Advanced  git

How it works (and why it is so complicated)

Tom Wiesing

KWARC second hour talk

March 29, 2016

Overview

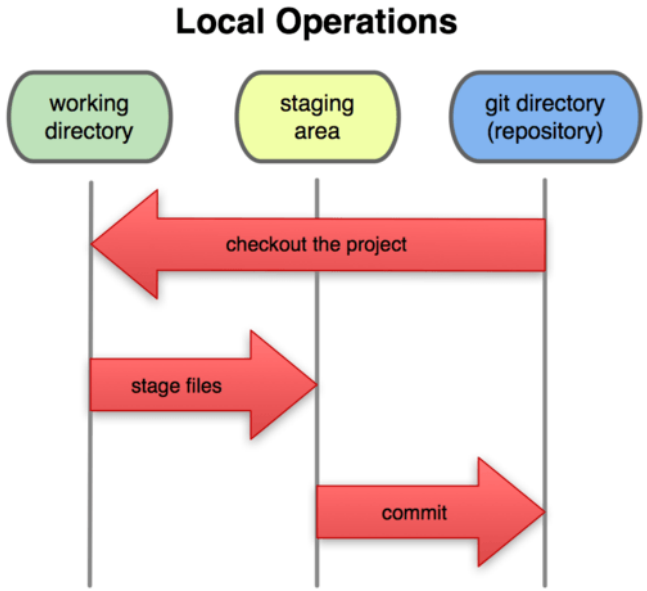
- The basics (what you should already know)
  - What is git?
  - Working directory, index & repository
- Object Storage (the "filesystem" of git)
  - BLOBs, Trees, Commits
- References in git
  - HEADs, Branches, TAGs
- Merging & Remotes (the fun part)
  - Merging & Rebasing
  - Remotes: Fetch, Push & Pull
- Conclusion (What git is and what it is not)

The basics (1): What is git?

- git – "the stupid content tracker"
  - open-source version control system
  - fast, scalable, distributable
- originally developed in 2005 for maintaining the linux kernel source code

The basics (2): Working directory, index & repository

- git maintains multiple versions of a project
- each repository has
  - a working directory (where files are editable)
  - a staging area (also called index)
  - a git directory (contains all the history of the repository)
- basic commands
  - git add, git commit, git checkout



Object Storage (1): Object overview

- git is a key-value store
  - keys = SHA-1 hashes
- Three main types of objects:
  - BLOBs (for file content)
  - Trees (for storing a directory of files)
  - Commits (to store multiple versions)

Object Storage (2): BLOBs

- stores the *content* of a file
- problem: no meta-information such as filename, path

```
$ echo 'test content' | git hash-object -w --stdin
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

```
$ git cat-file -p d670460b4b4aece5915caf5c68d12f560a9fe3e4
test content
```

- storing multiple versions of the same file is no problem

```
$ echo 'version 1' > test.txt
$ git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

```
$ echo 'version 2' > test.txt
$ git hash-object -w test.txt
1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

Object Storage (3): BLOBs continued

- we can checkout each version individually

```
$ git cat-file -p 83baae61804e65cc73a7201a7252750c76066a30 > test.txt
$ cat test.txt
version 1
```

```
$ git cat-file -p 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a > test.txt
$ cat test.txt
version 2
```

- the objects are just stored on disk

```
$ find .git/objects -type f
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
.git/objects/1f/7a7a472abf3dd9643fd615f6da379c4acb3e3a
```

- their type is also stored

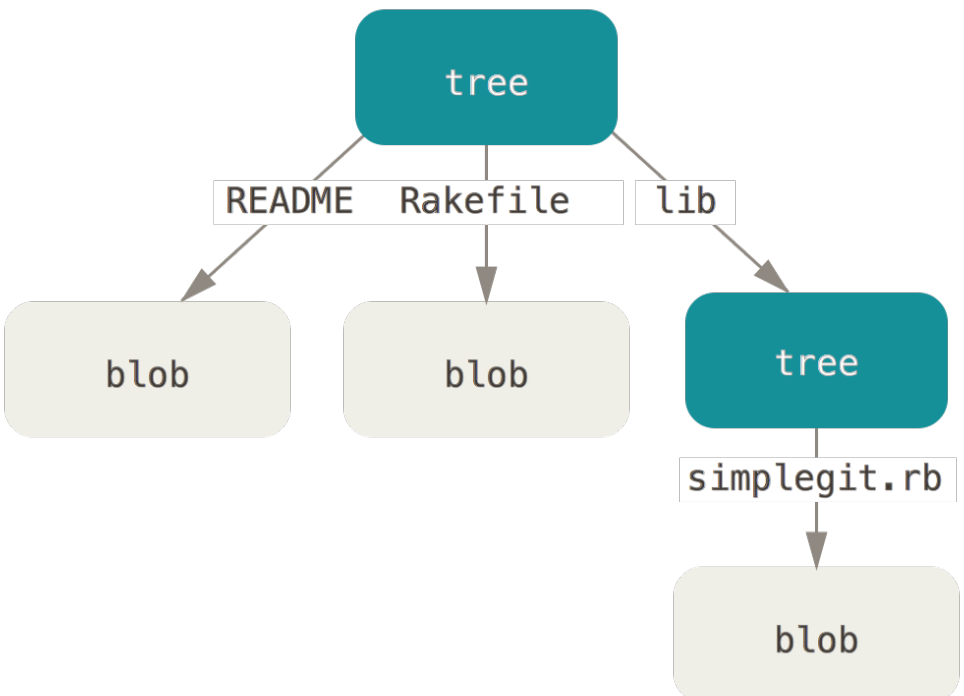
```
$ git cat-file -t 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a
blob
```

Object Storage (4): Trees

- each node represents a single directory
- must contain 1 or more entries (so no empty folders)
- includes file names + mode

```
$ git cat-file -p master^{tree}
100644 blob a906cb2a4a904a152e80877d4088654daad0c859      README
100644 blob 8f94139338f9404f26296befa88755fc2598c289      Rakefile
040000 tree 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0      lib
```

```
$ git cat-file -p 99f1a6d12cb4b6f19c8655fca46c3ecf317074e0
100644 blob 47c6340d6459e05787f644c2447d2595f5d3a54b      simplegit.rb
```



Object Storage (5): The Index as a Tree

- we can use a tree to represent the index
- we can update the tree with the file we created above

```
$ git update-index --add --cacheinfo 100644 \
  83baae61804e65cc73a7201a7252750c76066a30 test.txt
$ git write-tree
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git cat-file -p d8329fc1cc938780ffdd9f94e0d364e0ea74f579
100644 blob 83baae61804e65cc73a7201a7252750c76066a30      test.txt
```

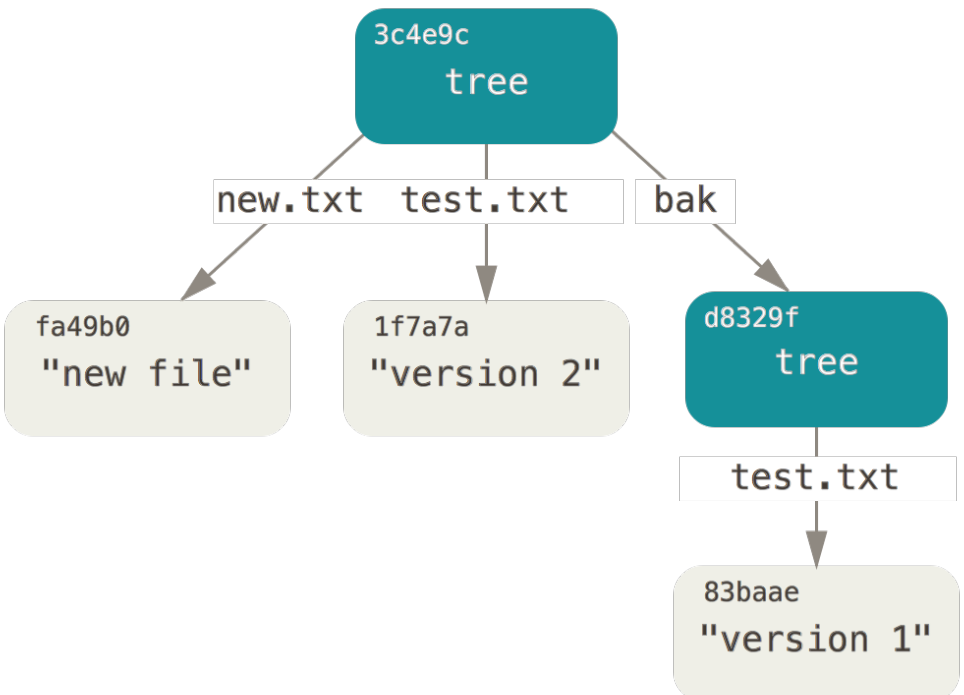
- we can add yet another file to it

```
$ echo 'new file' > new.txt
$ git update-index test.txt
$ git update-index --add new.txt
$ git write-tree
0155eb4229851634a0f03eb265b69f5a2d56f341
```

Object Storage (6): The Index as a Tree continued

- we can also add the same tree as a sub-directory

```
$ git read-tree --prefix=bak d8329fc1cc938780ffdd9f94e0d364e0ea74f579
$ git write-tree
3c4e9cd789d88d8d89c1073707c3585e41b0e614
$ git cat-file -p 3c4e9cd789d88d8d89c1073707c3585e41b0e614
040000 tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579      bak
100644 blob fa49b077972391ad58037050f2a75f74e3671e92      new.txt
100644 blob 1f7a7a472abf3dd9643fd615f6da379c4acb3e3a      test.txt
```



Object Storage (7): Commit objects

- we also want to store different commits
- each commit contains
  - a tree representing the current state
  - the parent commit
  - meta-information, such as author and time
  - (you may get different SHAs because of this)
- we can make a single commit

```
$ echo 'first commit' | git commit-tree d8329f
35f8b9255a9c68f80d90201ae14c39d9c9b66b2a
$ git cat-file -p 35f8b9
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Tom Wiesing <tkw01536@gmail.com> 1458923656 +0100
committer Tom Wiesing <tkw01536@gmail.com> 1458923656 +0100

first commit
```

- we can also make commits referencing earlier ones

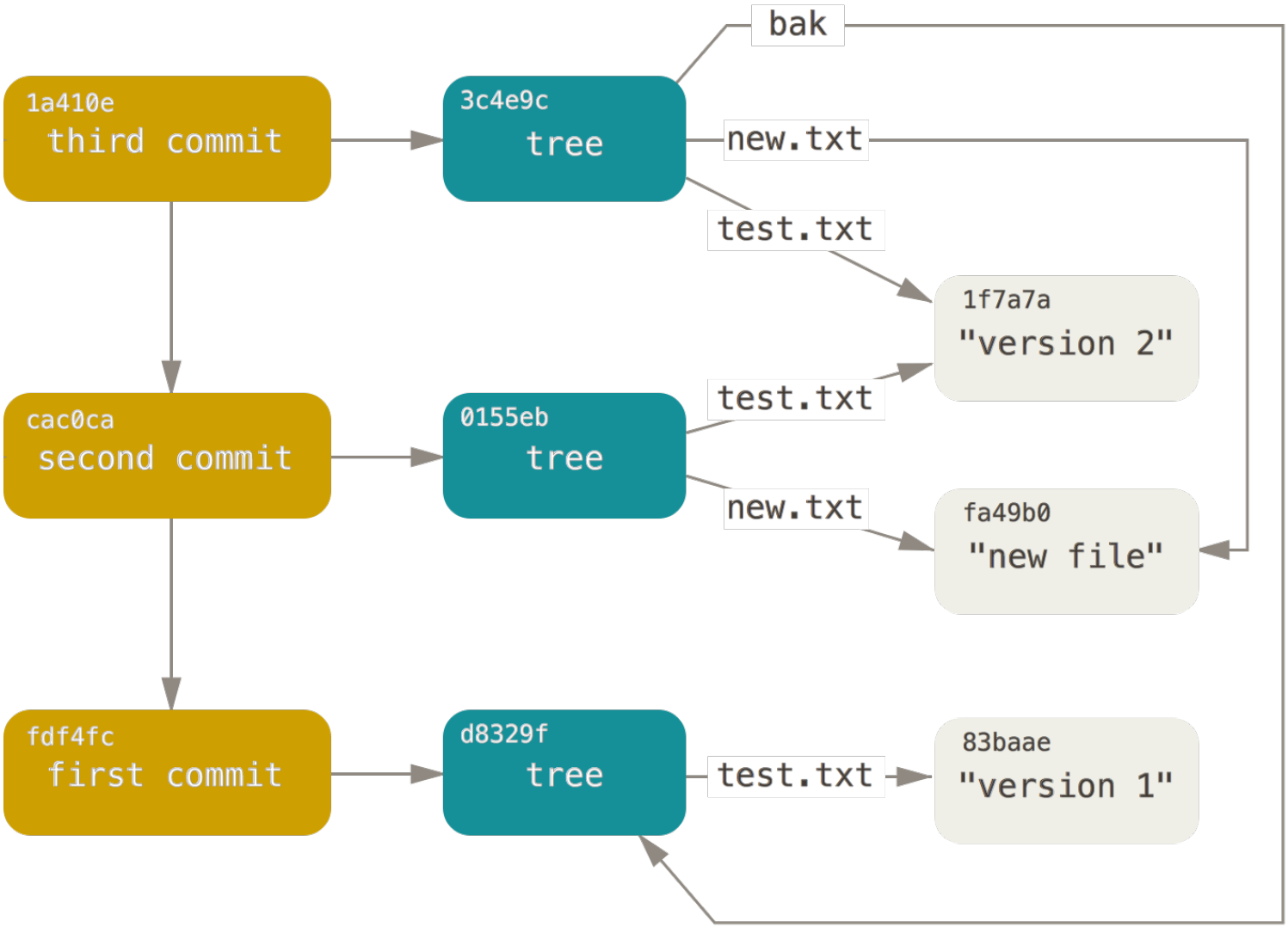
```
$ echo 'second commit' | git commit-tree 0155eb -p 35f8b9
0d9d54e2c438e22d6656fa1bdca7d76a36d3589c
$ echo 'third commit'  | git commit-tree 3c4e9c -p 0d9d54
970ac2c0207ad51caccce0a71d21283ff7109254
```

Object Storage (8): Commit objects continued

- we can now look at the history

```
$ git log --stat 970ac2
commit 970ac2c0207ad51caccce0a71d21283ff7109254
Author: Tom Wiesing <tkw01536@gmail.com>
Date:   Fri Mar 25 17:38:21 2016 +0100

    third commit

 bak/test.txt | 1 +
 1 file changed, 1 insertion(+)

commit 0d9d54e2c438e22d6656fa1bdca7d76a36d3589c
Author: Tom Wiesing <tkw01536@gmail.com>
Date:   Fri Mar 25 17:38:07 2016 +0100

    second commit

 new.txt  | 1 +
 test.txt | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)

commit 35f8b9255a9c68f80d90201ae14c39d9c9b66b2a
Author: Tom Wiesing <tkw01536@gmail.com>
Date:   Fri Mar 25 17:34:16 2016 +0100

    first commit

 test.txt | 1 +
 1 file changed, 1 insertion(+)
```

Object Storage (9): Commit objects continued



References (1): What are REFS?

- a reference is a pointer to a commit
- git stores these as simple files

```
$ find .git/refs
.git/refs
.git/refs/heads
.git/refs/tags
$ find .git/refs -type f
```

- we can write them manually

```
$ echo '970ac2c0207ad51caccce0a71d21283ff7109254' \
  > .git/refs/heads/master
$ git log --pretty=oneline master
970ac2c0207ad51caccce0a71d21283ff7109254 third commit
0d9d54e2c438e22d6656fa1bdca7d76a36d3589c second commit
35f8b9255a9c68f80d90201ae14c39d9c9b66b2a first commit
```

References (2): Branches

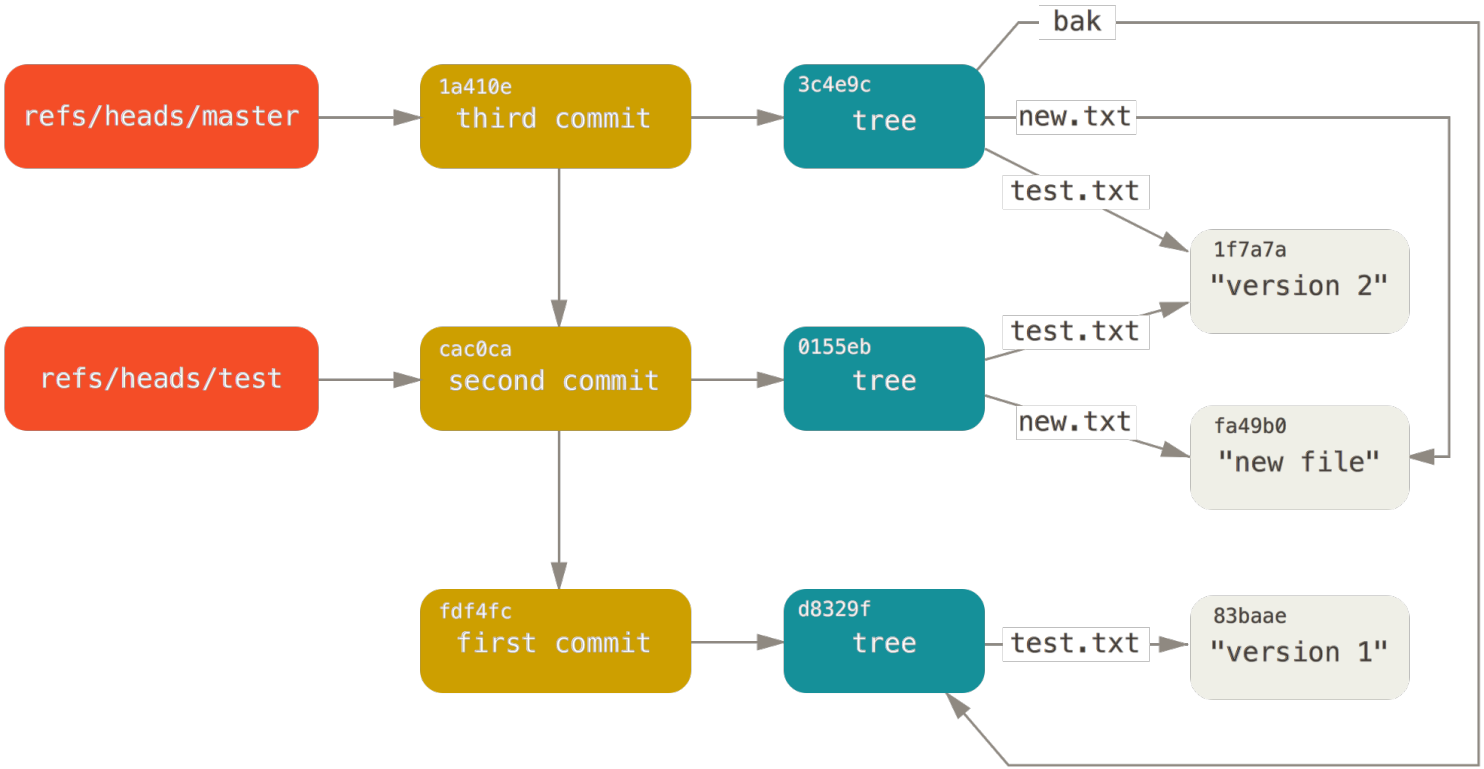
- we can use git update-ref instead

```
$ git update-ref refs/heads/master \
  970ac2c0207ad51caccce0a71d21283ff7109254
```

- each branch is just a simple REF (i. e. a pointer)
- we can create one easily

```
$ git update-ref refs/heads/test \
  0d9d54e2c438e22d6656fa1bdca7d76a36d3589c
$ git log --pretty=oneline test
0d9d54e2c438e22d6656fa1bdca7d76a36d3589c second commit
35f8b9255a9c68f80d90201ae14c39d9c9b66b2a first commit
```

References (3): Branches continued



References (4): The HEAD

- the HEAD is a *symbolic* reference to the current branch
- each branch has its own HEAD (as you have already seen)

```
$ cat .git/HEAD
ref: refs/heads/master
$ git checkout test
$ cat .git/HEAD
ref: refs/heads/test
```

- we could write the symbolic-ref manually
- but we can better use

```
$ git symbolic-ref HEAD refs/heads/test
$ cat .git/HEAD
ref: refs/heads/test
```

References (5): TAGs

- TAGs are similar to commits
- they point to a given version (identified by a commit)
- can be light-weight or annotated
- we do not go into details here

References (6): Remotes

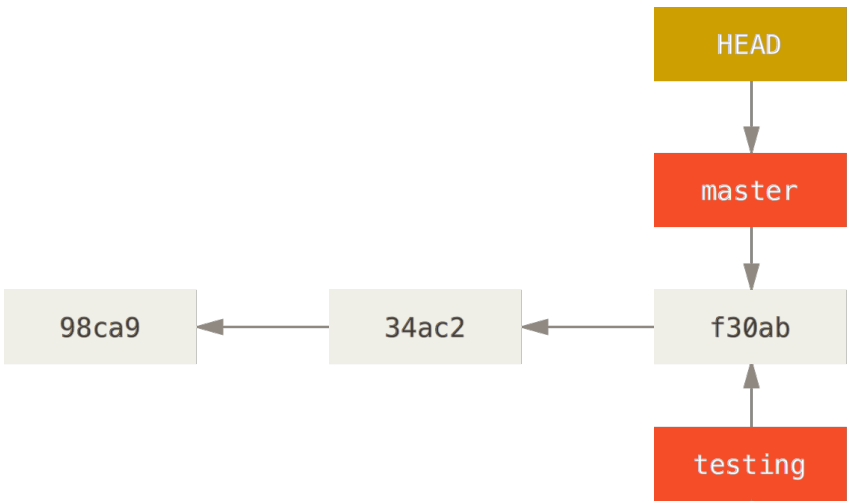
- remotes are similar to branches
- they are considered *read-only*
- whenever we fetch / push, they are updated
- we will go into remotes in a later section

Merging (1): Branching

- we have already learned what branches are
- we saw how to create them manually
- they are intended to be created with

```
$ git branch testing
```

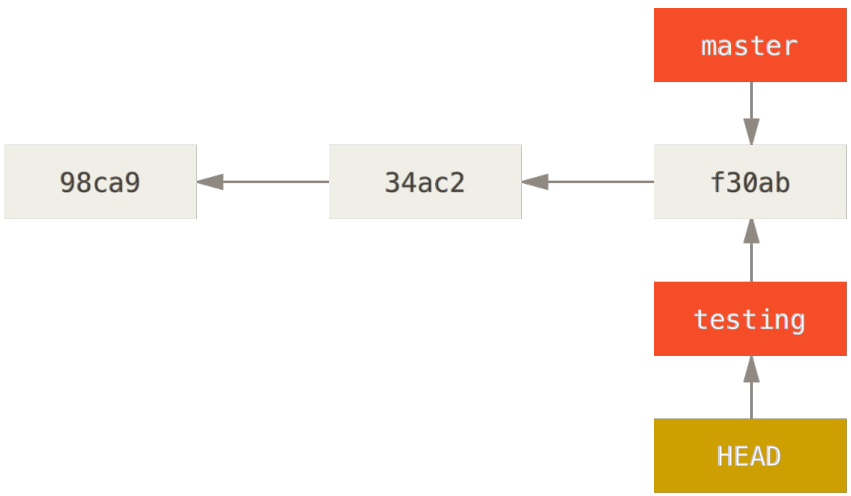
- the HEAD of the branch is just pointed to the last commit
- "master" branch = default name when running "git init"



Merging (2): Branching (continued)

- we can also switch the HEAD and update the working directory with

```
$ git checkout testing
```



Merging (3): Merging

- How to bring branches back together?
  - Merging or Rebasing
- Merging
  - via special "merge commits" with more than one parent
  - they merge changes together using different strategies
- Rebase
  - rewrite (change history) of all commits since the branching point
  - ensure both versions are preserved

Merging (4): How to merge

- fast-forward-merge = merging two HEADs at two different times
  - no conflicts, one just has more commits than the other
  - simply move the merged HEAD to the newest commit
- "Recursive" merge strategy
  - default strategy for non-fast-forward merges
  - if the same line was modified in conflicting ways there will be merge conflicts
- Other strategies (we do not go into details)
  - ours (take "our" version for everything)
  - theirs (take "their" version for everything)
  - octopus (merge more than 2 HEADs)
  - subtree
  - ...
- Merge conflicts can occur
- Git merges would fill a talk on their own

Merging (5): Remotes

- remotes = remote branches
  - we can retrieve commits from them
  - we can push commits to them
- each branch can have an "upstream" branch
  - they are called remote-tracking branches
  - they track the state of the remote
  - of the form remote/branch
  - "origin" is just the default name for the remote
- you can fetch commits from a remote branch
  - "git fetch REMOTE BRANCH"
  - this will update the remote reference , create "FETCH_HEAD" and pull all the commit data
  - afterwards it can be merged into the current branch
  - "git merge FETCH_HEAD"

Merging (6): Remotes continued

- "git pull" does this in one go
  - does some more magic (for example looking up tracked remote)
  - make sure everything is checked out into the working directory
  - sometimes it can be simpler to use "git fetch" and then "git merge"
- we also want to push content to the remote
  - update the server with the changes we made
  - we can use "git push REMOTE BRANCH"
  - each branch can be pushed individually or kept local only
  - only accepts fast-forward pushes
  - can be forced with a "–force" argument
- other remote operations
  - setting the tracking branch ("git branch -u REMOTE/BRANCH")
  - deleting a branch from the remote ("git push REMOTE –delete BRANCH")

Conclusion

- git is very powerful and useful
- git started as a toolkit for a VCS and by now has a more or less user friendly interface
- in order to use it properly it is important to understand the underlying model
- using git should not be only running memorized commands

The end

Thank you for your attention! Any Questions, Comments, etc?

- This talk has been adapted from the Pro Git book (by Scott Chacon and Ben Straub). It is licensed under Creative Commons Attribution Non Commercial Share Alike 3.0 license.