CMake 完整入门教程

任麒麟自网络资源收集整理

1 前言

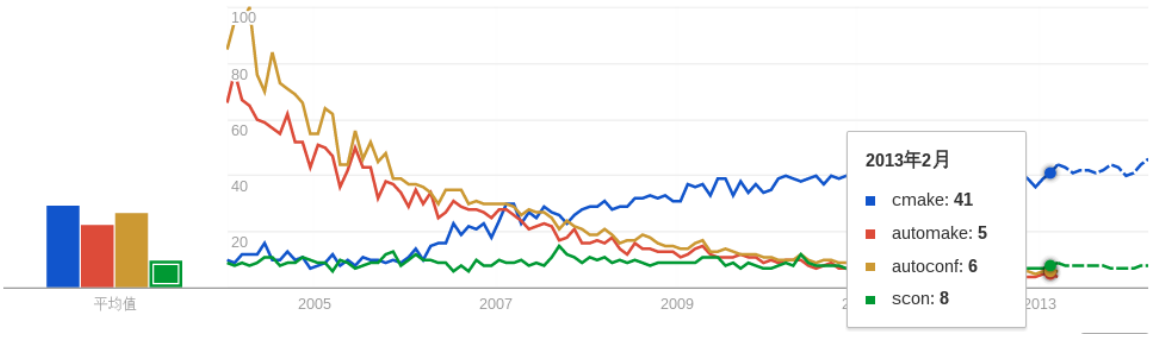
每一次学习新东西都是很有乐趣的，虽然刚开始会花费时间用来学习，但是实践证明，因为它们不同于老事物的优点，最后能为我省下的时间会更多。

网络上 cmake 的教程很多，但是我发现我很难找到一个完整、详细的中文版教程。本文档中的内容全部收集自网络。详细情况请见附录。

2 CMake 简介

CMake 是一个跨平台的安装(编译)工具，可以用简单的语句来描述所有平台的安装(编译过程)。他能够输出各种各样的 makefile 或者 project 文件，能测试编译器所支持的 C++特性，类似 UNIX 下的 automake。

由于 CMake 易于使用，以及在跨多平台的支持上做得更好，CMake 得到了越来越多的人的使用。下面的是来自 Google 的趋势图，可以看出 CMake 的应用情况。



2.1 CMakeLists.txt

CMake 靠的是 CMakeLists.txt 文件来生成工程的，事实上，CMakeList.txt 的编写就如使用 make 时编写 Makefile，只不过，相对来说 CMake 站的高度更高一些，所以虽然还是要编

写一个配置文件，但是 CMakefile 的编写比 makefile 轻松简单很多，而 CMake 最后其实还是通过生成 makefile 的方式来管理工程的（事实上，CMake 可以生成多种工程文件，甚至支持 eclipse 和 VC）。

CMakeLists.txt 里面则是具体的指令，用来告诉 CMake 如何生成工程。

2.2 编译和源代码分离

CMake 背后的逻辑思想是编译和源代码分离的原则。

通常 CMakeLists.txt 是和源代码放在一起的。一般每个子目录下都有一个 CMakeLists.txt 用于组织该目录下的文件。

而针对具体的平台和配置，我们可以单独创建一个目录，然后在该目录下生成特定平台和配置的工程文件。这样能够做到具体的工程文件不会和源代码文件混搭在一起。

2.3 CMakeLists.txt 自动继承父目录

子目录的 CMakeLists.txt 自动继承了父目录里的 CMakeLists.txt 所定义的一切宏、变量。这极大地减少了重复的代码。

3 CMake 安装

要想使用 cmake，先要安装它。去 www.cmake.org 下载一个最新的 cmake 版本。然后根据安装说明安装即可。

安装完毕后，打开系统命令行，输入：

```
cmake --version
```

如果你看到类似如下的字符串，说明你安装成功了。

```
cmake version 2.8.12.1
```

4 CMake 命令行指令

4.1 从命令行生成工程

对于一个已经配置好了 CMakeLists.txt 的项目来说，从命令行生成工程文件是很简单的一件事情。

下面是从命令行生成一个项目的工程文件的例子语句：

```
$cmake ..\Source -G "Visual Studio 10"
```

这条语句将在当前目录下，生成针对..\Source 目录的 Visual studio 2010 工程。 ..\Source 下必须已经定义了 CMakeLists.txt。

常用的 cmake 可以支持的工程类型为：

```
Visual Studio 10          = Generates Visual Studio 10 (2010) project files.
Visual Studio 11          = Generates Visual Studio 11 (2012) project.
Visual Studio 12          = Generates Visual Studio 12 (2013) project files.
MinGW Makefiles       = Generates a make file for use with mingw32-make.
Unix Makefiles            = Generates standard UNIX makefiles.
CodeBlocks - MinGW Makefiles    = Generates CodeBlocks project files.
CodeBlocks - NMake Makefiles    = Generates CodeBlocks project files.
CodeBlocks - Unix Makefiles         = Generates CodeBlocks project files.
Eclipse CDT4 - MinGW Makefiles  = Generates Eclipse CDT 4.0 project files.
Eclipse CDT4 - NMake Makefiles  = Generates Eclipse CDT 4.0 project files.
Eclipse CDT4 - Unix Makefiles       = Generates Eclipse CDT 4.0 project files.
```

4.2 生成 32 位和 64 位工程

对于 Windows MSVC，我们可以设定 CMake Generator 来确定生成 Win32 还是 Win64 工程文件，例如：

```
# 用于生成 Visual Studio 10 Win64 工程文件
$ cmake  -G "Visual Studio 10 Win64"
# 用于生成 Visual Studio 10 Win32 工程文件
```

```
$ cmake  -G "Visual Studio 10"
```

我们可以通过 CMake –help 来查看当前平台可用的 Generator

对于 UNIX 和类 UNIX 平台，我们可以通过编译器标志（选项）来控制进行 32 位还是 64 位构建。

4.3 从命令行定义全局变量

在执行 cmake 指令的时候，可以定义任意多个全局变量。这些全局变量可以直接在 CMakeLists.txt 中被使用。这是一项很方便的功能。例如，如果你希望利用 cmake 生成多种配置的工程，你可以将工程配置作为一个全局变量，在命令行指定。

在命令行定义全局变量的语法为：

```
$cmake  ..\Source  -G  "Visual Studio 10"   -DCONFIG=Debug   -DSSE=True
```

这条指令定义了两个全局变量：CONFIG 和 SSE，其值分别是"Debug"和"True"。

不要被这两个变量前面的-D 所迷惑。那只是用来告诉 cmake，要定义变量了。除此以外没有任何意义。

4.4 构建类型

CMake 为我们提供了四种构建类型：

- Debug
- Release
- MinSizeRel
- RelWithDebInfo

如果使用 CMake 为 Windows MSVC 生成 projects/workspaces 那么我们将得到上述的 4 种解决方案配置。

如果使用 CMake 生成 Makefile 时，我们需要做一些不同的工作。CMake 中存在一个变量 CMAKE_BUILD_TYPE 用于指定构建类型，此变量只用于基于 make 的生成器。我们可以这样指定构建类型：

```
$ cmake  -DCMAKE_BUILD_TYPE=Debug
```

这里的 CMAKE_BUILD_TYPE 的值为上述的 4 种构建类型中的一种。

4.5 直译模式

CMake 提供了直译模式，可以执行指定的 script 而不以生成 makefile 为目的，后面介绍的语法特色都可以在直译模式下练习。

```
$ cmake -P <script-file>
```

虽然这意味着我们可以将 CMake 拿来当作一般的 scripting language 使用，但 CMake 先天上就不是为了通用编程语言而设计，所以使用起来未必方便，特别是数学计算方面。

5 CMake 脚本基本语法

5.1 语法简介

CMake 的语法非常单纯，由指令 (command) 和注解所组成，所有的空白、换行、tab 都没有特殊作用，仅为语汇元素的区隔。

5.2 注释

凡是由 # 字符开头一直到换行字符间的内容皆会被视为注解，不会有任何作用。

```
# 这是注释
```

5.3 指令

5.3.1 基本语法

CMake script 由一连串的指令 (command) 组成，每个指令可有零至多个参数。使用指令的语法为指令名称加上小括号，括号内可以有零或若干个参数，指令则依照出现在 CMakeLists 当中的顺序执行。

指令是不分大小写的！

在 cmake 中，所有指令名称大小写都一视同仁，例如 Command、COMMAND 皆视为同一个指令。

例如 message 指令常用来输出讯息：

```
message(hello)
```

会输出：

```
hello
```

5.3.2 参数的格式

指令的参数通常使用空格、tab 或者换行来分隔，如：

```
command(arg1 arg2 arg3 ... argn)
command(
  arg1
  arg2
  arg3
  ...
  argn)
```

然而，值得注意的是，CMake 也支持用分号;来分隔参数。不过我强烈不建议使用分号来分隔参数。

5.3.3 在命令行查阅指令说明

输入：

```
cmake --help-command-list
```

可以查看到所有的指令列表。要想查阅某个指令的详细使用说明，例如 MESSAGE 指令，可以在命令行输入：

```
cmake --help-command message
```

5.3.4 在 CMake.org 网站上查阅指令说明

cmake 2.8.12 的指令说明可以在这个地址查阅到：

http://www.cmake.org/cmake/help/v2.8.12/cmake.html

5.4 变量

在撰写 CMakeLists 时可以使用变量储存资料以及作为指令的参数。

5.4.1 变量的特征

CMake 中的变量具有以下特征：

- 变量严格区分大小写！
- CMake 中的变量只有两种类型：字符串，和字符串数组。
- 变量无需声明即可赋值或者使用。未赋值的变量默认为一个空字符串。
- 与其他语言编程语言不同的是，CMake 脚本的语法中没有赋值操作。无论是赋值，还是比较、判断操作，都是通过内置指令来完成的。
- 变量可以认为都是全局的，哪怕在一个宏中定义的变量，也可以在宏的外面被访问到。

5.4.2 定义变量

字串和字串数组是 CMake 当中的唯一的两种变量类型。在 CMake 当中我们可以用 set() 指令来设定一个变量的值，变量会在第一次使用的时候自动初始化，无须宣告。提取变量值时通常必须在外面加上 ${} 符号，不过也有少数场合例外。

```
set(var hello)
message(${var})
```

会输出

```
hello
```

将字串用空白或分号分隔则表示字串数组。

```
set(foo this is a list)
set(foo this;is;a;list)
```

上面这两个指令作用完全相同，都是将变量 foo 值指定为一个字串数组，内含 this、is、a、list 四个字串。

如果在命令中，使用包含了字符串数组的变量作为参数会是怎样的情况呢？例如，下面的变量：

```
set(foo a b c)
```

将其作为参数传入一个指令：

```
command(${foo})
```

这等同于：

```
command(a b c)
```

将这个道理应用到其他地方。例如，要想在 foo 数组里面增加一个字符串怎么办呢？只要把 foo 变量作为一个参数传递进去就可以了：

```
set(foo ${foo} d)
```

执行了该指令后，变量 foo 中则包含了四个字串：a、b、c、d。

5.4.3 变量的递归代换

我们知道，要使用一个变量，语法${variable}可以提取出变量所存储的值。变量值的代换甚至可以递归进行，在撰写复杂的功能时可能很有用。例如：

```
set(var hello)
set(foo var)


message(${foo})
message(${${foo}})
```

会输出

```
var
hello
```

5.4.4 系统内建全局变量

CMake 预定义了一系列内建变量。请注意，所有的内建变量都是以大写来定义的。

例如：CMAKE_CURRENT_SOURCE_DIR，指的是当前处理的 CMakeLists.txt 所在的路径。

详细列表见后续章节。

5.4.5 cmake 调用环境变量的方式

使用$ENV{NAME}指令就可以调用系统的环境变量了。

比如

MESSAGE(STATUS “HOME dir: $ENV{HOME}”)

设置环境变量的方式是:

SET(ENV{变量名} 值)

5.5 字符串操作

5.5.1 不加引号直接使用字符串

在 CMake 中，指令的参数只有两种可能：

- 变量
- 字符串

如果字符串中不包含空格，那么可以不加引号，直接使用。例如：

```
set(var hello)
message(${var} world)
```

set 指令中使用了两个参数：第一个参数是字符串"var"，作为变量的名字；第二个参数是字符串"hello"，作为变量的值。

message 指令中使用了两个参数：变量 var，和字符串"world"。

5.5.2 在字符串中展开变量

在字符串中如果用${}将一个变量名包了起来，那么该变量也会被代换。

例如，如果我们执行下面的指令：

```
set(foo a b c d)
command("${foo}")
```

则相当于我们执行了 command("a b c d")。

5.5.3 使用特殊字符

在字串当中也可以插入空白、换行、分号等字符。例如：

```
set(a alpha beta gamma)
set(b "alpha beta gamma")
set(c "alpha
beta
gamma"
)
```

```
message("a = ${a}")
message("b = ${b}")
message("c = ${c}")
```

其输出为：

```
a = alpha;beta;gamma
b = alpha beta gamma
c = alpha
bata
gamma
```

注意：

- a 等于一个字串数组，内容为 alpha、beta、gamma 三个字串
- b 等于一个字串，内容为 alpha beta gamma
- c 等于一个字串，内容为以换行为分隔的 alpha beta gamma

5.5.4 转义字符串

CMake 大致上相容 C 语言当中的 Escape Sequence，如 \t \n 等等。如欲表示 CMake 当中的特殊字符时也可用 \ 标记。

```
set(bar "alpha beta gamma")
message("\${bar}: ${bar}")
```

上面的程式码输出

```
${bar}: alpha beta gamma
```

5.5.5 字符串连接

我们也可以利用 set 作字串串接：

```
set(a "alpha beta gamma")
set(b "${a} delta")
set(c ${a} "delta")
```

- b 等于一个字串，内容为 "alpha beta gamma dalta"

- c 等于一个字串数组，内容为 alpha beta gamma 和 "dalta" 两个字串

5.6 布尔值

在 CMake 当中有些字串被赋予了布尔值的意义，大小写差异会被忽略：

以下这些会被视为 FALSE：

- OFF
- FALSE
- N
- NO
- 0
- "" (空字串)
- 没被指派值的变量
- NOTFOUND
- 任何结尾是 -NOTFOUND 的字串

以下这些会被视为 TRUE:

- ON
- TRUE
- Y
- YE
- YES
- 1
- 其他不归类为 FALSE 的字串

5.7 数学计算

由于 CMake 当中并没有提供直接的数学运算符，所有的符号组合最终都会形成字串或字串数组。数学计算必须透过 math 指令解释：

```
math(EXPR var "1 + 2 * 3")
```

```
message("var = ${var}")
```

输出为

```
var = 7
```

6 脚本流程控制

CMake 也支援一般编程语言常用的流程控制和副程式，因此撰写弹性很大。

6.1 条件语句

CMake 的条件语句为 if、elseif、else、endif。

```
# 当 expr 值为下列其中之一時，执行 command1：
#  ON， 1， YES， TRUE， Y
# 当 expr 值为下列其中之一時，执行 command2：
#  OFF， 0， NO， FALSE， N， NOTFOUND， *-NOTFOUND， IGNORE

if(expr)
  command1(arg)
else(expr)
  command2(arg)
endif(expr)
```

版本较早的 CMake 要求在 else(...) 括号内必须填上对应的条件项目，然而很容易造成误导，例如

```
if(WIN32)
  ...
else(WIN32)
```

```
   command2(arg)
endif(WIN32)
```

乍看之下会以为 WIN32 为 TRUE 时执行 command2，但原意其实是 WIN32 为 FALSE 才执行 command2 ，因此在较新的版本中已经不强迫了。

```
# 以下也合法
if(WIN32)
   ...
else()
   command2(arg)
endif()
```

6.2 条件式计算规则

条件式的可以透过运算符组合，请参考运算符一章

```
if((expr) AND (expr OR (expr)))
```

在条件式当中即使不加 ${}，if 也会先尝试解释成变量。

```
# 下面两行意义相同
if (foo)
if (${foo})

# 下面两行意义相同
if (foo AND bar)
if (${foo} AND ${bar})
```

```

```

这里用 if 为例，while 亦为同理。

6.3 循环语句

CMake 的循环有两种：

- foreach ... endforeach
- while ... endwhile

```
set(V  alpha beta gamma)
message(${V})

foreach(i  ${V})
   message(${i})
endforeach()
```

Output:

```
alpha
beta
gamma
```

6.4 循环范围和步进

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

```
SET(A 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17)
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，直到遇到 ENDFOREACH 指令，整个语句块才会得到真正的执行。

7 运算符

表达式中可以包含运算符，运算符包括：

- 一元运算符，例如：EXISTS、COMMAND、DEFINED 等
- 二元运算符，例如：EQUAL、LESS、GREATER、STRLESS、STRGREATER 等
- NOT（非运算符）
- AND（与运算符）、OR（或运算符）

运算符优先级：一元运算符 > 二元运算符 > NOT > AND、OR

7.1 逻辑运算

当 expr 值为 FALSE 时成立。

```
if(NOT <expr>)
```

当 expr1 和 expr2 同时为 TRUE 时成立。

```
if(<expr1> AND <expr2>)
```

当 expr1 和 expr2 至少其中之一为 TRUE 时成立。

```
if(<expr1> OR <expr2>)
```

7.2 比较运算

7.2.1 数值比较

```
if(variable LESS number)
if(string LESS number)

if(variable GREATER number)
if(string GREATER number)

if(variable EQUAL number)
if(string EQUAL number)
```

7.2.2 字串比较

```
if(variable STRLESS string)
if(string STRLESS string)

if(variable STRGREATER string)
if(string STRGREATER string)

if(variable STREQUAL string)
if(string STREQUAL string)
```

字串比较依照 lexicographically order 决定大小。

LESS、GREATER、EQUAL、STRLESS、STRGREATER、STREQUAL 会分别检查左右算子是否为已定义过的变量，若是则采用变量值，否则采用字面值。

7.2.3 Regular Expression 比对

```
if(variable MATCHES regex)
if(string MATCHES regex)
```

MATCHS 会先检查左方算子是否为已定义过的变量，若是则会比对变量储存的字串值，否则将整串符号当成字串处理。

7.3 档案相关

判断档案和资料夹是否存在。行为只对完整路径是 well-defined。

```
if(EXISTS file-name)
if(EXISTS directory-name)
```

当 file1 比 file2 新，或者其中一个档案不存在时。行为只对完整路径是 well-defined。

```
if(file1 IS_NEWER_THAN file2)
```

判断给定的 path 是否是绝对路径。

```
if(IS_ABSOLUTE path)
```

7.4 其他

判断给定的 command-name 是否属于指令、function、macro。

```
if(COMMAND command-name)
```

判断给定的 variable-name 是否已经被定义过。

```
if(DEFINED variable-name)
```

8 自定义宏和函数

CMake 有两种设计子程序的方式：

- macro ... endmacro
- function ... endfunction

8.1 基本语法

基本语法如下：

```
macro(<name> [arg1 [arg2 [arg3 ...]]])
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endmacro(<name>)
function(<name> [arg1 [arg2 [arg3 ...]]])
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
endfunction(<name>)
```

下面是一个例子：

```
# 定义名为 print1 的 macro
macro(print1 MESSAGE)
  message(${MESSAGE})
endmacro(print1)

# 定义名为 print2 的 function
function(print2 MESSAGE)
  message(${MESSAGE})
```

```
endfunction(print2)


print1("from print1")
print2("from print2")
```

8.2 处理不定数目参数

以简单的求和函数为例，我们来看宏的一个示例：

```
macro(sum outvar)
   set(_args ${ARGN})
   set(result 0)

   foreach(_var ${_args})
      math(EXPR result "${result}+${_var}")
   endforeach()

   set(${outvar} ${result})
endmacro()

sum(addResult 1 2 3 4 5)
message("Result is :${addResult}")
```

上面是一段求和函数，我们来解读一下代码：

"${ARGN}"是 CMake 中的一个变量，指代宏中传入的多余参数。因为我们这个宏 sum 中只定义了一个参数"outvar"，其余需要求和的数字都是不定形式传入的，所以需要先将多余的参数传入一个单独的变量中。当然，在这个示例中，第一行代码显得多余，因为似乎没必要将额外参数单独放在一个变量中，但是建议这么做。

对上面这个宏再进一步加强：如果我们想限制这个宏中传入的参数数目（尽管在这个宏中实际上是不必要的），那么可以将宏改写一下：

```
macro(sum outvar)
   set(_args ${ARGN})
   list(LENGTH _args argLength)
```

```
    if(NOT argLength LESS 4)    # 限制不能超过 4 个数字
        message(FATAL_ERROR "to much args!")
    endif()
    set(result 0)

    foreach(_var ${ARGN})
        math(EXPR result "${result}+${_var}")
    endforeach()

    set(${outvar} ${result})
endmacro()

sum(addResult 1 2 3 4 5)
message("Result is :${addResult}")
```

8.3 宏和函数的区别

CMake 中的函数("function")与宏唯一的区别就在于，函数不能像宏那样将计算结果传出来(也不是完全不能，只是复杂一些)，并且函数中的变量是局部的，而宏中的变量在外面也可以被访问到，请看下例：

```
macro(macroTest)
    set(test1 "aaa")
endmacro()

function(funTest)
    set(test2 "bbb")
endfunction()

macroTest()
message("${test1}")

funTest()
```

```
message("${test2}")
```

运行这段代码后，只会打印出一条信息"aaa"，由此可以看到宏与函数的区别。

8.4 综合示例

最后我们来通过一个稍微复杂综合一点的宏来结束本小节。

下面的这个宏是找出指定数值范围内全部素数，并输出。

```
macro(GetPrime output maxNum)
  set(extArg ${ARGN})
  if(extArg)
    message(SEND_ERROR "To much args!")
  endif()

  # 没有判断传入的变量是否为数字类型
  set(result)
  foreach(_var RANGE 2 ${maxNum})
    set(isPrime 1)
    math(EXPR upplimit ${_var}-1)
    foreach(_subVar RANGE 2 ${upplimit})
      math(EXPR _temp "${_var}%${_subVar}")
      if(_temp EQUAL 0)
        set(isPrime 0)
        break()
      endif()
    endforeach()

    if(isPrime)
      list(APPEND result ${_var})
    endif()
  endforeach()
  set(output ${result})
endmacro()
```

```
GetPrime(output 100)
message("${output}")
```

9 在工程中查找并使用其他程序库的方法

在开发软件的时候我们会用到一些函数库，这些函数库在不同的系统中安装的位置可能不同，编译的时候需要首先找到这些软件包的头文件以及链接库所在的目录以便生成编译选项。例如一个需要使用博克利数据库项目，需要头文件 db_cxx.h 和链接库 libdb_cxx.so，现在该项目中有一个源代码文件 main.cpp，放在项目的根目录中。

9.1 程序库说明文件

在项目的根目录中创建目录 cmake/modules/，在 cmake/modules/ 下创建文件 Findlibdb_cxx.cmake，内容如下：

文件 Findlibdb_cxx.cmake

```
MESSAGE(STATUS "Using bundled Findlibdb.cmake...")
FIND_PATH(
  LIBDB_CXX_INCLUDE_DIR
  db_cxx.h
  /usr/include/
  /usr/local/include/
)

FIND_LIBRARY(
  LIBDB_CXX_LIBRARIES NAMES  db_cxx
  PATHS /usr/lib/ /usr/local/lib/
)
```

```
IF (LIBDB_CXX_INCLUDE_DIR AND LIBDB_CXX_LIBRARIES)
  SET(LIBDB_CXX_FOUND TRUE)
ENDIF (LIBDB_CXX_INCLUDE_DIR AND LIBDB_CXX_LIBRARIES)
```

文件 Findlibdb_cxx.cmake 的命名要符合规范: FindlibNAME.cmake ， 其中 NAME 是函数库的名称。Findlibdb_cxx.cmake 的语法与 CMakeLists.txt 相同。这里使用了三个命令：MESSAGE ， FIND_PATH 和 FIND_LIBRARY 。

命令 MESSAGE 会将参数的内容输出到终端。

命令 FIND_PATH 指明头文件查找的路径，原型如下：
find_path(<VAR> name1 [path1 path2 ...]) 该命令在参数 path* 指示的目录中查找文件 name1 并将查找到的路径保存在变量 VAR 中。第 3 行到 8 行的意思是在 /usr/include/ 和 /usr/local/include/ 中查找文件 db_cxx.h ， 并将 db_cxx.h 所在的路径保存在 LIBDB_CXX_INCLUDE_DIR 中。

命令 FIND_LIBRARY 同 FIND_PATH 类似，用于查找链接库并将结果保存在变量中。第 10 行到 13 行的意思是在目录/usr/lib/ 和 /usr/local/lib/ 中寻找名称为 db_cxx 的链接库，并将结果保存在 LIBDB_CXX_LIBRARIES 。

9.2 项目的根目录中的 CmakeList.txt

在项目的根目录中创建 CmakeList.txt ：

```
PROJECT(main)
CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
SET(CMAKE_SOURCE_DIR .)
SET(
   CMAKE_MODULE_PATH
   ${CMAKE_ROOT}/Modules
   ${CMAKE_SOURCE_DIR}/cmake/modules)
AUX_SOURCE_DIRECTORY(. DIR_SRCS)
ADD_EXECUTABLE(main ${DIR_SRCS})


FIND_PACKAGE( libdb_cxx REQUIRED)
```

```
MARK_AS_ADVANCED(
  LIBDB_CXX_INCLUDE_DIR
  LIBDB_CXX_LIBRARIES
)
IF (LIBDB_CXX_INCLUDE_DIR AND LIBDB_CXX_LIBRARIES)
  MESSAGE(STATUS "Found libdb libraries")
  INCLUDE_DIRECTORIES(${LIBDB_CXX_INCLUDE_DIR})
  MESSAGE( ${LIBDB_CXX_LIBRARIES} )
  TARGET_LINK_LIBRARIES(main ${LIBDB_CXX_LIBRARIES}
)
ENDIF (LIBDB_CXX_INCLUDE_DIR AND LIBDB_CXX_LIBRARIES)
```

在该文件中第 4 行表示到目录 ./cmake/modules 中查找 Findlibdb_cxx.cmake ， 8-19 行表示查找链接库和头文件的过程。第 8 行使用命令 FIND_PACKAGE 进行查找，这条命令执行后 CMake 会到变量 CMAKE_MODULE_PATH 指示的目录中查找文件 Findlibdb_cxx.cmake 并执行。第 13-19 行是条件判断语句，表示如果 LIBDB_CXX_INCLUDE_DIR 和 LIBDB_CXX_LIBRARIES 都已经被赋值，则设置编译时到 LIBDB_CXX_INCLUDE_DIR 寻找头文件并且设置可执行文件 main 需要与链接库 LIBDB_CXX_LIBRARIES 进行连接。

9.3 <name>_FOUND

标准的 Find 模块应该定义下面几个变量：

- <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。

我们再来看一个的例子,通过<name>_FOUND 来控制工程特性:

```
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)
```

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。

10 常用命令

10.1 project 命令

命令语法：project(<projectname> [languageName1 languageName2 ...])

命令简述：用于指定项目的名称

使用范例：project(Main)

10.2 cmake_minimum_required 命令

命令语法：cmake_minimum_required(VERSION major[.minor[.patch[.tweak]]] [FATAL_ERROR])

命令简述：用于指定需要的 CMake 的最低版本

使用范例：cmake_minimum_required(VERSION 2.8)

10.3 aux_source_directory 命令

命令语法：aux_source_directory(<dir> <variable>)

命令简述：用于将 dir 目录下的所有源文件的名字保存在变量 variable 中

使用范例：aux_source_directory(. DIR_SRCS)

10.4 add_executable 命令

命令语法：add_executable(<name> [WIN32] [MACOSX_BUNDLE] [EXCLUDE_FROM_ALL] source1 source2 … sourceN)

命令简述：用于指定从一组源文件 source1 source2 … sourceN 编译出一个可执行文件且命名为 name

使用范例：add_executable(Main ${DIR_SRCS})

10.5 add_library 命令

命令语法：add_library([STATIC | SHARED | MODULE] [EXCLUDE_FROM_ALL] source1 source2 … sourceN)

命令简述：用于指定从一组源文件 source1 source2 … sourceN 编译出一个库文件且命名为 name

使用范例：add_library(Lib ${DIR_SRCS})

10.6 add_dependencies 命令

命令语法：add_dependencies(target-name depend-target1 depend-target2 …)

命令简述：用于指定某个目标（可执行文件或者库文件）依赖于其他的目标。这里的目标必须是 add_executable、add_library、add_custom_target 命令创建的目标

10.7 add_subdirectory 命令

命令语法：add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])

命令简述：用于添加一个需要进行构建的子目录

使用范例：add_subdirectory(Lib)

10.8 target_link_libraries 命令

命令语法：target_link_libraries(<target> [item1 [item2 [...]]] [[debug|optimized|general]] …)

命令简述：用于指定 target 需要链接 item1 item2 …。这里 target 必须已经被创建，链接的 item 可以是已经存在的 target（依赖关系会自动添加）

使用范例：target_link_libraries(Main Lib)

10.9 set 命令

命令语法：set(<variable> <value> [[CACHE <type> <docstring> [FORCE]] | PARENT_SCOPE])

命令简述：用于设定变量 variable 的值为 value。如果指定了 CACHE 变量将被放入 Cache（缓存）中。

使用范例：set(ProjectName Main)

10.10 unset 命令

命令语法：unset(<variable> [CACHE])

命令简述：用于移除变量 variable。如果指定了 CACHE 变量将被从 Cache 中移除。

使用范例：unset(VAR CACHE)

10.11 message 命令

命令语法：message([STATUS|WARNING|AUTHOR_WARNING|FATAL_ERROR|SEND_ERROR] “message to display” …)

命令简述：用于输出信息

使用范例：message(“Hello World”)

10.12 include 命令

用来载入 CMakeLists.txt 文件，也用于载入预定义的 cmake 模块.

INCLUDE(file1 [OPTIONAL])

INCLUDE(module [OPTIONAL])

OPTIONAL 参数的作用是文件不存在也不会产生错误。

你可以指定载入一个文件，如果定义的是一个模块，那么将在 CMAKE_MODULE_PATH 中搜索这个模块并载入。

载入的内容将在处理到 INCLUDE 语句是直接执行。

10.13 include_directories 命令

命令语法：include_directories([AFTER|BEFORE] [SYSTEM] dir1 dir2 …)

命令简述：用于设定目录，这些设定的目录将被编译器用来查找 include 文件

使用范例：include_directories(${PROJECT_SOURCE_DIR}/lib)

10.14 find_file 命令

命令语法：find_file(<VAR> name1 [path1 path2 ...])

VAR 变量代表找到的文件全路径，包含文件名

10.15 find_path 命令

命令语法：find_path(<VAR> name1 [path1 path2 ...])

命令简述：用于查找包含文件 name1 的路径，如果找到则将路径保存在 VAR 中（此路径为一个绝对路径），如果没有找到则结果为 <VAR>-NOTFOUND。默认的情况下，VAR 会被保存在 Cache 中，这时候我们需要清除 VAR 才可以进行下一次查询（使用 unset 命令）。

使用范例：

```
find_path(LUA_INCLUDE_PATH lua.h ${LUA_INCLUDE_FIND_PATH})
if(NOT LUA_INCLUDE_PATH)
   message(SEND_ERROR "Header file lua.h not found")
endif()
```

10.16 find_library 命令

命令语法：find_library(<VAR> name1 [path1 path2 ...])

命令简述：用于查找库文件 name1 的路径，如果找到则将路径保存在 VAR 中（此路径为一个绝对路径），如果没有找到则结果为 <VAR>-NOTFOUND。一个类似的命令 link_directories 已经不太建议使用了

示例:

```
FIND_LIBRARY(libX X11 /usr/lib)
IF(NOT libX)
   MESSAGE(FATAL_ERROR “libX not found”)
ENDIF(NOT libX)
```

10.17 find_package 命令

命令语法：find_package(<name> [major.minor] [QUIET] [NO_MODULE]
[[REQUIRED|COMPONENTS] [componets...]])

用来调用预定义在 CMAKE_MODULE_PATH 下的 Find<name>.cmake 模块，你也可以自己定义 Find<name>模块，通过 SET(CMAKE_MODULE_PATH dir)将其放入工程的某个目录中供工程使用。

10.18 add_definitions 命令

命令语法：add_definitions(-DFOO -DBAR …)

命令简述：用于添加编译器命令行标志（选项），通常的情况下我们使用其来添加预处理器定义

使用范例：add_definitions(-D_UNICODE -DUNICODE)

如果要添加其他的编译器开关，可以通过 CMAKE_C_FLAGS 变量和 CMAKE_CXX_FLAGS 变量设置。

10.19 execute_process 命令

命令语法：

```
execute_process(COMMAND <cmd1> [args1...]]
        [COMMAND <cmd2> [args2...] [...]]
        [WORKING_DIRECTORY <directory>]
        [TIMEOUT <seconds>]
        [RESULT_VARIABLE <variable>]
        [OUTPUT_VARIABLE <variable>]
        [ERROR_VARIABLE <variable>]
        [INPUT_FILE <file>]
        [OUTPUT_FILE <file>]
        [ERROR_FILE <file>]
        [OUTPUT_QUIET]
        [ERROR_QUIET]
        [OUTPUT_STRIP_TRAILING_WHITESPACE]
        [ERROR_STRIP_TRAILING_WHITESPACE])
```

命令简述：用于执行一个或者多个外部命令。每一个命令的标准输出通过管道转为下一个命令的标准输入。WORKING_DIRECTORY 用于指定外部命令的工作目录，RESULT_VARIABLE 用于指定一个变量保存外部命令执行的结果，这个结果可能是最后一个执行的外部命令的退出码或者是一个描述错误条件的字符串，OUTPUT_VARIABLE 或者 ERROR_VARIABLE 用于指定一个变量保存标准输出或者标准错误，OUTPUT_QUIET 或者 ERROR_QUIET 用于忽略标准输出和标准错误。

使用范例：execute_process(COMMAND ls)

10.20 file 命令

命令简述：此命令提供了丰富的文件和目录的相关操作（这里仅说一下比较常用的）

使用范例：

```
# 目录的遍历
# GLOB 用于产生一个文件（目录）路径列表并保存在 variable 中
# 文件路径列表中的每个文件的文件名都能匹配 globbing expressions（非正则表达式，但是类似）
# 如果指定了 RELATIVE 路径，那么返回的文件路径列表中的路径为相对于 RELATIVE 的路径
# file(GLOB variable [RELATIVE path] [globbing expressions]...)

# 获取当前目录下的所有的文件（目录）的路径并保存到 ALL_FILE_PATH 变量中
file(GLOB ALL_FILE_PATH ./*)

# 获取当前目录下的 .h 文件的文件名并保存到 ALL_H_FILE 变量中
# 这里的变量 CMAKE_CURRENT_LIST_DIR 表示正在处理的 CMakeLists.txt 文件的所在的目录的绝对路径（2.8.3 以及以后版本才支持）
file(GLOB ALL_H_FILE RELATIVE ${CMAKE_CURRENT_LIST_DIR}
${CMAKE_CURRENT_LIST_DIR}/*.h)
```

11 CMake 常用变量

英文版的全部列表在这里：http://www.cmake.org/Wiki/CMake_Useful_Variables

11.1 CMake 系统信息

以下皆为字串类型

CMAKE_SYSTEM

系统全名，如 "Linux-2.4.22"，"FreeBSD-5.4-RELEASE" 或 "Windows 5.1"

CMAKE_SYSTEM_NAME

系统名称，如 "Linux"， "FreeBSD" or "Windows"

CMAKE_SYSTEM_VERSION

只有 CMAKE_SYSTEM 当中版本的部分

CMAKE_SYSTEM_PROCESSOR

CPU 名称，如 "Intel(R) Pentium(R) M processor 2.00GHz"

CMAKE_GENERATOR

在命令列所指定的 Generator 名称

11.2 系统旗标

以下皆为 BOOL 类型，若是与目前所属的操作系统/编译器相符，其值为 True，否则为 False。

UNIX

- 如果为真，表示为 UNIX-like 的系统，包括 Apple OS X 和 CygWin

WIN3

- 如果为真，表示为 Windows 系统，包括 CygWin
- 在 MINGW、CYGWIN、MSYS 亦为 True

APPLE

- 如果为真，表示为 Apple 系统

MINGW

MSYS

CYGWIN

BORLAND

WATCOM

MSVC， MSVC_IDE， CMAKE_COMPILER_2005， MSVC60， MSVC70， MSVC71，
MSVC80， MSVC90， MSVC10

不同版本的微软 Visual C++ 专案档

CMAKE_COMPILER_IS_GNUCC

目前使用 GNU C 编译器

CMAKE_COMPILER_IS_GNUCXX

目前使用 GNU C++ 编译器

CMAKE_SIZEOF_VOID_P

表示 void* 的大小（例如为 4 或者 8），可以使用其来判断当前构建为 32 位还是 64 位

11.3 资料夹和档案信息

CMAKE_SOURCE_DIR

内容为 source tree 根目录的完整路径，也就是 CMake 开始建置过程的进入点。

CMAKE_BINARY_DIR

内容为 binary tree 根目录的完整路径，在 in-source build 的时候值与 CMAKE_SOURCE_DIR 相同。

PROJECT_SOURCE_DIR

目前正在处理中的专案最上层目录，即内含 project() 指令的 CMakeLists 所在资料夹。

PROJECT_BINARY_DIR

目前所属专案的建置根目录。在 in-source build 时和 PROJECT_SOURCE_DIR 相同。

CMAKE_CURRENT_SOURCE_DIR

目前正在处理的 CMakeLists.txt 所在位置。

CMAKE_CURRENT_BINARY_DIR

目前正在处理的 CMakeLists.txt 对应的建置资料夹位置。在 in-source build 时和 CMAKE_CURRENT_SOURCE_DIR 相同。

CMAKE_CURRENT_LIST_DIR

表示正在处理的 CMakeLists.txt 文件的所在的目录的绝对路径（2.8.3 以及以后版本才支持）

CMAKE_ARCHIVE_OUTPUT_DIRECTORY

用于设置 ARCHIVE 目标的输出路径

CMAKE_LIBRARY_OUTPUT_DIRECTORY

用于设置 LIBRARY 目标的输出路径

CMAKE_RUNTIME_OUTPUT_DIRECTORY

用于设置 RUNTIME 目标的输出路径

11.4 编译选项

BUILD_SHARED_LIBS

将所有程式库 target 设成共享程式库，只对未指定类型的程式库有效。

CMAKE_BUILD_TYPE

控制建置类型，值可为下列之一:

- None: 编译器默认值
- Debug: 产生除错信息
- Release: 进行最佳化
- RelWithDebInfo: 进行最佳化，但仍然会启用 DEBUG flag

- MinSizeRel: 进行程式码最小化

特别要注意的是，CMAKE_BUILD_TYPE 在 configuration time 不会自动初始化为可读取的变量，必须要在使用者指定建置类型后才可以利用。

C 编译标志相关变量

- CMAKE_C_FLAGS
- CMAKE_C_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO]

是在呼叫 C 编译器时附加的额外选项。其中：

CMAKE_C_FLAGS 或 CMAKE_CXX_FLAGS 可以指定编译标志 CMAKE_C_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO] 或 CMAKE_CXX_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO] 则指定特定构建类型的编译标志，这些编译标志将被加入到 CMAKE_C_FLAGS 或 CMAKE_CXX_FLAGS 中去，例如，如果构建类型为 DEBUG，那么 CMAKE_CXX_FLAGS_DEBUG 将被加入到 CMAKE_CXX_FLAGS 中去。

C++ 编译标志相关变量

- CMAKE_CXX_FLAGS
- CMAKE_CXX_FLAGS_DEBUG
- CMAKE_CXX_FLAGS_RELEASE
- CMAKE_CXX_FLAGS_RELWITHDEBINFO

同上，但作用在 C++ 编译器 。

链接标志相关变量

- CMAKE_EXE_LINKER_FLAGS
- CMAKE_EXE_LINKER_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO]
- CMAKE_MODULE_LINKER_FLAGS
- CMAKE_MODULE_LINKER_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO]
- CMAKE_SHARED_LINKER_FLAGS
- CMAKE_SHARED_LINKER_FLAGS_[DEBUG|RELEASE|MINSIZEREL|RELWITHDEBINFO]

它们类似于编译标志相关变量。如 CMAKE_EXE_LINKER_FLAGS 是生成执行档时所使用的连结选项。CMAKE_SHARED_LINKER_FLAGS 是生成 shared library 时所使用的连结选项。

12 自定义编程语言

在默认的情况下 CMake 会启用 C、C++ 两种编程语言，CMake 内建了几种常见 C、C++ 编译器的相关知识，通常我们不需要费心思去操心编译细节。如果是 CMake 不认得的编程语言或编译器，就需要指定相关的编译规则。

举个具体的例子，撰写 Win32 程式时常用到 Windows Resource 语言，必须将 .rc 档案编译成 .res 档，再和其他 C、C++ 的 .obj 连结成最后的程式。在 Visual C++ 这并不会造成任何困扰，只要把 .rc 档当成一般的源代码一起喂给 add_executable() 等指令，Visual C++ 的统一编译窗口 cl.exe 会自动处理，不劳您费心。然而其它的编译器未必有如此贴心的服务，例如 MinGW 的 gcc 就不会主动呼叫 windres 来编译资源档，我们必须手动指定。

利用 enable_language() 可以启用一个编程语言，通常还必须要设定几个变量告诉 CMake 这个语言的编译器名称、源代码附档名、输出档附档名、命令列格式等等。由于 CMake 对 resource file 已经有部分的知识，我们只要设定编译器名称和用法即可。

```
if(MINGW)
   set(CMAKE_RC_COMPILER_INIT windres)
   enable_language(RC)
   set(CMAKE_RC_COMPILE_OBJECT
      "<CMAKE_RC_COMPILER> <FLAGS> -O coff <DEFINES> -i <SOURCE> -o <OBJECT>")
endif()
```

然后将 .rc 档当成一般的源代码加到 add_executable()、 add_library() 之中，CMake 就会自动辨识 .rc 档并且用上面 CMAKE_RC_COMPILE_OBJECT 给定的命令列格式编译。

13 CMake 使用实例

本节内容完全摘自网友 dbzhang800 的博客。我觉得很不错。感谢 dbzhang800 网友的详细讲解。

13.1 例子一

一个经典的 C 程序，如何用 cmake 来进行构建程序呢？

```
//main.c
#include <stdio.h>
int main()
{
   printf("Hello World!/n");
   return 0;
}
```

编写一个 CMakeList.txt 文件(可看做 cmake 的工程文件)：

```
project(HELLO)
set(SRC_LIST main.c)
add_executable(hello ${SRC_LIST})
```

然后，建立一个任意目录（比如本目录下创建一个 build 子目录），在该 build 目录下调用 cmake

- 注意：为了简单起见，我们从一开始就采用 cmake 的 out-of-source 方式来构建（即生成中间产物与源代码分离），并始终坚持这种方法，这也就是此处为什么单独创建一个目录，然后在该目录下执行 cmake 的原因

```
cmake .. -G"NMake Makefiles"
nmake
```

或者

```
cmake .. -G"MinGW Makefiles"
make
```

即可生成可执行程序 hello(.exe)

目录结构

```
+
|
```

```
+--- main.c
+--- CMakeList.txt
|
/--+ build/
  |
  +--- hello.exe
```

cmake 真的不太好用哈，使用 cmake 的过程，本身也就是一个编程的过程，只有多练才行。

我们先看看：前面提到的这些都是什么呢？

13.1.1 CMakeList.txt

第一行 project 不是强制性的，但最好始终都加上。这一行会引入两个变量

- HELLO_BINARY_DIR 和 HELLO_SOURCE_DIR

同时，cmake 自动定义了两个等价的变量

- PROJECT_BINARY_DIR 和 PROJECT_SOURCE_DIR

因为是 out-of-source 方式构建，所以我们要时刻区分这两个变量对应的目录

可以通过 **message** 来输出变量的值

```
message(${PROJECT_SOURCE_DIR})
```

set 命令用来设置变量

add_exectuable 告诉工程生成一个可执行文件。

add_library 则告诉生成一个库文件。

- 注意：CMakeList.txt 文件中，命令名字是不区分大小写的，而参数和变量是大小写相关的。

13.1.2 cmake 命令

cmake 命令后跟一个路径(..)，用来指出 CMakeList.txt 所在的位置。

由于系统中可能有多套构建环境，我们可以通过-G 来制定生成哪种工程文件，通过 cmake -h 可得到详细信息。

要显示执行构建过程中详细的信息(比如为了得到更详细的出错信息)，可以在 CMakeList.txt 内加入：

- SET(CMAKE_VERBOSE_MAKEFILE on)

或者执行 make 时

- $ make VERBOSE=1

或者

- $ export VERBOSE=1
- $ make

13.2 例子二

一个源文件的例子一似乎没什么意思，拆成 3 个文件再试试看：

- hello.h 头文件

```
#ifndef DBZHANG_HELLO_
#define DBZHANG_HELLO_
void hello(const char* name);
#endif //DBZHANG_HELLO_
```

- hello.c

```
#include <stdio.h>
#include "hello.h"

void hello(const char * name)
{
   printf ("Hello %s!/n", name);
}
```

- main.c

```
#include "hello.h"
int main()
```

```
{
    hello("World");
    return 0;
}
```

- 然后准备好 CMakeList.txt 文件

```
project(HELLO)
set(SRC_LIST main.c hello.c)
add_executable(hello ${SRC_LIST})
```

执行 cmake 的过程同上，目录结构

```
+
|
+--- main.c
+--- hello.h
+--- hello.c
+--- CMakeList.txt
|
/--+ build/
   |
   +--- hello.exe
```

例子很简单，没什么可说的。

13.3 例子三

接前面的例子，我们将 hello.c 生成一个库，然后再使用会怎么样？

改写一下前面的 CMakeList.txt 文件试试：

```
project(HELLO)
set(LIB_SRC hello.c)
set(APP_SRC main.c)
add_library(libhello ${LIB_SRC})
add_executable(hello ${APP_SRC})
```

```
target_link_libraries(hello libhello)
```

和前面相比，我们添加了一个新的目标 libhello，并将其链接进 hello 程序

然后想前面一样，运行 cmake，得到

```
+
|
+--- main.c
+--- hello.h
+--- hello.c
+--- CMakeList.txt
|
/--+ build/
  |
  +--- hello.exe
  +--- libhello.lib
```

里面有一点不爽，对不？

- 因为我的可执行程序(add_executable)占据了 hello 这个名字，所以 add_library 就不能使用这个名字了
- 然后，我们去了个 libhello 的名字，这将导致生成的库为 libhello.lib(或 liblibhello.a)，很不爽
- 想生成 hello.lib(或 libhello.a) 怎么办?

添加一行

```
set_target_properties(libhello PROPERTIES OUTPUT_NAME "hello")
```

就可以了

13.4 例子四

在前面，我们成功地使用了库，可是源代码放在同一个路径下，还是不太正规，怎么办呢？分开放呗

我们期待是这样一种结构

```
+
|
+--- CMakeList.txt
+--+ src/
| |
| +--- main.c
| /--- CMakeList.txt
|
+--+ libhello/
| |
| +--- hello.h
| +--- hello.c
| /--- CMakeList.txt
|
/--+ build/
```

哇，现在需要 3 个 CMakeList.txt 文件了，每个源文件目录都需要一个，还好，每一个都不是太复杂

- 顶层的 CMakeList.txt 文件

```
project(HELLO)
add_subdirectory(src)
add_subdirectory(libhello)
```

- src 中的 CMakeList.txt 文件

```
include_directories(${PROJECT_SOURCE_DIR}/libhello)
set(APP_SRC main.c)
add_executable(hello ${APP_SRC})
target_link_libraries(hello libhello)
```

- libhello 中的 CMakeList.txt 文件

```
set(LIB_SRC hello.c)
add_library(libhello ${LIB_SRC})
set_target_properties(libhello PROPERTIES OUTPUT_NAME "hello")
```

恩，和前面一样，建立一个 build 目录，在其内运行 cmake，然后可以得到

- build/src/hello.exe
- build/libhello/hello.lib

回头看看，这次多了点什么，顶层的 CMakeList.txt 文件中使用 add_subdirectory 告诉 cmake 去子目录寻找新的 CMakeList.txt 子文件

在 src 的 CMakeList.txt 文件中，新增加了 include_directories，用来指明头文件所在的路径。

13.5 例子五

前面还是有一点不爽：如果想让可执行文件在 bin 目录，库文件在 lib 目录怎么办？

就像下面显示的一样：

```
+ build/
|
+--+ bin/
| |
|  /--- hello.exe
|
/--+ lib/
   |
   /--- hello.lib
```

- 一种办法：修改顶级的 CMakeList.txt 文件

```
project(HELLO)
add_subdirectory(src bin)
add_subdirectory(libhello lib)
```

不是 build 中的目录默认和源代码中结构一样么，我们可以指定其对应的目录在 build 中的名字。

这样一来：build/src 就成了 build/bin 了，可是除了 hello.exe，中间产物也进来了。还不是我们最想要的。

- 另一种方法：不修改顶级的文件，修改其他两个文件

src/CMakeList.txt 文件

```
include_directories(${PROJECT_SOURCE_DIR}/libhello)
#link_directories(${PROJECT_BINARY_DIR}/lib)
set(APP_SRC main.c)
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
add_executable(hello ${APP_SRC})
target_link_libraries(hello libhello)
```

libhello/CMakeList.txt 文件

```
set(LIB_SRC hello.c)
add_library(libhello ${LIB_SRC})
set(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)
set_target_properties(libhello PROPERTIES OUTPUT_NAME "hello")
```

13.6 例子六

在例子三至五中，我们始终用的静态库，那么用动态库应该更酷一点吧。 试着写一下

如果不考虑 windows 下，这个例子应该是很简单的，只需要在上个例子的 libhello/CMakeList.txt 文件中的 add_library 命令中加入一个 SHARED 参数：

```
add_library(libhello SHARED ${LIB_SRC})
```

可是，我们既然用 cmake 了，还是兼顾不同的平台吧，于是，事情有点复杂：

- 修改 hello.h 文件

```
#ifndef DBZHANG_HELLO_
#define DBZHANG_HELLO_
#if defined _WIN32
  #if LIBHELLO_BUILD
    #define LIBHELLO_API __declspec(dllexport)
  #else
    #define LIBHELLO_API __declspec(dllimport)
```

```
  #endif
#else
  #define LIBHELLO_API
#endif
LIBHELLO_API void hello(const char* name);
#endif //DBZHANG_HELLO_
```

- 修改 libhello/CMakeList.txt 文件

```
set(LIB_SRC hello.c)
add_definitions("-DLIBHELLO_BUILD")
add_library(libhello SHARED ${LIB_SRC})
set(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)
set_target_properties(libhello PROPERTIES OUTPUT_NAME "hello")
```

恩，剩下来的工作就和原来一样了。

14 CMake 的局限性

世界上没有完美的东西。CMake 也有其局限性：

- CMake 并不遵守 GNU 规则。这使得 CMake 对一些开源软件的支持不够。例如，CMake 生成出的工程在 Linux 下不支持 make uninstall。
- CMake 和其他编译系统会打架。处理起来不容易。例如，如果不用 Qt 自带的 QMake，而是用 CMake 去编译 Qt 工程，那是一件费时费力的事情。
- CMake 不会认识开发者在 IDE 中新增加的文件。必须通过修改 CMakeLists.txt，才能让 CMake 知道有新的文件。

15 常用网络资源

15.1 CMake 文档

http://www.cmake.org/cmake/help/v2.8.8/cmake.html

15.2 CMake Wiki

http://www.cmake.org/Wiki/CMake

15.3 CMake 入门

http://zh.wikibooks.org/wiki/CMake_%E5%85%A5%E9%96%80

16 附录

引用资源列表如下。时间仓促，难免有遗漏。如果您发现我引用了你的文章请告知。如果您不希望您的文章被引用也请告知。可以到我的博客留言：http://blog.sina.com.cn/renqilin

- http://blog.csdn.net/dbzhang800/article/details/6314073
- http://tzc.is-programmer.com/show/476.html
- http://blog.csdn.net/vagrxie/article/details/4743484
- http://www.cs.swarthmore.edu/~adanner/tips/cmake.php
- http://www.cnblogs.com/doujiu/archive/2009/11/04/1596155.html
- http://blog.csdn.net/onion_autotest/article/details/7222954
- http://www.cnblogs.com/coderfenghc/archive/2013/01/20/2846621.html
- http://blog.csdn.net/gubenpeiyuan/article/details/8667279
- http://name5566.com/1795.html
- http://blog.sina.com.cn/s/blog_9ce5a1b501015avz.html
- http://nullget.sourceforge.net/?q=node/94
- http://blog.csdn.net/gubenpeiyuan/article/details/8667035
- http://blogs.gnome.org/swilmet/2012/09/05/switch-from-cmake-to-autotools/
- http://tech.uc.cn/?p=914