CMake使用

实例一

1.PROJECT

PROJECT 指令的语法是：

PROJECT(projectname [CXX] [C] [Java])

你可以用这个指令定义工程名称，并可指定工程支持的语言，支持的语言列表是可以忽略的，默认情况表示支持所有语言。这个指令隐式的定义了两个 cmake 变量：<projectname>_BINARY_DIR 以及<projectname>_SOURCE_DIR，这里就是 HELLO_BINARY_DIR 和 HELLO_SOURCE_DIR(所以 CMakeLists.txt 中两个 MESSAGE 指令可以直接使用了这两个变量)，因为采用的是内部编译，两个变量目前指的都是工程所在路径/backup/cmake/t1，后面我们会讲到外部编译，两者所指代的内容会有所不同。

同时 cmake 系统也帮助我们预定义了 PROJECT_BINARY_DIR 和 PROJECT_SOURCE_DIR 变量，他们的值分别跟 HELLO_BINARY_DIR 与 HELLO_SOURCE_DIR 一致。

为了统一起见，建议以后直接使用 PROJECT_BINARY_DIR，PROJECT_SOURCE_DIR，即使修改了工程名称，也不会影响这两个变量。如果使用了 <projectname>_SOURCE_DIR，修改工程名称后，需要同时修改这些变量。

2.SET

SET 指令的语法是：

SET(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE])

现阶段，你只需要了解 SET 指令可以用来显式的定义变量即可。

比如我们用到的是 SET(SRC_LIST main.c)，如果有多个源文件，也可以定义成：

SET(SRC_LIST main.c t1.c t2.c)。

3.MESSAGE

MESSAGE 指令的语法是：

MESSAGE([SEND_ERROR | STATUS | FATAL_ERROR] "message to display" ...)

这个指令用于向终端输出用户定义的信息，包含了三种类型：

SEND_ERROR，产生错误，生成过程被跳过。

SATUS，输出前缀为—的信息。

FATAL_ERROR，立即终止所有 cmake 过程.

我们在这里使用的是 STATUS 信息输出，演示了由 PROJECT 指令定义的两个隐式变量 HELLO_BINARY_DIR 和 HELLO_SOURCE_DIR。

4.ADD_EXECUTABLE

ADD_EXECUTABLE(hello ${SRC_LIST})

定义了这个工程会生成一个文件名为 hello 的可执行文件，相关的源文件是 SRC_LIST 中定义的源文件列表， 本例中你也可以直接写成 ADD_EXECUTABLE(hello main.c)。

在本例我们使用了${}来引用变量，这是 cmake 的变量应用方式，但是，有一些例外，比如在 IF 控制语句，变量是直接使用变量名引用，而不需要${}。如果使用了${}去应用变量，其实 IF 会去判断名为${}所代表的值的变量，那当然是不存在的了。

5.基本语法规则

1，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

2，指令(参数 1 参数 2...)

参数使用括弧括起，参数之间使用空格或分号分开。

以上面的 ADD_EXECUTABLE 指令为例，如果存在另外一个 func.c 源文件，就要写成：

ADD_EXECUTABLE(hello main.c func.c)或者

ADD_EXECUTABLE(hello main.c;func.c)

3，指令是大小写无关的，参数和变量是大小写相关的。但，推荐你全部使用大写指令。

这里需要特别解释的是作为工程名的 HELLO 和生成的可执行文件 hello 是没有任何关系的。

hello 定义了可执行文件的文件名，你完全可以写成：

ADD_EXECUTABLE(t1 main.c)

cmake 的语法还是比较灵活而且考虑到各种情况，比如

SET(SRC_LIST main.c)也可以写成 SET(SRC_LIST “main.c”)

是没有区别的，但是假设一个源文件的文件名是 fu nc.c(文件名中间包含了空格)。

这时候就必须使用双引号，如果写成了 SET(SRC_LIST fu nc.c)，就会出现错误，提示你找不到 fu 文件和 nc.c 文件。这种情况，就必须写成：

SET(SRC_LIST “fu nc.c”)

此外，你可以可以忽略掉 source 列表中的源文件后缀，比如可以写成

ADD_EXECUTABLE(t1 main)，cmake 会自动的在本目录查找 main.c 或者 main.cpp 等，当然，最好不要偷这个懒，以免这个目录确实存在一个 main.c 一个 main.

同时参数也可以使用分号来进行分割。

下面的例子也是合法的：

ADD_EXECUTABLE(t1 main.c t1.c)可以写成 ADD_EXECUTABLE(t1 main.c;t1.c).

我们只需要在编写 CMakeLists.txt 时注意形成统一的风格即可。

实例二

1.ADD_SUBDIRECTORY

ADD_SUBDIRECTORY(source_dir [binary_dir] [EXCLUDE_FROM_ALL])

这个指令用于向当前工程添加存放源文件的子目录，并可以指定中间二进制和目标二进制存放的位置。EXCLUDE_FROM_ALL 参数的含义是将这个目录从编译过程中排除，比如，工程的 example，可能就需要工程构建完成后，再进入 example 目录单独进行构建(当然，你也可以通过定义依赖来解决此类问题)。

上面的例子定义了将 src 子目录加入工程，并指定编译输出(包含编译中间结果)路径为 bin 目录。如果不进行 bin 目录的指定，那么编译结果(包括中间结果)都将存放在 build/src 目录(这个目录跟原有的 src 目录对应)，指定 bin 目录后，相当于在编译时将 src 重命名为 bin，所有的中间结果和目标二进制都将存放在 bin 目录。

这里需要提一下的是 SUBDIRS 指令，使用方法是：

SUBDIRS(dir1 dir2...)，但是这个指令已经不推荐使用。它可以一次添加多个子目录，并且，即使外部编译，子目录体系仍然会被保存。

如果我们在上面的例子中将 ADD_SUBDIRECTORY (src bin)修改为 SUBDIRS(src)。

那么在 build 目录中将出现一个 src 目录，生成的目标代码 hello 将存放在 src 目录中。

换个地方保存目标二进制

不论是 SUBDIRS 还是 ADD_SUBDIRECTORY 指令(不论是否指定编译输出目录)，我们都可以通过 SET 指令重新定义 EXECUTABLE_OUTPUT_PATH 和 LIBRARY_OUTPUT_PATH 变量来指定最终的目标二进制的位置(指最终生成的 hello 或者最终的共享库，不包含编译生成的中间文件)

SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)

SET(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)

在第一节我们提到了<projectname>_BINARY_DIR 和 PROJECT_BINARY_DIR 变量，他们指的编译发生的当前目录，如果是内部编译，就相当于 PROJECT_SOURCE_DIR 也就是工程代码所在目录，如果是外部编译，指的是外部编译所在目录，也就是本例中的 build 目录。

2.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。

INSTALL 指令包含了各种安装类型，我们需要一个个分开解释：

目标文件的安装：

```
INSTALL(TARGETS targets...
    [[ARCHIVE|LIBRARY|RUNTIME]
                [DESTINATION <dir>]
                [PERMISSIONS permissions...]
                [CONFIGURATIONS
  [Debug|Release|...]]
                [COMPONENT <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。

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}/<DESTINATION 定义的路径>

CMAKE_INSTALL_PREFIX的常见使用方法如下所示

```
cmake -DCMAKE_INSTALL_PREFIX=/usr .
```

举个简单的例子：

```
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

特别注意的是你不需要关心 TARGETS 具体生成的路径，只需要写上 TARGETS 名称就可以了。

目录的安装：

```
INSTALL(DIRECTORY dirs... DESTINATION <dir>
    [FILE_PERMISSIONS permissions...]
    [DIRECTORY_PERMISSIONS permissions...]
    [USE_SOURCE_PERMISSIONS]
    [CONFIGURATIONS [Debug|Release|...]]
    [COMPONENT <component>]
    [[PATTERN <pattern> | REGEX <regex>]
     [EXCLUDE] [PERMISSIONS permissions...]] [...])
```

这里主要介绍其中的 DIRECTORY、PATTERN 以及 PERMISSIONS 参数。

DIRECTORY 后面连接的是所在 Source 目录的相对路径，但务必注意：

abc 和 abc/有很大的区别。

如果目录名不以/结尾，那么这个目录将被安装为目标路径下的 abc，如果目录名以/结尾，代表将这个目录中的内容安装到目标路径，但不包括这个目录本身。

PATTERN 用于使用正则表达式进行过滤，PERMISSIONS 用于指定 PATTERN 过滤后的文件权限。

我们来看一个例子：

```
INSTALL(DIRECTORY icons scripts/ DESTINATION share/myproj
```

```
        PATTERN "CVS" EXCLUDE
        PATTERN "scripts/*"
        PERMISSIONS OWNER_EXECUTE OWNER_WRITE OWNER_READ
                GROUP_EXECUTE GROUP_READ)
```

这条指令的执行结果是：

将 icons 目录安装到 <prefix>/share/myproj，将 scripts/中的内容安装到 <prefix>/share/myproj

不包含目录名为 CVS 的目录，对于 scripts/*文件指定权限为 OWNER_EXECUTE OWNER WRITE OWNER READ GROUP EXECUTE GROUP READ.

实例三

1.ADD_LIBRARY

```
ADD_LIBRARY(libname    [SHARED|STATIC|MODULE]
  [EXCLUDE_FROM_ALL]
        source1 source2 ... sourceN)
```

你不需要写全 libhello.so，只需要填写 hello 即可，cmake 系统会自动为你生成 libhello.X

类型有三种：

SHARED，动态库

STATIC，静态库

MODULE，在使用 dyld 的系统有效，如果不支持 dyld，则被当作 SHARED 对待。

EXCLUDE_FROM_ALL 参数的意思是这个库不会被默认构建，除非有其他的组件依赖或者手工构建。

2.SET_TARGET_PROPERTIES

```
SET_TARGET_PROPERTIES(target1 target2 ...
            PROPERTIES prop1 value1
            prop2 value2 ...)
```

这条指令可以用来设置输出的名称，对于动态库，还可以用来指定动态库版本和 API 版本。

在本例中，我们需要作的是向 lib/CMakeLists.txt 中添加一条：

SET_TARGET_PROPERTIES(hello_static PROPERTIES OUTPUT_NAME "hello")

这样，我们就可以同时得到 libhello.so/libhello.a 两个库了。

与他对应的指令是：

GET_TARGET_PROPERTY(VAR target property)

具体用法如下例，我们向 lib/CMakeListst.txt 中添加：

```
GET_TARGET_PROPERTY(OUTPUT_VALUE hello_static OUTPUT_NAME)
MESSAGE(STATUS “This is the hello_static
OUTPUT_NAME:”${OUTPUT_VALUE})
```

如果没有这个属性定义，则返回 NOTFOUND.

3.动态版本号

按照规则，动态库是应该包含一个版本号的，我们可以看一下系统的动态库，一般情况是

libhello.so.1.2

libhello.so ->libhello.so.1

libhello.so.1->libhello.so.1.2

为了实现动态库版本号，我们仍然需要使用 SET_TARGET_PROPERTIES 指令。

具体使用方法如下：

SET_TARGET_PROPERTIES(hello PROPERTIES VERSION 1.2 SOVERSION 1)

VERSION 指代动态库版本，SOVERSION 指代 API 版本。

将上述指令加入 lib/CMakeLists.txt 中，重新构建看看结果。

在 build/lib 目录会生成：

libhello.so.1.2

libhello.so.1->libhello.so.1.2

libhello.so ->libhello.so.1

实例四

1.make查看细节

```
make VERBOSE=1
```

2.INCLUDE_DIRECTORIES

INCLUDE_DIRECTORIES([AFTER|BEFORE] [SYSTEM] dir1 dir2 ...)

这条指令可以用来向工程添加多个特定的头文件搜索路径，路径之间用空格分割，如果路径中包含了空格，可以使用双引号将它括起来，默认的行为是追加到当前的头文件搜索路径的后面，你可以通过两种方式来进行控制搜索路径添加的方式：

１，CMAKE_INCLUDE_DIRECTORIES_BEFORE，通过 SET 这个 cmake 变量为 on，可以将添加的头文件搜索路径放在已有路径的前面。

２，通过 AFTER 或者 BEFORE 参数，也可以控制是追加还是置前。

3.LINK_DIRECTORIES

```
LINK_DIRECTORIES(directory1 directory2 ...)
```

这个指令非常简单，添加非标准的共享库搜索路径，比如，在工程内部同时存在共享库和可执行二进制，在编译时就需要指定一下这些共享库的路径。这个例子中我们没有用到这个指令。

4.TARGET_LINK_LIBRARIES

```
TARGET_LINK_LIBRARIES(target library1
                      <debug | optimized> library2
                      ...)
```

这个指令可以用来为 target 添加需要链接的共享库，本例中是一个可执行文件，但是同样可以用于为自己编写的共享库添加共享库链接。

5.特殊的环境变量CMAKE_INCLUDE_PATH和CMAKE_LIBRARY_PATH

> 务必注意，这两个是环境变量而不是 cmake 变量。
>
> 使用方法是要在 bash 中用 export 或者在 csh 中使用 set 命令设置或者 CMAKE INCLUDE PATH=/home/include cmake ..等方式。

这两个变量主要是用来解决以前 autotools 工程中

--extra-include-dir 等参数的支持的。

也就是，如果头文件没有存放在常规路径(/usr/include, /usr/local/include 等)，则可以通过这些变量就行弥补。

我们以本例中的 hello.h 为例，它存放在/usr/include/hello 目录，所以直接查找肯定是找不到的。

前面我们直接使用了绝对路径 INCLUDE_DIRECTORIES(/usr/include/hello)告诉工程这个头文件目录。

为了将程序更智能一点，我们可以使用 CMAKE_INCLUDE_PATH 来进行，使用 bash 的方法如下：

```
export CMAKE_INCLUDE_PATH=/usr/include/hello
```

然后在头文件中将 INCLUDE_DIRECTORIES(/usr/include/hello)替换为：

```
FIND_PATH(myHeader hello.h)
IF(myHeader)
INCLUDE_DIRECTORIES(${myHeader})
ENDIF(myHeader)
```

这里简单说明一下，FIND_PATH 用来在指定路径中搜索文件名，比如：

FIND_PATH(myHeader NAMES hello.h PATHS /usr/include /usr/include/hello)

这里我们没有指定路径，但是，cmake 仍然可以帮我们找到 hello.h 存放的路径，就是因为我们设置了环境变量 CMAKE_INCLUDE_PATH。

如果你不使用 FIND_PATH，CMAKE_INCLUDE_PATH 变量的设置是没有作用的，你不能指望它会直接为编译器命令添加参数-I<CMAKE_INCLUDE_PATH>。

以此为例，CMAKE_LIBRARY_PATH 可以用在 FIND_LIBRARY 中。

同样，因为这些变量直接为 FIND_指令所使用，所以所有使用 FIND_指令的 cmake 模块都会受益。

常用变量和常用环境变量

1.引用方式

前面我们已经提到了，使用${}进行变量的引用。在 IF 等语句中，是直接使用变量名而不通过${}取值

2.自定义变量的方式

主要有隐式定义和显式定义两种，前面举了一个隐式定义的例子，就是 PROJECT 指令，他会隐式的定义<projectname>_BINARY_DIR 和<projectname>_SOURCE_DIR 两个变量。

显式定义的例子我们前面也提到了，使用 SET 指令，就可以构建一个自定义变量了。

比如：

SET(HELLO_SRC main.SOURCE_PATHc)，就 PROJECT_BINARY_DIR 可以通过${HELLO_SRC}来引用这个自定义变量了.

3.常用变量

1.CMAKE_BINARY_DIR PROJECT_BINARY_DIR _BINARY_DIR

这三个变量指代的内容是一致的，如果是 in source 编译，指得就是工程顶层目录，如果是 out-of-source 编译，指的是工程编译发生的目录。PROJECT_BINARY_DIR 跟其他指令稍有区别，现在，你可以理解为他们是一致的。

2.CMAKE_SOURCE_DIR PROJECT_SOURCE_DIR _SOURCE_DIR

这三个变量指代的内容是一致的，不论采用何种编译方式，都是工程顶层目录。

也就是在 in source 编译时，他跟 CMAKE_BINARY_DIR 等变量一致。
PROJECT SOURCE DIR 跟其他指令稍有区别，现在，你可以理解为他们是一致的。

3.CMAKE_CURRENT_SOURCE_DIR

指的是当前处理的 CMakeLists.txt 所在的路径，比如上面我们提到的 src 子目录。

4.CMAKE_CURRENT_BINARY_DIR

如果是 in-source 编译，它跟 CMAKE_CURRENT_SOURCE_DIR 一致，如果是 out-of-source 编译，他指的是 target 编译目录。

使用我们上面提到的 ADD_SUBDIRECTORY(src bin)可以更改这个变量的值。

使用 SET(EXECUTABLE_OUTPUT_PATH <新路径>)并不会对这个变量造成影响，它仅仅修改了最终目标文件存放的路径。

5.CMAKE_CURRENT_LIST_FILE

输出调用这个变量的 CMakeLists.txt 的完整路径

6.CMAKE_CURRENT_LIST_LINE

输出这个变量所在的行

7.CMAKE_MODULE_PATH

这个变量用来定义自己的 cmake 模块所在的路径。如果你的工程比较复杂，有可能会自己编写一些 cmake 模块，这些 cmake 模块是随你的工程发布的，为了让 cmake 在处理 CMakeLists.txt 时找到这些模块，你需要通过 SET 指令，将自己的 cmake 模块路径设置一下。

比如

```
SET(CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/cmake)
```

这时候你就可以通过 INCLUDE 指令来调用自己的模块了。

8.EXECUTABLE_OUTPUT_PATH和LIBRARY_OUTPUT_PATH

分别用来重新定义最终结果的存放目录，前面我们已经提到了这两个变量。

9.PROJECT_NAME

返回通过 PROJECT 指令定义的项目名称.

4.cmake调用环境变量的方式

使用$ENV{NAME}指令就可以调用系统的环境变量了。

比如

```
MESSAGE(STATUS “HOME dir: $ENV{HOME}”)
```

设置环境变量的方式是:

```
SET(ENV{变量名} 值)
```

1.CMAKE_INCLUDE_CURRENT_DIR

自动添加 CMAKE_CURRENT_BINARY_DIR 和 CMAKE_CURRENT_SOURCE_DIR 到当前处理的 CMakeLists.txt。相当于在每个 CMakeLists.txt 加入:

```
INCLUDE_DIRECTORIES(${CMAKE_CURRENT_BINARY_DIR}
${CMAKE_CURRENT_SOURCE_DIR})
```

2.CMAKE_INCLUDE_DIRECTORIES_PROJECT_BEFORE

将工程提供的头文件目录始终至于系统头文件目录的前面，当你定义的头文件确实跟系统发生冲突时可以提供一些帮助。

3.CMAKE_INCLUDE_PATH和CMAKE_LIBRARY_PATH

在上一节已经提及。

5.系统信息

1,CMAKE_MAJOR_VERSION，CMAKE 主版本号，比如 2.4.6 中的 2
2,CMAKE_MINOR_VERSION，CMAKE 次版本号，比如 2.4.6 中的 4
3,CMAKE_PATCH_VERSION，CMAKE 补丁等级，比如 2.4.6 中的 6
4,CMAKE_SYSTEM，系统名称，比如 Linux-2.6.22
5,CMAKE_SYSTEM_NAME，不包含版本的系统名，比如 Linux
6,CMAKE_SYSTEM_VERSION，系统版本，比如 2.6.22
7,CMAKE_SYSTEM_PROCESSOR，处理器名称，比如 i686.
8,UNIX，在所有的类 UNIX 平台为 TRUE，包括 OS X 和 cygwin
9,WIN32，在所有的 win32 平台为 TRUE，包括 cygwin

6.主要的开关项

1，CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS，用来控制 IF ELSE 语句的书写方式，在下一节语法部分会讲到。

2，BUILD_SHARED_LIBS

这个开关用来控制默认的库编译方式，如果不进行设置，使用 ADD_LIBRARY 并没有指定库类型的情况下，默认编译生成的库都是静态库。

如果 SET(BUILD_SHARED_LIBS ON)后，默认生成的为动态库。

３，CMAKE_C_FLAGS

设置 C 编译选项，也可以通过指令 ADD_DEFINITIONS()添加。

4，CMAKE_CXX_FLAGS

设置 C++编译选项，也可以通过指令 ADD DEFINITIONS()添加。

cmake常用指令

一、基本指令

1.ADD_DEFINITIONS

向 C/C++编译器添加-D 定义，比如：

ADD_DEFINITIONS(-DENABLE_DEBUG -DABC)，参数之间用空格分割。

如果你的代码中定义了#ifdef ENABLE_DEBUG #endif，这个代码块就会生效。

如果要添加其他的编译器开关，可以通过 CMAKE_C_FLAGS 变量和 CMAKE_CXX_FLAGS 变量设置。

2.ADD_DEPENDENCIES

定义 target 依赖的其他 target，确保在编译本 target 之前，其他的 target 已经被构建。

```
ADD_DEPENDENCIES(target-name depend-target1
                 depend-target2 ...)
```

3.ADD_EXECUTABLE、ADD_LIBRARY、ADD_SUBDIRECTORY

前面已经介绍。

4.ADD_TEST与ENABLE_TESTING指令

ENABLE_TESTING 指令用来控制 Makefile 是否构建 test 目标，涉及工程所有目录。语法很简单，没有任何参数，ENABLE_TESTING()，一般情况这个指令放在工程的主 CMakeLists.txt 中.

ADD_TEST 指令的语法是:

```
ADD_TEST(testname Exename arg1 arg2 ...)
```

testname 是自定义的 test 名称，Exename 可以是构建的目标文件也可以是外部脚本等等。后面连接传递给可执行文件的参数。如果没有在同一个 CMakeLists.txt 中打开 ENABLE_TESTING()指令，任何 ADD_TEST 都是无效的。

比如我们前面的 Helloworld 例子，可以在工程主 CMakeLists.txt 中添加

```
ADD_TEST(mytest ${PROJECT_BINARY_DIR}/bin/main)
ENABLE_TESTING()
```

生成 Makefile 后，就可以运行 make test 来执行测试了。

5.AUX_SOURCE_DIRECTORY

基本语法是:

```
AUX_SOURCE_DIRECTORY(dir VARIABLE)
```

作用是发现一个目录下所有的源代码文件并将列表存储在一个变量中，这个指令临时被用来自动构建源文件列表。因为目前 cmake 还不能自动发现新添加的源文件。

比如

```
AUX_SOURCE_DIRECTORY(. SRC_LIST)
ADD_EXECUTABLE(main ${SRC_LIST})
```

6.CMAKE_MINIMUM_REQUIRED

其语法为 CMAKE_MINIMUM_REQUIRED(VERSION versionNumber [FATAL_ERROR])

比如 CMAKE_MINIMUM_REQUIRED(VERSION 2.5 FATAL_ERROR)

如果 cmake 版本小与 2.5，则出现严重错误，整个过程中止。

7.EXEC_PROGRAM

在 CMakeLists.txt 处理过程中执行命令，并不会在生成的 Makefile 中执行。具体语法为:

```
EXEC_PROGRAM(Executable [directory in which to run]
                  [ARGS <arguments to executable>]
                  [OUTPUT_VARIABLE <var>]
                  [RETURN_VALUE <var>])
```

用于在指定的目录运行某个程序，通过 ARGS 添加参数，如果要获取输出和返回值，可通过 OUTPUT_VARIABLE 和 RETURN_VALUE 分别定义两个变量.

这个指令可以帮助你在 CMakeLists.txt 处理过程中支持任何命令，比如根据系统情况去修改代码文件等等。

举个简单的例子，我们要在 src 目录执行 ls 命令，并把结果和返回值存下来。

可以直接在src/CMakeLists.txt中添加：

```
EXEC_PROGRAM(ls ARGS "*.c" OUTPUT_VARIABLE LS_OUTPUT RETURN_VALUE
LS_RVALUE)
IF(not LS_RVALUE)
MESSAGE(STATUS "ls result: " ${LS_OUTPUT})
ENDIF(not LS_RVALUE)
```

在cmake 生成Makefile的过程中，就会执行ls命令，如果返回0，则说明成功执行，那么就输出ls *.c的结果。关于IF语句，后面的控制指令会提到。

8.FILE指令

文件操作指令，基本语法为：

```
        FILE(WRITE filename "message to write"... )
        FILE(APPEND filename "message to write"... )
        FILE(READ filename variable)
        FILE(GLOB  variable [RELATIVE path] [globbing
expressions]...)
        FILE(GLOB_RECURSE variable [RELATIVE path]
             [globbing expressions]...)
        FILE(REMOVE [directory]...)
        FILE(REMOVE_RECURSE [directory]...)
        FILE(MAKE_DIRECTORY [directory]...)
        FILE(RELATIVE_PATH variable directory file)
        FILE(TO_CMAKE_PATH path result)
        FILE(TO_NATIVE_PATH path result)
```

这里的语法都比较简单，不在展开介绍了。

9.INCLUDE指令

9，INCLUDE指令，用来载入CMakeLists.txt文件，也用于载入预定义的cmake模块.

```
        INCLUDE(file1 [OPTIONAL])
        INCLUDE(module [OPTIONAL])
```

OPTIONAL参数的作用是文件不存在也不会产生错误。

你可以指定载入一个文件，如果定义的是一个模块，那么将在CMAKE_MODULE_PATH中搜索这个模块并载入。

载入的内容将在处理到INCLUDE语句是直接执行。

二、INSTALL指令

INSTALL系列指令已经在前面的章节有非常详细的说明，这里不在赘述，可参考前面的安装部分。

三、FIND_指令

FIND_系列指令主要包含一下指令：

FIND_FILE(<VAR> name1 path1 path2 ...)
VAR 变量代表找到的文件全路径，包含文件名

FIND_LIBRARY(<VAR> name1 path1 path2 ...)
VAR 变量表示找到的库全路径，包含库文件名

FIND_PATH(<VAR> name1 path1 path2 ...)
VAR 变量代表包含这个文件的路径。

FIND_PROGRAM(<VAR> name1 path1 path2 ...)
VAR 变量代表包含这个程序的全路径。

FIND_PACKAGE(<name> [major.minor] [QUIET] [NO_MODULE]
[[REQUIRED|COMPONENTS] [componets...]])
用来调用预定义在 CMAKE_MODULE_PATH 下的 Find<name>.cmake 模块，你也可以自己定义 Find<name>模块，通过 SET(CMAKE_MODULE_PATH dir)将其放入工程的某个目录中供工程使用，我们在后面的章节会详细介绍 FIND_PACKAGE 的使用方法和 Find 模块的编写。

FIND_LIBRARY 示例：

```
FIND_LIBRARY(libX X11 /usr/lib)
IF(NOT libX)
MESSAGE(FATAL_ERROR “libX not found”)
ENDIF(NOT libX)
```

FIND_PACKAGE():

https://zhuanlan.zhihu.com/p/97369704

四、控制指令

1.IF指令

，基本语法为：

```
IF(expression)
  # THEN section.
  COMMAND1(ARGS ...)
```

```
  COMMAND2(ARGS ...)
  ...
ELSE(expression)
  # ELSE section.
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
ENDIF(expression)
```

另外一个指令是 ELSEIF，总体把握一个原则，凡是出现 IF 的地方一定要有对应的 ENDIF.出现 ELSEIF 的地方，ENDIF 是可选的。

表达式的使用方法如下：

IF(var)，如果变量不是：空，0，N，NO，OFF，FALSE，NOTFOUND 或 <var>_NOTFOUND 时，表达式为真。

IF(NOT var)，与上述条件相反。

IF(var1 AND var2)，当两个变量都为真是为真。

IF(var1 OR var2)，当两个变量其中一个为真时为真。

IF(COMMAND cmd)，当给定的 cmd 确实是命令并可以调用是为真。

IF(EXISTS dir)或者 IF(EXISTS file)，当目录名或者文件名存在时为真。

IF(file1 IS_NEWER_THAN file2)，当 file1 比 file2 新，或者 file1/file2 其中有一个不存在时为真，文件名请使用完整路径。

IF(IS_DIRECTORY dirname)，当 dirname 是目录时，为真。

IF(variable MATCHES regex)

IF(string MATCHES regex)

当给定的变量或者字符串能够匹配正则表达式 regex 时为真。比如：

```
IF("hello" MATCHES "ell")
MESSAGE("true")
ENDIF("hello" MATCHES "ell")
```

IF(variable LESS number)

IF(string LESS number)

IF(variable GREATER number)

IF(string GREATER number)

IF(variable EQUAL number)

IF(string EQUAL number)

数字比较表达式

IF(variable STRLESS string)

IF(string STRLESS string)

IF(variable STRGREATER string)

IF(string STRGREATER string)

IF(variable STREQUAL string)

IF(string STREQUAL string)

按照字母序的排列进行比较.

IF(DEFINED variable)，如果变量被定义，为真。

一个小例子，用来判断平台差异：

```
IF(WIN32)
     MESSAGE(STATUS “This is windows.”)
     #作一些 Windows 相关的操作
ELSE(WIN32)
     MESSAGE(STATUS “This is not windows”)
     #作一些非 Windows 相关的操作
ENDIF(WIN32)
```

上述代码用来控制在不同的平台进行不同的控制，但是，阅读起来却并不是那么舒服，ELSE(WIN32)之类的语句很容易引起歧义。

这就用到了我们在“常用变量”一节提到的 CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS 开关。

可以 SET(CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS ON)

这时候就可以写成：

```
IF(WIN32)
ELSE()
ENDIF()
```

如果配合 ELSEIF 使用，可能的写法是这样：

```
IF(WIN32)
#do something related to WIN32
ELSEIF(UNIX)
#do something related to UNIX
ELSEIF(APPLE)
#do something related to APPLE
ENDIF(WIN32)
```

2.WHILE

WHILE 指令的语法是：

```
WHILE(condition)
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
ENDWHILE(condition)
```

其真假判断条件可以参考 IF 指令。

3.FOREACH

FOREACH 指令的使用方法有三种形式：

1，列表

```
FOREACH(loop_var arg1 arg2 ...)
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
ENDFOREACH(loop_var)
```

像我们前面使用的 AUX_SOURCE_DIRECTORY 的例子

```
AUX_SOURCE_DIRECTORY(. SRC_LIST)
FOREACH(F ${SRC_LIST})
    MESSAGE(${F})
ENDFOREACH(F)
```

2，范围

```
FOREACH(loop_var RANGE total)
ENDFOREACH(loop_var)
```

从 0 到 total 以 1 为步进

举例如下：

```
FOREACH(VAR RANGE 10)
MESSAGE(${VAR})
ENDFOREACH(VAR)
```

最终得到的输出是：

```
0
1
2
3
4
5
6
7
8
9
10
```

３，范围和步进

```
FOREACH(loop_var RANGE start stop [step])
ENDFOREACH(loop_var)
```

从 start 开始到 stop 结束，以 step 为步进，

举例如下

```
FOREACH(A RANGE 5 15 3)
MESSAGE(${A})
ENDFOREACH(A)
```

最终得到的结果是：

```
5
8
11
14
```

这个指令需要注意的是，知道遇到 ENDFOREACH 指令，整个语句块才会得到真正的执行。