CMake 入门实战

从实例入手，讲解 CMake 的常见用法。

什么是 CMake

> All problems in computer science can be solved by another level of indirection.
>
> **David Wheeler**

你或许听过好几种 Make 工具，例如 GNU Make ，QT 的 qmake ，微软的 MS nmake，BSD Make（pmake），Makepp，等等。这些 Make 工具遵循着不同的规范和标准，所执行的 Makefile 格式也千差万别。这样就带来了一个严峻的问题：如果软件想跨平台，必须要保证能够在不同平台编译。而如果使用上面的 Make 工具，就得为每一种标准写一次 Makefile ，这将是一件让人抓狂的工作。

CMake 就是针对上面问题所设计的工具：它首先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 Unix 的 Makefile 或 Windows 的 Visual Studio 工程。从而做到“Write once, run everywhere”。显然，CMake 是一个比上述几种 make 更高级的编译配置工具。一些使用 CMake 作为项目架构系统的知名开源项目有 VTK、ITK、KDE、OpenCV、OSG 等 [1]。

在 linux 平台下使用 CMake 生成 Makefile 并编译的流程如下：

1. 编写 CMake 配置文件 CMakeLists.txt 。
2. 执行命令 `cmake PATH` 或者 `ccmake PATH` 生成 Makefile（`ccmake` 和 `cmake` 的区别在于前者提供了一个交互式的界面）。其

中， `PATH` 是 CMakeLists.txt 所在的目录。
3. 使用 `make` 命令进行编译。

本文将从实例入手，一步步讲解 CMake 的常见用法，文中所有的实例代码可以在这里找到。如果你读完仍觉得意犹未尽，可以继续学习我在文章末尾提供的其他资源。

入门案例：单个源文件

本节对应的源代码所在目录：Demo1。

对于简单的项目，只需要写几行代码就可以了。例如，假设现在我们的项目中只有一个源文件 main.cc ，该程序的用途是计算一个数的指数幂。

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /**
5   * power - Calculate the power of number.
6   * @param base: Base value.
7   * @param exponent: Exponent value.
8   *
9   * @return base raised to the power exponent.
10  */
11 double power(double base, int exponent)
12 {
13     int result = base;
14     int i;
15
16     if (exponent == 0) {
17         return 1;
18     }
19
20     for(i = 1; i < exponent; ++i){
21         result = result * base;
22     }
23
24     return result;
25 }
26
27 int main(int argc, char *argv[])
28 {
29     if (argc < 3){
30         printf("Usage: %s base exponent \n", argv[0]);
31         return 1;
32     }
33     double base = atof(argv[1]);
34     int exponent = atoi(argv[2]);
35     double result = power(base, exponent);
36     printf("%g ^ %d is %g\n", base, exponent, result);
37     return 0;
38 }
```

编写 CMakeLists.txt

首先编写 CMakeLists.txt 文件，并保存在与 main.cc 源文件同个目录下：

```
1 # CMake 最低版本号要求
2 cmake_minimum_required (VERSION 2.8)
3
4 # 项目信息
5 project (Demo1)
6
7 # 指定生成目标
8 add_executable(Demo main.cc)
```

CMakeLists.txt 的语法比较简单，由命令、注释和空格组成，其中命令是不区分大小写的。符号 # 后面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。

对于上面的 CMakeLists.txt 文件，依次出现了几个命令：

1. `cmake_minimum_required`：指定运行此配置文件所需的 CMake 的最低版本；
2. `project`：参数值是 `Demo1`，该命令表示项目的名称是 `Demo1` 。
3. `add_executable`： 将名为 main.cc 的源文件编译成一个名称为 Demo 的可执行文件。

编译项目

之后，在当前目录执行 `cmake .`，得到 Makefile 后再使用 `make` 命令编译得到 Demo1 可执行文件。

```
[ehome@xman Demo1]$ cmake .
-- The C compiler identification is GNU 4.8.2
-- The CXX compiler identification is GNU 4.8.2
-- Check for working C compiler: /usr/sbin/cc
-- Check for working C compiler: /usr/sbin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/sbin/c++
-- Check for working CXX compiler: /usr/sbin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ehome/Documents/programming/C/power/Demo1
[ehome@xman Demo1]$ make
Scanning dependencies of target Demo
[100%] Building C object CMakeFiles/Demo.dir/main.cc.o
Linking C executable Demo
[100%] Built target Demo
[ehome@xman Demo1]$ ./Demo 5 4
5 ^ 4 is 625
[ehome@xman Demo1]$ ./Demo 7 3
7 ^ 3 is 343
[ehome@xman Demo1]$ ./Demo 2 10
2 ^ 10 is 1024
```

多个源文件

同一目录，多个源文件

本小节对应的源代码所在目录：Demo2。

上面的例子只有单个源文件。现在假如把 `power` 函数单独写进一个名为 `MathFunctions.c` 的源文件里，使得这个工程变成如下的形式：

```
./Demo2
    |
    +--- main.cc
    |
    +--- MathFunctions.cc
    |
    +--- MathFunctions.h
```

这个时候，CMakeLists.txt 可以改成如下的形式：

```
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)

# 项目信息
project (Demo2)

# 指定生成目标
add_executable(Demo main.cc MathFunctions.cc)
```

唯一的改动只是在 `add_executable` 命令中增加了一个 `MathFunctions.cc` 源文件。这样写当然没什么问题，但是如果源文件很多，把所有源文件的名字都加进去将是一件烦人的工作。更省事的方法是使用 `aux_source_directory` 命令，该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>)
```

因此，可以修改 CMakeLists.txt 如下：

```
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)

# 项目信息
project (Demo2)

# 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})
```

这样，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。

多个目录，多个源文件

本小节对应的源代码所在目录：Demo3。

现在进一步将 MathFunctions.h 和 MathFunctions.cc 文件移动到 math 目录下。

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

对于这种情况，需要分别在项目根目录 Demo3 和 math 目录里各编写一个 CMakeLists.txt 文件。为了方便，我们可以先将 math 目录里的文件编译成静态库再由 main 函数调用。

如果想学习不使用静态库的处理，可以看看 @zhc2019github 贡献的一个示例。

根目录中的 CMakeLists.txt：

```
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)

# 项目信息
project (Demo3)

# 查找当前目录下的所有源文件
# 并将名称保存到 DIR_SRCS 变量
aux_source_directory(. DIR_SRCS)

# 添加 math 子目录
add_subdirectory(math)

# 指定生成目标
add_executable(Demo main.cc)

# 添加链接库
target_link_libraries(Demo MathFunctions)
```

该文件添加了下面的内容: 第3行，使用命令 `add_subdirectory` 指明本项目包含一个子目录 math，这样 math 目录下的 CMakeLists.txt 文件和源代码也会被处理 。第6行，使用命令 `target_link_libraries` 指明可执行文件 main 需要连接一个名为 MathFunctions 的链接库 。

子目录中的 CMakeLists.txt：

```
# 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})
```

在该文件中使用命令 `add_library` 将 src 目录中的源文件编译为静态链接库。

自定义编译选项

本节对应的源代码所在目录：Demo4。

CMake 允许为项目增加编译选项，从而可以根据用户的环境和需求选择最合适的编译方案。

例如，可以将 MathFunctions 库设为一个可选的库，如果该选项为 `ON` ，就使用该库定义的数学函数来进行运算。否则就调用标准库中的数学函数库。

修改 CMakeLists 文件

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

```
# CMake 最低版本号要求
cmake_minimum_required (VERSION 2.8)

# 项目信息
project (Demo4)

# 加入一个配置头文件，用于处理 CMake 对源码的设置
configure_file (
  "${PROJECT_SOURCE_DIR}/config.h.in"
  "${PROJECT_BINARY_DIR}/config.h"
  )

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

# 是否加入 MathFunctions 库
if (USE_MYMATH)
  include_directories ("${PROJECT_SOURCE_DIR}/math")
  add_subdirectory (math)
  set (EXTRA_LIBS ${EXTRA_LIBS} MathFunctions)
endif (USE_MYMATH)

# 查找当前目录下的所有源文件
# 并将名称保存到 DIR_SRCS 变量
aux_source_directory(. DIR_SRCS)

# 指定生成目标
add_executable(Demo ${DIR_SRCS})
target_link_libraries (Demo  ${EXTRA_LIBS})
```

其中：

1. 第7行的 `configure_file` 命令用于加入一个配置头文件 config.h ，这个文件由 CMake 从 config.h.in 生成，通过这样的机制，将可以通过预定义一些参数和变量来控制代码的生成。
2. 第13行的 `option` 命令添加了一个 `USE_MYMATH` 选项，并且默认值为 `ON` 。
3. 第17行根据 `USE_MYMATH` 变量的值来决定是否使用我们自己编写的 MathFunctions 库。

修改 main.cc 文件

之后修改 main.cc 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：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "config.h"
4
5  #ifdef USE_MYMATH
6    #include "math/MathFunctions.h"
7  #else
8    #include <math.h>
9  #endif
10
11
12 int main(int argc, char *argv[])
13 {
14     if (argc < 3){
15         printf("Usage: %s base exponent \n", argv[0]);
16         return 1;
17     }
18     double base = atof(argv[1]);
19     int exponent = atoi(argv[2]);
20
21 #ifdef USE_MYMATH
22     printf("Now we use our own Math library. \n");
23     double result = power(base, exponent);
24 #else
25     printf("Now we use the standard library. \n");
26     double result = pow(base, exponent);
27 #endif
28     printf("%g ^ %d is %g\n", base, exponent, result);
29     return 0;
30 }
```

编写 config.h.in 文件

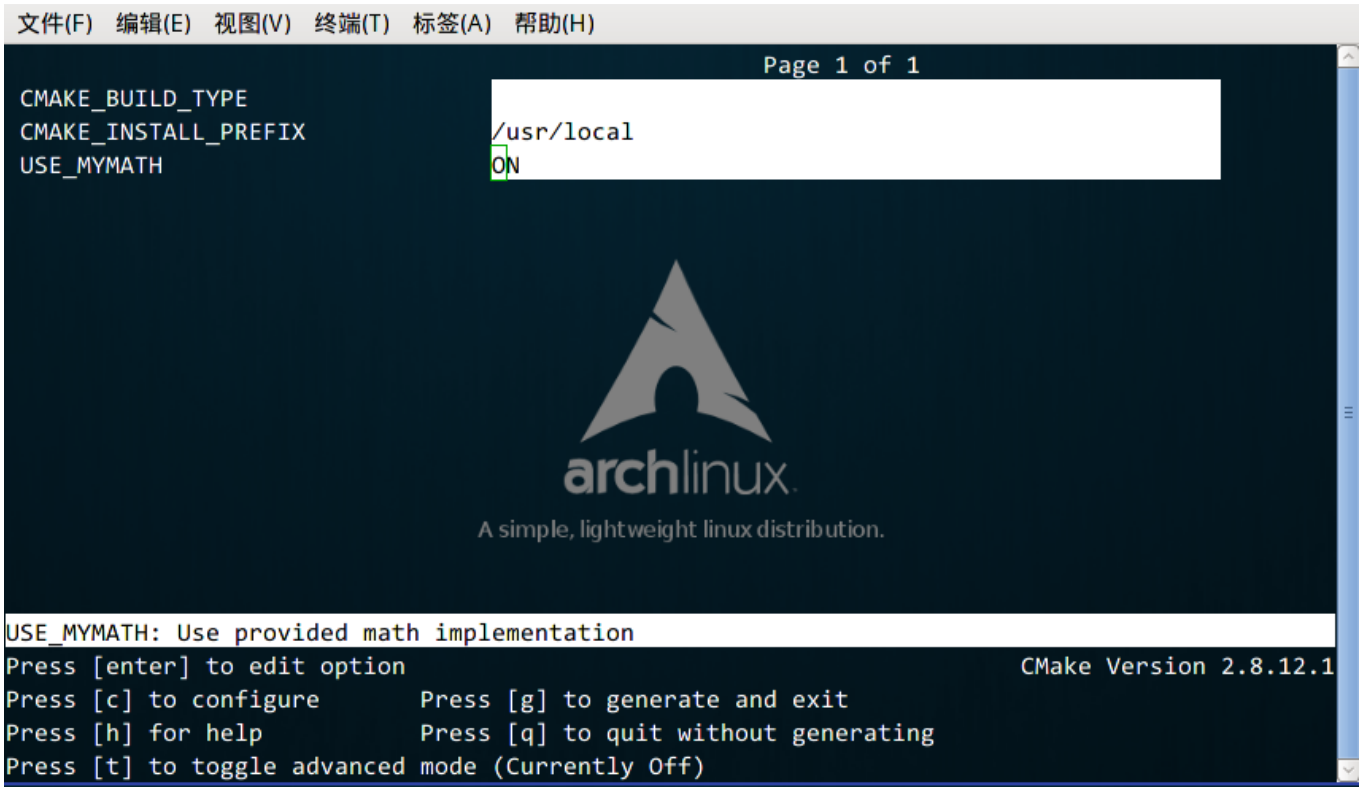
上面的程序值得注意的是第2行，这里引用了一个 config.h 文件，这个文件预定义了 `USE_MYMATH` 的值。但我们并不直接编写这个文件，为了方便从 CMakeLists.txt 中导入配置，我们编写一个 config.h.in 文件，内容如下：

```
1 #cmakedefine USE_MYMATH
```

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

编译项目

现在编译一下这个项目，为了便于交互式的选择该变量的值，可以使用 `ccmake` 命令（也可以使用 `cmake -i` 命令，该命令会提供一个会话式的交互式配置界面）：



CMake的交互式配置界面

从中可以找到刚刚定义的 `USE_MYMATH` 选项，按键盘的方向键可以在不同的选项窗口间跳转，按下 `enter` 键可以修改该选项。修改完成后可以按下 `c` 选项完成配置，之后再按 `g` 键确认生成 Makefile 。ccmake 的其他操作可以参考窗口下方给出的指令提示。

我们可以试试分别将 `USE_MYMATH` 设为 `ON` 和 `OFF` 得到的结果：

USE_MYMATH 为 ON

运行结果：

```
1 [ehome@xman Demo4]$ ./Demo
2 Now we use our own MathFunctions library.
3  7 ^ 3 = 343.000000
4  10 ^ 5 = 100000.000000
5  2 ^ 10 = 1024.000000
```

此时 config.h 的内容为：

```
1 #define USE_MYMATH
```

USE_MYMATH 为 OFF

运行结果：

```
1 [ehome@xman Demo4]$ ./Demo
2 Now we use the standard library.
3  7 ^ 3 = 343.000000
4  10 ^ 5 = 100000.000000
5  2 ^ 10 = 1024.000000
```

此时 config.h 的内容为：

```
1 /* #undef USE_MYMATH */
```

安装和测试

本节对应的源代码所在目录：Demo5。

CMake 也可以指定安装规则，以及添加测试。这两个功能分别可以通过在产生 Makefile 后使用 `make install` 和 `make test` 来执行。在以前的 GNU Makefile 里，你可能需要为此编写 `install` 和 `test` 两个伪目标和相应的规则，但在 CMake 里，这样的工作同样只需要简单的调用几条命令。

定制安装规则

首先先在 math/CMakeLists.txt 文件里添加下面两行：

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

指明 MathFunctions 库的安装路径。之后同样修改根目录的 CMakeLists 文件，在末尾添加下面几行：

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

通过上面的定制，生成的 Demo 文件和 MathFunctions 函数库 libMathFunctions.o 文件将会被复制到 `/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。我们可以验证一下（顺带一提的是，这里的 `/usr/local/` 是默认安装到的根目录，可以通过修改 `CMAKE_INSTALL_PREFIX` 变量的值来指定这些文件应该拷贝到哪个根目录）：

```
1  [ehome@xman Demo5]$ sudo make install
2  [ 50%] Built target MathFunctions
3  [100%] Built target Demo
4  Install the project...
5  -- Install configuration: ""
6  -- Installing: /usr/local/bin/Demo
7  -- Installing: /usr/local/include/config.h
8  -- Installing: /usr/local/bin/libMathFunctions.a
9  -- Up-to-date: /usr/local/include/MathFunctions.h
10 [ehome@xman Demo5]$ ls /usr/local/bin
11 Demo  libMathFunctions.a
12 [ehome@xman Demo5]$ ls /usr/local/include
13 config.h  MathFunctions.h
```

为工程添加测试

添加测试同样很简单。CMake 提供了一个称为 CTest 的测试工具。我们要做的只是在项目根目录的 CMakeLists 文件中调用一系列的 `add_test` 命令。

```
1  # 启用测试
2  enable_testing()
3
4  # 测试程序是否成功运行
5  add_test (test_run Demo 5 2)
6
7  # 测试帮助信息是否可以正常提示
8  add_test (test_usage Demo)
9  set_tests_properties (test_usage
10   PROPERTIES PASS_REGULAR_EXPRESSION "Usage: .* base exponent")
11
12 # 测试 5 的平方
13 add_test (test_5_2 Demo 5 2)
```

```
14
15 set_tests_properties (test_5_2
16  PROPERTIES PASS_REGULAR_EXPRESSION "is 25")
17
18 # 测试 10 的 5 次方
19 add_test (test_10_5 Demo 10 5)
20
21 set_tests_properties (test_10_5
22  PROPERTIES PASS_REGULAR_EXPRESSION "is 100000")
23
24 # 测试 2 的 10 次方
25 add_test (test_2_10 Demo 2 10)
26
27 set_tests_properties (test_2_10
28  PROPERTIES PASS_REGULAR_EXPRESSION "is 1024")
```

上面的代码包含了四个测试。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。

让我们看看测试的结果：

```
1  [ehome@xman Demo5]$ make test
2  Running tests...
3  Test project /home/ehome/Documents/programming/C/power/Demo5
4      Start 1: test_run
5  1/4 Test #1: test_run .........................   Passed    0.00 sec
6      Start 2: test_5_2
7  2/4 Test #2: test_5_2 .........................   Passed    0.00 sec
8      Start 3: test_10_5
9  3/4 Test #3: test_10_5 ........................   Passed    0.00 sec
10     Start 4: test_2_10
11 4/4 Test #4: test_2_10 ........................   Passed    0.00 sec
12
13 100% tests passed, 0 tests failed out of 4
14
15 Total Test time (real) =   0.01 sec
```

如果要测试更多的输入数据，像上面那样一个个写测试用例未免太繁琐。这时可以通过编写宏来实现：

```
1  # 定义一个宏，用来简化测试工作
2  macro (do_test arg1 arg2 result)
3    add_test (test_${arg1}_${arg2} Demo ${arg1} ${arg2})
4    set_tests_properties (test_${arg1}_${arg2}
5      PROPERTIES PASS_REGULAR_EXPRESSION ${result})
6  endmacro (do_test)
7
8  # 使用该宏进行一系列的数据测试
9  do_test (5 2 "is 25")
10 do_test (10 5 "is 100000")
11 do_test (2 10 "is 1024")
```

关于 CTest 的更详细的用法可以通过 `man 1 ctest` 参考 CTest 的文档。

支持 gdb

让 CMake 支持 gdb 的设置也很容易，只需要指定 `Debug` 模式下开启 `-g` 选项：

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

之后可以直接对生成的程序使用 gdb 来调试。

添加环境检查

本节对应的源代码所在目录：Demo6。

有时候可能要对系统环境做点检查，例如要使用一个平台相关的特性的时候。在这个例子中，我们检查系统是否自带 pow 函数。如果带有 pow 函数，就使用它；否则使用我们定义的 power 函数。

添加 CheckFunctionExists 宏

首先在顶层 CMakeLists 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。

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

将上面这段代码放在 `configure_file` 命令前。

预定义相关宏变量

接下来修改 config.h.in 文件，预定义相关的宏变量。

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

在代码中使用宏和函数

最后一步是修改 main.cc ，在代码中使用宏和函数：

```
1 #ifdef HAVE_POW
2     printf("Now we use the standard library. \n");
3     double result = pow(base, exponent);
4 #else
5     printf("Now we use our own Math library. \n");
6     double result = power(base, exponent);
7 #endif
```

添加版本号

本节对应的源代码所在目录：Demo7。

给项目添加和维护版本号是一个好习惯，这样有利于用户了解每个版本的维护情况，并及时了解当前所用的版本是否过时，或是否可能出现不兼容的情况。

首先修改顶层 CMakeLists 文件，在 `project` 命令之后加入如下两行：

```
1 set (Demo_VERSION_MAJOR 1)
2 set (Demo_VERSION_MINOR 0)
```

分别指定当前的项目的主版本号和副版本号。

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

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

这样就可以直接在代码中打印版本信息了：

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

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]);

#if defined (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;
}
```

生成安装包

本节对应的源代码所在目录：Demo8。

本节将学习如何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。为了完成这个任务，我们需要用到 CPack，它同样也是由 CMake 提供的一个工具，专门用于打包。

首先在顶层的 CMakeLists.txt 文件尾部添加下面几行：

```
# 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)
```

上面的代码做了以下几个工作：

1. 导入 InstallRequiredSystemLibraries 模块，以便之后导入 CPack 模块；
2. 设置一些 CPack 相关变量，包括版权信息和版本信息，其中版本信息用了上一节定义的版本号；
3. 导入 CPack 模块。

接下来的工作是像往常一样构建工程，并执行 `cpack` 命令。

- 生成二进制安装包：

```
cpack -C CPackConfig.cmake
```

- 生成源码安装包

```
cpack -C CPackSourceConfig.cmake
```

我们可以试一下。在生成项目后，执行 `cpack -C CPackConfig.cmake` 命令：

```
[ehome@xman Demo8]$ cpack -C CPackSourceConfig.cmake
CPack: Create package using STGZ
CPack: Install projects
CPack: - Run preinstall target for: Demo8
CPack: - Install project: Demo8
CPack: Create package
CPack: - package: /home/ehome/Documents/programming/C/power/Demo8/Demo8-1.0.1-Linux.sh generated.
CPack: Create package using TGZ
CPack: Install projects
CPack: - Run preinstall target for: Demo8
CPack: - Install project: Demo8
CPack: Create package
CPack: - package: /home/ehome/Documents/programming/C/power/Demo8/Demo8-1.0.1-Linux.tar.gz generated.
CPack: Create package using TZ
CPack: Install projects
CPack: - Run preinstall target for: Demo8
CPack: - Install project: Demo8
CPack: Create package
CPack: - package: /home/ehome/Documents/programming/C/power/Demo8/Demo8-1.0.1-Linux.tar.Z generated.
```

此时会在该目录下创建 3 个不同格式的二进制包文件：

```
[ehome@xman Demo8]$ ls Demo8-*
Demo8-1.0.1-Linux.sh  Demo8-1.0.1-Linux.tar.gz  Demo8-1.0.1-Linux.tar.Z
```

这 3 个二进制包文件所包含的内容是完全相同的。我们可以执行其中一个。此时会出现一个由 CPack 自动生成的交互式安装界面：

```
[ehome@xman Demo8]$ sh Demo8-1.0.1-Linux.sh
Demo8 Installer Version: 1.0.1, Copyright (c) Humanity
This is a self-extracting archive.
The archive will be extracted to: /home/ehome/Documents/programming/C/power/Demo8

If you want to stop extracting, please press <ctrl-C>.
The MIT License (MIT)

Copyright (c) 2013 Joseph Pan(http://hahack.com)

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.


Do you accept the license? [yN]:
y
By default the Demo8 will be installed in:
  "/home/ehome/Documents/programming/C/power/Demo8/Demo8-1.0.1-Linux"
Do you want to include the subdirectory Demo8-1.0.1-Linux?
Saying no will install in: "/home/ehome/Documents/programming/C/power/Demo8" [Yn]:
y

Using target directory: /home/ehome/Documents/programming/C/power/Demo8/Demo8-1.0.1-Linux
Extracting, please wait...

Unpacking finished successfully
```

完成后提示安装到了 Demo8-1.0.1-Linux 子目录中，我们可以进去执行该程序：

```
[ehome@xman Demo8]$ ./Demo8-1.0.1-Linux/bin/Demo 5 2
Now we use our own Math library.
5 ^ 2 is 25
```

关于 CPack 的更详细的用法可以通过 `man 1 cpack` 参考 CPack 的文档。

将其他平台的项目迁移到 CMake

CMake 可以很轻松地构建出在适合各个平台执行的工程环境。而如果当前的工程环境不是 CMake ，而是基于某个特定的平台，是否可以迁移到 CMake 呢？答案是可能的。下面针对几个常用的平台，列出了它们对应的迁移方案。

autotools

- am2cmake 可以将 autotools 系的项目转换到 CMake，这个工具的一个成功案例是 KDE 。
- Alternative Automake2CMake 可以转换使用 automake 的 KDevelop 工程项目。
- Converting autoconf tests

qmake

- qmake converter 可以转换使用 QT 的 qmake 的工程。

Visual Studio

- vcproj2cmake.rb 可以根据 Visual Studio 的工程文件（后缀名是 `.vcproj` 或 `.vcxproj`）生成 CMakeLists.txt 文件。
- vcproj2cmake.ps1 vcproj2cmake 的 PowerShell 版本。
- folders4cmake 根据 Visual Studio 项目文件生成相应的 “source_group” 信息，这些信息可以很方便的在 CMake 脚本中使用。支持 Visual Studio 9/10 工程文件。

CMakeLists.txt 自动推导

- gencmake 根据现有文件推导 CMakeLists.txt 文件。
- CMakeListGenerator 应用一套文件和目录分析创建出完整的 CMakeLists.txt 文件。仅支持 Win32 平台。

相关链接

1. 官方主页
2. 官方文档
3. 官方教程
4. Wiki
5. FAQ
6. bug tracker
7. 邮件列表：
    - cmake on Gmane
    - http://www.mail-archive.com/cmake@cmake.org/
    - http://marc.info/?l=cmake
8. 其他推荐文章
    - 在 linux 下使用 CMake 构建应用程序
    - cmake的一些小经验
    - Packaging Software with CPack
    - 视频教程: 《Getting Started with CMake》

类似工具

- SCons：Eric S. Raymond、Timothee Besset、Zed A. Shaw 等大神力荐的项目架构工具。和 CMake 的最大区别是使用 Python 作为执行脚本。
