Lecture 12: Git Internals and Advanced Features

Git as an Object-Oriented Database

- At its core, Git manages your project history as an object-oriented database.
- Every file, directory, and commit is stored as an object, each with a unique SHA-1 checksum as its identifier.
- This object database persists on disk, ensuring your history is safe even if you lose power or close your editor.
- The structure is similar to object-oriented programming, but the objects are stored in files on disk, not in RAM.
- This design enables Git to efficiently store, retrieve, and share complete project histories, and is a key reason for its power and flexibility.

ASCII Diagram: .git Directory Structure

```
.git/
├── branches/         # (Obsolete) Old branch storage
├── config            # Project-specific Git configuration
├── description       # (Obsolete) Short repo description
├── head              # Points to current branch/commit
├── hooks/            # Scripts for Git events (pre-commit, etc.)
├── info/
│   └── exclude       # Ignore rules not under version control
├── index             # Staging area (binary file)
├── logs/             # History of branch/HEAD moves
├── objects/          # All Git objects (blobs, trees, commits)
├── refs/             # Human-readable names (branches, tags)
```

Table: Git Object Types

Type	Description	Example Content
blob	File content	Source code, text files
tree	Directory structure (points to blobs/trees)	List of files, subdirectories
commit	Snapshot metadata + root tree reference	Author, message, parent, tree

Real-World Scenario: Power of Git's Object Database

Suppose you are collaborating on a software project. You make a series of changes, commit them, and later realize a bug was introduced. With Git's object database, you can use `git bisect` to efficiently find the exact commit that introduced the bug, or use `git checkout` to revert to any previous state. The object database ensures every version is stored efficiently and can be shared or restored at any time, enabling robust collaboration and recovery workflows.

---

1. Rebasing in Git

1.1 What is Rebasing?

- Rebasing is the process of reapplying a series of commits on top of another base tip.
- Used to clean up, reorder, or edit commits before publishing to a shared repository.
- Especially useful for unpublished local commits.

1.2 Interactive Rebase (`git rebase -i`)

- Allows you to edit commit history interactively.
- Command:

  ```
  git rebase -i <commit-id>
  ```

  - `<commit-id>`: The commit before the first one you want to rebase.
- In the editor, you can:
  - Change the order of commits
  - Delete commits
  - Edit commits (with `edit`)
  - Drop commits
- Example edit script:

```
pick abcd123 Add logging
pick efgh456 Fix bug
pick xyzw789 Update Makefile
```

Can be reordered or transformed into:

```
pick efgh456 Fix bug
edit abcd123 Add logging
drop xyzw789 Update Makefile
```

- To continue after resolving conflicts or editing:

```
git rebase --continue
```

- If conflicts arise:
  - Fix the conflict
  - `git add` to stage resolved files
  - Continue with `git rebase --continue`
  - Use `git status` to see the current state

1.3 Merge Conflicts During Rebase

- Changing the order of commits may lead to merge conflicts.
- Resolve them manually, stage, and continue the rebase process.
- The original history is preserved until you finish and force-update the branch.

---

2. Remote Repositories and Distributed Git

2.1 Git as a Distributed VCS

- Git is fundamentally distributed: there is no required central authority.
- Multiple repositories can exist (e.g., in different locations), each with equal status.
- In practice, some repositories (like Linus Torvalds' for Linux) are treated as authoritative, but Git itself does not enforce this.
- Each repository can know about "remotes" (other repositories it can fetch from or push to).

2.2 Viewing and Managing Remotes

- `git remote`: List all remotes
- `git remote -v`: List remotes with fetch and push URLs
- By convention, the default remote after cloning is called `origin`.
- You can add or change remotes:

```
git remote add <name> <url>
git remote set-url <name> <new-url>
```

2.3 Fetch vs Pull

Command	Behavior
`git fetch`	Fetch changes from remote without merging
`git pull`	Fetch + Merge (potentially causing merge conflicts)

- `git fetch` updates your repository's view of the remote branches (e.g., `origin/main`), but does not change your local branches.
- `git pull` is equivalent to `git fetch` followed by `git merge` (or `git rebase`, depending on config).
- Some developers prefer to fetch and rebase manually to avoid unnecessary merge commits.
- Tracking branches (created with `git branch --track <branch> origin/<branch>`) help keep local and remote branches in sync.

Example: Keeping Main in Sync

- Some developers always keep their `main` branch identical to `origin/main` and do development in feature branches, rebasing as needed.

---

3. Stashing

3.1 What is Stashing?

- Stashing temporarily saves work-in-progress changes when you're not ready to commit them.
- Useful if you need to switch context (e.g., fix a bug on another branch) without committing incomplete work.
- Commands:
  - `git stash push`: Save current uncommitted changes
  - `git stash apply`: Reapply the last saved stashed changes
  - `git stash list`: View stashed entries
  - `git stash show`: View contents of the top stash
  - `git stash drop`: Remove a stash entry
  - `git stash pop`: Apply and remove the top stash
- Alternative manual stashing:

```
git diff > patch.diff
git checkout branch
# Later
patch -p1 < patch.diff
```

- Stashing is preferred in workflows where every commit must build and pass tests.

---

4. Bisecting: Finding Bug Introductions

4.1 Git Bisect

- Used to find the commit that introduced a bug between two known good/bad states.
- Uses binary search ($\log_2 N$ complexity) to minimize the number of test runs.
- Commands:

```
git bisect start
git bisect bad           # current or specified bad commit
git bisect good <commit> # known good commit
git bisect run <script>  # automates bisection
```

- Example:

```
git bisect start
git bisect bad
git bisect good v23
git bisect run make check
```

- If a commit can't be tested (e.g., doesn't build):

```
git bisect skip <commit>
```

- Some projects enforce "never break the build" to make bisecting reliable.

---

5. Git Internals: The `.git/` Directory

5.1 Directory Structure

Path	Description
`branches/`	(Obsolete) Old branch storage; kept for backward compatibility
`config`	Project-specific Git configuration
`description`	(Obsolete) Short description of the repository
`head`	Points to the current branch or commit
`hooks/`	Scripts triggered by Git events (e.g., `pre-commit`)

Path	Description
`info/exclude`	Ignore rules not under version control
`index`	Staging area between working directory and the repository
`logs/`	History of branch/HEAD move events
`objects/`	Storage for all Git objects (blobs, trees, commits)
`refs/`	Human-readable names (branches, tags) mapped to SHA1

- Many files and directories are for backward compatibility or are rarely used (e.g., `branches/`, `description`).
- `hooks/` contains sample scripts; to activate, rename and customize as needed.
- `index` is a binary file used for efficient staging.
- `info/exclude` is like `.gitignore` but not under version control.
- `logs/` records the history of branch pointers (e.g., for recovering from mistakes).

5.2 Object Database

- Git stores all content as objects in `.git/objects/`.
- Objects are named by their SHA1 checksum, split into a two-character directory and a 38-character filename for efficiency.
- Early versions used flat storage, but this was slow for large repos.
- Objects are compressed binary files.
- There are also packed objects (for efficiency in large repos), but these are more advanced.

Object Types

Type	Description
blob	File content
tree	Directory structure (refers to blobs/trees)
commit	Snapshot metadata + reference to root tree

Viewing and Creating Objects

- Get type:

```
git cat-file -t <sha>
```

- Get content:

```
git cat-file -p <sha>
```

- Create object:

```
echo "hello" | git hash-object -w --stdin
```

- Example:

```
echo "Arma virumque cano" > aeneid.txt
git hash-object -w aeneid.txt
```

Trees

- Trees represent directories; entries contain:
  - Name (filename or subdirectory)
  - Type (`blob` or `tree`)
  - Mode (file permissions, e.g., 100644, 040000)
  - SHA1 (points to blob or subtree)
- Example:

```
git update-index --add --cacheinfo 100644 <blob-sha> filename.txt
git write-tree
```

Commits

- Commits link a tree and metadata about the snapshot.
- Contents:

```
tree <tree-sha>
parent <parent-commit-sha>
author <name> <email> <timestamp>
committer <name> <email> <timestamp>

<commit message>
```

- Create manually:

```
git commit-tree <tree-sha> -m "message"
```

---

6. Porcelain vs Plumbing

Category	Description	Examples
Porcelain	User-facing commands for day-to-day use	`git add`, `git commit`, `git push`
Plumbing	Low-level commands for scripting or development	`git hash-object`, `git cat-file`, `git write-tree`

- Porcelain commands are for everyday use; plumbing commands are for scripting, debugging, or extending Git.
- Example plumbing commands:
  - `git hash-object`: Create a new object from a file or input
  - `git cat-file`: Show type or contents of an object
  - `git write-tree`: Create a tree object from the index
  - `git commit-tree`: Create a commit object from a tree

---

7. Summary

This lecture provided an in-depth exploration of advanced Git features and internals. Key topics included interactive rebasing, distributed nature of Git, remotes and synchronization, stashing, bisecting, and a tour through the internal structure and types within the `.git/` directory. It introduced Git's core concept of representing file changes as a graph of SHA1-addressed objects, and distinguished between user-facing "porcelain" commands and low-level "plumbing" commands that interact directly with Git's object database.

Appendix: Useful Visualizations

Distributed Git Repositories (Example Topology)

```
Repo1 <----> Repo2 <----> Repo3
   ^                      |
   |                      v
  Repo0 <--------------- Repo4
```

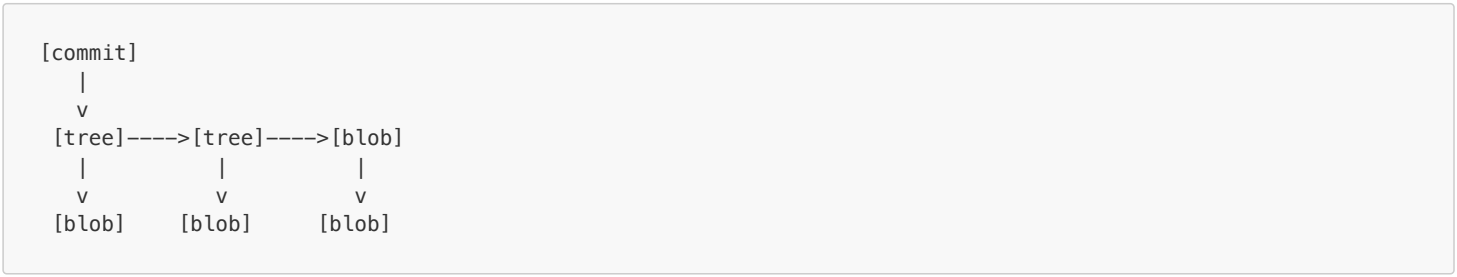
- Each repo can fetch/push from/to others; no single "master" required.

.git Directory Structure (Simplified)

```
.git/
├── branches/      # (obsolete)
├── config         # repo config
├── description    # (obsolete)
├── head           # current branch pointer
├── hooks/         # event scripts
├── info/exclude   # ignore rules
```

```
├── index          # staging area
├── logs/          # branch/HEAD history
├── objects/       # all objects (blobs, trees, commits)
├── refs/          # branch/tag names
```

Git Object Relationships



- Commits point to trees (directories), which point to blobs (files) and other trees (subdirectories).