Git

Susan Potter

6.1. Git in a Nutshell

Git enables the maintenance of a digital body of work (often, but not limited to, code) by many collaborators using a peer-to-peer network of repositories. It supports distributed workflows, allowing a body of work to either eventually converge or temporarily diverge.

This chapter will show how various aspects of Git work under the covers to enable this, and how it differs from other version control systems (VCSs).

6.2. Git's Origin

To understand Git's design philosophy better it is helpful to understand the circumstances in which the Git project was started in the Linux Kernel Community.

The Linux kernel was unusual, compared to most commercial software projects at that time, because of the large number of committers and the high variance of contributor involvement and knowledge of the existing codebase. The kernel had been maintained via tarballs and patches for years, and the core development community struggled to find a VCS that satisfied most of their needs.

Git is an open source project that was born out of those needs and frustrations in 2005. At that time the Linux kernel codebase was managed across two VCSs, BitKeeper and CVS, by different core developers. BitKeeper offered a different view of VCS history lineage than that offered by the popular open source VCSs at this time.

Days after BitMover, the maker of BitKeeper, announced it would revoke the licenses of some core Linux kernel developers, Linus Torvalds began development, in haste, of what was to become Git. He began by writing a collection of scripts to help him manage email patches to apply one after the other. The aim of this initial collection of scripts was to be able to abort merges quickly so the maintainer could modify the codebase mid-patch-stream to manually merge, then continue merging subsequent patches.

From the outset, Torvalds had one philosophical goal for Git—to be the anti-CVS—plus three usability design goals:

- Support distributed workflows similar to those enabled by BitKeeper
- Offer safeguards against content corruption
- Offer high performance

These design goals have been accomplished and maintained, to a degree, as I will attempt to show by dissecting Git's use of directed acyclic graphs (DAGs) for content storage, reference pointers for heads, object model representation, and remote protocol; and finally how Git tracks the merging of trees.

Despite BitKeeper influencing the original design of Git, it is implemented in fundamentally different ways and allows even more distributed plus local-only workflows, which were not possible with BitKeeper. Monotone, an open source distributed VCS started in 2003, was likely another inspiration during Git's early development.

Distributed version control systems offer great workflow flexibility, often at the expense of simplicity. Specific benefits of a distributed model include:

- Providing the ability for collaborators to work offline and commit incrementally.
- Allowing a collaborator to determine when his/her work is ready to share.
- Offering the collaborator access to the repository history when offline.
- Allowing the managed work to be published to multiple repositories, potentially with different branches or granularity of changes visible.

Around the time the Git project started, three other open source distributed VCS projects were initiated. (One of them, Mercurial, is discussed in Volume 1 of *The Architecture of Open Source Applications*.) All of these dVCS tools offer slightly different ways to enable highly flexible workflows, which centralized VCSs before them were not capable of handling directly. Note: Subversion has an extension named SVK maintained by different developers to support server-to-server synchronization.

Today popular and actively maintained open source dVCS projects include Bazaar, Darcs, Fossil, Git, Mercurial, and Veracity.

6.3. Version Control System Design

Now is a good time to take a step back and look at the alternative VCS solutions to Git. Understanding their differences will allow us to explore the architectural choices faced while developing Git.

A version control system usually has three core functional requirements, namely:

- Storing content
- Tracking changes to the content (history including merge metadata)
- Distributing the content and history with collaborators

Note: The third requirement above is not a functional requirement for all VCSs.

Content Storage

The most common design choices for storing content in the VCS world are with a delta-based changeset, or with directed acyclic graph (DAG) content representation.

Delta-based changesets encapsulate the differences between two versions of the flattened content, plus some metadata. Representing content as a directed acyclic graph involves objects forming a hierarchy which mirrors the content's filesystem tree as a snapshot of the commit (reusing the unchanged objects inside the tree where possible). Git stores content as a directed acyclic graph using different types of objects. The "Object Database" section later in this chapter describes the different types of objects that can form DAGs inside the Git repository.

Commit and Merge Histories

On the history and change-tracking front most VCS software uses one of the following approaches:

- Linear history
- Directed acyclic graph for history

Again Git uses a DAG, this time to store its history. Each commit contains metadata about its ancestors; a commit in Git can have zero or many (theoretically unlimited) parent commits. For example, the first commit in a Git repository would have zero parents, while the result of a three-way merge would have three parents.

Another primary difference between Git and Subversion and its linear history ancestors is its ability to directly support branching that will record most merge history cases.

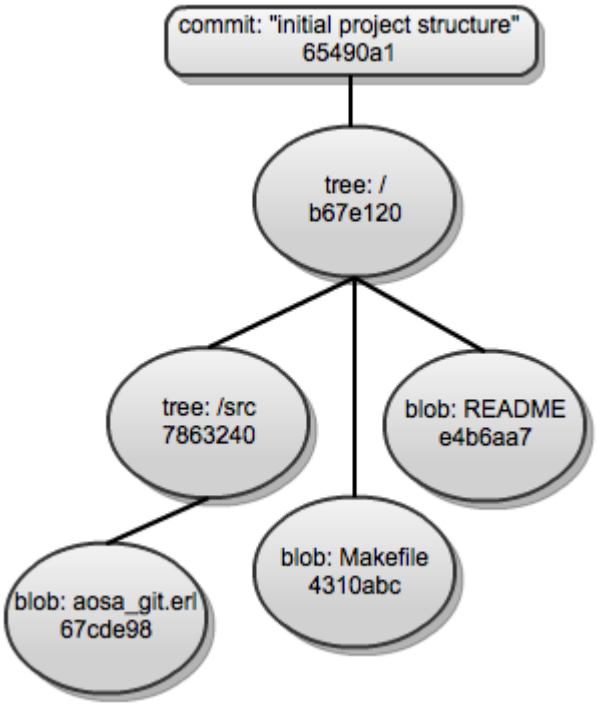

Figure 6.1: Example of a DAG representation in Git

Git enables full branching capability using directed acyclic graphs to store content. The history of a file is linked all the way up its directory structure (via nodes representing directories) to the root directory, which is then linked to a commit node. This commit node, in turn, can have one or more parents. This affords Git two properties that allow us to reason about history and content in more definite ways than the family of VCSs derived from RCS do, namely:

- When a content (i.e., file or directory) node in the graph has the same reference identity (the SHA in Git) as that in a different commit, the two nodes are guaranteed to contain the same content, allowing Git to short-circuit content diffing efficiently.
- When merging two branches we are merging the content of two nodes in a DAG. The DAG allows Git to "efficiently" (as compared to the RCS family of VCS) determine common ancestors.

Distribution

VCS solutions have handled content distribution of a working copy to collaborators on a project in one of three ways:

- Local-only: for VCS solutions that do not have the third functional requirement above.
- Central server: where all changes to the repository must transact via one specific repository for it to be recorded in history at all.
- Distributed model: where there will often be publicly accessible repositories for collaborators to "push" to, but commits can be made locally and pushed to these public nodes later, allowing offline work.

To demonstrate the benefits and limitations of each major design choice, we will consider a Subversion repository and a Git repository (on a server), with equivalent content (i.e., the HEAD of the default branch in the Git repository has the same content as the Subversion repository's latest revision on trunk). A developer, named Alex, has a local checkout of the Subversion repository and a local clone of the Git repository.

Let us say Alex makes a change to a 1 MB file in the local Subversion checkout, then commits the change. Locally, the checkout of the file mimics the latest change and local metadata is updated. During Alex's commit in the centralized Subversion repository, a diff is generated between the previous snapshot of the files and the new changes, and this diff is stored in the repository.

Contrast this with the way Git works. When Alex makes the same modification to the equivalent file in the local Git clone, the change will be recorded locally first, then Alex can "push" the local pending commits

to a public repository so the work can be shared with other collaborators on the project. The content changes are stored identically for each Git repository that the commit exists in. Upon the local commit (the simplest case), the local Git repository will create a new object representing a file for the changed file (with all its content inside). For each directory above the changed file (plus the repository root directory), a new tree object is created with a new identifier. A DAG is created starting from the newly created root tree object pointing to blobs (reusing existing blob references where the files content has not changed in this commit) and referencing the newly created blob in place of that file's previous blob object in the previous tree hierarchy. (A *blob* represents a file stored in the repository.)

At this point the commit is still local to the current Git clone on Alex's local device. When Alex "pushes" the commit to a publicly accessible Git repository this commit gets sent to that repository. After the public repository verifies that the commit can apply to the branch, the same objects are stored in the public repository as were originally created in the local Git repository.

There are a lot more moving parts in the Git scenario, both under the covers and for the user, requiring them to explicitly express intent to share changes with the remote repository separately from tracking the change as a commit locally. However, both levels of added complexity offer the team greater flexibility in terms of their workflow and publishing capabilities, as described in the "Git's Origin" section above.

In the Subversion scenario, the collaborator did not have to remember to push to the public remote repository when ready for others to view the changes made. When a small modification to a larger file is sent to the central Subversion repository the delta stored is much more efficient than storing the complete file contents for each version. However, as we will see later, there is a workaround for this that Git takes advantage of in certain scenarios.

6.4. The Toolkit

Today the Git ecosystem includes many command-line and UI tools on a number of operating systems (including Windows, which was originally barely supported). Most of these tools are mostly built on top of the Git core toolkit.

Due to the way Git was originally written by Linus, and its inception within the Linux community, it was written with a toolkit design philosophy very much in the Unix tradition of command line tools.

The Git toolkit is divided into two parts: the plumbing and the porcelain. The plumbing consists of low-level commands that enable basic content tracking and the manipulation of directed acyclic graphs (DAG). The porcelain is the smaller subset of `git` commands that most Git end users are likely to need to use for maintaining repositories and communicating between repositories for collaboration.

While the toolkit design has provided enough commands to offer fine-grained access to functionality for many scripters, application developers complained about the lack of a linkable library for Git. Since the Git binary calls `die()`, it is not reentrant and GUIs, web interfaces or longer running services would have to fork/exec a call to the Git binary, which can be slow.

Work is being done to improve the situation for application developers; see the "Current And Future Work" section for more information.

6.5. The Repository, Index and Working Areas

Let's get our hands dirty and dive into using Git locally, if only to understand a few fundamental concepts.

First to create a new initialized Git repository on our local filesystem (using a Unix inspired operating system) we can do:

```
$ mkdir testgit
$ cd testgit
$ git init
```

Now we have an empty, but initialized, Git repository sitting in our testgit directory. We can branch, commit, tag and even communicate with other local and remote Git repositories. Even communication with other types of VCS repositories is possible with just a handful of `git` commands.

The `git init` command creates a .git subdirectory inside of testgit. Let's have a peek inside it:

```
tree .git/
.git/
|-- HEAD
|-- config
|-- description
|-- hooks
|   |-- applypatch-msg.sample
|   |-- commit-msg.sample
|   |-- post-commit.sample
|   |-- post-receive.sample
|   |-- post-update.sample
|   |-- pre-applypatch.sample
|   |-- pre-commit.sample
|   |-- pre-rebase.sample
|   |-- prepare-commit-msg.sample
|   |-- update.sample
|-- info
|   |-- exclude
|-- objects
|   |-- info
|   |-- pack
|-- refs
    |-- heads
    |-- tags
```

The `.git` directory above is, by default, a subdirectory of the root working directory, `testgit`. It contains a few different types of files and directories:

- *Configuration*: the `.git/config`, `.git/description` and `.git/info/exclude` files essentially help configure the local repository.
- *Hooks*: the `.git/hooks` directory contains scripts that can be run on certain lifecycle events of the repository.
- *Staging Area*: the `.git/index` file (which is not yet present in our tree listing above) will provide a staging area for our working directory.
- *Object Database*: the `.git/objects` directory is the default Git object database, which contains all content or pointers to local content. All objects are immutable once created.
- *References*: the `.git/refs` directory is the default location for storing reference pointers for both local and remote branches, tags and heads. A reference is a pointer to an object, usually of type `tag` or `commit`. References are managed outside of the Object Database to allow the references to change where they point to as the repository evolves. Special cases of references may point to other references, e.g. `HEAD`.

The `.git` directory is the actual repository. The directory that contains the working set of files is the *working directory*, which is typically the parent of the `.git` directory (or *repository*). If you were creating a Git remote repository that would not have a working directory, you could initialize it using the `git init --bare` command. This would create just the pared-down repository files at the root, instead of creating the repository as a subdirectory under the working tree.

Another file of great importance is the *Git index*: `.git/index`. It provides the staging area between the local working directory and the local repository. The index is used to stage specific changes within one file (or more), to be committed all together. Even if you make changes related to various types of features, the commits can be made with like changes together, to more logically describe them in the commit message. To selectively stage specific changes in a file or set of files you can using `git add -p`.

The Git index, by default, is stored as a single file inside the repository directory. The paths to these three areas can be customized using environment variables.

It is helpful to understand the interactions that take place between these three areas (the repository, index and working areas) during the execution of a few core Git commands:

- `git checkout [branch]`
  This will move the HEAD reference of the local repository to branch reference path (e.g. `refs/heads/master`), populate the index with this head data and refresh the working directory to represent the tree at that head.
- `git add [files]`
  This will cross reference the checksums of the *files* specified with the corresponding entries in the Git index to see if the index for staged files needs updating with the working directory's version. Nothing changes in the Git directory (or repository).

Let us explore what this means more concretely by inspecting the contents of files under the `.git` directory (or repository).

```
$ GIT_DIR=$PWD/.git
$ cat $GIT_DIR/HEAD

ref: refs/heads/master

$ MY_CURRENT_BRANCH=$(cat .git/HEAD | sed 's/ref: //g')
$ cat $GIT_DIR/$MY_CURRENT_BRANCH

cat: .git/refs/heads/master: No such file or directory
```

We get an error because, before making any commits to a Git repository at all, no branches exist except the default branch in Git which is `master`, whether it exists yet or not.

Now if we make a new commit, the master branch is created by default for this commit. Let us do this (continuing in the same shell, retaining history and context):

```
$ git commit -m "Initial empty commit" --allow-empty
$ git branch

* master

$ cat $GIT_DIR/$MY_CURRENT_BRANCH

3bce5b130b17b7ce2f98d17b2998e32b1bc29d68

$ git cat-file -p $(cat $GIT_DIR/$MY_CURRENT_BRANCH)
```

What we are starting to see here is the content representation inside Git's object database.

6.6. The Object Database

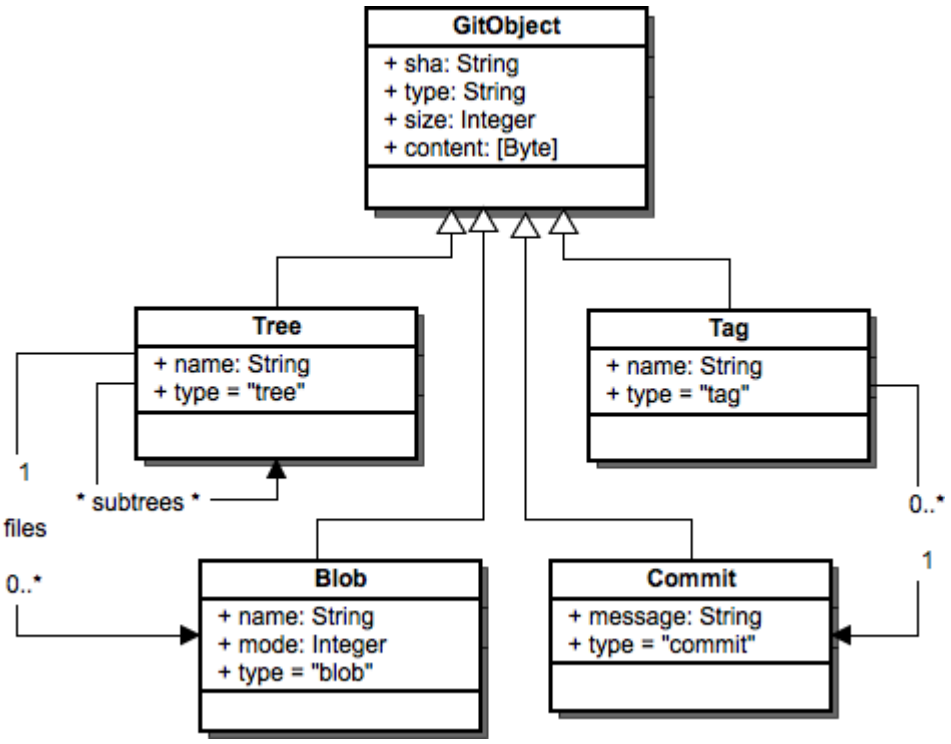

Figure 6.2: Git objects

Git has four basic primitive objects that every type of content in the local repository is built around. Each object type has the following attributes: *type*, *size* and *content*. The primitive object types are:

- *Tree*: an element in a tree can be another tree or a blob, when representing a content directory.
- *Blob*: a blob represents a file stored in the repository.
- *Commit*: a commit points to a tree representing the top-level directory for that commit as well as parent commits and standard attributes.
- *Tag*: a tag has a name and points to a commit at the point in the repository history that the tag represents.

All object primitives are referenced by a SHA, a 40-digit object identity, which has the following properties:

- If two objects are identical they will have the same SHA.
- if two objects are different they will have different SHAs.
- If an object was only copied partially or another form of data corruption occurred, recalculating the SHA of the current object will identify such corruption.

The first two properties of the SHA, relating to identity of the objects, is most useful in enabling Git's distributed model (the second goal of Git). The latter property enables some safeguards against corruption (the third goal of Git).

Despite the desirable results of using DAG-based storage for content storage and merge histories, for many repositories delta storage will be more space-efficient than using *loose* DAG objects.

6.7. Storage and Compression Techniques

Git tackles the storage space problem by packing objects in a compressed format, using an index file which points to offsets to locate specific objects in the corresponding *packed* file.

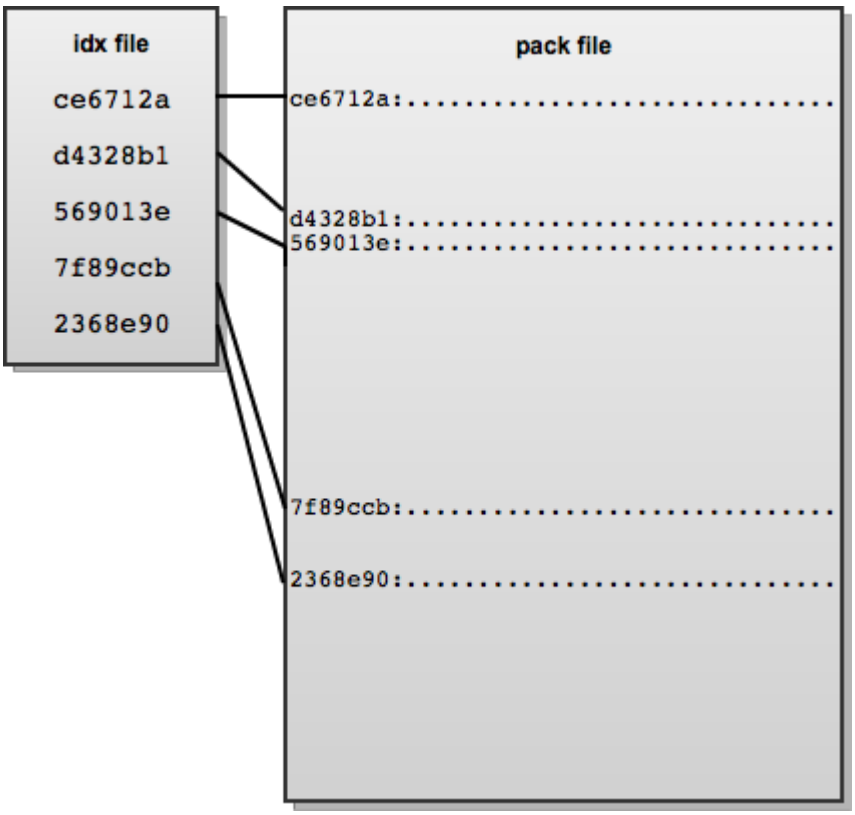

Figure 6.3: Diagram of a pack file with corresponding index file

We can count the number of loose (or unpacked) objects in the local Git repository using `git count-objects`. Now we can have Git pack loose objects in the object database, remove loose objects already packed, and find redundant pack files with Git plumbing commands if desired.

The pack file format in Git has evolved, with the initial format storing CRC checksums for the pack file and index file in the index file itself. However, this meant there was the possibility of undetectable corruption in the compressed data since the repacking phase did not involve any further checks. Version 2 of the pack file format overcomes this problem by including the CRC checksums of each compressed object in the pack index file. Version 2 also allows packfiles larger than 4 GB, which the initial format did not support. As a way to quickly detect pack file corruption the end of the pack file contains a 20-byte SHA1 sum of the ordered list of all the SHAs in that file. The emphasis of the newer pack file format is on helping fulfill Git's second usability design goal of safeguarding against data corruption.

For remote communication Git calculates the commits and content that need to be sent over the wire to synchronize repositories (or just a branch), and generates the pack file format on the fly to send back using the desired protocol of the client.

6.8. Merge Histories

As mentioned previously, Git differs fundamentally in merge history approach than the RCS family of VCSs. Subversion, for example, represents file or tree history in a linear progression; whatever has a higher revision number will supercede anything before it. Branching is not supported directly, only through an unenforced directory structure within the repository.

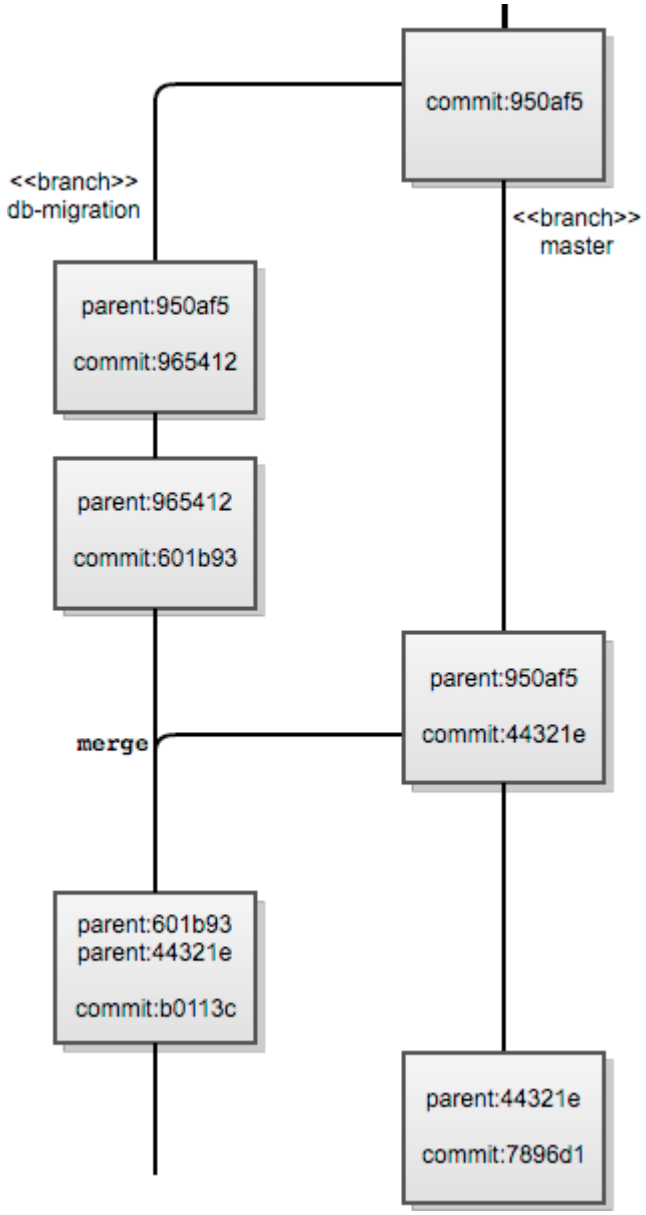

Figure 6.4: Diagram showing merge history lineage

Let us first use an example to show how this can be problematic when maintaining multiple branches of a work. Then we will look at a scenario to show its limitations.

When working on a "branch" in Subversion at the typical root `branches/branch-name`, we are working on directory subtree adjacent to the `trunk` (typically where the live or *master* equivalent code resides within). Let us say this branch is to represent parallel development of the `trunk` tree.

For example, we might be rewriting a codebase to use a different database. Part of the way through our rewrite we wish to merge in upstream changes from another branch subtree (not trunk). We merge in these changes, manually if necessary, and proceed with our rewrite. Later that day we finish our database vendor migration code changes on our `branches/branch-name` branch and merge our changes into `trunk`. The problem with the way linear-history VCSs like Subversion handle this is that there is no way to know that the changesets from the other branch are now contained within the trunk.

DAG-based merge history VCSs, like Git, handle this case reasonably well. Assuming the other branch does not contain commits that have not been merged into our database vendor migration branch (say, `db-migration` in our Git repository), we can determine—from the commit object parent relationships—that a commit on the `db-migration` branch contained the *tip* (or HEAD) of the other upstream branch. Note that a commit object can have zero or more (bounded by only the abilities of the merger)

parents. Therefore the merge commit on the `db-migration` branch *knows* it merged in the current HEAD of the current branch and the HEAD of the other upstream branch through the SHA hashes of the parents. The same is true of the merge commit in the `master` (the `trunk` equivalent in Git).

A question that is hard to answer definitively using DAG-based (and linear-based) merge histories is which commits are contained within each branch. For example, in the above scenario we assumed we merged into each branch all the changes from both branches. This may not be the case.

For simpler cases Git has the ability to cherry pick commits from other branches in to the current branch, assuming the commit can cleanly be applied to the branch.

6.9. What's Next?

As mentioned previously, Git core as we know it today is based on a toolkit design philosophy from the Unix world, which is very handy for scripting but less useful for embedding inside or linking with longer running applications or services. While there is Git support in many popular Integrated Development Environments today, adding this support and maintaining it has been more challenging than integrating support for VCSs that provide an easy-to-link-and-share library for multiple platforms.

To combat this, Shawn Pearce (of Google's Open Source Programs Office) spearheaded an effort to create a linkable Git library with more permissive licensing that did not inhibit use of the library. This was called libgit2. It did not find much traction until a student named Vincent Marti chose it for his Google Summer of Code project last year. Since then Vincent and Github engineers have continued contributing to the libgit2 project, and created bindings for numerous other popular languages such as Ruby, Python, PHP, .NET languages, Lua, and Objective-C.

Shawn Pearce also started a BSD-licensed pure Java library called JGit that supports many common operations on Git repositories. It is now maintained by the Eclipse Foundation for use in the Eclipse IDE Git integration.

Other interesting and experimental open source endeavours outside of the Git core project are a number of implementations using alternative datastores as backends for the Git object database such as:

- jgit_cassandra, which offers Git object persistence using Apache Cassandra, a hybrid datastore using Dynamo-style distribution with BigTable column family data model semantics.
- jgit_hbase, which enables read and write operations to Git objects stored in HBase, a distributed key-value datastore.
- libgit2-backends, which emerged from the libgit2 effort to create Git object database backends for multiple popular datastores such as Memcached, Redis, SQLite, and MySQL.

All of these open source projects are maintained independently of the Git core project.

As you can see, today there are a large number of ways to use the Git format. The face of Git is no longer just the toolkit command line interface of the Git Core project; rather it is the repository format and protocol to share between repositories.

As of this writing, most of these projects, according to their developers, have not reached a stable release, so work in the area still needs to be done but the future of Git appears bright.

6.10. Lessons Learned

In software, every design decision is ultimately a trade-off. As a power user of Git for version control and as someone who has developed software around the Git object database model, I have a deep fondness for Git in its present form. Therefore, these lessons learned are more of a reflection of common recurring complaints about Git that are due to design decisions and focus of the Git core developers.

One of the most common complaints by developers and managers who evaluate Git has been the lack of IDE integration on par with other VCS tools. The toolkit design of Git has made this more challenging than integrating other modern VCS tools into IDEs and related tools.

Earlier in Git's history some of the commands were implemented as shell scripts. These shell script command implementations made Git less portable, especially to Windows. I am sure the Git core developers did not lose sleep over this fact, but it has negatively impacted adoption of Git in larger organizations due to portability issues that were prevalent in the early days of Git's development. Today a project named Git for Windows has been started by volunteers to ensure new versions of Git are ported to Windows in a timely manner.

An indirect consequence of designing Git around a toolkit design with a lot of plumbing commands is that new users get lost quickly; from confusion about all the available subcommands to not being able to understand error messages because a low level plumbing task failed, there are many places for new users to go astray. This has made adopting Git harder for some developer teams.

Even with these complaints about Git, I am excited about the possibilities of future development on the Git Core project, plus all the related open source projects that have been launched from it.

Back to top
Back to *The Architecture of Open Source Applications*.