

## Ghi chú dành cho chuyên gia

**Chapter 2: Browsing the history**

**Section 2.1: "Regular" Git Log**

**git log**

will display all your commits with the author and hash. This will be shown over multiple pages if you have many commits. Use the `-1` key to exit the log, which to show a single line per commit, look at `gitk`.

By default, with no arguments, `git log` lists the commits made in that repository in order – that is, the most recent commits show up first. As you can see, this command includes the author's name and email, the date written, and the source.

Example from `text/code/cats` repository:

```
commit 87e9575626a21c0298776f14d570050007cf  
Merge: e2a9fd6 87e95756  
Author: Brian...  
Date: Mon Mar 24 15:52:07 2014 -0700  
Merge pull request #7204 from BrianShaw/ttx/where-art-thou  
Fix '1%' type in Where Art Thou description  
commit c88a20bd516a2be4ed97f7c76615af27065  
Author: BrianShaw...  
Date: Mon Mar 24 21:13:36 2014 +0800  
Fix '1%' type in Where Art Thou description  
commit 82ff105a0975141550d435f317af102598e7  
Merge: 2653025 2653026  
Author: Krishna...  
Date: Thu Mar 26 14:04 2014 -0500  
Merge pull request #718 from destroythous/114/unnecessary-03  
Remove unnecessary comma from CONTRIBUTING.ad
```

If you wish to limit your command to last n commits log you can simply pass a parameter. For example, if you wish to log last 2 commits logs

**git notes for Professionals**

**Chapter 10: Committing**

**Parameter**

- `-m, --message`: Message to include in the commit. Specifying this parameter bypasses Git's normal behavior of opening an editor.
- `--amend`: Specify that the changes currently staged should be added (amended) to the previous commit. Be careful, this can rewrite history!
- `--no-edit`: Use the selected commit message without launching an editor. For example, `git commit --amend --no-edit` amends a commit without changing its commit message.
- `-v, --verbose`: Commit all changes, including changes that aren't yet staged.
- `-date`: Manually set the date that will be associated with the commit.
- `--only`: Commit only the paths specified. This will not commit what you currently have staged unless told to do so.
- `--patch, -p`: Use the interactive patch selection interface to chose which changes to commit.
- `--SshKeyID, -S, --gpg-sign, --sign-with-keyID, -S, --no-gpg-sign`: Sign commit, GPG-sign commit, commandname COMMIT, gpgsign configuration variable.
- `--no-verify`: This option bypasses the pre-commit and commit-msg hooks. See also Hooks.

Commits with Git provide accountability by attributing authors with changes to code. Git offers multiple features for the specificity and security of commits. This topic explains and demonstrates proper practices and procedures in committing with Git.

**Section 10.1: Stage and commit changes**

**The basics**

After making changes to your source code, you should **stage** these changes with Git before you can commit them. For example, if you change `README.md` and program `.py`:

```
git add README.md program.py
```

This tells git that you want to add the files to the next commit you do.

Then, commit your changes with:

```
git commit
```

Note that this will open a text editor, which is often vim. If you are not familiar with vim, you might want to know that you can press `I` to go into insert mode, write your commit message, then press `Esc` and `:wq` to save and exit the text editor, simply include the `-m` flag with your message:

```
git commit -m "Commit message here"
```

Commit messages often follow some specific formatting rules, see Good commit messages for more information.

**Shortcuts**

If you have changed a lot of files in the directory, rather than listing each one of them, you could use:

**git notes for Professionals**

**Chapter 25: Cloning Repositories**

**Section 25.1: Shallow Clone**

Cloning a huge repository (like a project with multiple years of history) might take a long time, or fail because of the amount of data to be transferred. In cases where you don't need to have the full history available, you can do a shallow clone:

```
git clone [repo_url] --depth 1
```

The above command will fetch just the last commit from the remote repository. Be aware that you may not be able to resolve merges in a shallow repository. It's often a good idea to take at least as many commits as you are going to need to backtrack to resolve merges. For example, to instead get the last 50 commits:

```
git clone [repo_url] --depth 50
```

Later, if required, you can then fetch the rest of the repository:

```
git fetch --unshallow
```

**Section 25.2: Regular Clone**

To download the entire repository including the full history and all branches, type:

```
git clone [url]
```

The example above will place it in a directory with the same name as the repository name.

To download the repository and save it in a specific directory, type:

```
git clone [url] [directory]
```

For more details, visit Clone a repository.

**Section 25.3: Clone a specific branch**

To clone a specific branch of a repository, type `--branch <branch_name>` before the repository url:

```
git clone --branch <branch_name> [url] [directory]
```

To use the shorthand option for `--branch`, type `-b`. This command downloads entire repository and checks out `<branch_name>`.

To save disk space you can clone history holding only to single branch with:

```
git clone --branch <branch_name> --single-branch [url] [directory]
```

**git Notes for Professionals**

# Hơn 100 trang

những gợi ý và thủ thuật chuyên nghiệp

# Nội dung

<u>Về</u>	1
<u>Chương 1: Bắt đầu với Git</u>	2
<u>Phần 1.1: Tạo kho lưu trữ đầu tiên của bạn, sau đó thêm và chuyển giao các tệp</u>	2
<u>Phần 1.2: Sao chép kho lưu trữ</u>	4
<u>Phần 1.3: Mã chia sẻ Phần</u>	4
<u>1.4: Đặt tên người dùng và email của bạn Phần</u>	5
<u>1.5: Thiết lập điều khiển từ xa ngược dòng</u>	6
<u>Phần 1.6: Tìm hiểu về lệnh Phần 1.7:</u>	6
<u>Thiết lập SSH cho Git Phần</u>	6
<u>1.8: Cài đặt Git</u>	7
<u>Chương 2: Duyệt lịch sử</u>	10
<u>Phần 2.1: Nhật ký Git "thông thường"</u>	10
<u>Phần 2.2: Nhật ký đẹp hơn</u>	11
<u>Phần 2.3: Tô màu nhật ký</u>	11
<u>Phần 2.4: Nhật ký trực tuyến</u>	11
<u>Phần 2.5: Tìm kiếm nhật ký Phần</u>	12
<u>2.6: Liệt kê tất cả các đóng góp được nhóm theo tên tác giả Phần 2.7: Tìm kiếm chuỗi cam kết trong nhật ký git</u>	12
<u>Phần 2.8: Nhật ký cho một phạm vi dòng trong một tệp</u>	14
<u>lọc</u>	14
<u>Phần 2.10: Nhật ký với các thay đổi nội tuyến Phần</u>	14
<u>2.11: Nhật ký hiển thị các tệp đã cam kết Phần 2.12:</u>	15
<u>Hiển thị nội dung của một cam kết duy nhất Phần 2.13: Nhật ký Git giữa hai nhánh Phần 2.14: Một dòng hiển thị tên và thời gian của người thực hiện kể từ khi cam kết</u>	15
<u>Chương 3: Làm việc với Điều khiển từ xa Phần 3.1: Xóa một nhánh từ xa</u>	16
<u>Phần 3.2: Thay đổi URL từ xa Git</u>	17
<u>Phần 3.3: Danh sách các điều khiển từ xa hiện có</u>	17
<u>Phần 3.4: Xóa các bản sao cục bộ của các nhánh từ xa đã xóa</u>	17
<u>Phần 3.5: Cập nhật từ kho lưu trữ ngược dòng</u>	17
<u>Chương 4: Dàn dựng</u>	18
<u>Phần 4.1: Sắp xếp tất cả các thay đổi đối với tệp</u>	21
<u>Hủy phân đoạn một tệp có chứa các thay đổi</u>	21
<u>Phần 4.3: Thêm thay đổi theo khối</u>	21
<u>Phần 4.4: Thêm tương tác</u>	22
<u>Phần 4.5: Hiển thị các thay đổi theo giai đoạn</u>	22
<u>Phần 4.6: Sắp xếp một tệp duy nhất</u>	23

Phần 4.7: Giai đoạn xóa tệp Chương 5: Bỏ qua tệp và thư mục Phần 5.1: Bỏ qua tệp và thư mục bằng tệp .gitignore	23
Phần 5.2: Kiểm tra xem tệp có bị bỏ qua hay không	24
Phần 5.3: Ngoại lệ trong tệp .gitignore	24
Tệp .gitignore toàn cầu Phần 5.5: Bỏ qua các tệp đã được cam kết vào kho Git	27
Phần 5.6: Bỏ qua các tệp cục bộ mà không tuân theo các quy tắc bỏ qua Phần 5.7: Bỏ qua các thay đổi tiếp theo đối với một tệp (mà không xóa nó)	28
Phần 5.8: Bỏ qua một tập tin trong bất kỳ thư mục nào	29
Phần 5.9: Mẫu .gitignore được diền sẵn	29
Mục 5.10: Bỏ qua các tệp trong thư mục con (Nhiều tệp .gitignore)	30
Phần 5.11: Tạo một thư mục trống Phần 5.12: Tìm các tệp bị .gitignore bỏ qua	31
Phần 5.13: Chỉ bỏ qua một phần của tệp [sơ khai]	32
Phần 5.14: Bỏ qua những thay đổi trong tệp được theo dõi [sơ khai]	33
Mục 5.15: Xóa các tệp đã được cam kết nhưng được bao gồm trong .gitignore	34
<b>Chương 6: Git Di</b> Phần 6.1:	35
Hiển thị sự khác biệt trong nhánh làm việc Phần 6.2: Hiển thị các thay đổi giữa hai lần xác nhận Phần 6.3: Hiển thị sự khác biệt cho các tệp được phân đoạn Phần 6.4: So sánh các nhánh Phần 6.5:	35
Hiển thị cả các thay đổi theo giai đoạn và không theo giai đoạn	36
Phần 6.6: Hiển thị sự khác biệt cho một tệp hoặc thư mục cụ thể	36
Mục 6.7: Xem từ-dài cho dòng dài	37
Phần 6.8: Hiển thị sự khác biệt giữa phiên bản hiện tại và phiên bản cuối cùng	37
Phần 6.9: Tạo một di động tương thích với bản vá	37
Mục 6.10: sự khác biệt giữa hai cam kết hoặc chỉ nhánh	38
Mục 6.11: Sử dụng meld để xem tất cả các sửa đổi trong thư mục làm việc	38
Mục 6.12: Tệp văn bản và tệp nhị phân mã hóa Di UTF-16	38
<b>Chương 7: Hoàn tác</b>	40
Phần 7.1: Quay lại cam kết trước đó Phần 7.2: Hoàn tác các thay đổi	40
Phần 7.3: Sử dụng reflog	41
Phần 7.4: Hoàn tác việc sáp nhập	41
Phần 7.5: Hoàn nguyên một số cam kết hiện có Phần 7.6: Hoàn tác/Làm lại một loạt các cam kết	43
<b>Chương 8: Sáp nhập</b>	45
Phần 8.1: Tự động sáp nhập	45
Phần 8.2: Tìm tất cả các nhánh không có thay đổi được hợp nhất	45
Phần 8.3: Hủy bỏ việc hợp nhất	45
Phần 8.4: Hợp nhất với một cam kết Phần 8.5: Chỉ giữ các thay đổi từ một phía của hợp nhất	45
Mục 8.6: Hợp nhất một nhánh vào một nhánh khác Chương 9: Các mô-đun con	46
<b>Chương 9: Các mô-đun con</b>	47
Phần 9.1: Nhận bản kho lưu trữ Git có các mô-đun con Phần 9.2: Cập nhật một mô-đun con Phần 9.3: Thêm một mô-đun con Phần 9.4: Thiết lập một mô-đun con đi theo một nhánh Phần 9.5: Di chuyển một mô-đun con	47
	48
	48

<u>Phần 9.6: Xóa một nhánh</u>	49
<u>Chương 10: Cam kết</u>	50
<u>Phần 10.1: Giải đoạn và cam kết thay đổi</u>	50
<u>Phần 10.2: Thông báo cam kết tốt</u>	51
<u>Mục 10.3: Sửa đổi một cam kết Phản</u>	52
<u>10.4: Cam kết mà không cần mở trình soạn thảo Phần 10.5:</u>	53
<u>Cam kết thay đổi trực tiếp</u>	53
<u>Phản 10.6: Chọn những dòng nào sẽ được sắp xếp để thực hiện</u>	53
<u>Mục 10.7: Tạo một cam kết trống Mục 10.8: Cam kết thay</u>	54
<u>mặt người khác Mục 10.9: Cam kết ký GPG</u>	54
<u>Mục 10.10: Cam kết các thay đổi</u>	54
<u>trong các tệp cụ thể Mục 10.11: Cam kết vào một</u>	55
<u>ngày cụ thể Mục 10.12: Sửa đổi thời gian của một cam kết Mục 10.13:</u>	55
<u>Sửa đổi tác giả của một cam kết Chương 11: Bí danh Phản</u>	55
<u>11.1: Bí danh đơn giản</u>	56
<u>Phản 11.2: Danh sách/tìm kiếm các bí danh</u>	56
<u>hiện có Phản 11.3: Bí danh nâng cao</u>	56
<u>Phản 11.4: Tạm thời bỏ qua các tệp được theo dõi Phản 11.5:</u>	57
<u>Hiển thị nhật ký đẹp với biểu đồ nhánh</u>	58
<u>Phản 11.6: Xem tệp nào đang bị cấu hình gitignore của bạn bỏ qua</u>	59
<u>Phản 11.7: Cập nhật mã trong khi vẫn giữ lịch sử tuyến tính</u>	60
<u>Phản 11.8: Tệp được phân loại không theo giải đoạn</u>	60
<u>Chương 12: Tái cơ sở</u>	61
<u>Phản 12.1: Tái khởi động chi nhánh địa phương</u>	61
<u>Phản 12.2: Rebase: của chúng ta và của họ, địa phương và từ xa</u>	61
<u>Phản 12.3: Rebase tương tác</u>	63
<u>Mục 12.4: Rebase trở lại cam kết ban đầu</u>	64
<u>Phản 12.5: Định cấu hình autostash</u>	64
<u>Phản 12.6: Kiểm tra tất cả các cam kết trong quá</u>	65
<u>trình rebase Phản 12.7: Khởi động lại trước khi</u>	65
<u>xem xét mã Phản 12.8: Hủy bỏ một Rebase tương</u>	67
<u>tác Phản 12.9: Thiết lập git-pull để tự động thực hiện rebase thay vì hợp nhất</u>	68
<u>Mục 12.10: Đẩy lùi sau đợt rebase</u>	68
<u>Chương 13: Cấu hình</u>	69
<u>Phản 13.1: Cài đặt trình soạn thảo nào sẽ sử dụng Phản</u>	69
<u>13.2: Tự động sửa lỗi chính tả</u>	69
<u>Phản 13.3: Liệt kê và chỉnh sửa cấu hình hiện tại</u>	70
<u>Phản 13.4: Tên người dùng và địa chỉ email</u>	70
<u>Phản 13.5: Nhiều tên người dùng và địa chỉ email Phản 13.6: Nhiều cấu</u>	70
<u>hình git</u>	71
<u>Phản 13.7: Cấu hình kết thúc dòng</u>	72
<u>Phản 13.8: Cấu hình chỉ cho một lệnh</u>	72
<u>Phản 13.9: Thiết lập proxy</u>	72
<u>Chương 14: Phân nhánh</u>	74
<u>Phản 14.1: Tạo và kiểm tra các nhánh mới Phản 14.2: Liệt kê các nhánh Phản</u>	74
<u>14.3: Xóa một nhánh từ xa</u>	75
<u>Mục 14.4: Chuyển nhanh về nhánh trước</u>	76

<u>Phần 14.5: Kiểm tra một nhánh mới theo dõi một nhánh từ xa</u>	76
<u>Phần 14.6: Xóa một nhánh cục bộ</u>	76
<u>Phần 14.7: Tạo một nhánh mồi côi (tức là nhánh không có cam kết chính)</u>	77
<u>Mục 14.8: Đổi tên chi nhánh</u>	77
<u>Mục 14.9: Tìm kiếm trong nhánh Mục</u>	77
<u>14.10: Đẩy nhánh tới remote</u>	77
<u>Mục 14.11: Di chuyển nhánh HEAD hiện tại sang một cam kết tùy ý</u>	78
<u>Chương 15: Danh sách Rev</u>	79
<u>Mục 15.1: Liệt kê các cam kết trong master nhưng không có trong Origin/master</u>	79
<u>Chương 16: Nghiem nát</u>	80
<u>Phần 16.1: Xóa các cam kết gần đây mà không cần khởi động lại</u>	80
<u>Phần 16.2: Cam kết đè bẹp trong quá trình hợp nhất</u>	80
<u>Phần 16.3: Cam kết xóa trong quá trình rebase Phần 16.4: Tự động</u>	80
<u>xóa và sửa lỗi</u>	81
<u>Phần 16.5: Autosquash: Cam kết mã bạn muốn xóa trong quá trình rebase</u>	82
<u>Chương 17: Chọn anh đào</u>	83
<u>Phần 17.1: Sao chép một cam kết từ nhánh này sang nhánh khác</u>	83
<u>Sao chép một loạt các cam kết từ nhánh này sang nhánh khác Phần 17.3: Kiểm tra xem có cần chọn anh đào không</u>	83
<u>Phần 17.4: Tìm các cam kết chưa được áp dụng cho thư mục</u>	84
<u>Chương 18: Phục hồi</u>	85
<u>Phần 18.1: Khôi phục sau khi thiết lập lại</u>	85
<u>18.2: Khôi phục từ git stash Phần 18.3: Khôi phục từ một cam kết bị mất</u>	85
<u>Phần 18.4: Khôi phục tệp đã xóa sau khi cam kết</u>	86
<u>Mục 18.5: Khôi phục file về phiên bản trước</u>	86
<u>Mục 18.6: Khôi phục nhánh đã xóa</u>	87
<u>Chương 19: Git Clean</u>	88
<u>Phần 19.1: Làm sạch tương tác</u>	88
<u>Phần 19.2: Loại bỏ mạnh mẽ các tệp không bị theo dõi</u>	88
<u>19.3: Làm sạch các tệp bị bỏ qua</u>	88
<u>Phần 19.4: Làm sạch tất cả các thư mục không bị theo dõi</u>	88
<u>Chương 20: Sử dụng tệp .gitattributes</u>	90
<u>Phần 20.1: Chuẩn hóa kết thúc dòng tự động</u>	90
<u>Phân định tệp nhị phân Phần 20.3: Các mẫu .gitattribute được diễn sẵn</u>	90
<u>Mục 20.4: Tắt chuẩn hóa kết thúc dòng</u>	90
<u>Chương 21: Tệp .mailmap: Liên kết người đóng góp và bí danh email</u>	91
<u>Phần 21.1: Hợp nhất những người đóng góp theo bí danh để hiển thị số lượng cam kết trong shortlog</u>	91
<u>Chương 22: Phân tích các loại quy trình công việc</u>	92
<u>Phần 22.1: Quy trình làm việc tập trung</u>	92
<u>Phần 22.2: Quy trình làm việc của Gitflow</u>	93
<u>Phần 22.3: Quy trình làm việc của nhánh tính năng</u>	95
<u>Phần 22.4: Luồng GitHub</u>	95
<u>Phần 22.5: Quy trình phân nhánh</u>	96
<u>Kéo</u>	97
<u>Phần 23.1: Kéo các thay đổi vào kho lưu trữ cục bộ</u>	97
<u>Phần 23.2: Cập nhật các thay đổi cục bộ</u>	98
<u>Mục 23.3: Kéo, ghi đè cục bộ</u>	98

Mục 23.4: Kéo code từ xa	98
Mục 23.5: Lưu giữ lịch sử tuyển tính khi kéo	98
Mục 23.6: Kéo, "quyền bị từ chối"	99
<b>Chương 24: Cái móc</b>	100
Mục 24.1: Dây trước	100
Phần 24.2: Xác minh bản dựng Maven (hoặc hệ thống bản dựng khác) trước khi cam kết	101
Phần 24.3: Tự động chuyển tiếp một số lần đầy tới các kho lưu trữ khác	101
Phần 24.4: Thông báo cam kết	102
Mục 24.5: Móc cục bộ	102
Mục 24.6: Sau thanh toán	102
Mục 24.7: Sau cam kết	103
Mục 24.8: Sau khi nhận	103
Mục 24.9: Cam kết trước	103
Mục 24.10: Chuẩn bị thông báo cam kết	103
Mục 24.11: Trước cuộc nổi loạn	103
Mục 24.12: Nhận trước	104
Mục 24.13: Cập nhật	104
<b>Chương 25: Nhận bản kho lưu trữ</b>	105
Mục 25.1: Bản sao nóng	105
Phần 25.2: Bản sao thông thường Phần	105
25.3: Sao chép một nhánh cụ thể Phần 25.4: Sao	105
chép đệ quy	106
Phần 25.5: Sao chép bằng proxy	106
<b>Chương 26: Dánh dấu</b>	107
Phần 26.1: Stash là gì?	107
Phần 26.2: Tạo kho lưu trữ	108
Phần 26.3: Áp dụng và xóa stash Phần 26.4: Áp dụng	109
stash mà không xóa nó	109
Mục 26.5: Hiển thị kho lưu trữ	109
Mục 26.6: Lưu trữ một phần	109
Phần 26.7: Liệt kê các stash đã lưu	110
Phần 26.8: Di chuyển công việc đang thực hiện của bạn sang một nhánh khác Phần	110
26.9: Xóa stash	110
Phần 26.10: Áp dụng một phần của kho lưu trữ có tính năng thanh	110
toán Phần 26.11: Khôi phục các thay đổi trước đó từ kho lưu trữ Phần	110
26.12: Tích trữ tương tác	111
Mục 26.13: Khôi phục kho lưu trữ bị đánh rơi Chương	111
<b>27: Cây con</b>	113
Phần 27.1: Tạo, Kéo và Backport Cây con Chương 28: Dài	113
<b>tên</b>	114
Phần 28.1: Đổi tên thư mục	114
Mục 28.2: đổi tên nhánh cục bộ và nhánh từ xa	114
Mục 28.3: Đổi tên chi nhánh địa phương	114
<b>Chương 29: Dây mành</b>	115
Mục 29.1: Dây một đối tượng cụ thể đến một nhánh ở xa Mục 29.2: Dây	115
	116
Mục 29.3: Lực dây	117
Phần 29.4: Thẻ dây	117
Mục 29.5: Thay đổi hành vi dây mặc định	117

<u>Chương 30: Nội bộ</u>	119
Phản 30.1: Repo	119
Mục 30.2: Đối tượng	119
Mục 30.3: HEAD ref	119
Mục 30.4: Tài liệu tham khảo	119
Phản 30.5: Đối tượng cam kết	120
Phản 30.6: Đối tượng cây	121
Phản 30.7: Đối tượng Blob	121
Phản 30.8: Tạo cam kết mới	122
Phản 30.9: Di chuyển	122
HEAD	122
chiều xung quanh	122
chiều mới	122
Chương 31: git-	122
tfs	123
Phản 31.1: bản sao git-tfs	123
Phản 31.2: bản sao git-tfs từ kho git tràn	123
Phản 31.3: cài đặt git-tfs qua Chocolatey	123
Phản 31.4: git tfs Kiểm tra	123
tfs push	123
<u>Chương 32: Các thư mục trống trong Git</u>	124
Phản 32.1: Git không theo dõi các thư mục	124
<u>Chương 33: git-svn</u>	125
Phản 33.1: Nhận bản kho SVN	125
Mục 33.2: Đẩy các thay đổi cục bộ sang SVN	125
Mục 33.3: Làm việc cục bộ	125
Mục 33.4: Nhận những thay đổi mới nhất từ SVN	126
33.5: Xử lý các thư mục trống	126
<u>Chương 34: Lưu trữ</u>	127
Phản 34.1: Tạo kho lưu trữ git kho lưu trữ	127
Phản 34.2: Tạo kho lưu trữ git với tiền tố thư mục	127
Phản 34.3: Tạo kho lưu trữ git dựa trên nhánh, bản sửa đổi, thẻ hoặc thư mục cụ thể	128
<u>Chương 35: Viết lại lịch sử với filter-branch</u>	129
Phản 35.1: Thay đổi tác giả của các cam kết	129
Phản 35.2: Đặt committer bằng tác giả cam kết	129
<u>Chương 36: Di chuyển sang Git</u>	130
Mục 36.1: SubGit	130
Mục 36.2: Di chuyển từ SVN sang Git bằng tiện ích chuyển đổi Atlassian	130
Phản 36.3: Di chuyển Mercurial sang Git	131
36.4: Di chuyển từ Team Foundation Version Control (TFVC) sang Git	131
Phản 36.5: Di chuyển từ SVN sang Git bằng svn2git	132
<u>Chương 37: Trình diễn</u>	133
Phản 37.1: Tổng quan	133
<u>Chương 38: Giải quyết xung đột hợp nhất</u>	134
38.1: Giải quyết thủ công	134
<u>Chương 39: Gói Phản</u>	135
39.1: Tạo gói git trên máy cục bộ và sử dụng nó trên máy khác	135
<u>Chương 40: Hiển thị lịch sử cam kết bằng đồ họa với Gitk</u>	136
Mục 40.1: Hiển thị lịch sử cam kết cho một tệp	136
Hiển thị tất cả các cam kết giữa hai lần xác nhận	136
Mục 40.2: Hiển thị các cam kết kể từ phiên bản	136

<u>Chương 41: Chia đôi/Tìm kiếm các cam kết bị lỗi</u>	137
<u>Phản 41.1: Tìm kiếm nhị phân (git bisect)</u>	137
<u>Mục 41.2: Bán tự động tìm một cam kết bị lỗi</u>	137
<u>Chương 42: Đỗ lỗi</u>	139
<u>Phản 42.1: Chỉ hiển thị một số dòng</u>	139
<u>Nhất định Phản 42.2: Để tìm ra ai đã thay đổi tệp</u>	139
<u>Phản 42.3: Hiển thi cam kết sửa đổi một dòng làn cuối</u>	140
<u>Phản 42.4: Bỏ qua những thay đổi chỉ có khoảng trắng</u>	140
<u>Chương 43: Cú pháp sửa đổi Git</u>	141
<u>Mục 43.1: Chỉ định sửa đổi theo tên đối tượng Mục</u>	141
<u>Mục 43.2: Tên tham chiếu tương trưng: nhánh, thẻ, nhánh theo dõi từ xa</u>	141
<u>Bản sửa đổi mặc định: HEAD</u>	141
<u>Mục 43.4: Tham chiếu lại nhật ký: &lt;refname&gt;@(&lt;n&gt;)</u>	141
<u>Mục 43.5: Tham chiếu lại nhật ký: &lt;refname&gt;@(&lt;date&gt;)</u>	142
<u>Mục 43.6: Nhánh được theo dõi/ngược dòng: &lt;branchname&gt;@{upstream}</u>	142
<u>Mục 43.7: Cam kết chuỗi tổ tiên: &lt;rev&gt;^, &lt;rev&gt;-&lt;n&gt;, v.v.</u>	142
<u>Mục 43.8: Dereferencing các nhánh và thẻ: &lt;rev&gt;^0, &lt;rev&gt;^{&lt;type&gt;}</u>	143
<u>Mục 43.9: Cam kết phù hợp trè nhất: &lt;rev&gt;^&lt;text&gt;, :/&lt;text&gt;</u>	143
<u>Chương 44: Cây làm việc</u>	145
<u>Phản 44.1: Sử dụng cây công việc</u>	145
<u>Phản 44.2: Di chuyển cây công việc</u>	145
<u>Chương 45: Điều khiển từ xa Git</u>	147
<u>Phản 45.1: Hiển thị kho lưu trữ từ xa</u>	147
<u>Phản 45.2: Thay đổi url từ xa của kho lưu trữ Git của bạn</u>	147
<u>Phản 45.3: Xóa kho lưu trữ từ xa</u>	148
<u>Phản 45.4: Thêm kho lưu trữ từ xa</u>	148
<u>Phản 45.5: Hiển thị thêm thông tin về kho lưu trữ từ xa</u>	148
<u>Phản 45.6: Đổi tên kho lưu trữ từ xa</u>	149
<u>Chương 46: Lưu trữ tệp lớn Git (LFS)</u>	150
<u>Mục 46.1: Khai báo một số loại tệp nhất định để lưu trữ bên ngoài</u>	150
<u>Mục 46.2: Đặt cấu hình LFS cho tất cả các bản sao</u>	150
<u>Mục 46.3: Cài đặt LFS</u>	150
<u>Chương 47: Bản vá Git</u>	151
<u>Phản 47.1: Tạo bản vá</u>	151
<u>Mục 47.2: Áp dụng các bản vá</u>	152
<u>Chương 48: Thông kê Git</u>	153
<u>Phản 48.1: Dòng mã cho mỗi nhà phát triển</u>	153
<u>Phản 48.2: Liệt kê từng nhánh và ngày sửa đổi cuối cùng của nó</u>	153
<u>Phản 48.3: Cam kết cho mỗi nhà phát triển</u>	153
<u>Mục 48.4: Số lần cam kết mỗi ngày</u>	154
<u>Phản 48.5: Tổng số lần xác nhận trong một nhánh</u>	154
<u>Phản 48.6: Liệt kê tất cả các cam kết ở định dạng đẹp</u>	154
<u>Tìm tất cả các kho lưu trữ Git cục bộ trên máy tính</u>	154
<u>Mục 48.8: Hiển thị tổng số commit của mỗi tác giả</u>	154
<u>Sử dụng git send-email với Gmail</u>	155
<u>Mục 49.2: Soạn thảo</u>	155
<u>Phản 49.3: Gửi bản vá qua thư</u>	155
<u>50: Máy khách GUI Git</u>	157

<u>Phần 50: gitk và git-gui</u>	157
<u>Phần 50.2: Máy tính để bàn GitHub</u>	158
<u>Phần 50.3: Git Kraken</u>	159
<u>Mục 50.4: Cây nguồn</u>	159
<u>Mục 50.5: Tiện ích mở rộng Git</u>	159
<u>Mục 50.6: SmartGit</u>	159
<u>Chương 51: Reflog – Khôi phục các cam kết không được hiển thị trong nhật ký git</u>	160
<u>Phần 51.1: Khôi phục sau một cuộc nổ dây tài tệ</u>	160
<u>Chương 52: TortoiseGit</u>	161
<u>Mục 52.1: Cam kết Squash</u>	161
<u>Mục 52.2: Giả sử không thay đổi</u>	162
<u>Phần 52.3: Bỏ qua tệp và thư mục</u>	164
<u>Phân nhánh</u>	165
<u>Chương 53: Hợp nhất bên ngoài và ditools</u>	167
<u>Phần 53.1: Thiết lập KDiff3 làm công cụ hợp nhất</u>	167
<u>Phần 53.2: Thiết lập KDiff3 làm công cụ di chuyển</u>	167
<u>Phần 53.3: Thiết lập IntelliJ IDE làm công cụ hợp nhất (Windows)</u>	167
<u>Mục 53.4: Thiết lập IntelliJ IDE làm công cụ di động (Windows)</u>	167
<u>Mục 53.5: Thiết lập Beyond Compare</u>	168
<u>Chương 54: Cập nhật tên đối tượng trong tài liệu tham khảo</u>	169
<u>Phần 54.1: Cập nhật tên đối tượng trong tài liệu tham khảo</u>	169
<u>Chương 55: Tên nhánh Git trên Bash Ubuntu</u>	170
<u>Mục 55.1: Tên chỉ nhánh trong terminal</u>	170
<u>Chương 56: Móc phía máy khách Git</u>	171
<u>Phần 56.1: Móc đầy trước Git</u>	171
<u>Phần 56.2: Cài đặt Hook</u>	172
<u>Chương 57: Git rerere</u>	173
<u>Phần 57.1: Kích hoạt rerere</u>	173
<u>58: Thay đổi tên kho git</u>	174
<u>Phần 58.1: Thay đổi cài đặt cục bộ</u>	174
<u>Chương 59: Cẩn thận Git</u>	175
<u>Phần 59.1: Liệt kê tất cả các thẻ có sẵn</u>	175
<u>Phần 59.2: Tạo và đầy (các) thẻ trong GIT</u>	175
<u>Chương 60: Sắp xếp kho lưu trữ cục bộ và từ xa của bạn</u>	177
<u>Mục 60.1: Xóa các nhánh cục bộ đã bị xóa trên remote</u>	177
<u>Chương 61: di-cây</u>	178
<u>Phần 61.1: Xem các tệp đã thay đổi trong một cam kết cụ thể</u>	178
<u>Phần 61.2: Cách sử dụng</u>	178
<u>Mục 61.3: Các phương án di chuyển chung</u>	178
<u>Tín dụng</u>	179
<u>Bạn cũng có thể thích</u>	186

## Về

Vui lòng chia sẻ miễn phí bản PDF này với bất kỳ ai, phiên bản mới nhất của cuốn sách này có thể được tải xuống từ:

<https://goalkicker.com/GitBook>

Cuốn sách Ghi chú Git® dành cho Chuyên gia này được biên soạn từ Tài liệu về Stack Overflow, nội dung được viết bởi những người đẹp tại Stack Overflow.

Nội dung văn bản được phát hành theo Creative Commons BY-SA, xem phần ghi công ở cuối cuốn sách này về những người đã đóng góp cho các chương khác nhau. Hình ảnh có thể là bản quyền của chủ sở hữu tương ứng trừ khi có quy định khác

Đây là một cuốn sách miễn phí không chính thức được tạo ra cho mục đích giáo dục và không liên kết với (các) nhóm hoặc công ty Git® chính thức cũng như Stack Overflow. Tất cả các nhãn hiệu và nhãn hiệu đã đăng ký là tài sản của chủ sở hữu tương ứng của họ chủ sở hữu công ty

Thông tin được trình bày trong cuốn sách này không được đảm bảo là đúng và chính xác, bạn phải tự chịu rủi ro khi sử dụng

Vui lòng gửi phản hồi và chỉnh sửa tới [web@petercv.com](mailto:web@petercv.com)

# Chương 1: Bắt đầu với Git

Ngày phát hành phiên bản

[2.13](#) 2017-05-10

[2.12](#) 24-02-2017

[2.11.1](#) 2017-02-02

[2.11](#) 29-11-2016

[2.10.2](#) 28-10-2016

[2.10](#) 2016-09-02

[2.9](#) 2016-06-13

[2.8](#) 28-03-2016

[2.7](#) 2015-10-04

[2.6](#) 28-09-2015

[2.5](#) 27-07-2015

[2.4](#) 30-04-2015

[2.3](#) 2015-02-05

[2.2](#) 26-11-2014

[2.1](#) 16-08-2014

[2.0](#) 28-05-2014

[1.9](#) 14-02-2014

[1.8.3](#) 24-05-2013

[1.8](#) 21-10-2012

[1.7.10](#) 2012-04-06

[1.7](#) 13-02-2010

[1.6.5](#) 2009-10-10

[1.6.3](#) 2009-05-07

[1.6](#) 17-08-2008

[1.5.3](#) 2007-09-02

[1.5](#) 14-02-2007

[1.4](#) 2006-06-10

[1.3](#) 18-04-2006

[1.2](#) 2006-02-12

[1.1](#) 2006-01-08

[1.0](#) 21-12-2005

[0.99](#) 2005-07-11

Phần 1.1: Tạo kho lưu trữ đầu tiên của bạn, sau đó thêm và cam kết các tập tin

Tại dòng lệnh, trước tiên hãy xác minh rằng bạn đã cài đặt Git:

Trên tất cả các hệ điều hành:

```
git --version
```

Trên các hệ điều hành giống UNIX:

## git nào

Nếu không có gì được trả về hoặc lệnh không được nhận dạng, bạn có thể phải cài đặt Git trên hệ thống của mình bằng cách tải xuống và chạy trình cài đặt. Xem [trang chủ Git](#) để có hướng dẫn cài đặt đặc biệt rõ ràng và dễ dàng.

Sau khi cài đặt Git, hãy định cấu hình tên người dùng và địa chỉ email của bạn. Làm điều này trước khi thực hiện một cam kết.

Sau khi Git được cài đặt, hãy điều hướng đến thư mục bạn muốn đặt dưới sự kiểm soát phiên bản và tạo kho lưu trữ Git trống:

## khởi tạo git

Việc này sẽ tạo ra một thư mục ẩn, .git, chứa hệ thống ông nước cần thiết để Git hoạt động.

Tiếp theo, kiểm tra xem Git sẽ thêm những tập tin nào vào kho lưu trữ mới của bạn; Bước này đáng được quan tâm đặc biệt:

## trạng thái git

Xem lại danh sách kết quả của các tập tin; bạn có thể cho Git biết nên đặt tệp nào vào kiểm soát phiên bản (tránh thêm tệp có thông tin bí mật như mật khẩu hoặc tệp chỉ làm lộn xộn kho lưu trữ):

```
git add <tên file/thư mục #1> <tên file/thư mục #2> < ... >
```

Nếu tất cả các tệp trong danh sách phải được chia sẻ với tất cả những người có quyền truy cập vào kho lưu trữ, thì một lệnh duy nhất sẽ thêm mọi thứ trong thư mục hiện tại của bạn và các thư mục con của nó:

## git thêm .

Điều này sẽ "giai đoạn" tất cả các tệp sẽ được thêm vào kiểm soát phiên bản, chuẩn bị cho chúng được cam kết trong lần cam kết đầu tiên của bạn.

Đối với các tệp mà bạn không bao giờ muốn kiểm soát phiên bản, hãy tạo và điền vào tệp có tên .gitignore trước khi chạy lệnh thêm .

Cam kết tất cả các tệp đã được thêm vào, cùng với thông báo cam kết:

## git commit -m "Cam kết ban đầu"

Điều này tạo ra một cam kết mới với thông báo đã cho. Một cam kết giống như một bản lưu hoặc ảnh chụp nhanh toàn bộ dự án của bạn. Bây giờ bạn có thể đẩy hoặc tải nó lên kho lưu trữ từ xa và sau đó bạn có thể quay lại kho lưu trữ đó nếu cần.

Nếu bạn bỏ qua tham số -m , trình soạn thảo mặc định của bạn sẽ mở và bạn có thể chỉnh sửa và lưu thông báo cam kết ở đó.

Thêm một điều khiển từ xa

Để thêm một điều khiển từ xa mới, hãy sử dụng lệnh `git remote add` trên terminal, trong thư mục kho lưu trữ của bạn được lưu trữ tại.

Lệnh `git remote add` có hai đối số:

1. Tên từ xa, ví dụ: Origin 2. URL từ xa,  
chẳng hạn như `https://<your-git-service-address>/user/repo.git`

```
git từ xa thêm nguồn gốc https://<your-git-service-address>/owner/repository.git
```

LƯU Ý: Trước khi thêm điều khiển từ xa, bạn phải tạo kho lưu trữ cần thiết trong dịch vụ git của mình. Bạn sẽ có thể Đẩy/kéo các cam kết sau khi thêm điều khiển từ xa.

## Phần 1.2: Sao chép kho lưu trữ

Lệnh `git clone` được sử dụng để sao chép kho lưu trữ Git hiện có từ máy chủ sang máy cục bộ.

Ví dụ: để sao chép dự án GitHub:

```
cd <đường dẫn nơi bạn muốn bản sao tạo thư mục> git clone https://github.com/  
username/projectname.git
```

Để sao chép dự án BitBucket:

```
cd <đường dẫn nơi bạn muốn bản sao tạo thư mục> git clone https://  
yourusername@bitbucket.org/username/projectname.git
```

Điều này tạo ra một thư mục có tên `projectname` trên máy cục bộ, chứa tất cả các tệp trong kho Git từ xa. Điều này bao gồm các tệp nguồn cho dự án cũng như thư mục con `.git` chứa toàn bộ lịch sử và cấu hình cho dự án.

Để chỉ định tên khác của thư mục, ví dụ `MyFolder`:

```
git clone https://github.com/username/projectname.git MyFolder
```

Hoặc để sao chép trong thư mục hiện tại:

```
git clone https://github.com/username/projectname.git .
```

Ghi chú:

1. Khi sao chép vào một thư mục được chỉ định, thư mục đó phải trống hoặc không tồn tại.

2. Bạn cũng có thể sử dụng phiên bản `ssh` của lệnh:

```
git clone git@github.com:tên người dùng/tên dự án.git
```

Phiên bản `https` và phiên bản `ssh` tương đương nhau. Tuy nhiên, một số dịch vụ lưu trữ như GitHub **khuyên bạn nên rằng bạn sử dụng `https`** thay vì `ssh`.

## Phần 1.3: Chia sẻ mã

Để chia sẻ mã của mình, bạn tạo một kho lưu trữ trên máy chủ từ xa mà bạn sẽ sao chép kho lưu trữ cục bộ của mình vào đó.

Để giảm thiểu việc sử dụng dung lượng trên máy chủ từ xa, bạn tạo một kho lưu trữ trống: một kho lưu trữ chỉ có các đối tượng `.git` và không tạo bản sao hoạt động trong hệ thống tệp. Như một phần thường, bạn đặt điều khiển từ xa này làm máy chủ ngược tuyển để dễ dàng chia sẻ thông tin cập nhật với các lập trình viên khác.

Trên máy chủ từ xa:

```
git init --bare /path/to/repo.git
```

Trên máy cục bộ:

```
git từ xa thêm Origin ssh://username@server:/path/to/repo.git
```

(Lưu ý rằng ssh: chỉ là một cách khả thi để truy cập kho lưu trữ từ xa.)

Bây giờ sao chép kho lưu trữ cục bộ của bạn vào điều khiển từ xa:

```
git push --set-upstream Origin master
```

Việc thêm `--set-upstream` (hoặc `-u`) đã tạo một tham chiếu ngược dòng (theo dõi) được sử dụng bởi các lệnh Git không có đối số, ví dụ như `git pull`.

## Phần 1.4: Đặt tên người dùng và email của bạn

Bạn cần xác định mình là `ai` \*trước khi\* tạo bất kỳ cam kết nào. Điều đó sẽ cho phép các cam kết có đúng tên tác giả và email liên kết với chúng.

Nó không liên quan gì đến xác thực khi đẩy tới kho lưu trữ từ xa (ví dụ: khi đẩy tới kho lưu trữ từ xa bằng tài khoản GitHub, BitBucket hoặc GitLab của bạn)

Để khai báo danh tính đó cho tất cả các kho lưu trữ, hãy sử dụng `git config --global`

Điều này sẽ lưu cài đặt trong tệp `.gitconfig` của người dùng của bạn : ví dụ : `$HOME/.gitconfig` hoặc đối với Windows, `%USERPROFILE%\gitconfig`.

```
git config --global user.name "Tên của bạn" git config --global user.email mail@example.com
```

Để khai báo danh tính cho một kho lưu trữ, hãy sử dụng `git config` bên trong kho lưu trữ.

Điều này sẽ lưu trữ cài đặt bên trong kho lưu trữ riêng lẻ, trong tệp `$GIT_DIR/config`. ví dụ: `/path/to/your/repo/.git/config`.

```
cd /path/to/my/repo git config user.name "Đăng nhập của bạn tại nơi làm việc" git config user.email mail_at_work@example.com
```

Các cài đặt được lưu trữ trong tệp cấu hình của kho lưu trữ sẽ được ưu tiên hơn cấu hình chung khi bạn sử dụng kho lưu trữ đó.

Mẹo: nếu bạn có các danh tính khác nhau (một cho dự án nguồn mở, một cho công việc, một cho các kho lưu trữ riêng tư, ...) và bạn không muốn quên đặt đúng danh tính cho từng kho lưu trữ khác nhau mà bạn đang làm việc :

- Xóa danh tính toàn cầu

```
git config --global --remove-section user.name git config --global --remove-section user.email
```

Phiên bản 2.8

- Để buộc git chỉ tìm kiếm danh tính của bạn trong cài đặt của kho lưu trữ, không phải trong cấu hình chung:

```
cấu hình git --global user.useConfigOnly true
```

Bằng cách đó, nếu bạn quên đặt `user.name` và `user.email` cho một kho lưu trữ nhất định và cố gắng thực hiện cam kết, bạn sẽ thấy:

không có tên nào được cung cấp và tính năng tự động phát hiện bị tắt  
không có email nào được cung cấp và tính năng tự động phát hiện bị tắt

## Phần 1.5: Thiết lập remote ngược dòng

Nếu bạn đã sao chép một nhánh phân nhánh (ví dụ: một dự án nguồn mở trên Github), bạn có thể không có quyền truy cập đầy vào kho lưu trữ ngược dòng, vì vậy bạn cần cả nhánh phân nhánh của mình nhưng có thể tìm nạp kho lưu trữ ngược dòng.

Đầu tiên hãy kiểm tra tên từ xa:

```
$ git remote -v
https://github.com/myusername/repo.git (fetch) Origin https://
git@github.com/myusername/repo.git (push) Origin upstream # dòng này có thể có hoặc
đây
```

Nếu thương nguồn đã có sẵn (có trên một số phiên bản Git), bạn cần đặt URL (hiện tại nó trống):

```
$ git remote set-url ngược dòng https://github.com/projectusername/repo.git
```

Nếu thương nguồn không có ở đó hoặc nếu bạn cũng muốn thêm nhánh của bạn bè/dòng nghiệp (hiện tại họ không tồn tại):

```
$ git từ xa thêm ngược dòng https://github.com/projectusername/repo.git $ git từ xa thêm dave
https://github.com/dave/repo.git
```

## Phần 1.6: Tìm hiểu về lệnh

Để biết thêm thông tin về bất kỳ lệnh git nào - tức là chi tiết về chức năng của lệnh, các tùy chọn có sẵn và tài liệu khác - hãy sử dụng tùy chọn `--help` hoặc lệnh `trợ giúp`.

Ví dụ: để có được tất cả thông tin có sẵn về lệnh `git diff`, hãy sử dụng:

```
git diff --help git
help diff
```

Tương tự, để có được tất cả thông tin có sẵn về lệnh trạng thái, hãy sử dụng:

```
trạng thái git --help
trạng thái trợ giúp git
```

Nếu bạn chỉ muốn được trợ giúp nhanh chóng để chỉ cho bạn ý nghĩa của các dòng lệnh được sử dụng nhiều nhất, hãy sử dụng `-h`:

```
kiểm tra git -h
```

## Phần 1.7: Thiết lập SSH cho Git

Nếu bạn đang sử dụng Windows, hãy mở [Git Bash](#). Nếu bạn đang sử dụng Mac hoặc Linux, hãy mở Terminal.

Trước khi tạo khóa SSH, bạn có thể kiểm tra xem liệu mình có khóa SSH nào hiện có không.

Liệt kê nội dung của thư mục `~/.ssh` của bạn :

```
$ ls -al ~/.ssh #
Liệt kê tất cả các tệp trong thư mục ~/.ssh của bạn
```

Kiểm tra danh sách thư mục để xem bạn đã có khóa SSH công khai chưa. Theo mặc định, tên tệp của khóa chung là một trong các tên sau:

```
id_dsa.pub  
id_ecdsa.pub  
id_ed25519.pub  
id_rsa.pub
```

Nếu bạn thấy cặp khóa chung và khóa riêng hiện có được liệt kê mà bạn muốn sử dụng trên tài khoản Bitbucket, GitHub (hoặc tương tự), bạn có thể sao chép nội dung của tệp id\_\*. pub .

Nếu không, bạn có thể tạo cặp khóa chung và khóa riêng mới bằng lệnh sau:

```
$ ssh-keygen
```

Nhấn phím Enter hoặc Return để chấp nhận vị trí mặc định. Nhập và nhập lại cụm mật khẩu khi được nhắc hoặc để trống.

Đảm bảo khóa SSH của bạn được thêm vào tác nhân ssh. Khởi động tác nhân ssh ở chế độ nền nếu nó chưa chạy:

```
$ eval "$(ssh-agent -s)"
```

Thêm khóa SSH của bạn vào tác nhân ssh. Lưu ý rằng bạn sẽ cần thay thế id\_rsa trong lệnh bằng tên tệp khóa riêng của bạn:

```
$ ssh-add ~/.ssh/id_rsa
```

Nếu bạn muốn thay đổi ngược dòng của kho lưu trữ hiện có từ HTTPS sang SSH, bạn có thể chạy lệnh sau:

```
$ git nguồn gốc set-url từ xa ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

Để sao chép kho lưu trữ mới qua SSH, bạn có thể chạy lệnh sau:

```
$ git clone ssh://git@bitbucket.server.com:7999/projects/your_project.git
```

## Phần 1.8: Cài đặt Git

Hãy bắt đầu sử dụng một số Git. Điều đầu tiên trước tiên là bạn phải cài đặt nó. Bạn có thể lấy nó bằng nhiều cách; hai cách chính là cài đặt nó từ nguồn hoặc cài đặt gói hiện có cho nền tảng của bạn.

Cài đặt từ nguồn

Nếu có thể, nói chung việc cài đặt Git từ nguồn sẽ rất hữu ích vì bạn sẽ nhận được phiên bản mới nhất. Mỗi phiên bản Git có xu hướng bao gồm các cải tiến hữu ích về giao diện người dùng, do đó, tải phiên bản mới nhất thường là con đường tốt nhất nếu bạn cảm thấy thoải mái khi biên dịch phần mềm từ nguồn. Cũng có trường hợp nhiều bản phân phối Linux chứa các gói rất cũ; vì vậy trừ khi bạn đang sử dụng bản phân phối rất cập nhật hoặc đang sử dụng cổng lùi, cài đặt từ nguồn có thể là lựa chọn tốt nhất.

Để cài đặt Git, bạn cần có các thư viện sau mà Git phụ thuộc vào: Curl, zlib, openssl, expat và libiconv.

Ví dụ: nếu bạn đang sử dụng hệ thống có yum (chẳng hạn như Fedora) hoặc apt-get (chẳng hạn như hệ thống dựa trên Debian), bạn có thể sử dụng một trong các lệnh sau để cài đặt tất cả các phần phụ thuộc:

```
$ yum cài đặt Curl-devel expat-devel gettext-devel \
```

```
openssl-devel zlib-devel
```

```
$ apt-get cài đặt libcurl4-gnutls-dev libexpat1-dev gettext \
libbz-dev libssl-dev
```

Khi bạn có tất cả các phụ thuộc cần thiết, bạn có thể tiếp tục và lấy ảnh chụp nhanh mới nhất từ trang web Git:

<http://git-scm.com/download> Sau đó biên dịch và cài đặt:

```
$ tar -zxf git-1.7.2.2.tar.gz $ cd
git-1.7.2.2 $ make
prefix=/usr/local all $ sudo make
prefix=/usr/local install
```

Sau khi hoàn tất, bạn cũng có thể nhận Git thông qua chính Git để cập nhật:

```
$ git bản sao git://git.kernel.org/pub/scm/git/git.git
```

Cài đặt trên Linux

Nếu muốn cài đặt Git trên Linux thông qua trình cài đặt nhị phân, bạn thường có thể thực hiện việc này thông qua công cụ quản lý gói cơ bản đi kèm với bản phân phối của mình. Nếu bạn đang dùng Fedora, bạn có thể sử dụng yum:

```
$ ngnon cài đặt git
```

Hoặc nếu bạn đang sử dụng bản phân phối dựa trên Debian như Ubuntu, hãy thử apt-get:

```
$ apt-get cài đặt git
```

Cài đặt trên Mac

Có ba cách dễ dàng để cài đặt Git trên máy Mac. Cách dễ nhất là sử dụng trình cài đặt Git đồ họa mà bạn có thể tải xuống từ trang SourceForge.

<http://sourceforge.net/projects/git-osx-installer/>

Hình 1-7. Trình cài đặt Git OS X. Cách chính khác là cài đặt Git qua MacPorts (<http://www.macports.org>). Nếu bạn đã cài đặt MacPorts, hãy cài đặt Git qua

```
$ sudo cài đặt cổng git +svn +doc +bash_completion +gitweb
```

Bạn không cần phải thêm tất cả các tính năng bổ sung, nhưng có thể bạn sẽ muốn bao gồm +svn trong trường hợp bạn sử dụng Git với các kho Subversion (xem Chương 8).

Homebrew (<http://brew.sh/>) là một cách khác để cài đặt Git. Nếu bạn đã cài đặt Homebrew, hãy cài đặt Git qua

```
$ brew cài đặt git
```

Cài đặt trên Windows

Cài đặt Git trên Windows rất dễ dàng. Dự án msysGit có một trong những quy trình cài đặt dễ dàng hơn. Chỉ cần tải xuống tệp exe của trình cài đặt từ trang GitHub và chạy nó:

<http://msysgit.github.io>

Sau khi được cài đặt, bạn có cả phiên bản dòng lệnh (bao gồm ứng dụng khách SSH sẽ hữu ích sau này) và GUI tiêu chuẩn.

Lưu ý khi sử dụng Windows: bạn nên sử dụng Git với shell msysGit được cung cấp (kiểu Unix), nó cho phép sử dụng các dòng lệnh phức tạp được đưa ra trong cuốn sách này. Vì lý do nào đó, nếu bạn cần sử dụng bảng điều khiển dòng lệnh / shell gốc của Windows, bạn phải sử dụng dấu ngoặc kép thay vì dấu ngoặc đơn (đối với các tham số có dấu cách trong đó) và bạn phải trích dẫn các tham số kết thúc bằng dấu thập phân (^ ) nếu chúng ở cuối dòng, vì đó là biểu tượng tiếp tục trong Windows.

## Chương 2: Duyệt lịch sử

Tham số	Giải thích
-q, --quiet	Yêu tĩnh, ngăn chặn đầu ra khác biệt
--nguồn	Hiển thị nguồn cam kết
--use-mailmap --	Sử dụng tệp bản đồ thư (thay đổi thông tin người dùng để xác nhận người dùng)
trang trí[=...]	Tùy chọn trang trí
--L <n,m:file>	Hiển thị nhật ký cho phạm vi dòng cụ thể trong một tệp, đếm từ 1. Bắt đầu từ dòng n, đến dòng m. Cũng cho thấy sự khác biệt.
--show-signature -i,	Hiển thị chữ ký của các cam kết đã ký
--regexp-ignore-case	Khớp các mẫu giới hạn biểu thức chính quy mà không phân biệt chữ hoa chữ thường

### Phần 2.1: Nhật ký Git "thông thường"

#### nhật ký git

sẽ hiển thị tất cả các cam kết của bạn với tác giả và hàm băm. Điều này sẽ được hiển thị trên nhiều dòng cho mỗi lần xác nhận. (Nếu bạn muốn hiển thị một dòng cho mỗi lần xác nhận, hãy xem onelineing). Sử dụng phím q để thoát nhật ký.

Theo mặc định, không có đối số, git log liệt kê các cam kết được thực hiện trong kho lưu trữ đó theo thứ tự thời gian đảo ngược - nghĩa là các cam kết gần đây nhất hiển thị đầu tiên. Như bạn có thể thấy, lệnh này liệt kê từng cam kết với tổng kiểm tra SHA-1, tên và email của tác giả, ngày viết và thông báo cam kết. -

nguồn

Ví dụ (từ [Trại mã miễn phí kho](#)):

cam kết 87ef97f59e2a2f4dc425982f76f14a57d0900bcf

Hợp nhất: e50ff0d eb8b729

Tác giả: Brian

Ngày: Thứ năm 24 tháng 3 15:52:07 2016 -0700

Hợp nhất yêu cầu kéo # 7724 từ BKinahan/fix/where-art-thou

Sửa lỗi đánh máy 'nó' trong mô tả Where Art Thou

cam kết eb8b7298d516ea20a4aadb9797c7b6fd5af27ea5

Tác giả: BKinahan

Ngày: Thứ năm 24 tháng 3 21:11:36 2016 +0000

Sửa lỗi đánh máy 'nó' trong mô tả Where Art Thou

cam kết e50ff0d249705f41f55cd435f317dcfd02590ee7

Hợp nhất: 6b01875 2652d04

Tác giả: Mrugesh Mohapatra

Ngày: Thứ năm 24 tháng 3 14:26:04 2016 +0530

Hợp nhất yêu cầu kéo #7718 từ deathythe47/fix/unnecessary-commas

Xóa dấu phẩy không cần thiết khỏi CONTRIBUTING.md

Nếu bạn muốn giới hạn lệnh của mình ở n lần ghi nhật ký cuối cùng, bạn chỉ cần truyền một tham số. Ví dụ: nếu bạn muốn liệt kê 2 nhật ký cam kết cuối cùng

nhật ký git -2

## Mục 2.2: Khúc gỗ đẹp hơn

Để xem nhật ký có cấu trúc giống biểu đồ đẹp hơn, hãy sử dụng:

```
git log --trang trí --oneline --graph
```

đầu ra mẫu:

```
* e0c1cea (HEAD -> maint, tag: v2.9.3, Origin/maint) Git 2.9.3 * 9b601ea Hợp nhất  
nhánh 'jk/difftool-in-subdir' vào maint \| * 32b8c58 Difftool: sử dụng các  
hàm Git::* thay vì chuyển trạng thái | * 98f917e Difftool: tránh $GIT_DIR và $GIT_WORK_TREE | *  
9ec26e7 Difftool: sửa lỗi xử lý đối số trong các thư mục con * | f4fd627  
Hợp nhất nhánh 'jk/reset-ident-time-per-commit' vào maint  
...
```

Vì đây là một lệnh khá lớn nên bạn có thể gán bí danh:

```
cấu hình git --global bí danh.lol "log --trang trí --oneline --graph"
```

Để sử dụng phiên bản bí danh:

```
# lịch sử của chi nhánh hiện tại :  
git lol  
  
# lịch sử kết hợp của nhánh đang hoạt động (HEAD), nhánh phát triển và nhánh Origin/master : git lol  
HEAD develop Origin/master  
  
# lịch sử tổng hợp của mọi thư trong repo của bạn : git lol  
--all
```

## Phần 2.3: Tô màu nhật ký

```
git log --graph --pretty=format:'%C(red)%h%Creset %C(golden)%d%Creset %s %C(green)(%cr)  
%C(màu vàng)<%an>%Creset'
```

Tùy chọn định dạng cho phép bạn chỉ định định dạng đầu ra nhật ký của riêng mình:

Tham số	Chi tiết
Tùy chọn %C(color_name)	tô màu đầu ra sau khi nó viết tắt hàm băm
%h hoặc %H	xác nhận (sử dụng %H cho hàm băm hoàn chỉnh)
%Creset	đặt lại màu về màu thiết bị đầu cuối mặc định
%d	tên giới thiệu
%s	chủ đề [thông báo cam kết] ngày
%cr	của người gửi, liên quan đến ngày hiện tại
%MỘT	tên tác giả

## Phần 2.4: Nhật ký trực tuyến

nhật ký git --oneline

sẽ hiển thị tất cả các cam kết của bạn chỉ với phần đầu tiên của hàm băm và thông báo cam kết. Mỗi cam kết sẽ nằm trên một dòng duy nhất, như cờ oneline gợi ý.

Tùy chọn một dòng in từng cam kết trên một dòng, điều này rất hữu ích nếu bạn đang xem nhiều cam kết. - [nguồn](#)

Ví dụ (từ [Trại mã miễn phí](#) kho lưu trữ, với cùng một phần mã từ ví dụ khác):

```
87ef97f Hợp nhất yêu cầu kéo #7724 từ BKinahan/fix/where-art-thou eb8b729 Sửa lỗi đánh máy
'của nó' trong mô tả Where Art Thou e50ff0d Hợp nhất yêu cầu kéo #7718 từ
deathythe47/fix/unnecessary-comma 2652d04 Xóa dấu phẩy không cần thiết khỏi CONTRIBUTING.md 6b01875
Yêu cầu kéo hợp nhất #7667 từ zerkms/patch-1 766f088 Thuật ngữ toán tử gán
cố định d1e2468 Yêu cầu kéo hợp nhất #7690 từ BKinahan/fix/unsubscribe-
crash bed9de2 Hợp nhất yêu cầu kéo #7657 từ Rafase282/fix/
```

Nếu bạn muốn giới hạn lệnh của mình ở n lần ghi nhật ký cuối cùng, bạn chỉ cần truyền một tham số. Ví dụ: nếu bạn muốn liệt kê 2 nhật ký cam kết cuối cùng

```
nhật ký git -2 --oneline
```

## Phần 2.5: Tìm kiếm nhật ký

```
git log -S "#define MÃU"
```

Tìm kiếm thêm hoặc xóa chuỗi cụ thể hoặc chuỗi khớp được cung cấp REGEXP. Trong trường hợp này, chúng tôi đang tìm cách thêm/xóa chuỗi #define SAMPLES. Ví dụ:

```
+#xác định MÃU 100000
```

hoặc

```
-#xác định MÃU 100000
```

```
git log -G "#define MÃU"
```

Tìm kiếm các thay đổi trong các dòng chứa chuỗi cụ thể hoặc chuỗi khớp được cung cấp REGEXP. Ví dụ:

```
-#xác định MÃU 100000 +#xác định MÃU
100000000
```

## Phần 2.6: Liệt kê tất cả các bài đóng góp được nhóm theo tên tác giả

```
git shortlog tóm tắt git log và các nhóm theo tác giả
```

Nếu không có tham số nào được đưa ra, danh sách tất cả các cam kết được thực hiện cho mỗi người cam kết sẽ được hiển thị theo thứ tự thời gian.

```
$ git shortlog
```

Người xác nhận 1 (<number\_of\_commits>):

Tin nhắn cam kết 1

Thông báo cam kết 2

...  
Người xác nhận 2 (<number\_of\_commits>):

Tin nhắn cam kết 1

Thông báo cam kết 2

Để chỉ xem số lần xác nhận và loại bỏ mô tả cam kết, hãy chuyển vào tùy chọn tóm tắt:

-s

--bản tóm tắt

```
$ git shortlog -s
<number_of_commits> Người gửi 1
<number_of_commits> Người gửi 2
```

Để sắp xếp đầu ra theo số lần xác nhận thay vì theo thứ tự bảng chữ cái theo tên người gửi, hãy chuyển vào tùy chọn được đánh số:

-N

--được đánh số

Để thêm email của người cam kết, hãy thêm tùy chọn email:

-e

--e-mail

Tùy chọn định dạng tùy chỉnh cũng có thể được cung cấp nếu bạn muốn hiển thị thông tin khác ngoài chủ đề cam kết:

--định dạng

Đây có thể là bất kỳ chuỗi nào được chấp nhận bởi tùy chọn --format của `git log`.

Xem Nhật ký tô màu ở trên để biết thêm thông tin về điều này.

## Phần 2.7: Tìm kiếm chuỗi cam kết trong git log

Tìm kiếm nhật ký git bằng cách sử dụng một số chuỗi trong nhật ký:

```
nhật ký git [tùy chọn] --grep "search_string"
```

Ví dụ:

```
git log --all --grep "tệp đã xóa"
```

Sẽ tìm kiếm chuỗi `tệp đã xóa` trong tất cả nhật ký ở tất cả các nhánh.

Bắt đầu từ git 2.4+, tìm kiếm có thể được đảo ngược bằng tùy chọn `--invert-grep`.

Ví dụ:

```
git log --grep="thêm tập tin" --invert-grep
```

Sẽ hiển thị tất cả các cam kết không chứa tệp bổ sung.

## Phần 2.8: Ghi nhật ký một loạt dòng trong một tệp

```
$ git log -L 1,20:index.html commit
6a57fde739de66293231f6204cbd8b2feca3a869 Tác giả: John Doe
<john@doe.com> Ngày: Thứ Ba 22/03
16:33:42 2016 -0500
```

tin nhắn cam kết

```
diff --git a/index.html b/index.html --- a/
index.html +++ b/
index.html @@ -1,17
+1,20 @@
<!DOCTYPE HTML>
<html>
- <head>
- <bộ ký tự meta="utf-8">
+
+<head>
+ <meta charset="utf-8" > <meta
http-equiv="X-UA-Compatible" content="IE=edge" > <meta name="viewport"
content="width=device-width , quy mô ban đầu=1">
```

## Phần 2.9: Lọc nhật ký

nhật ký git --sau '3 ngày trước'

Ngày cụ thể cũng có tác dụng:

nhật ký git --sau 2016-05-01

Giống như các lệnh và cờ khác chấp nhận tham số ngày, định dạng ngày được phép cũng được hỗ trợ bởi ngày GNU (rất linh hoạt).

Bí danh của `--after` là `--since`.

Còn cũng tồn tại cho điều ngược lại: `--trước` và `--until`.

Bạn cũng có thể lọc nhật ký theo tác giả. ví dụ

nhật ký git --author=tác giả

## Phần 2.10: Đăng nhập với các thay đổi nội tuyến

Để xem nhật ký có các thay đổi nội tuyến, hãy sử dụng tùy chọn `-p` hoặc `--patch`.

nhật ký git --patch

Ví dụ (từ [Nhà khoa học Trello](#) kho)

```
bỏ qua 8ea1452aca481a837d9504f1b2c77ad013367d25
```

```
Tác giả: Raymond Chou <info@raychou.io> Ngày:
Thứ Tư ngày 2 tháng 3 10:35:25 2016 -0800
```

sửa lỗi liên kết README

```

diff --git a/README.md b/README.md chỉ số
1120a00..9bef0ce 100644 --- a/README.md
+++ b/README.md @@
-134,7 +134,7 @@
điều khiển chức năng đã bị ném, nhưng *sau khi* kiểm tra các chức năng khác và sẵn sàng ghi nhật ký. Tiêu chí để so khớp
lỗi dựa
trên hàm tạo và
tin nhắn.

-Bạn có thể tìm thấy ví dụ đầy đủ này tại \[examples/errors.js\]\(examples/error.js\).
+Bạn có thể tìm thấy ví dụ đầy đủ này tại \[examples/errors.js\]\(examples/errors.js\).

## Hành vi không đồng bộ

cam kết d3178a22716cc35b6a2bdd679a7ec24bc8c63ffa
:

```

## Mục 2.11: Nhật ký hiển thị các tập tin đã cam kết

[nhật ký git -stat](#)

Ví dụ:

```

cam kết 4ded994d7fc501451fa6e233361887a2365b91d1 Tác giả:
Manassés Souza <manasses.inatel@gmail.com> Ngày: Thứ Hai ngày 6
tháng 6 21:32:30 2016 -0300

```

```

MercadoLibre phụ thuộc java-sdk

mltracking-poc/.gitignore | 1 +
mltracking-poc/pom.xml | 14 ++++++----- 2 tập tin đã thay
đổi, 13 lần chèn(+), 2 lần xóa(-)

```

```
cam kết 506fff56190f75bc051248770fb0bcd976e3f9a5
```

```
Tác giả: Manassés Souza <manasses.inatel@gmail.com>
Ngày: Thứ bảy ngày 4 tháng 6 12:35:16 2016 -0300
```

```
[manasses] được tạo bởi SpringBoot khởi tạo
```

```

.gitignore | 42
+++++
mltracking-poc/mvnw | 233
+++++
mltracking-poc/mvnw.cmd | 145
+++++
mltracking-poc/pom.xml | 74
+++++
mltracking-poc/src/main/java;br\com\mls\mltracking\MLtrackingPocApplication.java mltracking-poc/src/main/
resources/application.properties mltracking-poc/src/test/java;br\com\mls\
mltracking\MLtrackingPocApplicationTests.java | 18 ++++ 7 tệp đã thay đổi, 524 lần chèn (+)
| 12 ++++ | 0

```

## Phân 2.12: Hiển thị nội dung của một lần xác nhận

Sử dụng [chương trình git](#) chúng ta có thể xem một cam kết duy nhất

[git hiển thị 48c83b3](#)

```
git hi&acute;n thi 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Ví dụ

```
cam kết 48c83b3690dfc7b0e622fd220f8f37c26a77c934
```

Tác giả: Matt Clark <mrc Clark32493@gmail.com>

Ngày: Thứ Tư ngày 4 tháng 5 18:26:40 2016 -0400

Thông báo cam kết sẽ được hiển thị ở đây.

```
khác --git a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
chỉ số 0b57e4a..fa8e6a5 100755
--- a/src/main/java/org/jdm/api/jenkins/BuildStatus.java
+++ b/src/main/java/org/jdm/api/jenkins/BuildStatus.java
@@ -50,7 +50,7 @@ enum công khai BuildStatus {

            colorMap.put(BuildStatus.UNSTABLE, Color.decode( "#FFFF55" ));
-
+            colorMap.put(BuildStatus.SUCCESS, Color.decode( "#55FF55" ));
            colorMap.put(BuildStatus.SUCCESS, Color.decode( "#33CC33" ));
            colorMap.put(BuildStatus.BUILDING, Color.decode( "#5555FF" ));
```

## Phần 2.13: Nhật ký Git giữa hai nhánh

```
git log master..foo sẽ hiển thị các cam kết trên foo chứ không phải trên master. Hữu ích để xem những gì cam kết bạn đã thêm kể từ khi phân nhánh!
```

## Mục 2.14: Một dòng hiển thị tên và thời gian của người thực hiện kể từ khi làm

```
cây = log --oneline --trang trí --source --pretty=format:'"%Cblue %h %Cgreen %ar %Cblue %an
%C(màu vàng) %d %Creset %s"' --all --graph
```

ví dụ

* 40554ac 3 tháng trước Alexander Zolotov gmandnepr/external_plugins	Hợp nhất yêu cầu kéo #95 từ
\   * e509f61 3 tháng trước Ievgen Degtiarenko   * 46d4cb6 3 tháng trước Ievgen Degtiarenko   * 6253da4 3 tháng trước Ievgen Degtiarenko   * 9fdb4e7 3 tháng trước Ievgen Degtiarenko quan trọng đối với intellij	Tài liệu về tài sản mới Chạy ý tưởng với các plugin bên ngoài Giải quyết các lớp plugin bên ngoài Giữ tên tạo tác ban đầu vì điều này có thể
* 22e82e4 3 tháng trước Ievgen Degtiarenko   / * bc3d2cb 3 tháng trước Alexander Zolotov	Khai báo plugin bên ngoài trong phần intellij
	Bỏ qua DTD trong plugin.xml

## Chương 3: Làm việc với điều khiển từ xa

### Phần 3.1: Xóa một nhánh từ xa

Để xóa một nhánh từ xa trong Git:

```
git push [tên từ xa] --delete [tên nhánh]
```

hoặc

```
git push [tên từ xa] : [tên nhánh]
```

### Phần 3.2: Thay đổi URL từ xa Git

Kiểm tra điều khiển từ xa hiện có

```
git remote -v #  
Origin https://github.com/username/repo.git (tìm nạp) # Origin https://  
github.com/username/repo.git (đầy)
```

Thay đổi URL kho lưu trữ

```
git remote set-url Origin https://github.com/username/repo2.git # Thay đổi URL của  
điều khiển từ xa 'origin'
```

Xác minh URL từ xa mới

```
git remote -v #  
Origin https://github.com/username/repo2.git (tìm nạp) # Origin https://  
github.com/username/repo2.git (push)
```

### Phần 3.3: Liệt kê các điều khiển từ xa hiện có

Liệt kê tất cả các điều khiển từ xa hiện có được liên kết với kho lưu trữ này:

```
git từ xa
```

Liệt kê chi tiết tất cả các điều khiển từ xa hiện có được liên kết với kho lưu trữ này, bao gồm các URL tìm nạp và đầy :

```
git từ xa --verbose
```

hoặc đơn giản

```
git từ xa -v
```

### Phần 3.4: Xóa bản sao cục bộ của các nhánh từ xa đã xóa

Nếu một nhánh từ xa đã bị xóa, kho lưu trữ cục bộ của bạn phải được yêu cầu cắt bớt tham chiếu đến nó.

Để tìa các nhánh đã xóa từ một điều khiển từ xa cụ thể:

```
git tm ngp [tên từ xa] --Prune
```

Để t~~m~~ia các nhánh đã xóa khỏi tất cả các điều khiển từ xa:

```
git tm ngp --all --Prune
```

## Ph~~ần~~ 3.5: Cập nhật từ Kho lưu trữ ngược dòng

Giả sử bạn thiết lập ngược dòng (như trong phần "thiết lập kho lưu trữ ngược dòng")

```
git lay tên từ xa git merge  
tên từ xa/tên nhánh
```

Lệnh kéo kết hợp t~~m~~n~~g~~p và hợp nhất.

```
kéo git
```

Lệnh kéo bằng cờ `--rebase` kết hợp t~~m~~n~~g~~p và rebase thay vì hợp nhất.

```
git pull --rebase tên chi nhánh tên từ xa
```

## Ph~~ần~~ 3.6: ls-remote

`git ls-remote` là một lệnh duy nhất cho phép bạn truy vấn một kho lưu trữ từ xa mà không cần phải sao chép/tìm nạp nó trước.

Nó sẽ liệt kê các ref/heads và ref/tags của repo từ xa đã nói.

Đôi khi bạn sẽ thấy `refs/tags/v0.1.6` và `refs/tags/v0.1.6^{}: the ^{}` để liệt kê thẻ chú thích bị hủy tham chiếu (tức là cam kết mà thẻ đang trỏ đến)

Kể từ git 2.8 (tháng 3 năm 2016), bạn có thể tránh mục nhập kép đó cho một thẻ và liệt kê trực tiếp các thẻ bị hủy đăng ký đó bằng:

```
git ls-remote --ref
```

Nó cũng có thể giúp giải quyết url thực tế được sử dụng bởi một kho lưu trữ từ xa khi bạn có cài đặt cấu hình "`url.<base>.insteadOf`" .

Nếu `git remote --get-url <aremotename>` trả về `https://server.com/user/repo`, và bạn đã đặt `git config url.ssh://git@server.com:.insteadOf https://server.com/`:

```
git ls-remote --get-url <aremotename> ssh://  
git@server.com:user/repo
```

## Ph~~ần~~ 3.7: Thêm kho lưu trữ từ xa mới

```
git remote thêm git-repository-url ngược dòng
```

Thêm kho lưu trữ git từ xa được đại diện bởi `git-repository-url` dưới dạng điều khiển từ xa mới có tên ngược dòng vào kho lưu trữ git

## Ph~~ần~~ 3.8: Thiết lập ngược dòng trên một nhánh mới

Bạn có thể tạo một nhánh mới và chuyển sang nhánh đó bằng cách sử dụng

```
kiểm tra git -b AP-57
```

Sau khi bạn sử dụng gitcheck để tạo một nhánh mới, bạn sẽ cần đặt nguồn gốc ngược dòng đó để chuyển sang sử dụng

```
git push --set-upstream Origin AP-57
```

Sau đó, bạn có thể sử dụng git push khi đang ở nhánh đó.

## Phần 3.9: Bắt đầu

Cú pháp đẩy tới một nhánh ở xa

```
git đẩy <remote_name> <branch_name>
```

Ví dụ

```
git đẩy nguồn gốc chính
```

## Phần 3.10: Đổi tên điều khiển từ xa

Để đổi tên điều khiển từ xa, sử dụng lệnh đổi tên `từ xa git`

Lệnh đổi tên `từ xa git` có hai đối số:

- Một tên từ xa hiện có, ví dụ: Origin
- Tên mới cho điều khiển từ xa, ví dụ: Destination

Nhận tên từ xa hiện có

```
git từ xa #
nguồn gốc
```

Kiểm tra điều khiển từ xa hiện có bằng URL

```
git remote -v #
Origin https://github.com/username/repo.git (tìm nạp) # Origin https://
github.com/username/repo.git (đẩy)
```

Đổi tên từ xa

```
git từ xa đổi tên đích xuất xứ
# Thay đổi tên từ xa từ 'nguồn gốc' thành 'đích'
```

Xác minh tên mới

```
git remote -v #
dịch https://github.com/username/repo.git (tìm nạp) # đích https://github.com/
username/repo.git (đẩy)
```

== Có thể có lỗi ==

1. Không thể đổi tên phần cấu hình 'remote.[old name]' thành 'remote.[new name]'

Lỗi này có nghĩa là điều khiển từ xa bạn đã thử, tên cũ (nguồn gốc) không tồn tại.

2. Remote [tên mới] đã tồn tại.

Thông báo lỗi là tự giải thích.

## Phần 3.11: Hiển thị thông tin về Điều khiển từ xa cụ thể

Xuất ra một số thông tin về một điều khiển từ xa đã biết: nguồn gốc

nguồn gốc chương trình từ xa git

Chỉ in URL của điều khiển từ xa:

cấu hình git --get remote.origin.url

Với 2.7+, điều đó cũng có thể thực hiện được, điều này được cho là tốt hơn so với phiên bản trên sử dụng lệnh config .

nguồn gốc get-url từ xa git

## Phần 3.12: Đặt URL cho Điều khiển từ xa cụ thể

Bạn có thể thay đổi url của điều khiển từ xa hiện có bằng lệnh

git remote set-url url tên từ xa

## Phần 3.13: Nhận URL cho điều khiển từ xa cụ thể

Bạn có thể lấy url cho điều khiển từ xa hiện có bằng cách sử dụng lệnh

git từ xa get-url <tên>

Theo mặc định, đây sẽ là

nguồn gốc get-url từ xa git

## Phần 3.14: Thay đổi kho lưu trữ từ xa

Để thay đổi URL của kho lưu trữ mà bạn muốn điều khiển từ xa trả tới, bạn có thể sử dụng tùy chọn set-url , như sau:

git remote set-url <remote\_name> <remote\_repository\_url>

Ví dụ:

git remote set-url heroku https://git.heroku.com/fictional-remote-repository.git

# Chương 4: Dàn dựng

## Phần 4.1: Sắp xếp tất cả các thay đổi đối với tệp

```
git thêm -A
```

Phiên bản 2.0

```
git thêm .
```

Trong phiên bản 2.x, `git add .` sẽ thực hiện tất cả các thay đổi đối với các tệp trong thư mục hiện tại và tất cả các thư mục con của nó. Tuy nhiên, trong 1.x, nó sẽ chỉ xử lý các tệp mới và đã sửa đổi, không xóa các tệp.

Sử dụng `git add -A` hoặc lệnh tương đương `git add --all` để xử lý tất cả các thay đổi đối với tệp trong bất kỳ phiên bản git nào.

## Phần 4.2: Hủy phân vùng một tệp có chứa các thay đổi

```
đặt lại git <filePath>
```

## Phần 4.3: Thêm thay đổi theo khôi

Bạn có thể xem "khôi" công việc nào sẽ được sắp xếp để cam kết bằng cờ vá lỗi:

```
git thêm -p
```

hoặc

```
git thêm --patch
```

Thao tác này mở ra lối nhắc tương tác cho phép bạn xem xét các điểm khác biệt và cho phép bạn quyết định xem bạn có muốn đưa chúng vào hay không.

Giai đoạn này [y,n,q,a,d/,s,e,?]?

- y sắp xếp phần này cho lần chuyển tiếp theo
- n không chuyển đoạn phần này cho lần chuyển tiếp tiếp
- thep q thoát; không phân đoạn phần này hoặc bất kỳ phần nào còn lại trong
- phần còn lại một giai đoạn phần này và tất cả các phần sau
- trong tệp d không phân đoạn phần này hoặc bất kỳ phần nào sau đó trong tệp
- g chọn một phần để đi tới /
- tìm kiếm phần phù hợp biểu thức chính quy đã cho j để lại
- phần này chưa quyết định, xem phần chưa quyết định tiếp theo J để phần
- này chưa quyết định, xem phần tiếp theo k để phần này
- chưa quyết định, xem phần trước chưa quyết định K để phần này chưa quyết
- định, xem phần trước chia phần hiện tại thành các phần nhỏ
- hơn e chỉnh sửa thủ công phần hiện tại? in hunk giúp đỡ
- 
- 

Điều này giúp bạn dễ dàng nắm bắt được những thay đổi mà bạn không muốn thực hiện.

Bạn cũng có thể mở cái này thông qua `git add --interactive` và chọn p.

## Phân 4.4: Thêm tương tác

`git add -i` (hoặc `--interactive`) sẽ cung cấp cho bạn giao diện tương tác nơi bạn có thể chỉnh sửa chỉ mục, để chuẩn bị những gì bạn muốn có trong lần cam kết tiếp theo. Bạn có thể thêm và xóa các thay đổi đối với toàn bộ tệp, thêm các tệp không bị theo dõi và xóa các tệp khỏi bị theo dõi mà còn chọn phần phụ của các thay đổi để đưa vào chỉ mục, bằng cách chọn các phần của những thay đổi cần được thêm vào, chia tách các phần đó hoặc thậm chí chỉnh sửa phần khác biệt. Nhiều công cụ cam kết đồ họa cho Git (ví dụ: `git gui`) bao gồm tính năng đó; điều này có thể dễ sử dụng hơn phiên bản dòng lệnh.

Nó rất hữu ích (1) nếu bạn có những thay đổi phức tạp trong thư mục làm việc mà bạn muốn đưa vào các cam kết riêng biệt, và không phải tất cả trong một cam kết duy nhất (2) nếu bạn đang ở giữa một cuộc nổi loạn tương tác và muốn chia quá lớn làm.

```
$ git thêm -i
      dàn dựng      con đường không có giải đoạn
1:     không thay      +4/-4 chỉ mục.js
2:     đổi +1/-0      không có gì gói.json

*** Lệnh ***
1: trạng thái          2: cập nhật          3: hoàn nguyên          4: thêm không bị theo dõi
5: vá                6: khác biệt        7: bỏ cuộc          8: giúp đỡ

Làm sao bây giờ?
```

Nửa trên của kết quả này hiển thị trạng thái hiện tại của chỉ mục được chia thành các cột theo giai đoạn và không theo giai đoạn:

- index.js đã thêm 4 dòng và xóa 4 dòng. Nó hiện không được tổ chức, như các báo cáo trạng thái hiện tại "không thay đổi." Khi tệp này được dàn dựng, bit +4/-4 sẽ được chuyển sang cột được dàn dựng và cột không được phân loại sẽ đọc "không có gì."
- pack.json đã được thêm một dòng và đã được dàn dựng. Không có thay đổi nào nữa kể từ khi nó được dàn dựng như được biểu thị bằng dòng "không có gì" bên dưới cột không được dàn dựng.

Nửa dưới cho thấy những gì bạn có thể làm. Nhập một số (1-8) hoặc một chữ cái (s, u, r, a, p, d, q, h).

trạng thái hiển thị đầu ra giống với phần trên cùng của đầu ra ở trên.

cập nhật cho phép bạn thực hiện các thay đổi tiếp theo đối với các cam kết theo giai đoạn bằng cú pháp bổ sung.

hoàn nguyên sẽ hoàn nguyên thông tin cam kết theo giai đoạn trả lại HEAD.

thêm không bị theo dõi cho phép bạn thêm các đường dẫn tệp trước đó không bị theo dõi bởi kiểm soát phiên bản.

bản vá cho phép chọn một đường dẫn từ đầu ra tương tự như trạng thái để phân tích thêm.

`diff` hiển thị những gì sẽ được cam kết.

quit thoát khỏi lệnh.

trợ giúp trình bày trợ giúp thêm về cách sử dụng lệnh này.

## Phân 4.5: Hiển thị các thay đổi theo giai đoạn

Để hiển thị các khối được sắp xếp cho cam kết:

```
git khác --được lưu trong bộ nhớ đệm
```

## Phần 4.6: Sắp xếp một tệp duy nhất

Để sắp xếp một tập tin để cảm kết, hãy chạy

```
git add <tên tệp>
```

## Phần 4.7: Giai đoạn xóa file

```
tên tệp git rm
```

Để xóa tệp khỏi git mà không xóa tệp khỏi đĩa, hãy sử dụng cờ **--cached**

```
git rm --tên tệp được lưu trong bộ nhớ cache
```

## Chương 5: Bỏ qua tập tin và thư mục

Chủ đề này minh họa cách tránh thêm các tệp không mong muốn (hoặc thay đổi tệp) trong kho lưu trữ Git. Có một số cách (.gitignore toàn cầu hoặc cục bộ, .git/exclude, `git update-index --assume-unchanged` và `git update-index -- Skip-tree`), nhưng hãy nhớ rằng Git đang quản lý nội dung, có nghĩa là: bỏ qua thực sự bỏ qua nội dung thư mục (tức là các tập tin). Theo mặc định, một thư mục trống sẽ bị bỏ qua vì dù sao thì nó cũng không thể được thêm vào.

### Phần 5.1: Bỏ qua tệp và thư mục bằng tệp .gitignore

Bạn có thể khiến Git bỏ qua một số tệp và thư mục nhất định – nghĩa là loại trừ chúng khỏi bị Git theo dõi – bằng cách tạo một hoặc nhiều .gitignore các tập tin trong kho lưu trữ của bạn.

Trong các dự án phần mềm, .gitignore thường chứa danh sách các tệp và/hoặc thư mục được tạo trong quá trình xây dựng hoặc trong thời gian chạy. Các mục trong tệp .gitignore có thể bao gồm tên hoặc đường dẫn đến:

1. tài nguyên tạm thời, ví dụ: bộ nhớ đệm, tệp nhật ký, mã được biên dịch, v.v.
2. tệp cấu hình cục bộ không được chia sẻ với các nhà phát triển khác
3. tệp chứa thông tin bí mật, chẳng hạn như mật khẩu đăng nhập, khóa và thông tin xác thực

Khi được tạo trong thư mục cấp cao nhất, các quy tắc sẽ áp dụng đệ quy cho tất cả các tệp và thư mục con trong toàn bộ kho lưu trữ. Khi được tạo trong thư mục con, các quy tắc sẽ áp dụng cho thư mục cụ thể đó và các thư mục con của nó.

Khi một tập tin hoặc thư mục bị bỏ qua, nó sẽ không:

1. được theo dõi bởi Git
2. được báo cáo bằng các lệnh như `git status` hoặc `git diff`
3. được tổ chức bằng các lệnh như `git add -A`

Trong trường hợp bất thường là bạn cần bỏ qua các tệp được theo dõi, cần đặc biệt cẩn thận. Xem: Bỏ qua các tệp đã được cam kết vào kho Git.

Ví dụ

Dưới đây là một số ví dụ chung về quy tắc trong tệp .gitignore , dựa trên mẫu tệp toàn cầu:

```
# Các dòng bắt đầu bằng `#` là các chú thích.

# Bỏ qua các file có tên 'file.ext'
file.ext

# Bình luận không được nằm trên cùng một dòng với quy tắc!
# Dòng sau bỏ qua các tập tin có tên 'file.ext # not a comment' file.ext # not a
comment

# Bỏ qua các file có đường dẫn đầy đủ.
# Điều này khớp với các tập tin trong thư mục gốc và các thư mục con. # tức là
otherfile.ext sẽ bị bỏ qua ở bất kỳ đâu trên cây. dir/otherdir/file.ext
otherfile.ext

# Bỏ qua các thư mục # Cả
thư mục và nội dung của nó sẽ bị bỏ qua. thùng rác/
gen/
```

```
# Mẫu toàn cầu cũng có thể được sử dụng ở đây để bỏ qua các đường dẫn có ký tự nhất định.  
# Ví dụ: quy tắc bên dưới sẽ khớp với cả build/ và Build/ [bB]uild/
```

```
# Không có dấu gạch chéo ở cuối, quy tắc sẽ khớp với một tệp và/hoặc # một thư mục,  
do đó, quy tắc sau sẽ bỏ qua cả tệp có tên `gen` # và thư mục có tên `gen`, cũng như mọi  
nội dung của thùng thư mục đó
```

thể hệ

```
# Bỏ qua các tệp theo phần mở rộng #  
Tất cả các tệp có phần mở rộng này sẽ bị bỏ qua trong # thư mục này  
và tất cả các thư mục con của nó. *.apk *.class
```

```
# Có thể kết hợp cả hai biểu mẫu để bỏ qua các tệp có # tiện ích mở rộng nhất định  
trong các thư mục nhất định. Các quy tắc sau đây sẽ # dư thừa với các quy tắc chung  
được xác định ở trên. java/*.apk gen/*.class
```

```
# Để chỉ bỏ qua các tệp ở thư mục cấp cao nhất chứ không phải trong # thư mục con của  
nó, hãy thêm tiền tố `/*/*.apk /*.class vào quy tắc
```

```
# Để bỏ qua mọi thư mục có tên DirectoryA # ở bất kỳ mức  
độ nào, hãy sử dụng ** trước DirectoryA  
# Đừng quên /, cuối cùng  
# Nếu không nó sẽ bỏ qua tất cả các file có tên DirectoryA, thay vì các thư mục  
**/Thư mụcA/  
# Điều này sẽ bỏ qua  
# Thư mụcA/  
# Thư mụcB/Thư mụcA/  
# Thư mụcC/Thư mụcB/Thư mụcA/  
# Nó sẽ không bỏ qua file có tên DirectoryA, ở mọi cấp độ
```

```
# Để bỏ qua bất kỳ thư mục nào có tên DirectoryB trong # thư mục  
có tên DirectoryA với bất kỳ số # thư mục nào ở giữa, hãy sử  
dụng ** giữa các thư mục
```

```
Thư mụcA/**/Thư mụcB/  
# Điều này sẽ bỏ qua  
# Thư mụcA/Thư mụcB/  
# Thư mụcA/Thư mụcQ/Thư mụcB/  
# DirectoryA/DirectoryQ/DirectoryW/DirectoryB/
```

```
# Để bỏ qua một tập hợp các tệp, có thể sử dụng các ký tự đại diện, như có thể thấy ở trên.  
# Một '*' duy nhất sẽ bỏ qua mọi thứ trong thư mục của bạn, kể cả tệp .gitignore.  
# Để loại trừ các tệp cụ thể khi sử dụng ký tự đại diện, hãy phủ nhận chúng.  
# Vậy là chúng bị loại khỏi danh sách bỏ qua: !.gitignore
```

```
# Sử dụng dấu gạch chéo ngược làm ký tự thoát để bỏ qua các tệp có hàm băm (#) # (được hỗ trợ  
kể từ 1.6.2.1) \#*#
```

Hầu hết các tệp .gitignore đều là tiêu chuẩn trên nhiều ngôn ngữ khác nhau, vì vậy để bắt đầu, đây là tập hợp [mẫu .gitignore](#) [các tập tin](#) được liệt kê theo ngôn ngữ để sao chép hoặc sao chép/sửa đổi vào dự án của bạn. Ngoài ra, đối với một dự án mới, bạn có thể xem xét việc tự động tạo tệp khởi động bằng [công cụ trực tuyến](#).

### Các dạng .gitignore khác

Các tệp .gitignore được dự định sẽ được cam kết như một phần của kho lưu trữ. Nếu bạn muốn bỏ qua một số tệp nhất định mà không tuân theo quy tắc bỏ qua, đây là một số tùy chọn:

- Chỉnh sửa tệp .git/info/exclude (sử dụng cú pháp tương tự như .gitignore). Các quy tắc sẽ mang tính toàn cầu trong phạm vi kho lưu trữ; Thiết lập
- tệp gitignore toàn cầu sẽ áp dụng các quy tắc bỏ qua cho tất cả các kho lưu trữ cục bộ của bạn:

Hơn nữa, bạn có thể bỏ qua các thay đổi cục bộ đối với các tệp được theo dõi mà không thay đổi cấu hình git chung bằng:

- `git update-index --skip-worktree [<file>...]`: dành cho các sửa đổi cục bộ nhỏ `git update-index`
- `--assume-unchanged [<file>...]`: dành cho các tệp sẵn sàng sản xuất, không thay đổi ở thượng nguồn

Xem [chi tiết về sự khác biệt giữa các cờ sau và chỉ mục cập nhật git tài liệu](#) để có thêm lựa chọn.

### Dọn dẹp các tập tin bị bỏ qua

Bạn có thể sử dụng `git clean -X` để dọn dẹp các tệp bị bỏ qua:

```
git clean -Xn #hiển thị danh sách các tệp bị bỏ qua git clean  
-Xf #remove các tệp được hiển thị trước đó
```

Lưu ý: -X (caps) chỉ xóa các tệp bị bỏ qua. Sử dụng -x (không viết hoa) để xóa các tệp không bị theo dõi.

Xem tài liệu `git clean` để biết thêm chi tiết.

Xem [hướng dẫn sử dụng Git](#) để biết thêm chi tiết.

## Phần 5.2: Kiểm tra xem một tập tin có bị bỏ qua hay không

Cái `git kiểm tra-bỏ qua` lệnh báo cáo về các tập tin bị Git bỏ qua.

Bạn có thể chuyển tên tệp trên dòng lệnh và `git check-ignore` sẽ liệt kê các tên tệp bị bỏ qua. Ví dụ:

```
$ mèo .gitignore  
*.o  
$ git kiểm tra-bỏ qua ví dụ.o Readme.md ví dụ.o
```

Ở đây, chỉ các tệp \*.o được xác định trong .gitignore, vì vậy Readme.md không được liệt kê trong đầu ra của `git check-ignore`.

Nếu bạn muốn xem dòng .gitignore chịu trách nhiệm bỏ qua một tệp, hãy thêm -v vào lệnh `git check-ignore`:

```
$ git check-ignore -v example.o Readme.md .gitignore:1:*.o  
example.o
```

Từ Git 1.7.6 trở đi, bạn cũng có thể sử dụng `git status --ignored` để xem các tệp bị bỏ qua. Bạn có thể tìm thêm thông tin về điều này trong [tài liệu chính thức](#) hoặc trong [Tìm tệp bị .gitignore bỏ qua](#).

## Phần 5.3: Ngoại lệ trong tệp .gitignore

Nếu bạn bỏ qua các tệp bằng cách sử dụng một mẫu nhưng có ngoại lệ, hãy thêm dấu chấm than (!) vào tiền tố ngoại lệ. Ví dụ:

```
*.txt
!important.txt
```

Ví dụ trên hướng dẫn Git bỏ qua tất cả các tệp có phần mở rộng .txt ngoại trừ các tệp có tên quan trọng.txt.

Nếu tệp nằm trong thư mục bị bỏ qua, bạn KHÔNG thể đưa lại tệp đó một cách dễ dàng:

```
thư
mục/ !folder/*.txt
```

Trong ví dụ này, tất cả các tệp .txt trong thư mục sẽ vẫn bị bỏ qua.

Cách đúng là đưa lại chính thư mục đó vào một dòng riêng, sau đó bỏ qua tất cả các tệp trong thư mục bằng \*, cuối cùng đưa lại \*.txt vào thư mục, như sau:

```
!folder/
folder/* !
folder/*.txt
```

Lưu ý: Đối với tên file bắt đầu bằng dấu chấm than, hãy thêm hai dấu chấm than hoặc thoát bằng ký tự \ :

```
! !bao gồm cái này \
loại trừ cái này
```

## Phần 5.4: Tệp .gitignore toàn cầu

Để Git bỏ qua một số tệp nhất định trên tất cả các kho lưu trữ, bạn có thể [tạo .gitignore toàn cầu](#) bằng lệnh sau trong terminal hoặc dấu nhắc lệnh của bạn:

```
$ git config --global core.excludesfile <Path_To_Global_gitignore_file>
```

Git bây giờ sẽ sử dụng cái này cùng với .gitignore của mỗi kho lưu trữ tài liệu. Quy tắc cho việc này là:

- Nếu tệp .gitignore cục bộ bao gồm một tệp rõ ràng trong khi .gitignore toàn cầu bỏ qua nó, thì .gitignore cục bộ sẽ được ưu tiên (tệp sẽ được bao gồm)
- Nếu kho lưu trữ được sao chép trên nhiều máy thì .gitignore toàn cầu phải được tải lên trên tất cả các máy hoặc ít nhất là bao gồm nó, vì các tệp bị bỏ qua sẽ được đẩy lên kho lưu trữ trong khi PC có .gitignore toàn cầu sẽ không cập nhật nó. Đây là lý do tại sao một .gitignore dành riêng cho kho lưu trữ lại là ý tưởng tốt hơn một kho lưu trữ toàn cầu nếu dự án được thực hiện bởi một nhóm

Tệp này là một nơi tốt để bỏ qua nền tảng, máy hoặc người dùng cụ thể, ví dụ: OSX .DS\_Store, Windows Thumbs.db hoặc Vim \*.ext~ và \*.ext.swp bỏ qua nếu bạn không muốn giữ chúng trong kho lưu trữ. Vì vậy, một thành viên trong nhóm làm việc trên OS X có thể thêm tất cả .DS\_STORE và \_MACOSX (điều này thực sự vô dụng), trong khi một thành viên khác trong nhóm trên Windows có thể bỏ qua tất cả Thumbs.bd

## Phần 5.5: Bỏ qua các tệp đã được cam kết

## kho lưu trữ Git

Nếu bạn đã thêm một tệp vào kho lưu trữ Git của mình và bây giờ muốn ngừng theo dõi tệp đó (để nó không xuất hiện trong các lần xác nhận trong tương lai), bạn có thể xóa tệp đó khỏi chỉ mục:

```
git rm --cached <file>
```

Điều này sẽ xóa tệp khỏi kho lưu trữ và ngăn không cho Git theo dõi những thay đổi tiếp theo. Tùy chọn `--cached` sẽ đảm bảo rằng tệp không bị xóa về mặt vật lý.

Lưu ý rằng nội dung được thêm trước đó của tệp sẽ vẫn hiển thị qua lịch sử Git.

Hãy nhớ rằng nếu bất kỳ ai khác lấy từ kho lưu trữ sau khi bạn xóa tệp khỏi chỉ mục, bản sao của họ sẽ bị xóa về mặt vật lý.

Bạn có thể khiến Git giả vờ rằng phiên bản thư mục làm việc của tệp đã được cập nhật và thay vào đó đọc phiên bản chỉ mục (do đó bỏ qua các thay đổi trong đó) bằng "[bỏ qua cây làm việc](#)" chút:

```
git cập nhật chỉ mục --skip-worktree <file>
```

Việc viết không bị ảnh hưởng bởi bit này, an toàn nội dung vẫn được ưu tiên hàng đầu. Bạn sẽ không bao giờ mất đi những thay đổi quý giá bị bỏ qua của mình; mặt khác, bit này xung đột với việc lưu trữ: để xóa bit này, hãy sử dụng

```
git update-index --no-skip-worktree <file>
```

Đôi khi, người ta khuyên bạn nên nói dối Git và cho rằng tệp đó không thay đổi nếu không kiểm tra nó.

Thoạt nhìn, nó có vẻ như đang bỏ qua bất kỳ thay đổi nào khác đối với tệp mà không xóa nó khỏi chỉ mục của nó:

```
git update-index --assume-unchanged <file>
```

Điều này sẽ buộc git bỏ qua mọi thay đổi được thực hiện trong tệp (hãy nhớ rằng nếu bạn lấy bất kỳ thay đổi nào đối với tệp này hoặc bạn cắt nó đi, những thay đổi bị bỏ qua của bạn sẽ bị mất)

Nếu bạn muốn git "quan tâm" đến tập tin này một lần nữa, hãy chạy lệnh sau:

```
git update-index --no-assume-unchanged <file>
```

## Phần 5.6: Bỏ qua các tệp cục bộ mà không tuân theo quy tắc

`.gitignore` bỏ qua các tệp cục bộ nhưng nó được định nghĩa riêng cho kho lưu trữ và được chia sẻ với những người đóng góp và người dùng khác. Bạn có thể đặt `.gitignore` toàn cầu, nhưng sau đó tất cả các kho lưu trữ của bạn sẽ chia sẻ các cài đặt đó.

Nếu bạn muốn bỏ qua một số tệp nhất định trong kho lưu trữ cục bộ và không biến tệp đó thành một phần của bất kỳ kho lưu trữ nào, hãy chỉnh sửa `.git/info/exclude` bên trong kho lưu trữ của bạn.

Ví dụ:

```
# những tập tin này chỉ bị bỏ qua trên repo này #
những quy tắc này không được chia sẻ với bất kỳ
ai # vì chúng mang tính cá
nhân
gtk_tests.py gui/
gtk/tests/* localhost
```

```
máy chủ pushReports.py/
```

## Phần 5.7: Bỏ qua những thay đổi tiếp theo đối với một tệp (mà không xóa nó)

Đôi khi bạn muốn giữ một tệp trong Git nhưng bỏ qua những thay đổi tiếp theo.

Yêu cầu Git bỏ qua các thay đổi đối với tệp hoặc thư mục bằng chỉ mục cập nhật:

```
git update-index --assume-unchanged my-file.txt
```

Lệnh trên hướng dẫn Git giả sử `my-file.txt` chưa bị thay đổi và không kiểm tra hoặc báo cáo các thay đổi. Tệp vẫn còn trong kho lưu trữ.

Điều này có thể hữu ích khi cung cấp các giá trị mặc định và cho phép ghi đè môi trường cục bộ, ví dụ:

```
# tạo một file với một số giá trị trong cat <<EOF
MySQL_USER=Ứng dụng
MySQL_PASSWORD=FIXME_SECRET_PASSWORD EOF > .env

# cam kết với Git git
add .env git
commit -m "Thêm mẫu .env"

# bỏ qua những thay đổi trong tương lai đối
với .env git update-index --assume-unchanged .env

# cập nhật mật khẩu vi .env của
bạn

# không thay đổi!
trạng thái git
```

## Phần 5.8: Bỏ qua một tập tin trong thư mục bất kỳ

Để bỏ qua một file `foo.txt` trong bất kỳ thư mục nào bạn chỉ cần viết tên của nó:

```
foo.txt # khớp với tất cả các tệp 'foo.txt' trong bất kỳ thư mục nào
```

Nếu bạn chỉ muốn bỏ qua tệp trong một phần của cây, bạn có thể chỉ định các thư mục con của một thư mục cụ thể bằng mẫu `**`:

```
bar/**/foo.txt # khớp với tất cả các tệp 'foo.txt' trong 'bar' và tất cả các thư mục con
```

Hoặc bạn có thể tạo tệp `.gitignore` trong thư mục `bar/`. Tương đương với ví dụ trước sẽ tạo tệp `bar/.gitignore` với các nội dung sau:

```
foo.txt # khớp với tất cả các tệp 'foo.txt' trong bất kỳ thư mục nào dưới bar/
```

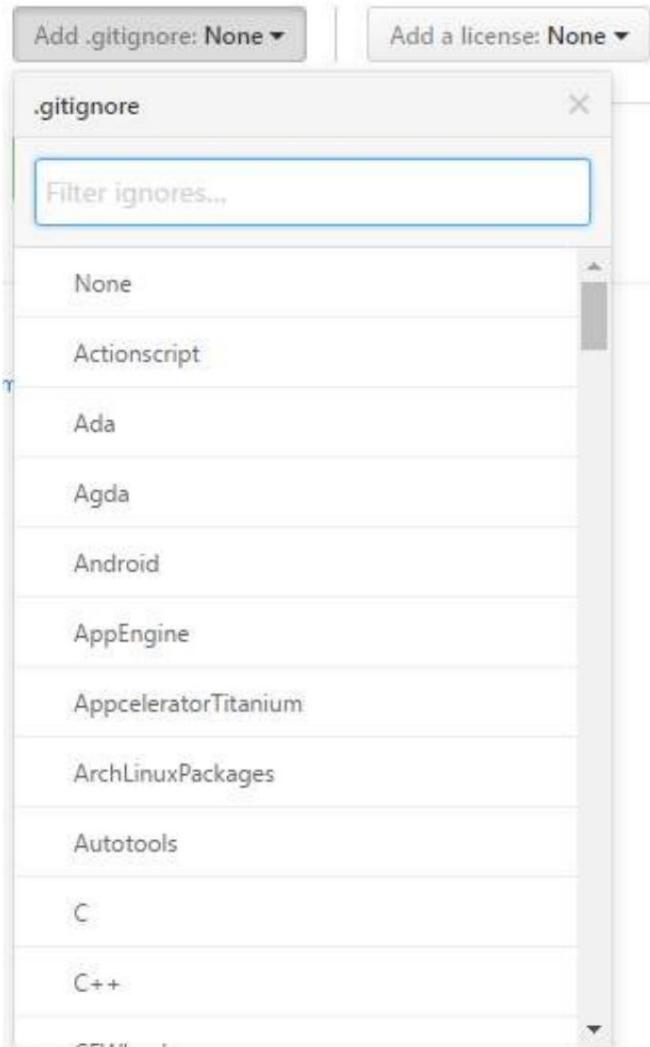
## Phần 5.9: Mẫu `.gitignore` được điền sẵn

Nếu bạn không chắc nên liệt kê quy tắc nào trong tệp `.gitignore` của mình hoặc bạn chỉ muốn thêm các ngoại lệ được chấp nhận chung

đối với dự án của bạn, bạn có thể chọn hoặc tạo tệp .gitignore :

- <https://www.gitignore.io/>
- <https://github.com/github/gitignore>

Nhiều dịch vụ lưu trữ như GitHub và BitBucket cung cấp khả năng tạo tệp .gitignore dựa trên ngôn ngữ lập trình và IDE mà bạn có thể đang sử dụng:



## Mục 5.10: Bỏ qua các tệp trong thư mục con (Nhiều tệp gitignore)

Giả sử bạn có cấu trúc kho lưu trữ như thế này:

```
ví dụ/
out.log src/
<files không được hiển thị>
    đầu ra.log
README.md
```

out.log trong thư mục ví dụ là hợp lệ và cần thiết để dự án thu thập thông tin trong khi cái bên dưới src/ được tạo trong khi gõ lỗi và không nên có trong lịch sử hoặc một phần của kho lưu trữ.

Có hai cách để bỏ qua tập tin này. Bạn có thể đặt đường dẫn tuyệt đối vào tệp .gitignore ở thư mục gốc của thư mục làm việc:

```
# /.gitignore
src/output.log
```

Ngoài ra, bạn có thể tạo tệp .gitignore trong thư mục src/ và bỏ qua tệp có liên quan đến .gitignore này:

```
# /src/.gitignore đầu
ra.log
```

## Phần 5.11: Tạo một thư mục trống

Không thể thêm và chuyển giao một thư mục trống trong Git do Git quản lý các tệp và đính kèm thư mục của chúng vào chúng, điều này làm giảm các cam kết và cải thiện tốc độ. Để giải quyết vấn đề này, có hai phương pháp:

Phương pháp một: .gitkeep

Một cách để giải quyết vấn đề này là sử dụng tệp .gitkeep để đăng ký thư mục cho Git. Để thực hiện việc này, chỉ cần tạo thư mục được yêu cầu và thêm tệp .gitkeep vào thư mục. Tệp này trống và không phục vụ bất kỳ mục đích nào khác ngoài việc chỉ đăng ký thư mục. Để thực hiện việc này trong Windows (có quy ước đặt tên tệp khó xử), chỉ cần mở git bash trong thư mục và chạy lệnh:

```
$ chạm .gitkeep
```

Lệnh này chỉ tạo một tệp .gitkeep trống trong thư mục hiện tại

Phương pháp hai: dummy.txt

Một cách hack khác cho việc này rất giống với cách trên và có thể thực hiện các bước tương tự, nhưng thay vì .gitkeep , bạn chỉ cần sử dụng dummy.txt . Điều này có thêm phần thường là có thể dễ dàng tạo nó trong Windows bằng menu ngữ cảnh. Và bạn cũng có thể để lại những tin nhắn vui nhộn trong đó. Bạn cũng có thể sử dụng tệp .gitkeep để theo dõi thư mục trống. .gitkeep thường là một tệp trống được thêm vào để theo dõi thư mục trống.

## Phần 5.12: Tìm tệp bị .gitignore bỏ qua

Bạn có thể liệt kê tất cả các tệp bị git bỏ qua trong thư mục hiện tại bằng lệnh:

```
trạng thái git --bị bỏ qua
```

Vì vậy, nếu chúng ta có cấu trúc kho lưu trữ như thế này:

```
.git .gitignore ./
example_1 ./dir/example_2 ./example_2
```

...và tệp .gitignore chứa:

```
ví dụ_2
```

...so với kết quả của lệnh sẽ là:

```
trạng thái $ git --bị bỏ qua
```

Trên nhánh chính

Cam kết ban đầu

Các tập tin không bị theo dõi:

(sử dụng "git add <file>..." để bao gồm những gì sẽ được cam kết)

.gitignore .example\_1

Các tệp bị bỏ qua:

(sử dụng "git add -f <file>..." để đưa vào những gì sẽ được cam kết)

thư

mục/ example\_2

Nếu bạn muốn liệt kê các tệp bị bỏ qua đệ quy trong thư mục, bạn phải sử dụng tham số bổ sung - --untracked-files=all

Kết quả sẽ trông như thế này:

\$ trạng thái git --ignored --untracked-files=all

Trên nhánh chính

Cam kết ban đầu

Các tập tin không bị theo dõi:

(sử dụng "git add <file>..." để bao gồm những gì sẽ được cam kết)

.gitignore ví

dụ\_1

Các tệp bị bỏ qua:

(sử dụng "git add -f <file>..." để đưa vào những gì sẽ được cam kết)

thư mục/example\_2

ví dụ\_2

## Mục 5.13: Chỉ bỏ qua một phần của tệp [sơ khai]

Đôi khi bạn có thể muốn có những thay đổi cục bộ trong một tệp mà bạn không muốn chuyển giao hoặc xuất bản. Lý tưởng nhất là cài đặt cục bộ nên được tập trung vào một tệp riêng biệt có thể được đặt vào .gitignore, nhưng đôi khi, như một giải pháp ngắn hạn, có thể hữu ích nếu có một cái gì đó cục bộ trong tệp đã đăng ký.

Bạn có thể làm cho Git "bỏ nhìn thấy" những dòng đó bằng bộ lọc sạch. Họ thậm chí sẽ không xuất hiện trong khác biệt.

Giả sử đây là đoạn trích từ file file1.c:

```
cài đặt cấu trúc s; s.host  
= "localhost"; s.port = 5653;  
s.auth = 1; s.port  
= 15653; //  
NOCOMMIT s.debug = 1; // NOCOMMIT s.auth  
= 0; // KHÔNG CAM KẾT
```

Bạn không muốn xuất bản dòng NOCOMMIT ở bất cứ đâu.

Tạo bộ lọc "nocommit" bằng cách thêm bộ lọc này vào tệp cấu hình Git như .git/config:

```
[bộ lọc "nocommit"]
clean=grep -v NOCOMMIT
```

Thêm (hoặc tạo) phần này vào .git/info/attributes hoặc .gitmodules:

```
bộ lọc file1.c =nocommit
```

Và dòng NOCOMMIT của bạn bị ẩn khỏi Git.

Hãy cẩn thận:

- Sử dụng bộ lọc sạch sẽ làm chậm quá trình xử lý tệp, đặc biệt là trên Windows.
- Dòng bị bỏ qua có thể biến mất khỏi tệp khi Git cập nhật nó. Nó có thể được khắc phục bằng bộ lọc nhòe, nhưng nó phức tạp hơn.
- Chưa được thử nghiệm trên Windows

## Phần 5.14: Bỏ qua những thay đổi trong tệp được theo dõi. [sơ khai]

.gitignore và .git/info/exclude chỉ hoạt động đối với các tệp không bị theo dõi.

Để đặt cờ bỏ qua trên tệp được theo dõi, hãy sử dụng lệnh update-index:

```
git cập nhật chỉ mục --skip-worktree myfile.c
```

Để hoàn nguyên điều này, hãy sử dụng:

```
git update-index --no-skip-worktree myfile.c
```

Bạn có thể thêm đoạn mã này vào cấu hình git toàn cầu của mình để có các lệnh `git ẩn`, `git unhide` và `git ẩn thuận tiện hơn`:

```
[bi
danh] ẩn = cập nhật-index --skip-worktree bỏ ẩn
      = cập nhật-index --no-skip-worktree ẩn = "git ls-
files -v | grep ^[hsS] | cut -c 3-"
```

Bạn cũng có thể sử dụng tùy chọn `--assume-unchanged` với chức năng chỉ mục cập nhật

```
git update-index --assume-unchanged <file>
```

Nếu bạn muốn xem lại tệp tin này để biết những thay đổi, hãy sử dụng

```
git update-index --no-assume-unchanged <file>
```

Khi cờ `--assume-unchanged` được chỉ định, người dùng hứa sẽ không thay đổi tệp và cho phép Git giả định rằng tệp cây đang làm việc khớp với những gì được ghi trong chỉ mục. Git sẽ thất bại trong trường hợp cần sửa đổi tệp này trong chỉ mục ví dụ như khi hợp nhất trong một cam kết; do đó, trong trường hợp tệp giả định không bị theo dõi bị thay đổi ngược dòng, bạn sẽ cần phải xử lý tình huống theo cách thủ công. Trọng tâm nằm ở hiệu suất trong trường hợp này.

Mặc dù cờ `--skip-worktree` rất hữu ích khi bạn hướng dẫn git không chạm vào một tệp cụ thể vì tệp đó sẽ được thay đổi cục bộ và bạn không muốn vô tình thực hiện các thay đổi (tức là tệp cấu hình/thuộc tính được định cấu hình cho một tệp cụ thể). môi trường). Skip-worktree được ưu tiên hơn giả định không thay đổi khi cả hai đều bộ.

## Mục 5.15: Xóa các tệp đã được cam kết nhưng có trong .gitignore

Đôi khi xảy ra trường hợp một tệp đang được theo dõi bởi git, nhưng sau đó một thời gian đã được thêm vào .gitignore, để ngừng theo dõi tệp đó. Tình huống rất phổ biến là quên dọn sạch các tệp như vậy trước khi thêm nó vào .gitignore. Trong trường hợp này, tệp cũ sẽ vẫn còn tồn tại trong kho lưu trữ.

Để khắc phục sự cố này, người ta có thể thực hiện xóa "chạy thử" mọi thứ trong kho lưu trữ, sau đó thêm lại tất cả các tệp. Miễn là bạn không có các thay đổi đang chờ xử lý và tham số `--cached` được truyền, lệnh này khá an toàn để chạy:

```
# Xóa mọi thứ khỏi chỉ mục (các tệp sẽ ở lại trong hệ thống tệp) $ git rm -r --cached .
# Thêm lại mọi thứ (chúng sẽ được thêm ở trạng thái hiện tại, bao gồm các thay đổi) $ git add .
# Cam kết, nếu có gì thay đổi. Bạn sẽ chỉ thấy các phần xóa $ git commit
-m 'Xóa tất cả các tệp có trong .gitignore'
# Cập nhật remote $ git
push Origin master
```

# Chương 6: Git Di

Tham số -p, -u,	Chi tiết
-patch Tạo bản vá	
-s, --no-patch	Ngăn chặn đầu ra khác biệt. Hữu ích cho các lệnh như <code>git show</code> hiển thị bản vá theo mặc định hoặc để hủy tác dụng của <code>--patch</code>
--thô	Tạo sự khác biệt ở định dạng thô
--diff-algorithm=	Chọn một thuật toán khác. Các biến thể như sau: myers, tối thiểu, kiên nhẫn, biểu đồ
--bản tóm tắt	Xuất bản tóm tắt có định nghĩa về thông tin tiêu đề mở rộng như tạo, đổi tên và thay đổi chế độ
--name-only	Hiển thị tên của các tệp đã thay đổi
--tên-trạng thái	Hiển thị tên và trạng thái của các tập tin đã thay đổi. Các trạng thái phổ biến nhất là M (Đã sửa đổi), A (Đã thêm) và D (Đã xóa)
--kiểm tra	Cảnh báo nếu các thay đổi gây ra lỗi xung đột hoặc lỗi khoảng trắng. Những gì được coi là lỗi khoảng trắng được kiểm soát bởi cấu hình <code>core.whitespace</code> . Theo mặc định, các khoảng trắng ở cuối (bao gồm các dòng chỉ bao gồm các khoảng trắng) và một ký tự khoảng trắng ngay sau ký tự tab bên trong dấu thụt lề đầu tiên của dòng được coi là lỗi khoảng trắng. Thoát với trạng thái khác 0 nếu tìm thấy vấn đề. Không tương thích với <code>--exit-code</code> . Thay vì một số ký tự đầu tiên, hãy hiển thị tên đối tượng blob trước và sau hình ảnh đầy đủ trên dòng "index" khi tạo đầu ra định dạng bản vá. Ngoài <code>--full-index</code> , xuất <code>ra</code> một khán
--chỉ mục đầy đủ	biệt nhị phân có thể được áp dụng với <code>git áp dụng</code>
--nhị phân	
-một văn bản	Hãy coi tất cả các tập tin dưới dạng văn bản.
--màu sắc	Đặt chế độ màu; tức là sử dụng <code>--color=always</code> nếu bạn muốn chuyển một khán biệt thành ít hơn và giữ nguyên màu của git

## Mục 6.1: Thể hiện sự khác biệt trong chi nhánh đang hoạt động

### git khác biệt

Điều này sẽ hiển thị những thay đổi chưa được thực hiện trên nhánh hiện tại so với cam kết trước đó. Nó sẽ chỉ hiển thị những thay đổi liên quan đến chỉ mục, nghĩa là nó hiển thị những gì bạn có thể thêm vào lần xác nhận tiếp theo nhưng chưa làm. Để thêm (giai đoạn) những thay đổi này, bạn có thể sử dụng `git add`.

Nếu một tệp được dàn dựng nhưng đã được sửa đổi sau khi nó được dàn dựng, `git diff` sẽ hiển thị sự khác biệt giữa tệp hiện tại và phiên bản được dàn dựng.

## Phần 6.2: Hiển thị các thay đổi giữa hai lần xác nhận

### git diff 1234abc..6789def # cũ mới

Ví dụ: Hiển thị các thay đổi được thực hiện trong 3 lần xác nhận gần nhất:

### git diff @~3..@ # HEAD -3 HEAD

Lưu ý: hai dấu chấm (...) là tùy chọn nhưng sẽ làm rõ hơn.

Điều này sẽ hiển thị sự khác biệt về mặt văn bản giữa các lần xác nhận, bắt kể chúng ở đâu trong cây.

## Phần 6.3: Hiển thị sự khác biệt cho các tệp được phân loại

### git diff --staged

Điều này sẽ hiển thị những thay đổi giữa cam kết trước đó và các tệp hiện đang được tổ chức.

LƯU Ý: Bạn cũng có thể sử dụng các lệnh sau để thực hiện điều tương tự:

```
git khác --được lưu trong bộ nhớ đệm
```

Đây chỉ là từ đồng nghĩa với **--staged** hoặc

```
trạng thái git -v
```

Điều này sẽ kích hoạt cài đặt chi tiết của lệnh trạng thái .

## Mục 6.4: So sánh các nhánh

Hiển thị sự thay đổi giữa đầu mới và đầu gốc:

```
git diff bản gốc mới # tương đương với bản gốc..mới
```

Hiển thị tất cả các thay đổi trên cài mới kể từ khi nó phân nhánh từ bản gốc:

```
git khác bản gốc...mới # tương đương với $(git merge-base original new)..new
```

Chỉ sử dụng một tham số như

git khác bản gốc

tương đương với

git khác bản gốc..HEAD

## Phần 6.5: Hiển thị cả thay đổi theo giai đoạn và không theo giai đoạn

Để hiển thị tất cả các thay đổi theo giai đoạn và không theo giai đoạn, hãy sử dụng:

```
git diff ĐẦU
```

LƯU Ý: Bạn cũng có thể sử dụng lệnh sau:

```
trạng thái git -vv
```

Sự khác biệt là đầu ra của cái sau sẽ thực sự cho bạn biết những thay đổi nào được thực hiện cho cam kết và cái nào không.

## Phần 6.6: Hiển thị sự khác biệt cho một tệp hoặc thư mục cụ thể

```
git khác myfile.txt
```

Hiển thị các thay đổi giữa cam kết trước đó của tệp được chỉ định (myfile.txt) và phiên bản được sửa đổi cục bộ chưa được phân loại.

Điều này cũng hoạt động cho các thư mục:

```
tài liệu khác biệt git
```

Phần trên cho thấy những thay đổi giữa cam kết trước đó của tất cả các tệp trong thư mục được chỉ định (tài liệu/) và các phiên bản được sửa đổi cục bộ của các tệp này chưa được phân loại.

Để hiển thị sự khác biệt giữa một số phiên bản của tệp trong một cam kết nhất định và phiên bản HEAD cục bộ, bạn có thể chỉ định cam kết mà bạn muốn so sánh với:

```
git khac 27fa75e myfile.txt
```

Hoặc nếu bạn muốn xem phiên bản giữa hai lần xác nhận riêng biệt:

```
git khac 27fa75e ada9b57 myfile.txt
```

Để hiển thị sự khác biệt giữa phiên bản được chỉ định bởi hàm băm ada9b57 và cam kết mới nhất trên nhánh my\_branchname chỉ dành cho thư mục tương đối có tên my\_changed\_directory/ bạn có thể thực hiện việc này:

```
git diff ada9b57 my_branchname my_changed_directory/
```

## Mục 6.7: Xem từ-di cho dòng dài

```
git diff [HEAD|--staged...] --word-diff
```

Thay vì hiển thị các dòng đã thay đổi, điều này sẽ hiển thị sự khác biệt trong các dòng. Ví dụ, thay vì:

```
-Xin chào thế giới  
+Xin chào thế giới!
```

Trong đó toàn bộ dòng được đánh dấu là đã thay đổi, word-diff sẽ thay đổi đầu ra thành:

```
Xin chào [-world-]{+world!+}
```

Bạn có thể bỏ qua các dấu [-, -], {+, +} bằng cách chỉ định --word-diff=color hoặc --color=words. Điều này sẽ chỉ sử dụng mã màu để đánh dấu sự khác biệt:

```
@@ -1 +1 @@  
Hello worldworld!
```

## Phần 6.8: Hiển thị sự khác biệt giữa phiên bản hiện tại và phiên bản cuối cùng

```
git diff ĐẦU^ ĐẦU
```

Điều này sẽ hiển thị những thay đổi giữa cam kết trước đó và cam kết hiện tại.

## Phần 6.9: Tạo một di động tương thích với bản vá

Đôi khi bạn chỉ cần một khác biệt để áp dụng bằng bản vá. git --diff thông thường không hoạt động. Thay vào đó hãy thử điều này:

```
git diff --no-prefix > some_file.patch
```

Sau đó, ở một nơi khác bạn có thể đảo ngược nó:

```
vá -p0 < some_file.patch
```

## Mục 6.10: sự khác biệt giữa hai cam kết hoặc chi nhánh

Để xem sự khác biệt giữa hai chi nhánh

```
git khác biệt <nhánh1>..<nhánh2>
```

Để xem sự khác biệt giữa hai chi nhánh

```
git diff <commitId1>..<commitId2>
```

Để xem khác biệt với nhánh hiện tại

```
git khác <nhánh/commitId>
```

Để xem tóm tắt các thay đổi

```
git diff --stat <branch/commitId>
```

Để xem các tập tin đã thay đổi sau một cam kết nhất định

```
git diff --chỉ tên <commitId>
```

Để xem các tập tin khác với một nhánh

```
git diff --chỉ tên <branchName>
```

Để xem các tập tin đã thay đổi trong một thư mục sau một cam kết nhất định

```
git diff --chỉ tên <commitId> <folder_path>
```

## Mục 6.11: Sử dụng meld để xem tất cả các sửa đổi trong thư mục làm việc

```
git DiffTool -t meld --dir-diff
```

sẽ hiển thị những thay đổi thư mục làm việc. Ngoài ra,

```
git DiffTool -t meld --dir-diff [COMMIT_A] [COMMIT_B]
```

sẽ cho thấy sự khác biệt giữa 2 cam kết cụ thể.

## Mục 6.12: Tệp văn bản và tệp nhị phân mã hóa Di UTF-16

Bạn có thể phân biệt các tệp được mã hóa UTF-16 (tệp chuỗi bản địa hóa hệ điều hành iOS và macOS là ví dụ) bằng cách chỉ định cách git phân biệt các tệp này.

Thêm phần sau vào tệp `~/.gitconfig` của bạn .

```
[diff "utf16"]
textconv = "iconv -f utf-16 -t utf-8"
```

iconv là một chương trình để chuyển đổi các bảng mã khác nhau.

Sau đó chỉnh sửa hoặc tạo tệp .gitattributes trong thư mục gốc của kho lưu trữ nơi bạn muốn sử dụng. Hoặc chỉ cần chỉnh sửa ~/.gitattributes.

```
*.strings diff=utf16
```

Điều này sẽ chuyển đổi tất cả các tệp kết thúc bằng .strings trước git diffs.

Bạn có thể thực hiện những điều tương tự đối với các tệp khác có thể được chuyển đổi thành văn bản.

Đối với các tệp plist nhị phân bạn chỉnh sửa .gitconfig

```
[khác "plist"]
textconv = plutil -convert xml1 -o -
```

và .gitattribut

```
*.plist khac=plist
```

# Chương 7: Hoàn tác

## Phần 7.1: Quay lại cam kết trước đó

Để quay lại lần xác nhận trước đó, trước tiên hãy tìm hàm bấm của lần xác nhận đó bằng `git log`.

Để tạm thời quay lại cam kết đó, hãy tách đầu của bạn bằng:

```
kiểm tra git 789abcd
```

Điều này đặt bạn vào cam kết 789abcd. Bây giờ bạn có thể thực hiện các cam kết mới bên trên cam kết cũ này mà không ảnh hưởng đến nhánh mà bạn đang đứng đầu. Mọi thay đổi có thể được thực hiện thành một nhánh thích hợp bằng cách sử dụng nhánh hoặc thanh toán `-b`.

Để quay lại cam kết trước đó trong khi vẫn giữ các thay đổi:

```
thiết lập lại git --soft 789abcd
```

Để khôi phục cam kết cuối cùng :

```
thiết lập lại git --đầu mềm~
```

Để loại bỏ vĩnh viễn mọi thay đổi được thực hiện sau một cam kết cụ thể, hãy sử dụng:

```
thiết lập lại git --hard 789abcd
```

Để loại bỏ vĩnh viễn mọi thay đổi được thực hiện sau lần xác nhận cuối cùng :

```
thiết lập lại git --đầu cứng~
```

Lưu ý: Mặc dù bạn có thể khôi phục các cam kết đã loại bỏ bằng cách sử dụng `reflog` và đặt lại, nhưng những thay đổi chưa được cam kết sẽ không thể khôi phục được. Sử dụng `kho git; git reset` thay vì `git reset --hard` để an toàn.

## Phần 7.2: Hoàn tác các thay đổi

Hoàn tác các thay đổi đối với một tập tin hoặc thư mục trong bàn sao làm việc.

```
kiểm tra git -- file.txt
```

Được sử dụng trên tất cả các đường dẫn tệp, đệ quy từ thư mục hiện tại, nó sẽ hoàn tác mọi thay đổi trong bàn sao làm việc.

```
kiểm tra git -- .
```

Để chỉ hoàn tác các phần thay đổi, hãy sử dụng `--patch`. Đối với mỗi thay đổi, bạn sẽ được hỏi xem có nên hoàn tác hay không.

```
kiểm tra git --patch -- thư mục
```

Để hoàn tác các thay đổi được thêm vào chỉ mục.

```
thiết lập lại git --hard
```

Nếu không có cờ `--hard`, thao tác này sẽ thực hiện thiết lập lại mềm.

Với các cam kết cục bộ mà bạn chưa chuyển sang điều khiển từ xa, bạn cũng có thể thực hiện thiết lập lại mềm. Do đó bạn có thể làm lại các tập tin

và sau đó là các cam kết.

git đặt lại Đầu ~ 2

Ví dụ trên sẽ hủy bỏ hai lần xác nhận cuối cùng của bạn và trả các tệp về bản sao làm việc của bạn. Sau đó, bạn có thể thực hiện thêm các thay đổi và cam kết mới.

Cần thận: Tất cả các thao tác này, ngoài việc đặt lại mềm, sẽ xóa vĩnh viễn các thay đổi của bạn. Để có tùy chọn an toàn hơn, hãy sử dụng `git stash -p` hoặc `git stash` tương ứng. Sau này bạn có thể hoàn tác bằng `stash pop` hoặc xóa vĩnh viễn bằng `stash drop`.

## Phần 7.3: Sử dụng reflog

Nếu bạn làm hỏng `rebase`, một tùy chọn để bắt đầu lại là quay lại cam kết (pre `rebase`). Bạn có thể thực hiện việc này bằng cách sử dụng `reflog` (có lịch sử mọi việc bạn đã làm trong 90 ngày qua - tính năng này có thể được định cấu hình):

```
$ git reflog
4a5cbb3 HEAD@{0}: rebase đã hoàn tất: quay lại refs/heads/foo 4a5cbb3 HEAD@{1}:
rebase: đã sửa cái này cái kia 904f7f0 HEAD@{2}: rebase:
kiểm tra ngược dòng/master 3cbe20a HEAD@{ 3}: cam kết: đã sửa lỗi
như vậy và như vậy
...
```

Bạn có thể thấy cam kết trước khi `rebase` là `HEAD@{3}` (bạn cũng có thể kiểm tra hàm `bash`):

kiểm tra git HEAD@{3}

Bây giờ bạn tạo một nhánh mới/xóa nhánh cũ/thử `rebase` lại.

Bạn cũng có thể đặt lại trực tiếp về một điểm trong `reflog` của mình, nhưng chỉ thực hiện việc này nếu bạn chắc chắn 100% đó là điều bạn muốn làm:

đặt lại git -hard HEAD@{3}

Điều này sẽ đặt cây `git` hiện tại của bạn khớp với trạng thái tại thời điểm đó (Xem Hoàn tác các thay đổi).

Điều này có thể được sử dụng nếu bạn tạm thời thấy một nhánh hoạt động tốt như thế nào khi được dựa trên một nhánh khác, nhưng bạn không muốn giữ kết quả.

## Mục 7.4: Hoàn tác việc hợp nhất

Hoàn tác việc hợp nhất chưa được đẩy tới điều khiển từ xa

Nếu bạn chưa đẩy việc hợp nhất của mình sang kho lưu trữ từ xa thì bạn có thể làm theo quy trình tương tự như khi hoàn tác cam kết mặc dù có một số khác biệt nhỏ.

Đặt lại là tùy chọn đơn giản nhất vì nó sẽ hoàn tác cả cam kết hợp nhất và mọi cam kết được thêm từ nhánh.

Tuy nhiên, bạn sẽ cần biết SHA cần đặt lại về cái gì, điều này có thể phức tạp vì `nhật ký git` của bạn giờ đây sẽ hiển thị các cam kết từ cả hai nhánh. Nếu bạn đặt lại về cam kết sai (ví dụ: một cam kết ở nhánh khác), nó có thể hủy công việc đã cam kết.

> git reset --hard <lần xác nhận cuối cùng từ nhánh bạn đang ở>

Hoặc, giả sử việc hợp nhất là cam kết gần đây nhất của bạn.

```
> git thiết lập lại ĐẦU~
```

Việc hoàn nguyên sẽ an toàn hơn vì nó sẽ không phá hủy công việc đã cam kết nhưng đòi hỏi nhiều công việc hơn vì bạn phải hoàn nguyên việc hoàn nguyên trước khi có thể hợp nhất nhánh trở lại (xem phần tiếp theo).

Hoàn tác việc hợp nhất được đẩy tới điều khiển từ xa

Giả sử bạn hợp nhất trong một tính năng mới (add-gremlins)

```
> tính năng hợp nhất git /add-gremlins
...
#Giải quyết mọi xung đột khi hợp nhất
> git commit #commit việc hợp nhất
...
> đẩy git
...
501b75d..17a51fd chủ -> chủ
```

Sau đó, bạn phát hiện ra rằng tính năng bạn vừa hợp nhất đã phá vỡ hệ thống đối với các nhà phát triển khác, tính năng này phải được hoàn tác ngay lập tức và việc sửa tính năng này sẽ mất quá nhiều thời gian nên bạn chỉ muốn hoàn tác việc hợp nhất.

```
> git hoàn nguyên -m 1 17a51fd
...
> đẩy git
...
17a51fd..e443799 chủ -> chủ
```

Tại thời điểm này, các gremlin đã ra khỏi hệ thống và các nhà phát triển đồng nghiệp của bạn đã ngừng la mắng bạn. Tuy nhiên, chúng tôi vẫn chưa kết thúc. Sau khi khắc phục sự cố với tính năng add-gremlins, bạn sẽ cần hoàn tác việc hoàn nguyên này trước khi có thể hợp nhất trở lại.

```
> tính năng kiểm tra git /add-gremlins
...
#Various cam kết sửa lỗi. > chủ kiểm tra
git
...
> git hoàn nguyên e443799
...
> tính năng hợp nhất git /add-gremlins
...
#Khắc phục mọi xung đột hợp nhất do sửa lỗi gây ra > git commit
#commit việc hợp nhất
...
> đẩy git
```

Tại thời điểm này, tính năng của bạn đã được thêm thành công. Tuy nhiên, do các lỗi thuộc loại này thường xuất hiện do xung đột hợp nhất nên quy trình làm việc hơi khác đôi khi sẽ hữu ích hơn vì nó cho phép bạn khắc phục xung đột hợp nhất trên nhánh của mình.

```
> tính năng kiểm tra git /add-gremlins
...
#Hợp nhất trong bản gốc và hoàn nguyên việc hoàn nguyên ngay lập tức. Điều này đặt nhánh của bạn ở
#trạng thái hỏng tương tự như nhánh chính trước đó.
> git hợp nhất chủ
...
> git hoàn nguyên e443799
...
#Bây giờ hãy tiếp tục và sửa lỗi (có nhiều cam kết khác nhau ở đây)
```

```
> chủ kiểm tra git
...
#Không cần hoàn nguyên việc hoàn nguyên vào thời điểm này vì việc này đã được thực hiện trước đó
> tính năng hợp nhất git /add-gremlins
...
#Khắc phục mọi xung đột hợp nhất do sửa lỗi gây ra > git commit
#commit việc hợp nhất
...
> đẩy git
```

## Phần 7.5: Hoàn nguyên một số cam kết hiện có

Sử dụng git Revert để hoàn nguyên các cam kết hiện có, đặc biệt khi các cam kết đó đã được đẩy đến kho lưu trữ từ xa. Nó ghi lại một số cam kết mới để đảo ngược tác dụng của một số cam kết trước đó mà bạn có thể thực hiện một cách an toàn mà không cần viết lại lịch sử.

Đừng sử dụng `git push --force` trừ khi bạn muốn hạ thấp sự phản đối của tất cả những người dùng khác của kho lưu trữ đó.

Không bao giờ viết lại lịch sử công cộng.

Ví dụ: nếu bạn vừa đưa ra một cam kết có lỗi và bạn cần sao lưu nó, hãy làm như sau:

```
git hoàn nguyên HEAD~1
git đẩy
```

Bây giờ bạn có thể tự do hoàn nguyên cam kết hoàn nguyên cục bộ, sửa mã của mình và đẩy mã tốt:

```
git hoàn nguyên HEAD~1
công việc .. công việc .. công
việc .. git .
add -A git commit -m "Cập nhật mã lỗi" git
push
```

Nếu cam kết bạn muốn hoàn nguyên đã quay trở lại trong lịch sử, bạn chỉ cần chuyển hàm bấm cam kết. Git sẽ tạo một cam kết phản đối hoàn tác cam kết ban đầu của bạn, bạn có thể đẩy cam kết này vào điều khiển từ xa một cách an toàn.

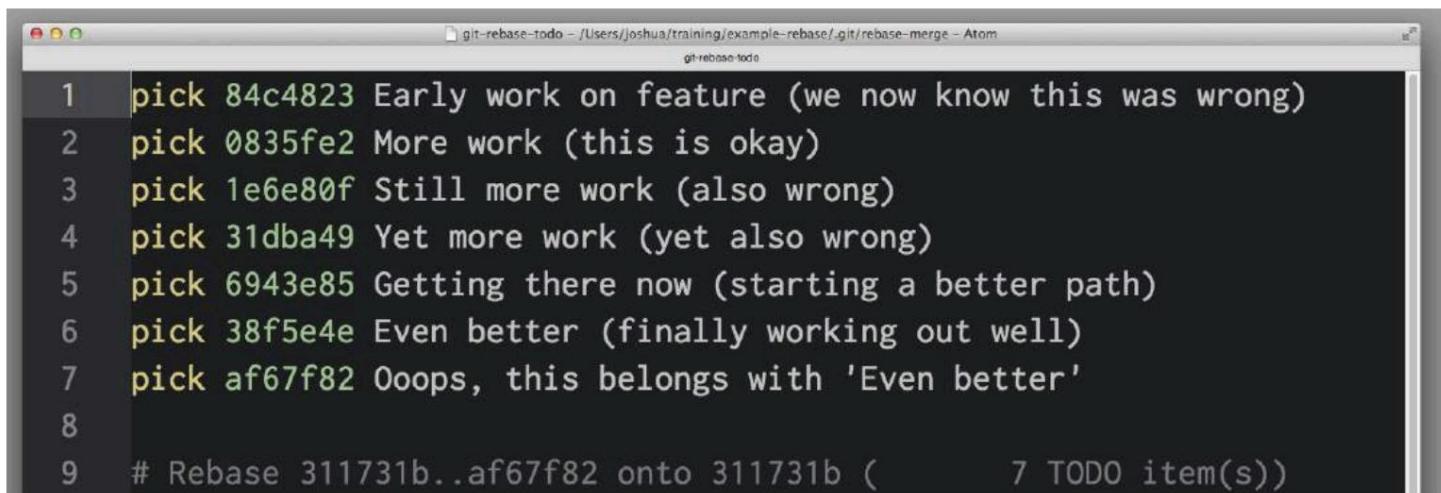
```
git hoàn nguyên 912aaaf0228338d0c8fb8cca0a064b0161a451fdc git đẩy
```

## Mục 7.6: Undo/Redo một loạt các commit

Giả sử bạn muốn hoàn tác hàng tá lần xác nhận và bạn chỉ muốn một số trong số đó.

```
git rebase -i <SHA trước đó>
```

`-i` đặt rebase ở "chế độ tương tác". Nó bắt đầu giống như rebase đã thảo luận ở trên, nhưng trước khi phát lại bất kỳ cam kết nào, nó tạm dừng và cho phép bạn sửa đổi nhẹ nhàng từng cam kết khi nó được phát lại.`rebase -i` sẽ mở trong trình soạn thảo văn bản mặc định của bạn, với một danh sách các cam kết được áp dụng, như thế này :



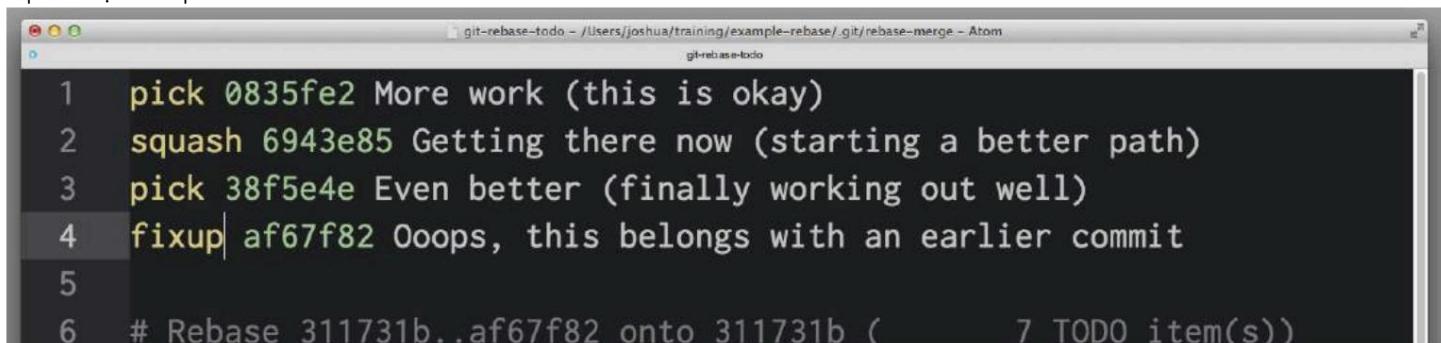
```

git-rebase-todo - /Users/joshua/training/example-rebase/.git/rebase-merge - Atom
git-rebase-todo

1 pick 84c4823 Early work on feature (we now know this was wrong)
2 pick 0835fe2 More work (this is okay)
3 pick 1e6e80f Still more work (also wrong)
4 pick 31dba49 Yet more work (yet also wrong)
5 pick 6943e85 Getting there now (starting a better path)
6 pick 38f5e4e Even better (finally working out well)
7 pick af67f82 Ooops, this belongs with 'Even better'
8
9 # Rebase 311731b..af67f82 onto 311731b (      7 TODO item(s))

```

Để bỏ một cam kết, chỉ cần xóa dòng đó trong trình soạn thảo của bạn. Nếu bạn không muốn các bad commit trong dự án của mình nữa, bạn có thể xóa dòng 1 và 3-4 ở trên. Nếu bạn muốn kết hợp hai commit với nhau, bạn có thể sử dụng lệnh squash hoặc fixup



```

git-rebase-todo - /Users/joshua/training/example-rebase/.git/rebase-merge - Atom
git-rebase-todo

1 pick 0835fe2 More work (this is okay)
2 squash 6943e85 Getting there now (starting a better path)
3 pick 38f5e4e Even better (finally working out well)
4 fixup af67f82 Ooops, this belongs with an earlier commit
5
6 # Rebase 311731b..af67f82 onto 311731b (      7 TODO item(s))

```

# Chương 8: Sáp nhập

Tham số	Chi tiết
-m	Thông báo được đưa vào cam kết hợp nhất
-v	Hiển thị đầu ra dài dòng
--Huỷ bỏ	Cố gắng hoàn nguyên tất cả các tệp về trạng thái
của chúng --ff-only	Hủy bỏ ngay lập tức khi cần có cam kết hợp nhất
--không-ff	Buộc tạo cam kết hợp nhất, ngay cả khi nó không bắt buộc
--no-commit	Giả vờ việc hợp nhất không thành công để cho phép kiểm tra và điều chỉnh kết quả
--stat	Hiển thị một khía cạnh sau khi hoàn thành hợp nhất
-n/--no-stat	Không hiển thị khía cạnh
--bí đáo	Cho phép một cam kết duy nhất trên nhánh hiện tại với các thay đổi đã hợp nhất

## Mục 8.1: Hợp nhất tự động

Khi các cam kết trên hai nhánh không xung đột, Git có thể tự động hợp nhất chúng:

```
~/Stack Overflow(branch:master) » git merge another_branch Tự động hợp nhất
file_a Hợp nhất được thực
hiện bằng chiến lược 'dệ quy' . tập tin_a | 2 +- 1
file đã thay đổi,
1 lần chèn(+), 1 lần xóa(-)
```

## Phần 8.2: Tìm tất cả các nhánh không có thay đổi được hợp nhất

Đôi khi bạn có thể có các nhánh xung quanh đã được sáp nhập các thay đổi của chúng thành nhánh chính. Điều này tìm thấy tất cả các nhánh không phải là nhánh chính và không có cam kết duy nhất so với nhánh chính. Điều này rất hữu ích cho việc tìm kiếm các nhánh không bị xóa sau khi PR được sáp nhập vào nhánh chính.

```
cho nhánh trong $(git Branch -r) ; do [ "$
{branch}" != "origin/master" ] && [ $(git diff master..${branch} | wc -l) -eq 0 ] && echo -e `git show --pretty= định dạng:"%ci %cr"
$branch | head -n 1`\\t$branch xong | sắp xếp -r
```

## Mục 8.3: Hủy bỏ việc hợp nhất

Sau khi bắt đầu hợp nhất, bạn có thể muốn dừng hợp nhất và đưa mọi thứ về trạng thái trước khi hợp nhất. Sử dụng --abort:

```
hợp nhất git --abort
```

## Phần 8.4: Hợp nhất với một cam kết

Hành vi mặc định là khi quá trình hợp nhất được giải quyết dưới dạng chuyển tiếp nhanh, chỉ cập nhật con trỏ nhánh mà không tạo cam kết hợp nhất. Sử dụng --no-ff để giải quyết.

```
git merge <branch_name> --no-ff -m "<tin nhắn cam kết>"
```

## Phần 8.5: Chỉ giữ các thay đổi từ một phía của việc hợp nhất

Trong quá trình hợp nhất, bạn có thể chuyển --ours hoặc --theirs tới git kiểm tra để thực hiện tất cả các thay đổi cho một tệp từ bên này hoặc bên kia của quá trình hợp nhất.

```
$ gitcheck --ours -- file1.txt # Sử dụng phiên bản file1 của chúng tôi, xóa tất cả các thay đổi của họ $ gitcheck --  
theirs -- file2.txt # Sử dụng phiên bản file2 của họ, xóa tất cả các thay đổi của chúng tôi
```

## Mục 8.6: Hợp nhất một nhánh vào một nhánh khác

git hợp nhất InBranch

Thao tác này sẽ hợp nhất nhánh đếnBranch vào nhánh mà bạn hiện đang ở. Ví dụ: nếu bạn hiện đang ở nhánh chính thì đếnBranch sẽ được hợp nhất thành nhánh chính.

Việc hợp nhất có thể tạo ra xung đột trong một số trường hợp. Nếu điều này xảy ra, bạn sẽ thấy thông báo Tự động hợp nhất không thành công; khắc phục xung đột và sau đó cam kết kết quả. Bạn sẽ cần chỉnh sửa thủ công các tệp bị xung đột hoặc để hoàn tác nỗ lực hợp nhất của mình, hãy chạy:

hợp nhất git --abort

# Chương 9: Các mô-đun con

## Phần 9.1: Nhân bản kho lưu trữ Git có các mô-đun con

Khi sao chép một kho lưu trữ sử dụng mô-đun con, bạn sẽ cần khởi tạo và cập nhật chúng.

```
$ git clone --recursive https://github.com/username/repo.git
```

Điều này sẽ sao chép các mô-đun con được tham chiếu và đặt chúng vào các thư mục thích hợp (bao gồm các mô-đun con bên trong các mô-đun con). Điều này tương đương với việc chạy `git submodule update --init --recursive` ngay sau khi sao chép xong.

## Phần 9.2: Cập nhật mô-đun con

Một mô hình con tham chiếu một cam kết cụ thể trong kho lưu trữ khác. Để kiểm tra trạng thái chính xác được tham chiếu cho tất cả các mô hình con, hãy chạy

```
cập nhật mô-đun con git --recursive
```

Đôi khi, thay vì sử dụng trạng thái được tham chiếu, bạn muốn cập nhật toàn bộ của mình lên trạng thái mới nhất của mô-đun con đó trên điều khiển từ xa. Để kiểm tra tất cả các mô-đun con về trạng thái mới nhất trên điều khiển từ xa bằng lệnh duy nhất, bạn có thể sử dụng

```
mô-đun con git foreach git pull <remote> <branch>
```

hoặc sử dụng các đối số `kéo git` mặc định

```
mô-đun con git foreach git pull
```

Lưu ý rằng điều này sẽ chỉ cập nhật bản sao làm việc cục bộ của bạn. Chạy `trạng thái git` sẽ liệt kê thư mục mô hình con là bẩn nếu nó thay đổi do lệnh này. Thay vào đó, để cập nhật kho lưu trữ của bạn để tham chiếu trạng thái mới, bạn phải thực hiện các thay đổi:

```
git thêm <submodule_directory> git  
cam kết
```

Có thể có một số thay đổi mà bạn thực hiện có thể gây ra xung đột khi hợp nhất nếu bạn sử dụng `git pull` để bạn có thể sử dụng `git pull --rebase` để tua lại các thay đổi của mình lên đầu, hầu hết thời gian, điều đó sẽ làm giảm nguy cơ xung đột. Ngoài ra nó kéo tất cả các chi nhánh về địa phương.

```
mô-đun con git foreach git pull --rebase
```

Để kiểm tra trạng thái mới nhất của một mô hình con cụ thể, bạn có thể sử dụng:

```
cập nhật mô-đun con git --remote <submodule_directory>
```

## Phần 9.3: Thêm mô-đun con

Bạn có thể bao gồm một kho lưu trữ Git khác làm thư mục trong dự án của mình, được Git theo dõi:

```
$ git mô-đun con thêm https://github.com/jquery/jquery.git
```

Bạn nên thêm và chuyển giao tệp .gitmodules mới ; điều này cho Git biết những mô-đun con nào sẽ được sao chép khi chạy cập nhật mô-đun con git .

## Mục 9.4: Thiết lập mô-đun con đi theo nhánh

Một mô hình con luôn được kiểm tra tại một cam kết cụ thể SHA1 ("gitlink", mục nhập đặc biệt trong chỉ mục của repo gốc)

Nhưng người ta có thể yêu cầu cập nhật mô hình con đó lên cam kết mới nhất của một nhánh của kho lưu trữ từ xa mô hình con.

Thay vì đi vào từng mô-đun con, thực hiện lệnh kiểm tra git abranch --track Origin/abranch, git pull, bạn có thể chỉ cần thực hiện (từ repo gốc) a:

cập nhật mô-đun con git --remote --recursive

Vì SHA1 của mô hình con sẽ thay đổi nên bạn vẫn cần phải thực hiện theo:

```
git thêm .
git commit -m "cập nhật mô-đun con"
```

Điều đó giả sử các mô-đun con là:

- hoặc được thêm vào một nhánh để theo dõi:

```
git submodule -b abranch -- /url/of submodule/repo
```

- hoặc được định cấu hình (cho mô-đun con hiện có) để đi theo một nhánh:

```
cd /path/to/parent/repo git
config -f .gitmodules submodule.asubmodule.branch abranch
```

## Phần 9.5: Di chuyển một mô hình con

Phiên bản > 1.8

Chạy:

```
$ git mv /path/to/module new/path/to/module
```

Phiên bản 1.8

1. Chính sửa .gitmodules và thay đổi đường dẫn của mô hình con một cách thích hợp và đặt nó vào chỉ mục bằng git add .gitmodules.
2. Nếu cần, hãy tạo thư mục mẹ của vị trí mới của mô hình con (mkdir -p /path/to).
3. Di chuyển tất cả nội dung từ thư mục cũ sang thư mục mới (mv -vi /path/to/module new/path/to/module).
4. Đảm bảo Git theo dõi thư mục này (git add /path/to).
5. Xóa thư mục cũ bằng git rm --cached /path/to/module.
6. Di chuyển thư mục .git/modules//path/to/module với tất cả nội dung của nó sang .git/modules//path/to/module.
7. Chính sửa tệp .git/modules//path/to/config, đảm bảo rằng mục cây công việc trả đến các vị trí mới, vì vậy trong ví dụ này nó phải là Worktree = ../../../../path/to/module. Thông thường nên có thêm hai thư mục .. then trong đường dẫn trực tiếp ở vị trí đó. .

Chỉnh sửa tệp /path/to/module/.git, đảm bảo rằng đường dẫn

trong đó trỏ đến vị trí mới chính xác bên trong thư mục .git của dự án chính , vì vậy trong ví dụ này  
gitdir: ../../../.git/modules//path/to/module.

dầu ra trạng thái git trông như thế này sau đó:

```
# Trên nhánh chính #
Những thay đổi cần cam kết: # (sử dụng "git reset HEAD <file>..." để bỏ giai đoạn) #

#      đã sửa đổi: .gitmodules được
đổi tên thành: old/path/to submodule -> new/path/to submodule
# #
```

8. Cuối cùng, cam kết thay đổi.

Ví dụ này từ [Stack Overflow](#), bởi [Axel Beckert](#)

## Phần 9.6: Xóa mô-đun con

Phiên bản > 1.8

Bạn có thể xóa một mô hình con (ví dụ the\_submodule) bằng cách gọi:

```
$ git mô-đun con deinit the_submodule $ git rm
the_submodule
```

- **git submodule deinit the\_submodule** xóa mục nhập của the\_submodules khỏi .git/config. Điều này loại trừ the\_submodule khỏi cập nhật mô hình con git , đồng bộ hóa mô hình con git và mô hình con git foreach và xóa nội dung cục bộ của nó ( nguồn ). Ngoài ra, điều này sẽ không được hiển thị dưới dạng thay đổi trong kho lưu trữ chính của bạn. **git submodule init** và **git submodule update** sẽ khôi phục lại mô hình con mà không có những thay đổi có thể cam kết trong kho lưu trữ chính của bạn.
- **git rm the\_submodule** sẽ xóa mô hình con khỏi cây công việc. Các tệp cũng như mục nhập của mô-đun con trong tệp .gitmodules ( nguồn ) sẽ biến mất. Tuy nhiên , nếu chỉ chạy **git rm the\_submodule** (không có **mô-đun con git deinit the\_submodule** trước đó) được chạy thì mục nhập của mô-đun con trong tệp .git/config của bạn sẽ vẫn còn.

Phiên bản < 1.8

Lấy từ [đây](#):

1. Xóa phần có liên quan khỏi tệp .gitmodules .
2. Giai đoạn thay đổi .gitmodules **git add .gitmodules** 3. Xóa phần có liên quan khỏi .git/config.
4. Chạy **git rm --cached path\_to\_submodule** (không có dấu gạch chéo).
5. Chạy **rm -rf .git/modules/path\_to\_submodule** 6. Cam kết **git commit -m "Removed submodule <name>"**
7. Xóa các tập tin mô-đun con hiện chưa được theo dõi
8. **rm -rf path\_to\_submodule**

# Chương 10: Cam kết

## Tham số

## Chi tiết

--tín nhắn, -m	Thông báo để đưa vào cam kết. Việc chỉ định tham số này sẽ bỏ qua hành vi mở trình soạn thảo thông thường của Git.
--sửa đổi	Chỉ định rằng những thay đổi hiện đang được thực hiện sẽ được thêm (sửa đổi) vào cam kết trước đó. Hãy cẩn thận, điều này có thể viết lại lịch sử!
-không chỉnh sửa	Sử dụng thông báo cam kết đã chọn mà không cần khởi chạy trình chỉnh sửa. Ví dụ: <code>git commit --amend --no-edit</code> sửa đổi một cam kết mà không thay đổi thông điệp cam kết của nó.
--tất cả, -a	Cam kết tất cả các thay đổi, bao gồm cả những thay đổi chưa được tổ chức.
--ngày	Đặt thủ công ngày sẽ được liên kết với cam kết.
--chỉ một	Chỉ cam kết các đường dẫn được chỉ định. Điều này sẽ không cam kết những gì bạn hiện đã dàn dựng trừ khi được yêu cầu làm như vậy.
-patch, -p	Sử dụng giao diện lựa chọn bản vá tương tác để chọn những thay đổi cần thực hiện.
-help	Hiển thị trang man cho <code>git commit</code> Sign commit,
-S[keyid], -S --gpg-sign[=keyid], -S --no-gpg-sign -n,	GPG-sign commit, countermand commit.gpgSign config Biến đổi
--no-verify	Tùy chọn này bỏ qua các hook pre-commit và commit-msg. Xem thêm Mô

Cam kết với Git cung cấp trách nhiệm giải trình bằng cách gán cho tác giả những thay đổi về mã. Git cung cấp nhiều tính năng về tính đặc hiệu và bảo mật của các cam kết. Chủ đề này giải thích và trình bày các phương pháp cũng như quy trình phù hợp khi cam kết với Git.

## Phần 10.1: Giai đoạn và cam kết thay đổi

## Những thứ cơ bản

Sau khi thực hiện các thay đổi đối với mã nguồn của mình, bạn nên xử lý các thay đổi đó bằng Git trước khi có thể thực hiện chúng.

Ví dụ: nếu bạn thay đổi README.md và Program.py:

`git thêm README.md chương trình.py`

Điều này cho git biết rằng bạn muốn thêm các tệp vào lần xác nhận tiếp theo mà bạn thực hiện.

Sau đó, cam kết thay đổi của bạn với

`cam kết git`

Lưu ý rằng thao tác này sẽ mở trình soạn thảo văn bản, thường là vim. Nếu bạn không quen với vim, bạn có thể muốn biết rằng bạn có thể nhấn i để chuyển sang chế độ chèn, viết thông báo cam kết, sau đó nhấn Esc và :wq để lưu và thoát. Để tránh mở trình soạn thảo văn bản, chỉ cần thêm cờ -m vào tin nhắn của bạn

`git commit -m "Thông báo cam kết tại đây"`

Thông báo cam kết thường tuân theo một số quy tắc định dạng cụ thể, hãy xem Thông báo cam kết tốt để biết thêm thông tin.

## Phím tắt

Nếu bạn đã thay đổi nhiều tệp trong thư mục, thay vì liệt kê từng tệp, bạn có thể sử dụng:

```
git thêm --tất cả          # tương đương với "git add -a"
```

Hoặc để thêm tất cả các thay đổi, không bao gồm các tệp đã bị xóa, từ thư mục cấp cao nhất và các thư mục con:

```
git thêm .
```

Hoặc chỉ thêm các tệp hiện đang được theo dõi ("cập nhật"):

```
git thêm -u
```

Nếu muốn, hãy xem lại các thay đổi theo giai đoạn:

```
trạng thái          # hiển thị danh sách các tập tin đã thay đổi # hiển
git git diff --cached    thị các thay đổi theo giai đoạn bên trong các tập tin theo giai đoạn
```

Cuối cùng, cam kết thay đổi:

```
git commit -m "Thông báo cam kết tại đây"
```

Ngoài ra, nếu bạn chỉ sửa đổi các tệp hiện có hoặc các tệp đã xóa và chưa tạo bất kỳ tệp mới nào, bạn có thể kết hợp các hành động của `git add` và `git commit` trong một lệnh duy nhất:

```
git commit -am "Thông báo cam kết tại đây"
```

Lưu ý rằng thao tác này sẽ xử lý tất cả các tệp đã sửa đổi theo cách tương tự như `git add --all`.

Dữ liệu nhạy cảm

Bạn không bao giờ nên cam kết bất kỳ dữ liệu nhạy cảm nào, chẳng hạn như mật khẩu hoặc thậm chí cả khóa riêng tư. Nếu trường hợp này xảy ra và các thay đổi đã được đẩy lên máy chủ trung tâm, hãy coi mọi dữ liệu nhạy cảm đã bị xâm phạm. Nếu không, có thể xóa dữ liệu đó sau đó. Một giải pháp nhanh chóng và dễ dàng là sử dụng "BFG Repo-Cleaner": <https://rtyley.github.io/bfg-repo-cleaner/>.

Lệnh `bfg --replace-text password.txt` my-repo.git đọc mật khẩu từ tệp `pass.txt` và thay thế chúng bằng `***REMOVED***`. Hoạt động này xem xét tất cả các cam kết trước đó của toàn bộ kho lưu trữ.

## Phần 10.2: Thông báo cam kết tốt

Điều quan trọng là ai đó duyệt qua `nhật ký git` để dễ dàng hiểu được nội dung của từng cam kết.

Thông báo cam kết tốt thường bao gồm một số nhiệm vụ hoặc vấn đề trong trình theo dõi và mô tả ngắn gọn về những gì đã được thực hiện và lý do cũng như đôi khi cả cách thức thực hiện.

Tin nhắn tốt hơn có thể trông giống như:

```
NHẬM VỤ-123: Triển khai đăng nhập thông qua OAuth
TASK-124: Thêm tính năng tự động thu nhỏ tệp JS/CSS
TASK-125: Sửa lỗi minifier khi tên > 200 ký tự
```

Trong khi các thông báo sau đây sẽ không hữu ích bằng:

```
sửa chữa          // Cái gì đã được sửa?
```

```
chỉ một chút thay đổi // Điều gì đã thay đổi?  
TASK-371 // Không có mô tả nào cả, người đọc sẽ cần tự mình xem trình theo dõi để biết lời giải thích IFoo được triển khai  
trong IBar // Tại sao lại cần nó?
```

Một cách dễ kiểm tra xem tin nhắn cam kết có được viết đúng tâm trạng hay không là thay chỗ trống bằng tin nhắn đó và xem nó có hợp lý hay không:

Nếu tôi thêm cam kết này, tôi sẽ \_\_\_\_\_ vào kho lưu trữ của tôi.

Bảy quy tắc của một thông điệp cam kết git tuyệt vời

1. Tách dòng chủ đề khỏi nội dung bằng một dòng trống
2. Giới hạn dòng chủ đề ở 50 ký tự
3. Viết hoa dòng chủ đề
4. Không kết thúc dòng chủ đề bằng dấu chấm
5. Sử dụng thẻ mệnh lệnh [lệnh](#) trong dòng chủ đề
6. Ngắt dòng thủ công mỗi dòng nội dung ở mức 72 ký tự
7. Sử dụng nội dung để giải thích cái gì và tại sao thay vì làm thế nào

[7 quy tắc từ blog của Chris Beams](#).

## Mục 10.3: Sửa đổi cam kết

Nếu cam kết mới nhất của bạn chưa được xuất bản (chưa được đẩy lên kho lưu trữ ngược dòng) thì bạn có thể sửa đổi cam kết của mình.

`cam kết git --amend`

Điều này sẽ đưa các thay đổi hiện đang được thực hiện vào cam kết trước đó.

Lưu ý: Điều này cũng có thể được sử dụng để chỉnh sửa thông báo cam kết không chính xác. Nó sẽ hiển thị trình soạn thảo mặc định (thường là vi / vim / emacs) và cho phép bạn thay đổi thông báo trước đó.

Để chỉ định nội tuyến thông điệp cam kết:

`git commit --amend -m "Thông báo cam kết mới"`

Hoặc để sử dụng thông báo cam kết trước đó mà không thay đổi nó:

`cam kết git --amend --no-chỉnh sửa`

Sửa đổi cập nhật ngày cam kết nhưng không ảnh hưởng đến ngày tác giả. Bạn có thể yêu cầu git làm mới thông tin.

`cam kết git --amend --reset-tác giả`

Bạn cũng có thể thay đổi tác giả của cam kết bằng:

`git commit --amend --author "Tác giả mới <email@address.com>"`

Lưu ý: Xin lưu ý rằng việc sửa đổi cam kết gần đây nhất sẽ thay thế hoàn toàn cam kết đó và cam kết trước đó sẽ bị xóa khỏi lịch sử của nhánh.

Điều này cần được ghi nhớ khi làm việc với các kho công cộng và trên các nhánh với các cộng tác viên khác.

Điều này có nghĩa là nếu cam kết trước đó đã được đẩy, sau khi sửa đổi nó, bạn sẽ phải nhấn `--force`.

## Phần 10.4: Cam kết mà không cần mở trình soạn thảo

Git thường sẽ mở một trình soạn thảo (như `vim` hoặc `emacs`) khi bạn chạy `git commit`. Truyền tùy chọn `-m` để chỉ định một thông báo từ dòng lệnh:

```
git commit -m "Thông báo cam kết tại đây"
```

Thông điệp cam kết của bạn có thể đi qua nhiều dòng:

```
git commit -m "Cam kết thông báo 'dòng chủ đề' tại đây"
```

```
Mô tả chi tiết hơn sẽ được trình bày ở đây (sau một dòng trống)."
```

Ngoài ra, bạn có thể chuyển vào nhiều đối số `-m`:

```
git commit -m "Tóm tắt cam kết" -m "Mô tả chi tiết hơn ở đây"
```

Xem [Cách viết tin nhắn cam kết Git](#).

[Hướng dẫn về kiểu thông báo cam kết Git của Udacity](#)

## Mục 10.5: Cam kết thay đổi trực tiếp

Thông thường, bạn phải sử dụng `git add` hoặc `git rm` để thêm các thay đổi vào chỉ mục trước khi có thẻ `git commit` chúng. Chuyển tùy chọn `-a` hoặc `--all` để tự động thêm mọi thay đổi (vào các tệp được theo dõi) vào chỉ mục, bao gồm cả việc xóa:

```
cam kết git -a
```

Nếu bạn cũng muốn thêm một thông báo cam kết, bạn sẽ làm:

```
git commit -a -m "thông báo cam kết của bạn ở đây"
```

Ngoài ra, bạn có thể tham gia hai lá cờ:

```
git commit -am "thông báo cam kết của bạn ở đây"
```

Bạn không nhất thiết phải cam kết tất cả các tệp cùng một lúc. Bỏ qua cờ `-a` hoặc `--all` và chỉ định tệp nào bạn muốn cam kết trực tiếp:

```
git commit path/to/a/file -m "thông báo cam kết của bạn ở đây"
```

Để cam kết trực tiếp nhiều tệp cụ thể, bạn cũng có thể chỉ định một hoặc nhiều tệp, thư mục và mẫu:

```
git commit path/to/a/file path/to/a/folder/* path/to/b/file -m "thông báo cam kết của bạn ở đây"
```

## Phần 10.6: Chọn dòng nào sẽ được sắp xếp để cam kết

Giả sử bạn có nhiều thay đổi trong một hoặc nhiều tệp nhưng từ mỗi tệp bạn chỉ muốn thực hiện một số thay đổi, bạn có thể chọn những thay đổi mong muốn bằng cách sử dụng:

```
git thêm -p
```

hoặc

```
git thêm -p [tệp]
```

Mỗi thay đổi của bạn sẽ được hiển thị riêng lẻ và với mỗi thay đổi, bạn sẽ được nhắc chọn một trong các tùy chọn sau:

y - Vâng, thêm cái này vào

n - Không, đừng thêm cái này vào

d - Không, không thêm đoạn này hoặc bất kỳ đoạn nào còn lại vào tệp này.

Hữu ích nếu bạn đã thêm những gì bạn muốn và muốn bỏ qua phần còn lại.

s - Chia khói thành các khói nhỏ hơn, nếu có thể

e - Chính sửa thủ công hunk. Đây có lẽ là lựa chọn mạnh mẽ nhất.

Nó sẽ mở phần lớn trong trình soạn thảo văn bản và bạn có thể chỉnh sửa nó nếu cần.

Điều này sẽ sắp xếp các phần của tập tin bạn chọn. Sau đó, bạn có thể thực hiện tất cả các thay đổi theo giai đoạn như thế này:

```
git commit -m 'Thông điệp cam kết'
```

Những thay đổi chưa được sắp xếp hoặc cam kết sẽ vẫn xuất hiện trong các tệp đang làm việc của bạn và có thể được cam kết sau nếu được yêu cầu. Hoặc nếu những thay đổi còn lại là không mong muốn, chúng có thể bị loại bỏ bằng:

```
thiết lập lại git --hard
```

Ngoài việc chia nhỏ một thay đổi lớn thành những cam kết nhỏ hơn, cách tiếp cận này còn hữu ích để xem xét những gì bạn sắp cam kết. Bằng cách xác nhận riêng từng thay đổi, bạn có cơ hội kiểm tra những gì mình đã viết và có thể tránh vô tình dàn dựng mã không mong muốn, chẳng hạn như các câu lệnh println/logging.

## Phần 10.7: Tạo một cam kết trống

Nói chung, các cam kết trống (hoặc các cam kết có trạng thái giống hệt với cấp độ gốc) là một lỗi.

Tuy nhiên, khi thử nghiệm các hook xây dựng, hệ thống CI và các hệ thống khác kích hoạt một cam kết, sẽ rất hữu ích khi có thể dễ dàng tạo các cam kết mà không cần phải chỉnh sửa/chạm vào một tệp giả.

Cam kết --allow-empty sẽ bỏ qua việc kiểm tra.

```
git commit -m "Đây là một cam kết trống" --allow-empty
```

## Mục 10.8: Cam kết thay mặt người khác

Nếu ai đó đã viết mã mà bạn đang cam kết, bạn có thể ghi công cho họ bằng tùy chọn --author :

```
git commit -m "msg" --author "John Smith <johnsmith@example.com>"
```

Bạn cũng có thể cung cấp một mẫu mà Git sẽ sử dụng để tìm kiếm các tác giả trước đó:

```
git commit -m "msg" --author "John"
```

Trong trường hợp này, thông tin tác giả từ cam kết gần đây nhất với tác giả có chứa "John" sẽ được sử dụng.

Trên GitHub, các cam kết được thực hiện theo một trong những cách trên sẽ hiển thị hình thu nhỏ của tác giả lớn, với hình thu nhỏ của người gửi và ở phía trước:

Commits on Apr 17, 2015



**Further improvements to soundness proof for addition.**

RenuAB committed with gebn on 23 Feb 2015

## Mục 10.9: Cam kết ký GPG

1. Xác định ID khóa của bạn

```
gpg --list-secret-keys --keyid-format DÀI
```

```
/Người dùng/davidcondrey/.gnupg/secring.gpg
```

```
-----  
giây 2048R/YOUR-16-DIGIT-KEY-ID YYYY-MM-DD [hết hạn: YYYY-MM-DD]
```

ID của bạn là mã gồm 16 chữ số theo sau dấu gạch chéo lên đầu tiên.

2. Xác định ID khóa của bạn trong cấu hình git của bạn

```
git config --global user.signingkey YOUR-16-DIGIT-KEY-ID
```

3. Kể từ phiên bản 1.7.9, git commit chấp nhận tùy chọn -S để đính kèm chữ ký vào các cam kết của bạn. Sử dụng tùy chọn này sẽ nhắc nhập cụm mật khẩu GPG của bạn và sẽ thêm chữ ký của bạn vào nhật ký cam kết.

```
git commit -S -m "Thông điệp cam kết của bạn"
```

## Mục 10.10: Cam kết thay đổi trong các tệp cụ thể

Bạn có thể cam kết các thay đổi được thực hiện đối với các tệp cụ thể và bỏ qua việc sắp xếp chúng bằng `git add`:

```
git cam kết file1.c file2.h
```

Hoặc trước tiên bạn có thể sắp xếp các tệp:

```
git thêm file1.c file2.h
```

và cam kết chúng sau:

```
cam kết git
```

## Mục 10.11: Cam kết vào một ngày cụ thể

```
git commit -m 'Sửa lỗi giao diện người dùng' --date 2016-07-01
```

Tham số `--date` đặt ngày tác giả. Ví dụ: ngày này sẽ xuất hiện trong đầu ra tiêu chuẩn của `nhật ký git`.

Để buộc ngày cam kết quá:

```
GIT_COMMITTER_DATE=2016-07-01 git commit -m 'Sửa lỗi giao diện người dùng' --date 2016-07-01
```

Tham số ngày chấp nhận các định dạng linh hoạt được hỗ trợ bởi ngày GNU, ví dụ:

```
git commit -m 'Sửa lỗi giao diện người dùng' --date hôm qua  
git commit -m 'Sửa lỗi giao diện người dùng' --date '3 ngày  
trước' git commit -m 'Sửa lỗi giao diện người dùng' --date '3 giờ trước'
```

Khi ngày không chỉ định thời gian, thời gian hiện tại sẽ được sử dụng và chỉ ngày sẽ bị ghi đè.

## Mục 10.12: Sửa đổi thời gian cam kết

Bạn cam sửa đổi thời gian của một cam kết bằng cách sử dụng

```
git commit --amend --date="Thứ năm ngày 28 tháng 7 11:30 2016 -0400"
```

hoặc thậm chí

```
git cam kết --amend --date="now"
```

## Mục 10.13: Sửa đổi tác giả của một cam kết

Nếu bạn thực hiện cam kết là tác giả sai, bạn có thể thay đổi nó và sau đó sửa đổi

```
git config user.name "Tên đầy đủ" git config  
user.email "email@example.com"  
  
cam kết git --amend --reset-tác giả
```

# Chương 11: Bí danh

## Mục 11.1: Bí danh đơn giản

Có hai cách tạo bí danh trong Git:

- với tệp `~/.gitconfig` :

```
[bí
danh] ci = cam kết
st = trạng thái
co = thanh toán
```

- với dòng lệnh:

```
git config --global alias.ci "cam kết" git config
--global alias.st "trạng thái" git config --
global alias.co "kiểm tra"
```

Sau khi bí danh được tạo - gõ:

- `git ci` thay vì `git commit`, `git st`
- thay vì `git status`, `git co` thay
- vì `git kiểm tra`.

Giống như các lệnh git thông thường, bí danh có thể được sử dụng bên cạnh các đối số. Ví dụ:

```
git ci -m "Thông báo cam kết..." git
co -b feature-42
```

## Mục 11.2: Danh sách/tìm kiếm bí danh hiện có

Bạn có thể [liệt kê các bí danh git hiện có](#) sử dụng `--get-regexp`:

```
$ git config --get-regexp '^alias\.'
```

Tìm kiếm bí danh

Để [tìm kiếm bí danh](#), thêm phần sau vào `.gitconfig` của bạn trong `[bí danh]`:

```
bí danh = !git config --list | grep ^bí danh\|. | cắt -c 7- | grep -Ei --color \'$1\' "#"
```

Sau đó bạn có thể:

- bí danh `git` - hiển thị TẤT CẢ bí
- danh `git` bí danh cam kết - chỉ các bí danh có chứa "cam kết"

## Mục 11.3: Bí danh nâng cao

Git cho phép bạn sử dụng các lệnh không phải git và `sh` đầy đủ cú pháp shell trong bí danh của bạn nếu bạn thêm tiền tố `!`.

Trong tệp `~/.gitconfig` của bạn :

```
[bí danh]
```

```
temp = !git add -A && git commit -m "Temp"
```

Thực tế là cú pháp shell đầy đủ có sẵn trong các bí danh có tiền tố này cũng có nghĩa là bạn có thể sử dụng các hàm shell để tạo các bí danh phức tạp hơn, chẳng hạn như các bí danh sử dụng đối số dòng lệnh:

```
[bí
danh] bỏ qua = "!f() { echo $1 >> .gitignore; }; f"
```

Bí danh ở trên xác định hàm f , sau đó chạy nó với bất kỳ đối số nào bạn chuyển cho bí danh. Vì vậy, việc chạy git bỏ qua .tmp/ sẽ thêm .tmp/ vào tệp .gitignore của bạn .

Trên thực tế, mẫu này hữu ích đến mức Git định nghĩa các biến \$1, \$2, v.v. cho bạn, do đó bạn thậm chí không cần phải xác định một hàm đặc biệt cho nó. (Nhưng hãy nhớ rằng dù sao thì Git cũng sẽ nối thêm các đối số, ngay cả khi bạn truy cập nó thông qua các biến này, vì vậy bạn có thể muốn thêm một lệnh giả vào cuối.)

Lưu ý rằng các bí danh có tiền tố ! theo cách này sẽ được chạy từ thư mục gốc của thanh toán git của bạn, ngay cả khi thư mục hiện tại của bạn nằm sâu hơn trong cây. Đây có thể là một cách hữu ích để chạy lệnh từ thư mục gốc mà không cần phải cd vào đó một cách rõ ràng.

```
[bí
danh] bỏ qua = "! echo $1 >> .gitignore"
```

## Phần 11.4: Tạm thời bỏ qua các tệp được theo dõi

Để tạm thời đánh dấu một tệp là bị bỏ qua (chuyển tệp dưới dạng tham số cho bí danh) - gõ:

```
unwatch = update-index --assume-unchanged
```

Để bắt đầu theo dõi lại tập tin - gõ:

```
xem = chỉ mục cập nhật --no-giả sử không thay đổi
```

Để liệt kê tất cả các tệp đã tạm thời bị bỏ qua - hãy gõ:

```
unwatched = "git ls-files -v | grep '^[:low:]'"
```

Để xóa danh sách chưa xem - gõ:

```
watchall = "git chưa được xem | xargs -L 1 -I % sh -c 'git watch `echo % | cut -c 2-`'"
```

Ví dụ về việc sử dụng bí danh:

```
git unwatch my_file.txt git
xem my_file.txt git chưa
xem git watchall
```

## Mục 11.5: Hiển thị log đẹp với biểu đồ nhánh

```
[bí
danh] log=log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short

lg = log --graph --date-order --first-parent \
    pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)( %quảng cáo) %C(in đậm)
```

```

lục lam)<%an>%Creset'
lgb = log --graph --date-order --branches --first-parent \
    --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(màu lục lam
đậm)<%an>%Creset'
lga = log --graph --date-order --all \
    --pretty=format:'%C(auto)%h%Creset %C(auto)%d%Creset %s %C(green)(%ad) %C(màu lục lam
đậm)<%an>%Creset'

```

Dưới đây là phần giải thích về các tùy chọn và phần giữ chỗ được sử dụng ở định dạng `--pretty` (danh sách đầy đủ có sẵn với `git help log`)

`--graph` - vẽ cây cam kết

`--date-order` - sử dụng thứ tự dấu thời gian cam kết khi có thể

`--first-parent` - chỉ theo dõi cha mẹ đầu tiên trên nút hợp nhất.

`--branches` - hiển thị tất cả các nhánh cục bộ (theo mặc định, chỉ nhánh hiện tại được hiển thị)

`--all` - hiển thị tất cả các nhánh cục bộ và từ xa

`%h` - giá trị băm cho cam kết (viết tắt)

`%ad` - Đầu ngày tháng (tác giả)

`%an` - Tên người dùng của tác giả

`%an` - Xác nhận tên người dùng

`%C(auto)` - để sử dụng các màu được xác định trong phần [color]

`%Creset` - để đặt lại màu

`%d` - --trang trí (tên nhánh và thẻ)

`%s` - thông báo cam kết

`%ad` - ngày tác giả (sẽ tuân theo chỉ thị `--date`) (và không phải ngày của người chuyển giao)

`%an` - tên tác giả (có thể là `%cn` cho tên người cam kết)

## Phần 11.6: Xem tệp nào đang bị cấu hình `.gitignore` của bạn bỏ qua

[bí danh]

```

bỏ qua = ! git ls-files --others --ignored --exclude-standard --directory \ && git ls-files --
          others -i --exclude-standard

```

Hiển thị một dòng trên mỗi tệp, do đó bạn có thể grep (chỉ thư mục):

```

$ git bị bỏ qua | grep '/'
$' .yardoc/ doc/

```

Hoặc đếm:

```
~$ git bị bỏ qua | wc -l # ôi,  
199811 thư mục chính của tôi ngày càng đông rồi
```

## Phần 11.7: Cập nhật mã trong khi vẫn giữ lịch sử tuyến tính

Đôi khi bạn cần lưu giữ lịch sử tuyến tính (không phân nhánh) của các lần xác nhận mã của mình. Nếu bạn đang làm việc trên một nhánh trong một thời gian, điều này có thể khó khăn nếu bạn phải thực hiện thao tác kéo git thông thường vì điều đó sẽ ghi lại sự hợp nhất với ngược dòng.

[bí danh]

lên = kéo -rebase

Điều này sẽ cập nhật với nguồn ngược dòng của bạn, sau đó áp dụng lại bất kỳ công việc nào bạn chưa đẩy lên trên bất cứ thứ gì bạn đã kéo xuống.

Để sử dụng:

đứng lên

## Phần 11.8: Tệp được phân loại không theo giai đoạn

Thông thường, để xóa các tệp được sắp xếp để cam kết bằng cách sử dụng cam kết `git reset`, `reset` có rất nhiều chức năng tùy thuộc vào các đối số được cung cấp cho nó. Để loại bỏ hoàn toàn tất cả các tệp được dàn dựng, chúng ta có thể sử dụng bí danh `git` để tạo bí danh mới sử dụng `reset` nhưng bây giờ chúng ta không cần phải nhớ cung cấp các đối số chính xác cho cài lại.

cấu hình `git --global alias.unstage "đặt lại --"`

Bây giờ, bất cứ khi nào bạn muốn hủy phần vùng các tệp giai đoạn, hãy nhập `git unstage` và bạn đã sẵn sàng.

## Chương 12: Nỗi loạn

Tham số	Chi tiết
--Tiếp tục	Khởi động lại quá trình khởi động lại sau khi giải quyết xung đột hợp nhất.
-Huỷ bỏ	Hủy bỏ thao tác rebase và đặt lại HEAD về nhánh ban đầu. Nếu nhánh được cung cấp khi hoạt động rebase được bắt đầu thì HEAD sẽ được đặt lại thành nhánh. Nếu không thì HEAD sẽ được đặt lại về vị trí ban đầu khi hoạt động rebase được bắt đầu.
--giữ trống --bỏ qua	Giữ các cam kết không thay đổi bất cứ điều gì từ cha mẹ của nó trong kết quả.
-m, --merge	Sử dụng các chiến lược hợp nhất để nỗi loạn. Khi chiến lược hợp nhất đệ quy (mặc định) được sử dụng, điều này cho phép rebase nhận biết được việc đổi tên ở phía thượng nguồn. Lưu ý rằng hợp nhất rebase hoạt động bằng cách phát lại từng cam kết từ nhánh đang hoạt động trên đầu nhánh ngược dòng. Vì lý do này, khi xung đột hợp nhất xảy ra, bên được báo cáo là của chúng tôi là chuỗi được cải tiến cho đến nay, bắt đầu từ ngược dòng và bên của họ là nhánh đang hoạt động. Nói cách khác, các bên được hoán đổi.
--stat	Hiển thị thông số khác biệt về những gì đã thay đổi ngược dòng kể từ lần rebase cuối cùng. Diffstat cũng được điều khiển bởi tùy chọn cấu hình rebase.stat.

Lệnh `-x`, `--exec` Thực hiện rebase tương tác, dừng giữa mỗi lần xác nhận và lệnh thực thi

### Mục 12.1: Tái cơ cấu chi nhánh địa phương

Khởi động lại áp dụng áp dụng lại một loạt các cam kết chồng lên một cam kết khác.

Để rebase một nhánh, hãy kiểm tra nhánh đó rồi rebase nó lên trên một nhánh khác.

chủ đề `kiểm tra git git`

```
rebase master # rebase nhánh hiện tại lên nhánh chính
```

Điều này sẽ gây ra:

Chủ đề A---B---C

/

D---E---F---G chủ nhân

Để biến thành:

Chủ đề A'---B'---C'

/

D---E---F---G chủ nhân

Các thao tác này có thể được kết hợp thành một lệnh duy nhất để kiểm tra nhánh và ngay lập tức khởi động lại nhánh đó:

```
git rebase master topic # rebase nhánh chủ đề vào nhánh chính
```

Quan trọng: Sau khi rebase, các cam kết được áp dụng sẽ có hàm băm khác. Bạn không nên rebase các cam kết mà bạn đã đẩy tới một máy chủ từ xa. Hậu quả có thể là không có khả năng `git push` nhánh khởi động lại cục bộ của bạn tới một máy chủ từ xa, để lại lựa chọn duy nhất của bạn là `git push --force`.

### Phần 12.2: Rebase: của chúng ta và của họ, địa phương và từ xa

Một rebase chuyển đổi ý nghĩa của "của chúng tôi" và "của họ":

```
chủ đề thanh toán git
git rebase chủ # rebase nhánh chủ đề trên nhánh chính
```

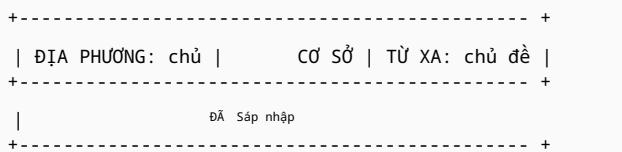
Bất cứ điều gì HEAD trả tới đều là "của chúng tôi"

Điều đầu tiên mà rebase thực hiện là đặt lại HEAD thành master; trước khi cam kết hái anh đào từ nhánh cũ chủ đề sang chủ đề mới (mỗi cam kết trong nhánh chủ đề cũ sẽ được viết lại và sẽ được xác định bởi một chủ đề khác hàm băm).

Đối với các thuật ngữ được sử dụng bởi các công cụ hợp nhất (không được nhầm lẫn với [ref cục bộ hoặc ref từ xa](#))

```
=> địa phương là chủ ("của chúng tôi"),
=> điều khiển từ xa là chủ đề ("của họ")
```

Điều đó có nghĩa là một công cụ hợp nhất/khác biệt sẽ hiển thị nhánh ngược dòng dưới dạng [cục bộ](#) (chính: nhánh mà bạn ở trên đó đang khởi động lại) và nhánh đang hoạt động ở chế độ từ xa (chủ đề: nhánh đang được khởi động lại)



Minh họa đảo ngược

Khi hợp nhất:

```
c--c--x--x--x(*) <- chủ đề nhánh hiện tại ('*'=HEAD)
 \
 \
 \--y--y--y <- nhánh khác cần hợp nhất
```

Chúng tôi không thay đổi chủ đề nhánh hiện tại, vì vậy những gì chúng tôi có vẫn là những gì chúng tôi đang làm (và chúng tôi hợp nhất từ nhánh khác)

```
c--c--x--x--x-----o(*) MERGE, vẫn thuộc chủ đề nhánh
      ^          /
      |
      /          /
      /          /
      \ \ \ -y--y--y--/  
của họ
```

Tên một cuộc nổi loạn:

Nhưng trong một cuộc rebase, chúng tôi đổi bên vì điều đầu tiên một cuộc rebase làm là kiểm tra nhánh ngược dòng để phát lại các cam kết hiện tại trên đó!

```
c--c--x--x--x(*) <- chủ đề nhánh hiện tại ('*'=HEAD)
 \
 \
 \--y--y--y <- nhánh ngược dòng
```

Ngược dòng `git rebase` trước tiên sẽ đặt HEAD thành nhánh ngược dòng, do đó chuyển đổi 'của chúng ta' và 'của họ' so với nhánh làm việc "hiện tại" trước đó.

```
c--c--x--x--x <- nhánh "hiện tại" cũ, nhánh "của họ" mới \ \ \ \ \--y--y--y(*) <- đặt HEAD
```

cho cam kết này, để phát lại x ở trên đó đây sẽ là "của chúng ta" mới

| Thượng nguồn

Rebase sau đó sẽ phát lại các cam kết 'của họ' trên nhánh chủ đề 'của chúng tôi' mới :

```
c--c--x--x--x <- cam kết "của họ" cũ, giờ là "bóng ma", có sẵn thông qua "reflogs" \ \ \ \ \--y--y--y--x--x--x(*) <- chủ đề khi tắt
```

cả x được phát lại, trả chủ đề nhánh vào cam kết này

| nhánh thượng nguồn

### Phần 12.3: Rebase tương tác

Ví dụ này nhằm mục đích mô tả cách người ta có thể sử dụng `git rebase` trong chế độ tương tác. Người ta mong đợi người ta có hiểu biết cơ bản về `git rebase` là gì và nó làm gì.

Rebase tương tác được bắt đầu bằng lệnh sau:

```
git rebase -i
```

Tùy chọn `-i` để cập đến chế độ tương tác. Bằng cách sử dụng rebase tương tác, người dùng có thể thay đổi các thông báo cam kết, cũng như sắp xếp lại, phân tách và/hoặc nén (kết hợp thành một) các cam kết.

Giả sử bạn muốn sắp xếp lại ba lần cam kết cuối cùng của mình. Để làm điều này bạn có thể chạy:

```
git rebase -i Đầu~3
```

Sau khi thực hiện hướng dẫn trên, một tệp sẽ được mở trong trình soạn thảo văn bản của bạn, nơi bạn có thể chọn cách thực hiện lại các cam kết của mình. Với mục đích của ví dụ này, chỉ cần thay đổi thứ tự các cam kết của bạn, lưu tệp và đóng trình chỉnh sửa. Điều này sẽ bắt đầu một đợt rebase theo thứ tự bạn đã áp dụng. Nếu bạn kiểm tra `nhật ký git`, bạn sẽ thấy các cam kết của mình theo thứ tự mới mà bạn đã chỉ định.

Viết lại thông điệp cam kết

Bây giờ, bạn đã quyết định rằng một trong những thông báo cam kết là mơ hồ và bạn muốn nó mang tính mô tả hơn. Hãy xem xét ba lần xác nhận cuối cùng bằng cách sử dụng cùng một lệnh.

```
git rebase -i Đầu~3
```

Thay vì sắp xếp lại thứ tự, các cam kết sẽ được khởi động lại, lần này chúng tôi sẽ thay đổi lựa chọn, mặc định, để diễn đạt lại cam kết mà bạn muốn thay đổi thông báo.

Khi bạn đóng trình chỉnh sửa, quá trình rebase sẽ bắt đầu và nó sẽ dừng ở thông báo cam kết cụ thể mà bạn muốn

viết lại. Điều này sẽ cho phép bạn thay đổi thông điệp cam kết thành bất cứ điều gì bạn muốn. Sau khi bạn đã thay đổi thông báo, chỉ cần đóng trình chỉnh sửa để tiếp tục.

#### Thay đổi nội dung của một cam kết

Bên cạnh việc thay đổi thông báo cam kết, bạn cũng có thể điều chỉnh những thay đổi được thực hiện bởi cam kết. Để làm như vậy chỉ cần thay đổi chọn thành chỉnh sửa cho một lần xác nhận. Git sẽ dừng khi đến cam kết đó và cung cấp các thay đổi ban đầu của cam kết trong khu vực tổ chức. Bây giờ bạn có thể điều chỉnh những thay đổi đó bằng cách hủy giai đoạn chúng hoặc thêm các thay đổi mới.

Ngay khi khu vực tổ chức chứa tất cả các thay đổi bạn muốn trong cam kết đó, hãy thực hiện các thay đổi đó. Thông báo cam kết cũ sẽ được hiển thị và có thể được điều chỉnh để phản ánh cam kết mới.

#### Chia một cam kết thành nhiều cam kết

Giả sử bạn đã thực hiện một cam kết nhưng sau đó lại quyết định rằng cam kết này có thể được chia thành hai hoặc nhiều cam kết.

Sử dụng lệnh tương tự như trước, thay thế chọn bằng chỉnh sửa và nhấn enter.

Bây giờ, git sẽ dừng ở cam kết mà bạn đã đánh dấu để chỉnh sửa và đặt tất cả nội dung của nó vào khu vực tổ chức. Từ thời điểm đó, bạn có thể chạy `git reset HEAD^` để đặt cam kết vào thư mục làm việc của mình. Sau đó, bạn có thể thêm và chuyển giao các tệp của mình theo một trình tự khác - thay vào đó, cuối cùng hãy chia một lần xác nhận thành n lần xác nhận.

#### Nén nhiều cam kết thành một

Giả sử bạn đã thực hiện một số công việc và có nhiều lần xác nhận mà bạn nghĩ có thể chỉ là một lần xác nhận duy nhất. Để làm được điều đó, bạn có thể thực hiện `git rebase -i HEAD~3`, thay thế 3 bằng số lượng cam kết thích hợp.

Lần này thay thế pick bằng bí thay thế. Trong quá trình rebase, cam kết mà bạn đã hướng dẫn xóa sẽ bị đè lên trên cam kết trước đó; thay vào đó hãy biến chúng thành một cam kết duy nhất.

### Mục 12.4: Rebase trở lại cam kết ban đầu

Kể từ Git [1.7.12](#) có thể rebase xuống cam kết gốc. Cam kết gốc là cam kết đầu tiên từng được thực hiện trong kho lưu trữ và thông thường không thể chỉnh sửa được. Sử dụng lệnh sau:

```
git rebase -i --root
```

### Phần 12.5: Định cấu hình autostash

Autostash là một tùy chọn cấu hình rất hữu ích khi sử dụng rebase cho những thay đổi cục bộ. Thông thường, bạn có thể cần đưa ra các cam kết từ nhánh thượng nguồn, nhưng vẫn chưa sẵn sàng để cam kết.

Tuy nhiên, Git không cho phép khởi động rebase nếu thư mục làm việc không sạch. Autostash để giải cứu:

<code>git config --global rebase.autostash git rebase @{u}</code>	# cấu hình một lần # ví dụ rebase trên nhánh ngược dòng
---	--

Autostash sẽ được áp dụng bất cứ khi nào quá trình rebase kết thúc. Việc rebase có kết thúc thành công hay không hoặc nó bị hủy bỏ không thành vấn đề. Dù bằng cách nào, autostash sẽ được áp dụng. Nếu rebase thành công và do đó cam kết cơ sở đã thay đổi, thì có thể có xung đột giữa autostash và cam kết mới. Trong trường hợp này, bạn sẽ phải giải quyết xung đột trước khi cam kết. Điều này không khác gì so với việc bạn lưu trữ công rồi áp dụng, do đó, không có nhược điểm nào khi thực hiện việc đó một cách tự động.

## Phần 12.6: Kiểm tra tất cả các cam kết trong quá trình rebase

Trước khi thực hiện một yêu cầu kéo, điều hữu ích là phải đảm bảo rằng quá trình biên dịch thành công và các bài kiểm tra đang được thực hiện cho mỗi lần xác nhận trong nhánh. Chúng ta có thể làm điều đó tự động bằng tham số `-x`.

Ví dụ:

```
git rebase -i -x làm
```

sẽ thực hiện rebase tương tác và dừng sau mỗi lần xác nhận để thực hiện thực hiện. Trong trường hợp **thực hiện** không thành công, git sẽ dừng lại để cho bạn cơ hội khắc phục sự cố và sửa đổi cam kết trước khi tiếp tục chọn cái tiếp theo.

## Mục 12.7: Khởi động lại trước khi xem xét mã

Bản tóm tắt

Mục tiêu này là sắp xếp lại tất cả các cam kết rải rác của bạn thành các cam kết có ý nghĩa hơn để đánh giá mã dễ dàng hơn. Nếu có quá nhiều lớp thay đổi trên quá nhiều tệp cùng một lúc thì việc đánh giá mã sẽ khó hơn. Nếu bạn có thể sắp xếp lại các cam kết được tạo theo trình tự thời gian thành các cam kết theo chủ đề thì quá trình xem xét mã sẽ dễ dàng hơn (và có thể sẽ có ít lỗi hơn xảy ra trong quá trình xem xét mã).

Ví dụ quá đơn giản hóa này không phải là chiến lược duy nhất để sử dụng git để đánh giá mã tốt hơn. Đó là cách tôi làm và đó là điều truyền cảm hứng cho những người khác xem xét cách thực hiện việc đánh giá mã và lịch sử git dễ dàng hơn/tốt hơn.

Điều này cũng thể hiện một cách sự phạm súc mạnh của rebase nói chung.

Ví dụ này giả sử bạn biết về việc khởi động lại tương tác.

Giả định:

- bạn đang làm việc trên một nhánh tính năng của master, tính năng của bạn có ba lớp chính: front-end, back-end, DB bạn đã thực hiện
- rất nhiều cam kết khi làm việc trên một nhánh tính năng. Mỗi cam kết chạm vào nhiều lớp tại một lần
- bạn muốn (cuối cùng) chỉ có ba lần xác nhận trong nhánh của bạn, một
  - lần chưa tất cả các thay đổi ở giao diện người dùng, một lần chưa tất cả các thay đổi ở phần sau, một lần chưa tất cả các thay đổi DB

Chiến lược:

- chúng ta sẽ thay đổi các cam kết theo trình tự thời gian thành các cam kết "theo chủ đề".
- đầu tiên, chia tất cả các cam kết thành nhiều cam kết nhỏ hơn -- mỗi cam kết chỉ chứa một chủ đề tại một thời điểm (trong ví dụ của chúng tôi, các chủ đề là giao diện người dùng, mặt sau, thay đổi DB)
- Sau đó sắp xếp lại các cam kết theo chủ đề của chúng ta và 'ép' chúng thành các cam kết theo chủ đề duy nhất

Ví dụ:

```
$ git log --oneline master..  
975430b Việc thêm db hoạt động: db.sql logic.rb  
3702650 cố gắng cho phép thêm các mục việc cần làm: page.html logic.rb  
43b075a bắn nháp đầu tiên: page.html và db.sql $  
git rebase -i bậc thầy
```

Điều này sẽ được hiển thị trong trình soạn thảo văn bản:

```
chọn 43b075a bản nháp đầu tiên: page.html và db.sql chọn  
3702650 có gắng cho phép thêm các mục việc cần làm: page.html logic.rb chọn 975430b db  
việc thêm hoạt động: db.sql logic.rb
```

Thay đổi nó thành thế này:

```
e 43b075a bản nháp đầu tiên: page.html và db.sql e  
3702650 cố gắng cho phép thêm các mục việc cần làm: page.html logic.rb e 975430b  
Việc thêm db hoạt động: db.sql logic.rb
```

Sau đó git sẽ áp dụng một cam kết tại một thời điểm. Sau mỗi lần xác nhận, nó sẽ hiển thị lời nhắc và sau đó bạn có thể thực hiện các thao tác sau:

```
Đã dừng ở 43b075a92a952faf999e76c4e4d7fa0f44576579... bản nháp đầu tiên: page.html và db.sql Bạn có thể sửa đổi cam kết ngay bây giờ, với
```

```
cam kết git --amend
```

Khi bạn hài lòng với những thay đổi của mình, hãy chạy

```
git rebase -tiếp tục
```

```
$ git status  
đang trong quá trình rebase ; lên 4975ae9  
Bạn hiện đang chỉnh sửa một cam kết trong khi khởi động lại 'tính năng' nhánh trên '4975ae9'.  
(sử dụng "git commit --amend" để sửa đổi cam kết hiện tại) (sử dụng  
"git rebase --continue" khi bạn hài lòng với những thay đổi của mình)
```

```
không có gì để cam kết, làm sạch thư mục làm việc $ git  
reset HEAD^ #Đây 'không cam kết' tất cả các thay đổi trong cam kết này. $ git status  
-s M db.sql M  
page.html  
$ git add  
db.sql #Đây giờ chúng ta sẽ tạo các cam kết theo chủ đề nhỏ hơn $ git commit -m  
"first Draft: db.sql" $ git add page.html $ git  
commit -m "bản nháp đầu  
tiên: page.html" $ git rebase --continue
```

Sau đó, bạn sẽ lặp lại các bước đó cho mỗi lần xác nhận. Cuối cùng, bạn có điều này:

```
$ git log --oneline  
0309336 db công việc thêm: logic.rb 06f81c9  
db công việc thêm: db.sql 3264de2 thêm  
các mục việc cần làm: page.html 675a02b thêm  
các mục việc cần làm: logic.rb 272c674 bản  
nháp đầu tiên: page.html 08c275d bản  
nháp đầu tiên: db .sql
```

Bây giờ chúng ta chạy rebase một lần nữa để sắp xếp lại và sắp xếp lại:

```
$ git rebase -i master
```

Điều này sẽ được hiển thị trong trình soạn thảo văn bản:

```
chọn 08c275d bản nháp đầu tiên: db.sql  
chọn 272c674 bản nháp đầu tiên: page.html chọn  
675a02b thêm các mục việc cần làm: logic.rb
```

```
chọn 3264de2 thêm các mục việc cần làm: page.html
06f81c9 db việc thêm các công việc: db.sql
0309336 db việc thêm các công việc: logic.rb
```

Thay đổi nó thành thế này:

```
chọn 08c275d bản nháp đầu tiên: db.sql s
06f81c9 công việc thêm db: db.sql chọn 675a02b
thêm các mục việc cần làm: logic.rb s 0309336 việc thêm
db: logic.rb chọn 272c674 bản nháp đầu tiên:
page.html s 3264de2 thêm các mục việc cần làm:
trang .html
```

LƯU Ý: hãy đảm bảo rằng bạn yêu cầu git rebase áp dụng/bỏ qua các cam kết theo chủ đề nhỏ hơn theo thứ tự chúng được cam kết theo trình tự thời gian. Nếu không, bạn có thể gặp phải những xung đột hợp nhất sai, không cần thiết cần giải quyết.

Khi quá trình rebase tương tác đó hoàn tất, bạn sẽ nhận được điều này:

```
$ git log --oneline master.. 74bdd5f
thêm todos: Lớp GUI e8d8f7e thêm todos: lớp
logic nghiệp vụ 121c578 thêm todos: lớp DB
```

Tóm tắt lại

Bây giờ bạn đã chuyển đổi các cam kết theo trình tự thời gian của mình thành các cam kết theo chủ đề. Trong cuộc sống thực, bạn có thể không cần phải làm điều này mọi lúc, nhưng khi bạn muốn hoặc cần làm điều này thì bây giờ bạn có thể. Ngoài ra, hy vọng bạn đã tìm hiểu thêm về git rebase.

## Mục 12.8: Hủy bỏ một cuộc nỗi loạn tương tác

Bạn đã bắt đầu một cuộc nỗi dậy tương tác. Trong trình chỉnh sửa nơi bạn chọn các cam kết của mình, bạn quyết định rằng có điều gì đó không ổn (ví dụ: thiếu một cam kết hoặc bạn đã chọn sai đích rebase) và bạn muốn hủy bỏ rebase.

Để thực hiện việc này, chỉ cần xóa tất cả các cam kết và hành động (tức là tất cả các dòng không bắt đầu bằng dấu # ) và quá trình rebase sẽ bị hủy bỏ!

Văn bản trợ giúp trong trình chỉnh sửa thực sự cung cấp gợi ý này:

```
# Rebase 36d15de..612f2f7 lên 36d15de (3 lệnh)
#
# Lệnh: # p,
pick = dùng commit # r, reword
= dùng commit, nhưng sửa thông điệp commit # e, edit = dùng commit, nhưng
dùng sửa # s, bí = dùng commit, nhưng gộp vào commit trước đó
# f , fixup = like "squash", nhưng loại bỏ thông điệp tường trình của cam kết
này # x, exec = run command (phản còn lại của dòng) bằng shell # # Những dòng này có thể
được sắp xếp lại; chúng được thực hiện từ trên xuống dưới. #
```

```
# Nếu bạn xóa một dòng ở đây RÀNG CAM KẾT SẼ BỊ MẤT. # # Tuy nhiên, nếu
```

```
bạn xóa mọi thứ, quá trình rebase sẽ bị hủy bỏ. # # Lưu ý rằng các cam kết trống sẽ được
nhận xét
```

## Phần 12.9: Thiết lập git-pull để tự động thực hiện rebase thay vì hợp nhất

Nếu nhóm của bạn đang tuân theo quy trình làm việc dựa trên rebase, thì việc thiết lập git để mỗi nhánh mới được tạo sẽ thực hiện thao tác rebase, thay vì thao tác hợp nhất, trong quá trình kéo git có thể là một điều thuận lợi.

Để thiết lập mọi nhánh mới tự động khởi động lại, hãy thêm phần sau vào `.gitconfig` hoặc `.git/config` của bạn:

```
[nhánh]
autosetuprebase = luôn luôn
```

Dòng lệnh: `git config [--global] Branch.autosetuprebase luôn`

Ngoài ra, bạn có thể thiết lập lệnh `git pull` để luôn hoạt động như thể tùy chọn `--rebase` đã được thông qua:

```
[kéo]
rebase = đúng
```

Dòng lệnh: `git config [--global] pull.rebase true`

## Mục 12.10: Đẩy lùi sau đợt rebase

Đôi khi bạn cần viết lại lịch sử bằng rebase, nhưng `git push` phàn nàn về việc làm như vậy vì bạn đã viết lại lịch sử.

Điều này có thể được giải quyết bằng `git push --force`, nhưng hãy xem xét `git push --force-with-lease`, cho biết rằng bạn muốn quá trình đẩy không thành công nếu nhánh theo dõi từ xa cục bộ khác với nhánh trên điều khiển từ xa, ví dụ: ai đó người khác được đẩy đến điều khiển từ xa sau lần tìn nạp cuối cùng. Điều này tránh việc vô tình ghi đè lên cú đẩy gần đây của người khác.

Lưu ý: `git push --force` và thậm chí `--force-with-lease` đối với vấn đề đó - có thể là một lệnh nguy hiểm vì nó viết lại lịch sử của nhánh. Nếu một người khác đã kéo nhánh trước khi bị đẩy cưỡng bức, thao tác `git pull` hoặc `git get` của người đó sẽ có lỗi do lịch sử cục bộ và lịch sử từ xa bị phân kỳ. Điều này có thể khiến người đó gặp phải những sai sót không mong muốn. Khi xem xét đẩy đủ các bản ghi lại, công việc của người dùng khác có thể được phục hồi, nhưng điều đó có thể dẫn đến lãng phí rất nhiều thời gian. Nếu bạn buộc phải thực hiện push tới một nhánh với những người đóng góp khác, hãy cố gắng phối hợp với họ để họ không gặp phải lỗi.

# Chương 13: Cấu hình

Tham số	Chi tiết
--hệ thống	Chỉnh sửa tệp cấu hình trên toàn hệ thống, được sử dụng cho mọi người dùng (trên Linux, tệp này nằm ở \$(prefix)/etc/gitconfig)
--tổng cấu	Chỉnh sửa tệp cấu hình chung, được sử dụng cho kho lưu trữ mà bạn làm việc (trên Linux, tệp này nằm ở ~/.gitconfig)
--địa phương	Chỉnh sửa tệp cấu hình dành riêng cho kho lưu trữ, được đặt tại .git/config trong kho lưu trữ của bạn; Đây là thiết lập mặc định

## Phần 13.1: Cài đặt trình soạn thảo nào sẽ sử dụng

Có một số cách để thiết lập trình soạn thảo nào sẽ được sử dụng cho việc chuyển giao, khởi động lại, v.v.

- Thay đổi cài đặt cấu hình core.editor .

```
$ git config --global core.editor nano
```

- Đặt biến môi trường GIT\_EDITOR .

Đối với một lệnh:

```
$ GIT_EDITOR=cam kết nano git
```

Hoặc cho tất cả các lệnh chạy trong một thiết bị đầu cuối. Lưu ý: Điều này chỉ áp dụng cho đến khi bạn đóng thiết bị đầu cuối.

```
$ xuất GIT_EDITOR=nano
```

- Để thay đổi trình soạn thảo cho tất cả các chương trình đầu cuối, không chỉ Git, hãy đặt biến môi trường VISUAL hoặc EDITOR .  
(Xem [HÌNH ẢNH vs BIÊN TẬP](#).)

```
$ xuất EDITOR=nano
```

Lưu ý: Như trên, điều này chỉ áp dụng cho thiết bị đầu cuối hiện tại; Shell của bạn thường sẽ có một tệp cấu hình để cho phép bạn thiết lập nó vĩnh viễn. (Ví dụ : [trên bash](#), hãy thêm dòng trên vào ~/.bashrc hoặc ~/.bash\_profile.)

Một số trình soạn thảo văn bản (chủ yếu là GUI) sẽ chỉ chạy một phiên bản mỗi lần và thường thoát nếu bạn đã mở một phiên bản của chúng. Nếu trường hợp này xảy ra với trình soạn thảo văn bản của bạn, Git sẽ in thông báo Đang hủy bỏ cam kết do thông báo cam kết trống. mà không cho phép bạn chỉnh sửa thông báo cam kết trước. Nếu điều này xảy ra với bạn, hãy tham khảo tài liệu của trình soạn thảo văn bản để xem liệu nó có cờ --wait (hoặc tương tự) khiến nó tạm dừng cho đến khi tài liệu được đóng lại hay không.

## Mục 13.2: Tự động sửa lỗi chính tả

```
cấu hình git --global help.autocore 17
```

Điều này cho phép tự động sửa lỗi trong git và sẽ tha thứ cho những lỗi nhỏ của bạn (ví dụ: số liệu thống kê git thay vì trạng thái git). Tham số bạn cung cấp cho help.autocore xác định khoảng thời gian hệ thống sẽ đợi, tính bằng phần mười giây, trước khi tự động áp dụng lệnh tự động sửa. Trong lệnh trên, 17 có nghĩa là git

nên đợi 1,7 giây trước khi áp dụng lệnh tự động sửa.

Tuy nhiên, những lỗi lớn hơn sẽ bị coi là thiếu lệnh, vì vậy việc gõ một cái gì đó như `git testingit` sẽ dẫn đến testit không phải là lệnh `git`.

### Phần 13.3: Liệt kê và chỉnh sửa cấu hình hiện tại

Cấu hình Git cho phép bạn tùy chỉnh cách hoạt động của git. Nó thường được sử dụng để đặt tên và email của bạn hoặc trình soạn thảo yêu thích hoặc cách thực hiện việc hợp nhất.

Để xem cấu hình hiện tại.

```
$ cấu hình git --list
...
core.editor=vim
credential.helper=osxkeychain
...
```

Để chỉnh sửa cấu hình:

```
$ git config <key> <value> $ git
config core.ignorecase true
```

Nếu bạn muốn thay đổi này đúng với tất cả các kho lưu trữ của mình, hãy sử dụng `--global`

```
$ git config --global user.name "Tên của bạn" $ git
config --global user.email "Email của bạn" $ git config
--global core.editor vi
```

Bạn có thể liệt kê lại để xem những thay đổi của mình.

### Phần 13.4: Tên người dùng và địa chỉ email

Ngay sau khi cài đặt Git, điều đầu tiên bạn nên làm là đặt tên người dùng và địa chỉ email của mình. Từ một shell, gõ:

```
git config --global user.name "Mr. Bean" git
config --global user.email mrbean@example.com
```

- `git config` là lệnh để nhận hoặc đặt tùy chọn `--global` có
- nghĩa là tệp cấu hình dành riêng cho tài khoản người dùng của bạn sẽ được chỉnh sửa `user.name` và
- `user.email` là khóa cho các biến cấu hình; `user` là phần của tập tin cấu hình. `name` và `email` là tên của các biến.
- "`Mr. Bean`" và `mrbean@example.com` là các giá trị bạn đang lưu trữ trong hai biến. Lưu ý các trích dẫn xung quanh "`Mr. Bean`", là bắt buộc vì giá trị bạn đang lưu trữ có chứa khoảng trắng.

### Phần 13.5: Nhiều tên người dùng và địa chỉ email

Kể từ Git 2.13, nhiều tên người dùng và địa chỉ email có thể được định cấu hình bằng cách sử dụng bộ lọc thư mục.

Ví dụ cho Windows: `.gitconfig`

Chỉnh sửa: `git config --global -e`

Thêm vào:

```
[includeIf "gitdir:D:/work"]
    đường dẫn = .gitconfig-work.config

[includeIf "gitdir:D:/opensource/"]
    đường dẫn = .gitconfig-opensource.config
```

## Ghi chú

- Thứ tự tùy thuộc, ai khớp cuối cùng thì "thắng". cần có / ở cuối -
- ví dụ: " gitdir:D:/work" sẽ không hoạt động. tiền tố gitdir : là bắt buộc.
- 

.gitconfig-work.config

Tệp trong cùng thư mục với .gitconfig

## Tên

```
[người dùng]
Email tiền = work@somewhere.com
```

.gitconfig-opensource.config

Tệp trong cùng thư mục với .gitconfig

## Tên

```
[người dùng]
= Email đẹp = cool@opensource.stuff
```

Ví dụ cho Linux

```
[includeIf "gitdir:~/work/"] path
    = .gitconfig-work [includeIf
"gitdir:~/opensource/"] path = .gitconfig-
opensource
```

Nội dung file và ghi chú thuộc mục Windows.

## Phần 13.6: Nhiều cấu hình git

Bạn có tối đa 5 nguồn cho cấu hình git:

- 6 tập tin:
  - %ALLUSERSPROFILE%\Git\Config (chỉ dành cho Windows)
  - (hệ thống) <git>/etc/gitconfig, với <git> là đường dẫn cài đặt git. (trên Windows, đó là <git>\mingw64\etc\gitconfig) (system)
  - \$XDG\_CONFIG\_HOME/git/config (chỉ Linux/Mac) (tổng thể)
  - ~/.gitconfig (Windows: %USERPROFILE%\.gitconfig) (cục bộ) .git/
 config (trong git repo \$GIT\_DIR) một tệp chuyên
  - dụng (với `git config -f`), được sử dụng chặng hạn để sửa đổi cấu hình của môđun con: `git config -f .gitmodules ...`
- dùng lệnh với `git -c git -c core.autocrlf=false fetch` sẽ ghi đè bất kỳ core.autocrlf nào khác thành `false`, chỉ dành cho lệnh tìm nạp đó .

Thứ tự rất quan trọng: bất kỳ cấu hình nào được đặt trong một nguồn đều có thể bị ghi đè bởi một nguồn được liệt kê bên dưới nó.

`git config --system/global/local` là lệnh liệt kê 3 nguồn đó, nhưng chỉ `git config -l` mới liệt kê tất cả các cấu hình đã được giải quyết.  
 "Đã giải quyết" có  
 nghĩa là nó chỉ liệt kê giá trị cấu hình bị ghi đè cuối cùng.

Kể từ git 2.8, nếu bạn muốn xem cấu hình nào đến từ tệp nào, bạn gõ:

```
cấu hình git --list --show-origin
```

## Phần 13.7: Định cấu hình kết thúc dòng

Sự miêu tả

Khi làm việc với một nhóm sử dụng các hệ điều hành (HĐH) khác nhau trong dự án, đôi khi bạn có thể gặp rắc rối khi xử lý phần cuối dòng.

Microsoft Windows

Khi làm việc trên hệ điều hành (OS) Microsoft Windows, các kết thúc dòng thường có dạng - xuống dòng + cấp dòng (CR+LF). Việc mở một tệp đã được chỉnh sửa bằng máy Unix như Linux hoặc OSX có thể gây rắc rối, khiến văn bản dường như không có kết thúc dòng nào cả. Điều này là do thực tế là các hệ thống Unix chỉ áp dụng các kết thúc dòng khác nhau của nguồn cấp dữ liệu dòng biểu mẫu (LF).

Để khắc phục điều này, bạn có thể chạy theo hướng dẫn

```
cấu hình git --global core.autocrlf=true
```

Khi thanh toán, Hướng dẫn này sẽ đảm bảo các kết thúc dòng được định cấu hình phù hợp với Hệ điều hành Microsoft Windows (LF -> CR+LF)

Dựa trên Unix (Linux/OSX)

Tương tự, có thể xảy ra sự cố khi người dùng trên hệ điều hành Unix có đọc các tệp đã được chỉnh sửa trên hệ điều hành Microsoft Windows. Để ngăn chặn bất kỳ sự cố không mong muốn nào, hãy chạy

```
cấu hình git --global core.autocrlf=input
```

Khi cam kết, điều này sẽ thay đổi kết thúc dòng từ CR+LF -> +LF

## Mục 13.8: cấu hình chỉ cho một lệnh

bạn có thể sử dụng **-c <name>=<value>** để chỉ thêm cấu hình cho một lệnh.

Để cam kết với tư cách là người dùng khác mà không phải thay đổi cài đặt của bạn trong .gitconfig :

```
git -c user.email = mail@example commit -m "một số tin nhắn"
```

Lưu ý: trong ví dụ đó, bạn không cần phải nhập chính xác cả user.name và user.email, git sẽ hoàn thành thông tin còn thiếu từ các lần xác nhận trước đó.

## Phần 13.9: Thiết lập proxy

Nếu bạn đứng sau một proxy, bạn phải nói với git về nó:

```
cấu hình git --global http.proxy http://my.proxy.com:portnumber
```

Nếu bạn không còn sử dụng proxy nữa:

```
cấu hình git --global --unset http.proxy
```

# Chương 14: Phân nhánh

Tham số	Chi tiết	
-d, --xóa	Xóa một nhánh. Nhánh phải được hợp nhất hoàn toàn trong nhánh ngược dòng của nó hoặc trong HEAD nếu không có ngược dòng nào được đặt <code>--track</code> hoặc <code>--set-upstream</code>	
-D	Phím tắt cho <code>--delete --force</code>	
-m, --di chuyển	Di chuyển/đổi tên một nhánh và reflog tương ứng	
-M	Phím tắt cho <code>--move --force</code>	
-r, --remotes	Liệt kê hoặc xóa (nếu được sử dụng với -d) các nhánh theo dõi từ xa -a, --all	Liệt kê cả các nhánh theo dõi từ xa và các nhánh cục bộ Kích hoạt chế độ danh sách. <code>git Branch &lt;pattern&gt;</code> sẽ cố gắng tạo một nhánh, sử dụng <code>git Branch -- list &lt;pattern&gt;</code> để liệt kê các nhánh phù hợp Nếu nhánh được chỉ định chưa tồn tại hoặc nếu <code>--force</code> đã được cung cấp, hoạt động chính xác như <code>--track</code> .
thay đổi	Mặt khác, hãy thiết lập cấu hình như <code>--track</code> khi tạo nhánh, ngoại trừ việc <code>--set-upstream</code> trỏ đến nhánh không	

## Phần 14.1: Tạo và kiểm tra các nhánh mới

Để tạo một nhánh mới, trong khi vẫn ở trên nhánh hiện tại, hãy sử dụng:

```
nhánh git <tên>
```

Nói chung, tên chi nhánh không được chứa dấu cách và phải tuân theo các thông số kỹ thuật khác được liệt kê [ở đây](#). Để chuyển sang một chi nhánh hiện có:

```
kiểm tra git <tên>
```

Để tạo một nhánh mới và chuyển sang nhánh đó:

```
kiểm tra git -b <tên>
```

Để tạo một nhánh tại một điểm khác với lần nhận cuối cùng của nhánh hiện tại (còn được gọi là HEAD), hãy sử dụng một trong các lệnh sau:

```
git nhánh <tên> [<điểm bắt đầu>] git kiểm tra -b <tên> [<điểm bắt đầu>]
```

<start-point> có thể là bất kỳ [bản sửa đổi](#) nào được biết đến bởi git (ví dụ: tên nhánh khác, cam kết SHA hoặc tham chiếu tương trưng như HEAD hoặc tên thẻ):

```
git kiểm tra -b <tên> some_other_branch git kiểm tra -b <tên> af295 git kiểm tra -b <tên> HEAD~5 git kiểm tra -b <tên> v1.0.5
```

Để tạo một nhánh từ một nhánh từ xa ( `<remote_name>` mặc định là Origin):

```
git nhánh <name> <remote_name>/<branch_name> git kiểm tra -b <name> <remote_name>/<branch_name>
```

Nếu một tên nhánh nhất định chỉ được tìm thấy trên một điều khiển từ xa, bạn chỉ cần sử dụng

```
kiểm tra git -b <branch_name>
```

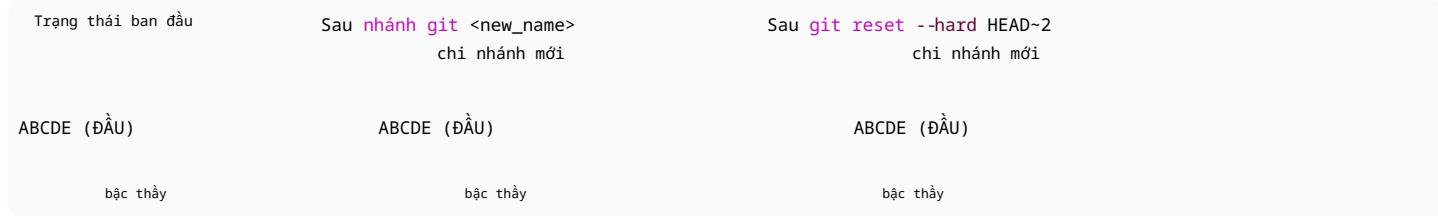
tương đương với

```
kiểm tra git -b <branch_name> <remote_name>/<branch_name>
```

Đôi khi bạn có thể cần chuyển một số cam kết gần đây của mình sang một nhánh mới. Điều này có thể đạt được bằng cách phân nhánh và "quay trở lại", như vậy:

```
nhánh git <new_name>
git reset --hard HEAD~2 # Quay lại 2 lần xác nhận, bạn sẽ mất công việc chưa được cam kết.
kiểm tra git <new_name>
```

Dưới đây là lời giải thích minh họa về kỹ thuật này:



## Mục 14.2: Niêm yết chi nhánh

Git cung cấp nhiều lệnh để liệt kê các nhánh. Tất cả các lệnh đều sử dụng chức năng của `nhánh git`, nó sẽ cung cấp danh sách các nhánh nhất định, tùy thuộc vào tùy chọn nào được đưa vào dòng lệnh. Git sẽ làm nếu có thể, cho biết nhánh hiện được chọn có dấu sao bên cạnh.

Mục tiêu	Yêu cầu
Liệt kê các chi nhánh địa phương	<code>nhánh git</code>
Liệt kê chi tiết các chi nhánh địa phương	<code>nhánh git -v</code>
Liệt kê các chi nhánh từ xa và địa phương	<code>nhánh git -a HOẶC nhánh git --all</code>
Liệt kê các nhánh từ xa và cục bộ (dài dòng) <code>git nhánh -av</code>	
Liệt kê các chi nhánh từ xa	<code>nhánh git -r</code>
Liệt kê các nhánh từ xa với cam kết mới nhất <code>git nhánh -rv</code>	
Liệt kê các nhánh được hợp nhất	<code>nhánh git --merged</code>
Liệt kê các chi nhánh chưa được hợp nhất	<code>nhánh git --không hợp nhất</code>
Liệt kê các nhánh chưa cam kết	<code>nhánh git -- chưa [&lt;commit&gt;]</code>

Ghi chú:

- Thêm v bổ sung vào -v ví dụ `$ git Branch -avv` hoặc `$ git Branch -vv` sẽ in tên của nhánh thương nguồn nữa.
- Những cành màu đỏ là những cành ở xa

## Phần 14.3: Xóa một nhánh từ xa

Để xóa một nhánh trên kho lưu trữ từ xa gốc, bạn có thể sử dụng cho phiên bản Git 1.5.0 trở lên

```
nguồn gốc git đẩy :<branchName>
```

và kể từ phiên bản Git 1.7.0, bạn có thể xóa một nhánh từ xa bằng cách sử dụng

```
nguồn gốc git đẩy --delete <branchName>
```

Để xóa một nhánh theo dõi từ xa cục bộ:

```
git nhánh --delete --remotes <remote>/<branch> git nhánh -dr  
<remote>/<branch> # Ngắn hơn
```

```
git get <remote> --Prune # Xóa nhiều nhánh theo dõi lỗi thời git get <remote> -p  
# Ngắn hơn
```

Để xóa một nhánh cục bộ. Lưu ý rằng thao tác này sẽ không xóa nhánh nếu nó có bất kỳ thay đổi nào chưa được hợp nhất:

```
nhánh git -d <branchName>
```

Để xóa một nhánh, ngay cả khi nó có những thay đổi chưa được hợp nhất:

```
nhánh git -D <branchName>
```

## Mục 14.4: Chuyển nhanh về nhánh trước

Bạn có thể nhanh chóng chuyển sang nhánh trước bằng cách sử dụng

```
kiểm tra git -
```

## Mục 14.5: Kiểm tra chi nhánh mới theo dõi chi nhánh từ xa

Có ba cách để tạo một tính năng nhánh mới theo dõi nguồn gốc/tính năng của nhánh từ xa:

- `git kiểm tra --track -b` tính năng nguồn gốc/tính năng, `git kiểm tra -t` nguồn gốc/tính năng, `git kiểm tra git` - giả sử rằng không có nhánh tính năng cục bộ và chỉ có một điều khiển từ xa với nhánh tính năng .

Để thiết lập ngược dòng để theo dõi nhánh từ xa - hãy gõ:

- `nhánh git --set-upstream-to=<remote>/<branch> <branch>`
- `nhánh git -u <remote>/<nhánh> <nhánh>`

Ở đâu:

- `<remote>` có thể là: Origin, develop hoặc do người dùng tạo,
- `<branch>` là nhánh của người dùng để theo dõi trên remote.

Để xác minh những chi nhánh từ xa mà chi nhánh địa phương của bạn đang theo dõi:

- `nhánh git -vv`

## Mục 14.6: Xóa chi nhánh cục bộ

```
$ git nhánh -d dev
```

Xóa nhánh có tên dev nếu các thay đổi của nó được hợp nhất với nhánh khác và sẽ không bị mất. Nếu nhánh dev chưa có thay đổi chưa được hợp nhất sẽ bị mất, `git nhánh -d` sẽ thất bại:

```
$ git Branch -d lỗi dev:  
Nhánh 'dev' chưa được hợp nhất hoàn toàn.  
Nếu bạn chắc chắn muốn xóa nó, hãy chạy 'git Branch -D dev'.
```

Theo thông báo cảnh báo, bạn có thể buộc xóa nhánh (và mất mọi thay đổi chưa được hợp nhất trong nhánh đó) bằng cách sử dụng cờ -D :

```
$ git nhánh -D dev
```

## Phần 14.7: Tạo một nhánh mồ côi (tức là nhánh không có cam kết gốc)

```
kiểm tra git --orphan chi nhánh mới mồ côi
```

Cam kết đầu tiên được thực hiện trên nhánh mới này sẽ không có cha mẹ và nó sẽ là gốc rễ của một lịch sử mới hoàn toàn ngắt kết nối với tất cả các nhánh và cam kết khác.

nguồn

## Mục 14.8: Đổi tên chi nhánh

Đổi tên chi nhánh bạn đã kiểm tra:

```
nhánh git -m new_branch_name
```

Đổi tên chi nhánh khác:

```
nhánh git -m nhánh_you_want_to_rename new_branch_name
```

## Mục 14.9: Tìm kiếm theo nhánh

Để liệt kê các nhánh cục bộ có chứa một cam kết hoặc thẻ cụ thể

```
nhánh git -- chứa <commit>
```

Để liệt kê các nhánh cục bộ và từ xa có chứa một cam kết hoặc thẻ cụ thể

```
nhánh git -a --contains <commit>
```

## Mục 14.10: Đẩy nhánh về remote

Sử dụng để đẩy các cam kết được thực hiện trên nhánh cục bộ của bạn tới kho lưu trữ từ xa.

Lệnh `git push` có hai đối số:

- Một tên từ xa, ví dụ: nguồn gốc
- Tên chi nhánh, ví dụ: master

Ví dụ:

```
git đẩy <REMOTENAME> <BRANCHNAME>
```

Ví dụ: bạn thường chạy `git push Origin master` để đẩy các thay đổi cục bộ vào kho lưu trữ trực tuyến của mình.

Sử dụng `-u` (viết tắt của `--set-upstream`) sẽ thiết lập thông tin theo dõi trong quá trình đẩy.

```
git push -u <REMOTENAME> <BRANCHNAME>
```

Theo mặc định, `git push` đẩy nhánh cục bộ đến nhánh từ xa có cùng tên. Ví dụ: nếu bạn có một tính năng cục bộ được gọi là `tính năng mới`, nếu bạn đẩy nhánh cục bộ thì nó cũng sẽ tạo ra một tính năng mới của nhánh từ xa. Nếu bạn muốn sử dụng một tên khác cho nhánh từ xa, hãy thêm tên từ xa sau tên nhánh cục bộ, cách nhau bằng `::`:

```
git push <REMOTENAME> <LOCALBRANCHNAME>:<REMOTEBRANCHNAME>
```

## Mục 14.11: Di chuyển nhánh HEAD hiện tại sang một cam kết tùy ý

Một nhánh chỉ là một con trỏ tới một cam kết, vì vậy bạn có thể tự do di chuyển nó xung quanh. Để làm cho nhánh tham chiếu đến `aabbcc` cam kết, hãy ra lệnh

```
thiết lập lại git --hard aabbcc
```

Xin lưu ý rằng điều này sẽ ghi đè lên cam kết hiện tại của chi nhánh của bạn và do đó, toàn bộ lịch sử của nó. Bạn có thể mất một số công việc bằng cách đưa ra lệnh này. Nếu đúng như vậy, bạn có thể sử dụng `reflog` để khôi phục các cam kết bị mất. Bạn nên thực hiện lệnh này trên một nhánh mới thay vì nhánh hiện tại của mình.

Tuy nhiên, lệnh này có thể đặc biệt hữu ích khi khởi động lại hoặc thực hiện các sửa đổi lịch sử lớn khác.

## Chương 15: Danh sách Rev

Tham số --

Chi tiết

oneline Hiển thị cam kết dưới dạng một dòng với tiêu đề của chúng.

### Mục 15.1: Liệt kê các cam kết trong master nhưng không có trong Origin/master

```
git rev-list --oneline master ^Origin/master
```

Git rev-list sẽ liệt kê các cam kết trong một nhánh không có trong nhánh khác. Nó là một công cụ tuyệt vời khi bạn đang cố gắng tìm hiểu xem mã đã được hợp nhất vào một nhánh hay chưa.

- Sử dụng tùy chọn `--oneline` sẽ hiển thị tiêu đề của mỗi lần xác nhận.
- Toán tử `^` loại trừ các cam kết trong nhánh được chỉ định khỏi danh sách.
- Bạn có thể vượt qua nhiều hơn hai nhánh nếu muốn. Ví dụ: `git rev-list foo bar ^baz` liệt kê các cam kết trong foo và bar, nhưng không liệt kê baz.

# Chương 16: Đè bếp

## Mục 16.1: Xóa các cam kết gần đây mà không cần khởi động lại

Nếu bạn muốn nén các lần xác nhận x trước đó thành một lần duy nhất, bạn có thể sử dụng các lệnh sau:

```
git reset --soft HEAD~x git
commit
```

Thay thế x bằng số lần xác nhận trước đó mà bạn muốn đưa vào cam kết bị xóa.

Lưu ý rằng điều này sẽ tạo ra một cam kết mới, về cơ bản là quên thông tin về các cam kết x trước đó bao gồm tác giả, tin nhắn và ngày tháng của chúng. Trước tiên, bạn có thể muốn sao chép-dán một thông báo cam kết hiện có.

## Mục 16.2: Xóa cam kết trong quá trình hợp nhất

Bạn có thể sử dụng `git merge --squash` để nén các thay đổi do một nhánh đưa vào thành một lần xác nhận duy nhất. Sẽ không có cam kết thực tế nào được tạo ra.

```
git merge --squash <branch> git
cam kết
```

Điều này ít nhiều tương đương với việc sử dụng `git reset`, nhưng thuận tiện hơn khi các thay đổi được kết hợp có tên tương ứng. So sánh:

```
git kiểm tra <nhanh> git
reset --soft $(git merge-base master <branch>) git commit
```

## Phần 16.3: Xóa các cam kết trong quá trình Rebase

Các cam kết có thể bị xóa trong quá trình `rebase git`. Bạn nên hiểu rõ về việc khởi động lại trước khi thử xóa các cam kết theo cách này.

1. Xác định cam kết nào bạn muốn khởi động lại và lưu ý hàm bấm cam kết của nó.
2. Chạy `git rebase -i [bấm cam kết]`.

Ngoài ra, bạn có thể nhập `HEAD~4` thay vì hàm bấm xác nhận, để xem lần xác nhận mới nhất và 4 lần xác nhận khác trước lần xác nhận mới nhất.

3. Trong trình soạn thảo mở ra khi chạy lệnh này, hãy xác định những cam kết nào bạn muốn xóa.  
Thay thế lựa chọn ở đầu những dòng đó bằng bí để nén chúng vào lần xác nhận trước đó.
4. Sau khi chọn những cam kết mà bạn muốn xóa, bạn sẽ được nhắc viết một thông báo cam kết.

Ghi nhật ký cam kết để xác định nơi cần rebase

```
> git log --oneline
612f2f7 Cam kết này không nên bị xóa d84b05d Cam kết
này sẽ bị xóa ac60234 Một cam kết khác 36d15de
Rebase từ đây
```

17692d1 Đã làm thêm một số thứ **nữa**

e647334 Một cam kết khác 2e30df6

Cam kết ban đầu

```
> git rebase -i 36d15de
```

Tại thời điểm này, trình soạn thảo bạn chọn sẽ bật lên nơi bạn có thể mô tả những gì bạn muốn làm với các cam kết. Git cung cấp trợ giúp trong phần bình luận. Nếu bạn để nguyên như vậy thì sẽ không có gì xảy ra vì mọi cam kết sẽ được giữ nguyên và thứ tự của chúng sẽ giống như trước khi rebase. Trong ví dụ này, chúng tôi áp dụng các lệnh sau:

pick ac60234 Một cam kết khác bí đáo d84b05d

Cam kết này nên được nén lại chọn 612f2f7 Cam kết này không nên bị nén

```
# Rebase 36d15de..612f2f7 lên 36d15de (3 lệnh) #
```

# Lệnh: # p,

pick = dùng commit # r, reword

= dùng commit, nhưng sửa thông điệp commit # e, edit = dùng commit, nhưng

dùng sửa # s, bì = dùng commit, nhưng gộp vào commit trước đó #

f , fixup = like "squash", nhưng loại bỏ thông điệp tường trình của cam kết này

# x, exec = run command (phần còn lại của dòng) bằng shell # # Nhữn dòng này có thể được sắp

xếp lại; chúng được thực hiện từ trên xuống dưới. # # Nếu bạn bỏ dòng ở đây RẰNG CAM

KẾT SẼ BỊ MẤT. #

```
# Tuy nhiên, nếu bạn xóa mọi thứ, quá trình rebase sẽ bị hủy bỏ. # # Lưu ý rằng các cam kết
```

trong sẽ được nhận xét

Nhật ký Git sau khi viết thông báo cam kết

```
> git log --oneline 77393eb
```

Không nên xóa cam kết này e090a8c Một cam kết khác 36d15de

Rebase từ đây

17692d1 Đã thực hiện **thêm** một số nội dung

e647334 Một cam kết khác 2e30df6

Cam kết ban đầu

## Mục 16.4: Tự động nén và sửa lỗi

Khi thực hiện các thay đổi, có thể chỉ định rằng cam kết trong tương lai sẽ bị nén thành một cam kết khác và điều này có thể được thực hiện như sau:

```
git commit --squash=[commit hash của commit mà commit này sẽ bị nén vào]
```

Người ta cũng có thể sử dụng --fixup=[commit hash] thay thế cho fixup.

Cũng có thể sử dụng các từ trong thông điệp cam kết thay vì hàm băm cam kết, như vậy,

```
git commit --squash :/things
```

nơi cam kết gần đây nhất với từ 'mọi thứ' sẽ được sử dụng.

Thông báo của những cam kết này sẽ bắt đầu bằng '**fixup!**' hoặc '**bí!**' tiếp theo là phần còn lại của thông báo cam kết mà các cam kết này sẽ được nén vào.

Khi khởi động lại cờ `--autosquash` nên sử dụng tính năng autosquash/fixup.

## Mục 16.5: Autosquash: Cam kết mã bạn muốn xóa trong quá trình rebase

Với lịch sử sau đây, hãy tưởng tượng bạn thực hiện một thay đổi mà bạn muốn đưa vào cam kết bbb2222 A cam kết thứ hai:

```
$ git log --oneline --trang trí ccc3333
(HEAD -> master) Cam kết thứ ba bbb2222 Cam kết thứ hai
aaa1111 Cam kết đầu tiên
9999999 Cam kết ban đầu
```

Khi bạn đã thực hiện các thay đổi của mình, bạn có thể thêm chúng vào chỉ mục như bình thường, sau đó xác nhận chúng bằng cách sử dụng đối số `--fixup` có tham chiếu đến cam kết mà bạn muốn đưa vào:

```
$ git thêm .
$ git commit --fixup bbb2222 [my-
feature-branch ddd4444] sửa lỗi! Cam kết thứ hai
```

Điều này sẽ tạo một cam kết mới với thông báo cam kết mà Git có thể nhận ra trong quá trình rebase tương tác:

```
Sửa lỗi $ git log --oneline --trang trí
ddd4444 (HEAD -> master) ! Cam kết thứ hai ccc3333 Cam kết
thứ ba bbb2222 Cam kết thứ hai
aaa1111 Cam kết đầu tiên

9999999 Cam kết ban đầu
```

Tiếp theo, thực hiện rebase tương tác với đối số `--autosquash`:

```
$ git rebase --autosquash --interactive HEAD~4
```

Git sẽ đề xuất bạn nén cam kết bạn đã thực hiện với cam kết `--fixup` vào đúng vị trí:

```
chọn aaa1111 Lần cam kết đầu tiên
chọn bbb2222 Bản sửa lỗi cam kết thứ
hai Bản sửa lỗi ddd4444! Cam kết thứ hai chọn
ccc3333 Cam kết thứ ba
```

Để tránh phải gõ `--autosquash` trên mỗi rebase, bạn có thể bật tùy chọn này theo mặc định:

```
$ git config --global rebase.autosquash true
```

## Chương 17: Háí Anh Đào

	Chi tiết
Tham số -e, --edit	
	edit Với tùy chọn này, <code>git Cherry-pick</code> sẽ cho phép bạn chỉnh sửa thông báo cam kết trước khi cam kết.
-x	Khi ghi lại cam kết, hãy thêm một dòng có nội dung "(anh đào được chọn từ cam kết .)" vào thông báo cam kết ban đầu để cho biết thay đổi này được anh đào chọn từ cam kết nào. Điều này chỉ được thực hiện đối với những quả háí anh đào mà không có xung đột.
--ff	Nếu HEAD hiện tại giống với cha mẹ của cam kết đã được chọn, thì việc chuyển tiếp nhanh đến cam kết này sẽ được thực hiện.
--Tiếp tục	Tiếp tục thao tác đang diễn ra bằng cách sử dụng thông tin trong <code>.git/sequencer</code> . Có thể được sử dụng để tiếp tục sau khi giải quyết xung đột trong thao tác chọn hoặc hoàn nguyên không thành công.
--tùy ý	Hãy quên đi hoạt động hiện tại đang diễn ra. Có thể được sử dụng để xóa trạng thái trình sắp xếp thứ tự sau khi chọn hoặc hoàn nguyên không thành công.
--Huỷ bỏ	Hủy thao tác và trở về trạng thái trình tự trước.

`Cherry-pick` lấy bản vá đã được giới thiệu trong một cam kết và cố gắng áp dụng lại nó trên nhánh mà bạn hiện đang ở TRÊN.

Nguồn: Sách Git SCM

### Phần 17.1: Sao chép một cam kết từ nhánh này sang nhánh khác

`git Cherry-pick <commit-hash>` sẽ áp dụng những thay đổi được thực hiện trong một cam kết hiện có cho một nhánh khác, đồng thời ghi lại một cam kết mới. Về cơ bản, bạn có thể sao chép các cam kết từ chi nhánh này sang chi nhánh khác.

Cho cây sau (Nguồn)

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 [chính]
```

```
  \ 76cada - 62ecb3 - b886a0 [tính năng]
```

Giả sử chúng tôi muốn sao chép `b886a0` sang bản chính (trên `5a6057`).

Chúng ta có thể chạy

```
chủ thanh toán git
git Cherry-pick b886a0
```

Bây giờ cây của chúng ta sẽ trông giống như sau:

```
dd2e86 - 946992 - 9143a9 - a6fd86 - 5a6057 - a66b23 [chính]
```

```
  \ 76cada - 62ecb3 - b886a0 [tính năng]
```

Trong đó cam kết mới `a66b23` có cùng nội dung (khác biệt nguồn, thông báo cam kết) với `b886a0` (nhưng có cha mẹ khác). Lưu ý rằng việc chọn anh đào sẽ chỉ nhận các thay đổi trên cam kết đó (`b886a0` trong trường hợp này) chứ không phải tất cả các thay đổi trong nhánh tính năng (để làm điều này, bạn sẽ phải sử dụng tính năng khởi động lại hoặc hợp nhất).

### Phần 17.2: Sao chép một loạt các cam kết từ nhánh này sang nhánh khác

`git Cherry-pick <commit-A>..<commit-B>` sẽ đặt mọi cam kết sau A trở lên và bao gồm cả B lên trên nhánh hiện đã được thanh toán.

```
git Cherry-pick <commit-A>^..<commit-B> sẽ đặt cam kết A và mọi cam kết lên đến và bao gồm cả B lên trên nhánh hiện đã được thanh toán.
```

### Mục 17.3: Kiểm tra xem có cần hái anh đào không

Trước khi bắt đầu quá trình chọn anh đào, bạn có thể kiểm tra xem cam kết bạn muốn chọn anh đào đã tồn tại trong nhánh mục tiêu hay chưa, trong trường hợp đó bạn không phải làm gì cả.

```
git Branch --contains <commit> liệt kê các nhánh cục bộ chứa cam kết đã chỉ định.
```

```
git Branch -r --contains <commit> cũng bao gồm các nhánh theo dõi từ xa trong danh sách.
```

### Phần 17.4: Tìm các cam kết chưa được áp dụng cho thượng nguồn

Lệnh `git Cherry` hiển thị những thay đổi chưa được chọn.

Ví dụ:

```
bậc thầy kiểm tra git
phát triển git anh đào
```

... và xem đầu ra một chút như thế này:

```
+ 492508acab7b454eee8b805f8ba906056eede0ff -
5ceb5a9077ddb9e78b1e8f24bfc70e674c627949
+ b4459544c000f4d51d1ec23f279d9cdb19c1d32b +
b6ce3b78e938644a293b2dd2a15b2fecb1b54cd9
```

Các cam kết với + sẽ là những cam kết chưa được chọn để phát triển.

Cú pháp:

```
git anh đào [-v] [<ngược dòng> [<head> [<giới hạn>]]]
```

Tùy chọn:

-v Hiển thị các chủ đề cam kết bên cạnh SHA1.

< upstream > Nhánh ngược dòng để tìm kiếm các cam kết tương đương. Mặc định là nhánh ngược dòng của HEAD.

< trướng > Chi nhánh làm việc; mặc định là ĐẦU.

< giới hạn > Không báo cáo các cam kết lên tới (và bao gồm) giới hạn.

Kiểm tra [tài liệu git-cherry](#) để biết thêm thông tin.

# Chương 18: Phục hồi

## Mục 18.1: Khôi phục sau khi thiết lập lại

Với Git, bạn có thể (gần như) luôn quay ngược dòng hồ

Đừng ngại thử nghiệm các lệnh viết lại lịch sử\*. Theo mặc định, Git không xóa các cam kết của bạn trong 90 ngày và trong thời gian đó, bạn có thể dễ dàng khôi phục chúng từ nhật ký lại:

```
$ git reset @~3 # quay lại 3 lần xác nhận $ git
reflog c4f708b
HEAD@{0}: reset: chuyển sang @~3 2c52489 HEAD@{1}:
commit: nhiều thay đổi hơn 4a5246d HEAD@{2}:
commit: coi trọng thay đổi e8571e4 HEAD@{3}: cam kết: thực
hiện một số thay đổi ... cam kết trước đó ...
$ git reset 2c52489 ...
và bạn quay lại nơi bạn đã bắt đầu
```

\* Tuy nhiên, hãy chú ý đến các tùy chọn như **--hard** và **--force** – chúng có thể loại bỏ dữ liệu.

\* Ngoài ra, tránh viết lại lịch sử trên bất kỳ chi nhánh nào mà bạn đang cộng tác.

## Phần 18.2: Khôi phục từ git stash

Để nhận được stash gần đây nhất của bạn sau khi chạy git stash, hãy sử dụng

**git stash áp dụng**

Để xem danh sách các kho lưu trữ của bạn, hãy sử dụng

**danh sách kho git**

Bạn sẽ nhận được một danh sách trông giống như thế này

```
stash@{0}: WIP trên master: 67a4e01 Hợp nhất các thử nghiệm thành develop
stash@{1}: WIP trên master: 70f0d95 Thêm vai trò người dùng vào localStorage khi người dùng đăng nhập
```

Chọn một kho git khác để khôi phục bằng số hiển thị cho kho lưu trữ bạn muốn

**git stash áp dụng stash@{2}**

Bạn cũng có thể chọn 'git stash pop', nó hoạt động tương tự như 'git stash apply' như ..

**git stash pop**

hoặc

**git stash pop stash@{2}**

Sự khác biệt giữa git stash áp dụng và git stash pop...

git stash pop: dữ liệu stash sẽ bị xóa khỏi danh sách stash.

Bản tái:

danh sách kho git

Bạn sẽ nhận được một danh sách trông giống như thế này

```
stash@{0}: WIP trên master: 67a4e01 Hợp nhất các thử nghiệm thành develop
stash@{1}: WIP trên master: 70f0d95 Thêm vai trò người dùng vào localStorage khi người dùng đăng nhập
```

Bây giờ bật dữ liệu stash bằng lệnh

git stash pop

Một lần nữa Kiểm tra danh sách lưu trữ

danh sách kho git

Bạn sẽ nhận được một danh sách trông giống như thế này

```
stash@{0}: WIP trên master: 70f0d95 Thêm vai trò người dùng vào localStorage khi người dùng đăng nhập
```

Bạn có thể thấy một dữ liệu trong kho lưu trữ bị xóa (xuất hiện) khỏi danh sách kho lưu trữ và stash@{1} trở thành stash@{0}.

### Phần 18.3: Khôi phục từ một cam kết bị mất

Trong trường hợp bạn đã hoàn nguyên về cam kết trước đây và mất cam kết mới hơn, bạn có thể khôi phục cam kết bị mất bằng cách chạy

git reflog

Sau đó tìm cam kết bị mất của bạn và đặt lại cam kết đó bằng cách thực hiện

```
git reset HEAD --hard <sha1-of-commit>
```

### Phần 18.4: Khôi phục tệp đã xóa sau khi cam kết

Trong trường hợp bạn vô tình xóa một tập tin và sau đó nhận ra rằng bạn cần lấy lại nó.

Trước tiên hãy tìm id cam kết của cam kết đã xóa tệp của bạn.

git log --diff-filter=D --tóm tắt

Sẽ cung cấp cho bạn một bản tóm tắt được sắp xếp về các cam kết đã xóa các tệp.

Sau đó tiến hành khôi phục tập tin bằng cách

```
kiểm tra git 81eeccf~1 <tên-tệp-mất-của-bạn>
```

(Thay thế 81eeccf bằng id cam kết của riêng bạn)

### Mục 18.5: Khôi phục file về phiên bản trước

Để khôi phục tệp về phiên bản trước, bạn có thể sử dụng reset.

```
git reset <sha1-of-commit> <tên tệp>
```

Nếu bạn đã thực hiện các thay đổi cục bộ đối với tệp (mà bạn không yêu cầu!) Bạn cũng có thể sử dụng tùy chọn `--hard`

## Mục 18.6: Khôi phục nhánh đã xóa

Để khôi phục một nhánh đã bị xóa, bạn cần tìm cam kết đúng đầu nhánh đã bị xóa bằng cách chạy

```
git reflog
```

Sau đó bạn có thể tạo lại nhánh bằng cách chạy

```
kiểm tra git -b <tên nhánh> <sha1-of-commit>
```

Bạn sẽ không thể khôi phục các nhánh đã bị xóa nếu ~~trình thu gom rác của git~~ đã xóa các cam kết lơ lửng - những cam kết không có ref. Luôn có bản sao lưu kho lưu trữ của bạn, đặc biệt khi bạn làm việc trong một nhóm nhỏ/dự án độc quyền

# Chương 19: Git Clean

## Tham số

## Chi tiết

Xóa các thư mục không bị theo dõi cũng như các tệp không bị theo dõi. Nếu một thư mục không bị theo dõi được quản lý bởi -d kho lưu trữ Git khác, thì theo mặc định, thư mục đó sẽ không bị xóa. Sử dụng tùy chọn -f hai lần nếu bạn thực sự muốn xóa thư mục đó.

Nếu biến cấu hình Git sạch. requireForce không được đặt thành false, git clean sẽ từ chối xóa các tập tin hoặc thư mục -f, --force bắt buộc trừ khi được cung cấp -f, -n hoặc -i. Git sẽ từ chối xóa các thư mục có thư mục hoặc tệp con .git trừ khi có -f thư hai. -i, --interactive Nhắc nhở tương

tác xóa từng tệp. -n, --dry-run Chỉ hiển thị danh sách các tệp cần xóa mà

không thực sự xóa chúng. -q, --quiet Chỉ hiển thị lỗi, không hiển thị danh sách các file đã xóa thành công.

## Phần 19.1: Làm sạch tương tác

`git sạch -i`

Sẽ in ra các mục cần xóa và yêu cầu xác nhận thông qua các lệnh như sau:

```
Sẽ xóa các mục sau: folder/file1.py
folder/file2.py
*** Lệnh *** 1:
clean 2: lọc theo
mẫu 5: thoát 6: trợ giúp Bởi giờ thì sao>
```

3: chọn theo số

4: hỏi từng người

Tùy chọn tương tác tôi có thể được thêm vào cùng với các tùy chọn khác như X, d, v.v.

## Mục 19.2: Loại bỏ mạnh mẽ các tập tin không bị theo dõi

`git sạch -f`

Sẽ loại bỏ tất cả các tập tin không bị theo dõi.

## Phần 19.3: Làm sạch các tập tin bị bỏ qua

`git sạch -fx`

Sẽ xóa tất cả các tệp bị bỏ qua khỏi thư mục hiện tại và tất cả các thư mục con.

`git sạch -Xn`

Sẽ xem trước tất cả các tập tin sẽ được làm sạch.

## Phần 19.4: Làm sạch tất cả các thư mục không bị theo dõi

`git sạch -fd`

Sẽ xóa tất cả các thư mục không bị theo dõi và các tập tin bên trong chúng. Nó sẽ bắt đầu tại thư mục làm việc hiện tại và sẽ lặp qua tất cả các thư mục con.

`git sạch -dn`

Sẽ xem trước tất cả các thư mục sẽ được làm sạch.

# Chương 20: Sử dụng tệp .gitattributes

## Mục 20.1: Chuẩn hóa kết thúc dòng tự động

Tạo tệp .gitattributes trong thư mục gốc của dự án có chứa:

\* văn bản=tự động

Điều này sẽ dẫn đến tất cả các tệp văn bản (như được xác định bởi Git) được cam kết với LF, nhưng được kiểm tra theo mặc định của hệ điều hành máy chủ.

Điều này tương đương với các giá trị mặc định core.autocrlf được đề xuất của:

- đầu vào trên Linux/macOS
- **đúng** trên Windows

## Phần 20.2: Xác định tệp nhị phân

Git khá giỏi trong việc xác định tệp nhị phân, nhưng bạn có thể chỉ định rõ ràng tệp nào là tệp nhị phân. Tạo tệp .gitattributes trong thư mục gốc của dự án có chứa:

\*.png nhị phân

nhị phân là thuộc tính macro tích hợp tương đương với `-diff -merge -text`.

## Mục 20.3: Mẫu .gitattribute được điền sẵn

Nếu bạn không chắc nên liệt kê quy tắc nào trong tệp .gitattributes hoặc bạn chỉ muốn thêm các thuộc tính được chấp nhận chung vào dự án của mình, bạn có thể chọn hoặc tạo tệp .gitattributes tại:

- <https://gitattributes.io/>
- <https://github.com/alexkaratarakis/gitattribut>

## Mục 20.4: Tắt chuẩn hóa kết thúc dòng

Tạo tệp .gitattributes trong thư mục gốc của dự án có chứa:

\* -văn bản

Điều này tương đương với việc đặt core.autocrlf = **false**.

## Chương 21: Tệp .mailmap: Liên kết cộng tác viên và bí danh email

### Mục 21.1: Hợp nhất những người đóng góp theo bí danh để hiển thị số lượng cam kết trong shortlog

Khi những người đóng góp thêm vào một dự án từ các máy hoặc hệ điều hành khác nhau, có thể xảy ra trường hợp họ sử dụng các địa chỉ email hoặc tên khác nhau cho việc này, điều này sẽ phân mảnh danh sách và số liệu thống kê của người đóng góp.

Chạy `git shortlog -sn` để lấy danh sách những người đóng góp và số lượng cam kết của họ có thể dẫn đến kết quả đầu ra sau:

```
Patrick Rothfuss 871
Elizabeth Mät Träng 762
E. Träng 184
Rothfuss, Patrick 90
```

Sự phân mảnh/phân tách này có thể được điều chỉnh bằng cách cung cấp tệp văn bản thuần túy `.mailmap`, chứa ánh xạ email.

Tất cả tên và địa chỉ email được liệt kê trong một dòng sẽ được liên kết tương ứng với thực thể được đặt tên đầu tiên.

Đối với ví dụ trên, ánh xạ có thể trông như thế này:

```
Patrick Rothfuss <fussy@kingkiller.com> Rothfuss, Patrick <fussy@kingkiller.com> Elizabeth Moon
<emoon@marines.mil> E. Moon <emoon@scifi.org>
```

Khi tệp này tồn tại trong thư mục gốc của dự án, việc chạy lại `git shortlog -sn` sẽ dẫn đến một danh sách cô đọng:

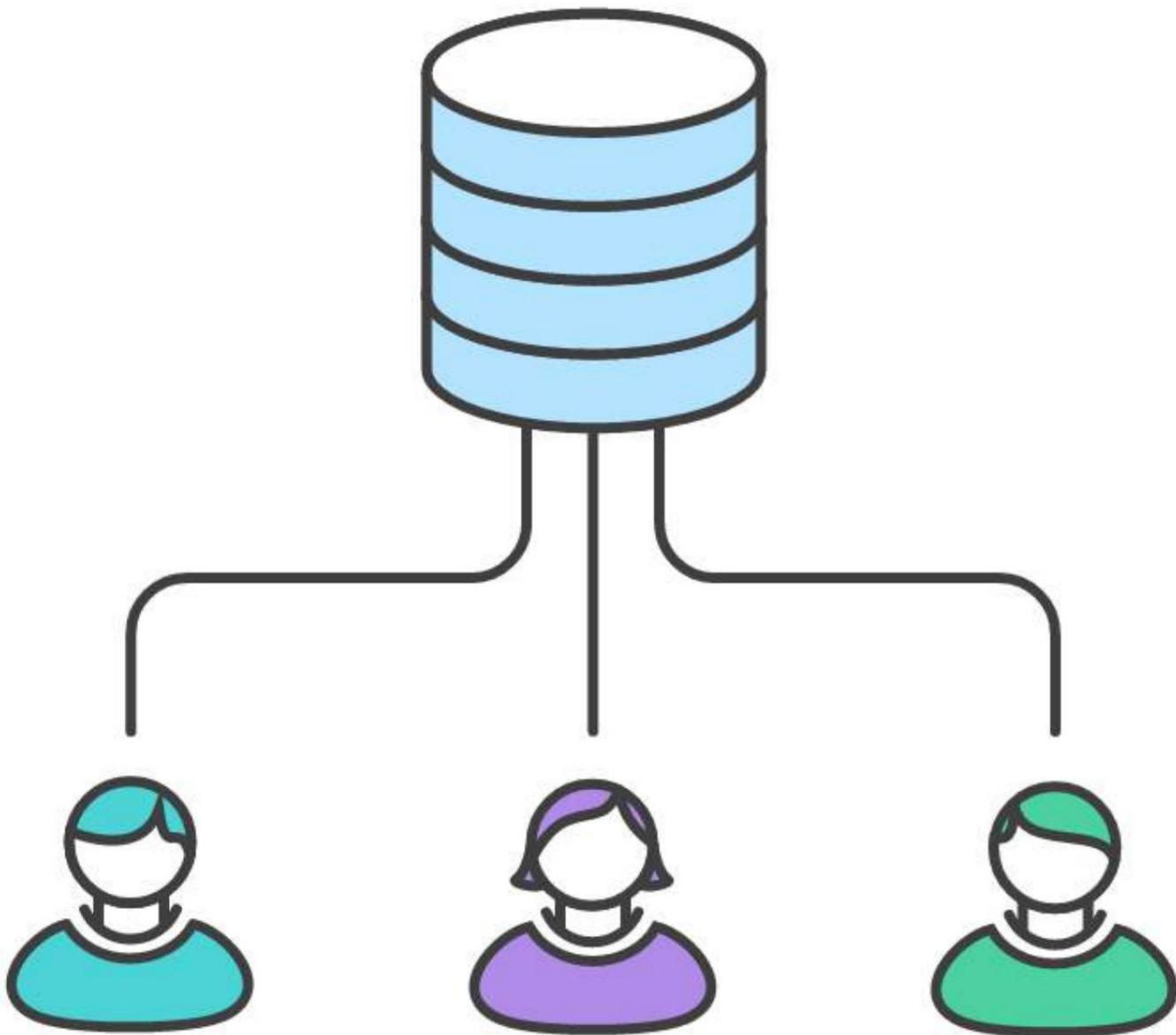
```
Patrick Rothfuss 961
Elizabeth Mät Träng 946
```

## Chương 22: Phân tích các loại quy trình công việc

### Phần 22.1: Quy trình làm việc tập trung

Với mô hình quy trình làm việc cơ bản này, một nhánh chính chứa tất cả sự phát triển tích cực. Những người đóng góp sẽ cần đặc biệt chắc chắn rằng họ thu thập được những thay đổi mới nhất trước khi tiếp tục phát triển, vì nhánh này sẽ thay đổi nhanh chóng. Mọi người đều có quyền truy cập vào kho lưu trữ này và có thể thực hiện các thay đổi ngay trên nhánh chính.

Hình ảnh trực quan của mô hình này:



Đây là mô hình kiểm soát phiên bản cổ điển, dựa trên đó các hệ thống cũ hơn như Subversion và CVS được xây dựng.

Phần mềm hoạt động theo cách này được gọi là Hệ thống kiểm soát phiên bản tập trung hoặc CVCS. Mặc dù Git có khả năng hoạt động theo cách này, nhưng có những nhược điểm đáng chú ý, chẳng hạn như phải thực hiện hợp nhất trước mỗi lần kéo. Một nhóm rất có thể làm việc theo cách này, nhưng việc giải quyết xung đột hợp nhất liên tục có thể sẽ tiêu tốn rất nhiều thời gian quý báu.

Đây là lý do tại sao Linus Torvalds tạo ra Git không phải dưới dạng CVCS mà là DVCS hoặc Hệ thống kiểm soát phiên bản phân tán, tương tự như Mercurial. Ưu điểm của cách làm mới này là tính linh hoạt được thể hiện trong các ví dụ khác trên trang này.

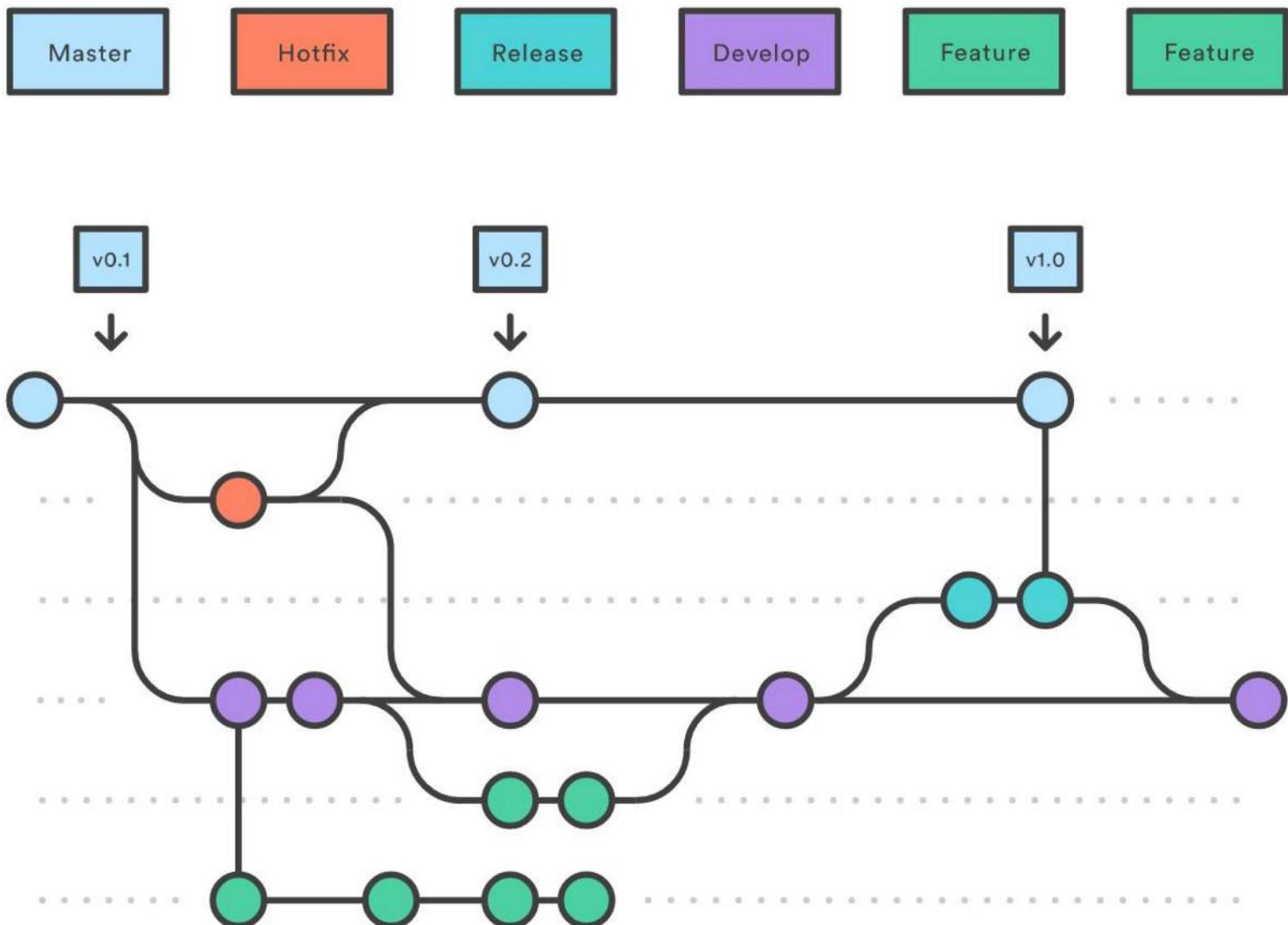
## Phân 22.2: Quy trình làm việc của Gitflow

Được đề xuất ban đầu bởi [Vincent Driessen](#), Gitflow là quy trình phát triển sử dụng git và một số nhánh được xác định trước. Đây có thể được coi là trường hợp đặc biệt của Quy trình làm việc của Chi nhánh tính năng.

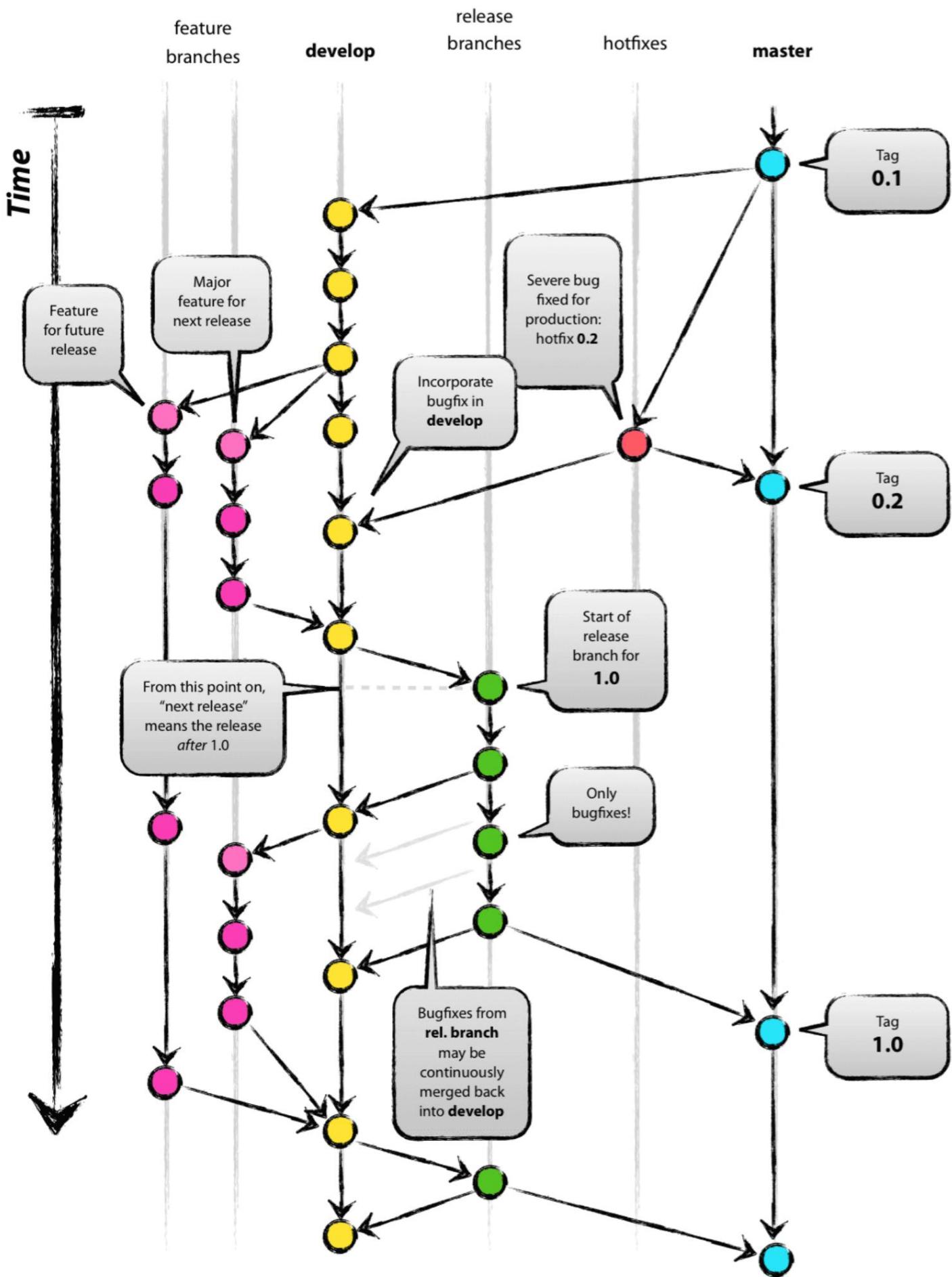
Ý tưởng của việc này là có các nhánh riêng biệt dành riêng cho các phần cụ thể trong quá trình phát triển:

- nhánh chính luôn là mã sản xuất gần nhất . Mã thử nghiệm không thuộc về nơi này. nhánh phát triển chứa tất cả
- sự phát triển mới nhất. Những thay đổi phát triển này có thể là khá nhiều thứ, nhưng những tính năng lớn hơn được dành riêng cho các nhánh riêng của chúng. Mã ở đây luôn được xử lý và hợp nhất thành bản phát hành trước khi phát hành/triển khai. các nhánh hotfix dành cho các bản sửa lỗi nhỏ, không
- thẻ đợi đến bản phát hành tiếp theo. các nhánh hotfix tách khỏi nhánh chính và được hợp nhất lại thành cả nhánh chính và nhánh phát triển. các nhánh phát hành được sử dụng để phát hành sự phát triển
- mới từ phát triển đến chính. Bất kỳ thay đổi nào vào phút cuối, chẳng hạn như thay đổi số phiên bản, đều được thực hiện trong nhánh phát hành, sau đó được hợp nhất lại thành nhánh chính và phát triển. Khi triển khai phiên bản mới, bản chính phải được gắn thẻ với số phiên bản hiện tại (ví dụ: sử dụng [phiên bản ngữ nghĩa](#)) để tham khảo trong tương lai và khôi phục dễ dàng. các [nhánh tính năng](#) được dành riêng cho các tính năng lớn hơn. Chúng
- được phát triển đặc biệt trong các nhánh được chỉ định và được tích hợp với phát triển khi hoàn thành. Các nhánh tính năng chuyên dụng giúp tách biệt quá trình phát triển và có thể triển khai các tính năng đã hoàn thành một cách độc lập với nhau.

Hình ảnh trực quan của mô hình này:



Biểu diễn ban đầu của mô hình này:



### Phần 22.3: Quy trình làm việc của nhánh tính năng

Ý tưởng cốt lõi đằng sau Quy trình làm việc của nhánh tính năng là tất cả quá trình phát triển tính năng sẽ diễn ra trong một nhánh chuyên dụng thay vì nhánh chính. Việc đóng gói này giúp nhiều nhà phát triển dễ dàng làm việc trên một tính năng cụ thể mà không làm ảnh hưởng đến cơ sở mã chính. Điều đó cũng có nghĩa là nhánh chính sẽ không bao giờ chứa mã bị hỏng, đây là một lợi thế rất lớn cho môi trường tích hợp liên tục.

Việc đóng gói phát triển tính năng cũng giúp có thể tận dụng các yêu cầu kéo, đây là một cách để bắt đầu các cuộc thảo luận xung quanh một nhánh. Họ cho các nhà phát triển khác cơ hội đăng nhập vào một tính năng trước khi nó được tích hợp vào dự án chính thức. Hoặc, nếu bạn gặp khó khăn khi đang thực hiện một tính năng, bạn có thể mở yêu cầu kéo để xin ý kiến từ đồng nghiệp của mình. Vấn đề là, yêu cầu kéo giúp nhóm của bạn dễ dàng nhận xét về công việc của nhau.

dựa trên [Hướng dẫn của Atlassian](#).

### Phần 22.4: Luồng GitHub

Phổ biến trong nhiều dự án nguồn mở nhưng không chỉ.

Nhánh chính của một vị trí cụ thể (Github, Gitlab, Bitbucket, máy chủ cục bộ) chứa phiên bản mới nhất có thể chuyển giao.

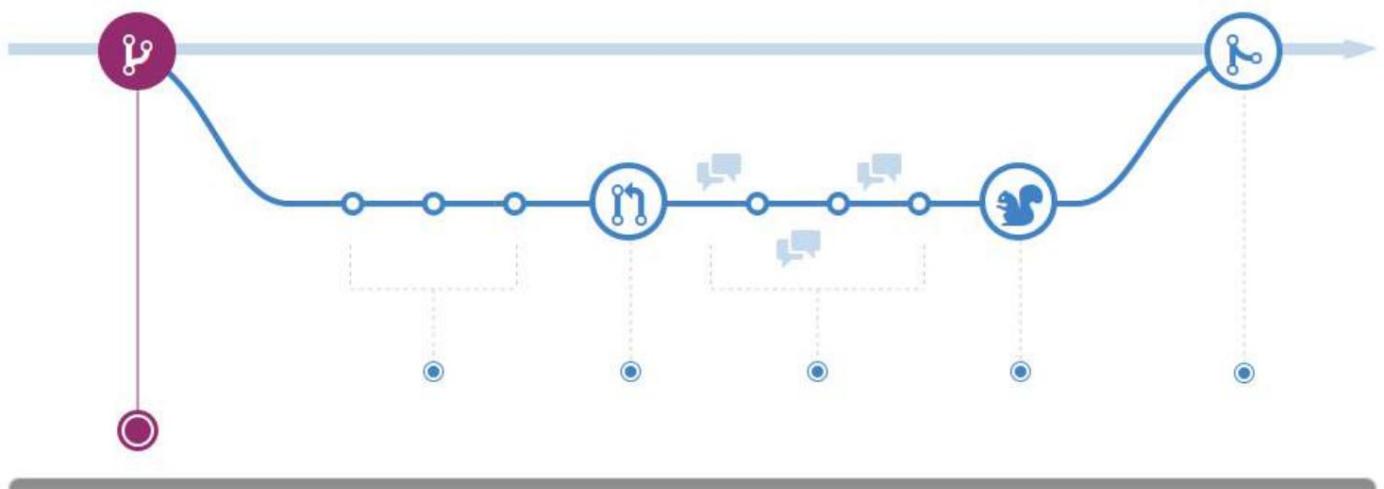
Đối với mỗi tính năng/sửa lỗi/thay đổi kiến trúc mới, mỗi nhà phát triển sẽ tạo một nhánh.

Các thay đổi xảy ra trên nhánh đó và có thể được thảo luận trong yêu cầu kéo, xem xét mã, v.v. Sau khi được chấp nhận, chúng sẽ được hợp nhất vào nhánh chính.

Toàn bộ dòng chảy của Scott Chacon:

- Mọi thứ trong nhánh chính đều có thể triển khai được
- Để làm việc với một cái gì đó mới, hãy tạo một nhánh có tên mô tả của nhánh chính (ví dụ: new-oauth2-scopes)
- Cam kết với nhánh đó cục bộ và thường xuyên đẩy công việc của bạn đến nhánh có cùng tên trên máy chủ
- Khi bạn cần phản hồi hoặc trợ giúp hoặc bạn nghĩ rằng chi nhánh đã sẵn sàng để hợp nhất, hãy mở yêu cầu kéo
- Sau khi người khác đã xem xét và đăng xuất tính năng này, bạn có thể hợp nhất nó thành tính năng chính
- Sau khi được hợp nhất và đẩy lên 'chính chủ', bạn có thể và nên triển khai ngay lập tức

Được trình bày lần đầu trên [trang web cá nhân của Scott Chacon](#).

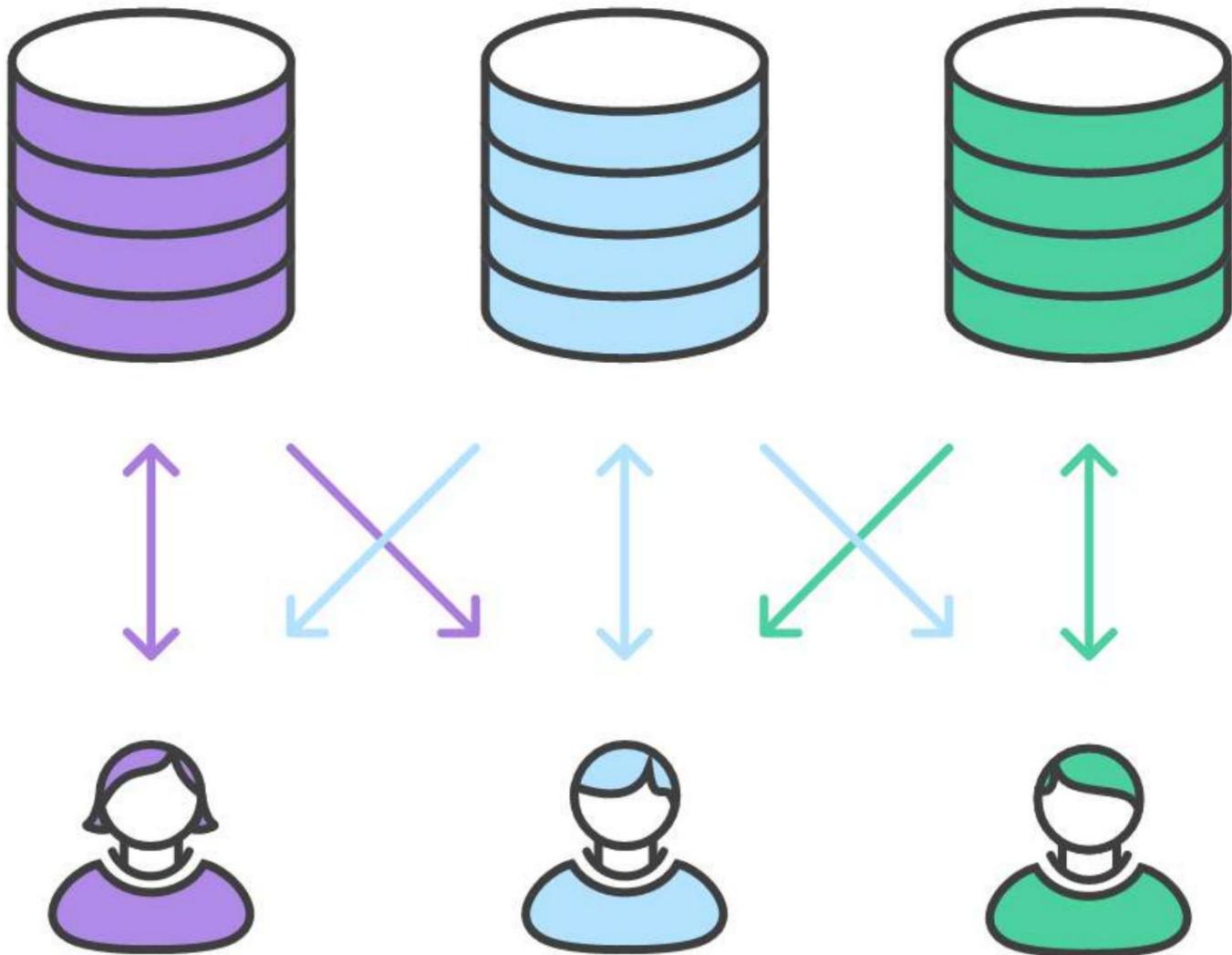


Hình ảnh được cung cấp bởi [tài liệu tham khảo GitHub Flow](#)

## Phần 22.5: Quy trình phân nhánh

Loại quy trình làm việc này về cơ bản khác với các loại quy trình khác được đề cập trong chủ đề này. Thay vì có một kho lưu trữ tập trung mà tất cả các nhà phát triển đều có quyền truy cập, mỗi nhà phát triển có kho lưu trữ riêng của mình được phân nhánh từ kho lưu trữ chính. Ưu điểm của việc này là các nhà phát triển có thể đăng lên kho lưu trữ của riêng họ thay vì kho lưu trữ chung và người bảo trì có thể tích hợp các thay đổi từ kho lưu trữ phân nhánh vào bản gốc bất cứ khi nào thích hợp.

Hình ảnh trực quan của quy trình công việc này như sau:



# Chương 23: Kéo

Thông số	Chi tiết
--im lặng	Không có tốc ký
-q	đầu ra văn bản cho --quiet
--dài dòng	đầu ra văn bản dài dòng. Đã chuyển sang lệnh tìm nạp và hợp nhất/rebase tương ứng. viết tắt
-v	cho -verbose
là --[no-]recurse-submodules[=yes on-demand no] kéo/kiểm tra)	Tìm nạp các cam kết mới cho các mô-đun con? (Không phải đây không phải là --[no-]recurse-submodules[=yes on-demand no] kéo/kiểm tra)

Không giống như việc đẩy bằng Git nơi các thay đổi cục bộ của bạn được gửi đến máy chủ của kho lưu trữ trung tâm, việc kéo bằng Git sẽ lấy mã hiện tại trên máy chủ và 'kéo' mã đó từ máy chủ của kho lưu trữ xuống máy cục bộ của bạn. Chủ đề này giải thích quy trình lấy mã từ kho lưu trữ bằng Git cũng như các tình huống mà người ta có thể gặp phải khi kéo mã khác vào bản sao cục bộ.

## Phần 23.1: Kéo các thay đổi vào kho lưu trữ cục bộ

### Kéo đơn giản

Khi bạn đang làm việc trên một kho lưu trữ từ xa (giả sử là GitHub) với người khác, đôi khi bạn sẽ muốn chia sẻ những thay đổi của mình với họ. Khi họ đã đẩy các thay đổi của mình sang kho lưu trữ từ xa, bạn có thể truy xuất những thay đổi đó bằng cách lấy từ kho lưu trữ này.

#### kéo git

Sẽ làm điều đó, trong phần lớn các trường hợp.

### Kéo từ một điều khiển từ xa hoặc chi nhánh khác

Bạn có thể lấy các thay đổi từ một điều khiển từ xa hoặc nhánh khác bằng cách chỉ định tên của chúng

#### tính năng xuất xứ git pull -A

Sẽ kéo tính năng nhánh-A gốc vào nhánh địa phương của bạn. Lưu ý rằng bạn có thể cung cấp trực tiếp URL thay vì tên từ xa và tên đối tượng như SHA cam kết thay vì tên chi nhánh.

### Kéo bằng tay

Để bắt chước hành vi của git pull, bạn có thể sử dụng git get sau đó git merge

```
git tìm nạp nguồn gốc # truy xuất các đối tượng và cập nhật các tham chiếu từ
nguồn gốc git merge Origin/feature-A # thực sự thực hiện việc hợp nhất
```

Điều này có thể cung cấp cho bạn nhiều quyền kiểm soát hơn và cho phép bạn kiểm tra nhánh từ xa trước khi hợp nhất nó. Thật vậy, sau khi tìm nạp, bạn có thể thấy các nhánh từ xa bằng git Branch -a và kiểm tra chúng bằng

```
git kiểm tra -b local-branch-name Origin/feature-A # kiểm tra chi nhánh từ xa # kiểm tra chi nhánh, thực
hiện cam kết, xóa bỏ, sửa đổi hoặc bắt cứ điều gì git kiểm tra sáp nhập các chi
nhánh # di chuyển đến chi nhánh đích
```

```
git merge local-branch-name # thực hiện hợp nhất
```

Điều này có thể rất thuận tiện khi xử lý các yêu cầu kéo.

## Phần 23.2: Cập nhật các thay đổi cục bộ

Khi có thay đổi cục bộ, lệnh `git pull` sẽ hủy báo cáo:

lỗi: Những thay đổi cục bộ của bạn đối với các tệp sau sẽ bị ghi đè bằng cách hợp nhất

Để cập nhật (như `svn update` đã làm với Subversion), bạn có thể chạy:

```
git stash
git pull --rebase git
stash pop
```

Một cách thuận tiện có thể là xác định bí danh bằng cách sử dụng:

Phiên bản < 2.9

```
git config --global alias.up '!git stash && git pull --rebase && git stash pop'
```

Phiên bản 2.9

```
git config --global alias.up 'pull --rebase --autostash'
```

Tiếp theo bạn có thể chỉ cần sử dụng:

đúng lên

## Mục 23.3: Kéo, ghi đè cục bộ

```
git tóm nạp
git reset --hard Origin/master
```

Cẩn thận: Mặc dù các cam kết bị loại bỏ bằng cách sử dụng `reset --hard` có thể được khôi phục bằng cách sử dụng `reflog` và `reset`, nhưng các thay đổi không được cam kết sẽ bị xóa vĩnh viễn.

Thay đổi `Origin` và `Master` thành điều khiển từ xa và nhánh mà bạn muốn kéo tới tương ứng, nếu chúng được đặt tên khác nhau.

## Mục 23.4: Kéo code từ xa

kéo git

## Mục 23.5: Lưu lịch sử tuyến tính khi kéo

Rebase khi kéo

Nếu bạn đang lấy các cam kết mới từ kho lưu trữ từ xa và bạn có các thay đổi cục bộ trên nhánh hiện tại thì `git` sẽ tự động hợp nhất phiên bản từ xa và phiên bản của bạn. Nếu bạn muốn giảm số lần hợp nhất trên nhánh của mình, bạn có thể yêu cầu `git` khởi động lại các cam kết của bạn trên phiên bản từ xa của nhánh.

```
git pull --rebase
```

Biến nó thành hành vi mặc định

Để biến điều này thành hành vi mặc định cho các nhánh mới được tạo, hãy gõ lệnh sau:

```
git config Branch.autosetuprebase luôn
```

Để thay đổi hành vi của một nhánh hiện có, hãy sử dụng:

```
nhánh cấu hình git.BRANCH_NAME.rebase đúng
```

Và

```
git pull --no-rebase
```

Để thực hiện thao tác kéo hợp nhất thông thường.

Kiểm tra xem có thể chuyển tiếp nhanh không

Để chỉ cho phép chuyển tiếp nhanh chỉ nhánh địa phương, bạn có thể sử dụng:

```
git pull --ff-only
```

Điều này sẽ hiển thị lỗi khi nhánh cục bộ không thể chuyển tiếp nhanh và cần được khởi động lại hoặc hợp nhất với nhánh ngược dòng.

## Mục 23.6: Kéo, "quyền bị từ chối"

Một số vấn đề có thể xảy ra nếu thư mục .git có quyền sai. Khắc phục sự cố này bằng cách đặt chủ sở hữu của thư mục .git hoàn chỉnh. Đôi khi xảy ra trường hợp người dùng khác lấy và thay đổi quyền của thư mục hoặc tệp .git .

Để khắc phục sự cố:

```
chown -R youruser:yourgroup .git/
```

# Chương 24: Cái móc

## Mục 24.1: Đẩy trước

Có sẵn trong [Git 1.8.2](#) và ở trên.

Phiên bản 1.8

Tuy nhiên, móc đẩy trước có thể được sử dụng để ngăn lực đẩy diễn ra. Những lý do khiến điều này hữu ích bao gồm: chặn các lần đẩy thủ công vô tình đến các nhánh cụ thể hoặc chặn các lần đẩy nếu quá trình kiểm tra đã thiết lập không thành công (kiểm tra đơn vị, cú pháp).

Móc đẩy trước được tạo bằng cách chỉ cần tạo một tệp có tên pre-push trong .git/hooks/ và (cảnh báo gotcha), đảm bảo tệp có thể thực thi được: `chmod +x ./git/hooks/pre-push`.

Đây là một ví dụ từ [Hannah Wolfe](#) ngăn chặn sự thúc đẩy để làm chủ:

```
#!/bin/bash

protected_branch='master'
current_branch=$(git biếu tương-ref HEAD | sed -e 's,.*/(.*)\,\1,')

if [ $protected_branch = $current_branch ] sau đó đọc -p
"Bạn

sắp đẩy chủ nhân, đó có phải là ý định của bạn không? [y|n] " echo if echo $REPLY | grep -E '^Yy$' > / -n 1 -r < /dev/tty
dev/
null then exit 0 # push sẽ thực thi

fi

exit 1 # push sẽ không thực thi nếu không

thoát 0 # đẩy sẽ thực thi
fi
```

Đây là một ví dụ từ [Volkan Unsal](#) để đảm bảo các bài kiểm tra RSpec vượt qua trước khi cho phép đẩy:

```
#!/usr/bin/env Ruby require
'pty' html_path =
"rspec_results.html" bắt đầu PTY.spawn( "rspec
spec --
format h > rspec_results.html" ) do |stdin, stdout, pid| bắt đầu stdin.each { |line| dòng in }

giải cứu Errno::EIO
end
end
giải cứu PTY::ChildExited
đặt "Thoát quá trình con!" kết thúc

# tìm hiểu xem có lỗi nào không html =
open(html_path).đọc ví dụ = html.match(
(\d+) ví dụ/)[0].to_i cứu 0 lỗi = html.match(/(\d+) lỗi/ )[0].to_i cứu 0 nếu lỗi
== 0 thì

lỗi = html.match(/(\d+) thất bại/)[0].to_i cứu 0 kết thúc đang chờ xử lý =
html.match(/(\d+) đang chờ xử lý/)[0].to_i cứu 0
```

```

nếu có lỗi.zero? đặt
"0 không thành công! #{examples} chạy, #{pending} đang chờ xử lý"
# Kết quả HTML khi chạy thử nghiệm thành công: # đặt "Xem
kết quả thông số kỹ thuật tại #{File.expand_path(html_path)}" sleep 1 exit 0 else

đặt "\aCAM KẾT THẤT BẠI!!" đặt
"Xem kết quả rspec của bạn tại #{File.expand_path(html_path)}" đặt đặt "#{errors} không
thành
công! #{examples} chạy, #{pending} đang chờ xử lý"
# Mở đầu ra HTML khi kiểm tra không thành công #
`open #{html_path}` exit 1

kết thúc

```

Như bạn có thể thấy, có rất nhiều khả năng, nhưng điều cốt lõi là thoát ra 0 nếu điều tốt xảy ra và thoát 1 nếu điều tồi tệ xảy ra. Bất cứ khi nào bạn thoát 1, thao tác đẩy sẽ bị ngăn chặn và mã của bạn sẽ ở trạng thái như trước khi chạy git Push....

Khi sử dụng mốc phía máy khách, hãy nhớ rằng người dùng có thể bỏ qua tất cả các mốc phía máy khách bằng cách sử dụng tùy chọn "--no-verify" khi nhấn nút. Nếu bạn đang dựa vào hook để thực thi quy trình, bạn có thể bị bỏng.

Tài liệu: [https://git-scm.com/docs/githooks#\\_pre\\_push](https://git-scm.com/docs/githooks#_pre_push) Mẫu chính thức:  
<https://github.com/>  
<git/git/blob/87c86dd14abe8db7d00b0df5661ef8cf147a72a3/templates/hooks--pre-push.sample>

## Phần 24.2: Xác minh bản dựng Maven (hoặc hệ thống xây dựng khác) trước khi cam kết

```
.git/hooks/pre-commit
```

```

#!/bin/sh if
[ -s pom.xml ]; sau đó echo
"Đang chạy mvn verify" mvn clean
verify if [ $? -ne
0 ]; sau đó echo "Xây dựng
Maven không thành công"
lỗi ra 1
fi
fi

```

## Mục 24.3: Tự động chuyển tiếp một số lần đẩy nhất định đến các kho lưu trữ khác

hook sau nhận có thể được sử dụng để tự động chuyển tiếp các lần đẩy đến kho lưu trữ khác.

```
$ cat .git/hooks/post-receive
```

```

#!/bin/bash

IFS=' '
trong khi đọc local_ref local_sha remote_ref remote_sha làm

echo "$remote_ref" | ví dụ: '^refs\heads\[AZ]+\-[0-9]+\$' >/dev/null && { ref=`echo $remote_ref | sed
-e 's/^refs\heads\///`
```

```
echo Chuyển tiếp nhánh tính năng sang kho lưu trữ khác: $ref git push
-q --force other_repos $ref
}
```

xong

Trong ví dụ này, biểu thức chính quy `egrep` tìm kiếm một định dạng nhánh cụ thể (ở đây: JIRA-12345 được sử dụng để đặt tên cho các vấn đề của Jira). Tất nhiên, bạn có thể bỏ phần này nếu bạn muốn chuyển tiếp tất cả các nhánh.

## Phần 24.4: Thông báo cam kết

Móc này tương tự như móc chuẩn bị-commit-msg , nhưng nó được gọi sau khi người dùng nhập thông báo cam kết chứ không phải trước đó. Điều này thường được sử dụng để cảnh báo các nhà phát triển nếu thông báo cam kết của họ có định dạng không chính xác.

Đối số duy nhất được truyền cho hook này là tên của tệp chứa thông báo. Nếu bạn không thích thông báo mà người dùng đã nhập, bạn có thể thay đổi tệp này tại chỗ (giống như chuẩn bị-commit-msg) hoặc bạn có thể hủy bỏ hoàn toàn cam kết bằng cách thoát với trạng thái khác 0.

Ví dụ sau đây được sử dụng để kiểm tra xem vé từ sau là một số có xuất hiện trong thông báo cam kết hay không

```
word="ticket [0-9]"
isPresent=$(grep -Eoh "$word" $1)

if [[ -z $isPresent ]] thì
    echo "Thông báo cam kết KO, $word bị thiếu"; lỗi ra 1; else echo
    "Commit message OK"; lỗi ra 0; fi
```

## Mục 24.5: Móc cục bộ

Móc cục bộ chỉ ảnh hưởng đến kho lưu trữ cục bộ nơi chúng cư trú. Mỗi nhà phát triển có thể thay đổi các hook cục bộ của riêng mình, vì vậy chúng không thể được sử dụng một cách đáng tin cậy như một cách để thực thi chính sách cam kết. Chúng được thiết kế để giúp các nhà phát triển dễ dàng tuân thủ các nguyên tắc nhất định và tránh các vấn đề tiềm ẩn trong tương lai.

Có sáu loại hook cục bộ: pre-commit, prepare-commit-msg, commit-msg, post-commit, post-checkout và pre-rebase.

Bốn móc đầu tiên liên quan đến các lần xác nhận và cho phép bạn có một số quyền kiểm soát đối với từng phần trong vòng đời của một lần xác nhận. Hai cái cuối cùng cho phép bạn thực hiện một số hành động bổ sung hoặc kiểm tra an toàn cho lệnh gitcheck và git rebase.

Tất cả các móc "trước-" cho phép bạn thay đổi hành động sắp diễn ra, trong khi các móc "sau-" được sử dụng chủ yếu cho các thông báo.

## Mục 24.6: Sau thanh toán

Hook này hoạt động tương tự như hook post-commit , nhưng nó được gọi bắt cứ khi nào bạn kiểm tra thành công một tham chiếu bằng `kiểm tra git`. Đây có thể là một công cụ hữu ích để xóa thư mục làm việc của bạn chứa các tệp được tạo tự động nếu không sẽ gây nhầm lẫn.

Móc này chấp nhận ba tham số:

1. tham chiếu của HEAD trước đó, 2.
- tham chiếu của HEAD mới và 3. cờ cho
- biết đó là kiểm tra nhánh hay kiểm tra tệp (tương ứng là 1 hoặc 0).

Trạng thái thoát của nó không ảnh hưởng đến lệnh `kiểm tra git`.

## Mục 24.7: Sau cam kết

Hook này được gọi ngay sau hook thông điệp cam kết. Nó không thể thay đổi kết quả của hoạt động `cam kết git`, do đó nó được sử dụng chủ yếu cho mục đích thông báo.

Tập lệnh không có tham số và trạng thái thoát của nó không ảnh hưởng đến cam kết dưới bất kỳ hình thức nào.

## Mục 24.8: Sau khi nhận

Móc này được gọi sau khi thao tác đầy thành công. Nó thường được sử dụng cho mục đích thông báo.

Tập lệnh không có tham số nhưng được gửi cùng thông tin như nhận trước thông qua đầu vào tiêu chuẩn:

```
<giá trị cũ> <giá trị mới> <tên tham chiếu>
```

## Mục 24.9: Cam kết trước

Hook này được thực thi mỗi khi bạn chạy `git commit`, để xác minh những gì sắp được cam kết. Bạn có thể sử dụng hook này để kiểm tra ảnh chụp nhanh sắp được chuyển giao.

Loại hook này hữu ích khi chạy thử nghiệm tự động nhằm đảm bảo cam kết đến không phá vỡ chức năng hiện có của dự án của bạn. Loại hook này cũng có thể kiểm tra lỗi khoảng trắng hoặc lỗi EOL.

Không có đối số nào được chuyển đến tập lệnh xác nhận trước và việc thoát với trạng thái khác 0 sẽ hủy bỏ toàn bộ cam kết.

## Mục 24.10: Chuẩn bị-cam kết-tin nhắn

Hook này được gọi sau hook pre-commit để điền vào trình soạn thảo văn bản một thông báo cam kết. Điều này thường được sử dụng để thay đổi các thông báo cam kết được tạo tự động cho các cam kết bị nén hoặc hợp nhất.

Một đến ba đối số được truyền vào hook này:

- Tên của tệp tạm thời chứa thông báo.
- Loại cam kết, thông báo (tùy
  - chọn -m hoặc -F), mẫu (tùy
  - chọn -t ), hợp nhất (nếu
  - đó là cam kết hợp nhất) hoặc xóa (nếu
  - nó đè bẹp các cam kết khác).
- Hàm băm SHA1 của cam kết có liên quan. Điều này chỉ được cung cấp nếu tùy chọn -c, -C hoặc --amend được cung cấp.

Tương tự như cam kết trước, việc thoát với trạng thái khác 0 sẽ hủy bỏ cam kết.

## Mục 24.11: Trước cuộc nổi loạn

Hook này được gọi trước khi `git rebase` bắt đầu thay đổi cấu trúc mã. Móc này thường được sử dụng để đảm bảo hoạt động rebase là phù hợp.

Móc này có 2 tham số:

1. nhánh ngược dòng mà chuỗi được phân nhánh từ đó và 2. nhánh đang được khởi động lại (trống khi khởi động lại nhánh hiện tại).

Bạn có thể hủy thao tác rebase bằng cách thoát với trạng thái khác 0.

## Mục 24.12: Nhận trước

Môc này được thực thi mỗi khi ai đó sử dụng `git push` để đẩy các cam kết vào kho lưu trữ. Nó luôn nằm trong kho lưu trữ từ xa là đích đến của lệnh đẩy chứ không phải trong kho lưu trữ gốc (cục bộ).

Môc chạy trước khi bất kỳ tài liệu tham khảo nào được cập nhật. Nó thường được sử dụng để thực thi bất kỳ loại chính sách phát triển nào.

Tập lệnh không có tham số, nhưng mỗi tham chiếu đang được đẩy sẽ được chuyển đến tập lệnh trên một dòng riêng trên đầu vào tiêu chuẩn theo định dạng sau:

```
<giá trị cũ> <giá trị mới> <tên tham chiếu>
```

## Mục 24.13: Cập nhật

Môc này được gọi sau khi nhận trước và nó hoạt động theo cách tương tự. Nó được gọi trước khi bất kỳ nội dung nào thực sự được cập nhật, nhưng được gọi riêng cho từng lượt giới thiệu được đẩy thay vì tất cả các lượt giới thiệu cùng một lúc.

Môc này chấp nhận 3 đối số sau:

- tên của ref đang được cập nhật, tên
- đối tượng cũ được lưu trong ref và tên đối
- tượng mới được lưu trong ref.

Đây chính là thông tin được chuyển đến nhận trước, nhưng vì bản cập nhật được gọi riêng cho từng lượt giới thiệu nên bạn có thể từ chối một số lượt giới thiệu trong khi vẫn cho phép những lượt giới thiệu khác.

# Chương 25: Nhân bản kho lưu trữ

## Mục 25.1: Bản sao nông

Việc sao chép một kho lưu trữ không lồ (như một dự án có lịch sử nhiều năm) có thể mất nhiều thời gian hoặc thất bại do lượng dữ liệu được chuyển. Trong trường hợp bạn không cần có sẵn toàn bộ lịch sử, bạn có thể tạo một bản sao nông:

```
git clone [repo_url] --deep 1
```

Lệnh trên sẽ chỉ tìm nạp lần xác nhận cuối cùng từ kho lưu trữ từ xa.

Xin lưu ý rằng bạn có thể không giải quyết được việc hợp nhất trong kho lưu trữ nông. Thông thường, bạn nên thực hiện ít nhất số lần cam kết mà bạn sẽ cần quay lại để giải quyết việc hợp nhất. Ví dụ: thay vào đó, để nhận được 50 lần xác nhận cuối cùng:

```
git clone [repo_url] --độ sâu 50
```

Sau này, nếu được yêu cầu, bạn có thể tìm nạp phần còn lại của kho lưu trữ:

Phiên bản 1.8.3

```
git tìm nạp --unshallow          # tương đương với gitfetch --deep=2147483647 # tìm
                                   nạp phần còn lại của kho lưu trữ
```

Phiên bản < 1.8.3

```
git tìm nạp --độ sâu=1000        # tìm nạp 1000 lần xác nhận cuối cùng
```

## Mục 25.2: Bản sao thông thường

Để tải xuống toàn bộ kho lưu trữ bao gồm toàn bộ lịch sử và tất cả các nhánh, hãy nhập:

```
bản sao git <url>
```

Ví dụ trên sẽ đặt nó vào một thư mục có cùng tên với tên kho lưu trữ.

Để tải xuống kho lưu trữ và lưu nó vào một thư mục cụ thể, hãy nhập:

```
git clone <url> [thư mục]
```

Để biết thêm chi tiết, hãy truy cập Sao chép kho lưu trữ.

## Phần 25.3: Sao chép một nhánh cụ thể

Để sao chép một nhánh cụ thể của kho lưu trữ, hãy nhập `--branch <branch name>` trước url kho lưu trữ:

```
git clone --branch <tên nhánh> <url> [thư mục]
```

Để sử dụng tùy chọn viết tắt cho `--branch`, hãy nhập `-b`. Lệnh này tải xuống toàn bộ kho lưu trữ và kiểm tra `< tên chi nhánh>`.

Để tiết kiệm dung lượng ổ đĩa, bạn có thể sao chép lịch sử chỉ dẫn đến một nhánh duy nhất bằng:

```
git clone --branch <branch_name> --single-branch <url> [thư mục]
```

Nếu `--single-branch` không được thêm vào lệnh, lịch sử của tất cả các nhánh sẽ được sao chép vào [thư mục]. Đây có thể là vấn đề với kho lưu trữ lớn.

Để hoàn tác sau cờ `--single-branch` và tìm nạp phần còn lại của kho lưu trữ, hãy sử dụng lệnh:

```
git config remote.origin.fetch "+refs/heads/*:refs/remotes/origin/*" git tìm nạp nguồn gốc
```

## Mục 25.4: Sao chép đệ quy

Phiên bản 1.6.5

```
git clone <url> --recursive
```

Sao chép kho lưu trữ và cũng sao chép tất cả các mô-đun con. Nếu bản thân các mô-đun con chứa các mô-đun con bổ sung, Git cũng sẽ sao chép các mô-đun con đó.

## Mục 25.5: Sao chép bằng proxy

Nếu bạn cần tải xuống các tệp bằng git dưới proxy, việc đặt máy chủ proxy trên toàn hệ thống là không đủ. Bạn cũng có thể thử cách sau:

```
git config --global http.proxy http://<proxy-server>:<port>/
```

# Chương 26: Trộm cắp

Tham số	Chi tiết
trình diễn	Hiển thị các thay đổi được ghi trong kho lưu trữ dưới dạng khác biệt giữa trạng thái được lưu trữ và trạng thái gốc ban đầu của nó. Khi không có <stash> nào được đưa ra, hiển thị cái mới nhất.
danh sách	Liệt kê các stash mà bạn hiện có. Mỗi stash được liệt kê với tên của nó (ví dụ: stash@{0} là stash mới nhất, stash@{1} là stash trước đó, v.v.), tên của nhánh hiện tại khi stash được tạo và một đoạn mã ngắn mô tả về cam kết mà kho lưu trữ dựa trên.
áp dụng	Xóa một trạng thái được lưu trữ duy nhất khỏi danh sách lưu trữ và áp dụng nó lên trên trạng thái cây đang làm việc hiện tại. Giống như pop, nhưng không xóa trạng thái khỏi danh sách lưu trữ.
thông thường	Loại bỏ tất cả các trạng thái được lưu trữ. Lưu ý rằng những trạng thái đó sau đó sẽ bị cắt bớt và có thể không thể phục hồi được.
làm rơi	Xóa một trạng thái được lưu trữ khỏi danh sách lưu trữ. Khi không có <stash> nào được đưa ra, nó sẽ xóa cái mới nhất. tức là stash@{0}, nếu không <stash> phải là tham chiếu nhật ký stash hợp lệ có dạng stash@{<revision>}.
tạo nên	Tạo một stash (là một đối tượng cam kết thông thường) và trả về tên đối tượng của nó mà không lưu trữ nó ở bất kỳ đâu trong không gian tên ref. Điều này nhằm mục đích hữu ích cho các tập lệnh. Đó có thể không phải là lệnh bạn muốn sử dụng; xem "lưu" ở trên.
cửa hàng	Lưu trữ một stash nhất định được tạo thông qua git stash create (là một cam kết hợp nhất lơ lửng) trong ref stash, cập nhật reflog stash. Điều này nhằm mục đích hữu ích cho các tập lệnh. Đó có thể không phải là lệnh bạn muốn sử dụng; xem "lưu" ở trên.

## Phần 26.1: Stash là gì?

Khi làm việc trên một dự án, bạn có thể đang thực hiện được một nửa quá trình thay đổi nhánh tính năng khi có một lỗi phát sinh đối với bản chính. Bạn chưa sẵn sàng cam kết mã của mình nhưng bạn cũng không muốn mất các thay đổi của mình. Đây là lúc `git stash` phát huy tác dụng.

Chạy `git status` trên một nhánh để hiển thị những thay đổi chưa được cam kết của bạn:

```
(master) $ git status Trên
nhánh master Nhánh
của bạn được cập nhật với 'origin/master'.
Những thay đổi không được tổ chức cho cam kết:
  (sử dụng "git add <file>..." để cập nhật nội dung sẽ được cam kết) (sử dụng
  "gitcheck -- <file>..." để loại bỏ các thay đổi trong thư mục làm việc)

    đã sửa đổi: business/com/test/core/actions/Photo.c

không có thay đổi nào được thêm vào cam kết (sử dụng "git add" và/hoặc "git commit -a")
```

Sau đó chạy `git stash` để lưu những thay đổi này vào ngăn xếp:

```
(chính) $ git stash
Đã lưu trạng thái chỉ mục và thư mục làm việc WIP trên bản gốc: 2f2a6e1
Hợp nhất yêu cầu kéo số 1 từ nhánh kiểm tra/kiểm tra
HEAD hiện ở mức 2f2a6e1 Hợp nhất yêu cầu kéo số 1 từ nhánh thử nghiệm/kiểm tra
```

Nếu bạn đã thêm các tập tin vào thư mục làm việc của mình thì những tập tin này cũng có thể được lưu trữ. Bạn chỉ cần giao diện chúng đầu tiên.

```
(chính) $ git stash
Đã lưu thư mục làm việc và trạng thái chỉ mục WIP trên bản gốc: (master)
$ git status
Trên nhánh chính
Các tập tin không bị theo dõi:
  (sử dụng "git add <file>..." để bao gồm những gì sẽ được cam kết)
```

```
NewPhoto.c
```

```
không có gì được thêm vào cam kết nhưng có các tệp không được theo dõi (sử dụng "git add" để theo dõi) (master) $ git stage NewPhoto.c
(master) $ git stash Đã lưu thư mục làm việc và trạng thái chỉ mục WIP trên master: (master) $
git status Trên nhánh master
không có gì để cam kết, làm việc sạch cây (master) $
```

Thư mục làm việc của bạn bây giờ đã sạch mọi thay đổi bạn đã thực hiện. Bạn có thể thấy điều này bằng cách chạy lại [trạng thái git](#):

```
(master) $ git status Trên
nhánh master Nhánh
của bạn được cập nhật với 'origin/master'. không có gì để cam kết, thư mục làm việc sạch sẽ
```

Để áp dụng stash cuối cùng, hãy chạy `git stash apply` (ngoài ra, bạn có thể áp dụng và xóa stash cuối cùng đã thay đổi bằng `git stash pop`):

```
(master) $ git stash apply Trên
nhánh master Nhánh
của bạn được cập nhật với 'origin/master'.
Những thay đổi không được tổ chức cho cam kết:
(sử dụng "git add <file>..." để cập nhật nội dung sẽ được cam kết) (sử dụng
"gitcheck -- <file>..." để loại bỏ các thay đổi trong thư mục làm việc)

đã sửa đổi: business/com/test/core/actions/Photo.c

không có thay đổi nào được thêm vào cam kết (sử dụng "git add" và/hoặc "git commit -a")
```

Tuy nhiên, lưu ý rằng việc lưu trữ đó không ghi nhớ nhánh bạn đang làm việc. Trong các ví dụ trên, người dùng đang lưu trữ bản gốc. Nếu họ chuyển sang nhánh dev , dev và chạy `git stash apply` thì stash cuối cùng sẽ được đặt trên nhánh dev .

```
(chính) $ git kiểm tra -b dev
Đã chuyển sang nhánh mới 'dev' (dev) $
git stash áp dụng
Trên nhánh dev
Những thay đổi không được tổ chức cho cam kết:
(sử dụng "git add <file>..." để cập nhật nội dung sẽ được cam kết) (sử dụng
"gitcheck -- <file>..." để loại bỏ các thay đổi trong thư mục làm việc)

đã sửa đổi: business/com/test/core/actions/Photo.c

không có thay đổi nào được thêm vào cam kết (sử dụng "git add" và/hoặc "git commit -a")
```

## Phần 26.2: Tạo kho lưu trữ

Lưu trạng thái hiện tại của thư mục làm việc và chỉ mục (còn được gọi là khu vực tổ chức) vào một chồng các ngăn chứa.

`kho git`

Để bao gồm tất cả các tệp không được theo dõi trong kho lưu trữ, hãy sử dụng cờ `--include-untracked` hoặc `-u` .

`git stash --include-untracked`

Để bao gồm một tin nhắn với kho lưu trữ của bạn để sau này dễ nhận dạng hơn

```
git stash lưu "<bắt cứ tin nhắn nào>"
```

Để rời khỏi khu vực tổ chức ở trạng thái hiện tại sau khi lưu trữ, hãy sử dụng cờ `--keep-index` hoặc `-k`.

```
git stash --keep-index
```

### Phần 26.3: Áp dụng và xóa stash

Để áp dụng ngay cuối cùng và xóa nó khỏi ngăn xếp - hãy gõ:

```
git stash pop
```

Để áp dụng stash cụ thể và xóa nó khỏi ngăn xếp - hãy gõ:

```
git stash pop stash@{n}
```

### Mục 26.4: Áp dụng stash mà không xóa nó

Áp dụng ngay cuối cùng mà không xóa nó khỏi ngăn xếp

```
git stash áp dụng
```

Hoặc một kho cụ thể

```
git stash áp dụng stash@{n}
```

### Mục 26.5: Hiển thị kho lưu trữ

Hiển thị các thay đổi được lưu trong lần lưu trữ cuối cùng

```
chương trình lưu trữ git
```

Hoặc một kho cụ thể

```
git stash hiển thị stash@{n}
```

Để hiển thị nội dung của các thay đổi được lưu cho kho cụ thể

```
chương trình git stash -p stash@{n}
```

### Mục 26.6: Lưu trữ một phần

Nếu bạn chỉ muốn lưu trữ một số khác biệt trong bộ làm việc của mình, bạn có thể sử dụng lưu trữ một phần.

```
kho git -p
```

Và sau đó chọn những phần cần lưu trữ một cách tương tác.

Kể từ phiên bản 2.13.0, bạn cũng có thể tránh ché độ tương tác và tạo một phần kho lưu trữ với thông số đường dẫn bằng cách sử dụng từ khóa push mới .

```
git stash Push -m "Stash một phần của tôi" -- app.config
```

## Phần 26.7: Liệt kê các stash đã lưu

danh sách kho git

Điều này sẽ liệt kê tất cả các stash trong ngăn xếp theo thứ tự thời gian đảo ngược.

Bạn sẽ nhận được một danh sách trông giống như thế này:

```
stash@{0}: WIP trên master: 67a4e01 Hợp nhất các thử nghiệm thành develop
stash@{1}: WIP trên master: 70f0d95 Thêm vai trò người dùng vào localStorage khi người dùng đăng nhập
```

Bạn có thể tham khảo stash cụ thể theo tên của nó, ví dụ stash@{1}.

## Mục 26.8: Di chuyển công việc đang thực hiện của bạn sang chi nhánh khác

Nếu trong khi làm việc, bạn nhận ra mình đang ở sai nhánh và bạn chưa tạo bất kỳ cam kết nào, bạn có thể dễ dàng chuyển công việc của mình sang nhánh sửa bằng cách sử dụng stashing:

```
git stash
git kiểm tra nhánh chính xác git
stash pop
```

Hãy nhớ `git stash pop` sẽ áp dụng stash cuối cùng và xóa nó khỏi danh sách stash. Để giữ stash trong danh sách và chỉ áp dụng cho một số nhánh bạn có thể sử dụng:

`git stash` áp dụng

## Mục 26.9: Xóa kho lưu trữ

Xóa tất cả kho lưu trữ

`git lưu trữ rõ ràng`

Loại bỏ stash cuối cùng

thả kho git

Hoặc một kho cụ thể

`git stash` thả stash@{n}

## Mục 26.10: Áp dụng một phần kho khi thanh toán

Bạn đã tạo một kho lưu trữ và chỉ muốn kiểm tra một số tệp trong kho lưu trữ đó.

`git kiểm tra stash@{0} -- myfile.txt`

## Mục 26.11: Khôi phục các thay đổi trước đó từ kho lưu trữ

Để nhận được stash gần đây nhất của bạn sau khi chạy `git stash`, hãy sử dụng

```
git stash áp dụng
```

Để xem danh sách các kho lưu trữ của bạn, hãy sử dụng

```
danh sách kho git
```

Bạn sẽ nhận được một danh sách trông giống như thế này

```
stash@{0}: WIP trên master: 67a4e01 Hợp nhất các thử nghiệm thành develop
stash@{1}: WIP trên master: 70f0d95 Thêm vai trò người dùng vào localStorage khi người dùng đăng nhập
```

Chọn một kho git khác để khôi phục bằng số hiển thị cho kho lưu trữ bạn muốn

```
git stash áp dụng stash@{2}
```

## Mục 26.12: Tích trữ tương tác

Stashing lấy trạng thái bản của thư mục làm việc của bạn - tức là các tệp được theo dõi đã sửa đổi và các thay đổi theo giai đoạn của bạn - và lưu nó vào một chồng các thay đổi chưa hoàn thành mà bạn có thể áp dụng lại bất kỳ lúc nào.

Chỉ lưu trữ các tập tin đã sửa đổi:

Giả sử bạn không muốn lưu trữ các tệp được dàn dựng và chỉ lưu trữ các tệp đã sửa đổi để bạn có thể sử dụng:

```
git stash --keep-index
```

Nó sẽ chỉ lưu trữ các tập tin đã sửa đổi.

Xóa các tập tin không bị theo dõi:

Stash không bao giờ lưu các tệp không bị theo dõi, nó chỉ lưu các tệp đã sửa đổi và phân loại. Vì vậy, giả sử nếu bạn cũng cần lưu trữ các tệp không bị theo dõi thì bạn có thể sử dụng tệp này:

```
kho git -u
```

điều này sẽ theo dõi các tập tin không bị theo dõi, được sắp xếp và sửa đổi.

Chỉ lưu trữ một số thay đổi cụ thể:

Giả sử bạn chỉ cần lưu trữ một phần mã từ tệp hoặc chỉ một số tệp từ tất cả các tệp đã sửa đổi và được lưu trữ thì bạn có thể thực hiện như thế này:

```
git stash --patch
```

Git sẽ không lưu trữ mọi thứ được sửa đổi mà thay vào đó sẽ nhắc bạn một cách tương tác những thay đổi nào bạn muốn lưu trữ và những thay đổi nào bạn muốn giữ trong thư mục làm việc của mình.

## Mục 26.13: Khôi phục kho bị đánh rơi

Nếu bạn vừa mới bật nó lên và terminal vẫn mở, bạn vẫn sẽ có giá trị bấm được in bởi `git stash` hiện lên màn hình:

```
$ git stash pop
```

[.]

Đã loại bỏ ref/stash@{0} (2ca03e22256be97f9e40f08e6d6773c7d41dbfd1)

(Lưu ý rằng git stash drop cũng tạo ra dòng tương tự.)

Nếu không, bạn có thể tìm thấy nó bằng cách sử dụng:

```
git fsck --no-reflog | awk '/dangling commit/ {print $3}'
```

Điều này sẽ hiển thị cho bạn tất cả các cam kết ở đầu biểu đồ cam kết không còn được tham chiếu từ bất kỳ nhánh hoặc thẻ nào nữa - mọi cam kết bị mất, bao gồm mọi cam kết stash mà bạn từng tạo, sẽ ở đâu đó trong biểu đồ đó.

Cách dễ nhất để tìm stash commit mà bạn muốn có lẽ là chuyển danh sách đó cho gitk:

```
gitk --all $( git fsck --no-reflog | awk '/dangling commit/ {print $3}' )
```

Thao tác này sẽ khởi chạy trình duyệt kho lưu trữ hiển thị cho bạn mọi cam kết trong kho lưu trữ từ trước đến nay, bất kể nó có có thẻ truy cập được hay không.

Bạn có thể thay thế gitk ở đó bằng một cái gì đó như `git log --graph --oneline --decorate` nếu bạn thích một biểu đồ đẹp trên bảng điều khiển hơn một ứng dụng GUI riêng biệt.

Để phát hiện các cam kết, hãy tìm các thông báo cam kết có dạng sau:

WIP trên somebranch: commithash Một số thông báo cam kết cũ

Khi bạn biết hàm bấm của cam kết mà bạn muốn, bạn có thể áp dụng nó dưới dạng stash:

```
git stash áp dụng sh_hash
```

Hoặc bạn có thể sử dụng menu ngữ cảnh trong gitk để tạo các nhánh cho bất kỳ cam kết nào không thể truy cập được mà bạn quan tâm.

Sau đó, bạn có thể làm bất cứ điều gì bạn muốn với chúng bằng cách tắt cả các công cụ thông thường. Khi bạn đã hoàn tất, chỉ cần thoát bay những cành cây đó một lần nữa.

# Chương 27: Cây con

## Phần 27.1: Tạo, kéo và gửi lại cây con

### Tạo cây con

Thêm một điều khiển từ xa mới có tên plugin trả đến kho lưu trữ của plugin:

```
git từ xa thêm plugin https://path.to/remotes/plugin.git
```

Sau đó, tạo một cây con chỉ định các plugin/demo tiền tố thư mục mới. plugin là tên từ xa và master để cập đến nhánh chính trên kho lưu trữ của cây con:

```
git subtree thêm --prefix=plugins/demo plugin master
```

### Kéo cập nhật cây con

Kéo các cam kết bình thường được thực hiện trong plugin:

```
git subtree pull --prefix=plugins/demo plugin master
```

### Cập nhật cây con backport

1. Chỉ định các cam kết được thực hiện trong siêu dự án sẽ được nhập lại:

```
git commit -am "những thay đổi mới sẽ được chuyển ngược lại"
```

2. Kiểm tra nhánh mới để hợp nhất, thiết lập để theo dõi kho lưu trữ cây con:

```
kiểm tra git -b plugin/master cảng sau
```

3. Cherry-pick backport:

```
git Cherry-pick -x --strategy=cây con chính
```

4. Đẩy các thay đổi về nguồn plugin:

```
cảng sau git push plugin:master
```

# Chương 28: Đổi tên

Tham số

Chi tiết

`-f` hoặc `--force` Buộc đổi tên hoặc di chuyển tệp ngay cả khi mục tiêu tồn tại

## Phần 28.1: Đổi tên thư mục

Để đổi tên thư mục từ `oldName` thành `newName`

thư mục `git mvToFolder/oldName` thư mục `ToFolder/newName`

Tiếp theo là `git commit` và/hoặc `git push`

Nếu lỗi này xảy ra:

gây tử vong: đổi tên 'directoryToFolder/oldName' không thành công: Đổi số không hợp lệ

Sử dụng lệnh sau:

thư mục `git mvToFolder/oldName temp &&` thư mục `git mv` tạm thời `ToFolder/newName`

## Mục 28.2: đổi tên nhánh cục bộ và nhánh từ xa

cách dễ nhất là kiểm tra chi nhánh địa phương:

`kiểm tra git old_branch`

sau đó đổi tên nhánh cục bộ, xóa điều khiển từ xa cũ và đặt nhánh mới được đổi tên thành ngược dòng:

```
nhánh git -m new_branch git push
Origin :old_branch git push --set-
upstream Origin new_branch
```

## Mục 28.3: Đổi tên chi nhánh địa phương

Bạn có thể đổi tên nhánh trong kho lưu trữ cục bộ bằng lệnh này:

`nhánh git -m old_name new_name`

# Chương 29: Đẩy lùi

## Tham số

## Chi tiết

`--lực lượng` Ghi đè giới thiệu từ xa để khớp với giới thiệu địa phương của bạn. Có thể khiêm kho lưu trữ từ xa mất các cam kết, vì vậy hãy cần thận khi sử dụng.

`--verbose` Chạy bằng lời nói.

`<remote>` Kho lưu trữ từ xa là đích đến của thao tác đẩy. `<refspec>...` Chỉ định tham chiếu từ xa nào sẽ cập nhật với tham chiếu hoặc đổi tượng cục bộ nào.

Sau khi thay đổi, dàn dựng và xác nhận mã bằng Git, việc đẩy là cần thiết để cung cấp các thay đổi của bạn cho người khác và chuyển các thay đổi cục bộ của bạn đến máy chủ kho lưu trữ. Chủ đề này sẽ đề cập đến cách đẩy mã đúng cách bằng Git.

## Mục 29.1: Đẩy một đối tượng cụ thể đến một nhánh từ xa

### Cú pháp chung

```
git push <remotename> <object>:<remotebranchname>
```

### Ví dụ

```
git Push Origin master:wip-yourname
```

Sẽ đẩy nhánh chính của bạn đến nhánh gốc `wip-yourname` (hầu hết thời gian, là kho lưu trữ mà bạn đã nhân bản từ đó).

### Xóa chi nhánh từ xa

Xóa nhánh từ xa tương đương với việc đẩy một đối tượng trống vào nó.

```
git push <remotename> :<remotebranchname>
```

### Ví dụ

```
git đẩy nguồn gốc :wip-yourname
```

Sẽ xóa nhánh từ xa `wip-yourname`

Thay vì sử dụng dấu hai chấm, bạn cũng có thể sử dụng cờ `--delete`, cờ này dễ đọc hơn trong một số trường hợp.

### Ví dụ

```
git push Origin --delete wip-yourname
```

### Đẩy một cam kết duy nhất

Nếu bạn có một cam kết duy nhất trong nhánh của mình mà bạn muốn đẩy tới một điều khiêm từ xa mà không cần đẩy bất cứ thứ gì khác, bạn có thể sử dụng cách sau

```
git push <remotename> <commit SHA>:<remotebranchname>
```

### Ví dụ

Giả sử một lịch sử git như thế này

```
eeb32bc Cam kết 1 - đã được đẩy 347d700 Cam
kết 2 - muốn đẩy e539af8 Cam kết 3 - chỉ
cục bộ 5d339db Cam kết 4 - chỉ cục bộ
```

để chỉ đẩy cam kết 347d700 tới chủ từ xa , hãy sử dụng lệnh sau

```
git đẩy nguồn gốc 347d700: master
```

## Mục 29.2: Đẩy

**đẩy git**

sẽ đẩy mã của bạn lên thượng nguồn hiện tại của bạn. Tùy thuộc vào cấu hình đẩy, nó sẽ đẩy mã từ nhánh hiện tại của bạn (mặc định trong Git 2.x) hoặc từ tất cả các nhánh (mặc định trong Git 1.x).

Chỉ định kho lưu trữ từ xa

Khi làm việc với git, việc có nhiều kho lưu trữ từ xa có thể rất hữu ích. Để chỉ định một kho lưu trữ từ xa để đẩy tới, chỉ cần thêm tên của nó vào lệnh.

```
nguồn gốc đẩy git
```

Chỉ định chi nhánh

Để đẩy đến một nhánh cụ thể, hãy nói feature\_x:

```
git đẩy nguồn gốc tính năng_x
```

Đặt nhánh theo dõi từ xa

Trừ khi nhánh bạn đang làm việc ban đầu đến từ một kho lưu trữ từ xa, nếu không chỉ sử dụng `git push` sẽ không hoạt động trong lần đầu tiên. Bạn phải thực hiện lệnh sau để yêu cầu git đẩy nhánh hiện tại đến một tổ hợp nhánh/điều khiển từ xa cụ thể

```
git push --set-upstream Origin master
```

Ở đây, master là tên nhánh trên nguồn gốc từ xa. Bạn có thể sử dụng -u làm cách viết tắt cho `--set-upstream`.

Đẩy sang kho lưu trữ mới

Để đẩy tới kho lưu trữ mà bạn chưa tạo hoặc trống:

1. Tạo kho lưu trữ trên GitHub (nếu có)
2. Sao chép url được cung cấp cho bạn, dưới dạng `https://github.com/USERNAME/REPO_NAME.git`
3. Đi tới kho lưu trữ cục bộ của bạn và thực thi `git remote add Origin URL`
  - Để xác minh nó đã được thêm, hãy chạy `git remote -v`
4. Chạy `git push Origin master`

Mã của bạn bây giờ sẽ có trên GitHub

Để biết thêm thông tin, hãy xem Thêm kho lưu trữ từ xa

## Giải trình

Mã đầy có nghĩa là git sẽ phân tích sự khác biệt giữa các cam kết cục bộ và từ xa của bạn và gửi chúng để viết lên thượng nguồn. Khi quá trình đầy thành công, kho lưu trữ cục bộ và kho lưu trữ từ xa của bạn sẽ được đồng bộ hóa và những người dùng khác có thể thấy các cam kết của bạn.

Để biết thêm chi tiết về các khái niệm "ngược dòng" và "hạ lưu", xem Ghi chú.

## Mục 29.3: Đẩy mạnh

Đôi khi, khi bạn có các thay đổi cục bộ không tương thích với các thay đổi từ xa (tức là khi bạn không thể chuyển tiếp nhanh nhánh từ xa hoặc nhánh từ xa không phải là tổ tiên trực tiếp của nhánh cục bộ của bạn), cách duy nhất để đẩy các thay đổi của bạn là đẩy mạnh.

`git đầy -f`

hoặc

`git đầy --lực lượng`

### Ghi chú quan trọng

Thao tác này sẽ ghi đè mọi thay đổi từ xa và điều khiển từ xa của bạn sẽ khớp với địa phương của bạn.

Chú ý: Sử dụng lệnh này có thể khiến kho lưu trữ từ xa bị mất các cam kết. Hơn nữa, bạn không nên thực hiện ép buộc nếu bạn đang chia sẻ kho lưu trữ từ xa này với người khác, vì lịch sử của họ sẽ giữ lại mọi cam kết bị ghi đè, do đó khiến công việc của họ không đồng bộ với kho lưu trữ từ xa.

Theo nguyên tắc chung, chỉ đẩy mạnh khi:

- Không ai ngoại trừ bạn đã thực hiện các thay đổi mà bạn đang cố ghi đè. Bạn có thể
- buộc mọi người sao chép một bản sao mới sau khi bị ép buộc và bắt mọi người áp dụng các thay đổi của họ cho nó (mọi người có thể ghét bạn vì điều này).

## Mục 29.4: Thẻ đầy

`git đầy --tags`

Đẩy tất cả các thẻ git trong kho lưu trữ cục bộ không nằm trong kho lưu trữ từ xa.

## Mục 29.5: Thay đổi hành vi đầy mặc định

Current cập nhật nhánh trên kho lưu trữ từ xa có chung tên với nhánh đang hoạt động hiện tại.

`cấu hình git Push.default hiện tại`

Đẩy đơn giản đến nhánh ngược dòng, nhưng sẽ không hoạt động nếu nhánh ngược dòng được gọi là cái gì khác.

`cấu hình git Push.default đơn giản`

Upstream dây tới nhánh upstream, bắt kè nó được gọi là gì.

```
git config Push.default ngược dòng
```

Việc so khớp sẽ đầy tất cả các nhánh khớp trên cục bộ và git config push.default từ xa ngược dòng

Sau khi bạn đã đặt kiểu ưa thích, hãy sử dụng

```
đây git
```

để cập nhật kho lưu trữ từ xa.

# Chương 30: Nội bộ

## Mục 30.1: Repo

Kho lưu trữ **git** là cấu trúc dữ liệu trên đĩa lưu trữ siêu dữ liệu cho một tập hợp các tệp và thư mục.

Nó nằm trong thư mục `.git/` của dự án của bạn. Mỗi khi bạn chuyển dữ liệu sang git, nó sẽ được lưu trữ ở đây. Ngược lại, `.git/` chứa mọi cam kết.

Cấu trúc cơ bản của nó là như thế này:

```
.git/
  object/
    refs/
```

## Mục 30.2: Đối tượng

**git** về cơ bản là một kho lưu trữ khóa-giá trị. Khi bạn thêm dữ liệu vào **git**, nó sẽ xây dựng một đối tượng và sử dụng hàm băm SHA-1 của nội dung đối tượng làm khóa.

Do đó, bất kỳ nội dung nào trong **git** đều có thể được tra cứu bằng hàm băm của nó:

```
tập tin git cat -p 4bb6f98
```

Có 4 loại đối tượng:

- bāi
- cāy
- lām
- nhān

## Mục 30.3: HEAD ref

HEAD là một ref đặc biệt. Nó luôn trỏ đến đối tượng hiện tại.

Bạn có thể biết nó hiện đang trỏ đến đâu bằng cách kiểm tra tệp `.git/HEAD`.

Thông thường, HEAD trỏ tới một ref khác:

```
$cat .git/HEAD
tham chiếu: refs/heads/mainline
```

Nhưng nó cũng có thể trỏ trực tiếp tới một đối tượng:

```
$ cat .git/HEAD
4bb6f98a223abc9345a0cef9200562333
```

Đây là cái được gọi là "đầu tách rời" - bởi vì HEAD không được gắn vào (chỉ vào) bất kỳ tham chiếu nào, mà chỉ trực tiếp vào một đối tượng.

## Mục 30.4: Tài liệu tham khảo

Một ref về cơ bản là một con trỏ. Đó là một cái tên trỏ đến một đối tượng. Ví dụ,

"chủ" -> 1a410e...

Chúng được lưu trữ trong ` .git/refs/heads/` trong các tệp văn bản thuần túy.

```
$ cat .git/refs/heads/mainline
4bb6f98a223abc9345a0cef9200562333
```

Đây thường là những gì được gọi là chi nhánh. Tuy nhiên, bạn sẽ lưu ý rằng trong git không có thư mục là nhánh - chỉ có ref.

Giờ đây, bạn có thể điều hướng git hoàn toàn bằng cách chuyển trực tiếp đến các đối tượng khác nhau bằng hàm băm của chúng. Nhưng điều này sẽ vô cùng bất tiện. Một ref cung cấp cho bạn một cái tên thuận tiện để cập đến các đối tượng. Sẽ dễ dàng hơn nhiều khi yêu cầu git đi đến một địa điểm cụ thể theo tên thay vì theo hàm băm.

## Mục 30.5: Đối tượng cam kết

Một cam kết có lẽ là loại đối tượng quen thuộc nhất với người dùng git, vì đó là những gì họ quen với việc tạo bằng các lệnh git commit.

Tuy nhiên, cam kết không trực tiếp chứa bất kỳ tệp hoặc dữ liệu đã thay đổi nào. Đúng hơn, nó chứa hầu hết siêu dữ liệu và các con trỏ tới các đối tượng khác chứa nội dung thực tế của cam kết.

Một cam kết chứa một số điều:

- băm của một cái cây
- hàm băm của cam kết cha tên
- tác giả/email, tên người cam kết/thông báo cam kết
- email

Bạn có thể xem nội dung của bất kỳ cam kết nào như thế này:

```
$ git cat-file cam kết 5bac93 cây
04d1daef... cha mẹ
b7850ef5... tác giả
Geddy Lee <glee@rush.com> người ủy thác
Neil Peart <npeart@rush.com>
```

Cam kết đầu tiên!

Cây

Một lưu ý rất quan trọng là các đối tượng cây lưu trữ **MỌI** tệp trong dự án của bạn và nó lưu trữ toàn bộ tệp không khác biệt. Điều này có nghĩa là mỗi cam kết chứa ảnh chụp nhanh của toàn bộ dự án\*.

\*Về mặt kỹ thuật, chỉ những tập tin đã thay đổi mới được lưu trữ. Nhưng đây là một chi tiết thực hiện hiệu quả hơn. Từ góc độ thiết kế, một cam kết nên được coi là chứa một bản sao hoàn chỉnh của dự án.

Cha mẹ

Dòng gốc chứa hàm băm của một đối tượng cam kết khác và có thể được coi là "con trỏ gốc" trỏ đến "cam kết trước đó". Điều này ngầm tạo thành một biểu đồ các cam kết được gọi là biểu đồ cam kết. Cụ thể, đó là **biểu đồ chu kỳ có hướng** (hoặc DAG).

## Mục 30.6: Đối tượng cây

Về cơ bản, cây đại diện cho một thư mục trong hệ thống tệp truyền thống: các thùng chứa lồng nhau cho các tệp hoặc thư mục khác.

Một cây có chứa:

- 0 đối tượng blob trở lên 0
- đối tượng cây trở lên

Giống như bạn có thể sử dụng `ls` hoặc `dir` để liệt kê nội dung của một thư mục, bạn có thể liệt kê nội dung của một đối tượng cây.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b .gitignore 100644
blob cc0956f1 Makefile
040000 cây 92e1ca7e src
...
```

Bạn có thể tra cứu các tệp trong một cam kết bằng cách trước tiên hãy tìm hàm băm của cây trong cam kết đó, sau đó xem xét nó cây:

```
$ git cat-file cam kết cây 4bb6f93a
07b1a631
cha mẹ...tác
giả...
người cam kết ...

$ git cat-file -p 07b1a631
100644 blob b91bba1b .gitignore 100644
blob cc0956f1 Makefile 040000 cây
92e1ca7e src
...
```

## Phần 30.7: Đối tượng Blob

Một blob chứa nội dung tệp nhị phân tùy ý. Thông thường, nó sẽ là văn bản thô như mã nguồn hoặc một bài viết trên blog.

Nhưng nó có thể dễ dàng là byte của tệp PNG hoặc bất kỳ thứ gì khác.

Nếu bạn có hàm băm của một đốm màu, bạn có thể xem nội dung của nó.

```
$ git cat-file -p d429810 gói
com.example.project

lớp Foo {
    ...
}
...
```

Ví dụ: bạn có thể duyệt một cây như trên và sau đó xem một trong các đốm màu trong đó.

```
$ git cat-file -p 07b1a631
100644 blob b91bba1b .gitignore 100644
blob cc0956f1 Makefile 040000 cây
92e1ca7e src 100644 blob
cae391ff README.txt

$ git cat-file -p cae391ff Chào
mừng đến với dự án của tôi! Đây là tập tin README
```

...

## Phần 30.8: Tạo cam kết mới

Lệnh `git commit` thực hiện một số điều:

1. Tạo các đốm màu và cây để thể hiện thư mục dự án của bạn - được lưu trữ trong `.git/objects`
2. Tạo một đối tượng cam kết mới với thông tin tác giả, thông báo cam kết và `cây` gốc từ bước 1 - cũng được lưu trữ trong `.git/objects`
3. Cập nhật tham chiếu HEAD trong `.git/HEAD` thành hàm băm của cam kết mới được tạo

Điều này dẫn đến một ảnh chụp nhanh mới về dự án của bạn được thêm vào `git` được kết nối với trạng thái trước đó.

## Mục 30.9: Di chuyển ĐẦU

Khi bạn chạy `git checkout` trên một cam kết (được chỉ định bằng hàm băm hoặc `ref`), bạn đang yêu cầu `git` làm cho thư mục làm việc của bạn trông giống như khi chụp ảnh nhanh.

1. Cập nhật các tệp trong thư mục làm việc để khớp với `cây` bên trong cam kết 2. Cập nhật HEAD để trở đến hàm băm hoặc `ref` được chỉ định

## Mục 30.10: Di chuyển giới thiệu xung quanh

Chạy `git reset --hard` di chuyển `ref` tới hàm băm/`ref` được chỉ định.

Di chuyển MyBranch sang b8dc53:

```
$ gitcheck MyBranch # di chuyển HEAD tới MyBranch $ git reset --
hard b8dc53 # làm cho MyBranch trở tới b8dc53
```

## Mục 30.11: Tạo Ref mới

Chạy `gitcheck -b <refname>` sẽ tạo một `ref` mới trỏ đến cam kết hiện tại.

```
$ mèo .git/head
1f324a

$ git kiểm tra -b TestBranch

$ cat .git/refs/heads/TestBranch 1f324a
```

# Chương 31: git-tfs

## Mục 31.1: bản sao git-tfs

Điều này sẽ tạo một thư mục có cùng tên với dự án, tức là /My.Project.Name

```
$ git tfs bản sao http://tfs:8080/tfs/DefaultCollection/ $/My.Project.Name
```

## Phần 31.2: git-tfs sao chép từ kho git trần

Nhân bản từ kho git nhanh hơn mười lần so với nhân bản trực tiếp từ TFVS và hoạt động tốt trong môi trường nhóm. Ít nhất một thành viên trong nhóm sẽ phải tạo kho lưu trữ git trần bằng cách thực hiện sao chép git-tfs thông thường trước. Sau đó, kho lưu trữ mới có thể được khởi động để hoạt động với TFVS.

```
$ git clone x:/fileshare/git/My.Project.Name.git $ cd  
My.Project.Name $ git  
tfs bootstrap $ git tfs  
pull
```

## Phần 31.3: cài đặt git-tfs qua Chocolatey

Phần sau giả định rằng bạn sẽ sử dụng kdiff3 để phân biệt tệp và mặc dù không cần thiết nhưng đó là một ý tưởng hay.

```
C:\> choco cài đặt kdiff3
```

Git có thể được cài đặt trước để bạn có thể nêu bất kỳ tham số nào bạn muốn. Ở đây tất cả các công cụ Unix cũng được cài đặt và 'NoAutoCrLf' có nghĩa là thanh toán nguyên trạng, cam kết nguyên trạng.

```
C:\> choco cài đặt git -params '"/GitAndUnixToolsOnPath /NoAutoCrLf"'
```

Đây là tất cả những gì bạn thực sự cần để có thể cài đặt git-tfs qua chocolatey.

```
C:\> choco cài đặt git-tfs
```

## Phần 31.4: git-tfs Đăng ký

Khởi chạy hộp thoại Đăng ký cho TFVS.

```
$ git tfs công cụ kiểm tra
```

Việc này sẽ lấy tất cả các cam kết cục bộ của bạn và tạo một lần đăng ký duy nhất.

## Mục 31.5: đẩy git-tfs

Đẩy tất cả các cam kết cục bộ vào điều khiển từ xa TFVS.

```
$ git tfs rcheckin
```

Lưu ý: điều này sẽ thay đổi nếu cần có Ghi chú đăng ký. Bạn có thể bỏ qua điều này bằng cách thêm git-tfs-force: rcheckin vào thông báo cam kết.

# Chương 32: Thư mục trống trong Git

## Phần 32.1: Git không theo dõi các thư mục

Giả sử bạn đã khởi tạo một dự án có cấu trúc thư mục sau:

```
/xây dựng
ứng dụng.js
```

Sau đó, bạn thêm mọi thứ bạn đã tạo cho đến nay và cam kết:

```
git init
git thêm .
git commit -m "Cam kết ban đầu"
```

Git sẽ chỉ theo dõi tệp app.js.

Giả sử bạn đã thêm một bước xây dựng vào ứng dụng của mình và dựa vào thư mục "build" ở đó làm thư mục đầu ra (và bạn không muốn biến nó thành hướng dẫn thiết lập mà mọi nhà phát triển phải tuân theo), một quy ước là bao gồm một Tệp ".gitkeep" bên trong thư mục và để Git theo dõi tệp đó.

```
/
build .gitkeep app.js
```

Sau đó thêm tệp mới này:

```
git add build/.gitkeep git
commit -m "Giữ thư mục bản dựng xung quanh"
```

Bây giờ Git sẽ theo dõi tệp build/.gitkeep và do đó thư mục bản dựng sẽ có sẵn khi thanh toán.

Một lần nữa, đây chỉ là quy ước chứ không phải tính năng Git.

# Chương 33: git-svn

## Mục 33.1: Nhân bản kho SVN

Bạn cần tạo một bản sao cục bộ mới của kho lưu trữ bằng lệnh

```
git svn sao chép SVN_REPO_ROOT_URL [DEST_FOLDER_PATH] -T TRUNK_REPO_PATH -t TAGS_REPO_PATH -b BRANCHES_REPO_PATH
```

Nếu kho lưu trữ SVN của bạn tuân theo bối cảnh tiêu chuẩn (thư mục thân cây, nhánh, thè), bạn có thể lưu một số thao tác gõ:

```
git svn bản sao -s SVN_REPO_ROOT_URL [DEST_FOLDER_PATH]
```

`git svn clone` kiểm tra từng bản sửa đổi SVN, từng bản một và thực hiện cam kết git trong kho lưu trữ cục bộ của bạn để tạo lại lịch sử. Nếu kho lưu trữ SVN có nhiều cam kết thì việc này sẽ mất một lúc.

Khi lệnh kết thúc, bạn sẽ có một kho lưu trữ git đầy đủ với một nhánh cục bộ được gọi là `master` để theo dõi nhánh trung kế trong kho SVN.

## Mục 33.2: Đẩy các thay đổi cục bộ sang SVN

Lệnh

```
git svn dcommit
```

Sẽ tạo bản sửa đổi SVN cho mỗi cam kết git cục bộ của bạn. Giống như SVN, lịch sử git cục bộ của bạn phải được đồng bộ hóa với những thay đổi mới nhất trong kho SVN, vì vậy nếu lệnh không thành công, trước tiên hãy thử thực hiện `svn rebase`.

## Mục 33.3: Làm việc tại địa phương

Chỉ cần sử dụng kho lưu trữ git cục bộ của bạn như một kho lưu trữ git bình thường, với các lệnh git thông thường:

- `git add FILE` và `git kiểm tra -- FILE` Để tạo/bỏ phân vùng một tập tin
- `git commit` Để lưu các thay đổi của bạn. Những cam kết đó sẽ cục bộ và sẽ không bị "đẩy" vào repo SVN, giống như trong kho git bình thường `git`
- `stash` và `git stash pop` Cho phép sử dụng `stash` `git`
- `reset HEAD --hard` Hoàn nguyên tất cả các thay đổi cục bộ
- của bạn `git log` Truy cập tất cả lịch sử trong kho
- lưu trữ `git rebase -i` để bạn có thể tự do viết lại lịch sử
- cục bộ của mình `git nhánh` và `git kiểm tra` để tạo các nhánh cục bộ

Vì tài liệu git-svn nêu rõ "Subversion là một hệ thống kém phức tạp hơn nhiều so với Git" nên bạn không thể sử dụng toàn bộ sức mạnh của git mà không làm xáo trộn lịch sử trong máy chủ Subversion. May mắn thay, các quy tắc rất đơn giản: Giữ lịch sử tuyến tính

Điều này có nghĩa là bạn có thể thực hiện hầu hết mọi thao tác git: tạo nhánh, xóa/sắp xếp lại/thu gọn các cam kết, di chuyển lịch sử, xóa các cam kết, v.v. Tất cả điều gì ngoại trừ hợp nhất. Thay vào đó, nếu bạn cần tích hợp lại lịch sử của các nhánh cục bộ, hãy sử dụng `git rebase`.

Khi bạn thực hiện hợp nhất, một cam kết hợp nhất sẽ được tạo. Điều đặc biệt về các cam kết hợp nhất là chúng có hai cha mẹ và điều đó làm cho lịch sử trở nên phi tuyến tính. Lịch sử phi tuyến tính sẽ gây nhầm lẫn cho SVN trong trường hợp bạn "đẩy" một cam kết hợp nhất vào kho lưu trữ.

Tuy nhiên, đừng lo lắng: bạn sẽ không phá vỡ bất cứ điều gì nếu bạn "đẩy" một cam kết hợp nhất git tới SVN. Nếu bạn làm như vậy thì khi nào

cam kết hợp nhất git được gửi đến máy chủ svn, nó sẽ chứa tất cả các thay đổi của tất cả các cam kết cho việc hợp nhất đó, do đó bạn sẽ mất lịch sử của những cam kết đó, nhưng không mất các thay đổi trong mã của bạn.

## Mục 33.4: Lấy những thay đổi mới nhất từ SVN

Tương đương với `git pull` là lệnh

```
git svn rebase
```

Thao tác này truy xuất tất cả các thay đổi từ kho lưu trữ SVN và áp dụng chúng dựa trên các cam kết cục bộ trong nhánh hiện tại của bạn.

Bạn cũng có thể sử dụng lệnh

```
tìm nạp git svn
```

để truy xuất các thay đổi từ kho lưu trữ SVN và đưa chúng vào máy cục bộ của bạn nhưng không áp dụng chúng cho chi nhánh cục bộ của bạn.

## Mục 33.5: Xử lý thư mục trống

git không nhận ra khái niệm thư mục, nó chỉ hoạt động với các tệp và đường dẫn tệp của chúng. Điều này có nghĩa là git không theo dõi các thư mục trống. Tuy nhiên, SVN thì có. Sử dụng `git-svn` có nghĩa là, theo mặc định, mọi thay đổi bạn thực hiện liên quan đến các thư mục trống bằng git sẽ không được truyền tới SVN.

Sử dụng cờ `--rmmdir` khi đưa ra nhận xét sẽ khắc phục sự cố này và xóa thư mục trống trong SVN nếu bạn xóa cục bộ tệp cuối cùng bên trong nó:

```
git svn dcommit --rmmdir
```

Thật không may, nó không loại bỏ các thư mục trống hiện có: bạn cần thực hiện thủ công.

Để tránh thêm cờ `rmmdir` khi bạn thực hiện một `dcommit` hoặc để đảm bảo an toàn nếu bạn đang sử dụng công cụ GUI git (như SourceTree), bạn có thể đặt hành vi này làm mặc định bằng lệnh:

```
cấu hình git --global svn.rmdir đúng
```

Điều này thay đổi tệp `.gitconfig` của bạn và thêm các dòng sau:

```
[svn]
rmmdir = đúng
```

Để xóa tất cả các tệp và thư mục không được theo dõi cần được giữ trống cho SVN, hãy sử dụng lệnh git:

```
git sạch -fd
```

Xin lưu ý: lệnh trước sẽ xóa tất cả các tệp không được theo dõi và các thư mục trống, ngay cả những tệp cần được SVN theo dõi! Nếu bạn cần tạo lại các thư mục trống được SVN theo dõi, hãy sử dụng lệnh

```
git svn mkdirs
```

Trong thực tế, điều này có nghĩa là nếu bạn muốn dọn sạch không gian làm việc của mình khỏi các tệp và thư mục không bị theo dõi, bạn phải luôn sử dụng cả hai lệnh để tạo lại các thư mục trống được SVN theo dõi:

```
git clean -fd && git svn mkdirs
```

# Chương 34: Lưu trữ

Tham số	Chi tiết
--format=<fmt>	Định dạng của kho lưu trữ kết quả: <code>tar</code> hoặc <code>zip</code> . Nếu tùy chọn này không được cung cấp và tệp đầu ra được chỉ định, định dạng sẽ được suy ra từ tên tệp nếu có thể. Ngược lại, mặc định là <code>tar</code> .
-l, --list	Hiển thị tất cả các định dạng có sẵn.
-v, --verbose	Báo cáo tiến trình tới <code>stderr</code> .
--prefix=<prefix>/ -o <file>, --output=<file>	Thêm <tiền tố>/ vào mỗi tên tệp trong kho lưu trữ. Viết kho lưu trữ vào <file> thay vì thiết bị xuất chuẩn.
--worktree-thuộc tính <thêm>	Tìm kiếm các thuộc tính trong tệp <code>.gitattributes</code> trong cây đang hoạt động. Đây có thể là bất kỳ tùy chọn nào mà phần phụ trợ của trình lưu trữ hiểu được. Đối với phần phụ trợ <code>zip</code> , việc sử dụng <code>-0</code> sẽ lưu trữ các tệp mà không làm xé chúng, trong khi <code>-1</code> đến <code>-9</code> có thể được sử dụng để điều chỉnh tốc độ và tỷ lệ nén.
--remote=<repo>	Truy xuất kho lưu trữ tar từ kho lưu trữ từ xa <repo> thay vì kho lưu trữ cục bộ. <code>--exec=&lt;git-upload-</code>
archive>	Được sử dụng với <code>--remote</code> để chỉ định đường dẫn đến <code>git-upload-archive</code> trên điều khiển từ xa.
<giống cây>	Cây hoặc cam kết tạo ra một kho lưu trữ cho. Nếu không có tham số tùy chọn, tất cả các tệp và thư mục trong thư mục làm việc hiện tại đều được đưa vào kho lưu trữ. Nếu một hoặc nhiều đường dẫn được chỉ định thì chỉ những đường dẫn này mới được đưa vào.

## Phần 34.1: Tạo kho lưu trữ git

Với `git archive`, có thể tạo các kho lưu trữ nén của một kho lưu trữ, ví dụ như để phân phối các bản phát hành.

Tạo một kho lưu trữ tar của bản sửa đổi HEAD hiện tại :

```
kho lưu trữ git --format tar HEAD | mèo > archive-HEAD.tar
```

Tạo một kho lưu trữ tar của bản sửa đổi HEAD hiện tại bằng nén gzip:

```
kho lưu trữ git --format tar HEAD | gzip > archive-HEAD.tar.gz
```

Điều này cũng có thể được thực hiện với (sẽ sử dụng cách xử lý `tar.gz` có sẵn):

```
kho lưu trữ git --format tar.gz HEAD > archive-HEAD.tar.gz
```

Tạo một kho lưu trữ zip của bản sửa đổi HEAD hiện tại :

```
git archive --format zip HEAD > archive-HEAD.zip
```

Ngoài ra, có thể chỉ định một tệp đầu ra có phần mở rộng hợp lệ và định dạng cũng như kiểu nén sẽ được suy ra từ nó:

```
kho lưu trữ git --output=archive-HEAD.tar.gz HEAD
```

## Phần 34.2: Tạo kho lưu trữ git với tiền tố thư mục

Việc sử dụng tiền tố khi tạo kho lưu trữ git được coi là một cách thực hành tốt, do đó việc trích xuất sẽ đặt tất cả các tệp vào trong một

danh mục. Để tạo một kho lưu trữ HEAD với tiền tố thư mục:

```
git archive --output=archive-HEAD.zip --prefix=src-directory-name HEAD
```

Khi giải nén tất cả các file sẽ được giải nén bên trong một thư mục có tên src-directory-name trong thư mục hiện tại.

### Phần 34.3: Tạo kho lưu trữ git dựa trên nhánh, bản sửa đổi, thẻ hoặc thư mục cụ thể

Cũng có thể tạo kho lưu trữ các mục khác ngoài HEAD, chẳng hạn như nhánh, cam kết, thẻ và thư mục.

Để tạo một kho lưu trữ của nhà phát triển chi nhánh địa phương:

```
git archive --output=archive-dev.zip --prefix=src-directory-name dev
```

Để tạo một kho lưu trữ của nhánh gốc/nhà phát triển từ xa:

```
git archive --output=archive-dev.zip --prefix=src-directory-name Origin/dev
```

Để tạo bản lưu trữ của thẻ v.01:

```
git archive --output=archive-v.01.zip --prefix=src-directory-name v.01
```

Tạo một kho lưu trữ các tệp bên trong một thư mục con (thư mục con) cụ thể của bản sửa đổi HEAD:

```
git archive zip --output=archive-sub-dir.zip --prefix=src-directory-name HEAD:sub-dir/
```

# Chương 35: Viết lại lịch sử với filter-branch

## Mục 35.1: Thay đổi tác giả của các cam kết

Bạn có thể sử dụng bộ lọc môi trường để thay đổi tác giả của các cam kết. Chỉ cần sửa đổi và xuất \$GIT\_AUTHOR\_NAME trong tập lệnh để thay đổi ai là tác giả của cam kết.

Tạo một file filter.sh với nội dung như sau:

```
nhếu [ "$GIT_AUTHOR_NAME" = "Tác giả để thay đổi từ" ] thì xuất
    GIT_AUTHOR_NAME="Tác giả để thay đổi thành" xuất
        GIT_AUTHOR_EMAIL="email.to.change.to@example.com"
fi
```

Sau đó chạy filter-branch từ dòng lệnh:

```
chmod +x ./filter.sh git
filter-branch --env-filter ./filter.sh
```

## Mục 35.2: Đặt git commiter bằng tác giả cam kết

Lệnh này, với phạm vi cam kết commit1..commit2, viết lại lịch sử để tác giả git commit cũng trở thành người chuyển giao git:

```
git filter-branch -f --commit-filter \'export
    GIT_COMMITTER_NAME=\"$GIT_AUTHOR_NAME\"; xuất
    GIT_COMMITTER_EMAIL=\"$GIT_AUTHOR_EMAIL\"; xuất
    GIT_COMMITTER_DATE=\"$GIT_AUTHOR_DATE\"; cây cam kết git $@\' \
-- cam kết1..commit2
```

## Chương 36: Di chuyển sang Git

### Mục 36.1: SubGit

SubGit có thể được sử dụng để thực hiện nhập một lần kho lưu trữ SVN vào git.

```
$ nhập subgit --không tương tác --svn-url http://svn.my.co/repos/myproject myproject.git
```

### Mục 36.2: Di chuyển từ SVN sang Git bằng tiện ích chuyển đổi Atlassian

Tải tiện ích chuyển đổi Atlassian [tại đây](#). Tiện ích này yêu cầu Java, vì vậy hãy đảm bảo rằng bạn có **JRE** Môi trường chạy thi hành Java được cài đặt trên máy bạn dự định thực hiện chuyển đổi.

Sử dụng lệnh `java -jar svn-migration-scripts.jar verify` để kiểm tra xem máy của bạn có thiếu bất kỳ chương trình nào cần thiết để hoàn tất quá trình chuyển đổi hay không. Cụ thể, lệnh này kiểm tra các tiện ích Git, subversion và `git-svn`. Nó cũng xác minh rằng bạn đang thực hiện di chuyển trên hệ thống tệp phân biệt chữ hoa chữ thường. Việc di chuyển sang Git phải được thực hiện trên hệ thống tệp phân biệt chữ hoa chữ thường để tránh làm hỏng kho lưu trữ.

Tiếp theo, bạn cần tạo một tệp tác giả. Subversion chỉ theo dõi những thay đổi theo tên người dùng của người gửi. Tuy nhiên, Git sử dụng hai thông tin để phân biệt người dùng: tên thật và địa chỉ email. Lệnh sau sẽ tạo một tệp văn bản ánh xạ tên người dùng Subversion với tên Git tương đương của chúng:

```
java -jar svn-migration-scripts.jar tác giả <svn-repo> tác giả.txt
```

trong đó `<svn-repo>` là URL của kho lưu trữ Subversion mà bạn muốn chuyển đổi. Sau khi chạy lệnh này, thông tin nhận dạng của người đóng góp sẽ được ánh xạ trong `Author.txt`. Địa chỉ email sẽ có dạng `<username>@mycompany.com`. Trong tệp tác giả, bạn sẽ cần thay đổi thủ công tên mặc định của từng người (theo mặc định đã trở thành tên người dùng của họ) thành tên thật của họ. Đảm bảo kiểm tra tính chính xác của tất cả các địa chỉ email trước khi tiếp tục.

Lệnh sau sẽ sao chép một repo svn dưới dạng Git:

```
git svn clone --stdlayout --authors-file=authors.txt <svn-repo> <git-repo-name>
```

trong đó `<svn-repo>` là URL kho lưu trữ tương tự được sử dụng ở trên và `<git-repo-name>` là tên thư mục trong thư mục hiện tại để sao chép kho lưu trữ vào. Có một số điều cần cân nhắc trước khi sử dụng lệnh này:

- Cờ `--stdlayout` ở trên cho Git biết rằng bạn đang sử dụng bộ cục tiêu chuẩn với các thư mục thân, nhánh và thẻ. Kho lưu trữ Subversion có bộ cục không chuẩn yêu cầu bạn chỉ định vị trí của thư mục trung kẽ, bất kỳ/tất cả các thư mục nhánh và thư mục thẻ. Điều này có thể được thực hiện bằng cách làm theo ví dụ sau: `git svn clone --trunk=/trunk --branches=/branches --branches=/bugfixes --tags=/tags --authors= file=authors.txt <svn-repo> <git-repo-name>`.
- Lệnh này có thể mất nhiều giờ để hoàn thành tùy thuộc vào kích thước kho lưu trữ của bạn.
- Để giảm thời gian chuyển đổi cho các kho lưu trữ lớn, quá trình chuyển đổi có thể được chạy trực tiếp trên máy chủ lưu trữ kho lưu trữ lật đổ nhằm loại bỏ chi phí mạng.

`git svn clone` nhập các nhánh Subversion (và thân cây) dưới dạng các nhánh từ xa bao gồm các thẻ Subversion (các nhánh từ xa có tiền tố là `tags/`). Để chuyển đổi chúng thành các nhánh và thẻ thực tế, hãy chạy các lệnh sau trên máy Linux theo thứ tự chúng được cung cấp. Sau khi chạy chúng, `git branch -a` sẽ hiển thị tên nhánh chính xác và `git tag -l` sẽ hiển thị các thẻ kho lưu trữ.

```
git for-each-ref refs/điều khiển từ xa/Xuất xứ/thẻ | cắt -d / -f 5- | grep -v @ | trong khi đọc tên thẻ; làm git tag $tagname
Origin/tags/$tagname; nhánh git -r -d Origin/tags/$tagname; đã hoàn thành git cho mỗi giới thiệu /
điều khiển từ xa | cắt -d / -f 4- | grep -v @ | trong khi đọc tên chi nhánh; làm git nhánh "$branchname" "refs/remotes/
origin/$branchname"; git nhánh -r -d "origin/$branchname"; xong
```

Quá trình chuyển đổi từ svn sang Git hiện đã hoàn tất! Chỉ cần đẩy kho lưu trữ cục bộ của bạn lên máy chủ và bạn có thể tiếp tục đóng góp bằng Git cũng như có lịch sử phiên bản được bảo toàn hoàn toàn từ svn.

### Phần 36.3: Di chuyển Mercurial sang Git

Người ta có thể sử dụng các phương pháp sau để nhập Mercurial Repo vào Git:

1. Sử dụng [tính năng xuất nhanh](#):

```
cd git clone git://repo.or.cz/fast-export.git git
init git_repo cd
git_repo ~/
fast-export/hg-fast-export.sh -r /path/to/old/mercurial_repo git thanh toán HEAD
```

2. Sử dụng [Hg-Git](#): Câu trả lời rất chi tiết tại đây: <https://stackoverflow.com/a/31827990/5283213>

3. Sử dụng [Trình nhập của GitHub](#): Làm theo hướng dẫn (chi tiết) tại [GitHub](#).

### Mục 36.4: Di chuyển từ Kiểm soát phiên bản Team Foundation (TFVC) sang Git

Bạn có thể di chuyển từ kiểm soát phiên bản nền tảng nhóm sang git bằng cách sử dụng công cụ nguồn mở có tên Git-TF. Quá trình di chuyển cũng sẽ chuyển lịch sử hiện tại của bạn bằng cách chuyển đổi đăng ký tfs thành cam kết git.

Để đưa giải pháp của bạn vào Git bằng cách sử dụng Git-TF, hãy làm theo các bước sau:

Tải xuống Git-TF

Bạn có thể tải xuống (và cài đặt) Git-TF từ Codeplex: [Git-TF @ Codeplex](#)

Sao chép giải pháp TFVC của bạn

Khởi chạy powershell (win) và gõ lệnh

```
bản sao git-tf http://my.tfs.server.address:port/tfs/mycollection '$/myproject/mybranch/mysolution' --deep
```

Công tắc --deep là từ khóa cần lưu ý vì điều này yêu cầu Git-Tf sao chép lịch sử đăng ký của bạn. Jetzt giờ bạn có một kho lưu trữ git cục bộ trong thư mục mà bạn đã gọi lệnh clone từ đó.

Dọn dẹp

- Thêm tệp .gitignore. Nếu bạn đang sử dụng Visual Studio, trình chỉnh sửa có thể thực hiện việc này cho bạn, nếu không, bạn có thể thực hiện việc này theo cách thủ công bằng cách tải xuống tệp hoàn chỉnh [github/gitignore](#).
- Xóa các liên kết kiểm soát nguồn TFS khỏi giải pháp (xóa tất cả các tệp \*.vsscc). Bạn cũng có thể sửa đổi tệp giải pháp của mình bằng cách xóa GlobalSection(TeamFoundationVersionControl).....EndGlobalSection

Cam kết & Đẩy

Hoàn tất quá trình chuyển đổi của bạn bằng cách cam kết và đẩy kho lưu trữ cục bộ của bạn vào điều khiển từ xa.

```
git thêm .
git commit -a -m "Kiểm soát nguồn giải pháp được chuyển đổi từ TFVC sang Git"

git từ xa thêm nguồn gốc https://my.remote/project/repo.git

git đẩy nguồn gốc chính
```

## Mục 36.5: Di chuyển từ SVN sang Git bằng svn2git

[svn2git](#) là trình bao bọc Ruby xung quanh hỗ trợ SVN gốc của git thông qua [git-svn](#), giúp bạn di chuyển các dự án từ Subversion sang Git, lưu giữ lịch sử (bao gồm lịch sử trung kẽ, thẻ và nhánh).

Ví dụ

Để di chuyển kho lưu trữ svn với bộ cục tiêu chuẩn (ví dụ: các nhánh, thẻ và trung kẽ ở cấp gốc của kho lưu trữ):

```
$ svn2git http://svn.example.com/path/to/repo
```

Để di chuyển kho lưu trữ svn không có bộ cục chuẩn:

```
$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --tags tags-dir --branches Branch-dir
```

Trong trường hợp bạn không muốn di chuyển (hoặc không có) các nhánh, thẻ hoặc đường trực, bạn có thể sử dụng các tùy chọn `--notrunk`, `--nobranches` và `--notags`.

Ví dụ: `$ svn2git http://svn.example.com/path/to/repo --trunk trunk-dir --notags --nobranches` sẽ chỉ di chuyển lịch sử trung kẽ.

Để giảm dung lượng mà kho lưu trữ mới yêu cầu, bạn có thể muốn loại trừ mọi thư mục hoặc tệp mà bạn đã từng thêm vào trong khi lẽ ra bạn không nên có (ví dụ: thư mục xây dựng hoặc kho lưu trữ):

```
$ svn2git http://svn.example.com/path/to/repo --exclude build --exclude '.*\.zip$'
```

Tối ưu hóa sau di chuyển

Nếu bạn đã có vài nghìn lần xác nhận (hoặc nhiều hơn) trong kho git mới tạo của mình, bạn có thể muốn giảm dung lượng sử dụng trước khi đẩy kho lưu trữ của mình lên điều khiển từ xa. Điều này có thể được thực hiện bằng lệnh sau:

```
$ git gc --tích cực
```

Lưu ý: Lệnh trước đó có thể mất tới vài giờ trên các kho lưu trữ lớn (hàng chục nghìn lần xác nhận và/hoặc hàng trăm megabyte lịch sử).

## Chương 37: Trình diễn

### Phần 37.1: Tổng quan

`git show` hiển thị các đối tượng Git khác nhau.

Đối với các cam kết:

Hiển thị thông báo cam kết và sự khác biệt của những thay đổi được giới thiệu.

Lệnh <code>git</code> hiển	Sự miêu tả
<code>thị</code>	hiển thị cam kết trước đó

`git show @~3` hiển thị lần xác nhận thứ 3 tính từ lần cuối

Đối với cây và đóm màu:

Hiển thị cây hoặc đóm màu.

Yêu cầu	Sự miêu tả
<code>chương trình git @~3:</code>	hiển thị thư mục gốc của dự án như 3 lần xác nhận trước (một cây)
<code>git show @~3:src/program.js</code>	hiển thị src/program.js như 3 lần xác nhận trước (một đóm màu)
<code>git show @:a.txt @:b.txt</code>	hiển thị a.txt được nối với b.txt từ cam kết hiện tại

Đối với thẻ:

Hiển thị thông báo thẻ và đối tượng được tham chiếu.

# Chương 38: Giải quyết xung đột hợp nhất

## Mục 38.1: Giải quyết thủ công

Trong khi thực hiện **hợp nhất git**, bạn có thể thấy rằng git báo lỗi "xung đột hợp nhất". Nó sẽ báo cáo cho bạn những tập tin có xung đột và bạn sẽ cần giải quyết xung đột.

Trạng thái git tại bất kỳ thời điểm nào sẽ giúp bạn biết những gì vẫn cần chỉnh sửa bằng thông báo hữu ích như

Trên nhánh chính

Bạn có đường dẫn chưa hợp nhất.

(sửa xung đột và chạy "**git commit**")

Đường dẫn chưa hợp nhất:

(sử dụng "**git add <file>...**" để đánh dấu độ phân giải)

cả hai đều được sửa đổi: **chỉ mục.html**

Không có thay đổi nào được thêm vào cam kết (sử dụng "**git add**" và/hoặc "**git commit -a**")

Git để lại các điểm đánh dấu trong các tệp để cho bạn biết xung đột phát sinh ở đâu:

```
<<<<<< HEAD: index.html #cho biết trạng thái chi nhánh hiện tại của bạn <div
id="footer">liên hệ : email@somedomain.com</div> ===== #indicates
sự gián đoạn giữa các xung đột <div id="footer"> vui lòng
liên hệ với chúng tôi
tại email@somedomain.com </div> >>>>>> iss2:
index.html #indicates trạng thái của nhánh khác (iss2 )
```

Để giải quyết xung đột, bạn phải chỉnh sửa vùng giữa các dấu <<<< và >>>>> cho phù hợp, xóa các dòng trạng thái (các <<<<, >>>> >> và ===== dòng) hoàn toàn. Sau đó **git add** index.html để đánh dấu nó đã được giải quyết và **git commit** để hoàn tất việc hợp nhất.

# Chương 39: Bó

## Phần 39.1: Tạo gói git trên máy cục bộ và sử dụng nó trên máy khác

Đôi khi bạn có thể muốn duy trì các phiên bản của kho git trên các máy không có kết nối mạng.

Các gói cho phép bạn đóng gói các đối tượng và tài liệu tham khảo git trong kho lưu trữ trên một máy và nhập chúng vào kho lưu trữ trên máy khác.

```
thẻ git 2016_07_24 gói
git tạo các thay đổi_between_tags.bundle [some_previous_tag]..2016_07_24
```

Bằng cách nào đó chuyển tệp Changes\_between\_tags.bundle sang máy từ xa; ví dụ, thông qua ổ ngón tay cái. Một khi bạn có nó ở đó:

```
git bó xác minh thay đổi_between_tags.bundle # đảm bảo gói đến nguyên vẹn git kiểm tra [một số nhánh] # trong kho lưu trữ trên máy từ xa git Bundle list-heads
Changes_between_tags.bundle # liệt kê các tham chiếu trong gói git pull Changes_between_tags.bundle [tham chiếu từ gói, ví dụ: trường cuối cùng từ đầu ra trước đó]
```

Điều ngược lại cũng có thể xảy ra. Khi bạn đã thực hiện các thay đổi trên kho lưu trữ từ xa, bạn có thể gói các vùng đồng bằng; đặt các thay đổi trên, ví dụ: ổ USB và hộp nhất chúng trở lại kho lưu trữ cục bộ để cả hai có thể đồng bộ hóa mà không yêu cầu truy cập giao thức git, ssh, rsync hoặc http trực tiếp giữa các máy.

## Chương 40: Hiển thị lịch sử cam kết bằng đồ họa với Gitk

### Mục 40.1: Hiển thị lịch sử cam kết cho một tệp

đường dẫn gitk /địa chỉ/myfile

### Mục 40.2: Hiển thị tất cả các lần xác nhận giữa hai lần xác nhận

Giả sử bạn có hai lần xác nhận d9e1db9 và 5651067 và muốn xem điều gì đã xảy ra giữa chúng. d9e1db9 là tổ tiên lâu đời nhất và 5651067 là hậu duệ cuối cùng trong chuỗi các cam kết.

gitk --ancestry-path d9e1db9 5651067

### Mục 40.3: Hiển thị các cam kết kể từ thẻ phiên bản

Nếu bạn có thẻ phiên bản v2.3, bạn có thể hiển thị tất cả các cam kết kể từ thẻ đó.

gitk v2.3..

## Chương 41: Chia đôi/Tìm kiếm các cam kết bị lỗi

### Mục 41.1: Tìm kiếm nhị phân (git bisect)

git chia đôi cho phép bạn tìm ra cam kết nào đã gây ra lỗi bằng cách sử dụng tìm kiếm nhị phân.

Bắt đầu bằng cách chia đôi một phiên bằng cách cung cấp hai tham chiếu cam kết: một cam kết tốt trước lỗi và một cam kết xấu sau lỗi. Nói chung, cam kết xấu là HEAD.

```
# bắt đầu phiên git bisect $ git
bisect start

# đưa ra một cam kết khi lỗi không tồn tại $ git
bisect good 49c747d

# đưa ra một cam kết khi có lỗi $ git
bisect bad HEAD
```

git bắt đầu tìm kiếm nhị phân: Nó chia bản sửa đổi làm đôi và chuyển kho lưu trữ sang bản sửa đổi trung gian.

Kiểm tra mã để xác định xem bản sửa đổi là tốt hay xấu:

```
# nói với git rằng bản sửa đổi là tốt,
# có nghĩa là nó không chứa lỗi $ git bisect good

# nếu bản sửa đổi có lỗi, # thì hãy nói
với git rằng nó tệ $ git
bisect bad
```

git sẽ tiếp tục chạy tìm kiếm nhị phân trên từng tập hợp con còn lại của các bản sửa đổi xấu tùy theo hướng dẫn của bạn. git sẽ trình bày một bản sửa đổi duy nhất, trừ khi cờ của bạn không chính xác, sẽ thể hiện chính xác bản sửa đổi có lỗi được đưa ra.

Sau đó hãy nhớ chạy `git bisect reset` để kết thúc phiên chia đôi và quay lại HEAD.

```
$ git thiết lập lại chia đôi
```

Nếu bạn có tập lệnh có thẻ kiểm tra lỗi, bạn có thể tự động hóa quy trình bằng:

```
$ git bisect run [script] [đối số]
```

Trong đó [tập lệnh] là đường dẫn đến tập lệnh của bạn và [đối số] là bất kỳ đối số nào sẽ được chuyển đến tập lệnh của bạn.

Chạy lệnh này sẽ tự động chạy qua tìm kiếm nhị phân, thực thi `git bisect good` hoặc `git bisect bad` ở mỗi bước tùy thuộc vào mã thoát của tập lệnh của bạn. Thoát bằng 0 biểu thị tốt, trong khi thoát bằng 1-124, 126 hoặc 127 biểu thị xấu. 125 chỉ ra rằng tập lệnh không thể kiểm tra bản sửa đổi đó (điều này sẽ kích hoạt bỏ qua `git bisect`).

### Mục 41.2: Bán tự động tìm một cam kết bị lỗi

Hãy tưởng tượng bạn đang ở nhánh chính và có điều gì đó không hoạt động như mong đợi (một hồi quy đã được đưa ra), nhưng bạn không biết ở đâu. Tất cả những gì bạn biết là, nó đang hoạt động trong bản phát hành gần đây nhất (ví dụ: được gắn thẻ hoặc bạn biết hàm băm cam kết, hãy lấy old-rel ở đây).

Git đã giúp bạn tìm ra cam kết bị lỗi dẫn đến lỗi quy với số bước rất thấp (tìm kiếm nhị phân).

Trước hết hãy bắt đầu chia đôi:

```
git bisect start master old-rel
```

Điều này sẽ cho git biết rằng master là phiên bản bị hỏng (hoặc phiên bản bị hỏng đầu tiên) và old-rel là phiên bản được biết đến cuối cùng.

Bây giờ Git sẽ kiểm tra một phần đầu tách rời ở giữa cả hai lần xác nhận. Bây giờ, bạn có thể thực hiện thử nghiệm của mình. Tùy thuộc vào việc nó có hoạt động hay không

```
git chia đôi tốt
```

hoặc

```
git chia đôi xấu
```

· Trong trường hợp cam kết này không thể được kiểm tra, bạn có thể dễ dàng `git reset` và kiểm tra cam kết đó, git sẽ xử lý việc này.

Sau một vài bước, git sẽ xuất ra hàm bấm xác nhận bị lỗi.

Để hủy bỏ quá trình chia đôi chỉ cần phát hành

```
thiết lập lại git chia đôi
```

và git sẽ khôi phục trạng thái trước đó.

# Chương 42: Đỗ lỗi

Tên tệp tham	Chi tiết
số	Tên file cần kiểm tra chi tiết
-f	Hiển thị tên tệp trong cam kết gốc
-e	Hiển thị email tác giả thay vì tên tác giả
-w	Bỏ qua khoảng trắng trong khi so sánh giữa phiên bản con và phiên bản cha mẹ
-L start,end	Chỉ hiển thị phạm vi dòng đã cho Ví dụ: <code>git đỗ lỗi -L 1,2 [tên tệp]</code>
--show-stats	Hiển thị số liệu thống kê bổ sung ở cuối đầu ra đỗ lỗi
-l	Hiển thị vòng quay dài (Mặc định: tắt)
-t	Hiển thị dấu thời gian thô (Mặc định: tắt)
-đảo ngược	Đi về phía trước thay vì lùi lại lịch sử
-p, --porcelain	Đầu ra cho máy tiêu thụ
-M	Phát hiện các dòng được di chuyển hoặc sao chép trong một tệp
-C	Ngoài -M, phát hiện các dòng được di chuyển hoặc sao chép từ các tệp khác đã được sửa đổi trong cùng một làm
-h	Hiển thị thông báo trợ giúp
-c	Sử dụng chế độ đầu ra tương tự như git-annotate (Mặc định: tắt)
-N	Hiển thị số dòng trong cam kết ban đầu (Mặc định: tắt)

## Mục 42.1: Chỉ hiển thị một số dòng nhất định

Đầu ra có thể bị hạn chế bằng cách chỉ định phạm vi dòng như

`git đỗ lỗi -L <bắt đầu>,<kết thúc>`

Nơi <start> và <end> có thể là:

- số dòng

`git đỗ lỗi -L 10,30`

- /regex/

`git đỗ lỗi -L /void main/, git đỗ lỗi -L 46,/void foo/`

- +offset, -offset (chỉ dành cho <end>)

`git đỗ lỗi -L 108,+30, git đỗ lỗi -L 215,-15`

Có thể chỉ định nhiều phạm vi dòng và cho phép phạm vi chồng chéo.

`git đỗ lỗi -L 10,30 -L 12,80 -L 120,+10 -L ^/void main/,+40`

## Phần 42.2: Để tìm ra ai đã thay đổi tập tin

```
// Hiển thị tác giả và cam kết trên mỗi dòng của tệp được chỉ định
kiểm tra đỗ lỗi git.c
```

```
// Hiển thị email tác giả và cam kết trên mỗi dòng của tệp git Blaze -e test.c  
được chỉ định
```

```
// Giới hạn việc lựa chọn các dòng theo phạm vi được chỉ định git lỗi  
lỗi -L 1,10 test.c
```

### Phần 42.3: Hiển thị cam kết sửa đổi một dòng lần cuối

```
git lỗi <tập tin>
```

sẽ hiển thị tệp với mỗi dòng được chú thích bằng cam kết sửa đổi lần cuối.

### Mục 42.4: Bỏ qua những thay đổi chỉ có khoảng trắng

Đôi khi các kho lưu trữ sẽ có các cam kết chỉ điều chỉnh khoảng trắng, ví dụ như sửa lỗi thụt lề hoặc chuyển đổi giữa các tab và dấu cách. Điều này gây khó khăn cho việc tìm ra cam kết nơi mã thực sự được viết.

```
git lỗi -w
```

sẽ bỏ qua những thay đổi chỉ có khoảng trắng để tìm ra dòng thực sự đến từ đâu.

# Chương 43: Cú pháp sửa đổi Git

## Mục 43.1: Chỉ định sửa đổi theo tên đối tượng

```
$ git hiển thị dae86e1950b1277e545cee180551750029cfe735
$ git hiển thị dae86e19
```

Bạn có thể chỉ định sửa đổi (hoặc thực tế là bất kỳ đối tượng nào: thẻ, cây tức là nội dung thư mục, blob tức là nội dung tệp) bằng SHA-1 tên đối tượng, chuỗi thập lục phân 40 byte đầy đủ hoặc chuỗi con duy nhất cho kho lưu trữ.

## Mục 43.2: Tên tham chiếu tương trưng: nhánh, thẻ, nhánh theo dõi từ xa

```
$ git log master # chỉ định nhánh
$ git show v1.0 # chỉ định thẻ
$ git show HEAD # chỉ định nhánh hiện tại
$ git show Origin # chỉ định nhánh theo dõi từ xa mặc định cho 'nguồn gốc' từ xa
```

Bạn có thể chỉ định sửa đổi bằng cách sử dụng tên tham chiếu mang tính biểu tượng, bao gồm các nhánh (ví dụ: 'master', 'next', 'maint'), thẻ (ví dụ: 'v1.0', 'v0.6.3-rc2'), các nhánh theo dõi từ xa (ví dụ: 'origin', 'origin/master') và đặc biệt ref chẳng hạn như 'HEAD' cho nhánh hiện tại.

Nếu tên tham chiếu mang tính biểu tượng không rõ ràng, ví dụ: nếu bạn có cả nhánh và thẻ có tên 'sửa' (có nhánh và không nên sử dụng thẻ có cùng tên), bạn cần chỉ định loại ref bạn muốn sử dụng:

```
$ git hiển thị đầu/sửa $ # hoặc 'refs/heads/fix', để chỉ định nhánh
git hiển thị thẻ/sửa # hoặc 'refs/tags/fix', để chỉ định thẻ
```

## Mục 43.3: Bản sửa đổi mặc định: HEAD

```
chương trình $ git # tương đương với 'git show HEAD'
```

'HEAD' đặt tên cho cam kết mà bạn dựa vào đó để thực hiện các thay đổi trong cây làm việc và thường là tên tương trưng cho chi nhánh hiện tại. Nhiều lệnh (nhưng không phải tất cả) lấy tham số sửa đổi mặc định là 'HEAD' nếu nó bị thiếu.

## Mục 43.4: Tham chiếu lại nhật ký: <refname>{@{<n>}}

```
$ git show @{1} $ # sử dụng reflog cho nhánh hiện tại
git show master@{1} $ git # sử dụng reflog cho nhánh 'master'
show HEAD@{1} # sử dụng reflog 'HEAD'
```

Một tham chiếu, thường là nhánh hoặc HEAD, theo sau là hậu tố @ với thông số thứ tự được đặt trong cặp dấu ngoặc nhọn (ví dụ: {1}, {15}) chỉ định giá trị trước thứ n của tham chiếu đó trong kho lưu trữ cục bộ của bạn. Bạn có thể kiểm tra các mục reflog gần đây với [git reflog](#) lệnh hoặc tùy chọn --walk-reflogs / -g cho [git log](#).

```
$ git đăng lại
08bb350 HEAD@{0}: đặt lại: di chuyển tới HEAD^
4ebf58d HEAD@{1}: commit: gitweb(1): Tham số truy vấn tài liệu
08bb350 HEAD@{2}: kéo: Chuyển tiếp nhanh
f34be46 HEAD@{3}: kiểm tra: chuyển từ af40944bda352190f05d22b7cb8fe88beb17f3a7 sang chủ
af40944 HEAD@{4}: kiểm tra: chuyển từ chính sang v2.6.3

$ git reflog gitweb-docs
```

```
4ebf58d gitweb-docs@{0}: nhánh: Được tạo từ chủ
```

Lưu ý: việc sử dụng `reflog` trên thực tế đã thay thế cơ chế sử dụng `ref ORIG_HEAD` cũ hơn (gần tương đương với `HEAD@{1}`).

## Mục 43.5: Tham chiếu lại nhật ký: <refname>@{<date>}

```
$ git show master@{ngày hôm qua} $ git
show HEAD@{5 phút trước} # hoặc HEAD@{5. Minutes.ago}
```

Một tham chiếu theo sau là hậu tố @ với thông số ngày được đặt trong cặp dấu ngoặc nhọn (ví dụ: {ngày hôm qua}, {1 tháng 2 tuần 3 ngày 1 giờ 1 giây trước} hoặc {1979-02-26 18:30:00}) chỉ định giá trị của ref tại thời điểm trước đó (hoặc điểm gần nhất với nó). Lưu ý rằng thao tác này sẽ tra cứu trạng thái của giới thiệu địa phương của bạn tại một thời điểm nhất định; ví dụ: chỉ nhánh 'chính' tại địa phương của bạn có gì vào tuần trước.

Bạn có thể sử dụng `git reflog` với công cụ xác định ngày để tra cứu thời gian chính xác nơi bạn đã làm điều gì đó để nhận được giới thiệu trong kho lưu trữ cục bộ.

```
$ git reflog HEAD@{now} 08bb350
HEAD@{Thứ bảy ngày 23 tháng 7 19:48:13 2016 +0200}: đặt lại: chuyển sang HEAD^ 4ebf58d HEAD@{Thứ
bảy ngày 23 tháng 7 19:39:20 2016 +0200}: cam kết: gitweb(1): Tham số truy vấn tài liệu 08bb350 HEAD@{Thứ bảy ngày 23 tháng 7
19:26:43 2016 +0200}: kéo: Chuyển tiếp nhanh
```

## Mục 43.6: Nhánh được theo dõi/ngược dòng: <branchname>@{upstream}

```
$ git log @{upstream}.. $ git           # những gì đã được thực hiện tại địa phương và chưa được công bố, chỉ nhánh hiện tại
show master@{upstream} # hiển thị ngược dòng của nhánh 'master'
```

Hậu tố @{upstream} được thêm vào tên nhánh (dạng ngắn <branchname>@{u}) để cập đến nhánh mà nhánh được chỉ định bởi tên nhánh được đặt để xây dựng trên (được định cấu hình với nhánh.<name>.remote và nhánh .<name>.merge hoặc với nhánh `git --set-upstream-to=<branch>`). Một tên nhánh bị thiếu mặc định là hiện tại.

Cùng với cú pháp cho phạm vi sửa đổi, sẽ rất hữu ích khi xem các cam kết mà nhánh của bạn đi trước thượng nguồn (các cam kết trong kho lưu trữ cục bộ của bạn chưa xuất hiện ở thượng nguồn) và những cam kết nào bạn đứng sau (các cam kết ngược dòng không được hợp nhất vào nhánh cục bộ), hoặc cả hai:

```
$ git log --oneline @{u}.. $ git log
--oneline ..@{u} $ git log --oneline
--left-right @{u}... # giống như ...@{ bạn}
```

## Mục 43.7: Chuỗi cam kết tổ tiên: <rev>^, <rev>~<n>, v.v.

```
$ git reset --hard HEAD^ $ git          # hủy bỏ lần xác nhận cuối cùng
rebase --HEAD tương tác ~5            # rebase 4 lần xác nhận cuối cùng
```

Hậu tố ^ cho tham số sửa đổi có nghĩa là cha mẹ đầu tiên của đối tượng cam kết đó. ^<n> có nghĩa là cha mẹ <n>-th (tức là <rev>^ tương đương với <rev>^1).

Hậu tố ~<n> cho tham số sửa đổi có nghĩa là đối tượng cam kết là tổ tiên thứ <n> của đối tượng cam kết được đặt tên, chỉ theo sau cha mẹ đầu tiên. Điều này có nghĩa là ví dụ <rev>~3 tương đương với <rev>^^. Là một phím tắt, <rev>~ có nghĩa là <rev>~1 và tương đương với <rev>^1 hoặc viết tắt là <rev>^ .

Cú pháp này có thể kết hợp được.

Để tìm những tên tương trưng như vậy, bạn có thể sử dụng `git name-rev` yêu cầu:

```
$ git name-rev 33db5f4d9027a10e477ccf054b2c1ab94f74c85a  
33db5f4d9027a10e477ccf054b2c1ab94f74c85a thê/v0.99~940
```

Lưu ý rằng `--pretty=oneline` chứ không phải `--oneline` phải được sử dụng trong ví dụ sau

```
$ git log --pretty=oneline | git name-rev --stdin --name-only master Lô chủ đề thứ  
sáu cho 2.10 master~1 Hợp nhất nhánh 'ls/p4-tmp-  
refs' master~2 Hợp nhất nhánh 'js/am-call-theirs-  
theirs-in -fallback-3way' [...] master~14^2 sideband.c: tối ưu hóa nhỏ việc sử dụng strbuf  
  
master~16^2 connect: đọc $GIT_SSH_COMMAND từ tệp cấu hình [...] master~22^2~1  
t7810 -grep.sh: sửa lỗi không nhất quán khoảng trắng master~22^2~2 t7810-grep.sh:  
sửa  
tên kiểm tra trùng lặp
```

## Mục 43.8: Ngắt tham chiếu các nhánh và thẻ: <rev>^0, <rev>^{<type>}

Trong một số trường hợp, hành vi của một lệnh phụ thuộc vào việc nó được đặt tên nhánh, tên thẻ hay một bản sửa đổi tùy ý. Bạn có thể sử dụng cú pháp "khỏi tham chiếu" nếu cần cú pháp sau.

Hậu tố ^ theo sau là tên loại đối tượng (thẻ, cam kết, cây, blob) được đặt trong cặp dấu ngoặc nhọn (ví dụ `v0.99.8^{commit}`) có nghĩa là hủy đăng ký đối tượng tại `<rev>` theo cách đệ quy cho đến khi một đối tượng thuộc loại `<type>` được tìm thấy hoặc đối tượng không thẻ được hủy đăng ký nữa. `<rev>^0` là viết tắt của `<rev>^{commit}`.

```
$ git thanh toán ĐẦU^0 # tương đương với 'gitcheck --detach' trong Git hiện đại
```

Hậu tố ^ theo sau là cặp dấu ngoặc nhọn trống (ví dụ `v0.99.8^{}`) có nghĩa là hủy đăng ký thẻ theo cách đệ quy cho đến khi tìm thấy đối tượng không phải thẻ.

So sánh

```
$ git show v1.0 $  
git cat-file -p v1.0 $ git  
thay thế --edit v1.0
```

với

```
$ git show v1.0^{} $ git  
cat-file -p v1.0^{} $ git thay thế  
--edit v1.0^{}
```

## Mục 43.9: Cam kết phù hợp trẻ nhất: <rev>^{</text>}, :<text>

```
$ git show HEAD^{/fix lỗi khó chịu} # tìm bắt đầu từ HEAD $ git show ':/fix lỗi  
khó chịu' # tìm bắt đầu từ bất kỳ chi nhánh nào
```

Dấu hai chấm ('::'), theo sau là dấu gạch chéo ('/'), theo sau là văn bản, đặt tên cho một cam kết có thông báo cam kết khớp với biểu thức chính quy được chỉ định. Tên này trả về cam kết phù hợp trẻ nhất có thể truy cập được từ bất kỳ giới thiệu nào.

Biểu thức chính quy có thể khớp với bất kỳ phần nào của thông báo cam kết. Để khớp các tin nhắn bắt đầu bằng một chuỗi, người ta có thể sử dụng ví dụ :/^foo. Trình tự đặc biệt :/! được dành riêng cho các sửa đổi cho những gì phù hợp. :/!-foo thực hiện khớp phủ định, trong khi :/!foo khớp với nghĩa đen ! ký tự, theo sau là foo.

Hậu tố ^ cho tham số sửa đổi, theo sau là cặp dấu ngoặc nhọn chứa văn bản dẫn đầu bởi dấu gạch chéo, giống như cú pháp :/  
<text> bên dưới, nó trả về cam kết khớp trễ nhất có thể truy cập được từ <rev> trước đó ^.

## Chương 44: Cây làm việc

Tham số	Chi tiết
-f --lực lượng	Theo mặc định, add từ chối tạo cây làm việc mới khi <branch> đã được cây làm việc khác kiểm tra. Tùy chọn này sẽ ghi đè biện pháp bảo vệ đó.
-b <nhánh mới> -B <nhánh mới>	Với phần thêm, hãy tạo một nhánh mới có tên <new-branch> bắt đầu tại <branch> và kiểm tra <new-branch> trong cây làm việc mới. Nếu <branch> bị bỏ qua, nó sẽ mặc định là HEAD. Theo mặc định, -b từ chối tạo nhánh mới nếu nó đã tồn tại. -B ghi đè biện pháp bảo vệ này, đặt lại <nhánh mới> thành <nhánh>.
--tách ra	Với thao tác thêm, tách HEAD trong cây làm việc mới.
--[không-] thanh toán	Theo mặc định, thêm kiểm tra <nhánh>, tuy nhiên, --no-checkout có thể được sử dụng để chặn kiểm tra nhằm thực hiện các tùy chỉnh, chẳng hạn như định cấu hình kiểm tra thưa thớt.
-n --chạy khô	Với việc cắt tỉa, dừng loại bỏ bất cứ thứ gì; chỉ cần báo cáo những gì nó sẽ loại bỏ.
--sử	Với danh sách, xuất ra ở định dạng dễ phân tích cú pháp cho tập lệnh. Định dạng này sẽ ổn định trên các phiên bản Git và bất kể cấu hình người dùng.
-v --dài dòng	Với Prune, hãy báo cáo tất cả các lần loại bỏ.
-hết hạn <thời gian>	Với việc cắt tỉa, chỉ hết hạn những cây đang làm việc chưa sử dụng cũ hơn <time>.

### Mục 44.1: Sử dụng cây công việc

Bạn đang trong quá trình phát triển một tính năng mới thì sép của bạn đến yêu cầu bạn phải sửa chữa điều gì đó ngay lập tức. Thông thường, bạn có thể muốn sử dụng `git stash` để tạm thời lưu trữ các thay đổi của mình. Tuy nhiên, tại thời điểm này, cây làm việc của bạn đang ở trạng thái lộn xộn (với các tệp mới, được di chuyển và bị xóa cũng như các bit và phần khác nằm rải rác xung quanh) và bạn không muốn làm xáo trộn tiến trình của mình.

Bằng cách thêm một cây làm việc, bạn tạo một cây làm việc được liên kết tạm thời để thực hiện khắc phục khẩn cấp, xóa nó khi hoàn tất và sau đó tiếp tục phiên mã hóa trước đó của mình:

```
$ git worktree add -b khẩn cấp-sửa chữa ../temp master $  
pushd ../temp # ...  
công việc công việc công việc ...  
$ git commit -a -m 'sửa chữa khẩn cấp cho sép' $ popd  
$ rm  
-rf ../temp $ git  
Worktree Prune
```

LƯU Ý: Trong ví dụ này, bản sửa lỗi vẫn nằm trong nhánh bản sửa lỗi khẩn cấp. Tại thời điểm này, bạn có thể muốn `git merge` hoặc `git format-patch` và sau đó xóa nhánh sửa lỗi khẩn cấp.

### Mục 44.2: Di chuyển cây làm việc

Hiện tại (kể từ phiên bản 2.11.0) không có chức năng tích hợp nào để di chuyển cây công việc đã có sẵn. Đây được liệt kê là lỗi chính thức (xem [https://git-scm.com/docs/git-worktree#\\_bugs](https://git-scm.com/docs/git-worktree#_bugs)).

Để khắc phục hạn chế này, có thể thực hiện các thao tác thủ công trực tiếp trong tệp tham chiếu .git.

Trong ví dụ này, bản sao chính của repo nằm ở /home/user/project-main và cây làm việc phụ được đặt tại /home/user/project-1 và chúng tôi muốn chuyển nó đến /home/user/project-2.

Không thực hiện bất kỳ lệnh git nào giữa các bước này, nếu không trình thu gom rác có thể được kích hoạt và các tham chiếu đến cây thứ cấp có thể bị mất. Thực hiện các bước này từ đầu đến cuối mà không bị gián đoạn:

- Thay đổi tệp .git của cây làm việc để trả đến vị trí mới bên trong cây chính. Tập tin /home/user/project-1/.git bây giờ sẽ chứa những nội dung sau:

```
gitdir: /home/user/project-main/.git/worktrees/project-2
```

- Đổi tên cây làm việc bên trong thư mục .git của dự án chính bằng cách di chuyển thư mục của cây làm việc đó tồn tại trong đó:

```
$ mv /home/user/project-main/.git/worktrees/project-1 /home/user/project-main/.git/worktrees/project-2
```

- Thay đổi tham chiếu bên trong /home/user/project-main/.git/worktrees/project-2/gitdir để trả đến địa điểm mới. Trong ví dụ này, tệp sẽ có nội dung sau:

```
/home/user/project-2/.git
```

- Cuối cùng, di chuyển cây làm việc của bạn đến vị trí mới:

```
$ mv /home/user/project-1 /home/user/project-2
```

Nếu bạn đã thực hiện mọi thứ một cách chính xác, việc liệt kê các cây công việc hiện có sẽ đề cập đến vị trí mới:

```
$ git Worktree list /  
home/user/project-main 23f78ad [master] /home/user/  
project-2 78ac3f3 [tên nhánh]
```

Bây giờ cũng có thể chạy `git Worktree Prune` một cách an toàn.

# Chương 45: Điều khiển từ xa Git

Tham số -v, --	Chi tiết
verbose	Chạy chi tiết.
-m <master>	Đặt head thành nhánh <master> của remote
--mirror=fetch	Các tham chiếu sẽ không được lưu trữ trong không gian tên refs/remotes mà thay vào đó sẽ được phản ánh trong kho lưu trữ cục bộ
--mirror=push	git push sẽ hoạt động như thẻ --mirror đã được thông qua
--no-tags	git tìm nạp <name> không nhập thẻ từ repo từ xa
-t <branch>	Chỉ định điều khiển từ xa để chỉ theo dõi <branch>
-f	git get <name> được chạy ngay sau khi thiết lập điều khiển từ xa
--tags	git get <name> nhập mọi thẻ từ repo từ xa
-a, --auto	HEAD của biểu tượng-ref được đặt thành cùng nhánh với HEAD của điều khiển từ xa
-d, --xóa	Tất cả các giới thiệu được liệt kê sẽ bị xóa khỏi kho lưu trữ từ xa
--thêm vào	Thêm <name> vào danh sách các nhánh hiện được theo dõi (set-branches)
--thêm vào	Thay vì thay đổi một số URL, URL mới sẽ được thêm vào (set-url)
--tất cả	Đẩy tất cả các nhánh.
--xóa bỏ	Tất cả các url phù hợp với <url> đều bị xóa. (đặt url)
--xô	URL đẩy được thao tác thay vì tìm nạp URL
-N	Đầu từ xa không được truy vấn trước bằng git ls-remote <name>, thông tin được lưu trong bộ nhớ cache sẽ được sử dụng thay vì
--chạy khô	báo cáo những cảnh nào sẽ được cắt tia, nhưng thực tế không cắt tia chúng
--cắt tia	Xóa các nhánh từ xa không có bản sao cục bộ

## Mục 45.1: Hiển thị kho lưu trữ từ xa

Để liệt kê tất cả các kho lưu trữ từ xa được định cấu hình, hãy sử dụng git remote.

Nó hiển thị tên viết tắt (bí danh) của từng bộ điều khiển từ xa mà bạn đã cấu hình.

```
$ git từ xa
phần thường
cao cấpPro
nguồn gốc
```

Để hiển thị thông tin chi tiết hơn, có thể sử dụng cờ --verbose hoặc -v. Đầu ra sẽ bao gồm URL và loại điều khiển từ xa (đẩy hoặc kéo):

```
$ git từ xa -v
premiumPro      https://github.com/user/CatClickerPro.git (tìm nạp)
premiumPro https://github.com/user/CatClickerPro.git (đẩy)
premium       https://github.com/user/CatClicker.git (tìm nạp)
cao cấp       https://github.com/user/CatClicker.git (đẩy)
xuất xứ       https://github.com/ud/starter.git (tìm nạp)
cao cấp       https://github.com/ud/starter.git (đẩy)
```

## Phần 45.2: Thay đổi url từ xa của kho Git của bạn

Bạn có thể muốn thực hiện việc này nếu kho lưu trữ từ xa được di chuyển. Lệnh thay đổi url từ xa là:

```
url thiết lập từ xa git
```

Phải mất 2 đối số: tên từ xa hiện có (nguồn gốc, ngược dòng) và url.

Kiểm tra url từ xa hiện tại của bạn:

```
git remote -v  
Origin https://bitbucket.com/develop/myrepo.git (tìm nạp) https://bitbucket.com/  
develop/myrepo.git (push) Origin
```

Thay đổi url từ xa của bạn:

```
git nguồn gốc set-url từ xa https://localserver/develop/myrepo.git
```

Kiểm tra lại url từ xa của bạn:

```
git remote -v  
Origin https://localserver/develop/myrepo.git (tìm nạp) https://localserver/  
develop/myrepo.git (push) Origin
```

## Mục 45.3: Xóa kho lưu trữ từ xa

Xóa điều khiển từ xa có tên <name>. Tất cả các nhánh theo dõi từ xa và cài đặt cấu hình cho điều khiển từ xa đều bị xóa.

Để xóa nhà phát triển kho lưu trữ từ xa :

```
git từ xa rm dev
```

## Mục 45.4: Thêm kho lưu trữ từ xa

Để thêm điều khiển từ xa, hãy sử dụng `git remote add` trong thư mục gốc của kho lưu trữ cục bộ của bạn.

Để thêm kho lưu trữ Git từ xa <url> dưới dạng tên viết tắt dễ dàng <name>, hãy sử dụng

```
git từ xa thêm <tên> <url>
```

Lệnh `git get <name>` sau đó có thể được sử dụng để tạo và cập nhật các nhánh theo dõi từ xa <tên>/<chi nhánh>.

## Phần 45.5: Hiển thị thêm thông tin về kho lưu trữ từ xa

Bạn có thể xem thêm thông tin về kho lưu trữ từ xa bằng `git remote show <remote repo alias>`

```
nguồn gốc chương trình từ xa git
```

kết quả:

```
nguồn gốc từ xa  
URL tìm nạp: https://localserver/develop/myrepo.git URL đầy: https://  
localserver/develop/myrepo.git Nhánh HEAD: master
```

```
Chi nhánh từ xa: được  
bậc thầy theo dõi
```

```
Các nhánh cục bộ được định cấu hình cho 'git pull':  
bậc thầy hợp nhất với chủ từ xa
```

Các giới thiệu cục bộ được định cấu hình cho 'git push':  
bậc thầy đầy đủ làm chủ (cập nhật)

## Mục 45.6: Đổi tên kho lưu trữ từ xa

Đổi tên điều khiển từ xa có tên <old> thành <new>. Tất cả các nhánh theo dõi từ xa và cài đặt cấu hình cho điều khiển từ xa được cập nhật.

Để đổi tên nhánh từ xa dev thành dev1 :

đổi tên từ xa git dev dev1

## Chương 46: Lưu trữ tệp lớn Git (LFS)

### Mục 46.1: Khai báo một số loại tệp nhất định để lưu trữ bên ngoài

Quy trình công việc phổ biến khi sử dụng Git LFS là khai báo tệp nào bị chặn thông qua hệ thống dựa trên quy tắc, giống như tệp `.gitignore`.

Phần lớn thời gian, các ký tự đại diện được sử dụng để chọn một số loại tệp nhất định để theo dõi chung.

Ví dụ: theo dõi `git lfs "*.psd"`

Khi một tệp phù hợp với mẫu trên được thêm vào, chúng đã được cam kết, khi nó được đẩy đến điều khiển từ xa, nó sẽ được tải lên riêng biệt, với một con trỏ thay thế tệp trong kho lưu trữ từ xa.

Sau khi một tệp được theo dõi bằng `lfs`, tệp `.gitattributes` của bạn sẽ được cập nhật tương ứng. Github khuyên bạn nên cam kết tệp `.gitattributes` cục bộ của mình, thay vì làm việc với tệp `.gitattributes` toàn cầu, để giúp đảm bảo bạn không gặp bất kỳ vấn đề nào khi làm việc với các dự án khác nhau.

### Mục 46.2: Đặt cấu hình LFS cho tất cả các bản sao

Để đặt các tùy chọn LFS áp dụng cho tất cả các bản sao, hãy tạo và cam kết một tệp có tên `.lfsconfig` ở thư mục gốc của kho lưu trữ. Tệp này có thể chỉ định các tùy chọn LFS giống như cách được phép trong `.git/config`.

Ví dụ: để loại trừ một tệp nhất định khỏi các lần tìm nạp LFS theo mặc định, hãy tạo và cam kết `.lfsconfig` bằng cách sau nội dung:

```
[lfs]
getexclude = Thực sựBigFile.wav
```

### Mục 46.3: Cài đặt LFS

Tải xuống và cài đặt qua Homebrew hoặc từ [trang web](#).

Đối với

```
Brew, brew install git-
lfs git lfs install
```

Thông thường, bạn cũng sẽ cần thực hiện một số thiết lập trên dịch vụ lưu trữ điều khiển từ xa của mình để cho phép nó hoạt động với `lfs`. Điều này sẽ khác nhau đối với mỗi máy chủ, nhưng có thể sẽ chỉ đánh dấu vào ô cho biết bạn muốn sử dụng `git lfs`.

# Chương 47: Bản vá Git

Tham số	Chi tiết
(<mbox> <Maildir>)...	Danh sách các tệp hộp thư để đọc các bản vá. Nếu bạn không cung cấp đối số này, lệnh sẽ đọc từ đầu vào tiêu chuẩn. Nếu bạn cung cấp các thư mục, chúng sẽ được coi là Maildirs.
-s, --signoff	Thêm dòng Signed-off-by: vào thông báo cam kết, sử dụng danh tính người gửi của chính bạn.
-q, --quiet	Hãy yên lặng. Chỉ in các thông báo lỗi.
-u, --utf8	Chuyển cờ -u tới <code>git mailinfo</code> . Thông báo nhật ký cam kết được đề xuất lấy từ e-mail được mã hóa lại thành mã hóa UTF-8 (biến cấu hình <code>i18n.commitencoding</code> có thể được sử dụng để chỉ định mã hóa ưu tiên của dự án nếu nó không phải là UTF-8). Bạn có thể sử dụng <code>--no-utf8</code> để ghi đè lên điều này.
--no-utf8	Chuyển cờ -n tới <code>git mailinfo</code> .
-3, --3đường	Khi bản vá không được áp dụng rõ ràng, hãy quay lại hợp nhất 3 chiều nếu bản vá ghi lại danh tính của các đóm màu mà nó được cho là áp dụng và chúng tôi có sẵn các đóm màu đó tại địa phương.
--ignore-date, --ignore-space-change, -- bỏ qua khoảng trống, -- khoảng trống=<option>, -C<n>, -p<n>, -- thư mục=<dir>, -- loại trừ=<đường dẫn>, -- bao gồm=<đường dẫn>, --reject	Những cờ này được chuyển tới chương trình git áp dụng bản vá.
-định dạng bản vá	Theo mặc định, lệnh sẽ cố gắng tự động phát hiện định dạng bản vá. Tùy chọn này cho phép người dùng bỏ qua việc phát hiện tự động và chỉ định định dạng bản vá mà (các) bản vá sẽ được hiểu là. Các định dạng hợp lệ là mbox, stgit, stgit-series và hg.
-i, --tương tác	Chạy tương tác.
--committer-date-is-author-date	Theo mặc định, lệnh ghi ngày trong thông báo email làm ngày tác giả cam kết và sử dụng thời gian tạo cam kết làm ngày của người cam kết. Điều này cho phép người dùng nói đổi về ngày của người gửi bằng cách sử dụng cùng giá trị với ngày của tác giả.
--bỏ qua ngày	Theo mặc định, lệnh ghi ngày trong thông báo email làm ngày tác giả cam kết và sử dụng thời gian tạo cam kết làm ngày của người cam kết. Điều này cho phép người dùng nói đổi về ngày tác giả bằng cách sử dụng cùng giá trị với ngày của người gửi.
-nhảy	Bỏ qua bản vá hiện tại. Điều này chỉ có ý nghĩa khi khởi động lại một bản vá bị hủy bỏ.
-S[<keyid>], --gpg-sign[=<keyid>]	Cam kết ký hiệu GPG.
--tiếp tục, -r, --đã giải quyết	Sau khi một bản vá bị lỗi (ví dụ: cố gắng áp dụng bản vá xung đột), người dùng đã áp dụng nó bằng tay và tệp chỉ mục lưu trữ kết quả của ứng dụng. Thực hiện một cam kết bằng cách sử dụng quyền tác giả và nhật ký cam kết được trích xuất từ thông báo email và tệp chỉ mục hiện tại, rồi tiếp tục.
--resolvemsg=<tin nhắn>	Khi xảy ra lỗi bản vá, <msg> sẽ được in ra màn hình trước khi thoát. Điều này sẽ ghi đè thông báo tiêu chuẩn thông báo cho bạn sử dụng <code>--continue</code> hoặc <code>--skip</code> để xử lý lỗi. Điều này chỉ dành cho sử dụng nội bộ giữa <code>git rebase</code> và <code>git am</code> .
-Huỷ bỏ	Khôi phục nhánh ban đầu và hủy bỏ thao tác vá lỗi.

## Mục 47.1: Tạo bản vá

Để tạo một bản vá, có hai bước.

1. Thực hiện các thay đổi của bạn và cam kết chúng.

2. Chạy `git format-patch <commit-reference>` để chuyển đổi tất cả các cam kết kể từ cam kết `<commit-reference>` (không bao gồm nó) thành các tệp vá.

Ví dụ: nếu các bản vá phải được tạo từ hai lần xác nhận mới nhất:

```
bản vá định dạng git ĐẦU ~~
```

Điều này sẽ tạo 2 tệp, một tệp cho mỗi lần xác nhận kể từ HEAD~~, như thế này:

```
0001-hello_world.patch 0002-bắt  
đầu.patch
```

## Mục 47.2: Áp dụng các bản vá

Chúng tôi có thể sử dụng `git apply some.patch` để áp dụng các thay đổi từ tệp .patch cho thư mục làm việc hiện tại của bạn. Họ sẽ không được dàn dựng và cần phải được cam kết.

Để áp dụng một bản vá dưới dạng một cam kết (với thông báo cam kết của nó), hãy sử dụng

```
git am some.patch
```

Để áp dụng tất cả các tệp vá cho cây:

```
git am *.patch
```

# Chương 48: Thống kê Git

Tham số	Chi tiết
-n, --đánh số	Sắp xếp đầu ra theo số lượng cam kết của mỗi tác giả thay vì thứ tự bảng chữ cái
-s, --tóm tắt	Chỉ cung cấp bản tóm tắt số lượng cam kết
-e, --email	Hiển thị địa chỉ email của từng tác giả
--format[=<format>]	Thay vì chủ đề cam kết, hãy sử dụng một số thông tin khác để mô tả từng cam kết. <format> có thể là bất kỳ chuỗi nào được tùy chọn --format của git log chấp nhận.
<phạm vi sửa đổi>	Linewrap đầu ra bằng cách gói mỗi dòng theo chiều rộng. Dòng đầu tiên của mỗi mục nhập -w[<width>[,<indent1>[,<indent2>]]] được thụt lề bằng số khoảng trắng thụt lề và các dòng tiếp theo được thụt lề bằng khoảng trắng thụt lề2 .
[--] <đường dẫn>	Chỉ hiển thị các cam kết trong phạm vi sửa đổi được chỉ định. Mặc định cho toàn bộ lịch sử cho đến lần cam kết hiện tại.
	Chỉ hiển thị các cam kết giải thích đường dẫn khớp với tệp xuất hiện như thế nào.
	Các đường dẫn có thể cần phải có tiền tố "-- " để tách chúng khỏi các tùy chọn hoặc phạm vi sửa đổi.

## Mục 48.1: Dòng mã cho mỗi nhà phát triển

```
git ls-tree -r ĐẦU | sed -Ee 's/^.{53}//' | \ trong khi đọc tên tệp;
làm tập tin "$filename"; xong | \ grep -E ': .*text' | sed -E -e 's/: .*//'
| \ trong khi đọc tên tệp; làm git dò lỗi --line-porcelain
"$filename"; xong | \ sed -n 's/^author //p' | \ sắp xếp | uniq -c | sắp xếp -rn
```

## Mục 48.2: Liệt kê từng chi nhánh và ngày sửa đổi lần cuối

```
cho k trong nhánh `git -a | sed s/^...`; làm echo -e `git log -1 --pretty=format:"%Cgreen%ci %Cblue%cr%Creset" $k --`\\t"$k";done |
loại
```

## Mục 48.3: Cam kết của mỗi nhà phát triển

Git shortlog được sử dụng để tóm tắt các kết quả đầu ra của nhật ký git và nhóm các cam kết theo tác giả.

Theo mặc định, tất cả các thông báo cam kết đều được hiển thị nhưng đối số **--summary** hoặc **-s** bỏ qua các thông báo và đưa ra danh sách các tác giả cùng với tổng số lần cam kết của họ.

**--numbered** hoặc **-n** thay đổi thứ tự từ bảng chữ cái (theo tác giả tăng dần) sang số lần xác nhận giảm dần.

```
git shortlog -sn          #Tên và số lần xác nhận

git shortlog -sne         #Names cùng với id email của họ và Số lần xác nhận

hoặc

git log --pretty=format%ae \ | tró măt --
'{ ++c[$0]; } END { for(cc in c) printf "%5d %s\n",c[cc],cc; }'
```

Lưu ý: Các cam kết của cùng một người không thể được nhóm lại với nhau khi tên và/hoặc địa chỉ email của họ được đánh vần khác nhau. Ví dụ John Doe và Johnny Doe sẽ xuất hiện riêng trong danh sách. Để giải quyết vấn đề này,

hãy tham khảo tính năng .mailmap .

#### Mục 48.4: Cam kết mỗi ngày

```
git log --pretty=format:"%ai" | ôi '{in " : "$1}" | sắp xếp -r | uniq -c
```

#### Mục 48.5: Tổng số lần xác nhận trong một nhánh

```
git log --pretty=oneline |wc -l
```

#### Mục 48.6: Liệt kê tất cả các cam kết ở định dạng đẹp

```
git log --pretty=format:"%Cgreen%ci %Cblue%cn %Cgreen%cr%Creset %s"
```

Điều này sẽ cung cấp một cái nhìn tổng quan tốt đẹp về tất cả các cam kết (1 trên mỗi dòng) với ngày, người dùng và thông báo cam kết.

Tùy chọn `--pretty` có nhiều phần giữ chỗ, mỗi phần bắt đầu bằng %. Tất cả các tùy chọn có thể được tìm thấy [ở đây](#).

#### Mục 48.7: Tìm tất cả kho lưu trữ Git cục bộ trên máy tính

Để liệt kê tất cả các vị trí git trên của bạn, bạn có thể chạy như sau

```
tìm $HOME -type d -name ".git"
```

Giả sử bạn có `định vị`, việc này sẽ nhanh hơn nhiều:

```
định vị .git |grep git$
```

Nếu bạn có gnu `định vị` hoặc `locate`, điều này sẽ chỉ chọn các thư mục git:

```
định vị -ber \\.git$
```

#### Mục 48.8: Hiển thị tổng số commit của mỗi tác giả

Để có được tổng số cam kết mà mỗi nhà phát triển hoặc cộng tác viên đã thực hiện trên một kho lưu trữ, bạn chỉ cần sử dụng `git shortlog`:

```
git shortlog -s
```

trong đó cung cấp tên tác giả và số lượng cam kết của mỗi người.

Ngoài ra, nếu bạn muốn tính kết quả trên tất cả các nhánh, hãy thêm cờ `--all` vào lệnh:

```
git shortlog -s --all
```

# Chương 49: git gửi email

## Mục 49.1: Sử dụng git send-email với Gmail

Bối cảnh: nếu bạn làm việc trên một dự án như nhân Linux, thay vì thực hiện yêu cầu kéo, bạn sẽ cần phải gửi cam kết của bạn với một listserv để xem xét. Mục này nêu chi tiết cách sử dụng email git-send với Gmail.

Thêm phần sau vào tệp .gitconfig của bạn:

```
[gửi email]
smtpserver = smtp.googlemail.com
smtpencryption = tls
smtpserverport = 587
smtpuser = name@gmail.com
```

Sau đó, trên web: Truy cập Google -> Tài khoản của tôi -> Ứng dụng & trang web được kết nối -> Cho phép ứng dụng kèm an toàn hơn -> BẬT

Để tạo một bộ bản vá:

```
git format-patch HEAD~~~~ --subject-prefix="PATCH <project-name>"
```

Sau đó gửi các bản vá đến listserv:

```
git send-email --annotate --to project-developers-list@listserve.example.com 00*.patch
```

Để tạo và gửi phiên bản cập nhật (phiên bản 2 trong ví dụ này) của bản vá:

```
git format-patch -v 2 HEAD~~~~ git send-
email --to project-developers-list@listserve.example.com v2-00*.patch
```

## Mục 49.2: Soạn thảo

--từ --	* Email từ:
[no-]đến --	* Gửi email tới:
[no-]cc --	* Email Cc:
[no-]bcc --	* Email Bcc:
subject --	* Email "Tiêu đề:"
in-reply-to --	* Email "Trả lời-đến:"
[no-]xmailer --	* Thêm tiêu đề "X-Mailer:" (mặc định).
[no-]annotate --	* Xem lại từng bản vá sẽ được gửi trong trình chỉnh sửa.
soạn --	* Mở trình soạn thảo để giới thiệu.
compose-encoding --8bit-	* Mã hóa giả sử để giới thiệu.
encoding --transfer-	* Mã hóa để giả sử thư 8 bit nếu không được khai báo
mã hóa	* Chuyển mã hóa để sử dụng (có thể in trích dẫn, 8bit, base64)

## Mục 49.3: Gửi bản vá qua thư

Giả sử bạn có nhiều cam kết đối với một dự án (ở đây là ulogd2, chi nhánh chính thức là git-svn) và bạn muốn gửi bản vá của bạn tới Danh sách gửi thư devel@netfilter.org. Để làm như vậy, chỉ cần mở shell ở thư mục gốc git và sử dụng:

```
git format-patch --stat -p --raw --signoff --subject-prefix="ULOGD PATCH" -o /tmp/ulogd2/ -n git-
svn
```

```
git send-email --compose --no-chain-reply-to --to devel@netfilter.org /tmp/ulogd2/
```

Lệnh đầu tiên sẽ tạo một loạt thư từ các bản vá trong /tmp/ulogd2/ với báo cáo thống kê và lệnh thứ hai sẽ bắt đầu trình soạn thảo của bạn soạn thư giới thiệu về bộ bản vá. Để tránh hàng loạt thư tê hại, người ta có thể sử dụng:

```
cấu hình git sendemail.chainreplyto false
```

[nguồn](#)

# Chương 50: Máy khách GUI Git

## Mục 50.1: gitk và git-gui

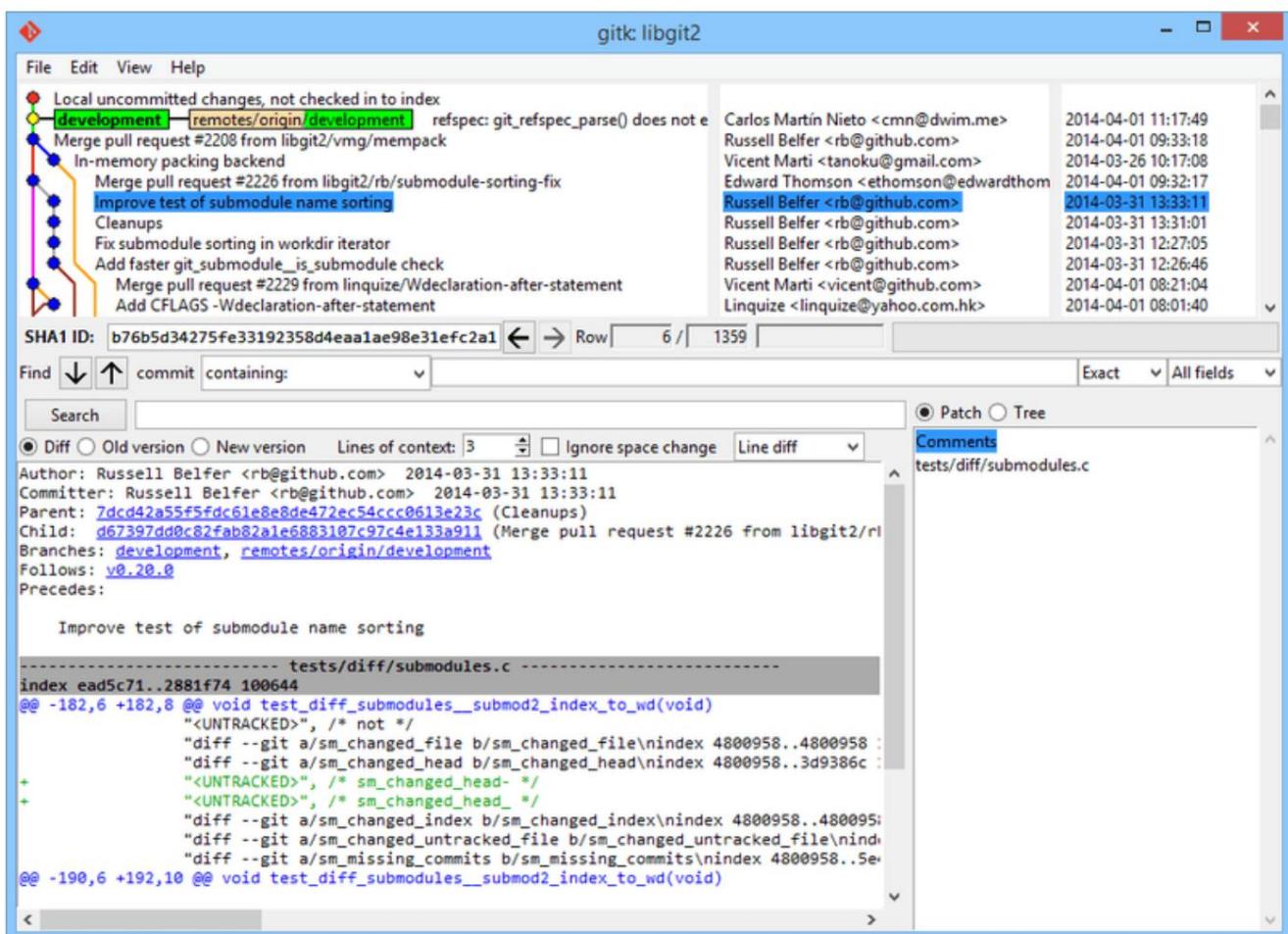
Khi cài đặt Git, bạn cũng nhận được các công cụ trực quan, gitk và git-gui.

gitk là trình xem lịch sử đồ họa. Hãy nghĩ về nó giống như một GUI shell mạnh mẽ trên git log và git grep. Đây là công cụ được sử dụng khi bạn đang cố gắng tìm kiếm điều gì đó đã xảy ra trong quá khứ hoặc trực quan hóa lịch sử dự án của mình.

Gitk dễ gọi nhất từ dòng lệnh. Chỉ cần cd vào kho Git và gõ:

```
$ gitk [tùy chọn nhật ký git]
```

Gitk chấp nhận nhiều tùy chọn dòng lệnh, hầu hết trong số đó được chuyển qua hành động nhật ký git cơ bản. Có lẽ một trong những cờ hữu ích nhất là cờ `--all`, cờ này yêu cầu gitk hiển thị các cam kết có thể truy cập được từ bất kỳ ref nào, không chỉ HEAD. Giao diện của Gitk trông như thế này:



Hình 1-1. Trình xem lịch sử gitk.

Ở trên cùng là thứ gì đó trông hơi giống đầu ra của git log --graph; mỗi dấu chấm thẻ hiện một cam kết, các dòng thẻ hiện mối quan hệ cha mẹ và các tham chiếu được hiển thị dưới dạng các hộp màu. Dấu chấm màu vàng biểu thị ĐẦU và dấu chấm màu đỏ biểu thị những thay đổi chưa trở thành cam kết. Ở phía dưới là chế độ xem cam kết đã chọn; các nhận xét và bản vá ở bên trái và chế độ xem tóm tắt ở bên phải. Ở giữa là tập hợp các điều khiển được sử dụng để tìm kiếm lịch sử.

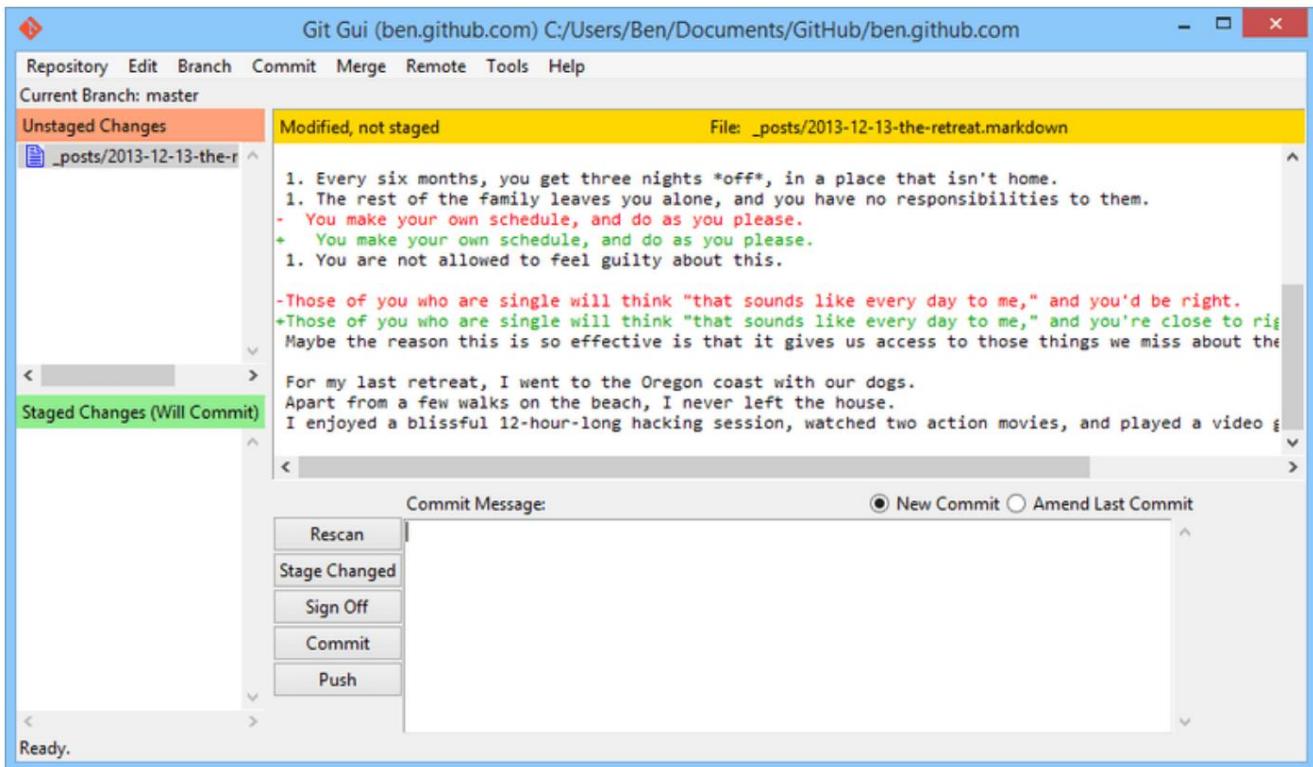
Bạn có thể truy cập nhiều chức năng liên quan đến git bằng cách nhấp chuột phải vào tên nhánh hoặc thông báo cam kết. Ví dụ: việc kiểm tra một nhánh khác hoặc chọn quả anh đào một cam kết có thể được thực hiện dễ dàng chỉ bằng một cú nhấp chuột.

Mặt khác, git-gui chủ yếu là một công cụ để tạo ra các cam kết. Nó cũng dễ gọi nhất từ dòng lệnh:

```
$ git gui
```

Và nó trông giống như thế này:

Công cụ cam kết git-gui .



Hình 1-2. Công cụ cam kết git-gui.

Bên trái là chỉ số; những thay đổi chưa được thực hiện ở trên cùng, những thay đổi được thực hiện ở phía dưới. Bạn có thể di chuyển toàn bộ tệp giữa hai trạng thái bằng cách nhấp vào biểu tượng của chúng hoặc bạn có thể chọn tệp để xem bằng cách nhấp vào tên của nó.

Ở trên cùng bên phải là chế độ xem khác biệt, hiển thị các thay đổi đối với tệp hiện được chọn. Bạn có thể tạo các khối riêng lẻ (hoặc các dòng riêng lẻ) bằng cách nhấp chuột phải vào khu vực này.

Ở phía dưới bên phải là khu vực thông báo và hành động. Nhập tin nhắn của bạn vào hộp văn bản và nhấp vào "Cam kết" để thực hiện điều gì đó tương tự như cam kết git. Bạn cũng có thể chọn sửa đổi cam kết cuối cùng bằng cách chọn nút radio "Sửa đổi", nút này sẽ cập nhật khu vực "Thay đổi theo giai đoạn" với nội dung của cam kết cuối cùng. Sau đó, bạn có thể chỉ cần thực hiện hoặc hủy thực hiện một số thay đổi, thay đổi thông báo cam kết và nhấp lại vào "Cam kết" để thay thế cam kết cũ bằng cam kết mới.

gitk và git-gui là ví dụ về các công cụ hướng nhiệm vụ. Mỗi trong số chúng được điều chỉnh cho một mục đích cụ thể (xem lịch sử và tạo các cam kết tương ứng) và bỏ qua các tính năng không cần thiết cho nhiệm vụ đó.

Nguồn: <https://git-scm.com/book/en/v2/Git-in-Other-Environments-Graphical-Interfaces>

## Phần 50.2: Máy tính để bàn GitHub

Trang web: <https://desktop.github.com> Miễn phí

Nền tảng: OS X và Windows

Nhà phát triển: [GitHub](#)

### Phần 50.3: Git Kraken

Trang web: <https://www.gitkraken.com> Giá: miễn phí

\$60/năm (miễn phí cho nguồn mở, giáo dục, phi lợi nhuận, khởi nghiệp hoặc sử dụng cá nhân)

Nền tảng: Linux, OS X, Windows

Nhà phát triển: Axosoft

### Mục 50.4: Cây nguồn

Trang web: <https://www.sourcetreeapp.com> Giá: miễn phí

(cần có tài khoản)

Nền tảng: OS X và Windows

Nhà phát triển: Atlassian

### Mục 50.5: Tiện ích mở rộng Git

Trang web: <https://gitextensions.github.io> Miễn phí

Nền tảng: Windows

### Mục 50.6: SmartGit

Trang web: <http://www.syntevo.com/smartgit/> Giá: Miễn phí

phi cho mục đích sử dụng phi thương mại. Giấy phép vĩnh viễn có giá 99 USD

Nền tảng: Linux, OS X, Windows

Nhà phát triển: [syntevo](#)

# Chương 51: Reflog - Khôi phục các cam kết không được hiển thị trong nhật ký git

## Mục 51.1: Phục hồi sau một đợt nỗi loạn tồi tệ

Giả sử bạn đã bắt đầu một cuộc nỗi loạn tương tác:

```
git rebase --interactive HEAD~20
```

và do nhầm lẫn, bạn đã xóa sạch hoặc bỏ đi một số cam kết mà bạn không muốn mất, nhưng sau đó đã hoàn thành quá trình rebase. Để khôi phục, hãy thực hiện `git reflog` và bạn có thể thấy một số kết quả như thế này:

```
aaaaaaaa HEAD@{0} rebase -i (kết thúc): quay lại refs/head/master bbbbbbbb HEAD@{1}
rebase -i (squash): Sửa lỗi phân tích cú pháp
...
cccccc HEAD@{n} rebase -i (bắt đầu): kiểm tra HEAD~20 dddddddd
HEAD@{n+1} ...
...
```

Trong trường hợp này, cam kết cuối cùng, dddddddd (hoặc `HEAD@{n+1}`) là phần cuối của nhánh tiền rebase của bạn. Do đó, để khôi phục cam kết đó (và tất cả các cam kết gốc, bao gồm cả những cam kết vô tình bị đè bẹp hoặc bị bỏ), hãy làm:

```
$ git kiểm tra HEAD@{n+1}
```

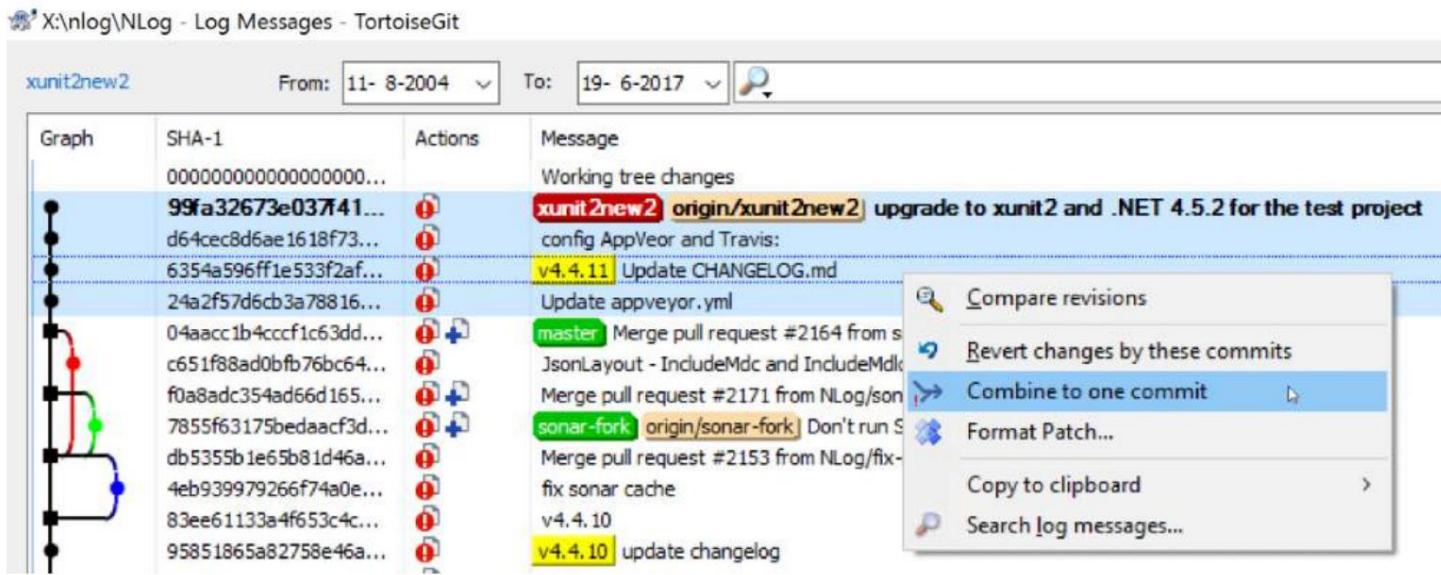
Sau đó, bạn có thể tạo một nhánh mới tại cam kết đó bằng `git checkout -b [branch]`. Xem Phân nhánh để biết thêm thông tin.

# Chương 52: TortoiseGit

## Mục 52.1: Cam kết của Squash

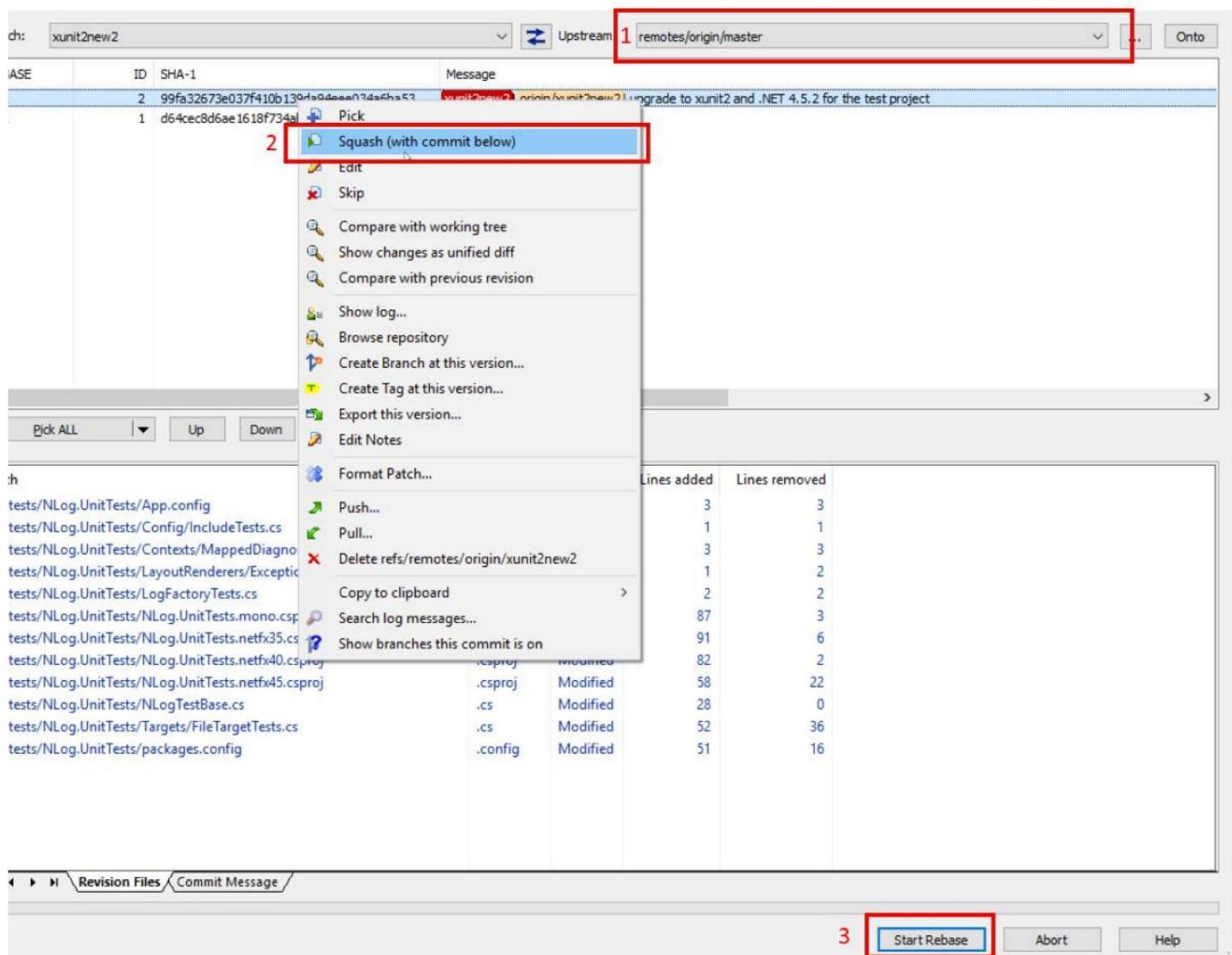
Cách dễ dàng

Điều này sẽ không hoạt động nếu có các cam kết hợp nhất trong lựa chọn của bạn



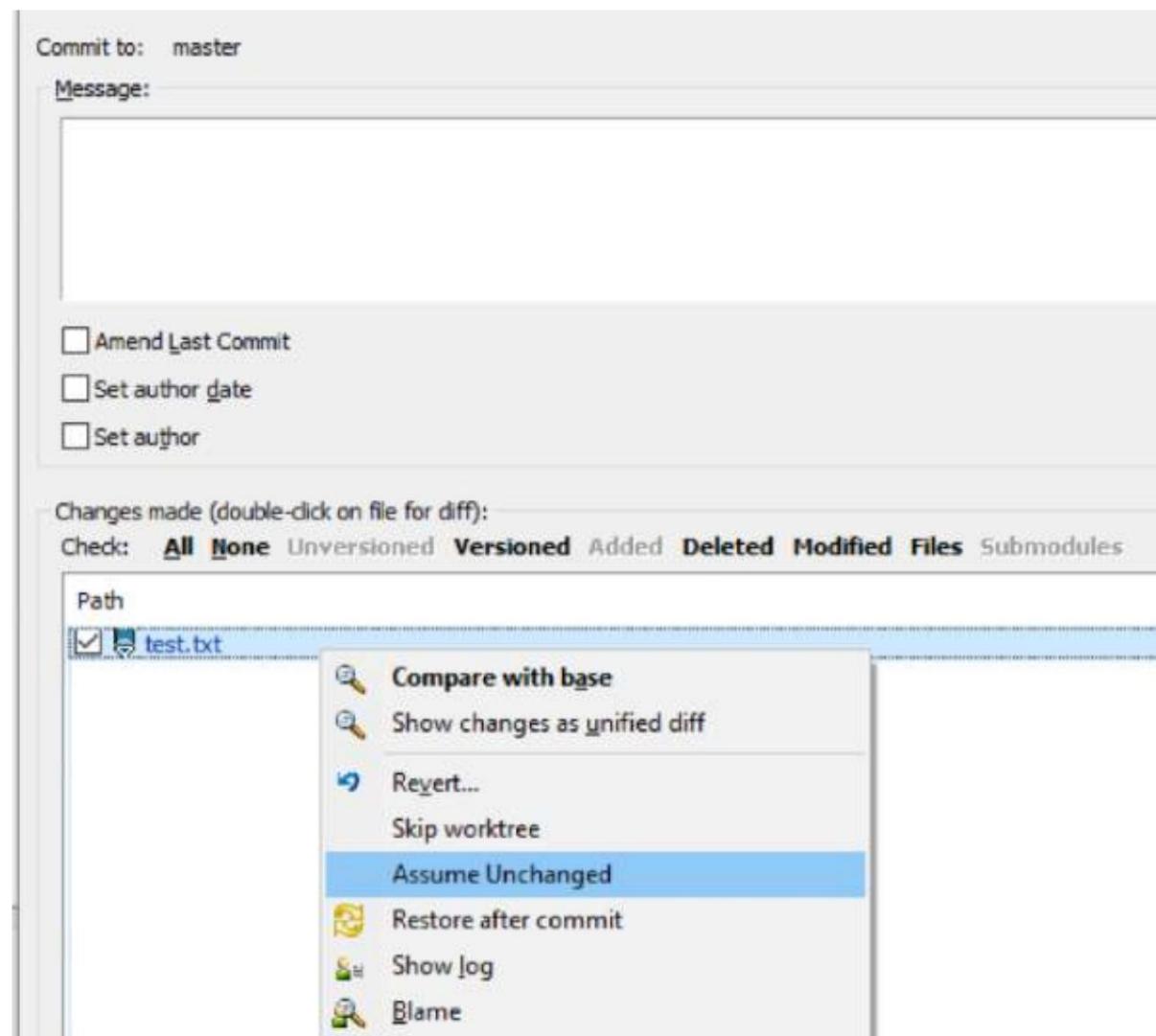
Cách nâng cao

Bắt đầu hộp thoại rebase:



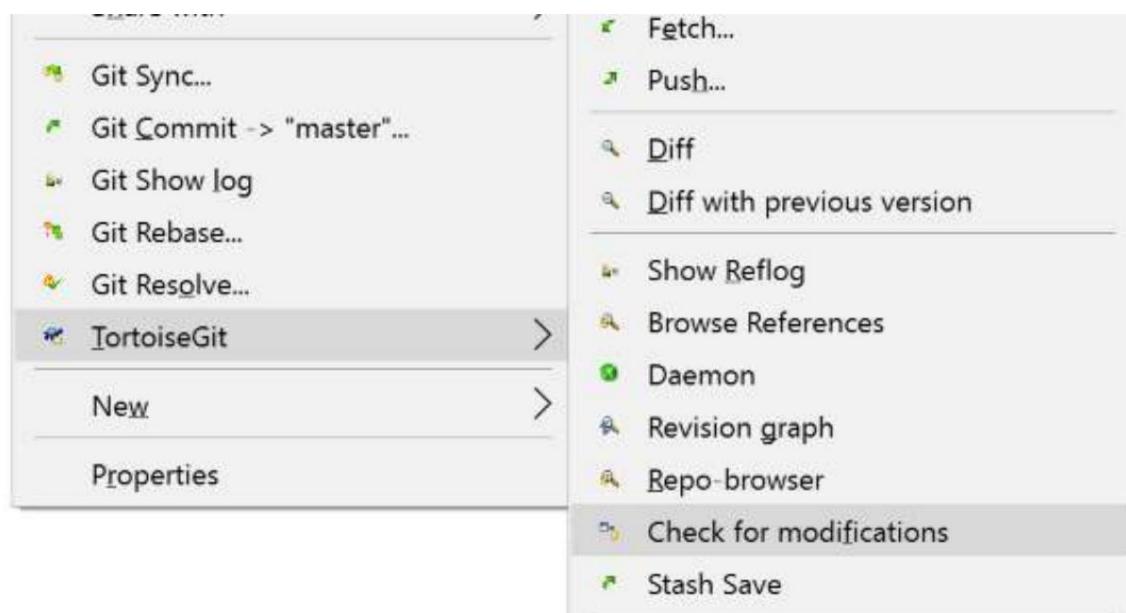
## Mục 52.2: Giả sử không thay đổi

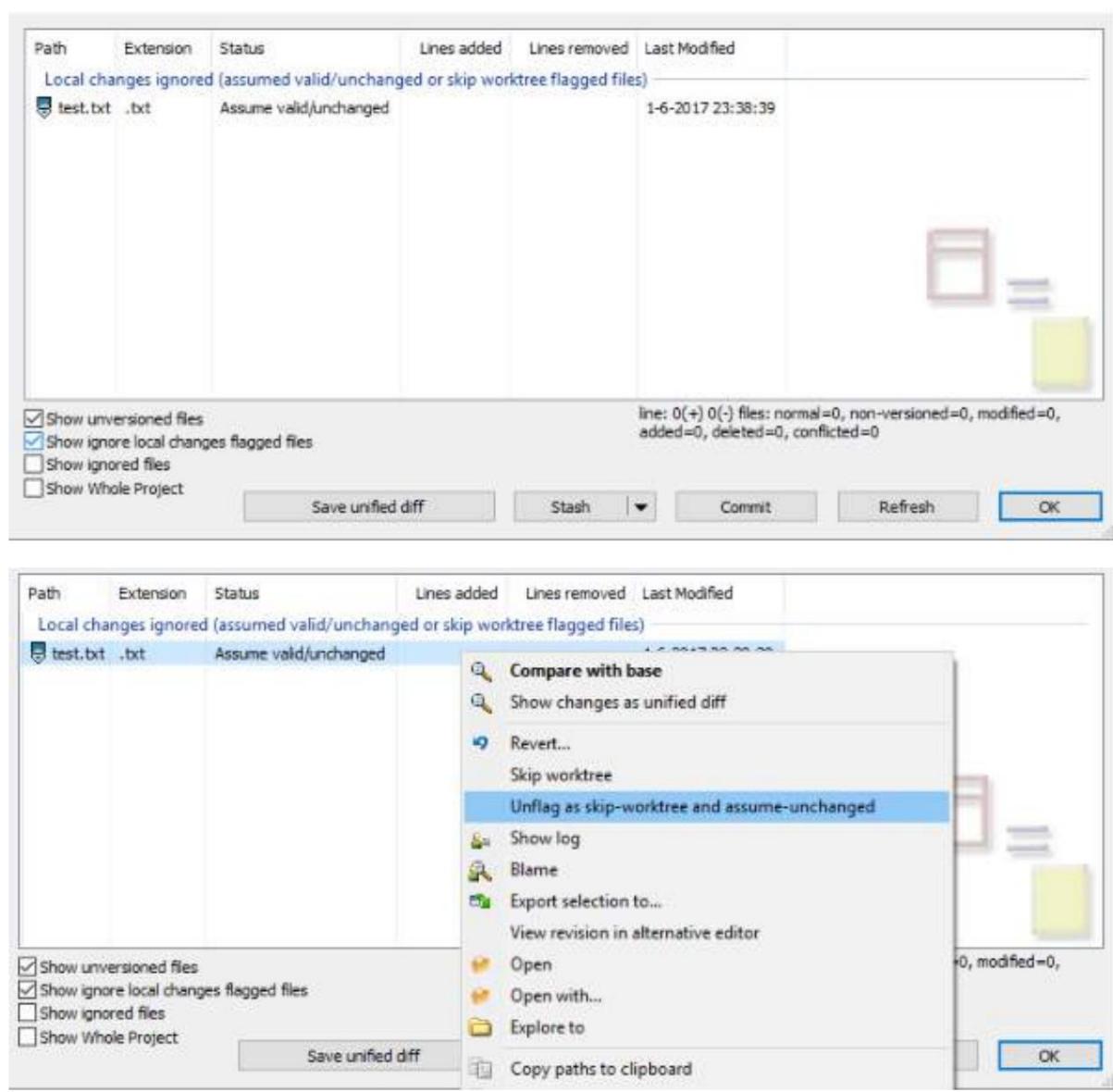
Nếu một tệp bị thay đổi nhưng bạn không muốn cam kết nó, hãy đặt tệp đó là "Giả sử không thay đổi"



Hoàn nguyên "Giả sử không thay đổi"

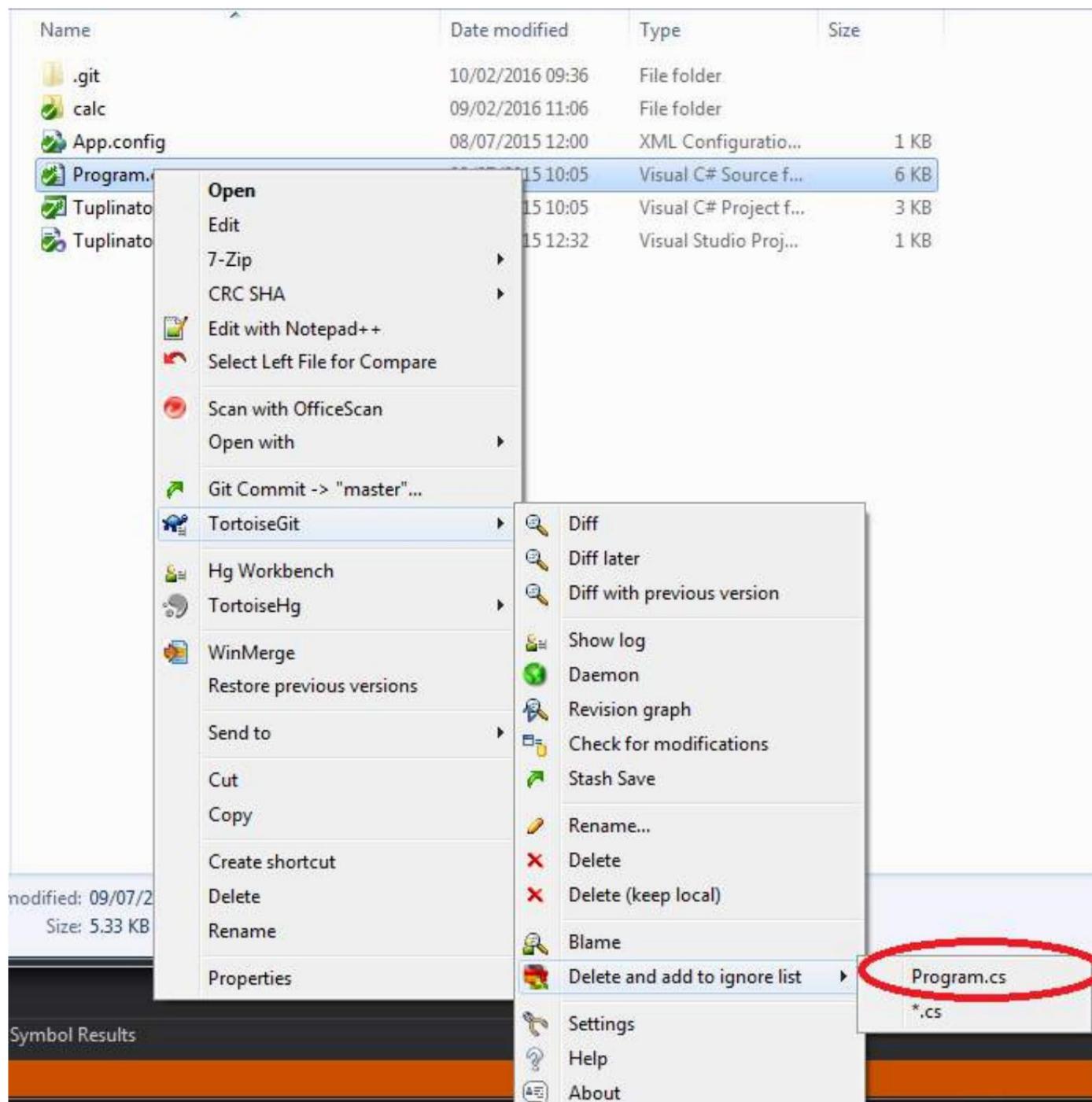
Cần một số bước:





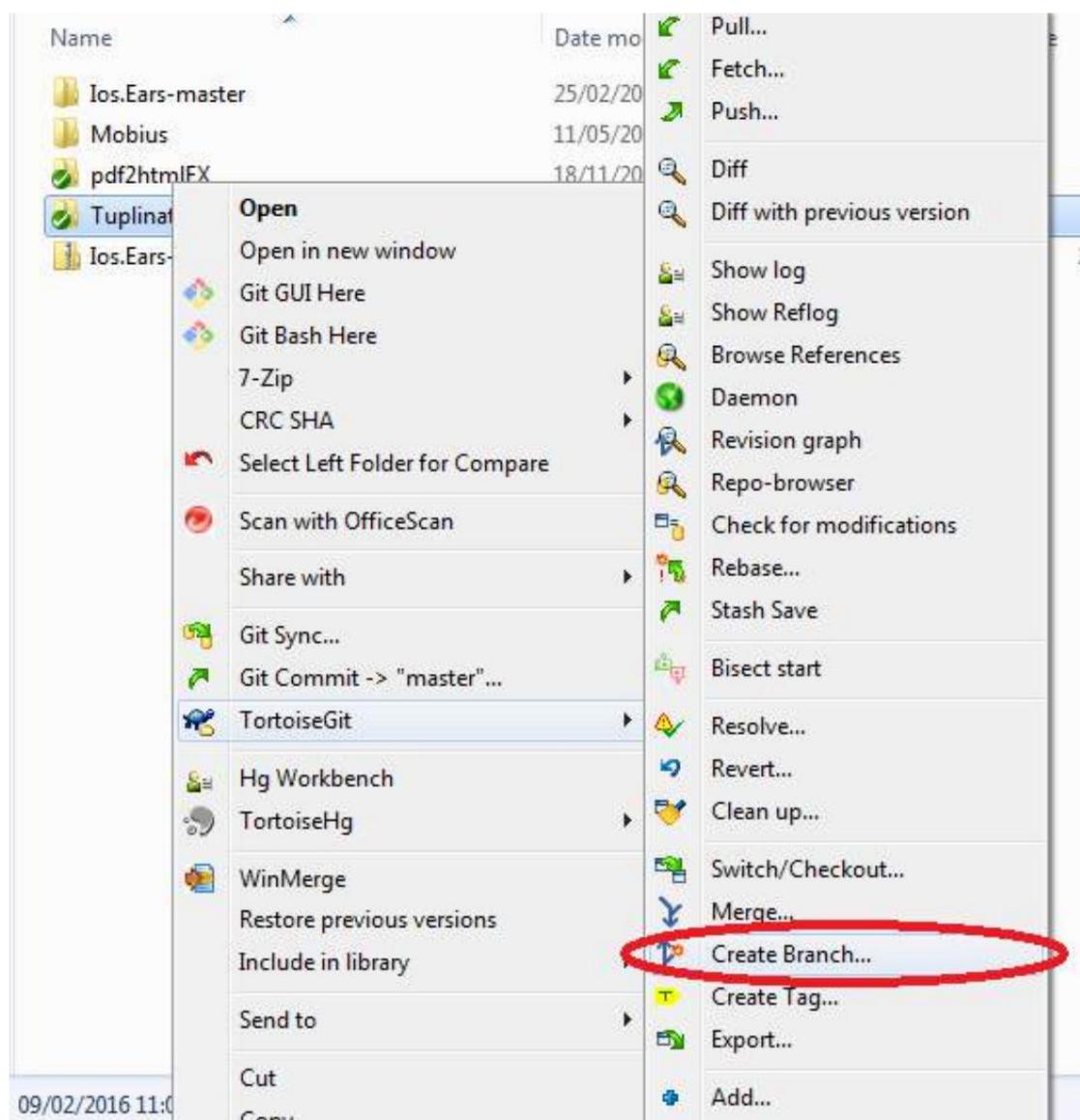
### Mục 52.3: Bỏ qua các tập tin và thư mục

Những người đang sử dụng TortoiseGit UI nhập Chuột phải vào tệp (hoặc thư mục) bạn muốn bỏ qua -> TortoiseGit -> Xóa và thêm vào danh sách bỏ qua, tại đây bạn có thể chọn bỏ qua tất cả các tệp thuộc loại đó hoặc tệp cụ thể này -> hộp thoại sẽ bật ra Nhập vào Ok và bạn sẽ hoàn tất.

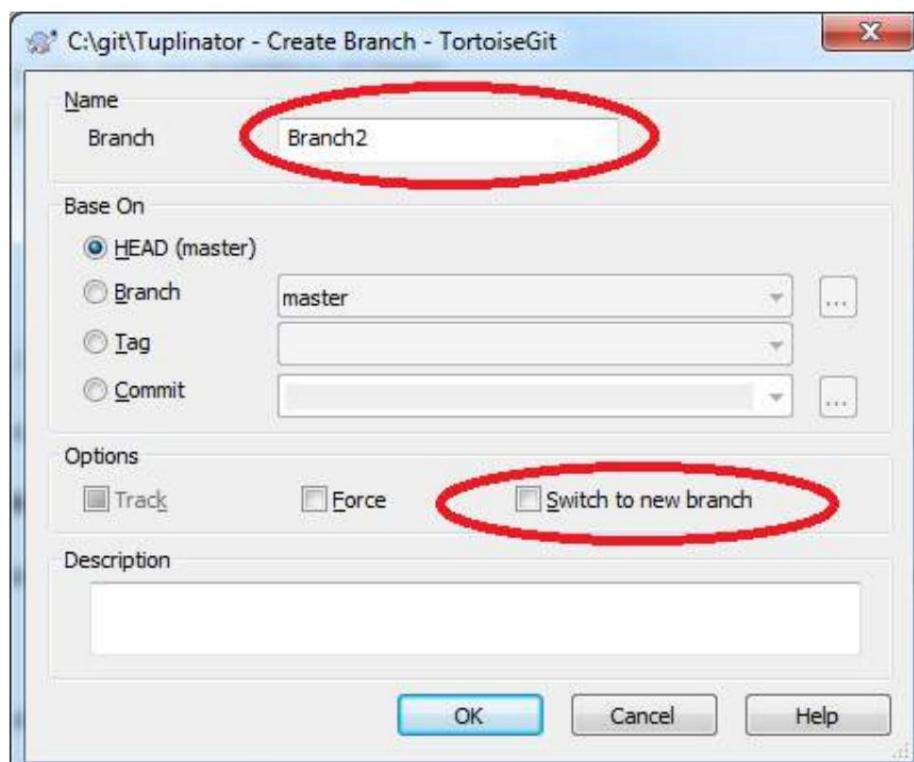


## Mục 52.4: Phân nhánh

Đối với những người đang sử dụng giao diện người dùng để phân nhánh, hãy nhấp chuột phải vào kho lưu trữ rồi Rùa Git -> Tạo nhánh...



Cửa sổ mới sẽ mở -> Đặt tên cho nhánh -> Đánh dấu vào ô Chuyển sang nhánh mới (Có thể bạn muốn bắt đầu làm việc với nó sau khi phân nhánh). -> Nhấn OK là xong.



# Chương 53: Hợp nhất bên ngoài và ditools

## Mục 53.1: Thiết lập KDiff3 làm công cụ hợp nhất

Phần sau đây nên được thêm vào tệp .gitconfig chung của bạn

```
công cụ
[hợp nhất] = kdiff3
[mergetool "kdiff3"] đường
dẫn = D:/Tệp chương trình (x86)/KDiff3/kdiff3.exe keepBackup =
false keepbackup = false
TrustExitCode = false
```

Nhớ đặt thuộc tính đường dẫn để trả đến thư mục bạn đã cài đặt KDiff3

## Mục 53.2: Thiết lập KDiff3 làm công cụ di

```
[diff]
tool = kdiff3
guitool = kdiff3
[difftool "kdiff3"] path
= D:/Program Files (x86)/KDiff3/kdiff3.exe cmd = \"D:/Program
Files (x86)/KDiff3/kdiff3.exe\" \"$LOCAL\" \"$REMOTE\"
```

## Phần 53.3: Thiết lập IntelliJ IDE làm công cụ hợp nhất (Windows)

```
[hợp
nhất] công cụ =
intellij [hợp nhất công cụ "intellij"]
cmd = cmd \"/C D:\\workspace\\tools\\symlink\\idea\\bin\\idea.bat merge $(cd $(dirname \"$LOCAL\") && pwd)$(basename \"$LOCAL\") $(cd $(dirname \"$REMOTE\") && pwd)$(basename \"$REMOTE\") $(cd $(dirname \"$BASE\") && pwd)$(basename \"$BASE\") $ (cd $(dirname \"$MERGED\") && pwd)$(basename \"$MERGED\")\" keepBackup = false keepbackup = false TrustExitCode = true
```

Điều đáng chú ý ở đây là thuộc tính cmd này không chấp nhận bất kỳ ký tự lạ nào trong đường dẫn. Nếu vị trí cài đặt IDE của bạn có các ký tự lạ trong đó (ví dụ: nó được cài đặt trong Tệp chương trình (x86), bạn sẽ phải tạo một liên kết tương ứng

## Mục 53.4: Thiết lập IntelliJ IDE làm công cụ di động (Windows)

```
công cụ
[khác biệt] =
intellij guitool = intellij
[difftool "intellij"] đường
dẫn = D:/Tệp chương trình (x86)/JetBrains/IntelliJ IDEA 2016.2/bin/idea.bat cmd = cmd \"/C D:\\workspace\\
\\tools\\symlink\\idea\\bin\\idea.bat diff $(cd $(dirname \"$LOCAL\")) && pwd)$(tên cơ sở \"$LOCAL\") $(cd $(dirname \"$REMOTE\")) && pwd)$(tên cơ sở \"$REMOTE\")\"
```

Điều đáng chú ý ở đây là thuộc tính cmd này không chấp nhận bất kỳ ký tự lạ nào trong đường dẫn. Nếu vị trí cài đặt IDE của bạn có các ký tự lạ trong đó (ví dụ: nó được cài đặt trong Tệp chương trình (x86), bạn sẽ phải tạo một liên kết tương ứng

## Mục 53.5: Thiết lập Beyond Compare

Bạn có thể đặt đường dẫn đến bcomp.exe

```
git config --global DiffTool.bc3.path 'c:\Program Files (x86)\Beyond Compare 3\bcomp.exe'
```

và cấu hình bc3 làm mặc định

```
cấu hình git --global diff.tool bc3
```

# Chương 54: Cập nhật tên đối tượng trong

## Thẩm quyền giải quyết

### Mục 54.1: Cập nhật tên đối tượng trong tài liệu tham khảo

Sử dụng

Cập nhật tên đối tượng được lưu trữ trong tài liệu tham khảo

TÓM TẮT

```
git update-ref [-m <reason>] (-d <ref> [<oldvalue>] | [--no-deref] [--create-reflog] <ref> <newvalue> [<oldvalue>] | - -stdin [-z])
```

Cú pháp chung

- Hủy tham chiếu các tham chiếu tượng trưng, cập nhật đầu nhánh hiện tại thành đối tượng mới.

```
git cập nhật-ref ĐẦU <giá trị mới>
```

- Lưu trữ giá trị mới trong ref, sau khi xác minh rằng giá trị hiện tại của ref khớp với giá trị cũ.

```
git cập nhật-ref refs/head/master <newvalue> <oldvalue>
```

Cú pháp trên chỉ cập nhật đầu nhánh chính thành giá trị mới nếu giá trị hiện tại của nó là giá trị cũ.

Sử dụng cờ -d để xóa <ref> có tên sau khi xác minh nó vẫn chưa <oldvalue>.

Sử dụng --create-reflog, update-ref sẽ tạo một reflog cho mỗi ref ngay cả khi một ref thông thường không được tạo.

Sử dụng cờ -z để chỉ định ở định dạng kết thúc NUL, có các giá trị như cập nhật, tạo, xóa, xác minh.

Cập nhật

Đặt <ref> thành <newvalue> sau khi xác minh <oldvalue>, nếu được cung cấp. Chỉ định 0 <newvalue> để đảm bảo ref không tồn tại sau khi cập nhật và/hoặc 0 <oldvalue> để đảm bảo ref không tồn tại trước khi cập nhật.

Tạo nên

Tạo <ref> với <newvalue> sau khi xác minh nó không tồn tại. <newvalue> đã cho có thể không bằng 0.

Xóa bỏ

Xóa <ref> sau khi xác minh nó tồn tại với <oldvalue>, nếu có. Nếu được cung cấp, <oldvalue> có thể không bằng 0.

Xác minh

Xác minh <ref> với <oldvalue> nhưng không thay đổi nó. Nếu <oldvalue> bằng 0 hoặc bị thiếu thì ref không được tồn tại.

# Chương 55: Tên nhánh Git trên Bash Ubuntu

Tài liệu này đề cập đến tên nhánh của git trên thiết bị đầu cuối bash . Các nhà phát triển chúng tôi cần tìm tên nhánh git rất thường xuyên. Chúng ta có thể thêm tên nhánh cùng với đường dẫn đến thư mục hiện tại.

## Mục 55.1: Tên chi nhánh trong terminal

PS1 là gì

PS1 biểu thị Chuỗi nhắc 1. Đây là một trong những dấu nhắc có sẵn trong shell Linux/UNIX. Khi bạn mở terminal, nó sẽ hiển thị nội dung được xác định trong biến PS1 trong dấu nhắc bash của bạn. Để thêm tên nhánh vào dấu nhắc bash, chúng ta phải chỉnh sửa biến PS1 (giá trị đặt của PS1 trong ~/.bash\_profile).

Hiển thị tên chi nhánh git

Thêm các dòng sau vào ~/.bash\_profile của bạn

```
git_branch()
{ git nhánh 2> /dev/null | sed -e '/^[*]/d' -e 's/* \(.*)/ (\1)/*' }

export PS1="\u@\h \[\033[32m\] \w\[ \033[33m\]\$(git_branch)\[\033[00m\] \$ "
```

Hàm git\_branch này sẽ tìm tên nhánh mà chúng ta đang sử dụng. Khi hoàn tất những thay đổi này, chúng ta có thể điều hướng đến git repo trên thiết bị đầu cuối và có thể thấy tên chi nhánh.

Chương 56: Móc phía máy khách Git

Giống như nhiều Hệ thống kiểm soát phiên bản khác, Git có cách kích hoạt các tập lệnh tùy chỉnh khi một số hành động quan trọng nhất định xảy ra. Có hai nhóm hook này: phía máy khách và phía máy chủ. Các hook phía máy khách được kích hoạt bởi các hoạt động như cam kết và hợp nhất, trong khi các hook phía máy chủ chạy trên các hoạt động mạng như nhận các cam kết được đẩy. Bạn có thể sử dụng những chiếc móc này vì nhiều lý do.

## Mục 56.1: Móc đẩy trước Git

tập lệnh đẩy trước được gọi bởi `git Push` sau khi nó đã kiểm tra trạng thái từ xa, nhưng trước khi mọi thứ được đẩy.

Nếu tập lệnh này thoát với trạng thái khác 0 thì sẽ không có gì được đẩy.

Hook này được gọi với các tham số sau:

\$1 -- Tên của điều khiển từ xa mà quá trình đẩy đang được thực hiện (Ví dụ: nguồn gốc)  
\$2 -- URL mà việc đẩy đang được thực hiện (Ví dụ: <https://.git>)

Thông tin về các cam kết đang được đẩy được cung cấp dưới dạng dòng tới đầu vào tiêu chuẩn ở dạng:

<local\_ref> <local\_sha1> <remote\_ref> <remote\_sha1>

Giá trị mẫu:

```
local_ref = refs/heads/master local_sha1
= 68a07ee4f6af8271dc40caae6cc23f283122ed11 remote_ref = refs/heads/master
remote_sha1 =
efd4d512f34b11e3cf5c12433bbbedd4b1532716f
```

Ví dụ dưới đây về tập lệnh pre-push được lấy từ `pre-push.sample` mặc định, được tạo tự động khi lưu trữ mới được khởi tạo bằng `git init`

# Mẫu này cho thấy cách ngăn chặn việc đẩy các cam kết trong đó thông báo tường trình bắt đầu # bằng "WIP" (đang tiến hành).

từ xa="\$1"  
url="\$2"

trong khi **đọc** local ref local sha remote ref remote sha làm

nếu [ "\$local\_sha" = \$740 ] thì

# Xử lý việc xác

1

nếu

không thì [ "\$remote sha" = \$z40 ] thì

```
# Nhánh mới, kiểm tra tất cả các lần xác nhận  
range="$local_sha" else
```

```
# Cập nhật lên nhánh hiện có, kiểm tra các cam kết mới  
range="$remote_sha..$local_sha"
```

fi

```
# Kiểm tra WIP commit
commit=`git rev-list -n 1 --grep '^WIP' "$range"` if [ -n
"$commit" ] then echo
">&2
    "Tìm thấy WIP commit trong $local_ref, không đầy " lối ra 1

    fi
fi
xong

lối ra 0
```

## Mục 56.2: Lắp móc

Tất cả các hook đều được lưu trữ trong thư mục con hook của thư mục Git. Trong hầu hết các dự án, đó là .git/hooks.

Để kích hoạt tập lệnh hook, hãy đặt một tệp vào thư mục con hooks của thư mục .git được đặt tên phù hợp (không có bất kỳ phần mở rộng nào) và có thể thực thi được.

## Chương 57: Git rerere

`rerere` (tái sử dụng độ phân giải đã ghi) cho phép bạn yêu cầu git nhớ cách bạn giải quyết một xung đột lớn. Điều này cho phép nó được tự động giải quyết vào lần tiếp theo git gặp phải xung đột tương tự.

### Mục 57.1: Kích hoạt lại

Để kích hoạt `rerere`, hãy chạy lệnh sau:

```
$ git config --global rerere.enabled true
```

Điều này có thể được thực hiện trong một kho lưu trữ cụ thể cũng như trên toàn cầu.

# Chương 58: Thay đổi tên kho git

Nếu bạn thay đổi tên kho lưu trữ ở phía xa, chẳng hạn như github hoặc bitbucket, khi bạn đầy mã hiện có của mình, bạn sẽ thấy lỗi: Lỗi nghiêm trọng, không tìm thấy kho lưu trữ\*\*.

## Mục 58.1: Thay đổi cài đặt cục bộ

Đi đến thiết bị đầu cuối,

```
cd projectFolder git
remote -v (nó sẽ hiển thị url git trước đó) git remote
set-url Origin https://username@bitbucket.org/username/newName.git git remote -v (kiểm tra kỹ, nó
sẽ hiển thị url git mới) git push (làm bất cứ điều gì bạn muốn.)
```

# Chương 59: Gắn thẻ Git

Giống như hầu hết các Hệ thống kiểm soát phiên bản (VCS), Git có khả năng gắn thẻ các điểm cụ thể trong lịch sử là quan trọng. Thông thường, mọi người sử dụng chức năng này để đánh dấu các điểm phát hành (v1.0, v.v.).

## Mục 59.1: Liệt kê tất cả các thẻ có sẵn

Sử dụng lệnh `git tag` liệt kê tất cả các thẻ có sẵn:

```
thẻ $ git
<đầu ra theo sau>
v0.1
v1.3
```

Lưu ý: các thẻ được xuất theo thứ tự bảng chữ cái .

Người ta cũng có thể tìm kiếm các thẻ có sẵn :

```
$ git tag -l "v1.8.5*" <đầu
ra theo sau> v1.8.5
v1.8.5-rc0
v1.8.5-rc1
v1.8.5-rc2
v1.8.5-rc3
v1.8.5.1
v1.8.5.2
v1.8.5.3
v1.8.5.4
v1.8.5.5
```

## Phần 59.2: Tạo và đẩy (các) thẻ trong GIT

Tạo một thẻ:

- Để tạo thẻ trên nhánh hiện tại của bạn:

```
thẻ git < tên thẻ >
```

Điều này sẽ tạo một thẻ cục bộ với trạng thái hiện tại của nhánh bạn đang ở.

- Để tạo một thẻ với mã nhận dạng cam kết:

```
thẻ git tên thẻ nhận dạng cam kết
```

Điều này sẽ tạo một thẻ cục bộ với mã nhận dạng cam kết của nhánh bạn đang ở.

Đẩy một cam kết trong GIT:

- Đẩy một thẻ riêng lẻ:

tên thẻ nguồn gốc `git đầy`

- Đầy tất cả các thẻ cùng một lúc

nguồn gốc `git đầy --tags`

# Chương 60: Dọn dẹp kho lưu trữ cục bộ và từ xa của bạn

## Mục 60.1: Xóa các nhánh cục bộ đã bị xóa trên remote

Để theo dõi từ xa giữa các nhánh từ xa cục bộ và đã xóa, hãy sử dụng

```
git tìm nạp -p
```

sau đó bạn có thể sử dụng

```
nhánh git -vv
```

để xem nhánh nào không còn được theo dõi nữa.

Các nhánh không còn được theo dõi sẽ có dạng bên dưới, chứa 'gone'

```
chi nhánh 12345e6 [nguồn gốc/chi nhánh: đã biến mất] Đã sửa lỗi
```

sau đó bạn có thể sử dụng kết hợp các lệnh trên, tìm kiếm vị trí 'git nhánh -vv' trả về 'biến mất' rồi sử dụng '-d' để xóa các nhánh

```
git tìm nạp -p && git nhánh -vv | awk '/: gone]/{print $1}' | nhánh xargs git -d
```

# Chương 61: di-cây

So sánh nội dung và chế độ của các đốm màu được tìm thấy thông qua hai đôi tượng cây.

Phần 61.1: Xem các tệp đã thay đổi trong một cam kết cụ thể

```
git diff-tree --no-commit-id --name-only -r COMMIT_ID
```

## Mục 61.2: Cách sử dụng

```
git diff-tree [--stdin] [-m] [-c] [-cc] [-s] [-v] [--pretty] [-t] [-r] [--root] [<common-diff-options>] <tree-ish> [<tree-ish>] [<path>...]
```

### Mục 61.3: Các phương án chung

Lýa chọn	Giải trình
-z	dầu ra khác biệt với các dòng được kết thúc bằng NUL.
-P	định dạng bản vá dầu ra.
-u	tử dòng nghĩa với -p.
--patch-with-raw	xuất ra cả bản vá và định dạng khác biệt.
--stat	hiển thị diffstat thay vì vá.
--numstat	hiển thị khác biệt số thay vì bản vá.
--patch-with-stat	xuất ra một bản vá và thêm vào trước diffstat của nó.
--name-only	chỉ hiển thị tên của các tập tin đã thay đổi.
--tên-trạng thái	hiển thị tên và trạng thái của các tập tin đã thay đổi.
--chỉ mục đầy đủ	hiển thị tên đổi tương đầy đủ trên các dòng chỉ mục.
--abbrev=<n>	viết tắt tên đổi tương trong tiêu đề cây khác và thô.
-R	trao đổi cặp tập tin đầu vào.
-B	phát hiện viết lại hoàn chỉnh.
-M	phát hiện đổi tên.
-C	phát hiện bắn sao
--find-copies-harder	thử các tệp không thay đổi làm ứng cử viên để phát hiện bắn sao.
-l<n>	giới hạn các nỗ lực đổi tên theo đường dẫn.
-0	sắp xếp lại các khác biệt tùy theo .
-S	cặp tệp tìm thấy có một bên duy nhất chứa chuỗi.
--pickaxe-tất cả	hiển thị tất cả các tập tin là văn bản.
-một văn bản	coi tất cả các tập tin là văn bản.

# Tín dụng

Xin chân thành cảm ơn tất cả những người từ Tài liệu Stack Overflow đã giúp cung cấp nội dung này, nhiều thay đổi hơn có thể được gửi tới [web@petercv.com](mailto:web@petercv.com) để nội dung mới được xuất bản hoặc cập nhật

<a href="#">Aaron Critchley</a>	Chương 6
<a href="#">Aaron Skomra</a>	Chương 49
<a href="#">avrug</a>	Chương 26
<a href="#">Abdullah</a>	chương 2
<a href="#">Abhijeet Kasurde</a>	Chương 6
<a href="#">adarsh</a>	Chương 16
<a href="#">Adi Lester</a>	Chương 6
<a href="#">AER</a>	Chương 12, 25 và 29
<a href="#">AesSedai101</a>	Chương 4, 11 và 53
<a href="#">Ahmed Metwally</a>	chương 2
<a href="#">Ajedi32</a>	chương 11
<a href="#">Ala Eddine JEBALI</a>	Chương 1
<a href="#">Alan</a>	Chương 10
<a href="#">Alex Stuckey</a>	Chương 46
<a href="#">Chim Alexander</a>	Chương 12
<a href="#">Allan Burleson</a>	Chương 1
<a href="#">Alu</a>	Chương 50
<a href="#">hồ phách</a>	Chương 45
<a href="#">Amitay Stern</a>	Chương 1 và 10
<a href="#">anderas</a>	Chương 6 và 12
<a href="#">AndiDog</a>	Chương 16
<a href="#">andipla</a>	Chương 44
<a href="#">Andrea Romagnoli</a>	Chương 25
<a href="#">Andrew Sklyarevsky</a>	Chương 10
<a href="#">Andy Hayden</a>	Chương 1, 4, 7, 10 và 25
<a href="#">AnimiVulpis</a>	Chương 1, 5 và 14
<a href="#">ANOE</a>	Chương 24
<a href="#">Anthony Staunton</a>	chương 11
<a href="#">Một người</a>	Chương 10 và 13
<a href="#">họ Apidae</a>	Chương 6
<a href="#">Aratz</a>	chương 2
<a href="#">A-sáp</a>	Chương 4 và 26
<a href="#">Asenar</a>	Chương 11 và 13
<a href="#">Ates Goral</a>	Chương 32
<a href="#">Atul Khanduri</a>	Chương 17 và 59
<a href="#">Ban</a>	Chương 14
<a href="#">nhạc tồi</a>	Chương 10 và 16
<a href="#">Ben</a>	Chương 5
<a href="#">Nhà triết học ngờ ngần</a>	Chương 25
<a href="#">Bob Tuckerman</a>	Chương 14
<a href="#">Bogin</a>	Chương 1, 7, 31 và 36
<a href="#">Božo Stojković</a>	Chương 5
<a href="#">bpoiss</a>	Chương 5
<a href="#">Braiam</a>	Chương 5
<a href="#">brentonstrin</a>	Chương 7 và 8
<a href="#">Brett</a>	chương 2
<a href="#">Brian</a>	Chương 1 và 7

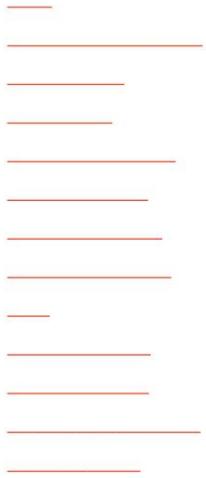
<a href="#">Brian Hinchey</a>	Chương 26
<a href="#">bstpierre</a>	chương 11
<a href="#">nụ</a>	Chương 17, 26 và 28
<a href="#">Bộ nhớ cache Staheli</a>	Chương 5, 10 và 13
<a href="#">Caleb Brinkman</a>	Chương 3 và 16
<a href="#">Charlie Egan</a>	Chương 6
<a href="#">Tần Hoàng</a>	Chương 20
<a href="#">Christian Maks</a>	Chương 24
<a href="#">Cody Guldner</a>	Chương 10 và 29
<a href="#">Collin M</a>	Chương 5
<a href="#">Truyện tranhSansMS</a>	Chương 9
<a href="#">Configure</a>	Chương 4, 22, 24, 36 và 44
<a href="#">cormacref</a>	Chương 10
<a href="#">Craig Brett</a>	Chương 1
<a href="#">Sáng tạo John co</a>	Chương 18
<a href="#">rùm người</a>	Chương 29
<a href="#">Dániel Kis</a>	Chương 45
<a href="#">dahlbyk</a>	Chương 20
<a href="#">dan</a>	Chương 14
<a href="#">Dan Hulme</a>	Chương 1
<a href="#">Daniel Käfer</a>	Chương 12, 14 và 50
<a href="#">Daniel Stradowski</a>	Chương 14
<a href="#">Dartmouth</a>	Chương 5, 25, 34, 43, 45, 47 và 48
<a href="#">David Ben Knoble</a>	Chương 38
<a href="#">davidcondrey</a>	Chương 2 và 10
<a href="#">Sâu</a>	Chương 10 và 26
<a href="#">Deepak Bansal</a>	Chương 14
<a href="#">Devesh Saini</a>	Chương 5
<a href="#">Thùng Dheeraj</a>	Chương 5
<a href="#">Dimitrios Mistriotis</a>	Chương 22
<a href="#">Đông Thắng</a>	Chương 49
<a href="#">dubek</a>	Chương 17
<a href="#">Duncan X Simpson</a>	Chương 14
<a href="#">e.doroskevic</a>	Chương 12 và 13
<a href="#">Ed Cottrell</a>	Chương 5
<a href="#">Eidolon</a>	Chương 24
<a href="#">Elizabeth</a>	Chương 45
<a href="#">enrico.bacis</a>	Chương 5
<a href="#">ericdwang</a>	Chương 10
<a href="#">eush77</a>	Chương 6, 11 và 16
<a href="#">eykanal</a>	Chương 1
<a href="#">Ezra miễn phí</a>	Chương 25
<a href="#">Fabio</a>	chương 2
<a href="#">Farhad Faghihi</a>	Chương 48
<a href="#">FeedTheWeb</a>	Chương 48
<a href="#">Chày</a>	Chương 2 và 24
<a href="#">mãi mãigenin</a>	Chương 3, 14 và 34
<a href="#">forresthopkinsa</a>	Chương 22
<a href="#">fracz</a>	Chương 5, 14 và 26
<a href="#">Fred Barclay nói</a>	Chương 1, 10 và 14
<a href="#">chuyện</a>	Chương 29
<a href="#">chức năng</a>	Chương 5
<a href="#">id người yêu</a>	Chương 49 và 61

<a href="#">gannesshkumar</a>	Chương 25
<a href="#">gavv</a>	Chương 14 và 35
<a href="#">George_Brighton,</a>	Chương 10
<a href="#">Georgebrock</a>	Chương 16
<a href="#">GüngPlusPlus</a>	Chương 26
<a href="#">Glenn_Smith</a>	Chương 35
<a href="#">gnis</a>	Chương 19
<a href="#">Greg_Bray</a>	Chương 50
<a href="#">Guillaume</a>	Chương 29
<a href="#">Guillaume_Pascal</a>	Chương 5 và 36
<a href="#">guleria</a>	chương 2
<a href="#">hardmooth</a>	Chương 22
<a href="#">heitortsergent</a>	Chương 3
<a href="#">Henrique_Barcelona</a>	Chương 1
<a href="#">Horen</a>	Chương 22
<a href="#">Hugo_Buff</a>	Chương 48
<a href="#">Hugo_Ferreira</a>	Chương 12
<a href="#">Igor_Ivancha</a>	Chương 10
<a href="#">Indregaard</a>	Chương 36
<a href="#">intboolstring</a>	Chương 2, 4, 5, 6, 9, 10, 12, 17 và 29
<a href="#">Isak_Combrinck</a>	Chương 57
<a href="#">JF</a>	Chương 9
<a href="#">Jack_Ryan</a>	Chương 6
<a href="#">JakeD</a>	Chương 6
<a href="#">Jakub_Narębski</a>	Chương 4, 5, 6, 26 và 43
<a href="#">james_lor</a>	Chương 14
<a href="#">James_Taylor</a>	Chương 10
<a href="#">janos</a>	Chương 1, 2 và 10
<a href="#">JaredE</a>	Chương 26
<a href="#">Jason</a>	Chương 14
<a href="#">Jav_Rock</a>	Chương 1, 45 và 49
<a href="#">Jeff_Puckett</a>	Chương 27
<a href="#">_____</a>	Chương 1, 3, 6 và 26
<a href="#">_____</a>	Chương 5
<a href="#">_____</a>	Chương 4
<a href="#">jeffdill2_Jens</a>	Chương 5
<a href="#">jkdev</a>	Chương 14
<a href="#">_____</a>	Chương 10
<a href="#">_____</a>	Chương 5
<a href="#">_____</a>	Chương 14
<a href="#">joaquinlpereyra</a>	Chương 1
<a href="#">Joel_Cornett</a>	Chương 10
<a href="#">joeytwiddle_JonasCz</a>	Chương 1 và 14
<a href="#">Jonathan_Jonathan</a>	Chương 6 và 12
<a href="#">Lam</a>	Chương 5
<a href="#">Jordan</a>	Chương 16
<a href="#">Knott</a>	chương 2
<a href="#">Joseph</a>	Chương 11 và 12
<a href="#">_____</a>	Chương 13 và 52
<a href="#">Dasenbrock</a>	Chương 3, 18 và 26
<a href="#">Joseph</a>	Chương 39
<a href="#">K._Strauss_joshng_jpkrohling</a>	Chương 5
<a href="#">tbandes Julian Julie David jwd630 Ka čer</a>	Chương 5
<a href="#">Kageetai</a>	Chương 1

<u>Kalpit</u>	Chương 3 và 45
<u>Kamiccolo</u>	chương 2
<u>Kapep</u>	Chương 5
<u>Kara</u>	Chương 26
<u>Karan Desai</u>	Chương 28
<u>cuộc đua ngựa</u>	Chương 14 và 25
<u>KartikKannapur</u>	Chương 1, 10, 25 và 48
<u>Kay V</u>	Chương 1 và 4
<u>Kelum Senanayake</u>	Chương 56
<u>Keyur Ramoliya</u>	Chương 54
<u>khanmizan</u>	Chương 6 và 14
<u>kirmann</u>	Chương 14
<u>kisanme</u>	Chương 14 và 17
<u>Kissaki</u>	Chương 23 và 31
<u>nút thắt</u>	Chương 5
<u>kofemann</u>	Chương 49
<u>Koraktor</u>	Chương 26
<u>quỳ lạy</u>	Chương 9
<u>KraigH</u>	chương 2
<u>TráiPhải92</u>	Chương 5
<u>LeGEC</u>	Chương 2 và 12
<u>Liam Ferris</u>	Chương 8
<u>Libin Varghese</u>	Chương 12
<u>Liju Thomas</u>	Chương 47
<u>Liyan Chang</u>	Chương 13
<u>Lochlan</u>	Chương 17
<u>nha triết học lạc lối</u>	Chương 24
<u>Luca Putzu</u>	Chương 12
<u>lucash</u>	Chương 12
<u>Maccard Mário</u>	Chương 29
<u>Meyrelles</u>	Chương 1
<u>Macktán công</u>	Chương 5
<u>điên</u>	chương 11
<u>Majid</u>	Chương 6, 10, 12, 14, 22 và 26
<u>manasouza</u>	Chương 2 và 26
<u>Manishh</u>	Chương 55
<u>Mario</u>	Chương 21
<u>Martin</u>	Chương 14
<u>Martin Pecka</u>	Chương 5
<u>Marvin</u>	Chương 5 và 29
<u>Matas Vaitkevicius</u>	Chương 52
<u>Mateusz Piotrowski</u>	Chương 16
<u>Matt Clark</u>	Chương 2 và 10
<u>Matt S</u>	Chương 29
<u>Matthew Hallatt</u>	Chương 2, 7, 10, 42 và 46
<u>MayeulC</u>	Chương 5, 10, 14, 19, 23 và 29
<u>MByD</u>	Chương 3
<u>Micah Smith</u>	Chương 10
<u>Micha Wiedenmann</u>	Chương 53
<u>Michael Mrozek</u>	Chương 12
<u>Michael Plotke</u>	Chương 21
<u>Mitch Talmadge</u>	Chương 5 và 14
<u>mkasberg</u>	Chương 15

<a href="#">mpromonet</a>	Chương 2, 9, 17 và 23
<a href="#">MrTux</a>	Chương 41
<a href="#">mwarsco</a>	Chương 24
<a href="#">mystarrocks</a>	Chương 3
<a href="#">n0shadow</a>	Chương 19
<a href="#">Narayan Acharya</a>	Chương 5
<a href="#">Nathan Arthur</a>	Chương 4
<a href="#">Nathaniel Ford</a>	Chương 6 và 7
<a href="#">Nemanja Boric</a>	Chương 12
<a href="#">Nemanja Trifunovic</a>	Chương 50
<a href="#">nepda</a>	Chương 14
<a href="#">Neui</a>	Chương 1 và 5
<a href="#">nighthawk454</a>	Chương 30 và 42
<a href="#">Nithin K Anil</a>	Chương 7
<a href="#">Nô-ê</a>	Chương 2, 8 và 14
<a href="#">Noushad PP</a>	Chương 14
<a href="#">Nuri Tasdemir</a>	Chương 5
<a href="#">nus</a>	Chương 11 và 38
<a href="#">ob1</a>	Chương 1
<a href="#">Thánh vịnh yêu tinh33</a>	Chương 6
<a href="#">cây trúc đào</a>	chương 2
<a href="#">olegtaranenko</a>	Chương 14
<a href="#">orkoden</a>	Chương 6 và 40
<a href="#">Ortomala Lokni</a>	Chương 6, 12, 13 và 26
<a href="#">Ozair Kafray</a>	Chương 14
<a href="#">PJMeisch</a>	Chương 28
<a href="#">Nhịp độ</a>	Chương 7
<a href="#">PaladiN</a>	Chương 5, 9 và 14
<a href="#">Patrick</a>	Chương 26
<a href="#">pcm</a>	Chương 29
<a href="#">Pedro Pinheiro</a>	Chương 2 và 50
<a href="#">chim cánh cut</a>	Chương 6, 8 và 11
<a href="#">Peter Amidon</a>	Chương 51
<a href="#">Peter Mitrano</a>	Chương 12, 25 và 26
<a href="#">Trác quangStereo</a>	Chương 28
<a href="#">pkowalczyk</a>	Chương 25
<a href="#">pktangyue</a>	Chương 5
<a href="#">Nhóm</a>	Chương 1 và 10
<a href="#">pogosama</a>	Chương 29
<a href="#">choc</a>	Chương 5
<a href="#">Priyanshu Shekhar</a>	Chương 13, 14, 19 và 42
<a href="#">pylang</a>	Chương 5 và 12
<a href="#">Raghav</a>	Chương 3
<a href="#">Ralf Rafael Frix</a>	Chương 3, 14, 19 và 26
<a href="#">Mã màu domàu xanh lá cây</a>	Chương 17
<a href="#">Rhys0</a>	Chương 5
<a href="#">Ricardo tình yêu</a>	Chương 33
<a href="#">Richard</a>	Chương 12
<a href="#">Richard Dally</a>	Chương 4
<a href="#">Richard Hamilton</a>	Chương 14
<a href="#">Dụng zrom</a>	Chương 5, 7, 10, 25 và 36
<a href="#">riyadhalnur</a>	chương 11
<a href="#">Roald Nefs</a>	Chương 1

<u>Robin</u>	Chương 14
<u>rokonoid</u>	Chương 5
<u>ronnyfm</u>	Chương 1
<u>Salah Eddine Lahniche</u>	Chương 3
<u>saml</u>	Chương 3
<u>Sardathrion</u>	Chương 22
<u>Sascha</u>	Chương 5
<u>Sói Sascha</u>	Chương 5
<u>SashaZd</u>	Chương 48
<u>Sazzad Hissain Khan</u>	Chương 1
<u>Scott Weldon</u>	Chương 3, 5, 22, 23, 41 và 51
<u>Sebastianb</u>	Chương 5 và 26
<u>SeeuD1</u>	Chương 5
<u>thợ đóng già</u>	Chương 46
<u>Shog9</u>	Chương 23
<u>Simone Carletti</u>	Chương 14 và 41
<u>sjas</u>	Chương 5
<u>SommerEngineering</u>	Chương 10
<u>sonali</u>	Chương 45
<u>Sonny Kim</u>	Chương 10
<u>tăng đột biến</u>	Chương 5
<u>Đá</u>	Chương 23
<u>laqargo</u>	Chương 18
<u>SurDin</u>	Chương 6
<u>Sam Cao</u>	Chương 16
<u>võ văn bản</u>	Chương 7
<u>Tiếng Thamilan</u>	Chương 23
<u>cảm ơn</u>	chương 11
<u>the12</u>	Chương 14
<u>TheDarkKnight</u>	Chương 36 và 59
<u>theJollySin</u>	Chương 5
<u>Thomas Crowley</u>	Chương 60
<u>tinlyx</u>	Chương 9
<u>Toby</u>	Chương 5 và 20
<u>Toby Allen</u>	chương 2
<u>Tom Gijselinck</u>	Chương 5
<u>Tom Hale</u>	chương 11
<u>Tomás Cañibano</u>	Chương 26 và 29
<u>Tomasz Bąk</u>	Chương 5
<u>Travis</u>	Chương 12
<u>Tyler virus</u>	Chương 1
<u>sốt phát ban</u>	Chương 1
<u>Hoàn tác</u>	Chương 8, 9, 10 và 25
<u>Úy Vi.</u>	Chương 14
<u>Victor Schröder</u>	Chương 5
<u>Vivian George</u>	Chương 5, 12 và 44
<u>Vlad</u>	Chương 3
<u>Vladimir F</u>	Chương 14
<u>Vogel612</u>	Chương 10
<u>VonC</u>	Chương 8
<u>Quạt mù tạt</u>	Chương 1, 3, 5, 9, 12 và 13
<u>Wilfred Hughes</u>	Chương 12
	Chương 5



## Bạn cũng có thể thích

