# 简述LLVM与Clang及其关系

随着 Android P 的逐步应用，越来越多的客户要求编译库时用 libc++ 来代替 libstdc++。libc++ 和 libstdc++ 这两个库有关系呢？它们两个都是 C++ 标准库，libc++ 是针对 Clang 编译器特别重写的 C++ 标准库，而 libstdc++ 则是 GCC 的对应 C++ 标准库了。从 Android 市场来说，Android NDK 已在具体应用中放弃了 GCC，全面转向 Clang，正如很早前 Android NDK 在 Changelog 中提到的那样：

Android NDK 从 r11 开始建议大家切换到 Clang，并且把 GCC 标记为 deprecated，将 GCC 版本锁定在 GCC 4.9 不再更新；

Android NDK 从 r13 起，默认使用 Clang 进行编译，但是暂时也没有把 GCC 删掉，Google 会一直等到 libc++ 足够稳定后再删掉 GCC；

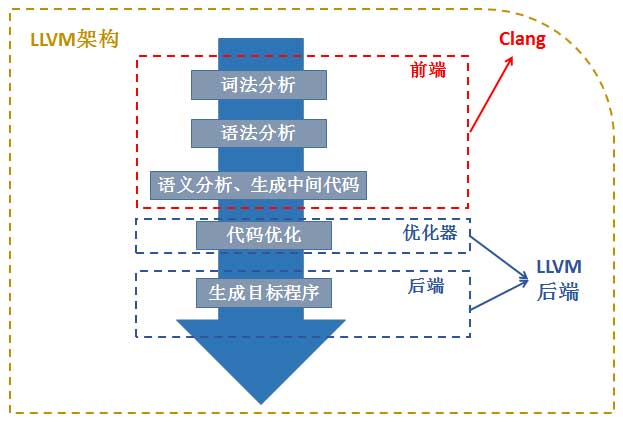
Android NDK 在 r17 中宣称不再支持 GCC 并在后续的 r18 中删掉 GCC，具体可见 NDK 的版本历史。

接下来，简要的介绍一下 Clang。Clang 是一个 C、C++、Objective-C 和 Objective-C++ 编程语言的编译器前端，采用底层虚拟机（LLVM）作为后端。至于为什么有了 GCC 还要开发 Clang？Clang 相比 GCC 又有什么优势呢？网上有很多信息可以参考，这里只简单提两点：（1）Clang 采用的是 BSD 协议的许可证，而 GCC 采用的是 GPL 协议，显然前者更为宽松；（2）Clang 是一个高度模块化开发的轻量级编译器，编译速度快、占用内存小、有着友好的出错提示。

然后说下 Clang 背后的 LLVM（Low Level Virtual Machine）。LLVM 是以 BSD 许可来开发的开源的编译器框架系统，基于 C++ 编写而成，利用虚拟技术来优化以任意程序语言编写的程序的编译时间、链接时间、运行时间以及空闲时间，最早以 C/C++ 为实现对象，对开发者保持开放，并兼容已有脚本。LLVM 计划启动于2000年，最初由University of Illinois at Urbana-Champaign的Chris Lattner主持开展，2006年 Chris Lattner 加盟苹果公司并致力于 LLVM 在苹果公司开发体系中的应用，所以苹果公司也是 LLVM 计划的主要资助者。目前 LLVM 因其宽松的许可协议，更好的模块化、更清晰的架构，成为很多厂商或者组织的选择，已经被苹果 IOS 开发工具、Facebook、Google 等各大公司采用，像 Swift、Rust 等语言都选择了以 LLVM 为后端。

在理解 LLVM 之前，先说下传统编译器的工作原理，基本上都是三段式的，可以分为前端、优化器和后端。前端负责解析源代码，检查语法错误，并将其翻译为抽象的语法树；优化器对这一中间代码进行优化，试图使代码更高效；后端则负责将优化器优化后的中间代码转换为目标机器的代码，这一过程后端会最大化的利用目标机器的特殊指令，以提高代码的性能。基于这个认知，我们可以认为 LLVM 包括了两个概念：一个广义的 LLVM 和一个狭义的 LLVM 。广义的 LLVM 指的是一个完整的 LLVM 编译器框架系统，包括了前端、优化器、后端、众多的库函数以及很多的模块；而狭义的 LLVM 则是聚焦于编译器后端功能的一系列模块和库，包括代码优化、代码生成、JIT 等。

下面大概讲一讲 LLVM 和 Clang 的关系。我们将它们对应于传统的编译器当中的几个独立的部分，这样能够更加方便明确的表述出它们之前的关系。



对应到这个图中，可以非常明确的找出它们的关系。整体的编译器架构就是 LLVM 架构；Clang 大致可以对应到编译器的前端，主要处理一些和具体机器无关的针对语言的分析操作；编译器的优化器和后端部分就是之前提到的 LLVM 后端，即狭义的 LLVM。

此外，由于LLVM的命名最早源自于底层虚拟机（Low Level Virtual Machine） 的首字母缩写，但这个项目的范围并不局限于创建一个虚拟机，这个缩写导致了大量的疑惑。LLVM 成长之后已成为众多编译工具及低级工具技术的统称，使得这个名字变得更不贴切，所以开发者决定放弃这个缩写的涵义，现在 LLVM 已独立成为一个品牌，适用于 LLVM 下的所有项目，包括 LLVM 中介码、LLVM 除错工具、LLVM C++ 标准库等。

# [Clang 15.0.0 git documentation](https://clang.llvm.org/docs/index.html)

https://clang.llvm.org/docs/LibTooling.html

# LibTooling

LibTooling is a library to support writing standalone tools based on Clang. This document will provide a basic walkthrough of how to write a tool using LibTooling.

For the information on how to setup Clang Tooling for LLVM see [**How To Setup Clang Tooling For LLVM**](https://clang.llvm.org/docs/HowToSetupToolingForLLVM.html)

## Introduction

Tools built with LibTooling, like Clang Plugins, run FrontendActions over code.

In this tutorial, we’ll demonstrate the different ways of running Clang’s SyntaxOnlyAction, which runs a quick syntax check, over a bunch of code.

## Parsing a code snippet in memory

If you ever wanted to run a FrontendAction over some sample code, for example to unit test parts of the Clang AST, runToolOnCode is what you looked for. Let me give you an example:

#include *"clang/Tooling/Tooling.h"*

TEST(runToolOnCode, CanSyntaxCheckCode) {

*// runToolOnCode returns whether the action was correctly run over the*

*// given code.*

EXPECT\_TRUE(runToolOnCode(std::make\_unique<clang::SyntaxOnlyAction>(), "class X {};"));

}

## Writing a standalone tool

Once you unit tested your FrontendAction to the point where it cannot possibly break, it’s time to create a standalone tool. For a standalone tool to run clang, it first needs to figure out what command line arguments to use for a specified file. To that end we create a CompilationDatabase. There are different ways to create a compilation database, and we need to support all of them depending on command-line options. There’s the CommonOptionsParser class that takes the responsibility to parse command-line parameters related to compilation databases and inputs, so that all tools share the implementation.

Parsing common tools options

CompilationDatabase can be read from a build directory or the command line. Using CommonOptionsParser allows for explicit specification of a compile command line, specification of build path using the -p command-line option, and automatic location of the compilation database using source files paths.

#include *"clang/Tooling/CommonOptionsParser.h"*

#include *"llvm/Support/CommandLine.h"*

**using** **namespace** clang::tooling;

*// Apply a custom category to all command-line options so that they are the*

*// only ones displayed.*

**static** llvm::cl::OptionCategory MyToolCategory("my-tool options");

int main(int argc, **const** char \*\*argv) {

*// CommonOptionsParser constructor will parse arguments and create a*

*// CompilationDatabase. In case of error it will terminate the program.*

CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);

*// Use OptionsParser.getCompilations() and OptionsParser.getSourcePathList()*

*// to retrieve CompilationDatabase and the list of input file paths.*

}

Creating and running a ClangTool

Once we have a CompilationDatabase, we can create a ClangTool and run our FrontendAction over some code. For example, to run the SyntaxOnlyAction over the files “a.cc” and “b.cc” one would write:

*// A clang tool can run over a number of sources in the same process...*

std::vector<std::string> Sources;

Sources.push\_back("a.cc");

Sources.push\_back("b.cc");

*// We hand the CompilationDatabase we created and the sources to run over into*

*// the tool constructor.*

ClangTool Tool(OptionsParser.getCompilations(), Sources);

*// The ClangTool needs a new FrontendAction for each translation unit we run*

*// on. Thus, it takes a FrontendActionFactory as parameter. To create a*

*// FrontendActionFactory from a given FrontendAction type, we call*

*// newFrontendActionFactory<clang::SyntaxOnlyAction>().*

int result = Tool.run(newFrontendActionFactory<clang::SyntaxOnlyAction>().get());

Putting it together — the first tool

Now we combine the two previous steps into our first real tool. A more advanced version of this example tool is also checked into the clang tree at tools/clang-check/ClangCheck.cpp.

*// Declares clang::SyntaxOnlyAction.*

#include *"clang/Frontend/FrontendActions.h"*

#include *"clang/Tooling/CommonOptionsParser.h"*

#include *"clang/Tooling/Tooling.h"*

*// Declares llvm::cl::extrahelp.*

#include *"llvm/Support/CommandLine.h"*

**using** **namespace** clang::tooling;

**using** **namespace** llvm;

*// Apply a custom category to all command-line options so that they are the*

*// only ones displayed.*

**static** cl::OptionCategory MyToolCategory("my-tool options");

*// CommonOptionsParser declares HelpMessage with a description of the common*

*// command-line options related to the compilation database and input files.*

*// It's nice to have this help message in all tools.*

**static** cl::extrahelp CommonHelp(CommonOptionsParser::HelpMessage);

*// A help message for this specific tool can be added afterwards.*

**static** cl::extrahelp MoreHelp("**\n**More help text...**\n**");

int main(int argc, **const** char \*\*argv) {

CommonOptionsParser OptionsParser(argc, argv, MyToolCategory);

ClangTool Tool(OptionsParser.getCompilations(),

OptionsParser.getSourcePathList());

**return** Tool.run(newFrontendActionFactory<clang::SyntaxOnlyAction>().get());

}

Running the tool on some code

When you check out and build clang, clang-check is already built and available to you in bin/clang-check inside your build directory.

You can run clang-check on a file in the llvm repository by specifying all the needed parameters after a “--” separator:

$ cd /path/to/source/llvm

$ export BD=/path/to/build/llvm

$ $BD/bin/clang-check tools/clang/tools/clang-check/ClangCheck.cpp -- **\**

clang++ -D\_\_STDC\_CONSTANT\_MACROS -D\_\_STDC\_LIMIT\_MACROS **\**

-Itools/clang/include -I$BD/include -Iinclude **\**

-Itools/clang/lib/Headers -c

As an alternative, you can also configure cmake to output a compile command database into its build directory:

*# Alternatively to calling cmake, use ccmake, toggle to advanced mode and*

*# set the parameter CMAKE\_EXPORT\_COMPILE\_COMMANDS from the UI.*

$ cmake -DCMAKE\_EXPORT\_COMPILE\_COMMANDS=ON .

This creates a file called compile\_commands.json in the build directory. Now you can run **clang-check** over files in the project by specifying the build path as first argument and some source files as further positional arguments:

$ cd /path/to/source/llvm

$ export BD=/path/to/build/llvm

$ $BD/bin/clang-check -p $BD tools/clang/tools/clang-check/ClangCheck.cpp

Builtin includes

Clang tools need their builtin headers and search for them the same way Clang does. Thus, the default location to look for builtin headers is in a path $(dirname /path/to/tool)/../lib/clang/3.3/include relative to the tool binary. This works out-of-the-box for tools running from llvm’s toplevel binary directory after building clang-resource-headers, or if the tool is running from the binary directory of a clang install next to the clang binary.

Tips: if your tool fails to find stddef.h or similar headers, call the tool with -v and look at the search paths it looks through.

Linking

For a list of libraries to link, look at one of the tools’ CMake files (for example [**clang-check/CMakeList.txt**](https://github.com/llvm/llvm-project/blob/main/clang/tools/clang-check/CMakeLists.txt)).

# How To Setup Clang Tooling For LLVM

Clang Tooling provides infrastructure to write tools that need syntactic and semantic information about a program. This term also relates to a set of specific tools using this infrastructure (e.g. clang-check). This document provides information on how to set up and use Clang Tooling for the LLVM source code.

## Introduction

Clang Tooling needs a compilation database to figure out specific build options for each file. Currently it can create a compilation database from the compile\_commands.json file, generated by CMake. When invoking clang tools, you can either specify a path to a build directory using a command line parameter -p or let Clang Tooling find this file in your source tree. In either case you need to configure your build using CMake to use clang tools.

## Setup Clang Tooling Using CMake and Make

If you intend to use make to build LLVM, you should have CMake 2.8.6 or later installed (can be found [**here**](https://cmake.org/)).

First, you need to generate Makefiles for LLVM with CMake. You need to make a build directory and run CMake from it:

**$** mkdir your/build/directory

**$** cd your/build/directory

**$** cmake -DCMAKE\_EXPORT\_COMPILE\_COMMANDS=ON path/to/llvm/sources

If you want to use clang instead of GCC, you can add -DCMAKE\_C\_COMPILER=/path/to/clang -DCMAKE\_CXX\_COMPILER=/path/to/clang++. You can also use ccmake, which provides a curses interface to configure CMake variables.

As a result, the new compile\_commands.json file should appear in the current directory. You should link it to the LLVM source tree so that Clang Tooling is able to use it:

**$** ln -s $PWD/compile\_commands.json path/to/llvm/source/

Now you are ready to build and test LLVM using make:

**$** make check-all

## Setup Clang Tooling Using CMake on Windows

For Windows developers, the Visual Studio project generators in CMake do not support [**CMAKE\_EXPORT\_COMPILE\_COMMANDS**](https://cmake.org/cmake/help/latest/variable/CMAKE_EXPORT_COMPILE_COMMANDS.html). However, the Ninja generator does support this variable and can be used on Windows to generate a suitable compile\_commands.json that invokes the MSVC compiler.

First, you will need to install [**Ninja**](https://ninja-build.org/). Once installed, the Ninja executable will need to be in your search path for CMake to locate it.

Next, assuming you already have Visual Studio installed on your machine, you need to have the appropriate environment variables configured so that CMake will locate the MSVC compiler for the Ninja generator. The [**documentation**](https://docs.microsoft.com/en-us/cpp/build/building-on-the-command-line?view=msvc-170#path_and_environment) describes the necessary environment variable settings, but the simplest thing is to use a [**developer command-prompt window**](https://docs.microsoft.com/en-us/cpp/build/building-on-the-command-line?view=msvc-170#developer_command_prompt_shortcuts) or call a [**developer command file**](https://docs.microsoft.com/en-us/cpp/build/building-on-the-command-line?view=msvc-170#developer_command_file_locations) to set the environment variables appropriately.

Now you can run CMake with the Ninja generator to export a compilation database:

C:\> mkdir build-ninja

C:\> cd build-ninja

C:\build-ninja> cmake -G Ninja -DCMAKE\_EXPORT\_COMPILE\_COMMANDS=ON path/to/llvm/sources

It is best to keep your Visual Studio IDE build folder separate from the Ninja build folder. This prevents the two build systems from negatively interacting with each other.

Once the compile\_commands.json file has been created by Ninja, you can use that compilation database with Clang Tooling. One caveat is that because there are indirect settings obtained through the environment variables, you may need to run any Clang Tooling executables through a command prompt window created for use with Visual Studio as described above. An alternative, e.g. for using the Visual Studio debugger on a Clang Tooling executable, is to ensure that the environment variables are also visible to the debugger settings. This can be done locally in Visual Studio’s debugger configuration locally or globally by launching the Visual Studio IDE from a suitable command-prompt window.

## Using Clang Tools

After you completed the previous steps, you are ready to run clang tools. If you have a recent clang installed, you should have clang-check in $PATH. Try to run it on any .cpp file inside the LLVM source tree:

**$** clang-check tools/clang/lib/Tooling/CompilationDatabase.cpp

If you’re using vim, it’s convenient to have clang-check integrated. Put this into your .vimrc:

function! ClangCheckImpl(cmd)

if &autowrite | wall | endif

echo "Running " . a:cmd . " ..."

let l:output = system(a:cmd)

cexpr l:output

cwindow

let w:quickfix\_title = a:cmd

if v:shell\_error != 0

cc

endif

let g:clang\_check\_last\_cmd = a:cmd

endfunction

function! ClangCheck()

let l:filename = expand('%')

if l:filename =~ '\.\(cpp\|cxx\|cc\|c\)$'

call ClangCheckImpl("clang-check " . l:filename)

elseif exists("g:clang\_check\_last\_cmd")

call ClangCheckImpl(g:clang\_check\_last\_cmd)

else

echo "Can't detect file's compilation arguments and no previous clang-check invocation!"

endif

endfunction

nmap <silent> <F5> :call ClangCheck()<CR><CR>

When editing a .cpp/.cxx/.cc/.c file, hit F5 to reparse the file. In case the current file has a different extension (for example, .h), F5 will re-run the last clang-check invocation made from this vim instance (if any). The output will go into the error window, which is opened automatically when clang-check finds errors, and can be re-opened with : cope.

Other clang-check options that can be useful when working with clang AST:

* -ast-print — Build ASTs and then pretty-print them.
* -ast-dump — Build ASTs and then debug dump them.
* -ast-dump-filter=<string> — Use with -ast-dump or -ast-print to dump/print only AST declaration nodes having a certain substring in a qualified name. Use -ast-list to list all filterable declaration node names.
* -ast-list — Build ASTs and print the list of declaration node qualified names.

Examples:

**$** clang-check tools/clang/tools/clang-check/ClangCheck.cpp -ast-dump -ast-dump-filter ActionFactory::newASTConsumer

Processing: tools/clang/tools/clang-check/ClangCheck.cpp.

Dumping ::ActionFactory::newASTConsumer:

clang::ASTConsumer \*newASTConsumer() (CompoundStmt 0x44da290 </home/alexfh/local/llvm/tools/clang/tools/clang-check/ClangCheck.cpp:64:40, line:72:3>

(IfStmt 0x44d97c8 <line:65:5, line:66:45>

<<<NULL>>>

(ImplicitCastExpr 0x44d96d0 <line:65:9> '\_Bool':'\_Bool' <UserDefinedConversion>

...

**$** clang-check tools/clang/tools/clang-check/ClangCheck.cpp -ast-print -ast-dump-filter ActionFactory::newASTConsumer

Processing: tools/clang/tools/clang-check/ClangCheck.cpp.

Printing <anonymous namespace>::ActionFactory::newASTConsumer:

clang::ASTConsumer \*newASTConsumer() {

if (this->ASTList.operator \_Bool())

return clang::CreateASTDeclNodeLister();

if (this->ASTDump.operator \_Bool())

return clang::CreateASTDumper(nullptr /\*Dump to stdout.\*/,

this->ASTDumpFilter);

if (this->ASTPrint.operator \_Bool())

return clang::CreateASTPrinter(&llvm::outs(), this->ASTDumpFilter);

return new clang::ASTConsumer();

}

## Using Ninja Build System

Optionally you can use the [**Ninja**](https://ninja-build.org/) build system instead of make. It is aimed at making your builds faster. Currently this step will require building Ninja from sources.

To take advantage of using Clang Tools along with Ninja build you need at least CMake 2.8.9.

Clone the Ninja git repository and build Ninja from sources:

**$** git clone git://github.com/martine/ninja.git

**$** cd ninja/

**$** ./bootstrap.py

This will result in a single binary ninja in the current directory. It doesn’t require installation and can just be copied to any location inside $PATH, say /usr/local/bin/:

**$** sudo cp ninja /usr/local/bin/

**$** sudo chmod a+rx /usr/local/bin/ninja

After doing all of this, you’ll need to generate Ninja build files for LLVM with CMake. You need to make a build directory and run CMake from it:

**$** mkdir your/build/directory

**$** cd your/build/directory

**$** cmake -G Ninja -DCMAKE\_EXPORT\_COMPILE\_COMMANDS=ON path/to/llvm/sources

If you want to use clang instead of GCC, you can add -DCMAKE\_C\_COMPILER=/path/to/clang -DCMAKE\_CXX\_COMPILER=/path/to/clang++. You can also use ccmake, which provides a curses interface to configure CMake variables in an interactive manner.

As a result, the new compile\_commands.json file should appear in the current directory. You should link it to the LLVM source tree so that Clang Tooling is able to use it:

**$** ln -s $PWD/compile\_commands.json path/to/llvm/source/

Now you are ready to build and test LLVM using Ninja:

**$** ninja check-all

Other target names can be used in the same way as with make.