Thực ra thì có nhiều cách đề làm điều này -- một khi bạn đã có các commit này ở ko chứa địa phương, bạn có thể hợp nhất chúng như với các commit ở các nhánh khác. Nghĩa là bạn có thể sử dụng các câu lệnh sau:

* git cherry-pick o/master
* git rebase o/master
* git merge o/master
* etc., etc.

Thực tế thì, quá trình nạp commit từ kho chứa từ xa và hợp nhất chúng phổ biết đến nỗi git cung cấp một câu lệnh để làm cả 2 điều này một lúc! Đó là git pull.

git fetch; git merge o/master

Bùùm -- ta tải xuống commit C3 với lệnh git fetch rồi sau đó hợp nhất nó với lệnh git merge o/master. Giờ thì nhánh master đã phản ánh trạng thái mới từ kho chứa từ xa (trong trường hợp này là origin)

Nếu thay vì dó ta dùng git pull thì điều gì xảy ra?

Cũng như nhau! Điều đó có nghĩa là git pull về cơ bản là cách làm tắt cho việc thực hiện tuần tự git fetch sau đó là hợp nhất các nhánh từ xa được nạp.

**Mô phỏng hợp tác**

Bây giờ khó đây -- để chuẩn bị cho những bài học tới, chúng tôi cần chỉ bạn cách kéo về những cập nhật có trên kho chứa từ xa.

Có nghĩa là ta cần "giả vờ" kho chứa từ xa được cập nhật bởi đồng nghiệp / bạn / cộng tác viên của bạn, có khi là một nhánh xác định hoặc là một số commit.

Để làm điều này, chúng tôi sẽ giới thiệu cho bạn một lệnh tự tạo git fakeTeamwork! Cái tên nói lên tất cả, cùng xem thử bản giới thiệu...

Hành xử mặc định của fakeTeamwork đơn giản là tạo ra một commit ở trên nhánh master từ xa.

git fakeTeamwork

Đó -- kho chứa từ xa đã được cập nhật thêm một commit, và ta chưa tải commit đó xuống vì ta chưa hề chạy lệnh git fetch.

Bạn cũng có thể chỉ định số lượng commit bằng cách thêm con số đằng sau câu lệnh.

git fakeTeamwork foo 3

Chỉ với một câu lệnh ta có thể giả lập tạo ra 3 commit trên nhánh foo từ xa của ta.

#work:

Các cấp độ tới sẽ khá là khó, nên ở cấp độ này chúng tôi sẽ yêu cầu bạn thêm một chút.

Sao chép một kho chứa từ xa (sử dụng git clone), sau đó mô phỏng một số thay đổi trong kho chứa từ xa mà bạn vừa tạo, sau đó thực hiện một số commit trên nhánh cục bộ của riêng bạn, sau đó kéo các thay đổi từ kho chứa từ xa. Nhiều bài học trong một bài.

**Git Push**

Được rồi, ta đã biết cách nạp thay đổi từ kho chứa từ xa và kết hợp chúng vào các nhánh cục bộ. Khá là tuyệt rồi... nhưng nếu tôi muốn chia sẻ tác phẩm tuyệt vời của tôi với mọi người khác thì sao?

Chà, cách tải lên thì phải ngược với tải xuống rồi. Vậy thì đối nghịch của git pull(kéo) là gì? git push(đẩy)!

git push có trách nhiệm tải lên thay đổi của bạn vào nhánh từ xa được chỉ định và cập nhật nhánh đó để kết hợp với commit đẩy lên của bạn. Một khi lệnh git push hoàn thành, tất cả bạn bè của bạn có thể tải xuống thay đổi của nhánh từ xa đó đó.

Bạn có thể xem git push là câu lệnh để "xuất bản" thành quả công việc của bạn. Lệnh này có nhiều tính năng tinh tế mà ta sẽ tìm hiểu nhanh thôi, nhưng giờ hãy cứ bắt đầu với từng bước nhỏ đã...

*lưu ý --git push mà không có tham số hành xử tùy biến phụ thuộc vào cài đặt của git là push.default. Giá trị mặc định cho cài đặt này phụ thuộc vào phiên bản git mà bạn đang sử dụng, còn ở bài học của chúng ta thì ta sẽ sử dụng giá trị upstream (ngược dòng). Bây giờ thì đó chưa phải là vấn đề gì lớn, nhưng chúng tôi khuyến nghị bạn kiểm tra cài đặt của mình trước khi đẩy lên dự án của bạn.*

**Diverged Work**

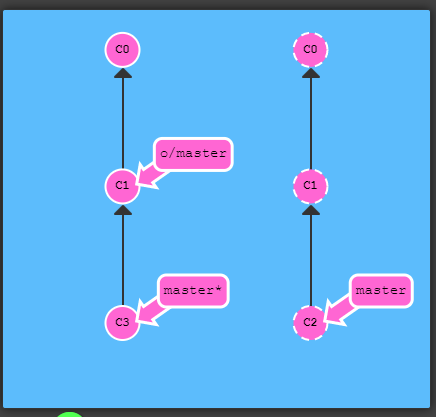
Cho đến giờ ta đã biết cách kéo(pull) commit từ nơi khác về và cách đẩy(push) lên những thay đổi của ta. Chúng nhìn có vẻ khá đơn giản, vậy tại sao người ta lại thấy lúng túng về chúng?

Khó khăn đến từ sự dị biệt của cây lịch sử trên kho chứa từ xa. Trước khi đi vào thảo luận chi tiết vấn đề này, hãy xem qua một ví dụ...

Tưởng tượng bạn sao chép một kho chứa vào thứ Hai và bắt đầu phát triển một tính năng mới. Đến thứ Sáu thì bạn đã sẵn sàng để xuất bản thành quả của mình -- nhưng không! Đồng nghiệp của bạn đã viết thêm hàng loạt mã trong một tuần vừa rồi và điều này làm cho chức năng của bạn trở nên lỗi thời. Và họ cũng đã xuất bản những commit này lên kho chứa từ xa chung, vậy giờ thành quả của bạn lại dựa trên phiên bản cũ của dự án mà nó không còn thích đáng nữa.

Trong trường hợp này, lệnh git push trở lên khá nhập nhằng. Nếu bạn dùng git push, liệu git nên thay đổi kho chứa từ xa trơ rveef trạng thái ngày thứ Hai? Hay nó nên cố gắng thêm mã của bạn vào trong khi không xóa mã mới? Hay là nó sẽ bỏ qua hoàn toàn mã của bạn vì nó đã lỗi thời?

Vì có quá nhiều tình huống mơ hồ (dị biệt lịch sử), git sẽ không cho phép bạn đẩy(push) thay đổi của mình. Nó sẽ ép bạn phải sát nhập trạng thái mới nhất của kho chứa từ xa vào thành phẩm của mình trước khi chia sẻ chúng.

git push

Thấy chứ? Không có gì xảy ra cả vì lệnh bị thất bại. git push thất bại vì commit mới nhất của bạn: C3 dựa trên commit C1 ở nhánh từ xa. Nhánh đó đã được cập nhật lên commit C2, nên git từ chối lệnh đẩy của bạn

Vậy thì giải quyết tình huống này sao giờ? Đơn giản thôi, tất cả những gì bạn cần làm là khiến cho thành quả của mình dựa trên phiên bản mới nhất của nhánh từ xa.

Có vài cách để làm điều này, nhưng cách trực tiếp nhất là bố trí lại (dùng rebase) để di chuyển thành quả của bạn. Cùng xem cách nó hoạt động nào.

Có cách nào khác để cập nhật thành quả của mình khi kho chứa từ xa được cập nhật không? Tất nhiên rồi! Hãy xem xét điều tương tự với merge.

Mặc dù git merge không dịch chuyển commit của bạn (thay vì đó nó tạo ra một commit hơp nhất), đó là cách để nói với git rằng bạn đã kết hợp tất cả thay đổi từ nhánh từ xa. Đó là bởi vì bây giờ nhánh từ xa đã trở thành một tổ tiên của nhánh cục bộ của bạn, nghĩa là commit của bạn có thể tham chiếu điến tất cả commit có ở nhánh tù xa.

Hãy xem qua bản biểu diễn sau...

git fetch; git rebase o/master; git push

Tuyệt vời! Có cách nào để làm việc này mà không phải gõ nhiều lệnh thế không nhỉ?

git fetch; git merge o/master; git push

Tất nhiên rồi -- bạn đã biết rằng git pull là lệnh tắt của fetch và merge. Tương tự như vậy, git pull --rebase là lệnh tắt của fetch và rebase!

Hãy xem thử cách lệnh tắt này hoạt động ra sao.

git pull –rebase; git push

Quy trình tìm nạp (fetch), bố trí lại/hợp nhất (rebase/merge), và đẩy (push) là khá phổ biết. Ta sẽ xem xét các phiên bản phức tạp hơn của những quy trình này ở các bài học sau, còn bây giờ thì hãy tập trung vào bài này đã.

Để giải quyết cấp độ này, hãy làm các bước sau:

* Nhân bản kho chứa của bạn
* Giả lập làm việc nhóm (1 commit)
* Tạo ra 1 commit địa phương
* Xuất bản thành quả của bạn dùng rebase

**Remote Rejected!**

If you work on a large collaborative team its likely that master is locked and requires some Pull Request process to merge changes. If you commit directly to master locally and try pushing you will be greeted with a message similar to this:

! [remote rejected] master -> master (TF402455: Pushes to this branch are not permitted; you must use a pull request to update this branch.)

**Why was it rejected?**

The remote rejected the push of commits directly to master because of the policy on master requiring pull requests to instead be used.

You meant to follow the process creating a branch then pushing that branch and doing a pull request, but you forgot and committed directly to master. Now you are stuck and cannot push your changes.

**The solution**

Create another branch called feature and push that to the remote. Also reset your master back to be in sync with the remote otherwise you may have issues next time you do a pull and someone else's commit conflicts with yours.