Modern CMake文档

- 安装
  - Windows安装
    - 网址：https://cmake.org/download/
  - macOS安装
    - 网址：https://cmake.org/download/，下载CMake，并正常安装
    - 安装完成之后，使用以下指令创建/usr/local/bin下的CMake的软连接
      - sudo "/Applications/CMake.app/Contents/bin/cmake-gui" --install
      - 注意：执行此命令的时候确保CMake处于关闭状态
    - 重新打开Terminal，即可正常使用 CMake 的各种指令了，也可以在应用程序列表中使用带 GUI 的 CMake 工具。
  - Linux安装
    - 网址：https://cmake.org/download/，下载对应版本的CMake（32位或者64位）
    - 将下载的安装包上传到Linux服务器，比如:/root
    - 输入以下命令进行解压
      - tar -zxvf cmake-3.13.0-rc1-Linux-x86_64.tar.gz
      - 注意：后面是官网下载的对应版本的名字
    - 把解压后的目录改名为：cmake
      - mv cmake-3.10.0-rc4-Linux-x86_64 cmake
    - 设置环境变量
      - 使用指令“vi .bash_profile”来设置环境变量，找到PATH=$PATH:$....这一行，后面添加CMake安装目录里面的bin目录的地址
      - 如果是在/root目录安装的CMake，那添加的目录就是：/root/cmake/bin
    - 安装完毕，环境变量设置成功之后，命令行输入：cmake --version检测是否安装成功
      - 输出：cmake version 3.13，表示安装成功
- 使用CMake生成项目
  - 使用Windows或者Linux生成项目
    - 进入项目目录（CMakeLists.txt所在目录），新建一个build文件夹，因为CMake会产生很多自己的中间文件。
    - 执行： cmake ../　就会在build目录产生项目文件，windows下面默认产生vs的项目。
    - 如果要产生其他编译器的makefile，就需要使用-G指定编译器
      - cmake -G "MinGW Makefiles" ../
    - 可以使用cmake --help 来查看使用的编译器的名字
    - 生成项目工程文件或者makefile之后，就可以使用对应的编译器来编译项目
  - 使用macOS生成项目

        - mac下基本操作和windows、Linux相同，不过cmake命令使用的是：cmake .. （没有右斜杠）
    - 注意：（默认已经配置好环境变量）
- CMake命令行选项的设置
    - 指定构建系统生成器：-G
        - 使用：-G 命令可以指定编译器，当前平台支持的编译器名称可以通过帮助手册查看：cmake --help，

          例如：cmake -G "Visual Studio 15 2017" ../ 使用vs2017构建工程
    - CMakeCache.txt文件
        - 当cmake第一次运行一个空的构建的时候，他就会创建一个CMakeCache.txt文件，文件里面存放了一些可以用来制定工程的设置，比如：变量、选项等
        - 对于同一个变量，如果Cache文件里面有设置，那么CMakeLists文件里就会优先使用Cache文件里面的同名变量。
        - CMakeLists里面通过设置了一个Cache里面没有的变量，那就将这个变量的值写入到Cache里面
        - 例子：
            - SET（var 1024）

              //变量var的值被设置成1024，如果变量var在Cache中已经存在，该命令不会覆盖cache里面的值
            - SET（var 1024..CACHE..）

              //如果var在Cache中存在，就优先使用Cache里面的值，如果不存在，就将该值写入Cache里面
            - SET（var..CACHE..FORCE）

              //无论Cache里面是否存在，都始终使用该值
    - 添加变量到Cache文件中：-D
        - 注意：-D后面不能有空格，例如：cmake -DCMAKE_BUILD_TYPE:STRING=Debug
    - 从Cache文件中删除变量：-U
        - 此选项和-D功能相反，从Cache文件中删除变量，支持使用*和？通配符
    - CMake命令行模式：-E
        - CMake提供了很多和平台无关的命令，在任何平台都可以使用：chdir, copy, copy_if_different等
        - 可以使用：cmake -E help进行查询
    - 打印运行的每一行CMake
        - 命令行选项中：--trace，将打印运行的每一行CMake，例如windows下执行：cmake --trace ..
        - 命令：--trace-source="filename"就会打印出有关filename的执行
    - 设置编译参数
        - add_definitions（-DENABLED），当在CMake里面添加该定义的时候，如果代码里面定义了#ifdef ENABLED #endif相关的片段，此时代码里面这一块代码就会生效
        - //add_definitions(“-Wall -ansi –pedantic –g”）
        - 该命令现已经被取代，使用： add_compile_definitions(WITH_OPENCV2)
    - 设置默认值命令：option
        - option命令可以帮助我们设置一个自定义的宏，如下：

- option(MY-MESSAGE "this is my message" ON)
- 第一个参数就是我们要设置的默认值的名字
- 第二个参数是对值的解释，类似于注释
- 第三个值是这个默认值的值，如果没有声明，CMake默认的是OFF
- 使用：设置好之后我们在命令行去使用的时候，也可以去给他设定值：cmake -DMY-MESSAGE=on ../
- 注意：使用的时候我们应该在值的前面加“D”
- 这条命令可将MY-MESSAGE的值设置为on，通过这个值我们可以去触发相关的判断

- CMake基础知识简介
  - 最低版本
    - 每一个CMake.txt的第一行都会写：cmake_minimum_required(VERSION 3.1)，该命令指定了CMake的最低版本是3.1
    - 命令名称cmake_minimum_required不区分大小写
    - 设置版本范围：cmake_minimum_required(VERSION 3.1...3.12)
      该命令表示支持3.1至3.12之间的版本
    - 判断CMake版本：
      - if(${CMAKE_VERSION} VERSION_LESS 3.12)
      - cmake_policy(VERSION ${CMAKE_MAJOR_VERSION}.${CMAKE_MINOR_VERSION})
      - endif()
      - 该命令表示：如果CMake版本小于3.12，则if块将为true，然后将设置为当前CMake版本；如果CMake版本高于3.12，if块为假，cmake_minimum_required将被正确执行
      - 注意：如果需要支持非命令行Windows版本则需在上面的if判断加上else分支，如下：

```
cmake_minimum_required(VERSION 3.1)
if(${CMAKE_VERSION} VERSION_LESS 3.12)
   cmake_policy(VERSION ${CMAKE_MAJOR_VERSION}.${CMAKE_MINOR_VERSION})
else()
   cmake_policy(VERSION 3.12)
endif()
```

  - 设置生成项目名称
    - 使用的命令：project（MyProject）
    - 表示我们生成的工程名字叫做：MyProject
    - 命令还可以标识项目支持的语言，写法：project（MyProject[C] [C++]）,不过通常将后面的参数省掉，因为默认支持所有语言
    - 使用该指令之后系统会自动创建两个变量：<projectname>_BINARY_DIR:二进制文件保存路径、<projectname>_SOURCE_DIR：源代码路径
    - 执行：project(MyProject)，就是定义了一个项目的名称为：MyProject，对应的就会生成两个变量：_BINARY_DIR和_SOURCE_DIR，但是cmake中其实已经有两个预定义的变量：PROJECT_BINARY_DIR和PROJECT_SOURCR_DIR
    - 关于两个变量是否相同，涉及到是内部构建还是外部构建
      - 内部构建

```
cmake ./
make
```

- 外部构建

  ```
  mkdir build
  cd ./build
  cmake ../
  make
  ```

- 内部构建和外部构建的不同在于：cmake 的工作目录不同。内部构建会将cmake生成的中间文件和可执行文件放在和项目同一目录；外部构建的话，中间文件和可执行文件会放在build目录。
- PROJECT_SOURCE_DIR和_SOURCE_DIR无论内部构建还是外部构建，指向的内容都是一样的，都指向工程的根目录
- PROJECT_BINARY_DIR和_BINARY_DIR指向的相同内容，内部构建的时候指向CMakeLists.txt文件的目录，外部构建的，指向target编译的目录

- 生成可执行文件
  - 语法：add_executable(exename srcname)

    exename:生成的可执行文件的名字
    srcname:以来的源文件
  - 该命令指定生成exe的名字以及指出需要依赖的源文件的文件名
  - 获取文件路径中的所有源文件
    - 命令：aux_sourcr_directory(<dir> <variable>)
    - 例子：aux_sourcr_directory(. DIR_SRCS)，将当前目录下的源文件名字存放到变量DIR_SRCS里面 ，如果源文件比较多，直接用DIR_SRCS变量即可
  - 生成可执行文件：add_executable(Demo ${DIR_SRCS})，将生成的可执行文件命名为：Demo.exe
- 生成lib库
  - 命令：add_library(libname [SHARED|STATIC|MODULE] [EXCLUDE_FROM_ALL] source1 source2 ... sourceN)

    libname:生成的库文件的名字
    [SHARED|STATIC|MODULE]：生成库文件的类型（动态库|静态库|模块）
    [EXCLUDE_FROM_ALL]：有这个参数表示该库不会被默认构建
    source2 ... sourceN：生成库依赖的源文件，如果源文件比较多，可以使用aux_sourcr_directory命令获取路径下所有源文件，具体章节参见：CMake基础知识简介->生成可执行文件->获取路径中所有源文件
  - 例子：add_library(ALib SHARE alib.cpp)
- 添加头文件目录
  - 命令1：target_include_directories(<target> [SYSTEM] [BEFORE] <INTERFACE|PUBLIC|PRIVATE> [items1...] [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])

    当我们添加子项目之后还需要设置一个include路径，例子：
    eg:target_include_directories(RigelEditor PUBLIC ./include/rgeditor)，表示给 RigelEditor 这个子项目添加一个库文件的路径
  - 命令2：include_directories([AFTER|BEFORE] [SYSTEM] dir1 [dir2 ...])

    参数解析：
    [AFTER|BEFORE]：指定了要添加路径是添加到原有列表之前还是之后
    [SYSTEM]：若指定了system参数，则把被包含的路径当做系统包含路径来处理
    dir1 [dir2 ...]把这些路径添加到CMakeLists及其子目录的CMakeLists的头文件包含项目中
    相当于g++选项中的-I的参数的作用
    举例：
    include_directories("/opt/MATLAB/R2012a/extern/include")

    - 两条指令的作用都是讲将include的目录添加到目标区别在于include_directories是CMake编译所有目标的目录进行添加，target_include_directories是将CMake编译的指定的特定目标的包含目录进行添加
- 添加需要链接的库文件路径
    - 命令1:target_link_libraries(<target> [item1 [item2 [...]]] [[debug|optimized|general] <item>] ...)
        - 作用：为给定的目标设置链接时使用的库（设置要链接的库文件的名称）
        - eg:target_link_libraries(MyProject a b.a c.so)　//将若干库文件链接到hello中，target_link_libraries里的库文件的顺序符合gcc/g++链接顺序规则，即：被依赖的库放在依赖他的库的后面，如果顺序有错，链接将会报错
        - 关键字：debug对应于调试配置
        - 关键字：optimized对应于所有其他的配置类型
        - 关键字：general对应于所有的配置（该属性是默认值）
    - 命令2：link_libraries
        - 作用：给当前工程链接需要的库文件（全路径）
        - eg:link_libraries(("/opt/MATLAB/R2012a/bin/glnxa64/libeng.so")//必须添加带名字的全路径
    - 区别：link_libraries和target_link_libraries命令的区别：target_link_libraries可以给工程或者库文件设置其需要链接的库文件，而且不需要填写全路径，但是link_libraries只能给工程添加依赖的库，而且必须添加全路径
    - 添加需要链接的库文件目录
        - 命令：link_directories（添加需要链接的库文件目录）
        - 语法：link_directories(directory1 directory2 ...)
        - 例子：link_directories("/opt/MATLAB/R2012a/bin/glnxa64")
    - 指令的区别：指令的前缀带target，表示针对某一个目标进行设置，必须指明设置的目标；include_directories是在编译时用，指明.h文件的路径；link_directoeies是在链接时用的，指明链接库的路径；target_link_libraries是指明链接库的名字，也就是具体谁链接到哪个库。link_libraries不常用，因为必须指明带文件名全路径
- 控制目标属性
    - 以上的几条命令的区分都是：是否带target前缀，在CMake里面，一个target有自己的属性集，如果我们没有显示的设置这些target的属性的话，CMake默认是由相关的全局属性来填充target的属性，我们如果需要单独的设置target的属性，需要使用命令：set_target_properties()
    - 命令格式

```
格式：
set_target_properties(target1 target2 ...
    PROPERTIES
    属性名称1  值
    属性名称2  值
    ...
    )
```

    - 控制编译选项的属性是：COMPILE_FLAGS
    - 控制链接选项的属性是：LINK_FLAGS
    - 控制输出路径的属性：EXECUTABLE_OUTPUT_PATH（exe的输出路径）、LIBRARY_OUTPUT_PATH（库文件的输出路径）
    - 举例：

命令：

```
set_target_properties(exe
    PROPERTIES
    LINK_FLAGS          -static
    LINK_FLAGS_RELEASE  -s
    )
```

- 这条指令会使得exe这个目标在所有的情况下都采用-static选项，而且在release build的时候 -static -s 选项。但是这个属性仅仅在exe这个target上面有效

- 变量和缓存
  - 局部变量
    - CMakeLists.txt相当于一个函数，第一个执行的CMakeLists.txt相当于主函数，正常设置的变量不能跨越CMakeLists.txt文件，相当于局部变量只在当前函数域里面作用一样，
    - 设置变量：set(MY_VARIABLE "value")
    - 变量的名称通常大写
    - 访问变量：${MY_VARIABLE}
  - 缓存变量
    - 缓存变量就是cache变量，相当于全局变量，都是在第一个执行的CMakeLists.txt里面被设置的，不过在子项目的CMakeLists.txt文件里面也是可以修改这个变量的，此时会影响父目录的CMakeLists.txt，这些变量用来配置整个工程，配置好之后对整个工程使用。
    - 设置缓存变量：set(MY_CACHE_VALUE "cache_value" CACHE INTERNAL "THIS IS MY CACHE VALUE")

      //THIS IS MY CACHE VALUE，这个字符串相当于对变量的描述说明，不能省略，但可以自己随便定义
  - 环境变量
    - 设置环境变量：set(ENV{variable_name} value)
    - 获取环境变量：$ENV{variable_name}
  - 内置变量
    - CMake里面包含大量的内置变量，和自定义的变量相同，常用的有以下：
    - CMAKE_C_COMPILER：指定C编译器
    - CMAKE_CXX_COMPILER：指定C++编译器
    - EXECUTABLE_OUTPUT_PATH：指定可执行文件的存放路径
    - LIBRARY_OUTPUT_PATH：指定库文件的放置路径
    - CMAKE_CURRENT_SOURCE_DIR：当前处理的CMakeLists.txt所在的路径
    - CMAKE_BUILD_TYPE：控制构建的时候是Debug还是Release

      命令：set(CMAKE_BUILD_TYPE Debug)
    - CMAKE_SOURCR_DIR：无论外部构建还是内部构建，都指的是工程的顶层目录（参考project命令执行之后，生成的_SOURCR_DIR以及CMake预定义的变量PROJECT_SOURCE_DIR）
    - CMAKE_BINARY_DIR：内部构建指的是工程顶层目录，外部构建指的是工程发生编译的目录（参考project命令执行之后，生成的_BINARY_DIR以及CMake预定义的变量PROJECT_BINARY_DIR）
    - CMAKE_CURRENT_LIST_LINE：输出这个内置变量所在的行
  - 缓存

- 缓存就是之前提到的CMakeCache文件，参见：CMake命令行选项的设置->CMakeCache.txt文件

- CMake基本控制语法
  - If
    - 基本语法

      ```
      if (expression)
        COMMAND1(ARGS ...)
        COMMAND2(ARGS ...)
        ...
      else (expression)
        COMMAND1(ARGS ...)
        COMMAND2(ARGS ...)
        ...
      endif (expression)
      ```

      注意：ENDIF要和IF对应
      - if (expression)，expression不为：空,0,N,NO,OFF,FALSE,NOTFOUND或<var >_NOTFOUND,为真
      - IF (not exp)，与上面相反
      - if (var1 AND var2)，var1且var2都为真，条件成立
      - if (var1 OR var2)，var1或var2其中某一个为真，条件成立
      - if (COMMAND cmd)， 如果cmd确实是命令并可调用，为真；
      - if (EXISTS dir) 如果目录存在，为真
      - if (EXISTS file) 如果文件存在，为真
      - if (file1 IS_NEWER_THAN file2)，当file1比file2新，或file1/file2中有一个不存在时为真，文件名需使用全路径
      - if (IS_DIRECTORY dir) 当dir是目录时，为真
      - if (DEFINED var) 如果变量被定义，为真
      - if (string MATCHES regex) 当给定变量或字符串能匹配正则表达式regex时，为真

        ```
        例:
        IF ("hello" MATCHES "ell")
        MESSAGE("true")
        ENDIF ("hello" MATCHES "ell")
        ```
    - 数字表达式
      - if (var LESS number)，var小于number为真
      - if (var GREATER number)，var大于number为真
      - if (var EQUAL number)，var等于number为真
    - 字母表顺序比较
      - IF (var1 STRLESS var2)，var1字母顺序小于var2为真
      - IF (var1 STRGREATER var2)，var1字母顺序大于var2为真
      - IF (var1 STREQUAL var2)，var1和var2字母顺序相等为真
  - While
    - 语法结构

      ```
      WHILE(condition)
        COMMAND1(ARGS ...)
        COMMAND2(ARGS ...)
        ...
      ENDWHILE(condition)
      ```

- 真假条件的判断参考if

- Foreach
  - FOREACH有三种使用形式的语法，且每个FOREACH都需要一个ENDFOREACH()与之匹配。
  - 列表循环
    - 语法

      ```
      FOREACH(loop_var arg1 arg2 ...)
          COMMAND1(ARGS ...)
          COMMAND2(ARGS ...)
       ...
      ENDFOREACH(loop_var)
      ```

    - 例子

      ```
      eg:
      AUX_SOURCE_DIRECTORY(. SRC_LIST)
      FOREACH(F ${SRC_LIST})
          MESSAGE(${F})
      ENDFOREACH(F)
      ```

    - 例子中，先将当前路径的源文件名放到变量SRC_LIST里面，然后遍历输出文件名
  - 范围循环
    - 语法

      ```
      FOREACH(loop_var RANGE total)
      COMMAND1(ARGS ...)
      COMMAND2(ARGS ...)
      ...
      ENDFOREACH(loop_var)
      ```

    - 例子

      ```
      eg:
      FOREACH(VAR RANGE 100)
      MESSAGE(${VAR})
      ENDFOREACH(VAR)
      ```

    - 例子中默认起点为0，步进为1，作用就是输出：0~100
  - 范围步进循环
    - 语法

      ```
      FOREACH(loop_var RANGE start stop [step])
      COMMAND1(ARGS ...)
      COMMAND2(ARGS ...)
      ...
      ENDFOREACH(loop_var)
      ```

    - 例子

      ```
      eg:
      FOREACH(A RANGE 0 100 10)
      MESSAGE(${A})
      ENDFOREACH(A)
      ```

    - 例子中，起点是0，终点是100，步进是10，输出：0,10,20,30,40,50,60,70,80,90,100

- 构建规范以及构建属性//
  - 用于指定构建规则以及程序使用要求的指令：target_include_directories(), target_compile_definitions(), target_compile_options()

- 指令格式
  - target_include_directories(<target> [SYSTEM] [BEFORE] <INTERFACE|PUBLIC|PRIVATE> [items1...] [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
    > Include的头文件的查找目录，也就是Gcc的[-Idir...]选项
  - target_compile_definitions(<target> <INTERFACE|PUBLIC|PRIVATE> [items1...][<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])
    > 通过命令行定义的宏变量
  - target_compile_options(<target> [BEFORE] <INTERFACE|PUBLIC|PRIVATE> [items1...] [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...]
    > gcc其他的一些编译选项指定，比如-fPIC
    >
    - -fPIC选项说明
      > 说明：-fPIC 作用于编译阶段，告诉编译器产生与位置无关代码(Position-Independent Code)，
      > 则产生的代码中，没有绝对地址，全部使用相对地址，故而代码可以被加载器加载到内存的任意
      > 位置，都可以正确的执行。这正是共享库所要求的，共享库被加载时，在内存的位置不是固定的。
    - -Idir选项说明
      > 说明：在你是用 #include "file" 的时候, gcc/g++ 会先在当前目录查找你所制定的头文件, 如果没有找到, 他会到缺省的头文件目录找, 如果使用 -I 制定了目录,他会先在你所制定的目录查找, 然后再按常规的顺序去找。
- 以上的额三个命令会生成 INCLUDE_DIRECTORIES, COMPILE_DEFINITIONS, COMPILE_OPTIONS变量的值.。或者 INTERFACE_INCLUDE_DIRECTORIES,INTERFACE_COMPILE_DEFINITIONS, INTERFACE_COMPILE_OPTIONS的值.
- 这三个命令都有三种可选模式: PRIVATE, PUBLIC。 INTERFACE. PRIVATE模式仅填充不是接口的目标属性; INTERFACE模式仅填充接口目标的属性.PUBLIC模式填充这两种的目标属性。
-

- 宏和函数
  - CMake里面可以定义自己的函数（function）和宏（macro）
  - 区别1：范围。函数是有范围的，而宏没有。如果希望函数设置的变量在函数的外部也可以看见，就需要使用PARENT_SCOPE来修饰，但是函数对于变量的控制会比较好，不会有变量泄露
  - 例子
    - 宏（macro）
      ```
      eg:
      macro( [arg1 [arg2 [arg3 ...]]])
          COMMAND1(ARGS ...)
          COMMAND2(ARGS ...)
          ...
      endmacro()
      ```
    - 函数（function）
      ```
      eg:
      function( [arg1 [arg2 [arg3 ...]]])
          COMMAND1(ARGS ...)
          COMMAND2(ARGS ...)
      ```

```
    ...
endfunction()
```

- 函数和宏的区别还在于，函数很难将计算结果传出来，使用宏就可以将一些值简单的传出来
- 例子

```
eg:
macro(macroTest)
    set(test1 "aaa")
endmacro()

function(funTest)
    set(test2 "bbb")
endfunction()

macroTest()
message("${test1}")

funTest()
message("${test2}")
```

- 运行上面这个代码，就会显示“aaa”，因为函数里面的test1是局部的，出了这个函数就出了他的作用域

- 和其他文件的交互
  - 在代码中使用CMake中定义的变量
    - 命令：configure_file
    - 作用：让普通文件也能使用CMake中的变量。
    - 语法

      configure_file(<input> <output>
      [COPYONLY] [ESCAPE_QUOTES] [@ONLY]
      [NEWLINE_STYLE [UNIX|DOS|WIN32|LF|CRLF]])
      解释：拷贝一个 <input>（输入文件） 文件到 <output> （输出文件），并且替换输入文件中被 @VAR@ 或者 ${VAR} 引用的变量值。每一个变量将被替换成当前的变量值
    - 参数解析
      - COPYONLY：只拷贝文件，不进行任何的变量替换。这个选项在指定了 NEWLINE_STYLE 选项时不能使用（无效）。
      - ESCAPE_QUOTES：躲过任何的反斜杠(C风格)转义。

        注：躲避转义，比如你有个变量在CMake中是这样的 set(FOO_STRING "\"foo\"") 那么在没有 ESCAPE_QUOTES 选项的状态下，通过变量替换将变为 ""foo""，如果指定了 ESCAPE_QUOTES 选项，变量将不变。
      - @ONLY：限制变量替换，让其只替换被 @VAR@ 引用的变量（那么 ${VAR} 格式的变量将不会被替换）。这在配置 ${VAR} 语法的脚本时是非常有用的。
      - NEWLINE_STYLE <style>：指定输出文件中的新行格式。UNIX 和 LF 的新行是 \n ， DOS 和 WIN32 和 CRLF 的新行格式是 \r\n 。 这个选项在指定了 COPYONLY 选项时不能使用（无效）。
  - 在CMake对文件的操作
    - file命令
      - file(WRITE filename "message to write"...)

- 解释：WRITE选项会写一条消息到名为filename中，如果文件存在，则会覆盖原文件，如果文件不存在，他将创建该文件
- file(APPEND filename "message to write"...)
- 解释：APPEND选项和WRITE选项一样，只是APPEND会写到文件的末尾
- file(READ filename variable [LIMIT numBytes] [OFFSET offset] [HEX])
- 解释：READ选项会将读取的文件内容存放到变量variable ，读取numBytes个字节，从offset位置开始，如果指定了[HEX]参数，二进制代码就会转换为十六进制的转换方式
- file(STRINGS filename variable [LIMIT_COUNT num] [LIMIT_INPUT numBytes] [LIMIT_OUTPUT numBytes] [LENGTH_MINIMUM numBytes] [LENGTH_MAXIMUM numBytes] [NEWLINE_CONSUME] [REGEX regex] [NO_HEX_CONVERSION])
- 解释：STRINGS标志，将会从一个文件中将ASCII字符串的list解析出来，然后储存在variable 变量中，文件中的二进制数据将会被忽略，回车换行符会被忽略（可以设置NO_HEX_CONVERSION选项来禁止这个功能）。LIMIT_COUNT：设定了返回字符串的最大数量；LIMIT_INPUT：设置了从输入文件中读取的最大字节数；LIMIT_OUTPUT：设置了在输出变量中允许存储的最大字节数；LENGTH_MINIMUM：设置了返回字符串的最小长度，小于该长度的字符串将会被忽略；LENGTH_MAXIMUM设置了返回字符串的最大长度，大于该长度的字符串将会被忽略；NEWLINE_CONSUME：该标志允许新行被包含到字符串中，而不是终止他们；REGEX：指定了返回的字符串必须满足的正则表达式
- 典型的使用方式：file(STRINGS myfile.txt myfile)
- 该命令在变量myfile中储存了一个list,该list每一项是myfile.txt中的一行文本
- file(GLOB variable [RELATIVE path] [globbing expressions]...)
- 解释：GLOB：该选项将会为所有匹配表达式的文件生成一个文件list，并将该list存放在variable 里面，文件名的查询表达式和正则表达式类似，
- 查询表达式的例子：①*.cpp -匹配所有后缀是.cpp的文件②*.vb? -匹配文件后缀是.vba——.vbz的文件③f[3-5].txt ： 匹配f3.txt,f4.txt,f5.txt文件
- file(GLOB_RECURSE variable [RELATIVE path] [FOLLOW_SYMLINKS] [globbing expressions]...)
- 解释：GLOB_RECURSE会生成一个类似于通常GLOB选项的list，不过该选项可以递归查找文件中的匹配项
- 比如：/dir/*.py -就会匹配所有在/dir文件下面的python文件，
- file(RENAME <oldname> <newname>)
- 解释：RENAME选项对同一个文件系统下的一个文件或目录重命名
- file(REMOVE [file1 ...])
- 解释：REMOVE选项将会删除指定的文件，包括在子路径下的文件
- file(REMOVE_RECURSE [file1 ...])
- 解释：REMOVE_RECURSE选项会删除给定的文件以及目录，包括非空目录
- file(MAKE_DIRECTORY [directory1 directory2 ...])
- 解释：MAKE_DIRECTORY选项将会创建指定的目录，如果它们的父目录不存在时，同样也会创建
- file(RELATIVE_PATH variable directory file)
- 解释：RELATIVE_PATH选项会确定从direcroty参数到指定文件的相对路径，然后存到变量variable中
- file(TO_CMAKE_PATH path result)

        - 解释：TO_CMAKE_PATH选项会把path转换为一个以unix的 / 开头的cmake风格的路径
        - file(TO_NATIVE_PATH path result)
        - 解释：TO_NATIVE_PATH选项与TO_CMAKE_PATH选项很相似，但是它会把cmake风格的路径转换为本地路径风格
        - file(DOWNLOAD url file [TIMEOUT timeout] [STATUS status] [LOG log] [EXPECTED_MD5 sum] [SHOW_PROGRESS])
        - 解释：DOWNLOAD将给定的url下载到指定的文件中，如果指定了LOG log，下载的日志将会被输出到log中，如果指定了STATUS status选项下载操作的状态就会被输出到status里面，该状态的返回值是一个长度为2的list，list第一个元素是操作的返回值，是一个数字 ，第二个返回值是错误的字符串，错误信息如果是0，就表示没有错误；如果指定了TIMEOUT time选项，time秒之后，操作就会推出。如果指定了EXPECTED_MD5 sum选项，下载操作会认证下载的文件的实际MD5和是否与期望值相匹配，如果不匹配，操作将返回一个错误；如果指定了SHOW_PROGRESS，进度信息会被打印出来，直到操作完成
    - source_group命令
        - 使用该命令可以将文件在VS中进行分组显示
        - source_group("Header Files" FILES ${HEADER_FILES})
        - 以上命令是将变量HEADER_FILES里面的文件，在VS显示的时候都显示在“Header Files”选项下面

- 如何构建项目
    - 工程文件结构
        - lib文件夹
            - libA.c
            - libB.c
            - CMakeLists.txt
        - include文件夹
            - includeA.h
            - inclueeB.h
            - CMakeLists.txt
        - main.c
        - CMakeLists.txt
    - 第一层CMakeLists

      内容如下：

      #项目名称
      project(main)

      #需要的cmake最低版本
      cmake_minium_required(VERSION 2.8)

      #将当前目录下的源文件名都赋给DIR_SRC目录
      aux_source_directories(. DIR_SRC)

      #添加include目录
      include_directories(include)

```
#生成可执行文件
add_executable(main ${DIR_SRC})

#添加子目录
add_subdirectories(lib)

#将生成的文件与动态库相连
target_link_libraries(main test)
#test是lib目录里面生成的
```

- lib目录下的CMakeLists

  内容如下：

  ```
  #将当前的源文件名字都添加到DIR_LIB变量下
  aux_source_director(. DIR_LIB)

  #生成库文件命名为test
  add_libraries(test ${DIR_LIB})
  ```

- include目录的CMakeLists可以为空，因为我们已经将include目录包含在第一层的文件里面

- 运行其他程序
  - 在配置时运行命令
    - 指令：execute_process

      参数：

      ```
      execute_process(COMMAND <cmd1> [args1...]]
                      [COMMAND <cmd2> [args2...] [...]]
                      [WORKING_DIRECTORY <directory>]
                      [TIMEOUT <seconds>]
                      [RESULT_VARIABLE <variable>]
                      [OUTPUT_VARIABLE <variable>]
                      [ERROR_VARIABLE <variable>]
                      [INPUT_FILE <file>]
                      [OUTPUT_FILE <file>]
                      [ERROR_FILE <file>]
                      [OUTPUT_QUIET]
                      [ERROR_QUIET]
                      [OUTPUT_STRIP_TRAILING_WHITESPACE]
                      [ERROR_STRIP_TRAILING_WHITESPACE])
      ```

    - 作用：这条指令可以执行系统命令，将输出保存到cmake变量或文件中去，运行一个或多个给定的命令序列，每一个进程的标准输出通过管道流向下一个进程的标准输入。
    - 参数解析
      - COMMAND：子进程的命令行，CMake使用操作系统的API直接执行子进程，所有的参数逐字传输，没有中间脚本参与，像“>”的输出重定向也会被直接的传输到子进程里面，当做普通的参数进行处理。
      - WORKING_DIRECTORY：指定的工作目录将会设置为子进程的工作目录
      - TIMEOUT：子进程如果在指定的秒数之内没有结束就会被中断
      - RESULT_VARIABLE：变量被设置为包含子进程的运算结果，也就是命令执行的最后结果将会保存在这个变量之中，返回码将是来自最后一个子进程的整数或者一个错误描述字符串
      - OUTPUT_VARIABLE、ERROR_VARIABLE：输出变量和错误变量//
      - INPUT_FILE、OUTPUT_FILE、ERROR_FILE：输入文件、输出文件、错误文件//

- OUTPUT_QUIET、ERROR_QUIET:输出忽略、错误忽略，标准输出和标准错误的结果将被默认忽略
- 例子

```
eg:
set(MAKE_CMD "/src/bin/make.bat")
MESSAGE("COMMAND: ${MAKE_CMD}")
execute_process(COMMAND "${MAKE_CMD}"
  RESULT_VARIABLE CMD_ERROR
  OUTPUT_FILE CMD_OUTPUT)
MESSAGE( STATUS "CMD_ERROR:" ${CMD_ERROR})
MESSAGE( STATUS "CMD_OUTPUT:" ${CMD_OUTPUT})
输出：
COMMAND:/src/bin/make.bat
CMD_ERROR:No such file or directory
CMD_OUTPUT:
（因为这个路径下面没有这个文件）
```

- 在构建的时运行命令

https://www.jianshu.com/p/0fc0e1613587

- 例子（调用python脚本生成头文件）：

```
find_package(PythonInterp REQUIRED)
add_custom_command(OUTPUT
"${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp"
   COMMAND "${PYTHON_EXECUTABLE}"
"${CMAKE_CURRENT_SOURCE_DIR}/scripts/GenerateHeader.py" --argument
   DEPENDS some_target)
add_custom_target(generate_header ALL
   DEPENDS "${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp")
install(FILES ${CMAKE_CURRENT_BINARY_DIR}/include/Generated.hpp
DESTINATION include)
```

- find_package：查找链接库

  如果编译的过程使用了外部的库，事先并不知道其头文件和链接库的位置，得在编译命令中加上包含外部库的查找路径，CMake中使用find_package方法

  - find_package（）命令查找***.cmake的顺序
    - 介绍这个命令之前，首先得介绍一个变量：CMAKE_MODULE_PATH
      - 工程比较大的时候，我们会创建自己的cmake模块，我们需要告诉cmake这个模块在哪里，CMake就是通过CMAKE_MODULE_PATH这个变量来获取模块路径的
      - 我们使用set来设置模块的路径：set(CMAKE_MODULE_PATH ${PROJECT_SOURCE_DIR}/cmake)
    - 如果上面的没有找到，就会在../.cmake/packages或者../uesr/local/share/中的包目录中查找：<库名字大写>Config.cmake或者<库名字小写>-config.cmake。这种查找模式称作Config模式。
    - 如果找到这个包，则可以通过在工程的顶层目录中的CMakeLists.txt中添加：include_directories(<Name>_INCLUDE_DIRS)来包含库的头文件，使用命令：target_link_libraries(源文件 <NAME>_LIBRARIES)将源文件以及库文件链接起来
    - 无论哪一种方式，只要找到***.cmake文件，***.cmake里面都会定义下面这些变量

      ```
      <NAME>_FOUND
      <NAME>_INCLUDE_DIRS or <NAME>_INCLUDES
      ```

> <NAME>_LIBRARIES or <NAME>_LIBRARIES or <NAME>_LIBS
> <NAME>_DEFINITIONS
> 注：<NAME>就是库名

- CMake中使用：cmake --help-module-list命令来查看当前CMake中有哪些支持的模块

- find_package（<Name>）命令首先会在模块路径,也就是刚才我们介绍的CMAKE_MODULE_PATH变量里面存放的路径中查找Find<Name>.cmake。查找路径依次为：变量${CMAKE_MODULE_PATH}中的所有目录，这种查找模式被称为Module模式。
- find_package
  - 命令参数
    - FIND_PACKAGE(<name> [version] [EXACT] [QUIET] [NO_MODULE] [[REQUIRED | COMPONENTS] [componets...]])
    - version：需要一个版本号，给出这个参数而没有给出EXACT，那个就是找到和给出的这个版本号相互兼容就符合条件
    - EXACT：要求版本号必须和version给出的精确匹配。
    - QUIET：会禁掉查找的包没有被发现的警告信息。对应于Find<Name>.cmake模块里面的的NAME_FIND_QUIETLY变量。
    - NO_MODULE：给出该指令之后，cmake将直接跳过Module模式的查找，直接使用Config模式查找。查找模式详见下方
    - REQUIRED：该选项表示如果没有找到需要的包就会停止并且报错
    - COMPONENTS：在REQUIRED选项之后，或者如果没有指定REQUIRED选项但是指定了COMPONENTS选项，在COMPONENTS后面就可以列出一些与包相关部分组件的清单
  - 搜索原理
    - Cmake可以支持很多外部内部的库，通过命令可以查看当前cmake支持的模块有哪些： cmake --help-module-list 。或者直接查看模块路径。Windosw的路径在cmak的安装目录：..\share\cmake-3.13\Modules
    - cmake本身是不提供任何搜索库的便捷方法，所有搜索库并给变量赋值的操作必须由cmake代码完成
    - find_psckage的搜索模式：
    - Module模式：搜索CMAKE_MODULE_PATH指定路径下的FindXXX.cmake文件（XXX就是我们要搜索的库的名字），这个CMAKE_MODULE_PATH变量是cmake预先定义，但是没有值，我们一旦给这个变量赋值之后，cmake就会最高优先级的在这个变量里面去查找，没有找到就在自己的安装库里面去找有没有FindXXX.cmake模块，找到之后，执行该文件从而找到XXX库，其中具体查找库并给XXX_INCLUDE_DIR和XXX_LIBRARIES这两个变量赋值的操作有FindXXX.cmake模块完成
    - Config模式：如果Module模式没有找到，则启用Config模式查找，搜索XXX_DIR路径下的XXXConfig.cmake文件，执行该文件从而找到XXX库，其中查找库以及给XXX_INCLUDE_DIR和XXX_LIBRARY赋值的操作都是由XXXConfig.cmake模块完成

- cmake默认采取的时Module模式，如果Module模式没有找到，才会使用Config模式查找，
- ---
- # version选项和EXACT

  下载的BZip2版本是：1.0.5
  find_package里面提供的版本是：1.0.4
  命令如下：
  find_package(BZip2 1.0 REQUIRED)
  此时没有指定EXACT参数，只要找到和这个版本兼容的版本也行，设置之后会给我们提示：

  ```
  Found BZip2: D:/bzip2-1.0.5/GnuWin32/lib/bzip2.lib (found suitable version "1.0.5", minimum required is "1.0.4")
  Looking for BZ2_bzCompressInit
  Looking for BZ2_bzCompressInit - found
  ```

  找到版本1.0.5，最低要求：1.0.4

  但是当我们设置精确查找选项之后：EXACT，Cmake就会报错

  ```
  CMake Error at C:/Program Files/CMake/share/cmake-3.13/Modules/FindPackageHandleStandardArgs.cmake:137 (message):
    Could NOT find BZip2: Found unsuitable version "1.0.5", but required is
    exact version "1.0.4" (found D:/bzip2-1.0.5/GnuWin32/lib/bzip2.lib)
  ```

  找到版本1.0.5，但是要求1.0.4
- ---
- COMPONENTS选项，有些库不是一个整体比如Qt，其中还包含QtOpenGL和QtXml组件，当我们需要使用库的组件的时候，就使用COMPONENTS这个选项
- find_package(Qt COMPONENTS QtOpenGL QtXml REQUIRED)
- ---
- # 找到之后给一些预定义的变量赋值
- 无论哪一种查找模式，只要找到包之后，就会给以下变量赋值：<NAME>_FOUND,<NAME>_INCLUDE_DIRS或者<NAME>_INCLUDES,<NAME>_LIBRARIES,<NAME>_DEFINITIONS。这些变量都在Find<NAME>.cmake文件中。
- 我们可以在CMakeLists.txt中使用<NAME>_FOUND变量来检测摆包是否被找到，
- ---
- find_package的本质是执行一个.cmake文件，相当于cmake的内置的脚本，这个脚本将设置我们之前提到的相关的变量，相当于根据传进来的参数来使用一个查找模块，每一个常用的库在cmake里面就有一个对应的查找模块。
- ---
- # find模块的编写流程：
- 首先使用：find_path和find_library查找模块的头文件以及库文件，然后将结果放到<NAME>_INCLUDE_DIR和<NAME>_LIBRARY里面
  - find_path()：
  - find_path(<VAR> name1 [path1 path2 ...])

    - 该命令搜索包含某个文件的路径，用于给定名字的文件所在路径，
    - 一条名为：<VAR>的变量的Cache将会被创建。
    - 如果在某个文件下面发现了该文件，路径就会被储存到变量里面，除非变量被清除，否则搜搜将不会进行。
    - 如果没有发现该文件，<VAR>里面储存的就是<VAR>-NOTFOUND
    - find_library():
    - find_library(<VAR> name1 [path1 path2 ...])
    - 查找一个库文件
- 设置：<NAME>_INCLUDE_DIRS为<NAME>_INCLUDE_DIR<dependency1>_INCLUDE_DIRS ...
- 设置 <name>_LIBRARIES 为 <name>_LIBRARY <dependency1>_LIBRARIES ...
- 调用宏 find_package_handle_standard_args() 设置 <name>_FOUND 并打印或失败信息
- ----------
- 我们以Cmake里面自带的bzip2库为例，Cmake的module目录里面有一个：FindBZip.cmake模块，我们使用find_package（BZip2），然后CMake就会给相关的变量赋值，我们就可以调用这个模块，就可以使用模块里面的变量，模块里面的变量有哪些，我们可以使用命令： cmake --help-module FindBZip2来查看，最后面的参数就是带上Find前缀之后的模块的名字。
- 假如一个程序需要使用BZip2库，编译器需要知道bzlib.h的位置，链接器需要知道bzip2库（动态链接：.so或者.dll）

```
cmake_minimum_required(VERSION 2.8)
project(helloworld)
add_executable(hello main.c)
#find_package包含BZip2库
find_package(BZip2)
#如果上一步查找，BZIP2_FOUND将会被设置成1
if(BZIP2_FOUND)
    include_directories(${BZIP_INCLUDE_DIRS})
    target_link_libraries(hello &{BZIP2_LIBRARIES})
endif(BZIP2_FOUND)
```

- 添加库的指令：find_package(BZip2 REQUIRED)
- CMake里面又很多内置的库，当我们使用find_package查找包的时候CMake首先会去CMAKE_MODULE_PATH这个变量存放的路径里面去寻找
- 注意：CMAKE_MODULE_PATH的路径设置需要在顶层的CMakeLists.txt里面去设置。
- find_package之后，变量：BZIP_INCLUDE_DIRS以及BZIP2_LIBRARIES就会被设置，然后我们使用include_directories以及target_link_libraries来使用即可
- ----------
- 使用一个cmake里面没有带的库

- cmake的/share/cmake-X.XX/Modules里面带的都是一些常用的库，如果我们现在需要使用一个cmake里面没有提供给我们find模块的库，做法如下：
- 首先模拟生成一个生成lib文件
  - 创建工程，名为：ThirdLIB
  - 生成一个lib文件，名为：ThirdDLL.lib
  - lib文件里面只提供了一个求和的add函数，返回两个int值的和
- 在CMakeLists.txt里面设置CMAKE_MODULE_PATH，这里设置的是本地的路径，这个路径存放的find模块

```
#设置CMAKE_MODULE_PATH
set(CMAKE_MODULE_PATH "C:\\Users\\DY\\Desktop\\CMakeEg2\\EG1\\EG1\\cmake\\modules")
```

- 编写自己的find模块
  - 注意：cmake使用find_package查找使用的库，当我们把库名字传进去之后，cmake会在按照指定的模式查找一个：Find<NAME>.cmake的文件，常用的库，cmake都会提供对应的.cmake文件，但是现在我们使用的是自己编写的库，所以cmake是没有提供的，需要自己编写
  - 编写大致流程已经给出，我们编写的文件名必须是：Find<Name>.cmake,现在就是“FindThidrDLL.cmake”
  - FindThirDLL.cmake内容如下

```
#-----------------------------------------这是一条分割线----------------------------------------------------------------
# 判断是否已经包含log4cpp
if (THIRDLIB_INCLUDE_DIR)
  message("DY:set value of THIRDLIB_FIND_QUIETLY")
  set(THIRDLIB_FIND_QUIETLY TRUE)
endif ()

# 查找头文件位置
find_path(THIRDLIB_INCLUDE_DIR
  mymath.h
  # 可以通过以下命令来手动制定查找路径
 "C:\\Users\\DY\\Desktop\\ThirdLIB\\ThirdLIB"
)

# 查找库文件位置
find_library(THIRDLIB_LIBRARY
  ThirdLIB
  "C:\\Users\\DY\\Desktop\\ThirdLIB\\Debug"
)

# 同时找到头文件位置和库文件位置时给相关变量赋值
if (THIRDLIB_INCLUDE_DIR AND THIRDLIB_LIBRARY)
  set(THIRDLIB_FOUND TRUE)
  set(THIRDLIB_LIBRARIES ${THIRDLIB_LIBRARY})
  set(THIRDLIB_INCLUDE_DIRS ${THIRDLIB_INCLUDE_DIR})
else ()
  set(THIRDLIB_FOUND FALSE)
  message(WARNING "LOG4CPP not found")
endif ()

# 打印一些错误信息
if (THIRDLIB_FOUND)
  if (NOT THIRDLIB_FIND_QUIETLY)
    message(STATUS "Found THIRDLIB: ${THIRDLIB_LIBRARIES}")
  endif ()
else ()
  if (THIRDLIB_FIND_REQUIRED)
    message(STATUS "Looked for THIRDLIB libraries named ${THIRDLIB_NAMES}.")
    message(FATAL_ERROR "Could NOT find THIRDLIB library")
  endif ()
endif ()

# 这个选项搞不懂，貌似是给cmake gui用，根据第一参数是CLEAR还是FORCR还是空，然后在cmake的gui里面做出一些处理
mark_as_advanced(
  THIRDLIB_LIBRARIES
  THIRDLIB_INCLUDE_DIRS
)
```

  - 修改CMakeLists.txt文件，在里面使用find_package命令添加模块、

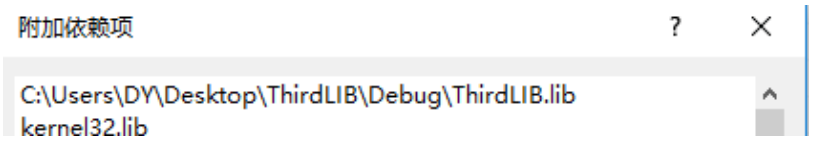
```
find_package(ThirdLIB REQUIRED)

if(NOT THIRDLIB_FOUND)
    message("没有找到ThirdLIB或者find模块出错")
endif(THIRDLIB_FOUND)

message("这是THIRDLIB_FOUND的值：${THIRDLIB_FOUND}")
message("这是THIRDLIB_LIBRARIES的值：${THIRDLIB_LIBRARIES}")
message("这是THIRDLIB_INCLUDE_DIRS的值：${THIRDLIB_INCLUDE_DIRS}")

include_directories(${THIRDLIB_INCLUDE_DIRS})
target_link_libraries(Test ${THIRDLIB_LIBRARIES})
```

- 完成之后，打开生成的工程，查看工程的依赖项，就会有ThirdLIB.lib的选项



- ----------
- 当我们在CMake里面使用一个库时候，如何在网上查找，比如现在需要查找：apr库，在goole里面搜索：find package apr cmake，就可以直接找到对应的CMake脚本。然后复制粘贴，创建.cmake文件，放在工程根目录下面的modules目录，没有则创建之
- ----------
-

- add_custom_command：（1）为某一个工程添加一个自定义的命令

```
add_custom_command(TARGET target
                   PRE_BUILD | PRE_LINK| POST_BUILD
                   COMMAND command1[ARGS] [args1...]
                   [COMMAND command2[ARGS] [args2...] ...]
                   [WORKING_DIRECTORYdir]
                   [COMMENT comment][VERBATIM])
```

作者：drybeans
链接：https://www.jianshu.com/p/66df9650a9e2
來源：简书
简书著作权归作者所有，任何形式的转载都请联系作者获得授权并注明出处。

  - 执行命令的时间由第二个参数决定

    1.PRE_BUILD - 命令将会在其他依赖项执行前执行
    2.PRE_LINK - 命令将会在其他依赖项执行完后执行
    3.POST_BUILD - 命令将会在目标构建完后执行。

  - 例子1：

```
add_custom_command (
 TARGET ${PROJECT_NAME}
 POST_BUILD
 COMMAND ${CMAKE_COMMAND} -E sleep 5
)
```

    #目标就是TARGET后面跟的工程，当PROJECT_NAME被生成的时候就会执行COMMAND后面的命令

  - 例子2：

```
add_custom_command(TARGET test_elf
PRE_BUILD
COMMAND
move E:/cfg/start.o ${CMAKE_BINARY_DIR}/. &&
)
```

    #在test_el执行依赖之前，将start.o文件复制到编译目录

- add_custom_command：（2）添加自定义命令来产生一个输出

```
add_custom_command(OUTPUT output1 [output2 ...]
                   COMMAND command1[ARGS] [args1...]
                   [COMMAND command2 [ARGS] [args2...] ...]
                   [MAIN_DEPENDENCYdepend]
```

```
            [DEPENDS[depends...]]
            [IMPLICIT_DEPENDS<lang1> depend1 ...]
            [WORKING_DIRECTORYdir]
            [COMMENT comment] [VERBATIM] [APPEND])
```

- 其中ARGS选项 是为了向后兼容，MAIN_DEPENDENCY选项是针对Visual Studio给出一个建议，这两选项可以忽略
- COMMAND：指定一些在构建阶段执行的命令。如果指定了多于一条的命令，他会按照顺序去执行。如果指定了一个可执行目标的名字（被add_executable()命令创建），他会自动被在构建阶段创建的可执行文件的路径替换，
- DEPENDS:指定目标依赖的文件，如果依赖的文件是和CMakeLists.txt相同目录的文件，则命令就会在CMakeLists.txt文件的，目录执行。如果没有指定DEPENDS，则只要缺少OUTPUT，该命令就会执行。如果指定的位置和CMAkeLists.txt不是同一位置，会先去创建依赖关系，先去将依赖的目标或者命令先去编译。
- WORKING_DIRECTORY：使用给定的当前目录执行命令，如果是相对路径，则相对于当前源目录对应的目录结构进行解析
- 例子

```
#首先生成creator的可执行文件
add_executable(creator creator.cxx)

#获取EXE_LOC的LOCATION属性存放到creator里面
get_target_property(creator EXE_LOC LOCATION)

#生成created.c文件
add_custom_command (
  OUTPUT ${PROJECT_BINARY_DIR}/created.c
  DEPENDS creator
  COMMAND ${EXE_LOC}
  ARGS ${PROJECT_BINARY_DIR}/created.c
  )

#使用上一步生成的created.c文件来生成Foo可执行文件
add_executable(Foo ${PROJECT_BINARY_DIR}/created.c)
```

- 注意：不要再多个相互独立的文件中使用该命令产生相同的文件，防止冲突。

- add_custom_target：增加定制目标

```
add_custom_target(Name [ALL] [command1 [args1...]]
            [COMMAND command2 [args2...] ...]
            [DEPENDS depend depend depend ... ]
            [BYPRODUCTS [files...]]
            [WORKING_DIRECTORY dir]
            [COMMENT comment]
            [VERBATIM] [USES_TERMINAL]
            [SOURCES src1 [src2...]])
```

- 命令 add_custom_target 可以增加定制目标，常常用于编译文档、运行测试用例等。

- add_custom_command和add_custom_target的区别
  - 命令命名里面的区别就在于：command和target，前者是自定义命令，后者是自定义目标
  - 目标：一般来说目标是调用：add_library或者add_executable生成的exe或者库，他们具有许多属性集，这些就是所谓目标，而使用

add_custom_target定义的叫做自定义目标，因此这些“目标”区别于正常的目标，他们不生成exe或者lib，但是仍然会具有一些正常目标相同的属性，构建他们的时候，只是调用了为他们设置的命令，如果自定义目标对于其他目标有依赖，那么就会优先生成依赖的那些目标

- 自定义命令：自定义命令不是一个“可构建”的对象，并且没有可以设置的属性，自定义命令是一个在构建依赖目标之前被调用的命令，自定义命令的依赖可以通过add_custom_command(TARGET target ...)形式显式设置，也可以通过add_custom_command(OUTPUT output1 ...)生成文件的形式隐式设置。显示执行的时候，每次构建目标，首先会执行自定义的命令，隐式执行的时候，如果自定义的命令依赖于其他文件，则在构建目标的时候先去执行生成其他文件。

- 如何添加C++项目中的常用选项

  如：如何支持C++11、如何支持IDE等

  - 如何激活C++11功能

    语法：target_compile_features(<target> <PRIVATE|PUBLIC|INTERFACE> <feature> [...])

    - target_compile_features(<project_name> PUBLIC cxx_std_11)
    - 参数target必须是由：add_executable或者add_library生成的目标
    - 另外一种支持C++标准的方法

      ```
      #设置C++标准级别
      set(CMAKE_CXX_STANDARD 11)

      #告诉CMake使用他它
      set(CMAKE_CXX_STANDARD_REQUIRED ON)

      #（可选）确保-std=C++11
      set(CMAKE_CXX_EXTENSIONS OFF)
      ```

  - CMake的过程间优化

    - 如果编译器不支持，就会将设置的过程间优化标记为错误，可以使用命令：check_ipo_supported()来查看

      ```
      #检测编译器是否支持过程间优化
      check_ipo_supported(RESULT result)

      #如果不支持，判断进不去
      if(result)
       #为工程foo设置过程间优化
       set_target_properties(foo PROPERTIES INTERPROCEDURAL_OPTIMIZATION TRUE)
      endif()
      ```

  - CMake的option简介

    - option命令可以设置默认值

      ```
      option(address "this is path for value" ON)
      ```

    - 命令表示，当用户没有设置address的时候，默认值就是ON，当用户显示的设置address的时候，address里面就是用户设置的值
    - 注意：加入有一些变量依赖了address，但是这些变量的使用在option语句之前，此时对于这些变量来说，address还是属于没有定义的。
    - 在用户没有提供ON或者OFF的时候，默认是OFF。如果option有改变，一定要清理CMakeCache.txt文件和CMakeFiles文件夹
    - CMake编译选项的管理

- 在工程的根目录，编写CMakeLists.txt，另外单独创建option.txt文件，专门管理编译选项
- 在CMakeLists.txt中加入：

  ```
  include option.txt
  ```
- 在option.txt中添加：

  ```
  /*USE_MYMATH 为编译开关，中间的字符串为描述信息，ON/OFF 为默认选项*/
  option (USE_MYMATH
          "Use tutorial provided math implementation" ON)
  ```
- 在编译之前，执行ccmake . 就会弹出cmake GUI，进行配置所有的编译开关，配置结束之后会生成一个CMakeCache.txt，配置后的编译选项会保存在这个文件中

  https://blog.csdn.net/haima1998/article/details/23352881

- 怎么生成依赖于其他option的option

  ```
  #设置option：USE_CURL
  option(USE_CURL "use libcurl" ON)

  #设置option：USE_MATH
  option(USE_MATH "use libm" ON)

  #设置一个option：DEPENT_USE_CURL,第二个参数是他的说明，ON后面的参数是一个表达式，当“USE_CURL”且“USE_MATH”为真的时候，DEPENT_USE_CURL取ON，为假取OFF
  cmake_dependent_option(DEPENT_USE_CURL "this is dependent on USE_CURL" ON
  "USE_CURL;NOT USE_MATH" OFF)
  ```
- 属性调试模块（CMakePrintHelpers）

  ```
  CMAKE_PRINT_PROPERTIES([TARGETS target1 .. targetN]
                         [SOURCES source1 .. sourceN]
                         [DIRECTORIES dir1 .. dirN]
                         [TESTS test1 .. testN]
                         [CACHE_ENTRIES entry1 .. entryN]
                         PROPERTIES prop1 .. propN )
  ```
  - 如果要检查foo目标的INTERFACE_INCLUDE_DIRS和LOCATION的值，则执行：

    ```
    cmake_print_properties （ TARGETS foo
      PROPERTIES
        INTERFACE_INCLUDE_DIRS
        LOCATION ）
    ```
- //CMake3.8以上叫做Modern CMake