Clang compiler frontend

Ivan Murashko

May 7, 2022

Contents

Chapter 1

Introduction

The book describes clang compiler frontend internals. Architecture design is described and also practical code examples provided [2].

Chapter 2

Environment setup

The chapter describes basic steps to be done to setup the environment be used for future experiments with clang.

2.1 Source code compilation

we are going to compile our source code in debug mode to be suitable for future investigations with debugger.

2.1.1 Getting the source code

The clang source is a part of LLVM. You can get it with the following command

```
git clone https://github.com/llvm/llvm-project.git
cd llvm-project
```

2.1.2 Configuration with cmake

Create a build folder where the compiler and related tools will be built

```
mkdir build
cd build
```

Run configure script

```
cmake -G Ninja -DCMAKE_BUILD_TYPE=RelWithDebInfo
↪  -DLLVM_TARGETS_TO_BUILD="X86"
↪  -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra"
↪  -DLLVM_USE_LINKER=gold -DLLVM_USE_SPLIT_DWARF=ON ../llvm
```

The are several options specified:

- `-DLLVM_TARGETS_TO_BUILD="X86"` specifies exact targets to be build. It will avoid build unnecessary targets
- `LLVM_ENABLE_PROJECTS="clang;clang-tools-extra"` specifies LLVM projects that we care about
- `LLVM_USE_LINKER=gold` - uses gold linker
- `LLVM_USE_SPLIT_DWARF=ON` - spits debug information into separate files. This option saves disk space as well as memory consumption during the LLVM build. The option require compiler used for clang build to support it

2.1.3 Build

The build is trivial

```
ninja clang
```

You can also run unit and end-to-end tests for the compiler with

```
ninja check-clang
```

The compiler binary can be found as `bin/clang` at the build folder.

Chapter 3

Architecture

You can find some info about clang internal architecture and relation with other LLVM components.

3.1 Clang and clang driver

When we spoke about **clang** we should separate 2 things:

- driver
- compiler frontend

Both of them are called **clang** but perform different operations. The driver invokes different stages of compilation process (see fig. 3.1). The stages are standard for ordinary compiler and nothing special is there:

- Frontend: it does lexical analysis and parsing.
- Middle-end: it does different optimization on the intermediate representation (LLVM-IR) code
- Backend: Native code generation
- Assembler: Running assembler

Figure 3.1: Clang driver

- Linker: Running linker

The driver is invoked by the following command

```
clang main.cpp -o main -lstdc++
```

The driver also adds a lot of additional arguments, for instance search paths for system includes that could be platform specific. You can use `-###` clang option to print actual command line used by the driver

```
$clang -### main.cpp -o main -lstdc++
clang version 12.0.1 (Fedora 12.0.1-1.fc34)
Target: x86_64-unknown-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
 "/usr/bin/clang-12" "-cc1" "-triple"
 ↪  "x86_64-unknown-linux-gnu" "-emit-obj" "-mrelax-all" ...
```

One may see that the clang compiler toolchain corresponds the pattern wildly described at different compiler books [3]. Despite the fact, the frontend part is quite different from a typical compiler frontend described at the books. The primary reason for this is C++ language and it's complexity. Some features (macros) can change the source code itself another (typedef) can effect on token kind. As result the relations between different frontend components can be shown as follows

Chapter 4

Features

You can find some info about different clang features at the chapter

4.1 Precompiled headers

Precompiled headers or **pch** is a clang feature that was designed with the goal to improve clang frontend performance. The basic idea was to create AST for a header file and reuse the AST for some purposes.

4.1.1 User guide

Generate you pch file is simple [1]. Suppose you have a header file with name **header.h**:

```
#pragma once

void foo() {
}
```

then you can generate a pch for it with

```
clang -x c++-header header.h -o header.pch
```

the option `-x c++-header` was used there. The option says that the header file has to be treated as a c++ header file. The output file is `header.pch`.

The precompiled headers generation is not enough and you may want to start using them. Typical C++ source file that uses the header may look like

```cpp
// test pchs

#include "header.h"

int main() {
  foo();
  return 0;
}
```

As you may see, the header is included as follows

```cpp
  ...
#include "header.h"
  ...
```

By default clang will not use a pch at the case and you have to specify it explicitly with

```
clang -include-pch header.pch main.cpp -o main -lstdc++
```

We can check the command with debugger and it will give us

```
$ lldb ~/local/llvm-project/build/bin/clang -- -cc1
 ↪  -include-pch header.pch main.cpp -fsyntax-only
...
(lldb) b clang::ASTReader::ReadAST
...
(lldb) r
...
   4231   llvm::SaveAndRestore<SourceLocation>
-> 4232     SetCurImportLocRAII(CurrentImportLoc, ImportLoc);
   4233   llvm::SaveAndRestore<Optional<ModuleKind>>
    ↪  SetCurModuleKindRAII(
   4234       CurrentDeserializingModuleKind, Type);
   4235
(lldb) p FileName
(llvm::StringRef) $0 = (Data = "header.pch", Length = 10)
```

Note that only the first `--include-pch` option will be processed, all others will be ignored. It reflects the fact that there can be only one precompiled header for a translation unit.

4.2 Modules

Modules can be considered as a next step in evolution of precompiled headers. They also represent an parsed AST in binary form but form a DAG (tree) i.e. one module can include more than one another module [1]

4.2.1 User guide

4.2.2 Module map file

The key point for clang modules is `modulemap` file. It describes relation between different modules and interface provided by the modules. The default name for the file is `module.modulemap`. Typical content is the following

```
module header1 {
  header "header1.h"
  export *
}
```

The modules can be divided into 2 main sections

1. Explicit modules
2. Implicit modules

Explicit modules

We will start with explicit modules first.

Implicit modules

4.3 Header-Map files

TBD

[1] Compare that with precompiled header where only one precompiled header can be introduced for each compilation unit

Chapter 5

Tools

There are some tools created on clang related libs

5.1 clang-tidy

TBD

5.2 clangd

TBD

Index

Bibliography

[1] Clang compiler user's manual. — 2022. — https://clang.llvm.org/docs/UsersManual.html.

[2] Murashko, I. Source code examples for clang compiler fronmtend. — 2022. — https://github.com/ivanmurashko/articles/clangbook/src.

[3] Torczon, L. Engineering A Compiler / Linda Torczon, Keith Cooper. — 2nd edition. — Elsevier Inc., 2012.