



一、CMake安装

- 1、Linux平台安装
- 2、Windows平台

二、CMake初步使用

- 1、CMake基本概述
 - (1). CMake工具特性
 - (2). CMake构建基本内容
 - (3). CMake构建基本流程
- 2、CMake构建C/C++ 工程基本方法
 - (1). 单目录文件构建方法
 - (2). 多目录文件构建方法

三、CMake构建正式工程

1. 正式的代码组织结构
 - (1). 常见的代码组织结构
 - (2). 文件夹名称及含义
2. 生成可执行文件
 - (1). 单个CMakeLists.txt写法
 - (2). 多个CMakeLists.txt写法
3. 生成动态库和静态库
 - (1). 不同名的动态库和静态库
 - (2). 同名的动态库和静态库
4. 链接动态库和静态库
 - (1). 包含库文件
 - (2). 包含库的头文件
5. 添加编译和控制选项
 - (1). 添加编译选项
 - (2). 添加控制选项
6. 平台移植

- (1).系统平台匹配
- (2).编译环境设置
- (2).交叉编译

四、参考资料

一、CMake安装


1、Linux平台安装

```
apt-get install cmake
```

2、Windows平台

下载地址

```
https://cmake.org/download/
```



AboutServicesResourcesDownload

Follow the directions. The OS-machine.tar.gz files are gzipped tar files of the install tree. The OS-machine.tar.z files are compressed tar files of the install tree. The tar file distributions can be untared in any directory. They are prefixed by the version of CMake. For example, the linux-x86_64 tar file is all under the directory cmake-linux-x86_64. This prefix can be removed as long as the share, bin, man and doc directories are moved relative to each other. To build the source distributions, unpack them with zip or tar and follow the instructions in README.rst at the top of the source tree. See also the [CMake 3.24 Release Notes](#).

Source distributions:

| Platform | Files |
|---------------------------------------|-------------------------------------|
| Unix/Linux Source (has \n line feeds) | cmake-3.24.1.tar.gz |
| Windows Source (has \r\n line feeds) | cmake-3.24.1.zip |

Binary distributions:

| Platform | Files |
|--------------------------|--|
| Windows x64 Installer: | cmake-3.24.1-windows-x86_64.msi |
| Windows x64 ZIP | cmake-3.24.1-windows-x86_64.zip |
| Windows i386 Installer: | cmake-3.24.1-windows-i386.msi |
| Windows i386 ZIP | cmake-3.24.1-windows-i386.zip |
| Windows ARM64 Installer: | cmake-3.24.1-windows-arm64.msi |
| Windows ARM64 ZIP | cmake-3.24.1-windows-arm64.zip |
| macOS 10.13 or later | cmake-3.24.1-macos-universal.dmg |

安装完之后将CMake文件夹下的bin文件路径添加到环境变量中

| | | |
|---|----------------|-----|
|  bin | 2022/8/18 1:57 | 文件夹 |
|  doc | 2022/8/18 1:57 | 文件夹 |
|  man | 2022/8/18 1:57 | 文件夹 |
|  share | 2022/8/18 1:57 | 文件夹 |

二、CMake初步使用

1、CMake基本概述

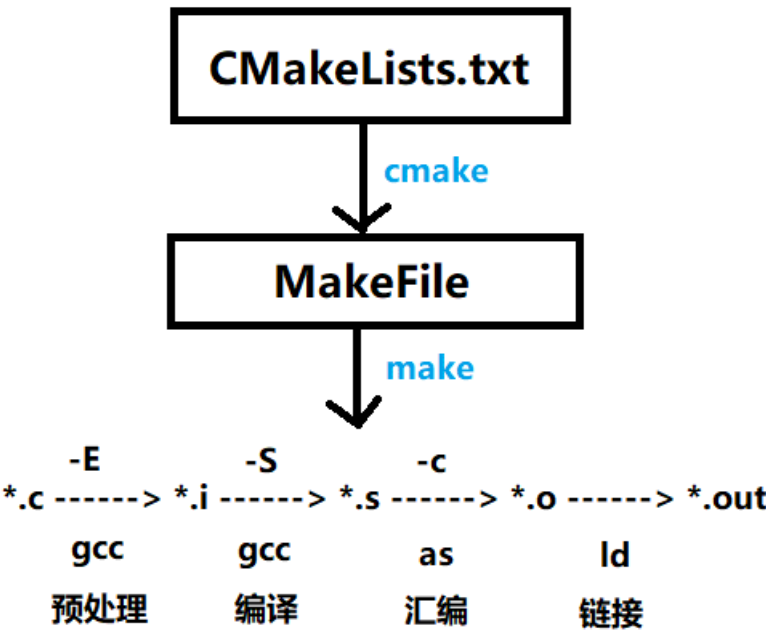
(1). CMake工具特性

cmake是跨平台的安装编译工具，只需要编写CMakeList.txt文件,能够自动生成工程文件和makefile文件，make编译工具的高阶。

当程序只有一个源文件时，可以直接使用gcc (或g++) 命令进行编译。但当程序包含多个源文件时，逐文件去编译，编译顺序可能出现混乱同时工作量较大。

-make工具可以看成是一个智能的批处理工具，它本身并没有编译和链接的功能，而是用类似于批处理的方式-通过调用makefile文件中用户指定的命令利用gcc (或g++) 来进行编译和链接。
-makefile在一些简单的工程中可以人工书写，但当工程较大时，手写makefile较为麻烦，同时更换平台需要修改makefile。

cmake工具可以根据CMakeLists.txt文件去生成makefile。



(2). CMake构建基本内容

| | | | |
|--|-----------------|----------|------|
|  CMakeLists.txt | 2022/8/24 11:47 | 文本文档 | 1 KB |
|  main.c | 2022/8/24 11:45 | C Source | 0 KB |

main.c文件内容：

```
#include <stdio.h>

int main(void)
{
    printf("Hello World\n");
    return 0;
}
```

CMakeLists.txt文件内容:

```
cmake_minimum_required (VERSION 2.8)
project (demo)
add_executable(main main.c)
```

终端命令:

```
cmake .
make
```

关于 Linux 的几个文件目录知识

- .或者 ./ 代表当前目录
- ..或者 ../ 代表上一级目录
- ../../ 代表上上一级目录


(3). CMake构建基本流程

- 1.编写 CMakeLists.txt 规则。
- 2.执行 cmake \$PATH 生成 Makefile(PATH 是CMakeLists.txt 所在的顶层目录)。
- 3.执行 make 进行编译。

2、CMake构建C/C++ 工程基本方法

(1). 单目录文件构建方法

单文件CMakelists.txt写法:

| | | | |
|--|-----------------|----------|------|
|  CMakeLists.txt | 2022/8/24 11:47 | 文本文档 | 1 KB |
|  main.c | 2022/8/24 11:45 | C Source | 0 KB |

CMakeLists.txt文件内容:







```
cmake_minimum_required (VERSION 2.8)
project (demo)
add_executable(main main.c)
```

第一行意思是表示cmake的最低版本要求是2.8。

第二行是表示本工程信息，也就是工程名叫demo。

第三行比较关键，表示最终要生成的可执行文件的名字叫main，使用的源文件是main.c。

多文件CMakeLists.txt写法：

| | | | |
|--|-----------------|--------------|------|
|  CMakeLists.txt | 2022/8/24 11:47 | 文本文档 | 1 KB |
|  main.c | 2022/8/25 9:21 | C Source | 1 KB |
|  subFunc1.c | 2022/8/25 9:21 | C Source | 1 KB |
|  subFunc1.h | 2022/8/25 9:21 | C/C++ Header | 1 KB |
|  subFunc2.c | 2022/8/25 9:21 | C Source | 1 KB |
|  subFunc2.h | 2022/8/25 9:21 | C/C++ Header | 1 KB |

- 方法1：手动添加所有源文件

```
cmake_minimum_required (VERSION 2.8)
project (demo)
add_executable(main main.c subFunc1.c subFunc2.c)
```

- 方法2：使用aux_source_directory命令添加源文件

```
aux_source_directory(<dir> <var>):
```

参数dir：表示所要添加源文件的子目录名称，如果使用 . 表示当前目录。

参数var：表示保存源文件信息的变量名称，一般默认写成 SRC_LIST 的形式。

方法1存在弊端，如果源文件比较多，在CMakeLists.txt中逐个添加源文件信息比较麻烦，可以使用 aux_source_directory命令把指定目录下所有的源文件存储在一个变量中，然后在 add_executable里通过 \${} 的形式引用该变量内容。

```
cmake_minimum_required (VERSION 2.8)
project (demo)
aux_source_directory(. SRC_LIST)
add_executable(main ${SRC_LIST})
```

- 方法3：使用set命令添加需要使用的源文件

```
set(<var> [filePath ...])
```

参数var：表示保存源文件信息的变量名称，一般默认写成 SRC_LIST 的形式。

参数filePath：表示需要添加的源文件路径地址。

如果并不需要把同一目录的所有源文件添加到一个变量中，可以使用set命令把需要用到的源文件添加进变量中。

```
cmake_minimum_required (VERSION 2.8)
project (demo)
set( SRC_LIST
    ./main.c
    ./subFunc1.c
    ./subFunc2.c)
add_executable(main ${SRC_LIST})
```

还可以使用`file``命令将指定文件夹内的指定后缀名的文件，存入一个指定变量。

```
file(GLOB <var> <dir>/*.c)
```

参数GLOB：预定义变量，表示产生一张与全局表达式匹配的所有文件的表，并存入指定变量。

参数var：表示变量名。

参数dir：指定目录路径。





举例：

```
file(GLOB SRC_FILES src/*.cc)
```

表示将src路径下的所有.cc文件，打包取个别名叫做变量SRC_FILES

此外还有参数GLOB_RECURSE，与GLOB类似，但会遍历匹配路径下面的所有子路径并匹配文件。

(2). 多目录文件构建方法

| | | | |
|--|-----------------|----------|------|
|  main.c | 2022/8/25 9:21 | C Source | 1 KB |
|  CMakeLists.txt | 2022/8/24 11:47 | 文本文档 | 1 KB |
|  subFunc2 | 2022/8/25 10:08 | 文件夹 | |
|  subFunc1 | 2022/8/25 10:08 | 文件夹 | |

```
|— main.c
|— CMakeLists.txt
|— subFunc2
    |— subFunc2.c
    |— subFunc2.h
|— subFunc1
    |— subFunc1.c
    |— subFunc1.h
```

多目录的情况下，头文件和和主函数不在一个目录下，主函数文件需要考虑如何引用头文件。

- 方法1：手动在主函数文件中通过include引用头文件位置

```
#include "subFunc1/subFunc1.h"
#include "subFunc2/subFunc2.h"
```

- 方法2：通过include_directories命令引用头文件所在的目录

```
include_directories([head_dir ...])
```

参数[head_dir]表示:头文件文件夹路径

方法1存在弊端,如果头文件比较多,在CMakeLists.txt中逐个添加源文件信息比较麻烦,可以使用 `include_directories` 命令添加指定目录下所有的头文件。

```
cmake_minimum_required (VERSION 2.8)
project (demo)
include_directories (subFunc1 subFunc2)
aux_source_directory (subFunc1 SRC_LIST1)
aux_source_directory (subFunc2 SRC_LIST2)
add_executable (main main.c ${SRC_LIST} ${SRC_LIST1})
```

三、CMake构建正式工程

1. 正式的代码组织结构






(1). 常见的代码组织结构

```
|— bin
|— lib
|— build
|— thirdparty
|— include
|— src
|— res
|— ...
```

(2). 文件夹名称及含义

bin文件夹: 一般表示存放项目生成的可执行文件。
lib文件夹: 一般表示存放项目生成的静态文件和动态文件。
bulid文件夹: 一般表示存放CMake构建项目时生成的临时文件。
thirdparty文件夹: 一般表示存放需要引用的第三方库文件。
include文件夹: 一般表示存放项目头文件。
src文件夹: 一般表示存放项目源文件。
res文件夹: 一般表示存放项目所需的资源文件。

2. 生成可执行文件

| | | | |
|--|-----------------|------|------|
|  bin | 2022/8/25 10:52 | 文件夹 | |
|  bulid | 2022/8/25 10:52 | 文件夹 | |
|  include | 2022/8/25 14:25 | 文件夹 | |
|  src | 2022/8/25 14:25 | 文件夹 | |
|  CMakeLists.txt | 2022/8/24 11:47 | 文本文档 | 1 KB |

(1). 单个CMakeLists.txt写法

```
|— bin
|— build
|— include
    |— subFunc2.h
    |— subFunc1.h
|—src
    |— subFunc2.c
    |— subFunc1.c
    |— main.c
|— CMakeLists.txt
```

```
cmake_minimum_required (VERSION 2.8)
project (demo)
set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
aux_source_directory (src SRC_LIST)
include_directories (include)
add_executable (main ${SRC_LIST})
```

EXECUTABLE_OUT_PATH 和 PROJECT_SOURCE_DIR 是CMake自带的预定义变量，其意义如下：

| 变量名 | 含义 |
|------------------------|-----------------|
| EXECUTABLE_OUTPUT_PATH | 目标二进制可执行文件的存放位置 |
| PROJECT_SOURCE_DIR | 工程的根目录 |

终端命令：

```
mkdir build
cd build
cmake ..
make
```

一般在build文件夹生成makefile等中间文件，防止这些中间文件污染项目的组织结构。如果编译出错，可直接删除build文件夹，该构建方法也被称为外部构建。

cmake命令区别：

| 命令 | 含义 |
|---------|------------------------------------|
| cmake . | 表示在 当前 目录下搜索CMakeList.txt文件，进行项目构建 |

`cmake ..` 表示在上一级目录下搜索CMakeLists.txt文件，进行项目构建

(2). 多个CMakeLists.txt写法

```
|— bin
|— build
|— include
    |— subFunc2.h
    |— subFunc1.h
|—src
    |— subFunc2.c
    |— subFunc1.c
    |— main.c
    |— CMakeLists.txt
|— CMakeLists.txt
```

- 最外层目录的CMakeLists.txt内容







```
cmake_minimum_required (VERSION 2.8)
project (demo)
add_subdirectory (src)
```

增加了一个`add_subdirectory`命令，这个命令可以向当前工程添加存放源文件的子目录，也可以指定中间二进制和目标二进制的存放位置。

- src目录的CMakeLists.txt内容

```
aux_source_directory (. SRC_LIST)
include_directories (../include)
add_executable (main ${SRC_LIST})
set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
```

3. 生成动态库和静态库

| | | | |
|--|-----------------|----------|------|
|  bulid | 2022/8/25 10:52 | 文件夹 | |
|  include | 2022/8/25 14:25 | 文件夹 | |
|  lib | 2022/8/25 16:25 | 文件夹 | |
|  src | 2022/8/25 17:53 | 文件夹 | |
|  main.c | 2022/8/25 9:21 | C Source | 1 KB |
|  CMakeLists.txt | 2022/8/24 11:47 | 文本文档 | 1 KB |

```
|— lib
|— build
|— include
    |— subFunc2.h
    |— subFunc1.h
|—src
    |— subFunc2.c
    |— subFunc1.c
|— main.c
|— CMakeLists.txt
```

(1). 不同名的动态库和静态库

使用`add_library`命令可以将是`src`目录内的源文件构建成动态库或静态库。

```
add_library(<libname> <libtype> [filepath ...])
```

参数`libname`: 表示生成的库名字。

参数`libtype`: 预定义变量

STATIC 静态库

SHARED 动态库

参数`filepath`: 表示源文件的路径。

```
cmake_minimum_required (VERSION 2.8)
project (demo)
set (SRC_LIST ${PROJECT_SOURCE_DIR}/subFunc/testFunc1.c
        ${PROJECT_SOURCE_DIR}/subFunc/testFunc2.c)
add_library (subFunc_shared SHARED ${SRC_LIST})
add_library (subFunc_static STATIC ${SRC_LIST})
set (LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
```

`LIBRARY_OUTPUT_PATH` 是CMake自带的预定义变量，表示库文件的存放位置。

(2). 同名的动态库和静态库

使用`add_library`命令只能生成不同名字的动态库和静态库，二者不可同名。如果同名的话，只会默认生成一种库文件。想让两种库文件同名，就要使用`set_target_properties`命令重命名。

```
set_target_properties (<oldlibname> PROPERTIES OUTPUT_NAME "
<newlibname>")
```

参数`oldlibname`: 表示需要重命名的库文件名称。

参数`newlibname`: 表示重命名后的库文件名称。






将通过`add_library`命令生成不同名字的动态库和静态库文件，使用`set_target_properties`命令，就可以使二者文件名相同。

```

cmake_minimum_required (VERSION 2.8)
project (demo)
set (SRC_LIST ${PROJECT_SOURCE_DIR}/subFunc/testFunc1.c
      ${PROJECT_SOURCE_DIR}/subFunc/testFunc2.c)
add_library (subFunc_shared SHARED ${SRC_LIST})
add_library (subFunc_static STATIC ${SRC_LIST})
set_target_properties (subFunc_shared PROPERTIES OUTPUT_NAME "subFunc")
set_target_properties (subFunc_static PROPERTIES OUTPUT_NAME "subFunc")
set (LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)

```

4. 链接动态库和静态库

| | | | |
|--|-----------------|------|------|
|  bin | 2022/8/26 9:10 | 文件夹 | |
|  bulid | 2022/8/25 10:52 | 文件夹 | |
|  src | 2022/8/26 9:12 | 文件夹 | |
|  thirdparty | 2022/8/25 16:25 | 文件夹 | |
|  CMakeLists.txt | 2022/8/24 11:47 | 文本文档 | 1 KB |

```

|-- bin
|-- build
|--src
    |-- main.c
|-- thridparty
    |-- include
        |-- subFunc2.h
        |-- subFunc1.h
    |-- lib
        |-- subFunc.a
        |-- subFunc.so
|-- CMakeLists.txt

```

一个库的使用一般包括两个方面，一个是库本身，另外一个库的头文件，所以cmake有两个函数对两个部分进行包含。

(1).包含库文件

包含库文件需要 `find_library` 和 `target_link_libraries` 命令，这两个命令的含义如下：

| 命令 | 含义 |
|------------------------------------|------------------------------|
| <code>find_library</code> | 表示在指定目录下查找指定库，并把库的绝对路径存放到变量里 |
| <code>target_link_libraries</code> | 表示把目标文件与库文件进行链接 |

- **find_library**命令使用:

```
find_library (<var> <lib> HINTS [libdir ...])
```

参数var: 表示库文件的变量名。

参数lib: 表示库文件, 如libname.a或libname.so。

参数libdir: 表示库文件所在的目录路径。

此外类似的命令还有还有find_package命令, find_package()命令适合引用已经配置好的Find.cmake、Config.cmake等文件的第三方库, 这些cmake文件保存了库的文件信息。Find.cmake脚本文件比较负责, 可通过一下python脚本自动生成:

```
#文件名auto_generate_findxxx_cmake.py, 放于第三方库的根目录
#自动生成脚本, 使用之前一般只需要更改变量lib_name为库名称即可
# 自动生成FindXXX.cmake脚本
import sys, getopt
import os

def write_help_info(file, lib_name):
    file.writelines("#-----demo-----\n")
    file.writelines("# \n")
    file.writelines("# set(CMAKE_MODULE_PATH =
${CMAKE_CURRENT_SOURCE_DIR}/cmake) \n")
    file.writelines("# project(import_" + lib_name + "_demo) \n")
    file.writelines("# find_package(" + lib_name + " REQUIRED) \n")
    file.writelines("# include_directories("${ + lib_name +
"_INCLUDE_DIRS}) \n")
    file.writelines("# add_executable(${PROJECT_NAME} main.cpp) \n")
    file.writelines("# target_link_libraries(${PROJECT_NAME} ${" +
lib_name + "_LIBRARIES}) \n")
    file.writelines("# \n")
    file.writelines("# ----- \n")

def search_lib_files(file_dir):
    libs = []
    for root, dirs, files in os.walk(file_dir):
        for file in files:
            if os.path.splitext(file)[1] == '.lib':
                libs.append(os.path.splitext(file)[0])
    return libs

def generate_cmake_file(argv):
    lib_name = "OpenCV"
    mini_cmake_version = "3.12"
    include_dir_name = "include"
    debug_lib_dir = "lib/debug"
    release_lib_dir = "lib/release"
    file = open("Find" + lib_name + ".cmake", "a")
```

```

write_help_info(file, lib_name)
file.writelines("# \n")
file.writelines("cmake_minimum_required(VERSION " +
mini_cmake_version + ") \n")
file.writelines("# \n")
file.writelines("if(NOT " + lib_name + "_FOUND) \n")
file.writelines("    set(" + lib_name + "_INCLUDE_DIRS
\"${CMAKE_CURRENT_LIST_DIR}/../" + include_dir_name + "\\") \n")
file.writelines("    set(" + lib_name + "_LIBRARIE_DIR_DEBUG
\"${CMAKE_CURRENT_LIST_DIR}/../" + debug_lib_dir + "\\") #debug \n")
file.writelines("    set(" + lib_name + "_LIBRARIE_DIR_RELEASE
\"${CMAKE_CURRENT_LIST_DIR}/../" + release_lib_dir + "\\") #release \n")
file.writelines("    set(" + lib_name + "_LIBRARIES_DIR ${" + lib_name
+ "_LIBRARIE_DIR_DEBUG} ${" + lib_name + "_LIBRARIE_DIR_RELEASE}) \n")

current_dir = os.getcwd()
debug_libs = search_lib_files(current_dir + "/" + debug_lib_dir)
release_libs = search_lib_files(current_dir + "/" + release_lib_dir)
file.writelines("# \n")
file.writelines("    #find and set the debug libraries. \n")
for d_lib in debug_libs:
    file.writelines("    find_library(" + d_lib + "_LIBRARY_DEBUG
NAMES " + d_lib + " PATHS ${" + lib_name + "_LIBRARIE_DIR_DEBUG}
PATH_SUFFIXES lib) \n")
file.writelines("# \n")
file.writelines("    #find and set the release libraries. \n")
for r_lib in release_libs:
    file.writelines("    find_library(" + r_lib + "_LIBRARY_DEBUG
NAMES " + r_lib + " PATHS ${" + lib_name + "_LIBRARIE_DIR_RELEASE}
PATH_SUFFIXES lib) \n")

file.writelines("# \n")
file.writelines("    # merge debug and release libs to " + lib_name +
"_LIBRARIES \n")
file.writelines("    include(SelectLibraryConfigurations) \n")
str_libs = ""
for lib in debug_libs:
    file.writelines("    select_library_configurations( " + lib + " )
\n")
    str_libs = str_libs + " ${" + lib + "_LIBRARY} "
file.writelines("    set(" + lib_name + "_LIBRARIES " + str_libs + ")
\n")
file.writelines("# \n")
file.writelines("    include(FindPackageHandleStandardArgs) \n")
file.writelines("    find_package_handle_standard_args(" + lib_name
+ " DEFAULT_MSG " + lib_name + "_INCLUDE_DIRS " + lib_name + "_LIBRARIES_DIR
+ lib_name + "_LIBRARIES) \n")
file.writelines("    mark_as_advanced(" + lib_name + "_INCLUDE_DIRS " +
lib_name + "_LIBRARIES_DIR " + lib_name + "_LIBRARIES) \n")

```

```

        file.writelines("endif() \n")
        file.close()

if __name__ == '__main__':
    generate_cmake_file(sys.argv[1:])

```

在 `find_path` 和 `find_library` 以及 `find_package` 时，会搜索一些默认的路径。当我们 将一些 `lib` 安装在非默认搜索路径时，`CMake` 就没法搜索到了，可设置路径到环境变量中：

```

SET (CMAKE_INCLUDE_PATH "include_path") #find_path, 查找头文件
SET (CMAKE_LIBRARY_PATH "lib_path") #find_library, 查找库文件
SET (CMAKE_MODULE_PATH "module_path") #find_package, 查找库

```

- **target_link_libraries**命令使用：

```
target_link_libraries (<objname> [libpath ...] )
```

参数objname：表示生成程序的名称。

参数libpath：表示库文件的路径。

```

cmake_minimum_required (VERSION 2.8)
project (demo)
set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
set (SRC_LIST ${PROJECT_SOURCE_DIR}/src/main.c)
include_directories (${PROJECT_SOURCE_DIR}/thridparty/include)
find_library(SUBFUNC_LIB subFunc HINTS
${PROJECT_SOURCE_DIR}/thridparty/lib)
add_executable (main ${SRC_LIST})
target_link_libraries (main ${SUBFUNC_LIB})

```

也可以直接使用**target_link_libraries**命令链接库文件，但是**target_link_libraries**命令 要放在**add_executable**命令之后。常见的第三方库有如OpenCV、OpenMP等，具体用法如 下：

```

cmake_minimum_required(VERSION 2.8)
project(idemo)
find_package(OpenCV REQUIRED)
find_package(OpenMP)
include_directories(${OpenCV_INCLUDE_DIRS})
add_executable(${PROJECT_NAME} main.cpp)
target_link_libraries(${PROJECT_NAME} ${OpenCV_LIBS})
if(OpenMP_CXX_FOUND)
    target_link_libraries(${PROJECT_NAME} PUBLIC OpenMP::OpenMP_CXX)
endif()

```

(2).包含库的头文件

包含库的头文件有两种类似的命令，分别是`include_directories`命令和`target_include_directories`命令，二者的主要区别如下：

| 命令 | 区别 |
|---|------------------------|
| <code>include_directories</code> | 头文件全局可见，所有目标都可访问此头文件路径 |
| <code>target_include_directories</code> | 头文件局部可见，指定目标才能访问的头文件路径 |

如果有不同目录相同名称的头文件，使用`include_directories`命令可能会出现冲突。

- `target_include_directories`命令使用：

```
target_include_directories (<objname> [head_dir...])
```

参数objname：表示生成程序的名称。

参数head_dir：表示库头文件目录的路径。

```
cmake_minimum_required (VERSION 2.8)
project (demo)
set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
set (SRC_LIST ${PROJECT_SOURCE_DIR}/src/main.c)
find_library(TESTFUNC_LIB subFunc HINTS
${PROJECT_SOURCE_DIR}/thirdparty/lib)
add_executable (main ${SRC_LIST})
target_link_libraries (main ${TESTFUNC_LIB})
target_include_directories (main
${PROJECT_SOURCE_DIR}/thirdparty/include)
```

5. 添加编译和控制选项

(1).添加编译选项

有时编译程序时想添加一些编译选项，如`-Wall`，`-std=c++11`等，就可以使用`add_compile_options`命令来进行操作。

```
add_compile_options(-std=c++11 -Wall)
```

(2).添加控制选项

在编译代码时如果想只编译一些指定的源码，可能会遇到如下两种情况：

1. 本来要生成多个bin或库文件，现在只想生成部分指定的bin或库文件
2. 对于同一个bin文件，只想编译其中部分代码（使用宏来控制）

可以使用`option`命令和条件控制语句，来实现对编译内容的控制。

```
option(<var> "<help_text>" <value>)
```

参数var: 表示option选项变量名
参数help_text: 表示描述、解释、备注信息
参数value: 表示选项初始化值（除ON而外全为OFF）

1. 第一种情况：指定文件进行编译

```
|— bin
|— build
|— include
|—src
    |— main2.c
    |— main1.c
    |— CMakeLists.txt
|— CMakeLists.txt
```

最外层CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)
project(demo)
option(MYOPTION "enable debug compilation" OFF)
set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
add_subdirectory(src)
```

内层CMakeLists.txt

```
cmake_minimum_required (VERSION 2.8)
if (MYOPTION)
    add_executable(main2 main2.c)
    message (STATUS "开关ON, 输出main2")
else()
    add_executable(main1 main1.c)
    message (STATUS "开关OFF, 输出main1")
endif()
```

这里if 语句会判断option开关变量MYOPTION的状态。如果是ON，则对应true。如果是OFF，则对应false。

message命令是给用户提示消息，STATUS是预定义的变量，表示非重要的消息，详细的使用如下：


```
message( [STATUS|WARNING|AUTHOR_WARNING|FATAL_ERROR|SEND_ERROR] "message
to display" ...)
```

(无) = 重要消息。
STATUS = 非重要消息。
WARNING = CMake 警告, 会继续执行。
AUTHOR_WARNING = CMake 警告 (dev), 会继续执行。
SEND_ERROR = CMake 错误, 继续执行, 但是会跳过生成的步骤。
FATAL_ERROR = CMake 错误, 终止所有处理过程。

CMake 的命令行工具会在 `stdout` 上显示 `STATUS` 消息, 在 `stderr` 上显示其他所有消息。CMake 的 GUI 会在它的 `log` 区域显示所有消息。交互式的对话框 (`ccmake` 和 `CMakeSetup`) 将会在状态行上一次显示一条 `STATUS` 消息, 而其他格式的消息会出现在交互式的弹出式对话。

终端命令:

```
cd build
cmake ..
make
```

输出结果:

开关OFF, 输出main1

在执行 `cmake` 时, 修改 `CMakeLists.txt` 中的 `option` 选项比较繁琐, 可以使用 `-D+选项名称`, 修改选项的值, 如下所示:

```
cd build
cmake .. -DMYOPTION=ON
make
```

输出结果:

开关ON, 输出main2

这样就可以通过选择 `option` 开关输出对应的程序, 此外更改的 `option` 开关选项会被缓存, 下次运行如果没做更改, 则默认上一次的值。

2. 第二种情况: 指定代码块进行编译

指定代码中那些代码进行编译, 需要在代码中插入想要的宏定义, CMake 在编译过程中读取这些宏进行选择性的编译处理。添加预处理器宏定义要用到 `add_definitions` 命令。

```
add_definitions(-D<Macro>)
```

参数 `Macro`: 源代码中宏的名字

`main.c` 文件内容:

```
#include <stdio.h>

int main(void)
{
#ifdef CASE1
    printf("代码块1被执行\n");
#endif
#ifdef CASE2
    printf("代码块2被执行\n");
#endif
return 0;
}
```

CMakeLists.txt文件内容:

```
cmake_minimum_required(VERSION 2.8)
project(demo)
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
option(CASE1 "print one message" OFF)
option(CASE2 "print another message" OFF)
if (CASE1)
    add_definitions(-DCASE1)
endif()
if (CASE2)
    add_definitions(-DCASE2)
endif()
add_executable(main1 main1.c)
```

如果要代码块1被执行, 代码块2不被执行, 修改相应option的开关即可。

终端命令:

```
cd build
cmake .. -DCASE1=ON -DCASE2=OFF
make
```

输出结果:

```
代码块1被执行
```

控制流除了条件控制流 `if` 以外, 还有循环控制流 `foreach` 和 `while`、过程定义宏和函数。

6. 平台移植

(1).系统平台匹配

系统信息文本获取：

```
cmake --system-information information.txt
```

系统信息控制台获取：

```
if (CMAKE_SYSTEM_NAME matches "Linux")
    message(STATUS "current platform: Linux ")
elseif (CMAKE_SYSTEM_NAME matches "Windows")
    message(STATUS "current platform: Windows")
elseif (CMAKE_SYSTEM_NAME matches "FreeBSD")
    message(STATUS "current platform: FreeBSD")
else ()
    message(STATUS "other platform: ${CMAKE_SYSTEM_NAME}")
endif()
```

或者：

```
if(WIN32)
    message(STATUS "current platform: windows")
elseif(APPLE)
    message(STATUS "current platform: Apple systems")
elseif(UNIX)
    message(STATUS "current platform: UNIX-like OS's")
endif()
```

(2).编译环境设置

1. CMake编译器预定义变量：表示C或CXX

- CMAKE__COMPILER_LOADED：如果为项目启用了语言，则将设置为 TRUE 。
- CMAKE__COMPILER_ID：编译器标识字符串，编译器供应商所特有。例如，GCC 用于 GNU编译器集合，AppleClang 用于macOS上的Clang，MSVC 用于Microsoft Visual Studio编译器。注意，不能保证为所有编译器或语言定义此变量。
- CMAKE_COMPILER_IS_GNU：如果语言是GNU编译器集合的一部分，则将此逻辑变量设置为 TRUE 。注意变量名的部分遵循GNU约定：C语言为 CC , C++语言为 CXX , Fortran语言为 G77 。
- CMAKE__COMPILER_VERSION：此变量包含一个字符串，该字符串给定语言的编译器版本。
- CMAKE__STANDARD：该变量表示语言标准。

2. 编译器设置：

方法一在终端指定编译器：

```
cmake .. -DCMAKE_CXX_COMPILER=/usr/local/gcc/bin/g++
```

方法二在CMakeLists.txt指定编译器

```
set (CMAKE_C_COMPILER "/usr/local/gcc/bin/gcc")
set (CMAKE_CXX_COMPILER "/usr/local/gcc/bin/g++")
```

注：这两条命令应该放在文件的开始位置（cmake_minimum_required命令之下，其他命令之上），否则可能无效。

设置编译器参数：

方法一：

```
add_compile_options(-std=c++11 -Wall -Werror)
```

方法二：

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -Wall -Werror")
```

两个方法的区别：

| add_compile_options | CMAKE_CXX_FLAGS |
|-------------------------|-----------------------|
| 对所有编译器有效 | 只针对C++文件 |
| 作用域全局，对所有CMakeLists.txt | 只针对当前模块CMakeLists.txt |

注意事项：

- 如果使用add_compile_options设置了C++参数，之后用C编译器可能会报错。
- CMAKE_CXX_FLAGS变量只在当前文件有效，如果项目中有多个模块，多个编译文件，那么需在每一个CMakeLists.txt文件中都添加对应的命令和参数。
- 对于一些在整个项目中通用的编译选项可以使用add_compile_options命令来添加比较方便，对于各个模块中的独立选项则使用CMAKE_CXX_FLAGS变量更好。

编译器选项预定义变量：

CMAKE_C_FLAGS: C语言编译器选项，对应于环境变量CFLAGS。

CMAKE_CXX_FLAGS: C++语言编译器选项，对应于环境变量CXXFLAGS。

CMAKE_CUDA_FLAGS: CUDA语言编译器选项，对应于环境变量CUDAFLAGS。

CMAKE_Fortran_FLAGS: Fortran语言编译器选项，对应于环境变量FFLAGS。

3. 编译器信息获取：

```
if ("${CMAKE_CXX_COMPILER_ID}" MATCHES "Clang")
    message(STATUS "current compiler: Clang")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "GNU")
    message(STATUS "current compiler: GNU")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "Intel")
    message(STATUS "current compiler: Intel")
elseif ("${CMAKE_CXX_COMPILER_ID}" STREQUAL "MSVC")
    message(STATUS "current compiler: MSVC")
endif()
```

更多信息获取;

```
if(CMAKE_CXX_COMPILER_LOADED)
    message(STATUS "C++编译器ID: ${CMAKE_CXX_COMPILER_ID}")
    if(CMAKE_COMPILER_IS_GNUCXX)
        message("编译器是GNU C++")
    else()
        message("编译器不是GNU c++")
    endif()
    message(STATUS "C++编译器的版本: ${CMAKE_CXX_COMPILER_VERSION}")
endif()

if(CMAKE_C_COMPILER_LOADED)
    message(STATUS "C编译器ID: ${CMAKE_C_COMPILER_ID}")
    if(CMAKE_COMPILER_IS_GNUCC)
        message("编译器是GNU C")
    else()
        message("编译器不是GNU c")
    endif()
    message(STATUS "C编译器的版本: ${CMAKE_C_COMPILER_VERSION}")
endif()
```

4. 生成器设置:

常用cmake构建命令

```
cmake ..
make
```

通常cmake会默认生成makefile文件,来支持make工具进行自动化编译。但cmake也支持了其他的自动化编译工具,如Ninja、Visual Studio等。如果使用其他生成器工具进行构建,CMake命令需要改变:

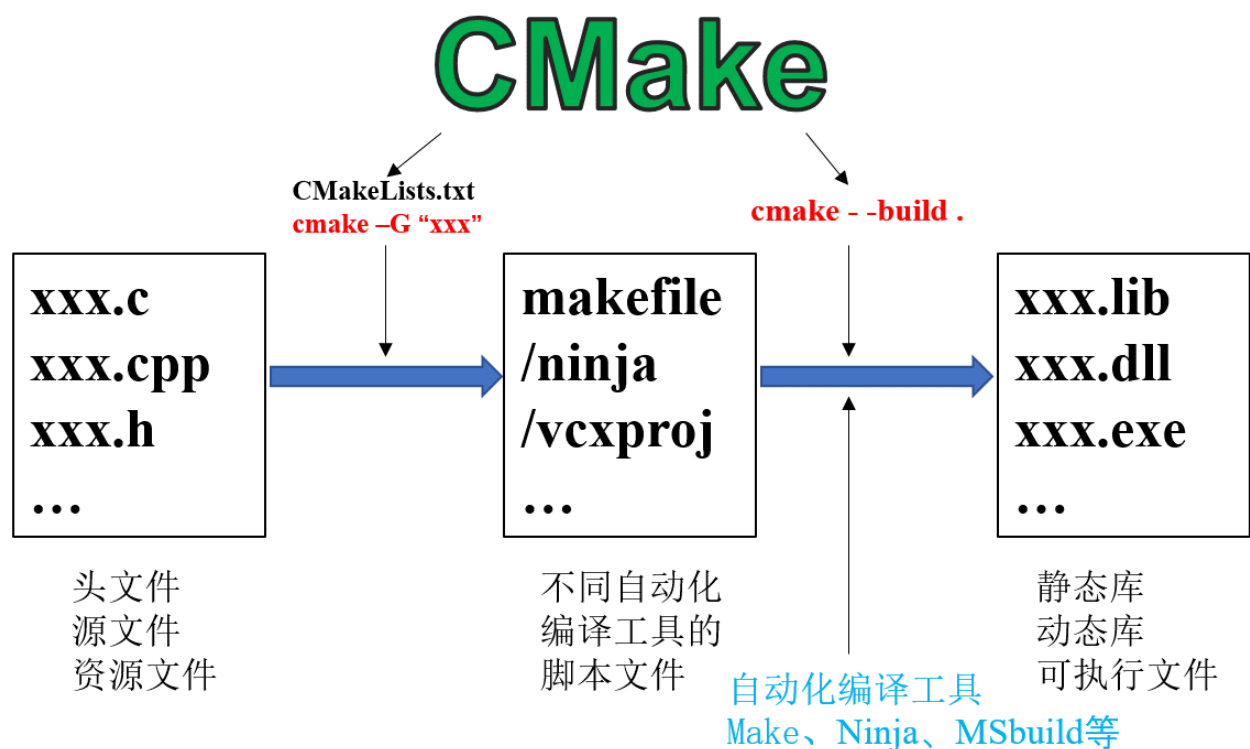
```
cmake .. -G Ninja
cmake .. -G "Visual Studio 2019" -A x64
cmake .. -G "Visual Studio 16" -A ARM
cmake .. -G "Visual Studio 16 2019" -A ARM64
```

```
cmake --build .
```

如果没有 `cmake --build .` 这条命令，就需要调用底层命令，比如 `make` 或者 `ninja` 这些工具进行编译。但现在 `cmake` 提供了一个统一的命令接口，不管用管底层，直接 `--build` 即可。但是有一个前提，系统已经配置好了自动化编译工具，否则会在 `cmake -G` 的时候，报找不到对应生成器。

可以使用 `help` 命令选项查找在平台上生成器名单，以及已安装的 CMake 版本：

```
cmake --help
```



CMake包含许多Command-Line，IDE和Extra生成器：

命令行构建工具生成器

- Borland Makefiles
- MSYS Makefiles
- MinGW Makefiles
- NMake Makefiles
- NMake Makefiles JOM
- Ninja
- Unix Makefiles
- WMake

IDE构建工具生成器

- Visual Studio 6
- Visual Studio 7

- Visual Studio 7 .NET 2003
- Visual Studio 8 2005年
- Visual Studio 9 2008年
- Visual Studio 10 2010年
- Visual Studio 11 2012年
- Visual Studio 12 2013年
- Visual Studio 14 2015年
- Visual Studio 15 2017年
- Visual Studio 16 2019
- Green Hills
- Xcode

Extra生成器

- CodeLite
- Eclipse CDT4
- KDevelop3(v3.10.3之后不受支持)

CMake的生成器设置可以为程序构建提供更好的跨平台特性。

(2).交叉编译

交叉编译是指在一个平台环境生成另一个平台环境的可执行代码，用于程序的移植。

在 PC 上进行开发时，需要使用编译器和链接器生成能够在我们机器上运行的可执行程序。但是当涉及到嵌入式开发时，情况就不同了。因为嵌入式设备的资源（CPU、RAM等）无法和 PC 相比，在设备上构建编译系统很麻烦或者根本不可能构建。因此通常做法是在 PC 上使用交叉编译工具链生成能够在嵌入式设备运行的可执行程序，然后再将程序放到设备中去执行。

使用CMake进行交叉编译分为三步：

1. 编写交叉编译过程中工具链编译配置文件 xxx.cmake 脚本文件
2. 终端命令 `cmake -DCMAKE_TOOLCHAIN_FILE= <cmakedir>/xxx.cmake <src-path>` 构建编译系统
3. 终端命令 `make`

第一步：编写工具链脚本文件

与CMakeLists.txt文件不同的是，工具链的配置命令写在xxx.cmake脚本文件中。一般工具链脚本文件命名为toolchain.cmake，也可以根据工程项目自己命名。

1. 设置目标系统和处理器架构：

```
set(CMAKE_SYSTEM_NAME <system>)
set(CMAKE_SYSTEM_PROCESSOR <processor>)
```

工具链脚本文件的许多设置主要以set命令为主。CMAKE_SYSTEM_NAME和CMAKE_SYSTEM_PROCESSOR是CMake预定义的变量，参数表示操作系统名称，常见的操作系统有Linux、QNX、WindowsCE和Android等。如果没有操作系统，那么就写Generic。参数表示处理器架构，如arm、x86和risc-v等。

2. 设置工具链编译器位置

```
set(CMAKE_C_COMPILER "<cpath>")
set(CMAKE_CXX_COMPILER "<c++path>")
参数<cpath>:表示工具链C语言编译器的位置
参数<c++path>:表示工具链C++语言编译器的位置
```

编译器一般都是工具链附带的一部分。

3. 指定目标平台链接库的位置

在开发时，需要使用系统库或第三方库的功能，在生成可执行文件时，将其进行链接。CMake提供了FIND_PROGRAM()，FIND_LIBRARY()，FIND_FILE()，FIND_PATH()和FIND_PACKAGE()实现相应的查找功能。如果我们在进行交叉编译时使用了上述指令，那么并不能生成可执行文件。因为默认情况下，上述指令查找的是主机上的相关文件，其并不适用于目标机器。

CMake也提供了链接目标机器库文件的变量：

- CMAKE_FIND_ROOT_PATH：设置其值为一系列的目录
(set(CMAKE_FIND_ROOT_PATH [dirpath...])，参数dirpath表示为目标机器库目录的路径，这样在执行FIND_XXX()指令时就会从这一系列的目录中进行查找。

跟随该变量的有下述3个变量，它们的值为NEVER、ONLY或BOTH：

- CMAKE_FIND_ROOT_PATH_MODE_PROGRAM：如果设置为NEVER，那么CMAKE_FIND_ROOT_PATH就不会对FIND_PROGRAM()产生影响，FIND_PROGRAM()不会在CMAKE_FIND_ROOT_PATH指定的目录中寻找；如果设置为ONLY，那么FIND_PROGRAM()只会从CMAKE_FIND_ROOT_PATH指定的目录中寻找；如果设置为BOTH，那么FIND_PROGRAM()会优先从CMAKE_FIND_ROOT_PATH指定的目录中寻找，再从默认的目录中寻找。

因为FIND_PROGRAM()大部分情况下用于寻找可执行程序，给后续的EXECUTE_PROCESS()或ADD_CUSTOM_COMMAND()指令使用。并且，只有主机在生成编译文件时使用该可执行程序。因此通常设置CMAKE_FIND_ROOT_PATH_MODE_PROGRAM为NEVER

```
(set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER) ;
```

- CMAKE_FIND_ROOT_PATH_MODE_LIBRARY：由于在进行交叉编译，所以只能使用FIND_LIBRARY()查找符合目标机器的库文件，因此设置该变量值为ONLY(set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY))，表示只从CMAKE_FIND_ROOT_PATH指定的目录中查找；
- CMAKE_FIND_ROOT_PATH_MODE_INCLUDE：同上，将其值设置为ONLY。

注意：上述命令必须写入脚本中，使用 `-DCMAKE_TOOLCHAIN_FILE=xxx.cmake` 的方式使用。不能直接写入 `CMakeLists.txt` 或使用 `include(xxx.cmake)`。

第二步CMake终端命令：

```
cmake -DCMAKE_TOOLCHAIN_FILE= <cmakedir>/xxx.cmake <srcdir>
参数<cmakedir>：表示查马克脚本所在的目录
参数<srcdir>：表示源文件所在的目录
```

第三步make终端命令

```
make
```

工程举例：arm平台linux系统

```
|— camke
    |— toolchain.cmake
    |— findxxx.cmake
|— bin
    |— debug
    |— release
|— build
|— arm-linux-gnueabi
    |— ...
|— include
|— src
    |— main.c
    |— CMakeLists.txt
|— CMakeLists.txt
|— auto_generate_findxxx_cmake.py
|— ...
```

编写CMakeLists.txt

```
cmake_minimum_required (VERSION 2.8)
project (demo)
add_subdirectory (src)
```

```
arm-linux-gnueabi
cmake_minimum_required (VERSION 2.8)
project (demo)
set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin/release)
set (SRC_LIST ${PROJECT_SOURCE_DIR}/src)
include_directories (${PROJECT_SOURCE_DIR}/include)
find_package (arm-linux-gnueabi)
add_executable (main ${SRC_LIST})
target_link_libraries (main ${arm-linux-gnueabi_lib})
```

编写toolchain.cmake

```

# 设置目标系统、处理器架构
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR arm)

# 设置工具链目录变量
set(TOOL_CHAIN_DIR ${CMAKE_CURRENT_LIST_DIR}/../)
set(TOOL_CHAIN_INCLUDE ${TOOL_CHAIN_DIR}/arm-linux-gnueabi/include
${TOOL_CHAIN_DIR}/arm-linux-gnueabi/libc/usr/include)
set(TOOL_CHAIN_LIB ${TOOL_CHAIN_DIR}/arm-linux-gnueabi/lib
${TOOL_CHAIN_DIR}/arm-linux-gnueabi/libc/usr/lib)

# 设置编译器位置
set(CMAKE_C_COMPILER "${TOOL_CHAIN_DIR}/bin/arm-linux-gnueabi-
gcc.exe")
set(CMAKE_CXX_COMPILER "${TOOL_CHAIN_DIR}/bin/arm-linux-gnueabi-
g++.exe")

# 设置cmake查找链接库主路径
set(CMAKE_FIND_ROOT_PATH ${TOOL_CHAIN_DIR}/arm-linux-gnueabi)

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
# 只在指定目录下查找库文件
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
# 只在指定目录下查找头文件
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
# 只在指定目录下查找依赖包
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)

# 包含工具链文件
include_directories(
    ${TOOL_CHAIN_DIR}/arm-linux-gnueabi/include
    ${TOOL_CHAIN_DIR}/arm-linux-gnueabi/libc/usr/include)

# 设置CMAKE_INCLUDE_PATH
set(CMAKE_INCLUDE_PATH ${TOOL_CHAIN_INCLUDE})

# 设置CMAKE_LIBRARY_PATH
set(CMAKE_LIBRARY_PATH ${TOOL_CHAIN_LIB})

```

CMAKE_CURRENT_LIST_DIR表示当前xxx.cmake文件所在的目录。

四、参考资料

<https://www.jianshu.com/p/43c1711fa7d5>

https://blog.csdn.net/qq_38292379/article/details/108206349

https://blog.csdn.net/qq_38292379/article/details/108206349

<https://www.jianshu.com/p/c2924130215a>

<http://www.wang-hj.cn/?p=2629>

https://blog.csdn.net/qq_43730206/article/details/114653193

<https://www.cnblogs.com/gooutlook/p/15773172.html>

<https://zhuanlan.zhihu.com/p/500020350>

<https://mangoroom.cn/tools/cmake-hands-on.html>

https://blog.csdn.net/sinat_31608641/article/details/122952787

https://blog.csdn.net/weixin_39766005/article/details/122435780

<https://www.jianshu.com/p/4bbc961c2de8>

<https://www.codenong.com/39501481/>

<https://blog.csdn.net/whatday/article/details/104376698>

https://blog.51cto.com/u_15080026/4256436

<https://www.zhihu.com/question/63537567>

https://blog.csdn.net/lhl_blog/article/details/123553686

<http://www.javashuo.com/article/p-dafbohro-nw.html>

https://zhuanlan.zhihu.com/p/97369704?utm_source=wechat_session

<https://zhuanlan.zhihu.com/p/500521290>

<https://www.jianshu.com/p/7b16815fe59d>

<https://www.h5w3.com/248006.html>

<https://zhuanlan.zhihu.com/p/77813702>

<https://zhuanlan.zhihu.com/p/487899911>

<https://zhuanlan.zhihu.com/p/487899911>

<http://t.zoukankan.com/linuxAndMcu-p-10675971.html>

<https://www.jianshu.com/p/a0915895dbbc>

<https://www.cnblogs.com/3d-gis/articles/14398733.html>

<https://blog.csdn.net/kangkanglhb88008/article/details/125894112>

https://blog.csdn.net/sinat_31608641/article/details/122149131

https://blog.csdn.net/wzj_110/article/details/116674655

<https://blog.csdn.net/alexhu2010q/article/details/103681788>

<https://www.iteye.com/blog/aigo-2294970>

https://blog.csdn.net/Jay_Xio/article/details/121099087

<https://www.codenong.com/25941536/>

<https://www.zhihu.com/question/63537567>

https://blog.csdn.net/qq_38446366/article/details/83541292

<https://wenku.baidu.com/view/70f32e133a68011ca300a6c30c2259010202f3b5.html>

<https://zhuanlan.zhihu.com/p/100367053>

<https://blog.csdn.net/u012156872/article/details/121605761>

https://blog.csdn.net/qq_33726635/article/details/121896441

<http://events.jianshu.io/p/37fbe3dd202b>

<https://blog.csdn.net/honk2012/article/details/89061526>

<https://www.csdn.net/tags/MtTaMgysNjE3MjMtYmxvZwOoOoOoOoOoO.html>

<https://my.oschina.net/biedamingming/blog/3198353>

<http://events.jianshu.io/p/37fbe3dd202b>

<https://blog.csdn.net/whatday/article/details/106191363/>

<https://www.freesion.com/article/749919883/>

<https://blog.csdn.net/zhangquan2015/article/details/91537827>

https://blog.csdn.net/qq_22748351/article/details/82312680

<http://www.wjhsh.net/flyinggod-p-13713646.html>