* + **command**

*这个是实用过程中最长用到的，相当于一般脚步语言中的基本语法，包括定义变量，foreach，string，if，builtin command都在这里。*

*可以用如下这些命令获取帮助：*

*cmake --help-commands*

*这个命令将给出所有cmake内置的命令的详细帮助，一般不知道自己要找什么或者想随机翻翻得时候，可以用这个。*

*我一般更常用的方法是将其重定向到less里边，然后在编辑器里边搜索关键字。*

*另外也可以用如下的办法层层缩小搜索范围：*

*cmake --help-command-list*

*cmake --help-command-list | grep find*

*skyscribe@skyscribe:~/program/ltesim/bld$ cmake --help-command-list | grep find  
find\_file  
find\_library  
find\_package  
find\_path  
find\_program*

*cmake --help-command find\_library*

*cmake version 2.6-patch 4  
------------------------------------------------------------------------------  
SingleItem*

*find\_library  
       Find a library.*

*find\_library(<VAR> name1 [path1 path2 ...])*

*This is the short-hand signature for the command that is sufficient in  
       many cases.  It is the same as find\_library(<VAR> name1 [PATHS path1  
       path2 ...])*

*find\_library(  
                    <VAR>  
                    name | NAMES name1 [name2 ...]  
                    [HINTS path1 [path2 ... ENV var]]  
                    [PATHS path1 [path2 ... ENV var]]  
                    [PATH\_SUFFIXES suffix1 [suffix2 ...]]  
                    [DOC "cache documentation string"]  
                    [NO\_DEFAULT\_PATH]  
                    [NO\_CMAKE\_ENVIRONMENT\_PATH]  
                    [NO\_CMAKE\_PATH]  
                    [NO\_SYSTEM\_ENVIRONMENT\_PATH]  
                    [NO\_CMAKE\_SYSTEM\_PATH]  
                    [CMAKE\_FIND\_ROOT\_PATH\_BOTH |  
                     ONLY\_CMAKE\_FIND\_ROOT\_PATH |  
                     NO\_CMAKE\_FIND\_ROOT\_PATH]  
                   )*

* + **variable**

*和command的帮助比较类似，只不过这里可以查找cmake自己定义了那些变量你可以直接使用，譬如OSName，是否是Windows，Unix等。*

*我最常用的一个例子：*

*cmake --help-variable-list | grep CMAKE | grep HOST*

*CMAKE\_HOST\_APPLE*

*CMAKE\_HOST\_SYSTEM*

*CMAKE\_HOST\_SYSTEM\_NAME*

*CMAKE\_HOST\_SYSTEM\_PROCESSOR*

*CMAKE\_HOST\_SYSTEM\_VERSION*

*CMAKE\_HOST\_UNIX*

*CMAKE\_HOST\_WIN32*

*这里查找所有CMake自己定义的builtin变量；一般和系统平台相关。*

*如果希望将所有生成的可执行文件、库放在同一的目录下，可以如此做：*

*这里的target\_dir是一个实现设置好的绝对路径。（CMake里边绝对路径比相对路径更少出问题，如果可能尽量用绝对路径）*

*# Targets directory*

*set(CMAKE\_ARCHIVE\_OUTPUT\_DIRECTORY ${target\_dir}/lib)*

*set(CMAKE\_LIBRARY\_OUTPUT\_DIRECTORY ${target\_dir}/lib)*

*set(CMAKE\_RUNTIME\_OUTPUT\_DIRECTORY ${target\_dir}/bin)*

* + **property**

*Property一般很少需要直接改动，除非你想修改一些默认的行为，譬如修改生成的动态库文件的soname等。*

*譬如需要在同一个目录下既生成动态库，也生成静态库，那么默认的情况下，cmake根据你提供的target名字自动生成类似的libtarget.so, libtarget.a，但是同一个project只能同时有一个，因为target必须唯一。*

*这时候，就可以通过修改taget对应的文件名，从而达到既生成动态库也产生静态库的目的。*

*譬如：*

*cmake --help-property-list | grep NAME*

*GENERATOR\_FILE\_NAME*

*IMPORTED\_SONAME*

*IMPORTED\_SONAME\_<CONFIG>*

*INSTALL\_NAME\_DIR*

*OUTPUT\_NAME*

*VS\_SCC\_PROJECTNAME*

*skyscribe@skyscribe:~$ cmake --help-property OUTPUT\_NAME*

*cmake version 2.6-patch 4*

*------------------------------------------------------------------------------*

*SingleItem*

*OUTPUT\_NAME*

*Sets the real name of a target when it is built.*

*Sets the real name of a target when it is built and can be used to*

*help create two targets of the same name even though CMake requires*

*unique logical target names. There is also a <CONFIG>\_OUTPUT\_NAME*

*that can set the output name on a per-configuration basis.*

* + **module**

*用于查找常用的模块，譬如boost，bzip2, python等。通过简单的include命令包含预定义的模块，就可以得到一些模块执行后定义好的变量，非常方便。*

*譬如常用的boost库，可以通过如下方式：*

*# Find boost 1.40*

*INCLUDE(FindBoost)*

*find\_package(Boost 1.40.0 COMPONENTS thread unit\_test\_framework)*

*if(NOT Boost\_FOUND)*

*message(STATUS "BOOST not found, test will not succeed!")*

*endif()*

*一般开头部分的解释都相当有用，可满足80%需求：*

*cmake --help-module FindBoost | head -40*

*cmake version 2.6-patch 4*

*------------------------------------------------------------------------------*

*SingleItem*

*FindBoost*

*Try to find Boost include dirs and libraries*

*Usage of this module as follows:*

*== Using Header-Only libraries from within Boost: ==*

*find\_package( Boost 1.36.0 )*

*if(Boost\_FOUND)*

*include\_directories(${Boost\_INCLUDE\_DIRS})*

*add\_executable(foo foo.cc)*

*endif()*

*== Using actual libraries from within Boost: ==*

*set(Boost\_USE\_STATIC\_LIBS ON)*

*set(Boost\_USE\_MULTITHREADED ON)*

*find\_package( Boost 1.36.0 COMPONENTS date\_time filesystem system ... )*

*if(Boost\_FOUND)*

*include\_directories(${Boost\_INCLUDE\_DIRS})*

*add\_executable(foo foo.cc)*

*target\_link\_libraries(foo ${Boost\_LIBRARIES})*

*endif()*

*The components list needs to contain actual names of boost libraries*

* **如何根据其生成的中间文件查看一些关键信息**

*CMake相比较于autotools的一个优势就在于其生成的中间文件组织的很有序，并且清晰易懂，不像autotools会生成天书一样的庞然大物（10000+的不鲜见）。*

*一般CMake对应的Makefile都是有层级结构的，并且会根据你的CMakeLists.txt间的相对结构在binary directory里边生成相应的目录结构。*

*譬如对于某一个target，一般binary tree下可以找到一个文件夹:  CMakeFiles/<targentName>.dir/,比如：*

*skyscribe@skyscribe:~/program/ltesim/bld/dev/simcluster/CMakeFiles/SIMCLUSTER.dir$ ls -l*

*total 84*

*-rw-r--r-- 1 skyscribe skyscribe 52533 2009-12-12 12:20 build.make*

*-rw-r--r-- 1 skyscribe skyscribe 1190 2009-12-12 12:20 cmake\_clean.cmake*

*-rw-r--r-- 1 skyscribe skyscribe 4519 2009-12-12 12:20 DependInfo.cmake*

*-rw-r--r-- 1 skyscribe skyscribe 94 2009-12-12 12:20 depend.make*

*-rw-r--r-- 1 skyscribe skyscribe 573 2009-12-12 12:20 flags.make*

*-rw-r--r-- 1 skyscribe skyscribe 1310 2009-12-12 12:20 link.txt*

*-rw-r--r-- 1 skyscribe skyscribe 406 2009-12-12 12:20 progress.make*

*drwxr-xr-x 2 skyscribe skyscribe 4096 2009-12-12 12:20 src*

*这里，每一个文件都是个很短小的文本文件，内容相当清晰明了。build.make一般包含中间生成文件的依赖规则，DependInfo.cmake一般包含源代码文件自身的依赖规则。*

*比较重要的是flags.make和link.txt，前者一般包含了类似于GCC的-I的相关信息，如搜索路径，宏定义等；后者则包含了最终生成target时候的linkage信息，库搜索路径等。*

*这些信息在出现问题的时候是个很好的辅助调试手段。*

* **文件查找、路径相关**
  + **include**

*一般常用的是：*

*include\_directories（）用于添加头文件的包含搜索路径*

*cmake --help-command include\_directories*

*cmake version 2.6-patch 4*

*------------------------------------------------------------------------------*

*SingleItem*

*include\_directories*

*Add include directories to the build.*

*include\_directories([AFTER|BEFORE] [SYSTEM] dir1 dir2 ...)*

*Add the given directories to those searched by the compiler for*

*include files. By default the directories are appended onto the*

*current list of directories. This default behavior can be changed by*

*setting CMAKE\_include\_directories\_BEFORE to ON. By using BEFORE or*

*AFTER you can select between appending and prepending, independent*

*from the default. If the SYSTEM option is given the compiler will be*

*told that the directories are meant as system include directories on*

*some platforms.*

*link\_directories()用于添加查找库文件的搜索路径*

*cmake --help-command link\_directories*

*cmake version 2.6-patch 4*

*------------------------------------------------------------------------------*

*SingleItem*

*link\_directories*

*Specify directories in which the linker will look for libraries.*

*link\_directories(directory1 directory2 ...)*

*Specify the paths in which the linker should search for libraries.*

*The command will apply only to targets created after it is called.*

*For historical reasons, relative paths given to this command are*

*passed to the linker unchanged (unlike many CMake commands which*

*interpret them relative to the current source directory).*

* + **library search**

*一般外部库的link方式可以通过两种方法来做，一种是显示添加路径，采用link\_directories()， 一种是通过find\_library()去查找对应的库的绝对路径。*

*后一种方法是更好的，因为它可以减少不少潜在的冲突。*

        一般find\_library会根据一些默认规则来搜索文件，如果找到，将会set传入的第一个变量参数、否则，对应的参数不被定义，并且有一个xxx-NOTFOUND被定义；可以通过这种方式来调试库搜索是否成功。

        对于库文件的名字而言，动态库搜索的时候会自动搜索libxxx.so (xxx.dll),静态库则是libxxx.a（xxx.lib），对于动态库和静态库混用的情况，可能会出现一些混乱，需要格外小心；一般尽量做匹配连接。

* + **rpath**

*所谓的rpath是和动态库的加载运行相关的。我一般采用如下的方式取代默认添加的rpath：*

*# RPATH and library search setting*

*SET(CMAKE\_SKIP\_BUILD\_RPATH FALSE)*

*SET(CMAKE\_BUILD\_WITH\_INSTALL\_RPATH FALSE)*

*SET(CMAKE\_INSTALL\_RPATH "${CMAKE\_INSTALL\_PREFIX}/nesim/lib")*

*SET(CMAKE\_INSTALL\_RPATH\_USE\_LINK\_PATH TRUE)*