**CMakeList 配置说明**

[1. 适用版本号 3](#_Toc811247159)

[2. CMake build system 3](#_Toc2068047628)

[ binary target 3](#_Toc759578632)

[ pseudo target(伪 target) 3](#_Toc2016016447)

[ improted target 4](#_Toc1647669843)

[ Alias Target 4](#_Toc1803827448)

[ binary excutable 4](#_Toc1470083955)

[ binary library 5](#_Toc1545003549)

[ build specification and useage requirements 6](#_Toc1879630198)

[ target property 6](#_Toc809316881)

[ directory-scoped command 7](#_Toc1415268895)

[ Interface Library 8](#_Toc1980242409)

[ output artifacts 9](#_Toc1241923758)

[1) runtime artifacts 9](#_Toc1738955577)

[2) library output artifact 9](#_Toc536351307)

[3) archive output artifact 9](#_Toc1976679656)

[ transitive useage requirements 10](#_Toc1540469936)

[3. Cmake命令 11](#_Toc962651019)

[ script command 11](#_Toc1463712457)

[1) confile\_file() 11](#_Toc1980272967)

[a. 配置文件使用场景 11](#_Toc980645311)

[2) find\_package() 12](#_Toc1267678499)

[ project command 13](#_Toc1832549861)

[1) install 13](#_Toc988476236)

[a. 概要 13](#_Toc703219284)

[b. install 常见选项 13](#_Toc98483159)

[c. install target 14](#_Toc1812761554)

[d. install(EXPORT) 17](#_Toc319138704)

[3) export() 18](#_Toc1858897695)

[ test command 18](#_Toc898576321)

[4. 常用配置说明 18](#_Toc582520398)

[ add\_excutable() 18](#_Toc522661206)

[ add\_subdirectory() 19](#_Toc819140302)

[ include directories () 19](#_Toc1342099030)

[ target\_link\_directories() 20](#_Toc391194006)

[ target\_link\_libraries() 20](#_Toc319326497)

[ link\_directories property 21](#_Toc998442830)

[ interface\_link\_directories property 21](#_Toc1861277961)

[ CMAKE\_INSTALL\_PREFIX 和DESTDIR 22](#_Toc1864330046)

[1) CMAKE\_INSTALL\_PREFIX 22](#_Toc730589380)

[2) DESTDIR 22](#_Toc523111194)

[ Interface Libraries 22](#_Toc1132115293)

[ cmake-generators 23](#_Toc563348141)

[1) 命令行　构建工具产生器（ｂuild tool generators) 23](#_Toc1765034952)

[e. MakeFile Generaors 23](#_Toc723587223)

[f. Ninja Generators 24](#_Toc1099699449)

[1) IDE 构建工具产生器 24](#_Toc1594230960)

[ cpack-genrators 24](#_Toc116573511)

[1) CPack Archive Generator 24](#_Toc2062350468)

[a. 常用变量对于压缩包打包器 24](#_Toc910459770)

[5. camke-modules 25](#_Toc2096846479)

[ cpack 25](#_Toc895512131)

[1) CPack Generators 25](#_Toc30654621)

[2) Target package and package\_source 26](#_Toc1781912692)

[3) CPack Generators常用的变量 26](#_Toc1883988367)

[ cpack-component 27](#_Toc733873905)

[1) 变量 28](#_Toc1880395851)

[2) 命令 29](#_Toc1549266273)

[ CMakePackageConfigHelpers 30](#_Toc1053012609)

[6. cmake-toolchains 30](#_Toc1591809898)

[ Language 30](#_Toc300358946)

[ CMAKE\_<LANG>\_COMPILER\_ID 31](#_Toc1635533007)

[ 交叉编译 31](#_Toc2114471105)

[7. cmake-packages 32](#_Toc1119499248)

[ Packages 的使用 32](#_Toc830148389)

[ Config-file Packages 33](#_Toc358181463)

[ Creating Packages 33](#_Toc1438825745)

[8. cmake-developer 35](#_Toc1828591220)

[ find\_modules 35](#_Toc71975776)

[1) 标准变量的名字 35](#_Toc1155672143)

[END 36](#_Toc411696952)

# 适用版本号

3.19.1

# CMake build system

基于cmake构建系统，是组织一组高水准的二进制target；每个target对应一个可执行文件或者一个库，或者含有自定义命令的自定义的target;在构建 系统之内，target 之间的依赖的表达，决定了构建顺序和回应变化重新产生的规则；

## binary target

可执行文件通过add\_excutable()产生；库通过add\_library()产生；这些产生的二进制文件又有合适的前缀和后缀，以及目标平台的扩展；二进制target之间的依赖通过使用target\_link\_libraries()这个命令表达；

add\_library(archive archive.cpp zip.cpp lzma.cpp)

add\_executable(zipapp zipapp.cpp)

target\_link\_libraries(zipapp archive)

archive 是一个静态的库，该archive 包含编译archive.cpp zip.cpp,lzma.cpp之后的实例；zipapp 定义成为一个可执行的target,通过编译和链接zipapp.cpp；当链接zipapp可执行executable过程中，静态库archive 也被链接进来；

## pseudo target(伪 target)

有一些target 类型没有构建系统的输出，而是引入外部依赖，别名，或者其他non-build 产品；Pseudo target就是这样的target;

## improted target

一个IMPORTED target 代表先前存在的依赖；这样的target一般被上游的包的定义，而且是固定的；在声明一个IMPORTED target之后可以通过使用target\_compile\_de finitions(),target\_include\_directories(),target\_compile\_options()或者target\_link\_ libraries()改变属性；

LOCATION 可以从IMPORTED target中读取出来；IMPORTED target的作用域是它定义的地方；可以被子目录访问，但父目录和同级的不行；也可以定义GLOBAL IMPORTED target 在整个构建系统都可以访问；

## Alias Target

别名target 是一个在只读背景下与一个二进制target相互交换的名称；别名target可以用于某个库的可执行单元测试，可以是同一构建系统也可以是基于用户配置；

add\_library(lib1 lib1.cpp)

install(TARGETS lib1 EXPORT lib1Export ${dest\_args})

install(EXPORT lib1Export NAMESPACE Upstream:: ${other\_args})

add\_library(Upstream::lib1 ALIAS lib1)

In another directory, we can link unconditionally to the Upstream::lib1 target, which may be an IMPORTED target from a package, or an ALIAS target if built as part of the same buildsystem.

if (NOT TARGET Upstream::lib1)

find\_package(lib1 REQUIRED)

endif()

add\_executable(exe1 exe1.cpp)

target\_link\_libraries(exe1 Upstream::lib1)

别名target 是 not-mutable ,non-installable non-exportable;一个名称可以通过ALISED\_TRAGET 属性读取出来测试一下；

get\_target\_property(\_aliased Upstream::lib1 ALIASED\_TARGET)

if(\_aliased)

message(STATUS "The name Upstream::lib1 is an ALIAS for ${\_aliased}.")

endif()

## binary excutable

add\_executable()命令定义了以个可以执行的target;

类似于add\_xxx()这样的命令，产生一条这样的规则--在构建时期使用一个可执行的target作为一个可执行的命令；这种构建规则可以确保可执行target 在运行之前先构建；

## binary library

默认情况下，add\_library()产生一个STATIC库，除非指定其他类型（SHARED,MODULE);

BUILD\_SHARED\_LIBS 变量也可以改变改变add\_library()的默认行为；

在整个构建系统背景下，一个库是SHARED 还是STATIC,是无关紧要的；依赖规范和其他APIs 会忽略库的类型对二者之间的区分；

MODULE 库是不同于以上二者的，一般来说MODULE库不会链接target;使用运行时技术作为一个插件加载；如果一个库不输出任何符号，这个库不能是SHARED库（SHARED库必须至少输出一个符号）；

SHARED 库可能被标记为创建macOS或者iOS框架类型的target属性;

add\_library(MyFramework SHARED MyFramework.cpp)

set\_target\_properties(MyFramework PROPERTIES

FRAMEWORK TRUE

FRAMEWORK\_VERSION A # Version "A" is macOS convention

MACOSX\_FRAMEWORK\_IDENTIFIER org.cmake.MyFramework

)

对象库定义了 一组非归档的对象文件，右提供的源码文件编译而来；这些对象文件通过$<TARGET\_OBJECTS:name> 语法添加到其他target;

add\_library(archive OBJECT archive.cpp zip.cpp lzma.cpp)

add\_library(archiveExtras STATIC $<TARGET\_OBJECTS:archive> extras.cpp)

add\_executable(test\_exe $<TARGET\_OBJECTS:archive> test.cpp)

target的链接步骤会使用它资源文件之外的对象文件集合；或者对象库链接到其他target;

add\_library(archive OBJECT archive.cpp zip.cpp lzma.cpp)

add\_library(archiveExtras STATIC extras.cpp)

target\_link\_libraries(archiveExtras PUBLIC archive)

add\_executable(test\_exe test.cpp)

target\_link\_libraries(test\_exe archive)

target的链接步骤会从对象库中使用对象文件，此外在编译target源文件的时候，对象库的使用要求将得到满足，更进一步说，这些使用要求会传递给target的依赖；

对象库 不能作为add\_xx(target)中的target,但是库中的对象文件可以通过$<TARGET\_OBJECTS: objlib>语法作为add\_xx(target output)中的output部分；

## build specification and useage requirements

target\_include\_directories(),target\_compile\_definitions()和target\_compile \_op tions()命令指定了二进制target的构建规范和使用要求；以上命令分别产生INCLUDE\_DIRECORIES,COMPILE\_DEFINITIONS, and COMPILE\_OPTION 这些目标属性；或者INTERFACE\_INCLUDE\_DIRECTORIES, INTERFACE\_COMPILE\_DEFI NITIONS 和INTERFACE\_COMPILE\_OPTIONS这些目标属性；INTERFACE\_INCLUDE \_DIRECTORIES,INTERFACE\_COMPILE\_DEFINITIONS and INTERFACE\_COMPILE \_O PTIONS 这些目标属性.

以上命令有用三种模式PRIVAT,PUBLIC 和INTERFACE;PRIVATE模式只会产生non\_INTERFACE 目标属性变量；INTERFACE模式只会产生INTERFACE目标属性变量；

PUBLIC模式会产生以上两种目标属性变量；

target\_compile\_definitions(archive

PRIVATE BUILDING\_WITH\_LZMA

INTERFACE USING\_ARCHIVE\_LIB

)

Note that usage requirements are not designed as a way to make downstreams use particular COMPILE\_OPTIONS or COMPILE\_DEFINITIONS etc for convenience only. The contents of the properties must be requirements, not merely recommendations or convenience.

See the Creating Relocatable Packages section of the cmake-packages(7) manual for discussion of additional care that must be taken when specifying usage requirements while creating packages for redistribution.

## target property

在编译二进制目标的源文件时候，INCLUDE\_DIRECTORIES; COMPILE\_DEFINITIONS; COMILE\_OPTIONS 以上这些target属性被合理的使用；

在INCLUDE\_DIRECTORIES属性的条目以-I或者-isystem 为前缀并且按照定义的顺序添加到编译行中；

在COMPILE\_DEFINITIONS的属性中的条目以-D 或者D为前缀以未指明的顺序添加到编译行中；DEFINE\_SYMBOL 目标属性被添加到编译定义，作为处理SHARED 和 MODULE库目标的一种便利；

在COMPILE\_OPTIONS 属性中的条目，将针对shell进行转义；若干编译选项有特别的处理，例如POSITION\_INDEPENDENT\_CODE(该属性在MODULE 和 SHARED 库默认是true,其他是false);

INTERFACE\_INCLUDE\_DIRECTORIES ，INTERFACE\_COMPILE\_DEFINITIONS 和 INTERFACE\_COMPILE\_OPTIONS 这些目标属性的内容是使用要求；以上这些属性指定了使用者必须使用的内容，以对target进行正确编译和链接；

对于任何二进制目标，使用在target\_link libraries()命令中指定的每个目标上的每个接口属性的内容

set(srcs archive.cpp zip.cpp)

if (LZMA\_FOUND)

list(APPEND srcs lzma.cpp)

endif()

add\_library(archive SHARED ${srcs})

if (LZMA\_FOUND)

# The archive library sources are compiled with -DBUILDING\_WITH\_LZMA

target\_compile\_definitions(archive PRIVATE BUILDING\_WITH\_LZMA)

endif()

target\_compile\_definitions(archive INTERFACE USING\_ARCHIVE\_LIB)

add\_executable(consumer)

# Link consumer to archive and consume its usage requirements. The consumer

# executable sources are compiled with -DUSING\_ARCHIVE\_LIB.

target\_link\_libraries(consumer archive)

因为将源码和相应的build的目录添加到相应的INCLUDE\_DIRECTORIES属性的要求是常见的,CMAKE\_INCLUDE\_CURRENT\_DIR变量可以方便的添加到所有target的INCLU DE \_DIRECTORIES属性;CMAKE\_INCLUDE\_CURRENT\_DIR\_IN\_INTERFACE 变量可以添加到所有的target的INTERFACE\_INCLUDE\_DIRECTORIES属性; target \_link \_libra ries()命令 可以更方便的不同目录的target;

## directory-scoped command

target\_include\_directories(),target\_compile\_definitions()和target\_compile\_opti ons()这些命令只能影响一个target;add\_compile\_definitions(),add\_compile\_option s()和include\_directories()功能类似但是在目录水平上的作用域，而不是target上的作用域；

## Interface Library

接口类型的库只有头文件，接口类型库不编译源码并且不产生库文件，因此没有LOCATION;

但是它会指定使用要求例如（INTERFACE\_INCLUDE\_DIRECTORIES, INTERFACE\_COM

PILE\_DEFINITIONS, INTERFACE\_COMPILE\_OPTIONS, INTERFACE\_LINK\_LIBRARIE

S,INTERFACE\_SOURCES,INTERFACE\_POSITION\_INDEPENDENT\_CODE);只有INTE RFACE模式下，target\_include\_directories(),target\_compile\_definition()和tar get\_compile\_options(),target\_sources()和target\_link\_libraries()这些命令才能使用接口类型的库；

add\_library(Eigen INTERFACE

src/eigen.h

src/vector.h

src/matrix.h

)

target\_include\_directories(Eigen INTERFACE

$<BUILD\_INTERFACE:${CMAKE\_CURRENT\_SOURCE\_DIR}/src>

$<INSTALL\_INTERFACE:include/Eigen>

)

add\_executable(exe1 exe1.cpp)

target\_link\_libraries(exe1 Eigen)

以上这个例子, 来自Eigen接口类型库的的使用要求在编译时期使用，不会影响链接；

add\_library(pic\_on INTERFACE)

set\_property(TARGET pic\_on PROPERTY INTERFACE\_POSITION\_INDEPENDENT\_CODE ON)

add\_library(pic\_off INTERFACE)

set\_property(TARGET pic\_off PROPERTY INTERFACE\_POSITION\_INDEPENDENT\_CODE OFF)

add\_library(enable\_rtti INTERFACE)

target\_compile\_options(enable\_rtti INTERFACE

$<$<OR:$<COMPILER\_ID:GNU>,$<COMPILER\_ID:Clang>>:-rtti>

)

add\_executable(exe1 exe1.cpp)

target\_link\_libraries(exe1 pic\_on enable\_rtti)

上面这个例子target exe1，是一个被链接的并且编译器指定的标志，封装在接口类型库中；

接口库可以被安装和输出，它们必须被单独安装；

set(Eigen\_headers

src/eigen.h

src/vector.h

src/matrix.h

)

add\_library(Eigen INTERFACE ${Eigen\_headers})

target\_include\_directories(Eigen INTERFACE

$<BUILD\_INTERFACE:${CMAKE\_CURRENT\_SOURCE\_DIR}/src>

$<INSTALL\_INTERFACE:include/Eigen>

)

install(TARGETS Eigen EXPORT eigenExport)

install(EXPORT eigenExport NAMESPACE Upstream::

DESTINATION lib/cmake/Eigen

)

install(FILES ${Eigen\_headers}

DESTINATION include/Eigen

)

## output artifacts

### runtime artifacts

在DLL平台上（windows),通过add\_library(SHARED)创建；或者通过add\_excutable()创建可执行文件；

通过RUNTIME\_OUTPUT\_DIRECTORY 和 RUNTIME\_OUTPUT\_NAME,可以改变构建产品的路径和文件名字；

### library output artifact

通过add\_library(MODULE) 创建的文件；

在NON-DLL平台上，add\_library(SHARED)创建的文件；

通过LIBRARY\_OUTPUT\_DIRECTORY 和LIBRARY\_OUTPUT\_NAME 可以改变 库类型输出的路径和文件名子；

### archive output artifact

add\_library(STATIC) 产生.a文件；

通过ARCHIVE\_OUTPUT\_DIRECTORY 和ARCHIVE\_OUTPUT\_NAME 控制压缩文件的输出路径和文件名字；

On DLL platforms: the import library file (e.g. .lib) of a shared library target created by the add\_library() command with the SHARED option. This file is only guaranteed to exist if the library exports at least one unmanaged symbol.

On DLL platforms: the import library file (e.g. .lib) of an executable target created by the add\_executable() command when its ENABLE\_EXPORTS target property is set.

On AIX: the linker import file (e.g. .imp) of an executable target created by the add\_executable() command when its ENABLE\_EXPORTS target property is set.

## transitive useage requirements

传递使用条件，即一个target使用条件可以传递给它的依赖；target\_link\_libraries()这个命令，通过PUBLIC PRIVATE INTERFACE,这些关键字控制这种传递；

在下面的例子中，archive 是archiveExtras的PUBLIC 类型的依赖，这个PUBLIC使用条件也会传递给customer;serial 是archiveExtras的PRIVATE类型的依赖，这个PRIVATR使用条件不会传递给customer;

一般来说，使用target\_link\_libraries(PRIVATE)指定的使用条件，依赖只存在库的源文件中，不在库的头文件中；如果还需要依赖库的头文件，需要使用target\_link\_libraries(PUBLIC) 命令作为PUBLIC 依赖的使用条件；如果只是仅仅依赖库的头文件，只需要使用target\_link\_libraries(Interface)这个命令；

add\_library(archive archive.cpp)

target\_compile\_definitions(archive INTERFACE USING\_ARCHIVE\_LIB)

add\_library(serialization serialization.cpp)

target\_compile\_definitions(serialization INTERFACE USING\_SERIALIZATION\_LIB)

add\_library(archiveExtras extras.cpp)

target\_link\_libraries(archiveExtras PUBLIC archive)

target\_link\_libraries(archiveExtras PRIVATE serialization)

# archiveExtras is compiled with -DUSING\_ARCHIVE\_LIB

# and -DUSING\_SERIALIZATION\_LIB

add\_executable(consumer consumer.cpp)

# consumer is compiled with -DUSING\_ARCHIVE\_LIB

target\_link\_libraries(consumer archiveExtras)

# Cmake命令

## script command

### confile\_file()

configure\_file(<input> <output>

[COPYONLY] [ESCAPE\_QUOTES] [@ONLY]

[NO\_SOURCE\_PERMISSIONS]

[NEWLINE\_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])

复制<input>文件到<output>文件，并且对于<input>文件中@VAR@或者

${VAR}的变量的值代替提交；每个变量的引用会被当前的变量的值所替代，如

果变量没有定义返回空字符串；更近一步说，

#define VAR 或者/\*#undef VAR\*/ 代替　＃cmakedefine VAR,

COPYONLY 不代替<input>文件中的变量引用和其他内容；不能和

NEWLINE\_STYLE一起使用；

ESCAPE\_QUOTES 用反斜杠转义任何替换的引号;

＠ONLY 只能是＠VAR@变量用值替换引用；对于配置使用${VAR}的脚本特别有用；

NO\_SOURCE\_PERMISSIONS　不转移原始文件的读写权限到复制的文件；

NEWLINE\_STYLE <style>　换行符号

UNIX 或者 LF风格的换行符号是\n

DOS WIN32 CRLF 风格的换行符号是\r\n

#### 配置文件使用场景

工程目录下创建foo.h.in 文件，文件内容如下：

#cmakedefine FOO\_ENABLE

#cmakedefine FOO\_STRING "@FOO\_STRING@"

工程目录下CMakeList.txt添加如下配置：

option(FOO\_ENABLE "Enable Foo" ON)

if(FOO\_ENABLE)

set(FOO\_STRING "foo")

endif()

configure\_file(foo.h.in foo.h @ONLY)

cmake运行以后会在build目录下产生对应的文件foo.h,在FOO\_ENABLE 为true的情况下foo.h文件内容是如下：

#define FOO\_ENABLE

#define FOO\_STRING "foo"

在FOO\_ENABLE为false　情况下foo.h的内容是

/\* #undef FOO\_ENABLE \*/

/\* #undef FOO\_STRING \*/

### find\_package()

find\_package(<PackageName> [version][EXACT][QUIET][MODULE]

[REQUIED][[COMPONENTS][componets...]]

[OPTIONAL\_COMPONTS components...]

[NO\_POLICY\_SCOPE])

从外部工程，找到和加载设置；<PackageName>\_FOUND 暗示该包是否被发现；该命令包含两种模式：一种是MODULE模式，另一种是CONFIG模式；以上命令是首选是MODULE模式，若没搜索到，则寻找CONFIG模式；但是如果MODULE被设置则不会在CONFIG模式寻找；

EXACT 表示与版本号完全一致；

没有设置MODULE 模式下，通过一个Find<PackageName>.cmake文件寻找包，首先CMAKE\_ MODULE\_PATH,然后区CMake安装提供的Find Modules 中寻找；用户可以通过设置变量CMAKE\_FIND\_PACAKAGE\_PREFER\_CONFIG 为true直接通过CONFIG模式寻找包而不必先经过MODULE模式；

下面是CONFIG模式的命令

find\_package(<PacakageName> [version] [EXACT][QUIET]

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

[OPTIONAL\_COMPONENTS components...]

[CONFIG|NO\_MODULE]

[NO\_POLICY\_SCOPE]

[NAMES name1[name2...]]

[CONFIGS config1[config2...]]

[HINTS path1[path2...]]

[PATHs path1[path2...]]

[PATH\_SUFFIXES suffix1[suffix2...]]

[NO\_DEFAULT\_PATH]

[NO\_PACKAGE\_ROOT\_PATH]

[NO\_CMAKE\_PATH]

[NO\_CMAKE\_ENVIRONMENT\_PATH]

[NO\_SYSTEM\_ENVIRONMENT\_PATH]

[NO\_CMAKE\_PACKAGE\_REGISTRY]

[NO\_CMAKE\_BUILDS\_PATH]

[NO\_CMAKE\_SYSTEM\_PATH]

[NO\_CMAKE\_SYSTEM\_PACKAGE\_REGISTRY]

[CMAKE\_FIND\_ROOT\_PATH

|ONLY\_CMAKE\_FIND\_ROOT\_PATH

|NO\_CMAKE\_FIND\_ROOT\_PATH])

在配置模式下面，搜索尝试找到包提供的配置文件；<PackageName>\_DIR的缓存实体被创建，拥有包含该配置文件的目录；在NAMES选项被设置的情况下，替代<PackageName> 搜索包；该命令搜索一个名字叫<PackageName>Config.cmake文件或者<lower-case-package-name>-config.cmake文件;在使用CONFIGS选项的情况下可以代替配置文件的名称；

待续

## project command

### install

#### 概要

install(TARGETS <target>...[...])

install({FILES|PROGRAMS}<file>...[...])

install(DIRECTORY <dir> ...[...])

install(SCRIPT <file>[...])

install(CODE <code>[...])

install(EXPORT <export-name>[...])

该命令产生工程的安装规则。在对应的build 目录下该命令会产生一个cmake\_in stall .cmake文件；

#### install 常见选项

DESTINATION 指定文件安装的磁盘路径，参数可以是绝对或者相对路径。但是绝对路径不被cpack installer 所支持。参数是相对路径的情况下，是相对于CMAKE\_INSTALL\_ PREFIX这个变量。（在UNIX系统上，CMAKE\_INSTALL\_PREFIX 默认值是/usr/local).使用DESTDIR机制，可以重新安排整个安装路径；

PERMISSIONS

指定安装文件的权限 OWNER\_READ OWNER\_WRITE OWNER\_EXCUTE; GROUP\_ READ GROUP\_WRITE GROUP\_EXCUTE; WORLD\_READ WORLD\_WRITE WOR D\_E XCUTE;

CONFIGURATIONS

指定使用那种安装配置（DEBUG RELEASE,etc)

COMPONENT

指定与安装规则关联的组件名称，例如 runtime 或者 development;在特定组件安装过程中，只有与这些特定的组件相关的安装规则被会被执行。在组件完全安装的过程中，除了被标记为EXLUDE\_FROM\_ALL的组件，所有的组件都会被安装。如果组件没有被提供，默认组件Unspecified 将会被创建。默认组件名字也可以通过CMAKE\_INSTALL\_DEFAUL T\_COMPONET\_NAME来控制修改。

EXCLUDE\_FROM\_ALL

指定排除完全安装之外的文件

RENAME

为安装的文件指定不同于原文件名的名字

OPTIOANL

如果该文件不存在，也不是错误

#### install target

install (targets targets...[EXPORT <export-name>]

[[ARCHIVE|LIBRARY|RUNTIME|OBJECTS|FRAMEWORK|BUNDLE|

PRIVATE\_HEADER|PUBLIC\_HEADER|RESOURCE]

[DESTINATION <dir>]

[PERMISSION permissions...]

[CONFIGURATIONS [Debug|Release|...]]

[COMPONENT <component>]

[NAMELINK\_COMPONENT <component>]

[OPTINAL][EXCLUDE\_FROM\_ALL]

[NAMELINK\_ONLY|NAMELINK\_SKIP]

][...]

[INCLUDES DESTINATION [<dir>...]]

)

ARCHIVE

除了ＭacOs系统上被标记为FRAMEWORK标记的静态库target；

DLL导入库(DLL import libraries)(不同于.dll库，后缀为.lib）target

LIBRARY

ＤLLs和ＭacOS上被标记为FRAMWORK之外共享库target；

RUNTIME

除了在MacOs上被标记为MACOSX\_BUNDLE的可执行文件的target

DLLs可执行文件的target(不包括导入库，导入库是ARCHIIVE);

OBJECTS

对象库相关的对象文件

FRAMWORK

在ＭacOS上，被标记为FRAMEWORK属性标签的static或者shared库都被视作 FRAMEWORK

BUNDLE

在MacOS系统上被标记为MACOSX\_BUNDLE属性的可执行文件；

PUBLIC\_HEADER

在被标记为FRAMEWORK的SHARED库中，指定公开的头文件；

PRIVATE\_HEADER

在被标记为FRAMEWORK的SHARED库中，指定私有头文件；

RESOURCE

FRAMWORK和BUNDLE，以及SHARE库 这些类型的target,通过Resource属性 可以设置一系列的资源文件，放在对应的目录；

add\_executable(ExecutableTarget

addDemo.c

resourcefile.txt

appresourcedir/appres.txt)

target\_link\_libraries(ExecutableTarget heymath mul)

set(RESOURCE\_FILES

resourcefile.txt

appresourcedir/appres.txt)

set\_target\_properties(ExecutableTarget PROPERTIES

MACOSX\_BUNDLE TRUE

MACOSX\_FRAMEWORK\_IDENTIFIER org.cmake.ExecutableTarget

RESOURCE "${RESOURCE\_FILES}")

对于Linux系统产生下列文件

ExecutableTarget

Resources

appres.txt

resourcefile.txt

For each of these arguments given, the arguments following them only apply to the target or file type specified in the argument. If none is given, the installation properties apply to all target types. If only one is given then only targets of that type will be installed (which can be used to install just a DLL or just an import library.)

对于常规的可执行文件，静态库和共享库，DESTINATION参数是不需要的；会有默认的destination会从GUNInstallDirs中取得合适的变量；DESTINATION经常用于模块库，框架，Apple Bunles;对于接口和对象库DESTINATION可以被省略；下面的表格展示了DESTINATION没有设置的情况下，不同的target的默认destination

|  |  |  |
| --- | --- | --- |
| target type | GNUInstallDirs Variable | Built-In Default |
| RUNTIME | ${CMAKE\_INSTALL\_BINDIR} | bin |
| LIBRARY | ${CMAKE\_INSTALL\_LIBDIR} | lib |
| ARCHIVE | ${CMAKE\_INSTALL\_LIBDIR} | lib |
| PRIVATE\_HEADER | ${CMAKE\_INSTALL\_INCLUDEDIR} | include |
| PUBLIC\_HEADER | ${CMAKE\_INSTALL\_INCLUDEDIR} | include |

NAMELINK\_COMPONET

一些平台上一个版本化的共享库，有一个符号链接，如下

libtools.so -> libtools.so.1

对于组件有类似的效果，必须在LIBRARY才有效果；

install(TARGETS mylib

LIBRARY

COMPONENT Libraries

NAMELINK\_COMPONENT Development

PUBLIC\_HEADER

COMPONENT Development

)

NAMELINK\_ONLY

在库类型的target安装的时候，NAMELINK\_ONLY造成只要nameklink安装；只要 版本化的库没有namelinks 或者没有版本化的库，NAMELINK\_ONLY不起作用；必 须是LIBRARY的作用域才能使用NAMELINK\_ONLY,否则会出错；

NAMELINK\_SKIP

类似NAMELINK\_ONLY,但是它是相反的效果；在库类型安装的时候， NAMELINK\_ONLY造成namelink之外的文件安装；对于版本化的共享库没有符号链 接，或者没有版本化的链接库，NAMELINK\_SKIP完全安装库文件；该选项必须在 LIBRARY作用域；

NAMELINK\_SKIP会导致NAMELINK\_COMPENT不生效；不建议二者一起使用；

EXPORT

该选项紧跟着<export-name>,必须放在任何target选项之前；要安装导出文件自身，可以调用install<export>,参看下面的文档；

INCLUDES DESTINATION

当使用install(export)导出的时候，该选项指定了一系列路径，这些路径会被添加到<targets>的INTERFACE\_INCLUDE\_DIRECTORIES 属性；

一个target可以反复调用install()安装不同位置

install(TARGETS myExe mySharedLib myStaticLib

RUNTIME DESTINATION bin

LIBRARY DESTINATION lib

ARCHIVE DESTINATION lib/static)

install(TARGETS mySharedLib DESTINATION /some/full/path)

#### install(EXPORT)

install(EXPORT <export-name> DESTINATIONB <dir>

[NAMESPACE <namespace>][FILE <name>.cmake]|

[PERMISSIONS permissions...]

[CONFIGURATIONS [Debug|Release|...]]

[EXPORT\_LINK\_INTERFACE\_LIBRARIES]

[COMPONENT <component>]

[EXCLUDE\_FROM\_ALL])

这种export 形式产生和安装一个.cmake文件，该.cmake文件包含将target从安装树导入到另一个工程的代码；target安装和install(TARGETS)中的export选项<export-name>相关联；如果NAMESPACE选项被配置<namespace>作为target的名称前缀写到.cmake文件；在没有设置FILE选项时候产生<export-name>.cmake名称的文件；如果设置EXPORT\_LINK\_INTERFACE\_LIBRARIES 关键字（CMP0022是新的情况下）造成属性的内容匹配要导出的（IMPORTED\_)?LINK\_INTERFACE\_LIBRARIES (\_<CO NFIG >)?

当COMPONENT选项被设置的时候，<component>暗含着依赖在导出集合中所有的

component;导出的.cmake文件要求每一个导出的组件都存在以便正确构建工程；？？

这种export 形式有利于外部工程使用已经在本工程构建安装的targets;看下面的例子

install(TARGETS myexe EXPORT myproj DESTINATION bin)

install(EXPORT myproj NAMESPACE mp\_ DESTINATION lib/myproj)

### export()

export(EXPORT <export-name>

[NAMESPACE <namespace>] [FILE <filename>])

创建一个名称为<filena me>.cmake文件，该文件会被外部工程include()，以便从当前build目录下输出targets 供外部的工程使用;在交叉编译时期，对构建在本机上的可运行的执行工具然后导入到另一个目标平台上的工程进行编译，这是非常有用的；如果NAMESPACE选项被设置，将作为target的前缀写到文件中；

Target的安装与使用install(TARGETS)命令中的EXPORT选项导出的名字相关联；

通过export(EXPORT)命令产生的文件指向特定的构建树并且永远不应该被安装；参考

install(EXPORT)命令从安装树中导出targets;

待续

## test command

待续

# 常用配置说明

## add\_excutable()

add\_excutable(<name>[WIN32][MACOSX\_BUNDLE][EXCLUDE\_FROM\_ALL][source1][source2...])

name 表示一个可执行的target，由source1 source2等源文件编译而来；如果稍后通过target\_source()指定源文件，这里可以省略source1 source2; name必须在整个工程内独一无二的；实际构建可执行的文件名字取决与本机平台惯例；

默认情况下，可执行文件创建在与源码一致的构建目录里；可以通过RUNTIME\_OUTPUT\_DIRECTORY 改变可执行文件位置；通过OUTPUT\_NAME 属性可以改变name之后文件的部分名字；

如果WIN32属性被设置，WIN32\_EXECUTABLE 将会设置到创建的target 上；

如果MACOSX\_BUNLDE属性被设置，MACOSX\_BUNDLE 将会设置到创建的target 上；

EXCLUDE\_FROM\_ALL 若该属性为true则，该target 排除当前目录其他target 以及父目录中的target,意味着运行该目录下或者父目录下的make该target 不会被构建； 若该属性为false 则该target包含当前目录的其他的target 以及父目录中的target;如果该属性没有设置,那么该target 被包含在当前目录所有的target中，更深一点讲，被包含在父目录下所有没有EXCLUDE\_FROM\_ALL 属性的target 中；

## add\_subdirectory()

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

添加要构建的子目录.source\_dir是放置CMakeList.txt 和源码的目录。可以是相对当前路径的相对路径，也可以是绝对路径；binary\_dir指定了构建之后的输出文件目录，如果是相对路径，会以当前输出目录（current\_output\_dir)计算路径，当然也可以是绝对路径；

在binary\_dir没有指定的情况下，会使用（未展开的相对路径）source\_dir代替；在add\_subdirectory处理source\_dir下的源文件之前，source\_dir下的CMakeList.txt文件会被CMake工具处理。

如果EXCLUDE\_FROM\_ALL 被指定，那么默认情况下source\_dir的target 就不能包含在父目录的target中，并且会被排除在IDE工程文件之外；在source\_dir中用户必须显式地构建target;这意味着当source\_dir是工程有用但不是必须的完全独立的一部分（例如一组例子）；尤其是source\_dir 是有自己的project(),以便在source\_dir中产生完整的构建文件；

值得注意的是内联的target会取代这种排斥；一个由父目录构建的target,并且依赖于子目录中的target，这种依赖性的target 会包含在父工程构建系统以满足依赖性；

## include directories ()

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

添加dir1 dir2 等目录给编译器搜索include files(头文件);　相对路径就是相对语当前源码的文件夹；这些目录被添加到当前的CMakeLists.txt 的INCLUDE\_DIRECTORIES 目录属性；这意味着这些目录也被添加到该CMakeList.text 内所有的target的INCLUDE\_DIRECTOR　IES 目标属性；这些目标属性值将会被生成器使用；

默认情况下这些目录会被添加当前目录的后面；也可以通过AFTER BEFORE 改变这种行为；当SYSTEM 被提供的时候，会告诉编译器这是某些平台的头文件目录；这些设置将产生一些效果，例如编译器忽略某些警告或者在依赖计算中固定安装系统不被考虑；

值得注意的是target\_include\_directories() 可以替代它；添加INCLUDE\_DIRECTORIES 给特定的target;

## target\_link\_directories()

target\_link\_directories(<target> [BEFORE]

<INTERFACE|PUBLIC|PRIVATE> [items1...]

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

当链接一个指定的target的时候,指定Linker(链接器）搜寻库的路径；每个item可以是相对路径也可以绝对路径，这些路径会被添加到link命令行中；target 必须是add\_excuteable() 或者是add\_library()命令创建的，绝对不能是别名target;

INTERFACE PUBLIC PRIVATE 指定items的作用域，PRIVATE 和PUBLIC的items路径会产生target的LINK\_DIRECTORIES的属性；PUBLIC和INTERFACE 的item路径会产生target 的INTERFACE\_LINK\_DIRECTORIES属性;每一个item指向一个链接目录，并且在添加它到相关属性之前转换成绝对路径；按照调用的顺序，重复调用同一个target添加item;

如果Before被指定，item 会被添加到相关属性的前面而不是按顺序添加；

值得注意的是这个命令很少情况下是必须的，在有其他选择的情况下应该避免使用；更倾向于传库的绝对路径，以确保正确的库被链接到；find\_library()命令提供了求完整路径，在调用target\_link\_libraries()的时候可以使用find\_library();但target\_link\_directories()搜索库的路径包括一下场景：

1. 类似Xcode的IDE，在构建的时候用户可以切换target的架构），但是库的完整路径不能被提供，因为一个完整路径只提供一种架构（例如不是统一的二进制）
2. 库本身可能有其他希望通过RPATH机制找到的私有库依赖关系，但有些链接器无法完全解码这些路径(例如，由于存在像$ORIGIN这样的东西)

## target\_link\_libraries()

Specify libraries or flags to use when linking a given target and/or its dependents. Usage requirements from linked library targets will be propagated. Usage requirements of a target’s dependencies affect compilation of its own sources.

target\_link\_libraries(<target> ... <item>... ...)

target已经 通过add\_executable() 和add\_library()创建，并且不是别名Target;按照调用的顺序反复调用同一个target添加条目；

item 可以是一个库类型的target, 产生库类型的target的时候必须是完整路径的可链接库; 在工程范围内，该库类型的target必须通过add\_library()创建，或者作为IMPORTED 库；如果工程内创建，一个顺序依赖会自动添加到构建系统，以确保在target链接之前，库类型的target是最新的；如果一个导入库有IMPORTED\_NO\_SONAME 目标属性，CMake会让链接器（Linker)搜寻库，而不是用全路径（/usr/lib/libfoo.so becomes -lfoo）；指向target的全路径 会自动被shell转义；

item也可以是指向以个库文件的全路径；产生链接行会保留该文件的全路径；在库文件有改变的情况下，构建系统会有一个依赖；在一些场景下,CMkake 让链接器搜索库，例如一个共享库被设置为没有SONAME field; 指向库文件的完整路径会被shell自动转义；

item也可以是库的名字；产生链接行会让链接器搜索库（e.g. foo becomes -lfoo or foo.lib）；

item也可以是 link标志，item 以- 或者-framework 开头，但不能是-l,被视作链接器的标志；请注意，对于传递依赖关系而言，此类标志将与任何其他库链接项一样处理，因此通常将它们指定为不会传播到依赖关系的私有链接项是安全的

target\_link\_libraries(<target>

<PRIVATE|PUBLIC|INTERFACE> <item>...

[<PRIVATE|PUBLIC|INTERFACE> <item>...]...)

若是target 和库之间是PUBLIC,成为链接接口组成成分（库的头文件包含在target 源文件和头文件中）；若是target 和库之间是PRIVATE,不成为链接接口组成成分（库的头文件包含在target 源文件,但不包含在头文件中）；若是target 和库之间是Interface,不用于链接target（库的头文件不包含在target 源文件和头文件中）；

target\_link\_libraries(<target> <item>...)

默认情况下，库的依赖是可以传递的；当target A 链接到 targetB,链接到target A的库；则该库可以链接到target B;这种传递性的链接接口被存储到INTERFACE\_LINK\_LIBRARIES 目标属性，并且可以被重写；

## link\_directories property

用于链接共享库，模块，可执行target的目录列表；这个属性是拥有分号分割的目录列表，该列表指向特定的target;使用target\_link\_directories可以增加更多的搜索目录；当target被创建的时候，该属性被LINK\_DIRECTORIES 目录属性初始化，并且被产生器（genrator）用于设置链接器（Linker）搜索目录;

## interface\_link\_directories property

一个库的公共链接目录要求的列表。target 可以填充此属型以便发布该target 编译头文件所需的链接目录；可用通过target\_link\_directories ()并且指定PUBLIC 和INTERFACE 关键字实现；工程可以直接读取和设置属性；

当target 的依赖用target\_link\_libraries()被指定,CMakge就可以从该target所有依赖读取这个属性，以确定构建使用者的属性；

## CMAKE\_INSTALL\_PREFIX 和DESTDIR

### CMAKE\_INSTALL\_PREFIX

install安装命令的安装路径。在UNIX 系统上CMAKE\_INSTALL\_PREFIX默认值是/usr/local,在Window系统上默认值是c:/Program Files/${PROJECT\_NAME}.

该路径可以添加到CMAKE\_SYSTEM\_PREFIX\_PATH 以便find\_package()

find\_program() find\_library() find\_path() find\_file()会搜索该路径。

### DESTDIR

在UNIX系统上面，为了重新定位整个安装，使用DESTDIR机制。DESTDIR 代表着安装路径。为了不使用默认的软件安装位置，调用类似如下命令：

make DESTDIR=/home/john install

在使用安装前缀(install prefix)/usr/local的情况下，这个命令会安装特定的软件到/home/john/usr/local

## Interface Libraries

一个接口类型库，不编译源码，并且在磁盘上不产生库文件，因此它无LOCATION这个属性；

用于指明使用条件例如INTERFACE\_INCLUDE\_DIRECTORIES,INTERFACE\_COMPILE DEFINITIONS,INTERFACE\_COMPILE\_OPTIONS,INTERFACE\_LINK\_LIBRARIES,INTERFACE\_SOURCES和INTERFACE\_POSOTION\_INDEPENDENT\_CODE；

只有target\_include\_directories(),target\_compile\_definitions() target\_compile\_ options(),target\_sources()和target\_link\_libraries()这些命令会使用Interface库；

从CMake 3.19开始，Interface库类型的target可以包含source files;包含源文件的的Interface库会被包含在产生的构建系统中作为构建的target;它不编译源文件，但是含有其他命令产生的文件；

add\_library(Eigen INTERFACE

src/eigen.h

src/vector.h

src/matrix.h

)

target\_include\_directories(Eigen INTERFACE

$<BUILD\_INTERFACE:${CMAKE\_CURRENT\_SOURCE\_DIR}/src>

$<INSTALL\_INTERFACE:include/Eigen>

)

add\_executable(exe1 exe1.cpp)

target\_link\_libraries(exe1 Eigen)

## cmake-generators

一个ＣMake Generators负责写入构建系统的输入文件。准确的说，对于一个构建树一个CMake Generators必须被选择以便决定构建系统使用那些；

CMake Generators是和平台相关的，特定平台上只能选择相关可用的CMake Generat ors;

### 命令行　构建工具产生器（ｂuild tool generators)

#### MakeFile Generaors

MSYS Makefiles构建工具产生器, 在MYSS 系统shell中使用ＭSYS make构建工具产生makefiles ；产生的ｍakefiles 使用　/bin/sh 作为shell 以便启动构建rules;并不适用于Ｗindow　命令提示符号；

MinGW Makefiles 构建工具产生器，在Windows命令提示符号下产生makefiles,以配合mingw32- make构建工具 使用；产生的makefiles 使用cmd.exe 作为shell 以便启动构建Rules;

Unix Makefiles 构建工具产生标准的makefiles.Unix makefiles的层次树被打到构建树中，使用标准的ＵNIX-style make 程序以便通过all target构建工程 和通过install target 安装整个工程；对于工程的每个子目录，都会对应一个ｍakefile被创建，包含一下targets:

all 子目录所依赖的target

install 如果有的话　子目录运行安装

install/strip 如果有的话　在子目录中运行紧跟ＣMAKE\_STRIP 命令的安装；CMAKE\_ S TRIP变量会包含平台的瘦身实体，即移除了符号信息的二进制；

test 如果有的话　子目录中运行测试步骤

package 如果有的话　子目录中的打包步骤

#### Ninja Generators

### IDE 构建工具产生器

Ｖisual Studio Generaors

Other Generaors

## cpack-genrators

### CPack Archive Generator

该打包产生器用于打包files 成一个archives(压缩包);支持一下格式：

TGZ (.tar.gz)

ZIP(.zip)

...

当该打包产生器从CPackSourceConfig.cmake 文件中调用的时候，会产生包含工程 目录中除标记为ＣPACK\_SOURCE\_IGNORE\_FILES的文件之外的所有文件的压缩包； 下面是一个例子

set(CPACK\_SOURCE\_GENERATOR "TGZ")

set(CPACK\_SOURCE\_IGNORE\_FILES

\\.git/

build/

".\*~$"

)

set(CPACK\_VERBATIM\_VARIABLES YES)

include(CPack)

当打包产生器从ＣPackConfig.cmake文件中调用，会包含通过install()命令安装的 所有文件压缩包；

#### 常用变量对于压缩包打包器

CPACK\_ARCHIVE\_FILE\_NAME

CPACK\_ARCHIVE\_<component>\_FILE\_NAME

没有后缀的打包文件名称，后缀决定于压缩包的格式，自动添加到文件名字后面；默认是< CPACK\_PACKAGE\_FILE\_NAME>[-<component>],通过‘－’分割；

CPACK\_ARCHIVE\_COMPONENT\_INSTALL

使组件能够打包；在开启的情况下，压缩包打包器会产生多个包；默认是关闭的，意味着一个包包含所有的组件；

# camke-modules

## cpack

cpack模块是CMake发行版本的一部分，可以产生CPackConfig.cmake和CPackResourceConfig.cmake这些配置文件；这些配置文件用于随后运行的cpack项目installer和source packages的产生；

依赖于CMake genrator,CPack模块可以添加两种不同的构建target,即package和package\_source;

产生的二进制产生器会包含通过CMake的install()命令产生的二进制文件；需要注意的是，install()命令必须使用相对路径；否则会被CPack忽略这些安装的文件；特定二进制安装器Installer可以通过配置，以便使用者可以选择特定的应用组件来安装；可以参考CPackComponent;

source packages(通过CPackSourceConfig.cmake配置和CPack Archive Generator产生)会包含除了CPACK\_SOURCE\_IGNORE\_FILES意外的所有工程路径；

### CPack Generators

CPACK\_GENETATOR变量在不同的上下文中有不同的意义；

1. 在CMakeLists.txt文件中,CPACK\_GENERATOR是generators的列表；并且当cpack无参数运行时候，会递归列表，为每一个generator产生对应一个package;
2. 在CPACK\_PROJECT\_CONFIG\_FILE中， CPACK\_GENERATOR是命名单个Gene rator的字符串；如果你需要每个Cpack Generator逻辑以便控制其他Ｃpack设置，你需要一个ＣAPCK\_PROJECT\_CONFIG\_FILE;如果配置的情况下，在每个生成器的基础上自动包含ＣPACK\_PROJECT\_CO NFIG\_FILE文件；它只需要包含重写；下面是它工作原理：

a. cpack运行

b. 包含CPackConfig.cmake

c. 递归通过-G 命令选项配置产生的Geneartors；若是没有设置，递归在CPackConfig.c make文件中设置的CPACK\_GENEATOR变量产生的Generators;

d.对于每一个Generator,设置CPACK\_GENERATOR到当前正常递归的，包含CPACK\_PR OJECT\_CONFIG\_FILE, 产生generator对应的包；

在CPackConfig.cmake中的CPACK\_GENERATOR变量中的每一个GENERATOR,cp ack 会内部重新设置CPACK\_GENERATOR到正在使用的genearator,并且包含CPACK\_PROJECT\_CONFIG\_FILE;

### Target package and package\_source

如果CMAKE配合Makefile，Ninja 或者XCode geneartor使用，然后include(CPack)会产生一个target package; 使用者可以通过cmake --build . --target package 或者 make package 或者ninja package调用，这使从CMake,Make,或者Ninja代替cpack构建二进制安装器成为可能；

如果CMake 配合Makefile 或者 Ninja Generator,通过include(CPack)可以产生target package\_source.为了构建source package,为了代替cpack -G TGZ --config CPackSourceConfig.cmake,使用者可以调用cmake --build . --target package\_source ,make package\_source 或者ninja package\_source.

### CPack Generators常用的变量

在CMakeLists.txt文件中include(cpack)之前可以设置一下变量

CPACK\_PACKAGE\_NAME 默认是工程名字

CPACK\_PACKAGE\_VENDOR//供应商 默认是Humanity(人类）

CPACK\_PACKAGE\_DIRECTORY

cpack 将要打包的路径；如果没有设置的话，默认是build dir;可以定义在cpack co nfig file 或者通过命令行 cpack -B ;

CPACK\_PACKAGE\_VERSION\_MAJOR

CPACK\_PACKAGE\_VERSION\_MINOR

CPACK\_PACKAGE\_VERSION\_PATCH

CPACK\_PACKAGE\_DESCIPTION

CPACK\_PACKAGE\_DESCIPTION\_FILE

CPACK\_PACKAGE\_DESCIPTION\_SUMMARY

CPACK\_PACKAGE\_HOMEPAGE\_UR 不设置的情况下默认为空

CPACK\_PACKAGE\_FILE\_NAME

默认情况下是${CPACK\_PACKAGE\_NAME}-${CPACK\_PACKAGE\_VERSION} -${CPACK\_SYSTEM\_NAME}

CPACK\_PACKAGE\_INSTALL\_DIRECTORY

目标系统上的安装目录，在安装前缀下（installation prefix），可以使用CPack generators 类似于NSIS 创建安装目录；

CPACK\_PACKAGE\_ICON

CPACK\_PACKAGE\_CHECKSUM

会产生额外的文件已检查package的checksum的算法 ;输出文件名字为${CPAC K\_ PACKAGE\_FILE\_NAME}.${CPACK\_PACKAGE\_CHECKSUM}

CPACK\_PROJECT\_CONFIG\_FILE

CPack-time 工程 CPack 配置文件。该文件在cpack time被包含，在CPack设置 CPA CK\_GENERATOR到实际使用的generator之后，每个生成器包含该文件；

CPACK\_RESOUCE\_FILE\_LICENSE

CPACK\_RESOURCE\_FILE\_README

CPACK\_RESOURCE\_FILE\_WELCOME

CPACK\_MONOLITHIC\_INSTALL //一体化安装

禁止基于组件安装的机制。当被设置的情况下，组件化规范设置被忽略，所有安装项目 被放置在一个pacakge内；一些CPack generators 默认是一体化安装并且可以通 过设置CPACK\_<GENNAME>\_COMPONENT\_INSTALL 为true 来进行组件化安 装；

CPACK\_GENETATOR

列出要使用的CPack generators;如果不指定，CPack 会创建一系统遵从命名范式 (CPACK\_BINARY\_<GENNAME>)选项,以便用户允许或者禁止特定的generator;如 果命令行cpack -G选项设置，可以重写CPACK\_BINARY\_<GENNAME>选项；

常用的Geneator有Unix Makefiles,Ninja等等，CMAKE\_GENERATOR的值不应 该由工程代码改变，可以通过命令行cmake -G 或者CMAKE\_GENRATOR 环境变量 设置；

CPACK\_OUTPUT\_CONFIG\_FILE

默认是CPackConfig.cmake文件

CPACK\_PACKAGE\_EXECUTABLES

列出所有的可执行文件和相关的text 标签以便创建开始菜单快捷键

CPACK\_STRIP\_FILES//被省略的文件

CPACK\_VERBATIM\_VARIABLES //保持原义

如果设置为true,以CPACK\_ 为前缀的变量,在写入配置文件之前会被转义；如果不 设置，类似于双引号和反斜杠会引起错误或者cpack收到的值是改变的；为了向后兼 容性默认是false;

CPACK\_SOURCE\_PACAKGE\_FILE\_NAME

CPACK\_SOURCE\_STRIP\_FILES

CPACK\_SOURCE\_GENERATOR

CPACK\_SOURCE\_OUTPUT\_CONFIG\_FILE

默认是CPackSourceConfig.cmake

CPACK\_SOURCE\_IGHORE\_FILES

## cpack-component

该模块自动包含在CPack 模块中；Cpack生成的安装器（尤其是图形化安装器）允许用户选择特定的组件进行安装；这个模块允许开发者自己配置这样的组件打包；

通过install()命令中COMPONENT来设置组件的内容，可以使用友好的名称，描述和内部依赖来标明组件；使用下面的命令通过分组自定义安装器；

为了不同的CPackGenerator指定不同的分组，使用CPACK\_PROJECT\_CONFIG\_FILE;

### 变量

CPACK\_COMPONENTS\_ALL 要安装的组件列表；默认的值由CPack计算出来和工程包含的所有组件；用户可以自己设置只包含特定的组件；get\_cmake\_property()命令可以获得CPACK\_COMPONENTS\_ALL变量的COMPONENTS属性，list(REMOVE\_ITEM)命令可以去除不想使用的组件；

get\_cmake\_property(CPACK\_COMPONENTS\_ALL COMPONENTS)

list(REMOVE\_ITEM CPACK\_COMPONENTS\_ALL "foo" "bar")

CPack\_<GENNAME>\_COMPONENT\_INSTALL 每一个CPack Generator(RPM, DEB，ＡRCHIVE...)都有一个默认的行为（一体化安装还是组件化安装）；例如RPM这种ＣPack Generator对应的行为是一体式构建;可以通过设置变量的值为０或者１（OFF/ON)改变这种行为；

CPACK\_COMPONENTS\_GROUPING对于要多个包组件化安装的CPack Generator,指定组件如何被分组，主要依赖于CPACK\_COMPONENTS\_GROUPING的值：

　 ONE\_PER\_GROUP(为每一个组件分组创建一个包)；

IGNORE(每个组件设置一个包）

ALL\_COMPONETS\_IS\_ONE(所有选择的组件打成一个包)

CPACK\_COMPONENT\_<componentName>\_DISPLAY\_NAME

CPACK\_COMPONENT\_<componentName>\_DESCRIPTION

CPACK\_COMPONENT\_<componentName>\_GROUP

该组件是哪个组的

CPACK\_COMPONENT\_<componentName>\_DEPENDS

CPACK\_COMPONENT\_<componentName>\_HIDDEN

设置为true就是用户不可感知的

CPACK\_COMPONENT\_<componentName>\_REQUIRED

设置为true就是该组件是不可缺少的

CPACK\_COMPONENT\_<componentName>\_DISABLED

设置为true的情况下默认不被选择安装

### 命令

cpack\_configure\_downloads(site

[UPLOAD\_DIRECTORY dirname]

[ALL]

[ADD\_REMOVE|NO\_ADD\_REMOVE])

该命令配置了安装时期已选择组件的下载；对于每一个可下载的组件，CPack会创建包含该组件所有文件的压缩包，并且会上传到提供的site;当用户选择那个组件安装的时候，安装器会下载并且解压缩；节省带宽的情况下，创建只下载已选择的组件的安装器这个特性非常有用；

UPLOAD\_DIRECTORY 表示ｃpack 为每个组件产生的压缩包存储的路径；如果不设置ＣPack 会使用Cmake 二进制目录路径来存储产生的压缩包；

Ａll 所有的组件被下载，否则只有指定的标记为Downloaded的组件或者拥有压缩包的组件才会下载；All还暗示着ADD\_REMOVE;

ADD\_REMOVE 暗示Cpack会安装这样一个安装器的副本，这个安装器通过调用

Windows的Add/Remove程序对话框来改变已安装组件的；（MacOsＸ　这个系统忽略这个设置）

cpack\_add\_install\_type(typename

[DISPLAY\_NAME name])

对于图形化安装器，添加包含一系列预先选择组件的新安装类型；该命令标识了一个应用常见使用所需要的预选的组件；一个组件可以通过cpack\_add\_component()命令中的INSTALL\_TYPES这个选项表示是那些安装类型；

cpack\_add\_component(componentName

[DISPLAY\_NAME name]

[DESCRIPTION decscription]

[HIDDEN|REQURIED|DISABLED]

[GROUP group]

[DEPENDS comp1 comp2 ...]

[INSTALL\_TYPES type1 type2 ...]

[DOWLOADED]

[ARCHIVE\_FILE fileName]

[PLIST filename])

DOWNLOAD　选项表明该组件直接下载而非不是由安装器打包的，可以通过cpack\_confi

gure\_downloads()这个命令指定下载网址

ARCHIVE\_FILE 为cpack 下载的组件创建压缩包指定文件名称

PLIST选项　当使用使用构建产物的generator的时候作为--component-plist的参数，传给pkgbuild

cpack\_add\_component\_group(groupname

[DISPLAY\_NAME name]

[DESCRIPTION description]

[PARENT\_GROUP parent]

[EXPANDED]

[BOLD\_TITLE])

PARENT\_GROUP 主要为了建立层级关系

EXPAND 默认会设置该选项，用于用户可以看见该分组内的所有components;

BOLD\_TITLE 标题粗写

## CMakePackageConfigHelpers

该模块用于创建xxxConfig.cmake文件，帮组其他工程找到和使用package;该模块下包含两个命令config\_package\_config\_file()和write\_basic\_package\_version\_file();

configure\_package\_config\_file(

<input> <output>

INSTALL\_DESTINATION <path>

[PATH\_VARS <var1><var2>...<varN>]

[NO\_SET\_AND\_CHECK\_MACRO]

[NO\_CHECK\_REQUIRED\_COMPONENTS\_MARCRO]

[INSTALL\_PREFIX <path>])

## FindPkgConfig

该模块用于寻找pkg-config可执行文件，找到的情况下可以使用pkg\_get\_variable(),

pkg\_check\_modules()和pkg\_search\_module()这些命令；以下命令便会设置PKG \_CONFIG\_FOUND,PKG\_CONFIG\_EXCUTABLE,PKG\_CONFIG\_VERSION\_STRING;

pkg\_check\_modules(<prefix> [REQUIRED][QUIET][NO\_MAKE\_PATH]

[NO\_CMAKE\_ENVIRONMENT\_PATH]

[IMPORTED\_TARGET [GLOBAL]]

<moduleSpec>[<moduleSpec>...])

在3.1以及之后CMAKE\_PREFIX\_PATH,CMAKE\_FRAMEWORK\_PATH和

CMAKE\_APPBUNDLE\_PATH 缓存和环境变量会添加到pkg-config的搜索路径；

IMPORTED\_TARGET被设置的时候，会创建一个PkgConfig::<prefix>target，可以用于target\_link\_libraries()的参数，GLOBAL标识着作为全局的；<moduleSpec>可以是foo,foo<2,foo>=3.1多种形式；

以下变量会被返回：

xxx\_FOUND;

如果module存在的情况下，返回为1;

XXX\_LIBRARIES;

XXX\_LINK\_LIBRARIES;

XXX\_LIBRARY\_DIRS;

XXX\_INCLUDE\_DIRS;

pkg\_get\_variable(<resultVar> <moduleName> <varName>

pkg\_search\_module(<prefix> [REQUIRED][QUIET]

[NO\_CMAKE\_PATH]

[NO\_CAKE\_ENVIRAONMENT\_PATH]

[IMPORTED\_TARGET [GLOBAL]]

<moudleSpec [<moduleSpec>...])

该命令和pkg\_check\_modules()作用一样，只是仅匹配找到的第一个module;

# cmake-toolchains

CMake使用一系列的工具链来编译 链接库和创建压缩包和驱动构建的其它任务；可用的工具链可支持的语言决定；在正常的构建过程中，基于系统自省和默认设置，CMake自动决定构建的工具链；在交叉编译场景下，一个工具链文件指定编译器和工具的路径这些信息；

## Language

language 可以通过project()这个命令来设置，语言特定的内置变量例如CMAKE\_CXX\_ COMPILER, CMAKE\_CXX\_COMILER\_ID 来设置通过project()命令；如果在工程目录下CMakeLists.txt 文件中没有定义，会悄悄产生一个；默认支持的语言是c或者cxx

project(C\_ONLY C)

或者

project(<project name> NONE)

enable\_language(CXX)

ENABLED\_LANGUAGES 这个全局变量包含当前支持的语言；

CMAKE\_<LANG>\_COMPILER是<LANG>对应的编译器的全路径；

CMAKE\_<LANG>\_COMPILER\_ID是CMake使用的编译器的标识id;

CMAKE\_<LANG>\_COMPILER\_VERSION 是CMake使用的编译器的版本号；

当编译特定语言的文件，CMAKE\_<LANG>\_FLAGS 变量和含有flags特定配置会被添加到编译命令；

和链接器被编译器的驱动器调用一样，CMake需要一种方式决定那个编译器调用链接器；这个被在target中源文件的语言计算出来；对于静态库，取决于依赖库的语言；

## CMAKE\_<LANG>\_COMPILER\_ID

编译器身份字符串；常见有一下：

Clang = LLVM Clang (clang.llvm.org)

GNU = GNU Compiler Collection (gcc.gnu.org)

Intel = Intel Compiler (intel.com)

...

## 交叉编译

Linux平台交叉编译

set(CMAKE\_SYSTEM\_NAME Linux)

set(CMAKE\_SYSTEM\_PROCESSOR arm)

set(CMAKE\_SYSROOT /home/devel/rasp-pi-rootfs)

set(CMAKE\_STAGING\_PREFIX /home/devel/stage)

set(tools /home/devel/gcc-4.7-linaro-rpi-gnueabihf)

set(CMAKE\_C\_COMPILER ${tools}/bin/arm-linux-gnueabihf-gcc)

set(CMAKE\_CXX\_COMPILER ${tools}/bin/arm-linux-gnueabihf-g++)

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(CMAKE\_FIND\_ROOT\_PATH\_MODE\_PACKAGE ONLY)

# cmake-packages

## Packages 的使用

Package 提供给cmake基于构建系统的依赖信息；搜索包使用find\_package()命令，该命令的结果要么是一组ＩMPORTED targets或者相应相关构建信息的变量；

cmake提供两种形式的包直接支持，分别是Ｃonfig-file Packages 和Ｆind-module packages;　通过findPkgConfig 模块提供pkg-config 包的间接支持；在所有的情况下和find\_packages()的调用一样；

# CMake provides a Qt4 find-module

find\_package(Qt4 4.7.0 REQUIRED)

# Qt provides a Qt5 package config file.

find\_package(Qt5Core 5.1.0 REQUIRED)

# Use pkg-config via the LibXml2 find-module

find\_package(LibXml2 REQUIRED)

在某些情况下，已经知道一个package configuration file 被提供，通过CONFIG关键字可以传递给find\_package()命令：

find\_package(Qt5Core 5.1.0 CONFIG REQUIRED)

find\_package(Qt5Gui 5.1.0 CONFIG)

ＭODULE类似的操作

find\_package(Qt4 4.7.0 MODULE REQUIRED)

准确地指定package的类型，当包找不到的时候可以展示错误信息；

find\_package(Qt5 5.1.0 CONFIG REQUIRED Widgets Xml Sql)

或者通过package定义的COMPONENTS 和OPTIONAL\_COMPONENTS

find\_package(Qt5 5.1.0 COMPONENTS Widgets

OPTIONAL\_COMPONENTS Xml Sql

)

CMAKE\_DISABLE\_FIND\_PACKAGE\_<packageName> 为true可以保证packageN ame对应的包无法被搜索到；

## Config-file Packages

一个config-file的包包含pacakge configuration file 和package version file;

看下面Foo的例子

<prefix>/include/foo-1.2/foo.h

<prefix>/lib/foo-1.2/libfoo.a

<prefix>/lib/cmake/foo-1.2/FooConfig.cmake

## Creating Packages

project(UpstreamLib)

set(CMAKE\_INCLUDE\_CURRENT\_DIR ON)

set(CMAKE\_INCLUDE\_CURRENT\_DIR\_IN\_INTERFACE ON)

set(Upstream\_VERSION 3.4.1)

#GenerateExportHeader 是一个 module

include(GenerateExportHeader)

add\_library(ClimbingStats SHARED climbingstats.cpp)

generate\_export\_header(ClimbingStats)

set\_property(TARGET ClimbingStats PROPERTY VERSION

${Upstream \_VERSION})

set\_property(TARGET ClimbingStats PROPERTY SOVERSION 3)

set\_property(TARGET ClimbingStats PROPERTY

INTERFACE\_ClimbingStats\_MAJOR\_VERSION 3)

set\_property(TARGET ClimbingStats APPEND PROPERTY

COMPATIBLE\_INTERFACE\_STRING ClimbingStats\_MAJOR\_VERSION)

install(TARGETS ClimbingStats EXPORT ClimbingStatsTargets

LIBRARY DESTINATION lib

ARCHIVE DESTINATION lib

RUNTIME DESTINATION bin

INCLUDES DESTINATION include

)

install(

FILES

climbingstats.h

"${CMAKE\_CURRENT\_BINARY\_DIR}/climbingstats\_export.h"

DESTINATION

include

COMPONENT

Devel

)

#CMakePacakgeConfigHelpers　这个Module提供了创建一个

#ConfigVersion.cmake文件的宏，这个文件用来设置package的版本号

＃find\_package()命令可以搜索版本匹配的package,并且设置一些类似

＃<PackageName>\_VERSION, <PackageName>\_VERSION\_MAJOR

#<PackageName>\_VERSION\_MINOR 这些变量等等；

include(CMakePackageConfigHelpers)

write\_basic\_package\_version\_file(

"${CMAKE\_CURRENT\_BINARY\_DIR}/ClimbingStats/ClimbingStatsConfigVer sion.cmake"

VERSION ${Upstream\_VERSION}

COMPATIBILITY AnyNewerVersion

)

export(EXPORT ClimbingStatsTargets

FILE "${CMAKE\_CURRENT\_BINARY\_DIR}/ClimbingStats/

ClimbingStatsTargets.cmake"

NAMESPACE Upstream::

)

configure\_file(cmake/ClimbingStatsConfig.cmake

"${CMAKE\_CURRENT\_BINARY\_DIR}/ClimbingStats/

ClimbingStatsConfig.cmake"

COPYONLY)

set(ConfigPackageLocation lib/cmake/ClimbingStats)

install(EXPORT ClimbingStatsTargets

FILE

ClimbingStatsTargets.cmake

NAMESPACE

Upstream::

DESTINATION

${ConfigPackageLocation})

install(

FILES

cmake/ClimbingStatsConfig.cmake

"${CMAKE\_CURRENT\_BINARY\_DIR}/ClimbingStats/

ClimbingStatsConfigVersion.cmake"

DESTINATION

${ConfigPackageLocation}

COMPONENT

Devel

)

# cmake-developer

## find\_modules

每一次搜索模块是通过find\_package(<PackageName>)命令定位到的Find<Packag eName>.cmake文件被加载；find\_module的首要任务是决定一个包是否在系统中存在，通过<PackageName> \_FOUND变量反映这个结果；

当upstream库没有提供config file package的时候find\_module非常有用；

传统的方式是使用变量（包括library 和excutable);可以参考下面的标准变量来；更现代的方法是通过提供imported target表现的 像config file packages一样,它的优势是可以传递使用条件；无论使用那种方式或者二者组合应考虑向后兼容性；

看下面这个例子

find\_package(Foo [major[.minor[.patch[.tweak]]]]

[EXACT] [QUIET] [REQUIRED]

[[COMPONENTS] [components...]]

[OPTIONAL\_COMPONENTS components...]

[NO\_POLICY\_SCOPE])

可以通过find\_package的文档详细地了解关于find\_module的变量设置；大部分通过使用FindPackageHandleStandardArgs 处理这些变量；

简而言之，模块应该是包的版本号与要求的兼容；

### 标准变量的名字

对于一个FindXXX.cmake文件所对应的模块，

Xxx\_INCLUDE\_DIRS

Xxx\_INCLUDE\_LIBRARIES

Xxx\_DEFINITIONS

当编译代码使用Xxx的时候的一些定义；

Xxx\_EXCUTABLE

Xxx\_Yyy\_EXCUTABLE

Xxx\_ROOT\_DIR

## END