第十二章 跨平台编译

主讲教师：同济大学计算机科学与技术学院 陈宇飞
同济大学计算机科学与技术学院 龚晓亮

目录

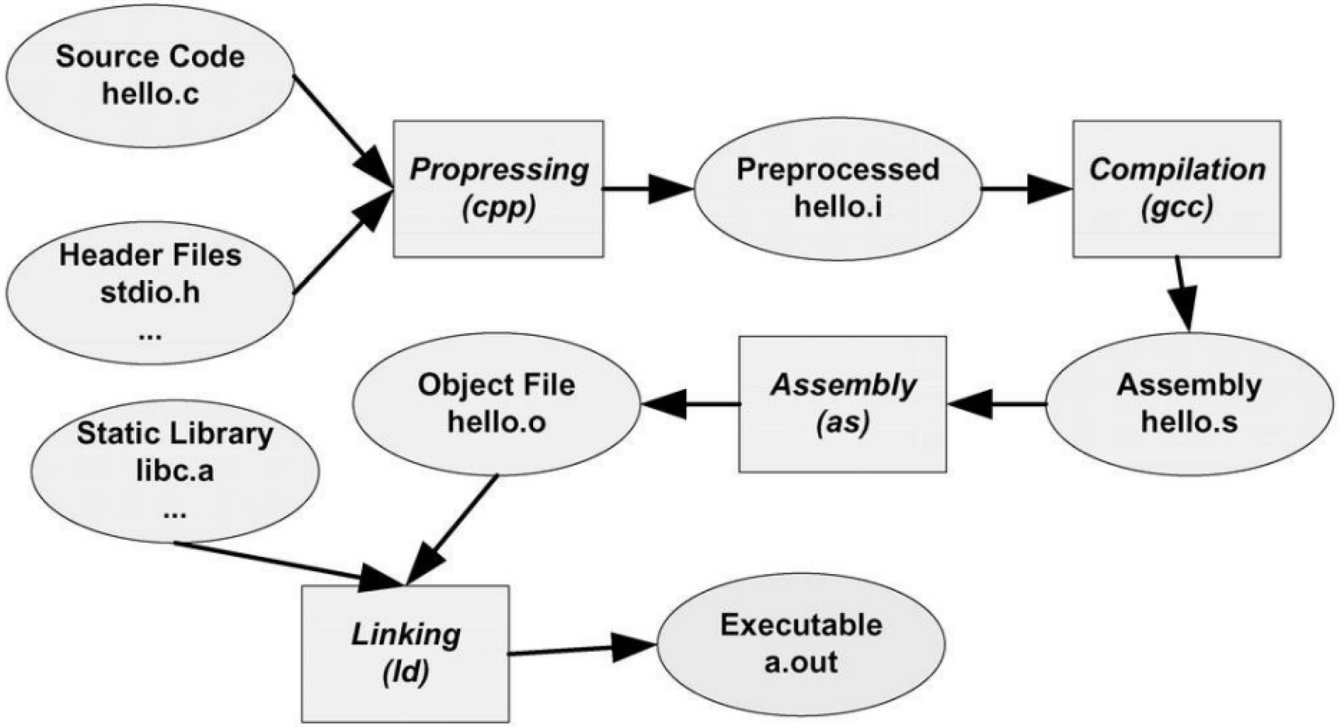
- 跨平台编译概述
- Cmake编译原理
- Makefile的使用

目录

- 跨平台编译概述
  - 需求背景
  - 基本概念

1.1 需求背景

- 编译器会将源代码翻译为目标机器的机器语言指令，链接器则把多个目标文件以及库文件组合起来形成完整的可执行文件。
- 不同操作系统底层架构和运行环境有很大差异，如二进制文件格式、系统调用接口、库文件实现等。因此，平台A（如Windows）编译出的可执行文件很可能无法直接在平台B（如Linux）运行。



1.1 需求背景

- 编程角度：
  - ① 避免使用平台特定的库函数和系统调用；
  - ② 对数据类型进行严格的定义和处理；
  - ③ 合理组织代码结构，隔离平台相关代码等等
- 编译角度：针对不同平台的特性，在编译过程中进行适当的配置和调整，例如使用不同的编译器选项、链接不同的库文件，从而让同一份源代码生成符合不同平台要求的可执行文件。（跨平台编译）

1.2 基本概念

- gcc：GNU编译器套件，相当于编译器，一般用于单文件的项目编译。
- makefile：一种自动化编译的脚本文件，定义了不同文件（如源文件、目标文件）之间的依赖关系，以及将源文件构建为目标文件的具体命令。
- make：一个命令行工具，读取 makefile 文件中的规则和命令，调用相应的编译器和链接器按照其中指定的命令进行编译和链接操作。
- Cmake：一个跨平台的构建系统生成工具。根据项目配置文件生成适合特定平台的构建文件，比如生成 make 工具使用的 makefile 文件。
- 简单来说，gcc 是编译的执行者， makefile 是编译规则的描述者， make 是根据 makefile 进行编译的调度者， Cmake 是更高级的构建文件生成者。

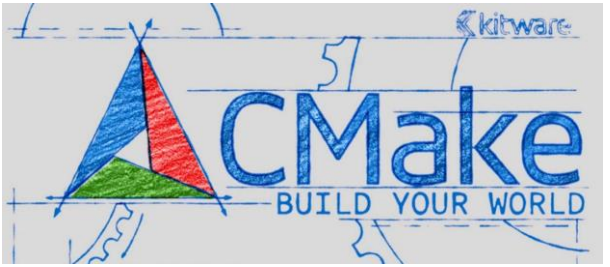
目录

- Cmake编译原理
  - 基本概念
  - 工作流程
  - 指令与变量
  - 控制结构
  - 综合案例

2.1 基本概念

- CMake本身是一个工具集，由五个可执行程序组成：cmake、ctest、cpack、cmake-gui和ccmake
  - cmake：核心工具，负责项目配置、生成构建文件并构建项目，产出可执行或库文件。
  - ctest：测试驱动程序，运行测试用例并报告结果。
  - cpack：打包程序，依据项目构建结果生成安装和源包。
  - cmake-gui：图形界面工具，方便用户设置。
  - ccmake：基于终端的图形界面，在终端以图形化交互修改构建配置。

- 官方网站：https://cmake.org/ （学会查文档）



2.2 工作流程

（1）配置阶段：编写配置文件

- 定义项目的基本信息，如项目名称、项目版本号等。
- 指定源文件和头文件的位置，可用add_executable()定义可执行文件。
- 定义目标文件之间的依赖关系和链接选项。如果项目使用了外部库，需要显式指定链接哪些库。

```
Msg.hpp + Msg.cpp
main.cpp
  → #include "Msg.hpp"
```

```
# CMakeLists.txt （CMake语法中，注释使用 # ）
# 设置最低 CMake 版本，低于 3.5 则报错
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
# 定义项目名为my_project，定义语言为 CXX (C++)
project(my_project LANGUAGES CXX)
# 从源文件生成静态库，名为 msg
add_library(msg STATIC Msg.hpp Msg.cpp)
# 创建可执行文件 hello-world
add_executable(hello-world main.cpp)
# 把 msg 库链接到 hello-world 可执行文件
target_link_libraries(hello-world message)
```

2.2 工作流程

（2）生成阶段：运行CMake命令

- CMake会读取CMakeLists.txt文件，并且检查系统环境，包括操作系统类型、编译器类型和版本等信息。（如，Linux可能检测到gcc编译器，Windows则可能检测到Visual C++编译器）
- 根据上述条件，CMake会生成适合该平台的构建文件。在Unix-like系统上，通常会生成Makefile文件；在Windows系统上，可能会生成Visual Studio项目文件（.vcxproj等）。

（3）构建阶段：使用生成的构建文件进行编译

- 在Unix-like系统上，可以使用make命令来进行实际的编译和链接操作。
- 在Windows系统上，生成的Visual Studio项目文件可以直接在Visual Studio集成开发环境中打开，然后使用Visual Studio的编译功能来构建项目。

2.3 指令与变量

- cmake_minimum_required指令 - 规定所需Cmake的最低版本。
  - 基本语法：cmake_minimum_required(VERSION <min_version>[...<policy_max>] [FATAL_ERROR])
  - ① <min_version>：项目所需的最低 CMake 版本号。版本号的格式通常为 major.minor.patch，例如3.10.0。
  - ② [...<policy_max>]：与<min_version>结合作为版本号范围。
  - ③ [FATAL_ERROR]：当添加了这个参数后，如果当前系统中的 CMake 版本低于指定的最低版本，CMake 会立即停止配置过程，并输出错误信息。
  - 例：cmake_minimum_required (VERSION 3.10.0...3.12.0 FATAL_ERROR)
- 注意，方括号代表可选参数，尖括号代表必填参数；小写变量名可以替换。

2.3 指令与变量

- project指令 - 提供项目名称，指定项目相关信息。
  - 基本语法：project(<PROJECT - NAME> [LANGUAGES <language - names>...]
    [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]] [HOMEPAGE_URL <url>]
    [DESCRIPTION <project - description>])
  - ① <PROJECT - NAME>：指定项目的名称。
  - ② [LANGUAGES <language - names>...]：用于指定项目所使用的编程语言。如果不指定，CMake 会根据源文件的扩展名自动推断编程语言。
  - ③ [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]：用于指定项目的版本号。
  - ④ [HOMEPAGE_URL <url>]：可以用来指定项目的主页网址。
  - ⑤ [DESCRIPTION <project - description>]：用于提供项目的简单描述信息。
  - 例：project(my_project LANGUAGES C CXX VERSION 3.10.2)

2.3 指令与变量

- add_executable指令 - 定义如何从给定的源文件构建出一个可执行程序。
  - 基本语法：add_executable(<target_name> [WIN32] [MACOSX_BUNDLE] [EXCLUDE_FROM_ALL] [<source>...])

① <target_name>：指定生成的可执行文件目标的名称。

② [WIN32]：在 Windows 平台上构建项目并且希望生成一个带有 Windows 特定属性（如控制台应用程序或窗口应用程序）的可执行文件时添加。

③ [MACOSX_BUNDLE]：在Mac OS 平台上构建项目时，希望将可执行文件打包成 Mac 应用程序包（Bundle）的形式时添加。

④ [EXCLUDE_FROM_ALL]：当添加这个参数时，生成的可执行文件目标将不会被默认构建。

⑤ [<source>...]：可变长度，用于指定构建可执行文件所需的源文件。

  - 例：add_executable(MyApp main.cpp helper.cpp)

2.3 指令与变量

- set指令 - 创建变量
  - 基本语法：set(<variable> <value>... [PARENT_SCOPE])。
  - ① <variable> <value>：变量名和变量值。如果不给变量值，则会销毁变量。如果给多个值，相当于创建了一个列表变量。
  - [PARENT_SCOPE]：如果指定，则变量会在上一级作用域创建。（CMake的作用域概念与C++中变量的作用域相似）
- 使用变量
  - 可以通过 ${<variable_name>} 的语法形式进行引用。
  - 还可以在源文件中通过配置头文件的方式进行使用。

2.3 指令与变量

- 使用变量
  - 还可以在源文件中通过配置头文件的方式进行使用。
  - ① 在CMakeLists.txt中使用configure_file指令创建一个配置头文件。
  - ② 在MyProjectConfig.h.in文件中使用 @<variable_name>@ 的语法形式引用变量。
  - ③ 在源文件中包含这个配置头文件，即可以使用其中定义的变量值。

- 变量名区分大小写并可以包含字符。虽然使用括号和引号参数允许在变量名中包含空格，但以后引用时，必须使用反斜杠来转义空格；因此，建议在变量名中只使用字母、数字字符、减号和下划线。

```
set(MyString1 "Text1")
set([My String2] "Text2")
set("My String 3" "Text3")

message(${MyString1})
message(${My\ String2})
message(${My\ String\ 3})
```

2.3 指令与变量

- message指令 - 打印信息。
  - 基本语法：message([<mode>] "message text" ...)。
  - ① <mode>：用于指定消息的输出模式，可能的取值有STATUS、WARNING、AUTHOR_WARNING、SEND_ERROR、FATAL_ERROR等。
  - ② “message text”：用于指定要输出的消息内容。消息内容可以是一个简单的字符串，也可以包含 CMake 变量，通过${}语法来引用变量的值。
  - 例：message(STATUS "Project source directory: ${PROJECT_SOURCE_DIR}")

```
# CMakeLists.txt
set(MyString "What a nice day!")
message(${MyString})
```

2.3 指令与变量

- configure_file指令 - 将一个输入文件复制到一个输出文件，并进行变量替换。
  - 通常用于将 CMake 中的变量值传递到源文件中，从而让源文件获取构建时的相关信息，如项目版本号、编译选项等。
  - 基本语法：configure_file(<input> <output> [COPYONLY] [ESCAPE_QUOTES] [@ONLY])

① <input>：输入文件的路径。这个文件通常是一个模板文件，其中包含了一些特殊的变量占位符，如 @<variable_name>@。

② <output>：输出文件的路径，即经过变量替换后的最终文件路径。

③ [COPYONLY]：如果指定了这个参数，那么 CMake 只会简单地复制文件，而不会进行任何变量替换操作。

④ [ESCAPE_QUOTES]：用于指定在变量替换过程中是否对引号进行转义。

⑤ [@ONLY]：用于指定只替换格式为@<variable_name>@的变量，忽略其他可能的变量语法。

2.3 指令与变量

```
# ① CMakeLists.txt
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)
set(MY_PROJECT_VERSION "1.0.0")
project(MyProject VERSION ${MY_PROJECT_VERSION})
configure_file(MyProjectConfig.h.in MyProjectConfig.h)
```

```
// ② MyProjectConfig.h.in
#define MY_PROJECT_VERSION_STRING "@MY_PROJECT_VERSION@"
```

```
// ③ main.cpp
#include "MyProjectConfig.h"
#include <iostream>
int main()
{
    std::cout << "Version: " << MY_PROJECT_VERSION_STRING << std::endl;
    return 0;
}
```

2.3 指令与变量

- add_library指令 - 使用指定的一个或多个文件为项目添加一个库。
  - 基本语法：add_library(<name> [<type>] [EXCLUDE_FROM_ALL] <sources>...)
  - ① <name>：指定库的名字。
  - ② [<type>]：指定库的类型，包括STATIC（静态库），SHARED（动态库）和MODULE（模块库）。如果不指定该参数，默认库的类型为STATIC；如果设置了BUILD_SHARED_LIBS变量，则默认为SHARED。
  - ③ [EXCLUDE_FROM_ALL]：当添加这个参数时，生成的库目标将不会被默认构建。
  - ④ <sources>...：可变长度，用于指定构建库所需的源文件。
  - 生成的library名会根据TYPE属性（STATIC或SHARED）转换为name.a或name.lib
  - 例：add_library(MyLibrary STATIC source1.cpp source2.cpp)

2.3 指令与变量

- target_link_libraries指令 - 将库文件与目标文件进行链接。
  - 基本语法：target_link_libraries(<target> <item>...)。（还有其他很多签名）
  - ① <target>：用于指定要链接库的目标。这个目标可以是通过add_executable创建的可执行文件目标，也可以是通过add_library创建的库目标。
  - ② <item>...：可变长度，用于指定要链接的库。可以是库名称（包括自定义的库和系统提供的库）或全路径库文件，后者不建议使用。
  - 例：target_link_libraries(MyExecutable MyLib)

2.3 指令与变量

- target_link_libraries指令 - 将库文件与目标文件进行链接。
  - 例1：一个图形用户界面（GUI）程序可能依赖于图形库（如 Qt 库）和操作系统的窗口管理库（如在 Windows 上的user32.dll）。

```
# 将GUIApp可执行文件与Qt5::Widgets库（假设已经正确配置了 Qt 库）和user32系统库进行链接
add_executable(GUIApp main.cpp)
target_link_libraries(GUIApp Qt5::Widgets user32)
```

  - 例2：高级库HighLevelLib依赖于一个基础库BaseLib.

```
add_library(BaseLib STATIC base_source.cpp)
add_library(HighLevelLib STATIC highlevel_source.cpp)
target_link_libraries(HighLevelLib BaseLib)
```

2.3 指令与变量

- set_target_properties指令 - 更改target属性。
  - 基本语法：set_target_properties(<target> PROPERTIES <property> <value> [<property> <value>]...)
  - ① <target>：用于指定要设置属性的target名称。target必须是在 CMakeLists.txt 文件中通过add_executable或add_library指令定义的目标。
  - ② <property> <value>：一组属性 - 值对。
  - 例：set_target_properties(MyExecutable PROPERTIES RUNTIME_OUTPUT_DIRECTORY ${PROJECT_BINARY_DIR}/bin)将可执行文件MyExecutable的运行时输出目录设置为${PROJECT_BINARY_DIR}/bin。

2.3 指令与变量

- target_include_directories指令 - 指定target的头文件搜索路径。
  - 基本语法：target_include_directories(<target> [SYSTEM] [BEFORE] <INTERFACE|PUBLIC|PRIVATE> [items1...] [<INTERFACE|PUBLIC|PRIVATE> [items2...]...])

① <target>：用于指定要设置头文件搜索路径的target。

② [SYSTEM]：用于指定头文件路径为系统头文件路径。

③ [BEFORE]：用于控制添加的头文件搜索路径的优先级。如果使用了BEFORE，添加的路径会被放在头文件搜索路径列表的前面。

④ <INTERFACE|PUBLIC|PRIVATE>：用于指定头文件路径的作用范围。包括

- PRIVATE 仅用于目标本身的编译，不会传递给链接这个目标的其他目标。
- INTERFACE 作为接口，当其他目标链接这个库时，可以使用这些路径来搜索头文件。
- PUBLIC 包含了PRIVATE和INTERFACE的功能。既可用于本身，也可被其他库使用。

2.3 指令与变量

- target_include_directories指令 - 指定target的头文件搜索路径。
  - 基本语法：target_include_directories(<target> [SYSTEM] [BEFORE] <INTERFACE|PUBLIC|PRIVATE> [items1...] [<INTERFACE|PUBLIC|PRIVATE> [items2...]...])

⑤ <items1...>：可变长度，用于指定头文件搜索路径。

- 例如，有一个数学库MathLib，头文件存放在${PROJECT_SOURCE_DIR}/math_include目录下，可以这样设置：

```
add_library(MathLib STATIC math_sources.cpp)
target_include_directories(MathLib PUBLIC ${PROJECT_SOURCE_DIR}/math_include)
```

当其他目标（如可执行文件或者其他库）链接MathLib时，就能正确找到数学库的头文件来进行编译。

2.3 指令与变量

- file指令 - 对文件和目录进行各种操作。
  - 读取文件内容到变量：file(READ <filename> <variable> [LIMIT <maxbytes>] [OFFSET <startbytes>] [HEX])。

① <filename>：要读取的文件的名称或路径。

② <variable>：用于存储读取内容的变量。

③ [LIMIT <maxbytes>]：用于限制读取的字节数。

④ [OFFSET <startbytes>]：用于指定从文件的哪个字节位置开始读取。

⑤ [HEX]：如果指定这个参数，文件内容将以十六进制形式读取。

  - 例如，file(READ ${PROJECT_SOURCE_DIR}/config.ini CONFIG_CONTENT)，尝试读取${PROJECT_SOURCE_DIR}/config.ini文件的内容到CONFIG_CONTENT变量中。

2.3 指令与变量

- file指令 – 对文件和目录进行各种操作。
  - 复制文件：file(COPY <files>... DESTINATION <dir> [FILE_PERMISSIONS <permissions>...] [DIRECTORY_PERMISSIONS <permissions>...] [NO_SOURCE_PERMISSIONS] [USE_SOURCE_PERMISSIONS])。

① <files>...：可变长度，用于指定要复制的文件。

② <dir>：目标目录的路径，即文件要复制到的位置。

③ [FILE_PERMISSIONS <permissions>...]：用于指定复制后的文件权限。

④ [DIRECTORY_PERMISSIONS <permissions>...]：用于指定复制后的目录权限。

⑤ [NO_SOURCE_PERMISSIONS]和[USE_SOURCE_PERMISSIONS]：控制是否使用源文件权限。

  - 例：file(COPY ${PROJECT_SOURCE_DIR}/src/file1.cpp ${PROJECT_SOURCE_DIR}/src/file2.cpp DESTINATION ${PROJECT_BINARY_DIR}/src)

2.3 指令与变量

- PROJECT_SOURCE_DIR 与 PROJECT_BINARY_DIR
  - 分别用于指定项目源代码根目录和CMake构建时的输出目录。
  - CMake自动生成，可更改。
- CMAKE_CXX_COMPILER 与 CMAKE_C_COMPILER
  - 分别用于指定 C++ 和 C 编译器的路径。
  - 在不同的操作系统环境下，CMake 会依据系统配置自动检测并填充相应的编译器路径。例如，在常见的 Linux 系统中，CMAKE_CXX_COMPILER 通常指向 g++ 编译器的路径，而在 Windows 系统中，若安装了 Visual C++ 编译器，则会指向其对应的可执行文件路径。
  - 在项目构建过程中，CMake 会依据这些变量所指定的编译器来执行源文件的编译操作，确保代码能够正确转换为目标平台可执行的机器码。

2.3 指令与变量

- CMAKE_CXX_FLAGS 与 CMAKE_C_FLAGS
  - 主要用于存储传递给 C++ 和 C 编译器的各类编译选项，例如优化级别、警告处理以及调试信息生成等。
  - 常见的编译选项包括
    ① -O2 或 -O3 ：用于指定优化级别，以提升生成代码的执行效率；
    ② -Wall 用于开启所有警告信息，帮助开发者发现潜在的代码问题；
    ③ -g 用于生成调试信息，以便在调试阶段能够更方便地进行代码调试和追踪。
- CMAKE_SYSTEM_NAME
  - 用于存储当前构建平台的名称，可能的取值包括Linux、Windows、Darwin（Mac OS）等。
  - 可以依据该变量的值，编写条件判断语句，针对不同的平台执行特定的构建配置操作。

2.4 控制结构

- 条件块 - 语法格式
  - 必须以if()开头并以endif()结尾
  - 在开头和结尾之间可以添加任意数量的elseif()，但只能有单独一个的、可选的else()
- 条件表达式的类型
  - 可以使用STREQUAL（字符串相等比较）、LESS（小于）、GREATER（大于）等操作符来比较变量的值。例如，if(${CMAKE_VERSION} GREATER 3.10)用于判断 CMake 版本是否大于 3.10。 也可以使用逻辑操作符组合多个比较条件，如AND、OR、NOT。
  - 可以使用DEFINED关键字来检查变量是否已经定义。例如，if(DEFINED MY_VARIABLE)用于判断MY_VARIABLE是否已经在之前的 CMake 脚本中定义；也可以使用EXISTS关键字来检查文件或目录是否存在。例如，if(EXISTS ${PROJECT_SOURCE_DIR}/config.ini)用于判断config.ini文件是否存在于项目源目录中。

2.4 控制结构

- while循环
  - 必须以while()开头创建并以endwhile()结尾。

```
while(<condition>)
<commands>
endwhile()
```

```
set(COUNT 0)
while(${COUNT} LESS 10)
    message(STATUS "Count: ${COUNT}")
    math(EXPR COUNT "${COUNT}+1")
endwhile()
```

- foreach循环
  - 类似C/C++的for循环风格控制，首尾分别是foreach()和endforeach()。

```
foreach(<loop_var> RANGE <min> <max> [<step>])
<commands>
endforeach()
```

  - <min>和<step>参数变量可选择配置，默认从0开始。min和max都必须是非负整数。
  - foreach()处理列表变量时十分便捷，在CMake中是更加常用和简洁的循环结构块。

2.4 控制结构

- foreach循环
  - foreach() 处理列表变量时，可以在LISTS后添加若干ITEMS一起遍历，也可以不写LISTS，直接写若干ITEMS用于遍历。

```
foreach(<loop_variable> IN [LISTS <lists>] [ITEMS <items>])
<commands>
endforeach()
```

```
set(SOURCE_FILES main.cpp helper.cpp utility.cpp)
foreach(file IN ${SOURCE_FILES})
    message(STATUS "Processing file: ${file}")
endforeach()
```

- 定义指令：包括宏定义指令 macro() 和函数定义指令 function() 。（结合CMake变量作用域相关知识自行学习）

2.5 综合案例

- 模拟一个简单的图形绘制项目，在不同操作系统下有不同的依赖库，并且会根据源文件目录下的多个源文件来构建可执行文件和库文件，同时通过配置文件传递项目版本信息到代码中

```
GraphicsProject/
├── CMakeLists.txt
├── src
│   ├── main.cpp
│   ├── draw_functions.cpp
│   ├── draw_functions.h
│   └── utils.cpp
├── include
│   └── common_utils.h
└── config
    └── project_config.h.in
```

2.5 综合案例

```
# 指定项目所需的最低CMake版本，若版本过低则报错并停止配置
cmake_minimum_required(VERSION 3.15 FATAL_ERROR)

# 定义项目名称、版本号，并指定使用的编程语言为C++
project(GraphicsProject VERSION 1.0 LANGUAGES CXX)

# 定义源文件列表变量，先初始化为空
set(ALL_SOURCE_FILES "")
set(SOURCE_FILES_FOR_EXECUTABLE "")
set(SOURCE_FILES_FOR_LIBRARY "")

# 使用file命令获取src目录下所有的.cpp源文件，将其存入ALL_SOURCE_FILES变量
file(GLOB_RECURSE ALL_SOURCE_FILES ${PROJECT_SOURCE_DIR}/src/*.cpp)
```

2.5 综合案例

```
# 循环遍历所有源文件，区分出用于构建可执行文件和库文件的源文件
foreach(file ${ALL_SOURCE_FILES})
    if(${file} MATCHES "main.cpp")
        list(APPEND SOURCE_FILES_FOR_EXECUTABLE ${file})
    else()
        list(APPEND SOURCE_FILES_FOR_LIBRARY ${file})
    endif()
endforeach()
```

2.5 综合案例

```
# 根据不同操作系统进行条件判断，添加特定的配置（这里以链接不同系统库为例）
if(${CMAKE_SYSTEM_NAME} STREQUAL "Linux")
    message(STATUS "Building on Linux system")
    set(SYSTEM_LIBRARY_FOR_DRAWING "X11")
    set(SYSTEM_LIBRARY_FOR_UTILS "pthread")
elseif(${CMAKE_SYSTEM_NAME} STREQUAL "Windows")
    message(STATUS "Building on Windows system")
    set(SYSTEM_LIBRARY_FOR_DRAWING "gdi32")
    set(SYSTEM_LIBRARY_FOR_UTILS "ws2_32")
else()
    message(STATUS "Building on other system")
    set(SYSTEM_LIBRARY_FOR_DRAWING "")
    set(SYSTEM_LIBRARY_FOR_UTILS "")
endif()
```

2.5 综合案例

```
# 创建一个名为DrawLib的静态库，使用相应的源文件构建
add_library(DrawLib STATIC ${SOURCE_FILES_FOR_LIBRARY})

# 将include目录添加到DrawLib库的头文件搜索路径，便于找到头文件
target_include_directories(DrawLib PUBLIC ${PROJECT_SOURCE_DIR}/include)

# 创建可执行文件GraphicsProject，使用之前定义好的源文件列表构建
add_executable(GraphicsProject ${SOURCE_FILES_FOR_EXECUTABLE})

# 将DrawLib库与GraphicsProject可执行文件进行链接
target_link_libraries(GraphicsProject DrawLib)
```

2.5 综合案例

```
# 根据操作系统链接对应的特定系统库（如果有）到可执行文件
if(SYSTEM_LIBRARY_FOR_DRAWING)
    target_link_libraries(GraphicsProject ${SYSTEM_LIBRARY_FOR_DRAWING})
endif()
if(SYSTEM_LIBRARY_FOR_UTILS)
    target_link_libraries(GraphicsProject ${SYSTEM_LIBRARY_FOR_UTILS})
endif()

# 设置可执行文件的输出目录属性
set_target_properties(GraphicsProject PROPERTIES RUNTIME_OUTPUT_DIRECTORY
${PROJECT_BINARY_DIR}/bin)
```

2.5 综合案例

```
# 使用configure_file指令生成配置文件，将项目版本号传递到代码中
configure_file(${PROJECT_SOURCE_DIR}/config/project_config.h.in
${PROJECT_BINARY_DIR}/config/project_config.h)

# 输出项目源文件目录和构建目录的路径信息，方便查看构建相关情况
message(STATUS "Project source directory: ${PROJECT_SOURCE_DIR}")
message(STATUS "Project binary directory: ${PROJECT_BINARY_DIR}")
```

目录

- Makefile的使用
  - 基本概念
  - 常用命令与选项
  - 变量
  - 通配符
  - 伪目标
  - 基本结构
  - 综合案例

3.1 基本概念

- 基本结构：主要由一系列规则构成，格式见右侧

```
target: dependencies
	command
```

① target：目标，可以是需要生成的文件名或自定义的操作名。

② dependencies：生成目标所依赖的文件或其他目标，可多个，也可不填。

③ command：构建目标的命令，编写当前操作系统支持的命令。必须以 Tab 键开头，可多个，也可不填（.all，见后续）。

➢ 如果存在相同的 target ，会合并依赖或报出警告：

```
target: depend1
target: depend2
# 两行会被合并为target1: depend1,depend2
	command2
```

```
target: depend1
	command1
target: depend2 # 多个目标无法合并
	command2 # 仅保留最后的目标
```

- 例：

```
my_target: # 下面的指令没有用到外部文件，因此此处不需要写依赖
	echo "This is my target" # Linux命令，用于输出
```

3.1 基本概念

- 工作原理（如何执行一个规则）：

① 检查目录下是否有和目标重名的文件，如果有，退出执行。

② 检查否有依赖项。如果有，先确保对应的依赖文件是否存在。

- 如果文件不存在，先在 Makefile 文件中寻找生成该文件的规则。
  - 如果能找到，先执行这些规则（递归），生成依赖文件；
  - 如果不能找到，退出执行。

③ 检查其时间戳是否比目标新，若存在时间戳更新的依赖文件，则执行指令。

3.2 常用命令与选项

- make：选择 Makefile中第一个目标执行。
- make [target]：选择 Makefile 中名为target的目标执行。
  - 命令行输入 make
    - 输出 This is the first target
  - 命令行输入 make second_target
    - 输出 This is the second target

```
# Makefile

first_target:
    echo "This is the first target"
second_target:
    echo "This is the second target"
```

- 注意，本节所有命令均在命令行输入，而非编写在 Makefile 中。
  - 拓展：在文件压缩大作业中我们尝试过在命令行运行exe并传参。然而为什么直接在命令行输入 make 系列命令，计算机也能找到并调用 exe 程序？（考虑环境变量）

3.2 常用命令与选项

- -n：只打印出执行构建任务时将要执行的命令，而不会真正执行。
  - 也被称为 “dry - run”（空运行）。
  - 一般用于检查 Makefile 中的规则是否正确编写。比如，在修改了 Makefile 中的编译规则后，可以先用 make -n 查看即将执行的命令是否符合预期，再实际执行构建。
- -p：会输出 Makefile 中的所有规则，包括默认规则和变量定义等信息。
- -s：在执行构建任务时，它会抑制命令的输出。
  - 指令被执行会默认输出指令内容（类似于回显），此时可以使用 make -s 抑制所有的输出，也可以针对特定不想要输出的指令，在行首添加@字符。

3.2 常用命令与选项

- -f [filename]：指定一个非标准名称的 Makefile 文件。
  - 在一些复杂的项目中，可能会有多个不同名称的构建脚本，使用这个命令就可以指定特定的构建脚本。
  - 例如，对于名为 build.mk 的构建文件，可以使用 make -f build.mk 执行构建规则。
- -C [directory]：用于改变工作目录到指定的目录后再读取 Makefile。
  - 例如，项目的源代码在 src 目录，而 Makefile 在项目根目录，通过 make -C src 可以在 src 目录下执行构建操作。
- -j [n]：用于开启并行构建。
  - n 表示并行任务的数量。例如，make -j4 会尝试同时执行 4 个构建任务，前提是这些任务之间没有依赖关系或者依赖关系已经满足。

3.3 变量

- 定义变量的格式通常是 变量名 =变量值。例如，CC = gcc 定义了一个名为 CC 的变量，其值为 gcc。
- 引用变量时，需要使用 $(变量名) 的格式。（注意，CMake是 ${} 格式）
- 变量赋值方式分为简单赋值和延迟赋值。
  - 简单赋值（:=）：当定义变量时，变量的值就被固定下来。
  - 延迟赋值（=）：变量的值在每次引用时根据当前的其他变量值重新计算。例如，“VAR1 = $(VAR2)”，如果 VAR2 的值在后面被修改，那么 VAR1 的值也会随之改变。

```
CC = g++
CFLAGS = -Wall -g

main.o: main.cpp
	$(CC) $(CFLAGS) -c main.cpp -o main.o # gcc编译相关指令，自行学习
```

3.3 变量

- $@：代表规则中的目标 target。
- $<：代表规则中的第一个依赖文件。
- $^：代表规则中所有的依赖文件。
- $?：代表规则中更新过的文件。
- 例：

```
mylib.so: obj1.o obj2.o obj3.o
        gcc -shared -o $@ $^
```

  - gcc -shared -o $@ $^ 会展开为 gcc -shared -o mylib.so obj1.o obj2.o obj3.o，调用gcc编译程序并传递这些参数，将这些目标文件构建成动态库 mylib.so。

3.4 通配符

- %：匹配任意字符串（包括空串）。在规则定义和函数操作中用于有规律的文件名转换和规则匹配，特别是在编译过程中为多个相似文件定义统一规则。
- *：匹配任意字符串（包括空串）。通常用于快速获取一组符合某种后缀或前缀等简单模式的文件列表，用于变量赋值或作为其他命令的参数。
- ?：匹配任意单个字符。%和*一般可以满足需求，这种通配符相对不常用。
- 通配符*只进行匹配，无法表达转换关系。例如，%.o：%.c 这种规则用于建立目标文件（.o）和源文件（.c）之间基于文件名的一对一匹配关系。如果写成 *.o：*.c ，Makefile将无法正确理解这种意图。因为*只是简单地匹配所有符合某种模式的文件，没有基于文件名部分相同的目标 - 依赖匹配机制。

3.5 伪目标

- （3.1节）工作原理（如何执行一个规则）：

  ① 检查目录下是否有和目标重名的文件，如果有，退出执行。 …

- Q1：如果一次执行生成后，想重新执行，需要手动删除产生的文件？

  - A1：可以编写规则，专门用于删除产生的文件。运行规则自动删除。

```
clean: # 只需要在命令行 make clean ，就可以删除相关文件
    rm -rf build/ # Linux命令，删除 build 目录及其内部所有文件
    rm -f *.o myapp # Linux命令，删除 myapp 可执行文件以及所有.o文件
# make存在一个内嵌隐含变量“RM = rm  -f”。
# 因此在书写clean规则的命令时可以使用变量$(RM)来代替rm。
```

- Q2：如果当前目录下有一个名为 clean 的文件，如何执行 clean ？

  - A2：可以使用伪目标（.PHONY）绕过 make 的文件重名检测。

3.5 伪目标

- 伪目标（phony targets）并不是实际要生成的文件名，而是用来执行一些命令或作为依赖关系的标签。设置了伪目标后， make 不会将对应的目标视作将要生成的文件，从而不进行文件是否存在的检测。
- 使用 .PHONY 关键字声明伪目标：.PHONY: 伪目标1 伪目标2...。
- 例：

```
.PHONY: cleanall cleanobj cleandiff

cleanall: cleanobj cleandiff
    rm program
cleanobj:
    rm *.o
cleandiff :
    rm *.diff
```

3.5 伪目标

- 通常会按照下面的说明进行相关的定义，并使用伪目标进行声明：

1. clean：通常用于清理项目构建过程中生成的文件。
   - 如目标文件（.o文件）、可执行文件、中间文件等。
2. all：通常作为总目标。
   - 依赖于其他实际的目标或者伪目标，用于构建整个项目。

```
all: myapp mylib # make all 会尝试生成所有依赖，即执行对应的命令
myapp: main.o util.o
    gcc -o myapp main.o util.o
mylib: file1.o file2.o
    ar -rcs mylib.a file1.o file2.o
.PHONY: all
```

3. test：用于执行项目的测试任务。

3.6 基本结构

- 循环结构 - foreach 函数
  - 基本语法：$(foreach var, list, text)，其中var是循环变量，list是值的列表，text是要对每个var的值执行的文本内容，通常包含对var的引用。
  - 例如，假设要将一组.c文件编译为.o文件，并且这些.c文件的文件名存储在变量SOURCE_FILES中，可以这样使用foreach函数：

```
SOURCE_FILES = file1.c file2.c file3.c
OBJECT_FILES = $(foreach src,$(SOURCE_FILES),$(patsubst %.c,%.o,$(src)))

all: $(OBJECT_FILES)
    gcc -o myapp $(OBJECT_FILES)
```

  - src会依次取SOURCE_FILES中的每个值（即每个.c文件名），然后$(patsubst %.c,%.o,$(src))会将每个.c文件名转换为对应的.o文件名，最终OBJECT_FILES变量就包含了所有转换后的.o文件名。

3.6 基本结构

- 条件选择结构 - ifeq 和 ifneq
  - 基本语法：ifeq (arg1,arg2)、ifneq (arg1,arg2)，跟着在条件满足时执行的命令块。
  - 例如，假设有一个变量BUILD_TYPE用于指定构建类型是debug还是release，可以根据不同的构建类型设置不同的编译选项：

```
CFLAGS = ...
BUILD_TYPE = ...

ifeq ($(BUILD_TYPE),debug)
    CFLAGS += -g
else
    CFLAGS += -O2
endif
```

3.7 综合案例

```
CC = gcc
SOURCE_FILES = main.c util.c math.c

# 根据源文件列表生成对应的目标文件列表
OBJECT_FILES = $(foreach src,$(SOURCE_FILES),$(patsubst %.c,%.o,$(src)))

# 定义构建类型变量
BUILD_TYPE = debug

# 根据不同构建类型设置不同的编译选项
ifeq ($(BUILD_TYPE),debug)
    CFLAGS += -g -Wall
else
    CFLAGS += -O2
endif

all: myapp
```

3.7 综合案例

```
# 编译规则，将所有目标文件链接成可执行文件
myapp: $(OBJECT_FILES)
    $(CC) $(CFLAGS) -o $@ $^

# 模式规则，用于将.c文件编译为.o文件
%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
    rm -f $(OBJECT_FILES) myapp

# 伪目标test，用于执行简单的测试
.PHONY: test
test: myapp
   ./myapp
```

总结

- 跨平台编译概述（了解）
- Cmake编译原理（了解）
- Makefile的使用（了解）