Makefile & CMake

目录

前言

想象一下我们有如下C++程序 `hello.cpp` :

```cpp
#include <iostream>
int main()
{
    std::cout << "hello world!" << std::endl;
}
```

我们需要在终端输入以下指令:

```
$ g++ hello.cpp -o hello
```

这时我们就可以生成可执行文件 `hello` 。但在实际应用场景中，我们可能会面临如下问题:

- 项目中的 `.h` 文件和 `.cpp` 文件十分繁多。
- 各 `.h` 文件、 `.cpp` 文件的依赖关系十分复杂。
- 多文件可能会出现重复编译的情况，拖慢编译速度。
- ...

为了解决这些问题， `makefile` 和 `CMake` 应运而生。

Makefile

在linux(Ubuntu)平台上， `make` 工具可以通过以下方式安装:

```
$ sudo apt-get install make
```

makefile文件描述了C/C++工程的编译规则，可以用来指明源文件的编译顺序、依赖关系、是否需要重新编译等，自动化编译C/C++项目（实际上也不止局限于C/C++项目）。

我们可以考虑以下实例:

```
.
├── invsqrt.cpp
├── invsqrt.h
├── main.cpp
└── makefile
```

makefile如下：

```
CXXFLAGS = -std=c++17 -O2

main: main.o invsqrt.o
    $(CXX) $(CXXFLAGS) -o $@ $^

main.o: main.cpp invsqrt.h
    $(CXX) $(CXXFLAGS) -o $@ -c $<

invsqrt.o: invsqrt.cpp invsqrt.h
    $(CXX) $(CXXFLAGS) -o $@ -c $<

.PHONY: clean
clean:
    rm main.o invsqrt.o main
```

此处，我们不需要理解每一段代码的具体含义——由于makefile文件的可读性较差，在日后的开发工作中，我们并不需要直接编写makefile（之后我们可以看到，我们可以直接通过CMake工具生成makefile）。

当然，目前仍然有许多开源项目使用makefile构建程序，因此了解如何使用makefile仍然是十分有必要的。大部分开源项目中的makefile支持以下指令：

- 在makefile的同目录下输入 `make`，就可以按照makefile所指定的编译规则自动编译整个工程。
- 在makefile的同目录下输入 `make clean`，可以删除编译生成的中间文件（如 `.o` 文件等）和可执行文件。
- 在makefile的同目录下输入 `make install`（一般需要root权限），可以安装编译好的可执行文件（默认路径为 `/usr/local/bin`，安装好后可以在命令行中直接调用）、库（默认路径为 `/usr/local/lib`，安装好后可以直接链接）、头文件（默认路径为 `/usr/local/include`，安装好后可以直接使用 `#include <xxx.h>` 引用）。

CMake

makefile存在以下问题：

- 代码可读性极差，难以维护。
- 语法复杂。
- 跨平台性差。比如linux平台下的makefile在windows下可能无法工作，因为linux的删除指令是 `rm`，windows下的删除指令是 `del`。
- ...

因此，在目前的C++工程中，我们多使用CMake来管理项目。CMake是一种跨平台的编译工具，可以用较为简洁易读的语法描述C++项目的编译、链接、安装过程等，在现代C++项目上得到了广泛应用。

Linux

在linux(Ubuntu)平台上，`cmake` 工具可以通过以下方式安装：

```
$ sudo apt-get install cmake
```

在linux平台上，cmake工具的使用一般分为两步 1) 使用 `CMakeLists.txt` 生成 `makefile`。 2) 使用 `makefile` 自动化编译项目。

1 第一个CMake项目⭐

CMake的项目文件叫做 `CMakeLists.txt` 。其放置位置如下图所示：

```
├── CMakeLists.txt
└── main.cpp
```

该项目的 `CMakeLists.txt` 中需要添加以下内容：

```
cmake_minimum_required(VERSION 3.5)
project (hello_world)
add_executable(hello_world main.cpp)
```

> **语法总结1**
>
> - `cmake_minimum_required(VERSION 3.5)` CMake需要的最小版本。CMake的版本可以在命令行中输入 `cmake --version` 获取，一般无强制要求。
> - `project(<project_name>)` 指定工程名称。
> - `add_executable(<executable_name> <cppfile_name>)` 生成可执行文件。

操作方法如下：

1. 输入 `cmake CMakeLists.txt` ，目录下将会生成一个 `Makefile` 文件。
2. 输入 `make` ，即可将源代码编译生成可执行文件。此处将会在与 `CMakeLists.txt` 相同目录的位置生成一个可执行文件 `hello_word` ，输入 `./hello_word` 即可运行该可执行文件。
3. 此外，输入 `make help` ，你也可以查看使用当前的 `Makefile` 所能执行的所有指令，例如 `make clean` （清楚生成的可执行文件和中间文件）。

2 多文件⭐

在平时的程设小作业中，我们习惯将所有的代码都写在一个 `.cpp` 文件中。但在实际工程中，为了方便代码复用和运行维护，通常将所有的文件划分为头文件(`.h`)，模块文件(`.cpp`)和主程序文件(`.cpp`)。

在本节中，我们将在头文件中声明一个计算平方根倒数的函数，在模块文件中实现其主体，然后在主函数中调用它。项目结构如下：

```
.
├── CMakeLists.txt
├── include
│   └── invsqrt.h
└── src
    ├── invsqrt.cpp
    └── main.cpp
```

> tips: 在C++工程中，我们通常在 `include/` 目录下放置头文件，在 `src/` 目录下放置源文件。

该项目的 `CMakeLists.txt` 中需要添加以下内容：

```
# build part
cmake_minimum_required(VERSION 3.5)
project(invsqrt)
set(SOURCES src/invsqrt.cpp src/main.cpp)
add_executable(invsqrt ${SOURCES})
target_include_directories(invsqrt PUBLIC ${PROJECT_SOURCE_DIR}/include)

# debug part
message("CMAKE_SOURCE_DIR: ${CMAKE_SOURCE_DIR}")
message("PROJECT_SOURCE_DIR: ${PROJECT_SOURCE_DIR}")

message("SOURCES: ${SOURCES}")
```

> **语法总结2**
>
> - `set(<variable> <value>)` 设置变量
> - `target_include_directories(<project_name> <INTERFACE|PUBLIC|PRIVATE> <headfile_directory>)` 指定所要包含的头文件。
> - `message("your message")` 在终端打印信息。

这里需要特别说明一下CMake中的变量使用。CMake中的变量分为两种：

- 显式变量：使用 `set` 指令定义的变量。
- 隐式变量：通过其它指令隐式生成的变量。如该项目中会隐式生成 `PROJECT_SOURCE_DIR` 变量，默认为 `CMakeLists.txt` 所在的文件夹。

CMake中有丰富的变量，用于定义工程目录、编译选项等，此处不做过多展开。想要了解更多，可以参考文末列出的参考文档。

3 静态库和动态库 ⭐

有些时候，出于方便复用、防止源码泄露等原因，我们需要将代码封装为静态库和动态库。CMake同样提供了生成静态库和动态库的功能。

3.1 静态库

在此处，我们将上一小节中计算平方根倒数的程序封装为静态库。项目结构如下：

```
.
├── CMakeLists.txt
├── include
│   └── invsqrt.h
└── src
    ├── invsqrt.cpp
    └── main.cpp
```

该项目的 `CMakeLists.txt` 中需要添加以下内容：

```
cmake_minimum_required(VERSION 3.5)
project(invsqrt)

# create static library
add_library(invsqrt_static STATIC src/invsqrt.cpp)
target_include_directories(invsqrt_static PUBLIC ${PROJECT_SOURCE_DIR}/include)

# create executable
add_executable(invsqrt src/main.cpp)
target_link_libraries(invsqrt PRIVATE invsqrt_static)
```

> **语法总结3**
>
> - `add_library(<library_name> STATIC <cppfile_name>)` 生成静态库
> - `target_link_libraries(<executable> <INTERFACE|PUBLIC|PRIVATE> <library_name>)` 指定所要链接的库。

此处我们使用一种更为优雅的生成方式——我们期望将生成的静态库、可执行文件输出到 `build` 文件夹里，而不是和主项目混杂在一起。为此我们需要输入以下指令：

```
$ mkdir build
$ cd build
$ cmake .. # 使用的是上一层目录的CMakeLists.txt，因此需要输入'..'
$ make
```

我们将会在 `build/` 目录下看到静态库 `libinvsqrt_static.a` 和可执行文件 `invsqrt` 。

3.2 动态库

项目目录结构同静态库一节。

该项目的 `CMakeLists.txt` 中需要添加以下内容：

```
cmake_minimum_required(VERSION 3.5)
project(invsqrt)

# create shared library
add_library(invsqrt_shared SHARED src/invsqrt.cpp)
target_include_directories(invsqrt_shared PUBLIC ${PROJECT_SOURCE_DIR}/include)

# create executable
add_executable(invsqrt src/main.cpp)
target_link_libraries(invsqrt PRIVATE invsqrt_shared)
```

> **语法总结4**
>
> - `add_library(<library_name> SHARED <cppfile_name>)` 生成动态库

同样按照上小节的方法生成项目。我们将会在 `build/` 目录下看到动态库 `libinvsqrt_shared.so` 和可执行文件 `invsqrt` 。

4 使用第三方库

在实际的C++工程中，我们可能需要链接一些开源的第三方库。CMake也提供了相关的配置方式。我们以谷歌开发的单元测试框架 `googletest` 为例：

> googletest的安装方法：
>
> ```
> $ git clone https://github.com/google/googletest.git
> # or git clone git@github.com:google/googletest.git
> $ cd googletest
> $ mkdir build
> $ cd build
> $ cmake ../
> $ make -j all
> $ make install
> # or sudo make install
> ```

项目结构如下：

```
.
├── CMakeLists.txt
├── include
│   └── mysqrt.h
└── src
    ├── mysqrt.cpp
    └── main.cpp
cmake_minimum_required(VERSION 2.6)

project(cmake_with_gtest)

set(SOURCES src/mysqrt.cpp src/main.cpp)

find_package(GTest)
message("GTEST_LIBRARIES: ${GTEST_LIBRARIES}")
message("GTEST_INCLUDE_DIRS: ${GTEST_INCLUDE_DIRS}")
```

```
include_directories(${GTEST_INCLUDE_DIRS} ${PROJECT_SOURCE_DIR}/include)

add_executable(cmake_with_gtest ${SOURCES})
target_link_libraries(cmake_with_gtest ${GTEST_LIBRARIES} pthread)
```

> **语法总结5**
>
> - `find_package(<package_name>)` 查询第三方库的位置。若查找成功，则初始化变量 `<package_name>_INCLUDE_DIR`（第三方库的头文件目录）以及 `<package_name>_LIBRARIES`（第三方库的静态/动态库目录）。

CMake支持的所有第三方库可以在https://cmake.org/cmake/help/latest/manual/cmake-modules.7.html中找到。

windows

我们也可以在windows上使用CMake管理C++程序：

1. 在创建新项目时选择“CMake项目”。
2. 创建新项目后，我们可以在文件夹中看到 `.cpp` 、 `.h` 和 `CMakeLists.txt` 模板。我们需要添加自己的头文件、源文件，并修改 `CMakeLists.txt` 。

```
# CMakeList.txt: CMakeProject1 的 CMake 项目，在此处包括源代码并定义
# 项目特定的逻辑。
#
cmake_minimum_required (VERSION 3.8)

# 将源代码添加到此项目的可执行文件。
add_executable (CMakeProject1 "invsqrt.cpp" "main.cpp" "invsqrt.h" )

# TODO: 如有需要，请添加测试并安装目标。
```

3. 配置好上述工程后，直接生成即可。

写在最后

CMake还有很多强大的功能：

- 设置C++工程的语言标准、编译优化选项。
- 层级文件之间 `CMakeLists.txt` 的相互调用，以便应用于目录层级更加复杂的C++工程。
- 对生成的库、可执行文件等进行安装。
- ...

略过上述内容不会对我们的教学产生太大影响。感兴趣的同学可以参考以下文章：

- CMake官方文档
- cmake-examples 该GitHub仓库中有很多开箱即用的CMake实例。
- 为什么编译c/c++要用makefile，而不是直接用shell呢？ 这篇博文详细地阐述了使用makefile的动机和意义（\xfgg/）。
- 跟我一起写Makefile Makefile教程。从中大家也可以看出Makefile的语法十分不友好...