Git Virtual File System Design History

02/20/2018 • 14 minutes to read

In this article

By: Saeed Noursalehi

GVFS virtualizes the file system beneath a Git repo to solve two main problems:

- Only download contents that the user needs
- Make local Git commands consider just the files that the user cares about, and not all the files that exist in the working directory

Our target use case is the Windows repo, which has over 3 million files in the working directory, totalling 270GB of source files. That's 270GB in the working directory, at the tip of master. To clone this repo, you would have to download a packfile that is about 100GB in size, which would take hours. And once you've succeeded in cloning it, local git operations like checkout (3 hours), status (8 minutes), and commit (30 minutes) would still take way too long to run, because all of those commands are linear on the number of files. Even with those challenges, we still wanted to move the Windows codebase to git, and allow developers to have a reasonable experience. At the same time, we wanted to make as few changes to git as possible, because one of the main advantages of moving to git is that it is a well-known tool, with lots of collective knowledge about how to use it.

Before we decided to build GVFS (hint: building a file system is hard), we explored a lot of other options. We'll cover how GVFS works in detail in an upcoming article, but for now, we'll discuss the other options we considered and why we landed on building a virtual file system.

Background

Why one repo?!

Let's tackle the first, most obvious question. Why does anyone need a repo this big?! Just make your repos smaller, and life will be so much better! Right?!

It's not quite so simple. Others have already documented the value of monorepos (plenty to read at).

At Microsoft, several large teams have gone down the path of breaking their code up into smaller repos, and many have decided that one large repo is better for them.

Splitting up a large codebase is no small task, but it's also not a panacea. Smaller repos would mean simpler scale challenges in each individual repo, but they also make it much more difficult to do cross-cutting changes that have to touch the code in all of those repos. Multiple repos also make your release process more complicated. In the end, if the scale issues can be solved, having one repo often makes the overall workflows far simpler.

The Azure DevOps product comprises several different services that all work together, and when we first moved to Git, we put each of those services in their own repo. Our reasoning was that there are fewer scale issues to deal with in any one repo, and having separate repos also helps enforce some code boundaries. In practice, however, the boundaries created more pain than benefit.

For one thing, many developers would have to make cross-cutting changes across repo boundaries. Managing those dependencies was painful, and ordering the commits and pull requests correctly was very inefficient. We had to build a lot of tooling to deal with those pain points, but those tools then had issues of their own, and our developers were very frustrated.

Secondly, our release process became much more complicated. While the Azure DevOps service deploys once every 3 weeks and each service can deploy mostly independently, we also ship a box product, Team Foundation Server, once a quarter. TFS is built out of the same codebase, and it has an added constraint that all of those services must be installed on the same machine and therefore must all be compatible with each other, e.g. they all need to agree on the versions of their shared dependencies. Allowing the codebases to diverge for three months and then trying to bring them back together for each TFS release proved to be very expensive.

In the end, we realized that putting all of the code in one repo would make our day to day development workflows much simpler. It also allowed us to enforce constraints such as: all services must depend on the same version of a dependency, and anyone who wants to update one of those components must also update their consumers. This creates a little more work up front, but saves a lot of work for each quarterly release. Of course this meant that we had to invest in improving our build tooling to enforce those sorts of things, and we also had to do more work to create strong code boundaries and avoid a mess of dependencies across all the various services.

Windows

The Windows team went through a similar thought process when planning their move to Git. There are several components of Windows that could feasibly live in their own separate repos, and for a while, that was the plan of record. The Windows codebase would live in a dozen or so repos. However, there were two problems with this plan. First of all, while most of those repos would be small enough to use with existing tools, at least one of the repos (OneCore) needed to be around 100GB in size, so we still couldn't get away from solving the scale issues for a large repo. And secondly, we were creating that same problem that cross-cutting changes are difficult to do, and one of the goals of moving Windows to one repo is to make it easier to do major refactorings, not more complicated.

Our design philosophy

Which brings us to our philosophy as tool developers: Tools should enable you to make the right choices for your codebase. If your code would be healthier and your team would be more productive by working in smaller repos, then the tools should make that possible and enable efficient workflows in that environment. And if your team would be more productive in a larger repo, the *tools* should not be the reason that you have to split your repo up into smaller chunks.

Alternatives considered

So we've been working on this question of: how do we enable teams to work with very large codebases in Git? We've gone through quite an evolution of design ideas over the last few years, and we'll talk about some of those ideas here.

Submodules

The first major attempt that we made to solve this problem was to use the existing concept of git submodules. Git allows one repo to reference another repo, so that when you checkout a commit in the parent repo, that commit also specifies which commit to checkout in each sub repo and where to place it in the working directory of the parent. At first glance, this appears like an ideal way to split up a large repo into lots of smaller ones. We certainly thought so, and we spent a few months working on a tool that would smooth over the command line experience of managing submodules.

However, submodules were really designed for the scenario where one repo is *consuming* code from another repo. In that model, submodules work really well, and the submodule is the moral equivalent of a NuGet or npm package. It's a library or a component that is independent of the parent repo, and the parent repo is mostly just a consumer. In cases where the parent needs to make some changes in the submodule, that can be done by making changes in that sub repo, going through its release process (after all, it's an independent library with its own code review, test, and release requirements), and then once those changes are accepted, update the parent repo to consume this new commit.

Our thought was, why not extend this concept a bit. We figured we could take a large codebase and split it apart, roughly speaking by taking each top-level folder and putting it into its own repo, and then create one super-repo that stitched all the subrepos back together. Then we created a tool that would allow you to do things like run "git status" across all the repos, make commits across multiple repos, push those changes, etc.

In the end, we dropped that approach, because it created nearly as many problems as it fixed. For one, we found that we were complicating people's workflows because now every single commit actually had to become *at least* two commits, if not more, due to the need to update the super repo to reference whatever child repo(s) you were modifying. Second, it's not really possible to do atomic commits or pushes across multiple repos – you can make it happen by forcing all pushes to go through a single bottleneck on a server, but that again creates lots of painful scenarios for developers. And third, most developers are not interested in becoming version control experts, they're interested in doing their day jobs and the tool just needs to work for them. Asking every developer to become a DAG expert in order to work with git is a hard enough challenge, but with this super-repo approach, we were now asking every developer to work with a collection of loosely coupled DAGs, and to keep all of that straight and never checkout/commit/push things out of order. It was just too much to ask.

Multiple repos, stitched together

Submodules didn't quite work out, but we still thought maybe we can create multiple repos and use some custom tooling to stitch them together. Android has been using this approach with repo.py, so we figured it was worth a look. And again, we found that the tradeoffs didn't work out well for us. Now that each repo is independent, the workflows are simplified when you're working inside one of those repos, but they're even more complicated if you need to work across repo boundaries. And release management is now a major pain, since there's nothing in your version control history itself that tells you what commit id's from the various repos should be brought together for a given build. So that means that you now need to build another versioning system for releases.

The major lesson learned from the above two approaches is: you should not slice up your repos into a smaller granularity than what you build and ship.

Git alternates

In the theme of trying to reuse existing features, git has a concept of an alternate object store. Whenever git needs to find a commit, tree, or blob object, it looks for loose objects in the .git\objects folder, pack files in

the .git\objects\pack folder, and if they're configured, it will also look in alternate paths for more loose objects and pack files.

So we went down the path of using alternates that point at network shares to avoid the need for copying all blobs from the server to the client when you clone or fetch. This approach sort of worked, but not efficiently, and it only addresses the problem of too much historical content, not the problems of too many files in the working directory and too many entries in the index.

This was another example where we were trying to use a feature for something other than what it was designed for, and the mismatch caused side effects that we couldn't live with. Alternates were designed for the use case where you've already cloned a repo once, and rather than clone it a second time and duplicate all of the objects, you can configure your second clone to share the object store from the first clone. Since those objects and packfiles are also assumed to be local, they can be accessed quickly, there's no need for a second layer of caching, and random-access IO in the .pack and .idx files is not a problem. All of those assumptions are violated when you point the alternate at a network share, and the performance really suffers as a result.

Shallow clones

Git has an existing feature to only download a limited number of commits, rather than all commits in history. However, this doesn't really help a repo as big as Windows, since any one commit and its associated trees and blobs come to over 80 GB in size. And given that most people do not need all of the contents of even a single commit in order to do their daily work, we needed a way to reduce that size. And like alternates, shallow clones don't help at all with the issue of just having too many files in the working directory.

Sparse-checkout

By default, when you checkout a commit, git will place all of the files referenced by that commit into your working directory. However, you can configure the .git\info\sparse-checkout file with a list of files, folders, and patterns, to tell git to only write a subset of the files. This was very promising for us, because most developers only need a small subset of the files to do their work. The major limitations of this feature are:

- The sparse-checkout only affects the working directory, but not the index. So even if you only need 50K files in your working directory, the index still has all 3M entries
- The sparse-checkout is static. So if you've configured it to include folder A and someone later adds a dependency on folder B that you have not included, your builds will start to break until you can figure out that you need to update your sparse-checkout paths
- The sparse-checkout does not affect what is downloaded when you clone or fetch, so even though you don't plan to checkout 95% of the repo, you still have to download it
- The UX of sparse-checkouts is somewhat difficult to use

All of that said, sparse-checkouts became a key part of our solution, as we'll discuss later.

LFS

When large, undiffable files are checked into a git repo, every modification creates a new copy of that file in the history and results in a lot of bloat. Git-LFS helps with this problem by rewriting the large blobs with text pointers to the content, and the actual content is then stored in a separate store. When you clone a repo, you only have to download the tiny pointer files, and then LFS will download the contents just for the versions of those large files that you actually checkout. The Windows team spent a fair bit of effort on making LFS work for them. This approach works well to reduce the size of a repo so they got some benefit from that, but unfortunately it does not reduce the number of files or the size of the index, which means it doesn't help at all

with the long runtimes of local commands. Therefore this approach still resulted in a repo that was not all that usable.

Virtual file system

From all of the above experiments, we learned:

- We really do need to have a single, large repo
- Most developers don't need most of the contents for their daily work, but they do need the flexibility to work in any part of the repo, without coming up against arbitrary boundaries
- We wanted to use the existing features of git as much as possible, and not make any changes to the client unless absolutely necessary

We eventually came up with the idea to virtualize the file system underneath the repo, which gave us a few key benefits:

- Only download blobs that the developer actually needs. For most people, this will be about 50-100K files, with their associated histories, and most of the contents of the repo never have to be copied.
  - It naturally solves the large, undiffable files problem as well
- With some clever tricks, we can convince git to only look at the files that the developer has actually worked with, so that operations like git status and checkout are no longer linear on 3M files, but on a much smaller number
- We can get the benefits of the sparse-checkout feature, namely only writing the necessary files to the working directory, without its drawbacks. Most importantly, the set of files that are checked out is now dynamic, and entirely based on the specific files that the developer needs.
- None of the build tools, IDEs, etc have to be updated to know that this is happening, because they can just open the files they need and the file system will make sure the contents show up

Of course, there would be many challenges too.

- Building a file system is, to put it mildly, difficult.
- We knew that performance is absolutely critical. Our developers would forgive us if the first file access is slow due to network latency, but the second file access better have no noticable overhead or people would come after us with pitchforks.
- We figured we'd have to be smart about prefetching some things, to avoid unnecessary latency. In general, the smaller the contents, the more you feel the latency. As an example, tree objects are tiny, and there are lots of them.
- We still had to figure out how to get git to play nicely on top of the virtualized file system. Could we actually convince git to not walk all 3.5M files even though it looks like they're all on disk? How thoroughly does git honor the sparse-checkout configuration in all of its commands? Are there cases where it will walk through too many blobs? (Hint: this is called foreshadowing)

In GVFS Architecture, we'll discuss how we designed GVFS to solve those issues and meet our performance goals.

Saeed Noursalehi is a Principal Program Manager on the Azure DevOps team at Microsoft, and works on making Git scale for the largest teams in Microsoft