CMake

CMake与MakeFIle都是项目构建工具，都具有跨平台性。使用CMake或者MakeFile，其实就是告诉计算机怎么对复杂进行编译、链接等操作。其中CMake的跨平台性使其比MakeFIle更加易用。

代码源文件（例如C与C++的源代码文件）生成可执行文件的过程是由一系列工具链完成的:

1. 预处理：比如去除注释，替换宏定义，展开头文件等等，源文件还未编译；
2. 编译：编译获得编译后的文件；
3. 汇编：汇编完成后获得多个二进制文件，例如.obj与.o；
4. 链接：链接后将各种二进制文件链接为一个二进制文件，相当于一个打包操作，打包获得一个可执行文件。
5. 如果遇到外部库导入的permission denied，请把可执行文件和该库放在同一个文件夹。

https://www.bilibili.com/video/BV14s4y1g7Zj/?p=10&spm_id_from=pageDriver&vd_source=dd00948d861cc6ba9d0db9eca667ed49

▼ 1 使用流程

使用CMake，平台上必须要安装一个CMake工具，检测是否安装了CMake如下：

```
cmake --version
```

1. 在项目根目录，创建一个文件，名为`CMakeLists.txt`
2. 配置cmake，生成makefile文件；
3. 生成cmake，执行make命令；

2 基础语法

2.1 程序框架

假设只有一个源文件，那么CMakeLists.txt中的内容如下：

```
cmake_minimum_required(VERSION 3.0) # 指定cmake的最低版本
project(ProjectName) # 指定构建的项目的名字，还可以传入其他参数，
add_executable(可执行程序名 源文件名称) # 源文件之间可以使用分号或
```

为了美观，我们把生成的文件都装在build目录里，也就是我们要cd进入build目录，在build目录里执行`cmake ..`

其次，`add_executable`最好放在程序的最后。

那么程序框架可以写成如下形式：

- **确定最低版本和项目名称；**
- **导入头文件**

- **导入静态库**
- **搜索源文件并生成可执行文件**
- **导入动态库，确定权限等级**

▼ 2.2 注释

- 行注释

```
# 这是一个行注释
```

- 块注释

```
#[[   这是一个块注释
        这是一个块注释
        这是一个块注释 ]]
```

▼ 2.3 配置、构建与执行

- 配置

```
cmake . # CMakeLists.txt 在当前目录
cmake .. # CMakeLists.txt 在上一目录
```

- 构建：实际上是运行配置后生成的MakeFile文件

```
make
```

- 执行：直接在shell窗口输入可执行程序名即可

```
可执行程序名 # 必须在可执行程序存在的目录下使用该命令
```

▼ 2.4 set命令

- 变量

使用set命令产生一个string类型变量，然后可以进行强制类型转换获得其他类型，大多数时候获得的都是字符串类型。

```
# 变量值可以为源文件、项目名等
SET(变量名 变量值 [CACHE TYPE DOCSTRING [FORCE]])

# way
SET(SRC_LIST add.c div.c main.c mult.c sub.c)

# 使用${变量名}访问变量值
add_executable(app &{SRC_LIST})
```

- 设置C++标准

```
SET(CMAKE_CXX_STANDARD 11)

SET(CMAKE_CXX_STANDARD 14)

SET(CMAKE_CXX_STANDARD 17)
```

  - 或者使用shell命令

```
cmake CMakeLists.txt文件路径 -DCMAKE_CXX_STANDARD=11
```

- 设置可执行文件存储的文件路径，建议使用绝对路径

```
SET(HOME /home/robin/linux/Sort)
SET(EXECUTABLE_OUPUT_PATH ${HOME}/bin)
```

▼ 2.5 搜索命令

当源代码数目太多，搜索命令可以用来搜索某路径下的源代码替代手写源代码名称：

- aux_source_directory

```
aux_source_directory(<dirPath> <variable>)

add_executable(app <variable>)

aux_source_directory(${PROJECT_SOURCE_DIR} SRC)
add_executable(app SRC)
```

- 当源文件不止在一个文件夹中时：
  - GLOB指定为当前的目录，GLOB_RECURSE指定为当前的目录和它的子目录。

```
file(GLOB/GLOB_RECURSE 变量名 要搜索的文件路径和文件类型后缀)

file(GLOB MAIN_SRC ${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp
```

▼ 2.6 指定头文件路径

如果说在源文件中使用了#include “Head.h”，但是该头文件在include文件夹中，那么编译这个源文件就会失败，因为这条指令是在该源文件的目录下寻找头文件。因此项目构建时需要重新制定头文件的搜索路径，如下：

```
include_directories(${PROJECT_SOURCE_DIR}/include)
```

▼ 3 库的创建和导入

▼ 3.1 创建库

- 制作静态库
  - linux里是libxxx.a，windows后缀为libxxx.lib；
  - 这里不生成一个可执行文件，所以程序框架不需要`add_executable(…)`。

```
add_library(库名称 STATIC 源文件1 [源文件2] ...)
```

- 制作动态库（拥有可执行权限）
  - linux里是libxxx.so，windows后缀为libxxx.dll；

    - 这里不生成一个可执行文件，所以程序框架不需要`add_executable(…)`。

```
add_library(库名称 SHARED 源文件1 [源文件2] ...)
```

- 使用库的方法
    - 发送两个数据：
        - 头文件
        - 库文件
    - 导入库和头文件

▼ 3.2 导入库

头文件里有库文件定义的一些函数、变量或者宏的声明，头文件的存在主要是为了代码的易读。导入库需要两个操作：

1. 指定include的路径；
2. 导入库并指定这个库的路径。

- **导入静态库**

```
link_libraries(<static lib> [<static lib> ...])
```

指定出全名是可以的，也可以只指定“掐头去尾”后的库名。如果是系统提供的库，则只需要指定名字。如果是自定义的，则还需要指定库的路径。

```
link_directories(<lib path> [<lib path>]) # 指定动态库、静
```

运行这些指令并生成可执行程序的时候，特指链接静态库，这些**静态库都会被打包到可执行程序中去。动态库内的数据不会被打包到可执行程序中，只有运行的时候才会被加载到内存。**

- **导入动态库**

```
target_link_libraries(<target>
                                                    <PRIVATE|PUB
                                                   [<PRIVATE|PUBLI
```

- target：**需要加载动态库的**文件的名称，可以是源文件、动态库文件也可以是可执行文件；
- **默认为PUBLIC**；如果是Private，那么就只能**就近链接**到前面的target中，只传递一次。例如：`target_link_libraries(a PRIVATE b PRIVATE c)`，那么b和c就只能链接到a中，此时再使用`target_link_libraries(d a)`，那么则只有a的内容可以被d访问，**b和c不能被链接到d中（也就是d不能直接使用b和c的内容，可以调用a，a可以调用b和c）**，第三方不知道你调用了哪个库；**interface能够隐藏调用的函数的来源，不会链接到target上，只会导出它的数据**。
- `target_link_libraries(a b c)`的意思就是a可以使用a，b，c的库，b可以使用c的库。**这个命令具有传递性；**
- **这个指令要写在add_executable的后面，因为dll文件不会和源文件一起被打包成可执行文件。**

▼ 4 宏

改变宏的值可以使用SET命令。

- CMAKE_CURRENT_SOURCE_DIR与PROJECT_SOURCE_DIR：执行CMake命令时，后面携带的路径（CMakeLists.txt所在的路径）；
- LIBRARY_OUTPUT_PATH：指定库的生成路径；
- EXECUTABLE_OUTPUT_PATH：指定可执行文件的生成路径，它也适用于动态库的生成路径指定；

▼ 5 项目案例