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

- $ export VERBOSE=1
- $ make

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。

在 Cmake 学习笔记一 中通过一串小例子简单学习了 cmake 的使用方式。
这次应该简单看看语法和常用的命令了。

简单的语法

- 注释

```
# 我是注释
```

- 命令语法

```
COMMAND(参数 1 参数 2 ...)
```

- 字符串列表

```
A;B;C # 分号分割或空格分隔的值
```

- 变量(字符串或字符串列表)

set(Foo a b c)	设置变量 Foo
command(${Foo})	等价于 command(a b c)
command("${Foo}")	等价于 command("a b c")
command("/${Foo}")	转义，和 a b c 无关联

- 流控制结构

```
IF()...ELSE()/ELSEIF()...ENDIF()
WHILE()...ENDWHILE()
FOREACH()...ENDFOREACH()
```

- 正则表达式

部分常用命令

INCLUDE_DIRECTORIES("dir1" "dir2" ...)	头文件路径，相当于编译器参数 -Idir1 -Idir2
LINK_DIRECTORIES("dir1" "dir2")	库文件路径。注意：

	由于历史原因，相对路径会原样传递给链接器。 尽量使用 FIND_LIBRARY 而避免使用这个。
AUX_SOURCE_DIRECTORY(“sourcedir” variable)	收集目录中的文件名并赋值给变量
ADD_EXECUTABLE	可执行程序目标
ADD_LIBRARY	库目标
ADD_CUSTOM_TARGET	自定义目标
ADD_DEPENDENCIES(target1 t2 t3)	目标 target1 依赖于 t2 t3
ADD_DEFINITIONS("-Wall -ansi")	本意是供设置 -D... /D... 等编译预处理需要的宏定义参数，对比 REMOVE_DEFINITIONS()
TARGET_LINK_LIBRARIES(target-name lib1 lib2 ...)	设置单个目标需要链接的库
LINK_LIBRARIES(lib1 lib2 ...)	设置所有目标需要链接的库
SET_TARGET_PROPERTIES(...)	设置目标的属性 OUTPUT_NAME, VERSION, ...
MESSAGE(...)	
INSTALL(FILES “f1” “f2”DESTINATION .)	DESTINATION 相对于 ${CMAKE_INSTALL_PREFIX}
SET(VAR value [CACHE TYPE DOCSTRING [FORCE]])	
LIST(APPEND|INSERT|LENGTH|GET| REMOVE_ITEM|REMOVE_AT|SORT ...)	列表操作
STRING(TOUPPER|TOLOWER|LENGTH| SUBSTRING|REPLACE|REGEX ...)	字符串操作
SEPARATE_ARGUMENTS(VAR)	转换空格分隔的字符串到列表
FILE(WRITE|READ|APPEND|GLOB| GLOB_RECURSE|REMOVE|MAKE_DIRECTORY ...)	文件操作
FIND_FILE	注意 CMAKE_INCLUDE_PATH
FIND_PATH	注意 CMAKE_INCLUDE_PATH
FIND_LIBRARY	注意 CMAKE_LIBRARY_PATH
FIND_PROGRAM	
FIND_PACKAGE	注意 CMAKE_MODULE_PATH
EXEC_PROGRAM(bin [work_dir] ARGS <..> [OUTPUT_VARIABLE var] [RETURN_VALUE var])	执行外部程序

OPTION(OPTION_VAR “description” [initial value])	

变量

工程路径

- CMAKE_SOURCE_DIR
- PROJECT_SOURCE_DIR
- <projectname>_SOURCE_DIR

这三个变量指代的内容是一致的，是工程顶层目录

- CMAKE_BINARY_DIR
- PROJECT_BINARY_DIR
- <projectname>_BINARY_DIR

这三个变量指代的内容是一致的，如果是 in source 编译，指得就是工程顶层目录，如果 是 out-of-source 编译，指的是工程编译发生的目录

- **CMAKE_CURRENT_SOURCE_DIR**

指的是当前处理的 CMakeLists.txt 所在的路径。

- **CMAKE_CURRRENT_BINARY_DIR**

如果是 in-source 编译，它跟 CMAKE_CURRENT_SOURCE_DIR 一致，如果是 out-ofsource 编译，他指的是 target 编译目录。

- **CMAKE_CURRENT_LIST_FILE**

输出调用这个变量的 CMakeLists.txt 的完整路径

CMAKE_BUILD_TYPE

控制 Debug 和 Release 模式的构建

- CMakeList.txt 文件

SET(CMAKE_BUILD_TYPE Debug)

- 命令行参数

cmake DCMAKE_BUILD_TYPE=Release

编译器参数

- CMAKE_C_FLAGS
- CMAKE_CXX_FLAGS

也可以通过指令 ADD_DEFINITIONS()添加

CMAKE_INCLUDE_PATH

- 配合 FIND_FILE() 以及 FIND_PATH() 使用。如果头文件没有存放在常规路径 (/usr/include, /usr/local/include 等)，

则可以通过这些变量就行弥补。如果不使用 FIND_FILE 和 FIND_PATH 的话，CMAKE_INCLUDE_PATH，没有任何作用。

- **CMAKE_LIBRARY_PATH**

配合 FIND_LIBRARY() 使用。否则没有任何作用

- **CMAKE_MODULE_PATH**

cmake 为上百个软件包提供了查找器(finder):FindXXXX.cmake

当使用非 cmake 自带的 finder 时，需要指定 finder 的路径，这就是 CMAKE_MODULE_PATH，配合 FIND_PACKAGE()使用

CMAKE_INSTALL_PREFIX

控制 make install 是文件会安装到什么地方。默认定义是/usr/local 或 %PROGRAMFILES%

BUILD_SHARED_LIBS

如果不进行设置，使用 ADD_LIBRARY 且没有指定库类型，默认编译生成的库是静态库。

UNIX 与 WIN32

- UNIX，在所有的类 UNIX 平台为 TRUE，包括 OS X 和 cygwin
- WIN32，在所有的 win32 平台为 TRUE，包括 cygwin

学习一下 cmake 的 finder。

finder 是神马东西？

当编译一个需要使用第三方库的软件时，我们需要知道：

去哪儿找头文件 .h	对比 GCC 的 -I 参数
去哪儿找库文件 (.so/.dll/.lib/.dylib/...)	对比 GCC 的 -L 参数
需要链接的库文件的名字	对比 GCC 的 -l 参数

这也是一个 finder 需要返回的最基本的信息。

如何使用?

比如说，我们需要一个第三方库 curl，那么我们的 CMakeLists.txt 需要指定头文件目录，和库文件，类似：

```
include_directiories(/usr/include)
target_link_libraries(myprogram curl)
```

如果借助于 cmake 提供的 finder 会怎么样呢？使用 cmake 的 Modules 目录下的 FindCURL.cmake，相应的 CMakeList.txt 文件：

```
find_package(CURL REQUIRED)
include_directories(${CURL_INCLUDE_DIR})
target_link_libraries(curltest ${CURL_LIBRARY})
```

或者

```
find_package(CURL)
if(CURL_FOUND)
include_directories(${CURL_INCLUDE_DIR})
target_link_libraries(curltest ${CURL_LIBRARY})
else(CURL_FOUND)
message(FATAL_ERROR "curl not found!")
endif(CURL_FOUND)
```

如果我们使用的 finder，不是 cmake 自带的怎么办？

- 放置位置：工程根目录下的 cmake/Modules/
- 然后在 CMakeList.txt 中添加

```
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH}
"${CMAKE_SOURCE_DIR}/cmake/Modules/")
```

find_package 如何工作

find_package 将会在 module 路径下查找 Find<name>.cmake。首先它搜索 ${CMAKE_MODULE_PATH} 中的所有路径，然后搜索 <CMAKE_ROOT>/share/cmake-x.y/Modules/

如果这个文件未找到，它将会查找 <Name>Config.cmake 或 <lower-case-name>-config.cmake 文件。这两个文件是库文件安装时自己安装的，将自己的路径硬编码到其中。

前者称为 module 模式，后者称为 config 模式

每个模块一般都会提供一下几个变量

- <name>_FOUND
- <name>_INCLUDE_DIR 或 <name>_INCLUDES
- <name>_LIBRARY 或 <name>_LIBRARIES 或 <name>_LIBS
- <name>_DEFINITIONS

编写 finder

- 首先使用 find_package 探测本软件包依赖的第三方库(参数 QUIETLY 和 REQUIRED 应该被传递)
- 如果 pkg-config 可用，则可以用其去探测 include/library 路径
- 分别使用 find_path 和 find_library 查找头文件和库文件
    - pkg-config 提供的路径仅作为参考
    - CMake 有很多硬编码的路径
    - 结果放到 <name>_INCLUDE_DIR 和 <name>_LIBRARY (**注意：**单数而不是复数)
- 设置 <name>_INCLUDE_DIRS 为 <name>_INCLUDE_DIR <dependency1>_INCLUDE_DIRS ...
- 设置 <name>_LIBRARIES 为 <name>_LIBRARY <dependency1>_LIBRARIES ...
    - 依赖使用复数，包自身使用单数形式（由 find_path 和 find_library 提供）
- 调用宏 find_package_handle_standard_args() 设置 <name>_FOUND 并打印或失败信息

接前面的一二三，学习一下 CMakeCache.txt 相关的东西。

CMakeCache.txt

可以将其想象成一个配置文件(在 Unix 环境下，我们可以认为它等价于传递给 configure 的参数)。

- CMakeLists.txt 中通过 set(... CACHE ...) 设置的变量
- CMakeLists.txt 中的 option() 提供的选项
- CMakeLists.txt 中 find_package() 等 find 命令引入变量
- 命令行 cmake . -D <var>:<type>=<value> 定义变量

cmake 第一次运行时将生成 CMakeCache.txt 文件，我们可以通过 ccmake 或 cmake-gui 或 make edit_cache 对其进行编辑。

对应于命令行 -D 定义变量，-U 用来删除变量（支持 globbing_expr），比如 cmake -U/*QT/* 将删除所有名字中带有 QT 的 cache 项。

变量与 Cache

cmake 的变量系统远比第一眼看上去复杂:

- 有些变量被 cache，有些则不被 cache
- 被 cache 的变量
    - 有的不能通过 ccmake 等进行编辑(internal)
    - 有的(带有描述和类型)可以被编辑(external)
        - 有的只在 ccmake 的 advanced 模式出现

看个例子：

- SET(var1 13)
    - 变量 var1 被设置成 13
    - 如果 var1 在 cache 中已经存在，该命令不会 overwrite cache 中的值
- SET(var1 13 ... CACHE ...)

    - 如果 cache 存在该变量，使用 cache 中变量
    - 如果 cache 中不存在，将该值写入 cache
- SET(var1 13 ... CACHE ... FORCE)
    - 不论 cache 中是否存在，始终使用该值

要习惯用帮助

cmake --help-command SET

find_xxx

为了避免每次运行都要进行头文件和库文件的探测，以及考虑到允许用户通过 ccmake 设置头文件路径和库文件的重要性，这些东西必须进行 cache。

- find_path 和 find_library 会自动 cache 他们的变量，如果变量已经存在且是一个有效值（即不是 -NOTFOUND 或 undefined）,他们将什么都不做。
- 另一方面，模块查找时输出的变量（<name>_FOUND,<name>_INCLUDE_DIRS,<name>_LIBRARIES）不应该被 cache

在 cmake 学习笔记(三) 中简单学习了 find_package 的 model 模式，在 cmake 学习笔记(四)中了解一个 CMakeCache 相关的东西。但靠这些知识还是不能看懂 PySide 使用 CMakeLists 文件，接下来继续学习 find_package 的 config 模式及 package configure 文件相关知识

find_package 的 config 模式

当 CMakeLists.txt 中使用 find_package 命令时，首先启用的是 module 模式：

- 按照 CMAKE_MODULE_PATH 路径和 cmake 的安装路径去搜索 finder 文件 Find<package>.cmake

如果 finder 未找到，则开始 config 模式：

- 将在下列路径下查找 配置 文件 <name>Config.cmake 或 <lower-case-name>-config.cmake

<prefix>/	(W)
<prefix>/(cmake|CMake)/	(W)
<prefix>/<name>*/	(W)
<prefix>/<name>*/(cmake|CMake)/	(W)
<prefix>/(share|lib)/cmake/<name>*/	(U)
<prefix>/(share|lib)/<name>*/	(U)

<prefix>/(share|lib)/<name>*/(cmake|CMake)/	(U)

- find_package 参数及规则见 manual

<name>Config.cmake

该文件至少需提供头文件路径和库文件信息。比如 ApiExtractorConfig.cmake 在 Windows 下一个例子：

```
# - try to find APIEXTRACTOR
#  APIEXTRACTOR_INCLUDE_DIR   - Directories to include to use
APIEXTRACTOR
#  APIEXTRACTOR_LIBRARIES     - Files to link against to use
APIEXTRACTOR

SET(APIEXTRACTOR_INCLUDE_DIR
"D:/shiboken/dist/include/apiextractor")
if(MSVC)
    SET(APIEXTRACTOR_LIBRARY
"D:/shiboken/dist/lib/apiextractor.lib")
elseif(WIN32)
    SET(APIEXTRACTOR_LIBRARY
"D:/shiboken/dist/bin/apiextractor.dll")
else()
    SET(APIEXTRACTOR_LIBRARY
"D:/shiboken/dist/lib/apiextractor.dll")
endif()
```

该文件是通过 configure_file 机制生成的，我们看看 ApiExtractorConfig.cmake.in 文件：

```
SET(APIEXTRACTOR_INCLUDE_DIR
"@CMAKE_INSTALL_PREFIX@/include/apiextractor@apiextractor_SUFFIX@
")
if(MSVC)
```

```
    SET(APIEXTRACTOR_LIBRARY
"@LIB_INSTALL_DIR@/@CMAKE_SHARED_LIBRARY_PREFIX@apiextractor@apie
xtractor_SUFFIX@@LIBRARY_OUTPUT_SUFFIX@.lib")
elseif(WIN32)
    SET(APIEXTRACTOR_LIBRARY
"@CMAKE_INSTALL_PREFIX@/bin/@CMAKE_SHARED_LIBRARY_PREFIX@apiextra
ctor@apiextractor_SUFFIX@@LIBRARY_OUTPUT_SUFFIX@@CMAKE_SHARED_LIB
RARY_SUFFIX@")
else()
    SET(APIEXTRACTOR_LIBRARY
"@LIB_INSTALL_DIR@/@CMAKE_SHARED_LIBRARY_PREFIX@apiextractor@apie
xtractor_SUFFIX@@LIBRARY_OUTPUT_SUFFIX@@CMAKE_SHARED_LIBRARY_SUFF
IX@")
endif()
```

对应的命令(变量的定义略过)

```
configure_file("${CMAKE_CURRENT_SOURCE_DIR}/ApiExtractorConfig.cm
ake.in" "${CMAKE_CURRENT_BINARY_DIR}/ApiExtractorConfig.cmake"
@ONLY)
```

<name>ConfigVersion.cmake

该文件用来比对版本是否匹配，看看 ApiExtractorConfigVersion.cmake.in 的内容：

```
set(PACKAGE_VERSION @apiextractor_VERSION@)

if("${PACKAGE_VERSION}" VERSION_LESS "${PACKAGE_FIND_VERSION}" )
   set(PACKAGE_VERSION_COMPATIBLE FALSE)
else("${PACKAGE_VERSION}" VERSION_LESS "${PACKAGE_FIND_VERSION}" )
   set(PACKAGE_VERSION_COMPATIBLE TRUE)
   if( "${PACKAGE_FIND_VERSION}" STREQUAL "${PACKAGE_VERSION}")
     set(PACKAGE_VERSION_EXACT TRUE)
   endif( "${PACKAGE_FIND_VERSION}" STREQUAL "${PACKAGE_VERSION}")
```

```
endif("${PACKAGE_VERSION}" VERSION_LESS "${PACKAGE_FIND_VERSION}" )
```

一般提供设置下面的变量

PACKAGE_VERSION	完整的版本字符串
PACKAGE_VERSION_EXACT	如果完全匹配为真
PACKAGE_VERSION_COMPATIBLE	如果兼容为真
PACKAGE_VERSION_UNSUITABLE	如果不可用为真

find_package 进而根据这些设置

<package>_VERSION	full provided version string
<package>_VERSION_MAJOR	major version if provided, else 0
<package>_VERSION_MINOR	minor version if provided, else 0
<package>_VERSION_PATCH	patch version if provided, else 0
<package>_VERSION_TWEAK	tweak version if provided, else 0

希望这是现阶段阻碍阅读 shiboken 和 PySide 源码的涉及 cmake 的最后一个障碍 ^_^

学习 cmake 的单元测试部分 ctest。

简单使用

最简单的使用 ctest 的方法，就是在 CMakeLists.txt 添加命令：

```
enable_testing()
```

- 该命令需要在源码的根目录文件内。

从这一刻起，就可以在工程中添加 add_test 命令了

```
add_test(NAME <name> [CONFIGURATIONS [Debug|Release|...]]
         [WORKING_DIRECTORY dir]
```

```
        COMMAND <command> [arg1 [arg2 ...]])
```

- name 指定一个名字
- Debug|Release 控制那种配置下生效
- dir 设置工作目录
- command
    - 如果是可执行程序目标，则会被 cmake 替换成生成的程序的全路径
    - 后面的参数可以使用 $<...> 这种语法，比如 $<TARGET_FILE:tgt> 指代 tgt 这个目标的全名

ApiExtractor

继续以 ApiExtractor 为例学习 ctest 的使用

顶层的 CMakeLists.txt 文件的内容片段：

```
option(BUILD_TESTS "Build tests." TRUE)
if (BUILD_TESTS)
    enable_testing()
    add_subdirectory(tests)
endif()
```

创建选项，让用户控制是否启用单元测试。如果启用，则添加进 tests 子目录，我们看其 CMakeLists.txt 文件

- 首先是创建一个 declare_test 的宏
    - 使用 qt4_automoc 进行 moc 处理
    - 生成可执行文件
    - 调用 add_test 加入测试

```
macro(declare_test testname)
    qt4_automoc("${testname}.cpp")
    add_executable(${testname} "${testname}.cpp")
    include_directories(${CMAKE_CURRENT_SOURCE_DIR}
${CMAKE_CURRENT_BINARY_DIR} ${apiextractor_SOURCE_DIR})
```

```
    target_link_libraries(${testname} ${QT_QTTEST_LIBRARY}
${QT_QTCORE_LIBRARY} ${QT_QTGUI_LIBRARY} apiextractor)
    add_test(${testname} ${testname})
endmacro(declare_test testname)
```

- 后续就简单了,需要的配置文件直接使用 configure_file 的 COPYONLY

```
declare_test(testabstractmetaclass)
declare_test(testabstractmetatype)
declare_test(testaddfunction)
declare_test(testarrayargument)
declare_test(testcodeinjection)
configure_file("${CMAKE_CURRENT_SOURCE_DIR}/utf8code.txt"
               "${CMAKE_CURRENT_BINARY_DIR}/utf8code.txt" COPYONLY)
declare_test(testcontainer)
```

Qt 单元测试

QTestLib 模块用起来还是很简单的，我们这儿稍微一下 cmake 和 qmake 的一点不同。

- 使用 qmake 时，我们只需要一个源文件，比如测试 QString 类时，写一个 testqstring.cpp 文件

```
#include <QtTest/QtTest>

class TestQString: public QObject
{
    Q_OBJECT
private slots:
    void toUpper();
};

void TestQString::toUpper()
{
    QString str = "Hello";
```

```
    QCOMPARE(str.toUpper(), QString("HELLO"));
 }

 QTEST_MAIN(TestQString)
 #include "testqstring.moc"
```

然后 pro 文件内启用 testlib 模块，其他和普通 Qt 程序一样了。

- 使用 cmake 时，我们将其分成两个文件

```
//testqstring.h
 #include <QtTest/QtTest>

 class TestQString: public QObject
 {
    Q_OBJECT
 private slots:
    void toUpper();
 };
```

与

```
//testqstring.cpp
 void TestQString::toUpper()
 {
    QString str = "Hello";
    QCOMPARE(str.toUpper(), QString("HELLO"));
 }

 QTEST_MAIN(TestQString)
 #include "testqstring.moc"
```

然后处理方式就是我们前面看到的那个宏了。

QTest 宏

随便看下 QTest 的宏

- QTEST_APPLESS_MAIN
- QTEST_NOOP_MAIN
- QTEST_MAIN

```
#define QTEST_APPLESS_MAIN(TestObject) /
int main(int argc, char *argv[]) /
{ /
    TestObject tc; /
    return QTest::qExec(&tc, argc, argv); /
}

#define QTEST_NOOP_MAIN /
int main(int argc, char *argv[]) /
{ /
    QObject tc; /
    return QTest::qExec(&tc, argc, argv); /
}

#define QTEST_MAIN(TestObject) /
int main(int argc, char *argv[]) /
{ /
    QCoreApplication app(argc, argv); /
    TestObject tc; /
    return QTest::qExec(&tc, argc, argv); /
}
```

最终都是调用 QTest::qExec,Manual 中对其有不少介绍了(略)。