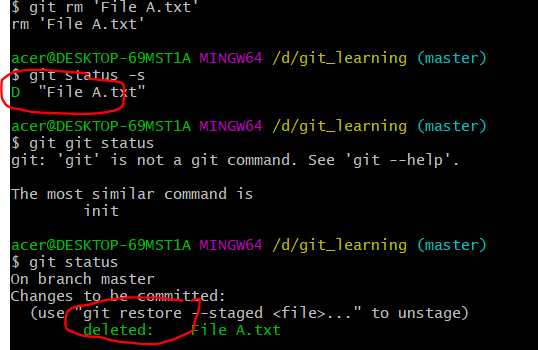
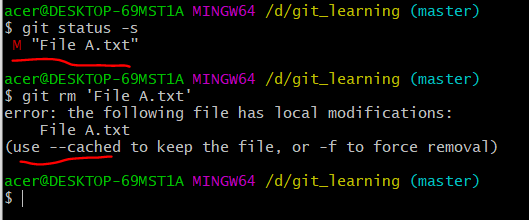
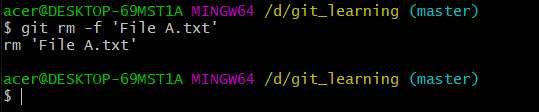
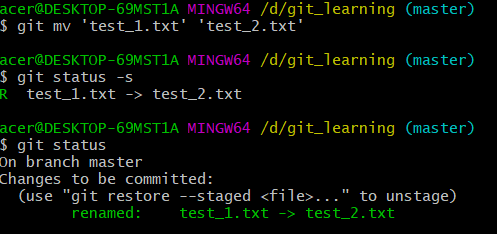
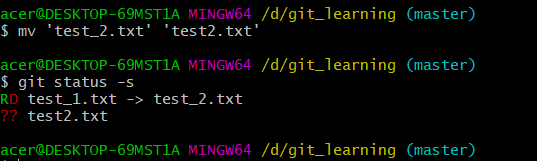
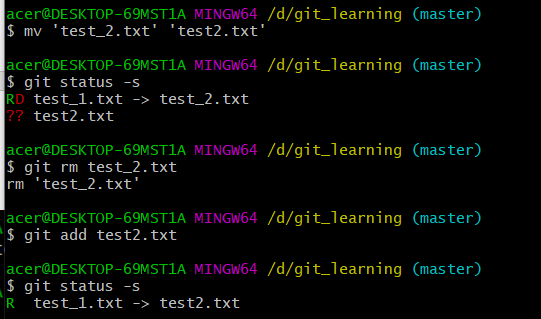
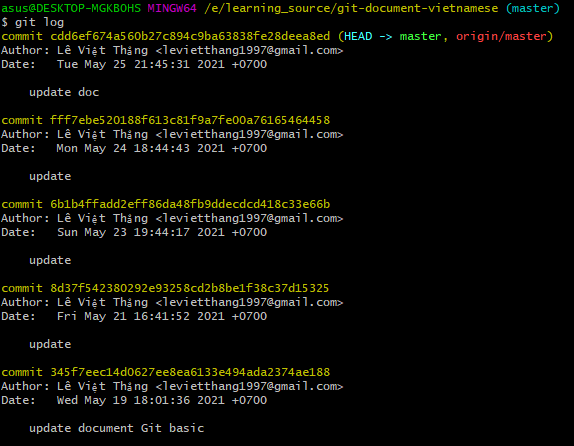
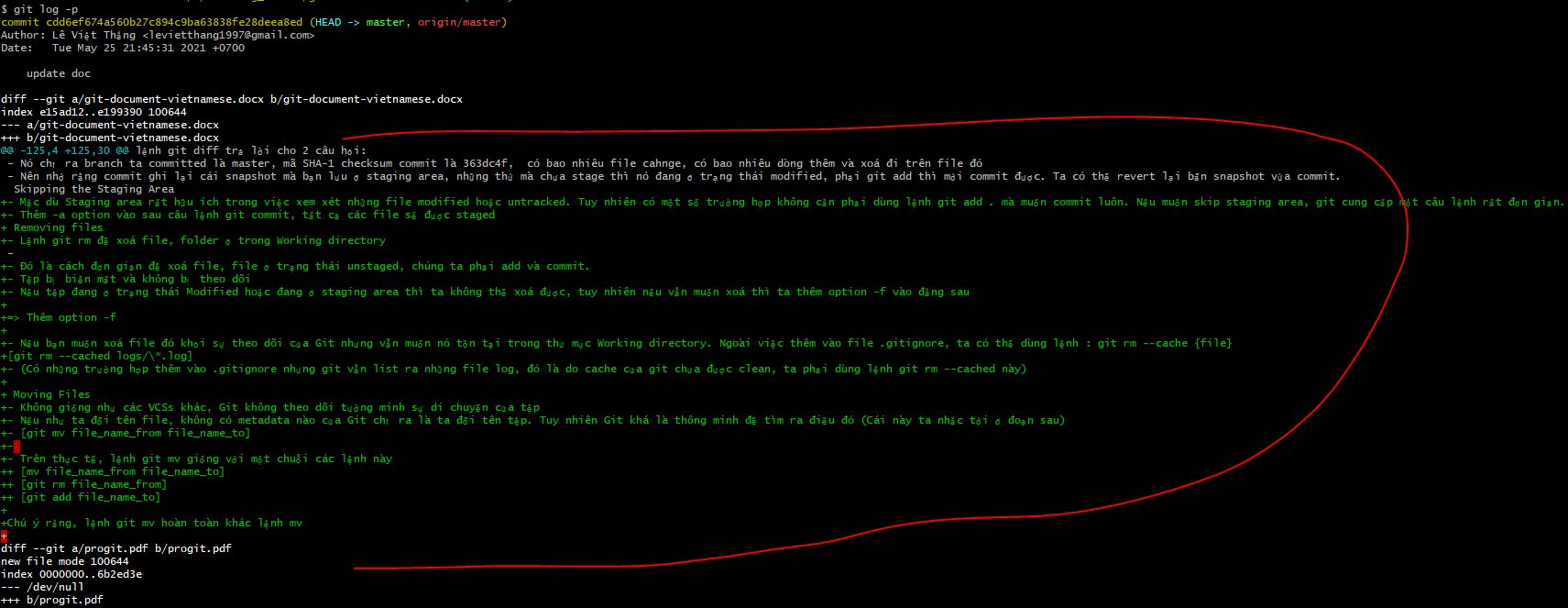
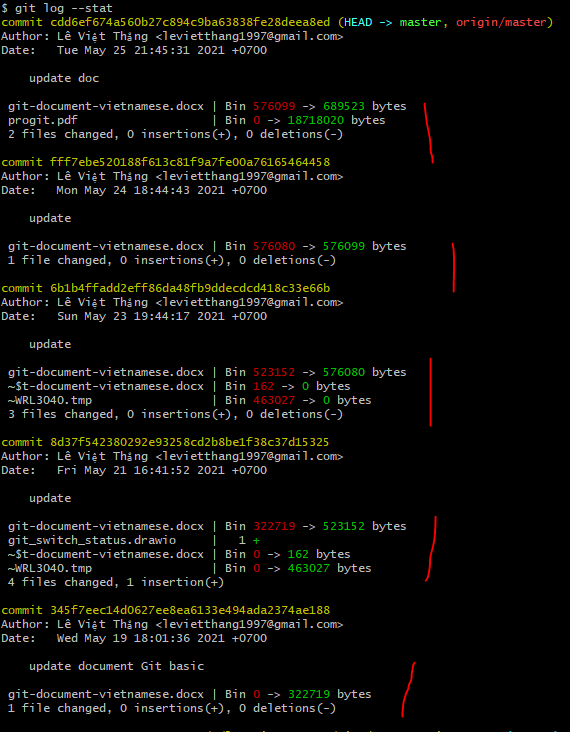
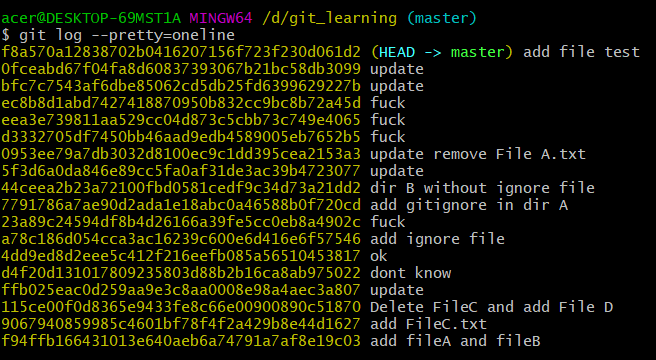
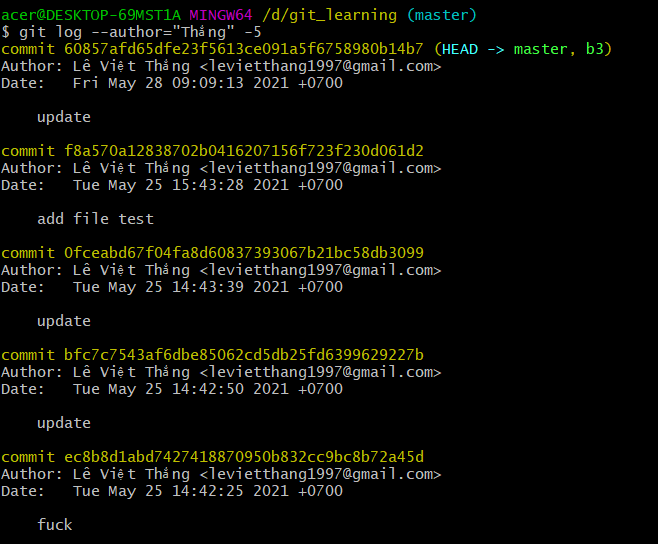
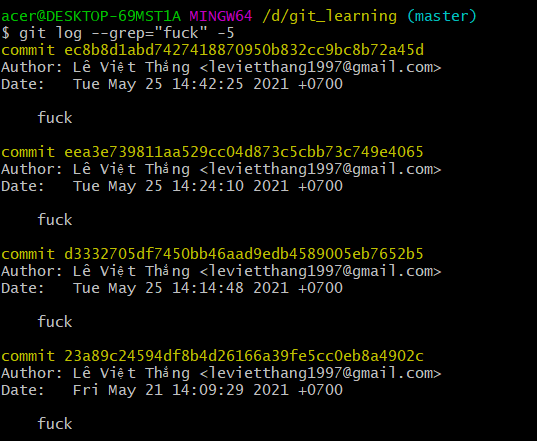
# Git master

1. Version Control là gì
   1. Version control là một hệ thống lắng nghe sự thay đổi của file hoặc một tập các files mọi lúc mọi nơi vì vậy ta có thể gọi các phiên bản của file. Với Version Control, ta có thể revert file về trạng thái trước đó, revert toàn bộ project về project trước đó, so sánh sự thay đổi mọi lúc, xem xem ai là người chỉnh sửa cuối cùng, ai tạo ra bug và khi nào và lúc nào. Nếu bạn làm hỏng hoặc mất dữ liệu thì có thể khôi phục lại được.
2. Local Version Control System : Nhiều người chọn phương pháp kiểm soát phiên bản của project là copy file sang thư mục khác (nếu thông minh thì thư mục đóng dấu theo thời gian). Cách tiếp cận này rất đơn giản và phổ biến tuy nhiên nó rất dễ xảy ra lỗi. Bạn quên mất bạn đang ở thư mục nào, hoặc là viết, chỉnh sửa sai file hoặc sao chép qua lại nhầm giữa các file. Để giải quyết vấn đề này, các lập trình viên đã phát triển VCS local nó có database đơn giản giữ tất cả các thay đổi của file dưới sự kiểm soát của revision control  
   - Một trong những VCS Tool phổ biến nhất là RCS, nó vẫn được sử dụng cho tới bây giờ
3. Centralized VCS (Quản lý vcs tập trung) (CVCS)  
   - Vấn đề đặt ra bây giờ là ta phải phát triển hệ thống cùng với các developer khác, Centralized Version Control System đã được tạo ra. Hệ thống này là 1 single server chứa tất cả các phiên bản của file. Tất cả các máy tính lấy file từ trung tâm đó.  
     
   - Thiết lập này mang nhiều lợi thế, đặc biệt là trên local VCS. Ở mức độ nhất định, Admin có quyền kiểm soát chi tiết xem ai có thể làm gì, việc quản lý CVCS dễ dàng hơn nhiều so với xử lý cơ sở dữ liệu local trên client.  
   - Tuy nhiên thiết lập này có một vài nhược điểm quan trọng. Điểm nổi bật nhất chính là máy chủ tập trung. Nếu nó bị chết trong vài giờ, không ai có thể lưu các thay đổi của các file dữ liệu mà họ đang làm việc. Hoặc nếu như ổ đĩa của con máy chủ bị hỏng, các bản sao lưu cần thiết không được lưu trữ thì ta mất hoàn toàn tất cả mọi thứ, may mắn thì có người nhanh chân sao lưu lại được dữ liệu trên máy cục bộ của họ.
4. Distributed Version Control System (DVCS)  
   - Đây là hệ thống kiểm soát phiên bản phân tán. Ở trong DVCS (giống như Git, Mercurial, Bazaar, Darcs), client không chỉ xem latest snapshot of the file, họ có thể xem kho lưu trữ và toàn bộ lịch sử của nó. Do đó nếu có bất kỳ server nào bị chết, bất kỳ client repository có thể copy dữ liệu để restore server mới. Mỗi bản sao là 1 bản sao lưu đầy đủ các dữ liệu.  
   
5. Git  
   - Hầu hết các hệ thống lưu thông tin dưới dạng file-based changes. Các hệ thống này nghĩ đến các thông tin chúng lưu trữ dưới dạng một tập hợp tệp và những thay đổi được thực hiện từng tệp theo thời gian  
     
   Git thì không làm như vậy. Trong thực tế, Git nghĩ dữ liệu của nó như một chuỗi các ảnh chụp nhanh của một hệ thống tệp thu nhỏ. Với Git, mỗi khi ta commit, hoặc lưu cái state của dự án, Git sẽ chụp ảnh tất cả các tệp của bạn trông như thế nào tại thời điểm đó và lưu trữ tham chiếu tới ảnh chụp nhanh đó. Để hiệu quả, nếu tệp không thay đổi, Git sẽ không lưu lại tệp, chỉ là một liên kết tới tệp giống hệt tệp trước đó mà nó đã được lưu trữ. Git nghĩ về dữ liệu của nó giống như một luồng các ảnh chụp nhanh  
     
   - Đây là điểm khác biệt quan trọng giữa Git và tất cả các VCS khác. Nó khiến Git xem xét lại hầu hết các khía cạnh của Version Control mà hầu hết các hệ thống copy từ phiên bản trước đó. Điều này làm cho Git giống như một hệ thống tệp nhỏ với một số công cụ cực kỳ mạnh mẽ được xây dựng lên, thay vì chỉ đơn giản là một VCS.
6. Mọi hoạt động với Git gần như đều là Local  
   - Hầu hết các hoạt động trong Git chỉ cần file Local và resources để hoạt động - không cần thông tin cần thiết từ một máy tính khác trên mạng của bạn. Toàn bộ lịch sử của dự án đều lưu trữ trên Local Disk nên nó gần như là tức thời. Ví dụ để mở lịch sử của dự án, Git không cần tới server lấy lại lịch sử và show cho bạn, nó chỉ cần lấy từ local database. Nếu bạn muốn so sánh sự thay đổi của 1 tệp với chính tệp đó ở thời điểm 1 tháng trước, git có thể truy vấn và tính toán sự thay đổi của tệp đó mà không cần phải kết nối đến server.
7. Tính toàn vẹn của Git (Git Integrity)  
   - Mọi thứ ở Git được checksummed trước khi nó được lưu trữ, sau đó được tham chiếu tới checksum đó. Điều đó có nghĩa là không thể thay đổi nội dung của bất kỳ tệp nào mà Git không biết về nó. Chức năng này được tích hợp vào Git ở cấp độ thấp nhất và không thể thiếu trong triết lý của nó. - - Cơ chế mà Git sử dụng để checksumming là SHA-1 hash. Ta có thể thấy hash value này ở khắp nơi trong Git vì nó sử dụng SHA rất nhiều. Trên thực tế, git lưu trữ thông tin file trên database không phải bằng file name mà nó là hash content of file.
8. Git thường chỉ thêm dữ liệu (Git generally only add data)  
   - Khi mà ta có một số hoạt động ở trong Git, gần như tất cả chúng chỉ Add data vào Database. Thật khó để khiến hệ thống làm điều gì mà nó không thể hoàn tác hoặc là xoá mất dữ liệu bằng bất kỳ cách nào. Đối với các VCS khác, bạn có thể mất hoặc xáo trộn các thay đổi mà mình chưa thực hiện. Nhưng nếu tạo 1 snapshot ở trong Git, nó rất khó bị mất đặc biệt nếu ta thường xuyên push database sang repository khác.
9. The three states  
   - Git có 3 trạng thái chính cho tệp là : modified, staged và committed.  
   + Modified được hiểu là ta đã thay đổi cái file (modified) nhưng mà chưa sử dụng lệnh git add và git commit.  
   + Staged : đánh dấu sẽ commit phiên bản hiện tại của tập tin đã chỉnh sửa trong lần commit sắp tới. Trạng thái này xảy ra khi sử dụng lệnh git add   
   + Committed được hiểu là data đã được lưu an toàn ở trong Database, đã commit thành công  
   - Dự án git có 3 phần chính : Working tree, Staging area, Git directory  
     
   - Working directory là 1 single checkout duy nhất của project. Các file này được kéo ra từ Compressed database (cơ sở dữ liệu nén ) ở trong Git directory và được đặt trên disk để ta sử dụng và sửa đổi. Nếu không dùng remote repository, thì đây là thư mục chính của dự án, branch master là bản chính, các branch khác là bản sao.  
   - Staging area là 1 file, được chứa trong Git directory. Nó chứa thông tin về trạng thái của các file trong 1 dự án.  
   - Git directory là cái nơi mà Git lưu trữ metadata và object database trong dự án. Nó là phần quan trọng nhất của Git. Nó tiếp nhận và lưu trữ commit từ staging area  
   - Quy trình cơ bản của Git (Git workflow) thường diễn ra ntn:   
   + Bạn thay đổi file ở trong working tree  
   + Bạn chỉ chọn lọc những thay đổi mà bạn muốn nó trở thành một phần của next commit. Nó thêm những thay đổi đó vào Staging Area.  
   + Bạn thực hiện commit, Nó lấy các tệp ở trong Staging Area và lưu vĩnh viễn cái snapshot của file đó ở Git directory.  
   - Nếu như một phiên bản cụ thể của tệp nằm trong Git directory, nó được gọi là committed. Nếu nó được thay đổi và được lưu ở Staging Area, nó gọi là Staged. Và nếu như nó được thay đổi từ khi nó được checked out nhưng mà chưa được staged thì gọi là modified
10. Lần đầu cài đặt Git  
    - [git config] : get và set tất cả các biến cấu hình kiểm soát tất cả các khía cạnh về cách Git look và các hoạt động của nó. Các biến này được lưu ở 3 nơi lưu trữ khác nhau đó là:   
    + [/path]/etc/gitconfig (ở windows là : C:\Program Files\Git\etc\gitconfig): Nó có chứa giá trị config có thể áp dụng cho bất kỳ user nào trong hệ thống và tất cả các repository của họ. Nếu thêm tuỳ chọn --system theo sau git config, nó đọc và ghi từ tệp này. Bởi vì sử dụng System configuration file, bạn cần phải là admin hoặc superuser để có quyền thay đổi nó.  
    + [~/.gitconfig hoặc ~/.gitconfig/git/config] (ở windows là C:\User\{$USER}\.gitconfig) file: Giá trị config chỉ dành riêng cho bạn, cho user đó. Bạn có thể làm Git đọc và ghi vào file đó bằng cách thêm --global option. Config này ảnh hưởng tới tất cả các repository của bạn trên hệ thống.  
    + config file ở trong Git directory (./git/config) : cấu hình này chỉ dành riêng cho 1 repository mà bạnd dang sử dụng, đọc và ghi vào tệp này bằng cách thêm tuỳ chọn --local.   
    - Ta có thể xem tất cả các config bằng cách dùng lệnh : git config --list --show-origin
11. Your Identity  
    - Điều đầu tiên cần làm là set name và email của user của bạn. Vì thông tin này là thông tin của user login, vì vậy ta phải cấu hình ở tuỳ chọn global:   
    + git config --global user.name “John Doe”  
    + git config --global user.email [johndoe@example.com](mailto:johndoe@example.com)
12. Your Editor  
    - git config --global core.editor emacs
13. Your default branch name  
    - Mặc định git tạo 1 branch được gọi là master khi mà tạo mới một repository với lệnh git init, tuy nhiên ta có thể đổi cái initial branch là branch khác.  
    - git config --global init.defaultBranch main
14. Checking your Settings  
    - git config --list  
    Để xem giá trị của config là gì, ta sử dụng lệnh :   
    - git config (--global / --system / --local) user.name . Nếu muốn config đó được cấu hình ở file nào, ta sử dụng thêm tuỳ chọn --show-origin
15. Git Basic
    1. Tạo một repository  
       - git init  
       - git add .  
       - git commit -m “first commit”  
       Hoặc có thể clone một repository : git clone ${URL}
    2. Lắng nghe sự thay đổi của một Repository  
       - Tất cả các file ở trong working directory đều có 2 trạng thái là được kiểm soát và không được kiểm soát (tracked vs untracked)  
       - Tracked file là những file mà nó ở last snapshot, nó có thể chưa modified, hoặc modified hoặc là staged (unmodified, modified, staged). Có nghĩa là Git đã biết thông tin về nó  
       - UnTracked file thì ngược lại, Untracked file là những file mới thêm vào Working directory nhưng chưa dùng lệnh git add, nó cũng ở trạng thái Untracked.Khi mà clone git repository lần đầu, tất cả các file đều có trạng thái là tracked và unmodified bởi vì Git vừa mới lấy về và ta chưa sửa nó.  
         
         
       (File A là file vừa mới thêm vào Working Directory, File B là file vừa bị xoá khỏi Working Directory, cả 2 đều ở trạng thái Un  
       - Những file ở trạng thái Unmodified, khi ta sửa nó, nó sẽ ở trạng thái Modified  
       
    3. Kiểm tra trạng thái của một file ở trong repository  
       - [git status]  
         
       khi gặp commit này, có nghĩa là tất cả các file đều ở trạng thái unmodified.  
         
       UnTracked đơn giản được hiểu là Git nhìn thấy file README và không tìm thấy nó ở previous snapshot (commit cũ)
    4. Tracking new file  
       - [git add] , đánh dấu trạng thái của file là staged
    5. Short status  
       - [git status] khá là đầy đủ nhưng mà nó dài dòng, dùng lệnh [git status -s] hoặc [git status --short] để xem nhanh trạng thái của các file.  
         
       + Tiền tố A : file đã được thêm vào staging area  
       + Tiền tố M : file đã được modified  
       + ?? là file UnTracked, nó vừa mới thêm vào, chưa add  
       + File A được modified và staged bằng lệnh git add, sau đó nó lại được modified tiếp, thì nó có prefix status là MM  
       
    6. Ignores File  
       - Thông thường ta có một lớp các file mà ta không muốn Git tự động add hoặc là thể hiện là UnTracked. Trong trường hợp này, ta phải liệt kê ra danh sách các tập tin để git ignore, với tên các file cần ignore được chứa trong file có tên là .gitignore   
         
       - \*.[oa] => bảo với Git rằng, bỏ qua Tất cả các file có đuôi là o hoặc là a.  
       - \*~ => bỏ qua tất cả các file có đuôi là ~, thường những file có đuôi là ~ là những file temp của các text editor  
       - Các luật để viết file .gitignore là:  
         
       # là comment, những dòng bắt đầu bằng # hoặc là blank line thì được bỏ qua  
       \*.a => tât cả những file có đuôi là a  
       !lib.a => nhưng mà file lib.a thì không bỏ qua  
       /TODO => chỉ ignore ở trong thư mục TODO không phải subdir của TODO  
       build/ => ignore tất cả các file trong thư mục có tên build  
       doc/\*.txt => ignore file doc/notes.txt nhưng không phải doc/server/arch.txt  
       doc/\*\*/\*.pdf => ignore tất cả pdf file ở trong doc và subdir của doc  
       - Thường thì 1 repository chỉ có 1 file .gitignore, tuy nhiên ta có thể tạo nhiều file .gitignore ở trong các thư mục con  
       - Đôi khi lệnh git ignore không hoạt động, là do git lưu cache, để giải phóng cache, ta dùng lệnh : [git rm -r --cached .]
    7. View Your Staged and UnStaged Changes  
       - Nếu như git status chưa đầy đủ thông tin, bạn muốn biết chính xác bạn đã thay đổi cái gì, không chỉ những file bạn đã thay đổi  
       - [git diff]  
         
       lệnh git diff trả lời cho 2 câu hỏi:  
       + Thông tin những gì bạn thay đổi file nhưng chưa staged, git diff show cho ta chi tiết line added và line removed  
       - Đây là những file ở trạng thái Modified, vậy còn những file staged, bạn đã lưu những gì trong đó, lệnh diff bình thường ko show ra.  
       - [git diff --staged] => show ra những thay đổi của những file ở trạng thái staged   
       - [git diff –cached] => giống –staged
    8. Commit Your changes  
       - Lưu ý rằng mọi thứ mới chỉ ở trạng thái unstaged.  
       - Những file mà ở created hay modified mà chưa chạy git add thì không được liệt kê vào danh sách commit. Các file đó ở trạng thái là Modified trên disk  
       - Cách đơn giản nhất để commit là : git commit  
         
       Đây là default message commit  
       - Ngoài ra có thể nhập nội dung message commit bằng flag –m, git commit –m “your content message”  
         
         
       - Nó chỉ ra branch ta committed là master, mã SHA-1 checksum commit là 363dc4f, có bao nhiêu file cahnge, có bao nhiêu dòng thêm và xoá đi trên file đó  
       - Nên nhớ rằng commit ghi lại cái snapshot mà bạn lưu ở staging area, những thứ mà chưa stage thì nó đang ở trạng thái modified, phải git add thì mới commit được. Ta có thể revert lại bẩn snapshot vừa commit.
    9. Skipping the Staging Area  
       - Mặc dù Staging area rất hữu ích trong việc xem xét những file modified hoặc untracked. Tuy nhiên có một số trường hợp không cần phải dùng lệnh git add . mà muốn commit luôn. Nếu muốn skip staging area, git cung cấp một câu lệnh rất đơn giản.  
       - Thêm -a option vào sau câu lệnh git commit, tất cả các file sẽ được staged
    10. Removing files  
        - Lệnh git rm để xoá file, folder ở trong Working directory  
        -   
        - Đó là cách đơn giản để xoá file, file ở trạng thái unstaged, chúng ta phải add và commit.  
        - Tệp bị biến mất và không bị theo dõi  
        - Nếu tệp đang ở trạng thái Modified hoặc đang ở staging area thì ta không thể xoá được, tuy nhiên nếu vẫn muốn xoá thì ta thêm option -f vào đằng sau  
          
        => Thêm option -f  
          
        - Nếu bạn muốn xoá file đó khỏi sự theo dõi của Git nhưng vẫn muốn nó tồn tại trong thư mục Working directory. Ngoài việc thêm vào file .gitignore, ta có thể dùng lệnh : git rm --cache {file}  
        [git rm --cached logs/\\*.log]  
        - (Có những trường hợp thêm vào .gitignore nhưng git vẫn list ra những file log, đó là do cache của git chưa được clean, ta phải dùng lệnh git rm --cached này)
    11. Moving Files  
        - Không giống như các VCSs khác, Git không theo dõi tường minh sự di chuyển của tệp  
        - Nếu như ta đổi tên file, không có metadata nào của Git chỉ ra là ta đổi tên tệp. Tuy nhiên Git khá là thông minh để tìm ra điều đó (Cái này ta nhắc tới ở đoạn sau)  
        - [git mv file\_name\_from file\_name\_to]  
        -   
        - Trên thực tế, lệnh git mv giống với một chuỗi các lệnh này  
        + [mv file\_name\_from file\_name\_to]  
        + [git rm file\_name\_from]  
        + [git add file\_name\_to]  
          
        Chú ý rằng, lệnh git mv hoàn toàn khác lệnh mv  
        
    12. Viewing commit History  
        - Sau khi tạo một số commit, hoặc là clone từ một repository. Bạn có thể muốn nhìn lại để xem điều gì đã xảy ra, tra lại log commit  
        - [git log]: Git liệt kê các commit được thực hiện trong repository order theo thời gian  
          
        Lệnh này liệt kê ra Author name, email, Date commit , message và mã SHA-1 checksum  
        - Có nhiều options cho lệnh git log giúp hiển thị những gì mà ta mong muốn, sau đây là một số option của lệnh git log  
        - option [-p hoặc –patch]: cung cấp sự khác biệt (patch output) ở mỗi commit ( giống lệnh git diff)  
          
          
        - Ta có thể giới hạn list commit bằng lệnh -2 ( 2 là số lượng commit muốn hiển thị), ví dụ: git log -2 hoặc git log –p -2  
        - Nếu muốn xem một số thống kê ngắn cho từng commit, sử dụng option –stat  
          
          
        - Options [--stat] sẽ show ra là có bao nhiêu file thay đổi và file đó thêm hoặc xoá bỏ những dòng nào  
        - Một option hữu dụng khác là [--pretty], tuỳ chọn này thay đổi định dạng đầu ra của log thành định dạng khác.   
          
        Option pretty=oneline là hiển thị từng commit history trên một dòng, ngoài ra còn có tuỳ chọn [short, full, fuller]  
        - Tuỳ chọn thú vị nhất là [--format]: cái mà ta xác định format log output của riêng chúng ta. git log --pretty=format:”%h - %an - %ar - %s” , bảng ký hiệu các định dạng dưới đây:

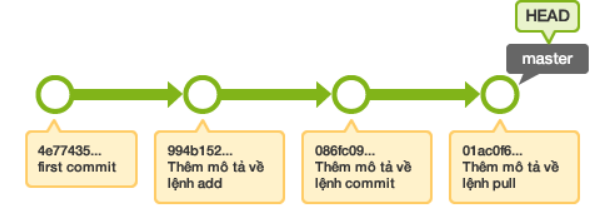
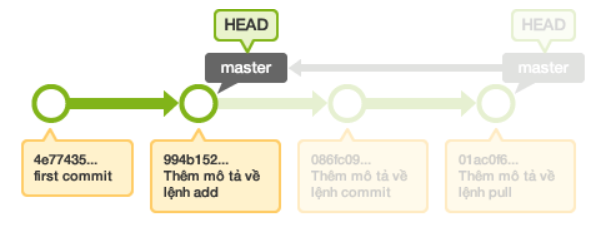
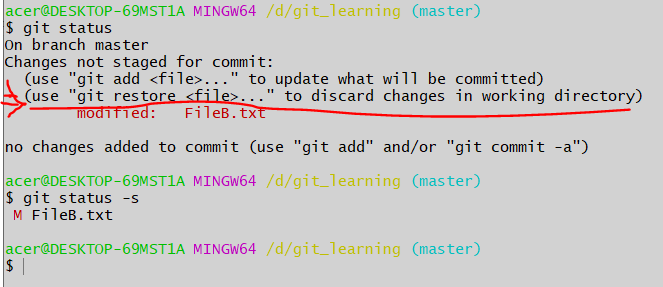
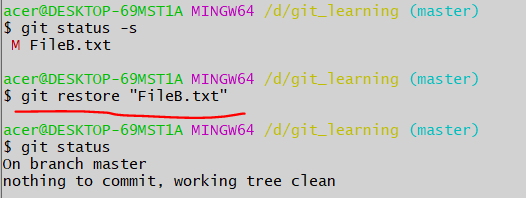
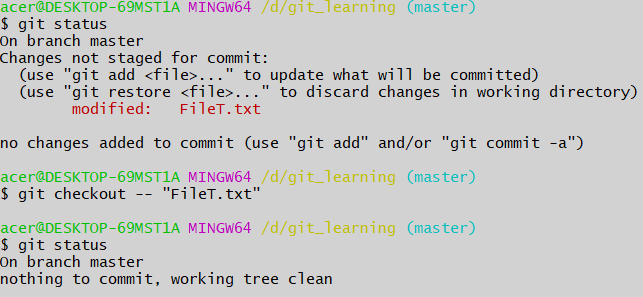
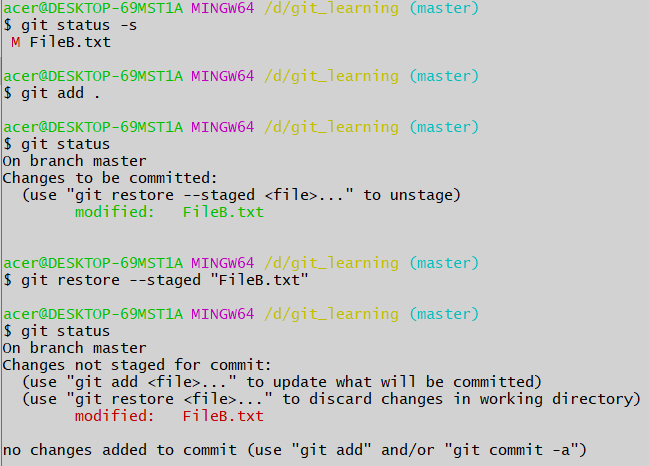
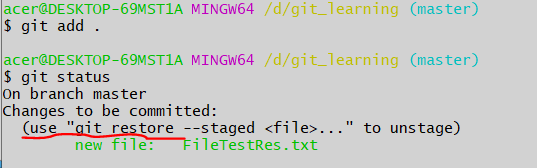
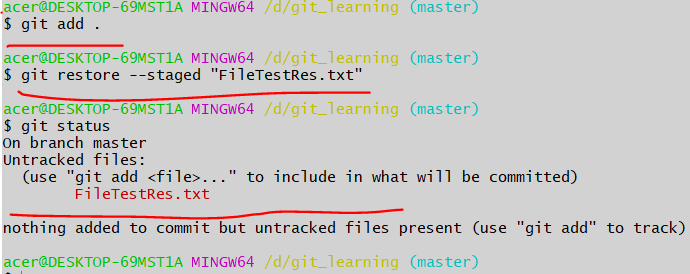
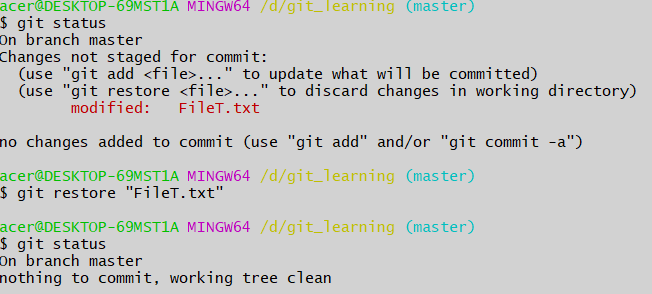
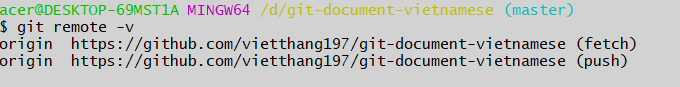
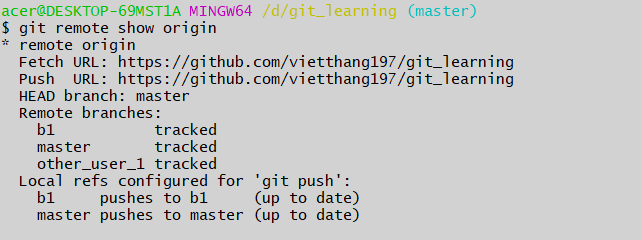
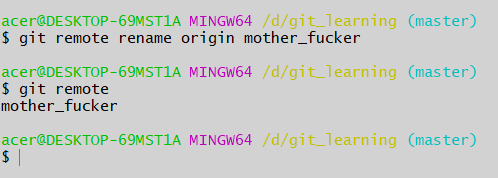
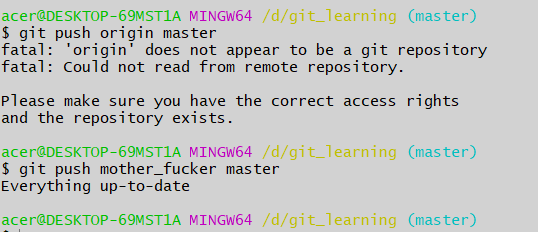
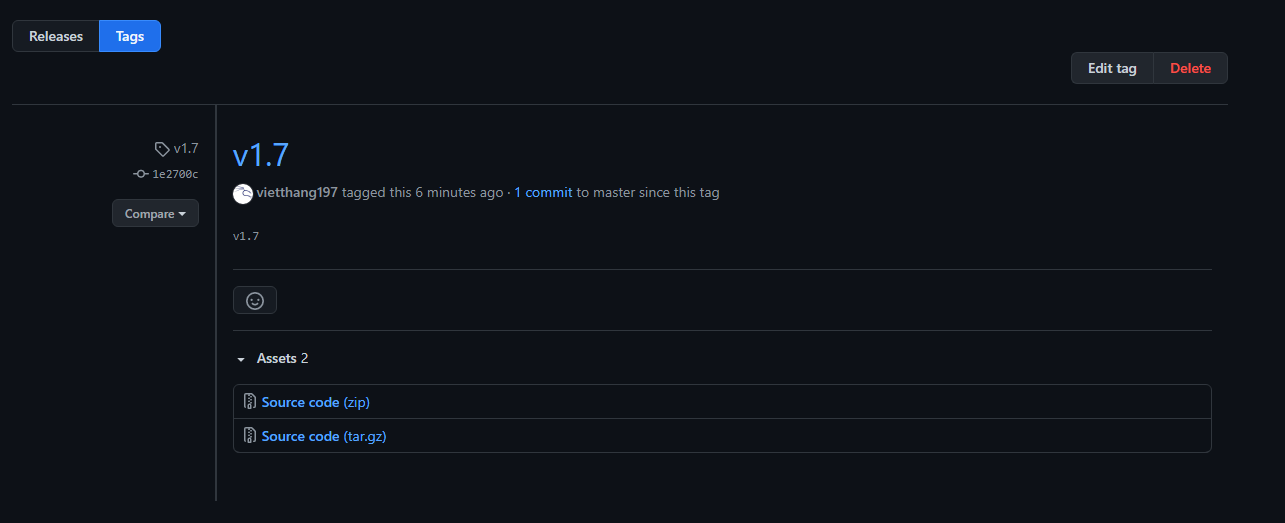
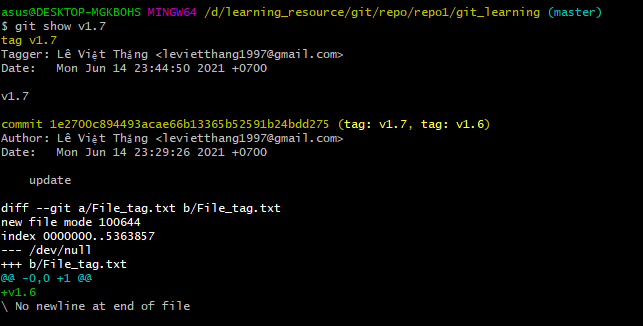
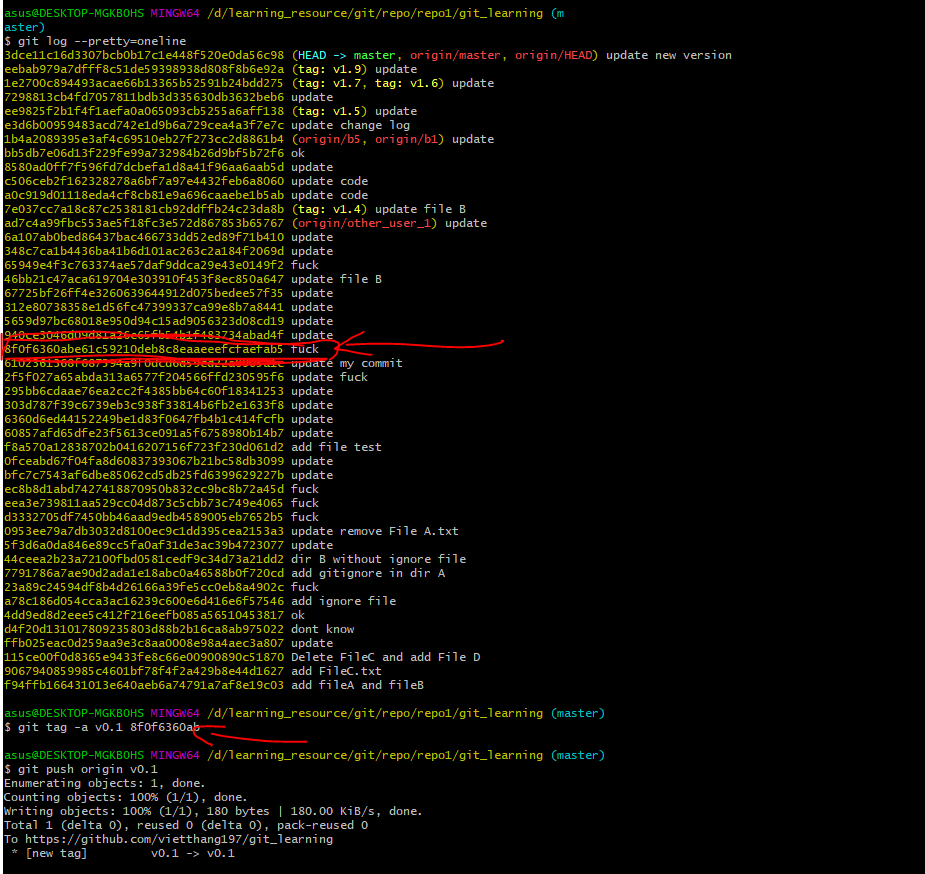
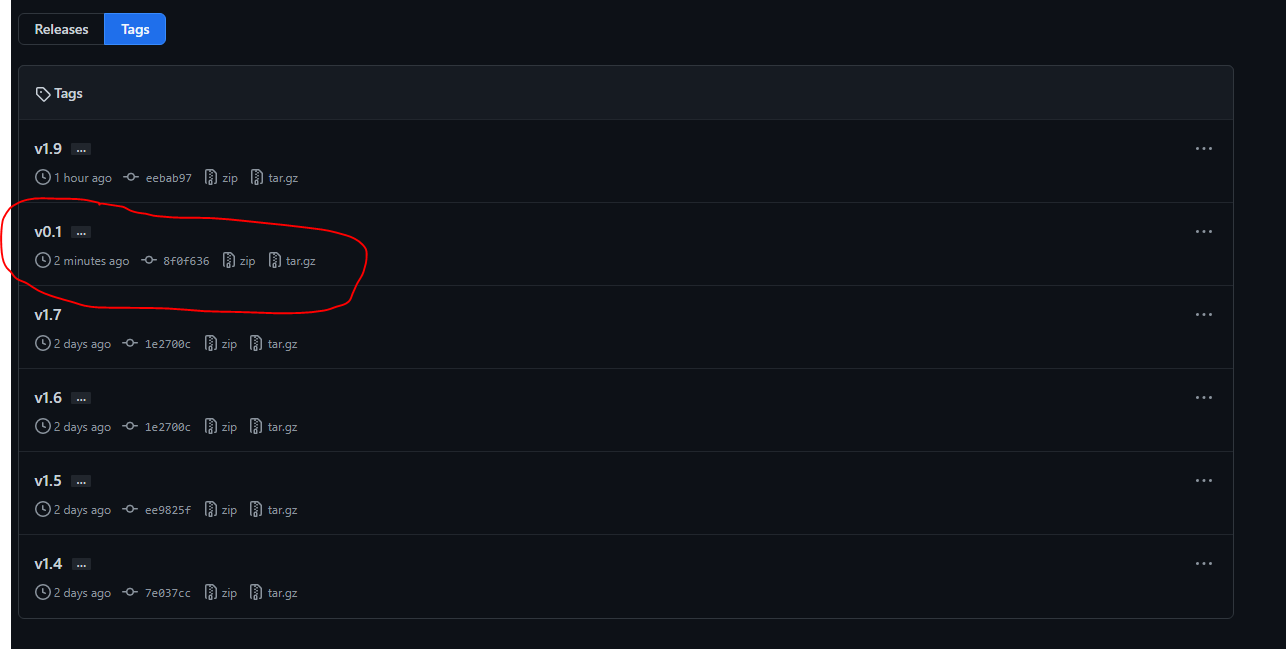
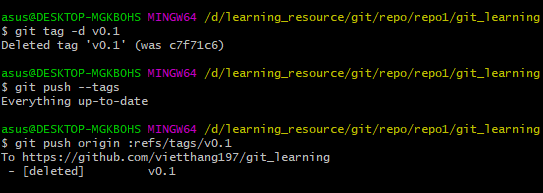
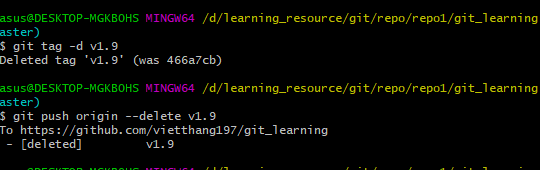
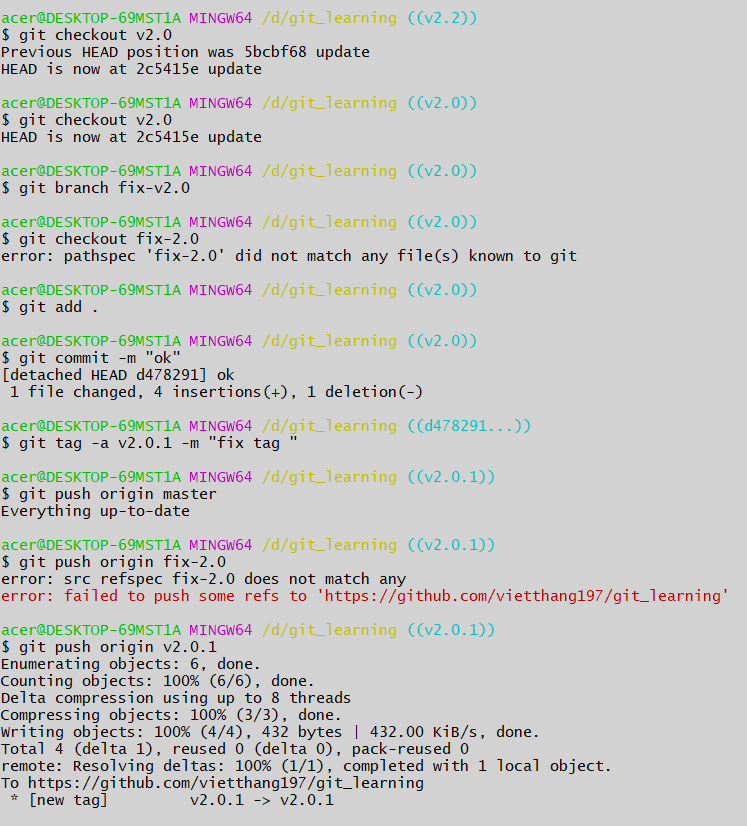
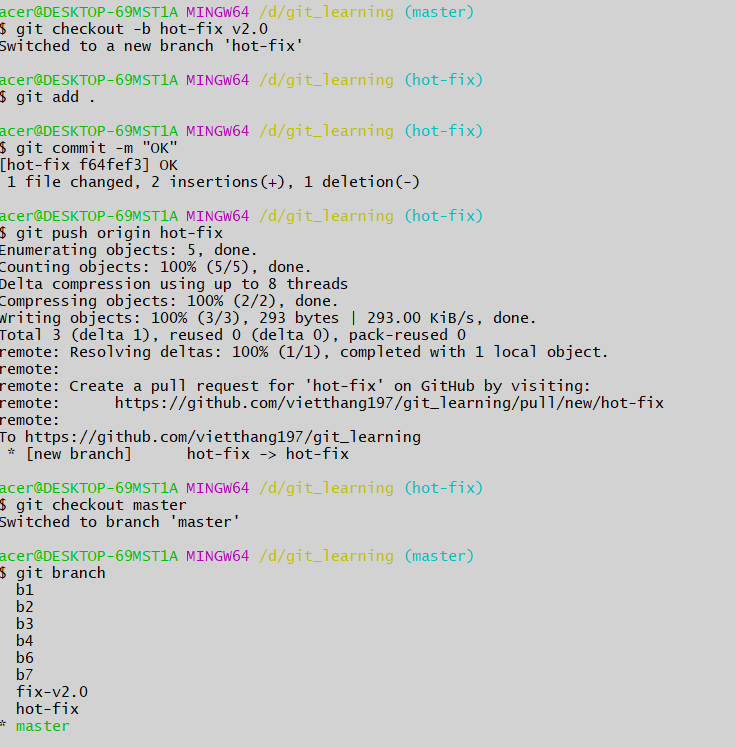
|  |  |
| --- | --- |
| Ký hiệu | Mô tả định dạng đầu ra |
| %H | Toàn bộ mã Hash commit |
| %h | Hiển thị một phần mã Hash commit |
| %T | Hiển thị mã Tree hash |
| %t | Hiển thị một phần mã Tree hash |
| %P | Hiển thị mã Parent hash |
| %p | Hiển thị một phần mã Parent hash |
| %an | Author name |
| %ae | Author email |
| %ad | Author date |
| %cn | Committer name |
| %ce | Committer email |
| %cd | Committer date |
| %cr | Committer date, relative |
| %s | Subject ( commit message) |

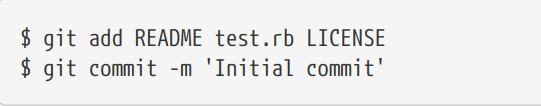
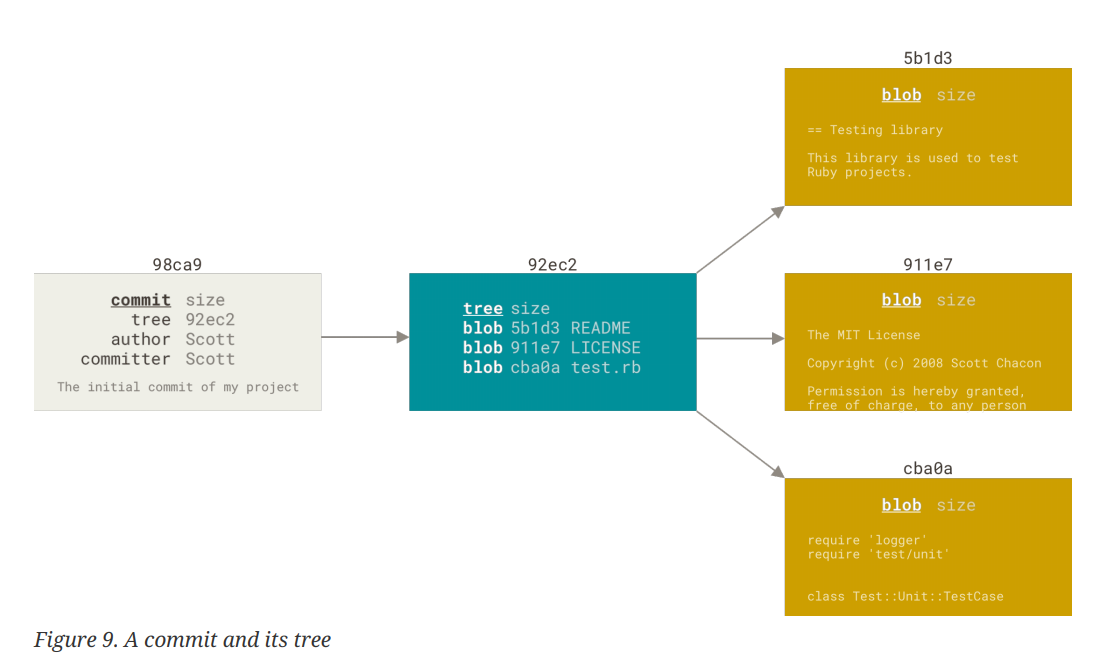
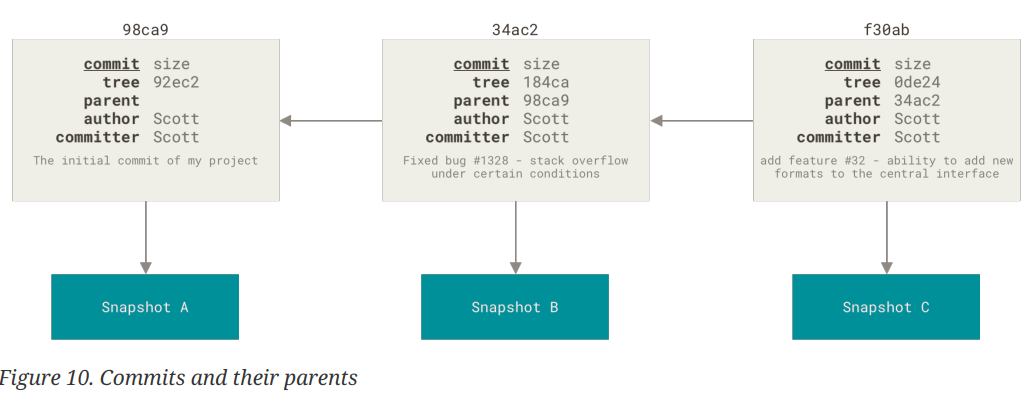
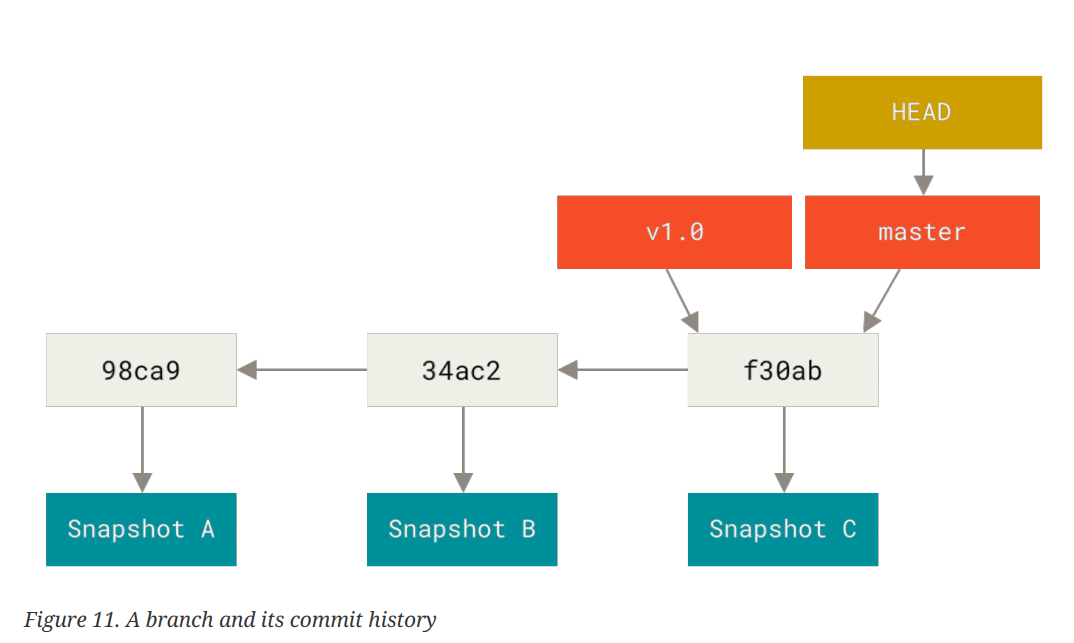
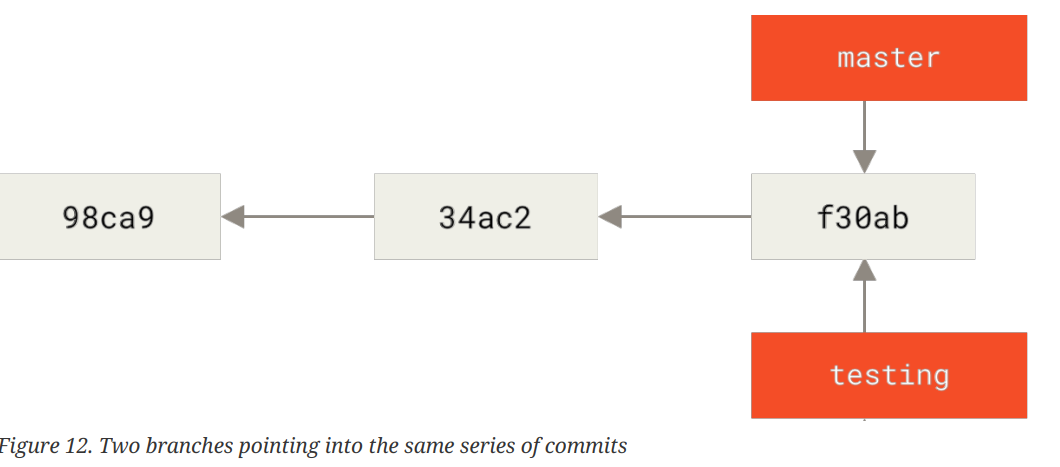
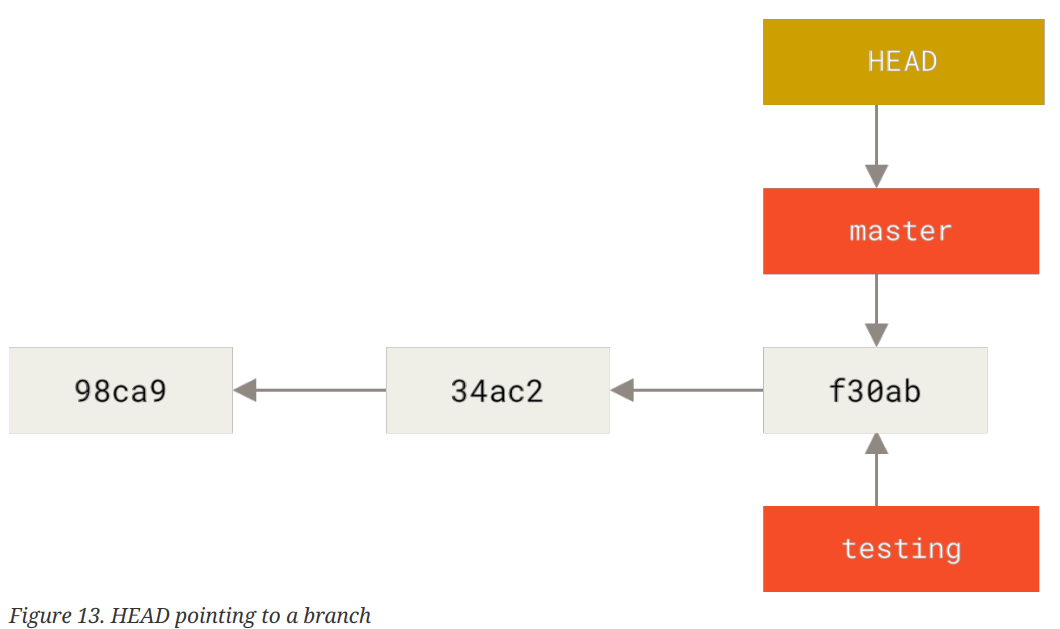
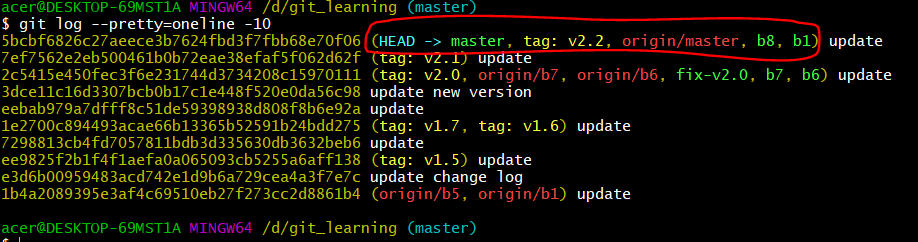
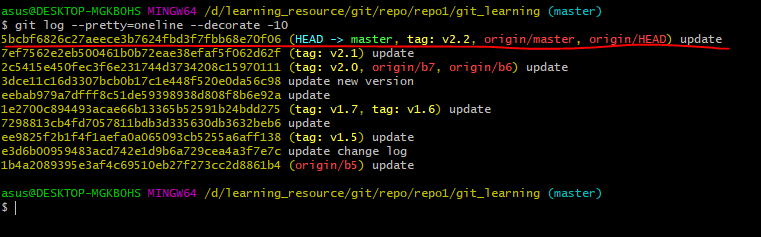
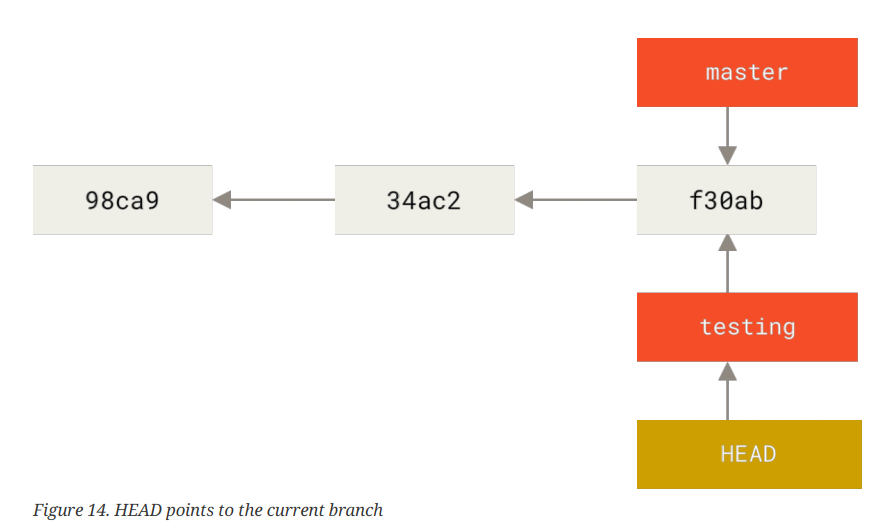
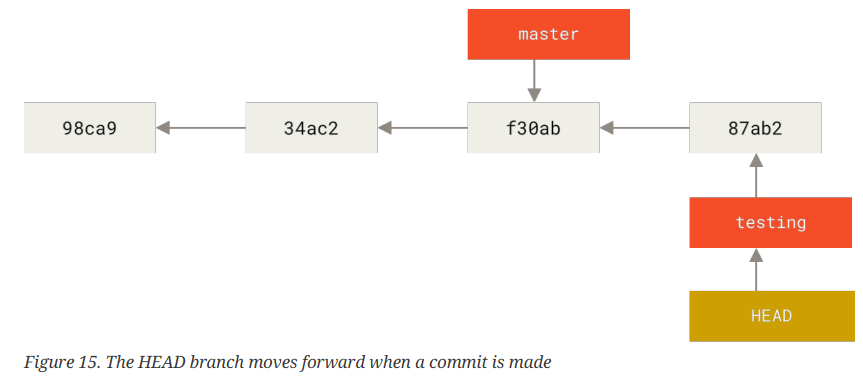
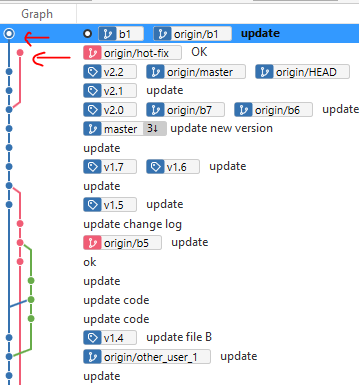
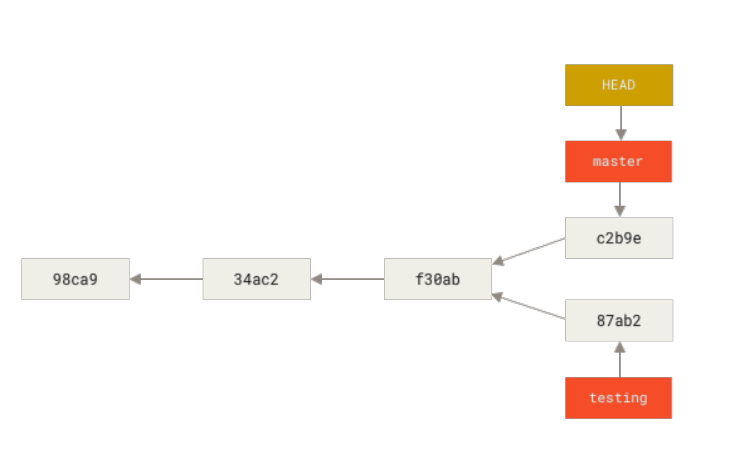
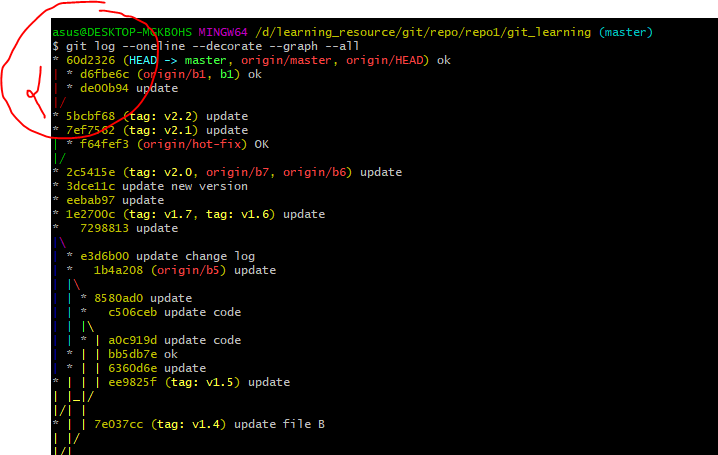
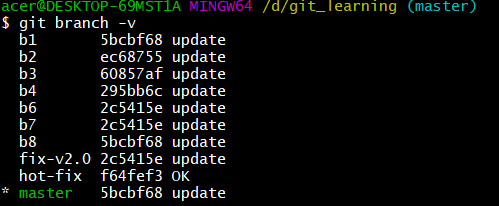
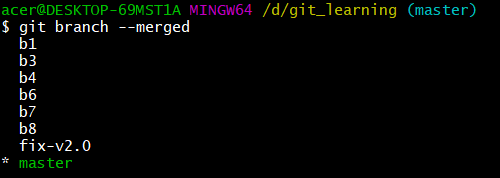
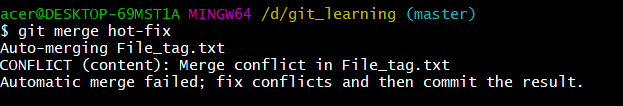
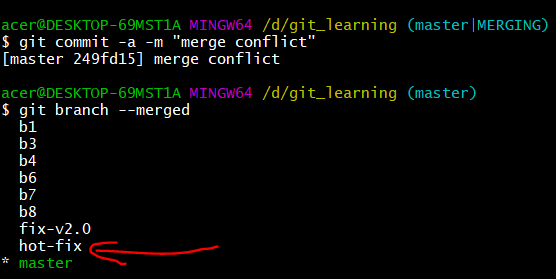
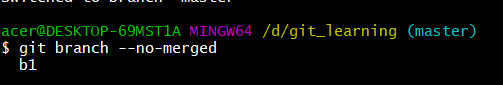
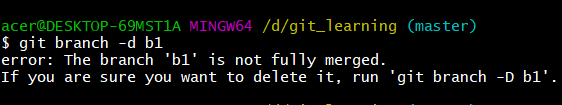
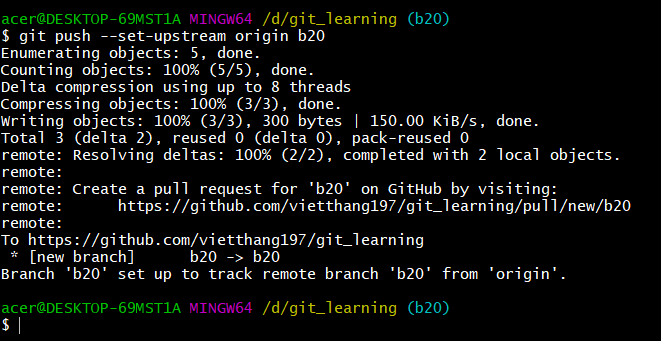
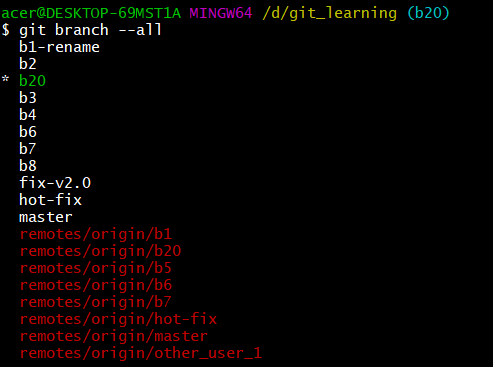
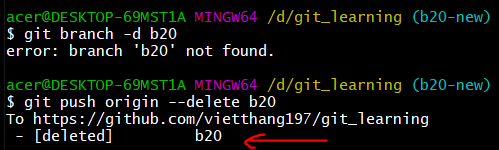
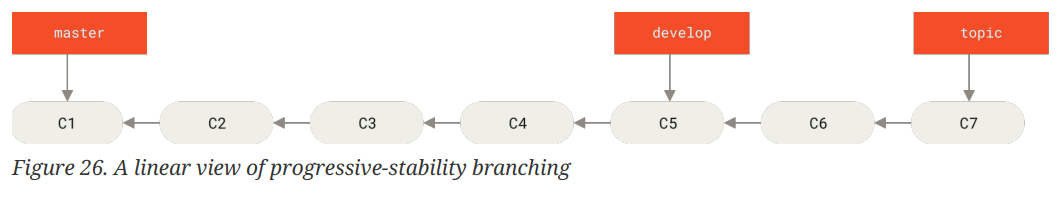
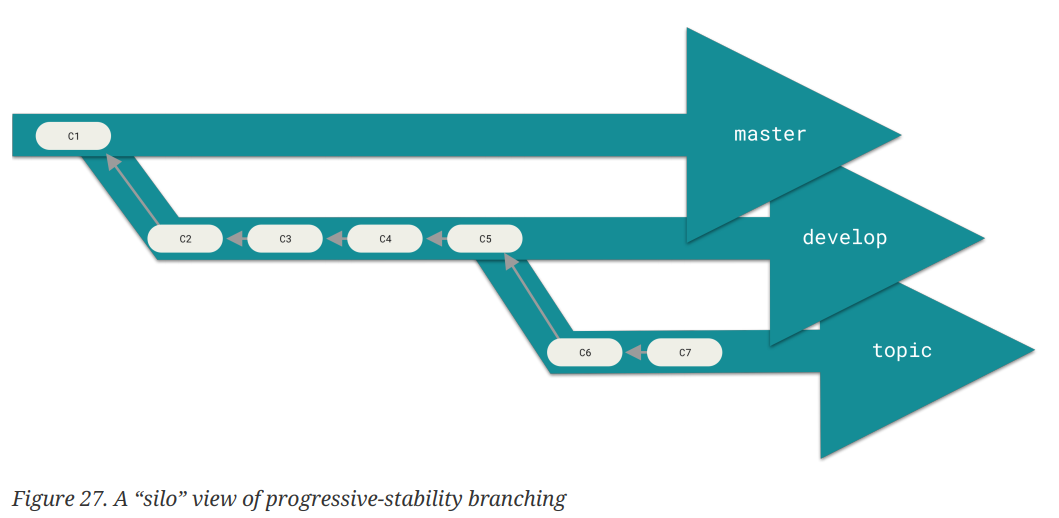
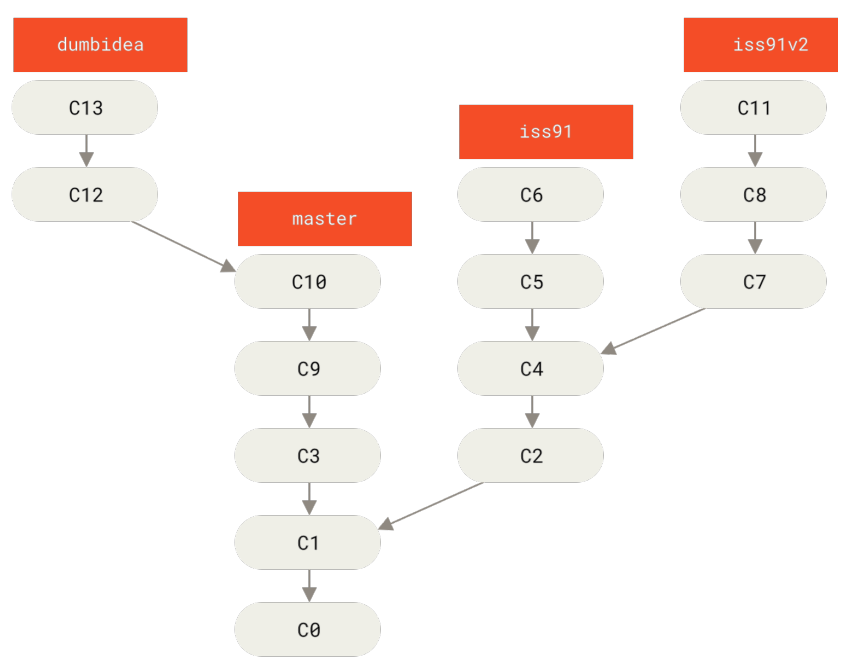
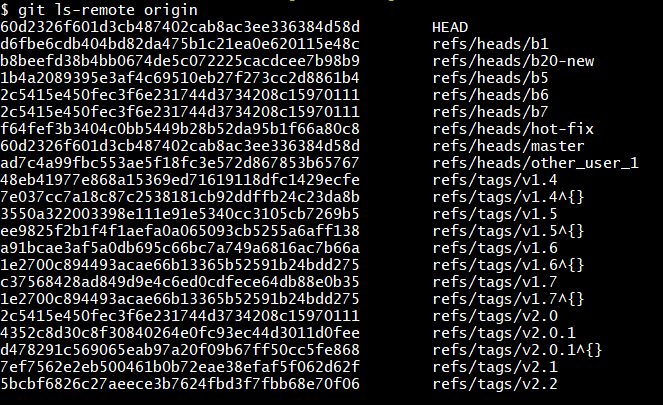
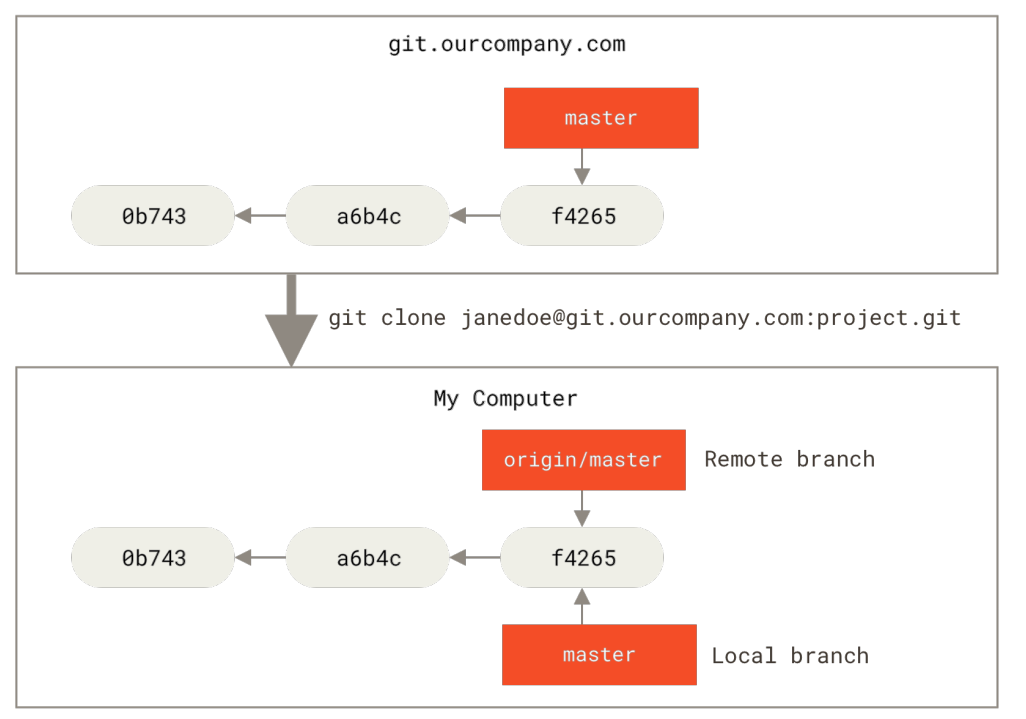
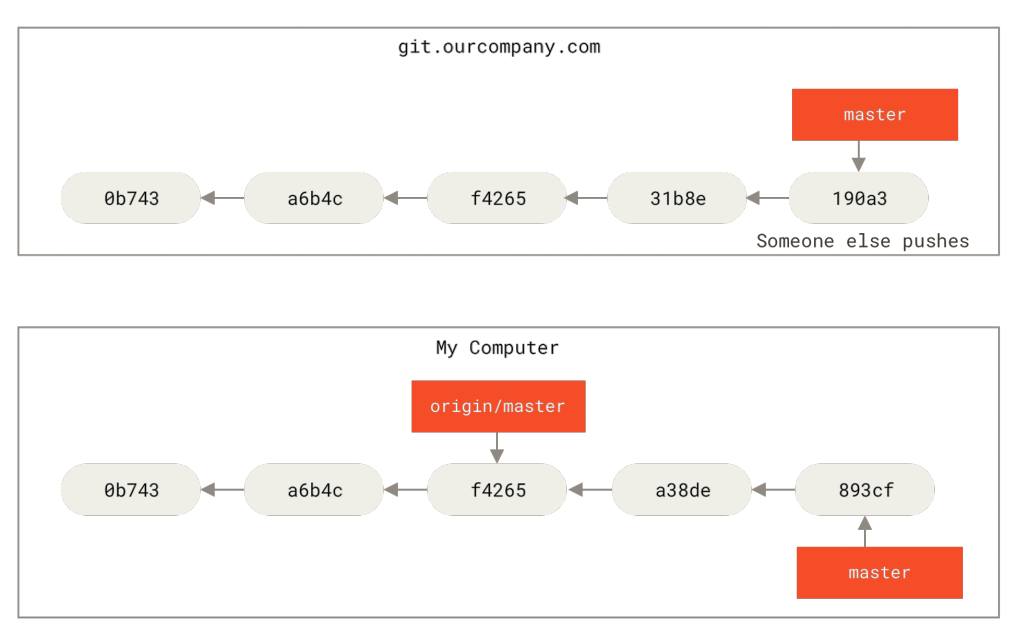
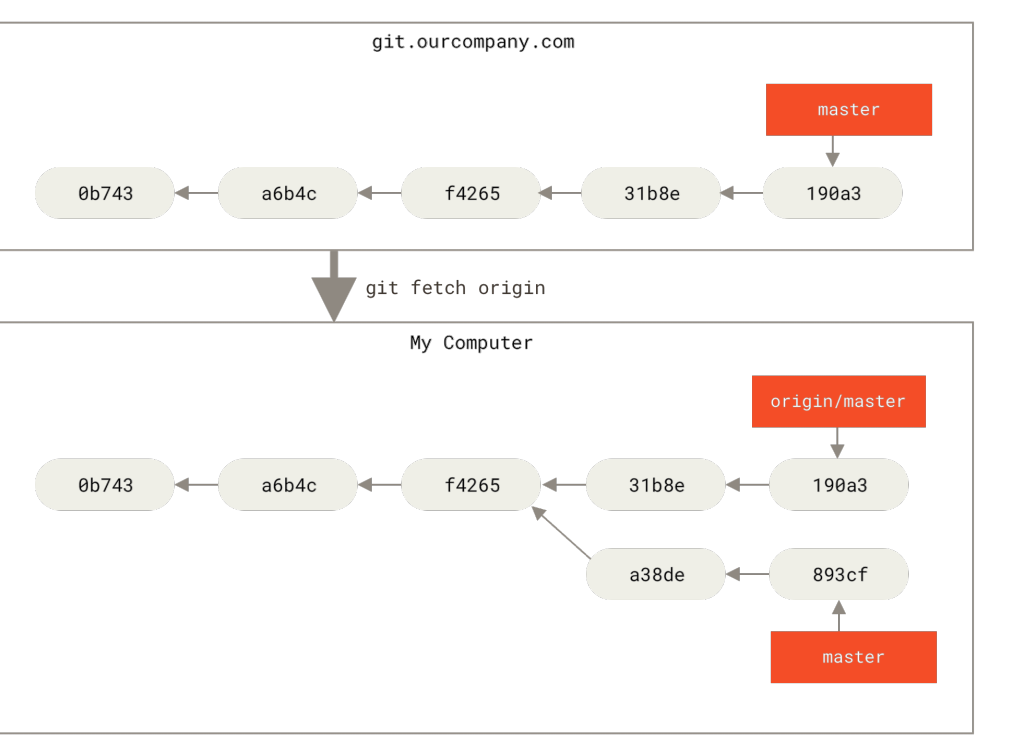
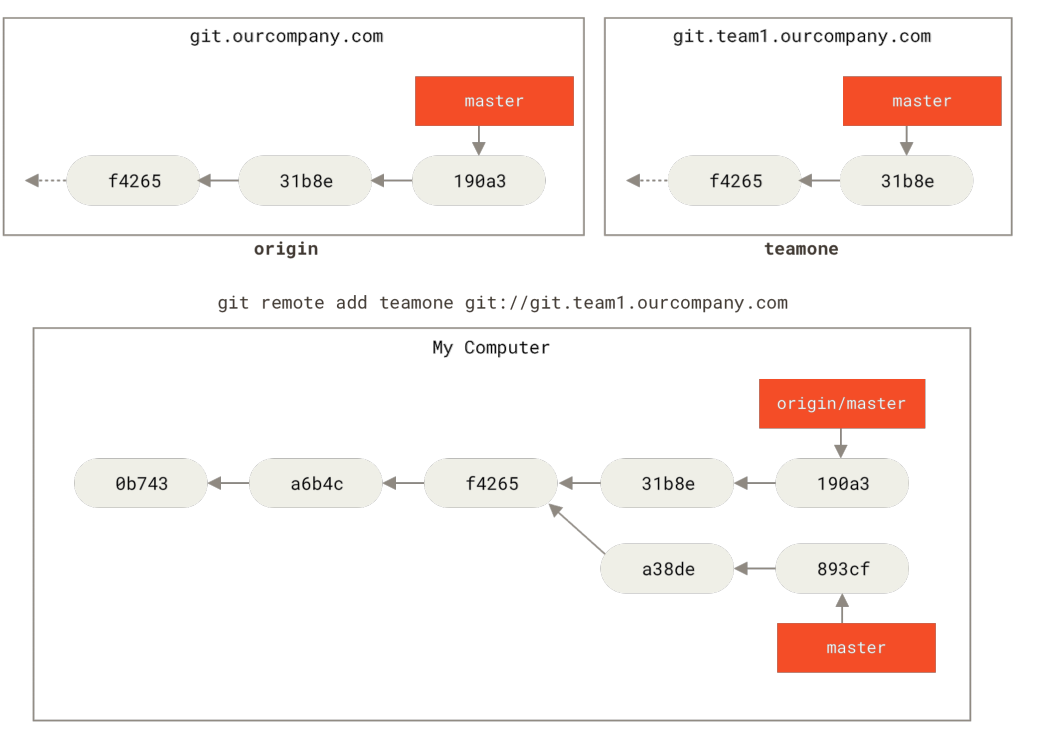
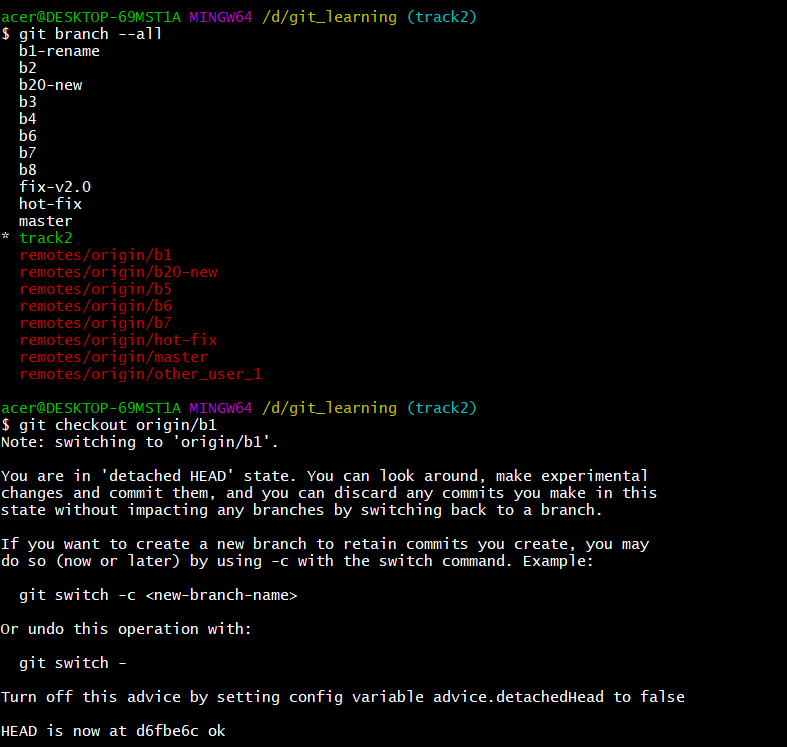
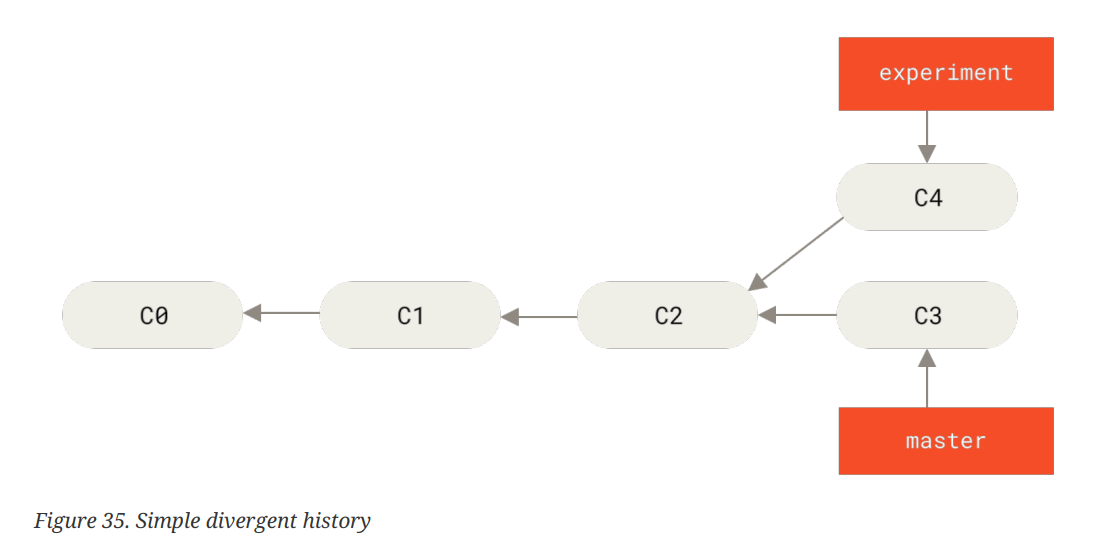
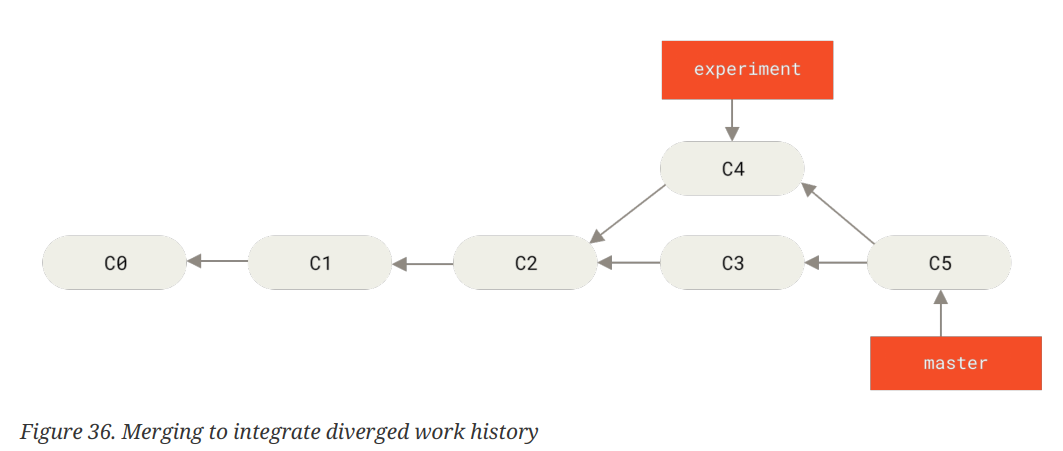
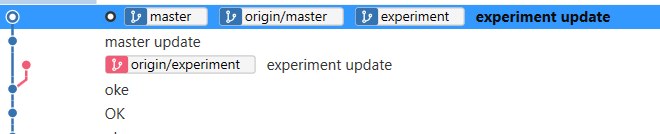
* 1. Sự khác biệt giữa Author và Committer. Giả sử Author là người code ra bản vá cho project, các committer sẽ là người áp dụng bản vá đó.  
     - pretty=oneline và pretty=format rất là hữu ích khi format display log, cùng với đó là một option khác là --graph  
     - Git log options  
     -

|  |  |
| --- | --- |
| Options | Des |
| -p | Show ra patch với mỗi commit ( commit này đổi những file nào, file đổi những dòng nào, nội dung của các dòng là gì) |
| --stat | Ngắn gọn hơn -p, --stat không hiển thị nội dung xoá và thay đổi,nó chỉ hiển thị xoá bao nhiêu dòng, thêm bao nhiêu dòng |
| --shortstat | Ngắn gọn hơn stat, chỉ thống kê có bao nhiêu file changed, bao nhiêu insert, delete |
| --name-only | Chỉ show mỗi tên các file thay đổi |
| --name-status | Chỉ hiển thị tên file và trạng thái thay đổi của file (M hay là D) |
| --abbrev-commit | Hiển thị ngắn gọn mã hash của commit |
| --relative-date | Ngày release bản commit |
| --graph | Hiển thị graph các branch và Merge History |
| --pretty | Format lại display output |
| --oneline | Hiển thị ngắn mã hash và nội dung commit |

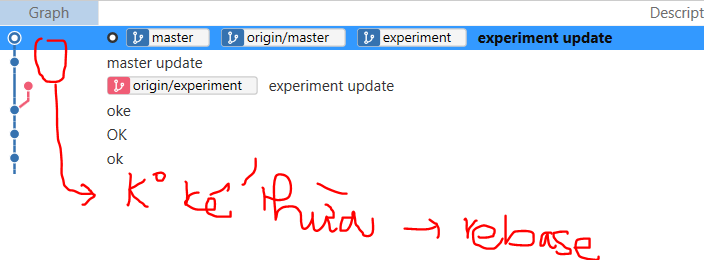
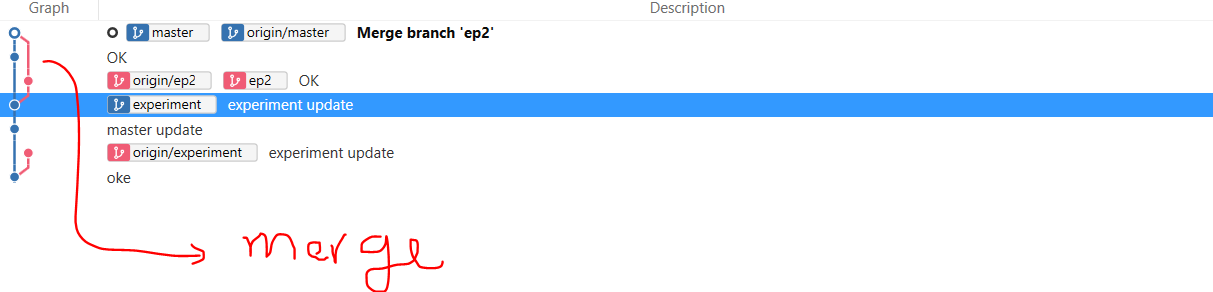
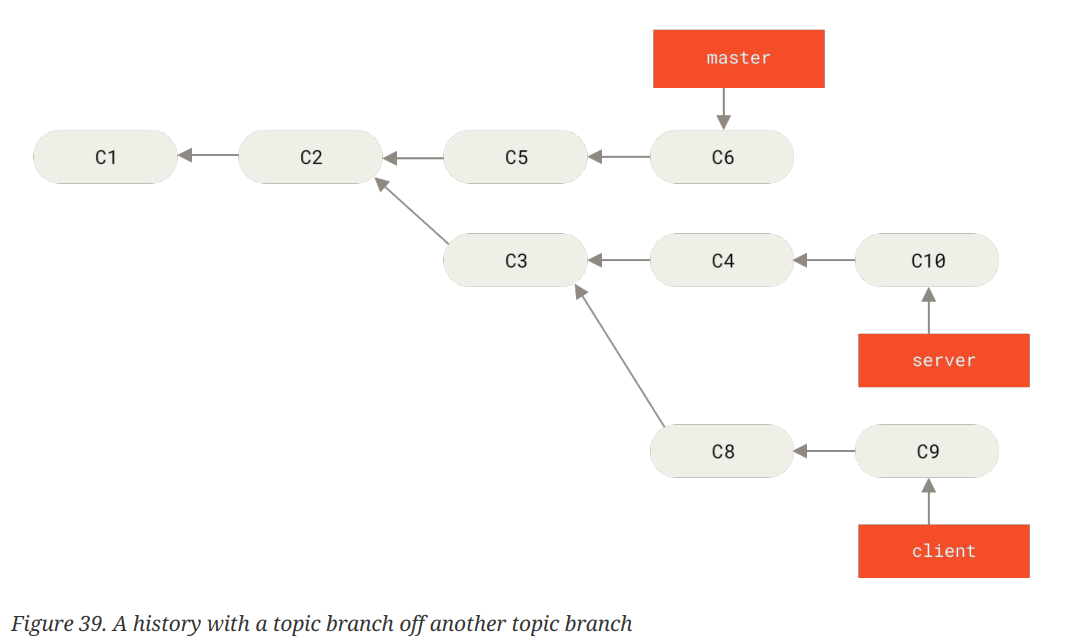
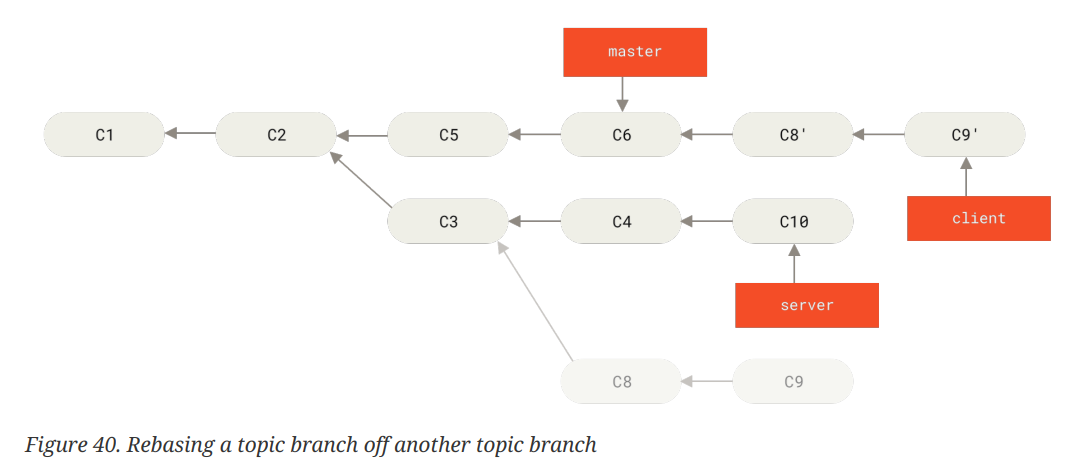
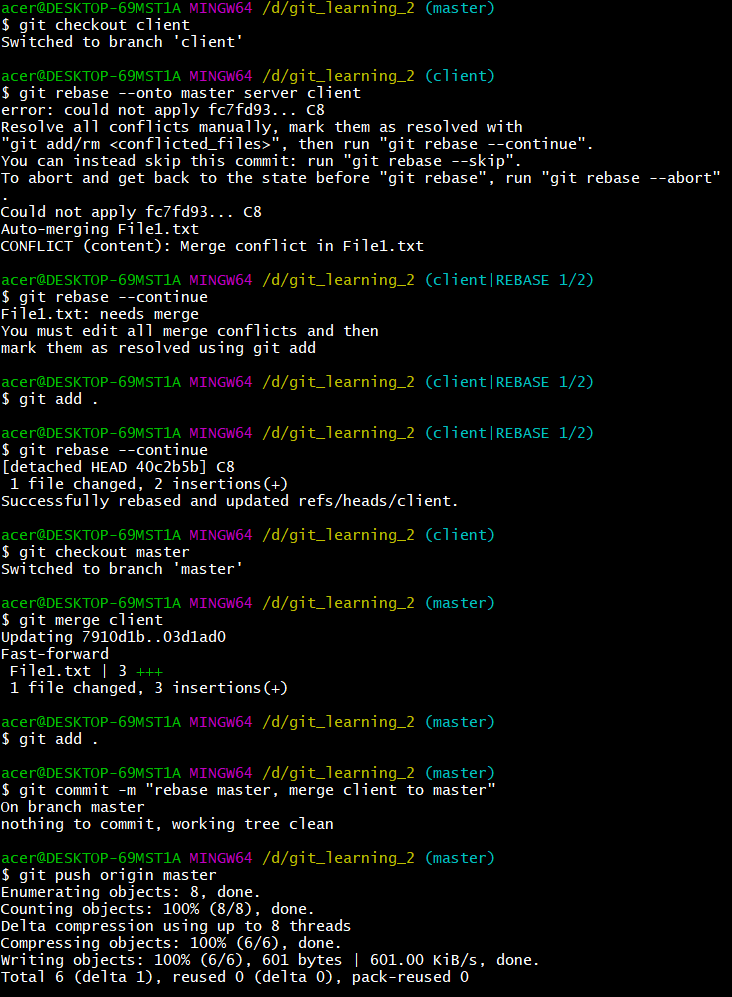
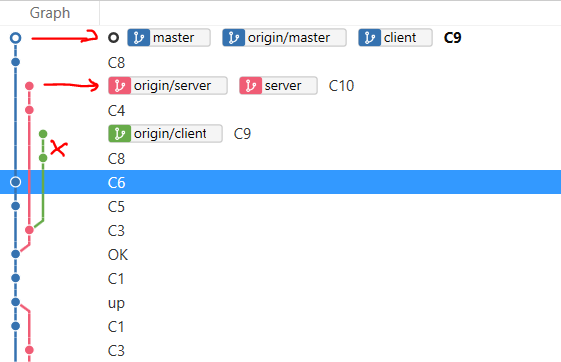
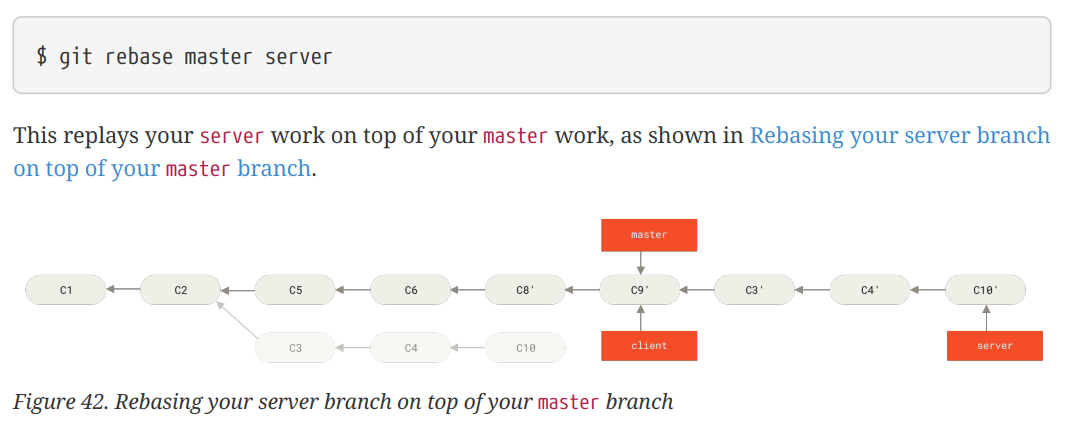
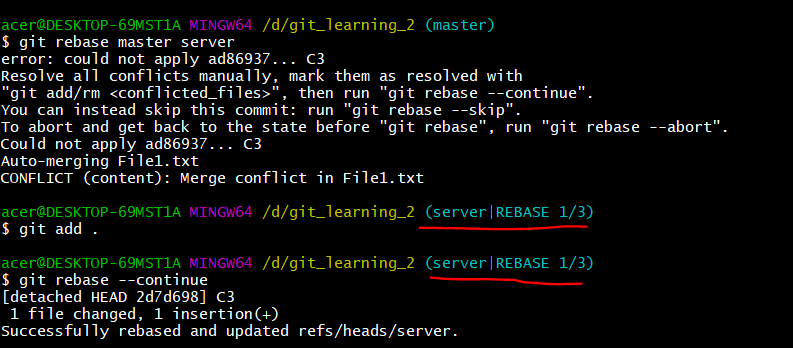
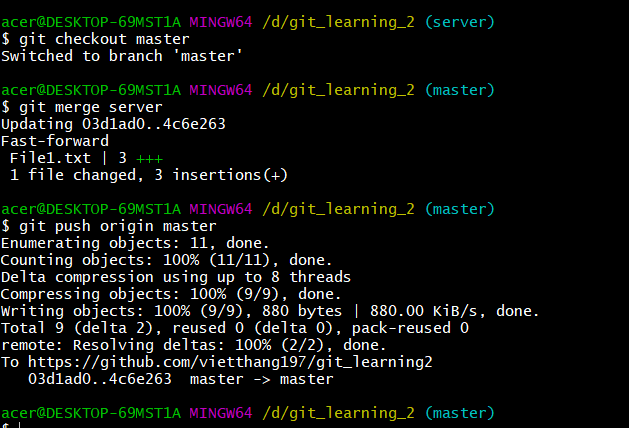
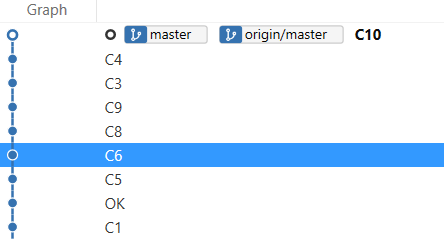
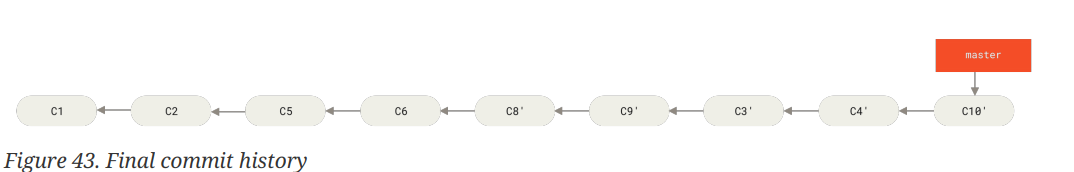
* 1. Limmiting Log Output  
     - Ngoài các option format output rất là hữu ích của git log, git còn có tuỳ chọn giới hạn dữ liệu select ra. Với option -n (n là số commit latest muốn hiển thị ra) ta có thể giới hạn số lượng commit hiển thị ra. Tuy nhiên nếu ta muốn hiển thị tất cả commit của ngày hôm qua, hôm kia, hôm x thì làm ntn?  
     - Time limiting option : --since | --until rất là hữu dụng trong trường hợp này. ví dụ  
     [git log --since=2.weeks] => lấy log commit từ 2 tuần trước đổ về đây, ta có thể specify day, days, week, weeks, year, years. Ngoài ra, ta có thể specify date format như sau   
     [git log --since=”2021-05-28”] hoặc [git log --since=”1 day ago”]  
     - Ngoài ra ta có thể git có thể lọc một số điều kiện để tìm kiếm log hiệu quả hơn ví dụ như  
     - options [--author] cho phép ta chỉ định author mà ta muốn lọc và [--grep] cho phép ta search key word ở trong message  
     ví dụ:  
     git log --author=”thắng”  
       
     git log --grep=”fuck”  
       
       
     - Một options khác cũng rất là hữu dụng đó chính là   
     option [-S]   
     [git log -S function\_name]  
     Giả sử như ta muốn log ra những commit mà thay đổi đoạn text function\_name trong file X nào đó, ta có thể dùng option này  
     - Một option tiếp theo cũng rất khả dụng là :   
     - [git log -- /path/to/file]: log tất cả những commit mà chỉnh sửa cái file này ở trong working directory (nên nhớ rằng file này phải nằm trong thư mục working directory)  
     - Bảng tổng hợp

|  |  |
| --- | --- |
| Option | description |
| -n | Chỉ show có n last commit |
| --since, --after | Since -> show commit kể từ ngày nào, after -> commit đó sau ngày bao nhiêu |
| --until, --before | Until -> show commit đến ngày bao nhiêu, before -> show commit trước ngày bao nhiêu |
| --author | Chỉ show những commit của author |
| --committer | Chỉ show những commit mà có dấu răng của committer |
| --grep | Chỉ show commit mà message của commit có chứa string tìm kiếm |
| -S | Show những commit mà add hoặc remove code có chứa string |

* 1. Undoing Things  
     - Ở bất kỳ giai đoạn nào, ta cũng muốn hoàn tác cái mẹ gì đó. chúng ta sẽ xem qua 1 cái tool rất là cơ bản để undoing changes mà chúng ta đã viết. Hãy cẩn thận vì không phải lúc nào ta cũng hoàn tác được các thao tác hoàn này. Nếu làm sai thì rất có khả năng mất code.  
     - Một trong những hoàn tác phổ biến khi mà chúng ta nhỡ tay commit quá sớm, ta có thể thêm option --amend  
     [ git commit --amend]  
     Giả sử như sau : Ta vừa add rồi commit một file A.txt, nhưng không may ta quên add và commit file B.txt, ta không muốn add rồi tạo commit khác (commit khác thì nó lại nảy ra mã sha-1 mới, ta muốn nó cùng commit vào cùng đợt của file A.txt). Để commit chung với commit của file A.txt, ta làm như sau:   
     - git add FileA.txt  
     - git commit -m “update fileA” => log commit là : 121357656723457  
     - git add FileB.txt  
     - git commit --amend => log commit vẫn là : 121357656723457  
     - Câu lệnh này nó sẽ mở cái edtior lên để add và chỉnh sửa, tuy nhiên mặc định ta lại ko có editor, nên do đó ta phải cấu hình editor bằng lệnh : git config --global core.editor {tên editor}  
     - amend chỉ hoạt động ở local khi mà ta chưa push commit lên repo nào đó. Nếu mà giả sử amend mà sử dụng được thì sẽ gây ra lỗi nếu trước khi ta amend đã có ông nào commit và push chèn trước rồi.
     1. Unstaging the staged file  
        - Tiếp theo ta sẽ làm việc với staging area và working directory.   
        - Nếu như ta dùng lệnh git add … để add file vào staging area, vậy làm sao để unstaged những file đó  
        - Lệnh : git reset HEAD <file> …  
        - Lệnh git reset là một lệnh nguy hiểm nếu như ta thêm flag --hard. [git reset --hard HEAD] Nếu như reset HEAD chỉ là đổi trạng thái của một file từ staged sang modified thì flag --hard cho bay luôn cái file đang change, nó thay bằng cái file của commit cũ   
          
        
     2. Unmodifying the Modified file  
        - Điều gì sẽ xảy ra nếu như ta nhận ra rằng không muốn giữ các thay đổi của mình ở file CONTRIBUTING.md  
        - Vậy thì có cách đơn giản nào để hoàn tác lại thay đổi này. Vâng, rất là may mắn, git status đã chỉ cho ta cách làm sao để làm điều này.  
        - Trường hợp đầu tiên là : Bạn đang chỉnh sửa một file ở trong working directory, file này đang có trạng thái là modified, tuy nhiên bạn chưa staging nó (chưa dùng lệnh git add), ta có thể dùng lệnh :  
        + git restore <file\_name>  
        + hoặc là lệnh git checkout -- <file\_name>  
          
          
          
          
          
          
        - Trường hợp thứ 2 là ta muốn unchanges file đã được staged và đợi commit, ta không muốn thay đổi file đó nữa. Ngoài cách dùng lệnh : git reset HEAD “FileB.txt”, ta có thể dùng lệnh:   
        git restore --staged “FileB.txt” để chuyển trạng thái về unstaged  
          
          
          
        git checkout cũng là dangerous command.  
          
        - Để restore all file, ta dùng : [git restore --staged .] hoặc [git checkout -- . ]  
          
        - Hãy nhớ rằng mọi thứ đã được committed ở Git đều có thể được khôi phục. Commit trên một nhánh đã bị xoá hoặc là commit được ghi đè bằng flag --amend cũng đều có thể rollback
     3. Undoing things with git restore  
        - Git version 2.23.0 cung cấp một command mới là : git restore, về cơ bản nó là một giải pháp thay thế cho git reset  
        - Từ phiên bản Git 2.23 trở đi, ta sẽ dùng git restore thay thế git reset cho các hoạt động hoàn tác  
        - Unstaging staged file with git restore  
        +   
        Ở ngay dưới câu lệnh git status, ta có thể thấy rằng: để unstage file (chuyển từ trạng thái staged sang unstaged) ta dùng lệnh [git restore -- staged <file\_name>]  
          
        File sẽ chuyển về trạng thái unstaged  
        + Nếu một file đang ở trạng thái modified, mà bạn chưa dùng lệnh git add, bạn muốn restore nó về thời điểm commit, ta dùng lệnh git restore mà ko cần flag --staged  
          
        Lưu ý là trường hợp này là file mà git đã quản lý (đã được commit từ trước, chưa dành cho các file thêm mới)
  2. Working with remotes  
     - Để cộng tác với người khác ở bất kỳ dự án nào, ta phải biết quản lý cái remote repository của chúng ta.  
     - Remote repository là kho lưu trữ các phiên bản của dự án ở trên internet hoặc là ở network nào đó.  
     - Quản lý repo dạy cho ta làm các việc như : add remote, remove remotes, quản lý branch. Ở phần này, ta sẽ được dạy về Remote Management skills  
     1. Showing your remote  
        - [git remote]: show ra các remote của repo  
          
        - Nếu ta thêm flag: [git remote -v] thì nó sẽ show ra cả url của remote đó  
        
     2. Adding remote repository  
        - Khi mà clone một repository, thì git đã ngầm add một remote có tên là origin vào danh sách các remote của repository  
        - Bây giờ ta sẽ học cách add remote một cách tường minh  
        - command [git remote add <short\_name> <url>]  
        - Giả sử ban đầu repo của bạn đang ở local, muốn push lên GitHub, thì ra vào github tạo một repository rồi dùng command git remote add, rồi push lên master  
        - command [git fetch <remote\_short\_name>] để lấy thông tin từ remote repository
     3. Fetching and Pulling from Remote  
        - Để get data từ remote repository, ta dùng command [git fetch <remote\_name>]  
        - Lệnh git fetch sẽ đi đến remote repository và lấy về những data mà local repository chưa có. Nó sẽ lưu data này vào local repository và ở nhánh FETCH\_HEAD, nếu muốn merge code, ta sẽ merge current branch với branch FETCH\_HEAD này  
        - Điều quan trọng là git fetch chỉ download data vào local repository, nó không tự động merge data đó với bất kỳ công việc nào của bạn hoặc là bản chỉnh sửa nào mà bạn đang làm. Ta phải merge một cách thủ công sau khi công việc chỉnh sửa hoàn thành.  
        - Nếu branch local (giả sử là branch X) của ta muốn theo dõi và lấy data của nó trên remote (cũng là branch X) thì ta có thể sử dụng lệnh pull. Git pull thực chất là git fetch + git merge. Git sẽ tự động merge cho bạn.
     4. Pushing to your remote  
        - Khi ở một thời điểm nào đó, ta muốn share code của ta lên remote cho mọi người cùng sử dụng, ta sử dụng lệnh : [ git push <remote\_name> <branch\_name>], giả sử muốn push lên branch master của remote origin, ta dùng lệnh: git push origin master  
        - Cái command này chỉ hoạt động nếu ta vừa cloned từ server về và ta có quyền ghi và chưa ai push code lên trong khoảng thời gian ta đang sửa.   
        - Nếu có ai đó cũng clone và edit và push code lên trước ta, thì khi ta push code sẽ bị reject. Để được push, ta phải fetch rồi merge thì mới được phép đẩy lên tiếp.
     5. Inspecting a Remote  
        - Nếu ta muốn xem thông tin cụ thể của một remote, ta có thể sử dụng command: [git remote show <remote\_name>]  
          
          
        - nó show ra remote branch có bao nhiêu và local branch có bao nhiêu
     6. Renaming and Removing Remotes  
        - Ta có thể dùng lệnh : [git remote rename <old\_name> <new\_name>] để đổi shortname của remote  
          
          
        -   
          
        - Nếu như ta muốn xoá bỏ remote vì một lý do củ chuối nào đó, ta có thể sử dụng lệnh : [git remote remove <remote\_name>] hoặc là [git remote rm <remote\_name>]
  3. Tagging  
     - Giống như hầu hết các VCSs khác, Git có khả năng gắn thẻ các điểm quan trọng ở trong repository  
     - Thông thường, người dùng sử dụng chức năng đánh dấu relase point (v1.0, v2.0). Trong phần này ta học cách xem thêm sửa xoá tagging  
     - Tag nó giống như Label ở trong java, nó gán nhãn cái commit của chúng ta để chúng ta dễ tham chiếu đến cái commit đó hơn (giả sử trước mỗi lần release code, ta phải commit và đánh dấu version của ứng dụng. Ta sẽ gán Tag commit tương ứng với version release. Sau có muốn rollback lại version nào đó, ta chỉ cần checkout lại tag là xong)  
     - Tóm gọn lại, tag để tham chiếu tới commit
     1. Listing your tag  
        - Sử dụng command: [git tag] với optional option: [-l hoặc --list] để liệt kê các tag  
        - Sử dụng [git tag –l “text\_search”] để lọc
     2. Create tag  
        - Có 2 loại tag và lightweigh và annotated tag  
        - lightweigh tag rất giống với branch, nó chỉ trỏ tới commit  
        - Annotated tag lưu trữ full data vào git database, nó là một mã checksumed có chứa các thông tin : tagger name, email, date, tag message và được signed & verified với GNU Privacy Guard
     3. Annotated tags  
        - lệnh tạo annotated tag: [git tag –a <version> -m “<message>”]  
        - ví dụ: [git tag –a v1.4 –m “version 1.4”]  
        - Để check info của tag, ta dùng lệnh [git show]  
        - ví dụ: git show v1.4  
        Khi lên github, ta có thể thấy thông tin của tag như sau  
          
          
        
     4. Lightweight tag  
        - Một cách tạo tag đơn giản hơn annotated tag là lightweight tag. Về cơ bản nó là 1 mã checksumed được lưu ở trong 1 file, không có chứa thêm các thông tin phụ nào khác giống như annotated tag.  
        - Để tạo Lightweight tag, ta ko cần những option : -a, -s, -m mà đơn giản là: [git tag v1.4]
     5. Tag later  
        - Giả sử commit cũ của bạn chưa tạo tag, để tạo tag cho những commit cũ, ta sử dụng lệnh: [git tag –a <tag\_name> <checksumed>]  
          
          
        
     6. Sharing Tag  
        - Tất cả các lệnh tạo tag vừa rồi mới chỉ được tạo ở local, chưa được đẩy lên remote. Tag nó giống như branch, ta không thể commit mà đẩy lên master được, mà ta phải đẩy lên branch của chúng ta hay chính là đẩy lên cái tên tag vừa tạo. Để share tag lên remote, ta dùng lệnh: [git push <remote\_name> <tag\_name>]  
        - Ví dụ: [git push origin v1.4]  
        - Nếu ta có rất nhiều tag muốn push, ta dùng lệnh: [git push <remote\_name> --tags]  
        - Nếu ta chỉ muốn push annotated tag, ta dùng lệnh: [git push <remote\_name> --follow-tags]
     7. Deleting tag  
        - Để xoá 1 tag, ta dùng lệnh: [git tag –d <tag\_name>]  
        - Để cập nhật tag vừa xoá lên remote, ta dùng lệnh: [git push origin :refs/tags/<tag\_name>]  
          
          
        - Ngoài cách rườm rà này ra thì ta có cách dễ nhớ hơn đó là: [git push origin –delete <tag\_name>]  
        
     8. Checkout tag  
        - Nếu muốn xem lại các nội dung của các tag cũ, ta có thể checkout với tag đó. Khi checkout thì repository của ta ở trạng thái “deteach HEAD”, tạo nên một điểm xấu, nếu không cẩn thận sẽ làm mất code.  
        - để checkout, ta dùng lệnh : [git checkout <tag\_name>], ví dụ: [git checkout v1.4]  
          
        - Để tạo nhánh trong tag, ta dùng lệnh : [git checkout -b <branch\_name> <tag\_name>], ví dụ : [git checkout -b hot-fix v1.4]  
        
     9. Git Aliases  
        - Trước khi đến với cái chương tiếp theo, ta dừng lại ở Git Alias. Ở trong phần này, Git sẽ giới thiệu cho chúng ta một tính năng rất là hấp dẫn, đó chính là alias giúp chúng ta thao tác với Git nhanh hơn, dễ dàng hơn.   
        - Để đặt alias cho git , ta sử dụng command, [git config --global alias.<command\_alias> <command\_name>].s  
        - Flag --global là áp dụng cho user os, tuy nhiên ta có thể chỉ định toàn hệ thống git hoặc là chỉ 1 repository nào đó bằng cách thay bằng các flag --system, --global  
        - Ví dụ: git config --local alias.co checkout hoặc là git config --system alias.co checkout hoặc là git config --global alias.co checkout

1. Git Branching  
   - Để thực sự hiểu cách Git phân nhánh ta cần phải biết git lưu trữ dữ liệu ra sao ở phần giới thiệu về Git  
   - Khi ta commit, Git sẽ lưu 1 đối tượng commit có chứa con trỏ, trỏ đến nội dung mà ta staged. Object đó cũng chứa các thông tin về tên người commit, message và nội dung mà ta taged, contrỏ trỏ đến commit,…   
   - Khi tạo commit bằng lệnh : [git commit] , Git checksums từng subdirectory và lưu chúng giống như giống như Tree ở trên Git repository. Git tạo một Commit Object có chứa các metadata và trỏ đến root project tree và nó có thể dễ dàng tạo lại snapshot khi cần.  
   - Giả sử ta có 3 file cần commit là : README test.rb LICENSE, ta dùng lệnh   
   [git add README test.rb LICENSE] và [git commit -m “UPDATE”]  
     
     
   Git repository bây giờ có chứa 5 objects: 3 blobs (3 file commit), 1 tree list contents của directory và one commit pointer to root tree  
     
   Nếu như ta thay đổi trong repository và tạo một vài commit, commit đó sẽ tạo một con trỏ, trỏ đến commit trước đó.  
     
   - 1 branch ở trong Git đơn giản là một con trỏ di động trỏ tới một trong những commit.Branch mặc định là master. Khi ta thực hiện commit, branch sẽ trỏ đến commit cuối cùng của ta. Lúc nào mà ta commit thì con trỏ của branch sẽ move tự động.  
     
   1. Create New Branch  
      - Để tạo branch, ta dùng lệnh: [git branch <branch\_name>]. Nó sẽ tạo branch ngay nhưng chưa switch sang branch mới.  
      - Git đời mới cung cấp lệnh mới: [git switch –c <branch\_name>] tương tự lệnh tạo branch trên  
      - Chuyện gì sẽ xảy ra nếu như ta tạo mới một branch? Điều này sẽ dẫn tới Git sẽ tạo ra một con trỏ cũng trỏ tới commit hiện tại  
        
      - Vậy làm sao để ta biết ta đang ở branch nào ? Git giữ một con trỏ đặc biệt có tên. Ở trong Git, nó là một con trỏ ở Local Repository mà ta đang làm việc. Ở trong trường hợp này ta chỉ ở trên nhánh master. Command : [git branch] chỉ tạo mới branch chứ nó không checkout branch ngay lập tức.  
        
      Dựa vào hình trên ta có thể thấy, branch thì trỏ vào commit còn con trỏ HEAD thì trỏ vào branch. Sử dụng command [git log với flag: --decorate] Nó sẽ show mã SHA của commit, branch đang trỏ vào commit đó kèm theo con trỏ HEAD  
        
      Như trên ảnh ta có thể thấy, con trỏ HEAD trỏ vào các branch master, b8, b1 và tag: v2.2
   2. Switching Branch  
      Để switch branch, ta sử dụng [git checkout <branch\_name>], git đời mới cung cấp lệnh tương tự lệnh checkout là : [git switch <branch\_name>]  
      - [git switch -] => checkout lại branch gần nhất  
        
      khi ta checkout thì con trỏ HEAD sẽ trỏ vào branch mà ta switch  
        
        
      Khi ta commit, con trỏ sẽ nối thêm 1 bậc  
        
      Thật đặc biệt, con trỏ HEAD và testing tiến lên, còn master thì vẫn đang trỏ ở commit cũ  
        
        
      Khi nhánh master cũng commit => sẽ tạo ra 2 commit con nối với commit cha  
        
      
   3. Basic Branching and Merging  
      - Dạy về merge conflict
   4. Branch Management
      1. Lệnh [git branch] nếu không có đối số thì sẽ hiển thị danh sách các branch ở local (chỉ ở local chứ không phải tất cả branch). Nếu muốn xem thông tin branch và commit mới nhất của branch đó, ta sử dụng option : [git branch -v]  
           
           
         
      2. Để xem những branch nào đã được merge code vào nhánh hiện tại của mình (giả sử là merge nhánh hot-fix vào master) ta dùng lệnh : [git branch --merged]  
           
           
         Như ta thấy, nhánh hot-fix chưa được merge vào nhánh master, bây giờ ta thử merge vào:  
           
           
         Sau khi giải quyết xung đột, ta commit  
           
           
         Ta đã thấy nhánh hot-fix ở danh sách các branch đã merge  
         Những branch nào chưa được merge vào nhánh hiện tại, ta dùng lệnh [git branch --no-merged] để show  
           
           
         Để xoá branch, ta dùng lệnh : [git branch -d <branch\_name>]. Tuy nhiên, những branch chưa được merge vào master thì nó sẽ lỗi không xoá được :  
           
         Nếu như ta vẫn quyết định xoá, ta dùng lệnh force: [git branch -D <branch\_name>]  
           
         - Giả sử ta đang ở branch testing, muốn check xem có những branch nào chưa được merge vào branch master, ta dùng lệnh : [git branch --no-merge <branch\_name>] và kiểm tra branch nào được merge rồi tương tự như vậy.
      3. Changing a branch name  
         - Trước khi tìm hiểu nội dung của đổi branch name, khuyên anh em không nên đổi tên của các branch: master, main hoặc có vai trò y hệt như master  
         - Ta có thể đổi tên của branch mà vẫn giữ nguyên được history, ngoài ra cũng đổi được tên branch ở trên remote.   
         - Đổi tên branch local với command: [git branch --move <old\_branch\_name> <new\_branch\_name>]  
         - Để đổi branch name trên remote, ta dùng lệnh: [git push --set-upstream <remote\_name> <new\_branch\_name>]  
           
           
           
         Bây giờ ta có thể check toàn bộ branch (cả remote + local) của project với lệnh : [git branch --all]  
           
           
         Tuy đã đổi tên ở local, tuy nhiên ở remote vẫn còn cả branch cũ và branch mới, để xoá branch cũ đi, ta dùng command: [git push <remote\_name> --delete <old\_branch\_name>]  
         
   5. Branching workflows
      1. Long-Running Branch  
         - Bởi vì Git sử dụng simple three-way merge, merging 1 nhánh vào 1 nhánh khác rất là dễ thực hiện.  
         - Điều đó có nghĩa là ta có vài branch hoạt động liên tục (không bị xoá như các nhánh hot-fix. Giả sử nhánh master, developer, … sẽ tồn tại trong suốt flow, các nhánh hot-fix sau khi merge sang các nhánh master, developer thì sẽ bị xoá). Có rất nhiều Git Developer làm việc theo cách này, ví dụ như chỉ có mã code xịn, ổn định ở trên nhánh master. Đi song song với master là development và next. 2 nhánh trên để test độ ổn định của ứng dụng, khi nào code thực sự ổn định mới được merge vào master  
           
         
      2. Topic branch  
         - Topic branch rất là hữu dụng trong mọi dự án và mọi quy mô  
         - Một Topic branch là một nhánh tồn tại trong thời gian ngắn , sử dụng cho một tính năng cụ thể.  
         - 
      3. Remotes Branch  
         - Remote reference là các con trỏ (tham chiếu ) trong Remote Repository, bao gồm tags, branch.  
         - Ta có thể lấy toàn bộ danh sách remote reference bằng lệnh : [git ls-remote <remote>] (Để lấy toàn bộ remote, dùng lệnh : git remote -v)  
           
           
         Remote reference bao gồm các tags, các nhánh,…  
           
           
         Điều khác biệt giữa remote branch và local branch (ở ảnh trên)  
           
         Khi ta commit ở local, local branch sẽ trỏ như sau  
           
           
         Khi mà cả remote và local commit, trạng thái sẽ xảy ra như sau:  
           
           
           
         Khi ta add nhiều remote, sẽ xảy ra trạng thái phân nhánh sau  
           
         (add remote bằng lệnh : git remote add git://{url})  
         Cách checkout sang remote branch  
         git branch --all  
         
      4. Git rebasing  
         - Ở trong Git, có 2 cách chính để gộp code từ nhánh này sang nhánh khác đó là merge và rebase. Ở trong phần này, ta sẽ được biết thế nào là rebasing, làm sao để thao tác với nó, tại sao nó là công cụ tuyệt vời và trong trường hợp nào thì không sử dụng nó.  
         - Nếu như ở phần merging, các commit vẫn còn sau khi merge  
           
         Cách dễ dàng nhất để kết hợp 2 branch experiment với master là sử dụng câu lệnh merge.  
           
         Nó sẽ hợp nhất 2 commit C4 và C3 rồi tạo ra commit mới là C5.   
         - Tuy nhiên, trong một diễn biến khác, ta có thể áp dụng bản thay đổi của C4 vào C3 và đó gọi là rebasing (you can take the path of the change that was introduced in C4 and reapply it on top of C3). Đối với rebase, ta có thể mang toàn bộ commit của 1 branch này sang một branch khác.  
         

|  |
| --- |
| acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (master)  $ git checkout -b experiment  Switched to a new branch 'experiment'  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (experiment)  $ git commit -a -m "experiment update"  [experiment 9f4d86e] experiment update  1 file changed, 9 deletions(-)  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (experiment)  $ git push origin experiment  Enumerating objects: 5, done.  Counting objects: 100% (5/5), done.  Delta compression using up to 8 threads  Compressing objects: 100% (2/2), done.  Writing objects: 100% (3/3), 311 bytes | 311.00 KiB/s, done.  Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  remote:  remote: Create a pull request for 'experiment' on GitHub by visiting:  remote: https://github.com/vietthang197/git\_learning2/pull/new/experiment  remote:  To https://github.com/vietthang197/git\_learning2  \* [new branch] experiment -> experiment  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (experiment)  $ git checkout master  Switched to branch 'master'  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (master)  $ git commit -a -m "master update"  [master b809705] master update  1 file changed, 9 deletions(-)  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (master)  $ git push origin master  Enumerating objects: 5, done.  Counting objects: 100% (5/5), done.  Delta compression using up to 8 threads  Compressing objects: 100% (2/2), done.  Writing objects: 100% (3/3), 307 bytes | 307.00 KiB/s, done.  Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  To https://github.com/vietthang197/git\_learning2  be6771a..b809705 master -> master  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (master)  $ git checkout experiment  Switched to branch 'experiment'  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (experiment)  $ git rebase master  error: could not apply 9f4d86e... experiment update  Resolve all conflicts manually, mark them as resolved with  "git add/rm <conflicted\_files>", then run "git rebase --continue".  You can instead skip this commit: run "git rebase --skip".  To abort and get back to the state before "git rebase", run "git rebase --abort".  Could not apply 9f4d86e... experiment update  Auto-merging File1.txt  CONFLICT (content): Merge conflict in File1.txt  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (experiment|REBASE 1/1)  $ git rebase --continue  File1.txt: needs merge  You must edit all merge conflicts and then  mark them as resolved using git add  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (experiment|REBASE 1/1)  $ git add .  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (experiment|REBASE 1/1)  $ git rebase --continue  [detached HEAD da639b4] experiment update  1 file changed, 1 insertion(+)  Successfully rebased and updated refs/heads/experiment.  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (experiment)  $ git checkout master  Switched to branch 'master'  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (master)  $ git merge experiment  Updating b809705..da639b4  Fast-forward  File1.txt | 1 +  1 file changed, 1 insertion(+)  acer@DESKTOP-69MST1A MINGW64 /d/git\_learning\_2 (master)  $ git push origin master  Enumerating objects: 5, done.  Counting objects: 100% (5/5), done.  Delta compression using up to 8 threads  Compressing objects: 100% (2/2), done.  Writing objects: 100% (3/3), 332 bytes | 332.00 KiB/s, done.  Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  To https://github.com/vietthang197/git\_learning2  b809705..da639b4 master -> master |

* + 1. Giải thích về rebase  
       - Ta có thể thấy, sau khi rebase (nói nôm na là gộp code) từ experient về master thì sẽ tạo ra một commit mới, nhưng commit mới nó sẽ không kế thừa từ 2 commit cha là experient và master. Mà commit mới này là commit của master.  
         
       Còn đối với merge, graph nó sẽ ntn  
         
       Nó sẽ kéo xuống ntn  
       - Về bản chất cuối cùng thì rebase nó cũng không khác gì merge cả, nó chỉ giúp cái history nhìn nó gọn gàng hơn.
    2. More Interesting Rebase  
       - Giả sử ta có trường hợp sau:  
         
       Git đang phân nhánh như thế này  
       - Nhánh client được chia ra từ server để thực hiện một chức năng khác. Đến commit C9, nhánh client đã hoàn thiện chức năng muốn đẩy lên master. Tuy nhiên ta muốn chức năng mới ở master deploy lên cho khách hàng xem, muốn giữ chức năng cũ ở client. Ta hoàn toàn có thể rebase từ client merge vào master bằng command  
       [git rebase--onto master server client]  
         
       - Lúc này sau khi merge, master sẽ mới hơn server  
         
         
         
         
       - Sau khi server code xong, nó sẽ thực hiện lệnh merge master vào server bằng lệnh : [git rebase master server] (Điều này có nghĩa là lấy toàn bộ code của master để ghép vào nhánh server. Target brach là server => server sẽ chứa code mới nhất và vượt cả master)  
         
         
         
       - Lúc này để master chứa code mới nhất, ta dùng lệnh : [git merge server]  
         
         
       - Sau khi xử lý xong thì ta xoá cmn 2 brank client và server đi vì giờ code mới nhất đã được tích hợp ở nhánh master rồi  
         
       [git branch -d client, git branch -d server]  
       [git push --delete origin client, git push --delete origin server]  
         
         
       lúc này commit thằng tưng  
         
       
    3. The Perils of Rebasing (Nguy cơ tiềm tàng từ Rebasing)  
       - Chi tiết thì đọc tiến anh, dài quá ko dịch hết được
    4. So sánh Merge với Rebase  
       - Commit history là lịch sử các bản ghi mà ghi lại những gì đã xảy ra. Nó là một tài liệu lịch sử, có giá trị theo đúng nghĩa của nó và không nên giả mạo