

命令

Vim

gcc 编译器 生成代码 调试库 编辑 makefile

系统编程 ~~☆☆☆~~

文件系统 ① 文件属性
② 文件内容

① 特点 资源

进程 ② 多进程

③ 一切皆文件
~~☆~~ ④ I/O 多路复用 → 非阻塞 进程间通信

管道

信号

~~线程~~ 线程 多线程
互斥和同步

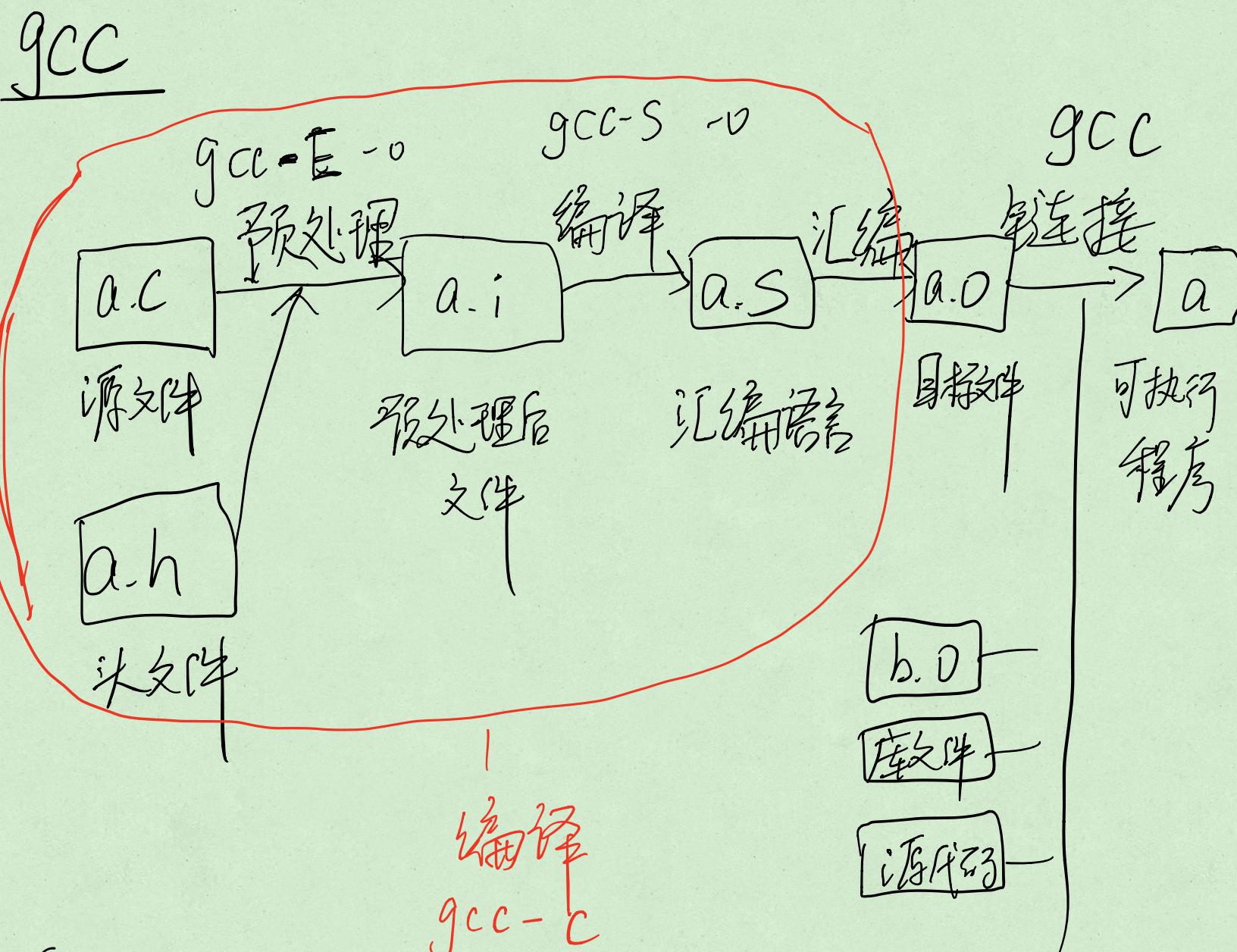
~~网络~~

① 网络概念 ③ epoll

② 编程

服务器端结构 → 进程池和线程池 SQL语句

项目 - 网盘



①

push pop call ret
mov sub lea 及地址
8 byte 4 byte
g L

xor mov jmp je jle

② 所有变量 → 地址 + 长度

起始、结束

③ 循环 和 goto 本质一样

④ 函数调用 程帧独立

汇编

AS - the portable GNU assembler

汇编

as test.c -o test.o

nm nm test.o 3. 使用了文件所有
符号

gcc -c ll.c 文件直接生成.o文件

objdump 反汇编

链接

把调用函数的名字 → 地址

ld (直接调用)

gcc (间接调用) → 函数 / 全局变量 定义 // 越过

gcc test.o -o test

2 次
全局变量

错误

执行可执行程序

./文件名

库文件

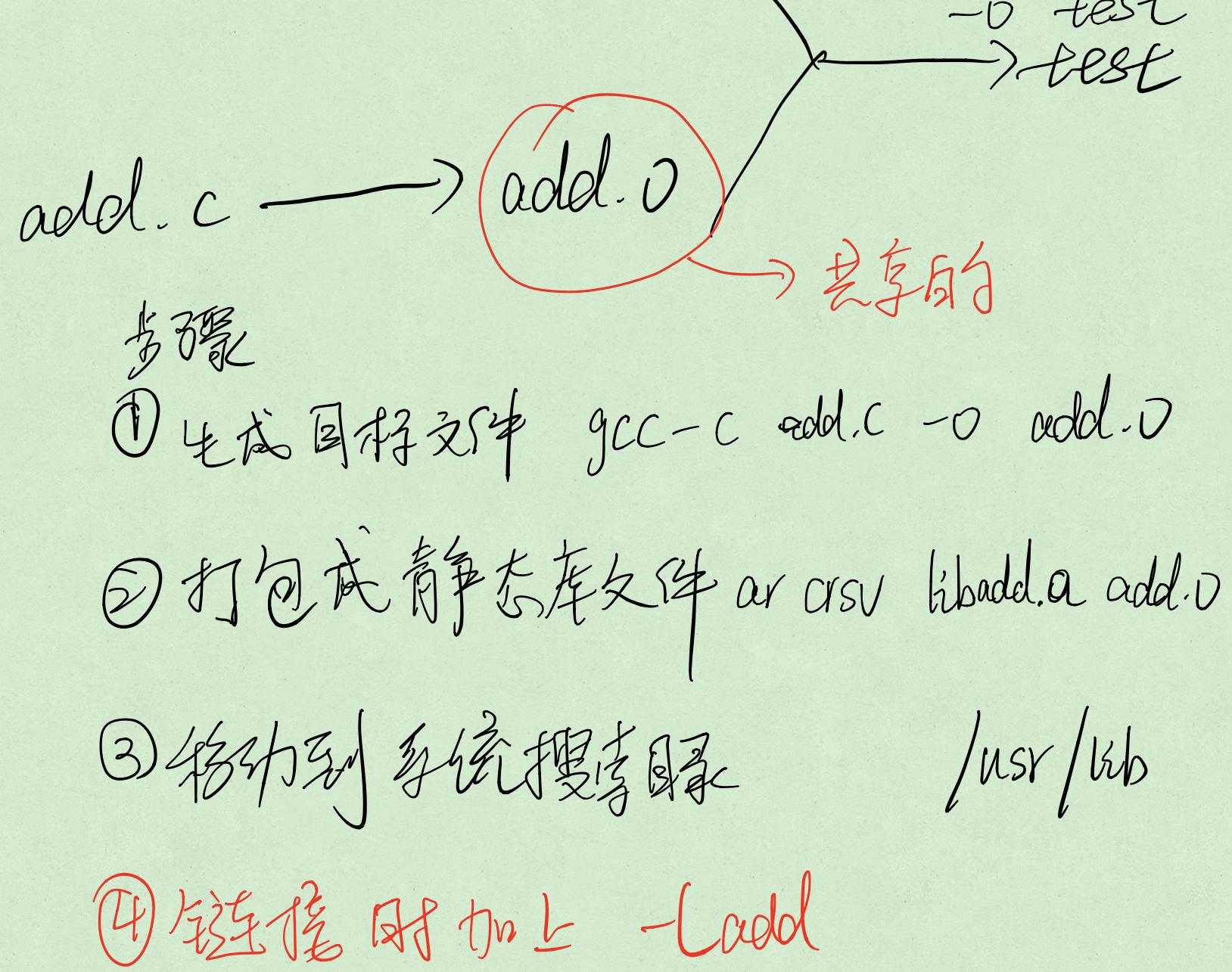
老子 公用的工具 特殊的.D文件

静态库和动态库

特点：	产品大	部署 容易	升级 困难
静态	大	容易	困难
动态	小	难	容易

生成静态库

test.c → test.o
gcc test.o add.o



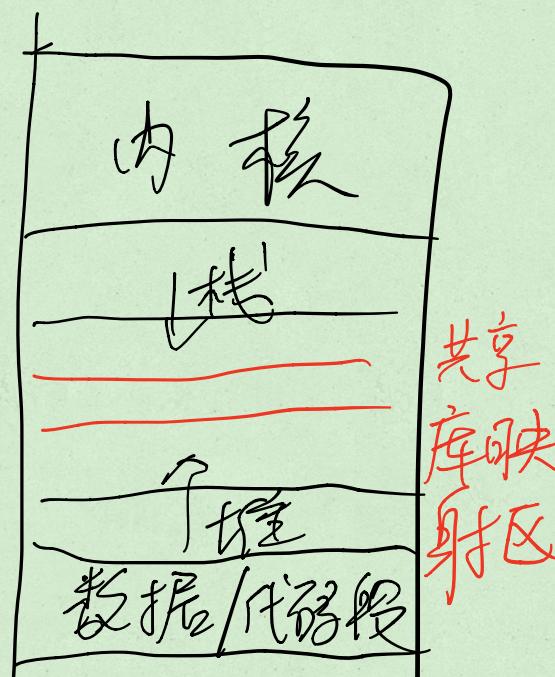
生成动态库

- ① 编译成目标文件

相对地址 → 位置无关代码

`加上`

`-fpic`



\$ gcc -c add.c -o add.o -fPIC

② 打包

\$ gcc -shared add.o -o libadd.so

③ 移动到系统库目录

\$ sudo cp libadd.so /usr/lib

④ 链接时加上 -Ladd

\$ gcc test.o -o test -Ladd

解决升级问题

libadd.so.0.1

↓ 版级

/usr/lib/libadd.so

gcc 其他选项

gcc test.c -D
相当于#define

gcc src/test.c -I include
增加搜索路径

- O₀ 编译优化 编译器作者修改指针和内存位置
不优化 ①结果不变
- O₁ → 产品 ②指令数量变少
- O₂ → 开源 执行速度变快
- O₃ ↓ 优化地越深 C与汇编的对应就乱了

编译警告 不允许出现警告

Warning

-wall

开警告

gdb

gdb - GNU Debugger

不带优化 - O₀
补充调试信息 - g } 编译时加上

gdb 文件名

gdb 命令

list / l 看文件内容
[文件名] [行号] [函数名]

run/r 运行

break/b [文件名] [行号] [函数名] 打断点

continue/c 继续运行

step/s VS F11

next/n VS F10

finish 跳出本次函数调用

info break/i b 查看断点信息

delete [num] 删/除断点(不加参数删所有)

ignore [num] [count] 忽略num断点count次

在 gdb 中查看监视

print/p 表达式

display 表达式

先 info display
再 undisplay [编辑]

在 gdb 中查看内存

x/

数据	字母	字母	每个单位的大小	ADDRESS
b	n	w	1b	
b	n	w	2b	
b	n	w	4b	
b	n	w	8b	

数字 字母 字母

多少单位

八进

十六进

十进

无符号十进

检查崩溃的内容

CORE 文件 (程序崩溃时刻内存的堆栈)

段错误 Segmentation fault

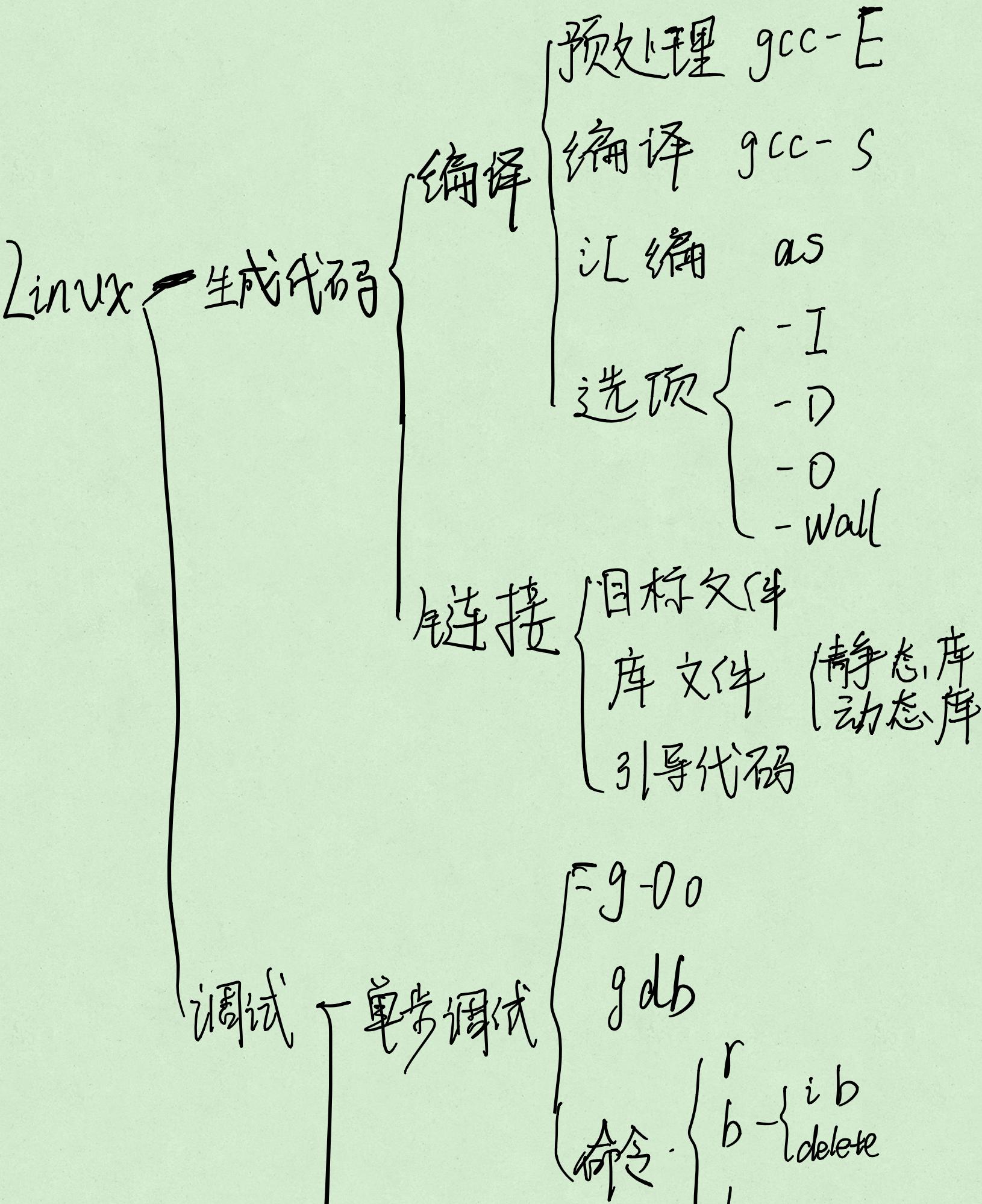
使用 core

- ① 编译加上 -g -O0
- ② ulimit -c unlimited
- ③ 执行程序
- ④ gdb 可执行 core

gdb 加命令行参数

set args

show args



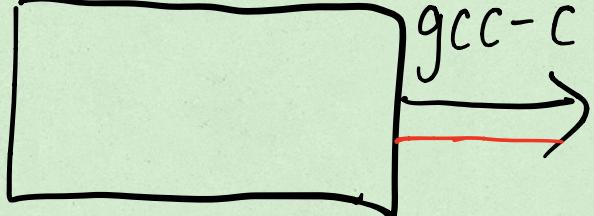
P
display
c
.n S
set args

出错调试

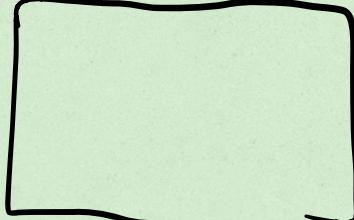
-g -O₀
ulimit -c unlimited
运行生成CORE文件
用gdb 打开core
使用bt命令
查看调用堆栈

增量编译

main.c



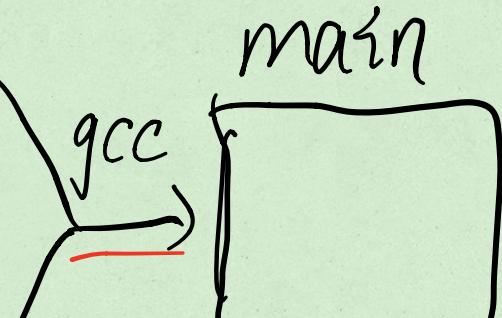
main.o



add.c

add.o

main



gcc -c

脚本

gcc -c main.c

gcc -c add.c

gcc main.o add.o -o main

makefile 增量编译生成代码

一种“目标-依赖”只有目标不存在或目标比依赖旧
才执行命令

makefile 实现

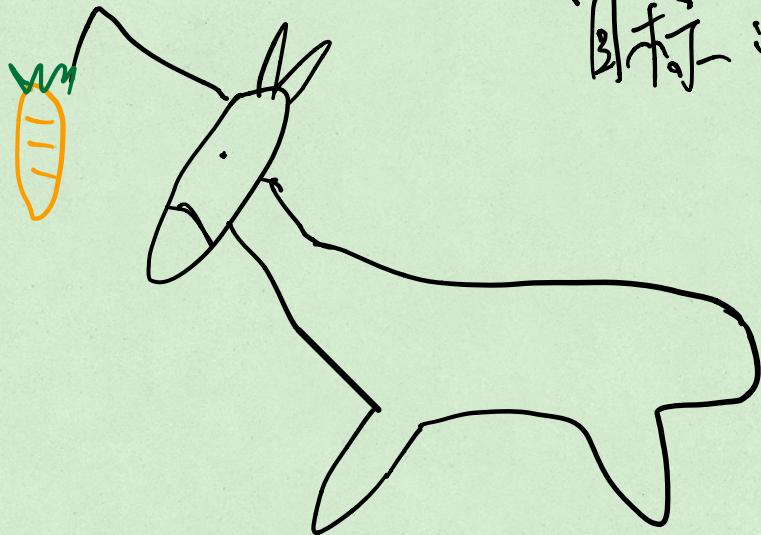
① 名字必须是 Makefile / makefile

② 规则集合

1个	0~多个
目标文件 : 依赖文件	
<tab>	命令 (0~多个)

③ 把最终生成文件 作为第一个规则的目标

伪目标



目标：依赖

命令

① 目标不存在

② 执行命令

生成不了目标

.PHONY: --

每次 make 都一定
执行的指令

变量

① 自定义变量

变量名 := 值 < 字符串

引用变量中(变量名)

② 预定义变量 CC RM

③ 自动变量 同一变量名 遵循规则而变

\$@ 目标文件 \$< 第一个依赖文件 \$^> 所有依赖
-C \$^ -D \$@

用%字符管理格式关系

%.*o* %.*c* 按格式从上一个规则的依赖去匹配

内置函数

Wildcard 通配符

从当前目录所有文件中

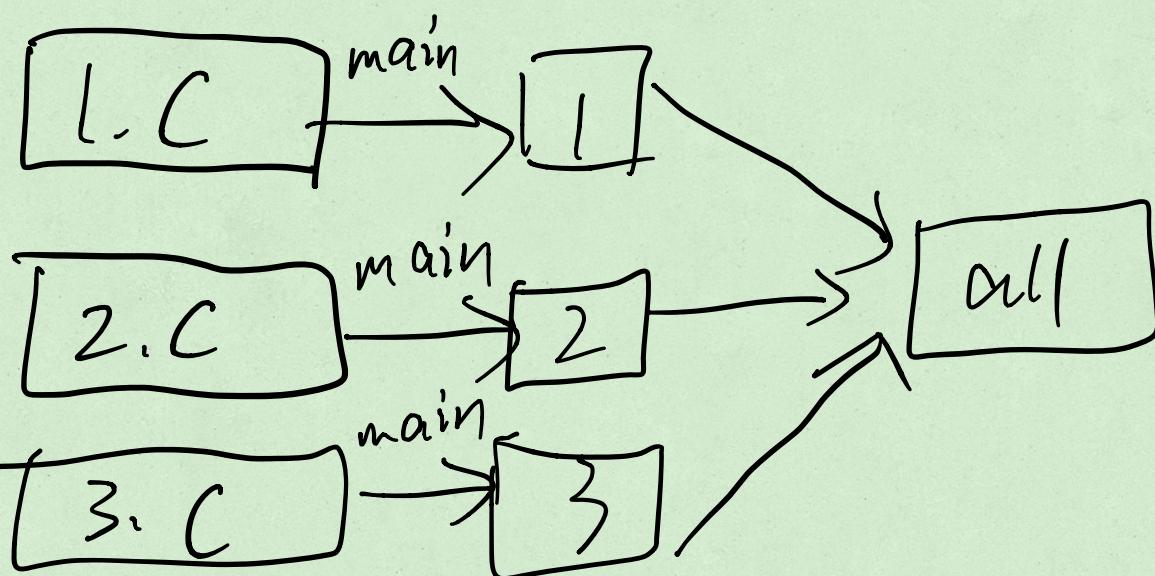
取出 符合要求

pat subst

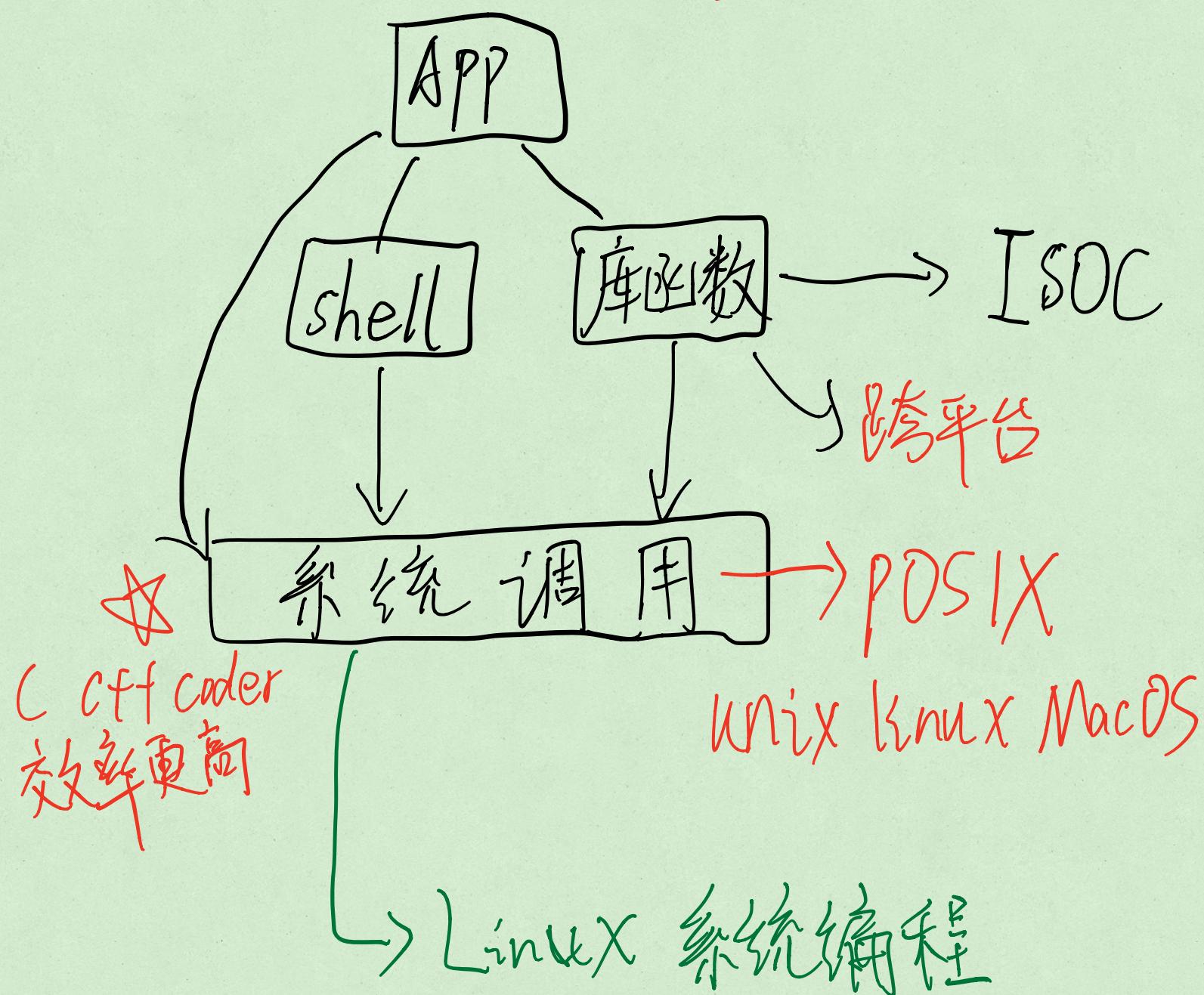
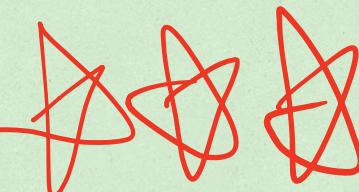
↓

pattern substitute

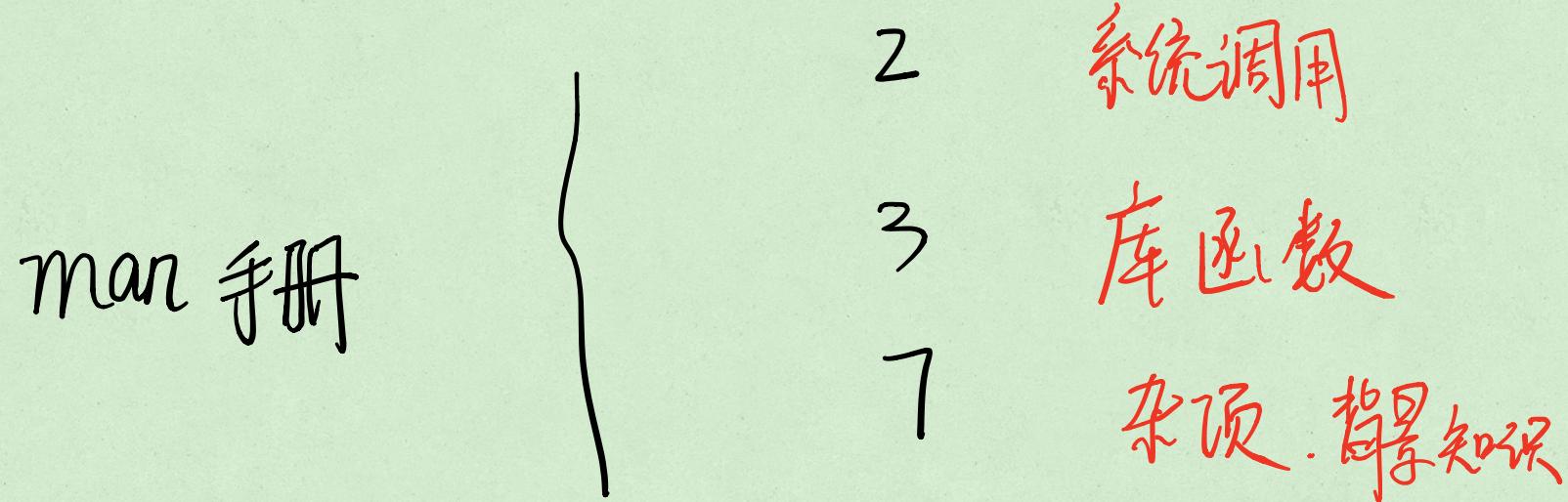
单独编译连接



Linux系统编程



如何学习系统编程



名字 → 声明 → 细节 → 返回值

① 先阅读名字

② 看声明和返回值

a. 头文件

b. 指针类型的参数 → 主调函数

const 不会修改 传入参数

无const 会 传入传出参数

c. 指针类型的返回值 → 主调是否要释放

d. 返回值不实现功能,

只用来处理报错信息

③ 细节 按需查看

文件

万物皆文件

文件类型

普通文件 目录文件 软连接

字符设备文件 鼠标

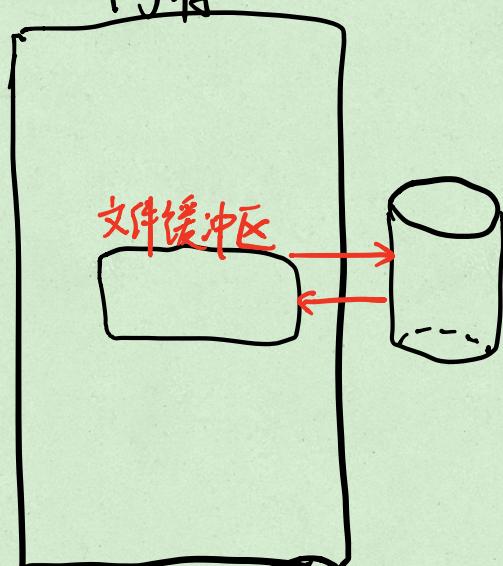
块设备文件 磁盘

管道文件 通信

socket

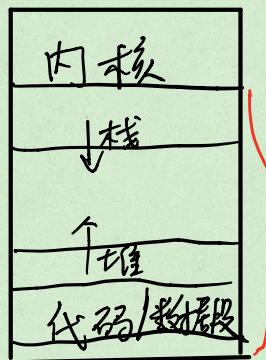
网络通信

文件使用



fopen

库函数



FILE

文件流 / 用户态文件缓冲区

追加模式

"a" append 只写追加——默认从文件结尾写入

"a+"

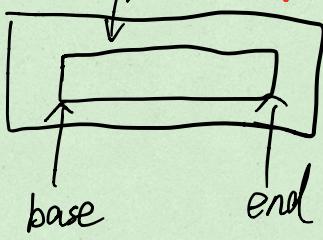
读写追加

打开时处于文件开头

日志系统

每次读写自动后移

写入时，跳到文件末尾

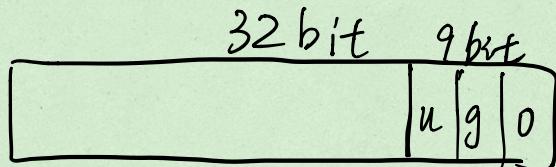


fread

fwrite

fseek

改变文件的权限



mode_t