使用 VSCode 和 CMake 开发 STM32CubeMX 工程

Qiu Kunyuan
11913019@mail.sustech.edu.cn

Sep 26, 2024

Abstract

本文简要介绍了如何在 VSCode 环境下使用 CMake 和 arm-gcc 交叉编译工具链开发 STM32CubeMX 工程。首先，概述了环境配置，包括必要的工具链安装和 VSCode 插件配置。接着，详细阐述了 Clangd 和 Cpptools 语言服务器的配置方法，以及如何生成和使用编译数据库。文章还提供了解决编译数据库中缺失系统包含路径问题的策略，并讨论了项目和用户配置文件对源码树的影响。最后，介绍了如何配置编译、烧写和调试过程，包括使用 PyOCD 和 objcopy 工具。

1 在 STM32CubeMX 创建的 CMake 工程中使用 VSCode 写代码

1.1 环境配置

需要事先安装到环境变量的工具:

1. arm-none-eabi 交叉编译工具链
   1. xPack gcc
   2. arm-llvm
   3. arm 官方提供 gcc(不建议)
2. STM32CubeCLT(用于获取 SVD 文件)
3. DAP 上位机
   1. OpenOCD
   2. PyOCD(需要先安装 python3)
4. CMake
5. busybox

1. GNU make for Windows64
2. ninja

除了独立安装上述工具之外, 也可以直接安装 PlatformIO IDE, 然后在 PlatformIO IDE 中安装 STM32 Platform, 这时便会在用户目录下下载上述的所有工具. 但是 PlatformIO 的构建系统可自定义参数很少, 无法通过手动修改 CMakeLists 的方式将`-isystem` 路径添加到 `compile_commandss.json` 中.

1.2 VSCode 配置

1.2.1 安装插件

1. Microsoft C/C++
2. Clangd
3. Microsoft CMake
4. CMake Language Support (1)
5. cortex-debug

(1) 在安装 CMake tools 插件后, 务必卸载 `twxs.cmake` 插件! 这个插件的代码提示能力基本为零, 而且还会 **aggressively** 地安装到 VSCode 中, 影响 CMake 文件的正常工作

1.2.2 Clangd 特性

编译数据库

编译数据库是由生成工具, 如 `bear make`、`cmake` 或 `ninja build` 等, 导出的 JSON 表, 包含每个源文件/头文件对应的绝对路径、编译选项、输出目录等结构化编译数据, 一般具有以下的格式:

```
[
  ...,
  {
  "directory": "D:/Electronics/stm32/lab/lab_led/build/Debug",
        "command":    "D:\\embedded_toolchains\\arm-none-eabi\\bin\\arm-none-eabi-gcc.exe    -DDEBUG    -DSTM32F103xE    -DUSE_HAL_DRIVER    -ID:/Electronics/stm32/lab/lab_led/cmake/stm32cubemx/../../Core/Inc    -IC:/Users/Falke/STM32Cube/Repository/STM32Cube_FW_F1_V1.8.6/Drivers/STM32F1xx_HAL_Driver/Inc    -IC:/Users/Falke/STM32Cube/Repository/STM32Cube_FW_F1_V1.8.6/Drivers/STM32F1xx_HAL_Driver/Inc/Legacy -IC:/Users/Falke/STM32Cube/Repository/STM32Cube_FW_F1_V1.8.6/Drivers/CMSIS/Device/ST/STM32F1xx/Include    -IC:/Users/Falke/STM32Cube/Repository/STM32Cube_FW_F1_V1.8.6/Drivers/CMSIS/Include -mcpu=cortex-m3  -Wall -Wextra -Wpedantic -fdata-sections -ffunction-sections -O0 -g3 -mcpu=cortex-m3   -Wall -Wextra -Wpedantic -fdata-sections -ffunction-sections -O0 -g3 -g -std=gnu11 -o CMakeFiles\\lab_led.dir\\Core\\Src\\gpio.c.obj -c D:\\Electronics\\stm32\\lab\\lab_led\\Core\\Src\\gpio.c",
  "file": "D:\\Electronics\\stm32\\lab\\lab_led\\Core\\Src\\gpio.c",
  "output": "CMakeFiles\\lab_led.dir\\Core\\Src\\gpio.c.obj"
 },
 ...
]
```

生成工具导出编译数据库一般通过直接解析项目文件或者拦截并记录编译命令的方式, 例如 `bear` 工具用下图所示的流程产生编译数据库:

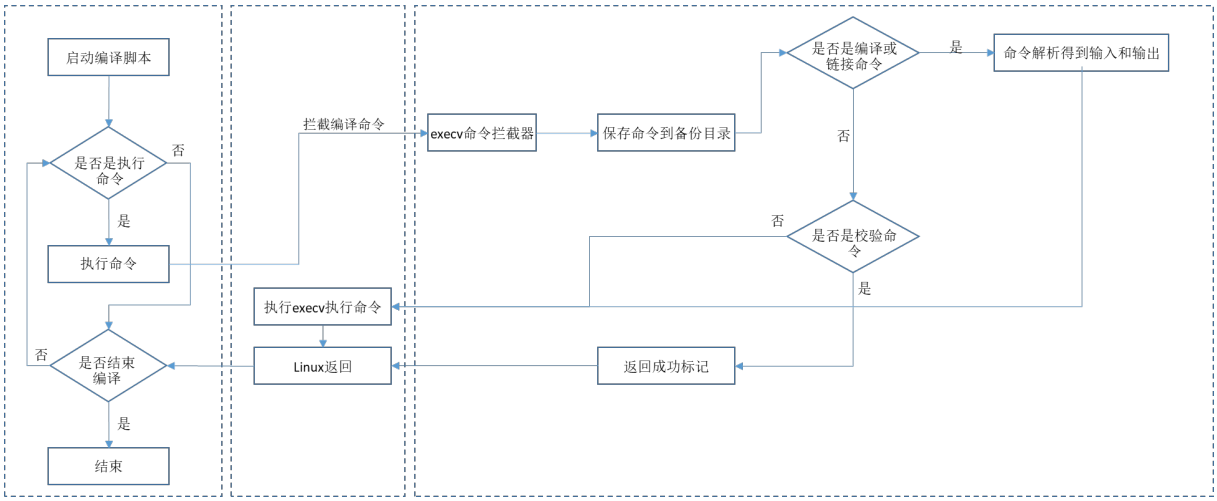

图 1　alt text

而 CMake 会一边解析 CMakeLists 一边生成编译数据库.

在使用 CMake 构建交叉编译工程时, 直接 `cmake build` 得到的编译数据库中缺少交叉编译器的 system includes 和 system root 信息, 导致 LS 找不到正确的标准库. 查阅 gcc 的 man page 得知可以用 `gcc -x c++ -Wp,-v <任意文件名>`获取 gcc 的 system include, 据此即可用 CMakeLists 执行外部命令行的方式获取这些 system include 目录:

```
if(CMAKE_EXPORT_COMPILE_COMMANDS)
    # This dreadful mess is to communicate to clang-tidy the C++ system includes. It seems that CMake
    # doesn't support using its own compile_commands.json database, and that clang-tidy doesn't
    # pick up non-default system headers.
    string(LENGTH ${TOOLCHAIN_PREFIX} PREFIX_LENGTH)
    MATH(EXPR PREFIX_LENGTH "${PREFIX_LENGTH}-1")
    string(SUBSTRING ${TOOLCHAIN_PREFIX} 0 ${PREFIX_LENGTH} TOOLCHAIN_TRIPLE)
    # above is specific for STM32CubeMX target triple variable
    execute_process(
            COMMAND cmd.exe /c ${CMAKE_C_COMPILER} -x c++ -Wp,-v 1 2>&1 | grep "^ .*${TOOLCHAIN_TRIPLE}.*"
            OUTPUT_VARIABLE COMPILER_HEADERS
            OUTPUT_STRIP_TRAILING_WHITESPACE
    )
    message(NOTICE "Find system includes: ${COMPILER_HEADERS}")
    string(REGEX REPLACE "[ \n\t]+" ";" INCLUDE_COMPILER_HEADERS ${COMPILER_HEADERS})

    set(CMAKE_C_STANDARD_INCLUDE_DIRECTORIES ${INCLUDE_COMPILER_HEADERS})
    set(CMAKE_CXX_STANDARD_INCLUDE_DIRECTORIES ${INCLUDE_COMPILER_HEADERS})
endif()
```

> 由于 `findstr.exe` 和 CMake `execute_process()` 命令存在兼容性问题, 因此应当使用 busybox 或者 gnuwin32 提供的 `grep.exe` 进行字符串查找.

在 CMakeLists 中进行上述配置后, 便可以尝试 F7 或者在终端中 `mkdir build && cd build && cmake .. -G "Unix Makefiles" && make -j` 编译项目. 这一步是为了检查 STM32CubeMX 生成的项目结构是否正确, 如果编译出错, 检查项目根目录下 `cmake/stm32cubemx/CMakeLists.txt` 中是否漏掉了 `Core/Src` 中的头文件、源文件, 以及宏定义是否正确.

配置文件影响的源码树

Clangd 语言服务器的配置文件分为 **项目配置文件** 和 **用户配置文件** 两种. 放置于项目根目录下的项目配置文件只能影响 Clangd 索引根目录以下源码树时的各种特性，而不能影响 Clangd 索引根目录以外源码树的特性. 因此, 如果在使用 CubeMX 生成代码时选择了 在工具链项目配置文件中添加必要的库文件作为引用 (Add necessary library files as reference in the toolchain project configuration file) 时, 将会导致 clangd 无法将 CMake 文件中的定义应用到通过 STM32CubeMX 安装的源码树, 因而无法正确显示各种条件编译段和头文件包含关系.

> Repeating from my last comment:
>
> > Project config files are scoped to apply only to files in the directory tree of the directory containing the config file.
>
> Since you reported originally that:
>
> > It works perfecly for any file in the project.
>
> > It doesn't work for system includes as or . For those files I get errors like this:
>
> So, if you have a project config file at `d:\Source\CodeDog\SWD\Software\SWD-STM32H745I-DISCO\.clangd`
>
> That **will apply to** `d:\Source\CodeDog\SWD\Software\SWD-STM32H745I-DISCO\SomeSubdirectory\SomeFile.cpp` (because that's in the directory tree `d:\Source\CodeDog\SWD\Software\SWD-STM32H745I-DISCO\`)
>
> It **will not apply to** `d:\source\embedded\gcc-arm-none-eabi-10.3-2021.10\arm-none-eabi\include\c++\10.3.1\string` (because that's not in the directory tree `d:\Source\CodeDog\SWD\Software\SWD-STM32H745I-DISCO\`)
>
> To apply to files outside the project directory, configuration needs to go into the **user** config file.

1.2.3 配置语言服务器插件

在打开工程之后, 不出意外的话你会看到源文件和头文件中充斥着大量的红波浪线. 这是因为 Cpptools 和 Clangd 均属于 语言服务器(Language Server, LS), 它们依据编译数据库中每个文件附带的编译选项, 对相应的源文件进行编译, 从而获得 AST 用于代码提示和错误检查.

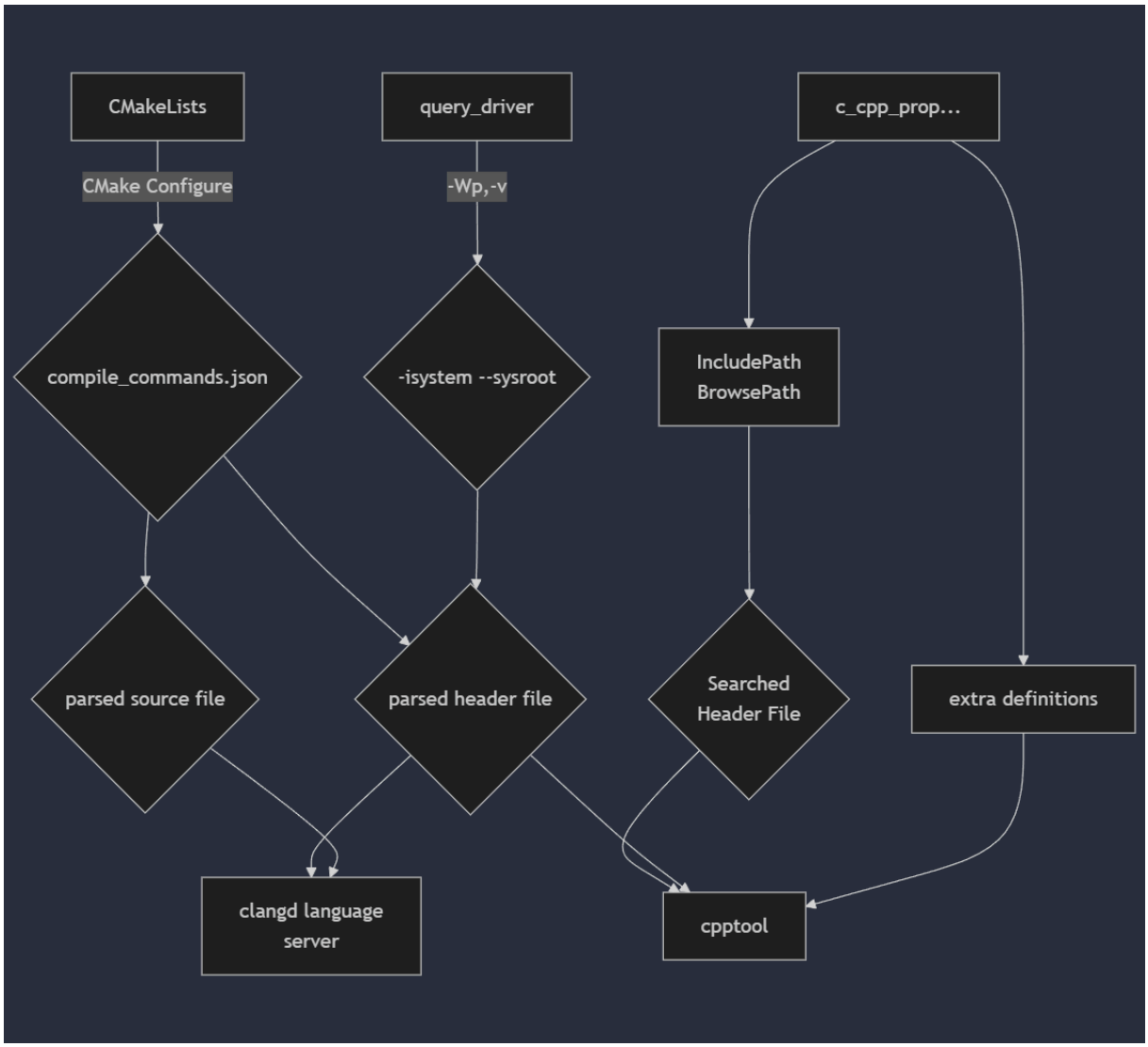

图 2　lsp_diagram

由于没有正确地配置项目, 导致这两个插件缺少必要的编译数据库(`compile_commands.json`)文件, 因此无法对头文件和库文件进行索引.

配置 CMake

STM32CubeMX 生成的 CMake 工程默认开启 `compile_commands.json` 的生成. 但是, 打开任意一个源文件后, 仍然会发现大量宏定义无法解析或解析错误, 有时还会报错缺少头文件, 或者在 STM32 工程中引用 VS2022 的头文件. 经过查阅大量文档和 issue tracker, 以下 3 个措施可以有效解决此问题:

1. 配置将 `compile_commands.json` 复制到项目根目录
2. 让 CMake Configure 时生成正确的 `sysroot`
3. 检查是否有 缺失的文件

由手册得知, Clangd 寻找编译数据库的方式不是从项目根目录向下遍历, 而是从目标文件向上遍历, 默认配置下生成到 `build` 或 `build/${mcuSeriesName}`目录下的编译数据库很难被 Clangd 找到. 因此, 可以通过 VSCode 扩展配置

```
{
  "cmake.copyCompileCommands": "${workspaceFolder}/compile_commands.json"
}
```

或者 CMake 配置

```
add_custom_target( # 增加一个没有输出的目标
  copy-compile-commands ALL # 增加到默认构建目标，使得它总是被构建
  ${CMAKE_COMMAND} -E copy_if_different
  ${CMAKE_BINARY_DIR}/compile_commands.json
  ${PROJECT_SOURCE_DIR}
  )
```

使得每次 `cmake build` 后 `compile_commands` 都会被复制到项目根目录.

配置 Clangd 和 Cpptool

Clangd 和 Cpptool 各有优缺点, 例如 Clangd 对源文件的解析能力更强, 可以准确解析多个文件嵌套的复杂宏定义, 而 Cpptool 在正确设置包含路径后找头文件的能力更强, 因此最好同时启用两个扩展.

为了让 Clangd 正确索引包含路径, 可以采取以下两个措施:

1. 配置 `query-driver`
2. 将合适的宏定义添加到 `.clangd` 和 `.clang-tidy` 文件中

以下 4 个措施能够让 Cpptool 采用正确的宏定义和包含路径:

1. 从 cmake 文件中找到正确的头文件目录(VSCode 头文件设置的大坑)
2. 手动配置正确的宏定义
3. 配置提供源和 compile_commands
4. 将 sysroot 正确传参到 cpptool 后端 clang-tidy

Clangd 使用 `query-driver` 命令行选项指定其后端, 当不指定时就会从环境变量中选择优先级最高(最靠前)的、且受 Clangd 支持的编译器. 为了让 Clangd 正确地找到交叉编译工具链的 `arm-none-eabi-gcc`, 需要手动指定这个 gcc 的名称和绝对路径.

微软 Cpptool 无法像它自己描述的那样通过`.../**`后缀对某个包含目录以下子目录进行递归搜索, 必须手动将每一级头文件目录添加到 `c_cpp_properties.json` 中. 这些头文件目录可以在`{项目根目录}/cmake/stm32cubemx/CMakeLists.txt` 中找到.

1.2.4 配置文件

综上所述, 正确(90% 概率正常工作)的语言服务器配置包含以下 4 项, 可以在附录中找到:

1. 工作区设置(小节 A.1)
2. clangd 设置(小节 A.3)
3. cpptool 设置(小节 A.2)
4. clangtidy 设置(小节 A.4)

1.3 编译烧写配置

由于没有找到合适的命令行 ISP 烧写工具, 因此主要介绍利用 ST-Link 或者 JLink(CMSIS-DAP) 调试器烧写. 这些调试器均可用 OpenOCD 或 PyOCD 作为上位机, 用 SWD 或 4 线 JTAG 连上 MCU 之后即可在命令行控制烧录和调试.

1.3.1 PyOCD

烧录命令:

```
pyocd flash -t {目标代号} .\build\Debug\lab_led.elf
```

调试配置:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Cortex Debug",
      "cwd": "${workspaceFolder}",
      "executable": "${workspaceFolder}/build/Debug/{CMake 项目名称}.elf",
      "targetId":"{目标代号}",
      "request": "launch",
      "type": "cortex-debug",
      "runToEntryPoint": "main",
      "servertype": "pyocd",
      "svdPath": "{你的 STM32CubeCLT 安装目录}/STMicroelectronics_CMSIS_SVD/STM32F103.svd"
    }
  ]
}
```

获取目标代号时, 可以使用 `pyocd list -t | grep {regex}`在结果中缩小范围. 例如在知道大概是 STM32F103RCT6 的情况下, 可以作以下的尝试:

1. 搜索 `pyocd list -t | grep stm32f103` -> 返回大量 f103 系列型号
2. 搜索 `pyocd list -t | grep stm32f103r.*` -> 返回符合该正则表达式的 `stm32f103rX`
3. 肉眼观察选中 `stm32f103rc`

1.3.2 elf 转 hex

使用 objcopy 工具将 elf 格式可执行文件转换为 hex 格式.

```
objcopy -O ihex {CMake 项目名称}.elf {CMake 项目名称}.hex
```

APPENDIX A 主要配置文件

A.1 工作区定义文件 *.code-workspace:

```
{
  ...,
  "settings": {
    "typescript.tsc.autoDetect": "off",
    "clangd.arguments": [
      "--all-scopes-completion",
      "--background-index",
      "--background-index-priority=low",
      "--completion-parse=auto",
      "--completion-style=detailed",
      "--enable-config",
      "--header-insertion-decorators",
      "--header-insertion=iwyu",
      "--pch-storage=memory",
      "--parse-forwarding-functions",
      "--query-driver=<你的工具链根目录>/bin/arm-none-eabi-gcc,<你的工具链根目录>/bin/arm-none-eabi-g++",
      "--ranking-model=decision_forest",
      "-j=24",
      "--log=error"
    ],
    "files.associations": {
      ".clang-format": "yaml",
      ".clang-tidy": "yaml",
      ".clangd": "yaml"
    },
    "cmake.copyCompileCommands": "${workspaceFolder}/compile_commands.json",
    "C_Cpp.default.browse.limitSymbolsToIncludedHeaders": true,
    "clangd.detectExtensionConflicts": false,
    "C_Cpp.intelliSenseEngine": "default",
    ...
  }
}
```

A.2 微软 Cpptool 设置文件 c_cpp_properties.json:

```
{
  "configurations": [
    {
      "name": "Win32",
      "includePath": [
        "{来自 CMakeLists.txt 的头文件路径}"
      ],
      "browse": {
        "limitSymbolsToIncludedHeaders": false,
        "path": [
          "{来自 CMakeLists.txt 的头文件路径}"
        ]
      },
      "defines": [
        "USE_HAL_DRIVER",
        "STM32F103xE"
      ],
      "configurationProvider": "ms-vscode.cmake-tools",
```

```
      "compileCommands": "${workspaceFolder}/compile_commands.json",
      "compilerPath": "D:/embedded_toolchains/arm-none-eabi/bin/arm-none-eabi-gcc.exe"
    }
  ],
  "version": 4
}
```

A.3 Clangd 配置文件`.clangd`:

```
CompileFlags:
  # Add: ["-Wall", "-Wextra", "-Wpedantic", "-ferror-limit=0"]
  # Remove: [-xc++,-W*, "-resource-dir*", "-ferror-limit*"]
  Compiler: arm-none-eabi-gcc
Diagnostics:
  Includes:
    AnalyzeAngledIncludes: true
    MissingIncludes: Strict
Index:
  StandardLibrary: Yes
```

A.4 微软 Cpptool 和 Clangd 共用配置文件`.clang-tidy`:

```
ExtraArgsBefore:
  [
    "--sysroot=...",
    "-I...",
    "-isystem",
    "...",
    "-isystem",
    "...",
    # 在 compile_commands.json 中找到对应的命令行选项, split(' ')后复制粘贴到此处
    "-DSTM32F103xE",
    "-DUSE_HAL_DRIVER"
  ]
```