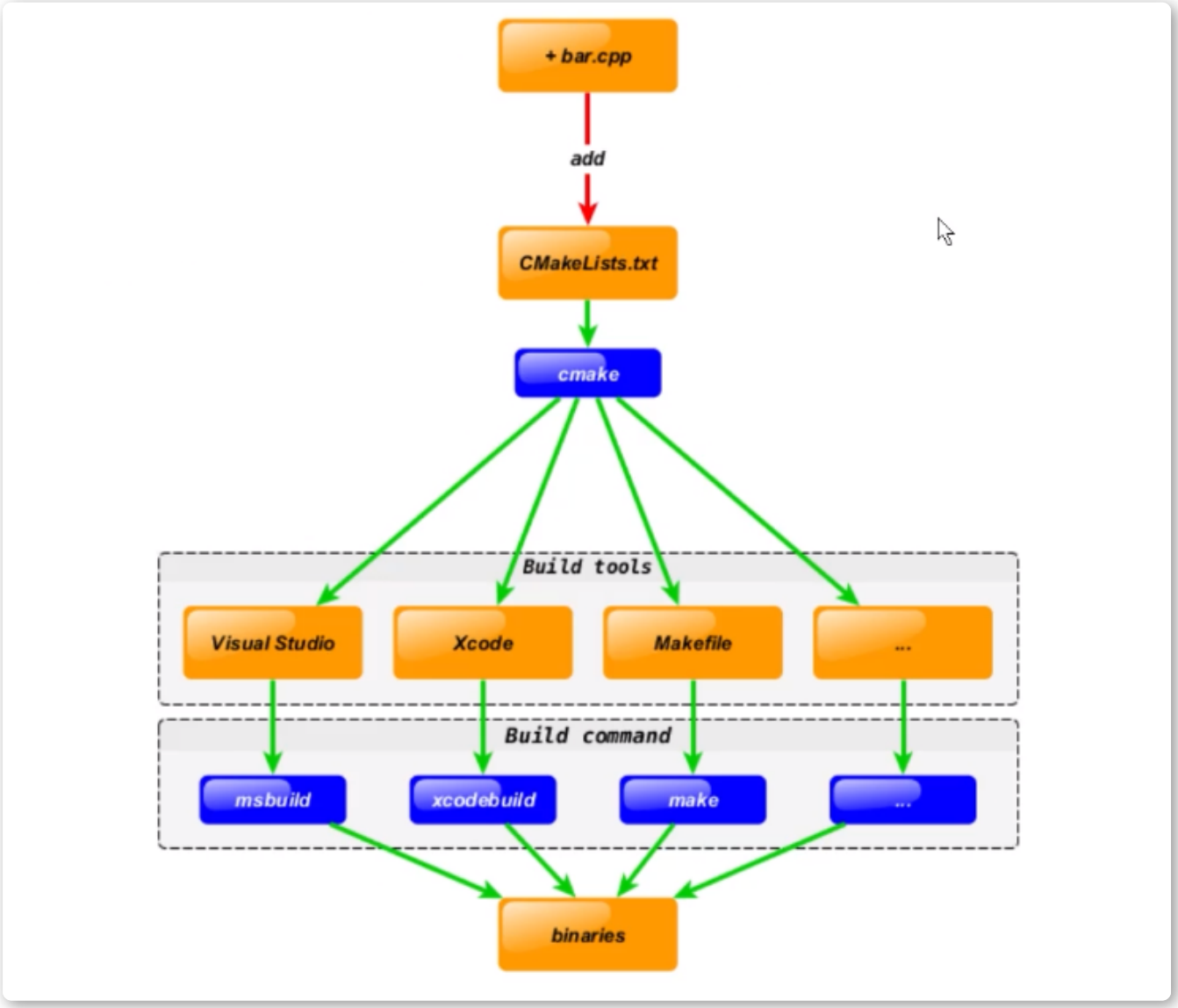
CMake笔记

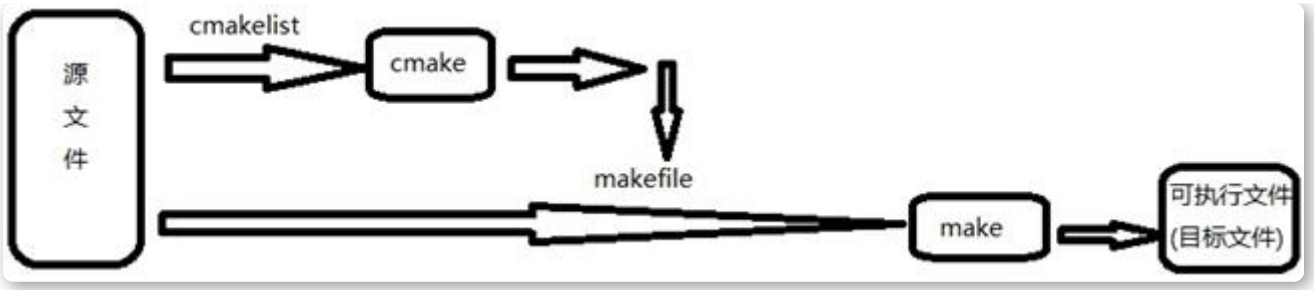
简介

CMake是一个跨平台的、开源的构建工具。cmake是makefile的上层工具，它们的目的正是为了产生可移植的makefile，并简化自己动手写makefile时的巨大工作量.目前很多开源的项目都可以通过CMake工具来轻松构建工程，例如博客之前分享的openHMD、hidapi、OSVR-Core等等，代码的分享者提供源代码和相应的Cmake配置文件，使用者就可以非常方便的在自己的电脑上构建相应的工程，进行开发和调试。

Cmake可以根据所在的操作系统生成相应类型的makefile

构建流程图如下：



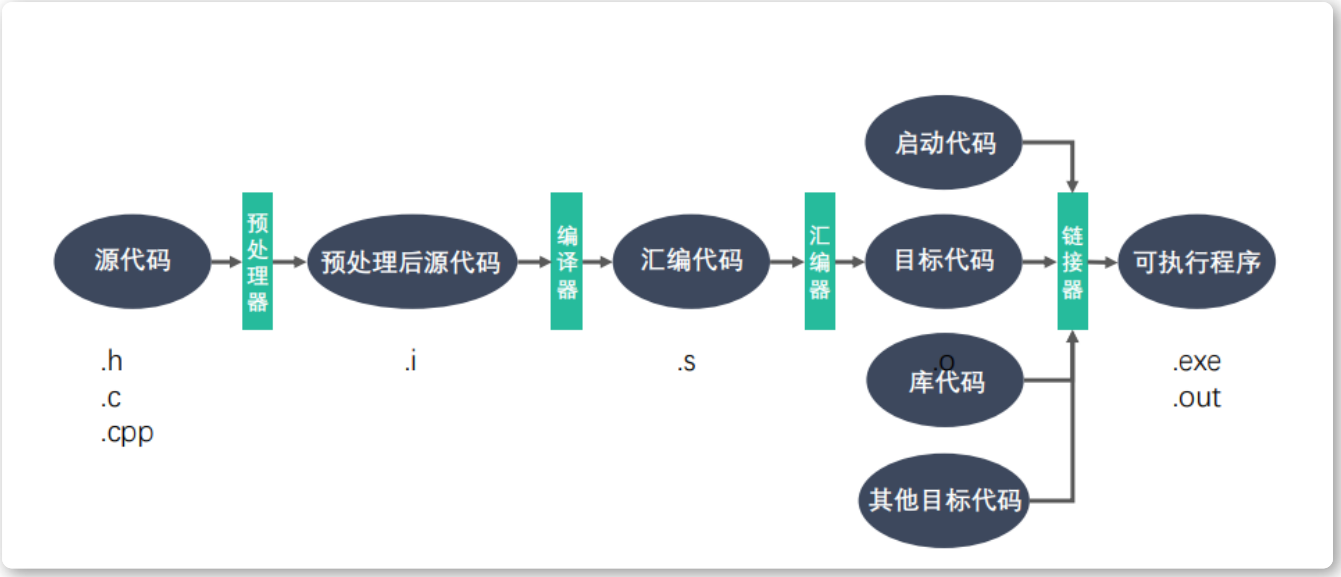


gcc/g++编译命令汇总

简介

- GCC 原名为 GNU C语言编译器（GNU C Compiler）
- GCC（GNU Compiler Collection，GNU编译器套件）是由 GNU 开发的编程语言 译器。GNU 编译器套件包括 C、C++、Objective-C、Java、Ada 和 Go 语言前 端，也包括了这些语言的库（如 libstdc++，libgcj等）
- GCC 不仅支持 C 的许多“方言”，也可以区别不同的 C 语言标准；可以使用命令行 选项来控制编译器在翻译源代码时应该遵循哪个 C 标准。例如，当使用命令行参数 `-std=c99` 启动 GCC 时，编译器支持 C99 标准。
- 安装命令 sudo apt install gcc g++ （版本 > 4.8.5）
- 查看版本 gcc/g++ -v/--version

GCC/G++ 工作流程



GCC和G++的区别

- gcc 和 g++都是GNU(组织)的一个编译器。
- 误区一：gcc 只能编译 c 代码，g++ 只能编译 c++ 代码。两者都可以，请注意：
  - 后缀为 .c 的，gcc 把它当作是 C 程序，而 g++ 当作是 c++ 程序
  - 后缀为 .cpp 的，两者都会认为是 C++ 程序，C++ 的语法规则更加严谨一些
  - 编译阶段，g++ 会调用 gcc，对于 C++ 代码，两者是等价的，但是因为 gcc 命令不能自动和 C++ 程序使用的库联接，所以通常用 g++ 来完成链接，为了统 一起见，干脆编译/链接统统用 g++ 了，这就给人一种错觉，好像 cpp 程序只 能用 g++ 似的 04 / gcc 和 g++ 的区别
- 误区二：gcc 不会定义 __cplusplus 宏，而 g++ 会
  - 实际上，这个宏只是标志着编译器将会把代码按 C 还是 C++ 语法来解释
  - 如上所述，如果后缀为 .c，并且采用 gcc 编译器，则该宏就是未定义的，否则， 就是已定义
- 误区三：编译只能用 gcc，链接只能用 g++
  - 严格来说，这句话不算错误，但是它混淆了概念，应该这样说：编译可以用 gcc/g++，而链接可以用 g++ 或者 gcc -lstdc++。
  - gcc 命令不能自动和C++程序使用的库联接，所以通常使用 g++ 来完成联接。 但在编译阶段，g++ 会自动调用 gcc，二者等价

GCC/G++ 命令常用参数选项

gcc编译选项	说明
-E	预处理指定的源文件，不进行编译
-S	编译指定的源文件，但是不进行汇编
-c	编译、汇编指定的源文件，但是不进行链接
-o [file1] [file2] / [file2] -o [file1]	将文件 file2 编译成可执行文件 file1
-I directory	指定 include 包含文件的搜索目录
-g	在编译的时候，生成调试信息，该程序可以被调试器调试
-D	在程序编译的时候，指定一个宏
-w	不生成任何警告信息

gcc编译选项	说明
-Wall	生成所有警告信息
-On	n的取值范围：0~3。编译器的优化选项的4个级别，-O0表示没有优化，-O1为缺省值，-O3优化级别最高
-l	在程序编译的时候，指定使用的库
-L	指定编译的时候，搜索的库的路径。
-fPIC/fpic	生成与位置无关的代码
-shared	生成共享目标文件，通常用在建立共享库时
-std	指定C方言，如:-std=c99，gcc默认的方言是GNU C

CMake语法特性介绍

- **基本语法格式：指令(参数1 参数2...)**
  - 参数要用括号括起来
  - 参数之间用空格或分号分隔

- **指令是大小写无关的，参数和变量是大小写相关的**

```
set(HELLO hello.cpp)
add_executable(hello main.cpp hello.cpp)
ADD_EXECUTABLE(hello main.cpp ${HELLO})
```

- **变量使用 ${} 方式取值，但是在 IF 控制语句中是直接使用变量名**

重要指令和CMake常用变量

重要指令

- **cmake_minimum_required - 指定CMake的最小版本要求**

- 语法：**cmake_minimum_required(VERSION versionNumber [FATAL_ERROR])**

```
# CMake最小版本要求为2.8.3
cmake_minimum_required(VERSION 2.8.3)
```

- **project - 定义工程名称，并可指定工程支持的语言**
  - 语法：**project(projectname [CXX] [C] [JAVA])**

```
# 指定工程名为HELLOWORLD
project(HELLOWORLD)
```

- **set - 显示的定义变量**
  - 语法：**set(VAR [VALUE] [CACHE TYPE DOCSTRING [FORCE]])**

```
# 定义SRC变量，其值为sayhello.cpp hellp.cpp
set(SRC sayhello.cpp hello.cpp)
```

- **include_directories - 向工程添加多个特定的头文件搜索路径-->相当于指定g++编译器的-I 参数**
  - 语法：**include_directories([AFTER] [BEFORE] [SYSTEM] dir1 dir2 ...)**

```
# 将/usr/include/myincludefolder 和 ./include 添加到头文件搜索路径
include_directories(/usr/include/myincludefolder ./include)
```

- **link_directories - 向工程添加多个特定的库文件搜索路径 -->相当于指定g++编译器的-L参数**
  - 语法：**link_directories(dir1 dir2 ...)**

```
# 将/usr/lib/mylibfolder 和 ./lib 添加到库文件搜索路径
link_directories(/usr/lib/mylibfolder ./lib)
```

- **add_library - 生成库文件**
  - 语法：**add_library(libname [SHARED|STATIC|MODULE] [EXCLUDE_FROM_ALL] source1 source2 ... sourceN)**
    - SHARED:动态库
    - STATIC:静态库
    - MODULE:C++比较少用到

```
# 通过 SRC 变量生成 libhello.so 共享库
add_library(libhello SHARED ${SRC})
```

- **add_compile_options - 添加编译参数**
  - 语法：**add_compile_options(<option> ...)**

```
# 添加编译参数 -Wall -std=c++11
add_compile_options(-Wall -std=c++11 -o2)
```

- **add_executable - 生成可执行文件**
  - 语法：**add_executable(exename source1 source2 ... sourceN)**

```
# 编译main.cpp生成可执行文件main
add_executable(main main.cpp)
```

- **target_link_libraries - 为 target 添加需要链接的共享库 --> 相当于g++指定编译器 -l 参数**
  - 语法：**target_link_libraries(target library1<debug | optimized> library2 ...)**

```
# 将hello动态文件链接到可执行文件main
target_link_libraries(main hello)
```

- **add_subdirectory - 向当前工程添加存放源文件的子目录，并可以指定中间二进制和目标二进制存放的位置**
  - 语法：**add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])**
  - 注意：要使用此命令，子目录中需要有一个 CMakeLists.txt

```
# 添加 src 子目录，src中需要有一个CMakeLists.txt
add_subdirectory(src)
```

- **aux_source_directory - 发现一个目录下所有的源代码文件并将列表存储在一个变量中，这个指令临时被用来自动构建源文件列表**
  - 语法：**aux_source_directory(dir VARIABLE)**

```
# 定义SRC变量，其值为当前目录下所有的源代码文件
aux_source_directory(. SRC)
# 编译SRC变量所代表的源代码文件，生成main可执行文件
add_executable(main ${SRC})
```

- **check_symbol_exists - 查找相关文件（FILES）里面是否包含相关符号[SYMBOL]，如果存在则设置VARIABLE 为1。**
  - 语法：check_symbol_exists( )
  - 注意：使用 `check_symbol_exists宏需要在` CMake文件中包含CheckSymbolExists，即：

    ```
    include(CheckSymbolExists)
    ```

  - 例子：

    ```
    # 在CMake文件中包含CheckSymbolExists
    include(CheckSymbolExists)

    # Check for macro SEEK_SET
    # 检查头文件stdio.h中是否有SEEK_SET宏，若有则定义HAVE_SEEK_SET
    缓存变量，固定值设置为1
    check_symbol_exists(SEEK_SET "stdio.h" HAVE_SEEK_SET)

    # Check for function fopen
    # 检查头文件stdio.h中是否有fopen函数，若有则定义HAVE_FOPEN缓存变
    量，固定值设置为1
    check_symbol_exists(fopen "stdio.h" HAVE_FOPEN)

    # check epoll and add config.h for the macro compilation
    include(CheckSymbolExists)
    check_symbol_exists(epoll_create "sys/epoll.h"
    EPOLL_EXISTS)
    if (EPOLL_EXISTS)
        # Linux下设置为epoll
        set(EPOLL_ENABLE 1 CACHE INTERNAL "enable epoll")

        # Linux下也设置为poll
    #    # set(EPOLL_ENABLE "" CACHE INTERNAL "not enable
    epoll")
    else ()
        set(EPOLL_ENABLE "" CACHE INTERNAL "not enable epoll")
    endif ()
    ```

CMake常用变量

- **CMAKE_C_FLAGS** - gcc编译选项
- **CMAKE_CXX_FLAGS** - g++编译选项

```
# 在CMAKE_CXX_FLAGS编译选项后追加-std=c++11
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11")
```

- **CMAKE_BUILD_TYPE** - 编译类型（Debug, Release）

```
# 设定编译类型为Debug，调试时使用
set(CMAKE_BUILD_TYPE Debug)
# 设定编译类型为Release，发布使用
set(CMAKE_BUILD_TYPE Release)
```

- **CMAKE_BINARY_DIR PROJECT_BINARY_DIR _BINARY_DIR**
  - 这三个变量指定的内容是一致的
  - 如果是 in source build，指的就是工程顶层目录
  - 如果是 out source build，指的是工程编译发生的目录
  - PROJECT_BINARY_DIR 跟其它指令稍有区别
- **CMAKE_SOURCE_DIR PROJECT_SOURCE_DIR _SOURCE_DIR**
  - 这三个变量指定的内容是一致的，不论采用哪种编译方式，都是工程顶层目录
  - 也就是在in source build 时，他跟 CMAKE_BINARY_DIR 等变量一致
  - PROJECT_SOURCE_DIR 跟其它指令稍有区别
- **CMAKE_C_COMPILER** - 指定C编译器
- **CMAKE_CXX_COMPILER** - 指定C++编译器
- **EXECUTABLE_OUTPUT_PATH** - 可执行文件输出的存放路径
- **LIBRARY_OUTPUT_PATH** - 库文件输出的存放路径

CMake编译工程

CMake目录结构：项目主目录存在一个 **CMakeLists.txt** 文件

两种方式设置编译规则：

1. 包含源文件的子文件夹 **包含CMakeLists.txt**文件，主目录的CMakeLists.txt通过add_subdirectory添加子目录即可；
2. 包含源文件的子文件夹 **不包含CMakeLists.txt**文件，子目录编译规则体现在主目录的CMakeLists.txt中；

编译流程

在Linux平台下使用 CMake 构建 C/C++ 工程的流程如下：

- 手动编写 CMakeLists.txt
- 执行命令 `cmake PATH` 生成 Makefile (PATH是顶层CMakeLists.txt所在的目录)
- 执行命令 `make` 进行编译

```
# important tips
.        # 表示当前目录
./       # 表示当前目录

..       # 表示上级目录
../      # 表示上级目录
```

两种构建方式

- **内部构建（in-source build）：不推荐使用**

  内部构建会在统计目录下生成大量中间文件，这些中间文件并不是我们最终所需要的，和工程源文件放在一起会显得杂乱无章。

  ```
  ## 内部构建

  # 在当前目录下，编译本目录的CMakeLists.txt，生成Makefile和其它文件
  $ cmake .
  # 执行make命令，生成target
  $ make
  ```

- **外部构建（out-of-source build）：推荐使用**

  将编译输出文件和源文件放在不同的目录中

```
## 外部构建

# 1.在当前目录下，创建build文件夹
$ mkdir build
# 2.进入到build文件夹
$ cd build
# 3. 编译上级目录的CMakeLists.txt，生成Makefile和其它文件
$ cmake ..
# 4. 执行make命令，生成target
$ make
```

实战

创建一个测试工程，目录结构如下：

```
sky@sky:~/demo/myTestProject$ tree .
.
├── build
├── CMakeLists.txt
├── include
│   ├── Gun.h
│   └── soldier.h
├── main.cpp
└── src
    ├── Gun.cpp
    └── soldier.cpp

3 directories, 6 files
```

CMakeLists.txt文件内容如下：

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)

project(SOLDIER)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -g -o2 -Wall -std=c++11")

include_directories(${CMAKE_SOURCE_DIR}/include)

add_executable(my_cmake_exe main.cpp src/Gun.cpp src/soldier.cpp)
```

进入build目录下，执行 cmake .. 命令，会生成一个makefile文件：

```
sky@sky:~/demo/myTestProject/build$ cmake ..
-- The C compiler identification is GNU 7.5.0
-- The CXX compiler identification is GNU 7.5.0
-- Check for working C compiler: /usr/bin/cc
-- Check for working C compiler: /usr/bin/cc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/sky/demo/myTestProject/build
```

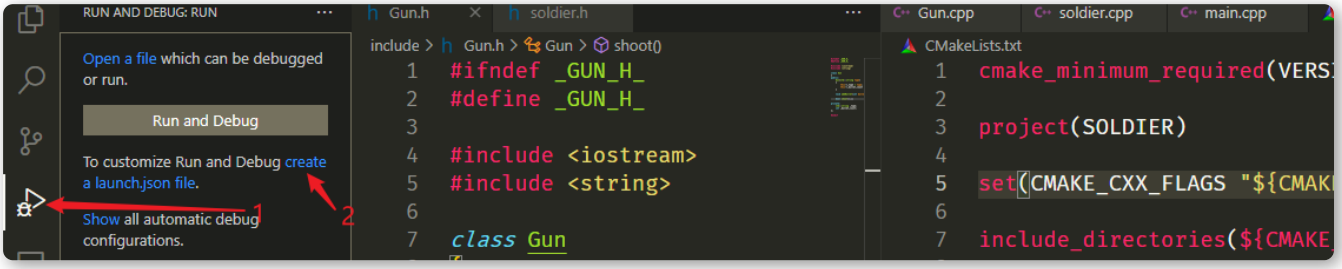
接着执行 make 命令：

```
sky@sky:~/demo/myTestProject/build$ make
Scanning dependencies of target my_cmake_exe
[ 25%] Building CXX object CMakeFiles/my_cmake_exe.dir/main.cpp.o
[ 50%] Building CXX object CMakeFiles/my_cmake_exe.dir/src/Gun.cpp.o
[ 75%] Building CXX object CMakeFiles/my_cmake_exe.dir/src/soldier.cpp.o
[100%] Linking CXX executable my_cmake_exe
[100%] Built target my_cmake_exe
sky@sky:~/demo/myTestProject/build$
```

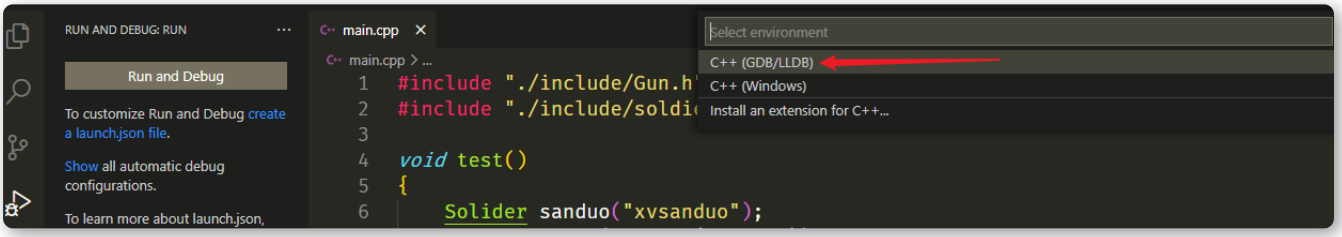
至此项目构建完成，可以运行看看。

vscode调试文件配置

1. 点击左侧工具栏的调试按钮，接着点击 "create a launch.json file" 选项：

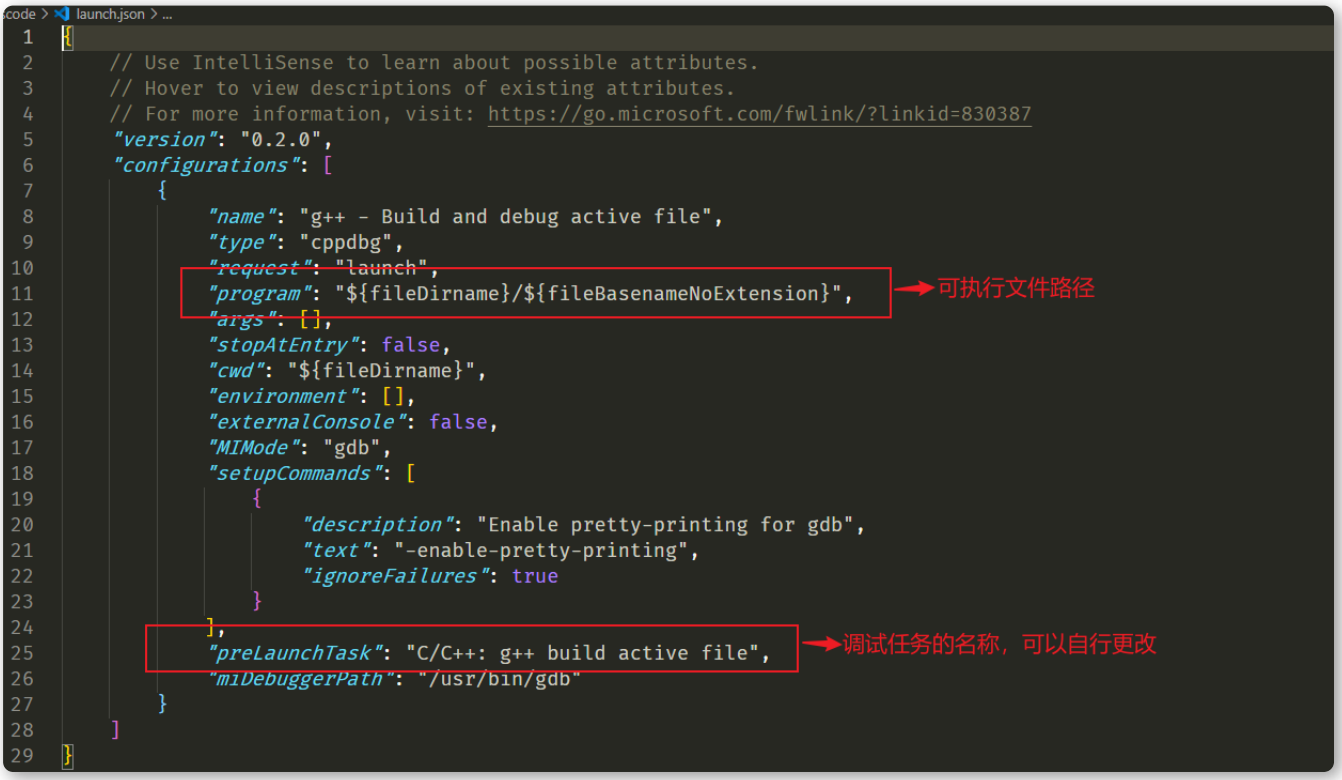


选择 **C++(GDB/LLDB):**



接着选择 **g++ Build and debug activate file**，出现错误提示直接点击 **Abort**跳过

生成的 **launch.json** 文件内容如下：

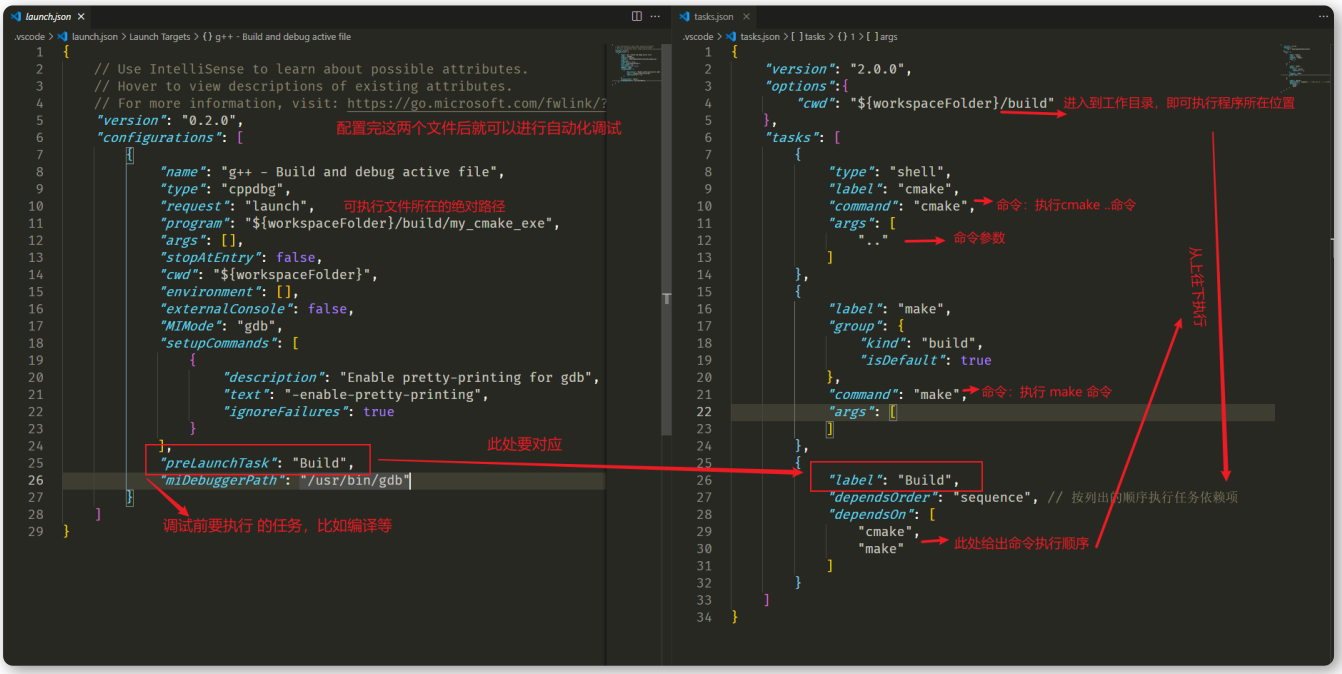


2. 调整CMakeLists.txt的内容，编译类型设置为调试模式：Debug，然后编译:

```
# CMakeLists.txt内容
cmake_minimum_required(VERSION 3.0)

project(SOLDIER)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -std=c++11")

# 设置为调试模式
set(CMAKE_BUILD_TYPE Debug)

include_directories(${CMAKE_SOURCE_DIR}/include)

add_executable(my_cmake_exe main.cpp src/Gun.cpp src/soldier.cpp)
```

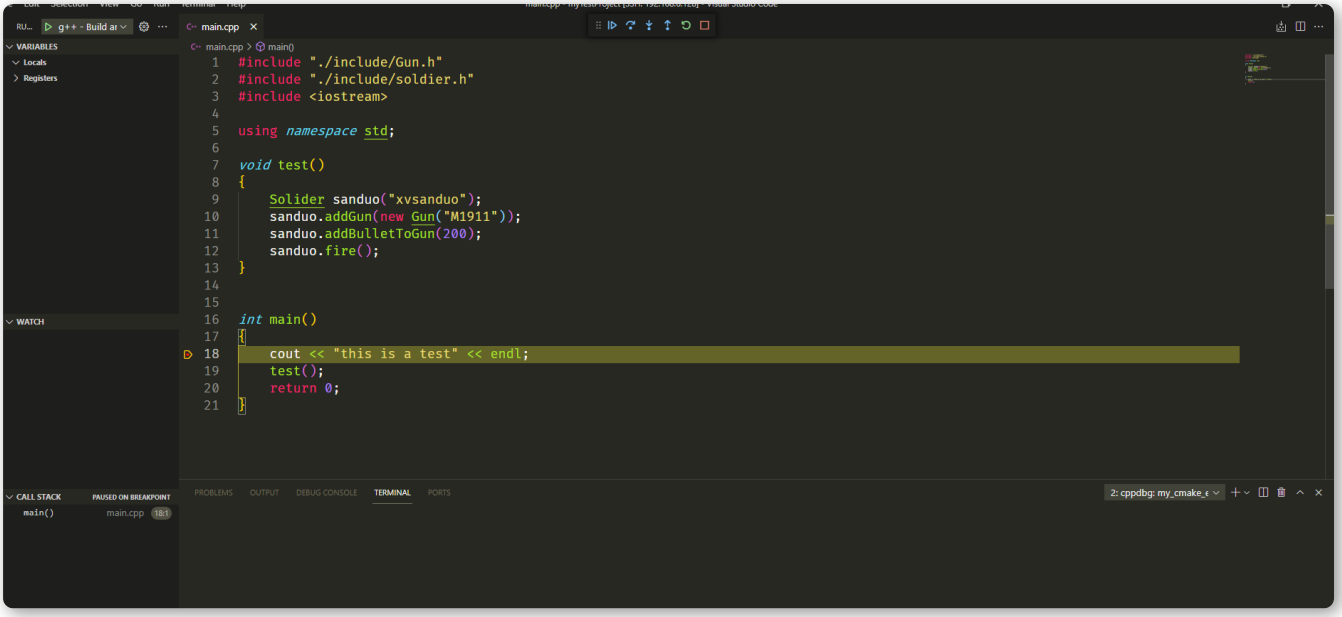
3.调整 launch.json 和 tasks.json 两个文件内容：

配置完以后，如果调整了代码内容，直接调试的时候会自动编译



4.最后按 F5 进行调试：

调试界面如下：



总结

- CMake简化了自己动手写makefile时的巨大工作量
- 可移植性好
- 现在越来越受欢迎