

CMake

说明

CMake能够输出各种各样的makefile或者project文件，能测试编译器所支持的C++特性。

类似与makefile的功能，告诉操作系统我的库文件如何进行编译和链接，但CMake实际上是先生成了一个makefile文件

流程如下图：



关于`cmake`的实际使用还需要再去查阅一下资料，以下内容仅仅用于学习`cmake`

语法

注释

□ 2.1.1 注释行

CMake 使用 `#` 进行 行注释，可以放在任何位置。

CMAKE

```
1 # 这是一个 CMakeLists.txt 文件  
2 cmake_minimum_required(VERSION 3.0.0)
```

□ 2.1.2 注释块

CMake 使用 `#[]` 形式进行 块注释。

CMAKE

```
1 #[[ 这是一个 CMakeLists.txt 文件。  
2 这是一个 CMakeLists.txt 文件  
3 这是一个 CMakeLists.txt 文件]]  
4 cmake_minimum_required(VERSION 3.0.0)
```

CMakeLists.txt

3. 添加 CMakeLists.txt 文件

在上述源文件所在目录下添加一个新文件 CMakeLists.txt，文件内容如下：

CMAKE

```
1 cmake_minimum_required(VERSION 3.0)
2 project(CALC)
3 add_executable(app add.c div.c main.c mult.c sub.c)
```

接下来依次介绍一下在 CMakeLists.txt 文件中添加的三个命令：

接下来依次介绍一下在 CMakeLists.txt 文件中添加的三个命令：

- `cmake_minimum_required`：指定使用的 cmake 的最低版本
 - 可选，非必须，如果不加可能不会有警告
- `project`：定义工程名称，并可指定工程的版本、工程描述、web 主页地址、支持的语言（默认情况支持所有语言），如果不需要这些都是可以忽略的，只需要指定出工程名字即可。

CMAKE

```
1 # PROJECT 指令的语法是:
2 project(<PROJECT-NAME> [<language-name>...])
3 project(<PROJECT-NAME>
4   [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]
5   [DESCRIPTION <project-description-string>]
6   [HOMEPAGE_URL <url-string>]
7   [LANGUAGES <language-name>...])
```

- `add_executable`：定义工程会生成一个可执行程序

CMAKE

```
1 add_executable(可执行程序名 源文件名称)
```

- 这里的可执行程序名和 `project` 中的项目名没有任何关系
- 源文件名可以是一个也可以是多个，如有多个可用空格或 ; 间隔

CMAKE

```
1 # 样式1
2 add_executable(app add.c div.c main.c mult.c sub.c)
3 # 样式2
4 add_executable(app add.c;div.c;main.c;mult.c;sub.c)
```

cmake 命令的参数必须是 CMakeLists.txt 文件所在的路径，可以创建一个新文件夹用来放置编译出来的文件。

set**□ 2.2.1 定义变量**

在上面的例子中一共提供了5个源文件，假设这五个源文件需要反复被使用，每次都直接将它们的名字写出来确实是很麻烦，此时我们就需要定义一个变量，将文件名对应的字符串存储起来，在cmake里定义变量需要使用 `set`。

CMAKE

```
1 # SET 指令的语法是:
2 # [] 中的参数为可选项, 如不需要可以不写
3 SET(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]])
```

- `VAR` : 变量名
- `VALUE` : 变量值

CMAKE

```
1 # 方式1: 各个源文件之间使用空格间隔
2 # set(SRC_LIST add.c div.c main.c mult.c sub.c)
3
4 # 方式2: 各个源文件之间使用分号 ; 间隔
5 set(SRC_LIST add.c;div.c;main.c;mult.c;sub.c)
6 add_executable(app ${SRC_LIST})
```

使用`set`的取值，记得上面标红的格式，`{}{}`

指定C++标准**1. 在 CMakeLists.txt 中通过 `set` 命令指定**

CMAKE

```
1 #增加-std=c++11
2 set(CMAKE_CXX_STANDARD 11)
3 #增加-std=c++14
4 set(CMAKE_CXX_STANDARD 14)
5 #增加-std=c++17
6 set(CMAKE_CXX_STANDARD 17)
```

2. 在执行 `cmake` 命令的时候指定出这个宏的值

SHELL

```
1 #增加-std=c++11
2 cmake CMakeLists.txt文件路径 -DCMAKE_CXX_STANDARD=11
3 #增加-std=c++14
4 cmake CMakeLists.txt文件路径 -DCMAKE_CXX_STANDARD=14
5 #增加-std=c++17
6 cmake CMakeLists.txt文件路径 -DCMAKE_CXX_STANDARD=17
```

指定输出位置

最好使用绝对路径

□ 2.2.3 指定输出的路径

在CMake中指定可执行程序输出的路径，也对应一个宏，叫做 `EXECUTABLE_OUTPUT_PATH`，它的值还是通过 `set` 命令进行设置：

```
CMAKE  
1 set(HOME /home/robin/Linux/Sort)  
2 set(EXECUTABLE_OUTPUT_PATH ${HOME}/bin)
```

- 第一行：定义一个变量用于存储一个绝对路径
- 第二行：将拼接好的路径值设置给 `EXECUTABLE_OUTPUT_PATH` 宏
 - 如果这个路径中的子目录不存在，会自动生成，无需自己手动创建

由于可执行程序是基于 `cmake` 命令生成的 `makefile` 文件然后再执行 `make` 命令得到的，所以如果此处指定可执行程序生成路径的时候使用的是相对路径 `./xxx/xxx`，那么这个路径中的 `./` 对应的就是 `makefile` 文件所在的那个目录。

搜索文件

如果一个项目里边的源文件很多，在编写 `CMakeLists.txt` 文件的时候不可能将项目目录的各个文件一一罗列出来，这样太麻烦也不现实。所以，在CMake中为我们提供了搜索文件的命令，可以使用 `aux_source_directory` 命令或者 `file` 命令。

② 2.3.1 方式1

在 CMake 中使用 `aux_source_directory` 命令可以查找某个路径下的 `所有源文件`，命令格式为：

```
1 aux_source_directory(< dir > < variable >)
```

- `dir`：要搜索的目录
- `variable`：将从 `dir` 目录下搜索到的源文件列表存储到该变量中

```
1 cmake_minimum_required(VERSION 3.0)
2 project(CALC)
3 include_directories(${PROJECT_SOURCE_DIR}/include)
4 # 搜索 src 目录下的源文件
5 aux_source_directory(${CMAKE_CURRENT_SOURCE_DIR}/src SRC_LIST)
6 add_executable(app ${SRC_LIST})
```

② 2.3.2 方式2

如果一个项目里边的源文件很多，在编写 `CMakeLists.txt` 文件的时候不可能将项目目录的各个文件一一罗列出来，这样太麻烦了。所以，在CMake中为我们提供了搜索文件的命令，他就是 `file` (当然，除了搜索以外通过 `file` 还可以做其他事情)。

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

- `GLOB`：将指定目录下搜索到的满足条件的所有文件名生成一个列表，并将其存储到变量中。
- `GLOB_RECURSE`：递归搜索指定目录，将搜索到的满足条件的文件名生成一个列表，并将其存储到变量中。

搜索当前目录的src目录下所有的源文件，并存储到变量中

```
1 file(GLOB MAIN_SRC ${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp)
2 file(GLOB MAIN_HEAD ${CMAKE_CURRENT_SOURCE_DIR}/include/*.h)
```

- `CMAKE_CURRENT_SOURCE_DIR` 宏表示当前访问的 `CMakeLists.txt` 文件所在的路径。
- 关于要搜索的文件路径和类型可加双引号，也可不加：

```
1 file(GLOB MAIN_HEAD "${CMAKE_CURRENT_SOURCE_DIR}/src/*.h")
```

实例

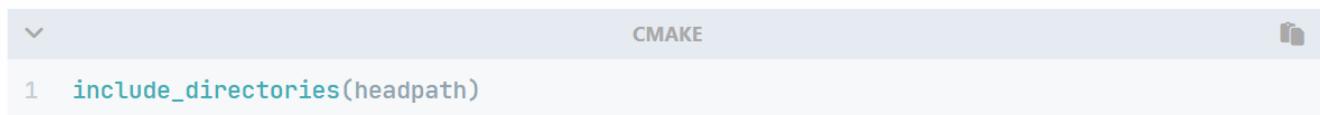
```
1 cmake_minimum_required(VERSION 3.15)
2 project(test)
3 # set(SRC add.cpp div.cpp mult.cpp main.cpp sub.cpp)
4 # aux_source_directory(${PROJECT_SOURCE_DIR} SRC)
5 file(GLOB SRC ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)
6 set(EXECUTABLE_OUTPUT_PATH /home/dabing/aa/bb/cc)
7 # set(CMAKE_CXX_STANDARD 11)
8 add_executable(app ${SRC})
```

其中的 `PROJECT_SOURCE_DIR` 和 `CMAKE_CURRENT_SOURCE_DIR` 都是用来指代 `CMakeLists` 文件所在位置的路径
指定头文件的路径

注意这里是指定头文件的目录的路径，而不是具体的某个头文件

② 2.4 包含头文件

在编译项目源文件的时候，很多时候都需要将源文件对应的头文件路径指定出来，这样才能保证在编译过程中编译器能够找到这些头文件，并顺利通过编译。在 CMake 中设置要包含的目录也很简单，通过一个命令就可以搞定了，他就是 `include_directories`：



```
1 include_directories(headpath)
```

举例说明，有源文件若干，其目录结构如下：

举例说明，有源文件若干，其目录结构如下：

```
1 $ tree
2 .
3 └── build
4     └── CMakeLists.txt
5     └── include
6         └── head.h
7     └── src
8         ├── add.cpp
9         ├── div.cpp
10        ├── main.cpp
11        ├── mult.cpp
12        └── sub.cpp
13
14 3 directories, 7 files
```

CMakeLists.txt 文件内容如下：

```
1 cmake_minimum_required(VERSION 3.0)
2 project(CALC)
3 set(CMAKE_CXX_STANDARD 11)
4 set(HOME /home/robin/Linux/calc)
5 set(EXECUTABLE_OUTPUT_PATH ${HOME}/bin/)
6 include_directories(${PROJECT_SOURCE_DIR}/include)
7 file(GLOB SRC_LIST ${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp)
8 add_executable(app ${SRC_LIST})
```

其中，第六行指定就是头文件的路径，`PROJECT_SOURCE_DIR` 宏对应的值就是我们在使用cmake命令时，后面紧跟的目录，一般是工程的根目录。

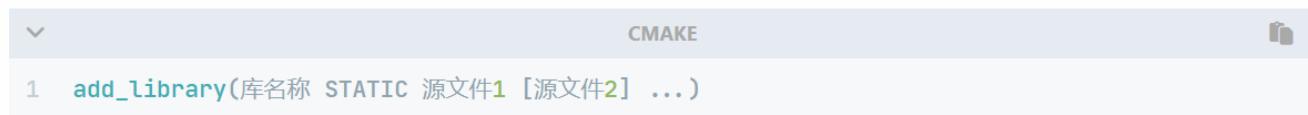
制作动态库和静态库

这个东西的作用在于你生成的库都是二进制文件，别人是没有办法知道你原代码是什么样子的，这样就相当于一次加密的行为

库文件就是包含了你所写的函数还有头文件之类的东西

静态库

在cmake中，如果要制作静态库，需要使用的命令如下：



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

在Linux中，静态库名字分为三部分：`lib + 库名字 + .a`，此处只需要指定出库的名字就可以了，另外两部分在生成该文件的时候会自动填充。

在Windows中虽然库名和Linux格式不同，但也只需指定出名字即可。

下面有一个目录，需要将 `src` 目录中的源文件编译成静态库，然后再使用：



```
1 .
2   └── build
3   └── CMakeLists.txt
4   └── include          # 头文件目录
5     └── head.h
6   └── main.cpp        # 用于测试的源文件
7   └── src              # 源文件目录
8     ├── add.cpp
9     ├── div.cpp
10    ├── mult.cpp
11    └── sub.cpp
```

根据上面的目录结构，可以这样编写 `CMakeLists.txt` 文件：



```
1 cmake_minimum_required(VERSION 3.0)
2 project(CALC)
3 include_directories(${PROJECT_SOURCE_DIR}/include)
4 file(GLOB SRC_LIST "${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp")
5 add_library(calc STATIC ${SRC_LIST})
```

这样最终就会生成对应的静态库文件 `libcalc.a`。

动态库

在cmake中，如果要制作动态库，需要使用的命令如下：

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

在Linux中，动态库名字分为三部分：`lib + 库名字 + .so`，此处只需要指定出库的名字就可以了，另外两部分在生成该文件的时候会自动填充。

在Windows中虽然库名和Linux格式不同，但也只需指定出名字即可。

根据上面的目录结构，可以这样编写 `CMakeLists.txt` 文件：

```
CMAKE
1 cmake_minimum_required(VERSION 3.0)
2 project(CALC)
3 include_directories(${PROJECT_SOURCE_DIR}/include)
4 file(GLOB SRC_LIST "${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp")
5 add_library(calc SHARED ${SRC_LIST})
```

这样最终就会生成对应的动态库文件 `libcalc.so`。

指定输出库文件的路径

② 方式2 - 都适用

由于在Linux下生成的静态库默认不具有可执行权限，所以在指定静态库生成的路径的时候就不能使用

`EXECUTABLE_OUTPUT_PATH` 宏了，而应该使用 `LIBRARY_OUTPUT_PATH`，这个宏对应静态库文件和动态库文件都适用。

```
CMAKE
1 cmake_minimum_required(VERSION 3.0)
2 project(CALC)
3 include_directories(${PROJECT_SOURCE_DIR}/include)
4 file(GLOB SRC_LIST "${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp")
5 # 设置动态库/静态库生成路径
6 set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
7 # 生成动态库
8 #add_library(calc SHARED ${SRC_LIST})
9 # 生成静态库
10 add_library(calc STATIC ${SRC_LIST})
```

和输出路径有关的这种东西，如果本身没有这个路径，`cmake`会自动创建这个路径

链接静态库

在cmake中，链接静态库的命令如下：

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

- **参数1**: 指定出要链接的静态库的名字
 - 可以是全名 `libxxx.a`
 - 也可以是掐头（`lib`）去尾（`.a`）之后的名字 `xxx`
- **参数2-N**: 要链接的其它静态库的名字

如果该静态库不是系统提供的（自己制作或者使用第三方提供的静态库）可能出现静态库找不到的情况，此时可以将静态库的路径也指定出来：

```
CMAKE  
1 link_directories(<lib path>)
```

这样，修改之后的 `CMakeLists.txt` 文件内容如下：

```
CMAKE  
1 cmake_minimum_required(VERSION 3.0)  
2 project(CALC)  
3 # 搜索指定目录下源文件  
4 file(GLOB SRC_LIST ${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp)  
5 # 包含头文件路径  
6 include_directories(${PROJECT_SOURCE_DIR}/include)  
7 # 包含静态库路径  
8 link_directories(${PROJECT_SOURCE_DIR}/Lib)  
9 # 链接静态库  
10 link_libraries(calc)  
11 add_executable(app ${SRC_LIST})
```

添加了第8行的代码，就可以根据参数指定的路径找到这个静态库了。

静态库和源文件都会被打包到应用程序中，动态库是在**应用程序调用对应的函数**了才开始加载到内存

链接动态库

在 `cmake` 中链接动态库的命令如下：

```
CMAKE
1 target_link_libraries(
2     <target>
3     <PRIVATE|PUBLIC|INTERFACE> <item>...
4     [<PRIVATE|PUBLIC|INTERFACE> <item>...]...)
```

- **target**: 指定要加载动态库的文件的名字
 - 该文件可能是一个源文件
 - 该文件可能是一个动态库文件
 - 该文件可能是一个可执行文件
- **PRIVATE|PUBLIC|INTERFACE**: 动态库的访问权限, 默认为 `PUBLIC`
 - 如果各个动态库之间没有依赖关系, 无需做任何设置, 三者没有区别, **一般无需指定, 使用默认的 PUBLIC 即可。**
 - **动态库的链接具有传递性**, 如果动态库 A 链接了动态库B、C, 动态库D链接了动态库A, 此时动态库D相当于也链接了动态库B、C, 并可以使用动态库B、C中定义的方法。

```
CMAKE
1 target_link_libraries(A B C)
2 target_link_libraries(D A)
```

- `PUBLIC` : 在public后面的库会被Link到前面的target中, 并且里面的符号也会被导出, 提供给第三方使用。
- `PRIVATE` : 在private后面的库仅被link到前面的target中, 并且终结掉, 第三方不能感知你调了啥库
- `INTERFACE` : 在interface后面引入的库不会被链接到前面的target中, 只会导出符号。

`public`权限类似`c++`, `private`权限不同与`c++`, 最后一个权限传递的信息最少

动态库在进程需要调用其中的函数的时候会加载到进程的虚拟内存空间中, 所以动态库可以同时被多个进程调用

动态库和静态库的区别

② 链接系统动态库

动态库的链接和静态库是完全不同的:

- 静态库会在生成可执行程序的链接阶段被打包到可执行程序中, 所以可执行程序启动, 静态库就被加载到内存中了。
- 动态库在生成可执行程序的链接阶段**不会**被打包到可执行程序中, 当可执行程序被启动并且调用了动态库中的函数的时候, 动态库才会被加载到内存

动态库的链接通常写在生成执行程序的后面

因此，在 `cmake` 中指定要链接的动态库的时候，**应该将命令写到生成了可执行文件之后：**



```
1 cmake_minimum_required(VERSION 3.0)
2 project(TEST)
3 file(GLOB SRC_LIST ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)
4 # 添加并指定最终生成的可执行程序名
5 add_executable(app ${SRC_LIST})
6 # 指定可执行程序要链接的动态库名字
7 target_link_libraries(app pthread)
```

在 `target_link_libraries(app pthread)` 中：

- `app`：对应的是最终生成的可执行程序的名字
- `pthread`：这是可执行程序要加载的动态库，这个库是系统提供的线程库，全名为 `libpthread.so`，在指定的时候一般会掐头（lib）去尾（.so）。

如果需要链接多个动态库，在相应的语法后面用空格隔开就可以

链接自己编写的动态库需要表明路径所在

这是因为可执行程序启动之后，去加载 `calc` 这个动态库，但是不知道这个动态库被放到了什么位置

解决动态库无法加载的问题，所以就加载失败了，在 CMake 中可以在生成可执行程序之前，通过命令指定出要链接的动态库的位置，指定静态库位置使用的也是这个命令：

```
link_directories(path)
```

所以修改之后的 `CMakeLists.txt` 文件应该是这样的：

```
cmake_minimum_required(VERSION 3.0)
project(TEST)
file(GLOB SRC_LIST ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)
# 指定源文件或者动态库对应的头文件路径
include_directories(${PROJECT_SOURCE_DIR}/include)
# 指定要链接的动态库的路径
link_directories(${PROJECT_SOURCE_DIR}/lib)
# 添加并生成一个可执行程序
add_executable(app ${SRC_LIST})
# 指定要链接的动态库
target_link_libraries(app pthread calc)
```

通过 `link_directories` 指定了动态库的路径之后，在执行生成的可执行程序的时候，就不会出现找不到动态库的问题了。

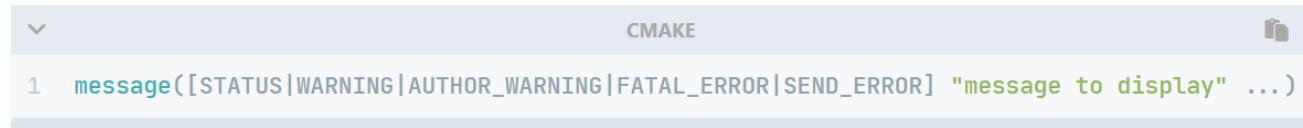


温馨提示：使用 `target_link_libraries` 命令就可以链接动态库，也可以链接静态库文件。

日志

也就是一种调试的办法，类似于 C 中的 `printf` 和 Rust 中的 `println!`，可以利用 `{}$` 来打印变量的信息，但是这个不同的参数可以决定是否中断程序

在CMake中可以用用户显示一条消息，该命令的名字为 `message`：



```
1 message([STATUS|WARNING|AUTHOR_WARNING|FATAL_ERROR|SEND_ERROR] "message to display" ...)
```

- `(无)`：重要消息
- `STATUS`：非重要消息
- `WARNING`：CMake 警告, 会继续执行
- `AUTHOR_WARNING`：CMake 警告 (dev), 会继续执行
- `SEND_ERROR`：CMake 错误, 继续执行, 但是会跳过生成的步骤
- `FATAL_ERROR`：CMake 错误, 终止所有处理过程

CMake的命令行工具会在`stdout`上显示 `STATUS` 消息，在`stderr`上显示其他所有消息。CMake的GUI会在它的log区域显示所有消息。

CMake警告和错误消息的文本显示使用的是一种简单的标记语言。文本没有缩进，超过长度的行会回卷，段落之间以新行做为分隔符。



```
1 # 输出一般日志信息
2 message(STATUS "source path: ${PROJECT_SOURCE_DIR}")
3 # 输出警告信息
4 message(WARNING "source path: ${PROJECT_SOURCE_DIR}")
5 # 输出错误信息
6 message(FATAL_ERROR "source path: ${PROJECT_SOURCE_DIR}")
```

变量操作

- set追加

② 2.8.1 追加

有时候项目中的源文件并不一定都在同一个目录中，但是这些源文件最终却需要一起进行编译来生成最终的可执行文件或者库文件。如果我们通过 `file` 命令对各个目录下的源文件进行搜索，最后还需要做一个字符串拼接的操作，关于字符串拼接可以使用 `set` 命令也可以使用 `list` 命令。

② 使用set拼接

如果使用`set`进行字符串拼接，对应的命令格式如下：

```
1 set(变量名1 ${变量名1} ${变量名2} ...)
```

关于上面的命令其实就是将从第二个参数开始往后所有的字符串进行拼接，最后将结果存储到第一个参数中，如果第一个参数中原来有数据会对原数据就行覆盖。

```
1 cmake_minimum_required(VERSION 3.0)
2 project(TEST)
3 set(TEMP "hello,world")
4 file(GLOB SRC_1 ${PROJECT_SOURCE_DIR}/src1/*.cpp)
5 file(GLOB SRC_2 ${PROJECT_SOURCE_DIR}/src2/*.cpp)
6 # 追加(拼接)
7 set(SRC_1 ${SRC_1} ${SRC_2} ${TEMP})
8 message(STATUS "message: ${SRC_1}")
```

- list追加

如果使用list进行字符串拼接，对应的命令格式如下：

```
1 list(APPEND <list> [<element> ...])
```

`list` 命令的功能比 `set` 要强大，字符串拼接只是它的其中一个功能，所以需要在它第一个参数的位置指定出我们要做的操作，`APPEND` 表示进行数据追加，后边的参数和 `set` 就一样了。

```
1 cmake_minimum_required(VERSION 3.0)
2 project(TEST)
3 set(TEMP "hello,world")
4 file(GLOB SRC_1 ${PROJECT_SOURCE_DIR}/src1/*.cpp)
5 file(GLOB SRC_2 ${PROJECT_SOURCE_DIR}/src2/*.cpp)
6 # 追加(拼接)
7 list(APPEND SRC_1 ${SRC_1} ${SRC_2} ${TEMP})
8 message(STATUS "message: ${SRC_1}")
```

在CMake中，使用 `set` 命令可以创建一个 `list`。一个在 `list` 内部是一个由 分号; 分割的一组字符串。例如，`set(var a b c d e)` 命令将会创建一个 `list:a;b;c;d;e`，但是最终打印变量值的时候得到的是 `abcde`。

```
1 set(tmp1 a;b;c;d;e)
2 set(tmp2 a b c d e)
3 message(${tmp1})
4 message(${tmp2})
```

输出的结果：

```
1 abcde
2 abcde
```

- list移除

我们在通过 `file` 搜索某个目录就得到了该目录下所有的源文件，但是其中有些源文件并不是我们所需要的，比如：

```
SHELL
1 $ tree
2 .
3   ├── add.cpp
4   ├── div.cpp
5   ├── main.cpp
6   ├── mult.cpp
7   └── sub.cpp
8
9 0 directories, 5 files
```

在当前这么目录有五个源文件，其中 `main.cpp` 是一个测试文件。如果我们想要把计算器相关的源文件生成一个动态库给别人使用，那么只需要 `add.cpp`、`div.cpp`、`mult.cpp`、`sub.cpp` 这四个源文件就可以了。此时，就需要将 `main.cpp` 从搜索到的数据中剔除出去，想要实现这个功能，也可以使用 `list`

```
CMAKE
1 list(REMOVE_ITEM <list> <value> [<value> ...])
```

通过上面的命令原型可以看到删除和追加数据类似，只不过是第一个参数变成了 `REMOVE_ITEM`。

```
CMAKE
1 cmake_minimum_required(VERSION 3.0)
2 project(TEST)
3 set(TEMP "hello,world")
4 file(GLOB SRC_1 ${PROJECT_SOURCE_DIR}/*.cpp)
5 # 移除前日志
6 message(STATUS "message: ${SRC_1}")
7 # 移除 main.cpp
8 list(REMOVE_ITEM SRC_1 ${PROJECT_SOURCE_DIR}/main.cpp)
9 # 移除后日志
10 message(STATUS "message: ${SRC_1}")
```

可以看到，在 第8行 把将要移除的文件的名字指定给 `list` 就可以了。但是一定要注意通过 `file` 命令搜索源文件的时候得到的是文件的绝对路径（在 `list` 中每个文件对应的路径都是一个item，并且都是绝对路径），那么在移除的时候也要将该文件的绝对路径指定出来才可以，否是移除操作不会成功。

- `list` 的其他用法

关于 `list` 命令还有其它功能，但是并不常用，在此就不一一进行举例介绍了。

1. 获取 list 的长度。

```
CMAKE
1 list(LENGTH <list> <output variable>)
```

- `LENGTH`：子命令 LENGTH 用于读取列表长度
- `<list>`：当前操作的列表
- `<output variable>`：新创建的变量，用于存储列表的长度。

2. 读取列表中指定索引的元素，可以指定多个索引

```
CMAKE
1 list(GET <list> <element index> [<element index> ...] <output variable>)
```

- `<list>`：当前操作的列表
- `<element index>`：列表元素的索引
 - 从0开始编号，索引0的元素为列表中的第一个元素；
 - 索引也可以是负数，`-1` 表示列表的最后一个元素，`-2` 表示列表倒数第二个元素，以此类推
 - 当索引（不管是正还是负）超过列表的长度，运行会报错
- `<output variable>`：新创建的变量，存储指定索引元素的返回结果，也是一个列表。

3. 将列表中的元素用连接符（字符串）连接起来组成一个字符串

```
CMAKE
1 list (JOIN <list> <glue> <output variable>)
```

- `<list>`：当前操作的列表
- `<glue>`：指定的连接符（字符串）
- `<output variable>`：新创建的变量，存储返回的字符串

4. 查找列表是否存在指定的元素，若果未找到，返回-1

```
CMAKE
1 list(FIND <list> <value> <output variable>)
```

- `<list>`：当前操作的列表
- `<value>`：需要在列表中搜索的元素
- `<output variable>`：新创建的变量
 - 如果列表 `<list>` 中存在 `<value>`，那么返回 `<value>` 在列表中的索引
 - 如果未找到则返回-1。

5. 将元素追加到列表中

```
CMAKE
20 / 34
```

```
1 list (APPEND <list> [<element> ...])
```

6. 在list中指定的位置插入若干元素

```
1 list(INSERT <list> <element_index> <element> [<element> ...])
```

7. 将元素插入到列表的0索引位置

```
1 list (PREPEND <list> [<element> ...])
```

8. 将列表中最后元素移除

```
1 list (POP_BACK <list> [<out-var>...])
```

9. 将列表中第一个元素移除

```
1 list (POP_FRONT <list> [<out-var>...])
```

10. 将指定的元素从列表中移除

```
1 list (REMOVE_ITEM <list> <value> [<value> ...])
```

11. 将指定索引的元素从列表中移除

```
1 list (REMOVE_AT <list> <index> [<index> ...])
```

12. 移除列表中的重复元素

```
1 list (REMOVE_DUPLICATES <list>)
```

13. 列表翻转

```
1 list(REVERSE <list>)
```

14. 列表排序

```
1 list (SORT <list> [COMPARE <compare>] [CASE <case>] [ORDER <order>])
```

- **COMPARE** : 指定排序方法。有如下几种值可选：
 - **STRING** :按照字母顺序进行排序，为默认的排序方法
 - **FILE_BASENAME** : 如果是一系列路径名，会使用basename进行排序
 - **NATURAL** : 使用自然数顺序排序
- **CASE** : 指明是否大小写敏感。有如下几种值可选：
 - **SENSITIVE** :按照大小写敏感的方式进行排序，为默认值
 - **INSENSITIVE** : 按照大小写不敏感方式进行排序
- **ORDER** : 指明排序的顺序。有如下几种值可选：
 - **ASCENDING** :按照升序排列，为默认值
 - **DESCENDING** : 按照降序排列

宏

利用宏的原因在于，在程序编写过程中会用到大量printf来进行打印测试，如果程序编写好了，再一条条删除非常麻烦，这时候宏的作用就体现出来了

在进行程序测试的时候，我们可以在代码中添加一些宏定义，通过这些宏来控制这些代码是否生效，如下所示：

```
✓ C++ 

1 #include <stdio.h>
2 #define NUMBER 3
3
4 int main()
5 {
6     int a = 10;
7     #ifdef DEBUG
8         printf("我是一个程序猿， 我不会爬树 ... \n");
9     #endif
10    for(int i=0; i<NUMBER; ++i)
11    {
12        printf("hello, GCC!!!\n");
13    }
14    return 0;
15 }
```

在程序的第七行对 `DEBUG` 宏进行了判断，如果该宏被定义了，那么第八行就会进行日志输出，如果没有定义这个宏，第八行就相当于被注释掉了，因此最终无法看到日志输入出（**上述代码中并没有定义这个宏**）。

为了让测试更灵活，我们可以不在代码中定义这个宏，而是在测试的时候去把它定义出来，其中一种方式就是在 `gcc/g++` 命令中去指定，如下：

```
✓ SHELL 

1 $ gcc test.c -DDEBUG -o app
```

在 `gcc/g++` 命令中通过参数 `-D` 指定出要定义的宏的名字，这样就相当于在代码中定义了一个宏，其名字为 `DEBUG`。

在 `CMake` 中我们也可以做类似的事情，对应的命令叫做 `add_definitions`：

```
✓ CMAKE 

1 add_definitions(-D宏名称)
```

针对于上面的源文件编写一个 `CMakeLists.txt`，内容如下：

```
✓ CMAKE 

1 cmake_minimum_required(VERSION 3.0)
2 project(TEST)
3 # 自定义 DEBUG 宏
4 add_definitions(-DDEBUG)
5 add_executable(app ./test.c)
```

通过这种方式，上述代码中的第八行日志就能够被输出出来了。

系统自定义好的宏

②3. 预定义宏

下面的列表中为大家整理了一些 CMake 中常用的宏：

宏	功能
PROJECT_SOURCE_DIR	使用cmake命令后紧跟的目录，一般是工程的根目录
PROJECT_BINARY_DIR	执行cmake命令的目录
CMAKE_CURRENT_SOURCE_DIR	当前处理的CMakeLists.txt所在的路径
CMAKE_CURRENT_BINARY_DIR	target 编译目录
EXECUTABLE_OUTPUT_PATH	重新定义目标二进制可执行文件的存放位置
LIBRARY_OUTPUT_PATH	重新定义目标链接库文件的存放位置
PROJECT_NAME	返回通过PROJECT指令定义的项目名称
CMAKE_BINARY_DIR	项目实际构建路径，假设在 build 目录进行的构建，那么得到的就是这个目录的路径

关于更加贴近现实的例子，也就是嵌套定义cmake，详情见 <https://subingwen.cn/cmake/CMake-advanced/>

也可以参考b站视频 <https://www.bilibili.com/video/BV14s4y1g7Zj?>

p=17&spm_id_from=pageDriver&vd_source=6145f2ccf40a9f978a5b12b81fe169f0

以下是截图

② 1. 嵌套的CMake

如果项目很大，或者项目中有很多的源码目录，在通过CMake管理项目的时候如果只使用一个 `CMakeLists.txt`，那么这个文件相对会比较复杂，有一种化繁为简的方式就是给每个源码目录都添加一个 `CMakeLists.txt` 文件（头文件目录不需要），这样每个文件都不会太复杂，而且更灵活，更容易维护。

先来看一下下面的这个的目录结构：

```
1 $ tree
2 .
3 └── build
4     ├── calc
5     │   ├── add.cpp
6     │   ├── CMakeLists.txt
7     │   ├── div.cpp
8     │   ├── mult.cpp
9     └── sub.cpp
10    └── CMakeLists.txt
11    └── include
12        ├── calc.h
13        └── sort.h
14    └── sort
15        ├── CMakeLists.txt
16        ├── insert.cpp
17        └── select.cpp
18    └── test1
19        ├── calc.cpp
20        └── CMakeLists.txt
21    └── test2
22        ├── CMakeLists.txt
23        └── sort.cpp
24
25 6 directories, 15 files
```

- `include` 目录：头文件目录
- `calc` 目录：目录中的四个源文件对应的加、减、乘、除算法
 - 对应的头文件是 `include` 中的 `calc.h`
- `sort` 目录：目录中的两个源文件对应的是插入排序和选择排序算法
 - 对应的头文件是 `include` 中的 `sort.h`
- `test1` 目录：测试目录，对加、减、乘、除算法进行测试
- `test2` 目录：测试目录，对排序算法进行测试

可以看到各个源文件目录所需要的 `CMakeLists.txt` 文件现在已经添加完毕了。接下来庖丁解牛，我们依次分析一下各个文件中需要添加的内容。

② 1.1 准备工作

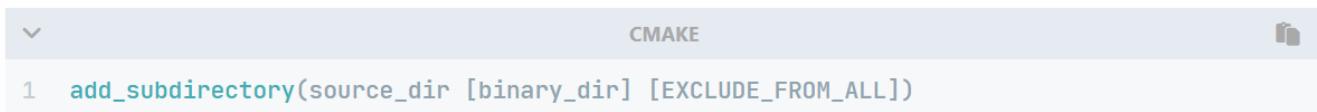
② 1.1.1 节点关系

众所周知，Linux的目录是树状结构，所以 嵌套的 CMake 也是一个树状结构，最顶层的 `CMakeLists.txt` 是根节点，其次都是子节点。因此，我们需要了解一些关于 `CMakeLists.txt` 文件变量作用域的一些信息：

- 根节点 `CMakeLists.txt` 中的变量全局有效
- 父节点 `CMakeLists.txt` 中的变量可以在子节点中使用
- 子节点 `CMakeLists.txt` 中的变量只能在当前节点中使用

② 1.1.2 添加子目录

接下来我们还需要知道在 CMake 中父子节点之间的关系是如何建立的，这里需要用到一个 CMake 命令：



The screenshot shows a code editor window with the title 'CMAKE'. The code in the editor is:

```
1 add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])
```

- `source_dir`：指定了 `CMakeLists.txt` 源文件和代码文件的位置，其实就是指定子目录
- `binary_dir`：指定了输出文件的路径，一般不需要指定，忽略即可。
- `EXCLUDE_FROM_ALL`：在子路径下的目标默认不会被包含到父路径的 `ALL` 目标里，并且也会被排除在IDE工程文件之外。用户必须显式构建在子路径下的目标。

通过这种方式 `CMakeLists.txt` 文件之间的父子关系就被构建出来了。

② 1.2 解决问题

在上面的目录中我们要做如下事情：

1. 通过 `test1` 目录 中的测试文件进行计算器相关的测试
2. 通过 `test2` 目录 中的测试文件进行排序相关的测试

现在相当于是要进行模块化测试，对于 `calc` 和 `sort` 目录中的源文件来说，可以将它们先编译成库文件（可以是静态库也可以是动态库）然后在提供给测试文件使用即可。库文件的本质其实还是代码，只不过是从文本格式变成了二进制格式。

② 1.2.1 根目录

根目录中的 `CMakeLists.txt` 文件内容如下：



```
cmake_minimum_required(VERSION 3.0)
project(test)
# 定义变量
# 静态库生成的路径
set(LIB_PATH ${CMAKE_CURRENT_SOURCE_DIR}/lib)
# 测试程序生成的路径
set(EXEC_PATH ${CMAKE_CURRENT_SOURCE_DIR}/bin)
# 头文件目录
set(HEAD_PATH ${CMAKE_CURRENT_SOURCE_DIR}/include)
```

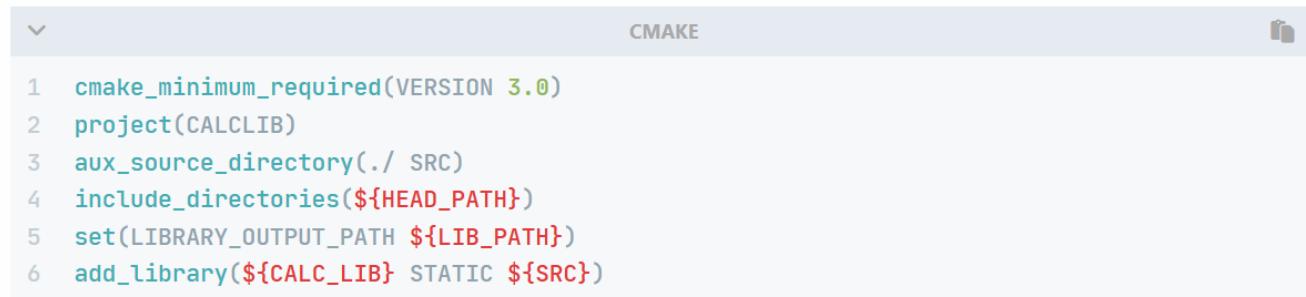
```
10 # 静态库的名字
11 set(CALC_LIB calc)
12 set(SORT_LIB sort)
13 # 可执行程序的名字
14 set(APP_NAME_1 test1)
15 set(APP_NAME_2 test2)
16 # 添加子目录
17 add_subdirectory(calc)
18 add_subdirectory(sort)
19 add_subdirectory(test1)
20 add_subdirectory(test2)
```

在根节点对应的文件中主要做了两件事情： 定义全局变量 和 添加子目录。

- 定义的全局变量主要是给子节点使用，目的是为了提高子节点中的 `CMakeLists.txt` 文件的可读性和可维护性，避免冗余并降低出差的概率。
- 一共添加了四个子目录，每个子目录中都有一个 `CMakeLists.txt` 文件，这样它们的父子关系就被确定下来了。

② 1.2.2 calc 目录

calc 目录中的 `CMakeLists.txt` 文件内容如下：



```
1 cmake_minimum_required(VERSION 3.0)
2 project(CALCLIB)
3 aux_source_directory(. SRC)
4 include_directories(${HEAD_PATH})
5 set(LIBRARY_OUTPUT_PATH ${LIB_PATH})
6 add_library(${CALC_LIB} STATIC ${SRC})
```

- 第3行 `aux_source_directory`：搜索当前目录（calc目录）下的所有源文件
- 第4行 `include_directories`：包含头文件路径，`HEAD_PATH` 是在根节点文件中定义的
- 第5行 `set`：设置库的生成的路径，`LIB_PATH` 是在根节点文件中定义的

- 第6行 `add_library` : 生成静态库, 静态库名字 `CALC_LIB` 是在根节点文件中定义的

② 1.2.3 sort 目录

sort 目录中的 `CMakeLists.txt` 文件内容如下:

```
1 cmake_minimum_required(VERSION 3.0)
2 project(SORTLIB)
3 aux_source_directory(./ SRC)
4 include_directories(${HEAD_PATH})
5 set(LIBRARY_OUTPUT_PATH ${LIB_PATH})
6 add_library(${SORT_LIB} SHARED ${SRC})
```

- 第6行 `add_library` : 生成动态库, 动态库名字 `SORT_LIB` 是在根节点文件中定义的

这个文件中的内容和 `calc` 节点文件中的内容类似, 只不过这次生成的是动态库。

 在生成库文件的时候, 这个库可以是静态库也可以是动态库, 一般需要根据实际情况来确定。如果生成的库比较大, 建议将其制作成动态库。

② 1.2.4 test1 目录

test1 目录中的 `CMakeLists.txt` 文件内容如下:

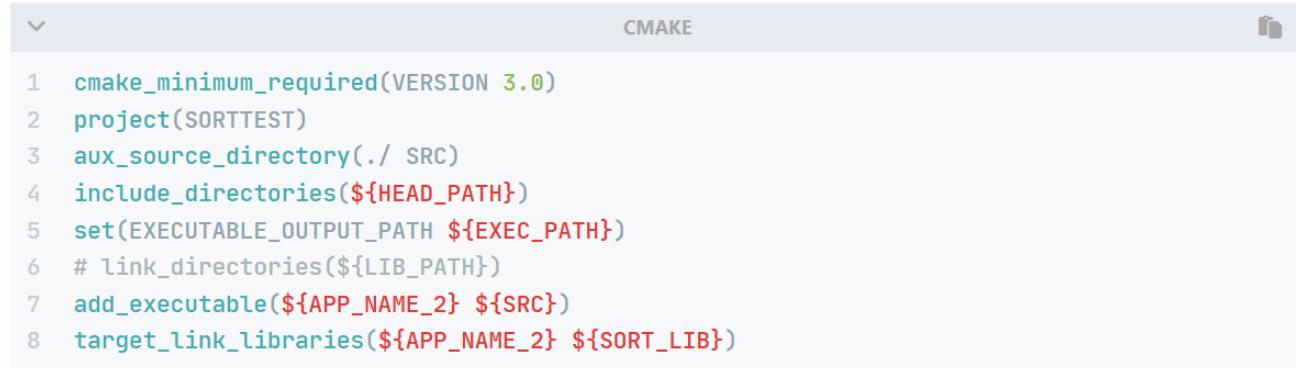
```
1 cmake_minimum_required(VERSION 3.0)
2 project(CALCTEST)
3 aux_source_directory(./ SRC)
4 include_directories(${HEAD_PATH})
5 # include_directories(${HEAD_PATH})
6 link_libraries(${CALC_LIB})
7 set(EXECUTABLE_OUTPUT_PATH ${EXEC_PATH})
8 add_executable(${APP_NAME_1} ${SRC})
```

- 第4行 `include_directories`：指定头文件路径，`HEAD_PATH` 变量是在根节点文件中定义的
- 第6行 `link_libraries`：指定可执行程序要链接的 静态库，`CALC_LIB` 变量是在根节点文件中定义的
- 第7行 `set`：指定可执行程序生成的路径，`EXEC_PATH` 变量是在根节点文件中定义的
- 第8行 `add_executable`：生成可执行程序，`APP_NAME_1` 变量是在根节点文件中定义的

此处的可执行程序链接的是静态库，最终静态库会被打包到可执行程序中，可执行程序启动之后，静态库也就随之被加载到内存中了。

② 1.2.5 test2 目录

test2 目录中的 `CMakeLists.txt` 文件内容如下：



```
cmake_minimum_required(VERSION 3.0)
project(SORTTEST)
aux_source_directory ./ SRC
include_directories(${HEAD_PATH})
set(EXECUTABLE_OUTPUT_PATH ${EXEC_PATH})
# link_directories(${LIB_PATH})
add_executable(${APP_NAME_2} ${SRC})
target_link_libraries(${APP_NAME_2} ${SORT_LIB})
```

- 第四行 `include_directories`：包含头文件路径，`HEAD_PATH` 变量是在根节点文件中定义的
- 第五行 `set`：指定可执行程序生成的路径，`EXEC_PATH` 变量是在根节点文件中定义的
- 第六行 `link_directories`：指定可执行程序要链接的动态库的路径，`LIB_PATH` 变量是在根节点文件中定义的
- 第七行 `add_executable`：生成可执行程序，`APP_NAME_2` 变量是在根节点文件中定义的
- 第八行 `target_link_libraries`：指定可执行程序要链接的动态库的名字

在生成可执行程序的时候，动态库不会被打包到可执行程序内部。当可执行程序启动之后动态库也不会被加载到内存，只有可执行程序调用了动态库中的函数的时候，动态库才会被加载到内存中，且多个进程可以共用内存中的同一个动态库，所以动态库又叫共享库。

⑥ 1.2.6 构建项目

一切准备就绪之后，开始构建项目，进入到根节点目录的 `build` 目录中，执行 `cmake` 命令，如下：

```
1 $ cmake ..
2 -- The C compiler identification is GNU 5.4.0
3 -- The CXX compiler identification is GNU 5.4.0
4 -- Check for working C compiler: /usr/bin/cc
5 -- Check for working C compiler: /usr/bin/cc -- works
6 -- Detecting C compiler ABI info
7 -- Detecting C compiler ABI info - done
8 -- Detecting C compile features
9 -- Detecting C compile features - done
10 -- Check for working CXX compiler: /usr/bin/c++
11 -- Check for working CXX compiler: /usr/bin/c++ -- works
12 -- Detecting CXX compiler ABI info
13 -- Detecting CXX compiler ABI info - done
14 -- Detecting CXX compile features
15 -- Detecting CXX compile features - done
16 -- Configuring done
17 -- Generating done
18 -- Build files have been written to: /home/robin/abc/cmake/calc/build
```

可以看到在 `build` 目录中生成了一些文件和目录，如下所示：

```
1 $ tree build -L 1
2 build
3   ├── calc          # 目录
4   ├── CMakeCache.txt # 文件
5   ├── CMakeFiles      # 目录
6   ├── cmake_install.cmake # 文件
7   ├── Makefile        # 文件
8   ├── sort           # 目录
9   ├── test1          # 目录
10  └── test2          # 目录
```

然后在 `build` 目录下执行 `make` 命令：

```
robin@os:~/abc/cmake/calc$ cd build/
robin@os:~/abc/cmake/calc/build$ make
Scanning dependencies of target calc
[ 8%] Building CXX object calc/CMakeFiles/calc.dir/div.cpp.o
[ 16%] Building CXX object calc/CMakeFiles/calc.dir/add.cpp.o
[ 25%] Building CXX object calc/CMakeFiles/calc.dir/sub.cpp.o
[ 33%] Building CXX object calc/CMakeFiles/calc.dir/mult.cpp.o
[ 41%] Linking CXX static library ../../lib/libcalc.a
[ 41%] Built target calc
Scanning dependencies of target sort
[ 50%] Building CXX object sort/CMakeFiles/sort.dir/insert.cpp.o
[ 58%] Building CXX object sort/CMakeFiles/sort.dir/select.cpp.o
[ 66%] Linking CXX shared library ../../lib/libsort.so
[ 66%] Built target sort
Scanning dependencies of target test1
[ 75%] Building CXX object test1/CMakeFiles/test1.dir/calc.cpp.o
[ 83%] Linking CXX executable ../../bin/test1
[ 83%] Built target test1
Scanning dependencies of target test2
[ 91%] Building CXX object test2/CMakeFiles/test2.dir/sort.cpp.o
[100%] Linking CXX executable ../../bin/test2
[100%] Built target test2
```

通过上图可以得到如下信息：

1. 在项目根目录的 `lib` 目录中生成了静态库 `libcalc.a`
2. 在项目根目录的 `lib` 目录中生成了动态库 `libsort.so`
3. 在项目根目录的 `bin` 目录中生成了可执行程序 `test1`
4. 在项目根目录的 `bin` 目录中生成了可执行程序 `test2`

最后再来看一下上面提到的这些文件是否真的被生成到对应的目录中了：

最后再来看一下上面提到的这些文件是否真的被生成到对应的目录中了：

```
1 $ tree bin/ lib/
2 bin/
3 |   test1
4 |   test2
5 lib/
6 |   libcalc.a
7 |   libsort.so
```

由此可见，真实不虚，至此，项目构建完毕。

写在最后：

 在项目中，如果将程序中的某个模块制作成了动态库或者静态库 并且在 `CMakeLists.txt` 中指定了库的输出目录，而后其它模块又需要加载这个生成的库文件，此时直接使用就可以了，如果没有指定库的输出路径或者需要直接加载外部提供的库文件，此时就需要使用 `link_directories` 将库文件路径指定出来。

静态库可以链接静态库，也可以链接动态库