Limited Refs

02/20/2018 • 11 minutes to read

In this article

By: Saeed Noursalehi

As we discussed in the previous article, one of the major scale challenges for Git is dealing with the linear costs associated with having too many refs. Let's look a little more closely at the problem, and how we've solved it. And of course, since there's no such thing as a free lunch, we'll discuss the tradeoffs created by our solution.

The problem with refs

What is a ref?

We often refer to this scale challenge as "too many branches" because it's easier to talk about, but in reality the challenge is "too many refs".

Recommended reading before going further:

- Really, the whole Pro Git book if you have the time: https://git-scm.com/book/en/v2
- For this specific issue, at least read Chapter 3 https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell and section 9.3 https://git-scm.com/book/en/v1/Git-Internals-Git-References

To summarize, Git history is a Directed Acyclic Graph (DAG) of commits, and a ref is a pointer to one of those commits. Branches and tags are special types of refs, but there are other kinds of refs as well. Refs are used to give convenient names to commits. They are also used in negotiations between the Git client and server to figure out how much contents to fetch and push. And in both cases, having lots and lots of refs can cause problems.

Linear perf costs

When you run "git fetch" or "git pull", the communication between the client and server looks something like this:

- Git looks up the URL of the repo. By default, this is the URL configured for the origin remote.
- Git sends a request to `/info/refs`, and the server responds with the list of all known refs and the current commit id that each ref is pointing to (known as the tip commit of the ref)
- Git takes that list of refs and compares it with the list that it currently has locally. For each of these tip commits, Git checks to see if it already has that commit id, and if not, it will have to request it from the server.
- The Git client then sends a request to `/upload-pack` with information about which commit ids it has

and which commit ids it wants. The client and server go back and forth a few times until they whittle the list of haves and wants down as much as they can.
- Finally the server is able to compute which commits it needs to send to the client, and it creates a packfile with all of the contents for those commits and sends that to the client
- If you want even more gory details, here's the full description of the protocol: https://github.com/git/git/blob/master/Documentation/technical/pack-protocol.txt

The main point to take away from all of that is that the client and server have to do a lot of calculations to figure out what to fetch, and the number of calculations grows linearly with the number of refs. So the more refs there are, the longer it takes to even figure out what to send, before the client can start downloading the new contents. Before we did anything to fix this, devs in the Azure DevOps repo had a pretty frustrating experience when they fetched. After typing "git fetch" or "git pull", they would see the first acknowledgement from the server ("remote: Microsoft (R) Visual Studio (R) Team Services") fairly quickly, and then... nothing... sometimes lasing 5 minutes or longer, before a message like "remote: Found X objects to send". Only then would the download of new contents actually begin. The Azure DevOps repo typically has as many as 20K refs in it, and the negotiations for that many refs take a while.

Usability costs

There are also usability issues with having too many refs. Have you ever tried running "git branch -a" in a repo with 20K branches? Good luck finding the one you want.

While refs are technically unstructured and can have whatever names you want to give them, we learned early on that you really need a good naming structure to help organize all those branches. In the Azure DevOps repo, we like to use the pattern `users//` for topic branches, and similar patterns for release branches (releases*/), team branches (teams/*), and other "types" of branches that we commonly use. We'll talk more about those best practices in an upcoming article. We have also updated the tools that we build (like the Azure DevOps web UI and Visual Studio's Team Explorer) to render branches as a hierarchy, but not all Git tools do this. See the example above about listing branches using the Git command line.

But even with those improvements, having 20K refs in the Azure DevOps repo, or 100K+ refs in the Windows repo presents a serious usability challenge for a dev who needs to find one branch out of that huge list.

Solution: Limited Refs

Our solution to this problem is to allow a repo to be configured such that it hides most of the refs from the client. We were able to do this with no changes at all to the Git client or to the protocol. Because the first step in the fetch protocol is that the client asks the server for the list of all refs, we simply changed our server to filter its response to that info/refs request, and the rest of the fetch protocol plus all of your existing Git tools continue to work as normal. The difficult part of this is: what refs should we include in that info/refs response? We spent a bit of time refining this, because there are various tradeoffs to consider. The fewer branches you respond with, the less time each person spends fetching, but then they might not get a branch that they care about. We also realized that there are cases where you really don't want the server to lie to you because you know you need every single branch.

To solve these issues, we've allowed for certain branches to be marked as "important" for all users of the repo, we've allowed users to mark additional branches that they know they care about, and we've also allowed you to bypass this feature entirely and specify that you really want all branches.

Important branches

There are a set of branches in any repo that everyone will be very likely to interact with, like the master and release branches. These refs can be configured to be included in the info/refs response for everyone, so that all users will see them. As an aside, it's very helpful to have a well-organized hierarchical naming pattern for your branches so that you can configure a small number of ref "folders" and not have to keep updating this list all the time.

For the Azure DevOps repo, we know that every user needs the master branch. We also create a release branch at the end of each sprint, with the naming pattern of `releases/`, and we figure most people will need these branches as well. So we've configured Limited Refs to always return these refs to all of our users. As I write this, this configuration alone reduces the number of visible refs in the Azure DevOps repo from about 20K down to 100.

For the Windows repo, even all of the "important" branches would be too many, so we've taken a more extreme position there and just included the master branch by default. More on this in the tradeoffs section below.

My branches

Any branch that you personally created is one that you almost certainly care about. So in addition to returning all of the branches that are configured as important, we also return to you all branches that you have personally created.

Favorite branches

And we know that no matter how much flexibility we provide in the configuration of important branches, we can never fine tune it perfectly for each individual person. Therefore in the Azure DevOps web UI (or via REST APIs) you can also mark any branch or branch "folder" as a favorite, and all of your favorite branches are also included in the responses to your info/refs requests.

The result

The net effect of all this is that the branches that you'll see in your local clone of a repo are:

- All of the repo's important branches
- All branches that you created
- All branches that you favorited

As a result of that, a typical user in the Azure DevOps repo now sees around 100 branches at clone time instead of close to 20K, and the negotiations at the beginning of fetch now take a couple of seconds instead of minutes. And a typical user in the Windows repo sees a couple dozen branches instead of 100K or more.

Bypassing Limited Refs

When you turn on Limited Refs for a repo and make it the default behavior, all of the filtering rules described above will apply to anyone who uses the normal repo URL. For a typical repo, this URL looks like `https://.visualstudio.com/Project/_git/Repo`. However, sometimes you just need access to all the refs, and for those situations, we've added an alternative form of the repo URL: `https://.visualstudio.com/Project/_git/_full/Repo`. By adding that `_full` part to the URL, you can bypass Limited Refs and get an unfiltered list of refs when you clone or fetch. This is often useful for build servers and other automated tools that know how to manage their own list of refs.

Tradeoffs

Finding other people's branches

The biggest issue with Limited Refs is that now users can't see all of the branches in the repo. This is of course the whole point – the fewer branches you have, the leaner your repo is. But there are plenty of times when you need to work with a branch that might not be included in the limited set that you're working with locally. The most common example is that someone posted a pull request for their topic branch, and you want to checkout that branch and run some local tests on it. The simplest answer to this is that you can just go mark that branch as a favorite and then fetch again. But this isn't a great answer on its own, because it would require you to have to remember a bunch of steps (go to the web, find the branch, favorite it, go to your repo, fetch) every time you want to work with someone else's branch.

In the Azure DevOps repo, what we've done is adopt a naming pattern that allows people to very easily favorite all of their nearby teammates' topic branches. Across the broader team, there are 400+ people, but any given feature team has around 10 or so people in it. So the pattern we've adopted is that each team creates all of their topic branches using the naming pattern `teams///`. Each person on a team can then just go favorite the `teams/` "branch folder" in the Azure DevOps web UI, and from then on they have access to all branches created by their nearby teammates.

The Windows team has taken a different approach. They've created a command line tool that will call the Azure DevOps REST APIs to favorite a branch.If a user wants to favorite a branch, they can just call that tooland pass in the name of the branch, and then run `git fetch` again.

Branch organization

This is perhaps the biggest issue with Limited Refs – it requires a fair bit of upfront work to get your branches under control using well-defined naming patterns. Otherwise, it becomes very difficult to manage the set of important branches, difficult to favorite "folders" of branches that you care about, etc.

In an upcoming article, we'll get into more detail about branch naming best practices.

Why not forks?

Astute readers often notice that Limited Refs looks an awful lot like forks, so why not just use forks?

This is a bigger topic that we may expand on in a future article, but the short answer is that forks were designed to solve a trust boundary problem. In the open source world, you often accept changes from people with whom you have no trust relationship. In that model, it's very important to confine the changes that other people can make and to selectively decide which parts of their changes you will accept into your own repo.

Within a team in a typical enterprise environment, there is no trust issue, because you all work for the same company. Forks can still be useful in this environment, and many teams prefer to work using forks. However, forks do add some additional hurdles, and it's often more efficient to just have everyone work in a single repo. For example, consider the scenario we talked about earlier where someone else pushed a branch and you just want to check it out locally and run some tests on it. With forks, you have to manage multiple remotes, carefully manage where you're pulling from and pushing to, etc.

So forks are great, and Limited Refs are great, and you need to decide which one is right for your team. In the end, forks solve a trust boundary problem, and Limited Refs solves a scale problem.

Next article in the series

- The Race to Push

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