**make 和 cmake 使用技术报告**

摘要：介绍了两种在软件开发中广泛应用的构建自动化工具：make 和 cmake。首先阐述 make 的基本原理、Makefile的语法结构、常用命令和高级特性，然后将探讨 cmake，解释其核心概念、CMakeLists.txt的编写方法、常用指令、模块化。报告还将对比分析 make 和 cmake 的优缺点、适用场景。通过本报告，将能够进一步理解 make 和 cmake ，提高项目构建的效率和可维护性。

**关键词：**构建自动化工具、软件开发、语法结构、核心概念、可维护性

**1引言**

**1.1 版本控制系统概述**

在软件开发过程中，将源代码转换为可执行程序或库需要涉及编译、链接、资源处理等多个步骤。手动执行这些步骤不仅繁琐、耗时，还容易出错。使用构建自动化工具能够根据预定义的规则自动完成这些任务，确保构建过程的一致性、可靠性和可重复性，从而提高开发效率。常用的构建自动化工具有：

1. make: 一个经典的构建自动化工具，主要用于 Unix 和类 Unix 系统。它通过读取Makefile文件来确定文件之间的依赖关系和构建规则。make 非常强大灵活，但对于大型、跨平台的项目，编写和维护 Makefile 可能会很复杂。
2. cmake: 是一个跨平台开源构建系统生成器。它不直接构建项目，而是读取CMakeLists.txt配置文件，并根据该文件生成特定构建环境的原生构建文件。Cmake可以简化复杂项目的构建配置，并提供跨平台支持。

**2 make工具详解**

**2.1 核心概念**

1. Makefile文件：make工具通过读取名为Makefile或makefile的文件来执行构建任务。此文件包含了项目的构建规则。
2. 目标 (Target)：可以是需要生成的文件名，如可执行文件或目标文件 (.o文件)；也可以是不代表实际文件的“伪目标”（Phony Target），如 clean、all。
3. 依赖 (Dependency/Prerequisite)：目标文件所依赖的文件。如果任何依赖文件比目标文件新，或者目标文件不存在，make就会执行相应的命令来重新生成目标。
4. 命令 (Command/Recipe)：用于生成目标的Shell命令序列。每条命令必须以制表符开头，而不是空格。

**2.2 Makefile 基本语法**

1. 规则 (Rule)：Makefile 的核心是规则，定义了如何生成一个或多个目标文件。基本格式：target ... : dependency ...

command1

command2

...

示例：myapp: main.o utils.o

g++ -o myapp main.o utils.o

1. 变量 (Variable)：Makefile 支持变量，可以简化和参数化构建过程。
2. 注释：以#号开头的行被视为注释。
   1. **make 的工作流程**
3. make 在当前目录查找名为 Makefile 或 makefile 的文件。
4. 如果未指定目标，make 会尝试构建 Makefile 中的第一个目标。
5. 对于要构建的每个目标，make 检查其依赖项：如果依赖项本身也是一个目标，则递归处理该依赖项；make 还会比较目标文件和其所有依赖文件的修改时间，如果任何依赖项比目标新，或目标不存在，则执行与该目标关联的命令。
   1. **make 的局限性**
6. 跨平台性差：Makefile 通常依赖于特定Shell命令和工具链，难以跨操作系统通用。
7. 复杂性：对于大型项目，Makefile 可能会非常庞大和难以维护。
8. 依赖管理：想要自动发现和管理外部依赖比较困难。
9. 语法特性：Tab 缩进要求等比较严格，对新手不友好。

**3. cmake 工具详解**

3.1 cmake 的核心概念

1. CMakeLists.txt文件：cmake的输入文件，包含项目的构建指令、依赖关系、目标定义等。使用cmake语言编写。
2. 命令 (Command)：cmake语言由一系列命令组成，命令不区分大小写，一般使用小写。
3. 变量 (Variable) 与属性 (Property)：cmake使用变量存储配置信息（如编译器路径、编译选项）。变量区分大小写。使用属性，可以更细致地控制目标（Target）的行为。
4. 目标 (Target)：cmake中的目标可以是可执行程序、静态库、动态库等。
5. 生成器 (Generator)：cmake的核心特性，根据CMakeLists.txt和选择的生成器生成特定构建系统的项目文件。也可以指定生成器。
6. 外部构建 (Out-of-Source Build)：将构建产物（Makefile、目标文件、可执行文件等）与源代码分离，通常在源代码树之外创建一个build目录。可以保持源码树干净，方便清理 (只需删除build目录)，支持同时为不同配置（Debug/Release）或不同平台构建。

3.2 选择cmake的好处

1. 跨平台：只需要编写CMakeLists.txt一次，就可以在多种操作系统和编译器上生成原生构建文件。
2. IDE 友好：能生成多种主流 IDE 的项目文件（如 Visual Studio, CLion）。
3. 依赖管理：提供 find\_package() 等机制来查找和使用外部库。
4. 可扩展性：支持模块化，可以编写自定义函数和宏。

3.3 CMakeLists.txt基本语法与常用命令

1. cmake\_minimum\_required(VERSION major.minor[.patch])：指定项目所需的最低 CMake 版本。通常是 `CMakeLists.txt` 的第一条命令。
2. project(ProjectName [VERSION major.minor] [LANGUAGES CXX C])：定义项目名称、可选的版本号和所用编程语言。此命令还会定义一些有用的变量，如 `PROJECT\_NAME`, `PROJECT\_SOURCE\_DIR`, `PROJECT\_BINARY\_DIR`。
3. set(VARIABLE value1 [value2 ...])：设置变量的值。在引用变量使用${VARIABLE}。
4. add\_executable(TargetName source1 [source2 ...])和add\_library(TargetName [STATIC | SHARED | MODULE] source1 [source2 ...])：定义可执行文件或库目标。
5. target\_include\_directories(TargetName [PUBLIC | PRIVATE | INTERFACE] dir1 [dir2 ...])：为目标指定头文件搜索路径。PRIVATE表示仅对此目标内部生效，INTERFACE表示仅对此目标的依赖者生效，PUBLIC表示对此目标内部及其依赖者均生效。
6. target\_link\_libraries(TargetName [PUBLIC | PRIVATE | INTERFACE] lib1 [lib2 ...])：为目标链接库。可以是其他 CMake 目标、库文件路径或库名。
7. target\_compile\_definitions(TargetName [PUBLIC | PRIVATE | INTERFACE] def1 [def2 ...])和target\_compile\_options(TargetName [PUBLIC | PRIVATE | INTERFACE] opt1 [opt2 ...])：为目标设置编译宏定义和编译选项。
8. message([MODE] “message” ...)：打印消息到控制台。

3.4 cmake 的工作流程

1. 配置阶段 (Configure)：
   * 1. 用户在构建目录运行 cmake <source\_directory> [options]。
     2. cmake 读取顶层 CMakeLists.txt 文件。
     3. 执行命令，设置变量，探测编译器、系统特性、依赖库等。
     4. 生成 CMakeCache.txt 文件，存储配置结果，避免重复探测。
2. 生成阶段 (Generate):

根据配置结果和选择的生成器，cmake 在构建目录生成原生的构建文件。

1. 构建阶段 (Build):

用户在构建目录运行原生构建工具，原生构建工具根据 cmake 生成的文件编译和链接项目。

3.5 模块与包管理

1. find\_package(<PackageName> [VERSION] [EXACT] [QUIET] [REQUIRED] [[COMPONENTS] [components...]])：cmake的核心功能之一，用于查找和加载外部库的设置。find\_package会在CMAKE\_MODULE\_PATH和 CMake 安装的模块目录中查找Find<PackageName>.cmake(Find模块) 或 `<PackageName>Config.cmake` / `<lowercase-package-name>-config.cmake` (Config模块)。成功找到后，会定义一些变量，如 `<PackageName>\_FOUND`, `<PackageName>\_INCLUDE\_DIRS`, `<PackageName>\_LIBRARIES`。
2. 编写 Find 模块：对于没有提供Config.cmake文件的库，可以自己编写Find<PackageName>.cmake模块，使用find\_path(),find\_library()等命令查找库的头文件和库文件。
3. FetchContent(CMake 3.11+)：一种在配置阶段下载和构建依赖项的机制，使得依赖管理更加内聚。

`

3.6 cmake 的优势

1. 跨平台构建：一套CMakeLists.txt可以适用于多种平台和编译器。
2. IDE 集成：能生成主流 IDE 的项目文件，方便开发和调试。
3. 依赖发现：find\_package 机制简化了外部库的使用。
4. 可测试性：集成 CTest 测试框架。
5. 可安装性：集成 CPack 打包工具。
6. 现代 C++ 支持：更好地适应现代 C++ 项目的复杂性。
7. 大型项目管理：通过 add\_subdirectory() 和模块化，易于管理大型项目。

**4.** **make 与 cmake 对比**

|  |  |  |
| --- | --- | --- |
| 特性 | Make | CMake |
| 主要功能 | 直接的构建自动化工具 | 构建系统生成器 |
| 输入文件 | Makefile | CMakeLists.txt |
| 跨平台性 | 较差，依赖特定系统和 Shell | 十分优秀 |
| IDE 集成 | 需手动配置 | 可生成多种 IDE 项目文件 |
| 依赖管理 | 手动处理或依赖特定编译器功能 | 内建 find\_package, FetchContent等机制 |
| 学习曲线 | 语法简单，但高级特性和大型项目维护复杂 | 概念较多，有门槛，但掌握后更容易管理复杂项目 |
| 项目规模 | 适合小型、单一平台项目 | 适合中大型、跨平台项目 |
| 构建配置 | 直接在 Makefile 中定义 | 通过 CMake 命令和变量配置，再生成原生构建文件 |
| 源码外构建 | 非原生支持，需手动实现 | 易于实现 |
| 社区与生态 | 成熟，历史悠久 | 非常活跃，适合现代C++项目 |

**5. 使用建议**

5.1 make 使用建议

1. 使用.PHONY声明伪目标。
2. 多用变量提高可维护性，避免硬编码。
3. 使用自动变量 ($@, $<, $^) 简化规则。
4. 对于 C/C++ 项目，利用编译器生成依赖自动处理头文件依赖。
5. 保持 Makefile 简洁。
6. 明确 Tab 缩进，命令前必须是 Tab。

5.2 cmake 使用建议

1. 使用外部构建 (Out-of-Source Builds)：始终在源代码树之外的独立目录（如 build/）中运行cmake和构建命令。
2. 优先使用target\_\*系列命令：如target\_include\_directories, target\_link\_libraries, target\_compile\_options, target\_compile\_definitions。
3. 模块化 CMakeLists.txt：对于大型项目，使用 add\_subdirectory()将项目划分为更小的逻辑单元/模块，每个子目录有自己的CMakeLists.txt；编写自定义函数和宏，封装常用逻辑；使用 install()命令定义安装规则，方便部署。
4. 使用 find\_package的现代方式：如果包提供了导入目标 (Imported Target)，优先链接导入目标，例如使用Threads::Threads而不是${THREADS\_LIBRARIES}。

**6. 总结**

make 是一个经典构建工具，适合小型、单一平台的项目，可以精细控制构建过程。但其有跨平台性差和维护大型项目复杂的缺点。

cmake 作为一个构建系统生成器，通过引入一层抽象，简化了跨平台项目的构建配置和管理。它支持多种原生构建系统和IDE，拥有强大的依赖管理能力，已成为现代C++项目（尤其是中大型和跨平台项目）的常用工具。

选择make还是cmake，取决于项目具体需求、规模、平台目标和团队熟悉度。对于新项目，尤其是需要跨平台或计划长期发展的项目，cmake通常是更优选。理解两者的核心原理和特性，有助于开发者更高效地管理和构建其软件项目。

参考文献

[1]《GNU Make Manual》，2024年，<https://cmake.org/documentation/>

[2] 《GNU make 手册》，2023年2月26日，https://www.gnu.org/software/make/manual/make.html