# CMake简介

CMake 是一个跨平台、开源的构建系统。它是一个集软件构建、测试、打包于一身的软件。它使用与平台和编译器独立的配置文件来对软件编译过程进行控制。

使用CmakeList.txt配置文件来描述构建过程。

# CMake使用流程

Cmake使用流程：

* + - * 1. 生成构建系统（buildsystem , 比如make工具对应的MakeFile ）;
        2. 执行构建，生成目标文件；
        3. 执行测试、安装和打包；

Cmake具体操作流程：

* 1. 书写CmakeLists.txt：在具有源码的工作空间建立CmakeLists.txt文件；
  2. 生成构建系统：在工作空间目录中生成构建系统，执行cmake -B build -S .,执行完成后，在项目的根目录下会创建Build目录，可以看到其中生成了Makefile文件；
  3. 执行构建：在工作空间目录下执行cmake –build build ; 也可使用make工具；
  4. build目录中有生成的可执行文件或者静态库；
  5. 运行CTest测试（可选）
  6. 安装项目（可选）

## 生成构建系统

生成构建系统（buildsystem ，比如make工具对应的Makefile）, 通过cmake命令生成构建系统；

|  |  |
| --- | --- |
| 参数 | 含义 |
| -S | 指定源文件根目录，必须包含一个CmakeLists.txt文件 |
| -B | 指定构建目录，构建生成的中间文件和目标文件的生成路径 |
| -D | 指定变量，格式为-D = ，-D后面的空格可以省略； |
| -G | 指定生成器 |

示例：

cmake -S . -B build -D CMAKE\_BUILD\_TYPE=Debug -D AUTHOR=RealCoolEngineer

详解：

* 1. 使用当前目录作为源文件目录，其中包含CmakeLists.txt文件；
  2. 使用build目录作为构建目录；
  3. 设定变量CMAKE\_BUILD\_TYPE的值为Debug；AUTHOR的值为RealCoolEngineer；

## 执行构建

cmake --build <build\_directory>

<dir> : 生成构建系统时指定的构建目录；

Ps : 如果你使用的是 Windows，可以使用 nmake、ninja 或 Visual Studio 来构建项目。

## 执行测试、安装或打包

## Visual Studio 集成 CMake

CMake 本身不会直接生成 .sln 文件，因为 .sln 是 Visual Studio 的解决方案文件格式。然而，当你使用 CMake 与 Visual Studio 集成时，CMake 可以生成 Visual Studio 项目文件（.vcxproj），并且 Visual Studio 会使用这些文件来创建一个 .sln 文件。

如果你在使用 CMake 生成构建系统时看到了 .sln 文件，这通常意味着你正在使用 Visual Studio 的 CMake 集成功能，或者你使用了某种工具或插件，它将 CMake 项目转换为 Visual Studio 解决方案。

在 Visual Studio 中使用 CMake 项目并生成 .sln 文件的步骤：

1. **安装 Visual Studio 和 CMake 工具**： 确保你的 Visual Studio 安装包括了 C++ 开发工具和 CMake 工具。在 Visual Studio 安装程序中，你可以选择 "使用 C++ 的桌面开发" 工作负载，它包含了 CMake 工具。
2. **打开 CMakeLists.txt 文件**： 在 Visual Studio 中，选择 "文件" > "打开" > "CMake..."，然后浏览到你的 CMakeLists.txt 文件所在的位置并打开它。
3. **配置 CMake**： Visual Studio 将显示 CMake 缓存和配置选项。在这里，你可以选择不同的构建类型（如 Debug 或 Release）和体系结构（如 x86 或 x64）。
4. **生成 Visual Studio 项目**： Visual Studio 将运行 CMake 来配置项目，并生成 .sln 和 .vcxproj 文件。这些文件将位于你在 Visual Studio 中创建的构建目录中，通常是在项目根目录下的 build 文件夹内。
5. **构建和运行项目**： 一旦 .sln 文件生成，你可以在 Visual Studio 中构建和运行项目，就像处理任何其他 Visual Studio 项目一样。

如果你在使用命令行生成构建系统，并且希望生成 Visual Studio 项目文件，你可以使用以下 CMake 命令：

cmake -G "Visual Studio 16 2019" -A x64 -S <source\_directory> -B <build\_directory>

这里 -G "Visual Studio 16 2019" 指定了生成 Visual Studio 2019 项目文件，-A x64 指定了生成 64 位项目，-S <source\_directory> 是你的源代码目录，而 -B <build\_directory> 是你想要存放生成的项目文件的构建目录。

执行上述命令后，CMake 将生成 .sln 和 .vcxproj 文件，你可以在 Visual Studio 中打开 .sln 文件来继续构建和调试你的项目。请注意，这个过程可能需要在你的系统上安装和配置适当的版本的 Visual Studio。

## MingW 集成 CMake

1. **安装 MinGW ：**首先，你需要安装 MinGW。你可以从 [MinGW 官方网站](http://www.mingw.org/) 或使用 [MSYS2](https://www.msys2.org/) 这样的发行版来安装 MinGW。安装时，确保选择了 C++ 编译器和其他必要的工具。
2. **配置环境变量 ：**安装 MinGW 后，你需要将 MinGW 的 bin 目录添加到系统的 PATH 环境变量中，这样你就可以在命令行中直接使用 gcc、g++ 等命令。
3. **创建构建目录 ：**在项目的源代码目录之外创建一个构建目录，以避免污染源代码：

mkdir build

cd build

1. **运行 CMake 配置命令 ：**在构建目录中，运行 CMake 来配置项目。你需要指定 MinGW 作为编译器。例如，如果你使用的是 MSYS2，可以使用以下命令：

cmake -G "MinGW Makefiles" -DCMAKE\_BUILD\_TYPE=Release -DCMAKE\_CXX\_COMPILER=g++ ..

这里 -G "MinGW Makefiles" 指定了使用 MinGW 作为生成器，-DCMAKE\_CXX\_COMPILER=g++ 指定了使用 g++ 作为 C++ 编译器。

1. **构建项目：**一旦 CMake 配置完成，你可以使用 make 命令来编译项目：

make

1. **运行测试（可选）** ：如果你的项目包含测试，可以使用 ctest 命令来运行它们：

ctest

1. **安装项目（可选）** ：如果你想安装编译的项目到系统路径，可以使用 make install 命令：

make install

# Cmake语法核心

Cmake的命令有不同类型，包括脚本命令、项目配置命令和测试命令；

Cmake语言在项目配置中组织为三种源文件类型：？？？？

# 常用指令

## 注释

* 单行注释

# 内容

* 多行注释(块注释)

#.....

#.....

## 变量

### 普通变量

设置变量可以是字符串，数字或列表；

* 命令1

set ( <variable\_name> <value> )

* 使用详解1

设置变量；

* 命令2

unset ( <variable\_name> )

* 使用详解2

取消设置的变量；

### Cache变量

Cache变量作用主要是为了提供用户配置选项，如果用户没有指定，则使用默认值；

* 命令

set( <variable> <value> ... CACHE <type> <docstring> [FORCE] )

set( CACHE\_VAR "Default cache value" CACHE STRING "A sample for cache variable" )

* 使用详解

主要为了提供可配置变量，比如编译开关；引用CACHE变量$CACHE{ <varialbe> } ;

Cache变量会被保存在构建目录下的CmakeCache.txt中，缓存起来之后是不变的，除非重新配置更新；

### 环境变量

修改当前处理进程的环境变量；

* 命令

set( ENV {<variable>} [<value>] )

set( ENV{ ENV\_VAR } "$ENV{PATH}" )

* 使用详解

引用环境：$ ENV { <varialbe> }

## CMake之$<>

生成表达式允许在CMake列表和字符串上下文中根据目标的属性或其他条件来动态地计算值；

生成表达式在构建生成阶段进行计算；

生成表达式提供了一种强大的方式来根据编译器、目标、配置类型等不同的上下文环境来设置变量值。它们可以在 target\_compile\_features、target\_compile\_options、target\_include\_directories 等命令中使用，以及在定义变量和设置条件时使用。

* 命令

$<...>

* 使用详解

1. **逻辑表达式**

$<condition:outcome> ：condition非零，表达式值为outcome；否则，表达式值为空字符串。

示例:

如果USE\_DEBUG\_MODE变量被设置为非零值，DEBUG\_MODE\_ENABLED 宏将被定义;

target\_compile\_definitions(my\_target PRIVATE $<$<BOOL:${USE\_DEBUG\_MODE}>:DEBUG\_MODE\_ENABLED>)

1. **信息表达式 ：**用于获取目标或配置的信息；

$<information> 或 $<information:input>

示例：

输出目标 <my\_target> 的所有包含目录;

message(STATUS "Include directories: $<TARGET\_PROPERTY:my\_target,INCLUDE\_DIRECTORIES>" )

1. **输出表达式 ：**基于输入参数生成一个输出；

$<operation> 或 $<operation:input>

示例：

将目标 my\_target 的所有包含目录连接成一个以 -I 开头的参数列表;

target\_include\_directories(my\_target PRIVATE $<JOIN:$<TARGET\_PROPERTY:INCLUDE\_DIRECTORIES>, -I>)

1. **构建接口表达式 ：**用于在构建时添加条件性的定义或链接选项，而不会影响已安装的库；

示例：

-Wall;-Wextra 编译选项只会在构建 my\_target 时应用，而不会传递给链接到 my\_target 的其他目标;

target\_compile\_options(my\_target PRIVATE $<BUILD\_INTERFACE:-Wall;-Wextra>)

* 示例

1. 根据目标的构建类型设置编译选项

**target\_compile\_options( my\_target PRIVATE $< $<CONFIG:Debug>:-O0 > )** : 会在 Debug 构建类型下为目标 my\_target 添加 -O0 编译选项 ；

1. 根据目标是否被链接来设置包含目录

**target\_include\_directories(my\_target PRIVATE $<$<BOOL:${some\_target}>:include/path>) :** 如果 some\_target 存在，则会添加 include/path 到目标 my\_target 的包含目录中。

1. 根据目标的类型设置不同的属性

**target\_compile\_features(my\_library $<IF:$<TARGET\_TYPE:STATIC\_LIBRARY>,cxx\_std\_11,cxx\_std\_14>) :** 如果目标 my\_library 是静态库类型，则会启用 C++11 标准；否则，启用 C++14 标准。

1. 根据编译器是否支持某个特性来设置编译选项

**target\_compile\_options(my\_target PRIVATE $<$<COMPILE\_LANGUAGE:CXX>:-std=c++17>) :** 如果编译语言是 C++，则会为目标 my\_target 添加 -std=c++17 编译选项

## CMake之message

打印日志

* 命令

message([STATUS|WARNING|AUTHOR\_WARNING|FATAL\_ERROR|SEND\_ERROR] "message to display" ... )

* 使用详解

消息的级别

1. (无) ：重要消息
2. STATUS ：非重要消息
3. WARNING：CMake 警告，会继续执行
4. AUTHOR\_WARNING：CMake 警告 (dev), 会继续执行
5. SEND\_ERROR：CMake 错误，继续执行，但是会跳过生成的步骤
6. FATAL\_ERROR：CMake 错误，终止所有处理过程

## 设置cmake最小版本

* 命令

cmake\_minimum\_required(VERSION 3.0)；

* 使用详解

可选;最好加上（CMakeLists.txt 文件中使用了一些高版本cmake特有的一些命令的时候，就需要加上这样一行，提醒用户升级到该版本之后再执行 cmake。）

## 设置cmake策略

隐式设置策略

* 命令

cmake\_policy( VERSION <min>[ ...<max> ] ) //通过版本来设置策略

* 使用详解

1. 调用cmake\_minimum\_required 会隐式调用cmake\_policy( VERSION ) , 指定版本引入的所有策略都将设置为NEW行为 ;
2. 如果没有特别的策略要指定，可以指定通过调用cmake\_mininum\_required即可，无需调用cmake\_policy ;
3. 示例

CMake最低版本为3.0 ：

cmake\_minimum\_required( VERSION 3.0 ) == cmake\_policy( VERSION 3.0 )

CMake最低版本为3.0，最高版本为3.7 ：

cmake\_minimum\_required( VERSION 3.0...3.7 ) == cmake\_policy( VERSION 3.0...3.7 )

如果需要运行 的 CMake 版本 高于 3.0，但是引入的策略为 CMake 版本 在 3.0 和 3.5 之间 ：

cmake\_minimum\_required( VERSION 3.0 )

cmake\_policy( VERSION 3.0...3.5 )

## 设置项目名称

* 命令

project( <project\_name>

[VERSION <major>[.<minor>[.<patch>[.<tweak>] ] ] ]

[DESCRIPTION <project-description-string>]

[HOMEPAGE\_URL <url-string>]

[LANGUAGES <language-name>...] )

* 使用详解

指定cmake工程的名称（VERSION ： 版本号 ， DESCRIPTION ： 项目描述 ，

HOMEPAGE\_URL : …… , LANGUAGES : 编译工程使用额语言）；

可选;最好加上（它会引入两个变量：demo\_BINARY\_DIR和demo\_SOURCE\_DIR ; 同时Cmake自动定义了两个等价的变量： PROJECT\_BINARY\_DIR 和 PROJECT\_SOURCE\_DIR ）

## 设置编译包含的源文件

搜索所有的cpp文件

* 命令

aux\_source\_directory( . SRC\_LIST ) # 搜索当前目录下的所有.cpp文件

add\_library(demo ${SRC\_LIST})

add\_library(demo demo.cpp test.cpp util.cpp)

* 使用详解

aux\_source\_directory(dir VAR) ： 发现一个目录dir下所有的源代码文件，并将列表存储在一个变量VAR中。

自定义搜索规则

* 命令

file(GLOB SRC\_LIST "\*.cpp" "protocol/\*.cpp")

add\_library(demo ${SRC\_LIST})

# 或者

file(GLOB SRC\_LIST "\*.cpp")

file(GLOB SRC\_PROTOCOL\_LIST "protocol/\*.cpp")

add\_library(demo ${SRC\_LIST} ${SRC\_PROTOCOL\_LIST})

# 或者

aux\_source\_directory(. SRC\_LIST)

aux\_source\_directory(protocol SRC\_PROTOCOL\_LIST)

add\_library(demo ${SRC\_LIST} ${SRC\_PROTOCOL\_LIST})

* 使用详解

file(GLOB files LIST\_DIRECTORIES false \*) //收集文件指令

GLOB命令将所有匹配的文件挑选出来，默认以字典顺序排序；

## 设置包含目录

* 命令

include\_directories

(

${CMAKE\_CURRENT\_SOURCE\_DIR}

${CMAKE\_CURRENT\_BINARY\_DIR}

${CMAKE\_CURRENT\_SOURCE\_DIR}/include

)

* 使用详解

Linux 下还可以通过如下方式设置包含的目录:

set( CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -I${CMAKE\_CURRENT\_SOURCE\_DIR}" )

## CMake之include

用于包含另一个 CMakeLists.txt 文件或 CMake 模块文件。这是一种将构建逻辑分解为可重用组件的方法，使得 CMakeLists.txt 文件更加清晰和易于管理。使用 include 可以让你定义公共的构建规则、查找依赖库、设置编译选项等，并在多个项目中重用这些规则。

* 命令

include( <file | module > [OPTIONAL] [RESULT\_VARIABLE <var> ]

[NO\_POLICY\_SCOPE] )

* 使用详解
  + 1. <file\_or\_module>: 要包含的 CMakeLists.txt 文件或模块文件的路径。如果是模块文件，它通常位于 Modules 目录(CMAKE\_MODULE\_PATH)中。
    2. [OPTIONAL]: 可选参数，如果指定，当文件不存在时不会生成错误。

路径：如果你不指定完整的路径，CMake 会在当前 CMakeLists.txt 文件所在的目录和 Modules 目录(CMAKE\_MODULE\_PATH)中查找指定的文件或模块。

循环依赖：避免创建循环依赖，因为这可能导致 CMake 配置失败。

模块缓存：如果你在多个项目中重用模块，确保模块不会在每次运行 CMake 时都被重新加载。一种方法是将模块放在 CMake 的安装目录中，这样它们就会被缓存，不会在每次配置时重新加载。

* 示例

若有一个名为 FindMyLibrary.cmake 的模块，你想在 CMakeLists.txt 文件中使用它来查找一个库：

# 包含 FindMyLibrary 模块

include(FindMyLibrary)

# 使用 find\_package 命令查找库

find\_package(MyLibrary REQUIRED)

## 设置链接库搜索目录

* 命令

link\_directories(${CMAKE\_CURRENT\_SOURCE\_DIR}/libs)

* 使用详解

Linux 下还可以通过如下方式设置链接库搜索的目录:

set(CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -L${CMAKE\_CURRENT\_SOURCE\_DIR}/libs")

## CMake之set\_target\_properties

* + - 命令

set\_target\_properties( target\_name

PROPERTIES property1 value1

property2 value2 ... )

* + - 使用详解

1. target\_name 是需要修改属性的目标的名称。
2. PROPERTIES 是指定属性和值的关键字。
3. property1、property2 等是要设置的属性名称。
4. value1、value2 等是相应属性的值。

## CMake之add\_executable

用于创建一个可执行文件目标。这个命令将源文件编译并链接成一个可执行程序，它是构建过程中生成最终运行文件的关键步骤;

* 命令

add\_executable(executable\_name source1 [source2 ...])

* 使用详解

1. executable\_name 是你想要创建的可执行文件的名称。这个名称也将作为构建目录中生成的可执行文件的文件名。
2. source1、source2 等是构成可执行文件的源代码文件的列表。这些文件可以是 .cpp、.c、.S 或其他 CMake 支持的源文件类型。

* 案例

# 生成库

# 1.静态库

add\_library(PrinterChar STATIC src/printer\_char.cpp)

# 2.动态库

add\_library(PrinterInt SHARED src/printer\_int.cpp)

# 添加需要构建的可执行文件

#------方法对比 1.1 - 不用库的写法

add\_executable(LinkLibraryDemo link\_library\_demo.cpp src/printer\_char.cpp src/printer\_int.cpp)

#------方法对比 1.2 - 用库的写法

add\_executable(LinkLibraryDemo demo/link\_library\_demo.cpp)

# 为目标添加静态库

target\_link\_libraries(LinkLibraryDemo PRIVATE PrinterChar)

# 为目标添加动态库

target\_link\_libraries(LinkLibraryDemo PRIVATE PrinterInt)

## CMake之add\_library

用于创建库目标的命令。这个命令用于定义一个库，它可以是静态库（.a 或 .lib），动态库（.so 或 .dll），或者对象库（.o）。库目标可以被其他目标（如可执行文件）链接，以使用库中定义的函数和数据。

* 命令

add\_library( TARGET <name> [STATIC | DYNAMIC | OBJECT] [EXCLUDE\_FROM\_ALL] source1 [source2 ...] )

* 使用详解

TARGET ：指定库的名称( <name> )；

STATIC、DYNAMIC和OBJECT ：指定了要创建的库的类型。如果不指定，CMake 默认创建静态库。

1. STATIC 创建静态库。
2. DYNAMIC 创建动态库（共享库）。
3. OBJECT 创建一个对象库，即一组编译后的对象文件，不链接成库。

EXCLUDE\_FROM\_ALL :

若指定，库目标将不会自动包含在 all 目标中，这通常用于当你只想在需要时显式构建库时。

source1 [source2 ...] ：是一个或多个源文件的列表，这些文件将被编译来创建库。

* 示例

# 生成库

# 1.静态库

add\_library(PrinterChar STATIC src/printer\_char.cpp)

# 2.动态库

add\_library(PrinterInt SHARED src/printer\_int.cpp)

库目标是可重定位的，这意味着它们可以在其他目标中被引用，无论是可执行文件还是其他库。通过 target\_link\_libraries 命令，你可以将一个目标链接到一个或多个库。

用于创建库目标，而当你使用 INTERFACE 关键字时，它会创建一个接口库。接口库是一种特殊的库目标，它不包含实际的编译代码，而是用来封装一组库的链接信息，这些库可以是其他已存在的库或者由当前项目生成的库。

* 命令

add\_library( <name> INTERFACE )

* 使用详解

<name>: 接口库的名称

接口库主要用于两种情况：

1. **封装依赖**：当你的项目依赖于其他库时，你可以创建一个接口库来封装这些依赖，使得其他目标（如可执行文件或其他库）可以简单地链接到这个接口库，而不需要关心具体的依赖细节。
2. **抽象库**：当你想要提供一个库的多个版本（例如，静态库和动态库）时，你可以创建一个接口库，然后为这个接口库定义多个别名，每个别名对应一个具体的库版本。

* 示例

假设你有一个项目，它依赖于两个不同的数学库 MathLibA 和 MathLibB。你可以创建一个接口库 MathInterface 来封装这些依赖：

# 定义接口库

add\_library( MathInterface INTERFACE )

# 链接到实际的库

target\_link\_libraries(MathInterface INTERFACE MathLibA MathLibB)

现在，其他目标可以简单地链接到 MathInterface 而不是单独链接到 MathLibA 和 MathLibB：

add\_executable(MyApp main.cpp)

target\_link\_libraries(MyApp PRIVATE MathInterface)

在这个例子中，MyApp 可执行文件将会间接链接到 MathLibA 和 MathLibB，通过 MathInterface 接口库。

## CMake之target\_sources

target\_sources用于为指定的目标（如可执行文件或库）添加源文件。

* 命令

target\_sources( TARGET target\_name

PRIVATE public\_header1 [public\_header2 ...]

PUBLIC interface\_header1 [interface\_header2 ...]

OBJECT object\_file1 [object\_file2 ...]

[BUNDLE interface\_header1 [interface\_header2 ...]]

[COMPILE\_OPTIONS ...]

[COMPILE\_DEFINITIONS ...]

[INCLUDE\_DIRECTORIES ...]

[PRE\_COMPILED\_HEADERS ...]

[SYSTEM\_INCLUDE\_DIRECTORIES ...]

[FRAMEWORKS ...]

[LIBRARIES ...] )

* 使用详解

TARGET ：指定目标名称<库文件/可执行程序>；

PRIVATE / PUBLIC  / OBJECT ：指定源文件作用域

1. PRIVATE 表示源文件仅用于指定目标，不会在链接其他目标时使用。
2. PUBLIC 表示源文件用于指定目标，并且在链接其他目标时也会被考虑。
3. OBJECT 表示源文件将被编译成对象文件，但不会直接链接到目标中，通常用于创建对象库。

* 示例

如果你已经使用 add\_executable 或 add\_library 定义了一个名为 my\_target 的目标，并且想要添加 source1.cpp 和 source2.cpp 作为其源文件，你可以这样写：

target\_sources(my\_target PRIVATE source1.cpp source2.cpp)

## CMake之target\_include\_directories

用于为指定的目标（如可执行文件或库）添加包含目录。这些目录将被编译器用来查找在源代码中通过 #include 指令引用的头文件。

* 命令

target\_include\_directories ( TARGET target\_name

[ PRIVATE | PUBLIC | INTERFACE ]

[BEFORE | AFTER]

[SYSTEM]

[ directories... ] )

* 使用详解

**TARGET** ：指定目标名称<库文件/可执行程序>；

**PRIVATE / PUBLIC  / INTERFACE** ：指定包含目录作用域

1. PRIVATE 表示目录仅用于指定目标的编译过程，不会传递给链接该目标的其他目标
2. PUBLIC 表示目录既用于指定目标的编译过程，也会传递给链接该目标的其他目标。
3. INTERFACE 表示这些目录仅用于其他链接了该目标的目标的编译过程。

**BEFORE / AFTER** ：用于控制包含目录的顺序。

BEFORE ：将指定的目录放在默认包含目录之前；

AFTER ：将指定的目录放在默认包含目录之后。

**SYSTEM** ：用于告诉编译器将这些目录作为系统目录处理 ，这可能会影响编译器的警告和错误报告。

**directories...** ：是一个或多个目录的路径列表，这些目录将被添加到包含路径中

* 示例

如果已经定义了一个名为 my\_library 的库目标，并且想要为这个目标添加一个包含目录 include，你可以这样写：

target\_include\_directories(my\_library PUBLIC include)

这将使得 include 目录中的头文件可用于 my\_library 的编译过程，并且当其他目标链接 my\_library 时，这些头文件也会被考虑。

## CMake之target\_compile\_features

target\_compile\_features用于为指定目标（如可执行文件或库）指定一组编译器特性（features），这些特性通常是 C++ 标准中的新特性。这个命令确保了目标在编译时会使用到这些特性，并且编译器会根据指定的特性来启用或禁用相应的编译选项。

* 命令

target\_compile\_features(TARGET target\_name

PRIVATE|PUBLIC|INTERFACE

features... )

* 使用详解

**TARGET** ：指定目标名称<库文件/可执行程序>。

**PRIVATE、PUBLIC 和 INTERFACE** ：指定了特性的作用域：

1. PRIVATE 表示这些特性仅用于指定目标的编译过程。
2. PUBLIC 表示这些特性用于指定目标的编译过程，并且会传递给链接该目标的其他目标。
3. INTERFACE 表示这些特性仅用于其他链接了该目标的目标的编译过程。

**features...** ：是一个或多个编译特性的列表。

* 示例

如果你想确保一个名为 my\_target 的目标在编译时支持 C++11 的特性，你可以这样写：

target\_compile\_features ( my\_target PUBLIC cxx\_std\_11 )

## CMake之target\_compile\_definitions

target\_compile\_definitions用于为指定的目标（可执行文件或库）添加预处理器定义。这些定义可以是宏定义，它们在编译过程中会被预处理器处理，从而影响代码的编译方式。这个命令可以用来定义编译时的常量、启用或禁用特定的代码段，或者提供条件编译的逻辑。

* 命令

target\_compile\_definitions( TARGET target\_name

[PRIVATE|PUBLIC|INTERFACE]

definition1 [definition2 ...])

* 使用详解

**TARGET** ：指定目标名称<库文件/可执行程序>。

**PRIVATE、PUBLIC和INTERFACE** ：指定了定义的作用域：

1. PRIVATE 表示这些定义仅用于指定目标的编译过程。
2. PUBLIC 表示这些定义既用于指定目标的编译过程，也会传递给链接该目标的其他目标。
3. INTERFACE 表示这些定义仅用于其他链接了该目标的目标的编译过程。

**definition1 [definition2 ...]** 是一个或多个预处理器定义的名称，可以是宏定义或条件编译指令。

* 示例

如果你想要为一个名为 my\_target 的目标添加一个预处理器定义 MY\_DEFINITION的宏，你可以这样写：

target\_compile\_definitions( my\_target PRIVATE MY\_DEFINITION )

## CMake之target\_link\_libraries

用于为指定目标（如可执行文件或库）添加在链接阶段所需链接的库；

* 命令

target\_link\_libraries( target\_name

[PRIVATE | PUBLIC | INTERFACE]

[items...] )

* 使用详解

target\_name：指定目标名称。

PRIVATE、PUBLIC、INTERFACE：这些关键字指定了链接库的作用域。

1. PRIVATE：链接库仅对目标本身可见，不会传递给依赖该目标的其他目标。
2. PUBLIC：链接库对目标本身可见，并且会传递给依赖该目标的其他目标。
3. INTERFACE：链接库仅对依赖该目标的其他目标可见，而不对目标本身可见。

items...：一个或多个要链接的库名称或库文件路径

* 示例

如果你想要链接一个名为 my\_library 的静态库到一个名为 my\_executable 的可执行文件 :

target\_link\_libraries(my\_executable PRIVATE my\_library)

## CMake之target\_compile\_options

用于为目标（如可执行文件或库）设置特定的编译选项。这些选项会被应用到目标的所有源文件上。这个命令提供了一种集中管理编译器选项的方法，而不需要在多个地方重复设置相同的编译选项。

* 命令

target\_compile\_options(target\_name PRIVATE|PUBLIC|INTERFACE [options...])

* 使用详解

1. target\_name: 目标的名称，它应该是使用 add\_executable 或 add\_library 定义的目标。
2. PRIVATE, PUBLIC, INTERFACE: 指定编译选项的作用范围。

PRIVATE: 选项仅对目标本身有效，不会传递给链接到该目标的其他目标。

PUBLIC: 选项对目标本身有效，也会传递给链接到该目标的其他目标。

INTERFACE: 选项仅对链接到该目标的其他目标有效，不会影响目标本身的编译。

1. [options...]: 一个或多个编译选项。

* 示例

假设你想要为一个名为 MyExecutable 的可执行文件添加一些特定的编译选项：

add\_executable(MyExecutable main.cpp)

target\_compile\_options(MyExecutable PRIVATE -Wall -Wextra)

在这个例子中，-Wall 和 -Wextra 是传递给编译器的选项，它们会被应用到 main.cpp 文件的编译过程中。使用 PRIVATE 关键字意味着这些选项不会传递给链接到 MyExecutable 的其他目标。

## CMake之add\_dependencies

用于为指定的一个或多个目标添加依赖于其他目标；

* 命令

add\_dependencies ( target1 [target2 ...]

[dependency1 [dependency2 ...]] )

* 使用详解

target1、target2 : 是依赖其他目标的目标名称列表；

dependency1、dependency2 : 是要依赖的目标名称列表

* 示例

如果你有一个名为 my\_application 的可执行文件，它依赖于名为 my\_library 的库，你可以使用以下命令来指定这个依赖关系：

add\_dependencies(my\_application my\_library)

## CMake之add\_subdirectory

用于将其他目录中的 CMake 项目或子项目添加到当前构建中。

* 命令

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

* 使用详解

会在当前CmakeLists.txt文件中的对应位置包含子目录中的CmakeLists.txt文件，并执行其中的命令。

source\_dir :

该参数指定一个子目录，子目录下包含CmakeLists.txt文件和代码文件。

子目录可以是相对路径也可以是绝对路径；

若是相对路径，则是相对当前目录的一个相对路径；

binary\_dir :

该参数指定一个目录，用于存放输出文件。

可以是相对路径也可以是绝对路径；

若是相对路径，则是相对当前输出目录的一个相对路径；

若该参数未指定，则默认的当前输出目录使用source\_dir;

EXCLUDE\_FROM\_ALL ：

可选参数。

当指定了该参数，则子目录下目标文件(可执行文件或库文件)不会被父目录下的目标文件(可执行文件或库文件)包含进去，父目录的CMakeLists.txt不会构建子目录的目标文件(可执行文件或库文件)，必须在子目录下显式构建。

若：当父目录的目标依赖于子目录的目标文件(可执行文件或库文件)，则子目录目标文件(可执行文件或库文件)仍然会被构建出来以满足依赖关系；（例如使用了：target\_link\_libraries）

示例：使用add\_subdirectory()编译库

考虑一种情况，你正在编写一个项目，其中包含一个库（假设名为my\_lib）和一个可执行文件（假设名为my\_exec）。你可能希望把库的源代码放在一个子目录中，然后在主目录中编译可执行文件。

你的项目结构可能如下：

project/

|-- CMakeLists.txt

|-- main.cpp

`-- my\_lib/

|-- CMakeLists.txt

`-- my\_lib.cpp

在my\_lib/CMakeLists.txt中，你可能有以下内容：

add\_library(my\_lib my\_lib.cpp)

在主CMakeLists.txt文件中，你需要添加my\_lib子目录，并链接my\_lib库以编译my\_exec：

cmake\_minimum\_required(VERSION 3.10)

project(my\_project)

add\_subdirectory(my\_lib)

add\_executable(my\_exec main.cpp)

target\_link\_libraries(my\_exec my\_lib)

这里，通过在主CMakeLists.txt文件中调用add\_subdirectory(my\_lib)，CMake会进入my\_lib子目录并处理其中的CMakeLists.txt文件。然后在主目录中，你可以引用在子目录中定义的目标（在这种情况下是my\_lib库），并将其链接到你的可执行文件。

## CMake之add\_custom\_command

允许你为目标（target）添加自定义的构建步骤。该命令通常与 add\_custom\_target 配合使用，以便在构建过程中执行特定的命令序列。add\_custom\_command 可以用来执行编译器无法直接处理的任务，比如运行自定义的脚本或者程序。

* 命令

add\_custom\_command( [TARGET target\_name]

[PRE\_BUILD | PRE\_LINK | POST\_BUILD]

[COMMAND command1 [ARGS arg1 arg2 ...] ...]

[WORKING\_DIRECTORY dir]

[COMMENT "comment"]

[VERBATIM]

[APPEND] )

* 使用详解

1. TARGET 指定了要添加自定义命令的目标名称。
2. PRE\_BUILD、PRE\_LINK、POST\_BUILD 指定了自定义命令的执行时机。PRE\_BUILD 在编译目标之前执行，PRE\_LINK 在链接目标之前执行，POST\_BUILD 在目标构建完成后执行。
3. COMMAND 指定了要执行的命令及其参数。
4. WORKING\_DIRECTORY 指定了执行命令的工作目录。
5. COMMENT 提供了一个简短的描述，用于在构建输出中显示。
6. VERBATIM 表示直接执行命令，不通过 shell，这对于需要精确控制的情况很有用。
7. APPEND 表示将命令添加到已存在的自定义命令列表的末尾，而不是替换它们。

* 示例

PRE\_BUILD 指定了自定义命令将在 MyExecutable 目标的构建开始之前执行。COMMAND 后面跟着两个命令：第一个命令创建了一个目录，第二个命令复制了一个脚本文件到这个目录。COMMENT 参数提供了执行命令时显示的注释信息，而 VERBATIM 确保命令和参数按原样传递；

add\_executable( MyExecutable main.cpp)

add\_custom\_command(

TARGET MyExecutable

PRE\_BUILD

COMMAND ${CMAKE\_COMMAND} -E

make\_directory ${CMAKE\_CURRENT\_BINARY\_DIR}/build

COMMAND ${CMAKE\_COMMAND} -E

copy\_if\_different

${CMAKE\_SOURCE\_DIR}/pre\_build\_script.sh

${CMAKE\_CURRENT\_BINARY\_DIR}/build/pre\_build\_script.sh

COMMENT "Running pre-build script"

VERBATIM )

## CMake之add\_custom\_target

用于在构建系统中添加一个自定义的目标（target）。这个目标可以是执行一些不涉及文件生成的命令，或者用于定义一些在构建过程中需要执行的特殊任务，比如运行测试、安装文件、清理构建目录等；

* 命令

add\_custom\_target( target\_name

[ALL]

[command1 [args1...]]

[COMMAND command2 [args2...]] [...]

)

* 使用详解

1. target\_name: 自定义目标的名称。
2. ALL: 可选参数，如果指定，这个目标将会在调用 make all 时被构建（默认情况下，只有可执行文件和库目标会在 make all 时被构建）。
3. COMMAND: 要执行的命令。
4. args: 传递给命令的参数。

* 示例

run\_tests 是自定义目标的名称。当使用 **cmake --build . --target run\_tests** 命令时，CMake 将会执行两个命令：首先打印 "Running tests..."，然后运行名为 TestExecutable 的程序（假设它是之前添加的一个可执行文件目标）：

add\_custom\_target( run\_tests

COMMAND ${CMAKE\_COMMAND} -E echo "Running tests..."

COMMAND TestExecutable )；

**添加依赖**

run\_tests 目标依赖于 TestExecutable 可执行文件目标。这意味着在运行测试之前，TestExecutable 将会被构建

add\_executable( TestExecutable test1.cpp test2.cpp) add\_custom\_target( run\_tests ALL

DEPENDS TestExecutable

COMMAND ${CMAKE\_COMMAND} -E echo "Running tests..."

COMMAND TestExecutable )

**清理自定义目标**

cleanup 目标将会删除一些常见的构建文件和目录，帮助你清理构建环境

add\_custom\_target( cleanup

COMMAND ${CMAKE\_COMMAND} -E remove -f \*~

COMMAND ${CMAKE\_COMMAND} -E remove -f CMakeCache.txt

COMMAND ${CMAKE\_COMMAND} -E remove -f CmakeFiles

COMMAND ${CMAKE\_COMMAND} -E remove -f Makefile

COMMAND ${CMAKE\_COMMAND} -E remove -f cmake\_install.cmake )

## CMake之find\_library

* 命令

find\_library( log-lib log )

* 使用详解

find\_library(var name 预编译库名) 查找到指定的预编译库，并将它的路径存储在变量中；默认的搜索路径为cmake包含的系统库，因此如果是NDK的公共库只需要指定库的name即可；

类似的还有：find\_file()/find\_path()/find\_program()/find\_package() ；

## CMake之option

* 命令

option( < option\_name > “<help\_text>” [default\_value] )

* 使用详解

定于一个可选参数，用于构建过程中的一些决策；

* + - 1. option\_name : 选项名称 ；
      2. help\_text : 选项含义 ；
      3. default\_value: 定义选项默认状态 ，一般是OFF或者ON ，默认值为OFF ；

除去ON外，其他所有值都认为是OFF；

## CMake之cmake\_dependent\_option

CMakeDependentOption是CMake内部的一个模块，该模块提供了一种机制来根据其他选项的状态启用或禁用 CMake 选项。这个模块允许开发者创建有条件的选项，这些选项的可用性取决于其他选项是否被用户设置。

CMakeDependentOption 模块定义了一个宏 cmake\_dependent\_option，它用于设置一个选项，并且可以指定该选项依赖于其他选项的状态。如果依赖的选项没有被启用，那么当前的选项将会自动被禁用，并且从用户的配置中隐藏。

* 命令

cmake\_dependent\_option( <option\_name>

"<help\_text>"

<default\_value>

<depends>

<force\_value> )

* 使用详解

根据一些选项而额外设置其他选项值；

option\_name : 选项名称。

help\_text : 选项含义。

default\_value : 选项的默认值。

depends :

* 1. 分号分割的列表全部为True时，<option\_name>为default\_value，。否则，<option\_name>的将设置为<force\_value>;
  2. 当分号分割的列表全部为True时，使用给定的<help\_text>和初始<value>;否则，用户设置任何值都会被保留，以备将来满足<depends>时使用；
* 示例

cmake\_dependent\_option ( ENABLE\_SAMPLE\_TESTING "Enable sample test programs" ON

"ENABLE\_SAMPLE\_BUILD;NOT DISABLE\_TESTING" OFF)

在此示例中，如果 ENABLE\_SAMPLE\_BUILD 为 ON，且 DISABLE\_TESTING 为 OFF， 则自动设置 ENABLE\_SAMPLE\_TESTING 为 ON，且该值可以通过cmake传入来覆盖，否则 ENABLE\_SAMPLE\_TESTING 会被强制设置为 OFF 且无法通过cmake传入覆盖。

## CMake之enable\_language

* 命令

enable\_language( languageName [OPTIONAL] )

* 使用详解

打开了CMake对参数中指定的语言的支持；

语言：

CXX

C

Fortran 等；

如果指定了OPTIONAL选项，用CMAKE\_<languageName>\_COMPILER\_WORKS变量来判断该语言是否被成功支持；

## configure\_file

* 命令

configure\_file( <input\_file> <output\_file>

[ NO\_SOURCE\_PERMISSIONS | USE\_SOURCE\_PERMISSIONS |

FILE\_PERMISSIONS <permissions> ...]

[ COPYONLY ] [ ESCAPE\_QUOTES ] [ @ONLY ]

[ NEWLINE\_STYLE [ UNIX | DOS | WIN32 | LF | CRLF ] ] )

* 使用详解

将<input\_file>的内容复制到<output\_file>文件，并替换<input\_file>内容中引用为@VAR@或${VAR}的变量为变量的当前值，若未定义变量，则替换为空字符串。

可选项：

COPYONLY ：仅拷贝<input\_file>文件里面的内容到<output\_file>文件，不进行替换

ESCAPE\_QUOTES ：使用反斜杠(C语言风格)来进行转义；

@ONLY ：限制替换，仅仅替换@VAR@变量，不替换${VAR}变量；

NEWLINE\_STYLE ：指定输入文件的新行格式，例如：Unix使用\n , Windows使用的是\r\n;

注：COPYONLY 与 NEWLINE\_STYLE 冲突，不能同时使用；

## 执行Cmake命令

* 命令

cmake -B <build\_directory> -S .

* 使用详解

1. -B <build\_directory>：指定构建目录（例如 build），这是 CMake 生成的中间文件和最终构建产物存放的地方。
2. -S .：指定源目录（当前目录 .），即包含 CMakeLists.txt 的目录。

如果你使用的是 Windows 并且已经安装了 Visual Studio，CMake 也可以集成到 Visual Studio 中，你可以直接从 Visual Studio 生成项目。

# Cmake之file

读：

file( READ <filename> <out-var> […] )

file( STRINGS <filename> <out-var> […] )

file( <HASH> <filename> <out-var> )

file( TIMESTAMP <filename> <out-var> […] )

file( GET\_RUNTIME\_DEPENDENCIES […] )

写：

file({ WRITE | APPEND } <filename> <content>... )

file({ TOUCH | TOUCH\_NOCREATE } [<file>...] )

file( GENERATE OUTPUT <output-file> [...] )

file( CONFIGURE OUTPUT <output-file> CONTENT <content> [...] )

文件系统:

file({GLOB | GLOB\_RECURSE} <out-var> [...] [<globbing-expr>...]) 指定目录搜索符合<globbing-expr>的文件

file(MAKE\_DIRECTORY [<dir>...]) : 创建目录

file({REMOVE | REMOVE\_RECURSE } [<files>...])

file(RENAME <oldname> <newname> [...])

file(COPY\_FILE <oldname> <newname> [...])

file({COPY | INSTALL} <file>... DESTINATION <dir> [...])

file(SIZE <filename> <out-var>)

file(READ\_SYMLINK <linkname> <out-var>)

file(CREATE\_LINK <original> <linkname> [...])

file(CHMOD <files>... <directories>... PERMISSIONS <permissions>... [...])

file(CHMOD\_RECURSE <files>... <directories>... PERMISSIONS <permissions>... [...])

路径转换:

file(REAL\_PATH <path> <out-var> [BASE\_DIRECTORY <dir>] [EXPAND\_TILDE])

file(RELATIVE\_PATH <out-var> <directory> <file>)

file({TO\_CMAKE\_PATH | TO\_NATIVE\_PATH} <path> <out-var>)

传输：

file(DOWNLOAD <url> [<file>] [...]) file(UPLOAD <file> <url> [...])

锁定：

file(LOCK <path> [...])

归档：

file( ARCHIVE\_CREATE OUTPUT <archive> PATHS <paths>... [...] )

file( ARCHIVE\_EXTRACT INPUT <archive> [...] )

## 读

### READ

file( READ <filename> <variable> [ OFFSET <offset> ] [ LIMIT <max-in> ] [ HEX ] )

* 描述

读取文件（filename）的内容保存到变量（variable）中；

* 参数

[]内的参数代表可选，从offset作为开始位置，最多读取max-in个字节的数据；

HEX：将数据转为十六进制标识（对二进制数据很有用）；

如果指定了HEX选项，输出的字母(a到f)都是小写。

* 示例

假如有config.cmake文件，内容如下：

set(TEST, "this is test")

则file的READ用法：

file( READ config.cmake contents )

string(REGEX REPLACE "set\\(([A-Za-z\_0-9]+)\\) \"([^\"]\*)\"" "\\1 \\2" contents "${contents}")

message("TEST: ${TEST}")

### STRINGS

file( STRINGS <filename> <variable> [<options>...] )

* 描述

解析<filename>中的ASCII字符串列表，并将其存储在<variable>中。忽略文件中的二进制数据。忽略回车(\r, CR)字符（遇到换行符才重新开始读行）；

* 参数

<filename> ： 文件名  
<variable> ： 变量名

<options> ：可选项如下

1. LENGTH\_MAXIMUM <max-len> ：只考虑长度不超过给定值（max-len）的字符串
2. LENGTH\_MINIMUM <min-len> ：只考虑不少于给定长度的字符串
3. LIMIT\_COUNT <max-num> ：限制要提取的不同字符串的数量
4. LIMIT\_INPUT <max-in> ：限制从文件中读取的输入字节数
5. LIMIT\_OUTPUT <max-out> ：限制（变量）中存储的总字节数。
6. NEWLINE\_CONSUME ：将换行符( \n , LF )视为字符串内容的一部分，而不是以换行符结束。
7. NO\_HEX\_CONVERSION ：除非提供此选项，否则英特尔十六进制和摩托罗拉S-record文件在读取时将自动转换为二进制。
8. REGEX <regex> ：只考虑匹配给定正则表达式的字符串
9. ENCODING <encoding-type> ：3.1版本新功能，考虑给定编码的字符串。目前支持的编码有:UTF-8、UTF-16LE、UTF-16BE、UTF-32LE、UTF-32BE。如果没有提供编码选项，并且文件有字节顺序标记，则编码选项将默认遵守字节顺序标记。

### 其他用到在补充

# Cmake之string

搜索和替换

string( FIND <string> <substring> <out-var> [...] )

string( REPLACE <match-string> <replace-string> <out-var> <input>... )

string( REGEX MATCH <match-regex> <out-var> <input>... )

string( REGEX MATCHALL <match-regex> <out-var> <input>... )

string( REGEX REPLACE <match-regex> <replace-expr> <out-var> <input>... )

操作

string( APPEND <string-var> [<input>...] )

string( PREPEND <string-var> [<input>...] )

string( CONCAT <out-var> [<input>...] )

string( JOIN <glue> <out-var> [<input>...] )

string( TOLOWER <string> <out-var> ) //转换字符串为小写

string( TOUPPER <string> <out-var> ) //转换字符串为大写

string( LENGTH <string> <out-var> )

string( SUBSTRING <string> <begin> <length> <out-var> )

string( STRIP <string> <out-var> )

string( GENEX\_STRIP <string> <out-var> )

string( REPEAT <string> <count> <out-var> )

比较

string( COMPARE <op> <string1> <string2> <out-var> )

哈希

string( <HASH> <out-var> <input> )

生成

string( ASCII <number>... <out-var> )

string( HEX <string> <out-var> )

string( CONFIGURE <string> <out-var> [...] )

string( MAKE\_C\_IDENTIFIER <string> <out-var> )

string( RANDOM [<option>...] <out-var> )

string( TIMESTAMP <out-var> [<format string>] [UTC] )

string( UUID <out-var> ... )

JSON

string( JSON <out-var> [ERROR\_VARIABLE <error-var>] {GET | TYPE | LENGTH | REMOVE} <json-string> <member|index> [<member|index> ...] )

string( JSON <out-var> [ERROR\_VARIABLE <error-var>] MEMBER <json-string> [<member|index> ...] <index> )

string( JSON <out-var> [ERROR\_VARIABLE <error-var>] SET <json-string> <member|index> [<member|index> ...] <value> )

string( JSON <out-var> [ERROR\_VARIABLE <error-var>] EQUAL <json-string1> <json-string2> )

## 查找

* 命令

string( FIND <string> <substring> <output\_variable> [REVERSE] )

* 使用详解

在<string>中查找<substring>，返回值存放于<output\_variable>,

存在：返回<string>中的下标 ;

未存在：返回 -1 ;

默认首次出现的匹配，若使用了“REVERSE”则为最后一次匹配；

注：下标从0开始，以字节为单位；因此遇到中文时，下标表示字符编码第一字节的位置；

## 替换

* 命令

string( REPLACE <match\_string> <replace\_string> <output\_variable> <input> [ <input>... ] )

* 使用详解

从所有<input> ...中查找<match\_string>并使用<replace\_string>替换，替换后的字符串存放于<output\_variable>。

多个输入时，先将所有输入连接后，再做查找替换。

## 正则表达式

### 查找

* 命令1

string( REGEX MATCH <regular\_expression> <output\_variable> <input> [ <input> ... ] )

* 使用详解

从所有<input> ...中查找<regular\_expression>匹配到的字符串，并存放于<output\_variable>，查不到输出为空字符串。多个输入时先连接再做操作。只匹配第一次。

* 命令2

string( REGEX MATCHALL <regular\_expression> <output\_variable> <input> [<input>...] )

* 使用详解

从所有<input> ...中查找<regular\_expression>匹配到的字符串，并存放于<output\_variable>，查不到输出为空字符串。多个输入时先连接再做操作。匹配所有符合的字符串，并连接起来输出。

### 替换

* 命令

string( REGEX REPLACE <regular\_expression> <replacement\_expression> <output\_variable> <input> [<input>...] )

* 使用详解

根据正则表达式查找，并替换。找不到则输出为输入字符串。

<replacement\_expression>可以使用\\1 \\2 ..,\\9来表示( )元字符匹配到的字符;这里使用两个\是因为 : “\”要先在字符串中转义，然后在正则匹配中进行转义。

\1 : 子表达式1

\2 : 子表达式2

……

\9 : 子表达式9

注意：子表达式以“()”分隔。

### 元字符

|  |  |
| --- | --- |
| 元字符 | 意义 |
| **^** | 匹配字符串开头 |
| $ | 匹配字符串结尾 |
| . | 匹配任意单个字符 |
| \<char> | 转义元字符。在字符串中要使用\\<char> |
| [] | 匹配括号内任何字符 |
| [^] | 匹配任何不在括号内的字符 |
| [a-e] | 表示范围，意义同[abcde] |
| \* | 匹配0次或多次 |
| + | 匹配1次或多次 |
| ? | 匹配0次或1次 |
| | | 或运算符 |
| () | 保存匹配的子表达式 |

## 字符串操作

尾部追加

头部追加

连接

大小写转换

长度

取子串

删除收尾空白字符

删除子串中的生成器表达式

复制字符串

比较

字符串加密

转换

数字/字符互转

变量替换

\_替换

UUID生成

# Cmake之list

读：

list(LENGTH <list><output variable>)

list(GET <list> <elementindex> [<element index> ...]<output variable>)

list(APPEND <list><element> [<element> ...])

list(FIND <list> <value><output variable>)

list(INSERT <list><element\_index> <element> [<element> ...])

list(REMOVE\_ITEM <list> <value>[<value> ...])

list(REMOVE\_AT <list><index> [<index> ...])

list(REMOVE\_DUPLICATES <list>)

list(REVERSE <list>)

list(SORT <list>)

list格式如下：

list (subcommand <list> [args...])

subcommand子命令：

LENGTH ：返回list的长度；

GET ：返回list中index的element到value中；

APPEND ：添加新的element到list中；

FIND ：返回list中element的index , 没有找到返回-1 ；

INSERT ：将新element 插入到list中index的位置；

REMOVE\_ITEM ：从list中删除某个element；

REMOVE\_AT ：从list中删除指定index的element；

REMOVE\_DUPLICATES ：从list中删除重复的element；

REVERSE ：将list的内容反转；

SORT ：将list按字母顺序排序；

## 列表创建

set (list\_test a b c d) # 创建列表变量"a;b;c;d"

## 列表读取

* 命令1

list( LENGTH <list> <output variable> )

* 使用详解

读取列表长度；

* 命令2

list ( GET <list> <element\_index> [<element\_index> ...] <output variable>)

* 使用详解

读取列表中指定索引的元素，可以指定多个索引；

<element\_index>：为列表元素的索引，从0开始编号；索引也可以为负数，-1：最后一个，-2：倒数第二个，以此类推；

<output variable>：为新创建的变量，存储指定索引元素的返回结果，也是一个列表；

## 列表查找

* 命令1

list (FIND <list> <value> <output variable>)

* 使用详解

如果列表<list>中存在<value>，那么返回<value>在列表中的索引，如果未找到则返回-1

# Cmake之条件语句

if( <condition> )

<commands>

elseif( <condition> )

<commands>

else()

<commands>

endif()

if语句括号内的condition的写法，有三种：

1. if( <constant> )

cmake 定义了一些变量 ;如：

* 1 , ON , YES , TRUE , Y ,非0数 表示true ;
* 0 ，OFF , NO , FALSE , N , IGNORE , NOTFOUND , 空字符串或以-NOTFOUND结尾的字符串，表示flase ;

1. if( <variable|string> )

if语句内可以是变量或者字符串，注：if语句内变量直接引用即可,不需要${};

1. 运用cmake的一些特殊命令

**逻辑操作**

if( NOT <condition> )

如果<condition>不为真，则NOT <condition>为真 ；

if( <cond1> AND <cond2> )

一假则假；

if( <cond1> OR <cond2> )

一真则真；

**存在性检查**

if( COMMAND command-name )

若给定名称是可以调用的命令、宏或函数，则为True ;

if( POLICY policy-id )

若给定的名称是一个已存在的策略（形式为CMP< NNNN >）,则为True ;

if( TARGET target-name ) ： target-name目标是否已存在，存在则为True

if( TEST test-name )

若给定名称是由add\_test()命令创建的现有测试名称，则为True ;

if( DEFINED <name> | CACHE {<NAME>} | ENV{ name } )

若定义了具有给定的变量、缓存变量或环境变量，则为True。变量的值无关紧要。请注意，宏参数不是变量;

if( <variable|string> IN\_LIST < variable> )

若给定元素包含在命名列表变量中，则为True；

**文件操作**

if( EXISTS path-to-file-or-dir )

* + - * 如果指定的文件或目录存在，则为True ;
      * 只对完整路径进行了定义，不能使用相对路径；
      * 解析符号链接，即如果指定的文件或目录是符号链接，如果符号链接的目标存在，则返回为True ;

if( file1 IS\_NEWER\_THAN file2 )

* + - * + 若file1比file2新，或者两个文件中有一个不存在，则为True ;
        + 行为只对完整路径进行定义;
        + 若文件时间戳完全相同，IS\_NEWER\_THAN比较将返回true;

if( IS\_DIRECTORY path-to-directory )

* + 若给定的名称是目录，则为True ;
  + 行为仅对完整路径定义良好 ;

if( IS\_SYMLINK file-name )

* + 若给定名称是符号链接，则为True ;
  + 行为仅对完整路径定义良好 ;

if( IS\_ABSOLUTE path )

* + 若给定路径是绝对路径，则为True ;
  + 空路径，则为false ;
  + 在Windows主机上，任何以驱动器号和冒号( C: )、正斜杠或反斜杠开头的路径都为True ;(C:NO\BASE\DIR 计算为true )
  + 在非Windows主机上，任何以波浪线（~）开头的路径，则为True

**比较**

if( <variable|string> MATCHES regex )

* 若给定的字符串或变量的值与给定的正则条件匹配，则为True。
* 关于正则表达式格式，请参见正则表达式规范。

if( <variable|string> LESS <variable|string> )

* 若给定字符串或变量的值是有效的数字，且小于右边的值，则为True。

if( <variable|string> GREATER <variable|string> )

* 若给定字符串或变量的值是有效的数字，且大于右边的值，则为True。

if( <variable|string> EQUAL <variable|string> )

* 若给定字符串或变量的值是有效的数字，且等于右边的值，则为True。

if( <variable|string> LESS\_EQUAL <variable|string> )

* 若给定字符串或变量的值是有效的数字，且小于等于右边的值，则为True。

if( <variable|string> GREATER\_EQUAL <variable|string> )

* 若给定字符串或变量的值是有效的数字，且大于等于右边的值，则为True。

if( <variable|string> STRLESS <variable|string> )

* 若给定字符串或变量的值按字典顺序小于右侧的字符串或变量，则为True。

if（ <variable|string> STREQUAL <variable|string> ）

* 若给定字符串或变量的值在词典中等于右侧的字符串或变量，则为True。

if（ <variable|string> STRLESS|EQUAL <variable|string> ）

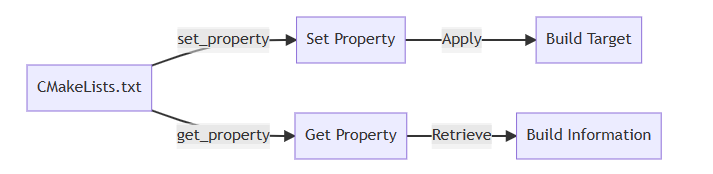
* 版本3.7中的新功能：若给定字符串或变量的值按字典顺序小于或等于右侧的字符串或变量，则为True。

if（<variable | string>STRGREATER|u EQUAL<variable | string>）

* 版本3.7中的新功能：若给定字符串或变量的值按字典顺序大于或等于右侧的字符串或变量，则为True。

# Cmake之属性管理

Cmake属性管理是构建系统的核心部分。通过使用set\_property和get\_property命令，开发者可以为目标、测试、源文件、目录及全局设置和获取属性。这些属性控制着构建过程和逻辑。



## CMake之set\_property

set\_property用于设置指定范围内的属性值；

* 命令

set\_property( <SCOPE> <NAME> <PROPERTY> <VALUE1> [VALUE2 ...] )

* 使用详解

1. <SCOPE>：定义属性的作用域。这可以是 GLOBAL（全局范围）、DIRECTORY（目录范围）、TARGET（目标范围）、SOURCE（源文件范围）、TEST（测试范围）或 CACHE（缓存条目范围）。
2. <NAME>：指定属性所属的实体名称。例如，目标的名称或源文件的路径。
3. <PROPERTY>：要设置的属性名称。
4. <VALUE1> [VALUE2 ...]：要设置的属性值。

## CMake之get\_property

get\_property 命令在CMake中用于获取已定义属性的值。这个命令可以用来查询各种类型的属性，包括目标属性、测试属性、源文件属性等；

* 命令

get\_property(<variable> <SCOPE> <NAME> PROPERTY <PROPERTY\_NAME>)

* 使用详解

1. <variable>：一个变量，用于存储获取到的属性值。
2. <SCOPE>：定义属性的作用域。这可以是 GLOBAL（全局范围）、DIRECTORY（目录范围）、TARGET（目标范围）、SOURCE（源文件范围）、TEST（测试范围）或 CACHE（缓存条目范围）。
3. <NAME>：指定属性所属的实体名称。例如，目标的名称或源文件的路径。
4. PROPERTY <PROPERTY\_NAME>：要获取的属性名称。
5. [SET]：可选参数，如果指定，只有当属性被显式设置时，命令才会成功。

# Cmake之find\_package

用于在CMake模块路径中查找并加载外部项目或库；该命令通常用于引入第三方库或框架；

* 命令

find\_package( <PackageName> [version] [EXACT] [QUIET] [MODULE]

   [REQUIRED] [[COMPONENTS] [components...]]

  [OPTIONAL\_COMPONENTS components...]

   [NO\_POLICY\_SCOPE] )

* 使用详解

几个重要的参数介绍：

* **PackageName**：待查找包的名称。此外它还决定两种搜索模下的.cmake文件名称：例如模块模式下的名称为Find<PackageName>.cmake，而配置模式下为<lowercasePackageName>-config.cmake/<lowercasePackageName>-config-version.cmake。
* **MODULE**：该选项指定find\_package命令只使用模块模式搜索方式查找。未指定该选项时，find\_package会优先使用模块模式搜索，仍未找到包时，会切换成配置模式搜索。
* **version**：待查找包的版本号要求，版本号为点分格式，由四个部分组成，每个部分都是一个数字，均为可选：major[.minor[.patch[.tweak]]]，例如1.1.1.1、1.0、等。同样也可以指定版本范围（CMake 3.19及之后才支持），格式为：versionMin...[<]versionMax，versionMin和versionMax均是major[.minor[.patch[.tweak]]]形式的版本号，默认情况下会包含这个指定区间两端的版本号，但如果指定了<，那么会排除掉versionMax，例如1.1.1.1...1.1.2.0、1.1.1.1...<1.1.2.0等。
* **EXACT**：该选项要求待查找包的版本必须与指定的版本精确匹配，因此如果指定的是一个版本范围，不能使用该参数。
* **QUIET**：禁止输出信息，正常情况当找到包时，CMake会打印一些信息，指定该选项时会禁止掉这些打印。例外是当同时指定QUIET时，如果找不到包，仍然会输出错误信息并终止执行过程。
* **REQUIRED**：当未找到满足条件的包（例如版本号不匹配，或指定组件未找到等），会终止CMake的执行过程，并输出一条错误信息。如果未指定该选项，即使未找到满足条件的包，CMake的执行过程也会继续。
* **COMPONENTS**：指定要查找的组件。通常一个包可能包含多个组件（可以理解为多个库，例如把C++的std看成一个包的概念，那么vector就是std下的其中一个组件），我们的工程可能会依赖包下的具体某个组件，因此可以通过这个选项来检测这些组件是否存在。**通常的约定是，该选项后的组件应该都找到时才认为包找到，否则认为未找到满足条件的包。这个约束会依赖包的.cmake来实现**，通过find\_package命令传入的COMPONENTS可以通过<PackName>\_FIND\_COMPONENTS这个变量来获得。

## Cmake支持哪些包

* 官方预定义的包

通常包含在…/share/cmake-3.17/Modules，也可以通过CMAKE\_MODULE\_PATH来指定；

* 非官方但支持Cmake的包

这一类包虽然不在cmake的预支持列表中，但自行下载使用cmake编译安装后也能被cmake识别，使用方法与官方预定义包一致，比如glog包

* 默认不支持cmake的包

最典型的是自定义编译的库文件，这类文件既不再cmake官方列表中，安装时默认也不会被cmake搜索到，此时就需要我们自行编写相应的配置文件（.cmake文件）并放到对应的路径中去

## Cmake如何查找包

* Cmake并不是直接去查找包本身，而是通过包对应的配置文件查找包。该文件说明了包一些相关信息，比如版本，目录等。在不同模式下，配置文件可能不同，但是只要目标包有对应的.cmake文件且存在于cmake的查找路径中，就可以被find\_package直接使用。
* 搜索路径顺序：
  + - 1. 从变量 CMAKE\_MODULE\_PATH 指定的路径中进行查找；
      2. 从Cmake安装路径中查找。Cmake会在其安装路径下提供很多.cmake文件，例如：/XXX/cmake/Modules目录下。
* 两种搜索模式：

模块模式（ Module mode ）

在该模式下，Cmake会搜索一个名为Find<PackageName>.cmake的文件，其中<PackageName>为待搜索包的名称；

若找到文件Find<PackageName>.cmake，Cmake会读取并处理该文件，简而言之，它负责检查一些条件（如版本号是否满足等）是否满足，并在找到包后，返回给调用者一些变量，用以获取包的详细信息。

一般来说，Find<PackageName>.cmake文件不是随包本身一起提供的，更多的是外部针对已有包的重新包装，例如操作系统、Cmake程序、甚至是调用find\_package命令的工程针对已有的包提供针对该包的.cmake文件。

配置模式（ Config Mode ）

该模式下，Cmake会搜索<lowercasePackageName>-config.cmake文件或<PackageName>Config.cmake文件。如果find\_package命令中指定了具体的版本，也会搜索<lowercasePackageName>-config-version.cmake或<PackageName>ConfigVersion.cmake文件，因此配置模式下通常会提供配置文件和版本文件，并且作为包的一部分一起提供给使用者；

## find\_package 的默认查找路径

1. **用户指定的路径**：通过 find\_package 命令的 PATHS、HINTS 或 NO\_DEFAULT\_PATH 选项指定的路径。
2. **CMAKE\_PREFIX\_PATH**：这是一个 CMake 缓存变量，包含一系列路径，find\_package 会在这些路径下搜索库的配置文件。这通常包括库的安装前缀。
3. **CMAKE\_FIND\_ROOT\_PATH**：这个变量用于指定根路径，find\_package 会在这个变量指定的目录下搜索库。这在交叉编译或需要控制搜索路径时特别有用。
4. **系统环境变量**：如 PATH、INCLUDE 等，find\_package 会搜索这些环境变量中包含的路径。
5. **标准系统路径**：这包括系统的标准库和头文件路径，如 /usr/local、/usr 等。
6. **CMake 自带的模块路径**：CMake 安装时自带的模块路径(/XXX/cmake/Modules)，用于查找 Find\*.cmake 文件。
7. **包配置文件**：如果库提供了 CMake 的配置文件（\*Config.cmake），find\_package 会使用这些文件来定位库。
8. **包版本文件**：如果库提供了版本文件（\*Version.cmake），find\_package 也会使用这些文件。
9. **使能包根路径**：如果设置了 CMAKE\_FIND\_USE\_PACKAGE\_ROOT\_PATH，则 find\_package 会在包的根路径下搜索。
10. **使能系统路径**：如果没有设置 NO\_CMAKE\_SYSTEM\_PATH，则 find\_package 会搜索系统路径。
11. **使能环境变量路径**：如果没有设置 NO\_CMAKE\_ENVIRONMENT\_PATH，则 find\_package 会搜索环境变量路径。
12. **使能 CMake 路径**：如果没有设置 NO\_CMAKE\_PATH，则 find\_package 会搜索 CMake 路径。

通过这些路径，find\_package 能够找到大多数常见的库和程序。如果默认路径不足以找到所需的库，可以通过设置上述变量或在 find\_package 命令中使用选项来指定额外的搜索路径。

# CMake之Export

* 命令

1. export( EXPORT < export-name > [NAMESPACE <namespace>] [FILE <filename>] )
2. export(TARGETS [ target1 [target2 [...] ] ] [ NAMESPACE <namespace> ] [APPEND] FILE <filename> [ EXPORT\_LINK\_INTERFACE\_LIBRARIES ] )
3. export(PACKAGE <packet\_name>) ：此export会记录在<packet\_name>Config.cmake的位置PACKAGE\_DIR;

* 使用详解

从生成树导出目标以供外部项目使用；

# CMake之模块

## Cmake之Ctest

CTest 是 CMake 提供的一个测试驱动程序，它与 CMake 紧密集成，允许开发者在 CMake 配置文件中添加测试指令，实现测试的自动化管理。CTest 可以用来执行多种类型的测试，包括单元测试、功能测试和性能测试等。它支持跨平台测试，并能够生成多种形式的测试报告。

流程：

* 1. 编译生成测试程序
  2. cmake中include(CTest)之后，使用命令enable\_test()，更改 BUILD\_TESTING变量 ,开启测试;
  3. cmake中使用add\_test添加测试用例;
  4. 设置测试属性；使用 set\_tests\_properties() 命令来设置测试的属性，例如检查测试输出是否包含特定的字符串。
  5. 编译成功后，运行测试：调用ctest或make test执行测试，测试结果会报告给测试系统

PS : 默认情况下，如果没有明确设置BUILD\_TESTING，CMake 会尝试查找测试文件，如果找到，它会将 BUILD\_TESTING 设置为 ON。

### CMake之enable\_testing

* 命令

enable\_testing()

* 使用详解

一定要在根目录下的CmakeLists.txt中开启，不然执行make test时会报错；

### CMake之set\_tests\_properties

* 命令

set\_tests\_properties( my\_test

PROPERTIES

PASS\_REGULAR\_EXPRESSION "Success")

* 使用详解

1. PROPERTIES: 设置测试的属性。
2. PASS\_REGULAR\_EXPRESSION: 测试通过的条件，如果测试输出匹配这个正则表达式。

### CMake之add\_test

用于添加测试用例 ，这个命令允许你定义测试，这些测试可以在构建过程中自动运行，以验证代码的正确性。

* 命令

add\_test( NAME <name>

[CONFIGURATIONS [Debug|Release|...]]

[WORKING\_DIRECTORY dir ]

COMMAND <command>

[arg1 [arg2 ...]] )

* 使用详解

NAME : 指定本测试的名称;

CONFIGURATIONS : 指定不同的编译版本下是否进行测试;

WORKING\_DIRECTORY : 指定工作路径

COMMAND : 指定可运行程序

args... : 传递给命令的任何参数

* 示例

如果你有一个名为 my\_test 的可执行文件，并且你想将它作为一个测试用例添加到 CTest 中，你可以这样写：

add\_test(NAME MyTest COMMAND my\_test)

MyTest 是测试的名称;

my\_test 是可执行文件的名称；

当你运行 ctest 命令时，MyTest 将作为测试用例执行；

## Cmake之CPack

### CPack简介

CPack 是 CMake 的一个组件，它提供了创建软件包的功能，包括二进制包和源代码包。CPack 支持多种不同的打包格式，使得软件可以在不同的操作系统和环境中轻松安装和部署。

### CPack关键特性

* 1. **支持的打包格式**

7Z 、DEB、RPM 、NSIS 、NSIS64 、NuGet 、ZIP

* 1. **配置Cpack**

|  |  |
| --- | --- |
| 变量名称 | 含义 |
| CPACK\_PACKAGE\_NAME | 指定包名；若没指定，默认用项目名称； |
| CPACK\_PACKAGE\_VENDOR | 包供应商名称； |
| CPACK\_PACKAGE\_DIRECTORY | 指定CPack打包的目录。   * 如果未设置，默认为构建目录。 * 这个变量可以在CPack配置文件中定义，也可以从CPack命令行选项“-B”中定义。如果设置命令行选项，会覆盖在配置文件中配置的值； |
| CPACK\_PACKAGE\_VERSION\_MAJOR | 包主版本 |
| CPACK\_PACKAGE\_VERSION\_MINOR | 包次版本 |
| CPACK\_PACKAGE\_VERSION\_PATCH | 包补丁版本 |
| CPACK\_PACKAGE\_DESCRIPTION\_FILE | 用于描述该项目的文本文件 |
| CPACK\_PACKAGE\_DESCRIPTION\_SUMMARY | 简短的描述该项目 |
| CPACK\_PACKAGE\_FILE\_NAME | 要生成的包文件的名称，不包括扩展名 |
| CPACK\_PACKAGE\_INSTALL\_DIRECTORY | 在目标系统上，安装该目录 |
| CPACK\_PACKAGE\_ICON | 包图标 |
| CPACK\_PACKAGE\_CHECKSUM |  |
| CPACK\_PROJECT\_CONFIG\_FILE |  |
| CPACK\_RESOURCE\_FILE\_LICENSE | 安装程序的许可证 |
| CPACK\_RESOURCE\_FILE\_README | 安装程序的自诉文件。  通常在安装过程中，详细描述项目的目的；并非所有Cpack生成器都使用此文件。 |
| CPACK\_RESOURCE\_FILE\_WELCOME |  |
| CPACK\_MONOLITHIC\_INSTALL |  |
| CPACK\_GENERATOR | 指定CPack生成的包类型（如：DEB , .7z , .zip等） |
| CPACK\_OUTPUT\_CONFIG\_FILE |  |
| CPACK\_PACKAGE\_EXECUTABLES |  |
| CPACK\_STRIP\_FILES |  |
| CPACK\_VERBATIM\_VARIABLES |  |
| CPACK\_SOURCE\_PACKAGE\_FILE\_NAME |  |
| CPACK\_SOURCE\_STRIP\_FILES |  |
| CPACK\_SOURCE\_GENERATOR | 用于源码包的生成器列表；与CPACK\_GENERATOR一样，如果未指定，则 CPack 将创建一组选项（例如，CPACK\_SOURCE\_ZIP），允许用户选择将生成哪些包。 |
| CPACK\_SOURCE\_OUTPUT\_CONFIG\_FILE |  |
| CPACK\_SOURCE\_IGNORE\_FILES |  |
| CPACK\_CMAKE\_GENERATOR |  |
| CPACK\_INSTALL\_CMAKE\_PROJECTS |  |
| CPACK\_SYSTEM\_NAME |  |
| CPACK\_PACKAGE\_VERSION |  |
| CPACK\_TOPLEVEL\_TAG |  |
| CPACK\_INSTALL\_COMMANDS |  |
| CPACK\_INSTALLED\_DIRECTORIES |  |
| CPACK\_PACKAGE\_INSTALL\_REGISTRY\_KEY |  |
| CPACK\_CREATE\_DESKTOP\_LINKS |  |
| CPACK\_BINARY\_<GENNAME> |  |

* 1. **打包源代码**

CPack 也可以用于创建源代码包。这通常用于分发项目的源代码，以便在其他系统上构建。你可以设置 CPACK\_SOURCE\_GENERATOR 来指定源代码包的生成器。

* 1. **高级配置**

CPack 提供了高级配置选项，允许你自定义安装脚本、预安装和后安装脚本、以及在打包过程中执行的其他操作。

* 1. **并行压缩**

CPack 支持并行压缩，这可以显著加快大型项目打包的时间。

* 1. **CPack 配置文件**

CPack 可以生成配置文件，如 CPackConfig.cmake 和 CPackSourceConfig.cmake，这些文件包含了打包过程中所需的配置信息；

* 1. **CPack 目标**

在某些 CMake 生成器中，如 Makefile 或 Ninja，CPack 可以创建 package 和 package\_source 目标，允许你通过构建系统来创建包。

### CPack使用方法

* 1. 在CmakeList.txt中，包含Cpack模块

include(CPack)

* 1. 设置CPack属性；（如：包名，包类型等）
  2. 指定需要打包的文件；（如：可执行文件，动态库，静态库）
  3. **生成包**：在配置好 CPack 后，你可以通过 CMake 生成相应的 CPackConfig.cmake 文件。然后，你可以使用 cpack 命令来生成包。例如，如果你想要生成一个 RPM 包，你可以在命令行中运行：

cpack -G RPM -

这将在当前构建目录下生成一个 RPM 包。

在配置好 CPack 后，你可以通过以下命令来生成安装包：

# 生成源代码压缩包

cpack -G TGZ -C CPackSourceConfig

# 生成 NSIS 安装器（Windows 平台）

cpack -G NSIS

这些命令会根据你的配置生成相应的安装包或压缩包。

### CPack详细说明

可以设置 CPack 相关的变量，并在构建过程中生成包。CPack 的详细配置和使用可以通过 CMake 的官方文档或者通过在命令行运行 cpack --help 来获取。

## Cmake之CPackDeb

内置（二进制）CPack Deb 生成器（仅限 Unix）

|  |  |
| --- | --- |
| 变量名称 | 含义 |
| CPACK\_DEBIAN\_PACKAGE\_NAME | 指定程序名，就是程序安装后的名字； |
| CPACK\_DEBIAN\_PACKAGE\_ARCHITECTURE | 指定运行的计算机架构 |
| CPACK\_DEBIAN\_COMPRESSION\_TYPE |  |
| CPACK\_DEBIAN\_PACKAGE\_DEPENDS | 指定所需的依赖 |
| CPACK\_DEBIAN\_PACKAGE\_MAINTAINER | 设置维护人 |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

## CMake之FetchContent

### FetchContent\_Declare

声明一个内容对象，该对象指向一个外部项目或资源，这些资源可以在构建过程中被下载和使用。

### FetchContent\_GetProperties

查询 FetchContent\_Declare 声明的外部内容对象的属性。可以帮助你检查外部依赖项的状态，例如它们是否已经被下载、是否已经构建，以及它们的源和二进制目录的位置。

### FetchContent\_MakeAvailable

# 确保内容已经被下载

FetchContent\_MakeAvailable(external\_project)

### FetchContent\_Populate

用于根据 FetchContent\_Declare 声明的内容对象下载和准备外部依赖项。当你声明了一个外部依赖项后，FetchContent\_Populate 命令会负责实际的下载（如果需要的话）、解压和配置过程，使得依赖项可以在你的 CMake 项目中使用。

## CMake之GNUInstallDirs

GNUInstallDirs 是 CMake 中的一个模块，它**提供了一组预定义的安装目录变量**，这些变量遵循 GNU 项目的文件系统层次结构标准（Filesystem Hierarchy Standard, FHS）。使用这些变量可以帮助你在安装软件时保持一致性，并且遵循广泛接受的 Unix/Linux 系统目录结构

包含该模块：

include( GNUInstallDirs )

一旦包含该模块，你就可以在 install 命令中使用它提供的变量。

### GNUInstallDirs 模块提供的变量

以下是 GNUInstallDirs 模块定义的一些常用变量：

* **BINDIR**: 用于安装用户可执行文件的目录（通常是 /usr/bin 或 /usr/local/bin）。
* **SBINDIR**: 用于安装系统管理员使用的可执行文件的目录（通常是 /usr/sbin 或 /usr/local/sbin）。
* **LIBEXECDIR**: 用于安装程序专用的辅助执行文件的目录（通常是 /usr/libexec 或 /usr/local/libexec）。
* **DATAROOTDIR**: 用于安装程序数据文件的根目录（通常是 /usr/share 或 /usr/local/share）。
* **DATADIR**: 用于安装程序特定数据文件的目录，通常是 DATAROOTDIR/your\_package。
* **SYSCONFDIR**: 用于安装系统配置文件的目录（通常是 /etc）。
* **LOCALSTATEDIR**: 用于安装程序运行时的本地状态数据的目录（通常是 /var）。
* **RUNSTATEDIR**: 用于安装程序运行时的状态数据的目录，通常是 LOCALSTATEDIR/run。
* **LIBDIR**: 用于安装程序库文件的目录（通常是 /usr/lib 或 /usr/local/lib）。
* **INCLUDEDIR**: 用于安装 C 头文件的目录（通常是 /usr/include 或 /usr/local/include）。
* **OLDINCLUDEDIR**: 用于安装旧的 C 头文件的目录，通常是 /usr/include。
* **DOCDIR**: 用于安装文档的目录，通常是 DATADIR/doc/your\_package。
* **INFODIR**: 用于安装 info 页面的目录，通常是 DATADIR/info。
* **MANDIR**: 用于安装手册页的目录（通常是 /usr/share/man 或 /usr/local/share/man）。

### CMake之install

用于指定在构建过程中应该安装的目标和文件。这个命令允许开发者定义安装规则，以便将可执行文件、库、头文件和其他资源**复制**到指定的安装目录。这样，当用户安装你的软件时，所有必要的组件都会被放置在正确的位置。

install命令为工程生成安装规则，同一个源文件目录下的安装规则按照install()命令的调用顺序在安装时（也就是使用make install）执行；

执行安装：通常是 make install 或 cmake --build . --target install 命令

目前6中安装方式：

install(TARGETS <target>... [...])

install({FILES | PROGRAMS} <file>... [...])

install(DIRECTORY <dir>... [...])

install(SCRIPT <file> [...])

install(CODE <code> [...])

install(EXPORT <export-name> [...])

#### install Target

* 命令

install( TARGETS targets...

[EXPORT <export-name>]

[RUNTIME\_DEPENDENCIES args...| RUNTIME\_DEPENDENCY\_SET <set-name>]

[

[ARCHIVE|LIBRARY|RUNTIME|OBJECTS|FRAMEWORK|BUNDLE|

PRIVATE\_HEADER|PUBLIC\_HEADER|RESOURCE|FILE\_SET <set-name>]

[DESTINATION <dir>]

[PERMISSIONS permissions...]

[CONFIGURATIONS [Debug|Release|...]]

[COMPONENT <component>]

[NAMELINK\_COMPONENT <component>]

[OPTIONAL] [EXCLUDE\_FROM\_ALL]

[NAMELINK\_ONLY|NAMELINK\_SKIP]

]

[...]

[INCLUDES DESTINATION [<dir> ...]]

)

* 使用详解

参数中的TARGET可以是很多种目标文件，最常见的是**通过ADD\_EXECUTABLE或者ADD\_LIBRARY定义的目标文件，即可执行二进制、动态库、静态库**

ARCHIVE ： 静态库（除了macOS）、dll导入库（也就是dll对应的lib文件）

安装目录（**GNUInstallDirs**）变量 ： ${CMAKE\_INSTALL\_LIBDIR}

默认安装文件夹：lib

LIBRARY ： 共享库（lib文件 ，不包括dll文件）

安装目录变量 ： ${CMAKE\_INSTALL\_LIBDIR}

默认安装文件夹 ：lib

RUNTIME ： 可执行二进制文件；

安装目录变量 ： ${CMAKE\_INSTALL\_BINDIR}

默认安装文件夹 ：bin

OBJECTS ： 关联对象库的对象文件；参考add\_library中的object library；

FRAMEWORK、BUNDLE、PRIVATE\_HEADER、PUBLIC\_HEADER、RESOURCE ：这些参数都是与apple平台相关的内容；

DESTINATION ：指定安装目录；

PERMISSIONS：指定安装文件的权限。有效权限是OWNER\_READ，OWNER\_WRITE，OWNER\_EXECUTE，GROUP\_READ，GROUP\_WRITE，GROUP\_EXECUTE，WORLD\_READ，WORLD\_WRITE，WORLD\_EXECUTE，SETUID和SETGID；

CONFIGURATIONS：指定安装规则适用的构建配置列表(DEBUG或RELEASE等)

EXCLUDE\_FROM\_ALL：指定该文件从完整安装中排除，仅作为特定于组件的安装的一部分进行安装；

OPTIONAL：如果要安装的文件不存在，则指定不是错误；

NAMELINK\_ONLY、NAMELINK\_SKIP、NAMELINK\_COMPONENT：component相关；

EXPORT：将库名导出，与install(EXPORT)共同作用，将target导出；

INCLUDES DESTINATION：在执行install(EXPORT)命令进行导出时，将指定目录列表添加到target的属性[INTERFACE\_INCLUDE\_DIRECTORIES](https://cmake.org/cmake/help/v3.20/prop_tgt/INTERFACE_INCLUDE_DIRECTORIES.html#prop_tgt:INTERFACE_INCLUDE_DIRECTORIES)中；

注意：

* + 1. 对于install(targets)命令而言，一条install可以安装多个target，而一个target也有可能被install多次（都生效，而不是覆盖）
    2. CONFIGURATIONS参数，此选项指定的值仅适用于此选项之后列出的选项：例如，要为调试和发布配置设置单独的安装路径，请执行以下操作：

install(TARGETS target

CONFIGURATIONS Debug

RUNTIME DESTINATION Debug/bin)

install(TARGETS target

CONFIGURATIONS Release

RUNTIME DESTINATION Release/bin)

也就是说，DEBUG和RELEASE版本的DESTINATION安装路径不同，那么DESTINATION必须在CONFIGUATIONS后面。

3）DESTINATION参数：对于常规的可执行文件、静态库文件、共享库文件，DESTINATION安装选项不是必须的，因为在未提供DESTINATION选项时，会从变量GNUInstallDirs获取一个默认值，如果该变量也未定义，会设置为内置的默认值。对于PUBLIC\_HEADER和PRIVATE\_HEADER指定与安装目标关联的公共头文件、私有头文件、文件集合，也是一样的规则。一般来说，模块库、Apple捆绑包、框架需要提供安装路径。接口和对象库可以不提供安装路径，但是他们的处理会有差别。

* 示例

INSTALL(TARGETS myrun mylib mystaticlib

RUNTIME DESTINATION ${CMAKE\_INSTALL\_BINDIR}

LIBRARY DESTINATION ${CMAKE\_INSTALL\_LIBDIR}

ARCHIVE DESTINATION ${CMAKE\_INSTALL\_LIBDIR}

)

可执行二进制myrun安装到${CMAKE\_INSTALL\_BINDIR}目录，动态库libmylib.so安装到${CMAKE\_INSTALL\_LIBDIR}目录，静态库libmystaticlib.a安装到${CMAKE\_INSTALL\_LIBDIR}目录。

#### install Files

* 命令

install ( < FILES | PROGRAMS > files...

TYPE <type> | DESTINATION < dir>

[ PERMISSIONS permissions ... ]

[ CONFIGURATIONS [Debug|Release|...] ]

[ COMPONENT <component> ]

[ RENAME <name>] [ OPTIONAL ] [ EXCLUDE\_FROM\_ALL ]

)

* 使用详解

拷贝文件至指定目录；

FILES | PROGRAMS ：两者都是安装文件，

使用FILES ：

FILES安装后的文件默认权限为OWNER\_WRITE , OWNER\_READ , GROUP\_READ和WORLD\_READ (如果设置了PERMISSIONS则按PERMISSIONS来)；

使用PROGRAMS ：

除了Files的权限之外，还多了OWNER\_EXECUTE, GROUP\_EXECUTE和WORLD\_EXECUTE三个权限。

安装的文件内容，可以是生成器表达式 $< . . . >,但是以任何表达式开始的文件都要求是完整路径(实际上不是表达式也需要全路径)

TYPE | DESTINATION : 两者用于指定文件安装的目标目录，两者不能同时存在。

使用Type ：

使用TYPE的话则会指定文件的类型，并从GNUInstallDirs中获取相应的变量，而如果没有定义此变量则使用内置的默认值。下表则是 文件类型对应的变量 和 默认路径。

|  |  |  |
| --- | --- | --- |
| **TYPE Argument** | **GNUInstallDirs Variable** | **Built-In Default** |
| BIN | ${CMAKE\_INSTALL\_BINDIR} | bin |
| SBIN | ${CMAKE\_INSTALL\_SBINDIR} | sbin |
| LIB | ${CMAKE\_INSTALL\_LIBDIR} | lib |
| INCLUDE | ${CMAKE\_INSTALL\_INCLUDEDIR} | include |
| SYSCONF | ${CMAKE\_INSTALL\_SYSCONFDIR} | etc |
| SHAREDSTATE | ${CMAKE\_INSTALL\_SHARESTATEDIR} | com |
| LOCALSTATE | ${CMAKE\_INSTALL\_LOCALSTATEDIR} | var |
| RUNSTATE | ${CMAKE\_INSTALL\_RUNSTATEDIR} | <LOCALSTATEdir>/run |
| DATA | ${CMAKE\_INSTALL\_DATADIR} | <DATAROOTdir> |
| INFO | ${CMAKE\_INSTALL\_INFODIR} | <DATAROOTdir>/info |
| LOCALE | ${CMAKE\_INSTALL\_LOCALEDIR} | <DATAROOTdir>/locale |
| MAN | ${CMAKE\_INSTALL\_MANDIR} | <DATAROOTdir>/man |
| DOC | ${CMAKE\_INSTALL\_DOCDIR} | <DATAROOTdir>/doc |

使用 DESTINATION ：

则使用DESTINATION参数提供的目录。

注意：

1. 一些类型的内建默认值会使用DATAROOT作为前缀，DATAROOT使用CMAKE\_INSTALL\_DATAROOTDIR变量（其内建默认值是share），不能直接使用DATAROOT作为TYPE参数，需要使用DATA。

* 示例

#### install Directory

* 命令

install( DIRECTORY dirs...

TYPE <type> | DESTINATION <dir>

[FILE\_PERMISSIONS permissions...]

[DIRECTORY\_PERMISSIONS permissions...]

[USE\_SOURCE\_PERMISSIONS] [OPTIONAL] [MESSAGE\_NEVER]

[CONFIGURATIONS [Debug|Release|...]]

[COMPONENT <component>] [EXCLUDE\_FROM\_ALL]

[ FILES\_MATCHING ] [ [PATTERN <pattern> | REGEX <regex>]

[EXCLUDE] [PERMISSIONS permissions...] ] [...]

)

* 使用详解

用于安装一个或多个目录到目标目录，目录结构被完整拷贝。每个目录的最后一项会被添加到目标目录，如果不想这样则可以 以 “ / ” 结尾，表示最后一项为空。

**权限控制**

FILE\_PERMISSIONS ：文件的权限

DIRECTORY\_PERMISSIONS ：目标目录的权限

USE\_SOURCE\_PERMISSIONS ：若使用了USE\_SOURCE\_PERMISSIONS ,而FILE\_PERMISSIONS没有指定，则文件权限与源文件权限一样；

如果没有指定FILE\_PERMISSIONS权限，则文件的权限与FILES命令的权限一样（OWNER\_WRITE, OWNER\_READ, GROUP\_READ和WORLD\_READ），而目录权限与PROGRAMS一样（OWNER\_WRITE, OWNER\_READ, GROUP\_READ，WORLD\_READ，OWNER\_EXECUTE, GROUP\_EXECUTE和WORLD\_EXECUTE）

**过滤条件控制**

目录的安装可通过 PATTERN 和 REGEX 选项细粒度的控制 ；

其中PATTERN匹配完整文件名，REGEX默认匹配文件名中的任何一部分(可以通过/或者$实现与PATTERN相同的效果)

默认情况下，所有文件和目录都会被install(不论PATTERN和REGEX的匹配结果),但是我们可以通过设置FILES\_MATCHING过滤没有匹配的文件。

* 示例1

#表示将build目录下的内容安装到test目录下

install(DIRECTORY ${PROJECT\_SOURCE\_DIR}/build/

DESTINATION ${PROJECT\_SOURCE\_DIR}/test)

#表示将build目录的内容安装到test目录下

install(DIRECTORY ${PROJECT\_SOURCE\_DIR}/build

DESTINATION ${PROJECT\_SOURCE\_DIR}/test)

* 示例2

#目录全部拷贝，安装以target开始的文件

install(DIRECTORY

${PROJECT\_SOURCE\_DIR}/build/

DESTINATION ${PROJECT\_SOURCE\_DIR}/test

FILES\_MATCHING PATTERN "target\*" )

#目录全部拷贝，只安装target文件

install(DIRECTORY

${PROJECT\_SOURCE\_DIR}/build/ DESTINATION

${PROJECT\_SOURCE\_DIR}/test

FILES\_MATCHING PATTERN "target")

#目录全部拷贝，安装文件名包含target的文件

install(DIRECTORY

${PROJECT\_SOURCE\_DIR}/build/ DESTINATION

${PROJECT\_SOURCE\_DIR}/test

FILES\_MATCHING REGEX "target")

#REGEX通过前缀$或后缀/达到与PATTERN相同的效果

install(DIRECTORY

${PROJECT\_SOURCE\_DIR}/build/ DESTINATION

${PROJECT\_SOURCE\_DIR}/test

FILES\_MATCHING REGEX "target/" )

install(DIRECTORY

${PROJECT\_SOURCE\_DIR}/build/ DESTINATION

${PROJECT\_SOURCE\_DIR}/test

FILES\_MATCHING REGEX "$target")

#### install Script / Code

* 命令

install(

[ [SCRIPT <file>] [CODE <code>] ]

[ ALL\_COMPONENTS | COMPONENT <component> ]

[EXCLUDE\_FROM\_ALL] [...]

)

* 使用详解

1. SCRIPT: 会在安装期间调用指定的CMake脚本，如果脚本是相对路径，那么会解析成相对于当前源文件目录的相对路径；
2. CODE: 会在安装期间调用指定的CMake代码，代码是通过双引号包含的一串字符。例如install(CODE "MESSAGE(\"Sample install message.\")")，会在安装期间打印一跳消息；
3. ALL\_COMPONENTS: 定制的安装脚本代码会在每个组成部分安装的时候执行，该选项与COMPONENT互斥；
4. <file> 和 <code> 可以使用生成器表达式(对于<file>，此时引用的是文件名字而不是文件内容)

* 示例

#### install Export

* 命令

install(

EXPORT <export-name> DESTINATION <dir>

[NAMESPACE <namespace>] [ [FILE <name>.cmake] |

[PERMISSIONS permissions...]

[CONFIGURATIONS [Debug|Release|...]]

[EXPORT\_LINK\_INTERFACE\_LIBRARIES]

[COMPONENT <component>]

[EXCLUDE\_FROM\_ALL]

)

* 使用详解

1. **EXPORT ：** 生成并安装一个CMake文件，该CMake文件包含了将目标从安装目录中导出信息到其他项目的代码。安装的内容是跟上面install(TARGETS)中EXPORT的选项一样。<export-name>关联的是使用install(TARGETS)命令中使用EXPORT选项指定的名称。
2. **FILE：**生成的CMake文件默认名为<export-name>.cmake，可以通过FILE指定其他名称，但必须是以.cmake为后缀的名称。
3. **NAMESPACE：**该选项会在写入导入文件中的信息前添加名称<namespace>。
4. **CONFIGURATIONS：**指定该选项，仅当安装了其中一个命名配置时，才会安装导入文件，并且导入文件只会引用匹配的目标配置。
5. **EXPORT\_LINK\_INTERFACE\_LIBRARIES：**匹配(IMPORTED\_) ? LINK\_INTERFACE\_LIBRARIES(\_<CONFIG>) ? 属性的内容会被导出。
6. **COMPONENT：**指定该选项，其后列出的<component>会隐式依赖于导出集合中提到的所有组件。为了依赖的工程被合适的构建，导出的<name>.cmake文件要求每个导出的组件顺序呈现。例如，一个工程可能定义Runtime和Development两个组件，其中Runtime中包含动态库文件，Development中包含静态库和头文件。导出集合通常也是Development的一部分，但是它会从Runtime和Development两个组件中导出目标。因此Development组件安装时，Runtime组件也需要被安装，反过来不成立。如果安装Development组件时未安装Runtime，尝试链接的工程会出现构建错误。像APT、RPM等包管理器，通常会将Runtime组件作为Development的依赖，在包的元数据中列出来，确保如果头文件和CMake导出文件存在时库文件被安装。

# 常用变量

|  |  |
| --- | --- |
| 变量名称 | 含义 |
| PROJECT\_SOURCE\_DIR | 工程根目录 |
| PROJECT\_BINARY\_DIR | 运行cmake命令的目录，通常是$(PROJECT\_SOURCE\_DIR)/build |
| CMAKE\_INCLUDE\_PATH | 环境变量 |
| CMAKE\_LIBRARY\_PATH | 环境变量 |
| CMAKE\_CURRENT\_SOURCE\_DIR | 当前处理的CmakeList.txt所在路径 |
| CMAKE\_CURRENT\_BINARY\_DIR | Target编译目录   1. 使用ADD\_SURDIRECTORY(src bin) 可以更改此变量的值； 2. SET( EXECUTABLE\_OUTPUT\_PATH <新路径> )并不会对此变量有影响,只是改变了最终目标文件的存储路径； |
| CMAKE\_CURRENT\_LIST\_FILE | 输出调用这个变量的CmakeLists.txt的完整路径 |
| CMAKE\_CURRENT\_LIST\_LINE | 输出该变量所在行 |
| CMAKE\_MODULE\_PATH | 定义自己的cmake模块所在路径   1. SET(CMAKE\_MODULE\_PATH ${PROJECT\_SOURCE\_DIR}/cmake),然后可以用INCLUDE命令来调用自己的模块 |
| EXECUTABLE\_OUTPUT\_PATH | 重新定义目标二进制可执行文件的存放位置 |
| LIBRARY\_OUTPUT\_PATH | 重新定义目标链接库文件的存放位置 |
| PROJECT\_NAME | 通过PROJECT指令定义的项目名称 |
| CMAKE\_PROJECT\_NAME | 第一个被创建的project name |
| CMAKE\_ALLOW\_LOOSE\_LOOP\_CONSTRUCTS | 用来控制IF ELSE语句的书写方式 |
| CMAKE\_CXX\_STANDARD | 目标CXX\_STANDARD属性的默认值。  此变量用于初始化所有目标上的 CXX\_STANDARD 属性 |

系统信息

|  |  |
| --- | --- |
| 变量名称 | 含义 |
| CMAKE\_MAJOR\_VERSION | cmake主版本号，如2.8.6中的2 |
| CMAKE\_MINOR\_VERSION | cmake 次版本号，如2.8.6中的8 |
| CMAKE\_PATCH\_VERSION | cmake补丁等级,如2.8.6中的6 |
| CMAKE\_SYSTEM | 系统名称，如Linux-2.6.22 |
| CAMKE\_SYSTEM\_NAME | 不包含版本的系统名，如Linux |
| CMAKE\_SYSTEM\_VERSION | 系统版本，如2.6.22 |
| CMAKE\_SYSTEM\_PROCESSOR | 处理器名称，如i686 |
| UNIX 在所有的类UNIX平台为TRUE,包括OS X和cygwin | |
| WIN32 在所有的win32平台为TRUE,包括cygwin | |

开关选项

|  |  |
| --- | --- |
| 变量名称 | 含义 |
| BUILD\_SHARED\_LIBS | 控制默认的库编译方式；  如果未进行设置，使用ADD\_LIBRARY时又没指定库类型，默认编译生成的库都是静态库。 |
| CMAKE\_C\_FLAGS | 设置C编译选项 |
| CMAKE\_CXX\_FLAGS | 设置C++编译选项 |

ARGC 记录传入参数的个数；

ARGV : 所有传入参数的list ;

ARGV0 , ARGV1 : 顺序代表传入的参数；

ARGN : 所有传入参数的list ;