wsl Ubuntu相关

  Bourne Again SHell, 简称 “bash”

shell是最基本的文字接口

shell 基于空格分割命令并进行解析，然后执行第一个单词代表的程序，并将后续的单词作为程序可以访问的参数

大多数的命令接受标记和选项（带有值的标记），它们以 - 开头，并可以改变程序的行为。通常，在执行程序时使用 -h 或 --help 标记可以打印帮助信息，以便了解有哪些可用的标记或选项。

shell命令

grep  文本搜索

pwd 显示当前位置

cd 跳转，“..”为上一层目录，“.”为这一层目录

ls 显示当前目录下的文件 -all显示所有文档和信息，-l显示额外信息

mv用于重命名或移动文件

rm移除文件，-r可移除文件夹

cp 拷贝文件

mkdir新建文件夹

touch 创建一个文件

man 查看指令功能

通过“↑”，“↓”可以查看历史指令并修改使用

Vim

Vim 是一个 多模态 编辑 器：它对于插入文字和操纵文字有不同的模式。

在正常模式，键入 i 进入插入模式。这样可以跟很多其他的编辑器一样修改内容，直到使用<ESC> 返回正常模式

在正常模式下键入 : 进入命令行模式。 在键入 : 后，你的光标会立即跳到屏幕下方的命令行。 这个模式有很多功能，包括打开，保存，关闭文件，以及退出。

:q 退出（关闭窗口）

:w 保存（写）

:wq 保存然后退出git

git版本控制

git init 创建git仓库

git add 工作区→暂存区（index）

git commit 暂存区→仓库（respository）

git commit -m “注释”

git status 查看文件状态

git log 查看日志

git log --pretty=oneline --abbrev-commit --all --graph

git reset --hard commitID(git log查看) 版本回退

--hard→连着暂存部分一起删掉

--soft →保留暂存部分，只删掉提交的

git diff 找不同

git remote add <name>(origin) <https://........连接远程仓库>

git remote rename <oldname> <newname>

git push origin <分支> 上传一个分支

git rebase <branch>（变基，搬家）

git stash 储存修改

git stash apply 回到储存的地方

git stash apply stash@{某一次存储}

C语言编译流程

C 语言的编译流程是将源代码（.c 文件）转换为可执行文件的过程。这个过程通常分为四个主要阶段：预处理、编译、汇编 和 链接。以下是每个阶段的详细说明：

1. 预处理

作用：处理源代码中的预处理指令（以 # 开头的指令），生成一个纯 C 代码文件。

主要任务：

展开头文件（#include）：将头文件的内容插入到源文件中。

宏替换（#define）：将宏定义替换为实际值。

条件编译（#if、#ifdef 等）：根据条件保留或删除代码。

删除注释：去掉源代码中的注释。

输入：.c 文件（源代码）。输出：.i 文件（预处理后的代码）。

2. 编译

作用：将预处理后的 C 代码转换为汇编代码。

主要任务：

语法检查：检查代码是否符合 C 语言语法规则。

生成汇编代码：将 C 代码转换为目标平台的汇编代码。

输入：.i 文件（预处理后的代码）。输出：.s 文件（汇编代码）。

3. 汇编

作用：将汇编代码转换为机器代码（目标文件）。

主要任务：

* 将汇编指令转换为机器指令。
* 生成目标文件（.o 文件），包含机器代码和符号表。

输入：.s 文件（汇编代码）。输出：.o 文件（目标文件）。

4. 链接

作用：将多个目标文件和库文件合并，生成最终的可执行文件。

主要任务：

解析符号引用：将目标文件中的未定义符号（如函数调用）与库文件或其他目标文件中的定义关联。

合并代码和数据：将所有目标文件和库文件中的代码和数据段合并。

生成可执行文件：生成一个可以在操作系统上运行的可执行文件。

输入：.o 文件（目标文件）和库文件（如 libc.a 或 libc.so）。

输出：可执行文件（如 a.out 或自定义名称）。

常用 GCC 编译选项

-E 只进行预处理，生成 .i 文件。

-S 只进行编译，生成 .s 文件。

-c 只进行汇编，生成 .o 文件。

-o 指定输出文件名。

-g 生成调试信息，用于调试工具（如 GDB）。

-Wall 启用所有警告信息。

-O 优化级别（如 -O1、-O2、-O3）。

-I 指定头文件搜索路径。

-L 指定库文件搜索路径。

-l 链接指定的库（如 -lm 链接数学库）。

MAKE工具

make是一条计算机指令，是在安装有GNU Make的计算机上的可执行指令。该指令是读入一个名为Makefile的文件，然后按一定规则执行这个文件中指定的指令。

Make 使最终用户可以在不知道构建细节的情况下构建和安装软件，因为这些细节都记录在提供的Makefile中。

基于源文件的改动，Make可以自动知道需要更新的软件；它也会自动决定文件更新的适当顺序，以避免要更新的文件依赖于另一个同样需要更新的文件。因此，在修改了程序的源代码并且执行Make后，不必重新完全编译所有文件，只需要重新编译直接或间接受到影响的文件就好了。

规则目标

Makefile告诉Make怎样执行一系列的指令去依靠源文件生成一个目标文件。Makefile中声明了一个依赖关系的列表，这个列表应当包含所有文件（无论是源文件或者目标）作为输入。以下是个简单的规则示例：target:   dependency1 dependency2

command1

command2

target：目标

denpendency：依赖

command：（编译）命令

dependencies中如果有一个以上的文件比target文件要新的话，command所定义的命令就会被执行。

targets是文件名，以空格分开，可以使用通配符。一般来说，我们的目标基本上是一个文件，但也有可能是多个文件。

command是命令行，不与“target:prerequisites”在一行，必须以 Tab 键开头。

make默认以第一个目标为最终目标。如果一个目标文件比他的所有依赖都新，这样他就是最新的了，不需要再去重新生成了，否则需要更新，要按照正确的顺序，在用来生成其他文件之前先更新自己。

makefile定义了一系列的规则来指定，哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为makefile就像一个Shell脚本一样，其中也可以执行操作系统的命令。

makefile带来的好处就是——“自动化编译”，一旦写好，只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发的效率。

一般来说，无论是C还是C++，首先要把源文件编译成中间代码文件，在Windows下也就是 .obj 文件，UNIX下是 .o 文件，即Object File，这个动作叫做编译（compile）。然后再把大量的Object File合成可执行文件，这个动作叫作链接（link）。

Makefile里主要包含了五个东西：显式规则、隐式规则、变量定义、指令和注释。

1. 显式规则。显式规则说明了如何生成一个或多个目标文件。这是由Makefile的书写者明显指出要生成的文件、文件的依赖文件和生成的命令。
2. 隐式规则。由于我们的make有自动推导的功能，所以隐式规则可以让我们比较简略地书写Makefile，这是由make所支持的。
3. 变量的定义。在Makefile中我们要定义一系列的变量，变量一般都是字符串，这个有点像你C语言中的宏定义，当Makefile被执行时，其中的变量都会被扩展到相应的引用位置上。
4. 指令。其包括了三个部分，一个是在一个Makefile中引用另一个Makefile，就像C语言中的include一样；另一个是指根据某些情况指定Makefile中的有效部分，就像C语言中的预编译#if一样；还有就是定义一个多行的命令。
5. 注释。Makefile中只有行注释，和UNIX的Shell脚本一样，其注释是用 # 字符，这个就像C/C++中的 // 一样。如果你要在你的Makefile中使用 # 字符，可以用反斜杠进行转义，如： \# 。

文件名最好用Makefile

make的工作方式

GNU的make工作时的执行步骤如下：

1. 读入所有的Makefile。
2. 读入被include的其它Makefile。
3. 初始化文件中的变量。
4. 推导隐式规则，并分析所有规则。
5. 为所有的目标文件创建依赖关系链。
6. 根据依赖关系，决定哪些目标要重新生成。
7. 执行生成命令。

一些C语言的函数

strcmp 是 C 标准库中的一个函数，用于比较两个字符串。它的原型如下：

int strcmp(const char \*str1, const char \*str2);

str1：第一个字符串。

str2：第二个字符串。

如果 str1 和 str2 相等，返回 0。

如果 str1 小于 str2，返回负数。

如果 str1 大于 str2，返回正数。

system(“<cmd>”)；输出一个命令

fopen() 用于打开文件，并返回一个指向文件的指针（FILE\*）。

FILE \*fopen(const char \*filename, const char \*mode);

filename：要打开的文件名（包括路径）。

mode：文件打开模式，常见的模式有：

"r"：只读模式（文件必须存在）。

"w"：写入模式（如果文件存在，内容会被清空；如果文件不存在，会创建文件）。

成功：返回一个指向文件的指针（FILE\*）。

失败：返回 NULL，并设置 errno。

fgets() 用于从文件中读取一行内容。

char \*fgets(char \*str, int n, FILE \*stream);

str：用于存储读取内容的缓冲区。

n：缓冲区的大小（最多读取 n-1 个字符）。

stream：指向文件的指针（通常由 fopen() 返回）。

成功：返回 str。

失败或到达文件末尾：返回 NULL

isspace();检查是否是空白字符

strchr(line, '#') 字符串 line 中查找字符 # 的第一次出现位置。

strtok函数

char \*strtok(char \*str, const char \*delim);

str‌：待分割的字符串。如果是第一次调用，传入要分割的字符串；之后的调用应传入NULL，以继续上一次的分割。

delim‌：包含所有分隔符的字符集合。函数会在这些字符处分割字符串。

‌初次调用‌：传入待分割的字符串str，函数会从str中找到第一个部分并返回指向该部分的指针。函数会用'\0'替换找到的分隔符，从而破坏原字符串。

后续调用‌：传入NULL，函数会继续从上次结束的位置查找下一部分并返回指向该部分的指针。如果找不到更多的分隔符，返回NULL。

strncpy函数用于将指定长度的字符串复制到字符数组中。

char \*strncpy(char \*dest, const char \*src, int n)，表示把src所指向的字符串中以src地址开始的前n个字节复制到dest所指的数组中，并返回被复制后的dest。

file\_exists(rule->dependencies[i])==0 )

图算法

图（Graph）一种非线性数据结构

图由 顶点和 边组成，通常表示为 G=(V,E)，V是顶点，E是边

图的表示方法：

邻接矩阵（Adjacency Matrix）：用二维数组表示顶点之间的连接关系。

邻接表（Adjacency List）：用链表或数组的数组表示每个顶点的邻居。

图的遍历是图算法的基础，用于访问图中的所有顶点。

深度优先搜索（DFS, Depth-First Search）：

从某个顶点出发，沿着一条路径尽可能深地访问顶点，直到无法继续为止，然后回溯。

实现：递归或栈。

广度优先搜索（BFS, Breadth-First Search）：

从某个顶点出发，逐层访问其邻居。

实现：队列。

拓扑排序（Topological Sorting）

用于有向无环图（DAG, Directed Acyclic Graph），将顶点排成线性序列，使得所有有向边从前往后指向。

实现：DFS 或 BFS（Kahn 算法）。

强连通分量（SCC, Strongly Connected Components）

用于有向图，将图分解为强连通子图。

构建依赖图的关键步骤：

定义顶点和边：

每个任务或事件是一个顶点。

如果任务 A 依赖于任务 B，则添加一条从 B 到 A 的有向边。

检测环：

使用拓扑排序或 DFS 检测图中是否存在环。

如果存在环，则依赖关系无法满足，需要调整。

拓扑排序：

对依赖图进行拓扑排序，得到任务的执行顺序。

如果图中存在环，则无法完成拓扑排序。

调度任务：

根据拓扑排序的结果，按顺序执行任务。

拓扑排序（Topological Sorting） 是一种针对有向无环图（DAG, Directed Acyclic Graph）的线性排序算法。它的核心思想是将图中的顶点排成一个序列，使得对于图中的每一条有向边 (u,v)，顶点 u 在序列中总是位于顶点 v 的前面。拓扑排序常用于解决依赖关系问题，例如任务调度、编译顺序等。

1. 计算每个顶点的入度（即有多少边指向该顶点）。
2. 将所有入度为 0 的顶点加入队列。
3. 从队列中取出一个顶点，将其加入结果列表，并将其所有邻居的入度减 1。
4. 如果某个邻居的入度变为 0，则将其加入队列。
5. 重复上述过程，直到队列为空。
6. 如果结果列表中的顶点数不等于图中的顶点数，则说明图中存在环。

图的入度（In-degree） 是图论中的一个重要概念，专门用于描述有向图（Directed Graph）中顶点的性质。它表示有多少条边指向该顶点。

文件操作

stat 函数

#include <sys/types.h>

#include <sys/stat.h>

#include <unistd.h>

int stat(const char \*pathname, struct stat \*statbuf);

pathname：文件的路径名。

statbuf：指向 struct stat 结构体的指针，用于存储文件的状态信息。

成功时返回 0。

失败时返回 -1，并设置 errno 以指示错误。

struct stat 结构体包含文件的状态信息

st\_mode mode\_t 文件类型和权限

st\_size off\_t 文件大小（字节）

st\_mtime time\_t 文件最后修改时间

st\_atime time\_t 文件最后访问时间

st\_ctime time\_t 文件状态最后更改时间

time\_t 是 C 语言中用于表示时间的数据类型，通常用于存储从某个固定时间点（通常是 1970 年 1 月 1 日 00:00:00 UTC，称为 Unix 纪元）到当前时间的秒数。它是标准库 <time.h> 中定义的类型，广泛用于时间相关的操作。

进程相关

#include <unistd.h>

fork()：

创建一个子进程。

创建成功时在子进程中返回0，在父进程中返回PID

pid < 0：创建子进程失败。

pid == 0：当前为子进程。

pid > 0：当前为父进程，返回子进程的 PID。

fork() 会创建一个与父进程几乎完全相同的子进程。

子进程复制父进程的地址空间、堆、栈、文件描述符等资源，从 fork() 的返回点开始执行，而不是从头开始执行程序。

父进程调用 wait() 或 waitpid() 等待子进程。

子进程的 getppid() 可以返回父进程的 PID。

父子进程可以通过管道、共享内存、信号等机制进行通信。

execvp()：

在子进程中执行命令。

args[0]：要执行的程序（/bin/sh）。

args：参数数组，最后一个元素必须为 NULL。

如果 execvp() 失败，会输出错误信息并退出子进程。

file：要执行的程序文件名（不需要完整路径，会在 PATH 环境变量中查找）。

argv：传递给新程序的参数列表，类似于 main 函数的 argv。

argv[0] 通常是程序名。

最后一个元素必须是 NULL。

成功时不会返回（因为当前进程的地址空间被替换）。

失败时返回 -1，并设置 errno。

waitpid(pid\_t pid, int \*status, int options)：

父进程等待子进程结束。

pid：子进程的 PID。

status：用于存储子进程的退出状态。

&status：存储子进程的退出状态。

options：控制 waitpid() 的行为，常用选项包括：

0：等待与调用进程同组的任意子进程。

如果子进程正常退出，返回子进程的退出状态。

如果子进程异常退出，返回 -1。

WIFEXITED(status)：检查子进程是否正常退出。

WEXITSTATUS(status)：获取子进程的退出状态。

WEXITSTATUS() 是一个宏，用于从退出状态中提取子进程的退出码。

它只能用于正常退出的子进程（即通过 WIFEXITED() 检查为真）。

int WEXITSTATUS(int status);

status：子进程的退出状态，通常由 wait() 或 waitpid() 返回。

子进程的退出码（0 到 255）。