CMake 语法简介

CMake 特点

1. 在每个源码目录下都有一个 CMakeLists.txt.
2. CMake 语句不区分大小写。一句一行，无行结束符号，注释用#
3. CMake 实际也是一种编程语言。CMake 根据 CMakeLists.txt 自动生成 Makefile.
4. CMake 比 Autotools 更简单明了

CMake 语法

语法规则

1. 变量使用**${}**方式取值，但是在 IF 语句中是直接使用变量名取值

   MESSAGE(STATUS “This is bin dir” $(PROJECT_BINARY_DIR))

   MESSAGE(STATUS “This is bin dir $(PROJECT_BINARY_DIR)”)

   上面两句等效。
2. 指令（参数 1 参数 2 …）,参数之间用空格或分号隔开。

   SET(SRC_LIST main.cpp hello.cpp)

   SET(SRC_LIST “main.cpp” “hello.cpp”)

   SET(SRC_LIST “main.cpp”;“hello.cpp”)
3. 内部构建和外部构建：在哪个目录下执行 cmake 命令,则在哪个目录构建

   In-source：编译过程文件和源码文件在同一目录下面(在工程目录下 cmake)

   Out-of-sourc:将编译目录和源码目录分割开(在非工程目录下 cmake)。
4. 常用变量及指令

CMake 变量		
序号	语句	注释
1	PROJECT_BINARY_DIR、PROJECT_SOURCE_DIR CMAKE_BINARY_DIR、CMAKE_SOURCE_DIR	工程目标文件目录 工程源文件目录
2	CMAKE_CURRENT_BINARY_DIR CMAKE_CURRENT_SOURCE_DIR	指当前处理的 CMakeLists.txt 所在的路径。
3	CMAKE_CURRENT_LIST_FILE CMAKE_CURRENT_LIST_LINE	输出调用这个变量的 CMakeLists.txt 的路径及行号
4	<project name>_BINARY_DIR <project name>_SOURCE_DIR	project name 工程目标文件 project name 源目标文件
5	EXECUTABLE_OUTPUT_PATH	最终目标二进制文件存放目录
6	LIBRARY_OUT_PATH	最终目标库文件存放目录
7	CMAKE_INSTALL_PREFIX	目标文件安装目录，默认目录为 /usr/local/bin
8	CMAKE_MODULE_PATH	定义自己的 CMake 模块所在路径

9	PROJECT_NAME	返回通过 PROJECT 指令定义的值
10	CMAKE_INCLUDE_CURRENT_DIR	自动添加 CMAKE_CURRENT_BINARY_DIR 和 CMAKE_CURRENT_SOURCE_DIR 添加到当前 CMakeLists.txt 处理。
11	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前面
12	CMAKE_MAJOR_VERSION CMAKE_MINOR_VERSION CMAKE_PATCH_VERSION	CMake 主版本号，2.4.6 中的 2 CMake 次版本号，2.4.6 中的 4 CMake 的补丁等级，2.4.6 中的 6
13	CMAKE_SYSTEM CMAKE_SYSTEM_NAME CMAKE_SYSTEM_VERSION CMAKE_SYSTEM_PROCESSOR	系统名称，如 Linux-2.6.26 Linux 2.6.26 I386
14	UNIX WIN32	在所有的类 UNIX 平台值为 TRUE，包括 MacOS 和 Cygwin 在所有的 WIN32 平台值为 TRUE,包括 Cygwin
15	CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS	开关选项，用来控制 if else 的书写方式
16	BUILD_SHARED_LIBS	开关，默认为静态库
17	CMAKE_C_FLAGS CMAKE_CXX_FLAGS	设置 C 编译选项 设置 C++编译选项
CMake 指令([]表示可选参数)		
序号	语句	注释
1	PROJECT(project name[CXX][C][Java])	定义工程名称(工程名与生成的目标文件名称是没有任何关系的)。此条指令隐含了两个变量 <project name>_BINARY_DIR <project name>_SOURCE_DIR
2	SET(var [value] [cache type docstring[force]])	自定义变量指令 set(SRC_LIST main.cpp hello.cpp) set(SRC_LIST “main.cpp” “test.cpp”)
3	MASSEGE([SEND_ERROR|STATUS|FATAL_ERROR] “massage to display” …)	SEND_ERROR:产生错误，生成过程被跳过 STATUS:输出前缀为---的信息 FATAL_ERROR:立即终止所有 CMake 过程
4	ADD_EXECUTABLE(target source_file…)	增加可执行目标文件， target 由 source_file 生成
5	ADD_SUBDIRECTORY(source_dir [binary_dir] [EXCLUDE FROM ALL])	增加子目录

6	SUBDIRS(dir1 dir2 …)	一次添加多个目录,即使外部编译，子目录体系仍然会被保存
7	INSTALL(TARGETS targets [[ARCHIVE|LIBRARY|RUNTIME] [DESTINATION <dir>] [PERMISSIONS permissions…] [CONFIGGURATIONS [Debug|Release|…]] [COMPONENT <component>] [OPTIONAL]][…])	安装目标文件 ARCHIVE 静态库文件 LIBRARY 动态库文件 RUNTIME 可执行文件 DESTINATION 定义安装路径，如果是绝对路径则覆盖了 CMAKE_INSTALL_PREFIX,否则是指相对 CMAKE_INSTALL_PREFIX 的相对路径
8	INSTALL(FILES files [[DESTINATION <dir>] [PERMISSIONS permissions…] [CONFIGGURATIONS [Debug|Release|…]] [COMPONENT <component>] [OPTIONAL]][…])	安装普通文件 可以指定权限，如果不指定，则默认是 644 权限
9	INSTALL(DIRECTORY dirs [[DESTINATION <dir>] [FILE_PERMISSIONS permissions…] [DIRECTORY_PERMISSIONS permissions…] [USE_SOURCE_PERMISSIONS permissions…] [CONFIGGURATIONS [Debug|Release|…]] [COMPONENT <component>] [[PATTERN <pattern> | REGEX <regex>] [EXCLUDE] [PERMISSIONS permissions…]] […])	安装目录 可以指定权限，如果不指定，则默认是 644 权限
10	ADD_LIBRARY(libname [SHARED|STATIC| MODULE] [EXCLUDE_FROM_ALL] source1 source2 … sourceN)	MODULE:在使用 dydl 的系统有效，如果支持 dydl,则默认为 SHARED
11	SET_TARGET_PROPERTIES(target1 target2 … PROPERTIES prop1 value prop2 value2 …)	设置目标输出的名字及属性 由于 TARGET 名字不能有重复，所以需在生成库文件再改为需要的名字，这时就要用到这个指令了。 相关变量： OUTPUT_NAME,输出名字(库，可执行文件名字，可以不用加后缀 OUTPUT_VALUE CLEAN_DIRECT_OUTPUT VERSION SOVERSION
12	GET_TARGET_PROPERTIES(VAR target property)	获取目标的属性
13	$ENV{NAME}	调用系统环境变量

14	SET(ENV{变量名}值)	设置环境变量值
15	ADD_DEFINITIONS 例:ADD_DEFINITIONS(-DENABLE_DEBUG)	向编译器添加-D 定义
16	ADD_DEPENDENCIES(target_name depend_target1 depend_target)	定义 target 依赖的其他 target
17	ADD_TEST(testname program arg1 arg2)	在打开了 ENABEL_TESTING 后有效
18	ENABEL_TESTING	不带任何参数，控制 Makefile 是否构建 test 目标，一般用在工程主 CMakeList.txt 中
19	AUX_SOURCE_DIRECTORY(dir VARIABLE) 例 AUX_SOURCE_DIRECTORY(. SRC_LIST),将当前目录下源文件名赋给变量 SRC_LIST	自动构建源文件列表
20	CMAKE_MINIMUM_REQUIRED(VERSION version_num [FATAL_ERROR])	检查 CMake 版本，若不满足，产生错误提示或退出
21	EXEC_PROGRAM(program [ARGS args] [OUTPUT_VARIABLE var] [RETURN_VALUE value])	ARGS 用于添加参数 OUTPUT_VARIABLE 用于获取命令输出 RETURN_VALUE 用于获取返回值
22	FILE 指令 FILE(WRITE filename “message” …) FILE(APPEND filename “message” …) FILE(READ filename variable) FILE(GLOB variable [RELATIVE path] [globing expressions]…) FILE(GLOB_RECURSE variable [RELATIVE path] [globing 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)	 写文件 添加内容到文件 读文件 移除目录 递归移除目录 创建目录
23	INCLUDE(file [OPTIONAL]) INCLUDE(module [OPTIONAL])	用来载入 CMakeLists.txt 文件或者 CMake 模块
24	FIND 指令 FIND_FILE(<VAR>name1 path1 path2 …) FIND_LIBRARY(<VAR>name1 path1 path2 …) FIND_PATH(<VAR>name1 path1 path2 …) FIND_PROGRAM(<VAR>name1 path1 path2 …) FIND_PACKAGE(<name> [major.minor] [QUITE] [NO_MODULE] [[REQUIRED|COMPONENTS] [components…]])	 VAR 变量 name1 代表找到的文件全路径，包含文件名 VAR 变量 name2 代表找到的文件全路径，包含库文件名 VAR 变量代表包含这个文件的路径 VAR 变量代表包含这个程序的全路径

5. 判断语句

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)

6. 循环语句

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。

7. 模块的使用和编写

其实使用纯粹依靠 cmake 本身提供的基本指令来管理工程是一件非常复杂的事情，所以，cmake 设计成了可扩展的架构，可以通过编写一些通用的模块来扩展 cmake.

在本章，我们准备首先介绍一下 cmake 提供的 FindCURL 模块的使用。然后，基于我们前面的 libhello 共享库，编写一个 FindHello.cmake 模块。

一，使用 FindCURL 模块

在/backup/cmake 目录建立 t5 目录，用于存放我们的 CURL 的例子。

建立 src 目录，并建立 src/main.c，内容如下：

```
#include <curl/curl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
FILE *fp;
int write_data(void *ptr, size_t size, size_t nmemb, void *stream)
{
    int written = fwrite(ptr, size, nmemb, (FILE *)fp);
    return written;
}
int main()
{
        const char * path = "/tmp/curl-test";
        const char * mode = "w";
        fp = fopen(path,mode);
       curl_global_init(CURL_GLOBAL_ALL);
       CURLcode res;
       CURL *curl = curl_easy_init();
        curl_easy_setopt(curl, CURLOPT_URL, "http://www.linux-ren.org");
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, write_data);
        curl_easy_setopt(curl, CURLOPT_VERBOSE, 1);
      res = curl_easy_perform(curl);
      curl_easy_cleanup(curl);
}
```

这段代码的作用是通过 curl 取回 www.linux-ren.org 的首页并写入/tmp/curl-test 文件中。

建立主工程文件 CMakeLists.txt

```
PROJECT(CURLTEST)
ADD_SUBDIRECTORY(src)
```

建立 src/CMakeLists.txt

```
ADD_EXECUTABLE(curltest main.c)
```

现在自然是没办法编译的，我们需要添加 curl 的头文件路径和库文件。

方法 1：

直接通过 INCLUDE_DIRECTORIES 和 TARGET_LINK_LIBRARIES 指令添加：

我们可以直接在 src/CMakeLists.txt 中添加：

```
INCLUDE_DIRECTORIES(/usr/include)
TARGET_LINK_LIBRARIES(curltest curl)
```

然后建立 build 目录进行外部构建即可。

现在我们要探讨的是使用 cmake 提供的 FindCURL 模块。
方法 2，使用 FindCURL 模块。
向 src/CMakeLists.txt 中添加：

```
FIND_PACKAGE(CURL)
IF(CURL_FOUND)
INCLUDE_DIRECTORIES(${CURL_INCLUDE_DIR})
TARGET_LINK_LIBRARIES(curltest ${CURL_LIBRARY})
ELSE(CURL_FOUND)
MESSAGE(FATAL_ERROR "CURL library not found")
ENDIF(CURL_FOUND)
```

对于系统预定义的 Find<name>.cmake 模块，使用方法一般如上例所示：
每一个模块都会定义以下几个变量

- <name>_FOUND
- <name>_INCLUDE_DIR or <name>_INCLUDES
- <name>_LIBRARY or <name>_LIBRARIES

你可以通过<name>_FOUND 来判断模块是否被找到，如果没有找到，按照工程的需要关闭某些特性、给出提醒或者中止编译，上面的例子就是报出致命错误并终止构建。
如果<name>_FOUND 为真，则将<name>_INCLUDE_DIR 加入 INCLUDE_DIRECTORIES，将<name>_LIBRARY 加入 TARGET_LINK_LIBRARIES 中。
我们再来看一个复杂的例子，通过<name>_FOUND 来控制工程特性：

```
SET(mySources viewer.c)
SET(optionalSources)
SET(optionalLibs)
FIND_PACKAGE(JPEG)
IF(JPEG_FOUND)
SET(optionalSources ${optionalSources} jpegview.c)
INCLUDE_DIRECTORIES( ${JPEG_INCLUDE_DIR} )
SET(optionalLibs ${optionalLibs} ${JPEG_LIBRARIES} )
ADD_DEFINITIONS(-DENABLE_JPEG_SUPPORT)
ENDIF(JPEG_FOUND)
IF(PNG_FOUND)
SET(optionalSources ${optionalSources} pngview.c)
INCLUDE_DIRECTORIES( ${PNG_INCLUDE_DIR} )
SET(optionalLibs ${optionalLibs} ${PNG_LIBRARIES} )
ADD_DEFINITIONS(-DENABLE_PNG_SUPPORT)
ENDIF(PNG_FOUND)
ADD_EXECUTABLE(viewer ${mySources} ${optionalSources} )
TARGET_LINK_LIBRARIES(viewer ${optionalLibs}
```

通过判断系统是否提供了 JPEG 库来决定程序是否支持 JPEG 功能。

二，编写属于自己的 FindHello 模块。

我们在此前的 t3 实例中，演示了构建动态库、静态库的过程并进行了安装。
接下来，我们在 t6 示例中演示如何自定义 FindHELLO 模块并使用这个模块构建工程：
请在建立/backup/cmake/中建立 t6 目录，并在其中建立 cmake 目录用于存放我们自己定义的 FindHELLO.cmake 模块，同时建立 src 目录，用于存放我们的源文件。
1，定义 cmake/FindHELLO.cmake 模块

```
FIND_PATH(HELLO_INCLUDE_DIR hello.h /usr/include/hello
```

```
/usr/local/include/hello)
FIND_LIBRARY(HELLO_LIBRARY NAMES hello PATH /usr/lib
/usr/local/lib)
IF (HELLO_INCLUDE_DIR AND HELLO_LIBRARY)
SET(HELLO_FOUND TRUE)
ENDIF (HELLO_INCLUDE_DIR AND HELLO_LIBRARY)
IF (HELLO_FOUND)
IF (NOT HELLO_FIND_QUIETLY)
MESSAGE(STATUS "Found Hello: ${HELLO_LIBRARY}")
ENDIF (NOT HELLO_FIND_QUIETLY)
ELSE (HELLO_FOUND)
IF (HELLO_FIND_REQUIRED)
MESSAGE(FATAL_ERROR "Could not find hello library")
ENDIF (HELLO_FIND_REQUIRED)
ENDIF (HELLO_FOUND)
```

针对上面的模块让我们再来回顾一下 FIND_PACKAGE 指令：

```
FIND_PACKAGE(<name> [major.minor] [QUIET] [NO_MODULE]
[[REQUIRED|COMPONENTS] [componets...]])
```

前面的 CURL 例子中我们使用了最简单的 FIND_PACKAGE 指令，其实他可以使用多种参数，QUIET 参数，对应与我们编写的 FindHELLO 中的 HELLO_FIND_QUIETLY，如果不指定这个参数，就会执行：

```
MESSAGE(STATUS "Found Hello: ${HELLO_LIBRARY}")
```

REQUIRED 参数，其含义是指这个共享库是否是工程必须的，如果使用了这个参数，说明这个链接库是必备库，如果找不到这个链接库，则工程不能编译。对应于 FindHELLO.cmake 模块中的 HELLO_FIND_REQUIRED 变量。

同样，我们在上面的模块中定义了 HELLO_FOUND, HELLO_INCLUDE_DIR,HELLO_LIBRARY 变量供开发者在 FIND_PACKAGE 指令中使用。

OK，下面建立 src/main.c，内容为：

```
#include <hello.h>
int main()
{
      HelloFunc();
     return 0;
}
```

建立 src/CMakeLists.txt 文件，内容如下：

```
FIND_PACKAGE(HELLO)
IF(HELLO_FOUND)
ADD_EXECUTABLE(hello main.c)
INCLUDE_DIRECTORIES(${HELLO_INCLUDE_DIR})
TARGET_LINK_LIBRARIES(hello ${HELLO_LIBRARY})
ENDIF(HELLO_FOUND)
```

为了能够让工程找到 FindHELLO.cmake 模块(存放在工程中的 cmake 目录)我们在主工程文件 CMakeLists.txt 中加入：

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

三，使用自定义的 FindHELLO 模块构建工程

仍然采用外部编译的方式，建立 build 目录，进入目录运行：

```
cmake ..
```

我们可以从输出中看到：
Found Hello: /usr/lib/libhello.so
如果我们把上面的 FIND_PACKAGE(HELLO)修改为 FIND_PACKAGE(HELLO QUIET),则不会看到上面的输出。
接下来就可以使用 make 命令构建工程，运行:
./src/hello 可以得到输出
Hello World。
说明工程成功构建。

四，如果没有找到 hello library 呢？

我们可以尝试将/usr/lib/libhello.x 移动到/tmp 目录，这样，按照 FindHELLO 模块的定义，就找不到 hello library 了，我们再来看一下构建结果：
cmake ..
仍然可以成功进行构建，但是这时候是没有办法编译的。
修改 FIND_PACKAGE(HELLO)为 FIND_PACKAGE(HELLO REQUIRED)，将 hello library 定义为工程必须的共享库。
这时候再次运行 cmake ..
我们得到如下输出：
CMake Error: Could not find hello library.
因为找不到 libhello.x，所以，整个 Makefile 生成过程被出错中止。

小结：

在本节中，我们学习了如何使用系统提供的 Find<NAME>模块并学习了自己编写 Find<NAME>模块以及如何在工程中使用这些模块。

后面的章节，我们会逐渐学习更多的 cmake 模块使用方法以及用 cmake 来管理 GTK 和 QT4 工程。

cmake 中如何指定非系统路径中的 qt 版本

在 cmake 命令行使用参数
-DQT_QMAKE_EXECUTABLE:FILEPATH=/path/to/your/qt/bin/qmake
如： cmake -DQT_QMAKE_EXECUTABLE:FILEPATH=/path/to/your/qt/bin/qmake .
将/path/to/your/qt/bin/qmake 换成不同的 qt 版本，可以生成依赖不同 qt 版本的 Makefile 文件。