# Tóm tắt các khái niệm cơ bản

Bạn là lập trình viên, và đôi khi bạn muốn đưa về trạng thái trước khi quậy phá của file code nào đó? Cách đơn giản nhất đó là sao chép lại file trước khi chỉnh sửa. Trường hợp dùng phương pháp này thì sẽ phải thường xuyên thực hiện việc thêm ngày đã thay đổi vào tên thư mục hay file. Tuy nhiên, việc tự mình sao chép file mỗi lần chỉnh sửa thì sẽ rất vất vả, và cũng dễ xảy ra nhầm lẫn.  
Và để giải quyết những vấn đề này thì các hệ thống quản lý phiên bản như Git đã được ra đời.

[#](https://blog.duyetdev.com/2015/04/git-va-cac-khai-niem-co-ban.html" \l "Gitlg) Git là gì?

Git là một trong những Hệ thống Quản lý Phiên bản Phân tán, vốn được phát triển nhằm quản lý mã nguồn (source code) của Linux.  
  
Trên Git, ta có thể lưu trạng thái của file dưới dạng lịch sử cập nhật. Vì thế, có thể đưa file đã chỉnh sửa một lần về trạng thái cũ hay có thể biết được file đã được chỉnh sửa chỗ nào.



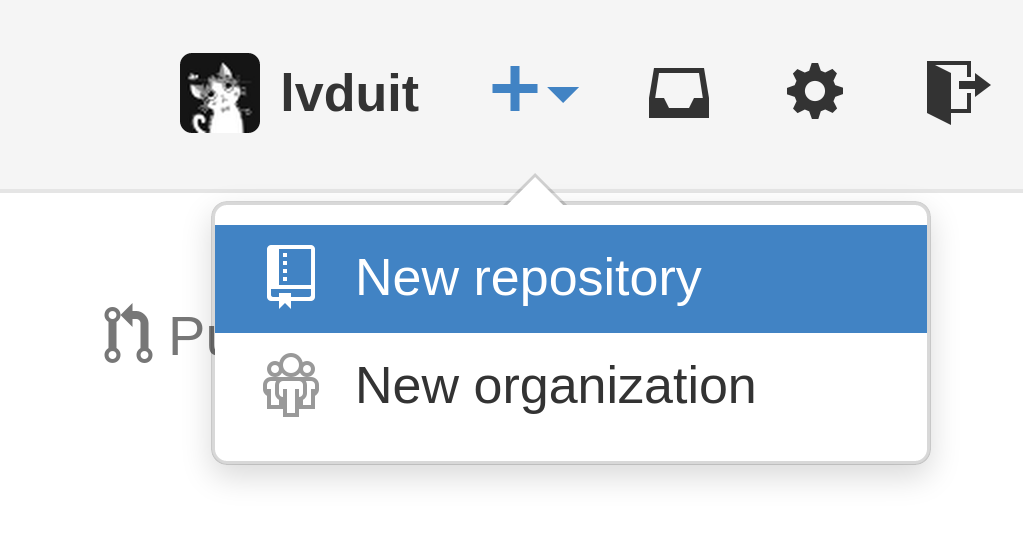
Thêm nữa, khi định ghi đè (overwrite) lên file mới nhất đã chỉnh sửa của người khác bằng file đã chỉnh sửa dựa trên file cũ, thì khi upload lên server sẽ hiện ra cảnh cáo. Vì thế, sẽ không xảy ra lỗi khi ghi đè lên nội dung chỉnh sửa của người khác mà không hề hay biết.  
  
Git sử dụng mô hình phân tán, ngược lại so với SVN hoặc CSV. Mỗi nơi lưu source sẽ đc gọi là repositories, không cần lưu trữ tập trung một nơi, mà mỗi thành viên trong team sẽ có một repository ở máy của riêng mình.  
Điều đó có nghĩa là nếu có 3 người A,B,C cùng làm việc trong 1 project. Thì bản thân repo trên máy của người A, người B, và người C có thể kết nối được với nhau.  
  
Khi quyết định thay đổi chỗ nào đó lên server ta chỉ cần một thao tác "push" nó lên server. Chúng ta vẫn có thể share thay đổi của chúng ta cho thành viên khác, bằng cách commit hoặc update trực tiếp từ máy của họ mà không phải thông qua repositories gốc trên server (thông qua share ssh cho nhau).  
  
Lợi ích

* An toàn hơn (vì mỗi bản copy của thành viên đều là full copy từ repository gốc, khi server bị down).
* Các thành viên vẫn có thể làm việc offline, họ vẫn có thể commit và update trên local của họ hoặc thậm chí với nhau mà không cần thông qua server.
* Khi server hoạt động trở lại, họ có thể cập nhật tất cả lên lại server.

Tiếp đây mình xin giới thiệu về các khái niệm mà bạn cần nắm trong git:

[#](https://blog.duyetdev.com/2015/04/git-va-cac-khai-niem-co-ban.html" \l "KhoRepoRepository) Kho Repo (Repository)

Repository hay được gọi tắt là Repo, đơn giản là nơi chứa tất cả những thông tin cần thiết để duy trì và quản lý các sửa đổi và lịch sử của toàn bộ project. Trong Repo có 2 cấu trúc dữ liệu chính là Object Store và Index. Tất cả dữ liệu của Repo đèu được chứa trong thư mục bạn đang làm việc dưới dạng folder ẩn có tên là .git



[#](https://blog.duyetdev.com/2015/04/git-va-cac-khai-niem-co-ban.html" \l "Remoterepositoryvlocalrepository) Remote repository và local repository

Đầu tiên, repository của Git được phân thành 2 loại là remote repository và local repository.

* Remote repository: Là repository để chia sẻ giữa nhiều người và bố trí trên server chuyên dụng.
* Local repository: Là repository bố trí trên máy của bản thân mình, dành cho một người dùng sử dụng.

Do repository phân thành 2 loại là local và remote nên với những công việc bình thường thì có thể sử dụng local repository. Khi muốn public nội dung công việc mà mình đã làm trên local repository, thì ta sẽ upload lên remote repository rồi public. Thêm nữa, thông qua remote repository bạn cũng có thể lấy về nội dung thay đổi của người khác.



Nguồn ảnh: www.git-tower.com

[#](https://blog.duyetdev.com/2015/04/git-va-cac-khai-niem-co-ban.html" \l "NhnhBranch) Nhánh (Branch)

Đây là một trong những thế mạnh của git là nhánh. Với git, việc quản lý nhánh rất dễ dàng. Mỗi nhánh trong Git gần giống như một workspace. Việc nhảy vào một nhánh để làm việc trong đó tương tự việc chuyển qua ngữ cảnh làm việc mới, và sau đó có thể nhanh chóng quay lại ngữ cảnh cũ.  
  
Nhánh (branch) được dùng để phát triển tính năng mới mà không làm ảnh hưởng đến code hiện tại.  
  
Nhánh master là nhánh “mặc định” khi bạn tạo một repository.  
Nhánh master thông thường là nhánh chính của ứng dụng. Ví dụ bạn thử nghiệm một tính năng mới và muốn không ảnh hưởng đến code chính bạn có thể tạo một nhánh mới và sau khi xong sẽ hợp nhất lại với nhánh master. Việc hợp nhất 2 nhánh lại được gọi là merge.



Nguồn ảnh: http://rogerdudler.github.io/git-guide/index.vi.html

[#](https://blog.duyetdev.com/2015/04/git-va-cac-khai-niem-co-ban.html" \l "TrnMerge) Trộn (Merge)

Trộn source từ một nhánh khác vào nhánh hiện tại. Bạn sẽ hiểu sâu hơn về nó trong các ví dụ ở các bài sau.  
Chú ý:

* Kiểm tra branch hiện đang làm việc trước khi merge
* Phải đẩy tất cả những thay đổi dưới máy local lên Git trước khi merge
* Trước khi merge phải lấy hết những thay đổi mới nhất của các branch khác, hay ít nhất là branch cần merge về máy
* Merge thành công thì nên đẩy source lên lại lên server
* Nên merge bằng GUI tool.

[#](https://blog.duyetdev.com/2015/04/git-va-cac-khai-niem-co-ban.html" \l "XungtConflict) Xung đột (Conflict)

Conflic là trường hợp có 2 sự thay đổi trong một dòng code và máy tính không thể tự quyết định dòng code nào là “đúng”. Đúng ở đây có nghĩa là “ý đồ của lập trình viên”.

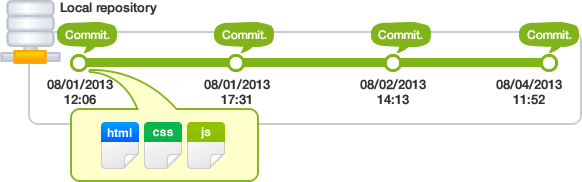
Để giải quyết mâu thuẫn bạn phải dùng “tay không” để sữa các xung đột này. Bạn chỉ việc nhìn vào file bị conflict và tự quyết định dòng code nào giữ lại, dòng nào xóa bỏ.

[#](https://blog.duyetdev.com/2015/04/git-va-cac-khai-niem-co-ban.html" \l "Commit) Commit

Để ghi lại việc thêm/thay đổi file hay thư mục vào repository thì sẽ thực hiện thao tác gọi là Commit.

Khi thực hiện commit, trong repository sẽ tạo ra commit (hoặc revision) đã ghi lại sự khác biệt từ trạng thái đã commit lần trước với trạng thái hiện tại.

Commit này đang được chứa tại repository, các commit nối tiếp với nhau theo thứ tự thời gian. Bằng việc lần theo commit này từ trạng thái mới nhất thì có thể biết được lịch sử thay đổi trong quá khứ hoặc nội dung thay đổi đó.



Các commit này, được đặt tên bởi 40 ký tự alphabet (mã md5 thì phải) không trùng nhau được băm từ thông tin commit. Bằng việc đặt tên cho commit, có thể chỉ định commit nào từ trong repository.

Mỗi commit đều có yêu cầu phải có commit message, để giải thích commit này là bạn đã làm gì trong này.

[#](https://blog.duyetdev.com/2015/04/git-va-cac-khai-niem-co-ban.html" \l "GitRemote) Git Remote

Để kết nối được với một repo khác người ta sử dụng một khái niệm gọi là remote.

Trên thực tế khi làm việc với nhau thì không như vậy, vì không phải máy ai cũng cài một “git server” để người khác kết nối được với mình. Thông thường thì chúng ta sẽ sử dụng một repo chung và các máy kết nối vào repo đó.

Có 2 “git repo server” được sử dụng nhiều là github.com và bitbucket.org.

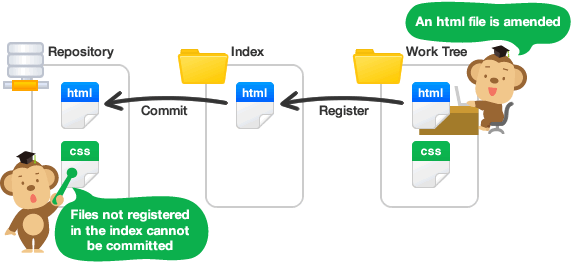
Trên thực tế khi có 2 người cùng làm việc với 1 project thì thông thường sẽ tạo một repo trên github hoặc bitbucket và repo trên máy người A sẽ kết nối với repo trên github và máy người B cũng kết nối với repo trên github/bitbucket. Từ đó source code của người A và người B sẽ được đồng bộ với nhau thông qua repo trên github/bitbucket.

Vì vậy, trước khi sử dụng git thì bạn nên đăng kí một tài khoản trên github.com hoặc bitbucket.org.

[#](https://blog.duyetdev.com/2015/04/git-va-cac-khai-niem-co-ban.html" \l "WorkingTreevIndex) Working Tree và Index

Trên Git, những thư mục được đặt trong sự quản lý của Git mà mọi người đang thực hiện công việc trong thực tế được gọi là working tree.

Và trên Git, giữa repository và working tree tồn tại một nơi gọi là index. Index là nơi để chuẩn bị cho việc commit lên repository.



Nguồn ảnh: backlogtool

Trên Git, khi đã thực hiện commit thì trạng thái sẽ không được ghi trực tiếp trong repository từ working tree, mà sẽ ghi trạng thái đã được thiết lập của index được xây dựng ở giữa đó. Vì thế, để ghi lại trạng thái của file bằng commit thì trước hết cần thông báo file trong index.

[#](https://blog.duyetdev.com/2015/04/git-va-cac-khai-niem-co-ban.html" \l "Kt) Kết

Còn khá nhiều định nghĩa trên Git, trên đây mình chỉ nói một vài định nghĩa quan trọng, mình sẽ cố gắng cập nhật đầy đủ sau.

Ở bài sau mình sẽ hướng dẫn cách sử dụng Git, sử dụng github, các làm việc trên git và xử lý các trường hợp thường gặp khi teamwork với Git.

# Series học Git với Thạch Phạm

## **[B01] Giới thiệu serie Git cơ bản**

**Tổng quan serie Git cơ bản**

Trong serie này, bạn sẽ được làm quen với hệ thống Git và cách sử dụng nó theo hướng dễ hiểu nhất. Bạn sẽ hiểu được các khái niệm và thuật ngữ trong Git như *Clone, Branch, Stage, Commit, Tracking, Merge, Pull, Push,*…và nó sẽ là câu trả lời cho bạn để hiểu Git là cái gì và cách sử dụng nó, tại sao nên sử dụng nó.

**Kiến thức yêu cầu**

Đây là một serie về Git nên mình cần các bạn có sẵn các kiến thức như sau:

* Biết rõ bạn đang tìm hiểu về cái gì.
* Đã từng sử dụng Linux trên máy tính cá nhân hoặc Server.
* Đã biết lập trình.

**Tài nguyên về Git**

Nếu bạn cảm thấy serie này mình hướng dẫn chưa được chi tiết mà bạn cần tìm hiểu thêm thì dưới đây là một số tài nguyên bạn nên tham khảo qua về Git.

* [Ebook về Git tiếng Việt](http://git-scm.com/book/vi/v1/B%E1%BA%AFt-%C4%90%E1%BA%A7u)
* [Ebook về Git tiếng Anh](http://git-scm.com/book/en/v2/Getting-Started-About-Version-Control) (nên đọc)

**Công cụ chuẩn bị**

Để sử dụng Git thì hầu như không cần chuẩn bị công cụ gì, ngoài việc mình khuyến khích các bạn nên thực hành trên hệ điều hành Linux như Ubuntu, Mac,…bạn có thể dùng phần mềm *VMWare Workstation* tạo ra một máy ảo và cài Ubuntu Desktop vào. Tuy nhiên nếu bạn vẫn muốn sử dụng Git trên Windows thì cũng được thôi nhưng do Git là một hệ thống phát triển trên nền Linux nên bạn dùng Linux sẽ tiện hơn.

Chỉ cần như vậy thôi, nếu bạn đã sẵn sàng thì chúng ta sẽ bắt đầu với Git trong bài viết tiếp theo.

# [Git] Git và Github là gì? Tại sao nên dùng?

Một câu hỏi mà nhiều người đặt ra khi nhìn thấy serie này đó là Git là cái gì? Nó là ngôn ngữ lập trình gì chăng? Hay một loại thức ăn nào đó? Một kỹ thuật thiết kế website sao? Hoàn toàn không, bạn sẽ hiểu được Git là cái gì sau khi đọc hết từng chữ trong bài viết này.

Trước khi nói qua về Git, mình có một câu chuyện ngắn muốn kể cho các bạn như sau.

Có hai anh chàng lập trình viên đang hợp tác với nhau cùng xây dựng một phần mềm cho khách hàng. Dĩ nhiên hai anh chàng này sẽ làm việc với nhau bằng cách mỗi người tự viết code một ít và gửi cho nhau qua email hoặc một phương thức gửi dữ liệu thông thường nào đó. Mỗi người sau khi nhận được code của nhau sẽ tiến hành tự xem và sửa lại, sau đó cùng nhau ráp nối vào phần mềm mà họ cần làm. Tuy nhiên, một hôm anh A bỗng nhiên vô tình viết code sai vào phần mềm dẫn tới sản phẩm bị lỗi mà trước đó cả hai anh đều không hề sao lưu lại do phần mềm quá lớn. Và thế là hai anh lại cùng nhau xây dựng lại tính năng bị lỗi và tiếp tục gửi cho nhau như vậy.

Thế bạn có thấy vấn đề gì xảy ra ở câu chuyện trên không? Mình thì mình thấy có hai vấn đề như sau:

1. Việc tự code riêng và gửi cho nhau qua email rất mất thời gian của nhau. Giá như anh A có thể chủ động xem những thay đổi của anh B từ xa và tiến hành gộp trực tiếp những thay đổi của anh B vào sản phẩm.
2. Việc sửa code mà không sao lưu khiến cho họ phải viết lại code từ đầu khi phần mềm bị lỗi.

Và hai vấn đề trên Git có thể sẽ giúp hai anh chàng ấy giải quyết nhanh gọn lẹ và đơn giản hơn rất nhiều. Ngoài ra Git còn làm nhiều việc hơn nữa mà chỉ có thể sử dụng bạn mới hiểu được.

**Git là gì?**

**Git** là tên gọi của một **Hệ thống quản lý phiên bản phân tán** (*Distributed Version Control System –****DVCS***) là một trong những hệ thống quản lý phiên bản phân tán phổ biến nhất hiện nay. DVCS nghĩa là hệ thống giúp mỗi máy tính có thể lưu trữ nhiều phiên bản khác nhau của một mã nguồn được nhân bản (**clone**) từ một kho chứa mã nguồn (**repository**), mỗi thay đổi vào mã nguồn trên máy tính sẽ có thể ủy thác (**commit**) rồi đưa lên máy chủ nơi đặt kho chứa chính. Và một máy tính khác (nếu họ có quyền truy cập) cũng có thể clone lại mã nguồn từ kho chứa hoặc clone lại một tập hợp các thay đổi mới nhất trên máy tính kia. Trong Git, thư mục làm việc trên máy tính gọi là **Working Tree**. Đại loại là như vậy.

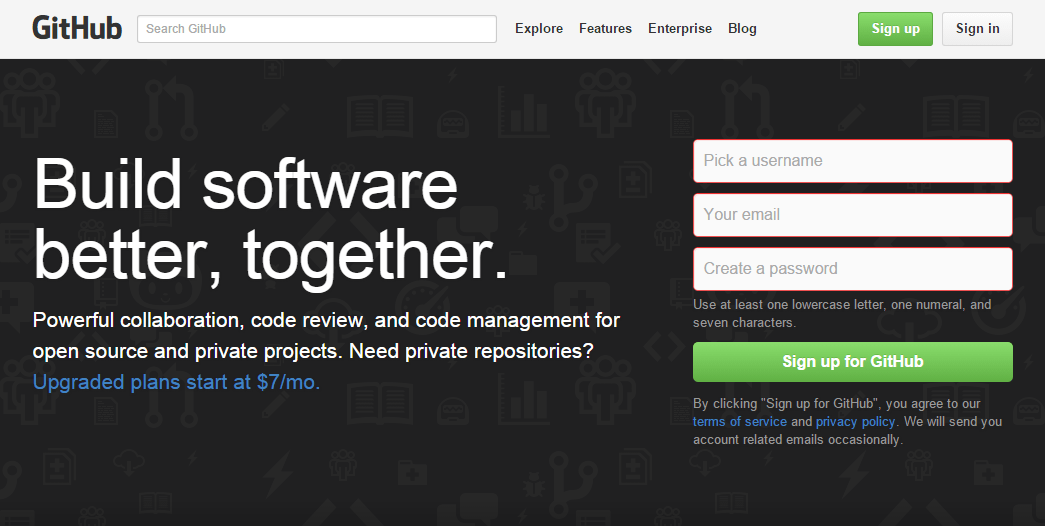


*Mô hình hoạt động của DVCS*

Ngoài ra, có một cách hiểu khác về Git đơn giản hơn đó là nó sẽ giúp bạn lưu lại các phiên bản của những lần thay đổi vào mã nguồn và có thể dễ dàng khôi phục lại dễ dàng mà không cần copy lại mã nguồn rồi cất vào đâu đó. Và một người khác có thể xem các thay đổi của bạn ở từng phiên bản,  họ cũng có thể đối chiếu các thay đổi của bạn rồi gộp phiên bản của bạn vào phiên bản của họ. Cuối cùng là tất cả có thể đưa các thay đổi vào mã nguồn của mình lên một kho chứa mã nguồn.

Cơ chế lưu trữ phiên bản của Git là nó sẽ tạo ra một “*ảnh chụp*” (***snapshot***) trên mỗi tập tin và thư mục sau khi commit, từ đó nó có thể cho phép bạn tái sử dụng lại một ảnh chụp nào đó mà bạn có thể hiểu đó là một phiên bản. Đây cũng chính là lợi thế của Git so với các DVCS khác khi nó không “lưu cứng” dữ liệu mà sẽ lưu với dạng snapshot.

**Github là gì?**



*Github là một dịch vụ máy chủ Repository*

Mình biết là có rất nhiều bạn khi nghe nói đến Git sẽ nghĩ ngay đến [Github](https://github.com/) và có thể sẽ có một số hiểu lầm với họ. Cũng xin nhắc lại rằng, Git là tên gọi của một mô hình hệ thống. Như mình đã giải thích ở trên, **các máy tính có thể clone lại mã nguồn từ một repository** và **Github chính là một dịch vụ máy chủ repository** công cộng, mỗi người có thể tạo tài khoản trên đó để tạo ra các kho chứa của riêng mình để có thể làm việc.

Mặc dù Git có thể làm việc với bất kỳ trên máy chủ Linux nào nhưng để dễ hiểu và thực tế hơn, mình sẽ hướng dẫn các bạn sử dụng Git với Github trong suốt serie này.

**Liên kết**: [Đăng ký tài khoản Github](https://github.com/join)

**Tại sao nên sử dụng Git?**

Có rất nhiều lợi thế để bạn nên sử dụng Git trong việc lập trình ngay từ hôm nay, bất kể là lập trình cái gì đi chăng nữa.

* Git dễ sử dụng, an toàn và nhanh chóng.
* Có thể giúp quy trình làm việc code theo nhóm đơn giản hơn rất nhiều bằng việc kết hợp các phân nhánh (branch).
* Bạn có thể làm việc ở bất cứ đâu vì chỉ cần clone mã nguồn từ kho chứa hoặc clone một phiên bản thay đổi nào đó từ kho chứa, hoặc một nhánh nào đó từ kho chứa.
* Dễ dàng trong việc deployment sản phẩm.
* Và nhiều hơn thế nữa.

Nếu bạn là một lập trình viên thì Git là một hệ thống bạn cần phải biết cách sử dụng, ít nhất là ngay từ bây giờ.

Và bắt đầu sử dụng Git thế nào thì ở bài sau chúng ta sẽ tìm hiểu tới.

# [Git] Cài đặt Git và thiết lập ban đầu

Để có thể sử dụng được Git, bạn phải cài ứng dụng Git vào máy tính để có thể sử dụng các dòng lệnh của Git vì toàn bộ quy trình làm việc với Git đều diễn ra các dòng lệnh (mặc dù ở Windows có một  số phần mềm sử dụng Git với GUI nhưng mình không đề cập ở serie này).

**Cài Git vào Linux**

Nếu bạn đang sử dụng hệ điều hành Ubuntu/Debian thì có thể sử dụng lệnh sau để cài Git.

**$** sudo apt-get install git

Hoặc lệnh sau để cài trên CentOS/Fedora/RHEL.

**$** yum install git

**Cài Git vào Mac OS**

Đối với Mac, bạn có thể sử dụng file installer tải tại địa chỉ <http://git-scm.com/download/mac> để cài đặt.

**Cài Git vào Windows**

Nếu bạn dùng Windows thì có thể tải file .exe cài đặt Git tại địa chỉ <http://git-scm.com/download/win>. Khi cài bạn có thể để nguyên tùy chọn mặc định mà không cần tùy chỉnh gì thêm nếu bạn chưa hiểu về nó.

Sau khi cài đặt Git vào Windows, bạn sẽ cần mở ứng dụng **Git Bash** lên để bắt đầu sử dụng các dòng lệnh của Git.

**Thiết lập chứng thực cá nhân**

Sau khi cài Git xong, việc đầu tiên bạn nên làm là khai báo tên và địa chỉ email vào trong file cấu hình của Git trên máy. Để làm điều này bạn sẽ cần sử dụng hai lệnh sau đây để thiết lập tên và email.

**$** git config --global user.name "Thach Pham"

**$** git config --global user.email "contact@thachpham.com"

Sau khi thiết lập xong, bạn có thể kiểm tra thông tin chứng thực trên user của bạn bằng cách xem tập tin ~/.gitconfig (nhắc lại rằng dấu ~ nghĩa là thư mục gốc của user).

**$** cat ~/.gitconfig

[user]

name = Thach Pham

email = contact@thachpham.com

Hoặc bạn cũng có thể dùng lệnh git config --list để ghi danh sách các thiết lập hiện tại mà bạn đã làm.

Như vậy là bạn đã xong bước ban đầu đó là cài đặt Git và thiết lập tên và email của mình vào Git để bắt đầu làm việc. Ở phần sau, mình sẽ hướng dẫn cách bạn tạo ra một local repository (kho chứa trên máy cá nhân) để xem Git hoạt động thế nào.

# [Git] Cách tạo một repository

**Repository** (kho chứa) nghĩa là nơi mà bạn sẽ lưu trữ mã nguồn và một người khác có thể sao chép (clone) lại mã nguồn đó nhằm làm việc. Repository có hai loại là *Local Repository* (Kho chứa trên máy cá nhân) và Remote *Repository* (Kho chứa trên một máy chủ từ xa).

Trong bài này, mình sẽ hướng dẫn bạn cách tạo local repository và remote repository (sử dụng Github) và làm việc với nó.

**Tạo local repository**

Trước hết, để tạo một repository thì bạn cần truy cập vào thư mục của mã nguồn với lệnh cd, sau đó sử dụng lệnh git init để khởi tạo repository trong thư mục đó. Ở ví dụ này, mình sẽ tạo tạo ra một thư mục mới để chứa code sau này và khởi tạo repository cho nó, mình sẽ dùng lệnh git init tên\_folder để nó tự khởi tạo thư mục.

**$** git init git\_example

Initialized empty Git repository in /home/thachpham/git\_example/.git/

Ở đoạn trên, nó hiển thị dòng thông báo mình đã khởi tạo một kho Git trống tại đường dẫn như trên. Lưu ý rằng thư mục ẩn .git/ là nơi nó sẽ chứa các thiết lập về Git cũng như lưu lại toàn bộ thông tin về kho chứa, bạn **không cần đụng chạm gì vào thư mục .git/** này.

Nếu kho chứa của bạn đã có sẵn mã nguồn thì bạn cần **phải đưa các tập tin về trạng thái Tracked** nhằm có thể làm việc được với Git. Để làm việc này, bạn sẽ cần sử dụng lệnh git add tên\_file, có thể sử dụng dấu \* để gom toàn bộ. Sau đó có thể sử dụng lệnh git status để xem danh sách các tập tin đã được tracked.

**$** git add readme.txt

**$** git status

On branch master

Initial commit

Changes to be committed:

(use "git rm --cached <file>..." to unstage)

new file: readme.txt

Và sau khi tập tin đã được đưa vào trạng thái tracked và nếu một tập tin đã tracked thì nó phải được đưa vào lại Staging Area (giải thích ở bài sau) cũng bằng lệnh git add thì bạn mới có thể tiến hành ủy thác (**commit**) nhằm lưu lại bản chụp các thay đổi. Lệnh commit sẽ có cấu trúc git commit -m "Lời nhắn", lúc này tất cả các tập đang trong trạng thái tracked (file mới) hoặc một tập tin đã được tracked nhưng có một sự thay đổi mới thì sẽ được commit.

$ git commit -m "First Commit"

[master (root-commit) 799db56] First Commit

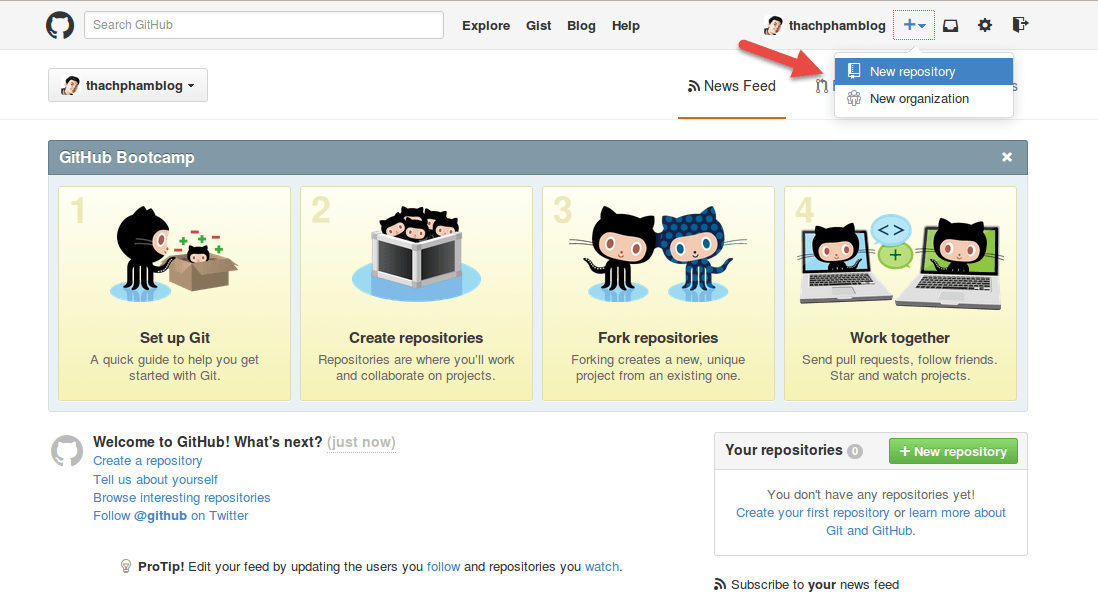
1 file changed, 0 insertions(+), 0 deletions(-)

create mode 100644 readme.txt

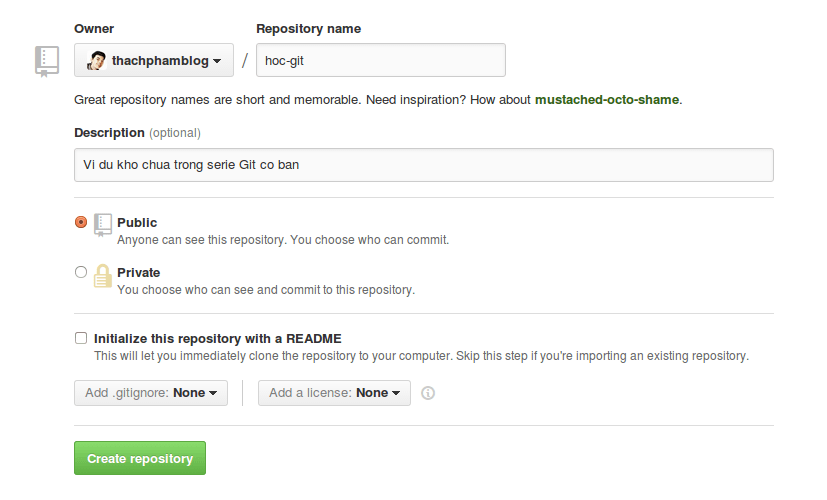
Bây giờ thì bạn đã hoàn thành việc commit lần đầu tiên các tập tin mà bạn đã đưa vào kho, mình sẽ nói kỹ hơn về việc commit ở các bài sau. Tóm lại là tới đây bạn đã có một kho chứa mã nguồn Git trên máy của bạn.

**Tạo repository trên Github và làm việc**

Trước tiên bạn cần đăng nhập vào [Github](https://github.com/), sau đó ấn vào dấu + trên menu và chọn *New repository*.



Bạn sẽ cần đặt tên cho kho chứa của bạn. Bạn có thể chọn loại kho chứa là **Public** (ai cũng có thể clone) và **Private** (chỉ có những người được cấp quyền mới có thể clone).



*Điền thông tin khi tạo Repository trên Github*

Khi tạo xong nó sẽ dẫn bạn tới trang hướng dẫn làm việc với kho chứa vừa tạo. Và kho chứa của bạn bây giờ sẽ có địa chỉ là https://github.com/$user-name/$repository, ví dụ https://github.com/thachphamblog/hoc-git.

Việc của bạn bây giờ là hãy clone cái kho chứa này về máy của mình bằng lệnh git clone địa\_chỉ.

**$** git clone https://github.com/thachphamblog/hoc-git

Cloning into 'hoc-git'...

warning: You appear to have cloned an empty repository.

Checking connectivity... done

Bây giờ hãy truy cập vào thư mục working tree (thư mục vừa clone repository về) và thử tạo ra một file tên là *README.md*, sau đó dùng lệnh git add để đưa file này vào Staging Area.

**$** cd hoc-git

**$** echo "# Huong dan Git co ban" > README.md

**$** git add README.md

**$** git commit -m "First commit on Github"

[master (root-commit) 6e729a4] First commit on Github

1 file changed, 1 insertion(+)

create mode 100644 README.md

Tuy nhiên sau khi commit xong, tập tin đã được commit sẽ vẫn không thể xuất hiện trong kho chứa trên Github mà bạn phải làm thêm một việc nữa đó là dùng lệnh git push để đẩy các tập tin đã được commit lên Github. Lưu ý rằng **bạn sẽ cần nhập tài khoản và mật khẩu Github**.

**$** git push origin master

Counting objects: 3, done.

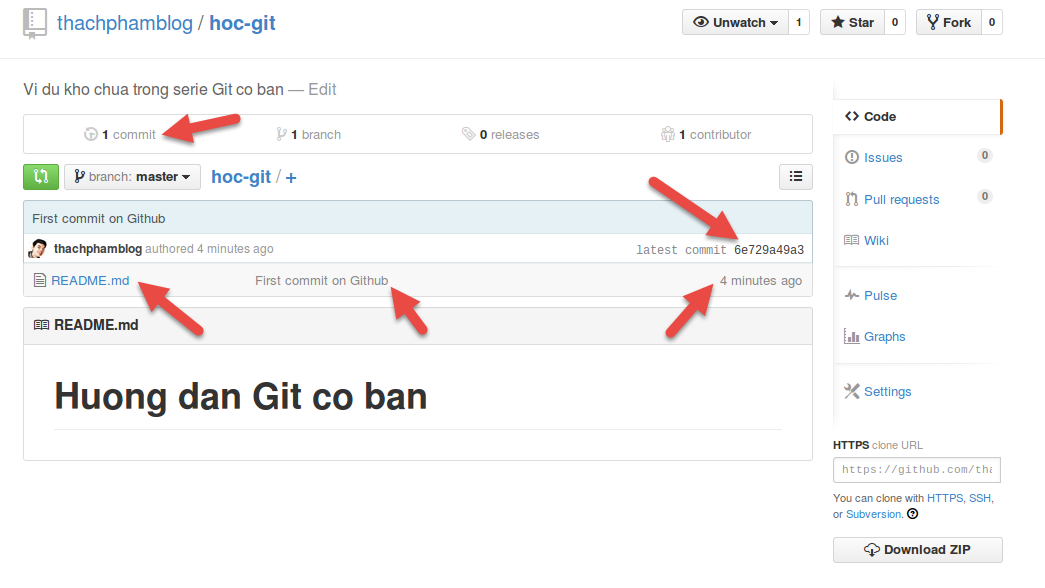
Writing objects: 100% (3/3), 244 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/thachphamblog/hoc-git

\* [new branch] master -> master

origin nghĩa là tên remote (xem ở bài sau) và master là tên branch, hai cái này mình sẽ giải thích kỹ hơn ở bài riêng của nó. Bây giờ bạn có thể kiểm tra kho chứa của bạn trên Github rồi đó.



*Kết quả sau khi push mã nguồn đã được commit lên Github*

Có thể bạn sẽ thấy mỗi khi push lên Github thì nó hiển thị ra thông báo quá dài dòng, bạn có thể cho ẩn các thông báo đi mà chỉ hiển thị dòng khai báo username và password trên Github thì có thể thiết lập với lệnh này.

git config --global push.default simple

Tuy nhiên nếu bạn không thích bị hỏi mật khẩu nữa thì có thể sử dụng SSH với Github, mình sẽ nói cái này sau.

**Lời kết**

Bây giờ bạn đã biết cách tạo một repository cho Git rồi chứ? Thật đơn giản phải không nào, bây giờ để tiếp tục các bài sau dễ dàng hơn, hãy thực hành tạo một reposity trên máy tính và tập commit, hoặc bạn có thể sử dụng Github để tập đưa mã nguồn của mình vào kho chứa cho thuần thục nhé.

# [Git] Hiểu thêm về Commit và Staging Area

Trong bài [tạo repository cho Git](https://thachpham.com/tools/cach-tao-repository-cho-git.html) mình có nhắc qua về cụm từ **Staging Area** và một tính năng là **commit** (ủy thác), vậy hai cái này là gì thì mình sẽ giải thích kỹ hơn trong bài này để bạn biết cách sử dụng cho đúng.

**Staging Area là gì?**

*Staging Area* nghĩa là một khu vực mà nó sẽ được chuẩn bị cho quá trình commit. Trước hết, bạn cần phải hiểu rằng trong các hệ thống quản lý phiên bản (Version Control System) thì các dữ liệu sẽ được lưu trữ ở hai nơi, một là thư mục bạn đang làm việc trên máy tính (working tree, mình không nhắc lại nữa đâu) và một là kho chứa mã nguồn (repository) sau khi bạn đã thực hiện thay đổi (ví dụ như kho chứa trên [Github](https://github.com/)).

Nhưng với Git thì nó có thêm một lựa chọn nữa đó là có thêm một khu vực trung gian gọi là **Staging Area** và đây chính là một lợi thế lớn của Git. Staging Area nghĩa là khu vực sẽ lưu trữ những thay đổi của bạn trên tập tin để nó có thể được commit, vì muốn commit tập tin nào thì tập tin đó phải nằm trong Staging Area. Một tập tin khi nằm trong Staging Area sẽ có trạng thái là **Stagged** (xem thêm ở dưới).



*Mô hình giải thích cách hoạt động của Staging Area.*

Và để đưa một tập tin vào Staging Area thì bạn sẽ cần phải sử dụng lệnh git add tên\_file mà mình đã có ví dụ ở phần trước.

**Commit là gì và nó hoạt động ra sao?**

Hiểu đơn giản hơn, commit nghĩa là một hành động để Git lưu lại một bản chụp (snapshot) của các sự thay đổi trong thư mục làm việc, và các tập tin và thư mục được thay đổi đã phải nằm trong Staging Area. Mỗi lần commit nó sẽ được lưu lại lịch sử chỉnh sửa của mã nguồn kèm theo tên và địa chỉ email của người commit. Ngoài ra trong Git bạn cũng có thể khôi phục lại tập tin trong lịch sử commit của nó để chia cho một phân nhánh (branch) khác, đây là mấu chốt của việc bạn sẽ dễ dàng khôi phục lại các thay đổi trước đó mà mình có giới thiệu qua ở phần giới thiệu serie này.

Và tất nhiên, lệnh commit trong Git sẽ là git commit -m "Lời nhắn".

Và nếu bạn **muốn đưa tập tin lên repository thì bạn phải commit nó trước** rồi sau đó lệnh git push origin master sẽ có nhiệm vụ đưa toàn bộ các tập tin đã được commit lên repository.

**Điều kiện gì để commit một tập tin?**

Nếu bạn muốn commit một tập tin đó, bạn sẽ cần phải đưa tập tin đó vào trạng thái tracked bằng lệnh git add tên\_file. Trong git có hai loại trạng thái chính đó là Tracked và Untracked, cụ thể:

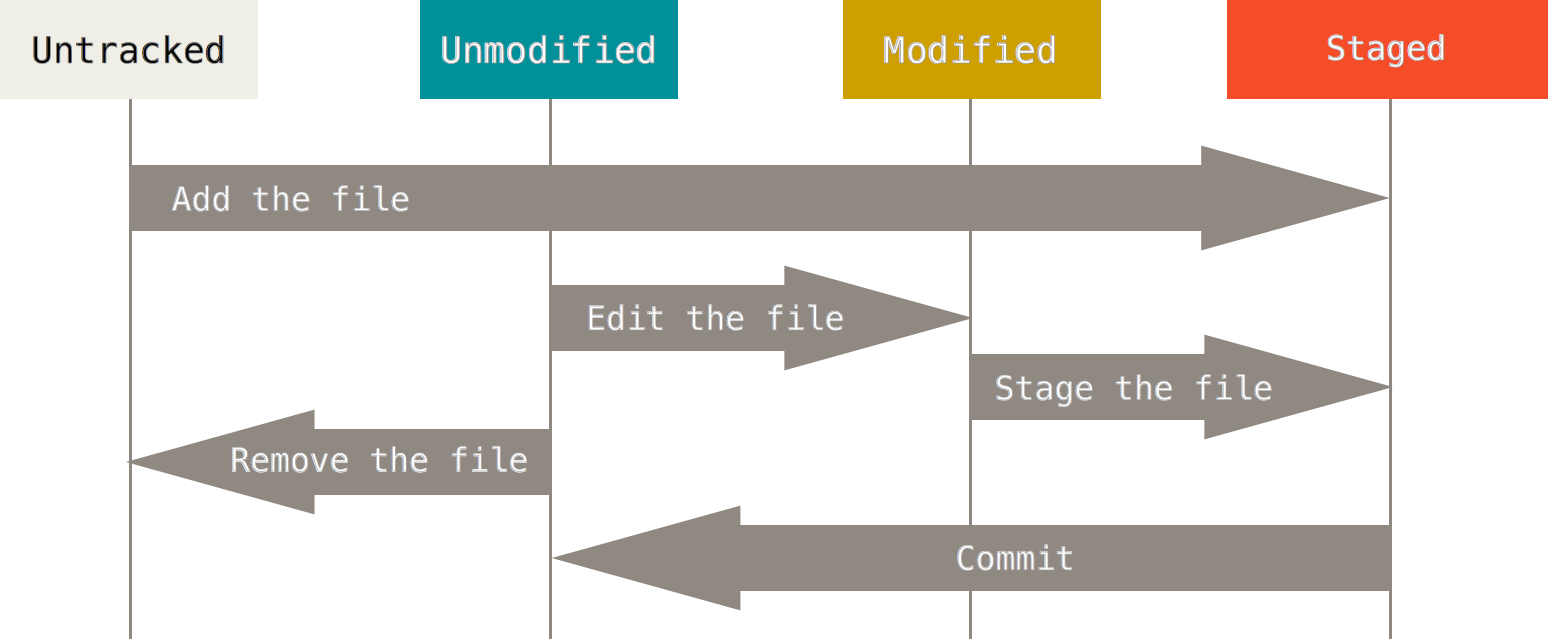
* **Tracked** – Là tập tin đã được đánh dấu theo dõi trong Git để bạn làm việc với nó. Và trạng thái Tracked nó sẽ có thêm các trạng thái phụ khác là Unmodified (chưa chỉnh sửa gì), Modified (đã chỉnh sửa) và Staged (đã sẵn sàng để commit).
* **Untracked** – Là tập tin còn lại mà bạn sẽ không muốn làm việc với nó trong Git.

Nhưng bạn phải nên biết rằng nếu tập tin đó đã được Tracked nhưng đang rơi vào trạng thái (Modified) thì nó vẫn sẽ không thể commit được mà bạn phải đưa nó về Staged cũng bằng lệnh git add.

**Bỏ qua Staging Are để commit**

Như mình có nói ở trên là một tập tin sau khi được thay đổi hay tạo mới thì nó phải được thêm vào Staging Area với lệnh git add. Tuy nhiên, bạn có thể đưa một tập tin đã được Tracked để commit mà không cần đưa nó vào Staging Area với tham số -a trong lệnh git commit. Ví dụ: git commit -a -m "Skipped Staging Are to commit".

**Tìm hiểu thêm về trạng thái**



*Vòng đợi trạng thái của các tập tin*

**Untracked**

Nếu bạn tạo ra hoặc thêm vào một tập tin mới vào trong thư mục làm việc của bạn thì nó sẽ ở trạng thái Untracked. Bây giờ mình thử tạo ra một tập tin mới tên là *faq.html*, sau đó dùng lệnh git status để xem trạng thái của Git trong thư mục làm việc.

**$** touch faq.html

**$** git status

On branch master

Your branch is up-to-date with 'origin/master'.

Untracked files:

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

faq.html

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

***Note***: Lệnh touch là tạo ra một tập tin rỗng.

Bây giờ bạn sẽ thấy nó đã liệt kê ra tên tập tin đang ở trạng thái Untracked. Để đưa nó về Tracked bạn sẽ sử dụng lệnh git add và xem lại trạng thái của nó.

**$** git add faq.html

**$** git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

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

new file: faq.html

Bây giờ bạn thấy, tập tin faq.html của mình đã được đưa về trạng thái Staged và nó có thể được commit. Tại sao? Vì bạn phải biết rằng nếu một tập tin ở trạng thái Untracked mà được đưa về Tracked thì nó sẽ nằm ở trạng thái Staged luôn, trừ khi bạn thay đổi nội dung tập tin này thì nó sẽ đưa về trạng thái Modified và nó không thể commit trừ khi bạn gõ lệnh git add cho nó.

**Tracked**

Một khi một tập tin đã được đưa về Tracked thì nó sẽ có thể thay đổi giữa 3 trạng thái khác nhau là **Modified**, **Unmodified** và **Staged**.

Trước hết bây giờ mình đã có một tập tin mới đã được đưa về Staged với lệnh git add như ví dụ trên. Bây giờ mình tiến hành thay đổi nội dung của tập tin faq.html này và xem kết quả của lệnh git status.

**$** git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes to be committed:

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

new file: faq.html

Changes not staged for commit:

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

(use "git checkout -- <file>..." to discard changes in working directory)

modified: faq.html

Bạn có thấy sự kỳ lạ ở tập tin *faq.html* không? Đó là nó được hiển thị ở hai trạng thái Staged (có thể commit) và Modified (không thể commit) hay còn gọi là Unstaged. Sở dĩ có sự kỳ lạ đó ở đây là bởi vì trước đó bạn đã tạo ra tập tin *faq.html* và đưa về Tracked thì nó cũng đã được đưa về Staged để có thể commit. Tuy nhiên sau đó bạn lại chỉnh sửa nội dung của nó nên nó đã có một phiên bản khác nằm ở trạng thái Modified (không thể commit). Nếu bây giờ bạn gõ lệnh git commit để ủy thác nó thì bản chụp của tập tin faq.html ở lần cuối cùng bạn gõ lệnh git add sẽ được commit lên chứ nó không chứa các nội dung mà bạn vừa thêm vào. Và để nó có thể commit tập tin faq.html đã được chỉnh sửa thì bạn phải gõ lại lệnh git add faq.html lần nữa.

**Chuyển tập tin từ Untracked về Tracked**

Trong Git, bạn có thể đưa một tập tin từ Tracked về Untracked với lệnh rm tên\_file. Lệnh rm sẽ giúp bạn đưa tập tin về trạng thái Untracked nhưng không xóa hẳn trong ổ cứng.

**$** rm faq.html

**$** git status

On branch master

Your branch is up-to-date with 'origin/master'.

Changes not staged for commit:

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

(use "git checkout -- <file>..." to discard changes in working directory)

deleted: faq.html

Còn nếu bạn muốn xóa nó luôn thì dùng lệnh git rm -f tên\_file và***nhớ cẩn thận*** khi dùng lệnh này.

**Lời kết**

Có thể bạn sẽ thấy những gì mình nói trong bài này hơi dài nhưng đó là những kiến thức rất cơ bản về Git mà bạn cần nắm chắc vì nếu bạn không hiểu vòng đời các trang thái của một tập tin trong Git thì chắc chắn sau này khi làm việc bạn sẽ bối rối khi gõ lệnh git commit và cứ hỏi tại sao lại không commit được vì muốn commit được bạn sẽ phải đưa nó về trạng thái thích hợp, tức là trạng thái Staged.

# [Git] Git Log và Undo Commit

Một việc bạn sẽ khá thường xuyên làm trong Git nếu làm việc theo nhóm đó là kiểm tra xem những ai đã commit vào dự án bạn đang làm việc, cũng như cách bạn undo lần commit trước đó nếu như bạn cảm thấy mình thiếu một tập tin nào đó trong lần commit trước để bổ sung vào.

**Xem git log**

Để xem lịch sử của các lần commit trước đó, bạn sử dụng lệnh git log là sẽ thấy.

**$** git log

commit 3f1ef84ada3dfd936735d8724f9bbb3437c77b19

Author: Thach Pham <contact@thachpham.com>

Date: Tue Apr 21 17:16:37 2015 -0700

Hihi

commit 6e729a49a36b31919daa6263f8f98f3a59d5bab3

Author: Thach Pham <contact@thachpham.com>

Date: Tue Apr 21 14:47:47 2015 -0700

First commit on Github

Bạn thấy chứ? Mỗi lần commit sẽ có một checksum riêng, và nó cũng có ghi rõ ai là người commit vào và commit vào ngày bao nhiêu, lúc nào.

Ngoài ra, bạn có thể chèn thêm tham số -p vào để hiển thị chi tiết của mỗi lần commit.

**$** git log -p

commit 3f1ef84ada3dfd936735d8724f9bbb3437c77b19

Author: Thach Pham <contact@thachpham.com>

Date: Tue Apr 21 17:16:37 2015 -0700

Hihi

diff --git a/README.md b/README.md

index db7e814..e08f24f 100644

--- a/README.md

+++ b/README.md

@@ -1 +1 @@

-# Huong dan Git co ban

+hhih\n

diff --git a/faq.html b/faq.html

new file mode 100644

index 0000000..e69de29

commit 6e729a49a36b31919daa6263f8f98f3a59d5bab3

Author: Thach Pham <contact@thachpham.com>

Date: Tue Apr 21 14:47:47 2015 -0700

First commit on Github

diff --git a/README.md b/README.md

new file mode 100644

index 0000000..db7e814

--- /dev/null

+++ b/README.md

@@ -0,0 +1 @@

+# Huong dan Git co ban

*Note*: Nếu log quá dài, dùng phím mũi tên lên xuống để đọc tiếp và ấn Ctrl + Z để thoát.

Hoặc nếu bạn muốn chỉ muốn xem 1 lần commit gần nhất thì thêm tham số -1 vào.

Bạn còn có thể sử dụng thêm một số tùy chọn xem log sau để tối ưu hơn quy trình đọc log.

* --since, --after: Xem các lần commit kể từ ngày nhất định.
* --until: Xem các lần commit trước từ ngày nhất định.
* --author: Xem các lần commit của một người nào đó.
* --grep: Lọc các chuỗi trong log và in ra.

Ví dụ:

**$** git log --author=contact@thachpham.com --pretty="%s"

Hihi

First commit on Github

Cái --pretty là gì mình sẽ nói ở dưới.

**Lọc log với --pretty**

Tham số --pretty rất có ích nếu bạn muốn lọc xem một đối tượng nào đó trong lịch sử commit, ví dụ như chỉ xem lời nhắn commit hoặc chỉ xem email của người commit.

Cách sử dụng tham số --pretty là bạn phải viết kèm các tag của nó như sau:

**$** git log --pretty="%tag"

Các %tag phải dược đặt trong cặp dấu ngoặc kép và bạn có thể sử dụng nhiều %tag khác nhau.

Danh sách các %tag:

* %H –  Commit hash
* %h – Abbreviated commit hash
* %T – Tree hash
* %t – Abbreviated tree hash
* %P – Parent hashes
* %p – Abbreviated parent hashes
* %an – Author name
* %ae – Author e-mail
* %ad – Author date (format respects the –date=option)
* %ar – Author date, relative
* %cn – Committer name
* %ce – Committer email
* %cd – Committer date
* %cr – Committer date, relative
* %s – Subject

Ví dụ:

**$** git log --pretty="%an - %s"

Thach Pham - Hihi

Thach Pham - First commit on Github

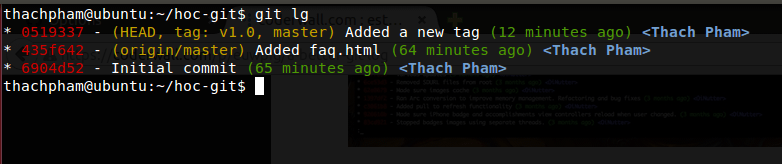
**Thủ thuật xem log đẹp hơn**

Bạn có thể thấy mỗi khi gõ git log mà không có các tham số kèm theo thì nó hiện ra rất khó nhìn. Tuy nhiên có một cách để bạn có thể làm gọn cái log mặc định, và thêm màu sắc cho nó để bạn dễ nhìn hơn.

Trước tiên bạn copy lệnh này để thiết lập.

git config --global alias.lg "log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit"

Bây giờ để xem log, bạn chỉ cần gõ git lg để xem. Bạn có thể kết hợp nó với các tham số ở trên. Cái quan trọng nhất là với cách xem này, bạn có thể thấy **HEAD** (con trỏ của branch hiện tại) mà sau này khi bạn tìm hiểu tới branch bạn sẽ thấy nó rất có ích.



**Undo Commit**

Nếu bạn cần xóa bỏ lần commit trước và cần undo để commit lại thì có thể sử dụng tham số --amend trong lệnh git commit.

**$** git log --pretty="%s"

Hihi

First commit on Github

**$** git commit --amend -m "Hehe"

[master 3682e56] Hehe

Date: Tue Apr 21 17:16:37 2015 -0700

2 files changed, 1 insertion(+), 1 deletion(-)

create mode 100644 faq.html

**$** git log --pretty="%s"

Hehe

First commit on Github

Lưu ý rằng undo nghĩa là bạn quay trở lại bước commit lần trước, do vậy nếu cần bổ sung tập tin nào vào để commit thì hãy đưa tập tin đó vào Staging Area trước.

**Bỏ tập tin ra khỏi Staging Area**

Nếu bạn đã đưa một tập tin nào đó vào Staging Area nhưng bây giờ bạn muốn loại bỏ nó ra khỏi đây để không phải bị commit theo thì có thể sử dụng lệnh git reset HEAD tên\_file. HEAD là cái gì thì tới phần phân nhánh (Branch) mình sẽ giải thích.

**Lời kết**

Trong bài này bạn đã biết cách thao tác để xem lại lịch sử của những lần commit để theo dõi các lần commit tốt hơn, cũng như cách bạn thao tác phục hồi liên quan đến việc Commit. Tuy nhiên, bạn có thấy là mình đã có nói qua về việc quản lý phiên bản trong Git chưa? Chưa có đúng không? Được rồi, vậy thì ở bài sau mình sẽ hướng dẫn các bạn làm quen với tính năng Tagging (đánh dấu) của Git và đây là cách để chúng ta quản lý từng phiên bản trong mã nguồn.

# [Git] Đánh dấu commit với Tag

Trong khi bạn commit các lần chỉnh sửa của mình thì mọi thứ sẽ được đều lưu vào log mà mình đã hướng dẫn bạn cách xem commit log ở [phần trước](https://thachpham.com/tools/git-git-log-va-undo-commit.html). Tuy nhiên nếu bạn commit quá nhiều thì sẽ gây khó khăn cho bạn về sau nếu cần xem lại thông tin của lần commit trước mà bạn có thể gắn thẻ đánh dấu (tag) cho mỗi commit và khi cần xem bạn chỉ cần sử dụng lệnh git show tên\_tag là đã có thông tin rất rõ ràng, ngoài ra nó còn giúp bạn dễ dàng diff (đối chiếu) sau này khi không cần nhớ checksum (dù chỉ cần nhớ vài ký tự đầu tiên) của mỗi commit mà chỉ cần nhớ tag, cũng như có thể tạo thêm branch từ tag để bạn thuận tiện hơn trong việc phân nhánh.

**Lightweight Tag và Annotated Tag**

Trong Git có hai kiểu tag chính đó là:

* **Lightweight Tag**: Các tag này chỉ đơn thuần là đánh dấu snapshot của commit.
* **Annotated Tag**: Với tag này, bạn có thể đặt tiêu đề cho tag, và khi xem nó sẽ có thông tin về người tag, ngày tag,….

**Cách tạo Lightweight Tag**

Trước tiên, bạn có thể gõ git tag để xem danh sách các tag có trong dự án của bạn. Sau đó để tạo thêm một tag, bạn có thể gõ git tag tên\_tag để tạo. Ví dụ như v1.0 chẳng hạn.

**$** git tag v1.0

**$** git tag

v1.0

Bây giờ bạn có thể xem thông tin của lần commit được gắn tag này bằng lệnh git show tên\_tag. Lưu ý rằng lệnh trên nó sẽ đánh dấu lần commit cuối cùng của bạn vào tag v1.0.

**$** git show v1.0

commit 05193375f7a7c1295fd26c6388d81e188f405b0b

Author: Thach Pham <contact@thachpham.com>

Date: Thu Apr 23 02:20:50 2015 -0700

Added a new tag

diff --git a/tag.html b/tag.html

new file mode 100644

index 0000000..e69de29

**Cách tạo Annotated Tag**

Để tạo Annotated Tag thì bạn cũng sử dụng lệnh git tag nhưng sẽ có thêm tham số -a và tham số -m để thiết lập lời nhắn cho tag này.

**$** git tag -a v1.0-an -m "Ra mat phien ban 1.0"

**$** git show v1.0-an

tag v1.0-an

Tagger: Thach Pham <contact@thachpham.com>

Date: Thu Apr 23 02:41:11 2015 -0700

Ra mat phien ban 1.0

commit d5a599e3385a8fc7a65958ed50bc8b54666b45ad

Author: Thach Pham <contact@thachpham.com>

Date: Thu Apr 23 02:40:31 2015 -0700

Commit for Annotated Tag

diff --git a/tag.html b/tag.html

index e69de29..fea03c1 100644

--- a/tag.html

+++ b/tag.html

@@ -0,0 +1 @@

+Annotated Tag

Bạn có thể thấy khi show ra, cái Annotated tag sẽ có nhiều thông tin hơn là so với cái tag thông thường, và đây cũng là kiểu tag bạn nên sử dụng để có nhiều thông tin hơn.

**Thêm tag cho các commit cũ**

Ở lệnh trên thì bạn chỉ tạo ra tag cho commit cuối cùng của bạn. Còn nếu bạn có rất nhiều commit trước đó mà cần gắn tag thì chỉ cần thêm mã checksum (hoặc một đoạn mã checksum) của lần commit đó.

Để xem mã checksum của các lần commit trước đó thì bạn có thể sử dụng git log với tham số --pretty với giá trị oneline để lọc log nhé.

**$** git log --pretty=oneline

d5a599e3385a8fc7a65958ed50bc8b54666b45ad Commit for Annotated Tag

05193375f7a7c1295fd26c6388d81e188f405b0b Added a new tag

435f642f951fbab1037fc2feef239ab26d6e6115 Added faq.html

6904d5232bf90821068279311e205e3e1ff929f1 Initial commit

Và bây giờ mình có thể đặt tag cho commit *Initial commit* thì sẽ khai báo một đoạn mã checksum của nó vào lệnh git tag như sau.

**$** git tag -a v0.0 6904d -m "Tag for inintial commit"

**$** git tag

v0.0

v1.0

v1.0-an

**Push Tag**

Mặc định lệnh git push sẽ không push các tag đã tạo lên repository mà bạn có thể dùng lệnh git push --tags để đẩy toàn bộ tag lên repository.

**$** git push --tags

Username for 'https://github.com': thachphamblog

Password for 'https://thachphamblog@github.com':

Counting objects: 7, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (6/6), done.

Writing objects: 100% (7/7), 775 bytes | 0 bytes/s, done.

Total 7 (delta 1), reused 0 (delta 0)

To https://github.com/thachphamblog/hoc-git.git

\* [new tag] v0.0 -> v0.0

\* [new tag] v1.0 -> v1.0

\* [new tag] v1.0-an -> v1.0-an

Nếu bạn dùng Github hay các dịch vụ tương tự thì bây giờ bạn sẽ thấy tag xuất hiện ở repository.



**Nhập tag vào branch**

Mặc dù ở bài sau mình sẽ giải thích qua branch nhưng tạm thời bây giờ bạn có thể hiểu branch là **một phân nhánh trong một cây dự án** để bạn sửa mã nguồn mà không ảnh hưởng đến phân nhánh gốc (master).

Bây giờ bạn có thể truy cập vào dữ liệu mà bạn đã commit thông qua tag kèm theo việc tạo một branch mới với lệnh git checkout -b tên\_branch tên\_tag. Ví dụ mình muốn đưa cái tag v1.0-an vào một branch mới tên version1 thì sẽ viết như sau:

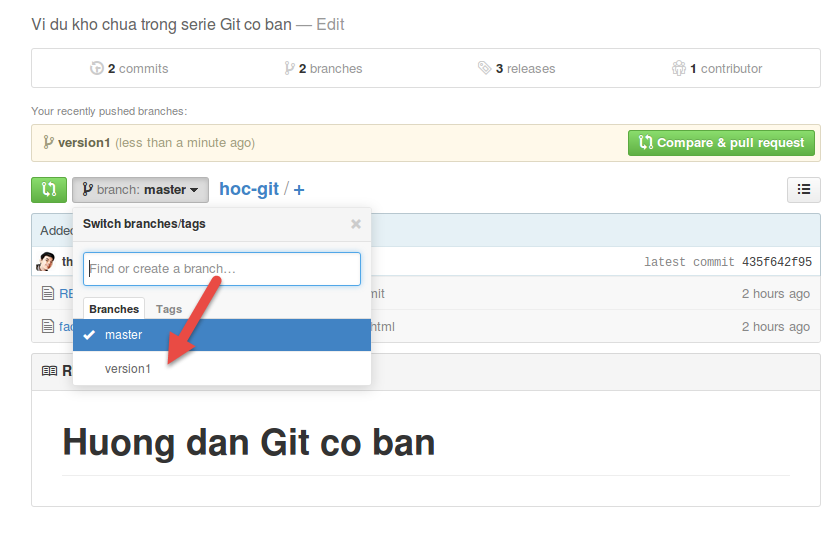
**$** git checkout -b version1 v1.0-an

Switched to a new branch 'version1'

Lúc này bạn đã tự động chuyển qua branch version1 thay vì master ban đầu, kèm theo đó là dữ liệu của lần commit được gắn tag v1.0-an.

Bây giờ nếu bạn muốn làm việc thêm với branch này thì cứ làm việc như khi bạn ở master. Để chuyển về lại master, bạn gõ lệnh git checkout master.

Để push cái branch này lên bạn có thể sử dụng lệnh git push origin version1. Nếu bạn quên thì mình nhắc lại nhé, origin chính là tên địa chỉ remote của repository mặc định mà Git tự đặt khi bạn clone.



Bây giờ bạn thử vào xem cái version1 và cái master xem có thấy sự khác nhau của nó không? Ở bài nói về branch mình sẽ giải thích cho bạn kỹ hơn. Đại loại là bạn tạo một phân nhánh khác để sửa code riêng mà không ảnh hưởng đến code chính.

**Lời kết**

Trong bài này chắc bạn đã thấy được sức mạnh của Git là như thế nào và khi nó kết hợp với branch thì sẽ rất tuyệt vời trong việc phân chia nhánh cho mã nguồn để bạn làm việc linh hoạt hơn, dễ dàng chuyển đổi qua các phiên bản mà không cần làm gì nhiều. Bạn đừng lo lắng khi chưa hiểu rõ về branch, mình sẽ giải thích ngay ở phần sau đây.

# [Git] Sơ lược Remote Respository và Origin

Quay trở lại bài [tạo repository](https://thachpham.com/tools/cach-tao-repository-cho-git.html), mình có hướng dẫn bạn cách tạo một repository trên dịch vụ Github và Clone nó về máy. Như vậy, cái Github là một máy chủ repository từ xa nên mình sẽ gọi nó là Remote Repository, nghĩa là repository này không nằm trên máy tính của bạn.

Và ở phần đó, bạn có thấy mình kêu các bạn gửi dữ liệu lên repository bằng cách dùng lệnh git push origin master sau khi commit không? Cái master là tên branch mà mình sẽ không nói ở bài này, nhưng cái origin trong đoạn đó chính là **tên remote repository**. Mặc định khi clone một repository thì nó tự đặt tên là origin.

Để kiểm tra tên remote, bạn có thể gõ lệnh git remote -v.

**$** git remote -v

origin https://github.com/thachphamblog/hoc-git.git (fetch)

origin https://github.com/thachphamblog/hoc-git.git (push)

Trong đó bạn có thể thấy cái repository mình đã clone đều được đặt tên là origin, và mỗi repository bạn có hai đều có hai hành động là fetch (lấy dữ liệu về từ server) và push (gửi dữ liệu lên server).

Nhìn lại đoạn lệnh git push origin master ở trên, điều đó có nghĩa là bạn gửi tất cả các thay đổi trên mã nguồn ở máy bạn lên remote tên là origin với branch master.

**Đổi tên remote**

Nếu bạn không thích tên origin thì có thể đổi tên nó lại nó bằng tên khác cho dễ quản lý nếu như bạn có nhiều remote trong một dự án với lệnh git remote rename tên\_cũ tên\_mới. Ví dụ mình cần đổi từ origin sang thach thì sẽ đổi như sau:

**$** git remote rename origin thach

**$** git remote -v

thach https://github.com/thachphamblog/hoc-git.git (fetch)

thach https://github.com/thachphamblog/hoc-git.git (push)

Bây giờ khi commit hay push bạn có thể gõ git push thach master để gửi mã nguồn lên remote repository này.

**Thêm một remote**

Trường hợp bạn cần thêm một cái remote để lấy dữ liệu khi cần thì có thể sử dụng lệnh git remote add tên\_remote URL. Ví dụ mình cần remote một repository và đặt tên nó là inuit thì sẽ viết như sau:

**$** git remote add unuit https://github.com/thachpham92/inuit.css-web-template

**$** git remote -v

thach https://github.com/thachphamblog/hoc-git (fetch)

thach https://github.com/thachphamblog/hoc-git (push)

inuit https://github.com/thachpham92/inuit.css-web-template (fetch)

inuit https://github.com/thachpham92/inuit.css-web-template (push)

Sau đó nếu bạn muốn lấy dữ liệu từ cái inuit kia về thì chỉ cần sử dụng lệnh git fetch inuit.

**$** git fetch inuit

warning: no common commits

remote: Counting objects: 94, done.

remote: Total 94 (delta 0), reused 0 (delta 0), pack

Unpacking objects: 100% (94/94), done.

From https://github.com/thachpham92/inuit.css-web-te

\* [new branch] master -> inuit/master

Lưu ý là lệnh git fetch nó chỉ lấy về và lưu vào database của Git trên máy chứ không được gộp vào repository của bạn. Để gộp vào bạn có thể gõ thêm lệnh git merge inuit, trong đó inuit là tên remote.

Còn nếu bạn muốn nó lấy về trực tiếp mà không cần gộp thì sử dụng lệnh git pull tên\_remote, tuy nhiên mình khuyến khích bạn nên gộp vào branch khi cần và nhớ cẩn thận trong việc gộp, tốt nhất nên tạo thư mục mới trong thư mục làm việc của bạn rồi vào đó mà lấy về.

**Sự khác nhau giữaclone, fetch và pull**

Có thể bây giờ bạn đã biết được 3 lệnh để lấy dữ liệu về từ repository đó là git clone, git fetch và git pull. Nhưng cả ba loại đều là lấy dữ liệu, thế sự khác nhau của nó là gì?

**git clone**

Lệnh này sẽ sao chép toàn bộ dữ liệu trên repository và sao chép luôn các thiết lập về repository, tức là nó sẽ tự động tạo một master branch trên máy tính của bạn. Lệnh này chỉ nên sử dụng khi bạn cần tạo mới một Git mới trên máy tính với toàn bộ dữ liệu và thiết lập của một remote repository.

**git pull**

Lệnh này sẽ tự động lấy toàn bộ dữ liệu từ remote repository và gộp vào cái branch hiện tại bạn đang làm việc.

**git fetch**

Lệnh này sẽ lấy toàn bộ dữ liệu từ remote repository nhưng sẽ cho phép bạn gộp thủ công vào một branch nào đó trên thư mục Git ở máy tính.

Tạm thời bạn nên hiểu thế, ở bài branch bạn sẽ hiểu sâu hơn.

**Các loại giao thức của Remote Repository**

Chúng ta không chỉ kết nối với một remote repository qua giao thức HTTP hay HTTPS mà còn có thể chọn nhiều giao thức khác, dưới đây là một vài giao thức remote repository.

**Local Repository**

Giao thức này nghĩa là bạn kết nối tới một repository nào đó trên chính máy tính của bạn và URL của giao thức sẽ có dạng /path/repository/.

**HTTP Repository**

Giao thức thông dụng nhất cũng như dễ hiểu nhất, thường được sử dụng nếu bạn dùng các dịch vụ remote repository như Github hay Assembla, nó sẽ bao gồm định dạng http://domain.com/repository.git hoặc https://domain.com/repository.git.

**SSH Repository**

Giao thức này thường được dùng trên các nhu cầu tạo một server repository riêng và kết nối thông qua giao thức SSH. Đường dẫn của giao thức này sẽ có dạng là user@server:/path/repository.git. Ở phần cuối serie mình sẽ hướng dẫn bạn cách tạo một repository server riêng và nó là lựa chọn rất tốt nếu bạn làm việc nhóm mà không cần phụ thuộc vào các dịch vụ như Github hay Assembla.

**Lời kết**

Trong bài này chắc bạn đã có thể thấy được Git linh hoạt như thế nào khi cho phép bạn lấy dữ liệu từ một remote repository hoặc nhiều repository khác nhau. Mặc khác, mình cũng muốn giải đáp cho các bạn có thắc mắc cái origin trên câu lệnh là gì vì trong các bài sau mình sẽ còn dùng khá nhiều câu lệnh có chứa origin trong đó nữa.

# [Git] Branch – Kỹ thuật phân nhánh

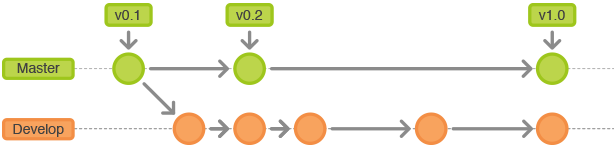
Trong khi làm việc với Git, bạn đã quá quen thuộc với việc chỉnh sửa mã nguồn, sau đó là commit mỗi khi chỉnh sửa xong và push lên remote repository nếu cần thiết. Nhưng bây giờ mình có một ví dụ đặt ra là mình muốn **tạo một phiên bản thử nghiệm với mã nguồn đang làm việc trong working tree hiện tại mà không gây ảnh hưởng đến các code hiện tại**. Vậy thì làm cách nào? Không lẽ clone một repository từ chính cái working tree hiện tại rồi sửa đổi hay sao? Như thế rất là mất công, mà lại không tối ưu và không thể đồng bộ hóa hoặc rất khó khăn để đồng bộ hóa.

Mà trong Git, chúng ta sẽ sử dụng một giải pháp khác tuyệt vời hơn, dễ dàng hơn gọi là **phân nhánh** (*branching*), mà cụ thể là phân nhánh cái gì? Đó là phân nhánh trong working tree hiện tại bạn đang làm việc đấy, và mỗi nhánh chúng ta sẽ gọi nó là một **branch**.

**Branch trong Git là gì?**

Khi bắt đầu khởi tạo một repository hoặc clone một repository, bạn sẽ có một nhánh (branch) chính tên là master (bạn có thể hiểu master là một cái thân cây). Đây là branch mà sẽ chứa toàn bộ các mã nguồn chính trong repository.

Đó là lý do tại sao, ở các phần trước khi chúng ta push hoặc pull hay làm một số việc khác thì lại có tham số master trong câu lệnh, đó nghĩa là chúng ta đang thực hiện thao tác trên branch master.



*Nhánh Master là nhánh chính mặc định trong working tree. Bạn có thể tạo ra một (hoặc nhiều nhánh mới) với tên là Develop chẳng hạn.*

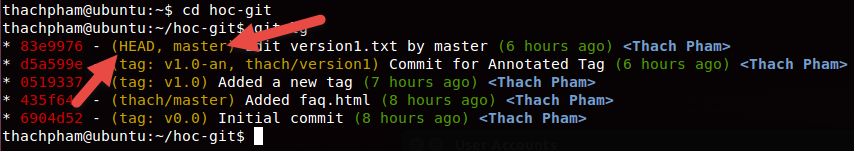
Bây giờ nếu bạn muốn tạo một sự thay đổi nào đó mà có thể trực tiếp sử dụng trên working tree hoặc commit, push lên repository mà không ảnh hưởng đến branch master thì sẽ cần tạo ra một branch mới với tên là develop chẳng hạn. Và từ đó mỗi khi bạn thực hiện lệnh checkout vào branch nào đó thì toàn bộ mã nguồn trên working tree của bạn sẽ được đổi sang môi trường dành cho branch đang checkout.

Giờ mình lấy một ví dụ đơn giản, bên branch master bạn tạo một tập tin *master.html* rồi commit lên, lúc này khi bạn qua cái branch develop (đã được tạo trước đó) thì cái tập tin *master.html* kia sẽ không có vì nó đã được commit bởi branchmaster, không liên quan gì tớidevelop. Tương tự, các thay đổi của bạn bên develop cũng sẽ không ảnh hưởng gì tới bên master cả. Điều này có một cái thú vị là nó tương tác trực tiếp trên máy tính của bạn, ví dụ tập tin index.html của master có nội dung khác và index.html bên develop thì khi bạn dùng lệnh checkout là nội dung trên máy tính nó tự đổi tương ứng, đó là lý do bạn có thể test nhiều phiên bản trên máy tính mà không cần đổi thư mục, chỉ cần checkout cái branch.

Giải thích thì hơi dài dòng, cứ vào làm ví dụ cụ thể cho dễ hiểu nhé.

**HEAD – con trỏ vị trí**

Trước khi nói tiếp về branch thì mình xin nói qua một xíu vềHEAD. Trong Git, từ khóa HEAD sẽ tượng trưng cho con trỏ chỉ cho bạn biết bạn đang nằm ở đâu. Bây giờ hãy xem lại bài hướng dẫn đọc log commit của mình, tìm tới phần tối ưu log và cài vào. Sau đó gõ lệnh git lg thì bạn sẽ thấy từ khóa HEAD cho bạn biết bạn đang ở đâu trong working tree.



Như ảnh trên thì có nghĩa là mình đang ở branchmaster.

Ngoài ra còn một cách khác để kiểm tra đó là đọc tập tin .git/HEAD

**$** cat .git/HEAD

ref: refs/heads/master

**Cách tạo một branch**

Trước tiên bạn có thể xem toàn bộ các branch mà bạn đang có trong working tree bằng lệnh git branch. Sau đó nếu muốn tạo thêm branch, chỉ cần gõ lệnh git branch tên\_brand. Ví dụ mình cần tạo branchdevelop.

**$** git branch develop

Bây giờ bạn có thể gõ lại lệnh git branch một lần nữa để xem sẽ thấy brand tên develop xuất hiện.

**Checkout một branch**

Checkout ở đây nghĩa là bạn truy cập kiểm tra mã nguồn trong branch đó để làm việc đấy. Để làm việc này, bạn sử dụng lệnh git checkout tên\_branch.

**$** git checkout develop

Switched to branch 'develop'

Lúc này bạn đã đổi sang branch develop rồi, để kiểm tra chắc ăn bạn có thể gõ các lệnh kiểm tra HEAD ở trên.

Bây giờ bạn sẽ làm việc trong branch mới chuyển hay nói đúng hơn là bạn đang làm việc ở chỗ mà cái HEAD đang trỏ tới. Để chuyển về branch chính thì gõ git checkout master.

Bây giờ bạn thử tạo một tập tin nào đó, sau đó commit ở branch develop rồi chuyển về branch master sẽ thấy những gì bạn đã làm ở branch develop hoàn toàn vô nghĩa ởmaster. Dưới đây là ví dụ về việc ở branch master không có tập tin develop.html được tạo ra từ branch develop.

**$** **touch develop.html**

**$** **git add .**

**$** **git commit -m "Commit develop.html from develop branch"**

[develop 8c68896] Commit develop.html from develop branch

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

create mode 100644 develop.html

**$** **git checkout master**

Switched to branch 'master'

Your branch is ahead of 'thach/master' by 3 commits.

(use "git push" to publish your local commits)

**$** **ls**

faq.html README.md tag.html version1.txt

Tương tự với việc sửa tập tin hay bất kỳ làm việc gì khác nó cũng chỉ áp dụng thay đổi ở branch bạn đang trỏ tới.

**Gộp dữ liệu từ một branch**

Nếu mỗi branch nó nằm riêng như vậy thì bạn muốn sử dụng các thay đổi ở một branch nào đó cho master thì sao? À, chúng ta có thể sử dụng lệnh git merge để chuyển dữ liệu từ một branch nào đó về branch mà bạn đang trỏ đến. Lưu ý là ở branch cần chuyển về đã phải được commit. Ví dụ mình cần chuyển dữ liệu từ branch develop về master thì sẽ làm lần lượt các lệnh sau:

**$** **git checkout master**

Already on 'master'

Your branch is ahead of 'thach/master' by 3 commits.

(use "git push" to publish your local commits)

**$** **git merge develop**

Updating 83e9976..8c68896

Fast-forward

develop.html | 0

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

create mode 100644 develop.html

**$** **ls**

develop.html faq.html README.md tag.html version1.txt

**Xóa branch**

Nếu bạn không cần dùng tới branch nào nữa thì có thể xóa với lệnh git branch -d tên\_branch. Lưu ý là cái branch muốn xóa phải được gộp dữ liệu (merge) về master.

**$** **git branch -d develop**

Deleted branch develop (was 8c68896).

Sau khi xóa xong, nó sẽ báo branch đó đã được móc vào commit với mã checksum nào (8c68896).

**Làm việc với remote branch**

Quay lại một xíu với bài remote repository, bây giờ bạn hãy tạo thêm một remote mới từ địa chỉ https://github.com/csswizardry/inuit.css và đặt tên cho remote này là inuit vào working tree của bạn.

**$** **git remote add inuit https://github.com/csswizardry/inuit.css**

**$** **git remote -v**

inuit https://github.com/csswizardry/inuit.css (fetch)

inuit https://github.com/csswizardry/inuit.css (push)

thach https://github.com/thachphamblog/hoc-git.git (fetch)

thach https://github.com/thachphamblog/hoc-git.git (push)

Bây giờ bạn có thể xem toàn bộ branch của cái remote inuit mới thêm vào bằng lệnh git remote show inuit.

**$** **git remote show inuit**

\* remote inuit

Fetch URL: https://github.com/csswizardry/inuit.css

Push URL: https://github.com/csswizardry/inuit.css

HEAD branch: master

Remote branches:

feature/docs new (next fetch will store in remotes/inuit)

feature/grids new (next fetch will store in remotes/inuit)

fix/beautons new (next fetch will store in remotes/inuit)

fix/grids new (next fetch will store in remotes/inuit)

gh-pages new (next fetch will store in remotes/inuit)

incoming new (next fetch will store in remotes/inuit)

master new (next fetch will store in remotes/inuit)

v5.0.1 new (next fetch will store in remotes/inuit)

v5.1 new (next fetch will store in remotes/inuit)

v5.1.0 new (next fetch will store in remotes/inuit)

Bây giờ bạn có thể chọn một cái remote branche cần fetch dữ liệu về. Ví dụ bây giờ mình sẽ tạo một branch mới cho working tree của mình tên làfix\_ui, sau đó nạp dữ liệu trong branch gh-pages của inuit thì mình sẽ lần lượt làm như sau.

**$** **git branch fix\_ui**

**$** **git checkout fix\_ui**

Switched to branch 'fix\_ui'

**$** **git pull inuit gh-pages**

warning: no common commits

remote: Counting objects: 309, done.

remote: Compressing objects: 100% (33/33), done.

remote: Total 309 (delta 4), reused 2 (delta 2), pack-reused 274

Receiving objects: 100% (309/309), 135.63 KiB | 0 bytes/s, done.

Resolving deltas: 100% (106/106), done.

From https://github.com/csswizardry/inuit.css

\* branch gh-pages -> FETCH\_HEAD

\* [new branch] gh-pages -> inuit/gh-pages

Auto-merging README.md

CONFLICT (add/add): Merge conflict in README.md

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

Có thể nó sẽ xảy ra lỗi conflict khi gộp file README.md cũng không quan trọng lắm nên bạn có thể bỏ qua bằng cách gõ lệnh git add README.md để track file này, vì bây giờ nếu bạn gõ lệnh ls ra thì đã có các file từ inuit rồi.

Bây giờ bạn có thể commit nó và thử push nó lên repository của bạn.

**$** **git commit -m "Commit from fix\_ui"**

[fix\_ui c1eb301] Commit from fix\_ui

**$** **git push thach fix\_ui**

Username for 'https://github.com': thachphamblog

Password for 'https://thachphamblog@github.com':

Counting objects: 317, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (201/201), done.

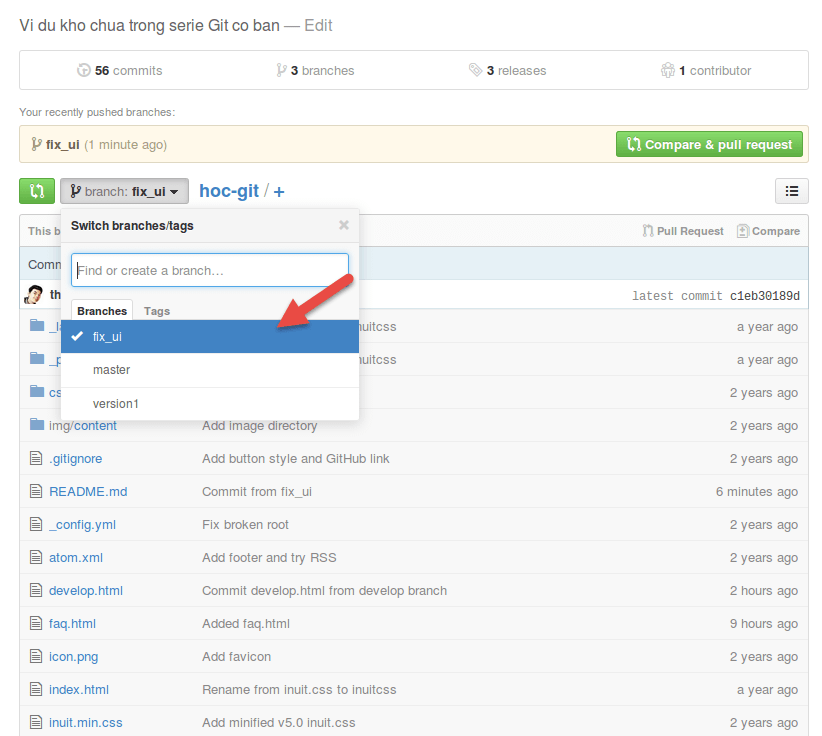
Writing objects: 100% (317/317), 135.77 KiB | 0 bytes/s, done.

Total 317 (delta 113), reused 304 (delta 106)

To https://github.com/thachphamblog/hoc-git.git

\* [new branch] fix\_ui -> fix\_ui

Nhắc lại rằng, thach nghĩa là tên remote của repository mà mình cần push lên và fix\_ui là tên branch mình cần push. Và đây là kết quả sau khi push lên repository của mình.



Và dĩ nhiên, bạn cũng có thể qua một branch khác và sử dụng lệnh git merge để gộp dữ liệu của branch này về.

Ngoài ra, kiến thức về branch còn có một kỹ thuật nữa cũng khá thú vị mà khi làm việc nhóm có thể sẽ cần đến, đó là **rebase branch** với mục đích hoán đổi vị trí của những lần commit. Tuy nhiên ở đây là serie cơ bản nên mình sẽ không nói qua mà bạn chỉ cần sử dụng thành thạo các kỹ thuật mà mình đã liệt kê trong đây là được.

**Lời kết**

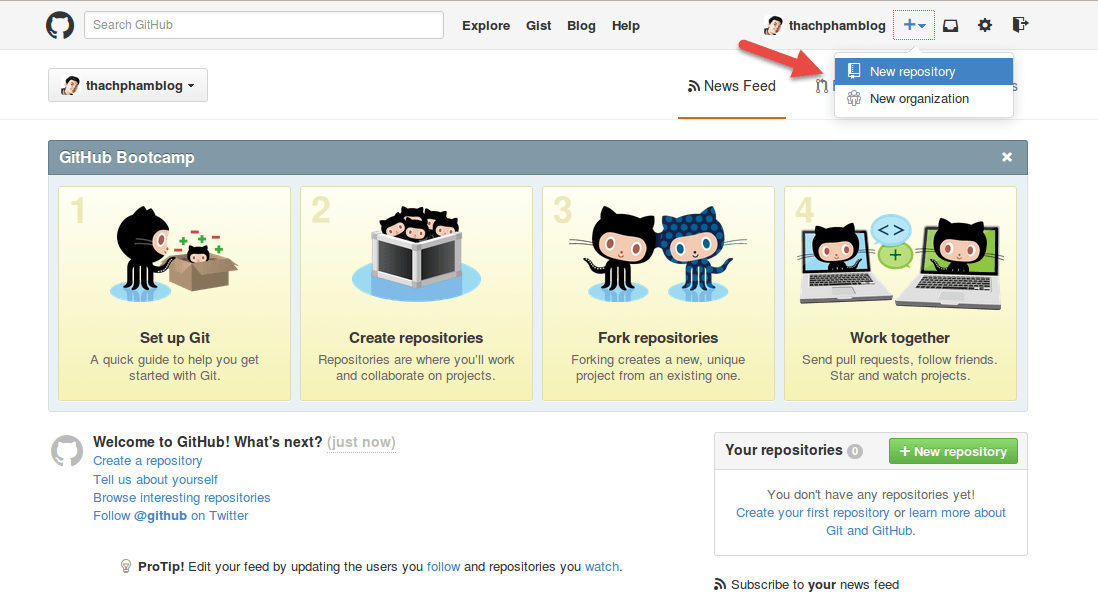
Vậy là tới đây mình đã kết thúc hướng dẫn xong các kỹ thuật quan trọng nhất trong Git mà bạn dù bất cứ sử dụng Git trong mục đích nào cũng phải sử dụng. Mình cũng đã hướng dẫn cặn kẽ nhất theo cách hiểu của mình theo hướng đơn giản nhất nên mình hy vọng bạn có thể hiểu nó. Tiếp tục ở bài sau, mình sẽ không nói qua về các kỹ thuật trong Git nữa mà mình sẽ giới thiệu cho các bạn một số dịch vụ máy chủ repository dành cho Git tốt nhất và đặc biệt là có gói miễn phí để bạn có thêm nhiều lựa chọn.

# [Git] Các dịch vụ repository miễn phí tốt nhất

Nếu không nói đến việc tự làm một repository server thì có lẽ phương án sử dụng các dịch vụ tạo repository dành cho Git là tối ưu nhất vì có giao diện dễ sử dụng, dễ thao tác mà lại có thể hỗ trợ các kiểu private repository (kho chứa riêng tư) để không công khai mã nguồn của bạn.

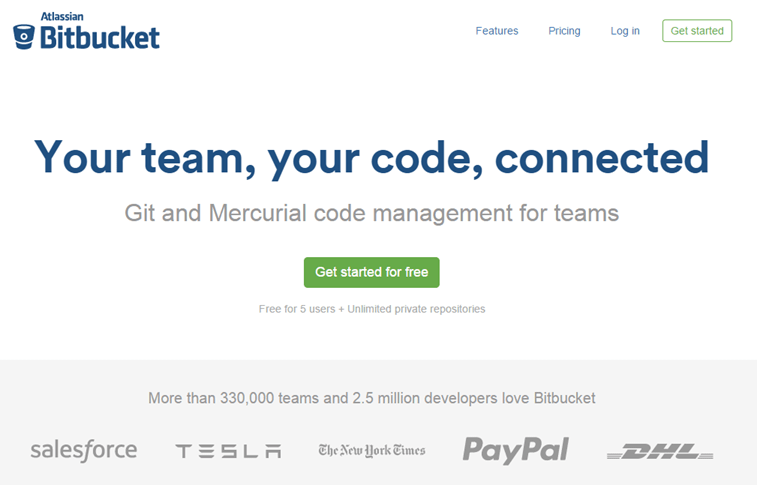
Dưới đây là một số dịch vụ repository có lựa chọn sử dụng miễn phí tốt nhất mà bạn nên sử dụng qua, trong đó có Github vốn đã quen thuộc với rất nhiều người.

**1.**[**Github**](https://github.com/)



Dịch vụ này (có thể được gọi là một mạng xã hội dành cho lập trình viên) có lẽ hầu như bạn đã đều biết qua vì sự phổ biến của nó. Với tài khoản miễn phí, bạn có thể tạo ra các repository dạng công khai và sử dụng. Ngoài ra, bạn có thể xem các repository của người khác trong phần Explore với rất nhiều mã nguồn hay và độc đáo, đây được xem là nét riêng của Github vì nhiều người tham gia vào đây không phải chỉ để dùng Git thôi đâu.

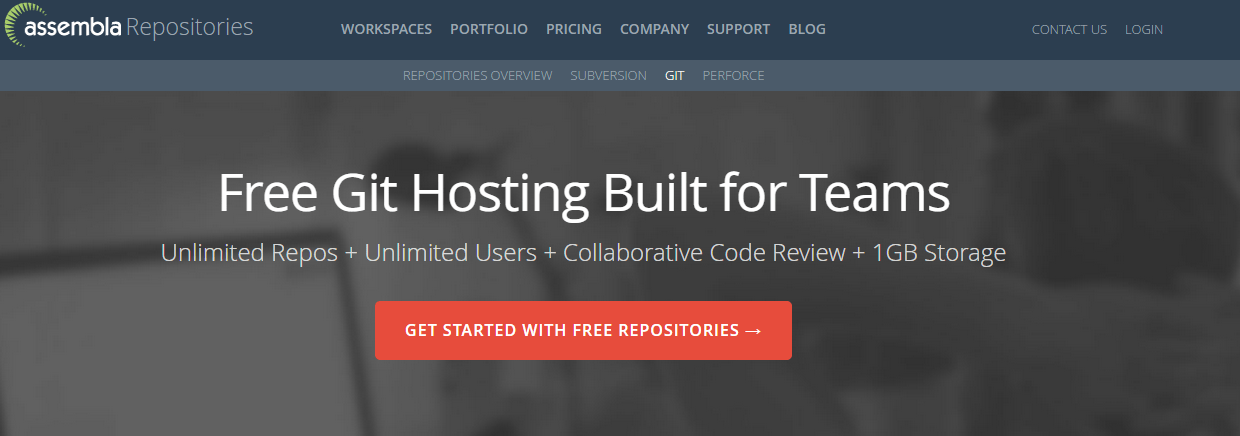
**2.**[**Bitbucket**](https://bitbucket.org/)



Nếu bạn cần server Git dành cho một nhóm ít hơn 5 người làm việc với nhau thì bạn có thể sẽ dùng miễn phí vĩnh viễn ở Bitbucket. Vì tài khoản miễn phí ở đây bạn sẽ được quyền tạo private repository không giới hạn nhưng sẽ giới hạn 5 thành viên được quyền truy cập vào private repository.

Các tính năng ở Bitbucket hầu như đầy đủ cho nhu cầu sử dụng Git, và ngay cả Thachpham.com cũng đã từng sử dụng Bitbucket trong việc nâng cấp website của mình nên thấy rất hài lòng.

**3.**[**Assembla**](https://www.assembla.com/home)



Trong khi Github và Bitbucket sẽ miễn phí không giới hạn repository tạo ra nhưng Assembla lại bị giới hạn 2 repository ở tài khoản miễn phí, nhưng nếu bạn cần lựa chọn thêm giao thức FTP/sFTP để gửi dữ liệu lên repository thì Assembla là lựa chọn cho bạn. Và tất cả repository tạo ra từ Assembla đều là private.

**Lời kết**

Chỉ với 3 lựa chọn ở trên thôi mình nghĩ là đã quá đủ cho bạn rồi mà đa phần hiện nay được sử dụng nhiều nhất vẫn là Github hoặc Bitbucket. Bạn cũng yên tâm rằng tất cả dịch vụ Git Repository đều có cách sử dụng giống nhau nên nếu khác chỉ là hạn mức sử dụng cho từng loại tài khoản.

Ở bài sau, cũng là bài cuối của serie, mình sẽ hướng dẫn bạn cách tự tạo một Repository Server trên máy chủ Linux của riêng bạn để sử dụng thoải mái và hoàn toàn bảo mật.

# [Git] Cách tự tạo một Repository Server

Trong một vài trường hợp cá biệt, có thể bạn sẽ không muốn nâng cấp tài khoản của bạn tại các dịch vụ Repository Hosting từ miễn phí lên trả phí để có thể tạo ra không giới hạn các kho chứa mã nguồn, mà bạn muốn tự tạo cho mình một Repository Server để thoải mái tạo ra bao nhiêu kho chứa cũng được. Thật vậy, bạn có thể thuê một [**VPS tại AZDIGI**](https://azdigi.com/thue-server-vps/) chỉ với 150.000/tháng thì bạn đã có thể tạo ra một Repository Server thật tốt và đủ để sử dụng cho team của bạn từ nhỏ tới trung bình.

**Chuẩn bị**

Một Linux Server, có thể là hệ điều hành nào cũng được nhưng nên sử dụng các hệ điều hành phổ biến và bạn dễ thao tác là được, ví dụ như Ubuntu, CentOS, Debian, Fedora,…Và bạn phải sử dụng user root.

Trong bài này thì mình sẽ sử dụng hệ điều hành Ubuntu 14.04 32-bits.

Nếu bạn tự cài đặt server thì hãy chắc chắn rằng đã cài đặt SSH và mở cổng SSH (mặc định là 22).

**Cài đặt Git**

Trước tiên, bạn hãy cài đặt Git vào server với lệnh:

**CentOS/RHEL/Fedora**

**$** yum install git

**Ubuntu/Debian**

**$** apt-get install git

Sau đó gõ git để kiểm tra xem có hiện nội dung hướng dẫn không, nếu có thi thành công.

**Tạo Repository**

Quy trình tạo Repository trên Server cũng giống như bài tạo repository mình đã viết trước đó. Nhưng để an toàn hơn, mình khuyến khích bạn làm theo từng bước của mình.

**Bước 1. Tạo user truy cập vào repository**

Bước này chúng ta sẽ tạo ra một user để khi kết nối từ máy tính cá nhân sẽ dùng nó chứ không nên kết nối thẳng vào user root nhé.

Ví dụ mình tạo ra một user tên là git\_user và.

**$** useradd git\_user

***Hiếm gặp***: Trường hợp khi tạo user nó không tự tạo ra thư mục riêng cho user này tại */home/git\_user* thì bạn có thể tạo thủ công bằng tài khoản root với lệnh mkdir /home/git\_user và dùng lệnh chown -R git\_user:git\_user /home/git\_user để cấp quyền thư mục này cho user *git\_user*.

Và thiết lập mật khẩu cho *git\_user* với lệnh passwd git\_user, lưu ý là mật khẩu này bạn chỉ nên sử dụng cho riêng bạn để truy cập vào user *git\_user* nhanh chứ khi remote repository ở máy tính cá nhân thì sẽ dùng SSH cho bảo mật mà không cần gõ lại user/pass khi push.

**Tham khảo**: [Quản trị & Tạo user trên Linux](https://thachpham.com/hosting-domain/vps-basic-quan-ly-user-va-sudo.html)

Thiết lập mật khẩu xong, hãy gõ lệnh su – git\_user để chuyển qua user git\_user và gõ cd ~ để quay về thư mục gốc của nó để làm việc (các thao tác ở dưới đều làm trên user git\_user này).

**Bước 2. Thiết lập SSH cho git\_user**

Bước này chúng ta sẽ thiết lập cho phép người dùng bên ngoài kết nối vào thư mục */home/git\_user* bằng giao thức SSH cho bảo mật và thuận tiện.

**Tham khảo**: [Bảo mật máy chủ với SSH Key](https://thachpham.com/hosting-domain/huong-dan-ssh-key.html)

Tạo thư mục *.ssh* cho user này.

**$** mkdir ~/.ssh/

Kế tiếp là tạo thêm tập tin *authorized\_keys* trong thư mục *.ssh/* của nó.

**$** touch ~/.ssh/authorized\_keys

Bây giờ hãy CHMOD thư mục*~/.ssh/* thành *700*.

**$** chmod 700 ~/.ssh/

Sau đó CHMOD file *authorized\_keys* thành *600*.

**$** chmod 600 ~/.ssh/authorized\_keys

**Bước 2. Tạo repository trên server**

Hãy chắc chắn là bạn đang truy cập vào user *git\_user* và đang ở thư mục gốc của user này.

Bây giờ sẽ tạo một tập tin .git cho repository bằng lệnh sau:

**$** git init --bare thachpham.git

Initialized empty Git repository in /home/git\_user/thachpham.git/

Xong roài đó.

**Kết nối với repository từ máy tính**

Để kết nối vào repository *thachpham.git* mà chúng ta vừa tạo, bạn phải tạo ra một SSH Key riêng và thêm nó vào tập tin *authorized\_keys* trong thư mục *.ssh* của *git\_user*.

Để tạo SSH Key, bạn mở Terminal (hoặc Git Bash ở Windows) lên và gõ lệnh sau:

**$** ssh-keygen -t rsa -C "Key cua Thach"

Generating public/private rsa key pair.

Enter file in which to save the key (/home/thachpham/.ssh/id\_rsa):

Enter passphrase (empty for no passphrase):

Enter same passphrase again:

Your identification has been saved in /home/thachpham/.ssh/id\_rsa.

Your public key has been saved in /home/thachpham/.ssh/id\_rsa.pub.

The key fingerprint is:

ce:50:14:df:7b:5e:a5:e2:1f:1e:af:66:79:56:57:87 Key cua Thach

The key's randomart image is:

+--[ RSA 2048]----+

| o. |

| . . . |

| . . . ..|

| . .E.+|

| . S o o +|

| + . + .o|

| o . +.o|

| o++o|

| o++.|

+-----------------+

Sau đó ở máy tính, bạn gõ lệnh cat ~/.ssh/id\_rsa.pub để xem public key và copy nó, sau đó dán vào file *authorized\_keys* trong thư mục */home/git\_user/.ssh/* trên server. Lưu ý là mỗi key sẽ là một dòng nếu sau này bạn thêm nhiều key vào.

Sau đó, bạn có thể clone repository trên server với lệnh sau.

**$** git init && git remote add origin git\_user@123.456.78.9:thachpham.git

Initialized empty Git repository in /home/thachpham/thachpham\_git/.git/

Trong đó,

* git\_user: Tên user sở hữu thư mục git trên server bạn đã tạo
* 123.456.78.9: là IP của server
* thachpham.git: Tên repository đã tạo trên server.

Và bây giờ bạn có thể thử commit và push một tập tin lên server.

**$** touch README.md

**$** git add .

**$** git commit -m "Initial commit"

[master (root-commit) 104f7b7] Initial commit

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

create mode 100644 README.md

thachpham@ubuntu:~/thachpham\_git$ git push origin master

Counting objects: 3, done.

Writing objects: 100% (3/3), 215 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To git\_user@46.101.154.122:thachpham.git

\* [new branch] master -> master

Nếu bạn lên server kiểm tra thì trong thư mục repository sẽ không thấy tập tin của bạn vừa push lên. Bạn có thể test thử bằng cách tạo một thư mục khác, sau đó clone lại repository này trên server sẽ thấy tập tin được kéo về.

**Lời kết**

Tới đây thì bạn cũng đã nắm khá rõ về cách hoạt động của Git, cũng như là biết cách tự tạo một repository server dành cho Git rồi nhỉ? Và đây cũng là bài cuối cùng trong serie Git cơ bản này và mình hy vọng là nó có thể giúp ích cho bạn nhiều trong việc sử dụng Git. Về video thì mình sẽ làm sau nhé.