# 1. 前言

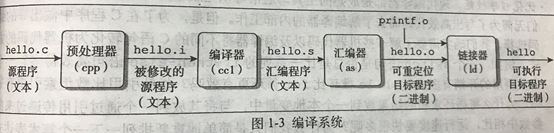
- CMake是一个跨平台的安装编译工具，可以用简单的语句来描述所有平台的安装(编译过程)。

- CMake可以说已经成为大部分C++开源项目标配

# 2. 程序编译的过程

​ 编译程序生成可执行文件四个阶段：

GCC驱动程序读取源文件.c比如hello.c，并把它翻译成一个可执行目标文件hello，需要经历一下四个阶段，如下图示：



(1)预处理阶段：预处理器（cpp）根据以字符#开头的命令，修改原始的C程序。比如hello.c中第一行的#include<stdio.h>命令告诉预处理器读取系统头文件stdio.h的内容，并把它直接插入程序文本中，结果就得到了另一个C程序，通常是以.i作为文件扩展名。

(2)编译阶段：编译器（ccl）将文本文件hello.i翻译成文本文件hello.s，它包含一个汇编语言程序。汇编语言程序中的每条语句都以一种标准的文本格式确切的描述了一条低级机器语言指令。

(3)汇编阶段：汇编器（as）将hello.s翻译成机器语言指令，把这些指令打包成一种可重定位目标程序的格式，并将结果保存在目标文件hello.o中。hello.o文件是一个二进制文件，它的字节编码是机器语言指令而不是字符，如果我们在文本文件中打开hello.o文件，看到的将是一堆乱码。

(4)链接阶段：链接器（ld）负责处理合并目标代码，生成一个可执行目标文件，可以被加载到内存中，由系统执行。

# 3. 语法特性介绍

## 3.1 基本语法格式：指令（参数1 参数2）

- 参数使用括弧( )括起

- 参数之间使用\*\*空格\*\*或\*\*分号\*\*分开

- 指令是大小写无关的，参数和变量是大小写相关的

EG:

set(HELLO hello.cpp)

add\_executable(hello main.cpp hello.cpp)

ADD\_EXECUTABLE(hello main.cpp ${HELLO})

变量使用{}方式取值，但是在 IF 控制语句中是直接是使用变量名

## 3.2 重要指令和CMake常用变量

### 3.2.1重要指令

#### （1）cmake\_minimum\_required——指定CMake的最小版本要求

eg：

# CMake最小版本要求为2.8.3

cmake\_minimum\_required(VERSION 2.8.3)

语法： cmake\_minimum\_required(VERSION versionNumber [FATAL\_ERROR])

#### （2）project——定义工程名称，并可指定工程支持的语言

Eg：

# 指定工程名为HELLOWORLD

project(HELLOWORLD)

语法：project(projectname [CXX] [C] [Java])

#### （3）set——显式的定义变量

set(<variable> <value>... [PARENT\_SCOPE]) #设置普通变量

set(<variable> <value>... CACHE <type> <docstring> [FORCE]) #设置缓存条目

set(ENV{<variable>} [<value>]) #设置环境变量

set命令可以设置普通变量、缓存条目、环境变量三种变量的值，分别对应上述三种命令格式。

1). set(<variable> <value>... [PARENT\_SCOPE]) #设置普通变量

set的值<value>...表示可以给变量设置0个或者多个值，当设置多个值时（大于2个），多个值会通过分号";"连接符连接成一个真实的值赋值给变量，当设置0个值时，实际上是把变量变为未设置状态，相当于调用unset命令。在函数内使用选项PARENT\_SCOPE，对应的作用域只能传递到调用它的函数。

将SRC设置为main.cpp ; hello.cpp

set(SRC sayhello.cpp hello.cpp)

2).set(<variable> <value>... CACHE <type> <docstring> [FORCE]) #设置缓存条目

将缓存条目variable设置为值<value>...，除非用户进行设置或使用了选项FORCE，默认情况下缓存条目的值不会被覆盖。

缓存条目可以通过CMAKE的GUI界面的add entry按钮来增加。缓存条目的实质为可以跨层级进行传递的变量，类似于全局变量。

缓存条目的<type>有：BOOL：布尔值；FILEPATH：文件路径；PATH：目录路径；STRING / STRINGS：文本行；INTERNAL：文本行，但是只用于内部，不对外呈现，主要用于运行过程中存储变量，因此使用该type意味着使用FORCE。

描述字符串 <docstring>: 单行文字,用于 CMAKE-GUI 的时提示用户.

3).set(ENV{<variable>} [<value>]) #设置环境变量

将环境变量设置为值<value>（注意没有...），接着使用$ENV{<variable>}会得到新的值。

环境变量设置的几个注意事项：

该命令设置的环境变量只在当前的cmake进程生效，既不会影响调用者的环境变量，也不会影响系统环境变量。

如果<value>值为空或者ENV{<variable>}后没有参数，则该命令会清除掉当前环境变量的值。

<value>后的参数会被忽略。

eg:set (ENV{CMAKE\_PREFIX\_PATH} "/test/sub")

message (">>> value = $ENV{CMAKE\_PREFIX\_PATH}")

#输出

>>> value = /test/sub

#### **（4）include\_directories——向工程添加多个特定的头文件搜索路径**

一般写在最外层CMakelists.txt中影响全局

Eg：

将/usr/include/myincludefolder 和 ./include 添加到头文件搜索路径

include\_directories(/usr/include/myincludefolder ./include)

- 语法： include\_directories([AFTER|BEFORE] [SYSTEM] dir1 dir2 …)

#### （5）link\_directories——向工程添加多个特定的库文件搜索路径

Eg：

将/usr/lib/mylibfolder 和 ./lib 添加到库文件搜索路径

link\_directories(/usr/lib/mylibfolder ./lib)

语法：link\_directories(dir1 dir2 …)

#### （6）add\_library——生成库文件

添加名为name的库，库的源文件可指定，也可用target\_sources()后续指定。

库的类型是STATIC(静态库)/SHARED(动态库)/MODULE(模块库)之一。生成的library名会根据STATIC或SHARED成为name.a或name.lib

Eg:

把hello.c 生成 libhello.so 共享库

add\_library(libhello SHARED hello.c)

- 语法 add\_library(<name> [STATIC | SHARED | MODULE]

[EXCLUDE\_FROM\_ALL]

[source1] [source2 ...])

改libhello.so的属性–输出名字

SET\_TARGET\_PROPERTIES(libhello PROPERTIES OUTPUT\_NAME "hello")

#### （7）add\_compile\_options——添加编译参数

Eg:

添加编译参数 -Wall ；-std=c++11

add\_compile\_options(-Wall -std=c++11 -O2)

- 语法：\*\*add\_compile\_options(<option> ...)\*\*

#### （8）add\_executable——生成可执行文件

Eg:

编译main.cpp生成可执行文件main

add\_executable(main main.cpp)

- 语法：

#### （9）target\_link\_libraries——为 target 添加需要链接的共享库

Eg：

# 将hello动态库文件链接到可执行文件main

target\_link\_libraries(main hello)

- 语法：\*\*target\_link\_libraries(target library1library2…)\*\*

#### （10）add\_subdirectory——向当前工程添加存放源文件的子目录，并可以指定中间二进制和目标二进制存放的位置

Eg：

# 添加src子目录，src中需有一个CMakeLists.txt

add\_subdirectory(src)

- 语法：\*\*add\_subdirectory(source\_dir [binary\_dir] [EXCLUDE\_FROM\_ALL])\*\*

#### （11）aux\_source\_directory——发现一个目录下所有的源代码文件并将列表存储在一个变量中，这个指令临时被用来自动构建源文件列表

Eg：

# 定义SRC变量，其值为当前目录下所有的源代码文件

aux\_source\_directory(. SRC)

# 编译SRC变量所代表的源代码文件，生成main可执行文件

add\_executable(main ${SRC})

#定义CONFGEN\_SRC\_LIST变量将server/confgen目录下的所有源文件都存储在这个变量中

aux\_source\_directory(server/confgen CONFGEN\_SRC\_LIST)

- 语法： aux\_source\_directory(dir VARIABLE)

另：file()与其功能相似

#### （12）find\_library——搜索指定动态文件路径

里面的内容为自定义的变量名、动态文件名、具体路径。

Eg：

#在指定目录下查找库libceres.so，并保存在LIBPATH变量中

find\_library ((LIBPATHS libceres.so /home/wenhaolun/ubuntu18.04\_lib/ceres/)

#链接共享库

target\_link\_libraries (main ${LIBPATHS})

#### （13）list——子命令APPEND用于将元素追加到列表

List有多种用法，这里只说关键字为APPEND的追加元素。

Eg：

set(mylist aaa bbb ccc)

MESSAGE("mylist: ${mylist}")

list(APPEND mylist xxx yyy zzz)

MESSAGE("mylist: ${mylist}")

>>mylist: aaa;bbb;ccc  
>>mylist: aaa;bbb;ccc;xxx;yyy;zzz

#### （14）source\_group\_by\_dir——生成VS工程中的目录

使用的时候，先在比较顶层的CMakeLists.txt中定义该宏。然后在添加工程（add\_library或者add\_executable）的CMakeLists.txt文件中调用该宏：

source\_group\_by\_dir(all\_files)就可以生成VS工程中的目录。

### 3.2.2 CMake常用变量

#### （1）CMAKE\_C\_FLAGS gcc编译选项

CMAKE\_CXX\_FLAGS g++编译选项

Eg：

# 在CMAKE\_CXX\_FLAGS编译选项后追加-std=c++11

set( CMAKE\_CXX\_FLAGS "${CMAKE\_CXX\_FLAGS} -std=c++11")

#### （2）CMAKE\_BUILD\_TYPE 编译类型(Debug, Release)

Eg：

# 设定编译类型为debug，调试时需要选择debug

set(CMAKE\_BUILD\_TYPE Debug)

# 设定编译类型为release，发布时需要选择release

set(CMAKE\_BUILD\_TYPE Release)

#### （3）CMAKE\_BINARY\_DIR

PROJECT\_BINARY\_DIR

<projectname>\_\_BINARY\_DIR

这三个变量指代的内容是一致的。

如果是 in source build，指的就是工程顶层目录。

如果是 out-of-source 编译,指的是工程编译发生的目录。

#### （4）CMAKE\_SOURCE\_DIR

PROJECT\_SOURCE\_DIR

<projectname>\_\_SOURCE\_DIR

这三个变量指代的内容是一致的,不论采用何种编译方式,都是工程顶层目录。

也就是在 in source build时,他跟 CMAKE\_BINARY\_DIR 等变量一致。

#### （5）CMAKE\_C\_COMPILER：指定C编译器

CMAKE\_CXX\_COMPILER：指定C++编译器

#### （6）EXECUTABLE\_OUTPUT\_PATH：可执行文件输出的存放路径

#### （7）LIBRARY\_OUTPUT\_PATH：库文件输出的存放路径

# 4. CMakeLists.txt文件创建

第一步：在CMakeLists.txt文件中指定如下几项，cmake版本、工程名、构建目标app的源文件

cmake\_minimum\_required(VERSION 3.10)

# set the project name

project(CalculateSqrt)

# add the executable

add\_executable(CalculateSqrt hello.cxx)

注：可以在src的同级目录下建立build目录。

然后在build目录下执行cmake ../src。

这样编译出来的东西和cmake生成的东西，都放到了build目录下了。并且

PROJECT\_BINARY\_DIR=全路径/build

PROJECT\_SOURCE\_DIR=全路径/src

PROJECT\_BINARY\_DIR是CMake生成一系列文件的目录，包括MakeFile等文件，如果你是让他们生成在build目录中就是/home/pi/Documents/learn/helloProject/build

如果是让他们生成在helloProject目录（cmake . 命令）

那么这个变量的值就是/home/pi/Documents/learn/helloProject

PROJECT\_SOURCE\_DIR是顶级的CMakeLists.txt所在的目录，也就是/home/pi/Documents/learn/helloProject目录

1.if (POLICY CMP0054)

cmake\_policy(SET CMP0054 NEW)

endif()

policy策略

2.option(address "This is a option for address" ON)

此时表示，如果用户没有定义过address,那我address的默认值就是ON,如果用户在命令行显示改变过address的值比如为OFF,那么在脚本中address的值就是OFF。

# CMake编译工程

CMake目录结构：项目主目录存在一个CMakeLists.txt文件

\*\*两种方式设置编译规则\*\*：

1. 包含源文件的子文件夹包含CMakeLists.txt文件，主目录的CMakeLists.txt通过add\_subdirectory添加子目录即可；

2. 包含源文件的子文件夹未包含CMakeLists.txt文件，子目录编译规则体现在主目录的CMakeLists.txt中

## 编译流程

\*\*在 window 平台下使用 CMake 构建C/C++工程的流程如下:\*\*

- 手动编写 CmakeLists.txt。

- 执行命令 `cmake PATH`( PATH 是顶层CMakeLists.txt 所在的目录 )。

## 两种构建方式

- \*\*内部构建(in-source build)\*\*：不推荐使用

内部构建会在同级目录下产生一大堆中间文件，这些中间文件并不是我们最终所需要的，和工程源文件放在一起会显得杂乱无章。

```

## 内部构建# 在当前目录下，编译本目录的CMakeLists.txtcmake .

```

- \*\*外部构建(out-of-source build)\*\*：==推荐使用==

将编译输出文件与源文件放到不同目录中

```

## 外部构建 # 1. 在当前目录下，创建build文件夹mkdir build # 2. 进入到build文件夹cd build# 3. 编译上级目录的CMakeLists.txt，生成Makefile和其他文件cmake ..

```

# 构建工程中出现的问题

- LINK : fatal error LNK1104: 无法打开文件“OGLX.lib”

使用link\_directories(../OGLX/lib)

\*\*解决方式\*\*：

​ 不能使用相对路径

​ 改为：link\_directories(${PROJECT\_SOURCE\_DIR}/OGLX/lib)

- ![image-20210826104627883](C:\Users\guo\_ning\AppData\Roaming\Typora\typora-user-images\image-20210826104627883.png)

\*\*解决方式\*\*：

​ 移除specific.c和specific.h文件

- ![image-20210826105221606](C:\Users\guo\_ning\AppData\Roaming\Typora\typora-user-images\image-20210826105221606.png)

\*\*解决方式\*\*：

1.打开该项目的“属性页”对话框。

2.单击“链接器”文件夹。

3.单击“命令行”属性页。

4.将 /SAFESEH:NO 键入“附加选项”框中，然后点击应用。

或者：

在CMakeLists.txt中添加：

set(CMAKE\_SHARED\_LINKER\_FLAGS "${CMAKE\_SHARED\_LINKER\_FLAGS} /SAFESEH:NO")

- atal error C1083: 无法打开包括文件: “aol\_EnginesLayer.h”: No such file or directory

\*\*解决方式\*\*：

工程不包含EI\_dll\platformcode下的文件

- ![image-20210826111916190](C:\Users\guo\_ning\AppData\Roaming\Typora\typora-user-images\image-20210826111916190.png)

\*\*解决方式\*\*：

链接OPENGEL32.lib

- D:\xie\simulation\TextureES\_new\PFD\_dll\Suite\specific.c(225): error C2371: “Step”: 重定义；不同的基类型

\*\*解决方式\*\*：

移除specific.c和specific.h文件

- 生成PFD.lib时 编译错误

\*\*解决方式\*\*

移除PFD\_dll\fixcode目录下的源文件

- \xie\simulation\TextureES\_new\DMH\_dll\KCG\DMHIOMProcess.c(4): fatal error C1083: 无法打开包括文件: “vxworks.h”: No such file or directory

\*\*解决方式\*\*：

移除DMHIOMProcess.c文件

- lib不是有效的Win32应用程序

\*\*解决方式\*\*：

那就是因为我把生成.lib文件的项目设置为了启动项目，将启动项目改一下，改到本应该是启动项目的那个项目上

右击该项，如下图：

![img](https://img-blog.csdnimg.cn/20200310174635811.png?x-oss-process=image/watermark,type\_ZmFuZ3poZW5naGVpdGk,shadow\_10,text\_aHR0cHM6Ly9ibG9nLmNzZG4ubmV0L3dlaXhpbl80MzIzNjQyOA==,size\_16,color\_FFFFFF,t\_70)

- 0xc000007b——应用程序无法正常启动

\*\*解决方式\*\*：