项目开始前的零碎准备

linux重要命令

pwd 打印当前工作目录

ls

- **用法3：ls 选项**

  含义：在列出指定路径下的文件/文件夹的名称，并以指定的格式进行显示。

```
# ls 选项 路径
ls -lah /home
# 选项解释:
    -l: 表示list，表示以详细列表的形式进行展示
    -a: 表示显示所有的文件/文件夹（包含了隐藏文件/文件夹）
    -h: 表示以可读性较高的形式显示
# ls -l 中 “-”表示改行对应的文档类型为文件，“d”表示文档类型为文件夹。
# 在Linux中隐藏文档一般都是以“.”开头
```

一次创建多层不存在的目录

mkdir -p a/b/c

创建新文件 touch

复制文件或文件夹到指定路径 cp

- **用法1：cp [被复制的文件路径][文件被复制到的路径]**

```
# cp命令来复制一个文件
cp /home/bing/myfile ./
```

- **用法2：cp -r 被复制的文件夹路径**

  含义：-r 表示递归复制，复制文件夹的时候需要加 -r

```
# 复制/home/bing/myfolder文件夹到根目录/下
cp -r /home/bing/myfolder /
```

移动文件到新的位置或者重命名 mv

mv - move (rename) files

- 作用：移动文件到新的位置，或者重命名文件
- 用法：mv 需要移动的文件路径

```
# 移动当前目录下myfile文件到根目录/下
mv myfile /myfile

# 移动当前目录下myfolder文件夹到根目录/下
mv myfolder /myfolder

# 移动当前目录下myfile到根目录/下，并重命名为myfile007
mv myfile myfile007
```

查看命令的使用 man

man ls

man cd

man man

查看之后按q退出

没有man的用help代替也可以

重启linux系统 reboot

关机 shutdown -h now

shutdown -h [时间]

开发环境的搭建

sudo 切换到root用户下执行命令

```
sudo apt update

sudo apt install build-essential gdb
//查看软件是否安装成功
gcc --version
g++ --version
gdb --version
sudo apt install cmake
cmake --version
```

g++编译器

c++的编译器

一般使用gcc指令编译c代码

一般使用g++指令编译c++代码

编译过程

```
#预处理命令
# -E 选项指示编译器仅对输入文件进行预处理
g++ -E test.cpp -o test.i
#编译
# -S 编译选项告诉g++在为c++代码产生了汇编语言文件后停止编译
#g++产生的汇编语言文件的缺省扩展名是 .S
g++ -S test,i -o test.S
#汇编
# -c选项告诉g++仅把源代码编译为机器语言的目标代码
# 缺省时g++建立的目标代码文件有一个.o的扩展名
g++ -c test.S -o test.o
#链接
#-o 编译选项来为将产生的可执行文件用指定的文件名
g++ test.o -o test
#上面这些命令写成一句就行
g++ test.cpp -o test
```

:set ts=4

设置tab键为4个空格

g++重要编译参数

1. -g 编译带调试信息的可执行文件

```
#-g选项告诉gcc产生能被gnu调试器gdb使用的调试信息，以调试程序
#产生待调试信息的可执行文件test
g++ -g test.cpp -o test
```

2. -O[n] 优化源代码

```
g++ -O2 test.cpp
#也可以直接生成可执行文件
g++ test.cpp -O2 test
#想要查看文件是否被优化可以查看文件的执行时间
time ./test
```

3. -l和-L 指定库文件|指定库文件路径

```
# -l参数用来指定程序要链接的库，-l参数紧接着就是库名
#在/lib 和/usr/lib 和/usr/local/lib里的库直接用-l参数就能链接
#链接glog库
g++ -lglog test.cpp

#如果库文件没在三个目录里，需要使用-L参数指定库文件所在目录
#-L参数跟着的是库文件所在的目录名

#链接mytest库
g++ -L/home/bing/mytestlibfolder -lmytest test.cpp
```

4. -I 指定头文件搜索目录

```
#-I
#/usr/include 目录一般是不用指定的，如果头文件不在这里需要用-I参数指定,绝对路径或者相对路径都可以
g++ -I/myinclude test.cpp
```

5. -Wall 打印警告信息

```
g++ -Wall test.cpp
```

6. -w 关闭警告信息

```
g++ -w test.cpp
```

7. -std=c++11 设置编译标准

```
g++ -std=c++11 test.cpp
```

8. -o 指定输出文件名

```
g++ test.cpp -o test
```

9. -D 定义宏

```
#在使用gcc/g++编译的时候定义宏
#常用场景
# -DDEBUG定义DEBUG宏，可能文件中有DEBUG宏部分的相关信息，用个DDEBUG来选择开启或者关闭DEBUG
```

示例代码：

```
// -Dname 定义宏name,默认定义内容为字符串“1”

#include <stdio.h>

int main()
{
    #ifdef DEBUG
        printf("DEBUG LOG\n");
    #endif
        printf("in\n");
}

// 1. 在编译的时候, 使用g++ -DDEBUG main.cpp
// 2. 第七行代码可以被执行
```

g++命令行编译

1. 链接静态库生成可执行文件

```
## 进入src目录下
$cd src

# 汇编，生成Swap.o文件
g++ Swap.cpp -c -I../include
# 生成静态库libSwap.a
ar rs libSwap.a Swap.o

## 回到上级目录
$cd ..

# 链接，生成可执行文件:staticmain
g++ main.cpp -Iinclude -Lsrc -lSwap -o staticmain
```

2. 链接动态库生成可执行文件

   -fPIC是一个表示与路径无关的选项

   链接**动态库**生成可执行文件②：

```
## 进入src目录下
$cd src

# 生成动态库libSwap.so
g++ Swap.cpp -I../include -fPIC -shared -o libSwap.so
## 上面命令等价于以下两条命令
# gcc Swap.cpp -I../include -c -fPIC
# gcc -shared -o libSwap.so Swap.o

## 回到上级目录
$cd ..

# 链接，生成可执行文件:sharemain
g++ main.cpp -Iinclude -Lsrc -lSwap -o sharemain
```

   动态库和静态库的区别是，连接到swap.cpp的时候，已经包含了静态库.a的文件，但是不包含.so动态库文件，需要现场调用

   直接运行使用动态库的可执行文件会失败，需要指定动态库文件所在目录（在默认路径中是不需要指定的），这就是静态库和动态库的不同

3.3.3 运行可执行文件

运行可执行文件①

```
# 运行可执行文件
./staticmain
```

运行可执行文件②

```
# 运行可执行文件
LD_LIBRARY_PATH=src ./sharemain
```

GDB调试器

前言

- GDB(GNU Debugger)是一个用来调试c++/c程序的功能强大的**调试器**，是linux系统开发c/c++最常用的调试器
- 程序员可以使用GDB跟踪程序中的错误，减少工作量
- VSCode是通过调用GDB调试器实现调试工作的
- windows系统中，常见的IDE内部已经嵌套了相应的调试器

GDB主要功能

- 设置断点
- 是程序在指定的代码行上暂停执行，便于观察
- 单步执行程序，便于调试
- 查看程序中变量值的变化
- 动态改变程序的执行环境
- 分析崩溃程序产生的core文件

常用调试命令参数

调试开始：执行 gdb exefilename,进入gdb调试程序，exefilename为要调试的可执行文件名

```
##以下命令后括号内为命令的简化使用，比如run(r),输入r就可以代替run

$(gdb)help(h) #查看命令帮助 help+命令

$(gdb)run(r) #重新开始运行文件（run-test:加载文本文件 run-bin:加载二进制文件）

$(gdb)start #单步执行，运行程序，停在第一行执行语句

$(gdb)list(l) #查看原代码(list-n, 从第n行开始查看代码。 list+函数名：查看具体函数)

$(gdb)set #设置变量的值

$(gdb)next(n) #单步调试，逐过程，函数直接执行

$(gdb)step(s) #单步调试，逐语句，跳入自定义函数内部执行

$(gdb)backtrace(bt) #查看函数的调用的栈帧和层级关系

$(gdb)frame(f) #切换函数的栈帧

$(gdb)info(i) #查看函数内部局部变量的数值

$(gdb)finish #结束当前函数，返回函数调用点

$(gdb)continue(c) #继续运行

$(gdb)print(p) #打印值及地址

$(gdb)quit(q) #退出gdb

$(gdb)break+num(b) #在第num行设置断点

$(gdb)info breakpoints #查看当前设置的所有断点

$(gdb)delete breakpoints num(d) #删除第num个断点

$(gdb)display #追踪查看具体变量的值
```

```
$(gdb)undisplay #取消追踪查看具体变量的值

$(gdb)watch #被设置观察点的变量发生修改时，打印显示

$(gdb)i watch #显示观察点

$(gdb)enable breakpoints#启用断点

$(gdb)disable breakpoints #禁用断点
```

注意：编译程序时需要加上-g,才能用gdb调试：g++ -g main.cpp -o main

回车键：重复上一命令

:set nu 显示行号

ctrl l 清屏

vscode快捷键

高频使用快捷键：

功能	快捷键	功能	快捷键
转到文件 / 其他常用操作	`Ctrl + P`	关闭当前文件	`` `Ctrl + W` ``
打开命令面板	`Ctrl + Shift + P`	当前行上移/下移	`Alt + Up/Down`
打开终端	`Ctrl +` `	变量统一重命名	`F2`
关闭侧边栏	`Ctrl + B`	转到定义处	`F12`
复制文本	`Ctrl+C`	粘贴文本	`Ctrl+V`
保存文件	`Ctrl+S`	撤销操作	`Ctrl+Z`

CMake

cmake是一个跨平台的安装编译工具，可以用简单的语句描述所有平台的安装（编译过程）

语法特性介绍

- 基本语法格式：指令（参数1，参数2...）
  - 参数使用**括弧**括起
  - 参数之间使用**空格**或**分号**分开
- 指令是大小写无关的，参数和变量是大小写相关的
- **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最小版本要求
  - 语法 cmake_minimum_required(VERSION versionNumber [FATAL_ERROR])

```
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])

```
project(HELLOWORLD)
```

- set-显式的定义变量
  - 语法 set(VAR [VALUE] [CACHE TYPE DOCATRING [FORCE]])

```
#定义SRC变量，其值为sayhello.cpp hello.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_directoriea(/usr/lib/mylibfolder ./lib)
```

- add_library-生成库文件
  - 语法 add_library(libname [SHARED|STATIC|MODULE] [EXCLUDE_FROM_ALL] source1 source2)

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

- add_compile_options-添加编译参数
  - 语法 add_compile_options(

```
#添加编译参数 -Wall -std=c++11 -O2
add_compile_options(-Wall -std=c++11 -O2)
```

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

```
add_executable(main main.cpp)
```

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

```
#将hello动态文件库链接到可执行文件main
add_executable(main hello)
```

- add_subdirectpry-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])

```
#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})
```

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
  -

```
#设定编译类型为debug,调试时选择debug
set(CMAKE_BUILD_TYPE Debug)
#设定编译类型为release,发布时需要release
set(CMAKE_BUILD_TYPE Release)
```

- - **CMAKE_BINARY_DIR**

    PROJECT_BINARY_DIR

    <projectname>_BINARY_DIR

    > 1. 这三个变量指代的内容是一致的。
    > 2. 如果是 in source build，指的就是工程顶层目录。
    > 3. 如果是 out-of-source 编译,指的是工程编译发生的目录。
    > 4. PROJECT_BINARY_DIR 跟其他指令稍有区别，不过现在，你可以理解为他们是一致的。

  - **CMAKE_SOURCE_DIR**

    PROJECT_SOURCE_DIR

    <projectname>_SOURCE_DIR

    > 1. 这三个变量指代的内容是一致的,不论采用何种编译方式,都是工程顶层目录。
    > 2. 也就是在 in source build时,他跟 CMAKE_BINARY_DIR 等变量一致。
    > 3. PROJECT_SOURCE_DIR 跟其他指令稍有区别,现在,你可以理解为他们是一致的。

- **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中有CMakeLists.txt文件，主目录的CMakeLists.txt通过add_directory添加子目录即可
2. 包含源文件的子文件夹中没有CMakeLists.txt文件，子目录的编译规则体现在主目录的CMakeLists.txt中

编译流程

1. 手动编写CMakeLists.txt
2. 手动执行命令cmake PATH生成Makefile(PATH是顶层CMakeLists.txt所在目录)
3. 执行make命令进行编译

CMake实践

基于VSCode和CMake实现C/C++开发 | Linux篇*哔哩哔哩*bilibili

使用VScode进行完整项目开发

基于VSCode和CMake实现C/C++开发 | Linux篇*哔哩哔哩*bilibili