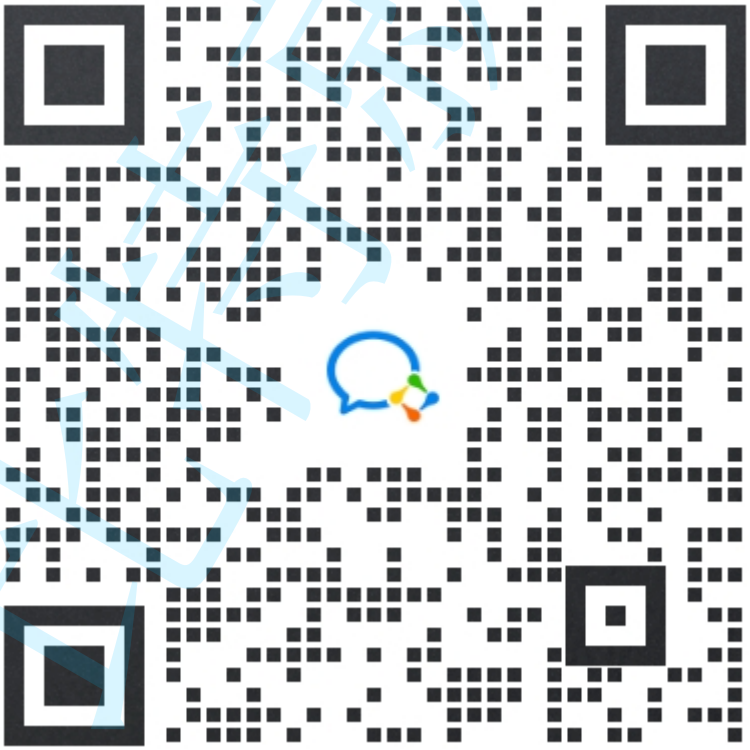
Cmake 使用教程

版权说明

对比特项目感兴趣，可以联系这个微信。



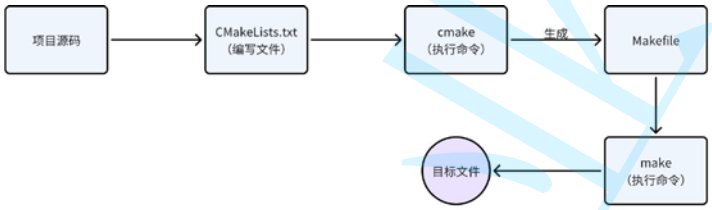
代码 & 板书链接

https://gitee.com/bitedu-tech/cpp-chatsystem

1. CMake 介绍

CMake 是一个开源、跨平台的构建系统，主要用于软件的构建、测试和打包。

CMake 使用平台无关的配置文件 CMakeLists.txt 来控制软件的编译过程，并生成适用于不同编译器环境的项目文件。例如，它可以生成 Unix 系统的 Makefile、Windows 下的 Visual Studio 项目文件或 Mac 的 Xcode 工程文件，从而简化了跨平台和交叉编译的工作流程。CMake 并不直接构建软件，而是产生标准的构建文件，然后使用这些文件在各自的构建环境中构建软件。



CMake 有以下几个特点：

- 开放源代码：使用类 BSD 许可发布
- 跨平台：并可生成编译配置文件，在 Linux/Unix 平台，生成 makefile；在苹果平台，可以生成 xcode；在 Windows 平台，可以生成 MSVC 的工程文件
- 能够管理大型项目：KDE4 就是最好的证明
- 简化编译构建过程和编译过程：Cmake 的工具链非常简单：cmake+make
- 高效率：按照 KDE 官方说法，CMake 构建 KDE4 的 kdelibs 要比使用autotools 来构建 KDE3.5.6 的 kdelibs 快 40%，主要是因为 Cmake 在工具链中没有 libtool
- 可扩展：可以为 cmake 编写特定功能的模块，扩充 cmake 功能

2. CMake 安装

- Ubuntu 22.04 安装 cmake

```
Shell
sudo apt update
sudo apt install cmake
```

确定 cmake 是否安装成功

```
Shell
```

```
cmake --version
```

查看 cmake 版本，至此，cmake 安装成功。

```
root@hcss-ecs-2618:~# cmake --version
cmake version 3.22.1

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

3. 入门样例 - Hello-world 工程

创建 hello-world 目录， 并在其目录下创建 main.cpp 源文件和 CMakeLists.txt 文件

```
Shell
root@hcss-ecs-2618:/home/zsc/cmake# tree hello-world/
hello-world/
├── CMakeLists.txt
└── main.cpp

0 directories, 2 files
```

main.cpp 源文件只是做一个简单的 hello world 打印

```
C++
#include <iostream>

using namespace std;

int main()
{
        std::cout << "hello world" << std::endl;
        return 0;
}
```

CMakeLists.txt 文件如下:

```
CMake
cmake_minimum_required(VERSION 3.0)
project(HELLO)
add_executable(hello main.cpp)
```

此时我们可以使用 cmake 来构建这个工程， 生成 makefile, 从而编译代码。

```
Shell
# cmake 生成 makefile
root@hcss-ecs-2618:/home/zsc/cmake/hello-world# cmake .
```

```
-- The C compiler identification is GNU 11.3.0
-- The CXX compiler identification is GNU 11.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zsc/cmake/hello-world

# 使用 makefile 编译代码
root@hcss-ecs-2618:/home/zsc/cmake/hello-world# make
[ 50%] Building CXX object CMakeFiles/hello.dir/main.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello

# 运行可执行文件
root@hcss-ecs-2618:/home/zsc/cmake/hello-world# ./hello
hello world
```

我们来看一下 CMakeLists.txt 文件，这个文件是 cmake 的构建定义文件，其文件名是大小写相关的。下面依次介绍一下在该文件中添加的三个指令：

- `cmake_minimum_required`：指定使用的 cmake 的最低版本。可选，如果不加会有警告
- `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
project(<PROJECT-NAME> [<language-name>...])
project(<PROJECT-NAME>
        [VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]
        [DESCRIPTION <project-description-string>]
        [HOMEPAGE_URL <url-string>]
        [LANGUAGES <language-name>...])
```

- `add_executable`：定义工程会生成一个可执行程序

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

- 注意：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
# 样式 1
add_executable(app test1.c test2.c test3.c)
# 样式 2
add_executable(app test1.c;test2.c;test3.c)
```

- cmake 命令：将 CMakeLists.txt 文件编辑好之后，就可以执行 cmake 命令了

```Shell
# cmake 命令原型
cmake CMakeLists.txt 文件所在路径
```

当执行 cmake 命令之后，CMakeLists.txt 中的命令就会被执行，所以一定要注意给 cmake 命令指定路径的时候一定不能出错。

4. CMake 的使用

4.1 注释

`CMake` 使用`#`进行行注释， 它可以放在任何位置。

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

CMake 支持大写、小写、混合大小写的命令。如果在编写 CMakeLists.txt 文件时使用的工具有对应的命令提示，那么大小写随缘即可，不要太过在意。

4.2 内部构建和外部构建

对于上面的 hello-world 例子，我们观察一下当我们执行 `cmake .` 指令之后源文件所在的目录是否多了一些文件？

```Shell
root@hcss-ecs-2618:/home/zsc/cmake/hello-world# tree -L 1
.
├── CMakeCache.txt                # new add
├── CMakeFiles                    # new add
├── cmake_install.cmake           # new add
├── CMakeLists.txt
├── main.cpp
└── Makefile                      # new add

1 directory, 5 files
```

我们可以看到如果在 CMakeLists.txt 文件所在目录执行了 cmake 命令之后就会生成一些目录和文件，如果再基于 makefile 文件执行 make 命令，程序在编译过程中还会生成一些中间文件和一个可执行文件，这样会导致整个项目目录看起来很混乱，不太容易管理和维护。这其实被称为 `内部构建`，但 CMake 强烈推荐的做法是 `外部构建`。

此时我们可以把生成的这些与项目源码无关的文件统一放到一个对应的目录里边，比如将这个目录命名为 build，这就叫做 `外部构建`。

```Shell
root@hcss-ecs-2618:/home/zsc/cmake/hello-world# mkdir build
root@hcss-ecs-2618:/home/zsc/cmake/hello-world# cd build/
root@hcss-ecs-2618:/home/zsc/cmake/hello-world/build# cmake ..
-- The C compiler identification is GNU 11.3.0
-- The CXX compiler identification is GNU 11.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zsc/cmake/hello-world/build
```

现在 cmake 命令是在 build 目录中执行的，但是 CMakeLists.txt 文件是 build 目录的上一级目录中，所以 cmake 命令后指定的路径为..，即当前目录的上一级目录。当命令执行完毕之后，在 build 目录中会生成一个 makefile 文件

```Shell
root@hcss-ecs-2618:/home/zsc/cmake/hello-world/build# tree -L 1
.
├── CMakeCache.txt
├── CMakeFiles
├── cmake_install.cmake
└── Makefile

1 directory, 3 files
```

4.3 定义变量

假如我们的项目中存在多个源文件，并且这些源文件需要被反复使用， 每次都直接将它们的名字写出来确实是很麻烦，此时我们就可以定义一个变量，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 指令。

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

`VAR` 表示变量名， `VALUE` 表示变量值。

```Shell
set(SRC_LIST test1.c test2.c test3.c)
add_executable(app  ${SRC_LIST})
```

注意：变量使用${}方式取值，但是在 if 控制语句中是直接使用变量名

4.4 指定使用的 C++标准

在编写 C++程序的时候，可能会用到 C++11、C++14、C++17、C++20 等新特性，那么就需要在编译的时候在编译命令中制定出要使用哪个标准

```Shell
$ g++ *.cpp -std=c++11 -o exec
```

上面的例子中通过参数-std=c++11 指定出要使用 c++11 标准编译程序，C++标准对应有一宏叫做 `CMAKE_CXX_STANDARD`。在 CMake 中想要指定 C++标准有两种方式：

- 在 CMakeLists.txt 中通过 set 命令指定

```CMake
#增加-std=c++11
set(CMAKE_CXX_STANDARD 11)
#增加-std=c++14
set(CMAKE_CXX_STANDARD 14)
```

- 在执行 cmake 命令的时候指定出这个宏的值

```CMake
#增加-std=c++11
cmake CMakeLists.txt 文件路径 -DCMAKE_CXX_STANDARD=11
#增加-std=c++14
cmake CMakeLists.txt 文件路径 -DCMAKE_CXX_STANDARD=14
```

4.5 指定可执行文件输出的路径

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，它的值也是可以通过 set 命令进行设置。

```CMake
# 定义 HOME 变量，存储一个绝对路径
set(HOME /home/xxx)
# 将拼接好的路径值设置给 EXECUTABLE_OUTPUT_PATH 宏
set(EXECUTABLE_OUTPUT_PATH ${HOME}/bin)
```

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。

4.6 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，这样太麻烦了。所以在 CMake 中为我们提供了搜索文件的命令:

- `aux_source_directory`

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

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

- `file`

```CMake
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
  - 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

```CMake
file(GLOB SRC_LISTS ${CMAKE_CURRENT_SOURCE_DIR}/src/*.cpp)
file(GLOB HEAD_LISTS ${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

4.7 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 中设置头文件路径也很简单，通过命令 `include_directories` 就可以搞定了。

```CMake
include_directories(headpath)
```

4.8 生成动态库/静态库

4.8.1 生成静态库

在 cmake 中，如果要制作静态库，需要使用的命令如下：

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

在 Linux 系统中，静态库名字分为三部分：`lib+库名字+.a`，所以库名称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。

4.8.2 生成动态库

在 cmake 中，如果要制作动态库，需要使用的命令如下：

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

在 Linux 系统中，动态库名字分为三部分：`lib+库名字+.so`，所以库名称也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。

4.8.3 指定库输出的路径

我们可以使用 `LIBRARY_OUTPUT_PATH` 宏来指定 动态库/静态库输出的路径。

```CMake
set(LIBRARY_OUTPUT_PATH ${PROJECT_SOURCE_DIR}/lib)
```

4.9 链接动态库/静态库

在编写程序的过程中，可能会用到一些系统提供的动态库或者自己制作出的动态库或者静态库文件，cmake 中也为我们提供了相关的加载静态库/动态库的命令。

4.9.1 链接静态库

在 cmake 中，链接静态库的命令如下：

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

参数为指定要链接的静态库的名字，可以是全名， 也可以是去掉 `lib` 和 `.a` 之后的名字。

如果该静态库是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，此时需要将静态库的路径也指定出来：

```CMake
link_directories(<lib path>)
```

4.9.2 链接动态库

在 cmake 中链接动态库的命令如下：

```CMake
target_link_libraries(
    <target>
    <PRIVATE|PUBLIC|INTERFACE> <item>...
    [<PRIVATE|PUBLIC|INTERFACE> <item>...]...)
```

- target：指定要加载动态库文件的名字
  - 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。
- PRIVATE|PUBLIC|INTERFACE 的区别：
  - 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

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。

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：

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

```
target_link_libraries(app pthread)
```

在 `target_link_libraries(app pthread)`中：app 表示最终生成的可执行程序的名字；pthread 表示可执行程序要加载的动态库， 全名为 `libpthread.so`, 在指定的时候一般会掐头（lib）去尾（.so）。

有些时候，当我们去链接第三方的动态库的时候， 如果不指定链接路径，会报错找不到动态库。此时，我们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：

```
CMake
link_directories(path)
```

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。

4.10 install 指令

install 指令用于定义安装规则，安装的内容可以包括目标二进制、动态库、静态库以及文件、目录、脚 本等。

4.10.1 安装可执行文件和库

```
CMake
INSTALL(TARGETS targets... [[ARCHIVE|LIBRARY|RUNTIME]
[DESTINATION ] [PERMISSIONS permissions...] [CONFIGURATIONS
[Debug|Release|...]] [COMPONENT ] [OPTIONAL] ] [...])
```

- 参数中的 TARGETS 后面跟的就是我们通过 ADD_EXECUTABLE 或者 ADD_LIBRARY 定义的目标文件， 可能是可执行二进制、动态库、静态库。
- 目标类型也就相对应的有三种，ARCHIVE 特指静态库，LIBRARY 特指动态库，RUNTIME 特指可执行目 标二进制。
- DESTINATION 定义了安装的路径，如果路径以/开头，那么指的是绝对路径，这时候 `CMAKE_INSTALL_PREFIX` 其实就无效了。如果你希望使用 `CMAKE_INSTALL_PREFIX` 来 定义安装路径，就要写成相对路径，即不要以/开头，那么安装后的路径就是 `${CMAKE_INSTALL_PREFIX}/...`

例子：

```
CMake
INSTALL(TARGETS myrun mylib mystaticlib RUNTIME DESTINATION bin
LIBRARY DESTINATION lib ARCHIVE DESTINATION libstatic )
```

上面的例子会将:

- 可执行二进制 `myrun` 安装到`${CMAKE_INSTALL_PREFIX}/bin`目录
- 动态库 `libmylib` 安装到`${CMAKE_INSTALL_PREFIX}/lib` 目录
- 静态库 `libmystaticlib` 安装到`${CMAKE_INSTALL_PREFIX}/libstatic` 目录

1. 可以使用-D 选项指定 CMAKE_INSTALL_PREFIX 参数，如 cmake .. -DCMAKE_INSTALL_PREFIX =/usr
2. 如果没有定义 CMAKE_INSTALL_PREFIX 会安装到什么地方？可以尝试一下，cmake ..;make;make install，你会发现 CMAKE_INSTALL_PREFIX 的默认定义 是/usr/local

4.10.2 安装普通文件

```CMake
INSTALL(FILES files... DESTINATION  [PERMISSIONS permissions...]
[CONFIGURATIONS [Debug|Release|...]] [COMPONENT ] [RENAME ]
[OPTIONAL])
```

可用于安装一般文件，并可以指定访问权限，文件名是此指令所在路径下的相对路径。如果默认不定义权限 PERMISSIONS，安装后的权限为：`OWNER_WRITE, OWNER_READ, GROUP_READ,`和`WORLD_READ`，即 644 权限。

4.10.3 安装非目标文件的可执行程序

```CMake
INSTALL(PROGRAMS files... DESTINATION  [PERMISSIONS
permissions...] [CONFIGURATIONS [Debug|Release|...]] [COMPONENT ]
[RENAME ] [OPTIONAL])
```

跟上面的 FILES 指令使用方法一样，唯一的不同是安装后权限为:`OWNER_EXECUTE, GROUP_EXECUTE,`和`WORLD_EXECUTE`，即 755 权限。

4.11 message 指令

在 CMake 中可以使用命令打印消息，该命令的名字为 message。

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

第一个参数通常不设置， 表示重要消息。

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

4.12 宏定义

在某些程序运行的时候，我们可能会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。

```C++
#include <iostream>

using namespace std;

int main()
{
#ifdef DEBUG
        std::cout << "debug info..." << std::endl;
#endif
        std::cout << "hello world" << std::endl;
        return 0;
}
```

在 CMake 中我们也可以使用 add_definitions 来定义宏：

```CMake
cmake_minimum_required(VERSION 3.0)
project(TEST)
```

```
# 自定义 DEBUG 宏
add_definitions(-DDEBUG)
add_executable(app ./example.cpp)
```

下面的列表中为大家整理了一些 CMake 中已经给我们定义好的宏：

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

4.13 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。

嵌套的 CMake 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

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：

```CMake
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。

下面我们实现一个加减乘除的案例：

```Shell
root@hcss-ecs-2618:/home/zsc/cmake/project# tree
.
├── build
├── CMakeLists.txt
├── include
│   └── calc.h
└── src
    ├── add.cpp
    ├── CMakeLists.txt
    ├── main.cpp
    ├── mul.cpp
    └── sub.cpp

3 directories, 7 files
```

- include 目录：头文件目录
- src：源文件目录
- build：是外部构建目录

可以看到目前存在两个 CMakeLists.txt，根目录存在一个，src 源文件目录存在一个，我们依次分析一下各个文件中需要添加的内容：

根目录中的 CMakeLists.txt 文件内容如下：

```CMake
cmake_minimum_required(VERSION 3.0)
project(PROJECT)
# 设置头文件目录变量
set(HEAD_PATH ${CMAKE_CURRENT_SOURCE_DIR}/include)
# 添加子目录，并且指定输出文件的路径为 bin 目录
add_subdirectory(src bin)
```

src 目录中的 CMakeLists.txt 文件内容如下：

```CMake
cmake_minimum_required(VERSION 3.0)
project(CALC)
# 获取当前目录下源文件列表放到 SRC 变量中
aux_source_directory(./ SRC)
# 包含头文件路径
include_directories(${HEAD_PATH})
# 新增可执行文件
add_executable(calc ${SRC})
```

开始构建项目：

```Shell
root@hcss-ecs-2618:/home/zsc/cmake/project/build# cmake ..
-- The C compiler identification is GNU 11.3.0
-- The CXX compiler identification is GNU 11.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zsc/cmake/project/build
```

可以看到在 build 目录中生成了一些文件和目录，如下所示：

```Shell
```

```
root@hcss-ecs-2618:/home/zsc/cmake/project/build# tree -L 1
.
├── bin
├── CMakeCache.txt
├── CMakeFiles
├── cmake_install.cmake
└── Makefile

2 directories, 3 files
```

然后使用 make 编译源码:

```
Shell
root@hcss-ecs-2618:/home/zsc/cmake/project/build# make
[ 20%] Building CXX object bin/CMakeFiles/calc.dir/add.cpp.o
[ 40%] Building CXX object bin/CMakeFiles/calc.dir/main.cpp.o
[ 60%] Building CXX object bin/CMakeFiles/calc.dir/mul.cpp.o
[ 80%] Building CXX object bin/CMakeFiles/calc.dir/sub.cpp.o
[100%] Linking CXX executable calc
[100%] Built target calc
```

可以看到 build/bin 目录下已经产生可执行文件了， 此时运行可执行文件就可以了。

```
Shell
root@hcss-ecs-2618:/home/zsc/cmake/project/build/bin# tree -L 1
.
├── calc
├── CMakeFiles
├── cmake_install.cmake
└── Makefile

1 directory, 3 files
```

运行可执行文件 calc:

```
Shell
root@hcss-ecs-2618:/home/zsc/cmake/project/build/bin# ./calc
a = 20, b = 10
a + b = 30
a - b = 10
a * b = 200
```

5. 综合案例

本章节我们来实现一个综合案例， 该案例涉及到：

1. 动态库和静态库的构建及链接
2. CMake 的嵌套
3. 包含头文件
4. 搜索文件
5. 动态库、静态库、可执行文件的安装

5.1 工程目录结构

先看一下我们综合案例工程的目录：

```
Shell
root@hcss-ecs-2618:/home/zsc/cmake/example# tree
.
├── build
├── calc
│   ├── add.cpp
│   ├── CMakeLists.txt
│   ├── mul.cpp
│   └── sub.cpp
├── CMakeLists.txt
├── include
│   ├── calc.h
│   └── sort.h
├── sort
│   ├── bubble.cpp
│   ├── CMakeLists.txt
│   └── insert.cpp
└── test
    ├── calc.cpp
    ├── CMakeLists.txt
    └── sort.cpp

5 directories, 13 files
```

- include 目录： 头文件目录，两个模块的头文件都放在这里
- calc 目录： 关于计算模块的源码都放在这里
- sort 目录：关于排序模块的源码都放在这里

- test 目录：两个源文件，分别对两个模块进行测试

下面我们分别解析一下每个目录的 `CMakeLists.txt` 文件。

5.2 根目录

```CMake
cmake_minimum_required(VERSION 3.0)
project(EXAMPLE)
# 库生成的路径
set(LIB_PATH ${CMAKE_CURRENT_BINARY_DIR}/lib)
# 可执行文件生成的路径
set(EXEC_PATH ${CMAKE_CURRENT_BINARY_DIR}/bin)
# 头文件目录
set(HEAD_PATH ${CMAKE_CURRENT_SOURCE_DIR}/include)

# 库的名字
set(CALC_LIB calc)
set(SORT_LIB sort)
set(EXECUTABLE_OUTPUT_PATH ${EXEC_PATH})
# 可执行文件的名字
set(APP_NAME_1 test1)
set(APP_NAME_2 test2)

# 添加子目录
add_subdirectory(calc)
add_subdirectory(sort)
add_subdirectory(test)
```

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。

5.3 calc 目录

```CMake
cmake_minimum_required(VERSION 3.0)
project(CALC)
# 获取当前目录的源文件列表放入到 SRC 变量中
aux_source_directory(./ SRC)
# 引入头文件路径
include_directories(${HEAD_PATH})
# 设置库的输出路径
set(LIBRARY_OUTPUT_PATH ${LIB_PATH})
# 封装静态库
```

```
add_library(${CALC_LIB} STATIC ${SRC})
# 安装静态库
install(TARGETS ${CALC_LIB} ARCHIVE DESTINATION libstatic)
```

5.4 sort 目录

```
CMake
cmake_minimum_required(VERSION 3.0)
project(SORT)
# 搜索当前目录下所有的源文件
aux_source_directory(./ SRC)
# 包含头文件路径
include_directories(${HEAD_PATH})
# 设置库的输出路径
set(LIBRARY_OUTPUT_PATH ${LIB_PATH})
# 封装动态库
add_library(${SORT_LIB} SHARED ${SRC})
# 安装动态库
install(TARGETS ${SORT_LIB} LIBRARY DESTINATION lib)
```

5.5 test 目录

```
CMake
cmake_minimum_required(VERSION 3.0)
project(TEST)
# 包含头文件路径
include_directories(${HEAD_PATH})
# 指定链接库的路径
link_directories(${LIB_PATH})
# 链接静态库
link_libraries(${CALC_LIB})
# 设置可执行文件输出路径
set(EXECUTABLE_OUTPUT_PATH ${EXEC_PATH})
# 生成可执行文件
add_executable(${APP_NAME_1} calc.cpp)
add_executable(${APP_NAME_2} sort.cpp)
# 链接动态库
target_link_libraries(${APP_NAME_2} ${SORT_LIB})
# 安装可执行程序
install(TARGETS ${APP_NAME_1} ${APP_NAME_2} RUNTIME DESTINATION
bin)
```

5.6 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

```
Shell
root@hcss-ecs-2618:/home/zsc/cmake/example/build# cmake ..
-- The C compiler identification is GNU 11.3.0
-- The CXX compiler identification is GNU 11.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/zsc/cmake/example/build
```

5.7 编译项目

```
Shell
root@hcss-ecs-2618:/home/zsc/cmake/example/build# make
[  9%] Building CXX object calc/CMakeFiles/calc.dir/add.cpp.o
[ 18%] Building CXX object calc/CMakeFiles/calc.dir/mul.cpp.o
[ 27%] Building CXX object calc/CMakeFiles/calc.dir/sub.cpp.o
[ 36%] Linking CXX static library ../lib/libcalc.a
[ 36%] Built target calc
[ 45%] Building CXX object sort/CMakeFiles/sort.dir/bubble.cpp.o
[ 54%] Building CXX object sort/CMakeFiles/sort.dir/insert.cpp.o
[ 63%] Linking CXX shared library ../lib/libsort.so
[ 63%] Built target sort
[ 72%] Building CXX object test/CMakeFiles/test1.dir/calc.cpp.o
[ 81%] Linking CXX executable ../bin/test1
[ 81%] Built target test1
[ 90%] Building CXX object test/CMakeFiles/test2.dir/sort.cpp.o
[100%] Linking CXX executable ../bin/test2
[100%] Built target test2
```

5.8 安装库和可执行程序

可以使用 make install 直接安装， 默认安装路径为 `/usr/local/...`目录下， 当然也可以在执行 cmake 指令的时候， 加上`-DCMAKE_INSTALL_PREFIX=/usr/`指定头文件和库的安装路径。

```Shell
root@hcss-ecs-2618:/home/zsc/cmake/example/build# make install
Consolidate compiler generated dependencies of target calc
[ 36%] Built target calc
Consolidate compiler generated dependencies of target sort
[ 63%] Built target sort
Consolidate compiler generated dependencies of target test1
[ 81%] Built target test1
Consolidate compiler generated dependencies of target test2
[100%] Built target test2
Install the project...
-- Install configuration: ""
-- Installing: /usr/local/libstatic/libcalc.a
-- Installing: /usr/local/lib/libsort.so
-- Installing: /usr/local/bin/test1
-- Set runtime path of "/usr/local/bin/test1" to ""
-- Installing: /usr/local/bin/test2
-- Set runtime path of "/usr/local/bin/test2" to ""
```