

CMake、Makefile 指南

作者：黄邦勇帅(原名：黄勇)

2023 年 10 月 20 日

前言

本文是对《C++语法详解》一书和《Qt 5.10 GUI 完全参考手册》的补充资料，以增进对C++编译程序底层的进一步理解。

本文对目前流行的 **makefie** 和 **CMake** 作了深入全面的讲解，**CMake** 是一款在全球范围内广受欢迎的跨平台的项目构建利器，已成为众多程序员的首选工具，被越来越多的组织接受和使用，如 **Qt**、**Visual Studio** 等。**CMake** 有着无法比拟的灵活性和易用性，使得项目的配置和构建变得清晰且易于维护，大大降低了复杂项目的混乱度，提升了开发效率。

本文章节划分合理，思路清楚，不显得杂乱无章，对每个章节的知识点讲解全面细致，各章节内容不相互重合累赘，方便复习查阅。

本文内容完全属于个人见解与参考文献的作者无关

限于水平有限，其中难免有误解之处，望指出更正，QQ: 42444472。

声明：禁止抄袭、复印、转载本文，本文作者拥有完全版权。

作者：黄勇



CMake、Makefile 指南总目录

第 1 篇 使用命令行工具..... 11

绪章 准备工作	11
一、工作环境及所需工具	11
二、基本配置	11
第 1 章 使用命令行工具构建程序	12
一、使用命令行编译 C++程序的步骤及使用的工具.....	12
二、静态库文件及其相关命令工具.....	13
三、使用命令工具构建可执行文件.....	14
四、g++命令的常用参数.....	15
五、cl 命令的常用参数.....	19
第 2 章 VC++动态库(DLL 文件).....	27
一、简单的创建一个 DLL 文件.....	27
二、link 命令简介.....	28
三、DLL 简介	29
四、创建和使用 DLL 文件的思维.....	30
五、VC++创建 DLL 文件的方法	32
六、查看修饰名称的方法	35
七、DEF 文件	36
八、导入库和导出文件(隐式链接).....	38
九、导入/导出类.....	43

第 2 篇 makefile..... 50

第 1 章 makefile 工具简介	50
第 2 章 makefile 基本原理	53
2.1 makefile 基础	53
2.1.1 为什么需要 makefile.....	53
2.1.2 makefile 基本语法结构.....	53
2.1.3 makefile 基本语法要求.....	54
2.1.4 makefile 执行规则的基本原则.....	54
2.2 makefile 目标	55
2.3 makefile 规则和 order-only 依赖	60
第 3 章 变量(也称为宏)、函数、条件语句.....	63
3.1 变量	63
3.2 define 指示符和 call、eval 函数(nmake 不适用)	67
3.2.1 使用 define 定义变量	67
3.2.2 使用 define 定义命令包	68
3.2.3 使用 call 函数向 define 中传递参数.....	69
3.2.4 将 define 与 eval 函数联合使用.....	70

3.3 自动变量、通配符和部分默认变量	71
3.3.1 自动变量	71
3.3.2 通配符	72
3.3.3 部分默认变量	72
3.4 变量和函数的展开过程(nmake 不适用).....	73
3.5 makefile 中的函数和条件语句	75
3.5.1 makefile 函数.....	75
3.5.2 makefile 条件语句.....	76
第 4 章 模式规则	79
4.1 静态模式规则(nmake 不适用).....	79
4.2 模式规则(nmake 不适用).....	80
4.2.1 模式规则匹配原理	80
4.2.2 模式规则的启动	82
4.2.3 搜索适用的模式规则及多目标模式模式规则.....	82
4.3 隐式模式规则(implicit pattern rules) (nmake 不适用)	84
4.4 中间文件、万能匹配规则、默认规则	86
4.4.1 中间文件(intermediate file).....	86
4.4.2 万能匹配规则(match-anything rule, 简称万能规则)	88
4.4.3 默认规则	89
4.5 隐式规则搜索算法	90
4.6 后缀规则(.SUFFIXES) (nmake 不适用)	92
第 5 章 make 读取 makefile 文件的方式	95
5.1 make 执行 makefile 文件的过程	95
5.1.1 常规读取 makefile 文件的方式.....	95
5.1.2 include 指示符.....	96
5.1.3 MAKEFILES 和 MAKEFILE_LIST 变量(nmake 不适用).....	97
5.1.4 重制 makefile (nmake 不适用)	98
5.1.5 make 执行过程总结.....	99
5.2 获取依赖	100
5.2.1 自动获取依赖(nmake 不适用)	100
5.2.2 目录搜索(nmake 不适用)	102
第 6 章 递归调用(nmake 不适用).....	107
6.1 make 工具的部分参数 (nmake 不适用).....	107
6.2 递归调用(nmake 不适用).....	108
6.2.1 递归调用与 MAKE 变量.....	109
6.2.2 传递变量给子 make (exprot、unexport 指示符).....	110
6.2.3 传递 make 命令参数给子 make(特殊变量 MAKEFLAGS).....	113
第 7 章 推理规则和内联文件(仅适用于 nmake).....	116
7.1 推理规则(仅适用于 nmake).....	116
7.1.1 基本语法及原理	116
7.1.2 预定义推理规则	118
7.2 内联文件(仅适用于 nmake).....	118
总结	121

第 3 篇 CMake	128
第 1 章 CMake 基础	128
1.1 CMake 基本设置	128
1.2 一个简单的示例	129
1.2.1、使用 CMake 构建一个 C++ 程序	129
1.2.2 分析 CMake 生成的文件	130
1.3 CMake 相关基本概念	132
1.3.1 CMake 文件	132
1.3.2 CMake 构建程序的过程	132
1.3.3 CMake 目录结构	133
1.3.4 cmake 命令行工具的使用	133
第 2 章 CMake 基本命令	136
2.1 CMake 基本语法	136
2.1.1 CMake 命令的参数	136
2.1.2 注释、变量、列表	138
2.1.3 生成器表达式\$<TARGET_OBJECTS:...>	139
2.2 cmake_policy()命令	140
2.3 cmake_minimum_required()命令	143
2.4 set()命令	143
2.4.1 常规变量	143
2.4.2 环境变量	144
2.4.3 缓存变量	145
2.5 function()、macro()、block()、return()命令	146
2.5.1 function()命令	146
2.5.2 macro()命令	147
2.5.3 block()命令	148
2.5.4 return()命令	148
2.6 if()命令(条件语句)	150
2.6.1 基本语法	150
2.6.2 基本的条件表达式	151
2.6.3 逻辑表达式	153
2.6.4 比较	153
2.6.5 存在性检测	154
2.6.6 文件或目录	154
2.7 foreach()命令	155
2.7.1 foreach 常规形式	155
2.7.2 foreach 变体 1	156
2.7.3 foreach 变体 2	156
2.7.4 foreach 变体 3	156
2.7.5 foreach 变体 4	157
2.8 while()命令	158
2.9 break()和 continue()命令	158
2.10 option()命令	159

2.11 unset()命令	159
第3章 CMake 属性	160
3.1 set_property()和 get_property()命令	160
3.1.1 set_property()命令	160
3.1.2 get_property()命令	162
3.2 define_property()命令	163
3.3 其他属性命令	166
3.3.1 set_target_properties()命令	166
3.3.2 set_directory_properties()命令	166
3.3.3 set_source_files_properties()命令	166
3.3.4 set_tests_properties()命令	167
3.3.5 get_cmake_property()命令	167
3.3.6 get_target_property()命令	167
3.3.7 get_directory_property()命令	167
3.3.8 get_source_file_property()命令	167
3.3.9 get_test_property()命令	168
第4章 CMake 项目	169
4.1 add_subdirectory()命令	169
4.1.1 CMake 项目的结构	169
4.1.2 add_subdirectory()命令	169
4.2 project()命令	174
4.2.1 project()命令	174
4.2.2 project()命令调用期间的执行步骤	178
第5章 使用 CMake 构建库文件和可执行文件	180
5.1 CMake 目标的分类	180
5.2 add_library()命令	181
5.2.1 add_library()命令基本语法	181
5.2.2 对象库目标(简称对象库)	185
5.2.3 接口库目标(简称接口库)	185
5.2.4 导入库目标(简称导入目标)	187
5.2.5 库目标的别名(别名库目标)	189
5.3 add_executable()命令	190
5.3.1 基本语法	190
5.3.2 导入可执行目标	190
5.3.3 可执行目标的别名	191
第6章 使用要求及编译参数	192
6.1 使用要求的基本概念	192
6.2 设置编译参数(COMPILE_OPTIONS 系列属性)	193
6.2.1 总览	193
6.2.2 target_compile_options()和 add_compile_options()命令	194
6.3 设置-D 编译参数(COMPILE_DEFINITIONS 系列属性)	200
6.3.1 总览	200
6.3.2 target_compile_definitions()和 add_compile_definitions()命令	201
6.4 设置编译特性(COMPILE_FEATURES 系统属性)	202

6.4.1 总览	202
6.4.2 target_compile_features()命令	203
6.4.3 <LANG>_STANDARD 目标属性以及与其相关的属性和变量	204
6.4.4 <LANG>_EXTENSIONS 目标属性以及与其相关的属性和变量	204
第 7 章 使用要求的传递	207
7.1 target_link_libraries()命令简介	207
7.2 传递使用要求的基本规则	209
7.2.1 使用要求传递的基本规则	209
7.2.2 本文的名称约定	212
7.3 链接依赖项(向链接命令添加文件).....	214
7.3.1 链接依赖项(向链接命令添加文件)概述	214
7.3.2 链接直接依赖项的规则	215
7.3.3 链接间接依赖项的规则	216
7.3.4 依赖项为生成器表达式的链接	218
7.3.5 链接依赖项总结	218
7.4 传递使用要求的详细规则	220
7.4.1 概述	220
7.4.2 传递使用要求的详细规则(INTERFACE_LINK_LIBRARIES 和 LINK_LIBRARIES 属性).....	220
7.5 对 target_link_libraries()命令进一步的讲解	223
7.5.1 target_link_libraries(PUBLIC)	223
7.5.2 target_link_libraries(PRIVATE)	224
7.5.3 target_link_libraries(INTERFACE)	227
7.6 INTERFACE_LINK_LIBRARIES_DIRECT 和 INTERFACE_LINK_LIBRARIES_DIRECT_EXCLUDE 目标属性	230
第 8 章 向 CMake 项目添加其他文件	233
8.1 添加源文件	233
8.1.1 添加源文件的属性和变量汇总	233
8.1.2 SOURCES 和 INTERFACE_SOURCE 目标属性	233
8.1.3 GENERATED 源文件属性	235
8.1.4 target_sources()命令--->常规语法	236
8.1.5 target_sources()命令--->文件集	237
8.1.6 aux_source_directory()命令	240
8.2 添加头文件包含目录(-I 编译参数)	240
8.2.1 总览	240
8.2.2 target_include_directories()命令	241
8.2.3 include_directories()命令	242
8.3 添加链接文件的库目录	245
8.3.1 总览	245
8.3.2 target_link_directories()命令	246
8.3.3 link_directories ()命令	247
8.3.4 link_libraries()命令	247
8.4 include()和 add_dependencies()命令	248
8.4.1 include()命令	248
8.4.2 add_dependencies()命令	249

第 9 章 使用 install()命令安装文件	250
9.1 使用变量或目标属性将程序文件输出到指定目录	250
9.1.1 输出工件(Output Artifacts)	250
9.1.2 将程序文件输出到指定目录的方法	251
9.1.3 使用 install()命令安装文件的方法	253
9.2 install(TARGETS) 安装目标文件	254
9.2.1 intstall()命令输出工件的类型	255
9.2.2 install()命令的默认安装路径	261
9.2.3 COMPONENT、EXCLUDE_FROM_ALL、OPTION 参数	264
9.2.4 PERMISSIONS、CONFIGURATIONS 参数	266
9.2.5 NAMELINK_ONLY、NAMELINK_SKIP、NAMELINK_COMPONENT 参数	266
9.2.6 INCLUDES DESTINATION [<dir> ...]参数	267
9.2.7 EXPORT<export-name>参数	268
9.2.8 RUNTIME_DEPENDENCIES、RUNTIME_DEPENDENCY_SET 参数	268
9.3 install(EXPORT) 安装导出	268
9.4 install(RUNTIME_DEPENDENCY_SET) 安装运行时依赖项	268
9.5 install(IMPORTED_RUNTIME_ARTIFACTS) 安装导入目标的运行时工件	271
9.6 install(<FILES PROGRAMS>) 安装文件	272
9.7 install(DIRECTORY) 安装目录	274
9.8 install(SCRIPT)和 install(CODE) 安装脚本和代码	276
第 10 章 导入目标和导出目标	278
10.1 导入目标	278
10.1.1 基础	278
10.1.2 设置导入目标的详细信息	278
10.2 导出目标	283
10.2.1 CMake 导出目标的思维	283
10.2.2 install(EXPORT)命令	285
10.2.3 export(TARGETS)和 export(EXPORT)命令	289
第 11 章 configure_file()命令(生成配置文件)	295
第 12 章 find_file()命令(查找文件)	298
12.1 find_file()命令的语法及基本参数	298
12.2 find_file()命令的搜索顺序	301
12.3 搜索指定根目录下的子目录	307
12.4 与搜索顺序有关的所有变量	309
12.4.1 开关变量	309
12.4.2 路径变量	310
12.5 其余的 find_*命令	312
第 13 章 find_package()命令(查找包)	315
13.1 前提基础知识	315
13.1.1 何为包?	315
13.1.2 CMake 包	315
13.1.3 怎样编写 CMake 包	315
13.1.4 CMake 包由谁提供	316
13.1.5 find_package()命令的作用	316

13.1.6 find_package()命令的格式及搜索模式.....	316
13.2 find_package()基本命令	318
13.3 find_package()完整命令	322
13.3.1 find_package()完整命令语法及基本参数.....	322
13.3.2 配置模式的搜索目录	324
13.3.3 配置模式的搜索前缀及搜索顺序	326
13.3.4 搜索指定根目录下的子目录	329
13.3.5 处理版本信息及版本文件变量	330
13.3.6 包(文件)接口变量.....	335
13.3.7 find_package()命令中各变量小结.....	337
13.4 使用 CMake 自带的模块 CMakePackageConfigHelpers 生成配置文件和版本文件.....	338
13.4.1 configure_package_config_file()命令	338
13.4.2 使用 write_basic_package_version_file()命令生成版本文件.....	344
第 14 章 使用 CMake 构建 Qt 程序	351
14.1 使用 CMake 构建一个简单的 Qt 程序	351
14.2 CMake 自动调用 uic.exe 工具生成头文件的过程及原理.....	353
14.2.1 Qt 构建工具	353
14.2.2 CMake 自动调用 uic.exe 工具生成头文件的过程及原理	354
14.3 CMake 自动调用 moc.exe 工具处理 Qt 的元对象系统	359
14.3.1 moc 简介	359
14.3.2 Qt 专有宏位于头文件中	359
14.3.3 Qt 专有宏位于源文件中	362
14.3.4 与 moc 有关的其他变量和属性	364
14.4 使用 Qt 提供的 CMake 命令构建 Qt 程序.....	365
14.4.1 qt_standard_project_setup()命令	366
14.4.2 qt_add_executable()命令	367
14.4.3 qt_add_library()命令	368
14.4.4 qt_finalize_target()命令	368
14.5 在 Qt 中使用 CMake 构建 Qt 程序	369
第 15 章 交叉编译(指定编译器等工具).....	375

第 1 部分 使用命令行工具目录

第 1 篇 使用命令行工具	11
绪章 准备工作	11
一、工作环境及所需工具	11
二、基本配置	11
第 1 章 使用命令行工具构建程序	12
一、使用命令行编译 C++ 程序的步骤及使用的工具.....	12
二、静态库文件及其相关命令工具.....	13
三、使用命令工具构建可执行文件.....	14
四、g++ 命令的常用参数.....	15
五、cl 命令的常用参数.....	19
第 2 章 VC++ 动态库(DLL 文件).....	27
一、简单的创建一个 DLL 文件.....	27
二、link 命令简介.....	28
三、DLL 简介	29
四、创建和使用 DLL 文件的思维.....	30
五、VC++ 创建 DLL 文件的方法	32
六、查看修饰名称的方法	35
七、DEF 文件	36
八、导入库和导出文件(隐式链接).....	38
九、导入/导出类.....	43

第 1 篇 使用命令行工具

绪章 准备工作

一、工作环境及所需工具

- 1、本系列文章讲解 windows 系统下 C++程序的构建情况，共分 3 部分，第 1 篇讲解基本命令工具的使用，第 2 篇讲解 Makefile 文件，第 3 篇讲解 CMake
- 2、本系列文章使用 Windows10 系统 64 位版本，需要安装以下软件或工具
 - Qt: 安装 Qt 的目的是使用其中的 mingw32-make、jom、g++等工具，当然，也可以分别单独安装这些工具。本文安装的 Qt 版本为 qt6.5.1，Qt Creator 的版本为 Qt Creator 10.0.1
 - Visual Studio 2022: 主要使用其中的 cl、link、lib、nmake 等工具。
 - CMake 3.27.0-rc4: 本系列文章未使用 Qt 自带的 CMake 2.24，因为版本低了一点。

二、基本配置

- 1、为了能在 windows 的 CMD 中更为方便的使用以上工具，需要配置环境变量，设置环境变量的方法为，右击“我的电脑”->“属性”，找到“高级系统设置”->“环境变量”。
- 2、配置 Qt 工具，
主要配置如 qmake、mingw32-make、g++等工具。在“系统变量”栏目下的“PATH”变量中添加如图 1-0 所示的内容(路径因版本及安装路径会有一些差异)

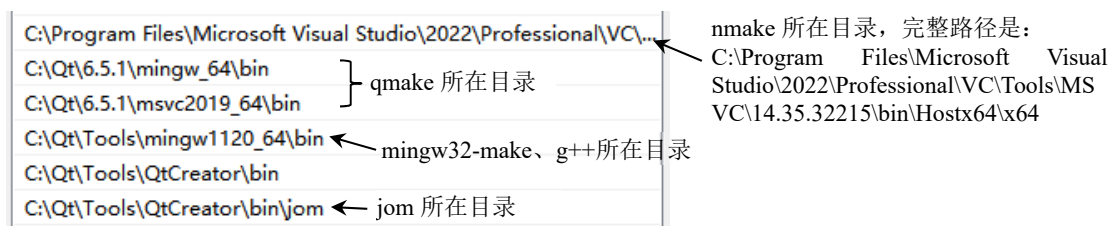


图 1-0 配置 Qt 工具的 PATH 环境变量

- 3、配置 CMake
将 cmake.exe 文件所在的目录添加到 PATH 系统变量中，具体路径依你所下载的 CMake 工具中的 cmake.exe 所在位置。若想使用 Qt 自带的 cmake.exe，则添加以下路径到 PATH 系统变量中(路径因版本及安装路径会有一些差异)
C:\Qt\Tools\CMake_64\bin
- 4、配置 VC++工具
要使用 VC++的 cl、link、lib 等工具，还需配置如下的环境变量(路径因版本及安装路径会有一些差异)
 - 1)、在 PATH 系统变量中增加以下目录
C:\Program Files\Microsoft Visual studio\2022\Professional\VC\Tools\MSVC\14.35.32215\bin\Hostx64\x64
 - 2)、新建 INCLUDE 系统变量，并在其中增加以下目录

C:\Program Files (x86)\Windows Kits\10\Include\10.0.22621.0\shared
C:\Program Files (x86)\Windows Kits\10\Include\10.0.22621.0\ucrt
C:\Program Files (x86)\Windows Kits\10\Include\10.0.22621.0\um
C:\Program Files (x86)\Windows Kits\10\Include\10.0.22621.0\winrt
C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\Tools\MSVC\14.35.32215\include

3)、新建 LIB 系统变量，并在其中增加以下目录

C:\Program Files\Microsoft Visual Studio\2022\Professional\VC\Tools\MSVC\14.35.32215\lib\x64
C:\Program Files (x86)\Windows Kits\10\Lib\10.0.22621.0\ucrt\x64
C:\Program Files (x86)\Windows Kits\10\Lib\10.0.22621.0\um\x64

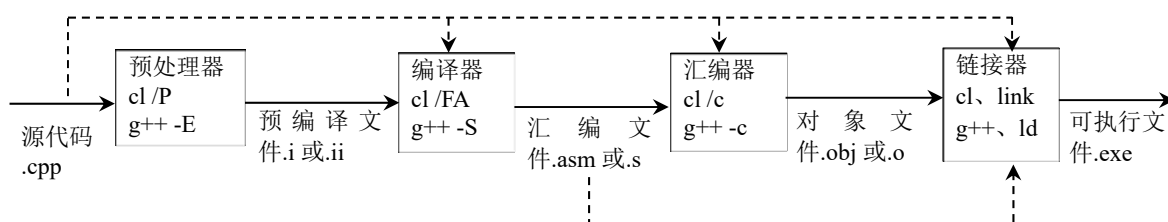
第 1 章 使用命令行工具构建程序

一、使用命令行编译 C++ 程序的步骤及使用的工具

1、C++编译过程分为 4 个阶段：预处理、编译、汇编、链接

- 预处理也称为预编译，该阶段主要处理 C++语言中的宏，即以“#”开始的指令和注释，该阶段不会检查错误。该阶段最终生成预编译文件，g++的后缀为.i，VC++后缀为.i。
- 编译阶段是整个流程的核心步骤，该阶段会对预处理文件进行词法分析、语法分析、语义分析、优化等，并且会检查语法错误。该阶段最终生成汇编文件，g++的后缀为.s，VC++后缀为.asm。
- 汇编阶段比较简单，就是把上一步生成的汇编文件翻译为机器代码，最终生成一个二进制文件，该文件被称为目标文件、中间文件、对象文件，g++的后缀为.o，VC++后缀为.obj。
- 链接阶段把多个不同的目标文件全部链接到一起，最终生成可执行文件，该步骤牵扯到的文件比较多。

2、图 1 为编译 4 阶段的图示，表 1-1 列出了 g++和 VC++对应于编译 4 阶段所使用的命令工具。



注：虚线表示可将此步骤的文件作为下一步骤的输入文件，cl 是 VC++的命令工具

图 1-1：编译的 4 个阶段

表 1-1 编译各阶段的命令工具

编译阶段	平台	命令工具	输入文件	输出文件	示例代码
预处理阶段	VC++	cl /P	.cpp	.i	cl /P /Fix.i a.cpp
	g++	g++ -E	.cpp	.i 或 .ii	g++ -E a.cpp -o a.i

编译阶段	VC++	cl /FA	.cpp、.i	.asm	cl /FA /Fax.asm a.cpp
	g++	g++ -S	.cpp、.i 或 .ii	.s	g++ -S a.cpp
汇编阶段	VC++	cl /c	.cpp、.asm	.obj	cl /c a.cpp
	g++	g++ -c	.cpp、.s	.o	g++ -c a.cpp
链接阶段	VC++	cl、link	.cpp、.obj、.asm、.lib 仅针对 cl 命令	.exe	cl a.cpp b.obj c.lib
	g++	g++、ld	.cpp、.o、.s、.a (g++) 仅针对 g++ 命令	.exe	g++ a.cpp b.o c.a

二、静态库文件及其相关命令工具

- 1、静态库(文件)是打包的对象文件。静态库不属于编译的 4 个阶段。静态库文件在 g++ 中也被称为文档 (archive)，其后缀为 .a，使用类似 ar 命令创建，VC++ 使用 lib 命令创建静态库文件，后缀为 .lib。
- 2、表 2 列出了 ar 和 lib 命令与静态库文件有关的常用参数，注意，ar 和 lib 命令还有其他非常多的功能，表 2 仅列出了其中一部分。对于 g++ 来讲，.a 文件仅是一个存档文件，其作用类似于一个打包工具，可以把任何文件都添加到 .a 文件中，但建议仅添加 .o 和 .a 文件。

表 1-2 g++ 的 ar 命令工具(静态库文件)

输入	输出	参数	说明	示例代码及说明
.o .a	.a	-q	创建并追加	ar -q x.a a.o //将 a.o 文件追加到 x.a(若不存在则创建)
		-r	创建并替换或插入	ar -r x.a a.o b.o //将 a.o、b.o 替换或插入 x.a(若不存在则创建)中
		-t	显示	ar -t x.a //显示 x.a 的内容
		-x	提取	ar -x x.a a.o //从 x.a 中提取(复制)出 a.o 文件
		-d	删除	ar -d x.a a.o //从 x.a 中删除 a.o 文件
		-a	定位到指定文件之后	ar -r -a b.o x.a c.o //将 c.o 添加到 b.o 之后。
		-b	定位到指定文件之前	ar -r -b b.o x.a c.o //将 c.o 添加到 b.o 之前。

表 1-3 VC++ 的 lib 命令工具(静态库文件)

输入	输出	参数	说明	示例代码及说明
.obj .lib	.lib .exp	无	创建.lib(默认名)	lib a.obj b.obj //创建 a.lib 并将 a.obj, b.obj 添加到其中
		/OUT:name	指定.lib 的名称	lib /OUT:x.lib a.obj b.obj //创建 x.lib 并添加 a.obj、b.obj
		/LIST[:file]	显示.lib 的内容	lib /LIST x.lib //显示 x.lib 的内容 lib /LIST:f.txt x.lib //将 x.lib 的内容显示到 f.txt 文件中
		/REMOVE:file	删除指定的 file	lib /MOVE:b.obj x.lib //将 b.obj 从 x.lib 中删除
		/EXTRACT:file	提取 file 到某处	lib /EXTRACT:b.obj x.lib //从 x.lib 中提取(复制) b.obj lib /extract:b.obj x.lib /OUT:c.obj //从 x.lib 提取 b.obj 并命名 c.obj

三、使用命令工具构建可执行文件

- 1、g++通常只需使用 g++、ar 命令就可以了，最常用的是 g++命令。VC++通常只需使用 cl、link、lib 命令就行了，使用最多的是 cl 命令。
- 2、本小节使用以下两个源文件，读者可自行创建类似的源文件，不需要创建太复杂，后文若未提及源文件的内容，读者可自行以此方式创建源文件。

代码清单 1-1: a.cpp

```
#include <iostream>
using namespace std;
void b();
int main(){ b();          cout<<"A"<<endl;          return 0;}
```

代码清单 1-2: b.cpp

```
#include <iostream>
using namespace std;
void b(){cout<<"B"<<endl;}
```

3、按编译步骤一步一步构建可执行文件

1)、g++

g++可使用-o 参数来指定输出文件的名称，但对于-E、-S、-c 参数时，-o 参数一次只能为一个文件指定输出文件名。

示例 1-1: g++按编译步骤构建可执行文件

```
//在 CMD 中输入以下命令
g++ -E a.cpp -o a.i    //①、预处理阶段。生成 a.i。
                        //须使用-o 指定输出文件名，否则，会将信息全显示到控制台上
g++ -E b.cpp -o b.i    //预处理阶段。生成 b.i
g++ -S a.i b.i         //②、编译阶段：生成 a.s 和 b.s
g++ -c a.s b.s         //③、汇编阶段：生成 a.o 和 b.o
g++ a.o b.o            //④、链接阶段：生成 a.exe。注：无论输入文件名为何，g++都默认生成 a.exe
a.exe                 //⑤、测试
```

2)、VC++

VC++生成.asm 文件会同时生成.obj 或.exe 文件，暂时还未找到单独生成.asm 文件的方法，并且这样生成的.asm 文件不能作为 cl、link 的输入文件，也不能作为专门处理.asm 文件的 ml.exe(64 位系统为 ml64.exe)命令的输入文件。

示例 1-2: VC++按编译步骤构建可执行文件

```
cl /P a.cpp b.cpp      //生成 a.i 和 b.i，可使用/Fxname(无空格)指定输出文件名
cl /c /FA /Faa.asm a.cpp //生成 a.asm 和 a.obj。/FA 表示生成.asm 文件，/Fa 指定.asm 文件名称
cl /c /FA /Fab.asm b.cpp //生成 b.asm 和 b.obj
cl /Fex.exe a.obj b.obj //生成 x.exe。/Fe(无空格，可省略后缀)用于指定输出的可执行文件的名称
//上一步也可使用 link 命令，如下
link /OUT:x.exe a.obj b.obj //生成 x.exe。其中/OUT 用于指定输出文件的名称
e.exxe                  //测试
```

4、常用的构建可执行文件的步骤

- 1)、无论是 g++ 还是 VC++，通常都使用以下几种方法来构建可执行文件
- ①、通过对象文件(.o 或.obj)来构建可执行文件
 - ②、通过静态库来(.a 或.lib)构建可执行文件。
 - ③、通过动态库(.dll 或.so)来构建可执行文件。动态库的方法详见后文。
 - ④、直接由源文件构建可执行文件。
- 2)、VC++ 主要使用 cl 和 link 命令，若 cl 命令不使用 /c 选项，则需要链接时会自动调用 link 命令，所以，很多地方都可使用 link 命令来代替 cl 命令。另外，VC++ 创建动态库(.dll 文件)比较麻烦，后文会对 link 命令和.dll 文件作详细的介绍。

示例 1-3: g++常用的构建可执行文件的方法

```
①、直接由源文件构建 exe 文件
    g++ -o x.exe a.cpp b.cpp           //生成 x.exe。

②、通过 .o 文件构建 exe 文件
    g++ -c a.cpp b.cpp                 //生成 a.o 和 b.o
    g++ -o x.exe a.o b.o               //生成 x.exe

③、通过静态库 (.a)来构建 exe 文件
    g++ -c a.cpp b.cpp                 //生成 a.o 和 b.o
    ar -q x.a a.o b.o                 //创建 x.a
    g++ x.a -o x.exe                  //生成 x.exe

④、通过动态库 (.so)来构建可执行文件
    g++ -shared b.cpp -o b.so          //生成 a.so，需使用 -o 参数，否则，可能不会创建 .so 文件
    g++ a.cpp b.so -o x.exe            //生成 x.exe
```

示例 1-4: VC++常用的构建可执行文件的方法

```
①、直接由源文件构建 exe 文件
    cl a.cpp b.cpp /FeY                //生成 y.exe。使用 /Fe(无空格，可省略后缀)指定输出的可执行文件名

②、通过 .obj 文件构建 exe 文件
    cl /c a.cpp b.cpp                 //生成 a.obj 和 b.obj
    cl a.obj b.obj /FeY               //生成 y.exe

③、通过静态库 (.lib)来构建 exe 文件
    //VC++在使用静态库构建 exe 时，不能将含有 main 函数的对象文件都放入 .lib 中。
    cl /c a.cpp b.cpp                 //生成 a.obj 和 b.obj
    lib /OUT:x.lib b.obj              //生成 x.lib。a.obj 不应放入 x.lib 中
    cl a.obj x.lib                    //生成 a.exe(默认名称)。

④、通过动态库 (.dll)来构建可执行文件，详见后文。
```

四、g++ 命令的常用参数

表 1-4 是 g++ 命令的部分常用参数

表 1-4 g++ 命令常用参数

注：g++ 的参数是大小写敏感的

1、生成文件	
-o	指定输出文件的名称。注：g++ 默认生成的可执行文件名称默认为 a.exe
-c	生成.o 文件(对象文件)。参见前文示例 1-1、示例 1-3

-E	生成.i 文件(预处理文件), 需同时指定-o 参数。参见示例 1-1
-C	在预处理时不删除注释, 通常和-E 一起使用。
-S	生成.s 文件(汇编文件) 。示例 1-1
-shared	尽量生成.so 文件(动态库文件), 应指定-o 参数以免不能生成.so 文件。参见示例 1-3
-save-temps	保留编译所产生的所有中间文件
-x c++ a.xx	将 C++源代码文件的后缀设为.xx, 要关闭该设置可使用-x none a.cpp。详见后文示例 1-5
@file	将 file 文件的内容作为输入。file 文件被称为响应文件(response file) 。详见后文示例 1-6
-include file	在其他文件之前包含 file。类似于在源代码中使用了#include <file>。详见后文示例 1-8

2、指定宏(详见后文示例 1-7)

-Dmacro	以 1 定义宏(预处理变量)macro, 参数 D 之后不需空格, 如-DXX 类似于在源代码中增加语句 #define XX 1
-Dmacro=n	定义宏, 比如-Dxx=3, 类似于#define xx 3
-Umacro	取消-D 参数的宏定义, 类似于#undef macro
-imacros file	将 file 文件中定义的宏添加到 g++的输入文件中

3、指定头文件搜索目录(详见后文示例 1-9)

以下头文件搜索顺序为-iquote、-I、-isystem、-idirafter

-Idir(大写 i)	在指定的目录查找头文件, 参数 I 之后不需空格
-I-	已废弃, 已被-iquote 代替。表示, 位于-I-左侧的-I 参数, 只对#include 命令的引号头文件进行搜索, 位于-I-右侧的-I 参数, 将对所有#include 命令的头文件名进行搜索, 无论头文件名有无引号。
-iquote dir	为#include 命令的非尖括号(或引号)头文件指定搜索目录
-isystem dir	为系统头文件指定搜索目录且优先于标准系统的目录被搜索
-dirafter dir	若在其他目录搜索不到头文件, 则搜索该目录
-nostdinc++	不搜索 C++标准系统包含(include)目录

4、指定库及搜索目录

-lxx(小写 L)	搜索库名称 xx.a 或 libxx.a, 若库未在标准目录下, 则需使用-L 参数指定目录
-Lg:/xx	在目录 g:/xx 下搜索文件
-lyy -Lg:/xx	在目录 g:/xx 下搜索文件 xx.a 或 libxx.a

5、优化、警告、调试相关

-O[n]	指定优化级别, 可以是-O0、-O1、-O2、-O3, 其中-O0 不优化, -O3 最高级别, -O1 是默认值
-w	关闭所有警告信息
-Wall	开启所有警告信息
-Wextra	开启额外警告信息
-Werror	将所有警告视为错误
-fexceptions	开启异常处理
-g	以默认格式生成调试信息
-ggdb	以默认扩展格式生成调试信息
-gstabs	生成 STABS 格式的调试信息

6、makefile 相关

-M	在控制台显示 make 依赖项, 即#include 指令所引起的所有依赖文件(含系统头文件), 注: 使用 -M 参数不会生成任何文件
-MD	同上, 但会生成文件, 并且会将依赖文件保存在.d 文件中
-MM	与-M 类似, 但忽略系统头文件

-MMD	与-MD 类似，但忽略系统头文件
7、语言规范相关	
-ansi	只支持 ANSI 标准的 c 语法。等同于-std=c89(C 语言)或-std=c++98(C++语言)
-fno-asm	禁止将 asm、typeof 等用作关键字
-fno-strict-prototype	认为不带参数的函数不是没有参数，而是没有显示的参数个数和类型说明
-fsigned-char	char 的默认类型为 signed
-fno-unsigned-char	同上
-funsigned-char	char 的默认类型为 unsigned
-fno-signed-char	同上
-Wa,option	将逗号分隔的 option 传递给汇编器。
-Wp,option	将逗号分隔的 option 传递给预处理器
-Wl,option	将逗号分隔的 option 传递给链接器。
-std=<standard>	指定 C++ 的标准。比如 -std=c++03 表示符合 2003 年修订的 C++ 标准，与-std=c++98 相同 -std=c++11 表示 c++11 标准，类似还有 c++14、c++17、c++20、c++23
-traditional	试图让编译器支持传统 C 语言特性
8、其他	
-v	显示编译器调用的程序，即，显示详细的编译、汇编、链接等命令
####	与-v 相同
--help	显示帮助信息，但内容比较少
-v --help	显示帮助信息，内容非常全面

示例 1-5: g++使用-x 参数指定源代码文件后缀

将代码清单 1-1 和 1-2 的 a.cpp 和 b.cpp 的文件的后缀修改为 a.xx 和 b.xx，然后在 CMD 中输入

```
g++ a.xx b.xx           //错误，无法识别文件 a.xx 和 b.xx
g++ -x c++ a.xx b.xx    //成功生成 a.exe
```

示例 1-6: g++使用@指定响应文件

新建一个 f.txt 的文件，并在其中输入以下内容

```
a.cpp b.cpp
```

然后在 CMD 中输入以下命令

```
g++ @f.txt              //成功生成 a.exe
```

示例 1-7: g++指定宏

输入以下代码并保存为 x.cpp

```
//x.cpp
#include <iostream>
int main(){ std::cout<<"A="<<E<<std::endl;    return 0;}
```

然后在 CMD 中输入以下命令

```
g++ x.cpp               //错误，名称 E 未定义
```

使用-D 参数指定宏

```
g++ x.cpp -DE          //生成 a.exe
a.exe                  //输出 A=1
```

使用-Dmacro=n 参数指定宏

```
g++ -x.cpp -DE=3       //生成 a.exe
a.exe                  //输出 A=3
```

使用-Umacro 参数取消宏定义

```
g++ -DE=4 x.cpp -UE -DE=5 //生成 a.exe。取消-DE=4，重新定义-DE=5
a.exe                     //输出 A=5
```

使用-include file 参数将 file 文件中定义的宏添加到输入文件。在 f.txt 中输入以下内容

```
//f.txt
#define E 6
```

在 CMD 中输入以下命令

```
g++ -include f.txt x.cpp //生成 a.exe。
a.exe                    //输出 A=6
```

示例 1-8: g++使用-include file 将文件包含(include)到输入文件

输入以下代码并保存为 x1.cpp，注意，以下代码没有#include<iostream>语句，用以验证 y1.cpp 被包含到 x1.cpp 中的具体位置

```
//x1.cpp
int main(){
    b();          //函数 b()未声明，直接调用
    std::cout<<"A"<<std::endl;
    return 0;}

```

输入以下代码并保存为 y1.cpp

```
//y1.cpp
#include <iostream>
void b(){    std::cout<<"B"<<std::endl; }

```

在 CMD 中输入以下命令

```
g++ x1.cpp y1.cpp //错误，因为 b()未声明
```

重新在 CMD 中输入以下命令

```
g++ x1.cpp -include y1.cpp //正确，生成 a.exe。相当于在 a.cpp 中增加语句#include "b.cpp"
a.exe                      //依次输出 B、A
```

示例 1-9: g++指定头文件搜索目录

输入以下代码并保存为 x2.cpp，

```
//x2.cpp
#include <iostream>
#include "f.h"          //包含双引号头文件
#include <g.h>           //包含尖括号头文件
int main(){    std::cout<<"f="<<f<<std::endl;    return 0;}
```

在 x2.cpp 文件所在目录新建两个目录 xx 和 yy，并在 xx 目录下创建一个 f.h 和 g.h 头文件，其中 g.h 头文件可以为空，在 f.h 文件中输入以下内容

```
//xx/f.h
int f=1;
```

然后在 yy 目录中新建一个 f.h 文件，并输入以下内容

```
//yy/f.h
int f=2;
```

在 CMD 中输入以下命令

```
g++ x2.cpp          //错误，找不到头文件 f.h 和 g.h
g++ -iquote xx x2.cpp  //错误，-iquote 只能为双引号头文件搜索目录，因此找不到头文件 g.h
```

-I、-idirafter、-isystem 参数都可以为尖括号头文件指定搜索目录

```
g++ -iquote xx -idirafter xx x2.cpp          //生成 a.exe
a.exe                                         //输出 f=1
g++ -Ixx x2.cpp                             //生成 a.exe
a.exe                                         //输出 f=1
```

搜索顺序-iquote > -I > -isystem > -idirafter

```
g++ -Ixx -iquote yy x2.cpp
a.exe                                         //输出 f=2。可见-iquote 优先于-I
```

五、cl 命令的常用参数

表 1-5 是 g++命令的部分常用参数

表 1-5 cl 命令常用参数

注：cl 的参数是大小写敏感的，且错误的参数会被忽略。
某些参数可合并使用，如/Oi /Os /Oy，可写成/Oisy

1、生成文件	
/c	生成.obj 文件(对象文件)，使用/Fo 参数指定文件名
/P	生成.i 文件(预处理文件)，使用/Fi 参数指定文件名
/FA	生成.asm 文件(汇编文件)，同时会生成.obj 或.exe 文件，cl 不能单独生成该文件，使用/Fa 参数指定文件名。ml(64 位机为 ml64)命令专门处理.asm 文件，但此处生成的.asm 不能作为 ml 的输入文件
/LD	生成.dll 文件(DLL 文件)，使用/Fe 参数指定文件名。若源代码有导出符号，还将同时生成导入库(.lib)和导出文件(.exp)，若在命令行使用了导出文件(.exp)，则不会生成导入库(.lib)

@file	指定响应文件(response file), 即, 将 file 文件的内容作为输入。该参数的使用方法与 g++的@参数类似, 详见前文示例 1-6	
/Foname 或/Fo:name	指定生成的.obj 文件的名称	若未使用/F...指定输出文件名, 则默认为输入文件基本名+对应的后缀。若.exe 未指定/Fe参数则是第一个输入文件的基本名+.exe
/Finame 或/Fi:name	指定生成的.i 文件的名称	
/Faname	指定生成的.asm 文件的名称	
/Fename 或/Fe:name	指定生成的.exe 或.dll 文件的名称	
/Zi	启用调试信息, 并生成一个单独的 PDB 文件来包含这些调试信息。	
/ZI(大写 i)	类似于/Zi, 但使用该参数生成的 PDB 文件支持编辑并继续功能, 并且/ZI 参数还会禁用代码中的所有#pragma optimize 语句。	
/Z7	启用调试信息, 并将这些信息包含在对象文件(.obj)中, 但, 若向链接器(link.exe)传递了/DEBUG 参数, 则仍会生成 PDB 文件	
/Fdbname 或/Fd:name	指定 PDB 文件的名称	
2、其他常用参数		
/nologo	取消显示版权信息	
/showincludes	在编译期间显示所有包含文件(包括嵌套的包含文件)的列表	
/link -option	将一个或多个 link 命令的参数传递给 link 命令	
3、宏和头文件		
/D name[=#[text]] /D "name[=#[text]]"	定义源文件的宏, 其使用方法与 g++的/D 类似, 详见前文示例 1-7, 其语法规则如下: <ul style="list-style-type: none">● /D 与 name 之间可以有空格也可以没有● name 与等号之间或等号与其值 text 之间不能有空格。● 若不为 name 指定值则默认为 1, 比如, /D E 等效于/D E=1	
/U[]symbol、/u	取消以前使用/D 定义的预处理器符号(即宏)。/u 可取消 Microsoft 定义的特定符号	
/I dir	指定包含文件(即头文件)的搜索目录, 即, 指定#include 的搜索目录, /I 和 dir 之间可有空格也可以没有, 若 dir 含有空格, 则需使用双引号括起来。其使用方法与 g++的/I 类似, 详见前文示例 1-9	
/X	防止在 PATH 和 INCLUDE 环境变量指定的目录搜索包含文件, 可将此参数与/I 配合使用, 以指定标准包含文件的备用路径。	
4、语言相关		
/Tpfilename /TP	将指定的文件视为 C++源文件, 即使没有.cpp 后缀。/Tp 一次只能指定一个文件, filename 和 /TP 之间的空格是可选的。/TP 表示将命令行上的所有文件都被视为 C++源文件。比如 <pre>cl a.xx //错误, 不能识别文件 a.xx cl /Tp a.xx //正确, 将 a.xx 视为 C++源文件 cl /Tp a.xx b.xx //错误, 不能识别文件 b.xx cl a.xx b.yy /TP //正确, 将 a.xx 和 b.yy 都视为 C++源文件</pre>	
/Tcfilename、/TC	与/Tp 和/TP 类似, 只是将文件视为 C 源文件(默认后缀为.c)。	
/std:option	指定 C 或 C++语言标准的版本, 该参数在 VS2017 以上版本提供。option 可取值有 c++14(默认)、c++17、c++20、c++latest、c11、c17	
/utf-8	将源和执行字符集设置为 UTF-8	
/J	将默认的 char 类型从 signed char 更改为 unsigned char, 并当将 char 扩展到 int 时, 对其进行扩展。若将 char 显示声明为 signed, 则/J 选项不起作用。	
/Za	禁用微软对 ANSI C89/ISO C90 不兼容的 C 语言的扩展	
/Zc:arg1[,arg2...]	用于指定 C++语言标准与编译器的行为, 使用逗号隔开多个/Zc 参数。比如/Zc:auto, 表示对	

	auto 强制实施 C++新标准(即自动推导变量类型), /Zc:auto-, 表示对 auto 不强制实施 C++新标准。/Zc 的参数比较多, 可参阅帮助文档或使用 cl /?命令查看
--	--

5、警告

编译器的警告以 Cnnnn 的形式表示, 比如 C4326 警告(1 级)。

参数/w1<n>, /w2<n>, /w3<n>, /w4<n>, /wd<n>, /wo<n>, /we<n>是互斥的。

/w	禁用所有警告。
/Wall	启用所有警告, 包含/W4 不包括的其他警告, 比如默认情况下关闭的警告。
/W<n>	设置警告的等级, n 取值 0~4, 其中 /W0: 禁止显示所有警告, 等效于/w /W1: 显示 1 级警告。/W1 是命令行的默认值 /W2: 显示 1 级和 2 级警告。 /W3: 显示 1 级、2 级、3 级警告。/W3 是 IDE 中的默认值 /W4: 显示 1、2、3 级警告, 以及默认情况下未关闭的所有 4 级警告。
/w1<n>, /w2<n>, /w3<n>, /w4<n>,	设置指定警告的警告级别, 这可以设置自己的警告标准, 而不使用 VS 提供的默认标准。比如 /w34326, 表示生成的 C4326 警告为 3 级警告, 而不是默认的 1 级警告。因此若/w34326 和/W2 同时使用, 则不会显示 C4326 警告。
/wd<n>	禁用指定的警告。比如/wd4326, 表示禁止显示警告 C4326
/wo<n>	仅显示一次指定的警告, 比如/wo4326, 将导致警告 C4326 只报告一次。
/we<n>	将指定的警告视为错误。比如/we4326, 将导致 C4326 警告被视为错误
/WX	将所有警告视为错误。
/Wv:xx[.yy[.zzzzz]]	禁用指定版本之后引入的警告, 仅显示指定版本或更早版本引入的警告
/WL	将额外消息附加到错误或警告消息中, 以显示到一行上, 使用分号进行隔开。通常, 错误和警告消息后面可能会出现额外的消息, 默认情况下, 这些额外消息会显示在新行上。
/sdl	启用更多安全功能和警告

6、优化

/favor:option	选择针对某个体系结构(即 CPU)进行优化。option 是以下值之一: AMD64、仅限 x64 INTEL64: 仅限 x64 ATOM: x86 和 x64, 表示 Intel Atom 处理器和 Intel Centrino Atom 处理器 blend: x86 和 x64, 生成针对 AMD 和 Intel 进行优化的代码。虽然该选项不能为特定的处理器提供最佳性能, 但能为广泛的 x86 和 x64 处理器提供最佳性能。
/Od	禁用所有优化并加快编译速度(默认值)。
/Os	优先优化代码大小
/Ot	优先优化代码速度。
/Ob<n>	控制内联函数的展开, n 可取 0~3, /Ob0 表示禁用内联展开(默认), /Ob 1 表示允许对标记有 inline、__inline、__forceinline 的函数或在类声明中定义的 C++成员函数进行展开。 /Ob 2 表示允许展开任何未明确标记为不内联的函数。 /Ob 3 是比/Ob 2 更积极的展开, /Ob3 自 VS2019 之后可用。
/Oy[-]	禁止在调用堆栈上创建帧指针(frame pointer), 以加快函数调用的速度。/Oy-表示禁用该参数。该参数在 x64 编译器中不可用。若需要基于帧的寻址, 则可在/Ox 之后指定 /Oy- 参数或使用 #pragma optimize("y",off), 以获得基于帧的寻址获得最大优化。/Oy 会取消显示帧指针信息,

	若启用调试信息(/Z7、/Zi、/ZI)，则建议指定/Oy-。
/Oi[-]	启用内部函数(intrinsic function)。将某些函数调用替换为内部函数或有助于提高运行速度的其他特殊形式的函数。内部函数由于没有函数调用的开销，其运行速度更快，但由于创建了其他代码，可能更大。/Oi 只是编译器的请求。
/Og	已弃用，建议改用/O1 或/O2。提供局部或全局优化、自动寄存器分配和循环优化。
/O1	代码大小最小化(最小大小)。等效于/Og /Os /Oy /Ob2 /GF /Gy
/O2	代码速度最大化(最大速度)。等效于/Og /Oi /Ot /Oy /Ob2 /GF /Gy
/Ox	代码速度优先，是/O2 的子集，不包含/GF 和/Gy，等效于/Ob2、/Oi、/Ot、/Oy。该参数与/O1、/O2、/Od 互斥。若将/Ox 与/Os 结合使用(即/Oxs)，则会取消速度优先。
/GF	消除重复字符串。编译器在内存创建相同字符串的单个副本，这种优化方法被称为字符串池，字符串池是只读的，若尝试修改/GF 下的字符串，会发生应用程序错误。字符串池允许原本为指向多个缓冲区的多个指针变成指向单个缓冲区的多个指针，比如，以下代码 p 和 p1 使用同一字符串初始化，使用/GF 编译，将使它们指向同一内存 char *p="aaa"; char *p1="aaa";
/Gy	允许编译器以打包函数(COMDAT)的形式打包单个函数
7、异常处理、浮点优化，详见下文正文	

1、结构化异常处理和/EH 参数

- 1)、结构化异常处理(Structured exception handling, SEH) 也被称为异步异常处理，是微软对 C 和 C++ 的扩展，并不是标准的 C 或 C++ 语法。SEH 主要是针对 C 语言的，因此，也被称为 C 结构化异常处理
- 2)、不建议使用 SEH，而是使用标准的 C++ 异常处理。但是，有时需要在 C++ 程序中混合使用 SEH 和标准 C++ 异常，此时需要使用/EHa 参数进行编译。注意，C 编译器不支持 C++ 异常处理语法，但 C++ 编译器支持 SEH 的语法(即支持 __try、__except、__finally)
- 3)、SEH 和标准 C++ 异常处理的区别
 - 标准 C++ 异常处理可处理多个类型，而 SEH 只处理一个类型(具体而言就是 unsigned int)
 - SEH 被称为异步的，因为异常是从属在正常控制流之后发生的，所以 SEH 也被称为异步异常处理。而标准 C++ 异常处理是完全同步的，这意味着异常仅发生在被引发时。
- 4)、SEH 有两种语法，如下：

```
__try 复合语句 __except(表达式) 复合语句
__try 复合语句 __finally 复合语句
```

关键定 __try、__except、__finally 是微软对 C 和 C++ 的扩展。__except 块，可以基于表达式的值来响应或消除异常，而 __finally 块，则无论异常是否导致终止，都将始终调用。

5)、/EH[option][-]参数

该参数用于指定异常处理的方式，/EH 参数主要用于设置是否启用结构化异常处理(SEH)，是否将 extern "C" 假定为 throw 异常，以及是否优化 noexcept 检查。各参数意义如下：

- “-” 表示清除上一个选项参数，option 可取 a、s、c、r 之一或其组合。
- a
启用标准 C++ 异常处理和 SEH。因此，支持使用 catch(...) 语法处理结构化异常和标准 C++ 异常。想要能处理 SEH，则需要指定该选项。
- s
只启用标准 C++ 异常处理不启用 SEH，因此，不能使用 catch(...) 语法捕获结构化异常。除非指

定/EHc, 否则, 假定声明为 `extern "C"` 的函数可能抛出(throw)C++异常。建议与/EHc 一起使用, 即/EHsc

- c

假定声明为 `extern "C"` 的函数不会抛出(throw)C++异常。与/EHa 一起使用时, 没有效果; 若未指定/EHs 或/EHa, 则忽略/EHc。建议与/EHs 一起使用, 即/EHsc

- r

始终为所有的 `noexcept` 函数生成运行时终止检查。若未指定/EHs 或/EHa, 则忽略/EHr。

- 默认异常处理行为

默认情况下(即, 不指定/EHsc、/EHs 或/EHa), 编译器始终生成支持 SEH 的代码, 支持 `catch(...)` 子句中的 SEH 处理程序。默认的异常展开代码不会销毁因异常而超出作用域的 `try` 块之外的自动 C++ 对象, 这在当抛出(throw)C++异常时, 可能会导致资源泄漏和未定义的行为。

- 关于/EHs、EHsc、EHa 的进一步说明

- 使用/EHs 或/EHsc, 则 `catch(...)` 子句都不会 `catch`(捕获)异步结构化异常, 并且在发生异步异常时作用域内的对象不会被销毁, 即使代码处理了异步异常也是如此。
- 使用/EHs 或/EHsc 时, 假定异常只发生在 `throw` 语句和函数调用中, 此假设可使编译器消除一些代码, 显著减小代码的大小, 若使用/EHa, 则可能会更大且较慢, 因为编译器不会主动优化 `try` 块。
- 建议不要将使用/EHa 编译的目标文件与使用/EHs 或/EHsc 编译的目标文件链接到同一个可执行模块中。
- 使用/EHa 时, 若在 C++ 程序中引发了 C 异常(结构化异常), 则可由结构化处理程序处理, 或由 C++ `catch` 处理程序处理。

2、浮点优点与/fp:option 参数

1)、对浮点代码进行优化, 可以提高程序速度。可对以下几个方面进行浮点优化: 中间结果、代数转换、浮点表达式的计算顺序、浮点环境、浮点收缩(contract)

2)、中间结果

在进行浮点表达式计算时, 通常会产生一些中间表达式(或称为中间结果、中间计算), 中间结果的精度通常能影响整个结果的精度。比如以下代码

```
float a, b, c, d, e;  
.....  
a = b*c + d*e
```

以上表达式通常按如下方式计算

```
float x = b*c;           //产生误差  
float y = d*e;           //再次产生误差  
a = x+y;
```

在以上计算步骤中的每一步中间计算都会产生单精度舍入误差, 为减少误差, 可使用更高的精度来计算中间结果, 比如以下代码就比上一步的精度更高

```
double x = b*c;          //使用更高精度的 double 计算中间结果  
double y = d*e;          //同上  
double z = x+y;  
a = (float)z;
```

很明显, 中间结果的精度越高, 则最终计算出来的结果的精度也越高。

3)、代数转换

代数转换是指的一些实数的算法规则，比如， a/b 转换为 $a*(1/b)$ 的运算，就是代数转换，这里将 a 除以 b 转换为 a 乘以 b 的倒数，再如， $a*(b+c)$ 转换为 $a*b+b*c$ ，也是代数转换。但是，实数的算法规则对于浮点数未必等效，比如，表 1-6 的实数算法规则对于浮点数就未必等效，即未必相等，比如，对于浮点数而言 $a*(b+c)$ 不一定等于 $a*b+b*c$ 。注意：实数和浮点数是不同的，浮点数是实数的一个子集。

表 1-6 对浮点数未必有效的实数算法规则

算法规则	说明
$(a+b)+c = a+(b+c)$	加法结合律
$(a*b)*c = a*(b*c)$	乘法结合律
$a*(b+c) = a*b + b*c$	分配律
$(a+b)(a-b) = a*a - b*b$	因式分解
$a/b = a*(1/b)$	使用倒数计算除法
$a * 1.0 = a$	乘法恒等式

4)、浮点表达式计算顺序(源代码排序)

浮点表达式计算顺序就是指的在计算多个浮点表达式时的计算顺序，通过改变表达式的计算顺序，有时能减少计算冗余，从而提高计算速度，比如以下代码

```
double a, b, c;
double x, y;
.....
x = a*a + b*b + c*c;
y = a + b + c;
```

若编译器按顺序计算 x 、 y ，则可能使用以下方式来生成代码

```
double a, b, c;
double x, y;
register r0, r1, r2;
.....
r0 = a;
r1 = r0*r0;           //r1= a*a;
r0 = b;
r2 = r0*r0;           //r2 = b*b
r0 = c;
r3 = r0*r0;           //r3 = c*c
r0 = r1+r2;
r0 = r0 +r3;
x = r0;               //x = r1 + r2 + r3
.....
r1 = a;
r2 = b;
r3 = c;
r0 = r1+r2;
r0 = r0 +r3;
```



```
y = r0;          //y = r1 + r2 + r3
```

以上运算过程有几个多余的运算，若能改变表达式的计算顺序则可消除冗余，比如

```
double a, b, c;
double x, y;
register r0, r1, r2;
.....
r1 = a;
r2 = b;
r3 = c;
r0 = r1+r2;
r0 = r0 +r3;
y = r0;          //y = r1 + r2 + r3
.....
r1 = r1*r1;      //减少了 r0=a;的赋值语句
r2 = r2*r2;      //同理，减少了 r0=b 的赋值
r3 = r3*r3;      //与以上类似
r0 = r1+r2;
r0 = r0 +r3;
y = r0;          //x = r1 + r2 + r3
```

5)、浮点环境

浮点环境是对浮点数的一些规范进行的设置，比如，浮点数的舍入方向(即舍入模式)，是否保留非规约(subnormal)值，无穷大是否设置为仿射模式，float、double 使用的默认有效数字(尾数)精度等，所以，禁止更改浮点环境最明显的影响就是不能更改舍入模式。

6)、浮点收缩(contract)

浮点收缩可以将浮点运算组合为单个指令，比如，混合乘加运算(FMA)，指类似于 $a*b+c$ 类型的运算，通常会执行两个操作，即先计算乘法，然后再相加，但若使用了浮点收缩，则会将 FMA 作为单个指令进行计算，并且在进行加法之前不会对中间乘积进行舍入操作，这样就没有中间舍入误差，但结果可能会与单独的乘法和加法运算不同。由于浮点收缩是作为单个指令执行的，所以，执行速度比单独计算更快。使用浮点收缩的优点在于提高了速度并减小了代码的大小，但浮点收缩无法检查中间值。

7)、/fp:option 参数

该参数用于指定如何优化浮点数计算，/fp 与 /Za 不兼容。option 必须是以下选项之一

- **contract(收缩)**

允许编译器考虑浮点收缩。这是 VS2022 新增的参数。使用浮点收缩的优点在于提高了速度并减小了代码的大小，所以，/fp:fast 默认会启用/fp:contract，但收缩无法检查中间结果，所以 /fp:strict 与/fp:contract 不兼容。

- **precise**

这是默认值。该模式在保证安全的前提下进行优化。使用该模式编译器在优化浮点操作时将严格遵守一组安全规则，这可以在保持浮点计算精确性的前提下生成高效的机器码，这种模式可创建既快速又精确的程序。说简单一点就是，该模式在保证安全的前提下，能优化则尽量优化。具体优化表现在以下几方面

- ①、中间结果：该模式总是以最高可行精度一致性地执行中间计算，并且总是以 FPU 寄存器精度执行中间计算，所以，中间结果的精确性与平台相关。在表达式求值期间，会在以下 4 种场合舍入到用户指定的精度：赋值、类型转换、参数传递、函数返回。
 - ②、代数转换：该模式不会对浮点表达式执行代数转换，除非可以保证转换的结果按位相同。
 - ③、浮点表达式计算顺序(源代码排序)：在保证优化安全的前提下，即，在最终结果不会改变且结果不依赖于浮点环境和浮点异常时，该模式允许编译器重新排列浮点表达式的计算顺序。也就是说，在某些情况下(比如，最终结果不相同)，该模式可以保留源代码排序，但在某些情况下，又可以允许重新排序。
 - ④、浮点收缩：在 VS2022 之前，会默认启用收缩，但 VS2022 之后，默认禁用收缩。
 - ⑤、浮点环境：禁止更改浮点环境。该模式假定在运行时不会访问或修改浮点环境。禁止更改浮点环境最明显的影响就是不能更改舍入模式。
 - ⑥、浮点异常：为便于产生更快的程序，默认情况下该模式禁用浮点异常
 - ⑦、特殊值：该模式会按 IEEE754 标准处理含有特殊值的表达式，比如，若 x 为 NaN，则 $x!=x$ 为 true。
- **fast**
允许牺牲精确性为代价换取速度的优化。该模式具有以下特点：
 - ①、中间结果：在赋值、类型转换、函数调用等情况时的舍入可能被忽略。可能使用更低的精度计算中间结果，以便产生更快更小的机器码，因此，无法保证中间计算的精度的一致性。
 - ②、代数转换：可能会进行代数转换，不保证转换的精确性和正确性。也就是说，即使计算的结果有可能不正确，但为了执行得更快，仍可能允许进行代数转换。
 - ③、浮点表达式计算顺序：可以对浮点表达式的计算重新排序，即使更改后可能改变最终结果。
 - ④、特殊值(如 NaN)：可能不会按照 IEEE754 的标准执行。
 - ⑤、浮点收缩：总是启用
 - ⑥、浮点环境：禁用
 - ⑦、浮点异常：总是禁用。因为与 `/fp:except` 不兼容
 - **strict**
提供了与 `/fp:precise` 的所有常规规则，但不同的是，默认情况下该模式启用了浮点异常，并总是启用浮点环境，但禁用了收缩。该模式的计算开销比 `/fp:precise` 更高，因为编译器需要插入额外的指令以捕获异常并允许在运行时修改浮点环境。
 - **except[-]**
启用浮点异常。默认情况下 `/fp:strict` 会启用浮点异常，而 `/fp:precise` 则不启用。该选项与 `/fp:fast` 不兼容，因此，`/fp:fast` 总是禁用浮点异常。`/fp:except-` 表示禁用浮点异常。

第 2 章 VC++ 动态库(DLL 文件)

一、简单的创建一个 DLL 文件

- 1、静态库 lib 和动态库 DLL 的一个显著区别是，静态库 lib 中的内容会被全部写入生成的可执行文件中，而动态库则不会，因此，把 lib 文件删除后生成可执行文件仍能运行，但删除 DLL 文件后可执行文件就会出错。另外，DLL 文件还能被其他可执行文件共享，而静态库则不行，只服务于当前的可执行文件。
- 2、VC++ 有两种调用 DLL 文件中的函数的方法
 - 隐式链接：这种方法通过一个中间文件来间接调用 DLL 中的函数，这个中间文件是 lib 文件，被称为导入库(文件)，但这个 lib 文件与静态库的 lib 文件是不同的，该文件中并不包含函数的代码，相当于是调用 DLL 函数的一个入口，而静态库的 lib 文件包含函数的代码。
 - 动态链接：这种方法使用 windows 的 LoadLibrary()、GetProcAddress() 等库函数来显示调用 DLL 中的函数，即需要使用 DLL 中的哪个函数就显示的调用哪个函数，但这种方法与编写 windows 的应用程序有关，所以本文不介绍。
- 3、VC++ 创建 DLL 的基本思维是，将需要共享的数据(即变量)或函数导出来，然后在调用方将这些导出的函数或数据导入调用方并使用之。导入/导出可简单的理解为导入/导出的是函数或数据的名称，当然，实际上并不仅仅是函数或数据的名称，这些名称还会包括其他一些信息，如函数的返回类型、参数等，所以，通常将其称为“符号(symbol)”，因此，DLL 的思维可简单的理解为：先导出符号，再导入符号并使用之。
- 4、下面以一个简单的示例介绍使用 __declspec(dllexport) 关键字创建及使用 DLL 的步骤。
 - 1)、创建一个 b.cpp 文件，并输入代码清单 1-3 的内容

代码清单 1-3: b.cpp(用于创建 DLL)

```
#include<iostream>
using namespace std;
/*使用__declspec(dllexport)关键字(注意，最前面是两个下划线_)将函数 f 导出。使用此关键字声明函数 f，表示函数 f 能被其他程序使用，即该函数可被共享，或称为将函数 f 暴露出来给其他其应用程序使用 */
__declspec(dllexport) void f() {cout<<"f"<<endl; }
```

- 2)、创建一个 b.h 文件，并输入以下内容

代码清单 1-4: b.h(用于创建 exe 文件)

```
/*使用__declspec(dllimport)关键字(注意，最前面是两个下划线_)声明导入函数 f。对于函数，该关键字可省略，使用该关键字的目的是可以加快查找 f 函数的速度。*/
__declspec(dllimport) void f();
```

- 3)、创建一个 a.cpp 文件，并输入以下内容

代码清单 1-5: a.cpp(用于创建 exe 文件)

```
#include<iostream>
#include"b.h" //包含头文件，头文件含 f()函数的声明
using namespace std;
int main(){
```

```
f(); //调用 DLL 中的函数 f()
cout<<"A"<<endl;
return 0;}
```

4)、在 CMD 中转到上述文件所在目录，输入以下命令

```
cl /LD b.cpp //使用 LD 参数。该命令会生成 b.obj、b.dll、b.lib、b.exp 四个文件
cl a.cpp b.lib //生成可执行文件，注意：不需指定 dll 文件，但需指定 b.lib 文件
a.exe //运行生成的程序。若删除 dll 文件则 a.exe 运行出错。
```

或输入以下命令

```
cl /c b.cpp //生成 b.obj
link /DLL b.obj //使用 /DLL 参数创建 .dll 文件，link 命令不能使用 .cpp 作为输入文件
link a.obj b.lib
a.exe
```

二、link 命令简介

- 1、link 是 VC++ 的链接命令，链接步骤是在将源代码编译成目标文件(.obj 文件)后执行的，link 将目标文件合并为可执行文件。使用 link 命令创建 exe 文件比较简单，只需将需要链接的对象文件或库文件添加到 link 命令行即可。本小节主要讲解怎样使用 link 创建 DLL 文件，DLL 文件的详细信息见后文。
- 2、cl 命令若不指定/c 参数，则在需要时会自动调用 link 命令，也就是说，虽然表面上只使用了 cl 命令，但实际上仍调用了 link 命令。cl 命令可使用参数对链接命令进行一些控制，比如 cl /LD 对应于 link /DLL，cl /Fdfilename 对应于 link /PDB:filename，cl filename.def 对应于 link /DEF:filename.def 等等。
- 3、link 的输入文件和输出文件
link 的输入文件不需指定后缀名，link 会检查每个输入文件以确定文件类型。link 命令可使用表 1-7 中的输入文件，同时 link 可创建表 1-8 中的输出文件。

表 1-7 link 命令的输入文件

注：link 命令不能接受.cpp 作为输入文件

后缀	中文名	说明
.obj	目标文件 对象文件	通用对象文件(Common Object File Format, COFF)的.obj 文件。.obj 文件也被称为中间文件、对象文件、目标文件。
.netmodule		link.exe 能接受 MSIL .obj 和 .netmodule 文件作为输入。.netmodule 文件由 cl /LN 命令或 link /NOASSEMBLY 创建。
.lib	静态库文件 导入库文件	VC++ 将 .lib 文件分为 COFF 标准库和 COFF 导入库。标准库就是常说的静态库文件，在其中含有 .obj 文件，说简单点就是，静态库文件只是将 .obj 文件打了一个包。导入库文件包含有 DLL 的位置和 DLL 导出函数的信息以及其他一些信息。
.exp	导出文件 export file	.exp 文件包含导出函数和数据项的信息，.exp 文件的作用类似于(但并不完全相同)DEF 文件，但 .exp 文件通常是在创建导入库的时候顺带创建的，有关 lib 和 exp 文件的更多内容，详见后文。当存在相互导入时，需使用 .exp 文件。若使用 .exp 文件创建 DLL，则 link 不会生成导入库，因为它假定 lib 已创建了一个。
.def	模块定义文件	该文件用于向 link.exe 提供有关导出、属性和有关链接的程序的信息。可使用 .def 文

	module-definition file	件创建 DLL 文件。由于该文件的信息可以由 lib 或 link 的/EXPORT 参数代替，还可以使用__declspec(dllexport)来指定导出名称，所以，.def 文件通常是不需要的。可使用 link 命令的/DEF 参数指定需调用的.def 文件的名称。
.pdb	程序数据库文件 program database	pdb 文件由 cl /Zi 命令创建，link 还会使用 pdb 文件保存.exe 或.dll 文件的调试信息。pdb 文件可以既是输出文件也是输入文件，link 在重构程序时会更新 pdb。
.res	资源文件	由 rc 命令创建，link 会自动将.res 文件转换为 COFF。
.exe	MS-DOS 存根文件 文件名	link 可以指定与 MS-DOS 一起运行的.exe 文件的名称。对应于 link 的/STUB 参数。
.txt	文件文件	@、/BASE 参数、/DEF 参数、/ORDER 参数，都可以指定文件文件，这些文件可以有任何后缀名，而不仅仅是.txt。@的使用方法与 g++的@参数类似，详见前文示例 1-6
.ilk	增量链接文件	用来存储增量链接信息。当进行增量链接(/INCREMENTAL 参数)时，link.exe 会更新该文件。若该文件缺失，link 会创建一个，若该文件不可用，link 将执行非增量链接

表 1-8 link 命令的输出文件

后缀	中文名	说明
.exe	可执行文件	默认生成.exe 文件
.dll	DLL 文件	使用/DLL 参数输出为.dll 文件
.ilk	增量链接文件	指定/INCREMENTAL 参数，即在增量模式下，将生成一个.ilk 文件，用于保存后续增量生成的状态信息。
.lib	导入库文件	若创建的程序(通常为 DLL)包含有导出，则还会生成一个.lib 文件(导入库文件)，除非生成中使用了.exp 文件。可使用/IMPLIB 参数控制导入库文件的名称
mapfile	映射文件	使用/MAP 参数将创建一个映射文件(mapfile)
.pdb	PDB 文件	使用/DEBUG 参数将创建一个 PDB 文件以包含程序的调试信息。

三、DLL 简介

1、DLL 基本思维---共享

动态链接库(DLL)简称动态库是一个模块，其中包含可由另一个模块(应用程序或 DLL)使用的函数和数据。说简单一点，DLL 文件中的某些函数或变量可被其他程序使用，他们是共享的，所以，DLL 也被称为共享库。

2、关键专用名词

- 1)、导出函数和内部函数：在 DLL 中分离出来的被其他程序所共享的函数被称为导出函数。DLL 中的其他函数是由 DLL 私有的内部函数，也就是说，内部函数只能由 DLL 使用。
- 2)、导出名称：在 DLL 中除了可导出函数外，还可导出数据(即变量)，本文将导出函数和导出的数据统称为导出名称。注意：导出名称实际上并不仅仅只包含函数或数据的名称，这些名称还会包括其他一些信息，如函数的返回类型、参数等，所以，通常将其称为“**符号(symbol)**”
- 3)、符号(symbol)：即 DLL 中导出的名称。
- 4)、导入库(或称为导入库文件)：含有 DLL 的位置和 DLL 导出函数的信息以及其他一些信息。在创建以静态链接方式使用 DLL 中的导出名称的应用程序时，需要将导入库添加到到 link 或 cl 命令行，即需要链接导入库文件。

3、DLL 与静态库

- 1)、从逻辑上来讲,使用 C++的全局变量和外部函数的源文件将其编译成静态库后,也能让其他程序使用这些函数和变量,实现类似共享的功能,但与 DLL 是有区别的。
- 2)、静态库和动态库的一个显著区别是,静态库中的内容会被全部写入生成的可执行文件中,而动态库则不会,因此
 - 使用静态库的可执行文件通常比使用 DLL 的更大。
 - 把静态库文件(.lib 文件)删除后,可执行文件仍能运行,但删除 DLL 文件后就会出错。
 - 另外, DLL 文件还能被其他可执行文件共享,而静态库则不行,只服务于当前的可执行文件。即, DLL 文件只需要一个,而静态库则会被写入各个可执行文件中,所以,实际上每个可执行文件中都存在一个相同的静态库。

4、DLL 和应用程序

- 1)、DLL 和应用程序都是可执行文件模块,这意味着 DLL 也是可执行的,只是缺少了类似 main 函数的入口函数,所以不能单独执行。其实,很多应用程序都不是一个完整的可单独执行的文件,他们都会被分割成多个相对独立的部分,比如,需要加载 DLL 的应用程序,其 DLL 就是被分割出来的那一部分,所以,这种需要 DLL 的应用程序,若缺少需要 DLL 也是不可执行的。
- 2)、DLL 和应用程序的区别在于
 - 应用程序可在系统中同时运行自身的多个实例,而 DLL 只能有一个实例。
 - 应用程序可作为进程加载,可以管理诸如堆栈、执行线程、全局内存、文件句柄、消息队列等资源,而 DLL 则不行。

5、使用 DLL 的好处

- 节省空间:使用 DLL 可节省磁盘空间,减小程序大小
- 共享内存:多个进程可以共同使用同一个相同的 DLL,并在内存中共享 DLL 只读部分的单个副本。
- 共享文件:多个应用程序可在磁盘上共享 DLL 的单个副本。
- 使用 DLL 更方便维护、更新、修复。当更新 DLL 时,不需重新编译或链接整个应用程序,只需处理 DLL 就可以了。

四、创建和使用 DLL 文件的思维

1、创建 DLL 文件的思维

根据 DLL 文件的共享原理,我们可以自然的想到以下的创建 DLL 文件的方法及步骤

- 1)、首先在源代码中确定出哪些函数或数据可以被共享,即,确定哪些函数或数据需要导出。我们自然的可以想到如下的确定导出函数或数据的方法
 - 将需要导出的函数或数据在源代码中使用一个特殊的关键字进行标记,比如, VC++使用 `__declspec(dllexport)` 进行标记,注意,最前面是两个下划线“_”。
 - 将需要导出的函数或数据的名称保存在一个特殊的文件中,然后使用该文件与包含这些导出名称的源代码一起创建 DLL 文件。比如, VC++使用 DEF 文件(后缀为.def)来定义 DLL 中的导出名称。
- 2)、确定好源代码中需要导出的函数或数据后,就可以使用 `cl` 或 `link` 命令创建 DLL 文件了。具体的创建 DLL 文件的方法详见后文。

2、使用 DLL 中的导出函数或数据的思维

要使用 DLL 中导出的函数或数据,很明显第一步应该加载(或链接)DLL 文件,加载 DLL 之后才能使用 DLL 中导出的函数或数据。VC++有以下两种加载(或链接)DLL 的方式(以导出函数为例)

- 1)、隐式链接:也称为静态加载或加载时(Load-Time)动态链接。

其原理为：操作系统将 DLL 与使用 DLL 的可执行文件同时加载，此时，调用 DLL 导出函数的方式与函数被静态链接并包含在可执行文件中的方式相同。具体执行原理及部分实现方法分为以下两阶段(详细的实现方法详见后文)：

- 创建可执行文件阶段

- ①、原理

- 当调用 DLL 的导出函数时，会生成一个外部函数引用，要解析这个外部引用，应用程序必须链接到创建 DLL 时创建的导入库文件(其后缀为.lib)，导入库仅用于加载 DLL 和实现对 DLL 中函数的调用，当在导入库中查找外部函数时，会告知链接器该函数的代码位于 DLL 中，于是，链接器将相关信息添加到可执行文件，告诉系统在进程启动时查找 DLL 代码的位置。说简单一点，此步骤的主要目的是写入一些信息，并让系统在启动进程时查找 DLL 的位置。

- ②、实现方法

- 具体的实现方式就是在使用 cl 或 link 命令创建含有 DLL 导出函数的可执行文件时，需链接该 DLL 的导入库。

- ③、示例：

- 假设，f.dll 的导入库为 f.lib，而 a.cpp 调用了 f.dll 中的导出函数，则在创建 a.exe 时，应使用以下命令

- ```
cl a.cpp f.lib //将 f.lib 链接到 a.cpp, f.lib 含有 DLL 的位置和有关导出函数的信息
```

或以下命令

```
cl /c a.cpp
link a.obj f.lib //将 f.lib 链接到 a.obj, f.lib 含有 DLL 的位置和有关导出函数的信息
```

- 运行可执行文件阶段

- 当系统启动包含 DLL 的程序时，将使用可执行文件中的信息查找所需的 DLL，若找不到 DLL，则终止进程，否则加载程序将 DLL 模块映射到进程地址空间中。说简单一点，以上过程主要就是将有关 DLL 的位置信息写入可执行文件中。

## 2)、显示链接：也被称为动态加载或运行时(Run-Time)动态链接。

其原理为：操作系统在运行时按需加载 DLL，即，需要使用 DLL 中的导出函数的时候才加载 DLL。显示链接不需要使用导入库。显示链接与编写 windows 的应用程序有关，所以本文不会重点介绍。使用此方式加载的 DLL，必须显示加载和卸载 DLL，并且必须通过函数指针来调用 DLL 中的导出函数。比如，使用类似于 LoadLibraryEx() 的函数加载 DLL 并获得模块句柄，然后再调用 GetProcAddress() 函数来获取 DLL 中导出函数的函数指针。以下是简化后的部分关键代码

```
#include "windows.h" //因为是 windows 应用程序，所以需包含该头文件
typedef void (*P)(); //P 是一个函数指针类型
P p1; //p1 是一个函数指针
HINSTANCE hDLL; //创建一个句柄 hDLL
hDLL = LoadLibrary("xx"); //加载 xx.dll 文件，并将获得的句柄赋给 hDLL
p1=(P)GetProcAddress(hDLL, "f"); //获取 xx.dll 中的名为 f 的导出函数的地址，并赋给函数指针 p1
.....
FreeLibrary(hDLL); //卸载 DLL
```

## 3、隐式链接和显示链接的区别

隐式链接和显示链接的一个明显区别在于，隐式链接在查找 DLL 时，若找不到，则会终止进程，而显示链接则不会终止，他会尝试从错误中恢复，比如，弹出一个对话框让用户指定所需的 DLL 文件的路径。另外，隐式链接需要导入库，而显示链接则不需要使用导入库。显示链接与编写 windows 的应用程序有关，而隐式链接则可用于非 windows 应用程序。本文不会详细讲解显示链接，但会详细讲

解隐式链接。

## 五、VC++创建 DLL 文件的方法

### 1、VC++创建 DLL 文件共有以下两步

#### 1)、确定导出名称，有以下两种方法

- 在源代码中使用 `__declspec(dllexport)` 关键字标记需要导出的函数或数据
- 使用 DEF 文件(后缀为 .def)
- 以上两种方法可以混合使用而不会出现错误。

#### 2)、使用 `cl /LD` 或 `link /DLL` 命令创建 DLL 文件。

### 2、由于 DEF 文件的内容可以被替代，所以，经过替代 DEF 文件后，实际上共有以下几种方法来导出名称，建议按以下顺序选择：

- 在源代码中使用 `__declspec(dllexport)` 关键字
- 使用 DEF 文件
- 使用 `link` 命令的 `/EXPORT` 参数，其语法如下(各选项的意义详见对 DEF 文件的讲解)：

```
/EXPORT:entryname[,@ordinal[,NONAME]][,DATA]
```

- 在源代码中使用以下 `#pragma` 指令

```
#pragma comment(linker,"export:definition")
```

其中 `definition` 的语法与 DEF 文件的 `EXPORT` 语句的语法类似。详见对 DEF 文件的讲解

### 3、使用 `__declspec(dllexport)` 关键字导出名称

可以在源代码中使用 `__declspec(dllexport)` 关键字将变量、函数、类、类成员函数导出，`__declspec(dllexport)` 将名称存储在 DLL 的导出表中。`__cdecl` 调用约定的函数不能使用 `__declspec(dllexport)` 关键字。使用 `__declspec(dllexport)` 关键字时可以不使用 DEF 文件。以下使用该关键字的语法要求

#### 1)、导出函数

```
__declspec(dllexport) void f(){} //将 f 标记为导出函数
```

#### 2)、导出类中的成员

导出类中所有公共数据成员和函数成员时，`__declspec(dllexport)` 关键字必须在类名的左侧，例如

```
class __declspec(dllexport) A:public B{}
```

### 示例 1-10：使用 `__declspec(dllexport)` 创建 DLL 文件

#### ①、编写如下源代码，并分别保存为 `f.cpp` 和 `e.cpp`

##### 代码清单 1-6: `f.cpp`(用于创建 DLL)

```
#include<iostream>
using namespace std;
__declspec(dllexport) void f() //将 f 导出，f 将被共享(即，可被其他程序使用)
{cout<<"F"<<endl; }
void g(){cout<<"G"<<endl; } //g 将是 DLL 的私有函数，不能被共享
```

##### 代码清单 1-7: `e.cpp`(用于创建 exe，使用隐式链接)

```
#include<iostream>
using namespace std;
```



```

void f(); //声明 DLL 中的导出函数, 省略了__declspec(dllexport)关键字
void g();
int main(){
 f(); //调用 DLL 中的导出函数
 //g(); //错误, 试图调用 DLL 的私有函数 g
 cout<<"Em="<<endl; return 0;}

```

②、在 CMD 命令行中输入以下命令以创建 DLL 文件

```
cl /LD f.cpp //使用 LD 参数创建一个名为 f.dll(默认名称)的文件
```

或, 输入以下命令

```

cl /c f.cpp //先生成一个 f.obj(默认名称)的对象文件
link /DLL f.obj //使用 f.obj 创建一个名为 f.dll(默认名称)的文件。

```

使用以上命令后, 总共会创建以下 4 个文件

f.obj、f.dll、f.exp、f.lib

其中 f.dll 是我们创建的 DLL 文件, f.obj 是对象文件, 这里的 f.lib 被称为导入库文件, 这不是常规的静态库文件, 这个文件是与 f.dll 有关的文件, 该文件中保存有有关 DLL 的导出函数的信息, 若使用隐式链接 DLL, 就需要使用该文件。关于 f.exp 和 f.lib 将在后文件进行详细详解。

③、接着再在 CMD 中输入以下命令以创建 exe 文件

```

cl /c e.cpp //先生成一个 e.obj(默认名称)的对象文件
link e.obj f.lib //将 e.obj 与导入库 f.lib 一起链接, 创建 e.exe
e.exe //依次输出“F”、“Em=”, 可见, e.exe 成功调用了 DLL 中的导出函数

```

或输入以下命令

```

cl e.cpp f.lib
e.exe

```

④、由本示例可见, 既可使用 cl 命令链接, 也可使用 link 命令链接, 使用 cl 命令似乎还更简洁, 但, 本文主要在于讲清楚创建 DLL 的详细过程, 在之后的示例中, 会尽量使用 link 命令, 而不使用 cl 命令。

#### 4、使用 DEF 文件导出名称

- 1)、DEF 文件被称为模块定义(module-definition)文件, 后缀为.def, 这是一个文本文件, 可使用记事本编辑, 其中包含有一个或多个描述 DLL 的各种特性的语句。使用 DEF 文件后, 可以在源代码中不使用\_\_declspec(dllexport)关键字导出名称, 但是 DEF 文件需要自己手动创建。
- 2)、使用 DEF 文件不但可以按名称导出, 还可以按序号导出名称, 这是使用\_\_declspec(dllexport)关键字与 DEF 的主要区别, 若只使用按序号导出生成的 DLL 文件, 则只会保存序号而不会保存名称, 这样的 DLL 文件大小会更小, 可节省空间, 若导出的名称非常多, 则可节省更多空间。
- 3)、但是 DEF 文件通常是不需要的, 可以被 link、lib 命令的参数/EXPORT 代替, 也可在源文件中使用\_\_declspec(dllexport)关键字代替 DEF 的功能
- 4)、最小的 DEF 文件应包含 LIBRARY 和 EXPORTS 两条语句, 有关 DEF 的详细内容请参阅后文对 DEF 文件的讲解

- LIBRARY 语句的语法如下:

```
LIBRARY dllName
```

此语句用于指定创建的 DLL 文件的名称，还可以在 link 命令使用/OUT 参数来指定创建的 DLL 名称。链接器会将此名称放在 DLL 的导入库中。其实这条语句可以省略，在使用 link 命令创建 DLL 时，会创建一个与对象文件同名的但后缀为.dll 的 DLL 文件。

- EXPORTS 语句

该语句列出了 DLL 的导出名称和序号值(可选)，序号值是一个跟在“@”之后的一个数字，其最大值为 DLL 导出函数的数量，最小值为 1。要导出数据(即变量)需指定 DATA 关键字。

#### 示例 1-11：一个简单的 DEF 文件

```
LIBRARY xx
EXPORTS
 f @1 NONAME
 g
 a DATA
```

以上 DEF 文件表示，创建一个名称为 xx.dll 的 DLL 文件，包含一个变量 a、函数 f 和 g，其中 f 的序号值为 1，选项 DATA 表示导出的是一个数据(即变量)，选项 NONAME 表示该导出名称仅使用序号值而不使用名称。

5)、若要导出 C++ 文件中的函数，则应在 DEF 文件中使用修饰(decorated)名称，或使用 extern "C" 通过标准 C 链接来定义导出函数，特别是当源代码中有函数重载时，必须使用修饰名称。

6)、名称修饰(decorated)

也被称为名称重整(name mangling)、名称粉碎(name mangling)、名称改编(name mangling)，是指编译器在编译期间为函数、变量、对象等创建内部名称，说简单一点，就是编译器内部对名称进行“重命名”。修饰名(或名称)并不仅仅是重命名了函数的名称，还包括函数返回类型、参数等信息，比如，void f();被 VC++ 重命名为“?f@@YAXXZ ”(因 VC++ 版本而有所不同)。查看修饰名的方法请参阅后文。

#### 示例 1-12：使用 DEF 文件创建 DLL

①、编写如下源代码，并分别保存为 f.cpp 和 e.cpp

##### 代码清单 1-8：f.cpp(用于创建 DLL)

```
#include<iostream>
using namespace std;
//以下代码不使用__declspec(dllexport)关键字
void f() {cout<<"F()"<<endl;}
void f(int) {cout<<"F(int)"<<endl;} //重载 f()函数
void f1() {cout<<"F1()"<<endl;}
```

##### 代码清单 1-9：e.cpp(用于创建 exe，使用隐式链接)

```
#include<iostream>
using namespace std;
void f(); void f(int); void f1(); //声明 DLL 中的导出函数
int main(){
 f(); f(2); f1(); //调用 DLL 中的导出函数
 cout<<"Em="<<endl;
 return 0;}
```

②、使用记事本编写如下代码，并保存为 d.def

#### 代码清单 1-10: d.def (DEF 文件, 用于创建 DLL)

```
EXPORTS
 ?f@@YAXXZ
 ?f@@YAXH@Z
 f1
```

其中?f@@YAXXZ 是 void f()的修饰名, ?f@@YAXH@Z 是 void f(int)的修饰名, f1 因为没有重载, 直接使用的函数名称。

#### ③、可使用以下几种方法创建 DLL 文件(建议使用 link 命令)

方法 1: 使用 link 命令

```
cl /c f.cpp //创建 f.obj 文件
link /DEF:d.def /DLL f.obj //使用/DEF 参数指定 def 文件, 并创建 f.dll 文件
```

方法 2: 使用 link 的参数/EXPORT 替换 DEF 文件

```
cl /c f.cpp
link /EXPORT:f1 /EXPORT:?f@@YAXXZ /EXPORT:?f@@YAXH@Z /DLL f.obj
```

方法 3: 使用 cl 的/LD 参数和 DEF 文件

```
cl /LD f.cpp d.def //直接指定 DEF 文件
```

方法 4: 使用 cl 的/LD 和/link 参数

```
cl /LD f.cpp /link /DEF:d.def
```

使用以上命令后, 总共会创建以下 4 个文件

f.obj、f.dll、f.exp、f.lib

注意: 若没有 f.lib 和 f.exp, 而只有 f.dll, 则表示以上命令没有成功使用 d.def 创建 f.dll, 此时的 f.dll 中不会有导出函数。

#### ③、接着再在 CMD 中输入以下命令以创建 exe 文件

```
link e.obj f.lib //将 e.obj 与导入函数 f.lib 一起链接, 创建 e.exe
e.exe //依次输出 “F()”、“F(int)”、“F1()”、“Em=”
//可见, e.exe 成功调用了 DLL 中的 3 个导出函数
```

## 六、查看修饰名称的方法

名称修饰(decorated)也被称为名称重整(name mangling)、名称粉碎(name mangling)、名称改编(name mangling)。VC++有如下几种查看修饰名称的方式

#### 1、使用 dumpbin /EXPORTS 命令, 如下:

```
dumpbin /EXPORTS a.lib
```

使用该命令可查看.lib、.obj、.dll、.exp 等文件的信息, 但只能在.lib 和.dll 文件中查看到导出函数的修饰名称, 这意味着在查看导出函数修饰名时需要先创建.lib 或.dll 文件。

#### 2、使用 link /MAP 命令, 如下:

```
link /MAP a.obj
```

使用该命令会生成一个后缀为.map 的文件，该文件被称为映射文件(mapfile)，这是一个文本文件，里面含有非常多的有关链接的信息，如，来自程序文件的时间戳、入口点、公共符号列表(包含符号名、平面地址、定义符号的.obj 文件)等。由于.map 文件的内容太多，若想要在其中查找函数的修饰名称比较麻烦，而且生成.map 文件时就意味着创建了.dll 或.exe 文件。

### 3、利用错误消息查看修饰名称

在含 main 函数的源代码中先声明函数但不定义，再直接调用该函数，此时会产生一个类似如下的错误信息

```
无法解析的外部符号 "void __cdecl g(void)" (?g@@YAXXZ),
```

其中的?g@@YAXXZ 就是“void g()”函数的修饰名称。

## 七、DEF 文件

1、DEF 文件被称为模块定义(module-definition)文件，后缀为.def，DEF 文件是一个文本文件，可使用记事本编辑，该文件提供有关导出函数以及与链接有关的信息，通常用于创建 DLL 文件，该文件可被 lib 或 link 的/EXPORT 参数和\_\_declspec(dllexport)关键字代替，所以，通常不需要使用 DEF 文件。

2、DEF 文件的基本语法规则如下：

- 区分大小写
- 注释由分号开头，注释不能与语句位于同一行。
- 含有空格或分号的长文件名需使用双引号括起来。
- 使用空格、制表符、换行符将各个语句、参数等隔开。
- NAME 或 LIBRARY 语句(若有)必须位于其他语句之前。
- SECTIONS 和 EXPORTS 语句可在 DEF 文件中多次出现。

3、在 DEF 文件中与 DLL 有关的语句是 LIBRARY 和 EXPORTS，这也是经常使用的两个语句。DEF 文件中的每个语句 link 命令都有一个与之对应的相同功能的参数。

4、EXPORTS 语句

其语法为：

```
EXPORTS
 definition
```

其中，definition 的语法如下：

```
entryname[=internal_name|other_module.exported_name] [@ordinal [NONAME]] [[PRIVATE] | [DATA]]
```

1)、EXPORTS 语句用于指定一个或多个导出名称，可以指定函数或数据(变量)的导出名称或序号。与该语句对应的 link 参数是/EXPORTS。

2)、每个 definition(定义)必须在单独的一行上。第一个 definition 可以与 EXPORTS 关键字在同一行。

3)、DEF 文件可以有多个 EXPORTS 语句。

4)、definition 的各选项的意义如下：

- ①、entryname 是导出的函数或变量的名称，这是必填字段，这个名称是调用方使用的名称。
- ②、internal\_name 是 DLL 中导出函数或变量的名称，若 DLL 中的导出名称与调用方使用的导出名称不同，则使用该字段。对于 C++函数，该字段应使用修饰名称或在源代码中使用 extern "C"来定义导出函数，当有函数重载的情形时，必须使用修饰名称。比如，若 DLL 中有导出函数 f，但希望调用方使用名称 x，则应使用以下代码

```
EXPORTS
```

```
x=f
```

- ③、`other_module.exported_name` 表示来自其他模块的名称，比如，若有来自其他模块 `yy` 的名称 `f`，并希望调用方使用名称 `x`，则应使用以下代码

```
EXPORTS
x=yy.f
```

- ④、若导出的名称来自另一个按序号导出的模块，则使用 `other_module.#ordinal` 指定序号，比如，若有一个其他模块 `yy` 且序号为 4 的函数，并希望调用方使用名称 `x`，则应使用以下代码

```
EXPORTS
x=yy.#4
```

- ⑤、`@ordinal` 用于指定序号，使用序号导出主要用于兼容旧版本，虽然使用序号可减小 DLL 的大小，但不建议使用，因为 `.lib` 文件将包含序号与函数之间的映射。
- ⑥、若指定 `NONAME` 关键字，则表示只按序号导出，而不按名称导出，这可减小 DLL 的大小。
- ⑦、`PRIVATE` 关键字用于防止将 `entryname` 包含在 `link` 命令生成的导入库中。这相当于该名称是 DLL 私有的，不能被共享。
- ⑧、`DATA` 关键字用于指定导出的是数据(即变量)而不是代码。

## 5、LIBRARY 语句

其语法为

```
LIBRARY [library][BASE=address]
```

- 1)、`LIBRARY` 用于告诉 `link` 创建一个 DLL 文件。`library` 用于指定创建的 DLL 文件的名称，DLL 名称还可以使用 `link` 命令的 `/OUT` 参数来指定。
- 2)、`BASE=address` 用于设置操作系统加载 DLL 的基址。默认的 DLL 位置为 `0x10000000`。其作用与 `link` 的 `/BASE` 参数相同。

## 6、HEAPSIZE 语句

其语法为

```
/HEAP:reserve[,commit]
```

其作用与 `linke` 命令的 `/HEAP` 参数相同，用于指定堆的大小。`reserve` 参数指定虚拟内存中的堆的总分配。默认堆大小为 1 MB。`commit` 参数指定一次性分配的物理内存量

## 7、NAME 语句

其语法为：

```
NAME [application][BASE=address]
```

用于指定主输出文件的名称。其作用与 `link` 的 `/OUT` 参数相同，若二者都指定了，则使用 `/OUT` 的名称。`BASE` 用于设置基址，其作用与使用 `link` 的 `/BASE` 相同，若生成 DLL 文件，则 `NAME` 仅影响 DLL 名称。

## 8、SECTIONS 语句

其语法为

```
SECTIONS
definitions
```

`definitions` 的格式为

```
.section_name specifier
```

- 1)、该语句用于设置映像(image)文件中的一个或多个段(section)的属性。其作用与 link 的/SECTION 参数相同, /SECTION 选项更改一个段的属性, 覆盖在编译该段(section)的.obj 文件时设置的属性。有关详细信息请参阅 PE 文件。
- 2)、每个 definitions 必须在单独的一行上。第一个 definitions 可以与 EXPORTS 关键字在同一行。
- 3)、DEF 文件可以有多个 EXPORTS 语句。
- 4)、.section\_name 是程序映像(image)中某个段(section)的名称,
- 5)、specifier 是以下访问修饰符之一(使用空格分隔):
  - EXECUTE: 此段可执行
  - READ: 允许读取操作
  - SHARED: 在所有加载映像(image)的进程之间共享该段
  - WRITE: 允许写入操作

## 9、STACKSIZE 语句

其语法为:

```
STACKSIZE reserve[,commit]
```

设置堆栈的大小(以字节为单位), 其作用与使用 link 的/STACK 参数相同。此语句对 DLL 不起作用。reserve 值指定虚拟内存中的总堆栈分配。commit 值由操作系统解释, 在 WindowsRT 中, 它指定一次性分配的物理内存量。

## 10、STUB 语句

其语法为:

```
STUB:filename
```

- 1)、该语句只在构建 VxD(virtual device driver, 虚拟设备驱动程序)时有效, 其作用与使用 link 的 /STUB 参数相同, /STUB 选项将 MS-DOS 存根程序附加到 Win32 程序, 任何有效的 MS-DOS 应用程序都可以是存根程序。如果在 MS-DOS 中执行文件, 则会调用存根程序, 它通常显示合适的消息。
- 2)、在构建 VxD 时, 允许指定一个文件名, 该文件名包含有助于 VxD 的 IMAGE\_DOS\_HEADER 结构(在 WINNT.H 中定义), 而不是默认头文件。

## 11、VERSION 语句

其语法为:

```
VERSION major[.minor]
```

在 exe 或 DLL 的头文件中放入一个数字(即版本号)。其作用与使用 link 的/VERSION 参数相同。

# 八、导入库和导出文件(隐式链接)

## 1、以隐式链接的方式使用 DLL 导出的名称需要以下步骤及文件:

- 1)、在源代码中使用 \_\_declspec(dllimport) 关键字声明 DLL 中导出的名称。该关键字是可选的, 但使用该关键字可生成更高效的代码。注意, 若要访问 DLL 中的数据(即变量)和对象, 则必须使用 \_\_declspec(dllimport) 关键字, 也就是说, 若导入的名称是函数, 则可省略 \_\_declspec(dllimport) 关键字。
- 2)、使用的 DLL 文件。

### 3)、链接导入库文件(后缀为.lib)。可使用以下两种方法链接导入库

- 将导入库文件添加到 link 命令行。
- 在源代码中使用如下的#pragma 指令

```
#pragma comment(lib, "libName")
```

其中, libName 是导入库的名称

## 2、导入库文件可使用以下方法创建

### 1)、使用 link 命令

- 当源代码中含有\_\_declspec(dllexport)关键字时,在使用 link 命令创建 DLL 文件时,会同时创建一个导入库文件(.lib 的文件)和一个导出文件(.exp 文件)。
- 在使用 link /DEF 命令创建 DLL 时,会同时创建一个导入库文件(.lib 的文件)和一个导出文件(.exp 文件)。
- 注意:如果不满足以上两个条件,而使用 link 命令创建 DLL,此时不会创建导入库文件和导出文件,因为源代码没有导出任何名称。以上两种方法在前文已多次讲解。

### 2)、使用 lib 命令。

通常不需要 lib 命令来创建导入库,因为,在链接一个含有导出名称的程序时,link 会自动创建一个导入库文件和导出文件。导出文件通常也是不需要使用的,但在相互导入(或称为循环导出)的情况时,需要使用 lib 命令,此时还必须使用导出文件。lib 命令除了可创建导入库和导出文件外还可创建静态库以及其他一些功能,本小节仅介绍使用 lib 命令创建导入库和导出文件的方法。

## 3、导出文件(.exp 文件)

- 1)、.exp 文件包含导出函数和数据项的信息,.exp 文件的作用类似于(但并不完全相同)DEF 文件,但.exp 文件通常是在创建导入库的时候顺带创建的。在相互导入的情况时,必须.exp 文件。由于.exp 文件与 DEF 文件类似,因此,可使用.exp 文件来创建 DLL,若使用.exp 文件创建 DLL,则 link 不会生成导入库,因为它假定 lib 已创建了一个。.

### 2)、.exp 文件与 DEF 文件的不同之处如下:

使用.exp 与 DEF 文件的不同之处在于,DEF 文件中的导出名称的信息是手动编写的,因此,在源代码中可以不使用\_\_declspec(dllexport)关键字来导出名称,而.exp 文件中导出名称的信息是根据源代码文件所生成的,因此,源代码中需要包含\_\_declspec(dllexport)关键字,否则,这些名称不会被导出。

### 3)、在以下情况下需要使用.exp 文件来创建 DLL

- 未使用 DEF 文件导出名称,也没有在源代码中使用\_\_declspec(dllexport)关键字导出名称,而是使用 lib 命令的/EXPORT 参数导出的名称,则这时需要使用.exp 来创建 DLL,若不这样,则创建的 DLL 将没有导出任何名称。详见示例 1-14。
- 相互导入时,需要使用.exp 文件。详见示例 1-15

## 4、使用 lib 命令创建导入库语法如下:

```
LIB /DEF[:deffile] [options] [objfile] [libfile]
```

- 1)、要使用 lib 创建导入库必须指定/DEF 参数,即使不需 DEF 文件。

- 2)、lib 命令的输入文件不能是.cpp 源代码文件,但可以是.obj 和.lib 文件。

### 2)、各参数意义如下:

deffile 表示 DEF 文件名称, options 表示 LIB 的其他参数, objfile 表示.obj 文件的名称, libfile 表示.lib 文件的名称。

- 3)、因此,使用 lib 共有以下三种方法创建导入库(同时还会创建导出文件)

- 指定的 objfile 或 libfile 中包含有\_\_declspec(dllexport)关键字定义的名称

- 指定一个 DEF 文件。
- 使用 lib 的/EXPORT 参数，其语法如下：

```
/EXPORT: entryname[=internalname][,@ordinal[,NONAME]][,DATA]
```

以上参数各选项的意义与 DEF 文件语句相同，请参阅 DEF 文件的讲解。

### 示例 1-13: 使用 lib 创建导入库

1)、本示例将演示

- 怎样单独使用 lib 命令创建导入库。
- 怎样使用.exp 文件创建 DLL 文件。
- \_\_declspec(dllexport)关键字的使用。

2)、编写如下源代码，并分别保存为 f.cpp 和 e.cpp

#### 代码清单 1-11: f.cpp(用于创建 DLL)

```
__declspec(dllexport) int a=1;
__declspec(dllexport) int b=2;
```

#### 代码清单 1-12: e.cpp(用于创建 exe)

```
#include<iostream>
using namespace std;
__declspec(dllimport) int a; //由于 a 是变量不是函数，所以，必须使用
 //__declspec(dllimport)关键字，
 //否则会被编译器理解为位于 e.cpp 中定义的新变量。
int b; //未使用__declspec(dllimport)关键字，变量名称 b 未被导入
 // 所以，b 是 e.cpp 中定义的新变量
int main(){
cout<<a<<endl; cout<<b<<endl; cout<<"Em="<<endl; return 0;}
```

3)、在 CMD 中输入以下命令

①、创建对象文件

```
cl /c e.cpp f.cpp //创建 e.obj 和 f.obj
```

②、创建导入库

```
lib /DEF f.obj //创建一个名为 f.lib 的导入库和 f.exp 的导出文件。注意，必须使用/DEF 参数
```

③、创建 DLL 文件

```
link /DLL f.obj f.exp //使用 f.exp 和 f.obj 创建一个名为 f.dll 的 DLL 文件
 //由于使用了.exp 文件，所以本命令不会再重复
 //创建一个名为 f.lib 的导入库文件。
```

若把 f.cpp 文件中变量 a 和 b 的 \_\_declspec(dllexport)关键字删除，然后生成一个 f.exp 文件，再由该 exp 文件来创建 f.dll，则 f.dll 中将不会包含有导出的变量 a 和 b。

④、创建 exe 文件

```
link e.obj f.lib //使用 lib 命令创建的 f.lib 创建 e.exe 文件
```

⑤、测试结果



```
e.exe //分别输出 1、0、Em。结果如预期。
```

4)、本示例若将 f.cpp 文件中的

```
__declspec(dllexport) int b=2;
```

修改为

```
extern int b;
```

则会出现 b 未定义的错误。

#### 示例 1-14: 使用 lib 的/EXPORT 参数创建导入库

1)、编写如下源代码, 并分别保存为 f.cpp 和 e.cpp

##### 代码清单 1-13: f.cpp(用于创建 DLL)

```
int a=1; //不使用__declspec(dllexport)关键字
int b=2; //同上
```

##### 代码清单 1-14: e.cpp(用于创建 exe)

```
#include<iostream>
using namespace std;
__declspec(dllimport) int a; //必须使用__declspec(dllimport)关键字
__declspec(dllimport) int x; //同上。我们准备使用/EXPORT 参数将 b 的名称导出为 x。
int main(){
 cout<<a<<endl;
 cout<<x<<endl;
 cout<<"Em="<<endl; return 0;}
e.exe
```

3)、在 CMD 中输入以下命令

```
cl /c e.cpp f.cpp //创建 e.obj 和 f.obj
lib /DEF /EXPORT:a /EXPORT:x=b f.obj //使用/EXPORT 创建 f.lib 和 f.exp, 而不创建 DLL
//注: /DEF 参数仍然是必须的
//由于本示例没有使用 DEF 文件, 也未使用__declspec(dllexport)关键字,
//所以, 在创建 DLL 时需要使用.exp 文件, 否则, 创建的 DLL 文件没有名称导出
link /DLL f.obj f.exp //使用 f.exp 和 f.obj 创建 f.dll 文件
link e.obj f.lib //创建 e.exe 文件
e.exe //测试。依次输出 1、2、Em。结果如预期。
```

若使用以下命令则不需使用.exp 文件来创建 DLL

```
cl /c e.cpp f.cpp //创建 e.obj 和 f.obj
//使用 link 命令的/EXPORT 导出名称, 并同时创建 f.dll
link /DLL /EXPORT:a /EXPORT:x=b f.obj
link e.obj f.lib //创建 e.exe 文件
e.exe //测试。
```

#### 5、相互导入

1)、比如 a.dll 和 b.dll, 相互导入是指, a.dll 需要使用 b.dll 中的名称, b.dll 又使用了 a.dll 中的名称。在这种情况下, 无论先创建哪一个 dll 文件, 都需要另一个 dll 的导入库作为输入, 也就是说, 若

不首先创建一个文件，则无法创建另一个文件。

- 2)、存在相互导入的情况时，必须使用 lib /DEF 命令和 exp 文件，此时，可以使用 lib /DEF 命令预先创建出导入库和导出文件而不创建 DLL 文件，无论有多少 DLL 文件，都可以先创建出导入库和导出文件。下面以示例进行讲解相互导入的原理。

#### 示例 1-15：相互导入

- 1)、编写如下源代码，并分别保存为 f.cpp、g.cpp、e.cpp，其中 f.cpp 和 g.cpp 中的变量 b 和 a 相互导入。

##### 代码清单 1-15：f.cpp(用于创建 DLL)

```
#include<iostream>
using namespace std;
__declspec(dllexport) int a=1; //导出 a
__declspec(dllimport) int b; //导入 b
__declspec(dllexport) void f() //导出 f
{cout<<"b="<<b<<endl; }
```

##### 代码清单 1-16：g.cpp(用于创建 DLL)

```
#include<iostream>
using namespace std;
__declspec(dllexport) int b=2; //导出 b
__declspec(dllimport) int a; //导入 a
__declspec(dllexport) void g() //导出 g
{cout<<"a="<<a<<endl; }
```

##### 代码清单 1-17：e.cpp(用于创建 exe)

```
#include<iostream>
using namespace std;
void f(); //导入 f。省略了__declspec(dllimport)关键字
void g(); //导入 g。同上
int main(){
f(); g(); cout<<"Em="<<endl; return 0;}
```

- 2)、在 CMD 中输入以下命令

##### ①、创建对象文件

```
cl /c e.cpp f.cpp g.cpp //创建 e.obj、f.obj、g.obj
```

##### ②、错误情形：

此时无论是使用

```
link /DLL f.obj
```

还是

```
link /DLL g.obj
```

都会产生错误，因为 f.cpp 和 g.cpp 相互导入。

##### ③、使用 lib /DEF 预先创建导入库和导出文件

```
lib /DEF f.obj //创建 f.lib 和 f.exp。也可先创建 g.lib 和 g.exp
```

#### ④、创建 g.dll

```
link /DLL g.obj f.lib //此步骤会生成 g.lib 和 g.dll
```

#### ⑤、创建 f.dll

```
link /DLL f.obj f.exp g.lib//也可不使用 f.exp，但这样会再次重复生成一个 f.lib
```

#### ⑥、创建 e.exe

```
link e.obj f.lib g.lib
```

#### ⑦、测试

```
e.exe //依次输出 b=2、a=1、Em=。结果如预期。
```

## 九、导入/导出类

### 1、\_\_declspec(dllexport)和\_\_declspec(dllimport)关键字

为描述方便，本小节将\_\_declspec(dllimport)关键字简称为 dllimport，将\_\_declspec(dllexport)关键字简称为 dllexport。dllexport 和 dllimport 不是 C/C++语言的规范，而是微软(Microsoft)对 C/C++的专用扩展，用于从 DLL 中导出或导入名称(函数、数据和对象)。

### 2、dllexport 和 dllimport 的基本语法规则

- 1)、必须具有外部链接才可进行导出/导入。有关 C++链接性的语法细节请参阅《C++语法详解》一书
- 2)、当使用 dllexport 和 dllimport 声明名称时，若使用 dllexport 意味着定义，使用 dllimport 意味着声明，这意味着
  - dllimport 不能作用于定义语句，否则会产生错误。
  - dllexport 既可用于声明语句也可用于定义语句，除非是使用带有 extern 的 dllexport 语句进行强制声明，否则 dllexport 语句会进行隐式定义。
- 3)、如果使用 dllexport 特性声明函数或对象，则其定义必须出现在同一程序的某个模块中。否则，将生成链接器错误。
- 4)、如果程序中的单个模块包含对同一函数或对象的 dllimport 和 dllexport 声明，则 dllexport 特性优先于 dllimport 特性。但是，会生成编译器警告。

### 示例 1-16: dllexport 和 dllimport 语法规则

```
//*****dllimport*****
__declspec(dllimport) void x(){} //错误，不能用于定义
__declspec(dllimport) void x1(); //正确，导入名称 x1

__declspec(dllimport) int a1=1; //错误，不能用于定义
static __declspec(dllimport) int a2; //错误，名称 a2 具有内部链接性，不能导入
__declspec(dllimport) int a3; //正确，这是一个声明，或导入名称 a3
void f(){
 static __declspec(dllimport) int a4; //错误，名称 a4 具有内部链接性，不能导入
 __declspec(dllimport) int a5; //①。正确，这是一个声明，或导入名称 a5
 __declspec(dllimport) int a6=1; //错误，不能用于定义
 extern __declspec(dllimport) int a7; //②。正确，这是一个声明，或导入名称 a7
```

```

}

//*****dllexport*****
__declspec(dllexport) int b; //正确，导出名称 b
extern __declspec(dllexport) int b1; //正确，导出名称 b1，这时声明，因此必须在某处定义 b1
static __declspec(dllexport) int b2; //错误，名称 b2 具有内部链接性，不能导出
void g()
{__declspec(dllexport) int b3; //③。错误，试图导出局部作用域的名称 b3
 __declspec(dllexport) int b4=1; //④。错误，同上
extern __declspec(dllexport) int b5; //⑤。正确，导出 b5，这是声明，必须在某处定义 b5
}

```

从以上示例的①、②处可以见到，可在局部作用域使用\_\_declspec(dllexport)关键字或带有 extern 的 \_\_declspec(dllexport)关键字导入名称，但要函数内部导出名称是行不通的(但可从类中导出成员)，语句⑤仅是正确的语法，若不带 extern 关键字将发生如③或④的错误。

### 3、导入/导出类的基本规则有：

- 1)、若使用 dllimport 或 dllexport 来声明类，则表示将导入或导出整个类，这种类被称为可导出类。
- 2)、若整个类已导入或导出，则禁止将类的成员声明为 dllimport 或 dllexport。
- 3)、可以使用 dllexport 导出成员函数和静态数据成员，但必须在同一程序中的某处提供定义，并且静态数据成员必须在头文件中定义。
- 4)、需要注意的是，如果要使用 DLL 中的类成员(成员函数或静态数据成员)，则这些类成员要么需要使用 dllexport 导出(也可导出整个类)，要么需要在头文件中定义。

#### 示例 1-17：导出整个类

- 1)、编写如下源代码，并分别保存为 f.h、f.cpp、e.cpp

##### 代码清单 1-18：f.h(用于创建 DLL)

```

class __declspec(dllexport) A{ //注意__declspec(dllexport)关键字的位置
public:
 int b;
 A(){b=0;}
 A(int x){b=x;}
 void f();

 //__declspec(dllexport) void f1(); //错误，整个类已被导出，不应再对成员使用 dllexport
};

```

##### 代码清单 1-19：f.cpp(用于创建 DLL)

```

#include<iostream>
#include "f.h" //包含 f.h 头文件
using namespace std;
void A::f(){cout<<"f(b)="<<b<<endl;} //定义类 A 的成员函数 f()

```

##### 代码清单 1-20：e.cpp(用于创建 exe)

```

#include<iostream>
#include "f.h" //包含 f.h 头文件
using namespace std;

```

```
A mal; //不要使用__declspec(dllexport)关键字，因为没有导出名称 mal
int main(){
 mal.b=3;
 mal.f();
 cout<<"mal.b="<<mal.b<<endl;
 cout<<"Em="<<endl;
 return 0;}

```

2)、在 CMD 中输入以下命令

```
cl /c e.cpp f.cpp //创建 e.obj、f.obj
link /dll f.obj //创建 f.dll、f.exp、f.lib
link e.obj f.lib //创建 e.exe
e.exe //测试，依次输出 f(b)=3、mal.b=3、Em=

```

3)、以上示例看似与导出的类毫无关系，但实际是有关系的，若把生成的 f.dll 文件删除，则 e.exe 文件将会产生找不到 f.dll 的错误。

### 示例 1-18：导出类中的成员

1)、编写如下源代码，并保存为 f.h(用于生成 f.dll)

#### 代码清单 1-21：f.h(用于创建 DLL)

```
class A{public:
 int b;
 static __declspec(dllexport) int b1;
 static int b2;
 static int b3;
 //__declspec(dllexport) int b4; //错误，b4 不是静态数据成员，不能导出
 A(){b=0;}
 A(int x){b=x;}
 void f();
 __declspec(dllexport) void f1();
};

int A::b1=3; //应在头文件中定义静态数据成员
int A::b2=4; //同上

```

#### 代码清单 1-22：f.cpp(用于创建 DLL)

```
#include<iostream>
#include "f.h" //包含 f.h 头文件
using namespace std;
void A::f(){cout<<"f(b)="<<b<<endl;}
__declspec(dllexport) void A::f1(){ //可省略__declspec(dllexport)关键字
 cout<<"f1(b)="<<b<<endl; }
int A::b3=5; //未在头文件中定义静态数据成员

```

#### 代码清单 1-23：e.cpp(用于创建 exe)

```
#include<iostream>
#include "f.h" //包含 f.h 头文件

```

```
using namespace std;
A ma;
//__declspec(dllimport) int A::b1; //语法错误。也就是说不能直接导入静态数据成员
int main(){
 //ma.f(); //错误，成员函数 f() 未导出
 ma.f1(); //输出 f1(b)=0
 cout<<"ma.b="<<ma.b<<endl; //输出 ma.b=0
 //输出静态数据成员的值。
 cout<<"ma.b1="<<ma.b1<<endl; //输出 ma.b1=3
 cout<<"ma.b2="<<ma.b2<<endl; //输出 ma.b2=4
 cout<<"A::b1="<<A::b1<<endl; //输出 A::b1=3
 cout<<"A::b2="<<A::b2<<endl; //输出 A::b2=4
 //以下输出将产生无法解析的外部符号的错误，因为静态数据成员 b3 未在头文件中定义
 //cout<<"ma.b3="<<ma.b3<<endl; //错误
 //cout<<"A::b3="<<A::b3<<endl; //错误
 return 0;}

```

2)、在 CMD 中输入以下命令

```
cl /c e.cpp f.cpp //创建 e.obj、f.obj
link /dll f.obj //创建 f.dll、f.exp、f.lib
link e.obj f.lib //创建 e.exe
e.exe //测试

```

#### 示例 1-19：使用 DLL 中的类成员注意事项

1)、编写如下源代码，并分别保存为 f.h、f.cpp、e.cpp

##### 代码清单 1-24：f.h(用于创建 DLL)

```
class A{public:
 int b;
 A(){b=0;}
 A(int x){b=x;}
 void f();
 __declspec(dllexport) void f1();
 void f2();
};
void A::f(){std::cout<<"f()="<<b<<std::endl;}

```

##### 代码清单 1-25：f.cpp(用于创建 DLL)

```
#include<iostream>
#include "f.h"
void A::f1(){std::cout<<"f1()="<<b<<std::endl;}
void A::f2(){std::cout<<"f2()="<<b<<std::endl;}
__declspec(dllexport) A ma(9);

```

##### 代码清单 1-26：e.cpp(用于创建 exe)

```
#include<iostream>
#include "f.h"

```

```

using namespace std;

__declspec(dllimport) A ma;
A mal;
int main(){
 ma.f(); //输出 9
 ma.fl(); //输出 9
 mal.f(); //输出 0
 mal.fl(); //输出 0
 //ma.f2(); //错误，无法解析的外部符号。因为 f2() 既未在头文件中定义也未被导出。
 //mal.f2(); //同上。
 cout<<"Em="<<endl;
 return 0;}

```

4)、在 CMD 中输入以下命令

```

cl /c e.cpp f.cpp //创建 e.obj、f.obj
link /dll f.obj //创建 f.dll、f.exp、f.lib
link e.obj f.lib //创建 e.exe
e.exe //测试

```

## 第 2 篇 makefile 目录

|                                                      |    |
|------------------------------------------------------|----|
| 第 2 篇 makefile.....                                  | 50 |
| 第 1 章 makefile 工具简介 .....                            | 50 |
| 第 2 章 makefile 基本原理 .....                            | 53 |
| 2.1 makefile 基础 .....                                | 53 |
| 2.1.1 为什么需要 makefile.....                            | 53 |
| 2.1.2 makefile 基本语法结构.....                           | 53 |
| 2.1.3 makefile 基本语法要求.....                           | 54 |
| 2.1.4 makefile 执行规则的基本原则.....                        | 54 |
| 2.2 makefile 目标 .....                                | 55 |
| 2.3 makefile 规则和 order-only 依赖 .....                 | 60 |
| 第 3 章 变量(也称为宏)、函数、条件语句.....                          | 63 |
| 3.1 变量 .....                                         | 63 |
| 3.2 define 指示符和 call、eval 函数(nmake 不适用) .....        | 67 |
| 3.2.1 使用 define 定义变量 .....                           | 67 |
| 3.2.2 使用 define 定义命令包 .....                          | 68 |
| 3.2.3 使用 call 函数向 define 中传递参数.....                  | 69 |
| 3.2.4 将 define 与 eval 函数联合使用.....                    | 70 |
| 3.3 自动变量、通配符和部分默认变量 .....                            | 71 |
| 3.3.1 自动变量 .....                                     | 71 |
| 3.3.2 通配符 .....                                      | 72 |
| 3.3.3 部分默认变量 .....                                   | 72 |
| 3.4 变量和函数的展开过程(nmake 不适用).....                       | 73 |
| 3.5 makefile 中的函数和条件语句 .....                         | 75 |
| 3.5.1 makefile 函数.....                               | 75 |
| 3.5.2 makefile 条件语句.....                             | 76 |
| 第 4 章 模式规则 .....                                     | 79 |
| 4.1 静态模式规则(nmake 不适用).....                           | 79 |
| 4.2 模式规则(nmake 不适用).....                             | 80 |
| 4.2.1 模式规则匹配原理 .....                                 | 80 |
| 4.2.2 模式规则的启动 .....                                  | 82 |
| 4.2.3 搜索适用的模式规则及多目标模式模式规则.....                       | 82 |
| 4.3 隐式模式规则(implicit pattern rules) (nmake 不适用) ..... | 84 |
| 4.4 中间文件、万能匹配规则、默认规则 .....                           | 86 |
| 4.4.1 中间文件(intermediate file) .....                  | 86 |
| 4.4.2 万能匹配规则(match-anything rule, 简称万能规则) .....      | 88 |
| 4.4.3 默认规则 .....                                     | 89 |
| 4.5 隐式规则搜索算法 .....                                   | 90 |



|                                                    |     |
|----------------------------------------------------|-----|
| 4.6 后缀规则(.SUFFIXES) (nmake 不适用) .....              | 92  |
| 第 5 章 make 读取 makefile 文件的方式 .....                 | 95  |
| 5.1 make 执行 makefile 文件的过程 .....                   | 95  |
| 5.1.1 常规读取 makefile 文件的方式.....                     | 95  |
| 5.1.2 include 指示符.....                             | 96  |
| 5.1.3 MAKEFILES 和 MAKEFILE_LIST 变量(nmake 不适用)..... | 97  |
| 5.1.4 重制 makefile (nmake 不适用) .....                | 98  |
| 5.1.5 make 执行过程总结.....                             | 99  |
| 5.2 获取依赖 .....                                     | 100 |
| 5.2.1 自动获取依赖(nmake 不适用) .....                      | 100 |
| 5.2.2 目录搜索(nmake 不适用) .....                        | 102 |
| 第 6 章 递归调用(nmake 不适用).....                         | 107 |
| 6.1 make 工具的部分参数 (nmake 不适用).....                  | 107 |
| 6.2 递归调用(nmake 不适用).....                           | 108 |
| 6.2.1 递归调用与 MAKE 变量.....                           | 109 |
| 6.2.2 传递变量给子 make (export、unexport 指示符) .....      | 110 |
| 6.2.3 传递 make 命令参数给子 make(特殊变量 MAKEFLAGS).....     | 113 |
| 第 7 章 推理规则和内联文件(仅适用于 nmake).....                   | 116 |
| 7.1 推理规则(仅适用于 nmake).....                          | 116 |
| 7.1.1 基本语法及原理 .....                                | 116 |
| 7.1.2 预定义推理规则 .....                                | 118 |
| 7.2 内联文件(仅适用于 nmake).....                          | 118 |
| 总结 .....                                           | 121 |

# 第 2 篇 makefile

## 第 1 章 makefile 工具简介

- 1、本小节所列出的路径或目录因所安装的软件版本及路径而异，仅作参考。
- 2、本小节将介绍 makefile 文件和 C++程序的编译工具，包括 cl、g++、nmake、jom、mingw32-make、Ninja、qmake、CMake，其中 cl 和 nmake 是 Visual Studio(简称 VS)的工具，其作用如表 1-1 所示。本文不会使用 IDE 来编译程序，而是使用命令工具来编译。
- 3、nmake、jom 和 GNU 的 mingw32-make、make 工具都是用来解释 makefile 文件的，本文将解释 makefile 文件的工具统称为 make 工具或 make。将执行 makefile 文件的命令如 jom.exe、nmake.exe、mingw32-make.exe 等统称为 make 命令，当单独指定某命令时省略后缀，比如 nmake 命令、jom 命令等。
- 4、像 C++一样，makefile 文件也有自己的编写规则，有自己的语法、关键字、函数和书写格式。由于各平台的 make 工具对 makefile 文件的解释有一些区别，因此他们的语法会有些不同，但绝大多数是相同的，通常是以 GNU 的 makefile 为标准的。本文以 GNU 的 makefile 和 VS 的 makefilee 标准为例进行讲解，重点讲解 GNU 的 makefile 标准，使用 mingw32-make 和 nmake 工具。
- 5、为便于区分各平台的 makefile 标准，本文将 GNU 的 makefile 标准称为 make，将 VS 的 makefile 标准称为 nmake。

表 1-1 编译工具简介

| 编译工具         | 说明                                   |
|--------------|--------------------------------------|
| cl           | Visual Studio 的 C++编译工具              |
| link         | Visual Studio 的 C++链接工具              |
| g++          | GNU 的 C++编译器                         |
| ld           | GNU 的链接工具                            |
| nmake        | Visual Studio 用于解释 makefile 文件的工具    |
| jom          | Qt 用于解释 makefile 文件的工具               |
| make         | GNU 用于解释 makefile 文件的工具              |
| mingw32-make | GNU 用于在 Windows 环境下解释 makefile 文件的工具 |
| Ninja        | 一个诞生于 Google Chrome 项目的构建工具。         |
| qmake        | Qt 用于生成 makefile 文件的工具               |
| cmake        | CMake 用于生成 makefile 文件的工具            |

## 6、C++编译过程分为 4 个阶段，即：预处理、编译、汇编、链接

- 预处理也称为预编译，该阶段主要处理 C++语言中的宏，即#开始的指令和注释，该阶段不会检查错误。该阶段最终生成预编译文件，g++的后缀为.i，VC++后缀为.i。
- 编译阶段是整个流程的核心步骤，该阶段会对预处理文件进行词法分析、语法分析、语义分析、优化等，并且会检查语法错误。该阶段最终生成汇编文件，g++的后缀为.s，VC++后缀为.asm。
- 汇编阶段比较简单，就是把上一步生成的汇编文件翻译为机器代码，最终生成一个二进制文件，该文件被称为目标文件或中间文件，g++的后缀为.o，VC++后缀为.obj。
- 链接阶段把多个不同的目标文件全部链接到一起，最终生成可执行文件，该步骤牵扯到的文件比较多。

## 7、使用记事本编写一个如代码清单 1-1 所示的代码，并将其保存为 m.cpp。

### 代码清单 1-1 m.cpp 文件的内容

```
#include<iostream>
using namespace std;
int main(){ cout<<"Qt"<<endl; return 0;}
```

在 cmd 中转至 m.cpp 文件所在目录，依次输入如图 1-1 所示命令。

|                                                                                                                                                            |                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>G:\qt1&gt;g++ -E m.cpp -o m.i G:\qt1&gt;g++ -S m.i -o m.s G:\qt1&gt;g++ -c m.s -o m.o G:\qt1&gt;g++ m.o -o m.exe G:\qt1&gt;m.exe Qt G:\qt1&gt;_</pre> | <p>参数-o 用于指定生成文件的名称</p> <p>1、预处理：使用参数-E，生成预编译文件，后缀为.i</p> <p>2、编译：使用参数-S(注意：这是大写 S)，生成汇编文件，后缀为.s</p> <p>3、汇编：使用参数-c(小写)，生成目标文件，后缀为.o</p> <p>4、链接：生成可执行文件，链接不需使用参数</p> <p>运行生成的可执行文件 m.exe</p> <p>m.exe 输出的内容</p> |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

图 1-1 C++编译的 4 个阶段

以上 4 个步骤可以使用以下命令中的其中一个命令一次性完成，第一个命令只能看到生成的 m.exe 文件，第二个命令使用参数-save-temps(注意，中间没有空格)可以保留编译 4 个阶段生成的相应文件。

```
g++ m.cpp -o m.exe
g++ m.cpp -save-temps -o m.exe
```

打开 m.cpp 所在目录，可以看到生成的全部文件，如图 1-1 所示。

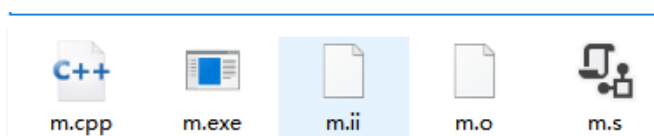


图 1-1 编译 4 个阶段 g++ 生成的相应文件



图 1-2 cl 命令生成的相应文件

## 8、编译的 4 个阶段，通常只关心汇编和链接这两个过程，即，只关心目标文件.o 或.obj 文件与可执行

文件。以下是使用 `cl` 命令生成 `obj` 文件和可执行文件的命令，图 1-2 是生成的相应文件。

```
cl /c m.cpp
```

```
cl m.obj
```

也可使用以下命令一次性完成以上步骤，其中参数 “/Fo:” 用于重命名生成的文件名。

```
cl m.cpp /Fo:m.exe
```

## 9、mingw32-make 和 nmake 命令基本使用方法

### 1)、mingw32-make -f xxxx

以上命令表示执行名称为 `xxxx` 的文件中的默认目标中的命令

### 2)、mingw32-make

以上命令表示执行默认文件(如 `makefile`)中的默认目标中的命令。若无特别说明，本文的所有示例均是指的执行默认目标中的命令。

### 3)、mingw32-make xxx

以上命令表示执行默认文件(如 `makefile`)中的目标 `xxx` 中的命令

### 4)、mingw32-make -f xxx yyy

以上命令表示执行名称为 `xxx` 的文件中的目标 `yyy` 中的命令

### 5)、mingw32-make AA=xx

以上命令表示执行默认文件(如 `makefile`)中的默认目标中的命令，并定义一个名称为 `AA`，值为 `xx` 的变量传递给 `makefile` 文件。

## 第 2 章 makefile 基本原理

### 2.1 makefile 基础

#### 2.1.1 为什么需要 makefile

- 1、当有多个文件时，需要使用类似以下的命令来编译程序

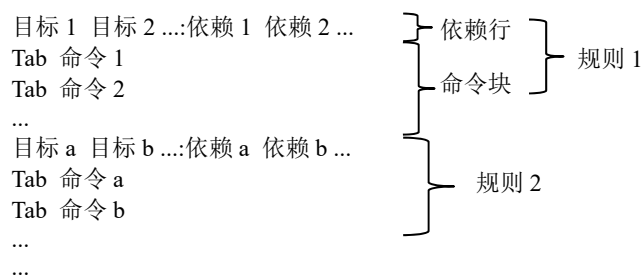
```
g++ -c a.cpp -o a.o
g++ -c b.cpp -o b.o
g++ -c c.cpp -o c.o
.....
g++ -o m.exe a.o b.o c.o
```

编译时需要一个一个的列出所有的文件名，当文件数量不是很多时，可以选择这样的编译方式，但是在一个大型项目中，文件的数量都是成百上千甚至上万的，除此之外，还有可能要去链接第三方库，而且还有可能涉及到文件链接顺序的问题，因此，如果每次编译都要列举出这些文件，这将是一个灾难，解决这个问题其实很简单，就是将这些命令打包成一个批处理文件或写成一个脚本，然后在编译时只需执行这个批处理文件或脚本就可以了。

- 2、使用脚本或批处理解决了列举文件的问题，但是，一个项目免不了要修改源文件，如果每次修改都去重新编译所有文件，对于一个大型项目来说，这可是非常耗费时间的东西，若文件特别大的话有可能花上半天甚至更多的时间。
- 3、那么，有没有一种办法既能列举文件又能在修改文件后只编译修改的部分，未修改的部分不编译呢？答案是有的，这就是 **makefile** 需要解决的问题，**makefile** 使用依赖规则来解决修改文件的问题。
- 4、**makefile** 定义了整个项目的编译、链接等规则，包括：哪些文件需要编译以及如何编译，哪些文件需要先编译，哪些需要后编译，哪些文件需要重新编译，需要创建哪些库文件以及如何创建，如何产生最终的可执行文件等等。编译整个项目需要涉及到的，在 **makefile** 中都可以进行描述。因此，**makefile** 使得项目的编译变得自动化，能使用一行命令完成自动化编译，不需每次都输入一堆源文件和命令，**makefile** 文件是很多编译器维护编译信息的常用工具，他们几乎都会生成 **makefile** 文件。

#### 2.1.2 makefile 基本语法结构

- 1、**makefile** 包含：规则、变量、指示符(directive)、条件语句、函数、注释等，其中规则又包含目标、依赖、命令。
- 2、**makefile** 的基本结构如下



- 3、**makefile** 由一系列规则组成，规则由一个或多个目标、零个或多个依赖以及零个或多个命令组成，依赖也被称为**先决条件**。同一规则中的目标和依赖本文将其统称为**依赖行**。
- 4、一个规则包含了：需要创建的目标，创建目标时所需的依赖关系，创建目标所需的命令，命令描述了怎样从依赖创建目标。所以，一个规则指的是在满足依赖关系的条件下使用命令创建目标，规则的目的是要创建其目标。
- 5、目标和依赖通常是真实存在的文件的名称，当然也可以不是。
- 6、**makefile** 并不关心命令是怎样工作的，**makefile** 只需在目标需要被更新时执行规则的命令即可，也就是说，规则的命令完全可以是一条与目标无关的命令，也可以没有命令，这完全取决于你为规则提供怎样的命令。

### 2.1.3 makefile 基本语法要求

- 1、使用#开头的单行注释，但#不能出现在变量引用或函数调用中。命令中的注释会被传递给执行命令的环境(如 shell, MS-DOS 等)，**makefile** 不解释命令中的注释，因此，不建议在命令中使用注释。另外需注意，对于 **nmake** 若使用中文注释有可能会产生问题，基于各种原因，建立将注释写在单独的一行上，也不要写在规则的命令中。
- 2、**makefile** 是基于行的语法，换行符意味着语句结束。因此，如果一行写不下时，可使用符号“\”续行，续行符之后不能有任何字符，包括空格和注释。
- 3、命令是可以执行的任意命令，一条命令占一行，命令可以以下方式开头
  - 使用 Tab 键开头。这是大多数 **make** 版本都支持的，建议使用此方法。
  - 在依赖行中以分号开头。
  - 使用特殊变量 **RECIPEPREFIX** 自定义开头字符(**nmake** 不支持)。如 **RECIPEPREFIX = >** 表示以字符“>”作为命令的开头。
  - 以空格开头(**nmake** 支持)
- 4、依赖行的第一个目标必须位于一行的开头。使用冒号“:”将目标和依赖隔开，多个目标和多个依赖之间使用空格隔开。**nmake** 的第一个目标前面不能有空格(因为 **nmake** 认为空格是命令的开头)。
- 5、目标和依赖必须位于同一行上。
- 6、目标和依赖的名字是任意的，可以是文件名也可以是一个标签。若目标名称只有一个字符，建议使用空格与冒号隔开，但更建议使用两个字符以上的名字，否则目标名称可能会与之后的冒号一起被识别为盘符，比如 C:可能会被识别为 C 盘。

### 2.1.4 makefile 执行规则的基本原则

**makefile** 执行规则的基本原则如下：

- 1、若目标文件不存在或已过期，则执行命令块中的命令，否则，什么也不做。过期是指目标文件的时间戳早于任意一个依赖的时间戳，或者说目标已过时、目标不是最新的、依赖比目标更新、依赖已被修改或更新。
- 2、但在创建(或重建)目标之前，根据其依赖列表从左到右依次寻找其依赖所在规则，并按以上相同的方法创建该目标的所有依赖，所有的依赖创建完之后才创建目标。
- 3、说简单一点就是，当依赖有修改，则创建目标，但在创建目标之前先按同样的方法从左到右依次创建该目标的所有依赖。

### 示例 2.1：编写一个简单的 makefile

#### ①、准备 makefile 文件

使用记事本编写一个如代码清单 2.1 所示的代码，并将其保存为 `makefile`(注：该文件无后缀)，当然，也可以保存为其他任意名称，使用名称 `makefile` 可以在使用 `make` 工具时不需使用 `-f` 参数指定文件名，本文为了简洁直接保存为名称“`makefile`”

#### #代码清单 2.1 : makefile

```
aa:
 echo "A"
```

#### ②、代码分析

本示例只有一个目标 `aa` 且没有依赖，`echo` 命令类似于 C/C++ 的 `printf()` 函数，用于在控制台显示字符。

#### ③、在 CMD 中转至 makefile 文件所在目录，然后输入以下命令

```
mingw32-make
```

输出

```
echo "A"
"A"
```

其中，第一行是 `makefile` 文件中目标 `aa` 的命令。若不想 `makefile` 中的命令也被输出一遍，可以在代码清单中的 `echo` 命令前加上“`@`”符号。

## 2.2 makefile 目标

- 1、`makefile` 的目标分为：默认目标(最终目标、终极目标)、伪目标、强制目标。
- 2、通常，`make` 工具默认执行的是 `makefile` 文件中第一个规则的第一个目标，这个目标被称为**默认目标、最终目标或终极目标**，但以下情况除外(`nmake` 不适用)
  - 目标名以点“`.`”开始且其后不存在斜线“`/`”。
  - 模式规则的目标模式对默认目标没有影响。
- 3、终极目标
  - 1)、终极目标是 `makefile` 最终需要创建或更新的目标，其他规则是在完成创建终极目标的过程中被连带出来的，若一个目标不是终极目标或其直接、间接依赖，则除非明确指定执行这个规则，否则这个规则不会被执行，相应的规则也不会被创建。

2)、终极目标通常使用一个名称为

all

的伪目标来表示，这是一个潜在的使用习惯。

#### 4、目标执行规则

目标若是真实文件名且无依赖，则被认为总是最新的，因此，当引用该目标时，其命令不会被执行。若想此目标的命令能被执行，则可以把该目标声明为伪目标。

#### 5、伪目标(或称为虚假目标)

1)、伪目标是指不代表真实文件名的目标。使用伪目标的目的通常仅仅是为了执行规则的命令，而不是创建目标文件，因此，也可以把伪目标称为标签。

##### 2)、声明伪目标的方法

可以使用显示声明的方式把一个与真实文件名同名的目标声明为伪目标，其方法是把目标声明为特殊目标

#### .PHONY

的依赖，这样，即使存在与显示声明的伪目标同名的真实文件，该目标仍然是伪目标，当执行该规则时，其命令仍会被执行。而且，显示声明的伪目标不会去查找隐式规则来创建它，这也提高了效率。因此，显示声明伪目标有以下好处

- 不用担心伪目标和真实文件重名而无法执行目标的命令。
- 显示声明的伪目标不会去查找隐式规则来创建它，可提高效率。

#### 3)、伪目标的执行规则

伪目标并不表示一个真正的文件，因此，被认为总是过时的，当执行含有伪目标的规则时，其命令总会被执行。但在执行带有伪目标的命令后，该目标被认为是已更新的，也就是说，伪目标被执行后被认为总是最新的。因此，当伪目标是其他目标的依赖时，由于该目标的依赖总是更新的，所以该目标的命令会被执行。

4)、伪目标不会被继承，也就是说，伪目标的依赖并不会成为伪的。

#### 6、强制目标

强制目标是指没有命令或依赖，且目标不是一个真实文件名的目标，这种目标通常命名为

#### FORCE

其实强制目标就是一个没有命令或依赖的伪目标，由于有些 make 版本不一定支持.PHONY 特殊目标(比如 nmake 就不支持)，所以，很多 make 版本中会使用强制目标。由于强制目标是伪目标，所以强制目标总是过时的，其命令总会被执行。

7、以下代码若 FORCE 不是真实的文件，则二者是等效的。

|          |              |           |
|----------|--------------|-----------|
| aa:FORCE | 若 FORCE 不是文件 | .PHONY:aa |
| @echo a  | ➡            | aa:       |
| FORCE:   |              | @echo a   |

左侧的 FORCE 目标总会被执行，执行后被认为已更新，所以，这时 FORCE 比目标 aa 更新，于是执行目标 aa 的命令，也就是说，目标 aa 总会被执行。右侧的目标 aa 是伪目标，所以目标 aa 总会被执行。



8、表 2-1 列出了目标和依赖各情况下目标是否执行(即目标的命令是否执行)的总结，其中的示例代码省略了命令。

表 2-1 目标执行规则总结

| 示例代码                        | 简要说明          | 命令是否执行   | 命令执行依据                            |
|-----------------------------|---------------|----------|-----------------------------------|
| a.cpp:b.cpp                 | 目标和依赖都是文件     | 若目标过期则执行 | 若依赖被修改(或更新)则认为目标过期                |
| a.cpp:                      | 目标是文件，无依赖     | 命令永不执行   | 目标被认为总是最新的(即永不过期)                 |
| a.cpp:bb                    | 目标是文件，依赖不是文件  | 命令始终执行   | 依赖必是另一规则的伪目标，所以，依赖总是更新的           |
| aa:.....                    | 目标不是文件，有无依赖均可 | 命令始终执行   | 该目标是伪目标，伪目标总是过期的                  |
| .PHONY:a.cpp<br>a.cpp:..... | 目标是文件，有无依赖均可  | 命令始终执行   | 虽然目标 a.cpp 是文件但被显示声明为伪目标，伪目标总是过期的 |

示例 2.2：目标执行规则 1

在 makefile 文件中输入代码清单 2.2 所示代码，其中，假设 a.cpp 是真实存在的文件。

代码清单 2.2：makefile 文件内容

```
#符号@表示不在命令行回显该命令
aa:bb
 @echo a
bb:a.cpp
 @echo b
a.cpp:
 @echo c
```

- ①、分析目标执行顺序
- 根据依赖关系，应按 a.cpp、bb、aa 的顺序创建目标。
- ②、按执行顺序分析各目标是否会被执行
- a.cpp 没有依赖并且是真实文件名，所以，该目标总是最新的，其命令不会被执行；
  - 目标 bb 不是真实文件名，是伪目标，伪目标总是过时的，因此其命令总会执行；
  - 目标 aa 依赖于伪目标 bb，由于伪目标被执行后被认为总是最新的，所以，其命令会被执行。
- ③、使用 make 工具执行 makefile 以验证以上分析
- 在 CMD 中转至 makefile 文件所在目录，然后输入以下命令
- ```
mingw32-make
```
- 以上命令执行的是默认目标 aa。以后若无特别说明，所有示例均是指的执行默认目标。因此输出
- ```
b
a
```
- 接着在 CMD 中输入以下命令
- ```
mingw32-make bb
```
- 以上命令执行指定的目标 bb，因此输出

b

示例 2.3: 目标执行规则 2

代码清单 2.3: makefile 文件内容

```
aa:b.cpp
    @echo a
b.cpp:a.cpp
    @echo b
a.cpp:
    @echo c
```

①、各目标的假设

在代码清单 2.3 中, 假设 aa 不是文件名, a.cpp 和 b.cpp 都是真实的文件, 并且 b.cpp 的时间戳比 a.cpp 的更新

②、分析目标执行顺序

根据依赖关系, 应按 a.cpp、b.cpp、aa 的顺序创建目标。

③、按执行顺序分析各目标是否会被执行

- 由于目标 a.cpp 是真实文件并且没有依赖, 因此该目标总是最新的, 其命令不会执行;
- 由于目标 b.cpp 是真实文件, 并且 b.cpp 比依赖 a.cpp 更新, 所以该目标的命令也不会被执行;
- 最后, 目标 aa 是伪目标, 伪目标总是过期的, 所以执行伪目标的命令
- 最终, 以上代码只会输出字母 a。

④、使用 make 工具执行 makefile 以验证以上分析

在 CMD 中转至 makefile 文件所在目录, 然后输入以下命令

```
mingw32-make b.cpp
```

输出

```
mingw32-make: 'b.cpp' is up to date.
```

以上输出提示 b.cpp 已是最新。

然后在 CMD 输入以下命令

```
mingw32-make
```

以上命令执行的是默认目标, 因此, 输出

a

示例 2.4: 使用 makefile 构建程序

在 makefile 所在目录下新建 a.cpp、b.cpp 二个文件, 并在其中输入代码清单 2.4 和 2.5 所示内容, 然后在 makefile 文件中输入代码清单 2.6 所示内容

代码清单 2.4: a.cpp 文件内容(主文件)

```
#include<iostream>
using namespace std;
void f();
int main(){    f();    return 0;}
```

代码清单 2.5: b.cpp 文件内容

```
#include<iostream>
using namespace std;
void f(){ cout<<"F"<<endl; }
```

代码清单 2.6: makefile 文件内容

```
m.exe:a.o b.o
    g++ a.o b.o -o m.exe
a.o:a.cpp
    g++ -c a.cpp
b.o:b.cpp
    g++ -c b.cpp
```

①、分析各目标

- 代码清单 2.6 中的语句 `g++ -c a.cpp` 会产生一个名称默认为 `a.o` 的文件，`b.o` 同理。由此可见使用 `makefile` 可以执行编译命令以构建 C++ 程序，因此，`makefile` 可用于编译。
- 除 `a.cpp` 和 `b.cpp` 是真实文件外，目标 `a.o`、`b.o`、`m.exe` 在各目标的命令被执行之前都不是真实的文件。

②、分析目标执行顺序

根据依赖关系，应按 `b.o`、`a.o`、`m.exe` 的顺序创建目标。我们可以根据各目标执行顺序，绘制出一个各目标之间依赖关系的依赖树，如图 2.3 所示，通过依赖树能更清楚的了解各目标之间的依赖关系。

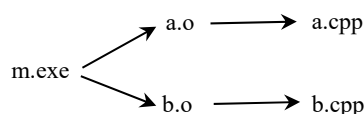


图 2.3 makefile 依赖树

③、按执行顺序分析各目标是否会被执行

- 由于目标 `b.o` 是伪目标，伪目标总是过期的，所以执行伪目标 `b.o` 的命令。
- 由于目标 `a.o` 是伪目标，伪目标总是过期的，所以执行伪目标 `a.o` 的命令。
- `m.exe` 是伪目标，伪目标总是过时的，而且，被执行后的伪目标 `a.o`、`b.o` 总是最新的，所以，执行 `m.exe` 的命令。

④、使用 `make` 工具执行 `makefile` 以验证以上分析

```
G:\qt1>mingw32-make
g++ -c a.cpp
g++ -c b.cpp
g++ a.o b.o -o m.exe

G:\qt1>mingw32-make
mingw32-make: 'm.exe' is up to date.

G:\qt1>m.exe
F
```

图 2.4 未修改文件前执行 `mingw32-make`

```
G:\qt1>mingw32-make
g++ -c b.cpp
g++ a.o b.o -o m.exe

G:\qt1>mingw32-make
mingw32-make: 'm.exe' is up to date.
```

图 2.5 修改 `b.cpp` 后再次执行 `mingw32-make`

在 `cmd` 中转至 `makefile` 所在目录，输入如图 2.4 所示命令，其中，第二次执行 `mingw32-make` 时未执行 `makefile` 文件中的任何命令，这是由于第一次执行后没有文件被改动，每个规则的目标都比依赖更新的缘故。

如果我们修改其中一个文件，比如修改 `b.cpp`，可以在文件中输入一个空格，然后保存，其主要目的是修改文件的时间戳，然后再执行 `mingw32-make` 命令，其结果如图 2.5 所示，由于 `b.cpp` 被更改，时间戳比 `b.o` 更新，所以执行目标 `b.o` 的命令，又由于 `g++ -c b.cpp` 重新生成了 `b.o` 文件，因此 `b.o` 的时间戳比 `m.exe` 更新，所以执行目标 `m.exe` 的命令。

⑤、若需要强制执行 `makefile` 规则的命令，可使用以下命令

```
nmake /A 目标名
```

```
mingw32-make -B 目标名
```

2.3 makefile 规则和 order-only 依赖

1、`makefile` 的规则分为：多目标规则、双冒号规则、隐式规则、模式规则、万能匹配规则等，本小节仅讲解 `makefile` 基本的规则，隐式规则、模式规则、万能匹配规则等更复杂的规则会在后文单独讲解。

2、多目标规则

多目标规则是指一个规则含有多个目标的情形，此时该规则相当于分开的多个规则。不推荐在 `makefile` 中使用多目标规则。比如以下两个规则是等效的，但是当多目标规则作为默认规则时 `nmake` 有一些不同。

aa bb:	等效于	aa:
@echo a	→	@echo a
		bb:
		@echo a

3、多规则目标

多规则目标是指同一目标出现在多个规则中的情形，但只能有一个目标有命令，此时这些目标的依赖可以累加，但是 `nmake` 和 `mingw32-make` 对依赖的累加顺序有些不一样。可以利用此方法向已存在的规则追加依赖文件。比如以下规则是等效的

aa:bb		aa:bb	等效于	aa: bb cc
aa:cc	或者	echo A	→	echo A
echo A		aa:cc		

4、双冒号规则

双冒号规则是指使用双冒号 “`::`” 代替普通单冒号 “`:`” 的规则，双冒号规则可以在多个规则中为同一目标指定不同的命令，若不使用双冒号，则会出现同一目标的规则太多的错误。不允许出现同一目标既是双冒号规则又是普通规则。双冒号规则具有以下不同点：

- 对于没有依赖的双冒号规则，当引用此规则的目标时，会无条件的执行该目标的命令，而无论目标是否是真实的文件名，而普通规则，当目标是真实文件时，此规则的命令不会被执行。
- 多个相同目标的双冒号规则会被独立的处理，而不会合并依赖到一个目标，也就是说，双冒号规则的某一个依赖被更新后，只会执行此规则定义的命令，其他同名目标的双冒号规则不会被执行。

示例 2.5：双冒号规则---指定多个目标

```
aa::cc
```

```

    @echo A
aa::dd
    @echo B
cc:
    @echo C
dd:
    @echo D

```

以上示例将依次输出 CADB。目标 aa 依赖于 cc，于是先执行目标 cc，然后执行第一行的目标 aa；然后执行目标 aa 的第二个依赖 dd 的命令，最后执行第二行的目标 aa

示例 2.6：双冒号规则---更新依赖

```

a.cpp:: b.cpp
    @echo A
a.cpp::c.cpp
    @echo B

```

假设 a.cpp、b.cpp、c.cpp 均是真实文件，若只修改了 b.cpp 将只输出 A，若只修改了 c.cpp 将只输出 B。

示例 2.7：双冒号规则---无条件执行

```

a.cpp::
    @echo A

```

无论目标 a.cpp 是否是真实文件都会被执行。

5、存档成员(nmake 不适用)

- 1)、存档文件是指的把多个文件打包装进一个文件中形成的文件，可以简单的将其理解为压缩文件之类的文件，在 makefile 中的存档文件通常是指的静态库文件(g++为.a 文件，VS 为.lib 文件)，静态库文件是多个中间文件(g++为.o 文件或 VS 为.obj 文件)的集合， g++使用 ar 工具操作静态库文件，VS 使用 lib 工具操作静态库文件。
- 2)、本小节将静态库简称库，静态库中保存的文件称为静态库成员简称库成员，同理，存档文件简称为存档，存档文件的成员简称为存档成员。除了 ar 或 lib 等专门的工具可以直接操作存档成员外，绝大多数工具都不能直接对存档成员进行操作。
- 3)、可以将存档的单个成员作为 makefile 的目标或依赖，其语法如下

archive(member...)

多个成员之间使用空格隔开，以上语法只能在规则的目标和依赖中使用，不能在命令中使用。

示例 2.8：存档成员---单个成员作为目标

```

a.lib(a.obj b.obj):FORCE
    @echo $$
FORCE:

```

以上示例在 CMD 中输入 mingw32-make 将输出 a.obj，输入 mingw32-make b.obj 将输出 b.obj。注：符号\$\$表示当前目标的全名，关于\$\$的讲解详见后文。

6、order-only 依赖(nmake 不适用)

- 1)、order-only 依赖通过在依赖列表中使用符号“|”来指定，在符号右侧的依赖叫做 order-only 依

赖，符号左侧的依赖是常规依赖，`order-only` 依赖的主要特点是不会检查 `order-only` 依赖是否被修改。

- 2)、若同一依赖同时被指定为常规依赖和 `order-only` 依赖，则常规依赖优先，因为，常规依赖是 `order-only` 依赖的超集。

示例 2.9: `order-only` 依赖

```
a.cpp:b.cpp|c.cpp
@echo a
```

假设 `a.cpp`、`b.cpp`、`c.cpp` 均是真实文件，则当 `b.cpp`(常规依赖)被修改后，会执行命令，但是，当 `c.cpp` 被修改后，不会执行命令。

第 3 章 变量(也称为宏)、函数、条件语句

3.1 变量

表 3.1 是本小节将讲解的所有内容

表 3.1 变量总结

	语法	示例	说明
定义变量	变量名 运算符 值	AA=xx	定义一个名为 AA 的变量，并将 xx 赋给 AA
引用变量	\$(变量名) 或 \${变量名}	\$(AA)	获取变量 AA 的值
递归展开变量	=	AA=xx	在引用变量时展开变量
简单展开变量	:=或::=	AA:=xx	在定义变量时展开变量
环境变量	无	\$(TEMP)	获取环境变量 TEMP 的值
目标特有变量	目标...: 变量定义语句	bb:AA=xx	变量 AA 仅在目标 bb 及其依赖可见
向变量中追加值	+=	AA += xx	向变量 AA 追加值 xx(在原值末尾增加一空格与新值分开)
条件赋值运算符	?=	AA?=xx	若变量 AA 之前未被定义，则定义变量 AA
变量值替换	\$(变量名: A=B)	\$(AA:ca=xx)	将变量 AA 中的字符 ca 替换为 xx
override 指示符	override 变量定义语句	override AA=xx	阻止变量 AA 被命令行指定的变量值覆盖
private 指示符	private 变量定义语句	private AA=xx	阻止变量 AA 被继承

1、变量基本语法

- 1)、与 C++等语言一样，makefile 也有变量，makefile 的变量很简单，就是一个变量名后面跟上一个运算符，运算符之后的内容是变量的值。
- 2)、引用变量的语法如下，

\$(变量名) 或 \${变量名}

第二种语法 nmake 不支持。若变量名是单个字符，则可以省略括号。

- 3)、变量名不能包括“：#、=、前置和尾随空白”，变量名是大小写敏感的。若需使用\$符号，需使用两个\$\$来表示。支持的运算符有=、:=、?=、+=，不同运算符对应不同风格的变量。但是 nmake 只支持=运算符。
- 4)、变量的展开仅仅是文本替换，也就是说变量值的字符串原模原样的出现在变量被引用的地方。

2、递归展开变量

- 1)、递归展开变量使用“=”或指令 define 定义，该类型变量在引用时而不是在定义时展开。比如

示例 3.1：递归展开变量

AA=\$(BB)

```
BB=$(CC)
CC=zz
aa:
    @echo $(AA)
```

最后输出 zz，变量 AA 的展开发生在执行 echo \$(AA) 时，在变量 AA 定义之处并未对其展开。替换过程为：首先 \$(AA) 被替换为 \$(BB)，然后 \$(BB) 被替换为 \$(CC)，最后 \$(CC) 被替换为 zz，

2)、递归展开变量的优点是，可以引用之前没有定义的变量，即可以在之后定义或是通过 make 命令行选项传递该变量，比如上例中的 BB 就定义在变量 AA 之后。

3)、递归展开变量的缺点是，

- 可能会陷入死循环，比如如下语句就会陷入死循环

```
AA=$(BB)
BB=$(AA)
```

- 第二个缺点是，若递归展开变量引用函数，则函数需在变量被引用的地方才会调用，由于同一变量可以在多处被引用，所以，这会导致每展开一次变量就要调用一次函数，并且还可能出现不可控或难以预料的错误，因为，无法确定函数在何时会被展开。

3、简单展开变量(nmake 不适用)

1)、简单展开变量使用 “:=” 或 “::=” 定义。简单展开变量对其他变量或函数的引用在定义变量时展开，而不需等到引用时才展开，所以，变量被定义后就是一个实际需要的文本串，其中不再包含任何变量的引用。比如

示例 3.2：简单展开变量

```
AA:=xx
BB:=$(AA)
AA:=yy
aa:
    @echo $(BB)
```

最后输出 xx，变量 AA 的展开发生在定义时，即 BB:=\$(AA) 时。在变量 BB 引用处并未对 AA 展开。替换过程为：首先在定义 BB 处将 \$(AA) 替换为 xx，然后在 @echo \$(BB) 处将 \$(BB) 替换为 xx。

2)、简单展开变量不能实现对其后定义变量的引用。但简单展开变量可以进行重新赋值。

4、环境变量

1)、系统中的所有环境变量对 make 工具是可见的，因此，可在 makefile 中引用任何定义的系统环境变量。使用环境变量很简单，就像引用 makefile 变量一样引用系统环境变量即可。比如，以下代码将输出环境变量 TEMP 的值，TEMP 在系统上通常表示系统临时文件的目录。

```
aa:
    @echo $(TEMP)
```

3)、makefile 中定义的变量或 make 命令形式定义的变量，会覆盖同名的环境变量。可使用 make 命令的 -e 参数阻止覆盖环境变量。

4)、不推荐使用环境变量，因为，环境变量具有全局特征，一旦一个环境变量定义错误，就会对系统

上的所有 makefile 变量产生影响。

5、目标特有变量(target-specific variable) (nmake 不适用)

- 1)、目标特有变量只在该目标的上下文中有有效，对其他目标无效，也就是说，目标特有变量是局部的，因此，可以在不同目标中为一个同名变量定义不同的值，相对目标特有变量来说，makefile 中定义的变量则是全局的。
- 2)、目标特有变量的语法为

目标...: 变量定义语句

目标特有变量可以使用任何有效的赋值运算符，即=、:=、+=、?=，也可以使用 export、unexport、override、private 指示符。定义目标特有变量应在单独一行列出，并且不应有命令。

- 3)、目标特有变量与 makefile 文件中定义的全局变量是两个不同的变量，因此，他们可以使用互不相同的赋值运算符。
- 4)、目标特有变量作用范围为该目标以及该目标的所有依赖。

示例 3.3：目标特有变量

```
AA=zz
all:bb
    @echo all=$(AA)
bb:AA=xx          #在单独一行指定目标特有变量
bb:cc dd
    @echo bb=$(AA)
cc:
dd:ee
    @echo dd=$(AA)
ee:
    @echo ee=$(AA)
```

以上示例依次输出 ee=xx, dd=xx, bb=xx, all=zz。本示例依次执行 ee、dd、cc、bb、all 目标，执行目标 bb 及其依赖时，输出的是目标特有变量 AA=xx 的值，而不是全局变量 AA=zz 的值，只有执行目标 all 时，输出的才是全局变量 AA=zz 的值。

6、向变量中追加值(nmake 不适用)

- 1)、向变量中追加值使用“+=”来实现，+=的作用就像是在变量的初始定义中额外的增加了一个值一样。
- 2)、若该变量之前已经有值，则在原值末尾增加一空格与新值分开。
- 3)、若变量之前未被定义，则“+=”与“=”一样定义一个递归展开变量。
- 4)、若变量之前被定义过，则+=定义的变量取决于最初定义的类型。如果变量之前是使用“:=”或“::=”定义的，则+=将其定义为一个简单展开变量。如果变量之前使用“=”定义，则+=将其定义为一个递归展开变量，比如

AA:=xx 等效于 AA:=xx
AA+=yy 等效于 AA:=\$(AA) yy

BB=xx 大致等效于 BB=xx
AA=\$(BB) 大致等效于 AA=\$(BB) yy
AA+=yy

7、条件赋值运算符(nmake 不适用)

条件赋值运算符“?=”只有在此变量之前未被定义的情况下才会定义该变量。定义为空值的变量仍

然被视为已定义，此时?=将不会定义该变量。比如

AA ?= xx	等效于	ifeq(\$(origin AA), undefined)
	→	AA = xx
		endif

ifeq 条件语句表示若变量 AA 之前未定义则定义 AA=xx，否则不定义变量 AA。origin 函数的作用是返回变量 AA 的一些信息，若 AA 从未定义过则返回 undefined

8、变量值替换

可以使用以下语法替换变量值中的字符

`$(变量名: A=B)`

表示将变量值中的字符 A 替换为 B，nmake 可以替换与 A 相同的所有字符，mingw32-make 只能替换末尾的字符，但 mingw32-make 可以使用通配符%，%表示任意一个字符。替换后不会修改原始变量的值。冒号之前不能有空格，冒号之后的任何空格都会被解释为文字。

示例 3.4：变量值替换

```
AA=cabca cadca adca
all:
    @echo $(AA:ca=xx)
```

nmake 输出 xxbxx xxdxx adxx，mingw32-make 输出 cabxx cadxx adxx

9、override 指示符(nmake 不适用)

通常，make 工具在命令行指定的变量值将替代(即覆盖)在 makefile 文件中定义的同名变量，若想阻止这种情况发生，可以在 makefile 中使用 override 指示符来声明该变量以阻止被覆盖。若变量在定义时使用了 override，则后续对其追加时，也必须使用 override 指示符，否则对此变量值的追加不会生效。

示例 3.5：override 指示符 1

```
override AA=aa
all:
    @echo $(AA)
```

以上示例若在 CMD 中输入 mingw32-make AA=xx，仍会输出 aa

示例 3.6：override 指示符 2

```
override AA+=bb
all:
    @echo $(AA)
```

以上示例，若在 CMD 中输入 mingw32-make AA=xx，则输出 xx bb。

10、private 指示符 (nmake 不适用)

被标记为 private 的变量将不会被继承，只在该变量所在的局部可见，这意味着，标记为 private 的目标特有变量不会被目标的先决条件(即依赖)继承，标记为 private 的全局变量不会被任何目标继承，

即不会在任何命令中可见。`private` 对于目标特有变量更有意义。

示例 3.7: `private` 指示符---阻止变量被继承

```
private MM=zz
bb:private NN=xx
bb:cc
    @echo bb=$(NN)
    @echo all=$(MM)
cc:
    @echo cc=$(NN)
```

以上代码依次输出 `cc=`, `bb=xx`, `all=`, 其中 `cc` 和 `all` 都是空值。

3.2 `define` 指示符和 `call`、`eval` 函数(nmake 不适用)

`define` 指示符(directive)以 `define` 开始, 以 `endef` 结束, `define` 有以下作用

- 用来定义变量。使用 `define` 定义的变量允许包含换行符、空格等字符。
- 用来定义命令包。
- 与 `call` 函数结合使用向 `define` 中传递参数。
- 与 `eval` 函数结合使用, 将其结果使用 `make` 工具来解析。

3.2.1 使用 `define` 定义变量

- 1、使用 `define` 定义变量的方法是, 在与 `define` 同行的后面指定要定义的变量名和一个赋值运算符, 随后的行是该变量的值。
- 2、`define` 定义的变量与普通变量除语法不一样之外, 其工作方式与普通变量一样, 若省略赋值运算符, 则假定为 “=” 并创建一个递归扩展变量, 该变量将在被引用的地方展开; 同理, 若指定 += 运算符, 则该变量与定义普通的 += 变量相同。
- 3、在使用 `define` 指示符定义一个多行变量时, 其定义体会被完整的展开到引用此变量的地方(包含注释), `make` 在引用此变量的地方对所有的定义体进行处理, 决定是注释还是内容。若希望将 `define` 定义体中的所有内容(含空格, 回车符等)原样展开, 请使用 `eval` 函数。
- 4、使用 `define` 可以定义回车换行符, 但在定义换行符时需在 `define` 中输入两个换行符。

示例 3.8: 使用 `define` 定义变量

```
AA=dd
define AA+=          #向原有变量追加值
    bb cc
@echo xx
@echo yy
endef
all:
    @echo $(AA)
```

以上代码将输出

```
dd      bb cc
xx
yy
```

其中 `bb` 的前面是一个制表符。以上代码相当于在 `CMD` 中输入以下命令(包括回车换行符)

```
@echo dd  bb cc 换行符@echo xx 换行符@echo yy
```

即，相当于在 `CMD` 中连续执行了 3 条命令，因此，目标 `all` 等效于以下代码，其中 `&&` 用于在 `CMD` 命令行中联袂两个命令

```
all:
```

```
@echo dd      bb cc&&@echo xx&&@echo yy
```

注意：变量 `AA` 展开后的代码是位于同一行上的，为什么是在同一行上，详见 3.4 小节。

示例 3.9：使用 `define` 定义换行符

#定义一个表示换行符的变量 `BB`，在 `define` 中应输入两个换行符

```
define BB

endif
bb:xxx$(BB) yyy
    @echo A$(BB)@echo C
xxx$(BB):
YYY:
```

以上示例将依次输出

```
A
C
```

以上代码相当于在 `CMD` 中执行 “`@echo A 换行符@echo C`” 命令，即在 `CMD` 中连续执行了两条命令，因此，在 `CMD` 中等效于以下命令

```
@echo A&&@echo C
```

3.2.2 使用 `define` 定义命令包

- 1、命令包把多个命令封装(或打包)在一起，然后通过这个命令包的名字引用这一组命令。
- 2、命令包使用 `define` 指示符定义，其定义方法与使用 `define` 定义变量类似，只是将其中的内容换成了命令。
- 3、由于 `define` 实际上定义的是一个变量，因此命令包的名字不能与其他变量名字相冲突。
- 4、引用 `define` 定义的命令包也只需像引用普通变量一样即可。
- 5、在引用 `define` 定义的变量名之前加上符号 `@`，则该符号会添加到命令包中定义的每一个命令行之中，从而可以使所有命令都不回显。
- 6、在使用 `define` 指示符定义一个命令包时，其定义体会被完整的展开到引用此变量的地方(包含注释)，`make` 在引用此变量的地方对所有的定义体进行处理，决定是注释还是内容。若希望将 `define` 定义体中的所有内容(含空格，回车符等)原样展开，请使用 `eval` 函数。

示例 3.10: 使用 define 定义包

```
define AA
echo xx
echo yy
endef
all:
    @$(AA)
```

以上代码将依次输出 xx yy，以上代码使用 define 定义的变量(本例是一个命令包)是递归展开变量，因此，需在该命令包被引用时才会展开。目标 all 等效于以下代码

```
all:
    @echo xx&&@echo yy
```

3.2.3 使用 call 函数向 define 中传递参数

1、call 函数的语法为

`$(call 变量名,参数 1,参数 2,...)`

- 2、call 函数对参数的数量没有限制。但无参数的 call 没有任何意义。当调用 call 函数时，在将参数赋值给临时变量之前展开参数，然后，参数 1，参数 2...会依次赋值给临时变量\$(1)、\$(2)....，而\$(0)代表变量本身。call 函数返回依次替换\$(1)、\$(2)...之后变量名所定义的表达式的计算值。
- 3、需要注意的是，第一个参数是指的变量名，而不是对变量的引用，并且这个变量名所代表的变量只能被定义为递归展开变量。
- 4、如果变量名是内置的函数的名称，则会调用该内置函数，即使还存在同名的其他变量。

示例 3.11: call 函数的用法

```
AA=aa $(1) $(2) bb
BB= $(call AA,xx,yy,zz)
all:
    @echo $(BB)
```

以上示例输出 aa xx yy bb。以上代码在@echo \$(BB)处展开 BB，调用 call 函数将 xx 传递给临时变量\$(1)、yy 传递给\$(2)、zz 传递给\$(3)，然后 call 函数返回变量 AA 定义的表达式的值，此时的值为 aa xx yy bb，然后将该值赋值给变量 BB，最终输出 aa xx yy bb。

示例 3.12: 将 call 函数的参数传递给 define 定义的变量

```
define AA
aa $(1) $(2) bb
endef
BB= $(call AA,xx,yy)
all:
    @echo $(BB)
```

以上示例输出 aa xx yy bb。本示例与示例 3.10 等效。

3.2.4 将 define 与 eval 函数联合使用

- 1、eval 函数的主要作用是将参数展开为 makefile 语法，然后在 make 工具解析时再次展开，并使用 make 工具解析该语法，因为要展开两次，因此，若需要使用符号\$则应使用\$\$。eval 函数的关键在于，第一次展开后的代码会使用 make 工具解析，而且只会解析展开的这部分代码，并且是按照 makefile 语法的规则来解析的。

示例 3.13: eval 函数的用法 1

```
define XX
aa bb:
    @echo bb $$@
endif
$(eval $(XX))
```

以上代码输出 bb aa。在调用\$(eval \$(XX))处展开参数\$(XX)，成为(以下代码是示例 3.13 的等效代码)

```
aa bb:
    @echo bb $@
```

然后再使用 make 工具解析以上内容，注意，此时是按 makefile 的语法来解析以上内容的。以上内容是一个完整的 makefile 规则，所以，按 makefile 的语法规则，最后输出 bb aa。注：\$@表示当前目标的名称。

- 2、从示例 3.13 可以看到，若将 XX 实现为执行某一功能，比如，生成.o 对象文件(或其他更复杂的功能)，则此时就可以把 XX 当作创建.o 文件的一个模板，同时，还可以配合 call 函数向 define 里面传递参数，这样模板就更完美了，在以后需要创建.o 文件时，就可以直接使用 eval 函数执行此模板非常便利的创建.o 文件。

示例 3.14: eval 函数的用法 2

```
define XX
aa:
endif
$(eval $(XX))
aa bb:
    @echo bb $@
```

以上代码中的\$(eval \$(XX))，会被展形为“aa:”，然后使用 make 工具解析此内容，即更新目标 aa，由于目标 aa 有两个规则，因此，执行同名的第二个规则的命令，输出 bb aa

示例 3.15: 使用 eval 函数与直接引用变量的区别

```
define XX
    @echo ss tt
endif
aa bb:
    $(eval $(XX))
```

以上代码将产生错误，因为在调用 eval 函数时，第一次将\$(XX)展开为

```
@echo ss tt
```

接着会使用 `make` 工具按 `makefile` 语法规则解析以上语句，该语句明显没有目标，所以产生错误。若将以上代码的最后一行使用 `$(XX)` 代替，即

```
aa bb:
    $(XX)
```

则代码能成功执行，输出 `ss tt`。

3.3 自动变量、通配符和部分默认变量

3.3.1 自动变量

- 1、`makefile` 定义了一些预定义变量，这些变量被称为自动变量，如表 3.2 所示。使用表 3.2 中的变量能进一步减短代码输入量，并且能自动判断目标和依赖名，比如

`m.exe:a.o b.o`
`g++ a.o b.o -o m.exe`

 等效于
 \longrightarrow
`m.exe:a.o b.o`
`g++ $^ -o $@`

- 2、需要注意的是，自动变量是一个变量，这意味着可以像引用普通变量那样引用自动变量，比如 `$@` 可以写成 `$(@)`，也就是说自动变量 `$@` 的变量名是 `@`，其他自动变量同理，从略。

表 3.2 `makefile` 定义的部分预定义变量

变量名	mingw32-make 意义	nmake 意义
<code>\$@</code>	当前目标的全名。若目标是存档文件，则表示该文档的文件名	当前目标的全名
<code>\$<</code>	规则第一个依赖的名称。	比当前目标更新的所有依赖，仅在推理规则的命令中有效
<code>\$^</code>	当前目标的所有依赖，不含重复的依赖，也不包含 <code>order-only</code> 依赖。若依赖是静态库文件，则 <code>\$^</code> 表示库成员名。	不支持
<code>\$+</code>	类似 <code>\$^</code> ，但保留了重复的依赖。	不支持
<code>\$?</code>	比当前目标更新的所有依赖，不含重复依赖。若依赖是存档文件成员，则 <code>\$^</code> 表示存档成员名。	同 <code>mingw32-make</code> ，但含重复依赖
<code>\$%</code>	若目标是存档文件的成员时，则代表存档文件的成员名，否则为空。比如，若目标是 <code>bb.a(c.o)</code> ，则 <code>\$%</code> 为 <code>c.o</code>	不支持
<code>\$ </code>	所有 <code>order-only</code> 依赖的名称	不支持
<code>\$*</code>	目标模式中 <code>%</code> 所代表的词干(stem)部分(含目录部分)，比如，若文件是 <code>xx/a.ff.b</code> ，若目标模式为 <code>a.%b</code> ，则 <code>\$*</code> 的值为 <code>xx/ff</code> 。在显示规则中不适用 <code>\$*</code> ，但若显示规则的目标带有一个可识别的后缀(即 <code>SUFFIXES</code> 包含的后缀)，则 <code>\$*</code> 表示除后缀以外的部分，比如 <code>xxx.cpp</code> ，则 <code>\$*</code> 为 <code>xxx</code> 。建议 <code>\$*</code> 只用于模式中。	当前目标全名，但不含后缀名
<code>**</code>	不支持	当前目标所有依赖，含重复的依赖
增加 D	在自动变量中增加 D 可以表示相应目标或依赖的目录部分(不	同 <code>mingw32-make</code>

或 F	含尾部斜杠), 或者增加 F 可以表示文件名部分, 比如, 若目标是 a/b/c.o, 则\$(@D)表示 a/b, \$(@F)表示 c.o, 同理其他自动变量也有 D 或 F 版本的变体(\$ 除外), 如\$(^D)、\$(^F)、\$(+D)、\$(+F)等等。	
-----	--	--

3.3.2 通配符

1、makefile 中的通配符有:

- *表示任意一个或多个字符, 比如*.o, 表示所有以.o 为后缀的文件。
- ?表示任意一个字符。比如, a?b.o, 表示所有以 a 开始, 后跟任意一个字符, 以 b.o 结尾的名称为 5 个字符的文件
- [...]表示方括号中的其中一个字符(nmake 不适用)。比如 a[xyz]b 表示 axb, ayb, azb。

2、通配符的展开规则如下:

- 在目标和依赖中的通配符会在 make 读取 makefile 时自动展开。
- 命令中的通配符在系统的命令环境(如 shell、MS-DOS 等)执行此命令时展开, 这意味着, 对命令中通配符的解释不属于 makefile 的范围。
- 其他地方的通配符只有在使用 wildcard 函数显示请求时才会展开。

3、并不是在 makefile 中的任何地方都适合使用通配符, 通配符只能用在规则的目标、依赖和命令中, 比如定义变量时的通配符就不会被处理, 比如定义变量 AA=*.o, 则 AA 并不能表示所有的以空格分开的.o 文件列表, AA 的值仅仅是 “*.o”, 要在其他地方使用通配符, 需使用 wildcard 函数, 比如 AA=\$(wildcar *.o)。

4、若规则的文件名包含通配符的字符(如*、?等), 在使用这个文件名时, 可使用反斜杠 “\” 进行转义, 如 a*b 表示 a*b。由于 “\” 表示转义字符, 所以, 如果是在 windows 这样的系统中指定文件路径时, 应使用正斜杠 “/”, 比如 aa:a\b*.o 应写为 aa:a/b/*.o。

5、若相应目录下存在有通配符指定的文件则不会有什么问题, 但是若把满足通配符条件的文件全部删除则会出现问题, 比如 aa:*.o, 若把相应目录下的所有.o 文件都删除, 则依赖的名称将是 “*.o”, 这可能不是我们所期望的结果(因为通常不会存在这样的文件), 这时可以考虑使用 wildcar 函数。

6、wildcar 函数的语法如下(nmake 不适用):

`$(wildcard 模式...)`

该函数的作用是, 将所有模式展开为使用空格分开的、与任一模式匹配的、已经存在的文件名列表, 若不存在这样的文件, 则返回空。该函数主要有两个重要的作用, 一个是在调用该函数的地方就会将通配符展开, 第二就是会返回空值, 而不是含通配符的原样字符串。

3.3.3 部分默认变量

makefile 定义了一些默认变量, 由于各 make 工具支持的情形不统一, 表 3.3 列出了其中几个, 变量的值可以使用 “echo \$(变量名)” 命令查看。

表 3.3 makefile 定义的部分默认变量

变量名	mingw32-make 对应值	nmake 对应值	说明
AS	as	ml 或 ml64	汇编器
RC	不支持	rc	资源编译器
CC	cc	cl	这三个变量表示的是相应语言编译器的名称
CPP	cc -E	cl	
CXX	g++	cl	
MAKE	MAKE 变量返回的值是当前使用的 make 工具的名称，若调用的是 nmake 则返回的值是 nmake.exe 的完整路径，mingw32-make 则不含路径		

3.4 变量和函数的展开过程(nmake 不适用)

1、make 工具的执行过程分为两个阶段

- 第一阶段读取所有的 makefile 文件，并内建所有的变量及其值、隐式和显示规则，并构建所有的目标及依赖关系图。
- 第二阶段，根据第一阶段的依赖关系图决定哪些目标应更新，并使用对应的规则来重建这些目标。

2、以上两个工作阶段直接影响到变量和函数的展开(expansion)时机。

- 1)、若展开发生在第一阶段，则称为立即(immediate)扩展的或立即展开的，此时，所有的变量和函数被展开在需要构建的依赖关系图的对应规则中。
- 2)、若展开不是立即的，则称为延期的(deferred)或延迟的，延期的构造部分的展开被延迟，直到该展开被使用时，或当他在一个直接的上下文中被引用时，或在第二阶段需要他时。

3、以下是变量的展开时机，以 immediate = deferred 为例，表示“=”左侧是立即展开的，“=”右侧是延迟展开的。

```

immediate = deferred
immediate ?= deferred
immediate := immediate
immediate += deferred or immediate
define immediate [= | ?= | := | +=]
    deferred or immediate
endef

```

4、以下是规则的展开时机，规则的定义总是以相同方式展开，目标和依赖会立即展开，命令总是延迟的。

```

immediate: immediate; deferred
    deferred

```

5、条件语句参数中的变量在 make 读取 makefile 时展开(即立即展开)并进行比较，这意味着，不能在参数中使用自动变量，因为自动变量在执行命令之前还未定义。

6、GNU make 在解析 makefile 文件时是以“行”为单位解析的，步骤如下：

- 1) 读入完整的一行，包插反斜杠转义的行。

- 2) 删除注释。
- 3) 若该行以 TAB 开始并处于规则上下文中，则将该行添加为当前规则的命令，并读取下一行。
- 4) 展开立即展开上下文中的元素。
- 5) 查找分隔符，如 “:” 或 “=” 等，以判断该行是变量还是规则。
- 6) 内建结果操作并读取下一行。

示例 3.16: 变量展开过程 1

```
AA=bb:;@echo A
$(AA)           #展开变量后等效于“bb:; @echo A”
```

以上示例输出 A。第一行代码并不会被判断为是一个名称为 “AA=b” 的目标，而会根据首先读取到的 “=” 将名称 AA 解释变量名，所以，第一行定义了一个变量 AA，该变量的值为 “bb:;@echo A”。第二行展开变量 AA，于是输出 A

示例 3.17: 变量展开过程 2

```
define AA
bb:
    @echo XXY
    @echo YYY
endif
$(AA)
```

以上示例将产生错误，因为 bb 的依赖未创建。展开变量 AA 后，其内容会被认为位于同一行，展开后等效于

bb:@echo XXX 换行符@echo YYY”

虽然其中包含换行符，但仍被认为在同一行，所以展开的结果表示 bb 含有依赖 “@echo”、“XXX 换行符”、“@echo”、“YYY”。因此，以上示例与以下代码等效

```
#定义一个表示换行符的变量 BB
define BB

endif
bb:@echo XXX$(BB)@echo YYY
```

示例 3.18: 变量展开过程 3

```
define AA
bb: ;
@echo XXX
@echo YYY
endif
$(AA)
```

以上示例依次输出 XXX，YYY，变量 AA 被展开后仍然位于同一行，展开后等效于

bb:;@echo XXX 换行符@echo YYY

因此，以上示例与以下代码等效

```
#定义一个表示换行符的变量 BB
define BB

endif
bb: ; @echo XXX$(BB)@echo YYY
```

若需要将 `define` 中的内容原本不变的展开，请使用 `eval` 函数。

3.5 makefile 中的函数和条件语句

与 C/C++ 等语言一样，`makefile` 也有函数和条件语句，由于各版本实现有差异，本文仅对函数和条件语句的语法结构作一简介。

3.5.1 makefile 函数

1、调用函数的语法格式为：

```
$(函数名 参数列表 1, 参数列表 2, 参数列表 3...)
```

函数使用类似引用变量的方式来调用，并且使用与引用变量相同的规则来展开。函数也可使用大括号括起来，建议使用圆括号，这样更容易与其他版本的代码兼容。参数列表之间使用逗号隔开，根据函数的功能不同，参数列表中若有多个参数则使用空格隔开。函数的参数在调用函数之前被展开。

2、下面列举两个内置函数以示说明

1)、`addprefix` 函数(不是 `nmake` 函数)用于给子串加上前缀，其语法形式为：

```
$(addprefix pre,name1 name2...)
```

其中 `addprefix` 是函数名，之后是参数列表，该函数的第一个参数列表只能有一个参数，但第二个参数列表允许有多个参数。比如

```
XXX=$(addprefix obj/, a b c)
```

```
all:
```

```
@echo $(XXX)
```

以上代码将输出：`obj/a obj/b obj/c`

2)、`join` 函数(不是 `nmake` 函数)用于联接两个字符串，其语法为：

```
$(join a1 a2...,b1 b2...),
```

该函数共有两个参数列表，每个列表中都允许有多个参数，其功能是把 `b1` 连接到 `a1` 后，`b2` 连接到 `a2` 后，以此类推，比如

```
XXX=$(join a1 obj/ a3,b1 b2)
```

```
all:
```

```
@echo $(XXX)
```

以上代码将输出：`a1b1 obj/b2 a3`

3、更多的内置函数请参阅 GNU make 和 VS 的相关文档。

3.5.2 makefile 条件语句

1、makefile 有以下 4 个条件指示符(nmake 不支持):

- ifeq: 是否相等。相等为 true, 否则为 false
- ifneq: 是否不相等。不相等为 true, 否则为 false
- ifdef: 变量是否有值。有值为 true, 否则为 false
- ifndef: 变量是否无值。有值为 true, 否则为 false

2、nmake 有以下的预处理指令(mingw32-make 不支持):

- !IF 常量表达式
可使用 c++之类的比较运算符, 如相等==, 不等!=, 大于>等, 若要比较字符串, 需用双引号括起来
- !IFDEF 变量名
变量名是否有值, 有值为 true, 否则为 false

3、下面是 ifeq 和!IF 的语法结构

!IF 常量表达式	ifeq (参数 1,参数 2)	ifeq "参数 1" "参数 2"	ifeq '参数 1' '参数 2'
.....
!ELSE	else	else	else
.....
!ENDIF	endif	endif	endif

语法要求如下:

- 其中 else 或!ELSE 语句部分可以没有。
- ifeq 表示展开参数 1 和参数 2, 然后进行比较, 若相等则为 true, 执行之后的语句, 否则执行 else 之后的语句。!IF 类似, 只不过是判断的常量表达式的比较结果。
- 对 ifeq 而言, 条件判断的处理是文本级别的, 这意味着不能进行类似于 3>2 的比较, 因为 3>2 是一个文本, 但!IF 可以进行比较。
- ifeq 的参数 1 和参数 2 还可以混用单引号与双引号, 比如, 参数 1 用单引号, 参数 2 用双引号等等。
- 特别注意, 若 ifeq 之后使用括号, 则需在 ifeq 之后使用一个空格与之后的括号隔开。

4、下面是 ifdef 和!IFDEF 的语法结构, 仅列出条件头部分, 之后的部分分别与 ifeq 和!IF 相同

```
ifdef 变量名
!IFDEF 变量名
```

其作用是, 若变量的值非空, 则为 true, 否则为 false。ifdef 只判断变量是否有值, 除了“xxx=”这种定义情况外, 其他任何方式的定义都会被 ifdef 认为是有值的, 也就是说, 即使把一个无值的变量赋值给别一个变量, 则之后的这个变量也会认为是有值的, 要测试空值, 可以使用 ifeq (\$(xxx),)。!IFDEF 与 ifdef 不同, 空值被!IFDEF 认为是被定义过的。

5、其余条件语句的语法与 ifeq 类似, 从略。

6、条件语句参数中的变量在 make 读取 makefile 时展开(即立即展开)并进行比较, 这意味着, 不能在参数中使用自动变量, 因为自动变量在执行命令之前还未定义。

示例 3.19: ifeq 位于目标之外(mingw32-mak)

```
ifeq (a,a)
XXX=A
endif
all:
    @echo $(XXX)
```

示例 3.20: !IF 位于目标之外(nmak)

```
!IF "aa"=="aa"
XXX=A
!ENDIF
all:
    @echo $(XXX)
```

以上两个示例的输出结果都为：A。需要注意的是，因为 ifeq 未在目标之中，所以，if 中的语句不能是 echo 之类的命令语句。nmake 的变量名 XXX 之前不能有空格，否则会被当作命令处理。

示例 3.21: ifeq 位于目标之内(mingw32-mak)

```
all:
ifeq (a,a)
    @echo A
endif
```

示例 3.22: !IF 位于目标之内(nmak)

```
all:
!IF 3>2
    @echo A
!ENDIF
```

以上两个示例的输出结果都为：A。由于将 ifeq 或!IF 语句放于目标 all 之中，因此，应在条件语句中使用命令。

示例 3.23: 测试变量是否有值 1(mingw32-mak)

```
#本示例输出 false
AAA=
all:
ifdef AAA
    @echo true
else
    @echo false
endif
```

示例 3.24: 测试变量是否有值 2(mingw32-mak)

```
#本示例输出 true
AAA=
BBB=$(AAA)    #即使变量 AAA 是空值，变量 BBB 也被认为是有值的
all:
ifdef BBB
    @echo true
```

```
else
    @echo false
endif
```

示例 3.25: ifeq 参数中的自动变量(mingw32-mak)

#本示例输出 false

```
all:
ifeq (all,$@)
    @echo true
else
    @echo false
endif
```

ifeq 的参数\$@是一个自动变量，自动变量在执行命令之前还未定义，所以 all 与\$@不相等，结果为假。

第 4 章 模式规则

本文将本章所讲解的静态模式规则、模式规则、隐式模式规则、万能模式规则等统称为模式规则。

4.1 静态模式规则(nmake 不适用)

- 1、静态模式规则(简称静态模式)的语法为(省略命令行):

目标...:目标模式:依赖模式...

静态模式规则的目标模式、依赖模式通常会包含模式字符%并且只会包含一次，该字符的作用与*相同，都表示任意一个或多个字符。一个规则由目标、依赖、命令三者组成，由语法可见，静态模式规则没有依赖的名称，所以**静态模式规则的关键在于怎样确定依赖的名称**。

- 2、静态模式规则的依赖由目标、目标模式、依赖模式三者共同决定，其方法是，将目标与目标模式匹配，并提取目标名称中的一部分字符串，这部分字符串被称为词干(stem)，然后使用该词干替换掉依赖模式中相应部分，以生成该目标的依赖的名称。

- 3、生成依赖名称的具体方法如下：

将目标模式中除%之外的字符串与目标名称精确匹配，再将目标名称中剩余的字符串与%匹配，这部分字符串被称为词干(stem)，然后使用该词干替换依赖模式中的模式字符%，从而得到一个新字符串，这个新字符串就是目标的依赖名称。比如

```
abc.o abc.lib abc.out : %.o :%.cpp
```

以上代码共有 3 个目标 abc.o、abc.lib、abc.out，目标模式为%.o，依赖模式为%.cpp，其中目标 abc.o 与%.o 符合匹配要求，其余两个目标不符合除%之外的精确匹配(比如，.out≠.o)，所以，词干为 abc，然后使用 abc 替换依赖模式中的%，得到目标 abc.o 的依赖为 abc.cpp。图 4.1 是静态模式规则的匹配过程

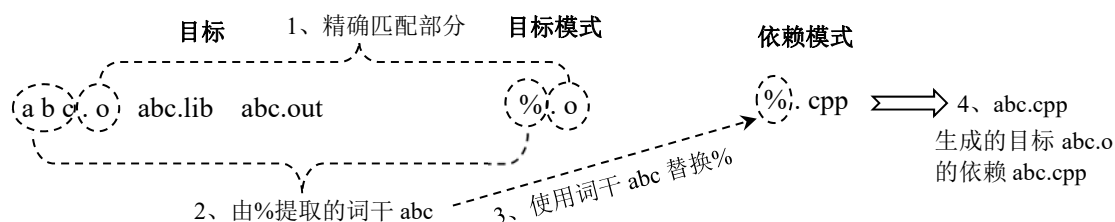


图 4.1 静态模式规则的依赖的生成步骤

- 4、在静态模式规则中可以指定多个目标，并由各自目标的名称根据目标模式构造出各自的依赖名称，可见，静态模式规则可以为多目标规则中的各目标构造出各自不同但相似的依赖，这是与常规的多目标规则都具有相同的依赖不相同的地方，所以，静态模式规则比常规的多目标规则更通用。

示例 4.1：静态模式规则

```
aa.o bb.o: %.o: %.d %.e
```

```

    @echo $@=$^
aa.d:
bb.d:
aa.e:
bb.e:

```

对于以上代码，输入命令 `wingw32-make aa.o bb.o`，会依次输出 `aa.o=aa.d aa.e`，`bb.o=bb.d bb.e`

5、语法要求

目标模式只能有一个，并且必须含有%，依赖模式可以有多个，可以不含有%，指定的每一个目标必须与目标模式匹配，否则执行 `make` 时将会得到一个警告。

示例 4.2：静态模式规则基本语法要求

```

abc.o abc.c abc.out : %.o : %.cpp %.ddd
    @echo $@=$^
abc.cpp:
abc.ddd:

```

对于以上代码，输入命令 `wingw32-make`，会依次输出 `abc.o=abc.cpp abc.ddd`，同时会显示 `abc.c` 和 `abc.out` 没有匹配的目标模式的警告。

- 静态模式规则中的%字符可通过反斜杠“\”进行转义，引用%的反斜杠可以由更多的反斜杠引用。引用%字符的反斜杠将在与文件名进行比较或替换词干之前从模式中删除，没有引用%字符的反斜杠不会受到影响。比如 `aa\%bb\%cc\`，将是 `aa%bb\%cc\`，最后两个反斜杠由于没有引用%，所以不受影响。

4.2 模式规则(nmake 不适用)

4.2.1 模式规则匹配原理

1、依赖文件不存在或缺失依赖文件或依赖缺失

依赖缺失是指目标的依赖所对应的真实文件不存在，并且该依赖不是另一个有命令的规则的目标，这种情况本文称为目标的依赖文件不存在，或目标缺失依赖文件，或目标的依赖缺失，或目标缺失依赖，比如

```

aa:bb
    @echo A
bb:

```

若 `bb` 对应的真实文件不存在，则表示依赖文件 `bb` 不存在，因此，目标 `aa` 的依赖缺失。再如

```

aa:bb
    @echo A
bb:
    @echo B

```


若 **bb** 对应的真实文件不存在，但由于 **bb** 是另一个有命令规则的目标，所以依赖文件 **bb** 是存在的，目标 **aa** 的依赖不是缺失的。

2、模式规则基本原理

模式规则不会主动执行，其作用是，当执行一个规则时，若该规则的目标的依赖缺失，则试图使用相应的模式规则的命令来创建该目标缺失的依赖。

3、模式规则的语法格式为(省略命令行)

目标模式.....依赖模式.....

语法要求如下：

- 目标模式必须是一个带有模式字符“%”的名称，其中%可匹配任何非空字符。
- 依赖模式可以包含%也可以不包含%，若依赖模式不包含%，则表示该依赖是所有符合目标模式的目标的依赖，比如%.x:aa.b，表示aa.b是所有名称以x为后缀的目标的依赖。
- 目标模式和依赖模式都可以是一个或多个。

4、模式规则的目标无目录(即不含有斜杠)时的匹配原理如下(与静态模式规则类似)：

模式规则的关键在于确定依赖的名称。若目标的依赖文件不存在，则搜索一个合适的模式规则，并将该缺失的依赖文件作为**模式规则的目标**，本文将该目标称为**传递目标**，然后依据模式规则的目标(即传递目标)、目标模式、依赖模式三者共同决定传递目标的依赖名称，此时的规则与静态模式规则相同(见图 4.1)，只是将目标更换为传递目标了，即，将目标模式中除%之外的字符串与传递目标名称精确匹配，再将传递目标名称中剩余的字符串与%匹配，这部分字符串被称为词干(stem)，然后使用该词干替换依赖模式中的模式字符%，从而得到一个新字符串，这个新字符串就是传递目标的依赖。图 4.2 说明了模式规则的匹配过程。注意：模式规则中的“目标模式”并不是真正的目标，也就是说模式规则在接收到传递目标之前是没有真正的目标存在的。

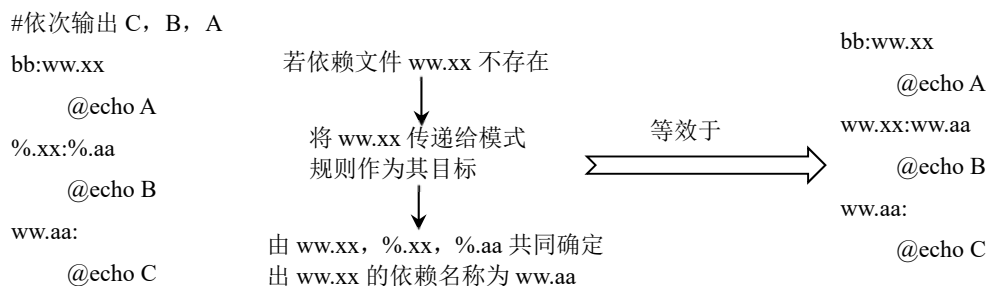


图 4.2 模式规则匹配原理(无目录)

5、模式规则的目标包含目录(即含有斜杠)时的匹配原理如下：

- 形成词干：在进行匹配时，将传递目标名称中包含的目录字符串在匹配之前移除，只进行基本文件名的匹配，匹配成功后，再将目录部分加入到匹配之后的字符串之前形成词干。
- 生成依赖名称：使用词干中的非目录部分替换依赖名称中的“%”形成一个新名称，然后再将目录部分加入到该新名称之前形成依赖名称的全路径名。下面以示例进行说明。

示例 4.3：模式规则的匹配原理(目标包含目录)

```

aa:dd/ee/aww.xx          #若依赖文件 dd/ee/aww.xx 不存在
    @echo A= $^
  
```

```

a%.xx:%.aa          #将 dd/ee/aww.xx 去掉目录后再进行匹配，最终词干为 dd/ee/ww
    @echo B= $*
dd/ee/ww.aa:        #最终确定出依赖名称为 dd/ee/ww.aa
    @echo D

```

以上示例将依次输出 D，B=dd/ee/ww，A=dd/ee/aww.xx，本示例还展示了自动变量\$*的作用(输出模式匹配中的词干)。将依赖文件 dd/ee/aww.xx 作为传递目标，并去掉目录部分后，即，使用 aww.xx 与目标模式 a%.xx 进行匹配，得到词干 ww，然后再将目录部分加入该词干，得到新的词干 dd/ee/ww，然后再由新的词干 dd/ee/ww 与依赖模式%.aa 匹配，得到依赖名称 dd/ee/ww.aa。

4.2.2 模式规则的启动

- 1、除了缺失依赖文件会导致模式规则的执行外，没有命令的规则、没有命令的双冒号规则也会导致搜索一个相应的模式规则来执行。除此之外，还可以通过命令行指定执行的目标来向模式规则传递一个目标，从而导致相应的模式规则被执行。下面以示例进行说明。

示例 4.4：启动模式规则的情形

```

ww.xx: :
%.xx:%.aa
    @echo B
ww.aa:
    @echo C
bb.aa:

```

若在 CMD 中输入 wingw32-make 将依次输出 C、B，这是按传统的模式规则原理执行的以上代码。若输入 wingw32-make bb.xx，则会输出 B，此时会将 bb.xx 作为传递目标来启动模式规则，最终生成依赖名称 bb.aa，但目标 bb.aa 无命令执行，所以最终输出 B。

- 2、通过以上讲解可知，模式规则不会主动启动，因为在匹配之前模式规则并没有真正的目标存在。可通过以下三种方法来启动模式规则
 - 依赖文件不存在(缺失依赖文件)
 - 没有命令的规则和没有命令的双冒号规则
 - 在命令行的 make 工具中明确指定一个相应的目标

4.2.3 搜索适用的模式规则及多目标模式模式规则

- 1、通常一个 makefile 会有多个模式规则存在，若模式规则被启动，这会导致对所有模式规则的搜索，并进行选择，最终将选择一个适用的模式规则。选择适用的模式规则需经过这几个步骤：启动模式规则，搜索所有模式规则，选择出与之匹配的模式规则，选择出适用的模式规则。
- 2、选择适用的模式规则的原理
 - 适用的模式规则是指的经过计算后的模式规则的依赖名称所对应的文件真实存在或该依赖可被创建(can be made)。可被创建是指，该依赖在 makefile 文件中被明确的作为目标或依赖提及，或者可通过递归的找到一个隐式归则来创建该依赖。比如

```

aa:bb      #判断依赖 bb 是否可被创建
bb:        #由于 bb 在该规则被明确的作为目标提及，所以 bb 可被创建。

```

- 若有多个适用的模式规则，则选择具有最短词干的模式规则(即最佳匹配模式规则)，若有多个模式规则具有最短词干，则选择最先出现在 `makefile` 文件中的模式规则。
- 除以上规则外，还应注意规则匹配的优先级问题，比如无需使用隐式规则的模式规则(比如，无依赖或依赖文件存在)始终优于需要使用隐式规则来创建依赖的模式规则

3、模式规则的搜索算法详见后文。下面列举两个简单的示例作为说明。

示例 4.5：选择适用的模式规则 1

```
bb:ww.xx
    @echo A
%.xx:%.aa
    @echo B
%.xx:%.bb
    @echo C
%.yy:%.cc      #与 ww.xx 不匹配
    @echo D
ww.aa:
```

以上示例中的所有依赖和目标的对应文件均假设不存在，由于依赖 `ww.xx` 不存在，所以会启动模式规则，于是搜索所有模式规则，选择出匹配的两个模式规则 `%.xx:%.aa` 和 `%xx%.bb`，通过计算后确定出这两个模式规则的依赖文件名分别为 `ww.aa` 和 `ww.bb`，由于 `ww.aa` 在代码中被明确作为规则“`ww.aa:`”的目标，所以，选择模式规则 `%.xx:%.aa` 为合适的模式规则，最终依次输出 B，A

示例 4.6：选择适用的模式规则 2

```
bb:abc.xx
    @echo A
%.xx:%.aa      #词干 abc
    @echo B
%.xx:%.bb      #词干 abc
    @echo C
a%.xx:%.cc     #词干 bc
    @echo D
bc.cc:
abc.aa:
abc.bb:
```

以上示例中的所有依赖和目标的对应文件均假设不存在。三个模式规则均与 `abc.xx` 匹配，但由于模式 `a%.xx:%.cc` 的词干最短，所以最终选择该模式规则，最终依次输出 D，A

4、现在来看看多目标模式模式规则。普通多目标规则的多个目标对应多个独立规则，这些规则各自有自己的命令行，但对于多目标模式模式规则来说，所有规则的目标模式共同拥有依赖和命令行，因此，一次命令执行后，规则不会再去检查是否需要重建符合它的模式的目标。下面以示例进行说明。

示例 4.7：多目标模式模式规则

```
%.xx %.yy:aa
    @echo A
aa:
```

若在 CMD 中输入

```
mingw32-make a.xx a.yy
```

将只输出一个 A 和目标 a.yy 什么也没做的提示。第一次使用 a.xx 与目标模式%.xx 匹配成功并执行命令后，不会再次检查 a.yy 是否与目标模式%.yy 匹配。因此，以上代码等效于：

```
a.xx:aa
    @echo A
a.yy:aa
aa:
```

4.3 隐式模式规则(implicit pattern rules) (nmake 不适用)

- 1、隐式模式规则是指 makefile 内置的模式规则，简称隐式规则，因此，隐式规则和模式规则的启动方式类似，当依赖文件不存在或规则没有命令时会启动隐式规则，但隐式规则不能在命令行明确指定一个目标来启动。
- 2、隐式规则可以被重新定义，只需使用与隐式规则具有相同的目标模式和依赖模式的模式规则就行了，新定义的规则在隐式规则序列中的位置由编写新规则所在的位置决定。若没有指定命令，则表示取消该隐式规则。
- 3、这里有必要对编译过程中，各文件之间的生成关系作一介绍，如下：
 - 1) 由.cpp 生成.i 或.ii 文件，命令类似：g++ -E a.cpp -o a.i
 - 2) 由.i 文件生成.s 文件(VS 为.asm 文件)，命令类似：g++ -S a.i
 - 3) 由.s 文件生成.o 文件(VS 为.obj 文件)，即中间文件或对象文件，命令类似：g++ -c a.s
 - 4) 由.o 文件生成.exe 文件，命令类似：g++ a.o
 - 5) 由.cpp 生成.exe 文件，命令类似：g++ a.cpp
 - 6) 由.cpp 生成.o 文件，命令类似：g++ -c a.cpp
 - 7) 由.cpp 生成.s 文件，命令类似：g++ -S a.cpp
 - 8) 由.s 文件生成.exe 文件，命令类似：g++ a.s

以上这些文件之前行成了一个关系对，比如(.cpp .i)表示可由.cpp 文件生成.i 文件，同理(.i .s)表示可由.i 文件生成.s 文件。以上这些编译文件之间的生成关系是 make 隐式规则常见的成对关系。下面是这些文件关系对的总结

```
(.cpp .i), (.cpp .s), (.cpp .o), (.cpp .exe), (.s .o), (.s .exe), (.o .exe), (.a .exe)
```

4、查看内置的隐式规则

make 内置的隐式规则是最基本的常用的并且通常是成对的关系。可以在命令行中输出以下命令查看所有隐式规则和隐式变量

```
mingw32-make -p
```

下面列举几个作为示例来讲解。

示例 4.8：内置的隐式规则

1)、使用 `cc` 工具编译 `.c` 文件的内置隐式规则为:

```
% .o: %.c
      $(COMPILE.c) $(OUTPUT_OPTION) $<
```

其中变量 `COMPILE.c` 和 `OUTPUT_OPTION` 的值可在 `-p` 选项输出的内容中找到其原始定义,也可在 `makefile` 文件中使用 `@echo $(COMPILE.c)` 查看展开后的最终值。本示例

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c

OUTPUT_OPTION = -o $@
```

其中 `CC = cc`, 其余变量 `CFLAGS`、`CFLAGS`、`TARGET_ARCH` 未定义, 所以, 以上代码等效于:

```
% .o: %.c
      cc -c -o $@ $<
```

即, 使用 `cc` 工具编译编译依赖 `%.c` 指定的文件, 并将其输出为 `%.o` 指定的文件。

2)、使用 `pc` 工具编译 `Pascal` 程序的内置隐式规则为:

```
% .o: %.p
      $(COMPILE.p) $(OUTPUT_OPTION) $<
```

3)、使用 `f77` 工具编译 `Fortran/Ratfor` 程序的内置隐式规则为:

```
% .o: %.f
      $(COMPILE.f) $(OUTPUT_OPTION) $<
```

4)、使用 `as` 工具(`g++`)编译汇编程序的内置隐式规则为:

```
% .o: %.s
      $(COMPILE.s) -o $@ $<
```

5)、使用 `yacc` 工具生成 `.c` 文件的内置隐式规则为:

```
% .c: %.y
      $(YACC.y) $<
      mv -f y.tab.c $@
```

6)、链接单一 `.o` 文件为可执行程序程序的内置隐式规则为:

```
%: %.o
      $(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

5、禁用隐式规则

有时需要禁用隐式规则, 只有两种取消隐式规则的方法, 如下:

- 在命令行使用 `-r` 或 `-R` 参数取消所有隐式规则和隐式变量。比如, 输入 “`mingw32-make -R`”。
- 对隐式规则进行重定义。
- 另外, 还应在 `makefile` 文件中使用以下语句禁用隐式的后缀规则

```
.SUFFIXES:
```

除了隐式的模式规则外, 很多隐式规则还被使用后缀规则进行了定义, 因此, 即使使用了 `-r` 或 `-R` 参数, 仍可能会有隐式的后缀规则执行, 为此, 要彻底禁用隐式规则, 还需要禁用后缀规则。

6、需要注意的是，给目标文件指定明确的依赖文件并不会影响隐式规则的搜索，比如

```
xxx.o:xxx.p
```

若工作目录下同时存在 `xxx.cpp` 和 `xxx.p` 文件，则会使用 `g++` 编译 `xxx.cpp` 来生成 `xxx.o`，这是因为隐式规则列表中 `.cpp` 文件的隐式规则在 `.p` 文件隐式规则之前。

7、如果 `makefile` 中包含的隐式规则过多，可能会影响执行效率，考虑以下简单的代码

```
aa:xxx.c
    @echo A
xxx.c:
```

本示例假设 `xxx.c` 对应的真实文件不存在，因此，本示例会启用隐式规则来创建 `xxx.c` 文件，此时，`make` 会将 `xxx.c` 当作可执行文件 `xxx.c.exe` 来创建，同理，若依赖是另一个不存在的文件的名称，比如 `yyy`，则 `make` 会将其当作 `yyy.exe` 来创建，所以，以上代码，`make` 会考虑以下方式来创建 `xxx.c.exe` 文件

- 通过链接一个 `xxx.c.o` 文件创建，该隐藏规则对应为

```
%: %.o
    $(LINK.o) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

- 通过 C 编译并链接一个 `xxx.c.c` 文件一步到位来创建，该隐藏规则对应为

```
%: %.c
    $(LINK.c) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

- 通过 Pascal 编译并链接一个从 `foo.c.p` 来创建，该隐藏规则对应为

```
%: %.p
    $(LINK.p) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

- 以及其他一些更多的可能。

由于 `xxx.c.o`、`xxx.c.c`、`xxx.c.p` 这样的文件通常是不存在的，所以，最终会导致以上这些创建 `xxx.c` 文件的方式失败，但是，这些创建过程仍被 `make` 考虑了，这将影响效率。在命令行输入

```
mingw32-make -d
```

执行以上代码可以看到该代码尝试执行了非常多的隐藏规则。`-d` 参数表示输出所有的调试信息。

4.4 中间文件、万能匹配规则、默认规则

4.4.1 中间文件(intermediate file)

- 1、链：若一个目标文件需要一系列规则才能创建它，则称这一系列规则为一个链。同一个隐式规则只能在一个链中出现一次，这可以防止在搜索隐式规则链时出现死循环。
- 2、中间文件是指在一个规则链中间的文件，但该文件应是不存在的文件，而且不能在 `makefile` 文件中

被作为目标或依赖提及(即, 不能在 `makefile` 中被明确提及), 并且该中间文件能根据已存在的文件使用隐式规则重建。比如, 以下语句

```
aa:xx.o
```

假设 `xx.o` 文件不存在, 并且在工作目录下存在一个名为 `xx.y` 的文件(这是一个 Yacc 文件), 则 `make` 会使用隐式规则由 `xx.y` 创建一个 `xx.c` 文件, 再由 `xx.c` 文件创建 `xx.o` 文件, 这里的 `xx.c` 文件就是中间文件。

- 3、中间文件的显著特点是中间文件在 `make` 执行结束后会被自动删除, 删除时会打印出执行的命令以显示删除了哪些中间文件。

4、.INTERMEDIATE 特殊目标

将文件作为 `.INTERMEDIATE` 特殊目标的依赖, 可把该文件标记为中间文件, 即使这些文件在 `makefile` 中被明确提及。被指定为中间文件后, 这些文件在 `make` 执行结束后会被自动删除。

5、.NOTINTERMEDIATE 特殊目标

将文件作为 `.NOTINTERMEDIATE` 特殊目标的依赖, 可取消该中间文件, 若不为 `.NOTINTERMEDIATE` 指定依赖, 则将禁用 `makefile` 中的所有中间文件。

6、.SECONDARY 特殊变量

使用 `.SECONDARY` 特殊变量可用于保留指定的文件不被自动删除, 被 `.SECONDARY` 指定的文件被称为次要文件(secondary 文件), 次要文件同时也是中间文件。

示例 4.9: 中间文件 1

```
bb:ww.o
    @echo A
ww.o:
    @echo B
    g++ -c a.cpp -o ww.o
.INTERMEDIATE:ww.o      #将 ww.o 标记为中间文件
```

`ww.o` 是中间文件(强制指定)。在命令行输入 `mingw32-make` 可以看到打印出的删除中间文件的信息。以上示例需自行准备源文件 `a.cpp`, 并要求在执行 `make` 命令之前 `ww.o` 文件不存在, 并且要求在规则 `ww.o` 的命令中创建 `ww.o` 文件。

示例 4.10: 中间文件 2

```
bb:aa
    @echo A
aa:aa.o
    @echo B
aa.o:
    g++ -c aa.cpp
```

`aa.o` 不是中间文件因为未从隐式规则创建。以上示例需自行准备源文件 `aa.cpp`, 并假设 `aa.o` 不存在。

示例 4.11: 中间文件 3

```
bb:aa
    @echo A
aa:aa.o
    @echo B
```

```
aa.o:
```

aa.o 是中间文件(由隐式规则创建)，无命令的目标“aa.o:”会导致启用隐式规则。以上示例需自行准备源文件 aa.cpp，并假设 aa.o 不存在。

4.4.2 万能匹配规则(match-anything rule, 简称万能规则)

- 1、万能匹配规则(match-anything rule): 是指模式规则的目标模式只有一个%(此时可匹配任何名称)的规则，万能匹配规则非常有用。
- 2、万能匹配规则的效率比隐式规则更低，若一个万能规则包含一个使用隐式规则创建的依赖文件，则由于万能匹配规则匹配的情况比较多并且比较复杂，从而可能会试图多次通过隐式规则来创建依赖文件，虽然最终这些文件未被创建成功，但仍会去执行这些检测，由于次数比较多，从而影响效率。
- 3、为了提高万能匹配规则的效率可以使用双冒号将万能匹配规则标记为终端的(terminal)。终端万能匹配规则除非在依赖存在的情况下才会被执行，即使它的依赖可由隐式规则创建也不会被执行，也就是说，终端万能匹配规则没有进一步的链，即，隐式规则到此结束。

示例 4.12: 非终端万能匹配规则

```
bb:xx
    @echo A
%:a.o                #非终端万能规则。a.o 会使用隐式规则创建
    @echo B=$@
```

以上示例需自行准备源文件 a.cpp，本示例会使用隐式规则创建文件 a.o 然后会被删除。在命令行输入 mingw32-make，其输出如下(为什么会输出第二行的代码详见 makefile 重制 5.1.4 小节):

```
g++      -c -o a.o a.cpp
B=makefile
B=xx
A
rm a.
```

示例 4.13: 终端万能匹配规则

```
bb:xx
    @echo A
%::a.o                #终端万能规则不会使用隐式规则创建 a.o，即隐式规则的启动到此终止
    @echo B=$@
```

以上示例需自行准备源文件 a.cpp，但本示例不会调用隐式规则创建文件 a.o，最终会由于终端万能规则的依赖 a.o 不存在而产生错误，若文件 a.o 真实存在，则以上示例可执行。

- 4、非终端万能匹配规则(non-terminal match-anythin)不能用于隐式规则(或模式规则)的依赖，即，隐式规则(或模式规则)的依赖不能作为非终端万能规则的目标；在匹配模式规则时，只会优先匹配显式的模式规则，而不会匹配非终端万能匹配规则，可以将其简单的理解为，非终端万能规则的优先级低于显式模式规则。下面以示例进行说明。

示例 4.14: 非终端万能匹配规则的优先级

```
bb:bx
    @echo A
```



```

%x:%.cpp      #假设 b.cpp 真实存在
@echo B=$@
bb:bx         #假设 a.cpp 真实存在
@echo C=$@

```

匹配: $bb:bx \rightarrow \%x:\%.cpp$ 等效于 $bx:b.cpp$

以上示例假设 `a.cpp` 和 `b.cpp` 文件真实存在。依次输出 `B=bx`, `A`, 可见依赖 `bx` 优先匹配的是显示的模式规则 `%x:%.cpp` 而不是非终端万能匹配规则 `%:a.cpp`。

示例 4.15: 非终端万能规则的匹配原理 1

```

bb:bx
@echo A
%x:%yy      #假设 byy 不存在
@echo B=$@
%:a.cpp     #假设 a.cpp 真实存在
@echo C=$@

```

匹配: $bb:bx \rightarrow \%x:\%yy$ 等效于 $bx:byy$ 不会匹配 $\rightarrow \%:a.cpp$
产生错误(byy 不存在)

以上示例会产生错误。以上示例, 依赖 `bx` 优先匹配显示的模式规则 `%x:%yy`, 生成名称为 `byy` 的依赖, 由于 `byy` 的真实文件不存在, 需搜索适用的模式规则, 但模式规则的依赖不能作为非终端万能规则的目标, 因此, 非终端万能规则不会被搜索, 最终因为 `byy` 没有合适的模式规则而产生错误。

示例 4.16: 非终端万能规则优先级的匹配原理 2

```

bb:bx
@echo A
%x:%yy      #假设 byy 不存在
@echo B=$@
%:a.cpp     #假设 a.cpp 真实存在
@echo C=$@
byy:        #此规则无命令, 会导致启用模式规则

```

匹配: $bb:bx \rightarrow \%x:\%yy$ 等效于 $bx:byy$ 匹配 $\rightarrow byy:$ 匹配 $\rightarrow \%:a.cpp$ 等效于 $byy:a.cpp$

以上示例依次输出 `C=byy`, `B=bx`, `A`。本示例的 `byy` 不属于模式规则的依赖, 所以会搜索非终端万能规则。

4.4.3 默认规则

- 1、默认规则通常用于在没有合适的显示规则和隐式规则时使用。
- 2、最简单的方法就是将一个终端万能匹配规则用于默认规则。
- 3、.DEFAULT 特殊目标
 - .DEFAULT 特殊目标通常被用于默认规则, .DEFAULT 特殊目标用于没有在显示规则中作为目标并且没有隐式规则适用的依赖。
- 4、.DEFAULT 特殊目标和终端万能匹配规则是有区别的, .DEFAULT 仅用于依赖, 而终端万能匹配规则属于模式规则, 目标和依赖都适用。详见下面的示例
- 5、没有命令的 .DEFAULT 会取消掉之前所有使用 .DEFAULT 指定的命令。注意, 只能取消在此之前的命令。

以下示例的目标和依赖假设都不存在对应的文件

示例 4.17: .DEFAULT 作为默认规则

```
#输出 C=bb
aa:bb
.DEFAULT:      #仅适用于 bb
    @echo C=$@
```

示例 4.18: .DEFAULT 作为默认规则仅适用于依赖

```
#依次输出 C=dd
aa:bb
bb:dd
.DEFAULT:      #仅适用于 dd
    @echo C=$@
```

示例 4.19: 终端万能匹配规则作为默认规则

```
#依次输出 C=bb, C=aa
aa:bb
%:            #aa 和 bb 都适用
    @echo C=$@
```

示例 4.20: 终端万能匹配规则作为默认规则适用于目标和依赖

```
#依次输出 C=dd, C=bb, C=aa
aa:bb
bb:dd
%:            #aa 和 bb 都适用
    @echo C=$@
```

4.5 隐式规则搜索算法

- 1、可以看到，内置的隐式规则非常多，除此之外，还有可能会定义比较多的显示模式规则，因此，对于一个给定的目标，需要在众多的规则中进行搜索，以找到适用的规则。
- 2、本小节的搜索算法适用于
 - 1) 没有命令的双冒号规则
 - 2) 没有命令的普通规则的每个目标
 - 3) 不是任何规则目标的每个依赖
 - 4) 在搜索规则链时，来自隐式规则链中的依赖。
- 3、后缀规则会在 `make` 命令读取 `makefile` 文件时被转换为等效的模式规则。对于形如 `A(M)` 的存档成员目标，以下的搜索算法会执行两次，第一次的目标是整个目标名，若搜索失败，则以“M”作为目标名进行第二次搜索。
- 4、具体搜索过程如下：
 - 1) 将目标 `T` 的名称分离为目录部分 `D` 和文件名部分 `N`，比如 `aa/bb.o`，则 `D` 为 `aa/`，`N` 为 `bb.o`
 - 2) 列出所有与 `T` 或 `N` 匹配的模式规则，若模式规则的目标中含有斜杠，则与 `T` 匹配，否则与 `N` 匹

配。

- 3) 只要这个模式规则列表中有一个非万能规则(可简单理解为只要含有常规规则), 或者如果 T 是隐式规则的依赖, 则将列表中所有的非终端万能规则删除。
 - 4) 删除列表中的所有没有命令行的规则
 - 5) 对这个模式规则列表中的所有模式规则:
 - (1) 计算出模式规则的词干 S, S 应是 T 或 N 中匹配%的非空部分
 - (2) 计算依赖文件名称。把依赖模式中的%替换为 S, 若目标模式中不含斜杠, 则把 D 加到替换之后的每一个依赖文件开头, 形成完整的依赖文件名。
 - (3) 测试规则的所有依赖文件是否存在或是否应该存在, 若所有的依赖文件都存在、应该存在或没有依赖, 则退出查找, 使用该规则。应该存在是指若一个文件名在 `makefile` 中作为目标或目标 T 的显示依赖被提及, 则认为该文件应该存在。
 - 6) 若到第 5 步还未找到合适的模式规则, 则进一步搜索, 对列表中的每个模式规则:
 - (1) 若该规则是终端的(`terminal`), 则忽略它, 并继续执行下一个规则
 - (2) 像之前一样计算依赖文件名称(同步骤 5)
 - (3) 测试此规则的依赖文件是否存在或是否应该存在
 - (4) 对于此规则中不存在的依赖文件, 递归的调用这个算法, 以查看是否可由隐式规则来创建该依赖文件。
 - (5) 若所有的依赖文件都存在、应该存在、或可由一个隐式规则来创建, 则退出查找, 使用该规则
 - 7) 若未找到模式规则, 则再次尝试步骤 5 和 6, 并修改“应该存在”的定义为“若文件名作为目标被提及或作为任何目标的显示依赖被提及, 则该文件应该存在”, 此步骤用于与旧版本的 GNU `make` 兼容, 不推荐依赖此步骤。
 - 8) 若没有隐式规则可创建这个规则的依赖文件, 则执行特殊目标 `DEFAULT` 所指定的命令, 若 `makefile` 中没有定义 `DEFAULT`, 则表示没有可用的命令来创建 T, `make` 退出。
- 5、一旦找到了适合的模式规则, 除匹配 T 或 N 的模式之外, 对其他模式规则中的目标模式进行配置, 使用词干 S 替换其中的%, 将得到的文件名保存直到执行更新目标文件 T 的命令, 在命令执行以后, 把每一个储存的文件名放入数据库, 并标记为已更新, 其时间戳和文件 T 相同。
- 6、图 4.3 为隐式规则搜索算法的流程图, 该流程图简化了部分分支, 以目标 N 为例。其中矩形中的内容表示需要被搜索的模式规则, 比如含有“常规模式、非终端模式、终端万能模式”的矩形, 表示整个过程还需要对这 3 种类型的模式进行搜索。
- 7、模式规则的优先级总结
- 显示模式规则优先于隐式规则。
 - 若缺失依赖文件同时符合多个模式规则, 则执行与第一个匹配的模式规则。
 - 依赖文件存在或被提及的规则, 优先于那些使用隐式规则来创建其依赖文件的规则。
 - 隐式规则中, 优化后的规则优先于分步链。
 - 非终端万能规则的优先级低于显式模式规则。
 - 不需要链接其他隐式规则就可以满足的规则(例如, 没有依赖或其依赖已经存在或已经提到的规则)总是优先于必须通过链接其他隐式规则来实现依赖的规则

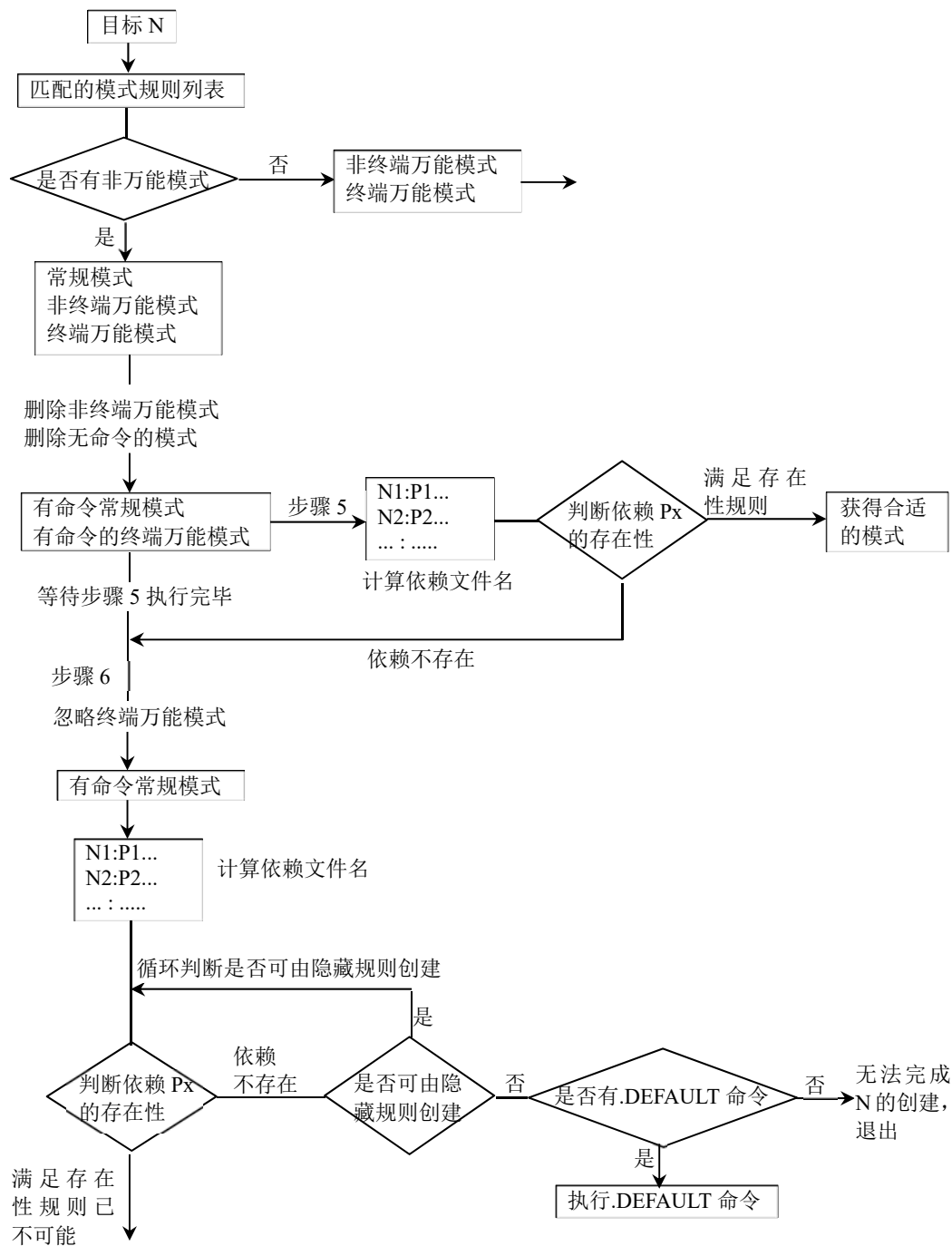


图 4.3 隐式规则搜索算法

4.6 后缀规则(.SUFFIXES) (nmake 不适用)

- 1、后缀规则是旧式的隐式规则，新版使用模式规则，为了兼容旧版本以及其他版本的 `make`，有必要讲解一下后缀规则。

2、后缀规则有双后缀和单后缀两种。

3、双后缀规则语法为：

.依赖后缀.目标后缀

以上各字符间无空格。不能为后缀规则指定依赖，否则会被当作普通规则。双后缀规则等价于“

%目标后缀:%依赖后缀

比如“.c.o”等价于模式规则“%.o:%.c”，即，匹配以“目标后缀.o”结尾的所有目标文件，然后将目标文件名的后缀替换为“依赖后缀.c”以形成依赖名称。

3、单后缀规则

单后缀规则只有一个依赖后缀名，比如“.c”等价于模式规则“%:%.c”，即，单后缀规则可以匹配任何目标文件，其依赖可通过将“依赖后缀.c”直接追加到目标文件名之后而得到。

4、可识别后缀(列表)

可识别后缀(列表)简称为后缀列表指的是特殊目标.SUFFIXES 的依赖的名字，可通过向该特殊目标增加依赖来添加一个可识别的后缀，比如

.SUFFIXES: .a .b

表示把后缀.a 和.b 加入到可识别后缀列表的末尾。

5、后缀不在后缀列表中的规则将被禁用。这意味着，自定义的后缀规则的后缀也需要添加到后缀列表中，否则将不可用。

示例 4.21：双后缀规则

```
.SUFFIXES: .bb      #必须将后缀.bb 添加到后缀列表，否则自定义的后缀规则将不可用
aa:xx.bb
    @echo B
.a.bb:              #自定义的双后缀规则。匹配 xx.bb，计算出的依赖名称为 xx.a
    @echo A
xx.a:
    @echo D
```

以上示例将依次输出 D、A、B。依赖 xx.bb 匹配双后缀规则.a.bb:，最后计算出依赖名称为 xx.a，然后执行目标 xx.a 的命令输出 D，然后执行双后缀规则的命令输出 A，最后执行目标 aa 的命令输出 B。

示例 4.22：单后缀规则

```
.SUFFIXES: .bb      #必须将后缀.bb 添加到后缀列表，否则自定义的后缀规则将不可用
aa:xx
    @echo B
.bb:                #自定义的单后缀规则。匹配 xx，计算出的依赖名称为 xx.bb
    @echo A
xx.bb:
    @echo D
```

以上示例将依次输出 D、A、B。依 xx 匹配单后缀规则.bb:，最后计算出依赖名称为 xx.bb，然后依

次执行相应目标的命令输出 D、A、B。

6、删除默认的后缀规则

若需要重置或删除默认的可识别后缀，应先删除所有的可识别后缀之后再重新添加，即

```
.SUFFIXES:                #删除已定义的所有后缀  
.SUFFIXES: .a .b          #重新添加新的后缀
```

- 7、与没有命令行的模式规则不同，没有命令行的后缀规则是无意义的，不能用于取消之前定义的后缀规则，它将实现的是将后缀规则作为目标加入到 **make** 数据库中。
- 8、可以通过变量 **SUFFIXES** 查看所有的可识别后缀列表，但需注意，该变量在 **make** 读取所有的 **makefile** 文件之前会被重置为默认值。还应注意，不应使用 **SUFFIXE** 变量来修改可识别后缀列表，应使用特殊目标 **.SUFFIXES** 来修改后缀列表。

第 5 章 make 读取 makefile 文件的方式

5.1 make 执行 makefile 文件的过程

5.1.1 常规读取 makefile 文件的方式

- 1、默认情况下 make 程序会在工作目录中按照文件名顺序 GNUmakefile、makefile、Makefile 查找 makefile 文件读取并执行。若 make 工具未找到以上文件，则不读取任何其他文件。不建议使用 GNUmakefile 作为 makefile 文件的文件名，因为该名称只适用于 GNU make。

- 2、make 命令的-f 选项

表示读取由-f 选项指定的文件，若使用了-f 选项则不会检查默认的 makefile 文件名。比如，

```
mingw32-make -f abcde
```

表示读取名称为 abcde 的 makefile 文件，再如

```
mingw32-make -f abcde xxx
```

表示执行名称为 abcde 的 makefile 文件中的目标 xxx 中的命令

- 3、若要读取多个 makefile 文件，则需要指定多个-f 选项，多个 makefile 文件将会按照指定的顺序链接并被 make 解析执行。比如，
假设 makefile1 文件的内容为：

```
aa:bb
```

```
@echo A
```

makefile2 文件的内容为：

```
bb:
```

```
@echo B
```

在命令行输入

```
mingw32-make -f makefile1 -f makefile2
```

将依次输出 B，A

- 4、可以在 CMD 中使用 make 工具直接指定目标以使用 make 的隐式规则来创建文件，这种情况下不需要有 makefile 文件。比如，若当前目录存在一个名为 m.cpp 的文件，则若在命令行输入以下命令：

```
mingw32-make m.o
```

然后，可以看到执行了如下命令，

```
g++ -c -o m.o m.cpp
```

并且文件 m.o 或 m.obj 被创建(可在目录中查看是否创建了 m.o 或 m.obj)。

若输入以下命令

```
mingw32-make m
```

将使用隐式规则直接由 `m.cpp` 生成可执行文件 `m.exe`

5.1.2 include 指示符

1、`include` 指示符告诉 `make` 暂停读取当前的 `makefile`，并依次读取 `include` 列出的一个或多个文件，完成后再继续当前 `makefile` 的读取。

2、`include` 的语法为

```
include 文件名...
```

其中，文件名可以使用通配符，第一个字符可以是空格，但不能是 `TAB`(或特殊变量 `RECIPEPREFIX` 的值)，若以 `TAB` 开头，则会将其视为命令来处理。`include` 和文件名之间，以及文件名之间使用空格隔开，可以在行尾使用 `#` 注释，若文件名含有变量或函数引用，则展开它们。

示例 5.1：使用 `include` 指示符加载 `makefile` 文件

假设 `makefile` 文件的内容如下：

```
#makefile1 文件的内容
bb:
    @echo B
```

假设 `makefile` 的内容如下：

```
#makefile 文件的内容
aa:bb
    @echo A
include makefile1      #加载 makefile1 文件。注意，不能以 TAB 开头
```

然后在命令行输入 `mingw32-make` 将依次输出 B，A。

3、`include` 指示符常用在以下两种情形使用：

- 若多个 `makefile` 需要相同的一组共用变量或模式规则，此时可以将这些共同的变量和模式规则定义在一个文件中，然后在需要使用的 `makefile` 中使用 `include` 包含进来，这种用法类似于 C/C++ 使用 `include` 包含头文件。
- 另一种用法是，当希望从源文件自动产生依赖时(在命令行使用 `make` 工具的 `-M` 或 `-MM` 选项)，可以将产生的依赖放入一个 `makefile` 文件中，然后在主 `makefile` 文件中使用 `include` 包含该文件。

4、`include` 指示符的搜索目录

若 `include` 指定的文件名不是以斜杠开头(在 MS-DOS 环境下不是以驱动器号和冒号开头)，并且未在当前目录下找到指定的文件，则首先搜索使用 `-I`(大写字母 `i`)参数指定的目录，然后会搜索一些系统设置的默认目录，如 `usr/local/include` 等。可以在命令行中使用特殊参数 `-` (例如 `-I-`)禁用默认目录搜索。

示例 5.2：include 指示符的搜索目录(-I 参数的使用)

假设 g:/qt4/xx/makefile1 的内容如下:

```
# g:/qt4/xx/makefile1 文件的内容
bb:

    @echo B
```

假设 g:/qt1/makefile 的内容如下:

```
#g:/qt1/makefile 文件的内容
aa:bb

    @echo A
include makefile1
```

然后在命令行输入命令(指定-I 参数)

```
mingw32-make -IG:/qt4/xx
```

将依次输出 B, A。注: -I 之后没有空格

- 5、若在所有目录下都未找到指定的 makefile 文件,则会发送一个警告提示,但并不会导致致命错误而立即退出,而是继续处理包含 include 的 makefile 文件的后续内容,当完成读取整个 makefile 后,make 会试图使用规则来重制任何过期的或 include 指定的但未找到的 makefile 文件,只有当不能重制 makefile 文件时,才会导致致命错误并退出。
- 6、使用-include(或 sinclude)指示符可以让 make 忽略由于不存在或无法重制 makefile 时的错误信息,除此之外,与 include 效果相同。

5.1.3 MAKEFILES 和 MAKEFILE_LIST 变量(nmake 不适用)

- 1、若在当前系统中定义了一个 MAKEFILES 的环境变量,则 make 首先将此变量的值添加到变量 MAKEFILE_LIST(makefile 名称列表)中,以便在其他的 makefile 文件之前被读取。但是,最终目标永远不会是 MAKEFILES 指定的 makefile 文件中的目标。若最终没有找到 MAKEFILES 列出的某一个 makefile 文件,不会提示错误,也不会退出。
- 2、MAKEFILES 很少在实际应用中被设置。若在环境变量中被设置,则在多级 make 调用时,每一级 make 都会读取 MAKEFILES 变量所指的文件,这会容易产生混乱。MAKEFILES 通常用在 make 递归调用过程中的通信,可以使用该变量来指定一个定义了通用的隐式规则和变量的文件,比如设置默认搜索路径,这样设置的隐式规则和变量可以被任何 make 进程所使用,不过需要注意的是,MAKEFILES 是系统级别的环境变量,若更换了系统,则不一定会设置 MAKEFILES 环境变量。

3、make 读取 makefile 文件的顺序

make 首先读取环境变量 MAKEFILES 所指定的文件列表,然后是工作目录下的默认文件名,之后是 include 所指定的文件。

4、MAKEFILE_LIST 变量

make 工具在读取多个 makefile 文件时,在这些文件解析执行之前,make 会将读取的文件名自动的依次追加到变量 MAKEFILE_LIST 之中。这些 makefile 文件包括来自环境变量 MAKEFILES 指定的、命令行指定的、当前工作目录下默认的、以及 include 指定的 makefile 文件。所以,可以通过查看或测试 MAKEFILE_LIST 变量的最后一个文件名来判断当前 make 程序正在处理的 makefile 文件名。

5.1.4 重制 makefile (nmake 不适用)

- 1、若 makefile 文件由其他文件(如 CMake, qmake 等)重新生成, 那么就需要重新读取最新的 makefile, 此时 make 的处理方式是:

读取所有 makefile 文件后, make 按照处理 makefile 文件的顺序将每个 makefile 文件作为目标, 并寻找更新它们的规则, 如果并行构建被启用, 则 makefile 也将并行重建。若在 makefile 文件中或另一个 makefile 文件中存在一个更新某个 makefile 文件的规则或隐式规则, 则更新该 makefile 文件。检查完所有的 makefile 文件之后, 如果有任何的 makefile 被更新了, 则 make 从头开始, 重新读取所有的 makefile 文件, 并再次尝试更新它们, 但通常不会被再次更新, 因为已经是最新了。读取完成以后, 再开始解析已经读取的 makefile 文件。需要注意的是, 如果 make 不能找到或创建任何 makefile 文件, 这并不是错误, 因为, 不需要 makefile 文件也能执行 make 命令(即使用隐式规则创建目标)

- 2、可以使用一些方法来阻止 makefile 文件的更新, 比如, 在 makefile 文件中使用 makefile 文件名作为目标编写一个空命令的显示规则, 比如

```
makefile;;
```

注意: 末尾有一个分号, 表示使用空命令

- 3、以下情形 make 不会重制 makefile, 因为可能导致无限循环,

- 带有命令的目标最终被计算为 makefile 文件名的, 并且没有依赖的双冒号规则, 比如,

```
makefile;;
```

```
%::;
```

以上代码的末尾有一个分号, 表示使用空命令。

- 标记为伪的(.PHONY)makefile 文件, 比如.PHONY:makefile

示例 5.3: 重制 makefile 1

```
aa:
    @echo A
makefile:FORCE
    @echo B=$@
FORCE:
```

以上示例假设 makefile 文件的名称为默认名称 makefile, 该示例依次输出 B=makefile、A。目标 makefile 在默认目标 aa 之前被执行, 可见, make 读取 makefile 文件之后的第一件事情就是将 makefile 文件名作为目标在 makefile 文件中寻找规则, 以更新 makefile 文件, 本示例不会产生死循环, 因为目标 makefile 没有对 makefile 文件的内容进行更改。

示例 5.4: 重制 makefile 2

```
aa:
    @echo B
makefile:
    @echo C=$@
```

以上示例假设 makefile 文件的名称为默认名称 makefile, 本示例输出 B。本示例的目标 makefile 会被 mingw32-make 工具判断为死循环, 从而被忽略执行。

示例 5.5: 重制 makefile 3

```
aa:
    @echo A
%::
    @echo B=$@
```

以上示例假设 makefile 文件的名称为默认名称 makefile，本示例输出 B，make 工具在读取 makefile 文件后，将 makefile 作为目标名称在 makefile 文件中寻找到匹配的万能模式规则 “%::”，但由于该规则是双冒号规则，此情况会被 make 工具判定为死循环，而忽略执行。

示例 5.6: 重制 makefile 4

```
aa:
    @echo A
%::FORCE
    @echo B=$@
FORCE:
```

以上示例假设 makefile 文件的名称为默认名称 makefile，本示例依次输出 B=makefile、A。虽然万能匹配模式规则 “%::FORCE” 可能会导致死循环，但，由于该规则未修改 makefile 文件，所以本示例不会产生死循环。我们明确的要求执行该规则，因此，该规则会被执行。

- 4、最后需要注意的是，make 工具和 makefile 文件是两回事，make 工具只是解释 makefile 文件的，我们完全可以在 makefile 文件中输入能更改 makefile 文件自身内容的命令，然后通过 make 工具执行该命令来更改 makefile 文件，比如

```
aa:
    @echo A>makefile
```

以上代码执行 mingw32-make 命令后，再打开 makefile 文件，将发现该文件的内容被更改为 echo 的输出内容 A，之前的代码不存在了。注：>表示将左侧的结果输入到右侧的文件中

5.1.5 make 执行过程总结

以下是 make 执行过程的总结

- 1) 依次读取环境变量 MAKEFILES 指定的 makefile 文件列表
- 2) 读取工作目录下的 makefile 文件，按名称顺序 GNUmakefile、makefile、Makefile 首先找到哪个就读取哪个。
- 3) 依次读取工作目录 makefile 文件中使用 include 包含的 makefile 文件
- 4) 在读取的所有 makefile 文件中，查找重建 makefile 文件的规则。
- 5) 初始化变量值并展开需要立即展开的变量和函数，并根据预设条件确定执行分支
- 6) 根据最终目标以及其他目标的依赖关系建立依赖关系图(或称为依赖关系链表)
- 7) 执行依赖关系链中除最终目标以外的所有规则
- 8) 执行最终目标所有规则

5.2 获取依赖

5.2.1 自动获取依赖(nmake 不适用)

- 1、一个源程序代码，通常会包含非常多的头文件，此时就需要在 `makefile` 中编写当头文件被更改后重建目标文件的规则，其规则类似如下代码所示

```
main.o: def.h
```

对于一个大型工程，需要在 `makefile` 中编写非常多的类似以上的规则，而且在源文件中添加或删除一个头文件后，必须非常小心的去修改 `makefile` 文件，对于大型工程，这个工作量是比较费力费时的。

- 2、自动获取依赖(g++的-M 或-MM 参数)

为了避免这个麻烦，可以使用带-M 或-MM 参数的 g++ 工具来自动寻找源文件中所包含的头文件，并且还能自动生成文件的依赖关系，其中-M 参数含有标准库头文件，-MM 参数不包含标准库头文件，通常情况下标准库头文件是不会被更改的，没有必要列出来。比如，输入以下命令

```
g++ -MM a.cpp
```

将输出

```
a.o: a.cpp b.h c.h
```

以上示例假设 `b.h`、`c.h` 是 `a.cpp` 中包含的头文件且 `a.cpp`、`b.h`、`c.h` 都是真实存在的。可见，编译器已为我们生成了相应的规则，无需手动编写了。

- 3、有了依赖关系的数据，然后只需将这些数据保存到一个文件中，再使用 `include` 包含进 `makefile` 文件就可以了。注：可在命令行中使用“>”将左侧命令执行的结果输出到右侧指定的文件中，若要向右侧文件追加内容可使用“>>”。比如

```
g++ -MM a.cpp > tt.txt
```

以上命令表示将执行 g++ 后的结果输出到 `tt.txt` 文件中。

示例 5.7：自动获取依赖并生成依赖文件

①、C++源文件准备

在 `g:/qt5` 的 `a.cpp` 中输入以下代码

```
g:/qt5/a.cpp(主源文件)
```

```
#include<iostream>
#include "d.h"
#include "e.h"
extern int b;
int main(){          std::cout<<"A"<<b<<std::endl;          return 0;}
```

在 `g:/qt5` 的 `b.cpp` 中输入以下代码

```
g:/qt5/b.cpp(主源文件)
```

```
#include "f.h"
```

```
int b=1;
```

②、C++头文件准备

在 g:/qt5 中新建 d.h、e.h、f.h 三个空的头文件。

③、在 g:/qt5 的 makefile 文件中编写如下代码

```
source = a.cpp b.cpp
a.exe:a.o b.o
    @echo A
    g++ a.o b.o
a.o:
    g++ -c -o a.o a.cpp
%.d:%.cpp
    g++ -MM $< > $@
include $(source:.cpp=.d)      #include 应放在最终目标之后,
                                #以免包含进来的文件中的目标成为最终目标
```

在 CMD 中转至 g:/qt5 并输入以下命令

```
mingw32-make
```

输出

```
g++ -MM b.cpp > b.d
g++ -MM a.cpp > a.d
g++ -c -o a.o a.cpp
g++      -c -o b.o b.cpp
A
g++ a.o b.o
```

本示例总共会生成 a.o、b.o、a.d、b.d、a.exe 五个文件。下面来看看以上代码的执行步骤：

1) 其中

```
g++ -MM $<>$@
```

表示将命令 g++ -MM \$< 执行的结果输出到文件 \$@ 中。

2) 首先，make 读取所有 makefile 文件，包括 include 包含进来的 makefile 文件，所以，本示例在读入 makefile 主文件后，首先执行 include 语句，该语句将 source 变量值中的 .cpp 字符替换为 .d 字符(变量值替换的语法见 3.1 小节)，所以依次读入 a.d 和 b.d 文件。

3) 将 makefile、a.d、b.d 作为目标，在所有 makefile 文件中查找匹配该目标的规则，找到模式规则 %.d:%.cpp 匹配 a.d 和 b.d，依次执行该规则的命令，分别自动生成 a.cpp 和 b.cpp 的依赖关系并分别输入到文件 a.d 和 b.d 中，然后再重复以上过程检查所有的 makefile 文件。至此，读入了 a.d 和 b.d 文件，最终，a.d 在本示例包含如下内容

```
a.o: a.cpp d.h e.h
```

b.d 本示例包含以下内容

```
b.o: b.cpp f.h
```

4) 以上过程完成后，查找默认目标，本示例为规则 a.exe:a.o 的目标，该规则依赖于 a.o 和 b.o 文件，由于以 b.o 作为目标所在的规则 b.o: b.cpp f.h (位于 b.d 文件中)没有命令，所以使用隐式规

则创建 b.o 文件，本示例使用隐式规则的命令是

```
g++ -c -o b.o b.cpp
```

以 a.o 作为目标的规则有 a.o:a.cpp d.h e.h(位于 a.d 文件中)和 a.o:，由于第 1 个规则没有命令，所以，执行后面这个规则中的命令，即，执行以下命令，该命令会创建一个 a.o 文件

```
g++ -c -o a.o a.cpp
```

5) 最后执行规则 a.exe:a.o 的命令输出 A 并使用 a.o 和 b.o 构建一个 a.exe 文件。至此整个执行过程完成。

6) 本示例读入包含的 makefile 文件后，其代码等效于以下代码

```
source = a.cpp b.cpp
a.exe:a.o b.o
    @echo A
    g++ a.o b.o
a.o:
    g++ -c -o a.o a.cpp
%.d:%.cpp
    g++ -MM $< > $@
a.o: a.cpp d.h e.h
b.o: b.cpp f.h
```

5.2.2 目录搜索(nmake 不适用)

1、在大型工程中，通常会将源代码、.o、.h 等文件分别放在不同的目录以进行区分管理，这时，就需要使用 make 的目录搜索功能在指定的目录下查找依赖文件了。

2、GNU make 通过 VPATH、GPAT 特殊变量和 vpath 指示符来实现目录搜索功能。VPATH、GPAT、vpath 不但会搜索依赖文件，同时也会搜索目标文件。但是无论是哪种搜索方式，当前目录永远都是第一搜索目录。

3、VPATH 变量语法如下：

```
VPATH = 目录 1 目录 2 ...
```

多个目录之间使用空格或冒号“:”隔开，在 MS-DOS 中还可以使用分号“;”隔开，make 将按照 VPATH 指定的顺序搜索目录。VPATH 变量指定的目录对所有类型的文件有效。

4、选择性搜索(vpath 指示符)

vpath 指示符可以实现为不同类型的文件指定不同的搜索目录，vpath 搜索被称为选择性搜索，共有三种语法格式，分别为

1) vpath 模式 目录 1 目录 2 ...

为所有匹配“模式”的文件搜索指定的目录，多个目录之间使用冒号“:”或空格隔开，在 MS-DOS 中还可以使用分号“;”隔开。“模式”通常应包含模式字符“%”，若没有%，则表示仅为指定的单独一个文件搜索目录。

2) vpath 模式

清除之前为符合“模式”的文件设置的搜索目录

3) vpath

清除所有已设置的搜索目录

- 5、若对相同的“模式”使用了多个 vpath 语句，则按 vpath 语句在 makefile 中出现的先后顺序来搜索目录。比如

```
vpath %.c aa bb
vpath % dd
vpath %.c ee
```

则搜索后缀为.c 的文件的顺序为依次搜索目录 aa、bb、dd、ee

- 6、通过目录搜索获得的路径名，可能不是规则的依赖列表中实际提供的路径名，因此

- 使用目录搜索找到的路径名可能需要被废弃(或丢弃)，通常，应废弃目标文件的全名，而不应废弃依赖文件全名，否则，规则将无法执行。
- 在规则的命令行中应使用自动化变量来代替依赖文件和目标文件。

- 7、make 对目录搜索找到的路径名是保存或废弃的算法如下：

- 1) 首先，若目标文件在工作目录下不存在，则执行目录搜索。
- 2) 若目录搜索成功，则将搜索到的完全文件名(即，同时含路径和文件名)保存为临时的目标文件。
- 3) 对规则中的所有依赖文件使用相同的方法处理。
- 4) 接着决定规则的目标是否需要重建，分以下两种情形
 - 目标不需重建：则目录搜索找到的路径名有效，使用搜索到的路径名。
 - 目标需要重建：则搜索到的目标文件路径名无效，目标文件将会在工作目录下被重建(即，使用当前目录)，也就是说，此时在工作目录和搜索到的路径下有两个目标文件(若目标文件被真实重建到工作目录下)，但工作目录下的目标文件是最新的。需要注意的是，搜索到的依赖文件的路径名始终有效，没有被废弃。

8、GPATH 变量

除以上算法外，其他版本的 make 可能会使用一种更简洁的算法，即：若目标文件在工作目录下不存在，并且搜索到的目标文件路径名存在，则无论该目标是否需要重建，都在搜索到的目标文件路径名下重建目标文件。GNU make 可以使用 GPATH 变量来实现这种算法，该变量的语法与 VPATH 相同。GPATH 变量应与 VPATH 或 vpath 同时使用，当搜索到目标文件的路径名时，而目标文件又同时出现在 GPATH 定义的列表的路径中，则搜索到目标文件的路径名将不会被废弃(即，目标文件使用搜索到的路径名)。

- 9、VPATH、vpath 指定的搜索目录对隐式规则同样有效。

示例 5.8、5.9、5.10 均假设 a.o，a.cpp 是真实文件，a.o 位于目录 g:/qt5/xx，a.cpp 位于目录 g:/qt5/yy。

示例 5.8：使用 VPATH 变量搜索目录 1

#本示例会收到“G:/qt5/xx/a.o 是最新的”提示

```
VPATH=G:/qt5/xx g:/qt5/yy
```

```
a.o:                                #假设 a.o 是真实文件，位于目录 g:/qt5 中
```

```
@echo $@
```

本示例将在 g:/qt5/xx 和 g:/qt5/yy 目录中搜索 a.o 文件，最终在 g:/qt5 目录中找到 a.o 文件。由于本示例的目标 a.o 不需重建，所以，目标文件使用搜索到的路径名。从 CMD 命令行收到的提示可以看到，目标文件使用的是搜索到的完整文件名。

示例 5.9: 使用 VPATH 变量搜索目录 2

```
#本示例输出 a.o
VPATH=G:/qt5/xx g:/qt5/yy
a.o:FORCE                                #假设 a.o 是真实文件, 位于目录 g:/qt5 中
    @echo $@
FORCE:
```

本示例将在 g:/qt5/xx 和 g:/qt5/yy 目录中搜索 a.o 文件, 最终在 g:/qt5 目录中找到 a.o 文件。由于 a.o 会被重建(并未被真正重建), 所以, 目标文件不使用搜索到的路径名, 而使用当前目录, 所以输出的目标文件只有文件名 a.o。

示例 5.10: 使用 VPATH 和 GPATH 变量搜索目录

```
#本示例输出 G:/qt5/xx/a.o
VPATH=G:/qt5/xx
GPATH=G:/qt5/xx
a.o:FORCE                                #假设 a.o 是真实文件, 位于目录 g:/qt5 中
    @echo $@
FORCE:
```

本示例将在 g:/qt5/xx 和 g:/qt5/yy 目录中搜索 a.o 文件, 最终在 g:/qt5 目录中找到 a.o 文件。本示例的目标 a.o 会被重建(并未被真正重建), 但由于增加了 GPATH 变量, 所以, 使用目标文件的完整路径名, 所以输出 G:/qt5/xx/a.o

示例 5.11: 使用 VPATH 和 vpath 搜索目录

①、C++文件准备

在 g:/qt5/yy 目录下准备一个 a.cpp 文件, 读者可自行准备代码, 只要能运行即可。

在 g:/qt5/xx 目录下准备一个 a.o 文件, 该文件可以是一个空文件, 但应比 a.cpp 的时间更旧。

②、在 g:/qt5 目录下的 makefile 文件中输入以下代码

```
VPATH=G:/qt5/xx
vpath %.cpp g:/qt5/xx g:/qt5/yy
a.o:a.cpp
    g++ -c $^ -o $@
    @echo y=$^;t=$@
```

本示例的目的在于演示重建目标文件时目标文件 a.o 的路径。

在 CMD 中转至 g:/qt5, 并输入以下命令:

mingw32-make

输出:

```
g++ -c g:/qt5/yy/a.cpp -o a.o
y=g:/qt5/yy/a.cpp;t=a.o
```

本示例将在 g:/qt5/xx 目录下搜索目标文件 a.o, 将在 g:/qt5/xx 和 g:/qt5/yy 目录下搜索依赖文件 a.cpp。最终, 在 g:/qt5/xx 目录下找到 a.o 文件, 在 g:/qt5/yy 找到 a.cpp 文件。由于 a.cpp 比 a.o 更新, 所以, 本示例的目标 a.o 需要重建, 于是执行目标 a.o 的命令使用 g++ 命令构建 a.o 文件。在构建 a.o 的过程中, 使用的依赖文件的目录是搜索目录, 即 g:/qt5/yy/a.cpp, 但由于目标 a.o 需要重建, 所以, 使用的目标文件的目录是当前目录。所以, 本示例将目标文件重建在了当

前目录，这时在当前目录和搜索目录 `g:/qt5/xx` 中同时都存在一个 `a.o` 文件，但当前目录下的 `a.o` 文件是最新的。

示例 5.12: 使用 `VPATH`、`GPATH`、`vpath` 搜索目录

①、C++文件准备

在 `g:/qt5/yy` 目录下准备一个 `a.cpp` 文件，读者可自行准备代码，只要能运行即可。

在 `g:/qt5/xx` 目录下准备一个 `a.o` 文件，该文件可以是一个空文件，但应比 `a.cpp` 的时间更旧。

②、在 `g:/qt5` 目录下的 `makefile` 文件中输入以下代码

```
VPATH=G:/qt5/xx
vpath %.cpp g:/qt5/xx g:/qt5/yy
GPATH=G:/qt5/xx
a.o:a.cpp
    g++ -c $^ -o $@
    @echo y=$^;t=$@
```

本示例的目的在于演示重建目标文件时目标文件 `a.o` 的路径。

在 CMD 中转至 `g:/qt5`，并输入以下命令：

`mingw32-make`

输出：

```
g++ -c g:/qt5/yy/a.cpp -o G:/qt5/xx/a.o
y=g:/qt5/yy/a.cpp;t=G:/qt5/xx/a.o
```

本示例与示例 5.11 是相同的，唯一的区别在于，由于本示例使用了 `GPATH` 变量，该变量将导致在重建目标 `a.o` 时使用搜索目录(即 `g:/qt5/xx`)而非当前目录，所以，本示例最终将目标 `a.o` 重建到了搜索目录中。

- 10、静态库文件(如 `a` 或 `lib` 文件)、共享库文件(`.so` 或 `.dll` 文件)也可以通过目录搜索找到。这只需为规则的依赖指定一个类似 “`-lname`” (第一个字母是小写 `L`) 的依赖文件名即可实现。
- 11、当依赖列表中有一个 “`-lname`” 形式的依赖时，`make` 将使用名称 “`name`” 首先搜索共享库文件，若未找到，则搜索静态库文件。具体过程如下：
 - 1) 在工作目录下搜索名称为 `libname.so` 的文件。
 - 2) 若未找到，则继续在 `VPATH` 或 `vpath` 指定的搜索目录下搜索。
 - 3) 若还未找到，则搜索系统库文件的默认目录，如 `usr/local/lib` 等(各系统可能各不相同)。
 - 4) 若还未找到，则 `make` 依照以上顺序查找名为 `libname.a` 的文件。
- 12、`.LIBPATTERNS` 特殊变量
 - 1)、库文件搜索时的文件名 `libname.so` 和 `libname.a` 是由特殊变量 `.LIBPATTERNS` 指定的。
 - 2)、在本人所在系统 `.LIBPATTERNS` 变量的默认值为

```
lib%.dll.a %.dll.a lib%.a %.lib lib%.dll %.dll
```

建议使用 `mingw32-make -p` 查看该变量的值。

- 3)、`.LIBPATTERNS` 变量的每个值都有一个模式字符 “`%`”，该变量的作用是，当出现 “`-lname`” 这样的依赖时，使用 `name` 替换 `.LIBPATTERNS` 变量的第一个值中的 `%`，并将替换后的名称作为库文件名执行以上的目录搜索，若找到，则使用该文件，若未找到，则使用 `name` 替换。

LIBPATTERNS 变量中的第二个值中的%，进行同样的查找，直至所有的值都完成查找。

- 4)、因此，若将LIBPATTERNS 变量的值设置为空，便可以取消库的目录搜索功能，也可给该变量指定其他值及顺序以控制库文件的搜索顺序。

示例 5.13：库的目录搜索

```
VPATH=g:/qt4/xx g:/qt4/yy
xx:-laa
@echo $^
```

若在 g:/qt4/yy 目录下存在文件 aa.lib 则本例将输出 g:/qt4/yy/aa.lib

第 6 章 递归调用(nmake 不适用)

6.1 make 工具的部分参数 (nmake 不适用)

1、make 工具(本文以 mingw32-make 为例)有 3 个特殊的参数 -t、-n、-q，若 make 工具带有这 3 个参数之一，则不会执行规则中的命令行，其中

- -t 表示更新所有目标文件的时间戳为当前系统时间，该命令仅会更改文件的时间而不会修改文件的任何内容。类似于 Linux 的 touch 命令。
- -n 表示只打印规则中的命令，但不执行该命令。此参数可用于查看某个规则执行了什么命令。
- -q 表示询问模式，不运行任何命令，并且无输出，但 make 会返回一个查询状态，返回 0 表示没有目标需要重建，1 表示存在需要重建的目标，2 表示有错误发生。

示例 6.1: 使用 make 工具的 -t、-n、-q 参数

```
aa:
    @echo A
    $(CXX) -c a.cpp -o a.o
```

①、若在 CMD 中输入

```
mingw32-make -t
```

则会显示“touch aa”，但不会执行目标 aa 的命令，此参数的意义表示，若工作目录下存在 aa 文件则会将该文件的日期修改为当前系统时间，不会对文件内容作任何更改，若不存在 aa 文件则会创建一个空的名称为 aa 的文件。

②、若在 CMD 中输入

```
mingw32-make -n
```

则会显示“echo A”和“g++ -c a.cpp -o a.o”，但不会执行目标 aa 的命令，也就是说字符 A 不会输出，文件 a.o 也未被重建。此参数的意义仅在于把目标中的命令打印出来。

③、若在 CMD 中输入

```
mingw32-make -q
```

则什么也不会输出，并且也没执行任何命令。

2、在规则中的命令之前使用符号“+”，可以使即使带有 -t、-n、-q 参数的 make 工具也执行该命令。比如

```
aa:
    +@echo A
    $(CXX) -c a.cpp -o a.o
```

若在 CMD 中输入

```
mingw32-make -n
```

则会依次显示“echo A”、“A”、“g++ -c a.cpp -o a.o”，其中，字符 A 是执行的 echo A 命令输出的。

3、在单个进程中执行规则的命令(&&符号)

1)、规则中的每一行命令都在一个独立的子 shell 进程中执行，也就是说，每一行命令都是在一个独立的 shell 进程中完成的，多行命令之间的执行相互独立，不存在依赖关系。

2)、若想使规则的多条命令在同一个 shell 进程中调用，可以使用以下方法：

- 使用符号“&&”将规则的多条命令写在同一行上。
- 使用特殊目标.ONESHELL。ONESHELL 可以使规则的所有命令在一个单独的 shell 进程调用。若提供了.ONESHELL，则只会检查命令的第一行是否含有特殊字符(如@、-、+等)，后续行将自动包含这些特殊字符。

4、并行执行

通常情况下，同一时刻只有一个命令在执行，下一命令需等待当前命令执行完之后才会开始执行，但是，可以使用-j 参数来告诉 make 工具在同一时刻可以允许多条命令被同时执行(这被称为并行执行)，但是，-j 参数对 MS-DOS 无效，因为 MS-DOS 是单任务系统。

示例 6.2：单个进程中执行规则的命令

①、g:/qt5/xx/makefile1 文件的内容如下：

```
#g:/qt5/xx/makefile1
bb:
    @echo B
```

②、g:/qt5/makefile2 文件的内容如下：

```
#g:/qt5/makefile2
aa:
    @cd xx
    mingw32-make -f makefile1
```

③、g:/qt5/makefile3 文件的内容如下：

```
#g:/qt5/makefile3
.ONESHELL:
aa:
    @cd xx
    mingw32-make -f makefile1
```

④、当执行 makefile2 文件时，以上示例会产生错误，这是因为规则的第一行命令@cd xx 与第二行命令是相互独立不相关的命令，第一行对目录的改变不会对第二行命令产生影响，所以，在第二行会出现找不到 makefile1 文件的错误。

⑤、当执行 makefile3 文件时，会输出 B。该示例使用了.ONESHELL 特殊目标，使规则的命令行中的命令位于同一个进程之中，所以，第一行命令@cd xx 对目录的改变会影响第二行的命令，从而使第二行在目录 xx 下找到了文件 makefile1。

6.2 递归调用(nmake 不适用)

为方便描述以及避免混淆，本小节把在 CMD(或 shell)中输入的 make 命令表述为“mingw32-make 命令”或称为“make 命令”。

6.2.1 递归调用与 MAKE 变量

在 makefile 文件中使用 make 工具来执行其他 makefile 文件的过程称为递归。递归调用时建议使用 MAKE 变量而不是直接使用 make 命令名(如 mingw32-make, nmake 等命令)。在规则的命令行中使用 MAKE 变量而不是 make 命令，可以使 MAKE 变量具有与在规则的命令行开头使用“+”字符相同的效果，也就是说使用带有 -t、-n 或 -q 参数的 mingw32-make 命令时，使用 MAKE 变量的规则命令会被执行，就像该命令行之前有“+”字符一样。但要注意的是，只有 MAKE 变量直接出现在规则的命令中才有这个效果，若是通过其他变量来引用的 MAKE 变量，则这个效果将不适用。下面以示例进行讲解。

示例 6.3: MAKE 变量

①、g:/qt5/makefile3 文件的内容如下：

```
#g:/qt5/makefile3(子 makefile)
bb:
    @echo B
```

②、g:/qt5/makefile2 文件的内容如下：

```
#g:/qt5/makefile2
aa:
    @echo A
    mingw32-make -f makefile3
    @echo C
```

③、g:/qt5/makefile1 文件的内容如下：

```
#g:/qt5/makefile1
aa:
    @echo A
    $(MAKE) -f makefile3
    @echo C
```

④、对于 makefile1 文件，在 CMD 中输入命令

```
mingw32-make -n -f makefile1
```

输出如下(注意，这些命令都不会被执行，只是显示了出来)：

```
echo A
mingw32-make -f makefile3
mingw32-make[1]: Entering directory 'G:/qt5'
echo B
mingw32-make[1]: Leaving directory 'G:/qt5'
echo C
```

可以看到，本示例显示了 mingw32-make -f makefile3 需要执行的规则命令 echo B，但未执行该命令，这说明 -n 参数也被传递给了 mingw32-make -f makefile3 命令。因此，本示例在使用 -n 参数的情况下也执行了 makefile1 中的 \$(MAKE) -f makefile3 命令。

⑤、对于 `makefile2` 文件，在 CMD 中输入命令

```
mingw32-make -n -f makefile2
```

输出如下(注意，这些命令都不会被执行，只是显示了出来)

```
echo A
mingw32-make -f makefile3
echo C
```

可以看到，没有显示递归调用需执行的命令 `echo B`，这说明在使用 `-n` 参数的情况下不会执行 `makefile2` 中的 `mingw32-make -f makefile3` 命令。

⑥、若对 `makefile1` 输入

```
mingw32-make -t -f makefile1
```

则会创建两个空文件 `aa` 和 `bb`，这说明执行了命令 `$(MAKE) -f makefile3`，并且还带有 `-t` 参数。

而对 `makefile2` 输入

```
mingw32-make -t -f makefile2
```

只会创建一个空文件 `aa`，这说明未执行命令 `mingw32-make -f makefile3`

6.2.2 传递变量给子 make (export、unexport 指示符)

- 1、默认情况下，若变量未被明确指定可以被传递(或称为被导出)，则上层 `make` 不会将其所执行的 `makefile` 文件中定义的变量传递给子 `make` 进程，但是，上层 `make` 会将已经存在的环境变量和使用命令行指定的变量传递给子 `make` 进程，通常这些变量只能由数字、字母和下划线组成。
- 2、使用 `export` 指示符声明的变量将被传递给子 `make` 进程，即，使用 `export` 声明的变量可以被导出。上层传递给子 `make` 进程的变量不会覆盖子 `make` 进程所执行 `makefile` 文件中的同名变量定义，除非使用 `-e` 参数。

3、`export` 的语法如下：

```
export 变量名或变量定义 ...
```

4、`unexport` 指示符

若不希望某个变量被传递给子 `make`，可使用 `unexport` 指示符，其语法如下：

```
unexport 变量名或变量定义 ...
```

示例 6.4: export 指定符的用法

①、`g:/qt5/makefile2` 文件的内容如下：

```
#g:/qt5/makefile2(子 makefile)
AA=ss
bb:
    @echo AA=$(AA)
    @echo BB=$(BB)
    @echo DD=$(DD)
```

②、`g:/qt5/makefile` 文件的内容如下：

```
#g:/qt5/makefile(主 makefile)
```

```
export AA=xx #export 用法一
BB=yy
export BB    #export 用法二
DD=zz
aa:
    @echo A=
    mingw32-make -f makefile2
```

③、在 CMD 中转至 g:/qt5，并输入以下命令执行主 makefile 文件

```
mingw32-make
输出
A=
mingw32-make -f makefile2
mingw32-make[1]: Entering directory 'G:/qt5'
AA=ss
BB=yy
DD=
mingw32-make[1]: Leaving directory 'G:/qt5'
```

由输出可见，在主 makefile 中定义的变量 AA 未覆盖子 makefile2 中定义的同名变量 AA，主 makefile 中定义的变量 BB 被成功的导出，而主 makefile 中定义的 DD，由于未被 export 声明，所以未被导出。

5、没有任何参数的 export 指示符(即单独使用 export)表示导出没有在 export 或 unexport 中明确声明的所有变量，这是老版本 make 的默认行为，在新版本中应使用特殊目标

.EXPORT_ALL_VARIABLES

来代替，但是，若变量名中含有特殊字符(除字母、数字和下划线以外的字符)的变量可能不会被导出，对这种特殊命名的变量，需要使用 export 明确声明。不带任何参数的 unexport 指示符的意义与 export 类似，由于单独使用 unexport 是默认行为，除非在之前单独使用过 export，否则没有实际意义。

6、最后一个出现的 export 或 unexport 指示符决定整个 make 运行过程中变量是否被导出。

7、export 或 unexport 如果有对变量或函数的引用，则会被立即展开。

示例 6.5：不带参数的 export 指示符

①、g:/qt5/makefile3 文件的内容如下：

```
#g:/qt5/makefile2(子 makefile)
bb:
    @echo BB=$(BB)
    @echo DD=$(DD)
```

②、g:/qt5/makefile2 文件的内容如下：

```
#g:/qt5/makefile2(主 makefile)
export BB=yy
DD=zz
```

```
unexport
aa:
    @echo A=
    @mingw32-make -f makefile3
```

③、g:/qt5/makefile1 文件的内容如下：

```
#g:/qt5/makefile1(主 makefile)
export BB=yy
DD=zz
export
aa:
    @echo A=
    @mingw32-make -f makefile3
```

④、在 CMD 中转至 g:/qt5，并输入以下命令执行 makefile1 文件

```
mingw32-make -f makefile1
```

输出

```
A=
mingw32-make[1]: Entering directory 'G:/qt5'
BB=yy
DD=zz
mingw32-make[1]: Leaving directory 'G:/qt5'
```

可见，未被 export 明确声明的变量 DD 被不带参数的 export 指示符成功导出。

⑤、在 CMD 中转至 g:/qt5，并输入以下命令执行 makefile2 文件

```
mingw32-make -f makefile2
```

输出

```
A=
mingw32-make[1]: Entering directory 'G:/qt5'
BB=yy
DD=
mingw32-make[1]: Leaving directory 'G:/qt5'
```

可见，单独的 unexport 声明并未影响被 export 明确声明的变量 BB，变量 BB 的值仍被导出了。

8、特殊变量

1)、若变量是由 make 默认创建的，通常不会向下传递。特殊变量 MAKEFLAGS 会在整个 make 的执行过程中始终被自动的传递给所有的子 make，除非对这个变量明确使用 unexport 进行声明，MAKEFLAGS 的具体讲解详见下一小节。另外特殊变量 MAKEFILES 和 SHELL 也有特殊处理，本文从略。

2)、特殊变量 MAKELEVEL 代表递归调用的深度，最上级的值为 0，下一级为 1，...，以此类推。下面以示例讲解。

示例 6.6：使用特殊变量 MAKELEVEL 获取递归调用的深度

- ①、g:/qt5/makefile2 文件的内容如下：

```
#g:/qt5/makefile2(子 makefile)
bb:
    @echo BB=$(MAKELEVEL)
```

- ②、g:/qt5/makefile 文件的内容如下：

```
#g:/qt5/makefile (主 makefile)
aa:
    @echo A=$(MAKELEVEL)
    @mingw32-make -f makefile2
```

- ③、在 CMD 中转至 g:/qt5，并输入以下命令执行主 makefile 文件

```
mingw32-make
```

输出

```
A=0
mingw32-make[1]: Entering directory 'G:/qt5'
BB=1
mingw32-make[1]: Leaving directory 'G:/qt5'
```

其中 AA=0，表示主 makefile 的深度为 0，BB=1 表示子 makefile 的深度为 1，本示例只调用了两级深度。

6.2.3 传递 make 命令参数给子 make(特殊变量 MAKEFLAGS)

- 1、在命令行(如 CMD)上定义的变量和 make 命令参数都会自动通过特殊变量 MAKEFLAGS 传递给子 make，除了 -C，-f，-o，-W 参数之外。
- 2、由于 make 命令参数是自动传递的，所以，若不希望传递 make 命令参数，需要在调用子 make 时对变量 MAKEFLAGS 的值赋空。
- 3、如果在调用子 make 时也指定了 make 命令参数，则该参数并不会清除主 make 命令参数并重新设置为指定的参数，而是将二者合并，所以，若不希望使用主 make 传递进来的参数，应在调用子 make 时对变量 MAKEFLAGS 进行明确的赋值。
- 4、另外需要注意的是，子 make 命令会默认包含一个 -w 参数。由于 MS-DOS 不支持并行，本文不对 -j 参数作讲解。
- 5、为了兼容老版本，MFLAGS 变量被保留了下来，为了与其他版本兼容增加了 GNUMAKEFLAGS 变量。这两个变量与 MAKEFLAGS 变量的功能类似，MFLAGS 不支持传递命令行变量，而 GNUMAKEFLAGS 会在 MAKEFLAGS 变量之前被解析，当 make 解析 MAKEFLAGS 时，会读取 GNUMAKEFLAGS 中设置的参数，同时，解析完 GNUMAKEFLAGS 之后，会将该变量的值清空以防在递归调用时被重复标记。
- 6、最后，建议不要随意更改变量 MAKEFLAGS 的值，以免引起奇怪的结果，更不要在环境变量中设置 MAKEFLAGS 的值。

示例 6.7：使用 MAKEFLAGS 变量传递命令行参数

- ①、g:/qt5/makefile2 文件的内容如下：

```
#g:/qt5/makefile2(子 makefile)
bb:
    @echo BB=$(MAKEFLAGS)
```

- ②、g:/qt5/makefile 文件的内容如下：

```
#g:/qt5/makefile(主 makefile)
aa:
    @echo A=$(MAKEFLAGS)
    @mingw32-make -f makefile2 -s
```

- ③、在 CMD 中转至 g:/qt5，并输入以下命令执行主 makefile 文件

```
mingw32-make -w -k
```

输出

```
mingw32-make: Entering directory 'G:/qt5'
A=kw
mingw32-make[1]: Entering directory 'G:/qt5'
BB=ksw
mingw32-make[1]: Leaving directory 'G:/qt5'
mingw32-make: Leaving directory 'G:/qt5'
```

可见，CMD 中输入的参数 -w -k 与调用子 make 时设置的参数 s 被合并之后传递给了子 make。若希望子 make 仅使用 -s 参数而不接受主 make 传递进来的参数，应将主 makefile 最后一行的代码修改为以下代码

```
@mingw32-make -f makefile2 MAKEFLAGS=s
```

7、特殊变量 MAKEOVERRIDES

虽然在命令行上定义的变量也会通过 MAKEFLAGS 传递给子 make，但其实命令行的变量定义出现在特殊变量 MAKEOVERRIDES 中，只是 MAKEFLAGS 引用了该变量，所以，如果希望只传递命令行参数而不传递命令行变量定义，可以将 MAKEOVERRIDES 重置为空。

示例 6.8：使用 MAKEOVERRIDES 传递命令行上定义的变量

- ①、g:/qt5/makefile2 文件的内容如下：

```
#g:/qt5/makefile2(子 makefile)
bb:
    @echo $(BB)=$(MAKEFLAGS)
    @echo EE=$(MAKEOVERRIDES)
```

- ②、g:/qt5/makefile 文件的内容如下：

```
#g:/qt5/makefile(主 makefile)
aa:
    @echo A=$(MAKEFLAGS)
    @echo AA= $(MAKEOVERRIDES)
```

```
@mingw32-make -f makefile2
```

③、在 CMD 中转至 g:/qt5，并输入以下命令执行主 makefile 文件

```
mingw32-make BB=xx
```

输出

```
A= -- BB=xx
```

```
AA= BB=xx
```

```
mingw32-make[1]: Entering directory 'G:/qt5'
```

```
xx=w -- BB=xx
```

```
EE=BB=xx
```

```
mingw32-make[1]: Leaving directory 'G:/qt5'
```

第 7 章 推理规则和内联文件(仅适用于 nmake)

7.1 推理规则(仅适用于 nmake)

7.1.1 基本语法及原理

- 1、推理规则的作用是当一个规则的目标的依赖文件不存在并且不是另一个有命令的规则的目标时(称为依赖缺失或缺失依赖), 则按照推理规则的原理, 使用推理规则的命令来创建该缺失的依赖文件。
nmake 的推理规则与 GNU make 的后缀规则类似。

- 2、推理规则的语法如下:

```
{from_path}.from_ext{to_path}.to_ext:  
    命令
```

其中, from_path 和 to_path 分别是依赖文件和目标文件的路径, from_ext 是依赖文件后缀名, to_ext 是目标文件后缀名, 后缀名不区分大小写, 可以引用变量来表示后缀名, 除了冒号之前可以有空格外, 冒号之前的其他部分都不能有空格, 冒号之后只能是空格、分号(用于指定命令)、注释符号“#”, 或换行符。若要指定当前目录可使用空的大括号“{ }”或直接不要大括号以句点开始。

- 3、推理规则执行原理如下:

若目标的依赖缺失, 则尝试使用推理规则的命令来创建该依赖文件。其具体步骤为: 检查依赖文件的路径和后缀名是否分别与推理规则的 to_path 和 to_ext 匹配, 若匹配, 则推理规则在路径 from_path 中查找具有与依赖文件相同基本名称(即, 除后缀之外的名称)的另一文件, 然后根据此文件创建缺失的依赖文件。可见, 推理规则不会主动执行, 需要满足一定的条件才会执行推理规则。下面列举一示例进行说明

示例 7.1: 推理规则

①、源文件准备

在 g:\qt5\目录下的 a.cpp 中输入以下代码

```
//g:\qt5\a.cpp(主源文件)  
#include<iostream>  
extern int b;  
int main(){    std::cout<<"A"<<std::endl; return 0;}
```

在 g:\qt5\目录下的 b.cpp 中输入以下代码

```
//g:\qt5\b.cpp  
int b=1;
```

②、在 g:\qt5\目录下的 makefile 中输入以下代码

```
#g:\qt5\makefile
{}.cpp{}.obj:
    @echo AAAAAA
    cl /c /nologo /EHsc $<
xx:a.obj b.obj
    @echo BBBB
yy:
    @echo CCCC
a.obj:
b.obj:
```

③、代码说明

makefile 文件中的预定义变量\$<只能用于推理规则中，表示比当前目标更新的所有依赖。参数/EHsc和/nologo用于减少输出不必要的内容，@echo AAAAAA用于测试是否执行了该推理规则。**makefile** 文件中的最后两行代码 a.obj:和 b.obj:，由于该两行代码没执行任何命令，所以不会影响推理规则的执行。

④、在 CMD 中转至 g:\qt5，并输入以下命令执行主 **makefile** 文件

```
nmake /nologo

输出

AAAAAA
      cl /c /nologo /EHsc .\a.cpp
a.cpp
AAAAAA
      cl /c /nologo /EHsc .\b.cpp
b.cpp
BBBB
```

可以看到，该命令执行了推理规则的命令，目标 xx 缺失依赖文件 a.obj 和 b.obj，根据 a.obj 和 b.obj 所在路径(即当前目录)和后缀名.obj 找到匹配的推理规则，然后推理规则从 from_path(本例为当前目录)检查具有与 a.obj 和 b.obj 相同基本名称的文件，找到 a.cpp 和 b.cpp，然后执行推理规则的命令创建缺失的依赖文件 a.obj 和 b.obj，我们可以在目录 g:\qt5 下看到创建的 a.obj 和 b.obj 文件。若当前目录没有与 a.obj 和 b.obj 相同基本名称的文件，则不会执行推理规则的命令。

⑤、然后接着在 CMD 中输入以下命令

```
nmake /nologo yy
```

由于目标 yy 不存在缺失依赖文件的情形，所以未执行推理规则。

4、批处理推理规则

批处理推理规则与标准推理规则的唯一语法区别是以双冒号结尾，在示例 7.1 中的 **makefile** 文件中的第 1 行的末尾添加一个冒号(即，使用双冒号)，然后把之前生成的 a.obj 和 b.obj 删除，再在 cmd 中输入

```
nmake /nologo

输出

AAAAAA
```

```
cl /c /nologo /EHsc .\a.cpp .\b.cpp
a.cpp
b.cpp
正在生成代码...
BBBB
```

可以看到，批处理推理规则只使用了一条 `cl` 命令(即一次性执行)，而标准推理规则使用了两条 `cl` 命令，这就是标准推理规则与批处理推理规则的区别。

7.1.2 预定义推理规则

- 1、`nmake` 为 `.c`, `.cpp` 等常见后缀名定义了一些预定义的推理规则，可以在 CMD 中使用 `nmake /p` 查看预定义的推理规则。预定义推理规则类似于 GNU `make` 隐式的后缀推理规则。
- 2、当没有合适的推理规则时，会执行预定义的推理规则来创建缺失的依赖文件。
- 3、以 `.cpp` 为例，预定义推理规则的代码如下，其余后缀名的代码与以下代码类似，其中预定义变量 `CPP` 的值为 `cl`，变量 `CPPFLAGS` 用于指定参数，通常该变量未定义。

```
.cpp.obj::
    $(CPP) $(CPPFLAGS) /c $<
.cpp.exe:
    $(CPP) $(CPPFLAGS) $<
```

- 4、点指令 `.SUFFIXES` 列出了预定义推理规则匹配的后缀名列表，可以使用 `nmake /p` 查看 `.SUFFIXES` 列表。
- 5、特别注意：自定义推理规则的后缀名需要添加到 `.SUFFIXES` 列表，否则，不会执行自定义的推理规则。
- 6、清除、添加、更改 `.SUFFIXES` 列表可分别在 `makefile` 文件中使用以下语法
 - 1)、清除列表
`.SUFFIXES:`
 - 2)、添加列表
`.SUFFIXES:后缀名列表`
 - 3)、更改列表(需要先清除再添加)
`.SUFFIXES:`
`.SUFFIXES:指定的后缀名`
- 7、清除了 `.SUFFIXES` 列表也就意味着清除了所有预定义的推理规则。
- 8、禁用预定义推理规则的方法
要想不执行预定义的推理规则，可以在 `makefile` 文件中清除 `.SUFFIXES` 列表。

7.2 内联文件(仅适用于 nmake)

1、内联文件是在 **makefile** 中输入该文件的内容并使用这些内容创建的文件，内联文件可以用作 **cl**、**link** 等命令的输入文件(也被称为命令行文件)。

2、内联文件在 **makefile** 规则的命令行中使用以下语法创建

```
命令 <<[文件名 1] <<[文件名 2] ...  
文件名 1 的内容  
<<[KEEP|NOKEEP]  
文件名 2 的内容  
<<[KEEP|NOKEEP]  
...
```

- 1)、内联文件在 **makefile** 规则的命令行以双尖括号 “<<” 开头，并在单独的一行的开头使用双尖括号 “<<” 标记该内联文件的结尾。
- 2)、若指定文件名，则在双尖括号 “<<” 后不能有空格，文件名可以使用路径。
- 3)、若指定了文件名，则将在当前目录或指定的目录创建该内联文件，并会覆盖现有的文件，若未指定文件名，则会在 **TMP**(临时目录)创建内联文件。
- 4)、若指定了以前的重复的文件名，则替换之前的文件。若想保留创建的内联文件，则在结尾的双尖括号 “<<” 后指定 **KEEP**。若想要创建的内容文件在其他地方被引用，则应指定文件名。

3、内联文件的内容不能有指令或 **makefile** 注释。

示例 7.2：内联文件

```
all:  
    @cl @<<g:\qt5\xx.txt  
a.cpp b.cpp  
/EHsc  
<<keep
```

其中第一个@是 **makeline** 的命令，表示不回显 **makefile** 的命令，第 2 个@是 **cl** 的命令，表示为 **cl** 指定一个输入文件(**link** 命令也是使用@指定输入文件)，然后输入文件的内容会被用作 **cl** 命令的参数。以上示例表示，将第一个双尖括号下一行至第二个双尖括号之间的内容输入到 **g:\qt3\xx.txt** 文件中，然后将该文件的内容作为 **cl** 的参数，因此，以上 **makefile** 的代码等价于以下代码，

```
all:  
    @cl a.cpp b.cpp /EHsc
```

若存在源文件 **a.cpp** 和 **b.cpp**，则执行以上 **makefile** 文件后会在目录 **g:\qt5** 中能找到一个名为 **xx.txt** 的文件，其中的内容便是 **cl** 的参数。还能在 **g:\qt5** 目录下找到生成的 **a.obj**、**b.obj** 和 **a.exe** 三个文件。

4、指定部分文件名(仅适用于 **nmake**)

- 1)、完整的文件名由 4 部分组成，即驱动器名、路径、基本名称、后缀名(扩展名)。
- 2)、**nmake** 指定部分文件名的语法用于指示第一个依赖文件的名称，使用 **%s** 或 **%F** 可以表示完整的文件名称，使用 **%[[parts]F** 表示文件名的各部分名称，其中 **parts** 可取值为 **d**、**p**、**f**、**e**。

示例 7.3：指定部分文件名

```
#g:\qt5\makefile  
all:G:\qt5\a.txt G:\qt5\m.cpp
```

```
@echo %s
@echo %|F
@echo %|dF
@echo %|pF
@echo %|fF
@echo %|eF
```

以上示例假设在 g:/qt5 目录下存在 a.txt 和 m.cpp 文件，因此，其代码中的

```
%s 或 %F = G:\qt5\a.txt
%|dF = G:
%|pF = \qt5\
%|fF = a
%|eF = .txt
```

在 CMD 中转至 g:/qt5，并输入以下命令，

```
nmake
```

输出

```
G:\qt5\a.txt
G:\qt5\a.txt
G:
\qt5\
a
.txt
```

作者：黄邦勇帅(原名：黄勇)

2023 年 10 月 20 日



总结

表 1 mingw32-make 命令各参数总结

参数	说明
-B	强制重建规则的目标
-C DIR	切换到目录 DIR 之后执行 make
-d	显示调试信息
-e	使环境变量定义覆盖 makefile 中的同名变量
-f file	执行指定的文件
-h	显示帮助信息
-i	忽略规则命令执行的错误
-I DIR	在 DIR 目录下搜索 include 指定的文件
-j	指定并行执行的命令数目
-k	执行命令错误时不终止执行
-n	打印出规则的命令但不执行
-o file	指定的文件 file 不需重建
-p	命令执行前，显示隐式规则，隐式变量等信息
-q	询问模式，返回 make 执行结果编号，不执行规则的命令也不输出内容
-r	取消隐式规则及隐式后缀规则
-R	取消隐式变量，同时开启-r 参数
-s	禁用命令回显
-t	修改目标的时间戳，但不修改文件内容
-w	打印执行的 makefile 文件所在的目录
-W file	逻辑上修改指定文件的时间戳，但实际上未修改

表 2 特殊符号总结

符号	说明
@	在规则命令前表示禁止回显该命令。 用于 cl 或 link 命令指定输入文件，并将文件中的内容作为命令的参数
&&	用于把规则的多个命令连接在一行上
-	忽略错误信息
+	即使 make 命令带有-t, -p, -n 参数，该规则的命令也会执行
<<	用于 nmake 创建内联文件
>	将左侧的结果输出到右侧的文件中，如 echo A>a.txt，在 a.txt 中输入字符 A
>>	将左侧的结果追加到右侧文件中
-lname	作为依赖搜索库文件
-include 或 sinclude	忽略错误信息，除此之外，与 include 效果相同

表 3 特殊目标总结

特殊目标用法	说明
.ONESHELL:	使多条规则命令在同一个 shell 进程运行
.EXPORT_ALL_VARIABLES:	用于代替单独使用的 export
.INTERMEDIATE: 依赖文件	把依赖文件标记为中间文件
.NOTINTERMEDIATE: 依赖文件	取消中间文件
.SECONDARY: 依赖文件	保留指定的文件不被自动删除

.DEFAULT::命令	指定默认规则
.SUFFIXES:后缀名	修改可识别后缀列表

表 4 特殊变量总结

变量名	说明
.RECIPEPREFIX	指定命令行的开头前缀，如.RECIPEPREFIX =>表示命令行以>开始
SUFFIXES	查看所有的可识别后缀列表
MAKEFILE_LIST	已读取的 makefile 文件名列表
MAKELEVEL	递归调用的深度，最上级的值为 0
MAKEFLAGS	向子 make 传递命令行(如 CMD)上定义的变量和 make 命令参数
.LIBPATTERNS	当存在“-lname”这样的依赖时，使用 name 替换.LIBPATTERNS 变量的值中的%，并将替换后的名称作为库文件名搜索目录

第3篇 CMake 目录

第3篇 CMake	128
第1章 CMake 基础	128
1.1 CMake 基本设置	128
1.2 一个简单的示例	129
1.2.1、使用 CMake 构建一个 C++ 程序	129
1.2.2 分析 CMake 生成的文件	130
1.3 CMake 相关基本概念	132
1.3.1 CMake 文件	132
1.3.2 CMake 构建程序的过程	132
1.3.3 CMake 目录结构	133
1.3.4 cmake 命令行工具的使用	133
第2章 CMake 基本命令	136
2.1 CMake 基本语法	136
2.1.1 CMake 命令的参数	136
2.1.2 注释、变量、列表	138
2.1.3 生成器表达式 <code>\$<TARGET_OBJECTS:..></code>	139
2.2 <code>cmake_policy()</code> 命令	140
2.3 <code>cmake_minimum_required()</code> 命令	143
2.4 <code>set()</code> 命令	143
2.4.1 常规变量	143
2.4.2 环境变量	144
2.4.3 缓存变量	145
2.5 <code>function()</code> 、 <code>macro()</code> 、 <code>block()</code> 、 <code>return()</code> 命令	146
2.5.1 <code>function()</code> 命令	146
2.5.2 <code>macro()</code> 命令	147
2.5.3 <code>block()</code> 命令	148
2.5.4 <code>return()</code> 命令	148
2.6 <code>if()</code> 命令(条件语句)	150
2.6.1 基本语法	150
2.6.2 基本的条件表达式	151
2.6.3 逻辑表达式	153
2.6.4 比较	153
2.6.5 存在性检测	154
2.6.6 文件或目录	154
2.7 <code>foreach()</code> 命令	155
2.7.1 <code>foreach</code> 常规形式	155
2.7.2 <code>foreach</code> 变体 1	156

2.7.3 foreach 变体 2	156
2.7.4 foreach 变体 3	156
2.7.5 foreach 变体 4	157
2.8 while() 命令	158
2.9 break() 和 continue() 命令	158
2.10 option() 命令	159
2.11 unset() 命令	159
第 3 章 CMake 属性	160
3.1 set_property() 和 get_property() 命令	160
3.1.1 set_property() 命令	160
3.1.2 get_property() 命令	162
3.2 define_property() 命令	163
3.3 其他属性命令	166
3.3.1 set_target_properties() 命令	166
3.3.2 set_directory_properties() 命令	166
3.3.3 set_source_files_properties() 命令	166
3.3.4 set_tests_properties() 命令	167
3.3.5 get_cmake_property() 命令	167
3.3.6 get_target_property() 命令	167
3.3.7 get_directory_property() 命令	167
3.3.8 get_source_file_property() 命令	167
3.3.9 get_test_property() 命令	168
第 4 章 CMake 项目	169
4.1 add_subdirectory() 命令	169
4.1.1 CMake 项目的结构	169
4.1.2 add_subdirectory() 命令	169
4.2 project() 命令	174
4.2.1 project() 命令	174
4.2.2 project() 命令调用期间的执行步骤	178
第 5 章 使用 CMake 构建库文件和可执行文件	180
5.1 CMake 目标的分类	180
5.2 add_library() 命令	181
5.2.1 add_library() 命令基本语法	181
5.2.2 对象库目标(简称对象库)	185
5.2.3 接口库目标(简称接口库)	185
5.2.4 导入库目标(简称导入目标)	187
5.2.5 库目标的别名(别名库目标)	189
5.3 add_executable() 命令	190
5.3.1 基本语法	190
5.3.2 导入可执行目标	190
5.3.3 可执行目标的别名	191
第 6 章 使用要求及编译参数	192
6.1 使用要求的基本概念	192
6.2 设置编译参数(COMPILE_OPTIONS 系列属性)	193

6.2.1 总览	193
6.2.2 target_compile_options()和 add_compile_options()命令	194
6.3 设置-D 编译参数(COMPILE_DEFINITIONS 系列属性)	200
6.3.1 总览	200
6.3.2 target_compile_definitions()和 add_compile_definitions()命令	201
6.4 设置编译特性(COMPILE_FEATURES 系统属性)	202
6.4.1 总览	202
6.4.2 target_compile_features()命令	203
6.4.3 <LANG>_STANDARD 目标属性以及与其相关的属性和变量	204
6.4.4 <LANG>_EXTENSIONS 目标属性以及与其相关的属性和变量	204
第 7 章 使用要求的传递	207
7.1 target_link_libraries()命令简介	207
7.2 传递使用要求的基本规则	209
7.2.1 使用要求传递的基本规则	209
7.2.2 本文的名称约定	212
7.3 链接依赖项(向链接命令添加文件).....	214
7.3.1 链接依赖项(向链接命令添加文件)概述	214
7.3.2 链接直接依赖项的规则	215
7.3.3 链接间接依赖项的规则	216
7.3.4 依赖项为生成器表达式的链接	218
7.3.5 链接依赖项总结	218
7.4 传递使用要求的详细规则	220
7.4.1 概述	220
7.4.2 传递使用要求的详细规则(INTERFACE_LINK_LIBRARIES 和 LINK_LIBRARIES 属性).....	220
7.5 对 target_link_libraries()命令进一步的讲解	223
7.5.1 target_link_libraries(PUBLIC)	223
7.5.2 target_link_libraries(PRIVATE)	224
7.5.3 target_link_libraries(INTERFACE)	227
7.6 INTERFACE_LINK_LIBRARIES_DIRECT 和 INTERFACE_LINK_LIBRARIES_DIRECT_EXCLUDE 目标属性	230
第 8 章 向 CMake 项目添加其他文件	233
8.1 添加源文件	233
8.1.1 添加源文件的属性和变量汇总	233
8.1.2 SOURCES 和 INTERFACE_SOURCE 目标属性.....	233
8.1.3 GENERATED 源文件属性	235
8.1.4 target_sources()命令--->常规语法	236
8.1.5 target_sources()命令--->文件集	237
8.1.6 aux_source_directory()命令	240
8.2 添加头文件包含目录(-I 编译参数)	240
8.2.1 总览	240
8.2.2 target_include_directories()命令	241
8.2.3 include_directories()命令	242
8.3 添加链接文件的库目录	245
8.3.1 总览	245

8.3.2 target_link_directories()命令	246
8.3.3 link_directories ()命令	247
8.3.4 link_libraries()命令	247
8.4 include()和 add_dependencies()命令	248
8.4.1 include()命令	248
8.4.2 add_dependencies()命令	249
第 9 章 使用 install()命令安装文件	250
9.1 使用变量或目标属性将程序文件输出到指定目录	250
9.1.1 输出工件(Output Artifacts)	250
9.1.2 将程序文件输出到指定目录的方法	251
9.1.3 使用 install()命令安装文件的方法	253
9.2 install(TARGETS) 安装目标文件	254
9.2.1 intstall()命令输出工件的类型	255
9.2.2 install()命令的默认安装路径	261
9.2.3 COMPONENT、EXCLUDE_FROM_ALL、OPTION 参数.....	264
9.2.4 PERMISSIONS、CONFIGURATIONS 参数.....	266
9.2.5 NAMELINK_ONLY、NAMELINK_SKIP、NAMELINK_COMPONENT 参数.....	266
9.2.6 INCLUDES DESTINATION [<dir> ...]参数	267
9.2.7 EXPORT<export-name>参数.....	268
9.2.8 RUNTIME_DEPENDENCIES、RUNTIME_DEPENDENCY_SET 参数.....	268
9.3 install(EXPORT) 安装导出	268
9.4 install(RUNTIME_DEPENDENCY_SET) 安装运行时依赖项.....	268
9.5 install(IMPORTED_RUNTIME_ARTIFACTS) 安装导入目标的运行时工件.....	271
9.6 install(<FILES PROGRAMS>) 安装文件	272
9.7 install(DIRECTORY) 安装目录.....	274
9.8 install(SCRIPT)和 install(CODE) 安装脚本和代码.....	276
第 10 章 导入目标和导出目标	278
10.1 导入目标	278
10.1.1 基础	278
10.1.2 设置导入目标的详细信息	278
10.2 导出目标	283
10.2.1 CMake 导出目标的思维	283
10.2.2 install(EXPORT)命令.....	285
10.2.3 export(TARGETS)和 export(EXPORT)命令	289
第 11 章 configure_file()命令(生成配置文件).....	295
第 12 章 find_file()命令(查找文件).....	298
12.1 find_file()命令的语法及基本参数	298
12.2 find_file()命令的搜索顺序	301
12.3 搜索指定根目录下的子目录	307
12.4 与搜索顺序有关的所有变量	309
12.4.1 开关变量	309
12.4.2 路径变量	310
12.5 其余的 find_*命令.....	312
第 13 章 find_package()命令(查找包).....	315

13.1 前提基础知识	315
13.1.1 何为包?	315
13.1.2 CMake 包	315
13.1.3 怎样编写 CMake 包	315
13.1.4 CMake 包由谁提供	316
13.1.5 find_package()命令的作用	316
13.1.6 find_package()命令的格式及搜索模式.....	316
13.2 find_package()基本命令	318
13.3 find_package()完整命令	322
13.3.1 find_package()完整命令语法及基本参数.....	322
13.3.2 配置模式的搜索目录	324
13.3.3 配置模式的搜索前缀及搜索顺序	326
13.3.4 搜索指定根目录下的子目录	329
13.3.5 处理版本信息及版本文件变量	330
13.3.6 包(文件)接口变量	335
13.3.7 find_package()命令中各变量小结.....	337
13.4 使用 CMake 自带的模块 CMakePackageConfigHelpers 生成配置文件和版本文件.....	338
13.4.1 configure_package_config_file()命令	338
13.4.2 使用 write_basic_package_version_file()命令生成版本文件.....	344
第 14 章 使用 CMake 构建 Qt 程序	351
14.1 使用 CMake 构建一个简单的 Qt 程序	351
14.2 CMake 自动调用 uic.exe 工具生成头文件的过程及原理.....	353
14.2.1 Qt 构建工具	353
14.2.2 CMake 自动调用 uic.exe 工具生成头文件的过程及原理	354
14.3 CMake 自动调用 moc.exe 工具处理 Qt 的元对象系统	359
14.3.1 moc 简介	359
14.3.2 Qt 专有宏位于头文件中	359
14.3.3 Qt 专有宏位于源文件中	362
14.3.4 与 moc 有关的其他变量和属性	364
14.4 使用 Qt 提供的 CMake 命令构建 Qt 程序.....	365
14.4.1 qt_standard_project_setup()命令	366
14.4.2 qt_add_executable()命令	367
14.4.3 qt_add_library()命令	368
14.4.4 qt_finalize_target()命令	368
14.5 在 Qt 中使用 CMake 构建 Qt 程序	369
第 15 章 交叉编译(指定编译器等工具).....	375

第 3 篇 CMake

说明:

本文需熟悉 `makefile` 文件的相关知识以及使用命令工具构建 C++ 程序, 如使用 `g++`、`cl`、`link` 等, 特别应对 `dll` 文件的生成过程有所了解, 若读者不熟悉, 可参阅前文讲解。

由于各系统的目录分隔符各不相同, 本文中的 “/” 和 “\” 都表示目录分隔符, 但尽量使用正斜杠 “/”。

本文的 C++ 示例代码都是非常简单的代码, 若示例中出现未列出示例代码的情形, 读者可自行准备 C++ 源文件。

第 1 章 CMake 基础

1.1 CMake 基本设置

- 1、CMake 是用于生成 `makefile` 文件的工具, CMake 功能强大, 更适用于大型项目的管理, CMake 有自己的语法规则, 因此, 本文讲的 CMake 分为两个部分, 一部分是根据 CMake 语法规则编写的 CMake 文件, 一部分是解释该文件的 `cmake.exe` 命令行工具。CMake 的语法是基于命令的, 也就是说, CMake 文件的任何代码都是以命令的形式出现的。
- 2、CMake 主文件的名称为 “CMakeLists.txt”, 这是一个纯文本文件, CMake 根据该文件生成相应的 `makefile` 文件。
- 3、由于使用 `makefile` 的目的是创建目标并当依赖更新 (或目标过时) 时更新目标, 所以, CMake 的目的也是创建目标, 并建立目标的依赖关系, 当依赖更新时更新目标。目标可以是可执行程序(使用 `add_executable()` 命令构建)、库文件(使用 `add_library()` 命令构建)等。由于 CMake 和 `makefile` 都有 “目标” 这一概念, 本文若无特别说明, “目标” 一词都是指的 CMake 目标。
- 4、生成器(generator)
 - 1)、CMake 将解释 `makefile` 文件的构建程序称为生成器或生成工具, 如 `mingw32-make`、`make`、`ninja`、`jom`、`nmake` 等, 都叫做生成器; 除了能生成 `makefile` 文件之外, CMake 还能生成 Visual Studio 的项目文件、Ninja 文件等其他文件, 因此, Visual Studio 也被 CMake 称为生成器(但同时也是编译器)。本文讲解使用 CMake 生成 `makefile` 文件, 使用 `mingw32-make` 生成器。
 - 2)、生成器可使用以下方式指定
 - 在命令行中使用 `cmake.exe` 工具的 -G 参数指定, 比如

```
cmake . -G "MinGW Makefiles"
```
 - 创建一个以下的系统环境变量

```
CMAKE_GENERATOR = MinGW Makefiles
```

建议使用环境变量指定生成器, 以免每次都要指定生成器。注意: 若安装有 VS, 会首先默认生成 VS 的项目文件(proj 文件), 而不会生成 `makefile` 文件。
 - 注意: 不要在 CMake 项目(即 CMakeLists.txt 文件)中使用 CMAKE_GENERATOR 变量指定生成器, 该变量在 CMake 文件中使用是无效的。

- 5、编译工具(如 cl、g++等)等其他工具的选择和指定方法详见第 15 章对交叉编译的讲解。
- 6、本文使用的工具及其版本情况
- 本文使用的操作系统为 windows10 22H2(64bit)
 - cmake 的版本为 CMake 3.27.0-rc4
 - 本文使用 mingw32-make 生成器，其版本为 GNU Make 4.2.1。
 - 本文使用的编译器为 VC++，其版本为 Visual Studio 2022，主要使用其中的 cl、link 等命令工具。编译器由 CMake 根据当前系统自动选择，若要指定自己的编译器，则需要手动配置比较多的 CMake 变量，详见本文章末。

1.2 一个简单的示例

1.2.1、使用 CMake 构建一个 C++ 程序

1、C++源代码准备

在 g:/qt1 目录下的 a.cpp 和 b.cpp 中编写如下代码

代码清单 1-1: g:/qt1/a.cpp

```
#include<iostream>
extern int b;
int main(){
    std::cout<<"E="<<b<<std::endl;    return 0;}
```

代码清单 1-2: g:/qt1/b.cpp

```
int b=1;
```

2、编写 CMake 代码

然后在 g:/qt1 目录下的 CMakeLists.txt 文件中编写如下代码

代码清单 1-3: g:/qt1/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.27)    #指定 CMake 的最低需求版本
project (hello)                          #指定项目名称
add_executable(xxx a.cpp b.cpp)          #使用文件 a.cpp、b.cpp 构建一个名为 xxx 的可执行程序
```

3、使用各种命令工具构建程序

- 1)、在使用 cmake 命令之前，需在环境变量增加一个 CMAKE_GENERATOR = MinGW Makefiles 的环境变量，否则，若安装有 VS，会首先默认生成 VS 的项目文件(proj 文件)，而不会生成 makefile 文件。
- 2)、然后在 CMD 中转到 CMakeLists.txt 所在目录(即 g:/qt1)，并输入以下命令(注意，末尾有一个小数点)

```
cmake .
```

此时，可以在 CMakeLists.txt 所在目录找到生成的 makefile 文件(使用 CMake 的主要目的就是要生成 makefile 文件)及其他文件和文件夹。

- 3)、接着在 CMD 中输入

```
mingw32-make
```

或(注意以下命令末尾的小数点)

```
cmake --build .
```

便在 g:\qt1 中生成了一个 xxx.exe 文件, 然后再在 CMD 中输入

```
xxx.exe
```

就能看到程序输出的内容了。

4、其他事项

在第一次输入 mingw32-make 命令生成可执行文件之前, 可以先输入 mingw32-make -n 查看 mingw32-make 会执行哪些命令。若不希望执行 mingw32-make 命令时显示大量的“注意: 包含文件:”信息, 可找到文件 G:\qt1\CMakeFiles\xxx.dir\build.make, 然后使用记事本打开, 并将其中的“/showIncludes”删除(本示例共有两处)。

1.2.2 分析 CMake 生成的文件

本小节使用上一小节的示例生成的结果

- 1、用记事本打开 g:\qt1 目录中的 makefile 文件, 可得出其依赖关系图如图 1-1 所示, 图中只列出了最关键的命令(变量已展开), 该命令执行了 CMakeFiles\Makefile2 文件中的 all 目标。



图 1-1 主 makefile 的依赖关系图

- 2、用记事本打开 g:\qt1\CMakeFiles\Makefile2 文件, 可得出其依赖关系图如图 1-2 所示(变量已展开), 图中的省略号表示目录 CMakeFiles/xxx.dir。可以看到, 该文件执行了 CMakeFiles/xxx.dir/build.make 文件中的 CMakeFiles/xxx.dir/depend 和 CMakeFiles/xxx.dir/build 目标。

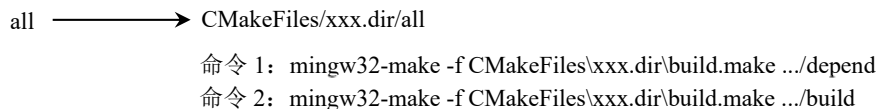


图 1-2 CMakeFiles/Makefile2 的依赖关系图

- 3、用记事本打开 CMakeFiles/xxx.dir/build.make 文件, 可得出其依赖关系图如图 1-3 所示(变量已展开), 图中的省略号表示目录 CMakeFiles/xxx.dir。

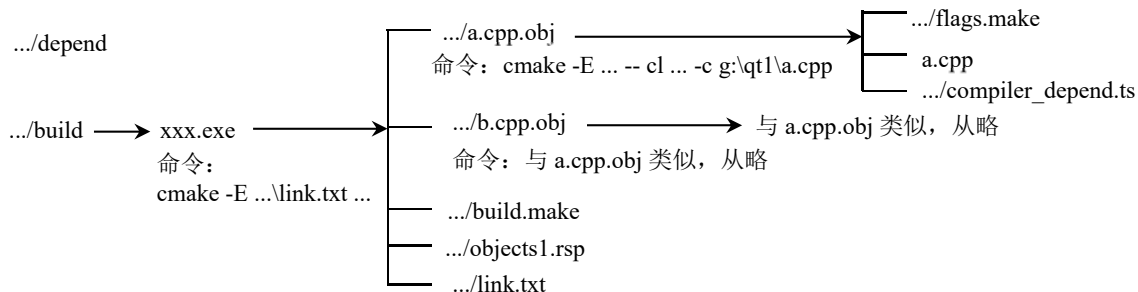


图 1-3 build.make 的依赖关系图

- 1)、build.make 文件首先使用 include 包含了 CMakeFiles/xxx.dir 目录中的以下 4 个文件, 可使用记事本打开这四个文件查看其中的内容。

depend.make、compiler_depend.make、progress.make、flags.make

2)、build.make 文件中的关键是以下三个目标

- xxx.exe
- CMakeFiles/xxx.dir/a.cpp.obj
- CMakeFiles/xxx.dir/b.cpp.obj

这三个目标使用 `cmake` 的命令模式(-E 参数)调用了 `cl` 工具和 `link` 工具(注: 本示例默认使用 VS 的工具),

A、分析 xxx.exe 目标的命令

xxx.exe 目标的命令如下:

```
cmake -E cmake_link_script CMakeFiles\xxx.dir\link.txt --verbose=$(VERBOSE)
```

表示使用脚本的形式调用 `link` 命令, 该命令位于 `CMakeFiles/xxx.dir/link.txt` 文件中, 使用记事本打开该文件, 并将命令所在的路径以及 `link` 的一些参数省略后, 得到:

#CMakeFiles\xxx.dir\link.txt 文件部分内容

```
cmake.exe -E vs_link_exe --intdir=CMakeFiles\xxx.dir
```

```
--rc=rc.exe --mt=mt.exe --manifests
```

```
-- link.exe /nologo @CMakeFiles\xxx.dir\objects1.rsp /out:xxx.exe ...
```

以上 `cmake` 命令表示, 执行符号 “--” 之后的命令, 所以, 这里执行了 `rc.exe`, `mt.exe`, `link.exe` 等命令, 本示例我们只关心 `link` 命令, `link` 后的 `@` 表示将其后的文件的内容作为 `link` 的参数, 使用记事本打开 `objects1.rsp` 文件, 其内容如下

#CMakeFiles\xxx.dir\objects1.rsp 文件

```
CMakeFiles/xxx.dir/a.cpp.obj CMakeFiles/xxx.dir/b.cpp.obj
```

该文件保存了 `link` 所链接的对象文件, 所以, 最终的 `link` 命令为(简化后)

```
link /nologo
```

```
CMakeFiles/xxx.dir/a.cpp.obj CMakeFiles/xxx.dir/b.cpp.obj /out:xxx.exe ...
```

即, 根据 `a.cpp.obj` 和 `b.cpp.obj` 生成可执行文件 `xxx.exe`。

B、分析其他两个目标的命令

同理, 找到 `CMakeFiles/xxx.dir/a.cpp.obj` 或 `CMakeFiles/xxx.dir/b.cpp.obj` 目标, 可以看到, 这两个目标同样使用了 `cmake` 的 -E 参数间接的调用了 `cl` 命令来创建 `.obj` 文件, 其形式为(简化后):

```
cl.exe /nologo $(CXX_DEFINES) $(CXX_INCLUDES) $(CXX_FLAGS) /showIncludes  
/FoCMakeFiles\nn.dir\a.cpp.obj /FdCMakeFiles\nn.dir/ -c G:\qt1\a.cpp
```

其中, `-c` 参数表示创建 `.obj` 文件, 这是关键参数, `/Fo` 参数用于指定输出文件的完整路径名, 参数后无空格。变量 `$(CXX_DEFINES)`、`$(CXX_INCLUDES)`、`$(CXX_FLAGS)` 的值保存在 `include()` 命令包含进来的 `flags.make` 文件中(可使用记事本打开), 这些值指定了 `cl` 命令的额外参数, 在以后可以通过 `CMake` 的命令、属性、变量对其进行修改。

4、综上所述, 可以看到以下目录和文件比较重要

- CMakeFiles\xxx.dir 目录。其中，xxx 表示构建的目标的名称，每构建一个目标都有一个相对应的*.dir 目录。
- CMakeFiles\xxx.dir 目录下的 build.make、link.txt、objects1.rsp、flags.make 四个文件比较关键，其中
 - ◆ build.make 包含有构建目标的 makefile 代码
 - ◆ link.txt 文件包含了 link 命令(链接)、lib 命令(创建库)等命令。
 - ◆ objects1.rsp 文件包含了 link、lib 等命令需要链接的文件。
 - ◆ flags.make 文件包含了编译工具(如 cl 命令)的参数。

1.3 CMake 相关基本概念

1.3.1 CMake 文件

1、CMakeLists.txt 文件

CMakeLists.txt 文件是一个纯文本文件，CMake 必须从顶级目录的 CMakeLists.txt 文件开始构建程序。顶级目录的 CMakeLists.txt 文件被称为主文件，是 CMake 的入口点，该文件包含整个 CMake 构建内容。关于子目录含有 CMakeLists.txt 文件的情形详见后文。

2、CMake 脚本文件

脚本文件的显著特点是可以使用 cmake 的 -P 参数单独运行，即可在命令行使用以下命令执行脚本文件

cmake -P 文件名

CMake 脚本文件是一个纯文本文件，通常使用.cmake 作为后缀(注意：后缀名并没有强制规定)，脚本文件不能含有关于构建目标的代码，这意味着脚本文件不能生成构建系统、不允许定义构建目标，这也意味着脚本文件不能含有与构建目标有关的 CMake 命令，比如，project()、add_subdirectory()、add_executable()等命令都不允许在脚本文件中出现。

3、构建文件：CMake 中的构建文件指的是 makefile、Ninja、VS 项目文件等文件，这些文件可由生成工具构建 C++ 程序。

4、输出工件(Output Artifact)

CMake 把由 CMake 目标创建的真实文件称为输出工件，这些文件是通过调用编译器生成的.exe、.lib、.dll 等文件(依使用的系统而定)，但要注意的是，并不是所有的 CMake 目标都会生成真实存在的文件。有关输出工件更详细的内容详见后文。

5、模块(module)文件：指的是以.cmake 为后缀的文件。

1.3.2 CMake 构建程序的过程

1、构建工具：因为调用生成工具(如 mingw32-make、nmake 等)能够间接调用编译器构建程序，所以本文将生成器(即生成工具)称为构建工具。

2、cmake.exe、生成器、编译器的区别

- 1)、记住：编译器才是真正构建 C++ 程序的工具，除此之外的 `cmake.exe`、`mingw32-make`、`make`、`jom` 等工具都不会构建 C++ 程序，他们即使能构建 C++ 程序也是通过间接调用编译器而完成的。
- 2)、生成器的主要作用是解释 `makefile` 文件，该文件通常会调用编译器构建 C++ 程序，也就是说，使用生成工具可以构建 C++ 程序(由 `makefile` 文件间接调用编译器)，这也是为什么把生成器称为构建工具的原因(其实真正的构建工具是编译器)。
- 3)、`cmake.exe` 的主要作用是根据 CMake 文件生成构建文件(如 `makefile` 文件)，注意：`cmake.exe` 只会生成构建文件，不会进一步解释构建文件，即使需要进一步解释构建文件也必须调用生成工具(通过 `cmake --build` 间接调用)，这意味着 `cmake.exe` 不会构建 C++ 程序，要构建 C++ 程序必须调用编译器(直接或间接调用)。

3、生成(generation)阶段和构建(build)阶段

- 1)、生成阶段是指的生成实际的构建文件的过程，比如生成 `Makefile` 文件、`Ninja` 文件、`VS` 项目文件等的过程。该阶段通常会调用 `cmake.exe` 命令。
- 2)、构建阶段指的是使用构建工具(如 `make`、`Ninja`、`mingw32-make`、`VS` 等)根据生成的构建文件，编译源代码并链接目标文件的过程，也就是说，这一阶段才会真正的调用(间接调用)编译器编译和链接程序。该阶段通常使用 `mingw32-make`、`nmake`、`make` 等命令，或使用 `cmake --build` 命令。
- 3)、以上两个阶段也常被统称为构建，比如，构建一个 CMake 文件，构建一个项目。但是，有时候会对这两个阶段进行区分，从以上过程可见，CMake 实际只负责生成阶段，构建阶段需使用专门的构建工具如 `mingw32-make`、`make`、`namek` 等来完成，而且生成阶段并不会对源文件进行真正的编译和链接等操作，也就是说这一阶段不会生成如 `.obj`、`.exe`、`.lib` 等编译和链接阶段(即构建阶段)才会生成的文件，生成阶段产生的文件仅与构建文件(如 `makefile` 文件)有关，只有当使用构建工具，如 `make`，执行构建文件时才会生成 `.obj`、`.lib`、`.exe` 等编译和链接阶段(即构建阶段)的文件。

1.3.3 CMake 目录结构

- 1、源目录：CMakeLists.txt 文件所在的目录被称为源目录。这里需要注意的是，源目录并不是指的 C++ 源文件所在的目录，C++ 源文件所在的目录与 CMake 的源目录没有关系。
- 2、构建目录：存储构建文件所在的目录被称为构建目录。
- 3、源树：所有的源目录通常会被组织成一个树型的结构，这个结构被称为源树，顶层 CMakeLists.txt 文件所在的源目录便是源树的根目录。
- 4、构建树：所有的构建目录通常会被组织成一个树型的结构，这个结构被称为构建树，顶层的构建目录便是构建树的根，CMake 会创建一个 CMakeCache.txt 文件来标识构建树的顶层目录，也就是说，含有 CMakeCache.txt 文件的目录就是构建树的根目录。
- 5、通常使用树的根目录来称呼该树，因此，源树有两层意思，一是指的整个源树结构，另一个是指源树的根目录，同理，构建树也作如此理解。
- 6、CMake 目录结构更详细的示例，详见第 4 章

1.3.4 cmake 命令行工具的使用

- 1、要使用 CMake 构建程序，就需要使用 CMake 自带的命令工具 `cmake.exe`(以后简称 `cmake`)，本小节将讲解 `cmake` 工具的基本用法。
- 2、生成构建文件可使用以下命令之一

```
cmake [<options>] -B <path-build> [-S <path-source>]  
cmake [<options>] <path-source> | path-to-existing-build>
```

1)、其中-B 参数指定构建文件的输出路径,即构建目录,-S 指定 CMakeLists.txt 文件的路径,即源目录。options 是指的其他参数,如-D、-G、-U 等参数。path-to-existing-build 表示一个已经存在的构建目录,在该目录下必须存在一个 CMakeCache.txt 文件(该文件通常是上次执行 cmake 命令生成的),注:存在 CMakeCache.txt 文件意味着该目录是构建树的根目录。

2)、从以上命令可见,在使用 cmake 生成构建文件时必须指定一个路径。

3)、下面列举一些使用示例。假设当前 CMD 位于 g:/qt1 目录

```
①、g:\qt1> cmake -B f:/x
```

以上命令表示执行当前目录(即 g:/qt1)下的 CMakeLists.txt 文件,并将构建文件输出到 f/x,若在当前目录中不存在 CMakeLists.txt 文件,则将产生错误。

```
②、g:/qt1> cmake -B f:/x -S d:/y
```

以上命令表示执行 d:/y 中的 CMakeLists.txt 文件,并将构建文件输出到 f/x。

```
③、g:\qt1> cmake .
```

注意末尾有一个小数点。以上命令表示执行当前目录下的 CMakeLists.txt 文件,并将构建文件输出到当前目录。

```
④、g:/qt1> cmake f:/x
```

以上命令分以下情况

- 若在 f/x 中存在 CMakeLists.txt 文件,但不存在 CMakeCache.txt 文件,则执行 f/x 中的 CMakeLists.txt 文件,并将构建文件输出到当前目录。
- 若在 f/x 中存在 CMakeCache.txt 文件(构建目录),但不存在 CMakeLists.txt 文件,则执行当前目录中的 CMakeLists.txt 文件,并将构建文件输出到 f/x。
- 若在 f/x 即存在 CMakeLists.txt 文件,也存在 CMakeCache.txt 文件,则按第 2 种情形处理。

3、源内构建和源外构建

1)、源内构建是指将构建文件输出到源目录的构建方式,这种方式将使构建文件与源目录中的 CMakeLists.txt 等文件存放在一起,不能互相区分。

2)、源外构建是指构建文件与源目录分开存放的构建方式。

4、CMake 可使用以下方法构建程序(通常会设用编译器、链接器等工具)

1)、根据构建文件的类型使用对应的 mingw32-make、nmake、jom 等构建工具构建程序。

2)、使用 cmake 的--build 参数构建,命令如下:

```
cmake --build <dir> [<options>]
```

其中<dir>是构建树的路径,即 CMakeCache.txt 文件所在目录,这意味着<dir>指定的目录必须是已经存在的且该目录下必须有 CMakeCache.txt 文件。options 是指定其他的一些选项,如

- -t <tgt> 表示构建指定的 makefile 目标
- --config <cfg>指定配置模式,如 debug、release 等。

使用 cmake --build 的好处是,不需了解生成的构建文件需要使用哪种构建工具构建,也不需了解生

成的构建文件的目录结构(有时构建文件的目录结构比较复杂)

5、脚本文件使用 `cmake -P` 执行，前文已讲过，从略。

第 2 章 CMake 基本命令

- 1、本章将介绍 CMake 的基本语法及一些基本命令，其余命令详见后续章节。
- 2、本章的命令除少数几个命令外，都是脚本命令，这意味着可将示例代码复制到一个文件中，再使用 `-P` 选项来单独执行，如下所示

```
cmake -P <文件名>
```

- 3、在本章及之后的章节还会使用到 CMake 的 `message()` 命令，该命令用于在控制台显示一条指定的消息，比如

```
message(AAA) 或
```

```
message("AAA")
```

将在控制台显示消息 AAA。

2.1 CMake 基本语法

2.1.1 CMake 命令的参数

- 1、CMake 的语法是基于命令的，也就是说，CMake 文件的任何代码都是以命令的形式出现的。这是与 `qmake`、`makefile` 不同的地方。
- 2、CMake 命令由命令名、左圆括号、参数、右圆括号组成，参数以空格分隔。CMake 按照命令出现的顺序进行解析。命令名称不区分大小写，建议使用小写。除了用于分隔参数之外，所有空白(即空格、换行符、制表符)都会被忽略，因此，只要命令名和左括号在同一行，命令就可以跨多行。
- 3、命令参数区分大小写，各参数间以空格分格。有三种类型的命令参数，分别是：方括号参数(bracket argument)，引号参数(quoted argument)，无引号参数(unquoted argument)
- 4、方括号参数(bracket argument)

其语法格式为

```
[=[...[内容]=...]
```

各符号之间不能有空格，以左方括号开始，后跟着零个或多个等号“=”，等号之后再跟着一个左方括号，这意味着，方括号参数至少是以两个方括号“[[”开始的。等号的数量表示方括号的长度，右方括号必须使用相同数量的等号。可以将“[=[...”当作一个整体来理解，即，方括号参数左方括号的形式为“[=[...””，右方括号作类似理解。方括号参数有以下特点

- 方括号参数不能嵌套。
- 紧接在左方括号后面的换行符会被忽略。
- 不会对方括号中的内容进行解析，也就是说，转义序列或变量引用不会被解析。
- 一个方括号参数总是一个参数，而不是多个参数，即使含有分号“;”字符。

- 3.0 以前的 CMake 版本不支持方括号参数

示例 2.1: 方括号参数

```
message([==[AA aa bb]=]cc]==])
```

以上代码将输出

AA aa bb]=]cc。

可将以上代码保存到 a.cmake(或其他名称)中, 再在命令行使用 “cmake -P a.cmake” 来测试。

本示例的 message 命令只有一个参数而不是多个参数。本示例的方括号参数以长度为 2 的左方括号 “[==” 开始, 所以, 不会以长度为 1 的右方括号 “]=” 结束, 而是以长度为 2 的右方括号结束, 即在 “]==]” 处结束

5、引号参数(Quoted Argument)

语法格式为:

“内容”

在双引号之中的所有文本都是引号参数的内容。引号参数有以下特点

- 引号参数的内容不能有双引号和反斜杠 “\”, 因为第二个双引号被视为内容的结束, 而\会被视为转义字符。要在内容中使用这两个字符, 需使用\转义。
- 转义序列和变量引用会被解析。
- 引号参数总是一个参数, 而不是多个参数, 即使含有分号 “;” 字符。
- 以奇数个反斜杠 “\” 结尾的行上的最后一个反斜杠, 会被视为续行符,
- 3.0 以前的 CMake 版本不支持引号参数中的续行符

示例 2.2: 引号参数

```
message("aa\nbb cc \  
dd \"e\  
f  
g")
```

以上示例输出:

```
aa  
bb cc dd "e\  
f  
g
```

虽然 message 参数的内容非常多, 但这里只有一个参数。其中\n 被转义为回车换行符, \" 被转义为", 注意, 这个双引号不是结束双引号, 因为该双引号被转义了。第一行的最后一个反斜杠是续行符, 由于第二行不是以奇数个反斜杠结尾的, 所以最后一个反斜杠不是续行符, 事实上, “\\” 被转义为 “\”。

6、无引号参数(Unquoted Argument)

无引号参数就是没有任何引号的参数, 其内容是除 “空白、(、)、#、”、\” 之外的所有文本, 要使用这些字符, 需使用反斜杠转义。其中, 空白是指空格、TAB、和回车换行符。无引号参数有以下特点:

- 转义序列和变量引用会被解析
- 非空元素会被作为参数提供给命令调用
- 无引号参数可以给命令调用提供零个或多个参数

- 结果值的划分(divide)方式与列表(List)相同。
- 为了兼容旧式的 CMake 代码，无引号参数中可能会含有引号字符串，这时，未转义的双引号必须平衡，并且双引号不能出现在无引号参数的开头，比如，"b"c、"bc 都将是错误的。

示例 2.3：无引号参数

```
message(a"b"cb b\tb ee)
```

以上示例将输出

```
a"b"cb      bee
```

输出时忽略了参数间的空白，虽然只输出了一行，但这里向 message 提供了 3 个参数，分别是“a"b"cb”、“b b”和“ee”

7、表 2.1 是命令参数的对比

表 2.1 各命令参数的对比

参数类型	语法形式	转义序列或变量引用	是否可续行	对分号的处理
方括号参数	[=...[内容]=...]	不会被解析	不可以	使用 message 命令会输出分号
引号参数	"内容"	会被解析	可以	使用 message 命令会输出分号
无引号参数	内容	会被解析	不可以	使用 message 命令不会输出分号

2.1.2 注释、变量、列表

1、注释

CMake 有两种注释：方括号注释(Bracket Comment)和行注释(Line Comment)。无论哪种注释，都是以#开头，并且该字符不能出现在括号参数和引号参数中，也不能在无引号参数中使用\转义。方括号注释以#开始后跟一方括号参数，方括号注释可实现多行注释，行注释以#开始直到本行结束。比如

```
message(aa #[[xx]] bb)      #[[xx]]是方括号注释
message(aa bb)  #[==[xx]==]  ##[==[xx]==]是方括号注释
```

注意，CMake3.0 之前不支持方括号注释

2、变量

- 1)、CMake 的变量总是被解释为字符串类型，尽管会被有些命令解释为其他类型。
- 2)、变量的值可以使用 set()和 unset()命令设置或取消，除此之外，还有其他命令也能修改变量的值。若变量未定义，则该变量的值为空字符串。比如

```
set(AA aa bb)      #定义一个名为 AA 的变量，其值为 aa、bb
```

- 3)、变量名是区分大小写的，可以由几乎任何字符组成，建议只使用字母、数字、_和-命名变量。
- 4)、变量的作用范围有块作用域、函数作用域、目录作用域、持续缓存等。详见 set()、function()、block()等命令。
- 5)、引用变量的形式为

```
${变量名}
```

变量引用可以嵌套，此时，从内到外进行求值，比如，\${aa\${BB}cc}，先对\${BB}求值，然后再

对产生的新变量`${aa...cc}`求值。

6)、有关变量的其他内容请参阅 `set()` 命令

3、列表(List)

- 1)、列表中的元素在 CMake 中以分号分隔，比如 `a b;c d;e`，共有 3 个列表元素 `a b`、`c d`、`e`。
- 2)、CMake 中的所有值都以字符串的形式存储，列表也会被存储为以分号分隔的字符串。但是 `set()` 命令将多个值以列表的形式存储到变量中，实际上，列表也是使用 `set()` 命令创建的，以下是创建列表的方法

```
set(AA aa bb cc)           #创建一个列表。AA 的值为 aa;bb;cc
message("${AA}")           #输出 aa;bb;cc
set(BB "aa bb cc")         #注意：这不是创建一个列表。AA 的值为 aa bb cc
message("${BB}")           #输出 aa bb cc
```

- 3)、许多命令会将字符串视为列表，在这种情况下，应以分号“;”为界，将其划分为列表的元素，比如，若 `a b;c;d` 是一个无引号参数，当将其视为列表时共有 3 个列表元素，分别是 `a b`、`c`、`d`。
- 4)、特别注意：大多数命令不会转义列表元素中的分号字符，这会导致嵌套的列表元素降低到未嵌套的水平，比如

```
set(AA aa "bb;cc dd")      #AA 的值为 aa;bb;cc dd
```

AA 的值并不是两个列表元素 `aa` 和 `bb;cc dd`，而是三个列表元素 `aa`、`bb`、`cc dd`

- 5)、虽然分号有时会被用来分隔参数，但在多个参数之间最好使用空格分隔，否则有可能产生错误。比如：`message(aa;bb)` 将输出 `aabb`，但 `message([[aa]];[[bb]])` 将产生错误

- 6)、通常情况下，列表不支持包含分号的列表元素，为避免产生问题，应注意以下问题：

- CMake 的许多命令、变量和属性接受以分号分隔的列表，应避免将包含分号元素的列表传递给这些接口。
- 若需要处理含有分号的字符串时，可将其中的分号使用其他字符代替，然后在合适的时候再将其转换回来，比如 `a;b`，可使用 `a|b` 代替，在合适的时候，再将 `|` 转换回分号。
- 在命令调用中，尽量使用引号参数，因为引号参数会保留分号。

2.1.3 生成器表达式 `$<TARGET_OBJECTS:...>`

- 1、生成器表达式是指具有形式为“`$<...>`”的表达式，比如

- `$<condition:true_string>`
当 `condition` 为 1 时，则结果为 `true_string`，为 0 时，结果为空字符串，为其他值时产生错误。
- `$<IF:condition,true_string,false_string>`
表示当 `condition` 为 1 时，则结果为 `true_string`，为 0 时，结果为 `false_string`，为其他值时将产生错误。
- `$<AND:conditions>`
其中 `conditions` 是一个由逗号分隔的布尔表达式列表，所有表达式的值必须是 1 或 0，若表达式列表的所有值都为 1 时，则结果为 1，若任何一个表达式的结果为 0，则结果为 0。
- 更多的生成表达式请参阅 CMake 官方文档。

- 2、生成器表达式在生成构建系统期间求值，而不是在处理 `CMakeLists.txt` 文件期间求值，因此，无法使用 `message()` 命令查看其值，但可以使用以下方法来查看生成器表达式的值

```
add_custom_target(genexdebug COMMAND ${CMAKE_COMMAND} -E echo "$<...>")
```

其中，变量`<CMAKE_COMMAND>`的值是 `cmake`，即，以上命令相当于是执行以下命令

```
cmake -E echo "$<...>"
```

`cmake` 命令的 `-E` 参数表示，执行 `-E` 之后的命令，所以，以上命令的最终结果是执行 `-E` 之后的 `echo` 命令，所以，最终执行的命令是

```
echo "$<...>"
```

然后使用 `make` 工具执行 `genexdebug` 目标，即，形式如下：

```
mingw32-make genexdebug
```

就能看到生成器表达式的值了。以上命令表示，执行 `makefile` 文件的 `genexdebug` 目标。

3、另一种方法是使用以下命令把生成器表达式的信息保存到文件 `filename` 中

```
file(GENERATE OUTPUT filename CONTENT "$<...>")
```

示例 2.4：生成器表达式

①、在目录 `g:/qt1` 下的 `CMakeLists.txt` 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(hh)
add_custom_target(xxx COMMAND ${CMAKE_COMMAND} -E echo "$<AND:1,1,0>")
add_custom_target(xxx1 COMMAND ${CMAKE_COMMAND} -E echo "$<IF:1,aaa,bbb>")
file(GENERATE OUTPUT www.txt CONTENT "$<1:ddd>")
```

②、在 CMD 中转至 `g:/qt1` 并输入以下命令(注意：`cmake` 后面有一个小数点)

```
cmake .
```

```
ming32-make xxx xxx1
```

依次输出 0、aaa，然后在当前目录找到并打开 `www.txt` 文件，可在文件中见到“ddd”字符串。

2.2 cmake_policy()命令

1、`cmake_policy()`命令的作用

`cmake_policy()`命令用于设置是否使用 `CMake` 新版本的功能，在 `CMake` 中将这些功能称为策略 (policy)，每个新策略都有一个形式为 `CMP<NNNN>` 的标识，其中 `<NNNN>` 是一个整数，每个策略都有 `OLD` 和 `NEW` 两个行为，`NEW` 表示使用新策略，`OLD` 表示使用旧的策略，所有的策略默认都为 `OLD` 行为并且会产生一个警告以要求设置策略。所有策略的 `OLD` 行为表示在将来会被弃用，在 `CMake` 的未来版本中都可能会被删除。

2、`cmake_policy()`命令可以单独设置某个策略，也可以基于 `CMake` 版本设置策略，建议使用基于 `CMake` 版本的设置方法。

3、基于版本设置策略

其语法为

```
cmake_policy(VERSION <min>[...<max>])
```

1)、其中 min 和 max 是 CMake 的版本编号，版本编号的形式为

```
主版本号.次版本号.[补丁].[微调]]
```

比如 3.27.0，其主版本号为 3，次版本号为 27，补丁号为 0。min 的值必须至少是 2.4，最大为当前运行的版本，若当前运行的版本比 3.12 还低，则...max 部分被忽略。

2)、cmake_policy(VERSION)命令表示在给定的 CMake 版本范围编写 CMake 代码，该命令将 min (或 max，如果有的话) 版本或更早版本的所有策略全部设置为 NEW，在之后版本中引入的所有策略将被取消设置。

3)、cmake_minimum_required(VERSION)命令隐式调用 cmake_policy(VERSION)命令。

示例 2.5: 基于版本设置策略

```
cmake_policy(VERSION 3.27)      #使用 3.27 版本，即，将 3.27 及更早的版本的所有策略设置为 NEW
cmake_policy(VERSION 3.0...3.27) #与上一例效果相同
```

4、单独设置策略 cmake_policy(SET)

其语法为：

```
cmake_policy(SET CMP<NNNN> NEW|OLD)
```

其中，NNNN 是策略的数字编号，OLD 表示使用旧策略，NEW 表示使用新的策略。比如

```
cmake_policy(SET CMP0126 NEW)
```

表示启用策略 CMP0126 的新功能，若被设置为 OLD 则表示使用旧的功能。

5、检查策略状态 cmake_policy(GET)

其语法为：

```
cmake_policy(GET CMP<NNNN> <variable>)
```

检查策略 CMP<NNNN>是否被设置为 OLD 或 NEW，并将结果保存在变量 variable 中，否则变量 variable 的值为空。

6、策略堆栈(Stack)

其语法为：

```
cmake_policy(PUSH|POP)
```

1)、其中 PUSH 表示入栈，POP 表示出栈， PUSH 和 POP 必须配对使用。

2)、CMake 将策略设置保存在堆栈中，因此，cmake_policy()命令的设置只会影响堆栈顶部的条目(entry)，CMake 为每个子目录自动管理策略堆栈，以保护其父目录和兄弟目录，除非 include()和 find_package()命令使用 NO_POLICY_SCOPE 选项，否则，CMake 还管理由这两个命令加载的脚本的新条目。

3)、该命令对于单独设置一小部分代码的使用不同的策略非常有用。比如

```
cmake_policy(PUSH)
cmake_policy(SET CMP0012 NEW)
.....
cmake_policy(POP)
```

以上代码启用的策略 CMP0012 仅在启动开始至 POP 之间的代码段范围内有效。

4)、命令 cmake_minimum_required(VERSION)、cmake_policy(VERSION)、cmake_policy(SET)只能影响策略堆栈顶部上的条目。

5)、block()命令会在执行 endblock()命令时自动执行 POP 操作，详见 block()命令。

示例 2.6：使用策略堆栈

```
function(xx)                                #创建一个名称为 xx 的函数，函数会创建一个作用域，详见 function() 命令
    cmake_policy(PUSH)                      #推入策略堆栈条目
    cmake_policy(SET CMP0012 NEW)          #启用策略 CMP0012
    if(TRUE)
        message(aa)
        cmake_policy(POP)                  #弹出策略堆栈条目
    else()
        message(bb)
        cmake_policy(POP)
    endif()
endfunction()                              #函数结束
xx()                                       #调用函数
if(TRUE)                                  #此处会得到一个不能识别 TRUE 的警告
    message(cc)
else()
    message(dd)
endif()
```

以上示例会依次输出 aa, dd(可将代码复制到文件中并使用 `cmake -P` 测试)。该示例用于测试策略 CMP0012 的作用范围，启用 CMP0012 会将字符常量 TRUE 识别为条件表达式的真，否则不能识别。由于策略 CMP0012 在函数 xx 被入栈和出栈，所以，其作用范围仅在函数 xx 的作用域内，在函数 xx 中的 if 能识别字符串 TRUE，但在 xx 函数外的 if 不能识别字符串 TRUE。

示例 2.7：不使用策略堆栈

```
function(xx)
    cmake_policy(SET CMP0012 NEW)          #启用策略 CMP0012，但未使用策略堆栈
    if(TRUE)
        message(aa)
    else()
        message(bb)
    endif()
endfunction()
xx()
if(TRUE)
    message(cc)
else()
    message(dd)
endif()
```

以上示例会依次输出 aa, cc，由于该示例未将策略 CMP0012 在函数 xx 中入栈，所以，该策略作用在最外层作用域的策略堆栈条目上，因此，函数 xx 内和函数 xx 外的 if 都能识别字符串 TRUE。

2.3 cmake_minimum_required()命令

1、cmake_minimum_required()命令语法为:

```
cmake_minimum_required(VERSION <min>[...policy_max] [FATAL_ERROR])
```

该命令用于设置项目的最低需求版本，该命令隐式调用 cmake_policy(VERSION)命令，因此，除 FATAL_ERROR 参数外，其余行为均与 cmake_policy(VERSION)命令相同，所以，详细的内容请参阅 2.2 节，从略。

2、cmake_minimum_required()命令与 cmake_policy(VERSION)命令有以下不同:

- 选项 FATAL_ERROR 在 2.6 及更高版本会被接受并被忽略，在 2.4 及以下版本会产生一个错误。
- cmake_minimum_required()命令会将 CMAKE_MINIMUM_REQUIRED_VERSION 变量的值设置为 <min>，若在函数内调用 cmake_minimum_required()命令，则该变量的值仅在该函数内起作用，在函数外将不起作用，因此，不建议在函数中使用 cmake_minimum_required()命令。
- **CMake 要求必须指定项目所需的最低版本**，这样可以更方便的使用 CMake 的最新功能，CMake 要求在顶级 CMakeLists.txt 文件的开头(即，第一行)使用 cmake_minimum_required()命令指定最低版本而不是 cmake_policy(VERSION)命令，虽然二者作用差不多。

2.4 set()命令

set()命令用于设置常规变量、环境变量和缓存变量的值，set()命令将多个参数以一个分号分隔的字符串连接起来存储到变量中，注意：set 命令不会将值追加到变量中，而是按照 set()命令的执行顺序定义。

2.4.1 常规变量

设置常规变量的语法为

```
set(<变量名> <值>... [PARENT_SCOPE])
```

引用常规变量的语法为:

```
${变量名}
```

- 1、若给出了<值>则设置为变量的值。若没有给出值，则取消变量的值，相当于 unset(<变量名>)。需要注意的是，set 命令不会将值追加到变量中而是按照 set()命令的执行顺序定义。
- 2、选项 PARENT_SCOPE 表示设置父作用域(parent scope)中的变量的值，但在当前作用域中，变量的值仍然保持先前状态的值。

示例 2.8: 设置常规变量

```
set(AA aa bb)
message("AA=${AA}")      #输出 aa:bb, 变量以分号分隔的字符串的形式被存储
set(AA cc)
```

```
message("AA=${AA}") #输出 cc。set 不会把新值追加到变量中
```

示例 2.9: 设置父作用域中的变量

```
set(AA bb) #①
function(xx) #②定义函数 xx，函数创建了一个作用域
    message(AA1=${AA})
    set(AA cc PARENT_SCOPE) #③设置父作用域中的变量 AA 的值
    message(AA2=${AA})
endfunction() #④函数结束
message(AA3=${AA}) #⑤输出 AA2=bb
xx() #⑥调用函数 xx，此步会执行函数中的 set() 命令
message(AA4=${AA}) #⑦输出 AA3=cc
xx() #⑧再次调用函数 xx
message(AA5=${AA}) #⑨输出 AA4=cc
```

- 1) 以上代码依次输出 AA3=bb、AA1=bb、AA2=bb、AA4=cc 和 AA1=cc、AA2=cc、AA5=cc。
- 2) 在②处定义了一个函数 xx，在函数被调用之前，函数中的语句不会被执行，因此，首先执行在⑤处的语句，输出 AA3=bb，
- 3) 在⑥处调用函数 xx，此时执行 xx 中的第一个 message 输出 AA1=bb，然后执行 set 设置父作用域中的变量 AA(即，①处定义的变量 AA)，然后执行 xx 中的第二个 message，由于在当前作用域中，变量 AA 的值仍然保持先前状态的值，所以输出 AA2=bb，
- 4) 然后执行⑦处的 message，此时由于函数 xx 调用完毕，①处的变量 AA 的值在函数中被更改为 cc，所以，此处输出 AA4=cc，
- 5) 在⑧处再次调用函数 xx，其原理与前面类似，依次输出 AA1=cc，AA2=cc，并再次将①处的 AA 设置为 cc，最后执行⑨处的 message 输出 AA5=cc

2.4.2 环境变量

设置环境变量的语法为

```
set(ENV{变量名} 值)
```

引用环境变量的语法为:

```
$ENV{变量名}
```

- 1、若未指定值，则清除环境变量的值，除第一个值之后的值会被忽略，并发出警告。set 命令不会将值追加到环境变量中。
- 2、环境变量与普通变量有以下不同:
 - 环境变量具有全局作用域，且永远不会被缓存。
 - 对环境变量值的更改，只影响正在运行的 CMake 进程，不会影响整个环境变量，更改的值不会被写回调用进程，且不会被后续的构建或测试进程看到。比如

```
set(ENV{TEMP} dd) #更改环境变量的值，但只影响当前进程
message($ENV{TEMP})
```

以上示例仅在当前 CMake 进程输出 dd，但系统环境变量的实际值 TEMP 并未被更改。

示例 2.10: 设置环境变量

```
set(ENV{AA} aa bb) #环境变量只接受第一个值 aa，这里会收到一个警告(因为指定了多个值)
```



```
message( "AA=$ENV{AA}" )      #输出 AA=aa
set(ENV{AA} cc)
message( "AA=$ENV{AA}" )      #输出 AA=cc。
```

2.4.3 缓存变量

设置缓存变量的语法为

```
set(<变量名> <值>... CACHE <type> <docstring> [FORCE])
```

引用缓存变量的语法为:

```
$CACHE{变量名}
```

- 1、缓存变量也称为缓存项(cache entry)或缓存条目，默认情况下，缓存项不会覆盖已存在的缓存项的值，除非使用 FORCE 选项。<docstring>必须是一行文本，以为 cmake-gui 提供该选项的一个快速摘要，cmake-gui 是一个有关 CMake 的图形界面工具。
- 2、<type>必须是以下值之下：
 - BOOL：布尔开关值
 - FILEPATH：磁盘上文件的路径
 - PATH：磁盘上目录的路径
 - STRING：一行文本
 - INTERNAL：一行文本，用于持久存储变量，使用该选项意味着设置 FORCE
- 3、缓存变量的两个主要特点是：
 - 缓存变量的值在项目构建树中的多次运行中持续存在
 - 当对变量引用\${变量名}求值时，CMake 首先搜索与该名称同名的常规变量，若不存在，则搜索缓存变量。因此，取消常规变量的设置可能会使之前隐藏的缓存项暴露出来，若希望变量的值为空字符串，建议使用 set(<变量名> "")。
 - 若在 CMake 项目模式下，缓存变量的值会被保存在执行 cmake 命令后生成的名为 CMakeCache.txt 的文件中，再次执行 cmake 命令时将直接从该文件中读取值。
- 4、缓存变量与同名的常规变量的关系

当策略 CMP0126(CMake 3.21)为 NEW 时，不会从当前作用域中删除任何相同名称的常规变量，为 OLD 时(默认情形)会在以下情形从当前作用域中删除所有同名常规变量，这意味着，此时不能直接访问缓存变量

- 之前不存在该名称的缓存变量
- 设置缓存变量时使用了 FORCE 或 INTERNAL 关键字
- 该名称的缓存变量存在，但未指定<type>，这种情形发生在从命令行创建缓存变量。

示例 2.11：缓存变量之间的覆盖规则

```
set(AA aa CACHE STRING xx)
set(AA bb CACHE STRING xx)      #不会覆盖已存在的缓存变量
message( $CACHE{AA} )           #输出 aa
set(BB aa CACHE STRING xx)
set(BB bb CACHE STRING xx FORCE) #使用 FORCE 覆盖已存在的缓存变量
message( $CACHE{BB} )           #输出 bb
```

以上示例依次输出 aa、bb

示例 2.12：缓存变量与常规变量间的关系

```
set(AA aa CACHE STRING xx)
message(${AA})           #输出 aa，因未找到常规变量，所以搜索缓存变量
#不设置 CMP0126 策略
set(BB cc)               #常规变量
set(BB aa CACHE STRING xx) #默认情况下，会删除同名的常规变量
message($CACHE{BB} ${BB}) #输出 aaaa
#设置 CMP0126 策略
cmake_policy(SET CMP0126 NEW) #将策略 CMP0126 设置为 NEW
set(CC cc)               #常规变量
set(CC aa CACHE STRING xx) #不会删除同名的常规变量
message($CACHE{CC} ${CC}) #输出 aacc
```

以上示例依次输出 aa、aaaa、aacc

示例 2.13：缓存变量的持续存在性

```
function(xx)              #①函数作用域，详见 function() 命令
  set(AA aa CACHE STRING xx FORCE) #缓存变量 AA
  block(SCOPE_FOR VARIABLES)      #②块作用域，详见 block() 命令
    set(AA cc CACHE STRING xx FORCE) #更改缓存变量 AA 的值
    message(aa=$CACHE{AA})         #③输出 aa=cc
  endblock()
  message(bb=$CACHE{AA})          #④输出 bb=cc
endfunction()
xx()
message(cc=$CACHE{AA})           #⑤输出 cc=cc
```

以上示例依次输出 aa=cc、bb=cc、cc=cc，从示例可见，缓存变量创建于函数作用域，但最外层作用域⑤处的 message 仍能访问该变量。缓存变量的值在最内层的块作用域中被更改，但在外层的 block 作用域④处的 message 和最外层作用域⑤处的 message 都输出相同的值。

2.5 function()、macro()、block()、return()命令

2.5.1 function()命令

function()的语法为：

```
function(<name> [<arg1> ...])
<命令>...
endfunction()
```

调用函数的语法为：

函数名(参数 ...)

- 1、function()命令可用于创建一个函数，函数调用时，函数名不区分大小写。
- 2、函数以 function 开始，以 endfunction()结束，函数的名称为<name>，参数为<arg1> ...，
- 3、函数中的命令在函数被调用之前不会被执行。
- 4、定义函数时的参数被称为形式参数，调用函数时的参数称为传入参数。传入参数的数量可以比形式参数多，但不能比形式参数数量少。形式参数可以像引用常规变量一样被引用。
- 5、一个函数会创建一个新的作用域，有关函数作用域的示例请参阅示例 2.8。
- 6、在函数中还可以使用以下内置的特殊变量
 - ARGC：表示传入参数的数量
 - ARGV：表示传入参数的值列表
 - ARGV0, ARGV1...：表示第一个传入参数的值、第二个传入参数的值...，若 ARGV 后指定的数目大于传入参数的个数，则是未定义的行为
 - ARGN：表示比形式参数多出的那部分传入参数的值列表，比如，形式参数为(a b)，传入参数为(x y z c d)，则 ARGN 为 zcd。

示例 2.14：使用函数

```
function(xx y z)           #定义一个名称为 xx 带有两个形式参数的函数
  message(${y})             #输出形式参数 y 的值 a
  message(${ARGC})          #输出传入参数的数量 6
  message(${ARGV})          #输出传入参数的值列表 abcdef
  message(${ARGN})          #输出比形式参数多出的那部分传入参数的值列表 cdef
  message(${ARGV3})         #输出第 4 个传入参数的值 d
endfunction(xx)
xx(a b c d e f )           #调用函数 xx
```

以上示例依次输出 a、6、abcdef、cdef、d

2.5.2 macro()命令

macro ()命令用于创建一个宏，其语法为：

```
macro (<name> [<arg1> ...])
<命令>...
endfunction()
```

调用宏的语法为

宏名(参数 ...)

调用宏时，宏名不区分大小写，宏与函数类似，但有以下区别

- ARGC、ARGV、ARGN 和 ARGV0, ARGV1...等在函数中是变量，但是在宏之中他们不是变量，虽然在宏中也可以像引用变量那样引用他们，但是有区别，这在 if()命令中能看到区别。
- 执行函数调用时，是从调用语句转移到函数体来执行函数，但执行宏就像将宏主体粘贴到调用语句的位置一样。
- 建议不要在宏中使用 return()命令。

2.5.3 block()命令

block()命令用于创建一个块作用域，其语法为(注意，此命令在 3.25 版本引入)：

```
block([SCOPE_FOR [POLICIES] [VARIABLES] ] [PROPAGATE <变量名> ...])
<命令>...
endblock()
```

与函数和宏不同的是，块不需要调用，当执行到块时，将 block()到 endblock()之间的所有命令记录下来，当解析到 endblock()时，就解析 block()作用域内的命令，然后删除创建的作用域。各选项意义如下：

- SCOPE_FOR：指定需创建哪些作用域，若未指定该选项，相当于 block(SCOPE_FOR VARIABLES POLICIES)
 - POLICIES：创建一个新的策略(policy)作用域，相当于 cmake_policy(PUSH)，详见 cmake_policy()命令(2.2 节)。
 - VARIABLES：创建一个新的变量作用域。
- PROPAGATE：设置或取消设置父作用域中指定的变量，类似于 set()命令的 PARENT_SCOPE 选项。

示例 2.15: block()命令之 SCOPE_FOR VARIABLES 参数

```
set(AA aa)
block(SCOPE_FOR VARIABLES)    #创建块作用域---新的变量作用域
    set(AA bb)                #此命令不影响父作用域中的 AA
    message(${AA})            #输出 bb
endblock()                    #块作用域结束
message(${AA})                #输出 aa
set(AA cc)
message(${AA})                #输出 cc
```

以上示例依次输出 bb、aa、cc

示例 2.16: block()命令之 PROPAGATE 参数

```
set(AA aa)                    #①
block(PROPAGATE AA)           #创建块作用域---可更改父作用域的变量
    set(AA bb)                #此命令改变①处 AA 的值
    message(${AA})            #输出 bb
endblock()                    #块作用域结束
message(${AA})                #输出 bb
set(AA cc)
message(${AA})                #输出 cc
```

以上示例依次输出 bb、bb、cc

2.5.4 return()命令

return()命令用于从函数、目录和文件返回，其语法为

```
return([PROPAGATE <变量名>...])
```

1、return()命令返回的规则如下：

- 1)、若在函数中遇到 `return()` 命令，则将控制权返回给该函数的调用者，注意，与 `function()` 不同，`macro()` 原地展开，因此，不能处理 `return()`
- 2)、当在 `include()` 或 `find_package()` 命令包含的文件中遇到 `return()` 命令时，会导致当前文件停止处理，并将控制权返回给包含的文件。
- 3)、若在没有被其他文件包含的文件中遇到 `return()` 命令，如 `CMakeLists.txt`，则调用使用 `cmake_language(DEFER)` 命令预先定义的延迟调用，并把控制权返回给父目录(如果有的话)。延迟调用的简要语法如下，详细内容请参 CMake 官方文档：

`cmake_language(DEFER CALL 命令 参数)`

其中 `DEFER` 表示定义一个延迟调用，`CALL` 之后是需要执行的命令及相应的参数。

- 2、`PROPAGATE` 选项：用于设置或取消设置父目录或函数调用者作用域中指定的变量，类似于 `set()` 命令的 `PARENT_SCOPE` 选项。该选项与 `block()` 结合使用更有用，`return()` 将通过 `block()` 创建的任何封闭块作用域传播到指定的变量。若在函数内，这确保变量被传播到函数的调用者，而不管函数中的任何块；如果不在函数内，则确保将变量传播到父文件或目录作用域。该选项在 3.25 版本引入，默认情况下，CMake 不检测 `return()` 的参数，除非将 `CMP0140` 设置为 `NEW`。

示例 2.17: return()语句的延迟调用

```
cmake_minimum_required(VERSION 3.27)      #启用 CMP0140 策略
project(ProjectName)
set(AA aa)
cmake_language(DEFER CALL message "XX")    #预先定义的延迟调用执行一条 message 命令
message(${AA})
return()                                   #return 之后的语句不会被执行
cmake_language(DEFER CALL message "YY")    #该语句不会被执行，这不是预先定义的延迟调用
message(${yyy})                            #该语句不会被执行
```

以上代码应保存到 `CMakeLists.txt` 文件中，并需在 CMD 中输入命令 “`cmake .`” 调用，将依次输出 `aa`，`XX`，注意，使用 `-P` 选项可能不会执行延迟调用。

示例 2.18: 无 return()时变量的作用域

```
set(AA aa)                                #1
function(xx)
  message(AA1=${AA})
  set(AA bb)                               #2
  message(AA2=${AA})
  block(SCOPE_FOR VARIABLES)
    set(AA cc PARENT_SCOPE)                #设置 1 处的 AA，AA 的值在当前作用域保持不变
    message(AA3=${AA})                     #输出 AA1=bb
  endblock()
  message(AA4=${AA})                       #输出 AA2=cc
endfunction()
xx()
message(AA5=${AA})                         #输出 AA3=aa
```

以上示例依次输出 `AA1=aa`、`AA2=bb`、`AA3=bb`、`AA4=cc`、`AA5=aa`，其中，

- `AA2` 输出的是函数作用域 `xx` 中定义的变量 `AA` 的值(即 2 处定义的值)

- 块作用域中的 `set` 设置的是父作用域即函数作用域 `xx` 中的变量 `AA` 的值，但该变量的值在离开当前作用域之前会保持以前的值不变，所以，`AA3` 输出的仍然是函数作用域中定义的变量 `AA` 的值 `bb`。
- `AA4` 输出的是在块作用域中更改的函数作用域中的变量 `AA` 的新值 `cc`。
- `AA5` 输出的是第一行 `set` 定义的 `AA` 的值 `aa`，该值从未被修改过。

示例 2.19: 含 `return()` 命令时变量的作用域

```
cmake_policy(SET CMP0140 NEW)  #启用 CMP0140 策略
set(AA aa)                      #1
function(xx)
    set(AA bb)                  #2
    block(SCOPE_FOR VARIABLES)
        set(AA cc)              #3
        message(AA1=${AA})      #输出 AA1=cc
        return(PROPAGATE AA)    #把 3 处对 AA 的修改直接返回到 4 处
    endblock()
    message(AA2=${AA})           #此命令被 return 跳过
endfunction()
xx()                            #4
message(AA3=${AA})              #输出 AA3=cc
```

以上示例依次输出 `AA1=cc`、`AA3=cc`

2.6 if()命令(条件语句)

2.6.1 基本语法

`if()` 命令类似于 C/C++ 语言中的条件语句，其语法如下：

```
if(<condition>)
<命令>
elseif(<condition>)    #可选的，并且 elseif 可重复
<命令>
else()                  #可选的
<命令>
endif()
```

`if()` 命令表示，按照 CMake 的运算规则计算 `if` 子句的条件参数 `<condition>`，若结果为真，则执行 `if` 块中的命令，否则，以同样的方式处理可选的 `elseif` 块，最后，若所有的 `<condition>` 都不为真，则执行可选

的 else 块中的命令。参数<condition>是一个条件表达式，下面介绍一些常用的条件表达式

2.6.2 基本的条件表达式

1、if(<常量>)

若常量为 1、ON、YES、TRUE、Y 或非零数字(含浮点数)，则为“真”；若常量为 0、OFF、NO、FALSE、N、IGNORE、NOTFOUND、空字符串或以后缀-NOTFOUND 结尾，则为“假”。这些字符常量不区分大小写，若参数不是以上字符常量之一，则将其视为变量或字符串，并遵守 if(<变量>)或 if(<string>)的规则。需要启用 CMP0012，否则 CMake 无法识字符常量，比如 TRUE、YES 等将无法被识别。

2、if(<变量>)

若给定的变量的值是非“假”常量值，则为真；否则为假，包括未定义的变量。**注意，macro()的参数不是变量**，环境变量也不能使用此方法测试，其结果总是假。

3、if(<string>)

1)、若 string 未加引号，则只将其解释为变量名或关键字。注意：若 if 的参数是无引号参数，且是变量名时，会被自动展开。

2)、若<string>是引号参数或方括号参数，则当括起来的值是真常量时是真，否则为假，如 if("TRUE")、if([[y]])、if([=[yes]=])是真，而 if("dkf")、if([[dfs]])是假。

3)、若<string>是引号参数或方括号参数，并且括起来的值是一个变量名时，应遵守以下规则

- 若未启用 CMP0054 策略(即 OLD 行为)，则展开该变量或解释引号或方括号括起来的關鍵字
- 若启用 CMP0054 策略(即 NEW 行为)，则不展开该变量，也不解释引号或方括号括起来的關鍵字。

4、表 2.2 是基本条件表达式的总结对比

表 2.2 if()基本条件表达式总结

条件表达式	判定方法	是否展开变量
<变量>	非“假”常量，则为真	
引号参数或方括号参数	非“真”常量，则为假	展开变量(CMP0054 NEW) 不展开变量(CMP0054 OLD)
无引号参数	解释为变量名或关键字	展开变量

示例 2.20: if()命令----不启用 CMP0054(OLD 行为)

```
cmake_policy(SET CMP0054 OLD) #不启用 CMP0054
set(AA xx)
if("AA")                      #结果为真，这里会将 AA 理解为变量，该变量的值是非“假”常量值
    message(aa)
endif()
if("AA" STREQUAL "xx")        #结果为真，将变量 AA 展开，类似于 if("xx" STREQUAL "xx")
    message(bb)
endif()
```

以上代码依次输出 aa、bb

示例 2.21: if()命令----启用 CMP0054(NEW 行为)

```

cmake_policy(SET CMP0054 NEW)          #启用 CMP0054
set(AA xx)
if("AA")                                #结果为假，只会将 AA 当作一个字符串，该字符串非真常量值，所以为假
    message(cc)
else()
    message(dd)
endif()
if("AA" STREQUAL "xx")                  #结果为假，不会将 AA 当作变量并展开
    message(ee)
else()
    message(ff)
endif()

```

示例 2.22: if()命令---无引号参数和引号参数 1

```

if(xxx)                                #结果为假，将 xxx 视为变量，但该变量未定义
    message(aa)
endif()
if("xxx")                              #结果为假。xxx 非“真”常量，结果为假
    message(bb)
endif()

```

示例 2.23: if()命令---无引号参数和引号参数 2

```

cmake_policy(SET CMP0054 NEW)
set(xxx fff)
if(xxx)                                #结果为真，将 xxx 视为变量，展开变量，变量的值“非假”，结果为真
    message(aa)
endif()
if("xxx")                              #结果为假。启用了 CMP0054，不将 xxx 视为变量。xxx 非“真”常量，结果为假
    message(bb)
endif()

```

示例 2.24: if()命令---无引号参数 1

```

set(BB n)
set(AA BB)
if(${AA})                              #结果为 false，首先展开 AA 得到${BB}，再展开为 n
    message(aa)
else()
    message(bb)
endif()

```

示例 2.25: if()命令---无引号参数 2

```

set(BB n)
set(AA BB)
if(AA)                                 #结果为 true。首先展开 AA 得到 BB，到此不再展开
    message(aa)
else()
    message(bb)
endif()

```


2.6.3 逻辑表达式

注：以下的参数若是变量会被自动展开。

- 1、if(NOT <condition>): 若 condition 为假，则为真。
- 2、if(<con1> AND <con2>): 若 con1 和 con2 都为真，则为真
- 3、if(<con1> OR <con2>): 若 con1 或 con2 其中之一为真，则为真
- 4、if((condition) AND (condition OR (condition))) : 这是一种使用 AND 和 OR 连接起来的嵌套用法，可以嵌套多层，当嵌套时，从最内层括号开始判断真假，然后判断外层括号。

示例 2.26: if()命令---逻辑表达式

```
cmake_policy(SET CMP0012 NEW)
set(AA xx)
if((AA AND (AA AND y)) OR n OR n)      #结果为真
    message(bb)
endif()
```

2.6.4 比较

注：以下的参数若是变量会被自动展开。

- 1、if(<var|string> MATCHES <regex>)
若变量 var 或字符串 string 的值与给定的正则表达式 regex 匹配，则为 true。有关正则表达式的内容请参阅相关资料，本文从略。
- 2、if(<var|string> LESS <var|string>)
要求 var 或 string 是数字，若左侧的数字小于(LESS)右侧的数字时为 true，否则为 false。可将 LESS 替换为 GREATER、EQUAL、LESS_EQUAL、GREATER_EQUAL，则分别表示大于、等于、小于等于、大于等于。
- 3、if(<var|string> STRLESS <var|string>)
按字典顺序比较左侧与右侧字符串的大小，同理，可将 STRLESS 替换为 STRGREATER、STREQUAL、STRLESS_EQUAL、STRGREATER_EQUAL
- 4、if(<var|string> VERSION_LESS <var|string>)
以上命令用于版本比较，若左侧版本小于右侧版本则为真。版本格式为 major[.minor[.patch[.tweak]]]。同理，可将 VERSION_LESS 替换为 VERSION_GREATER、VERSION_EQUAL、VERSION_LESS_EQUAL、VERSION_GREATER_EQUAL
- 5、if(<var|string> PATH_EQUAL <var|string>)
判断两路径是否相等，其中，反斜杠不会转换为正斜杠，并且多个路径分隔符(使用正斜杠表示)会被表示为一个分隔符，注意：需启用 CMP0139 (3.24 版本)。比如
cmake_policy(SET CMP0139 NEW)
if(a/b/c PATH_EQUAL a/b/c) #结果为 true
if(a/b/c PATH_EQUAL a/b/c) #错误，应使用正斜杠表示路径

2.6.5 存在性检测

1、if(COMMAND <command-name>)

若由 `command-name` 给定的命令、宏、函数能被调用，则为 `true`。比如 `if(COMMAND set)` 为真，因为 CMake 的 `set` 命令能被调用，结果为真

2、if(POLICY <policy-id>)

若给定的策略存在，则为 `true`，比如 `if(POLICY CMP0026)` 为 `true`，而 `if(POLICY CMP0888)` 为 `false`，因为，目前 CMake 还不存在 `CMP0888` 的策略。

3、if(TARGET <target-name>)

若名称 `target-name` 是通过调用 `add_executable()`、`add_library()`、`add_custom_target()` 命令创建的已存在的逻辑名称，则为 `true`。

4、if(TEST <test-name>)

若 `test-name` 是由 `add_test()` 命令创建的已存在的测试名称，则为 `true`。

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

测试名称 `name` 是变量、缓存变量还是环境变量，变量的值无关紧要。注意，`macro()` 命令的参数不是变量。另外，不能直接测试 `name` 是否是常规变量，因为，无论 `name` 是常规变量还是缓存变量，`if(DEFINED name)` 都为 `true`，比如：

```
set(AA xx CACHE STRING tt)
if(DEFINED CACHE{AA})      #结果为 true
    message(aa)
endif()
if(DEFINED AA)              #结果为 true。可见，使用该命令不能判断 AA 是常规变量还是缓存变量
    message(aa)
endif()
```

6、if(<var|string> IN_LIST <variable>)

若 `var` 或 `string` 在 `variable` 中，则为 `true`，需启用 `CMP0057`。比如

```
cmake_policy(SET CMP0057 NEW) #启用 CMP0057
set(BB xx YY ZZ)
set(AA xx)
if(AA IN_LIST BB)           #结果为 true
    message(aa)
endif()
if(YY IN_LIST BB)           #结果为 true
    message(bb)
endif()
```

2.6.6 文件或目录

1、if(EXISTS <file-or-dir>)

若指定的文件或目录存在，则为 `true`，前导的 `~/` 不会被展开为主目录，被认为是相对路径。解析符

号链接(类似于 windows 的快捷方式, 但有一些区别), 即, 如果符号链接的目标存在, 则返回 true。比如

```
if(EXISTS "g:/qt1/yy")      #若指定的 yy 存在则为真
    message(aa)
endif()
```

2、if(<file1> IS_NEWER_THAN <file2>)

若 file1 比 file2 更新或其中一个文件不存在, 则为 true, 若 file1 和 file2 时间戳完全相同, 也返回 true。

3、if(IS_DIRECTORY <path>)

若 path 是目录, 则为 true。

4、if(IS_SYMLINK <path>)

若 path 是符号链接, 则为 true。

5、if(IS_ABSOLUTE <path>)

若 path 是绝对路径, 则为 true。在 windows 中, 任何以驱动器号和冒号、正斜杠或反斜杠开头的路径都为 true; 在非 windows 系统中, 任何以~开头的路径都为 true。比如

```
if(IS_ABSOLUTE C:fd/f)      #在 Windows 系统中, 结果为 true。
                             #注意: 路径 C:fd/f 是非法的, 但仍为 TRUE, 因为是以冒号开头的
```

补充知识: Windows 系统创建符号链接的方法(类似快捷键, 但不是快捷键)

在 CMD 中输入以下命令

```
mklink /D yy g:\qt1\xxx
```

其中/D 选项表示创建一个目录符号链接, 以上命令表示, 创建一个链接到目标 xxx 的名称为 yy 的目录符号链接, 此处要求 xxx 是一个文件夹的名称。

2.7 foreach()命令

2.7.1 foreach 常规形式

其语法为:

```
foreach(<loop_var> <items>)
<命令>
endforeach()
```

- 1、其中 loop_var 被称为循环变量, items 是由分号或空白分隔的元素列表。注意有关列表的问题。
- 2、foreach()命令在 endforeach()后调用, 其执行过程为: 为 items 中的每一个元素调用一次命令, 也就是说 items 有多少个元素就会调用多少次命令, 并在调用命令前, 将循环变量 loop_var 的值设置为 items 中当前项的值。循环变量 loop_var 的作用域被限制为循环作用域。
- 3、若未启用 CMP0124 策略(3.21 版本引入), 则在循环结束时清除 loop_var, 但不会取消设置(unset)该变量, 也就是说该变量的定义仍被保持, 但为空, 若启用了 CMP0124 则当循环结束时取消设置

loop_var。注意，即使未启用 CMP0124，在 3.27 版本也不会发出警告信息。

示例 2.27: foreach()基本使用方法

```
set(AA xx yy zz)
foreach(BB ${AA})           #也可使用 foreach(BB xx yy zz)
    message(${BB})
endforeach()
```

以上示例依次输出 xx、yy、zz

示例 2.28: foreach()与列表

```
set(CC 1 "4;5 6")           #由于 set 的特殊性，变量 CC 将其值保存为 1;4;5 6
foreach(XX ${CC})
    message("XX=${XX}")
endforeach()
foreach(YY 1 "4;5 6")        #引号参数始终是一个参数
    message("YY=${YY}")
endforeach()
```

以上示例依次输出如下：XX=1、XX=4、XX=5 6、YY=1、YY=4;5 6

2.7.2 foreach 变体 1

其语法为：

```
foreach(<loop_var> RANGE <stop>)
```

表示从 0 开始循环，直到大于等于 stop 为止，loop_var 被依次赋值 0~stop

2.7.3 foreach 变体 2

其语法为：

```
foreach(<loop_var> RANGE <start> <stop> [<step>])
```

表示以步长(或增量)step 从 start 开始循环直到大于等于 stop 为止。

示例 2.29: foreach()以步长增长循环

```
foreach(BB RANGE 1 5 3)
    message(${BB})
endforeach()
```

以上示例依次输出 1、4。本示例以步长 3 从 1 开始循环，直到大于等于 5 为止。即，第一次循环为 1，第二次循环为 1+3=4，第 3 次循环为 4+3=7>5，终止循环。

2.7.4 foreach 变体 3

其语法为：

```
foreach(<loop_var> IN [LISTS [<lists>]] [ITEMS [<items>]])
```

遍历循环<lists>或<items>中的项，并将其元素值赋给变量<loop_var>。其中<lists>是一个使用分号或空白分隔的变量名列表，也就是说<lists>指定名称会被处理为变量名；<items>是使用分号或空白分隔的列表元素，也就是说<items>指定的名称不会被处理为变量名，因此，LISTS A 和 ITEMS \${A}是等价的。

示例 2.30: foreach(IN)的使用

```
set(AA 1 2)
set(BB 3)
set(CC "4;5 6")
foreach(XX IN LISTS AA;BB CC ITEMS 7 8)
    message(${XX})
endforeach()
```

以上示例依次输出 1、2、3、4、5 6、7、8

示例 2.31: foreach(IN)----LISTS 与 ITEMS 的区别

```
set(AA 1 2)
set(BB 3)
set(CC "4;5 6")
foreach(XX IN LISTS AA;BB dd ITEMS CC ${CC})
    message(${XX})
endforeach()
```

以上示例依次输出 1、2、3、CC、4、5 6。首先遍历 LISTS 中指定的变量 AA，将值 1 和 2 赋给变量 XX，然后遍历 BB，将值 3 赋给 XX，然后遍历 dd，这里会把 dd 当作变量，由于变量 dd 未定义，所以没有值，然后遍历 ITEMS 指定的 CC，由于 ITEMS 不会把 CC 当作变量，而是当作一个普通的字符串，所以将字符串 CC 赋值给 XX，最后将 CC 的值赋值 4;5 6 给 XX。

2.7.5 foreach 变体 4

其语法为：

```
foreach(<loop_var>... IN ZIP_LISTS <lists>)
```

- 1、其中 lists 是一个使用分号或空白分隔的变量名列表，<loop_var>...表示可以指定一个或多个循环变量 loop_var。
- 2、如果只指定了一个循环变量 loop_var，则使用一系列的名为 loop_var_N 的变量接受对应的由 lists 指定的变量中的值。若指定了多个循环变量，则循环变量的总数应该与<lists>指定的变量的总数相等，否则无法循环迭代。

示例 2.32: foreach(IN ZIP_LISTS)的使用

```
set(AA 1 2)
set(BB 3)
set(CC 1 "4;5 6")
foreach(XX IN ZIP_LISTS AA BB CC)
    message("XX_0=${XX_0}   XX_1=${XX_1}   XX_2=${XX_2}   XX_3=${XX_3}")
endforeach()

foreach(XX YY ZZ IN ZIP_LISTS AA BB CC)
    message("XX=${XX}   YY=${YY}   ZZ=${ZZ}")
```

```
endforeach()
```

以上示例输出如下：

```
XX_0=1 XX_1=3 XX_2=1 XX_3=
XX_0=2 XX_1=  XX_2=4 XX_3=
XX_0=  XX_1=  XX_2=5 6    XX_3=
XX=1   YY=3   ZZ=1
XX=2   YY=    ZZ=4
XX=    YY=    ZZ=5 6
```

2.8 while()命令

while()命令的语法为：

```
while(<condition>)
<命令>
endwhile()
```

当执行到 endwhile()时，若<condition>为真，则执行命令。<condition>的取值情况请参阅 if()命令。需要注意的是，只要<condition>为真，就会循环执行命令，直到<condition>为假，所以，在 while()的命令中需要有结束循环的代码，否则会陷入死循环。可以使用 break()直接跳出循环语句。

示例 2.33：使用 while()命令

```
cmake_policy(SET CMP0012 NEW)      # CMP0012 用于识别 TRUE、Y 等字符串
set(AA y)
while(AA)
    message(aa)
    set(AA n)                       #将 AA 的值设置为 n，以退出循环。
endwhile()
while(y)
    message(bb)
    break()                         #使用 break() 命令退出循环
endwhile()
```

2.9 break()和 continue()命令

其语法分别如下：

```
break()
continue()
```

break()和 continue()用于跳出 foreach()和 while()循环，其原理与 C/C++的 break 和 continue 关键字类似，break()命令用于跳出整个循环，continue()用于跳过当前循环 continue()命令之后的语句进入下一轮循环。

2.10 option()命令

其语法为:

```
option(<variable> "<help_text>" [value])
```

- 1、以上命令用于为变量<variable>指定一个布尔值，即，创建一个布尔变量，也就是说，使用 `option` 创建的变量只能有 ON 和 OFF 两个值之一。如果未提供初始值<value>，则默认值为 OFF。如果已经将<variable>设置为普通变量或缓存变量，则该命令不执行任何操作。
- 2、在 CMake 项目模式下，将创建一个布尔缓存变量。在 CMake 脚本模式下，将创建一个布尔变量。

2.11 unset()命令

- 1、`unset` 命令用于取消变量、缓存变量或环境变量的设置。
- 2、取消常规变量和缓存变量的语法为:

```
unset(<variable> [CACHE | PARENT_SCOPE])
```

以上命令表示从当前作用域中删除一个普通变量，使其变为未定义。如果存在 `CACHE`，则删除缓存变量而不是普通变量。如果存在 `PARENT_SCOPE`，则从当前作用域上方的作用域中删除该变量。

- 3、取消环境变量的语法为:

```
unset(ENV{<variable>})
```

以上命令表示从当前可用的环境变量中删除<variable>。后续调用 `$ENV{<variable>}` 将返回空字符串。该命令只影响当前的 CMake 进程，而不是调用 CMake 的进程，也不是整个系统环境，也不是后续构建或测试进程的环境变量。

第3章 CMake 属性

许多 CMake 对象(如目标、目录、源文件)都具有与其关联的属性，属性是附加到特定对象上的一个“键-值”对，通常使用 `set_property()` 和 `get_property()` 命令设置和访问属性的值，除此之外，还有更为方便的 `set_XXX_property()` 和 `get_XXX_properties()` 命令。我们可以定义自己的属性和值，CMake 也自带了很多属性。

CMake 自带的属性名称(包括变量名称)中的 `<LANG>` 表示使用的编程语言，如 `CXX` 表示 C++ 语言，比如 `<LANG>_STANDARD` 属性，若是 C++ 语言则为 `CXX_STANDARD`。`<CONFIG>` 表示编译的方式(也称为编译模式、配置模式、构建模式)，通常有 `DEBUG` 和 `RELEASE` 两种方式，比如 `LOCATION_<CONFIG>` 属性，若是 `DEBUG` 模式，则为 `LOCATION_DEBUG`。

3.1 `set_property()` 和 `get_property()` 命令

3.1.1 `set_property()` 命令

其语法为：

```
set_property(<GLOBAL |  
    DIRECTORY [<dir>] |  
    TARGET [<target1>...] |  
    SOURCE [<src1>...] [DIRECTORY <dirs>...] [TARGET_DIRECTORY <targets>...] |  
    INSTALL [<file1>...] |  
    TEST [<test1>...] |  
    CACHE [<entry1>...] >  
[APPEND] [APPEND_STRING]  
PROPERTY <name> [<value1>...] )
```

- 1、该命令用于在一个作用域(或称为作用范围)上为零个或多个对象设置一个属性，此命令一次只能设置一个属性，要一次设置多个属性可使用后文介绍的 `set_XXX_properties()` 命令。
- 2、属性的作用范围(也称为作用域)，是指属性能产生作用的范围，比如，若属性的作用范围是目标，则该属性只是指定的目标上起作用。
- 3、为方便讲解，本文把作用域为 `GLOBAL` 的属性称为全局属性，作用域为 `TARGET` 的属性称为目标属性，同理，还有目录属性、源文件属性、安装属性、测试属性、缓存属性。其中，全局属性、缓存属性的使用示例见 3.1.2 小节，源文件属性的使用示例见 3.2 小节
- 4、各选项的意义如下：

- 该命令的第一个参数用于指定属性的作用域，必须是表 3.1 所列选项之一。
- PROPERTY 选项是必须的，其后是需设置的属性名，value 是以分号分隔的列表形式的属性值。
- APPEND 选项用于向现有属性追加属性值。
- APPEND_STRING 选项表示以字符串的形式向属性追加属性值，这意味着，追加后的值是一个更长的字符串，而不是一个列表。
- 若属性在指定的作用域中不存在，则 APPEND 或 APPEND_STRING 就像没有给出一样。当对定义为支持 INHERITED 行为的属性使用 APPEND 或 APPEND_STRING 时(参见 define_property() 命令)，INHERITED 行为不会发生。

表 3.1 set_property()命令的第一个参数

参数	说明
GLOBAL	在全局作用域创建属性
DIRECTORY	在指定的目录作用域创建属性。若未指定 dir，则默认为当前目录。3.19 版本后可使用二进制目录
TARGET	为指定的零个或多个目标创建属性，该目标必须是已经存在的，并且不能是别名目标。这里的目标指的是 CMake 目标，如使用 add_library()命令或 add_executable()命令创建的目标。另见 set_target_properties()命令
SOURCE	<p>为指定的零个或多个源文件创建属性。默认情况下，源文件属性只对在同一目录(CMakeLists.txt)中添加的目标可见。可见性可以在其他目录作用域中由以下选项设置</p> <p>1、DIRECTORY <dirs>...:</p> <p>源文件属性将在每个 dirs 目录的作用域中设置，CMake 必须已经知道这些目录中的每一个，或是通过调用 add_subdirectory()添加的目录，或是顶级源目录。相对路径被视为相对于当前源目录的路径。3.19 版本以后 dirs 可使用二进制目录。比如：</p> <pre>set_property(SOURCE g:/qt2/c.cpp DIRECTORY g:/qt1)</pre> <p>表示为 g:/qt1 目录(该目录 CMake 必须已经知道)下的 CMakeLists.txt 文件中添加的 g:/qt2/c.cpp 源文件设置属性，该属性只对 g:/qt2/c.cpp 有效且限于 g:/qt1/CMakeLists.tx 文件中的源文件。</p> <p>2、TARGET_DIRECTORY <targets>...:</p> <p>源文件属性将在“创建的任何指定 targets 的每个目录作用域中”设置(因此，targets 必须已经存在)。比如，目标 a 创建于目录 x，目标 b 创建于目录 y，则 TARGET_DIRECTORY a b 表示将在目标 a 和 b 所在的目录 x 和 y 设置源文件属性。</p> <p>另见 set_source_files_properties()命令。始终存在名称为 LOCATION 的源文件属性(即，源文件的完整路径)。源文件属性的具体使用示例见后文示例 6.2。</p>
INSTALL	为指定的零个或多个已安装的文件路径创建属性，这些属性会提供给 CPack 以影响部署。路径必须使用正斜杠分隔，并且区分大小写。属性的键和值都可以使用生成器表达式，特定的属性可能适用于已安装的文件或目录。当前安装的文件属性，仅为 WIX 生成器定义，其中给定的路径相对于安装前缀。
TEST	仅限于调用该命令的目录，可以指定零个或多个测试，另见 set_tests_properties()命令。可以使用生成器表达式为 add_test(NAME)创建的测试指定测试属性值。
CACHE	<p>为指定的零个或多个现有缓存变量创建属性，可设置的 CACHE 属性有：</p> <ul style="list-style-type: none"> ● ADVANCED: 其值是一个布尔值 ● HELPSTRING: 其值是一个描述字符串 ● STRING: 其值是一个字符串 ● TYPE: 其值是缓存变量的类型，即 BOOL、STRING、FILEPATH、PATH、INTERNAL

3.1.2 get_property()命令

其语法为:

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

该命令用于从某个作用域内的一个对象获取一个属性。各选项意义如下:

- <variable>是一个变量, 用于存储获取的属性值。
- 第二个参数表示获取属性的作用域, 必须是表 3.2 所列选项之一。
- SET 选项表示将变量<variable>设置为布尔值, 以指示是否已设置该属性。
- DEFINED 选项表示将变量<variable>设置为布尔值, 以指示是否已定义该属性(使用 define_property()命令)。
- BRIEF_DOCS 或 FULL_DOCS 表示将变量<variable>设置为所获取属性的文档字符串(由 define_property()命令设置), 若属性未定义, 则返回 NOTFOUND。详见 define_property()命令。

表 3.2 get_property()命令的第一个参数

选项	说明
GLOBAL	获取全局作用域中创建的属性
DIRECTORY	获取在指定目录作用域中创建的属性。若未指定 dir, 则默认为当前目录。3.19 版本后可使用二进制目录, 另请参阅 get_directory_property()命令
TARGET	获取为指定的目标创建的属性, 另见 get_target_property()命令
SOURCE	获取为指定的源文件创建的属性。默认情况下, 从当前源目录读取源文件属性, 可以使用以下选项指定目录作用域 1、DIRECTORY <dir>: 从目录作用域 dir 中读取源文件属性, CMake 必须已经知道该目录, 或是调用 add_subdirectory()添加的目录, 或是顶级源目录。相对路径被视为相对于当前源目录。3.19 版本以后 dir 可使用二进制目录。 2、TARGET_DIRECTORY <target> 从创建 target 的目录作用域中获取源文件属性(因此, targets 必须已经存在)。 另见 get_source_file_property()命令。始终存在名称为 LOCATION 的源文件属性(即, 源文件的完整路径)。源文件属性的具体使用示例见后文示例 6.2。
INSTALL	从已安装的文件路径获取创建的属性。
TEST	获取为指定的测试创建的属性, 另见 get_test_property()命令。

CACHE	获取为指定的缓存变量创建的属性，请参阅 <code>set_property()</code> 命令的 CACHE 选项
VARIABLE	作用域是唯一的且不接受名称

示例 3.1：在全局作用域创建一个属性

```
set_property( GLOBAL PROPERTY xx aa;bb cc) #在全局作用域创建一个名称为 xx 的属性
get_property(AA GLOBAL PROPERTY xx)      #从全局作用域获取属性 xx 的值并保存在变量 AA 中
message("${AA}")                          #输出 aa;bb;cc
foreach(BB ${AA})                         #属性的值以列表的形式保存
    message("${BB}")
endforeach()
```

示例 3.2：向属性追加属性值

```
set_property( GLOBAL PROPERTY xx aa;bb cc)
set_property( GLOBAL APPEND PROPERTY xx dd ee) #向属性 xx 增加属性值
get_property(AA GLOBAL PROPERTY xx)
message("${AA}") #输出 aa;bb;cc;dd;ee
foreach(BB ${AA})
    message("${BB}") #依次输出 aa、bb、cc、dd、ee
endforeach()
```

示例 3.3：以字符串的形式向属性追加属性值

```
set_property( GLOBAL PROPERTY xx aa;bb cc)
set_property( GLOBAL APPEND_STRING PROPERTY xx dd ee) #以字符串形式向属性 xx 增加值
get_property(AA GLOBAL PROPERTY xx)
message("${AA}") #输出 aa;bb;ccdd;ee，注意：ccdd 之间无分号，这是与上一示例的区别
foreach(BB ${AA})
    message("${BB}") #依次输出 aa、bb、ccdd、ee，注意 ccdd 组合成字符串
endforeach()
```

示例 3.4：给缓存变量指定属性

```
set(CC tt CACHE STRING xxf) #设置缓存变量
set_property(CACHE CC PROPERTY TYPE BOOL) #为缓存变量指定 TYPE 属性，CC 实际类型为 STRING
get_property(AA CACHE CC PROPERTY TYPE) #获取缓存变量 CC 的属性 TYPE 的属性值
message("${CC}") #输出 tt
message("${AA}") #输出 BOOL
```

3.2 define_property()命令

`define_property()` 命令的语法为：

```
define_property(<GLOBAL | DIRECTORY | TARGET | SOURCE | TEST | VARIABLE | CACHED_VARIABLE>
               PROPERTY <name>
               [INHERITED]
               [BRIEF_DOCS <brief-doc> [docs...]]
```

[FULL_DOCS <full-doc> [docs...]]

[INITIALIZE_FROM_VARIABLE <variable>])

该命令仅仅定义一个属性，不能设置属性值，主要用于定义属性的初始化或继承方式(INHERITED 选项)，通常需要与 `set_property()` 和 `get_property()` 命令一起使用。各参数意义如下：

1、第一个参数用于指定属性的作用域，与 `set_property()` 和 `get_property()` 不同，该命令只需指定作用域的类型而不需给出实际的作用域。即

- GLOBAL：与一个全局属性关联
- DIRECTORY：与一个目录属性关联
- TARGET：与一个目标属性相关联
- SOURCE：与一个源文件属性关联
- TEST：与一个测试属性相关联
- VARIABLE：与一个 CMake 语言变量关联
- CACHED_VARIABLE：与一个 CMake 缓存变量关联

2、INHERITED 参数

该参数表示该属性可以从父作用域继承。如果指定了该参数，若在给定的作用域中未找到所请求的属性，则 `get_property()` 命令将链接(chain)到更高的作用域中查找该属性，具体为：

- 若是 DIRECTORY 作用域则链接到其父作用域继续向上搜索，直到找到该属性或没有父目录为止。若在顶级目录还未找到，则链接到 GLOBAL 作用域继续查找。
- 若是 TARGET、SOURCE、TEST 属性，则链接到 DIRECTORY 作用域，并根据需要进一步链接到目录作用域。

注意，这种作用域链接行为仅适用于对 `get_property()`、`get_directory_property()`、`get_target_property()`、`get_source_file_property()`、`get_test_property()` 的调用。因此，在使用 `set_property()` 命令使用 APPEND 或 APPEND_STRING 追加内容时将不会考虑继承值。

3、BRIEF_DOCS 和 FULL_DOCS 参数后面是一个字符串，该字符串是一个简短和完整的文档描述，CMake 不使用此文档，可通过 `get_property()` 命令的相应选项获取该文档的内容。

4、INITIALIZE_FROM_VARIABLE 参数指定一个变量 `variable`，属性应从该变量初始化。该变量只能与 TARGET 属性一起使用，变量 `variable` 的名称必须以属性名结尾，并且不能以 CMAKE_或_CMAKE_ 开头，属性名必须至少包含一个下划线。

示例 3.5: define_property()命令----继承属性

①、在目录 `g:/qt1` 下的 `CMakeLists.txt` 中编写如下代码

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
#设置目录属性指定的目录需与 CMakeLists.txt 文件一致
set_property(DIRECTORY g:/qt1 PROPERTY ss gg)
set_property(GLOBAL PROPERTY tt ff)

#定义目标属性 ss 和 tt 可以从父作用域继承
define_property(TARGET PROPERTY ss INHERITED)
```

```

define_property(TARGET PROPERTY tt INHERITED)

#创建一个 CMake 目标 kk，本示例不会构建程序，源文件 a.cpp 可以是一个空文件
add_library(kk STATIC a.cpp)

#以下命令不会考虑继承值
set_property(TARGET kk APPEND PROPERTY ss hh)    #①

#获取目标 kk 的属性 ss 和 tt
get_property(AA1 TARGET kk PROPERTY tt )
get_property(AA2 TARGET kk PROPERTY ss)

message(AA1=${AA1})
message(AA2=${AA2})

```

②、在 CMD 中转至 g:/qt1 并输入以下命令

cmake .

将依次输出 AA1=ff、AA2=hh。

③、分析代码

本示例不是一个脚本代码，不能使用 `cmake -P` 命令执行。本示例并未为目标 `kk` 创建属性 `ss` 和 `tt`，其中 `tt` 的值继承目录属性 `tt`。由于在代码中对目标 `kk` 的属性 `ss` 使用了 `set_property(APPEND)` 命令追加值，而在追加值时不会考虑继承值，所以，在示例中①处的代码会将清除掉继承值。

示例 3.6: `define_property()` 命令---使用变量初始化属性

①、在目录 g:/qt1 下的 CMakeLists.txt 中编写如下代码

```

cmake_minimum_required(VERSION 3.27)
project(xxxx)
set(BBs_ss gg)
set(tt ff)
#属性名必须至少含有一个下划线，变量名必须以属性名结尾
define_property(TARGET PROPERTY s_ss INITIALIZE_FROM_VARIABLE BBs_ss)
#定义一个变量属性
define_property(VARIABLE PROPERTY tt INHERITED)

#创建一个 CMake 目标，本示例不会构建程序，源文件 a.cpp 可以是一个空文件
add_library(kk STATIC a.cpp)

#获取属性 ss 和 tt
get_property(AA1 VARIABLE PROPERTY tt )
get_property(AA2 TARGET kk PROPERTY s_ss)

message(AA1=${AA1})
message(AA2=${AA2})

```

②、在 CMD 中转至 g:/qt1 并输入以下命令

cmake .

将依次输出 AA1=ff、AA2=gg。

示例 3.7: define_property()命令---BRIEF_DOCS 选项

在目录 g:/qt1 下的 CMakeLists.txt 中编写如下代码

```
cmake_minimum_required(VERSION 3.27)
project(hh)
define_property(GLOBAL PROPERTY xx BRIEF_DOCS bb ccd)
get_property(AA GLOBAL PROPERTY xx BRIEF_DOCS)      #变量 AA 被设置为 bbccd
message(AA=${AA})      #输出 bbccd
```

3.3 其他属性命令

3.3.1 set_target_properties()命令

其语法为:

```
set_target_properties(target1 target2 ... PROPERTIES prop1 value1 prop2 value2 ...)
```

以“键-值”对的形式为一个或多个目标设置一个或多个属性，比如

```
set_target_properties(xx1 xx2 PROPERTIES aa 11 bb 22)
```

表示为目标 xx1 设置两个属性 aa 和 bb，其值分别为 11 和 22，同理 xx2 也有两个属性 aa 和 bb

3.3.2 set_directory_properties()命令

其语法为:

```
set_directory_properties(PROPERTIES prop1 value1 [prop2 value2] ...)
```

以“键-值”对的形式设置当前目录和子目录的属性，该命令可一次性设置多个属性。

3.3.3 set_source_files_properties()命令

其语法为:

```
set_source_files_properties(<files> ...
    [DIRECTORY <dirs> ...]
    [TARGET_DIRECTORY <targets> ...]
    PROPERTIES <prop1> <value1> [<prop2> <value2>] ...)
```

以“键-值”对的形式为一个或多个源文件设置一个或多个属性，参数 DIRECTORY 和 TARGET_DIRECTORY 的意义与 set_property()命令的同名参数相同，从略。

3.3.4 set_tests_properties()命令

其语法为:

```
set_tests_properties(test1 [test2...] PROPERTIES prop1 value1 prop2 value2)
```

以“键-值”对的形式为一个或多个测试设置一个或多个属性，若未找到测试，将产生一个错误。

3.3.5 get_cmake_property()命令

其语法为:

```
get_cmake_property(<var> <property>)
```

获取全局属性<property>的值，并将其存储在变量<var>中，若未找到属性，则<var>被设置为 NOTFOUND。由于历史原因，该命令还能获取 VARIABLES、MACROS 目录属性的值和一个特殊的 COMPONENTS 全局属性的值，COMPONENTS 与 install()命令有关。

3.3.6 get_target_property()命令

其语法为:

```
get_target_property(<VAR> target property)
```

从目标 target 获取属性 property 的值，并存储在变量<VAR>中。若未找到属性，则对于非 INHERITED 属性(参见 define_property()命令)，将 VAR 设置为 VAR-NOTFOUND，对于 INHERITED 属性继续按 define_property()描述的规则搜索父作用域，若仍未找到该属性，则将 VAR 设置为空字符串。

3.3.7 get_directory_property()命令

其语法为:

```
get_directory_property(<variable> [DIRECTORY <dir>] <prop-name>)
```

获取由 dir 指定的目录作用域中的属性<prop-name>的值，并将其存储在<variable>中，参数 DIRECTORY 的意义与 get_property()命令的同名参数相同，从略。若未找到属性，则返回空字符串。

3.3.8 get_source_file_property()命令

其语法为:

```
get_source_file_property(<variable> <file>  
[DIRECTORY <dir> | TARGET_DIRECTORY <target>] <property>)
```

从文件 file 获取属性 property 的值，并存储在变量 variable 中，默认情况下，从当前源目录的作用域读取源文件的属性。若未找到属性，则对于非 INHERITED 属性(参见 define_property()命令)，将 variable 设置为 NOTFOUND，对于 INHERITED 属性继续按 define_property()描述的规则搜索父作用域，若仍未找到该属性，则将 variable 设置为空字符串。始终存在属性 LOCATION(即，源文件的完整路径)。参数 DIRECTORY 和 TARGET_DIRECTORY 的意义与 get_property()命令的同名参数相同，从略

3.3.9 get_test_property()命令

其语法为:

```
get_test_property(test property VAR)
```

从测试 test 获取属性 property 的值，并存储在变量 VAR 中。若未找到属性，则对于非 INHERITED 属性 (参见 define_property()命令)，将 VAR 设置为 NOTFOUND，对于 INHERITED 属性继续按 define_property()描述的规则搜索父作用域，若仍未找到该属性，则将 VAR 设置为空字符串。

第 4 章 CMake 项目

4.1 add_subdirectory()命令

4.1.1 CMake 项目的结构

- 1、对于一个大型项目，通常会将每个部分独立出来单独构建或测试，本文将这种独立出来的部分称为子项目。
- 2、通常每个独立部分位于主要部分的子目录下，每个子目录下必须有一个 CMakeLists.tx 文件(称为子 CMakeLists.txt)，作为该子目录的入口点，通常子 CMakeLists.txt 应该包含自己的 project()命令，以便在子目录中形成一个完整的构建系统，当然，子 CMakeLists.txt 也可以不包含 project()命令。
- 3、根目录下的主 CMakeLists.txt 文件负责调用各子目录下的子 CMakeLists.txt 文件，add_subdirectory()命令就是用于告诉 CMake 需要在哪些子目录下去寻找子 CMakeLists.txt 的。当运行 cmake 命令时，首先查找根目录下的主 CMakeLists.txt 文件，然后递归地处理每个子目录中的子 CMakeLists.txt 文件，每个 CMakeLists.txt 文件都会生成一个对应的 makefile 文件。

4.1.2 add_subdirectory()命令

其语法如下：

```
add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL] [SYSTEM])
```

该命令用于添加一个子目录。各参数意义如下：

1、source_dir 参数

该参数用于指定子 CMakeLists.txt 所在的子目录，可以是相对路径和绝对路径，相对路径相对于当前源目录。若指定的目录不是当前源目录的子目录，则必须指定 binary_dir。

2、binary_dir 参数

该参数指定存放子项目构建文件的目录，可以是相对路径和绝对路径，相对路径相对于当前构建目录。若未指定 binary_dir，则使用 source_dir 的值

3、EXCLUDE_FROM_ALL 参数

该参数表示子目录中的目标将不被包含在父目录的 ALL 目标中(指 makefile 文件的 ALL 目标)，并且将从 IDE 项目文件中排除。使用此选项在构建程序时，可以选择性的不构建子目录中的目标。注意，目标间依赖会使该参数无效，如果父项目构建的目标依赖于子目录中的目标，则依赖的目标将包含在父项目构建系统中以满足依赖关系。

4、SYSTEM 参数

适用于 3.25 及以上版本。若指定了此参数，则子目录的 SYSTEM 目录属性将被设置为 true。SYSTEM 属性用于初始化在该子目录中创建的每个非导入目标的 SYSTEM 目标属性。

5、SYSTEM 目标属性

1)、适用于 3.25 及以上版本。其作用是将目标指定为系统目标，这有以下效果：

- 在编译消费者(最终使用者)时，INTERFACE_INCLUDE_DIRECTORIES 的条目被视为系统包含目录。INTERFACE_SYSTEM_INCLUDE_DIRECTORIES 的条目不受影响，并且将始终被视为系统包含目录。
- 在 Apple 平台上，如果 FRAMEWORK 目标属性为 true，则框架目录被视为 system。

2)、对于导入的目标，该属性默认为 true，这意味着：

- 如果 FRAMEWORK 目标属性为 true，则它们的 INTERFACE_INCLUDE_DIRECTORIES 和框架目录默认被视为系统目录。
- 如果它们的 SYSTEM 属性为 false，则它们的 INTERFACE_INCLUDE_DIRECTORIES 和框架都不会被视为 system。

3)、可以使用 EXPORT_NO_SYSTEM 属性来更改目标系统属性在安装时的设置方式。

6、表 4.1 是与源树和构建树有关的变量和属性

表 4.1 与源树和构建树有关的属性和变量

类别	变量或属性名	说明	
目标属性	SOURCE_DIR	只读属性，存储的是在被定义的目标的目录中 CMAKE_CURRENT_SOURCE_DIR 变量的值	
目录属性	SOURCE_DIR	只读属性，存储的是读取它的源目录的绝对路径。	
变量	CMAKE_CURRENT_SOURCE_DIR	CMake 当前正在处理的源目录的完整路径	在 cmake -P 模式下运行时，cmake 将这四个变量设置为当前工作目录
	CMAKE_SOURCE_DIR	当前 CMake 源树顶层的完整路径，对于源内构建(in-source build)，这将与 CMAKE_BINARY_DIR 相同。	
	CMAKE_BINARY_DIR	当前 CMake 构建树顶层的完整路径，对于源内构建将与 CMAKE_SOURCE_DIR 相同。	
	CMAKE_CURRENT_BINARY_DIR	当前正在处理的 CMake 构建目录的完整路径，通过 add_subdirectory()添加的每个目录将在构建树中创建一个二进制目录，并在处理它时设置该变量。对于源内构建，这将与当前源目录相同。	

示例 4.1: CMake 项目结构---理解构建树和源树

①、在目录 g/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt (主文件，根目录)
cmake_minimum_required(VERSION 3.27)
project(xx)
add_subdirectory(f:/x aa)           #添加子目录 f:/x
add_subdirectory(m:/x m:/y)         #添加子目录 m:/x
message(根目录 g:/qt1)
```

②、在目录 f:/x 中的 CMakeLists.txt 中编写如下代码

```
#f:/x/CMakeLists.txt (子目录)
add_subdirectory(d:/y bb)           #再添加一个子目录 d:/y
message(子目录 f:/x)
```

③、在目录 d:/y 中的 CMakeLists.txt 中编写如下代码

```
#d:/y/CMakeLists.txt (孙目录)
```

```
message(孙目录 d: /y)
```

④、在目录 m:/x 中的 CMakeLists.txt 中编写如下代码

```
#m:/x/CMakeLists.txt (子目录)
```

```
message(子目录 m: /x)
```

⑤、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake -B e:/x
```

以上命令表示，执行 g:/qt1 下的 CMakeLists.txt 并将构建文件输出到 e:/x 中，执行以上命令会依次输出以下内容，从输出内容的顺序可以看到 CMake 执行的顺序

孙目录 d:/y、子目录 f:/x、子目录 m:/x、根目录

⑥、分析本示例的目录结构----源树和构建树

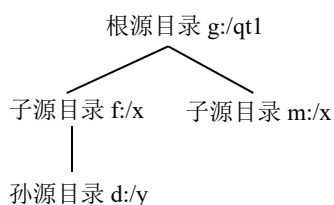


图 4.1 示例 4.1 的源树结构

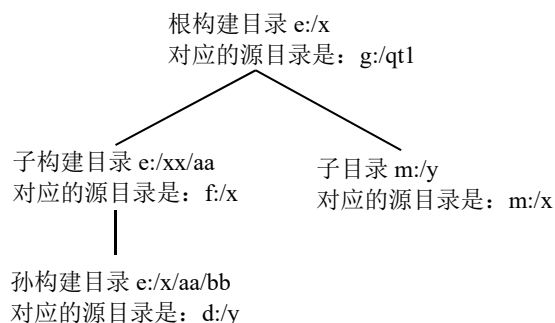


图 4.2 示例 4.1 的构建树结构

A、源树

本示例可以从各目录的 CMakeLists.txt 文件中非常容易的画出如图 4.1 所示的源树结构，由图可见，源树在逻辑上形成树形结构但实际目录并不一定形成树形结构。

B、构建树

根目录：从 CMD 输入的命令可见，构建树的根目录为

e:/x

子目录：从主 CMakeLists.txt 文件中可见，子构建目录有两个，一个是 m:/y，另一个是相对于当前构建目录的相对目录 aa，由于在执行主 CMakeLists.txt 文件时，其当前构建目录是 e:/x，所以另一个子构建目录是 e:/x/aa，最终得到本示例的两个子构建目录为：

m:/y 和 e:/x/aa

孙构建目录：从子目录 f:/x 下的子 CMakeLists.txt 文件中可知，孙构建目录是相对于当前构建目录的相对目录 bb，由于此时正在构建的是 f:/x/CMakeLists.txt 文件，所以，此时的当前构建目录是 e:/x/aa，所以，孙目录是：

e:/x/aa/bb

最终得出如图 4.2 所示的构建树结构。在以上目录中都可以找到一个名为 `makefile` 的文件，并且在 `e:/x` 目录下有一个名为 `CMakeCache.txt` 的文件，前文已讲过，该文件是构建树根目录的标志性文件。

从构建树结构可以看到，构建树与源树在物理上完全没有关系，而且构建树在逻辑上形成树形结构但实际目录并不一定形成树形结构。

示例 4-2: CMake 项目结构---构建文件、初步认识输出工件以及 `EXCLUDE_FROM_ALL` 参数

①、源文件准备

//g:/qt1/a.cpp (主源文件)

```
#include<iostream>
extern int b;
extern int c;
int main(){
std::cout<<"b="<<b<<" ;c="<<c<<std::endl;    return 0;}
```

//g:/qt1/b.cpp

```
int b=1;
```

//g:/qt1/c.cpp

```
int c=2;
```

//g:/qt1/d.cpp

```
int d=3;
```

②、CMakeLists 文件准备

#g:/qt1/CMakeLists.txt (主目录)

```
cmake_minimum_required(VERSION 3.27)
project(xx)
#注意以下变量的位置，需位于 add_subdirectory()之前
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY f:/x/a)    #指定静态库文件的输出目录
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY f:/x/b)    #指定可执行文件的输出目录
add_subdirectory(xx)
add_subdirectory(yy EXCLUDE_FROM_ALL)        #不被包含在父目录的 ALL 目标中
add_subdirectory(zz EXCLUDE_FROM_ALL)        #不被包含在父目录的 ALL 目标中
message(====根目录====)
add_executable(nn a.cpp)                      #根据 a.cpp 构建一个可执行文件
#目标 kk(构建于子目录 yy)在上面的代码中被排除在主目录外，但在以下语句中明确指定需要依赖于
#子目录 yy 构建的目录。
target_link_libraries(nn jj kk)               #将 jj 和 kk 链接到 nn(此步会调用链接命令，如 link.exe)
```

#g:/qt1/xx/CMakeLists.txt (子目录)

```
message(=====子目录 xx=====)
add_library(jj STATIC g:/qt1/b.cpp)
```

#g:/qt1/yy/CMakeLists.txt (子目录)

```
message(=====子目录 yy=====)
add_library(kk STATIC g:/qt1/c.cpp)
```

#g:/qt1/zz/CMakeLists.txt (子目录)

```
message(=====子目录 zz=====)
add_library(hh STATIC g:/qt1/d.cpp)
```

- ③、本示例的目的在于了解构建文件和输出工件之间的区别以及 `EXCLUDE_FROM_ALL` 参数的原理。输出工件是 CMake 的构建目标实际生成的真实文件，这些文件需要使用编译器来构建，而构建文件则是指的 `makefile` 等文件，构建文件不需要使用编译器构建。

- ④、本示例使用的以下变量

```
CMAKE_ARCHIVE_OUTPUT_DIRECTORY
CMAKE_RUNTIME_OUTPUT_DIRECTORY
```

分别用于指定构建的静态库文件和可执行文件的输出路径，静态库文件和可执行文件是 CMake 的输出工件之一，有关输出工件的更详细的内容，详见后文对输出工件的讲解，对于本示例只需了解其作用即可。

- ⑤、本示例使用的命令

`add_library()`: 用于构建一个库文件，该命令会导致调用 `cl.exe`、`lib.exe` 等编译命令

`add_executable()`: 用于构建一个可执行文件

`target_link_libraries()`: 此命令用于将构建出来的库文件链接起来，这会调用链接命令，如 `link.exe` 等命令。

更详细的内容详见后面章节讲解，对于本示例只需了解其作用即可。

- ⑥、本示例准备构建一个名为 `nn.exe` 的文件，该文件需要使用到 `a.cpp`、`b.cpp`、`c.cpp` 三个源文件，所以，本示例中的 `d.cpp` 文件是多余的，可以不用构建，`d.cpp` 在子目标 `g:/qt1/zz` 中构建，为此，在主 `CMakeLists.txt` 中使用 `EXCLUDE_FROM_ALL` 将 `g:/qt1/zz` 构建的目标从主目录的 `ALL` 目标(这是 `makefile` 目标，后文会讲解)中排除，排除之后将不会从主目录构建子目录的目标，但，使用明确依赖例外，如本例的子目录 `g:/qt1/yy`。

- ⑦、下面我们来看以上示例的具体运行结果。在 CMD 中转至 `g:/qt1` 并输入以下命令

```
cmake .
mingw32-make
```

- ⑧、查看目录结构

执行以上命令后，会在以下文件夹看到一个名称为 `makefile` 的文件

```
g:/qt1
g:/qt1/xx
g:/qt1/yy
g:/qt1/zz
```

在以下文件夹中的 `build.make` 文件(这是一个 `makefile` 文件)会调用编译器命令间接构建相应的目标，比如 `jj.dir/build.make` 会调用编译器的 `lib.exe` 命令构建 `jj.lib`，其余类似

```
G:\qt1\CMakeFiles\nn.dir
G:\qt1\xx\CMakeFiles\jj.dir
G:\qt1\yy\CMakeFiles\kk.dir
G:\qt1\zz\CMakeFiles\hh.dir
```

在 `f:/x/a` 文件夹中可以看到构建好的 `jj.lib` 和 `kk.lib` 两个静态库文件。

在 `f:/x/b` 文件夹中可以看到最终构建的 `nn.exe` 程序和 `nn.ilc`、`nn.pdb` 文件。

- ⑨、查看主 `makefile` 文件的内容，以查看 CMake 构建 `nn.exe` 程序的过程

用记事本打开 g:/qt1 目录(主目录)下的 makefile 文件, 可以看到该文件间接调用了 CMakeFiles\Makefile2 文件中的 all 目标(这是一个 makefile 目标)

用记事本打开 g:/qt1/ CMakeFiles 目录下的 Makefile2 文件(这是一个 makefile 文件), 可看到以下代码(以下代码的命令部分作了简化):

```
#g:/qt1/ CMakeFiles/ Makefile2
all: CMakeFiles/nn.dir/all ①
all: xx/all
.....
xx/all: xx/CMakeFiles/jj.dir/all ②
.....
CMakeFiles/nn.dir/all: xx/CMakeFiles/jj.dir/all ③
CMakeFiles/nn.dir/all: yy/CMakeFiles/kk.dir/all
    $(MAKE) -f CMakeFiles\nn.dir\build.make CMakeFiles/nn.dir/build
.....
xx/CMakeFiles/jj.dir/all: ④
    $(MAKE) -f xx\CMakeFiles\jj.dir\build.make xx/CMakeFiles/jj.dir/build
.....
yy/CMakeFiles/kk.dir/all: ⑤
    $(MAKE) -f yy\CMakeFiles\kk.dir\build.make yy/CMakeFiles/kk.dir/build
```

add_subdirectory()命令的 EXCLUDE_FROM_ALL 参数将子目录的目标排除 ALL 目标指的便是在①处被排除, 从以上代码可见, 在①处并未将排除的子目录 yy 和 zz 添加到 all 目标。但由于本示例在 target_link_libraries()命令中明确的指定了对子目录 yy 中的目标 kk 的依赖, 所以在③处将 yy 的 kk 目标添加到了主目录的依赖中, 最终在⑤处, 使用 make 命令间接调用 yy\CMakeFiles\kk.dir 目录中的 build.make 文件并间接调用编译器命令构建 kk.lib。

4.2 project()命令

4.2.1 project()命令

project()命令的语法为:

```
project(<pro-name> [<language-name>...])
project(<pro-name>
    [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]
    [DESCRIPTION <description-string>]
    [HOMEPAGE_URL <url-string>]
    [LANGUAGES <language-name>...])
```

1、该命令用于设置一个项目名称, 并设置一些变量的值(见表 4.2)。项目的顶级 CMakeLists.txt 文件必须

调用 `project()` 命令，若未调用，则会产生一个警告并假设有一个 `project(Project)` 的命令，并使用默认语言 C 和 CXX。`project()` 命令应在 `cmake_minimum_required()` 命令之后调用

- 调用 `project()` 命令后设置的变量如表 4.2 所示，设置的其余变量需由 `project()` 命令的其他选项决定，详见后文

4.2 调用 `project()` 命令后设置的变量

变量名	值
<code>PROJECT_NAME</code>	最近一次调用 <code>project()</code> 命令的项目名。
<code>CMAKE_PROJECT_NAME</code>	顶级项目名。需在顶级 <code>CMakeLists.txt</code> 调用 <code>project()</code> 命令。
<code>PROJECT_SOURCE_DIR</code>	最近一次调用 <code>project()</code> 命令的源目录(绝对路径)和二进制目录(即构建目录，绝对路径)
<code>PROJECT_BINARY_DIR</code>	
<code><pro-name>_SOURCE_DIR</code>	最近一次调用项目名称为 <code><pro-name></code> 的 <code>project()</code> 命令的源目录(绝对路径)和二进制目录(即构建目录，绝对路径)。可以理解为指定项目的源目录和二进制目录。
<code><pro-name>_BINARY_DIR</code>	
<code>PROJECT_IS_TOP_LEVEL</code>	最近一次调用的 <code>project()</code> 命令是否是顶级目录
<code><pro-name>_IS_TOP_LEVEL</code>	指定的项目是否为顶级项目

3、VERSION 参数

该参数用于指定项目的版本号，使用该参数必须启用 `CMP0048`，版本号使用“主版本号.次版本号.补丁.微调”(即 `major.minor.patch.tweak`) 的格式指定，该参数会设置表 4.3 所示的变量

4.3 `project()` 命令 VERSION 参数设置的变量

变量名	值
<code>CMAKE_PROJECT_VERSION</code>	顶级项目的版本号。需在顶级 <code>CMakeLists.txt</code> 调用 <code>project()</code> 命令
<code>PROJECT_VERSION</code>	最近一调用 <code>project()</code> 命令的版本号
<code><pro-name>_VERSION</code>	最近一次调用项目名称为 <code><pro-name></code> 的 <code>project()</code> 命令的版本号
<code>PROJECT_VERSION_MAJOR</code>	<code>project()</code> 命令设置的 <code>PROJECT_VERSION</code> 变量的主版本号(<code>major</code>)、次版本号(<code>minor</code>)、补丁版本号(<code>patch</code>)、微调版本号(<code>tweak</code>)
<code>PROJECT_VERSION_MINOR</code>	
<code>PROJECT_VERSION_PATCH</code>	
<code>PROJECT_VERSION_TWEAK</code>	
<code><pro-name>_VERSION_MAJOR</code>	由 <code>project()</code> 命令设置的 <code><pro-name>_VERSION</code> 变量的主版本号(<code>major</code>)、次版本号(<code>minor</code>)、补丁版本号(<code>patch</code>)、微调版本号(<code>tweak</code>)
<code><pro-name>_VERSION_MINOR</code>	
<code><pro-name>_VERSION_PATCH</code>	
<code><pro-name>_VERSION_TWEAK</code>	

4、DESCRIPTION 参数

该参数设置一个项目的简短描述，通常是一个较短的字符串，该选项会设置以下变量

- `CMAKE_PROJECT_DESCRIPTION`: 顶级项目的简短描述(即 `DESCRIPTION` 参数的值)。需在顶级 `CMakeLists.txt` 调用 `project()` 命令
- `PROJECT_DESCRIPTION`: 最近一调用 `project()` 命令的简短描述(即 `DESCRIPTION` 参数的值)。
- `<pro-name>_DESCRIPTION`: 最近一次调用项目名称为 `<pro-name>` 的 `project()` 命令的简短描述(即 `DESCRIPTION` 参数的值)。

5、HOMEPAGE_URL 参数

该参数设置一个项目的主页 URL，该选项会设置以下变量

- CMAKE_HOMEPAGE_URL：顶级项目的 URL。需在顶级 CMakeLists.txt 调用 project() 命令
- PROJECT_HOMEPAGE_URL：最近一调用 project() 命令的 URL
- <pro-name>_HOMEPAGE_URL：最近一次调用项目名称为<pro-name>的 project() 命令的 URL

6、LANGUAGES 参数

该参数设置项目使用的语言，CMake 支持 C，CXX(即 C++)，CSharp，CUDA，OBJC，OBJCXX 等语言，若未指定语言，则默认启用 C 和 CXX 语言。如果启用 ASM，则最后列出它。若将语言指定为 NONE 或使用了 LANGUAGES 关键字而不指定任何语言，则跳过启用任何语言。

示例 4.3：理解最近一次调用 project() 命令

①、在目录 g:/qt1 下的 CMakeLists.txt 中编写如下代码

```
cmake_minimum_required(VERSION 3.27)
project(xx)
project(xx1)
message(AA1=${PROJECT_NAME})
project(xx2)
message(AA2=${PROJECT_NAME})
```

②、在 CMD 中转至 g:/qt1 并输入以下命令

cmake .

以上命令依次输出 AA1=xx1、AA2=xx2。变量 PROJECT_NAME 的值是最近一次调用 project() 命令中的项目名称。

示例 4.4：理解 project() 命令设置的各变量的值

①、在目录 g:/qt1 下的 CMakeLists.txt 中编写如下代码

```
cmake_minimum_required(VERSION 3.27)
project(xx)
add_subdirectory(xx)          #添加子目录，该命令详见后文
add_subdirectory(yy)
message(=====顶级项目=====)
message(\n 最近项目及顶级项目名称)
message(AA1=${PROJECT_NAME})
message(AA2=${CMAKE_PROJECT_NAME})
message(\n 最近项目及指定项目的源目录)
message(AA3=${PROJECT_SOURCE_DIR})
message(AA4=${xx_SOURCE_DIR})
message(\n 最近及指定项目是否是顶级项目)
message(AA5=${PROJECT_IS_TOP_LEVEL})
message(AA6=${xx_IS_TOP_LEVEL})
message(=====\n)
```

②、在目录 g:/qt1/xx 下的 CMakeLists.txt 中编写如下代码


```

message(====子目录xx--无project命令====)
message(\n 最近项目及顶级项目名称)
message(BB1=${PROJECT_NAME})
message(BB2=${CMAKE_PROJECT_NAME})
message(\n 最近项目及指定项目的源目录)
message(BB3=${PROJECT_SOURCE_DIR})
message(BB4=${mm_SOURCE_DIR})
message(\n 最近及指定项目是否是顶级项目)
message(BB5=${PROJECT_IS_TOP_LEVEL})
message(BB6=${mm_IS_TOP_LEVEL})
message(=====\n)

```

③、在目录 g:/qt1/yy 下的 CMakeLists.txt 中编写如下代码

```

project(mm)
message(====子目录yy--mm项目====)
message(\n 最近项目及顶级项目名称)
message(CC1=${PROJECT_NAME})
message(CC2=${CMAKE_PROJECT_NAME})
message(\n 最近项目及指定项目的源目录)
message(CC3=${PROJECT_SOURCE_DIR})
message(CC4=${mm_SOURCE_DIR})
message(\n 最近及指定项目是否是顶级项目)
message(CC5=${PROJECT_IS_TOP_LEVEL})
message(CC6=${mm_IS_TOP_LEVEL})
message(=====\n)

```

④、在 CMD 中转至 g:/qt1 并输入以下命令

cmake .

以上命令的输出如图 4-3 所示。本示例使用了 add_subdirectory() 命令，使用该命令后 CMake 会执行子目录中的 CMakeLists.txt 文件的内容。从输出的 CC1=mm 和 CC2=xx 可以看出最近调用的 project() 命令的项目名称与顶级项目名称的区别，从 BB3=G:/qt1 和 BB4=G:/qt1/yy 可以看出最近调用的 project() 命令的源目录与指定项目的源目录的区别。

```

G:\qt1>cmake .
====子目录xx--无project命令====
最近项目及顶级项目名称
BB1=xx
BB2=xx
最近项目及指定项目的源目录
BB3=G:/qt1
BB4=G:/qt1/yy
最近及指定项目是否是顶级项目
BB5=ON
BB6=OFF
=====

====子目录yy--mm项目====
最近项目及顶级项目名称
CC1=mm
CC2=xx
最近项目及指定项目的源目录
CC3=G:/qt1/yy
CC4=G:/qt1/yy
最近及指定项目是否是顶级项目
CC5=OFF
CC6=OFF
=====

=====顶级项目=====
最近项目及顶级项目名称
AA1=xx
AA2=xx
最近项目及指定项目的源目录
AA3=G:/qt1
AA4=G:/qt1
最近及指定项目是否是顶级项目
AA5=ON
AA6=ON
=====

```

图 4-3 示例 4.4 的输出

4.2.2 project()命令调用期间的执行步骤

1、了解 project()命令的执行步骤，可以使我们在执行期间的不同点包含一些指定的文件来执行一些其他的额外工作。在执行 project()之前会执行以下步骤：

- 1)、对于每个 project()调用，无论项目名称如何，都会包含 CMAKE_PROJECT_INCLUDE_BEFORE 变量指定的文件。适用 3.15 及以上版本。
- 2)、如果 project()命令指定<pro-name>作为其项目名，则包含 CMAKE_PROJECT_<pro-name>_INCLUDE_BEFORE 变量指定的文件。适用 3.17 及以上版本。
- 3)、前文介绍的 project()命令设置的各种变量。
- 4)、对于第一个 project()，只调用：
 - ◆ 如果设置了 CMAKE_TOOLCHAIN_FILE (工具链文件)，则至少读取一次。它可以被多次读取，也可以在以后启用语言时再次读取。
 - ◆ 设置描述主机和目标平台的变量。此时还可能会设置特定于语言的变量(比如，CMAKE_<LANG>_COMPILER 等)，也可能不会设置。在第一次运行时，可能定义的唯一特定于语言的变量是工具链文件可能已经设置的变量。在随后的运行中，可能会设置从前一次运行中缓存的特定于语言的变量。
 - ◆ 包含 CMAKE_PROJECT_TOP_LEVEL_INCLUDES 变量指定的文件，此后，CMake 将忽略该变量。适用 3.24 及以上版本。
- 5)、启用调用中指定的语言，若未提供，则使用默认语言。第一次启用语言时，可能会重新读取工具链文件。
- 6)、对于每个 project()调用，无论项目名称如何，则包含 CMAKE_PROJECT_INCLUDE 变量指定的文件。
- 7)、如果 project()命令指定了项目名<pro-name>，则包含 CMAKE_PROJECT_<pro-name>_INCLUDE 指定的文件。

2、以上步骤共涉及以下变量

- 1)、CMAKE_PROJECT_INCLUDE_BEFORE:

该变量是 project()命令调用的第一步，用于指定一个 CMake 文件。

- 2)、CMAKE_PROJECT_<pro-name>_INCLUDE_BEFORE

该变量是项目名称为<pro-name>的 project()命令调用的第一步，用于指定一个 CMake 文件。

- 3)、CMAKE_TOOLCHAIN_FILE

该变量用于指定工具链文件的路径。允许使用相对路径，并且首先解释为相对于构建目录，如果没有找到，则解释为相对于源目录。工具链主要用于交叉编译

- 4)、CMAKE_PROJECT_TOP_LEVEL_INCLUDES

适用于 3.24 及以上版本，该变量是一个以分号分隔的 CMake 文件列表，作为第一个 project()调用的一部分。此变量用于指定为构建执行一次性设置的文件，主要是为用户添加特定于环境的设

置，而不是用于指定工具链的详细信息，如添加自定义的构建类型等。默认为空。

5)、CMAKE_PROJECT_INCLUDE

该变量是 `project()` 命令调用的最后一步，用于指定一个 CMake 文件。

6)、CMAKE_PROJECT_<pro-name>_INCLUDE

该变量是项目名称为 <pro-name> 的 `project()` 命令调用的最后一步，用于指定一个 CMake 文件。

示例 4.5: `project()` 命令执行步骤

- ①、在目录 `g:/qt1` 中的 `CMakeLists.txt` 中编写如下代码

```
cmake_minimum_required(VERSION 3.27)
#以下变量设置应位于 project() 命令之前
set(CMAKE_PROJECT_xx_INCLUDE g:/qt1/e.txt)
set(CMAKE_PROJECT_INCLUDE g:/qt1/d.txt)
set(CMAKE_PROJECT_TOP_LEVEL_INCLUDES g:/qt1/c.txt)
set(CMAKE_PROJECT_xx_INCLUDE_BEFORE g:/qt1/b.txt)
set(CMAKE_PROJECT_INCLUDE_BEFORE g:/qt1/a.txt)
project(xx)
```

- ②、在 `g:/qt1` 中的 `a.txt` 中编写如下代码

```
message(AA)
```

- ③、在 `g:/qt1` 中的 `b.txt` 编写如下代码

```
message(BB)
```

- ④、在 `g:/qt1` 中的 `c.txt`、`d.txt`、`e.txt` 中如以上方法编写类似代码

- ⑤、在 CMD 中转至 `g:/qt1` 并输入以下命令

```
cmake .
```

依次输出 AA、BB、CC、DD、EE，从输出的内容可以见到 `project()` 命令的执行步骤。

第 5 章 使用 CMake 构建库文件和可执行文件

注：学习本章必须熟悉在命令行使用 `cl.exe`、`link.exe` 等工具构建 `dll` 文件的方法。`dll` 相关内容详见本人所作相关章节，从略。

5.1 CMake 目标的分类

- 1、使用 CMake 的最终目的是根据源文件使用编译命令构建出一个需要的目标，对于 C++ 语言，这个目标是 C++ 程序文件，如 `.lib`、`.exe`、`.dll` 等文件，但 CMake 不会全程参与构建这个目标，CMake 只负责第一个阶段，即生成阶段，这一阶段会生成构建文件(如 `makefile` 文件)，然后在之后的构建阶段由构建工具(如 `mingw32-make`)执行构建文件的代码最终间接调用编译器来构建目标并最终生成需要的程序文件。
- 2、由于目标贯穿整个构建过程，在各个阶段目标的名称和表现形式可能会各不相同，所以，在这里有必要对目标进行区分，本文将 CMake 文件中的目标称为 CMake 目标，`makefile` 文件中的目标称为 `makefile` 目标，最终构建出的程序文件称为最终目标。因此，CMake 的目的就是要从一个 CMake 目标构建出最终目标，最终目标通常是库文件、可执行文件、共享库文件等真实存在的文件，当然，构建出最终目标的过程还会生成一些附加的文件。
- 3、CMake 对目标的描述

要构建一个最终目标通常会有很多条件，比如需要创建的目标类型(如库文件、可执行文件等)、目标所依赖的文件、需要的头文件、需要的编译参数、需要的链接参数等，CMake 分别使用命令、属性和变量来对目标的这些条件进行描述。CMake 使用以下方法来指定需要构建的最终目标的类型

- 1)、CMake 使用 `add_library()` 命令来指定需要构建的各种库文件(静态库、动态库、导入库等)，该命令会在生成的构建文件中间接调用编译器(以 VC++ 为例)的 `cl.exe`、`lib.exe`、`link.exe` 等命令，具体调用什么命令依所构建的库而定。
- 2)、CMake 使用 `add_executable()` 命令来指定需要构建的可执行程序文件(如 `.exe` 文件)，该命令会在生成的构建文件中间接调用编译器的编译命令和链接命令，比如 VC++ 的 `cl.exe`、`link.exe` 等命令，具体调用什么命令依所构建的程序而定。
- 3)、CMake 使用 `target_link_libraries()` 命令来链接各种文件，该命令会在生成的构建文件中间接调用编译器的链接命令，比如 VC++ 的 `link.exe` 命令，该命令比较复杂，会在后文详细介绍，但本章需要懂得对 `target_link_libraries()` 命令的基本使用方法。

4、CMake 目标的分类

- 1)、根据不同的最终目标和实际构建的需要，CMake 将目标分为二进制目标和伪目标。
- 2)、二进制目标是指的最终会构建成二进制文件的目标，二进制文件是指的程序文件，如静态库文件、共享库文件、可执行文件等文件，这些文件是真实存在的程序文件。因此，CMake 的二进制目标又分为静态库目标、共享库(或动态库)目标、模块库目标、可执行目标、对象库目标。本文将

静态库目标简称为静态库，其余 CMake 目标作类似简称。所以，在本文应注意区分静态库、静态库目标、静态库文件等类似名称的区别，带有文件二字如静态库文件，则意味着是真实存在的文件，带有目标(或不带)二字如静态库(目标)，则只是一个 CMake 目标，这是个逻辑上的名称不是指的真实文件。表 5.1 对此进行了总结

3)、伪目标是指最终不会构建真实文件的目标，又分为导入目标、别名目标、接口库目标，详见表 5.2。

5.1 CMake 二进制目标

目标名称	命令	对应的最终目标
可执行目标	add_executable()	如.exe 文件。根据平台不同而不同
静态库目标	add_library(STATIC)	如.a、.lib 文件。根据平台不同而不同
共享库目标	add_library(SHARED)	如：.so、.dll 文件。根据平台不同而不同
模块库目标	add_library(MODULE)	如：.so、.dll 文件。模块库和共享库的区别见后文
对象库目标	add_library(OBJECT)	如.obj 文件。对象库目标不会实际生成库文件。

表 5.2 CMake 的伪目标

目标名称	分类	命令	简介
导入目标	导入可执行目标	add_executable(IMPORTED)	将已存在文件导入为 CMake 的目标
	导入库目标	add_library (IMPORTED)	
别名目标	别名可执行目标	add_executable(ALIAS)	可执行目标的别名
	别名库目标	add_library (ALIAS)	库的别名
接口库目标		add_library (INTERFACE)	可在目标上设置属性，设置的这些属性不能作用于目标本身，只能供其他目标使用

5.2 add_library()命令

add_library()命令可以构建多个不同的目标，有多个变体，下面分别介绍

5.2.1 add_library()命令基本语法

其语法为：

```
add_library(<name> [STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL] [<source>...])
```

该命令表示向项目添加一个名为<name>的库目标，该目标将从列出的源文件<source>...构建。各参数意义如下：

1、<name>参数

<name>在项目中必须是唯一的，<name>是一个逻辑目标名称，构建的实际的库文件名称依所使用的系统而定，如，g++编译器的名称类似 lib<name>.a，VS 编译器的名称类似为<name>.lib。

2、<source>...参数

该参数用于指定构建的库目标所需要的源文件列表。对于 3.11 及以上版本，若之后使用 `target_source()` 添加源文件，则可省略该参数。对于 3.1 及以上版本的 <source> 参数可使用形式为 `$<...>` 的生成器表达式。

3、STATIC、SHARED、MODULE 参数

以上参数用于指定库的类型，具体为：

- **STATIC** 表示静态库，后缀通常为 .a 或 .lib。
- **SHARED** 表示共享库(或称为动态库)，后缀通常为 .so 或 .dll。
- **MODULE** 表示模块库，后缀通常为 .so 或 .dll。模块库和共享库的区别见后文
- 默认类型

若未显示指定库的类型，则库的类型由变量 `BUILD_SHARED_LIBS` 的值来决定，若为 `true`，则为 **SHARED** 类型，否则为 **STATIC** 类型。

- **SHARED** 和 **MODULE** 类型会自动将目标属性 `POSITION_INDEPENDENT_CODE` 的值设置为 `ON`，该目标属性的作用是决定是否创建与位置无关的可执行文件或共享库，若在创建目标时设置该属性，则值由 `CMAKE_POSITION_INDEPENDENT_CODE` 变量初始化。

- **macOS 相关**

若将静态库或共享库目标的 `FRAMEWORK` 目标属性设置为 `TRUE`，则将在 macOS 和 iOS 上构建框架包(Framework Bundle)。如果在创建目标时设置 `FRAMEWORK` 属性，则由 `CMAKE_FRAMEWORK` 变量的值初始化。

4、EXCLUDE_FROM_ALL 参数

若指定了 `EXCLUDE_FROM_ALL`，则 `EXCLUDE_FROM_ALL` 目标属性将被设置为 `TRUE`。若 `EXCLUDE_FROM_ALL` 目标属性被设置为 `TRUE`，将从包含目录及其祖先目录的 `all` 目标中排除该目标(读者可通过查看生成的 `makefile` 文件中的 `all` 目标以理解该属性)，此时，`make` 工具不会构建目标，若该属性的值为 `FALSE`(默认)则目标不会被排除。说简单一点就是，若 `EXCLUDE_FROM_ALL` 目标属性被设置为 `TRUE`，则构建工具不会构建该目标，若为 `FALSE`(默认)，则会构建该目标。但是，即使将 `EXCLUDE_FROM_ALL` 设置为 `TRUE`，目标仍可能会在 `install(TARGETS)` 命令中列出。详细原理可参阅 4.1 小节 `add_subdirectory()` 命令的同名参数。

5、模块库和共享库的区别(以 Windows 为例)

- 1)、模块库是不链接到其他目标的插件，但可以在运行时使用类似于 `dlopen` 的函数动态加载，模块库是一种使用运行时技术作为插件加载的类型。
- 2)、CMake 中的模块库对于 Windows 而言仍然是一个后缀为 .dll 的文件(与共享库后缀相同)，但该 .dll 文件没有导入库。若源代码含有导出符号，则在创建 DLL 文件时，通常会默认创建一个导入库文件(.lib)和导出文件(.exp)。但是，若源代码没有导出任何符号，则这样创建出来的 DLL 不会创建导入库和导出文件，CMake 将这类 DLL 称为模块库文件，使用 `add_library(MODULE)` 来创建，实际上，只要是使用 `add_library(MODULE)` 创建的 DLL，无论有无导出符号都会被 CMake 认为是模块库。注意：dll 文件的导入库文件和静态库文件的后缀都是 .lib，但这是两种完全不同类型的文件。
- 3)、模块库目标的一个重要特点是，模块库目标不能出现在 `target_link_libraries()` 命令的右侧，即模块库目标不能被链接。但模块库可以链接静态目标、接口目标、对象目标、`add_library(SHARED)` 创建的目标。

比如：

```
add_library(jj MODULE g.cpp)           // 创建模块库
```

```
add_library(kk STATIC a.cpp)
target_link_libraries(kk jj)           //错误，模块库 jj 不能被链接到 kk
```

再如：

```
add_library(jj STATIC h.cpp)
add_library(kk MODULE g.cpp)
target_link_libraries(kk jj)           //正确，模块库 kk 可以链接静态库
```

- 6、本小节涉及到的变量和属性如表 5.3 所示，其中变量、目标属性、目录属性的区别在于其作用范围不同，设置变量可作用于该变量之后的所有目标，目标属性仅对单个目标有效，目录属性则对该目录中的所有目标有效。

表 5.3 add_library()命令涉及的部分变量和属性

类别	名称	说明
变量	BUILD_SHARED_LIBS	该变量决定 add_library()命令构建的目标的默认类型，若为 true，则为 SHARED 类型，否则为 STATIC 类型。
	CMAKE_POSITION_INDEPENDENT_CODE	用于初始化 FRAMEWORK 目标属性
	CMAKE_FRAMEWORK	用于初始化 POSITION_INDEPENDENT_CODE 目标属性
目标属性	POSITION_INDEPENDENT_CODE	决定是否创建与位置无关的可执行文件或共享库，若在创建目标时设置该属性，则值由 CMAKE_POSITION_INDEPENDENT_CODE 变量初始化。
	FRAMEWORK	若在将静态库或共享库目标的 FRAMEWORK 目标属性设置为 TRUE，则将在 macOS 和 iOS 上构建框架包(Framework Bundle)。如果在创建目标时设置 FRAMEWORK 属性，则由 CMAKE_FRAMEWORK 变量的值初始化。
	EXCLUDE_FROM_ALL	若 EXCLUDE_FROM_ALL 目标属性被设置为 TRUE，将从包含目录及其祖先目录的 all 目标中排除该目标
目录属性	EXCLUDE_FROM_ALL	若 EXCLUDE_FROM_ALL 目标属性被设置为 TRUE，将从包含目录及其祖先目录的 all 目标中排除该目标

示例 5.1：构建一个静态库文件

- ①、在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp (主源文件)
#include<iostream>
extern int b;
int main(){
    std::cout<<"b="<<b<<std::endl;    return 0;}

```

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
int b=1;

```

- ②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(hh)
add_library(jj STATIC a.cpp b.cpp)

```

- ③、在 CMD 命令行中转到 a.cpp 和 b.cpp 所在目录，并输入以下代码(注意末尾有一小数点)

```
cmake .
mingw32-make
```

然后便可在 g:/qt1 目录看到本示例生成的名为 jj.lib 的静态库文件。

示例 5.2：构建一个共享库文件

①、C++源文件准备

在目录 g:/qt1 中的 f.cpp 中编写如下代码

```
//g:/qt1/f.cpp (用于生成 dll 文件)
```

```
//使用__declspec(dllexport)声明变量，否则不能生成 dll 文件的导入库文件
__declspec(dllexport) int a=1;
```

在目录 g:/qt1 中的 e.cpp 中编写如下代码

```
//g:/qt1/a.cpp (主源文件)
```

```
#include<iostream>
__declspec(dllimport) int a;      //导入名称 a
int main(){
    std::cout<<"a="<<a<<std::endl;    return 0;}

```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.27)
project(hh)
add_library(kk SHARED f.cpp)      #目标名必须与 e.cpp 中指定的一致
add_executable(nn e.cpp)
target_link_libraries(nn kk)
```

③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```
cmake .          #生成阶段
mingw32-make     #构建阶段
nn.exe           #测试。输出 a=1，符合预期
```

④、查看构建文件以了解构建过程

打开以下目录

```
CMakeFiles\nn.dir
```

需要熟悉 build.make、link.txt、objects1.rsp、flags.make 文件的内容。在 link.txt 中可以找到类似以下的语句

```
link.exe .... kk.lib
```

说明本示例链接了由 add_library()命令生成的 kk.lib 文件，该文件是 dll 的导入库文件，注意：kk.lib 不是静态库文件，虽然他们后缀相同。

⑤、查看生成的文件

在 g:/qt1 目录下可以看到生成的以下重要的 C++程序文件

```
kk.dll、kk.lib、kk.exp、nn.exe
```

其中 kk.lib 是 kk.dll 的导入库文件(注意，不是该文件不是静态库文件，虽然他们后缀相同)，kk.exp 是导出文件

对象文件，如 kk.obj 被构建在 G:\qt1\CMakeFiles\kk.dir 目录中，nn.obj 被构建在 G:\qt1\CMakeFiles\nn.dir 目录中。

5.2.2 对象库目标(简称对象库)

CMake 构建对象库目标的语法为：

```
add_library(<name> OBJECT [<source>...])
```

以上命令创建一个名为 name 的对象库目标，对象库不会生成库文件，即不能在相应目录中类似找到 name.lib 的文件，但会生成对象文件(即.obj 文件)，并且可在 add_library()、add_executable()等命令中引用这个对象库目标以作为这些命令的源文件，引用的方式为\${TARGET_OBJECTS:name}，这是一个生成表达式。有关对象库的其他使用方法详见第 7 章

示例 5.3：对象库的使用

- ①、在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp
#include<iostream>
extern int b;
int main(){    std::cout<<b<<std::endl;    return 0;}
```

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
int b=1;
```

- ②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXX)
#创建一个名为 jj 的对象库
add_library(jj OBJECT b.cpp)
#将对象库 jj 和 a.cpp 作为库文件 kk 的源文件
add_library(kk STATIC a.cpp ${TARGET_OBJECTS:jj})
```

- ③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```
cmake .
mingw32-make
```

然后便可在 g:/qt1 目录中看到本示例生成的名为 kk.lib 的静态库文件，输入以下命令便可看到 kk.lib 包含了 a.cpp.obj 和 b.cpp.obj 两个对象文件，注：lib /LIST 用于显示.lib 文件包含的文件。

```
lib /LIST kk.lib
```

5.2.3 接口库目标(简称接口库)

CMake 构建接口库目标的语法为：

```
add_library(<name> INTERFACE)
```

3.19 及以上版本，可以为接口库指定源文件。其语法为：

```
add_library(<name> INTERFACE [<source>...] [EXCLUDE_FROM_ALL])
```

- 1、以上命令创建一个名为 **name** 的接口库目标。接口库目标的主要作用是用于传递使用要求(Usage Requirement)，使用要求是一个以 **INTERFACE_** 开头的目标属性，也就是说，接口库目标的主要作用是传递属性。接口库目标不会编译源代码也不会生成任何文件，或者说接口库目标不会调用编译器命令构建 C++ 程序，所以没有 **LOCATION** 属性，但是接口库目标可以设置 **INTERFACE_*** 属性，并可以传递这些属性给其他目标，传递属性需使用 **target_link_libraries()** 命令。有关使用要求及 **INTERFACE_*** 属性的详细内容，详见后文。
- 2、3.15 及以上版本
接口库目标可以具有 **PUBLIC_HEADER** 和 **PRIVATE_HEADER** 属性，可以使用 **install(TARGETS)** 命令安装这些属性指定的头文件。
- 3、3.19 及以上版本
若接口库目标具有源文件(即设置了 **SOURCES** 目标属性)或头文件集(header sets)(即设置了 **HEADER_SETS** 目标属性)，则该接口库目标就像通过 **add_custom_target()** 命令定义的目标一样，将作为构建目标出现在生成的构建系统(buildsystem)中，它不编译任何源代码，但包含由 **add_custom_command()** 命令创建的自定义命令的构建规则。
- 4、**EXCLUDE_FROM_ALL** 参数详见 **add_subdirectory()** 命令的同名参数
- 5、有关接口库目标的更详细的使用方法详见后文。

示例 5.4：使用接口库目标传递 **INTERFACE_*** 属性

- ①、在目录 **g:/qt1** 中的 **a.cpp** 中编写如下代码

```
//g:/qt1/a.cpp
#include<iostream>
using namespace std;
int main(){    cout<<"EE="<<DD<<endl;        //其中，DD 需使用编译工具-D 参数传递一个宏
return 0;}
```

- ②、在目录 **g:/qt1** 中的 **CMakeLists.txt** 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXXX)
add_library(jj INTERFACE )          #创建一个名为 jj 的接口库目标
#设置一个 INTERFACE_* 属性，该属性为编译命令传递一个“-DDD=3”的参数
set_property(TARGET jj PROPERTY INTERFACE_COMPILE_OPTIONS -DDD=3)
add_executable(nn a.cpp)           #根据 a.cpp 创建一个可执行程序 nn
target_link_libraries(nn jj)       #将接口库目标 jj 设置的属性传递给目标 nn
```

- ③、在 **CMD** 中转至 **g:/qt1** 并输入以下命令，以验证结果

```
cmake .
mingw32-make
nn.exe                #测试。输出 EE=3，符合预期
```

④、从示例可见，成功的将接口库目标 `jj` 设置的属性传递给了目标 `nn`。

⑤、查看生成的文件结构

打开 `G:\qt1\CMakeFiles` 目录，发现该目录下只有一个名为 `nn.dir` 的文件夹，没有名为 `jj.dir` 的文件夹，这说明 CMake 不会为目标 `jj` 构建任何 C++ 程序文件。使用记事本打开 `G:\qt1\CMakeFiles\nn.dir` 文件夹中的 `flags.make` 文件，可在其中发现以下内容

```
CXX_FLAGS = ..... -DDD=3
```

其中的 `CXX_FLAGS` 变量的值会作为 `G:\qt1\CMakeFiles\nn.dir\build.make` 文件中构建 `a.cpp.obj` 文件的 `cl.exe` 命令的参数，即，`CXX_FLAGS` 变量的值会被传递给编译器的 `cl.exe` 命令。

5.2.4 导入库目标(简称导入目标)

其语法为

```
add_library(<name> <type> IMPORTED [GLOBAL])
```

1、创建一个名为 `name` 类型为 `type` 的导入库目标，并将 `IMPORTED` 属性设置为 `TRUE`，其作用域为创建它的目录及子目录，但可以设置为 `GLOBAL` 作用域。导入目标的详细信息需要通过设置以 `IMPORTED_` 和 `INTERFACE_` 开头的属性来指定，例如，使用 `IMPORTED_LOCATION` 目标属性设置导入目标在磁盘上的完整路径等等。

2、使用导入库目标的主要目的是将已存在的文件导入为 CMake 目标，并可以在 CMake 中像使用其他常规目标一样使用导入库目标，因此，导入库目标不会编译源代码(即不会被构建)，或者说导入库目标不会调用编译器命令构建 C++ 程序。也就是说，这里只是创建了一个名称，并未创建任何真实的文件，但这个名称可以像其他目标一样在 CMake 项目中被引用，导入目标的详细信怎使用 `IMPORTED_` 和 `INTERFACE_` 开头的目标属性来指定。

2、<type>参数用于指定导入目标的类型，必须是以下取值之一：

`STATIC`、`SHARED`、`MODULE`、`UNKNOWN`、`OBJECT`、`INTERFACE`

3、特别注意：导入库目标的简称不是导入库而是导入目标。导入库在本文通常是指的 `.dll` 文件对应的导入库(一个后缀与静态库相同的 `.lib` 文件)

4、更详细的内容请参阅后文第 10 章对导入/导出目标的讲解。

示例 5.5：导入静态库目标

①、源文件准备

在目录 `g:\qt1` 中的 `a.cpp` 中编写如下代码

```
//g:/qt1/a.cpp(主源文件)
#include<iostream>
extern int b;
extern int c;
int main(){ std::cout<<"b="<<b<<" ;c="<<c<<std::endl;    return 0;}
```

在目录 `g:\qt1` 中的 `b.cpp` 中编写如下代码

```
//g:/qt1/b.cpp
int b=1;
```

在目录 g:/qt1 中的 c.cpp 中编写如下代码

```
//g:/qt1/c.cpp
int c=2;
```

②、准备需要导入的静态库文件

- A、本示例准备将由 b.cpp 的构建的静态库文件导入到 CMake 中。
- B、在 CMD 中转至 g:/qt1 并输入以下命令创建一个对象文件 b.obj

```
cl /DWIN32 /D_WINDOWS /EHsc /Ob0 /Od /RTC1 -MDd -Zi /c b.cpp
```

使用这么多参数的原因是因为 CMake 创建的.obj 文件使用了这些参数。直接使用/c 参数创建.obj 文件，与使用以上参数创建的.obj 文件不能被链接在一起使用。

- C、接着使用 lib 命令创建一个静态库文件 b.lib，命令如下

```
lib b.obj
```

③、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt

cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj STATIC IMPORTED)           #导入一个静态库目标 jj
add_executable(nn a.cpp c.cpp)           #根据 a.cpp、c.cpp 创建一个可执行文件 nn.exe

#指定目标 jj 所关联的文件，必须是完整路径名，此处指定的文件会在链接阶段链接到目标 nn
set_property(TARGET jj PROPERTY IMPORTED_LOCATION g:/qt1/b.lib)

target_link_libraries(nn jj)              #将 jj 链接到目标 nn
```

④、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make
nn.exe           #测试。输出 b=1;c=2，符合预期
```

⑤、在 CMakeFiles\nn.dir 目录下，打开 link.txt 文件，可以看到 link 命令链接了 b.lib 文件。

示例 5.6：导入目标---导入 DLL 文件

①、源文件准备

在 g:/qt1 目录下的 f.cpp 中编写如下代码

```
//f.cpp(用于生成 DLL 文件)
__declspec(dllexport) int a=1;
```

在 g:/qt1 目录下的 e.cpp 中编写如下代码

```
//e.cpp(exe 文件)
#include<iostream>
__declspec(dllimport) int a;
int main(){      std::cout<<"E="<<a<<std::endl;      return 0;}
```

②、准备需要导入的文件

在 CMD 中转至 g:/qt1 并输入以下命令

cl /LD f.cpp //创建 f.dll 和 f.lib(导入库)两个文件，我们将在之后导入这两个文件到 CMake 项目

③、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

#g:/qt1/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.27)
project(XXXX)
add_library(kk SHARED IMPORTED)    #创建一个 SHARED 类型的导入目标 kk
#指定主库文件(即.dll)的位置，此步骤不是必须。
set_property(TARGET kk PROPERTY IMPORTED_LOCATION g:/qt1/f.dll)
#指定 DLL 文件对应的导入库文件，此步骤是必须项，不能省略。
set_property(TARGET kk PROPERTY IMPORTED_IMPLIB f.lib)
add_executable(nn e.cpp )
target_link_libraries(nn kk)
```

④、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make
nn.exe           //测试结果，符合预期。
```

5.2.5 库目标的别名(别名库目标)

其语法为：

add_library(<name> ALIAS <target>)

- 1、给目标 target 创建一个别名 name，别名不会作为 make 目标出现在生成的构建系统中
- 2、别名不能用于以下情形：
 - 不能作为 make 的目标出现在生成的构建系统中
 - 不能用于修改目标 target 的属性，即，不能作为 set_property()、set_target_properties()、target_link_libraries()等命令的操作数，
 - 不能用于安装或导出。
- 3、别名可用于以下情形：
 - 可以用作链接的目标
 - 可以使用别名来读取自定义(custom)命令和自定义目标的可执行文件的属性。
 - 可以使用 if(TARGET)来测试别名是否存在。
- 4、3.11 及以上版本可将别名用于 GLOBAL 导入库目标。在 3.18 及以上版本支持非 GLOBAL 导入目标，此时的别名的作用域是创建它的目录和子目录，目标属性 ALIAS_GLOBAL 可用于检查别名是否是全局的。
- 5、可以通过读取 ALIASED_TARGET 目标属性来测试一个名称是否为别名。

5.3 add_executable()命令

5.3.1 基本语法

其语法为:

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE] [EXCLUDE_FROM_ALL] [source1] [source2...])
```

- 1、该命令表示添加一个名为 `name` 的可执行目标，该目标将从列出的源文件 `source1`, `source2...` 构建。若之后使用 `target_source()` 添加源文件，则可省略参数 `source`。3.1 及以上版本的 `source` 参数可使用形式为 `$<...>` 的生成器表达式。该命令的基本用法在前文的章节已使用过多次，示例从略。
- 2、各参数意义如下：

- `<name>`
在项目中必须是唯一的，`name` 是一个逻辑目标名称，构建的实际的可执行文件名称依所使用的平台而定，如，windows 平台的名称是 `<name>.exe`。
- `[source1] [source2...]`
`add_executable()` 命令可以指定 `.cpp`、`.obj` 等文件，但不能指定 `.lib` 等库文件，`.lib` 等库文件应使用 `target_link_libraries()` 命令进行链接。
- `WIN32` 参数
用于构建 MFC 程序，若 `WIN32` 被指定，则 `WIN32_EXECUTABLE` 目标属性将被设置为 `ON`，此时，在 windows 上，将使用 `WinMain()` 作为入口点构建目标文件，而不是 `main()`。注：`WinMain` 是构建 MFC 程序的入口点函数。
- `MACOSX_BUNDLE` 参数
用于在 macOS 或 iOS 上构建可执行文件作为应用程序包(Application Bundle)。若该参数被指定，则 `MACOSX_BUNDLE` 目标属性将被设置为 `ON`。
- `EXCLUDE_FROM_ALL` 参数
若指定了 `EXCLUDE_FROM_ALL`，则 `EXCLUDE_FROM_ALL` 属性将被设置为 `TRUE`。若 `EXCLUDE_FROM_ALL` 属性被设置为 `TRUE`，则将从包含目录及其祖先目录的 `all` 目标中排除该目标(读者可通过查看生成的 `makefile` 文件中的 `all` 目标以理解该属性)，此时，`make` 工具不会构建目标，若该属性的值为 `FALSE`(默认)则目标不会被排除。说简单一点就是，若 `EXCLUDE_FROM_ALL` 属性被设置为 `TRUE`，则 `make` 工具不会构建目标，若为 `FALSE`(默认)，则会构建目标。但是，即使将 `EXCLUDE_FROM_ALL` 设置为 `TRUE`，目标仍可能会在 `install(TARGETS)` 命令中列出。更详细的内容可参阅 `add_subdirectory()` 命令

5.3.2 导入可执行目标

其语法为:

```
add_executable(<name> IMPORTED [GLOBAL])
```

该命令用于导入项目外的目标(或可执行文件)，这种文件被称为导入可执行文件。该命令可以方便的引用从 `add_custom_command()` 命令生成的文件。使用该命令后，目标属性 `IMPORTED` 的值为 `TRUE`，其

作用域为创建它的目录及子目录，但可以设置为 GLOBAL 作用域。导入可执行目标的详细信息需要通过设置以 IMPORTED_开头的属性来设置，其中最重要的是 IMPORTED_LOCATION 及 IMPORTED_LOCATION_<CONFIG>。导入可执行目标的原理与导入库目标类似，示例从略。

5.3.3 可执行目标的别名

其语法为：

```
add_executable(<name> ALIAS <target>)
```

- 1、该命令表示为目标 `target` 取一个别名 `name`，别名不能用于以下情形：
 - 不能作为 `make` 的目标出现在生成的构建系统中
 - 不能用于修改目标 `target` 的属性，即，不能作为 `set_property()`、`set_target_properties()`、`target_link_libraries()` 等命令的操作数，
 - 不能用于安装或导出。
- 2、别名可用于以下情形：
 - 可以使用别名来读取自定义(custom)命令和自定义目标的可执行文件的属性。
 - 可以使用 `if(TARGET)` 来测试别名是否存在。
- 3、3.11 及以上版本可将别名用于 GLOBAL 导入目标。在 3.18 及以上版本支持非 GLOBAL 导入目标，此时的别名的作用域是创建它的目录和子目录，目标属性 `ALIAS_GLOBAL` 可用于检查别名是否是全局的。

第 6 章 使用要求及编译参数

6.1 使用要求的基本概念

- 1、使用要求(Usage Requirements)是指的以 INTERFACE 开始以及对应的非 INTERFACE 目标属性。目前所有的使用要求都是目标属性，不过不是所有的目标属性都是使用要求，但使用要求是目标属性。
- 2、使用要求具有以下特点：
 - 使用要求是以 INTERFACE_开头的目标属性，并且通常具有与之对应的去掉 INTERFACE_之后的非 INTERFACE 目标属性。但是并不是所有的以 INTERFACE_开头的目标属性都有与之对应的去掉 INTERFACE_后的目标属性，比如 INTERFACE_SYSTEM_INCLUDE_DIRECTORIES 目标属性就没有对应的 SYSTEM_INCLUDE_DIRECTORIES 目标属性
 - CMake 使用 INTERFACE_*目标属性和对应的非 INTERFACE 目标属性来控制怎样传递(或传播)使用要求。使用要求的传递详见后文。
 - 使用要求通常有专门的命令对其进行设置，这些命令通常以 target_开头，且这些命令含有 PUBLIC、PRIVATE、INTERFACE 三个参数，这三个参数可用于控制使用要求以怎样的方式进行传递(类似于 C++的继承，但有不同)，CMake 将其称为使用要求的作用范围(或称为作用域)。
- 3、比如：INTERFACE_COMPILE_OPTIONS 目标属性是一个使用要求，有一个与之对应的 COMPILE_OPTIONS 目标属性。专门用来设置该属性的命令为 target_compile_options()，该命令含有 PUBLIC、PRIVATE、INTERFACE 三个参数，这三个参数可以控制使用要求以怎样的方式进行传递。
- 4、由于使用要求是属性，因此，可以使用 set_property()和 get_property()命令设置和读取所有的使用要求，同理，使用要求还是目标属性，因此还可使用的 set_target_properties()命令设置其值，使用相应的 get 命令读取其值。但是使用这些属性命令与专门用来设置使用要求的命令不同，他们没有控制传递使用要求方式的 PUBLIC、PRIVATE、INTERFACE 三个参数，只能对每个属性进行单独设置，比较麻烦。
- 5、表 6.1 是本章将讲解的几个使用要求，也是常见的编译参数，以 VC++为例，是指的 cl.exe 的参数。

表 6.1 常见的编译命令参数(使用要求)

使用要求(目标属性)	设置使用要求的命令	简要说明
INCLUDE_DIRECTORIES INTERFACE_INCLUDE_DIRECTORIES	target_include_directories()	指定头文件所在目录 向编译命令添加-I 或-issystem 参数
COMPILE_DEFINITIONS INTERFACE_COMPILE_DEFINITIONS	target_complid_definitions()	使用编译命令定义一个宏 向编译命令添加-D 或/D 参数
COMPILE_OPTIONS INTERFACE_COMPILE_OPTIONS	target_compile_options()	向编译命令添加所有参数
COMPILE_FEATURES INTERFACE_COMPILE_FEATURES	target_compile_features()	指定编译器所用的语言标准或单独的语言标准特性，有可能会向编译命令添加-std 参数

6.2 设置编译参数(COMPILE_OPTIONS 系列属性)

6.2.1 总览

- 1、编译参数是指的编译命令的参数，而不是链接命令的参数，如 `cl.exe` 命令的参数。注意：本文使用 VC++ 编译器，所以，设置的编译参数适用于 VC++ 的 `cl.exe` 命令。
- 2、可以使用表 6.2 中的属性或变量以及对应的命令向目标中的所有语言指定编译目标时的任意参数。对于添加预处理器定义和包含目录的参数，建议使用更具体的 `target_compile_definitions()` 和 `target_include_directories()` 命令。
- 3、变量、目标属性、目录属性的区别在于其作用范围不同，设置变量可作用于该变量之后的所有目标，目标属性仅对单个目标有效，目录属性则对该目录中的所有目标有效。
- 4、CMake 将编译参数分为多个部分，这多个部分分别使用以下变量或属性来设置
 - `CMAKE_<LANG>_FLAGS` 变量。默认值为 `/DWIN32 /D_WINDOWS /EHsc`
 - `CMAKE_<LANG>_FLAGS_<CONFIG>` 变量。默认值为 `/Ob0 /Od /RTC1`
 - `CMAKE_CXX_FLAGS_RELEASE` 变量。默认值为 `/O2 /Ob2 /DNDEBUG`
 - `COMPILE_OPTIONS` 系列属性。无默认值。因此，以上 3 个部分虽然都是设置的编译参数，但他们是相互独立不相关的。
- 5、重复的参数会被合并，比如，若同时指定参数 `-X A -X B`，将被合并为 `-X A B`。
- 6、怎样设置和获取属性值在前文已讲解过。本小节重点讲解间接设置这些属性的 `add_compile_options()` 和 `target_compile_options()` 命令。有关使用要求，即 `INTERFACE_COMPILE_OPTIONS` 属性，的详细讲解详见下一章。

表 6.2 设置编译参数的属性或变量

类别	变量或属性名	说明
目标属性	<code>COMPILE_OPTIONS</code> <code>INTERFACE_COMPILE_OPTIONS</code>	指定编译参数(以分号分隔)。可由 <code>target_compile_options()</code> 命令设置其属性值。其中 <code>COMPILE_OPTIONS</code> 属性在创建目标时由目录属性 <code>COMPILE_OPTIONS</code> 初始化
目录属性	<code>COMPILE_OPTIONS</code>	指定编译参数(以分号分隔)。可由 <code>add_compile_options()</code> 命令设置其属性值。使用目录属性的好处是可同时为该目录中的多个目标指定相同的值。
源文件属性	<code>COMPILE_OPTIONS</code>	指定编译参数(以分号分隔)。
变量	<code>CMAKE_<LANG>_FLAGS</code> <code>CMAKE_<LANG>_FLAGS_<CONFIG></code>	以下是使用 VC++ 编译器、minGW 生成器时部分变量所对应的默认参数值： <ul style="list-style-type: none">● <code>CMAKE_CXX_FLAGS</code> = <code>/DWIN32 /D_WINDOWS /EHsc</code>● <code>CMAKE_CXX_FLAGS_DEBUG</code> = <code>/Ob0 /Od /RTC1</code>● <code>CMAKE_CXX_FLAGS_RELEASE</code> = <code>/O2 /Ob2 /DNDEBUG</code>

6.2.2 target_compile_options()和 add_compile_options()命令

1、target_compile_options()命令的语法如下：

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

- 1)、该命令用于向目标属性 COMPILE_OPTIONS 和 INTERFACE_COMPILE_OPTIONS 追加值(注意追加值与修改值的区别)。
- 2)、目标 target 必须是由 add_executable()或 add_library()等命令创建的，且不能是别名目标。
- 3)、若指定了 BEFORE 则将内容添加到属性前而不是追加到末尾(3.16 及以下版本有可能会忽略 BEFORE 关键字)。
- 4)、PRIVATE 和 PUBLIC 选项将填充 COMPILE_OPTIONS 目标属性，INTERFACE 和 PUBLIC 选项将填充 INTERFACE_COMPILE_OPTIONS 目标属性，PRIVATE、PUBLIC、PUBLIC 三个选项还会影响属性的传播，详见后文。
- 5)、3.11 及以上版本将允许在导入库目标上设置 INTERFACE 选项。

2、add_compile_options()命令的语法如下：

```
add_compile_options(<option> ...)
```

该命令用于向 COMPILE_OPTIONS 目录属性追加值(注意追加值与修改值的区别)。这些值在当前目录及以下目录编译目标时使用。

示例 6.1：设置编译参数(cl.exe 命令)

①、源文件准备

在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp(主源文件)
#include<iostream>
void f();
int main(){
    std::cout<<"====a.cpp===="<<std::endl;
    std::cout<<"CC="<<CC<<std::endl;
    //std::cout<<"DD="<<DD<<std::endl;
    std::cout<<"EE="<<EE<<std::endl;
    std::cout<<"FF="<<FF<<std::endl;
    std::cout<<"GG="<<GG<<std::endl;
    std::cout<<"HH="<<HH<<std::endl;
    f();
    return 0;}
```

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
#include <iostream>
void f(){
    std::cout<<"====b.cpp===="<<std::endl;
```

```
std::cout<<"CC="<<CC<<std::endl;
std::cout<<"DD="<<DD<<std::endl;
std::cout<<"EE="<<EE<<std::endl;
std::cout<<"FF="<<FF<<std::endl;
std::cout<<"GG="<<GG<<std::endl;
//std::cout<<"HH="<<HH<<std::endl;    }
```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

#g:/qt1/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
set_property(DIRECTORY g:/qt1 PROPERTY COMPILE_OPTIONS -DCC=1)    #❶目录属性
add_library(jj STATIC b.cpp )
set_property(TARGET jj APPEND PROPERTY COMPILE_OPTIONS -DDD=2)    #❷追加参数

#set(CMAKE_CXX_FLAGS -DEE1=21 -DEE2=22 )    #❸错误参数
set(CMAKE_CXX_FLAGS "-DEE=3 -DFF=4" )    #❹更新原有参数
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} -DGG=5" )    #❺向原有参数追加值
add_executable(nn a.cpp)
target_link_libraries(nn jj)
target_compile_options(nn PUBLIC -DHH=6)    #❻追加编译参数-DDD=2

#以下所有代码都用于测试结果
get_property(AA1 TARGET nn PROPERTY COMPILE_OPTIONS )
get_property(AA2 TARGET nn PROPERTY INTERFACE_COMPILE_OPTIONS )
get_property(AA3 TARGET jj PROPERTY COMPILE_OPTIONS )
get_property(AA4 TARGET jj PROPERTY INTERFACE_COMPILE_OPTIONS )
get_property(AA5 DIRECTORY g:/qt1 PROPERTY COMPILE_OPTIONS)
get_property(AA6 SOURCE b.cpp PROPERTY COMPILE_OPTIONS)
message(build=${CMAKE_BUILD_TYPE})    #输出 DEBUG。当前编译模式
message(AA1=${AA1})    #输出 AA1=-DCC=1-DHH=2
message(AA2=${AA2})    #输出 AA2=-DHH=2
message(AA3=${AA3})    #输出 AA3=-DCC=1-DDD=1
message(AA4=${AA4})    #输出 AA4=
message(AA5=${AA5})    #输出 AA5=-DCC=1
message(AA6=${AA6})    #输出 AA6=
message(CF=${CMAKE_CXX_FLAGS})    #输出 CF=-DEE=3 -DFF=4
message(CFD=${CMAKE_CXX_FLAGS_DEBUG})    #输出 CFD=/Ob0 /Od /RTC1 -DGG=5
```

③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```
cmake .
mingw32-make
nn.exe
```

以上命令的 CMake 部分输出(即执行 cmake .命令后输出的内容为):

```
build=Debug
```

```
AA1=-DCC=1-DHH=6
AA2=-DHH=6
AA3=-DCC=1-DDD=2
AA4=
AA5=-DCC=1
AA6=
CF=-DEE=3 -DFF=4
CFD=/Ob0 /Od /RTC1 -DGG=5
```

C++程序输出(即执行 nn.exe 输出的内容为):

```
====a.cpp=====
CC=1
EE=3
FF=4
GG=5
HH=6
====b.cpp=====
CC=1
DD=2
EE=3
FF=4
GG=5
```

④、分析 CMake 代码

- 1)、本示例在❸处设置的参数是错误的，因为其参数将被保存为“-DEE1=21;-DEE2=22”，参数间含有分号且无空格，该参数会以此形式直接传递给编译器的 cl.exe 命令，对于 cl.exe 来讲这是一个不合法的参数。正确的设置多个参数的方法是使用❹和❺处的方法，使用双引号括起来，并在参数间加上空格，其中第❺处展示了怎样向原有参数追加值的方法。
- 2)、本示例在❶处设置了目录属性，该属性的值将对该目录范围内此命令之后的所有目标生效，因此，将对目标 jj 和 nn 产生作用，在❷处向 jj 追加 COMPILE_OPTIONS 属性值，在❸处向 nn 追加 COMPILE_OPTIONS 属性值，所以最终 jj 的 COMPILE_OPTIONS 属性值为

```
COMPILE_OPTIONS = -DCC=1-DDD=2
```

nn 的 COMPILE_OPTIONS 属性值为

```
COMPILE_OPTIONS =-DCC=1-DHH=6
```

- 3)、本示例在❹处和❺处设置的编译参数与 COMPILE_OPTIONS 属性设置的参数，三者之间是相互独立互不影响的，所以，最终为 jj 设置的编译参数有(实际的参数顺序可能不相同)

```
-DCC=1 -DDD=2 -DEE=3 -DFF=4 -DGG=5
```

最终为 nn 设置的编译参数有(实际的参数顺序可能不相同)

```
-DCC=1 -DDD=2 -DEE=3 -DFF=4 -DHH=6
```

⑤、CMake 文件查看及分析

查看 G:\qt1\CMakeFiles\jj.dir 目录下的文件 flags.make，可以看到如下内容

```
CXX_FLAGS = -DEE=3 -DFF=4 /Ob0 /Od /RTC1 -DGG=5 -MDd -Zi -DCC=1 -DDD=2
```

查看 G:\qt1\CMakeFiles\nn.dir 目录下的文件 flags.make，可以看到如下内容

```
CXX_FLAGS = -DEE=3 -DFF=4 /Ob0 /Od /RTC1 -DGG=5 -MDd -Zi -DCC=1 -DHH=6
```

可见，设置的参数已成功地被添加到了编译命令 cl.exe 的参数中。

然后查看本示例设置的参数是否有添加到其他地方，如下所示：

- 查看 CMakeFiles\nn.dir\link.txt 可知，本示例设置的参数未添加到目标 nn 的 link 命令，
- 查看 CMakeFiles\jj.dir\link.txt 可知，本示例设置的参数未添加到目标 jj 的 lib 命令
- 查看 CMakeFiles\nn.dir 和 CMakeFiles\jj.dir 目录下的 build 文件，分别找到以下 makefile 目标
 - ◆ CMakeFiles\jj.dir\b.cpp.obj
 - ◆ jj.lib
 - ◆ CMakeFiles\nn.dir\a.cpp.obj:
 - ◆ nn.exe:

可见，各目标使用的 link、lib 命令均未在这些命令中增加本示例指定的参数，其中的 cl 命令除了增加了在 flags.make 文件中指定的参数外，也没有再额外增加参数。

示例 6.2：理解源文件属性

①、源文件准备

在目录 g:\qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp(主源文件)
#include<iostream>
void f();
void g();
int main(){      f();    g();    return 0;}
```

在目录 g:\qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
#include <iostream>
void f(){ std::cout<<"DDf"<<std::endl;    }
```

在目录 g:\qt1\xx 中的 b.cpp 中编写如下代码

```
//g:/qt1/xx/b.cpp
#include <iostream>
void g(){ std::cout<<"DDg"<<std::endl;    }
```

②、在目录 g:\qt1 中的 CMakeLists.txt 中编写如下代码，

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXX)
add_subdirectory(xx)          #添加子目录
set_property(DIRECTORY g:/qt1 PROPERTY COMPILE_OPTIONS -DBB)      #❶
add_library(jj STATIC b.cpp xx/b.cpp)
add_executable(nn a.cpp b.cpp)
target_link_libraries(nn jj kk)
set_property(TARGET jj APPEND PROPERTY COMPILE_OPTIONS -DCC)      #❷
set_property(SOURCE b.cpp APPEND PROPERTY COMPILE_OPTIONS -DDD)   #❸
```

```

set_property(SOURCE b.cpp TARGET_DIRECTORY jj
              APPEND PROPERTY COMPILE_OPTIONS -DEE)          # ④
set_property(SOURCE xx/b.cpp
              APPEND PROPERTY COMPILE_OPTIONS -DFF)          # ⑤
set_property(SOURCE xx/b.cpp DIRECTORY g:/qt1/xx
              APPEND PROPERTY COMPILE_OPTIONS -DGG)          # ⑥
set_property(SOURCE b.cpp DIRECTORY g:/qt1
              APPEND PROPERTY COMPILE_OPTIONS -DHH)          # ⑦
set_property(SOURCE xx/b.cpp DIRECTORY g:/qt1
              APPEND PROPERTY COMPILE_OPTIONS -Dii)          # ⑧

```

在 g:/qt1/xx 目录下新建一个空的 CMakeLists.txt 文件(该文件主要是为了保证 CMake 能正常运行)

#g:/qt1/xx/CMakeLists.txt

```
add_library(kk STATIC b.cpp g:/qt1/b.cpp )
```

③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```

cmake .
mingw32-make
nn.exe          #分别依次输出 DDf、DDg

```

④、分析 CMake 代码

本示例对同一文件进行了多次编译，在实际程序中这是没有必要的，本示例的主要作用在于演示文件属性的原理。

本示例在①处设置了一个目录属性，该属性将为此命令之后的所有目标添加-DBB 编译参数，因此，该编译参数将在编译以下文件时有效。注意：对目标 kk 无效，因为目标 kk 位于①的前面

-DBB:

```

jj::g:/qt1/b.cpp
jj::g:/qt1/xx/b.cpp
nn::g:/qt1/a.cpp
nn::g:/qt1/b.cpp

```

其中 jj::g:/qt1/b.cpp 表示目标 jj 对应的 g:/qt1/b.cpp 文件，其余类似。

在②处为目标 jj 追加了-DCC 编译参数，因此，该编译参数将在编译以下文件时有效：

-DCC:

```

jj::g:/qt1/b.cpp
jj::g:/qt1/xx/b.cpp

```

在③处为 g:/qt1/CMakeLists.txt 文件中添加的源文件 b.cpp 追加了-DDD 编译参数，因此，该编译参数将在编译以下文件时有效：

-DDD:

```

jj::g:/qt1/b.cpp
nn::g:/qt1/b.cpp

```

在④处为目标 jj 所在目录下 CMakeLists.txt 文件中添加的源文件 b.cpp 追加了-DEE 编译参数，因此，该编译参数将在编译以下文件时有效：

-DEE:

```
jj::g:/qt1/b.cpp
```

```
nn::g:/qt1/b.cpp
```

在❸处为目录 g:/qt1 下的 CMakeLists.txt 文件中添加的源文件 xx/b.cpp 追加了-DFF 编译参数，因此，该编译参数将在编译以下文件时有效：

```
-DFF:
```

```
jj::g:/qt1/xx/b.cpp
```

在❹处为目录 g:/qt1/xx 下的 CMakeLists.txt 文件中添加的源文件 xx/b.cpp 添加了-DGG 编译参数，因此，该编译参数将在编译以下文件时有效：

```
-DGG:
```

```
kk::g:/qt1/xx/b.cpp
```

在❺处为目录 g:/qt1 下的 CMakeLists.txt 文件中添加的源文件 b.cpp 追加了-DHH 编译参数，因此，该编译参数将在编译以下文件时有效：

```
-DHH:
```

```
jj::g:/qt1/b.cpp
```

```
nn::g:/qt1/b.cpp
```

在❻处为目录 g:/qt1 下的 CMakeLists.txt 文件中添加的源文件 xx/b.cpp 追加了-Dii 编译参数，因此，该编译参数将在编译以下文件时有效：

```
-Dii:
```

```
jj::g:/qt1/xx/b.cpp
```

最终为各源文件添加的编译参数为：

```
jj::g:/qt1/b.cpp = -DBB -DCC -DDD -DEE -DHH
```

```
jj::g:/qt1/xx/b.cpp = -DBB -DCC -DFF -Dii
```

```
nn::g:/qt1/a.cpp = -DBB
```

```
nn::g:/qt1/b.cpp = -DBB -DDD -DEE -DHH
```

```
kk::g:/qt1/b.cpp = 无
```

```
kk::g:/qt1/xx/b.cpp = -DGG
```

其中-DDD、-DEE、-DFF、-DGG、-DHH、-Dii 通过源文件属性添加。

⑤、CMake 文件查看及分析

1)、查看 G:\qt1\CMakeFiles\jj.dir 目录下的文件 flags.make，可以找到如下内容

```
CXX_FLAGS = /DWIN32 /D_WINDOWS /EHsc /Ob0 /Od /RTC1 -MDd -Zi -DBB -DCC
```

```
# Custom options: CMakeFiles\jj.dir\b.cpp.obj_OPTIONS = -DDD;-DEE;-DHH
```

```
# Custom options: CMakeFiles\jj.dir\xx\b.cpp.obj_OPTIONS = -DFF;-Dii
```

后面两行注释表示当编译该两行的对象文件时需要使用的源文件参数。以上的 CXX_FLAGS 变量(这是 makefile 变量)所指定的编译参数将对 build.make 文件中的所有源文件有效，即在编译 b.cpp 和 xx/b.cpp 时均有效。

2)、查看 G:\qt1\CMakeFiles\jj.dir 目录下的文件 build.make，可以找到如下内容

```
CMakeFiles\jj.dir\b.cpp.obj: b.cpp
```

```
cl.exe $(CXX_DEFINES) $(CXX_INCLUDES) $(CXX_FLAGS) -DDD -DEE -DHH ..... -c G:\qt1\b.cpp
```

这是调用 cl.exe 编译 g:/qt1/b.cpp 的代码，其中\$(CXX_DEFINES) \$(CXX_INCLUDES)

\$(CXX_FLAGS)是以 makefile 变量的方式指定编译参数，本示例的编译参数-DBB 和-DCC 通过

`$(CXX_FLAGS)`变量指定给 `cl.exe` 命令，在这里我们还看到了为该条命令单独添加的`-DDD -DEE -DHH` 三个参数，这三个参数便是通过 `CMakeLists.txt` 文件以不同方式的源文件属性添加的，所以，编译 `jj` 目标的 `g:/qt1/b.cpp` 文件时的编译参数为(不含默认参数)：

`-DBB -DCC -DDD -DEE -DHH`

同理，可在该文件中继续找到以下内容

`CMakeFiles/jj.dir/xx/b.cpp.obj: xx/b.cpp`

`cl.exe $(CXX_DEFINES) $(CXX_INCLUDES) $(CXX_FLAGS) -DFF -Dii -c G:/qt1/xx/b.cpp`

所以，编译 `jj` 目标的 `g:/qt1/xx/b.cpp` 文件时的编译参数为(不含默认参数)：

`-DBB -DCC -DFF -Dii`

3)、以同样的方式查看 `G:/qt1/CMakeFiles/nn.dir` 目录下的文件 `flags.make` 和 `build.make`，可得到

编译 `nn` 目标的 `g:/qt1/a.cpp` 文件时的编译参数为(不含默认参数)：

`-DBB`

编译 `nn` 目标的 `g:/qt1/b.cpp` 文件时的编译参数为(不含默认参数)：

`-DBB -DDD -DEE -DHH`

4)、以同样的方式查看 `G:/qt1/xx/CMakeFiles/kk.dir` 目录下的文件 `flags.make` 和 `build.make`，可得到

编译 `kk` 目标的 `g:/qt1/b.cpp` 文件时的编译参数为(不含默认参数)：

无

编译 `kk` 目标的 `g:/qt1/xx/b.cpp` 文件时的编译参数为(不含默认参数)：

`-DGG`

6.3 设置-D 编译参数(COMPILE_DEFINITIONS 系列属性)

6.3.1 总览

1、前一小节讲解的方法也可以添加-D 编译参数，本小节讲解的内容是专门的更具体的添加-D 编译参数的方法。

2、可以使用以下属性向编译命令添加-D 或/D 参数

- 目标属性：`COMPILE_DEFINITIONS` 和 `INTERFACE_COMPILE_DEFINITIONS`。
命令 `target_compile_definitions()`可设置这两个属性的值
- 目录属性：`COMPILE_DEFINITIONS`。命令 `add_compile_definitions()`可设置这个属性的值。
- 源文件属性：`COMPILE_DEFINITIONS`

3.26 及以上版本，指定的以上属性值的-D 前缀将被删除。这些属性对某些特殊的需要转义的字符可

能不能正确的支持。

6.3.2 target_compile_definitions()和 add_compile_definitions()命令

1、target_compile_definitions()命令的语法如下：

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

该命令用于向目标属性 COMPILE_DEFINITIONS 和 INTERFACE_COMPILE_DEFINITIONS 追加值，即设置编译<target>时的-D 或/D 参数。目标<target>必须是由 add_executable()或 add_library()等命令创建的，且不能是别名目标。PRIVATE 和 PUBLIC 选项将填充目标<target>的 COMPILE_DEFINITIONS 属性。PUBLIC 和 INTERFACE 选项将填充目标<target>的 INTERFACE_COMPILE_DEFINITIONS 属性。3.11 及以上版本将允许在导入目标上设置 INTERFACE 选项。在 item 中指定的前缀-D 将被删除，空的 item 将被忽略。因此，以下语句是等价的

```
target_compile_definitions(xx PUBLIC EE)
target_compile_definitions(xx PUBLIC -DEE)    # -D 被删除，等价于 EE
target_compile_definitions(xx PUBLIC "" EE)    # 空项被忽略，等价于 EE
target_compile_definitions(xx PUBLIC -D EE)    # -D 被删除，此项变成空项，然后被忽略，最终仍等价于 EE
```

注意：许多编译器会将 -DEE 解释为 -DEE=1。

2、add_compile_definitions()命令的语法如下：

```
add_compile_definitions(<definition> ...)
```

该命令用于向当前 CMakeLists 文件的目录属性 COMPILE_DEFINITIONS 追加值。3.26 及以上版本，指定的-D 前缀将被删除。

示例 6.3：使用 add_compile_definitions()和 target_compile_definitions()命令

①、源文件准备

在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp(主源文件)
#include<iostream>
void f();
int main(){    std::cout<<"BB"<<"CC"<<"DD"<<std::endl;
    f();return 0;}
```

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
#include <iostream>
void f(){    std::cout<<"f()="<<"DD"<<std::endl;    }
```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码，

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXX)
```

```

add_library(kk STATIC b.cpp)
add_executable(nn a.cpp)
target_link_libraries(nn kk)
add_compile_definitions(DD=2)  #该命令无需放于 add_executable()或 add_library()之前
set_property(TARGET nn PROPERTY COMPILE_DEFINITIONS -DCC=3)
target_compile_definitions(nn PUBLIC BB=4)

```

③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```

cmake .
mingw32-make
nn.exe          #依次输出 432 和 f()=2

```

④、在 G:\qt1\CMakeFiles\kk.dir 目录下的 flasg.make 文件中，可以找到以下内容

```
CXX_DEFINES = -DDD=2
```

在 G:\qt1\CMakeFiles\kk.dir 目录下的 flasg.make 文件中，可以找到以下内容

```
CXX_DEFINES = -DBB=4 -DCC=3 -DDD=2
```

可见，设置的参数-DDD=2 同时被添加到了编译目标 nn 和 kk 的 cl 命令中

注意：使用

```
get_property(AA3 TARGET kk PROPERTY COMPILE_DEFINITIONS)
```

不能获取到由目录属性 COMPILE_DEFINITIONS 传递给目标属性 COMPILE_DEFINITIONS 的值-DDD，但-DDD 参数已被添加到 cl 命令中了。

6.4 设置编译特性(COMPILE_FEATURES 系统属性)

6.4.1 总览

1、编译特性

编译特性说简单一点就是编译器所能支持的语言标准定义的特性，设置编译特性就是设置编译器是否应支持这些语言标准定义的某些特性。比如，编译器是否支持 long long 类型，是否支持内联名称空间，是否支持 C++11(这意味着支持 C++11 的所有特性)等，其中的 long long 类型、内联名称空间、C++11 标准等特性被称为编译特性。

2、COMPILE_FEATURES 目标属性以及与其相关的变量和属性

可以使用表 6.3 所示的属性或/和变量以及对应的命令来设置编译器的特定功能(即编译特性)，这些设置可能会使 CMake 向编译器添加指定语言标准的-std 或/std 参数。

3、编译参数-std 或/std

若指定的编译特性需要额外的添加编译参数，比如-std 参数，则会自动添加该参数。如果编译器的默认标准级别至少是设置的特性，则 CMake 可能会忽略-std 参数。如果编译器的默认扩展模式与 <LANG>_EXTENSIONS 目标属性不匹配，或者设置了<LANG>_STANDARD 目标属性，则仍然有可能会添加-std 参数。

表 6.3 设置编译特性

类别	属性或变量名	说明
----	--------	----

目标属性	COMPILE_FEATURES INTERFACE_COMPILE_FEATURES	<code>target_compile_features()</code> 命令可填充左侧属性的值。这些属性用于指定编译器的编译特性，其值必须是本表下一行列出的三个变量所列特性的子集，若指定的值在以下变量中不存在，则会产生错误。若要设置编译特性，应在左侧的目标属性上操作。
	<LANG>_STANDARD	此属性指定使用的语言标准，对于 CXX_STANDARD 可取值有 98、11、14、17、20、23、26。详细规则见正文内容。
变量	CMAKE_C_COMPILE_FEATURES CMAKE_CXX_COMPILE_FEATURES CMAKE_CUDA_COMPILE_FEATURES	存储的是可用的 C/C++/CUDA 编译器的特性列表，这些列表分别是对应的本表下一行对应的全局属性列出的特性的子集。不建议修改左侧变量的值。
	CMAKE_<LANG>_STANDARD	该变量的值用于初始化<LANG>_STANDARD 目标属性，其余与<LANG>_STANDARD 相同，使用该变量可同时为多个目标设置相同的语言标准
全局属性	CMAKE_C_KNOWN_FEATURES CMAKE_CXX_KNOWN_FEATURES CMAKE_CUDA_KNOWN_FEATURES	<p>包含 CMake 已知的 C/C++/CUDA 编译器的所有编译特性，无论编译器是否支持这些特性。若这些特性在对应的编译器中可用，则在本表上一行中对应的三个变量中列出其值。也就是说，左侧的属性拥有一系列的值，每一个值都描述了编译器的某个或某些特性，比如，是否支持 C++11(这意味着支持 C++11 的所有特性)，是否支持 long long 类型等，不建议修改这些属性的值。以 CMAKE_CXX_KNOWN_FEATURES 为例，该属性拥有描述编译器是否支持 C++语言标准所有特性的值，如</p> <ul style="list-style-type: none"> ● <code>cxx_std_98</code>: 表示编译器至少需要支持 C++98 标准 ● <code>cxx_std_11</code>: 表示编译器至少需要支持 C++11 标准 ● <code>cxx_std_14</code>: 表示编译器至少需要支持 C++14 标准 ● <code>cxx_std_17</code>: 表示编译器至少需要支持 C++17 标准 ● <code>cxx_std_20</code>: 表示编译器至少需要支持 C++20 标准 <p>该属性也拥有描述是否支持语言标准独特性的值，单独的特性通常是比较特别的较少使用的特性，所以，提供的都是低级版本的单独特性，CMake 未提供比 C++17 或更高版本的单独特性。比如</p> <ul style="list-style-type: none"> ● <code>cxx_template_template_parameters</code>: 表示模板模板参数(C++98) ● <code>cxx_inline_namespaces</code>: 表示内联名称空间(C++11) ● <code>cxx_long_long_type</code>: 表示 long long 类型(C++11) ● <code>cxx_variable_templates</code>: 表示变量模板(C++14) ● <code>cxx_binary_literals</code>: 表示二进制字面值(C++14) <p>属性的所有取值详见 CMake 帮助文档。</p>

6.4.2 target_compile_features()命令

其语法如下：

```
target_compile_features(<target> <PRIVATE|PUBLIC|INTERFACE> <feature> [...])
```

为目标<target>追加所需的编译特性<feature>，各参数意义如下：

- 指定的目标<target>必须是由一个 `add_execute()`或 `add_library()`命令创建的并且不能是别名目标。
- 若特性<feature>未在 CMAKE_C_COMPILE_FEATURES、CMAKE_CUDA_COMPILE_FEATURES

或 CMAKE_CXX_COMPILE_FEATURES 变量中列出，将报告一个错误。

- PRIVATE 和 PUBLIC 选项将填充 COMPILE_FEATURES 目标属性。PUBLIC 和 INTERFACE 选项将填充 INTERFACE_COMPILE_FEATURES 目标属性。

3.11 及以上版本允许在导入目标上设置 INTERFACE 项。

6.4.3 <LANG>_STANDARD 目标属性以及与其相关的属性和变量

- 1、<LANG>_STANDARD 目标属性用于指定构建目标时所要求的语言标准的版本，比如，对于指定 C++ 语言标准的 CXX_STANDARD 目标属性可取值有 98、11、14、17、20、23、26，分别对应于 C++98、C++11、C++14、C++17、C++20、C++23、C++26 标准。如果指定的语言标准高于编译器实际支持的最新标准，则将回退到支持的最新标准。
- 2、若在创建目标时设置了 CMAKE_<LANG>_STANDARD 变量，则目标属性<LANG>_STANDARD 由该变量的值初始化。
- 3、可将目标属性<LANG>_STANDARD_REQUIRED 的值设置为 TRUE 来控制<LANG>_STANDARD 目标属性指定的标准必须被满足，若不能满足要求，将产生错误，若该属性的值为 FALSE 或未设置时，由<LANG>_STANDARD 指定的标准将被回退到当前支持的最新标准。
- 4、若在创建目标时设置了 CMAKE_<LANG>_STANDARD_REQUIRED 变量，则目标属性<LANG>_STANDARD_REQUIRED 由该变量的值初始化。
- 5、使用变量和使用目标属性的主要区别在于，使用变量可以为多个目标指定相同的参数，使用目标属性只能每次为指定的单个目标设置参数。

6.4.4 <LANG>_EXTENSIONS 目标属性以及与其相关的属性和变量

- 1、<LANG>_EXTENSIONS 目标属性可用于设置是否为目标启用特定于编译器的扩展。
- 2、如果在创建目标时设置了 CMAKE_<LANG>_EXTENSIONS 变量，则将其值作为<LANG>_EXTENSIONS 目标属性的默认值；若变量 CMAKE_<LANG>_EXTENSIONS 的值也未设置，则将变量 CMAKE_<LANG>_EXTENSIONS_DEFAULT 的值作为<LANG>_EXTENSIONS 目标属性的默认值。变量 CMAKE_<LANG>_EXTENSIONS_DEFAULT 是编译器的默认扩展模式，其值是只读的，若修改其值将是未定义的行为。

表 2-XXX 对目标属性<LANG>_STANDARD 和<LANG>_EXTENSIONS 进行了总结

表 2-XXX 部分目标属性的总结

目标属性	初始化左侧属性值的变量	简介
<LANG>_STANDARD	CMAKE_<LANG>_STANDARD	指定语言标准的版本
<LANG>_STANDARD_REQUIRED	CMAKE_<LANG>_STANDARD_REQUIRED	使指定语言标准的版本必须被满足
<LANG>_EXTENSIONS	CMAKE_<LANG>_EXTENSIONS	是否启用特定于编译器的扩展
	CMAKE_<LANG>_EXTENSIONS_DEFAULT	

示例 6.4: COMPILE_FEATURES 目标属性

说明：本示例适合于为单独的目标指定各自的编译特性

```
cmake_minimum_required(VERSION 3.27)
project(hh)
add_library(jj STATIC b.cpp c.cpp)
```

```

add_executable(nn a.cpp)
target_link_libraries(nn jj)
set_property(TARGET nn PROPERTY COMPILE_FEATURES cxx_std_17)
#向属性 COMPILE_FEATURES 追加值
target_compile_features(nn PUBLIC cxx_alias_templates )
#错误, eee 不在 CMAKE_CXX_COMPILE_FEATURES 变量的值之中
#target_compile_features(nn PUBLIC eee )
#以下代码用于测试
get_property(AAjI TARGET nn PROPERTY COMPILE_FEATURES )
get_property(BB GLOBAL PROPERTY CMAKE_CXX_KNOWN_FEATURES)
message("AAjI=${AAjI}\n")      #输出 cxx_std_17;cxx_alias_templates
message("BB=${BB}\n")          #输出全局属性 CMAKE_CXX_KNOWN_FEATURES 的值, 其值比较多
message("DD=${CMAKE_CXX_COMPILE_FEATURES}") #同上, 但其值是其子集

```

本示例使用 `set_property()` 和 `target_compile_features()` 命令为目标 `nn` 指定了两个编译特性, `cxx_std_17` 和 `cxx_alias_templates`。执行 `cmake` 命令后, 打开 `CMakeFiles\nn.dir\flags.make` 文件, 可以看到为 `cl` 增加了参数 `-std:c++17`, 注意: 参数 `-std` 需满足一定条件才会增加。

示例 6.5: 使用 CMAKE_CXX_STANDARD 变量设置 C++ 语言标准

说明: 本示例演示为多个或所有目标指定相同的语言标准

```

cmake_minimum_required(VERSION 3.27)
project(hh)
#指定 C++17, 注意, 该语句必须在需要构建的目标之前, 这是一种常用的指定语言标准版本的方式
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED TRUE)      #编译器必须满足 C++17 的标准, 否则, 将产生错误
add_library(jj STATIC b.cpp c.cpp)
add_executable(nn a.cpp)
target_link_libraries(nn jj)
get_property(AA1 TARGET nn PROPERTY CXX_STANDARD )
get_property(AA2 TARGET jj PROPERTY CXX_STANDARD )
get_property(AA3 TARGET nn PROPERTY CXX_STANDARD_REQUIRED )
get_property(AA4 TARGET jj PROPERTY CXX_STANDARD_REQUIRED )
message(AA1=${AA1})      #输出 AA1=17
message(AA2=${AA2})      #输出 AA2=17
message(AA3=${AA3})      #输出 AA3=TRUE
message(AA4=${AA4})      #输出 AA4=TRUE

```

执行 `cmake` 命令后, 打开 `CMakeFiles\nn.dir\flags.make` 和 `CMakeFiles\jj.dir\flags.make` 文件, 可以看到两个文件都为 `cl` 增加了参数 `-std:c++17`, 因为 `set(CMAKE_CXX_STANDARD 17)` 位于目标 `nn` 和 `jj` 之前, 所以会同时将变量 `CMAKE_CXX_STANDARD` 的值设置为 `nn` 和 `jj` 的两个目标属性 `CXX_STANDARD` 的默认值。本示例通过变量 `CMAKE_CXX_STANDARD` 间接设置了多个目标的属性 `CXX_STANDARD` 的值, 通过变量 `CMAKE_CXX_STANDARD_REQUIRED` 间接设置多个目标的属性 `CXX_STANDARD_REQUIRED` 的值

示例 6.6: 使用 <LANG>_STANDARD 目标属性为单独的目标设置 C++ 语言标准

```

cmake_minimum_required(VERSION 3.27)

```

```
project(hh)
add_library(jj STATIC b.cpp c.cpp)
add_executable(nn a.cpp)
target_link_libraries(nn jj)
#使用 set_property()命令，单独的为各个目标设置各自的 C++语言标准。
set_property(TARGET nn PROPERTY CXX_STANDARD 26)
#以下命令将出错，因为目前的编译器还不支持 C++26
#set_property(TARGET nn PROPERTY CXX_STANDARD_REQUIRED TRUE)
set_property(TARGET jj PROPERTY CXX_STANDARD 17)
set_property(TARGET jj PROPERTY CXX_STANDARD_REQUIRED TRUE)
```

执行 `cmake` 命令后，打开 `CMakeFiles\nn.dir\flags.make` 文件，可以看到该文件为 `cl` 增加了参数 `-std:c++latest`，打开 `CMakeFiles\jj.dir\flags.make` 文件，可以看到该文件为 `cl` 增加了参数 `-std:c++17`。

第 7 章 使用要求的传递

1、`target_link_libraries()`命令在使用要求的传递中拥有重要的作用，本章首先讲解该命令的基本使用方法，与使用要求有关的内容详见本章后续章节。

2、`target_link_libraries()`命令有以下作用：

- 接受来自依赖项的使用要求
- 链接依赖项。将依赖项所对应的文件添加到链接命令(如 `link` 命令)。

7.1 `target_link_libraries()`命令简介

`target_link_libraries()`命令的语法为

```
target_link_libraries(<target> <item>...)
target_link_libraries(<target>
    <PRIVATE|PUBLIC|INTERFACE> <item>...
    [<PRIVATE|PUBLIC|INTERFACE> <item>...]...)
```

- 1、以上命令表示在链接阶段将<item>追加到目标<target>中，即，将指定的依赖项<item>添加到链接命令中。注意：这里只是对 `target_link_libraries()`命令的一种概括性的描述，并不是详细的讲解，有关 `target_link_libraries()`命令更具体详细的内容，详见本章后续内容。
- 2、3.13 及以上版本，目标<target>可以不与 `target_link_libraries()`调用在相同的目录中创建
- 3、在 CMake 中被称为“将<target>链接到<item>”或“<item>被链接到<target>”，由于“链接到、被链接到”的说法不好理解且易产生混淆，本文尽量不使用这一说法。
- 4、各参数意义如下：

1)、<target>

<target>必须是由 `add_executable()`或 `add_library()`等命令创建的目标，并且不能是别名目标。

2)、PUBLIC、PRIVATE、INTERFACE 参数

以上参数主要用于使用要求的传递，默认为 PUBLIC。以上参数对 `LINK_LIBRARIES` 和 `INTERFACE_LINK_LIBRARIES` 目标属性的填充与其他 `target_XXX()`命令对 `INTERFACE_*`或非 `INTERFACE` 的填充有少许不同，详见本章后续内容。

3)、<item>

<item>也被称为**依赖项或依赖**，可以是以下值：

- (1) 库目标名称。即 `add_library()`命令创建的目标。
- (2) 库文件完整路径。即，真实存在的库文件完整路径，完整路径不会进行任何名称转换。
- (3) 生成器表达式。
- (4) 一个单纯的库名。

即不含后缀的名称，CMake 会自动对名称进行相应的转换，比如，`xx` 会被转换为 `xx.lib` 或-

lxx(小写 L), 其中-lxx 适用于 g++或 ld 命令, 表示搜索名为 xx.a 或 libxx.a 的文件。

(5) 链接标志(即链接参数)

以“-”开头, 但不能是“-l(小写 L)”或“-framework”, 比如

```
target_link_libraries(nn -LIBPATHPATH:g:/qt1/xx)
```

表示向 link 命令添加一个参数-LIBPATHPATH:g:/qt1/xx。

注意: 使用该命令添加的链接参数的位置可能会不正确, 可以使用 LINK_OPTIONS 目标属性或 target_link_options()命令显示添加链接参数。

(6) 关键字 debug、optimized、general, 在其后必须指定一个<item>, 指定这些关键字之后, <item>将只适用于相应的生成配置, debug 只适用于 debug 配置, optimized 关键字与其他所有配置相对应, general 关键字对应于所有配置。

(7) 含有双冒号“::”的<item>, 将被假定为 IMPROTED 库目标(导入库目标)或 ALIAS 库目标(别名库目标), 如 foo::BAR。若不存在这样的目标, 将产生错误。详见 CMP0028

(8) 总结

<item>除了能指定链接参数、库目标名称外, item 只能指定.lib、.a 等库文件, 不能是其他文件。

示例 7.1: target_link_libraries()命令的使用

①、源文件准备

在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp(主源文件)
#include<iostream>
extern int b;
extern int c;
int main(){    std::cout<<b<<c<<std::endl;    return 0;}
```

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
int b=1;
```

在目录 g:/qt1 中的 c.cpp 中编写如下代码

```
//g:/qt1/c.cpp
int c=1;
```

②、准备编译好的文件

在 CMD 中转至 g:/qt1 并输入以下命令, 以生成一个 c.obj 和 c.lib 文件

```
cl /DWIN32 /D_WINDOWS /EHsc /Ob0 /Od /RTC1 -MDd -Zi /c b.cpp
lib c.obj
```

③、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXX)
add_library(kk STATIC b.cpp)
add_executable(nn a.cpp) #❶
target_link_libraries(nn kk c.lib) #❷
```


#❷处的命令可使用以下等效语句代替。

```
#target_link_libraries(nn kk c)      #kk 是库目标名称, c.lib 是真实存在的文件
#target_link_libraries(nn kk g:/qt1/c.lib)
#target_link_libraries(nn g:/qt1/kk.lib g:/qt1/c.lib)
#target_link_libraries(nn kk.lib g:/qt1/c.lib)
#target_link_libraries(nn kk g:/qt1/c.obj)
#以下命令是错误的
#target_link_libraries(nn kk g:/qt1/c) #找不到 c.lib。完整路径不会对名称进行转换。
#target_link_libraries(nn kk c.obj)    #找不到 c.obj.lib。非完整路径的名称会进行转换
#❶和❷两处语句也可由以下命令替代。
#add_executable(nn a.cpp c.obj)         #❸
#target_link_libraries(nn kk )          #❹
```

④、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```
cmake .
mingw32-make
nn.exe          #输出 21，符合预期
```

⑤、CMake 示例代码分析

- 1)、❶和❷两处的语句可使用❸和❹分别替代，但要注意 add_executable()命令可以指定.cpp、.obj 等文件，但不能指定.lib 等库文件。
- 2)、❸和❹没有把 c.obj 或 c.lib 链接到目标 nn，而是将 c.obj 作为 nn 的源文件添加到 nn。这二者区别如下：
 - 因为 add_executable()不能指定库文件，这意味着当有多个对象文件需要链接时，需在 add_executable()命令中逐个指定，这样比较麻烦。
 - 使用❶和❷处的命令，会将 c.lib 或 c.obj 添加到 CMakeFiles\ nn.dir\link.txt 文件中，即作为链接方式添加
 - 使用❸和❹处的命令，会将 c.obj 添加到 CMakeFiles\ nn.dir\ objects1.rsp 文件中，即作为对象文件添加

7.2 传递使用要求的基本规则

7.2.1 使用要求传递的基本规则

- 1、此处仅讲解传递使用要求的一般性规则，对于包含链接使用要求以及更详细的传递规则详见后文。
- 2、使用要求是指的以 INTERFACE 开始以及对应的非 INTERFACE 目标属性。注意：有些以 INTERFACE 开头的目标属性没有对应的非 INTERFACE 属性。
- 3、使用要求传递的基本规则为：
 - 1) 传递规则

只有目标的 INTERFACE 目标属性才可以传递给另一个目标，也就是说，若目标的 INTERFACE 属性有值，则该目标才能传递该属性值，或者说，该属性值或该使用要求是可传递的，否则不

可被传递。

2) 接口规则

目标的 INTERFACE 属性(即, 使用要求)不会对目标自身起作用, 只有目标的非 INTERFACE 属性才会对目标自身起作用。也就是说, 若只有目标的 INTERFACE 属性有值, 而对应的非 INTERFACE 属性没有该值, 则该目标自己不能使用该属性值, 只能传递该属性值, 也就是说, 该目标仅仅负责中转 INTERFACE 属性的值, 所以, 这时的目标相当于是一个接口或中转站。

3) CMake 实现传递的方法

CMake 使用 `target_link_libraries()`命令从其依赖项中读取 INTERFACE 目标属性并将其值追加到该命令的目标所对应的非 INTERFACE 目标属性, 以此接收来自依赖项的使用要求。说简单一点就是, 要想目标的使用要求传递给另一个目标, 则必须使用 `target_link_libraries()`命令或与该命令相关的链接属性将二者关联起来, 否则, 只能使用该目标自己的使用要求。需要注意的是, 不能使用 `get_property()`命令读取到其他目标传递进来的使用要求的值。比如

```
target_link_libraries(a b)
```

将读取依赖项 b 的 INTERFACE 目标属性, 并将其值追加到目标 a 的对应的非 INTERFACE 目标属性中, 这样, 目标 a 便接收了来自依赖项 b 的使用要求, 若 a 不使用 `target_link_libraries()`命令或与该命令相关的链接属性, 则, a 不能接收到来自 b 的使用要求, 只能使用目标 a 自己的使用要求。注意, 使用 `get_property(TARGET a ...)`不能读取从依赖项 b 传递给目标 a 的属性值。

4) 传递形式的控制

由 `target_XXX()`系列命令的 PUBLIC、PRIVATE、INTERFACE 三个选项通过对 INTERFACE 属性和非 INTERFACE 属性的赋值来达到控制传递的方式, 其规则如下:

- ◆ PRIVATE: 只能设置非 INTERFACE 属性, 这意味着目标不能传递该属性。注意: `target_link_libraries()`命令是个例外, 详见后文。
- ◆ INTERFACE: 只能设置 INTERFACE 属性, 这意味着该属性不会对拥有该属性的目标自身产生作用, 或者说, 拥有该属性的目标自己不能使用该属性, 但可以传递该属性。即, 该目标是一个纯粹的传递使用要求的中转站或接口。
- ◆ PUBLIC: 可以同时设置 INTERFACE 属性和非 INTERFACE 属性。这意味着, 该属性即对目标产生作用, 也可以被传递。

4、以上传递规则是整个 CMake 传递的基础, 可以概括为: INTERFACE_*是一种类似于接口的属性, 其特点是, 设置该属性的目标, 自己不能使用该属性的值, 但可以传递给其他目标使用, 其他目标使用 `target_link_libraries()`命令来接收该属性值, 这个命令将该属性值读取到对应的非 INTERFACE 属性。传递的形式可以由 PUBLIC、PRIVATE、INTERFACE 三个选项控制。

5、可以进一步将传递规则理解为“拥有 INTERFACE 属性的目标可以作为中转站或接口”。

6、另外, 可以将目标属性作为值填充给变量 `CMAKE_DEBUG_TARGET_PROPERTIES` 以在调试时查看哪些目标设置了该属性, 使用方法为:

```
set(CMAKE_DEBUG_TARGET_PROPERTIES COMPILE_OPTIONS)
```

使用以上命令后, 在执行 `cmak.exe` 命令时就能看到哪些目标设置了 `COMPILE_OPTIONS` 属性。目前只支持以下属性:

```
AUTOUIC_OPTIONS
```

```

COMPILE_DEFINITIONS
COMPILE_FEATURES
COMPILE_OPTIONS
INCLUDE_DIRECTORIES
LINK_DIRECTORIES
LINK_OPTIONS
POSITION_INDEPENDENT_CODE
SOURCES

```

示例 7.2：传递使用要求的基本规则

- ①、源文件准备：自行准备两个源文件 b.cpp 和 c.cpp
- ②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

#g:/qt1/CMakeLists.txt

```

cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj STATIC b.cpp)
set_property(TARGET jj PROPERTY COMPILE_OPTIONS -DEE=3)           #该属性不可传递
set_property(TARGET jj PROPERTY INTERFACE_COMPILE_OPTIONS -DFF=4)  #❶该属性可传递
#也可使用以下两条命令代替以上两条命令
#target_compile_options(jj PRIVATE -DEE=3)
#target_compile_options(jj INTERFACE -DFF=4)

add_library(kk STATIC c.cpp)
set_property(TARGET kk PROPERTY COMPILE_OPTIONS -DGG=5)           #该属性不可传递
#也可使用以下命令代替上一条命令
#target_compile_options(kk PRIVATE -DGG=5)

#通过以下命令将 jj 的使用要求传递给 kk
target_link_libraries(kk jj)           #kk 将得到参数-DGG=5 和从 jj 传递来的-DFF=4

#以下语句用于测试
get_property(AA1 TARGET jj PROPERTY COMPILE_OPTIONS )
get_property(AA2 TARGET jj PROPERTY INTERFACE_COMPILE_OPTIONS )
get_property(AA3 TARGET kk PROPERTY COMPILE_OPTIONS )
get_property(AA4 TARGET kk PROPERTY INTERFACE_COMPILE_OPTIONS )
message(AA1=${AA1})
message(AA2=${AA2})
message(AA3=${AA3})
message(AA4=${AA4})
#显示哪些目标设置了 COMPILE_OPTIONS 目标属性
set(CMAKE_DEBUG_TARGET_PROPERTIES COMPILE_OPTIONS )

```

- ③、本示例最终使用参数-DGG=5、-DFF=4 编译 c.cpp，使用参数-DEE=3 编译 b.cpp。注意：b.cpp 并未使用❶处设置的-DFF=4 编译，因为这是一个以 INTERFACE_开始的属性，目标自己不能使用该属

性。

- ④、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```
cmake .  
mingw32-make
```

在 CMD 中输入“cmake .”后，输出如图 7.1 所示。可以查看 CMakeFiles\kk.dir 和 CMakeFiles\kk.dir 目录下的 flags.make 文件以查看通过 COMPILE_OPTIONS 属性设置的编译参数情况。

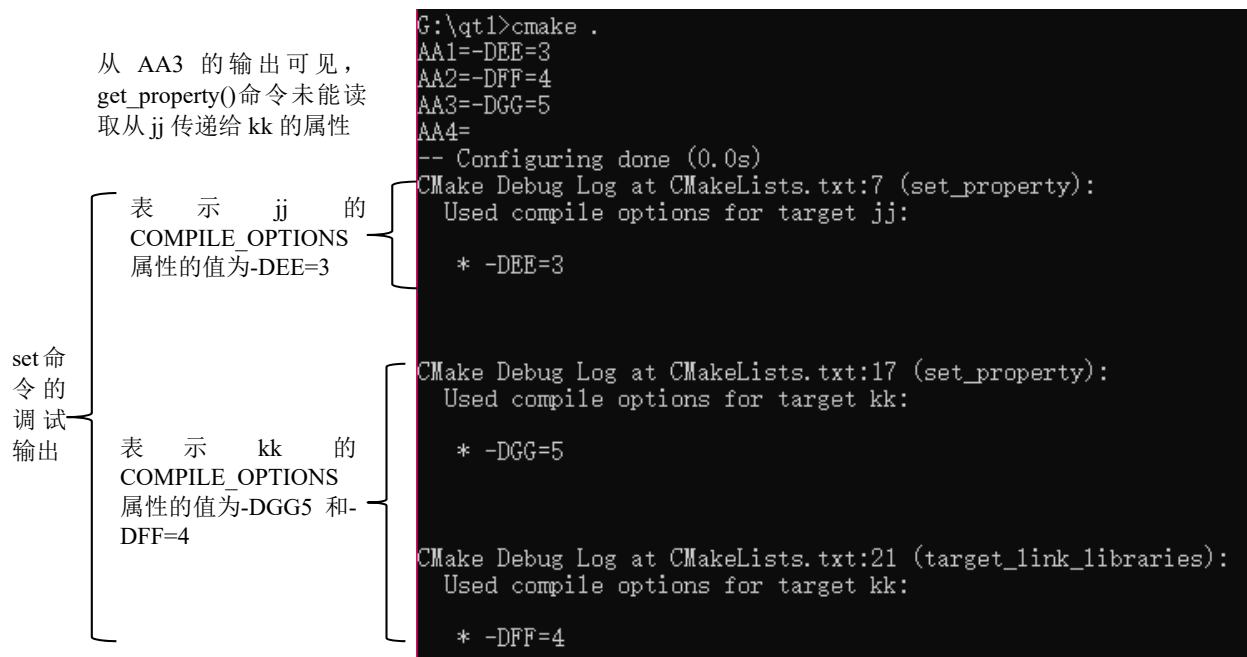


图 7.1 示例 7.2 的输出

7.2.2 本文的名称约定

- 1、为便于讲解，以及避免众多名称之间的混淆，有必要对 CMake 中使用的名称作一个基本的约定。
- 2、编译和链接过程简述。对于以下 CMake 代码：

```
add_library(jj STATIC c.cpp)  
add_library(kk STATIC b.cpp)  
target_link_libraries(kk jj)  
add_executable(nn a.cpp)  
target_link_libraries(nn kk)
```

以上语句对应的 VC++命令如下(仅列出简化后的关键参数):

#创建 jj.lib 文件	
cl /c /Foc.cpp.obj c.cpp	#编译：使用 c.cpp 创建一个名为 c.cpp.obj 的对象文件
lib /out:jj.lib c.cpp.obj	#根据对象文件创建 jj.lib。注意：lib 命令不是链接命令
#创建 kk.lib 文件	
cl /c /Fob.cpp.obj b.cpp	#编译：使用 b.cpp 创建一个名为 b.cpp.obj 的对象文件
lib /out:kk.lib b.cpp.obj	#根据对象文件创建 kk.lib。

```
#创建 nn.exe
cl /c /Foa.cpp.obj a.cpp          #编译：使用 a.cpp 创建一个名为 a.cpp.obj 的对象文件
link a.cpp.obj /out:nn.exe kk.lib jj.lib  #链接：使用 a.cpp.obj、jj.lib、kk.lib 创建 nn.exe 文件
```

3、为方便讲解，本文引入以下名称约定

■ 使用要求

指的是以 INTERFACE 开头以及对应的非 INTERFACE 目标属性。注意：有些以 INTERFACE 开头的目标属性没有对应的非 INTERFACE 属性。比如 INTERFACE_LINK_LIBRARIES_DIRECT 就没有对应的 LINK_LIBRARIES_DIRECT 属性。

■ 使用要求的值

由于使用要求的名称通常比较长，为更便于描述，本文将使用要求的值也称为使用要求。比如 INTERFACE_LINK_LIBRARIES=kk，COMPILE_OPTIONS=-DEE=3，将 kk、-DEE=3 也称为使用要求

■ 可传递和不可传递使用要求

以 INTERFACE 开头的使用要求被称为可传递使用要求，因为这些使用要求可传递给其他目标。同理，不以 INTERFACE 开头的使用要求，被称为不可传递使用要求。

■ 链接使用要求(或链接属性)

与链接有关的使用要求被称为链接使用要求，有时也称为链接属性，这些属性通常是以 INTERFACE_LINK 开头及其对应的非 INTERFACE 属性，比如 INTERFACE_LINK_LIBRARIES、INTERFACE_LINK_LIBRARIES_DIRECT、LINK_LIBRARIES、INTERFACE_LINK_LIBRARIES_DIRECT_EXCLUDE 等等。同理，与链接无关的使用要求被称为非链接使用要求，比如编译参数 INTERFACE_COMPILE_OPTIONS、COMPILE_OPTIONS、INTERFACE_COMPILE_FEATURES、COMPILE_FEATURES、INTERFACE_COMPILE_DEFINITIONS、COMPILE_DEFINITIONS 等等。

■ 链接依赖项

链接使用要求的别称，因为链接使用要求的值通常是 target_link_libraries() 命令的依赖项，所以，链接使用要求也被称为链接依赖项，本文将其简称为依赖项。由于链接依赖项含有多重含义，比如，链接依赖项 kk，可理解为将 kk 添加到目标的链接命令中，也可理解为 kk 是目标的链接使用要求，基于此，本文会尽量不把链接使用要求称为链接依赖项，也就是说，本文所说的“链接依赖项”中的“链接”通常理解为动词。

■ 基于以上规则，有以下名称：

可传递链接使用要求，可传递链接依赖项，可传递非链接使用要求，不可传递非链接使用要求等名称，比如 nn 的 INTERFACE_LINK_LIBRARIES 的值为 kk，则，可以称 kk 为 nn 的可传递链接使用要求、可传递链接依赖项、链接依赖项等。

4、对于命令 target_link_libraries(nn kk) 约定如下名称：

■ 依赖目标

将 nn 称为目标或依赖目标，比如，以上示例，kk 的依赖目标是 nn，而 nn 的依赖目标目前未知，因此，nn 的依赖目标指的是 nn 所依赖的下一个目标。

■ 依赖项、直接依赖项

将 kk 称为依赖项、nn 的依赖项或 nn 的直接依赖项，

■ 间接依赖项

将 nn 的直接依赖项 kk 的直接依赖项或更远的依赖项，比如 jj，称为 nn 的间接依赖项，同理，jj 的直接依赖项仍是 nn 的间接依赖项，以此类推。

- 目标文件
将创建的 `nn.*` 文件称为目标文件

5、对于 `add_library()` 命令，约定如下名称：

- 将 `add_library(jj STATIC ...)` 命令中的 `jj` 称为静态库目标简称静态库
- 将 `add_library(jj SHARED ...)` 中的 `jj` 称为动态库目标简称动态库。
- 将 `add_library(jj OBJECT ...)` 中的 `jj` 称为对象库目标简称对象库。
- 将 `add_library(jj INTERFACE)` 中的 `jj` 称为接口库目标简称接口库。
- 将静态库，动态库，对象库，接口库统称为库。注意：静态库、动态库、对象库、接口库都不是真实的文件，他们只是 `add_library()` 命令的目标名称
- 将 `.lib` 文件(或 `.a`)称为静态库文件，如 `jj.lib`、`kk.lib` 等。将 `.dll`(或 `.so`) 文件称为动态库文件。将 `.obj`(或 `.o`) 文件称为对象文件，如 `a.cpp.obj`、`b.cpp.obj` 都是对象文件。接口库没有相应的接口库文件。
- 将静态库文件、动态库文件、对象库文件统称为库文件。

6、所以，在本文中，依赖项的名称可能与静态库、动态库等名称相同(如 `target_link_libraries(nn kk)` 中的 `kk`)，换一种说法就是，依赖项的类型有静态库、动态库、接口库、对象库等，读者应注意区分，同时，还应注意区分以下名称：静态库、静态库文件、对象库、对象文件等，本文尽量避免产生混淆。

7、双冒号表示法

本文使用双冒号表示右侧的内容属于左侧，比如

```
nn::LINK_LIBRARIES = b
```

表示右侧的 `LINK_LIBRARIES` 属性属于左侧的目标 `nn`，其属性值为 `b`，即，目标 `nn` 的属性 `LINK_LIBRARIES` 的值为 `b`

7.3 链接依赖项(向链接命令添加文件)

7.3.1 链接依赖项(向链接命令添加文件)概述

1、前文介绍的与编译有关的参数是非链接使用要求，所以会通过 `target_link_libraries()` 命令将其直接或间接依赖项的非链接使用要求的值添加到依赖目标的编译命令，与此不同的是，`target_link_libraries()` 命令在处理链接使用要求时，会将其直接或间接依赖项所对应的文件(或链接参数)添加到依赖目标的链接命令。本小节主要讲解添加文件的问题(即链接问题)，具体添加什么文件根据依赖项的类型而定，或者说，本小节主要讲解根据依赖项所对应的文件类型将其添加到依赖目标的链接命令的规则。本文不讲解添加链接参数的问题。

2、比如

```
target_link_libraries(a b)    或  
target_link_libraries(a PUBLIC b)
```

以上命令会作如下操作：

1)、处理使用要求

- 将 `b` 放在 `a` 的 `INTERFACE_LINK_LIBRARIES` 和 `LINK_LIBRARIES` 属性中

- 将 b 的使用要求传递给 a
- 将 b 的直接依赖项声明的使用要求传递给 a

2)、链接依赖项(即，向 a 的链接命令添加文件):

- 添加 a 所对应的对象文件：将目标 a 本身所对应的对象文件(比如 a.cpp.obj)添加到链接命令。
- 添加 b 所对应的文件：将 a 的直接依赖项 b 所对应的文件，假设有文件 b.lib，添加到 a 的链接命令(假设 a 有链接命令)。注意：此步骤取决于目标 a 的 LINK_LIBRARIES 属性的值是否为 b 以及 b 的类型，本例是 PUBLIC 链接，并假设 b 有文件 b.lib。
- 添加间接依赖项(即 b 的依赖项)所对应的文件：读取依赖项 b 的 INTERFACE_LINK_LIBRARIES 属性的值，假设为 c，然后将其对应的某个文件，假设为 c.lib，添加到(或传递到)a 的链接命令。
- 假设间接依赖项 c 不再有依赖项，则以上命令一共向链接命令添加了 3 个文件
a.cpp.obj、b.lib、c.lib，
- 根据对 INTERFACE_LINK_LIBRARIES 或 LINK_LIBRARIES 属性的设置情况以及其他对应的规则，可能会缺少某个环节对应的文件。详细的内容见下文。

7.3.2 链接直接依赖项的规则

链接直接依赖项的规则为：读取目标的 LINK_LIBRARIES 属性的值，若有值，则该值便是目标的直接依赖项，然后根据该值所对应依赖项的类型作如下处理，否则目标没有直接依赖项，不适用本规则。

- 若目标的直接依赖是静态库或动态库，若目标有链接命令，则将直接依赖项所对应的库文件(如.lib 文件)添加到目标的链接命令，若目标没有链接命令，则不添加与直接依赖项对应的文件。
- 若目标的直接依赖项是对象库，由于对象库不会使用 lib 命令创建对应的对象库文件(即.lib 文件)，所以将对象库所对应的对象文件(如.obj 文件)添加到目标的对应命令。该规则可以总结为：直接依赖项的对象文件会被链接。注意，由于添加的是对象文件，所以，这里不需要目标必须有链接命令。另外还需注意：此种方法添加的对象文件的路径位于 CMakeFiles/*.dir/objects1.rsp 文件中，而不是位于 CMakeFiles/*.dir/link.txt 文件中

示例 7.3：链接直接静态库依赖项--->目标有链接命令

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk STATIC b.cpp)
add_executable (nn a.cpp)          #目标 nn 使用链接命令(如 link 命令)
target_link_libraries(nn kk)       #向 nn 的 link 命令添加 kk.lib 和 a.cpp.obj
```

目标 nn 的直接依赖项 kk 是静态库且 nn 有链接命令，所以，将 kk 所对应的库文件(如 kk.lib)添加到目标的链接命令。以上命令一共向 nn 的链接命令添加了 2 个文件 a.cpp.obj 和 kk.lib，以上 CMake 命令对应于以下的 VC++命令(简化后)。

注：以上代码执行后，读者可自行查看生成的 makefile 文件内容以了解情况，主要需查看

CMakeFiles/*.dir 目录下的 build.make、link.txt、objects1.rsp 文件，本示例之后的其余示例读者均可自行查看 makefile 文件内容，不再重述。

```
cl /c /Fob.cpp.obj b.cpp          #创建 kk 的对象文件 b.cpp.obj
lib /out:kk.lib b.cpp.obj         #创建 kk 的库文件 kk.lib
cl /c /Foa.cpp.obj a.cpp          #创建 nn 的对象文件
link a.cpp.obj /out:nn.exe kk.lib #向 nn 的链接命令添加 kk.lib 和 a.cpp.obj
```

示例 7.4: 链接直接静态库依赖项--->目标无链接命令

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk STATIC b.cpp)
add_library(nn STATIC a.cpp)      #目标 nn 未使用链接命令, 本例假设使用 lib 命令
target_link_libraries(nn kk)      #向 nn 的 lib 命令添加 a.cp.obj
```

目标 nn 的直接依赖项 kk 是静态库但 nn 没有链接命令(本示例为 lib.exe 命令), 所以, 不会将 kk 对应的文件(如 b.cpp.obj 或 b.lib 等)添加到目标 nn 的 lib.exe 命令。以上命令最终仅向目标 nn 的 lib.exe 命令添加一个文件 a.cpp.obj。以上 CMake 命令对应于以下的 VC++命令(简化后)

```
cl /c /Fob.cpp.obj b.cpp          #创建 kk 的对象文件 b.cpp.obj
lib /out:kk.lib b.cpp.obj         #创建 kk 的库文件 kk.lib
cl /c /Foa.cpp.obj a.cpp          #创建 nn 的对象文件
lib /out:nn.lib a.cpp.obj         #nn 无链接命令, 仅向 nn 的 lib 命令添加了其自身的对象文件
```

示例 7.5: 链接直接依赖项--->对象库

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk OBJECT b.cpp)      #kk 是对象库
add_library(nn STATIC a.cpp)      #目标 nn 未使用链接命令, 本例假设使用 lib 命令
target_link_libraries(nn kk)      #向 nn 的 lib 命令添加 a.cp.obj 和 b.cpp.obj
```

目标 nn 的直接依赖项 kk 是对象库, 无论 nn 有无链接命令, 均将对象库 kk 所对应的对象文件 b.cpp.obj 添加到目标 nn 的对应命令, 因此, 本例向目标 nn 的 lib 命令共添加两个文件 a.cpp.obj 和 b.cpp.obj。以上 CMake 命令对应于以下的 VC++命令(简化后)

```
cl /c /Fob.cpp.obj b.cpp          #创建 kk 的对象文件, 注意: 没有创建库文件的步骤
cl /c /Foa.cpp.obj a.cpp          #创建 nn 的对象文件
lib /out:nn.lib a.cpp.obj b.cpp.obj # nn 无链接命令, 但向 nn 的 lib 命令中添加 a.cpp.obj 和 b.cpp.obj
```

7.3.3 链接间接依赖项的规则

1、链接间接依赖项实际上是传递的间接依赖项所对应的库文件(如.lib 或.a 文件), 这意味着, 如果间接依赖项没有对应的库文件(注意: 对象文件不会被传递), 则会出现无文件可传递的情形。

2、链接间接依赖项的规则为:

读取直接依赖项的 INTERFACE_LINK_LIBRARIES 属性的值, 若有值, 则该值便是目标的间接依赖项, 然后根据该值所对应的类型作如下处理, 否则目标没有间接依赖项, 不适用本规则。

- 若目标的间接依赖项是静态库或动态库, 若目标有链接命令, 则将间接依赖项所对应的库文件(如.lib 文件)添加到目标的链接命令, 若目标没有链接命令, 则不添加与间接依赖项对应的文件。间接依赖项的依赖项重复使用以上规则。
- 若目标的间接依赖项是对象库, 则不将对象库所对应的对象文件(如.obj 文件)添加到目标的对应命令, 注意, 这里不需要目标必须有链接命令, 另外还需注意, 对象库没有对应的库文件(如.lib 文件)。该规则可以总结为: 间接依赖项的对象文件不会被链接

示例 7.6: 链接间接静态库依赖项--->目标有链接命令

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
```


<code>add_library(jj STATIC c.cpp)</code>	#jj 是 nn 的间接依赖项, 是 kk 的直接依赖项, 且是静态库
<code>add_library(kk STATIC b.cpp)</code>	#kk 是 nn 的直接依赖项
<code>target_link_libraries(kk jj)</code>	#将 b.cpp.obj 添加到 kk 的 lib 命令
<code>add_executable (nn a.cpp)</code>	#目标 nn 使用链接命令
<code>target_link_libraries(nn kk)</code>	#jj 通过 kk 将其对应的 jj.lib 添加到(或传递到)nn 的链接命令

目标 nn 的依赖项 kk 的依赖项是 jj, 即 jj 是 nn 的间接依赖项且是静态库, 所以, 将 jj.lib 添加到目标 nn 的链接命令。以上命令最终向目标 nn 的 link 命令添加了三个文件 a.cpp.obj、kk.lib 和 jj.lib。以上 CMake 命令对应于以下的 VC++命令(简化后)

<code>cl /c /Fob.cpp.obj c.cpp</code>	#创建 jj 的对象文件 b.cpp.obj
<code>lib /out:jj.lib c.cpp.obj</code>	#创建 jj 的库文件 jj.lib
<code>cl /c /Fob.cpp.obj b.cpp</code>	#创建 kk 的对象文件 b.cpp.obj
<code>lib /out:kk.lib b.cpp.obj</code>	#创建 kk 的库文件 kk.lib
<code>cl /c /Foa.cpp.obj a.cpp</code>	#创建 nn 的对象文件
<code>link a.cpp.obj /out:nn.exe kk.lib jj.lib</code>	#向 nn 的链接命令添加了 a.cpp.obj、kk.lib、jj.lib

示例 7.7: 链接间接静态库依赖项--->目标无链接命令

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj STATIC c.cpp)      #jj 是 nn 的间接依赖项, 是 kk 的直接依赖项, 且是静态库
add_library(kk STATIC b.cpp)     #kk 是 nn 的直接依赖项
target_link_libraries(kk jj)     #将 b.cpp.obj 添加到 kk 的 lib 命令
add_library(nn STATIC a.cpp)     #目标 nn 没有链接命令
target_link_libraries(nn kk)     #仅向 nn 的 lib 命令添加 a.cpp.obj, jj 对应的 jj.lib 未添加
```

由于 nn 没有链接命令, 所以不会将 nn 的直接依赖项和间接依赖项对应的文件添加到 nn 的对应命令, 所以, 最后仅添加 a.cpp.obj 到 nn 的 lib 命令。但本示例中, 仍可通过 nn 将 jj 对应的库文件 jj.lib 传递到其他目标。以上 CMake 命令对应于以下的 VC++命令(简化后)

<code>cl /c /Fob.cpp.obj c.cpp</code>	#创建 jj 的对象文件 b.cpp.obj
<code>lib /out:jj.lib c.cpp.obj</code>	#创建 jj 的库文件 jj.lib
<code>cl /c /Fob.cpp.obj b.cpp</code>	#创建 kk 的对象文件 b.cpp.obj
<code>lib /out:kk.lib b.cpp.obj</code>	#创建 kk 的库文件 kk.lib
<code>cl /c /Foa.cpp.obj a.cpp</code>	#创建 nn 的对象文件
<code>lib /out:nn.lib a.cpp.obj</code>	#nn 无链接命令, 仅向 nn 的 lib 命令添加了其自身的对象文件 a.cpp.obj

示例 7.8: 链接间接依赖项---> 对象库

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj OBJECT c.cpp)     #jj 是 nn 的间接依赖项, 是 kk 的直接依赖项, 且是对象库
add_library(kk STATIC b.cpp)     #kk 是 nn 的直接依赖项
target_link_libraries(kk jj)     #将 b.cpp.obj 和 c.cpp.obj 添加到 kk 的 lib 命令
add_executable (nn a.cpp)       #目标 nn 使用链接命令
target_link_libraries(nn kk)     #向 nn 的链接命令添加 a.cpp.obj、kk.lib, 未添加 c.cpp.obj
```

由于 jj 没有对应的库文件 jj.lib, 所以不会向 nn 的链接命令添加 jj.lib, 由于 jj 是 nn 的间接依赖, 因此, 也不会添加对象库 jj 对应的对象文件 c.cpp.obj。以上 CMake 命令对应于以下的 VC++命令(简化后)

<code>cl /c /Foc.cpp.obj c.cpp</code>	#创建 jj 的对象文件, 注意: 没有创建库文件的步骤
---------------------------------------	------------------------------

```

cl /c /Fob.cpp.obj b.cpp          #创建 kk 的对象文件
lib /out:kk.lib b.cpp.obj c.cpp.obj # kk 无链接命令，但向 kk 的 lib 命令中添加了 b.cpp.obj 和 c.cpp.obj
cl /c /Foa.cpp.obj a.cpp          #创建 nn 的对象文件
link a.cpp.obj /out:nn.exe kk.lib  #向 nn 的链接命令添加了 a.cpp.obj、kk.lib，未添加 c.cpp.obj

```

7.3.4 依赖项为生成器表达式的链接

若依赖项使用生成器表达式

`$<TARGET_OBJECTS:tgt>`

则可以将 `tgt` 对应的对象文件添加到链接命令中，但不会传递 `tgt` 的其他使用要求。下面以示例进行说明

示例 7.9：依赖项为生成器表达式的链接 1

```

cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk OBJECT f.cpp)
target_compile_options(kk PUBLIC -DEE=3)
add_executable(nn e.cpp)
target_link_libraries(nn $<TARGET_OBJECTS:kk>) #不会使用 kk 的-DEE=3 编译 e.cpp

```

以上命令不会使用 `-DEE=3` 编译 `e.cpp`，但会向 `nn` 的链接命令添加 `f.cpp.obj` 和 `e.cpp.obj`。若将最后一行代码修改为 `target_link_libraries(nn kk)`，则会使用 `-DEE=3` 编译 `e.cpp`，并且仍然会向 `nn` 的链接命令添加 `f.cpp.obj` (直接依赖项的对象文件会被链接)。

示例 7.10：依赖项为生成器表达式的链接 2

```

cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj OBJECT c.cpp)
target_compile_options(jj PUBLIC -DEE=3)
add_library(kk STATIC b.cpp)
#转发 jj 对应的对象文件，但不转发 jj 对应的-DEE=3 参数
target_link_libraries(kk $<TARGET_OBJECTS:jj>) #❶
add_executable(nn a.cpp)
#向 nn 的链接命令添加 a.cpp.obj、kk.lib、c.cpp.obj，但不使用-DEE=3 编译 a.cpp
target_link_libraries(nn kk) #❷

```

向 `nn` 的链接命令添加 `a.cpp.obj`、`kk.lib`、`c.cpp.obj`，但不会使用 `-DEE=3` 编译 `a.cpp`。若把❶处的命令修改为 `target_link_libraries(kk jj)`，则在❷处将不会添加 `jj` 对应的对象文件 `c.cpp.obj` (间接依赖项的对象文件不会被链接)，但会使用 `-DEE=3` 编译 `a.cpp`

7.3.5 链接依赖项总结

表 7.1 和表 7.2 是对链接依赖项的总结

表 7.1 链接直接依赖项

	类型	CMake 命令	目标属性	VC++命令
情形 1	静态库	add_library(kk STATIC b.cpp)	无	cl /c /Fob.cpp.obj b.cpp lib /out:kk.lib b.cpp.obj
	目标有链接	add_executable(nn a.cpp) target_link_libraries(nn kk)	INTERFACE_LINK_LIBRARIES=kk LINK_LIBRARIES=kk	cl /c /Foa.cpp.obj a.cpp link a.cpp.obj /out:nn.exe kk.lib
情形 2	静态库	add_library(kk STATIC b.cpp)	无	cl /c /Fob.cpp.obj b.cpp lib /out:kk.lib b.cpp.obj
	目标无链接	add_library (nn STATIC a.cpp) target_link_libraries(nn kk)	INTERFACE_LINK_LIBRARIES=kk LINK_LIBRARIES=kk	cl /c /Foa.cpp.obj a.cpp lib /out:nn.lib a.cpp.obj
情形 3	对象库	add_library(kk OBJECT b.cpp)	无	cl /c /Fob.cpp.obj b.cpp 注意：无库文件
	目标无链接	add_library (nn STATIC a.cpp) target_link_libraries(nn kk)	INTERFACE_LINK_LIBRARIES=kk LINK_LIBRARIES=kk	cl /c /Foa.cpp.obj a.cpp lib /out:nn.lib a.cpp.obj b.cpp.obj

表 7.2 链接间接依赖项

	类型	CMake 命令	目标属性	VC++命令
情形 1	说明：情形 1 将 jj 对应的库文件 jj.lib 传递给了 nn 的链接命令			
	静态库 (间接)	add_library(jj STATIC c.cpp)	无	cl /c /Fob.cpp.obj c.cpp lib /out:jj.lib c.cpp.obj
	静态库 (直接)	add_library(kk STATIC b.cpp) target_link_libraries(kk jj)	INTERFACE_LINK_LIBRARIES=jj LINK_LIBRARIES=jj	cl /c /Fob.cpp.obj b.cpp lib /out:kk.lib b.cpp.obj
	目标有链接	add_executable (nn a.cpp) target_link_libraries(nn kk)	INTERFACE_LINK_LIBRARIES=kk LINK_LIBRARIES=kk	cl /c /Foa.cpp.obj a.cpp link a.cpp.obj /out:nn.exe kk.lib jj.lib
情形 2	说明：情形 2 的目标没有链接，但仍可能过 nn 继续传递 jj 对应的 jj.lib 到其他目标			
	静态库 (间接)	add_library(jj STATIC c.cpp)	无	cl /c /Fob.cpp.obj c.cpp lib /out:jj.lib c.cpp.obj
	静态库 (直接)	add_library(kk STATIC b.cpp) target_link_libraries(kk jj)	INTERFACE_LINK_LIBRARIES=jj LINK_LIBRARIES=jj	cl /c /Fob.cpp.obj b.cpp lib /out:kk.lib b.cpp.obj
	目标无链接	add_library(nn STATIC a.cpp) target_link_libraries(nn kk)	INTERFACE_LINK_LIBRARIES=kk LINK_LIBRARIES=kk	cl /c /Foa.cpp.obj a.cpp lib /out:nn.lib a.cpp.obj
情形 3	说明：情形 3 的 jj 没有库文件可传递			
	对象库 (间接)	add_library(jj OBJECT c.cpp)	无	cl /c /Foc.cpp.obj c.cpp 注意：无库文件
	静态库 (直接)	add_library(kk STATIC b.cpp) target_link_libraries(kk jj)	INTERFACE_LINK_LIBRARIES=jj LINK_LIBRARIES=jj	cl /c /Fob.cpp.obj b.cpp lib /out:kk.lib b.cpp.obj c.cpp.obj
	目标无链接	add_executable (nn a.cpp) target_link_libraries(nn kk)	INTERFACE_LINK_LIBRARIES=kk LINK_LIBRARIES=kk	cl /c /Foa.cpp.obj a.cpp link a.cpp.obj /out:nn.exe kk.lib

7.4 传递使用要求的详细规则

7.4.1 概述

- 1、由前文讲解可知，链接使用要求会向链接命令(如 `link` 命令)添加参数(含文件及链接参数)，注意：前文仅介绍了添加文件，实际上还可向链接命令添加链接参数。前文介绍 `target_link_libraries()` 命令都是在 `PUBLIC` 参数的条件下进行的，并未讲解 `target_link_libraries()` 命令传递使用要求的详细规则，本小节将讲解 `target_link_libraries()` 命令传递使用要求时的详细规则。
- 2、注意：本文不讲解链接参数的传递，只讲解链接文件，所以，`LINK_LIBRARIES` 和 `INTERFACE_LINK_LIBRARIES` 属性的值是一个依赖项的名称，而不是链接参数的名称。链接参数的规则与链接文件是类似的。
- 3、为便于讲解，本小节作以下名称约定，以 `target_link_libraries(nn XXX kk)` 为例：
 - 此处约定名称的主要目的在于区分“非链接使用要求”和“链接使用要求”，以及链接使用要求具体指的是什么。本文将“使用要求的值”也称为“使用要求”，而链接使用要求的值通常就是依赖项的名称，所以，这里的 `kk` 实际上是指的 `nn` 的链接使用要求或其值，即 `nn` 的链接使用要求是指的 `kk`。
 - `kk` 可传递给 `nn`
是指的可将 `kk` 对应的库文件(如 `kk.lib`)添加到 `nn` 的链接命令，或者说是指的，可将 `nn` 的链接使用要求(即 `kk`)传递给 `nn` 自己。
 - 可将 `kk` 的非链接使用要求传递给 `nn`
是指的可将 `kk` 的非链接使用要求所对应的值添加到 `nn` 对应的命令中，比如，将有关编译参数(非链接使用要求)添加到 `nn` 的编译命令(比如 `cl.exe` 命令)。
 - 同理，“可将 `kk` 的链接使用要求(假设为 `jj`)传递给 `nn`”，则表示可将 `jj` 对应的库文件(如 `jj.lib`)添加到 `nn` 的链接命令。

7.4.2 传递使用要求的详细规则 (INTERFACE_LINK_LIBRARIES 和 LINK_LIBRARIES 属性)

- 1、本文只讲解 `target_link_libraries()` 命令传递使用要求的规则，其他命令如 `target_compile_options()` 命令传递使用要求的规则比 `target_link_libraries()` 命令简单，可参考本文的规则。
- 2、`target_compile_options()` 命令是否可被传递使用要求与以下属性有关

```
INTERFACE_LINK_LIBRARIES
LINK_LIBRARIES
```

这两个属性的值可通过 `target_link_libraries()` 命令的 `PUBLIC`、`PRIVATE`、`INTERFACE` 参数控制。注意：这两个属性的值通常是一个依赖项的名称。

- 3、以下为使用要求的详细传递规则，以 `target_link_libraries(nn XXX kk)` 为例：

1)、LINK_LIBRARIES 属性

若目标 `nn` 的 `LINK_LIBRARIES` 属性的值为依赖项的名称，则该依赖项及该依赖项的可传递使用要求可传递给 `nn`，否则，这些使用要求不能传递给 `nn`。这说明以下问题：

①、对自身使用要求的影响

若目标 `nn` 想使用自己的链接使用要求(注意, 不含目标 `nn` 的非链接使用要求), 则必须为 `LINK_LIBRARIES` 属性设置值, 否则, 目标 `nn` 不能使用自己的链接使用要求。

②、是否接受其他目标的使用要求

若想要目标 `nn` 接受来自目标 `kk` 的可传递使用要求(包括链接使用要求和非链接使用要求), 则目标 `nn` 的 `LINK_LIBRARIES` 属性必须有值 `kk`, 否则, `nn` 不能接受来自 `kk` 的所有使用要求。

■ 示例:

```
nn::LINK_LIBRARIES=kk
```

则可将 `kk` 及 `kk` 的可传递使用要求传递给目标 `nn`, 但 `nn` 自己的非链接使用要求与此属性无关。这里的 `kk` 其实是 `nn` 的链接使用要求, 因此, 这里的“可将 `kk` 传递给 `nn`”, 是指的将 `nn` 的链接使用要求传递给 `nn`, 即 `nn` 自己可以使用自己的链接使用要求。

2)、`INTERFACE_LINK_LIBRARIES` 属性

分以下情形

①、若目标 `nn` 的 `INTERFACE_LINK_LIBRARIES` 属性的值为某依赖项的名称, 比如依赖项的名称为 `kk`(注意: `kk` 还是 `nn` 的链接使用要求), 则该依赖项(`kk`)及该依赖项的可传递使用要求(包括链接使用要求和非链接使用要求)可通过该属性的目标(即 `nn`)传递给 `nn` 的依赖目标, 即, 可通过 `nn` 继续传递给 `nn` 的下一个目标。注意: 目标 `nn` 的非链接使用要求与此属性无关。

②、若目标 `nn` 的 `INTERFACE_LINK_LIBRARIES` 属性的值为 `$<LINK_ONLY:kk>`, 其中 `kk` 是某依赖项的名称, 则依赖项 `kk` 及 `kk` 的可传递链接使用要求可通过 `nn` 继续传递给下一个目标, 但是, `kk` 的可传递非链接使用要求终止于此, 不能再继续传递。可以理解为 `$<LINK_ONLY:kk>` 终止了 `kk` 的可传递非链接使用要求的继续传递。若想终止 `kk` 的链接使用要求被继续传递, 则不能对 `INTERFACE_LINK_LIBRARIES` 属性设置值, 这也意味着会终止所有使用要求的继续传递。注意: 目标 `nn` 的非链接使用要求与此属性无关。

■ 注意: 其他命令如 `target_compile_options()` 命令, 主要用于控制非链接使用要求的传递, 非链接使用要求对应的属性, 即 `INTERFACE_*` 属性, 不存在类似 `$<LINK_ONLY:kk>` 的值, 其传递规则参考本规则的第 1 条规则进行。

■ 以上规则这说明以下问题

- 目标 `nn` 的非链接使用要求与此属性无关。
- 若目标 `nn` 想将自己的链接使用要求传递给另一个目标, 则必须为 `INTERFACE_LINK_LIBRARIES` 属性设置值, 其值可以是依赖项名称或 `$<LINK_ONLY:...>`, 否则, 目标 `nn` 的链接使用要求(或该属性的依赖项值)不能被传递。
- 若想将目标 `kk` 的可传递非链接使用要求通过目标 `nn` 继续传递给 `nn` 的下一个目标, 则必须将 `nn` 的 `INTERFACE_LINK_LIBRARIES` 属性设置为值 `kk`, 不能是 `$<LINK_ONLY:kk>`, 否则, `kk` 的可传递非链接使用要求不能通过 `nn` 继续传递。
- 若想终止所有的使用要求被继续传递, 必须不对 `INTERFACE_LINK_LIBRARIES` 属性设置值, 或者说, 想要使用要求通过 `nn` 继续传递, 必须对 `INTERFACE_LINK_LIBRARIES` 属性设置值, 否则, 使用要求将终止于 `nn`

■ 示例:

```
nn::INTERFACE_LINK_LIBRARIES=kk,
```

则可将 `kk` 及 `kk` 的可传递使用要求通过目标 `nn` 传递给 `nn` 的下一个目标(即 `nn` 的依赖目标)。这里的 `kk` 是 `nn` 的可传递链接使用要求, 因此, 这里的“可将 `kk` 传递给 `nn` 的下一个目标”是指的可将 `nn` 的可传递链接使用要求传递给 `nn` 的下一个目标。

4、以上规则可简述为：

- 目标 `nn` 的 `INTERFACE_LINK_LIBRARIES` 和 `LINK_LIBRARIES` 属性用于判断 `nn` 是否可使用自己的链接使用要求和依赖项的可传递使用要求，以及是否可通过 `nn` 继续传递这些使用要求。需要注意的是：`nn` 的非链接使用要求与这两个属性无关，不受这两个属性影响。

或者说

- 目标 `nn` 的 `LINK_LIBRARIES` 属性的值决定是否将该依赖项以及该依赖项的可传递使用要求传递给 `nn` 自己，即 `nn` 自己是否能使用这些使用要求。只有该属性有值时，才能将这些使用要求传递给 `nn`，否则不能传递给 `nn`，也就是说 `nn` 不能使用这些使用要求。或者理解为 `nn` 的 `LINK_LIBRARIES` 属性指定了 `nn` 的直接链接使用要求(或称为直接链接依赖项)。只有当 `nn` 有直接链接使用要求时，`nn` 才能链接这些依赖项，从而才能使用这些依赖项传递进来的使用要求，因此，若 `nn` 没有直接链接使用要求也就不能链接这些依赖项以及不能使用这些依赖项传递进来的使用要求，从而没有依赖项可链接，也没有使用要求可被 `nn` 使用。需要注意的是：`nn` 的非链接使用要求与这属性无关，不受影响。
- `nn` 的 `INTERFACE_LINK_LIBRARIES` 属性的值决定该依赖项以及该依赖项的可传递使用要求是否可通过 `nn` 继续传递给 `nn` 的下一个目标。需要注意的是：`nn` 的非链接使用要求的传递性与此属性无关，不受影响。

5、`target_link_libraries()`命令通过使用 `PUBLIC`、`PRIVATE`、`INTERFACE` 三个选项来对 `INTERFACE_LINK_LIBRARIES` 和 `LINK_LIBRARIES` 属性的值进行设置，从而决定怎样处理目标及依赖项的使用要求，所以，`target_link_libraries(nn XXX kk)`命令，其中 `XXX` 是 `PUBLIC`、`PRIVATE`、`INTERFACE` 选项之一，有以下作用：

- 处理目标 `nn` 自己的链接使用要求
根据 `target_link_libraries()`命令中的 `PUBLIC`、`PRIVATE`、`INTERFACE` 选项，判断 `nn` 的链接使用要求(即 `kk`)是否可继续传递给下一个目标以及自己是否能使用该链接使用要求。这里需要注意的是，目标 `nn` 的非链接使用要求与 `target_link_libraries()`命令无关。
- 处理直接依赖项 `kk` 的使用要求
根据 `target_link_libraries()`命令中的 `PUBLIC`、`PRIVATE`、`INTERFACE` 选项，判断 `kk` 的可传递使用要求(包括链接和非链接使用要求)是否可应用于目标 `nn` 以及是否可继续传递给 `nn` 的下一个目标(即 `nn` 的依赖目标)。
- `kk` 的直接依赖项(即 `nn` 的间接依赖项)的可传递使用要求继续使用以上规则处理。

6、表 7.3 列出了 `target_link_libraries()`命令对这两个属性的设置以及对使用要求的处理情况。

表 7.3 使用要求的传递规则

目标	目标的属性取值	传递规则
nn	LINK_LIBRARIES=kk INTERFACE_LINK_LIBRARIES=kk	对应于 <code>target_link_libraries(nn PUBLIC kk)</code> 可将 <code>kk</code> 及 <code>kk</code> 的可传递使用要求(包括链接和非链接使用要求)传递给 <code>nn</code> 以及 <code>nn</code> 的依赖目标。注意： <code>kk</code> 是 <code>nn</code> 的链接使用要求，下同。
	LINK_LIBRARIES 无值 INTERFACE_LINK_LIBRARIES=kk	对应于 <code>target_link_libraries(nn INTERFACE kk)</code> 不会将 <code>kk</code> 及 <code>kk</code> 的可传递使用要求传递给 <code>nn</code> ，但能传递给 <code>nn</code> 的依赖目标，即， <code>nn</code> 相当于是一个中转站(或接口)，说简单点就是，可以继续传递给下一个目标，但

	目标 nn 自己不能使用这些使用要求。
LINK_LIBRARIES=kk INTERFACE_LINK_LIBRARIES 无值	对应于 target_link_libraries(nn PRIVATE kk)且目标 nn 是动态库或可执行文件。 可将 kk 及 kk 的可传递使用要求传递给 nn，但不能传递给 nn 的依赖目标，即不能继续传递给下一个目标。
LINK_LIBRARIES=kk INTERFACE_LINK_LIBRARIES=\${<LINK_ONLY:kk>}	对应于 target_link_libraries(nn PRIVATE kk) 且目标 nn 是静态库或对象库。 将 kk 传递给 nn 以及 nn 的依赖目标。 将 kk 的可传递链接使用要求传递给 nn 及 nn 的依赖目标，但 kk 的可传递非链接使用要求只能传递给 nn，不能传递给 nn 的依赖目标，即，kk 的非链接使用要求的传递终止于 nn，不能继续传递，但是，kk 的可传递链接使用要求仍可继续传递。可以理解为 \${<LINK_ONLY:kk>}终止了 kk 的可传递非链接使用要求的继续传递。
LINK_LIBRARIES 无值 INTERFACE_LINK_LIBRARIES=\${<LINK_ONLY:kk>}	此情形不能使用 target_link_libraries()命令设置，本文不重点讲解。从设置的属性的值可以得出以下结论： 将 kk 及 kk 的可传递链接使用要求传递给 nn 的依赖目标，但不能传递给 nn 自己。kk 的可传递非链接使用要求既不能传递给 nn，也不能传递给 nn 的依赖目标

7.5 对 target_link_libraries()命令进一步的讲解

7.5.1 target_link_libraries(PUBLIC)

比如：target_link_libraries(nn PUBLIC kk)或 target_link_libraries(nn kk)

以上命令表示将 kk、kk 的可传递使用要求以及 kk 的直接依赖项的可传递使用要求，传递给 nn 及 nn 的依赖目标。表 7.4 是对该处理的总结，详细的处理内容如下：

- 处理依赖目标 nn 自身的使用要求
 - 设置 nn 的链接属性：


```
INTERFACE_LINK_LIBRARIES=kk
LINK_LIBRARIES=kk
```
 - 处理 nn 的非链接使用要求：根据 nn 的非链接使用要求的设置情况进行处理
 - 处理 nn 的链接使用要求：
 - 由于 LINK_LIBRARIES=kk，所以，将 kk 传递给 nn
 - 由于 INTERFACE_LINK_LIBRARIES=kk，所以，将 kk 传递给 nn 的依赖目标。
- 处理直接依赖项 kk
 - 处理 kk 的非链接使用要求：将 kk 的可传递非链接使用要求传递给 nn 及 nn 的依赖目标
 - 处理 kk 的链接使用要求：将 kk 的可传递链接使用要求传递给 nn 及 nn 的依赖目标
- 处理 kk 的直接依赖项

- 将 kk 的直接依赖项的可传递非链接使用要求传递给 nn 及 nn 的依赖目标
- 将 kk 的直接依赖项的可传递链接使用要求传递给 nn 及 nn 的依赖目标

表 7.4 target_link_libraries(nn PUBLIC kk)

处理对象	处理的属性	处理规则
处理依赖目标 nn	设置 nn 的链接属性	LINK_LIBRARIES=kk INTERFACE_LINK_LIBRARIES=kk
	非链接使用要求	视设置情况处理
	链接使用要求	将 kk 传递给 nn 及 nn 的依赖目标
处理直接依赖项 kk	可传递非链接使用要求	传递给 nn 及 nn 的依赖目标
	可传递链接使用要求	传递给 nn 及 nn 的依赖目标
处理 kk 的直接依赖项	可传递非链接使用要求	传递给 nn 及 nn 的依赖目标
	可传递链接使用要求	传递给 nn 及 nn 的依赖目标

7.5.2 target_link_libraries(PRIVATE)

比如：target_link_libraries(nn PRIVATE kk)

分以下两种情形：

1、情形 1：nn 是静态库或对象库：

将 kk、kk 的可传递链接使用要求及 kk 的直接依赖项的可传递链接使用要求传递到 nn 及 nn 的依赖目标，但是，kk 的可传递非链接使用要求以及 kk 的直接依赖项的可传递非链接使用要求将终止于 nn，即，这些使用要求可以传递给 nn，但不会传递给 nn 的依赖目标。表 7.5 是对该处理的总结，详细的处理内容如下：

- 处理依赖目标 nn 自身的使用要求
 - 设置 nn 的链接属性：


```
LINK_LIBRARIES = kk
INTERFACE_LINK_LIBRARIES = $<LINK_ONLY:kk>
```
 - 处理 nn 的非链接使用要求：根据 nn 的非链接使用要求的设置情况进行处理
 - 处理 nn 的链接使用要求：
 - 由于 LINK_LIBRARIES=kk，所以，将 kk 传递给 nn
 - 由于 INTERFACE_LINK_LIBRARIES=\$<LINK_ONLY:kk>，所以，将 kk 传递给 nn 的依赖目标。
- 处理直接依赖项 kk
 - 处理 kk 的非链接使用要求：kk 的可传递非链接使用要求终止于 nn，即，这些使用要求会被传递给 nn，但不会传递给 nn 的依赖目标
 - 处理 kk 的链接使用要求：将 kk 的可传递链接使用要求传递给 nn 及 nn 的依赖目标
- 处理 kk 的直接依赖项
 - kk 的直接依赖项的可传递非链接使用要求终止于 nn，即，这些使用要求会被传递给 nn，但不会传递给 nn 的依赖目标
 - 将 kk 的直接依赖项的可传递链接使用要求传递给 nn 及 nn 的依赖目标

表 7.5 target_link_libraries(nn PRIVATE kk)，其中 nn 为静态库或对象库

处理对象	处理的属性	处理规则
处理依赖目标 nn	设置 nn 的链接属性	LINK_LIBRARIES=kk INTERFACE_LINK_LIBRARIES= \$<LINK_ONLY:kk>
	非链接使用要求	视设置情况处理
	链接使用要求	将 kk 传递给 nn 及 nn 的依赖目标
处理直接依赖项 kk	可传递非链接使用要求	由于 nn 的 INTERFACE_LINK_LIBRARIES= \$<LINK_ONLY:kk>，所以，kk 的非链接使用要求终止于 nn，即这些使用要求可以传递给 nn，但不会传递给 nn 的依赖目标
	可传递链接使用要求	传递给 nn 及 nn 的依赖目标
处理 kk 的直接依赖项	可传递非链接使用要求	终止于 nn，即传递给 nn，但不会传递给 nn 的依赖目标
	可传递链接使用要求	传递给 nn 及 nn 的依赖目标

示例 7.11: target_link_libraries(kk PRIVATE jj)且 kk 是静态库

①、源文件准备

在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp(主源文件)
#include<iostream>
extern int b;
extern int c;
int main(){    std::cout<<b<<c<<std::endl;    return 0;}
```

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
int b=1;
```

在目录 g:/qt1 中的 c.cpp 中编写如下代码

```
//g:/qt1/c.cpp
int c=1;
```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj STATIC c.cpp)                #jj 库
target_compile_options(jj PUBLIC -DEE=3)   #❶-DEE=3 传递给 jj
add_library(kk STATIC b.cpp)                #kk 库
target_compile_options(kk PUBLIC -DFF=4)   #❷-DFF=4 传递给 kk
target_link_libraries(kk PRIVATE jj)        #❸-DEE=3 传递给 kk 并终止于 kk，但-DFF=4 仍可传递
add_executable(nn a.cpp)                    #nn 可执行文件
target_link_libraries(nn kk)                #使用-DFF=4 编译，向 link 添加 a.cpp.obj, kk.lib, jj.lib
```

表 7.6 示例 1 设置的属性及对应的 VC++命令

处理对象	设置的目标属性	对应的 VC++命令
jj 库	COMPILE_OPTIONS=-DEE=3	cl -DEE=3 /c /Foc.cpp.obj c.cpp
	INTERFACE_COMPILE_OPTIONS=-DEE=3	lib /out:jj.lib c.cpp.obj
kk 库	COMPILE_OPTIONS=-DFF=4	cl -DFF=4 -DEE=3 /c /Fob.cpp.obj b.cpp
	INTERFACE_COMPILE_OPTIONS=-DFF=4	lib /out:kk.lib b.cpp.obj
	LINK_LIBRARIES=jj	
	INTERFACE_LINK_LIBRARIES=\$<LINK_ONLY:jj>	
nn	LINK_LIBRARIES=kk	cl -DFF=4 /c /Foa.cpp.obj a.cpp
	INTERFACE_LINK_LIBRARIES=kk	link a.cpp.obj /out:nn.exe kk.lib jj.lib

表 7.6 是本示例各对象设置的属性及对应的 VC++命令。以下是处理各对象的步骤

A: 处理 nn

- 1) 处理 nn 的非链接使用要求: nn 没有非链接使用要求, 不需处理。注意: nn 的非链接使用要求与 nn 的 LINK_LIBRARIES 和 INTERFACE_LINK_LIBRARIES 属性无关。
- 2) 读取 nn 的 LINK_LIBRARIES 属性的值, 该属性有值 kk, 说明可将 kk 及 kk 的可传递使用要求传递给 nn。
- 3) 处理 nn 的链接使用要求: 将 kk.lib 添加到 nn 的链接命令。
- 4) 处理 kk 的可传递使用要求: 由②知, kk 的非链接使用要求-DFF=4 是可传递的, 所以, 将该参数添加到 nn 的编译命令。
- 5) 处理其他使用要求: 由③知, kk 的 INTERFACE_LINK_LIBRARIES 属性的值为 \$<LINK_ONLY:jj>, 说明 kk 的链接使用要求 jj 可传递给 nn, kk 的直接依赖项 jj 的非链接使用要求-DEE=3 终止于 kk, 不能传递给 nn, 所以, 最终将 jj.lib 添加到 nn 的链接命令, 不会将 jj 的可传递非链接使用要求-DEE=3 添加到 nn 的编译命令, 由于 jj 没有进一步的 target_link_libraries()命令, 所以到此结束, 最终结果是, 使用-DFF=4 编译 nn 的 a.cpp, 将 a.cpp.obj、kk.lib、jj.lib 添加到 nn 的链接命令。

B: 处理 kk

- 1) 处理 kk 的非链接使用要求: 由②知, -DFF=4 是 kk 的非链接使用要求, 所以将该参数添加到 kk 的编译命令。注意: 此时的 kk 的非链接使用要求也是与 kk 的 LINK_LIBRARIES 和 INTERFACE_LINK_LIBRARIES 属性无关的。
- 2) 读取 kk 的 LINK_LIBRARIES 属性的值, 该属性有值 jj, 说明可将 jj 及 jj 的可传递使用要求传递给 kk
- 3) 处理 kk 的链接使用要求: 由于 jj 是静态库, 并且 kk 没有链接命令, 所以不向 kk 的 lib 命令添加文件(参见 7.3.2)。
- 4) 处理 jj 的可传递使用要求: 由①知, jj 的非链接使用要求-DEE=3 是可传递的, 所以, 将该参数添加到 kk 的编译命令。本示例 jj 没有链接使用要求, 所以不需处理。至此结束对 kk 的处理, 最终结果是, 使用-DEE=3 和-DFF=4 编译 kk 的 b.cpp, 将 b.cpp.obj 添加到 kk 的 lib 命令。

C: 处理 jj

由于 jj 没有 target_link_libraries()命令, 所以比较简单, 使用-DEE=3 编译 jj 的 c.cpp, 将 c.cpp.obj 添加到 jj 的 lib 命令。

2、情形 2: nn 是共享库或可执行文件

将 `kk`、`kk` 的可传递使用要求(包括链接使用要求和非链接使用要求)及 `kk` 的直接依赖项的可传递使用要求传递给 `nn`，但不能传递给 `nn` 的依赖目标，即，这些使用要求都不会被继续传递给下一个目标。表 7.7 是对该处理的总结，详细的处理内容如下：

- 处理依赖目标 `nn` 自身的使用要求
 - 设置 `nn` 的链接属性：
 `LINK_LIBRARIES=kk`
 `INTERFACE_LINK_LIBRARIES` 无值
 - 处理 `nn` 的非链接使用要求：根据 `nn` 的非链接使用要求的设置情况进行处理
 - 处理 `nn` 的链接使用要求：
 - 由于 `LINK_LIBRARIES=kk`，所以，将 `kk` 传递给 `nn`
 - 由于 `INTERFACE_LINK_LIBRARIES` 无值，所以，不会将 `kk` 传递给 `nn` 的依赖目标。
- 处理直接依赖项 `kk`
`kk` 的可传递链接和非链接使用要求都将终止于 `nn`，即，这些使用要求会被传递给 `nn`，但不会传递给 `nn` 的依赖目标
- 处理 `kk` 的直接依赖项
`kk` 的直接依赖项的可传递链接和非链接使用要求都将终止于 `nn`，即，这些使用要求会被传递给 `nn`，但不会传递给 `nn` 的依赖目标

表 7.7 `target_link_libraries(nn PRIVATE kk)`，其中 `nn` 为动态库或可执行文件

处理对象	处理的属性	处理规则
处理依赖目标 <code>nn</code>	设置 <code>nn</code> 的属性	<code>LINK_LIBRARIES=kk</code> <code>INTERFACE_LINK_LIBRARIES</code> 无值
	非链接使用要求	视设置情况处理
	链接使用要求	将 <code>kk</code> 传递给 <code>nn</code> ，但不会传递给 <code>nn</code> 的依赖目标
处理直接依赖项 <code>kk</code>	非链接使用要求	二者都终止于 <code>nn</code> ，即都会传递给 <code>nn</code> ，但不会传递给 <code>nn</code> 的依赖目标
	链接使用要求	
处理 <code>kk</code> 的直接依赖项	非链接使用要求	二者都终止于 <code>nn</code> ，即都会传递给 <code>nn</code> ，但不会传递给 <code>nn</code> 的依赖目标
	链接使用要求	

7.5.3 `target_link_libraries(INTERFACE)`

比如：`target_link_libraries(nn INTERFACE kk)`

表示 `nn` 仅仅作为中转站或接口，即，`nn` 自己不接受任何使用要求，包括 `nn` 自己的链接使用要求(注意，`nn` 的非链接使用要求不受影响)，但可将这些使用要求继续传递给 `nn` 的下一个目标。表 7.8 是对该处理的总结，详细的处理内容如下：

- 处理依赖目标 `nn` 自身的使用要求
 - 设置 `nn` 的链接属性：
 `LINK_LIBRARIES` 无值
 `INTERFACE_LINK_LIBRARIES = kk`
 - 处理 `nn` 的非链接使用要求：根据 `nn` 的非链接使用要求的设置情况进行处理
 - 处理 `nn` 的链接使用要求：

- 由于 LINK_LIBRARIES 无值，所以，不会将 kk 传递给 nn
- 由于 INTERFACE_LINK_LIBRARIES=kk，所以，将 kk 传递给 nn 的依赖目标。
- 处理直接依赖项 kk
 - kk 的可传递链接和非链接使用要求不会传递给 nn，但会传递给 nn 的依赖目标
- 处理 kk 的直接依赖项
 - kk 的直接依赖项的可传递链接和非链接使用要求不会传递给 nn，但会传递给 nn 的依赖目标

表 7.8 target_link_libraries(nn INTERFACE kk)

处理对象	处理的属性	处理规则
处理依赖目标 nn	设置 nn 的属性	LINK_LIBRARIES=无值 INTERFACE_LINK_LIBRARIES= kk
	非链接使用要求	视设置情况处理
	链接使用要求	不会将 kk 传递给 nn，但会将 kk 传递给 nn 的依赖目标
处理直接依赖项 kk	非链接使用要求	不会传递给 nn，但会传递给 nn 的依赖目标(即，nn 为中转站)
	链接使用要求	
处理 kk 的直接依赖项	非链接使用要求	不会传递给 nn，但会传递给 nn 的依赖目标(即，nn 为中转站)
	链接使用要求	

示例 7.12：使用要求的传递

- ①、源文件准备：读者可自行准备源文件
- ②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

#g:/qt1/CMakeLists.txt

```

cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(ii STATIC d.cpp)                #❶ii 库
target_compile_options(ii PUBLIC -DDD=2)    #❷
add_library(jj STATIC c.cpp)                #❸jj 库
target_compile_options(jj PRIVATE -DEE=3)    #❹
target_link_libraries(jj INTERFACE ii)       #❺
add_library(kk STATIC b.cpp)                #❻kk 库
target_compile_options(kk PUBLIC -DFF=4)     #❼
target_link_libraries(kk PRIVATE jj)         #❽
add_executable(nn a.cpp)                    #❾nn
target_link_libraries(nn kk)

```

表 7.9 示例 2 设置的属性及对应的 VC++命令

处理对象	设置的目标属性	对应的 VC++命令
ii 库	COMPILE_OPTIONS=-DDD=2	cl -DDD=2 /c /Fod.cpp.obj d.cpp
	INTERFACE_COMPILE_OPTIONS=-DDD=2	lib /out:ii.lib d.cpp.obj

jj 库	COMPILE_OPTIONS=-DEE=3 INTERFACE_COMPILE_OPTIONS 无值 LINK_LIBRARIES 无值 INTERFACE_LINK_LIBRARIES=ii	cl -DEE=3 /c /Foc.cpp.obj c.cpp lib /out:jj.lib c.cpp.obj 注：jj 不使用自己的链接使用要求和传递进来的使用要求，但仍可继续传递。jj 的非链接使用要求只能自己使用，不能被传递
kk 库	COMPILE_OPTIONS=-DFF=4 INTERFACE_COMPILE_OPTIONS=-DFF=4 LINK_LIBRARIES=jj INTERFACE_LINK_LIBRARIES=\${LINK_ONLY:jj}>	cl -DDD=2 -DFF=4 /c /Fob.cpp.obj b.cpp lib /out:kk.lib b.cpp.obj 注：kk 会阻止来自 jj 的非链接使用要求，但自己的非链接使用要求仍可被传递
nn	LINK_LIBRARIES=kk INTERFACE_LINK_LIBRARIES=kk	cl -DFF=4 /c /Foa.cpp.obj a.cpp link a.cpp.obj /out:nn.exe kk.lib jj.lib ii.lib

表 7.9 是本示例各对象设置的属性及对应的 VC++命令。以下是处理各对象的步骤

A：处理 nn

- 1) 处理 nn 的非链接使用要求：nn 没有非链接使用要求，不需处理。注意：nn 的非链接使用要求与 nn 的 LINK_LIBRARIES 和 INTERFACE_LINK_LIBRARIES 属性无关。
- 2) 读取 nn 的 LINK_LIBRARIES 属性的值，该属性有值 kk，说明可将 kk 及 kk 的可传递使用要求传递给 nn。
- 3) 处理 nn 的链接使用要求：将 kk.lib 添加到 nn 的链接命令。
- 4) 处理 kk 的可传递使用要求：由⑦知，kk 的非链接使用要求-DFF=4 是可传递的，所以，将该参数添加到 nn 的编译命令。注意：kk 的非链接使用要求是否可传递，不是参考的 INTERFACE_LINK_LIBRARIES 属性，而是参考的 INTERFACE_COMPILE_OPTIONS 属性
- 5) 处理其他使用要求：由⑧知，kk 的 INTERFACE_LINK_LIBRARIES 属性的值为 \${LINK_ONLY:jj}>，说明 kk 的链接使用要求 jj 可传递给 nn，kk 的直接依赖项 jj 的非链接使用要求-DEE=3 终止于 kk(其实-DEE=3 不可传递)，不能传递给 nn，所以，最终将 jj.lib 添加到 nn 的链接命令，不会将 jj 的可传递非链接使用要求-DEE=3 添加到 nn 的编译命令，同理，jj 的直接依赖项 ii 的非链接使用要求也终止于 kk，但链接使用要求 ii 仍可继续传递，继续查看 jj 的 INTERFACE_LINK_LIBRARIES 属性的值为 ii，说明 ii 的可传递使用要求不会被 jj 阻止，可正常传递，最后查看 ii，由于 kk 已经终止了 ii 的非链接使用要求的传递，并且 ii 也没有 target_link_libraries()命令，至此处理结束。
- 6) 最终结果是，使用-DFF=4 编译 nn 的 a.cpp，将 a.cpp.obj、kk.lib、jj.lib、ii.lib 添加到 nn 的链接命令。

B：处理 kk

- 1) 处理 kk 的非链接使用要求：由⑦知，-DFF=4 是 kk 的非链接使用要求，所以将该参数添加到 kk 的编译命令。注意：此时的 kk 的非链接使用要求也与 kk 的 INTERFACE_LINK_LIBRARIES 和 LINK_LIBRARIES 属性无关。
- 2) 读取 kk 的 LINK_LIBRARIES 属性的值，该属性有值 jj，说明可将 jj 及 jj 的可传递使用要求传递给 kk
- 3) 处理 kk 的链接使用要求：由于 jj 是静态库，并且 kk 没有链接命令，所以不向 kk 的 lib 命令添加文件。
- 4) 处理 jj 的可传递链接使用要求：由于 ii 是静态库，并且 kk 没有链接命令，所以不向 kk 的 lib 命令添加文件。

- 5) 处理 jj 的可传递使用要求：由④知，jj 的非链接使用要求-DEE=3 是不可传递的，所以，不会将该参数添加到 kk 的编译命令。
- 6) 处理 ii 的可传递使用要求：本示例 ii 没有链接使用要求，所以不需处理。由②知，ii 的非链接使用要求-DDD=2 是可传递的，所以，将该参数添加到 kk 的编译命令。至此结束对 kk 的处理。
- 7) 最终结果是，使用-DDD=2、-DEE=3 和-DFF=4 编译 kk 的 b.cpp，将 b.cpp.obj 添加到 kk 的 lib 命令。

C: 处理 jj

由④知，jj 的非链接使用要求是 PRIVATE 的，表示 jj 的非链接使用要求不会被传递，但 jj 自己可以使用，所以，使用-DEE=3 编译 jj 的 c.cpp。注意：jj 的非链接使用要求与 jj 的 INTERFACE_LINK_LIBRARIES 和 LINK_LIBRARIES 属性无关。然后查看 jj 的 LINK_LIBRARIES 属性，该属性无值，所以，jj 不会使用自己的链接使用要求以及传递进来的使用要求，所以，最终仅使用 jj 自己的非链接使用要求-DEE=3 编译 jj 的 c.cpp，并将 c.cpp.obj 添加到 jj 的 lib 命令。

D: 处理 ii

由于 ii 没有 target_link_libraries() 命令，所以比较简单，使用-DDD=2 编译 ii 的 d.cpp，将 d.cpp.obj 添加到 ii 的 lib 命令。

7.6 INTERFACE_LINK_LIBRARIES_DIRECT 和 INTERFACE_LINK_LIBRARIES_DIRECT_EXCLUDE 目标属性

- 1、INTERFACE_LINK_LIBRARIES_DIRECT 属性用于为依赖目标指定直接依赖项，该属性的目标需要位于依赖目标的依赖链中，否则，该属性的设置不会对依赖目标起作用。需要注意的是，若存在循环依赖，可能依赖项会被链接多次，在这种情况下，可使用 INTERFACE_LINK_LIBRARIES_DIRECT_EXCLUDE 属性来排除多余的循环依赖项。还需注意的是，最终依赖目标自己不会位于自己的依赖链中，比如


```
target_link_libraries(nn kk)
```

 kk 位于 nn 的依赖链中，但 nn 不会位于 nn 的依赖链中。
- 2、INTERFACE_LINK_LIBRARIES_DIRECT_EXCLUDE 属性用于排除依赖目标的直接依赖项，该属性排除的是该属性目标所在依赖链上的最终依赖目标的直接依赖项，需要注意的是，间接依赖项不会被排除，排除后的直接依赖项还可以以间接依赖项的形式添加到依赖目标的依赖链上。
- 3、以上两个属性没有对应的非 INTERFACE 属性，并且需要通过 set_property() 等专门设置属性的命令对其进行设置。

示例 7.13: INTERFACE_LINK_LIBRARIES_DIRECT 属性的使用

- ①、源文件准备：读者可自行准备源文件
- ②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
```

```

project(xxxx)
add_library(hh STATIC e.cpp)
add_library(ii STATIC d.cpp)
target_link_libraries(ii PRIVATE hh )           # ii←hh, 即 hh 添加到 ii 链接命令
add_library(jj STATIC c.cpp)
target_link_libraries(jj PRIVATE ii )           # jj←ii
add_library(kk STATIC b.cpp)                     #注意: kk 没有与 jj 建立链接关系
add_executable(nn a.cpp)
target_link_libraries(nn kk)                     # nn←kk

#通过 kk 将 jj 设置为依赖目标 nn 的直接依赖项。
#注意: kk 需位于 nn 的依赖链中, 否则以下属性将不会对 nn 起作用。
set_property(TARGET kk PROPERTY INTERFACE_LINK_LIBRARIES_DIRECT jj)

#注意: 以下设置将不会对 nn 起作用, 因为 nn 不位于 nn 的依赖链中
#set_property(TARGET nn PROPERTY INTERFACE_LINK_LIBRARIES_DIRECT jj)

```

以上示例将把依赖 `jj.lib`、`kk.lib`、`ii.lib`、`hh.lib` 添加到 `nn` 的 `link` 命令中。

示例 7.14: 依赖项被链接多次

- ①、源文件准备: 读者可自行准备源文件
- ②、在目录 `g:/qt1` 中的 `CMakeLists.txt` 中编写如下代码

#g:/qt1/CMakeLists.txt

```

cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(hh STATIC e.cpp)
add_library(ii STATIC d.cpp)
target_link_libraries(ii PRIVATE hh )           #ii←hh
add_library(jj STATIC c.cpp)
target_link_libraries(jj PRIVATE ii )           # jj←ii
add_library(kk STATIC b.cpp)
target_link_libraries(kk PUBLIC jj)             # kk←jj
add_executable(nn a.cpp)
target_link_libraries(nn hh)                     #nn←hh
#通过 hh 将 kk 设置为依赖目标 nn 的直接依赖项
set_property(TARGET hh PROPERTY INTERFACE_LINK_LIBRARIES_DIRECT kk)
#以下设置将不会对 nn 起作用, 因为 jj 不位于 nn 的依赖链中
#set_property(TARGET jj PROPERTY INTERFACE_LINK_LIBRARIES_DIRECT kk)
#以下命令表示, 通过 ii 从依赖目标中移除直接依赖项 hh
#set_property(TARGET ii PROPERTY INTERFACE_LINK_LIBRARIES_DIRECT_EXCLUDE hh)

```

以上示例将依次把 `kk.lib`、`hh.lib`、`jj.lib`、`ii.lib`、`hh.lib` 添加到 `nn` 的 `link` 命令中, 其中 `hh.lib` 被添加了两次, 一次在直接链接时, 一次在通过 `kk` 间接链接时。若删除最后一行的注释, 以上示例将依赖向 `nn` 的 `link` 命令添加 `kk.lib`、`jj.lib`、`ii.lib`、`hh.lib`。

示例 7.15: INTERFACE_LINK_LIBRARIES_DIRECT_EXCLUDE 属性的使用

- ①、源文件准备：读者可自行准备源文件
- ②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

#g:/qt1/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj STATIC c.cpp)
add_library(kk STATIC b.cpp)
target_link_libraries(kk PUBLIC jj)
add_executable(nn a.cpp)
target_link_libraries(nn kk)
#通过 jj 将依赖目标 nn 的直接依赖项 kk 移除
set_property(TARGET jj PROPERTY INTERFACE_LINK_LIBRARIES_DIRECT_EXCLUDE kk)
```

以上命令将不会向 nn 的 link 命令添加 kk.lib、jj.lib，因为 kk 已从 nn 的直接依赖项中移除，不再是 nn 的直接依赖项了。

第 8 章 向 CMake 项目添加其他文件

8.1 添加源文件

8.1.1 添加源文件的属性和变量汇总

可以使用表 8.1 中的属性或变量以及对应的命令向目标中添加源文件，建议使用 `target_source()` 命令添加源文件，因为该命令更直观更方便。

表 8.1 与源文件有关的属性和变量

类别	变量或属性名	说明
目标属性	SOURCES	用于指定目标源文件的路径列表。以下命令会设置该属性且是常用方法： <code>add_executable()</code> <code>add_library()</code> <code>add_custom_target()</code> <code>target_sources()</code>
	INTERFACE_SOURCES	可以使用 <code>target_sources()</code> 命令使用 <code>PUBLIC</code> 和 <code>INTERFACE</code> 填充该属性。此属性只能使用绝对路径。需要注意的是，此属性不会被 <code>add_executable()</code> 、 <code>add_library()</code> 、 <code>add_custom_target()</code> 命令填充。
源文件属性	GENERATED	用于标识一个文件是否是生成文件(<code>generated file</code>)，即该文件是由外部生成的(比如由另一构建步骤生成的)或是由 CMake 本身执行生成的。其值为 <code>TRUE</code> 或 <code>FALSE</code> ，若被设置为 <code>TRUE</code> ，则会免除该文件的存在性或有效性检查

8.1.2 SOURCES 和 INTERFACE_SOURCE 目标属性

1、SOURCES 目标属性用于指定目标源文件的路径列表。以下命令会设置该属性且是常用方法：

```
add_executable()
add_library()
add_custom_target()
target_sources()
```

2、SOURCES 目标属性指定路径的规则如下：

- 若路径以生成器表达式指定，则应使用绝对路径，否则是未定义的行为。
- 若路径未被指定为绝对路径，则生成文件(`generated file`)的路径将被视为相对于目标的构建目录，具体见示例 8.2。
- 若路径不是以生成器表达式开头、不是绝对路径、也不是生成文件(`generated file`)，则按以下规则选择第一个匹配的路径(可简单理解为相对路径源文件的搜索规则)：
 - 如果指定路径下的文件相对于目标的源目录存在，则使用该文件

- 若 CMP0115(3.20 及以上版本)未设置为 NEW，则尝试将每个已知的源文件的扩展名追加到路径中，并检查相对于目标源目录是否存在。注：CMP0115 若为 OLD，则源文件可以不使用后缀，CMake 会自动添加；若为 NEW 则应指定后缀。
- 重复以上两步骤，但这次相对于目标的二进制目录(即构建目录)。也就是说，使用相对路径时，源文件也可以位于构建目录中。
- 注意，以上的决策是在生成时做出的，而不是在构建时。

3、INTERFACE_SOURCE 目标属性只能使用绝对路径，由于该属性以 INTERFACE 开始，因此，INTERFACE_SOURCE 和 SOURCE 是使用要求且是非链接使用要求，他们会按传递非链接使用要求的规则被传递。但要注意，与 SOURCE 不同的是，INTERFACE_SOURCE 属性只会被 target_sources()命令填充，不会被 add_executable()、add_library()、add_custom_target()填充。

示例 8.1：相对路径源文件的搜索规则

①、源文件准备(注意各源文件路径)

在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp(主源文件)
#include<iostream>
extern int b;
extern int c;
int main(){    std::cout<<b<<c<<std::endl;    return 0;}
```

在目录 g:/qt1/xx 中的 b.cpp 中编写如下代码

```
//g:/qt1/xx/b.cpp
int b=1;
```

在目录 g:/qt1/yy/aa 中的 c.cpp 中编写如下代码

```
//g:/qt1/yy/aa/c.cpp
int c=2;
```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXX)
add_executable(nn a.cpp xx/b.cpp)
set_property(TARGET nn APPEND PROPERTY SOURCES aa/c.cpp)
get_property(AA TARGET nn PROPERTY SOURCES)
message("AA=${AA}")
```

③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```
cmake -B yy
```

执行以上命令后输出，注意：从以下输出并不能判断各源文件具体的路径。

```
AA=a.cpp;xx/b.cpp;aa/c.cpp
```

本示例将在 g:/qt1/yy 目录下生成 CMake 的构建文件，所以，本示例源文件的相对路径将在当前源目录 g:/qt1 中查找，若未找到则会在构建目录 g:/qt1/yy 中查找，所以，本示例的源文件分别位于

```
g:/qt1/a.cpp
```

```
g:/qt1/xx/b.cpp
```

```
g:/qt1/yy/aa/c.cpp
```

8.1.3 GENERATED 源文件属性

- 1、GENERATED 源文件属性用于标识一个文件是否是生成文件(generated file)，即，该文件是由外部生成的(比如由另一构建步骤生成的)或是由 CMake 本身执行生成的。
- 2、GENERATED 属性可被设置为 TRUE 或 FALSE，这是一个全有或全无(all-or-nothing)的属性，若被设置为 TRUE，则不能再被删除或取消设置，并且会免除该文件的存在性或有效性检查。从 3.20 开始(CMP0118)，GENERATED 源文件属性在所有目录中可见。比如

```
add_executable(nn ee.cpp)
```

```
set_property(SOURCE ee.cpp PROPERTY GENERATED TRUE)
```

若文件 ee.cpp 不存在，也能成功使用 cmake 命令生成 makefile 文件，但使用 mingw32-make 等 make 命令执行会产生找不到 ee.cpp 文件的错误(这是编译器发出的错误)。若没有第二行代码，则执行 cmake 命令会产生 ee.cpp 不存在的错误(这是 CMake 发出的错误)。

- 3、以下情形创建的文件会将 GENERATED 属性标记为 TRUE：
 - 通过执行在构建期间运行的 add_custom_command()等命令创建的文件
 - 在构建期间运行的 add_custom_command()或 add_custom_target()命令的 BYPRODUCTS(副产品)之一。
 - 由 CMake AUTOGEN 操作创建的，如 AUTOMOC、AUTORCC 或者 AUTOUIIC
- 4、Makefile 生成器将在 make clean 期间删除 GENERATED 文件。

示例 8.2：生成文件的相对路径

①、源文件准备

在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp(主源文件)
```

```
#include<iostream>
```

```
int main(){      std::cout<<"A"<<std::endl;      return 0;}
```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.27)
```

```
project(xxxx)
```

```
add_executable(nn a.cpp)
```

```
set_property(SOURCE a.cpp PROPERTY GENERATED TRUE)      #将 a.cpp 设置为生成文件
```

③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```
cmake -B yy
```

```
cd yy      #转至构建目录。注：makefile 文件位于该目录
```

```
mingw32-make
```

```
nn.exe
```

以上示例构建于 g:/qt1/yy 目录，由由于 a.cpp 被指定为生成文件，所以，在执行 cmake 命令时无论在构

建目录中是否含有 a.cpp，都不会产生错误，但在执行 mingw32-make 等 make 命令编译源文件时，若 a.cpp 不在构建目录中，则会产生错误。实际上，观察 CMakeFiles\... 目录下的 build.make 文件，可找到如下编译命令(简化后)

```
cl.exe ... -c g:\qt1\yy\a.cpp
```

可见，在编译时实际指定的 a.cpp 的路径是 g:\qt1\yy，即，CMake 的构建目录。

8.1.4 target_sources()命令--->常规语法

1、target_sources()命令有两个变体，一个是常规的语法，另一个与创建文件集(FILE SET)有关。

2、target_sources()常规语法的形式为：

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

3、该命令用于向目标<target>添加源文件，各参数意义如下：

- <target>必须是由 add_executable()、add_library()或 add_custom_target()等命令创建的，且不能是别名(ALIAS)目标。
- PUBLIC 将填充<target>的 SOURCES 属性和 INTERFACE_SOURCES 属性
- PRIVATE 将填充<target>的 SOURCES 属性。由 add_custom_target()创建的目标只能具有 PRIVATE 参数。
- INTERFACE 将填充<target>的 INTERFACE_SOURCES 属性。
- 对同一<target>的重复调用，将按照调用的顺序追加

4、以下是各版本的更新规则

- 3.3 及以上版本允许使用 INTERFACE_SOURCES 导出目标
- 3.11 及以上版本允许在 IMPORTED 目标(导入目标)上设置 INTERFACE
- 3.13 版本将相对源文件路径解释为相对于当前源目录，即 CMAKE_CURRENT_SOURCE_DIR，另见 CMP0076。在以下情况下将源文件的相对路径转换为绝对路径：
 - 源文件被添加到目标的 INTERFACE_SOURCES 属性中。
 - 目标的 SOURCE_DIR 属性不同于 CMAKE_CURRENT_SOURCE_DIR。
 - 以生成器表达式开头的路径保持不变，使用绝对路径。

5、target_sources()命令的常规语法与目标属性 SOURCES 和 INTERFACE_SOURCES 的用法一致，只不过使用 target_sources()命令能更方便的设置这两个属性的值，请参阅 8.1.2 小节。由于 INTERFACE 属性是使用要求，所以，在这里列举一个有关源文件的传递性质

示例 8.3：源文件的可传递性

①、源文件准备

在目录 g:\qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp(主源文件)
#include<iostream>
int main(){      std::cout<<"A"<<b<<c<<d<<std::endl;      return 0;}
```

在目录 g:\qt1 中的 b.cpp、c.cpp、d.cpp 中分别编写如下代码

```
//g:/qt1/b.cpp
```

```

        int b=1;
//g:/qt1/c.cpp
        int c=2;
//g:/qt1/d.cpp
        int d=3

```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

#g:/qt1/CMakeLists.txt

```

cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj STATIC d.cpp)           #该命令只能设置 SOURCES 目标属性
target_sources(jj PUBLIC c.cpp)       #同时设置 SOURCES 和 INTERFACE_SOURCES 目标属性
                                       #这样，c.cpp 源文件便可以传递给其他依赖目标了
add_library(kk STATIC b.cpp)
target_link_libraries(kk PUBLIC jj)    #将 b.cpp.obj 和 c.cpp.obj 添加到 kk 的 lib 命令
add_executable(nn a.cpp)
target_link_libraries(nn kk)           #将 a.cpp.obj、c.cpp.obj、kk.lib、jj.lib
                                       #添加到 nn 的 link 命令

```

③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```

cmake .
mingw32-make
nn.exe

```

由于 SOURCES 和 INTERFACE_SOURCES 目标属性是非链接使用要求，因此，不需要依赖目标有链接命令，所以，源文件 c.cpp 对应的对象文件 c.cpp.obj 会被添加到 kk 的 lib 命令，同理，也会 c.cpp.obj 添加到 nn 的 link 命令(这里是将 c.cpp 作为非链接使用要求传递的)，可以分别在文件夹 CMakeFiles\kk.dir 和 CMakeFiles\nn.dir 的 link.txt 和 objects1.rsp 文件中找到将 c.cpp.obj 添加到 kk 和 nn 相应命令中的代码。

8.1.5 target_sources()命令--->文件集

其语法为：

```

target_sources(<target>
    [<INTERFACE|PUBLIC|PRIVATE>
        [FILE_SET <set>
            [TYPE <type>]
            [BASE_DIRS <dirs>...]
            [FILES <files>...]
        ]...
    ]...)

```

- 1、将文件集<set>添加到目标<target>，或将文件<files>添加到现有的文件集<set>，一个目标<targe>可以有零个或多个文件集。

- 2、文件集就是文件的集合，文件集将会在 `install()`命令中有使用，所以文件集的具体使用示例详见 `install()`命令。
- 3、出于 IDE 集成的目的，PRIVATE 或 PUBLIC 文件集中的文件被标记为源文件。此外，HEADERS 文件集中的文件将其 `HEADER_FILE_ONLY` 源文件属性设置为 TRUE。INTERFACE 或 PUBLIC 文件集中的文件可以使用 `install(TARGETS)`命令安装，并使用 `install(EXPORT)`和 `export()`命令导出。
- 4、默认文件集以其类型(即 TYPE 参数)命名。
- 5、各参数意义如下：
 - **FILE_SET <set>**
要创建或添加的文件集的名称，只能包含字母、数字和下划线。以大写字母开头的文件集保留给 CMake 预定义的内置文件集，所有其他集合名称不能以大写字母或下划线开头。
 - **TYPE <type>**
指定文件集的类型，只能指定以下类型，指定其他类型是错误的。若文件集的名称是类型名之一，则此参数可以省略。对于其他文件集，TYPE 是必须的。
 - **HEADERS**
准备通过 `#include` 使用的源文件
 - **CXX_MODULES**
适用于 3.28 及以上版本。包含 C++接口模块或分区单元(partition unit)的源文件(即使用 `export` 关键字的源)。除导入库目标外，该文件集类型可能没有 INTERFACE 作用域。
 - **BASE_DIRS <dirs> ...**
指定文件集的基本目录，任何相对路径都被视为相对于当前源目录(即 `CMAKE_CURRENT_SOURCE_DIR`)。若第一次创建文件集时未指定 `BASE_DIRS`，则添加 `CMAKE_CURRENT_SOURCE_DIR` 的值，此参数支持生成器表达式。文件集的两个基本目录不能互为子目录。
 - **FILES <files>...**
指定添加到文件集中的文件，每个文件必须位于其中一个基本目录中，或其中一个基本目录的子目录中，此参数支持生成器表达式。若指定了相对路径，则被认为是相对于 `CMAKE_CURRENT_SOURCE_DIR`，一个例外是以 `$<` 开始(即生成器表达式)的路径，在对生成器表达式求值后，这些路径被视为相对于目标源目录的路径。
- 6、`target_sources(FILE_SET)`命令会设置以下目标属性
 - 对于 HEADERS 类型的文件集：
 - `HEADER_SETS`
 - `INTERFACE_HEADER_SETS`
 - `HEADER_SET`
 - `HEADER_SET_<NAME>`
 - `HEADER_DIRS`
 - `HEADER_DIRS_<NAME>`
 - 对于 CXX_MODULES 类型的文件集：
 - `CXX_MODULE_SETS`
 - `INTERFACE_CXX_MODULE_SETS`
 - `CXX_MODULE_SET`
 - `CXX_MODULE_SET_<NAME>`
 - `CXX_MODULE_DIRS`
 - `CXX_MODULE_DIRS_<NAME>`

7、与包含目录有关的目标属性也可通过该命令修改，如下：

- INCLUDE_DIRECTORIES
若 TYPE 是 HEADERS，且文件集的作用域是 PRIVATE 或 PUBLIC，则文件集的所有 BASE_DIRS 都包装在\$<BUILD_INTEFACE>中并追加到此属性。
- INTERFACE_INCLUDE_DIRECTORIES
若 TYPE 是 HEADERS，且文件集的作用域是 INTERFACE 或 PUBLIC，则文件集的所有 BASE_DIRS 都包装在\$<BUILD_INTEFACE>中并追加到此属性。

以上规则意味着，可通过文件集来为库设置头文件包含目录(-I 参数)。注意：INCLUDE_DIRECTORIES 和 INTERFACE_INCLUDE_DIRECTORIES 是使用要求，会按使用要求的规则传递。

示例 8.4：使用文件集向库添加包含目录(-I 参数)

①、源文件准备

在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp(主源文件)
#include<iostream>
#include "g.h"
int main(){      std::cout<<"A"<<b<<c<<d<<std::endl;      return 0;}
```

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
int b=1;
```

在目录 g:/qt1 中的 c.cpp 中编写如下代码

```
//g:/qt1/b.cpp
int c=2;
```

在目录 g:/qt1/xx 中的 g.h 中编写如下代码

```
//g:/qt1/xx/g.h(头文件)
int d=3;
```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)

add_library(jj c.cpp)
add_library(kk b.cpp)
target_sources(jj PUBLIC
    FILE_SET s
    TYPE HEADERS
    BASE_DIRS xx          #xx 会自动成为库 jj 的包含目录
    #BASE_DIRS xx xx/y    #错误，基本目录不能互为子目录
    FILES xx/g.h
    #FILES g.h )          #错误，文件必须位于基本目录或其子目录中，g.h 位于 g:/qt1，并不在 g:/qt1/xx 中
```

```

add_executable(nn a.cpp)
target_link_libraries(kk jj)
target_link_libraries(nn kk)

get_property(AA1 TARGET jj PROPERTY INCLUDE_DIRECTORIES )
get_property(AA2 TARGET jj PROPERTY INTERFACE_INCLUDE_DIRECTORIES )
message(AA1=${AA1})
message(AA2=${AA2})

```

③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```

cmake .
mingw32-make
nn.exe

```

执行 cmake .后，会输出以下内容

```

AA1=$<BUILD_INTERFACE:G:/qt1/xx>
AA2=$<BUILD_INTERFACE:G:/qt1/xx>

```

由输出可见，由于文件集的 TYPE 是 HEADERS，且作用域是 PUBLIC，因此，同时为 INCLUDE_DIRECTORIES 和 INTERFACE_INCLUDE_DIRECTORIES 目标属性设置了值(即指定了-I 参数，或者说设置了包含目录)。

在 G:\qt1\CMakeFiles\jj.dir、G:\qt1\CMakeFiles\kk.dir、G:\qt1\CMakeFiles\nn.dir 目录下的 flags.make 文件中，都可以找到以下内容，这说明-I 参数被成功传递。

```
CXX_INCLUDES = -IG:/qt1/xx
```

8.1.6 aux_source_directory()命令

其语法为：

```
aux_source_directory(<dir> <variable>)
```

该命令用于查找目录<dir>中的所有源文件名称，并将其名称列表存储在变量<variable>中。这个命令可以快速的一次性将整个目录中的源文件添加到项目中。以下是其使用方法：

```

aux_source_directory(g:/qt1/yy xx)      #将目录 g:/qt1/yy 中的所有源文件名保存到 xx 中
add_executable(nn ${xx})                #将变量 xx 中保存的源文件名添加到目标 nn 中

```

8.2 添加头文件包含目录(-I 编译参数)

8.2.1 总览

- 1、可以使用表 8.2 所示的属性或/和命令来设置编译器的包含文件(即头文件)搜索目录，即设置包含头文件所在的目录，本文将这种目录称为包含目录，也即设置-I (大写字母 i)编译参数。
- 2、由于 INTERFACE_INCLUDE_DIRECTORIES 属性以 INTERFACE 开始，因此，INCLUDE_DIRECTORIES 和 INTERFACE_INCLUDE_DIRECTORIES 是使用要求且是非链接使用要

求，他们会按传递非链接使用要求的规则被传递。但要注意，与 INCLUDE_DIRECTORIES 不同的是，INTERFACE_INCLUDE_DIRECTORIES 属性只会被 target_include_directories()命令填充，不会被 include_directories()命令填充。

表 8.2 设置包含目录的属性 (-I 编译参数)

类别	属性名	说明
目标属性	INCLUDE_DIRECTORIES	为目标指定包含目录，即添加-I(大写 i)编译参数，此属性不应使用相对路径。可使用 target_include_directories()和 include_directories()命令填充此属性。其初始值从目录属性 INCLUDE_DIRECTORIES 获取。
	INTERFACE_INCLUDE_DIRECTORIES	为目标指定包含目录，即添加-I(大写 i)编译参数。可使用 target_include_directories()命令填充此属性。此属性设置的值可以被传递，其余与 INCLUDE_DIRECTORIES 目标属性相同。
	INTERFAINTERFACE_SYSTEM_INCLUDE_DIRECTORIES	将指定的目录设置为系统包含目录。应使用带有 SYSTEM 关键字的 target_include_directories()命令设置该属性，而不是直接设置该属性
目录属性	INCLUDE_DIRECTORIES	可使用 include_directories()命令填充此属性。若父目录有初始值，则从父目录获取初始值。使用此属性可以同时为多个目标指定相同的包含目录，此属性不应使用相对路径。该属性也用于设置同名的目标属性的初始值，需要注意的是，若使用 set_property()或 set_directory_properties()命令更新该属性时，不会同步更新相应的目标属性的值，即该属性可以初始化目标属性，但不会更新目标属性。
源文件属性	INCLUDE_DIRECTORIES	为源文件指定头文件的目录(以分号分隔)，不应使用相对路径。由于技术限制，该属性指定的目录优先于同名的目标属性定义的目录
	HEADER_FILE_ONLY	将某文件标识为仅仅是头文件，这意味着，若此属性为 ON，则不会对指定的源文件进行编译(比如，不会生成该文件对应的.obj 文件)

8.2.2 target_include_directories()命令

其语法如下：

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

该命令用于向目标<target>添加包含头文件的目录<item>，即添加包含目录，也即设置-I (大写字母 i)编译参数。3.11 及以上版本允许在导入目标上设置 INTERFACE 参数。各参数意义如下：

1、<target>

<target>必须是由 add_executable()或 add_library()等命令创建的，且不能是别名目标。

2、AFTER 或 BEFORE 参数

AFTER 和 BEFORE 用于指示是将目录追加到末尾还是前置到开头。

3、SYSTEM 参数

该参数表示将指定的目录设置为系统包含目录，这可能会导致编译器抑制(suppress)警告。不管指定的

顺序如何，系统包含目录都会在普通包含目录之后搜索。若 SYSTEM 与 PUBLIC 或 INTERFACE 一起被指定，则目标属性 INTERFAINTERFACE_SYSTEM_INCLUDE_DIRECTORIES 将被此时指定的目录填充，应使用带有 SYSTEM 关键字的 target_include_directories() 命令设置该属性，而不是直接设置该属性，因为直接设置该属性不会使这些目录在编译期间被使用。

4、PUBLIC 参数填充 INCLUDE_DIRECTORIES 和 INTERFACE_INCLUDE_DIRECTORIES 目标属性。

5、PRIVATE 参数将填充 INCLUDE_DIRECTORIES 目标属性。

6、INTERFACE 参数将填充 INTERFACE_INCLUDE_DIRECTORIES 目标属性

7、items

该参数表示指定的包含目录，可以是绝对路径或相对路径，相对路径被解释为相对于当前源目录，即变量 CMAKE_CURRENT_SOURCE_DIR 的值，并将在存储到相应的目标属性中之前转换为绝对路径。若路径以生成器表达式开始，则始终被假定为绝对路径(但有一个例外)，在使用 \$<BUILD_INTERFACE:...> 生成器表达式时不应使用相对路径，因为它不会被转换为绝对路径，但 \$<INSTALL_INTERFACE:...> 生成器表达式可以使用相对路径，并被解释为相对于 installation 前缀的绝对路径。

8.2.3 include_directories() 命令

其语法为：

```
include_directories([AFTER | BEFORE] [SYSTEM] dir1 [dir2 ...])
```

- 1、该命令用于向构建中添加包含头文件的目录，即添加包含目录，也即设置 -I (大写字母 i) 编译参数。包含目录被添加到当前 CMakeLists 文件的 INCLUDE_DIRECTORIES 目录属性中，同时也被添加到当前 CMakeLists 文件中每个目标的 INCLUDE_DIRECTORIES 目标属性中。使用该命令可以为同一目录下的多个目标同时添加包含目录。
- 2、相对路径被解释为相对于当前源目录。
- 3、默认情况下，目录被追加到当前目录列表的末尾，可以通过将变量 CMAKE_INCLUDE_DIRECTORIES_BEFORE 设置为 ON 来禁止这种行为，或显示地使用 AFTER 或 BEFORE 参数选择是将目录追加到末尾还是前置到开头。
CMAKE_INCLUDE_DIRECTORIES_BEFORE 变量可以控制所有的 include_directories() 命令的行为，而使用 AFTER 或 BEFORE 参数仅能控制当前正在使用的 include_directories() 命令的行为。
CMAKE_INCLUDE_DIRECTORIES_BEFORE 只对 include_directories() 命令有效，对 target_include_directories() 命令无效。
- 5、SYSTEM 参数表示将指定的目录设置为某些平台上的系统包含目录，这可能会导致编译器抑制 (suppress) 警告。不管指定的顺序如何，系统包含目录都会在普通包含目录之后搜索。

示例 8.5：使用 INCLUDE_DIRECTORIES 目标属性为单个目标指定包含目录(-I 编译参数)

①、源文件准备

在目录 g:/qt1 中的 g.cpp 中编写如下代码

```
//g:/qt1/g.cpp(源文件)
```

```
#include<iostream>
#include "w.h"
using namespace std;
int main(){
    #ifdef XX
```

```

        cout<<"XX="<<aa<<endl;
    #endif
    #ifdef YY
        cout<<"YY="<<bb<<endl;
    #endif
    cout<<"EE"<<endl;
    return 0;}

```

在目录 `g:/qt1/xx` 中的 `w.h` 中编写如下代码

//g:/qt1/xx/w.h (头文件)

```

#define XX
int aa=1;

```

在目录 `g:/qt1/yy` 中的 `w.h` 中编写如下代码

//g:/qt1/yy/w.h (头文件)

```

#define YY
int bb=2;

```

②、在目录 `g:/qt1` 中的 `CMakeLists.txt` 中编写如下代码

#g:/qt1/CMakeLists.txt

```

cmake_minimum_required(VERSION 3.27)
project(XXXX)
add_executable(nn g.cpp)
#注意：以下命令不能使用相对路径
set_property(TARGET nn PROPERTY INCLUDE_DIRECTORIES g:/qt1/xx)
#将以下设置的目录追加于上一命令指定的目录之前
target_include_directories(nn BEFORE PUBLIC yy)

#以下的所有代码都为测试语句
get_property(AA1 TARGET nn PROPERTY INCLUDE_DIRECTORIES)
get_property(AA2 TARGET nn PROPERTY INTERFACE_INCLUDE_DIRECTORIES)
get_property(AA3 TARGET nn PROPERTY INTERFACE_SYSTEM_INCLUDE_DIRECTORIES)
get_property(AA4 DIRECTORY PROPERTY INCLUDE_DIRECTORIES)
message("AA1=${AA1}")           #输出 AA1=G:/qt1/yy;g:/qt1/xx
message("AA2=${AA2}")           #输出 G:/qt1/yy
message("AA3=${AA3}")           #输出 AA3=
message("AA4=${AA4}")           #输出 AA4=

```

③、在 `CMD` 中转至 `g:/qt1` 并输入以下命令，以验证结果

```

cmake .
mingw32-make
nn.exe

```

从执行 `nn` 后输出的 `YY=2` 可以看到，`g.cpp` 中的代码使用了来自 `g:/qt1/yy` 中的头文件 `w.h`，而未使用 `g:/qt/xx` 下的 `w.h`，这是因为 `yy` 目录位于 `xx` 目录之前。从执行 `cmake .` 命令之后输出的结果可以看到，`target_include_directories()` 命令指定的目录被同时设置到了目录属性

INTERFACE_INCLUDE_DIRECTORIES 和 INCLUDE_DIRECTORIES, 而 set_property()命令指定的目录只设置了目录属性 INCLUDE_DIRECTORIES。

④、查构建文件的内容

查看 G:\qt1\CMakeFiles\nn.dir\flags.make 文件, 可在其中找到以下文本

```
CXX_INCLUDES = -Ig:\qt1\yy -Ig:\qt1\xx
```

这说明本示例向 cl 命令添加了两个 -I 参数, 并且使用 target_include_directories()命令添加的目录位于 set_property()命令指定的目录之前。

示例 8.6: 使用 INCLUDE_DIRECTORIES 目录属性同时为多个目标指定相同的包含目录(-I 选项)

①、源文件准备

将前一示例 g.cpp 中的内容复制到 h.cpp 中, 且把 g.cpp 和 h.cpp 放于 g:/qt1 目录中。两个头文件 w.h 的位置和内容保持不变。

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.27)
project(XXXX)
#以下命令需位于需要设置的目标的 add_executable()命令之前, 且不能使用相对路径
set_property(DIRECTORY PROPERTY INCLUDE_DIRECTORIES g:/qt1/xx)
add_executable(nn g.cpp)    #将包含 g:/qt1/xx 下的头文件
add_executable(mm h.cpp)    #将包含 g:/qt1/xx 下的头文件
include_directories(yy)     #该命令不需要放于 add_executable()命令之前, 且可以使用相对路径
```

③、在 CMD 中转至 g:/qt1 并输入以下命令, 以验证结果

```
cmake .
mingw32-make
nn.exe
mm.exe
```

从执行 nn.exe 和 mm.exe 后输出的 XX=1 可以看到, 目标 nn 和 mm 都包含了 g:/qt1/xx 目录下的 w.h 头文件, 这是因为 xx 目录位于 yy 目录之前

④、查构建文件的内容

查看 G:\qt1\CMakeFiles\nn.dir 和 G:\qt1\CMakeFiles\mm.dir 目录下的 flags.make 文件, 在两文件中都可找到以下内容,

```
CXX_INCLUDES = -Ig:\qt1\xx -IG:\qt1\yy
```

这说明本示例分别向目标 nn 和 mm 的 cl 命令添加了两个 -I 参数

示例 8.7: SYSTEM 参数

①、源文件准备。使用示例 8.4 的源文件和头文件以及目录。

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.27)
```

```

project(xxxx)
add_executable(nn g.cpp)
target_include_directories(nn SYSTEM PUBLIC yy)           #指定 SYSTEM 选项
#以下所有代码都是测试语句
get_property(AA1 TARGET nn PROPERTY INCLUDE_DIRECTORIES)
get_property(AA2 TARGET nn PROPERTY INTERFACE_INCLUDE_DIRECTORIES)
get_property(AA3 TARGET nn PROPERTY INTERFACE_SYSTEM_INCLUDE_DIRECTORIES)
get_property(AA4 DIRECTORY PROPERTY INCLUDE_DIRECTORIES)
message(AA1=${AA1})      #输出 AA1=G:/qt1/yy
message(AA2=${AA2})      #输出 AA2=G:/qt1/yy
message(AA3=${AA3})      #输出 AA3=G:/qt1/yy
message(AA4=${AA4})      #AA4=

```

③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```

cmake .
mingw32-make
nn.exe

```

从执行 nn.exe 后输出的 YY=2 可以看到，目标 nn 包含了 g:/qt1/yy 目录下的 w.h 头文件。从执行 cmake .命令后的输出可以看到，使用 SYSTEM 参数的 target_include_directories()命令指定的目录，同时填充了 INCLUDE_DIRECTORIES、INTERFACE_INCLUDE_DIRECTORIES、INTERFACE_SYSTEM_INCLUDE_DIRECTORIES 三个属性

④、查构建文件的内容

查看 G:\qt1\CMakeFiles\nn.dir 目录下的 flags.make 文件，在该文件中可找到以下内容：

```
CXX_INCLUDES = -external:IG:\qt1\yy -external:W0
```

其中参数-external:I 表示定义外部头文件的目录，然后可以使用参数“-external:Wn”控制外部头文件的警告级别为 n (0~4)，其中 W0 表示关闭警告。注：若使用 g++命令将导至添加-isystem 参数。从这里可以看到 SYSTEM 参数具体添加的编译参数。

8.3 添加链接文件的库目录

8.3.1 总览

- 1、可以使用表 8.3 所示的属性或/和命令来设置链接命令的库目录，即设置库文件(如.lib 文件)所在的目录，本文将该目录称为搜索库目录。以 VC++命令为例，设置搜索库目录就是设置 link 命令的“-LIBPATH”链接参数。
- 2、由于 INTERFACE_LINK_DIRECTORIES 属性以 INTERFACE 开始并且含有 LINK 字符，因此，LINK_DIRECTORIES 和 INTERFACE_LINK_DIRECTORIES 是使用要求且是链接使用要求，他们会按传递链接使用要求的规则被传递。

表 8.3 设置库文件目录(-LIBPATH 链接参数)

类别	属性名	说明
目标属性	LINK_DIRECTORIES	可使用 <code>target_link_directories()</code> 命令填充此属性。表示链接命令的搜索库目录(以分号分隔)，即设置 <code>link</code> 命令的 <code>- LIBPATH</code> 参数。该属性在创建目标时由 <code>LINK_DIRECTORIES</code> 目录属性初始化。
	INTERFACE_LINK_DIRECTORIES	可使用 <code>target_link_directories()</code> 命令用 <code>PUBLIC</code> 和 <code>INTERFACE</code> 关键字填充这个属性。此属性设置的值可以被传递，其余与 <code>LINK_DIRECTORIES</code> 目标属性相同。
目录属性	LINK_DIRECTORIES	可使用 <code>link_directories()</code> 命令填充。此属性在目录范围内设置目标的搜索库目录(<code>- LIBPATH</code> 参数)，目录以分号分隔，也就是说，可以使用此属性对多个目标设置相同的库目录。该属性从父目录(如果有)获取初始值。该属性用于初始化 <code>LINK_DIRECTORIES</code> 目标属性。

8.3.2 target_link_directories()命令

其语法如下：

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

- 1、该命令用于指定链接命令在链接目标<target>时的搜索库目录 `items`，以 `VC++` 命令为例，其结果就是设置 `link` 命令的 `“- LIBPATH”` 参数。此命令很少必须使用，所以，在有替代的情况下，应尽量避免使用。
- 2、各参数意义如下：
 - <target>必须是由 `add_executable()` 或 `add_library()` 等命令创建的，且不能是别名(`ALIAS`)目标。对同一个目标的重复调用，按照调用的顺序追加 `items`。
 - `items` 是目录名称，可以是绝对路径或相对路径，相对路径相对于当前源目录，若有必要，在将其添加到相关属性之前将其转换为绝对路径。应尽量使用绝对路径，以确保可以始终正确链接。
 - `BEFORE` 参数表示将相关内容追加到相关属性之前，而不是追加到末尾。
 - `PUBLIC` 参数填充 `INTERFACE_LINK_DIRECTORIES` 和 `LINK_DIRECTORIES` 目标属性。
 - `PRIVATE` 参数将填充 `LINK_DIRECTORIES` 目标属性。
 - `INTERFACE` 参数将填充 `INTERFACE_LINK_DIRECTORIES` 目标属性。导入目标只支持该参数。

- 3、此命令用法比较简单，下面列举一个简单的示例

```
add_library(jj SHARED c.cpp)
target_link_directories(jj PUBLIC yy)
```

假设当前源目录为 `g:/qt1`，则以上命令将向 `jj` 的 `link` 命令添加参数 `“-LIBPATH:G:/qt1”`，可在 `CMakeFiles/jj.dir` 文件夹中的 `link.txt` 文件中找到该参数。

示例 8.8: INTERFACE_LINK_DIRECTORIES 及其对应的非 INTERFACE 属性是链接使用要求

- ①、源文件准备。读者自行准备合适的源文件。

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj STATIC c.cpp)           #jj 库
target_link_directories(jj PUBLIC yy)  #jj 的链接使用要求
target_compile_options(jj PUBLIC -DEE=3) #jj 的非链接使用要求
add_library(kk STATIC b.cpp)           #kk 库
target_link_libraries(kk PRIVATE jj)    #PRIVATE 将阻止 jj 的非链接使用要求的传递，
                                         #但 jj 的链接使用要求不会被阻止
add_executable(nn a.cpp)                #nn 可执行文件
target_link_libraries(nn PUBLIC kk)      #向 link 命令添加-LIBPATH 参数及 kk.lib 和 jj.lib
```

INTERFACE_LINK_DIRECTORIES 和 LINK_DIRECTORIES 属性是链接使用要求，target_link_libraries()命令的 PRIVATE 选项不会阻止链接使用要求的传递。由于目标 kk 没有链接命令，所以，目标 kk 未添加-LIBPATH 参数。可在 CMakeFiles\nn.dir 文件夹中的 link.txt 文件中找到以下内容：

```
link.exe ..... -LIBPATH:G:\qt1\yy kk.lib jj.lib .....
```

8.3.3 link_directories ()命令

其语法如下：

```
link_directories([AFTER|BEFORE] directory1 [directory2 ...])
```

- 1、该命令用于指定链接命令在链接时的搜索库目录 directory，以 VC++命令为例，其结果就是设置 link 命令的“-LIBPATH”参数。
- 2、此命令将在目录范围内设置库目录，也就是说，可以使用此命令对多个目标设置相同的库目录。但要注意，此命令需放于需要设置的目标的前面，在此命令之后的目标都将添加由该命令指定的库目录。
- 3、此命令很少必须使用，所以，在有替代的情况下，应尽量避免使用，若必须要添加库目录，最好是使用 target_link_directories()命令。
- 4、搜索库目录可以是绝对路径或相对路径，相对路径相对于当前源目录，根据需要，将相对路径转换为绝对路径。
- 5、指定的搜索库目录被添加到当前 CMakeLists.txt 文件的 LINK_DIRECTORIES 目录属性中。
- 6、默认情况下，指定的目录被追加到末尾。可通过将变量 CMAKE_LINK_DIRECTORIES_BEFORE 设置为 ON 来更改此默认行为，也可显式地使用 AFTER 将目录追加到末尾，或指定 BEFORE 将目录前置于开头。

8.3.4 link_libraries()命令

其语法如下：

```
link_libraries([item1 [item2 [...]]
               [[debug|optimized|general] <item>] ...])
```

只要可能，应优先使用 `target_link_libraries()` 命令。该命令表示，将指定的库 `item` 链接到在此命令之后出现的 `add_executable()` 或 `add_library()` 等命令的所有目标中(注意：需要有链接命令)，也就是说，该命令需位于 `add_executable()` 或 `add_library()` 之前。`debug`、`optimized`、`general` 是指的编译模式，比如，`debug` 是调试模式。下面列举一个用法示例：

```
add_library(jj STATIC c.cpp)
link_libraries(jj)           #将 jj.lib 添加到之后出现的目标中
add_library(kk STATIC c.cpp) #因为 kk 没有 link 命令，所以不添加 jj.lib
add_library(nn SHARED e.cpp) #将 jj.lib 添加到 nn 的 link 命令
add_executable(mm d.cpp)     #将 jj.lib 添加到 mm 的 link 命令
```

8.4 include()和 add_dependencies()命令

8.4.1 include()命令

其语法为：

```
include(<file|module> [OPTIONAL] [RESULT_VARIABLE <var>] [NO_POLICY_SCOPE])
```

- 1、该命令表示，将指定的文件或模块加载到当前项目并运行其代码，并将指定的文件或模块的完整路径保存在 `RESULT_VARIABLE` 参数指定的变量中，若加载失败，则设置为 `NOTFOUND`。
- 2、`OPTIONAL` 参数表示，若指定的文件不存在也不会产生错误。
- 3、若指定的是模块(即 `.cmake` 文件)而不是文件，则首先在 `CMAKE_MODULE_PATH` 变量指定的路径中搜索名为 `*.cmake` 的文件，然后在 CMake 模块目录中搜索。有一个例外：如果调用 `include()` 的文件本身位于 CMake 内置模块目录中，则首先搜索 CMake 内置模块目录，然后搜索 `CMAKE_MODULE_PATH`。
- 4、若指定的模块文件位于 `CMAKE_MODULE_PATH` 变量指定的路径或 CMake 模块目录中，则可以只指定文件名而不指定后缀 `.cmake` 和路径。注意：若指定了路径，则不能省略 `.cmake` 后缀。
- 5、CMake 模块目录如下(仅供参考)：

<CMake 安装目录>/share/cmake-3.27/Modules

示例 8.9: include()的使用

①、文件准备。

在 `g:/qt1/yy` 中的 `tt.cmake` 文件中编写如下代码

```
#g:/qt1/yy/tt.cmake
message(XXXXXXX)
```

②、在目录 `g:/qt1` 中的 `CMakeLists.txt` 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(yyyy)
#设置模块搜索路径
set(CMAKE_MODULE_PATH g:/qt1/yy)
```



```
#tt.cmake 位于 CMAKE_MODULE_PATH 指定的路径中，可以省略后缀和路径
include(tt RESULT_VARIABLE AA)
message(AA=${AA})
```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
```

执行以上命令后会输出以下内容

```
XXXXXXXXXX
```

```
AA=G:/qt1/yy/tt.cmake
```

8.4.2 add_dependencies()命令

其语法为：

```
add_dependencies(<target> [<target-dependency>]...)
```

- 1、以上命令表示使顶级<target>依赖于其他顶级目标，以确保它们在<target>之前构建。注意：<target>需要是顶级目标，<target-dependency>也需要是顶级目标
- 2、顶级目标是由 add_executable()、add_library()或 add_custom_target()命令之一创建的目标。
- 3、添加到导入的目标或接口库的依赖关系将在其位置传递，因为目标本身不会构建。
- 4、3.3 及以上版本允许向接口库添加依赖关系。

第 9 章 使用 install()命令安装文件

本章需要对 dll 文件有所了解，在 5.2.1 小节讲解了共享库和模块库的区别

9.1 使用变量或目标属性将程序文件输出到指定目录

9.1.1 输出工件(Output Artifacts)

- 1、CMake 把由 CMake 目标构建的真实文件称为输出工件(Output Artifacts)，即输出工件是指的构建的真实存在的程序文件(即最终目标)。输出工件在 dll 平台和非 dll 平台之间有一些不同，dll 平台是指包括 Cygwin 在内的所有基于 windows 系统的平台。
- 2、由于构建后生成的程序文件(即输出工件)类型比较多，如.obj、.lib、.so、.dll、.a、.exe 等，所以，CMake 对这些文件进行了分类。CMake 可以构建三种类型的输出工件(artifact)：运行时输出工件、库输出工件和存档(archive)输出工件，表 XXX 和图 2-XXX 是对输出工件的汇总。
- 3、运行时输出工件(Runtime Output Artifact)
 - 由 add_executable()命令构建的可执行文件，如.exe 文件。
 - 在 DLL 平台上，由 add_library(SHARED)命令构建的共享库目标，如.dll 文件。
 - 运行时输出工件的位置和名称可使用 RUNTIME_OUTPUT_DIRECTORY 和 RUNTIME_OUTPUT_NAME 目标属性进行控制。
- 4、库输出工件(Library Output Artifact):
 - 由 add_library(MODULE)命令构建的可加载模块文件，如.dll，.so 文件等。
 - 在非 DLL 平台上，add_library(SHARED)命令构建的共享库文件，如.so，.dylib 等文件。
 - 库输出工件的位置和名称可使用 LIBRARY_OUTPUT_DIRECTORY 和 LIBRARY_OUTPUT_NAME 目标属性控制。
- 5、存档(archive)输出工件(Archive Output Artifact):
 - 由 add_library(STATIC)命令构建的静态库文件，如.lib 或.a 文件。
 - 在 DLL 平台上，由 add_library(SHARED)命令构建的共享库目标的导入库文件，如.lib 文件，注意：dll 的导入库文件与静态库文件拥有相同的后缀.lib，但二者是不同的文件类型。
 - 在 DLL 平台上，当设置了目标属性 ENABLE_EXPORTS 时，由 add_executable()命令构建的可执行目标的导入库文件，如.lib 文件。ENABLE_EXPORTS 目标属性的使用见示例 9.7
 - 在 AIX 上，当设置了目标属性 ENABLE_EXPORTS 时，由 add_executable()命令构建的可执行目标的链接器导入文件，如.imp 文件。
 - 在 macOS 上，当设置了目标属性 ENABLE_EXPORTS 时，由 add_library(SHARED)命令构建的共享库目标的链接器导入文件，如.tbd 文件。
 - 存档输出工件的位置和名称可使用目标属性 ARCHIVE_OUTPUT_DIRECTORY 和 ARCHIVE_OUTPUT_NAME 控制

表 9.1 CMake 的输出工件

类别	条件	文件后缀	使用的 CMake 命令	相关属性
运行时输出工件	无	.exe	add_executable()	RUNTIME_OUTPUT_DIRECTORY
	DLL 平台	.dll(共享库)	add_library(SHARED)	RUNTIME_OUTPUT_NAME
库输出工件	无	.dll(模块库)、.so	add_library(MODULE)	LIBRARY_OUTPUT_DIRECTORY
	非 DLL 平台	.so、.dylib	add_library(SHARED)	LIBRARY_OUTPUT_NAME
存档输出工件	无	.lib(静态库)、.a	add_library(STATIC)	ARCHIVE_OUTPUT_DIRECTORY ARCHIVE_OUTPUT_NAME
	DLL 平台	.lib(导入库)	add_library(SHARED)	
	在 DLL 平台、AIX、macOS 上，设置目标属性 ENABLE_EXPORTS(见示例 9.7)	.lib(导入库)	add_executable()	
		.imp	add_executable()	
		.tbd	add_library(SHARED)	



图 9.1 CMake 输出工件

9.1.2 将程序文件输出到指定目录的方法

- 1、将程序文件(输出工件)输出到指定目录的方法有两种，一种是使用表 9.2 的目标属性或对应的变量，一种是使用 install()命令，他们的区别在于：
 - install()命令需要单独使用类似 make install 的 CMD 命令(比如 mingw32-make install)单独安装，否则，install()命令将无效。详见下文。
 - 默认情况下 install()命令是将构建的目标文件复制一份到指定的目录，或通过设置 CMAKE_INSTALL_MODE 环境变量，在指定目录为目标文件创建一个符号链接，但无论如何，使用 install()命令构建的原始目标文件的位置并未改变，而使用表 9.2 中的目标属性或变量，会直接将原始目标文件移动到指定的目录。
 - 使用表 9.2 中的变量可一次为多个目标设置相同的输出路径，但，使用 install()和表 9.2 中的目标属性一次只能为一个目标设置输出路径。
- 2、表 9.3 所示的目标属性可为输出工件指定名称。
- 3、使用变量或目标属性将输出工件输出到指定目录的使用方法比较简单，从略，但要区分构建目录、源目录、输出工件目录三者的区别，这方面的示例可以参数示例 4.2

表 9.2 使用变量或目标属性将输出工件输出到指定目录

类别	属性名	说明
目标属性	ARCHIVE_OUTPUT_DIRECTORY	为构建的存档(Archive)目标文件指定目录，多配置生成器(如 VS, Xcode 等)将每个配置的子目录追加到指定的目录，除非使用生成器表达式。如果设置了 CMAKE_ARCHIVE_OUTPUT_DIRECTORY 变量，则该属性由该变量的值初始化。
	ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>	ARCHIVE_OUTPUT_DIRECTORY 目标属性的单配置版本，但多配置生成器(VS, Xcode)不会将单配置子目录追加到指定的目录。如果设置了 CMAKE_ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>变量，则该属性由该变量的值初始化。
	LIBRARY_OUTPUT_DIRECTORY	为构建的库(Library)目标文件指定目录，多配置生成器(如 VS, Xcode 等)将每个配置的子目录追加到指定的目录，除非使用生成器表达式。如果设置了 CMAKE_LIBRARY_OUTPUT_DIRECTORY 变量，则该属性由该变量的值初始化。
	LIBRARY_OUTPUT_DIRECTORY_<CONFIG>	LIBRARY_OUTPUT_DIRECTORY 目标属性的单配置版本，但多配置生成器(VS, Xcode)不会将单配置子目录追加到指定的目录中。如果设置了 CMAKE_LIBRARY_OUTPUT_DIRECTORY_<CONFIG>变量，则该属性由该变量的值初始化。
	RUNTIME_OUTPUT_DIRECTORY	为构建的运行时(Runtime)目标文件指定目录，多配置生成器(如 VS, Xcode 等)将每个配置的子目录追加到指定的目录，除非使用生成器表达式。如果设置了 CMAKE_RUNTIME_OUTPUT_DIRECTORY 变量，则该属性由该变量的值初始化。
	RUNTIME_OUTPUT_DIRECTORY_<CONFIG>	RUNTIME_OUTPUT_DIRECTORY 目标属性的单配置版本，但多配置生成器(VS, Xcode)不会将单配置子目录追加到指定的目录中。如果设置了 CMAKE_RUNTIME_OUTPUT_DIRECTORY_<CONFIG>变量，则该属性由该变量的值初始化。
变量	CMAKE_ARCHIVE_OUTPUT_DIRECTORY	为构建的所有存档(Archive)目标文件指定目录。该变量用于初始化所有目标上的 ARCHIVE_OUTPUT_DIRECTORY 目标属性。
	CMAKE_ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>	上一变量的单配置版本。该变量用于初始化所有目标上的 ARCHIVE_OUTPUT_DIRECTORY_<CONFIG>目标属性。
	CMAKE_LIBRARY_OUTPUT_DIRECTORY	为构建的所有库(Library)目标文件指定目录。该变量用于初始化所有目标上的 LIBRARY_OUTPUT_DIRECTORY 目标属性。
	CMAKE_LIBRARY_OUTPUT_DIRECTORY_<CONFIG>	上一变量的单配置版本。该变量用于初始化所有目标上的 LIBRARY_OUTPUT_DIRECTORY_<CONFIG>目标属性。
	CMAKE_RUNTIME_OUTPUT_DIRECTORY	为构建的所有运行时(Runtime)目标文件指定目录。该变量用于初始化所有目标上的 RUNTIME_OUTPUT_DIRECTORY 目标属性。
	CMAKE_RUNTIME_OUTPUT_DIRECTORY_<CONFIG>	上一变量的单配置版本。该变量用于初始化所有目标上的 RUNTIME_OUTPUT_DIRECTORY_<CONFIG>目标属性。
	CMAKE_INSTALL_DEFAULT_DIRECTORY_PERMISSIONS	在安装文件期间通过 install()和 file(install)隐式创建的目录的默认权限。如果调用 make install 并隐式创建目录，则它们将获得由该变量设置的权限，如果未设置该变量，则获得特定于平台的默认权限。该变量的值是 install()命令 permissions 部分可以使用的权限列表。

表 9.3 为输出工件指定名称的变量	
目标属性	说明
ARCHIVE_OUTPUT_NAME	为构建的存档(Archive)目标文件指定名称，这会覆盖
ARCHIVE_OUTPUT_NAME_<CONFIG>	OUTPUT_NAME 和 OUTPUT_NAME_<CONFIG>目标属性
LIBRARY_OUTPUT_NAME	为构建的库(Library)目标文件指定名称，这会覆盖 OUTPUT_NAME
LIBRARY_OUTPUT_NAME_<CONFIG>	和 OUTPUT_NAME_<CONFIG>目标属性
RUNTIME_OUTPUT_NAME	为构建的运行时(Runtime)目标文件指定名称，这会覆盖
RUNTIME_OUTPUT_NAME_<CONFIG>	OUTPUT_NAME 和 OUTPUT_NAME_<CONFIG>目标属性
OUTPUT_NAME	为构建的可执行文件或库目标文件指定基本的名称，若未设置，则使用
OUTPUT_NAME_<CONFIG>	默认的逻辑目标名称，默认为未设置。

9.1.3 使用 install()命令安装文件的方法

1 前提：以下方法需在 CMakeLists.txt 文件中使用 install()命令并使用 cmake.exe 命令执行 CMakeLists.txt 文件后才能使用，此时，会生成一个名为 cmake_install.cmake 的脚本文件(可使用记事本打开)，该文件包含有安装文件的详细代码。

2、CMake 安装文件的方法为(CMD 命令)

1)、使用 mingw32-make install 命令

```
cmake . #此步会生成 cmake_install.cmake 文件，本示例不需明确使用此文件
mingw32-make install
```

2)、使用 cmake --install 命令

```
cmake . #生成 cmake_install.cmake 文件
mingw32-make #需要先创建安装的目标文件，否则会产生原始文件不存在的错误
cmake --install <dir> [--option] #安装上一步生成的文件，其中 dir 是 cmake_install.cmake 文件所在目录
#option 表示安装时的选项，详见后文示例对--component 参数的使用
```

4)、使用 cmake --P 命令

```
cmake . #生成一个名为 cmake_install.cmake 的文件
mingw32-make #需要先创建安装的目标文件，否则会产生原始文件不存在的错误
cmake -P cmake_install.cmake #直接运行 cmake_install.cmake 脚本文件安装
```

示例 9.1: install()命令的简单使用

①、源文件准备。读者自行准备合适的源文件。

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXX)
#set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY bb) #若使用变量设置输出工件的路径则需放置在前面
```

```

add_library(kk STATIC b.cpp)
add_executable(nn a.cpp )
target_link_libraries(nn kk)
#ARCHIVE 表示静态库文件，DESTINATION 表示目的地，所以，以下语句表示把 kk.lib 安装到 g:/qt1/aa
install(TARGETS kk ARCHIVE DESTINATION g:/qt1/aa )
#使用目标属性设置输出工件的输出路径
set_property(TARGET nn PROPERTY RUNTIME_OUTPUT_DIRECTORY bb)

```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```

cmake .
mingw32-make install          #注意：一定要使用 install，否则 install()命令不会被执行

```

或输入以下命令安装文件

```

cmake .          #此时会生成一个名为 cmake_install.cmake 的文件
mingw32-make     #必须先构建需要安装的文件
cmake --install . #注意末尾有一个小数点“.”，表示在当前目录查找 cmake_install.cmake 文件

```

④、查看运行结果

进入 g:/qt1 目录，可以看到，

- kk.lib 同时位于 g:/qt1 和 g:/qt1/aa 目录，其中 g:/qt1/aa 中的 kk.lib 是使用 install()复制进去的，g:/qt1 中的 kk.lib 是原本输出的位置。也就是说，install()将 kk.lib 从目录 g:/qt1 复制到了 g:/qt1/bb
- nn.exe 位于 g:/qt1/bb 目录，这是使用目标属性 RUNTIME_OUTPUT_DIRECTORY 移动进去的，若不设置该目标属性，nn.exe 将位于目录 g:/qt1，也就是说，该目标属性将 nn.exe 从目录 g:/qt1 移动到了 g:/qt1/bb

9.2 install(TARGETS) 安装目标文件

install(TARGETS)的语法为：

```

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>|CXX_MODULES_BMI]          #目标输出工件类型
    [DESTINATION <dir>]                             #安装目的地
    [PERMISSIONS permissions...]                   #权限
    [CONFIGURATIONS [Debug|Release|...]]           #配置模式，如 DEBUG、RELEASE
    [COMPONENT <component>]                         #指定组件名(用于部分安装)
    [NAMELINK_COMPONENT <component>]
    [OPTIONAL]                                       #即使文件不存在也不产生错误
  ]

```

```

[EXCLUDE_FROM_ALL]                #排除在完整安装之外
[NAMELINK_ONLY|NAMELINK_SKIP]
] [...]
[INCLUDES DESTINATION [<dir> ...]]
)

```

- 1、`install()`命令有多个版本，`install(TARGETS)`版本用于将指定目标的输出工件安装到指定的目录。
- 2、默认情况下 `install()`命令是将构建的目标文件(即输出工件)复制一份到指定的目录，或通过设置 `CMAKE_INSTALL_MODE` 环境变量，在指定目录为目标文件创建一个符号链接，但无论如何，使用 `install()`命令构建的原始目标文件的位置并未改变。
- 3、`TARGETS` 参数表示需安装的目标的名称
- 4、`DESTINATION`(目的地) 参数表示指定文件将安装到的磁盘的目录，可以是相对路径或绝对路径。相对路径是相对于 `CMAKE_INSTALL_PREFIX` 变量设置的前缀，该变量在 Unix 上的默认值为 `/usr/local`，在 Windows10 上的默认值为 `C:\Program Files (x86)\${PROJECT_NAME}`。建议使用相对路径。
- 5、`install(TARGETS)`命令的参数比较多，但最重要，使用最多的是 `DESTINATION` 和目标输出工件类型两个参数，下面将讲解 `install()`命令的其余参数。

9.2.1 `install()`命令输出工件的类型

可安装的目标的输出工件类型如表 9.4 所示。

表 9.4 `install(TARGETS)`的输出工件类型

输出工件类型	对应文件
ARCHIVE	静态库文件(.lib、.a)、 DLL 导入库文件(.lib)，若未指定 <code>RUNTIME</code> 则还包含 DLL 文件 在 AIX、macOS 上启用了 <code>ENABLE_EXPORTS</code> 的链接器导入文件
LIBRARY	模块库文件，比如，使用 <code>add_library(MODULE)</code> 创建的 DLL 文件。不包括 macOS 标记为 <code>FRAMEWORK</code> 、共享库 DLL，即不包括使用 <code>add_library(SHARED)</code> 创建的 DLL。必须为模块库指定 <code>DESTINATION</code> 参数。关于模块库和共享库的区别见 5.2.1 小节。
RUNTIME	共享库 DLL、可执行文件(如.exe 文件)，但不包括 macOS 上标记为 <code>MACOSX_BUNDLE</code> 的可执行文件。关于模块库和共享库的区别见 5.2.1 小节。
OBJECTS	对象文件(如.obj、.o)
FRAMEWORK	macOS 带有 <code>FRAMEWORK</code> 属性的静态库和共享库。必须为 <code>FRAMEWORK</code> 指定 <code>DESTINATION</code> 参数。
BUNDLE	macOS 带有 <code>MACOSX_BUNDLE</code> 属性的可执行文件。必须为 <code>BUNDLE</code> 指定 <code>DESTINATION</code> 参数。
PUBLIC_HEADER	非 apple 平台上，与库关联的任何 <code>PUBLIC_HEADER</code> 文件。该参数在 apple 平台上的 <code>FRAMEWORK</code> 库中被忽略。该参数需设置 <code>PUBLIC_HEADER</code> 目标属性才会生效。
PRIVATE_HEADER	类似于 <code>PUBLIC_HEADER</code> ，但用于 <code>PRIVATE_HEADER</code> 文件。该参数需设置 <code>PRIVATE_HEADER</code> 目标属性才会生效。
RESOURCE	类似于 <code>PUBLIC_HEADER</code> 和 <code>PRIVATE_HEADER</code> ，但用于 <code>RESOURCE</code> 文件。该参数需设置 <code>RESOURCE</code> 目标属性才会生效。
FILE_SET <set>	若文件集 <code>set</code> 存在，且是 <code>PUBLIC</code> 或 <code>INTERFACE</code> ，则该集中的任何文件。相对于文件集的

	基本目录的目录结构将被保留。
CXX_MODULES_BMI	在 CXX_MODULES 类型的文件集中来自 PUBLIC 源的 C++模块的任何文件。指定空的 DESTINATION 将禁止安装这些文件(用于泛型(generic)代码)。
不指定以上参数	设置的安装属性适用于所有目标输出工件类型。比如 install(TARGETS kk DESTINATION g:/qt1/aa) 表示将目标 kk 的各输出工件类型所对应的文件安装到目录 g:/qt1/aa。

示例 9.2：安装静态库文件和可执行文件

①、源文件准备

在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp
#include <iostream>
extern int b;
int main(){ std::cout<<"A="<<b<<std::endl;    return 0;}
```

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
int b=1;
```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk STATIC b.cpp)           #生成 kk.lib(静态库)
add_executable(nn a.cpp )              #生成 nn.exe
target_link_libraries(nn kk)
install(TARGETS kk
    ARCHIVE DESTINATION g:/qt1/aa
    RUNTIME DESTINATION g:/qt1/bb
    LIBRARY DESTINATION g:/qt1/cc )
install(TARGETS nn
    ARCHIVE DESTINATION g:/qt1/aa
    RUNTIME DESTINATION g:/qt1/bb
    LIBRARY DESTINATION g:/qt1/cc )
#以上安装命令可合并在一起，如以下所示
#[[
install(TARGETS kk nn
    ARCHIVE DESTINATION g:/qt1/aa
    RUNTIME DESTINATION g:/qt1/bb
    LIBRARY DESTINATION g:/qt1/cc )
]]
```

③、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```
cmake .
```



```
mingw32-make install
```

④、查看运行结果

- `kk.lib` 分别位于 `g:/qt1`(原始位置)和 `g:/qt1/aa`(安装位置)
- `nn.exe` 分别位于 `g:/qt1`(原始位置)和 `g:/qt1/bb`(安装位置)
- 因为 `add_library(kk STATIC b.cpp)`命令不会生成.exe 文件、DLL 文件，所以 `install` 的 `RUNTIME` 和 `LIBRARY` 参数对目标 `kk` 不起作用
- 因为 `add_executable(nn a.cpp)`命令不会生成.lib 文件、DLL 文件，所以 `install` 的 `ARCHIVE` 和 `LIBRARY` 参数对目标 `nn` 不起作用

示例 9.3: 安装头文件

①、源文件准备

在目录 `g:/qt1` 中的 `a.cpp` 中编写如下代码，并在该目录下创建 `f.h` 和 `g.h` 两个空的头文件

```
//g:/qt1/a.cpp
#include <iostream>
#include "f.h"
#include "g.h"
int main(){ std::cout<<"A"<<std::endl; return 0;}
```

②、在目录 `g:/qt1` 中的 `CMakeLists.txt` 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXX)
add_executable(nn a.cpp )
#必须设置 PUBLIC_HEADER 或 PRIVATE_HEADER 目录属性其中之一，否则 install()命令将不起作用
set_property(TARGET nn PROPERTY PUBLIC_HEADER f.h)
set_property(TARGET nn PROPERTY PRIVATE_HEADER g.h)
install(TARGETS nn
        PUBLIC_HEADER DESTINATION g:/qt1/aa
        PRIVATE_HEADER DESTINATION g:/qt1/bb
        RUNTIME DESTINATION g:/qt1)      #需指定 RUNTIME，否则会在默认目录安装 exe 文件
```

③、在 CMD 中转至 `g:/qt1` 并输入以下命令

```
cmake .
mingw32-make install
```

④、查看运行结果

- `f.h` 分别位于 `g:/qt1`(原始位置)和 `g:/qt1/aa`(安装位置)
- `g.h` 分别位于 `g:/qt1`(原始位置)和 `g:/qt1/bb`(安装位置)

示例 9.4: 安装带有导出符号的 DLL 文件(共享库 DLL 文件)

①、源文件准备

在目录 `g:/qt1` 中的 `a.cpp` 中编写如下代码

```
//g:/qt1/a.cpp
#include <iostream>
__declspec(dllimport) int b; //导入名称(或称为符号)b
```

```
int main(){
    std::cout<<"A="<<b<<std::endl;
    return 0;}
```

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
__declspec(dllexport) int b=1;    //将名称(或称为符号)b 导出
```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
# g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk SHARED b.cpp)      #生成 kk.dll(共享库)和 kk.lib(导入库)
add_executable(nn a.cpp )        #生成 nn.exe
target_link_libraries(nn kk)
install(TARGETS kk
    ARCHIVE DESTINATION g:/qt1/aa
    LIBRARY DESTINATION g:/qt1/bb
    RUNTIME DESTINATION g:/qt1/cc)
install(TARGETS nn
    ARCHIVE DESTINATION g:/qt1/aa
    LIBRARY DESTINATION g:/qt1/bb
    RUNTIME DESTINATION g:/qt1/cc)
```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make install
```

④、查看运行结果

- kk.lib(导入库)分别位于 g:/qt1(原始位置)和 g:/qt1/aa(安装位置)
- kk.dll 和 nn.exe 分别位于 g:/qt1(原始位置)和 g:/qt1/cc(安装位置)
- 本示例不会生成模块库文件所以 install()命令的 LIBRARY 参数对目标 kk 和 nn 都不起作用

示例 9.5: 安装无导出符号的 DLL 文件(模块库 DLL 文件)

①、在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//g:/qt1/b.cpp
int b=1;    //没有任何符号(或名称)导出
```

②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
# g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk MODULE b.cpp)      //生成模块库 DLL, 此类型 DLL 不会生成 .lib 文件
install(TARGETS kk
    ARCHIVE DESTINATION g:/qt1/aa
    LIBRARY DESTINATION g:/qt1/bb    //模块库的安装目录
    RUNTIME DESTINATION g:/qt1/cc)
```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .  
mingw32-make install
```

④、查看运行结果

- kk.dll(模块库)分别位于 g:/qt1(原始位置)和 g:/qt1/bb(安装位置)
- 本示例不会生成共享库 DLL 文件、.exe 文件、.lib 文件，所以 install()命令的 ARCHIVE 和 RUNTIME 参数对目标 kk 不起作用

示例 9.6: 安装对象库文件

①、在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
//g:/qt1/a.cpp  
  
#include <iostream>  
extern int b;  
  
int main(){std::cout<<"A="<<b<<std::endl;return 0;}
```

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
// g:/qt1/b.cpp  
  
int b=1;
```

②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
# g:/qt1/CMakeLists.txt  
  
cmake_minimum_required(VERSION 3.27)  
project(XXXX)  
  
#必须是 add_library(OBJECT)创建的对象库，否则 install 会不起作用  
add_library(kk OBJECT b.cpp)  
add_executable(nn a.cpp )  
target_link_libraries(nn kk)  
  
install(TARGETS kk  
        OBJECTS DESTINATION g:/qt1/dd)  
  
install(TARGETS nn  
        RUNTIME DESTINATION g:/qt1/cc  
        OBJECTS DESTINATION g:/qt1/dd)
```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .  
mingw32-make install
```

④、查看运行结果

- b.cpp.obj 分别位于 G:\qt1\CMakeFiles\kk.dir (原始位置)和 G:\qt1\dd\objects-Debug\kk (安装位置)
- 本示例 add_library(kk OBJECT b.cpp)命令只会生成.obj 文件，不会生成其他文件，所以，不需指定其他文件的安装目录。
- add_executable(nn a.cpp)命令虽然也会创建.obj 文件，但不是由 add_library(OBJECT)命令创建的对象目标，所以，install()命令的 OBJECTS 参数对目标 nn 不起作用

示例 9.7: 安装使用 ENABLE_EXPORTS 目标属性创建的导入库

①、在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
// g:/qt1/a.cpp
```

```
#include <iostream>

__declspec(dllexport) int a=1;          //导出名称

int main(){    std::cout<<"A"<<std::endl;    return 0;}
```

在目录 g:/qt1 中的 e.cpp 中编写如下代码

```
// g:/qt1/e.cpp

#include<iostream>

__declspec(dllimport) int a;          //导入名称

int main(){    std::cout<<"E="<<a<<std::endl;    return 0;}
```

②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
# g:/qt1/CMakeLists.txt

cmake_minimum_required(VERSION 3.27)
project(xxxx)
#生成 kk.exe 和 kk.lib(导入库)
add_executable(kk a.cpp )
#必须设置以下属性, 否则 kk 不能被其他目标链接
set_property(TARGET kk PROPERTY ENABLE_EXPORTS TRUE)
add_executable(nn e.cpp )    #生成 nn.exe
target_link_libraries(nn kk)
install(TARGETS kk
        ARCHIVE DESTINATION g:/qt1/aa
        LIBRARY DESTINATION g:/qt1/bb
        RUNTIME DESTINATION g:/qt1/cc)
```

本示例没有在 DLL 文件中导出符号, 而是在 exe 文件中导出符号, 所以, 本示例不会生成 DLL 文件, 而是生成一个 exe 文件, 并且会同时生成一个该 exe 的导入库文件(kk.lib 文件)和导出文件(kk.exp 文件)。

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make install
```

④、查看运行结果

- kk.lib 分别位于 G:\qt1 (原始位置)和 G:\qt1\aa (安装位置)
- kk.exe 分别位于 G:\qt1 (原始位置)和 G:\qt1\cc(安装位置)

示例 9.8: 安装文件集

①、在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
// g:/qt1/a.cpp

#include <iostream>

int main(){    std::cout<<"A"<<std::endl;    return 0;}
```

②、在 g:/qt1/aa 目录下创建一个空的 f.h 文件, 在 g:/qt1/bb 目录下创建一个空的 g.h 文件

③、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码,

```
# g:/qt1/CMakeLists.txt

cmake_minimum_required(VERSION 3.27)
```

```

project(xxxx)
add_executable(nn a.cpp )
#创建文件集 s
target_sources(nn PUBLIC FILE_SET s TYPE HEADERS FILES aa/f.h bb/g.h )
install(TARGETS nn
        RUNTIME DESTINATION g:/qt1/cc
        FILE_SET s DESTINATION g:/qt1/dd)      #安装文件集 s

```

④、在 CMD 中转至 g:/qt1 并输入以下命令

```

cmake .
mingw32-make install

```

⑤、查看运行结果

- f.h 分别位于 G:\qt1\aa (原始位置)和 G:\qt1\dd\aa (安装位置)
- g.h 分别位于 G:\qt1\aa (原始位置)和 G:\qt1\dd\bb (安装位置)
- 以上示例仅出于演示目的，f.h 和 g.h 并未被实际使用。

9.2.2 install()命令的默认安装路径

1、因为输出工件类型和 DESTINATION 参数都可省略，因此，当这两个参数都省略的话，比如

```
install(TARGETS kk),
```

以上语句，理论上讲各输出工件类型所对应的文件会被安装到各自的默认目录，但并不全是这样。

2、下面是 install()命令默认目录的规则

- 1)、对于可执行文件、静态库文件、共享库文件、文件集、PUBLIC_HEADER、PRIVATE_HEADER，可以省略 DESTINATION 参数，此时将从 CMake 自带的 GNUInstallDirs 模块中的变量获取默认路径。导入库文件和共享库 DLL 文件更进一步的默认安装规则见下文
- 2)、必须为模块库文件、Apple bundle 和 frameworks 提供 DESTINATION 参数。
- 3)、接口库文件和对象库文件也可以省略 DESTINATION 参数，但处理方式与此不同，详见后文。

3、共享库 DLL 文件的默认路径

- 1)、若未指定 RUNTIME 和 ARCHIVE 的目录，则将 RUNTIME 和 ARCHIVE 对应的文件安装到默认目录。
- 2)、若指定了 RUNTIME、ARCHIVE 二者之一的目录，则将二者对应的文件安装到指定的目录，但其他文件不会被安装。比如，若只指定了 ARCHIVE，则只会将导入库文件(.lib)安装到指定目录，对应的.dll 文件不会被安装。
- 3)、若 RUNTIME 和 ARCHIVE 都指定了目录，则将二者对应的文件，安装到各自的目录。

4、GNUInstallDirs 模块

- 1)、GNUInstallDirs 是一个 CMake 自带的模块，在该模块中定义了一些变量，这些变量用于指定安装目录。注：模块可使用 include()命令包含进来。
- 2)、下面列举 GNUInstallDirs 模块中几个常用的变量
 - CMAKE_INSTALL_<dir>：通常是一个相对于安装前缀(CMAKE_INSTALL_PREFIX 变量)的相对路径，但也允许是绝对路径。<dir>取值见下文
 - CMAKE_INSTALL_FULL_<dir>：由 CMAKE_INSTALL_<dir>生成绝对的路径，若该值不是

绝对路径，则添加 CMAKE_INSTALL_PREFIX 变量的值构造一个绝对路径。<dir>的取值见下文。比如，若 CMAKE_INSTALL_<dir>的值为 bin，CMAKE_INSTALL_PREFIX 的值为 g:/qt，则 CMAKE_INSTALL_FULL_<dir>的值为 g:/qt/bin。CMAKE_INSTALL_PREFIX 变量在 Unix 上的默认值为/usr/local，在 Windows10 上的默认值为 C:\Program Files (x86)\\${PROJECT_NAME}(需要有对该文件夹的访问权限)

- <dir>常用取值有(仅列举部分常用取值，全部取值见表 9.6)：BINDIR、LIBDIR、INCLUDEDIR，表 9.5 是<dir>的取值与输出工件类型及其对应目录的关系

表 9.5 install()命令的默认目录

相对路径相对于 CMAKE_INSTALL_PREFIX(安装前缀)变量，其默认值为		
Unix: /usr/local		
Windows10: C:\Program Files (x86)\\${PROJECT_NAME}(需要有对该文件夹的访问权限)		
输出工件类型	GNUInstallDirs 中的变量	默认目录
RUNTIME	CMAKE_INSTALL_BINDIR	bin
LIBRARY	CMAKE_INSTALL_LIBDIR	lib
ARCHIVE	CMAKE_INSTALL_LIBDIR	lib
PRIVATE_HEADER	CMAKE_INSTALL_INCLUDEDIR	include
PUBLIC_HEADER	CMAKE_INSTALL_INCLUDEDIR	include
FILE_SET(type HEADERS)	CMAKE_INSTALL_INCLUDEDIR	include

示例 9.9：安装静态库、可执行文件到默认目录

- ①、在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
// g:/qt1/a.cpp
#include <iostream>
extern int b;
int main(){    std::cout<<"A="<<b<<std::endl;    return 0;}
```

在目录 g:/qt1 中的 a.cpp 中编写如下代码

```
// g:/qt1/b.cpp
int b=1;
```

- ②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码，

```
# g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXX)
add_library(kk STATIC b.cpp)           #生成 kk.lib
add_executable(nn a.cpp )              #生成 nn.exe
target_link_libraries(nn kk)
set(CMAKE_INSTALL_PREFIX g:/qt1/aa)   #设置默认目录的前缀
include(GNUInstallDirs)               #将 GNUInstallDirs 模块包含进来。此步可省略。
install(TARGETS kk)                   #安装到默认目录
install(TARGETS nn)                   #安装到默认目录
```

```
#也可使用以下命令
#install(TARGETS kk ARCHIVE)
#install(TARGETS nn RUNTIME)
```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make install
```

④、查看运行结果

- kk.lib 分别位于 G:\qt1 (原始位置)和 G:\qt1\aa\lib (默认安装位置)
- nn.exe 分别位于 G:\qt1 (原始位置)和 G:\qt1\aa\bin (默认安装位置)

示例 9.10: 安装共享库 DLL 及其导入库文件到默认目录

①、在目录 g:/qt1 中的 f.cpp 中编写如下代码

```
// g:/qt1/f.cpp
__declspec(dllexport) int a=1;
```

在目录 g:/qt1 中的 g.cpp 中编写如下代码

```
//g:/qt1/g.cpp
__declspec(dllexport) int b=2;
```

②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码，

```
# g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXX)
add_library(kk SHARED f.cpp)          #生成 kk.dll (共享库 DLL)和 kk.lib(导入库)
add_library(nn SHARED g.cpp)          #生成 nn.dll (共享库 DLL)和 nn.lib(导入库)
set(CMAKE_INSTALL_PREFIX g:/qt1/aa)  #设置默认目录前缀
include(GNUInstallDirs)
install(TARGETS kk)
#nn.lib 安装到 g:/qt/bb, 但 nn.dll 不会被安装。
install(TARGETS nn ARCHIVE DESTINATION g:/qt1/bb)
```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make install
```

④、查看运行结果

- kk.lib 分别位于 G:\qt1 (原始位置)和 G:\qt1\aa\lib (默认安装位置)
- kk.dll 分别位于 G:\qt1 (原始位置)和 G:\qt1\aa\bin (默认安装位置)
- nn.lib 分别位于 G:\qt1 (原始位置)和 G:\qt1\bb (指定安装位置)
- nn.dll 位于 G:\qt1 (原始位置), 未被安装到其他位置, 因为 install(TARGETS nn ...)命令只指定了 ARCHIVE 参数(安装.lib), 未指定 RUNTIME 参数, 在这种情况下, nn.dll 不会被安装。

示例 9.11: 安装模块库 DLL 文件到默认目录

①、在目录 g:/qt1 中的 f.cpp 中编写如下代码

```
//g:/qt1/f.cpp
int a=1;
```

在目录 g:/qt1 中的 g.cpp 中编写如下代码

```
//g:/qt1/g.cpp
int b=2;
```

②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码，

```
# g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk MODULE f.cpp)           #生成 kk.dll (模块库)
add_library(nn MODULE g.cpp)           #生成 nn.dll (模块库)
set(CMAKE_INSTALL_PREFIX g:/qt1/aa)   #设置默认目录前缀
include(GNUInstallDirs)                #此步可省略
#install(TARGETS kk)                    #错误，必须为模块库指定安装目录
#install(TARGETS kk LIBRARY)            #错误，同上
install(TARGETS kk DESTINATION g:/qt1/aa) #安装 kk.dll 到 g:/qt1/aa
install(TARGETS nn LIBRARY DESTINATION g:/qt1/bb)
```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make install
```

④、查看运行结果

- kk.dll 分别位于 G:\qt1 (原始位置)和 G:\qt1\aa (指定安装位置)
- nn.dll 分别位于 G:\qt1 (原始位置)和 G:\qt1\bb (指定安装位置)

9.2.3 COMPONENT、EXCLUDE_FROM_ALL、OPTION 参数

- 1、EXCLUDE_FROM_ALL 表示将该文件排除在完整安装之外，注意：该参数只对完整安装起作用。
- 2、OPTIONAL 参数表示若安装的文件不存在，则不产生错误。
- 3、COMPONENT 参数用于将需要安装的文件分组，然后在 CMD 命令行中使用以下命令安装指定的分组，这样就可以只安装指定分组的文件，即部分安装。

```
cmake --install yyy --component xxx
```

为与 CMake 官方一致，本文将分组后的每一个分组称为组件(component)。

- 4、若未指定组件，则创建一个名为 Unspecified 的默认组件，默认组件名称可使用以下变量设置

```
CMAKE_INSTALL_DEFAULT_COMPONENT_NAME
```

示例 9.12: COMPONENT 参数的使用方法

```
#EXCLUDE_FROM_ALL、OPTIONAL 以及后文将要介绍的类似参数的使用方法与本示例相同。
install(TARGETS kk nn COMPONENT xx           #目标 kk 和 nn 的所有相关文件都位于 xx 分组
        ARCHIVE ... COMPONENT aa
        PUBLIC_HEADER ... COMPONENT bb
        PRIVATE_HEADER ...
        RUNTIME DESTINATION ... COMPONENT aa)
```

以上命令表示，ARCHIVE 和 RUNTIME 所对应的文件位于 aa 分组，PUBLIC_HEADER 对应的文

件位于 bb 分组，同时，ARCHIVE、RUNTIME、PUBLIC_HEADER、PUBLIC_HEADER 都位于 xx 分组，即分组 xx 作用于整个目标 nn 和 kk。

示例 9.13: 安装指定组件的文件(COMPONENT 和 EXCLUDE_FROM_ALL 参数)

- ①、在目录 g:/qt1 中的 f.cpp 中编写如下代码

```
//g:/qt1/f.cpp
__declspec(dllexport) int a=1;
```

在目录 g:/qt1 中的 e.cpp 中编写如下代码

```
//g:/qt1/e.cpp
#include<iostream>
__declspec(dllimport) int a;
int main(){    std::cout<<"E="<<a<<std::endl;    return 0;}
```

- ②、在 g:/qt1 目录下新建两个名为 f.h 和 g.h 的空的头文件

- ③、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码，

```
# g:/qt1/CMakeLists.txt

cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk SHARED f.cpp)
add_executable(nn e.cpp )
target_link_libraries(nn kk)
set_property(TARGET kk PROPERTY PUBLIC_HEADER f.h)
set_property(TARGET kk PROPERTY PRIVATE_HEADER g.h)

install(TARGETS kk
        ARCHIVE DESTINATION g:/qt1/aa COMPONENT tt EXCLUDE_FROM_ALL
        PUBLIC_HEADER DESTINATION g:/qt1/bb COMPONENT tt
        PRIVATE_HEADER DESTINATION g:/qt1/cc
        RUNTIME DESTINATION g:/qt1/dd COMPONENT tt
)
```

- ④、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make                #生成需要安装的目标文件
cmake --install . --component tt    #安装组件 tt 指定的文件，注意 install 后面有个小数点 “.”
```

由于以上命令不是完整安装，所以 EXCLUDE_FROM_ALL 参数不起作用，又由于 PRIVATE_HEADER 不属于组件 tt，所以 g.h 不会被安装。最终，以上命令将把 kk.lib 安装到 g:/qt1/aa，f.h 安装到 g:/qt1/bb，kk.dll 安装到 g:/qt1/dd

- ⑤、删除以上命令构建和安装的所有文件，并在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make install        #完整安装
```

以上命令是完整安装命令，EXCLUDE_FROM_ALL 参数限定的项不会被安装，即 ARCHIVE 不会被安装，所以，以上命令会把 kk.dll 安装到 g:/qt1/dd，f.h 安装到 g:/qt1/bb，g.h 安装到 g:/qt1/cc，kk.lib

不会被安装。

⑥、使用 `cmake -P` 完整安装的命令如下：

```
cmake -P cmake_install.cmake
```

⑦、使用 `cmake -P` 安装组件 `tt` (部分安装)的命令如下：

```
cmake -DCMAKE_INSTALL_COMPONENT=tt -P cmake_install.cmake
```

其中 `-D` 参数表示定义一个变量，其后可有空格也可没有空格。

9.2.4 PERMISSIONS、CONFIGURATIONS 参数

1、`PERMISSIONS` 参数用于指定已安装文件的权限，若所指定的权限在平台上没有意义，则忽略。其值可取：`OWNER_READ`、`OWNER_WRITE`、`OWNER_EXECUTE`、`GROUP_READ`、`GROUP_WRITE`、`GROUP_EXECUTE`、`WORLD_READ`、`WORLD_WRITE`、`WORLD_EXECUTE`、`SETUID`、`SETGID`。

2、`CONFIGURATIONS` 参数用于指定安装规则所应用的配置模式(如 `Debug`，`Release` 等)。比如

```
install(TARGETS kk
        ARCHIVE DESTINATION g:/qt1/aa CONFIGURATIONS Release
        RUNTIME DESTINATION g:/qt1/bb CONFIGURATIONS Debug)
```

若在 `Release` 模式，则将 `ARCHIVE` 所对应文件安装到 `g:/qt1/aa`，在 `Debug` 模式则将 `RUTIME` 所对应文件安装到 `g:/qt1/bb`

再如：

```
install(TARGETS kk CONFIGURATIONS Release
        ARCHIVE DESTINATION g:/qt1/aa
        RUNTIME DESTINATION g:/qt1/bb)
install(TARGETS kk CONFIGURATIONS Debug
        ARCHIVE DESTINATION g:/qt1/xx
        RUNTIME DESTINATION g:/qt1/yy)
```

在 `Release` 模式，则将 `ARCHIVE` 所对应文件安装到 `g:/qt1/aa`，将 `RUTIME` 所对应文件安装到 `g:/qt1/bb`。在 `Debug` 模式，则将 `ARCHIVE` 所对应文件安装到 `g:/qt1/xx`，将 `RUTIME` 所对应文件安装到 `g:/qt1/yy`。

9.2.5 NAMELINK_ONLY、NAMELINK_SKIP、NAMELINK_COMPONENT 参

数

1、`NAMELINK_ONLY`、`NAMELINK_SKIP`、`NAMELINK_COMPONENT` 参数用于安装带版本控制的共享库。

2、使用 `g++` 可创建带版本控制的共享库文件，详细的创建(生成)带版本控制的共享库文件，以及这些相关的共享库文件是怎样工作和相互联系的，请参阅相关内容，本文不作讲解。

- 3、NAMELINK_ONLY、NAMELINK_SKIP、NAMELINK_COMPONENT 三个参数用于安装名称类似于 libxxx.so 和 libxxx.so.2 的文件，其中 libxxx.so 被 CMake 称为库(或库文件)，libxxx.so.2 被 CMake 称为 namelike(链接名称)。
- 4、若在 LIBRARY 参数之外使用这三个参数将是错误的，在 3.27 及以上版本这三个参数可用于 ARCHIVE 参数来管理 macOS 上启用了 ENABLE_EXPORTS 的共享库创建的链接器导入文件，说简单一点就是，在 3.27 版本，这三个参数可用于 ARCHIVE 参数(但仅作用于 macOS)。
- 5、NAMELINK_COMPONENT 的作用类似于 COMPONENT，用于给需安装的文件进行分组，若未指定，则默认为 COMPONENT 的值。
- 6、NAMELINK_ONLY 参数表示在安装库目标时仅安装 namelike(链接名称)，若库没有版本控制，则该参数不安装任何东西。当指定该参数时，NAMELINK_COMPONENT 和 COMPONENT 都可用来指定链接名称的安装组件(即安装时的分组名称)，但通常应首先 COMPONENT。
- 7、NAMELINK_SKIP 参数表示在安装库目标时，只安装库文件而不安装 namelike(链接名称)，若库没有版本控制，则该参数将安装库。当指定该参数时，则 NAMELINK_COMPONENT 不起作用。
- 8、若 NAMELINK_ONLY 和 NAMELINK_SKIP 参数都没指定，则装同时安装库和 namelike。
- 9、示例：

```
install(TARGETS kk
        ARCHIVE DESTINATION g:/qt1/aa
        LIBRARY DESTINATION g:/qt1/bb
        COMPONENT ss
        NAMELINK_COMPONENT tt
        RUNTIME DESTINATION g:/qt1/cc COMPONENT tt)
```

若选择安装 tt 组件(分组)，则 RUNTIME 和 namelink(链接名称)会被安装，但不会安装库(文件)；若选择安装 ss 组件，则只会安装库，而不会安装 RUNTIME 和 namelink。

带版本控制的共享库简介

带版本控制的共享库的特点是文件名后缀含有共享库的版本号，以 Linux 为例，与这种共享库相关的名字共有三个：

- **real name:** 即共享库本身的文件名，其名称类似于 libxxx.so.2.3.3456，通常含有完整的版本号，其中，lib 是 Linux 约定的前缀，xxx 是共享库名称，.so 是共享库后缀，2.3.3456 是共享库的版本号，由“主版本号+小版本号+补丁号”组成。
- **so-name:** 其格式类似为 libxxx.so.2，这是加载共享库时使用的文件名，其名称与 real name 一致，唯一的区别是 soname 只含有主版本号。
- **namelink:** 其格式类似 libxxx.so，这是共享库的链接名称，是在编译时的链接阶段使用的名称，其名称与 real name 和 soname 一致，其区别是没有版本号。

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

该参数用于指定一个目录列表，当通过 install(EXPORT)命令导出时，这些目录将被添加到<targets>的 INTERFACE_INCLUDE_DIRECTORIES 目标属性中。如果指定了相对路径，则将其视为相对于 \$<INSTALL_PREFIX>的路径。

9.2.7 EXPORT<export-name>参数

- 1、EXPORT <export-name>参数需要与 install(EXPORT)或 export()命令结合使用。该参数的详细使用方法详见后文导入与导出。
- 2、EXPORT <export-name>参数用于将安装的目标导出，并将其放入导出集<export-name>中，该参数必须出现在其他参数之前，若需要生成.cmake 文件，则还需要调用 install(EXPORT)或 export()命令。
- 3、若目标是含有 PUBLIC 或 INTERFACE 的文件集，则必须使用 FILE_SET 参数来指定这些文件集。

9.2.8 RUNTIME_DEPENDENCIES、RUNTIME_DEPENDENCY_SET 参数

- 1、以上参数需结合 install(RUNTIME_DEPENDENCY_SET)命令使用，见 9.4 小节。
- 2、RUNTIME_DEPENDENCY_SET 会将已安装的可执行文件、共享库和模块目标的所有运行时依赖项添加到指定的运行时依赖项集<set-name>中，然后使用 install(RUNTIME_DEPENDENCY_SET)命令安装该集合。
- 3、RUNTIME_DEPENDENCIES 与上一个参数类似，该参数会导致将安装的可执行文件、共享库和模块目标的所有运行时依赖项与目标本身一起安装。
- 4、使用以下命令

```
install(TARGETS nn RUNTIME_DEPENDENCIES ... )
```

在语义上等效于以下两条命令

```
install(TARGETS ... RUNTIME_DEPENDENCY_SET xxx)
install(TARGETS nn RUNTIME_DEPENDENCY_SET xxx args...)
```

其中 xxx 是随机生成的集合名称，args...是以下取值

- DIRECTORIES
- PRE_INCLUDE_REGEXES
- PRE_EXCLUDE_REGEXES
- POST_INCLUDE_REGEXES
- POST_EXCLUDE_REGEXES
- POST_INCLUDE_FILES
- POST_EXCLUDE_FILES

9.3 install(EXPORT) 安装导出

详见第 10 章导入与导出目标

9.4 install(RUNTIME_DEPENDENCY_SET) 安装运行时依赖项

其语法为：

```
install(RUNTIME_DEPENDENCY_SET <set-name>
```

```

[[LIBRARY|RUNTIME|FRAMEWORK]
 [DESTINATION <dir>]
 [PERMISSIONS permissions...]
 [CONFIGURATIONS [Debug|Release|...]]
 [COMPONENT <component>]
 [NAMELINK_COMPONENT <component>]
 [OPTIONAL] [EXCLUDE_FROM_ALL]
 ] [...]
 [PRE_INCLUDE_REGEXES regexes...]
 [PRE_EXCLUDE_REGEXES regexes...]
 [POST_INCLUDE_REGEXES regexes...]
 [POST_EXCLUDE_REGEXES regexes...]
 [POST_INCLUDE_FILES files...]
 [POST_EXCLUDE_FILES files...]
 [DIRECTORIES directories...]
 )

```

- 1、该命令用于安装由 `install(TARGETS)`或 `install(IMPORTED_RUNTIME_ARTIFACTS)`命令创建的运行时依赖项集。
- 2、在 DLL 平台上，运行时依赖集所属目标的依赖文件安装到 `RUNTIME` 参数对应的目的地，在非 DLL 平台上，安装到 `LIBRARY` 参数对应的目的地。`macOS` 框架安装到 `FRAMEWORK` 参数对应的目的地。在构建树中构建的目标永远不会作为运行时依赖项安装，也不会作为它们自己的依赖项安装，除非目标本身是通过 `install(TARGETS)`安装的。

3、工作原理：

- 1)、在生成的安装脚本文件 `cmake_install.cmake` 中调用以下命令来递归获取给定文件所依赖的库文件列表。

```
file(GET_RUNTIME_DEPENDENCIES)
```

以上命令的作用是获取指定文件所依赖的库文件列表并保存到指定的变量中，比如

```

file(GET_RUNTIME_DEPENDENCIES
     RESOLVED_DEPENDENCIES_VAR AA
     EXECUTABLES g:/qt1/nn.exe)

```

以上命令表示获取 `nn.exe` 文件所依赖的库文件列表并将其保存到变量 `AA` 中。其中 `EXECUTABLES` 表示获取可执行文件的依赖文件，除 `EXECUTABLES` 之外还可指定以下参数：

- **LIBRARIES**：表示获取使用 `add_library(SHARED)`创建的库文件(如.dll 对应的导入库)的依赖文件。
- **MODULES**：表示获取使用 `add_library(MODULE)`创建的模块库文件的依赖文件。
- **BUNDLE_EXECUTABLE**：用于 Apple 平台。

- 2)、`install()`命令会向 `file()`命令传递一个参数，其规则为

- 若依赖项集所属的目标是可执行目标则向 `file()`命令传递一个 `EXECUTABLES` 参数
- 若依赖项集所属的目标是共享库目标则向 `file()`命令传递 `LIBRARIES` 参数

- 若依赖项集所属的目标是模块库目标则向 `file()` 命令传递 `MODULES` 参数
- 在 macOS 上，如果其中一个可执行目标的目标属性 `MACOSX_BUNDLE` 为 `TRUE`，那么该可执行目标将作为 `BUNDLE_EXECUTABLE` 参数传递。在 macOS 的运行时依赖集中，最多只能有一个这样的 `bundle` 可执行目标。`MACOSX_BUNDLE` 目标属性对其他平台没有影响。
- 以下子参数作为对应的参数转发给 `file(GET_RUNTIME_DEPENDENCIES)`。它们都支持生成器表达式。

- `DIRECTORIES` <directories>
- `PRE_INCLUDE_REGEXES` <regexes>
- `PRE_EXCLUDE_REGEXES` <regexes>
- `POST_INCLUDE_REGEXES` <regexes>
- `POST_EXCLUDE_REGEXES` <regexes>
- `POST_INCLUDE_FILES` <files>
- `POST_EXCLUDE_FILES` <files>

3)、注意，`file(GET_RUNTIME_DEPENDENCIES)`只支持收集 Windows、Linux 和 macOS 平台的运行时依赖文件，所以 `install(RUNTIME_DEPENDENCY_SET)`也有同样的限制。

4)、比如：

```
install(TARGETS kk
    RUNTIME_DEPENDENCY_SET ss
    ARCHIVE DESTINATION g:/qt1/aa
    RUNTIME DESTINATION g:/qt1/bb
    LIBRARY DESTINATION g:/qt1/cc
)

install(RUNTIME_DEPENDENCY_SET ss
    LIBRARY DESTINATION g:/qt1/xx      #非 DLL 平台安装的目的地
    RUNTIME DESTINATION g:/qt1/yy     #DLL 平台安装的目的地
)
```

若依赖项集 `ss` 所属目标 `kk` 是使用 `add_library(SHARED)` 创建的共享库目标，则在 `cmake_install.cmake` 文件中将有类似以下内容的代码。获取到的依赖文件在 windows 平台上(DLL 平台)，将被安装到 `g:/qt1/yy`，在非 DLL 平台上将被安装到 `g:/qt1/xx`。

```
file(GET_RUNTIME_DEPENDENCIES
    RESOLVED_DEPENDENCIES_VAR _CMAKE_DEPS
    LIBRARIES
    "G:/qt1/kk.dll" )
```

若依赖项集 `ss` 所属目标 `kk` 是使用可执行目标，则在 `cmake_install.cmake` 文件中将有类似以下内容的代码。获取到的依赖文件在 windows 平台上(DLL 平台)，将被安装到 `g:/qt1/yy`，在非 DLL 平台上将被安装到 `g:/qt1/xx`。

```
file(GET_RUNTIME_DEPENDENCIES
    RESOLVED_DEPENDENCIES_VAR _CMAKE_DEPS
    EXECUTABLES
    "G:/qt1/nn.exe"
)
```

4、`install(RUNTIME_DEPENDENCY_SET)`命令的其余参数与 `install(TARGETS)`相同。

9.5 install(IMPORTED_RUNTIME_ARTIFACTS) 安装导入目标的运行时工件

其语法如下：

```
install(IMPORTED_RUNTIME_ARTIFACTS targets...
    [RUNTIME_DEPENDENCY_SET <set-name>]
    [[LIBRARY|RUNTIME|FRAMEWORK|BUNDLE]
    [DESTINATION <dir>]
    [PERMISSIONS permissions...]
    [CONFIGURATIONS [Debug|Release|...]]
    [COMPONENT <component>]
    [OPTIONAL] [EXCLUDE_FROM_ALL]
    ] [...]
)
```

该命令只安装导入目标的运行时工件(除 FRAMEWORK 库、MACOSX_BUNDLE 可执行文件和 BUNDLE CFBundles 外)，比如，不会安装与 DLL 关联的头文件和导入库。对于 FRAMEWORK 库、MACOSX_BUNDLE 可执行文件和 BUNDLE CFBundles 将安装整个目录。该命令的参数与 install(TARGETS)相同。

示例 9.14：安装导入目标的运行时工件

①、在目录 g:/qt1 中的 f.cpp 中编写如下代码

```
//g:/qt1/f.cpp(用于生成 dll 文件)
__declspec(dllexport) int a=1;
```

②、准备需要导入的文件

在 CMD 中转至 g:/qt1 并输入以下命令

```
cl /LD f.cpp
```

③、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码，

```
# g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(XXXX)
add_library(kk SHARED IMPORTED)
#指定导入目标关联的 dll 文件和导入库文件的路径
set_property(TARGET kk PROPERTY IMPORTED_LOCATION g:/qt1/f.dll)
set_property(TARGET kk PROPERTY IMPORTED_IMPLIB g:/qt1/f.lib)
set(CMAKE_INSTALL_PREFIX g:/qt1/aa)
install(IMPORTED_RUNTIME_ARTIFACTS kk
    RUNTIME DESTINATION g:/qt1/bb
    LIBRARY DESTINATION g:/qt1/cc)
```

```
#使用以下命令将产生错误
#install(TARGETS kk RUNTIME DESTINATION g:/qt1/bb)
```

④、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make install
```

⑤、查看运行结果

- kk.dll 分别位于 G:\qt1 (原始位置)和 G:\qt1\bb (指定安装位置)
- f.lib 不会被安装。install(IMPORTED_RUNTIME_ARTIFACTS)命令不会安装导入库文件。

9.6 install(<FILES|PROGRAMS>) 安装文件

其语法如下：

```
install(<FILES|PROGRAMS> files...
      TYPE <type> | DESTINATION <dir>
      [PERMISSIONS permissions...]
      [CONFIGURATIONS [Debug|Release|...]]
      [COMPONENT <component>]
      [RENAME <name>] [OPTIONAL] [EXCLUDE_FROM_ALL])
```

- 1、该命令用于安装文件，默认情况下可将该命令理解为复制文件。若要安装头文件，建议使用文件集 (参见 target_sources(FILE_SET))。
- 2、若文件名以相对路径指定，则相对于当前源目录。若未指定 PERMISSIONS 参数，则默认具有 OWNER_WRITE、OWNER_READ、GROUP_READ 和 WORLD_READ 权限。
- 3、PROGRAMS 参数与 FILES 相同，只是默认权限不相同，PROGRAMS 默认权限还包括 OWNER_EXECUTE、GROUP_EXECUTE 和 WORLD_EXECUTE。
- 4、RENAME 参数用于为已安装的文件指定一个名称，仅当安装单个文件时才允许重命名。
- 5、TYPE 或 DESTINATION 参数用于指定安装的目的地，二者必须指定其一，但不能同时指定。DESTINATION 参数用于直接指定安装路径，而 TYPE 通过指定一个文件类型(见表 9.6)，然后从 GNUInstallDirs 模块中获取相应的变量来自动设置路径，若未定义对应的变量，则使用内置默认值，需要注意的是，GNUInstallDirs 中的路径相对于变量 CMAKE_INSTALL_PREFIX 的值，在 win10 系统，该变量的默认值为 C:\Program Files (x86)\<project-name>(注意是否有访问权限)。

6、GNUInstallDirs 模块

- 1)、GNUInstallDirs 是一个 CMake 自带的模块，在该模块中定义了一些变量，这些变量用于指定安装目录。注：模块可使用 include()命令包含进来。
- 2)、以下是 GNUInstallDirs 模块中定义的变量
 - CMAKE_INSTALL_<dir>：通常是一个相对于安装前缀(CMAKE_INSTALL_PREFIX 变量)的相对路径，但也允许是绝对路径。<dir>取值见表 9.6
 - CMAKE_INSTALL_FULL_<dir>：由 CMAKE_INSTALL_<dir>生成绝对的路径，若该值不是绝对路径，则添加 CMAKE_INSTALL_PREFIX 变量的值构造一个绝对路径。<dir>取值见表 9.6。比如，若 CMAKE_INSTALL_<dir>的值为 bin，CMAKE_INSTALL_PREFIX 的值为

g:/qt, 则 CMAKE_INSTALL_FULL_<dir>的值为 g:/qt/bin。CMAKE_INSTALL_PREFIX 变量在 Unix 上的默认值为/usr/local, 在 Windows10 上的默认值为 C:\Program Files (x86)\\${PROJECT_NAME}(需要有对该文件夹的访问权限)

- 7、表 9.6 列出了 TYPE 参数支持的文件类型与 GNUInstallDirs 中的对应变量及<dir>取值和内置默认值。表 9.6 路径的表示方法如下:

比如, <LOCALSTATE dir>/run 表示 var/run, 其中<LOCALSTATE dir>是 TYPE 取值为 LOCALSTATE 的路径 var。比如

```
install(FILES a.cpp TYPE RUNSTATE)          #安装 a.cpp 到目录<install-prefix>/var/run
```

<DATAROOT dir>前缀使用 GNUInstallDirs 模块中的 CMAKE_INSTALL_DATAROOTDIR 变量的值, 其内置默认值为 share。比如

```
install(FILES a.cpp TYPE INFO)              #安装 a.cpp 到目录<install-prefix>/share/info
```

- 8、install(<FILES|PROGRAMS>)命令的其余参数与 install(TARGETS)相同。

- 9、注意, 不能将多个文件分别安装在不同的目录, 因此以下语句是错误的

```
install(FILES a.cpp TYPE RUNSTATE          #错误, 不能多个文件安装在多个目录
        b.cpp TYPE INFO
        c.cpp DESTINATION g:/qt1/aa)
```

表 9.6 TYPE 支持的文件类型

TYPE 取值	<dir>取值	GNUInstallDirs 变量	内置默认值
BIN	BINDIR	\${CMAKE_INSTALL_BINDIR}	bin
SBIN	SBINDIR	\${CMAKE_INSTALL_SBINDIR}	sbin
LIB	LIBDIR	\${CMAKE_INSTALL_LIBDIR}	lib
INCLUDE	INCLUDEDIR	\${CMAKE_INSTALL_INCLUDEDIR}	include
SYSCONF	SYSCONFDIR	\${CMAKE_INSTALL_SYSCONFDIR}	etc
SHAREDSTATE	SHARESTATEDIR	\${CMAKE_INSTALL_SHARESTATEDIR}	com
LOCALSTATE	LOCALSTATEDIR	\${CMAKE_INSTALL_LOCALSTATEDIR}	var
RUNSTATE	RUNSTATEDIR	\${CMAKE_INSTALL_RUNSTATEDIR}	<LOCALSTATE dir>/run
DATA	DATADIR	\${CMAKE_INSTALL_DATADIR}	<DATAROOT dir>
INFO	INFODIR	\${CMAKE_INSTALL_INFODIR}	<DATAROOT dir>/info
LOCALE	LOCALEDIR	\${CMAKE_INSTALL_LOCALEDIR}	<DATAROOT dir>/locale
MAN	MANDIR	\${CMAKE_INSTALL_MANDIR}	<DATAROOT dir>/man
DOC	DOCDIR	\${CMAKE_INSTALL_DOCDIR}	<DATAROOT dir>/doc
无	LIBEXECDIR	\${CMAKE_INSTALL_LIBEXECDIR}	libexec
无	DATAROOTDIR	\${CMAKE_INSTALL_DATAROOTDIR}	share
无	OLDINCLUDEDIR	\${CMAKE_INSTALL_OLDINCLUDEDIR}	/usr/include

示例 9.15: 安装文件

- ①、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
cmake_minimum_required(VERSION 3.27)
```

```
project(xxxx)

#将 a.cpp、b.cpp、c.cpp 安装(默认为复制)到 g:/qt1/cc, 前提是这 3 个文件必须存在
install(FILES a.cpp b.cpp c.cpp DESTINATION g:/qt1/cc )
```

②、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
```

```
mingw32-make install
```

执行以上命令后, 可在目录 g:/qt1/cc 中看到安装的三个文件。

9.7 install(DIRECTORY) 安装目录

其语法为:

```
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...]
        ]
        [...])
```

- 1、该命令用于安装目录, 在默认情况下, 可以将其理解为复制目录(因此, 也会复制目录中的所有内容)。若要安装头文件目录, 建议使用文件集。各参数意义如下
- 2、CONFIGURATIONS、COMPONENT、OPTIONAL、EXCLUDE_FROM_ALL 参数与 install(TARGETS)相同, 从略。
- 3、TYPE 或 DESTINATION 参数与 install(<FILES|PROGRAMS>)相同, 请参阅相关内容, 从略。
- 4、MESSAGE_NEVER 参数将会禁止显示文件的安装状态。
- 5、目录名的指定(DIRECTORY 参数)

若目录名的路径不是以斜杠结尾的, 则会将路径上的最后一个目录安装到 DESTINATION(目的地), 若目录名的路径是以斜杠结尾的, 则会将该目录下的所有内容安装到 DESTINATION。相对路径的目

录名相对于当前源目录。若未指定目录名，则使用 DESTINATION 的路径，但不会安装任何东西。
图 9.2 说明了目录名的指定规则

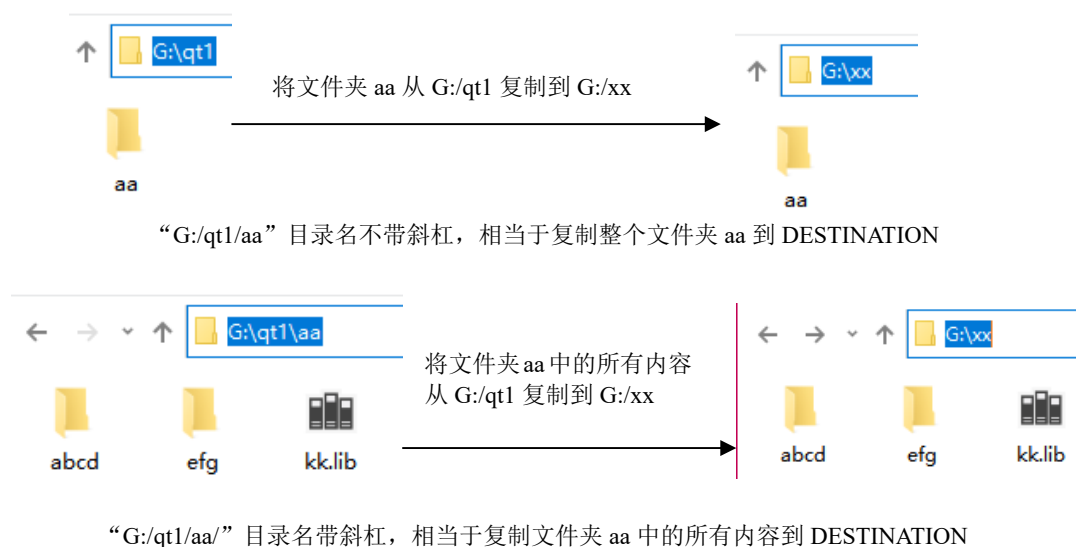


图 9.2 目录名(DIRECTORY 参数)的指定规则

6、权限

FILE_PERMISSIONS、DIRECTORY_PERMISSIONS、USE_SOURCE_PERMISSIONS 参数用于指定文件和目录的权限。若指定了 USE_SOURCE_PERMISSIONS，没有指定 FILE_PERMISSIONS，则文件权限将从源目录结构中复制。若没有指定权限，文件权限将被赋予 install(FILES)形式指定的默认权限，目录将被赋予 install(PROGRAMS)形式指定的默认权限。

7、模式匹配

PATTERN、REGEX 参数可以对需要安装的内容进行过滤，即进行更细粒度(即更进一步)的控制，但是，这两个参数不能单独使用，需与 FILES_MATCHING、EXCLUDE 参数配合使用，因为，在默认情况下，无论是否匹配都会进行安全安装(即，不会进行任何过滤)。下面是过滤规则

- PATTERN 参数必须完全匹配，REGEX 可以匹配任何部分，比如 PATTERN abcd 和 REGEX abcd 则 abc 与 PATTERN 不匹配，但与 REGEX 是匹配的。
- FILES_MATCHING 用于禁止安装任何不匹配的文件(不含目录)，即安装匹配的文件和目录。
- EXCLUDE 表示禁止安装匹配的文件或目录。
- 若不指定 FILES_MATCHING 或 EXCLUDE，则无论是否匹配都会进行安全安装
- PERMISSIONS 表示覆盖匹配的文件或目录的权限。

示例 9.16：安装目录---目录名的指定

①、目录结构如图 9.3

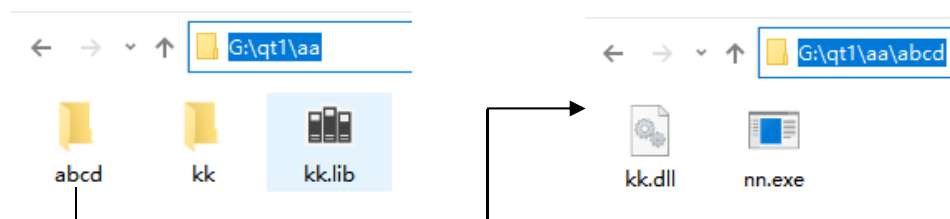


图 9.3 G:/qt1/aa 目录的内容

②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
# g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
install(DIRECTORY aa DESTINATION g:/qt1/bb)          #目录名不带斜杠
install(DIRECTORY aa/ DESTINATION g:/qt1/cc)          #目录名带斜杠
```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make install
```

执行以上命令后，在 G:/qt1/bb 中可以看到，将文件夹 aa 复制到了该目录中；在 G:/qt1/cc 中可以看到，将文件夹 aa 中的所有文件和目录复制到了 G:/qt1/cc 中。

示例 9.17: 安装目录---使用模式过滤安装

①、目录结构使用上一示例

②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
# g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
install(DIRECTORY aa/ DESTINATION g:/qt1/dd
        PATTERN abcd )          #安全安装，因为未指定 FILES_MATCHING 或 EXCLUDE
install(DIRECTORY aa/ DESTINATION g:/qt1/ee
        FILES_MATCHING PATTERN "kk*" ) #安装与 kk*匹配的文件和目录
install(DIRECTORY aa/ DESTINATION g:/qt1/ff
        PATTERN kk* EXCLUDE)      #禁止安装与 kk*匹配的文件或目录
```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make install
```

执行以上命令后，在 G:/qt1/dd 中可以看到，将文件夹 aa 中的所有文件和目录复制到了该目录中(即完全安装)；在 G:/qt1/ee 中可以看到，除文件 nn.exe 没安装外，其余文件或文件夹都已安装；在中 g:/qt1/ff 可看到，只安装了 abcd 文件夹及 nn.exe，文件夹 kk、kk.lib 和 kk.dll 未被安装。

9.8 install(SCRIPT)和 install(CODE) 安装脚本和代码

其语法为：

```
install([[SCRIPT <file>] [CODE <code>]]
        [ALL_COMPONENTS | COMPONENT <component>]
        [EXCLUDE_FROM_ALL] [...])
```

该命令表示在安装过程中调用给定的 CMake 脚本文件(SCRIPT 参数)或调用给定的 CMake 代码(CODE 参数)，代码使用双引号括起来。比如

```
install( SCRIPT ff.cmake          #在安装时调用 ff.cmake 文件
```

```
CODE  "message(BBB)" )      #在安装时输出一条消息
```

第 10 章 导入目标和导出目标

10.1 导入目标

10.1.1 基础

- 1、逻辑目标是指的 CMake 中的命令，如 `add_executable()`、`add_library()`等创建的 CMake 目标，因此，逻辑目标可能没有与之关联的真实文件(CMake 称为输出工件)，也可能有一个或多个文件与之关联的真实文件。比如

```
add_library(kk SHARED ...)
```

```
add_library(nn INTERFACE)
```

逻辑目标 `kk` 与真实文件 `kk.kib` 和 `kk.dll` 关联，逻辑目标 `nn` 没有与之关联的真实文件。

- 2、使用导入库目标的主要目的是将已存在的文件导入为 CMake 项目内部的逻辑目标，并可以在 CMake 中像使用其他常规目标一样使用导入目标，因此，导入目标不会编译源代码(即不会被构建)，或者说导入库目标不会调用编译器命令构建 C++ 程序。

- 3、这里需要注意的是，导入目标是逻辑目标，并且，导入目标不一定必须关联真实文件，比如

```
add_library(kk INTERFACE IMPORTED)
```

以上命令的导入目标 `kk`，是一个接口类型的导入目标，该导入目标没有与之关联的真实文件(也不需要)。

- 4、导入目标需要两个大的步骤：

- 第一步，使用 `add_executable()`或 `add_library()`命令的 `IMPORTED` 参数，创建并设置导入目标的名称。
- 第二步，设置以 `IMPORTED_`和 `INTERFACE_`开头的属性来指定导入目标的详细信息，比如，设置导入的文件的路径、导入目标的使用要求等，例如，使用 `IMPORTED_LOCATION` 目标属性设置导入目标在磁盘上的完整路径。

10.1.2 设置导入目标的详细信息

- 1、导入目标使用 `add_library(IMPORTED)`和 `add_executable(IMPORTED)`命令创建，其语法如下：

```
add_library(<name> <type> IMPORTED [GLOBAL])
```

```
add_executable(<name> IMPORTED [GLOBAL])
```

以上命令创建一个名为`<name>`的导入目标，并将 `IMPORTED` 目标属性设置为 `True`。由于导入目标不会被构建，因此，这里只是创建了一个名称，并未创建任何真实的文件，但这个名称可以像其他目标一样在 CMake 项目中被引用。导入目标通常被 `target_link_libraries()`命令引用。导入目标使用以 `IMPORTED_`和 `INTERFACE_`开头的属性来指定其的详细信息，默认情况下，导入目标的名称的作用范围为创建它的目录及以下目录，但可以使用 `GLOBAL` 参数来扩展可见性，以便在全局可访问。

2、`add_executable()`命令导入的是可执行文件(如.exe 文件)。使用 `IMPORTED_LOCATION` 或 `IMPORTED_LOCATION_<CONFIG>`目标属性指定文件的位置。

3、`add_library()`命令使用`<type>`参数指定导入的文件类型，必须是以下取值之一：

STATIC、SHARED、MODULE、UNKNOWN、OBJECT、INTERFACE

4、`<type>`参数对应的文件及其设置详细信息的方法

1)、STATIC 类型

- STATIC 类型关联的真实文件是静态库文件，如.lib、.a 等。
- 使用 `IMPORTED_LOCATION` 或 `IMPORTED_LOCATION_<CONFIG>`目标属性来指定主库文件在磁盘上的位置。使用要求可以在 `INTERFACE_*`属性中指定。

2)、SHARED 类型

- 关联的文件
 - SHARED 类型关联的是带有导出符号的共享库文件，这种共享库文件通常含有一个与之关联的导入库。
 - 在 Windows 平台通常包括两种类型的文件，DLL 文件(.dll)和与之相关联的导入库文件(.lib)。
 - 对于非 Windows 平台是指带版本控制的共享库文件(见 9.2.5 小节)，其主库文件名是 libxxx.so 文件或.dylib 文件，soname 文件名是 libxxx.so.n，其中 n 是一个整数表示版本号。
- 设置详细信息的方法
 - 对于 Windows 平台
 - 应使用 `IMPORTED_IMPLIB` 或 `IMPORTED_IMPLIB_<CONFIG>`目标属性指定 DLL 的导入库文件(.lib 或.dll.a)在磁盘上的位置；
 - 使用 `IMPORTED_LOCATION` 目标属性指定 DLL 文件的位置(可选，但 `$<TARGET_RUNTIME_DLLS:tgt>`生成器表达式需要)。
 - 其他使用要求可以在 `INTERFACE_*`属性中指定。
 - 需要注意的是，dll 文件并不会参与可执行文件的编译或链接过程，即，在使用 `link` 或 `cl` 命令生成可执行文件时不需要指定 dll 文件，所以，`IMPORTED_LOCATION` 指定的 dll 路径并不会改变可执行文件搜索 dll 文件的位置，dll 文件仍需放于与可执行文件相同的目录或系统环境变量中，否则，会因找不到 dll 文件而出错。
 - 对于非 Windows 平台
 - 若引用的库文件有一个 soname(或 macOS 上有一个以@rpath/开头的 LC_ID_DYLIB)，则应使用 `IMPORTED_SONAME` 目标属性指定该文件的位置。
 - 若引用的库文件没有 SONAME，但平台支持，则应设置 `IMPORTED_NO_SONAME` 目标属性。
 - 使用 `IMPORTED_LOCATION` 或 `IMPORTED_LOCATION_<CONFIG>`目标属性来指定主库文件在磁盘上的位置。
 - 使用要求可以在 `INTERFACE_*`属性中指定。

3)、MODULE 类型

- MODULE 关联的是模块库文件，模块库文件是指的没有导出符号的共享库文件，因此，模块库文件没有与之关联的导入库，在 Windows 平台上，模块库文件的后缀为.dll。
- 使用 `IMPORTED_LOCATION` 或 `IMPORTED_LOCATION_<CONFIG>`目标属性来指定主库文件在磁盘上的位置。使用要求可以在 `INTERFACE_*`属性中指定。

4)、INTERFACE 类型

INTERFACE 表示不引用磁盘上的任何库或对象文件，但可以使用 INTERFACE_*属性指定使用要求。

5)、OBJECT 类型

OBJECT 表示引用位于项目外部的对象文件，使用 IMPORTED_OBJECTS 或 IMPORTED_OBJECTS_<CONFIG>目标属性指定对象文件在磁盘上的位置，使用要求使用 INTERFACE_*属性指定。

6)、UNKNOWN 类型

- UNKNOWN 表示不必知道它是什么类型的库，通常只用于查找模块(Find Modules)，允许使用导入库的路径(通常使用 find_library()命令找到)，而不必知道是什么类型的库，因为在 Windows 上，静态库和 DLL 的导入库具有相同的文件扩展名.lib，所以，在 Windows 上特别有用。
- 使用 IMPORTED_LOCATION 或 IMPORTED_LOCATION_<CONFIG>目标属性来指定主库文件在磁盘上的位置。使用要求可以在 INTERFACE_*属性中指定。

5、表 10.1 是常见的导入的文件类型及其对应的导入目标类型以及设置该导入目标详细信息的目标属性的快速查找表。

6、表 10.2 是 IMPORTED_开头的目标属性

表 10.1 导入目标常见文件及其对应的详细信息的目标属性

注：每一个目标属性都有一个对应的<CONFIG>版本。

导入目标类型	文件类型	常见后缀	目标属性
add_executable(IMPORTED)	可执行文件	.exe	IMPORTED_LOCATION
STATIC	静态库文件	.lib、.a	IMPORTED_LOCATION
SHARED	共享库文件	libxx.so(主库)、.dll(主库)	IMPORTED_LOCATION
	带导出符号	libxx.so.n(SONAME)、.lib(导入库)	IMPORTED_IMPLIB
MODULE	共享库文件 不带导出符号	.dll	IMPORTED_LOCATION
OBJECT	对象文件	.obj	IMPORTED_OBJECTS
INTERFACE	无	无	IMPORTED_LIBNAME
UNKNOWN	不确定	通常用于查找模块	IMPORTED_LOCATION

表 10.2 IMPORTED_*目标属性

属性名	说明
IMPORTED_LOCATION IMPORTED_LOCATION_<CONFIG>	<p>该目标属性用于指定导入目标的主文件在磁盘上的位置(需指定文件名)，导入目标的主文件与导入目标的类型有关，如下：</p> <ul style="list-style-type: none"> ● 可执行文件(.exe)：则表示可执行文件的位置 ● STATIC 库文件或模块文件：则表示库文件或模块文件的位置 ● 非 DLL 平台的共享库文件，则是共享库文件的位置。 ● 对于 DLL，则是.dll 文件的位置。 ● 对于 UNKNOWN 库，则是要链接的文件的位置。

	<ul style="list-style-type: none"> ● macOS 上的应用程序包，则是 bundle 文件夹中 Contents/macOS 中可执行文件的位置。 ● macOS 上的框架，则是库文件符号链接(symlink)的位置。 ● 对于非导入目标，此属性被忽略。 ● IMPORTED_LOCATION_<CONFIG>会覆盖 IMPORTED_LOCATION 设置的值
IMPORTED_IMPLIB IMPORTED_IMPLIB_<CONFIG>	<p>该目标属性用于指定导入目标的导入库文件在磁盘上的位置(需指定文件名)，导入目标的导入库文件与导入目标的类型有关，如下：</p> <ul style="list-style-type: none"> ● 在 DLL 平台上，是 DLL 的导入库文件(.lib 文件)的位置。 ● 在 AIX 上，对于导出符号的可执行文件，是创建的导入文件(如.imp) ● 在 macOS 上，对于共享库，是创建的导入文件(如.tbd)。对于框架，则是框架文件夹内符号链接(symlink).tbd 文件的位置，。 ● 对于非导入目标，此属性被忽略。 ● IMPORTED_IMPLIB_<CONFIG>会覆盖 IMPORTED_IMPLIB 设置的值
IMPORTED_SONAME IMPORTED_SONAME_<CONFIG>	<p>该目标属性用于指定导入目标的 soname 文件在磁盘上的位置。该目标属性只在支持 soname 的平台上才有意义。对于非导入目标，此属性被忽略。</p>
IMPORTED_OBJECTS IMPORTED_OBJECTS_<CONFIG>	<p>该目标属性用于指定对象库类型导入目标的对象文件的绝对路径列表，以分号分隔。</p> <p>IMPORTED_OBJECTS_<CONFIG>可能会覆盖 IMPORTED_OBJECTS 设置的值(之所以是可能覆盖而不是一定覆盖，是因为 Apple 有一个例外，详见参考手册)。</p> <p>对于非导入目标，此属性被忽略。</p>
IMPORTED_NO_SONAME IMPORTED_NO_SONAME_<CONFIG>	<p>该目标属性表示，若导入目标没有 soname 文件，则为 true。对于非导入目标，此属性被忽略。</p>
LOCATION LOCATION_<CONFIG>	<p>只读属性。获取导入目标的位置，该属性仅为库和可执行目标定义。对于非导入的目标，提供此属性是为了与 CMake 2.4 及以下版本兼容。</p>
IMPORTED_CONFIGURATIONS	

示例 10.1：导入目标---导入静态库

①、源文件准备

在目录 g:/qt1 中的 b.cpp 中编写如下代码

```
//b.cpp(用于生成.lib 文件)
int b=1;
```

在 g:/qt1 目录下的 a.cpp 中编写如下代码

```
//a.cpp(exe 文件)
#include<iostream>
extern int b;
int main(){
std::cout<<"E="<<b<<std::endl;    return 0;}
```

②、准备需要导入的文件

在 CMD 中转至 g:/qt1 并输入以下命令

```
cl /c b.cpp
```

```
lib b.obj //创建一个名为 b.lib 的文件，在之后我们将导入该文件到 CMake 项目
```

③、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
# g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk STATIC IMPORTED) #创建一个 STATIC 类型的导入目标 kk
#将 kk 与文件 b.lib 相关联，注意，需指定完整路径。
set_property(TARGET kk PROPERTY IMPORTED_LOCATION_DEBUG g:/qt1/b.lib)
add_executable(nn a.cpp )
target_link_libraries(nn kk) #现在可以像使用其他目标一样使用导入目标了
```

④、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
```

```
mingw32-make
```

```
nn.exe //测试结果，符合预期。
```

示例 10.2：导入共享库目标

①、源文件准备

在目录 g:/qt1 中的 f.cpp 中编写如下代码

```
//g:/qt1/f.cpp(用于生成 dll 文件)
```

```
__declspec(dllexport) int a=1;
```

目录 g:/qt1 中的 e.cpp 中编写如下代码

```
//g:/qt1/e.cpp(主源文件)
```

```
#include<iostream>
__declspec(dllimport) int a;
int main(){ std::cout<<"a="<<a<<std::endl; return 0;}
```

②、准备需要导入的共享库文件

A、本示例准备将由 f.cpp 的构建的导入库文件导入到 CMake 中。

B、在 CMD 中转至 g:/qt1 并输入以下命令创建一个共享库文件 f.dll 及对应的导入库文件 f.lib

```
cl /LD /Fe:yy\ /Fo:yy\ f.cpp
```

使用 Fe 和 Fo 参数将生成的文件输出到当前目录的子目录 yy 中(注意：子目录 yy 必须预先存在，否则以上命令将产生错误)，这样做的目的是能更好的验证导入的共享库目标的使用方法，以上命令最重要的是生成了 f.dll 和 f.lib 两个文件。

③、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj SHARED IMPORTED ) #导入一个共享库目标 jj
add_executable(nn e.cpp)
```

```
target_link_libraries(nn jj)           #将 jj 链接到目标 nn

#指定目标 jj 所关联的文件，需完整路径，这里指定的 f.lib 会在链接阶段链接到目标 nn
set_property(TARGET jj PROPERTY IMPORTED_IMPLIB g:/qt1/yy/f.lib)
#以下步骤不是必须的，所指定的路径并不会影响目标 nn 对 dll 文件的查找。
#所以对 dll 文件的路径可任意指定(建议指定实际位置)
set_property(TARGET nn PROPERTY IMPORTED_LOCATION g:/qt1/yy/f.dll)
```

④、在 CMD 中转至 g:/qt1 并输入以下命令，以验证结果

```
cmake .
mingw32-make
nn.exe           #错误，找不到 f.dll
```

在执行 nn.exe 时会产生找不到 f.dll 文件的错误，这说明在 CMakeLists.txt 文件中对 f.dll 文件路径的指定，并不影响 nn.exe 对 f.dll 的查找，所以，需要把 f.dll 复制到与 nn.exe 相同的目录或系统环境变量中指定的目录中，然后再运行 nn.exe，就能正确运行了。

通过以上两个示例可以看到，导入库目标后，在属性 IMPORTED_* 设置的文件其实才是真正导入的文件，并且该文件可使用 target_link_libraries() 命令在链接阶段链接到目标，所以，只要对相应属性指定了需要导入的正确的文件，目标都能成功构建，这意味着，使用 add_library(STATIC IMPORTED) 命令也可以导入 dll 文件，只需对目标属性 IMPORTED_LOCATION 设置正确的与 dll 文件相关联的导入库文件的路径就可以了(当然，不建议这样做)。同理，导入模块库目标 add_library(STATIC MODULE) 与导入对象库目标 add_library(STATIC OBJECT) 与此原理类似，故不再重述，从略。

10.2 导出目标

10.2.1 CMake 导出目标的思维

- 1、一个 CMake 项目，通常意味着需要一个 CMakeLists.txt 文件(当然，可能还会包含其他文件)，所以，可以简单的将一个 CMake 项目理解为一个 CMakeLists.txt 文件。
- 2、导出目标的作用是将 CMake 项目中的逻辑目标导出去，以方便供其他 CMake 项目使用。
- 3、导出目标的思维及步骤

可以自然的想到，若要导出一个供其他 CMake 项目使用的逻辑目标，需要提供以下信息以及解决以下问题

- 1)、导出的逻辑目标的名称。

CMake 使用 EXPORT 关键字来标识需要导出的逻辑目标。在这里可以提供一些附加的属性，比如是否允许修改该名称，然后在其他 CMake 项目中使用这个新名称。比如，假设需要导出的逻辑目标的名称为 kk，那么，是否允许将 kk 修改为另一个名称 nn，以便在其他 CMake 项目中使用这个新名称 nn。使用 EXPORT_NAME 目标属性可以重命名导出目标的名称。

- 2)、导出目标的详细信息。

CMake 将导出目标的信息分为两类，分别保存在以 INTERFACE_ 和 IMPORTED_ 开头的属性中，其中 INTERFACE_ 保存的是使用要求的信息，IMPORTED_ 保存的是导出目标对应的真实文件的位置信息。这些信息可以使用以下方法设置：

INTERFACE_开头的属性需要显示设置(即使用类似 `set_property()` 之类的命令设置)。显示设置的以 IMPORTED_开头的属性没有效果, 不会被导出, 因此, 以 IMPORTED_开头的属性需要间接设置, 有如下两个方法:

- 使用 `install(TARGETS)` 命令间接设置 IMPORTED_开头的属性。使用该命令的好处是, 可以显示指定与导出目标相对应的真实文件的路径, 但不好之处是, 需要将原始的真实文件安装(默认为复制)到这个位置。
- 使用 `export(TARGETS)` 命令间接设置 IMPORTED_开头的属性, 使用该命令指定的是与导出目标对应的真实文件的构建路径, 并且不能更改, 其好处就是不用安装真实文件。

3)、怎样把多个逻辑目标同时导出呢?

CMake 实现的方式是, 将导出的众多逻辑目标进行分组(或者说归为一个集合中), 其分组的方式非常简单, 只需给每个分组(或集合)取一个名称就可以了, 然后按组将名称导出目标即可。比如, 某公司准备分三个批次组织 30 个员工去旅游, 于是把这些员工分为三组每组 10 个人, 于是给第一组取个名称叫“A 组”, 第二组为“B 组”, 第三组为“C 组”, 这样就成功的完成了分组, 在组织员工出去旅游时, 只需按组组织员工一次就能安排 10 个人出去旅游。CMake 实现的方式与此类似, 即, 只需将导出的逻辑目标放入一个分组(或集合)里, 然后给这个分组取一个名称就可以了, 为描述方便, 本文将这个分组称为**导出集(export set)**, 比如, 假设有目标 `kk` 和 `nn`, 我们打算同时导出这两个目标, 那么, 可以按如下方式来设计

```
kk EXPORT xx          #导出目标 kk, 同时使 kk 位于分组 xx(或称为 xx 集)
nn EXPORT xx          #导出目标 nn, 同时使 nn 位于分组 xx
```

这样 `kk` 和 `nn` 都位于同一个分组中, 当我们导出分组“xx”时就能同时导出目标 `kk` 和 `nn`。

CMake 指定导出集的方法是使用 `install(TARGETS)` 命令, 因此, `install(TARGETS)` 命令不但可以设置 IMPORTED_开头的属性还能指定导出集。

4)、以怎样的方式保存以上信息以及怎样将这些信息传递给其他 CMake 项目呢?

CMake 使用的办法是将这些信息保存到一个后缀为 `.cmake` 的脚本文件中, CMake 可使用以下命令来创建该脚本文件。

```
install(EXPORT)
export(TARGETS)
export(EXPORT)
```

创建的 `xxx.cmake` 文件可能还会附带一个 `xxx-*.cmake` 的文件, 这两个文件含有比较重要的类似于以下的内容(以下内容是经过简化并合并两文件内容后的结果)

```
add_library(kk SHARED IMPORTED)
set_property(...)
set_target_properties(...)
```

该文件使用了 `add_library(IMPORTED)` 命令将导出的目标导入, 这样的话, 在其他 CMake 项目使用这个导出的名称时就可以不需使用 `add_library(IMPORTED)` 命令导入目标了, 这样更方便使用。由 `set_property()` 命令可见, 导出的目标的信息, 以属性的形式保存在 `.cmake` 脚本文件中。

5)、CMake 项目怎样接收从其他 CMake 项目导出的以上信息呢?

CMake 的实现方式是, 在需要使用由外部导出的逻辑目标的项目中, 使用 `include` 命令将上一步生成的 `xxx.cmake` 文件包含进来便可, 然后, 就可以像使用其他目标一样使用这个从外部导出的逻辑目标了。

6)、其他信息。

怎样区分某个目标是从另一个项目导出的目标呢? CMake 使用的方法是, 在导出目标时使用 `NAMESPACE`(名称空间)参数为导出的目标添加一个前缀以示区分, 比如, 若导出的目标名称为

kk，若使用 NAMESPACE 指定一个前缀 “mm::”，则导出目标的名称将为 “mm::kk”，这样，当看到含有双冒号的名称时，就能一眼认出这个目标是从其他项目导出的目标了。注：前缀名称是任意的，并未强制必须使用双冒号，但这是一种习惯性用法，比如 mm-、mm、mm+ 等前缀名称都是合法的。

4、下面总结一下以上步骤的思路，可得出 CMake 导出目标的步骤如下：

- 1)、确定需导出的目标。
- 2)、显示设置导出目标的 INTERFACE_属性。
- 3)、使用 install(EXPORT)或 export(TARGETS)命令间接设置导出目标的 IMPORTED_属性。其中 install(EXPORT)命令可指定导出集但不能生成.cmake 文件。export(TARGETS)不能指定导出集但能直接生成.cmake 文件。
- 4)、使用 install(EXPORT)、export(TARGETS)或 export(EXPORT)命令生成一个<name>.cmake 文件，其中
 - install(EXPORT)、export(EXPORT)命令可以为指定的导出集生成一个<name>.cmake 文件。
 - export(TARGETS)不能为导出集生成<name>.cmake 文件，但能直接为指定的目标生成<name>.cmake 文件。

5)、使用 include()包含生成的<name>.cmake 文件。

<name>.cmake 文件含有类似以下代码的语句

```
add_library(xx::kk SHARED IMPORTED)
```

该语句会导入目标 xx::kk，目标 xx::kk 是由 install()或 export()命令导出的目标。然后在另一个 CMake 项目使用类似如下语句包含<name>.cmake 文件

```
include(<export-name>.cmake)
```

然后就可以在新 CMake 项目中使用导出的目标 xx::kk 了。

5、因此 CMake 总共有如图 10.1 所示的 3 种方法来导出文件，图中最上面一行是导出目标的步骤，在下面对应的虚线框中的内容是该步骤需要使用的 CMake 命令或方法。

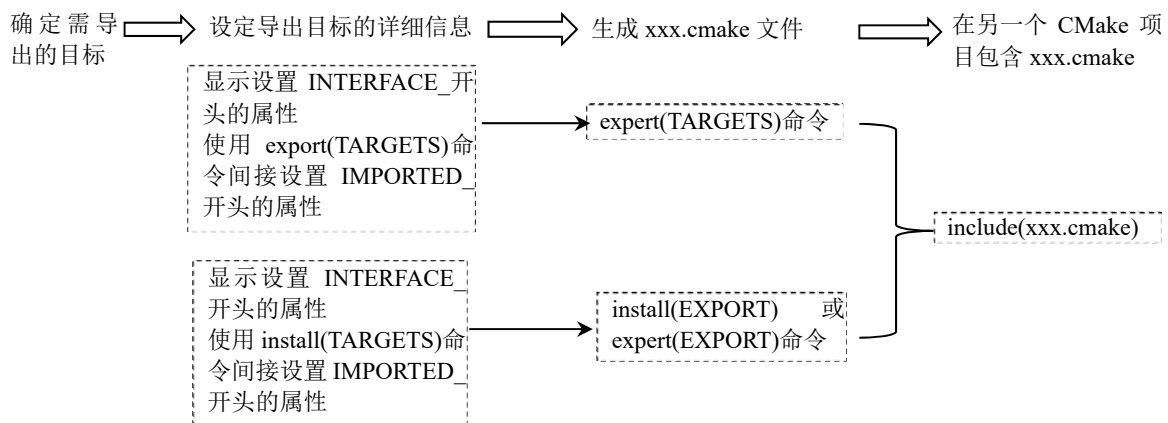


图 10.1 CMake 导出目标的步骤及方法、命令

10.2.2 install(EXPORT)命令

其语法为：

```
install( EXPORT <export-name>                #指定需导出的导出集
        DESTINATION <dir>                      #设置<name>.cmake 的位置
```

```

[NAMESPACE <namespace>]           #为导出的目标添加一个前缀
[FILE <name>.cmake]                 #指定导出的.cmake 文件的名称
[PERMISSIONS permissions...]       #设置权限
[CONFIGURATIONS <config>...]      #指定配置模式
[CXX_MODULES_DIRECTORY <directory>]
[EXPORT_LINK_INTERFACE_LIBRARIES]  #用于兼容性
[COMPONENT <component>]           #指定需导出的目标的分组
[EXCLUDE_FROM_ALL]                #从完整安装中排出
)

```

`install(EXPORT_ANDROID_MK <export-name> DESTINATION <dir> [...])` #用于安卓系统

- 1、`install(EXPORT)`需结合 `install(TARGETS)`命令或 `export(EXPORT)`命令一起使用，并且还需要在另一个 CMake 项目中使用 `include()`将生成的.cmake 文件包含进去。
- 2、`install(EXPORT)`有两个作用：
 - 间接设置导出目标的 `IMPORTED_`属性(即，与导出目标对应的真实文件的位置信息)。
 - 为导出目标创建(或生成)一个.cmake 文件。该.cmake 文件包含导出目标的详细信息，然后另一个 CMake 项目可以使用 `include()`命令将该.cmake 包含进来，然后就可以在这个新的 CMake 项目中使用导出的目标了。
- 4、`EXPORT` 参用于指定需要导出的导出集。`<export-name>`需要与 `install(TARGETS)`的 `EXPORT` 参数的 `<export-name>`的名称相同。
- 5、`FILE` 参用于指定生成的.cmake 文件名为`<name>.cmake`，必须是.cmake 后缀。默认为`<export-name>.cmake`。需要注意的是，安装的`<name>.cmake` 文件可能会附带一个额外的名称为`< name>-.cmake` 的配置文件。
- 6、`DESTINATION` 参用于指定生成的.cmake 文件的安装目录。
- 7、`NAMESPACE` 参用于为导出的目标添加一个前缀，以与其他常规目标相区别。这样便能根据前缀方便的识别出该目标是从其他项目导出的。
- 8、以下参数与 `install(TARGETS)`中的同名参数作用相同： `CONFIGURATIONS`(配置模式)、`PERMISSIONS`(权限)、`EXCLUDE_FROM_ALL`(从完整安装中排除)
- 9、`install(EXPORT_ANDROID_MK ...)`用于安卓系统的导出，本文不作讲解。
- 10、`CXX_MODULES_DIRECTORY <directory>`参数用于指定一个子目录来存储导出集中目标的 C++模块信息。
- 11、当指定了 `COMPONENT` 参数时，将隐式包含导出的所有组件(或分组)。比如

```

install(TARGETS kk EXPORT pp
    ARCHIVE DESTINATION g:/qt1/aa COMPONENT ss
    RUNTIME DESTINATION g:/qt1/rr COMPONENT tt )
install(EXPORT pp
    DESTINATION g:/qt1/rr
    COMPONENT ss)          #将包含 ss 和 tt

```

以上代码中的 `install(EXPORT ... COMPONENT ss)`将包括导出目标 `kk` 的所有组件，即将包含 `ss` 和 `tt`。因此，当安装 `ss` 时，会在 `pp.cmake` 中导入 `ARCHIVE` 和 `RUNTIME` 对应的文件；但，如果只安装 `tt` (`tt` 未在 `install(EXPORT)`中指定)，则不会生成 `pp.cmake` 文件，因此，也就不会导入 `ARCHIVE`

和 RUNTIME 对应的文件。

- 12、EXPORT_LINK_INTERFACE_LIBRARIES 参数用于兼容性，用于导出旧式的目标属性，当 CMP0022 为 NEW 时，若使用了旧式的目标属性 LINK_INTERFACE_LIBRARIES 或 LINK_INTERFACE_LIBRARIES_<CONFIG>时，则需要使用此参数导出目标的旧式目标属性，否则，会产生错误，导出后的目标属性名为

(IMPORTED_)?LINK_INTERFACE_LIBRARIES(_<CONFIG>)?,

注：其中的“?”是正则表达式语法。

比如：

```
cmake_minimum_required(VERSION 3.27)          #此命令会使 CMP0022 被设为 NEW
project(xxxx)
install(TARGETS kk EXPORT pp ARCHIVE DESTINATION g:/qt1/aa)
#本示例出于演示目的，使用直接设置属性的方式
set_property(TARGET kk PROPERTY LINK_INTERFACE_LIBRARIES jj2) #设置旧式目标属性
install(EXPORT pp DESTINATION g:/qt1/rr
EXPORT_LINK_INTERFACE_LIBRARIES )             #因使用了旧式目标属性，所以，必须使用此参数
```

以上代码当 pp.cmake 被安装后，kk 的 LINK_INTERFACE_LIBRARIES 目标属性将被导出为 IMPORTED_LINK_INTERFACE_LIBRARIES_DEBUG

示例 10.3：使用 install()命令导出、导入一个目标

①、源文件准备

在目录 g:/qt1 中的 f.cpp 中编写如下代码

```
// g:/qt1/f.cpp
__declspec(dllexport) int a=1;
```

②、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt(导出目标的项目)
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(kk SHARED f.cpp)
#①设置导出集并设置导出目标的 IMPORTED_*属性
install(TARGETS kk
        EXPORT pp                                #将目标 kk 导出，并使其位于导出集 pp 中
        ARCHIVE DESTINATION g:/qt1/aa          #kk.lib 的安装位置
        RUNTIME DESTINATION g:/qt1/rr          #f.dll 的安装位置
        )
#②生成 ww.cmake 文件
install(EXPORT pp                                #为导出集 pp 生成一个名为 ww.cmake 的文件
        NAMESPACE x-                            #在导出的目标名 kk 之前加上前缀 x-
        FILE ww.cmake                          #将生成的文件.cmake 文件重命名为 ww.cmake(默认为 pp.cmake)
        DESTINATION g:/qt1/rr)                  #指定生成的 ww.cmake 的目录为 g:/qt1/rr
```

③、为新项目准备源文件

在 g:/qt1/rr 目录下的 e.cpp

```
//g:/qt1/rr/e.cpp(为新项目准备的文件)
```

```
#include<iostream>
__declspec(dllimport) int a;
int main(){ std::cout<<"E="<<a<<std::endl;    return 0;}
```

④、在 g:/qt1/rr 目录下的 CMakeLists.txt 中编写如下代码

```
# g:/qt1/rr/CMakeLists.txt(新项目：使用导出目标的项目)
```

```
cmake_minimum_required(VERSION 3.27)
project(XXXX)
include(ww.cmake)          #⑤包含由 g:/qt1/CMakeLists.txt 生成的 ww.cmake 文件
add_executable(nn e.cpp )
target_link_libraries(nn x-kk) #链接 ww.cmake 文件中导入的目标 x-kk
```

⑤、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
mingw32-make install
```

执行以上命令后，会安装以下 4 个文件

```
g:/qt1/aa/kk.lib
g:/qt1/rr/kk.dll
g:/qt1/rr/ww.cmake
g:/qt1/rr/ww-debug.cmake
```

其中.cmake 文件可使用记事本打开，

⑥、分析文件内容

用记事本打开 g:/qt1/rr/ww.cmake 可查找到以下语句(以下内容是经过简化并展开变量后的结果)

```
.....
include(ww-debug.cmake)
.....
add_library(x-kk SHARED IMPORTED)
```

以上会包含 ww-debug.cmake 文件，并使用 add_library(IMPORTED)语句导入一个共享库，其中名称 kk 来自 install(TARGETS kk ...)，名称 x-来自 install(EXPORT pp NAMESPACE x- ...)

再打开 g:/qt1/rr/ww-debug.cmake 可看到以下语句。

```
set_target_properties(x-kk PROPERTIES
    IMPORTED_IMPLIB_DEBUG "g:/qt1/aa/kk.lib"
    IMPORTED_LOCATION_DEBUG "g:/qt1/rr/kk.dll" )
```

将以上语与与 ww.cmake 文件结合，可知 ww.cmake 中的 add_library(x-kk SHARED IMPORTED)语句导入了 kk.lib 和 kk.dll 两个文件。

⑦、接下来在 CMD 中转至 g:/qt1/rr 并输入以下命令

```
cmake .
mingw32-make
nn          #检测结果符合预期
```


10.2.3 export(TARGETS)和 export(EXPORT)命令

其语法分别为

```
export(TARGETS <target>... [NAMESPACE <namespace>]
      [APPEND] FILE <filename> [EXPORT_LINK_INTERFACE_LIBRARIES]
      [CXX_MODULES_DIRECTORY <directory>])
```

```
export(EXPORT <export-name> [NAMESPACE <namespace>] [FILE <filename>]
      [CXX_MODULES_DIRECTORY <directory>])
```

- 1、以上命令除 APPEND 参数外，其余参数与 install(EXPORT)相同。APPEND 表示将导出的目标的信息追加到(而不是覆盖)已存在的<name>.cmake 文件中。
- 2、这两个命令导出的目标的 IMPORTED_属性的位置是构建目录的位置，不能由自己指定 IMPORTED_属性的位置。其中 export(TARGETS)可以直接导出指定的目标并生成.cmake 文件，而 export(EXPORT)只能导出指定的导出集中的目标，并为其生成.cmake 文件。因此 export(EXPORT)需与 install(TARGETS)结合使用，其中 install(TARGETS)命令用于生成导出集；而 export(TARGETS)可以单独使用，不需导出集，这是二者的区别。详见后文的示例。

示例 10.4：使用 install(EXPORT)导出目标

①、编写源文件

在目录 g:/qt1 中的 f.cpp 中编写如下代码

```
//f.cpp(用于生成.dll 文件)
__declspec(dllexport) int a=1;
```

在目录 g:/qt1 中的 g.cpp 中编写如下代码

```
//g.cpp(用于生成.lib 文件)
int b=2;
```

在目录 g:/qt1 中的 e.cpp 中编写如下代码

```
//e.cpp(用于生成.exe 文件)
#include<iostream>
#include "f.h"           //本示例还会演示怎样导出头文件目录
#include "g.h"
__declspec(dllimport) int a;
extern int b;
int main(){ std::cout<<"E="<<a<<":"<<b<<std::endl;    return 0;}
```

②、在 g:/qt1 目录下创建一个名为 f.h 和 g.h 的空头文件

③、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt(导出目标的项目)
#本示例将导出以下内容
#一个-DDD=3 的编译参数
#一个导入库 f.lib、一个静态库 g.lib、一个 DLL 文件 f.dll。
#注意：头文件 f.h 和 g.h 并不是被导出的，而是直接被安装到了指定的位置。
```

```

cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj SHARED f.cpp)
add_library(kk STATIC g.cpp)
add_executable(nn e.cpp )
target_link_libraries(nn jj kk)

#*****

#步骤❶: 设置导出目标 jj 的 INTERFACE_*属性
#*****

#以属性的方式安装头文件
set_property(TARGET jj PROPERTY PUBLIC_HEADER f.h)
#给目标 jj 显示设置 INTERFACE_*属性
set_property(TARGET jj PROPERTY INTERFACE_COMPILE_DEFINITIONS -DDD=3)
#以下属性不是以 INTERFACE_开头的, 因此不会被导出
set_property(TARGET jj PROPERTY COMPILE_DEFINITIONS -DEE=4)
#*****

#步骤❷: 使用 install() 命令间接设置导出目标 jj 和 kk 的 IMPORTED_*属性
#*****

install(TARGETS jj EXPORT yy                                #将目标 jj 导出, 并使其位于导出集 yy 中
        ARCHIVE DESTINATION g:/qt2/aa                      #间接设置 IMPORTED_LOCATION 属性
        RUNTIME DESTINATION g:/qt2                          #忽略。目标 jj 没有 exe 或 dll 文件
        PUBLIC_HEADER DESTINATION g:/qt2/bb                 #安装头文件 f.h
        INCLUDES DESTINATION g:/qt2/cc )                   #间接设置 INTERFACE_INCLUDE_DIRECTORIES

install(TARGETS kk EXPORT yy                                #将目标 yy 导出, 并使其位于导出集 yy 中
        ARCHIVE DESTINATION g:/qt2/aa                      #间接设置 IMPORTED_IMPLIB 属性
        RUNTIME DESTINATION g:/qt2/aa                      #间接设置 IMPORTED_LOCATION 属性
)

install(FILES g.h DESTINATION g:/qt2/cc) #安装头文件 g.h

#*****

#步骤❸: 生成 xxx.cmake 文件
#*****

install(EXPORT yy                                            #导出名称为 yy 的导出集中的所有目标
        FILE tt.cmake                                       #将导出的信息保存在 tt.cmake 文件中(注意: 不能指定路径)
        NAMESPACE mm::                                     #在导出的目标前加上前缀, 以表示该目标是导出(或导入)的
        DESTINATION g:/qt2 ) #指定设置 tt.camke 文件的保存位置

```

④、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
```

```
mingw32-make install
```

输入以上命令后, 会安装以下文件:

```
g:/qt2/aa/jj.lib
```

```
g:/qt2/jj.dll
```

```

g:/qt2/bb/f.h
g:/qt2/aa/kk.lib
g:/qt2/cc/g.h
g:/qt2/tt.cmake
g:/qt2/tt-debug.cmake

```

⑤、查看 tt.cmake 和 tt-*.cmake 文件的内容(其他文件的内容读者可自行查看)

转到 g:/qt2 文件夹，可以看到安装的 tt.cmake 和 tt-*.cmake 文件，其中有以下比较重要的内容(以下内容是经过简化并将两文件内容合并后的结果)

```

add_library(mm::jj SHARED IMPORTED)           #导入目标并命名为 mm::jj
#设置 jj 的 INTERFACE_*属性
set_target_properties(mm::jj PROPERTIES
    INTERFACE_COMPILE_DEFINITIONS "-DDD=3"      #指定一个-DDD=3 的参数
    INTERFACE_INCLUDE_DIRECTORIES "g:/qt2/cc")    #指定头文件包含目录
#设置 jj 的 IMPORTED_*属性
set_property(TARGET mm::jj APPEND PROPERTY IMPORTED_CONFIGURATIONS DEBUG)
set_target_properties(mm::jj PROPERTIES
    IMPORTED_IMPLIB_DEBUG "g:/qt2/aa/jj.lib"      #指定导入库文件的位置
    IMPORTED_LOCATION_DEBUG "g:/qt2/jj.dll" )     #指定.dll 的位置

#目标 kk 的设置与 jj 类似
add_library(mm::kk STATIC IMPORTED)
set_property(TARGET mm::kk APPEND PROPERTY IMPORTED_CONFIGURATIONS DEBUG)
set_target_properties(mm::kk PROPERTIES
    IMPORTED_LINK_INTERFACE_LANGUAGES_DEBUG "CXX"
    IMPORTED_LOCATION_DEBUG "g:/qt2/aa/kk.lib" )

```

⑥、为新项目准备源文件

在 g:/qt2 目录下的 b.cpp 中编写如下代码

```

//g:/qt2/b.cpp(为新项目准备的文件)
#include<iostream>
#include "bb/f.h"           //包含头文件。因为 f.h 位于 g:/qt2/bb 中，所以应使用 bb/f.h
#include "cc/g.h"           //同上
__declspec(dllimport) int a; //导入名称 a，该名称位于 f.dll 中
extern int b;               //名称 b 定义于 kk.lib 中
int main(){ std::cout<<"B="<<DD<<":"<<a<<":"<<b<<std::endl;    return 0;}

```

⑦、在 g:/qt2 目录下的 CMakeLists.txt 中编写如下代码

```

#g:/qt2/CMakeLists.txt(新项目)
cmake_minimum_required(VERSION 3.27)
project(XXXX)
include(tt.cmake)           #步骤④：包含前面生成的 tt.cmake
add_executable(nn b.cpp )
target_link_libraries(nn mm::jj mm::kk) #mm::前缀提示目标 jj 和 kk 是导入的目标

```

⑧、在 CMD 中转至 g:/qt2 并输入以下命令

```

cmake .
mingw32-make
nn.exe          #测试，输出 B=3:1:2，符合预期。

```

示例 10.5：使用 export(TARGETS)导出目标

①、编写源文件

本示例使用上一示例在 g:/qt1 目录下的 f.cpp、g.cpp、e.cpp、f.h、g.h 文件。

②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```

#g:/qt1/CMakeLists.txt(导出目标的项目)
cmake_minimum_required(VERSION 3.27)
project(xxxx)
add_library(jj SHARED f.cpp)
add_library(kk STATIC g.cpp)
add_executable(nn e.cpp )
target_link_libraries(nn jj kk)
#步骤❶：设置 INTERFACE_*属性
set_property(TARGET jj PROPERTY INTERFACE_INCLUDE_DIRECTORIES g:/qt1)
set_property(TARGET jj PROPERTY INTERFACE_COMPILE_DEFINITIONS -DDD=3)
#步骤❷：生成 tt.cmake 文件，并间接设置 IMPORTED_*属性，本示例不需指定导出集
#由于只能使用默认的构建目录位置(不能更改)，所以无需设置 IMPORTED_*属性
export(TARGETS jj kk          #导出目标 jj 和 kk
      FILE g:/qt2/tt.cmake    #tt.cmake 可使用完全路径
      NAMESPACE mm::)        #为导出的目标指定一个前缀

```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```

cmake .
mingw32-make

```

④、查看 g:/qt2/tt.cmake 文件的内容，可以看到，与导出目标对应的真实文件的位置是构建目录的位置。

⑤、为新项目准备源文件

在 g:/qt2 目录下的 b.cpp 中编写如下代码，

```

//g:/qt2/b.cpp(为新项目准备的文件)
#include<iostream>
#include " f.h"          //包含头文件
#include " g.h"          //同上
__declspec(dllimport) int a;  //导入名称 a，该名称位于 f.dll 中
extern int b;            //名称 b 定义于 kk.lib 中
int main(){ std::cout<<"B="<<DD<<":"<<a<<":"<<b<<std::endl;    return 0;}

```

⑥、在 g:/qt2 目录下的 CMakeLists.txt 中编写如下代码

```

#g:/qt2/CMakeLists.txt(新项目)
cmake_minimum_required(VERSION 3.27)
project(xxxx)
include(tt.cmake)
add_executable(nn b.cpp )
target_link_libraries(nn mm::jj mm::kk)

```

- ⑦、在 CMD 中转至 g:/qt2 并输入以下命令

```
cmake .  
mingw32-make
```

- ⑧、将 g:/qt1 文件夹中生成的 jj.dll 复制到 g:/qt2 中，再在 CMD 中输入以下命令
nn.exe #测试，输出 B=3:1:2，符合预期。

示例 10.6：使用 export(EXPORT)导出目标

- ①、编写源文件

本示例使用上一示例在 g:/qt1 目录下的 f.cpp、g.cpp、e.cpp、f.h、g.h 文件。

- ②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

#g:/qt1/CMakeLists.txt(导出目标的项目)

```
cmake_minimum_required(VERSION 3.27)  
project(xxxx)  
add_library(jj SHARED f.cpp)  
add_library(kk STATIC g.cpp)  
add_executable(nn e.cpp )  
target_link_libraries(nn jj kk)  
  
#步骤❶：设置 INTERFACE_*属性  
set_property(TARGET jj PROPERTY INTERFACE_INCLUDE_DIRECTORIES g:/qt1)  
set_property(TARGET jj PROPERTY INTERFACE_COMPILE_DEFINITIONS -DDD=3)  
  
#步骤❷：使用 install()间接设置 IMPORTED_*属性并指定导出集  
install(TARGETS jj EXPORT yy                   #指定导出集 yy  
        ARCHIVE DESTINATION g:/qt2/aa  
        RUNTIME DESTINATION g:/qt2 )  
  
install(TARGETS kk EXPORT yy                   #指定导出集 yy  
        ARCHIVE DESTINATION g:/qt2/aa  
        RUNTIME DESTINATION g:/qt2/aa )  
  
#步骤❸：生成 tt.cmake 文件  
export(EXPORT yy                               #导出导出集中的目标  
       FILE g:/qt2/tt.cmake                   #指定生成的.cmake 文件的名称和路径  
       NAMESPACE mm: )                        #为导出的目标指定一个前缀
```

- ③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .  
mingw32-make
```

- ④、查看 g:/qt2/tt.cmake 文件的内容，读者可自行分析。

- ⑤、准备新项目

使用上一示例在 g:/qt2 目录下的创建的 b.cpp 以及 CMakeLists.txt 文件

- ⑥、在 CMD 中转至 g:/qt2 并输入以下命令

```
cmake .  
mingw32-make
```

- ⑦、将 g:/qt1 文件夹中生成的 jj.dll 复制到 g:/qt2 中，再在 CMD 中输入以下命令

nn.exe #测试，输出 B=3:1:2，符合预期。

第 11 章 `configure_file()`命令(生成配置文件)

`configure_file()`命令的语法如下:

```
configure_file(<input> <output>
               [NO_SOURCE_PERMISSIONS | USE_SOURCE_PERMISSIONS |
               FILE_PERMISSIONS <permissions>...]
               [COPYONLY] [ESCAPE_QUOTES] [@ONLY]
               [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])
```

- 1、该命令用于从一个文件生成另一个文件，其中新文件会按照 CMake 的替换规则替换原始文件中的一些内容。
- 2、<input>参数
表示输入文件，是内容被替换前的原始文件，通常使用后缀.in。输入文件必须指定文件而不能是目录，相对路径相对于 `CMAKE_CURRENT_SOURCE_DIR` 变量的值。
- 3、<output>参数
表示输出文件，是内容被替换后的新文件。输出文件可以指定目录或文件，相对路径相对于 `CMAKE_CURRENT_BINARY_DIR` 变量的值。若输出文件指定的是目录则使用与输入文件相同的文件名。
- 4、输出文件的权限使用以下参数设置
 - `NO_SOURCE_PERMISSIONS` 表示不将输入文件的权限转移到输出文件，默认权限为标准的 644 值(`-rw-r--r--`)
 - `USE_SOURCE_PERMISSIONS` 表示将输入文件的权限转移到输出文件。若未指定权限，则这是默认行为。
 - `FILE_PERMISSIONS <permissions>...`表示忽略输入文件的权限，而对输出文件使用指定的权限
- 5、`COPYONLY` 参数
表示仅仅复制文件而不替换内容。此参数不能与 `NEWLINE_STYLE` 一起使用。
- 6、`ESCAPE_QUOTES` 参数
表示使用反斜杠转义任何替换的引号(C-style)
- 7、`@ONLY` 参数
表示将变量替换限制为`@VAR@`形式的引用。
- 8、`NEWLINE_STYLE` 参数
用于为输出的文件指定换行符的格式，对于 UNIX 或 LF 使用`\n`换行符，对于 DOS、WIN32 或 CRLF 使用`\r\n`换行符。此参数不能与 `COPYONLY` 一起使用。
- 9、`configure_file()`命令的内容替换规则
 - 1)、规则 1: 将输入文件中的`@VAR@`、`${VAR}`、`$CACHE{VAR}`、`$ENV{VAR}`
每个变量引用 `VAR` 替换为变量的当前值，若未定义该变量，则替换为空字符串。
 - 2)、规则 2: 若输入行有如下形式的代码`#cmakedefine VAR ...`

则会被替换为(若 VAR 不是 if()命令的 false 常量)

```
#define VAR ...
```

或(若 VAR 是 if()命令的 false 常量)

```
/* #undef VAR */
```

以上规则可理解为: 若 VAR 是 false 则直接注释掉

3)、规则 3: 若输入行有如下形式的代码

```
#cmakedefine01 VAR
```

则会被替换为(若 VAR 是 if()命令的 false 常量)

```
#define VAR 0
```

或(若 VAR 不是 if()命令的 false 常量)

```
#define VAR 1
```

以上规则可理解为: 若 VAR 是 false 则定义 VAR=0, 否则 VAR=1

4)、规则 4: 若输入行有如下形式的代码

```
#cmakedefine01 VAR ...
```

则会被替换为(若 VAR 是 if()命令的 false 常量)

```
# define VAR ... 0
```

或(若 VAR 不是 if()命令的 false 常量)

```
# define VAR ... 1
```

5)、3.10 及以上版本, 可以在#和 cmakedefine 或 cmakedefine01 之间使用空格或制表符, 这些空格或制表符将被保留在输出行中。

示例 11.1: configure_file()命令的使用----基本替换规则

①、输入文件准备

在 g:/qt1 目录下的 tt.txt 中编写如下代码

```
${AA}+"${BB}"+"@CC@+DD+{EE}
```

该文件是替换前的原始文件, 其中\${AA}、\${BB}、@CC@将会被替换, 其余文本被保留。

②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.27)
```

```
project(xxxx)
```

```
set(AA xx)           #设置变量 AA 的值
```

```
set(CC zz)           #设置变量 CC 的值
```

```
configure_file(tt.txt g:/qt1/ss.txt)    #由 tt.txt 生成 ss.txt。注: 文件后缀未作规定
```

③、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
```

打开 g:/qt1/ss.txt 文件, 可看到该文件的内容如下

```
xx+" "+zz+DD+{EE}
```

其中 xx 由原始文件的\${AA}替换而来, 空格由原始文件的\${BB}替换而来, zz 由原始文件的\${CC}替换而来, 其余字符与原始文件一致。

示例 11.2: configure_file()命令的使用---其他替换规则

- ①、输入文件的内容如图 11.1 所示，位于 g:/qt1，其名称为 tt.txt
- ②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt

cmake_minimum_required(VERSION 3.27)
project(xxxx)

set(AA xx)
set(CC zz)
OPTION(FF "xx" OFF)
configure_file(tt.txt g:/qt1/ss.txt)
```

- ③、在 CMD 中转至 g:/qt1 并输入以下命令
cmake .

最终，tt.txt 及转换后的 ss.txt 的内容如图 11.1 所示，由于生成的 ss.txt 中的内容都是 #define 指令，所以生成的 ss.txt 文件可以作为 C++ 程序的头文件而被包含在 C++ 源代码中。

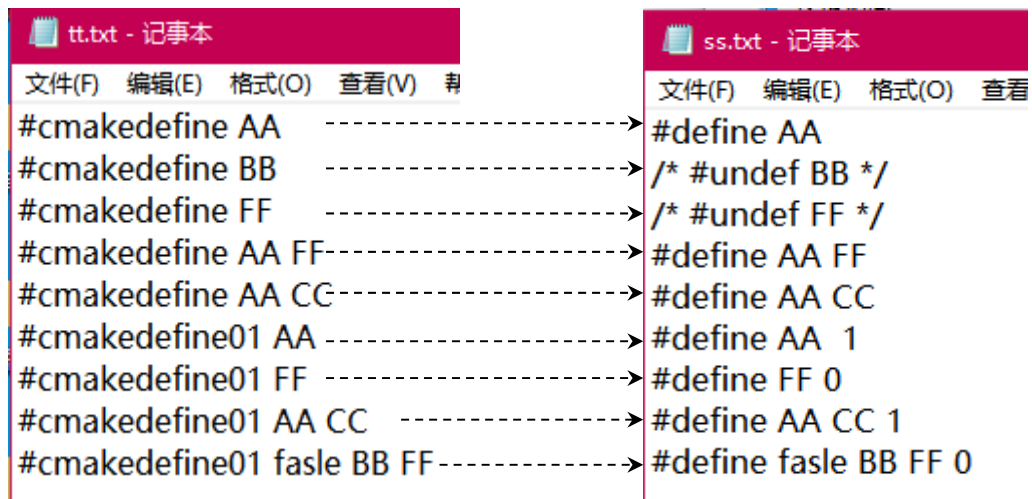


图 11.1 示例 11.2 的输入文件和输出文件的内容

第 12 章 find_file()命令(查找文件)

12.1 find_file()命令的语法及基本参数

find_file()命令速记语法为:

```
find_file (<VAR> name1 [path1 path2 ...])
```

find_file()命令完整语法为:

```
find_file ( <VAR>
            name | NAMES name1 [name2 ...]
            [HINTS [path | ENV var]... ]
            [PATHS [path | ENV var]... ]
            [REGISTRY_VIEW (64|32|64_32|32_64|HOST|TARGET|BOTH)]
            [PATH_SUFFIXES suffix1 [suffix2 ...]]
            [VALIDATOR function]
            [DOC "cache documentation string"]
            [NO_CACHE]
            [REQUIRED]
            [NO_DEFAULT_PATH]
            [NO_PACKAGE_ROOT_PATH]
            [NO_CMAKE_PATH]
            [NO_CMAKE_ENVIRONMENT_PATH]
            [NO_SYSTEM_ENVIRONMENT_PATH]
            [NO_CMAKE_SYSTEM_PATH]
            [NO_CMAKE_INSTALL_PREFIX]
            [CMAKE_FIND_ROOT_PATH_BOTH |
             ONLY_CMAKE_FIND_ROOT_PATH |
             NO_CMAKE_FIND_ROOT_PATH] )
```

- 1、该命令用于查找指定文件的完整路径(含文件名)，并创建一个缓存变量<VAR> (普通变量需指定 NO_CACHE 参数)保存该路径。其搜索规则是：一旦找到相应的文件就结束搜索，并将该文件的完整路径作为结果值；若未找到，则其结果值是<VAR>_NOTFOUND。注意：find_file()命令默认不会搜索构建目录、源目录等目录，这些目录需使用 PATHS 参数、HINTS 参数、或 CMake 变量来明确指定(详见后文搜索顺序)。
- 2、下面是几个常用的参数的意义(其余参数见后文):
 - NAMES: 指定一个或多个需要查找的文件名称。
 - HINTS 和 PATHS: 指定除默认位置外的搜索目录。ENV var 表示从环境变量 var 中读取路径。

- NO_CACHE: 创建一个普通变量(而不是缓存变量)来存储搜索到的结果。
- REQUIRED: 若未搜索到指定的文件则停止处理并显示错误消息。
- DOC: 为<VAR>缓存变量指定一个文档字符串。

3、<VAR>参数及缓存变量

- 1)、该命令虽然可使用 NAMES 指定多个文件名，但，只会存储第一个查找到的文件的完整路径(即一旦找到就结束搜索)。
- 2)、若<VAR>是缓存变量，则一旦查找到结果，除非清除该变量，否则不会重复搜索，即，缓存变量会阻止重复搜索。注：若使用 NO_CACHE 将<VAR>指定为普通变量，则会进行重复搜索。
- 3)、缓存变量在执行 cmake 命令后，会将其值保存在 CMakeCache.txt 文件中，在下次执行 cmake 命令时会直接读取在该文件中的缓存变量的值。因此，清除缓存变量的方式有以下三种：
 - 使用 unset(<VAR> CACHE)命令清除。
 - 删除生成的 CMakeCache.txt 文件
 - 将 CMakeCache.txt 文件中的类似以下的语句删除(注意：以“//”开头的语句也要删掉)

//Path to a file.

<VAR>:FILEPATH=G:/qt1/a.cpp

示例 12.1: 一个简单的 find_file()命令

- ①、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.27)
project(XXXX)
unset(AA CACHE)           #首先，消除缓存变量 AA 的值，以允许重复搜索
find_file(AA NAMES b.cpp c.cpp PATHS g:/qt1)    #变量 AA 是缓存变量
message("AA=${AA}")
```

- ②、假设在 g:/qt1 目录中同时含有 b.cpp 和 c.cpp 文件，本示例将在默认目录和指定的目录 g:/qt1 中按顺序查找文件 b.cpp 和 c.cpp，并创建一个缓存变量 AA 保存查找到的完整路径(含文件名)，因此，本示例 AA 的值为 g:/qt1/b.cpp，因为一旦找到 b.cpp，就结束搜索，此时即使 g:/qt1 中还含有 c.cpp 文件，也不会再查找。

- ③、验证缓存变量阻止重复搜索：

将以上示例的 unset(AA CACHE)注释掉或删除，然后在 CMD 中输入以下命令

```
cmake .
```

此时会输出 AA = g:/qt1/b.cpp。然后将真实文件 b.cpp 删除(注意：不要删除上次执行“cmake .”命令后生成的文件)，然后在 CMD 中输入以下命令

```
cmake .
```

此时仍然会输出 AA=g:/qt1/b.cpp，可见，即使在 g:/qt1 目录中含有 c.cpp 文件也不会再进行搜索，若保留以上示例中的 unset(AA CACHE)语句，则每次执行 cmake .命令都会重新进行搜索。

4、PATH_SUFFIXES 参数

该参数为搜索目录指定一个后缀，比如

```
find_file(AA NAMES a.cpp PATHS g:/qt1 g:/qt2 PATH_SUFFIXES xx)
```

将按顺序在 g:/qt1/xx、g:/qt1、g:/qt2/xx、g:/qt2 目录下搜索文件 a.cpp，并创建一个缓存变量 AA 保存查找到的路径(含文件名)。

再如

```
find_file(AA NAMES a.cpp PATHS g:/qt1 PATH_SUFFIXES xx yy)
```

将按顺序在 g:/qt1/xx、g:/qt1/yy、g:/qt1 目录下搜索文件 a.cpp，并创建一个缓存变量 AA 保存查找到的路径(含文件名)

5、VALIDATOR 参数(验证函数)

该参数用于指定一个验证函数(不能指定宏)。该参数会向验证函数传递以下两个参数：

■ 第一个参数是：CMAKE_FIND_FILE_VALIDATOR_STATUS

以上变量被称为结果变量名称。当指定验证函数时，该参数的值为 TRUE。除非在调用作用域中将该参数的值设置为 FALSE，否则接受搜索到的结果值并结束搜索，也就是说，若该参数的值为 TRUE，则一旦搜索到结果就结束搜索，若在调用作用域中为 FALSE 则放弃搜索到的结果并继续搜索。

■ 第二个参数是：搜索到的文件的绝对路径。

示例 12.2：find_file()命令---验证函数的使用

①、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

```
#g:/qt1/CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.27)
project(XXXX)
unset(AA CACHE)           #取消缓存变量 AA 的值
function(f a b)           #创建一个名为 f 带有两个参数的函数
    message("a=${a}" )    #显示参数 a 的值 CMAKE_FIND_FILE_VALIDATOR_STATUS
    message("b=${b}" )    #显示参数 b 的值
    #以下 if 语句表示，若找到 c.cpp 的路径，则放弃该结果并继续搜索
    if(NOT b MATCHES c.cpp) #若 b 中不含有 c.cpp 字符串
        set(${a} 0 PARENT_SCOPE) #将调用作用域中的
                                   #CMAKE_FIND_FILE_VALIDATOR_STATUS 设为 0
    endif()
endfunction()
find_file(AA NAMES b.cpp c.cpp PATHS g:/qt1 VALIDATOR f)
message("AA=${AA}")       #输出 AA=g:/qt1/c.cpp
```

②、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
```

执行以上命令后，会输出以下内容(假设在 g:/qt1 目录同时含有 b.cpp 和 c.cpp 文件)

```
a=CMAKE_FIND_FILE_VALIDATOR_STATUS
b=G:/qt1/b.cpp
a=CMAKE_FIND_FILE_VALIDATOR_STATUS
b=G:/qt1/c.cpp
AA=G:/qt1/c.cpp
```

③、本示例会调用两次 f 函数，第一次调用 f 函数会输出 b=g:/qt1/b.cpp，由于 if 的结果为 true，所以

CMAKE_FIND_FILE_VALIDATOR_STATUS 在调用作用域中的值被设置为 FALSE，因此，放弃此次搜索结果，继续搜索；第二次调用 f 函数会输出 b=g:/qt1/c.cpp，由于 if 的结果为 false，所以不会对 CMAKE_FIND_FILE_VALIDATOR_STATUS 的值进行设置，本次调用后 CMAKE_FIND_FILE_VALIDATOR_STATUS 在调用作用域中的值为 TRUE，所以结束搜索，并使用这次搜索到的结果，最终 AA=g:/qt1/c.cpp。若未指定验证函数，则缓存变量 AA 的值为 g:/qt1/b.cpp。

6、NO_XXX_PATH：与搜索路径和顺序有关，表示忽略由相应变量指定的路径，详见后文。

7、REGISTRY_VIEW 参数(注册表视图)

该参数只在 Windows 平台上有意义，在其他平台上会被忽略。该参数用于指定必须查询的注册表视图。若未指定，则当 CMP0134 为 NEW 时使用 TARGET 视图，为 OLD 时，默认视图请参阅 CMP0134。以下是该参数下各子参数的意义：

- 64：查询 64 位注册表。在 32 位 windows 上总是返回字符串 /REGISTRY-NOTFOUND
- 32：查询 32 位注册表。
- 64_32：查询 64 和 32 位视图(view)，并为每个视图生成路径。
- 32_63：查询 32 和 64 位视图(view)，并为每个视图生成路径。
- HOST：查询与主机体系结构匹配的注册表：64 位 Windows 为 64 位，32 位 Windows 为 32 位。
- TARGET：查询与 CMAKE_SIZEOF_VOID_P 变量指定的体系结构匹配的注册表，若未定义，则退回到 HOST 视图。
- BOTH：查询 32 和 64 位视图。若定义了 CMAKE_SIZEOF_VOID_P 变量，则根据该变量的内容使用以下视图：

8:64_32

4:32_64

若没有定义 CMAKE_SIZEOF_VOID_P 变量，则依赖于主机的体系结构：

64 位：64_32

32 位：32

12.2 find_file()命令的搜索顺序

- 1、find_file()命令的其他参数都与搜索的顺序有关，下面将介绍 CMake 对文件的搜索顺序。
- 2、注意：find_file()命令默认不会搜索构建目录、源目录等目录，这些目录需使用 PATHS 参数、HINTS 参数、或 CMake 变量来明确指定。
- 3、若指定了 NO_DEFAULT_PATH，则表示禁用默认搜索路径，即，相当于指定了所有的 NO_*，否则按以下顺序搜索。

1)、搜索包目录(包详见第 13 章)

若是从查找模块(find module)或包调用 find_file()命令，则分以下情形

- ①、若是从查找模块(find module)或通过调用 find_package(<PackageName>)加载的任何其他脚本(即.cmake 文件)中调用 find_file()命令，则按顺序搜索由以下变量指定的路径：

- <PackageName>_ROOT, 包名<PackageName>保留大小写。
 - <PACKAGENAME>_ROOT, 包名<PACKAGENAME>是大写的, 适用于 3.27 及以上版本。
 - ENV{<PackageName>_ROOT}, 包名<PackageName>保留大小写。
 - ENV{<PACKAGENAME>_ROOT}, 包名<PACKAGENAME>是大写的, 适用于 3.27 及以上版本。
- ②、如果从嵌套的查找模块(find module)或配置包(一种.cmake 文件)中调用, 则搜索顺序是(路由变量指定)
- <CurrentPackage>_ROOT
 - ENV{<CurrentPackage>_ROOT}
 - <ParentPackage>_ROOT
 - ENV{<ParentPackage>_ROOT}等
- ③、如果设置了 CMAKE_LIBRARY_ARCHITECTURE, 则搜索<prefix>/include/<arch>, 如果从 find_package(<PackageName>)加载的查找模块中调用, 对于<PackageName>_ROOT 变量和 <PackageName>_ROOT 环境变量中的每个<prefix>, 则搜索<prefix>/include
- ④、跳过此搜索规则的方法
- 在 find_file()命令中指定 NO_PACKAGE_ROOT_PATH 参数
 - 将变量 CMAKE_FIND_USE_PACKAGE_ROOT_PATH 设置为 FALSE

2)、搜索缓存变量

- ①、按顺序搜索以下 CMake 特定缓存变量中指定的路径。这些变量可以在命令行中使用-DVAR=value 指定, 这些值将被解释为以分号分隔的列表。
- 如果设置了 CMAKE_LIBRARY_ARCHITECTURE, 则为<prefix>/include/<arch>;
 - 对于变量 CMAKE_PREFIX_PATH(搜索目录前缀)中的每个<prefix>, 则搜索 <prefix>/include。默认为空。因此, 会按顺序搜索以下路径
 - CMAKE_PREFIX_PATH/include
 - CMAKE_PREFIX_PATH 变量指定的路径。
 - CMAKE_INCLUDE_PATH。默认为空。
 - CMAKE_FRAMEWORK_PATH (适用于 macOS)

②、跳过此搜索规则的方法

- 在 find_file()命令中指定 NO_CMAKE_PATH 参数
- 将 CMAKE_FIND_USE_CMAKE_PATH 变量的值设为 FALSE

3)、搜索环境变量

- ①、按顺序搜索以下的特定于 CMake 的环境变量中指定的路径。
- 如果设置了 CMAKE_LIBRARY_ARCHITECTURE, 则为<prefix>/include/<arch>;
 - 对于变量 CMAKE_PREFIX_PATH(搜索目录前缀)中的每个<prefix>, 则搜索 <prefix>/include。默认为空。因此, 会按顺序搜索以下路径

- CMAKE_PREFIX_PATH/include
- CMAKE_PREFIX_PATH 变量指定的路径
- CMAKE_INCLUDE_PATH。默认为空。
- CMAKE_FRAMEWORK_PATH (适用于 macOS)

②、跳过此搜索规则的方法

- 在 find_file()命令中指定 NO_CMAKE_ENVIRONMENT_PATH 参数
- 将 CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH 变量的值设为 FALSE

4)、搜索由 HINTS 参数指定的路径。

5)、搜索标准系统环境变量

①、按顺序搜索以下的标准系统环境变量。

- INCLUDE 或 PATH 系统环境变量指定的目录
- 在 Windows 主机上，若设置了 CMAKE_LIBRARY_ARCHITECTURE，则设置为 <prefix>/include/<arch>;
- PATH 系统环境变量中的<prefix>/bin 或<prefix>/sbin 对应<prefix>/include，因此，会按顺序搜索以下路径
 - <prefix>/include
 - <prefix>
 - <prefix>/bin 或<prefix>/sbin

PATH 中的其他项对应<entry>/include。因此，会按顺序搜索以下路径

- <entry>/include
- <entry>

②、跳过此搜索规则的方法

- 在 find_file()命令中指定了 NO_SYSTEM_ENVIRONMENT_PATH 参数
- 将 CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH 变量的值设为 FALSE

6)、搜索系统目录

①、按顺序搜索当前系统的平台文件中定义的以下 CMake 变量。这些变量通常是安装软件的位置。

- 如果设置了 CMAKE_LIBRARY_ARCHITECTURE，则为<prefix>/include/<arch>;
- 对于变量 CMAKE_SYSTEM_PREFIX_PATH(**系统目录前缀**)中的每个<prefix>，则搜索 <prefix>/include。因此，会按顺序搜索以下路径
 - CMAKE_SYSTEM_PREFIX_PATH/include
 - CMAKE_SYSTEM_PREFIX_PATH

CMAKE_SYSTEM_PREFIX_PATH 默认情况下包括以下路径(注意，还有添加子目录 /include 的版本)

- 当前系统的系统目录，通常是安装软件的目录，因平台而异，比如，对于 Windows 系统可能是 C:/Program Files;C:/Program Files (x86);
- 变量 CMAKE_INSTALL_PREFIX(**安装前缀**)指定的目录。

➤ 变量 CMAKE_STAGING_PREFIX(分段前缀)指定的目录。

注意：搜索目录时实际搜索的是 CMAKE_SYSTEM_PREFIX_PATH 变量指定的路径，安装前缀和分段前缀只可以间接改变 CMAKE_SYSTEM_PREFIX_PATH 变量的部分值，搜索目录时并不以安装前缀和分段前缀指定的路径为准来搜索，需把这两个变量的值传递给 CMAKE_SYSTEM_PREFIX_PATH 变量才会搜索他们设置的路径。

注意：由于 CMAKE_SYSTEM_PREFIX_PATH 变量在第一次调用 project()或 enable_language()命令时初始化，因此，安装前缀、分段前缀、CMAKE_FIND_NO_INSTALL_PREFIX 变量的值必须在此之前设置才能生效。

- CMAKE_SYSTEM_INCLUDE_PATH。默认为空
- CMAKE_SYSTEM_FRAMEWORK_PATH。适用于 macOS。

②、跳过此搜索规则的方法

- 若在 find_file()命令中指定了 NO_CMAKE_INSTALL_PREFIX，或者将 CMAKE_FIND_USE_INSTALL_PREFIX 变量设置为 FALSE，则可以跳过 CMAKE_INSTALL_PREFIX 和 CMAKE_STAGING_PREFIX 的查找。
- 若在 find_file()命令中指定了 NO_CMAKE_SYSTEM_PATH，或者将 CMAKE_FIND_USE_CMAKE_SYSTEM_PATH 设置为 FALSE，则可以跳过此处的所有搜索规则。

7)、搜索由 PATHS 选项指定的路径或该命令的速记版本。

4、CMAKE_IGNORE_PATH, CMAKE_IGNORE_PREFIX_PATH, CMAKE_SYSTEM_IGNORE_PATH 和 CMAKE_SYSTEM_IGNORE_PREFIX_PATH 变量也可以导致忽略上述一些位置。

5、表 12.1 是对 find_file()命令搜索顺序的总结

表 12.1 find_file()命令的搜索顺序

说明：未考虑设置 CMAKE_LIBRARY_ARCHITECTURE 变量的情形

搜索顺序	规则	说明
禁用搜索	搜索路径	若指定 NO_DEFAULT_PATH 参数表示禁用默认搜索路径，即相当于指定了所有的 NO_*参数，否则按以下顺序搜索
1、包目录	搜索路径	<PackageName>_ROOT <PACKAGENAME>_ROOT ENV{<PackageName>_ROOT} ENV{<PACKAGENAME>_ROOT}
	跳过搜索的方法	CMAKE_FIND_USE_PACKAGE_ROOT_PATH 变量设为 FALSE 指定 NO_PACKAGE_ROOT_PATH 参数
2、缓存变量	搜索路径	CMAKE_PREFIX_PATH/include CMAKE_PREFIX_PATH (搜索目录前缀，默认为空) CMAKE_INCLUDE_PATH(默认为空) CMAKE_FRAMEWORK_PATH(适用 macOS)
	跳过搜索的方法	CMAKE_FIND_USE_CMAKE_PATH 变量设为 FALSE 指定 NO_CMAKE_PATH 参数

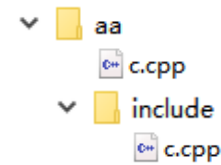
3、环境变量	搜索路径	CMAKE_PREFIX_PATH/include CMAKE_PREFIX_PATH (搜索目录前缀，默认为空) CMAKE_INCLUDE_PATH (默认为空) CMAKE_FRAMEWORK_PATH(适用 macOS)
	跳过搜索的方法	CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH 变量设为 FALSE 指定 NO_CMAKE_ENVIRONMENT_PATH 参数
4、HINTS 参数	搜索路径	指定的路径
5、标准系统环境变量	搜索路径	INCLUDE 或 PATH 系统环境变量指定的目录 对于 PATH 中的每个<prefix>/[s]bin 对应<prefix>/include，按顺序搜索以下路径 <ul style="list-style-type: none"> ● <prefix>/include ● <prefix> ● <prefix>/[s]bin PATH 中的其他项对应<entry>/include，按顺序搜索以下路径 <ul style="list-style-type: none"> ● <entry>/include ● <entry>
	跳过搜索的方法	CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH 变量设为 FALSE 指定 NO_SYSTEM_ENVIRONMENT_PATH 参数
6、系统目录	搜索路径	①、CMAKE_SYSTEM_PREFIX_PATH/include CMAKE_SYSTEM_PREFIX_PATH(系统目录前缀) 以上变量默认包含以下目录。 <ul style="list-style-type: none"> ● Windows 系统可能是 C:/Program Files;C:/Program Files (x86); ● CMAKE_INSTALL_PREFIX(安装前缀) ● CMAKE_STAGING_PREFIX(分段前缀) 注意，还有添加子目录 /include 的版本。若想通过以上变量改变 CMAKE_SYSTEM_PREFIX_PATH 的值，需在 project()之前设置以上变量 ②、CMAKE_SYSTEM_INCLUDE_PATH(默认为空) ③、CMAKE_SYSTEM_FRAMEWORK_PATH(适用 macOS)
	跳过搜索的方法	①、以下方法跳过查找 CMAKE_INSTALL_PREFIX 和 CMAKE_STAGING_PREFIX <ul style="list-style-type: none"> ● CMAKE_FIND_USE_INSTALL_PREFIX 变量设为 FALSE ● 指定 NO_CMAKE_INSTALL_PREFIX 参数 ②、以下方法跳过此处的所有搜索规则 <ul style="list-style-type: none"> ● CMAKE_FIND_USE_CMAKE_SYSTEM_PATH 变量设为 FALSE ● 指定 NO_CMAKE_SYSTEM_PATH 参数
7、PATHS 参数	搜索路径	指定的路径
8、其他	CMAKE_IGNORE_PATH, CMAKE_IGNORE_PREFIX_PATH, CMAKE_SYSTEM_IGNORE_PATH 和 CMAKE_SYSTEM_IGNORE_PREFIX_PATH 变量也可以导致忽略上述一些位置。	

示例 12.3: find_file()命令的搜索顺序

①、本示例将使用验证函数来验证 find_file()命令的搜索顺序，由于该函数只能捕获到已搜索到的文件的路径，所以，需要在每个搜索路径中都添加需要搜索的文件，否则，无法判断 find_file()是否搜索过该路径。

②、文件准备

在 g:/qt1 目录中创建一个 c.cpp 文件(可以是空文件)及 aa、bb、cc、dd、ee、ff、gg、hh、ii、xx、include、sbin 文件夹，并在这些文件夹中创建一个 include 文件夹和 c.cpp 文件，最后在 include 文件夹中创建一个 c.cpp 文件，其中 aa 文件夹结构如右图所示(其余文件夹中的内容与 aa 相同)。



③、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

#g:/qt1/CMakeLists.txt

```
#需在 project() 之前设置以下两个变量才能使设置的值影响 CMAKE_SYSTEM_PREFIX_PATH 变量的值
set(CMAKE_INSTALL_PREFIX g:/qt1/ff)          #⑥ 系统目录--系统目录前缀之安装前缀
set(CMAKE_STAGING_PREFIX g:/qt1/gg)          #⑦ 系统目录--系统目录前缀之分段前缀
cmake_minimum_required(VERSION 3.27)
project(xxxx)

#=====find_file() 命令将按所标示的顺序搜索路径=====
set(CMAKE_PREFIX_PATH g:/qt1/aa)             #① 缓存变量
set(CMAKE_INCLUDE_PATH g:/qt1/bb)            #② 缓存变量
set(ENV{CMAKE_PREFIX_PATH} g:/qt1/cc)         #③ 环境变量
set(ENV{CMAKE_INCLUDE_PATH} g:/qt1/dd)        #④ 环境变量
set(ENV{PATH} "g:/qt1/ee;g:/qt1/sbin")        #⑤ 系统环境变量
#注意：应使用 list 追加值。set 会清除之前的值
list(APPEND CMAKE_SYSTEM_PREFIX_PATH g:/qt1/hh) #⑧ 系统目录--系统目录前缀之其他路径
message("ppp=${CMAKE_SYSTEM_PREFIX_PATH}") #验证 CMAKE_SYSTEM_PREFIX_PATH 的值有哪些
set(CMAKE_SYSTEM_INCLUDE_PATH g:/qt1/ii)      #⑨ 系统目录

#=====指定一个验证函数，用于验证搜索顺序=====
function(f a b)                               #第二个参数保存有搜索到的路径，详见对验证函数的讲解
message("b=${b}")                             #输出搜索到的路径
if(NOT b MATCHES G:/qt1/xx)                   #若搜索到的路径与 G:/qt1/xx 不匹配，则为 TRUE
    set(${a} 0 PARENT_SCOPE)                  #放弃本次搜索，继续下一次搜索
    message(=====)
endif()
endfunction()

unset(AA CACHE)                               #清除缓存变量，以便能重复赋值
find_file(AA NAMES c.cpp
    PATHS g:/qt1/xx                           #⑩ PATHS 参数
    VALIDATOR f )                             #添加验证函数 f
message("AA=${AA}")                           #查看最终搜索到的路径
```

④、在 CMD 中转至 g:/qt1 并输入以下命令

cmake .

然后可以看到如图 12.1 的输出内容，CMakeLists.txt 代码中的搜索顺序只是一个粗略的搜索顺序，图 12.1 中的“b=...”是详细的搜索路径顺序。

ppp=C:/Program Files;C:/Program Files (x86);C:/Qt/Tools/CMake_64;g:/qt1/ff;g:/qt1/gg;g:/qt1/hh	系统目录前缀: CMAKE_SYSTEM_PREFIX_PATH 变量的值
b=G:/qt1/aa/include/c.cpp ----->	(1) 缓存变量: CMAKE_PREFIX_PATH (搜索目录前缀)
=====	
b=G:/qt1/aa/c.cpp	(2) 缓存变量: CMAKE_PREFIX_PATH (搜索目录前缀)
=====	
b=G:/qt1/bb/c.cpp	(3) 缓存变量: CMAKE_INCLUDE_PATH
=====	
b=G:/qt1/cc/include/c.cpp ----->	(4) 环境变量: ENV{CMAKE_PREFIX_PATH} (搜索目录前缀)
=====	
b=G:/qt1/cc/c.cpp	(5) 环境变量: ENV{CMAKE_PREFIX_PATH} (搜索目录前缀)
=====	
b=G:/qt1/dd/c.cpp	(6) 环境变量: ENV{CMAKE_INCLUDE_PATH}
=====	
b=G:/qt1/ee/include/c.cpp ----->	(7) 系统环境变量: ENV{PATH}, 其他前缀
=====	
b=G:/qt1/ee/c.cpp	(8) 系统环境变量: ENV{PATH}, 其他前缀
=====	
b=G:/qt1/include/c.cpp	(9) 系统环境变量: ENV{PATH}, <prefix>/[s]bin 前缀
=====	
b=G:/qt1/c.cpp	(10) 系统环境变量: ENV{PATH}, <prefix>/[s]bin 前缀
=====	
b=G:/qt1/ee/c.cpp	(11) 系统环境变量: ENV{PATH}, 其他前缀
=====	
b=G:/qt1/sbin/c.cpp	(12) 系统环境变量: ENV{PATH}, <prefix>/[s]bin 前缀
=====	
b=G:/qt1/ff/include/c.cpp ----->	(13) 系统目录: 系统目录前缀 CMAKE_SYSTEM_PREFIX_PATH 之安装前缀
=====	
b=G:/qt1/ff/c.cpp ----->	(14) 系统目录: 系统目录前缀 CMAKE_SYSTEM_PREFIX_PATH 之安装前缀
=====	
b=G:/qt1/gg/include/c.cpp	(15) 系统目录: 系统目录前缀 CMAKE_SYSTEM_PREFIX_PATH 之分段前缀
=====	
b=G:/qt1/gg/c.cpp	(16) 系统目录: 系统目录前缀 CMAKE_SYSTEM_PREFIX_PATH 之分段前缀
=====	
b=G:/qt1/hh/include/c.cpp	(17) 系统目录: 系统目录前缀 CMAKE_SYSTEM_PREFIX_PATH 之其他路径
=====	
b=G:/qt1/hh/c.cpp	(18) 系统目录: 系统目录前缀 CMAKE_SYSTEM_PREFIX_PATH 之其他路径
=====	
b=G:/qt1/ii/c.cpp	(19) 系统目录: CMAKE_SYSTEM_INCLUDE_PATH
=====	
b=G:/qt1/xx/c.cpp ----->	(20) PATHS 参数: find_file()命令的 PATHS 参数
AA=G:/qt1/xx/c.cpp ----->	最终搜索到的路径是 PATHS 参数的路径

图 12.1 示例 12.3 的搜索顺序

12.3 搜索指定根目录下的子目录

- 1、CMake 将这里指定的这个根目录称为 re-root(重生根目录)。比如，有一系列目录 g:/qt1/aa, g:/qt1/bb, g:/qt2/xx, 若只搜索指定的根目录 g:/qt1, 则只会搜索 g:/qt1/aa、g:/qt1/bb。注意：经测试，建议在指定了根目录后，不要再重复在 find_file()命令中指定搜索的根目录及根目录的父目录，否则，搜索顺序及目录将不确定。

- 2、指定根目录的方法

CMake 使用 CMAKE_FIND_ROOT_PATH 变量(默认为空)和 CMAKE_SYSROOT 变量来指定根目

录，但 CMAKE_SYSROOT 变量只能在 CMAKE_TOOLCHAIN_FILE 变量指定的工具链文件中设置。

3、搜索顺序

默认情况下，首先搜索 CMAKE_FIND_ROOT_PATH 中列出的目录，然后搜索 CMAKE_SYSROOT 目录，然后搜索非根目录。

4、可使用以下方法来控制是否启用此搜索规则

1)、设置 CMAKE_FIND_ROOT_PATH_MODE_INCLUDE 变量的值

- 如果设置为 ONLY，则只搜索 CMAKE_FIND_ROOT_PATH 中的根目录。
- 如果设置为 NEVER，那么 CMAKE_FIND_ROOT_PATH 中的根将被忽略，只使用主机系统根。
- 如果设置为 BOTH，那么将搜索主机系统路径和 CMAKE_FIND_ROOT_PATH 中的路径。

2)、在 find_file()命令中指定以下参数

- CMAKE_FIND_ROOT_PATH_BOTH: 按照 find_file()的搜索规则搜索。
- NO_CMAKE_FIND_ROOT_PATH: 不使用 CMAKE_FIND_ROOT_PATH 变量
- ONLY_CMAKE_FIND_ROOT_PATH: 只搜索 CMAKE_FIND_ROOT_PATH 变量和 CMAKE_STAGING_PREFIX 变量(分段前缀)以下的目录。注意：由于分段前缀还用于设置 CMAKE_SYSTEM_PREFIX_PATH(系统目录前缀)，因此，经测试，不建议将分段前缀与 CMAKE_FIND_ROOT_PATH 变量和系统目录前缀混合使用，以免产生不确定性；即，不要在 project()命令前设置分段前缀，然后又在之后设置 CMAKE_FIND_ROOT_PATH 变量的值，在这种情况下可能会产生搜索顺序的不确定性。

示例 12.4：搜索指定根目录下的子目录

①、将 c.cpp(可以为空文件)复制到 g:/qt1、g:/qt1/aa、g:/qt1/bb、g:/qt1/aa/include、g:/qt2、g:/qt2/aa、g:/qt3、g:/qt3/aa 目录。

②、在 g:/qt1 目录下的 CMakeLists.txt 中编写如下代码

#g:/qt1/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.27)
project(XXXX)
unset(AA CACHE) #清除缓存以便能重复搜索
set(CMAKE_FIND_ROOT_PATH G:/qt1) #指定根目录
set(CMAKE_STAGING_PREFIX g:/qt2) #在 project() 命令之后指定根目录，以避免设置系统目录前缀
#=====验证函数=====
function(f a b)
message("b=${b}" )
if(NOT b MATCHES xxx) #此 if 表示放弃所有搜索结果，本示例主要用于演示搜索顺序
    set(${a} 0 PARENT_SCOPE)
    message(=====)
endif()
endfunction()
#=====find_file()函数=====
find_file(AA NAMES c.cpp PATHS
    g:/qt1 #g:/qt1 已被指定为根目录，不建议在 find_file()重复指定
    g:/qt1/aa
```

```

g:/qt1/bb
g:/qt1/aa/include
#g:                                #不建议在 find_file() 重复指定根目录及其父目录
#g:/qt2                            #g:/qt2 已被指定为分段前缀, 也不建议重复指定
g:/qt2/aa
g:/qt3
g:/qt3/aa

ONLY_CMAKE_FIND_ROOT_PATH    #只搜索根目录和分段前缀
VALIDATOR f)                  #指定验证函数

message("AA=${AA}")

```

③、在 CMD 中转至 g:/qt1 并输入以下命令

cmake .

其结果如右图所示, 由结果可见, 只搜索了指定的根目

录 g:/qt1 和分段前缀的子目录 g:/qt2, 因此, g:/qt3、

g:/qt3/aa 未被搜索。

```

b=G:/qt1/aa/c.cpp
=====
b=G:/qt1/bb/c.cpp
=====
b=G:/qt1/aa/include/c.cpp
=====
b=G:/qt2/aa/c.cpp
=====
AA=AA-NOTFOUND

```

12.4 与搜索顺序有关的所有变量

12.4.1 开关变量

1、CMAKE_FIND_USE_PACKAGE_ROOT_PATH 变量

控制是否搜索由 <PackageName>_ROOT 变量提供的路径。默认无值, 相当于 TRUE。适用于 find_program()、find_library()、find_file()、find_path()、find_package() 命令, 由 find_* 指定的对应参数值(NO_PACKAGE_ROOT_PATH)优于该变量设置的值。

2、CMAKE_FIND_USE_CMAKE_PATH 变量

控制是否搜索由 CMake 特定缓存变量提供的路径。默认无值, 相当于 TRUE。适用于 find_program()、find_library()、find_file()、find_path()、find_package() 命令, 由 find_* 指定的对应参数值(即 NO_CMAKE_PATH)优于该变量设置的值。

3、CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH 变量

控制是否搜索由 CMake 特定环境变量提供的路径。默认无值, 相当于 TRUE。适用于 find_program()、find_library()、find_file()、find_path()、find_package() 命令, 由 find_* 指定的对应参数值(即 NO_CMAKE_ENVIRONMENT_PATH)优于该变量设置的值。

4、CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH 变量

控制是否搜索由标准系统环境变量提供的路径。默认无值, 相当于 TRUE。适用于 find_program()、find_library()、find_file()、find_path()、find_package() 命令, 由 find_* 指定的对应参数值(即

NO_SYSTEM_ENVIRONMENT_PATH)优于该变量设置的值。

5、CMAKE_FIND_USE_CMAKE_SYSTEM_PATH 变量

控制是否搜索由平台特定的 CMake 变量提供的路径。默认无值，相当于 TRUE。适用于 find_program()、find_library()、find_file()、find_path()、find_package()命令，由 find_*指定的对应参数值(即 NO_CMAKE_SYSTEM_PATH)优于该变量设置的值。

6、CMAKE_FIND_USE_INSTALL_PREFIX 变量

3.24 及以上版本。控制是否在 CMAKE_INSTALL_PREFIX 和 CMAKE_STAGING_PREFIX 变量中搜索位置。默认无值，相当于 TRUE。适用于 find_program()、find_library()、find_file()、find_path()、find_package()命令，由 find_*指定的对应参数值(即 NO_CMAKE_INSTALL_PREFIX)优于该变量设置的值。由于向后兼容 CMAKE_FIND_NO_INSTALL_PREFIX，find 命令的行为会根据这个变量是否存在而改变，如下表所示

CMAKE_FIND_USE_INSTALL_PREFIX	CMAKE_FIND_NO_INSTALL_PREFIX	是否搜索
未定义	ON	否
未定义	Off 未定义	是
Off	On	否
Off	Off 未定义	否
On	On	是
On	Off 未定义	是

7、CMAKE_FIND_NO_INSTALL_PREFIX 变量

若为 TRUE，则 CMAKE_SYSTEM_PREFIX_PATH 变量中不包含 CMAKE_INSTALL_PREFIX 和 CMAKE_STAGING_PREFIX 变量的值。注意：CMAKE_SYSTEM_PREFIX_PATH 变量在第一次调用 project()或 enable_language()命令时初始化，因此，必须在 project()之前设置 CMAKE_FIND_NO_INSTALL_PREFIX 变量的值才能生效。注意，CMAKE_FIND_NO_INSTALL_PREFIX 变量受到 CMAKE_FIND_USE_INSTALL_PREFIX 变量的影响。

12.42 路径变量

1、CMAKE_<LANG>_LIBRARY_ARCHITECTURE 或 CMAKE_LIBRARY_ARCHITECTURE

为<LANG>检测到的目标体系结构库目录名。如果<LANG>编译器将特定于体系结构的系统库搜索目录(如<prefix>/lib/<arch>)传递给链接器，则该变量包含由 CMake 检测到的<arch>名称。

2、CMAKE_PREFIX_PATH 变量及环境变量(搜索目录前缀)

该变量可以为 find_package()、find_program()、find_library()、find_file()和 find_path()命令设置一个搜索目录的前缀，以分号分隔。每个命令都会依各自的特点添加相应的子目录(如 bin、lib 或 include)。默认为空值。除此之外，还有一个相同功能的环境变量 CMAKE_PREFIX_PATH。比如，对于 find_file()命令，若 CMAKE_PREFIX_PATH=g:/qt1，则将添加 include 子目录，因此完整目录是 g:/qt1/include

3、CMAKE_INCLUDE_PATH 变量及环境变量

指定 find_file()和 find_path()命令的搜索路径的目录列表，以分号分隔。默认为空值。除此之外，还

有一个相同功能的环境变量 `CMAKE_INCLUDE_PATH`。

4、`CMAKE_SYSTEM_INCLUDE_PATH` 变量(无对应环境变量)

指定 `find_file()`和 `find_path()`命令的搜索路径的目录列表，以分号分隔。默认情况下，它包含当前系统的标准目录。该变量并不准备被项目修改，为此使用 `CMAKE_INCLUDE_PATH`。

5、`CMAKE_FRAMEWORK_PATH` 变量及环境变量(适用于 macOS)

指定 `find_library()`、`find_package()`、`find_path()`和 `find_file()`命令使用的 macOS 框架的搜索路径的目录列表，以分号分隔。除此之外，还有一个相同功能的环境变量 `CMAKE_FRAMEWORK_PATH`。

6、`CMAKE_SYSTEM_FRAMEWORK_PATH` 变量(适用于 macOS，无对应环境变量)

查找 `find_library()`、`find_package()`、`find_path()`、`find_file()`命令使用的 macOS 框架的路径。默认情况下，它包含当前系统的标准目录。该变量不准备被项目修改，为此使用 `CMAKE_FRAMEWORK_PATH`。

7、`CMAKE_SYSTEM_PREFIX_PATH` 变量(系统目录前缀，无对应环境变量)

1)、该变量可以为 `find_package()`、`find_program()`、`find_library()`、`find_file()`和 `find_path()`命令设置一个搜索目录的前缀，以分号分隔。每个命令都会依各自的特点添加相应的子目录(如 `bin`、`lib` 或 `include`)。比如，对于 `find_file()`命令，若 `CMAKE_SYSTEM_PREFIX_PATH=g:/qt1`，则将添加 `include` 子目录，因此完整目录是 `g:/qt1/include`

2)、默认情况下，该变量包含以下目录(注意，需根据使用的 `fin_*`命令添加适当的子目录)：

- 当前系统的系统目录，通常是安装软件的目录，因平台而异，比如，对于 Windows 系统可能是 `C:/Program Files`;`C:/Program Files (x86)`;
- 变量 `CMAKE_INSTALL_PREFIX`(安装前缀)指定的目录。
- 变量 `CMAKE_STAGING_PREFIX`(分段前缀)指定的目录。这个变量主要就是用来为 `find_*`指定搜索前缀的。

3)、可以通过在第一个 `project()`调用之前设置 `CMAKE_FIND_NO_INSTALL_PREFIX` 变量为 `TRUE` 来排除安装前缀和分段(staging)前缀。

4)、注意：由于 `CMAKE_SYSTEM_PREFIX_PATH` 变量在第一次调用 `project()`或 `enable_language()`命令时初始化，因此，必须在 `project()`命令之前设置 `CMAKE_FIND_NO_INSTALL_PREFIX`、`CMAKE_INSTALL_PREFIX`、`CMAKE_STAGING_PREFIX` 变量的值才能生效。

8、`CMAKE_STAGING_PREFIX` 变量(分段前缀，无对应环境变量)

该变量可被 `find_*`命令用作搜索前缀，可以通过设置 `CMAKE_FIND_NO_INSTALL_PREFIX` 变量来控制。如果任何传递给链接器的 `RPATH/RUNPATH` 表项包含 `CMAKE_STAGING_PREFIX`，则匹配的路径段(fragments)将被 `CMAKE_INSTALL_PREFIX` 替换。

9、`CMAKE_SYSROOT` 变量(无对应环境变量)

在 `--sysroot` 参数中传递给编译器的路径。如果支持的话，`CMAKE_SYSROOT` 内容通过 `--sysroot` 参数传递给编译器。如果安装时需要，也会从 `RPATH/RUNPATH` 中删除该路径。`CMAKE_SYSROOT` 也用于为 `find_*`命令添加搜索前缀。此变量只能在 `CMAKE_TOOLCHAIN_FILE` 变量指定的工具链文件中设置。

10、`CMAKE_FIND_ROOT_PATH` 变量(指定根目录，无对应环境变量)

需在文件系统上搜索的根路径列表，以分号分隔，CMake 使用此列表中的路径作为替代根，使用 **`find_package()`**，**`find_library()`**等命令查找文件系统项。默认为空。这个变量在交叉编译时非常有用。

11、CMAKE_FIND_ROOT_PATH_MODE_INCLUDE 变量

这个变量控制 CMAKE_FIND_ROOT_PATH 和 CMAKE_SYSROOT 是否被 find_file() 和 find_path() 使用。如果设置为 ONLY，则只搜索 CMAKE_FIND_ROOT_PATH 中的根目录。如果设置为 NEVER，那么 CMAKE_FIND_ROOT_PATH 中的根将被忽略，只使用主机系统根。如果设置为 BOTH，那么将搜索主机系统路径和 CMAKE_FIND_ROOT_PATH 中的路径。

12.5 其余的 find_* 命令

1、find_library() 命令

find_library() 命令只能用于查找静态库，其他文件(如 .cpp, .dll 等)该命令不能识别，除此之外与 find_file() 命令类似。

速记语法为：

```
find_library (<VAR> name1 [path1 path2 ...])
```

完整语法为：

```
find_library (
    <VAR>
    name | NAMES name1 [name2 ...] [NAMES_PER_DIR]
    [HINTS [path | ENV var]... ]
    [PATHS [path | ENV var]... ]
    [REGISTRY_VIEW (64|32|64_32|32_64|HOST|TARGET|BOTH)]
    [PATH_SUFFIXES suffix1 [suffix2 ...]]
    [VALIDATOR function]
    [DOC "cache documentation string"]
    [NO_CACHE]
    [REQUIRED]
    [NO_DEFAULT_PATH]
    [NO_PACKAGE_ROOT_PATH]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    [NO_SYSTEM_ENVIRONMENT_PATH]
    [NO_CMAKE_SYSTEM_PATH]
    [NO_CMAKE_INSTALL_PREFIX]
    [CMAKE_FIND_ROOT_PATH_BOTH |
    ONLY_CMAKE_FIND_ROOT_PATH |
    NO_CMAKE_FIND_ROOT_PATH]
)
```


2、find_path()命令

find_path()命令只能用于查找包含指定文件的路径,其结果是一个不含文件名的目录,如 g:/qt、g:/qt1/aa等。除此之外与 find_file()命令类似。

速记语法为:

```
find_path (<VAR> name1 [path1 path2 ...])
```

完整语法为:

```
find_path (
    <VAR>
    name | NAMES name1 [name2 ...]
    [HINTS [path | ENV var]... ]
    [PATHS [path | ENV var]... ]
    [REGISTRY_VIEW (64|32|64_32|32_64|HOST|TARGET|BOTH)]
    [PATH_SUFFIXES suffix1 [suffix2 ...]]
    [VALIDATOR function]
    [DOC "cache documentation string"]
    [NO_CACHE]
    [REQUIRED]
    [NO_DEFAULT_PATH]
    [NO_PACKAGE_ROOT_PATH]
    [NO_CMAKE_PATH]
    [NO_CMAKE_ENVIRONMENT_PATH]
    [NO_SYSTEM_ENVIRONMENT_PATH]
    [NO_CMAKE_SYSTEM_PATH]
    [NO_CMAKE_INSTALL_PREFIX]
    [CMAKE_FIND_ROOT_PATH_BOTH |
    ONLY_CMAKE_FIND_ROOT_PATH |
    NO_CMAKE_FIND_ROOT_PATH]
)
```

3、find_program()命令

find_program()命令用于查找程序。除此之外与 find_file()命令类似。

速记语法为:

```
find_program (<VAR> name1 [path1 path2 ...])
```

完整语法为:

```
find_program (
    <VAR>
    name | NAMES name1 [name2 ...] [NAMES_PER_DIR]
```

```
[HINTS [path | ENV var]... ]
[PATHS [path | ENV var]... ]
[REGISTRY_VIEW (64|32|64_32|32_64|HOST|TARGET|BOTH)]
[PATH_SUFFIXES suffix1 [suffix2 ...]]
[VALIDATOR function]
[DOC "cache documentation string"]
[NO_CACHE]
[REQUIRED]
[NO_DEFAULT_PATH]
[NO_PACKAGE_ROOT_PATH]
[NO_CMAKE_PATH]
[NO_CMAKE_ENVIRONMENT_PATH]
[NO_SYSTEM_ENVIRONMENT_PATH]
[NO_CMAKE_SYSTEM_PATH]
[NO_CMAKE_INSTALL_PREFIX]
[CMAKE_FIND_ROOT_PATH_BOTH |
 ONLY_CMAKE_FIND_ROOT_PATH |
 NO_CMAKE_FIND_ROOT_PATH]
)
```

4、find_package()命令用于查找 CMake 包，该命令详见下一章。

第 13 章 find_package()命令(查找包)

13.1 前提基础知识

13.1.1 何为包?

- 1、包可以简单的理解为含有一系列文件的集合，可将其类比为文件夹或压缩文件。因此，一个与程序有关的包通常是含有头文件(如.h)、静态库文件(如.lib)、动态库文件(如.dll)、与编译参数有关的文件、使用手册等文件的集合，所以，包的形式可以是一个文件夹、压缩文件、也可以是一个安装程序，只要其中包含有所需的文件即可，本文所指的包是以文件夹形式存在的包。由此可见，我们想要创建一个自己的包是非常容易的，只需将需要的文件放入一个文件夹中即可，该文件夹便是包。
- 2、包由谁提供？包通常由第三方提供，CMake 不提供包。当然，我们也可以自己创建一个包。

13.1.2 CMake 包

- 1、CMake 将含有以下名称的.cmake 文件称为包，本文将其统称为 **CMake 包**：
 - Find<PackageName>.cmake: 该文件被称为查找模块(包) (find module)
 - <lowercasePackageName>-config.cmake 或<PackageName>Config.cmake: 这些文件被称为配置文件(包)。
- 2、由以上内容可见，CMake 中的包与前一小节(13.1.1)提到的包的概念有混淆，为方便讲解，本文有时会混用包的概念，但尽量把查找模块(包)或配置文件(包)称为“CMake 包”。注意从上下文区分“包”具体指的什么“包”。
- 3、CMake 包的作用
查找模块或配置文件之所以被称为包，是因为这些.cmake 文件含有包中的.h、.lib、.dll 等文件的路径信息及其他信息(如编译参数)，因此，只需通过访问.cmake 文件便能知道包中的所有文件的路径以及其他信息，这样，就间接找到了包中的文件。这是 CMake 包的重要作用。
- 4、CMake 包怎样存储包的信息
传统的方法是将包的信息全部使用变量来存储，更现代的方法是尽量使用导入目标，使用导入目标的好处是可以传递使用要求。为了保持 CMake 包之间的一致性，所有 CMake 包中使用的变量最好使用统一的名称，这些名称被称为标准变量名，比如使用 xxx_FOUND(其中 xxx 表示包名)表示是否找到了包，使用 xxx_LIBRARY 表示库的路径等。

13.1.3 怎样编写 CMake 包

- 1、自己手动编写，最简单的 CMake 包是一个什么内容都没有的空包，当然，这样的包没有什么实际用处。
- 2、借助 CMake 提供的 CMakePackageConfigHelpers 模块中的命令生成一些基本的信息，然后手动补充

其他内容。

- 3、Linux 系统可借助 `pkg-config` 命令和 `.pc` 文件来生成 CMake 包。注：本文不会对 `pkg-config` 命令作讲解。
- 4、使用 CMake 提供的 `FetchContent` 模块中的命令从网上下载包，本文也不对此内容进行讲解。
- 5、通过依赖提供程序(dependency provider)提供。依赖提供程序可能是第三方包管理器，也可能是开发人员实现的自定义代码。依赖提供程序需要使用 `cmake_language(SET_DEPENDENCY_PROVIDER)` 命令，此命令本文也不进行讲解。

13.1.4 CMake 包由谁提供

- 1、CMake 包可以是提供包的第三方提供，也可以是对已经存在的包而特意编写的，也可以是由 CMake 官方提供的，当然，也可以自己编写。
- 2、配置文件(包)通常由提供包的上游提供，以帮助下游使用包，当然，也可以根据需要而自己编写。第三方为 CMake 提供库、头文件等的首选方式应该是提供配置文件。
- 3、查找模块通常是一种预先编写好的 CMake 包，不应由提供包的上游提供，而是由下游提供，比如，由操作系统提供、CMake 自带、甚至自己编写一个 `Find*.cmake` 文件。由于下游不提供包，所以，
 - 查找模块中指定的文件很容易过时。
 - 查找模块中文件的位置通常根据平台使用猜测的方式来估计需查找的文件的位置，因此，这些位置不一定有需要的文件存在。
 - CMake 拥有一些自带的为第三方提供的 `Find*.cmake` 文件，这些文件由 CMake 官方进行维护，因此，其关联的包可能落后于最新版本。

13.1.5 find_package()命令的作用

- 1、`find_package()`用于查找、检查、加载 CMake 包以及返回有关包状态的信息。
- 2、查找：由于 CMake 将含有包相关信息的 `.cmake` 文件称为包(注意：并不是所有以 `.cmake` 为后缀的文件)，所以，所谓 `find_package()`命令查找包，其实质是查找特定名称的 `.cmake` 文件(即 CMake 包)。
- 3、加载：`find_package()`命令不但会查找 `.cmake` 文件，还会加载该文件，可以将其简单理解为使用 `include()`将该文件包含进来，这样就可以在项目中使用 `.cmake` 文件中创建的与包有关的变量了。注意，这只是一种简单且不完全的理解方式，这样理解可将问题简化。
- 4、检查：`find_package()`命令除了读取和加载 `.cmake` 文件外，还会检查 `.cmake` 文件是否满足要求，比如版本要求是否满足。
- 5、返回信息：`find_package()`命令还会定义变量以返回有关包状态的信息以供 `.cmake` 文件或项目使用。

13.1.6 find_package()命令的格式及搜索模式

- 1、`find_package()`命令有两种格式：基本命令格式和完整命令格式，并且还有两种搜索模式：模块模式(Module mode)和配置模式(Config mode)。
- 2、模块模式(Module mode)
 - 1)、模块模式仅支持基本命令。
 - 2)、模块模式搜索以下的 `.cmake` 文件(即查找模块)。查找模块通常不由提供包的上游提供，而是由

下游提供。

Find<PackageName>.cmake

3)、模块模式搜索的路径及顺序为:

- 首先搜索 CMAKE_MODULE_PATH 变量中指定的路径(使用分号分隔, 使用正斜杠 “/”)。
- 然后搜索安装 CMake 程序时由 CMake 自带的查找模块(CMake 自带的 Find*.cmake 文件)的路径, 在该路径下提供了非常多的 Find*.cmake 文件, 其路径通常为 “...\\share\\cmake-3.27\\Modules\\” (该路径仅供参考, 视安装的 CMake 程序而定)

4)、模块模式返回的变量

模块模式不会自动设置<PackageName>_FOUND 变量的值, 该值需要由查找模块设置(即在.cmake 文件中手动设置)。同理, 模块模式也没有<PackageName>_DIR 变量。

3、配置模式(Config mode)

1)、基本命令和完整命令都支持配置模式。

2)、配置模式查找以下.cmake 文件(即配置文件), 配置文件通常由提供包的上游提供, 以帮助下游使用包。

- <lowercasePackageName>-config.cmake 或
- <PackageName>Config.cmake

若指定了版本详细信息, 还将查找以下.cmake 文件, 以下文件被称为**版本文件或配置版本文件**

- <lowercasePackageName>-config-version.cmake 或
- <PackageName>ConfigVersion.cmake。

3)、配置模式搜索的路径比模块模式复杂很多, 与 find_file()命令类似, 详见后文。需要注意的是, 无论哪种搜索模式, 默认情况下都不会搜索源目录、构建目录等目录。

4)、配置模式返回的变量

配置模式会自动设置一些包状态信息的变量, 这些变量可在项目中使用, 比如会自动设置<PackageName>_FOUND 变量以指示是否找到了包, 并且还可以在对应的.cmake 文件中设置该变量的值以控制是否找到了包; 会把<PackageName>_DIR 缓存变量设置为配置文件的位置。详细内容详见后文。

4、基本命令和完整命令

1)、基本命令首先在模块模式下搜索, 若未找到包则返回到配置模式, 可将变量

CMAKE_FIND_PACKAGE_PREFER_CONFIG 设置为 TRUE 来反转这种搜索情况, 基本命令还可以使用 MODULE 参数强制只使用模块模式搜索。

2)、完整命令只能使用配置模式搜索。

5、综上所述, find_package()命令并不会查找所有的以.cmake 为后缀的文件, 只会查找以下五种类型的.cmake 文件

- 查找以 Find 开始的.cmake 文件, 即 Find*.cmake
- 查找以 -config.cmake 结尾的文件, 即*-config.cmake
- 查找以 Config.cmake 结尾的文件, 即*Config.cmake
- 查找以 -config-version.cmake 结尾的文件, 即*-config-version.cmake
- 查找以 ConfigVersion.cmake 结尾的文件, 即*ConfigVersion.cmake

6、表 13.1 是对模块模式和配置模式的总结

表 13.1 模块模式和配置模式的区别

	模块模式	配置模式
支持的命令格式	基本命令	基本命令 完整命令
查找的文件名	Find<PackageName>.cmake (查找模块)	配置模式搜索以下配置文件： <ul style="list-style-type: none"> ● <lowercasePackageName>-config.cmake 或 ● <PackageName>Config.cmake 若指定了版本信息，还将查找以下版本文件 <ul style="list-style-type: none"> ● <lowercasePackageName>-config-version.cmake 或 ● <PackageName>ConfigVersion.cmake
搜索路径及顺序	1、CMAKE_MODULE_PATH 2、CMake 自带的查找模块的路径	配置模式的搜索路径比模块模式更复杂，与 find_file() 搜索的路径类似。详见后文
返回的变量	不会处理以下变量 <PackageName>_FOUND <PackageName>_DIR	会处理以下变量 <PackageName>_FOUND 表示是否找到了包 <PackageName>_DIR 表示配置文件的位置

13.2 find_package()基本命令

find_package()基本命令的语法为：

```
find_package(<PackageName>
    [version]                #指定版本号
    [EXACT]                  #精确匹配版本号
    [QUIET]                  #禁止显示消息
    [MODULE]                 #只使用模块模式搜索
    [REQUIRED]               #若未找到包则产生致命错误，并停止执行
    [[COMPONENTS] [components...]] #指定组件，且所有组件必须满足要求
    [OPTIONAL_COMPONENTS components...] #指定组件，但组件不需满足要求
    [REGISTRY_VIEW           #注册表相关，详见 find_file()
    (64|32|64_32|32_64|HOST|TARGET|BOTH)]
    [GLOBAL]                 #将导入目标提升到全局作用域(scope)
    [NO_POLICY_SCOPE]        #详见 cmake_policy()命令
    [BYPASS_PROVIDER])       #是否仅由依赖提供程序调用
```

- 1、基本命令搜索的文件名及路径见上一小节的表 13.1 所示(注：需视使用的搜索模式而定)。
- 2、默认情况下，基本命令首先在模块模式下搜索，若未找到包(即.cmake 文件)则返回到配置模式，可将

变量 `CMAKE_FIND_PACKAGE_PREFER_CONFIG` 设置为 `TRUE` 来反转这种搜索情况。

- 3、基本命令有一个显著的特点，就是不能明确指定搜索的路径，即没有 `PATHS` 或 `HINTS` 参数，因此，基本命令只能搜索模块模式和/或配置模式(视使用的搜索模式而定)的默认路径。无论哪种搜索模式，默认情况下都不会搜索源目录、构建目录等目录。
- 4、在配置模式下，`find_package()`命令会自动设置`<PackageName>_FOUND` 变量的值，以指示是否找到了`.cmake` 文件，并且可在对应的`.cmake` 文件中设置该变量的值以控制是否找到了包；在模块模式下，`<PackageName>_FOUND` 变量的值需在对应的`.cmake` 文件中手动设置。配置模式还会把`<PackageName>_DIR` 缓存变量设置为配置文件的位置，而模块模式则不会。
- 5、各参数意义如下：

1)、MODULE 参数

表示只能使用模块模式搜索，不能回退到配置模式。若不指定该参数，则优先使用模块模式搜索，若未找到包，则使用配置模式搜索

2)、[version]参数

用于指定与包兼容的版本号。由于模块模式不支持查找版本文件，因此，指定版本号意味着需要进入配置模式，但是，虽然指定了版本号，基本命令仍会按照默认的处理方式，先按模块模式搜索 `Find<PackageName>.cmake` 文件，若未找到该文件才会按配置模式处理版本问题。版本文件可自己创建，也可使用 CMake 自带的模块 `CMakePackageConfigHelpers` 创建。有关版本文件的使用详见 13.3.5。可指定两种形式的版本号：

- 指定单一版本号：格式为 `major[.minor[.patch[.tweak]]]`，即“主版本号[.次版本号[.补丁[.调整]]”，比如 2.3.12.41 等
- 指定版本范围：适用于 3.19 及以上版本。其格式为 `versionMin...[<]versionMax`，其中 `versionMin` 和 `versionMax` 与单一版本号的格式相同，默认情况下包括两个端点，若指定了“<”，则将排除上端点。

3)、EXACT 参数

表示精确匹配版本号，因此，使用此参数，则不能使用版本范围来指定版本号。

4)、QUIET 参数

表示禁止显示消息。正常情况下，CMake 会显示一些查找包过程的消息，使用此参数将禁止显示这些消息。

5)、REQUIRED 参数

表示若未找到包则停止执行并产生错误消息；若不指定该参数，则即使未找到包也会继续执行。每个非 `REQUIRED` 的 `find_package()`调用可以通过设置以下变量来启用或禁用 `REQUIRED`

- 将 `CMAKE_REQUIRE_FIND_PACKAGE_<PackageName>` 变量设置为 `TRUE`，将使该包为 `REQUIRED`
- 将 `CMAKE_DISABLE_FIND_PACKAGE_<PackageName>` 设置为 `TRUE`，将禁用该包，同时也禁止重定向到 `FetchContent`(一个 CMake 提供的模块)提供的包
- 同时将以上两个变量设置为 `TRUE` 是错误的。

6)、COMPONENTS 参数

用于指定包的组件(或分组)。通常，一个包可以包含多个组件，若指定了该参数则这些组件中的任何一个不满足要求，则整个包都会被认为没有找到。注意：组件是否满足要求需在`.cmake` 文件中设置，包是否有组件取决于包的提供方。若指定了 `REQUIRED` 参数，则可省略 `COMPONENTS` 参数并在 `REQUIRED` 后直接指定所需的组件。有关组件的内容详见 13.3.6。

7)、OPTIONAL_COMPONENTS 参数

也用于指定组件，但这些组件若不能满足要求，则仍然认为找到了整个包。

8)、REGISTRY_VIEW 及其子参数

适用于 3.24 及以上版本，用于指定查询哪些注册表视图，该关键字只适用于 Windows 平台，在其他平台上将被忽略，如何解释注册表视图信息取决于目标包。该参数与 `find_file()` 命令的同名参数意义相同，详见 `find_file()` 命令(第 12 章)。

9)、GLOBAL 参数

用于将导入的目标的作用范围提升到全局范围，也可使用设置变量 `CMAKE_FIND_PACKAGE_TARGETS_GLOBAL` 来达到此目的。

10)、NO_POLICY_SCOPE 参数，参见 `cmake_policy()` 命令。

11)、BYPASS_PROVIDER 参数

适用于 3.24 及以上版本，表示仅允许在依赖提供程序(dependency provider)调用 `find_package()`。详见 `cmake_language()` 命令 CMake 官方参考手册(本文不讲解此命令)。

6、表 13.2 列出了与基本命令有关的部分变量，其余变量详见后文“包文件接口变量”。

表 13.2 `find_package()` 基本命令部分变量

变量名	说明
<code>CMAKE_MODULE_PATH</code>	指定模块模式搜索的路径，以分号分隔，使用正斜杠"/"。默认为空
<code>CMAKE_FIND_PACKAGE_PREFER_CONFIG</code>	设置为 TRUE 将使基本命令先按配置模式搜索再按模块模式搜索
<code>CMAKE_FIND_PACKAGE_TARGETS_GLOBAL</code>	设置为 TRUE 将使 <code>find_package()</code> 命令的所有导入目标提升到 GLOBAL 作用域。默认为 OFF
<code>CMAKE_REQUIRE_FIND_PACKAGE_<PackageName></code>	设置为 TRUE，将使该包为 REQUIRED
<code>CMAKE_DISABLE_FIND_PACKAGE_<PackageName></code>	设置为 TRUE，将禁用该包
<code><PackageName>_FOUND</code>	指示是否找到了包。模块模式不会自动设置该变量的值，需在 .cmake 文件中设置。

示例 13.1: `find_package()` 基本命令-----模块模式加载包

①、准备包

在目录 `g:/qt1` 中的 `a.cpp` 中编写如下代码

```
//g:/qt1/a.cpp(用作包)
#include<iostream>
extern int b;
int main(){    std::cout<<"E="<<b<<std::endl;    return 0;}
```

在目录 `g:/qt1` 中的 `b.cpp` 中编写如下代码

```
//g:/qt1/b.cpp(用作包)
int b=1;
```

②、准备 CMake 包----查找模块

在另一个目录如 g:/qt2 目录下的 FindAA.cmake 文件中编写如下代码

```
#g:/qt2/FindAA.cmake(查找模块)

message(====这是 FindAA.cmake====)

#创建变量 xx1 和 xx2 保存源文件的路径

set(xx1 g:/qt1/a.cpp)

set(xx2 g:/qt1/b.cpp)

message("====结束 FindAA.cmake====")
```

③、准备 CMake 包----配置文件

在 g:/qt2 目录下的 AAConfig.cmake 文件中编写如下代码

```
#g:/qt2/AAConfig.cmake (配置文件)

message("====这是 AAConfig.cmake====")

set(xx1 g:/qt1/a.cpp)

set(xx2 g:/qt1/b.cpp)

message("====结束 AAConfig.cmake====")
```

④、在 g:/qt2 目录下的 CMakeLists.txt 中编写如下代码

```
#g:/qt2/CMakeLists.txt

cmake_minimum_required(VERSION 3.27)

project(XXXX)

set(CMAKE_MODULE_PATH g:/qt2)      #设置模块模式搜索路径

set(AA_DIR g:/qt2)                  #指定配置模式搜索路径

#查找并加载.cmake 文件，首先搜索模块模式，若未找到，再搜索配置模式

find_package( AA )

#将 Find*.cmake 文件中变量 xx1 和 xx2 指定的路径作为 add_executable() 命令的源文件路径

add_executable(nn ${xx1} ${xx2} )
```

⑤、在 CMD 中转至 g:/qt2 并输入以下命令

cmake .

注意：若要多次执行 cmake 命令测试结果，建议把每次执行 cmake 命令后生成的文件删除后再重新执行 cmake 命令，特别应删除 CMakeCache.txt 文件，否则，某些数据不会更新(因为可能会存在缓存变量)。

⑥、执行以上命令后，会有类似以下的信息输出，这说明本示例查找到了 FindAA.cmake 文件，而未查找到 AAConfig.cmake 文件，并且 find_file() 命令会加载 FindAA.cmake 文件。

```
====这是 FindAA.cmake====
====结束 FindAA.cmake====
```

⑦、接着在 CMD 中输入以下命令

```
mingw32-make
nn.exe          #测试结果，符合预期
```

示例 13.2: find_package() 基本命令----配置模式加载包

①、文件准备。仍然使用上一示例的文件，即使用 g:/qt1/a.cpp、g:/qt1/b.cpp、g:/qt2/FindAA.cmake、g:/qt2/AAConfig.cmake 四个文件

②、在 g:/qt2 目录下的 CMakeLists.txt 文件中编写如下代码

```
#g:/qt2/CMakeLists.txt

cmake_minimum_required(VERSION 3.27)
project(XXXX)

set(CMAKE_MODULE_PATH g:/qt2)           #指定模块模式搜索路径
set(AA_DIR g:/qt2)                       #指定配置模式搜索路径
#设置以下变量为 TRUE 以使 find_package() 命令首先按配置模式搜索.cmake 文件
set(CMAKE_FIND_PACKAGE_PREFER_CONFIG 1)
find_package( AA )

#find_package(AA PATHS g:/qt2)           #注意：该命令是完整命令而不是基本命令
#所需的 a.cpp 和 b.cpp 文件的路径位于配置文件中的变量 xx1 和 xx2 中
add_executable(nn ${xx1} ${xx2} )
```

③、在 CMD 中转至 g:/qt2 并输入以下命令

```
cmake .
```

执行以上命令后，会有类似以下的信息输出，这说明本示例查找到了 AAConfig.cmake 文件，而未查找到 FindAA.cmake 文件，并且 find_file() 命令会加载.cmake 文件。

```
====这是 AAConfig.cmake=====
====结束 AAConfig.cmake=====
```

④、接着在 CMD 中输入以下命令

```
mingw32-make
nn.exe           #测试结果，符合预期
```

13.3 find_package()完整命令

13.3.1 find_package()完整命令语法及基本参数

find_package()完整命令语法为：

```
find_package(<PackageName>
    [version]           #指定版本号
    [EXACT]             #精确匹配版本号
    [QUIET]             #禁止显示消息
    [REQUIRED]          #若未找到包则产生致命错误，并停止执行
    [[COMPONENTS] [components...]] #指定组件，且所有组件必须满足要求
    [OPTIONAL_COMPONENTS components...] #指定组件，但组件可以不需满足要求
    [CONFIG|NO_MODULE]  #二者效果等同。强制使用纯配置模式
    [GLOBAL]            #将导入目标提升到全局作用域(scope)
    [NO_POLICY_SCOPE]   #详见 cmake_policy() 命令
```

[BYPASS_PROVIDER]	#是否仅由依赖提供程序调用
[NAMES name1 [name2 ...]]	#指定要搜索的包名, 忽略<PackageName>
[CONFIGS config1 [config2 ...]]	#搜索指定的文件名, 必须是.cmake 后缀
[HINTS path1 [path2 ...]]	#指定配置模式搜索路径
[PATHS path1 [path2 ...]]	#同上
[REGISTRY_VIEW (64 32 64_32 32_64 HOST TARGET BOTH)]	#注册表相关, 详见 find_file()
[PATH_SUFFIXES suffix1 [suffix2 ...]]	#为搜索路径指定一个后缀(即子目录)
[NO_DEFAULT_PATH]	#禁用默认搜索路径
[NO_PACKAGE_ROOT_PATH]	
[NO_CMAKE_PATH]	
[NO_CMAKE_ENVIRONMENT_PATH]	
[NO_SYSTEM_ENVIRONMENT_PATH]	
[NO_CMAKE_PACKAGE_REGISTRY]	
[NO_CMAKE_BUILDS_PATH]	# Deprecated; does nothing.
[NO_CMAKE_SYSTEM_PATH]	
[NO_CMAKE_INSTALL_PREFIX]	
[NO_CMAKE_SYSTEM_PACKAGE_REGISTRY]	
[CMAKE_FIND_ROOT_PATH_BOTH ONLY_CMAKE_FIND_ROOT_PATH NO_CMAKE_FIND_ROOT_PATH])	

- 完整命令搜索的文件名见 13.1.6 小节的表 13.1(注: 需视使用的搜索模式而定)。配置模式的搜索路径和顺序详见 13.3.2 和 13.3.3 小节。
- 配置模式将提供以下变量(由 find_package()命令创建)以返回有关查找.cmake 文件的信息
 - <PackageName>_DIR: 这是一个缓存变量, 存储配置文件的目录。该变量可在配置文件和项目中使用。
 - <PackageName>_CONFIG: 配置文件的完整路径。该变量可在项目中使用, 不能在配置文件中
 - <PackageName>_CONSIDERED_CONFIG: 搜索包的适当版本时所考虑的所有配置文件。该变量可在项目中使用, 不能在配置文件中
 - <PackageName>_CONSIDERED_VERSIONS: 上一变量的相关版本文件。该变量可在项目中使用, 不能在配置文件中
- 其中, 参数 version、EXACT、QUIET、REQUIRED、COMPONENTS、OPTIONAL_COMPONENTS、GLOBAL、NO_POLICY_SCOPE、BYPASS_PROVIDER、REGISTRY_VIEW 与基本命令相同, 详见 find_package()基本命令(13.2 小节)。
- CONFIG 和 NO_MODULE 参数
二者意义相同, 都表示强制使用纯配置模式。纯配置模式会跳过模块模式搜索, 立即进行配置模式搜索。
- NAMES 参数

默认情况下，完整命令搜索的包名称为<PackageName>，若指定了 NAMES 则搜索指定的包名而忽略 <PackageName>。

6、CONFIGS 参数

搜索指定的文件名，指定的文件名必须是.cmake 后缀。使用该参数可以不搜索默认配置文件名 <PackageName>Config.cmake 或 <lowercasePackageName>-config.cmake。比如

```
find_package( BB PATHS g:/qt2 CONFIGS YY.cmake)
```

将搜索 YY.cmake，而不搜索 BBConfig.cmake 或 bb-config.cmake 文件。

7、PATHS、HINTS 参数

指定配置模式下搜索.cmake 文件的路径。

8、PATH_SUFFIXES 参数

该参数为搜索目录指定一个后缀(即子目录)，比如

```
find_package(AA PATHS g:/qt1 g:/qt2 PATH_SUFFIXES xx )
```

将按顺序在 g:/qt1、g:/qt1/xx、g:/qt2、g:/qt2/xx 目录下搜索.cmake 文件。

再如

```
find_file(AA PATHS g:/qt1 PATH_SUFFIXES xx yy)
```

将按顺序在 g:/qt1、g:/qt1/xx、g:/qt1/yy 目录下搜索.cmake 文件

9、NO_XXX_PATH 参数

与搜索路径和顺序有关，表示忽略由相应变量指定的路径，详见 13.3.3。

13.3.2 配置模式的搜索目录

- 1、CMake 会创建一个搜索前缀(或路径)<prefix>，并在这些前缀下搜索表 13.3 所示目录，说简单点就是，CMake 会搜索设置的搜索路径<prefix>以及路径<prefix>下的子目录(由表 13.3 指定的子目录)。其中

- <prefix>是设置的搜索路径(如 PATHS 参数设置的路径)，完整的搜索路径详见 13.3.3。
- <name>是指的 CMake 搜索的包名称即<PackageName>或 NAMES 参数指定的名称，且不区分大小写。<name>*是指的以<name>开头的任意名称。
- 表 13.3 中的目录仅仅是一种使用上的习惯，不是强制规定。所以，对于所有 Windows 和 Unix 的目录仍然会在所有平台上搜索。Apple 的目录用于在 Apple 平台上，CMAKE_FIND_FRAMEWORK 和 CMAKE_FIND_APPBUNDLE 变量决定了优先顺序。

- 2、若设置了 CMAKE_LIBRARY_ARCHITECTURE 变量，则启用 lib/<arch>路径，其中，lib*可能是 lib64、lib32、libx32 或 lib 中的一个或多个，按以下顺序搜索

- 若 FIND_LIBRARY_USE_LIB64_PATHS 全局属性设置为 TRUE，则在 64 位平台上搜索 lib64 路径。
- 若 FIND_LIBRARY_USE_LIB32_PATHS 全局属性设置为 TRUE，则在 32 位平台上搜索 lib32 路径。
- 若 FIND_LIBRARY_USE_LIBX32_PATHS 全局属性设置为 TRUE，则在 x32 位平台上搜索 libx32 路径。
- 总是搜索 lib 路径。

3、注册表相关(仅限 Windows)

- 在 3.24 及以上版本，且在 Windows 平台上，可搜索注册表，在其他平台上将被忽略。其规则为
- 使用专用语法将注册表查询作为 HINTS 和 PATHS 参数指定的目录的一部分。可以使用 REGISTRY_VIEW 及其子参数来管理 HINTS 和 PATHS 的一部分指定的 Windows 注册表查询。REGISTRY_VIEW 及其子参数与 find_file()命令的相同参数意义相同，详见 find_file()命令(第 12 章了)。

表 13.3 find_package()命令配置模式搜索目录

搜索目录	适用平台
<prefix>/	Windows
<prefix>/(<cmake CMake)/	Windows
<prefix>/<name>*/	Windows
<prefix>/<name>*/(<cmake CMake)/	Windows
<prefix>/<name>*/(<cmake CMake)/<name>*/	Windows
<prefix>/(<lib/<arch> lib* share)/cmake/<name>*/	Unix
<prefix>/(<lib/<arch> lib* share)/<name>*/	Unix
<prefix>/(<lib/<arch> lib* share)/<name>*/(<cmake CMake)/	Unix
<prefix>/<name>*/ (<lib/<arch> lib* share)/cmake/<name>*/	Windows/Unix
<prefix>/<name>*/ (<lib/<arch> lib* share)/<name>*/	Windows/Unix
<prefix>/<name>*/ (<lib/<arch> lib* share)/<name>*/(<cmake CMake)/	Windows/Unix
<prefix>/<name>.framework/Resources/	Apple
<prefix>/<name>.framework/Resources/CMake/	Apple
<prefix>/<name>.framework/Versions/*/Resources/	Apple
<prefix>/<name>.framework/Versions/*/Resources/Cmake/	Apple
<prefix>/<name>.app/Contents/Resources/	Apple
<prefix>/<name>.app/Contents/Resources/Cmake/	Apple

示例 13.3: find_package()配置模式的搜索目录

①、准备配置文件

在 G:/qt2/share/ccaeefge 目录下的 CCConfig.cmake 文件中编写如下代码

```
#G:/qt2/share/ccaeefge/CCConfig.cmake
message("====这是 g:/qt2/share/cc*/CCConfig.cmake=====")
```

在 G:/qt2/ddfertg/cmake 目录下的 DDConfig.cmake 文件中编写如下代码

```
#G:\qt2\ddfertg\cmake\DDConfig.cmake
message("====这是 g:/qt2/dd*/cmake/DDConfig.cmake=====")
```

②、在 g:/qt2 目录下的 CMakeLists.txt 文件中编写如下代码

```
#g:/qt2/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
find_package(CC PATHS g:/qt2 )      #指定搜索前缀
find_package(DD PATHS g:/qt2 )      #指定搜索前缀
```

③、在 CMD 中转至 g:/qt2 并输入以下命令

```
cmake .
```

执行以上命令后，会有类似以下的信息输出，这说明本示例在 G:/qt2(前缀)的子目录 share/cc*和 dd*/cmake 下搜索.cmake 文件。本示例是在 Windows10 系统上运行的，这说明，在 Windows 系统上也搜索了 Unix 下的目录 share/cc*，因为，表 13.3 中的目录仅仅是一种使用上的习惯，不是强制规定。

```
=====这是 g:/qt2/share/cc*/CCConfig.cmake=====
=====这是 g:/qt2/dd*/cmake/DDConfig.cmake=====
```

13.3.3 配置模式的搜索前缀及搜索顺序

若指定了 NO_DEFAULT_PATH，则表示禁用默认搜索路径，即，相当于指定了所有的 NO_*，否则按以下顺序搜索，需注意以下问题：

- <PackageName>_DIR 变量是由 find_package()命令创建的。
- 搜索路径的绝大部分与 find_file()相同，其中各变量的含义可参考 find_file()命令(第 12 章)。

1、首先搜索 CMAKE_FIND_PACKAGE_REDIRECTS_DIR 变量指定的目录中搜索.cmake 文件，即使在模块模式下也是如此。适用于 3.24 及以上版本。CMAKE_FIND_PACKAGE_REDIRECTS_DIR 是个只读变量，其主要作用是用于与 FetchContent_MakeAvailable()集成，FetchContent_MakeAvailable()命令可能会在变量 CMAKE_FIND_PACKAGE_REDIRECTS_DIR 指定的目录中创建文件。

2、搜索包根目录

- <PackageName>_ROOT，包名<PackageName>保留大小写。
- <PACKAGENAME>_ROOT，包名<PACKAGENAME>是大写的，适用于 3.27 及以上版本。
- ENV{<PackageName>_ROOT}，包名<PackageName>保留大小写。
- ENV{<PACKAGENAME>_ROOT}，包名<PACKAGENAME>是大写的，适用于 3.27 及以上版本。

若是从查找模块中调用，则父模块的查找模块的根路径在当前包的路径之后搜索。即

- <CurrentPackage>_ROOT
- ENV{<CurrentPackage>_ROOT}
- <ParentPackage>_ROOT
- ENV{<ParentPackage>_ROOT}等

跳过此搜索规则的方法

- 在 find_package()命令中指定 NO_PACKAGE_ROOT_PATH 参数
- 将变量 CMAKE_FIND_USE_PACKAGE_ROOT_PATH 设置为 FALSE

3、搜索缓存变量

按顺序搜索以下 CMake 特定缓存变量中指定的路径。这些变量可以在命令行中使用-DVAR=value 指定，这些值将被解释为以分号分隔的列表。

- CMAKE_PREFIX_PATH(搜索目录前缀)。
- CMAKE_FRAMEWORK_PATH (适用于 macOS)
- CMAKE_APPBUNDLE_PATH(适用于 macOS)

②、跳过此搜索规则的方法

- 在 find_package()命令中指定 NO_CMAKE_PATH 参数
- 将 CMAKE_FIND_USE_CMAKE_PATH 变量的值设为 FALSE

4、搜索环境变量

①、按顺序搜索以下的特定于 CMake 的环境变量中指定的路径。

- <PackageName>_DIR(由 **find_package()**命令创建)
- CMAKE_PREFIX_PATH(搜索目录前缀)
- CMAKE_FRAMEWORK_PATH(适用于 macOS)
- CMAKE_APPBUNDLE_PATH(适用于 macOS)

②、跳过此搜索规则的方法

- 在 **find_package()**命令中指定 NO_CMAKE_ENVIRONMENT_PATH 参数
- 将 CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH 变量的值设为 FALSE

5、搜索由 HINTS 参数指定的路径。

6、搜索标准系统环境变量

①、搜索标准系统环境变量 PATH。以/bin 或/sbin 结尾的路径自动转换为它们的父目录。

②、跳过此搜索规则的方法

- 在 **find_package()**命令中指定了 NO_SYSTEM_ENVIRONMENT_PATH 参数
- 将 CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH 变量的值设为 FALSE

7、搜索存储在 CMake 用户包注册表中的路径。若在 **find_package()**命令中指定了 NO_CMAKE_PACKAGE_REGISTRY 参数或将 CMAKE_FIND_USE_PACKAGE_REGISTRY 变量的值设为 FALSE，则可跳过此搜索规则

8、搜索系统目录

①、按顺序搜索当前系统的平台文件中定义的以下 CMake 变量。这些变量通常是安装软件的位置

- CMAKE_SYSTEM_PREFIX_PATH(系统目录前缀)。该变量默认情况下包括以下路径
 - 当前系统的系统目录，通常是安装软件的目录，因平台而异，比如，对于 Windows 系统可能是 C:/Program Files;C:/Program Files (x86);
 - 变量 CMAKE_INSTALL_PREFIX(安装前缀)指定的目录。
 - 变量 CMAKE_STAGING_PREFIX(分段前缀)指定的目录。

注意：搜索目录时实际搜索的是 CMAKE_SYSTEM_PREFIX_PATH 变量指定的路径，安装前缀和分段前缀只可以间接改变 CMAKE_SYSTEM_PREFIX_PATH 变量的部分值，搜索目录时并不以安装前缀和分段前缀指定的路径为准来搜索，搜索时需把这两个变量的值传递给 CMAKE_SYSTEM_PREFIX_PATH 变量才会搜索他们设置的路径。

注意：由于 CMAKE_SYSTEM_PREFIX_PATH 变量在第一次调用 **project()**或 **enable_language()**命令时初始化，因此，安装前缀、分段前缀、CMAKE_FIND_NO_INSTALL_PREFIX 变量的值必须在 **project()**之前设置才能生效。

- CMAKE_SYSTEM_FRAMEWORK_PATH。适用于 macOS。
- CMAKE_SYSTEM_APPBUNDLE_PATH。适用于 macOS。

②、跳过此搜索规则的方法

- 若在 **find_package()**命令中指定了 NO_CMAKE_INSTALL_PREFIX，或者将 CMAKE_FIND_USE_INSTALL_PREFIX 变量设置为 FALSE，则可以跳过 CMAKE_INSTALL_PREFIX 和 CMAKE_STAGING_PREFIX 的查找。
- 若在 **find_package()**命令中指定了 NO_CMAKE_SYSTEM_PATH，或者将 CMAKE_FIND_USE_CMAKE_SYSTEM_PATH 设置为 FALSE，则可以跳过此处的所有搜索规则。

9、搜索存储在 CMake 系统包注册表中的路径。若在 **find_package()**命令中指定了

NO_CMAKE_SYSTEM_PACKAGE_REGISTRY 参数或将

CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY 变量的值设为 FALSE，则可跳过此搜索规则

10、搜索由 PATHS 参数指定的路径。

11、CMAKE_IGNORE_PATH, CMAKE_IGNORE_PREFIX_PATH, CMAKE_SYSTEM_IGNORE_PATH 和 CMAKE_SYSTEM_IGNORE_PREFIX_PATH 变量也可以导致忽略上述一些位置。

12、表 13.4 是对 find_package() 命令配置模式搜索顺序的总结，由于 find_package() 命令的搜索顺序和规则与 find_file() 命令类似，详细示例请参阅 find_file() 命令(第 12 章)，从略。

表 13.4 find_package() 命令配置模式的搜索顺序

注：<PackageName>_ROOT 和 <PackageName>_DIR 是由 find_package() 命令创建的变量

搜索顺序	规则	说明
禁用搜索	搜索路径	若指定 NO_DEFAULT_PATH 参数表示禁用默认搜索路径，即相当于指定了所有的 NO_* 参数，否则按以下顺序搜索
1、3.24 新增	搜索路径	CMAKE_FIND_PACKAGE_REDIRECTS_DIR 主要作用是用于与 FetchContent_MakeAvailable() 集成
2、包根目录	搜索路径	<PackageName>_ROOT <PACKAGENAME>_ROOT ENV{<PackageName>_ROOT} ENV{<PACKAGENAME>_ROOT}
	跳过搜索的方法	CMAKE_FIND_USE_PACKAGE_ROOT_PATH 变量设为 FALSE 指定 NO_PACKAGE_ROOT_PATH 参数
3、缓存变量	搜索路径	CMAKE_PREFIX_PATH(搜索目录前缀) CMAKE_FRAMEWORK_PATH(适用于 macOS) CMAKE_APPBUNDLE_PATH(适用于 macOS)
	跳过搜索的方法	CMAKE_FIND_USE_CMAKE_PATH 变量设为 FALSE 指定 NO_CMAKE_PATH 参数
4、环境变量	搜索路径	<PackageName>_DIR CMAKE_PREFIX_PATH(搜索目录前缀) CMAKE_FRAMEWORK_PATH(适用于 macOS) CMAKE_APPBUNDLE_PATH(适用于 macOS)
	跳过搜索的方法	CMAKE_FIND_USE_CMAKE_ENVIRONMENT_PATH 变量设为 FALSE 指定 NO_CMAKE_ENVIRONMENT_PATH 参数
5、HINTS 参数	搜索路径	指定的路径
6、标准系统环境变量	搜索路径	PATH 以/bin 或/sbin 结尾的路径自动转换为它们的父目录。
	跳过搜索的方法	CMAKE_FIND_USE_SYSTEM_ENVIRONMENT_PATH 变量设为 FALSE 指定 NO_SYSTEM_ENVIRONMENT_PATH 参数
7、用户注册表	搜索路径	存储在 CMake 用户包注册表中的路径。
	跳过搜索的方法	CMAKE_FIND_USE_PACKAGE_REGISTRY 变量的值设为 FALSE 指定 NO_CMAKE_PACKAGE_REGISTRY 参数
8、系统目录	搜索路径	①、CMAKE_SYSTEM_PREFIX_PATH(系统目录前缀)。 该变量默认情况下包括以下路径

		<ul style="list-style-type: none"> ● 当前系统的系统目录，通常是安装软件的目录，因平台而异，比如，对于 Windows 系统可能是 C:/Program Files;C:/Program Files (x86); ● 变量 CMAKE_INSTALL_PREFIX(安装前缀)指定的目录。 ● 变量 CMAKE_STAGING_PREFIX(分段前缀)指定的目录。 <p>②、CMAKE_SYSTEM_FRAMEWORK_PATH (适用于 macOS)</p> <p>③、CMAKE_SYSTEM_APPBUNDLE_PATH (适用于 macOS)</p>
	跳过搜索的方法	<p>①、以下方法跳过查找 CMAKE_INSTALL_PREFIX 和 CMAKE_STAGING_PREFIX</p> <ul style="list-style-type: none"> ● CMAKE_FIND_USE_INSTALL_PREFIX 变量设为 FALSE ● 指定 NO_CMAKE_INSTALL_PREFIX 参数 <p>②、以下方法跳过此处的所有搜索规则</p> <ul style="list-style-type: none"> ● CMAKE_FIND_USE_CMAKE_SYSTEM_PATH 变量设为 FALSE ● 指定 NO_CMAKE_SYSTEM_PATH 参数
9、系统包注册表	搜索路径	搜索存储在 CMake 系统包注册表中的路径。
	跳过搜索的方法	CMAKE_FIND_USE_SYSTEM_PACKAGE_REGISTRY 变量设为 FALSE 指定 NO_CMAKE_SYSTEM_PACKAGE_REGISTRY 参数
10、PATHS 参数	搜索路径	指定的路径
11、其他	无	CMAKE_IGNORE_PATH, CMAKE_IGNORE_PREFIX_PATH, CMAKE_SYSTEM_IGNORE_PATH 和 CMAKE_SYSTEM_IGNORE_PREFIX_PATH 变量也可以导致忽略上述一些位置。

13.3.4 搜索指定根目录下的子目录

- 1、该小节的原理与 find_file()的原理相同，详细示例可参阅 find_file()命令(第 12 章)。
- 2、CMake 将这里指定的这个根目录称为 re-root(重生根目录)。比如，有一系列目录 g:/qt1/aa, g:/qt1/bb, g:/qt2/xx, 若只搜索指定的根目录 g:/qt1, 则只会搜索 g:/qt1/aa、g:/qt1/bb。注意：经测试，建议在指定了根目录后，不要再重复在 find_file()命令中指定搜索根目录及根目录的父目录，否则，搜索顺序及目录将不确定。
- 3、CMake 使用 CMAKE_FIND_ROOT_PATH 变量(默认为空)和 CMAKE_SYSROOT 变量来指定根目录，但 CMAKE_SYSROOT 变量只能在 CMAKE_TOOLCHAIN_FILE 变量指定的工具链文件中设置。
- 4、搜索顺序：默认情况下，首先搜索 CMAKE_FIND_ROOT_PATH 中列出的目录，然后搜索 CMAKE_SYSROOT 目录，然后搜索非根目录。
- 5、可使用以下方法来控制是否启用此搜索规则
 - ①、设置 CMAKE_FIND_ROOT_PATH_MODE_INCLUDE 变量的值
 - 如果设置为 ONLY, 则只搜索 CMAKE_FIND_ROOT_PATH 中的根目录。
 - 如果设置为 NEVER, 那么 CMAKE_FIND_ROOT_PATH 中的根将被忽略，只使用主机系统根。
 - 如果设置为 BOTH, 那么将搜索主机系统路径和 CMAKE_FIND_ROOT_PATH 中的路径。
 - ②、在 find_file()命令中指定以下参数
 - CMAKE_FIND_ROOT_PATH_BOTH: 按照 find_package()的搜索规则搜索。
 - NO_CMAKE_FIND_ROOT_PATH: 不使用 CMAKE_FIND_ROOT_PATH 变量

- **ONLY_CMAKE_FIND_ROOT_PATH**: 只搜索 **CMAKE_FIND_ROOT_PATH** 变量和 **CMAKE_STAGING_PREFIX** 变量(分段前缀)以下的目录。注意: 由于分段前缀还用于设置 **CMAKE_SYSTEM_PREFIX_PATH**(系统目录前缀), 因此, 经测试, 不建议将分段前缀与 **CMAKE_FIND_ROOT_PATH** 变量和系统目录前缀混合使用, 以免产生不确定性; 即, 不要在 **project()**命令前设置分段前缀, 然后又在之后设置 **CMAKE_FIND_ROOT_PATH** 变量的值, 在这种情况下可能会产生搜索顺序的不确定性。

13.3.5 处理版本信息及版本文件变量

- 1、只要存在版本文件, 无论有无指定[version]参数, CMake 都会首先读取与配置文件匹配的版本文件, 以测试是否与请求的版本兼容, 若版本文件判断为兼容, 则接受该配置文件。也就是说, 配置文件的版本是否兼容是在版本文件中进行判断的。
- 2、当指定了[version]参数时, 配置模式将只查找与请求版本兼容的配置文件, 若给出了 EXACT 参数, 则只会查找与请求版本完全匹配的配置文件, 若没有可用的版本文件, 则假定配置文件不与任何版本兼容。
- 3、需要注意的是, 版本是否兼容需自己手动在版本文件中设置, CMake 不负责设置, 也就是说, 完全可以不按照以上要求来编写代码, 但建议按以上规则处理。
- 4、版本文件可以自己创建, 也可使用 CMake 自带的模块 **CMakePackageConfigHelpers** 创建。
- 5、**find_package()**命令加载版本文件时, 创建以下变量(本文将其称为**版本文件变量**)并执行以下步骤

1)、设置基本信息

首先设置以下变量的值(由 **find_package()**命令自动设置)。版本文件使用以下变量来检查是否与请求的版本兼容或精确匹配。当版本文件执行完成, **find_package()**命令就会清除这些变量的作用域, 这意味着, 这些变量只能在版本文件中使用, 不能在项目中使用也不能在配置文件中使用。

①、全局信息

- **PACKAGE_FIND_NAME**: 包名<PackageName>
- **PACKAGE_FIND_VERSION_COMPLETE**: 无论是指定单个版本还是范围版本, 此变量都会被定义, 并保存为请求的完整的版本号(单一或范围版本号)

②、单一版本

若指定的是范围版本, 则以下变量将保存为范围版本的下限值(即下Endpoint)

- **PACKAGE_FIND_VERSION**: 请求的完整的单一版本号
- **PACKAGE_FIND_VERSION_MAJOR**: 请求的主版本号, 否则为 0
- **PACKAGE_FIND_VERSION_MINOR**: 请求的次版本号, 否则为 0
- **PACKAGE_FIND_VERSION_PATCH**: 请求的补丁号, 否则为 0
- **PACKAGE_FIND_VERSION_TWEAK**: 请求的微调版本号, 否则为 0
- **PACKAGE_FIND_VERSION_COUNT**: 版本分组个数, 取值为 0~4

③、范围版本

- **PACKAGE_FIND_VERSION_RANGE**: 请求的完整的范围版本号
- **PACKAGE_FIND_VERSION_RANGE_MIN**: 是否包含范围版本的下限(下Endpoint), 目前仅支持 INCLUDE(包含下限)
- **PACKAGE_FIND_VERSION_RANGE_MAX**: 是否包含版本范围的上限(上Endpoint), 支持 INCLUDE(包含)或 EXCLUDE(不包含)
- **PACKAGE_FIND_VERSION_MIN**: 请求的完整的范围下限版本号

- PACKAGE_FIND_VERSION_MIN_MAJOR: 请求的下端点的主版本号, 否则为 0
- PACKAGE_FIND_VERSION_MIN_MINOR: 请求的下端点的次版本号, 否则为 0
- PACKAGE_FIND_VERSION_MIN_PATCH: 请求的下端点的补丁号, 否则为 0
- PACKAGE_FIND_VERSION_MIN_TWEAK: 请求的下端点的微调版本号, 否则为 0
- PACKAGE_FIND_VERSION_MIN_COUNT: 请求的下端点的版本分组数, 取值范围为 0~4
- PACKAGE_FIND_VERSION_MAX: 请求的完整的范围上限版本号
- PACKAGE_FIND_VERSION_MAX_MAJOR: 请求的上端点的主版本号, 否则为 0
- PACKAGE_FIND_VERSION_MAX_MINOR: 请求的上端点的次版本号, 否则为 0
- PACKAGE_FIND_VERSION_MAX_PATCH: 请求的上端点的补丁号, 否则为 0
- PACKAGE_FIND_VERSION_MAX_TWEAK: 请求的上端点的微调版本号, 否则为 0
- PACKAGE_FIND_VERSION_MAX_COUNT: 请求的上端点的版本分组数, 取值范围为 0~4

2)、检测版本是否可接受

版本文件通过设置以下变量的值(需在版本文件中手动设置), 以确定是否提供可接受的版本。当版本文件执行完成, `find_package()`命令就会清除这些变量的作用域, 这意味着, 这些变量只能在版本文件中使用, 不能在项目中使用也不能在配置文件中使用。

- PACKAGE_VERSION: 提供的完整版本字符串。注意, 这个变量的值是找到的可接受的版本号, 所以, 该变量的值将影响<PackageName>_CONSIDERED_VERSIONS 变量的值及以稍后将讲到的返回的<PackageName>_VERSION 和<PackageName>_VERSION_*变量的值。
- PACKAGE_VERSION_EXACT: 若版本完全匹配, 则为 True
- PACKAGE_VERSION_COMPATIBLE: 若版本兼容, 则为 True
- PACKAGE_VERSION_UNSUITABLE: 若不适用于任何版本, 则为 True

3)、返回相关信息

若版本是可接受的, 则设置以下变量的值(由 `find_package()`命令自动设置), 以供项目和配置文件中使⤵用, 并加载相应的配置文件。注意: 以下变量是在版本文件检测之后由 `find_package()`命令返回给调用者使用的变量, 因此, 这些变量不能在版本文件中使用。

- <PackageName>_VERSION: 提供的完整版本字符串
- <PackageName>_VERSION_MAJOR: 提供的主要版本号, 否则为 0。
- <PackageName>_VERSION_MINOR: 提供的次要版本号, 否则为 0。
- <PackageName>_VERSION_PATCH: 提供的补丁号, 否则为 0。
- <PackageName>_VERSION_TWEAK: 提供的微调版本号, 否则为 0。
- <PackageName>_VERSION_COUNT: 版本分组数, 取值范围为 0~4

4)、当有多个配置文件可用时(即, 有多个版本兼容), 则不指定选择哪一个版本, 因此, 不会尝试选择最高或最接近的版本号, 除非设置了 CMAKE_FIND_PACKAGE_SORT_ORDER 变量。可使用以下两个变量控制兼容性的顺序:

- CMAKE_FIND_PACKAGE_SORT_ORDER
- CMAKE_FIND_PACKAGE_SORT_DIRECTION

示例 13.4: find_package()命令--使用版本文件

①、准备包

在 `g:/qt1` 目录下的 `a.cpp` 中编写如下代码

```
//g:/qt1/a.cpp(用作包)
#include<iostream>
extern int b;
```

```
int main(){      std::cout<<"E="<<b<<std::endl;      return 0;}
```

在 g:/qt1 目录下的 b.cpp 中编写如下代码，

```
//g:/qt1/b.cpp(用作包)
int b=1;
```

②、准备版本文件

在另一个目录如 g:/qt2 目录下的 AAConfigVersion.cmake 文件中编写如下代码

#g:/qt2/AAConfigVersion.cmake(版本文件)

```
message(\n=====这是 AAConfigVersion.cmake=====)
#以下 message 语句用于测试，以验证各变量的输出值以及这些值由谁设置的，读者可删除这些内容
message(\n*****自动设置版本信息--全局信息*****)
message("包名=           ${PACKAGE_FIND_NAME}")
message("完整版本号=       ${PACKAGE_FIND_VERSION_COMPLETE}")

message(\n*****自动设置版本信息--单一版本*****)
message("完整的单一版本号=  ${PACKAGE_FIND_VERSION}")
message("单一主版本号=      ${PACKAGE_FIND_VERSION_MAJOR}")
message("单一主版本号=      ${PACKAGE_FIND_VERSION_MINOR}")
message("单一补丁号=        ${PACKAGE_FIND_VERSION_PATCH}")
message("单一微调版本号=     ${PACKAGE_FIND_VERSION_TWEAK}")
message("单一版本分组数=     ${PACKAGE_FIND_VERSION_COUNT}")

message(\n*****自动设置版本信息--范围版本*****)
message("完整的范围版本号=  ${PACKAGE_FIND_VERSION_RANGE}")
message("是否包含下端点=     ${PACKAGE_FIND_VERSION_RANGE_MIN}")
message("是否包含上端点=     ${PACKAGE_FIND_VERSION_RANGE_MAX}")
message("完整的范围下限版本号= ${PACKAGE_FIND_VERSION_MIN}")
message("下限主版本号=       ${PACKAGE_FIND_VERSION_MIN_MAJOR}")
message("下限次版本号=       ${PACKAGE_FIND_VERSION_MIN_MINOR}")
message("下限补丁号=         ${PACKAGE_FIND_VERSION_MIN_PATCH}")
message("下限微调版本号=     ${PACKAGE_FIND_VERSION_MIN_TWEAK}")
message("下限版本分组数=     ${PACKAGE_FIND_VERSION_MIN_COUNT}")

message("\n完整的范围上限版本号=${PACKAGE_FIND_VERSION_MAX}")
message("上限主版本号=       ${PACKAGE_FIND_VERSION_MAX_MAJOR}")
message("上限次版本号=       ${PACKAGE_FIND_VERSION_MAX_MINOR}")
message("上限补丁号=         ${PACKAGE_FIND_VERSION_MAX_PATCH}")
message("上限微调版本号=     ${PACKAGE_FIND_VERSION_MAX_TWEAK}")
message("上限版本分组数=     ${PACKAGE_FIND_VERSION_MAX_COUNT}")

message(\n*****需手动设置--确认是否可接受*****)
message("最终可接受的版本号=${PACKAGE_VERSION}")
message("版本是否精确匹配=   ${PACKAGE_VERSION_EXACT}")
message("版本是否兼容=       ${PACKAGE_VERSION_COMPATIBLE})")
```

```

message("版本不适用=    ${PACKAGE_VERSION_UNSUITABLE}")

message(\n*****自动设置版本返回信息*****)
message("返回最终的版本号=    ${AA_VERSION}")
message("返回主版本号=    ${AA_VERSION_MAJOR}")
message("返回次版本号=    ${AA_VERSION_MINOR}")
message("返回补丁号=    ${AA_VERSION_PATCH}")
message("返回微调版本号=    ${AA_VERSION_TWEAK}")
message("返回版本分组数=    ${AA_VERSION_COUNT}")

#以下语句需自己手动编写，以判断版本是否兼容以及设置最终兼容的版本号是多少
#set(PACKAGE_VERSION_EXACT 1)                #若完全匹配则为 1，对本示例此要求可有可无
if(PACKAGE_FIND_VERSION VERSION_GREATER_EQUAL 3.0.0)    #若版本大于等于 3.0.0
    set(PACKAGE_VERSION_COMPATIBLE 1)                #该版本兼容
    set(PACKAGE_VERSION 3.2.1)                        #设置兼容的最终版本号
else()
    set(PACKAGE_VERSION_COMPATIBLE 0)    #否则该版本不兼容，若不兼容则不会加载配置文件
endif()

message(\n=====结束 AAConfigVersion=====\n)

```

③、准备配置文件

在 g:/qt2 目录下的 AAConfig.cmake 文件中编写如下代码

```

#g:/qt2/AAConfig.cmake (配置文件)

message(\n\n====这是 AAConfig.cmake=====)
set(xx1 g:/qt1/a.cpp)
set(xx2 g:/qt1/b.cpp)
message(====结束 AAConfig.cmake=====\\n\\n)

```

④、在 g:/qt2 目录下的 CMakeLists.txt 文件中编写如下代码

```

#g:/qt2/CMakeLists.txt

cmake_minimum_required(VERSION 3.27)
project(XXXX)
find_package(AA 3.2.1...6.5.4 PATHS g:/qt2 )
#所需的 a.cpp 和 b.cpp 文件的路径位于配置文件中的变量 xx1 和 xx2 中
add_executable(nn ${xx1} ${xx2} )

message(\n====这是 CMakeLists.txt=====)
message("返回最终的版本号=    ${AA_VERSION}")
message("返回主版本号=    ${AA_VERSION_MAJOR}")
message("返回次版本号=    ${AA_VERSION_MINOR}")
message("返回补丁号=    ${AA_VERSION_PATCH}")
message("返回微调版本号=    ${AA_VERSION_TWEAK}")
message("返回版本分组数=    ${AA_VERSION_COUNT}")
message(====结束 CMakeLists.txt=====\\n)

```

⑤、在 CMD 中转至 g:/qt2 并输入以下命令

cmake .

⑥、执行以上命令后，会有类似如图 13.1 的信息输出。

A、CMake 首先调用了版本文件，然后根据版本文件中对变量

```
PACKAGE_VERSION_EXACT、
PACKAGE_VERSION_COMPATIBLE、
PACKAGE_VERSION_UNSUITABLE
```

设置的值以确定配置文件的版本是否是可接受的(兼容或完全匹配)，若有可接受的版本，则将最终可接受的版本存储在 PACKAGE_VERSION 变量中返回给调用方。

B、从输出的信息可以看到，返回给调用方的变量，在版本文件中不可用，但这些变量可在项目(或配置文件)中使用。返回的这些变量的值与变量 PACKAGE_VERSION 的值是一致的，且由 find_package()自动设置。

C、除返回给调用方的变量和需手动设置以判断版本兼容性的变量外，其余变量的值均由 find_package()自动设置，且只能在版本文件中使用。

⑦、接下来，继续在 CMD 中输入以下命令

mingw32-make

nn.exe #测试，结果符合预期

```
=====这是AAConfigVersion.cmake=====
*****自动设置版本信息--全局信息*****
包名= AA
完整版本号= 3.2.1...6.5.4

*****自动设置版本信息--单一版本*****
完整的单一版本号= 3.2.1
单一主版本号= 3
单一次版本号= 2
单一补丁号= 1
单一微调版本号= 0
单一版本分组数= 3

*****自动设置版本信息--范围版本*****
完整的范围版本号= 3.2.1...6.5.4
是否包含下端点= INCLUDE
是否包含上端点= INCLUDE
完整的范围下限版本号= 3.2.1
下限主版本号= 3
下限次版本号= 2
下限补丁号= 1
下限微调版本号= 0
下限版本分组数= 3

完整的范围上限版本号= 6.5.4
上限主版本号= 6
上限次版本号= 5
上限补丁号= 4
上限微调版本号= 0
上限版本分组数= 3

*****需手动设置--确认是否可接受*****
最终可接受的版本号=
版本是否精确匹配=
版本是否兼容=
版本不适用=

*****自动设置版本返回信息*****
返回最终的版本号=
返回主版本号=
返回次版本号=
返回补丁号=
返回微调版本号=
返回版本分组数=

=====结束AAConfigVersion=====

=====这是AAConfig.cmake=====
=====结束AAConfig.cmake=====

=====这是CMakeLists.txt=====
返回最终的版本号= 3.2.1
返回主版本号= 3
返回次版本号= 2
返回补丁号= 1
返回微调版本号= 0
返回版本分组数= 3
=====结束CMakeLists.txt=====
```

图 13.1 示例 13.4 输出的部分内容

13.3.6 包(文件)接口变量

- 1、包(文件)接口变量由 `find_package()` 命令创建，用来返回有关包状态的信息以供配置文件使用(注意：项目和版本文件都不能使用)，通常以 `<PackageName>` 开头。包文件接口变量具有以下特点
 - 包接口变量的值在返回之前被恢复到原始状态，也就是说，这些包接口变量只在配置文件中有值，因此，在项目中或版本文件中不能使用包接口变量的值。
 - 在模块模式中，加载的查找模块(即 CMake 自带的 `Find*.cmake` 文件)负责处理这些变量。在配置模式中，`find_package()` 命令会自动处理 `REQUIRED`、`QUIET`、`[version]` 参数，但组件留给配置文件处理，配置文件可以将 `<PackageName>_FOUND` 设置为 `false`，以告诉 `find_package` 组件需求未被满足。以上内容可简单理解为：模块模式下，包接口变量的值需在查找模块中手动设置。配置模式下，包接口变量的值会由 `find_package()` 命令自动设置(但组件除外)，并且可在配置文件中设置 `<PackageName>_FOUND` 的值以控制组件是否满足要求。
- 2、CMake 定义的包接口变量如下：

包接口变量中与版本有关的参数与版本文件中的意义相同，通用参数取值决定于 `find_package()` 命令是否指定了相关参数。

①、通用参数

- `CMAKE_FIND_PACKAGE_NAME`：被搜索的包名 `<PackageName>`
- `<PackageName>_FIND_REQUIRED`：若指定了 `REQUIRED` 参数，则为 `True`
- `<PackageName>_FIND_QUIETLY`：若指定了 `QUIET` 参数，则为 `True`
- `<PackageName>_FIND_REGISTRY_VIEW`：若指定了 `REGISTRY_VIEW` 参数，则为 `True`
- `<PackageName>_FIND_COMPONENTS`：指定的组件(含可选组件)
- `<PackageName>_FIND_REQUIRED_<c>`：若组件 `<c>` 是必需的，则为 `True`，若是可选的，则为 `false`
- `<PackageName>_FIND_VERSION_COMPLETE`：无论是指定单个版本还是范围版本，此变量都会被定义，并保存为请求的完整的版本号(单一或范围版本号)

②、单一版本参数

若指定的是范围版本，则以下变量将保存为范围版本的下限值(即下端点)。

- `<PackageName>_FIND_VERSION`：请求的完整的单一版本号
- `<PackageName>_FIND_VERSION_MAJOR`：请求的主版本号，否则为 0
- `<PackageName>_FIND_VERSION_MINOR`：请求的次版本号，否则为 0。
`<PackageName>_FIND_VERSION_PATCH`：请求的被丁号，否则为 0
- `<PackageName>_FIND_VERSION_TWEAK`：请求的微调版本号，否则为 0
- `<PackageName>_FIND_VERSION_COUNT`：版本分组个数，取值为 0~4
- `<PackageName>_FIND_VERSION_EXACT`：若指定了 `EXACT` 参数，则为 `True`

③、范围版本参数

- `<PackageName>_FIND_VERSION_RANGE`：请求的完整的范围版本号
- `<PackageName>_FIND_VERSION_RANGE_MIN`：是否包含范围版本的下限(下端点)，目前仅支持 `INCLUDE`(包含下限)
- `<PackageName>_FIND_VERSION_RANGE_MAX`：是否包含版本范围的上限(上端点)，支持 `INCLUDE`(包含)或 `EXCLUDE`(不包含)

- <PackageName>_FIND_VERSION_MIN: 请求的完整的范围下限版本号
- <PackageName>_FIND_VERSION_MIN_MAJOR: 请求的下端点的主版本号, 否则为 0
- <PackageName>_FIND_VERSION_MIN_MINOR: 请求的下端点的次版本号, 否则为 0
- <PackageName>_FIND_VERSION_MIN_PATCH: 请求的下端点的补丁号, 否则为 0
- <PackageName>_FIND_VERSION_MIN_TWEAK: 请求的下端点的微调版本号, 否则为 0
- <PackageName>_FIND_VERSION_MIN_COUNT: 请求的下端点的版本分组数, 取值范围为 0~4
- <PackageName>_FIND_VERSION_MAX: 请求的完整的范围上限版本号
- <PackageName>_FIND_VERSION_MAX_MAJOR: 请求的上端点的主版本号, 否则为 0
- <PackageName>_FIND_VERSION_MAX_MINOR: 请求的上端点的次版本号, 否则为 0
- <PackageName>_FIND_VERSION_MAX_PATCH: 请求的上端点的补丁号, 否则为 0
- <PackageName>_FIND_VERSION_MAX_TWEAK: 请求的上端点的微调版本号, 否则为 0
- <PackageName>_FIND_VERSION_MAX_COUNT: 请求的上端点的版本分组数, 取值范围为 0~4

示例 13.5: find_package()命令----指定组件

①、准备包: 使用上一示例在 g:/qt1 目录下的 a.cpp 和 b.cpp

②、准备配置文件

在另一个目录如 g:/qt2 目录下的 AAConfig.cmake 文件中编写如下代码

#g:/qt2/AAConfig.cmake(配置文件)

```
message(\n\n====这是 AAConfig.cmake=====\n\n)
set(EE 1)                                #用作开关
set(LL ss tt)                            #将已有的组件保存在变量 LL 中

foreach(PP ${AA_FIND_COMPONENTS})      #将 find_package()命令指定的组件逐个赋值给 PP
    if(NOT PP IN_LIST LL)                #若 PP 不位于 LL 中, 则将开关设为 0, 并跳出循环
        set(EE 0)
        break()
    endif()
endforeach()

if( EE )                                #若开关 EE 为真, 则添加源文件目录, 否则组件不满足要求
    set(xx1 g:/qt1/a.cpp)
    set(xx2 g:/qt1/b.cpp)
else()
    set(AA_FOUND 0)                      #表示未找到包
endif()
message(====结束 AAConfig.cmake=====\n\n)
```

③、在 g:/qt2 目录下的 CMakeLists.txt 文件中编写如下代码

#g:/qt2/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.27)
project(XXXX)
find_package(AA PATHS g:/qt2 COMPONENTS ss tt )    #指定组件 ss 和 tt
add_executable(nn ${xx1} ${xx2} )
```

④、在 CMD 中转至 g:/qt2 并输入以下命令


```
cmake .
mingw32-make
nn.exe          #测试，符合预期
```

13.3.7 find_package()命令中各变量小结

- 1、版本文件变量的值除少部分外，都是由 find_package()自动设置，除用于返回信息的变量外，其余变量只能在版本文件中使用。
- 2、包接口变量的值若在模块模式，则需在查找模块中手动设置；在配置模式，则由 find_package()自动设置(组件除外)。包接口变量只能在配置文件或查找模块中使用。
- 3、find_package()返回的能在项目中使用的的所有变量如下：

1)、通用变量

- <PackageName>_FOUND：指示是否找到了包。模块模式不会自动设置该变量的值，需在 Find*.cmake 文件中手动设置；配置模式可在配置文件中设置<PackageName>_FOUND 的值以控制组件是否满足要求。
- <PackageName>_CONFIG：配置文件的完整路径。该变量可在项目中使用，不能在配置文件中使用时。
- <PackageName>_CONSIDERED_CONFIG：搜索包的适当版本时所考虑的所有配置文件。该变量可在项目中使用，不能在配置文件中使用时。
- <PackageName>_CONSIDERED_VERSIONS：上一变量的相关版本文件。该变量可在项目中使用，不能在配置文件中使用时。

2)、与版本有关的变量。以下变量来自版本文件的返回值。

- <PackageName>_FOUND：可以根据该变量的值为判断是否成功找到了包，若为 TRUE 则表示找到了包(仅限配置模式)。
- <PackageName>_DIR：配置文件的位置(缓存变量，仅限配置模式)。
- <PackageName>_VERSION：提供的完整版本字符串
- <PackageName>_VERSION_MAJOR：提供的主要版本号，否则为 0。
- <PackageName>_VERSION_MINOR：提供的次要版本号，否则为 0。
- <PackageName>_VERSION_PATCH：提供的补丁号，否则为 0。
- <PackageName>_VERSION_TWEAK：提供的微调版本号，否则为 0。
- <PackageName>_VERSION_COUNT：版本分组数，取值范围为 0~4

4、可以使用以下方法来判断 CMake 包中定义了哪些变量及其值

使用 CMake 自带的 FindPackageHandleStandardArgs 模块中的 find_package_handle_standard_args 命令，其方法如下(详细内容请参阅 CMake 帮助文档)：

```
include(FindPackageHandleStandardArgs)
find_package_handle_standard_args(AA "xxxx" aa bb)
```

- 若以上命令位于配置文件中，则表示，在 AAConfig.cmake 中，若变量 aa、bb 未定义值，则认为未找到包，变量 AA_FOUND 被设置为 FALSE；若变量 aa、bb 都定义了，则认为找到了包，变量 AA_FOUND 被设置为 TRUE。
- 若以上命令位于 CMakeLists.txt 文件中，则可检测变量 aa、bb 是否在 AAConfig.cmake 中被设置，若有值，若未设置值，则会显示信息"xxxx"。

13.4 使用 CMake 自带的模块 CMakePackageConfigHelpers 生成配置文件和版本文件

CMakePackageConfigHelpers 是 CMake 自带的模块，该模块中有如下两个命令，分别用于生成配置文件和版本文件。

`configure_package_config_file()`: 用于生成配置文件
`write_basic_package_version_file`: 用于生成版本文件

13.4.1 `configure_package_config_file()`命令

`configure_package_config_file()`命令的语法如下：

```
configure_package_config_file(  
    <input> <output>  
    INSTALL_DESTINATION <path>  
    [PATH_VARS <var1> <var2> ... <varN>]  
    [NO_SET_AND_CHECK_MACRO]  
    [NO_CHECK_REQUIRED_COMPONENTS_MACRO]  
    [INSTALL_PREFIX <path>] )
```

1、`configure_package_config_file()`命令的作用与 `configure_file()`命令类似，都是使用一个输入文件 `<input>` 根据一些替换规则来生成一个输出文件 `<output>`，其替换规则与 `configure_file()`命令相同，详见 `configure_file()`命令(第 11 章)。使用 `configure_package_config_file()`的好处是有助于生成可重定位的包。

2、各参数意义如下：

- `<input>`表示输入文件
- `<output>`表示输出文件
- 剩余的参数与“确定生成的配置文件和包中文件的相对位置”有关。

3、下面讲解怎样确定配置文件和包中文件的相对位置关系

1)、以包中的源文件为例，假设 `<prefix>`表示安装位置，`<current>`表示配置文件所在位置，`<source>`表示包中的源文件所在位置，并假设配置文件位于以下位置

$$\text{<current>} = \text{<prefix>/a/b/c}$$

包中的源文件位于以下位置

$$\text{<source>} = \text{<prefix>/d/e/f}$$

2)、因为 CMake 可以获取到配置文件所在位置(即，该位置是已知的)，因此，我们的目的是需要知道在 `CMakeLists.txt` 文件中如何从已知的配置文件的位置获取到包中文件的位置。从以上目录结构可见，只需将配置文件当前所在位置向上返回三级目录到达安装前缀之后再追加的一个相对

路径，便能找到源文件所在位置，即

```
<source> = <current>/../../d/e/f
```

这样，追加的相对路径 `d/e/f` 始终相对于安装前缀所在位置，无论配置文件位于何处，都是如此，但有个前提“配置文件和源文件拥有相同的安装前缀”，而安装前缀通常在 `CMakeLists.txt` 文件中指定，因此，安装前缀也是已知的。因此，根据以上原理，若要在 `CMakeLists.txt` 文件中确定源文件的位置(即，包中文件的位置)，只需指定一个相对路径和安装前缀即可，而不必在乎配置文件位于何处，这样，包便可随配置文件一起移动，从而达到可重定位的目的。

4、`configure_package_config_file()`命令怎样实现以上原理的方法

1)、CMake 使用以下变量来获取配置文件所在位置

```
CMAKE_CURRENT_LIST_DIR
```

以上变量的作用是获取当前正在处理的列表文件所在的目录，其值是动态的。

2)、使用以下方法来指定安装前缀

使用 `CMAKE_INSTALL_PREFIX` 变量或 `INSTALL_PREFIX` 参数

3)、计算需要返回几级目录

A、使用 `INSTALL_DESTINATION` 参数和安装前缀共同确定需要向上返回几级目录。返回上级目录的层数是 `INSTALL_DESTINATION` 参数的目录层数与安装前缀的目录层数之差，比如，假设

```
安装前缀 = g:/qt1  
INSTALL_DESTINATION = g:/qt1/a/b/c
```

由于 `INSTALL_DESTINATION` 参数指定的目录比安装前缀多三层目录，所以，需向上返回三级目录。从此处可见，二者指定的目录必须是父子关系，且安装前缀必须是父目录，否则，将可能无法获取到包中文件的正确位置。注：`INSTALL_DESTINATION` 参数可使用相对路径，相对路径相对于安装前缀。

B、从以上原理可见，`INSTALL_DESTINATION` 指定的目录只需要以安装前缀为根目录即可，这意味着安装前缀和 `INSTALL_DESTINATION` 参数可任意指定，只需满足父子关系即可，但是，建议将“安装前缀+`INSTALL_DESTINATION` 参数”组成的位置指定为实际生成的输出文件的位置。

C、注意：从以上原理可见，`INSTALL_DESTINATION` 并不是输出文件的实际安装目的地，CMake 并不会将输出文件自动安装到 `INSTALL_DESTINATION` 参数指定的位置去，因此，这个位置仅仅是输出文件理论上应该安装到的位置，实际上输出文件在这个位置上并不一定真实存在，需另想办法把输出文件复制或移动到这个位置去，其方法有以下几种：

- 直接在 `<output>` 参数中以完整路径的形式指定输出文件
- 将输出文件手动复制或移动到理论安装位置去
- 使用 `install()` 命令安装到理论安装位置

4)、使用参数 `PATH_VARS` 指定的变量 `<var>` 的值来指定包中文件的相对位置。

5)、通过以上设置，包中文件所在位置为

```
<source> = ${CMAKE_CURRENT_LIST_DIR}/../../...../<var>
```

其中，返回上级目录的层数是 INSTALL_DESTINATION 参数指定的目录与安装前缀的目录层数之差。

示例 13.6：使用 `configure_package_config_file()` 命令自动生成配置文件

①、准备输入文件

在目录 `g:/qt2` 下的输入文件 `tt.in` 文件(文件名可任意，无强制规定)中编写如下代码。该文件中只有一行代码，本示例的目的是演示以下代码自动生成的输出文件的内容。

#g:/qt2/tt.in(输入文件)

```
@PACKAGE_INIT@
```

②、在目录 `g:/qt2` 下的 `CMakeLists.txt` 中编写如下代码

#g:/qt2/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.27)
project(XXXX)
#包含 CMake 自带的模块
include(CMakePackageConfigHelpers)
#以下命令是 CMakePackageConfigHelpers 模块中的命令
configure_package_config_file(
    tt.in
    tt.cmake
    INSTALL_DESTINATION g:/qt2 ) #注意：此参数并不是输出文件的安装目的地
```

③、CMake 代码分析

以上代码的 `INSTALL_DESTINATION` 参数指定的是输出文件(配置文件)的理论安装位置，这个位置应是安装前缀的子目录，用于与安装前缀共同计算需要向上返加几级目录，否则，可能会无法定位包中文件的位置。由于本示例的 `INSTALL_DESTINATION` 参数不是安装前缀子目录，所以最终会生成一个不正确的包位置。

④、在 CMD 中转至 `g:/qt2` 并输入以下命令

```
cmake .
```

⑤、分析生成的输出文件

可以在 `g:/qt2`(本示例输出文件的生成位置)中找到生成的 `tt.cmake` 文件，该文件有如下内容

#g:/qt2/tt.cmake

```
##### Expanded from @PACKAGE_INIT@ by configure_package_config_file() #####
#####从@PACKAGE_INIT@扩展为 configure_package_config_file() #####
##### Any changes to this file will be overwritten by the next CMake run #####
##### 对该文件的任何更改都将被下一次 CMake 运行覆盖 #####
##### The input file was tt.in #####

get_filename_component(PACKAGE_PREFIX_DIR
    "${CMAKE_CURRENT_LIST_DIR}/C:/Program Files (x86)/XXXX" ABSOLUTE)

macro(set_and_check _var _file) #创建一个宏
    set(${_var} "${_file}")
    if(NOT EXISTS "${_file}") #如果指定的目录或文件存在，则为真
```

```

        message(FATAL_ERROR "File or directory ${_file} referenced by variable ${_var}
does not exist !")
    endif()
endmacro()

macro(check_required_components _NAME)           #创建一个宏
    foreach(comp ${${_NAME}_FIND_COMPONENTS})    #将指定的组件循环赋值给 comp
        if(NOT ${_NAME}_${comp}_FOUND)
            if(${_NAME}_FIND_REQUIRED_${comp})    #若组件 comp 是必须的, 则为 true
                set(${_NAME}_FOUND FALSE)         #将包设置为未找到
            endif()
        endif()
    endforeach()
endmacro()

#####

```

A、可以看到，只需在输入文件中使用一行代码@PACKAGE_INIT@便生成了以上这么多的内容。

B、以上代码总共有三条命令，如下：

- 创建一个 `set_and_check` 宏，该宏有两个参数，然后调用了 `set()` 命令，并对 `set()` 命令第二个参数指定的文件或目录的存在性进行测试，说简单一点，`set_and_check` 宏就是一个对变量的指定的路径或文件的存在性进行简单测试的 `set()` 命令。
- 创建一个 `check_required_components` 宏，该宏只有一个参数，其作用是对包的组件进行检测。若传递的参数是一个包的名字，则该宏中使用的变量是 13.3.6 小节讲解的包接口变量。
- 调用 `get_filename_component()` 命令，该命令是一个 CMake 命令，其作用是将变量 `PACKAGE_PREFIX_DIR` 的值设置为后面一个参数指定的绝对路径。其中 `CMAKE_CURRENT_LIST_DIR` 变量表示当前正在处理的列表文件所在的目录，取值范围是动态的。也就是说，若当前正在处理的文件在 `g:/qt2` 目录下，则

```
PACKAGE_PREFIX_DIR = g:/qt2/ C:/Program Files (x86)/xxxx
```

以上路径在 Windows 中不是一个正确的路径，其中子目录“C:/Program Files (x86)/xxxx”是本示例的默认安装前缀，由于本示例未设置安装前缀，也未指定 `INSTALL_PREFIX` 参数，所以使用了默认安装前缀，由于安装前缀不是本示例 `INSTALL_DESTINATION` 参数指定目录的父目录，所以才出现以上不正确的路径，这就是为什么 `INSTALL_DESTINATION` 指定的位置需要以安装前缀为根目录的原因。注：以上不正确路径的生成规则是，若 `INSTALL_DESTINATION` 参数与安装前缀不存在父子关系，则直接将安装前缀添加到 `CMAKE_CURRENT_LIST_DIR` 变量指定的目录之后

5、以下是 `configure_package_config_file()` 命令其余各参数的作用

- `NO_SET_AND_CHECK_MACRO` 参数的作用是在输出文件中不生成 `set_and_check` 宏
- `NO_CHECK_REQUIRED_COMPONENTS_MACRO` 参数的作用是在输出文件中不生成 `check_required_components` 宏
- `INSTALL_PREFIX` 和 `INSTALL_DESTINATION` 参数

- `INSTALL_PREFIX` 用于指定安装前缀，必须是绝对路径，若未指定该参数，则使用 `CMAKE_INSTALL_PREFIX` 变量。
- `INSTALL_DESTINATION` 参数可以使用相对路径和绝对路径，相对路径相对于安装前缀。该参数指定的目录必须是安装前缀的子目录。注意：该参数指定的目录并不是输出文件的实际安装位置。
- `INSTALL_DESTINATION` 参数和安装前缀共同确定需要向上返回几级目录，返回上级目录的层数是 `INSTALL_DESTINATION` 参数的目录层数与安装前缀的目录层数之差，
- `PATH_VARS` 参数用于指定一系列变量<var1>到<varN>，然后，CMake 会为这些变量生成一个名称为 `PACKAGE_<var.>` 的辅助变量。使用该参数的主要目的是使<var1>到<varN>成为包中文件的相对路径。其方法如下：

在输入文件中使用 `set_and_check` 宏并以这些辅助变量作为参数(注：该宏应在 `@PACKAGE_INIT@` 之后调用)，比如

假设 `CMakeLists.txt` 有如下语句

```
set(X aa)                #创建一个变量 x 用于生成辅助变量
configure_package_config_file(
...
PATH_VARS X              #该参数会生成一个辅助变量 PACKAGE_X
...)
```

若在输入文件中有如下语句

```
set_and_check(FF @PACKAGE_X@)
```

则在输出文件中会将以上语句替换为以下语句，注意，若未指定参数 `PATH_VARS X`，则不会进行以下替换

```
set_and_check(FF ${PACKAGE_PREFIX_DIR}/aa)
```

即，将 `X` 变量的值作为 `${PACKAGE_PREFIX_DIR}` 的子目录，可见，成功的使 `X` 变量的值成为了相对路径。

示例 13.7: `configure_package_config_file()` 命令----生成包中文件可重定位的配置文件

- ①、目的：本示例用于演示，用于生成可随包一起移动的配置文件的
- ②、准备包

在 `g:/qt1/d/e` 目录下准备源代码文件 `a.cpp` 和 `b.cpp`(读者可自行准备，只要代码能运行即可)，我们准备使用文件夹 `d` 作为包，该包有一个目录 `e`，在目录 `e` 中有两个源文件。

- ③、准备输入文件

在目录 `g:/qt2` 下的输入文件 `tt.in` 文件(文件名可任意，无强制规定)中编写如下代码。

#g:/qt2/tt.in(输入文件)

```
message(=====这是 ttConfig.cmake====)
@PACKAGE_INIT@
set_and_check(FF @PACKAGE_X@)
set(xx1 ${FF}/a.cpp)
set(xx2 ${FF}/b.cpp)
message(=====结束 ttConfig.cmake====\n)
```

④、在目录如 g:/qt2 目录下的 CMakeLists.txt 中编写如下代码

```
#g:/qt2/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx)
include(CMakePackageConfigHelpers)
set(X "d/e") #设置包中文件相对于安装前缀的位置
configure_package_config_file(
    tt.in
    g:/qt1/a/b/ttConfig.cmake
    #输出文件理论安装位置，若使用绝对路径，则该位置需是安装前缀的子目录
    INSTALL_DESTINATION g:/qt1/a/b
    PATH_VARS X #用于生成辅助变量 PACKAGE_X
    INSTALL_PREFIX g:/qt1 ) #设置安装前缀

#以下两段代码可以不出现在本代码中，本示例主要用于生成可随包移动的配置文件，
#因此，以下代码完全可以移至另一个 CMakeLists.txt 文件中
find_package(tt PATHS g:/qt1/a/b )
add_executable(nn ${xx1} ${xx2} )
```

以上代码使用完整路径指定输出文件的位置，这样可以避免使用 install()命令安装输出文件。

⑤、在 CMD 中转至 g:/qt2 并输入以下命令

```
cmake .
mingw32-make
nn.exe #测试结果，符合预期
```

⑥、下面是生成的配置文件 ttConfig.cmake 的部分关键代码

```
get_filename_component(PACKAGE_PREFIX_DIR
    "${CMAKE_CURRENT_LIST_DIR}/../../" ABSOLUTE)
....
set_and_check(FF ${PACKAGE_PREFIX_DIR}/d/e)
set(xx1 ${FF}/a.cpp)
set(xx2 ${FF}/b.cpp)
```

以上代码的 CMAKE_CURRENT_LIST_DIR 变量获取配置文件所在的目录 g:/qt1/a/b，因此

```
${FF}=${PACKAGE_PREFIX_DIR}/d/e=g:/qt1/a/b/../../d/e = g:/qt1/d/e
```

⑦、把生成的配置文件和包移动到其他位置

由于 INSTALL_DESTINATION 参数与安装前缀相差两级目录，因此，包的位置会相对于配置文件向上返回两级目录，所以，本示例的配置文件需位于两级目录中，否则，无法正确找到包的位置。但只需保证目录层数正确即可，目录名没有限制，比如位于“a/b/”，位于“x/y/”等目录中均可。

下面我们将文件夹 d(我们准备的包)移动到 f:/qt3，将 g:/qt1/a/b 中生成的配置文件 ttConfig.cmake 移动到 f:/qt3 的 x/y 目录中，然后在任一目录，比如 d:/xx 目录下的 CMakeLists.txt 文件中输入以下代码

```
cmake_minimum_required(VERSION 3.27)
project(xxxx)
```

```
find_package(tt PATHS f:/qt3/x/y )
add_executable(nn ${xx1} ${xx2} )
```

然后在 CMD 中转至 d:/xx 并输入以下命令，程序成功运行

```
cmake .
mingw32-make
nn.exe          #测试，符合预期
```

也就是说，只需把配置文件(或含配置文件的目录)和包一起移动，无论移动到何处，只需在 CMakeLists.txt 中输入以下代码便能成功获取包中文件的位置，这样就达到了可重定位的目的。

```
find_package(tt PATHS XXX)          #XXX 为配置文件所在位置。
```

13.4.2 使用 write_basic_package_version_file()命令生成版本文件

该命令完整语法如下：

```
write_basic_package_version_file(
    <filename>
    [VERSION <major.minor.patch>]
    COMPATIBILITY <AnyNewerVersion|SameMajorVersion|SameMinorVersion|ExactVersion>
    [ARCH_INDEPENDENT] )
```

- 1、以上命令用于自动生成版本文件，自动生成的版本文件包含有基本的版本判断逻辑，如果需要更复杂详细的判断规则，则需要手动编写自己的版本文件。
- 2、在 CMake 内部，这个宏执行 `configure_file()`来创建结果版本文件，根据兼容性的不同，使用相应的 `BasicConfigVersion-<COMPATIBILITY>.cmake.in` 文件。注意，这些文件是 CMake 内部的。
- 3、<filename>是输出的文件名，该名称应出现在构建树中。
- 4、VERSION 为版本号，即准备为包生成的版本号。若未指定版本号，则使用 `PROJECT_VERSION` 变量的值，若未设置，则出错。
- 5、COMPATIBILITY：指定兼容模式，其中
 - AnyNewerVersion：表示安装的版本(即此命令指定的版本号)更新，或与请求的版本(即 `find_package()`命令指定的版本号)完全相同，则认为是兼容的。
 - SameMajorVersion：表示安装的主版本号与请求的版本相同，则认为是兼容的。比如，若请求 3.2，则 4.2 将认为是不兼容的。
 - SameMinorVersion：表示安装的主版本号和次版本号与请求的版本相同，则认为是兼容的。比如，若请求 3.2，则 3.4 将认为是不兼容的。
 - ExactVersion：表示安装的版本与请求的版本完全匹配(不考虑微调版本号)，才认为是兼容的。该模式与范围版本不兼容，若为范围版本指定了该模式，将显示一个警告。
- 6、ARCH_INDEPENDENT：若指定了该参数，则表示在任何体系结构上版本都是兼容的，否则将执行体系结构检查，并且只有在体系结构完全匹配时才认为是兼容的，比如，若是 32 位体系结构，则只能在 32 位体系结构上使用才被认为是兼容的，除非指定了 `ARCH_INDEPENDENT` 参数。

示例 13.8：使用 write_basic_package_version_file ()命令生成版本文件

- ①、在目录 `g:/qt2` 下的 `CMakeLists.txt` 中编写如下代码

```
#g:/qt2/CMakeLists.txt
```



```

cmake_minimum_required(VERSION 3.27)
project(xxxx)
include(CMakePackageConfigHelpers)
write_basic_package_version_file(
    g:/qt2/ttConfigVersion.cmake    #输出文件可以指定路径
    VERSION 11.22.33                #指定的包的版本
    COMPATIBILITY                    #以下兼容模式任选一个
    #ExactVersion                    #完全匹配
    #SameMajorVersion                #匹配主版本号
    #SameMinorVersion                #匹配次版本号
    AnyNewerVersion )                #安装的版本更新或完全匹配

```

②、在 CMD 中转至 g:/qt2 并输入以下命令

cmake .

③、在 g:/qt2 中找到生成的 ttConfigVersion.cmake 文件，该文件内容如下：

#g:/qt2/ttConfigVersion.cmak(兼容模式为 AnyNewerVersion)

This is a basic version file for the Config-mode of find_package().

#这是 find_package()的配置模式的基本版本文件。

It is used by write_basic_package_version_file() as input file for configure_file()

to create a version-file which can be installed along a config.cmake file.

#write_basic_package_version_file()使用它作为 configure_file()的输入文件来创建一个版本文件，

#该版本文件可以与 config.cmake 文件一起安装。

The created file sets PACKAGE_VERSION_EXACT if the current version string and

the requested version string are exactly the same and it sets

PACKAGE_VERSION_COMPATIBLE if the current version is >= requested version.

#如果当前版本字符串和请求的版本字符串完全相同，则创建的文件设置 PACKAGE_VERSION_EXACT;

#如果当前版本是>=请求的版本，则设置 PACKAGE_VERSION_COMPATIBLE

The variable CVF_VERSION must be set before calling configure_file().

#变量 CVF_VERSION 必须在调用 configure_file()之前设置。

set(PACKAGE_VERSION "11.22.33")

if(PACKAGE_FIND_VERSION_RANGE) #第 1 个 if。若 find_package()命令请求的是范围版本，则为真

Package version must be in the requested version range. 包版本必须在请求的版本范围内

if(#第 2 个 if

(PACKAGE_FIND_VERSION_RANGE_MIN STREQUAL "INCLUDE" #包含下端点

AND

#此文件指定的版本小于 find_package()命令请求的范围版本下限，则为真

PACKAGE_VERSION VERSION_LESS PACKAGE_FIND_VERSION_MIN)

OR (

```

(PACKAGE_FIND_VERSION_RANGE_MAX STREQUAL "INCLUDE"      #包含上端点
AND
#此文件指定的版本大于 find_package()命令请求的范围版本上限，则为真
PACKAGE_VERSION VERSION_GREATER PACKAGE_FIND_VERSION_MAX)
OR (
    PACKAGE_FIND_VERSION_RANGE_MAX STREQUAL "EXCLUDE"      #排除上端点
    AND
    #此文件指定的版本大于等于 find_package()命令请求的范围版本上限，则为真
    PACKAGE_VERSION VERSION_GREATER_EQUAL PACKAGE_FIND_VERSION_MAX)
)
)
set(PACKAGE_VERSION_COMPATIBLE FALSE)                    #版本不兼容
endif()
set(PACKAGE_VERSION_COMPATIBLE TRUE)                     #版本兼容
endif()                                                    #第 2 个 if 结束
else()                                                      #匹配第 1 个 if。否则请求的不是范围版本
#此文件指定的版本小于 find_package()命令请求的版本(即，请求的版本更新)，则为真
if(PACKAGE_VERSION VERSION_LESS PACKAGE_FIND_VERSION)
    set(PACKAGE_VERSION_COMPATIBLE FALSE)
else()                                                      #否则，请求的版本更陈旧
    set(PACKAGE_VERSION_COMPATIBLE TRUE)                  #版本兼容
    #此文件指定的版本等于请求的版本(即，完全匹配)，则为真
    if(PACKAGE_FIND_VERSION STREQUAL PACKAGE_VERSION)
        set(PACKAGE_VERSION_EXACT TRUE)                  #设置版本完全匹配
    endif()
endif()
endif()
endif()

# if the installed or the using project don't have CMAKE_SIZEOF_VOID_P set, ignore it:
#如果已安装或正在使用的项目没有设置 CMAKE_SIZEOF_VOID_P，则忽略它:
if("${CMAKE_SIZEOF_VOID_P}" STREQUAL "" OR "8" STREQUAL "")
    return()
endif()

# check that the installed version has the same 32/64bit-ness as the one which is currently searching:
#检查安装的版本是否与当前正在搜索的版本具有相同的 32/64 位:
if(NOT CMAKE_SIZEOF_VOID_P STREQUAL "8")
    math(EXPR installedBits "8 * 8")
    set(PACKAGE_VERSION "${PACKAGE_VERSION} (${installedBits}bit)")
    set(PACKAGE_VERSION_UNSUITABLE TRUE)
endif()

```

④、若在本示例中将兼容性设置为 SameMinorVersion，则生成的 ttConfigVersion.cmake 文件的内容如下

#g:/qt2/ttConfigVersion.cmak(兼容模式为 SameMinorVersion)

```
# This is a basic version file for the Config-mode of find_package().
# It is used by write_basic_package_version_file() as input file for configure_file()
# to create a version-file which can be installed along a config.cmake file.
#
# The created file sets PACKAGE_VERSION_EXACT if the current version string and
# the requested version string are exactly the same and it sets
# PACKAGE_VERSION_COMPATIBLE if the current version is >= requested version,
# but only if the requested major and minor versions are the same as the current one.
#如果当前版本字符串和请求的版本字符串完全相同，则创建的文件设置 PACKAGE_VERSION_EXACT;
#如果当前版本是>=请求的版本，则设置 PACKAGE_VERSION_COMPATIBLE，但前提是请求的主要和
#次要版本与当前版本相同。

# The variable CVF_VERSION must be set before calling configure_file().
#变量 CVF_VERSION 必须在调用 configure_file()之前设置

set(PACKAGE_VERSION "11.22.33")

#此文件指定的版本小于 find_package()命令请求的版本(即，请求的版本更新)，则为真
if(PACKAGE_VERSION VERSION_LESS PACKAGE_FIND_VERSION)           #第 1 个 if
    set(PACKAGE_VERSION_COMPATIBLE FALSE)                        #版本不兼容
else()                                                            #匹配第 1 个 if
    #使用正则表达式将版本号 11.22.33 拆分为两部分 11 和 22
    if("11.22.33" MATCHES "^[0-9]+\.[0-9]+")                    #第 1.1 个 if
        #变量 CMAKE_MATCH_n 返回匹配的正规表达式的第 n 组的值
        set(CVF_VERSION_MAJOR "${CMAKE_MATCH_1}") #将主版本号赋值给变量 CVF_VERSION_MAJOR
        set(CVF_VERSION_MINOR "${CMAKE_MATCH_2}") #将次版本号赋值给变量 CVF_VERSION_MINOR

        #如果 CVF_VERSION_MAJOR(即主版本号)不为 0，则为真
        if(NOT CVF_VERSION_MAJOR VERSION_EQUAL 0)              #第 1.1.1 个 if
            #以下语句表示，将最后一个参数与正规表达式"^0_"匹配，
            #并将结果使用第 4 个参数的内容替换之后输出到变量 CVF_VERSION_MAJOR 中
            #以下语句的作用是去除主版本号中的前导 0，比如 011 将被替换为 11
            string(REGEX REPLACE "^0+" "" CVF_VERSION_MAJOR "${CVF_VERSION_MAJOR}")
        endif()                                                 #第 1.1.1 个 if 结束

        if(NOT CVF_VERSION_MINOR VERSION_EQUAL 0)              #第 1.1.2 个 if
            #去除次版本号中的前导 0
            string(REGEX REPLACE "^0+" "" CVF_VERSION_MINOR "${CVF_VERSION_MINOR}")
        endif()                                                 #第 1.1.2 个 if 结束

    else()                                                       #匹配第 1.1 个 if
        set(CVF_VERSION_MAJOR "11.22.33")
        set(CVF_VERSION_MINOR "")
```

```

endif()                                                    #第 1.1 个 if 结束

#若 find_package()命令请求的是范围版本，则为真
if(PACKAGE_FIND_VERSION_RANGE)                            #第 1.2 个 if
    # both endpoints of the range must have the expected major and minor versions
    #范围的两个端点都必须具有预期的主要和次要版本

    #将次版本号加 1，然后将结果赋给变量 CVF_VERSION_MINOR_NEXT
    math(EXPR CVF_VERSION_MINOR_NEXT "${CVF_VERSION_MINOR} + 1")

    if(                                                    #1.2.1 个 if
        NOT (
            #请求的下限主版本等于此文件指定的主版本
            PACKAGE_FIND_VERSION_MIN_MAJOR STREQUAL CVF_VERSION_MAJOR
            AND
            #请求的下限次版本等于此文件指定的次版本
            PACKAGE_FIND_VERSION_MIN_MINOR STREQUAL CVF_VERSION_MINOR
        )
        OR (
            #第 1 个 OR
            (PACKAGE_FIND_VERSION_RANGE_MAX STREQUAL "INCLUDE"    #包含上端点
            AND
            NOT (
                #请求的上限主版本等于此文件指定的主版本
                PACKAGE_FIND_VERSION_MAX_MAJOR STREQUAL CVF_VERSION_MAJOR
                AND
                #请求的上限次版本等于此文件指定的次版本
                PACKAGE_FIND_VERSION_MAX_MINOR STREQUAL CVF_VERSION_MINOR)
            )
            OR (
                #第 2 个 OR
                PACKAGE_FIND_VERSION_RANGE_MAX STREQUAL "EXCLUDE"    #排除上端点
                AND
                NOT
                #请求的上限版本小于等于此文件指定的主版本和次版本+1
                PACKAGE_FIND_VERSION_MAX VERSION_LESS_EQUAL
                ${CVF_VERSION_MAJOR}.${CVF_VERSION_MINOR_NEXT}
            )
        )
        #匹配第 1 个 OR
    )
    #匹配 1.2.1 个 if
    set(PACKAGE_VERSION_COMPATIBLE FALSE)
elseif(
    PACKAGE_FIND_VERSION_MIN_MAJOR STREQUAL CVF_VERSION_MAJOR
    AND
    PACKAGE_FIND_VERSION_MIN_MINOR STREQUAL CVF_VERSION_MINOR
    AND (

```

```

(PACKAGE_FIND_VERSION_RANGE_MAX STREQUAL "INCLUDE"
AND
PACKAGE_VERSION VERSION_LESS_EQUAL PACKAGE_FIND_VERSION_MAX)
OR (
PACKAGE_FIND_VERSION_RANGE_MAX STREQUAL "EXCLUDE"
AND
PACKAGE_VERSION VERSION_LESS PACKAGE_FIND_VERSION_MAX)
)
)
set(PACKAGE_VERSION_COMPATIBLE TRUE)
else()
set(PACKAGE_VERSION_COMPATIBLE FALSE)
endif()
else()
endif()
if(NOT PACKAGE_FIND_VERSION_MAJOR VERSION_EQUAL 0)
string(REGEX REPLACE "^0+" "" PACKAGE_FIND_VERSION_MAJOR
"${PACKAGE_FIND_VERSION_MAJOR}")
endif()

if(NOT PACKAGE_FIND_VERSION_MINOR VERSION_EQUAL 0)
string(REGEX REPLACE "^0+" "" PACKAGE_FIND_VERSION_MINOR
"${PACKAGE_FIND_VERSION_MINOR}")
endif()

if((PACKAGE_FIND_VERSION_MAJOR STREQUAL CVF_VERSION_MAJOR) AND
(PACKAGE_FIND_VERSION_MINOR STREQUAL CVF_VERSION_MINOR))
set(PACKAGE_VERSION_COMPATIBLE TRUE)
else()
set(PACKAGE_VERSION_COMPATIBLE FALSE)
endif()

if(PACKAGE_FIND_VERSION STREQUAL PACKAGE_VERSION)
set(PACKAGE_VERSION_EXACT TRUE)
endif()
endif()
endif()
endif()

# if the installed or the using project don't have CMAKE_SIZEOF_VOID_P set, ignore it:
if("${CMAKE_SIZEOF_VOID_P}" STREQUAL "" OR "8" STREQUAL "")
return()
endif()

# check that the installed version has the same 32/64bit-ness as the one which is currently searching:
if(NOT CMAKE_SIZEOF_VOID_P STREQUAL "8")

```

```
math(EXPR installedBits "8 * 8")
set(PACKAGE_VERSION "${PACKAGE_VERSION} (${installedBits}bit)")
set(PACKAGE_VERSION_UNSUITABLE TRUE)
endif()
```

第 14 章 使用 CMake 构建 Qt 程序

14.1 使用 CMake 构建一个简单的 Qt 程序

1、准备工作

- 首先，需要安装 Qt 软件，本文主要讲解 windows10 系统下的 Qt 程序并使用 VC++编译器。
- 使用的 qt 版本为 qt6.6.0，Qt Creator 的版本号为 Qt Creator 11.0.3。Qt6 的源代码文件必须使用 UTF-8 编码。
- VC++编译器版本为 Visual Studio 2022 (64 位)，即 Visual Studio 17.5.4
- CMake 版本为 3.27
- mingw32-make 版本为 GNU Make 4.2.1

2、CMake 所在位置

如果在安装 Qt 时选择了安装 Qt 自带的 CMake 程序，则会在以下目录(仅供参考，实际目录以安装路径而定)找到与 CMake 有关的 cmake.exe、cmake-gui.exe 等程序

C:\Qt\Tools\CMake_64\bin

同理，与 CMake 有关的其他文件位于以下目录(仅供参考)

C:\Qt\Tools\CMake_64

注：Qt 自带的 CMake 版本可能会比较陈旧，若要使用更新版本的 CMake，则只需把新下载的 CMake 相关文件复制到 C:\Qt\Tools\CMake_64 目录即可。

3、Qt 提供的 CMake 包(即.cmake 文件)所在位置

1)、g++

若在安装 Qt 时选择了安装 g++(即 MinGw)，则可在以下目录找到 Qt 自带的 CMake 包

C:\Qt\6.6.0\mingw_64\lib\cmake

在该文件夹中包含有数量比较多的文件夹，每个文件夹之下都有相应的 CMake 包(通常是 CMake 的配置文件)。

通常我们只需查找 CMake 配置文件 Qt6Config.cmake 即可(读者可用记事本打开，以查看其中的内容)，该文件会根据指定的组件(Qt 称其为模块，如 Core，Widgets 等)自动找到相应的其他 CMake 包。Qt6Config.cmake 文件位于以下目录

C:\Qt\6.6.0\mingw_64\lib\cmake\Qt6

2)、MSVC

若在安装 Qt 时选择了安装 MSVC，则可在以下目录找到 Qt 自带的 CMake 包，这些包的名称与 C:\Qt\6.5.1\mingw_64\lib\cmake 目录下的名称是一模一样的。

C:\Qt\6.6.0\msvc2019_64\lib\cmake

同理，在以下目录也存在一个名称为 Qt6Config.cmake 的 CMake 配置文件

C:\Qt\6.6.0\mingw_64\lib\cmake\Qt6

- 4、要使用 CMake 构建一个 Qt 程序，我们需要使用 Qt 提供的 CMake 包，即需要 Qt 提供的源文件、库、头文件等文件的路径信息及其他信息，Qt 提供的文件是如此的多的，我们不可能完全知道所有文件的确切位置，幸运的是，Qt 提供了相应的 CMake 包(通常是一个 CMake 配置文件，名称为“*Config.cmake”)，在 CMake 包中包含有 Qt 提供的相应文件的路径等信息，因此，要使用 CMake 构建一个 Qt 程序需加载 Qt 提供的 CMake 包，通常只需加载 Qt 提供的 CMake 配置文件 Qt6Config.cmake(即 CMake 包名为 Qt6 的包)。
- 5、Qt 的包被分为了多个组件(Qt 称为模块)，比如 Core、Gui、Widgets 等组件，所以，在加载 Qt 包时还应指定需要加载的组件，每个被加载的 Qt 组件都定义了一个 CMake 库目标，其称名以 Qt6::开头，后跟模块名，比如 Qt6::Core、Qt6::Widgets 等，在构建 Qt 程序时，需将相应的库目标名称传递给 target_link_libraries()命令以使用相应的库。
- 6、从 Qt 加载的导入目标是用与配置 Qt 时相同的配置创建的，
 - 若 Qt 配置了 -debug，则会创建一个带有 DEBUG 配置的导入目标。
 - 若 Qt 配置了 -release，则会创建一个带有 RELEASE 配置的导入目标。
 - 若 Qt 配置了 -release 和 -debug，则会创建一个带有 RELEASE 和 DEBUG 配置的导入目标。

3、使用 CMake 构建一个简单的 Qt 程序

下面以示例的形式来讲解，怎样使用 CMake 构建 Qt 程序

示例 14.1：构建一个简单的 Qt 程序

①、源文件准备

在目录 g:/qt3 下的 b.cpp 中编写如下的 Qt 代码，以下代码用于创建一个简单的对话框

//g:/qt3/b.cpp (Qt 源代码)

```
#include<QApplication>          /*Qt 中的每一个类都有一个与其同名的头文件，
                                因此要使用哪个类，就需要包含相应的头文件*/

#include<QDialog>
#include<QLabel>
int main(int argc, char *argv[]){
    QApplication a(argc,argv);    /*任何一个 widgets 程序都需要包含一个 QApplication
                                对象，该对象用于管理程序的资源、事件等*/

    QDialog d;                    //创建一个对话框，Qt 使用 QDialog 类对象创建对话框
    d.resize(200,100);            //设置对话框的大小。
    QLabel s(&d);                 /*创建一个标签，并把该标签放在对话框 d 之中。
                                同理，Qt 使用 QLabel 类对象创建标签*/
    s.setText("AAA");             //设置标签显示的文本。
    s.move(50,50);                //设置标签位于对话框中的位置。
    d.show();                     //显示对话框，默认情况下部件是不可见的，因此需显示。
    a.exec();                     /*程序进入消息(或事件)循环，等待用户的输入并进行响应，
                                这时程序将控制权交给 Qt 用于完成事件处理。*/

    return 0;                    }
}
```

②、在目录 g:/qt3 下的 CMakeLists.txt 中编写如下代码

g:/qt3/CMakeLists.txt


```

cmake_minimum_required(VERSION 3.27)
project(xxxx LANGUAGES CXX)

#指定 C++标准为 C++17。Qt 需要 C++17 标准。
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

#设置搜索 CMake 包的路径以免找到错误的包，若出现找到不包的情形，可以使用以下路径
# C:/Qt/6.6.0/msvc2019_64/lib/cmake
set(CMAKE_PREFIX_PATH C:/Qt/6.6.0/msvc2019_64/lib/cmake/Qt6 )

find_package(Qt6                                #查找名称为 Qt6 的包
  COMPONENTS Widgets                          #指定需要的组件(在 Qt 中被称为模块)
  #PATHS C:/Qt/6.6.0/msvc2019_64/lib/cmake/Qt6 #不建议在此处指定路径
)

message("${Qt6_DIR}")                          #查看是否找到了正确的包
add_executable(nn b.cpp )                      #生成可执行文件
target_link_libraries(nn Qt6::Widgets)         #将 Qt6 包中的 Widgets 目标链接到 nn

```

③、CMake 代码说明

- A、本示例只需使用到 Qt 的 Widgets 模块(CMake 称其为组件)，所以，只需在 find_package()命令中指定该模块即可。
- B、以上 CMake 代码设置了 find_package()命令的搜索路径，这是因为，在安装 Qt 时，如果同时安装了 MinGw 和 MSVC，会出现两个相同名称的 Qt6Config.cmake 文件，本示例使用的是 MSVC，所以，为避免默认情况下错误的搜索到为 MinGw 编写的 Qt6Config.cmake 文件，而明确的指定了 find_package()命令的搜索路径，路径应在 CMAKE_PREFIX_PATH 变量中指定，而不是在 PATHS 参数指定，因为按照 CMake 的搜索顺序，PATHS 参数指定的路径具有最低的搜索优先级，若在 PATHS 中指定路径，不一定会被搜索到。
- C、注意：由于 Qt6::Widgets 依赖于 Qt6::Core，所以，若 target_link_libraries()链接到 Qt6::Widgets，则还会链接到 Qt6::Core

④、在 CMD 中转至 g:/qt3 并输入以下命令

```

cmake .
mingw32-make
nn.exe      #测试，成功弹出一个对话框，结果符合预期

```

14.2 CMake 自动调用 uic.exe 工具生成头文件的过程及原理

14.21 Qt 构建工具

- 1、有些 Qt 代码需要使用 Qt 专门的工具才能构建，如使用界面编辑器生成的 ui 文件需要使用 uic.exe 工具来构建，带有元对象系统的代码需要借助 moc.exe 工具，资源系统需要使用 rcc.exe 工具，这些工具位于以下目录

C:\Qt\6.5.1\msvc2019_64\bin 或
C:\Qt\6.5.1\mingw_64\bin

2、使用表 14.1 中的属性或变量可以控制 CMake 是否允许在需要时自动调用 Qt 构建工具，

表 14.1 CMake 控制 Qt 构建工具变量和属性

类别	名称	说明
目标属性	AUTOUIIC	是否为 Qt 目标自动调用 uic.exe 工具。若此属性为 ON，则 CMake 将在构建时扫描头文件和源文件并在需要时自动调用 uic.exe 工具，若在创建目标时设置了 CMAKE_AUTOUIIC 变量，则由该变量的值初始化。目前支持 Qt4~Qt6
	AUTOMOC	是否为 Qt 目标自动调用 moc.exe 工具。若此属性为 ON，则 CMake 将在构建时扫描头文件和源文件并在需要时自动调用 moc.exe 工具。若在创建目标时设置了 CMAKE_AUTOMOC 变量，则由该变量的值初始化。目前支持 Qt4~Qt6
	AUTORCC	是否为 Qt 目标自动调用 rcc.exe 工具。若此属性为 ON 时，则 CMake 将处理在构建时作为目标源添加的.qrc 文件，并自动调用 rcc.exe 工具。若在创建目标时设置了 CMAKE_AUTORCC 变量，则由该变量的值初始化。目前支持 Qt4~Qt6
变量	CMAKE_AUTOUIIC	该变量用于初始化所有目标上的 AUTOUIIC 目标属性的值。
	CMAKE_AUTOMOC	该变量用于初始化所有目标上的 AUTOMOC 目标属性的值
	CMAKE_AUTORCC	该变量用于初始化所有目标上的 AUTORCC 目标属性的值。
源文件属性	SKIP_AUTOUIIC	将源文件排除在 AUTOUIIC 处理之外，即，该源文件不由 CMake 自动调用 uic.exe 命令处理。
	SKIP_AUTOMOC	将源文件排除在 AUTOMOC 处理之外，即，该源文件不由 CMake 自动调用 moc.exe 命令处理。
	SKIP_AUTORCC	将源文件排除在 AUTORCC 处理之外，即，该源文件不由 CMake 自动调用 rcc.exe 命令处理。
	SKIP_AUTOGEN	从 AUTOMOC，AUTOUIIC 和 AUTORCC 处理中排除源文件(对于 Qt 项目)。

14.2.2 CMake 自动调用 uic.exe 工具生成头文件的过程及原理

1、AUTOUIIC 目标属性控制 CMake 是否在适当的时候自动调用 uic.exe 工具，以下为 CMake 调用 uic.exe 的过程：

1)、在构建时，CMake 从目标的源代码中扫描每个头文件和源文件，以查找如下形式的语句

```
#include "ui_<name>.h"
```

注意：头文件必须以 ui_ 开头，并且在#include 和"ui_<name>.h"之间一定要有空格，否则 CMake 不会自动调用 uic.exe 工具。

2)、一旦找到以上形式的 include 语句，CMake 就会搜索名为<name>.ui 的 ui 文件，并将该文件作为 uic.exe 的输入文件。注意：<name>.ui 文件的基本名称必须与#include 中的 ui_<name>.h 基本名称一致，即，二者的<name>部分必须有相同的名称。搜索 ui 文件的路径及顺序如下：

- <source_dir>/<name>.ui

- `<source_dir>/<path><name>.ui`
- `<AUTOUIIC_SEARCH_PATHS>/<name>.ui`
- `<AUTOUIIC_SEARCH_PATHS>/<path><name>.ui`

其中, `<source_dir>`是 C++源文件的目录, `<AUTOUIIC_SEARCH_PATHS>`是目标属性 `AUTOUIIC_SEARCH_PATHS` 指定的路径。

3)、若找到了`<name>.ui` 文件, 则调用 `uic.exe` 工具在以下目录生成名为 `ui_<name>.h` 的头文件(使用 `uic.exe` 工具的主要目的就是由 `.ui` 文件生成相应的头文件)

- 单配置器生成器: `<AUTOGEN_BUILD_DIR>/include`
- 多配置器生成器: `<AUTOGEN_BUILD_DIR>/include_<CONFIG>`

以上目录会被自动添加到目标的 `INCLUDE_DIRECTORIES` 目标属性。其中 `<AUTOGEN_BUILD_DIR>`是目标属性 `AUTOGEN_BUILD_DIR` 指定的值, 默认值为

`<dir>/<target-name>_autogen`

其中`<dir>`为 `CMAKE_CURRENT_BINARY_DIR`, `<target-name>`为目标的逻辑名称

2、为 `uic.exe` 命令指定额外的参数

- 可以使用 `AUTOUIIC_OPTIONS` 目标属性来向 `uic.exe` 指定参数, 目标属性只能为单个目标指定参数。若在创建目标时设置 `CMAKE_AUTOUIIC_OPTIONS` 变量, 则由该变量的值初始化, 否则为空字符串。
- 使用 `CMAKE_AUTOUIIC_OPTIONS` 变量可以为其后的所有目标的 `uic.exe` 指定参数。
- 还可使用 `AUTOUIIC_OPTIONS` 源文件属性以在文件范围内为 `uic.exe` 指定参数, 源文件属性会覆盖相同名称的目标属性的值。
- 另外, 还可设置 `INTERFACE_AUTOUIIC_OPTIONS` 目标属性, 该目标属性是使用要求, 因此, 可按使用要求的规则进行传递。`CMAKE_DEBUG_TARGET_PROPERTIES` 变量用来跟踪 `INTERFACE_AUTOUIIC_OPTIONS` 目标属性的原始目标。

3、`AUTOGEN_TARGET_DEPENDS` 目标属性

由于具有 `AUTOUIIC` 或 `AUTOMOC` 属性的目标 CMake 会为其创建一个名为 `*_autogen` 的目标, 该目标用于生成 `moc` 和 `uic` 文件, 由于这个目标是在生成时创建的, 所以不能使用 `add_dependencies()`命令来添加其依赖关系。相反, 应使用 `AUTOGEN_TARGET_DEPENDS` 目标属性设置 `*_autogen` 目标的依赖项列表, 其值可以是目标名或文件名。

4、最后, 可以使用 `SKIP_AUTOUIIC` 或 `SKIP_AUTOGEN` 源文件属性将源文件排除在 `AUTOUIIC` 处理之外, 即, 该源文件不由 CMake 自动调用 `uic.exe` 命令处理。

5、表 14.2 是 CMake 处理 `ui` 文件时需要使用到的变量和属性

表 14.2 CMake 处理 `ui` 文件的变量和属性

类别	名称	说明
目标属性	<code>AUTOUIIC_SEARCH_PATHS</code>	用于指定查找 <code>.ui</code> 文件的搜索路径, 若在创建目标时设置了 <code>CMAKE_AUTOUIIC_SEARCH_PATHS</code> 变量, 则由该变量的值初始化, 否则为空。

	AUTOGEN_BUILD_DIR	指定 AUTOMOC, AUTOUIC 和 AUTORCC 为目标生成的文件的目录(需指定绝对路径)。该目录是根据需要创建的, 并自动添加到 ADDITIONAL_CLEAN_FILES 目标属性中。当未设置或空值时, 使用目录<dir>/<target-name>_autogen, 其中<dir>为 CMAKE_CURRENT_BINARY_DIR, <target-name>为目标的逻辑名称。默认情况下, 该属性的值是未设置的。
	AUTOUIC_OPTIONS	为 uic.exe 提供的额外的参数, 若在创建目标时设置 CMAKE_AUTOUIC_OPTIONS 变量, 则由该变量的值初始化, 否则为空字符串。
	AUTOGEN_TARGET_DEPENDS	为具有 AUTOUIC 或 AUTOMOC 属性的目标创建的*_autogen 目标添加的依赖项列表, 其值可以是目标名或文件名
	INTERFACE_AUTOUIC_OPTIONS	为 uic.exe 提供的额外的参数, 这是使用要求。
源文件属性	AUTOUIC_OPTIONS	为 uic.exe 提供的额外的参数, 默认为空, 在源文件上的 AUTOUIC_OPTIONS 属性会覆盖相同名称的目标属性。
	SKIP_AUTOUIC	将源文件排除在 AUTOUIC 处理之外(对于 Qt 项目)。
	SKIP_AUTOGEN	从 AUTOMOC, AUTOUIC 和 AUTORCC 处理中排除源文件(对于 Qt 项目)。
变量	CMAKE_AUTOUIC_SEARCH_PATHS	用于指定查找.ui 文件的搜索路径, 该变量用于初始化所有目标上的 AUTOUIC_SEARCH_PATHS 目标属性, 默认为空值
	CMAKE_AUTOUIC_OPTIONS	为 uic.exe 提供的额外的参数, 该变量用于初始化所有目标上的 AUTOUIC_OPTIONS 目标属性

示例 14.2: 在 CMake 中自动调用 Qt 的 uic.exe 构建工具

①、准备 ui 文件

在目录 g:/qt3 下准备一个名称为 dg.ui 的文件, 可以使用 Qt 的界面编辑器编辑一个含有对话框的 ui 文件, 也可将以下内容保存为 dg.ui 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
  <class>dg2</class>
  <widget class="QDialog" name="dg2">
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>400</width>
        <height>300</height>
      </rect>
    </property>
    <property name="windowTitle">
      <string>Dialog</string>
    </property>
    <widget class="QLabel" name="label">
```

```

<property name="geometry">
  <rect>
    <x>130</x>
    <y>130</y>
    <width>53</width>
    <height>16</height>
  </rect>
</property>
<property name="text">
  <string>AAA</string>
</property>
</widget>
</widget>
<resources/>
<connections/>
</ui>

```

②、准备源文件

在目录 g:/qt3 下的 a.cpp 中编写如下的 Qt 代码，以下代码用于创建一个简单的对话框

//g:/qt3/a.cpp (Qt 源文件)

```

#include "ui_dg.h"          //注意：必须在#include 与"ui_dg.h"之间留一空格，
                           //否则 CMake 不会自动调用 uic.exe 工具。

int main(int argc, char *argv[]){
    QApplication a(argc,argv);
    QDialog d;
    Ui::dg2 ui;
    ui.setupUi(&d);
    d.show();
    a.exec();
    return 0;    }

```

③、在目录 g:/qt3 下的 CMakeLists.txt 中编写如下代码

#g:/qt3/CMakeLists.txt

```

cmake_minimum_required(VERSION 3.27)
project(xxxx LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_AUTOUIC ON)          #让 CMake 自动调用 uic.exe 工具

#设置搜索 CMake 包的路径以免找到错误的包，若出现找到不包的情形，可以使用以下路径
# C:/Qt/6.6.0/msvc2019_64/lib/cmake
set(CMAKE_PREFIX_PATH C:/Qt/6.6.0/msvc2019_64/lib/cmake/Qt6)
find_package(Qt6 COMPONENTS Widgets )
add_executable(nn a.cpp)

```

```
target_link_libraries(nn Qt6::Widgets )
```

④、在 CMD 中转至 g:/qt3 并输入以下命令

```
cmake .
mingw32-make
nn.exe      #测试，成功弹出一个对话框，结果符合预期
```

⑤、查看生成的文件及目录结构

A、执行 mingw32-make 命令后，会创建一个以下的目录结构

```
g:/qt3/nn_autogen/include
```

在其中生成了需要的 ui_dg.h 头文件。以上路径可通过 AUTOGEN_BUILD_DIR 目标属性更改。

B、可在 G:\qt3\CMakeFiles 目录下找到一个名为

```
nn_autogen.dir
```

的文件夹，通常，名称为*.dir 的文件夹是 CMake 为其逻辑目标而创建的文件夹，在该文件夹中包含有构建该逻辑目标的 makefile 代码，因此，这意味着 CMake 在后台创建了一个名为 nn_autogen 的 CMake 逻辑目标。

C、查看 G:\qt3\CMakeFiles\nn.dir\ flags.make 文件，其中增加了以下重要的编译参数

```
-external:IG:\qt3\nn_autogen\include
```

D、通过追踪 makefile 代码，可知，makefile 会执行 G:\qt3\CMakeFiles\nn_autogen.dir\build.make 文件中的下面语句

```
CMakeFiles/nn_autogen:
.....
C:\Qt\Tools\CMake_64\bin\cmake.exe -E cmake_autogen
G:/qt3/CMakeFiles/nn_autogen.dir/AutogenInfo.json Debug
```

以上语句表明，CMake 调用 uic.exe 工具是通过一个名为 AutogenInfo.json 的 json 文件执行的。

E、JSON(JavaScript Object Notation，即 JavaScript 对象标记法)文件简介

JSON 是一种存储数据的文本文件，JSON 完全独立于编程语言，便于在多种语言之间进行数据交换，也便于网络传输、机器解析和生成。JSON 的语法特点如下(仅是简略描述)：

- 数据以“键-值”对的形式和数组的形式存储，
- 使用“{}”表示对象，
- 使用“[]”表示数组，
- 每个名称后跟一个冒号“:”，
- 逗号用于分隔，
- 字符串必须使用双引号括起来

F、通过以上讲解，我们能粗略的看懂 AutogenInfo.json 文件的内容，比如，其中的语句

```
"UIC_OPTIONS": []
```

表示给 uic.exe 指定的参数存储在 UIC_OPTIONS 名称之后的数组中。

再如:

"UIC_SEARCH_PATHS" : []

表示指定的搜索 ui 文件的路径存储在 UIC_SEARCH_PATHS 名称之后的数组中。

G、CMake 对 JSON 文件的内容以及解释方法有 CMake 专门的规则，比如，为 ui 文件生成的 JSON 文件必须有 CMAKE_SOURCE_DIR、CMAKE_BINARY_DIR 键，其值必须是字符串等等。

H、可以使用以下 CMake 命令来执行 JSON 文件的内容

```
cmake.exe -E cmake_autogen xxx.json debug
```

注意：末尾需要有一个配置选项 debug 或 release

14.3 CMake 自动调用 moc.exe 工具处理 Qt 的元对象系统

14.3.1 moc 简介

- 1、moc 全称是 Meta-Object Compiler(元对象编译器)，它是一个读取并分析 C++源文件的工具，若发现一个或多个包含了 Qt 专有宏(如 Q_OBJECT 等)声明的类，则会生成另外一个包含了 Q_OBJECT 宏实现代码的 C++源文件(该源文件通常名称为 moc_*.cpp)，这个新的源文件要么被#include 包含到类的源文件中，要么被编译链接到类的实现中(通常是使用此方法)。注意：新文件不会替换掉旧文件，而是与原文件一起编译。
- 2、Qt 的元对象系统提供了对象间通信的机制、动态属性等 Qt 专有的特性，这些特性是 Qt 对原有的 C++标准的扩展，因此，使用标准的 C++编译器不能编译含有元对象系统的 Qt 程序，对此在编译 Qt 程序之前，需要把扩展的语法去掉，该功能就是 moc 要做的事。
- 3、Qt 的 moc.exe 工具位于以下目录
C:\Qt\6.6.0\msvc2019_64\bin 或
C:\Qt\6.6.0\mingw_64\bin
- 4、有关 moc 的更详细的信息请参阅《Qt5.10 GUI 完全参考手册》第 2.1 小节的内容。
- 5、AUTOMOC 目标属性控制 CMake 是否在适当的时候自动调用 moc.exe 工具
- 6、由于是否需要调用 moc.exe 工具是由代码中是否包含有 Qt 专有的宏(如 Q_OBJECT)而定的，而 Qt 专有的宏可位于头文件，也可位于源文件中，所以，CMake 分两种情形来调用 moc.exe 工具。

14.3.2 Qt 专有宏位于头文件中

- 1、Qt 专有宏位于头文件时，CMake 的处理过程为：搜索具有与源文件相同基本名称但后缀不同的头文件，然后在这些头文件中搜索 Qt 专有宏，一旦找到 Qt 专有宏，就调用 moc 将其编译为一个名称为 moc_*.cpp 的源文件。
- 2、具体处理过程如下：

注：以下的基本名称是指“<source_name>”所代表的名称，以下过程中的源文件、头文件、.moc 文件，其基本名称部分必须相同

1)、搜索头文件

从目标的源代码中计算出一个应该被扫描的头文件列表，目标源文件中的所有头文件都被添加到这个扫描列表中，对目标源文件中的所有 C++ 源文件 `<source_name>.<source_extension>`，CMake 搜索具有相同基本名称 (`<source_name>.<header_extension>`) 的常规头文件和具有相同基本名称加 `_p` 后缀的私有头文件 (`<source_name>_p.<header_extension>`)，并将这些头文件添加到扫描列表中。

2)、搜索 Qt 专有宏

在构建时，CMake 从列表中扫描每个未知或修改的头文件，并搜索：

- 一个来自 `AUTOMOC_MACRO_NAMES` (详见后文) 目标属性指定的 Qt 宏。
`AUTOMOC_MACRO_NAMES` 目标属性的值含有 Qt 的专有宏名称。
- 来自 `Q_PLUGIN_METADATA` 宏的 `FILE` 参数的额外文件依赖项。
- 由 `AUTOMOC_DEPEND_FILTERS` 目标属性中定义的过滤器检测的额外文件依赖项。

3)、调用 moc.exe

若找到了 Qt 宏，则由 `moc.exe` 将头文件编译为一个名称为 `moc_<source_name>.cpp` 的文件，该文件将被 CMake 生成的另一个文件 `mocs_compilation.cpp` 或 `mocs_compilation_${CONFIG}.cpp` 自动包含，即含有类似如下的语句

```
#include "moc_<source_name>.cpp"
```

`mocs_compilation.cpp` 或 `mocs_compilation_${CONFIG}.cpp` 文件会与目标的源文件一起被链接。

4)、总结

A、以上步骤最终使用 `moc.exe` 生成了以下两个文件

- `moc_<source_name>.cpp`
- `mocs_compilation.cpp` 或 `mocs_compilation_${CONFIG}.cpp`

B、`moc_<source_name>.cpp` 文件的输出路径为

- 单配置器生成器： `<AUTOGEN_BUILD_DIR>/<SOURCE_DIR_CHECKSUM>`
- 多配置器生成器：
`<AUTOGEN_BUILD_DIR>/include_<CONFIG>/<SOURCE_DIR_CHECKSUM>`

C、`mocs_compilation.cpp` 或 `mocs_compilation_${CONFIG}.cpp` 文件的输出路径为

- `<AUTOGEN_BUILD_DIR>/mocs_compilation.cpp`，或
- `<AUTOGEN_BUILD_DIR>/mocs_compilation_${CONFIG}.cpp`

其中 `<SOURCE_DIR_CHECKSUM>` 是从 `moc` 输入文件的相对父目录路径计算的校验和。这个方案允许在不同的目录中有多个同名的输入文件。`<AUTOGEN_BUILD_DIR>` 是目标属性 `AUTOGEN_BUILD_DIR` 指定的值，默认值为

```
<dir>/<target-name>_autogen
```

其中 `<dir>` 为变量 `CMAKE_CURRENT_BINARY_DIR` 的值，`<target-name>` 为目标的逻辑名称

示例 14.3：CMake 自动调用 moc.exe 工具---Qt 专有宏位于头文件中

①、准备 C++ 文件

在目录 `g:/qt3` 下的 `c.h` 中编写如下的 Qt 代码。注意：除后缀外，源文件和头文件的名称必须相同，这是 CMake 的要求。


```
//g:/qt3/c.h(基本名称必须与源文件相同)
```

```
#include<QObject>
class A:public QObject {
Q_OBJECT          //这是 Qt 专有宏，需启用 moc.exe 进行处理
public:           A(){QDebug("DDD\n");}      };
```

在目录 g:/qt3 下的 c.cpp 中编写如下的 Qt 代码

```
//g:/qt3/c.cpp(基本名称必须与头文件相同)
```

```
#include "c.h"
void main(){ A ma; }
```

②、在目录 g:/qt3 下的 CMakeLists.txt 中编写如下代码

```
#g:/qt3/CMakeLists.txt
```

```
cmake_minimum_required(VERSION 3.27)
project(xxxx LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)                                #Qt6 要求支持 C++17 标准
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_AUTOMOC ON)                                     #自动调用 moc.exe
#设置搜索 CMake 包的路径以免找到错误的包，若出现找不到包的情形，可以使用以下路径
# C:/Qt/6.6.0/msvc2019_64/lib/cmake
set(CMAKE_PREFIX_PATH C:/Qt/6.6.0/msvc2019_64/lib/cmake/Qt6)
find_package(Qt6 COMPONENTS Widgets)
add_executable(nn c.cpp)
target_link_libraries(nn Qt6::Widgets )
```

③、在 CMD 中转至 g:/qt3 并输入以下命令

```
cmake .
mingw32-make
nn.exe          //输出 DDD，符合预期
```

④、查看生成的文件及目录结构

执行 mingw32-make 命令后，会创建一个以下的目录结构(可通过 AUTOGEN_BUILD_DIR 目标属性更改)

```
g:/qt3/nn_autogen
├── mocs_compilation.cpp
├── include
└── EWIEGA46WW
    ├── moc_c.cpp
    └── moc_c.cpp.d
```

文件夹的名称 EWIEGA46WW 的是根据 moc 输入文件的相对父目录路径计算的校验和。在 nn_autogen 文件夹中生成了 mocs_compilation.cpp 源文件，在 nn_autogen/EWIEGA46WW 文件夹中生成了 moc_c.cpp 和 moc_c.cpp.d 文件，其中 moc_c.cpp 是由处理 Qt 专有宏的 moc.exe 程序生成的，CMake 并没有直接编译该文件，而是将该文件包含在了 mocs_compilation.cpp 源文件中，打开 mocs_compilation.cpp 源文件可看到如下代码：

```
#include "EWIEGA46WW/moc_c.cpp"
```

CMake 会编译 `mocs_compilation.cpp` 文件并与目标 `nn` 的源文件 `c.cpp` 一起链接。

- ⑤、可在 `G:\qt3\CMakeFiles` 目录下找到一个名为 `nn_autogen.dir` 的文件夹，这说明 CMake 在后台创建了一个名为 `nn_autogen` 的 CMake 逻辑目标
- ⑥、通过追踪 `makefile` 代码，可知，`makefile` 会执行 `G:\qt3\CMakeFiles\nn_autogen.dir\build.make` 文件中的下面语句

```
CMakeFiles/nn_autogen:
.....
C:\Qt\Tools\CMake_64\bin\cmake.exe -E cmake_autogen
G:/qt3/CMakeFiles/nn_autogen.dir/AutogenInfo.json Debug
```

以上语句表明，CMake 调用 `cmake.exe` 工具是通过一个名为 `AutogenInfo.json` 的 json 文件执行的，有关 JSON 文件的讲解，参阅前文对 `uic.exe` 的讲解

- ⑦、通过进一步追踪 `makefile` 代码，可知，CMake 会编译 `nn_autogen/mocs_compilation.cpp` 文件，并将编译后的 `.obj` 文件与 `c.cpp.obj` 一起进行了链接，以下是追踪过程：

在 `G:\qt3\CMakeFiles\nn.dir\build.make` 文件中，可找到以下代码

```
nn.exe: CMakeFiles/nn.dir/nn_autogen/mocs_compilation.cpp.obj
nn.exe: CMakeFiles/nn.dir/c.cpp.obj
```

以上语句表明，要构建 `nn.exe` 需要以上两个依赖，在 `build.make` 文件中继续查找，可找到以下代码（经简化整理）

```
CMakeFiles/nn.dir/nn_autogen/mocs_compilation.cpp.obj: nn_autogen/mocs_compilation.cpp
cl.exe /nologo ..... -c G:\qt3\nn_autogen\mocs_compilation.cpp
```

以上语句表示，CMake 编译了 `G:\qt3\nn_autogen\mocs_compilation.cpp` 文件。进一步追踪 `makefile` 代码可知，最后会将 `mocs_compilation.obj` 和 `c.cpp.obj` 文件一起链接。

- ⑧、查看 `G:\qt3\CMakeFiles\nn.dir\flags.make` 文件，其中增加了以下重要的编译参数

```
-external:IG:\qt3\nn_autogen\include
```

14.3.3 Qt 专有宏位于源文件中

- 1、在构建时，CMake 从目标的源代码中扫描每个未知或修改的 C++ 源文件，并搜索：

- 一个来自 `AUTOMOC_MACRO_NAMES` (详见后文) 目标属性指定的 Qt 宏。
`AUTOMOC_MACRO_NAMES` 目标属性的值含有 Qt 的专有宏名称。
- 包含的 `moc` 头文件 (这里并不是指 `*.h` 头文件，而是指的 `*.moc` 文件)
- 来自 `Q_PLUGIN_METADATA` 宏的 `FILE` 参数的额外文件依赖项。
- 由 `AUTOMOC_DEPEND_FILTERS` 目标属性中定义的过滤器检测的额外文件依赖项。

- 2、若找到了 Qt 专有宏，并且在 C++ 源文件中包含有一个将由 CMake 生成的以 `.moc` 为后缀的文件，即源文件中包含一个以下的 `include` 语句

```
#include <<source_name>.moc> //或者
#include "<source_name>.moc"
```

则 CMake 将调用 moc.exe 生成一个名为<source_name>.moc 的文件，该文件的输出路径为

- 单配置器生成器: <AUTOGEN_BUILD_DIR>/include
- 多配置器生成器: <AUTOGEN_BUILD_DIR>/include_<CONFIG>

以上目录会被自动添加到目标的 INCLUDE_DIRECTORIES 目标属性。其中
<AUTOGEN_BUILD_DIR>是目标属性 AUTOGEN_BUILD_DIR 指定的值，默认值为

<dir>/<target-name>_autogen

其中<dir>为 CMAKE_CURRENT_BINARY_DIR， <target-name>为目标的逻辑名称

示例 14.4: CMake 自动调用 moc.exe 工具---Qt 专有宏位于源文件中

①、准备 C++ 文件

在目录 g:/qt3 下的 d.cpp 中编写如下的 Qt 代码

```
//g:/qt3/d.cpp
#include<QObject>
class A:public QObject {
    Q_OBJECT          //这是 Qt 专有宏，需启用 moc.exe 进行处理
public: A(){qDebug("DDD\n");} };
//d.moc 便是 CMake 自动调用的 moc.exe 生成的文件，可使用记事本打开。
//以下语句必须位于类 A 的定义之后，因为在 d.moc 中用到了类 A，此时必须见到类 A 的完整定义。
#include "d.moc"
void main(){A ma;}
```

②、在目录 g:/qt3 下的 CMakeLists.txt 中编写如下代码

```
#g:/qt3/CMakeLists.txt
cmake_minimum_required(VERSION 3.27)
project(xxxx LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)          #Qt6 要求支持 C++17 标准
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_AUTOMOC ON)               #自动调用 moc.exe
#设置搜索 CMake 包的路径以免找到错误的包，若出现找到不包的情形，可以使用以下路径
# C:/Qt/6.6.0/msvc2019_64/lib/cmake
set(CMAKE_PREFIX_PATH C:/Qt/6.6.0/msvc2019_64/lib/cmake/Qt6)

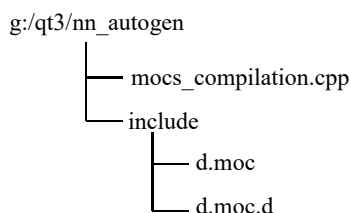
find_package(Qt6 COMPONENTS Widgets)
add_executable(nn d.cpp)
target_link_libraries(nn Qt6::Widgets )
```

③、在 CMD 中转至 g:/qt3 并输入以下命令

```
cmake .
mingw32-make
nn.exe          //输出 DDD，符合预期
```

④、查看生成的文件及目录结构

执行 mingw32-make 命令后，会创建一个以下的目录结构(可通过 AUTOGEN_BUILD_DIR 目标属性更改)



与上一示例不同，本示例没有 EWIEGA46WW 文件夹。在 nn_autogen 文件夹中生成了 mocs_compilation.cpp 源文件，在 nn_autogen/ include 文件夹中生成了 d.moc 和 d.moc.d 文件，其中 d.moc 是由处理 Qt 专有宏的 moc.exe 程序生成的，该文件直接被包含在了源文件中。而 mocs_compilation.cpp 源文件中只定义了一个枚举，打开 mocs_compilation.cpp 源文件可看到如下代码：

```
enum some_compilers { need_more_than_nothing };
```

注意：mocs_compilation.cpp 仍然会被编译并链接。

⑤、本示例生成的其他目录及文件与上一示例类似，从略。

14.3.4 与 moc 有关的其他变量和属性

1、为 moc.exe 命令指定额外的参数

- 可以使用 AUTOMOC_MOC_OPTIONS 目标属性来向 moc.exe 指定参数，目标属性只能为单个目标指定参数。若在创建目标时设置 CMAKE_AUTOMOC_MOC_OPTIONS 变量，则由该变量的值初始化，否则为空字符串。
- 使用 CMAKE_AUTOMOC_MOC_OPTIONS 变量可以为其后的所有目标的 moc.exe 指定参数。

注意：与 uic.exe 不同，moc.exe 没有 AUTOMOC_MOC_OPTIONS 源文件属性和相应的以 INTERFACE_ 开头的使用要求。

2、可以使用 SKIP_AUTOMOC 或 SKIP_AUTOGEN 源文件属性将源文件排除在 AUTOMOC 处理之外，即，该源文件不由 CMake 自动调用 moc.exe 命令处理。

3、AUTOMOC_MACRO_NAMES 目标属性

指定一个以分号分隔的名称列表，以确定该名称是否需要使用 moc.exe 来处理一个 C++ 文件。指定的这些名称应含有 Qt 的专有宏。也就是说，若 C++ 源文件（或头文件）中含有由 AUTOMOC_MACRO_NAMES 指定的名称，则会被 moc.exe 处理。默认情况下，AUTOMOC_MACRO_NAMES 是从 CMAKE_AUTOMOC_MACRO_NAMES 变量初始化的。

4、CMAKE_AUTOMOC_MACRO_NAMES 变量

该变量用于初始化所有目标上的 AUTOMOC_MACRO_NAMES 属性。其默认值为：

Q_OBJECT;Q_GADGET;Q_NAMESPACE;Q_NAMESPACE_EXPORT;Q_GADGET_EXPORT

5、INTERFACE_AUTOMOC_MACRO_NAMES 目标属性

这是使用要求（详见使用要求章节的讲解），默认为空值，作用与 AUTOMOC_MACRO_NAMES 目标属性类似，可用于传递。当一个启用了 AUTOMOC 的目标链接到一个设置了 INTERFACE_AUTOMOC_MACRO_NAMES 的库时，目标将继承列出的宏名称，并将它们与它自己的 AUTOMOC_MACRO_NAMES 属性中指定的宏名称合并。然后，目标将自动为源文件生成 MOC 文件，其中也包含继承的宏名称，而不仅仅是在其自己的 AUTOMOC_MACRO_NAMES 属性中指定的宏名称。

6、AUTOGEN_TARGET_DEPENDS 目标属性

由于具有 AUTOUIC 或 AUTOMOC 属性的目标 CMake 会为其创建一个为*_autogen 的目标，该目标用于生成 moc 和 uic 文件，由于这个目标是在生成时创建的，所以不能使用 add_dependencies()命令来添加其依赖关系。相反，应使用 AUTOGEN_TARGET_DEPENDS 目标属性设置*_autogen 目标的依赖项列表，其值可以是目标名或文件名。

7、表 14.3 是 CMake 自动调用 moc.exe 时相关的变量和属性

表 14.3 CMake 自动调用 moc.exe 时相关的变量和属性

类别	名称	说明
目标属性	AUTOGEN_BUILD_DIR	与 uic.exe 相同，请参阅表 4.2
	AUTOMOC_MOC_OPTIONS	为 moc.exe 提供的额外的参数，若在创建目标时设置 CMAKE_AUTOMOC_MOC_OPTIONS 变量，则由该变量的值初始化，否则为空字符串
	AUTOMOC_MACRO_NAMES	指定一个以分号分隔的名称列表，以确定是否需要使用 moc.exe 来处理一个 C++ 文件，默认情况下是从 CMAKE_AUTOMOC_MACRO_NAMES 变量初始化的。
	INTERFACE_AUTOMOC_MACRO_NAMES	这是使用要求。默认为空值，作用与 AUTOMOC_MACRO_NAMES 目标属性类似，可用于传递。
	AUTOGEN_TARGET_DEPENDS	为具有 AUTOUIC 或 AUTOMOC 属性的目标创建的*_autogen 目标添加的依赖项列表，其值可以是目标名或文件名
源文件属性	SKIP_AUTOMOC	将源文件排除在 AUTOMOC 处理之外(对于 Qt 项目)。
	SKIP_AUTOGEN	从 AUTOMOC，AUTOUIC 和 AUTORCC 处理中排除源文件(对于 Qt 项目)。
变量	CMAKE_AUTOMOC_MOC_OPTIONS	为 moc.exe 提供的额外的参数，该变量用于初始化所有目标上的 AUTOMOC_MOC_OPTIONS 目标属性
	CMAKE_AUTOMOC_MACRO_NAMES	该变量用于初始化所有目标上的 AUTOMOC_MACRO_NAMES 属性。其默认值为：Q_OBJECT;Q_GADGET;Q_NAMESPACE;Q_NAMESPACE_EXPORT;Q_GADGET_EXPORT

14.4 使用 Qt 提供的 CMake 命令构建 Qt 程序

- 1、为便于构建 Qt 程序，Qt 对原有的一些 CMake 命令进行了包装(或扩展)，以方便处理 Qt 专有的事项。Qt 包装的 CMake 命令绝大部分在 Qt6 包的 Core 组件中定义，本文在此后使用 Qt6::Core 表示 Qt6 包中的 Core 组件，同理 Qt6::DBus 表示 Qt6 包中的 DBus 组件。
- 2、以下是一些常用的定义于 Qt6::Core 中的命令，本小节将讲解以下命令，其余命令可参阅 Qt 官方帮助文档

```
qt_standard_project_setup()
qt_add_executable()
qt_add_library()
```

qt_finalize_target()

14.4.1 qt_standard_project_setup()命令

其语法为:

```
qt_standard_project_setup(  
    [REQUIRES <version>]  
    [SUPPORTS_UP_TO <version>])
```

- 1、该命令定义于 Qt6::Core 组件，在 Qt6.3 中引入。该命令的主要作用是处理了常规的构建 Qt 程序的工作。通过将 QT_NO_STANDARD_PROJECT_SETUP 变量(定义于 Qt6::Core 组件)设置为 true 可以禁用 qt_standard_project_setup()命令。
- 2、调用 qt_standard_project_setup()命令的位置
应在第一次使用 find_package(Qt6)语句之后立即调用，通常在顶级的 CMakeLists.txt 文件中，并且在定义任何目标之前。
- 3、qt_standard_project_setup()命令各参数意义如下：
 - REQUIRES 参数：适用 Qt 6.5 及以上版本，指定必须满足的 Qt 最低版本。若指定此参数，则会在 Qt 中启用所有的建议更改，这会导致较旧的 Qt 版本产生错误。
 - SUPPORTS_UP_TO 参数：表示启用指定版本之前中引入的任何更改(即兼容之前的旧版本)。类似于 CMake 的 cmake_policy()命令。
- 4、qt_standard_project_setup()命令处理的事情如下：
 - 1)、设置 CMake 变量 CMAKE_AUTOMOC、CMAKE_AUTOUIC 的值(若未设置)为 true，这是使用 qt_standard_project_setup()命令的主要原因。默认情况下，其作用范围为当前及以下的目录范围。注意：需在 qt_standard_project_setup()命令之后创建的目标才会生效。
 - 2)、自动包含 CMake 的 GNUInstallDirs 模块，这为 CMAKE_INSTALL_BINDIR、CMAKE_INSTALL_LIBDIR 等变量定义了适当的默认值。
 - 3)、若目标是 Windows 时，若 CMAKE_RUNTIME_OUTPUT_DIRECTORY 变量未设置，则将其设置为 \${CMAKE_CURRENT_BINARY_DIR}
 - 4)、若目标平台不是 Apple 或 Windows 时(但需支持 RPATH)，扩展 CMAKE_INSTALL_RPATH 变量，具体怎样扩展的，详见 Qt 官方帮助文档。CMAKE_INSTALL_RPATH 变量的作用是指定要在已安装的目标中使用的 rpath(仅对支持的平台)
 - 5)、将 CMake 的 USE_FOLDERS 全局属性设置为 ON，将 Qt 定义的属性 QT_TARGETS_FOLDER 设置为 QtInternalTargets，支持文件夹的 IDE 将在此文件夹中显示 Qt 内部目标(Qt-internal target)。其中 QT_TARGETS_FOLDER 是在 Qt6::Core 组件中定义的属性，其值是由 Qt 的 CMake 命令添加的内部目标文件夹的名称。此属性仅在 CMake 的 USE_FOLDERS 属性为 ON 时生效，在 Qt 6.5 中引入，处于预览状态，在将来版本中可能会更改，默认情况下该属性没有值。

示例 14.5：使用 qt_standard_project_setup()命令构建 Qt 程序

- ①、在目录 g:/qt3 下的 d.cpp 中编写如下的 Qt 代码

```
//g:/qt3/d.cpp  
#include<QObject>
```

```
class A:public QObject {
    Q_OBJECT          //这是 Qt 专有宏，需启用 moc.exe 进行处理
public:    A(){qDebug( "DDD\n");}    };
//以下语句必须位于类 A 之后，因为由 d.moc 中用到了类 A，此时必须见到类 A 的完整定义。
#include "d.moc"
void main(){A ma;}
```

②、在目录 g:/qt3 下的 CMakeLists.txt 中编写如下代码

#g:/qt3/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.27)
project(XXXX LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_PREFIX_PATH C:/Qt/6.6.0/msvc2019_64/lib/cmake/Qt6)
find_package(Qt6 COMPONENTS Widgets )

#以下命令必须位于 find_package() 命令之后。该命令可以替换对 CMAKE_AUTOMOC 变量的设置
qt_standard_project_setup()
add_executable(nn d.cpp)
target_link_libraries(nn PUBLIC Qt6::Widgets )
```

③、在 CMD 中转至 g:/qt3 并输入以下命令

```
cmake .
mingw32-make
nn.exe          //输出 DDD，符合预期
```

14.4.2 qt_add_executable()命令

其语法为：

```
qt_add_executable(target
    [WIN32] [MACOSX_BUNDLE]
    [MANUAL_FINALIZATION]
    sources...)
```

1、该命令定义于 Qt6::Core 组件，在 Qt 6.0 中引入，该命令必须位于 find_package(Qt6)之后。该命令是 CMake 内置的 add_executable() 命令的包装器，执行以下工作：

- 为目标平台创建一个 CMake 目标。在除 Android 之外的所有平台上，所有参数都将传递给标准的 CMake add_executable() 命令，除了 MANUAL_FINALIZATION 参数(如果存在)外。在 Android 上，将创建一个 MODULE 库，任何 WIN32 或 MACOSX_BUNDLE 参数将被忽略。
- 将目标链接到 Qt::Core 库。
- 处理 CMake 目标的终止化(finalization)。创建目标后，通常还需要进一步处理或执行终止化步骤，这取决于平台、目标的类型和目标的各种属性，通常由用 qt_finalize_target() 命令实现，详见后文。当使用 CMake 3.19 或更高版本时，目标终止化会自动延迟到当前目录作用域的末尾。当使用低于 3.19 的 CMake 版本时，不支持自动延迟。在这种情况下，在该命令返回之前立即执行目标终止化。

2、MANUAL_FINALIZATION 参数用来指示将在以后的某个时间显式地调用 `qt_finalize_target()`命令以处理终止化。通常，不应该使用该参数，除非项目必须支持 CMake 3.18 或更早的版本。当使用 CMake 3.19 或更高版本时，会自动终止化项目。当使用旧的 CMake 版本时，应该在根 CMakeLists.txt 文件的末尾手动调用，这对 Android 很重要。

3、`qt_add_executable()`命令的使用方法与 CMake 的 `add_executable()`命令相同，从略。

4、需要注意的是，在使用了 `qt_add_executable()`命令后，需在 `target_link_libraries()`命令中明确指定 PUBLIC、PRIVATE 等参数，比如

```
qt_add_executable(nn ...)  
....  
#target_link_libraries(nn aa ....)           #错误  
target_link_libraries(nn PUBLIC aa....)      #需指定 PUBLIC、PRIVATE 等参数
```

14.4.3 qt_add_library()命令

其语法为：

```
qt_add_library(target  
    [STATIC | SHARED | MODULE | INTERFACE | OBJECT]  
    [MANUAL_FINALIZATION]  
    sources... )
```

1、该命令定义于 Qt6::Core 组件，在 Qt6.2 中引入，该命令必须位于 `find_package(Qt6)`之后。该命令是 CMake 内置的 `add_library()`命令的包装器，执行以下工作：

- 创建一个 CMake 库目标。提供的任何 sources 都将传递给对应的 `add_library()`的内部调用。
- 处理 CMake 目标的终止化(finalization)。在 Qt 当前版本中，库目标的终止化不执行任何操作，但在未来版本可能会添加功能。终止化的其他规则与 `qt_add_executable()`命令类似，从略。

2、如果没有指定创建的库的类型，则创建的库类型取决于 Qt 是如何构建的。如果 Qt 是静态构建的，那么将创建一个静态库，否则将创建共享库。注意，这与 CMake 的 `add_library()`命令的工作方式不同，其中 CMake 变量 BUILD_SHARED_LIBS 控制创建的库的类型，`qt_add_library()`命令在决定库类型时不考虑 BUILD_SHARED_LIBS。

3、`qt_add_library` 命令的使用方法与 CMake 的 `add_library ()`命令相同，从略。

14.4.4 qt_finalize_target()命令

其语法为：

```
qt_finalize_target(target)
```

1、该命令定义于 Qt6::Core 组件，在 Qt6.2 中引入，该命令必须位于 `find_package(Qt6)`之后。

2、创建目标后，通常还需要进一步处理或执行终止化步骤，这取决于平台、目标的类型和目标的各种属性。这些步骤预计将在与创建目标的目录范围相同的目录范围内执行，因此也应该从同一目录范围调用该命令。

3、该命令执行以下操作：

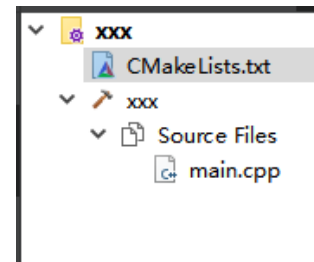
- 对于可执行目标，该命令执行一些内部处理以促进 Qt 插件的自动链接，这对于静态构建的 Qt 程序或低于 3.21 版本的 CMake 很重要，该命令的其他处理事项因使用的平台(如 iOS、Android)而异，以后的 Qt 版本可能还会进一步扩展。
- 对于库目标，在 Qt 当前版本中，终止化不执行任何操作，但在未来版本可能会添加功能

14.5 在 Qt 中使用 CMake 构建 Qt 程序

1、要在 Qt 中使用 CMake 需在安装 Qt 时安装 CMake。 本文以示例的形式讲解。

2、创建一个纯 C++应用项目

打开 Qt Creator，选择【文件】-->【New Project ...】，然后在对话框左侧选择“Non-Qt Project”，右侧选择“Plain C++ Application”，然后项目名称输入“xxx”，路径选择“g:/qt4”，然后在选择构建系统时选择“CMake”，其余选项按常规要求即可，完成后的项目结构如右图所示。



3、查看创建的目录及文件

在 g:/qt4 目录中可以看到名为“xxx”和“build-xxx-VC_64bit-Debug”的文件夹，其中 xxx 文件夹中包含有 Qt 的源文件 main.cpp 和 CMakeLists.txt 文件。build-xxx-VC_64bit-Debug 文件夹是调用 cmake.exe 构建程序时的输出目录，其中的文件夹 CMakeFiles 和文件 Makefile、CMakeCache.txt 等都是我们比较熟悉的 CMake 生成的文件。

4、查看 CMakeLists.txt 文件的内容，该文件内容如下：

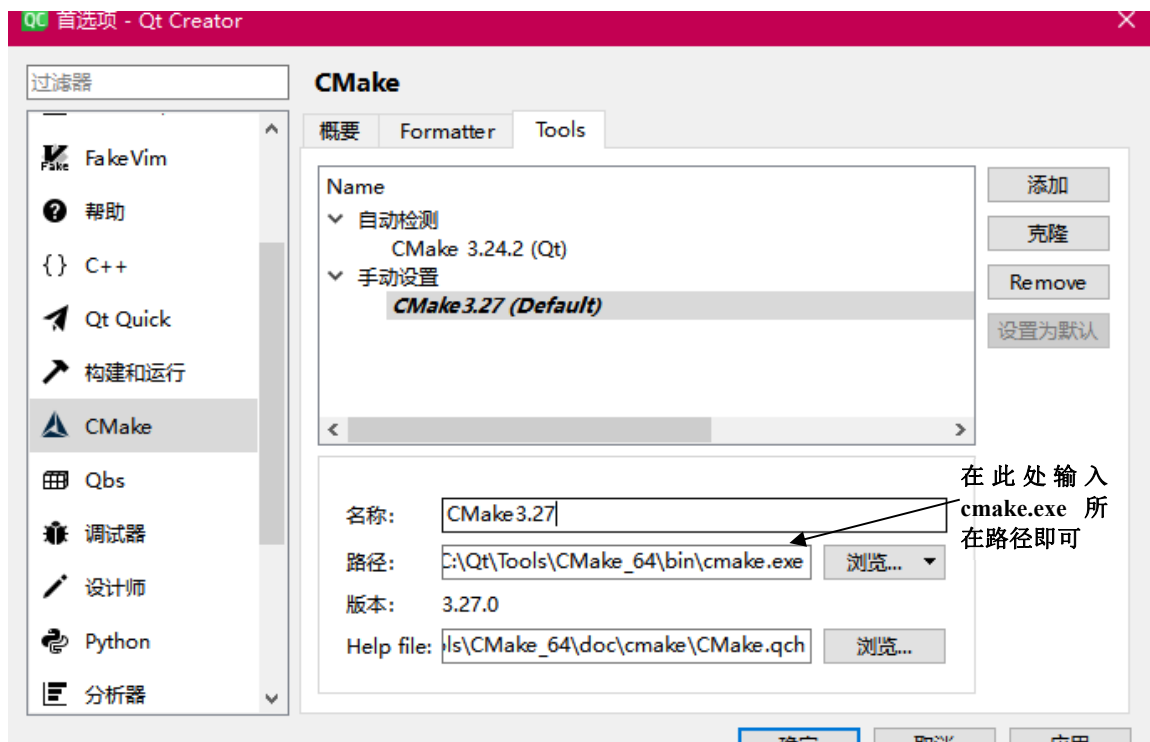
```
cmake_minimum_required(VERSION 3.5)
project(xxx LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 17)                                #指定 C++标准 17
set(CMAKE_CXX_STANDARD_REQUIRED ON)                       #必须满足 C++17 标准
add_executable(xxx main.cpp)
#以下是安装代码，对于本示例而言，这是不必要的
include(GNUInstallDirs)
install(TARGETS xxx                                        #安装(默认情况下，可理解为复制)目标 xxx 生成的文件到指定目录
        LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
        RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR} )
```

4、设置 CMake 相关项

第一次使用 Qt Creator 可能需要对 CMake 相关项进行一些设置，下面讲解这些设置。

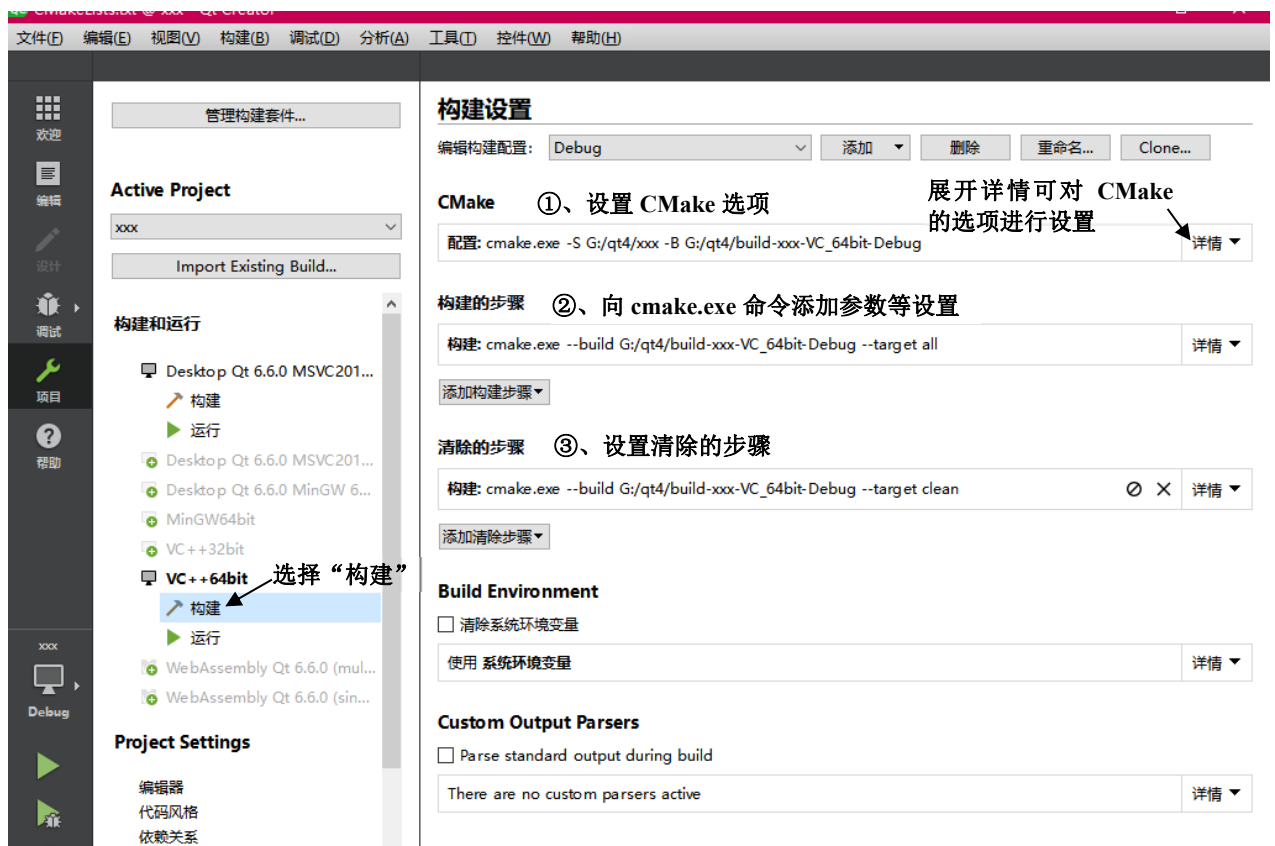
1)、添加 cmake.exe 工具

在 Qt Creator 中选择【编辑】-->【Preferences】，在弹出的对话框的左侧选择“CMake”，然后在右侧选择“Tools”，如下图所示

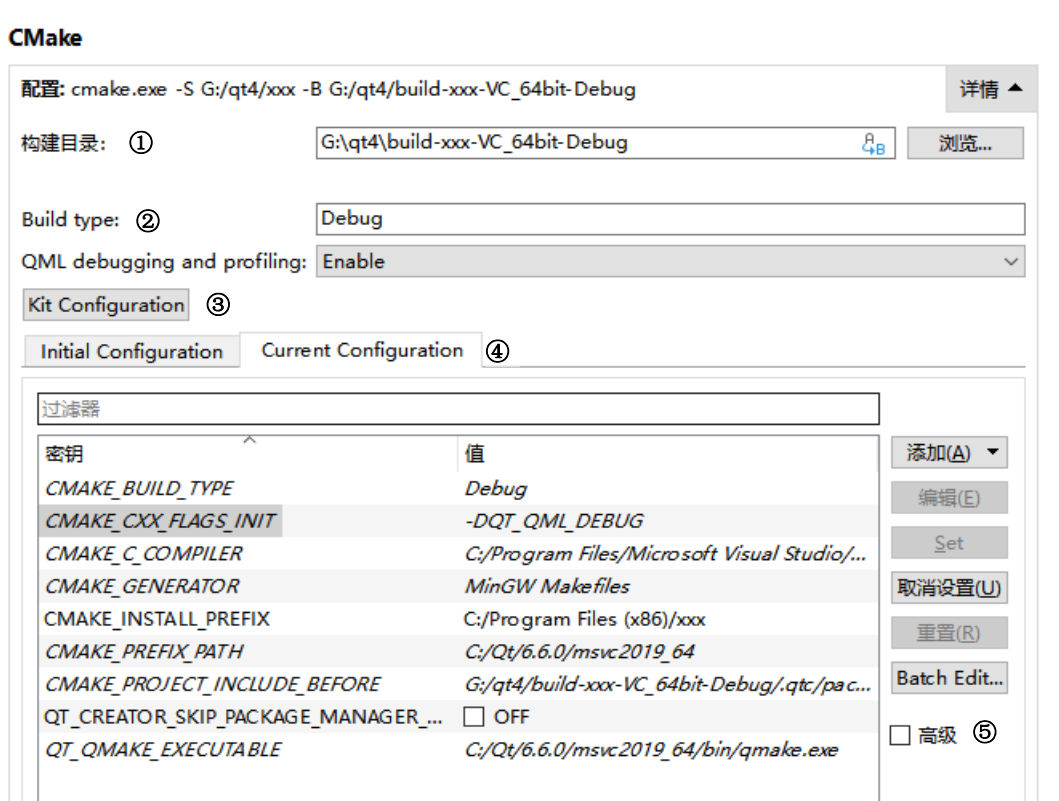


2)、设置 CMake 相关配置

在 Qt Creator 的左侧栏点击【项目】(或按下 Ctrl+5)，然后在左侧选择之前配置好的构件再在其下选择“构建”，出现如下图所示界面



上图中的①处可对 CMake 进行一些设置，点击右侧的详情，其界面如下图所示，



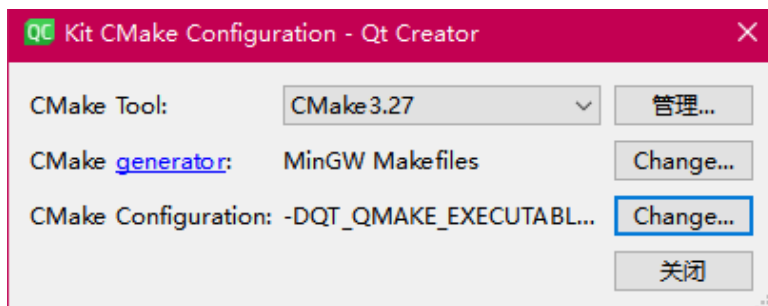
上图中各选项如下：

①处是执行 cmake.exe 命令后文件的输出目录(即构建目录)

②处可设置构建模式，如 Debug(调试模式)、Release(发布模式)

③处可选择 CMake 生成器，点击 3 处的按钮弹出如下图所示界面。

④处是列出的 CMake 变量的初始值及当前值，点击 5 处的复选框可以显示 CMake 的更多变量及其值，我们可以在这里可视化的设置 CMake 变量的值，这还是比较方便的。




CMake Tool: 选择 cmake.exe 程序，即，我们在【Preferences...】中配置好的 cmake.exe。

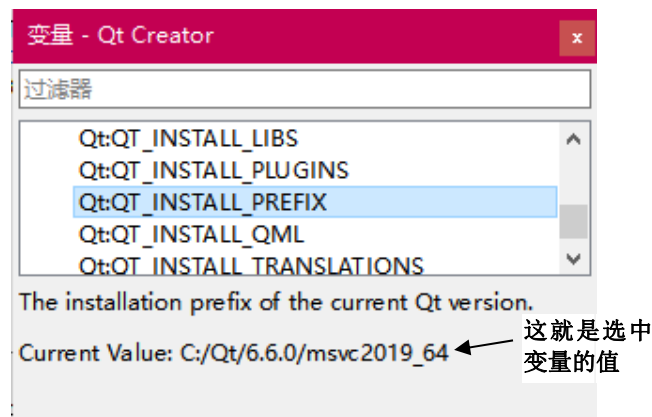
CMake generator: 选择 CMake 的生成器，如 MinGw Makefiles(本示例选择此生成器)、Ninja、jom、NMake 等

CMake Configuration: 在这里可设置传递给 cmake.exe 的-D 参数(即指定一个宏)，点击右侧的“Change...”按钮，在其中可以看到 Qt 预设的如下的宏，也可在其中输入自己的宏

```
-DQT_QMAKE_EXECUTABLE:FILEPATH=%{Qt:qmakeExecutable}
-DCMAKE_PREFIX_PATH:PATH=%{Qt:QT_INSTALL_PREFIX}
-DCMAKE_C_COMPILER:FILEPATH=%{Compiler:Executable:C}
-DCMAKE_CXX_COMPILER:FILEPATH=%{Compiler:Executable:Cxx}
```

其中 CMAKE_PREFIX_PATH 是 CMake 变量，这个变量的值将影响 CMake 命令 find_package(Qt

6)查找 CMake 包的路径，以上指定的宏右侧的以%开头的值，可点击按钮 ，然后在弹出的对话框中查看，如下图所示



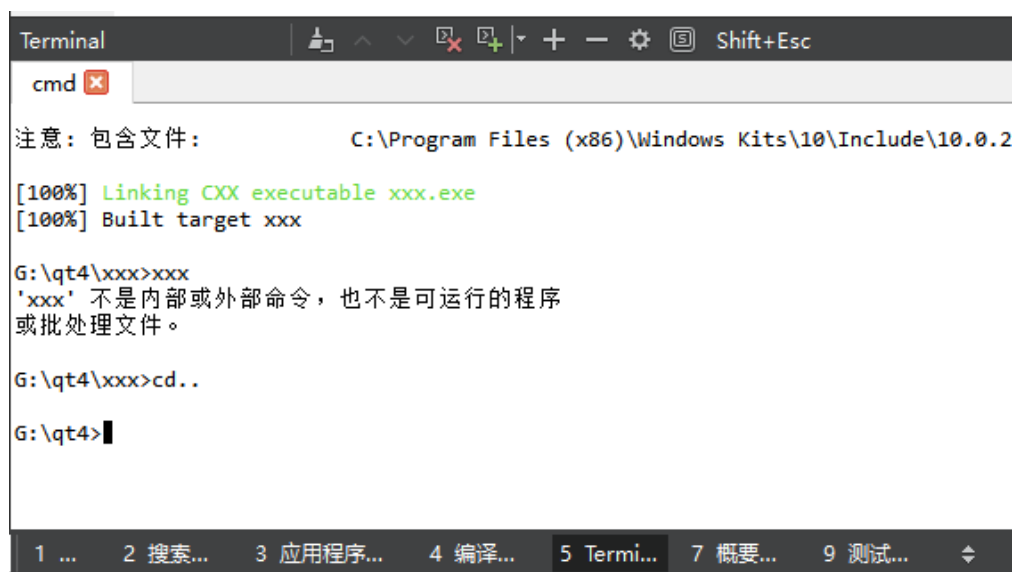
5、构建程序

本示例准备使用以下配置运行程序

- 使用的生成器为 MinGw Makefiles
- 编译器为 VC++

选择好配置后，直接在 Qt Creator 中按下 “Ctrl+R” 即可运行。

除此之外还可在 Qt Creator 的 CMD 终端中以命令的形式构建程序，其方法为：Qt Creator 底部点击 “Terminal” (若未发现，则可以按下 “Alt+6”)，其界面如下图所示，然后，我们便可以像在 CMD 中一样使用命令构建程序了。



6、构建 Qt 程序

以上的代码是纯 C++ 代码，现在我们准备构建一个 Qt 程序，为此，在 Qt Creator 的 main.cpp 中输入以下代码

```
//main.cpp

#include <QDialog>           //在 CMake 的 find_package(Qt6) 语句执行前，包含的 Qt 头文件
                             //Qt Creator 都会提示找不到头文件的错误(因为还未加载 Qt 包)

#include <QLabel>
#include <QApplication>
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    QApplication a(argc,argv);
    cout << "Hello World!" << endl;

    QDialog d;                //创建一个对话框，Qt 使用 QDialog 类对象创建对话框。
    d.resize(200,100);        //设置对话框的大小。
    QLabel s(&d);              //创建一个标签，并把该标签放在对话框 d 之中。
    s.setText("AAA");          //设置标签显示的文本。
    s.move(50,50);             //设置标签位于对话框中的位置。
    d.show();                  //显示对话框，默认情况下部件是不可见的，因此需显示。
    a.exec();                  //程序进入消息(或事件)循环。
    return 0;    }
}
```

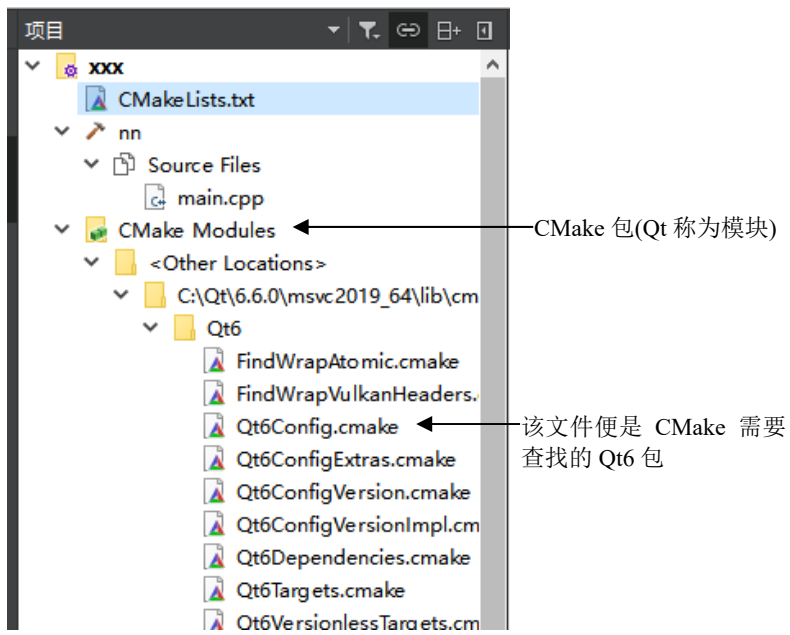
然后在 CMakeLists.txt 文件中输入以下代码

```
#CMakeLists.txt

cmake_minimum_required(VERSION 3.27)      #版本 3.27，Qt 预编写代码的 CMake 版本为 3.5
project(xxx LANGUAGES CXX)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
find_package(Qt6 COMPONENTS Widgets )    #查找 Qt6Config.cmake 文件(CMake 包 Qt6)
qt_standard_project_setup()               #自动调用 moc.exe、uic.exe 工具(本示例不需要)
add_executable(nn main.cpp )
target_link_libraries(nn PUBLIC Qt6::Widgets )
#以下是安装代码，对于本示例而言，这是不必要的。
include(GNUInstallDirs)
install(TARGETS nn
        LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
        RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR} )
```

注：以上代码未设置查找 CMake 包 Qt6Config.cmake 路径的变量 CMAKE_PREFIX_PATH 的值，这是因为 Qt Creator 已经为我们设置好了。

- 7、编写好以上代码后，在 Qt Creator 中按下 Ctrl+R 即可运行程序，该程序会弹出一个简单的对话框。执行 Qt 程序后，Qt Creator 项目栏的界面如下图所示



第 15 章 交叉编译(指定编译器等工具)

1、需要注意的是，编译器和生成器是两种不同类型的工具，第一章已讲解过怎样指定生成器，本章重点讲解怎样指定编译器等其他工具的方法，下面把指定生成器的方法再次罗列出来，以方便以后查阅。

2、生成器可使用以下方式指定

- 在命令行中使用 `cmake.exe` 工具的 `-G` 参数指定，比如

```
cmake . -G "MinGW Makefiles"
```

- 创建一个以下的系统环境变量

```
CMAKE_GENERATOR = MinGW Makefiles
```

建议使用环境变量指定生成器，以免每次都要指定生成器。注意：若安装有 VS，会首先默认生成 VS 的项目文件(proj 文件)，而不会生成 makefile 文件。

- 注意：不要在 CMake 项目(即 CMakeLists.txt 文件)中使用 `CMAKE_GENERATOR` 变量指定生成器，该变量在 CMake 文件中使用是无效的。这意味着生成器只能使用以上两种方法指定。

3、工具链：CMake 将用于编译、链接等的工具统称为工具链(toolchain)

4、默认工具链的选择

CMake 根据系统检测和默认值自动确定主机构建的工具链，这意味着，没有办法更改 CMake 选择的默认工具链。若不想使用默认的工具链，则需要使用本章介绍的方法。

5、指定工具链的方法

CMake 使用设置变量的方法来指定工具链，因为各工具牵涉到的选项比较多，这也意味着需要设置较多的变量，如果每次执行 CMake 代码都设置一次变量这会非常麻烦，CMake 使用文件的形式来解决这个问题，即，把设置好的所有变量保存在一个文件中(一个纯文本文件)，CMake 将这个文件称为**工具链文件**，然后使用变量 `CMAKE_TOOLCHAIN_FILE` 加载这个文件。

6、不建议在项目文件中(此处是指在 `project()` 命令之后)设置工具链的相关变量，应使用

`CMAKE_TOOLCHAIN_FILE` 变量来指定工具链。在项目文件中设置工具链文件会设置较多的变量。

7、指定工具链文件的方法有如下几种：

- 在 `project()` 命令之前设置 `CMAKE_TOOLCHAIN_FILE` 变量的值以指定工具链文件。在 4.1.2 小节的讲解的 `project()` 命令的执行步骤可知，在执行 `project()` 命令之前，会优先读取 `CMAKE_TOOLCHAIN_FILE` 变量的值。注意：不要在 `project()` 命令之后设置 `CMAKE_TOOLCHAIN_FILE` 变量的值，这会不起作用。
- 使用 `cmake` 命令的 `-D` 参数在 CMD 命令行设置 `CMAKE_TOOLCHAIN_FILE` 变量的值。
- 将 `CMAKE_TOOLCHAIN_FILE` 变量的值设置为系统环境变量。使用环境变量可达到指定默认工具链的效果。

8、下面是工具链文件通常需要设置的一些变量(注意：并不是所有变量都必须指定，可根据实际需要而

指定)

- 使用的编程语言可在 `project()` 命令中指定。
- `CMAKE_<LANG>_COMPILER`: 指定<LANG>语言的编译器的路径
- `CMAKE_<LANG>_FLAGS`: 指定<LANG>语言编译器的编译参数。
- `CMAKE_LINKER`: 指定链接器的路径。
- `CMAKE_SYSTEM_NAME`: 指定操作系统名称, 如 Windows、Linux、WindowsCE 等等。
- `CMAKE_MAKE_PROGRAM`: 指定生成器的路径。
- `CMAKE_<LANG>_COMPILER_ID`: 指定编译器的 ID
- `CMAKE_<LANG>_COMPILER_VERSION`: 指定编译器的版本

9、下面是使用 g++ 编译器的一些设置(仅供参考), 以下变量不需全都指定, 可根据实际需要而指定

```
#1、指定 C++编译器
set(CMAKE_CXX_COMPILER "C:/Qt/Tools/mingw1120_64/bin/g++.exe")
#2、指定 C 编译器
set(CMAKE_C_COMPILER "C:/Qt/Tools/mingw1120_64/bin/gcc.exe")
#3、指定链接器。若构建的程序需要链接, 则必须指定链接器
set(CMAKE_LINKER "C:/Qt/Tools/mingw1120_64/bin/ld.exe")
#4、指定 C++标准库
set(CMAKE_CXX_STANDARD_LIBRARIES "-lkernel32 -luser32 -lgdi32 -lwinspool -lshell32 -lole32 -loleaut32 -luuid -lcomdlg32 -ladvapi32")
#5、指定 C 标准库
set(CMAKE_C_STANDARD_LIBRARIES "-lkernel32 -luser32 -lgdi32 -lwinspool -lshell32 -lole32 -loleaut32 -luuid -lcomdlg32 -ladvapi32")
#6、若指定了对应的生成器, 以下变量应设置为缓存变量, 否则会出错。
set(CMAKE_MAKE_PROGRAM "C:/Qt/Tools/mingw1120_64/bin/mingw32-make.exe" CACHE STRING
xxx)
#7、使用的系统名称
set(CMAKE_SYSTEM_NAME Windows)
```

10、下面是使用 MSVC 编译器的一些设置(仅供参考), 以下变量不需全都指定, 可根据实际需要而指定

```
#1、指定 C++编译器
set(CMAKE_CXX_COMPILER "C:/Program Files/Microsoft Visual Studio/2022/Professional/VC/Tools/MSVC/14.35.32215/bin/HostX64/x64/cl.exe")
#2、指定 C 编译器
set(CMAKE_C_COMPILER "C:/Program Files/Microsoft Visual Studio/2022/Professional/VC/Tools/MSVC/14.35.32215/bin/HostX64/x64/cl.exe")
#3、指定链接器。若构建的程序需要链接, 则必须指定链接器
set(CMAKE_LINKER "C:/Program Files/Microsoft Visual Studio/2022/Professional/VC/Tools/MSVC/14.35.32215/bin/Hostx64/x64/link.exe")
#4、指定 C++标准库
set(CMAKE_CXX_STANDARD_LIBRARIES "kernel32.lib user32.lib gdi32.lib winspool.lib shell32.lib ole32.lib oleaut32.lib uuid.lib comdlg32.lib advapi32.lib")
#5、指定 C 标准库
set(CMAKE_C_STANDARD_LIBRARIES "kernel32.lib user32.lib gdi32.lib winspool.lib shell32.lib ole32.lib oleaut32.lib uuid.lib comdlg32.lib advapi32.lib")
```


#6、若指定了对应的生成器，以下变量应设置为缓存变量，否则会出错。

```
set(CMAKE_MAKE_PROGRAM "C:/Qt/Tools/mingw1120_64/bin/mingw32-make.exe" CACHE STRING  
xxx)
```

#7、使用的系统名称

```
set(CMAKE_SYSTEM_NAME Windows)
```

示例 15.1：使用工具链文件为 CMake 指定编译器

1、准备工具链文件

在 g:/qt1/yy 目录下的 g++.txt 文件中编写如下代码，本示例准备将 g++.txt 作为工具链文件，注意：工具链文件就是一个纯文本文件，对文件后缀没有强制要求。

#g:/qt1/yy/tt.txt

#指定 C++编译器

```
set(CMAKE_CXX_COMPILER "C:/Qt/Tools/mingw1120_64/bin/g++.exe")
```

#指定 C 编译器

```
set(CMAKE_C_COMPILER "C:/Qt/Tools/mingw1120_64/bin/gcc.exe")
```

#指定链接器

```
set(CMAKE_LINKER "C:/Qt/Tools/mingw1120_64/bin/ld.exe")
```

②、C++源文件准备。读者自行准备。

③、在目录 g:/qt1 中的 CMakeLists.txt 中编写如下代码

#g:/qt1/CMakeLists.txt

#本示例在 project() 命令之前指定工具链文件。在执行 project() 命令之前，会优先读取以下变量的值

```
set(CMAKE_TOOLCHAIN_FILE g:/qt1/yy/g++.txt)
```

```
cmake_minimum_required(VERSION 3.27)
```

```
project(XXXX)
```

```
add_library(kk b.cpp)
```

```
add_executable(nn a.cpp)
```

```
target_link_libraries(nn kk)
```

#不要在项目文件中使用以下变量指定生成器，该变量在项目文件中不会生效

```
#set(CMAKE_GENERATOR "MinGW Makefiles")
```

④、在 CMD 中转至 g:/qt1 并输入以下命令

```
cmake .
```

```
mingw32-make install
```

```
nn.exe
```

读者可自行查看 CMake 生成的文件是否是使用 g++编译器编译的，从略。

11、下面是更全的使用 g++编译器的一些设置(仅供参考)，以下变量不需全都指定，可根据实际需要而指定

#g++编译器

```
set(CMAKE_ADDR2LINE "C:/Qt/Tools/mingw1120_64/bin/addr2line.exe")
```

```
set(CMAKE_AR "C:/Qt/Tools/mingw1120_64/bin/ar.exe")
```

#编译模式

```
set(CMAKE_BUILD_TYPE Debug)
```

#C++编译器及其参数

```

set(CMAKE_CXX_COMPILER "C:/Qt/Tools/mingw1120_64/bin/g++.exe")
set(CMAKE_CXX_COMPILER_AR "C:/Qt/Tools/mingw1120_64/bin/gcc-ar.exe")
set(CMAKE_CXX_COMPILER_RANLIB "C:/Qt/Tools/mingw1120_64/bin/gcc-ranlib.exe")
set(CMAKE_CXX_FLAGS )
set(CMAKE_CXX_FLAGS_DEBUG -g)
set(CMAKE_CXX_FLAGS_MINSIZEREL "-Os -DNDEBUG")
set(CMAKE_CXX_FLAGS_RELEASE "-O3 -DNDEBUG")
set(CMAKE_CXX_FLAGS_RELWITHDEBINFO "-O2 -g -DNDEBUG")
#C++标准库
set(CMAKE_CXX_STANDARD_LIBRARIES "-lkernel32 -luser32 -lgdi32 -lwinspool -lshell32
-lole32 -loleaut32 -luuid -lcomdlg32 -ladvapi32")
#C 编译器及其参数
set(CMAKE_C_COMPILER "C:/Qt/Tools/mingw1120_64/bin/gcc.exe")
set(CMAKE_C_COMPILER_AR "C:/Qt/Tools/mingw1120_64/bin/gcc-ar.exe")
set(CMAKE_C_COMPILER_RANLIB "C:/Qt/Tools/mingw1120_64/bin/gcc-ranlib.exe")
set(CMAKE_C_FLAGS )
set(CMAKE_C_FLAGS_DEBUG -g)
set(CMAKE_C_FLAGS_MINSIZEREL "-Os -DNDEBUG")
set(CMAKE_C_FLAGS_RELEASE "-O3 -DNDEBUG")
set(CMAKE_C_FLAGS_RELWITHDEBINFO "-O2 -g -DNDEBUG")
#C 标准库
set(CMAKE_C_STANDARD_LIBRARIES "-lkernel32 -luser32 -lgdi32 -lwinspool -lshell32 -
lole32 -loleaut32 -luuid -lcomdlg32 -ladvapi32")
#dlltool 工具
set(CMAKE_DLLTOOL "C:/Qt/Tools/mingw1120_64/bin/dlltool.exe")
#安装前缀
set(CMAKE_INSTALL_PREFIX "C:/Program Files (x86)/xxx")
#链接器
set(CMAKE_LINKER "C:/Qt/Tools/mingw1120_64/bin/ld.exe")
#生成器的路径。若指定了对应的生成器，以下变量应设置为缓存变量，否则会出错。
set(CMAKE_MAKE_PROGRAM "C:/Qt/Tools/mingw1120_64/bin/mingw32-make.exe" CACHE
STRING xxx)
#其他工具
set(CMAKE_NM "C:/Qt/Tools/mingw1120_64/bin/nm.exe")
set(CMAKE_OBJCOPY "C:/Qt/Tools/mingw1120_64/bin/objcopy.exe")
set(CMAKE_OBJDUMP "C:/Qt/Tools/mingw1120_64/bin/objdump.exe")
set(CMAKE_RANLIB "C:/Qt/Tools/mingw1120_64/bin/ranlib.exe")
set(CMAKE_RC_COMPILER "C:/Qt/Tools/mingw1120_64/bin/windres.exe")
set(CMAKE_READSELF "C:/Qt/Tools/mingw1120_64/bin/readelf.exe")
set(CMAKE_STRIP "C:/Qt/Tools/mingw1120_64/bin/strip.exe")
set(CMAKE_TAPI CMAKE_TAPI-NOTFOUND)
#运行的系统
set(CMAKE_SYSTEM_NAME Windows)
set(CMAKE_SYSTEM_VERSION 10.0.19045)

```

12、下面是更全的使用 MSVC 编译器的一些设置(仅供参考)，以下变量不需全都指定，可根据实际需要而指定

#MSVC 编译器

```
set(CMAKE_AR "C:/Program Files/Microsoft Visual Studio/2022/Professional/VC/Tools/MSVC/14.35.32215/bin/Hostx64/x64/lib.exe")
#编译模式
set(CMAKE_BUILD_TYPE Debug)
#C++编译器及其参数
set(CMAKE_CXX_COMPILER "C:/Program Files/Microsoft Visual Studio/2022/Professional/VC/Tools/MSVC/14.35.32215/bin/HostX64/x64/cl.exe")
set(CMAKE_CXX_FLAGS "/DWIN32 /D_WINDOWS /EHsc")
set(CMAKE_CXX_FLAGS_DEBUG "/Ob0 /Od /RTC1")
set(CMAKE_CXX_FLAGS_MINSIZEREL "/O1 /Ob1 /DNDEBUG")
set(CMAKE_CXX_FLAGS_RELEASE "/O2 /Ob2 /DNDEBUG")
set(CMAKE_CXX_FLAGS_RELWITHDEBINFO "/O2 /Ob1 /DNDEBUG")
#C++标准库
set(CMAKE_CXX_STANDARD_LIBRARIES "kernel32.lib user32.lib gdi32.lib winspool.lib shell32.lib ole32.lib oleaut32.lib uuid.lib comdlg32.lib advapi32.lib")
#C 编译器及其参数
set(CMAKE_C_COMPILER "C:/Program Files/Microsoft Visual Studio/2022/Professional/VC/Tools/MSVC/14.35.32215/bin/HostX64/x64/cl.exe")
set(CMAKE_C_FLAGS "/DWIN32 /D_WINDOWS")
set(CMAKE_C_FLAGS_DEBUG "/Ob0 /Od /RTC1")
set(CMAKE_C_FLAGS_MINSIZEREL "/O1 /Ob1 /DNDEBUG")
set(CMAKE_C_FLAGS_RELEASE "/O2 /Ob2 /DNDEBUG")
set(CMAKE_C_FLAGS_RELWITHDEBINFO "/O2 /Ob1 /DNDEBUG")
#C 标准库
set(CMAKE_C_STANDARD_LIBRARIES "kernel32.lib user32.lib gdi32.lib winspool.lib shell32.lib ole32.lib oleaut32.lib uuid.lib comdlg32.lib advapi32.lib")
#其他链接参数
set(CMAKE_EXE_LINKER_FLAGS /machine:x64)
set(CMAKE_EXE_LINKER_FLAGS_DEBUG "/debug /INCREMENTAL")
set(CMAKE_EXE_LINKER_FLAGS_MINSIZEREL "/INCREMENTAL:NO")
set(CMAKE_EXE_LINKER_FLAGS_RELEASE "/INCREMENTAL:NO")
set(CMAKE_EXE_LINKER_FLAGS_RELWITHDEBINFO "/debug /INCREMENTAL")
#安装前缀
set(CMAKE_INSTALL_PREFIX "C:/Program Files (x86)/xxx")
#链接器
set(CMAKE_LINKER "C:/Program Files/Microsoft Visual Studio/2022/Professional/VC/Tools/MSVC/14.35.32215/bin/Hostx64/x64/link.exe")
#生成器的路径。若指定了对应的生成器，以下变量应设置为缓存变量，否则会出错。
set(CMAKE_MAKE_PROGRAM "C:/Qt/Tools/mingw1120_64/bin/mingw32-make.exe" CACHE STRING xxx)
#链接器参数
```

```

set(CMAKE_MODULE_LINKER_FLAGS /machine:x64)
set(CMAKE_MODULE_LINKER_FLAGS_DEBUG "/debug /INCREMENTAL")
set(CMAKE_MODULE_LINKER_FLAGS_MINSIZEREL /INCREMENTAL:NO)
set(CMAKE_MODULE_LINKER_FLAGS_RELEASE /INCREMENTAL:NO)
set(CMAKE_MODULE_LINKER_FLAGS_RELWITHDEBINFO "/debug /INCREMENTAL")
set(CMAKE_SHARED_LINKER_FLAGS "/machine:x64")
set(CMAKE_SHARED_LINKER_FLAGS_DEBUG "/debug /INCREMENTAL")
set(CMAKE_SHARED_LINKER_FLAGS_MINSIZEREL /INCREMENTAL:NO)
set(CMAKE_SHARED_LINKER_FLAGS_RELEASE /INCREMENTAL:NO)
set(CMAKE_SHARED_LINKER_FLAGS_RELWITHDEBINFO "/debug /INCREMENTAL")
set(CMAKE_STATIC_LINKER_FLAGS /machine:x64)
#其他工具及参数
set(CMAKE_MT "C:/Program Files (x86)/Windows Kits/10/bin/10.0.22621.0/x64/mt.exe")
set(CMAKE_RC_COMPILER "C:/Program Files (x86)/Windows Kits/10/bin/10.0.22621.0/x64/rc.exe")
set(CMAKE_RC_FLAGS -DWIN32)
set(CMAKE_RC_FLAGS_DEBUG "-D_DEBUG")
#运行的系统
set(CMAKE_SYSTEM_NAME Windows)
set(CMAKE_SYSTEM_VERSION 10.0.19045)

```

作者：黄邦勇帅（原名：黄勇）

2023 年 10 月 20 日



参考资料

微软官方帮助文档

GNU Make 官方帮助文档

CMake 官方帮助文档