Cmake快速入门

GNU开发工具——CMake快速入门

一、CMake简介

不同Make工具，如GNU Make、QT的qmake、微软的MS nmake、BSD Make（pmake）等，遵循着不同的规范和标准，所执行的Makefile格式也不同。如果软件想跨平台，必须要保证能够在不同平台编译。而如果使用Make工具，必须为不同的Make工具编写不同的Makefile。

CMake是一个比Make工具更高级的编译配置工具，是一个跨平台的、开源的构建系统（BuildSystem）。CMake允许开发者编写一种平台无关的**CMakeList.txt**文件来定制整个编译流程，然后再根据目标用户的平台进一步生成所需的本地化Makefile和工程文件，如：为Unix平台生成Makefile文件（使用GCC编译），为Windows MSVC生成projects/workspaces（使用VS IDE编译）或Makefile文件（使用nmake编译）。使用CMake作为项目架构系统的知名开源项目有VTK、ITK、KDE、OpenCV、OSG等。

二、CMake管理工程

在Linux平台下使用CMake生成Makefile并编译的流程如下：
A、编写CMake配置文件CMakeLists.txt
B、执行命令cmake PATH生成Makefile，PATH是CMakeLists.txt所在的目录。
C、使用make命令进行编译。

三、单个源文件工程

1、源文件编写

假设项目test中只有一个main.cpp源文件，程序用途是计算一个数的指数幂。

```
#include <stdio.h>
#include <stdlib.h>
/**
 * power - Calculate the power of number.
 * @param base: Base value.
 * @param exponent: Exponent value.
 *
 * @return base raised to the power exponent.
 */
double power(double base, int exponent)
{
    int result = base;
    int i;

    if (exponent == 0)
    {
        return 1;
    }

    for(i = 1; i < exponent; ++i)
    {
        result = result * base;
```

```
    }
    return result;
}
int main(int argc, char *argv[])
{
    if(argc < 3)
    {
        printf("Usage: %s base exponent \n", argv[0]);
        return 1;
    }
    double base = atof(argv[1]);
    int exponent = atoi(argv[2]);
    double result = power(base, exponent);
    printf("%g ^ %d is %g\n", base, exponent, result);
    return 0;
}
```

2、编写CMakeLists.txt

在main.cpp源文件目录test下编写CMakeLists.txt文件。

```
#CMake最低版本号要求
cmake_minimum_required (VERSION 2.8)
#项目信息
project (demo)
#指定生成目标
add_executable(demo main.cpp)
```

CMakeLists.txt由命令、注释和空格组成，其中命令是不区分大小写。符号#后的内容被认为是注释。命令由命令名称、小括号和参数组成，参数之间使用空格进行间隔。

本例中CMakeLists.txt文件的命令如下：

`cmake_minimum_required`：指定运行本配置文件所需的CMake的最低版本；

`project`：参数值是demo，表示项目的名称是demo。

`add_executable`：将名为main.cpp的源文件编译成一个名称为demo的可执行文件。

3、编译工程

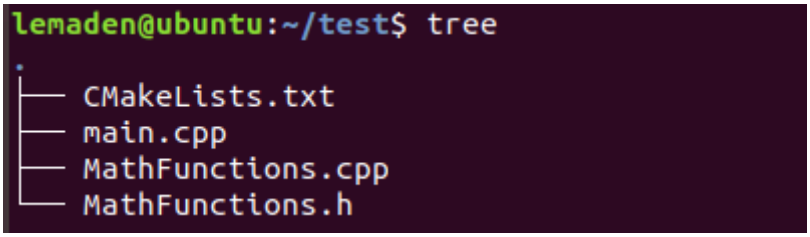
在源码根目录下创建一个build目录，进入build目录，执行`cmake ..`，生成Makefile，再使用`make`命令编译得到demo可执行文件。

通常，建议在源码根目录下创建一个独立的build构建编译目录，将构建过程产生的临时文件等文件与源码隔离，避免源码被污染。

四、单目录多源文件工程

1、源文件编写

假如把`power`函数单独写进一个名为`MathFunctions.cpp`的源文件里，使得这个工程变成如下的形式：



MathFunctions.h文件：

```
/**
 * power - Calculate the power of number.
 * @param base: Base value.
 * @param exponent: Exponent value.
 *
 * @return base raised to the power exponent.
 */

double power(double base, int exponent);
```

MathFunctions.cpp文件：

```
double power(double base, int exponent)
{
    int result = base;
    int i;

    if (exponent == 0)
    {
        return 1;
    }

    for(i = 1; i < exponent; ++i)
    {
        result = result * base;
    }
    return result;
}
```

main.cpp文件：

```
#include <stdio.h>
#include <stdlib.h>
#include "MathFunctions.h"

int main(int argc, char *argv[])
{
    if(argc < 3)
    {
        printf("Usage: %s base exponent \n", argv[0]);
        return 1;
    }

    double base = atof(argv[1]);
    int exponent = atoi(argv[2]);
    double result = power(base, exponent);
    printf("%g ^ %d is %g\n", base, exponent, result);
```

```
    return 0;
}
```

2、编写CMakeLists.txt

```
#CMake最低版本号要求
cmake_minimum_required(VERSION 2.8)
#项目信息
project(demo)
#指定生成目标
add_executable(demomain.cpp MathFunctions.cpp)
```

`add_executable` 命令中增加了一个 `MathFunctions.cpp` 源文件，但如果源文件很多，可以使用 `aux_source_directory` 命令， `aux_source_directory` 命令会查找指定目录下的所有源文件，然后将结果存进指定变量名。其语法如下：

`aux_source_directory(dir variable)`

修改后CMakeLists.txt如下：

```
#CMake最低版本号要求
cmake_minimum_required(VERSION 2.8)
# 项目信息
project(demo)
#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})
```

CMake会将当前目录所有源文件的文件名赋值给变量DIR_SRCS ，再指示变量DIR_SRCS中的源文件需要编译成一个名称为demo的可执行文件。

五、多文件多目录工程

1、源码文件编写

创建一个math目录，将MathFunctions.h和MathFunctions.cpp文件移动到math目录下。在工程目录根目录test和子目录math里各编写一个CMakeLists.txt文件，可以先将math目录里的文件编译成静态库再由main函数调用。

math子目录：
`MathFunctions.h` 文件：

```
/**
 * power - Calculate the power of number.
 * @param base: Base value.
 * @param exponent: Exponent value.
 *
 * @return base raised to the power exponent.
 */

double power(double base, int exponent);
```

`MathFunctions.cpp` 文件：

```
double power(double base, int exponent)
{
    int result = base;
    int i;

    if (exponent == 0)
    {
        return 1;
    }

    for(i = 1; i < exponent; ++i)
    {
        result = result * base;
    }
    return result;
}
```

根目录源文件：

```
#include <stdio.h>
#include <stdlib.h>
#include "math/MathFunctions.h"

int main(int argc, char *argv[])
{
    if(argc < 3)
    {
        printf("Usage: %s base exponent \n", argv[0]);
        return 1;
    }

    double base = atof(argv[1]);
    int exponent = atoi(argv[2]);
    double result = power(base, exponent);
    printf("%g ^ %d is %g\n", base, exponent, result);

    return 0;
}
```

2、CMakeLists.txt文件编写

根目录的CMakeLists.txt文件：

```
# CMake最低版本号要求
cmake_minimum_required(VERSION 2.8)
# 项目信息
project(demo)
#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${DIR_SRCS})
# 添加链接库
target_link_libraries(demo MathFunctions)
```

`add_subdirectory`命令指明本工程包含一个子目录math，math目录下的 CMakeLists.txt文件和源代码也会被处理 。`target_link_libraries`命令指明可执行文件demo需要连接一个名为MathFunctions的链接库 。

math子目录的CMakeLists.txt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})
```

`add_library`命令将math目录中的源文件编译为静态链接库。

六、自定义编译选项

1、自定义编译选项简介

CMake允许为工程增加编译选项，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MathFunctions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。

2、CMakeLists 文件编写

在根目录的CMakeLists.txt文件指定自定义编译选项：

```
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)
# 项目信息
project (demo)
set(CMAKE_INCLUDE_CURRENT_DIR ON)
# 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})
```

`configure_file`命令用于加入一个配置头文件config.h，config.h文件由CMake从config.h.in生成，通过预定义一些参数和变量来控制代码的生成。

`option`命令添加了一个USE_MYMATH选项，并且默认值为ON。

根据USE_MYMATH变量的值来决定是否使用自己编写的MathFunctions库。

3、修改源文件的调用

修改main.cpp文件，让其根据USE_MYMATH的预定义值来决定是否调用标准库还是MathFunctions库。

```
#include <cstdio>
#include <cstdlib>
#include <config.h>

#ifdef USE_MYMATH
#include <MathFunctions.h>
#else
#include <cmath>
#endif


int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        printf("Usage: %s base exponent \n", argv[0]);
        return 1;
    }

    double base = atof(argv[1]);
    int exponent = atoi(argv[2]);

#ifdef USE_MYMATH
    printf("Now we use our own Math library. \n");
    double result = power(base, exponent);
#else
    printf("Now we use the standard library. \n");
    double result = pow(base, exponent);
#endif

    printf("%g ^ %d is %g\n", base, exponent, result);
    return 0;
}
```

4、编写config.h.in文件

main.cpp文件包含了一个config.h文件，config.h文件预定义了USE_MYMATH 的值。但不会直接编写config.h文件，为了方便从CMakeLists.txt中导入配置，通常编写一个config.h.in文件，内容如下：

```
#cmakedefine USE_MYMATH
```

CMake会自动根据CMakeLists.txt配置文件中的设置自动生成config.h文件。

5、编译工程

修改CMakeLists.txt文件，USE_MYMATH为OFF，使用标准库。

```
# 是否使用自己的MathFunctions库
option (USE_MYMATH
           "Use provided math implementation" OFF)
```

在build目录下`cmake ..,make`，执行程序。

七、安装和测试

1、定制安装规则

在math/CMakeLists.txt文件指定MathFunctions库的安装规则：

```
#指定MathFunctions库的安装路径
install(TARGETS MathFunctions DESTINATION bin)
install(FILES MathFunctions.h DESTINATION include)
```

修改根目录的CMakeLists.txt文件指定目标文件的安装规则：

```
#指定安装路径
install(TARGETS test DESTINATION bin)
install(FILES "${PROJECT_BINARY_DIR}/config.h"
         DESTINATION include)
```

通过对安装规则的定制，生成的目标文件和MathFunctions函数库 libMathFunctions.o文件将会被拷贝到/usr/local/bin中，而MathFunctions.h和生成的config.h文件则会被复制到/usr/local/include中。

/usr/local是默认安装到的根目录，可以通过修改 CMAKE_INSTALL_PREFIX 变量的值来指定文件应该拷贝到哪个根目录。

2、为工程添加测试

CMake提供了一个CTest测试工具。在项目根目录的CMakeLists.txt文件中调用一系列的add_test 命令。

```
    #启用测试
    enable_testing()
    #测试程序是否成功运行
    add_test(test_run demo 5 2)
    #测试帮助信息是否可以正常提示
    add_test(test_usage demo)
    set_tests_properties(test_usage
      PROPERTIES PASS_REGULAR_EXPRESSION "Usage: .* base exponent")
```

```
#测试5的平方
add_test(test_5_2 demo 5 2)
set_tests_properties(test_5_2
 PROPERTIES PASS_REGULAR_EXPRESSION "is 25")
#测试10的5次方
add_test(test_10_5 demo 10 5)
set_tests_properties(test_10_5
 PROPERTIES PASS_REGULAR_EXPRESSION "is 100000")
#测试2的10次方
add_test(test_2_10 demo 2 10)
set_tests_properties(test_2_10
 PROPERTIES PASS_REGULAR_EXPRESSION "is 1024")
```

第一个测试test_run用来测试程序是否成功运行并返回0值。剩下的三个测试分别用来测试 5 的 平方、10 的 5 次方、2 的 10 次方是否都能得到正确的结果。其中PASS_REGULAR_EXPRESSION用来测试输出是否包含后面跟着的字符串。

如果要测试更多的输入数据，可以通过编写宏来实现：

```
# 启用测试
enable_testing()

# 测试程序是否成功运行
add_test (test_run demo 5 2)

# 测试帮助信息是否可以正常提示
add_test (test_usage demo)
set_tests_properties (test_usage
  PROPERTIES PASS_REGULAR_EXPRESSION "Usage: .* base exponent")

# 测试 5 的平方
# add_test (test_5_2 Demo 5 2)

# set_tests_properties (test_5_2
#  PROPERTIES PASS_REGULAR_EXPRESSION "is 25")

# 测试 10 的 5 次方
# add_test (test_10_5 Demo 10 5)

# set_tests_properties (test_10_5
#  PROPERTIES PASS_REGULAR_EXPRESSION "is 100000")

# 测试 2 的 10 次方
# add_test (test_2_10 Demo 2 10)

# set_tests_properties (test_2_10
#  PROPERTIES PASS_REGULAR_EXPRESSION "is 1024")

# 定义一个宏，用来简化测试工作
macro (do_test arg1 arg2 result)
  add_test (test_${arg1}_${arg2} demo ${arg1} ${arg2})
  set_tests_properties (test_${arg1}_${arg2}
    PROPERTIES PASS_REGULAR_EXPRESSION ${result})
endmacro (do_test)

# 利用 do_test 宏，测试一系列数据
do_test (5 2 "is 25")
```

```
    do_test (10 5 "is 100000")
    do_test (2 10 "is 1024")
```

八、GDB支持

让CMake支持gdb的设置只需要指定Debug模式下开启-g选项：

```
set(CMAKE_BUILD_TYPE "Debug")
set(CMAKE_CXX_FLAGS_DEBUG "$ENV{CXXFLAGS} -O0 -Wall -g -ggdb")
set(CMAKE_CXX_FLAGS_RELEASE "$ENV{CXXFLAGS} -O3 -Wall")
```

生成的程序可以直接使用gdb来调试。

九、添加环境检查

使用平台相关的特性时，需要对系统环境做检查。检查系统是否自带pow函数，如果有pow函数，就使用；否则使用自定义的power函数。

1、添加 CheckFunctionExists 宏

首先在顶层CMakeLists.txt文件中添加CheckFunctionExists.cmake 宏，并调用check_function_exists命令测试链接器是否能够在链接阶段找到 pow函数。

```
#检查系统是否支持 pow 函数
include (${CMAKE_ROOT}/Modules/CheckFunctionExists.cmake)
check_function_exists (pow HAVE_POW)
#check_function_exists需要放在configure_file命令前。
```

2、预定义相关宏变量

修改 config.h.in 文件，预定义相关的宏变量。

```
// does the platform provide pow function?
#cmakedefine HAVE_POW
```

3、在代码中使用宏和函数

修改 main.cpp文件 ，在代码中使用宏和函数。

```
#include <stdio.h>
#include <stdlib.h>
#include <config.h>

#ifdef HAVE_POW
#include <math.h>
#else
#include <MathFunctions.h>
#endif

int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        printf("Usage: %s base exponent \n", argv[0]);
```

```
        return 1;
    }
    double base = atof(argv[1]);
    int exponent = atoi(argv[2]);

#ifdef HAVE_POW
    printf("Now we use the standard library. \n");
    double result = pow(base, exponent);
#else
    printf("Now we use our own Math library. \n");
    double result = power(base, exponent);
#endif

    printf("%g ^ %d is %g\n", base, exponent, result);
    return 0;
}
```

十、添加版本号

修改顶层CMakeLists.txt文件，在project命令后分别指定当前的项目的主版本号和副版本号。

```
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)
# 项目信息
project (demo)

set (Demo_VERSION_MAJOR 1)
set (Demo_VERSION_MINOR 0)

set(CMAKE_INCLUDE_CURRENT_DIR ON)

#检查系统是否支持 pow 函数
include (${CMAKE_ROOT}/Modules/CheckFunctionExists.cmake)
check_function_exists (pow HAVE_POW)

# 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"
  )

# 是否加入 MathFunctions 库
if (NOT HAVE_POW)
  include_directories ("${PROJECT_SOURCE_DIR}/math")
  add_subdirectory (math)
  set (EXTRA_LIBS ${EXTRA_LIBS} MathFunctions)
endif (NOT HAVE_POW)

# 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})

#指定安装路径
install(TARGETS demo DESTINATION bin)
```

```
install(FILES "${PROJECT_BINARY_DIR}/config.h"
        DESTINATION include)

    # 启用测试
    enable_testing()

    # 测试程序是否成功运行
    add_test (test_run demo 5 2)

    # 测试帮助信息是否可以正常提示
    add_test (test_usage demo)
    set_tests_properties (test_usage
      PROPERTIES PASS_REGULAR_EXPRESSION "Usage: .* base exponent")

    # 测试 5 的平方
    # add_test (test_5_2 Demo 5 2)

    # set_tests_properties (test_5_2
    #  PROPERTIES PASS_REGULAR_EXPRESSION "is 25")

    # 测试 10 的 5 次方
    # add_test (test_10_5 Demo 10 5)

    # set_tests_properties (test_10_5
    #  PROPERTIES PASS_REGULAR_EXPRESSION "is 100000")

    # 测试 2 的 10 次方
    # add_test (test_2_10 Demo 2 10)

    # set_tests_properties (test_2_10
    #  PROPERTIES PASS_REGULAR_EXPRESSION "is 1024")

    # 定义一个宏，用来简化测试工作
    macro (do_test arg1 arg2 result)
      add_test (test_${arg1}_${arg2} demo ${arg1} ${arg2})
      set_tests_properties (test_${arg1}_${arg2}
        PROPERTIES PASS_REGULAR_EXPRESSION ${result})
    endmacro (do_test)

    # 利用 do_test 宏，测试一系列数据
    do_test (5 2 "is 25")
    do_test (10 5 "is 100000")
    do_test (2 10 "is 1024")
```

分别指定当前的项目的主版本号和副版本号。

为了在代码中获取版本信息，可以修改 config.h.in 文件，添加两个预定义变量：

```
// the configured options and settings for Tutorial
#define Demo_VERSION_MAJOR @Demo_VERSION_MAJOR@
#define Demo_VERSION_MINOR @Demo_VERSION_MINOR@

// does the platform provide pow function?
#cmakedefine HAVE_POW
```

直接在源码中使用：

```c
#include <stdio.h>
#include <stdlib.h>
#include <config.h>

#ifdef HAVE_POW
#include <math.h>
#else
#include <MathFunctions.h>
#endif

int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        // print version info
        printf("%s Version %d.%d\n",
               argv[0],
               Demo_VERSION_MAJOR,
               Demo_VERSION_MINOR);
        printf("Usage: %s base exponent \n", argv[0]);
        return 1;
    }

    double base = atof(argv[1]);
    int exponent = atoi(argv[2]);

#ifdef HAVE_POW
    printf("Now we use the standard library. \n");
    double result = pow(base, exponent);
#else
    printf("Now we use our own Math library. \n");
    double result = power(base, exponent);
#endif

    printf("%g ^ %d is %g\n", base, exponent, result);
    return 0;
}
```

十一、生成安装包

1、增加CPack模块

CMake提供了一个专门用于打包的工具CPack，用于配置生成各种平台上的安装包，包括二进制安装包和源码安装包。

首先在顶层的CMakeLists.txt文件尾部添加下面几行：

```cmake
# 构建一个 CPack 安装包
include (InstallRequiredSystemLibraries)
set (CPACK_RESOURCE_FILE_LICENSE
  "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
set (CPACK_PACKAGE_VERSION_MAJOR "${Demo_VERSION_MAJOR}")
set (CPACK_PACKAGE_VERSION_MINOR "${Demo_VERSION_MINOR}")
include (CPack)
```

导入InstallRequiredSystemLibraries模块，便于导入CPack模块；

设置一些CPack相关变量，包括版权信息和版本信息导入CPack模块。

在顶层目录下创建License.txt文件内如如下：

```
The MIT License (MIT)

Copyright (c) 2018 Scorpio Studio

Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of
the Software, and to permit persons to whom the Software is furnished to do so,
subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR
COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER
IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN
CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

2、生成安装包

生成二进制安装包：

`cpack -C CPackConfig.cmake`

生成源码安装包:

`cpack -C CPackSourceConfig.cmake`

上述两个命令都会在目录下创建3个不同格式的二进制包文件：

demo-1.0.1-Linux.tar.gz
demo-1.0.1-Linux.tar.Z
demo-1.0.1-Linux.sh

3个二进制包文件所包含的内容是完全相同的。

GNU开发工具——CMake进阶

一、CMake基础指令

1、cmake_minimum_required

`cmake_minimum_required (VERSION 2.8)`

cmake_minimum_required用于规定cmake程序的最低版本，可选。如果CMakeLists.txt文件中使用了高版本cmake特有的一些命令时，就需要使用cmake_minimum_required对CMake进行版本限制，提醒用户升级到相应版本后再执行cmake。

2、project

```
project(project_name)
```

`project`用于指定项目的名称。项目最终编译生成的可执行文件并不一定是项目名称，由另一条命令确定。在cmake中有两个预定义变量：`projectname_BINARY_DIR`以及`projectname_SOURCE_DIR`，同时cmake还预定义了`PROJECT_BINARY_DIR`和`PROJECT_SOURCE_DIR`变量。内部编译情况下，`projectname_BINARY_DIR`与`PROJECT_BINARY_DIR`相同；外部编译情况下，`PROJECT_BINARY_DIR`指向build构建目录。工程实践中，推荐使用`PROJECT_BINARY_DIR`和`PROJECT_SOURCE_DIR`变量，即使项目名称发生变化也不会影响CMakeLists.txt文件。

3、外部构建

通过执行cmake生成MakeFile文件，执行make进行编译安装，其方法如下：

方法一（内部构建）：

在工程CMakeLists.txt所在目录下执行：

```
cmake .
make
```

方法二（外部构建）：

在工程CMakeLists.txt所在目录下执行：

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

```
cmake -B build
cmake --build build
cmake --build build --config release
```

两种方法最大的不同在于执行cmake和make的工作路径不同。第一种方法中，cmake生成的所有中间文件和可执行文件都会存放在项目目录中；而第二种方法中，中间文件和可执行文件都将存放在build目录中，避免了源代码被污染，第二中方法的build目录可以创建在任意目录下，只需要在执行cmake时指定工程的CMakeLists.txt目录即可。由于第二种方法的生成、编译和安装是发生在不同于项目目录的其它目录中，因此第二种方法称为外部构建。

CMake官方推荐使用外部构建方法，可以避免源码被污染。

4、add_subdirectory

```
add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])
```

`add_subdirectory`指令用于向当前工程添加存放源文件的子目录，并可以指定中间二进制和目标二进制存放的位置。`EXCLUDE_FROM_ALL`参数的含义是将相应目录从编译过程中排除。如工程的example目录可能需要工程构建完成后再进入example目录单独进行构建。

5、aux_source_directory

```
aux_source_directory(dir variable)
```

`aux_source_directory`用于搜集在指定目录下所有的源文件的文件名（不包括头文件），将输出结果列表储存在指定的变量中。`aux_source_directory`不会对目录下的子目录进行递归调用。

`aux_source_directory`可以很方便的为一个库或可执行目标获取源文件的清单，但`aux_source_directory`的缺点在于，如果CMakeLists.txt中使用了`aux_source_directory`，那么CMake将不能生成一个可以感知新的源文件何时被加进来的构建系统（即新文件的加入并不会导致CMakeLists.txt过时，从而不能引起CMake重新运行，开发者可以手动重新运行CMake来产生一个新的

构建系统）。通常，CMake生成的构建系统能够感知何时需要重新运行CMake，因为需要修改CMakeLists.txt来引入一个新的源文件。当源文件仅仅是加到了目录下，但没有修改CMakeLists.txt文件，使用者只能手动重新运行CMake来产生一个包含新文件的构建系统。

6、编译选项设置

设置编译选项可以通过`add_compile_options`命令，也可以通过`set`命令修改CMAKE_CXX_FLAGS或CMAKE_C_FLAGS。`add_compile_options`命令添加的编译选项是针对所有编译器的(包括c和c编译器)，而`set`命令设置CMAKE_C_FLAGS或CMAKE_CXX_FLAGS变量则是分别只针对c和c++编译器的。

#判断编译器类型,如果是gcc编译器,则在编译选项中加入c++11支持：

```
if(CMAKE_COMPILER_IS_GNUCXX)
    add_compile_options(-std=c++11)
    message(STATUS "optional:-std=c++11")
endif(CMAKE_COMPILER_IS_GNUCXX)
```

设置C标准：

- `set(CMAKE_CXX_STANDARD 11)`

设置C编译器：

- `set(CMAKE_C_COMPILER "gcc")`

设置C编译器：

- `set(CMAKE_CXX_COMPILER "g++")`

设置C编译器编译选项：

- `set（CMAKE_C_FLAGS“$ {CMAKE_C_FLAGS} -fPIC”）`

设置C++编译器编译选项：

- `set(CMAKE_CXX_FLAGS "-std=c++11 ${CMAKE_CXX_FLAGS}")`

设置可执行文件输出目录：

- `SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)`

设置库文件输出目录：

- `SET(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)`

设置Build类型：

- `set(CMAKE_BUILD_TYPE MATCHES "Debug")`

Build类型为Debug, Release, RelWitchDebInfo，RelWitchDebInfo，MinSizeRel。

```
IF (CMAKE_BUILD_TYPE MATCHES "Debug"
        OR CMAKE_BUILD_TYPE MATCHES "None")
    MESSAGE(STATUS "CMAKE_BUILD_TYPE is Debug")
ELSEIF (CMAKE_BUILD_TYPE MATCHES "Release")
    MESSAGE(STATUS "CMAKE_BUILD_TYPE is Release")
ELSEIF (CMAKE_BUILD_TYPE MATCHES "RelWitchDebInfo")
    MESSAGE(STATUS "CMAKE_BUILD_TYPE is RelWitchDebInfo")
ELSEIF (CMAKE_BUILD_TYPE MATCHES "MinSizeRel")
    MESSAGE(STATUS "CMAKE_BUILD_TYPE is MinSizeRel")
ELSE ()
    MESSAGE(STATUS "unknown CMAKE_BUILD_TYPE = " ${CMAKE_BUILD_TYPE})
ENDIF ()
```

设置构建库的类型：

- `set(BUILD_SHARED_LIBS shared)`

库类型为shared和static

增加编译选项：

```
add_definitions("-Wall -lpthread -g")
if( CMAKE_BUILD_TYPE STREQUAL "Release")
  SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -O3 -std=c++11 -fPIC")
else()
  SET(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -g -O0 -std=c++11 -fPIC")#设置寻找外部库的cmake参数的
endif()
message("*** ${PROJECT_NAME}: Build type:" ${CMAKE_BUILD_TYPE}
${CMAKE_CXX_FLAGS} "***")
```

7、find_package

```
find_package(<package> [version] [EXACT] [QUIET]
             [[REQUIRED|COMPONENTS] [components...]]
               [NO_POLICY_SCOPE])
```

QUIET选项将会禁掉包没有被发现时的警告信息。REQUIRED选项表示如果报没有找到的话，cmake的过程会终止，并输出警告信息。

`find_package`可以根据cmake内置的.cmake脚本去查找相应库的模块，调用`find_package`成功后，会有相应的变量生成。如调用`find_package(Qt5Widgets)`后，会生成变量Qt5Widgets_FOUND，Qt5Widgets_INCLUDE_DIRS，然后在CMakeLists.txt里可以使用使用上述变量。

`find_package`会调用预定义在CMAKE_MODULE_PATH下的 Findname.cmake模块。可以自己定义Findname模块，将其放入工程的某个目录中，通过 SET(CMAKE_MODULE_PATH dir)设置查找路径，供工程FIND_PACKAGE使用。

8、find_library

```
find_library (
          <VAR>
          name | NAMES name1 [name2 ...] [NAMES_PER_DIR]
          [HINTS path1 [path2 ... ENV var]]
          [PATHS path1 [path2 ... ENV var]]
```

```
            [PATH_SUFFIXES suffix1 [suffix2 ...]]
            [DOC "cache documentation string"]
            [NO_DEFAULT_PATH]
            [NO_CMAKE_ENVIRONMENT_PATH]
            [NO_CMAKE_PATH]
            [NO_SYSTEM_ENVIRONMENT_PATH]
            [NO_CMAKE_SYSTEM_PATH]
            [CMAKE_FIND_ROOT_PATH_BOTH |
             ONLY_CMAKE_FIND_ROOT_PATH |
             NO_CMAKE_FIND_ROOT_PATH]
           )
```

`FIND_LIBRARY(RUNTIME_LIB rt /usr/local/lib NO_DEFAULT_PATH)`

cmake会在目录中查找，如果所有目录中都没有，RUNTIME_LIB就会被赋值为NO_DEFAULT_PATH。

9、include_directories

`include_directories([AFTER|BEFORE] [SYSTEM] dir1 [dir2 ...])`

`include_directories` 用于指定编译过程中编译器搜索头文件的路径。 默认情况下，将指定目录追加到目录列表最后。通过使用BEFORE或AFTER可以指定目录追加在目录列表的前面或后面。如果设定SYSTEM选项，编译器认定为系统包含目录。

当项目需要的头文件不在系统默认的搜索路径时，需要指定相应头文件路径。

10、link_directories

`link_directories(directory1 directory2 ...)`

`link_directories` 用于指定要链接的库文件的路径，可选。 `find_package` 和 `find_library` 指令可以得到库文件的绝对路径。如果自己写的动态库文件放在自己新建的目录下时，可以用 `link_directories` 指令指定该目录的路径以便工程能够找到。

11、link_libraires

`link_libraries(library1 debug | optimized library2 ...)`

库名称可以是绝对路径，库名，支持多个。

12、add_executable

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

`ADD_EXECUTABLE（ExcutableName SRC）`

`ADD_EXECUTABLE` 定义了工程生成的可执行文件名称，相关的源文件是 SRC中定义的源文件列表。

可以通过 `SET` 指令重新定义 `EXECUTABLE_OUTPUT_PATH` 和 `LIBRARY_OUTPUT_PATH` 变量来指定最终的目标二进制的位置(指最终生成的可执行文件或者最终的共享库，而不包含编译生成的中间文件)。

```
SET(EXECUTABLE_OUTPUT_PATH ${PROJECT_BINARY_DIR}/bin)
SET(LIBRARY_OUTPUT_PATH ${PROJECT_BINARY_DIR}/lib)
```

13、add_library

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

`add_library`用于将指定的源文件生成库文件，然后添加到工程中。

name表示库文件的名字，库文件会根据命令里列出的源文件来创建。而STATIC、SHARED和MODULE的作用是指定生成的库文件的类型。STATIC库是目标文件的归档文件，在链接其它目标的时候使用。SHARED库会被动态链接（动态链接库），在运行时会被加载。MODULE库是一种不会被链接到其它目标中的插件，但可能会在运行时使用dlopen-系列的函数。

14、target_link_libraries

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

`target_link_libraries`用于指定在链接目标文件的时候需要链接的外部库，可以解决外部库的依赖问题。
target是指通过`add_executable()`和`add_library()`指令生成已经创建的目标文件。而[item]表示库文件没有后缀的名字。默认情况下，库依赖项是传递的。当target目标链接到另一个目标时，链接到target目标的库也会出现在另一个目标的链接线上。

15、设置安装目录

`SET(CMAKE_INSTALL_PREFIX /usr/local)`

显式将CMAKE_INSTALL_PREFIX的值定义为/usr/local，在外部构建情况下执行`make install`命令时，`make`会将生成的可执行文件拷贝到/usr/local/bin目录下。

可执行文件的安装路径CMAKE_INSTALL_PREFIX也可以在执行`cmake`命令的时候指定，cmake参数如下：
cmake -DCMAKE_INSTALL_PREFIX=/usr ...

如果cmake参数和CMakeLists.txt文件中都不指定CMAKE_INSTALL_PREFIX值的话，则默认为/usr/local。

16、安装选项

执行INSTALL命令时需要注意CMAKE_INSTALL_PREFIX参数的值。INSTALL命令形式如下：

```
INSTALL(TARGETS targets...
    [[ARCHIVE|LIBRARY|RUNTIME]
    [DESTINATION < dir >]
    [PERMISSIONS permissions...]
    [CONFIGURATIONS
    [Debug|Release|...]]
    [COMPONENT < component >]
    [OPTIONAL]
    ] [...])
```

参数TARGETS后跟目标是通过ADD_EXECUTABLE或者ADD_LIBRARY定义的目标文件，可能是可执行二进制、动态库、静态库。

DESTINATION定义安装的路径，如果路径以/开头，那么是绝对路径，此时CMAKE_INSTALL_PREFIX将无效。如果希望使用CMAKE_INSTALL_PREFIX来定义安装路径，需要写成相对路径，即不要以/开头，安装路径就是`${CMAKE_INSTALL_PREFIX} /destination`定义的路径

TARGETS指定的目标文件不需要指定路径，只需要写上TARGETS名称就。

非目标文件的可执行程序安装(如脚本):

```
INSTALL(PROGRAMS files... DESTINATION < dir >
    [PERMISSIONS permissions...]
    [CONFIGURATIONS [Debug|Release|...]]
    [COMPONENT < component >]
    [RENAME < name >] [OPTIONAL])
```

安装后权限为755。

安装一个目录的命令如下:

```
INSTALL(DIRECTORY dirs... DESTINATION < dir >
    [FILE_PERMISSIONS permissions...]
    [DIRECTORY_PERMISSIONS permissions...]
    [USE_SOURCE_PERMISSIONS]
    [CONFIGURATIONS [Debug|Release|...]]
    [COMPONENT < component >]
    [[PATTERN < pattern > | REGEX < regex >]
    [EXCLUDE] [PERMISSIONS permissions...]] [...])
```

DIRECTORY后连接的是所在Source目录的相对路径，但务必注意:abc 和 abc/有很大的区别。如果目录名不以/结尾，那么目录将被安装为目标路径下的 abc，如果目录名以/结尾，代表将目录中的内容安装到目标路径，但不包括目录本身。

```
INSTALL(DIRECTORY icons scripts/ DESTINATION share/myproj
    PATTERN "CVS" EXCLUDE
    PATTERN "scripts/*"
    PERMISSIONS OWNER_EXECUTE OWNER_WRITE OWNER_READ
    GROUP_EXECUTE GROUP_READ)
```

将icons目录安装到prefix/share/myproj，将scripts/中的内容安装到prefix/share/myproj，`不包含目录名为CVS的目录，对于`scripts/*`文件指定权限为755。`INSTALL(DIRECTORY doc/ DESTINATION share/doc/crnode)

17、message

`message([STATUS|WARNING|AUTHOR_WARNING|FATAL_ERROR|SEND_ERROR] "message to display" ...)`

(无)　= 重要消息

STATUS = 非重要消息

WARNING = CMake警告，会继续执行

AUTHOR_WARNING = CMake警告(dev)，会继续执行

SEND_ERROR = CMake错误，继续执行，但会跳过生成步骤 FATAL_ERROR = CMake错误，终止所有处理过程

18、编译选项

在CMAKE中，设置编译选项可以通过set命令修改CMAKE_CXX_FLAGS或CMAKE_C_FLAGS，也可以使用

`add_compile_options`命令增加编译选项。

`ADD_COMPILE_OPTIONS`命令添加的编译选项是针对所有编译器的(包括C和C编译器)，`set`命令设置CMAKE_C_FLAGS或CMAKE_CXX_FLAGS变量增加的编译选项只针对C和C编译器。

19、宏定义

CMake中添加使用 ADD_DEFINITIONS(-DMACRONAME)增加宏定义。

```
ADD_DEFINITIONS(-DUSE_MATH -DANDROID)
```

20、多线程支持

```
find_package (Threads)
add_executable (XXXX)
target_link_libraries (XXXX ${CMAKE_THREAD_LIBS_INIT})
```

二、CMake高级指令

1、操作符

操作符是大小写敏感的。

一元操作符有：EXISTS，COMMAND，DEFINED

二元操作符有：EQUAL，LESS，LESS_EQUAL，GREATER，GREATER_EQUAL，STREQUAL，STRLESS，STRLESS_EQUAL,，STRGREATER，STRGREATER_EQUAL，VERSION_EQUAL，VERSION_LESS，VERSION_LESS_EQUAL，VERSION_GREATER，VERSION_GREATER_EQUAL，MATCHES

逻辑操作符有：NOT，AND，OR

2、布尔常量

布尔常量值是大小写敏感的。

true：1，ON，YES，TRUE，Y，非0的值。

false：0，OFF，NO，FALSE，N，IGNORE，NOTFOUND，空字符串""，以-NOTFOUND结尾的字符串。

3、if...else

```
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(DEFINED variable)#如果变量被定义，为真。
```

```
if(表达式)
  # 要执行的命令块
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
elseif(表达式2)
  # 要执行的命令块
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
else(表达式)
  # 要执行的命令块
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
endif(表达式)
```

elseif和else是可选的，可以有多条elseif ，缩进和空格对语句解析没有影响。

- IF (COMMAND cmd)

如果cmd确实是命令并可调用，为真

- IF (EXISTS dir)
- IF (EXISTS file)

如果目录或文件存在，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 (var MATCHES regex)

var可以用var名，也可以用${var}

- IF (string MATCHES regex)

当给定的变量或者字符串能够匹配正则表达式regex时为真

- IF (variable LESS number)

variable小于number时为真

- IF (string LESS number)

string小于number时为真

- IF (variable GREATER number)

variable大于number时为真

- IF (string GREATER number)

string大于number时为真

- IF (variable EQUAL number)

variable等于number时为真

- IF (string EQUAL number)

string等于number时为真

- IF (variable STRLESS string)

variable小于字符串string

- IF (string STRLESS string)

字符串string小于字符串string

- IF (variable STRGREATER string)

variable大于字符串string

- IF (string STRGREATER string)

字符串string大于字符串string

- IF (variable STREQUAL string)

Variable等于字符串string

- IF (string STREQUAL string)

字符串string等于字符串string

一个小例子，用来判断平台差异：

```
IF(WIN32)
MESSAGE(STATUS “This is windows.”)
#作一些 Windows 相关的操作

ELSE(WIN32)
MESSAGE(STATUS “This is not windows”)
#作一些非 Windows 相关的操作

ENDIF(WIN32)
```

上述代码用来控制在不同的平台进行不同的控制，但是，阅读起来却并不是那么舒服，ELSE(WIN32)之类的语句很容易引起歧义。

这就用到了我们在“常用变量”一节提到的 CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS开关。

```
SET(CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS ON)
```

这时候就可以写成：

```
IF(WIN32)

ELSE()

ENDIF()
```

如果配合 ELSEIF 使用，可能的写法是这样：

```
IF(WIN32)
#do something related to WIN32
ELSEIF(UNIX)
#do something related to UNIX
ELSEIF(APPLE)
#do something related to APPLE
ENDIF(WIN32)
```

不同操作系统平台的判断代码如下：

```
IF (WIN32)
    #do something related to WIN32
ELSEIF (UNIX)
    #do something related to UNIX
ELSEIF(APPLE)
    #do something related to APPLE
ENDIF (WIN32)
```

4、while

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

5、foreach

foreach列表语法：

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

实例如下：

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

foreach范围语法：

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

实例如下：

```
#从0到total以 1 为步进
FOREACH(VAR RANGE 10)
   MESSAGE(${VAR})
ENDFOREACH(VAR)
```

0 1 2 ... 10

foreach范围和步进语法：

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

从`start`开始到`stop`结束，以`step`为步进，直到遇到`ENDFOREACH`指令，整个语句块才会得到真正的执行。

```
FOREACH(A RANGE 5 15 3)
    MESSAGE(${A})
ENDFOREACH(A)
```

6、共享变量

通常，使用`set`命令定义的变量能从父目录传递到子目录，但不能在同级目录间传递，因此用set定义的变量无法共享，需要用`set(variable value CACHE INTERNAL docstring )`定义变量，把变量加入到CMakeCache.txt，然后各级目录共享会访问到变量。variable为变量名称，value为变量的值，docstring为变量描述，不能为空。

`set_property`提供了实现共享变量的方法，但`set_property`不会将变量写入CMakeCache.txt，而是写入内存中。

当用`set_property`定义的property时，第一个指定作用域(scope)的参数设为GLOBAL，这个property在cmake运行期间作用域就是全局的。然后其他目录下的CMakeLists.txt可以用`get_property`来读取这个property
在opencl目录的CMakeLists.txt中定义一个名为INCLUDE_OPENCL_1_2 的global property：
`set_property(GLOBAL PROPERTY INCLUDE_OPENCL_1_2 "${CMAKE_CURRENT_LIST_DIR}/include/1.2" )`

在其它模块的CMakeLists.txt中读取property：

`get_property(INCLUDE_OPENCL GLOBAL PROPERTY "INCLUDE_OPENCL_1_2" )`

7、set_target_properties

`set_target_properties(target1 target2 ...PROPERTIES prop1 value1 prop2 value2 ...)`

`set_target_properties`指令可以用来设置输出的名称。对于动态库，还可以用来指定动态库版本和API版本。为了实现动态库版本号，使用`set_target_properties`指令方法如下：

`set_target_properties (hello PROPERTIES VERSION 1.2 SOVERSION 1)`

VERSION指代动态库版本，SOVERSION指代API版本。

Hello库包含Hello.cpp、Hello.h两个文件。

Hello.h文件如下：

```
#include <iostream>

class Hello
{
public:
    void Print();
};
```

Hello.cpp文件如下：

```
#include "Hello.h"

void Hello::Print()
{
    std::cout << "Hello world." << std::endl;
}
```

CMakeLists.txt文件如下：

```
cmake_minimum_required(VERSION 2.8.9)
SET (LIBHELLO_SRC Hello.cpp)
# 添加动态库，关键词为shared，不需要写全libhello.so，
ADD_LIBRARY (hello SHARED ${LIBHELLO_SRC})
# 添加静态库，关键词为static，不需要写全libhello_static.a
# target不能重名，因此静态库的target不同与动态库的target重名，修改为hello_static
ADD_LIBRARY (hello_static STATIC ${LIBHELLO_SRC})
# 通常，静态库名字跟动态库名字是一致的，只是扩展名不同；
# 即：静态库名为 libhello.a；  动态库名为libhello.so；
# 因此，希望"hello_static"在输出时，以"hello"的名字显示
SET_TARGET_PROPERTIES (hello_static PROPERTIES OUTPUT_NAME "hello")
GET_TARGET_PROPERTY (OUTPUT_VALUE hello_static OUTPUT_NAME)
MESSAGE (STATUS "This is the hello_static OUTPUT_NAME: " ${OUTPUT_VALUE})
# cmake在构建一个新的target时，会尝试清理掉其它使用target名字的库，
# 因此，在构建libhello.a时，就会清理掉libhello.so.
# 为了避免清理问题，比如再次使用SET_TARGET_PROPERTIES定义 CLEAN_DIRECT_OUTPUT属性。
SET_TARGET_PROPERTIES (hello_static PROPERTIES CLEAN_DIRECT_OUTPUT 1)
SET_TARGET_PROPERTIES (hello PROPERTIES CLEAN_DIRECT_OUTPUT 1)

#通常，动态库包含一个版本号，VERSION指代动态库版本，SOVERSION指代API版本。
SET_TARGET_PROPERTIES (hello PROPERTIES VERSION 1.2 SOVERSION 1)
```

```
#构建完成后需要将libhello.a, libhello.so.x以及hello.h安装到系统目录，
# 将hello的共享库安装到<prefix>/lib目录；
# 将hello.h安装<prefix>/include/hello目录。
INSTALL (TARGETS hello hello_static LIBRARY DESTINATION lib
ARCHIVE DESTINATION lib)
INSTALL (FILES hello.h DESTINATION include/hello)
```

创建build目录，进入build目录。

`cmake …/`

`make`

执行安装：

`sudo make install`

相应的库文件和头文件分别被安装到/usr/local/lib和/usr/local/include/hello目录下。

8、get_target_property

```
get_target_property(OUTPUT_VALUE hello_static OUTPUT_NAME)
message(STATUS "This is the hello_static OUTPUT_NAME: "${OUTPUT_VALUE})
```

如果没有定义hello_static变量的OUTPUT_NAME属性，则OUTPUT_VALUE被赋值NOTFOUND。

9、set_property

```
set_property(<GLOBAL                            |
               DIRECTORY [dir]                  |
               TARGET    [target1 [target2 ...]] |
               SOURCE    [src1 [src2 ...]]       |
               TEST      [test1 [test2 ...]]     |
               CACHE     [entry1 [entry2 ...]]>
              [APPEND][APPEND_STRING]
              PROPERTY <name>[value1 [value2 ...]])
```

在某个域中对零个或多个对象设置一个属性。第一个参数决定该属性设置所在的域，必须为下面中的其中之一：

GLOBAL域是唯一的，并且不接特殊的任何名字。

DIRECTORY域默认为当前目录，但可以用全路径或相对路径指定其它目录（指定目录必须已经被CMake处理）。
TARGET域可命名零或多个已经存在的目标。

SOURCE域可命名零或多个源文件，源文件属性只对在相同目录下的目标是可见的(CMakeLists.txt)。

TEST域可命名零或多个已存在的测试。

CACHE域必须命名零或多个已存在条目的cache。

必选项PROPERTY后为要设置的属性的名字。其它参数用于构建以分号隔开的列表形式的属性值。如果指定了APPEND选项，则指定的列表将会追加到任何已存在的属性值当中。如果指定了APPEND_STRING选项，则会将值作为字符串追加到任何已存在的属性值。

10、get_property

```
get_property(<variable>
                <GLOBAL             |
                 DIRECTORY [dir]    |
                 TARGET    <target> |
                 SOURCE    <source> |
                 TEST      <test>   |
                 CACHE     <entry>  |
                 VARIABLE>
                PROPERTY <name>
                [SET | DEFINED |BRIEF_DOCS | FULL_DOCS])
```

必选项PROPERTY后面紧跟着要获取的属性的名字。如果指定了SET选项，则变量会被设置为一个布尔值，表明该属性是否已设置。如果指定了DEFINED选项，则变量也会被设置为一个布尔值，表明该属性是否已定义（如通过`define_property`）。如果定义了BRIEF_DOCS或FULL_DOCS选项，则该变量被设置为一个字符串，包含了对请求的属性的文档。如果该属性没有相关文件，则会返回NOTFOUND。

```
get_target_property(OUTPUT_VALUE all STATUS)
if(${OUTPUT_VALUE} STREQUAL OUTPUT_VALUE-NOTFOUND)
    #Target all 不存在
else()
    #Target all 存在
endif()
```

11、list

定义列表

`set(列表名 值1 值2 … 值N)`
`set(列表名 “值1;值2; …;值N”)`

```
set(list_var 1 2 3 4) # list_var = 1;2;3;4
set(list_foo "5;6;7;8") # list_foo = 5;6;7;8
message(${list_var})#输出：  1234
message(${list_foo})#输出：5678
message("${list_var}")#输出：1;2;3;4
message("${list_foo}")#输出：5;6;7;8
```

不加引号的引用cmake将自动在分号处进行切分成多个列表元素，并将其作为多个独立的参数传给命令。加引号的引用cmake不会进行切分并保持分号不动，把整个引号内的内容当作一个参数传给命令。

常用列表操作如下：

`list(LENGTH list output variable)`

LENGTH返回列表的长度

```
list(GET <list> <elementindex> [<element index> ...]
       <output variable>)
```

GET返回列表中指定下标的元素

```
list(APPEND <list><element> [<element> ...])
```

APPEND添加新元素到列表中

```
list(FIND <list> <value><output variable>)
```

FIND查找list列表中为value的值

```
list(INSERT <list><element_index> <element> [<element> ...])
```

INSERT 将新元素插入到列表中指定的位置

```
list(REMOVE_ITEM <list> <value>[<value> ...])
```

REMOVE_ITEM从列表中删除某个元素

```
list(REMOVE_AT <list><index> [<index> ...])
```

REMOVE_AT从列表中删除指定下标的元素

```
list(REMOVE_DUPLICATES <list>)
```

REMOVE_DUPLICATES从列表中删除重复的元素

```
list(REVERSE <list>)
```

REVERSE将列表的内容实地反转，改变的是列表本身，而不是其副本

```
list(SORT <list>)
```

SORT将列表按字母顺序实地排序，改变的是列表本身，而不是其副本。

列表的操作方法会在当前的CMake变量域创建一些新值，即使列表本身是在父域中定义的，LIST命令也只会在当前域创建新的变量值，为了将操作结果向上传递，需要通过SET PARENT_SCOPE， SET CACHE INTERNAL或其他值域扩展的方法。

当指定索引值时，element index为大于或等于0的值，从列表的开始处索引，0代表列表的第一个元素。如果element index为小于或等于-1的值，从列表的结尾处索引，-1代表列表的最后一个元素。

12、file

`file(WRITE filename "message towrite"... )`

WRITE将一则信息写入文件'filename'中，如果文件存在，会覆盖，如果不存在，会创建文件。

```
file(APPEND filename "message to write"... )
```

APPEND将信息内容追加到文件末尾。

`file(READ filename variable [LIMIT numBytes] [OFFSEToffset] [HEX])`

READ会读取文件的内容并将其存入到变量中。会在给定的偏移量处开始读取最多numBytes个字节。如果指定了HEX参数，二进制数据将会被转换成十进制表示形式并存储到变量中。

`file(<MD5|SHA1|SHA224|SHA256|SHA384|SHA512> filenamevariable)`

MD5, SHA1, SHA224, SHA256, SHA384, SHA512会计算出文件内容对应的加密散列。

```
file(STRINGS filename variable [LIMIT_COUNT num]
     [LIMIT_INPUT numBytes] [LIMIT_OUTPUTnumBytes]
     [LENGTH_MINIMUM numBytes] [LENGTH_MAXIMUMnumBytes]
     [NEWLINE_CONSUME] [REGEX regex]
     [NO_HEX_CONVERSION])
```

STRINGS从文件中解析出ASCII字符串列表并存储在变量中。文件中的二进制数据将被忽略，回车符(CR)也会被忽略。可以解析Intel Hex和Motorola S-record文件，两种文件在读取时会自动转换为二进制格式，可以使用参数NO_HEX_CONVERSION禁用自动转换。LIMIT_COUNT设置可返回的最大数量的字符串。LIMIT_INPUT 设置从输入文件中可读取的最大字节数。LIMIT_OUTPUT设置了存储在输出变量中最大的字节数。 LENGTH_MINIMUM设置了返回的字符串的最小长度。小于这个长度的字符串将被忽略。LENGTH_MAXIMUM 设置返回的字符串的最大长度。大于这个长度的字符串将被切分为长度不大于于最大长度值的子字符串。NEWLINE_CONSUME 允许换行符包含进字符串中而不是截断它们。REGEX 指定了返回的字符串必须匹配的正则表达式的模式。典型用法

`file(STRINGS file.txt myfile)`

将输入文件的每行内容存储在变量"myfile"中。

`file(GLOB variable [RELATIVE path] [globbingexpressions]...)`

GLOB 会产生一个由所有匹配globbing表达式的文件组成的列表，并将其保存到变量中。Globbing表达式与正则表达式类似，但更简单。如果指定了RELATIVE 标记，返回的结果将是与指定的路径相对的路径构成的列表。 (通常不推荐使用GLOB命令来从源码树中收集源文件列表。

globbing 表达式包括：

```
*.cxx      - match all files with extension cxx
*.vt?      - match all files with extension vta,...,vtz
f[3-5].txt - match files f3.txt,f4.txt, f5.txt
```

```
file(GLOB_RECURSE variable [RELATIVE path]
     [FOLLOW_SYMLINKS] [globbingexpressions]...)
```

GLOB_RECURSE会遍历匹配目录的所有文件以及子目录下面的文件。对于属于符号链接的子目录，只有FOLLOW_SYMLINKS指定1或者cmake策略CMP0009没有设置为NEW时，才会遍历链接目录。

```
file(RENAME <oldname> <newname>)
```

RENAME 将文件系统中的文件或目录移动到目标位置，并自动替换目标位置处的文件或目录。

- `file(REMOVE [file1 ...])`

REMOVE 会删除指定的文件以及子目录下的文件。

- `file(REMOVE_RECURSE [file1 ...])`

REMOVE_RECURSE 会删除指定的文件及子目录，包括非空目录。

- `file(MAKE_DIRECTORY [directory1 directory2 ...])`

MAKE_DIRECTORY在指定目录处创建子目录，如果父目录不存在，会创建父目录。

- `file(RELATIVE_PATH variable directory file)`

RELATIVE_PAT推断出指定文件相对于特定目录的路径。

- `file(TO_CMAKE_PATH path result)`

TO_CMAKE_PATH会将路径转换成cmake风格的路径表达形式。

- `file(TO_NATIVE_PATH path result)`

TO_NATIVE_PATH与TO_CMAKE_PATH类似，但执行反向操作，将cmake风格的路径转换为操作系统特定风格的路径表式形式。

```
file(DOWNLOAD url file [INACTIVITY_TIMEOUT timeout]
     [TIMEOUT timeout] [STATUS status] [LOGlog] [SHOW_PROGRESS]
     [EXPECTED_MD5 sum])
```

DOWNLOAD下载指定URL的资源到指定的文件上。如果指定了LOG 参数，将会把下载的日志保存到相应的变量中。如果指定了STATUS变量，操作的状态信息就会保存在相应的变量中。返回的状态是一个长度为2的列表。第一个元素是操作的返回值。0表示操作过程中无错误发生。如果指定了TIMEOUT，单位于秒，且必须为整数，那么在指定的时间后，操作将会结束。如果指定了INACTIVITY_TIMEOUT，操作在处于活动状态超过指定的秒数后，应该停止。如果指定了EXPECTED_MD5，如果操作会检验下载后的文件的实际md5校验和是否与预期的匹配，如果不匹配，操作将会失败，并返回相应的错误码。如果指定了SHOW_PROGRESS，那么进度的信息将会被打印成状态信息直到操作完成。

```
file(UPLOADfilename url [INACTIVITY_TIMEOUT timeout]
     [TIMEOUT timeout] [STATUS status][LOG log] [SHOW_PROGRESS])
```

UPLOAD执行的是一个上传操作。参数含义与DOWNLOAD 一致。

```
file(<COPY|INSTALL> files... DESTINATION<dir>
     [FILE_PERMISSIONS permissions...]
     [DIRECTORY_PERMISSIONSpermissions...]
     [NO_SOURCE_PERMISSIONS][USE_SOURCE_PERMISSIONS]
     [FILES_MATCHING]
     [[PATTERN <pattern> | REGEX<regex>]
     [EXCLUDE] [PERMISSIONSpermissions...]] [...])
```

COPY表示复制文件，目录以及符号链接到一个目标文件夹中。输入路径将视为相对于当前源码目录的路径。目标路径则是相对于当前的构建目录。复制保留输入文件的一些权限属性。

13、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。

```
EXEC_PROGRAM(Executable [dir where to run] [ARGS <args>][OUTPUT_VARIABLE <var>]
[RETURN_VALUE <value>])
```

用于在指定目录运行某个程序（默认为当前CMakeLists.txt所在目录）,通过ARGS添加参数,通过OUTPUT_VARIABLE和RETURN_VALUE获取输出和返回值

举例，在src目录执行ls命令，并把结果和返回值存下来。

```
# 在src中运行ls命令,在src/CMakeLists.txt添加
EXEC_PROGRAM(ls ARGS "*.c" OUTPUT_VARIABLE LS_OUTPUT RETURN_VALUE LS_RVALUE)
IF (not LS_RVALUE)
    MESSAGE(STATUS "ls result: " ${LS_OUTPUT}) # 缩进仅为美观,语法无要求
ENDIF(not LS_RVALUE)
```

在cmake生成Makefile的过程中，就会执行ls命令，如果返回0，则会说明成功执行，那么就输出ls *.c的结果。

14、ENV

`$ENV{VAR}`是对环境变量VAR的引用，cmake支持变量嵌套引用，解引用的顺序从内到外。

```
message("PATH = $ENV{PATH}")
```

15、function

自定义函数命令格式如下：

```
function(<name> [arg1 [arg2 [arg3 ...]]])
# 自定义命令块
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
endfunction(<name>)
```

函数名为name，参数为arg1, arg2, arg3, ...的函数命令。参数之间用空格进行分隔，如果某一参数里面包含空格最好用双引号把该参数包起来（引号内的所有字符串只当作一个参数），比如"arg1"。如果不指定参数列表，则函数可以接受任意的参数，ARGC内置变量表明传入参数的个数，ARGV0, ARGV1, ARGV2, ...内置变量可以获得对应传入的参数，ARGV内置变量可以获得整个参数列表。

```
# print函数定义
function(print x y z)
  message("Calling function 'print':")
  message("  x = ${x}")
  message("  y = ${y}")
  message("  z = ${z}")
  message("ARGC = ${ARGC} arg1 = ${ARGV0} arg2 = ${ARGV1} arg3 = ${ARGV2} all
args = ${ARGV}")
# endfunction(print)
# 函数调用
print(1 2 3)
```

16、macro

自定义宏命令如下：

```
macro(<name> [arg1 [arg2 [arg3 ...]]])
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
endmacro(<name>
```

GNU开发工具——CMake模块

一、find_packge高级功能

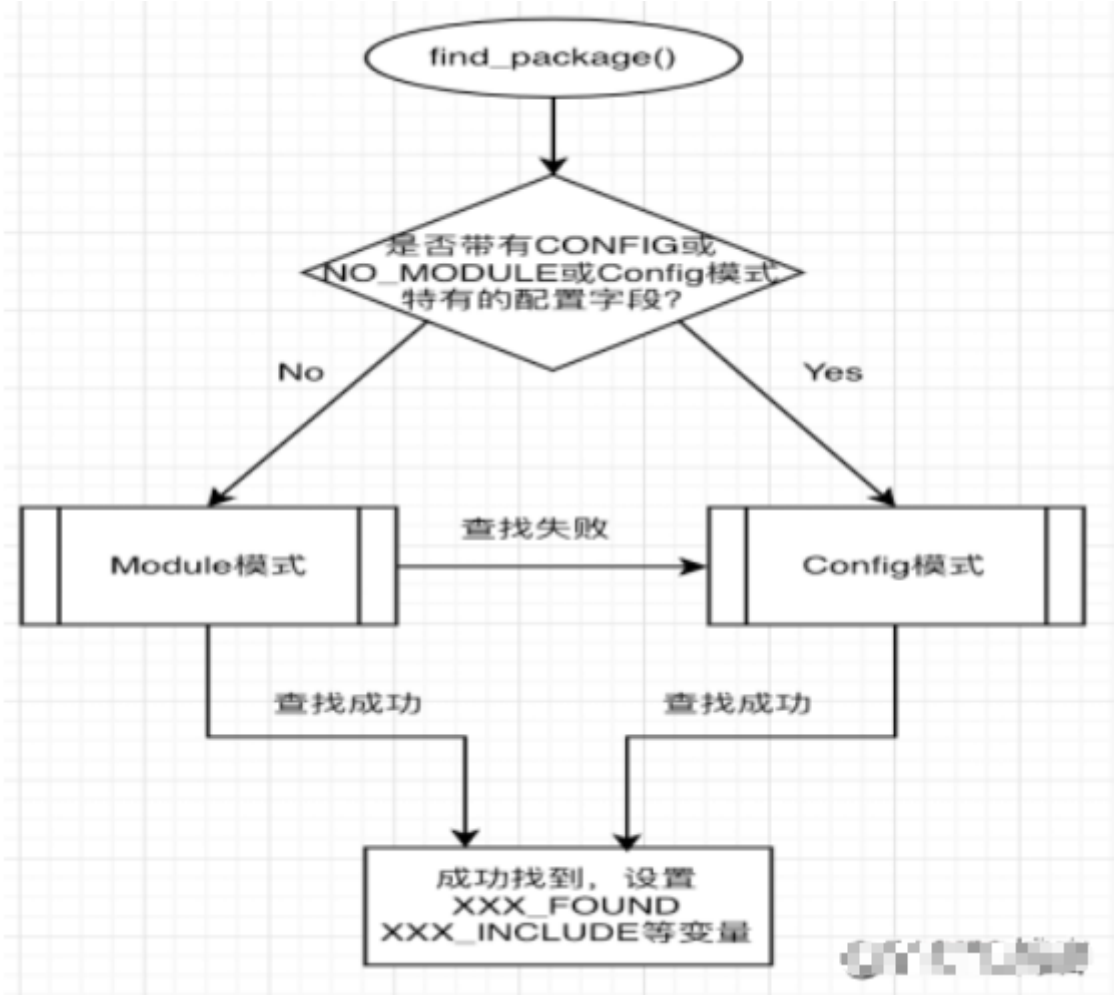
1、find_package的模式

find_package() 有Module模式(基本用法)和Config模式(完全用法)，其中Module模式是基础，Config模式则提供复杂高级功能。

`find_package`是否使用Config模式可以通过下列条件判断：

(1) `find_package()`中指定CONFIG关键字
(2) `find_package()`中指定NO_MODULE关键字
(3) `find_package()`中使用了不再Module模式下所有支持配置的关键字

2、find_package工作流程



3、find_package的Module模式

```
find_package(<PackageName> [version] [EXACT] [QUIET] [MODULE] [REQUIRED]
[[COMPONENTS] [components...]] [OPTIONAL_COMPONENTS components...]
[NO_POLICY_SCOPE])
```

精确地

version和EXACT，可选，version指定版本，如果指定就必须检查找到的包的版本是否和version兼容。如果指定EXACT则表示必须完全匹配的版本而不是兼容版本就可以。

QUIET，可选字段，表示如果查找失败，不会在屏幕进行输出（但如果指定了REQUIRED字段，则QUIET无效，仍然会输出查找失败提示语）。

MODULE，可选字段。前面提到说“如果Module模式查找失败则回退到Config模式进行查找”，但是假如设定了MODULE选项，那么就只在Module模式查找，如果Module模式下查找失败并不回落到Config模式查找。

REQUIRED可选字段。表示一定要找到包，找不到的话就立即停掉整个cmake。而如果不指定REQUIRED则cmake会继续执行。

COMPONENTS，components:可选字段，表示查找的包中必须要找到的组件(components），如果有任何一个找不到就算失败，类似于REQUIRED，导致cmake停止执行。

Module模式下需要查找到名为FindPackageName.cmake的文件。

先在CMAKE_MODULE_PATH变量对应的路径中查找。如果路径为空，或者路径中查找失败，则在cmake module directory（cmake安装时的Modules目录，比如/usr/local/share/cmake/Modules）查找。

/usr/share/cmake-3.16/Modules/FindProtobuf.cmake

4、find_package的Config模式

```
find_package(<PackageName> [version] [EXACT] [QUIET]
[REQUIRED] [[COMPONENTS] [components...]]
[CONFIG|NO_MODULE]
[NO_POLICY_SCOPE]
[NAMES name1 [name2 ...]]
[CONFIGS config1 [config2 ...]]
[HINTS path1 [path2 ... ]]
[PATHS path1 [path2 ... ]]
[PATH_SUFFIXES suffix1 [suffix2 ...]]
[NO_DEFAULT_PATH]
[NO_PACKAGE_ROOT_PATH]
[NO_CMAKE_PATH]
[NO_CMAKE_ENVIRONMENT_PATH]
[NO_SYSTEM_ENVIRONMENT_PATH]
[NO_CMAKE_PACKAGE_REGISTRY]
[NO_CMAKE_BUILDS_PATH] # Deprecated; does nothing.
[NO_CMAKE_SYSTEM_PATH]
[NO_CMAKE_SYSTEM_PACKAGE_REGISTRY]
[CMAKE_FIND_ROOT_PATH_BOTH |
ONLY_CMAKE_FIND_ROOT_PATH |
NO_CMAKE_FIND_ROOT_PATH])
```

Config模式下的查找顺序，比Module模式下要多得多，新版本的CMake比老版本的有更多的查找顺序（新增的在最优先的查找顺序）。Config模式下需要查找到名为lower-case-package-name-config.cmake或PackageNameConfig.cmake文件。查找顺序如下：

（1）名为PackageName_ROOT的cmake变量或环境变量。CMake3.12新增。设定CMP0074 Policy来关闭。如果定义了PackageName_DIR cmake变量，那么PackageName_ROOT 不起作用。

```
cmake_minimum_required(VERSION 3.13)
project(OpenCVDemo)
set(OpenCV_ROOT "/usr/local/lib/opencv_249/build")
set(OpenCV_DIR "/usr/local/lib/opencv_300/build")

find_package(OpenCV QUIET
    NO_MODULE
    NO_DEFAULT_PATH
    NO_CMAKE_PATH
    NO_CMAKE_ENVIRONMENT_PATH
    NO_SYSTEM_ENVIRONMENT_PATH
    NO_CMAKE_PACKAGE_REGISTRY
    NO_CMAKE_BUILDS_PATH
    NO_CMAKE_SYSTEM_PATH
    NO_CMAKE_SYSTEM_PACKAGE_REGISTRY
)
message(STATUS "OpenCV library status:")
message(STATUS "    version: ${OpenCV_VERSION}")
message(STATUS "    libraries: ${OpenCV_LIBS}")
```

```
message(STATUS "    include path: ${OpenCV_INCLUDE_DIRS}")
```

上述代码会找到opencv300，OpenCV_DIR变量的值有效OpenCV_ROOT变量无效。

（2）cmake特定的缓存变量

```
CMAKE_PREFIX_PATH
CMAKE_FRAMEWORK_PATH
CMAKE_APPBUNDLE_PATH
```

可以通过设定NO_CMAKE_PATH来关闭特定缓存变量的查找顺序

（3）CMake特定的环境变量

```
PackageName_DIR
CMAKE_PREFIX_PATH
CMAKE_FRAMEWORK_PATH
CMAKE_APPBUNDLE_PATH
```

可以通过NO_CMAKE_ENVIRONMENT_PATH来跳过。

（4）HINT字段指定的路径

（5）搜索标准的系统环境变量PATH。

其中如果是以/bin或者/sbin结尾的，会自动转化为其父目录。

通过指定NO_SYSTEM_ENVIRONMENT_PATH来跳过。

（6）存储在CMake的"User Package Registry"(用户包注册表)中的路径。

通过设定NO_CMAKE_PACKAGE_REGISTRY，或设定
CMAKE_FIND_PACKAGE_NO_PACKAGE_REGISTRY为true，来避开。

（7）设定为当前系统定义的cmake变量：

`CMAKE_SYSTEM_PREFIX_PATH`
`CMAKE_SYSTEM_FRAMEWORK_PATH`
`CMAKE_SYSTEM_APPBUNDLE_PATH`

通过设定NO_CMAKE_SYSTEM_PATH来跳过。

（8）在cmake的"System Package Registry"(系统包注册表）中查找。

通过设定NO_CMAKE_SYSTEM_PACKAGE_REGISTRY跳过，或者通过设定
CMAKE_FIND_PACKAGE_NO_SYSTEM_PACKAGE_REGISTRY为true避开。

（9）从PATHS字段指定的路径中查找。

二、CMake自定义模块

1、find_xxx指令

- `FIND_FILE(VAR name path1 path2 …)`

VAR变量代表找到的文件全路径，包含文件名。

- `FIND_LIBRARY(VAR name path1 path2 …)`

VAR变量代表找到的库全路径，包含库文件名。

- `FIND_PATH(VAR name path1 path2 …)`

VAR变量代表包含文件的路径

- `FIND_PROGRAM(VAR name path1 path2 …)`

VAR变量代表包含程序的全路径

- `FIND_PACKAGE(<name> [major.minor] [QUIET] [NO_MODULE] [[REQUIRED | COMPONENTS] [componets …]])`

用来调用预定义在CMAKE_MODULE_PATH下的Findname.cmake模块，可以自己定义Findname模块，通过

SET(CMAKE_MODULE_PATH dir)将其放入工程的某个目录供工程使用。

```
FIND_LIBRARY(libX X11 /usr/lib)
IF (NOT libx)
    MESSAGE(FATAL_ERROR "libX not found")
ENDIF(NOT libX)
```

2、系统预定义模块使用

系统预定义了各种模块，通常需要使用INCLUDE指令显式的调用, 但FIND_PACKAGE指令是一个特例，可以直接调用预定义的模块。

每一个模块都会定义以下3个变量：

```
<name>_FOUND
<name>_INCLUDE_DIR or <name>_INCLUDES
<name>_LIBRARY or <name>_LIBRARIES
```

因此，可以通过name_FOUND来判断模块是否被找到，如果没有找到，按照工程的需要关闭某些特性、给出提醒或者中止构建。

对于系统预定义的Findname.cmake模块，使用方法如下：

```
FIND_PACKAGE(NAME)
IF(NAME_FOUND)
    INCLUDE_DIRECTORIES(${NAME_INCLUDE_DIR})
    TARGET_LINK_LIBRARIES(targetname ${NAME_LIBRARY})
ELSE(NAME_FOUND)
    MESSAGE(FATAL_ERROR ”NAME library not found”)
ENDIF(NAME_FOUND)
```

工程实践中使用示例如下：

```
#通过<name>_FOUND来控制工程特性：
SET(SOURCES Viewer.cpp)
SET(OptionalSources)
SET(OptionalLibs)
FIND_PACKAGE(JPEG)
IF(JPEG_FOUND)
    SET(OptionalSources ${OptionalSources} JPEGView.cpp)
    INCLUDE_DIRECTORIES( ${JPEG_INCLUDE_DIR} )
    SET(optionalLibs ${OptionalLibs} ${JPEG_LIBRARIES} )
    ADD_DEFINITIONS(-DENABLE_JPEG_SUPPORT)
```

```
ENDIF(JPEG_FOUND)

IF(PNG_FOUND)
    SET(OptionalSources ${OptionalSources} PNGView.cpp)
    INCLUDE_DIRECTORIES( ${PNG_INCLUDE_DIR} )
    SET(OptionalLibs ${OptionalLibs} ${PNG_LIBRARIES} )
    ADD_DEFINITIONS(-DENABLE_PNG_SUPPORT)
ENDIF(PNG_FOUND)

ADD_EXECUTABLE(viewer ${SOURCES} ${OptionalSources} )
TARGET_LINK_LIBRARIES(Viewer ${OptionalLibs})
#通过判断系统是否提供了JPEG、PNG模块来决定程序是否支持JPEG、PNG功能。
```

3、自定义模块实现

本文基于libHello动态库，编写一个FindHelo.cmake模块。

创建一个FindHello项目目录，进入FindHello。

创建一个Hello目录，进入Hello目录，创建FindHelo.cmake文件，FindHelo.cmake文件如下：

```
#查找hello库头文件的安装路径
FIND_PATH(HELLO_INCLUDE_DIR Hello.h /usr/local/include/hello)
#查找hello库的安装路径
FIND_LIBRARY(HELLO_LIBRARY NAMES hello /usr/local/lib/libhello.so)

IF (HELLO_INCLUDE_DIR AND HELLO_LIBRARY)
    SET(HELLO_FOUND TRUE)
ENDIF (HELLO_INCLUDE_DIR AND HELLO_LIBRARY)

IF (HELLO_FOUND)
    IF (NOT HELLO_FIND_QUIETLY)
    MESSAGE(STATUS "Found Hello: ${HELLO_LIBRARY}")
    ENDIF (NOT HELLO_FIND_QUIETLY)
ELSE (HELLO_FOUND)
    IF (HELLO_FIND_REQUIRED)
    MESSAGE(FATAL_ERROR "Could not find hello library")
    ENDIF (HELLO_FIND_REQUIRED)
ENDIF (HELLO_FOUND)
```

返回FindHello目录，创建main.cpp文件：

```
#include <Hello.h>

int main(int argc, char* argv[])
{
    Hello hello;
    hello.Print();
    return 0;
}
```

创建工程CMakeLists.txt文件如下：

```
cmake_minimum_required(VERSION 2.8)
PROJECT(FindHello)
# 设置自定义模块路径
```

```
SET(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${PROJECT_SOURCE_DIR}/Hello/")
MESSAGE(STATUS "CMAKE_MODULE_PATH " ${PROJECT_SOURCE_DIR}/Hello)
#查找自定义Hello模块
FIND_PACKAGE(Hello)
IF(HELLO_FOUND)
    ADD_EXECUTABLE(hello main.cpp)
    INCLUDE_DIRECTORIES(${HELLO_INCLUDE_DIR})
    MESSAGE(STATUS "LINK " ${HELLO_LIBRARY})
    TARGET_LINK_LIBRARIES(hello ${HELLO_LIBRARY})
ELSE(HELLO_FOUND)
    MESSAGE(STATUS "Could not find hello library")
ENDIF(HELLO_FOUND)
```

创建build目录，进行build目录进行构建。

`cmake …/`

`make`

`./hello`

结果打印出Hello world.

GNU开发工具——CMake构建Qt工程实践

一、CMake构建Qt工程

1、Qt工程源码

创建Migration目录，在目录下创建main.cpp文件：

```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel label(QString("Hello Qt%1!").arg(int(QT_VERSION >> 16)));
    label.setAlignment(Qt::AlignCenter);
    label.resize(200, 100);
    label.show();
    return app.exec();
}
```

2、Qt4和Qt5兼容宏

编写兼容Qt4和Qt5的CMake宏，QtMigration.cmake文件如下：

```
# 定义宏QT_USE_MODULES
macro(QT_USE_MODULES _target)
    # Enable AUTOMOC
    set_target_properties(${_target} PROPERTIES AUTOMOC TRUE)
    # Local variables
    set(_modules_qt4)
    set(_modules_qt5)
    # Prepare modules
    foreach(_module ${ARGN})
```

```
        list(APPEND _modules_qt4 Qt${_module})
        list(APPEND _modules_qt5 ${_module})
        if(_module MATCHES "Gui")
            list(APPEND _modules_qt5 "Widgets")
        endif(_module MATCHES "Gui")
    endforeach(_module ${ARGN})
    list(REMOVE_DUPLICATES _modules_qt4)
    list(REMOVE_DUPLICATES _modules_qt5)
    # Find Qt libraries
    find_package(Qt5 QUIET COMPONENTS ${_modules_qt5})
    if(Qt5_FOUND)
        qt5_use_modules(${_target} ${_modules_qt5})
    else(Qt5_FOUND)
        find_package(Qt4 QUIET COMPONENTS ${_modules_qt4})
        if(Qt4_FOUND OR QT4_FOUND)
            include(${QT_USE_FILE})
            include_directories(${QT_INCLUDES})
            add_definitions(${QT_DEFINITIONS})
            target_link_libraries(${_target} ${QT_LIBRARIES})
        endif(Qt4_FOUND OR QT4_FOUND)
    endif(Qt5_FOUND)
endmacro(QT_USE_MODULES)
```

3、CMakeLists.txt编写

编写工程CMakeLists.txt文件，内容如下：

```
cmake_minimum_required(VERSION 2.8.9)
project(Migration)
list(APPEND CMAKE_MODULE_PATH ${CMAKE_CURRENT_LIST_DIR})
aux_source_directory(. SRC_LIST)
add_executable(${PROJECT_NAME} ${SRC_LIST})
include(QtMigration.cmake)
QT_USE_MODULES(${PROJECT_NAME} Core Gui)
```

4、构建工程

Migration工程内创建build目录，进入build目录，进行构建。

```
cmake ..
make
```

5、CMake的Qt相关变量

对于Qt4，使用FIND_PACKAGE后，会生成有效的Qt4_FOUND，QT_USE_FILE，QT_INCLUDES，QT_LIBRARIES变量。

```
FIND_PACKAGE(Qt4 REQUIRED Core Gui)
if(Qt4_FOUND)
    INCLUDE(${QT_USE_FILE})
    INCLUDE_DIRECTORIES(${QT_INCLUDES})
    TARGET_LINK_LIBRARIES(${PROJECT_NAME} ${QT_LIBRARIES})
endif()
```

对于Qt5，使用FIND_PACKAGE后，会生成有效的Qt5_FOUND，Qt5Core_INCLUDE_DIRS，Qt5Xml_INCLUDE_DIRS，Qt5Gui_INCLUDE_DIRS，Qt5Widgets_INCLUDE_DIRS，Qt5OpenGL_INCLUDE_DIRS，Qt5Widgets_LIBRARIES，Qt5Core_LIBRARIES，Qt5Gui_LIBRARIES，Qt5Xml_LIBRARIES，Qt5OpenGL_LIBRARIES等相应模块的变量。

```
FIND_PACKAGE(Qt5 REQUIRED Core Gui Widgets OpenGL Xml)
if(Qt5_FOUND)
    INCLUDE_DIRECTORIES(${Qt5Core_INCLUDE_DIRS} ${Qt5Xml_INCLUDE_DIRS}
        ${Qt5Gui_INCLUDE_DIRS} ${Qt5Widgets_INCLUDE_DIRS}
${Qt5OpenGL_INCLUDE_DIRS})
    #定义QT_LIBRARIES
    SET(QT_LIBRARIES ${Qt5Widgets_LIBRARIES} ${Qt5Core_LIBRARIES}
        ${Qt5Gui_LIBRARIES} ${Qt5Xml_LIBRARIES} ${Qt5OpenGL_LIBRARIES})
    TARGET_LINK_LIBRARIES(${PROJECT_NAME} ${QT_LIBRARIES})
endif(Qt5_FOUND)
```

开启MOC支持

- `set(CMAKE_AUTOMOC ON)`

开启RCC支持

- `set(CMAKE_AUTORCC ON)`

开启UIC支持

- `set(CMAKE_AUTOUIC ON)`

设置Qt安装目录：

- `set(CMAKE_PREFIX_PATH "Qt to path ") //bin lib include目录层`