

Cmake For Linux

- 1 观前提醒:
- 2 1. 首先明确Cmake的概念: 一种开源、跨平台的构建工具, 可以通过编写简单的配置文件去生成本地的Makefile
- 3 2. 所有CMakeLists文件中指令的含义详见附 I
- 4 3. 本文档所用到的所有源程序, 头文件都在附IV
- 5 4. 本文重复使用的主程序详见附III
- 6 5. 本文在Typora环境下编辑, 所以Typora阅读体验最佳
- 7 正式开始:

0. 安装

- 1 安装: `sudo apt install cmake`, 如果安装出现`正在等待缓存锁`, 暴力重启即可
- 2 查看版本: `cmake --version`, 本次安装的是`3.22.1`

1. 同一目录一个文件示例

1.1. 目录结构

```
1 ./DHY
2 main.c
3 CMakeLists.txt
```

1.2. 程序编写

```
1 /*main.c*/
2 #include <stdio.h>
3 int main()
4 {
5     printf("Cmake\n");
6     return 0;
7 }
```

```
1 #CMakeLists.txt
2 cmake_minimum_required (VERSION 3.10)
3 project (demo)
4 add_executable(main main.c)
```

1.3. 构建与运行

- 1 1. 在根目录./DHY文件夹出打开终端
- 2 2. 输入Cmake .后成功生成Makefile
- 3 3. 输入make需要的elf文件main也成功生成了
- 4 4. 运行./main输出Cmake字符串
- 5 5. 输入make clean可以删除main文件

2. 同一目录下多个源文件

2.1. 目录结构

```
├── CMakeLists.txt
├── main.c
├── testFunc1.c
├── testFunc1.h
├── testFunc.c
└── testFunc.h
```

2.2. 编写程序

2.2.1. main.c

```
1 | 选取main1.c
```

2.2.2. CMakeLists.txt

```
1 | cmake_minimum_required (VERSION 3.10)
2 | project (demo)
3 | add_executable(main main.c testFunc.c) #在add_executable的参数里把
   | testFunc.c加进来
```

2.3. 构建与运行

```
1 | 照样是根目录打开终端: cmake .→make→./main
```

2.4. 补充

2.4.1. aux_source_directory(dir var) 指令

含义详见附页，这里给出一个示例，将上述 CMakeLists.txt 改为如下，照样可以运行

```
1 | cmake_minimum_required (VERSION 3.10)
2 | project (demo)
3 | aux_source_directory(. SRC_LIST)
4 | add_executable(main ${SRC_LIST})
```

2.4.2. Set 指令

含义详见附页，这里给出一个示例，将上述 CMakeLists.txt 改为如下，照样可以运行

```
1 | cmake_minimum_required (VERSION 3.10)
2 | project (demo)
3 | set( SRC_LIST
4 |     ./main.c
5 |     ./testFunc.c)
6 | add_executable(main ${SRC_LIST})
```

至于要不要加 ./testFunc.h，结果是加不加都一样

3. 不同目录下多个源文件

3.1. 目录结构

```
.
├── CMakeLists.txt
├── main.c
├── test_func
│   ├── testFunc.c
│   └── testFunc.h
├── test_func1
│   └── testFunc1.c
└── testFunc1.h
```

3.2. 程序编写

3.2.1. main.c

```
1  /*方法1: 选取main2.c*/
2  /*方法2*/
3  #include "test_func/testFunc.h"
4  #include "test_func1/testFunc1.h"
5  #include <stdio.h>
6  int main(void)
7  {
8      func(100);
9      func1(200);
10     return 0;
11 }
```

3.2.2. CMakeLists.txt

```
1  cmake_minimum_required (VERSION 3.10)
2  project (demo)
3  include_directories (test_func test_func1)
4  aux_source_directory (test_func SRC_LIST)
5  aux_source_directory (test_func1 SRC_LIST1)
6  add_executable (main main.c ${SRC_LIST} ${SRC_LIST1})
```

方法一必须需要 `include_directories (test_func test_func1)`

方法二就不一定了

4. 更正规的组织方式

4.1. 目录组织结构

```
.
├── bin
├── build
├── CMakeLists.txt
├── include
│   ├── testFunc1.h
│   └── testFunc.h
└── src
    ├── CMakeLists.txt
    ├── main.c
    ├── testFunc1.c
    └── testFunc.c
```

4.2. 编写程序

4.2.1. main.c

```
1 | 选取main2.c
```

4.2.2. 建立两个CMakeLists.txt

①最外层目录下的CMakeLists.txt

```
1 | cmake_minimum_required (VERSION 3.10)
2 | project (demo)
3 | add_subdirectory (src)
```

②./src目录下的CMakeLists.txt

```
1 | aux_source_directory (. SRC_LIST)
2 | include_directories (./include)
3 | add_executable (main ${SRC_LIST})
4 | set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
```

4.3. 程序运行

1. 记得要在./build目录下打开终端运行Cmake: 这样终端的指令变为了cmake .., 这是防止cmake运行时生成的附带文件就会跟源码文件混在一起。Makefile会在./build目录下生成。
2. 在./build目录下运行make: 这样可执行文件main就生成在了./bin文件夹中
3. 切换到./bin目录执行main

4.4. 补充: CMakeLists.txt的另一种写法

只用一个最外层的CMakeLists.txt, 对比两种写法

```

1 | ./DHY/CMakeLists.txt
2 | cmake_minimum_required (VERSION 3.10)
3 | project (demo)
4 | add_subdirectory (src)
5 |
6 | ./DHY/src/CMakeLists.txt
7 | aux_source_directory (. SRC_LIST)#####核心不同点
8 | include_directories (../include)
9 | add_executable (main ${SRC_LIST})
10 | set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

```

```

1 | ./DHY/CMakeLists.txt
2 | cmake_minimum_required (VERSION 3.10)
3 | project (demo)
4 | aux_source_directory (src SRC_LIST)#####核心不同点
5 | include_directories (include)
6 | add_executable (main ${SRC_LIST})
7 | set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

```

注意:

```

1 | ../include , 指向当前CMakeLists.txt文件所在目录的上一级目录中的include子目
   | 录
2 | include , 直接指向当前CMakeLists.txt文件所在目录下的include子目录
3 | 然后:
4 | 记得把./src目录下的CMakeLists.txt删除, 程序运行过程相同

```

5. 动态库和静态库的编译控制

- 1 | 有时只需要编译出动态库和静态库, 然后等着让其它程序去使用。现在只留下testFunc.h和TestFunc.c
- 2 | 一些每次的释义间附页

5.1. 目录结构

```

.
├── build
├── CMakeLists.txt
├── lib
└── testFunc
    ├── testFunc.c
    └── testFunc.h

```

3 directories, 3 files

5.2. CMakeLists.txt

```
1 cmake_minimum_required (VERSION 3.10)
2 project (demo)
3 set (SRC_LIST ${PROJECT_SOURCE_DIR}/testFunc/testFunc.c)
4 add_library (testFunc_shared SHARED ${SRC_LIST})
5 add_library (testFunc_static STATIC ${SRC_LIST})
6 set_target_properties (testFunc_shared PROPERTIES OUTPUT_NAME
  "testFunc")
7 set_target_properties (testFunc_static PROPERTIES OUTPUT_NAME
  "testFunc")
8 set (LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
```

5.3. 程序运行

```
1 在./build目录cmake ..然后再make，回到./lib目录，成功生成了动态/静态库：
  libtestFunc.a libtestFunc.so
```

5.4. PS

```
1 set_target_properties重定义了库的输出名称，
2 如不用set_target_properties那么库名就是add_library里定义的名称（第一个参
  数）
3 但是连续2次使用add_library指定库名称时，这个名称不能相同
4 然而set_target_properties可以把名称设置为相同
5 但但但是这样最终生成的库文件后缀不同（.so/.a），就像这里一样
```

6. 对库进行链接

6.1. 目录结构

```
— bin
— build
— CMakeLists.txt
— src
  — main.c
— testFunc
  — inc
    — testFunc.h
  — lib
    — libtestFunc.a
    — libtestFunc.so
```

6 directories, 5 files

6.2. 编写程序

6.2.1. main.c

```
1 选取main1.c
```

6.2.2. CMakeLists.txt

```
1 cmake_minimum_required (VERSION 3.10)
2 project (demo)
3 set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
4 set (SRC_LIST ${PROJECT_SOURCE_DIR}/src/main.c)
5 include_directories (${PROJECT_SOURCE_DIR}/testFunc/inc) # find
  testFunc.h
6 find_library(TESTFUNC_LIB testFunc HINTS
  ${PROJECT_SOURCE_DIR}/testFunc/lib)
7 add_executable (main ${SRC_LIST})
8 target_link_libraries (main ${TESTFUNC_LIB})
```

6.3. 运行

- 1 ./build目录下，然后运行cmake .. 然后make
- 2 进入到./bin目录发现main已经生成，运行
- 3 PS: 使用find_library的好处是在执行cmake ..时就会去查找库是否存在，这样可以提前发现错误，不用等到链接时。

6.4. 其它

- 1 在./lib目录下有`testFunc`的静态库和动态库
- 2 find_library(TESTFUNC_LIB testFunc HINTS
 \${PROJECT_SOURCE_DIR}/testFunc/lib}默认是查找动态库
- 3
- 4 如果想直接指定使用动态库还是静态库，可以写成
- 5 find_library(TESTFUNC_LIB libtestFunc.so
 HINTS\${PROJECT_SOURCE_DIR}/testFunc/lib}
- 6 或者
- 7 find_library(TESTFUNC_LIB libtestFunc.a HINTS
 \${PROJECT_SOURCE_DIR}/testFunc/lib}

附页

附 I ： CMakeLists指令and示例

I.1. cmake_minimum_required ()

- 1 指定Cmake的最低版本
- 2 1.cmake_minimum_required (VERSION 3.10): 意思是表示cmake的最低版本要求是3.10，这个版本不能太低否则会报错

I.2. project ()

- 1 作用仅限于指定工程名
- 2
- 3 1.project (demo): 工程名叫demo

I.3. add_executable()

- 1 指定所需的源文件
- 2
- 3 1.add_executable(main main.c): 最终要生成的elf文件的名字叫main, 使用的源文件是main.c
- 4 2.add_executable(main main.c testFunc.c): 使用同一目录下多个源文件main.c和testFunc.c
- 5 3.add_executable(main \${SRC_LIST}): SRC_LIST是一个存有源文件列表和主程序的变量, 这段的意思是, 调用然后使用里面的源文件
- 6 4.add_executable (main main.c \${SRC_LIST} \${SRC_LIST1}): 使用main.c和调用SRC_LIST和SRC_LIST1中的源文件

I.4. aux_source_directory(dir var)

- 1 把指定目录下所有的源文件存储在一个变量中, 参数dir是指定目录, 第二个参数var是用于存放源文件列表的变量, 其弊端在于会把指定目录下的所有源文件都加进来
- 2
- 3 1.aux_source_directory(. SRC_LIST): 把当前目录下所有源文件存列表存放到变量SRC_LIST里
- 4 2.aux_source_directory (test_func1 SRC_LIST1): 将test_func1目录下的源文件加到变量SRC_LIST中

I.5. Set()

- 1 用于设置一个变量的值
- 2
- 3
- 4 1.如下: 新建变量来存放需要的源文件
- 5 set(SRC_LIST
- 6 ./main.c
- 7 ./testFunc.c)
- 8 - 定义了变量SRC_LIST, 包含源文件main.c和testFunc.c
- 9
- 10 2.如下: 设置变量的值来改变二进制文件存储位置
- 11 set (EXECUTABLE_OUTPUT_PATH \${PROJECT_SOURCE_DIR}/bin)
- 12 # EXECUTABLE_OUTPUT_PATH: 目标二进制可执行文件的存放位置
- 13 # PROJECT_SOURCE_DIR: 工程的根目录
- 14 # 二者皆为内置变量
- 15 - 此命令将所有可执行文件的输出目录设置为项目源目录下的 ./bin子目录。
- 16
- 17 3. 如下: 新建变量来存放文件路径
- 18 set (SRC_LIST \${PROJECT_SOURCE_DIR}/testFunc/testFunc.c)
- 19 # PROJECT_SOURCE_DIR: 工程的根目录
- 20 - 定义一个变量SRC_LIST, 并设置其值为源文件testFunc.c的路径, 该文件位于 \${PROJECT_SOURCE_DIR}/testFunc目录下
- 21
- 22 4.如下: 设置变量的值来指导库文件的输出目录
- 23 set (LIBRARY_OUTPUT_PATH \${PROJECT_SOURCE_DIR}/lib)
- 24 # PROJECT_SOURCE_DIR: 工程的根目录
- 25 # LIBRARY_OUTPUT_PATH: 库文件的默认输出路径, 这里设置为工程目录下的lib目录
- 26
- 27 5.如下: 设置变量的值来指导可执行文件的输出目录


```
28 set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)
29 # PROJECT_SOURCE_DIR: 工程的根目录
30 # EXECUTABLE_OUTPUT_PATH: 可执行文件的默认输出路径, 这里设置为工程目录下的
lib目录
```

I.6. include_directories()

```
1 向工程添加多个指定头文件的搜索路径, 路径之间用空格分隔
2
3 1.include_directories (test_func test_func1):
4 考虑这种情况, testFunc.h和testFunc1.h分别在test_func test_func1文件夹中,
而且main.c里包含了testFunc.h和testFunc1.h。如果没这条语句指定头文件的搜索路
径就会无法编译
5 2.include_directories (../include):
6 为编译过程添加包含目录(中的头文件), ../include指向当前CMakeLists.txt文件所
在目录的上一级目录中的include子目录, 将在该目录下查找头文件, 由此#include指令
可以直接引用此目录中的头文件。
7 3.include_directories (${PROJECT_SOURCE_DIR}/testFunc/inc):
8 也可以指定一个目录, 使编译器在这个目录下查找头文件
```

I.7. add_subdirectory()

```
1 完整格式是add_subdirectory(source_dir [binary_dir]
[EXCLUDE_FROM_ALL])
2 - source_dir: 子目录的路径, 其中包含要添加到构建过程中的另一个CMakeLists.txt
3 - binary_dir: 是可选参数, 用于指定子目录中生成的二进制文件make的存放位置。如果
不提供此参数, CMake将使用默认的二进制目录。
4 - EXCLUDE_FROM_ALL: 是可选标志, 如果设置则默执行常规构建时, 这个子目录不会被构
建, 除非明确请求。
5
6 1.add_subdirectory(src): 命令告诉CMake在./src目录下查找另一个
CMakeLists.txt文件并执行其中的指令
7 2.add_subdirectory(src build_dir): CMake将查找src目录下的
CMakeLists.txt并执行其中的指令。所有由这个子CMakeLists.txt文件产生的二进制
文件将被放在build_dir中。
```

I.8. add_library ()

```
1 定义并创建一个新的库
2
3 1.add_library (testFunc_shared SHARED ${SRC_LIST}): 定义一个新的动态库
testFunc_shared。这个库将从SRC_LIST变量(变量储存的是一个路径)中指定的源文件
编译而来
4 2.add_library (testFunc_static STATIC ${SRC_LIST}): 定义一个新的静态库
testFunc_static。这个库会从SRC_LIST变量(变量储存的是一个路径)中指定的源文件
编译而来
```

I.9. `set_target_properties()`

- 1 用于设置特定目标（如库或可执行文件）的属性和属性值
- 2
- 3 `1.set_target_properties (testFunc_shared PROPERTIES OUTPUT_NAME "testFunc")`: 设置testFunc_shared库的属性，使其输出文件名为testFunc，而不是默认 testFunc_shared

I.10. `find_library()`

- 1 顾名思义，查找库
- 2
- 3 `1.find_library(TESTFUNC_LIB testFunc HINTS ${PROJECT_SOURCE_DIR}/testFunc/lib)`:
- 4 在`${PROJECT_SOURCE_DIR}/testFunc/lib`目录下查找名为testFunc的库，并将其路径存储在TESTFUNC_LIB变量中
- 5 PS: 当使用HINTS提供路径时，CMake会优先在这些路径中搜索库，如果没有找到，CMake会继续在默认的库路径中搜索

I.11. `target_link_libraries()`

- 1 `1.target_link_libraries (main ${TESTFUNC_LIB})`: TESTFUNC_LIB变量中包含了testFunc库的路径，该指令是将先前找到的testFunc库链接到main可执行文件。

附II：名词释义

II.1. 静态库

- 1 1. 一个文件集合，包含一组编译但未经链接的代码\数据
- 2 2. 通常有.a (Unix\linux\macOS) 或.lib (windows) 扩展名
- 3 3. 从静态库链接程序时，库中代码\数据会被复制到可执行文件中

II.2. 动态库

- 1 1. 一个运行时（不是链接时）被加载的代码\数据集
- 2 2. 它们通常有.so (Unix\Linux) .dylib (macOS) 或.dll (windows) 扩展名
- 3 3. 从动态库中链接一个程序时，不是复制库的代码和数据，而是引用库的位置

II.3. 编译控制

- 1 1. 编译时指定编译器如何处理源代码，例如生成哪种类型的输出（可执行文件、静态库或动态库）
- 2 2. 在CMake的上下文中，编译控制是通过 `CMakeLists.txt` 文件中的命令来实现的。

附III：示例中重复用到的主程序

```
1 /*main1.c*/
2 #include <stdio.h>
3 #include "testFunc.h"
4 int main(void)
```

```

5  {
6      func(100);
7      return 0;
8  }
9  /*main2.c*/
10 #include <stdio.h>
11 #include "testFunc.h"
12 #include "testFunc1.h"
13 int main(void)
14 {
15     func(100);
16     func1(200);
17     return 0;
18 }

```

附IV：示例源文件与头文件程序

```

1  /*testFunc.c*/
2  #include <stdio.h>
3  #include "testFunc.h"
4  void func(int data)
5  {
6      printf("data is %d\n", data);
7  }
8
9  /*testFunc.h*/
10 #ifndef _TEST_FUNC_H_
11 #define _TEST_FUNC_H_
12 void func(int data);
13 #endif
14
15 /*testFunc1.h*/
16 #ifndef _TEST_FUNC1_H_
17 #define _TEST_FUNC1_H_
18 void func1(int data);
19 #endif
20
21 /*testFunc1.c*/
22 #include <stdio.h>
23 #include "testFunc1.h"
24 void func1(int data)
25 {
26     printf("data is %d\n", data);
27 }

```