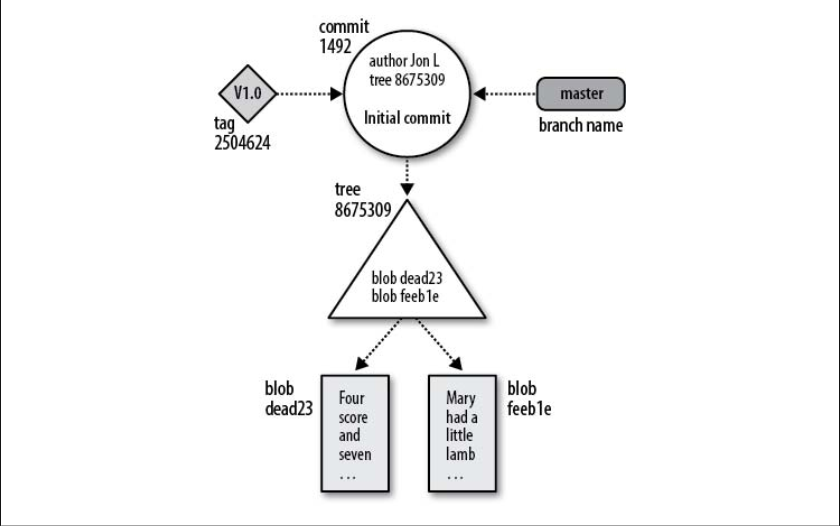
* 2 cách để tạo repository
  + Tạo
  + Clone
* Git đặt thông tin các revision trong thư mục .git
* Git phân biệt 2 hoạt động add và commit để tránh volatility
* Cần config để git biết mình là ai: git config
* Git có những biến local như: GIT\_AUTHOR\_NAME, GIT\_AUTHOR\_EMAIL, nếu set những biến này, nó sẽ ghi đè những lện config
* 1 file đã được git add vào index thì lần sau commit có thể ko cần add nó, vì nó đã được add
* Alias: thiết lập từ khóa cho các câu lệnh git dài thường dùng

Basic Concepts

* Repository:
  + Git duy trì 2 cấu trúc dữ liệu cơ bản: object store và index, được lưu ở .git
* Object Types
  + Object store là trái tim của repository
  + Chứa data gốc, log, author, … để khôi phục bất cứ phiên bản, nhánh nào
  + Object store chứa 4 loại object: blobs, trees, commits, tags
  + Blobs: binary large object, đại diện cho mỗi version của 1 file
    - Là 1 thứ mờ ảo
  + Trees: đại diện cho cấp độ của thông tin thư mục, tương tác với tree khác, xây dựng cấu trúc thư mục, file
  + Commits: data về thay đổi trong repo, trỏ tới tree có trạng thái của repo tại thời điểm commit đc thực hiện, mỗi commit có 1 commit parent, trừ commit root. Mỗi commit có thể tham chiếu tới commit khác, ko chỉ parent của nó
  + Tags: gắn 1 tên cho 1 object, thường là commit
  + Git nén những obj trên thành pack file, trong object store
* Index
  + File tạm thời, mô tả cấu trúc thư mục của repo ở 1 số thời điểm
  + Lưu những thay đổi trước khi commit
  + Git cho phép thay đổi index qua những bước đã đc định nghĩa
* Content-Addressable Names
  + Object store giống như 1 content-addressable storage system
  + Mỗi obj có 1 tên duy nhất đc tạo ra bởi SHA1, commitID là 1 SHA1
* Tracks Content
  + Git là hệ thống giám sát nội dung
  + Object store hoạt động dựa trên tính toán hasj của nôi dung của các object, ko phải dựa vào tên của file hay thư mục. Git giám sát nội dung, ko phải file
  + Git tính toán nôi dung file bằng SHA1, nên nếu có 2 file có cùng nội dung, chúng chỉ đc lưu bằng blob obj trong object store
  + Git lưu lại mỗi phiên bản của file bằng hash, chứ ko phải lưu sự khác nhau
* Pathname vs Content
  + Git lưu lại name của file, thư mục để có thể khôi phục nội dung của repo, chứ ko quan tâm đến name?
  + Cấu trúc dữ liệu của git ko giống như cấu trúc thư mục, file của user
* Pack Files
  + Git có lưu sự thay đổi, là sự thay đổi của nội dung, chứ ko phải sự thay đổi của 1 file. Do đó nó ko quan tâm sự thay đổi đó là của cùng 1 file hay ko
  + Nếu thấy có 2 nội dung trong object store, hơi giống nhau, nó sẽ tính sự khác biệt và lưu sự khác biệt đó, chứ ko quan tâm 2 nội dung đó có cùng 1 file hay ko
* Object Store picture



* Khi git add, git tạo các object blob cho các nội dung của các file,chưa tạo ra object tree mà thay đổi index với pathname và thông tin blob tương ứng
* Commit đầu tiên sẽ chứa tất cả các commit của các tree, blob
* Sử dụng hash SHA1 giúp so sánh 2 object hiệu quả cho dù 2 object nội dung lớn và phức tạp
* Tree hierarchies
* Commit
  + Chứa các tree, blob, author, date
  + Khi tạo 1 commit, có thể cho nó 1 hoặc nhiều parents
* Tags
  + 2 loại: lightweight, annotated
  + Lightweight: trỏ tới 1 commit obj, là private của repo. Ko tạo ra obj trong object store
  + Annotated: tạo ra object, bao gồm message, GnuPG theo RFC4880
  + 1 số lệnh chỉ hoạt động với loại annotated
  + Tag trỏ tới commit, cũng có nghĩa trỏ tới tất cả các thành phần con

**File Management and the Index**

* Index là tập hợp các chỉnh sửa có thể sẽ đc lưu lại
* Khi commit, git sẽ kiểm tra index, ko phải thư mục làm việc

**File Classifications**

* Git phân loại file thành 3 nhóm: tracked, ignored và untracked
* Tracked: đã ở trong repo hoặc trong index
* Ignored: cần khai báo, gồm những file note, tạm thời, … Git duy trì 1 danh sách các file ignored
* Untracked: những file còn lại
* Git add: chuyển 1 file untracked sang tracked, nếu dùng với tên thư mục, thì sẽ add toàn bộ thư mục đó. Khi add, git sẽ tạo SHA1 cho file đó
* Git rm: xóa file khỏi repo và thư mục làm việc. Git sẽ xóa file ra khỏi index, hoặc index và thư mục làm việc, chứ ko chỉ xóa khỏi thư mục làm việc. Mặc dù bị xóa nhưng nó vẫn còn trong lịch sử của repo, object store và có thể đc khôi phục lại

Trc khi xóa, git sẽ kiểm tra coi cái version cuối file đó trong thư mục làm việc có trùng với version trong repo hay ko

* **.gitignore file**
* .gitignore có thể nằm ở các thư mục trong repo, và ta có thể đưa tên file muốn bỏ qua vào file này, hoặc vào .gitignore ở thư mục ngoài cùng
* File này có thể chứa pattern
* File này sẽ đc clone, nên có thể ảnh hưởng đến các repo #, nên có thể dùng .git/info/exclude file, giống như .gitignore file

**Commits**

* Khi commit, Git sẽ tạo snapshot từ index và đặt vào object store
* Git sẽ so sánh trạng thái hiện tại với snapshot trc đó, và tạo các blob mới cho các file thay đổi, tạo tree mới cho các thư mục thay đổi
* Các snapshot sẽ nối liên tục với nhau, trỏ tới cái trc đó
* Thao tác merge tạo ra commit tự động
* Mỗi commit là 1 atomic changeset
* Sự thay đổi changset giữa 2 snapshot là sự chuyển đổi từ 1 tree sang tree khác
* SHA1 ID: trỏ tới commit trực tiếp, là tên tuyệt đối
* HEAD: lần commit gần nhất
* Ref, symref là SHA1 trỏ tới object, thường là commit object. Tên branch, tag đều là ref
* 1 branch local tên dev có tên đầy đủ là refs/heads/dev. Remote branch là refs/remotes/, tag là refs/tags. Do đó, 1 tag v1.2.3 thực chất là refs/tags/v1.2.3
* 1 số symref:
  + HEAD: commit gần nhất của branch hiện tại
  + ORIG\_HEAD: commit gần nhất thứ 2
  + …
* Relative Commit Names:
  + Commit có nhiều commit parent, thì đó là kết quả của merge
  + Cho commit C có nhiều parent, C^1, C^2, … là các commit parent, C~1 là parent đầu, C~2 là grandparent đầu, C~3 là great-grandparent đầu, …
  + Viết tắt: C^ = C^1, C~ = C~1, C^^ = C^1^1 = C~2, …
* Commit History
* Commit có thể đi đến từ commit Y là commit có đường đi tới Y
  + Git log Y: log ra tất cả commit có thể đi đến từ Y, bao gồm Y
  + Git log ^X: log ra tất cả commit ko đi đc đến từ X, ko có X
  + ..end = HEAD..end; start.. = start..HEAD
* Git bisect: đầu tiên, mình chỉ ra version nào là bad(bug), version nào là good, git bisect sẽ checkout về version ở giữa, mình xác định version đó good, bad, từ đó, git bisect xác định khoảng commit bad, và tiếp tục checkout ở giữa.

Git bisect dùng để đi qua các commit và xác định commit nào bad

* Git blame: xem ai, commit nào đã chỉnh sửa 1 file
* Git log –Sstring: tìm sự thay đổi của 1 string(thêm, xóa) trong 1 file qua các commit

**Branches**