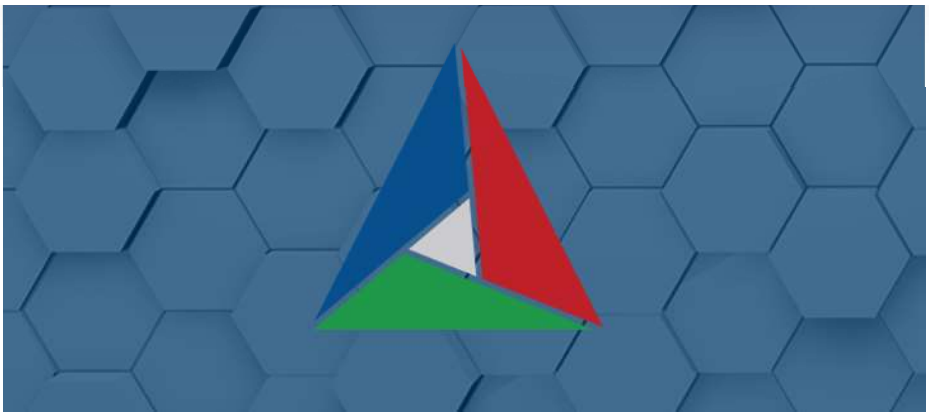

CMake 良心教程，教你从入门到入魂

程序员阿德 关注他

麦茬茬、雪地冰激凌、拾柏等人赞同

收起

环境配置
构建最小项目
编译和运行
建与内部构建
优化 CMakeLists.txt ...
PROJECT_NAME
本号和配置头文件
译时间戳
++ 标准
添加库
将库设置为可选项
添加库的使用要求
build 目录介绍

在完成 轻松打造 Qt Creator 开发环境（Qt5.14+CMake3.22+OpenCV4.5） 后，续集来了。

这篇文章主要介绍 CMake 的使用，看完这篇文章后，CMake 的绝大多数使用方法你都能掌握。本篇文章采用循序渐进的方法带你一步步逐渐进阶 CMake，通过多个示例，告诉你如何使用 CMake 解决常见的构建系统问题。

> 各位爱学习的朋友，收藏的同时点个赞行不，点赞的人多，这篇文章才能帮助到更多的人。

step0：环境配置

开始前说明一下，我的环境是 Windows10 + CMake + MinGW，MinGW 就是 GCC 的 Windows 移植版本。

环境安装我就不介绍了，不是这篇文章的重点，知乎有很多相关教程，这里提供相关工具下载链接：

- 构建工具：Download | CMake
- 编译工具：Downloads - MinGW-w64

需要注意的是，CMake 和 MinGW 安装好后，要手动添加到环境变量。

step 1：构建最小项目

最基本的项目是将一个源代码文件生成可执行文件。对于这么简单的项目，只需要一个三行的 CMakeLists.txt 文件即可，这是本篇教程的起点。在 step1 目录中创建一个 CMakeLists.txt 文件，如下所示：

```
cmake_minimum_required(VERSION 3.15)

# set the project name
project(Tutorial)

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

`cmake_minimum_required` 指定使用 CMake 的最低版本号， `project` 指定项目名称，`add_executable` 用来生成可执行文件，需要指定生成可执行文件的名称和相关源文件。

注意，此示例在 CMakeLi 令。tutorial.cpp 文件在 ;

```cpp
// tutorial.cpp

#include <cmath>
#include <cstdlib>
#include <iostream>
#include <string>

int main(int argc, char* argv[])
{
    if (argc < 2) {
        std::cout << "Usage: " << argv[0] << " number" << std::endl;
        return 1;
    }

    // convert input to double
    const double inputValue = atof(argv[1]);

    // calculate square root
    const double outputValue = sqrt(inputValue);
    std::cout << "The square root of " << inputValue
              << " is " << outputValue
              << std::endl;
    return 0;
}
```

构建、编译和运行

现在就可以构建和运行我们的项目了，就是先运行 `cmake` 命令来构建项目，然后使用你选择的编译工具进行编译。

先从命令行进入到 step1 目录，并创建一个构建目录 build，接下来，进入 build 目录并运行 CMake 来配置项目，并生成构建系统：

```
mkdir build
cd build
cmake -G"MinGW Makefiles" ..
```

构建系统是需要指定 CMakeLists.txt 所在路径，此时在 build 目录下，所以用 `..` 表示 CMakeLists.txt 在上一级目录。

Windows 下，CMake 默认使用微软的 MSVC 作为编译器，我想使用 MinGW 编译器，可以通过 `-G` 参数来进行指定，只有第一次构建项目时需要指定。

此时在 build 目录下会生成 Makefile 文件，然后调用编译器来实际编译和链接项目：

```
cmake --build .
```

`--build` 指定编译生成的文件存放目录，其中就包括可执行文件， `.` 表示存放到当前目录，

在 build 目录下生成了一个 Tutorial.exe 可执行文件，试着执行它：

```
> Tutorial.exe 5
The square root of 5 is 2.23607
```

该程序计算 5 的平方根，从输出结果看已经得到了正确的结果。

此时目录结构为：

```
step1/
    build/
```

```
CMakeLists.txt
tutorial.cpp
```

外部构建与内部构建

这里创建了一个 build 目录存放编译产物，可以避免编译产物与代码文件混在一起，这种叫做外部构建。

还有一种内部构建，即直接在项目根目录下进行构建系统与编译，这时构建和编译命令就更改为：

```
cmake -G"MinGW Makefiles" .
cmake --build .
```

内部构建会使得项目文件很混乱，一般直接用外部构建即可。

step 2：优化 CMakeLists.txt 文件

set 与 PROJECT_NAME

这是之前见过的 CMakeLists.txt 文件：

```
cmake_minimum_required(VERSION 3.15)

# set the project name
project(Tutorial)

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

指定了项目名后，后面可能会有多个地方用到这个项目名，如果更改了这个名字，就要改多个地方，比较麻烦，那么可以使用 `PROJECT_NAME` 来表示项目名。

```
add_executable(${PROJECT_NAME} tutorial.cpp)
```

生成可执行文件需要指定相关的源文件，如果有多个，那么就用空格隔开，比如：

```
add_executable(${PROJECT_NAME} a.cpp b.cpp c.cpp)
```

我们也可以用一个变量来表示这多个源文件：

```
set(SRC_LIST a.cpp b.cpp c.cpp)
add_executable(${PROJECT_NAME} ${SRC_LIST})
```

`set` 命令指定 SRC_LIST 变量来表示多个源文件，用 `${var_name}` 获取变量的值。

于是原来的 CMakeLists.txt 文件就可以变成如下所示：

```
cmake_minimum_required(VERSION 3.15)

# set the project name
project(Tutorial)

SET(SRC_LIST tutorial.cpp)

# add the executable
add_executable(${PROJECT_NAME} ${SRC_LIST})
```

这样看起来就很简洁。

添加版本号和配置头文件

我们可以在 CMakeLists.txt 为可执行文件和项目提供一个版本号。首先，修改 CMakeLists.txt 文件，使用 project 命令设置项目名称和版本号。

```
cmake_minimum_required(VERSION 3.15)

# set the project name and version
project(Tutorial VERSION 1.0.2)

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

然后，配置头文件将版本号传递给源代码：

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

由于 TutorialConfig.h 文件这里被设置为自动写入 build 目录，因此需要将该目录添加到搜索头文件的路径列表中，也可以修改为写到其它目录。

将以下行添加到 CMakeLists.txt 文件的末尾：

```
target_include_directories(${PROJECT_NAME} PUBLIC
                           ${PROJECT_BINARY_DIR}
                           )
```

`PROJECT_BINARY_DIR` 表示当前工程的二进制路径，即编译产物会存放到该路径，此时 `PROJECT_BINARY_DIR` 就是 build 所在路径。

然后手动创建 TutorialConfig.h.in 文件，包含以下内容：

```
// the configured options and settings for Tutorial
#define Tutorial_VERSION_MAJOR @PROJECT_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @PROJECT_VERSION_MINOR@
#define Tutorial_VERSION_PATCH @PROJECT_VERSION_PATCH@
```

当使用 CMake 构建项目后，会在 build 中生成一个 TutorialConfig.h 文件，内容如下：

```
// the configured options and settings for Tutorial
#define Tutorial_VERSION_MAJOR 1
#define Tutorial_VERSION_MINOR 0
#define Tutorial_VERSION_PATCH 2
```

下一步在 tutorial.cpp 包含头文件 TutorialConfig.h，最后通过以下代码打印出可执行文件的名称和版本号。

```
    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;
    }
```

添加编译时间戳

有时候我们需要知道编译时的时间戳，并在程序运行时打印出来。

那就需要在 CMakeLists.txt 中添加如下这句：

```
string(TIMESTAMP COMP
```

这表示将时间戳已指定格式保存到 COMPILE_TIME 变量中。

然后修改上面的 TutorialConfig.h.in 文件：

```
// the configured options and settings for Tutorial
#define Tutorial_VERSION_MAJOR @PROJECT_VERSION_MAJOR@
#define Tutorial_VERSION_MINOR @PROJECT_VERSION_MINOR@
#define Tutorial_VERSION_PATCH @PROJECT_VERSION_PATCH@

#define TIMESTAMP @COMPILE_TIME@
```

在构建项目后，TutorialConfig.h 文件就会自动增加一句：

```
#define TIMESTAMP 20230220-203532
```

这样就可以在源码中打印出 TIMESTAMP 的值了。

指定 C++ 标准

接下来将 step1/tutorial.cpp 源码中的 atof 替换为 `std::stod` ，这是 C++11 的特性，并删除 `#include<cstdlib>` 。

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

在 CMake 中支持特定 C++标准的最简单方法是使用 `CMAKE_CXX_STANDARD` 标准变量。在 CMakeLists.txt 中设置 `CMAKE_CXX_STANDARD` 为11， `CMAKE_CXX_STANDARD_REQUIRED` 设置为 True。确保在 `add_executable` 命令之前添加 `CMAKE_CXX_STANDARD_REQUIRED` 命令。

```
cmake_minimum_required(VERSION 3.15)

# set the project name and version
project(${PROJECT_NAME} VERSION 1.0)

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

需要注意的是，如果你的gcc编译器版本够高，也可以不用指定 C++ 版本为 11。从 GCC 6.1 开始，当不指定任何版本 C++ 标准时，默认版本是 C++ 14，如果你想用 C++17 的语言，还是需要指定的。

修改完成后，需要对代码进行重新编译 `cmake --build .` ，此时可以不用进行项目构建。

此时目录结构为：

```
step2/
    build/
    CMakeLists.txt
    tutorial.cpp
    TutorialConfig.h.in
```

step 3：添加库

现在我们将向项目中添加一个库，这个库包含计算数字平方根的实现，可执行文件使用这个库，而不是编译器提供的标准平方根函数。

我们把库放在名为 MathFunctions 的子目录中。此目录包含头文件 MathFunctions.h 和源文件 mysqrt.cpp。源文件有一个名为 mysqrt 的函数，它提供了与编译器的 sqrt 函数类似的功能，MathFunctions.h 则是该

在 MathFunctions 目录下创建一个 CMakeLists.txt 文件，并添加以下一行：

```
# MathFunctions/CMakeLists.txt
add_library(MathFunctions mysqrt.cpp)
```

表示添加一个叫 MathFunctions 的库文件。

CMake 中的 target 有可执行文件和库文件，分别使用 `add_executable` 和 `add_library` 命令生成，除了指定生成的可执行文件名/库文件名，还需要指定相关的源文件。

此时文件结构为：

```
step3/
    build/
    MathFunctions/
        CMakeLists.txt
        MathFunctions.h
        mysqrt.cpp
    CMakeLists.txt
    tutorial.cpp
    TutorialConfig.h.in
```

为了使用 MathFunctions 这个库，我们将在顶级 CMakeLists.txt 文件中添加一个 `add_subdirectory(MathFunctions)` 命令指定库所在子目录，该子目录下应包含 CMakeLists.txt 文件和代码文件。

可执行文件要使用库文件，需要能够找到库文件和对应的头文件，可以分别通过 `target_link_libraries` 和 `target_include_directories` 来指定。

使用 `target_link_libraries` 将新的库文件添加到可执行文件中，使用 `target_include_directories` 将 MathFunctions 添加为头文件目录，添加到 Tutorial 目标上，以便 mysqrt.h 可以被找到。

顶级 CMakeLists.txt 的最后几行如下所示：

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

# add the executable
add_executable(${PROJECT_NAME} tutorial.cpp)

target_link_libraries(${PROJECT_NAME} PUBLIC MathFunctions)

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

MathFunctions 库就算添加完成了，接下来就是在主函数使用该库中的函数，先在 tutorial.cpp 文件中添加头文件：

```
#include "MathFunctions.h"
```

然后使用 `mysqrt` 函数即可：

```
const double outputValue = mysqrt(inputValue);
```

step 4：将库设置为i

现在将 MathFunctions 库设为可选的，虽然对于本教程来说，没有必要这样做，但对于较大的项目来说，这种情况很常见。

第一步是向顶级 CMakeLists.txt 文件添加一个选项。

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

`option` 表示提供用户可以选择的选项。命令格式为： `option(<variable> "description [initial value])` 。

`USE_MYMATH` 这个选项缺省值为 ON，用户可以更改这个值。此设置将存储在缓存中，以便用户不需要在每次构建项目时设置该值。

下一个更改是使 MathFunctions 库的构建和链接成为条件。于是创建一个 if 语句，该语句检查选项 `USE_MYMATH` 的值。

```
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(${PROJECT_NAME} tutorial.cpp)

target_link_libraries(${PROJECT_NAME} PUBLIC ${EXTRA_LIBS})

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

在 if 块中，有 `add_subdirectory` 命令和 `list` 命令， `APPEND` 表示将元素 `MathFunctions` 追加到列表 `EXTRA_LIBS` 中，将元素 `${PROJECT_SOURCE_DIR}/MathFunctions` 追加到列表 `EXTRA_INCLUDES` 中。 `EXTRA_LIBS` 存储 MathFunctions 库， `EXTRA_INCLUDES` 存储 MathFunctions 头文件。

变量 `EXTRA_LIBS` 用来保存需要链接到可执行程序的可选库，变量 `EXTRA_INCLUDES` 用来保存可选的头文件搜索路径。这是处理可选组件的经典方法，我将在下一步介绍现代方法。

接下来对源代码的进行修改。首先，在 tutorial.cpp 中包含 MathFunctions.h 头文件：

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

然后，还在 tutorial.cpp 中，使用 `USE_MYMATH` 选择使用哪个平方根函数：

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

因为源代码使用了 `USE_MYMATH` 宏，可以用下面的行添加到 tutorialconfig.h.in 文档中：

```
// TutorialConfig.h.in
#cmakedefine USE_MYMATH
```

现在使用 cmake 命令构建项目，并运行生成的 Tutorial 可执行文件。

```
build> cmake -G"MinGW Makefiles" ..
build> cmake --build .
build> Tutorial.exe 8
Computing sqrt of 8 to be 4.5
Computing sqrt of 8 to be 3.13889
Computing sqrt of 8 to be 2.84378
Computing sqrt of 8 to be 2.82847
Computing sqrt of 8 to be 2.82843
Computing sqrt of 8 to be 2.82843
Computing sqrt of 8 to be 2.82843
Computing sqrt of 8 to be 2.82843
Computing sqrt of 8 to be 2.82843
Computing sqrt of 8 to be 2.82843
The square root of 8 is 2.82843
```

默认调用 `mysqrt` 函数，也可以在构建项目时指定 `USE_MYMATH` 的值为 OFF：

```
> cmake -DUSE_MYMATH=OFF ..
> cmake --build .
```

此时会调用自带的 `sqrt` 函数。

step 5：添加库的使用要求

使用要求会对库或可执行程序的链接、头文件包含命令行提供了更好的控制，也使 CMake 中目标的传递目标属性更加可控。利用使用要求的主要命令是：

- target_compile_definitions()
- target_compile_options()
- target_include_directories()
- target_link_libraries()

现在重构一下 step4 中的代码，使用更加现代的 CMake 方法来包含 MathFunctions 库的头文件。

首先声明，链接 MathFunctions 库的任何可执行文件/库文件都需要包含 MathFunctions 目录作为头文件路径，而 MathFunctions 本身不需要包含，这被称为 `INTERFACE` 使用要求。

`INTERFACE` 是指消费者需要、但生产者不需要的那些东西。在 `MathFunctions/CMakeLists.txt` 最后添加：

```
# MathFunctions/CMakeLists.txt
target_include_directories(MathFunctions
          INTERFACE ${CMAKE_CURRENT_SOURCE_DIR}
          )
```

`CMAKE_CURRENT_SOURCE_DIR` 表示 MathFunctions 库所在目录。

现在我们已经为 MathFunctions 指定了使用要求 `INTERFACE`，那么可以从顶级 CMakeLists.txt 中删除 `EXTRA_INCLUDES` 变量的相关使用：

```
if(USE_MYMATH)
  add_subdirectory(MathFunctions)
  list(APPEND EXTRA_LIBS MathFunctions)
  list(APPEND EXTRA_INCLUDES ${PROJECT_SOURCE_DIR}/MathFunctions)   # 删除此行
endif()

...

# add the binary tree
```

```
# so that we will find TutorialConfig.h
target_include_directories(${PROJECT_NAME} PUBLIC
                           ${PROJECT_BINARY_DIR}
                           ${EXTRA_INCLUDES}   # 删除此行
                           )
```

现在只要是链接了 MathFunctions 库，就会自动包含 MathFunctions 所在目录的头文件，简洁而优雅。

这里补充两个知识点：

1、使用要求除了 `INTERFACE` ，还有 `PRIVATE` 和 `PUBLIC` 。 `INTERFACE` 表示消费者需要生产者不需要， `PRIVATE` 表示消费者不需要生产者需要， `PUBLIC` 表示消费者和生产者都需要。

2、这里使用 `add_library` 命令生成的 MathFunctions 库其实是静态链接库。动态库和静态库的区别是：静态库在**链接阶段**会被链接到最终目标中（比如可执行程序），缺点是同一个静态库如果被不同的程序引用，那么内存中会存在这个静态库函数的多份拷贝。动态库在链接阶段不会被拷贝最终目标中，程序在**运行阶段**才会加载这个动态库。所以多个程序就算引用了同一个动态库，内存中也只存在一份动态库函数的拷贝。

step 6：build 目录介绍

在文本中，我都是创建了一个 build 用来存放 cmake 构建和编译的产物，这里简单说下里面有些什么东西。

```
build/
    CMakeCache.txt
    CMakeFiles/
    cmake_install.cmake
    Makefile
    Tutorial.exe
    TutorialConfig.h
    MathFunctions/
```

其中 `Makefile` 是 cmake 根据顶级 CMakeLists.txt 生成的构建文件，通过该文件可以对整个项目进行编译。

`Tutorial.exe` 就是生成的可执行文件，通过该文件运行程序。

`TutorialConfig.h` 是用于配置信息的头文件，是 cmake 根据 `TutorialConfig.h.in` 文件自动生成的。

还有个 MathFunctions 文件夹：

```
MathFunctions/
    CMakeFiles/
    cmake_install.cmake
    Makefile
    libMathFunctions.a
```

其中 `Makefile` 是 cmake 根据 MathFunctions 目录下的 CMakeLists.txt 生成的构建文件。

`libMathFunctions.a` 则是 MathFunctions 静态链接库，可执行文件会通过这个库调用 `mysqrt` 函数。

原文链接（可以领取源码）：

CMake 很难，因为你没看过这篇文章
mp.weixin.qq.com/s?__biz=Mzg4ODAxM...