

智慧星空（上海）工程技术有限公司

## CMake 构建软件开发规范

目 次

前 言 ..... 2

1 目的 ..... 3

2 范围 ..... 3

3 术语和定义 ..... 3

4 概述 ..... 3

5 常用指令 ..... 4

6 模块化构建 ..... 7

7 构建目录输出 ..... 9

8 批量编译参数设计 ..... 10

9 外部脚本依赖设计 ..... 10


10 C/C++构建分离 ..... 11

11 并行编译加速 ..... 11

12 CMake 调试..... 12

13 相关文件标准 ..... 13

14 附录 ..... 13

	CMake 构建软件开发规范	编号：Q/ISTAR-T0143-2024
		版本：A
		密级：三级
		页码：第 2 页 共 13 页

前 言

本标准的编写格式符合GB1.1《标准化工作导则 第1部分：标准的结构和编写》的规定。


本标准主要起草部门：软件工程部。

本标准主要起草人：穆昌根。

本标准由公司标准化主管部门归口。

本标准为首次发布。

序号	修订日期	修订内容	版本号	修订人
1	2024-05-13	新建	A	穆昌根

	CMake 构建软件开发规范	编号: Q/ISTAR-T0143-2024
		版本: A
		密级: 三级
		页码: 第 3 页 共 13 页

## 1 目的

该手册定义了CMake工具构建和打包软件的完整流程，详细说明常用指令/功能的含义和示例用法，用于指导开发人员使用CMake工具自定义软件构建和打包过程。

## 2 范围

该手册描述了公司设备软件系统开发的源码构建和打包流程，所有C/C++软件开发相关的工作均应遵循本文中定义的流程和规范。

## 3 术语和定义

Git	代码版本控制工具
CMake	Cross-platform Make 代码跨平台构建工具
CI	Continous Intergration 持续集成
CD	Continous Delivery/Deployment 持续交付/部署
GCC	GNU Compiler Collection GNU编译器套装
Clang	LLVM编译器前端
IDE	Integrated Development Environment 集成开发环境

## 4 概述

CMake (Cross-platform Make) 是一款开源的跨平台自动化构建工具，开发者通过编写与平台无关的CMakeLists.txt来指定大型C/C++项目的编译流程，根据目标用户的平台生成所需的本地Makefile和工程文件，可以做到“Write Once, Run Everywhere.”。

CMake构建和打包软件的完整流程，分为配置、生成和编译三个阶段，具体内容可参见图1。其中配置阶段生成CMakeCache.txt，生成阶段输出Makefiles或者平台项目文件（如Visual Studio需要的sln文件），最后的构建是完成编译、链接、生成目标文件（可执行文件或库文件）和打包软件。

### 4.1 CMake 构建特性

- a) 跨平台
  - 1) 支持多种操作系统，如 Windows、Linux 和 Mac；
  - 2) 支持多种编译器，如 GCC(gcc/g++)、MSVC、Clang。
- b) 交叉编译
  - 1) 在一个平台生成另一个平台的可执行代码；
  - 2) 举例：在 Linux 平台上编译出在 ARM 架构嵌入式设备上运行的代码。
- c) CI/CD 重要构成
  - 1) CMake 与代码管理平台（如 GitLab）一起构成项目的 CI/CD 流程。

CMake Process Diagram

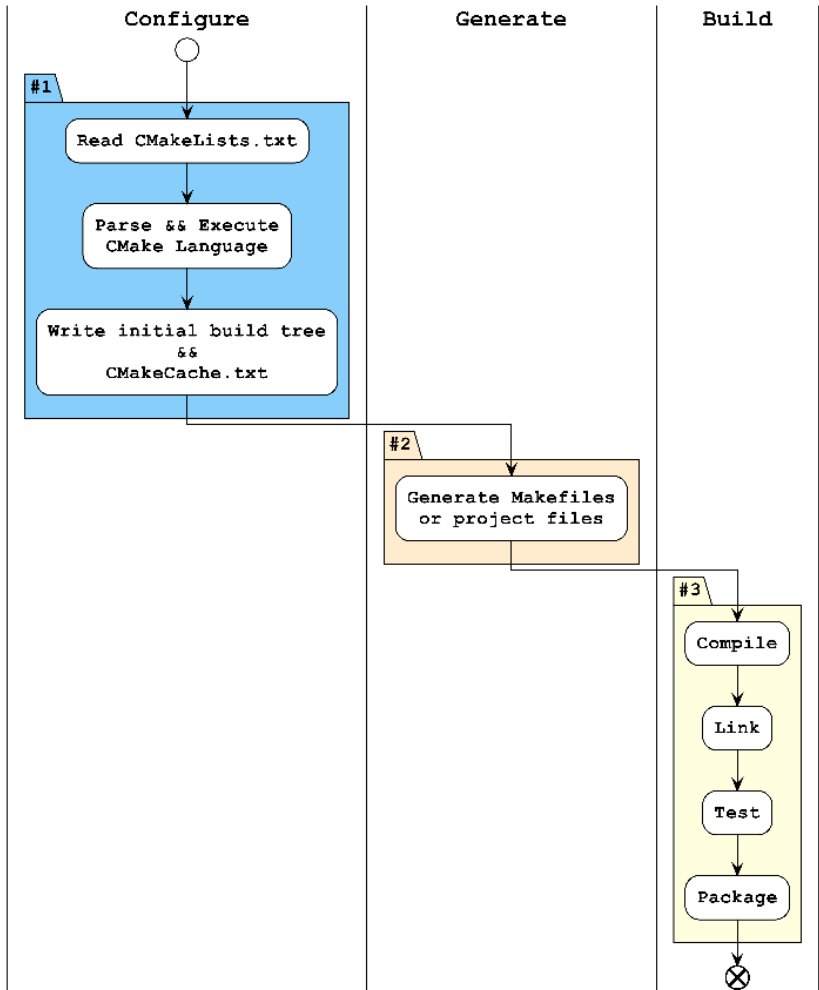


图 1 CMake 构建完整流程示意图

5 常用指令


CMake中常用指令介绍主要包括基本指令、变量与函数和高级特性三部分内容，基本涵盖了中大型C/C++项目中使用频次最多的指令和特性。

5.1 基本指令

下面以一个最简洁的C++软件项目的CMakeLists.txt来说明CMake基本指令的使用和构建软件系统的过程。

代码 1 最简 C++项目 CMakeLists.txt 示例

```
1  # 指定 CMake 最低版本要求
2  cmake_minimum_required(VERSION 3.5)
3
4  # 指定项目名称和编程语言
5  project(MCTRL LANGUAGES CXX)
6
```

	CMake 构建软件开发规范	编号: Q/ISTAR-T0143-2024
		版本: A
		密级: 三级
		页码: 第 5 页 共 13 页

```
7 # 指定生成目标文件 (output)
8 # add_executable: 生成可执行文件
9 # add_library: 生成共享库文件
10 add_executable(output example.cpp)
```

上述三条指令，完成了对一个简单c++项目的构建过程指定：

- 在项目目录下执行 `cmake .`，生成 Makefile 文件；
- 指定编译器根据Makefile指令编译整个项目；
- 编译生成的可执行文件output可以在目标平台上（假定运行环境已满足）正常运行。

## 5.2 变量&函数

上述的例子介绍了单一源文件的CMakeLists.txt的编写，对于多源文件需要用到源文件查找指令，并将查找到的文件保存到指定变量。

和其他计算机编程语言一样，CMake中的变量用于分类命名，简化开发者对于复杂系统的理解。

根据变量生命周期和应用场景的不同，CMake中的变量又可以分为：普通变量、缓存变量、环境变量、列表变量、布尔变量和内部变量。

由于篇幅有限，本文重点介绍CMake中普通变量和内部变量如何与其他指令一起使用完成特定功能。

### 5.2.1 file

指定递归寻找指定路径下所有源文件并保存到指定变量，以便后续引用。CMake中通常用`{var}`表示解引用。

代码 2 搜索源文件保存到指定变量

```
1 # 搜索特定目录下所有 cpp 文件保存在变量 SRC 中
2 file(GLOB_RECURSE SRC "${PROJECT_SOURCE_DIR}/*.cpp")
```

上述指令中，指定了在CMake内部变量PROJECT\_SOURCE\_DIR根目录下递归寻找所有cpp文件，并将搜索到的信息保存到普通变量SRC中。

### 5.2.2 function

CMake中提供了函数function命令的用法，丰富了在CMakeLists.txt中的逻辑处理能力。

CMake中函数function命令使用方法如下：

代码 3 function 指令原型


```
1 function(<name> [arg1 [arg2 ...]])
2     command1(arg1 ...) # 注意本行有缩进
3     command1(arg2 ...)
4     .....
5     return
6 endfunction(<name>) # <name>可不写，如果写须和 function 一致
```

CMake中函数支持return命令退出，同时也支持可变参数，函数内部支持使用内置变量（如ARGVX, ARGC, ARGV）。CMake中function()定义的函数作用域是全局的，即可以在子源码和父源码中使用。

举例如下：

代码 4 function 指令示例

```
1 # function 示例，函数名 test
2 function(test arg1 arg2)
```

	CMake 构建软件开发规范	编号: Q/ISTAR-T0143-2024
		版本: A
		密级: 三级
		页码: 第 6 页 共 13 页

```
3 message( "ARGC: ${ARGC}" )
4 message( "ARGV: ${ARGV}" )
5
6 # 循环打印参数
7 set(i 0)
8 foreach(loop ${ARGV})
9     message( "arg${i}: " ${loop})
10    math(EXPR I "${i}+1" )
11 endforeach()
12 endfunction()
13
14 #调用函数 test
15 test(A B C D E F)
```

### 5.3 高级特性

#### 5.3.1 find\_package

指定项目所需的依赖包，自动配置头文件和动态链接库，包括CMake官方库和指定的第三方库，分别对应Module模式和Config模式。为了方便在项目中引入外部依赖包，CMake官方预定义了部分依赖包Module，存储在path\_to\_your\_cmake/share/cmake-<version>/Modules目录下，每个以Find<LibraryName>.cmake命名的文件对应一个依赖包。

以官方curl库举例如下：

代码 5 find\_package 指令引入官方库示例

```
1 find_package(CURL)
2 add_executable(curltest curltest.cpp)
3 if(CURL_FOUND)
4     target_include_directories(curltest PRIVATE ${CURL_INCLUDE_DIR})
5     target_link_libraries(curltest ${CURL_LIBRARY})
6 else(CURL_FOUND)
7     message(FATAL_ERROR "CURL library not found")
8 endif(CURL_FOUND)
```

如果需要引入非官方库（前提支持CMake编译安装），需要在CMAKE\_PREFIX\_PATH内置变量中指定库的寻找路径。

以引入Qt相关库为例：

代码 6 添加库文件搜索路径

```
1 set(Qt "C:/Qt/Qt5.12.12/5.12.12/msvc2017_64") # Qt 目录
2 list(APPEND CMAKE_PREFIX_PATH ${Qt}) # 添加进 CMAKE_PREFIX_PATH
```

接下来和官方库引用类似，在CMakeLists.txt中使用find\_package。

命令引用已安装的第三方库。


举例如下：

代码 7 find\_package 指令引入三方库示例

```
1 find_package(Qt5 COMPONENTS Core Network Gui REQUIRED)
```

此处不再赘述第三方库查找失败的异常处理逻辑。

#### 5.3.2 target\_include\_directories

	CMake 构建软件开发规范	编号: Q/ISTAR-T0143-2024
		版本: A
		密级: 三级
		页码: 第 7 页 共 13 页

用于指定目标文件生成时依赖的头文件。

5.3.3 target\_compile\_definitions

用于指定目标文件生成时的编译定义，可以是宏定义或CMake表达式，传递给编译器的宏定义，进而目标文件的编译方式。

5.3.4 target\_compile\_options

用于指定目标文件生成时的编译选项。

5.3.5 target\_link\_directories

用于指定目标文件生成链接阶段所需的库文件路径。

5.3.6 target\_link\_libraries

用于指定目标文件生成链接阶段所需的库文件名称。  
举例如下：

代码 8 target\_xx 指令示例

```
1 target_compile_definitions(  
2   ${PROJECT_NAME} PUBLIC -DQS_HAS_JSON)  
3 target_link_directories(${PROJECT_NAME} PUBLIC  
4   "${CMAKE_BINARY_DIR}/DeviceDriver/${<IF:${CONFIG:Debug}>, Debug, Release}>")  
5 target_link_libraries(${PROJECT_NAME} PRIVATE DeviceDriver)  
6 target_compile_options(${PROJECT_NAME} PUBLIC /bigobj)
```

其中PUBLIC、PRIVATE以及INTERFACE关键字用于控制头文件和库文件的传播范围。




5.3.7 add\_custom\_command

提供给开发者自定义命令的指令，通常用于自动生成代码、自定义编译步骤和自定义清理步骤等。  
举例如下：

代码 9 add\_custom\_command 指令示例

```
1 add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD COMMAND ${CMAKE_COMMAND} -E  
2   copy_if_different ${TARGET_FILE:Qt5::Core} ${TARGET_FILE_DIR:${PROJECT_NAME}})
```

上述例子用于在构建目标完成构建后拷贝Qt5::Core动态链接库至目标生成目录，其中运用了CMake的生成器表达式功能，部分参数解释如下：

-  TARGET \${PROJECT\_NAME} # 执行自定义命令的构建目标
-  POST\_BUILD # 执行时机，这里是构建完成后
-  COMMAND # 核心构成，系统命令、脚本或其他构建工具

6 模块化构建



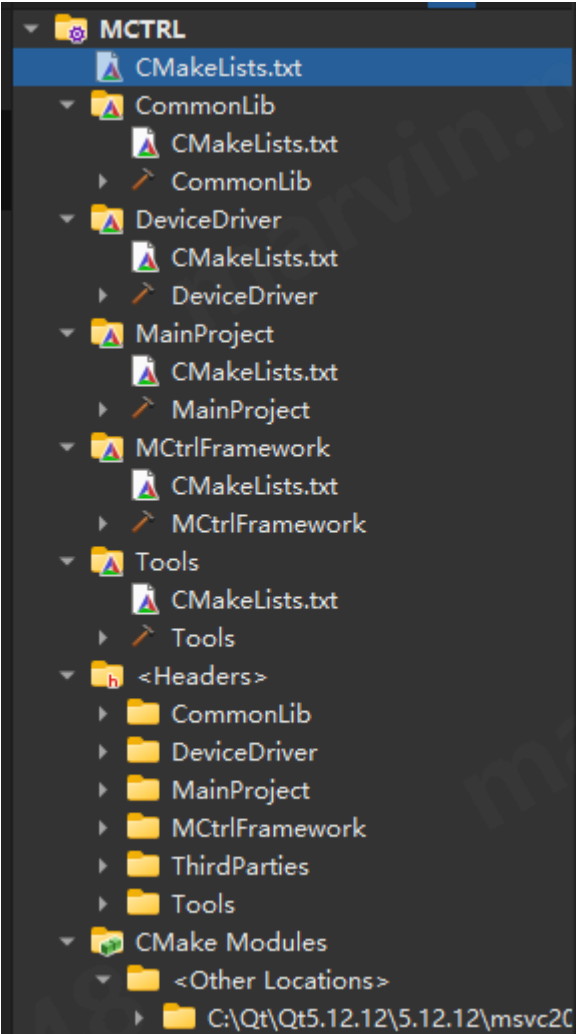


图 2 典型 C/C++工程目录树状拓扑结构

6.1 树状工程目录

如前所述的是单一项目的CMakeLists.txt的编写规则，上图展示了一个典型C/C++工程目录对应的树状拓扑结构。MCTRL工程下囊括了多个项目，每一个项目对应一个自己的CMakeLists.txt。CMake执行时，会从工程的根目录下开始解析CMakeLists.txt（后续称为顶层CMakeLists），以下为MCTRL目录下CMakeLists.txt的内容：

代码 9 顶层 CMakeLists.txt 示例

```
1  cmake_minimum_required(VERSION 3.5)
2
3  project(MCTRL LANGUAGES CXX)
4
5  set(CMAKE_AUTOMOC ON) # moc 编译器打开
6  set(CMAKE_CXX_STANDARD 11)
7  set(CMAKE_CXX_STANDARD_REQUIRED ON)
8  set(CMAKE_INCLUDE_CURRENT_DIR ON)
9
10 set(Qt "C:/Qt/Qt5.12.12/5.12.12/msvc2017_64")
11 list(APPEND CMAKE_PREFIX_PATH ${Qt})
12
```

```
13 add_subdirectory(Tools)
14 add_subdirectory(CommonLib)
15 add_subdirectory(MCtrlFramework)
16 add_subdirectory(DeviceDriver)
17 add_subdirectory(MainProject)
```

可以看出，顶层CMakeLists.txt主要使用add\_subdirectory指令枚举和添加工程下所有需要的项  
目。

6.2 模块化构建优势

- a) 项目构建解耦  
每个项目对应一个CMakeLists.txt，可以视为一个独立的子系统，不同的子系统之间独立构  
建，互不干扰。
- b) 赋予灵活性  
每个项目可以根据项目特性和需求定制化构建，如需要不同的编译选项或者依赖于特定版本  
库。
- c) 增加可拓展  
如果需要添加新的项目，只需要创建相应的CMakeLists.txt即可，无需修改其他项目的配置  
文件，易于拓展。
- d) 抽象复用性  
因为项目的独立性和完整性，使其很容易迁移到其他项目或者系统中被复用。

7 构建目录输出

7.1 结构化设计目的


作为项目组织结构的重要部分，构建目录输出的设计会直接影响开发效率和项目的可维护性。原  
因如下：

- a) 规范的构建目录输出设计能够使项目结构清晰，便于理解和定位代码。一个好的构建目录  
输出设计，可以将源代码文件、头文件、库文件以及其他资源文件按照一定的规则进行组织，  
使得开发者能够快速定位到需要的文件，提高开发效率。类比于Visual Studio和Qt Creator  
的默认文件目录构成即可有体会；
- b) 规范的构建目录输出设计有利于项目的编译和构建。CMake中提供了设置输出目录的指令，可  
以将生成的目标文件（如库文件和可执行文件）输出到指定的目录中，从而避免源代码目录  
的污染，同时也易于管理编译生成的文件；
- c) 规范的构建目录输出设计有利于版本控制和CI。通常在版本控制中，只对源代码进行管理，  
不关心编译生成文件。规范化的输出结构设计，可以极大简化版本管理过程。在CI中，通过  
构建目录输出设计，将编译生成文件输出到CI服务器指定目录下，便于后续的测试和部署操  
作。

举例说明如下：

代码 10 构建目录输出示例

```
1 # 指定编译输出目录
2 set(OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/build)
3 # 设置目标属性
4 set_target_properties(
5 my_target PROPERTIES
```

	CMake 构建软件开发规范	编号: Q/ISTAR-T0143-2024
		版本: A
		密级: 三级
		页码: 第 10 页 共 13 页

```
6  RUNTIME_OUTPUT_DIRECTORY ${OUTPUT_DIRECTORY}/bin
7  LIBRARY_OUTPUT_DIRECTORY  ${OUTPUT_DIRECTORY}/lib
8  ARCHIVE_OUTPUT_DIRECTORY  ${OUTPUT_DIRECTORY}/archive
9  )
```

类比上述设置，也可以设置Release和Debug输出路径。

## 8 批量编译参数设计

### 8.1 常用编译参数

编译参数是编译器在执行编译指令时附带的参数，不仅会影响编译过程和效率，同时也会影响生成文件的功能和性能。

#### a) 性能参数

通过参数来指定不同优化等级，如-O1、-O2和-O3，对应着不同的优化程度，直接影响程序的性能。-finline-functions（内联函数）则可以进一步优化执行文件性能。

#### b) 调试参数

用于生成和调试相关的信息或者文件，如-g(生成调试信息)、-fsanitize=address(地址检查)。

#### c) 项目管理

编译参数同样可以用来设置项目管理方式，如-MD(生成依赖信息)、-fvisibility=hidden(隐藏符号)可以提高构建和链接效率。

举例说明如下：

代码 11 编译参数示例

```
1  # 全局开启 C++ 文件使用-O2 级别优化，对所有目标生效
2  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O2")
3  #只对目标设置编译参数
4  target_compile_options(my_target PRIVATE -std=c++11)
5  #特定目录下设置编译参数，当前目录及其子目录下目标开启警告
6  add_compile_options(-Wall)
```

## 9 外部脚本依赖设计

### 9.1 启动第三方脚本/程序


外部脚本作为大型程序的自动化构建重要组成部分，额外为开发者提供了灵活构建过程定义。

在CMake中，可以使用add\_custom\_command和add\_custom\_target命令来设计外部脚本依赖。

举例说明如下：

代码 12 启动第三方脚本示例

```
1  add_custom_command(
2  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/generated_source.cpp
3  COMMAND python ${CMAKE_CURRENT_SOURCE_DIR}/generate_source.py >
   ${CMAKE_CURRENT_BINARY_DIR}/generated_source.cpp
4  DEPENDS ${CMAKE_CURRENT_SOURCE_DIR}/generate_source.py
5  )
6  add_custom_target(
7  my_target
8  DEPENDS ${CMAKE_CURRENT_BINARY_DIR}/generated_source.cpp
9  )
```

	CMake 构建软件开发规范	编号：Q/ISTAR-T0143-2024
		版本：A
		密级：三级
		页码：第 11 页 共 13 页

上述例子中，创建了一个名为my\_target的自定义目标，它依赖于generated\_source.cpp文件。当开始构建目标时，CMake会先检查依赖generated\_source.cpp是否存在，如果不存在或者它的依赖generate\_source.py发生变化，CMake会自动执行自定义命令，通过运行python脚本生成构建所需的generated\_source.cpp文件。

10 C/C++构建分离

10.1 分离构建必要性

在大型项目中，针对C/C++混合编程的需要，C/C++分离构建可以帮助更好管理代码，提高代码的可读性和可维护性，是一种良好的项目管理和构建风格。

CMake中，主要使用目标属性和编译选项来实现C/C++的分离设计。比如用target\_compile\_feature和set\_target\_properties指令来管理C/C++构建过程。

举例说明如下：

代码 13 C/C++分离构建示例

```
1 # 设置目标使用 C99 标准
2 target_compile_feature(my_target1 PUBLIC c_std_99)
3
4 # 设置目标使用 C++11 标准以及关闭拓展属性
5 target_compile_feature(my_target2 PUBLIC cxx_std_11)
6 set_target_properties(my_target2 PROPERTIES CXX_EXTENSIONS OFF)
```

11 并行编译加速

大型项目由于源文件数量多，引用库庞杂，有必要通过硬件资源加速编译过程，提高开发和调试效率。不同平台或编译器通过CMake开启并行加速方式会有差异。本文仅以Windows平台cl编译器举例说明加速指令使用。

举例说明如下：建立bat脚本。

代码 14 Batch 脚本并行编译加速示例

```
1 md build # 指定构建输出目录
2 cd /d build
3 cmake .. # cmake 执行顶层 CMakeList.txt
4 cmake --build . --config Release -- /p:CL_MP=true /p:CL_MPCount=!mpcount!
```

上述脚本的核心指令是/p:CL\_MP=true /p:CL\_MPCount=!mpcount!开启cl.exe并行编译，并行进程数量为mpcount。如下图所示开启并行编译时CPU资源100%占用，编译过程加速明显。

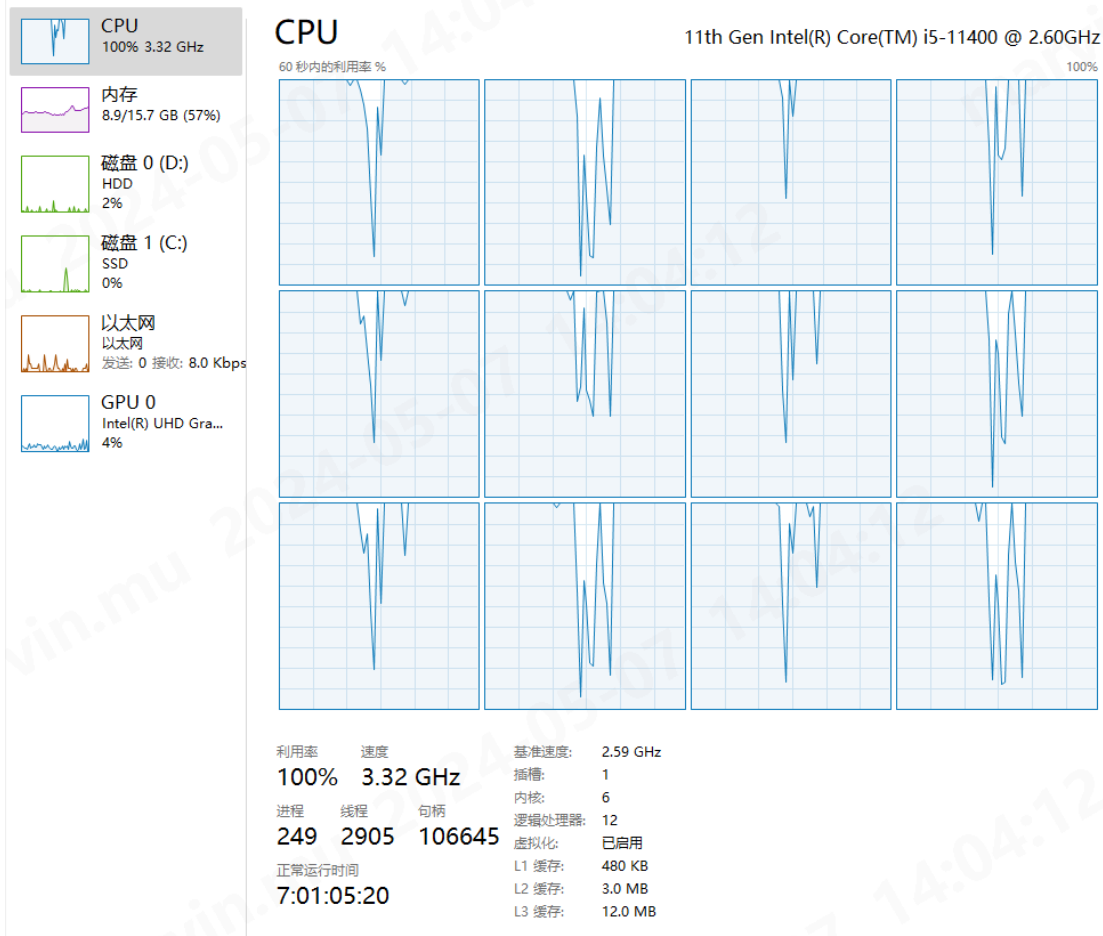


图 3 windows 平台 msvc 并行编译加速效果

12 CMake 调试

CMake中常用调试方法主要有message指令打印和IDE编译日志监控窗口。

12.1 message 指令打印

代码 15 message 指令示例

```
1 # 打印当前 pc 逻辑核心数量
2 message("${CPU_NUMBER_OFLOGICAL_CORES}")
```

12.2 IDE 编译输出窗口日志

```
编译输出
14:09:21: 为项目MCTRL执行步骤 ...
14:09:21: 正在启动 "D:\Program Files\CMake\bin\cmake.exe" --build D:\Marvin\CMake\Mctrl\build --target ALL_BUILD --config Release

用于 .NET Framework 的 Microsoft (R) 生成引擎版本 16.11.2+f32259642
版权所有 (C) Microsoft Corporation。保留所有权利。

Automatic MOC for target CommonLib
CommonLib.vcxproj -> D:\Marvin\CMake\Mctrl\build\CommonLib\Release\CommonLib.lib
Automatic MOC for target DeviceDriver
DeviceDriver.vcxproj -> D:\Marvin\CMake\Mctrl\build\DeviceDriver\Release\DeviceDriver.lib
Automatic MOC for target MCtrlFramework
MCtrlFramework.vcxproj -> D:\Marvin\CMake\Mctrl\build\MCtrlFramework\Release\MCtrlFramework.lib
Automatic MOC for target Tools
Tools.vcxproj -> D:\Marvin\CMake\Mctrl\build\Tools\Release\Tools.lib
Automatic MOC for target MainProject
MainProject.vcxproj -> D:\Marvin\CMake\Mctrl\build\MainProject\Release\MainProject.exe
14:09:23: 进程"D:\Program Files\CMake\bin\cmake.exe"正常退出。
14:09:23: Elapsed time: 00:02.
```

图 4 Qt Creator 构建输出日志窗口

13 相关文件标准

Q/ISTAR-WI-35 《软件开发流程规范》

14 附录

附录A CMake参考文档 <https://cmake.org/cmake/help/latest>

附录B 构建实例参考 Git/IAS/Mctrl (GUI)