

凌逆战

Never give up, become better yourself. Fight for my family!

博客园 首页 新随笔 联系 订阅 管理

目录导航

随笔 - 262 文章 - 0 评论 - 612 阅读 - 96万

公告

纸上得来终觉浅
绝知此事要躬行

这篇文章不错，不妨关注我吧，这是对我最大的鼓励，我将为你创造更加优秀的文章

鼓励一下：关注

昵称：凌逆战
园龄：5年8个月
粉丝：1449
关注：16
+加关注

搜索

找找看

随笔分类 (256)

语音信号处理(25)
语音频带扩展
(BWE)(18)
语音增强(SE)(38)

CMake 从入门到精通

目录

- 安装CMake
 - windows平台
 - linux平台
- CMake编译可执行文件
- 通过案例学习CMake编写
 - 同一目录多个源文件
 - 同一目录下多个源文件
 - 不同目录多个源文件
 - 生成动态库或静态库
 - 对库进行链接
 - 添加编译选项
 - 添加控制选项
- CMake常用指令
 - 添加Library
- CMake 控制指令
 - IF 指令
 - FOREACH 指令
 - WHILE 指令
- 参考

0

关注 | 顶部 | 评论

- 声学回声消除
(AEC)(9)
- 论文翻译(12)
- 深度学习(45)
- 机器学习(15)
- Python学习笔记
(34)
- C学习笔记(23)
- 生活总结(10)
- Web前后端(27)

阅读排行榜

1. librosa语音信号
处理(57903)
2. 声学回声消除(A
coustic Echo Canc
ellation)原理与实
现(50702)
3. 快速傅里叶变换
及python代码实现
(44244)
4. 语音信号的梅尔
频率倒谱系数(MF
CC)的原理讲解及p
ython实现(36267)
5. python做语音信
号处理(32597)
6. 波束形成算法综
述(28805)
7. 自适应滤波器算
法综述以及代码实
现(27651)
8. 主动噪声控制
(Active Noise Con

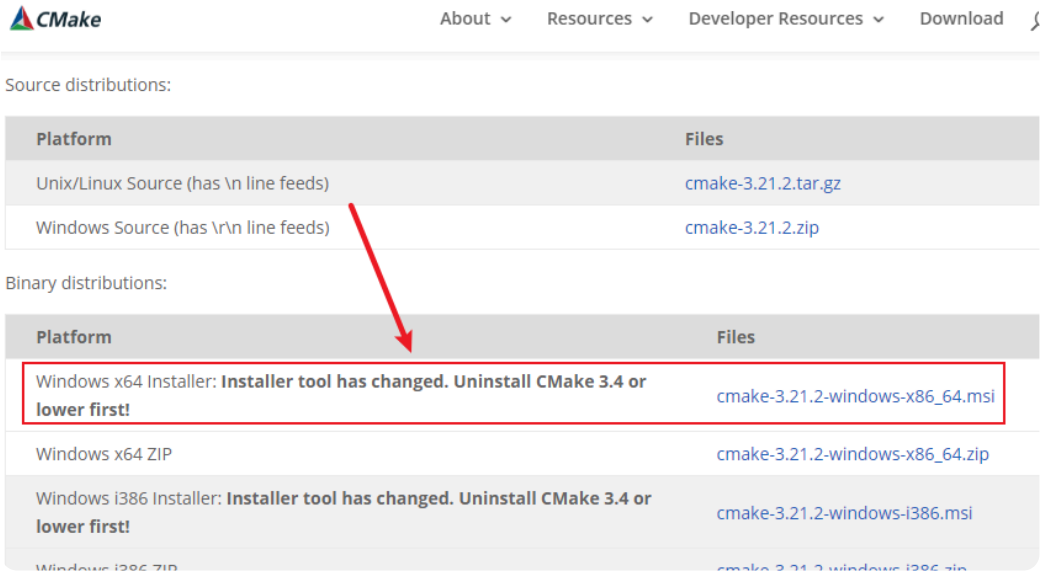
CMake 是一个跨平台的、开源的构建工
具。`cmake` 是 `makefile` 的上层工具，它们的目正是为了产生
可移植的 **makefile**，并简化自己动手写 **makefile**时的巨大工作
量。

目前很多开源的项目都可以通过CMake工具来轻松构建工程。

安装CMake

windows平台

1、根据自己想要下载的版本和系统位数，CMake官网下载
CMake：



2、后面一直点下一步就好了，这里推荐为所有用户添加CMake系
统路径

3、打开cmd，输入 `cmake -version`，出现版本号就算安装成功。

linux平台

本文使用ubuntu18.04，安装cmake使用如下命令，

```
sudo apt install cmake
```

安装完成后，在终端下输入 `cmake -version` 安
装成功。

CMake编译可执行文件

目
录
导
航

0

关注 | 顶部 | 评论

- trol, ANC)理论及Matlab代码实现(19712)
- 9. 语音数据增强及python实现(18054)
- 10. 全连接神经网络(DNN)(17106)

推荐排行榜

- 1. 电脑组装之硬件选择(170)
- 2. 声学回声消除(Acoustic Echo Cancellation)原理与实现(57)
- 3. 基于深度学习的回声消除系统与Pytorch实现(51)
- 4. 自适应滤波器算法综述以及代码实现(46)
- 5. python做语音信号处理(44)

最新评论

- 1. Re:快速傅里叶变换及python代码实现
写的很好，点赞
--不懂美食的吃货
- 2. Re:如何快速了解一个行业
好文,谢谢分享!

我们写好源代码和主函数，可以用命令行编译成可执行文件。如果我们直接编译成可执行文件给别人，可能会由于平台的不同直接给别人编译好的可执行文件可能跑不起来，举个栗子，我本地是Mac 电脑，我编译好的执行文件在Mac 电脑上跑没问题，但服务端同学是Linux 的就跑不起来。所以我们写好编译脚本把源码给他们，他们基于自己的平台编译就OK 了。

第一步：创建项目编译过程中代码存储的文件夹，并进入到文件夹内部

```
mkdir cmake-build-debug
cd cmake-build-debug\
```

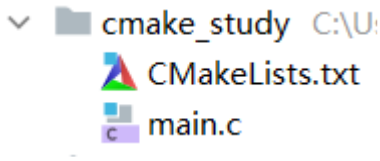
第二步：执行 `cmake ..` 表示构建上一级目录下 CMakeLists.txt 的配置，并在当前目录下生成 Makefile 等文件

```
cmake ..
```


第三步：执行 `cmake --build .` 或 `make` 将上步生成的 Makefile等文件生成可执行文件，后面就可以使用可执行文件了

通过案例学习CMake编写

同一目录多个源文件



编写 cmake 配置文件 `CMakeLists.txt` :



```
# cmake最低版本需求
cmake_minimum_required(VERSION 3.10)

# 工程名称
project (cmake_study)

# 设置C标准还是C++标准
set (CMAKE_C_STANDARD 11)
```

0

关注 | 顶部 | 评论

--afeizai

3. Re:如何快速了解一个行业牛！

--Linybo2008

4. Re:如何快速了解一个行业赞

--似风又似雨

5. Re:如何快速了解一个行业

@智客工坊👍...

--凌逆战

```
add_executable(cmake_study
    main.c)
```



为了不让编译产生的中间文件污染我们的工程，我们可以创建一个 `cmake-build-debug` 目录进入执行 `cmake` 构建工具。如果没有错误，执行成功后会在 `cmake-build-debug` 目录下产生 `Makefile` 文件。

`cmake` 命令便按照 `CMakeLists` 配置文件运行构建 `Makefile` 文件，然后我们执行 `make` 命令就可以编译我们的项目了。



```
$ mkdir cmake-build-debug
$ cd cmake-build-debug/
$ cmake ..

-rw-r--r--  1 root root 13591 Jul 20 12:09 CMakeCache.txt
drwxr-xr-x 14 root root   448 Jul 20 12:09 CMakeFiles
-rw-r--r--  1 root root  5034 Jul 20 12:09 Makefile
-rw-r--r--  1 root root  1508 Jul 20 12:09 cmake_install.cmake
-rwxr-xr-x  1 root root  9104 Jul 20 12:09 cmake_study

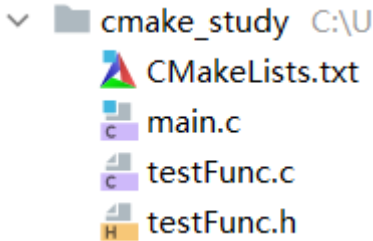
$ make
$ ./cmake_study
Hello, World!
```



以上就是大致的 `cmake` 构建运行过程。

从上面的过程可以看出，`cmake` 的重点在配置 `CMakeLists.txt` 文件。

同一目录下多个源文件





在当前目录下再添加2个文件，`testFunc.c`和`testFunc.h`


目录导航

0

关注 | 顶部 | 评论

 testFunc.c testFunc.h

修改main.c，调用testFunc.h里声明的函数func()，


 main.c

修改CMakeLists.txt，在add_executable的参数里把源文件 `testFunc.c` 加进来

```
cmake_minimum_required(VERSION 3.10)
project (cmake_study)

add_executable(cmake_study
               main.c testFunc.c)
```

如果源文件过多，一个一个添加显然不够高效，使用 `aux_source_directory` 把当前目录下的源文件存列表存放到列表变量里，然后在 `add_executable` 里调用列表变量。




```
cmake_minimum_required(VERSION 3.10)
project (cmake_study)

# 将当前文件夹下的源代码，收集到变量src_path
aux_source_directory(. src_path)

add_executable(cmake_study ${src_path})
```



`aux_source_directory()` 也存在弊端，它会把指定目录下的所有源文件都加进来，可能会加入一些我们不需要的文件，此时我们可以使用 `set` 命令新建变量来存放需要的源文件，如下，



```
cmake_minimum_required (VERSION 3.10)
project (cmake_study)

set( SRC_LIST
    ./main.c
    ./testFunc1.c
```

0

[关注](#) | [顶部](#) | [评论](#)

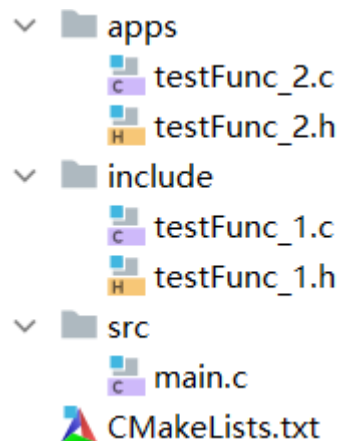
```
./testFunc.c)
```

```
add_executable(main ${SRC_LIST})
```



不同目录多个源文件

一般来说，当程序文件比较多时，我们会进行分类管理，把代码根据功能放在不同的目录下，这样方便查找。我们多个apps目录和include目录，



一般来说，我们可以在 main.c里使用include来指定路径，如下

```
#include "test_func/testFunc.h"
#include "test_func1/testFunc1.h"
```

只是这种写法不好看。我们可以使用CMake文件来添加路径



```
cmake_minimum_required(VERSION 3.10)
project(cmake_study C)

# 添加 头文件 的搜索路径
include_directories(include apps)
# 也可以分开写
#include_directories(include)
#include_directories(apps)

# 将路径的 源文件 收集到变量列表
aux_source_directory (include include_path)
aux_source_directory (apps app_path)
```

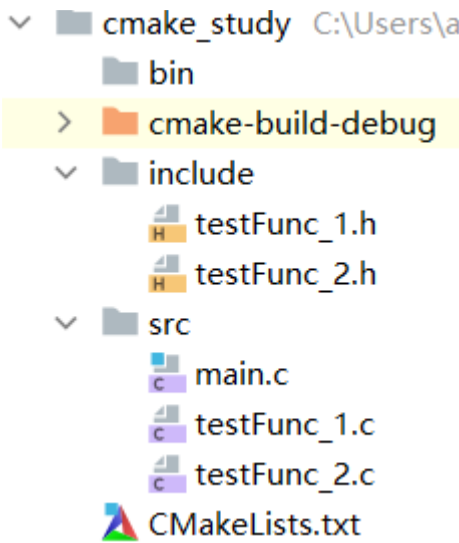
0

[关注](#) | [顶部](#) | [评论](#)

```
add_executable(cmake_study
               src/main.c ${include_path} ${app_path})
```



更多的情况是把头文件和源文件放在不同的文件夹中，如下



- **src:** 放 源文件
- **include:** 放 头文件
- **cmake-build-debug:** 放生成的对象文件
- **bin:** 放输出的文件

+ main.c

+ testFunc.c

+ testFunc1.c

+ testFunc.h

+ testFunc1.h

最外层的CMakeLists.txt文件



```
cmake_minimum_required (VERSION 3.19)
project (cmake_study)

# 定义变量
set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

# 向当前工程添加存放源文件的子目录
add_subdirectory (src)
```

0

关注 | 顶部 | 评论



`set`命令是用于定义变量的，`EXECUTABLE_OUT_PATH` 和 `PROJECT_SOURCE_DIR` 是CMake自带的预定义变量，其意义如下，

- `EXECUTABLE_OUTPUT_PATH`：目标二进制可执行文件的存放位置
- `PROJECT_SOURCE_DIR`：工程的根目录

所以，这里`set`的意思是把存放输出文件的位置设置为工程根目录下的`bin`目录

`add_subdirectory`命令，这个命令可以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的存放位置，具体用法可以百度。

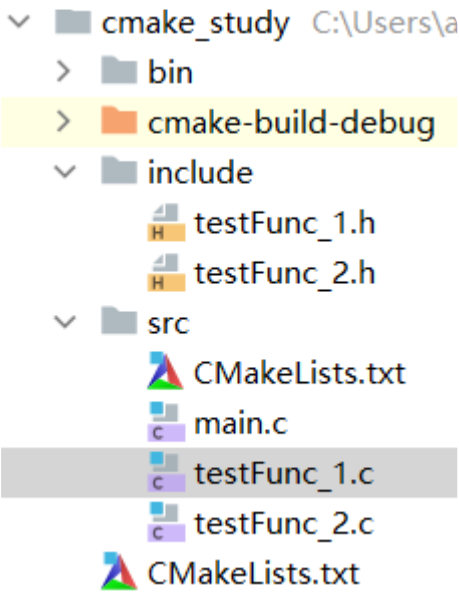
这里指定`src`目录下存放了源文件，当执行`cmake`时，就会进入`src`目录下去找`src`目录下的`CMakeLists.txt`，所以在`src`目录下也建立一个`CMakeLists.txt`，内容如下

```
include_directories (../include)

aux_source_directory (. SRC_LIST)

add_executable (main ${SRC_LIST})
```

添加好以上这2个`CMakeLists.txt`后，整体文件结构如下




0

关注 | 顶部 | 评论

下面来运行`cmake`，不过这次先让我们切到工程根目录下，然后输入命令 `cmake .`，`Makefile`会生成

`debug` 目录下生成，然后运行 `make`，再切到bin目录下，发现main已经生成，并运行测试，

当然，我们也可以在外层 只使用一个CMakeLists.txt




```
cmake_minimum_required (VERSION 3.19)
project (cmake_study)

set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

# 将源文件 添加到变量列表
aux_source_directory (src src_list)

# 添加头文件搜索路径
include_directories (include)

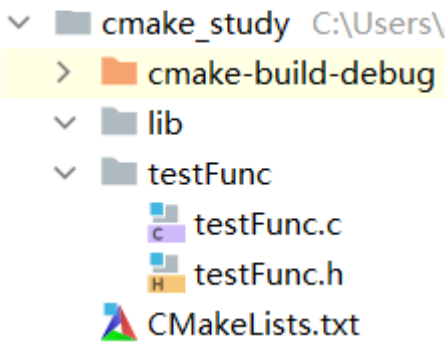
add_executable (main ${src_list})
```




生成动态库或静态库

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。

让我们看下这种情况该如何使用cmake。首先按照如下重新组织文件，只留下testFunc.h和TestFunc.c，



我们会在build目录下运行cmake，并把生成的库文件存放到lib目录下。CMakeLists.txt内容如下：



```
cmake_minimum_required(VERSION 3.10)
project (cmake_study)
```

0

关注 | 顶部 | 评论

```
# 新建变量SRC_LIST
set(SRC_LIST ${PROJECT_SOURCE_DIR}/testFunc/testFunc.c)

# 对 源文件变量 生成动态库 testFunc_shared
add_library(testFunc_shared SHARED ${SRC_LIST})
# 对 源文件变量 生成静态库 testFunc_static
add_library(testFunc_static STATIC ${SRC_LIST})

# 设置最终生成的库的名称
set_target_properties(testFunc_shared PROPERTIES OUTPUT_NAME "testFunc")
set_target_properties(testFunc_static PROPERTIES OUTPUT_NAME "testFunc")

# 设置库文件的输出路径
```



cmake文件解读：

- **add_library**：生成动态库或静态库
 - 第1个参数：指定库的名字
 - 第2个参数：决定是动态还是静态，如果没有就默认静态
 - 第3个参数：指定生成库的源文件
- **set_target_properties**：设置最终生成的库的名称，还有其它功能，如设置库的版本号等等
- **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

我们在cmake-build-debug目录下执行 `cmake ..` 和 `make` 。
这时候lib目录下生成了动态库 `libtestFunc.so` 和静态库 `libtestFunc.a` 。

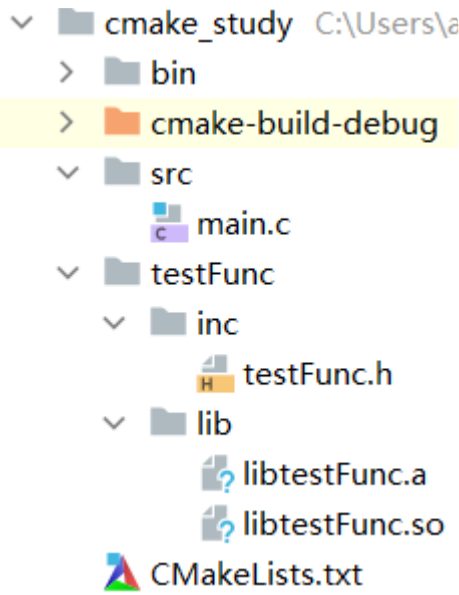
PS：前面使用set_target_properties重新定义了库的输出名称，如果不使用set_target_properties也可以，那么库的名称就是add_library里定义的名称，只是连续2次使用add_library指定库名称时（第一个参数），这个名称不能相同，而set_target_properties可以把名称设置为相同，只是最终生成的库文件后缀不同（是.a），这样相对来说会好看点。

对库进行链接

0

关注 | 顶部 | 评论

既然我们已经生成了库，那么就进行链接测试下。重新建一个工程目录，然后把上节生成的库拷贝过来，然后在在工程目录下新建src目录和bin目录，在src目录下添加一个main.c，整体结构如下：



+

main.c

工程目录下的CMakeLists.txt内容如下：

```
cmake_minimum_required(VERSION 3.10)
project(cmake_study)

# 输出bin文件路径
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

# 将源代码添加到变量
set(src_list ${PROJECT_SOURCE_DIR}/src/main.c)

# 添加头文件搜索路径
include_directories(${PROJECT_SOURCE_DIR}/testFunc/inc)

# 在指定路径下查找库，并把库的绝对路径存放到变量里
find_library(TESTFUNC_LIB testFunc HINTS ${PROJECT_SOURCE_DIR}/testFunc)

# 执行源文件
add_executable(main ${src_list})
```

0

关注 | 顶部 | 评论

把目标文件与库文件进行链接



cmake文件解读：

- **find_library**：在指定目录下查找指定库，并把库的绝对路径存放到变量里。会查找库是否存在，
 - 第一个参数：是变量名称
 - 第二个参数：是库名称
 - 第三个参数：是HINTS，提示
 - 第4个参数：是路径
 - 其它用法可以参考cmake文档
- **target_link_libraries**：把目标文件与库文件进行链接

我们在cmake-build-debug目录下执行 `cmake ..` 和 `make` 。最后进入到bin目录下查看，发现main已经生成

ps：在lib目录下有testFunc的静态库和动态库，find_library默认是查找动态库，如果想直接指定使用动态库还是静态库，可以写

成 `find_library(TESTFUNC_LIB libtestFunc.so ...)` 或者 `find_library(TESTFUNC_LIB libtestFunc.a ...)`

ps：查看elf文件使用了哪些库，可以使用 `readelf -d ./xx` 来看

ps：Linux查看某个库的位置/是否安装 `ldconfig -p | grep 库名`

说明：

- `ldconfig -p`：打印当前缓存所保存的所有库的名字。
- `grep libnccl`：用管道符解析libpcap.so是否已加入缓存中。

ldconfig主要是在默认搜寻目录/lib和/usr/lib以及动态库配置文件/etc/ld.so.conf内所列的目录下，搜索出可共享的动态链接库（格式如lib*.so*），进而创建出动态装入程序(ld.so)所需的连接和缓存文件。缓存文件默认为/etc/ld.so.cache，此文件保存了当前系统的动态链接库名字列表，为了让动态链接库为系统所识别，需要运行动态链接库的管理命令ldconfig，此执行程序在

添加编译选项

0

关注 | 顶部 | 评论

- `CMAKE_C_COMPILER`：指定C编译器
- `CMAKE_CXX_COMPILER`：指定C++编译器
- `CMAKE_C_FLAGS`：指定编译C文件时编译选项，也可以通过 `add_definitions`命令添加编译选项

在cmake脚本中，设置编译选项（配置编译器）有如下三种方法：

(1) `add_compile_options`命令

```
add_compile_options(-Wall -Werror -Wstrict-prototypes -Wmissing-prototy
```

(2) `add_definitions`命令

```
add_definitions("-Wall -Werror -Wstrict-prototypes -Wmissing-prototypes
```

(3) `set`命令修改`CMAKE_CXX_FLAGS`或`CMAKE_C_FLAGS`

```
set(CMAKE_C_FLAGS "-Wall -Werror -Wstrict-prototypes -Wmissing-prototy
```

使用这三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`命令和`add_definitions`添加的编译选项是针对所有编译器的(包括c和c++编译器)，

`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。

添加控制选项

有时希望在编译代码时只编译一些指定的源码，可以使用cmake的option命令，主要遇到的情况分为2种：

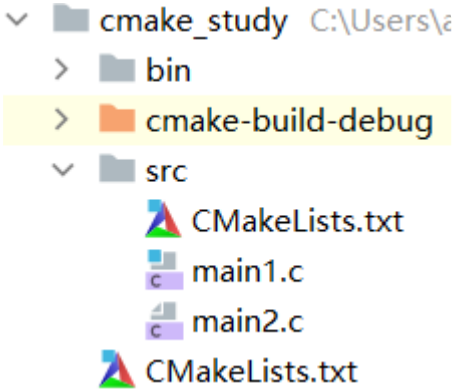
1. 本来要生成多个bin或库文件，现在只想生成部分指定的bin或库文件
2. 对于同一个bin文件，只想编译其中部分代码（使用宏来控制）

第一种情况

本来要生成多个bin或库文件，现在只想生成部分指定的bin或库文件。假设我们现在的工程会生成2个bin文件，现在整体结构体如下：

0

关注 | 顶部 | 评论



main1.c

main2.c

外层的CMakeLists.txt内容如下:



```
cmake_minimum_required(VERSION 3.10)
project(cmake_study)

# 描述选项
option(MYDEBUG "enable debug compilation" OFF)

# 设置输出bin的地址
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)


# 添加源文件的子目录
add_subdirectory(src)
```



option:

- 第一个参数: 是这个option的名字
- 第二个参数: 是字符串, 用来描述这个option是来干嘛的
- 第三个参数: 是option的值, ON或OFF, 也可以不写, 不写就是默认OFF

src目录下的CMakeLists.txt如下:



```
cmake_minimum_required(VERSION 3.10)
project(cmake_study)

# 执行源文件
```

0

关注 | 顶部 | 评论

```
add_executable(main1 main1.c)

if (MYDEBUG)
    add_executable(main2 main2.c)    # 执行源文件
else()
    message(STATUS "Currently is not in debug mode")
endif()
```



这里使用了if-else来根据option来决定是否编译main2.c，如果想编译出main2，就把MYDEBUG设置为ON，或者通过cmake命令

令 `cmake .. -DMYDEBUG=ON`

然后cd到build目录下输入 `cmake .. && make` 就可以只编译出main1。

第二种情况

对于同一个bin文件，只想编译其中部分代码（使用宏来控制）。假设我们有个main.c，其内容如下，



```
#include <stdio.h>

int main(void) {
#ifdef WWW1
    printf("hello world1\n");
#endif

#ifdef WWW2
    printf("hello world2\n");
#endif

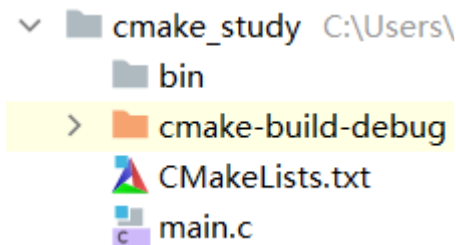
    return 0;
}
```



目录结构如下：

0

[关注](#) | [顶部](#) | [评论](#)



可以通过定义宏来控制打印的信息，我们CMakeLists.txt内容如下：

```
cmake_minimum_required(VERSION 3.10)
project(cmake_study)

# 设置输出bin文件的地址
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

# 设置选项WWW1和WWW2，默认关闭
option(WWW1 "print one message" OFF)
option(WWW2 "print another message" OFF)

if (WWW1)
    add_compile_options(-DWWW1)
endif ()

if (WWW2)
    add_compile_options(-DWWW2)
endif ()

# 执行源文件
add_executable(main main.c)
```

cd到build目录下执行cmake .. && make，然后到bin目录下执行./main，可以看到打印为空，

接着分别按照下面指令去执行，然后查看打印效果，

- cmake .. -DWWW1=ON -DWWW2=OFF && make
- cmake .. -DWWW1=OFF -DWWW2=ON && make
- cmake .. -DWWW1=ON -DWWW2=ON && make

0

[关注](#) | [顶部](#) | [评论](#)

这里有个小坑要注意下：假设使用cmake设置了一个option叫A，下次再设置别的option例如B，如果没有删除上次执行cmake时产生的缓存文件，那么这次虽然没设置A，也会默认使用A上次的option值。

所以如果option有变化，要么删除上次执行cmake时产生的缓存文件，要么把所有的option都显式的指定其值。

CMake常用指令

cmake会自动定义两个变量

- `${PROJECT_SOURCE_DIR}`：当前工程最上层的目录
- `${PROJECT_BINARY_DIR}`：当前工程的构建目录

`cmake_minimum_required(VERSION *.*)`：定义cmake的最低兼容版本。

`project(****)`：工程名称

`set(CMAKE_C_STANDARD 11)`：指定C语言标准

`set(CMAKE_CXX_STANDARD 11)`：指定C++语言标准

include_directories

`include_directories([AFTER|BEFORE] [SYSTEM] dir1 [dir2 ...])`

添加路径到头文件的搜索路径

add_subdirectory

`add_subdirectory(NAME)`

添加一个文件夹进行编译，该文件夹下的编译该文件夹下的源码。NAME是想对于调用CMakeListst.txt的相对路径。

add_executable

0

关注 | 顶部 | 评论

```
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
                 [EXCLUDE_FROM_ALL]
                 source1 [source2 ...])
```

利用源码文件生成目标可执行程序。

- **name:** 工程所要构建的目标名称
- **WIN32/...:** 目标app运行的平台(可忽略)
- **source1:** 构建目标App的源文件

添加Library

现在我们尝试添加一个library到我们的工程。这个lib提供一个自定义的计算平方根的函数，用来替换编译器提供的函数。

lib的源文件放到一个叫MathFunctions的子目录中，在目录下新建CMakeList.txt文件，添加如下的一行

```
# 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)
# 生成链接库
add_library (MathFunctions ${DIR_LIB_SRCS})
```

源文件mysqrt.cxx包含一个函数mysqrt用于计算平方根。代码如下

⊕

mysqrt.cxx

还需要添加一个头文件MathFunctions.h以提供接口给main函数调用

```
double mysqrt(double x);
```

现在的目录结构



```
.
├── build
├── CMakeLists.txt
├── MathFunctions
│   ├── CMakeLists.txt
│   ├── MathFunctions.h
│   └── mysqrt.cxx
```

0

关注 | 顶部 | 评论

TutorialConfig.h.in

tutorial.cxx



CMakeLists.txt文件需要相应做如下改动

- 添加一行`add_subdirectory`来保证新加的library在工程构建过程中被编译。
- 添加新的头文件搜索路径`MathFunction/MathFunctions.h`。
- 添加新的library到`executable`。



```
include_directories("${PROJECT_SOURCE_DIR}/MathFunctions")
add_subdirectory(MathFunctions)

# 添加executable
add_executable(Demo main.cxx)
# 使用命令 target_link_libraries 指明可执行文件 main 需要连接一个名为 MathFunctions
target_link_libraries(Demo MathFunctions)
```



add_definitions

```
add_definitions(-DENABLE_DEBUG -DABC)
```

向 C/C++编译器添加 -D 定义。如果你的代码中定义了`#ifdef ENABLE_DEBUG #endif`，这个代码块就会生效。

add_dependencies

```
add_dependencies(target-name depend-target1 depend-target2)
```

定义 `target` 依赖的其他 `target`，确保在编译本 `target` 之前，依赖的 `target` 已经被构建。

0

关注 | 顶部 | 评论

aux_source_directory

```
aux_source_directory(dir VARIABLE)
```

查找dir目录下所有的源码文件，存储在一个变量中。在add_executable的时候就不用一个一个的源码文件添加了，例如：

```
aux_source_directory(. SRC_LIST)
add_executable(main ${SRC_LIST})
```

find_package

```
find_package(<PackageName> [version] [EXACT] [QUIET] [MODULE] [REQUIREI
```

查找并从外部项目加载设置。<PackageName>_FOUND 将设置为指示是否找到该软件包。找到软件包后，将通过软件包本身记录的变量和“导入的目标”提供特定于软件包的信息。该QUIET选项禁用信息性消息，包括那些如果未找到则表示无法找到软件包的消息REQUIRED。REQUIRED如果找不到软件包，该选项将停止处理并显示一条错误消息。

COMPONENTS选项后（或REQUIRED选项后，如果有的话）可能会列出所需组件的特定于包装的列表。后面可能会列出其他可选组件OPTIONAL_COMPONENTS。可用组件及其对是否认为找到包的影响由目标包定义。

link_libraries

```
link_libraries([item1 [item2 [...]] [[debug|optimized|general] <item>]
```

将库链接到以后添加的所有目标。

add_library

```
add_library(<name> [STATIC | SHARED | MODULE] [source1] [source2 ...])
```

根据源码文件生成目标库。

STATIC，SHARED 或者 MODULE 可以指定目标库的链接类型。STATIC库是链接其他目标时使用的目标文件的静态版本，是静态链接的，并在运行时加载

0

[关注](#) | [顶部](#) | [评论](#)

enable_testing

```
enable_testing()
```

控制 Makefile 是否构建 test 目标，涉及工程所有目录。一般情况这个指令放在工程的主CMakeLists.txt 中。

add_test

```
add_test(testname Exename arg1 arg2 ...)
```

testname 是自定义的 test 名称，Exename 可以是构建的目标文件也可以是外部脚本等等。后面连接传递给可执行文件的参数。如果没有在同一个 CMakeLists.txt 中打开ENABLE_TESTING()指令，任何 ADD_TEST 都是无效的。

exec_program

在 CMakeLists.txt 处理过程中执行命令，并不会在生成的 Makefile 中执行。具体语法为：

```
exec_program(Executable [directory in which to run]
             [ARGS <arguments to executable>]
             [OUTPUT_VARIABLE <var>]
             [RETURN_VALUE <var>])
```

用于在指定的目录运行某个程序，通过 ARGS 添加参数，如果要获取输出和返回值，可通过OUTPUT_VARIABLE 和 RETURN_VALUE 分别定义两个变量。

这个指令可以帮助你 CMakeLists.txt 处理过程中支持任何命令，比如根据系统情况去修改代码文件等等。

FILE 指令

文件操作指令



```
FILE(WRITE filename "message to write"... )
FILE(APPEND filename "message to write"... )
FILE(READ filename variable)
FILE(GLOB variable [RELATIVE path] [globbing expression]...)
```

0

[关注](#) | [顶部](#) | [评论](#)

```
FILE(GLOB_RECURSE variable [RELATIVE path] [globbing expression_r_rs]).
FILE(REMOVE [directory]...)
FILE(REMOVE_RECURSE [directory]...)
FILE(MAKE_DIRECTORY [directory]...)
FILE(RELATIVE_PATH variable directory file)
FILE(TO_CMAKE_PATH path result)
```



CMake 控制指令

IF 指令



```
if(<condition>)
    <commands>
elseif(<condition>) # optional block, can be repeated
    <commands>
else()              # optional block
    <commands>
endif()
```

#####

IF(var),如果变量不是:空,0,N, NO, OFF, FALSE, NOTFOUND 或<var>_NOTFOUND 时,
IF(NOT var),与上述条件相反。
IF(var1 AND var2),当两个变量都为真是为真。
IF(var1 OR var2),当两个变量其中一个为真时为真。
IF(COMMAND cmd),当给定的 cmd 确实是命令并可以调用是为真。
IF(EXISTS dir) 或者 IF(EXISTS file),当目录名或者文件名存在时为真。
IF(file1 IS_NEWER_THAN file2),当 file1 比 file2 新,或者 file1/file2 其中
IF(IS_DIRECTORY dirname),当 dirname 是目录时,为真。
IF(variable MATCHES regex)
IF(string MATCHES regex)



FOREACH 指令

```
foreach(<loop_var> <items>)
    <commands>
endforeach()
```

0

关注 | 顶部 | 评论

其中 `<items>` 是以分号或空格分隔的项目列表。记录`foreach`匹配和匹配之间的所有命令`endforeach`而不调用。一旦`endforeach`评估，命令的记录列表中的每个项目调用一次 `<items>`。在每次迭代开始时，变量`loop_var`将设置为当前项的值。

WHILE 指令

```
while(<condition>)  
    <commands>  
endwhile()
```

`while`和匹配之间的所有命令 `endwhile()`被记录而不被调用。一旦`endwhile()`如果被评估，则只要为 `<condition>` 真，就会调用记录的命令列表。

参考

【CSDN】Linux下CMake简明教程

CMake设置编译选项的几种方法

以后在学

<https://www.jianshu.com/p/6df3857462cd>

作者：凌逆战

欢迎任何形式的转载，但请务必注明出处。

限于本人水平，如果文章和代码有表述不当之处，还请不吝赐教。

本文章不做任何商业用途，仅作为自学所用，文章后面会有参考链接，我可能会复制原作者的话，如果介意，我会修改或者删除。

分类：C学习笔记

好文要顶

关注我

收藏该文

0

关注 | 顶部 | 评论



凌逆战
粉丝 - 1449 关注 - 16
会员号: 382

+加关注

« 上一篇: 论文翻译: 2020_TinyLSTMs: Efficient Neural Speech Enhancement for Hearing Aids
» 下一篇: 语音领域视频及书籍(不定期更新)

目
录
导
航

posted @ 2022-05-02 17:35 凌逆战 阅读(1904) 评论(0) 编辑 收藏 举报

登录后才能查看或发表评论, 立即 [登录](#) 或者 [逛逛](#) 博客园首页

Copyright © 2024 凌逆战

Powered by .NET 8.0 on Kubernetes

0

[关注](#) | [顶部](#) | [评论](#)