Git Workshop

Peter Lundgren

September 6, 2016

- Introductions
- We've got a lot to talk about today
- We've also got a lot of time
- Ask questions
- Tell me if I'm going too fast

More Information

This presentation is avaliable at
`https://github.com/peterlundgren/git-workshop`

Download Git at `https://git-scm.com`

Learn more about Git at `https://progit.org`

Table of Contents

What is Git?

git *noun* \'git\
British
: a foolish or worthless person

What is Git?

Random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.

From the git README

What is Git?

Git is an open source,
distributed version control system
designed for speed and efficiency

Official tagline of Git

What is Git?

Git is an **open source**,
distributed version control system
designed for speed and efficiency

- LGPL-2.1
- https://github.com/git/git

Linus Torvalds

"I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'Git'"

- Linus Torvalds

- First release was 7 April 2005 from Linus Torvalds
- Kernel hacker mentality. It was written to manage the Linux kernel. So, it won't stop you from shooting yourself in the foot.

Junio Hamano

- Maintained by Junio Hamano since since 26 July 2005

And Many More

Junio C Hamano
Jeff King
Shawn O. Pearce
Linus Torvalds
Nguyn Thi Ngc Duy
Johannes Schindelin
Michael Haggerty
Jonathan Nieder
Ren Scharfe
Eric Wong
Jakub Narbski
Christian Couder
Johannes Sixt
Felipe Contreras
Nicolas Pitre
Paul Mackerras
Thomas Rast
Brandon Casey
var Arnfjr Bjarmason
Matthieu Moy
Michael J Gruber
Simon Hausmann
Jiang Xin
Petr Baudis
Alex Riesen
Stefan Beller

Ramkumar Ramachandra
Elia Pinto
Eric Sunshine
Johan Herland
Miklos Vajna
Ramsay Jones
Daniel Barkalow
J. Bruce Fields
SZEDER Gbor
David Aguilar
John Keeping
Pete Wyckoff
Elijah Newren
Kay Sievers
Pierre Habouzit
Jens Lehmann
Stephen Boyd
Tay Ray Chuan
Ralf Thielow
Paul Tan
Alexandre Julliard
Karsten Blees
Martin von Zweigbergk
Pat Thoyts
Clemens Buchacher
Giuseppe Bilotta

Alexander Gavrilov
Avery Pennarun
Jay Soffian
Torsten Bgershausen
brian m. carlson
Erik Faye-Lund
Karthik Nayak
Fredrik Kuivinen
Nanako Shiraishi
Ronnie Sahlberg
Jim Meyering
Frank Lichtenheld
Jon Seymour
Steffen Prohaska
Brian Gernhardt
Gerrit Pape
Martin Langhoff
Heiko Voigt
Mike Hommey
David Turner
Vasco Almeida
Peter Krefting
Jonas Fonseca
Markus Heidelberg
Matthias Lederhofer
Bert Wesarg

Lars Hjemli
Stephan Beyer
Michele Ballabio
Matthias Urlichs
Kirill Smelkov
Martin Koegler
Nick Hengeveld
Christian Stimming
Andy Parkins
Sergey Vlasov
H. Peter Anvin
Luben Tuikov
Ryan Anderson
Charles Bailey
Mark Levedahl
Sebastian Schuberth
Luke Diamand
Philip Oakley
Thomas Ackermann
Trn Ngc Qun
Ben Walton
Lars Schneider
Pavel Roskin
Santi Bjar
Adam Spiers
Dmitry Potapov

Marius Storm-Olsen
Sverre Rabbelier
Dan McGee
Jon Loeliger
Sean Estabrooks
Sven Verdoolaege
W. Trevor King
Sam Vilain
Bjrn Gustavsson
Carlos Martn Nieto
Uwe Kleine-Knig
Aneesh Kumar K.V
Lukas Sandstrm
Matthew Ogilvie
Han-Wen Nienhuys
Michael G. Schwern
Theodore Ts'o
Wincent Colaiuta
David Barr
Micha Kiedrowicz
Zbigniew Jdrzejewski-Szmek
Andreas Ericsson
Clestin Matte
Patrick Steinhardt

Dennis Stosberg
Kyle J. McKay
Tanay Abhra
Alex Henrie
Antoine Pelisse
David Kastrup
Jacob Keller
Jean-Noel Avila
Ralf Wildenhues
Stefano Lattarini
Julian Phillips
Eric W. Biederman
Ilari Liusvaara
Martin Waitz
Timo Hirvonen
Yann Dirson
Bjrn Steinbrink
Kevin Ballard
Richard Hansen
Thomas Gummerer
Carlos Rica
Kjetil Barvik
Kristian Hgsberg
Max Kirillov
Michael S. Tsirkin
Paolo Bonzini
Andy Whitcroft
Jason Riedy

Kevin Bracey
Mark Lodato
Michael Witten
Robert Fitzsimons
Robin Rosenberg
Tim Henigan
Alexander Shopov
Dmitry Ivankov
Karl Wiberg
Peter Eriksen
Brad King
Josef Weidendorfer
Matthias Kestenholz
Vitor Antunes
Adam Roben
Anders Kaseorg
Brian Downing
Jari Aalto
Lee Marlow

- Just a few of Git's contributors
- 1430 Contributors listed as of 2016-08-30

What is Git?

Git is an open source,
distributed version control system
designed for speed and efficiency

Centralized Version Control

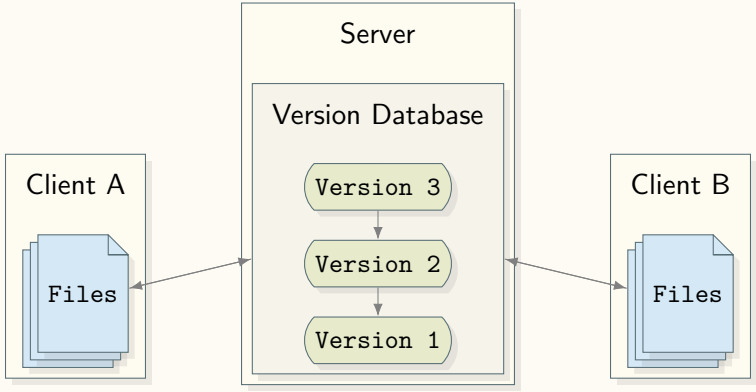


Figure 1: Centralized Version Control

- Client, server model
- Version database on only one server
- Download a snapshot
- Send incremental changes to the server
- Division of responsibility; some things only server can do, some things only client can do.

Distributed Version Control

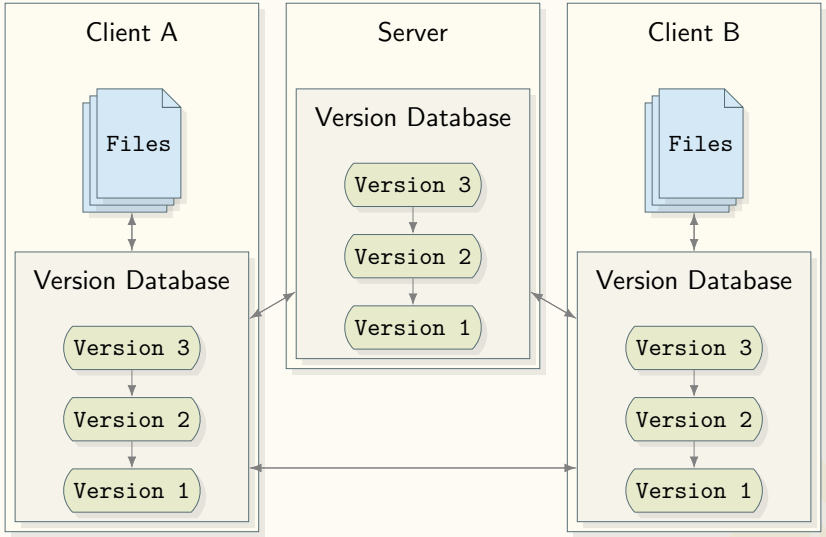


Figure 2: Distributed Version Control

- Peer to peer
- Version database on every machine
- Clients can talk to each other
- Download the entire repository
- Operate locally, share explicitely
- You can have a central server
- Servers are only different in that, as an optimization, they don't have working copies of the files. The clients are actually more featureful than the server.

What is Git?

Git is an open source,
distributed version control system
designed for **speed and efficiency**

Git is Fast

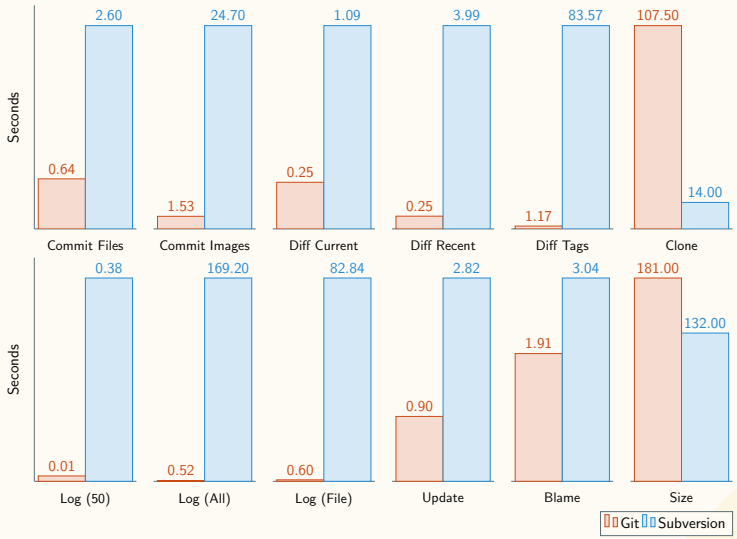


Figure 3: Runtime of Git and Subversion Commands

- Data from Scott Chacon http://git-scm.com/about/small-and-fast
- All operations are local except explicit synchronization
- No network access needed to:
  - Perform a diff
  - View file history
  - Commit changes
  - Merge branches
  - Switch branches
  - Checkout another revision
  - Blame a file
  - Search for the change that introduced a bug

What is Git?

Immutable

(almost) never removes data

- You will hear about rewriting history.
- **Question**: How many people have heard about rewriting history in Git?
- Git does not rewrite history.
- What Git does, is write a new history and move a pointer to it.
- Old history is still in the database.
- If you delete a branch, you're not deleting the work on that branch, you're deleting a pointer to it.
- Git keeps a log off all of this, so you can go back and find it.

What is Git?

Cryptographically **secure**

- Everything is hashed and addressed by its hash.
- Change content, change how you get that content.
- Sign tags and commits with PGP.
- Git can detect corruption.

Git is Popular

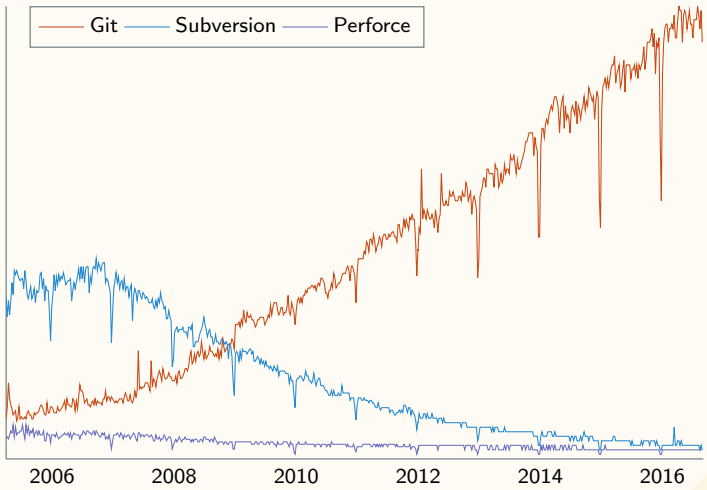


Figure 4: Google Trends Since First Git Release

- Dip every Christmas

Table of Contents

Objectives

- Understand how Git works and how to apply that to day to day development
- Learn the basic 12 everyday commands
- Know how to undo mistakes
- Learn how to use Git to collaborate
- Learn how to find help

12 Everyday Commands

- add
- diff
- merge
- branch
- fetch
- push
- checkout
- help
- rebase
- commit
- log
- status

- Git has 160 subcommands in 2.9.3
- I'll cover about 20 of them
- These are the 12 you'll use daily

Finding Help

Demo 1: `git help`

- I'll cover one of them right now.
- **Demo 1**: `git help`

12 Everyday Commands

- add
- diff
- merge
- branch
- fetch
- push
- checkout
- **help**
- rebase
- commit
- log
- status

- One down, 11 to go.

Learn 4 Ways

- Conceptual
- Commands
- Implementation
- Try It

- Conceptual - Computer science lecture. Diagrams of directed acyclic graphs and reachability. I'll lecture. We'll watch some lectures from Scott Chacon (Author of "Pro Git", CIO GitHub).
- Commands - Practical. How to use common commands.
- Implementation - How Git works under the hood.
- Try It - Practice!

Table of Contents

Your First Repository

Demo 2: `git init`

- **Demo 2**: `git init`

Table of Contents

Three Stage Thinking

- Edit
- Add
- Commit

Three Stage Thinking

Demo 3: Three Stage Thinking

- **Demo 3**: Three Stage Thinking

Lesson 1

Lesson 1: Three Stage Thinking

- **Lesson 1**: Three Stage Thinking

Commit Messages

The Technical Bits

- Short (aim for 50 characters or less) summary
- Followed by a blank line
- Body wrapped to 72 characters

- **Webpage**: `http://tbaggery.com/2008/04/19/a-note-about-git-commit-messages.html`
- Summary line must be separated by a blank space or many tools get a little confused
- Some views truncate the summary line; soft 50; hard 72
- Hard wrap body to 72 characters
- `git log`, `git format-patch`, etc do not wrap message

Commit Messages

The Conventional Bits

- Make your commits atomic
- Justify your changes; write detailed messages
- Write is the imperative: "Fix bug" and not "Fixed bug" or "Fixes bug"
- Present tense for current commit
- Past tense for earlier commits
- Future tense for later commits
- No period on subject line
- Meta-data at the bottom

- These are common expectations
- Like most social conventions, will be used to judge you more so than that they are technically superior
- I include them here so that you can understand and fit in
- Atomic: words like and, also, consider splitting commit
- Justify: open-source mailing list mentality; what, why, how
- Imperative style dates back to, at least, GNU changelogs
- Meta-data at the bottom: signed-of-by, change-id, issue tracker
- Look at a linux kernel log

12 Everyday Commands

- **add**
- **diff**
- merge
- branch
- fetch
- push
- checkout
- **help**
- rebase
- **commit**
- **log**
- **status**

- You've already seen these

Table of Contents

Trees, Hashes, and Blobs

oh My!

- **Video**: `http://youtu.be/ZDR433b0HJY?t=13m17s` - 0:21:02
- By Scott Chacon (Author of "Pro Git", CIO GitHub)

Trees, Hashes, and Blobs

Demo 4: Trees, Hashes, and Blobs

- **Demo 4**: Trees, Hashes, and Blobs

Table of Contents

Branch and Merge

Video

- **Video**: `http://youtu.be/ZDR433b0HJY?t=21m05s` - 0:30:35

Branches

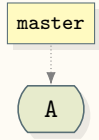


Figure 5: Branches are Pointers to Commits

- By default, 'git init' will create a master branch
- Most repositories have a master branch because most people are too lazy to change defaults
- Branches are pointers that point to commits

HEAD

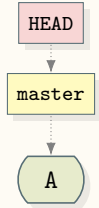


Figure 6: HEAD Points to Your Current Branch

- HEAD points to current branch
- HEAD is what you have checked out on your filesytem
- HEAD is the parent of your next commit

git branch

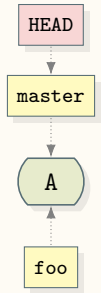


Figure 7: Creating a New Branch

```
$ git branch foo
```

- HEAD points to current branch
- HEAD is what you have checked out on your filesytem
- HEAD is the parent of your next commit
- Branches are cheap and fast. Writes 41 bytes to a file; that's it.
- `git branch foo` Creates a new branch called foo pointing to the same commit that HEAD is pointing to.

git checkout

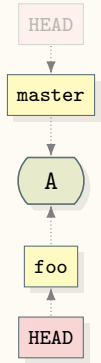


Figure 8: Switching Branches

```
$ git checkout foo
$ git branch
* foo
  master
```

- `git checkout` switches the current branch by changing what `HEAD` points to. If necessary, it will update your filesystem to match the commit pointed to by the branch.
- `git branch` will show you all of the local branches and put a star next to your current branch.

Make a Commit

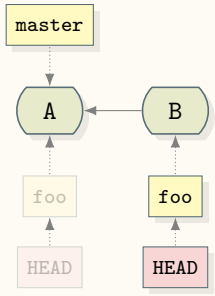


Figure 9: Make a Commit

```
$ git commit
```

- `git commit` Creates a new commit who's parent is whatever commit HEAD is pointing at. Then, it moves the branch `HEAD` is pointing at to the new commit.
- The only branch that moves is what `HEAD` points at.
- If you're ever scared about doing something, drop a branch behind. As long as you don't have a branch checked out, it's impossible to lose where it was.

Make Another Commit

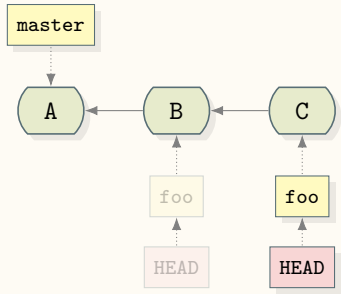


Figure 10: Make Another Commit

```
$ git commit
```

Checkout a New Branch

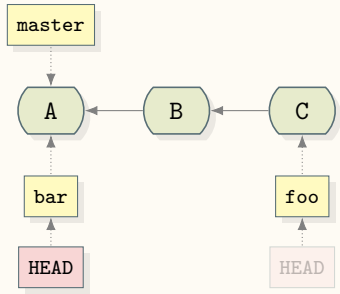


Figure 11: Checkout a New Branch

```
$ git checkout -b bar master
```

- `git checkout -b` is a shortcut for creating a branch and immediately checking it out.

Work on a New Branch

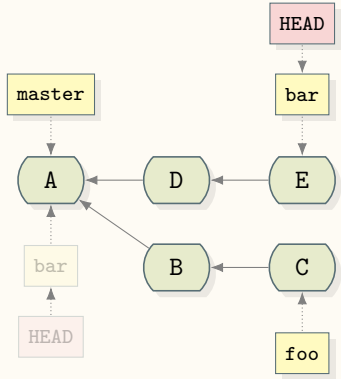


Figure 12: Work on a New Branch

```
$ git commit
$ git commit
```

- Make two commits on branch bar

Merging

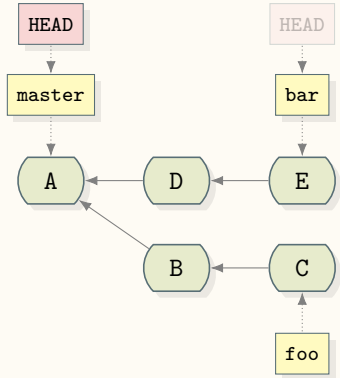


Figure 13: Checkout master

```
$ git checkout master
```

- Checkout the branch you want to modify / merge into
- Switch back to the master branch
- Next, I want the master have the changes on the bar branch
- **Question**: What should happen?
- Git checks to see if master is reachable from bar. If it is, it does the easiest possible thing.

Fast-Forward Merge

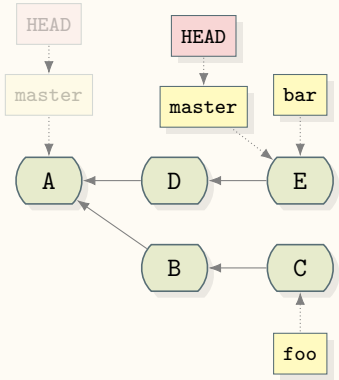


Figure 14: Fast-Forward Merge

```
$ git merge bar
```

- Want master took look like bar.
- It moves master up to the same commit that bar is at.
- Next, merge foo.
- **Question**: What should happen?
- It's going to do a non-fast-forward merge. It has to create a new tree. The snapshot with both master and foo doesn't exist yet.

Non-Fast-Forward Merge

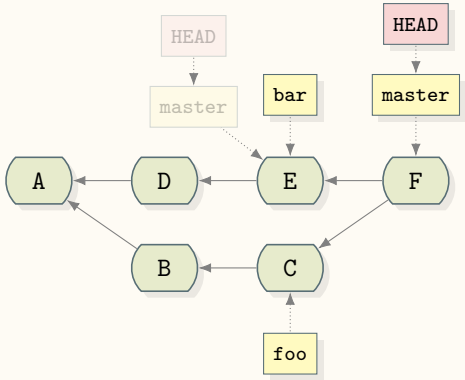


Figure 15: Non-Fast-Forward Merge

```
$ git merge foo
```

- Git created a new snapshot, F, and moved master to it.
- F now has both the changes on foo and bar.
- Git encodes this by having a commit with two parents.
- Neither foo nor bar moved. Branches that are not checked out will not move.

Lesson 2

Lesson 2: Branching and Merging

- **Lesson 2**: Branching and Merging
- Demo lesson afterwards.
- cat .git/HEAD
- cat .git/refs/heads/feature/change-name
- Branches are just 41 byte files.

Tags

Branches that Don't Move

- If you understand branches, then tags are easy.

Lesson 3

Lesson 3: Tags

- **Lesson 3**: Tags
- Start discussion: Lightweight vs. annotated tags.

12 Everyday Commands

- **add**
- **diff**
- **merge**
- **branch**
- fetch
- push
- **checkout**
- **help**
- rebase
- **commit**
- **log**
- **status**

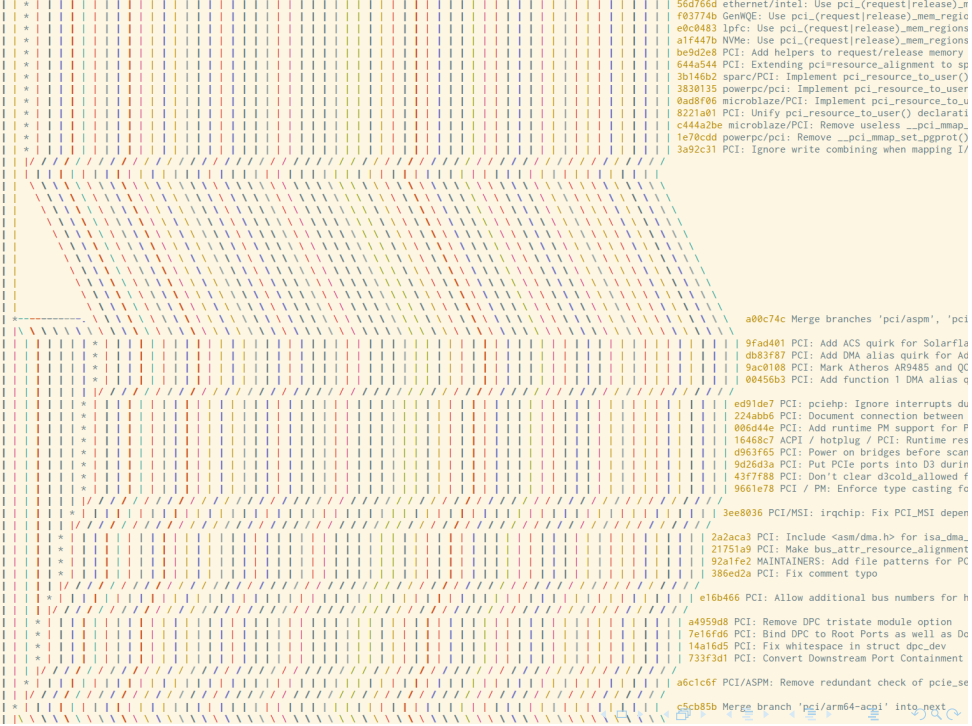
- We've added branch, checkout, and merge

Table of Contents

Rebase

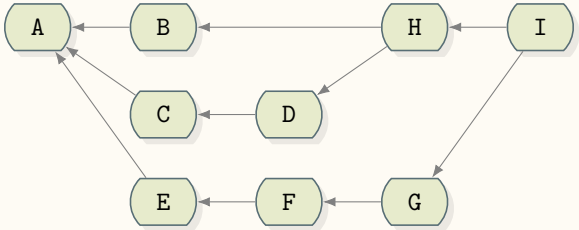
With great power comes great responsibility

- Swiss-army-knife of rewriting history
- The ability to edit or rewrite history sets Git apart from other version control systems



- Long running branches can be confusing
- This is from the linux kernel

Developing in Parallel or Taking Turns



(a) Multiple Concurrent Branches

(b) Linear History

Figure 16: Two Ways to Tell a Story

- What if we could use our version control tool to pretend that we took turns even when we develop in parallel
- Edit the top history to look like the bottom
- **Question**: Is this even a good idea?
- The first use of `git rebase` we will look at will do this

Rebase

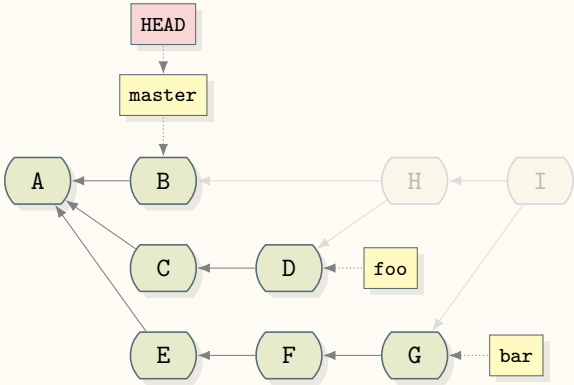


Figure 17: Two Branches to Rebase

- Instead of creating the merge commits H and I we'll rebase foo and bar onto master in order to create a linear history

Rebase

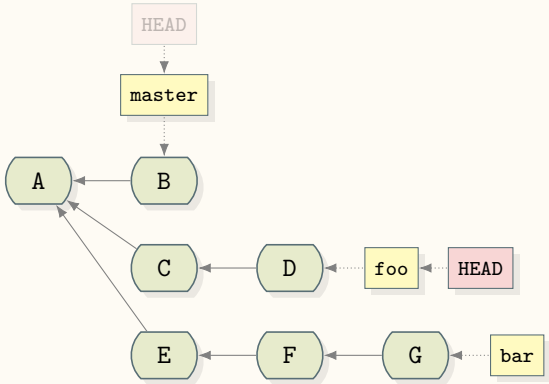


Figure 18: Checkout foo

```
$ git checkout foo
```

- First, we'll rebase foo onto master
- We must checkout foo first
- Remember Git will not touch branches you don't have checked out
- We want to take the branch foo, containing commits C and D, and reaply them after commit B

Rebase

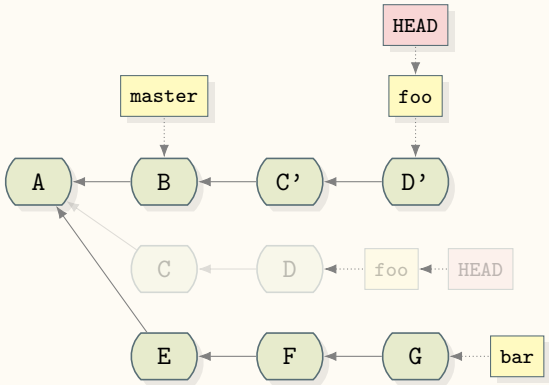


Figure 19: Rebase foo onto master

```
$ git rebase master
```

- ... to do that, we use the command `git rebase master`
- Commits C' and D' have the
  - same deltas as C and D,
  - same commit message,
  - same author,
- but different sha1sum because they have
  - different parents,
  - and different trees (different snapshots);
  - because the snapshots also contain the changes in B
- Rebase works by
  - first finding a common ancestor (A)
  - making patches for each commit in the source branch
  - and reapplying them on the destination branch

Rebase

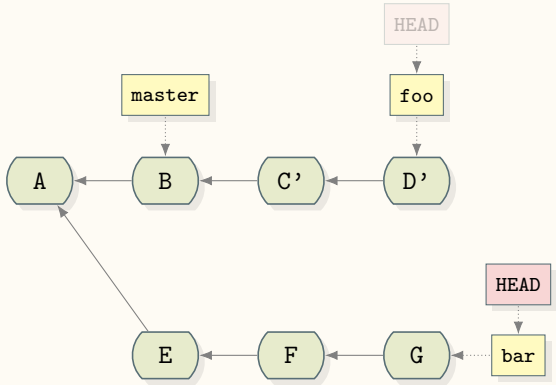


Figure 20: Checkout bar

```
$ git checkout bar
```

- Repeat the process for branch bar

Rebase

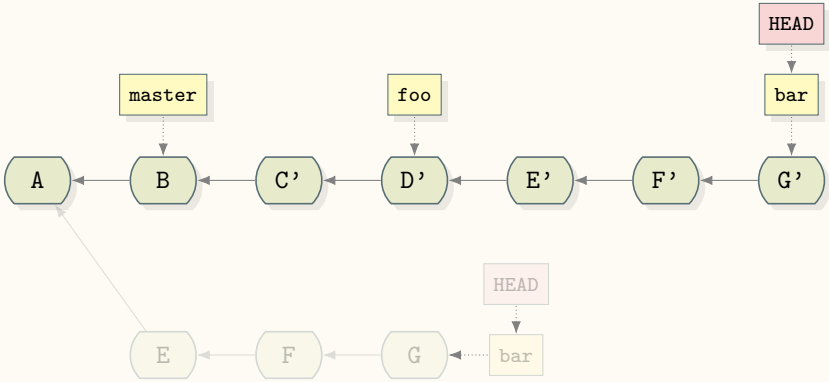


Figure 21: Rebase bar Onto foo

```
$ git rebase foo
```

- Reapply E, F, and G after D'

Rebase

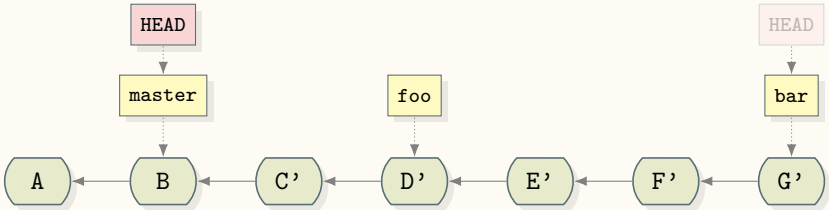


Figure 22: Update master

```
$ git checkout master
```

- Now that we have the version of history that we want, let's update master
- First checkout master

Rebase

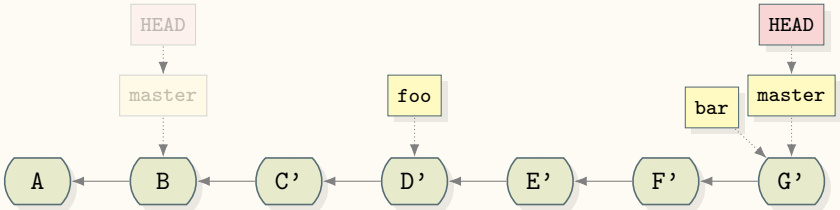


Figure 23: Merge Changes Into master

```
$ git merge bar
```

- Merge bar into master
- **Question**: What kind of merge is this? Fast-forward merge

Rebase

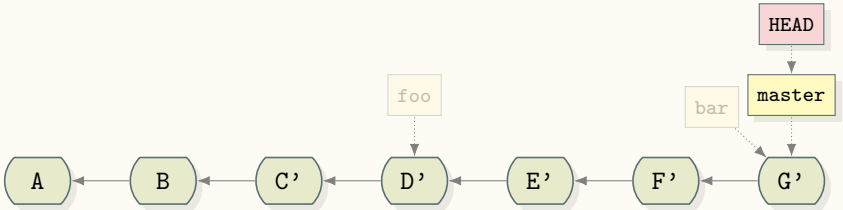


Figure 24: Delete Branches

```
$ git branch -d foo
$ git branch -d bar
```

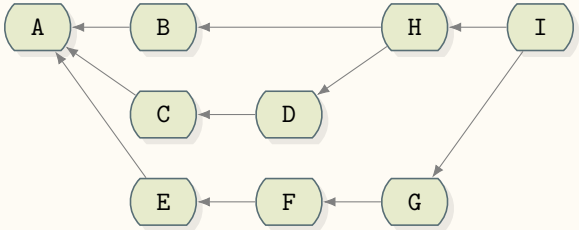
- If foo and bar are not needed anymore

Lesson 4

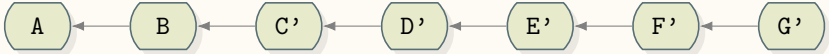
Lesson 4: Rebase

- **Lesson 4**: Rebase

Merge vs. Rebase



(a) Multiple Concurrent Branches



(b) Linear History

Figure 25: Two Ways to Tell a Story

- Two ways to apply your changes to the mainline
- Start discussion: Merge vs. Rebase

Interactive Rebase

Demo 5: `git rebase --interactive`

- **Demo 5**: `git rebase --interactive`
- Reorder, add, remove, edit, reword, squash commits

Lesson 5

Lesson 5: Interactive Rebase

- **Lesson 5**: Interactive Rebase

12 Everyday Commands

- **add**
- **diff**
- **merge**
- **branch**
- fetch
- push
- **checkout**
- **help**
- **rebase**
- **commit**
- **log**
- **status**

- Now we've learned rebase

Table of Contents

Stash

```
$ git checkout foo
error: Your local changes to the following files would
be overwritten by checkout:
    a.txt
    b.txt
Please commit your changes or stash them before you
switch branches.
Aborting
$ git merge foo
error: The following untracked working tree files would
be overwritten by merge:
    a.txt
    b.txt
Please move or remove them before you merge.
Aborting
$ git rebase -i HEAD~3
Cannot rebase: You have unstaged changes.
Please commit or stash them.
```

- You'll run into these error message before too long
- Checkout, merge, and rebase all want clean working directories
- so that they don't mess with your unstaged and uncommitted work
- They recommend to commit or stash your changes. What is stash?

Stash

Demo 6: `git stash`

- **Demo 6**: `git stash`

Table of Contents

Managing Conflicts

- `git merge`
- `git rebase`
- `git stash pop`

- Merge, rebase, and stash pop can result in conflicts

Merge Conflict

```
$ git merge bugfixes
Auto-merging hello.c
CONFLICT (content): Merge conflict in hello.c
Automatic merge failed; fix conflicts and then commit
the result.
$ git status
# On branch master
# You have unmerged paths.
#   (fix conflicts and run "git commit")
#   (use "git merge --abort" to abort the merge)
#
# Unmerged paths:
#   (use "git add <file>..." to mark resolution)
#
#       both modified:   hello.c
#
no changes added to commit (use "git add" and/or
"git commit -a")
```

- If a merge conflicts, it will stop before creating the merge commit
- `git status` will remind you that you have conflicts (unmerged paths).
- Either abort or resolve the changes

Merge Conflict Markers

```
#include <stdio.h>

int main(void)
{
<<<<<<< HEAD
    printf("Hello Class");
=======
    printf("Hello World\n");
    return 0;
>>>>>>> bugfixes
}
```

- Traditional conflict markers.
- Same as in svn.

Merge Conflict Markers

```
$ git config --global merge.conflictstyle diff3
```

```
#include <stdio.h>

int main(void)
{
<<<<<<< HEAD
    printf("Hello Class");
||||||| merged common ancestors
    printf("Hello World");
=======
    printf("Hello World\n");
    return 0;
>>>>>>> bugfixes
}
```

- Alternatively, the diff3 style, which I highly recommend, adds the merge base in the middle

Merge Tools

```
$ git mergetool --tool=<tool>
```

- araxis
- bc
- bc3
- codecompare
- deltawalker
- diffmerge
- diffuse
- ecmerge
- emerge
- examdiff
- gvimdiff
- gvimdiff2
- gvimdiff3
- kdiff3
- meld
- opendiff
- p4merge
- tkdiff
- tortoisemerge
- vimdiff
- vimdiff2
- vimdiff3
- winmerge

- If you'd like to use a merge tool,
- these are all supported out of the box
- Can use others with a little bit of configuration to tell git how to launch them

Merge Conflicts

Git stops before creating merge commit. Either:

1. Abort with `git merge --abort`

or

1. Resolve the conflicts
2. Mark files resolved with `git add <file>`
3. Finish the merge with `git commit`

- Easier said than done...
- Conflicts are recorded in merge commit message

Lesson 6

Lesson 6: Merge Conflicts

- ... so let's do it
- **Lesson 6**: Merge Conflicts

Rebase Conflict

```
$ git rebase master
...
CONFLICT (content): Merge conflict in hello.c
error: Failed to merge in the changes.
Patch failed at 0002 Print newline
The copy of the patch that failed is found in:
.git/rebase-apply/patch

When you have resolved this problem, run
"git rebase --continue".
If you prefer to skip this patch, run
"git rebase --skip" instead.
To check out the original branch and stop rebasing, run
"git rebase --abort".
```

- Rebase applies each commit one at a time
- If a rebase conflicts, it will stop before creating the commit at whichever commit in the rebase conflicts

Rebase Conflict

```
$ git status
# rebase in progress; onto 71f26c3
# You are currently rebasing branch 'bugfixes' on '71f26c3'.
#   (fix conflicts and then run "git rebase --continue")
#   (use "git rebase --skip" to skip this patch)
#   (use "git rebase --abort" to check out the original branch)
#
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#       both modified:   hello.c
#
no changes added to commit (use "git add" and/or
"git commit -a")
```

- `git status` will remind you that you have a rebase in progress with conflicts (unmerged paths).
- Either abort, skip, or resolve the changes
- The skip option is new
- Skip will not attempt to apply that change as if you had ommitted it with a `git rebase -i`

Rebase Conflicts

Git stops before each conflicting commit. Either:

1. Abort with `git rebase --abort`

or

1. Skip one patch with `git rebase --skip`

or

1. Resolve the conflicts
2. Mark files resolved with `git add <file>`
3. Continue the rebase with `git rebase --continue`

- Similar process as a merge conflict but with the added option to skip
- Note that the command to continue is no longer `git commit`

Lesson 7

Lesson 7: Rebase Conflicts

- **Lesson 7**: Rebase Conflicts
- After lesson: **Question**: How was this different than resolving the merge conflict?
- You can get similar one commit at a time merge conflict resolution with `git rerere`

Stash Conflict

```
$ git stash pop
Auto-merging hello.c
CONFLICT (content): Merge conflict in hello.c
$ git status
# On branch master
# Unmerged paths:
#   (use "git reset HEAD <file>..." to unstage)
#   (use "git add <file>..." to mark resolution)
#
#       both modified:   hello.c
#
no changes added to commit (use "git add" and/or
"git commit -a")
```

- This is starting to look familiar

Stash Conflicts

Git applies the changes with conflict markers but leaves the change on the stack.

1. Resolve the conflicts
2. Mark files resolved with `git add <file>` or `git reset HEAD <file>`
3. If you don't need the stash anymore `git stash drop`
4. Go back to work

- Conflicting `git stash apply` is one of the difficult
- Will skip a lesson for stash, you get the idea

Table of Contents

Reachability and Garbage Collection

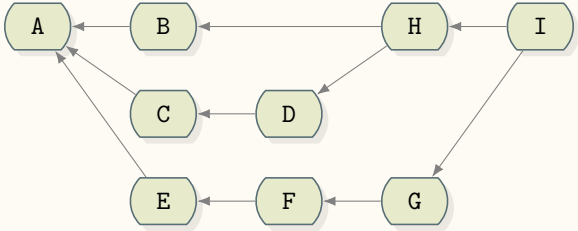


Figure 26: Reachability

- **Question**: What is reachable from H if you follow the parent pointers?

Reachability and Garbage Collection

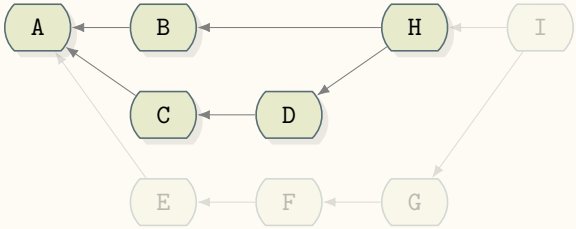


Figure 27: Reachable from H

- Reachable from H

Reachability and Garbage Collection

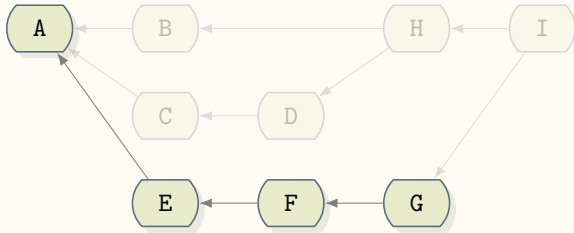


Figure 28: Reachable from G

- Reachable from G

Reachability and Garbage Collection

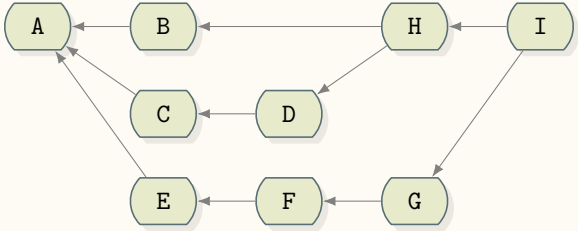


Figure 29: Reachable from I

- Everything on this graph is reachable from I

Reachability and Garbage Collection

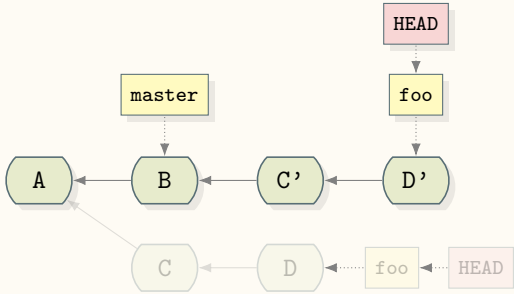


Figure 30: Rebase foo onto master

```
$ git rebase master
```

- Revisiting earlier rebase example
- Calling this rewriting history was a bit of a misnomer
- An alternate history C' and D' are created and `HEAD` and foo point to the alternate history
- The old history is still there. It is no longer referenced by foo
- `git log` doesn't show it
- `git log -all` doesn't show it
- But it's still in your database
- `git show <sha1sum>` of either C or D still works
- C and D won't be garbage collected for at least 30 days
- We'll come back to garbage collection later

Recovering Your Work

Made a mistake?
Need a change from several amends ago?
Messed up a rebase?

- `git reflog`
- `git fsck`
- `git reset`

- You can (almost) always recover your work
- If you've ever added a file, that snapshot is in your database
- We can recover it

Lesson 8

Lesson 8: Recovering Your Work

- **Lesson 8**: Recovering Your Work

Reachability and Garbage Collection

Garbage Collection

- Objects that are not reachable from branches, tags, and 30 days old
- Reflog expires at 90 days, but objects may be garbaged collected after 30
- That said, gc doesn't run very often in most use cases
- gc.auto (6700) loose objects
- gc.autoPackLimit (50) pack files
- Garbage collection can only happen if you run a git command
- All of this is configurable

Table of Contents

Remotes

Playing with Others

- **Video**: `http://youtu.be/ZDR433b0HJY?t=58m45s` - 1:10:15

Lesson 9

Lesson 9: Remotes

- **Lesson 9**: Remotes
- I apologize for how silly that was; you're not likely to ever have that many people trying to push at the same time
- If you haven't finished, don't worry about it
- It is important to recognize that Git does not do ANY merge resolutions on the remote
- While this is much safer than the alternative, it does present a bottleneck if too many people must push to the same remote
- Large, open source projects with many contributors tend to use more pull oriented workflows
- `git format-patch`, `git send-email`, GitHub pull requests, and so on

12 Everyday Commands

- **add**
- **diff**
- **merge**
- **branch**
- **fetch**
- **push**
- **checkout**
- **help**
- **rebase**
- **commit**
- **log**
- **status**

- Now we've covered all of them

Table of Contents

Submodules

Demo 7: `git submodule`

- **Demo 7**: `git submodule`

Table of Contents

Pros and Cons

- What is Git good at?
- What is Git bad at?
- When should you use Git?
- When should you not use Git?
- How is Git different from other version control systems?

- **Discussion**: Pros and Cons
- Start Pros, Neutral, Cons list

Pros and Cons

Creating a new repository is trivial

- Here's a list of things that we might not have covered yet
- `git init`
- Don't need a dedicated server
- All you need is a file system

Pros and Cons

Staging Area

- More complicated
- Help you specify what changes belong in your next commit
- Distinction between whats on disk and what youre about to commit
- Commit part of some files changes

Pros and Cons

Peer-to-peer synchronization over several protocols

- git://
- ssh://
- http://
- https://
- file://
- nfs
- email
- RFC 1149

- Peer-to-peer, from anyone to anyone
- RFC 1149: A Standard for the Transmission of IP Datagrams on Avian Carriers
- Hashed, with commit messages, optionally cryptographically signed
- If you can transfer a file, you can share changes

Pros and Cons

Branching and Merging

- Branches are a fundamental part of version control
- A git branch is a 41 byte file
- Branches are local, can be messy
- They dont require permission
- They dont have to be public
- Merging algorithm was better than SVNs until 1.8 in 2013

Pros and Cons

Access Control

- SVN must be managed at the central server
- SVN can assign permissions per directory
- Git is all or nothing per repository, read or write access, that's it
- Can add a lot more control via other tools
- Can't partially limit read access at all

Pros and Cons

Obliterate

- Completely remove something from history
- This is an option you won't need often
- Accidentally added very large files
- Legal/security reasons
- Many copies of the history
- Changes are detectable

Pros and Cons

Locks

- No built-in way to do locks
- Locks require a central server
- Some git servers like gitolite support file locking

Pros and Cons

Binaries and Very Large Repositories

- Binaries dont compress well, all versions stored locally
- Entire repository must be cloned
- Binaries dont benefit from branching and merging

Pros and Cons

First Class Commits and Rewriting History

- Commits can be manipulated
- Edited, squashed, split, reordered, cherry-pick, share, signed-off
- Distinction between auditing your progress and publishing a finished work
- Implications for review
- Not a complete audit trail

Pros and Cons

Revision Hashes Instead of Revision Numbers

- No linear list of revision numbers

Pros and Cons

Ease of Use

- Git is challenging
- We've been here for quite a few hours already

Table of Contents

Local Infrastructure

- Fill to suit

Table of Contents

Questions?