CMake使用教程

1. 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。
4. 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：

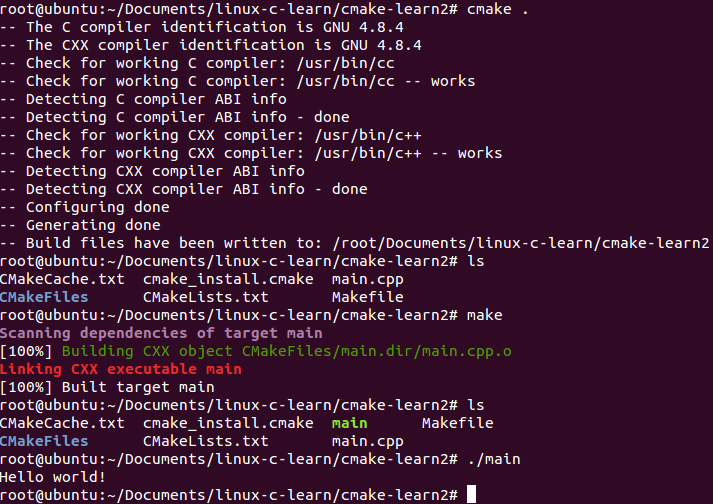
|  |
| --- |
| 1 aux\_source\_directory(<dir> <variable>) |

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：



|  |
| --- |
| 图 1 camke 的运行结果图 |

1. 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 。

1. 第二步，子目录中的 CmakeLists.txt

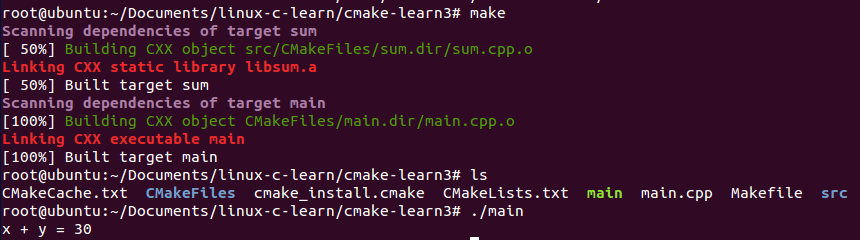
在子目录 src 中创建 CmakeLists.txt。文件内容如下:

清单7 目录src中的CmakeLists.txt

|  |
| --- |
| AUX\_SOURCE\_DIRECTORY(. DIR\_SUM\_SRCS)  ADD\_LIBRARY ( sum ${DIR\_SUM\_SRCS}) |

1. 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:



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。

1. 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。
3. 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。

1. 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}) |

1. 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。

1. 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。

1. 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：

C:\Users\hitao_ma\AppData\Local\Temp\1521512022(1).png

1. 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。

1. 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。

1. 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) |

1. 外部构建

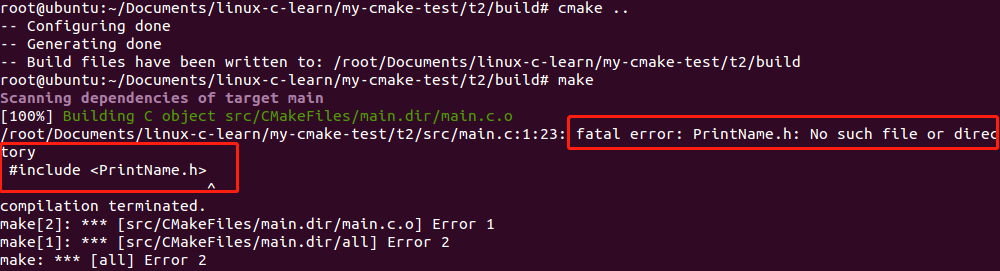
按照习惯，仍然建立build目录，使用cmake ..方式构建。

过程：

cmake ..

make

构建失败，错误输出如下：



1. 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：

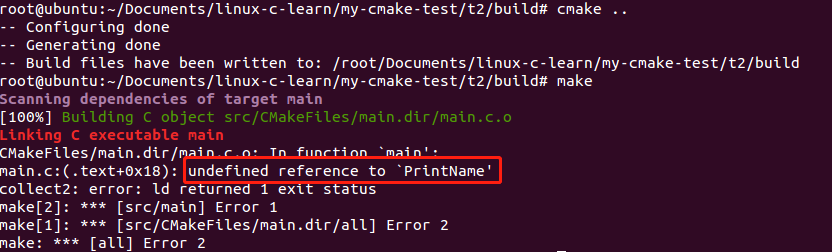
INCLUDE\_DIRECTORIES，其完整语法为：

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：



因为我们并没有link到共享库libprintname上。

1. 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：

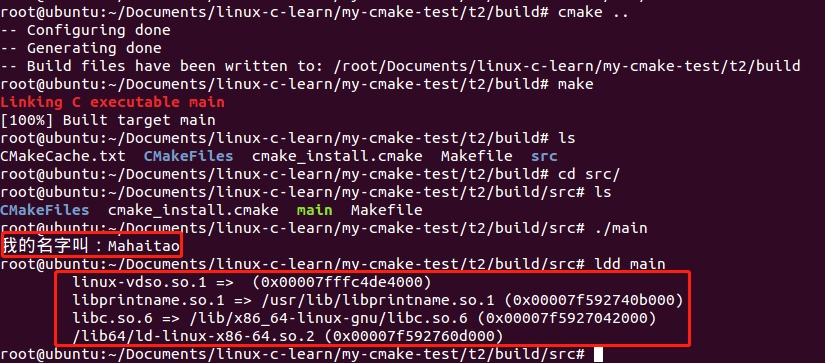
|  |
| --- |
| ADD\_EXECUTABLE(main main.c)  INCLUDE\_DIRECTORIES(/usr/include/printname)  TARGET\_LINK\_LIBRARIES(main printname) |

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：



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)

1. 语法总结
2. 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。

1. 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)。

1. 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。

1. 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)。

1. 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。

1. 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。

1. 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。

1. 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。

1. 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。

1. INSTALL指令的语法是：
2. 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

1. 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。

1. 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

1. 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。