CMake 保姆级教程（下）

发表于 2023-03-15 | 更新于 2023-10-12 | CMake
| 字数总计: 5.8k | 阅读时长: 21分钟 | 阅读量: 24821 | 评论数: 36

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：

```
$ tree
.
├── build
├── calc
│   ├── add.cpp
│   ├── CMakeLists.txt
│   ├── div.cpp
│   ├── mult.cpp
│   └── sub.cpp
├── CMakeLists.txt
├── include
│   ├── calc.h
│   └── sort.h
├── sort
│   ├── CMakeLists.txt
│   ├── insert.cpp
│   └── select.cpp
├── test1
│   ├── calc.cpp
│   └── CMakeLists.txt
```

```
└── test2
    ├── CMakeLists.txt
    └── sort.cpp

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。

1.1 准备工作

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

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：

```
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。

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。

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)
# 静态库的名字
set(CALC_LIB calc)
set(SORT_LIB sort)
# 可执行程序的名字
set(APP_NAME_1 test1)
set(APP_NAME_2 test2)
# 添加子目录
add_subdirectory(calc)
add_subdirectory(sort)
add_subdirectory(test1)
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。

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

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。

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(${HEAD_PATH})
5  link_directories(${LIB_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。

1.2.5 test2 目录

test2 目录中的 `CMakeLists.txt` 文件内容如下：

```
1   cmake_minimum_required(VERSION 3.0)
2   project(SORTTEST)
3   aux_source_directory(./ SRC)
4   include_directories(${HEAD_PATH})
5   set(EXECUTABLE_OUTPUT_PATH ${EXEC_PATH})
6   link_directories(${LIB_PATH})
7   add_executable(${APP_NAME_2} ${SRC})
8   target_link_libraries(${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。

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：

```
$ cmake ..
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- 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/robin/abc/cmake/c
```

可以看到在 build 目录中生成了一些文件和目录，如下所示：

```
$ tree build -L 1
build
├── calc                    # 目录
├── CMakeCache.txt          # 文件
├── CMakeFiles              # 目录
├── cmake_install.cmake     # 文件
├── Makefile                # 文件
├── sort                    # 目录
├── test1                   # 目录
└── 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：

```
$ tree bin/ lib/
bin/
├── test1
└── test2
lib/
├── libcalc.a
└── 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。

2. 流程控制

在 CMake 的 CMakeLists.txt 中也可以进行流程控制，也就是说可以像写 shell 脚本那样进行 条件判断 和 循环 。

2.1 条件判断

关于条件判断其语法格式如下：

```
if(<condition>)
  <commands>
elseif(<condition>) # 可选快, 可以重复
  <commands>
else()              # 可选快
  <commands>
endif()
```

在进行条件判断的时候，如果有多个条件，那么可以写多个 `elseif` ，最后一个条件可以使用 `else` ，但是**开始和结束是必须要成对出现的**，分别为： `if` 和 `endif` 。

2.1.1 基本表达式

```
if(<expression>)
```

如果是基本表达式， `expression` 有以下三种情况： 常量 、 变量 、 字符串 。

- 如果是 `1` ， `ON` ， `YES` ， `TRUE` ， `Y` ，非零值， 非空字符串 时，条件判断返回 `True`
- 如果是 `0` ， `OFF` ， `NO` ， `FALSE` ， `N` ， `IGNORE` ， `NOTFOUND` ， 空字符串 时，条件判断返回 `False`

2.1.2 逻辑判断

- **NOT**

  ```
  if(NOT <condition>)
  ```

  其实这就是一个取反操作，如果条件 `condition` 为 `True` 将返回 `False` ，如果条件 `condition` 为 `False` 将返回 `True` 。

- **AND**

  ```
  if(<cond1> AND <cond2>)
  ```

  如果 `cond1` 和 `cond2` 同时为 `True` ， 返回 `True` 否则返回 `False` 。

- **OR**

  ```
  if(<cond1> OR <cond2>)
  ```

  如果 `cond1` 和 `cond2` 两个条件中至少有一个为 `True` ， 返回 `True` ， 如果两个条件都为 `False` 则返回 `False` 。

2.1.3 比较

- **基于数值的比较**

  ```
  if(<variable|string> LESS <variable|string>)
  if(<variable|string> GREATER <variable|string>)
  if(<variable|string> EQUAL <variable|string>)
  if(<variable|string> LESS_EQUAL <variable|string>)
  if(<variable|string> GREATER_EQUAL <variable|string>)
  ```

  - `LESS` ： 如果左侧数值 小于 右侧，返回 `True`

- GREATER： 如果左侧数值 大于 右侧，返回 True
- EQUAL： 如果左侧数值 等于 右侧，返回 True
- LESS_EQUAL： 如果左侧数值 小于等于 右侧，返回 True
- GREATER_EQUAL： 如果左侧数值 大于等于 右侧，返回 True

- **基于字符串的比较**

```
if(<variable|string> STRLESS <variable|string>)
if(<variable|string> STRGREATER <variable|string>)
if(<variable|string> STREQUAL <variable|string>)
if(<variable|string> STRLESS_EQUAL <variable|string>)
if(<variable|string> STRGREATER_EQUAL <variable|string>)
```

- STRLESS： 如果左侧字符串 小于 右侧，返回 True
- STRGREATER： 如果左侧字符串 大于 右侧，返回 True
- STREQUAL： 如果左侧字符串 等于 右侧，返回 True
- STRLESS_EQUAL： 如果左侧字符串 小于等于 右侧，返回 True
- STRGREATER_EQUAL： 如果左侧字符串 大于等于 右侧，返回 True

2.1.4 文件操作

1. 判断文件或者目录是否存在

```
if(EXISTS path-to-file-or-directory)
```

如果文件或者目录存在返回 True， 否则返回 False 。

2. 判断是不是目录

```
if(IS_DIRECTORY path)
```

- 此处目录的 path 必须是绝对路径
- 如果目录存在返回 True， 目录不存在返回 False 。

3. 判断是不是软连接

```
if(IS_SYMLINK file-name)
```

- 此处的 file-name 对应的路径必须是绝对路径
- 如果软链接存在返回 True， 软链接不存在返回 False 。
- 软链接相当于 Windows 里的快捷方式

4. 判断是不是绝对路径

```
if(IS_ABSOLUTE path)
```

- 关于绝对路径:

- 如果是 Linux， 该路径需要从根目录开始描述
- 如果是 Windows， 该路径需要从盘符开始描述

- 如果是绝对路径返回 True， 如果不是绝对路径返回 False 。

2.1.5 其它

- 判断某个元素是否在列表中

```
if(<variable|string> IN_LIST <variable>)
```

  - CMake 版本要求：大于等于3.3
  - 如果这个元素在列表中返回 True， 否则返回 False 。
- 比较两个路径是否相等

```
if(<variable|string> PATH_EQUAL <variable|string>)
```

  - CMake 版本要求：大于等于3.24
  - 如果这个元素在列表中返回 True， 否则返回 False 。

  关于路径的比较其实就是另个字符串的比较，如果路径格式书写没有问题也可以通过下面这种方式进行比较：

```
if(<variable|string> STREQUAL <variable|string>)
```

  我们在书写某个路径的时候，可能由于误操作会多写几个分隔符，比如把 /a/b/c 写成 /a//b///c， 此时通过 STREQUAL 对这两个字符串进行比较肯定是不相等的，但是通过 PATH_EQUAL 去比较两个路径，得到的结果确实相等的，可以看下面的例子：

```
cmake_minimum_required(VERSION 3.26)
project(test)

if("/home//robin///Linux" PATH_EQUAL "/home/robin/Linux
    message("路径相等")
else()
    message("路径不相等")
endif()

message(STATUS "@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@")

if("/home//robin///Linux" STREQUAL "/home/robin/Linux")
    message("路径相等")
else()
    message("路径不相等")
endif()
```

  输出的日志信息如下:

```
路径相等
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
路径不相等
```

通过得到的结果我们可以得到一个结论：在进行路径比较的时候，如果使用 PATH_EQUAL 可以自动剔除路径中多余的分割线然后再进行路径的对比，使用 STREQUAL 则只能进行字符串比较。

> ☞ 关于 if 的更多条件判断，请参考官方文档

2.2 循环

在 CMake 中循环有两种方式，分别是： `foreach` 和 `while` 。

2.2.1 foreach

使用 foreach 进行循环，语法格式如下：

```
foreach(<loop_var> <items>)
    <commands>
endforeach()
```

通过 `foreach` 我们就可以对 `items` 中的数据进行遍历，然后通过 `loop_var` 将遍历到的当前的值取出，在取值的时候有以下几种用法：

方法1

```
foreach(<loop_var> RANGE <stop>)
```

- `RANGE` ： 关键字，表示要遍历范围
- `stop` ： 这是一个 正整数，表示范围的结束值 ， 在遍历的时候 从 `0` 开始，最大值为 `stop` 。
- `loop_var` ： 存储每次循环取出的值

举例说明：

```
cmake_minimum_required(VERSION 3.2)
project(test)
# 循环
foreach(item RANGE 10)
    message(STATUS "当前遍历的值为: ${item}" )
endforeach()
```

输出的日志信息是这样的：

```
$ cmake ..
-- 当前遍历的值为: 0
```

```
-- 当前遍历的值为: 1
-- 当前遍历的值为: 2
-- 当前遍历的值为: 3
-- 当前遍历的值为: 4
-- 当前遍历的值为: 5
-- 当前遍历的值为: 6
-- 当前遍历的值为: 7
-- 当前遍历的值为: 8
-- 当前遍历的值为: 9
-- 当前遍历的值为: 10
-- Configuring done
-- Generating done
-- Build files have been written to: /home/robin/abc/a/build
```

再次强调：在对一个整数区间进行遍历的时候，得到的范围是这样的 【0，stop】，右侧是闭区间包含 stop 这个值。

方法2

```
foreach(<loop_var> RANGE <start> <stop> [<step>])
```

这是上面 方法1 的加强版，我们在遍历一个整数区间的时候，除了可以指定起始范围，还可以指定步长。

- RANGE： 关键字，表示要遍历范围
- start： 这是一个 正整数，表示范围的起始值，也就是说最小值为 start
- stop： 这是一个 正整数，表示范围的结束值，也就是说最大值为 stop
- step： 控制每次遍历的时候以怎样的步长增长， 默认为1，可以不设置
- loop_var： 存储每次循环取出的值

举例说明：

```
cmake_minimum_required(VERSION 3.2)
project(test)

foreach(item RANGE 10 30 2)
    message(STATUS "当前遍历的值为: ${item}" )
endforeach()
```

输出的结果如下：

```
$ cmake ..
-- 当前遍历的值为: 10
-- 当前遍历的值为: 12
-- 当前遍历的值为: 14
-- 当前遍历的值为: 16
-- 当前遍历的值为: 18
-- 当前遍历的值为: 20
-- 当前遍历的值为: 22
```

```
-- 当前遍历的值为: 24
-- 当前遍历的值为: 26
-- 当前遍历的值为: 28
-- 当前遍历的值为: 30
-- Configuring done
-- Generating done
-- Build files have been written to: /home/robin/abc/a/build
```

再次强调：在使用上面的方式对一个整数区间进行遍历的时候，得到的范围是这样的 【start，stop】，左右两侧都是闭区间，包含 start 和 stop 这两个值，步长 step 默认为1，可以不设置。

方法3

```
foreach(<loop_var> IN [LISTS [<lists>]] [ITEMS [<items>]])
```

这是 foreach 的另一个变体，通过这种方式我们可以对更加复杂的数据进行遍历，前两种方式只适用于对某个正整数范围内的遍历。

- IN： 关键字，表示在 xxx 里边
- LISTS： 关键字，对应的是列表 list ， 通过 set、list 可以获得
- ITEMS： 关键字，对应的也是列表
- loop_var： 存储每次循环取出的值

```
cmake_minimum_required(VERSION 3.2)
project(test)
# 创建 list
set(WORD a b c d)
set(NAME ace sabo luffy)
# 遍历 list
foreach(item IN LISTS WORD NAME)
    message(STATUS "当前遍历的值为: ${item}" )
endforeach()
```

在上面的例子中，创建了两个 list 列表，在遍历的时候对它们两个都进行了遍历（可以根据实际需求选择同时遍历多个或者只遍历一个）。输出的日志信息如下：

```
$ cd build/
$ cmake ..
-- 当前遍历的值为: a
-- 当前遍历的值为: b
-- 当前遍历的值为: c
-- 当前遍历的值为: d
-- 当前遍历的值为: ace
-- 当前遍历的值为: sabo
-- 当前遍历的值为: luffy
-- Configuring done
```

```
-- Generating done
-- Build files have been written to: /home/robin/abc/a/build
```

一共输出了7个字符串，说明遍历是没有问题的。接下来看另外一种方式：

```
cmake_minimum_required(VERSION 3.2)
project(test)

set(WORD a b c "d e f")
set(NAME ace sabo luffy)
foreach(item IN ITEMS ${WORD} ${NAME})
    message(STATUS "当前遍历的值为: ${item}" )
endforeach()
```

在上面的例子中，遍历过程中将关键字 LISTS 改成了 ITEMS ，后边跟的还是一个或者多个列表，只不过此时需要通过 ${} 将列表中的值取出。其输出的信息和上一个例子是一样的：

```
$ cd build/
$ cmake ..
-- 当前遍历的值为: a
-- 当前遍历的值为: b
-- 当前遍历的值为: c
-- 当前遍历的值为: d e f
-- 当前遍历的值为: ace
-- 当前遍历的值为: sabo
-- 当前遍历的值为: luffy
-- Configuring done
-- Generating done
-- Build files have been written to: /home/robin/abc/a/build
```

小细节：在通过 set 组织列表的时候，如果某个字符串中有空格，可以通过双引号将其包裹起来，具体的操作方法可以参考上面的例子。

方法4

注意事项：这种循环方式要求CMake的版本大于等于 3.17。

```
foreach(<loop_var>... IN ZIP_LISTS <lists>)
```

通过这种方式，遍历的还是一个或多个列表，可以理解为是 方式3 的加强版。因为通过上面的方式遍历多个列表，但是又想把指定列表中的元素取出来使用是做不到的，在这个加强版中就可以轻松实现。

- loop_var ： 存储每次循环取出的值，可以根据要遍历的列表的数量指定多个变量，用于存储对应的列表当前取出的那个值。
  - 如果指定了多个变量名，它们的数量应该和列表的数量相等

- 如果只给出了一个 loop_var，那么它将一系列的 loop_var_N 变量来存储对应列表中的当前项，也就是说 loop_var_0 对应第一个列表，loop_var_1 对应第二个列表，以此类推.....
- 如果遍历的多个列表中一个列表较短，当它遍历完成之后将不会再参与后续的遍历（因为其它列表还没有遍历完）。

- IN：关键字，表示在 xxx 里边
- ZIP_LISTS：关键字，对应的是列表 list，通过 set 、list 可以获得

```
cmake_minimum_required(VERSION 3.17)
project(test)
# 通过list给列表添加数据
list(APPEND WORD hello world "hello world")
list(APPEND NAME ace sabo luffy zoro sanji)
# 遍历列表
foreach(item1 item2 IN ZIP_LISTS WORD NAME)
    message(STATUS "当前遍历的值为: item1 = ${item1}, item2=${
endforeach()

message("============================")
# 遍历列表
foreach(item  IN ZIP_LISTS WORD NAME)
    message(STATUS "当前遍历的值为: item1 = ${item_0}, item2=$
endforeach()
```

在这个例子中关于列表数据的添加是通过 list 来实现的。在遍历列表的时候一共使用了两种方式，一种提供了多个变量来存储当前列表中的值，另一种只有一个变量，但是实际取值的时候需要通过 变量名_0、变量名_1、变量名_N 的方式来操作，注意事项：第一个列表对应的编号是0，第一个列表对应的编号是0，第一个列表对应的编号是0。

上面的例子输出的结果如下：

```
$ cd build/
$ cmake ..
-- 当前遍历的值为: item1 = hello, item2=ace
-- 当前遍历的值为: item1 = world, item2=sabo
-- 当前遍历的值为: item1 = hello world, item2=luffy
-- 当前遍历的值为: item1 = , item2=zoro
-- 当前遍历的值为: item1 = , item2=sanji
============================
-- 当前遍历的值为: item1 = hello, item2=ace
-- 当前遍历的值为: item1 = world, item2=sabo
-- 当前遍历的值为: item1 = hello world, item2=luffy
-- 当前遍历的值为: item1 = , item2=zoro
-- 当前遍历的值为: item1 = , item2=sanji
-- Configuring done (0.0s)
-- Generating done (0.0s)
-- Build files have been written to: /home/robin/abc/a/build
```

2.2.2 while

除了使用 `foreach` 也可以使用 `while` 进行循环，关于循环结束对应的条件判断的书写格式和 `if/elseif` 是一样的。 `while` 的语法格式如下：

```
while(<condition>)
    <commands>
endwhile()
```

`while` 循环比较简单，只需要指定出循环结束的条件即可：

```
cmake_minimum_required(VERSION 3.5)
project(test)
# 创建一个列表 NAME
set(NAME luffy sanji zoro nami robin)
# 得到列表长度
list(LENGTH NAME LEN)
# 循环
while(${LEN} GREATER  0)
    message(STATUS "names = ${NAME}")
    # 弹出列表头部元素
    list(POP_FRONT NAME)
    # 更新列表长度
    list(LENGTH NAME LEN)
endwhile()
```

输出的结果如下:

```
$ cd build/
$ cmake ..
-- names = luffy;sanji;zoro;nami;robin
-- names = sanji;zoro;nami;robin
-- names = zoro;nami;robin
-- names = nami;robin
-- names = robin
-- Configuring done (0.0s)
-- Generating done (0.0s)
-- Build files have been written to: /home/robin/abc/a/build
```

可以看到当列表中的元素全部被弹出之后，列表的长度变成了0，此时 `while` 循环也就退出了。