Git internals: object store & refs

Key takeaways

1. Every object lives under its hash in .git/objects; names and content are separate objects.
2. origin/* is a fetch-time cache; git branch -vv reads your sync state against it.

Inside .git/objects

Blobs store content under its own hash; trees and commits are small objects full of hashes

Blob: content in, hash out

```
# store app.py; dir = first 2 hash chars
$ git hash-object -w app.py
c5b8d1e7a9...
$ ls .git/objects/c5/
b8d1e7a9...

# ask the object's type, then its bytes
$ git cat-file -t c5b8d1
blob
$ git cat-file -p c5b8d1
def main():
    run(retries=3)
```

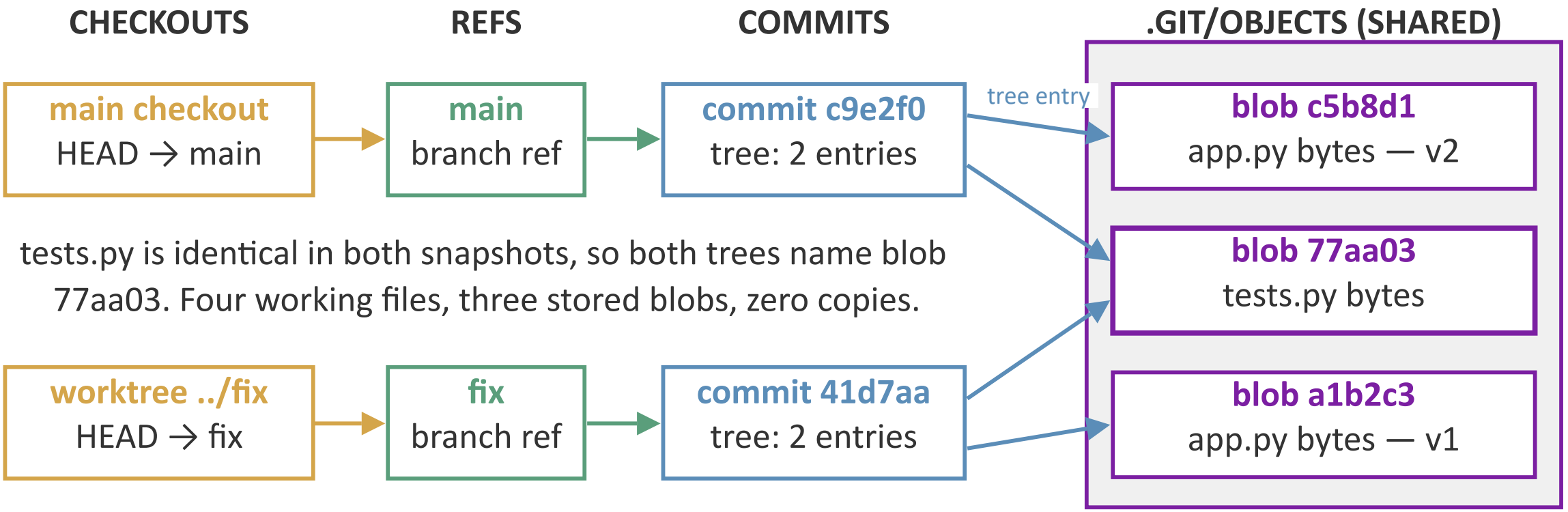
Tree and commit: metadata naming the blobs

```
# HEAD's tree: filenames -> blob hashes
$ git cat-file -p 'HEAD^{tree}'
100644 blob c5b8d1   app.py
100644 blob 77aa03   tests.py

# the commit object itself
$ git cat-file -p HEAD
tree 8f30ab
parent 3e91cc
author Witchel <...> 1755...

Add retry logic
```

The blob has no filename and the tree has no file bytes: names and content are separate objects, all addressed by hash.

One blob store, many references

Branches, commits, and worktrees add hash pointers, never copies of content



tests.py is identical in both snapshots, so both trees name blob 77aa03. Four working files, three stored blobs, zero copies.

Creating a branch or a worktree copies zero blobs: every checkout shares one object store, which is why per-agent worktrees are cheap.

Deleting files does not shrink the repo

A clone fetches all of history; .gitignore before the first commit is the only cheap fix

git rm only edits the newest tree

```
# measure before: model.bin checked in
$ du -sh .git
1.1G    .git

# delete the file in a new commit
$ git rm model.bin
$ git commit -m "Remove training data"

# measure after: nothing got smaller
$ du -sh .git
1.1G    .git
```

The blob stays reachable from history

```
# which commits ever touch model.bin?
$ git log --oneline --all -- model.bin
b7d2e4 Add training data

# the only real fix rewrites history:
# every later commit gets a NEW hash;
# every clone and fork must recover
$ git filter-repo --invert-paths \
      --path model.bin
```

History is append-only: the delete is one more snapshot, and every earlier snapshot keeps the blob. The only shrink is a history rewrite — dangerous, since it renames every later commit.

Watch the refs move

Time flows down: you commit twice, a teammate pushes once, you fetch

time	YOU — LOCAL CLONE	GITHUB — origin	TEAMMATE'S CLONE
everyone in sync	main d4f8	main d4f8	main d4f8
you commit twice	**main e7a1** origin/main d4f8 **— 2 new commits**	main d4f8	main d4f8
teammate commits, pushes	main e7a1 origin/main d4f8 **— you see nothing**	**main 9c1d** — the push landed	**main 9c1d**
you run git fetch	main e7a1 **origin/main 9c1d** **now: ahead 2, behind 1**	main 9c1d	main 9c1d

origin/main is shown only where it disagrees with its clone's main — everywhere else the two are equal.

main is this clone's local branch; origin/main is the last value fetched from GitHub.

Print the branches, read the situation

After the fetch: git branch -vv reads local state; -r lists the origin/* cache

Local branches and their upstreams

```
$ git branch -vv
* main  e7a1 [origin/main: ahead 2, behind 1]
  fix   41d7 [origin/fix: gone]
  spike b3c9

# ahead 2  -> your 2 commits, not pushed
# behind 1 -> the teammate's push
# gone     -> branch was deleted on GitHub
# no [..]  -> never pushed: this clone only
```

Remote-tracking branches in the cache

```
# refresh the origin/* cache first
$ git fetch
$ git branch -r
  origin/main
  origin/fix
  origin/hash-index

# hash-index: your teammate pushed it, but
# this clone has never checked it out;
# switch makes a local branch from origin/*
$ git switch hash-index
```

Each line is a diagnosis: unpushed commits, a branch deleted on GitHub, or remote work this clone has never seen.