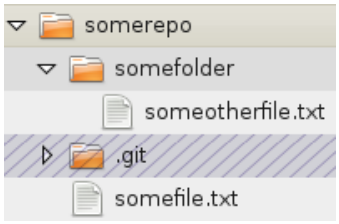
Tạo repository mới

Lệnh *git init* tạo một repository loại git tại thư mục hiện tại:

```
$ git init
```

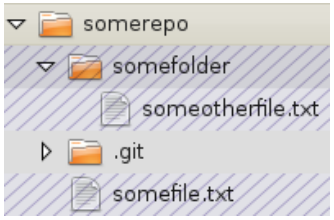
Khác với các hệ quản lý phiên bản khác, git cho phép đặt repo hoàn toàn tại máy tính local. Nội dung của toàn bộ *repository* được lưu tại thư mục *.git* tại thư mục vừa chạy init:



Tất nhiên, git cũng có thể làm việc với các repo nằm trên mạng hoặc repo tổng, nhưng việc đó không bắt buộc. Ta sẽ nói đến chủ đề làm việc với repo mạng ở các mục sau. Hiện giờ, ta có thể dùng git để tạo repo tại máy địa phương tùy ý.

The working tree

Các file của bạn trong thư mục repo được gọi là *working tree*:



The staging index

Git lưu trữ nội bộ một thứ gọi là *index*, nó là ảnh chụp các file trong project của bạn. Sau khi bạn tạo một repo, nó là repo rỗng và index của git rỗng (kể cả nếu trong thư mục đã có sẵn một số file và thư mục). Bạn phải tự tay ghi nhận (stage) các file từ *working tree* của mình vào *index* bằng lệnh *git add*:

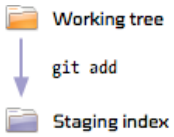
```
> git add somefile.txt
```

git add chạy đệ quy, bạn có thể add cả thư mục theo cách này:

```
> git add somefolder
```

Không chỉ cần add đối với những file chưa từng vào index, công việc cũng như vậy nếu bạn sửa một file tại working tree – bạn phải add thay đổi đó vào index bằng lệnh *git add*:

```
> edit somefile.txt
> git add somefile.txt
```



Cần nhớ rằng index là ảnh chụp của toàn bộ các file trong project chứ không chỉ là một danh sách các file đã bị thay đổi.

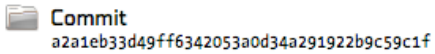
Commit

git commit lấy nội dung của index và tạo một bản *commit* mới

```
> git commit -m "the 1st commit"
```

Commit là một hoạt động hoàn toàn địa phương, nó không liên quan đến việc gửi cái gì tới một server nào trên mạng. Nó chỉ lấy nội dung của index và lưu lại một ảnh chụp của project như được ghi trong index:

Tương tự với index, một *commit* là một ảnh chụp toàn bộ các file trong project. Mỗi commit được gắn một nhãn xác định duy nhất commit đó (nhãn này được tính từ kết quả SHA-1 hash của nội dung được chụp lại):



Do tính chất xác định duy nhất của nhãn commit, nó cho phép ta lấy được chính xác những gì chúng ta có trong project tại thời điểm thực hiện commit đó. Nhãn commit tuy dài nhưng ta chỉ cần dùng 7 chữ số đầu tiên là đủ để xác định commit khi cần chỉ định nó trong lệnh (xem ví dụ tại các mục sau).

Xem xem cái gì sẽ được commit

git status cho bạn biết *working tree* hiện khác biệt như thế nào so với *index* và *index* hiện khác gì so với *commit* lần gần nhất:

```
> git status
```

Đầu tiên, bạn sẽ thấy những thay đổi đã được add vào *index*. Danh sách này là những gì sẽ nằm trong lần commit tiếp theo:

```
# Changes to be committed:
#
#       modified:   changed_file_added.txt
```

Tiếp theo là những thay đổi đã được thực hiện với working tree, nhưng chưa được add vào *index*:

```
# Changed but not updated:
#
#       modified:   changed_file.txt
#
# Untracked files:
#
#       newfile.txt
```

Shortcut: Cách add các thay đổi ngay khi thực hiện commit

Khi thực hiện commit, bạn có thể để git tự add các file đã bị sửa đổi bằng cách sử dụng tùy chọn *-a*:

```
> git commit -a -m "commit message"
```

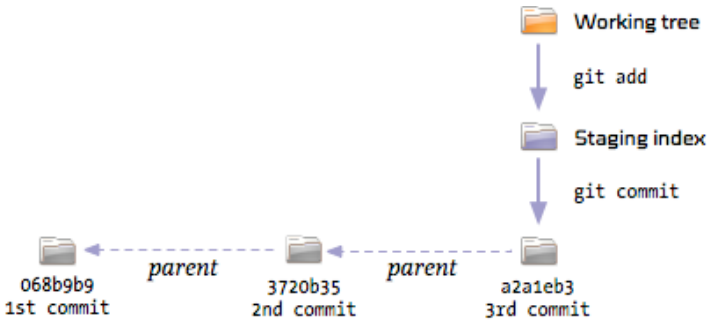
Lệnh trên sẽ add tất cả các sửa đổi (ngoại trừ các file mới tạo) vào index trước khi commit.

Lịch sử commit

Quy trình thực hiện việc soạn các file trong repo git trông như sau:

- Sửa nội dung file nằm tại *working tree*.
- Stage sửa đổi đã thực hiện vào *index* bằng lệnh *git add*.
- Commit nội dung *index* bằng lệnh *git commit*.

Khi bạn thực hiện quy trình trên nhiều lần, mỗi lần bạn sẽ tạo một *commit* mới, nó trỏ về commit trước nó:



Đó là cách git lưu lại lịch sử của project. Nó lưu lại các ảnh chụp của các file trong project dưới dạng các commit. Mỗi commit này trỏ ngược lại commit tiền thân của mình.

Bạn có thể xem lịch sử này bằng lệnh *git log*:

```
> git log

commit 068b9b9...
Author: Bob <bob@example.com>
Date:   Wed Jun 17 17:21:16 2009 +0200

    the 3rd commit

commit 3720b35...
Author: Bob <bob@example.com>
Date:   Wed Jun 17 17:21:10 2009 +0200

    the 2nd commit

commit a2a1eb3...
Author: Bob <bob@example.com>
Date:   Wed Jun 17 17:21:10 2009 +0200

    the 1st commit
```

git cũng cung cấp phiên bản ngắn gọn hơn như sau:

```
> git log --pretty=oneline --abbrev-commit

068b9b9 the 3rd commit
3720b35 the 2nd commit
a2a1eb3 the 1st commit
```

Git configuration: user settings

Nếu nhìn lịch sử commit đầy đủ, bạn sẽ thấy nó có chứa thông tin về tác giả. Git lấy thông tin này ở đâu khi nó tạo commit. Đầu tiên là nó cố đoán từ username và hostname của bạn. Bạn cũng có thể tự tay cấu hình cho project hiện tại như sau:

```
> git config user.name "Bob"
> git config user.email "bob@example.com"
```

Bạn cũng có thể cấu hình một lần cho tất cả các git project của bạn

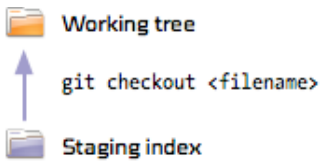
```
> git config --global user.name "Bob"
> git config --global user.email "bob@example.com"
```

Hủy bỏ các sửa đổi

Nếu bạn đã thực hiện một số sửa đổi trong project nhưng muốn quay lại tình trạng cũ. Cách làm như sau, tùy theo tình huống.

Nếu bạn chưa add chúng vào *index*, bạn có thể khôi phục để file/thư mục đó tại working tree quay trở loại nội dung đã lưu trong index bằng lệnh *git checkout <filename>*:

```
> git checkout somefile.txt
```



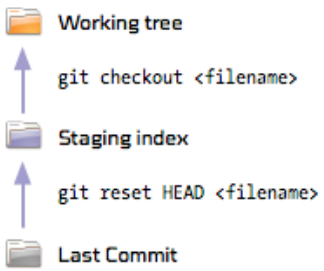
Nếu bạn đã add sửa đổi đó vào *index*, bạn có thể khôi phục *index* về tình trạng của lần commit gần nhất bằng lệnh *git reset*:

```
> git reset HEAD somefile.txt
```

Hoặc khôi phục toàn bộ *index*:

```
> git reset HEAD
```

HEAD luôn luôn chỉ tới lần commit gần nhất đã làm. Dùng lệnh trên, *index* được khôi phụ từ lần commit trước, tiếp theo bạn có thể dùng lệnh *git checkout* để khôi phục cả working tree:



Reverting commits – hủy commit cũ

Nếu bạn đã commit sửa đổi, bạn có thể dùng lệnh *git revert <commit>* để undo commit đó.

```
git revert 068b9b9
```

Lệnh trên sẽ tạo một commit thứ hai undo thay đổi của commit nói trên:

```
8b54ea7 Revert "the 3rd commit"
068b9b9 the 3rd commit
3720b35 the 2nd commit
a2a1eb3 the 1st commit
```

Ta cũng có thể undo các commit cũ hơn:

```
> git revert 3720b35
> git log --pretty=oneline --abbrev-commit

ab621c7 Revert "the 2nd commit"
068b9b9 the 3rd commit
3720b35 the 2nd commit
a2a1eb3 the 1st commit
```

Tuy nhiên, việc undo một commit cũ hơn có thể gây xung đột (conflict) nếu các commit mới hơn nó cũng sửa trùng các nội dung đó. Cách giải quyết các conflict sẽ được nói đến sau.

Xem diff giữa các commit

Để xem khác biệt giữa một commit với tiền thân của nó, ta dùng lệnh *git show <commit>*:

```
> git show 3720b35
```

Để so sánh hai commit cụ thể với nhau, ta dùng lệnh *git diff <commit_from>..<commit_to>*:

```
> git diff a2a1eb3..068b9b9
```

Để xem diff của toàn bộ lịch sử, dùng lệnh *git log -p*

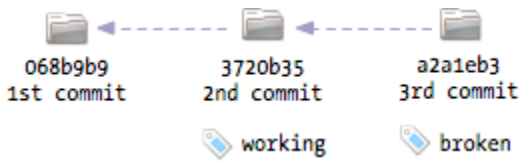
```
> git log -p
```

Tagging commits – gắn tag cho commit

Lệnh *git tag <name> <commit>* gắn một tag cho một commit. Nếu bỏ qua phần *<commit>*, tag sẽ được gắn cho commit gần nhất:

```
> git tag working 3720b35
> git tag broken
```

Tag chỉ là một cái nhãn mà có thể dùng để gọi tên commit được gắn nhãn:



Có thể dùng tag ở bất cứ ngữ cảnh nào mà ta có thể dùng nhãn hash, chẳng hạn trong lệnh *git diff*:

```
> git diff working..broken
```

Tag là công cụ hoàn hảo để đặt tên cho các mốc cụ thể trong lịch sử project. Người ta thường dùng tag dạng version number khi commit các bản release của project – bằng cách này ta có thể dễ dàng tìm thấy các commit quan trọng:

```
> git tag v1.0.3
```

Branches - Nhánh

Như vậy ta đã học cách tạo các commit và làm việc với chúng. Đến đây, lịch sử project của ta mới có dạng tuyến tính.

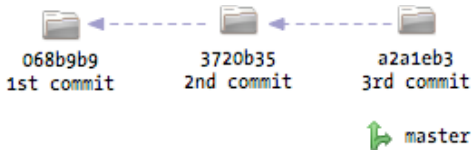
Giả sử ta muốn phát triển một tính năng mới cho project. Đây là một việc lớn, và ta muốn tách các thay đổi này ra khỏi các thay đổi khác. Ta có thể dùng branch cho việc này.

Từ đầu đến giờ, ta đã làm việc với một branch có tên là *master*. Branch này được tạo tự động khi ta tạo repo. Bạn có thể xem danh sách các branch của repo bằng lệnh *git branch*:

```
> git branch

* master
```

Như vậy là đã có một branch (nhánh) có tên *master*. Dấu * kí hiệu đây là branch mà hiện tại chúng ta đang làm việc. Một branch gần giống như một *tag*. Một branch bao giờ cùng chỉ tới một commit nào đó, trong ngữ cảnh hiện tại , nó đang chỉ tới commit mới nhất mà ta đã thực hiện. Ta có thể kiểm tra điều đó bằng lệnh:

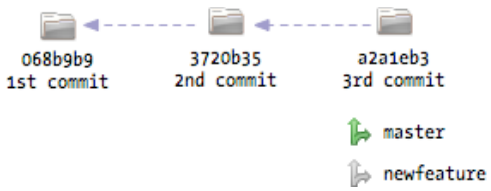
```
> git branch -v

* master 068b9b9 the 3rd commit
```



Để theo dõi các thay đổi tại một branch khác, ta phải tạo một branch mới bằng lệnh *git branch <name> <commit>*. Commit đã chọn sẽ là điểm bắt đầu của nhánh mới – nếu ta bỏ qua phần <commit>, git sẽ chọn commit gần nhất:

```
> git branch newfeature
```

Lệnh trên sẽ tạo một branch mới có tên *newfeature* dựa trên commit mới nhất của branch *master*:



Tại mỗi thời điểm, ta chỉ có một branch hiện hành và đang làm việc với branch đó. Nếu gọi lệnh *git branch*, ta sẽ nhìn thấy branch mới với tên *newfeature* đã được tạo, nhưng branch hiện hành vẫn là *master*:

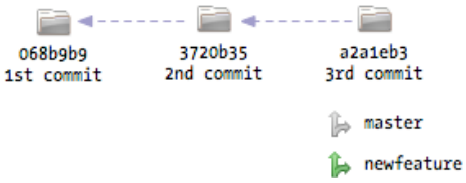
```
> git branch

* master
  newfeature
```

Ta có thể chuyển giữa các branch bằng lệnh *git checkout <branchname>*. Đây cũng chính là lệnh ta đã dùng để lấy file từ staging index đổ vào working tree. Ý nghĩa của *git checkout* tùy theo tham số của lệnh đó.

```
> git checkout newfeature
```

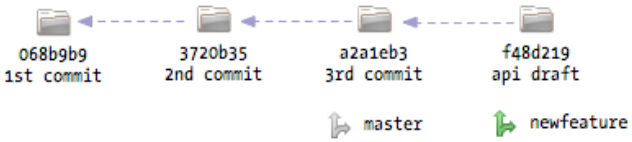
Bây giờ branch hiện hành trong repo là *newfeature*:

```
> git branch

  master
* newfeature
```



Ta có thể bắt đầu làm việc với branch mới bằng những lệnh đã học như *git add*, *git commit*, v.v.. Hãy xem chuyện gì xảy ra khi ta tạo một commit mới tại branch hiện tại:

```
> edit somefile.txt
> git commit -a -m "api draft"
```

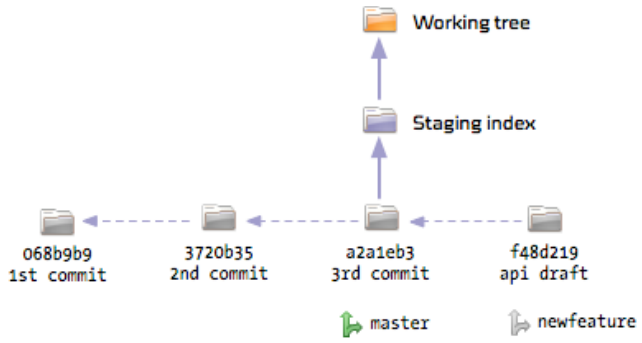


Commit được add vào repo và branch *newfeature* bây giờ đang chỉ tới commit mới. Về mặt khái niệm, một branch có thể được xem như một con trỏ tới một commit nào đó. Mỗi khi bạn commit gì đó, "con trỏ" này của branch hiện hành sẽ được chuyển tới chỉ vào commit mới. Đó là tất cả những gì cần đến để phân biệt các branch.

Để tiếp tục ví dụ, giả sử tạm thời ta đã làm việc xong với branch newfeature và muốn quay lại làm việc tại branch master. Ta quay lại branch *master* bằng lệnh sau:

```
> git checkout master
```

Lệnh trên sẽ chuyển branch hiện hành về lại *master*. Nó cũng sẽ reset *index* và *working tree* của ta về nội dung của lần commit cuối cùng tại master:

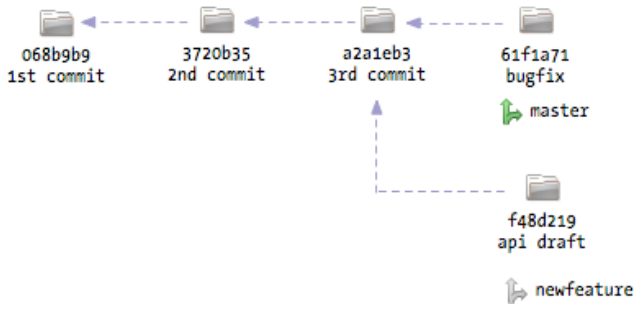


Ta sẽ thấy project trở về trạng thái giống hệt như thời điểm ta tách branch mới. Sẽ không nhìn các thay đổi đã làm cho branch *newfeature*.

Vậy chuyện gì xảy ra nếu bây giờ ta thêm một số sửa đổi và commit chúng?

```
> edit somefile.txt
> git commit -a -m "bugfix"
```

Cũng giống như trước: Một commit mới sẽ được tạo từ commit gần nhất cho *master* và *master* sẽ chỉ tới commit mới:



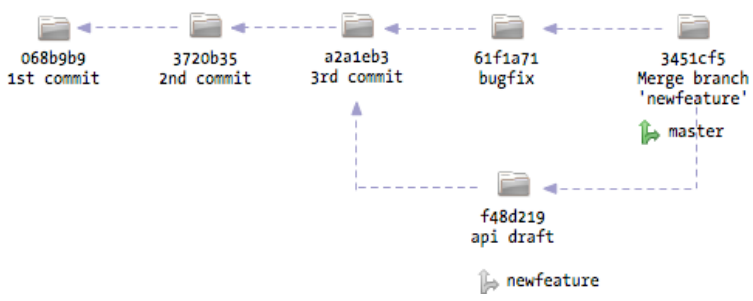
Branch là cơ chế nhẹ và nhanh của git, ta có thể tạo nhiều branch tùy ý. Một số lập trình viên thích làm việc với một branch *stable* và một branch *unstable*, trong khi người khác lại tạo branch cho mỗi tính năng mới mà họ tạo – tùy bạn chọn cách của mình.

Merging – hợp nhất các branch

Giả sử ta đã hoàn thành tính năng mới và muốn đưa nó vào branch *master*. Ta dùng lệnh *git merge* trong khi master đang là branch hiện hành:

```
> git merge newfeature
```

Nếu không có conflict, git sẽ tạo một commit mới chứa các thay đổi đã thực hiện ở cả hai branch:



Git thường rất thông minh khi merge. Tuy nhiên, nếu cùng một nội dung đã được sửa ở cả hai branch, ta sẽ có tình trạng conflict:

```
CONFLICT (content): Merge conflict in somefile.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Ta phải tự tay giải quyết conflict này. Nếu bạn xem nội dung file bị conflict, bạn sẽ thấy các dòng có conflict bị đánh dấu. Tại đầu file sẽ có phiên bản của branch hiện hành, bên dưới là phiên bản của branch được trộn vào:

```
<<<<<<< HEAD:somefile.txt
this change was done in master
=======
this change was done in newfeature
>>>>>>> newfeature:somefile.txt
```

Bạn phải tự tay giải quyết conflict này và xóa bỏ các marker đánh dấu. Sau đó, bạn add file vào *index* và commit kết quả:

```
> git add somefile.txt
> git commit
```

Sau khi đã merge xong các branch, bạn có thể xóa branch newfeature đi nếu bạn không cần đến nó nữa:

```
> git branch -d newfeature
```

Bạn cũng có thể tiếp tục làm việc với branch đó rồi sau đó lại merge lại vào master. Git sẽ đủ thông minh để biết phần nào của branch đã được merge và sẽ chỉ merge các thay đổi mới.