Name5566
分享我的所学所感

CMake 入门

2012 年 03 月 24 日 name5566 No Comments

参考文献列表：
http://www.cmake.org/
http://www.ibm.com/developerworks/cn/linux/l-cn-cmake/index.html

文本主要针对 CMake 2.8 编写

CMake 是一个跨平台的，开源的构建系统（Build System）。CMake 可以通过 CMakeLists.txt 文件来产生特定平台的标准的构建文件，例如为 Unix 平台生成 makefiles，为 Windows MSVC 生成 projects/workspaces。

CMake Hello World

首先编写一个简单的程序（main.cpp）：

```
1. #include <stdio.h>
2. int main()
3. {
4.     printf("Hello World");
5.     return 0;
6. }
```

编写 CMakeLists.txt 并和 main.cpp 放在同一个目录下：

```
1. project(Main)
2. cmake_minimum_required(VERSION 2.8)
3. aux_source_directory(. DIR_SRCS)
4. add_executable(Main ${DIR_SRCS})
```

在 CMakeLists.txt 所在的目录下执行 CMake . 生成构建文件：

```
1. D:\HelloWorld>CMake .
2. -- Building for: Visual Studio 10
3. -- The C compiler identification is MSVC 16.0.303
   19.1
4. -- The CXX compiler identification is MSVC 16.0.3
   0319.1
5. -- Check for working C compiler using: Visual Stu
   dio 10
6. -- Check for working C compiler using: Visual Stu
   dio 10 -- works
7. -- Detecting C compiler ABI info
8. -- Detecting C compiler ABI info - done
9. -- Check for working CXX compiler using: Visual S
   tudio 10
10. -- Check for working CXX compiler using: Visual S
    tudio 10 -- works
11. -- Detecting CXX compiler ABI info
12. -- Detecting CXX compiler ABI info - done
13. -- Configuring done
14. -- Generating done
15. -- Build files have been written to: D:/HelloWorl
    d
```

这里使用的平台为 Windows 并且安装了 VS2010，CMake 为我们生成了 VS2010 的构建文件，我们可以使用 VS2010 来构建应用程序。

CMake 的基本语法

我们编写的 CMakeLists.txt 需要符合一定的语法规则。一个 CMakeLists.txt 文件主要由 CMake 命令组成。

注释的写法

在 CMake 中，注释由 # 字符开始到此行的结束。

命令简介

我们见到的 project、add_executable 等为命令（Commands），命令名不区分大小写（参数需区分大小写），命令由命令名、参数列表组成，参数间使用空格进行分隔。使用一对双引号包裹的被认为是一个参数。我们的命令可以是一个内置命令（如 project），也可以是一个用户定义的宏（macro）或者函数（function）。

数据类型

CMake 的基本数据类型是字符串，一组字符串在一起被叫做一个 list（列表），例如：

```
1. # 通过 set 命令构建一个 list VAR
2. set(VAR a b c)
```

使用语法 ${VariableName} 来访问名字为 VariableName 的变量的值（变量名区

分大小写）。需要注意的是，即使在字符串中也可以使用 ${VariableName} 来访问变量的值：

```
set(VAR a b c)
# 输出 VAR = a;b;c
message("VAR = ${VAR}")
```

使用语法 $ENV{VariableName} 来访问环境变量的值（ENV{VariableName} 则表示环境变量本身）：

```
# 输出环境变量 PATH 的值
message($ENV{PATH})
```

条件控制和循环结构

条件控制命令为 if 命令

```
if(expression)
    # ...
elseif(expression2)
    # ...
else()
    # ...
endif()
```

对于 if(string) 来说：

1. 如果 string 为（不区分大小写）1、ON、YES、TRUE、Y、非 0 的数则表示真
2. 如果 string 为（不区分大小写）0、OFF、NO、FALSE、N、IGNORE、空字符串、以 -NOTFOUND 结尾的字符串则表示假
3. 如果 string 不符合上面两种情况，则 string 被认为是一个变量的名字。变量的值为第二条所述的各值则表示假，否则表示真

来看一个例子：

```
# 此策略（Policy）在 CMake 2.8.0 才被引入
# 因此这里需要指定最低 CMake 版本为 2.8
cmake_minimum_required(VERSION 2.8)

set(YES 0)

# 输出 True
if(YES)
    message(True)
else()
    message(False)
endif()

# 输出 False
if(${YES})
    message(True)
else()
    message(False)
endif()
```

表达式中可以包含操作符，操作符包括：

1. 一元操作符，例如：EXISTS、COMMAND、DEFINED 等
2. 二元操作符，例如：EQUAL、LESS、GREATER、STRLESS、STRGREATER 等
3. NOT（非操作符）
4. AND（与操作符）、OR（或操作符）

操作符优先级：一元操作符 > 二元操作符 > NOT > AND、OR

常用操作符介绍：

1. if(NOT expression)
   为真的前提是 expression 为假
2. if(expr1 AND expr2)
   为真的前提是 expr1 和 expr2 都为真
3. if(expr1 OR expr2)
   为真的前提是 expr1 或者 expr2 为真
4. if(COMMAND command-name)
   为真的前提是存在 command-name 命令、宏或函数且能够被调用
5. if(EXISTS name)
   为真的前提是存在 name 的文件或者目录（应该使用绝对路径）
6. if(file1 IS_NEWER_THAN file2)
   为真的前提是 file1 比 file2 新或者 file1、file2 中有一个文件不存在（应该使用绝对路径）
7. if(IS_DIRECTORY directory-name)
   为真的前提是 directory-name 表示的是一个目录（应该使用绝对路径）
8. if(variable|string MATCHES regex)
   为真的前提是变量值或者字符串匹配 regex 正则表达式
9. if(variable|string LESS variable|string)
   if(variable|string GREATER variable|string)
   if(variable|string EQUAL variable|string)
   为真的前提是变量值或者字符串为有效的数字且满足小于（大于、等于）的条件
10. if(variable|string STRLESS variable|string)
    if(variable|string STRGREATER variable|string)

if(variable|string STREQUAL variable|string)

为真的前提是变量值或者字符串以字典序满足小于（大于、等于）的条件

11. if(DEFINED variable)

为真的前提是 variable 表示的变量被定义了

foreach 循环范例：

```
1. set(VAR a b c)
2. foreach(f ${VAR})
3.     message(${f})
4. endforeach()
```

while 循环范例：

```
1. set(VAR 5)
2. while(${VAR} GREATER 0)
3.     message(${VAR})
4.     math(EXPR VAR "${VAR} - 1")
5. endwhile()
```

函数和宏的定义

函数会为变量创建一个局部作用域，而宏则使用全局作用域。范例：

```
1. # 定义一个宏 hello
2. macro(hello MESSAGE)
3.     message(${MESSAGE})
4. endmacro()
```

```
# 调用宏 hello
hello("hello world")
# 定义一个函数 hello
function(hello MESSAGE)
    message(${MESSAGE})
endfunction()
```

函数和宏可以通过命令 return() 返回，但是函数和宏的返回值必须通过参数传递出去。例如：

```
cmake_minimum_required(VERSION 2.8)

function(get_func RESULT)
    # RESULT 的值为实参的值，因此需要使用 ${RESULT}
    # 这里使用 PARENT_SCOPE 是因为函数会构建一个局部作用域
    set(${RESULT} "Hello Function" PARENT_SCOPE)
endfunction()

macro(get_macro RESULT)
    set(${RESULT} "Hello Macro")
endmacro()

get_func(V1)
# 输出 Hello Function
message(${V1})

get_macro(V2)
# 输出 Hello Macro
message(${V2})
```

字符串的一些问题

字符串可跨行且支持转移字符，例如：

```
set(VAR "hello
world")
# 输出结果为:
# ${VAR} = hello
# world
message("\${VAR} = ${VAR}")
```

CMake 常用命令

这里介绍一下常用的命令（CMake 2.8 的命令可以在此查询）：

project 命令

命令语法：project(<projectname> [languageName1 languageName2 …])
命令简述：用于指定项目的名称
使用范例：project(Main)

cmake_minimum_required 命令

命令语法：cmake_minimum_required(VERSION major[.minor[.patch[.tweak]]] [FATAL_ERROR])
命令简述：用于指定需要的 CMake 的最低版本
使用范例：cmake_minimum_required(VERSION 2.8)

aux_source_directory 命令

命令语法：aux_source_directory(<dir> <variable>)
命令简述：用于将 dir 目录下的所有源文件的名字保存在变量 variable 中

使用范例：aux_source_directory(. DIR_SRCS)

add_executable 命令

命令语法：add_executable(<name> [WIN32] [MACOSX_BUNDLE] [EXCLUDE_FROM_ALL] source1 source2 … sourceN)
命令简述：用于指定从一组源文件 source1 source2 … sourceN 编译出一个可执行文件且命名为 name
使用范例：add_executable(Main ${DIR_SRCS})

add_library 命令

命令语法：add_library([STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL] source1 source2 … sourceN)
命令简述：用于指定从一组源文件 source1 source2 … sourceN 编译出一个库文件且命名为 name
使用范例：add_library(Lib ${DIR_SRCS})

add_dependencies 命令

命令语法：add_dependencies(target-name depend-target1 depend-target2 …)
命令简述：用于指定某个目标（可执行文件或者库文件）依赖于其他的目标。这里的目标必须是 add_executable、add_library、add_custom_target 命令创建的目标

add_subdirectory 命令

命令语法：add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])
命令简述：用于添加一个需要进行构建的子目录
使用范例：add_subdirectory(Lib)

target_link_libraries 命令

命令语法：target_link_libraries(<target> [item1 [item2 […]]] [[debug|optimized|general]] …)

命令简述：用于指定 target 需要链接 item1 item2 …。这里 target 必须已经被创建，链接的 item 可以是已经存在的 target（依赖关系会自动添加）
使用范例：target_link_libraries(Main Lib)

set 命令

命令语法：set(<variable> <value> [[CACHE <type> <docstring> [FORCE]] | PARENT_SCOPE])
命令简述：用于设定变量 variable 的值为 value。如果指定了 CACHE 变量将被放入 Cache（缓存）中。
使用范例：set(ProjectName Main)

unset 命令

命令语法：unset(<variable> [CACHE])
命令简述：用于移除变量 variable。如果指定了 CACHE 变量将被从 Cache 中移除。
使用范例：unset(VAR CACHE)

message 命令

命令语法：
message([STATUS|WARNING|AUTHOR_WARNING|FATAL_ERROR|SEND_ERROR] “message to display” …)
命令简述：用于输出信息
使用范例：message(“Hello World”)

include_directories 命令

命令语法：include_directories([AFTER|BEFORE] [SYSTEM] dir1 dir2 …)
命令简述：用于设定目录，这些设定的目录将被编译器用来查找 include 文件
使用范例：include_directories(${PROJECT_SOURCE_DIR}/lib)

find_path 命令

命令语法：find_path(<VAR> name1 [path1 path2 …])
命令简述：用于查找包含文件 name1 的路径，如果找到则将路径保存在 VAR 中（此路径为一个绝对路径），如果没有找到则结果为 <VAR>-NOTFOUND。默认的情况下，VAR 会被保存在 Cache 中，这时候我们需要清除 VAR 才可以进行下一次查询（使用 unset 命令）。
使用范例：

```
1. find_path(LUA_INCLUDE_PATH lua.h ${LUA_INCLUDE_FI
   ND_PATH})
2. if(NOT LUA_INCLUDE_PATH)
3.     message(SEND_ERROR "Header file lua.h not fou
   nd")
4. endif()
```

find_library 命令

命令语法：find_library(<VAR> name1 [path1 path2 …])
命令简述：用于查找库文件 name1 的路径，如果找到则将路径保存在 VAR 中（此路径为一个绝对路径），如果没有找到则结果为 <VAR>-NOTFOUND。一个类似的命令 link_directories 已经不太建议使用了

add_definitions 命令

命令语法：add_definitions(-DFOO -DBAR …)
命令简述：用于添加编译器命令行标志（选项），通常的情况下我们使用其来添加预处理器定义
使用范例：add_definitions(-D_UNICODE -DUNICODE)

execute_process 命令

命令语法：

```
execute_process(COMMAND <cmd1> [args1...]]
                [COMMAND <cmd2> [args2...] [...]]
                [WORKING_DIRECTORY <directory>]
                [TIMEOUT <seconds>]
                [RESULT_VARIABLE <variable>]
                [OUTPUT_VARIABLE <variable>]
                [ERROR_VARIABLE <variable>]
                [INPUT_FILE <file>]
                [OUTPUT_FILE <file>]
                [ERROR_FILE <file>]
                [OUTPUT_QUIET]
                [ERROR_QUIET]
                [OUTPUT_STRIP_TRAILING_WHITESPACE]
                [ERROR_STRIP_TRAILING_WHITESPACE])
```

命令简述：用于执行一个或者多个外部命令。每一个命令的标准输出通过管道转为下一个命令的标准输入。WORKING_DIRECTORY 用于指定外部命令的工作目录，RESULT_VARIABLE 用于指定一个变量保存外部命令执行的结果，这个结果可能是最后一个执行的外部命令的退出码或者是一个描述错误条件的字符串，OUTPUT_VARIABLE 或者 ERROR_VARIABLE 用于指定一个变量保存标准输出或者标准错误，OUTPUT_QUIET 或者 ERROR_QUIET 用于忽略标准输出和标准错误。

使用范例：execute_process(COMMAND ls)

file 命令

命令简述：此命令提供了丰富的文件和目录的相关操作（这里仅说一下比较常用的）

使用范例：

```
# 目录的遍历
# GLOB 用于产生一个文件（目录）路径列表并保存在 variable 中
# 文件路径列表中的每个文件的文件名都能匹配 globbing expressions（非正则表达式，但是类似）
# 如果指定了 RELATIVE 路径，那么返回的文件路径列表中的路径为相对于 RELATIVE 的路径
# file(GLOB variable [RELATIVE path] [globbing expressions]...)

# 获取当前目录下的所有的文件（目录）的路径并保存到 ALL_FILE_PATH 变量中
file(GLOB ALL_FILE_PATH ./*)

# 获取当前目录下的 .h 文件的文件名并保存到 ALL_H_FILE 变量中
# 这里的变量 CMAKE_CURRENT_LIST_DIR 表示正在处理的 CMakeLists.txt 文件的所在的目录的绝对路径（2.8.3 以及以后版本才支持）
file(GLOB ALL_H_FILE RELATIVE ${CMAKE_CURRENT_LIST_DIR} ${CMAKE_CURRENT_LIST_DIR}/*.h)
```

CMake 常用变量

1. UNIX 如果为真，表示为 UNIX-like 的系统，包括 Apple OS X 和 CygWin
2. WIN32 如果为真，表示为 Windows 系统，包括 CygWin
3. APPLE 如果为真，表示为 Apple 系统
4. CMAKE_SIZEOF_VOID_P 表示 void* 的大小（例如为 4 或者 8），可以使用其来判断当前构建为 32 位还是 64 位
5. CMAKE_CURRENT_LIST_DIR 表示正在处理的 CMakeLists.txt 文件的所在

的目录的绝对路径（2.8.3 以及以后版本才支持）

6. CMAKE_ARCHIVE_OUTPUT_DIRECTORY 用于设置 ARCHIVE 目标的输出路径
7. CMAKE_LIBRARY_OUTPUT_DIRECTORY 用于设置 LIBRARY 目标的输出路径
8. CMAKE_RUNTIME_OUTPUT_DIRECTORY 用于设置 RUNTIME 目标的输出路径

构建类型

CMake 为我们提供了四种构建类型：

1. Debug
2. Release
3. MinSizeRel
4. RelWithDebInfo

如果使用 CMake 为 Windows MSVC 生成 projects/workspaces 那么我们将得到上述的 4 种解决方案配置。

如果使用 CMake 生成 Makefile 时，我们需要做一些不同的工作。CMake 中存在一个变量 CMAKE_BUILD_TYPE 用于指定构建类型，此变量只用于基于 make 的生成器。我们可以这样指定构建类型：

```
1. $ CMake -DCMAKE_BUILD_TYPE=Debug .
```

这里的 CMAKE_BUILD_TYPE 的值为上述的 4 种构建类型中的一种。

编译和链接标志（选项）

C 编译标志相关变量：

1. CMAKE_C_FLAGS
2. CMAKE_C_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO]

C++ 编译标志相关变量：

1. CMAKE_CXX_FLAGS
2. CMAKE_CXX_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO]

CMAKE_C_FLAGS 或 CMAKE_CXX_FLAGS 可以指定编译标志 CMAKE_C_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO] 或 CMAKE_CXX_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO] 则指定特定构建类型的编译标志，这些编译标志将被加入到 CMAKE_C_FLAGS 或 CMAKE_CXX_FLAGS 中去，例如，如果构建类型为 DEBUG，那么 CMAKE_CXX_FLAGS_DEBUG 将被加入到 CMAKE_CXX_FLAGS 中去

链接标志相关变量：

1. CMAKE_EXE_LINKER_FLAGS
2. CMAKE_EXE_LINKER_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO]
3. CMAKE_MODULE_LINKER_FLAGS
4. CMAKE_MODULE_LINKER_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO]
5. CMAKE_SHARED_LINKER_FLAGS
6. CMAKE_SHARED_LINKER_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO]

它们类似于编译标志相关变量

编译 32 位和 64 位程序

对于 Windows MSVC，我们可以设定 CMake Generator 来确定生成 Win32 还是 Win64 工程文件，例如：

```
# 用于生成 Visual Studio 10 Win64 工程文件
CMake -G "Visual Studio 10 Win64"
# 用于生成 Visual Studio 10 Win32 工程文件
CMake -G "Visual Studio 10"
```

我们可以通过 CMake –help 来查看当前平台可用的 Generator

对于 UNIX 和类 UNIX 平台，我们可以通过编译器标志（选项）来控制进行 32 位还是 64 位构建。

多源文件目录的处理方式

我们在每一个源码目录中都会放置一个 CMakeLists.txt 文件。我们现在假定有这么一个工程：

```
HelloWorld
|
+------- Main.cpp
|
+------- CMakeLists.txt
|
+------- Lib
         |
         +------- Lib.cpp
         |
         +------- Lib.h
         |
         +------- CMakeLists.txt
```

这里 Lib 目录下的文件将被编译为一个库。首先，我们看一下 Lib 目录下的 CMakeLists.txt 文件：

```
1. aux_source_directory(. DIR_SRCS)
2. add_library(Lib ${DIR_SRCS})
```

然后，看一下 HelloWorld 目录下的 CMakeLists.txt 文件：

```
1. project(Main)
2. cmake_minimum_required(VERSION 2.8)
3. add_subdirectory(Lib)
4. aux_source_directory(. DIR_SRCS)
5. add_executable(Main ${DIR_SRCS})
6. target_link_libraries(Main Lib)
```

这里使用了 add_subdirectory 指定了需要进行构建的子目录，并且使用了 target_link_libraries 命令，表示 Main 可执行文件需要链接 Lib 库。我们执行 CMake . 命令，首先会执行 HelloWorld 目录下的 CMakeLists.txt 中的命令，当执行到 add_subdirectory(Lib) 命令的时候会进入 Lib 子目录并执行其中的 CMakeLists.txt 文件。

外部构建（out of source builds）

我们在 CMakeLists.txt 所在目录下执行 CMake . 会生成大量的文件，这些文件和我们的源文件混在一起不好管理，我们采用外部构建的方式来解决这个问题。以上面的 HelloWorld 工程来做解释：

1. 在 HelloWorld 目录下建立一个 Build 目录（Build 目录可以建立在如何地

方）

2. 进入 Build 目录并进行外部构建 CMake ..（语法为 CMake <CMakeLists.txt 的路径>，这里使用 CMake .. 表明了 CMakeLists.txt 在 Build 目录的父目录中）。这样 CMake 将在 Build 目录下生成文件

标签：Computer Science, Tools

分类：未分类
