**CMakeList 配置说明**

[1. 适用版本号 2](#_Toc1768376722)

[2. CMake build system 2](#_Toc1175605132)

[ binary target 2](#_Toc866306592)

[ binary excutable 2](#_Toc169720371)

[3. 常用配置说明 3](#_Toc2117913285)

[ add\_excutable() 3](#_Toc1363190337)

[ add\_subdirectory() 3](#_Toc921609179)

[ include directories () 4](#_Toc1308254046)

[ target\_link\_directories() 4](#_Toc1906962711)

[ target\_link\_libraries() 5](#_Toc1642392521)

[ target property 6](#_Toc419765298)

[ link\_directories property 7](#_Toc1810492014)

[ interface\_link\_directories property 7](#_Toc615796183)

# 适用版本号

3.19.1

# CMake build system

基于cmake构建系统，是组织一组高水准的二进制target；每个target对应一个可执行文件或者一个库，或者含有自定义命令的自定义的target;在构建 系统之内，target 之间的依赖的表达，决定了构建顺序和回应变化重新产生的规则；

## binary target

可执行文件通过add\_excutable()产生；库通过add\_library()产生；这些产生的二进制文件又有合适的前缀和后缀，以及目标平台的扩展；二进制target之间的依赖通过使用target\_link\_libraries()这个命令表达；

add\_library(archive archive.cpp zip.cpp lzma.cpp)

add\_executable(zipapp zipapp.cpp)

target\_link\_libraries(zipapp archive)

archive 是一个静态的库，该archive 包含编译archive.cpp zip.cpp,lzma.cpp之后的实例；zipapp 定义成为一个可执行的target,通过编译和链接zipapp.cpp；当链接zipapp可执行executable过程中，静态库archive 也被链接进来；

## pseudo target(伪 target)

有一些target 类型没有构建系统的输出，而是引入外部依赖，别名，或者其他non-build 产品；Pseudo target就是这样的target;

## improted target

一个IMPORTED target 代表先前存在的依赖；这样的target一般被上游的包的定义，而且是固定的；在声明一个IMPORTED target之后可以通过使用target\_compile\_de finitions(),target\_include\_directories(),target\_compile\_options()或者target\_link\_ libraries()改变属性；

LOCATION 可以从IMPORTED target中读取出来；IMPORTED target的作用域是它定义的地方；可以被子目录访问，但父目录和同级的不行；也可以定义GLOBAL IMPORTED target 在整个构建系统都可以访问；

## Alias Target

别名target 是一个在只读背景下与一个二进制target相互交换的名称；别名target可以用于某个库的可执行单元测试，可以是同一构建系统也可以是基于用户配置；

add\_library(lib1 lib1.cpp)

install(TARGETS lib1 EXPORT lib1Export ${dest\_args})

install(EXPORT lib1Export NAMESPACE Upstream:: ${other\_args})

add\_library(Upstream::lib1 ALIAS lib1)

In another directory, we can link unconditionally to the Upstream::lib1 target, which may be an IMPORTED target from a package, or an ALIAS target if built as part of the same buildsystem.

if (NOT TARGET Upstream::lib1)

find\_package(lib1 REQUIRED)

endif()

add\_executable(exe1 exe1.cpp)

target\_link\_libraries(exe1 Upstream::lib1)

别名target 是 not-mutable ,non-installable non-exportable;一个名称可以通过ALISED\_TRAGET 属性读取出来测试一下；

get\_target\_property(\_aliased Upstream::lib1 ALIASED\_TARGET)

if(\_aliased)

message(STATUS "The name Upstream::lib1 is an ALIAS for ${\_aliased}.")

endif()

## binary excutable

add\_executable()命令定义了以个可以执行的target;

类似于add\_xxx()这样的命令，产生一条这样的规则--在构建时期使用一个可执行的target作为一个可执行的命令；这种构建规则可以确保可执行target 在运行之前先构建；

## binary library

默认情况下，add\_library()产生一个STATIC库，除非指定其他类型（SHARED,MODULE);

BUILD\_SHARED\_LIBS 变量也可以改变改变add\_library()的默认行为；

在整个构建系统背景下，一个库是SHARED 还是STATIC,是无关紧要的；依赖规范和其他APIs 会忽略库的类型对二者之间的区分；

MODULE 库是不同于以上二者的，一般来说MODULE库不会链接target;使用运行时技术作为一个插件加载；如果一个库不输出任何符号，这个库不能是SHARED库（SHARED库必须至少输出一个符号）；

SHARED 库可能被标记为创建macOS或者iOS框架类型的target属性;

add\_library(MyFramework SHARED MyFramework.cpp)

set\_target\_properties(MyFramework PROPERTIES

FRAMEWORK TRUE

FRAMEWORK\_VERSION A # Version "A" is macOS convention

MACOSX\_FRAMEWORK\_IDENTIFIER org.cmake.MyFramework

)

对象库定义了 一组非归档的对象文件，右提供的源码文件编译而来；这些对象文件通过$<TARGET\_OBJECTS:name> 语法添加到其他target;

add\_library(archive OBJECT archive.cpp zip.cpp lzma.cpp)

add\_library(archiveExtras STATIC $<TARGET\_OBJECTS:archive> extras.cpp)

add\_executable(test\_exe $<TARGET\_OBJECTS:archive> test.cpp)

target的链接步骤会使用它资源文件之外的对象文件集合；或者对象库链接到其他target;

add\_library(archive OBJECT archive.cpp zip.cpp lzma.cpp)

add\_library(archiveExtras STATIC extras.cpp)

target\_link\_libraries(archiveExtras PUBLIC archive)

add\_executable(test\_exe test.cpp)

target\_link\_libraries(test\_exe archive)

target的链接步骤会从对象库中使用对象文件，此外在编译target源文件的时候，对象库的使用要求将得到满足，更进一步说，这些使用要求会传递给target的依赖；

对象库 不能作为add\_xx(target)中的target,但是库中的对象文件可以通过$<TARGET\_OBJECTS: objlib>语法作为add\_xx(target output)中的output部分；

## build specification and useage requirements

target\_include\_directories(),target\_compile\_definitions()和target\_compile \_op tions()命令指定了二进制target的构建规范和使用要求；以上命令分别产生INCLUDE\_DIRECORIES,COMPILE\_DEFINITIONS, and COMPILE\_OPTION 这些目标属性；或者INTERFACE\_INCLUDE\_DIRECTORIES, INTERFACE\_COMPILE\_DEFI NITIONS 和INTERFACE\_COMPILE\_OPTIONS这些目标属性；INTERFACE\_INCLUDE \_DIRECTORIES,INTERFACE\_COMPILE\_DEFINITIONS and INTERFACE\_COMPILE \_O PTIONS 这些目标属性.

以上命令有用三种模式PRIVAT,PUBLIC 和INTERFACE;PRIVATE模式只会产生non\_INTERFACE 目标属性变量；INTERFACE模式只会产生INTERFACE目标属性变量；

PUBLIC模式会产生以上两种目标属性变量；

target\_compile\_definitions(archive

PRIVATE BUILDING\_WITH\_LZMA

INTERFACE USING\_ARCHIVE\_LIB

)

Note that usage requirements are not designed as a way to make downstreams use particular COMPILE\_OPTIONS or COMPILE\_DEFINITIONS etc for convenience only. The contents of the properties must be requirements, not merely recommendations or convenience.

See the Creating Relocatable Packages section of the cmake-packages(7) manual for discussion of additional care that must be taken when specifying usage requirements while creating packages for redistribution.

## target property

在编译二进制目标的源文件时候，INCLUDE\_DIRECTORIES; COMPILE\_DEFINITIONS; COMILE\_OPTIONS 以上这些target属性被合理的使用；

在INCLUDE\_DIRECTORIES属性的条目以-I或者-isystem 为前缀并且按照定义的顺序添加到编译行中；

在COMPILE\_DEFINITIONS的属性中的条目以-D 或者D为前缀以未指明的顺序添加到编译行中；DEFINE\_SYMBOL 目标属性被添加到编译定义，作为处理SHARED 和 MODULE库目标的一种便利；

在COMPILE\_OPTIONS 属性中的条目，将针对shell进行转义；若干编译选项有特别的处理，例如POSITION\_INDEPENDENT\_CODE(该属性在MODULE 和 SHARED 库默认是true,其他是false);

INTERFACE\_INCLUDE\_DIRECTORIES ，INTERFACE\_COMPILE\_DEFINITIONS 和 INTERFACE\_COMPILE\_OPTIONS 这些目标属性的内容是使用要求；以上这些属性指定了使用者必须使用的内容，以对target进行正确编译和链接；

对于任何二进制目标，使用在target\_link libraries()命令中指定的每个目标上的每个接口属性的内容

set(srcs archive.cpp zip.cpp)

if (LZMA\_FOUND)

list(APPEND srcs lzma.cpp)

endif()

add\_library(archive SHARED ${srcs})

if (LZMA\_FOUND)

# The archive library sources are compiled with -DBUILDING\_WITH\_LZMA

target\_compile\_definitions(archive PRIVATE BUILDING\_WITH\_LZMA)

endif()

target\_compile\_definitions(archive INTERFACE USING\_ARCHIVE\_LIB)

add\_executable(consumer)

# Link consumer to archive and consume its usage requirements. The consumer

# executable sources are compiled with -DUSING\_ARCHIVE\_LIB.

target\_link\_libraries(consumer archive)

因为将源码和相应的build的目录添加到相应的INCLUDE\_DIRECTORIES属性的要求是常见的,CMAKE\_INCLUDE\_CURRENT\_DIR变量可以方便的添加到所有target的INCLU DE \_DIRECTORIES属性;CMAKE\_INCLUDE\_CURRENT\_DIR\_IN\_INTERFACE 变量可以添加到所有的target的INTERFACE\_INCLUDE\_DIRECTORIES属性; target \_link \_libra ries()命令 可以更方便的不同目录的target;

## directory-scoped command

target\_include\_directories(),target\_compile\_definitions()和target\_compile\_opti ons()这些命令只能影响一个target;add\_compile\_definitions(),add\_compile\_option s()和include\_directories()功能类似但是在目录水平上的作用域，而不是target上的作用域；

## Interface Library

接口类型的库只有头文件，接口类型库不编译源码并且不产生库文件，因此没有LOCATION;

但是它会指定使用要求例如（INTERFACE\_INCLUDE\_DIRECTORIES, INTERFACE\_COM

PILE\_DEFINITIONS, INTERFACE\_COMPILE\_OPTIONS, INTERFACE\_LINK\_LIBRARIE

S,INTERFACE\_SOURCES,INTERFACE\_POSITION\_INDEPENDENT\_CODE);只有INTE RFACE模式下，target\_include\_directories(),target\_compile\_definition()和tar get\_compile\_options(),target\_sources()和target\_link\_libraries()这些命令才能使用接口类型的库；

add\_library(Eigen INTERFACE

src/eigen.h

src/vector.h

src/matrix.h

)

target\_include\_directories(Eigen INTERFACE

$<BUILD\_INTERFACE:${CMAKE\_CURRENT\_SOURCE\_DIR}/src>

$<INSTALL\_INTERFACE:include/Eigen>

)

add\_executable(exe1 exe1.cpp)

target\_link\_libraries(exe1 Eigen)

以上这个例子, 来自Eigen接口类型库的的使用要求在编译时期使用，不会影响链接；

add\_library(pic\_on INTERFACE)

set\_property(TARGET pic\_on PROPERTY INTERFACE\_POSITION\_INDEPENDENT\_CODE ON)

add\_library(pic\_off INTERFACE)

set\_property(TARGET pic\_off PROPERTY INTERFACE\_POSITION\_INDEPENDENT\_CODE OFF)

add\_library(enable\_rtti INTERFACE)

target\_compile\_options(enable\_rtti INTERFACE

$<$<OR:$<COMPILER\_ID:GNU>,$<COMPILER\_ID:Clang>>:-rtti>

)

add\_executable(exe1 exe1.cpp)

target\_link\_libraries(exe1 pic\_on enable\_rtti)

上面这个例子target exe1，是一个被链接的并且编译器指定的标志，封装在接口类型库中；

接口库可以被安装和输出，它们必须被单独安装；

set(Eigen\_headers

src/eigen.h

src/vector.h

src/matrix.h

)

add\_library(Eigen INTERFACE ${Eigen\_headers})

target\_include\_directories(Eigen INTERFACE

$<BUILD\_INTERFACE:${CMAKE\_CURRENT\_SOURCE\_DIR}/src>

$<INSTALL\_INTERFACE:include/Eigen>

)

install(TARGETS Eigen EXPORT eigenExport)

install(EXPORT eigenExport NAMESPACE Upstream::

DESTINATION lib/cmake/Eigen

)

install(FILES ${Eigen\_headers}

DESTINATION include/Eigen

)

## output artifacts

### runtime artifacts

在DLL平台上（windows),通过add\_library(SHARED)创建；或者通过add\_excutable()创建可执行文件；

通过RUNTIME\_OUTPUT\_DIRECTORY 和 RUNTIME\_OUTPUT\_NAME,可以改变构建产品的路径和文件名字；

### library output artifact

通过add\_library(MODULE) 创建的文件；

在NON-DLL平台上，add\_library(SHARED)创建的文件；

通过LIBRARY\_OUTPUT\_DIRECTORY 和LIBRARY\_OUTPUT\_NAME 可以改变 库类型输出的路径和文件名子；

### archive output artifact

add\_library(STATIC) 产生.a文件；

通过ARCHIVE\_OUTPUT\_DIRECTORY 和ARCHIVE\_OUTPUT\_NAME 控制压缩文件的输出路径和文件名字；

On DLL platforms: the import library file (e.g. .lib) of a shared library target created by the add\_library() command with the SHARED option. This file is only guaranteed to exist if the library exports at least one unmanaged symbol.

On DLL platforms: the import library file (e.g. .lib) of an executable target created by the add\_executable() command when its ENABLE\_EXPORTS target property is set.

On AIX: the linker import file (e.g. .imp) of an executable target created by the add\_executable() command when its ENABLE\_EXPORTS target property is set.

## transitive useage requirements

传递使用条件，即一个target使用条件可以传递给它的依赖；target\_link\_libraries()这个命令，通过PUBLIC PRIVATE INTERFACE,这些关键字控制这种传递；

在下面的例子中，archive 是archiveExtras的PUBLIC 类型的依赖，这个PUBLIC使用条件也会传递给customer;serial 是archiveExtras的PRIVATE类型的依赖，这个PRIVATR使用条件不会传递给customer;

一般来说，使用target\_link\_libraries(PRIVATE)指定的使用条件，依赖只存在库的源文件中，不在库的头文件中；如果还需要依赖库的头文件，需要使用target\_link\_libraries(PUBLIC) 命令作为PUBLIC 依赖的使用条件；如果只是仅仅依赖库的头文件，只需要使用target\_link\_libraries(Interface)这个命令；

add\_library(archive archive.cpp)

target\_compile\_definitions(archive INTERFACE USING\_ARCHIVE\_LIB)

add\_library(serialization serialization.cpp)

target\_compile\_definitions(serialization INTERFACE USING\_SERIALIZATION\_LIB)

add\_library(archiveExtras extras.cpp)

target\_link\_libraries(archiveExtras PUBLIC archive)

target\_link\_libraries(archiveExtras PRIVATE serialization)

# archiveExtras is compiled with -DUSING\_ARCHIVE\_LIB

# and -DUSING\_SERIALIZATION\_LIB

add\_executable(consumer consumer.cpp)

# consumer is compiled with -DUSING\_ARCHIVE\_LIB

target\_link\_libraries(consumer archiveExtras)

# 常用配置说明

## add\_excutable()

add\_excutable(<name>[WIN32][MACOSX\_BUNDLE][EXCLUDE\_FROM\_ALL][source1][source2...])

name 表示一个可执行的target，由source1 source2等源文件编译而来；如果稍后通过target\_source()指定源文件，这里可以省略source1 source2; name必须在整个工程内独一无二的；实际构建可执行的文件名字取决与本机平台惯例；

默认情况下，可执行文件创建在与源码一致的构建目录里；可以通过RUNTIME\_OUTPUT\_DIRECTORY 改变可执行文件位置；通过OUTPUT\_NAME 属性可以改变name之后文件的部分名字；

如果WIN32属性被设置，WIN32\_EXECUTABLE 将会设置到创建的target 上；

如果MACOSX\_BUNLDE属性被设置，MACOSX\_BUNDLE 将会设置到创建的target 上；

EXCLUDE\_FROM\_ALL 若该属性为true则，该target 排除当前目录其他target 以及父目录中的target,意味着运行该目录下或者父目录下的make该target 不会被构建； 若该属性为false 则该target包含当前目录的其他的target 以及父目录中的target;如果该属性没有设置,那么该target 被包含在当前目录所有的target中，更深一点讲，被包含在父目录下所有没有EXCLUDE\_FROM\_ALL 属性的target 中；

## add\_subdirectory()

add\_subdirectory(source\_dir [binary\_dir] [EXCLUDE\_FROM\_ALL]

添加要构建的子目录.source\_dir是放置CMakeList.txt 和源码的目录。可以是相对当前路径的相对路径，也可以是绝对路径；binary\_dir指定了构建之后的输出文件目录，如果是相对路径，会以当前输出目录（current\_output\_dir)计算路径，当然也可以是绝对路径；

在binary\_dir没有指定的情况下，会使用（未展开的相对路径）source\_dir代替；在add\_subdirectory处理source\_dir下的源文件之前，source\_dir下的CMakeList.txt文件会被CMake工具处理。

如果EXCLUDE\_FROM\_ALL 被指定，那么默认情况下source\_dir的target 就不能包含在父目录的target中，并且会被排除在IDE工程文件之外；在source\_dir中用户必须显式地构建target;这意味着当source\_dir是工程有用但不是必须的完全独立的一部分（例如一组例子）；尤其是source\_dir 是有自己的project(),以便在source\_dir中产生完整的构建文件；

值得注意的是内联的target会取代这种排斥；一个由父目录构建的target,并且依赖于子目录中的target，这种依赖性的target 会包含在父工程构建系统以满足依赖性；

## include directories ()

include\_directories([AFTER][BEFORE][SYSTEM]dir1[dir2...])

添加dir1 dir2 等目录给编译器搜索include files(头文件);　相对路径就是相对语当前源码的文件夹；这些目录被添加到当前的CMakeLists.txt 的INCLUDE\_DIRECTORIES 目录属性；这意味着这些目录也被添加到该CMakeList.text 内所有的target的INCLUDE\_DIRECTOR　IES 目标属性；这些目标属性值将会被生成器使用；

默认情况下这些目录会被添加当前目录的后面；也可以通过AFTER BEFORE 改变这种行为；当SYSTEM 被提供的时候，会告诉编译器这是某些平台的头文件目录；这些设置将产生一些效果，例如编译器忽略某些警告或者在依赖计算中固定安装系统不被考虑；

值得注意的是target\_include\_directories() 可以替代它；添加INCLUDE\_DIRECTORIES 给特定的target;

## target\_link\_directories()

target\_link\_directories(<target> [BEFORE]

<INTERFACE|PUBLIC|PRIVATE> [items1...]

[<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])

当链接一个指定的target的时候,指定Linker(链接器）搜寻库的路径；每个item可以是相对路径也可以绝对路径，这些路径会被添加到link命令行中；target 必须是add\_excuteable() 或者是add\_library()命令创建的，绝对不能是别名target;

INTERFACE PUBLIC PRIVATE 指定items的作用域，PRIVATE 和PUBLIC的items路径会产生target的LINK\_DIRECTORIES的属性；PUBLIC和INTERFACE 的item路径会产生target 的INTERFACE\_LINK\_DIRECTORIES属性;每一个item指向一个链接目录，并且在添加它到相关属性之前转换成绝对路径；按照调用的顺序，重复调用同一个target添加item;

如果Before被指定，item 会被添加到相关属性的前面而不是按顺序添加；

值得注意的是这个命令很少情况下是必须的，在有其他选择的情况下应该避免使用；更倾向于传库的绝对路径，以确保正确的库被链接到；find\_library()命令提供了求完整路径，在调用target\_link\_libraries()的时候可以使用find\_library();但target\_link\_directories()搜索库的路径包括一下场景：

1. 类似Xcode的IDE，在构建的时候用户可以切换target的架构），但是库的完整路径不能被提供，因为一个完整路径只提供一种架构（例如不是统一的二进制）
2. 库本身可能有其他希望通过RPATH机制找到的私有库依赖关系，但有些链接器无法完全解码这些路径(例如，由于存在像$ORIGIN这样的东西)

## target\_link\_libraries()

Specify libraries or flags to use when linking a given target and/or its dependents. Usage requirements from linked library targets will be propagated. Usage requirements of a target’s dependencies affect compilation of its own sources.

target\_link\_libraries(<target> ... <item>... ...)

target已经 通过add\_executable() 和add\_library()创建，并且不是别名Target;按照调用的顺序反复调用同一个target添加条目；

item 可以是一个库类型的target, 产生库类型的target的时候必须是完整路径的可链接库; 在工程范围内，该库类型的target必须通过add\_library()创建，或者作为IMPORTED 库；如果工程内创建，一个顺序依赖会自动添加到构建系统，以确保在target链接之前，库类型的target是最新的；如果一个导入库有IMPORTED\_NO\_SONAME 目标属性，CMake会让链接器（Linker)搜寻库，而不是用全路径（/usr/lib/libfoo.so becomes -lfoo）；指向target的全路径 会自动被shell转义；

item也可以是指向以个库文件的全路径；产生链接行会保留该文件的全路径；在库文件有改变的情况下，构建系统会有一个依赖；在一些场景下,CMkake 让链接器搜索库，例如一个共享库被设置为没有SONAME field; 指向库文件的完整路径会被shell自动转义；

item也可以是库的名字；产生链接行会让链接器搜索库（e.g. foo becomes -lfoo or foo.lib）；

item也可以是 link标志，item 以- 或者-framework 开头，但不能是-l,被视作链接器的标志；请注意，对于传递依赖关系而言，此类标志将与任何其他库链接项一样处理，因此通常将它们指定为不会传播到依赖关系的私有链接项是安全的

target\_link\_libraries(<target>

<PRIVATE|PUBLIC|INTERFACE> <item>...

[<PRIVATE|PUBLIC|INTERFACE> <item>...]...)

若是target 和库之间是PUBLIC,成为链接接口组成成分（库的头文件包含在target 源文件和头文件中）；若是target 和库之间是PRIVATE,不成为链接接口组成成分（库的头文件包含在target 源文件,但不包含在头文件中）；若是target 和库之间是Interface,不用于链接target（库的头文件不包含在target 源文件和头文件中）；

target\_link\_libraries(<target> <item>...)

默认情况下，库的依赖是可以传递的；当target A 链接到 targetB,链接到target A的库；则该库可以链接到target B;这种传递性的链接接口被存储到INTERFACE\_LINK\_LIBRARIES 目标属性，并且可以被重写；

## link\_directories property

用于链接共享库，模块，可执行target的目录列表；这个属性是拥有分号分割的目录列表，该列表指向特定的target;使用target\_link\_directories可以增加更多的搜索目录；当target被创建的时候，该属性被LINK\_DIRECTORIES 目录属性初始化，并且被产生器（genrator）用于设置链接器（Linker）搜索目录;

## interface\_link\_directories property

一个库的公共链接目录要求的列表。target 可以填充此属型以便发布该target 编译头文件所需的链接目录；可用通过target\_link\_directories ()并且指定PUBLIC 和INTERFACE 关键字实现；工程可以直接读取和设置属性；

当target 的依赖用target\_link\_libraries()被指定,CMakge就可以从该target所有依赖读取这个属性，以确定构建使用者的属性；

END