

## CMake零基础最佳入门实践[全]



杨明  
软件开发行业 CAE技术总监

关注他

43 人赞同了该文章

是否使用CMake是判断一个人是C++代码爱好者还是职业程序员的重要依据之一。

全文共约2.7万字，建议收藏阅读。

本文为我本人基于使用经验独家整理，如有谬误请指出，转载请私聊申请（WeChat ID：ym\_cae）。

本文将通过一个最简单的项目案例说明CMake的基本使用，然后引入大量的详实案例说明各个语法的用法和场景，并总结了CMake的预命名含义。最后布置了一个小作业：基于本文档解析知识大型线性代数库Eigen+的CMakeList.txt。我有信心只此一篇博客即让你成为CMake专家（拜托点个喜欢、收藏和关注）~

为了将源码转化为最终用户可以实际使用的东西，需要使用到编译器、链接器、测试框架、打包系统等，这些都增加了开发高质量、健壮性软件的复杂性，虽然一些 IDE+ 能够将这些过程简化一点，但是开发跨平台的软件并不是总能用到这些 IDE 的特性。CMake提供了这样一套标准的管理工具。



CMake logo

CMake 是一个开源的、跨平台的自动化构建系统，它主要用于跨平台的项目生成和管理。CMake 不直接构建软件，而是它从源代码+和构建参数文件（通常是 CMakeLists.txt 文件）中生成成本地构建环境所需的配置文件，也即为各种平台生成标准的构建文件。

CMake可以产生多种构建文件，如Unix/Linux下的Makefile或Win-dows Visual C+++的 projects/workspaces。它使得开发者可使用各种平台上的原生构建系统，这是CMake区别于Automake等其他类似系统的重要特点，也是CMake的重要优势。相对于Automake，它提供了更多对Windows的支持。CMake虽然使用简单，但同样具备大项目管理能力。

下载CMake：

Download CMake  
[cmake.org/download/](https://cmake.org/download/)

### 一个最简单的CMake工程

假如我有一个example.cpp代码如下，用于计算数字的平方根：

```
// example.cpp
#include <cmath>
#include <cstdlib>
```

```
int main(int argc, char* argv[])
{
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " number" << std::endl;
        return 1;
    }
    // convert input to double
    const double inputValue = atof(argv[1]);

    // calculate square root
    const double outputValue = sqrt(inputValue);
    std::cout << "The square root of " << inputValue
              << " is " << outputValue
              << std::endl;
    return 0;
}
```

在example.cpp所在的同级目录下，构建一个CMakeLists.txt，内容如下：

```
# 指定 CMake 的最低版本
cmake_minimum_required(VERSION 3.15)

# 定义项目名称
project(Example)

# 添加可执行文件
add_executable(Example example.cpp)

# 显示编译选项
message(STATUS "CMAKE_CXX_COMPILER is ${CMAKE_CXX_COMPILER}")
```

cmake\_minimum\_required 指定使用 CMake 的最低版本号，project 指定项目名称，add\_executable 用来生成可执行文件，需要指定生成可执行文件的名称和相关源文件。注意，此示例在 CMakeLists.txt 文件中使用小写命令。CMake 支持大写、小写和混合大小写命令。

## 构建、编译和运行

可以采用MinGW来编译，也可以再Visual Studio Code<sup>+</sup>和Visual Studio上编译。

### • MinGW编译CMake工程

下面介绍采用MinGW-64编译的教程，首先下载MinGW：

Downloads - MinGW-w64  
[www.mingw-w64.org/downloads/](http://www.mingw-w64.org/downloads/)

先运行 cmake 命令来构建项目，然后使用你选择的编译工具进行编译。

先从命令行进入到 step1 目录，并创建一个构建目录 build，接下来，进入 build 目录并运行 CMake 来配置项目，并生成构建系统：

```
mkdir build
cd build
cmake -G"MinGW Makefiles" ..
```

构建系统是需要指定 CMakeLists.txt 所在路径，此时在 build 目录下，所以用 .. 表示 CMakeLists.txt 在上一级目录。

Windows 下，CMake 默认使用微软的 MSVC<sup>+</sup> 作为编译器，我想使用 MinGW 编译器，可以通

```
cmake --build .
```

--build 指定编译生成的文件存放目录，其中就包括可执行文件，. 表示存放到目前目录，在 build 目录下生成了一个 Example.exe 可执行文件，试着执行它：

```
Example.exe 5
```

```
The square root of 5 is 2.23607
```

该程序计算 5 的平方根，从输出结果看已经得到了正确的结果。  
此时目录结构为：

```
step1/  
  build/  
  CMakeLists.txt  
  example.cpp
```

#### • Visual Studio Code编译CMake工程

可以参照下面教程安装CMake和编译：

windows10使用vscode+cmake编译  
c++代码 - mohist - 博客园  
[www.cnblogs.com/pandamohist/p/14531...](http://www.cnblogs.com/pandamohist/p/14531...)



Visual Studio可以自动配置程序。

## CMake 语法命令进阶

通过上面这个案例，你已经知道如何编写一个CMakeList.txt文件，并且通过它来编译和配置你的程序。下面将为你深入介绍一些进阶的语法指令。我将从下面几个方面分类展开介绍这些语法指令，这些介绍比较全面，可以作为日常参考的手册，以便大家随时查找。

语法结构中"[ ]"所包含的内容是可选项，不是必填参数。

- 基础配置：设置CMake项目基本信息
- 构建目标：构建可执行文件
- 文件操作：管理文件和目录
- 控制流：条件判断和逻辑循环
- 测试与安装：集成测试与安装命令
- 消息和调试：输出调试信息
- 宏和函数：定义可复用逻辑块
- 自定义命令：CMake自定义操作
- 模块管理：引入外部模块和文件

### 基础配置

这些功能用于设置 CMake 项目的基本信息和全局配置。

- `cmake_minimum_required(VERSION x.x.x)`：指定所需的最低 CMake 版本。使用举例：

```
cmake_minimum_required(VERSION <版本号>)
```

持的语言（如 C，CXX，Fortran），并指定项目版本号。使用举例：

```
project(MyApp LANGUAGES CXX VERSION 1.0)
```

- **set(<变量名> <值> [CACHE <类型> <文档说明>] [FORCE])**: 设置变量的值。CACHE将变量设置为缓存变量。FORCE：强制覆盖已有值。使用举例：

```
# 设置可能的mkl路径，以便后面使用find_package查找mkl位置
set(MKL_POSSIBLE_PATHS
    "/opt/intel/mkl"
    "C:\\Program Files (x86)\\Intel\\oneAPI\\mkl\\latest"
    "/usr/local/mkl"
    "D:/Intel/oneAPI/mkl/latest"
)
```

- **unset(<变量名> [CACHE])**: 移除变量。CACHE：清除缓存中的变量。使用举例：

```
# 先设置变量，然后根据条件取消设置
set(SOME_VAR "some value")
if(NOT SOME_CONDITION)
    unset(SOME_VAR) # 取消变量
    unset(SOME_VAR CACHE) #取消缓存
endif()

# 检查变量是否被取消设置
if(NOT DEFINED SOME_VAR)
    message(STATUS "SOME_VAR is not set")
endif()
```

- **cmake\_policy(SET <策略> <行为>)**: 设置 CMake 的行为策略。随着CMake版本的更新，一些旧的行为可能会被新的更安全或更一致的行为所取代，而 cmake\_policy 命令允许开发者明确指定他们希望遵循的策略。使用举例：

```
# 根据CMake版本条件设置策略
if(CMAKE_VERSION VERSION_GREATER 3.10)
    cmake_policy(SET CMP0001 NEW)
endif()
```

- **set\_property**，用于设置指定目标（目标可以是变量、文件、目录、库、可执行文件等）的属性，语法结构如下：

```
set_property(< GLOBAL          # GLOBAL域是唯一的，并且不接受特殊的任何名字
    DIRECTORY [dir]          #DIRECTORY域默认为当前目录，但也可以用全路径或相对路径
    TARGET [target1 [target2 ...]] # TARGET域可命名零或多个已经存在的目标
    SOURCE [src1 [src2 ...]]    # SOURCE域可命名零或多个源文件
    TEST [test1 [test2 ...]]   # TEST域可命名零或多个已存在的测试
    CACHE [entry1 [entry2 ...]]> # CACHE域必须命名零或多个已存在条目的cache
    [APPEND][APPEND_STRING]
    PROPERTY <name>[value1 [value2 ...]]
    # 必选项PROPERTY后面紧跟着要设置的属性的名字。其他的参数用于构建以分号隔开的
    # 如果指定了APPEND选项，则指定的列表将会追加到任何已存在的属性值当中。
    # 如果指定了APPEND_STRING选项，则会将值作为字符串追加到任何已存在的属性值。
```

set\_property使用举例[1]：设置执行目标的属性

```
# 设置可执行文件的属性
set_property(TARGET my_executable PROPERTY OUTPUT_NAME "my_app")

# 设置可执行文件的编译选项
set_property(TARGET my_executable PROPERTY COMPILE_OPTIONS "-Wall;-Wextra")
```

#### set\_property使用举例[2]：设置源文件属性

```
# 设置源文件的属性，采用-O3优化编译
set_property(SOURCE main.cpp PROPERTY COMPILE_FLAGS "-O3")
```

#### set\_property使用举例[3]：变量属性

```
# 设置变量的属性
set_property(VARIABLE MY_VARIABLE PROPERTY TYPE STRING)
```

#### set\_property使用举例[4]：设置缓存变量属性

```
# 创建一个缓存变量
set(MY_CACHE_VARIABLE "default_value" CACHE STRING "Description of my cache var")

# 设置缓存变量的属性
set_property(CACHE MY_CACHE_VARIABLE PROPERTY STRINGS "option1;option2;option3")
```

#### set\_property使用举例[5]：设置接口的属性

```
# 创建一个库
add_library(my_library STATIC my_library.cpp)

# 设置库的接口属性
set_property(TARGET my_library PROPERTY INTERFACE_COMPILE_DEFINITIONS "MY_LIBRA")

# 其他目标链接到这个库时，会自动包含这些接口属性
target_link_libraries(my_executable PRIVATE my_library)
```

- **get\_property**：获取指定目标（可以是变量、文件、目录、库、可执行文件等）的属性值。语法结构如下：

```
get_property(<variable>
             <GLOBAL |
             DIRECTORY [dir] |
             TARGET <target> |
             SOURCE <source> |
             TEST <test> |
             CACHE <entry> |
             VARIABLE>
             PROPERTY <name>
             [SET | DEFINED | BRIEF_DOCS | FULL_DOCS])
```

#### get\_property使用举例[1]：获取目标属性

```
# 假设已经设置了目标属性
set_property(TARGET my_executable PROPERTY OUTPUT_NAME "my_app")

# 获取目标的属性并打印出来
```

#### get\_property使用举例[2]: 获取源文件属性

```
# 设置源文件属性
set_property(SOURCE src/main.cpp PROPERTY COMPILE_FLAGS "-O3")

# 获取并打印源文件属性
get_property(SOURCE_COMPILE_FLAGS SOURCE src/main.cpp PROPERTY COMPILE_FLAGS)
message(STATUS "Compile flags for src/main.cpp: ${SOURCE_COMPILE_FLAGS}")
```

#### get\_property使用举例[3]: 获取目录属性

```
# 设置目录属性
set_property(DIRECTORY PROPERTY MY_DIRECTORY_PROPERTY "directory_value")

# 获取并打印目录属性
get_property(DIRECTORY_PROPERTY DIRECTORY PROPERTY MY_DIRECTORY_PROPERTY)
message(STATUS "Directory property: ${DIRECTORY_PROPERTY}")
```

#### get\_property使用举例[4]: 获取缓存变量属性

```
# 设置缓存变量属性
set_property(CACHE MY_CACHE_VARIABLE PROPERTY HELPSTRING "This is a cache vari

# 获取并打印缓存变量属性
get_property(CACHE_VARIABLE_HELPSTRING CACHE MY_CACHE_VARIABLE PROPERTY HELPSTR
message(STATUS "Cache variable help string: ${CACHE_VARIABLE_HELPSTRING}")
```

#### get\_property使用举例[5]: 检查属性是否存在

```
# 检查属性是否存在
get_property(HAVE_PROPERTY TARGET my_executable PROPERTY MY_CUSTOM_PROPERTY SET
if(HAVE_PROPERTY)
    message(STATUS "MY_CUSTOM_PROPERTY is set for my_executable")
else()
    message(STATUS "MY_CUSTOM_PROPERTY is not set for my_executable")
endif()
```

### 构建目标

这些语句用于定义可执行文件、库及其相关构建规则。

- **add\_executable**(<目标名> [<源文件>...]) : 定义一个可执行文件目标。 <目标名> : 可执行文件名称。 <源文件> : 源文件列表。使用举例:

```
add_executable(MyProject main.cpp utils.cpp)
```

- **add\_library**(<目标名> [STATIC | SHARED | MODULE] [<源文件>...]) : 定义库的目标并指定源文件的命令, 可以是**静态库**\*或共享库。 STATIC : 静态库 (默认)。 SHARED : 动态库。 MODULE : 模块库 (不被链接, 动态加载)。

#### add\_library使用举例[1]: 添加静态库 (其他类型的库类似)

```
# 创建一个名为 my_library 的静态库, 包含 src1.cpp 和 src2.cpp
add_library(my_library STATIC src1.cpp src2.cpp)
```

```
# 创建一个名为 my_interface 的接口库
# 接口库不包含实现，只包含接口定义
add_library(my_interface INTERFACE)
```

#### add\_library使用举例[3]: 添加库的编译选项

```
# 创建一个库，并为其添加编译选项
add_library(my_library STATIC src1.cpp src2.cpp)
target_compile_options(my_library PRIVATE -Wall -Wextra)
```

#### add\_library使用举例[4]: 添加库的链接库<sup>+</sup>

```
# 创建两个库，其中一个库链接到另一个库
add_library(my_library STATIC src1.cpp src2.cpp)
add_library(my_other_library STATIC other_src1.cpp other_src2.cpp)

# 将 my_other_library 链接到 my_library
target_link_libraries(my_library PRIVATE my_other_library)
```

#### add\_library使用举例[5]: 添加库的依赖项

```
# 创建一个库，并为其添加依赖项
add_library(my_library STATIC src1.cpp src2.cpp)

# 添加依赖项，例如，确保在编译 my_library 之前构建另一个目标
add_dependencies(my_library my_other_target)
```

- **target\_link\_libraries(<目标名> [PUBLIC | PRIVATE | INTERFACE] <库名>...):** 指定要链接到目标（通常是可执行文件或库）的库。PUBLIC: 库对目标及其依赖可见。PRIVATE: 库仅对目标可见。INTERFACE: 库仅对目标的依赖可见。

#### target\_link\_libraries使用举例[1]: 链接库的标准用法（单个和多个）

```
# 假设两个命名分别为lib1和lib2的库
add_library(lib1 STATIC lib1_source.cpp)
add_library(lib2 STATIC lib2_source.cpp)

# 创建一个可执行文件，并链接到 lib1和lib2库，如果已知系统库，比如pthread+，可以直接添上
add_executable(my_executable main.cpp)
# lib1和lib2的出现顺序代表了链接顺序
target_link_libraries(my_executable PRIVATE lib1 lib2)
```

#### target\_link\_libraries使用举例[2]: 使用接口库链接编译定义和包含目录

```
# 创建一个接口库
add_library(my_interface INTERFACE)

# 为接口库添加编译定义和包含目录
target_compile_definitions(my_interface INTERFACE MY_INTERFACE_DEF)
target_include_directories(my_interface INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}/i

# 创建一个可执行文件，并链接到 my_interface 接口库
add_executable(my_executable main.cpp)
target_link_libraries(my_executable PRIVATE my_interface)
```

- **target\_include\_directories(<目标名> [PUBLIC | PRIVATE | INTERFACE] <路径>...):**  
PUBLIC: 路径对目标及其依赖可见。PRIVATE: 路径仅对目标可见。INTERFACE: 路径仅对目

target\_include\_directories使用举例：为目标添加包含目录的标准用法

```
# 创建两个可执行文件
add_executable(exe1 main1.cpp)
add_executable(exe2 main2.cpp)

# 为 exe1 和 exe2 添加包含目录
target_include_directories(exe1 PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/include)
target_include_directories(exe2 PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

参见上面target\_link\_libraries的使用方法，结合target\_include\_directories，可以完成接口库链接编译定义和包含目录功能。

- **target\_compile\_options(<目标名> [PUBLIC | PRIVATE | INTERFACE] <选项>...)**：目标（通常是可执行文件或库）添加编译器选项的命令。PUBLIC：选项对目标及其依赖可见。PRIVATE：选项仅对目标可见。INTERFACE：选项仅对目标的依赖可见。

target\_compile\_options使用举例[1]：为特定目标添加编译选项

```
# 创建一个可执行文件
add_executable(my_executable main.cpp)

# 为 my_executable 添加编译选项
target_compile_options(my_executable PRIVATE -Wall -Wextra)

# -Wall 和 -Wextra 是 GCC 和 Clang 编译器的选项，用于打开所有警告信息，
# PRIVATE 表示这些编译选项只应用于 my_executable，不会传播给链接到它的其他目标。
```

target\_compile\_options使用举例[2]：条件添加编译选项

```
# 创建一个可执行文件
add_executable(my_executable main.cpp)

# 根据编译器类型条件添加编译选项
if(CMAKE_CXX_COMPILER_ID STREQUAL "GNU")
    target_compile_options(my_executable PRIVATE -fPIC)
elseif(CMAKE_CXX_COMPILER_ID STREQUAL "Clang")
    target_compile_options(my_executable PRIVATE -fPIC)
endif()

# 这里，-fPIC 是一个编译选项，用于生成位置无关代码，这在创建动态库时是必需的。
# 根据编译器的不同（GCC 或 Clang），我们为 my_executable 添加了这个选项。
```

- **target\_compile\_definitions**：为目标（通常是可执行文件或库）添加编译时定义的宏的命令。

语法格式：

```
target_compile_definitions(<target>
    <INTERFACE|PUBLIC|PRIVATE> [items1...]
    [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

使用案例，创建一个条件编译：

```
# 创建一个可执行文件
add_executable(my_executable main.cpp)

# 根据平台条件添加编译定义
if(WIN32)
    target_compile_definitions(my_executable PRIVATE WINDOWS_PLATFORM)
elseif(UNIX AND NOT APPLE)
```



```
target_compile_definitions(my_executable PRIVATE MACOS_PLATFORM)
endif()
```

- `target_sources`：为已定义的目标（例如可执行文件或库）添加源文件。这个命令可以在多个地方调用，以向目标添加源文件。

语法格式：

```
target_sources(<target>
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

使用举例：为多个目标添加源文件

```
# 定义两个目标
add_executable(exe1 exe1_main.cpp)
add_library(lib1 STATIC lib1_source.cpp)

# 定义公共源文件
set(COMMON_SOURCES common_source.cpp)

# 为两个目标添加公共源文件
target_sources(exe1 PRIVATE ${COMMON_SOURCES})
target_sources(lib1 PRIVATE ${COMMON_SOURCES})
```

- `target_link_options`：为目标（通常是可执行文件或库）指定链接器选项的命令

语法格式：

```
target_link_options(<target> [BEFORE]
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
```

使用举例：为可执行文件和库建立链接选项

```
# 创建一个可执行文件
add_executable(my_executable main.cpp)

# 为 my_executable 添加链接选项
target_link_options(my_executable PRIVATE -Wl,--as-needed)

# 创建一个静态库
add_library(my_library STATIC lib_source.cpp)

# 为 my_library 添加链接选项
target_link_options(my_library PRIVATE -Wl,--no-undefined)
```

## 文件操作

这些语法指令用于管理源文件、配置文件以及目录操作。

- `find_package(<包名> [版本号] [REQUIRED | QUIET] [MODULE | CONFIG])`：查找外部库或包。<包名>：包名称。[版本号]：指定版本号。REQUIRED：未找到时报错。QUIET：未找到时静默。MODULE：使用模块模式查找。CONFIG：使用配置模式查找。

`find_package`使用举例[1]：查找指定版本的包

```
# 使用 SomePackage 提供的功能
add_executable(my_executable main.cpp)
target_link_libraries(my_executable SomePackage::SomeLibrary)
```

find\_package使用举例[2]: 查找配置文件

```
# 查找 "SomePackage" 的配置文件
find_package(SomePackage CONFIG REQUIRED)

# 使用 SomePackage 提供的目标
add_executable(my_executable main.cpp)
target_link_libraries(my_executable some_package_target)
```

find\_package使用举例[3]: 查找模块文件

```
# 查找 "SomePackage" 的模块文件
find_package(SomePackage MODULE REQUIRED)

# 使用 SomePackage 提供的变量
include_directories(${SomePackage_INCLUDE_DIRS})
add_executable(my_executable main.cpp)
target_link_libraries(my_executable ${SomePackage_LIBRARIES})
```

- **find\_library(<变量名> <库名> [PATHS <路径列表>])**: 在系统上搜索指定的库。这个命令会查找静态库 (.a)、动态库 (.so、.dll) 或其他库文件, 并返回库的路径。<变量名>: 保存结果的变量。<库名>: 库名称。PATHS: 指定搜索路径。

find\_library使用举例[1]: 在制定目录下查找特定名称的库

```
# 指定查找路径
set(MYLIB_SEARCH_PATHS /usr/local/lib /opt/mylib/lib)

# 查找名为 "mylib" 的库, 并指定查找路径
find_library(MYLIB_LIBRARY NAMES mylib PATHS ${MYLIB_SEARCH_PATHS})

# 使用找到的库
target_link_libraries(my_executable ${MYLIB_LIBRARY})
```

find\_library使用举例[2]: 查找库并设置缓存

```
# 查找名为 "mylib" 的库, 并将其路径缓存起来
find_library(MYLIB_LIBRARY NAMES mylib CACHE)

# 使用找到的库
target_link_libraries(my_executable ${MYLIB_LIBRARY})
# 在这个例子中, CACHE 选项表示如果 MYLIB_LIBRARY 已经被缓存,
# 那么 find_library 命令将使用缓存的值, 而不是再次搜索。
```

find\_library使用举例[3]: 查找库并设置环境变量

```
# 设置环境变量以帮助查找库
set(CMAKE_LIBRARY_PATH ${CMAKE_LIBRARY_PATH} $ENV{LIBRARY_PATH})

# 查找名为 "mylib" 的库
find_library(MYLIB_LIBRARY NAMES mylib)

# 使用找到的库
target_link_libraries(my_executable ${MYLIB_LIBRARY})
```

<后缀>...] [NO\_DEFAULT\_PATH] [REQUIRED] [DOC "文档字符串"]): 在指定路径中查找某个文件或目录, 并将找到的路径存储到变量中。 <变量名>: 用于存储找到的路径的变量名称。  
 <文件名>: 需要查找的文件名(可以包含相对路径)。 HINTS: 指定额外的提示路径。  
 PATHS: 指定搜索路径。 ATH\_SUFFIXES: 指定路径后缀。 NO\_DEFAULT\_PATH: 不使用默认的搜索路径。 REQUIRED: 如果未找到文件, 停止配置并报错。 DOC: 变量的说明文档。

find\_path使用举例[1]: 在指定路径下查找包含特定头文件的目录

```
# 指定查找路径
set(MY_HEADER_SEARCH_PATHS /usr/local/include /opt/mylib/include)

# 查找包含 "my_header.h" 的目录, 并指定查找路径
find_path(MY_HEADER_DIR NAMES my_header.h PATHS ${MY_HEADER_SEARCH_PATHS})

# 使用找到的目录
if(MY_HEADER_DIR)
    include_directories(${MY_HEADER_DIR})
    add_executable(my_executable main.cpp)
endif()
```

find\_path使用举例[2]: 查找库的包含目录

```
# 查找 "mylib" 库的包含目录
find_path(MYLIB_INCLUDE_DIR NAMES mylib.h PATHS /usr/lib/mylib/include /usr/shc

# 使用找到的目录
if(MYLIB_INCLUDE_DIR)
    include_directories(${MYLIB_INCLUDE_DIR})
    add_executable(my_executable main.cpp)
    target_link_libraries(my_executable mylib)
endif()
```

- **include\_directories([AFTER|BEFORE] [SYSTEM] <路径>...)**: 指定头文件搜索路径。  
 AFTER: 将路径添加到当前头文件路径列表的末尾(默认行为)。 BEFORE: 将路径添加到当前头文件路径列表的开头。 SYSTEM: 将路径标记为系统路径, 抑制编译器的警告。 <路径>: 头文件所在的路径, 可以是绝对路径或相对路径。

include\_directories使用举例: 添加多个目录

```
# 添加多个包含目录
# 定义包含目录的变量
set(MY_INCLUDE_DIRECTORIES
    ${CMAKE_CURRENT_SOURCE_DIR}/include
    ${CMAKE_CURRENT_SOURCE_DIR}/another_include
)
include_directories(${MY_INCLUDE_DIRECTORIES})

# 创建一个可执行文件
add_executable(my_executable main.cpp)
```

- **aux\_source\_directory(<dir> <variable>)**: 自动收集指定目录下的所有源文件。其中, <dir> 是你想要搜索的目录, 而 <variable> 是一个变量, 用于存储找到的所有源文件的列表, 默认仅处理后缀名为.c和.cpp的文件。**注意**: 这个命令不提供任何自定义规则来选择或排除特定的源文件, 应该慎用(更应该主要使用add\_executable, 逐个添加所需要的文件)。使用举例:

```
# 用法1: 手动列出源文件
set(MY_SOURCES main.cpp util.cpp renderer.cpp)
```

```
aux_source_directory(src MY_SOURCES)
```

# 用法2: 自动获取当前目录下的所有源文件, 并存入 `SRC_LIST`:

```
aux_source_directory(. SRC_LIST)
add_executable(MyProject ${SRC_LIST})
```

- `link_directories(dir1 dir2 ...)`: 添加编译器链接时搜索库文件的目录的命令。 <路径>: 库文件的搜索路径, 可以是绝对路径或相对路径。**注意**: 该命令添加的路径适用于当前目录及其子目录。推荐使用 `target_link_libraries` 进行目标级别的链接管理, 以避免全局路径污染。使用举例:

```
# 定义库目录的变量
set(MY_LIBRARY_DIRECTORIES
    ${CMAKE_CURRENT_SOURCE_DIR}/lib
    ${CMAKE_CURRENT_SOURCE_DIR}/another_lib
)

# 添加库目录
link_directories(${MY_LIBRARY_DIRECTORIES})

# 创建一个可执行文件
add_executable(my_executable main.cpp)

# 链接到库
target_link_libraries(my_executable mylibrary anotherlib)
```

- `configure_file(<输入文件> <输出文件> [COPYONLY])`: 该命令作用是将一个源文件复制到目标位置, 并在此过程中执行变量替换。这个命令常用于生成配置文件, 例如将 `.in` 文件 (带有占位符的模板文件) 转换成实际的头文件或配置文件。COPYONLY: 仅复制文件, 不替换变量。使用举例:

```
# 定义一些变量
set(MY_VAR "Hello, World!")
set(ANOTHER_VAR 1234)

# 使用 configure_file 命令将 src/config.h.in 复制到 include/config.h
# 并替换其中的 @MY_VAR@ 和 @ANOTHER_VAR@ 占位符
configure_file(src/config.h.in include/config.h)

# 添加一个可执行文件
add_executable(my_executable main.cpp)

# 目标会自动包含 include 目录, 因为 configure_file 生成的文件在那里
```

- `file(<操作> [参数]...)`: `file` 命令用于对文件和目录执行各种操作, 如复制文件、删除文件、创建目录等。COPY: 复制文件。WRITE: 写入文件。READ: 读取文件。GLOB: 获取符合模式的文件列表。REMOVE: 删除文件。使用举例:

```
# 1. 复制文件从 src.txt 到 build_dir/src.txt
file(COPY src.txt DESTINATION build_dir/)

# 2. 删除文件 old_file.txt
file(REMOVE old_file.txt)

# 3. 创建目录 new_dir
file(MAKE_DIRECTORY new_dir)

# 4. 检查文件 file.txt 是否存在
file(EXISTS file.txt FILE_EXISTS)

# 使用变量 FILE_EXISTS
```

```
else()
    message(STATUS "File does not exist.")
endif()
```

## 控制流

实现条件判断和循环逻辑。

- **if / elseif / else**: 条件语句。使用举例:

```
if(<条件>)
    # 操作
elseif(<条件>)
    # 操作
else()
    # 操作
endif()
```

- **foreach(<loop\_var> <items>) <commands> endforeach()**: `foreach` 命令在 CMake 中用于循环遍历列表中的每个元素，并对每个元素执行一系列操作。下面是一个使用案例:

```
# 定义一个包含多个源文件的列表
set(SOURCE_FILES main.cpp utils.cpp helpers.cpp)

# 使用 foreach 循环遍历列表中的每个源文件
foreach(SOURCE_FILE ${SOURCE_FILES})
    # 对每个源文件执行操作，例如添加编译定义
    get_filename_component(SOURCE_FILE_NAME ${SOURCE_FILE} NAME) # 提取文件名
    string(REPLACE ".cpp" ".h" SOURCE_FILE_NAME "${SOURCE_FILE_NAME}") # 假设对应的
    target_compile_definitions(my_executable PRIVATE "${SOURCE_FILE_NAME}_SOURCE="
endforeach()
```

在这个例子中，我们首先定义了一个包含三个源文件的列表 `SOURCE_FILES`。然后，我们使用 `foreach` 命令遍历这个列表，对每个源文件执行一系列操作。在 `foreach` 循环内部，我们使用 `get_filename_component` 命令提取每个源文件的文件名，并使用 `string(REPLACE ...)` 命令构造对应的头文件名。接着，我们使用 `target_compile_definitions` 命令为目标 `my_executable` 添加一个编译定义，该定义基于源文件名。

- **while(<condition>) <commands> endwhile()**: `while` 语句用于执行一个循环，直到指定的条件不再为真。

```
# 定义一个计数器变量
set(COUNT 0)

# 定义一个最大值
set(MAX_COUNT 5)

# 使用 while 循环直到计数器达到最大值
while(COUNT LESS ${MAX_COUNT})
    # 在每次循环时打印计数器的值
    message(STATUS "Count is ${COUNT}")

    # 增加计数器的值
    math(EXPR COUNT "${COUNT} + 1")
endwhile()

# 最后打印结束消息
message(STATUS "Reached maximum count.")
```

```
# 定义一个计数器变量
set(COUNT 0)

# 使用 while 循环
while(COUNT LESS 5)
    # 增加计数器的值
    math(EXPR COUNT "${COUNT} + 1")

    # 如果计数器的值是 3, 使用 continue 跳过当前迭代
    if(COUNT EQUAL 3)
        continue()
    endif()

    # 如果计数器的值是 5, 使用 break 退出循环
    if(COUNT EQUAL 5)
        break()
    endif()

    # 打印计数器的值
    message(STATUS "Count: ${COUNT}")
endwhile()
```

## 测试和安装

用于集成测试和安装配置。

- **enable\_testing()**: 启用测试。
- **add\_test(<测试名> <可执行文件> [<参数>...])**: 添加测试。<测试名>: 测试的名称。<可执行文件>: 要运行的测试程序或脚本。<参数>: 传递给测试程序的参数 (可选)。**注意**: 必须启用 `enable_testing()` 才能使用测试功能。测试可以通过 `ctest` 命令运行。使用举例:

```
# 添加一个测试, 执行可执行文件 test_app 并传递参数
add_test(NAME TestAppWithArgs COMMAND test_app --option1 --option2 value)
```

- **set\_tests\_properties(<tests>... [DIRECTORY <dir>] PROPERTIES <prop1> <value1> [<prop2> <value2>]...)**: 设置测试属性的命令, 这些属性可以控制测试的行为, 比如超时、工作目录、环境变量、依赖关系等。

`set_tests_properties`使用举例[1]: 设置测试超时

```
# 指定测试属性
set_tests_properties(TestAppWithArgs PROPERTIES TIMEOUT 30)
# 定义了一个名为 TestAppWithArgs 的测试, 它执行 test_app 并传递两个命令行参数。
# 设置了测试的超时时间为 30 秒。
```

`set_tests_properties`使用举例[2]: 设置工作目录

```
# 添加一个测试
add_test(NAME MyTest COMMAND my_test_executable)

# 设置测试的工作目录为项目的二进制目录
set_tests_properties(MyTest PROPERTIES WORKING_DIRECTORY ${CMAKE_BINARY_DIR})
```

`set_tests_properties`使用举例[3]: 设置环境变量

```
# 设置环境变量 TEST_ENV 为 "test_value"
set_tests_properties(MyTest PROPERTIES ENVIRONMENT "TEST_ENV=test_value")
```

set\_tests\_properties使用举例[4]:设置测试依赖

```
# 添加两个测试
add_test(NAME MyTest COMMAND my_test_executable)
add_test(NAME MyOtherTest COMMAND my_other_test_executable)

# 设置 MyOtherTest 依赖于 MyTest
set_tests_properties(MyOtherTest PROPERTIES DEPENDS MyTest)
```

- **install(TARGETS <目标>... [EXPORT <名称>] [RUNTIME|LIBRARY|ARCHIVE|... DESTINATION <路径>] [INCLUDES DESTINATION <路径>])**: TARGETS: 指定需要安装的目标（如可执行文件或库）。RUNTIME: 安装运行时目标（适用于可执行文件）。LIBRARY: 安装共享库。ARCHIVE: 安装静态库。EXPORT: 导出目标，用于生成 CMake 的导入文件。DESTINATION: 安装的目标路径。INCLUDES DESTINATION: 指定头文件安装路径。
- **install(FILES <文件>... DESTINATION <路径>)**: FILES: 安装单个文件。DESTINATION: 安装的目标路径。使用举例:

```
install(TARGETS main DESTINATION bin)
```

- **install(DIRECTORY <目录>... DESTINATION <路径> [...])**: DIRECTORY: 安装目录。DESTINATION: 安装的目标路径。使用举例:

```
# 安装目标
install(TARGETS my_executable my_library
        RUNTIME DESTINATION bin
        LIBRARY DESTINATION lib
        ARCHIVE DESTINATION lib/static)

# 安装文件
install(FILES README.md LICENSE DESTINATION share/doc)

# 安装目录
install(DIRECTORY include/ DESTINATION include)
```

## 消息和调试

- **message([<模式>] <消息内容>)**: 打印消息以帮助调试。<模式>: STATUS: 输出普通状态信息。使用举例:

```
message(STATUS "Configuring project...")
message(WARNING "This is a warning message.")
message(FATAL_ERROR "Configuration failed!")
```

- **debug**: CMake 本身没有 debug 指令，但可以通过 message 和变量检查来实现调试功能。

```
set(MY_VAR "value") message(STATUS "MY_VAR = ${MY_VAR}")
```

- **WARNING**: 输出警告信息，**FATAL\_ERROR**: 输出错误信息并终止执行，**DEPRECATION**: 输出弃用信息。<消息内容>: 需要输出的内容，

## 宏和函数

- **macro**：定义可复用的宏。使用举例：

```
macro(print_message MESSAGE)
    message(STATUS ${MESSAGE})
endmacro()

print_message("Hello, Macro!")
```

- **function**：定义可复用的函数（支持作用域隔离）。使用举例：

```
function(add_library_and_link TARGET_NAME LIB_NAME)
    add_library(${LIB_NAME} STATIC ${LIB_NAME}.cpp)
    target_link_libraries(${TARGET_NAME} PRIVATE ${LIB_NAME})
endfunction()

add_library_and_link(main mylib)
```

- **return**：在函数中退出。
- **cmake\_parse\_arguments**：解析函数参数。

## 自定义命令和操作

- **add\_custom\_command**(OUTPUT <生成的文件> COMMAND <命令> [ARGS <参数>...] [DEPENDS <依赖文件>...] [WORKING\_DIRECTORY <目录>] [COMMENT <注释>] [VERBATIM])：自定义命令与生成器。OUTPUT：命令生成的文件或目标。COMMAND：执行的命令。DEPENDS：命令的依赖文件或目标。WORKING\_DIRECTORY：命令执行时的工作目录。VERBATIM：确保命令参数被正确转义。
- **add\_custom\_target**：定义自定义目标。
- **execute\_process**：在置阶段执行外部命令或脚本，并可以捕获其输出。这个命令通常用于执行系统命令、编译器或工具，以及获取系统信息等。

语法格式：

```
execute_process(COMMAND <cmd1> [<arguments>]
               [COMMAND <cmd2> [<arguments>]]...
               [WORKING_DIRECTORY <directory>]
               [TIMEOUT <seconds>]
               [RESULT_VARIABLE <variable>]
               [RESULTS_VARIABLE <variable>]
               [OUTPUT_VARIABLE <variable>]
               [ERROR_VARIABLE <variable>]
               [INPUT_FILE <file>]
               [OUTPUT_FILE <file>]
               [ERROR_FILE <file>]
               [OUTPUT_QUIET]
               [ERROR_QUIET]
               [COMMAND_ECHO <where>]
               [OUTPUT_STRIP_TRAILING_WHITESPACE]
               [ERROR_STRIP_TRAILING_WHITESPACE]
               [ENCODING <name>]
               [ECHO_OUTPUT_VARIABLE]
```



execute\_process使用举例[1]:执行系统命令并捕获输出

```
# 执行命令并捕获输出
execute_process(
    COMMAND ${CMAKE_COMMAND} -E echo "Hello from execute_process"
    OUTPUT_VARIABLE output
    ERROR_VARIABLE error
    RESULT_VARIABLE result
)

# 打印输出和错误信息
message(STATUS "Output: ${output}")
message(STATUS "Error: ${error}")
message(STATUS "Result: ${result}")
```

execute\_process使用举例[2]:执行命令并使用输入

```
# 执行命令并指定工作目录
execute_process(
    COMMAND ${CMAKE_COMMAND} -E echo "Current working directory is ${CMAKE_CURRENT_SOURCE_DIR}"
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
    OUTPUT_VARIABLE output
)

# 打印输出
message(STATUS "Output: ${output}")
```

- **add\_dependencies(<target> [<target-dependency>]...)**: 为目标（通常是可执行文件或库）添加依赖关系。这意味着在构建指定目标之前，必须先构建它所依赖的其他目标。

```
# 创建一个自定义命令，生成一个中间文件
add_custom_command(
    OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/generated_file.h
    COMMAND ${CMAKE_COMMAND} -E touch ${CMAKE_CURRENT_BINARY_DIR}/generated_file.h
    DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/source_file.cpp
    COMMENT "Generating file"
)

# 创建一个自定义目标，用于生成文件
add_custom_target(GenerateHeader DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/generated_file.h)

# 创建一个可执行文件
add_executable(my_executable main.cpp)

# 为目标 my_executable 添加依赖，确保在编译 my_executable 之前生成文件
add_dependencies(my_executable GenerateHeader)
```

## 模块管理

- **include(<文件名>)**: 加载其他 CMake 脚本。使用举例:

```
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(MKL DEFAULT_MSG MKL_INCLUDE_DIRS)
# MKL: 指定当前查找的包或库的名称。在这个例子中，目标是 MKL（英特尔数学核心函数库）
# DEFAULT_MSG: 指定消息的类型。DEFAULT_MSG 用于生成默认的成功或失败消息，这些消息将告诉用户
# MKL_INCLUDE_DIRS: 传递库查找过程中设置的变量。如果这些变量被正确地设置了，那么函数将认为这个
```

- `list(<操作> <列表变量> [参数]...)`：操作列表变量。 `APPEND`：向列表末尾添加元素。  
`REMOVE_ITEM`：移除指定元素。 `LENGTH`：获取列表长度。 `GET`：获取指定索引的值。  
`SORT`：对列表排序。使用举例：

```
set(MY_LIST a b c)
list(APPEND MY_LIST d e)
list(SORT MY_LIST)
message(STATUS "MY_LIST = ${MY_LIST}")
```

通过这些命令，CMake 可以灵活地处理复杂的构建需求，自动化管理编译和链接过程。它不仅限于 C/C++ 项目，也支持多种其他语言和工具链，使得它成为现代软件工程中不可或缺的一部分。

## CMake预定义变量

CMake 中的预留名主要包括预定义变量，这些变量是 CMake 在构建系统时自动设置的，用于提供构建环境和项目配置的详细信息。以下是一些常用的 CMake 预定义变量

### 1. 项目和源码路径相关：

- **CMAKE\_SOURCE\_DIR**：指向包含顶层 CMakeLists.txt 文件的目录的完整路径，是整个项目的源码根目录。
- **PROJECT\_SOURCE\_DIR**：指向当前处理的 CMakeLists.txt 所在项目的源码目录。在多项目（子项目）配置中，它指向当前项目的源码目录。
- **CMAKE\_BINARY\_DIR**：指向顶级构建目录的路径，即运行 CMake 配置命令的目录。
- **PROJECT\_BINARY\_DIR**：指向当前项目的构建目录的路径。在多项目配置中，每个项目可以有各自的构建目录。

### 2. 系统和平台信息：

- **CMAKE\_SYSTEM\_NAME**：系统名称，如 Windows、Linux、Darwin (macOS) 等。
- **CMAKE\_SYSTEM\_PROCESSOR**：处理器架构，如 x86\_64、arm64 等。

### 3. CMake 版本信息：

- **CMAKE\_VERSION**：正在运行的 CMake 的版本号。

### 4. 安装配置：

- **CMAKE\_INSTALL\_PREFIX**：定义安装目标时的前缀路径，默认通常是 /usr/local 在 UNIX 系统上，或者是一个基于 Program Files 的路径在 Windows 上。

### 5. 其他系统信息：

- **CMAKE\_SYSTEM\_VERSION**：系统的版本号或其他版本信息。
- **CMAKE\_HOST\_SYSTEM\_NAME**：构建 CMake 时使用的系统名称，与 `CMAKE_SYSTEM_NAME` 类似，但适用于交叉编译<sup>+</sup>。
- **CMAKE\_HOST\_SYSTEM\_PROCESSOR**：CMake 构建时使用的系统的处理器架构。
- **CMAKE\_CROSSCOMPILING**：如果当前构建是交叉编译，则此变量值为真。
- **CMAKE\_SIZEOF\_VOID\_P**：CMake 编译器检测到的目标体系结构的 void \* 的大小，可用于确定是 32 位还是 64 位系统。

## 练练手

写在最后

全文很长，感谢你能看完。整理更是不易，如有谬误，请你及时指正！我们团队在持续探索CAE<sup>+</sup>相关技术和开源代码。为了更好地促进青年人员互助，我们也发起了计算力学、[多场耦合<sup>+</sup>](#)、MFEM、FEniCS(x)等高质量交流社群（拒绝任何广告）。感兴趣的可以加我为好友一起交流（备注要加什么群）。

Q ID：19561403080。Wechat ID: ym\_cae


MFEM 中文交流群（QQ）：760401312

FEniCS 中文交流群（QQ）：375342576

[数值仿真计算<sup>+</sup>](#)交流群（QQ）：738423273

编辑于 2024-12-23 11:51 · IP 属地北京

内容所属专栏



**C++编程进阶**  
C++进阶学习笔记

订阅专栏

[CMake](#) [跨平台开发](#) [C / C++](#)







理性发言，友善互动

2 条评论

默认 最新



**杨明** 作者      
几万字整理不容易呀  
12-22 · 北京

 回复  喜欢



**Olive**  
好😁  
12-22 · 北京

 回复  喜欢

推荐阅读



**CMake应用：基础篇**  
很酷的程序... 发表于CMake...



**CMake教程系列-00-简介**  
余朔钰

**cmake使用方法**  
cmake使用方法（  
助文档技巧命令  
add\_compile\_opt  
& add\_defin  
法说明列表和字符  
号，字符...  
杰特JET