CMake---优雅的构建C/C++软件项目实践(1) - 云+社区 - 腾讯云

笔记本:	印象笔记
创建时间:	2021/12/9 15:55
URL:	https://cloud.tencent.com/developer/article/1606950?from=article.detail.1590285

CMake---优雅的构建C/C++软件项目实践(1)

2020-03-31 阅读 1.3K

首先说明的是本篇文章不从cmake的整个语法上去讲述，而是从一个实际项目的构建上入手，去了解如何优雅的去构建一个软件项目，搭建一个 `C/C++` 软件项目基本的依赖组件，最后形成一个构建 `C/C++` 软件项目的模板，方便后面新项目的重复使用。相信对我们日常的软件项目构建都会有很好的收获。废话不都说，开始。

1 我们需要知道的基础

首先熟悉cmake的一些基操，我们就可以信手捏来的、优雅去构建一个项目，避免踩到不必要的坑。涉及到的有：

- cmake的变量作用域?
- cmake中的数据结构?
- 宏函数与函数?
- 如何去构建动静态库和找到这些库?
- 如何去实现支持多平台的项目构建?
- 如何去构建一个应用?
- 如何实现项目的最后install?
- 如何很友好的去展示构建过程的各种级别信息?
- 如何适配cmake-gui，采用友好的ccmake或者cmake-gui实现构建?

这里概括性说明下常用的cmake知识，总的来说cmake的作用就是让我们找到依赖的头文件和库文件，去编译源文件、链接目标文件(静态库也是目标文件的一个集合)，最后生成可执行文件或动/静态库：

- `INCLUDE_DIRECTORIES` 将给定的目录添加到编译器用于搜索包含文件(如头文件)的目录中，相对路径被解释为相对于当前源目录。注意目录仅是被添加到当前CMakeLists文件，作用于当前CMakeLists文件相关的库、可执行文件或者 `子模块` 编译，对于两个不同CMakeLists.cmake并列的作用是无效的。区别于 `TARGET_INCLUDE_DIRECTORIES` ，这个命令的作用只是作用于指定的目标，为指定的目标添加搜索路径。类似的还有 `TARGET_LINK_LIBRARIES` 命令(添加需要链接的库文件目录)。
- `PROJECT_SOURCE_DIR` : 无疑只要是有包含最新PROJECT()命令声明的CMakeLists.txt，则都是相对当该CMakeLists.txt路径。
- `CMAKE_SOURCE_DIR` : 构建整个项目时，可能你依赖的第三方项目，这个变量的值就是最顶层CMakeLists.txt的路径。
- 在 `find_path` 和 `find_library` 以及 `find_package` 时，会搜索一些默认的路径。当我们将一些lib安装在非默认搜索路径时，cmake就没法搜索到了，可设置：
  - SET(CMAKE_INCLUDE_PATH "include_path") // find_path，查找头文件
  - SET(CMAKE_LIBRARY_PATH "lib_path") // find_library，查找库文件
  - SET(CMAKE_MODULE_PATH "module_path") // find_package
- 寻找3rdparty也不一定需要自己去编写FindXX.cmake，也可以直接用include(xxx.cmake)结合 `find_file` 命令实现寻找依赖库，find_file寻找到的结果存放到CACHE变量，示例：

```
# Once done, this will define
#
#  NANOMSG_INCLUDE_DIR - the NANOMSG include directory
#  NANOMSG_LIBRARY_DIR - the SPDLOG library directory
#  NANOMSG_LIBS - link these to use NANOMSG
#
#  SPDLOG_INCLUDE_DIR - the SPDLOG include directory
#  SPDLOG_LIBRARY_DIR - the SPDLOG library directory
#  SPDLG_LIBS - link these to use SPDLOG

MACRO(LOAD_LIBNANOMSG os arch)
```

```
    SET(3RDPARTY_DIR ${PROJECT_SOURCE_DIR}/3rdparty/target/${${os}}_${${arch}})
    MESSAGE(STATUS "3RDPARTY_DIR: ${3RDPARTY_DIR}")
    FIND_FILE(NANOMSG_INCLUDE_DIR include ${3RDPARTY_DIR} NO_DEFAULT_PATH)
    FIND_FILE(NANOMSG_LIBRARY_DIR lib ${3RDPARTY_DIR} NO_DEFAULT_PATH)

    SET(NANOMSG_LIBS
        nanomsg
        pthread
        anl
        PARENT_SCOPE
    )
    IF(NANOMSG_INCLUDE_DIR)
        MESSAGE(STATUS "NANOMSG_LIBS : ${NANOMSG_LIBS}")
    ELSE()
        MESSAGE(FATAL_ERROR "NANOMSG_LIBS not found!")
    ENDIF()
ENDMACRO()
```

- 条件控制切换示例：

```
# set target
if (NOT YOUR_TARGET_OS)
    set(YOUR_TARGET_OS linux)
endif()

if (NOT YOUR_TARGET_ARCH)
    set(YOUR_TARGET_ARCH x86_64)
endif()

if (NOT YOUR_BUILD_TYPE)
    set (YOUR_BUILD_TYPE Release)
endif()

......

if(${YOUR_TARGET_ARCH} MATCHES "(arm*)|(aarch64)")
    ......
elseif(${YOUR_TARGET_ARCH} MATCHES x86*)
    ......
```

- 交叉编译：`CMAKE_TOOLCHAIN_FILE` 变量，

```
MESSAGE(STATUS "Configure Cross Compiler")

IF(NOT TOOLCHAIN_ROOTDIR)
    MESSAGE(STATUS "Cross-Compiler defaut root path: $ENV{HOME}/Softwares/arm-himix200-linux")
    SET(TOOLCHAIN_ROOTDIR "$ENV{HOME}/Softwares/arm-himix200-linux")
ENDIF()

SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_SYSTEM_PROCESSOR arm)

SET(CMAKE_C_COMPILER       ${TOOLCHAIN_ROOTDIR}/bin/arm-himix200-linux-gcc)
SET(CMAKE_CXX_COMPILER     ${TOOLCHAIN_ROOTDIR}/bin/arm-himix200-linux-g++)

# set searching rules for cross-compiler
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)

SET(YOUR_TARGET_OS linux)
SET(YOUR_TARGET_ARCH armv7-a)

SET(CMAKE_CXX_FLAGS "-std=c++11 -march=armv7-a -mfloat-abi=softfp -mfpu=neon-vfpv4 ${CMAKE_CXX_FLAGS}")
```

- `AUX_SOURCE_DIRECTORY` 不会递归包含子目录，仅包含指定的dir目录
- `ADD_SUBDIRECTORY` 子模块的编译，可以将子文件夹中或者指定外部文件夹下CMakeLists.txt执行相关编译工作。

- `ADD_LIBRARY` 编译一个动/静态库或者模块，设定的名字需在整个工程中是独一无二的，而且在整个同一个工程中，跟父子文件夹路径无关，我们便可以通过 `TARGET_LINK_LIBRARIES` 依赖该模块。
- `ADD_DEFINITIONS(-DTEST -DFOO="foo")` 添加 `FOO` 和 `TEST` 宏定义。

2 我们要优雅做到的构建

对于一个较大的软件项目，我们会依赖很多第三方的项目，包括源码依赖或者库依赖，然后完整的构建自己的软件项目，则需要去构建依赖项目或者找到我们所需要库；另外，软件项目会考虑到可移植性，即能够在不同的平台上也能够很好友的去构建项目以及将项目转移到另一个开发环境时能够快速的开始构建。

除了上面所说的，我们还需要考虑我们实际软件项目的架构结构，源码结构，可以让开发人员更清晰的、更快速的了解整个项目。

除此之外，C/C++ 程序员长期以来手动管理依赖，即手动查找、安装依赖，再配置构建工具（如 cmake）使用依赖。cmake 还提供了一系列 find_package 方法帮助简化配置依赖， cmake 还支持多项目/模块管理，如果依赖源码同时被 cmake 管理构建，那么情况会简单很多，这种方式称为源码级依赖管理。随着代码管理工具 git 出现并被广泛使用，git submodule 提供了一种不错的源码级依赖管理办法。

综上，优雅的构建软件项目，我们实现：

- 软件项目源码依赖第三方项目
- 软件项目库依赖第三方项目
- 软件项目结构清晰
- 软件项目构建在转换新环境下快速实现构建
- 软件项目构建过程中的信息友好展示
- 软件项目构建完成后打包发布
- 软件项目支持跨平台构建
- 软件项目支持交叉构建
- git submodule & cmake管理/构建源码级依赖

另外，我们还实现一个可复用的C/C++最小开发框架（这个到后续文章中讲述）：

- 支持日志记录
- 支持任务池/线程池
- 支持常用相关基础操作组件
  - 时间日期操作
  - 文件读写操作
- 支持valgrind内存泄露检查工具
- 支持静态代码检查
- 支持项目文档自动化
- .....

3 优雅的软件项目结构模板

3.1 模板一

一个独立的应用，应用模块之间是相互联系的，彼此难以分开，这样简单的将所有源文件放一起，头文件放一起，这个对于不是很复杂的应用是很快速的去开始构建和源文件修改操作：

```
.
├── 3rdparty
├── cmake
├── include
├── src
├── doc
├── tests
├── benchmarks
├── docker
├── CMakeLists.txt
```

3.2 模板二

源文件与头文件分功能模块存放，这种方式是比较简单，但是如果成为其他项目的3rdparty，则需要在安装上将头文件分离出来，不能很方便的被其他项目直接引用，个人觉得适用于App类项目，而非SDK项目(比如nanomsg这个开源消息中间件库就是将头文件和源文件放一起，但是作为SDK供外部链接就不是很直接、很方便了，需要做install操作之后才可以或者是将头文件搜索范围设置到依赖项目的src级别，且src目录下模块分类很明确)：

```
├── 3rdparty
    ├── submodule # 存放源码依赖
```

```
    ├── target # 存放库依赖
    ├── CMakeLists.txt
    ├── cmake # 存放 find_package cmake文件
├── cmake
├── platforms
│   └── linux
│       └── arm.toolchain.cmake
├── src
    ├── moudle1
        ├── source & include file
    ├── moudle2
        ├── source & include file
    ├── ......
├── doc
├── tests
├── samples
├── benchmarks
├── docker
├── CMakeLists.txt
```

3.3 模板三

该软件项目可以分为很多模块，各个模块可以互相独立，也可以组合在一起，典型的如opencv项目，当然这个也适用于应用项目，但是应用项目的话目录结构太深，开发编辑上稍有不便：

```
├── 3rdparty
├── cmake
├── platforms
│   └── linux
│       └── arm.toolchain.cmake
├── include  该目录只是各功能模块头文件的一个汇总包含
├── modules
    ├── moudle1
        ├── src
        ├── include
    ├── moudle2
    ├── ......
├── doc
├── tests
├── samples
├── benchmarks
├── docker
├── CMakeLists.txt
```

4 优雅的软件项目结构模板CMake实现

这里我们只去实现模板二，其他模板大同小异。如上面模板章节所述，我们

4.1 目录结构确定

```
.
├── 3rdparty                # 第三方库源码依赖和库依赖存放位置
│   ├── CMakeLists.txt      # 第三方库源码依赖编译CMakeLists文件
│   ├── spdlog              # 源码依赖示例项目spdlog(github可搜索)
│   └── target              # 库依赖存放目录
│       ├── linux_armv7-a   # 以平台和架构命名区分
│       │   ├── include     # 头文件存放目录
│       │   └── lib         # 库文件存放目录
│       └── linux_x86-64
│           ├── include
│           └── lib
├── cmake                   # 存放项目相关的cmakem模块文件
│   ├── load_3rdparty.cmake
│   ├── messagecolor.cmake
│   ├── toolchain_options.cmake
│   └── utils.cmake
├── CMakeLists.txt          # 项目根目录CMakeLists文件，cmake入口文件
├── conf                    # 项目配置文件存放目录
```

```
├── doc                     # 项目文档存放目录
├── platforms               # 项目平台性相关内容存放目录，包括交叉编译
│   └── linux
│       └── arm.himix200.cmake
├── README.md               # 项目说明
├── scripts                 # 相关脚本存放目录，包括持续集成和部署相关
├── src                     # 项目源码目录
│   ├── CMakeLists.txt
│   ├── common
│   ├── logger
│   └── main
└── tests                   # 测试示例源码存放目录
    ├── CMakeLists.txt
    └── test_logger.cpp
```

4.2 项目版本的管理

不管是SDK或者是APP项目，都会有一个版本，用来记录软件发布的每个节点。软件版本可以方便用户或者自己清楚的知道每个版本都有哪些内容的更新，可以对版本做出使用的选择或者解决版本中遇到的bug。实现版本的管理，需要能够在编译过程中清楚的体现当前版本号，在软件中也能够获取版本号。这里版本编号的管理使用常见的 `major.minor(.patch)` 格式，major是最大的版本编号，minor为其次，patch对应着小版本里的补丁级别。当有极大的更新时，会增加major的版号，而当有大更新，但不至于更新major时，会更新minor的版号，若更新比较小，例如只是bug fixing，则会更新patch的版号。版本号格式示例：`v1.0` 、 `v1.2.2` 等。

在优雅的构建软件模板中，我们将版本信息放置于 `src/common/version.hpp` 文件中：

> 注：所有的文件路径都是相对项目根目录而言。

```
#pragma once

// for cmake
// 用于在CMakeLists文件中解析用
// 0.1.0
#define HELLO_APP_VER_MAJOR 0
#define HELLO_APP_VER_MINOR 1
#define HELLO_APP_VER_PATCH 0

#define HELLO_APP_VERSION (HELLO_APP_VER_MAJOR * 10000 + HELLO_APP_VER_MINOR * 100 + HELLO_APP_VER_PATCH)

// for source code
// 用于在项目源码中获取版本号字符串
// v0.1.0
#define _HELLO_APP_STR(s) #s
#define HELLO_PROJECT_VERSION(major, minor, patch) "v" _HELLO_APP_STR(major.minor.patch)
```

在CMakeLists模块文件中我们去解析该文件获取版本号到CMake变量中，在 `cmake/utils.cmake` 添加宏函数：

```
FUNCTION(hello_app_extract_version)
    FILE(READ "${CMAKE_CURRENT_LIST_DIR}/src/common/version.hpp" file_contents)
    STRING(REGEX MATCH "HELLO_APP_VER_MAJOR ([0-9]+)" _  "${file_contents}")
    IF(NOT CMAKE_MATCH_COUNT EQUAL 1)
        MESSAGE(FATAL_ERROR "Could not extract major version number from version.hpp")
    ENDIF()
    SET(ver_major ${CMAKE_MATCH_1})

    STRING(REGEX MATCH "HELLO_APP_VER_MINOR ([0-9]+)" _  "${file_contents}")
    IF(NOT CMAKE_MATCH_COUNT EQUAL 1)
        MESSAGE(FATAL_ERROR "Could not extract minor version number from version.hpp")
    ENDIF()
    SET(ver_minor ${CMAKE_MATCH_1})
    STRING(REGEX MATCH "HELLO_APP_VER_PATCH ([0-9]+)" _  "${file_contents}")
    IF(NOT CMAKE_MATCH_COUNT EQUAL 1)
        MESSAGE(FATAL_ERROR "Could not extract patch version number from version.hpp")
    ENDIF()
    SET(ver_patch ${CMAKE_MATCH_1})

    SET(HELLO_APP_VERSION_MAJOR ${ver_major} PARENT_SCOPE)
    SET (HELLO_APP_VERSION "${ver_major}.${ver_minor}.${ver_patch}" PARENT_SCOPE)
ENDFUNCTION()
```

在根目录CMakeLists中调用版本宏：

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.4)

#-----------------------------------------
# Project setting
#-----------------------------------------
INCLUDE(cmake/utils.cmake)
HELLO_APP_EXTRACT_VERSION()

PROJECT(HelloApp VERSION ${HELLO_APP_VERSION} LANGUAGES CXX)

MESSAGE(INFO "--------------------------------")
MESSAGE(STATUS "Build HelloApp: ${HELLO_APP_VERSION}")
```

在后面的动静态库生成中就可以设定SOVERSION了，如：

```
SET_TARGET_PROPERTIES(MyLib PROPERTIES VERSION ${HELLO_APP_VERSION}
                                       SOVERSION ${HELLO_APP_VERSION_MAJOR})
```

这样就会生成一个 `liMyLibr.so => liMyLib.so.0 => libMyLib.so.0.1.1` 的库和相关软链接。不过这个操作谨慎使用，因为在android平台jni依赖带版本的库是无法找到的。

4.3 第三方库库依赖

第三方库依赖需要我们自己写库和头文件查找函数，三方库存放位置以平台和架构作为区分，目录结构随着工程的创建就基本不会改变了。库发现宏函数如下示例：

```
# Once done, this will define
#
#  SPDLOG_INCLUDE_DIR - the SPDLOG include directory
#  SPDLOG_LIBRARY_DIR - the SPDLOG library directory
#  SPDLG_LIBS - link these to use SPDLOG
#
#  ......

MACRO(LOAD_LIBSPDLOG os arch)
    SET(3RDPARTY_DIR ${PROJECT_SOURCE_DIR}/3rdparty/target/${${os}}_${${arch}})
    MESSAGE(STATUS "3RDPARTY_DIR: ${3RDPARTY_DIR}")
    FIND_FILE(SPDLOG_INCLUDE_DIR include ${3RDPARTY_DIR} NO_DEFAULT_PATH)
    FIND_FILE(SPDLOG_LIBRARY_DIR lib ${3RDPARTY_DIR} NO_DEFAULT_PATH)

    SET(SPDLOG_LIBS
        spdlog
        pthread
        #PARENT_SCOPE no parent
    )
    IF(SPDLOG_INCLUDE_DIR)
        SET(SPDLOG_LIBRARY_DIR "${SPDLOG_LIBRARY_DIR}/spdlog")
        MESSAGE(STATUS "SPDLOG_INCLUDE_DIR : ${SPDLOG_INCLUDE_DIR}")
        MESSAGE(STATUS "SPDLOG_LIBRARY_DIR : ${SPDLOG_LIBRARY_DIR}")
        MESSAGE(STATUS "SPDLOG_LIBS : ${SPDLOG_LIBS}")
    ELSE()
        MESSAGE(FATAL_ERROR "SPDLOG_LIBS not found!")
    ENDIF()
ENDMACRO()
```

> 注意：如 `SPDLOG_LIBS` 变量如果宏函数在根目录CMakeLists中调用，所以变量作用域可以作用到所有子目录，如果不是在根目录调用，则需要设置 `PARENT_SCOPE` 属性。

在主CMakeLists中调用宏函数实现三方库的信息导入：

```
INCLUDE(cmake/load_3rdparty.cmake)

IF(NOT YOUR_TARGET_OS)
    SET(YOUR_TARGET_OS linux)
ENDIF()
```

```
IF(NOT YOUR_TARGET_ARCH)
    SET(YOUR_TARGET_ARCH x86-64)
ENDIF()
MESSAGE(STATUS "Your target os : ${YOUR_TARGET_OS}")
MESSAGE(STATUS "Your target arch : ${YOUR_TARGET_ARCH}")

LOAD_LIBSPDLOG(YOUR_TARGET_OS YOUR_TARGET_ARCH)
```

4.4 第三方库源码依赖

如果你想依赖第三方项目源码，一起编译，则我们可以通过 `git submodule` 来管理第三方源码，实现源码依赖和它的版本管理。当然你可以不用git submodule，直接将源码手动放入 `3rdparty` 目录中。

添加一个git submodule:

```
# url为git项目地址
# path为项目存放目录，可以多级目录，目录名一般为项目名称
# git add <url.git> <path>
# 示例，执行后，会直接拉取项目源码到3rdparty/spdlog目录下，并创建.gitmodule在仓库根目录下
$ git submodule add  https://github.com/gabime/spdlog.git 3rdparty/spdlog
```

还可以做到带指定分支进行添加操作:

```
# 注意：命令需要在项目根目录下执行，第一次会直接拉取源码，不用update
$ git submodule add -b v1.x   https://github.com/gabime/spdlog.git 3rdparty/spdlog
$ git submodule update --remote
```

最后的 `.gitmodules` 文件为:

```
[submodule "3rdparty/spdlog"]
    path = 3rdparty/spdlog
    url = https://github.com/gabime/spdlog.git
    branch = v1.x
```

实现三方项目源码编译（首先你依赖的三方项目源码是支持CMake构建方式的），在 `3rdparty/CMakeLists.txt` 中编写:

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.4)
PROJECT(HiApp3rdparty)

ADD_SUBDIRECTORY(spdlog)
```

在根目录CMakeLists.txt中包含3rdparty中CMakeLists.txt，就可以编译第三方库了:

```
ADD_SUBDIRECTORY(3rdparty)
```

通过 `TARGET_LINK_LIBRARIES` 就可以指定第三方项目名称实现链接。

4.5 功能模块添加

4.5.1 功能模块编译

比如我们要添加一个日志模块，实现对 `spdlog` 项目的一个二次封装，更好的在自己的项目中使用，那么我们建立 `src/logger` 目录，里面新建 `logger.hpp` 、 `logger.cpp` 和 `CMakeLists.txt` 三个文件，其中CMakeLists.txt内容是对该日志模块实现编译:

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.4)

AUX_SOURCE_DIRECTORY(. CURRENT_DIR_SRCS)
ADD_LIBRARY(module_logger ${CURRENT_DIR_SRCS})
# SPDLOG_LIBS 为spdlog项目库名称
TARGET_LINK_LIBRARIES(module_logger ${SPDLOG_LIBS})
```

然后在 `src/CMakeLists.txt` 中包含该日志模块的编译:

```
ADD_SUBDIRECTORY(logger)
```

在根目录 `CMakeLists.txt` 中包含子目录 `src` ，从而实现功能模块的构建:

```
ADD_SUBDIRECTORY(src)
```

> 注：为了演示，库依赖和源码依赖都是用的spdlog，这里实现日志模块的话需要选择其中一种方式。

4.5.2 可执行文件编译

如果我们需要实现可执行文件对日志模块的调用，我们可以添加 `src/main/main.cpp` 文件，在 `src/CMakeLists.txt` 中添加对可执行文件的编译：

```
# main app
SET(SRC_LIST ./main/main.cpp)

ADD_EXECUTABLE(HiApp ${SRC_LIST})
# 配置可执行文件输出目录
SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
TARGET_LINK_LIBRARIES(HelloApp module_logger)
```

当然，如果使用 `c++11` 的特性，我们可以专门创建一个cmake文件 `cmake/toolchain_options.cmake` 来配置编译选项，在其中配置 `c++11` 编译选项，并在主CMakeLists.txt中包含该cmake文件：

```
# compiler configuration
# 从cmake3.1版本开始才支持CMAKE_CXX_STANDARD配置项
IF(CMAKE_VERSION VERSION_LESS "3.1")
    IF(CMAKE_CXX_COMPILER_ID STREQUAL "GNU")
        SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=gnu++11")
    ENDIF()
ELSE()
    SET(CMAKE_CXX_STANDARD 11)
ENDIF()
```

4.6 测试样例添加

测试样例放于 `tests` 目录，并在该目录下建立CMakeLists.txt文件用于构建所有测试demo，并在主CMakeLists.txt下包含 `tests` 目录：

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.4)

PROJECT(Tests)

INCLUDE_DIRECTORIES(
    ${SPDLOG_INCLUDE_DIR}
    ${CMAKE_SOURCE_DIR}/src
)

LINK_DIRECTORIES(
    ${SPDLOG_LIBRARY_DIR}
)

FILE(GLOB APP_SOURCES *.cpp)
FOREACH(testsourcefile ${APP_SOURCES})
    STRING(REGEX MATCH "[^/]+$" testsourcefilewithoutpath ${testsourcefile})
    STRING(REPLACE ".cpp" "" testname ${testsourcefilewithoutpath})
    ADD_EXECUTABLE( ${testname} ${testsourcefile})
    SET(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR}/bin/tests)
    TARGET_LINK_LIBRARIES(${testname}
        ${SPDLOG_LIBS}
        module_logger
        )
ENDFOREACH(testsourcefile ${APP_SOURCES})
```

然后就可以在 `tests` 目录下添加测试程序了，比如 `test_logger.cpp` 或者更多的测试demo， `tests/CMakeLists.txt` 会自动将 `tests` 目录下所有源文件逐个进行可执行文件生成构建。整个测试样例的构建就完成了。

4.7 交叉编译配置

CMake给我们提供了交叉编译的变量设置，即 `CMAKE_TOOLCHAIN_FILE` 这个变量，只要我们指定交叉编译的cmake配置文件，那么cmake会导入该配置文件的中编译器配置，编译选项配置等。我们设计的交叉编译工具链配置文件存放目录在 `platforms/` 下，这里我们使用华为海思的一个编译工具，我们按类别命名，创建一个工具栏cmake配置文件 `platforms/linux/arm.himix200.cmake` ：

```
MESSAGE(STATUS "Configure Cross Compiler")
SET(CMAKE_SYSTEM_NAME Linux)
```

```
SET(CMAKE_SYSTEM_PROCESSOR arm)

SET(CMAKE_C_COMPILER       arm-himix200-linux-gcc)
SET(CMAKE_CXX_COMPILER     arm-himix200-linux-g++)

# set searching rules for cross-compiler
SET(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
SET(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
SET(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)

SET(YOUR_TARGET_OS linux)
SET(YOUR_TARGET_ARCH armv7-a)

SET(CMAKE_SKIP_BUILD_RPATH TRUE)
SET(CMAKE_SKIP_RPATH TRUE)

# set ${CMAKE_C_FLAGS} and ${CMAKE_CXX_FLAGS}flag for cross-compiled process
#SET(CROSS_COMPILATION_ARM himix200)
#SET(CROSS_COMPILATION_ARCHITECTURE armv7-a)

# set g++ param
# -fopenmp link libgomp
SET(CMAKE_CXX_FLAGS "-std=c++11 -march=armv7-a -mfloat-abi=softfp -mfpu=neon-vfpv4 \
    -ffunction-sections \
    -fdata-sections -O2 -fstack-protector-strong -lm -ldl -lstdc++\
    ${CMAKE_CXX_FLAGS}")
```

> 注意：交叉编译工具链是需要在编译主机上安装好的。另外第三方库库依赖也需要对应编译出工具链版本（源码依赖除外）。

命令行执行交叉编译：

```
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_TOOLCHAIN_FILE=../platforms/linux/arm.himix200.cmake
$ make -j
```

这样就实现了交叉编译，你也可以配置其他的交叉编译工具链。

4.8 其他

4.8.1 cmake message命令颜色凸显

我们还可以自定义初始化cmake构建的 `message` 命令打印颜色，可以方便快速的凸显出错误信息，我们可以创建一个文件 `cmake/messagecolor.cmake` ：

```
IF(NOT WIN32)
    STRING(ASCII 27 Esc)
    SET(ColourReset "${Esc}[m")
    SET(ColourBold  "${Esc}[1m")
    SET(Red         "${Esc}[31m")
    SET(Green       "${Esc}[32m")
    SET(Yellow      "${Esc}[33m")
    SET(Blue        "${Esc}[34m")
    SET(MAGENTA     "${Esc}[35m")
    SET(Cyan        "${Esc}[36m")
    SET(White       "${Esc}[37m")
    SET(BoldRed     "${Esc}[1;31m")
    SET(BoldGreen   "${Esc}[1;32m")
    SET(BoldYellow  "${Esc}[1;33m")
    SET(BOLDBLUE    "${Esc}[1;34m")
    SET(BOLDMAGENTA "${Esc}[1;35m")
    SET(BoldCyan    "${Esc}[1;36m")
    SET(BOLDWHITE   "${Esc}[1;37m")
ENDIF()

FUNCTION(message)
    LIST(GET ARGV 0 MessageType)
    IF(MessageType STREQUAL FATAL_ERROR OR MessageType STREQUAL SEND_ERROR)
        LIST(REMOVE_AT ARGV 0)
```

```
        _message(${MessageType} "${BoldRed}${ARGV}${ColourReset}")
    ELSEIF(MessageType STREQUAL WARNING)
        LIST(REMOVE_AT ARGV 0)
        _message(${MessageType}
        "${BoldYellow}${ARGV}${ColourReset}")
    ELSEIF(MessageType STREQUAL AUTHOR_WARNING)
        LIST(REMOVE_AT ARGV 0)
        _message(${MessageType} "${BoldCyan}${ARGV}${ColourReset}")
    ELSEIF(MessageType STREQUAL STATUS)
        LIST(REMOVE_AT ARGV 0)
        _message(${MessageType} "${Green}${ARGV}${ColourReset}")
    ELSEIF(MessageType STREQUAL INFO)
        LIST(REMOVE_AT ARGV 0)
        _message("${Blue}${ARGV}${ColourReset}")
    ELSE()
        _message("${ARGV}")
ENDIF()
```

在主CMakeLists.txt中导入该cmake文件，则可以改变message命令各个级别打印的颜色显示。

4.8.2 Debug与Release构建

为了方便debug，我们在开发过程中一般是编译 `Debug` 版本的库或者应用，可以利用gdb调试很轻松的就可以发现错误具体所在。在主cmake文件中我们只需要加如下设置即可：

```
IF(NOT CMAKE_BUILD_TYPE)
    SET(CMAKE_BUILD_TYPE "Debug" CACHE STRING "Choose Release or Debug" FORCE)
ENDIF()

MESSAGE(STATUS "Build type: " ${CMAKE_BUILD_TYPE})
```

在执行cmake命令的时候可以设置 `CMAKE_BUILD_TYPE` 变量值切换 `Debug` 或者 `Release` 版本编译：

```
$ cmake .. -DCMAKE_BUILD_TYPE=Release
```

4.8.3 构建后安装

对于SDK项目，我们需要对外提供头文件和编译完成后的库文件，就需要用到cmake提供的 `install` 命令了。

我们安装需求是：

- `src` 目录下的每个模块头文件都能够安装，并按原目录存放安装
- 库文件安装放于 `lib` 目录下
- 可执行文件包括测试文件放于 `bin` 目录

首先模块头文件的安装实现均在 `src/{module}/CMakeLists.txt` 中实现，这里是安装目录，并过滤掉 `.cpp` 或者 `.c` 文件以及 `CMakeLists.txt` 文件，以 `logger` 模块为例：

```
INSTALL(DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
    DESTINATION ${CMAKE_INSTALL_PREFIX}/include
    FILES_MATCHING
    PATTERN "*.h"
    PATTERN "*.hpp"
    PATTERN "CMakeLists.txt" EXCLUDE
    )
```

> 注意：在UNIX系统上， `CMAKE_INSTALL_PREFIX` 变量默认指向 `/usr/local` ，在Windows系统上，默认指向 `c:/Program Files/${PROJECT_NAME}` 。

然后是库文件的安装，也相关 `ADD_LIBRARY` 命令调用后中实现：

```
INSTALL(TARGETS module_logger
    ARCHIVE DESTINATION lib
    LIBRARY DESTINATION lib
    RUNTIME DESTINATION bin)
```

最后是可执行文件的安装，跟安装库是一样的，添加到 `ADD_EXECUTABLE` 命令调用的后面，只是因为是可执行文件，属于 `RUNTIME` 类型，cmake会自动安装到我们设置的bin目录，这里以 `HelloApp` 为例：

```
INSTALL(TARGETS HelloApp
    ARCHIVE DESTINATION lib
    LIBRARY DESTINATION lib
    RUNTIME DESTINATION bin)
```

执行安装命令：

```
$ make install DESTDIR=$PWD/install
```

则会在相对当前目录 `install/usr/local` 目录下生成：

```
.
├── bin
│   ├── HelloApp
│   └── test_logger
├── include
│   ├── common
│   │   ├── common.hpp
│   │   └── version.hpp
│   └── logger
│       └── logger.hpp
└── lib
    └── libmodule_logger.a
```

至此，安装完成。

5 总结

“工欲善其事，必先利其器”，把基础筑好，在软件开发过程中也是很重要的，就如项目中需求明确一样，本篇文章我把 `C/C++` 项目开发的整体框架形成一个模板，不断总结改进，方便后续类似项目的快速开发。本篇文章也主要实现项目构建方面的内容，下一篇准备实现一个基本 `C/C++` 框架所必须的基础模块，包括日志模块、线程池、常用基础功能函数模块、配置导入模块、单元测试、内存泄露检查等。如有问题或者改进，一起来交流学习，最后欢迎大家关注我的公众号 `小白AI` ，不打广告，不为了写而写，只为了分享自己的学习过程^_^。

整个构建模板源码可以在我的github上找到，欢迎star：https://github.com/yicm/CMakeCppProjectTemplate

6 参考资料

- http://www.oolap.com/cxx-dependency-management-old
- http://www.yeolar.com/note/2014/12/16/cmake-how-to-find-libraries/