目录

[cmake\_minimum\_requried 2](#_Toc35459075)

[project 2](#_Toc35459076)

[configure\_file 2](#_Toc35459077)

[target\_include\_directories 3](#_Toc35459078)

[set 3](#_Toc35459079)

[add\_library 4](#_Toc35459080)

[add\_subdirectory 4](#_Toc35459081)

[option 5](#_Toc35459082)

[check\_symbol\_exists 5](#_Toc35459083)

[check\_cxx\_symbol\_exists 5](#_Toc35459084)

[add\_compile\_definations 6](#_Toc35459085)

[add\_compile\_options 6](#_Toc35459086)

[include\_directories 6](#_Toc35459087)

[查找第三方库，并加载其配置（开源库等） 6](#_Toc35459088)

[find\_package 6](#_Toc35459089)

[find\_path 7](#_Toc35459090)

[find\_library 7](#_Toc35459091)

[get\_filename\_component 7](#_Toc35459092)

# cmake\_minimum\_requried

1. 指定最低cmake版本， 必须在CmakeLists.txt最上面写cmake\_minimum\_requried (VERSION <ver\_no>)
2. 指定工程名称, 设置工程名称，并在CmakeLists.txt中产生一个CMAKE\_PROJECT\_NAME的变量， 其值为工程名称。

同时该命令也将产生如下两个变量：

PROJECT\_SOURCE\_DIR， 代码路径

PROJECT\_BINARY\_DIR， 可执行程序路径

project()

存在如下两种版本：

1. project(<PROJECT-NAME> [<language-name>…])
2. project(<PROJECT-NAME>

[VERSION] <major>[.<minor>[.<patch>[.<tweak>]]]

[DESCRIPTION <project-description-string>]

[HOMEPAGE\_URL <url-string>]

[LANGUAGES <language-name>…]

)

1. 配置文件，cmake中推荐将配置文件写为动态的，如在config.h.in中写入，通过cmake命令将其生成config.h。这一步通常是拷贝操作，在config.h.in中可以通过宏定义将CmakeLists.txt中的一些变量导入到config.h中。

configure\_file(${PROJECT\_SOURCE\_DIR}/config.h.in ${PROJECT\_BINARY\_DIR}/config.h)

通过该命令，config.h将在PROJECT\_BINARY\_DIR目录下，在编码过程中，需要将PROJECT\_BINARY\_DIR加入到工程路径中，以便于查找头文件。

1. 将文件夹include到项目中，以便于查找头文件

target\_include\_directories(<target> [SYSTEM] [BEFORE]

<INTERFACE | PUBLIC | PRIVATE> [items1…]

[<INTERFACE | PUBLIC | PRIVATE>[items2…] …])

以下关于参数说明：

1. <target>必须是一个已经由add\_executable或者add\_library创建了的名称。并且不能是IMPORTED
2. INTERFACE | PUBLIC | PRIVATE， 用于产生不同的作用域,让cmake决定是否需要包含源码路径。并且产生不同的宏：

INTERFACE ------ INTERFACE\_INCLUDE\_DIRECTORY，在动态库中指定该选项可以让所有包含该动态库的工程引用该路径下的源码， 但是该库并不一定需要。

其他 ------ INCLUDE\_DIRECTORY

如：在库的CMakeLists.txt的最后加上如下命令：

target\_include\_directories(<LIB\_NAME> INTERFACE ${CMAKE\_CURRENT\_SOURCE\_DIR})

则，所有调用该库的工程不需要再额外包含库所在工程头文件指定的源码（.h）

1. 增加C++特性，如增加C++ 11的支持等。通过使用CMAKE\_CXX\_STANDARD变量为工程提供C++特性。同时设置CMAKE\_CXX\_STANDARD\_REQURIED变量用于C++特性检查

set()

set(CMAKE\_CXX\_STANDARD 11) #C++11

set(CMAKE\_CXX\_STANDARD\_REQUIRED true) #启用特性检查

set(<variable> <value>)

当<value>没有设置时，将移除<variable>

1. 生成库，在项目中生成一个指定名称的库，可以是如下库：

1）、Normal库， 通常是静态库、动态库或者接口库

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

[EXCLUDE\_FROM\_ALL]

[sources]…)

其中sources可以使用aux\_source\_directories指定路径下所有文件代替。

也可以使用target\_sources命令指定文件

STATIC： 静态库

SHARED：动态库

MODULE：可以通过dlopen（linux）或者LoadLibrary（win）打开，通常作为插件使用（接口）

如果类库没有导出任何符号，则不应当将其输出为动态库（SHARED），原因为cmkae在windows下，期望动态库有导出的符号:

Windows: \_\_cdeclspec(dllexport)

Linux: \_\_attribute\_\_((visibility(“default”))) // “hidden” 表示符号不可见

2）、Imported库

add\_library(<name> <SHARED | STATIC | MODULE | UNKNOWN> IMPORTED [GLOBAL])

将一个外部库导入到工程中，并为其提供一个名称供工程中其他模块使用，该命令配合set\_property(<target> PROPERTY IMPORTED\_LOCATION <loc>)使用，将动态库<loc>导入为<target>

3）、Object库，一个非压缩的.o(.obj)文件集合

4）、Alias 库，将库名称映射一下

5）、Interface库，并不会实际产生一个文件，并且是可变的，与Imported Target类似。主要是提供一个接口

1. 要求编译库（让add\_library在动态库文件夹中生效）

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

1）source\_dir: 当前CMakeLists.txt及代码所在目录，可以是相对路径(相对于顶层CMakeLists.txt的路径)，也可以是绝对路径

2）binary\_dir： 用于指定生产库所在路径

3）EXCLUDE\_FROM\_ALL： 该选项用于将库中的源码不加载到工程中

1. 配置

option(<OPT\_NAME> “description” ON)

打开一个选型，OPT\_NAME可以是自定义的

1. 条件判断

if(condifion)

# condition

end (condition)

1. 列表

list()

1. 检查头文件是否存在

## **check\_include\_files / check\_include\_file / check\_include\_file\_cxx**

check\_include\_files("<includes>" <variable> [LANGUAGE <language>]) *# needs "include(CheckIncludeFiles)"*

check\_include\_file(<include> <variable> [<flags>]) *# needs "include(CheckIncludeFile)"*

check\_include\_file\_cxx(<include> <variable> [<flags>]) # needs "include(CheckIncludeFileCXX)"

示例：

check\_include\_file(errno.h HAVE\_ERRNO\_H)

check\_include\_file(fcntl.h HAVE\_FCNTL\_H)

1. 符号是否存在检测。检测符号是否在指定的头文件中定义了。

1）、如果符号是宏， 支持

2）、如果符号是函数或者变量，则必须能够被link才会支持

3）、如果符号是类型、枚举值或者关键字则不能被检测到

c语言 检测：

check\_symbol\_exists(<symbol> <files> <variable>)

c++检测：

check\_cxx\_symbol\_exists(<symbol> <files> <variable>)

1. include

在使用上面的check检测函数前，应当要include其所在的cmake文件才能被使用，例如

include(CheckSymbolExists)

include(CheckIncludeFile)

include(CheckIncludeFiles)

include(CheckCSourceCompiles)

include(CheckTypeSize)

include(CPack)

include(GNUInstallDirs)

include(CMakePackageConfigHelpers)

1. 编译控制

预编译宏：

add\_compile\_definations(<macro\_name>...)，在CMakeLists.txt

添加配置宏：

#define or #cmakedefine， 在config.h.in中

添加编译选项：

add\_compile\_options(<options> ...)，可设置编译告警级别等

包含文件，用于便于编译查找 ：

include\_directories([AFTER | BEFORE] [SYSTEM] dir1 [dir2 ...]); SYSTEM选项告诉编译器到系统目录中查找

# 查找第三方库，并加载其配置（开源库等）

find\_package(<package-name> [version] [EXACT] [QUIET] [MODULE]

[REQUIRED] [[COMPONENTS][components ...]]

[OPTIONAL\_COMPONENTS components...]

[NO\_POLICY\_SCOPE])

无论是都找到都将则产生一个<package-name>\_FOUND的变量，只是找到时if判断语句为true， 否则为false。

查找三方库时，需要加载一个名为Find<package-name>.cmake的文件（module模式）。查找三方库时有两种模式：module模式及config模式，config模式需要的参数太多，略。

1. 查找头文件路径，配合find\_package，在Find<package-name>.cmake中使用，将三方库头文件的路径加载到某一个变量中。

find\_path(<variable> file-name.h/hpp

PATH

[path1]

[path2]

...

)

从一系列路径（path1、 path2 ...）中，查找出名称为file-name的文件，并将路径追加到<variable>中。

如果从这些路径中都没有找到，则产生一个<variable>-NOTFOUND的变量。

解析环境变量： $ENV{env\_name } windows， linux

1. 查找链接库路径， 配合find\_package， 在Find<package-name>.cmake中使用，将三方库的路径加载到某一个变量中

find\_library(<variable> NAMES lib\_name.lib/.so/etc

PATH

[path1]

[path2]

...

)

如果找到了， 将链接库的路径存放在variable中。

如果没有找到，返回一个<variable>-NOTFOUND的变量

解析环境变量： $ENV{env\_name } windows， linux

1. 获取完整文件名中的特定部分

get\_filename\_component(<variable> filename

PATH | #路径

ABSOLUTE | #绝对路径

NAME | #文件名

EXT | #扩展名

NAME\_WE | #去掉扩展名的名称

REALPATH #真实完整路径（去掉link）

[CACHE]

)

1. 其他

https://zhuanlan.zhihu.com/p/488312337