CMakeLists.txt

由 enlai.feng创建, 最后修改于三月 13, 2024

CMakeLists.txt 是由 CMake 构建系统使用的配置文件，用于定义项目的构建过程。CMake 是一个跨平台的自动化构建系统，它使用 CMakeLists.txt 文件来确定如何编译源代码和链接程序。

在 CMakeLists.txt 文件中，你可以指定包括但不限于以下内容：

```
project(MyProject VERSION 1.0)   //项目名称和版本：定义项目的名称和版本号
cmake_minimum_required(VERSION 3.10) //最小 CMake 版本：指定能够处理该 CMakeLists.txt 文件的最低 CMake 版本
add_definitions(-DMY_MACRO=1)    //编译器配置：设置编译器标志、定义宏等。
add_executable(MyExecutable main.cpp) //源文件和目标：指定用于构建目标（可执行文件、库等）的源文件。
target_link_libraries(MyExecutable PRIVATE mylib)   //库依赖关系：链接到其他库，无论是系统库还是项目中定义的库。
target_include_directories(MyExecutable PRIVATE include/)   //包含目录：指定编译器应该在哪里查找头文件。
add_subdirectory(src)   //子目录：如果项目组织为多个目录，可以添加子目录，这些目录中也可能包含它们自己的 CMakeLists
option(USE_MY_FEATURE "Use my feature" ON)   //构建选项：定义构建配置选项。
install(TARGETS MyExecutable DESTINATION bin)   //安装规则：定义如何安装程序（例如，复制可执行文件、库和资源到系统

enable_testing()
add_test(NAME MyTest COMMAND TestCommand)    //测试：添加测试用例，以便使用 CTest 测试工具进行测试。
```

CMakeLists.txt 文件通常位于项目的根目录中，并且可能引用子目录中的其他 CMakeLists.txt 文件，形成项目的构建层次结构。通过在命令行上运行 cmake 命令，CMake 会处理这个文件（和任何引用的文件），生成适合当前系统的构建文件（例如，Makefile 或 Visual Studio 项目文件）。\

- 编写CMakeLists.txt最常用的功能就是**调用其他的.h头文件和.so/.a库文件，将.cpp/.c/.cc文件编译成可执行文件或者新的库文件。**

```
# 本CMakeLists.txt的project名称
# 会自动创建两个变量，PROJECT_SOURCE_DIR和PROJECT_NAME
# ${PROJECT_SOURCE_DIR}：本CMakeLists.txt所在的文件夹路径
# ${PROJECT_NAME}：本CMakeLists.txt的project名称
project(xxx)

# 获取路径下所有的.cpp/.c/.cc文件，并赋值给变量中
aux_source_directory(路径 变量)

# 给文件名/路径名或其他字符串起别名，用${变量}获取变量内容
set(变量 文件名/路径/...)

# 添加编译选项
add_definitions(编译选项)

# 打印消息
message(消息)

# 编译子文件夹的CMakeLists.txt
add_subdirectory(子文件夹名称)

# 将.cpp/.c/.cc文件生成.a静态库
# 注意，库文件名称通常为libxxx.so，在这里只要写xxx即可
add_library(库文件名称 STATIC 文件)

# 将.cpp/.c/.cc文件生成可执行文件
add_executable(可执行文件名称 文件)

# 规定.h头文件路径
include_directories(路径)

# 规定.so/.a库文件路径
```

```
link_directories(路径)

# 对add_library或add_executable生成的文件进行链接操作
# 注意，库文件名称通常为libxxx.so，在这里只要写xxx即可
target_link_libraries(库文件名称/可执行文件名称 链接的库文件名称)
```

通常一个CMakeLists.txt需按照下面的流程：

```
project(xxx)                                          #必须

add_subdirectory(子文件夹名称)                         #父目录必须，子目录不必

add_library(库文件名称 STATIC 文件)                    #通常子目录(二选一)
add_executable(可执行文件名称 文件)                    #通常父目录(二选一)

include_directories(路径)                              #必须
link_directories(路径)                                 #必须

target_link_libraries(库文件名称/可执行文件名称 链接的库文件名称)        #必须
```

除了这些之外，就是些set变量的语句，if判断的语句，或者其他编译选项的语句，但基本结构都是这样的。

实例

实例的功能是生成和解析proto文件，分为C++和python版本。其中，C++版本就是采用CMakeLists.txt编写的，目录结构如下：

```
|---example_person.cpp
|---proto_pb2
        |--Person.pb.cc
        |--Person.pb.h
|---proto_buf
        |---General_buf_read.h
        |---General_buf_write.h
|---protobuf
        |---bin
                |---...
        |---include
                |---...
        |---lib
                |---...
```

目录结构含义：

- protobuf：Google提供的相关解析库和头文件，被proto_pb2文件夹内引用；
- proto_pb2：封装的Person结构和Person相关的处理函数，被proto_buf文件夹内引用；
- proto_buf：封装的read和write函数，被example_persom.cpp文件引用。

也就是说：

example_person.cpp–>proto_buf文件夹–>proto_pb2文件夹–>protobuf文件夹

步骤

CMakeLists.txt的创建

在需要进行编译的文件夹内编写CMakeLists.txt，即含有.cpp/.c/.cc的文件夹内：

即目录结构如下：

```
|---example_person.cpp
|---CMakeLists.txt
|---proto_pb2
```

```
        |--Person.pb.cc
        |--Person.pb.h
        |--CMakeLists.txt
|---proto_buf
        |---General_buf_read.h
        |---General_buf_write.h
|---protobuf
        |---bin
                |---...
        |---include
                |---...
        |---lib
                |---...
```

CMakeLists.txt的编写

本项目的`CMakeLists.txt`的文件数量是2个，目录层次结构为上下层关系。通常的解决方案，就是**将下层目录编译成一个静态库文件，让上层目录直接读取和调用，而上层目录就直接生成一个可执行文件。**

上层CMakeLists.txt的内容为：

```
cmake_minimum_required(VERSION 3.0)
project(example_person)

# 如果代码需要支持C++11，就直接加上这句
SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++0x")
# 如果想要生成的可执行文件拥有符号表，可以gdb调试，就直接加上这句
add_definitions("-Wall -g")

# 设置变量，下面的代码都可以用到
set(GOOGLE_PROTOBUF_DIR ${PROJECT_SOURCE_DIR}/protobuf)
set(PROTO_PB_DIR ${PROJECT_SOURCE_DIR}/proto_pb2)
set(PROTO_BUF_DIR ${PROJECT_SOURCE_DIR}/proto_buf)

# 编译子文件夹的CMakeLists.txt
add_subdirectory(proto_pb2)

# 规定.h头文件路径
include_directories(${PROJECT_SOURCE_DIR}
    ${PROTO_PB_DIR} ${PROTO_BUF_DIR}
)

# 生成可执行文件
add_executable(${PROJECT_NAME} example_person.cpp )

# 链接操作
target_link_libraries(${PROJECT_NAME}
    general_pb2)
```

这一段可能看不到两个地方，第一是链接操作的`general_pb2`，第二是按照上文的`CMakeLists.txt`的流程，并没有规定`link_directories`的库文件地址，这两个其实是一个道理，运行到`add_subdirectory`这一句时，会先将子文件夹进行编译，而`libgeneral_pb2.a`是在子文件夹中生成出来的库文件。子文件夹运行完后，父文件夹就已经知道了`libgeneral_pb2.a`这个库，因而不需要`link_directories`了。**add_subdirectory起到的作用。另一方面，在add_subdirector之前set的各个变量，在子文件夹中是可以调用的！**

下层CMakeLists.txt的内容为：

```
project(general_pb2)

aux_source_directory(${PROJECT_SOURCE_DIR} PB_FILES)

add_library(${PROJECT_NAME} STATIC ${PB_FILES})
```

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

link_directories(${GOOGLE_PROTOBUF_DIR}/lib/)

target_link_libraries(${PROJECT_NAME}
    protobuf
)
```

在这里，GOOGLE_PROTOBUF_DIR是上层CMakeLists.txt中定义的，libprotobuf.a是在${GOOGLE_PROTOBUF_DIR}/lib/目录下的。

显然可见，这就是一个标准的CMakeLixts.txt的流程。

CMakeLists.txt的编译

一般CMakeLists.txt是，**在最顶层创建build文件夹，然后编译。**即：

```
mkdir build && cd build
cmake ..
make
```

最终生成可执行文件example_person。

可以通过以下命令来运行该可执行文件：

```
./example_person
```