Clangd automatic index

sammccall@google.com - 2018-10-08 - Proposal - this document is PUBLIC

TL;DR

Clangd's current static/dynamic index system puts too much burden on the user to create and update a static index. Most users don't do this, and get limited code completion and cross-references functionality.
For small and medium projects, clangd should manage the index itself instead. This doc proposes how.

Current state, and the problem

Clangd has a single view of the index, which overlays two concrete indexes:

Static index	**Dynamic index**
Manual: built by running clangd-indexer	Automatic: built as files are opened/edited
Complete: all symbols in the project	Incomplete: symbols included by opened files
Stale: needs manual rebuild	Up-to-date: sees editor changes
Efficient: each symbol occurs once	Inefficient: one copy of symbol per including TU

The idea is that each covers for the weaknesses of the other: the static index provides good coverage, and the dynamic index shows the latest changes. The dynamic index doesn't have to be compact as it only needs to have relatively few symbols.

However the requirement that the static index be manually built means **many people won't bother**:[1]

- casual users don't know/care how the server works, and rightly don't want to read docs
- building the static index up-front is slow and inconvenient
- clangd "seems to work" without it

These users get features that work only sometimes or incompletely (particularly xref features).
People that do build the index have a complicated setup experience, and still won't rebuild the index between every clangd run, leading to stale results.

Meanwhile, competing language servers (particularly cquery and ccls) build a full index on startup, and provide working cross-reference features without complicated setup. This is a much better UX.

Elements of a good solution

Our starting point should be "index everything on startup", but there are reasons we don't do that today.
We'll need some changes to get this to work:

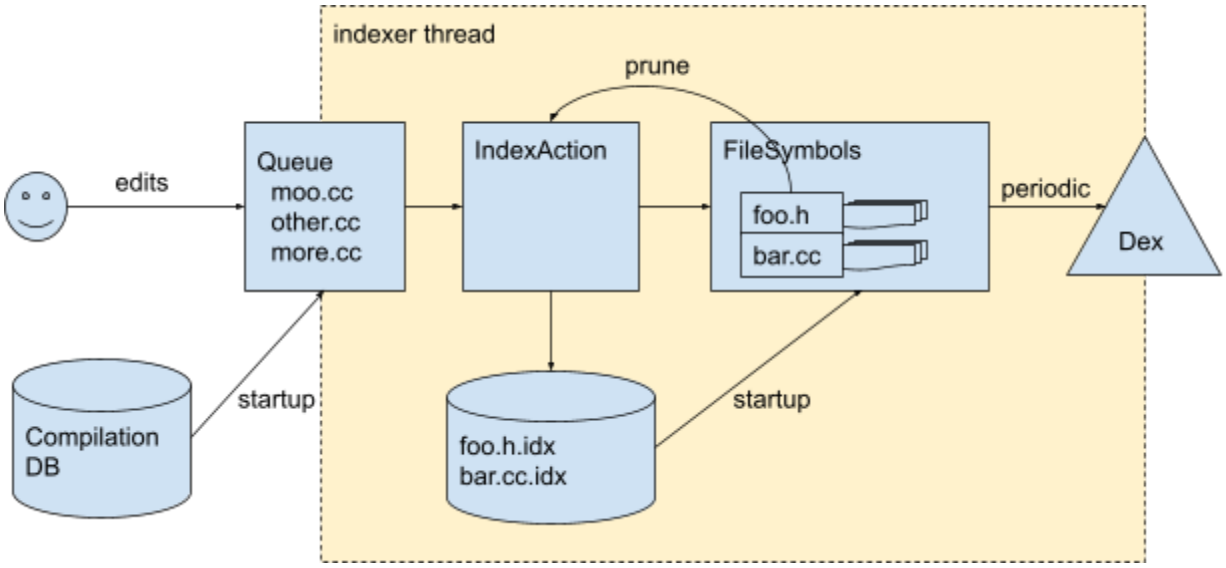
- **Background indexing on dedicated threads**. On first startup, there may be thousands of files we need to index. They will need to form an orderly queue! We may choose to prioritize open files.

[1] The current approach is highly effective in a monorepo environment where we maintain the static index as a service!

- **Incrementally persist the index to disk**. Burning CPU on *every* startup is wasteful, we should cache the results. Per-file granularity lets us make progress even in short editor sessions.
- **Associate symbols with files instead of TUs**. Today's dynamic index is partitioned per CPP file, and stores all symbols transitively visible in the TU. Header symbols are duplicated often. This wastes indexing CPU, index RAM, and disk space. We should move to a file-centric model with one entry per header. (This will come at a cost of accuracy for some patterns, like .inc files).
- **Lightweight indexing of edited files**. It's valuable for edits to files to show up in the index with minimal latency (e.g. for find-definition). Requiring full background indexing as the user types will be slower and prevent progress on other files. A minimal indexing of the AST-for-diagnostics will help here.

Proposal

High-level flow



A loop running in a dedicated thread is responsible for indexing translation units from a queue.
Initially, the queue is populated from the compilation database, as the user edits files[2] they are added to the queue for reindexing.
The indexing is a standalone FrontendAction with no AST/preamble reuse. Inputs are TU granularity, outputs are file-granularity: as a TUs is indexed, the results are partitioned by declaring file. The results are stored both in memory (from which the live index can be rebuilt) and on disk (for fast startup next time).
A hash of file contents is also retained for invalidation - IndexAction should skip symbols from files whose index is up-to-date.

Open questions:

1. How do we handle reindexing of headers - which TU to rebuild?
2. Are there advantages to running the indexer out-of-process, decoupling from clangd lifetime?
3. Can this model work with an external static index (e.g. for large monorepos)

Disk storage

As described above, we store one index shard per source file.

[2] In future, also observed file changes on disk.

On-disk we should at least initially use the binary static index storage format. It's fairly compact, and fast to read/write.
We should write the files in a .clangd-index/ under the compilation database root,[3] which is typically the source tree root. Users will be advised to mark these files as .gitignore etc.
Directory structure of indexed files may be complicated, we will write a flat foo.cc_ABCD.idx, where foo.cc is the bare filename, and ABCD is a hash of the absolute directory. Full path is encoded inside the file.

Open questions:

- How much are we going to blow up the index size by not having cross-file string deduplication?
- Should multiple clangd processes in a workspace coordinate somehow?
- Would we get important performance boosts by merging files somehow (e.g. by directory)

Queue/indexer loop

The simplest model here is a single thread that processes TUs in simple FIFO fashion. When initially populated from the compilation database, this is essentially arbitrary order (random shuffle may help get good header coverage sooner).
To ensure the user sees results for headers in the TU ASAP, we should prioritize TUs coming from user edits over those from the CDB or FS changes. Simply pushing these on the **front** of the queue may be enough. We should only do this when the file is totally unindexed, to avoid starving the queue.

Open questions:

- Indexing separate TUs is embarassingly parallel, but also resource-intensive. Is using multiple threads here a good idea?
- Being able to index header files directly would simplify many things, would it work in practice?

Indexer details

The SymbolCollector must be configured to collect all relevant details currently retained by static index: refs, include paths, etc. (Doc comments are an open question - these are known to be very slow). Some changes are required here: refs are only indexed for the main file, but we want them for headers too.
Partitioning symbols by file is slightly tricky, symbols may be redeclared etc. As a first cut, we can emit a symbol both in its canonical declaration location, and its definition location (if distinct).
Later, we may want to avoid merging Symbol data from redecls in different files, as this cross-file leakage causes some instability of output. This couples nicely with filename-based filtering *before* constructing Symbols, which will be more efficient.

Open questions:

- #refs is an important code completion signal. Currently, it relies on every TU with a reference to a symbol emitting a copy of it with #refs = 1, and then merging all the copies. How do we replace this? (Maybe Dex itself should fill this in based on Refs data?)

Lightweight indexing

The motivating assumption here is that recently edited files are relevant when working in other files, so keeping e.g. definition locations up-to-date is important. Obvious example: working on a .h/.cpp file pair.

---

[3] Alternative: in a temporary directory (too hidden from user?). Alternative: in the user's CMake external build tree; read the compile_commands.json symlink (hard to generalize?)

The implementation is very similar to the current main-file/preamble indexing split within the dynamic index: the lightweight index can be build for almost-free from the diagnostics AST, and overlaid on the automatic index. The main difference is that there's more overlap: the automatic index will have e.g. refs for the main file, so MergedIndex will have to do a little more work to reconcile them.

Open questions:

- How important is this? Can we ship without it?

Minimal version and validation

The least invasive version is to add the indexer queue, building from the compilation database only, and use it as a replacement for the static index. (Leaving the dynamic index in place). We'd use the static indexer action unchanged, and partition the output.
This should mostly work, with limitations:

- it won't update as the user edits, but the dynamic index will keep the visible state up to date
- there's obvious duplication between the static index and dynamic preamble index
- partitioning/filtering the output of the indexer action isn't the most efficient approach
- there's no solution to the #refs problem, so ranking will suffer
- refs will be duplicated between dynamic and automatic index
- without persistence, we start from scratch every time

It should be sufficient for us to tell whether this approach is able to build a full index in a reasonable amount of time, and whether cross-refs etc work.