

# 课件-CMake实战

---

## 重点内容

### 安装cmake

#### 1 安装 cmake

1.1 卸载已经安装的旧版的CMake[非必需]

1.2 文件下载解压:

1.3 创建软链接

#### 2 单个文件目录实现

##### 2.1 基本工程

语法: PROJECT

语法: SET

语法: MESSAGE

语法: ADD\_EXECUTABLE

##### 2.2 改进工程结构

语法: DCMAKE\_INSTALL\_PREFIX

语法: ADD\_SUBDIRECTORY

语法: INSTALL 第6章节详细讲解

#### 3 多个目录实现

##### 3.1 子目录编译成库文件

语法: INCLUDE\_DIRECTORIES

语法: ADD\_SUBDIRECTORY

语法: ADD\_LIBRARY

语法: TARGET\_LINK\_LIBRARIES

##### 3.2 子目录使用源码编译

语法: AUX\_SOURCE\_DIRECTORY

#### 4 生成库

##### 4.1 生成动态库

##### 4.2 生成静态库+安装到指定目录

#### 5 调用库

5.1 调用静态库

5.2 调用动态库

## 6 设置安装目录

6.1 install命令

目标文件TARGETS 的安装

普通文件的安装

目录的安装

安装时脚本的运行

## 7 设置执行目录+编译debug和release版本

7.1 编译debug版本release版本

语法: PROJECT\_SOURCE\_DIR

7.2 编译选项

## 8 跨平台

参考

零声学院 Darren 326873713

C/C++Linux服务器开发/高级架构师 <https://ke.qq.com/course/420945?tuin=137bb271>

# 重点内容

## 安装cmake

- 单个目录实现
- 多个目录实现
- 生成静态库
- 生成动态库
- 调用静态库
- 调用动态库
- 设置执行目录
- 设置安装目录
- 编译debug和release版本

官方文档: <https://cmake.org/cmake/help/v3.21/>

# 1 安装 cmake

可以下载更新的版本：

[https://github.com/Kitware/CMake/releases/download/v3.21.4/cmake-3.21.4-linux-x86\\_64.tar.gz](https://github.com/Kitware/CMake/releases/download/v3.21.4/cmake-3.21.4-linux-x86_64.tar.gz)

## 1.1 卸载已经安装的旧版的CMake[非必需]

```
1 apt-get autoremove cmake
```

Bash

复制代码

## 1.2 文件下载解压：

```
1 wget https://cmake.org/files/v3.9/cmake-3.9.1-Linux-x86_64.tar.gz
```

Bash

复制代码

解压：

```
1 tar zxvf cmake-3.9.1-Linux-x86_64.tar.gz
```

Bash

复制代码

查看解压后目录：

```
1 tree -L 2 cmake-3.9.1-Linux-x86_64
2 cmake-3.9.1-Linux-x86_64
3 |  bin
4 |  |  cmake
5 |  |  cmake
6 |  |  cmake-gui
7 |  |  cpack
8 |  |  ctest
9 |  doc
10 |  |  cmake
11 |  man
12 |  |  man1
13 |  |  man7
14 |  share
15 |  |  aclocal
16 |  |  applications
17 |  |  cmake-3.9
18 |  |  icons
19 |  |  mime
20 12 directories, 5 files
```

Bash

复制代码

bin下面有各种cmake家族的产品程序.

## 1.3 创建软链接

注: 文件路径是可以指定的, 一般选择在 `/opt` 或 `/usr` 路径下, 这里选择 `/opt`

Bash 复制代码

```
1 mv cmake-3.9.1-Linux-x86_64 /opt/cmake-3.9.1
2 ln -sf /opt/cmake-3.9.1/bin/* /usr/bin/
```

## 2 单个文件目录实现

### 2.1 基本工程

Makefile 复制代码

```
1 # 单个目录实现
2 # CMake 最低版本号要求
3 cmake_minimum_required (VERSION 2.8)
4 # 手动加入文件
5 SET(SRC_LIST main.c)
6 MESSAGE(STATUS "THIS IS BINARY DIR " ${PROJECT_BINARY_DIR})
7 MESSAGE(STATUS "THIS IS SOURCE DIR " ${PROJECT_SOURCE_DIR})
8 ADD_EXECUTABLE(0voice ${SRC_LIST})
```

参考: [src-cmake/2.1-1](#)

### 语法: PROJECT

指令	PROJECT
语法	PROJECT(projectname [CXX] [C] [Java])
说明	用于指定工程名称, 并可指定工程支持的语言 (支持的语言列表可以忽略, 默认支持所有语言)。这个指令隐式的定义了两个cmake变量: <projectname>_BINARY_DIR 和 <projectname>_SOURCE_DIR。cmake帮我们预定义PROJECT_BINARY_DIR和PROJECT_SOURCE_DIR变量。建议使用这两个变量, 即使修改了工程名称, 也不会影响这两个变量。如果使用了<projectname>_SOURCE_DIR, 修改工程名称后, 需要同时修改这些变量。

### 语法: SET

指令	SET
语法	SET(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]])
说明	SET 指令可以用来显式的定义变量，比如SET(SRC_LIST main.c)。如果有多个源文件,也可以定义成SET(SRC_LIST main.c t1.c t2.c)。

## 语法：MESSAGE

指令	MESSAGE
语法	MESSAGE([SEND_ERROR   STATUS   FATAL_ERROR] "message to display" ...)
说明	这个指令用于向终端输出用户定义的信息，它包含了三种类型： SEND_ERROR：产生错误，生成过程被跳过 STATUS：输出前缀为-的信息。 FATAL_ERROR：立即终止所有cmake过程。

## 语法：ADD\_EXECUTABLE

指令	ADD_EXECUTABLE
语法	ADD_EXECUTABLE([BINARY] [SOURCE_LIST])
说明	定义了这个工程会生成一个文件名为[BINARY]可执行文件，相关的源文件是SOURCE_LIST 中定义的源文件列表

## 2.2 改进工程结构

```

1  .
2  ├── build
3  ├── CMakeLists.txt
4  ├── doc
5  │   ├── darren.txt
6  │   └── README.MD
7  └── src
8     ├── CMakeLists.txt
9     └── main.c

```

工程： `src-cmake/2.2-1`

该工程实现更为简洁的工程目录。

```
cmake -DCMAKE_INSTALL_PREFIX=/tmp/usr ..
```

顶层CMakeLists.txt

```

1
2  # CMake 最低版本号要求
3  cmake_minimum_required (VERSION 2.8)
4  PROJECT(VOICE)
5
6  # 添加子目录
7  ADD_SUBDIRECTORY(src)
8
9  #INSTALL(FILES COPYRIGHT README DESTINATION share/doc/cmake/voice)
10 # 安装doc到 share/doc/cmake/voice目录
11 # 默认/usr/local/
12 #指定自定义目录, 比如 cmake -DCMAKE_INSTALL_PREFIX=/tmp/usr ..
13 INSTALL(DIRECTORY doc/ DESTINATION share/doc/cmake/voice)

```

## 语法：DCMAKE\_INSTALL\_PREFIX

cmake时传递 安装目录，比如 `cmake -DCMAKE_INSTALL_PREFIX=/tmp/usr ..`

## 语法：ADD\_SUBDIRECTORY

其中：

指令	ADD_SUBDIRECTORY
语法	ADD_SUBDIRECTORY(source_dir [binary_dir] [EXCLUDE_FROM_ALL])
说明	此指令用于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。EXCLUDE_FROM_ALL参数的含义是将这个目录从编译过程中排除，比如，工程的example，可能就需要工程构建完成后，再进入example目录单独进行构建(当然，你也可以通过定义依赖来解决此类问题)

语法：INSTALL 第6章节详细讲解

INSTALL指令用于定义安装规则，安装的内容可以包括目标二进制、动态库、静态库以及文件、目录、脚本等。INSTALL指令包含了各种安装类型，我们需要一个个分开解释：

类型	目标文件
指令	INSTALL
语法	INSTALL( <b>TARGETS</b> targets... [[ARCHIVE LIBRARY RUNTIME] [DESTINATION <dir>] [PERMISSIONS permissions...] [CONFIGURATIONS [Debug Release ...]] [COMPONENT <component>] [OPTIONAL] ] [...])
说明	参数中的TARGETS后面跟的就是我们通过ADD_EXECUTABLE或者ADD_LIBRARY定义的目标文件，可能是可执行二进制、动态库、静态库。目标类型也就相对应的有三种，ARCHIVE特指静态库，LIBRARY特指动态库，RUNTIME特指可执行目标二进制。DESTINATION定义了安装的路径。

类型	普通文件
指令	INSTALL
语法	INSTALL( <b>FILES</b> files... DESTINATION <dir> [PERMISSIONS permissions...] [CONFIGURATIONS [Debug Release ...]] [COMPONENT <component>] [RENAME <name>] [OPTIONAL])
说明	可用于安装一般文件，并可以指定访问权限，文件名是此指令所在路径下的相对路径。如果默认不定义权限 PERMISSIONS，安装后的权限为：OWNER_WRITE, OWNER_READ, GROUP_READ,和WORLD_READ，即644权限。
类型	非目标文件的可执行程序（如脚本之类）
指令	INSTALL
语法	INSTALL( <b>PROGRAMS</b> files... DESTINATION <dir> [PERMISSIONS permissions...] [CONFIGURATIONS [Debug Release ...]] [COMPONENT <component>] [RENAME <name>] [OPTIONAL])
说明	跟上面的FILES指令使用方法一样，唯一的不同是安装后权限为:OWNER_EXECUTE, GROUP_EXECUTE, 和WORLD_EXECUTE，即755权限。



类型	目录
指令	INSTALL
语法	INSTALL( <b>DIRECTORY</b> dirs... DESTINATION <dir> [FILE_PERMISSIONS permissions...] [DIRECTORY_PERMISSIONS permissions...] [USE_SOURCE_PERMISSIONS] [CONFIGURATIONS [Debug Release ...]] [COMPONENT <component>] [[PATTERN <pattern>   REGEX <regex>] [EXCLUDE] [PERMISSIONS permissions...]] [...])
说明	主要介绍其中的DIRECTORY、PATTERN和PERMISSIONS参数： DIRECTORY：后面连接的是所在Source目录的相对路径。 PATTERN：用于使用正则表达式进行过滤。 PERMISSIONS：用于指定PATTERN过滤后的文件权限。

## 3 多个目录实现

### 3.1 子目录编译成库文件

工程：3.1-1

Basic | 复制代码

```
1  ├── CMakeLists.txt
2  ├── doc
3  │   ├── darren.txt
4  │   └── README.MD
5  └── src
      ├── CMakeLists.txt
      ├── dir1
      │   ├── CMakeLists.txt
      │   ├── dir1.c
      │   └── dir1.h
      ├── dir2
      │   ├── CMakeLists.txt
      │   ├── dir2.c
      │   └── dir2.h
      └── main.c
```

## 语法：INCLUDE\_DIRECTORIES

找头文件

```
INCLUDE_DIRECTORIES("${CMAKE_CURRENT_SOURCE_DIR}/dir1")
```

## 语法：ADD\_SUBDIRECTORY

添加子目录

```
ADD_SUBDIRECTORY("${CMAKE_CURRENT_SOURCE_DIR}/dir1")
```

## 语法：ADD\_LIBRARY

ADD\_LIBRARY( hello\_shared SHARED libHelloSLAM.cpp ) # 生成动态库

ADD\_LIBRARY( hello\_shared **STATIC** libHelloSLAM.cpp ) 生成静态库

## 语法：TARGET\_LINK\_LIBRARIES

链接库到执行文件上

```
TARGET_LINK_LIBRARIES(darren dir1 dir2)
```

## 3.2 子目录使用源码编译

工程 3.2-1



src

```
1  # 单个目录实现
2  # CMake 最低版本号要求
3  cmake_minimum_required (VERSION 2.8)
4  # 工程
5  PROJECT(0VOICE)
6  # 手动加入文件
7  SET(SRC_LIST main.c)
8  MESSAGE(STATUS "THIS IS BINARY DIR " ${PROJECT_BINARY_DIR})
9  MESSAGE(STATUS "THIS IS SOURCE DIR " ${PROJECT_SOURCE_DIR})
10
11 #设置子目录
12 set(SUB_DIR_LIST "${CMAKE_CURRENT_SOURCE_DIR}/dir1"
    "${CMAKE_CURRENT_SOURCE_DIR}/dir2")
13
14 foreach(SUB_DIR ${SUB_DIR_LIST})
15     #遍历源文件
16     aux_source_directory(${SUB_DIR} SRC_LIST)
17 endforeach()
18
19 # 添加头文件路径
20 INCLUDE_DIRECTORIES("dir1")
21 INCLUDE_DIRECTORIES("dir2")
22
23
24 ADD_EXECUTABLE(darren ${SRC_LIST} )
25
26
27 # 将执行文件安装到bin目录
28 INSTALL(TARGETS darren RUNTIME DESTINATION bin)
```

## 语法：AUX\_SOURCE\_DIRECTORY

找在某个路径下的所有源文件

`aux_source_directory(<dir> <variable>)`

## 4 生成库

### 4.1 生成动态库

工程4.1

```
1  # 设置release版本还是debug版本
2  if(${CMAKE_BUILD_TYPE} MATCHES "Release")
3      MESSAGE(STATUS "Release版本")
4      SET(BuildType "Release")
5  else()
6      SET(BuildType "Debug")
7      MESSAGE(STATUS "Debug版本")
8  endif()
9
10 #设置lib库目录
11 SET(RELEASE_DIR ${PROJECT_SOURCE_DIR}/release)
12 # debug和release版本目录不一样
13 #设置生成的so动态库最后输出的路径
14 SET(LIBRARY_OUTPUT_PATH ${RELEASE_DIR}/linux/${BuildType})
15 # -fPIC 动态库必须的选项
16 ADD_COMPILE_OPTIONS(-fPIC)
17
18 # 查找当前目录下的所有源文件
19 # 并将名称保存到 DIR_LIB_SRCS 变量
20 AUX_SOURCE_DIRECTORY(. DIR_LIB_SRCS)
21 # 生成静态库链接库Dir1
22 #ADD_LIBRARY (Dir1 ${DIR_LIB_SRCS})
23 # 生成动态库
24 ADD_LIBRARY (Dir1 SHARED ${DIR_LIB_SRCS})
```

PROJECT\_SOURCE\_DIR 跟着最近的工程的目录

## 4.2 生成静态库+安装到指定目录

工程4.2

```

1  # 设置release版本还是debug版本
2  if(${CMAKE_BUILD_TYPE} MATCHES "Release")
3      MESSAGE(STATUS "Release版本")
4      SET(BuildType "Release")
5  else()
6      SET(BuildType "Debug")
7      MESSAGE(STATUS "Debug版本")
8  endif()
9
10 #设置lib库目录
11 SET(RELEASE_DIR ${PROJECT_SOURCE_DIR}/release)
12 # debug和release版本目录不一样
13 #设置生成的so动态库最后输出的路径
14 SET(LIBRARY_OUTPUT_PATH ${RELEASE_DIR}/linux/${BuildType})
15 ADD_COMPILE_OPTIONS(-fPIC)
16
17 # 查找当前目录下的所有源文件
18 # 并将名称保存到 DIR_LIB_SRCS 变量
19 AUX_SOURCE_DIRECTORY(. DIR_LIB_SRCS)
20 # 生成静态库链接库Dir1
21 ADD_LIBRARY (Dir1 ${DIR_LIB_SRCS})
22 # 将库文件安装到lib目录
23 INSTALL(TARGETS Dir1 DESTINATION lib)
24 # 将头文件include
25 INSTALL(FILES dir1.h DESTINATION include)

```

## 编译安装

```

1  ubuntu% cmake -DCMAKE_INSTALL_PREFIX=/tmp/usr ..
2  ubuntu% make
3  ubuntu% make install
4  [100%] Built target Dir1
5  Install the project...
6  -- Install configuration: ""
7  -- Up-to-date: /tmp/usr/lib/libDir1.a
8  -- Installing: /tmp/usr/include/dir1.h

```


将静态库安装到lib，头文件安装到include

进一步参考：<http://www.mamicode.com/info-detail-2439626.html>

## 5 调用库

### 5.1 调用静态库


Basic

 复制代码

```
1  # CMake 最低版本号要求
2  cmake_minimum_required (VERSION 2.8)
3  # 工程
4  PROJECT(0VOICE)
5  # 手动加入文件
6  SET(SRC_LIST main.c)
7  MESSAGE(STATUS "THIS IS BINARY DIR " ${PROJECT_BINARY_DIR})
8  MESSAGE(STATUS "THIS IS SOURCE DIR " ${PROJECT_SOURCE_DIR})
9
10 INCLUDE_DIRECTORIES("${CMAKE_CURRENT_SOURCE_DIR}/lib")
11 # 库的路径
12 LINK_DIRECTORIES("${CMAKE_CURRENT_SOURCE_DIR}/lib")
13 # 生成执行文件
14 ADD_EXECUTABLE(darren ${SRC_LIST})
15 # 引用动态库
16 TARGET_LINK_LIBRARIES(darren Dir1)
```

### 5.2 调用动态库

Basic

 复制代码

```
1  # 单个目录实现
2  # CMake 最低版本号要求
3  cmake_minimum_required (VERSION 2.8)
4  # 工程
5  PROJECT(0VOICE)
6  # 手动加入文件
7  SET(SRC_LIST main.c)
8  MESSAGE(STATUS "THIS IS BINARY DIR " ${PROJECT_BINARY_DIR})
9  MESSAGE(STATUS "THIS IS SOURCE DIR " ${PROJECT_SOURCE_DIR})
10
11 INCLUDE_DIRECTORIES("${CMAKE_CURRENT_SOURCE_DIR}/lib")
12
13 LINK_DIRECTORIES("${CMAKE_CURRENT_SOURCE_DIR}/lib")
14 # 引用动态库
15 ADD_EXECUTABLE(darren ${SRC_LIST})
16 #同时静态库、动态库 优先连接动态库
17 #TARGET_LINK_LIBRARIES(darren Dir1)
18 # 强制使用静态库
19 TARGET_LINK_LIBRARIES(darren libDir1.a)
```

如果同时存在动态库和静态库，优先连接动态库。

强制静态库TARGET\_LINK\_LIBRARIES(darren libDir1.a)

## 6 设置安装目录

在cmake的时候，最常见的几个步骤就是：

```
1  mkdir build && cd build
2  cmake ..
3  make
4  make install
```

C++ | 复制代码

显然并不需要，作为一个经常需要被运行的指令，官方提供了一个命令install，只需要经过该命令的安装内容，不需要显示地定义install目标。此时，make install就是运行该命令的内容。

### 6.1 install命令

install用于指定在安装时运行的规则。它可以用来安装很多内容，可以包括目标二进制、动态库、静态库以及文件、目录、脚本等：

```
1  install(TARGETS <target>... [...])
2  install({FILES | PROGRAMS} <file>... [...])
3  install(DIRECTORY <dir>... [...])
4  install(SCRIPT <file> [...])
5  install(CODE <code> [...])
6  install(EXPORT <export-name> [...])
```

C++ | 复制代码

有时候，也会用到一个非常有用的变量CMAKE\_INSTALL\_PREFIX，用于指定cmake install时的相对地址前缀。用法如：

```
1  cmake -DCMAKE_INSTALL_PREFIX=/usr ..
```

C++ | 复制代码

### 目标文件TARGETS 的安装

```

1  install(TARGETS targets... [EXPORT <export-name>]
2      [[ARCHIVE|LIBRARY|RUNTIME|OBJECTS|FRAMEWORK|BUNDLE|
3      PRIVATE_HEADER|PUBLIC_HEADER|RESOURCE]
4      [DESTINATION <dir>]
5      [PERMISSIONS permissions...]
6      [CONFIGURATIONS [Debug|Release|...]]
7      [COMPONENT <component>]
8      [NAMELINK_COMPONENT <component>]
9      [OPTIONAL] [EXCLUDE_FROM_ALL]
10     [NAMELINK_ONLY|NAMELINK_SKIP]
11     ] [...]
12     [INCLUDES DESTINATION [<dir> ...]]
13 )

```

参数中的**TARGET**可以是很多种目标文件，最常见的是通过**ADD\_EXECUTABLE**或者**ADD\_LIBRARY**定义的目标文件，即可执行二进制、动态库、静态库

目标文件	内容	安装目录变量	默认安装文件夹
<b>ARCHIVE</b>	<b>静态库</b>	\${CMAKE_INSTALL_LIBDIR}	lib
<b>LIBRARY</b>	<b>动态库</b>	\${CMAKE_INSTALL_LIBDIR}	lib
<b>RUNTIME</b>	<b>可执行二进制文件</b>	\${CMAKE_INSTALL_BINDIR}	bin
PUBLIC_HEADER	与库关联的PUBLIC头文件	\${CMAKE_INSTALL_INCLUDEDIR}	include
PRIVATE_HEADER	与库关联的PRIVATE头文件	\${CMAKE_INSTALL_INCLUDEDIR}	include

为了符合一般的默认安装路径，如果设置了**DESTINATION**参数，推荐配置在安装目录变量下的文件夹。

例如：



```

1  INSTALL(TARGETS myrun mylib mystaticlib
2      RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
3      LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
4      ARCHIVE DESTINATION ${CMAKE_INSTALL_LIBDIR}
5  )

```

上面的例子会将：可执行二进制myrun安装到\${CMAKE\_INSTALL\_BINDIR}目录，动态库libmylib.so安装到\${CMAKE\_INSTALL\_LIBDIR}目录，静态库libmystaticlib.a安装到\${CMAKE\_INSTALL\_LIBDIR}目录。

该命令的其他一些参数的含义：

- DESTINATION：指定磁盘上要安装文件的目录；
- PERMISSIONS：指定安装文件的权限。有效权限是OWNER\_READ，OWNER\_WRITE，OWNER\_EXECUTE，GROUP\_READ，GROUP\_WRITE，GROUP\_EXECUTE，WORLD\_READ，WORLD\_WRITE，WORLD\_EXECUTE，SETUID和SETGID；
- CONFIGURATIONS：指定安装规则适用的构建配置列表(DEBUG或RELEASE等)；
- EXCLUDE\_FROM\_ALL：指定该文件从完整安装中排除，仅作为特定于组件的安装的一部分进行安装；
- OPTIONAL：如果要安装的文件不存在，则指定不是错误。

注意一下CONFIGURATIONS参数，此选项指定的值仅适用于此选项之后列出的选项：例如，要为调试和发布配置设置单独的安装路径，请执行以下操作：

```

1  install(TARGETS target
2      CONFIGURATIONS Debug
3      RUNTIME DESTINATION Debug/bin)
4  install(TARGETS target
5      CONFIGURATIONS Release
6      RUNTIME DESTINATION Release/bin)

```

也就是说，DEBUG和RELEASE版本的DESTINATION安装路径不同，那么DESTINATION必须在CONFIGURATIONS后面。

## 普通文件的安装

```
1  install(<FILES|PROGRAMS> files...
2      TYPE <type> | DESTINATION <dir>
3      [PERMISSIONS permissions...]
4      [CONFIGURATIONS [Debug|Release|...]]
5      [COMPONENT <component>]
6      [RENAME <name>] [OPTIONAL] [EXCLUDE_FROM_ALL])
```

FILES|PROGRAMS若为相对路径给出的文件名，将相对于当前源目录进行解释。其中，FILES为普通的文本文件，PROGRAMS指的是非目标文件的可执行程序(如脚本文件)。

如果未提供PERMISSIONS参数，默认情况下，普通的文本文件将具有OWNER\_WRITE, OWNER\_READ, GROUP\_READ和WORLD\_READ权限，即644权限；而非目标文件的可执行程序将具有OWNER\_EXECUTE, GROUP\_EXECUTE,和WORLD\_EXECUTE，即755权限。

其中，不同的TYPE，cmake也提供了默认的安装路径，如下表：

TYPE类型	安装目录变量	默认安装文件夹
BIN	\${CMAKE_INSTALL_BINDIR}	bin
SBIN	\${CMAKE_INSTALL_SBINDIR}	sbin
LIB	\${CMAKE_INSTALL_LIBDIR}	lib
INCLUDE	\${CMAKE_INSTALL_INCLUDEDIR}	include
SYSCONF	\${CMAKE_INSTALL_SYSCONFDIR}	etc
SHAREDSTATE	\${CMAKE_INSTALL_SHAREDSTATEDIR}	com
LOCALSTATE	\${CMAKE_INSTALL_LOCALSTATEDIR}	var
RUNSTATE	\${CMAKE_INSTALL_RUNSTATEDIR}	/run
DATA	\${CMAKE_INSTALL_DATADIR}	
INFO	\${CMAKE_INSTALL_INFODIR}	/info
LOCALE	\${CMAKE_INSTALL_LOCALEDIR}	/locale
MAN	\${CMAKE_INSTALL_MANDIR}	/man
DOC	\${CMAKE_INSTALL_DOCDIR}	/doc

请注意，某些类型的内置默认值使用 **DATAROOT** 目录作为前缀，以 **CMAKE\_INSTALL\_DATAROOTDIR** 变量值为内容。

该命令的其他一些参数的含义：

- DESTINATION：指定磁盘上要安装文件的目录；
- PERMISSIONS：指定安装文件的权限。有效权限是OWNER\_READ，OWNER\_WRITE，OWNER\_EXECUTE，GROUP\_READ，GROUP\_WRITE，GROUP\_EXECUTE，WORLD\_READ，WORLD\_WRITE，WORLD\_EXECUTE，SETUID和SETGID；
- CONFIGURATIONS：指定安装规则适用的构建配置列表(DEBUG或RELEASE等)；

- EXCLUDE\_FROM\_ALL：指定该文件从完整安装中排除，仅作为特定于组件的安装的一部分进行安装；
- OPTIONAL：如果要安装的文件不存在，则指定不是错误；
- RENAME：指定已安装文件的名称，该名称可能与原始文件不同。仅当命令安装了单个文件时，才允许重命名。

## 目录的安装

```

1  install(DIRECTORY dirs...
2      TYPE <type> | DESTINATION <dir>
3      [FILE_PERMISSIONS permissions...]
4      [DIRECTORY_PERMISSIONS permissions...]
5      [USE_SOURCE_PERMISSIONS] [OPTIONAL] [MESSAGE_NEVER]
6      [CONFIGURATIONS [Debug|Release|...]]
7      [COMPONENT <component>] [EXCLUDE_FROM_ALL]
8      [FILES_MATCHING]
9      [[PATTERN <pattern> | REGEX <regex>]
10     [EXCLUDE] [PERMISSIONS permissions...]] [...])
  
```

该命令将一个或多个目录的内容安装到给定的目的地，目录结构被逐个复制到目标位置。每个目录名称的最后一个组成部分都附加到目标目录中，但是可以使用后跟斜杠来避免这种情况，因为它将最后一个组成部分留空。这是什么意思呢？

比如，DIRECTORY后面如果是abc意味着abc这个目录会安装在目标路径下，abc/意味着abc这个目录的内容会被安装在目标路径下，而abc目录本身却不会被安装。即，如果目录名不以/结尾，那么这个目录将被安装为目标路径下的abc，如果目录名以/结尾，代表将这个目录中的内容安装到目标路径，但不包括这个目录本身。

---

版权声明：本文为CSDN博主「Yngz\_Miao」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

FILE\_PERMISSIONS和DIRECTORY\_PERMISSIONS选项指定对目标中文件和目录的权限。如果指定了USE\_SOURCE\_PERMISSIONS而未指定FILE\_PERMISSIONS，则将从源目录结构中复制文件权限。如果未指定权限，则将为文件提供在命令的FILES形式中指定的默认权限(644权限)，而目录将被赋予在命令的PROGRAMS形式中指定的默认权限(755权限)。

可以使用PATTERN或REGEX选项以精细的粒度控制目录的安装，可以指定一个通配模式或正则表达式以匹配输入目录中遇到的目录或文件。PATTERN仅匹配完整的文件名，而REGEX将匹配文件名的任何部

分，但它可以使用/和\$模拟PATTERN行为。

某些跟随PATTERN或REGEX表达式后的参数，仅应用于满足表达式的文件或目录。如：EXCLUDE选项将跳过匹配的文件或目录。PERMISSIONS选项将覆盖匹配文件或目录的权限设置。

例如：

```
1 install(DIRECTORY icons scripts/ DESTINATION share/myproj
2         PATTERN "CVS" EXCLUDE
3         PATTERN "scripts/*"
4         PERMISSIONS OWNER_EXECUTE OWNER_WRITE OWNER_READ
5         GROUP_EXECUTE GROUP_READ)
```

C++ | 复制代码

这条命令的执行结果是：将icons目录安装到share/myproj，将scripts/中的内容安装到share/myproj，两个目录均不包含目录名为CVS的子目录，对于scripts/\*的文件指定权限为OWNER\_EXECUTE，OWNER\_WRITE，OWNER\_READ，GROUP\_EXECUTE，GROUP\_READ。

## 安装时脚本的运行

有时候需要在install的过程中打印一些语句，或者执行一些cmake指令：

```
1 install([[SCRIPT <file>] [CODE <code>]]
2         [COMPONENT <component>] [EXCLUDE_FROM_ALL] [...])
```

C++ | 复制代码

SCRIPT参数将在安装过程中调用给定的CMake脚本文件(即.cmake脚本文件)，如果脚本文件名是相对路径，则将相对于当前源目录进行解释。CODE参数将在安装过程中调用给定的CMake代码。将代码指定为双引号字符串内的单个参数。

例如：

```
1 install(CODE "MESSAGE(\"Sample install message.\")")
```

C++ | 复制代码

这条命令将会在install的过程中执行cmake代码，打印语句。

## 7 设置执行目录+编译debug和release版本

### 7.1 编译debug版本release版本

## 工程7.1

Basic | 复制代码

```
1  .
2  ├── CMakeLists.txt
3  ├── doc
4  │   ├── darren.txt
5  │   └── README.md
6  ├── release
7  │   └── linux
8  │       ├── Debug
9  │       └── Release
10 └── src
    ├── CMakeLists.txt
    ├── dir1
    │   ├── CMakeLists.txt
    │   ├── dir1.c
    │   └── dir1.h
    ├── dir2
    │   ├── CMakeLists.txt
    │   ├── dir2.c
    │   └── dir2.h
    ├── main.c
    ├── Makefile
    ├── README.md
    ├── release
    └── linux
```

### 编译debug版本

```
ubuntu% cmake -DCMAKE_INSTALL_PREFIX=/tmp/usr ..
```

```
-- Debug版本
```

```
-- Debug版本
```

```
-- Debug版本
```

```
-- Configuring done
```

```
-- Generating done
```

```
-- Build files have been written to: /mnt/hgfs/0voice/vip/20210128-makefile-cmake/src-cmake/7.1/build
```

```
ubuntu% make install
```

```
[ 33%] Built target Dir2
```

```
[ 66%] Built target Dir1
```

```
[100%] Built target multi-dir
```

```
Install the project...
```

```
-- Install configuration: ""
```

```
-- Up-to-date: /tmp/usr/share/doc/cmake/0voice
-- Up-to-date: /tmp/usr/share/doc/cmake/0voice/darren.txt
-- Up-to-date: /tmp/usr/share/doc/cmake/0voice/README.md
-- Installing: /tmp/usr/bin/multi-dir
-- Set runtime path of "/tmp/usr/bin/multi-dir" to ""
-- Installing: /tmp/usr/lib/libDir1.so
-- Installing: /tmp/usr/include/dir1.h
-- Installing: /tmp/usr/lib/libDir2.so
-- Installing: /tmp/usr/include/dir2.h
ubuntu% /tmp/usr/bin/multi-dir
```

编译release版本

```
cmake -DCMAKE_BUILD_TYPE=Release ..
```

## 语法: PROJECT\_SOURCE\_DIR

PROJECT\_SOURCE\_DIR为包含PROJECT()的最近一个CMakeLists.txt文件所在的文件夹。

## 7.2 编译选项

```
1  # 设置release版本还是debug版本
2  if(${CMAKE_BUILD_TYPE} MATCHES "Release")
3      message(STATUS "Release版本")
4      set(BuildType "Release")
5      SET(CMAKE_C_FLAGS "$ENV{CFLAGS} -DNODEBUG -O3 -Wall")
6      SET(CMAKE_CXX_FLAGS "$ENV{CXXFLAGS} -DNODEBUG -O3 -Wall")
7      MESSAGE(STATUS "CXXFLAGS: " ${CMAKE_CXX_FLAGS})
8      MESSAGE(STATUS "CFLAGS: " ${CMAKE_C_FLAGS})
9  else()
10     set(BuildType "Debug")
11     message(STATUS "Debug版本")
12     SET(CMAKE_CXX_FLAGS "$ENV{CXXFLAGS} -Wall -O0 -g")
13     # SET(CMAKE_C_FLAGS "$ENV{CFLAGS} -O0 -g")
14     SET(CMAKE_C_FLAGS "$ENV{CFLAGS} -O0 -g")
15     MESSAGE(STATUS "CXXFLAGS: " ${CMAKE_CXX_FLAGS})
16     MESSAGE(STATUS "CFLAGS: " ${CMAKE_C_FLAGS})
17 endif()
```

编译debug版本

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

编译release版本

```
cmake -DCMAKE_BUILD_TYPE=Release ..
```

在cmake脚本中，设置编译选项可以通过 `add_compile_options` 命令，也可以通过set命令修改 `CMAKE_CXX_FLAGS` 或 `CMAKE_C_FLAGS`。

使用这两种方式在有的情况下效果是一样的，但请注意它们还是有区别的：

`add_compile_options` 命令添加的编译选项是针对所有编译器的(包括c和c++编译器)，而set命令设置 `CMAKE_C_FLAGS` 或 `CMAKE_CXX_FLAGS` 变量则是分别只针对c和c++编译器的。

例如下面的代码

```
1  #判断编译器类型,如果是gcc编译器,则在编译选项中加入c++11支持
2  if(CMAKE_COMPILER_IS_GNUCXX)
3      add_compile_options(-std=c++11)
4      message(STATUS "optional:-std=c++11")
5  endif(CMAKE_COMPILER_IS_GNUCXX) • 1
```

Bash 复制代码

使用 `add_compile_options` 添加 `-std=c++11` 选项，是想在编译c++代码时加上c++11支持选项。但是因为 `add_compile_options` 是针对所有类型编译器的，所以在编译c代码时，就会产生如下warning

```
J:\workspace\facecl.gcc>make b64
[ 50%] Building C object libb64/CMakeFiles/b64.dir/libb64-1.2.1/src/cdecode.c.obj
cc1.exe: warning: command line option '-std=c++11' is valid for C++/ObjC++ but not for C
[100%] Building C object libb64/CMakeFiles/b64.dir/libb64-1.2.1/src/cencode.c.obj
cc1.exe: warning: command line option '-std=c++11' is valid for C++/ObjC++ but not for C
Linking C static library libb64.a
[100%] Built target b64
```

虽然并不影响编译，但看着的确是不爽啊，要消除这个warning,就不能使用 `add_compile_options`，而是只针对c++编译器添加这个option。

所以如下修改代码，则警告消除。

```
1  #判断编译器类型,如果是gcc编译器,则在编译选项中加入c++11支持
2  if(CMAKE_COMPILER_IS_GNUCXX)
3      set(CMAKE_CXX_FLAGS "-std=c++11 ${CMAKE_CXX_FLAGS}")
4      message(STATUS "optional:-std=c++11")
5  endif(CMAKE_COMPILER_IS_GNUCXX)
```

Plain Text 复制代码



举一反三，我们就可以想到，`add_definitions` 这个命令也是同样针对所有编译器，一样注意这个区别。

## 8 跨平台

参考ztoolkit C++11写的代码，线程池、thread、mutex

## 参考

<https://github.com/yngzMiao/yngzmiao-blogs/tree/master/2019Q4/20191105>。