shell上批量替换

$sed "s/int/float/g" 文件名

只显示修改后的结果，不修改实际内容

-i 修改实际内容

文件对比

vimdiff

vim进入时直接从某一行开始

vim +[行号/字符串] 文件名

配置文件

.vimrc running command

.bashrc shell的配置

```
export PS1="\[\e[37;40m\][\[\e[32;40m\]\u\[\e[37;40m\]@\h
\[\e[36;40m\]\w\[\e[0m\]]\\$ "
```

vimtutor

$ vimtutor　　训练模式

编译工具

gcc

clang + llvm

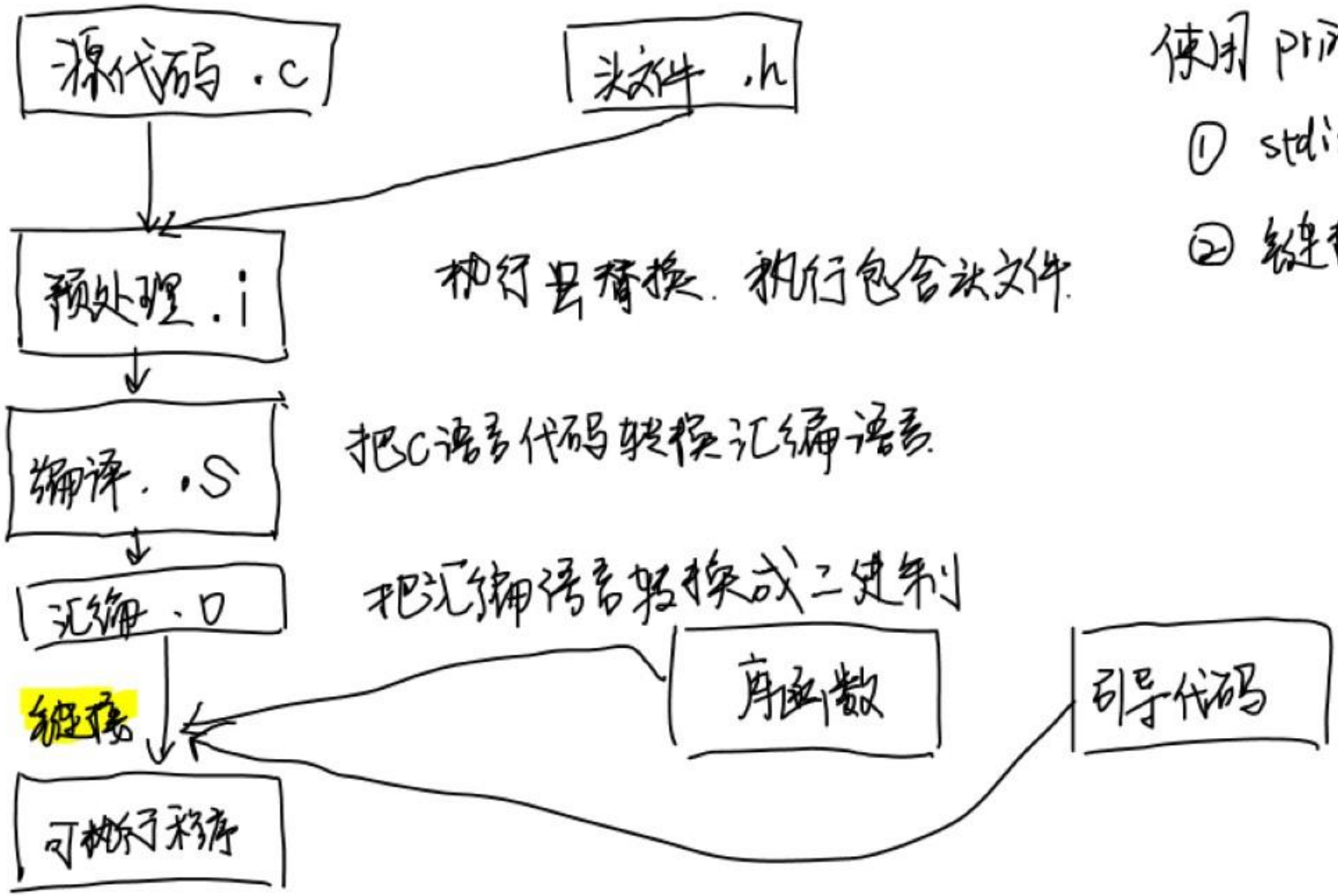
~~工具链~~ SDK

$gcc -v

gcc .c

g++ .c / .cc

程序的编译过程



使用 printf

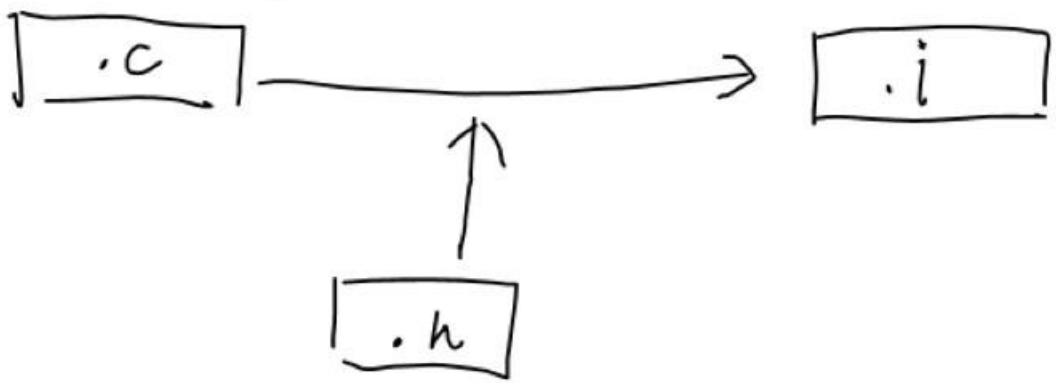
① stdio.h 放着 printf 的函数声明

② 链接时，要找到库文件

预处理

#include
#define ⟶ 展开

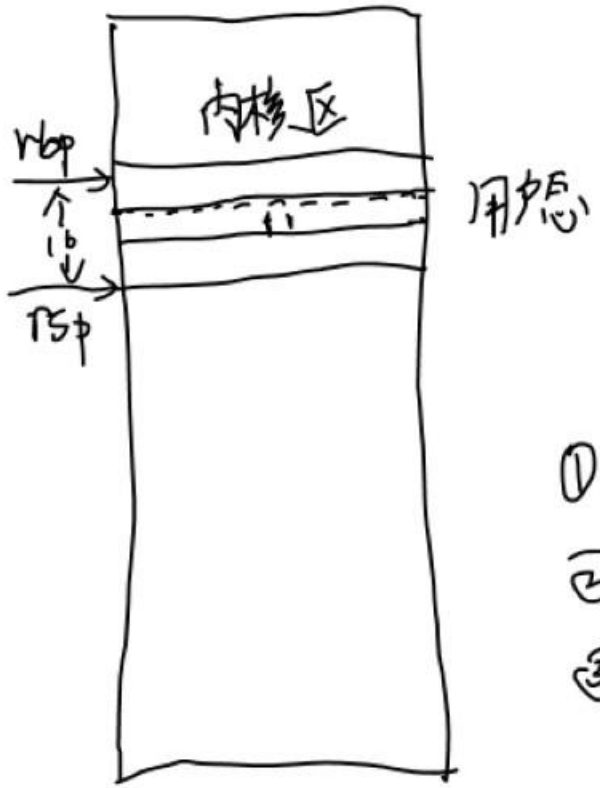
$gcc -E 源文件 -o 目标文件.

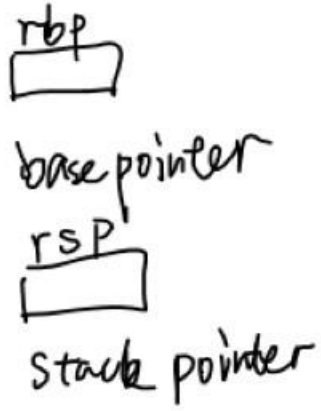


编译

找看得懂的地方阅读

$gcc -S 源文件 -o 目标文件.





① 函数首先分配栈帧
② 变量都会转成地址
③ 函数调用是使用其他函数的代码
④ 循环都用跳转实现

```
 1     .file   "main.c"
 2     .text
 3     .section    .rodata
 4 .LC0:
 5     .string "i = %d\n"
 6     .text
 7     .globl  main
 8     .type   main, @function
 9 main:
10 .LFB0:
11     .cfi_startproc
12     pushq   %rbp                    压栈
13     .cfi_def_cfa_offset 16
14     .cfi_offset 6, -16
15     movq    %rsp, %rbp              栈顶和栈底同样
16     .cfi_def_cfa_register 6
17     subq    $16, %rsp               分配了一个栈帧
18     movl    $11, -4(%rbp)           <- i
19     movl    -4(%rbp), %eax
20     movl    %eax, %esi
21     leaq    .LC0(%rip), %rdi
22     movl    $0, %eax
23     call    printf@PLT
24     movl    $0, %eax
25     leave
26     .cfi_def_cfa 7, 8
27     ret
28     .cfi_endproc
29 .LFE0:
```

```
    .file   "main.c"
    .text
    .section    .rodata
.LC0:
    .string "I am print j = %d\n"
    .text
    .globl  print
    .type   print, @function
print:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movl    -4(%rbp), %eax
    movl    %eax, %esi
    leaq    .LC0(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    nop
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE0:
    .size   print, .-print
    .section    .rodata
.LC1:
    .string "i = %d\n"
.LC2:
    .string "Hello world!"
    .text
    .globl  main
    .type   main, @function
main:
.LFB1:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $48, %rsp
    movq    %fs:40, %rax
    movq    %rax, -8(%rbp)
    xorl    %eax, %eax
    movl    $1, -20(%rbp)
    movl    $2, -16(%rbp)
    movl    $3, -12(%rbp)
    movl    $11, -40(%rbp)
    movl    -40(%rbp), %eax
    movl    %eax, %esi
    leaq    .LC1(%rip), %rdi
    movl    $0, %eax
    call    printf@PLT
    movl    $10, -36(%rbp)
    movl    -36(%rbp), %eax
    movl    %eax, %edi
    call    print
    movl    $4, -12(%rbp)
    leaq    -20(%rbp), %rax
    movq    %rax, -32(%rbp)
    movl    $0, -44(%rbp)
    jmp .L3
.L4:
    leaq    .LC2(%rip), %rax
    call    puts@PLT
    addl    $1, -44(%rbp)
.L3:
    cmpl    $4, -44(%rbp)
    jle .L4
    movl    $0, %eax
    movq    -8(%rbp), %rdx
    xorq    %fs:40, %rdx
    je  .L6
    call    __stack_chk_fail
.L6:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE1:
    .size   main, .-main
```

压栈

传递参数

压栈

常数赋值

取地址

汇编

.S ⟶ .O

$gcc 汇编文件 -o 目标文件

$nm 目标文件 (查看符号表)

```
[liao@ubuntu ~/test/gcc]$ nm main.o
                 U _GLOBAL_OFFSET_TABLE_
0000000000000024 T main
0000000000000000 T print
                 U printf
                 U puts
                 U __stack_chk_fail
```

地址值. 状态 待调用的函数.

U. 地址未确定

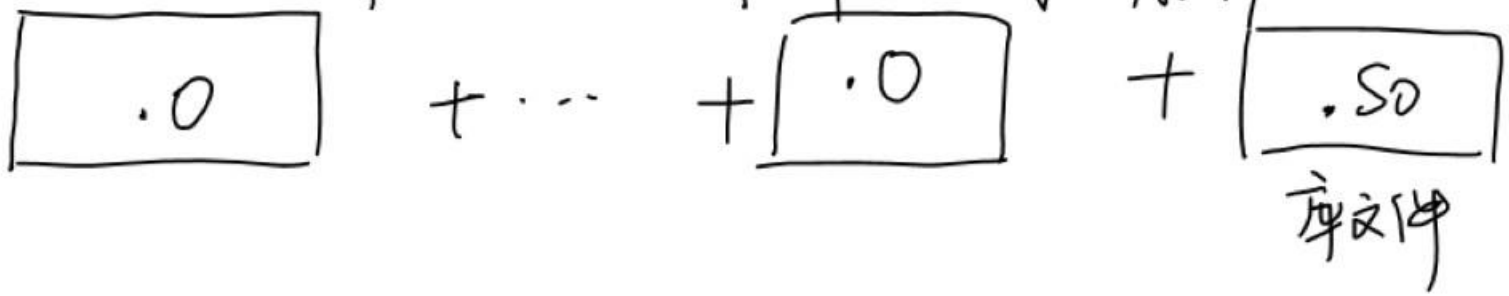
T. 本段内地址

链接

把.o文件的地址确定下来.

$gcc 目标文件 -o 可执行文件

$ld 目标文件 [其他依赖库文件] -o 可执行文件



./main 执行main程序

gcc的参数

参数	含义
-E	预处理 .i
-S	编译成汇编文件 .s
-o 目标	生成文件到目标里面
$as	汇编文件转换成二进制目标文件 .o
$nm	查看二进制的符号表
$ld	链接
-c	不需要使用-o 自动生成目标文件 .o

其他编译选项

编译选项	功能
-I 目录名	指定头文件的目录
-D 宏名	相当于在代码头部添加了一个 #define 宏
-Wall	打开编译警告
-O[123]	控制优化级别

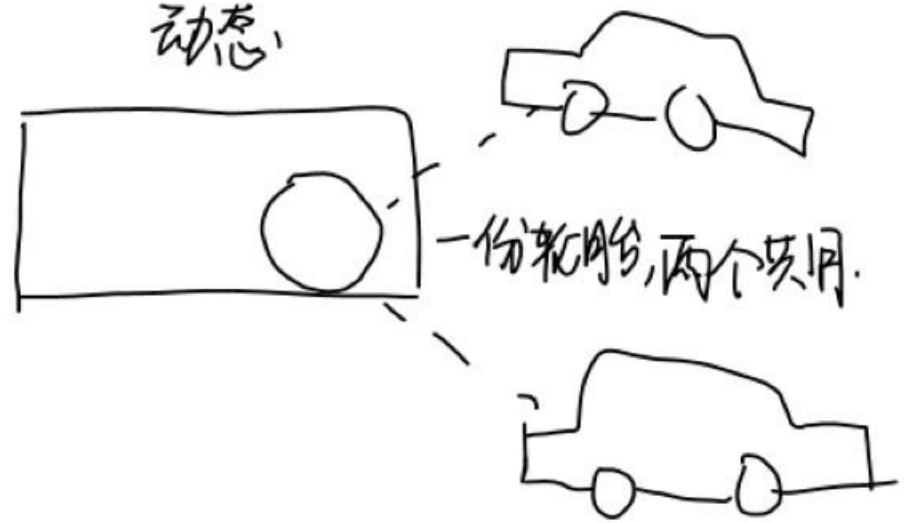
好处：方便调整文件结构

好处：设置一个开关，一份代码，多份版本

库

静态库　　轮胎

动态库



库

静态：链接的时候，把库文件打包到程序里面

动态：链接的时候，给库文件的位置，在运行时再加载到内存

静态　容易部署　难以升级　体积大

动态　不容易部署　容易升级　体积小

静态库

$ gcc -c add.c add.o

打包

```
$ ar crsv libadd.a add.o
```

绿色的部分是固定

黄色的部分是库的名字

$ cp libadd.a /usr/lib

$ gcc -o main main.c -ladd

动态库

$gcc -c add.c -fpic 位置无关的

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

$gcc -o main main.c -ladd

$ ldd 查看依赖的动态库.

版本更新

libadd.so

↳ 符号链接，软链接

①删除原软链接

②重新建立链接 $ln -s ____ libadd.so

在改动另一个文件的路径.

libadd.so.0.0.0

↓

libadd.so.0.0.1

GDB调试器

① 程序编译时. 要加上 -g.

② $gdb 可执行程序.

x 查看内存 (help x)

```
(gdb) x/20cb &str
```

show args 查看命令行参数

set args 设置命令行参数.

backtrace/bt 查看调用栈

l/list 显示代码 行号/函数名

r/run 运行

b/break 打断点 delete 删除

info b 显示所有断点

s/step f11

n/next f10

finish 跳出当函数

c/continue 运行当下个断点

p/print 监视

段错误和core文件

① ulimit -a ⇒ ulimit -c unlimited

UNIX limit

② 运行程序

③ gdb ./可执行程序 core bt查看堆栈.