CMake的命令行用法

CMake的命令行用法比GUI的用法复杂，但是功能更加强大，值得一学。以下是CMake命令行调用的方法：

```
1 cmake [<options>] (<path-to-source> | <path-to-existing-build>)
2 cmake [(-D<var>=<value>)...] -P <cmake-script-file>
3 cmake --build <dir> [<options>] [-- <build-tool-options>...]
4 cmake -E <command> [<options>...]
5 cmake --find-package <options>...
```

生成编译工程文件

cmake <CMakeLists.txt_Path>就是生成可以编译工程文件。当时运行的目录在哪里，生成的可编译工程文件就在哪个目录。比如CMakeLists.txt文件在f:cmake目录，而当时在f:cmakebuild目录运行cmake ..，则生成的编译工程文件在f:cmakebuild目录。

也可以再生成编译工程文件的时候通过-D来添加变量值，比如CMakeLists.txt内容如下：

```
1 cmake_minimum_required (VERSION 2.6)
2 project (a)
3 message(${EXECUTABLE_OUTPUT_PATH})
4 add_executable(b b.cpp)
```

我们可以通过-D选择来设置EXECUTABLE_OUTPUT_PATH的值，也是编译的文件的输出目录：

```
1 cmake -D EXECUTABLE_OUTPUT_PATH="another_output" ..
```

这样，我们就给CMakeLists.txt编译脚本传递了新的EXECUTABLE_OUTPUT_PATH值。

编译工程

CMake除了生成编译工程文件，它也可以调用系统的编译工程来编译工程，如：

```
1 cmake --build .
```

默认是编译debug模式，也可以传递在–后面传递MSBUILD参数来控制：

```
1 cmake --build . -- /p:Configuration=Release
```

命令行工具模式

CMake有一个-E的命令行工具模式，提供了一些常用的功能，比复制文件、创建目录、读写注册表、读写环境变量、计算md5值等等。脚本可以调用这些功能。

编写CMakeLists.txt

创建可以执行程序工程

首先从创建一个最简单的可执行程序开始，CMakeLists.txt内容如下：

```
1
2
3
4
5
cmake_minimum_required (VERSION 2.6)
project (LearnCMake)
message(${LearnCMake_SOURCE_DIR})
message(${LearnCMake_BINARY_DIR})
add_executable(FirstExecutable hello_world.cpp)
```

第1行是cmake需要的最低版本，目前这个是VERSION 2.6，一般不用修改。

第2~4行表示我们创建了一个名为LearnCMake工程，对应生成一个LearnCMake.sln。project函数表示创建一个工程。同时，也生成了4个变量：

- PROJECT_SOURCE_DIR, <PROJECT-NAME>_SOURCE_DIR。工程的源代码目录。
- PROJECT_BINARY_DIR, <PROJECT-NAME>_BINARY_DIR。工程的二进制文件目录。

第5行表示添加一个名为FirstExecutable的可执行程序项目，它的源代码为hello_world.cpp。下面是add_executable的完整用法：

```
1
2
3
add_executable(<name> [WIN32] [MACOSX_BUNDLE]
        [EXCLUDE_FROM_ALL]
        source1 [source2 ...])
```

默认的是创建控制台工程，加上WIN32表示创建的是win32工程，如下：

```
1
add_executable(FirstExecutable WIN32 hello_world.cpp)
```

后面是项目的代码，可以添加多个代码文件，用空格分开。

创建库工程

创建库工程跟创建可执行程序工程类似，创建库工程使用add_library函数，如下例子：

```
1
2
3
4
5
6
cmake_minimum_required (VERSION 3.0)
project (LearnCMake)
add_library(FirstLibrary first_library.cpp)
add_library(SecondLibrary second_library.cpp)
add_executable(FirstExecutable hello_world.cpp)
target_link_libraries(FirstExecutable FirstLibrary)
```

add_library的用法如下：

```
1
```

```
2
3
add_library(<name> [STATIC | SHARED | MODULE]
            [EXCLUDE_FROM_ALL]
            source1 [source2 ...])
```

默认的是静态库，也可以显式的设置库是否为静态库、动态库或者是模块。另外BUILD_SHARED_LIBS也可控制编译成哪种库。

target_link_libraries用来链接库，用法如下：

```
1
2
target_link_libraries(<target> [item1 [item2 [...]]]
                      [[debug|optimized|general] <item>] ...)
```

set设置变量

add_library、add_executable都可以添加多个源文件，如下：

```
1
2
3
4
cmake_minimum_required (VERSION 3.0)
project (LearnCMake)
add_executable(FirstExecutable main.cpp app_util.h app_util.cpp)
add_library(FirstLibrary app_util.h app_util.cpp)
```

我们可以通过定义一个AppUtilSrc变量来代替app_util.h app_util.cpp，如下：

```
1
2
3
4
5
cmake_minimum_required (VERSION 3.0)
project (LearnCMake)
set(AppUtilSrcs app_util.h app_util.cpp)
add_executable(FirstExecutable main.cpp ${AppUtilSrcs})
add_library(FirstLibrary ${AppUtilSrcs})
```

效果是跟上面等价的。还可以累积值：

```
1
2
set(AppUtilSrcs app_util.h app_util.cpp)
set(AppUtilSrcs ${AppUtilSrcs} b.cpp)
```

这样AppUtilSrcs就代表着3个文件了。

设置编译模式

设置mt编译模式：

```
1
2
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE}
/MT")
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG}
/MTd")
```

设置md编译模式：

```
set(CMAKE_CXX_FLAGS_RELEASE "${CMAKE_CXX_FLAGS_RELEASE} /MD")
set(CMAKE_CXX_FLAGS_DEBUG "${CMAKE_CXX_FLAGS_DEBUG} /MDd")
```

默认是多字节模式，设置成unicode模式：

```
add_definitions(-D_UNICODE)
```

另外add_definitions还可以设置其他的选项。

添加其他CMakeLists.txt

一个CMakeLists.txt里面的target如果要链接其他CMakeLists.txt中的target，可以使用add_subdirectory，我们以使用googletest库为例：

```
add_subdirectory("../thirdparty/googletest/googletest/" gtest)
file(GLOB_RECURSE gtest_lib_head_files "../thirdparty/googletest/googletest/*.h")
source_group("gtest" FILES ${gtest_lib_head_files})
include_directories("../thirdparty/googletest/googletest/include")
aux_source_directory("./pbase_unittest/src" pbase_unittest_src_files)
file(GLOB_RECURSE pbase_unittest_include_files "./pbase_unittest/include/*.h")
add_executable(pbase_unittest ${pbase_unittest_src_files}${pbase_unittest_include_files} ${gtest_lib_head_files})
target_link_libraries(pbase_unittest gtest)
```

代码控制

如果想把./pbase/src目录下的所有源文件加入到工程，可以用aux_source_directory把某个目录下的源文件加入到某个变量中，稍后就可以使用这个变量代表的代码了，如：

```
aux_source_directory("./pbase/src" pbase_lib_src_files)
add_library(pbase ${pbase_lib_src_files})
```

添加头文件包含目录是：

```
include_directories("../thirdparty/googletest/googletest/include")
```

但是include_directories中的文件不会体现先visual studio工程中，而aux_source_directory只会添加源文件，会忽略头文件，如果想生存的visual studio工程里面也包含头文件，可以这样：

```
# add head files
file(GLOB_RECURSE pbase_lib_head_files "./pbase/include/*.h")
add_library(pbase ${pbase_lib_head_files})
```

如果想生存visual studio中的filter，可以使用source_group：

```
file(GLOB_RECURSE gtest_lib_head_files "../thirdparty/googletest/googletest/*.h")
source_group("gtest" FILES ${gtest_lib_head_files})
```

最终添加头文件到工程里标准模板是：

```
file(GLOB_RECURSE gtest_lib_head_files "../thirdparty/googletest/googletest/*.h")
source_group("gtest" FILES ${gtest_lib_head_files})
include_directories("../thirdparty/googletest/googletest/include")
add_executable(pbase_unittest ${gtest_lib_head_files})
```