cmake快速入门

转载 墨尘深巷 2018-09-12 10:10:16 101764 收藏 288
分类专栏： 命令及工具 文章标签： cmake Makefile

本博文的大概框架：

1， cmake 的介绍，下载，安装和使用

2， cmake 的手册详解，我关注了 -C和-G 的使用

3， 在Linux中构建cmake 的工程

第一个问题： cmake 介绍，下载和安装以及使用：
https://fukun.org/archives/0421949.html

cmake是kitware公司以及一些开源开发者在开发几个工具套件(VTK)的过程中所产生的衍生品。后来经过发展，最终形成体系，在2001年成为一个独立的开放源代码项目。其官方网站是www.cmake.org，可以通过访问官方网站来获得更多关于cmake的信息，而且目前官方的英文文档比以前有了很大的改进，可以作为实践中的参考手册。

cmake的流行离不开KDE4的选择。KDE开发者在使用autotools近10年之后，终于决定为KDE4项目选择一个新的工程构建工具。之所以如此，用KDE开发者们自己话来说，就是：只有少数几个“编译专家”能够掌握KDE现在的构建体系。在经历了unsermake，scons以及cmake的选型和尝试之后，KDE4最终决定使用cmake作为自己的构建系统。在迁移过程中，进展一场的顺利，并获得了cmake开发者的支持。所以，目前的KDE4开发版本已经完全使用cmake来进行构建。

随着cmake 在KDE4项目中的成功，越来越多的项目正在使用cmake作为其构建工具，这也使得cmake正在成为一个主流的构建体系。

一、为何要使用项目构建工具?

为何要使用cmake和autotools之类的项目构建工具？ 我想，这恐怕是刚刚接触软件项目的人最应该问的问题之一了。

“Hello, world！“这个最经典的程序相信我们每个人都写过。无论在什么平台下，编译和运行这个程序都仅需要非常简单的操作。但事实上，hello,world最多只能算是一个实例程序，根本算不上一个真正的软件项目。

任何一个软件项目，除了写代码之外，还有一个更为重要的任务，就是如何组织和管理这些代码，使项目代码层次结构清晰易读，这对以后的维护工作大有裨益。使想一下，如果把一个像KDE4那么大的项目像hello world那样，把全部代码都放到一个main.cpp文件中，那将会是多么恐怖的一件事情。别说KDE4，就是我们随便一个几千行代码的小项目，也不会有人干这种蠢事。

决定代码的组织方式及其编译方式，也是程序设计的一部分。因此，我们需要cmake和autotools这样的工具来帮助我们构建并维护项目代码。
看到这里，也许你会想到makefile，makefile不就是管理代码自动化编译的工具吗？为什么还要用别的构建工具?

其实，cmake和autotools正是makefile的上层工具，它们的目的正是为了产生可移植的makefile，并简化自己动手写makefile时的巨大工作量。如果你自己动手写过makefile，你会发现，makefile通常依赖于你当前的编译平台，而且编写makefile的工作量比较大，解决依赖关系时也容易出错。因此，对于大多数项目，应当考虑使用更自动化一些的 cmake或者autotools来生成makefile，而不是上来就动手编写。

总之，项目构建工具能够帮我们在不同平台上更好地组织和管理我们的代码及其编译过程，这是我们使用它的主要原因。

二、cmake的主要特点：

cmake和autotools是不同的项目管理工具，有各自的特点和用户群。存在即为合理，因此我们不会对两者进行优劣比较，这里只给出cmake的一些主要特点：

1.开放源代码,使用类 BSD 许可发布。

2.跨平台,并可生成 native 编译配置文件,在 Linux/Unix 平台,生成 makefile,在 苹果平台,可以生成 xcode,在 Windows 平台,可以生成 MSVC 的工程文件。

3.能够管理大型项目,KDE4 就是最好的证明。

4.简化编译构建过程和编译过程。Cmake 的工具链非常简单:cmake+make。

5.高效率，按照 KDE 官方说法,CMake 构建 KDE4 的 kdelibs 要比使用 autotools 来构建 KDE3.5.6 的 kdelibs 快 40%,主要是因为 Cmake 在工具链中没有 libtool。

6.可扩展,可以为 cmake 编写特定功能的模块,扩充 cmake 功能。

三、安装cmake

安装cmake 对任何用户而言都不该再成为一个问题。几乎所有主流的Linux发行版的源中都包含有cmake的安装包，直接从源中添加即可。当然，也可以在官方网站下载源代码自行编译安装。

对于Windows和Mac用户，cmake的官方网站上有相应的安装包，下载安装即可，无须赘述。

注：为了能够测试本文中的实例程序，如果读者的Linux系统中所带的cmake版本低于2.6，请从官网下载2.6版本或以上的源代码进行编译并安装。
在linux下安装cmake

首先下载源码包

http://www.cmake.org/cmake/resources/software.html

这里下载的是cmake-2.6.4.tar.gz

随便找个目录解压缩

Example

```
tar -xzvf cmake-2.6.4.tar.gz
cd cmake-2.6.4
```

依次执行：

Example

```
./bootstrap
  make
make install
```

cmake 会默认安装在 /usr/local/bin 下面

四、从“Hello, world!”开始

了解cmake的基本原理并在系统中安好cmake后，我们就可以用cmake来演示那个最经典的”Hello, world!”了。

第一步，我们给这个项目起个名字——就叫HELLO吧。因此，第一部为项目代码建立目录hello，与此项目有关的所有代码和文档都位于此目录下。

第二步，在hello目录下建立一个main.c文件，其代码如下：

Example

```
#include
int main(void)
{
        printf(”Hello,Worldn”);
        return 0;
}
```

第三步，在hello目录下建立一个新的文件CMakeLists.txt，它就是cmake所处理的“代码
“。其实，使用cmake管理项目本身也是在编程，所

在CMakeLists.txt文件中输入下面的代码(#后面的内容为代码行注释):

```
#cmake最低版本需求，不加入此行会受到警告信息
CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
PROJECT(HELLO) #项目名称
#把当前目录(.)下所有源代码文件和头文件加入变量SRC_LIST
AUX_SOURCE_DIRECTORY(. SRC_LIST)
#生成应用程序 hello (在windows下会自动生成hello.exe)
ADD_EXECUTABLE(hello ${SRC_LIST})
```

至此，整个hello项目就已经构建完毕，可以进行编译了。

第四步，编译项目。

为了使用外部编译方式编译项目，需要先在目录hello下新建一个目录build(也可以是其他任何目录名)。现在，项目整体的目录结构为:

```
hello/
|– CMakeLists.txt
 |– build /
`– main.c
```

在windows下，cmake提供了图形界面，设定hello为source目录，build为二进制目录，然后点击configure即可开始构建，之后进入build目录运行make命令编译。

在linux命令行下，首先进入目录build，然后运行命令(注：后面的“..”不可缺少):

该命令使cmake检测编译环境，并生成相应的makefile。接着，运行命令make进行编译。编译后，生成的所有中间文件和可执行文件会在build目录下。 下面是我在ubuntu上的运行过程:

```
$ ls
hello
$ cd hello/build/
$ ls
$ cmake ..
– The C compiler identification is GNU
– The CXX compiler identification is GNU
– Check for working C compiler: /usr/bin/gcc
– Check for working C compiler: /usr/bin/gcc – works
– Detecting C compiler ABI info
 – Detecting C compiler ABI info - done
– Check for working CXX compiler: /usr/bin/c++
– Check for working CXX compiler: /usr/bin/c++ – works
– Detecting CXX compiler ABI info
– Detecting CXX compiler ABI info - done
– Configuring done
– Generating done
– Build files have been written to: /home/kermit/Project/cmake/hello/build
$ make
Scanning dependencies of target hello
[100%] Building C object CMakeFiles/hello.dir/main.c.o
Linking C executable hello
[100%] Built target hello
$ ls
CMakeCache.txt CMakeFiles cmake_install.cmake hello Makefile
$ ./hello
Hello,World
```

上面，我们提到了一个名词，叫外部编译方式。其实，cmake还可以直接在当前目录进行编译，无须建立build目录。但是，这种做法会将所有生成的中间文件和源代码混在一起，而且cmake生成的makefile无法跟踪所有的中间文件，即无法使用”make distclean”命令将所有的中间文件删除。因此，我们推荐建立build目录进行编译，所有的中间文件都会生成在build目录下，需要删除时直接清空该目录即可。这就是所谓的外部编译方式。

第二个问题： cmake 的手册详解:
http://www.cnblogs.com/coderfenghc/archive/2012/06/16/CMake_ch_01.html

第三个问题：在Linux中使用cmake 构建应用程序：
http://www.ibm.com/developerworks/cn/linux/l-cn-cmake/

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 Visual C++ 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 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

```
tiger@tiger-laptop:~/test/cmake/step1$ cmake .
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/tiger/test/cmake/step1
tiger@tiger-laptop:~/test/cmake/step1$ make
Scanning dependencies of target main
[100%] Building CXX object CMakeFiles/main.dir/main.cpp.o
Linking CXX executable main
[100%] Built target main
tiger@tiger-laptop:~/test/cmake/step1$ ./main
Hello word!
```

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

```
./step2
    |
    +--- main.cpp
    |
    +--- src
           |
           +-- Test1.h
           |
           +-- Test1.cpp
```

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

```
1  01 MESSAGE(STATUS "Using bundled Findlibdb.cmake...")
2  0203 FIND_PATH(
3  04   LIBDB_CXX_INCLUDE_DIR
4  05   db_cxx.h
5  06   /usr/include/
6  07   /usr/local/include/
7  08   )
8  09
9  10 FIND_LIBRARY(
10 11   LIBDB_CXX_LIBRARIES NAMES  db_cxx
11 12   PATHS /usr/lib/ /usr/local/lib/
12 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})

0708 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

```
tiger@tiger-laptop:~/learn/cmake/step3$ cmake .
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Using bundled Findlibdb.cmake...
-- Found libdb libraries
/usr/lib/libdb_cxx.so
-- Configuring done
-- Generating done
-- Build files have been written to: /home/tiger/learn/cmake/step3
tiger@tiger-laptop:~/learn/cmake/step3$ make
Scanning dependencies of target main
[ 50%] Building CXX object CMakeFiles/main.dir/main.cpp.o
[100%] Building CXX object CMakeFiles/main.dir/Student.cpp.o
Linking CXX executable main
[100%] Built target main
```

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

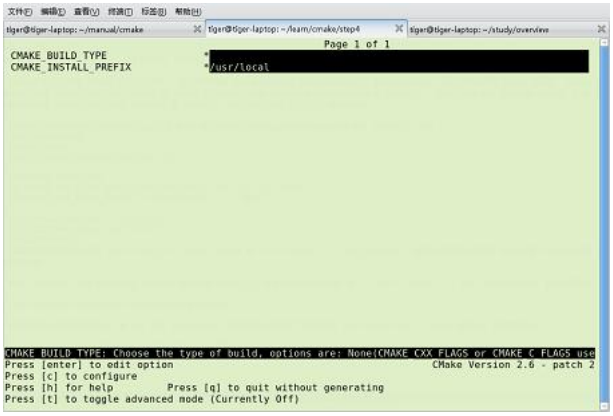
```
1 PROJECT(main)
2 CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
3 SET(CMAKE_SOURCE_DIR .)
45 SET(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb")
6 SET(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")
78 AUX_SOURCE_DIRECTORY(. DIR_SRCS)
9 ADD_EXECUTABLE(main ${DIR_SRCS})
```

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。

转载于：https://blog.csdn.net/appleml/article/details/52829412

相关文章：https://blog.csdn.net/wzzfeitian/article/details/40963457

https://blog.csdn.net/k346k346/article/details/46898635

语　法：https://www.mawenbao.com/note/cmake.html#fn1

https://www.jianshu.com/p/8909efe13308

https://gearyyoung.gitbooks.io/cmake-api/content/cmake/CMake%E8%AF%AD%E6%B3%95.html