# Make 与 CMake 构建工具技术报告

课程：开源技术与应用

姓名：李富麟

学号：2023302111204

专业：软件工程

学院：计算机学院

May 21, 2025

# 目录

# 1 引言

软件开发不仅局限于代码编写，其关键环节在于将所有源代码构建为可用的软件制品。此构建过程可手动执行，然当项目规模扩展时，手动构建将面临诸多挑战，如效率低下及易出错。

Make 与 CMake 是软件开发领域中广泛应用的构建自动化工具。在具有一定规模的 C/C++ 项目中，源文件数量庞大，编译与链接过程错综复杂，手动管理此类过程不仅耗时且极易引入错误。Make 工具借助于 Makefile 文件来定义项目的构建规则，进而实现编译与链接过程的自动化。然而，当项目需在不同平台或采用不同编译器进行构建时，Makefile 在可移植性方面的局限性便显现出来。CMake 应运而生，作为一种跨平台的构建系统生成工具，它通过解析 CMakeLists.txt 文件，能够生成适用于特定平台和编译器的 Makefile（或其他构建系统的项目文件），从而显著简化了跨平台项目的构建与管理流程。

学习 Make 与 CMake 的核心目的在于掌握项目自动化构建的基础方法，深入理解构建过程中的依赖关系管理机制，并培养为不同规模与需求的项目甄选适宜构建工具的能力，以期提升开发效率与项目可维护性。本报告旨在阐述 Make 与 CMake 的基础概念、核心原理及常用命令，并通过具体示例展示其在实际项目中的应用。

# 2 基础概念与原理

## 2.1 Make 与 Makefile

- Make 是一款经典的构建自动化工具，它依据 Makefile 文件中定义的规则来执行项目构建任务。
- Makefile 是一个文本格式的配置文件，其内部包含一系列规则，用以指导 Make 如何编译和链接程序。每条规则通常由以下三个核心部分组成：
  - 目标（Target）：指明构建过程期望生成的最终产物，例如可执行文件或库文件。
  - 依赖（Prerequisites/Dependencies）：列出生成该目标所必需的源文件或其他依赖目标。
  - 命令（Commands）：详细说明了为达成目标而需要顺序执行的 Shell 命令序列。Makefile 的核心机制在于精确定义文件间的依赖关系。当执行 make 命令时，Make 会检查依赖文件的时间戳。若依赖文件较目标文件更新，或目标文件不存在，Make 便会执行规则中定义的命令以重新构建目标。
- 变量（Variables）：Makefile 支持用户自定义变量，用以存储常用字符串（如编译器名称、编译选项等），便于统一修改和集中管理，增强了 Makefile 的灵活性与可维护性。
- 伪目标（Phony Targets）：例如 clean、all 等，此类目标并非指代实际的文件名，而是用于执行预定义的特定命令序列，如清理构建产物或构建所有目标。

## 2.2 CMake 与 CMakeLists.txt

- CMake 是一款先进的跨平台构建系统生成工具。它通过解析 CMakeLists.txt 配置文件，能够为多种目标平台和编译器生成相应的本地构建脚本（如 Makefile 或其他构建

系统的项目文件），进而极大简化了跨平台项目的构建与管理。这意味着 CMake 本身并不直接执行编译任务，而是扮演着上层构建系统生成器的角色。相较而言，Make 的直接输出是已编译的二进制文件，这些文件可在目标计算机上直接运行。

- CMakeLists.txt：此为 CMake 的核心配置文件，采用 CMake 特有的脚本语言编写。该文件详细定义了项目的源文件列表、库依赖关系、编译选项、以及可执行文件或库的生成规则等关键信息。
- CMake 的核心优势在于其卓越的跨平台性。开发者仅需维护一份 CMakeLists.txt 文件，CMake 便能在各种操作系统和编译器环境下生成与之适配的本地构建文件，从而高效实现” 一次编写，多处构建” 的理想目标。
- 源码外构建（Out-of-source build）：CMake 极力推荐将构建过程中生成的中间文件及最终产物与源代码目录分离。具体做法是在源代码目录之外创建一个独立的构建目录（例如 build/）。此举措的主要优点在于能够保持源代码目录的整洁性，并极大地方便了对构建产物的清理工作。
- 模块化与库管理：CMake 提供了强大的模块化支持以及对外部库的查找与链接功能（例如通过 find_package 命令），这使得管理复杂项目及其第三方依赖关系变得更为高效与便捷。
- 通常，cmake 命令用于配置项目并生成构建系统所需的本地构建文件，随后调用 make（或所选构建工具对应的命令）来执行实际的编译和链接操作。

# 3 常用命令与功能示例

## 3.1 Make 示例

假设存在一个基础的 C++ 项目，其结构包含源文件 main.cpp、helper.cpp 以及头文件 helper.h。

```
#ifndef HELPER_H
#define HELPER_H

void print_message();

#endif
```

Listing 1: helper.h

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

void print_message() {
    std::cout << "Hello from helper function!" << std::endl;
}
```

Listing 2: helper.cpp

```cpp
#include "helper.h"
#include <iostream>

int main() {
    print_message();
    return 0;
}
```

Listing 3: main.cpp

一个基础的 Makefile 如下所示：

```makefile
# 定义编译器
CXX = g++
# 定义编译选项
CXXFLAGS = -Wall -std=c++11
# 定义链接选项
LDFLAGS =
# 定义可执行文件名
TARGET = my_program

# 定义源文件和目标文件
SRCS = main.cpp helper.cpp
OBJS = $(SRCS:.cpp=.o) # 将 .cpp 后缀替换为 .o

# 默认目标，通常是第一个目标
all: $(TARGET)

# 链接规则：如何生成最终的可执行文件
$(TARGET): $(OBJS)
    $(CXX) $(LDFLAGS) -o $(TARGET) $(OBJS)

# 编译规则：如何从 .cpp 文件生成 .o 文件
# $< 代表第一个依赖文件（即 .cpp），$@ 代表目标文件（即 .o）
%.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $< -o $@

# 清理规则
clean:
  rm -f $(OBJS) $(TARGET)

# 将 all 和 clean 声明为伪目标
.PHONY: all clean
```

Listing 4: Makefile

Make 常用命令及说明:

- **make**: 执行 Makefile 中定义的第一个目标（通常默认为 all），完成项目的编译与链接。

```
$ make
g++ -Wall -std=c++11 -c main.cpp -o main.o
g++ -Wall -std=c++11 -c helper.cpp -o helper.o
g++  -o my_program main.o helper.o

```

- **make clean**: 执行 clean 伪目标中定义的命令，用以清除构建过程中生成的目标文件和可执行文件。

```
$ make clean
rm -f main.o helper.o my_program

```

- **make <target_name>**: 执行 Makefile 中名为 <target_name> 的特定目标。

说明：

- `CXX` 与 `CXXFLAGS` 是 Makefile 中常用的预定义变量，分别用于指定 C++ 编译器（如 g++）及其编译选项（如优化级别、警告控制等）。
- `$(SRCS:.cpp=.o)` 是 GNU Make 提供的替换引用语法，用于将变量 `SRCS` 中所有以 `.cpp` 结尾的文件名，批量替换为以 `.o` 结尾的目标文件名。
- `%.o: %.cpp` 是一种模式规则，定义了如何将任意 `.cpp` 文件编译为对应的 `.o` 文件。这种写法具有通用性，能大幅简化重复规则的编写，提升构建效率。
- `.PHONY` 用于声明伪目标，表示这些目标并不对应实际文件。这样即使当前目录下存在同名文件，也不会影响对应命令的执行，避免依赖检查误判导致跳过构建步骤。

## 3.2 CMake 示例

针对前述的 C++ 项目，一个基础的 CMakeLists.txt 配置文件示例如下：

```
# CMake 最低版本要求
cmake_minimum_required(VERSION 3.10)

# 项目名称
project(MyProject)

# 设置 C++ 标准
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# 添加可执行文件目标，并指定源文件
add_executable(my_program main.cpp helper.cpp)

# 如果 helper.h 在特定 include 目录下，可以添加：
# target_include_directories(my_program PUBLIC include)
```

Listing 5: CMakeLists.txt

CMake 常用构建流程（推荐采用源码外构建方式）：

1. 在项目根目录下创建一个构建目录（例如 build）并进入该目录：

```
mkdir build
cd build

```

2. 运行 cmake 命令，指向包含 CMakeLists.txt 的源码目录（在此例中是上一级目录..），生成构建系统文件（例如 Makefile）：

```
$ cmake ..
-- The CXX compiler identification is AppleClang 15.0.0.15000309
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /Applications/Xcode.app/
Contents/Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/c++
 - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done (0.3s)
-- Generating done (0.0s)
-- Build files have been written to: /path/to/your/project/build

```

3. 运行实际的构建命令（例如 make）：

```
$ make
[ 33%] Building CXX object CMakeFiles/my_program.dir/main.cpp.o
[ 66%] Building CXX object CMakeFiles/my_program.dir/helper.cpp.o
[100%] Linking CXX executable my_program
[100\%] Built target my\_program

```

编译成功后，可执行文件 my_program 将生成于 build 目录内。

4. 若 CMake 生成的是 Makefile，则可运行 make clean 命令来清理构建产物：

```
$ make clean
[100%] Deleting CMakeFiles/my_program.dir/main.cpp.o CMakeFiles/
my_program.dir/helper.cpp.o CMakeFiles/my_program.dir/
cmake_install.cmake CMakeFiles/my_program.dir/Makefile CMakeFiles/
my_program.dir/cmake_pch.h.gch

```

说明:

- cmake_minimum_required(VERSION 3.10)：此命令用以声明项目构建所需的 CMake 最低版本号。若当前 CMake 版本低于此设定，将会报错并终止配置过程。
- project(MyProject)：此命令用于定义项目的名称。执行后，CMake 会自动设置一系列与项目相关的内置变量，例如 PROJECT_NAME（值为 MyProject）、PROJECT_SOURCE_DIR 等。
- set(CMAKE_CXX_STANDARD 11)：通过此命令设置 C++ 语言标准为 C++11。相应的，还常配合 set(CMAKE_CXX_STANDARD_REQUIRED True) 以确保该标准被严格执行，并在编译器不支持时报错。

- add_executable(my_program main.cpp helper.cpp)：此命令指示 CMake 创建一个名为 my_program 的可执行文件目标，该目标由源文件 main.cpp 和 helper.cpp 编译链接而成。CMake 会自动处理这些源文件之间的依赖关系以及中间目标文件（.o 文件）的生成与管理。

- 优势：与功能等效的 Makefile 相比，CMakeLists.txt 通常更为简洁明了，特别是在处理涉及多目录结构、多库依赖以及跨平台编译等复杂场景时，CMake 的优势尤为突出。例如，CMake 能够自动检测当前构建环境中的编译器类型、操作系统特性，并据此生成相应的、平台优化的构建指令。

# 4 实际案例

## 4.1 Make 在中小型项目中的应用

在个人开发者或小型团队所进行的中小型 C/C++ 项目中，若项目结构相对简单，且主要在单一目标平台（例如 Linux）上进行开发与构建，采用 Make 通常能够满足需求并保持较高的构建效率。例如，对于一个包含若干源文件并需要自定义编译选项的命令行工具，其 Makefile 的典型结构可能如下所示：

```
CC = gcc
CFLAGS = -Wall -O2 -g
LDFLAGS =

SRCS = main.c utils.c network.c
OBJS = $(SRCS:.c=.o)
TARGET = my_tool

all: $(TARGET)

$(TARGET): $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $(OBJS)

%.o: %.c
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    rm -f $(OBJS) $(TARGET)

.PHONY: all clean
```

Listing 6: Makefile

**适用性分析与核心优势**：在上述场景中采用 Make 具有显著优势。对于熟悉其语法的开发者而言，直接利用 Make 和 Makefile 进行项目配置，过程直观且迅速。Makefile 明确定义了文件间的依赖关系，使得 make 命令能够智能判断哪些文件需要重新编译，从而避免了不必要的重复编译，有效节省了构建时间。同时，通过使用变量（如 CFLAGS、SRCS），对编译选项的调整或源文件的增删操作变得更为便捷和集中。Make 的主要贡献在于实现了编译与链接过程的自动化，替代了以往手动逐个编译源文件并链接目标文件的繁琐且易错的操作。通过预设的 clean 目标，可以方便地一键清除所有构建产物，保持项目目录的

整洁。此外，Makefile 为项目定义了一套标准的构建流程，便于团队成员共享和遵循，确保了构建的一致性。

## 4.2 CMake 在大型项目中的应用

当项目规模扩大，需求复杂度提升，例如需要支持多种操作系统（如 Windows、macOS、Linux）、兼容多种编译器（如 GCC、Clang、MSVC），或者项目结构复杂、包含多个子模块、并依赖众多外部库时，CMake 的优势便得以充分体现。

以一个典型的跨平台 GUI 应用程序为例，该程序若选用 Qt 作为其图形库，其顶层 CMakeLists.txt 文件可能包含如下核心配置：

```
cmake_minimum_required(VERSION 3.15)
project(MyCrossPlatformApp LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# 自动处理 Qt moc, uic, rcc (Meta-Object Compiler, User Interface Compiler, Resource Compiler)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTORCC ON)

# 查找 Qt6 组件
find_package(Qt6 COMPONENTS Core Gui Widgets REQUIRED)

# 添加源文件
set(APP_SOURCES
    src/main.cpp
    src/mainwindow.cpp
    src/utils.cpp
)

# 添加头文件目录
include_directories(include)

# 创建可执行文件
add_executable(${PROJECT_NAME} ${APP_SOURCES} src/mainwindow.ui resources.qrc)

# 链接 Qt 库
target_link_libraries(${PROJECT_NAME} PRIVATE Qt6::Core Qt6::Gui Qt6::Widgets)

# 安装规则（可选）
install(TARGETS ${PROJECT_NAME}
    RUNTIME DESTINATION bin
    LIBRARY DESTINATION lib
    ARCHIVE DESTINATION lib/static
)
```

Listing 7: CMakeLists.txt

**适用性分析与核心优势**：在此类复杂项目中，CMake 的选用带来了多方面的益处。首先，开发者仅需编写并维护一份 CMakeLists.txt 文件，即可在 Windows 平台（生成 Visual Studio 项目）、Linux 平台（生成 Makefile）以及 macOS 平台（生成 Xcode 项目或 Makefile）等多种环境下完成项目的构建，有效解决了跨平台构建这一核心难题。其次，CMake 通过 find_package 等命令极大地简化了外部库（如 Qt）的查找、配置与集成过程，并能自动处理 Qt 框架特有的元对象编译、用户界面编译和资源编译等预处理步骤（通过 CMAKE_AUTOMOC 等变量）。对于包含多个子目录和独立模块的大型项目，CMake 通过 add_subdirectory() 命令提供了层次化的项目组织与管理能力，增强了项目的结构清晰度和可维护性。总体而言，CMake 通过高级抽象指令屏蔽了底层编译链接的诸多细节，使得配置文件更易读写，同时提供了对复杂依赖关系的自动化管理，并支持生成多种主流 IDE 的项目文件和后端构建工具的脚本，赋予了开发团队极大的灵活性。

# 5 总结与体会

通过本次学习和实践，我对 Make 和 CMake 这两种构建工具有了更深入的理解。

## 5.1 掌握的技能

1. Makefile 编写基础：深入理解了 Makefile 的基本构成要素，包括目标（Target）、依赖（Dependencies）与命令（Commands）的定义与作用，并掌握了利用变量、模式规则（Pattern Rules）及伪目标（Phony Targets）来组织和优化构建逻辑的方法。现已具备为中小型 C/C++ 项目编写功能完备且结构合理的 Makefile 的能力。

2. CMakeLists.txt 编写入门：初步掌握了 CMakeLists.txt 的核心语法及常用指令，如 project、add_executable、target_link_libraries、set 等。深刻认识到 CMake 作为一种高级构建系统生成器的关键角色，并理解其在实现跨平台构建方面所展现的显著优势。

3. 依赖关系管理：认识到无论是 Make 还是 CMake，其核心任务之一均在于精确描述项目内各文件间的依赖关系，以确保仅当依赖变更时才重新编译相关文件，从而显著提高构建效率。Make 主要通过显式声明依赖来实现，而 CMake 则在更高抽象层次上处理依赖关系，并能更有效地管理复杂的外部库依赖。

4. 源码外构建（Out-of-source Build）实践：学习并实际操作了 CMake 所推荐的源码外构建方法。此方法有助于保持源代码目录的纯净与整洁，并使得构建产物的管理与清理更为便捷高效。

5. 构建工具选择策略：现已能够基于项目规模、技术复杂度、跨平台需求、团队成员对工具的熟悉程度等多维度因素，进行初步的权衡与判断，以确定在特定场景下选用 Make 或 CMake 何者更为适宜。一般而言，对于结构简单、主要面向特定平台的项目，Make 以其轻量级和直接性为佳；而对于结构复杂、有强烈跨平台需求的项目，CMake 则能展现出更强的适应性和管理优势。

## 5.2 困难与挑战

1. Makefile 语法细节的掌握：Makefile 的语法规范，尤其是对 Tab 字符作为命令前导的强制性要求，以及其内部函数、条件语句等高级特性，在初学阶段构成了理解和应用

的难点，且相关错误往往不易调试。

2. CMake 的学习曲线：尽管 CMake 旨在简化跨平台构建的复杂性，但其自成体系的脚本语言、数量繁多的命令及内置变量，客观上形成了一定的学习门槛。要达到熟练运用并编写出结构优雅、执行高效的 CMakeLists.txt 文件，尚需投入更多的时间进行学习与实践经验的积累。

3. 构建问题的调试与排错：在构建过程中遭遇失败时，Make 和 CMake 所提供的错误信息有时可能不够明确具体，对问题的精确定位往往需要开发者对构建工具的内部工作机制及整个构建流程有较为深入的理解。

4. 对不同构建系统生成器的理解：CMake 能够生成适配多种后端构建系统的项目文件（例如 Makefiles、Ninja 构建脚本、Visual Studio 解决方案等）。深入理解这些不同生成器之间的特性差异、适用场景以及如何根据项目需求选择最合适的生成器，是未来需要进一步学习和探索的方面。

5. 大型复杂项目的 CMake 组织策略：对于结构异常庞大且高度模块化的项目，如何设计一套清晰、可维护的 CMakeLists.txt 文件组织结构，以及如何高效管理各模块间的复杂依赖关系和接口定义，无疑是一项具有挑战性的任务。

## 5.3 未来学习与应用展望

熟练掌握 Make 与 CMake 对于未来的软件开发实践具有至关重要的意义。无论是在参与各类开源项目、完成课程设计任务，抑或是未来的职业发展道路上，自动化构建均已成为一项不可或缺的基础技能。深入理解构建工具的核心原理，将使开发者能够更加从容地应对不同类型项目的编译与构建需求，从而显著提升开发效率，并有效减少不必要的重复性劳动。展望未来，计划将投入更多精力深入研习 CMake 的高级特性，包括但不限于其模块化机制、利用如 FetchContent 等方式集成外部库的策略、通过 CTest 框架实现自动化测试的支持，以及运用 CPack 工具进行项目打包与分发等。此举旨在全面提升应对复杂项目构建挑战的能力。

# 6 附录

## 6.1 Make 常用变量

**说明：**

- `$@`：规则中的目标文件名。
- `$<`：规则中的第一个依赖文件名。
- `$^`：规则中的所有依赖文件名列表，以空格分隔，不包含重复项。
- `$+`：与 `$^` 类似，但保留重复依赖。
- `$?`：比目标文件更新的所有依赖文件列表，以空格分隔。
- `$(CC)`：C 编译器名称，默认值为 `cc`。

- `$(CXX)`：C++ 编译器名称，默认值为 g++。
- `$(CFLAGS)`：传递给 C 编译器的编译选项。
- `$(CXXFLAGS)`：传递给 C++ 编译器的编译选项。
- `$(LDFLAGS)`：传递给链接器的选项。
- `$(RM)`：用于删除文件的命令，默认值为 `rm -f`。
- `SRCS`：源文件列表变量，通常用于指定所有 `.c` 或 `.cpp` 文件。
- `OBJS`：目标文件列表变量，通常由源文件替换后缀得到。
- `TARGET`：最终可执行文件的名称。

## 6.2 CMake 常用命令与变量

### 6.2.1 常用命令

- `cmake_minimum_required(VERSION X.Y)`：指定最低 CMake 版本。
- `project(ProjectName [LANGUAGES CXX C])`：定义项目名称和支持的语言。
- `add_executable(target_name source1 source2 ...)`：创建可执行文件。
- `add_library(library_name [STATIC | SHARED | MODULE] source1 source2 ...)`：创建库文件。
- `target_include_directories(target_name [PUBLIC | PRIVATE | INTERFACE] dir1 dir2 ...)`：为目标添加头文件搜索路径。
- `target_link_libraries(target_name [PUBLIC | PRIVATE | INTERFACE] lib1 lib2 ...)`：为目标链接库。
- `set(VARIABLE value [CACHE type "docstring" [FORCE]])`：设置变量。
- `message([STATUS | WARNING | AUTHOR_WARNING | FATAL_ERROR] "message_text")`：输出消息。
- `find_package(PackageName [VERSION] [REQUIRED] [COMPONENTS comp1 comp2])`：查找并加载外部包。
- `add_subdirectory(source_dir [binary_dir])`：添加子目录进行构建。
- `install(TARGETS target_name DESTINATION dir)`：定义安装规则。

### 6.2.2 常用变量

- CMAKE_C_COMPILER：C 编译器路径。
- CMAKE_CXX_COMPILER：C++ 编译器路径。
- CMAKE_C_FLAGS：C 编译选项。
- CMAKE_CXX_FLAGS：C++ 编译选项。
- CMAKE_BUILD_TYPE：构建类型（如 Debug、Release、RelWithDebInfo、MinSizeRel）。通常在配置阶段通过 cmake -DCMAKE_BUILD_TYPE=Release .. 设置。
- PROJECT_SOURCE_DIR：项目顶层源文件目录。
- PROJECT_BINARY_DIR：项目顶层构建目录。
- CMAKE_CURRENT_SOURCE_DIR：当前处理的 CMakeLists.txt 所在源文件目录。
- CMAKE_CURRENT_BINARY_DIR：当前处理的 CMakeLists.txt 对应构建目录。
- EXECUTABLE_OUTPUT_PATH：可执行文件输出路径。
- LIBRARY_OUTPUT_PATH：库文件输出路径。