**Tips**

**1. 查看cmake 版本**

终端输入：cmake –version

2、切换不同版本cmake

建立软连接：ln -sf /opt/cmake-3.13.1/bin/\*  /usr/bin/

如果出现CMake Error: Could not find CMAKE\_ROOT !!!问题说明旧版本cmake缓存还在，

hash -r清除缓存

**第五章**

**第六章 CMake**

**6.1 基本语法：**

参数用括号括起来；

参数之间用空格或者分号分开；

指令是大小写无关的，参数和变量是大小写相关的

变量使用${}方式取值，在if控制语句中可以直接取值

**6.2 重要指令：**

* **cmake\_minimum\_required：指定cmake最小版本要求**

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

例：cmake\_minimum\_rqquired(VERSION 2.8.3)

该中括号里表示如果最小版本不符合要求，采取什么备用措施

中括号里面是可选项，用不到可以不写

* **project：定义工程名，并可指定工程支持的语言**

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

例：project(HELLOWORLD)

* **set：显示的定义变量**

语法：set(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]])

例：set(SRC sayhello.cpp hello.cpp)

# 定义SRC变量，其值为sayhello.cpp hello.cpp

VALUE指的是变量值

* **include\_directories：向工程添加多个特定的头文件搜索路径**

相当于指定g++编译器的-i参数

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

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

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

* **link\_directories：向工程添加多个特定的库文件搜索路径**

语法：link\_directories(dir1 dir2 ...)

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

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

* **add\_library：生产库文件**

语法：add\_library(libname [SHARED|ATATIC|MODULE] [EXCLUDE\_FROM\_ALL] source1 source2 ...)

例：add\_library(hello SHARED ${src})

# 通过变量SRC生成libhello.so共享库

* **add\_compile\_optinons：添加编译参数**

语法：add\_compile\_optinons(<option>…)

例：add\_compile\_options(-wall -std=c++11 -o2）

# 添加编译参数 -wall -std=c++11 -o2

* **add\_executable：生成可执行文件**

语法：add\_executable(exename source1 source2 … sourceN)

例：add\_executable(main main.cpp)

# 编译main.cpp生成可执行文件main

* **target\_link\_libraries：为target添加需要链接的共享库**

相当于g++编译器-l参数

语法：target\_link\_libraries(target library1<debug|optimized>library2...)

例：target\_link\_libraries(main hello)

# 将hello动态库文件链接到可执行文件main

* **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**

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

例：add\_subdirectory(src)

# 添加src子目录，src中需有一个CMakeLists.txt

* **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，这个临时被用来自动构建源文件列表**

语法：aux\_source\_directory(dir VARIABLE)

例：# 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mian可执行文件

add\_executable(main ${SRC})

**6.3 常用变量**

* **CMAKE\_C\_FLAGS：gcc编译选项**
* **CMAKE\_CXX\_FLAGS：g++编译选项**

例：# 在CMAKE\_CXX\_FLAGS编译选项后追加-std=c++11

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

* **CMAKE\_BUILD\_TYPE：编译类型(Debug,Release)**

例：# 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)

* **CMAKE\_BINARY\_DIR**

**PROJECT\_BINARY\_DIR**

<projectname>**\_BINARY\_DIR**

1、这三个变量指代的内容是一样的；

2、如果是in source build，指的就是工程顶层目录；

3、如果是out-of-sorce编译，指的是工程编译发生的目录；

4、PROJECT\_BINARY\_DIR 根其他指令稍有区别，不过现在可以理解为是一致的。

* **CMAKE\_SOURCE\_DIR**

**PROJECT\_SOURCE\_DIR**

<projectname>**\_SOURCE\_DIR**

1、这三个变量指代的内容是一致的，不论采用何种编译方式，都是工程顶层项目；

2、也就是在 in source build 时，它跟CMAKE\_BINARY\_DIR 等变量一致；

3、PROJECT\_SOURCE\_DIR 跟其他指令稍有不同，这里可以理解他们是一致的。

* **CMAKE\_C\_COMPILER：指定C编译器**
* **CMAKE\_CXX\_COMPILER：指定C++编译器**
* **EXECUTABLE\_OUTPUT\_PATH：可执行文件输出的存放路径**
* **LIBRARY\_OUTPUT\_PATH：库文件输出的存放路径**

**6.4 Cmake 编译工程**

Cmake 目录结构：项目主目录中存在一个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 文件通过add\_subdirectory 添加到子目录即可；

2、包含源文件的子文件夹不包含CMakeLists.txt 文件，子目录编译规则体现在主目录的CMakeLists.txt 文件中。

**6.4.1 编译流程**

* linux平台下：

（1）手动编写 CMakeLists.txt 文件；

（2）执行命令 cmake PATH 生成Makefile （PATH 是顶层CMakeLists.txt 文件所在目录）；

（3）执行命令 make 进行编译。

**6.3.2 两种构建方式**

* 内部构建（不推荐）

内部构建会在同级目录下产生一大堆中间文件，会和工程源文件放在一起显得比较乱。

* 外部构建

将编译输出文件与源文件放到不同目录中

# 1. 在当前目录下，创建build文件夹

mkdir build

# 2. 进入build 文件夹

cd build

# 3. 编译上级目录的CMakeLists.txt 文件，生成Makefile和其他文件

cmake ..

ps：..代表上级目录，在内部构建中由于没有建立build文件夹，所以使用的是cmake . ，代表编译当前目录下的CMakeLists.txt文件

# 4. 执行make 命令，生成target

make

6.4 实战

6.4.1 helloworld工程

第一步，编写CMakeLists.txt文件：

1. 指定cmake版本

2. 命名project

3. add\_execuatable（可执行文件名称  cpp文件名称）

为了方便，两个文件名称通常取得一样

6.4.2 多目录工程-直接编译

6.4.3 多目录工程-生成库编译