<https://blog.csdn.net/damenhanter/article/details/53859779>

# 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](http://lib.csdn.net/base/softwaretest" \o "软件测试知识库" \t "_blank)**编译器所支持的C++特性,类似UNIX下的automake。

### CMake 使用方法

    CMake的所有的语句都写在一个叫:CMakeLists.txt的文件中。当CMakeLists.txt文件确定后,可以用ccmake命令对相关 的变量值进行配置。这个命令必须指向CMakeLists.txt所在的目录。配置完成之后,应用cmake命令生成相应的makefile（在Unix like系统下）或者 project文件（指定用window下的相应编程工具编译时）。

    其基本操作流程为：

1. $> ccmake directory
2. $> cmake directory
3. $> make

  其中directory为CMakeList.txt所在目录；

* 第一条语句用于配置编译选项，如VTK\_DIR目录 ，一般这一步不需要配置，直接执行第二条语句即可，但当出现错误时，这里就需要认为配置了，这一步才真正派上用场；
* 第二条命令用于根据CMakeLists.txt生成Makefile文件；
* 第三条命令用于执行Makefile文件，编译程序，生成可执行文件；

CMake的执行就是这么简单，其难点在于如何编写CMakeLists.txt文件，下面结合例子简单介绍CMakeLists.txt的编写，看下面这个CMakeLists.txt

1. #project name
2. PROJECT(test\_math)
3. #head file path
4. INCLUDE\_DIRECTORIES(
5. include
6. )
7. #source directory
8. AUX\_SOURCE\_DIRECTORY(src DIR\_SRCS)
9. #set environment variable
10. SET(TEST\_MATH
11. ${DIR\_SRCS}
12. )
13. #set extern libraries
14. SET(LIBRARIES
15. libm.so
16. )
17. #add executable file
18. ADD\_EXECUTABLE(../bin/bin ${TEST\_MATH})
19. #add link library
20. TARGET\_LINK\_LIBRARIES(../bin/bin ${LIBRARIES})

            或者用下面这个CMakeLists.txt

**[cpp]** [view plain](http://blog.csdn.net/tianmohust/article/details/9096777) [copy](http://blog.csdn.net/tianmohust/article/details/9096777)

[在CODE上查看代码片](https://code.csdn.net/snippets/1623303)

1. #project name
2. PROJECT(test\_math)
4. add\_definitions("-Wall -lpthread -g")
6. #head file path
7. INCLUDE\_DIRECTORIES(
8. include
9. )
11. #source directory
12. AUX\_SOURCE\_DIRECTORY(src DIR\_SRCS)
14. #set environment variable
15. SET(TEST\_MATH
16. ${DIR\_SRCS}
17. )
19. #set extern libraries
20. SET(LIBRARIES
21. libm.so
22. )
24. # set output binary path
25. SET(EXECUTABLE\_OUTPUT\_PATH ${PROJECT\_BINARY\_DIR}/bin)
27. SET(FS\_BUILD\_BINARY\_PREFIX "Yfs")
29. #add executable file
30. ADD\_EXECUTABLE(${FS\_BUILD\_BINARY\_PREFIX}sqrt ${TEST\_MATH})
32. #add link library
33. TARGET\_LINK\_LIBRARIES(${FS\_BUILD\_BINARY\_PREFIX}sqrt ${LIBRARIES})

 这是一个测试数学函数的程序的CMakeLists.txt，"#"后面为注释的内容，CMake的命令全部为大写

第2行指定生成的工程名为test\_math

第4行指定头文件目录为include

第8行指定源文件目录为src，并将其赋值给环境变量DIR\_SRCS

第10行设定环境变量TEST\_MATH的值为环境变量DIR\_SRCS的值，此处用于显示如何用环境变量对环境变量进行赋值

第14行将数学函数库赋值给环境变量LIBRARIES，当然，可以不用这个环境变量，而在后面直接使用该库名

第18行用于指定生成文件，将环境变量TEST\_MATH目录下的所有文件编译生成../bin目录下的可执行文件bin

第20行指定../bin/bin执行时的链接库为环境变量LIBRARIES的值－libm.so

下面给出源文件  
/src/main.c:

1. #include<stdio.h>
2. #include"../include/a.h"
3. int main()
4. {
5. double b=25.0;
6. double a=0.0;
7. a=get\_sqrt(b);
9. [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html" \t "_blank)("a is %lf, b is %lf**\n**",a,b);
10. return 0;
11. }

/src/a.c

1. #include"../include/a.h"
2. double get\_sqrt(double var1)
3. {
4. return sqrt(var1);
5. }

/include/a.h

 #ifndef  A\_FILE\_HEADER\_INC

2. #define  A\_FILE\_HEADER\_INC
3. #include<math.h>
5. double get\_sqrt(double var1);
7. #endif

将CMakeLists.txt放在当前目录下，执行CMakeLists.txt

1. $> cmake .
2. $> make

即可生成可执行文件，在目录/bin下的bin文件，好了运行看其效果是否和所想一样。

====================

补充：

## 一、      基本使用

安装：下载二进制包后可直接解压使用

从源码安装则执行命令：./bootstrap; make; make install——尝试执行bootstrap失败

使用：cmake dir\_path，生成工程文件或makefile文件

## 二、      概念

**out-of-source** build，与in-source build相对，即将编译输出文件与源文件放到不同目录中；

## 三、      基本结构

1，依赖CMakeLists.txt文件，项目主目标一个，主目录中可指定包含的子目录；

2，在项目CMakeLists.txt中使用project指定项目名称，add\_subdirectory添加子目录

3，子目录CMakeLists.txt将从父目录CMakeLists.txt继承设置（TBD，待检验）

## 四、      语法

1.       #注释

2.       变量：使用set命令显式定义及赋值，在非if语句中，使用${}引用，if中直接使用变量名引用；后续的set命令会清理变量原来的值；

3.       command (args ...)  #命令不分大小写，参数使用空格分隔，使用双引号引起参数中空格

4.       set(var a;b;c) <=> set(var a b c)  #定义变量var并赋值为a;b;c这样一个string list

5.       Add\_executable(${var}) <=> Add\_executable(a b c)   #变量使用${xxx}引用

6.       条件语句：

if(var) #var 非empty 0 N No OFF FALSE... #非运算使用NOT

       …

else()/elseif() … endif(var)

7.       循环语句

Set(VAR a b c)

Foreach(f ${VAR})       …Endforeach(f)

8.       循环语句

WHILE() … ENDWHILE()

## 五、      内部变量

**CMAKE\_C\_COMPILER**：指定C编译器

**CMAKE\_CXX\_COMPILER**：

**CMAKE\_C\_FLAGS**：编译C文件时的选项，如-g；也可以通过add\_definitions添加编译选项

**EXECUTABLE\_OUTPUT\_PATH**：可执行文件的存放路径

**LIBRARY\_OUTPUT\_PATH**：库文件路径

**CMAKE\_BUILD\_TYPE**:：build 类型(Debug, Release, ...)，CMAKE\_BUILD\_TYPE=Debug

**BUILD\_SHARED\_LIBS**：Switch between shared and static libraries

**内置变量的使用：**

>> 在CMakeLists.txt中指定，使用set

>> cmake命令中使用，如cmake -DBUILD\_SHARED\_LIBS=OFF

## 六、      命令

**project** (HELLO)   #指定项目名称，生成的VC项目的名称；

>>使用${HELLO\_SOURCE\_DIR}表示项目根目录

**include\_directories**：指定头文件的搜索路径，相当于指定gcc的-I参数

>> include\_directories (${HELLO\_SOURCE\_DIR}/Hello)  #增加Hello为include目录

**link\_directories**：动态链接库或静态链接库的搜索路径，相当于gcc的-L参数

       >> link\_directories (${HELLO\_BINARY\_DIR}/Hello)     #增加Hello为link目录

**add\_subdirectory**：包含子目录

       >> add\_subdirectory (Hello)

**add\_executable**：编译可执行程序，指定编译，好像也可以添加.o文件

       >> add\_executable (helloDemo demo.cxx demo\_b.cxx)   #将cxx编译成可执行文件——

**add\_definitions**：添加编译参数

>> add\_definitions(-DDEBUG)将在gcc命令行添加DEBUG宏定义；

>> add\_definitions( “-Wall -ansi –pedantic –g”)

**target\_link\_libraries**：添加链接库,相同于指定-l参数

>> target\_link\_libraries(demo Hello) #将可执行文件与Hello连接成最终文件demo

**add\_library**:

>> add\_library(Hello hello.cxx)  #将hello.cxx编译成静态库如libHello.a

**add\_custom\_target**:

**message**( status|fatal\_error, “message”):

**set\_target\_properties**( ... ): lots of properties... OUTPUT\_NAME, VERSION, ....

**link\_libraries**( lib1 lib2 ...): All targets link with the same set of libs

## 七、      说明

1，CMAKE生成的makefile能够处理好.h文件更改时只编译需要的cpp文件；

## 八、      FAQ

### 1）  怎样获得一个目录下的所有源文件

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

>> 将dir中所有源文件（不包括头文件）保存到变量variable中，然后可以add\_executable (ss7gw ${variable})这样使用。

### 2）  怎样指定项目编译目标

>>  project命令指定

### 3）  怎样添加动态库和静态库

>> target\_link\_libraries命令添加即可

### 4）  怎样在执行CMAKE时打印消息

>> message([SEND\_ERROR | STATUS | FATAL\_ERROR] "message to display" ...)

>> 注意大小写

### 5）  怎样指定头文件与库文件路径

>> include\_directories与link\_directories

>>可以多次调用以设置多个路径

**>> link\_directories仅对其后面的targets起作用**

### 6）  怎样区分debug、release版本

>>建立debug/release两目录，分别在其中执行cmake -DCMAKE\_BUILD\_TYPE=Debug（或Release），需要编译不同版本时进入不同目录执行make即可；

**Debug版会使用参数-g；Release版使用-O3 –DNDEBUG**

>> 另一种设置方法——例如DEBUG版设置编译参数DDEBUG

IF(DEBUG\_mode)

    add\_definitions(-DDEBUG)

ENDIF()

在执行cmake时增加参数即可，例如cmake -D DEBUG\_mode=ON

### 7）  怎样设置条件编译

例如**debug版设置编译选项DEBUG**，并且更改不应改变CMakelist.txt

>> 使用option command，eg：

option(DEBUG\_mode "ON for debug or OFF for release" ON)

IF(DEBUG\_mode)

    add\_definitions(-DDEBUG)

ENDIF()

>> 使其生效的方法：首先cmake生成makefile，然后make edit\_cache编辑编译选项；[**Linux**](http://lib.csdn.net/base/linux)下会打开一个文本框，可以更改，该完后再make生成目标文件——emacs不支持make edit\_cache；

>> 局限：这种方法不能直接设置生成的makefile，而是必须使用命令在make前设置参数；对于debug、release版本，相当于需要两个目录，分别先cmake一次，然后分别make edit\_cache一次；

>> 期望的效果：在执行cmake时直接通过参数指定一个开关项，生成相应的makefile——可以这样做，例如cmake –DDEBUGVERSION=ON

### 8）  怎样添加编译宏定义

>> 使用add\_definitions命令，见命令部分说明

### 9）  怎样添加编译依赖项

用于确保编译目标项目前依赖项必须先构建好

>>add\_dependencies

### 10）        怎样指定目标文件目录

>> 建立一个新的目录，在该目录中执行cmake生成Makefile文件，这样编译结果会保存在该目录——类似

>> SET\_TARGET\_PROPERTIES(ss7gw PROPERTIES

                      RUNTIME\_OUTPUT\_DIRECTORY "${BIN\_DIR}")

### 11）        很多文件夹，难道需要把每个文件夹编译成一个库文件？

>> 可以不在子目录中使用CMakeList.txt，直接在上层目录中指定子目录

### 12）        怎样设定依赖的cmake版本

>>cmake\_minimum\_required(VERSION 2.6)

### 13）        相对路径怎么指定

>> ${projectname\_SOURCE\_DIR}表示根源文件目录，${ projectname \_BINARY\_DIR}表示根二进制文件目录？

### 14）        怎样设置编译中间文件的目录

>> TBD

### 15）        怎样在IF语句中使用字串或数字比较

>>数字比较LESS、GREATER、EQUAL，字串比STRLESS、STRGREATER、STREQUAL，

>> Eg：

set(CMAKE\_ALLOW\_LOOSE\_LOOP\_CONSTRUCTS ON)

set(AAA abc)

IF(AAA STREQUAL abc)

    message(STATUS "true")   #应该打印true

ENDIF()

### 16）        更改h文件时是否只编译必须的cpp文件

>> 是

### 17）        机器上安装了VC7和VC8，CMAKE会自动搜索编译器，但是怎样指定某个版本？

>> TBD

### 18）        怎样根据OS指定编译选项

>> IF( APPLE ); IF( UNIX ); IF( WIN32 )

### 19）        能否自动执行某些编译前、后命令？

>> 可以，TBD

### 20）        怎样打印make的输出

make VERBOSE=1

## 参考文献：

[1] CMake\_Tutorial.pdf

[2] CMake使用总结，http://blog.csdn[**.NET**](http://lib.csdn.net/base/dotnet)/keensword007/archive/2008/07/16/2663235.aspx

[3] <http://www.cmake.org/>

[4] 安装包中文档

[5] Andrej Cedilnik，[HOWTO](http://journals.iranscience.net:800/www.linuxjournal.com/www.linuxjournal.com/categories.php@op=newindex&catid=36" \t "_blank): Cross-Platform Software Development Using CMake，October, 2003

[6] Cjacker，CMake实践.PDF

=================================================================

**在linux下使用CMake构建应用程序**

本文介绍了一个跨平台的自动化构建系统 CMake 在 [linux](http://linux.chinaitlab.com/" \t "_blank) 上的使用方法。 CMake 是一个比 automake 更加容易使用的工具，能够使程序员从复杂的编译连接过程中解脱出来。文中通过一些例子介绍使用 CMake 处理多源文件目录的方法、查找并使用其他开发包的方法以及生成 debug 版和 release 版程序的方法。

    CMake 简介

    CMake 是一个跨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](http://windows.chinaitlab.com/" \t "_blank) Visual [C++](http://c.chinaitlab.com/) 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的生成。CMake 提供了比 autoconfig 更简洁的语法。在 [linux](http://linux.chinaitlab.com/" \t "_blank) 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。

    第一个工程

    现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:

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

    该命令会把参数 <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 提供了一个图形化的操作界面。cmake 命令的执行方式如下:

|  |
| --- |
| 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,如下图所示。

    图 1. camke 的运行结果

**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下:

图 2. 源代码分布情况

    其中 src 目录下的文件要编译成一个链接库。

    第一步，项目主目录中的 CMakeLists.txt

    在目录 step2 中创建文件 CMakeLists.txt 。文件内容如下:

    清单 3 目录 step2 中的 CMakeLists.txt

|  |
| --- |
| 1 PROJECT(main)  2 CMAKE\_MINIMUM\_REQUIRED(VERSION 2.6)  3 ADD\_SUBDIRECTORY( src )  4 AUX\_SOURCE\_DIRECTORY(. DIR\_SRCS)  5 ADD\_EXECUTABLE(main ${DIR\_SRCS} )  6 TARGET\_LINK\_LIBRARIES( main Test ) |

    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Test的链接库 。

    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 4. 目录 src 中的 CmakeLists.txt

|  |
| --- |
| 1 AUX\_SOURCE\_DIRECTORY(. DIR\_TEST1\_SRCS)  2 ADD\_LIBRARY ( Test ${DIR\_TEST1\_SRCS}) |

    在该文件中使用命令 ADD\_LIBRARY 将 src 目录中的源文件编译为共享库。

    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 step2 中依次执行命令 “cmake .” 和 “make” 得到结果如下:

    图3. 处理多源文件目录时 cmake 的执行结果

    在执行 cmake 的过程中,首先解析目录 step2 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。

**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**](http://lib.csdn.net/base/mysql)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。

    第一步，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：

    清单 5. 文件 Findlibdb\_cxx.cmake

|  |
| --- |
| 01 MESSAGE(STATUS "Using bundled Findlibdb.cmake...") 02  03 FIND\_PATH( 04   LIBDB\_CXX\_INCLUDE\_DIR 05   db\_cxx.h  06   /usr/include/  07   /usr/local/include/  08   ) 09  10 FIND\_LIBRARY( 11   LIBDB\_CXX\_LIBRARIES NAMES  db\_cxx 12   PATHS /usr/lib/ /usr/local/lib/ 13   ) |

    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。清单5第3－8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。清单5第10－13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 。  
    第二步, 项目的根目录中的 CmakeList.txt

    在项目的根目录中创建 CmakeList.txt ：

    清单 6. 可以查找链接库的 CMakeList.txt

|  |
| --- |
| 01 PROJECT(main) 02 CMAKE\_MINIMUM\_REQUIRED(VERSION 2.6) 03 SET(CMAKE\_SOURCE\_DIR .) 04 SET(CMAKE\_MODULE\_PATH ${CMAKE\_ROOT}/Modules${CMAKE\_SOURCE\_DIR}/cmake/modules)  05 AUX\_SOURCE\_DIRECTORY(. DIR\_SRCS) 06 ADD\_EXECUTABLE(main ${DIR\_SRCS}) 07  08 FIND\_PACKAGE( libdb\_cxx REQUIRED) 09 MARK\_AS\_ADVANCED( 10 LIBDB\_CXX\_INCLUDE\_DIR 11 LIBDB\_CXX\_LIBRARIES 12 ) 13 IF (LIBDB\_CXX\_INCLUDE\_DIR AND LIBDB\_CXX\_LIBRARIES) 14 MESSAGE(STATUS "Found libdb libraries") 15    INCLUDE\_DIRECTORIES(${LIBDB\_CXX\_INCLUDE\_DIR}) 16     MESSAGE( ${LIBDB\_CXX\_LIBRARIES} ) 17     TARGET\_LINK\_LIBRARIES(main ${LIBDB\_CXX\_LIBRARIES} 18 ) 19 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。

    第三步，执行 cmake

    完成 Findlibdb\_cxx.cmake 和 CMakeList.txt 的编写后在项目的根目录依次执行 “cmake . ” 和 “make ” 可以进行编译,结果如下图所示：

    图 4. 使用其他程序库时 cmake 的执行结果

**使用 cmake 生成 debug 版和 release 版的程序**

    在 Visual Studio 中我们可以生成 debug 版和 release 版的程序,使用 CMake 我们也可以达到上述效果。debug 版的项目生成的可执行文件需要有调试信息并且不需要进行优化,而 release 版的不需要调试信息但需要优化。这些特性在 gcc/g++ 中是通过编译时的参数来决定的,如果将优化程度调到最高需要设置参数-O3,最低是 -O0 即不做优化;添加调试信息的参数是 -g -ggdb ,如果不添加这个参数,调试信息就不会被包含在生成的二进制文件中。

    CMake 中有一个变量 CMAKE\_BUILD\_TYPE ,可以的取值是 Debug Release RelWithDebInfo 和 MinSizeRel。当这个变量值为 Debug 的时候,CMake 会使用变量 CMAKE\_CXX\_FLAGS\_DEBUG 和 CMAKE\_C\_FLAGS\_DEBUG 中的字符串作为编译选项生成 Makefile ,当这个变量值为 Release 的时候,工程会使用变量 CMAKE\_CXX\_FLAGS\_RELEASE 和 CMAKE\_C\_FLAGS\_RELEASE 选项生成 Makefile。

    现假设项目中只有一个文件 main.cpp ,下面是一个可以选择生成 debug 版和 release 版的程序的 CMakeList.txt ：

    清单 7

|  |
| --- |
| 1 PROJECT(main)  2 CMAKE\_MINIMUM\_REQUIRED(VERSION 2.6)  3 SET(CMAKE\_SOURCE\_DIR .)  4  5 SET(CMAKE\_CXX\_FLAGS\_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb")  6 SET(CMAKE\_CXX\_FLAGS\_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")  7  8 AUX\_SOURCE\_DIRECTORY(. DIR\_SRCS)  9 ADD\_EXECUTABLE(main ${DIR\_SRCS}) |

    第 5 和 6 行设置了两个变量 CMAKE\_CXX\_FLAGS\_DEBUG 和 CMAKE\_CXX\_FLAGS\_RELEASE, 这两个变量是分别用于 debug 和 release 的编译选项。编辑 CMakeList.txt 后需要执行 ccmake 命令生成 Makefile 。在进入项目的根目录,输入 "ccmake ." 进入一个图形化界面,如下图所示：

    图 5. ccmake 的界面

    按照界面中的提示进行操作,按 "c" 进行 configure ,这时界面中显示出了配置变量 CMAKE\_BUILD\_TYPE 的条目。如下图所示：

    图 6. 执行了 configure 以后 ccmake 的界面

    下面我们首先生成 Debug 版的 Makefile ：将变量 CMAKE\_BUILD\_TYPE 设置为 Debug ,按 "c" 进行 configure ，按 "g" 生成 Makefile 并退出。这时执行命令 find \* | xargs grep "O0" 后结果如下:

    清单 8 find \* | xargs grep "O0"的执行结果

|  |
| --- |
| CMakeFiles/main.dir/flags.make:CXX\_FLAGS = -O0 -Wall -g -ggdb  CMakeFiles/main.dir/link.txt:/usr/bin/c++ -O0 -Wall -g -ggdb  CMakeFiles/main.dir/main.cpp.o -o main -rdynamic  CMakeLists.txt:SET(CMAKE\_CXX\_FLAGS\_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb") |

    这个结果说明生成的 Makefile 中使用了变量 CMAKE\_CXX\_FLAGS\_DEBUG 作为编译时的参数。

    下面我们将生成 Release 版的 Makefile ：再次执行命令 "ccmake ." 将变量CMAKE\_BUILD\_TYPE 设置为 Release ,生成 Makefile 并退出。执行命令 find \* | xargs grep "O0" 后结果如下：

    清单 9 find \* | xargs grep "O0"的执行结果

|  |
| --- |
| CMakeLists.txt:SET(CMAKE\_CXX\_FLAGS\_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb") |

    而执行命令 find \* | xargs grep "O3" 后结果如下:

    清单 10. find \* | xargs grep "O3"的执行结果

|  |
| --- |
| CMakeCache.txt:CMAKE\_CXX\_FLAGS\_RELEASE:STRING=-O3 -DNDEBUG  CMakeCache.txt:CMAKE\_C\_FLAGS\_RELEASE:STRING=-O3 -DNDEBUG  CMakeFiles/main.dir/flags.make:CXX\_FLAGS = -O3 -Wall  CMakeFiles/main.dir/link.txt:/usr/bin/c++ -O3 -Wall  CMakeFiles/main.dir/main.cpp.o -o main -rdynamic  CMakeLists.txt:SET(CMAKE\_CXX\_FLAGS\_RELEASE "$ENV{CXXFLAGS} -O3 -Wall") |

    这两个结果说明生成的 Makefile 中使用了变量 CMAKE\_CXX\_FLAGS\_RELEASE 作为编译时的参数。