

# C/C++Linux服务器开发

## 高级架构师课程

三年课程沉淀

五次精益升级

十年行业积累

百个实战项目

十万内容受众

讲师:darren/326873713



扫一扫 升职加薪

班主任:柚子/2690491738

# 讲师介绍--专业来自专注和实力



**Darren老师**

曾供职于国内知名半导体公司（珠海扬智/深圳联发科），曾在某互联网公司担任音视频通话项目经理。主要从事音视频驱动、多媒体中间件、流媒体服务器的开发，开发过即时通讯+音视频通话的大型项目，在音视频、C/C++/GO Linux服务器领域有丰富的实战经验。



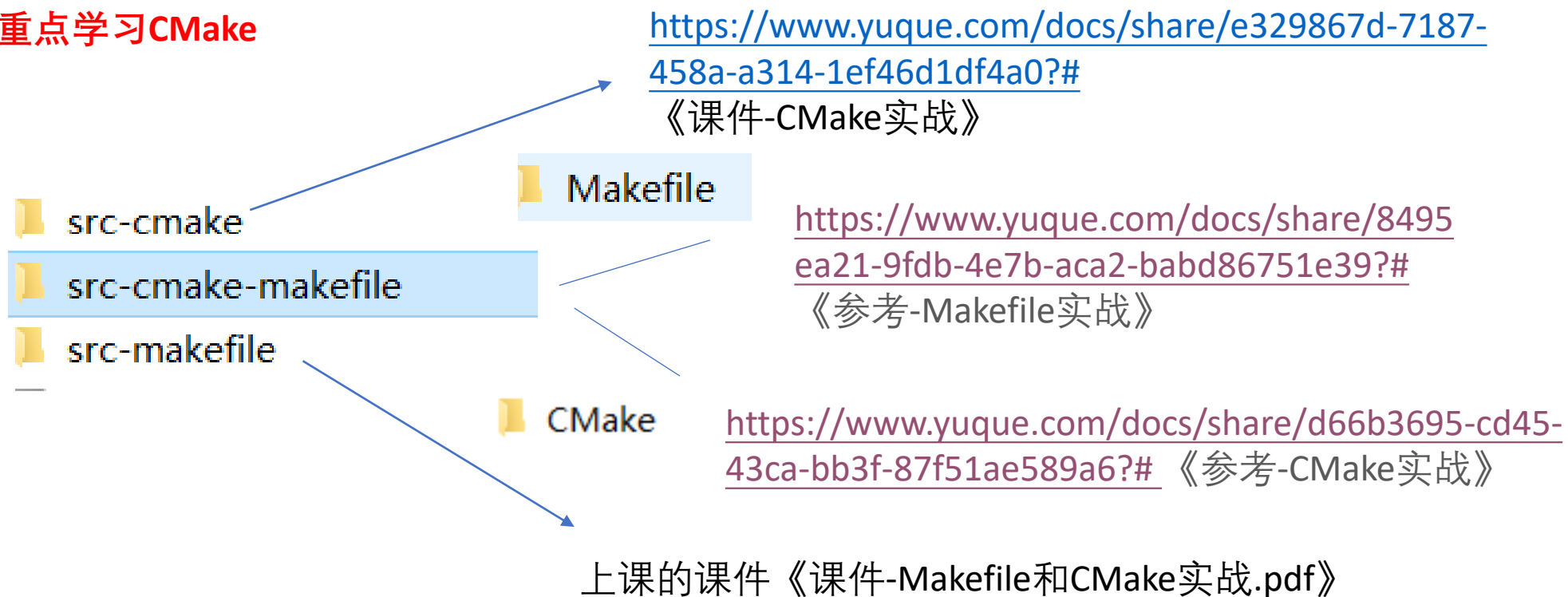
# Makefile和CMake实践

Makefile

Cmake

## 源码对应的路径和文档

### 重点学习CMake



## 1.1 简单Makefile

```
all:
    @echo "hello all"
test:
    @echo "hello test"
```

范例1.1

```
$ make
hello all
$ make test
hello test
$ make all
hello all
```

```
test:
    @echo "hello test"
all:
    @echo "hello all"
```

范例1.2

```
$ make
hello test
$ make test
hello test
$ make all
hello all
```

```
all: test
    @echo "hello all"
test:
    @echo "hello test"
```

范例1.3

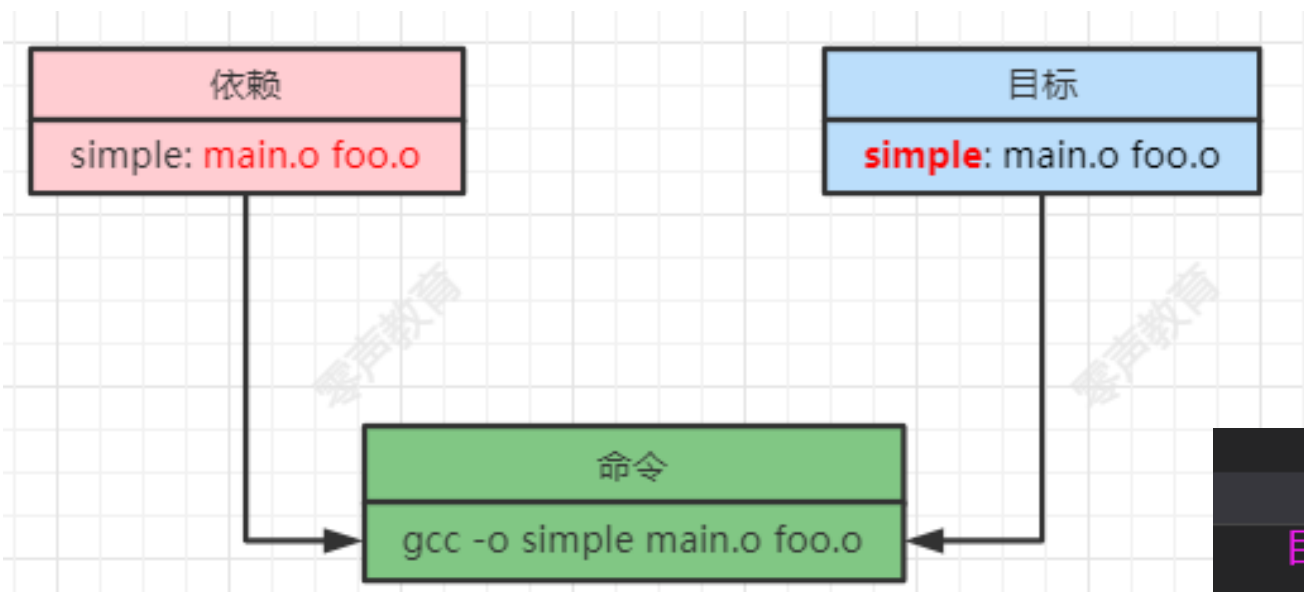
```
$ make
hello test
hello all
$ make test
hello test
$ make all
hello test
hello all
```

重点:

1. 目标、依赖、命令
2. all有什么意义
3. all和test的顺序问题
4. 空格符号的影响



## 1.2 Makefile三要素



```
1.4.1 > M Makefile
1  .PHONY: main clean
2  simple: main.o foo.o
3      gcc -o simple main.o foo.o
4  main.o: main.c
5      gcc -o main.o -c main.c
6  foo.o: foo.c
7      gcc -o foo.o -c foo.c
8  clean:
9      rm simple main.o foo.o
```

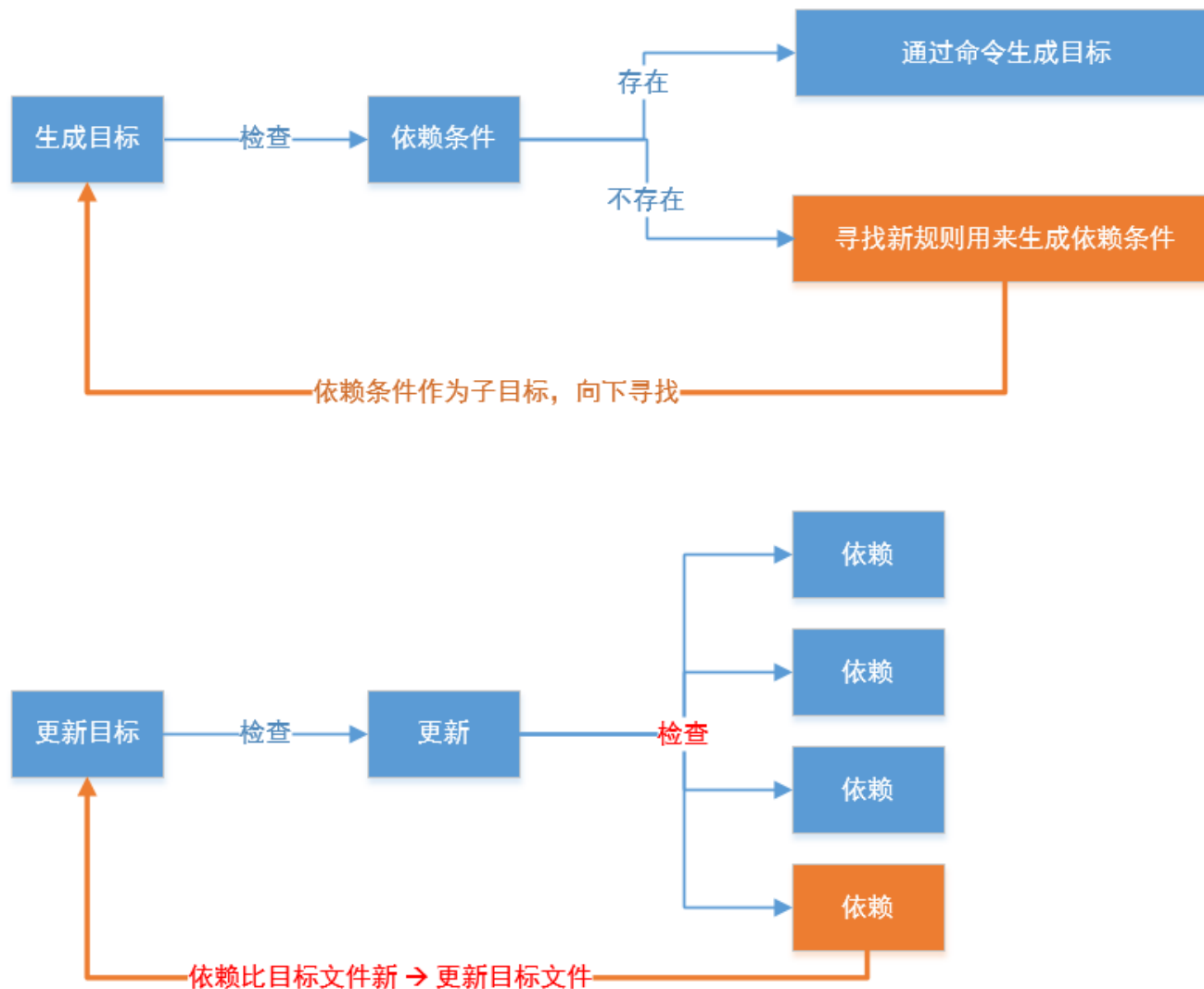
Annotations in the image:

- 目标** (Target) points to line 2: `simple: main.o foo.o`.
- 依赖** (Prerequisite) points to the prerequisites `main.o foo.o` on line 2.
- 命令** (Command) points to the command `gcc -o simple main.o foo.o` on line 3.
- 子目标** (Sub-target) points to line 4: `main.o: main.c`.



## 1.3 Makefile工作

### 工作原理



## 1.4.1 编译程序

```
2 √ simple: main.o foo.o
3   | gcc -o simple main.o foo.o
4 √ main.o: main.c
5   | gcc -o main.o -c main.c
6 √ foo.o: foo.c
7   | gcc -o foo.o -c foo.c
8 √ clean:
9   | rm simple main.o foo.o
```

1. 目标
2. 依赖
3. 命令

范例1.4.1





## 1.4.1 编译程序-伪对象. PHONY

```
2 √ simple: main.o foo.o
3   | gcc -o simple main.o foo.o
4 √ main.o: main.c
5   | gcc -o main.o -c main.c
6 √ foo.o: foo.c
7   | gcc -o foo.o -c foo.c
8 √ clean:
9   | rm simple main.o foo.o
```



```
1 .PHONY: main clean
2 simple: main.o foo.o
3   | gcc -o simple main.o foo.o
4 main.o: main.c
5   | gcc -o main.o -c main.c
6 foo.o: foo.c
7   | gcc -o foo.o -c foo.c
8 clean:
9   | rm simple main.o foo.o
```

范例1.4.1

1. 在程序所在的目录创建一个 clean 文件  
    .PHONY: main clean
2. 执行make clean
3. 提示: make: 'clean' is up to date.



## 1.5.1 变量

```
.PHONY: clean
CC = gcc
RM = rm
EXE = simple
OBJS = main.o foo.o
$(EXE): $(OBJS)
|   $(CC) -o $(EXE) $(OBJS)
main.o: main.c
|   $(CC) -o main.o -c main.c
foo.o: foo.c
|   $(CC) -o foo.o -c foo.c
clean:
|   $(RM) $(EXE) $(OBJS)
```

- CC 保存编译器名
- RM 用于指示删除文件的命令
- EXE 存放可执行文件名
- OBJS放置所有的目标文件名

范例1.5.1



## 1.5.2 自动变量

```
.PHONY: all
all: first second third
    @echo "\$$@ = $$@"
    @echo "$$^ = $$^"
    @echo "$$< = $$<"
first:
    @echo "1 first"
second:
    @echo "2 second"
third:
    @echo "3 third"
```

范例1.5.2

```
1 first
2 second
3 third
$@ = all
$$^ = first second third
$$< = first
```

- **\$@**用于表示一个规则中的目标。当我们的一个规则中有多个目标时，\$@所指的是其中任何造成命令被运行的**目标**。
- **\$\$^**则表示的是规则中的所有先择条件。
- **\$\$<**表示的是规则中的第一个先决条件。



## 1.5.3 自动变量-编译

```
.PHONY: clean
CC = gcc
RM = rm
EXE = simple
#SRCS = main.c foo.c
SRCS = $(wildcard *.c)
# 把.c换成对应的.o
#OBJS = foo.o foo2.o main.o
OBJS = $(patsubst %.c,%.o,$(SRCS))

$(EXE): $(OBJS)
| $(CC) -o $@ $^
%.o: %.c
| $(CC) -o $@ -c $^
clean:
| $(RM) $(EXE) $(OBJS)
```

**wildcard** 是通配符函数，通过它可以得到我们所需的文件形式：\$(wildcard pattern)

**patsubst** 函数是用来进行**字符串替换**的，其形式是：\$(patsubst pattern, replacement, text)

范例1.5.3



## 1.5.4 依赖第三方库

```
1 CROSS =
2 # 定义CC为gcc编译
3 CC = $(CROSS)gcc
4 # 定义CXX为g++编译
5 CXX = $(CROSS)g++
6 # 定义DEBUG 方式为 -g -O2
7 DEBUG = -g -O2
8 CFLAGS = $(DEBUG) -Wall -c
9 RM = rm -rf
10
11 # /定义SRC为当前工程目录下所有的.cpp文件
12 SRCS = $(wildcard ./*.c)
13 # 定义OBJS为SRCS对应的.o文件
14 OBJS = $(patsubst %.c, %.o, $(SRCS))
15 # 定义HEADER_PATH为当前工程中的头文件路径
16 HEADER_PATH = -I ./include/
17 # 定义LIB_PATH为当前工程中的头文件路径
18 LIB_PATH = -L ./lib/
19 # 输出当前LIB_PATH中的内容
20 $(warning LIB_PATH)
21 # 制定LIBS链接库的名称
22 LIBS=-lpthread
23 # lib中的库文件名称为libpthread.so
```

```
25 # 定义当前生成的版本
26 VERSION = 1.0.0
27 # 定义生成可执行文件的名称
28 TARGET = simple.$(VERSION)
29
30 $(TARGET) : $(OBJS)
31 √ # 告诉编译器生成可执行文件时库存放的目录, 以及库的名字
32 | $(CXX) $^ -o $@ $(LIB_PATH) $(LIBS)
33 $(OBJS):%.o : %.c
34 √ #告诉编译器生成中间文件时头文件的所在目录
35 | $(CXX) $(CFLAGS) $< -o $@ $(HEADER_PATH)
36 √ clean:
37 | $(RM) $(TARGET) *.o
```

范例1.5.4



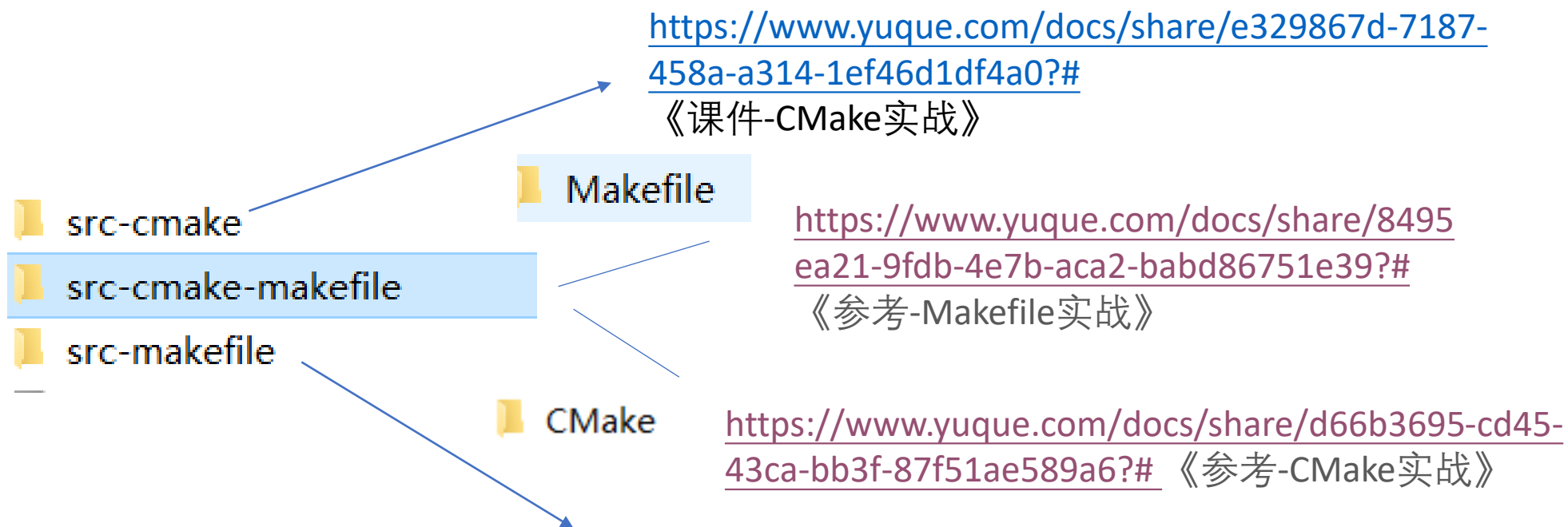
## 1.6 更复杂的范例

参考范例：

2.53

Huge

NtyCo



上课的课件《课件-Makefile和CMake实战.pdf》



## 2 CMake

CMake是一个跨平台的安装（[编译](#)）工具，可以用简单的语句来描述所有平台的安装(编译过程)。它能够输出各种各样的makefile或者project文件，能测试[编译器](#)所支持的C++特性,类似[UNIX](#)下的automake。只是 CMake 的[组态档](#)取名为CMakeLists.txt。Cmake 并不直接建构出最终的软件，而是产生标准的建构档（如 Unix 的 Makefile 或 [Windows Visual C++](#) 的 projects/workspaces），然后再依一般的建构方式使用。

```
lqf@ubuntu:~$ cmake -version  
cmake version 3.21.3
```



## 2.1 源码编译安装cmake

参考该链接：

<https://www.yuque.com/docs/share/d66b3695-cd45-43ca-bb3f-87f51ae589a6?#> 《参考-CMake实战》





## 2.2 cmake编译方式

- 当前目录`cmake .`
- 创建build目录，进入build目录后再`cmake ..`



## 2.2 单个文件目录实现1

```
# 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})
```

范例: src-cmake/2.1-1

- PROJECT(projectname [CXX] [C] [Java])
- SET(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]])
- MESSAGE([SEND\_ERROR | STATUS | FATAL\_ERROR] "message to display" ...)
- ADD\_EXECUTABLE([BINARY] [SOURCE\_LIST])



## 2.2 单个文件目录实现2

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

安装目录到某个路径

默认路径: `/usr/local/`

指定路径: `cmake -DCMAKE_INSTALL_PREFIX=/tmp/usr ..`

# CMake 最低版本号要求

`cmake_minimum_required (VERSION 2.8)`

`PROJECT(VOICE)`

# 添加子目录

`ADD_SUBDIRECTORY(src)`

`#INSTALL(FILES COPYRIGHT README DESTINATION`

`share/doc/cmake/0voice)`

# 安装doc到 share/doc/cmake/0voice 目录

# 默认/usr/local/

#指定自定义目录, 比如 `cmake -`

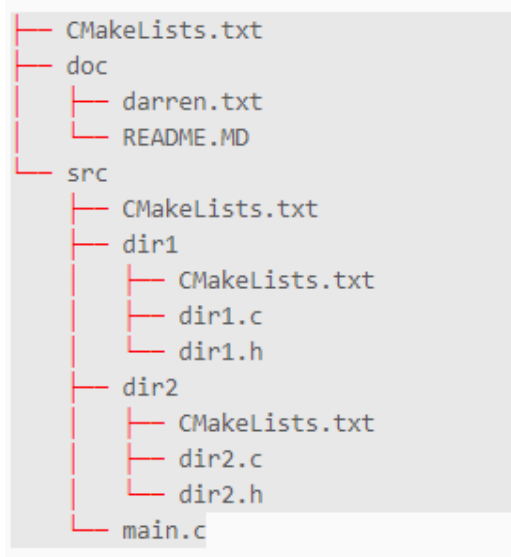
`DCMAKE_INSTALL_PREFIX=/tmp/usr ..`

`INSTALL(DIRECTORY doc/ DESTINATION share/doc/cmake/0voice)`



## 3.1 多个目录实现-子目录编译成库文件

工程：3.1-1



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** 生成库文件

`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

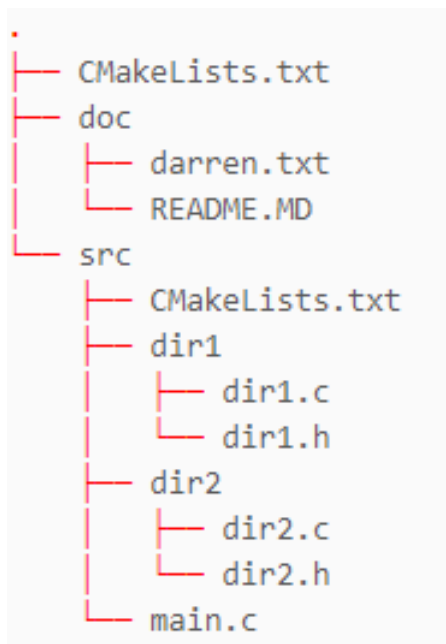
`TARGET_LINK_LIBRARIES(darren dir1 dir2)`

[cmake-properties\(7\) — CMake 3.24.0-rc4 Documentation](#)



## 3.2 多个目录实现-子目录使用源码编译

工程：3.2-1



语法：AUX\_SOURCE\_DIRECTORY

找在某个路径下的所有源文件

`aux_source_directory(<dir> <variable>)`

```
1  AUX_SOURCE_DIRECTORY(. DIR_SRCS)
2  ADD_LIBRARY(dir2 ${DIR_SRCS})
```



## 4.1 生成库-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})
```



## 4.2 生成库-生成静态库+安装到指定目录

```
# 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:

ubuntu% cmake -DCMAKE\_INSTALL\_PREFIX=/tmp/usr ..

ubuntu% make

ubuntu% make install

工程: 4.2



## 5.1 调用库-调用静态库

工程：5.1 5.2

```
# 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)
```

```
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

工程：6.1

install: 配置程序打包过程中的目标 (TARGETS)、文件 (FILES)、路径 (DIRECTORY)、代码 (CODE) 和输出配置 (EXPORT)

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

```
install(DIRECTORY src/${SUB_DIR} DESTINATION ${INSTALL_PATH_INCLUDE} FILES_MATCHING PATTERN "*.h")
foreach()
```

```
install(TARGETS ${CMAKE_PROJECT_NAME}_static ARCHIVE DESTINATION ${INSTALL_PATH_LIB})
```



## 7.1 设置执行目录+编译debug和release版本

工程7.1

```
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
```

Debug版本: `cmake -DCMAKE_INSTALL_PREFIX=/tmp/usr ..`

Release版本: `cmake -DCMAKE_BUILD_TYPE=Release ..`



## 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_FLAGS "-O0 -g")
    SET(CMAKE_C_FLAGS "$ENV{CFLAGS} -O0 -g")
    MESSAGE(STATUS "CXXFLAGS: " ${CMAKE_CXX_FLAGS})
    MESSAGE(STATUS "CFLAGS: " ${CMAKE_C_FLAGS})
endif()
```

