cmake 学习笔记(一) - 1+1=10 - 博客频道

分类：

tools（ *32* ）

cmake/qmake（ *16* ）

目录(?)[+]

- 最大的Qt4程序群(KDE4)采用cmake作为构建系统
- Qt4的python绑定(pyside)采用了cmake作为构建系统
- 开源的图像处理库 opencv 采用cmake 作为构建系统
- ...

看来不学习一下cmake是不行了，一点一点来吧，找个最简单的C程序，慢慢复杂化，试试看：

例子一	单个源文件 main.c
例子二	==>分解成多个 main.c hello.h hello.c
例子三	==>先生成一个静态库，链接该库
例子四	==>将源文件放置到不同的目录
例子五	==>控制生成的程序和库所在的目录
例子六	==>使用动态库而不是静态库

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？

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。

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

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。

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

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。

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")
```

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。

顶
20