A Practical Guide to Using Git (From a Kernel Maintainer's Perspective)

James Bottomley

Novell

13 July 2009

Introduction

- Talk based on “unconference” presentation at FreedomHEC in Los Angeles
- Git is huge, so will not cover *all* of git, so ask if you want to know something
- Git has two modes of use:
  - Historical: Tree never rebases (this is the way Linus uses it), and
  - Developmental: Use is to manage uncommitted patch sets (a bit like quilt)

Brief History of Git

- Source control began in Linux as the need to manage patch inputs efficiently
  - Before, Linus revewed every patch
  - After, only subsystem maintainers review patches that go via subsystem trees
  - => scaling.
- After SCO it continued as the need to track contributions
- Initial tool for this was Bitkeeper.

Bitkeeper

- Fully distributed nicely scaleable non master based source tree management system
- Initial use for Linux was early in 2002
- Final rupture was in 2005
- Bitkeeper worked extremely well for those three years in spite of the complaints about its being proprietary.

Developer Certificate of Origin

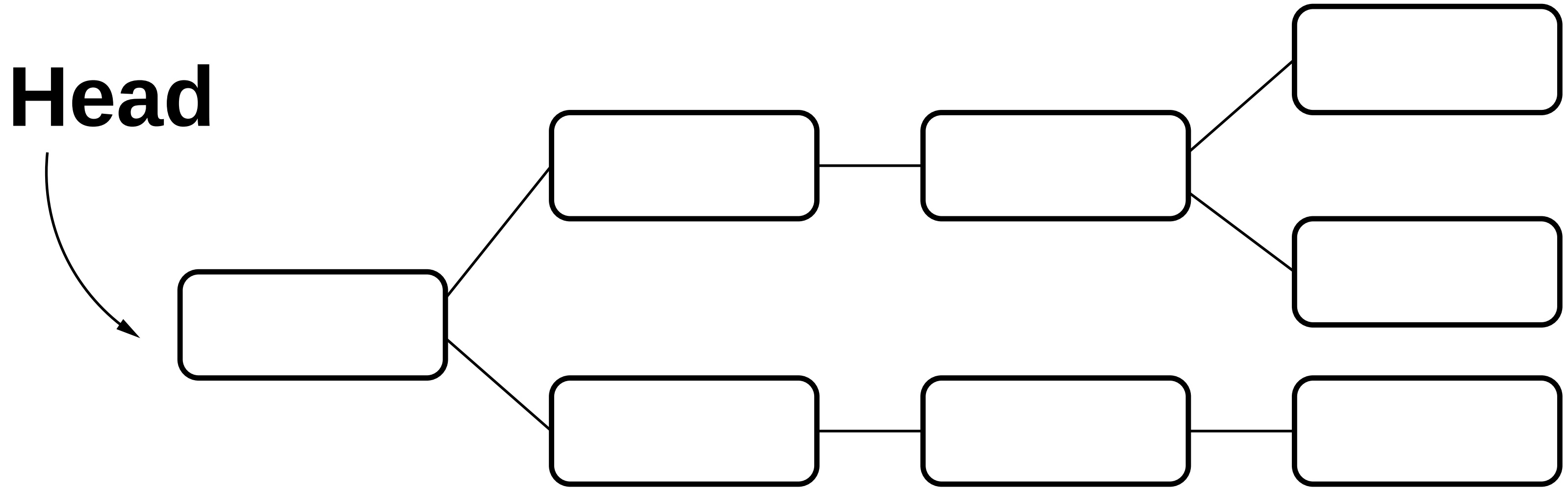
- Introduced in response to SCO suite
- Forced by need to know origins of patch added to kernel
- `Signed-off-by` means I know where the patch came from (at least as far back as the previous signoff)
- `Acked-by` means something different.
- Adhering to it requires the concept of immutable history
  - Linux development is now tied unbreakably to tools like bitkeeper

Origins of Git

- 2.6.12-rc2 was the last release to use Bitkeeper.
- After that, a large slew of kernel developers began developing git.
- Mercurial (hg) was also begun at this time as a response to the Bitkeeper induced crisis.
- Concepts were based on distributed source control learned from Bitkeeper
- Especially by the need to preserve immutable history.
- But were corrected for perceived mistakes Bitkeeper made.

Basic Concepts

- Git is a *tree* tracking tool, not a *change* tracking tool.
- Fundamental objects in Git are trees joined by commits.



The problems begin

- If Trees are the object, there are many files that remain the same between commits
- This would involve horrible duplication (multiple copies of the same file)
- Solution is to make git Content Accessible
- Every object is named and indexed by its content (sha1 hash)

Tracking trees and Content

- Renames now easy ... tree name changes but sha1 remains the same
- However, lack of change information between commits makes it very hard to track renames, adds and deletes.
- Easy if file contents don't change, but if they do can only do probability analysis to establish the rename.
- Fundamental principle of git: Making things happen is very easy; Finding what changed it much harder
  - Classic example is which commits touched this file.

Heads in Git

- Any given commit has one (or more) parents
- This forms a tree.
- The root commit is the only one that has no parent
- However, your current work is usually at the head of the tree.
- So, need pointer to the current working head of the tree
- `refs/heads` is where this is stored
- The head is automatically advanced as commits are made

Branches in Git

- Very simple.
- Every commit is a potential branch
- Git keeps track of branches via tree heads
- Git also keeps an idea of the curent working branch (what's checked out)
- Because git is content accessible, could store every git tree for every project in the same repository
  - As long as you remember where the heads are

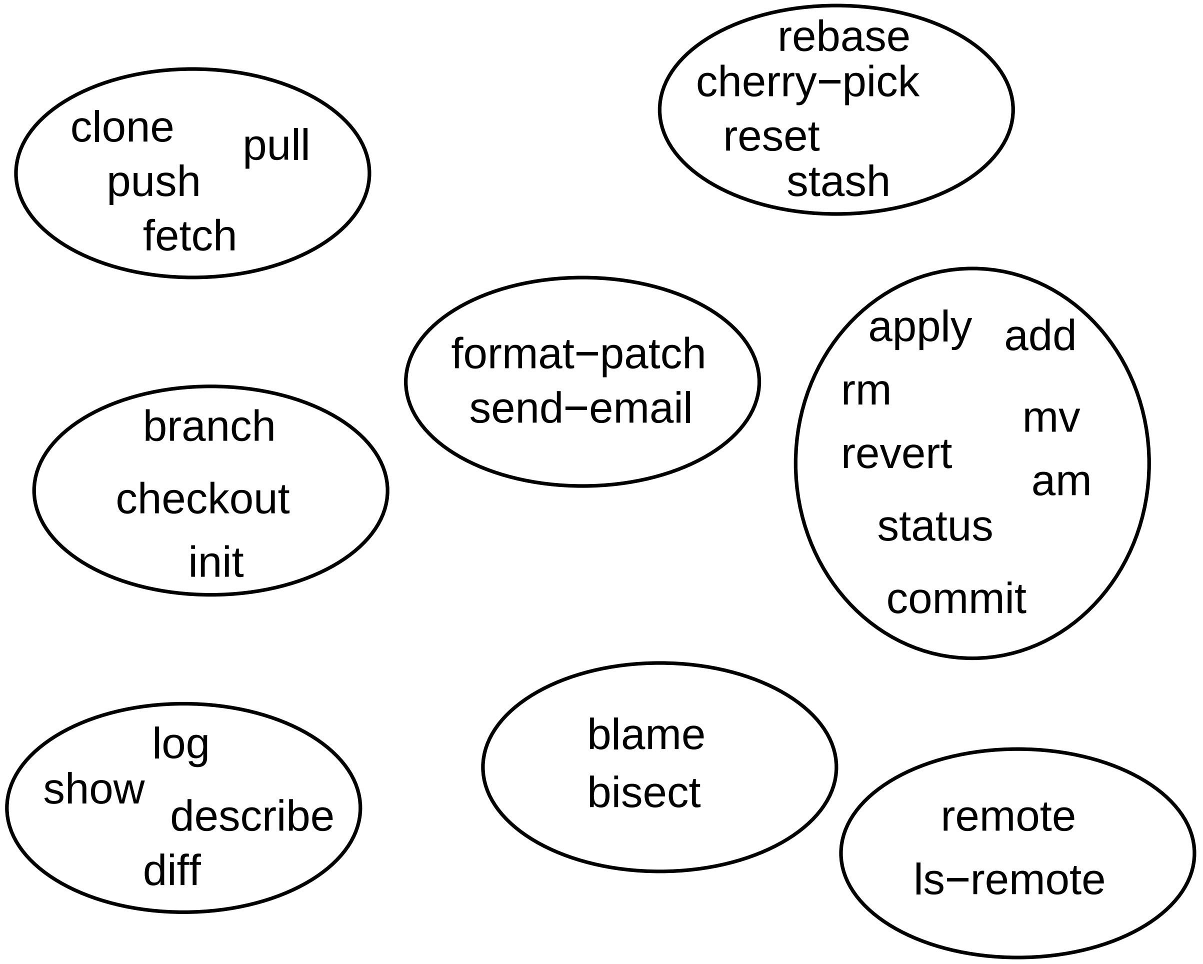
Branches and Tags

- A tag is just a label on a commit.
- A branch tracks a head, so it advances as commits are added
- A tag is purely static and allows you to get back to previous state (or label it)
- Branches tend to be local to your tree; tags may be shared with others
  - e.g. the labels linus puts on kernels: v2.6.32-rc3

Merging

- Since git has no special weave based file formats
- or any requirement to track changes at all
- Merging occurs simply when a commit has more than one parent
- There's no prescription of the merging algorithm at all
- At the moment, git uses a pluggable set for finding the best merge

Git Commands


clone
pull
push
fetch
rebase
cherry−pick
reset
stash
format−patch
send−email
apply
add
rm
mv
revert
am
status
commit
branch
checkout
init
log
show
describe
diff
blame
bisect
remote
ls−remote