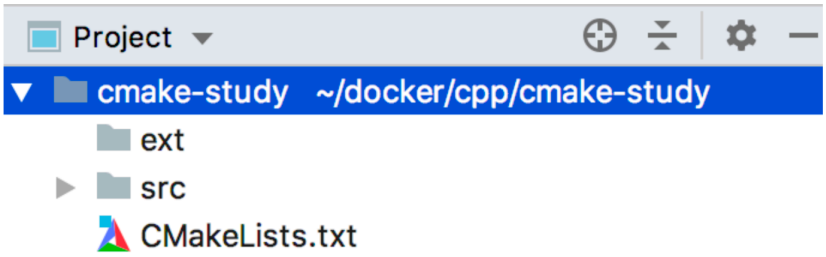
CMake 是一个跨平台的、开源的构建工具。`cmake` 是 `makefile` 的上层工具，它们的目的正是为了产生可移植的makefile，并简化自己动手写makefile时的巨大工作量。



c/c++ 项目工程部署如下：

- `src`：源码工程目录
- `ext`：第三方依赖库文件与头文件
- `CMakeLists.txt`：cmake 构建配置文件

简单案例

源文件编写：`src/main.cc`

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

编写CMAKE 配置文件 `CMakeLists.txt`

```
# cmake 最低版本需求
cmake_minimum_required(VERSION 3.13)

# 工程名称
project(cmake_study)

# 设置
set(CMAKE_CXX_STANDARD 11)

# 编译源码生成目标
add_executable(cmake_study src/main.cc)
```

cmake 命令便按照 `CMakeLists` 配置文件运行构建`Makefile`文件

```
$ mkdir build
$ cd build/
$ cmake ..
```

为了不让编译产生的中间文件污染我们的工程，我们可以创建一个 `build` 目录进入执行 `cmake` 构建工具. 如果没有错误， 执行成功后会在 `build` 目录下产生 `Makefile` 文件。然后我们执行 `make` 命令就可以编译我们的项目了。以上就是大致的 cmake 构建运行过程。**从上面的过程可以看出，cmake 的重点在配置 `CMakeLists.txt` 文件。**

CMakeLists 变量篇

我们可以使用 `SET(set)` 来定义变量：

- **语法**： `SET(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE])`
- **指令功能**: 用来显式的定义变量
- **例子**: `SET (SRC_LST main.c other.c)`
- **说明**: 用变量代替值，例子中定义 `SRC_LST` 代替后面的字符串。

可以使用 `${NAME}` 来获取变量的名称。

cmake 常用变量

环境变量名	描述
CMAKE_BINARY_DIR, PROJECT_BINARY_DIR, `<projectname>`_BINARY_DIR	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。PROJECT_BINARY_DIR 跟其他指令稍有区别,现在,你可以理解为他们是一致的。
CMAKE_SOURCE_DIR, PROJECT_SOURCE_DIR, `<projectname>`_SOURCE_DIR	工程顶层目录。
CMAKE_CURRENT_SOURCE_DIR	当前处理的 CMakeLists.txt 所在的路径,比如上面我们提到的 src 子目录。
CMAKE_CURRRENT_BINARY_DIR	如果是 `in-source` 编译,它跟 CMAKE_CURRENT_SOURCE_DIR 一致,如果是 `out-of-source` 编译,他指的是 target 编译目录。
EXECUTABLE_OUTPUT_PATH , LIBRARY_OUTPUT_PATH	最终目标文件存放的路径。
PROJECT_NAME	通过 PROJECT 指令定义的项目名称。

cmake 系统信息

系统信息变量名	描述
CMAKE_MAJOR_VERSION	CMAKE 主版本号,比如 2.4.6 中的 2
CMAKE_MINOR_VERSION	CMAKE 次版本号,比如 2.4.6 中的 4
CMAKE_PATCH_VERSION	CMAKE 补丁等级,比如 2.4.6 中的 6
CMAKE_SYSTEM	系统名称,比如 Linux-2.6.22
CMAKE_SYSTEM_NAME	不包含版本的系统名,比如 Linux
CMAKE_SYSTEM_VERSION	系统版本,比如 2.6.22
CMAKE_SYSTEM_PROCESSOR	处理器名称,比如 i686.
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

cmake 编译选项

编译控制开关名	描述
BUILD_SHARED_LIBS	使用 `ADD_LIBRARY` 时生成动态库
BUILD_STATIC_LIBS	使用 `ADD_LIBRARY` 时生成静态库
CMAKE_C_FLAGS	设置 C 编译选项,也可以通过指令 ADD_DEFINITIONS()添加。
CMAKE_CXX_FLAGS	设置 C++编译选项,也可以通过指令 ADD_DEFINITIONS()添加。

CMake 常用指令

ADD_DEFINITIONS:

- 语法：`ADD_DEFINITIONS(-DENABLE_DEBUG -DABC)`
- 作用：向 C/C++编译器添加 `-D` 定义. 如果你的代码中定义了`#ifdef ENABLE_DEBUG #endif`,这个代码块就会生效。

ADD_DEPENDENCIES

- 语法: `ADD_DEPENDENCIES(target-name depend-target1 depend-target2 ...)`
- 作用：定义 target 依赖的其他 target, 确保在编译本 target 之前,其他的 target 已经被构建。

ADD_EXECUTABLE

- 语法：`ADD_EXECUTABLE(<name> [source1] [source2 ...])`
- 利用源码文件生成目标可执行程序。

ADD_LIBRARY

- 语法：`ADD_LIBRARY(<name> [STATIC | SHARED | MODULE] [source1] [source2 ...])`
- 根据源码文件生成目标库。`STATIC`,`SHARED` 或者 `MODULE` 可以指定要创建的库的类型。 STATIC 库是链接其他目标时使用的目标文件的存档。 SHARED库是动态链接的，并在运行时加载。

ADD_SUBDIRECTORY

- 语法：`ADD_SUBDIRECTORY(NAME)`
- 添加一个文件夹进行编译，该文件夹下的 CMakeLists.txt 负责编译该文件夹下的源码. NAME是想对于调用add_subdirectory的CMakeListst.txt的相对路径.

ENABLE_TESTING

- 语法: `ENABLE_TESTING()`
- 控制 Makefile 是否构建 test 目标,涉及工程所有目录。 一般情况这个指令放在工程的主 CMakeLists.txt 中。

ADD_TEST

- 语法：`ADD_TEST(testname Exename arg1 arg2 ...)`
- `testname` 是自定义的 test 名称,`Exename` 可以是构建的目标文件也可以是外部脚本等等。 后面连接传递给可执行文件的参数。 如果没有在同一个 CMakeLists.txt 中打开`ENABLE_TESTING()`指令, 任何 `ADD_TEST` 都是无效的。

AUX_SOURCE_DIRECTORY

- 语法：`AUX_SOURCE_DIRECTORY(dir VARIABLE)`
- 作用是发现一个目录下所有的源代码文件并将列表存储在一个变量中,这个指令临时被用来自动构建源文件列表。因为目前 cmake 还不能自动发现新添加的源文件。比如：

```
AUX_SOURCE_DIRECTORY(. SRC_LIST)
ADD_EXECUTABLE(main ${SRC_LIST})
```

CMAKE_MINIMUM_REQUIRED

- 语法：`CMAKE_MINIMUM_REQUIRED`
- 定义 cmake 的最低兼容版本 比如 `CMAKE_MINIMUM_REQUIRED(VERSION 2.5 FATAL_ERROR)` 如果 cmake 版本小与 2.5,则出现严重错误,整个过程中止。

EXEC_PROGRAM

- 在 CMakeLists.txt 处理过程中执行命令,并不会在生成的 Makefile 中执行。 具体语法为：

```
EXEC_PROGRAM(Executable [directory in which to run]
                 [ARGS <arguments to executable>]
                 [OUTPUT_VARIABLE <var>]
                 [RETURN_VALUE <var>])
```

- 用于在指定的目录运行某个程序,通过 ARGS 添加参数,如果要获取输出和返回值,可通过 OUTPUT_VARIABLE 和 RETURN_VALUE 分别定义两个变量。这个指令可以帮助你在 CMakeLists.txt 处理过程中支持任何命令,比如根据系统情况去修改代码文件等等。

FILE 指令

- 文件操作指令，语法：

```
FILE(WRITE filename "message to write"... )
 FILE(APPEND filename "message to write"... )
 FILE(READ filename variable)
 FILE(GLOB variable [RELATIVE path] [globbing expression_r_rs]...)
 FILE(GLOB_RECURSE variable [RELATIVE path] [globbing expression_r_rs]...)
 FILE(REMOVE [directory]...)
 FILE(REMOVE_RECURSE [directory]...)
 FILE(MAKE_DIRECTORY [directory]...)
 FILE(RELATIVE_PATH variable directory file)
 FILE(TO_CMAKE_PATH path result)
 FILE(TO_NATIVE_PATH path result)
```

CMake 控制指令

IF 指令

- 语法为：

```
if(<condition>)
  <commands>
elseif(<condition>) # optional block, can be repeated
  <commands>
else()              # optional block
  <commands>
endif()
```

- 条件可为：

```
#####

IF(var),如果变量不是:空,0,N, NO, OFF, FALSE, NOTFOUND 或<var>_NOTFOUND 时,表达式为真。
IF(NOT var ),与上述条件相反。
IF(var1 AND var2),当两个变量都为真是为真。
IF(var1 OR var2),当两个变量其中一个为真时为真。
IF(COMMAND cmd),当给定的 cmd 确实是命令并可以调用是为真。
IF(EXISTS dir)或者 IF(EXISTS file),当目录名或者文件名存在时为真。
IF(file1 IS_NEWER_THAN file2),当 file1 比 file2 新,或者 file1/file2 其中有一个不存在时为真,文件名请使用完整路径。
IF(IS_DIRECTORY dirname),当 dirname 是目录时,为真。
IF(variable MATCHES regex)
IF(string MATCHES regex)
```

FOREACH 指令

- 语法：

```
foreach(<loop_var> <items>)
  <commands>
endforeach()
```

其中`<items>`是以分号或空格分隔的项目列表。记录foreach匹配和匹配之间的所有命令endforeach而不调用。 一旦endforeach评估，命令的记录列表中的每个项目调用一次`<items>`。在每次迭代开始时，变量loop_var将设置为当前项的值。

WHILE 指令

- 语法：

```
while(<condition>)
  <commands>
endwhile()
```

while和匹配之间的所有命令 endwhile()被记录而不被调用。 一旦endwhile()如果被评估，则只要为`<condition>`真，就会调用记录的命令列表。

实战

多个目录，多个源文件

- 文件部署：

```
./Demo3
    |
    +--- main.cc
    |
    +--- math/
          |
          +--- MathFunctions.cc
          |
          +--- MathFunctions.h
```

- 顶层的`CMakeLists.txt`

```
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)
# 项目信息
project (Demo3)
# 查找当前目录下的所有源文件
# 并将名称保存到 DIR_SRCS 变量
aux_source_directory(. DIR_SRCS)
# 添加 math 子目录
add_subdirectory(math)
# 指定生成目标
add_executable(Demo main.cc)
# 添加链接库
target_link_libraries(Demo MathFunctions)
```

- math 中使用`CMakeLists.txt`生成静态库文件：

```
# 查找当前目录下的所有源文件
# 并将名称保存到 DIR_LIB_SRCS 变量
aux_source_directory(. DIR_LIB_SRCS)
# 生成链接库
add_library (MathFunctions ${DIR_LIB_SRCS})
```

添加第三方依赖库：现在工程需要第三方依赖库`jsoncpp`

- 文件部署为：

```
./cmake-start
  |
  +---src/main.cc
  +---ext/jsoncpp/
           |
           + ---include/
           + ---lib/
  +---CMakeLists.txt
```

- 顶层的CMakeLists：

```
cmake_minimum_required (VERSION 2.8)

project(cmake)

add_definitions(-std=gnu++11)

set(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)

include_directories(${CMAKE_SOURCE_DIR}/ext/jsoncpp/include)
link_directories(${CMAKE_SOURCE_DIR}/ext/jsoncpp/lib)
```

```
aux_source_directory(src SRC)

add_executable(demon ${SRC})

target_link_libraries(demon kmsjsoncpp)
```

自定义编译选项

CMake 允许为项目增加编译选项，从而可以根据用户的环境和需求选择最合适的编译方案。例如，可以将 MathFunctions 库设为一个可选的库，如果该选项为 `ON` ，就使用该库定义的数学函数来进行运算。否则就调用标准库中的数学函数库。

- **修改 CMakeLists 文件** 我们要做的第一步是在顶层的 CMakeLists.txt 文件中添加该选项：

```
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)
# 项目信息
project (Demo4)
# 加入一个配置头文件，用于处理 CMake 对源码的设置
configure_file (
  "${PROJECT_SOURCE_DIR}/config.h.in"
  "${PROJECT_BINARY_DIR}/config.h"
  )
# 是否使用自己的 MathFunctions 库
option (USE_MYMATH
       "Use provided math implementation" ON)
# 是否加入 MathFunctions 库
if (USE_MYMATH)
  include_directories ("${PROJECT_SOURCE_DIR}/math")
  add_subdirectory (math)
  set (EXTRA_LIBS ${EXTRA_LIBS} MathFunctions)
endif (USE_MYMATH)
# 查找当前目录下的所有源文件
# 并将名称保存到 DIR_SRCS 变量
aux_source_directory(. DIR_SRCS)
# 指定生成目标
add_executable(Demo ${DIR_SRCS})
target_link_libraries (Demo  ${EXTRA_LIBS})
```