CMake 相关

一.第一次尝试结果：
我将源码目录建为 src，编译目录建为 build.
然后在 src 下建立 main，用于放 main 相关的文件，再在 src 下建立 lib1，用于放一个小库。
Magic Happens like this:
(1)main 和 lib1 中的 CMakeLists.txt，只需要写上和 Build Target 相关的 command。这里是 ADD_LIBRARY()或 ADD_EXECUTABLE()，另外因为 main 要链接 lib1 库，所以要添加 Build Flags(Options)相关的：TARGET_LINK_LIBRARIES()。
(2)然后在 main 和 lib1 的同级目录，即 src 下，建立工程的总的 CMakeLists.txt。这里面就放 SUBDIRS()以及和 Build Flags(Options)相关的就好。Build Flags(Options)可用 INCLUDE_DIRECTORIES()来将 lib1 写入，这样，main 中的程序要用 lib1 库，就只需要写"lib.h"就好，而不需要给出路径。然后，不用写 LINK_DIRECTORIES()，可能有些版本要写。但我这个版本，不用给出工程中生成的库的路径，就可以链接。

还有一个 Magic 是，只需要一遍 cmake，之后如果源文件或 CMakeLists.txt 做了更改，都不必要 cmake，直接 make，它会看情况 rerun cmake.

我习惯于：公共的东西，放在 top 的 CMakeLists.txt 里，各个库和执行文件相关的放在各个目录的 CMakeLists.txt 里，比如各个库怎么装。top 的 CMakeLists.txt 的例子：

```
PROJECT(helloworld)
SUBDIRS(main lib1 lib2)
#设置工程需要的头文件路径（包括工程内的）
INCLUDE_DIRECTORIES(./lib1 ./lib2 ${CMAKE_BINARY_DIR}/config)
#设置用户选项
OPTION(HELLO1 "Using Lib1" ON)
#将 config.h.in 转为 config.h
CONFIGURE_FILE(${CMAKE_SOURCE_DIR}/config/config.h.in
${CMAKE_BINARY_DIR}/config/config.h)
#设置工程变量
SET(CMAKE_INSTALL_PREFIX /home/evan/local)
```

二. 小结：
1、基本命令：
- Build Targets:
SET()
SUBDIRS()
ADD_LIBRARY()
ADD_EXECUTABLE()
PROJECT()

- Build Flags and Options:
INCLUDE_DIRECTORIES()
LINK_DIRECTORIES()
TARGET_LINK_LIBRARIES()

- Flow Control Constructs
IF 的例子：
```
IF(UNIX)
IF(APPLE)
    SET(GUI "Cocoa"
ELSE(APPLE)
    SET(GUI "X11"
ENDIF(APPLE)
ELSE(UNIX)
IF(WIN32)
    SET(GUI "Win32"
ELSE(WIN32)
    SET(GUI "Unknown"
ENDIF(WIN32)
ENDIF(UNIX)
```

MESSAGE 的例子：
```
MESSAGE("GUI system is ${GUI}")
```

FOREACH 例子：就是可以简化对多个命令的调用，相当于 bash 的 for 语句。
```
SET(SOURCES source1 source2 source3)
FOREACH(source ${SOURCES})
ADD_EXECUTABLE(${source} ${source}.c)
ENDFOREACH(source)
```

- Useful Variables:
最常用的：
CMAKE_BINARY_DIR: build 的 top 目录
CMAKE_SOURCE_DIR: src 的 top 目录
CMAKE_INSTALL_PREFIX: 安装路径的前缀
CMAKE_BUILD_TYPE: 如 Release, Debug 等。
EXECUTABLE_OUTPUT_PATH:
LIBRARY_OUTPUT_PATH:

见 CMake Useful Variables 文档（html)

2、条件编译：
(1)给用户提供选项：
3 steps：

* 取好宏的名字，比如 LINK_LIB1
* 在 top 的 CMakeLists.txt 中写上 OPTION(LINK_LIB1 "Using Lib1" ON)命令。当然写在 top 的 CMakeLists.txt 是因为为了方便查看。
* 在用宏的文件的 CMakeLists.txt 中写上

```
IF(LINK_LIB1)
    SET_SOURCE_FILES_PROPERTIES(main.cpp COMPILE_FLAGS -DLINK_LIB1)
ENDIF(LINK_LIB1)
```

其中，SET_SOURCE_.... 是对 main.cpp 这个文件，设置其编译选项。
你也可以用 ADD_DEFINITIONS(-DLINK_LIB1)，这样的话，所有的文件在编译时都会加上这个选项。
就这三步就好了！！！
然后，想控制选项的打开关闭，可以在命令行敲:
cmake -DLINK_LIB1:BOOL=OFF src/ (默认是打开，因为你写了个 ON 在那里)

其他辅助的命令: （但这个不常用，而是用 cmake 提供的 modules 代替，如 CheckIncludeFile.cmake，可见 man）

```
FIND_PATH(PYTHON_INCLUDE_PATH Python.h
/usr/include
/usr/local/include)
```

表明你取了个变量名：PYTHON_INCLUDE_PATH，用于存放搜索 Python.h 的结果。
(2)检测不同平台，头文件，库等:
不同平台：IF(WIN32) IF(UNIX) 都是预先定义好的。可直接使用。
头文件： 这个可加载 cmake 的 modules:

```
INCLUDE(${CMAKE_ROOT}/Modules/CheckIncludeFiles.cmake)
```

然后使用时，命令名和那个 modules 的名字一样，HAVE_UNISTD_H 为自己定义的变量

```
CHECK_INCLUDE_FILES("unistd.h" HAVE_UNISTD_H)
```

之后，就可以向编译器传递变量 HAVE_UNISTD_H 了，如，这样使用：

```
IF(HAVE_UNISTD_H)
    ADD_DEFINITIONS(-DHAVE_UNISTD_H)
ENDIF(HAVE_UNISTD_H)
```

当然，你的源文件需要含有检测 HAVE_UNISTD_H 的宏。
库：其他的检测，如库，函数，符号等存在与否，都可以加载相应的 modules，见 CMake: How To Write Platform Checks.末尾。
(3)更好的办法:
看看上面(1)(2)两点，他们提供的特性，就是 autotools 提供的两大基本特性。但为了完成这种条件编译，方法是给编译器传递-D 标志(-D 标志这种特性为大多数编译器所支持）。而还有一种方法，是 autotools 用的，就是生成一个 config.h 文件，然后让需要条件编译的源文件包含这个头文件。即，所有-D 标志，现在在 config.h 中了，不用向编译器传递了。
这样，就不要用 ADD_DEFINITIONS()之类的命令了，cmake 会根据 OPTION()，CHECK_INCLUDE_FILES()等命令，自动填好 config.h。
方法如下：
* 自己找个地方，写 config.h 文件（任何名字都行），里面内容：

#ifndef CONFIG_H
#define CONFIG_H
#cmakedefine LINK_LIB1
#endif
其中，LINK_LIB1 会被自动换为如 #define LINK_LIB1 或 #undefine ..
当然，这个头文件，你需要添加到 INCLUDE_DIRECTORIES() 中去。
* 在 CMakeLists.txt（我的是在 top 的那个里面）加上 CONFIGURE_FILE()命令，见 man。
只是注意，两个路径，都要是 full path。第一个路径为包括 config.h.in 的全路径，第二个路径为生成的 config.h 的全路径，一般习惯是放在你的 build 目录下去。所以，第一个路径可以使用系统预定义的变量：CMAKE_SOURCE_DIR 来代替你的 src 的全路径，再加上你的子路径如/config/config.h.in 即可，第二个路径可以使用 CMAKE_BINARY_DIR，指代的是 build 的全路径，再加上你的子路径，如 /config/config.h 即可。当然，这第二个路径要放到 INCLUDE_DIRECTORIES()，才能让你的程序使用到。
* 这样就可以了！！如 OPTION 里面定义的值，只要你写到 config.h 中去了，就会被 CONFIGURE_FILE()命令转换为正常的#define 或#undefine.
其他的 IF(LINK_LIB1)等，仍然可以使用，这样，你可以根据用户的选择，来用 TARGET_LINK_LIBRARIES()链接不同的库。

注：当然，如果是用 OPTION()命令接受用户的选择，那么用户只能在 cmake -DLINK_LIB1:BOOL=OFF src/ 这里设置。也就是说，如果你把这个包给用户的话，用户要熟悉用 cmake 这样来设置。或者使用你写的或别人写的 wrapper，如 ccmake.

3. 如何在使用一个库时，加上-I, -L, -l 和 -D
-I, -L, -l 和-D 是传递给编译器的主要参数，意义很明显。
cmake 提供的众多 module 中，主要有两个可以干这事情：

(1)UsePkgConfig 模块： 背后使用的是系统的 pkg-config
pkg-config 的好处，太明显了，如编译和链接 gtk 程序：
gcc helloworld.c `pkg-config --cflags --libs gtk+-2.0`
它的原理就是把 prefix/lib/pkgconfig 目录下的，你指定名字的 xx.pc 文件中的信息读出来给你，你要--libs，就给你-l，查看.pc 就知道了。得到-I 通过--cflags，要链接的库-l 通过--libs，省得你手敲的麻烦，你不信，试试 pkg-config 命令，它找了一大堆头文件路径和库名出来。
cmake 提供了 module UsePkgConfig 来对 pkg-config 支持，具体可 man，如编译链接 gtk 可这样：

假设编译的执行文件是 helloworld
ADD_EXECUTABLE(helloworld b.c)

加上 gtk 的支持可以这样：
INCLUDE(UsePkgConfig)

PKGCONFIG(gtk+-2.0 includedir libdir linkflags cflags)

然后，PKGCONFIG 中的变量就被填上了，就可以用 cmake 告知编译器参数的如下四条命令：
INCLUDE_DIRECTORIES(${includedir}) #-I。
LINK_DIRECTORIES(${libdir}) #-L
TARGET_LINK_LIBRARIES(helloworld ${linkflags}) #-l
ADD_DEFINITIONS(${cflags}) #-D

注意：
a. INCLUDE 和 INCLUDE_DIRECTORIES 是不一样的，前者是 cmake 用来包含入其他的 cmake lists 文件或是 cmake 的模块文件，而后者是用于传给编译器头文件路径的参数。
b. 由于 wxWidgets 不采用普通的 pkg-config，而是自己带了一个 wx-config，所以，没有普通的 xx.pc 文件可供 UsePkgConfig 使用，所以，应该采取 FIND_PACKAGE 模块，见下。

(2)FIND_PACKAGE
如你写 FIND_PACKAGE(wxWidgets)，它就去调用 module 中的 FindwxWidgets.cmake 这个模块，然后你可以得到几个变量，man 下，并搜索 FindwxWidgets，就知道可以得到哪些变量，就知道怎么用了。这个 FindwxWidgets.cmake 就是使用了 wx-config！而不是 pkg-config，所以，看起来 FIND_PACKAGE 比 UsePkgConfig 灵活。

基本模式是：
FIND_PACKAGE(wxWidgets REQUIRED)
IF(wxWidgets_FOUND)
    INCLUDE(${wxWidgets_USE_FILE})#这个变量是便利的-D -I... 如果要分开，看 man 吧。
    TARGET_LINK_LIBRARIES(rose ${wxWidgets_LIBRARIES})
ENDIF(wxWidgets_FOUND)

4. INSTALL 命令：
这个命令可以使得 cmake 生成的 Makefile 文件含有 install 目标。
- INSTALL(FILES 源文件,资源文件等 DESTINATION 路径)
- INSTALL(PROGRAMS 脚本等非二进制的但可执行文件 DESTINATION 路径)
- INSTALL(TARGETS 二进制程序 动态库 静态库
    RUNTIME DESTINATION 路径
    LIBRARY DESTINATION 路径
    ARCHIVE DESTINATION 路径)
例子：
INSTALL(TARGETS myExe mySharedLib myStaticLib
    RUNTIME DESTINATION bin
    LIBRARY DESTINATION lib

ARCHIVE DESTINATION lib/static)
意思是：这里有一些 Targets，然后 cmake 会自动检查是属于 RUNTIME，LIBRARY 还是 ARCHIVE，然后执行按 RUNTIME/LIBRARY/ARCHIVE 后面的属性来。
- INSTALL(DIRECTORY .....)
这个比较方便的安装整个目录，比如资源文件目录等，很方便。暂时不用。

注意：
(1)路径一般填写相对路径: 即，之前你应该填好系统预定义的变量：
CMAKE_INSTALL_PREFIX，然后，比如你写 bin，则就装到
${CMAKE_INSTALL_PREFIX}/bin 下了。如果你在这里写的是绝对路径（在 linux 下即以/开头），则就按绝对路径处理了。
(2)对 RUNTIME/LIBRARY/ARCHIVE 的说明：
RUNTIME: 可执行文件，DLL 形式动态库的 DLL 部分
LIBRARY: 模块库，非 DLL 形式的动态库
ARCHIVE: 静态库，DLL 形式动态库的 Import Lib 部分。
(3)装好的库，要让工具找到：
动态库：将你装的路径放入/etc/ld.so.conf（man ldconfig 就知道了），然后 ldconfig 一下。程序运行时才能够动态链接上。但它不影响编译链接时！！，如果不在 cmake 中指定链接目录，我还不知道怎么做。
静态库：暂时不知道怎样调整 gcc 的默认链接目录。
(4)PERMISSIONS，你要加的话可以加。

5. 交叉编译：
从 cmake 2.6 开始支持，我的系统上暂时是 2.4。
见 CMake Cross Compiling.htm。比如你装了 mingw，然后就有一些编译和链接等工具如 i386-mingw-gcc 等，就可以使用 cmake 2.6 的一些命令，来指定这些工具，当然还有一些所需要的头文件和库，都可以用它来指定。
暂不看它。

6. 和 make 的区别：
(1)命令行参数写法：
make 的参数写法 make CXXFLAGS="-Dxx -Ixx"
cmake 的参数写法 cmake -Dxx:xx=xx -Ixxx

说明：关于 autotools 工具链： autoconf, automake & libtool 不想深究。
只想了解到这种地步：

1. autoconf: 提供了对不同平台做检测（库，头文件，编译器等），和提供选项给用户进行按需编译这两项主要功能，来使得生产 config.h 文件。用户的程序需要包含这个文件，并自己设置#ifdef 这类的宏来准备条件编译，这样，不同平台上的编译，就可以“自动的”（auto）进行了。
用例:
（1）不同平台做检测（库，头文件，编译器等）：显然可能存在好多可用的功能相同的库，也可能某个平台上暂时没有装库，那么用户可以要求 configure 去检测，并把检测结果放入 config.h，然后用户的源代码里因为已经预备好了条件编译，就可以根据实际情况“自动的”进行编译。
（2）提供选项给用户进行按需编译：显然用户可能不想编译所有的组件，他可以选择一些功能编译进去，这样，开发者就需要自己写些 m4 宏（就是 shell 脚本），然后加入 autoconf，这样最后生成的 configure 文件，就可以接可以收用户的选项了。
可以看出，这两个主要功能都是和“条件编译”相结合的。
2. automake：这个就是为了简化 Makefile 的编写，你只需要写 Makefile.am，这个的语法相对于 Makefile 简单一点。
3. libtool：这个目的是提供一个统一的命令行接口，来在不同的平台编译出“静态和动态库”，这个工具确实有用，因为你要认识到，不同的编译器，相同编译器的不同版本，不同平台在安装库后需要做的配置等等，都是不同的。

cmake 的好处就是：不用学这么多的工具，他就是一个工具，用简单的语法提供相当多的特性。而且，速度快得多，生成 makfile 文件小得多