CMake官网教程地址: https://cmake.org/cmake/help/latest/guide/tutorial/index.html

介绍

这份渐进式的CMake教程覆盖了构建系统时CMake来处理的一些常见的问题.在一个样例项目中来探讨不同主题是如何结合应用是非常有助于理解的.教程文档和源码可以从CMake源路径下的`Help/guide/tutorial`文件夹中获得(译者注:已同步至当前Git).每一步都有其独立的子目录,这些子目录也包含可以作为出发点的代码.教程样例是循序渐进的,每一步都提供了前一步的完整解决方案.

Step1: 一个基本出发点

最基础的项目是基于源代码的一个可执行构建.对于简单项目.三行的`CMakeLists.txt`就满足了全部需要的内容.这就是这篇教程的开始点.在`Step1`路径下创建一个`CMakeLists.txt`文件如下:

```
cmake_minimum_required(VERSION 3.10)

# set the project name
project(Tutorial)

# add the executable
add_executable(Tutorial tutorial.cxx)
```

注意在`CMakeLists.txt`文件中的命令都使用了小写.CMake支持大小写混用命令.`tutorial.cxx`的源代码在`Step1`文件夹下,可用以计算平方根.

添加版本号 & 配置头文件

首个添加的特性是给我们的项目和可执行文件提供版本号.尽管我们可以在源代码中添加版本号,但是使用`CMakeLists.txt`是更灵活的方式.

首先,修改`CMakeLists.txt`,使用`project()`命令来设定项目名和版本号.

```
cmake_minimum_required(VERSION 3.10)

# set the project name and version
project(Tutorial VERSION 1.0)
```

然后制定一个头文件来将版本号传递到源码里:

```
configure_file(TutorialConfig.h.in TutorialConfig.h)
```

因为指定的文件会被写入二进制结构中,我们必须将这一目录添加到搜索include文件的列表中.在 `CMakeLists.txt` 文件结尾写入:

```
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           )
```

使用你喜欢的IDE,在源路径下创建 `TutorialConfig.h.in` ,并写入下述内容:

```
// the configured options and settings for Tutorial
#define Tutorial_VERSION_MAJOR @Tutorial_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @Tutorial_VERSION_MINOR@
```

当CMake生成这个头文件时, `@Tutorial_VERSION_MAJOR@` 和 `@Tutorial_VERSION_MINOR@` 的值会被自定替换.

接下来调整 `tutorial.cxx` 来包含头文件 `TutorialConfig.h` .

最后,更新 `tutorial.cxx` 如下以打印可执行文件名和版本号:

```
    if (argc < 2) {
    // report version
    std::cout << argv[0] << " Version " << Tutorial_VERSION_MAJOR << "."
                << Tutorial_VERSION_MINOR << std::endl;
    std::cout << "Usage: " << argv[0] << " number" << std::endl;
    return 1;
    }
```

指定C++标准

接下来,我们通过在 `tutorial.cxx` 中将 `atof` 替换为 `std::stod` 来给我们的项目增加一些C++11特性.同时,移除 `#include <cstdlib>` .

```
const double inputValue = std::stod(argv[1]);
```

我们需要在CMake代码中显式地声明以使用正确的配置.最简单的方式是在CMake中通过使用 `CMAKE_CXX_STANDARD` 以启用对特定版本C++标准的支持.对于本篇教程.

将`CMakeLists.txt`中的`CMAKE_CXX_STANDARD`设为11,`CMAKE_CXX_STANDARD_REQUIRED`设为`True`.并将`CMAKE_CXX_STANDARD`声明置于`add_executable`前.

```
cmake_minimum_required(VERSION 3.10)

# set the project name and version
project(Tutorial VERSION 1.0)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

构建与测试

运行`cmake`可执行文件,或者`cmake-gui`来配置项目,然后使用所选的构建工具来构建它.

例如,从命令行中,我们要进入`Help/guide/tutorial`目录下并建立一个build目录:

```
mkdir Step1_build
```

之后,进入build目录,然后运行CMake来配置项目,并生成原生构建系统:

```
cd Step1_build
cmake ../Step1
```

然后调用这个构建系统来实际编译/链接项目:

```
cmake --build .
```

最后,尝试用下述命令来使用新构建的`Tutorial`:

```
Tutorial 4294967296
Tutorial 10
Tutorial
```

添加库

现在我们会向我们的项目中添加一个库.这个库会包含我们计算数字平方根的实现.可执行文件就可以使用库而非编译器提供的平方根函数.

本篇教程里,我们会把库放在一个叫做`MathFunctions`的子文件夹下.这个目录已经包含了一个头文件`MathFunctions.h`,也包含了一个源文件`mysqrt.cxx`. 源文件中包含一个名为`mysqrt`的函数,提供了编译器中`sqrt`相近功能.

向`MathFunctions`文件夹中新增下面的单行`CMakeLists.txt`文件:

```
add_library(MathFunctions mysqrt.cxx)
```

为了使用新的库,我们在顶级的`CMakeLists.txt`中加入`add_subdirectory()`来构建库.我们向可执行文件加入新的库,并将`MathFunctions`添加为包含目录,这样就可以查询得到`mqsqrt.h`头文件了.顶级`CMakeLists.txt`的最后几行应该如下:

```
# add the MathFunctions library
add_subdirectory(MathFunctions)

# add the executable
add_executable(Tutorial tutorial.cxx)

target_link_libraries(Tutorial PUBLIC MathFunctions)

# add the binary tree to the search path for include files
# so that we will find TutorialConfig.h
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           "${PROJECT_SOURCE_DIR}/MathFunctions"
                           )
```

接下来我们让MathFunctions库可以作为可选项.尽管本次教程不需要这样,但大型项目中这很常见.第一步是在顶层`CMakeLists.txt`中增加选项:

```
option(USE_MYMATH "Use tutorial provided math implementation" ON)

# configure a header file to pass some of the CMake settings
# to the source code
configure_file(TutorialConfig.h.in TutorialConfig.h)
```

这一选项会在`cmake-gui`或`ccmake`中显示,默认值为ON,也可被用户修改.这一选项会被存储在缓存中,用户无需每次都设定.

下一项更改是将MathFunctions库的构建和链接设定成可选的.我们在顶级`CMakeLists.txt`的结尾做如下修改:

```
if(USE_MYMATH)
  add_subdirectory(MathFunctions)
  list(APPEND EXTRA_LIBS MathFunctions)
  list(APPEND EXTRA_INCLUDES "${PROJECT_SOURCE_DIR}/MathFunctions")
endif()

# add the executable
add_executable(Tutorial tutorial.cxx)

target_link_libraries(Tutorial PUBLIC ${EXTRA_LIBS})

# add the binary tree to the search path for include files
# so that we will find TutorialConfig.h
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           ${EXTRA_INCLUDES}
                           )
```

变量`EXTRA_LIBS`收集了之后可以在可执行文件中链接的可选库.变量`EXTRA_INCLUDES`也相应的用于收集可选的头文件.在处理很多可选项时,这是一种经典的处理方式.下一步我们会用新方式来做.

相应的源代码改动就比较直接了.首先,在`tutorial.cxx`中,如果需要则包含`MathFunctions.h`:

```
#ifdef USE_MYMATH
#  include "MathFunctions.h"
#endif
```

然后在同一个文件中,让`USE_MYMATH`变量控制函数的选择:

```
#ifdef USE_MYMATH
  const double outputValue = mysqrt(inputValue);
#else
  const double outputValue = sqrt(inputValue);
#endif
```

因为现在源代码中需要`USE_MYMATH`变量,我们可以在`TutorialConfig.h.in`文件中加入下述这行:

```
#cmakedefine USE_MYMATH
```

练习: 为什么我们在选项`USE_MYMATH`后配置`TutorialConfig.h.in`.如果我们把这两条交换会发生什么.

运行`cmake`或者`cmake-gui`来配置项目,然后构建,在运行构建出的可执行文件.

现在让我们更新`USE_MYMATH`的值.最简单的方式是使用`cmake-gui`或终端中的`ccmake`.或者如果想在命令行中修改这一选项:

```
cmake ../Step2 -DUSE_MYMATH=OFF
```

重新构建然后运行.

哪个函数结果更好,sqrt还是mysqrt?

Step3:对库添加使用依赖

使用依赖能够让我们更好地控制库或者可执行程序使用的链接和包含.也提供了对CMake内的可及属性的更充分的控制.控制使用依赖的首要命令包括:

- target_compile_definitions
- target_compile_options
- target_include_directories
- target_link_libraries

让我们用现代CMake的方式重构Step2中的使用依赖的部分. 我们首先明确任何链接到MathFunctions的对象都需要包含当前源目录(译者注：指`MathFunctions`目录),除了MathFunctions本身.所以这可以作为一个`INTERFACE`使用依赖.

记住`INTERFACE`指的是那些消费者需要而生产者不需要的东西.
在`MathFunctions/CMakeLists.txt`的结尾加入:

```
target_include_directories(MathFunctions
          INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
          )
```

现在我们已经指定了MathFunctions的使用依赖,我们就可以安全地移除顶级`CMakeLists.txt`文件中的`EXTRA_INCLUDES`变量:

```
if(USE_MYMATH)
  add_subdirectory(MathFunctions)
  list(APPEND EXTRA_LIBS MathFunctions)
endif()
```

以及:

```
target_include_directories(Tutorial PUBLIC
                           "${PROJECT_BINARY_DIR}"
                           )
```

完成后,运行`cmake`或者`cmake-gui`来配置项目并通过在build目录下`cmake --build .`构建运行即可.

安装与测试

现在我们开始给项目添加安装规则和测试支持.

安装规则

安装规则非常简单: 对于MathFunctions,我们希望安装库和头文件,对于应用,我们希望安装可执行文件和配置头.

所以在`MathFunctions/CMakeLists.txt`的结尾我们添加:

```
install(TARGETS MathFunctions DESTINATION lib)
install(FILES MathFunctions.h DESTINATION include)
```

在顶层`CMakeLists.txt`的结尾添加:

```
install(TARGETS Tutorial DESTINATION bin)
install(FILES "${PROJECT_BINARY_DIR}/TutorialConfig.h"
  DESTINATION include
  )
```

这就是建立一个tutorial的基础本地安装所需要的全部内容.

现在我们运行`cmake`或者`cmake-gui`来配置项目并构建.

然后通过在命令行中运行`cmake`的`install`选项来执行安装步骤(3.15引入,早先版本必须使用make install).对于多配制工具,要记得使用`--config`来指定配置.如果使用IDE,直接构建`INSTALL`目标即可.这一步会安装适合的头文件,库和可执行文件:

```
camke --install .
```

CMake变量`CMAKE_INSTALL_PREFIX`用于确定文件安装的根目录.如果使用`cmake --install`命令,安装前驻可以被`--prefix`参数覆写:

```
cmake --install . --prefix "/home/myuser/installdir"
```

浏览安装目录然后验证安装的Tutorial可以运行.

测试支持

接下来让我们测试我们的应用,在顶级`CMakeLists.txt`的结尾,我们可以打开测试功能然后加一些基本测试来验证安装正确.

```
enable_testing()

# does the application run
add_test(NAME Runs COMMAND Tutorial 25)

# does the usage message work?
add_test(NAME Usage COMMAND Tutorial)
set_tests_properties(Usage
  PROPERTIES PASS_REGULAR_EXPRESSION "Usage:.*number"
  )

# define a function to simplify adding tests
function(do_test target arg result)
  add_test(NAME Comp${arg} COMMAND ${target} ${arg})
  set_tests_properties(Comp${arg}
    PROPERTIES PASS_REGULAR_EXPRESSION ${result}
    )
endfunction(do_test)

# do a bunch of result based tests
do_test(Tutorial 4 "4 is 2")
do_test(Tutorial 9 "9 is 3")
do_test(Tutorial 5 "5 is 2.236")
do_test(Tutorial 7 "7 is 2.645")
do_test(Tutorial 25 "25 is 5")
do_test(Tutorial -25 "-25 is [-nan|nan|0]")
do_test(Tutorial 0.0001 "0.0001 is 0.01")
```

第一个测试简单验证了应用运行,没有段错误或其他崩溃发生.并有一个0返回值.这是CTest测试的基本格式.

下一个测试使用了`PASS_REGULAR_EXPRESSION`测试属性来验证测试输出包含特定字符串.用于在参数输入数量不对时打印使用信息.

最后,我们有一个叫做`do_test`的函数来运行应用并验证计算平方根的结果对于给定输出是正确的.对于每次`do_test`的调用,项目中就会被加入一个带有指定的名字,输入和期待结果的测试.

重新构建应用然后进入二进制目录并运行`ctest`可执行文件: `ctest -N`和`ctest -VV`(译者注: 注意是两个V).对于多配置生成器(例如Visual Studio),配置类型必须通过`-C <mode>`来指定.例如,如果想要在Debug模式下运行测试,则需要在构建目录下(而非Debug目录下)执行`ctest -`

`C Debug -VV`.在同样的目录下,使用`-C Release`则可以以Release模式运行.或者也可以在IDE中构建`RUN_TESTS`目标.

增加系统自检

现在我们想项目中增加一些代码,而这些代码依赖的特性可能是目标平台没有的.例如,我们要加入的代码依赖于目标平台是否有 `log` 和 `exp` 函数.当然,对于每个平台而言,这些功能都是有的,但是在本篇教程中,我们假定这些功能不是都存在的.

如果平台有 `log` 和 `exp` 那么我们就在 `mysqrt` 函数里使用.我们首先在顶层 `CMakeLists.txt` 里用 `CheckSymbolExists` 来测试这些函数的可用性.在一些平台,我们会需要连接到 `m` 库如果 `log` 和 `exp` 没有被招到,就需要在 `m` 库里再试试.

```
include(CheckSymbolExists)
check_symbol_exists(log "math.h" HAVE_LOG)
check_symbol_exists(exp "math.h" HAVE_EXP)
if(NOT (HAVE_LOG AND HAVE_EXP))
  unset(HAVE_LOG CACHE)
  unset(HAVE_EXP CACHE)
  set(CMAKE_REQUIRED_LIBRARIES "m")
  check_symbol_exists(log "math.h" HAVE_LOG)
  check_symbol_exists(exp "math.h" HAVE_EXP)
  if(HAVE_LOG AND HAVE_EXP)
    target_link_libraries(MathFunctions PRIVATE m)
  endif()
endif()
```

现在让我们给 `TutorialConfig.h.in` 添加一些定义,这样我们就可以在 `mysqrt.cxx` 里使用了:

```
// does the platform provide exp and log functions?
#cmakedefine HAVE_LOG
#cmakedefine HAVE_EXP
```

如果 `log` 和 `exp` 在系统上可用,那么我们就在 `mysqrt` 里使用它们.
在 `MathFunctions/mysqrt.cxx` 里的 `mysqrt` 里添加下述代码(别忘了返回值之前加 `#endif`):

```
#if defined(HAVE_LOG) && defined(HAVE_EXP)
  double result = exp(log(x) * 0.5);
  std::cout << "Computing sqrt of " << x << " to be " << result
            << " using log and exp" << std::endl;
#else
  double result = x;
```

我们也需要修改 `mysqrt.cxx` 来包含 `cmath` :

```
#include <cmath>
```

运行 `cmake` 或者 `cmake-gui` 来配置项目,然后构建并执行Tutorial.

会注意到我们没有使用 `log` 和 `exp` ,即使我们认为它们应该是可用的.我们应该很容易发现,我们忘记在 `mysqrt.cxx` 中包含 `TutorialConfig.h` 了.

我们也需要更新 `MathFunctions/CMakeLists.txt` ,这样 `mysqrt.cxx` 才能够定位文件:

```
target_include_directories(MathFunctions
          INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
          PRIVATE ${CMAKE_BINARY_DIR}
          )
```

这样更新后,继续并构建项目,然后运行Tutorial.如果 `log` 和 `exp` 仍然没有被使用,打开构建目录下的生成的 `Tutorial.h` 文件,可能他们在当前系统下不可用的.

那个函数目前结果更好呢,sqrt还是mysqrt?

指定编译定义

除了在 `TutorialConfig.h` 中存储 `HAVE_LOG` 和 `HAVE_EXP` 值以外更好的地方么?让我们试试使用 `target_compile_definitions()` .

首先将定义从 `TutorialConfig.h` 中移除,我们不再需要从 `mysqrt.cxx` 中包含 `TutorialConfig.h` 或者在 `MathFunctions/CMakeLists.txt` 中额外包含了.

接下来我们可以把 `HAVE_LOG` 和 `HAVR_EXP` 的检查移动到 `MathFunctions/CMakeLists.txt` 中,然后把这些值设定为 `PRIVATE` 编译定义.

```
include(CheckSymbolExists)
check_symbol_exists(log "math.h" HAVE_LOG)
check_symbol_exists(exp "math.h" HAVE_EXP)
if(NOT (HAVE_LOG AND HAVE_EXP))
  unset(HAVE_LOG CACHE)
  unset(HAVE_EXP CACHE)
  set(CMAKE_REQUIRED_LIBRARIES "m")
  check_symbol_exists(log "math.h" HAVE_LOG)
  check_symbol_exists(exp "math.h" HAVE_EXP)
  if(HAVE_LOG AND HAVE_EXP)
    target_link_libraries(MathFunctions PRIVATE m)
  endif()
endif()
```

```
# add compile definitions
if(HAVE_LOG AND HAVE_EXP)
  target_compile_definitions(MathFunctions
                             PRIVATE "HAVE_LOG" "HAVE_EXP")
endif()
```

这样调整更新后,重新构建项目,再运行Tutorial并确认结果和此前一致.

添加自定义命令和生成文件

假设对于本次教程而言,我们决定我们不再想使用平台的 `log` 和 `exp` 函数,并希望生成一些会在 `mysqrt` 函数里使用到的预计算值表.在本节,我们会建立这个表并作为构建的一步,然后编译到我们的应用中.

首先,我们移除 `MathFunctions/CMakeLists.txt` 中的对 `log` 和 `exp` 的检查.然后移除 `mysqrt.cxx` 中对 `HAVE_LOG` 和 `HAVR_EXP` 的检查.同事,我们也可以移除 `#include <cmath>`

在 `MathFunctions` 子目录下,有一个新文件 `MakeTable.cxx` 用于生成表格.

浏览这个文件可以发现,表格是C++代码生成的并且输出文件名是通过参数传入的.

下一步是在 `MathFunctions/CMakeLists.txt` 中添加合适的命令来构建 `MakeTable` 可执行文件,然后作为构建流程的一部分来运行.需要一些命令来完成这一步.

首先,在 `MathFunctions/CMakeLists.txt` 的开头,添加 `MakeTable` 为可执行文件目标.

```
add_executable(MakeTable MakeTable.cxx)
```

然后我们添加一项定义命令来指定怎么通过运行 `MakeTable` 创建表格.

```
add_custom_command(
  OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/Table.h
  DEPENDS MakeTable
  )
```

接下来我们要让CMake知道 `mysqrt.cxx` 依赖创建的 `Table.h` .这是通过将生成 `Table.h` 添加到 `MathFunctions` 库的源列表.

```
add_library(MathFunctions
            mysqrt.cxx
            ${CMAKE_CURRENT_BINARY_DIR}/Table.h
            )
```

现在让我们用上生成的表,首先,修改 `mysqrt.cxx` 来包含 `Table.h` 然后我们可以重写 `mysqrt` 函数以使用这张表:

```
double mysqrt(double x)
{
  if (x <= 0) {
    return 0;
  }

  // use the table to help find an initial value
  double result = x;
  if (x >= 1 && x < 10) {
    std::cout << "Use the table to help find an initial value " << std::endl;
    result = sqrtTable[static_cast<int>(x)];
  }

  // do ten iterations
  for (int i = 0; i < 10; ++i) {
    if (result <= 0) {
      result = 0.1;
    }
    double delta = x - (result * result);
    result = result + 0.5 * delta / result;
    std::cout << "Computing sqrt of " << x << " to be " << result << std::endl;
  }

  return result;
}
```

运行 `cmake` 或者 `cmake-gui` 来配置项目并构建.

当项目被构建时,首先被构建的是 `MakeTable` ,然后会运行 `MakeTable` 并创建 `Table.h` .最后会编译包含了 `Table.h` 的 `mysqrt.cxx` 来创建MathFunctions库.

运行Tutorial可执行文件然后验证使用了表格.

构建安装器.

下一步,我们假定我们想要发布我们的项目,以便其他人可以使用我们的项目.我们想在多种平台上发布二进制和源代码.这和我们在第四步里所做的有所不同.第四步里我们安装的是从源代码构建的二进制.在本例中,我们会构建支持二进制安装和包管理特性的安装包.为此,我们会使用CPack来生成对应平台的安装器.具体而言,我们需要在我们顶级的 `CMakeLists.txt` 底添加几行:

```
include(InstallRequiredSystemLibraries)
set(CPACK_RESOURCE_FILE_LICENSE "${CMAKE_CURRENT_SOURCE_DIR}/License.txt")
set(CPACK_PACKAGE_VERSION_MAJOR "${Tutorial_VERSION_MAJOR}")
set(CPACK_PACKAGE_VERSION_MINOR "${Tutorial_VERSION_MINOR}")
include(CPack)
```

就这样就可以.我们通过包含InstallRequiredSystemLibraries来开始.这一模块会包含任何项目所需的当前平台的运行库.下一步我们设定一些CPack变量到我们存储项目许可和版本信息的位置.版本信息早先在本篇教程里设定好了. `license.txt` 在这一步被包含在顶级源目录中.

最后,我们包含CPack module.CPack模块会使用这些变量和当前系统的其他变量来配置安装器.

下一步是和正常一样构建项目然后运行cpack可执行文件.从binary目录下运行以下命令以构建二进制发布:

```
cpack
```

为了指定生成器,使用 `-G` 选项,对于多配置构建,使用 `-C` 来指定配置,例如:

```
cpack -G ZIP -C Debug
```

为了构建一个源代码发布,可以使用:

```
cpack --config CPackSourceConfig.cmake
```

或者运行 `make package` ,或者在IDE中右键 `Package` 目录然后 `Build Project` .

运行在二进制文件夹中的安装器,然后运行安装的可执行文件并验证可以运行.

增加对Dashboard的支持

添加对测试提交到仪表盘的支持是很简单的.我们在测试支持一步中已经给我们的项目定义了一系列测试.现在我们只需要运行这些测试并将他们提交到仪表盘上即可.为了包含仪表盘的支持,我们在顶级`CMakeLists.txt`里包含`CTest`模块.

将

```
# enable testing
enable_testing()
```

替换为

```
# enable dashboard scripting
include(CTest)
```

CTest模块会自动调用`enable_testing()`,所以我们就可以从CMake文件里移除这一语句.

我们也需要在顶级目录下(我们指定项目名和提交到面板的目录)建立一个`CTestConfig.cmake`文件.

```
set(CTEST_PROJECT_NAME "CMakeTutorial")
set(CTEST_NIGHTLY_START_TIME "00:00:00 EST")

set(CTEST_DROP_METHOD "http")
set(CTEST_DROP_SITE "my.cdash.org")
set(CTEST_DROP_LOCATION "/submit.php?project=CMakeTutorial")
set(CTEST_DROP_SITE_CDASH TRUE)
```

`ctest`可执行文件会在运行时读取这个文件.可以运行`cmake`或者`cmake-gui`来配置项目但是不构建项目来建立一个简单的面板.切换到二进制树目录下然后运行:

```
ctest [-VV] -C Debug -D Experimental
```

或者在IDE中构建`Experimental`目标.

`ctest`可执行文件会构建和测试项目并提交结果到Kitware的公共面板:https://my.cdash.org/index.php?project=CMakeTutorial.

混合静态和共享

在本节,我们会展示`BUILD_SHARED_LIBS`变量是怎么样用于控制`add_library()`的表现.并且容许控制没有显式类型(`STATIC`, `SHARED` `MODULE`或者`OBJECT`)的库的构建.

我们需要在顶级`CMakeLists.txt`里增加`BUILD_SHARED_LIBS`.我们用`option()`命令来让用户可以选开或者关.

下一步我们要重构MathFunctions来使其成为一个封装了调用`mysqrt`或者`sqrt`的真实的库,而非需要调用代码来实现这个逻辑.这也意味着`USE_MYMATH`不再控制构建MathFunctions而是控制库的行为.

第一步是更新顶级`CMakeLists.txt`的第一节如下:

```
cmake_minimum_required(VERSION 3.10)

# set the project name and version
project(Tutorial VERSION 1.0)

# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# control where the static and shared libraries are built so that on windows
# we don't need to tinker with the path to run the executable
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${PROJECT_BINARY_DIR}")

option(BUILD_SHARED_LIBS "Build using shared libraries" ON)

# configure a header file to pass the version number only
configure_file(TutorialConfig.h.in TutorialConfig.h)

# add the MathFunctions library
add_subdirectory(MathFunctions)

# add the executable
add_executable(Tutorial tutorial.cxx)
target_link_libraries(Tutorial PUBLIC MathFunctions)
```

既然我们已经让MathFunctions总被使用.我们需要更新库的逻辑.因此在`MathFunctions/CMakeLists.txt`里我们需要建立一个SqrtLibrary.这个库会在`USE_MYMATH`启用的条件下构建并安装.现在,因为这只是一篇教程,我们显式地要求SqrtLibrary构建为静态库就可以了.

结果是`MathFunctions/CMakeLists.txt`应该如下:

```
# add the library that runs
add_library(MathFunctions MathFunctions.cxx)

# state that anybody linking to us needs to include the current source dir
# to find MathFunctions.h, while we don't.
target_include_directories(MathFunctions
                           INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
                           )

# should we use our own math functions
option(USE_MYMATH "Use tutorial provided math implementation" ON)
if(USE_MYMATH)

  target_compile_definitions(MathFunctions PRIVATE "USE_MYMATH")

  # first we add the executable that generates the table
  add_executable(MakeTable MakeTable.cxx)

  # add the command to generate the source code
  add_custom_command(
    OUTPUT ${CMAKE_CURRENT_BINARY_DIR}/Table.h
    COMMAND MakeTable ${CMAKE_CURRENT_BINARY_DIR}/Table.h
    DEPENDS MakeTable
    )

  # library that just does sqrt
  add_library(SqrtLibrary STATIC
              mysqrt.cxx
              ${CMAKE_CURRENT_BINARY_DIR}/Table.h
              )

  # state that we depend on our binary dir to find Table.h
  target_include_directories(SqrtLibrary PRIVATE
                             ${CMAKE_CURRENT_BINARY_DIR}
                             )

  target_link_libraries(MathFunctions PRIVATE SqrtLibrary)
endif()

# define the symbol stating we are using the declspec(dllexport) when
# building on windows
target_compile_definitions(MathFunctions PRIVATE "EXPORTING_MYMATH")

# install rules
set(installable_libs MathFunctions)
if(TARGET SqrtLibrary)
  list(APPEND installable_libs SqrtLibrary)
endif()
```

```
install(TARGETS ${installable_libs} DESTINATION lib)
install(FILES MathFunctions.h DESTINATION include)
```

下一步更新 `MathFunctions/mysqrt.cxx` 以使用 `mathfunctions` 和 `detail` 命名空间:

```
#include <iostream>

#include "MathFunctions.h"

// include the generated table
#include "Table.h"

namespace mathfunctions {
namespace detail {
// a hack square root calculation using simple operations
double mysqrt(double x)
{
  if (x <= 0) {
    return 0;
  }

  // use the table to help find an initial value
  double result = x;
  if (x >= 1 && x < 10) {
    std::cout << "Use the table to help find an initial value " << std::endl;
    result = sqrtTable[static_cast<int>(x)];
  }

  // do ten iterations
  for (int i = 0; i < 10; ++i) {
    if (result <= 0) {
      result = 0.1;
    }
    double delta = x - (result * result);
    result = result + 0.5 * delta / result;
    std::cout << "Computing sqrt of " << x << " to be " << result << std::endl;
  }

  return result;
}
}
}
```

我们也需要在 `tutorial.cxx` 中进行调整,所有不再使用 `USE_MYMATH`:

1. 总是包含 `MathFunctions.h`
2. 总是使用 `mathfunctions::sqrt`

3. 不包含cmath

最后更新`MathFunctions/MathFunctions.h`来用dll导出定义:

```
#if defined(_WIN32)
#  if defined(EXPORTING_MYMATH)
#    define DECLSPEC __declspec(dllexport)
#  else
#    define DECLSPEC __declspec(dllimport)
#  endif
#else // non windows
#  define DECLSPEC
#endif

namespace mathfunctions {
double DECLSPEC sqrt(double x);
}
```

这时,如果你构建任何东西,可能会注意到链接失败,因为我们在试图将一个不包含位置无关代码(PIC)（译者注：指生成的代码中无绝对跳转指令，跳转都为相对跳转，详见维基百科）的静态库（译者注：指`SqrtLibrary`）和另一个包含位置无关代码(PIC)的库（译者注：指`MathFunctions`）组合在一起.解决方案是显式地将SqrtLibrary的`POSITION_INDEPENDENT_CODE`目标属性设定为True,不管是什么构建类型.

```
  # state that SqrtLibrary need PIC when the default is shared libraries
  set_target_properties(SqrtLibrary PROPERTIES
                        POSITION_INDEPENDENT_CODE ${BUILD_SHARED_LIBS}
                        )

  target_link_libraries(MathFunctions PRIVATE SqrtLibrary)
```

练习: 我们修改`MathFunctions.h`来使用dll导出定义.使用CMake文档你能否招到一个辅助模块来简化么?

增加生成表达式

生成器表达式是在构建系统生成期间执行以生成对于每一特定配置专有的信息.

生成表达式可以在许多目标属性内容中使用,诸如 `LINK_LIBRARIES` , `INCLUDE_DIRECTORIES` , `COMPILE_DEFINITIONS` 和其他一些属性.生成表达式也可以在使用命令丰富这些属性的时候使用,例如 `target_link_libraries()` , `target_include_directories()` , `target_compile_definitions()` 和其他命令.

生成表达式可用于启用条件链接,在编译时的条件定义,条件包含目录等等.这些条件可能基于构建配置,目标属性,平台信息或者其他可查询信息.

生成表达式有着不同的类型,包括逻辑表达式,信息表达式和输出表达式.

逻辑表达式用于创建条件输出.基本表达式是01表达式, `$<0:...>` 结果是一个空字符串, `$<1:...>` 结果是 `"..."` 的内容.它们也同样是可嵌套的.

生成表达式的普遍用法是依据不同条件添加编译器标志,例如语言级别的或者警告.一个好的模式是把这些信息和允许传播这些信息的 `INTERFACE` 目标关联起来.让我们从构建一个 `INTERFACE` 目标并指定需要的C++标准级别是 `11` 而非 `CMAKE_CXX_STANDARD` 开始.

故下述代码:

```
# specify the C++ standard
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
```

被替换为:

```
add_library(tutorial_compiler_flags INTERFACE)
target_compile_features(tutorial_compiler_flags INTERFACE cxx_std_11)
```

下一步我们添加项目所需的预期的编译器警告标志.因为警告标志基于编译器,我们用 `COMPILE_LAND_AND_ID` 生成器表达式来控制在给定的语言和一系列编译器id下,哪些标志被使用.如下所示:

```
set(gcc_like_cxx "$<COMPILE_LANG_AND_ID:CXX,ARMClang,AppleClang,Clang,GNU>")
set(msvc_cxx "$<COMPILE_LANG_AND_ID:CXX,MSVC>")
target_compile_options(tutorial_compiler_flags INTERFACE
```

```
  "$<${gcc_like_cxx}:$<BUILD_INTERFACE:-Wall;-Wextra;-Wshadow;-Wformat=2;-
Wunused>>"
  "$<${msvc_cxx}:$<BUILD_INTERFACE:-W3>>"
)
```

我们能发现警告信息被封装在`BUILD_INTERFACE`条件内.这样可以让安装我们项目的用户不会继承我们的警告标志.

练习:修改`MathFunctions/CMakeLists.txt`,使得所有的目标都有`target_link_libraries()`来调用`tutorial_compiler_flags`.

增加导出配置

在第四步中,我们为CMake增加了安装库和头文件的功能.在第七步我们增加了打包这些信息以便可以发布给给其他人的能力.

下一步是增加必要信息使得其他的CMake项目可以使用我们的项目,无论是基于构建目录,本地安装还是作为软件包使用.

第一步是更新我们的 `install(TARGETS)` 命令来不仅仅指定 `DESTINATION` 也指定 `EXPORT` . `EXPORT` 关键字生成一个CMake文件,其中含有能够导入安装树中安装命令所列出的所有目标的代码.于是我们可以通过更新 `MathFunctions/CMakeLists.txt` 里的 `install` 命令来显式地导出(`EXPORT`)MathFunctions库:

```
set(installable_libs MathFunctions tutorial_compiler_flags)
if(TARGET SqrtLibrary)
  list(APPEND installable_libs SqrtLibrary)
endif()
install(TARGETS ${installable_libs}
        DESTINATION lib
        EXPORT MathFunctionsTargets)
install(FILES MathFunctions.h DESTINATION include)
```

现在我们已经导出了MathFunctions,我们也需要显式地安装生成的 `MathFunctionsTargets.cmake` 文件.这是通过在顶级 `CMakeLists.txt` 的底部添加:

```
install(EXPORT MathFunctionsTargets
  FILE MathFunctionsTargets.cmake
  DESTINATION lib/cmake/MathFunctions
)
```

这时应该试着运行CMake.如果设置都正确的话,CMake应该会报错如下:

```
Target "MathFunctions" INTERFACE_INCLUDE_DIRECTORIES property contains
path:

  "/Users/robert/Documents/CMakeClass/Tutorial/Step11/MathFunctions"

which is prefixed in the source directory.
```

CMake报错描述的是在生成导出信息期间,会导出内在绑定到当前设备的路径,在其他设备上可能无效.解决方案是更新MathFunctions的 `target_include_directories()` ,以明确在从构建

目录使用和从安装包使用时需要不同的`INTERFACE`位置.这意味着MathFunctions的`target_include_directories()`调用应改为如下:

```
target_include_directories(MathFunctions
                           INTERFACE
                            $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}>
                            $<INSTALL_INTERFACE:include>
                           )
```

这一项更新后,我们就可以重新运行CMake并验证不再有警告了.

到这里,我们已经将CMake配置打包好了所需的目标信息.但是我们也仍然需要生成`MathFunctionsConfig.cmake`以使得CMake的`find_package()`能够找到我们的项目.所以我们继续在顶级项目下建立名为`Config.cmake.in`的文件并写入下述代码:

```
@PACKAGE_INIT@

include ( "${CMAKE_CURRENT_LIST_DIR}/MathFunctionsTargets.cmake" )
```

然后为了正确配置安装这个文件.将下述内容写入顶级`CMakeLists.txt`的结尾:

```
install(EXPORT MathFunctionsTargets
  FILE MathFunctionsTargets.cmake
  DESTINATION lib/cmake/MathFunctions
)

include(CMakePackageConfigHelpers)
# generate the config file that is includes the exports
configure_package_config_file(${CMAKE_CURRENT_SOURCE_DIR}/Config.cmake.in
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake"
  INSTALL_DESTINATION "lib/cmake/example"
  NO_SET_AND_CHECK_MACRO
  NO_CHECK_REQUIRED_COMPONENTS_MACRO
  )
# generate the version file for the config file
write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfigVersion.cmake"
  VERSION "${Tutorial_VERSION_MAJOR}.${Tutorial_VERSION_MINOR}"
  COMPATIBILITY AnyNewerVersion
)

# install the configuration file
install(FILES
  ${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsConfig.cmake
```

```
  DESTINATION lib/cmake/MathFunctions
  )


```

这里,我们已经生成了一个可重定位的CMake配置并是可以在项目安装或者打包后使用的.如果我们想让我们的项目也能够在构建目录使用.我们只需要在顶级 `CMakeLists.txt` 的结尾添加下述内容:

```
export(EXPORT MathFunctionsTargets
  FILE "${CMAKE_CURRENT_BINARY_DIR}/MathFunctionsTargets.cmake"
)
```

有这一导出后,我们现在生成一个 `Targets.cmake` ,使得在构建目录里配置好的 `MathFunctionsConfig.cmake` 可以被其他项目使用而无需安装.

打包Debug和Release

注意: 这个例子对于单配置生成器有效.而对于多配置生成器无效(例如Visual Studio)

默认情况下,CMake模型是一个构建目录只包含一个配置,也就是Debug,Release,MinSizeRel或者RelWithDebInfo等的构建目录.但是是可以通过CPack打包多个构建目录以及构建一个包含同一项目多个配置的包.

首先,我们想要确认debug和release的构建使用不同的可执行和库名字.让我们使用`d`作为调试的可执行和库的后缀.

在顶级`CMakeLists.txt`文件的开头设定`CMAKE_DEBUG_POSTFIX`:

```
set(CMAKE_DEBUG_POSTFIX d)

add_library(tutorial_compiler_flags INTERFACE)
```

给tutorial的可执行文件添加`DEBUG_POSTFIX`属性:

```
add_executable(Tutorial tutorial.cxx)
set_target_properties(Tutorial PROPERTIES DEBUG_POSTFIX ${CMAKE_DEBUG_POSTFIX})

target_link_libraries(Tutorial PUBLIC MathFunctions)
```

让我们也给MathFunctions库添加版本号.在`MathFunctions/CMakeLists.txt`设定`VERSION`和`SOVERSION`属性:

```
set_property(TARGET MathFunctions PROPERTY VERSION "1.0.0")
set_property(TARGET MathFunctions PROPERTY SOVERSION "1")
```

从`Step12`目录,创建`debug`和`release`子目录,结构如下:

```
- Step12
   - debug
   - release
```

现在我们需要设定debug和release构建.我们可以用`CMAKE_BUILD_TYPE`来设定配置类型:

```
cd debug
cmake -DCMAKE_BUILD_TYPE=Debug ..
cmake --build .
cd ../release
cmake -DCMAKE_BUILD_TYPE=Release ..
cmake --build .
```

现在debug和release的构建都完成了.我们可以用自定义配置文件来将两个构建都打包到同一个发布包中.在 `Step12` 目录里,创建一个名为 `MultiConfig.cmake` 的文件.在文件中,首先包含通过 `cmake` 创建的默认配置文件.

接下来,用 `CPACK_INSTALL_CMAKE_PROJECTS` 变量来指定安装哪个项目.这时,我们想安装debug和release两个版本:

```
include("release/CPackConfig.cmake")

set(CPACK_INSTALL_CMAKE_PROJECTS
    "debug;Tutorial;ALL;/"
    "release;Tutorial;ALL;/"
    )
```

从 `Step12` 目录下,运行 `cpack` 命令通过 `config` 选项来指定我们自定义的配置文件:

```
cpack --config MultiCPackConfig.cmake
```