<https://www.cnblogs.com/alphagl/p/6280061.html>

### CMake学习

本篇分享一下有关CMake的一些学习心得以及相关使用。

**作者：**[**AlphaGL**](http://www.cnblogs.com/alphagl/)**。版权所有，欢迎保留原文链接进行转载 :)**

本文目录如下：

* [1、CMake介绍](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-intro)
* [2、CMake安装与使用](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-install-use)
  + [2.1 CMake的安装](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-install)
  + [2.2 CMake-gui的使用](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-gui-use)
  + [2.3 CMake命令行的使用](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-command-use)
  + [2.4 CMake命令行说明](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-command-dis)
* [3、CMake语法及CMakeLists.txt编写](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-grammer-w)
  + [3.1 CMake语法](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-grammer)
    - [3.1.1 常用的命令](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-command)
    - [3.1.2 常用的变量](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-var)
  + [3.2 CMakeLists.txt编写](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-w)
    - [3.2.1 将cJSON构建为静态库](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-static)
    - [3.2.1 将cJSON外部依赖库链接进可执行文件中](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-extra)
    - [3.2.2 将cJSON库改为可选](https://www.cnblogs.com/alphagl/p/6280061.html#cmake-option)

### 1、CMake介绍

CMake全称为“cross platform make”，是一个开源的跨平台自动化构建系统。使用指定名为CMakeLists.txt的配置文件可以控制软件的构建、测试和打包等流程。同时，通过编写平台无关的CMakeLists.txt文件和需要简单的配置，CMake就能生成对应目标平台的构建文件，例如：类Unix系统的makefile文件、Windows的Visual Studio工程或者Mac的Xcode工程，大大简化了跨平台和交叉编译方面的工作。

当然，类似的make工具也很多，Autocof、JAM、QMake、SCons甚至ANT，目的都是指定一套规则来简化整个构建编译流程。CMake工具链简单、灵活，且跨平台，很多知名项目都在使用CMake构建。适合以C、C++或者java等编译语言的项目。

### 2、CMake安装与使用

##### 2.1 CMake的安装

CMake下载地址为：  
<https://cmake.org/download/>

##### 2.2 CMake-gui的使用

CMake的安装过程很简单，这里就不再赘述了。其中CMake提供带GUI界面的工具cmake-gui。本文主要是以命令行的形式来讲述CMake的使用，这里就稍微提一下cmake-gui的使用。原理都是差不多的。

简述操作步骤如下：  
1、在本地目录新建cmake文件夹(例如：我的目录为E:\workplace\cmake)，并在cmake目录中创建bin目录用于存放构建过程中产生的临时文件和目标文件(例如：我的目录为E:\workplace\cmake\bin)。

2、在cmake目录中新建hello\_cmake.c与CMakeLists.txt文件，内容如下：  
**hello\_cmake.c**：

#include <stdio.h>

int main() {

printf("Hello CMake\n");

return 0;

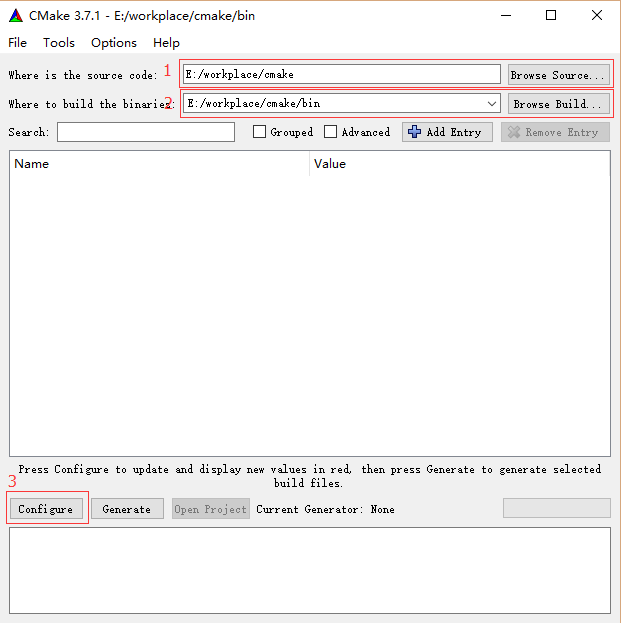
}

**CMakeLists.txt**：

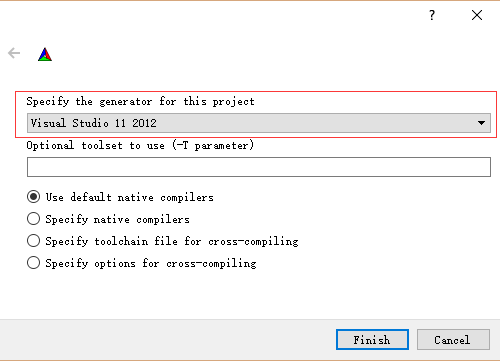
cmake\_minimum\_required (VERSION 2.8)

project (HelloCMake)

add\_executable(HelloCMake hello\_cmake.c)

3、运行cmake-gui，会打开如下界面：  


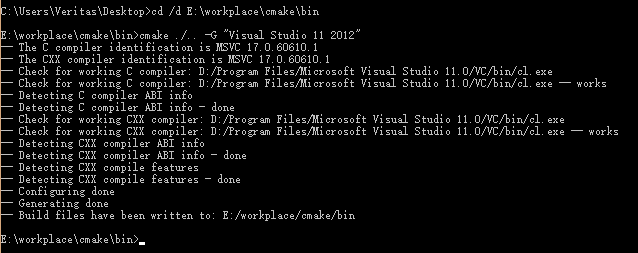
(1)源码所在的路径，即CMakeLists.txt所在的路径。本例为E:\workplace\cmake  
(2)构建过程中产生的文件的路径。本例为：E:\workplace\cmake\bin，注意这个路径可以与(1)路径相同，但若项目很大产生的文件很多要执行清理等操作时会比较麻烦。  
(3)执行configure。

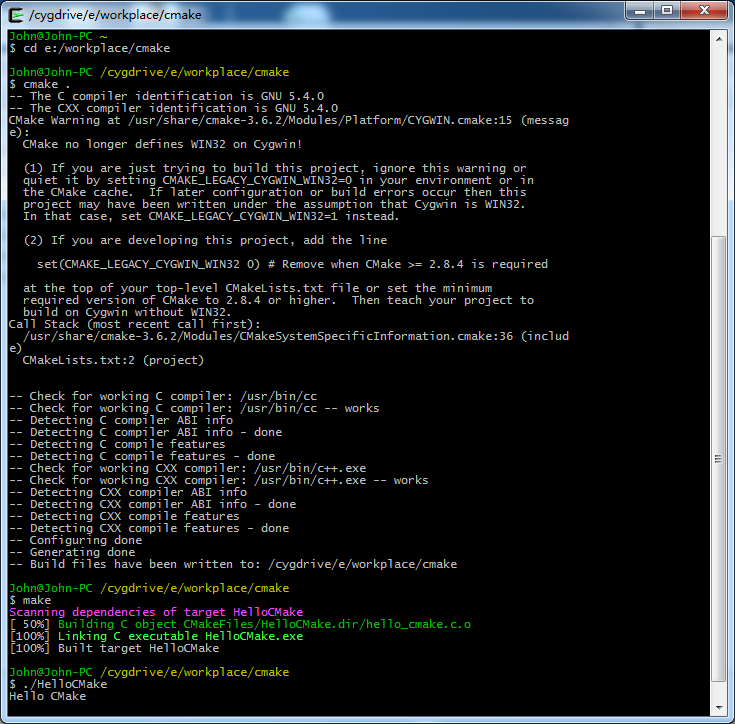
  
这里指定生成的构建工程为VS2012的项目工程。点击Finish即可自动编译，并生成对应的\*.sln的项目文件。

在Visual Studio中打开该项目，并执行。可以看到正确输出了"Hello CMake"。

##### 2.3 CMake命令行的使用

这里，依然使用刚才的示例来说明下CMake命令行的方式的使用(清理bin目录下的文件即可)，使用步骤如下：  
(1)将CMake安装路径添加到Path环境变量中。Mac版本可以终端中输入命令：export PATH=/Applications/CMake.app/Contents/bin:$PATH即可。

(2)在Windows命令行中执行如下命令：  
  
在当前构建的路径，执行cmake，cmake后面参数为CMakeLists.txt所在路径，-G用来指定构建生成目标平台的项目工程。分别对应上面的GUI的操作。

(3)在类Unix的操作系统中，可以执行如下操作。这里我还是用的上面的示例代码，我用Cygwin模拟下对应的cmake操作，如图：  
  
在Cygwin中新版的CMake会报警告(在32位Windows系统上则会报错)。

由此可见，在Unix系列的操作系统上，使用CMake步骤分为：

1. 编写CMakeLists.txt配置文件。
2. 执行cmake命令生成makefile文件。
3. 执行make命令编译执行，生成最终的目标文件。
4. 运行最终生成的可执行文件即可，这步可选。

##### 2.4 CMake命令行说明

CMake命令行的选项可以在命令行终端上，输入cmake --help查看。更详尽的解释可以查看CMake的官方手册。

[*cmake帮助手册*](https://cmake.org/cmake/help/v3.7/manual/cmake.1.html)

CMake命令行格式为：

(1) cmake [<options>] (<path-to-source> | <path-to-existing-build>)

(2) cmake [(-D <var>=<value>)...] -P <cmake-script-file>

(3) cmake --build <dir> [<options>...] [-- <build-tool-options>...]

(4) cmake -E <command> [<options>...]

(5) cmake --find-package <options>...

常见的使用方式是第一种。这里也主要介绍这种方式。  
(1)[<options>]，表示option为可选的。path-to-source和path-to-existing-build二选一，分别表示一个CMakeLists.txt所在的路径和一个已存在的构建工程所在的路径。例如：

cmake .

这里option为空，构建的路径为用户所在的当前路径。

其中option常用的有：  
**-G <generator-name>**：  
指定构建系统生成器，当前平台所支持的generator-name可以通过帮助手册查看。例如： cmake -G "Visual Studio 11 2012" E:\workplace\cmake，生成VS2012的构建工程。

**-D <var>:<type>=<value>, -D <var>=<value>**：  
添加变量及值到CMakeCache.txt中。例如：cmake -D EXECUTABLE\_OUTPUT\_PATH="bin" .，会在CMakeCache.txt中添加一条

//No help, variable specified on the command line.

EXECUTABLE\_OUTPUT\_PATH:UNINITIALIZED=bin

这样，可以在CMakeLists.txt文件中读取该变量的值。例如：message(${EXECUTABLE\_OUTPUT\_PATH})

**-U <globbing\_expr>**：  
此选项可用于从CMakeCache.txt文件中删除一个或多个变量，支持使用\*和？的匹配。与-D对应，使用是须谨慎，可能会导致CMakeCache.txt不工作。

**-i**：  
以向导的方式运行CMake。此选项会弹出一系列的提示，要求用户回答关于工程配置的一些问题。这些结果会被用来设置cmake的缓存值。注意，新版的CMake可能不再支持此选项。

**-E**：  
CMake命令行模式。CMake提供了一系列与平台无关的命令。例如：copy，make\_directory，echo等，更多详细参见cmake -E help。

### 3、CMake语法及CMakeLists.txt编写

用CMake构建一个项目工程，是通过一个或多个CMakeLists.txt文件来控制的。CMakeLists.txt中包含一系列命令来描述需要执行的构建。

##### 3.1 CMake语法

在CMakeLists.txt中的命令的语法，都是形如下面这种格式：

command (args...)

command：是命令的名字。  
args：是参数的列表。多个参数使用空格隔开。

##### 3.1.1 常用的命令

**cmake\_minimum\_required**：  
设置项目要求的CMake最低版本号，如果当前版本的CMake低于所需的值，它将停止处理项目并报告错误。注意在project之前调用该命令，一般在CMakeLists.txt文件开头调用。命令格式为：

cmake\_minimum\_required(VERSION major.minor[.patch[.tweak]]

[FATAL\_ERROR])

使用示例：

cmake\_minimum\_required(VERSION 2.8.5)

**add\_custom\_command**：  
该命令可以为生成的构建系统添加一条自定义的构建规则。这里又包含两种使用方式，一种是通过自定义命令在构建中生成输出文件，另外一种是向构建目标添加自定义命令。命令格式分别为：  
(1)生成文件

add\_custom\_command(OUTPUT output1 [output2 ...]

COMMAND command1 [ARGS] [args1...]

[COMMAND command2 [ARGS] [args2...] ...]

[MAIN\_DEPENDENCY depend]

[DEPENDS [depends...]]

[BYPRODUCTS [files...]]

[IMPLICIT\_DEPENDS <lang1> depend1

[<lang2> depend2] ...]

[WORKING\_DIRECTORY dir]

[COMMENT comment]

[DEPFILE depfile]

[VERBATIM] [APPEND] [USES\_TERMINAL])

参数介绍：  
OUTPUT：  
指定命令预期产生的输出文件。如果输出文件的名称是相对路径，即相对于当前的构建的源目录路径。输出文件可以指定多个output1,output2(可选)等。

COMMAND：  
指定要在构建时执行的命令行。如果指定多个COMMAND，它们讲按顺心执行。ARGS参数是为了向后兼容，为可选参数。args1和args2为参数，多个参数用空格隔开。

MAIN\_DEPENDENCY：  
可选命令，指定命令的主要输入源文件。

DEPENDS：  
指定命令所依赖的文件。

BYPRODUCTS：  
可选命令，指定命令预期产生的文件，但其修改时间可能会比依赖性更新，也可能不会更新。

IMPLICIT\_DEPENDS：  
可选命令，请求扫描输入文件的隐式依赖关系。给定的语言指定应使用相应的依赖性扫描器的编程语言。目前只支持C和CXX语言扫描器。必须为IMPLICIT\_DEPENDS列表中的每个文件指定语言。从扫描中发现的依赖关系在构建时添加到自定义命令的依赖关系。请注意，IMPLICIT\_DEPENDS选项目前仅支持Makefile生成器，并且将被其他生成器忽略。

WORKING\_DIRECTORY：  
可选命令，使用给定的当前工作目录执行命令。如果它是相对路径，它将相对于对应于当前源目录的构建树目录。

COMMENT：  
可选命令，在构建时执行命令之前显示给定消息。

DEPFILE：  
可选命令，为Ninja生成器指定一个.d depfile。 .d文件保存通常由自定义命令本身发出的依赖关系。对其他生成器使用DEPFILE是一个错误。

使用实例：

add\_executable(MakeTable MakeTable.cxx)

add\_custom\_command (

OUTPUT ${CMAKE\_CURRENT\_BINARY\_DIR}/Table.h

COMMAND MakeTable ${CMAKE\_CURRENT\_BINARY\_DIR}/Table.h

DEPENDS MakeTable

COMMENT "This is a test"

)

(2)自定义构建事件

add\_custom\_command(TARGET <target>

PRE\_BUILD | PRE\_LINK | POST\_BUILD

COMMAND command1 [ARGS] [args1...]

[COMMAND command2 [ARGS] [args2...] ...]

[BYPRODUCTS [files...]]

[WORKING\_DIRECTORY dir]

[COMMENT comment]

[VERBATIM] [USES\_TERMINAL])

参数介绍：  
TARGET：  
定义了与构建指定相关联的新命令。当已经存在是，相应的command将不再执行。

PRE\_BUILD：  
在目标中执行任何其他规则之前运行。这仅在Visual Studio 7或更高版本上受支持。对于所有其他生成器PRE\_BUILD将被视为PRE\_LINK。

PRE\_LINK：  
在编译源之后运行，但在链接二进制文件或运行静态库的库管理器或存档器工具之前运行。

POST\_BUILD：  
在目标中的所有其他规则都已执行后运行。

使用实例：

add\_custom\_command(TARGET ${APP\_NAME}

PRE\_BUILD

COMMAND ${CMAKE\_COMMAND} -E copy\_directory ${CMAKE\_CURRENT\_SOURCE\_DIR}/Resources ${CMAKE\_CURRENT\_BINARY\_DIR})

**add\_custom\_target**：  
该命令可以给指定名称的目标执行指定的命令，该目标没有输出文件，并始终被构建。命令的格式为：

add\_custom\_target(Name [ALL] [command1 [args1...]]

[COMMAND command2 [args2...] ...]

[DEPENDS depend depend depend ... ]

[BYPRODUCTS [files...]]

[WORKING\_DIRECTORY dir]

[COMMENT comment]

[VERBATIM] [USES\_TERMINAL]

[SOURCES src1 [src2...]])

参数介绍(上面介绍过含义相同的参数，这里就不再赘述了)：  
Name：  
指定目标的名称。

ALL：  
表明此目标应添加到默认构建目标，以便每次都将运行（该命令名称不能为ALL）

SOURCES：  
指定要包括在自定义目标中的其他源文件。指定的源文件将被添加到IDE项目文件中，以方便编辑，即使它们没有构建规则。

使用示例：

add\_custom\_target(APP ALL

DEPENDS ${APP\_NAME} # 依赖add\_custom\_command输出的jar包

COMMENT "building cassdk\_jni.jar"

)

**add\_definitions**：  
为源文件的编译添加由-D引入的宏定义。命令格式为：

add\_definitions(-DFOO -DBAR ...)

使用示例：

add\_definitions(-DWIN32)

**add\_dependencies**：  
使顶级目标依赖于其他顶级目标，以确保它们在该目标之前构建。这里的顶级目标是由add\_executable，add\_library或add\_custom\_target命令之一创建的目标。  
使用示例：

add\_custom\_target(mylib DEPENDS ${MYLIB})

add\_executable(${APP\_NAME} ${SRC\_LIST})

add\_dependencies(${APP\_NAME} mylib)

**add\_executable**：  
使用指定的源文件给项目添加一个可执行文件。命令格式为：

add\_executable(<name> [WIN32] [MACOSX\_BUNDLE]

[EXCLUDE\_FROM\_ALL]

source1 [source2 ...])

参数介绍：  
name：  
该命令调用列出的源文件来构建的可执行目标。 对应于逻辑目标名称，在项目中必须是全局唯一的。构建的可执行文件的实际文件名是基于本机平台的约定。

WIN32：  
如果给出WIN32，则在创建的目标上设置属性WIN32\_EXECUTABLE。

MACOSX\_BUNDLE:  
如果给定MACOSX\_BUNDLE，将在创建的目标上设置相应的属性。

EXCLUDE\_FROM\_ALL：  
如果给定EXCLUDE\_FROM\_ALL，将在创建的目标上设置相应的属性。

source：  
源码列表。

使用示例：

add\_executable(HelloCMake hello\_cmake.c)

**add\_library**：  
使用指定的源文件给项目添加一个库。命令格式为：

add\_library(<name> [STATIC | SHARED | MODULE]

[EXCLUDE\_FROM\_ALL]

source1 [source2 ...])

参数介绍：  
name：  
该命令调用列出的源文件来构建的库目标。对应于逻辑目标名称，在项目中必须是全局唯一的。

STATIC：  
静态库，在链接其他目标时使用。

SHARED：  
动态链接库，运行时加载。

MODULE：  
不会被链接到其它目标中，但是可能会在运行时使用dlopen-系列的函数动态链接。

使用示例：

add\_library(HelloCMake hello\_cmake.c)

**add\_subdirectory**:  
向构建中添加子目录。命令格式为：

add\_subdirectory(source\_dir [binary\_dir]

[EXCLUDE\_FROM\_ALL])

使用示例：

add\_subdirectory(${SRC\_ROOT})

**aux\_source\_directory**：  
查找目录中的所有源文件。命令格式为：

aux\_source\_directory(<dir> <variable>)

查找指定目录dir中所有源文件的名称，并将列表存储在提供的variable中。

使用示例：

aux\_source\_directory(. DIR\_SRCS)

add\_executable(${APP\_NAME} ${DIR\_SRCS})

**configure\_file**：  
将文件复制到其他位置并修改其内容。命令格式为：

configure\_file(<input> <output>

[COPYONLY] [ESCAPE\_QUOTES] [@ONLY]

[NEWLINE\_STYLE [UNIX|DOS|WIN32|LF|CRLF] ])

使用示例：

configure\_file (

"${PROJECT\_SOURCE\_DIR}/Config.h.in"

"${PROJECT\_BINARY\_DIR}/Config.h"

)

**file**：  
文件操作相关的命令。命令格式为：

file(WRITE <filename> <content>...)

file(APPEND <filename> <content>...)

file(READ <filename> <variable>

[OFFSET <offset>] [LIMIT <max-in>] [HEX])

file(STRINGS <filename> <variable> [<options>...])

file(<MD5|SHA1|SHA224|SHA256|SHA384|SHA512> <filename> <variable>)

file(GLOB <variable>

[LIST\_DIRECTORIES true|false] [RELATIVE <path>]

[<globbing-expressions>...])

file(GLOB\_RECURSE <variable> [FOLLOW\_SYMLINKS]

[LIST\_DIRECTORIES true|false] [RELATIVE <path>]

[<globbing-expressions>...])

file(RENAME <oldname> <newname>)

file(REMOVE [<files>...])

file(REMOVE\_RECURSE [<files>...])

file(MAKE\_DIRECTORY [<directories>...])

file(RELATIVE\_PATH <variable> <directory> <file>)

file(TO\_CMAKE\_PATH "<path>" <variable>)

file(TO\_NATIVE\_PATH "<path>" <variable>)

file(DOWNLOAD <url> <file> [<options>...])

file(UPLOAD <file> <url> [<options>...])

file(TIMESTAMP <filename> <variable> [<format>] [UTC])

file(GENERATE OUTPUT output-file

<INPUT input-file|CONTENT content>

[CONDITION expression])

file(<COPY|INSTALL> <files>... DESTINATION <dir>

[FILE\_PERMISSIONS <permissions>...]

[DIRECTORY\_PERMISSIONS <permissions>...]

[NO\_SOURCE\_PERMISSIONS] [USE\_SOURCE\_PERMISSIONS]

[FILES\_MATCHING]

[[PATTERN <pattern> | REGEX <regex>]

[EXCLUDE] [PERMISSIONS <permissions>...]] [...])

file(LOCK <path> [DIRECTORY] [RELEASE]

[GUARD <FUNCTION|FILE|PROCESS>]

[RESULT\_VARIABLE <variable>]

[TIMEOUT <seconds>])

以上都是文件相关的操作，这里就不详细解释。  
使用示例为：

# 查找src目录下所有以hello开头的文件并保存到SRC\_FILES变量里

file(GLOB SRC\_FILES "src/hello\*")

# 递归查找

file(GLOB\_RECURSE SRC\_FILES "src/hello\*")

**find\_file**：  
查找一个文件的完整路径。命令格式为：

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

使用示例：

find\_file(HELLO\_H hello.h)

**find\_library**：  
查找一个库文件。命令格式为：

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

使用示例：

find\_library(LUA lua5.1 /usr/lib /lib)

**find\_package**：  
查找并加载外部项目的设置。命令格式为：

find\_package(<package> [version] [EXACT] [QUIET] [MODULE]

[REQUIRED] [[COMPONENTS] [components...]]

[OPTIONAL\_COMPONENTS components...]

[NO\_POLICY\_SCOPE])

使用示例为：

find\_package(Protobuf)

**find\_path**：  
查找包含某个文件的路径。命令格式为：

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

使用示例：

find\_path(DIR\_SRCS hello.h .)

**include\_directories**：  
将给定的目录添加到编译器用于搜索包含文件的目录。相对路径则相对于当前源目录。命令格式为：

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

使用示例：

include\_directories(

${CMAKE\_CURRENT\_SOURCE\_DIR}

${CMAKE\_CURRENT\_SOURCE\_DIR}/cocos

${CMAKE\_CURRENT\_SOURCE\_DIR}/cocos/platform

${CMAKE\_CURRENT\_SOURCE\_DIR}/extensions

${CMAKE\_CURRENT\_SOURCE\_DIR}/external

)

**include**：  
包含其他目录的CMakeLists.txt文件。命令格式为：

include(<file|module> [OPTIONAL] [RESULT\_VARIABLE <VAR>]

[NO\_POLICY\_SCOPE])

使用示例：

include(platform/CMakeLists.txt)

**link\_directories**：  
指定链接器查找库的路径。命令格式为：

link\_directories(directory1 directory2 ...)

使用示例：

link\_directories(${PROJECT\_SOURCE\_DIR}/lib)

**list**：  
列表相关的操作。命令格式为：

list(LENGTH <list> <output variable>)

list(GET <list> <element index> [<element index> ...]

<output variable>)

list(APPEND <list> [<element> ...])

list(FILTER <list> <INCLUDE|EXCLUDE> REGEX <regular\_expression>)

list(FIND <list> <value> <output variable>)

list(INSERT <list> <element\_index> <element> [<element> ...])

list(REMOVE\_ITEM <list> <value> [<value> ...])

list(REMOVE\_AT <list> <index> [<index> ...])

list(REMOVE\_DUPLICATES <list>)

list(REVERSE <list>)

list(SORT <list>)

使用示例：

list(APPEND SRC\_LIST

${PROTO\_SRC}

)

**message**：  
向用户显示消息。命令格式为：

message([<mode>] "message to display" ...)

参数说明：  
mode：  
可选的值为none，STATUS，WARNING，AUTHOR\_WARNING，SEND\_ERROR，FATAL\_ERROR，DEPRECATION。  
使用示例：

message(STATUS "This is BINARY dir " ${HELLO\_BINARY\_DIR})

**option**：  
提供用户可以选择的选项。命令格式为：

option(<option\_variable> "help string describing option"

[initial value])

使用示例：

option (USE\_MYMATH "Use tutorial provided math implementation" ON)

**project**：  
为整个工程设置一个工程名。命令格式为：

project(<PROJECT-NAME> [LANGUAGES] [<language-name>...])

project(<PROJECT-NAME>

[VERSION <major>[.<minor>[.<patch>[.<tweak>]]]]

[LANGUAGES <language-name>...])

使用示例：

project (HelloCMake)

**set**：  
将一个CMAKE变量设置为给定值。命令格式为：

set(<variable> <value>... [PARENT\_SCOPE])

使用示例：

set(COCOS2D\_ROOT ${CMAKE\_SOURCE\_DIR}/cocos2d)

**set\_target\_properties**：  
设置目标的一些属性来改变它们构建的方式。命令格式为：

set\_target\_properties(target1 target2 ...

PROPERTIES prop1 value1

prop2 value2 ...)

使用示例为：

set\_target\_properties(cocos2d

PROPERTIES

ARCHIVE\_OUTPUT\_DIRECTORY "${CMAKE\_BINARY\_DIR}/lib"

LIBRARY\_OUTPUT\_DIRECTORY "${CMAKE\_BINARY\_DIR}/lib"

VERSION "${COCOS2D\_X\_VERSION}"

)

**string**：  
字符串相关操作。命令格式为：

string(FIND <string> <substring> <output variable> [REVERSE])

string(REPLACE <match\_string>

<replace\_string> <output variable>

<input> [<input>...])

string(REGEX MATCH <regular\_expression>

<output variable> <input> [<input>...])

string(REGEX MATCHALL <regular\_expression>

<output variable> <input> [<input>...])

string(REGEX REPLACE <regular\_expression>

<replace\_expression> <output variable>

<input> [<input>...])

string(APPEND <string variable> [<input>...])

string(CONCAT <output variable> [<input>...])

string(TOLOWER <string1> <output variable>)

string(TOUPPER <string1> <output variable>)

string(LENGTH <string> <output variable>)

string(SUBSTRING <string> <begin> <length> <output variable>)

string(STRIP <string> <output variable>)

string(GENEX\_STRIP <input string> <output variable>)

string(COMPARE LESS <string1> <string2> <output variable>)

string(COMPARE GREATER <string1> <string2> <output variable>)

string(COMPARE EQUAL <string1> <string2> <output variable>)

string(COMPARE NOTEQUAL <string1> <string2> <output variable>)

string(COMPARE LESS\_EQUAL <string1> <string2> <output variable>)

string(COMPARE GREATER\_EQUAL <string1> <string2> <output variable>)

string(<MD5|SHA1|SHA224|SHA256|SHA384|SHA512>

<output variable> <input>)

string(ASCII <number> [<number> ...] <output variable>)

string(CONFIGURE <string1> <output variable>

[@ONLY] [ESCAPE\_QUOTES])

string(RANDOM [LENGTH <length>] [ALPHABET <alphabet>]

[RANDOM\_SEED <seed>] <output variable>)

string(TIMESTAMP <output variable> [<format string>] [UTC])

string(MAKE\_C\_IDENTIFIER <input string> <output variable>)

string(UUID <output variable> NAMESPACE <namespace> NAME <name>

TYPE <MD5|SHA1> [UPPER])

使用示例：

string(REPLACE "${PROJECT\_SOURCE\_DIR}/hello.c" "" DIR\_SRCS "${DIR\_ROOT}")

**target\_link\_libraries**：  
将给定的库链接到一个目标上。命令格式为：

target\_link\_libraries(<target> ... <item>... ...)

使用示例：

target\_link\_libraries(luacocos2d cocos2d)

##### 3.1.2 常用的变量

使用${}进行变量的引用。例如：message(${Hello\_VERSION})，Hello为工程名。CMake提供了很多有用的变量。以下仅列举常用的变量：

**CMAKE\_BINARY\_DIR**：  
构建树的顶层路径

**CMAKE\_COMMAND**：  
指向CMake可执行文件的完整路径

**CMAKE\_CURRENT\_BINARY\_DIR**：  
当前正在被处理的二进制目录的路径。

**CMAKE\_CURRENT\_SOURCE\_DIR**：  
指向正在被处理的源码目录的路径。

**CMAKE\_HOME\_DIRECTORY**：  
指向源码树顶层的路径。

**CMAKE\_PROJECT\_NAME**：  
当前工程的工程名。

**CMAKE\_ROOT**：  
CMake的安装路径。

**CMAKE\_SOURCE\_DIR**：  
源码树的顶层路径。

**CMAKE\_VERSION**：  
cmake的完整版本号。

**PROJECT\_BINARY\_DIR**：  
指向当前编译工程构建的全路径。

**<PROJECT-NAME>\_BINARY\_DIR**：  
指向当前编译工程构建的全路径。

**<PROJECT-NAME>\_SOURCE\_DIR**：  
指向构建工程的全路径。

**PROJECT\_SOURCE\_DIR**：  
指向构建工程的全路径。

**PROJECT\_NAME**：  
project命令传递的工程名参数。

**<PROJECT-NAME>\_VERSION**：  
项目的完整版本号。

##### 3.2 CMakeLists.txt编写

有了上面的基础，再编写CMakeLists.txt自然会事半功倍。下面，以几个小实例来说下通过CMakeLists.txt的来构建项目。

这里cJSON库为例来说明下CMakeLists.txt的写法。当然，这里的代码并严谨，仅用来演示CMakeList的用法。

##### 3.2.1 将cJSON构建为静态库

(1)在本地建立cJSONdemo1的目录工程，并将cJSON库源代码拷贝到目录中，并在该目录新建CMakeLists.txt文件。目录结构如下：

cJSONdemo1

├── cJSON\_Utils.h

├── cJSON\_Utils.c

├── cJSON.h

├── cJSON.c

└── CMakeLists.txt

CMakeLists.txt文件内容如下：

cmake\_minimum\_required(VERSION 2.8.5)

project(cJSON-lib)

set(CJSON\_SRC cJSON.c cJSON\_Utils.c)

add\_library(cjson STATIC ${CJSON\_SRC})

在终端下执行如下操作：  


(2)自动搜索目录源码  
在上面cJSONdemo1的基础上做一些改进。前面提过set(<variable> <value>...)，可以预见在cJSON库源码越来越多的情况下，会变成这样：

set(CJSON\_SRC cJSON.c cJSON1.c cJSON2.c cJSON3.c cJSON4.c cJSON5.c)

这样，源文件越多，需要添加次数就越多。而且，每增加一个源文件就需要修改CMakeLists.txt文件，“耦合性”太大。这里，可以使用aux\_source\_directory来自动查找源文件。CMakeLists.txt文件最终如下：

cmake\_minimum\_required(VERSION 2.8.5)

project(cJSON-lib)

aux\_source\_directory(. CJSON\_SRC)

add\_library(cjson STATIC ${CJSON\_SRC})

(3)递归搜索目录源码  
若将cJSONdemo改成包含子目录，子目录中又包含源码的形式，有多级目录。如下

cJSONdemo1

│── cJSON\_Utils.h

│── cJSON\_Utils.c

│── cJSON.h

│── cJSON.c

│── CMakeLists.txt

└── foo

├── cJSON1.h

├── cJSON1.c

├── cJSON2.h

├── cJSON2.c

└── goo

├── cJSON3.h

├── cJSON3.c

├── cJSON4.h

└── cJSON4.c

可以使用file命令，来自动递归查找相应的源文件。CMakeLists.txt文件最终如下：

cmake\_minimum\_required(VERSION 2.8.5)

project(cJSON-lib)

file(GLOB\_RECURSE CJSON\_SRC ${CMAKE\_CURRENT\_SOURCE\_DIR}/\*.c)

add\_library(cjson STATIC ${CJSON\_SRC})

(4)指定构建库的名字，路径和前缀。CMakeLists.txt文件最终如下：

cmake\_minimum\_required(VERSION 2.8.5)

project(cJSON-lib)

file(GLOB\_RECURSE CJSON\_SRC ${CMAKE\_CURRENT\_SOURCE\_DIR}/\*.c)

add\_library(cjson STATIC ${CJSON\_SRC})

set\_target\_properties(cjson PROPERTIES OUTPUT\_NAME "json")

set(LIBRARY\_OUTPUT\_PATH ${PROJECT\_BINARY\_DIR}/static)

set(CMAKE\_STATIC\_LIBRARY\_PREFIX "")

最终效果如图：  
  
会生成cJSONdemo1/static/json.a。

##### 3.2.1 将cJSON外部依赖库链接进可执行文件中

通过上面过程了解了将cJSON库构建文件静态库的过程。下面，再添加测试代码来调用cJSON库，并最终构建为可执行文件。目录如下：

cJSONdemo2

│── test.c

│── CMakeLists.txt

└── lib

├── cJScJSON\_UtilsON1.h

├── cJSON\_Utils.c

├── cJSON.h

├── cJSON.c

└── CMakeLists.txt

**test.c**：

#include <stdio.h>

#include <stdlib.h>

#include "lib/cJSON.h"

void parser(char\* text) {

char \*out;

cJSON \*json;

json = cJSON\_Parse(text);

if (!json) {

printf("Error before: [%s]\n", cJSON\_GetErrorPtr());

}else {

out = cJSON\_Print(json);

cJSON\_Delete(json);

printf("%s\n", out);

free(out);

}

}

int main(int argc, char \* argv[]) {

char text[]="[\"Sunday\", \"Monday\", \"Tuesday\", \"Wednesday\", \"Thursday\", \"Friday\", \"Saturday\"]";

parser(text);

return 0;

}

**cJSONdemo2/CMakeLists.txt**：

cmake\_minimum\_required(VERSION 2.8.5)

project(cjson-example)

aux\_source\_directory(. CJSON\_EXAMPLE\_SRC)

add\_subdirectory(./lib)

add\_executable(cjson-example ${CJSON\_EXAMPLE\_SRC})

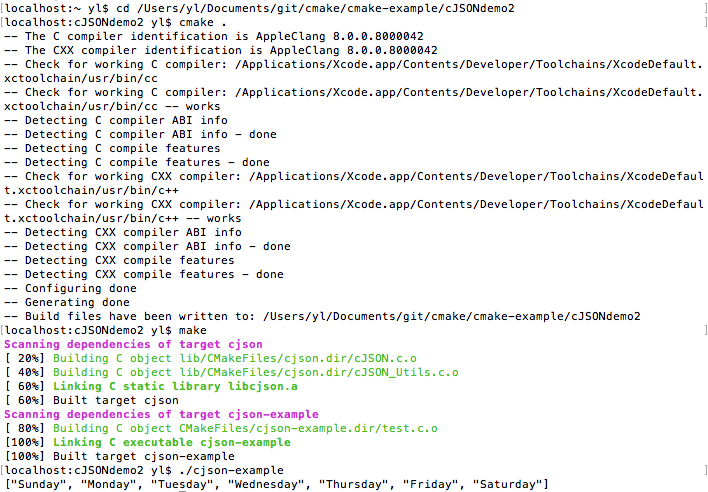
target\_link\_libraries(cjson-example cjson)

**cJSONdemo2/lib/CMakeLists.txt**：

cmake\_minimum\_required(VERSION 2.8.5)

aux\_source\_directory(. CJSON\_SRC)

add\_library(cjson STATIC ${CJSON\_SRC})

在终端下执行如下操作：  


##### 3.2.2 将cJSON库改为可选

在上面cJSONdemo2的基础上，新建cJSONConfig.h.in并相应修改test.c。目录如下：

cJSONdemo3

│── test.c

│── cJSONConfig.h.in

│── CMakeLists.txt

└── lib

├── cJScJSON\_UtilsON1.h

├── cJSON\_Utils.c

├── cJSON.h

├── cJSON.c

└── CMakeLists.txt

**cJSONConfig.h.in**：

#cmakedefine USE\_CJSON

**test.c**：

#include <stdio.h>

#include <stdlib.h>

#ifdef USE\_MYMATH

#include "lib/cJSON.h"

void parser(char\* text) {

char \*out;

cJSON \*json;

json = cJSON\_Parse(text);

if (!json) {

printf("Error before: [%s]\n", cJSON\_GetErrorPtr());

}else {

out = cJSON\_Print(json);

cJSON\_Delete(json);

printf("%s\n", out);

free(out);

}

}

#endif

int main(int argc, char \* argv[]) {

char text[]="[\"Sunday\", \"Monday\", \"Tuesday\", \"Wednesday\", \"Thursday\", \"Friday\", \"Saturday\"]";

#ifdef USE\_MYMATH

parser(text);

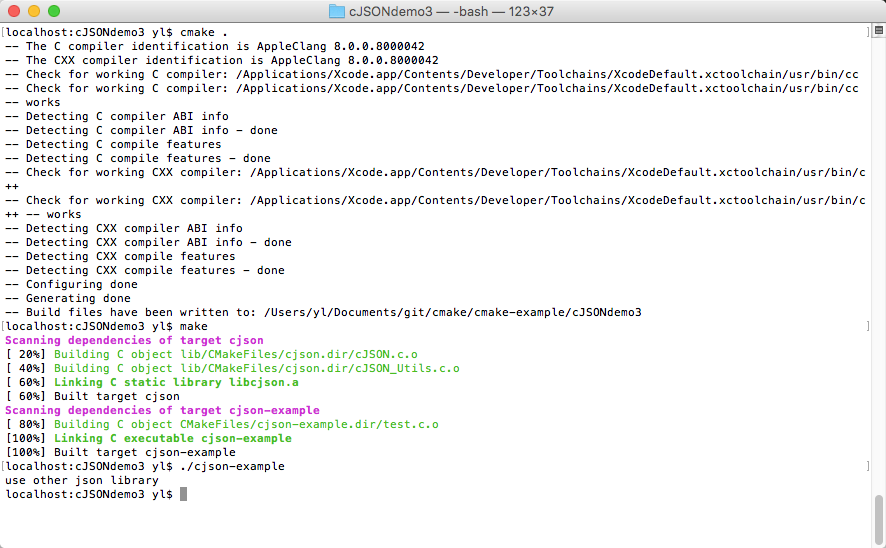
#else

printf("use other json library\n");

#endif

return 0;

}

最终效果如下：  


以上只是通过一些简单的示例来说明了CMake基础及相关应用。更多高级功能需要日常的实践及查询CMake官方文档。

技术交流QQ群：528655025  
作者：AlphaGL  
出处：<http://www.cnblogs.com/alphagl/>  
版权所有，欢迎保留原文链接进行转载 :)