怎样使用CMake快捷地生成Makefile?

当我们在类Unix系统下编写C/C++程序时，不可避免的会使用到Makefile进行编译和链接管理。但想要编写高质量的Makefile却并不容易（特别是在源码层次复杂，软件交付件较多的时候），于是，CMake诞生了。

> *CMake is an open-source, cross-platform family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice. CMake是一个开源跨平台的一组工具集，用于软件的构建，测试和打包。CMake利用简单的平台和不依赖编译器的配置文件可以生成适配任何编译环境的原生Makefile，从而管理软件的编译过程。——**摘自CMake官网***

本文的目标是，一步一步教会大家CMake到底怎么用。

学习路线：

1. CMake一般使用流程
2. 使用CMake构建多层次源码
3. 使用CMake构建多种交付件

> *C++和C语言在CMake的使用上完全相同，简单起见，本文均使用纯C语言作为案例。*

一. CMake一般使用流程

CMake是通过文件CMakeLists.txt管理编译过程。构建一个软件一般包含以下步骤：

- 编写CMakeLists.txt。文件中需要定义：要编译的源文件有哪些，要输出什么应用程序，编译时需要链接什么库，头文件的搜索路径等等。一般将该文件放到源码包的根路径。
- 在源码包根路径下创建build目录。进入build目录执行 `cmake ../` 生成适配当前环境的Makefile。

  > *创建build目录作为编译目录的形式称为外部编译（out-of-source build）。好处是：所有编译过程中产生的文件都只会存在于build目录，原始的源码文件会保持整洁。*

- 执行 `make` 进行编译。需要的话，还可以继续执行 `make install` 进行安装。

案例：用CMake构建源码包Sample。目录Sample中只有一个文件hello_world.c，内容如下：

```c
#include <stdio.h>
int main()
{
   puts("hello world!");
   return 0;
}
```

第一步，编写CMakeLists.txt文件

- 在Sample目录下创建CMakeLists.txt。

```
$ tree Sample/
Sample/
├── CMakeLists.txt
└── hello_world.c
```

- 类似编写Makefile，在CMakeLists.txt中对编译过程做简单的定义。

```
#指定当前文件所依赖的最低CMake版本，若没有必要用到高版本的CMake，建议指定较低的版本3.0
cmake_minimum_required (VERSION 3.0)
#指定当前工程名称，一般设定成源码包的根目录名
project("Sample")
#指定要构建的应用程序，参数的含义是：将参数2（hello_world.c）编译成参数1（hello_world）
add_executable(hello_world hello_world.c)
```

第二步，执行cmake

- 创建build目录

```
$ tree Sample/
Sample/
├── build
├── CMakeLists.txt
└── hello_world.c
```

- 进入build目录执行 `cmake ../` 生成Makefile

```
$ pwd
Sample/build

# cmake的第一个参数（../）用于指定当前工程的CMakeLists.txt所在路
```

```
# cmake的第一个参数（../）用于指定当前工程的CMakeLists.txt所在路径
$ cmake ../
cmake ../
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - 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
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: Sample/build
# 执行过程包含：编译工具链检测，必要的依赖库检测，生成Makefile等。

#执行成功后会生成如下文件或目录。最关键的就是Makefile生成了，意味着后续可以基于Makefile执行相应的构建指令
$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  Makefile
```

第三步，编译并测试

上述步骤已经帮助我们生成了Makefile，现在只需要按照普通Makefile的使用方式使用即可。

```
$ make
Scanning dependencies of target hello_world
[ 50%] Building C object CMakeFiles/hello_world.dir/hello_world.o
[100%] Linking C executable hello_world
[100%] Built target hello_world
# 编译完成后，可以看到当前生成了可执行文件hello_world
$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  hello_world  Makefile
#测试OK
$ ./hello_world
hello world!
```

小结：

- add_executable是上述CMakeLists.txt文件中最核心的命令。当我们构建可执行的应用程序时，必然要使用该命令指定输出文件和依赖源码。该命令的更多用法，参考官方文档：https://cmake.org/cmake/help/v3.0/command/add_executable.html
- 类似Makefile，CMakeLists.txt文件有自己一套比较完善的语法，在CMake官网可以找到相关参考。我们这边不需要大而全的掌握所有语法，只需要会用一些关键的命令即可。

> *这是一个最简单项目的CMake使用案例，但无论多复杂的项目，CMake的使用方法都是相同的。本文后续将只关注CMakeLists.txt文件的编写，后续的构建过程（cmake make等命令）将不再赘述。*

二. 使用CMake构建多层次源码

一般地，我们的软件都是由若干个源文件一同编译出来的。因此，我们需要在add_executable命令中指定多个源文件，例如：`add_executable（test a.c b.c c.c d.c）`。但这显然不是一个好的解决办法（过于冗长且不方便扩展）。 所以我们接下来讨论**怎样编写CMakeLists.txt来管理更复杂的源码结构**

多文件编译的CMakeLists.txt编写的核心思想是**变量化**。编写的步骤包括：

- 搜索当前目录和子目录下的所有源文件，并存储到变量里。
- 使用变量指定待编译的源文件

案例：用CMake构建源码包Sample。源码包内包含：

1. 文件main.c作为程序入口
2. 目录modules存放若干模块的源文件
3. 目录utils存放通用的算法和实用工具的源码文件

```
$ tree Sample/
Sample/
├── main.c
├── modules
│   ├── module_a.c
│   └── module_a.h
└── utils
    ├── convert.c
    └── convert.h
```

- convert.h和convert.c定义和声明了小写转大写的函数

```
/*这是convert.h的内容*/
#ifndef _CONVERT_H_
#define _CONVERT_H_


char ConvertUpper(char lowercase);
```

```
char ConvertUpper(char _lowercase);

#endif

/*这是convert.c的内容*/
#include "convert.h"

char ConvertUpper(char _lowercase)
{
    char upper = _lowercase - 32;

    return upper;
}
```

- module_a.c和module_a.h中定义和声明了打印大写字母的函数

```
/*这是module_a.h的内容*/
#ifndef _MODULE_A_H_
#define _MODULE_A_H_

void PrintUpperCass(char _letter);

#endif

/*这是module_a.c的内容*/
#include <stdio.h>
#include "module_a.h"
#include "../utils/convert.h"

void PrintUpperCass(char _letter)
{
    char upper = ConvertUpper(_letter);
    printf("%c\n", upper);
}
```

- main.c中定义了main函数，用来调用PrintUpperCass输出字母A

```
#include "modules/module_a.h"

int main()
{
    PrintUpperCass('a');

    return 0;
}
```

接下来我们在源码包的根路径下创建CMakeLists.txt文件并编写其内容。

```
$ tree Sample/
Sample/
├── CMakeLists.txt  #重点关注怎样编写该文件
├── main.c
├── modules
│   ├── module_a.c
│   └── module_a.h
└── utils
    ├── convert.c
    └── convert.h
```

编写CMakeLists.txt文件

- 使用命令aux_source_directory可以搜索某个目录下所有源文件并保存到变量中。
- 使用变量指定待编译的文件。CMakeLists.txt文件的语法中规定，使用${}的形式访问变量

```
#指定当前文件所依赖的最低CMake版本，若没有必要用到高版本的CMake，建议指定较低的版本3.0
cmake_minimum_required (VERSION 3.0)
#指定当前工程名称，一般设定成源码包的根目录名
project("Sample")
#用变量ROOT_SRC存放根路径下的所有源文件（本例中就是main.c）
aux_source_directory(. ROOT_SRC)
#用变量MODULES_SRC存放modules目录下的所有源文件（本例中是module_a.c）
aux_source_directory(./modules MODULES_SRC)
#用变量UTILS_SRC存放utils目录下的所有源文件（本例中是convert.c）
aux_source_directory(./utils UTILS_SRC)
#指定使用上述三个变量中的所有文件编译成main这个可执行文件
add_executable(main ${ROOT_SRC} ${MODULES_SRC} ${UTILS_SRC})
```

> *命令add_executable可以添加多个参数用于指定待编译文件，文件之间使用空格隔开即可*

构建并测试

```
# 创建并进入build目录用于外部编译
$ mkdir build
$ cd build/
#执行cmake生成makefile
$ cmake ../
```

```
$ cmake ../
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - 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
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: Sample/build
# cmake执行成功，执行make进行构建
$ make
Scanning dependencies of target main
[ 25%] Building C object CMakeFiles/main.dir/main.c.o
[ 50%] Building C object CMakeFiles/main.dir/modules/module_a.c.o
[ 75%] Building C object CMakeFiles/main.dir/utils/convert.c.o
[100%] Linking C executable main
[100%] Built target main
# 可执行文件main构建成功
$ ls
CMakeCache.txt  CMakeFiles  cmake_install.cmake  main  Makefile
#测试成功
$ ./main
A
```

小结:

- 因为使用了aux_source_directory命令，所以，在当前存在的目录下添加任何新的源文件，都可以被搜索到并执行编译。
- 添加新文件后，要重新执行`cmake ../`，用来生成包含新文件的Makefile

三. 使用CMake构建多种交付件

很多时候，我们创建的软件项目不只包含可执行程序，还有可能包含一些开发库用于功能复用和开放第三方开发。所以，接下来我们看一下**如何使用CMake同时构建可执行程序和动态库。**

构建多种交付件一般包含三步：

- 在根路径CMakeLists.txt中指定各种交付件的输出路径和链接关系

- 在根路径CMakeLists.txt中指定子目录用于区别不同交付件
- 在指定的子目录中编写CMakeLists.txt并添加输出交付件的描述

案例：基于前边第二章的源码，之前的功能保留，添加功能：将大写字母变小写的功能编译成动态库。

修改根路径CMakeLists.txt

- 指定应用程序和动态库的输出路径

按照惯例，一般可执行程序应该放置到XXXX/bin/目录下，动态库一般放置到XXXX/lib/目录下。因此我们通过分别设置两个变量的形式，将可执行程序输出到build/bin/目录下，将动态库输出到build/lib/目录下。

> *在CMake中已经提供了一个内置变量PROJECT_BINARY_DIR代表执行cmake命令的路径。在此例中变量PROJECT_BINARY_DIR代表的是build目录。*

- 移除原来对utils子目录的源码搜索，改为指定工程子目录

> *一般建议基于源码目录拆分交付件，不同的交付件由各自目录下的CMakeLists.txt管理。在本例中utils目录用来编译动态库，所以指定utils为工程子目录。*

- 添加可执行程序对动态库的依赖

```
#指定当前文件所依赖的最低CMake版本，若没有必要用到高版本的CMake，建议指定较低的版本3.0
cmake_minimum_required (VERSION 3.0)
#指定当前工程名称，一般设定成源码包的根目录名
project("Sample")
#EXECUTABLE_OUTPUT_PATH用于指定可执行文件的输出路径，LIBRARY_OUTPUT_PATH用于指定动态库输出路径
set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
set(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)
#源码搜索路径（移除了utils子目录的搜索）
aux_source_directory(. ROOT_SRC)
aux_source_directory(./modules MODULES_SRC)
#指定当前工程包含子目录utils
add_subdirectory(utils)
#构建可执行程序的命令中只需要指定modules目录下的源码和根路径下的源码即可
add_executable(main ${ROOT_SRC} ${MODULES_SRC})
#指定可执行程序的依赖：main依赖（需要链接）sample_utils（这里指定动态的名称即可）
target_link_libraries(main sample_utils)
```

修改子目录CMakeLists.txt

- 在子目录utils下创建CMakeLists.txt

```
Sample/
├── CMakeLists.txt
├── main.c
├── modules
│   ├── module_a.c
│   └── module_a.h
└── utils
    ├── CMakeLists.txt #该文件用于管理utils目录下的编译过程
    ├── convert.c
    └── convert.h
```

- 添加源码搜索和输出动态库的命令

```
# 搜索当前目录（utils）下所有文件
aux_source_directory(. UTILS_SRC)
#指定该目录下所有源文件编译成动态库。参数1是动态库名称，参数2是链接类型，SHARED代表动态链接，参数3既是待编译的文件
add_library(sample_utils SHARED ${UTILS_SRC})
```

构建并测试

```
$ mkdir build
$ cd build/
$ cmake ../
-- The C compiler identification is GNU 5.4.0
-- The CXX compiler identification is GNU 5.4.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - 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
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done

-- Build files have been written to: Sample/build
```

```
20  -- Build files have been written to: Sample/build
21  $ make
22  Scanning dependencies of target sample_utils
23  [ 20%] Building C object utils/CMakeFiles/sample_utils.dir/convert.c.o
24  [ 40%] Linking C shared library ../lib/libsample_utils.so
25  [ 40%] Built target sample_utils
26  Scanning dependencies of target main
27  [ 60%] Building C object CMakeFiles/main.dir/main.c.o
28  [ 80%] Building C object CMakeFiles/main.dir/modules/module_a.c.o
29  [100%] Linking C executable bin/main
30  [100%] Built target main
31  #编译成功后可以看到生成了两个目录分别是bin和lib，其中分别存放了可执行程序和动态库
32  $ ls
33  bin CMakeCache.txt CMakeFiles cmake_install.cmake lib Makefile utils
34  $ ls bin/
35  main
36  #动态库的文件名按照惯例会设定为lib库名.so
37  $ ls lib/
38  libsample_utils.so
39  #测试成功
40  $ ./bin/main
41  A
```

小结:

- set命令用于设置变量，更多可设置的命令以及设置方式应参考：https://cmake.org/cmake/help/v3.0/manual/cmake-variables.7.html
- add_subdirectory命令可以用于指定当前编译的目录下待编译的子目录。子目录中必须包含一个CMakeLists.txt文件
- target_link_libraries命令用来指定待编译的交付件依赖的链接库。若有多个依赖(包括系统库的依赖)时可以填入多个参数。比如 `target_link_libraries（out openssl XXX pthread）` 代表out的依赖包括系统库openssl，pthread，和非系统库XXX。
- add_library命令用于指定将源码编译成链接库，若指定SHARED关键字则代表编译成动态库，否则则编译成静态库。根据库类型不同，最终的输出文件会自动添加lib，.a， .so等前后缀。

总结

至此，我们大概了解了CMake的使用方式和其配置文件CMakeLists.txt的简单编写。并且可以通过CMake生成比较好用的Makefile。 一般地，使用CMake进行编译管理分为以下几步

1. 编写CMakeLists.txt文件（类似Makefile的编写思路）
    - 为不同的交付件指定依赖的源码（add_executable和add_library命令）
    - 指定目录层次依赖（add_subdirectory命令）

        - 指定交付件链接时的依赖关系（target_link_libraries命令）

    2. 创建build目录并执行 `cmake ../`
    3. 执行make构建

额外地，CMake是一个功能相当丰富的编译管理软件。

- 在CMake中没有引用任何操作系统相关的编译链接指令，所以，CMake是可以跨平台使用的。
- CMake还可以通过继续丰富CMakeLists.txt文件实现类似 `make install` 的功能和单元测试的功能。更多使用方式请参考官方文档：https://cmake.org/cmake/help/v3.0/index.html