Moving LLVM Projects to GitHub

Introduction

This is a proposal to move our current revision control system from our own hosted Subversion to Git-Hub. Below are the financial and technical arguments as to why we need such a move and how will people (and validation infrastructure) continue to work with a Git-based LLVM.

There will be a survey pointing at this document which we'll use to gauge the community's reaction and, if we collectively decide to move, the time-frame. Be sure to make your view count.

This proposal is divided into the following parts:

- Outline of the reasons to move to Git and GitHub
- Description on the options
- What some examples of workflow will look like (compared to currently)
- The proposed migration plan

What This Proposal is *Not* About

Changing the development policy: the development of LLVM will continue as it exists now.

This proposal relates only to moving the hosting of our source-code repository from SVN hosted on our own servers to Git hosted on GitHub. We are not proposing other workflow changes here. That is, it should not be assumed that moving to GitHub implies using GitHub's issue tracking, or using the GitHub UI for pull-requests and/or code-review.

Every existing contributor will get commit access on demand under the same condition as currently. Those who don't have an existing GitHub account will have to create one in order to continue having commit access.

Why Git, and Why GitHub?

Why Move At All?

One of the reasons for the move, and why this discussion started in the first place, is that we currently host our own Subversion server and Git mirror in a voluntary basis. The LLVM Foundation sponsors the server and provides limited support, but there is only so much it can do.

Volunteers are not sysadmins themselves, but compiler engineers that happen to know a thing or two about hosting servers. We also don't have 24/7 support, and we sometimes wake up to see that continuous integration is broken because the SVN server is either down or unresponsive.

On the other hand, there are multiple services out there (GitHub, GitLab, BitBucket among others) that offer that same service (24/7 stability, disk space, Git server, code browsing, forking facilities, etc) for free.

Why Git?

It seems that Git is new coders first choice nowadays. A lot of them have never used SVN, CVS, or anything else. Websites like GitHub have changed the landscape of open source contributions, reducing the cost of first contribution and fostering collaboration.

Git is also the version control many (most?) LLVM developers use. Despite the sources being stored in a SVN server, these developers are already using Git through the Git-SVN integration.

Git allows you to:

- Commit, squash, merge, and fork locally without touching the remote server.
- Maintain as many local branches as you like, letting you maintain multiple threads of development.
- Collaborate on these branches (e.g. through your own fork of llvm on GitHub).
- Inspect the repository history (blame, log, bisect) without Internet access.

In addition, because Git seems to be replacing many OSS projects' version control systems, there are many tools that are built over Git. Future tooling may be more likely to support Git first (if not only).

Why GitHub?

GitHub, like GitLab and BitBucket, provides free code hosting for open source projects. Any of these could replace the code-hosting infrastructure that we have today.

These services also have a dedicated team to monitor, migrate, improve and distribute the contents of the repositories depending on region and load.

All things being equal, GitHub has one important advantage over GitLab and BitBucket: It offers read-write **SVN** access to the repository (https://github.com/blog/626-announcing-svn-support). This would enable people to continue working post-migration as though our code were still canonically in an SVN repository.

In addition, there are already multiple LLVM mirrors on GitHub, indicating that part of our community has already settled there.

On Managing Revision Numbers with Git ¶

The current SVN repository hosts all the LLVM sub-projects alongside each other. A single revision number (e.g. r123456) thus identifies a consistent version of all LLVM sub-projects.

Git does not use sequential integer revision number but instead uses a hash to identify each commit. (Linus mentioned that the lack of such revision number is "the only real design mistake" in Git [TorvaldRevNum].)

The loss of a sequential integer revision number has been a sticking point in past discussions about Git:

- "The 'branch' I most care about is mainline, and losing the ability to say 'fixed in r1234' (with some sort of monotonically increasing number) would be a tragic loss." [LattnerRevNum]
- "I like those results sorted by time and the chronology should be obvious, but timestamps are incredibly cumbersome and make it difficult to verify that a given checkout matches a given set of results." [TrickRevNum]
- "There is still the major regression with unreadable version numbers. Given the amount of

Bugzilla traffic with 'Fixed in...', that's a non-trivial issue." [JSonnRevNum]
- "Sequential IDs are important for LNT and llvmlab bisection tool." [MatthewsRevNum].

However, Git can emulate this increasing revision number: `git rev-list —count <commit-hash>`. This identifier is unique only within a single branch, but this means the tuple `(num, branch-name)` uniquely identifies a commit.

We can thus use this revision number to ensure that e.g. `clang -v` reports a user-friendly revision number (e.g. `master-12345` or `4.0-5321`). This should be enough to address the objections raised above with respect to this aspect of Git.

What About Branches and Merges?

In contrast to SVN, Git makes branching easy. Git's commit history is represented as a DAG, a departure from SVN's linear history.

However, we propose to *enforce linear history* in our canonical Git repository repository. (This is not uncommon amongst many large users of Git.)

We'll do this with a combination of client-side and server-side hooks. GitHub offers a feature called `Status Checks`: a branch protected by `status checks` requires commits to be whitelisted before the push can happen. A supplied pre-push hook on the client side will run and check the history, before whitelisting the commit being pushed [statuschecks].

What About Commit Emails?

An extra bot will need to be set up to continue to send emails for every commit. We'll keep the exact same email format as we currently have (a change is possible later, but beyond the scope of the current discussion), the only difference being changing the URL from `http://llvm.org/viewvc/...` to `http://github.org/llvm/...`.

It would also be possible to rely on GitHub integrated email service, but the email format differs significantly (there is no inline diff or attached patch for instance).

One or Multiple Repositories?

There are two major proposals for how to structure our Git repository: The "multirepo" and the "monorepo".

1. *Multirepo* - Moving each SVN sub-project into its own separate Git repository.
2. *Monorepo* - Moving all the LLVM sub-projects into a single Git repository.

The first proposal would mimic the existing official separate read-only Git repositories (e.g. http://llvm.org/git/compiler-rt.git), while the second one would mimic an export of the SVN repository (i.e. it would look similar to https://github.com/llvm-project/llvm-project, where each sub-project has its own top-level directory).

With the Monorepo, the existing single-subproject mirrors (i.e. for example http://llvm.org/git/compiler-rt.git) with git-svn read-write access would continue to be maintained.

There are other impacts that are less immediate and less technical: the first proposal of keeping the repository separate implies that the sub-projects are more independent from each other, while the second proposal encourage better code sharing and refactoring across projects, for example reusing a

datastructure initially in LLDB by moving it into libSupport. It would also be easier to decide to extract some pieces of libSupport and/or ADT to a new top-level *independent* library that can be reused in libcxxabi for instance. Finally, it also encourages to update all the sub-projects when changing API or refactoring code ("git grep" works across sub-projects for instance).

As another example, some developers think that the division between e.g. clang and clang-tools-extra is not useful. With the monorepo, we can move code around as we wish and preserve history. With the multirepo, moving clang-tools-extra into clang would be more complicated than a simple `git mv` command, and we would end up losing history.

Some concerns have been raised that having a single repository would be a burden for downstream users that have interest in only a single repository, however this is addressed by keeping the single-subproject Git mirrors for each project just as we do today. Also the GitHub SVN bridge allows to contribute to a single sub-project the same way it is possible today (see below before/after section for more details).

Finally, nobody will be forced to compile projects they don't want to build. The exact structure is TBD, but even if you use the monorepo directly, we'll ensure that it's easy to set up your build to compile only a few particular sub-projects.

How Do We Handle A Single Revision Number Across Multiple Repositories?

A key need is to be able to check out multiple projects (i.e. lldb+llvm or clang+llvm+libcxx for example) at a specific revision.

Under the monorepo, this is a non-issue. That proposal maintains the property of the existing SVN repository that the sub-projects move synchronously, and a single revision number (or commit hash) identifies the state of the development across all projects.

Under the multirepo, things are more involved. We describe here the proposed solution.

Fundamentally, separated Git repositories imply that a tuple of revisions (one entry per repository) is needed to describe the state across repositories/sub-projects. For example, a given version of clang would be *<LLVM-12345, clang-5432, libcxx-123, etc.>*.

To make this more convenient, a separate *umbrella* repository would be provided. This repository would be used for the sole purpose of understanding the sequence (with some granularity) in which commits were added across repository and to provide a single revision number.

This umbrella repository will be read-only and periodically updated to record the above tuple. The proposed form to record this is to use Git [submodules], possibly along with a set of scripts to help check out a specific revision of the LLVM distribution.

A regular LLVM developer does not need to interact with the umbrella repository – the individual repositories can be checked out independently – but you would need to use the umbrella repository to bisect or to check out old revisions of llvm plus another sub-project at a consistent version.

This umbrella repository will be updated automatically by a bot (running on notice from a webhook on every push, and periodically). Note that commits in different repositories pushed within the same time frame may be visible together or in undefined order in the umbrella repository.

Later sections illustrates how end-users interacts with this repository for various use-cases, and the Previews section links to a usable repository illustrating this structure.

Workflow Before/After

This section goes through a few examples of workflows.

Checkout/Clone a Single Project, without Commit Access

Except the URL, nothing changes. The possibilities today are:

```
svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
# or with Git
git clone http://llvm.org/git/llvm.git
```

After the move to GitHub, you would do either:

```
git clone https://github.com/llvm-project/llvm.git
# or using the GitHub svn native bridge
svn co https://github.com/llvm-project/llvm/trunk
```

The above works for both the monorepo and the multirepo, as we'll maintain the existing read-only views of the individual sub-projects.

Checkout/Clone a Single Project, with Commit Access

Currently

```
# direct SVN checkout
svn co https://user@llvm.org/svn/llvm-project/llvm/trunk llvm
# or using the read-only Git view, with git-svn
git clone http://llvm.org/git/llvm.git
cd llvm
git svn init https://llvm.org/svn/llvm-project/llvm/trunk --username=<username>
git config svn-remote.svn.fetch :refs/remotes/origin/master
git svn rebase -l  # -l avoids fetching ahead of the git mirror.
```

Commits are performed using *`svn commit`* or with the sequence *`git commit`* and *`git svn dcommit`*.

Multirepo Proposal

With the multirepo proposal, nothing changes but the URL, and commits can be performed using *`svn commit`* or *`git commit`* and *`git push`*:

```
git clone https://github.com/llvm/llvm.git llvm
# or using the GitHub svn native bridge
svn co https://github.com/llvm/llvm/trunk/ llvm
```

Monorepo Proposal

With the monorepo, there are multiple possibilities to achieve this. First, you could just clone the full repository:

```
git clone https://github.com/llvm/llvm-projects.git llvm
# or using the GitHub svn native bridge
svn co https://github.com/llvm/llvm-projects/trunk/ llvm
```

At this point you have every sub-project (llvm, clang, lld, lldb, ...), which **doesn't imply you have to build all of them**. You can still build only compiler-rt for instance. In this way it's not different from someone who would check out all the projects with SVN today.

You can commit as normal using *git commit* and *git push* or *svn commit*, and read the history for a single project (*git log libcxx* for example).

If you don't want to have the sources for all the sub-projects checked out for, there are again a few options.

First, you could hide the other directories using a Git sparse checkout:

```
git config core.sparseCheckout true
echo /compiler-rt > .git/info/sparse-checkout
git read-tree -mu HEAD
```

The data for all sub-projects is still in your *.git* directory, but in your checkout, you only see *compiler-rt*. Git compresses its history, so a clone of everything is only about 2x as much data as a clone of llvm only (and in any case this is dwarfed by the size of e.g. a llvm build dir).

Before you push, you'll need to fetch and rebase as normal. However when you fetch you'll likely pull in changes to sub-projects you don't care about. You may need to rebuild and retest, but only if the fetch included changes to a sub-project that your change depends on. You can check this by running:

```
git log origin/master@{1}..origin/master libcxx
```

This shows you all of the changes to *libcxx* since you last fetched. This command can be hidden in a script so that *git llvmpush* would perform all these steps, fail only if such a dependent change exists, and show immediately the change that prevented the push. An immediate repeat of the command would (almost) certainly result in a successful push. (This is an extra step that you don't need in the multirepo, but for those of us who work on a sub-project that depends on llvm, it has the advantage that we can check whether we pulled in any changes to say clang *or* llvm.)

A second option is to use svn via the GitHub svn native bridge:

```
svn co https://github.com/llvm/llvm-projects/trunk/compiler-rt compiler-rt  —username=...
```

This checks out only compiler-rt and provides commit access using "svn commit", in the same way as it would do today.

Finally, you could use *git-svn* and one of the sub-project mirrors:

```
# Clone from the single read-only Git repo
git clone http://llvm.org/git/llvm.git
cd llvm
# Configure the SVN remote and initialize the svn metadata
$ git svn init https://github.com/joker-eph/llvm-project/trunk/llvm —username=...
git config svn-remote.svn.fetch :refs/remotes/origin/master
git svn rebase -l
```

In this case the repository contains only a single sub-project, and commits can be made using *git svn dcommit*, again **exactly as we do today**.

Checkout/Clone Multiple Projects, with Commit Access

Let's look how to assemble llvm+clang+libcxx at a given revision.

Currently

```
svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm -r $REVISION
```

```
cd llvm/tools
svn co http://llvm.org/svn/llvm-project/clang/trunk clang -r $REVISION
cd ../projects
svn co http://llvm.org/svn/llvm-project/libcxx/trunk libcxx -r $REVISION
```

Or using git-svn:

```
git clone http://llvm.org/git/llvm.git
cd llvm/
git svn init https://llvm.org/svn/llvm-project/llvm/trunk --username=<username>
git config svn-remote.svn.fetch :refs/remotes/origin/master
git svn rebase -l
git checkout `git svn find-rev -B r258109`
cd tools
git clone http://llvm.org/git/clang.git
cd clang/
git svn init https://llvm.org/svn/llvm-project/clang/trunk --username=<username>
git config svn-remote.svn.fetch :refs/remotes/origin/master
git svn rebase -l
git checkout `git svn find-rev -B r258109`
cd ../../projects/
git clone http://llvm.org/git/libcxx.git
cd libcxx
git svn init https://llvm.org/svn/llvm-project/libcxx/trunk --username=<username>
git config svn-remote.svn.fetch :refs/remotes/origin/master
git svn rebase -l
git checkout `git svn find-rev -B r258109`
```

Note that the list would be longer with more sub-projects.

Multirepo Proposal

With the multirepo proposal, the umbrella repository enters the dance. This is where the mapping from a single revision number to the individual repositories revisions is stored.:

```
git clone https://github.com/llvm-beanz/llvm-submodules
cd llvm-submodules
git checkout $REVISION
git submodule init
git submodule update clang llvm libcxx
# the list of subproject is optional, `git submodule update` would get them all.
```

At this point the clang, llvm, and libcxx individual repositories are cloned and stored alongside each other. There exist flags you can use to inform CMake of your directory structure, and alternatively you can just symlink *clang* to *llvm/tools/clang*, etc.

Monorepo Proposal

The repository contains natively the source for every sub-projects at the right revision, which makes this straightforward:

```
git clone https://github.com/llvm/llvm-projects.git llvm
cd llvm
git checkout $REVISION
```

As before, at this point clang, llvm, and libcxx are stored in directories alongside each other.

Commit an API Change in LLVM and Update the Sub-projects

Today this is easy for subversion users, and possible but not straighfoward for git-svn users. Few Git users try to e.g. update LLD or Clang in the same commit as they change an LLVM API.

The multirepo proposal does not address this: one would have to commit and push separately in every individual repository. It might be possible to establish a protocol whereby users add a special token to their commit messages that causes the umbrella repo's updater bot to group all of them into a single revision.

The monorepo proposal handles this natively and makes this use case trivial.

Branching/Stashing/Updating for Local Development or Experiments

Currently

SVN does not allow this use case, but developers that are currently using git-svn can do it. Let's look in practice what it means when dealing with multiple sub-projects.

To update the repository to tip of trunk:

```
git pull
cd tools/clang
git pull
cd ../../projects/libcxx
git pull
```

To create a new branch:

```
git checkout -b MyBranch
cd tools/clang
git checkout -b MyBranch
cd ../../projects/libcxx
git checkout -b MyBranch
```

To switch branches:

```
git checkout AnotherBranch
cd tools/clang
git checkout AnotherBranch
cd ../../projects/libcxx
git checkout AnotherBranch
```

Multirepo Proposal

The multirepo works the same as the current Git workflow: every command needs to be applied to each of the individual repositories.

Monorepo Proposal

Regular Git commands are sufficient, because everything is in a single repository:

To update the repository to tip of trunk:

```
git pull
```

To create a new branch:

```
git checkout -b MyBranch
```

To switch branches:

```
git checkout AnotherBranch
```

Bisecting

Assuming a developer is looking for a bug in clang (or lld, or lldb, ...).

Currently

SVN does not have builtin bisection support, but the single revision across sub-projects makes it straightforward to script around.

Using the existing Git read-only view of the repositories, it is possible to use the native Git bisection script over the llvm repository, and use some scripting to synchronize the clang repository to match the llvm revision.

Multirepo Proposal

With the multi-repositories proposal, the cross-repository synchronization is achieved using the umbrella repository. This repository contains only submodules for the other sub-projects. The native Git bisection can be used on the umbrella repository directly. A subtlety is that the bisect script itself needs to make sure the submodules are updated accordingly.

For example, to find which commit introduces a regression where clang-3.9 crashes but not clang-3.8 passes, one should be able to simply do:

```
git bisect start release_39 release_38
git bisect run ./bisect_script.sh
```

With the *`bisect_script.sh`* script being:

```
#!/bin/sh
cd $UMBRELLA_DIRECTORY
git submodule update llvm clang libcxx #....
cd $BUILD_DIR

ninja clang || exit 125   # an exit code of 125 asks "git bisect"
                          # to "skip" the current commit

./bin/clang some_crash_test.cpp
```

When the *`git bisect run`* command returns, the umbrella repository is set to the state where the regression is introduced, one can inspect the history on every sub-project compared to the previous revision in the umbrella (it is possible that one commit in the umbrella repository includes multiple commits in the sub-projects).

Monorepo Proposal

Bisecting on the monorepo is straightforward and almost identical to the multirepo situation explained above. The granularity is finer since each individual commits in every sub-projects participate in the bisection. The bisection script does not need to include the *`git submodule update`* step.

Living Downstream

Depending on which of the multirepo or the monorepo proposal gets accepted, and depending on the integration scheme, downstream projects may be differently impacted and have different options.

- If you were pulling from the SVN repo before the switch to Git. The monorepo will allow you to continue to use SVN. The main caveat is that you'll need to be prepared for a one-time change to the revision numbers. The multirepo proposal still offers an SVN access to each individual

sub-project, but the SVN revision for each sub-project won't be synchronized.
- If you were pulling from one of the existing read-only Git repos, this also will continue to work as before as they will continue to exist in any of the proposal.

Under the monorepo proposal, you have a third option: migrating your fork to the monorepo. This can be particularly beneficial if your fork touches multiple sub-projects (e.g. llvm and clang), because now you can commingle commits to llvm and clang in a single repository.

As a demonstration, we've migrated the "CHERI" fork to the monorepo in two ways:

- Using a script that rewrites history (including merges) so that it looks like the fork always lived in the monorepo [LebarCHERI]. The upside of this is when you check out an old revision, you get a copy of all llvm sub-projects at a consistent revision. (For instance, if it's a clang fork, when you check out an old revision you'll get a consistent version of llvm proper.) The downside is that this changes the fork's commit hashes.
- Merging the fork into the monorepo [AminiCHERI]. This preserves the fork's commit hashes, but when you check out an old commit you only get the one sub-project.

If you keep a split-repository solution downstream, upstreaming patches to the monorepo is always possible (the splitrepo is obvious): you can apply the patches in the appropriate subdirectory of the monorepo.

Monorepo Variant

A variant of the monorepo proposal is to group together in a single repository only the projects that are *rev-locked* to LLVM (clang, lld, lldb, ...) and leave projects like libcxx and compiler-rt in their own individual and separate repositories.

Note however that many users of the monorepo would benefit from having all of the pieces needed for a full toolchain present in one repository. And for newcomers, getting and building a toolchain is easier.

Also, developers who hack only on one of these sub-projects can continue to use the single sub-project Git mirrors, so their workflow is unchanged. (That is, they aren't forced to download or check out all of llvm, clang, etc. just to make a change to libcxx.)

Previews

FIXME: make something more official/testable and update all the URLs in the examples above.

Example of a working version:

- Repository: https://github.com/llvm-beanz/llvm-submodules
- Update bot: http://beanz-bot.com:8180/jenkins/job/submodule-update/

Remaining Issues

LNT and llvmlab will need to be updated: they rely on unique monotonically increasing integer across branch [MatthewsRevNum].

Straw Man Migration Plan

STEP #1 : Before The Move

1. Update docs to mention the move, so people are aware of what is going on.
2. Set up a read-only version of the GitHub project, mirroring our current SVN repository.
3. Add the required bots to implement the commit emails, as well as the umbrella repository update (if the multirepo is selected) or the read-only Git views for the sub-projects (if the monorepo is selected).

STEP #2 : Git Move

4. Update the buildbots to pick up updates and commits from the GitHub repository. Not all bots have to migrate at this point, but it'll help provide infrastructure testing.
5. Update Phabricator to pick up commits from the GitHub repository.
6. Instruct downstream integrators to pick up commits from the GitHub repository.
7. Review and prepare an update for the LLVM documentation.

Until this point nothing has changed for developers, it will just boil down to a lot of work for buildbot and other infrastructure owners.

Once all dependencies are cleared, and all problems have been solved:

STEP #3: Write Access Move

8. Collect developers' GitHub account information, and add them to the project.
9. Switch the SVN repository to read-only and allow pushes to the GitHub repository.
10. Update the documentation
11. Mirror Git to SVN.

STEP #4 : Post Move

10. Archive the SVN repository.
11. Update links on the LLVM website pointing to viewvc/klaus/phab etc. to point to GitHub instead.

[LattnerRevNum] Chris Lattner, http://lists.llvm.org/pipermail/llvm-dev/2011-July/041739.html

[TrickRevNum] Andrew Trick, http://lists.llvm.org/pipermail/llvm-dev/2011-July/041721.html

[JSonnRevNum] Joerg Sonnenberg, http://lists.llvm.org/pipermail/llvm-dev/2011-July/041688.html

[TorvaldRevNum] Linus Torvald, http://git.661346.n2.nabble.com/Git-commit-generation-numbers-td6584414.html

[MatthewsRevNum] *(1, 2)* Chris Matthews, http://lists.llvm.org/pipermail/cfe-dev/2016-July/049886.html

[submodules] Git submodules, https://git-scm.com/book/en/v2/Git-Tools-Submodules)

[statuschecks] GitHub status-checks, https://help.github.com/articles/about-required-status-checks/

[LebarCHERI] Port *CHERI* to a single repository rewriting history, http://lists.llvm.org/pipermail/llvm-dev/2016-July/102787.html

[AminiCHERI] Port *CHERI* to a single repository preserving history, http://lists.llvm.org/pipermail/llvm-dev/2016-July/102804.html