> 施磊老师网上课堂： **https://fixbug.ke.qq.com/**

CMake简介

使用简单方便，可以跨平台，构建项目编译环境。尤其比直接写Makefile简单（在构建大型工程编译时，需要写大量的文件依赖关系），可以通过简单的CMake生成负责的Makefile文件。

CMake安装

ubuntu上直接执行 **sudo apt install cmake** 安装完成，可以通过cmake -version查看其版本：

```
tony@tony-virtual-machine:~$ cmake -version
cmake version 3.10.2

CMake suite maintained and supported by Kitware (kitware.com/cmake).
```

CMake使用介绍

cmake命令会执行目录下的CMakeLists.txt配置文件里面的配置项，一个基本的CMakeLists.txt的配置文件内容如下：

```
cmake_minimum_required (VERSION 2.8)    #要求cmake最低的版本号
project (demo) # 定义当前工程名字
set(CMAKE_BUILD_TYPE "Debug")#设置debug模式，如果没有这一行将不能调试设断点
set(CMAKE_CXX_FLAGS ${CMAKE_CXX_FLAGS} -g)
add_executable(main main.c)

#进入子目录下执行 CMakeLists.txt文件    这里的lib和tests里面都有可编译的代码文件
add_subdirectory(lib)
add_subdirectory(tests)
```

示例一

生成一个main.cpp源文件，输出"hello world"，然后在同级目录创建一个CMakeLists.txt文件，内容如下：

```
cmake_minimum_required (VERSION 2.8)    #要求cmake最低的版本号
project (demo) # 定义当前工程名字
set(CMAKE_BUILD_TYPE "Debug")#设置debug模式，如果没有这一行将不能调试设断点
add_executable(main main.cpp)
```

保存退出，执行cmake .命令，输出如下：

```
 tony@tony-virtual-machine:~/code/cmake/rpc$ cmake .
 -- The C compiler identification is GNU 7.4.0
 -- The CXX compiler identification is GNU 7.4.0
 -- Check for working C compiler: /usr/bin/cc
 -- Check for working C compiler: /usr/bin/cc -- works
```

```
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/tony/code/cmake/rpc
```

ls查看目录，发现除了CMake生成的一些中间文件，还生成好了Makefile文件

```
tony@tony-virtual-machine:~/code/cmake/rpc$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  CMakeLists.txt  main.cpp  Makefile
```

make开始编译，最终生成可执行文件main

```
tony@tony-virtual-machine:~/code/cmake/rpc$ make
Scanning dependencies of target main
[ 50%] Building CXX object CMakeFiles/main.dir/main.cpp.o
[100%] Linking CXX executable main
[100%] Built target main
```

查看生成的可执行文件：

```
tony@tony-virtual-machine:~/code/cmake/rpc$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  CMakeLists.txt  main  main.cpp  Makefile
```

上面生成的Makefile里面实现了clean，所以make clean可以清除生成的文件，然后重新编译源码。

示例二

如果需要编译的有多个源文件，可以都添加到add_executable(main main.cpp test.cpp)列表当中，但是如果源文件太多，一个个添加到add_executable的源文件列表中，就太麻烦了，此时可以用**aux_source_directory(dir var)**来定义源文件列表，使用如下：

```
cmake_minimum_required (VERSION 2.8)
project (demo)
aux_source_directory(. SRC_LIST) # 定义变量，存储当前目录下的所有源文件
add_executable(main ${SRC_LIST})
```

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命令去新建变量来存放需要的源文件，如下

```
cmake_minimum_required (VERSION 2.8)
project (demo)
set( SRC_LIST
     ./main.cpp
     ./test.cpp)
add_executable(main ${SRC_LIST})
```

示例三 - 一个正式的工程构建

一个正式的源码工程应该有这几个目录：

-bin 存放最终的可执行文件
-build 存放编译中间文件
-include 头文件
--sum.h
--minor.h

-src 源代码文件
--sum.cpp
--minor.cpp

main.cpp

-CMakeLists.txt

CMakeLists.txt如下：

```
cmake_minimum_required (VERSION 2.8)
project (math)

# 设置cmake的全局变量
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

#添加头文件路径，相当于makefile里面的-I
include_directories(${PROJECT_SOURCE_DIR}/include)

aux_source_directory (src SRC_LIST)

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

然后在build目录里面执行cmake .. 命令，这样所有的编译中间文件都会在build目录下，最终的可执行文件会在bin目录里面

```
tony@tony-virtual-machine:~/code/cmake/rpc/build$ cmake ..
-- The C compiler identification is GNU 7.4.0
-- The CXX compiler identification is GNU 7.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/tony/code/cmake/rpc/build
tony@tony-virtual-machine:~/code/cmake/rpc/build$ make
Scanning dependencies of target main
[ 25%] Building CXX object CMakeFiles/main.dir/main.cpp.o
[ 50%] Building CXX object CMakeFiles/main.dir/src/minor.cpp.o
[ 75%] Building CXX object CMakeFiles/main.dir/src/sum.cpp.o
[100%] Linking CXX executable ../bin/main
[100%] Built target main
```

```
tony@tony-virtual-machine:~/code/cmake/rpc$ cd bin/
tony@tony-virtual-machine:~/code/cmake/rpc/bin$ ls
main
```

静态库和动态库的编译控制

把上面的sum和minor源文件直接生成静态库或者动态库，让外部程序进行链接使用，代码结构如下：

-bin 存放最终的可执行文件
-build 存放编译中间文件
-lib 存放编译生成的库文件
-include 头文件
--sum.h
--minor.h

-src 源代码文件
--sum.cpp
--minor.cpp
--CMakeLists.txt

-test 测试代码
--main.cpp
--CMakeLists.txt

-CMakeLists.txt

最外层的CMakeLists.txt是总控制编译，内容如下：

```
cmake_minimum_required (VERSION 2.8)
project (math)

add_subdirectory (test)
add_subdirectory (src)
```

src里面的源代码要生成静态库或动态库，CMakeLists.txt内容如下：

```
set (LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
# 生成库，动态库是SHARED，静态库是STATIC
add_library (sum SHARED sum.cpp)
add_library (minor SHARED minor.cpp)
# 修改库的名字
#set_target_properties (sum PROPERTIES OUTPUT_NAME "libsum")
#set_target_properties (minor PROPERTIES OUTPUT_NAME "libminor")
```

test里面的CMakeLists.txt内容如下：

```
set (EXECUTABLE_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/bin)

include_directories (../include) # 头文件搜索路径
link_directories (${PROJECT_SOURCE_DIR}/lib) # 库文件搜索路径

add_executable (main main.cpp) # 指定生成的可执行文件
target_link_libraries (main sum minor) # 执行可执行文件需要依赖的库
```

在build目录下执行cmake ..命令，然后执行make，如下

```
tony@tony-virtual-machine:~/code/cmake/rpc02/build$ make
[ 16%] Building CXX object src/CMakeFiles/minor.dir/minor.cpp.o
[ 33%] Linking CXX shared library ../../lib/libminor.so
[ 33%] Built target minor
[ 50%] Building CXX object src/CMakeFiles/sum.dir/sum.cpp.o
[ 66%] Linking CXX shared library ../../lib/libsum.so
[ 66%] Built target sum
Scanning dependencies of target main
[ 83%] Building CXX object test/CMakeFiles/main.dir/main.cpp.o
[100%] Linking CXX executable ../../bin/main
[100%] Built target main
```

查看生成的可执行文件，检验其链接的库有哪些

```
tony@tony-virtual-machine:~/code/cmake/rpc02/bin$ ls
main
tony@tony-virtual-machine:~/code/cmake/rpc02/bin$ ./main
20 + 10 = 30
20 - 10 = 10
tony@tony-virtual-machine:~/code/cmake/rpc02/bin$ readelf -d ./main

Dynamic section at offset 0x1d48 contains 31 entries:
  标记        类型                         名称/值
 0x0000000000000001 (NEEDED)             共享库：[libsum.so]
 0x0000000000000001 (NEEDED)             共享库：[libminor.so]
 0x0000000000000001 (NEEDED)             共享库：[libstdc++.so.6]
 0x0000000000000001 (NEEDED)             共享库：[libc.so.6]
```

CMake常用的预定义变量

PROJECT_NAME：通过 project() 指定项目名称

PROJECT_SOURCE_DIR：工程的根目录

PROJECT_BINARY_DIR：执行 cmake 命令的目录

CMAKE_CURRENT_SOURCE_DIR：当前 CMakeList.txt 文件所在的目录

CMAKE_CURRENT_BINARY_DIR：编译目录，可使用 add subdirectory 来修改

EXECUTABLE_OUTPUT_PATH：二进制可执行文件输出位置

LIBRARY_OUTPUT_PATH：库文件输出位置

BUILD_SHARED_LIBS：默认的库编译方式（shared 或 static），默认为 static

CMAKE_C_FLAGS：设置 C 编译选项

CMAKE_CXX_FLAGS：设置 C++ 编译选项

CMAKE_CXX_FLAGS_DEBUG：设置编译类型 Debug 时的编译选项

CMAKE_CXX_FLAGS_RELEASE：设置编译类型 Release 时的编译选项

CMAKE_GENERATOR：编译器名称

CMAKE_COMMAND：CMake 可执行文件本身的全路径

CMAKE_BUILD_TYPE：工程编译生成的版本， Debug / Release