# CMake学习笔记

CMake是一个开源的可扩展工具，用于独立于编译器的管理构建过程。CMake必须和本地构建系统联合使用，在每个源码目录中，需要编写CMakeLists.txt文件，以声明如何生成标准的构建文件（例如GNU Make的Makefiles，或者MSVS的解决方案）。

CMake支持所有平台的内部构建（in-source build）和外部构建（out-of-source build）。内部构建的源码目录和二进制目录为同一目录，即CMake会改变源码目录的内容。通过外部构建，可以针对单个源码树进行多重构建（Multiple builds ）。

CMake会生成一个方便用户编辑的缓存文件，当其运行时，会定位头文件、库、可执行文件，这些信息被收集到缓存文件中。用户可以在生成本地构建文件之前编辑它。

CMake命令行支持自动或者交互式的运行。CMake还提供了一个基于QT的GUI，其名称为cmake-gui。注意此GUI同样依赖于环境变量的正确设置。

CMakeLists.txt包含一系列的命令，每个命令都是 `COMMAND (args…)` 的形式，多个参数使用空白符分隔。CMake提供了很多预定义命令，你可以方便的扩展自己的命令。

CMake支持简单的变量，它们或者是字符串，或者是字符串的列表。引用一个变量的语法是 `${VAR_NAME}` 。

如果向一个命令传递列表变量，效果等同于向它逐个传递列表成员：

```
set(V 1 2 3)    # V的值是1 2 3
command(${V})   # 等价于command(1 2 3)
```

要把一个列表变量作为整体传递，只需要加上双引号即可：

```
command("${V}")   # 等价于command("1 2 3")
```

CMake可以直接访问环境变量和Windows注册表，前者使用语法 `$ENV{VAR}` ，后者使用语法 `[HKEY_CURRENT_USER\\SOFTWARE\\path;key]` 。示例：

```

```

```
# 读取环境变量
message(STATUS "LD_LIBRARY_PATH=$ENV{LD_LIBRARY_PATH}" )
# 设置环境变量
set(ENV{PATH} "/home/alex/scripts")
```

1. 支持多个底层构建工具，例如GNU Make、MSVC、XCode等等，可以生成这些构建工具需要的配置文件
2. 通过分析环境变量、Windows注册表等，自动搜索构建所需的程序、库、头文件
3. 支持创建复杂的命令
4. 很方便的在共享库、静态库两种构建方式之间切换
5. 自动生成、维护C/C++文件依赖关系，并且在大部分平台上支持并行构建

在开发跨平台软件时，CMake具有以下额外优势：

1. 可以测试机器字节序和其它硬件特性
2. 统一的构建配置文件
3. 支持依赖于机器特定信息的配置，例如文件的位置

Shell

```
# Ubuntu
sudo apt-get install cmake
# Redhat
yum install cmake
# Mac OS X with Macports
sudo port install cmake
# Window https://cmake.org/files/v3.5/cmake-3.5.2-win32-x86.zip
```

C++源码：

C++

```
#include <iostream>

using namespace std;

int main() {
    return 0;
}
```

要通过CMake编译上述文件，需要在同一目录下放置CMakeLists.txt文件：

CMakeLists.txt

```
# 需要最小的CMake版本
cmake_minimum_required(VERSION 3.3)
# 工程的名称，会作为MSVS的Workspace的名字
project(intellij_taste)

# 全局变量：CMAKE_SOURCE_DIR CMake的起始目录，即源码的根目录
# 全局变量：PROJECT_NAME 工程的名称
# 全局变量：PROJECT_SOURCE_DIR 工程的源码根目录的完整路径

# 全局变量：构建输出目录。默认的，对于内部构建，此变量的值等于CMAKE_SOURCE_DIR；否则等于构建树的根目录
set(CMAKE_BINARY_DIR ${CMAKE_SOURCE_DIR}/bin)  # ${}语法用于引用变量
# 全局变量：可执行文件的输出路径
set(EXECUTABLE_OUTPUT_PATH ${CMAKE_BINARY_DIR})
# 全局变量：库文件的输出路径
set(LIBRARY_OUTPUT_PATH ${CMAKE_BINARY_DIR})
# 设置头文件位置
include_directories("${PROJECT_SOURCE_DIR}")
# 设置C++标志位
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
# 设置源文件集合
set(SOURCE_FILES main.cpp)
# 添加需要构建的可执行文件，第二个以及后续参数是用于构建此文件的源码文件
add_executable(intellij_taste ${SOURCE_FILES})
```

在上述目录中执行下面两条命令，即可执行构建：

Shell

```
# 创建一个build子目录作为构建树
mkdir build && cd build && cmake .. && cd ..

# 在build/bin子目录中生成可执行文件:
# cmake --build <dir> [options] [-- [native-options]]
cmake --build build -- -j3  # --表示把其余选项传递给底层构建工具

# 注意，亦可使用底层构建系统，例如make命令或者MSVC的IDE
cd build
make -j3
```

CMake包含一系列重要的概念抽象，包括目标（Targets）、生成器（Generators）、命令（Commands）等，这些命令均被实现为C++类。理解这些概念后才能编写高效的CMakeLists文件。

下面列出这些概念之间的基本关系：

1. 源文件：对应了典型的C/C++源代码
2. 目标：多个源文件联合成为目标，目标通常是可执行文件或者库
3. 目录：表示源码树中的一个目录，常常包含一个CMakeLists.txt文件，一或多个目标与之关联
4. 本地生成器（Local generator）：每个目录有一个本地生成器，负责为此目录生成Makefiles，或者工程文件
5. 全局生成器（Global generator）：所有本地生成器共享一个全局生成器，后者负责监管构建过程，全局生成器由CMake本身创建并驱动

CMake的执行开始时，会创建一个cmake对象并把命令行参数传递给它。cmake对象管理整体的配置过程，持有构建过程的全局信息（例如缓存值）。cmake会依据用户的选择来创建合适的全局生成器（VS、Makefiles等等），并把构建过程的控制权转交给全局生成器（调用configure和generate方法）。

全局生成器负责管理配置信息，并生成所有Makefiles/工程文件。一般情况下全局生成器把具体工作委托给本地生成器执行，全局生成器为每个目录创建一个本地生成器。全局/本地生成器的分工取决于实现，例如：

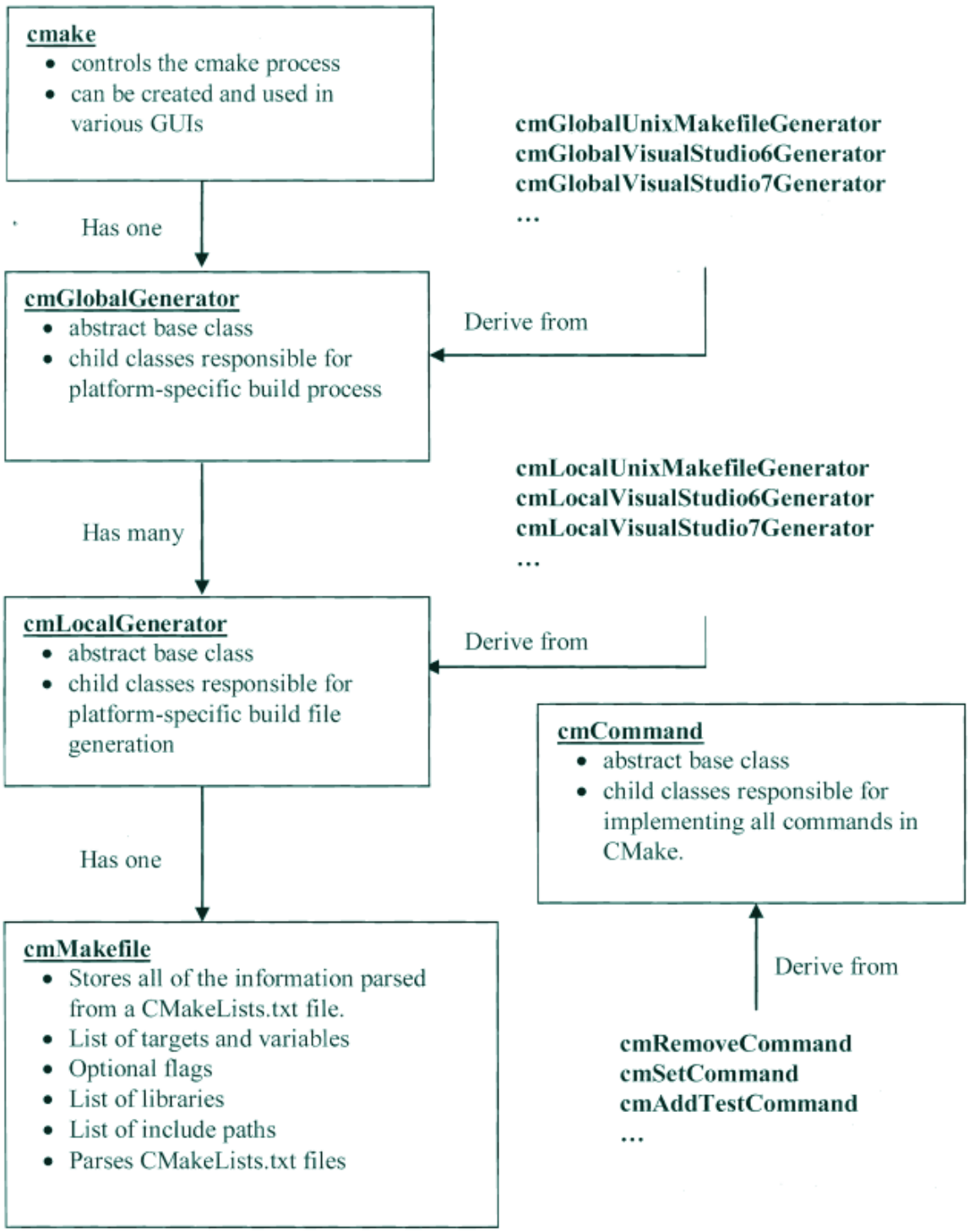
1. 对于VS，全局生成器负责生成解决方案文件，本地生成器负责每个目标的工程文件
2. 对于Makefiles，全局生成器生成总体的Makefile，本地生成器则负责生成大部分Makefile

每个本地生成器包含一个cmMakefile对象，其中存放CMakeList.txt的解析结果。

CMake的每一个命令也被实现为C++类，该类主要包括两个成员：

| 成员 | 说明 |
| --- | --- |
| InitialPass() | 接受当前目录的cmMakefile对象、命令参数作为入参。命令的执行结果存放在cmMakefile对象中 |
| LastPass() | 在整个CMake工程所有命令的InitialPass()都执行后再执行。大部分命令不实现此方法 |

下图显示cmake、生成器、cmMakefile、命令等类型的关系：




cmMakefile对象中存放的最重要的对象是目标（Targets），目标代表可执行文件、库、实用工具等。每个 `add_library` 、 `add_executable` 、 `add_custom_target` 命令都会创建一个目标。

下面的语句创建一个库目标：

```
# 创建一个静态库，包含两个源文件
add_library(foo STATIC foo1.c foo2.c)
```

上述命令声明的foo可以作为库名称在工程的任何地方使用。CMake知道如何将此名称转换为库文件。命令的（可选的）第二个参数声明库的类型，有效值包括：

| 库类型 | 说明 |
| --- | --- |
| STATIC | 目标必须构建为静态库 |
| SHARED | 目标必须构建为共享库 |
| MODULE | 目标必须构建为支持在运行时动态加载到可执行文件中的模块<br>对于除了Mac OS X之外的系统，此取值等价于SHARED |

如果不声明库类型，则CMake依据变量 `BUILD_SHARED_LIBS` 判断应该构建为共享库还是静态库，如果此变量不设置，构建为静态库。

与库目标类似，可执行目标也可以指定特定的选项，例如WIN32会导致操作系统调用WinMain而不是main函数。

使用 `set_target_properties` 或者 `get_target_properties` 命令，或者更通用的 `set_property` 、`get_property` 命令，可以读写目标的属性，示例：

```
# 修改目录使用的头文件目录，注意，全局的include_directories会此目标忽略
set_property (TARGET jsonrpc
  PROPERTY INCLUDE_DIRECTORIES
    ${JSONCPP_INCLUDE_DIRS}
    ${Boost_INCLUDE_DIRS}
    ${CMAKE_CURRENT_SOURCE_DIR}
    ${CMAKE_CURRENT_SOURCE_DIR}/jsonrpc
)

# 同时设置多个属性
set_target_properties(jsonrpc PROPERTIES VERSION ${PROJECT_VERSION} SOVERSION ${PROJECT_VER
```

常用目标属性如下表：

| 属性 | 说明 |
| --- | --- |
| LINK_FLAGS | 指定链接标记 |
| COMPILE_FLAGS | 指定编译标记 |
| INCLUDE_DIRECTORIES | 指定目标需要引用的头文件目录 |

| 属性 | 说明 |
| --- | --- |
| PUBLIC_HEADER | 共享的库目标提供的公共头文件 |
| VERSION<br>SOVERSION | 对于共享库来说，VERSION、SOVERSION允许让你分别设置构建版本、API版本。通常SOVERSION更加稳定不变<br>如果设置了NO_SONAME属性，则SOVERSION属性被自动忽略 |
| OUTPUT_NAME | 目标输出文件名称 |

全部目标属性请参考官网 (https://cmake.org/cmake/help/v3.0/manual/cmake-properties.7.html#properties-on-targets)。

使用 `target_link_libraries` 命令，可以指定目标需要链接的库的列表。列表的元素可以是库、库的全路径、通过add_library命令添加的库名称。

对于声明的每个库，CMake会跟踪其依赖的所有其它库，这种依赖关系需要用上述命令来设置：

```
add_library(foo foo.cpp)
#foo库依赖于bar库
target_link_libraries(foo bar)

add_executable(foobar foobar.cpp)
#foobar显式依赖foo，隐式依赖bar，后两者都会被链接到foobar中
target_link_libraries(foobar foo)
```

和Target类似，源文件也被建模为C++类，也支持读写属性（通过set_source_files_properties、get_source_files_properties或更加一般的命令）。最常用属性包括：

| 属性 | 说明 |
| --- | --- |
| COMPILE_FLAGS | 针对特定源文件的编译器标记，可以包含-D、-I之类的标记 |
| GENERATED | 指示此文件是否在构建过程中生成，这种文件在CMake首次运行时不存在，因而计算依赖关系时要特殊考虑 |
| OBJECT_DEPENDS | 添加此源文件额外依赖的其它文件。CMake会自动分析C、C++的源文件依赖，因而此选项很少使用 |
| WRAP_EXCLUDE | CMake不直接使用该属性。但是某些命令和扩展读取该属性，判断何时/如何把C++类包装到其它语言，例如Python |

其它偶尔可能用到的CMake类型包括Directory、Generator、Test、Property等。Directory、Generator、Test的实例同样（与目录、源文件类似）关联属性。

属性是一种键值存储，它关联到一个对象。读写属性最一般的方法是上面提到的get/set_property命令。所有可用的属性可以通过 `cmake -help-property-list` 得到。

目录的属性包括：

| 属性 | 说明 |
| --- | --- |
| ADDITIONAL_MAKE_CLEAN_FILES | 指定一系列需要在make clean时清除掉的文件的列表<br>默认的CMake会清除所有生成的文件 |
| EXCLUDE_FROM_ALL | 指示此目录和子目录中所有的目标，是否应当从默认构建中排除<br>子目录的IDE工程文件/Makefile将从顶级IDE工程文件/Makefile中排除 |
| LISTFILE_STACK | 最要在调试CMake脚本时用到，列出当前正在被处理的文件的列表 |

目录和生成器对象会在CMake处理你的源码树时自动创建。

CMakeLists中的变量和普通编程语言中的变量很类似，变量的值要么是单个值，要么是列表。CMake自动定义一系列重要的变量。

要引用变量，必须使用 `${VARNAME}` 语法，要设置变量的值，需要使用set命令。

CMake中变量的作用域和普通编程语言略有不同，当你设置一个变量后，变量对当前CMakeLists文件、当前函数、以及子目录的CMakeLists、任何通过 `INCLUDE` 包含进来的文件、任何调用的宏或函数可见。

当处理一个子目录、调用一个函数时，CMake创建一个新的作用域，其复制当前作用域全部变量，在子作用域中对变量的修改不会对父作用域产生影响。要修改父作用域中的变量，可以在set时指定特殊选项：

```
set (name Alex PARENT_SCOPE)
```

变量的值可以是一个列表，这样的变量可以被展开为多个值：

```
set(fruit apple peach strawberry )
foreach(f ${fruit})
    message("Do you want ${f}")
endforeach()
```

常用变量如下表：

| 变量 | 说明 |
| --- | --- |

| 变量 | 说明 |
| --- | --- |
| CMAKE_C_FLAGS | C编译标记，示例：`set(CMAKE_C_FLAGS "-std=c11 -pthread")` |
| CMAKE_CXX_FLAGS | C++编译标记 |
| CMAKE_C_FLAGS_DEBUG | 用于Debug配置的C编译标记，示例：`set(CMAKE_C_FLAGS_DEBUG "-g -O0")` |
| CMAKE_CXX_FLAGS_DEBUG | 用于Debug配置的C++编译标记 |
| CMAKE_C_FLAGS_RELEASE | 用于Release配置的C编译标记 |
| CMAKE_CXX_FLAGS_RELEASE | 用于Release配置的C++编译标记 |

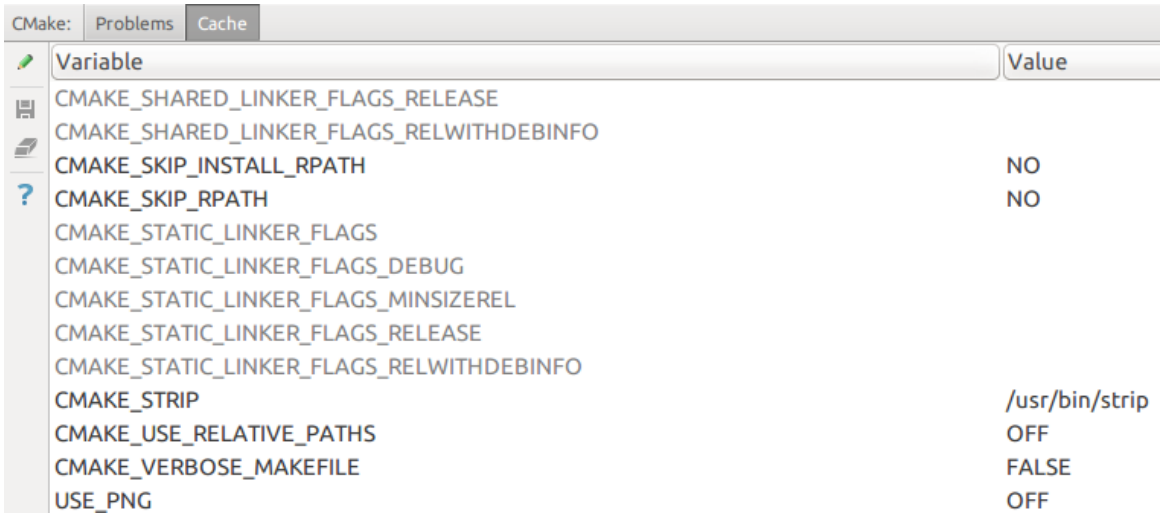
有些时候你可能期望用户通过CMake的UI输入一些变量的值， 这时变量必须作为缓存条目。当CMake运行时，它会向二进制目录输出缓存文件（Cache file），缓存文件中的变量值通过CMake的UI展示给用户。

使用这种缓存的目的之一是，存储用户的选项，避免重新运行CMake时，反复要求用户输入相同的信息。

`option` 命令可以创建一个Boolean变量（ON/OFF）并将其存储在缓存中：

```
1 option(USE_PNG "Do you want to use the png library?")
```

用户可以通过UI设置USE_PNG的值，并且在未来这一值会保存在缓存中。使用CLion作为IDE时，可以在CMake窗口中点击Cache选项卡，查看或者编辑缓存条目：




除了 `option` 命令之外， `find_file` 也可以用来创建缓存条目。为 `set` 命令指定特殊参数，亦可创建缓存条目：

```
# CACHE选项表示此变量作为缓存条目
# ON为默认值
# BOOL为变量类型，支持BOOL、PATH、FILEPATH、STRING
set(USE_PNG ON CACHE BOOL "Do you want to use the png library?")
```

缓存条目的另外一个目的是， 存储那些难以确定的关键变量，这些变量可能对用户不可见。通常这些变量是系统相关的，例如 `CMAKE_WORDS_BIGENDIAN` 。这类值可能需要CMake编译并运行一个程序来确定，一旦确定，即缓存。

位于缓存中的变量具有一个属性指示它是否为“进阶的”（advanced），默认的CMake GUI隐藏进阶条目。要标记一个缓存条目为进阶的，可以：

```
mark_as_advanced(VAR_NAME)
```

某些情况下，你可能需要限制缓存条目的值范围在一个有限的集合中，这是可以设置条目的 `STRINGS` 属性，提供值列表。在GUI中，这种条目的字段会展示为下拉列表：

```
# 设置名为CRYPT_BACKEND的缓存条目的值为Open SSL
set(CRYPT_BACKEND "Open SSL" CACHE STRING)
# 设置上述缓存条目的取值范围
set_property(CACHE CRYPT_BACKEND PROPERTY STRINGS "Open SSL" "LibDES")
```

即使变量存在于缓存，你仍然可以在CMakeLists中覆盖它（改变作用域中此变量的值）。只需要不带CACHE选项调用set命令，即可覆盖缓存中同名变量的值。

另一方面，一旦变量值已经缓存，你一般无法在CMakeLists中改变缓存的值（与上述覆盖是两回事）。也就是说，当缓存中有VARNAME时， `set(VARNAME ON CACHE BOOL )` 不会有任何作用。要强制改变缓存中的值并覆盖当前作用域的值，可以联合使用 `FORCE` 和 `CACHE` 选项。

构建配置允许工程使用不同方式构建：debug、optimized或者任何其它标记。CMake默认支持四种构建配置：

| 构建配置 | 说明 |
| --- | --- |
| Debug | 启用基本的调试（编译器的）标记 |
| Release | 基本的优化配置 |
| MinSizeRel | 生成最小化的，但不一定是最快的代码 |
| RelWithDebugInfo | 优化构建，但是同时携带调试信息 |

依据生成器的不同，CMake处理构建配置的方式有所差异，CMake尽可能遵循底层本地构建系统的约定，这意味着使用Makefiles、VS时构建配置影响构建的方式有所不同：

1. VS支持构建配置的概念，在IDE中你可以选择Debug、Release配置，CMake只需要对接到VS的构建配置即可

2. Makefile默认同时（CMake运行时）只能有一种配置被激活。使用 `CMAKE_BUILD_TYPE` 变量可以指定目标配置。如果此变量为空，则不给构建添加额外标记。如果此变量设置为上面四种构建配置之一，则相应的变量、规则——例如 `CMAKE_CXX_FLAGS_<CONFIGNAME>` 被添加到compile line中。可以使用下面的方式来分别基于Debug、Release配置进行构建：

```Shell
# 创建工程目录的两个兄弟目录，CD到其中分别执行：
cmake ../project -DCMAKE_BUILD_TYPE:STRING=Debug
cmake ../project -DCMAKE_BUILD_TYPE:STRING=Release
```

CMake由CMakeLists.txt驱动，此文件包含构建需要的一切信息。

除了用于分隔命令参数，其余空白符一律被忽略。反斜杠可以用来指示转义字符。

| 命令 | 说明 |
| --- | --- |
| project | 顶层CMakeLists.txt中应当包含的第一个命令，声明工程的名字和使用的编程语言：<br>`project (projectname, [CXX], [C], [JAVA], [NONE])`<br>如果不指定语言，默认CMake启用C/C++，如果指定为CXX则C语言的支持自动加入<br>对于工程中出现的每个project命令，CMake会创建一个顶级的IDE工程文件（或Makefile文件）。此工程文件中会包含：<br>1. 所有CMakeLists.txt中声明的目标<br>2. 所有通过 `add_subdirectory` 命令添加的子目录。如果为命令指定 `EXCLUDE_FROM_ALL` 选项，则此工程文件/Makefile不会包含到顶级工程文件/Makefile中，对于那种需要从主构建流传中排除的子工程（例如examples子工程），这个选项有用 |
| set | 设置变量值或列表 |
| remove | 从变量值的列表中移除一个单值 |
| separate_arguments | 基于空格，把单个字符串分隔为列表 |
| add_executable | 定义目标（可执行文件/库），以及目录由哪些源文件组成<br>对于VS，源文件将会出现在IDE中，但是默认的项目使用的头文件不会包含在IDE中，要改变此行为，只需要将头文件添加到源文件列表中 |
| add_library | |

和普通编程语言一样，CMake支持条件、循环控制结构，同时支持子过程（macro、function）

```
if (FOO)
else(FOO)
endif(FOO)
# 上面把if的条件在else、endif中重复，这是可选的。因此我们可以简单的写作：
if (FOO)
else()
endif()
```

在else、endif上重复条件，有助于if-else-endif匹配检查，特别是多层嵌套时。

CMake同样支持elseif：

```
if(MSVC80)
  #...
elseif(MSVC90)
  #...
elseif(APPLE)
  #...
endif()
```

条件命令支持受限的表达式语法，如下表所列：

| 语法 | 说明 |
| --- | --- |
| if ( variable ) | 当if命令参数的值不是：0、FALSE、OFF、NO、NOTFOUND、*-NOTFOUND、IGNORE时，表达式的值为真，注意不区分大小写<br>variable可以不用${}包围 |
| if ( NOT variable ) | 上面取反<br>variable可以不用${}包围 |
| if ( variable1 AND variable2 ) | 逻辑与，所有逻辑操作支持用括号来提升优先级 |
| if ( variable1 OR variable2 ) | 逻辑或 |
| if ( num1 EQUAL num2 ) | 数字相等比较，其它操作符包括LESS、GREATER |
| if ( str1 STREQUAL str2 ) | 字典序相等比较，其它操作符包括STRLESS、STRGREATER |
| if ( v1 VERSION_EQUAL v2) | `marjor[.minor[.patch[.tweak]]]` 风格的版本号相等比较，其它操作符包括VERSION_LESS、VERSION_GREATER |

| 语法 | 说明 |
| --- | --- |
| if ( COMMAND commandname ) | 如果指定的命令可以调用 |
| if ( DEFINED variable ) | 如果指定的变量被定义，不管它的值真假 |
| if ( EXISTS file-name ) | 如果指定的文件或者目录存在 |
| if ( IS_DIRECTORY name ) | 如果给定的name是一个目录 |
| if ( IS_ABSOLUTE name ) | 如果给定的name是一个绝对路径 |
| if ( n1 IS_NEWER_TAN n2 ) | 如果文件n1的修改时间大于n2 |
| if ( variable MATCHES regex )<br><br>if ( string MATCHES regex ) | 如果给定的变量或者字符串匹配正则式：<br>`set(name Alex)`<br>`if(${name} MATCHES A.*x)`<br>`  message(${name})`<br>`endif()` |

CMake操作符优先级从高到底：

1. 括号分组：()
2. 前缀一元操作符：EXISTS、COMMAND、DEFINED
3. 比较操作符：EQUAL、LESS、GREATER及其变体，以及MATCHES
4. 逻辑非：NOT
5. 逻辑或于：AND、OR

```
foreach (item  list)
    # do something with item
endforeach (item)
```

此命令用于迭代一个列表，第一个参数是每次迭代使用变量的名称，其余参数为被迭代的列表

注意，在循环内部，你可以使用迭代变量构造另外一个变量的名字，例如 `${NAME_OF_${item}}`

此命令用于基于条件的迭代：

```
while(${COUNT} LESS 2000)
    set(TASK_COUNT, ${COUNT})
endwhile()
```

此命令用于中断foreach/while循环。

CMake中的函数很类似于C/C++函数。你可以向函数传递参数，除了依据形参名外，你还可以使用 `ARGC` 、 `ARGV` 、 `ARGN` 、 `ARG0` 、 `ARG1` ...等形式，在函数内部访问入参。

函数内部是一个新作用域，类似于add_subdirectory生成的新作用域一样，函数调用前的作用域被拷贝并传递到函数内部，函数返回时，新作用域消失。

函数的第一个形参是函数的名称，其它参数构成传统的形参列表：

```
function(println msg)
    message(${msg} "\n")
    set ( msg ${msg} PARENT_SCOPE )  #设置父作用域中变量的值
endfunction()

println(Hello)
```

此命令拥有从函数中返回，或者在listfile命令中提前结束。

宏于函数类似，但是宏不会创建新的作用域。传递给宏的参数也不被作为变量看待，而是在执行宏前替换为字符串：

```
macro (println msg)       #同样的，括号中第一个项目是宏的名称
    message(${msg} "\n")
endmacro()
```

对于宏， ARGC、ARG0、ARG1等也可以使用。ARG0代表传递给宏的第一个参数。

CMake是一个不断进化的工具，随着新版本的推出，会不断有新的命令被加入。很多时候，我们需要检查当前CMake版本是否支持某些特性。

我们可以使用if命令判断某个命令是否可用：

```
if(COMMAND some_new_command)
    #...
endif()
```

或者直接检查CMake的版本：

```
if (${CMAKE_VERSION} VERSION_GREATER 1.6.1)
endif()
```

另外，还可以声明要求的最低的CMake版本：

```
cmake_minimum_required(VERSION 2.8)
```

所谓模块，仅仅是存放到一个文件中，一系列CMake命令的集合。我们可以用 `include` 命令将模块包含到CMakeLists.txt中。举例：

```
# 此模块用于查找TCL库
include (FindTCL)
# 找到后，将其加入到链接依赖中
target_link_libraries (FOO ${TCL_LIBRARY})
```

包含一个模块时，可以使用绝对路径，或者是基于CMAKE_MODULE_PATH的相对路径，如果此变量未设置，默认为CMake的安装目录的Modules子目录。

模块依据用途的不同可以分为：

| 类别 | 说明 |
| --- | --- |
| 查找类模块 | 查找软件元素——例如头文件、库——的位置<br>CMake提供了大量这类模块，如果目录库/头文件找不到，模块往往提供一个缓存条目，便于用户手工指定<br>下面是一个查找PNG模块的例子：<br>`# png库依赖于zlib`<br>`include(FindZLIB)  # 查找zlib库`<br>`if (ZLIB_FOUND)    # 往往在找到后设置LIBNAME_FOUND变量`<br>`    # 查找头文件位置并存入变量`<br>`    find_path(PNG_PNG_INCLUDE_DIR png.h /usr/local/include /usr/include)`<br>`    # 查找库文件位置并存入变量`<br>`    find_library(PNG_LIBRARY png /usr/lib /usr/local/lib)`<br>`    if (PNG_LIBRARY AND PNG_PNG_INCLUDE_DIR)`<br>`        # 合并ZLIB头文件和库到PNG的`<br>`        set(PNG_INCLUDE_DIR ${PNG_PNG_INCLUDE_DIR} ${ZLIB_INCLUDE_DIR})`<br>`        set(PNG_LIBRARIES ${PNG_LIBRARY} ${ZLIB_LIBRARY})`<br>`        # 设置已找到标记`<br>`        set(PNG_FOUND YES)`<br>`    endif ()`<br>`endif ()` |
| 系统探测模块 | 探测系统的特性，例如浮点数长度、对ASCI C++刘的支持<br>很多这类模块具有Test、Check前缀，例如TestBigEndian、CheckTypeSize |
| 实用工具模块 | 用于添加额外的功能，例如处理一个CMake工程依赖于其它CMake工程的情况 |

由于某些原因，在版本升级后，CMake可能不提供完全的向后兼容。这意味着使用新版的CMake处理基于旧版本的CMakeLists.txt时会出现问题。CMake引入策略这一特性，帮助用户和开发者处理此向后兼容问题。

策略机制实现以下目标：

1. 既有的工程能够用任何比CMakeLists作者使用的、更新版本的CMake构建。用户不应该需要修改CMakeLists代码，但是可能出现警告信息
2. 新特性的修正，老接口的Bug修复应当被执行，而非因向后兼容性的要求而搁置
3. 任何对CMake的改变，会导致CMakeLists文件必须更改的，应当加以文档说明。每个这样的改变应当具有唯一的标识符以便查阅文档，改变仅在工程提示自己支持的情况下才启用
4. 最终将会移除向后兼容性的代码，不再支持古老版本的CMake。因此而构建失败的工程必须得到有价值的错误提示

CMake中的所有策略被分配一个 `CMPNNNN` 形式的名称，其中NNNN是一个整数值编号。策略同时支持出于兼容性目的的旧行为，以及“正确的”新行为。每个策略包含出现动机、新旧行为的详细说明文档。

可以在工程中对每个策略进行配置，设置其值为NEW或者OLD，CMake将遵从测量设置，从而表现出不同的构建行为。

设置策略有几种方式，最简单的是设置策略为特定的CMake版本： `cmake_policy(VERSION 2.6)` 。这样所有2.6版本之前引入的策略都被标记为NEW，而2.6之后引入的策略则标记为“未设置”，以便产生警告信息。

注：cmake_minimum_required命令同样会设置策略，因此仅在需要定制子目录的策略时才以VERSION选项调用cmake_policy命令。

以SET选项调用cmake_policy可以明确的设置单个策略。以CMP0002为例，该策略的新行为要求所有逻辑目标具有全局独特的名字。下面的命令可以抑制存在重复目标名时的警告信息：

```
cmake_policy(SET CMP0002 OLD)
```

```
# 设置库的寻找目录
link_directories(/path/to)
add_executable(myexe myexe.c)
target_link_libraries (myexe A B)

# 或者
add_executable(myexe myexe.c)
# 使用绝对路径
target_link_libraries (myexe /path/to/libA.so /path/to/libB.so )
```

类Unix操作系统的系统库常常位于/usr/lib或者/lib目录。这些目录被链接器作为隐含的库搜索目录，因此 `find_library(M_LIB m)` 将从/usr/lib/libm.so定位到Math库。

问题是，某些平台会依据体系结构的不同，提供库的不同版本：

```
# IRIX机器
/usr/lib/libm.so         # ELF o32
/usr/lib32/libm.so       # ELF n32
/usr/lib64/libm.so       # ELF 64
# Solaris
/usr/lib/lim.so          # sparcv8架构
/usr/lib/sparcv9/lim.so  # sparcv9架构
```

find_library命令不知道各种体系结构特定的系统如何定义上面的目录规则，因此此命令可能找到不匹配的体系结构的库文件。

此问题的一个解决办法是让链接器自动寻找库所在目录（不使用link_directories或者指定绝对路径），不幸的是，此办法无法区分库的动态、静态版本。CMake实际使用的妥协做法是：

1. 存在于隐含库搜索目录中的库，且链接器支持类似-Bstatic的选项来指定使用静态库，使用-l选项传递库名称
2. 其它情况下，传递库绝对路径给链接器

共享库和可加载模块有利于重用：

1. 缩短compile/link/run周期
2. 共享库重新构建时，依赖于它的共享库/可执行文件甚至不需要重新构建
3. 减少磁盘和内存消耗，因为同一共享库只需要一份

相比静态库，共享库更像是可执行文件，大部分系统要求共享库上具有可执行权限。和可执行文件一样，共享库可以链接到其它共享库。

对于静态库来说，一个object文件是最小单元；而共享库（包括其依赖）本身是一个最小单元。链接器可以从静态库中挑出需要的object文件，但是对于共享库及其依赖的其它共享库，都需要存在。

共享库和静态库的另外一个不同是库的声明顺序，指定静态库时顺序很重要，因为大部分链接器仅仅遍历库列表一次来寻找符号，依赖其它静态库的静态库必须放在列表前面。

当决定在工程使用共享库时，开发者必须面对几个问题。

在大部分UNIX系统中，默认所有符号被导出。在Windows系统中，开发者必须明确告知编译器哪些符号被导入（使用符号时）/导出（创建符号时）

当从UNIX移植项目到Windows平台时，你可以：

1. 创建一个额外的.def文件，或者

2. 使用微软的C/C++语言扩展——`__declspec(dllexport)`、`__declspec(dllimport)`声明的符号分别被导出、导入

如果一个源文件在创建、使用一个库时都需要使用，则必须使用宏来处理。CMake在Windows下构建共享库（DDL）时，会自动定义宏`${LIBNAME}_EXPORTS`。我们可以利用此宏：

```C++
#if defined(WIN32)
    #if defined(vtkCommon_EXPORTS)
        #define VTK_COMMON_EXPORT __declspec(dllexport)
    #else
        #define VTK_COMMON_EXPORT __declspec(dllimport)
    #endif
#else
    #define VTK_COMMON_EXPORT
#endif
```

这样，VTK_COMMON_EXPORT在UNIX中为空白；在Windows下构建共享库时为__declspec(dllexport)。

UNIX和Windows存在一个重要的和符号需求相关的差异：Windows上的DLL需要完全解析，也就是在创建时必须链接所有符号；而UNIX允许共享库在运行时从可执行文件或者其它共享库中获取符号。因而在UNIX中，CMake会给可执行目标一个标记，允许它被共享库调用。

另外一个需要提及的关于C++全局对象的陷阱是，加载或者链接了C++共享库的main函数，必须基于C++的编译器来链接，否则cout之类的全局对象可能在使用时尚未初始化。

由于链接到共享库的可执行文件必须在运行时能找到这些库，特殊的环境变量或者链接器标记必须被使用。

不同系统都提供了工具，用以查看可执行文件实际上使用的是哪个库：

1. UNIX系统的`ldd`命令：显示可执行文件使用哪些库。在Mac OS X上使用`otool -L`
2. Windows系统的`depends`程序，功能类似

在很多UNIX系统中，可以使用环境变量`LD_LIBRARY_PATH`来告诉应用程序到哪里寻找库，而在Windows中，环境变量`PATH`同时用来寻找DLL和可执行文件。CMake会默认把运行时库的路径信息存放到可执行文件中，因此前述环境变量并不必须。但是某些时候你可能需要关闭这个特性，设置`CMAKE_SKIP_RPATH=false`即可。

使用共享库时，运行时加载的库，应当与链接时期望的库的“版本”一致，即功能上没有不兼容的变化。

如何识别这种变化并没有一致的规范，某些UNIX系统通过**soname**来版本化共享库，所谓soname就是在共享库名称后附加可选的数字后缀，例如libx.so.1。仅当共享库的接口发生不兼容变化时soname才改变，如果libx从1.0到1.9维持了一致性的接口，它们的soname应该一致。注意soname和文件名不是一回事，1.3版本的libx的文件名可能叫libx.so.1.3，但是它的soname可能是libx.so.1。

soname是共享库文件的一个头字段。当可执行文件和共享库链接时，共享库的soname也存放到可执行文件中，因此在运行时加载时，可执行文件可以根据soname来寻找版本匹配的共享库文件。

当安装libx库到系统时，可以建立符号链接 `libx.so -> libx.so.1` ，当libx出现不兼容升级时，则修改前述符号链接，例如 `libx.so -> libx.so.2` 。这样，新的程序总是和最新的libx版本进行链接。另一方面，这种符号链接用法可以扩展为，将某个soname链接到特定文件，例如 `libx.so.1 -> libx.so.1.3` ，如果1.3版本有一个BUG在1.3.2中修复，可以修改前述符号链接为 `libx.so.1 -> libx.so.1.3.2` ，这样基于libx.so.1链接的可执行文件在获得BUG修复的同时，能够找到正确的共享库。

CMake支持这种基于soname的版本号编码机制，只要底层平台支持soname，可以设置共享库目标的属性：

```
# VERSION，指定一个版本号，用于创建文件名
# SOVERSION，指定一个版本号，用于生成SONAME头
set_target_properties (x PROPERTIES VERSION 1.2 SOVERSION 4)
```

设置上述属性后，安装共享库时会产生如下文件和符号链接：

```
libx.so.1.2
libx.so.4 -> libx.so.1.2
libx.so -> libx.so.4
```

如果仅指定两个版本号中的一个，那么另外一个自动与之相同。

软件通常被安装到和源码、构建树无关的位置上。CMake提供一个 `install` 命令，来说明一个工程如何被安装。正确使用这个命令后：

1. 对于基于Makefile的生成器，用户只需要执行 `make install` 或者 `nmake install` 即可完成安装
2. 对于基于GUI的平台，例如XCode、VS，用户只需要构建INSTALL目标

对install的每一次调用都会指定某些安装规则，这些规则会依据命令调用的顺序被执行。

install命令提供了若干“签名”（类似于子命令），签名作为第一个参数传入，可用的签名包括：

| 签名 | 说明 |
| --- | --- |
| install(TARGETS...) | 安装工程中目标对应的二进制文件 |
| install(FILES...) | 一般性的文件安装，包括头文件、文档、软件需要的数据文件 |
| install(PROGRAMS...) | 安装不是由当前工程构建的文件，例如Shell脚本，与FILES签名类似，只是文件被授予可执行权限 |
| install(DIRECTORY...) | 安装一个完整的目录树，例如包含了图标、图片的资源目录 |
| install(SCRIPT...) | 指定一个用户提供的、在安装过程中（典型的是pre-install、post-install）执行的CMake脚本 |
| install(CODE...) | 与SCRIPT类似，只是脚本以内联字符串形式提供 |

前四个签名都用于创建文件的安装规则，需要安装的目标、目录、文件紧接着签名列出。其余和安装相关的信息，以关键字参数的形式附加，大部分签名支持以下关键字：

| 关键字 | 说明 |
| --- | --- |
| DESTINATION | 说明在何处放置被安装的文件，后面必须紧跟一个目录，此目录可以指定为绝对路径。如果使用相对路径，则相对于安装时指定的前缀，前缀可能由缓存条目`CMAKE_INSTALL_PREFIX`指定。前缀的默认值：<br>1 UNIX：/usr/local<br>2 Windows：系统盘符:\Program Files\工程名称 |
| PERMISSIONS | 说明如何设置被安装文件的权限（UNIX文件模式），仅在需要覆盖签名默认权限的情况下使用，可用的权限为：[OWNER\|GROUP\|WORLD] [READ\|WRITE\|EXECUTE]、SETUID、SETGID<br>某些平台不完整支持上述权限，这种情况下自动忽略此关键字 |
| CONFIGURATIONS | 指定规则应用到的构建配置（Release、Debug...）的列表<br>没有应用到的构建配置，不会执行此命令调用产生的规则 |
| COMPONENT | 指定规则应用到的组件。某些工程把安装划分为多个组件，以便分别打包<br>例如某个工程可能包含三个组件：<br>1 Runtime：包含运行软件需要的文件<br>2 Development：包含基于软件进行开发需要的文件<br>3 Documentation：包含软件的手册和帮助文档<br>没有应用到的组件，不会执行此命令调用产生的规则<br>默认情况下，会安装所有组件，因而此关键字不产生任何影响。如果要安装特定组件，必须手工调用安装脚本 |
| OPTIONAL | 指示如果期望的待安装文件不存在时，不是一个错误，仅仅忽略之 |

以此签名调用install命令，以便构建过程中创建的库、可执行文件。详细调用格式为：

```
install ( TARGETS
    targets...  # 基于add_executable/add_library创建的目标的列表
    [
        # 通过TARGETS签名安装的文件可以分为三类：
        # ARCHIVE 静态库（UNIX/Cygwin/MinGW的.A、Windows的.LIB）
        #         DLL的可链接（Linkable）导入库（Cygwin/MinGW的.DLL.A、Windows的.LIB）
        # LIBRARY 可加载模块、共享库（.SO）
        # RUNTIME 可执行文件、动态链接库（.DLL）
        # 如果指定下面一行的某个关键字，则后续的关键字仅针对特定类型的文件，否则针对所有文件
        [ARCHIVE|LIBRARY|RUNTIME|FRAMEWORK|BUNDLE|PRIVATE_HEADER|PUBLIC_HEADER|RESOURCE]
        [DESTINATION <dir>]
        [PERMISSIOS permissions...]
        [CONFIGURATIONS [Debug|Release|...]]
        [COMPONENT component]
        [OPTIONAL]
        [EXPORT <export name>]
        # 下面的关键字仅用于LIBRARY类型，仅针对支持namelink、版本化共享库的平台
        # 对于符号链接lib<name>.so -> lib<name>.so.1，后者是soname，前者称为namelink，namelink用
        # NAMELINK_ONLY导致仅仅共享库的namelink被安装；NAMELINK_SKIP导致除了namelink之外的文件被安
        # 如果不指定，那么namelink、共享库的文件都被安装
        [NAMELINK_ONLY|NAMELINK_SKIP]
    ] [
        ... #仅需要针对不同类型（ARCHIVE|LIBRARY|RUNTIME...）分别设置关键字时，才会出现
    ]
)
```

注意上面代码中关于文件分类的规则，把同属于共享库的.SO、.DLL分别划分到LIBRART、RUNTIME是有意的设计，因为Windows平台下，DLL通常和EXE存放在一个目录，这样才能确保DLL能够被找到并加载。下面的调用确保共享库目标mySharedLib产生的所有文件在所有平台上均安装到期望的位置：

```
install ( TARGETS myExecutable mySharedLib myStaticLib myPlugin
    RUNTIME DESTINATION bin             COMPONENT Runtime
    LIBRARY DESTINATION lib             COMPONENT Runtime
    ARCHIVE DESTINATION lib/myproject   Component Development    #静态库只有在二次开发时才需要
)
```

很多工程可能需要安装与目标无关的任何文件，这时可以使用一般目的的FILES签名：

```
install (FILES files...      #需要被安装的文件的列表，如果是相对路径，相对于当前Source目录
    DESTINATION <dir>        #目标位置，如果是相对路径，相对于安装Prefix
    [PERMISSIOS permissions...] #默认权限644
    [CONFIGURATIONS [Debug|Release|...]]
    [COMPONENT component]
    [RENAME <name>]  #为文件指定新的名称，要求文件列表只有一个元素
    [OPTIONAL]
)
```

某些工程可能安装额外的助手程序——Shell脚本或者Python脚本。这时可以使用PROGRAMS签名。此签名和FILES一样，只是默认权限为755。

有时候我们需要安装包含了大量资源文件的整个目录，此时使用DIRECTORY签名：

```
install (DIRECTORY dirs...    # 需要被安装的目录的列表，如果是相对路径，相对于当前Source目录
    # 目标位置，此目录确保被创建。如果设置为share/myproject，则：
    # data/icons 被安装到/share/myproject/icons，注意输入目录的所有祖先目录被忽略
    # data/ 被安装到/share/myproject，注意结尾的斜杠，会导致此目录下所有内容被安装，因此data/类似于dat
    DESTINATION <dir>
    # 默认权限：文件与FILES一样644，目录与PROGRAMS一样755，下面两个关键字用于修改默认行为
    [FILE_PERMISSIOS permissions...]
    [DIRECTORY_PERMISSIOS permissions...]
    # 和文件来源保持一致的权限
    [USE_SOURCE_PERMISSIOS]
    [CONFIGURATIONS [Debug|Release|...]]
    [COMPONENT component]
    [
        # 排除某些文件，或者为某些文件指定特殊的权限
        # PATTERN用于UNIX风格通配符匹配；REGEX用于正则式匹配
        [PATTERN <pattern> | REGEX <regex>]
        # 是否把匹配的文件排除，不安装
        [EXCLUDE]
        # 设置匹配文件的权限
        [PERMISSIOS permissions...]
    ]
    [
    ...  #排除或者chmod其它匹配文件
    ]
)
```

拷贝文件到安装树下（Installation tree）不是安装过程的唯一内容，有时候需要执行特定的逻辑。这时可以使用SCRIPT或者CODE签名：

```
install(SCRIPT scr.cmake)   # scr.cmake为某个CMake脚本名称
install(CODE "message(Hello)") #直接跟着脚本内容
```

注意脚本不是在CMakeLists.txt处理过程中，而是在安装过程中执行，因而在脚本中不能访问CMakeLists.txt定义的变量。尽管如此， `CMAKE_INSTALL_PREFIX` 、 `CMAKE_INSTALL_CONFIG_NAME` 、 `CMAKE_INSTALL_COMPONENT` 会被设置为真实的安装前缀、构建配置、组件类型。

OS自带的、第三方提供的或者工程本身生成的共享库，是某些可执行文件能够运行的前提条件。由OS提供的自然不需要额外安装；工程本身产生的库由add_library命令说明，一般通过install命令安装到系统。需要额外考虑的是第三方库。

CMake提供两个模块，用于简化共享库的处理。

使用该模块的 `get_prerequisites()` 函数，可以分析一个可执行文件的依赖。将可执行文件的路径传递给此函数，其会输出运行此文件必须的依赖库的列表，包括传递性依赖。该函数使用各平台上的Native工具：dumpbin（Windows）、otool（Mac）、ldd（Linux）进行依赖分析。

使用该模块的 `fixup_bundle()` 函数，可以依据可执行文件的相对位置，拷贝和修复共享库（依赖）。

对于Mac的bundle应用，需要的共享库会被嵌入到bundle中，并调用install_name_tool生成一个自包含bundle。

对于Widnows，需要的共享库会被拷贝到exe所在目录，可执行文件运行时会自动寻找并加载。

要使用fixup_bundle()函数，首先安装某个可执行目标，然后创建一个可以在安装时执行的CMake脚本，在此脚本中调用：

FixBundle.cmake

```
include (BundleUtilities)
# 安装树中的可执行文件的路径
set (bundle "${CMAKE_INSTALL_PREFIX}/myExecutable@CMAKE_EXECUTABLE_SUFFIX@")
# 无法通过依赖分析到达的依赖库的列表
set (other_libs "")
# 可以寻找到前置依赖库的目录的列表
set (dirs "@LIBRARY_OUTPUT_PATH@")

# 调用
fixup_bundle("${bundle}" "${other_libs}" "${dirs}")
```

CMake 2.6开始，支持在两个CMake工程之间导入导出目标。

导入目标这一机制，用于将项目外部的磁盘文件转换为逻辑的CMake目标。在调用add_executable、add_library命令时，传递 `IMPORTED` 选项，即可定义导入目标。CMake不会为导入目标生成构建文件，导入目标仅仅用于便利的引用外部的可执行文件和库。

下面的例子定义了一个导入的可执行文件，仅仅将其作为命令调用：

```
# 声明一个名为generator的导入目标
add_executable(generator IMPORTED)
# 设置目标的实际位置
set_property(TARGET generator PROPERTY IMPORT_LOCATION "/path/to/generator")
# 调用自定义命令，即添加一条定制的构建规则
# 底层构建系统执行类似这样的命令/path/to/generator /project/binary/dir/generated.c
add_custom_command(OUTPUT generated.c COMMAND generator generated.c)
```

下面的例子定义了一个导入的库，并与之链接：

```
add_library(foo IMPORTED)

# Linux
set_property(TARGET foo PROPERTY IMPORTED_LOCATION "/path/to/libfoo.a")
# Windows下需要同时导入.lib和.dll
set_property(TARGET foo PROPERTY IMPORTED_LOCATION "/path/to/libfoo.dll")
set_property(TARGET foo PROPERTY IMPORTED_IMPLIB "/path/to/libfoo.lib")
# 具有多个构建配置的库，可以作为单个目标导入
set_property(TARGET foo PROPERTY IMPORTED_LOCATION_RELEASE "/path/to/libfoo.a")
set_property(TARGET foo PROPERTY IMPORTED_LOCATION_DEBUG   "/path/to/debug/libfoo.a")
add_executable(myexe src1.c)
target_link_libraries(myexe foo)
```

尽管导入机制很有用，但是作为导入者来说，你必须知道目标在磁盘的位置。

使用导出机制，可以在提供目标文件的同时，提供一个文件，帮助其它工程导入。联合使用 `install(TARGETS)` 和 `install(EXPORTS)` 可以在安装目标的同时，把CMake文件也安装到机器上：

```
add_executable(generator generator.c)
# EXPORT选项导致生成一个助手文件，该文件是一个CMake脚本，可以让其它工程方便的导入generator
install(TARGET generator DESTINATION lib/myporj/generators EXPORT myproj-targets)
# 安装助手文件
install(EXPORT myproj-targets DESTINATION lib/myproj)
```

助手文件的内容可以是：

```
# get_filename_component命令拥有得到一个全路径的某个部分
# 第一个参数：结果变量；第二个参数：待解析的路径；第三个参数，需要得到的部分，可以是DIRECTORY/NAME/EXT/PA
# CMAKE_CURRENT_LIST_FILE当前正在处理文件的路径
get_filename_component(_self "${CMAKE_CURRENT_LIST_FILE}" PATH)
# 解析出安装前缀的绝对路径
get_filename_component(PREFIX "${_self}/../.." ABSOLUTE)
# 添加导入目标
add_executable(generator IMPORTED)
# 通过计算出的路径引用目标
set_property(TARGET generator PROPERTY IMPORTED_LOCATION "${PREFIX}/lib/myproj/generators/g
```

注意上面这个脚本依据自身位置动态计算出目标位置，即使移动安装目录，也不会失效。

其它工程只需要包含助手文件即可：

```
include(/lib/myproj/myproj-targets.cmake)
# generator已经导入
add_custom_command(OUTPUT generated.c COMMAND generator generated.c)
```

注意，单个助手文件可以容纳多个目标，甚至这些目标不在同一个目录中：

```
# A/CMakeLists.txt
add_executable(generator generator.c)
install(TARGETS generator DESTINATION lib/myproj/generators EXPORT myproj-targets)
# B/CMakeLists.txt
add_library(foo STATIC foo1.c)
install(TARGETS foo DESTINATION lib EXPORT myproj-targets)  #导出为同一个EXPORT

# CMakeLists.txt
add_subdirectory(A)
add_subdirectory(B)
install(EXPORT myproj-targets DESTINATION lib/myproj)
```

典型情况下，在第三方工程需要导入之前，当前工程已经构建并安装，因此导出一般是基于安装树的。

CMake直接从构建树导出 ，这样第三方工程可以参考构建树来导入，这样就可以避免安装当前工程了。

使用 `export` 命令可以直接从构建树生成一个助手文件：

```
add_executable(generator generator.c)
export (TARGETS generator FILE myproj-exports.cmake)
```

第三方工程可以include当前工程构建树下的myproj-exprots.cmake文件，其中包含导入generator需要的全部信息。

这种导出方式在交叉编译场景下可以用到。

系统探测，即检测在其上构建的系统的各种环境信息，是构建跨平台库或者应用程序的关键因素。

很多C/C++程序依赖于外部的库，然后在编译和链接一个工程时，如何找到已经存在的头文件和库并不容易。因为开发程序的机器，和构建并安装程序的机器中，库的安装位置可能不一样。CMake提供多种特性，辅助开发者把外部库集成到工程中。

与集成外部库相关的命令包括： `find_library` 、 `find_path` 、 `find_program` 、 `find_package` 。对于大部分C/C++库，使用前两个命令一般足够和系统上已安装的库进行链接，这两个命令分别可以用来定位库文件、头文件所在目录。举例：

```
# 寻找一个库
find_library(
        TIFF_LIBRARY
        NAMES tiff tiff2  #只需要库的basename，不需要平台特定的前缀、后缀。前面的库优先
        #额外的路径，支持Windows注册表条目，例如[HKEY_CURRENT_USER\\Software\\Path;Build1]
        PATHS /usr/local/lib /usr/lib       #前面的路径优先
)
# 寻找一般性的文件，仅支持一个待查找文件，支持多个路径
find_path(
        TIFF_INCLUDES
        tiff.h
        /usr/local/include /usr/include
)
include_directories(${TIFF_INCLUDES})
add_executable(mytiff mytiff.c)
target_link_libraries(mytiff ${TIFF_LIBRARY})
```

注意：

1. find_*命令总是会寻找PATH环境变量
2. find_*命令会自动创建对应的缓存条目（文件没找到的情况下值为 `VAR-NOTFOUND` ），便于用户手工修改。这样即使CMake没有找到文件，用户还可以手工的修复

在跨平台软件中，应当避免使用平台特定的代码，例如：

```
C
```

```
// 基于系统的判断
#ifdef defined(SUN) && defined(HPUX)
    foobar();
#endif
```

这会降低代码的可移植性，每当需要支持新的系统，都要改变代码。即便非要使用宏，也最好使用基于特性，而不是基于系统的判断。可以改造上述代码为：

C

```
#ifdef HAS_FOOBAR_FUNC
    forbar();
#endif
```

通过 `try_compile` 、 `try_run` 命令，CMake可以用来自动生成类似上面的HAS_***宏定义。这些命令编译/执行一小段代码，以探测系统特性：

testFoobarModule.cmake

```
try_compile(
        HAS_FOOBAR_FUNC
        ${CMAKE_BINARY_DIR}
        ${PROJECT_SOURCE_DIR}/testFoobar.c  #尝试调用forbar()函数
)
```

如果编译成功，则CMake变量HAS_FOOBAR_FUNC为真。我们可以通过 `add_definitions` 命令或者配置头文件（更好），来设置HAS_FOOBAR_FUNC为宏定义。

如果单纯的编译并不够，还需要获知探测代码的运行结果，可以使用：

TestByteOrder.c C

```
int main() {
    union {
        int i;
        char c;
    } u;
    u.i = 65;
    exit( u.c == 'A' );
}
```

```
try_run(
        RUN_RESULT_VAR  #尝试运行的返回结果
        COMPILE_RESULT_VAR  #编译结果
        ${CMAKE_BINARY_DIR}
        ${PROJECT_SOURCE_DIR}/Modules/TestByteOrder.c
        OUTPUTVAR OUTPUT  #运行的任何输出
)
```

运用上述命令的运行结果，可以根据字节序的不同来定制构建过程或者设置宏定义。对于较小的测试程序，可以不特定编写文件，使用 `file` 命令即可：

```
file(WRITE ${CMAKE_BINARY_DIR}/tmp/testc "int main(){return 0;}")
```

在CMake/Modules中预定义了若干CMake用，可简化日常工作。这些宏常常需要查看当前 `CMAKE_REQUIRED_FLAGS` 、 `CMAKE_REQUIRED_LIBRARIES` 变量的值，以便添加额外的编译标记，或者链接以测试：

| 模块 | 说明 |
| --- | --- |
| CheckFunctionExists.cmake | 检查一个特定的C函数是否存在于系统中。接受两个参数，第一个参数是待测试的函数名，第二个参数是存放测试结果的变量<br>该宏会查看上述两个变量 |
| CheckIncludeFile.cmake | 检查一个头文件是否存在于系统中。第一个参数是头文件名称，第二个参数是存放测试结果的变量，第三个参数是可选的编译标记，如果不指定，使用CMAKE_REQUIRED_FLAGS |
| CheckIncludeFileCXX.cmake | 与上面类似，但是用于C++程序。第一个参数是头文件名称，第二个参数是存放测试结果的变量，第三个参数是可选的编译标记 |
| CheckLibraryExists.cmake | 检查一个库是否存在于系统。接受4个参数：待测试库名称、库中待测试函数名称、库的寻找位置、测试结果<br>该宏会查看上述两个变量 |
| CheckSymbolExists.cmake | 检查某个符号是否在头文件中定义。接受3个参数：待测试符号名称、尝试包含的头文件列表、测试结果<br>该宏会查看上述两个变量 |
| CheckTypeSize.cmake | 确定某个类型的长度（字节数）。接受2个参数：待测试类型、测试结果<br>该宏会查看上述两个变量 |
| CheckVariableExists.cmake | 检查某个全局变量是否存在。接受2个参数：待测试全局变量名称、测试结果。仅用于C变量<br>该宏会查看上述两个变量 |

CMake提供 `find_package(Package [version])` 命令来查找符合CPack包规则的软件包。

该命令可以在两个模式下运行：

1. Module模式：此模式下CMake会依次扫描 `CMAKE_MODULE_PATH` 、CMake安装目录。尝试寻找到一个名称为 `Find<Package>.cmake` 的查找模块。如果找到则加载之，并调用其来寻找目标包的全部组件。查找模块针对特定包编写，它了解此包的全部版本，能找到包的库或者其它文件。CMake提供了很多常用的查找模块
2. Config模式：如果Module模式下没有定位到查找模块，命令自动切换到Config模式（你也可以显式的调用该模式）。在该模式下，命令会寻找包配置文件（package configuration file）：目标包提供的、一个名为 `<Package>Config[Version].cmake` 或者 `<package>-config[-version].cmake` 的文件。只要给出包的名称，命令就知道从何处寻找包配置文件，可能的位置是 `<prefix>/lib/<package>/<package>-config.cmake`

CMake的内置查找模块，在找到包后，一般会定义一系列的变量供当前工程使用：

| 变量名称约定 | 说明 |
| --- | --- |
| <PKG>_INCLUDE_DIRS | 包的头文件所在目录 |
| <PKG>_LIBRARIES | 包提供的库的完整路径 |
| <PKG>_DEFINITIONS | 使用包时，编译代码需要用的宏定义 |
| <PKG>_EXECUTABLE | 包提供的PKG工具所在目录 |
| <PKG>_<TOOL>_EXECUTABLE | 包提供的TOOL工具所在目录 |
| <PKG>_ROOT_DIR | PKG包的安装根目录 |
| <PKG>_VERSION_<VER> | 如果PKG的VER版本被找到，则定义为真 |
| <PKG>_<CMP>_FOUND | 如果PKG的CMP组件被找到，则定义为真 |
| <PKG>_FOUND | 如果PKG被找到则定义为真 |

要传递参数给编译器，可以指定命令行，或者使用一个预先配置好的头文件。

调用 `add_definitions` 命令，可以向编译器传递宏定义：

```
#定义一个布尔的缓存条目
option(DENIG_BUILD "Enable debug messages")
if (DEBUG_BUILD)
    #添加宏定义
    add_definitions(-DDEBUG_MSG)
endif ()
```

如果要细粒度的控制宏定义，可以设置目录、目标、源文件的 `COMPILE_DEFINITIONS` 属性：

```
add_library(mylib src1.c src2.c)
# 可以添加APPEND选项，追加值而不是覆盖
set_property(DIRECTORY PROPERTY COMPILE_DEFINITIONS A AV=1)
set_property(TARGET mylib PROPERTY COMPILE_DEFINITIONS B BV=2)
set_property(SOURCE src1.c PROPERTY COMPILE_DEFINITIONS C CV=3)
# 执行上述命令后，编译参数分别为：
# src1.c -DA -DAV=1 -DB -DBV=2 -DC -DCV=3
# src2.c -DA -DAV=1 -DB -DBV=2
# main.c -DA -DAV=1
```

这种方式更可维护，大部分工程应当使用该方式。应用程序只需要引入预先配置好的头文件即可，不必编写复杂的CMake规则。

我们可以把头文件看作一种配置文件，而要生成配置文件，可使用 `configure_file(input output [@ONLY])` 命令，此命令需要一个“输入文件”，输入文件可以包含三种变量定义方式：

```C
// 第一种方式
#cmakedefine VARIABLE
// 如果VARIABLE为真，则输出：
#define VARIABLE
// 否则输出：
/* #undef VARIABLE */

//第二种方式，直接输出变量的值。如果confgure_file命令传递@ONLY选项，则这种方式不能使用
${VARIABLE}

//第三种方式，直接输出变量的值
@VARIABLE@
```

配置文件应当输出到二进制树，而不是源码树，避免代码污染。因为单个CMake的源码树可以供多种构建树或平台使用，它们生成的配置文件常常是不一样的。你可能需要用include_directories命令将配置文件所在目录作为头文件目录。

除了头文件以外， `configure_file` 命令亦可用来生成包的配置文件。在“查找包”一节我们已经讨论过，包配置文件供其它工程发现本包。

很多时候，“构建”一个工程不仅仅是简单的编译、链接、拷贝，额外的工作——例如利用文档工具生成文档——需要在构建过程中完成。

通过定制命令和目标，CMake可以被扩展以支持任意的任务（或者说规则）。

定制命令时，面临的一个重要问题是可移植性：

1. 各平台上用于完成一项任务的工具不同，以复制文件为例，UNIX使用cp命令，Windows则使用copy命令
2. 目标在各平台上的名字不同，例如库x在UNIX上可能叫libx.so，Windows上则叫x.dll

CMake提供了两个主要工具，解决上面两个可移植性问题。

使用 `cmake -E arguments` 调用，可以执行一些跨平台的操作，在CMakeLists.txt中可以通过定制命令来调用cmake命令，cmake这个可执行文件可以用变量 `CMAKE_COMMAND` 引用。

支持的操作（arguments）包括：

| 操作 | 说明 |
| --- | --- |
| chdir dir command args | 改变当前目录为dir然后执行指定的命令 |
| copy file destination | 拷贝文件 |
| copy_if_different infile outfile | 如果两个文件不一样，则从infile拷贝到outfile |
| copy_directory source destination | 拷贝source目录（包括子目录）中全部文件到destination目录 |
| remove file1 file2... | 从磁盘上删除文件 |
| echo string | 打印到标准输出 |
| time command args | 运行一个命令并且计算耗时 |

CMake不限制你仅使用cmake命令，事实上你可以使用任何命令，但是要注意可移植性问题。一个通用的实践是，通过 `find_program` 找到一个程序，然后在定制命令中调用之。

CMake提供了一系列预定义的变量，描述系统的特征：

| 变量 | 说明 |
| --- | --- |
| EXE_EXTENSION | 可执行文件的扩展名，Windows平台是.exe，UNIX是空 |
| CMAKE_CFG_INTDIR | 诸如VS、XCode这样的开发环境，根据构建配置的不同，使用不同的子目录，例如Debug、Release<br>在一个库、可执行文件、目标文件上执行一个命令时，你往往需要知道它们的完整路径<br>改变了在UNIX上通常是 `./` 而VS则是 `$(INTDIR)/` |
| CMAKE_CURRENT_BINARY_DIR | 与当前CMakeList文件关联的输出目录的完整路径<br>可能与PROJECT_BINARY_DIR（当前工程二进制树的顶级目录）不同 |
| CMAKE_CURRENT_SOURCE_DIR | 与当前CMakeList文件关联的源码目录的完整路径<br>可能与PROJECT_SOURCE_DIR（当前工程源码树的顶级目录）不同 |
| EXECUTABLE_OUTPUT_PATH | 某些工程指定可执行文件需要生成到的目录，该变量指示其完整路径 |
| LIBRARY_OUTPUT_PATH | 某些工程指定库文件需要生成到的目录，该变量指示其完整路径 |

| 变量 | 说明 |
| --- | --- |
| CMAKE_SHARED_MODULE_PREFIX<br>CMAKE_SHARED_MODULE_SUFFIX | 共享模块文件的前后缀 |
| CMAKE_SHARED_LIBRARY_PREFIX<br>CMAKE_SHARED_LIBRARY_SUFFIX | 共享库文件的前后缀 |
| CMAKE_LIBRARY_PREFIX<br>CMAKE_LIBRARY_SUFFIX | 静态库文件的前后缀 |

add_custom_command有两个主要的签名：TARGET、OUTPUT，分别用于为目标或者文件添加额外的规则。其中TARGET签名语法如下：

```
add_custom_command(
    #目标的名称
    TARGET target
    #执行触发时机:
    #pre_build，在目标任何依赖文件被构建之前执行
    #pre_link，在所有依赖已经构建好，但是尚未链接时执行
    #post_build，在目标已经构建好后执行
    PRE_BUILD | PRE_LINK | POST_BUILD
    # command为可执行文件的名称
    COMMAND command [ARGS arg1 arg2 ... ]
    [COMMAND command [ARGS arg1 arg2 ... ] ...]
    # 注释，在定制命令运行时打印
    [COMMENT comment]
)
```

下面是一个例子，在目标构建好后立即拷贝之：

```
add_executable(myExe my.c)
get_target_property(EXE_LOC myExe LOCATION)
add_custom_command(
    TARGET myExe
    POST_BUILD
    COMMAND ${CMAKE_COMMAND} ARGS -E copy ${EXE_LOC} /QC/files
)
```

add_custom_command的另外一个用途是指定生成一个文件的规则。这种情况下，已有的用于生成目标文件的规则被替换掉。语法如下：

```
```

```
add_custom_command (
    # 指定命令运行生成的结果文件，最好指定完整路径
    OUTPUT output1 [output2 ...]
    # 需要执行的命令
    COMMAND command [ARGS [args ...]]
    [ COMMAND command [ARGS [args ...]] ...]
    # 主要用于VS
    [MAIN_DEPENDENCY depend]
    # 命令依赖于的文件，最好指定完整路径（依赖是目标则不必），这些文件中的任何一个变化后，命令都需要重新执行
    [DEPENDS [depends ...]]
    [COMMENT comment]
)
```

在某些库的构建过程中，例如TIFF，会先编译并构建一个可执行文件，再用此可执行文件生成还有系统特定信息的源码，而此源码参与库的最终构建。这种场景下，可以使用add_custom_command来生成源码。

CMake支持除了库、可执行文件之外的，更一般概念上的目标，称为定制目标。生成文档、运行测试、更新Web服务器都可以抽象为目标。

要添加定制目标，需要调用下面的命令：

```
add_custom_target(
    # name为目标的名称，如果使用Makefile生成器，你可以调用make name来生成此目标
    # ALL，表示该目标包含在ALL_BUILD目标中，自动构建
    name [ALL]
    # 执行的命令
    [command arg arg ...]
    # 此目标依赖的文件的列表，最好指定完整路径（依赖是目标则不必），这些文件可以是add_custom_command(OUTP
    [DEPENDS dep dep ...]
)
```

所谓交叉编译，就是指软件在一个平台（Build host）上构建，而在另外一个平台（Target Platform）上运行。目标平台往往是另外一个OS甚至没有OS，也常常使用与构建平台不一样的硬件，这导致目标平台根本不能运行开发环境。交叉编译的典型应用是嵌入式开发，程序需要在路由器、传感器之类的特殊硬件上运行。

交叉编译依赖于工具链。工具链是针对目标平台的一整套工具，包括编译器、链接器，以及目标平台的全套头文件、库。

从2.6开始，CMake完整的支持交叉编译，包括Linux-Windows交叉编译，或者PC-嵌入设备交叉编译。在交叉编译场景下，会面临以下问题：

1. CMake无法自动的检测目标平台
2. CMake无法在默认系统目录寻找库、头文件
3. 构建出来的可执行文件无法运行

CMake区分构建平台、运行平台的信息，让给用户可以解决交叉编译相关的问题，避免例如运行虚拟机的额外需求。

通过一个所谓工具链（Toolchain）文件，我们可以告知CMake关于目标平台的任何必要信息。CMakeList.txt必须被调整以适应目标平台和构建平台具有不同属性的情况。下面是一个Linux下基于MinGW交叉编译器，交叉编译Windows程序使用的工具链文件：

TC-mingw.cmake

```
#目标平台的名称
set( CMAKE_SYSTEM_NAME Windows )

#指定C/C++编译器为交叉编译器，只有交叉编译器才知道如何构建目标平台上的二进制文件
set( CMAKE_C_COMPILER i586-mingw32msvc-gcc )
set( CMAKE_CXX_COMPILER i586-mingw32msvc-g++ )

#指定目标平台环境的位置，这一位置在构建平台中，但是存放的是目标平台需要的头文件、库
set( CMAKE_FIND_ROOT_PATH /usr/i586-mingw32msvc /home/alex/mingw-install )

#调整find_***命令的行为
#仅在构建平台上寻找程序
set( CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER )
#仅在目标平台环境中寻找头文件、库
set( CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY )
set( CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY )
```

通过下面的命令，可以指示CMake使用上述工具链文件：

Shell

```
cd src/build
cmake -DCMAKE_TOOLCHAIN_FILE=~/TC-mingw.cmake ...
```

CMAKE_TOOLCHAIN_FILE必须在最初运行时设置，后续其值会保存为缓存条目。每个目标平台一般只需要一个工具链文件。

在工具链文件中，可能需要设置以下变量：

| 变量 | 说明 |
| --- | --- |
| CMAKE_SYSTEM_NAME | 该变量必须设置，指示目标平台的名称。典型的名称如Linux、Windows，如果目标平台是无OS的嵌入式平台，设置为Generic。此名称会用来生成平台文件的名称，例如Linux.cmake、Windows-gcc.cmake<br>手工设置此变量后，CMake认为当前是在做交叉编译，自动设置变量 `CMAKE_CROSSCOMPILING` 为真 |
| CMAKE_SYSTEM_VERSION | 可选的，目标平台的版本 |

| 变量 | 说明 |
| --- | --- |
| CMAKE_SYSTEM_PROCESSOR | 可选的，目标平台处理器或者硬件的名称。CMake用此变量加载文件：`${CMAKE_SYSTEM_NAME}-COMPILER_ID-${CMAKE_SYSTEM_PROCESSOR}.cmake`，该文件用来修改设置，例如目标的编译标记<br>仅在你需要为目标平台指定特殊的编译设置时，才需要设置该变量 |
| CMAKE_C_COMPILER | 指定C编译器的完整路径或者名字。如果指定为完整路径，这CMake倾向于到对应目录寻找binutils、linker、C++编译器等其它内容。如果指定的编译器是一个GNU交叉编译器，则CMake会自动寻找到对应的C++编译器，例如从arm-elf-gcc找到arm-elf-c++<br>C编译器亦可通过环境变量CC设置 |
| CMAKE_CXX_COMPILER | 指定C++编译器的完整路径或者名字。对于GNU工具链，只需要设置CMAKE_C_COMPILER，此变量不必设置<br>C++编译器亦可通过环境变量CXX设置 |
| CMAKE_FIND_ROOT_PATH | 指定一组包含了目标平台环境的目录，这些目录供所有find_**命令使用<br>假设目标平台环境安装在/opt/eldk/ppc_74xx，设置变量为此路径。find_library寻找jpeg库时会最终定位到/opt/eldk/ppc_74xx/lib/libjpeg.so |
| CMAKE_FIND_ROOT_PATH_MODE_PROGRAM<br>CMAKE_FIND_ROOT_PATH_MODE_LIBRARY<br>CMAKE_FIND_ROOT_PATH_MODE_INCLUDE | 分别设置find_program、find_library、find_include命令的默认行为。可以设置为：<br>1 NEVER CMAKE_FIND_ROOT_PATH对命令无效<br>2 ONLY 仅在CMAKE_FIND_ROOT_PATH目录中搜索<br>3 BOTH 默认值，都搜索 |

注：形如CMAKE_SYSTEM_XXX的变量，总是在描述目标平台。如果要描述当前构建平台，可以使用相应的CMAKE_HOST_SYSTEM_XXX变量。

CMake提供了一些变量，用于粗粒度的测试系统特征：

1. 目标平台类型指示变量：UNIX、WIN32、APPLE
2. 构建平台类型指示变量：CMAKE_HOST_UNIX、 CMAKE_HOST_WIN32、CMAKE_HOST_APPLE

更加细化的测试变量，可以使用上节提到的CMAKE_SYSTEM_XXX、CMAKE_HOST_SYSTEM_XXX变量。

CMake中使用CHECK_INCLUDE_FILES、CHECK_C_SOURCE_RUNS等宏来测试平台属性，这些宏通常使用try_compile、try_run命令。

try_run无法正常运行，因为交叉编译出的可执行文件不能在构建平台上运行。try_run被调用时，它首先尝试编译，如果成功它会检查CMAKE_CROSSCOMPILING变量，该变量为真的话它不会尝试运行，而是设置两个缓存变量，供用户后续修改。考虑下面这个例子：

```
try_run(
        SHARED_LIBRARY_PATH_TYPE
        SHARED_LIBRARY_PATH_INFO_COMPILED
        ${PROJECT_BINARY_DIR}/CMakeTmp
        ${PROJECT_SOURCE_DIR}/CMake/SharedLPathInfo.cxx
        OUTPUT_VARIABLE OUTPUT
        ARGS "LDPATH"
)
```

如果SharedLPathInfo.cxx编译成功，SHARED_LIBRARY_PATH_INFO_COMPILED被设置为真。而交叉编译时无法运行可执行文件，因此CMake创建一个缓存条目： `SHARED_LIBRARY_PATH_TYPE=PLEASE_FILL_OUT-FAILED_TO_RUN` ，该条目必须被手工的设置为SharedLPathInfo的在目标平台上的退出码。如果指定了OUTPUT_VARIABLE选项，CMake还会创建一个缓存条目 `SHARED_LIBRARY_PATH_TYPE__TRYRUN_OUTPUT=PLEASE_FILL_OUT-NOTFOUND` ，该条目必须手工设置为SharedLPathInfo在目标平台上的标准输出/错误。

要手工运行，可以考虑构建目录下的 `cmTryCompileExec-SHARED_LIBRARY_PATH_TYPE` 到目标平台下执行。

除了设置缓存条目外，还可以把运行结果记录到 `${CMAKE_BINARY_DIR}/TryRunResults.cmake` 中，该文件由CMake自动创建，其中包含所有CMake无法确定的变量，并记录可执行文件位置、源码位置、运行参数等信息。我们可以根据运行结果填充这些变量的值，然后为cmake调用缓存：

```Shell
cmake -C ~/TryRunResults-myproj-eldk-ppc.cmake
```

本节示例一个完整的交叉编译的过程。

交叉编译的第一步是寻找合适的工具链，如果你已经安装好工具链，则可以跳过这一步。

不同工程——包括基于Linux的PDA和嵌入式设备厂商的——在Linux上处理交叉编译的途径不同，它们有自己的构建流程和工具链。CMake可以使用这些工具链，只要它们是基于普通文件系统的。

一个提供较为完整的目标平台环境的工具链套件是Embedded Linux Development Toolkit（ELDK），该套件支持ARM、PowerPC、MIPS等目标平台。ELDK或者其它工具链可以被安装在构建平台的任意位置，例如：

Shell

```
# 工具链
/home/alex/eldk-mips/usr/bin
# 目标平台环境
/home/alex/eldk-mpis/mips_4KC/
```

下载并安装好工具链后，下一步就是编写工具链文件，注意我们在上面提到过，每个工具链只需要一个这样的文件：

~/CPP/cmake/toolchains/ToolChain-eldk-mips4K.cmake

```
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_C_COMPILER /home/alex/eldk-mips/usr/bin/mips_4KC-gcc)
set(CMAKE_CXX_COMPILER /home/alex/eldk-mips/usr/bin/mips_4KC-g++)
set(CMAKE_FIND_ROOT_PATH /home/alex/eldk-mips/mips-4KC /home/alex/eldk-mips-extra-install)

set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE  ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY  ONLY)
```

工具链文件可以存放在任何地方，推荐将其存放在统一的目录，方便其它工程重用。

Hello.c C

```
int main(){
    exit(0)
}
```

CMakeLists.txt

```
project(Hello)
add_executable(Hello Hello.c)
```

Shell

```
mkdir eldk-mips
cd eldk-mips
# 调用cmake，指定工具链文件
cmake -DCMAKE_TOOLCHAIN_FILE=~/CPP/cmake/toolchains/ToolChain-eldk-mips4K.cmake ..
# 生成器已经生成Makefile，可以构建
make VERBOSE=1
```

不但支持具有OS的目标平台的交叉编译，CMake还可以针对那些没有OS的微控制器进行交叉编译。

本节的例子使用SDCC编译器，该编译器可以运行于主流系统下，支持8/16位微控制器的交叉编译。

首先，还是编写工具链文件：

```
set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_SYSTEM_PROCESSOR i8501)
set(CMAKE_C_COMPILER "D:/CPP/sdcc/bin/sdcc.exe")
```

对于没有OS的目标平台，其系统名称设置为Generic。CMake假设Generic平台不支持共享库。

很多微控制器工程不需要依赖任何外部库，因此往往不需要设置影响find_**的变量。

CMakeLists文件：

Toolchain-sdcc.cmake

```
# 明确声明使用C语言，因为SDCC不支持C++
project (Blink C)
add_library(blink blink.c)
add_executable(hello main.c)JSONCPP_INCLUDE_DIRS
target_link_libraries(hello blink)
```

执行构建：

MS DOS

```
rem 使用MS NMake生成器驱动构建
cmake -G "NMake Makefiles" -DCMAKE_TOOLCHAIN_FILE=D:/CPP/cmake/toolchains/Toolchain-sdcc.cma
rem 执行构建
namke
```

此命令是cmake的命令行接口

Shell

```
cmake [options] <path-to-source>
cmake [options] <path-to-existing-build>
```

| 选项 | 说明 |
| --- | --- |
| -C <initial-cache> | 预加载一个脚本，以生成缓存 |
| -D <var>:<type>=<value> | 定义一个cmake缓存条目 |
| -U <globbing_expr> | 从缓存中移除匹配的条目 |
| -G <generator-name> | 指定一个生成器（构建系统） |
| -T <toolset-name> | 指定生成器支持的工具集 |

| 选项 | 说明 |
| --- | --- |
| -Wno-dev | 抑制开发者警告 |
| -E | cmake命令模式 |
| -i | 以向导模式运行 |
| -L[A][H] | 列出所有非进阶缓存变量 |
| --build <dir> | 在dir中构建二进制树 |
| --find-package | 以类似于pkg-config的方式来查找包 |
| --graphviz=[file] | 生成依赖的graphviz |
| --system-information [file] | 输出系统信息 |
| --debug-trycompile | 不删除trycompile构建树 |
| --debug-output | 以调试模式运行 |
| --trace | 以trace模式运行（更多调试信息） |
| --warn-uninitialized | 对未初始化值进行警告 |
| --warn-unused-vars | 对未使用变量进行警告 |

```Shell
cmake
    -DCMAKE_BUILD_TYPE:STRING=Debug          # 构建配置
    -DCMAKE_INSTALL_PREFIX:STRING=/usr       # 安装位置
```

其它变量参考这里。

可以使用CMake 3.1引入的变量：

```
set(CMAKE_CXX_STANDARD 11)
```

如果要对老版本CMake兼容，参考下面的宏：

```
macro(use_cxx11)
  if (CMAKE_VERSION VERSION_LESS "3.1")
    if (CMAKE_CXX_COMPILER_ID STREQUAL "GNU")
      set (CMAKE_CXX_FLAGS "-std=gnu++11 ${CMAKE_CXX_FLAGS}")
    endif ()
  else ()
    set (CMAKE_CXX_STANDARD 11)
  endif ()
endmacro(use_cxx11)
```

全局性宏定义：

```
# 定义宏MACRO_NAME，注意前面的-D
add_definitions(-DMACRO_NAME)
# 赋值
add_definitions(-DMACRO_NAME=${value})
```

针对某个目标：

```
# 仅仅能用于 add_executable() 或 add_library() 添加的目标

# 格式:
target_compile_definitions(<target>
  # PUBLIC、INTERFACE可以将宏定义传递给target的PUBLIC、INTERFACE条目
  <INTERFACE|PUBLIC|PRIVATE> [items1...]
  [<INTERFACE|PUBLIC|PRIVATE> [items2...] ...])

# 示例:
target_compile_definitions(hello PRIVATE A=1 B=0)

# 你也可以直接设置目标属性  COMPILE_DEFINITIONS
```

使用环境变量：

```Shell
export CC=/home/alex/CPP/lib/gcc/7.2.0/bin/gcc
export CXX=/home/alex/CPP/lib/gcc/7.2.0/bin/g++
```

在基于CLion开发时，将上述内容添加到clion.sh脚本中

分享这篇文章到：

← Previous Post (https://blog.gmem.cc/essay-20150427)

MongoDB学习笔记 (https://blog.gmem.cc/mongodb-study-note) →

您可以使用以下网站的账户登录：

*姓名（必填）*

*Email（必填）*

*网站*

*说点什么吧*

发布