# linux开发工具笔记

## gcc工具

### gcc编译选项

|  |  |
| --- | --- |
| 选项名 | 作用 |
| -o | 产生目标（.i、.s、.o、可执行文件等） |
| -E | 只运行C预编译器 |
| -S | 告诉编译器产生汇编语言文件后停止编译，产生的汇编语言文件扩展名为.s |
| -Wall | 使gcc对源文件的代码有问题的地方发出警告 |
| -Idir | 将dir目录加入搜索头文件的目录路径 (大写的i) |
| -Ldir | 将dir目录加入搜索库的目录路径 |
| -llib | 链接lib库 (小写的L) |
| -g | 在目标文件中嵌入调试信息，以便gdb之类的调试程序调试 |

### gcc头文件和库文件位置

|  |
| --- |
| 头文件和库文件位置 |
| /usr/include 及其子目录底下的include文件夹 |
| /usr/local/include 及其子目录底下的include文件夹 |
| /usr/lib |
| /usr/local/lib |

### 静态库和动态库

|  |
| --- |
| 静态库与共享库(默认都是以lib开头) |
| 静态库(.a)：程序在编译链接的时候把库的代码链接到可执行文件中。程序运行时候不在需要静态库 |
| 共享库(.so或.sa)：程序在运行的时候才去连接共享库的代码，多个程序共享使用库的代码 |
| 两者区别：静态库每个程序都有一份，会造成内存浪费。动态库是在内存中只有一份。   1. 一个与共享库链接的可执行文件仅仅包含它用到的函数入口地址的一个表，而不是外部函数所在目标文件的整个机器码。 2. 在可执行文件开始运行之前，外部函数的机器码由操作系统从磁盘上的共享库中复制到内存中，这个过程为动态链接（dynamic linking） 3. 共享库可以在多个程序间共享，所以动态链接使得可执行文件更小，节省了磁盘空间。操作系统采用虚拟内存机制允许物理内存中的一份共享库被要用到该库的所有进程共用，节省了内存和磁盘空间。 |

|  |
| --- |
| 生成静态库----就是将.o文件归档为一个.a文件 |
| gcc -Wall hello.c -c libhello.o |
| ar rcs libhello.a libhello.o (ar是gun归档工具，rsc表示replace and create) |
| gcc -Wall main.c libhello.o -o main 或者 gcc -Wall -L. main.c -o main -lhello(注意库文件不要跟在-L选项后面, main.c需要包含hello.h头文件) |

|  |
| --- |
| 库搜索路径 |
| C\_INCLUDE\_PATH、LIBRARY\_PATH (c++ 头文件 CPP\_INCLUDE\_PATH) |
| ①从左到右搜索-I -L指定的目录 |
| ②由环境变量指定的目录 |
| ③由系统指定的目录 |
| 指定LIBRARY\_PATT  使用vim编辑~/.bash\_rc文件在里面增加一行  export LD\_LIBRARY\_PATH=/home/pi/study\_gcc:$LD\_LIBRARY\_PATH  --> . ~/.bash\_rc (使.bash\_rc文件生效执行)  --> sudo ldconfig |

|  |
| --- |
| 生成动态库库 |
| gcc -shared -fPIC hello.c -o libhello.o (shared表示生成共享库 PID: position indepence code) |
| gcc -Wall -L. main.c -o main -lhello (小写的L -lhello) |
| 当静态库和动态库同时存在时，优先使用动态库 |

|  |
| --- |
| 运行动态库 |
| 1.拷贝.so文件到系统共享库路径下 一般指 /usr/lib |
| 2.更改LD\_LIBRARY\_PATH (可配置当前用户和全局) |
| 3.ldconfig (全局配置)  配置ld.so.conf, ldconfig更新ld.so.cache |

|  |
| --- |
| ldd查看链接共享库 |
| ldd main  linux-vdso.so.1 (0x7ef17000)  /usr/lib/arm-linux-gnueabihf/libarmmem.so (0x76f9a000)  libhello.so (0x76f89000)  libc.so.6 => /lib/arm-linux-gnueabihf/libc.so.6 (0x76e35000)  /lib/ld-linux-armhf.so.3 (0x54b42000) |

## Makefile

|  |
| --- |
| make工具通过一个Makefile的文件来完成并自动维护编译工作。Makefile文件描述了整个工程的编译、链接等规则 |

### Makefile基本规则

|  |
| --- |
| 基本规则 |
| TARGET... : DEPENDENCIES...  COMMAND  ... |
| 目标(TARGET)程序产生的文件，如可执行文件和目标文件；目标也可以是要执行的动作，如clean，也称为伪目标(不是为了生成目标文件，是为了做些其他的动作) |
| 依赖(DEPENDENCIES)是用来产生目标的输入文件列表，一个目标通常依赖于多个文件 |
| 命令(COMMAND)是make执行的命令(命令是shell或是在shell下执行的程序)。  注意：每个命令行的起始字符串必须为TAB字符！ |
| 如果DEPENDENCIES中有个一或者多个文件更新的话，COMMAND就要执行，这就是Makefile最核心的内容。 |
| .PHONY 之后的目标是伪目标，被显示成为伪目标就算有个文件名称相同还是会执行伪目标 |
| Makefile action -f other\_makefile (从其他makefile文件执行actioin) |

### Makefile自动化变量

|  |  |
| --- | --- |
| 选项名 | 作用 |
| $@ | 规则的目标名 |
| $< | 规则的第一个依赖文件名 |
| $^ | 规则的所有依赖文件名 |

### Makefile编译多个可执行文件

|  |
| --- |
| 模式规则：%.o:%.c |
| 后缀规则：.c.o: |
| demo:  .PHONY:clean all  CC=gcc  CFLAGS= -Wall -g  BIN=01test 02test 03test  all:$(BIN)  #%.o:%.c  # $(CC) $(CFLAGS) -c $< -o $@  # 和上面的效果是一样的  .c.o:  $(CC) $(CFLAGS) -c $< -o $@  01test:01test.o  $(CC) $(CFLAGS) $^ -o $@  02test:02test.o  $(CC) $(CFLAGS) $^ -o $@  03test:03test.o pub.o  $(CC) $(CFLAGS) $^ -o $@  clean:  rm -rf \*.o $(BIN) |

### make常用内嵌函数

|  |  |
| --- | --- |
| 函数模式 | 作用 |
| $(function arguments) | 函数调用 |
| $(wildcard PATTERN) | 当前目录下匹配模式文件(src=$(wildcard \*.c)) |
| $(patsubst PATTERN, REPLACEMENT, TEXT) | 模式替换函数(如：$(patsubst %.c, %.o, $src) == $(src:.c=.o)) |
| shell函数 | 执行shell命令($(shell ls -d \*/)) |

### 多级目录Makefile

|  |
| --- |
| 多级目录Makefile |
| demo\_1:源码位于多个文件夹生成一个执行文件  目录结构：  .  ├── bll  │ ├── bll.c  │ ├── bll.h  ├── dal  │ ├── dal.c  │ ├── dal.h  ├── main.c  ├── Makefile  └── ui  ├── ui.c  ├── ui.h  Makefile：  CC = gcc  CFLAGS = -Wall -g  BIN = main  SUBDIR = $(shell ls -d \*/)  ROOTSRC = $(wildcard \*.c)  #ROOTOBJ = $(patsubst %.c, %.o, $(ROOTSRC))  ROOTOBJ = $(ROOTSRC:%.c=%.o)  SUBSRC = $(shell find $(SUBDIR) -name '\*.c')  #SUBOBJ = $(patsubst %.c, %.o, $(SUBSRC))  #两者功能是一样的  SUBOBJ = $(SUBSRC:%.c=%.o)  $(BIN):$(ROOTOBJ) $(SUBOBJ)  $(CC) $(CFLAGS) -o $(BIN) $(ROOTOBJ) $(SUBOBJ)  .c.o:  $(CC) $(CFLAGS) -c $< -o $@  .PHONY:clean  clean:  rm -rf $(BIN) $(ROOTOBJ) $(SUBOBJ) |
| demo\_2:生成多个可执行文件  目录结构：  .  ├── Makefile  ├── test01  │   ├── Makefile  │   └── test1.c  └── test02  ├── Makefile  └── test2.c  2 directories, 5 files  上层Makefile：  SUBIDRS = test01 test02  .PHONY: default all clean $(SUBIDRS)  default:all  all clean:  #$(MAKE)相当于make  $(MAKE) $(SUBIDRS) TARGET=$@  $(SUBIDRS):  # -C表示进入目录并且执行当前目录下的Makefile文件，并且参数为$(TARGET)  $(MAKE) -C $@ $(TARGET)  test01(test02与之类似) Makefile:  CC = gcc  CFALGS = -Wall -g  BIN = test1  OBJS = test1.o  .PHONY:all clean print  all: print $(BIN)  print:  @echo "----- amke all in $(PWD) -----"  $(BIN):$(OBJS)  $(CC) $(CFLAGS) $^ -o $@  %.o:%c  $(CC) $(CFALGS) -c $< -o $@  clean:  @echo "----- make clean in $(PWD) -----"  rm -rf $(BIN) $(OBJS) |

## GDB调试