CMake 使用教程

一、 CMake 简介

CMake 是一个快平台的自动化建构系统,它使用一个名为 CMakeLists.txt 的文件来描述构建过程,可以产生标准的构建文件,如 Unix 的 Makefile 或 Windows Visual C++的 projects/workspaces。文件 CMakeLists.txt 需要手工编写，也可以通过编写脚本进行半自动化生成。CMake 提供了比 autoconfig 更简洁的语法，在 Linux 平台下使用 CMake 生成 Makefile 并编译的流程如下：

1. 编写 CMakeLists.txt。
2. 执行命令“cmake PATH”或者“ccmake PATH”生成 Makefile（PATH 是 CMakeLists.txt 所在的目录）。
3. 使用 make 命令进行编译。

二、 第一个工程

现在假设我们的项目中只有一个源文件 main.cpp

清单 1 源文件 main.cpp

```
1 #include<iostream>
2
3 int main()
4 {
5     std::cout<<"Hello word!"<<std::endl;
6     return 0;
7 }
```

为了构建该项目,我们需要编写文件 CMakeLists.txt 并将其与 main.cpp 放在 同一个目录下:

清单 2 CMakeLists.txt

```
1 PROJECT(main)
2 CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
3 AUX_SOURCE_DIRECTORY(. DIR_SRCS)
4 ADD_EXECUTABLE(main ${DIR_SRCS})
```

CMakeLists.txt 的语法比较简单,由命令、注释和空格组成,其中命令是不区分大小写的,符号"#"后面的内容被认为是注释。命令由命令名称、小括号和参数组成,参数之间使用空格进行间隔。例如对于清单 2 的 CMakeLists.txt 文件:第一行是一条命令,名称是 PROJECT ,参数是 main ,该命令表示项目的名称是

main 。第二行的命令限定了 CMake 的版本。第三行使用命令 AUX_SOURCE_DIRECTORY 将当前目录中的源文件名称赋值给变量 DIR_SRCS。CMake 手册中对命令 AUX_SOURCE_DIRECTORY 的描述如下：

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

该命令会把<dir>中所有的源文件名称赋值给参数<variable>。第四行使用命令 ADD_EXECUTABLE 指示变量 DIR_SRCS 中的源文件需要编译成一个名称为 main 的可执行文件。

完成了文件 CMakeLists.txt 的编写后 需要使用 cmake 或 ccmake 命令生成 Makefile。ccmake 与命令 cmake 的不同之处在于 ccmake 提供了一个图形化操作的操作界面。cmake 命令的执行方式如下：

```
1 cmake [options] <path-to-source>
```

这里我们进入了 main.cpp 所在的目录后执行“cmake .”后就可以得到 Makefile 并使用 make 进行编译,如图 1 所示：

```
root@ubuntu:~/Documents/linux-c-learn/cmake-learn2# cmake .
-- The C compiler identification is GNU 4.8.4
-- The CXX compiler identification is GNU 4.8.4
-- 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
-- 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
-- Configuring done
-- Generating done
-- Build files have been written to: /root/Documents/linux-c-learn/cmake-learn2
root@ubuntu:~/Documents/linux-c-learn/cmake-learn2# ls
CMakeCache.txt  cmake_install.cmake  main.cpp
CMakeFiles      CMakeLists.txt       Makefile
root@ubuntu:~/Documents/linux-c-learn/cmake-learn2# make
Scanning dependencies of target main
[100%] Building CXX object CMakeFiles/main.dir/main.cpp.o
Linking CXX executable main
[100%] Built target main
root@ubuntu:~/Documents/linux-c-learn/cmake-learn2# ls
CMakeCache.txt  cmake_install.cmake  main      Makefile
CMakeFiles      CMakeLists.txt       main.cpp
root@ubuntu:~/Documents/linux-c-learn/cmake-learn2# ./main
Hello world!
root@ubuntu:~/Documents/linux-c-learn/cmake-learn2#
```

图 1 camke 的运行结果图

三、 处理多源文件目录的方法

CMake 处理源代码分布在不同目录中的情况也十分简单。现假设我们的源代码分布情况如图 2:

图 2 源代码分布情况图

清单 3 源文件 main.cpp

```
#include <iostream>
#include "./src/sum.h"

int main() {
    int x,y,z;
    x = 10;
    y = 20;
    z = sum(&x,&y);
    std::cout << "x + y = " << z << std::endl;
}
```

清单 4 sum.h

```
#include <iostream>
int sum(int *x, int *y);
```

清单 5 sum.cpp

```
#include "sum.h"

int sum(int *x, int *y) {
    return *x + *y;
}
```

其中 src 目录下的文件要编译成一个链接库。

1. 第一步，项目主目录中的 CMakeLists.txt

在目录 cmake-learn3 中创建文件 CMakeLists.txt。文件内容如下:

清单 6 目录 cmake-learn3 中的 CMakeLists.txt

```
PROJECT(main)
CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
ADD_SUBDIRECTORY( src )
AUX_SOURCE_DIRECTORY(. DIR_SRCS)
ADD_EXECUTABLE(main ${DIR_SRCS}  )
TARGET_LINK_LIBRARIES( main sum )
```

相对于清单 2，该文件添加了下面的内容：第三行，使用命令 ADD_SUBDIRECTORY 指明本项目包含一个子目录 src。第六行，使用命令 TARGET_LINK_LIBRARIES 指明可执行文件 main 需要连接一个名为 sum 的链接库 。

2. 第二步，子目录中的 CmakeLists.txt

在子目录 src 中创建 CmakeLists.txt。文件内容如下:

清单 7 目录 src 中的 CmakeLists.txt

```
AUX_SOURCE_DIRECTORY(. DIR_SUM_SRCS)
ADD_LIBRARY ( sum ${DIR_SUM_SRCS})
```

3. 第三步，执行 cmake

至此我们完成了项目中所有 CMakeLists.txt 文件的编写,进入目录 cmake-learn3 中依次执行命令“cmake .”和“make”得到结果如下:

```
root@ubuntu:~/Documents/linux-c-learn/cmake-learn3# make
Scanning dependencies of target sum
[ 50%] Building CXX object src/CMakeFiles/sum.dir/sum.cpp.o
Linking CXX static library libsum.a
[ 50%] Built target sum
Scanning dependencies of target main
[100%] Building CXX object CMakeFiles/main.dir/main.cpp.o
Linking CXX executable main
[100%] Built target main
root@ubuntu:~/Documents/linux-c-learn/cmake-learn3# ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  CMakeLists.txt  main  main.cpp  Makefile  src
root@ubuntu:~/Documents/linux-c-learn/cmake-learn3# ./main
x + y = 30
```

在执行 cmake 的过程中,首先解析目录 cmake-learn3 中的 CMakeLists.txt，当程序执行命令 ADD_SUBDIRECTORY(src) 时进入目录 src 对其中的 CMakeLists.txt 进行解析。

四、 如何使用外部共享库和头文件

本部分内容分两块，第一块是自己构建静态库和动态库，并将其安装到系统中；第二部分就是编写一个程序使用我们构建的共享库。

第一部分：构建静态库与动态库

本部分任务：

1. 建立一个静态库和动态库，提供 PrintName 函数供其他程序编程使用，PrintName 向终端输出“我是 XXX”字符串。
2. 安装头文件和共享库。

(一) 准备工作

在 my-cmake-test/目录下创建 t1 目录，用于存放本部分涉及到的工程。

（二） 建立共享库

cd /root/Documents/linux-c-learn/my-cmake-test/t1

mkdir lib

在 t1 目录下创建 CMakeLists.txt，内容如下：

```
PROJECT(PRINTNAMELIB)
CMAKE_MINIMUM_REQUIRED(VERSION 2.8)
ADD_SUBDIRECTORY(lib)
```

在 lib 目录下建立两个源文件 PrintName.c 与 PrintName.h

PrintName.c 内容如下：

```
#include "PrintName.h"

void PrintName(char* name) {
    printf("我的名字叫：");
    printf("%s\n",name);
}
```

PrintName.h 内容如下：

```
#ifndef PRINTNAME_H
#define PRINTNAME_H
#include <stdio.h>
#include <stdlib.h>
void PrintName(char* name);
#endif
```

在 lib 目录下建立 CMakeList.txt，内容如下：

```
SET(LIBPRINTNAME_SRC PrintName.c)
ADD_LIBRARY(printname SHARED ${LIBPRINTNAME_SRC})
```

（三） 编译共享库

采用 out-of-source 编译的方式，按照习惯，我们建立一个 build 目录，在 build 目录中

cmake ..

make

这时，你就可以在 lib 目录得到一个 libprintname.so，这就是我们期望的共享库。

（四） 添加静态库

同样使用上面的指令，我们在支持动态库的基础上再为工程添加一个静态库，按照一般的习惯，静态库名字跟动态库名字应该是一致的，只不过后缀是.a 罢了。

下面我们用这个指令再来添加静态库：

ADD_LIBRARY(printname STATIC ${LIBHELLO_SRC})（测试可以）

然后再在 build 目录进行外部编译，我们会发现，静态库根本没有被构建，仍然只生成了一个动态库。因为 printname 作为一个 target 是不能重名的，所以，静态库构建指令无效。

如果我们把上面的 hello 修改为 printname_static:

ADD_LIBRARY(printname_static STATIC ${LIBHELLO_SRC})（测试可以）

就可以构建一个 libprintname_static.a 的静态库了。

这种结果显示不是我们想要的，我们需要的是名字相同的静态库和动态库，因为 target 名称是唯一的，所以，我们肯定不能通过 ADD_LIBRARY 指令来实现了。这时候我们需要用到另外一个指令：

我们需要做的是向 lib/CMakeLists.txt 中添加两条：

ADD_LIBRARY(printname_static STATIC ${LIBHELLO_SRC})

SET_TARGET_PROPERTIES(printname_static PROPERTIES OUTPUT_NAME "printname")

注释：（先命名成 libprintname_static.a，再改名成 libprintname.a）

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

（五） 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：

libprintname.so.1.2

libprintname.so ->libprintname.so.1

libprintname.so.1->libprintname.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(printname 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：

```
libprintname.so -> libprintname.so.1
libprintname.so.1 -> libprintname.so.1.2
libprintname.so.1.2
```

（六） 安装共享库和头文件

我们需要将 libprintname.a， libprintname.so.x 以及 PrintName.h 安装到系统目录，才能真正让其他人开发使用，在本例中我们将 printname 的共享库安装到<prefix>/lib 目录，将 printname.h 安装到<prefix>/include/printname 目录。

利用 INSTALL 指令，向 lib/CMakeLists.txt 中添加如下指令：

```
INSTALL(TARGETS printname printname_static
LIBRARY DESTINATION lib
ARCHIVE DESTINATION lib)
INSTALL(FILES PrintName.h DESTINATION include/printname)
```

注意，静态库要使用 ARCHIVE 关键字。

通过：

cmake -DCMAKE_INSTALL_PREFIX=/usr ..

make

make install

我们就可以将文件和共享库安装到系统目录 /usr/lib 和 /usr/include/printname

最终的 lib/CMakeLists.txt 文件内容如下：

```
SET(LIBPRINTNAME_SRC PrintName.c)
ADD_LIBRARY(printname SHARED ${LIBPRINTNAME_SRC})
ADD_LIBRARY(printname_static STATIC ${LIBPRINTNAME_SRC})
SET_TARGET_PROPERTIES(printname_static PROPERTIES OUTPUT_NAME
"printname")
SET_TARGET_PROPERTIES(printname PROPERTIES VERSION 1.2 SOVERSION 1)
INSTALL(TARGETS printname printname_static
LIBRARY DESTINATION lib
ARCHIVE DESTINATION lib)
INSTALL(FILES PrintName.h DESTINATION include/printname)
```

第二部分：使用共享库和头文件

上一节我们已经完成了 libprintname 动态库的构建以及安装，本节我们的任务很简单：编写一个程序使用我们上一节构建的共享库。

(一) 准备工作

在 my-cmake-test/目录下创建 t2 目录，用于存放本部分涉及到的工程。

(二) 编写源文件

重复之前的步骤，在 t2 目录下建立 src 目录，编写源文件 main.c，内容如下：

```
#include <PrintName.h>
```

```
int main() {
    char *name = "Mahaitao";
    PrintName(name);
    return 0;
}
```

在 t2 根目录下编写主过程文件 CMakeLists.txt，内容如下：

```
CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
PROJECT(PRINTMYNAME)
ADD_SUBDIRECTORY(src)
```

（三） 外部构建

按照习惯，仍然建立 build 目录，使用 cmake ..方式构建。

过程：

cmake ..

make

构建失败，错误输出如下：

```
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build# cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /root/Documents/linux-c-learn/my-cmake-test/t2/build
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build# make
Scanning dependencies of target main
[100%] Building C object src/CMakeFiles/main.dir/main.c.o
/root/Documents/linux-c-learn/my-cmake-test/t2/src/main.c:1:23: fatal error: PrintName.h: No such file or direc
tory
 #include <PrintName.h>
                       ^
compilation terminated.
make[2]: *** [src/CMakeFiles/main.dir/main.c.o] Error 1
make[1]: *** [src/CMakeFiles/main.dir/all] Error 2
make: *** [all] Error 2
```

（四） 引入头文件搜索路径

PrintName.h 位于/usr/include/printname 目录中，并没有位于系统标准的头文件路径，为了让我们的工程能够找到 PrintName.h 头文件，我们需要引入一个新的指令：

INCLUDE_DIRECTORIES，其完整语法为：

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

现在我们在 src/CMakeLists.txt 中添加一个头文件搜索路径，方式很简单，加入：

INCLUDE_DIRECTORIES(/usr/include/printname)

进入 build 目录，重新进行构建，这是找不到 PrintName.h 的错误已经消失，但是出现了一个新的错误：

```
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build# cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /root/Documents/linux-c-learn/my-cmake-test/t2/build
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build# make
Scanning dependencies of target main
[100%] Building C object src/CMakeFiles/main.dir/main.c.o
Linking C executable main
CMakeFiles/main.dir/main.c.o: In function `main':
main.c:(.text+0x18): undefined reference to `PrintName'
collect2: error: ld returned 1 exit status
make[2]: *** [src/main] Error 1
make[1]: *** [src/CMakeFiles/main.dir/all] Error 2
make: *** [all] Error 2
```

因为我们并没有 link 到共享库 libprintname 上。

（五） 为 target 添加共享库

我们现在需要完成的任务是将目标文件链接到 libprintname。为了解决我们前面遇到的 HelloFunc 未定义错误，我们需要作的是向 src/CMakeLists.txt 中添加如下指令：

TARGET_LINK_LIBRARIES(main printname)

也可以写成：

TARGET_LINK_LIBRARIES(main printname.so)

这里的 printname 指的是我们上一节构建的共享库 libprintname。

此时 src/CMakeLists.txt 完整的定义如下：

```
ADD_EXECUTABLE(main main.c)
INCLUDE_DIRECTORIES(/usr/include/printname)
TARGET_LINK_LIBRARIES(main printname)
```

进入 build 目录重新进行构建。

cmake ..

make

这时我们就得到了一个连接到 libprintname 的可执行程序 main，位于 build/src 目录。运行结果如下：

```
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build# cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /root/Documents/linux-c-learn/my-cmake-test/t2/build
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build# make
Linking C executable main
[100%] Built target main
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build# ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile  src
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build# cd src/
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build/src# ls
CMakeFiles  cmake_install.cmake  main  Makefile
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build/src# ./main
我的名字叫：Mahaitao
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build/src# ldd main
        linux-vdso.so.1 =>  (0x00007fffc4de4000)
        libprintname.so.1 => /usr/lib/libprintname.so.1 (0x00007f592740b000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5927042000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f592760d000)
root@ubuntu:~/Documents/linux-c-learn/my-cmake-test/t2/build/src#
```

我们来检查一下 main 的链接情况，可以清楚的看到 main 确实链接了共享库 libprintname，而且链接的是动态库 libhello.so.1

那如何链接到静态库呢？

方法很简单：

将 TARGET_LINK_LIBRRARIES 指令修改为:

TARGET_LINK_LIBRARIES(main libhello.a)

五、 语法总结

1. 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

同时 cmake 系统也帮助我们预定义了 PROJECT_BINARY_DIR 和 PROJECT_SOURCE_DIR 变量。

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 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 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。

4. ADD_SUBDIRECTORY 指令的语法是：

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)。

5. SUBDIRS 指令的语法是：

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。

6. ADD_LIBRARY 指令的语法是：

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，若不支持 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。

7. SET_TARGET_PROPERTIES 指令的语法是：

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。

8. INCLUDE_DIRECTORIES 指令的语法是（引入头文件搜索路径）：

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。

9. LINK_DIRECTORIES 和 TARGET_LINK_LIBRARIES 指令的语法是(为 target 添加共享库)：

LINK_DIRECTORIES 的全部语法是：

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。

TARGET_LINK_LIBRARIES 的全部语法是:

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。

10. INSTALL 指令的语法是：

1) 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。

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。

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

2) 普通文件的安装：

```
INSTALL(FILES files... DESTINATION <dir>
[PERMISSIONS permissions...]
[CONFIGURATIONS [Debug|Release|...]]
[COMPONENT <component>]
[RENAME <name>] [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。

3) 非目标文件的可执行程序安装（比如脚本之类）：

```
INSTALL(PROGRAMS files... DESTINATION <dir>
[PERMISSIONS permissions...]
[CONFIGURATIONS [Debug|Release|...]]
[COMPONENT <component>]
[RENAME <name>] [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) 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。