# CMake实战-课件

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

带书签版本参考《源码和文档说明.txt》

# 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/v3.19/

# 1 安装 cmake

## 1.1 卸载已经安装的旧版的CMake[非必需]

```
apt-get autoremove cmake
```

## 1.2 文件下载解压：

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

解压：

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

查看解压后目录:

```
tree -L 2 cmake-3.9.1-Linux-x86_64
cmake-3.9.1-Linux-x86_64
├── bin
│   ├── ccmake
│   ├── cmake
│   ├── cmake-gui
│   ├── cpack
│   └── ctest
├── doc
│   └── cmake
├── man
│   ├── man1
│   └── man7
└── share
    ├── aclocal
    ├── applications
    ├── cmake-3.9
    ├── icons
    └── mime
12 directories, 5 files
```

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`

```
mv cmake-3.9.1-Linux-x86_64 /opt/cmake-3.9.1
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

```
# 单个目录实现
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)
# 手动加入文件
SET(SRC_LIST main.c)
MESSAGE(STATUS "THIS IS BINARY DIR " ${PROJECT_BINARY_DIR})
MESSAGE(STATUS "THIS IS SOURCE DIR " ${PROJECT_SOURCE_DIR})
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：<br>SEND_ERROR：产生错误，生成过程被跳过<br>STATUS：输出前缀为-的信息。<br>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

```
.
├── build
├── CMakeLists.txt
├── doc
│   ├── darren.txt
│   └── README.MD
└── src
    ├── CMakeLists.txt
    └── 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

## 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

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(TARGETS targets...<br>[[ARCHIVE\|LIBRARY\|RUNTIME]<br>[DESTINATION <dir>]<br>[PERMISSIONS permissions...]<br>[CONFIGURATIONS<br>[Debug\|Release\|...]]<br>[COMPONENT <component>]<br>[OPTIONAL]<br>] [...]) |
| | |

| | |
|---|---|
| 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(FILES files... DESTINATION <dir><br>[PERMISSIONS permissions...]<br>[CONFIGURATIONS [Debug\|Release\|...]]<br>[COMPONENT <component>]<br>[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(PROGRAMS files... DESTINATION <dir><br>[PERMISSIONS permissions...]<br>[CONFIGURATIONS [Debug\|Release\|...]]<br>[COMPONENT <component>]<br>[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(DIRECTORY dirs... DESTINATION <dir><br>[FILE_PERMISSIONS permissions...] |

| | |
|---|---|
| | [DIRECTORY_PERMISSIONS permissions...]<br>[USE_SOURCE_PERMISSIONS]<br>[CONFIGURATIONS [Debug\|Release\|...]]<br>[COMPONENT <component>]<br>[[PATTERN <pattern> \| REGEX <regex>]<br>[EXCLUDE] [PERMISSIONS permissions...]] [...]) |
| 说明 | 主要介绍其中的DIRECTORY、PATTERN和PERMISSIONS参数：<br>DIRECTORY：后面连接的是所在Source目录的相对路径。<br>PATTERN：用于使用正则表达式进行过滤。<br>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

```
├── CMakeLists.txt
├── doc
│   ├── darren.txt
│   └── README.MD
└── 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

```
.
├── CMakeLists.txt
├── doc
│   ├── darren.txt
│   └── README.MD
└── src
    ├── CMakeLists.txt
    ├── dir1
    │   ├── dir1.c
    │   └── dir1.h
    ├── dir2
    │   ├── dir2.c
    │   └── dir2.h
    └── main.c
```

```
src
    ├─── CMakeLists.txt
```

```
# 单个目录实现
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)
# 工程
PROJECT(0VOICE)
# 手动加入文件
SET(SRC_LIST main.c)
MESSAGE(STATUS "THIS IS BINARY DIR " ${PROJECT_BINARY_DIR})
MESSAGE(STATUS "THIS IS SOURCE DIR " ${PROJECT_SOURCE_DIR})

#设置子目录
set(SUB_DIR_LIST "${CMAKE_CURRENT_SOURCE_DIR}/dir1" "${CMAKE_CURRENT_SOURCE_DIR}/dir2")

foreach(SUB_DIR ${SUB_DIR_LIST})
    #遍历源文件
    aux_source_directory(${SUB_DIR} SRC_LIST)
endforeach()

# 添加头文件路径
INCLUDE_DIRECTORIES("dir1")
INCLUDE_DIRECTORIES("dir2")


ADD_EXECUTABLE(darren ${SRC_LIST} )


# 将执行文件安装到bin目录
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

```
# 设置release版本还是debug版本
if(${CMAKE_BUILD_TYPE} MATCHES "Release")
    MESSAGE(STATUS "Release版本")
    SET(BuildType "Release")
else()
    SET(BuildType "Debug")
    MESSAGE(STATUS "Debug版本")
endif()

#设置lib库目录
SET(RELEASE_DIR ${PROJECT_SOURCE_DIR}/release)
# debug和release版本目录不一样
#设置生成的so动态库最后输出的路径
SET(LIBRARY_OUTPUT_PATH ${RELEASE_DIR}/linux/${BuildType})
# -fPIC 动态库必须的选项
ADD_COMPILE_OPTIONS(-fPIC)

# 查找当前目录下的所有源文件
# 并将名称保存到 DIR_LIB_SRCS 变量
AUX_SOURCE_DIRECTORY(. DIR_LIB_SRCS)
# 生成静态库链接库Dir1
#ADD_LIBRARY (Dir1 ${DIR_LIB_SRCS})
# 生成动态库
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

```
# 设置release版本还是debug版本
if(${CMAKE_BUILD_TYPE} MATCHES "Release")
    MESSAGE(STATUS "Release版本")
    SET(BuildType "Release")
else()
    SET(BuildType "Debug")
    MESSAGE(STATUS "Debug版本")
endif()

#设置lib库目录
SET(RELEASE_DIR ${PROJECT_SOURCE_DIR}/release)
# debug和release版本目录不一样
#设置生成的so动态库最后输出的路径
SET(LIBRARY_OUTPUT_PATH ${RELEASE_DIR}/linux/${BuildType})
ADD_COMPILE_OPTIONS(-fPIC)

# 查找当前目录下的所有源文件
# 并将名称保存到 DIR_LIB_SRCS 变量
AUX_SOURCE_DIRECTORY(. DIR_LIB_SRCS)
# 生成静态库链接库Dir1
ADD_LIBRARY (Dir1 ${DIR_LIB_SRCS})
# 将库文件安装到lib目录
INSTALL(TARGETS Dir1 DESTINATION lib)
# 将头文件include
INSTALL(FILES dir1.h DESTINATION include)
```

编译安装

```
ubuntu% cmake -DCMAKE_INSTALL_PREFIX=/tmp/usr ..
ubuntu% make
ubuntu% make install
[100%] Built target Dir1
Install the project...
-- Install configuration: ""
-- Up-to-date: /tmp/usr/lib/libDir1.a
```

```
-- 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

```
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)
# 工程
PROJECT(0VOICE)
# 手动加入文件
SET(SRC_LIST main.c)
MESSAGE(STATUS "THIS IS BINARY DIR " ${PROJECT_BINARY_DIR})
MESSAGE(STATUS "THIS IS SOURCE DIR " ${PROJECT_SOURCE_DIR})

INCLUDE_DIRECTORIES("${CMAKE_CURRENT_SOURCE_DIR}/lib")
# 库的路径
LINK_DIRECTORIES("${CMAKE_CURRENT_SOURCE_DIR}/lib")
# 生成执行文件
ADD_EXECUTABLE(darren  ${SRC_LIST})
# 引用动态库
TARGET_LINK_LIBRARIES(darren Dir1)
```

## 5.2 调用动态库

```
# 单个目录实现
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)
# 工程
```

```
PROJECT(0VOICE)
# 手动加入文件
SET(SRC_LIST main.c)
MESSAGE(STATUS "THIS IS BINARY DIR " ${PROJECT_BINARY_DIR})
MESSAGE(STATUS "THIS IS SOURCE DIR " ${PROJECT_SOURCE_DIR})

INCLUDE_DIRECTORIES("${CMAKE_CURRENT_SOURCE_DIR}/lib")

LINK_DIRECTORIES("${CMAKE_CURRENT_SOURCE_DIR}/lib")
# 引用动态库
ADD_EXECUTABLE(darren  ${SRC_LIST})
#同时静态库、动态库 优先连接动态库
#TARGET_LINK_LIBRARIES(darren Dir1)
# 强制使用静态库
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

参考2.2

# 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

```
.
├── CMakeLists.txt
├── doc
│   ├── darren.txt
│   └── README.md
├── release
│   └── linux
```

```
8  │        ├── Debug
9  │        └── Release
10 └── src
11     ├── CMakeLists.txt
12     ├── dir1
13     │   ├── CMakeLists.txt
14     │   ├── dir1.c
15     │   └── dir1.h
16     ├── dir2
17     │   ├── CMakeLists.txt
18     │   ├── dir2.c
19     │   └── dir2.h
20     ├── main.c
21     ├── Makefile
22     ├── README.md
23     └── release
24         └── 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

```
# 设置release版本还是debug版本
if(${CMAKE_BUILD_TYPE} MATCHES "Release")
    message(STATUS "Release版本")
    set(BuildType "Release")
    SET(CMAKE_C_FLAGS "$ENV{CFLAGS} -DNODEBUG -O3 -Wall")
    SET(CMAKE_CXX_FLAGS "$ENV{CXXFLAGS} -DNODEBUG -O3 -Wall")
    MESSAGE(STATUS "CXXFLAGS: " ${CMAKE_CXX_FLAGS})
    MESSAGE(STATUS "CFLAGS: " ${CMAKE_C_FLAGS})
else()
    set(BuildType "Debug")
    message(STATUS "Debug版本")
    SET(CMAKE_CXX_FLAGS "$ENV{CXXFLAGS} -Wall  -O0 -g")
 #   SET(CMAKE_C_FILAGS "-O0 -g")
    SET(CMAKE_C_FLAGS "$ENV{CFLAGS} -O0 -g")
    MESSAGE(STATUS "CXXFLAGS: " ${CMAKE_CXX_FLAGS})
    MESSAGE(STATUS "CFLAGS: " ${CMAKE_C_FILAGS})
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

```
#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
if(CMAKE_COMPILER_IS_GNUCXX)
    add_compile_options(-std=c++11)
    message(STATUS "optional:-std=c++11")
endif(CMAKE_COMPILER_IS_GNUCXX)• 1
```

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。

```
#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
if(CMAKE_COMPILER_IS_GNUCXX)
    set(CMAKE_CXX_FLAGS "-std=c++11 ${CMAKE_CXX_FLAGS}")
    message(STATUS "optional:-std=c++11")
endif(CMAKE_COMPILER_IS_GNUCXX)
```

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zltoolkit