CMake Learn

跳过概述，直接看教程

说在前面

什么是gcc

`gcc` 是 GNU Compiler Collection（GNU编译器套装）的缩写，它是由GNU计划开发的一套编程语言编译器。`gcc` 主要用于编译和链接C、C++、Fortran等编程语言的源代码，它是一个强大而灵活的编译器套件，支持多种平台和操作系统

gcc常用编译参数

- `-c`:: 指定源文件

```
gcc -c myfile.c
```

- `-o`： 指定输出文件名称

```
gcc -o myprogram myfile.c
# 将myfile.c编译为myprogram
```

- `-On`： 指定编译器优化级别。从 `-O0` 到 `-O3`
- `-g`： 生成debug信息，用于调试

```
gcc -g myfile.c
```

- `-l`： 指定链接库文件

```
gcc -lmylib myprogram.c
```

- `-L`： 指定库文件路径

```
gcc -L/path/to/lib myprogram.c
```

- `-static`：生成静态库
- `-shared`：生成动态库

传统gcc编译方式

当编译c++文件的时候，简单情况下，我们通常会使用gcc直接进行编译,例如：

```
g++ -o myprogram file1.cpp file2.cpp
# 该命令将file1.cpp file2.cpp编译为一个叫myprogram的可执行文件
```

但是这仅限于简单的编译任务，当文件变多，项目变复杂的时候，在手动去用gcc编译就变得不现实，我们需要处理繁杂的头文件引用关系、target生成的依赖关系、还有众多的编译链接参数，这通常会让我们感觉到手足无措

为什么要用CMake

构建工具

> 构建工具是一种用于自动化软件项目构建过程的工具，其主要任务是将源代码转换为可执行文件或库。构建工具负责编译源代码、链接库、管理依赖关系、生成文档等一系列任务，以便开发者能够更轻松地管理和部署他们的软件项目

主要的构建工具包括但不限于：

- Make
- Ninja

为什么要使用构建工具呢?

1. **自动化构建过程：** 构建工具能够自动执行编译、链接、打包等繁琐的构建任务，提高了开发效率。
2. **跨平台支持：** 跨平台构建工具如CMake允许项目在不同的操作系统上进行构建，减少了维护多个构建系统文件的工作。
3. **依赖管理：** 构建工具可以管理项目的依赖关系，自动下载和配置所需的库和工具。
4. **一致的构建规则：** 构建工具通过配置文件或脚本提供了一致的构建规则，使得不同开发者在不同环境中能够按照相同的规则构建项目。

5. **增量构建：** 构建工具通常能够进行增量构建，只重新构建发生变化的部分，减少了不必要的重复工作。
6. **测试和部署：** 构建工具可以集成测试和部署步骤，帮助确保项目的质量和可靠性。

使用CMake进行编译

> 使用CMake的主要原因是简化跨平台项目的构建过程和管理，提供了更灵活、可维护的方式来定义和配置项目的构建规则

1. **跨平台支持：** CMake能够根据目标平台生成适用于该平台的构建系统文件，使得你的项目能够在不同的操作系统上进行构建，而不必为每个平台手动编写不同的构建命令。
2. **简化配置：** CMake提供了简单的脚本语言，让你在一个地方描述项目的构建规则，而不是在多个平台上维护不同的构建命令。这使得配置项目变得更加简单和一致。
3. **模块化项目结构：** 通过CMake，你可以将项目拆分为多个子项目，每个子项目有自己的构建规则。这有助于组织和管理大型项目，使得代码结构更加清晰。
4. **灵活选择构建工具：** CMake支持多种构建工具，如Make、Ninja、Visual Studio等。你可以根据自己的偏好和项目需求选择适合的构建工具，而不必为每个平台编写特定的构建脚本。
5. **易于移植：** 使用CMake，项目的构建规则与特定平台解耦，使得在不同的操作系统或编译器中切换更为容易。只需重新运行CMake，就能够生成适配目标平台的构建系统文件。

CMake教程

准备工作

CMake工具

准备下CMake的工具链，这里就不进行赘述了

CMakeLists.txt

在项目的根目录下创建一个CMakeLists.txt文件，用于描述项目的构建规则。这个文件包含了一系列的指令，定义了项目的源文件、目标、依赖关系等

构建目录

在项目根目录外创建一个构建目录，用于存放生成的构建系统文件和编译中间文件。这通常是为了保持源代码目录的清洁

了解CMake变量

在CMake中，变量是一种用于存储和传递信息的机制。变量可以包含文本、路径、数字等不同类型的数据。CMake中的变量分为几种类型，包括普通变量、缓存变量、环境变量等

- **定义变量**

```
set(my_variable "Hello, CMake!")
```

- **获取变量的值**

> 通过 ${} 获取变量的值

```
message(STATUS "Value of my_variable: ${my_variable}")
```

单个文件编译

> 将单个c++源文件编译为可执行文件

目录结构

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

文件内容

main.cc

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}

```

CMakeLists.txt

```
# set minimum cmake version
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)

# project name and language
project(singleFile LANGUAGES CXX)

add_executable(main main.cc)
```

- `cmake_minimum_required(VERSION 3.5 FATAL_ERROR)`：指定项目所需的最低CMake版本为3.5。如果安装的CMake版本低于指定版本，将产生致命错误并停止构建。
- `project(singleFile LANGUAGES CXX)`：定义项目的名称为 `singleFile`，并指定项目使用的编程语言为C++（`CXX` 表示C++）。这一行也可以包含其他参数，例如 `VERSION` 来指定项目的版本号。
- `add_executable(main main.cc)`：声明一个可执行文件的目标。这一行指示CMake编译器使用 `main.cc` 作为源文件，生成的可执行文件的名称为 `main`。这个目标将在后续的构建过程中被编译。相当于 `g++ -o main main.cc`

两种编译方式

in-source（源码内部构建）

在这种构建方式下，构建过程发生在源代码目录中，生成的可执行文件、库文件等会直接放在源码目录中。虽然这是最简单直观的方式，但它有一些缺点，最主要的是可能会污染源代码目录

```
cmake .
make
```

out-of-source（源码外部构建）

在这种构建方式下，构建过程发生在源码目录之外的一个独立的构建目录中。这种方式有助于保持源码目录的干净，并允许在同一个源码目录下创建多个独立的构建目录，以支持不同的构建配置

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

> 建议只用外部构建

include头文件

锚点

目录结构

```
.
├── CMakeLists.txt
├── main.cc
└── main.h
```

文件内容

main.h

> HelloWorld函数的定义

```h
#ifndef CMAKE_LEARN_MAIN_H_ // 头文件保护，建议每个头文件都加，防止重定义
#define CMAKE_LEARN_MAIN_H_

#include <iostream>

void HelloWorld();

#endif // CMAKE_LEARN_MAIN_H_
```

main.cc

> HelloWorld函数的实现和main函数

```cpp
#include "main.h"

void HelloWorld() {
    std::cout << "Hello, World!" << std::endl;
}
int main() {
    HelloWorld();
    return 0;
}


```

CMakeLists.txt

```cmake
# set minimum cmake version
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)

# project name and language
project(singleFile LANGUAGES CXX)

add_executable(main main.cc)

target_include_directories(main PRIVATE ${CMAKE_SOURCE_DIR})
```

> c++中的include
>
> - `#include "header.h"` ： 使用双引号包裹的形式
>
> 这种形式是用于包含用户自定义的头文件，通常是项目内部的头文件。编译器首先在当前源文件的目录中查找该头文件，如果找不到，则会在系统的标准头文件路径中继续查找
>
> - `#include <header.h>` ： 使用尖括号包裹的形式
>
> 这种形式是用于包含标准库头文件或系统提供的头文件。编译器会直接在系统的标准头文件路径中查找该头文件

`target_include_directories` ： 函数用于向指定的 `target` （例如可执行文件、库）添加头文件搜索路径，这样，编译器在编译目标时就能够找到需要包含的头文件。

常用语法:

```
1 target_include_directories(target PRIVATE DIR)
```

- `target` 在这里表示一个构建目标，这个目标可以是库文件、可执行文件等，这里的target是由第7行的 `add_executable` 生成， 名字叫做 `main`
- `PRIVATE` 在这里指定添加的路径的可见性， `PRIVATE` 表示路径仅对目标可见，此外还可以是 `PUBLIC` （表示路径对目标和依赖于目标的目标都可见）、 `INTERFACE` （表示路径在接口中可见），这里搞不懂没关系，知道有这几个就行，默认写 `PRIVATE` 就好
- `DIR` ： DIR可以是一个CMake变量， 也可以直接写绝对路径，这里 `${CMAKE_SOURCE_DIR}` 表示项目代码根目录的绝对路径
- `CMAKE_SOURCE_DIR` ： 是一个预定义的CMake变量，表示当前 CMakeLists.txt 所在的目录的绝对路径。具体来说，它是用于定义项目的**根目录**，即包含项目的顶层 CMakeLists.txt 文件的目录，这里要着重记忆一下，后面还有子文件夹，每个文件夹里面也包含CMakeLists.txt，但是只有代码顶层的这个CMakeLists.txt才叫做顶层CMakeLists.txt，而 `CMAKE_SOURCE_DIR` 就是指向顶层CMakeLists.txt所在路径的一个预设变量（预设变量可以理解为你在CMakeLists.txt可以直接访问的一些变量，它是工具指定的）
- `target_include_directories` 的位置： `target_include_directories` 需要放在生成target的语句之后，这个很好理解，如果当前都没生成target，那怎么为target指定头文件搜索路径呢?
- `target_include_directories` 的效果： `target_include_directories` 的效果相当于g++的 `-I`

include其它路径的头文件

目录结构

```
.
├── CMakeLists.txt
├── include
│   └── main.h
└── main.cc
```

文件内容

main.h和main.cc文件内容和include头文件一致

CMakeLists.txt

- 第一种写法:

```cmake
# set minimum cmake version
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)

# project name and language
project(singleFile LANGUAGES CXX)

add_executable(main main.cc)

target_include_directories(main PRIVATE ${CMAKE_SOURCE_DIR})
```

这种写法还是将顶层路径加入到了main的头文件搜索路径下面，但是值得注意的是我们现在的main.h是放在include目录下，并没有在顶层目录，这样会导致main.cc里面 `#include "main.h"` 语句找不到头文件

```
# root @ iZ2vca7ty2zomyxrqq57t8Z in ~/workspace/project/build [17:34:41]
$ make
[ 50%] Building CXX object CMakeFiles/main.dir/main.cc.o
/root/workspace/project/cmake-learn/main.cc:1:10: fatal error: main.h: No such file or directory
    1 | #include "main.h"
      |          ^~~~~~~~
compilation terminated.
make[2]: *** [CMakeFiles/main.dir/build.make:76: CMakeFiles/main.dir/main.cc.o] Error 1
make[1]: *** [CMakeFiles/Makefile2:83: CMakeFiles/main.dir/all] Error 2
make: *** [Makefile:91: all] Error 2
(base)
```

此时我们可以适当修改该头文件引用为 `#include "include/main.h"` 即可

- 第二种写法：

```cmake
# set minimum cmake version
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)

# project name and language
project(singleFile LANGUAGES CXX)

add_executable(main main.cc)

target_include_directories(main PRIVATE ${CMAKE_SOURCE_DIR}/include)
```

这种写法还是将顶层路径下的include加入到了main的头文件搜索路径下面，所以直接使用 `#include "main.h"` 可以正确搜索到头文件，不用修改头文件引用

多个源文件的编译

目录结构

```
.
├── CMakeLists.txt
├── func.cc
├── include
│   └── func.h
└── main.cc
```

文件内容

func.h

> HelloWorld函数的定义

```h
#ifndef CMAKE_LEARN_MAIN_H_
#define CMAKE_LEARN_MAIN_H_

#include <iostream>

void HelloWorld();

#endif // CMAKE_LEARN_MAIN_H_
```

func.cc

> HelloWorld函数的实现

```cc
#include "func.h"

void HelloWorld() {
    std::cout << "Hello, World!" << std::endl;
}
```

main.cc

> main函数

```
#include "func.h"

int main() {
    HelloWorld();
    return 0;
}
```

CMakeLists.txt

```
# set minimum cmake version
cmake_minimum_required(VERSION 3.5 FATAL_ERROR)

# project name and language
project(singleFile LANGUAGES CXX)

add_executable(main main.cc func.cc)

target_include_directories(main PRIVATE ${CMAKE_SOURCE_DIR})
```

这里与上述例子的区别在于第7行，这里指定了多个源文件

TODO(dingtao.lu)：写一下怎么封装多个源文件到一个变量里面