Linux 下文件编译与调试

3.1 gcc/g++编译器

对于.c 格式的 C 文件，可以采用 gcc 或 g++编译

对于 .cc、.cpp 格式的 C++文件，应该采用 g++进行编译

常用的选项：

-c 表示编译源文件

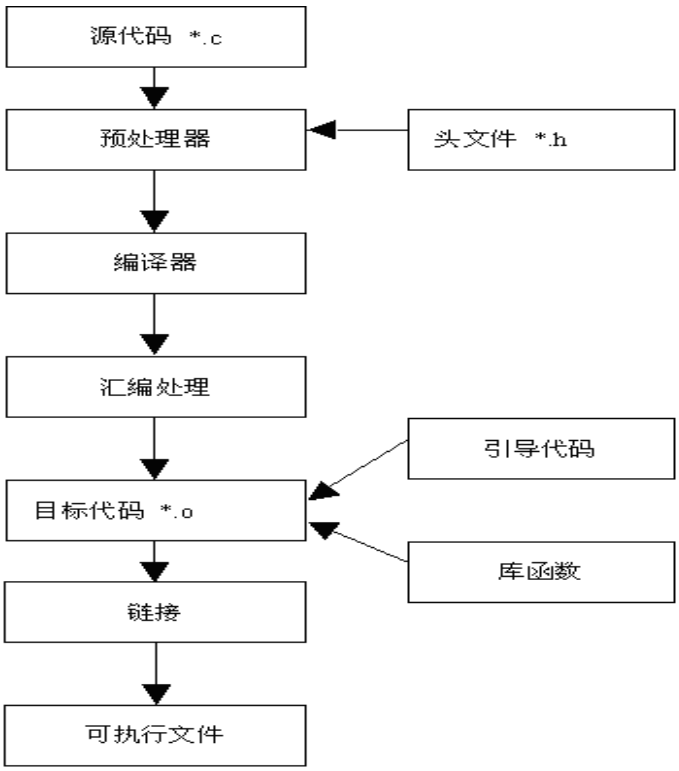
-o 表示输出目标文件

-g 表示在目标文件中产生调试信息，用于 gdb 调试

-D<宏定义> 编译时将宏定义传入进去

-Wall 打开所有类型的警告。

3.1.1 gcc 编译过程：预编译→编译→汇编→链接



当我们进行编译的时候,要使用一系列的工具,我们称之为工具链. 其中包括:预处理器,编译,汇编器 as,连接器. 一个编译过程包括下面几个阶段：

(1)预处理：预处理器将对源文件中的宏进行展开。

(2)编译：gcc 将 c 文件编译成 汇编文件。

(3)汇编：as 将汇编文件编译成机器码。

(4)链接：将目标文件和外部符号进行连接，得到一个可执行二进制文件。

下面以一个很简单的 test.c 来探讨这个过程。

```
#include <stdio.h>

#define NUMBER    1+2
```

```
int main()
{
    int x = NUMBER* NUMBER;
    printf("x=%d\n",x);
    return 0;
}
```

(1) **预处理**：gcc –E test.c -o test.i 我们用 cat 查看 test.i 的内容如下：int x=1+2*1+2;我们可以看到，文件中宏定义 NUMBER 出现的位置被 1+2 替换掉了，其它的内容保持不变。

(2) **编译**：gcc -S test.i –o test.s 通过 cat test.s 查看 test.s 的内容为代码。

(3) **汇编**：as test.s -o test.o 利用 as 将汇编文件编译成机器码。得到输出文件为 test.o. test.o 中为目标机器上的二进制文件. 用 nm 查看文件中的符号： nm test.o 输出如下：00000000 T main。有的编译器上会显示：00000000 b .bss 00000000 d .data 00000000 t .text U ___main U __alloca 00000000 T _main 既然已经是二进制目标文件了，能不能执行呢？试一下./test.o,提示 cannotexecute binary file.原来___main 前面的 U 表示这个符号的地址还没有定下来，T 表示这个符号属于代码。

(4)**链接**：gcc –o test test.o,将所有的.o 文件链接起来生产可执行程序。

3.1.2 gcc 所支持后缀名及常用选项

后 缀 名	所对应的语言	后 缀 名	所对应的语言
.c	C 原始程序	.s/.S	汇编语言原始程序
.C/.cc/.cxx	C++原始程序	.h	预处理文件（头文件）
.m	Objective-C 原始程序	.o	目标文件
.i	已经过预处理的 C 原始程序	.a/.so	编译后的库文件
.ii	已经过预处理的 C++原始程序		

选项	含义
-c	只编译不链接，生成目标文件“.o”
-S	只编译不汇编，生成汇编代码
-E	只进行预编译，不做其他处理
-g	在可执行程序中包含标准调试信息
-o file	指定将 file 文件作为输出文件
-v	打印出编译器内部编译各过程的命令行信息和编译器的版本
-I dir	在头文件的搜索路径列表中添加 dir 目录

预处理阶段：对包含的头文件（#include）和宏定义（#define、#ifdef 等）进行处理

gcc **–E** hello.**c** –o hello.**i**　　//-o 表示输出为指定文件类型　-E 将源文件（*.c）转换为（*.i）

编译阶段：检查代码规范性、语法错误等，在检查无误后把代码翻译成汇编语言

gcc **–S** hello.**i** –o hello.**s**　　//-S 将已预处理的 C 原始程序（*.i）转换为（*.s）

链接阶段：将.s 的文件以及库文件整合起来链接为可执行程序

gcc –o hello.**exe** hello.**s** //最后将汇编语言原始程序(*.s)和一些库函数整合成（*.exe）

Example1:

```
#include <stdio.h>
#define MAX 100
#define max(a,b) ((a)>(b)?(a):(b))     //宏定义，执行-E 之后被替换
main()
{
        printf("MAX=%d\n",MAX);
        printf("max(3,4)=%d\n",max(3,4));
}
```

//法一：

gcc –E project1.c –o project1.i //预编译，生成已预编译过的 C 原始程序*.i

gcc –S project1.i –o project1.s //编译，生成汇编语言原始程序*.s

gcc –o project1.exe project1.s //链接，生成可执行程序

//法二：

gcc –c project1.c –o project1.o //编译

gcc –o project1.exe project1.o //链接

//法三：

gcc –o project1.exe project1.c //编译并链接

Example2:

```
        #include <stdio.h>
main()
{
                #ifdef cjy        //表示如果定义了 cjy，即命令行参数传了
cjy，就执行下面的输出
                        printf("cjy is defined!\n");
                #else
                        printf("cjy is not defined!\n");
                #endif
                printf("main exit\n");
}
```

gcc –E project2.c –o project2.i –D cjy //条件编译，用-D 传递，如果没有传 cjy 则执行#else

gcc –S project2.i –o project2.s

gcc –o project2.exe project2.s

或：gcc –o project2 project2.c –D cjy

3.1.3 gcc 库选项

选　　项	含　　义
-static	进行静态编译，即链接静态库，禁止使用动态库
-shared	1．可以生成动态库文件 2．进行动态编译，尽可能地链接动态库，只有没有动态库时才会链接同名的静态库（默认选项，即可省略）
-L dir	在库文件的搜索路径列表中添加 dir 目录
-lname	链接称为 libname.a（静态库）或者 libname.so（动态库）的库文件。若两个库都存在，则根据编译方式（-static 还是-shared）而进行链接
-fPIC（或-fpic）	生成使用相对地址的位置无关的目标代码（Position Independent Code）。然后通常使用 gcc 的-static 选项从该 PIC 目标文件生成动态库文件。

3.1.4 函数库分为静态库和动态库。

静态库是目标文件.a 的归档文件（格式为 lib*name*.a）。如果在编译某个程序时链接静态库，则链接器将会搜索静态库并直接拷贝到该程序的可执行二进制文件到当前文件中；

动态库（格式为 lib*name*.so[.*主版本号.次版本号.发行号*]）。在程序编译时并不会被链接到目标代码中，而是在程序运行时才被载入。

创建静态库

$ gcc　-c　add.c　　//编译 add.c 源文件生成 add.o 目标文件

$ **ar　crsv**　libadd.a　add.o　//对目标文件*.o 进行归档，生成 lib*.a，将库文件 libadd.a 拷贝到/lib 或者/usr/lib 下（系统默认搜索库路径）

$ gcc　-o　main　main.c　–ladd

（-ladd 表示链接库文件 libadd.a/.so ）

$./main

创建动态库

$ gcc -fPIC -Wall -c add.c

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

$ gcc -o main main.c –ladd

在运行 main 前，需要注册动态库的路径。将库文件拷贝到/lib 或者/usr/lib 下（系统默认搜索库路径）。

$ cp libadd.so /lib //通常采用的方法，→ cp lib*.so /lib

$ ./main

静态库与动态库的比较：

动态库只在执行时才被链接使用，不是直接编译为可执行文件，并且一个动态库可以被多个程序使用故可称为共享库。

静态库将会整合到程序中，在程序执行时不用加载静态库。 因此，静态库会使你的程序臃肿并且难以升级，但比较容易部署。而动态库会使你的程序轻便易于升级但难以部署。

gcc --- 警告选项

选　　项	含　　义
-ansi	支持符合 ANSI 标准的 C 程序
-pedantic	允许发出 ANSI C 标准所列的全部警告信息
-pedantic-error	允许发出 ANSI C 标准所列的全部错误信息
-w	关闭所有告警
-Wall	允许发出 gcc 提供的所有有用的报警信息
-werror	把所有的告警信息转化为错误信息，并在告警发生时终止编译过程

对于如下程序：

#include <stdio.h>

```
void main()
{
    long long temp = 1;
    printf("This is a bad code!\n");
    return 0;
}
```

-ansi : 生成标准语法（ANSI C 标准）所要求的警告信息（并不列出所有警告）

$ gcc **–ansi** warning.c –o warning

warning.c: 在函数“main”中：

warning.c:7 警告：在无返回值的函数中，“return”带返回值

warning.c:4 警告：“main”的返回类型不是“int”

可以看出，该选项并没有发现“long long”这个无效数据类型的错误

-pedantic : 列出 ANSI C 标准的全部警告信息。

$ gcc **–pedantic** warning.c –o warning

warning.c: 在函数“main”中：

warning.c:5 警告：ISO C89 不支持“long long”

warning.c:7 警告：在无返回值的函数中，“return”带返回值

warning.c:4 警告：“main”的返回类型不是“int”

-Wall : 列出所有的警告信息（常用）

$ gcc **–Wall** warning.c –o warning

warning.c:4 警告：“main”的返回类型不是“int”

warning.c: 在函数“main”中：

warning.c:7 警告：在无返回值的函数中，“return”带返回值

warning.c:5 警告：未使用的变量“tmp”

$gcc **–Werror** warning.c –o warming

通常用的是-Wall 显示所有有用的报警信息。

3.1.5 gcc 优化选项

gcc 对代码进行优化通过选项“-On”来控制优化级别（n 是整数）。不同的优化级别对应不同的优化处理工作。如使用优化选项“-O1”主要进行线程跳转和延迟退栈两种优化。使用优化选项“-O2”除了完成所有“-O1”级别的优化之外，还要进行一些额外的调整工作，如处理其指令调度等。选项“-O3”则还包括循环展开或其他一些与处理器特性相关的优化工作。虽然优化选项可以加速代码的运行速度，但对于调试而言将是一个很大的挑战。因为代码在经过优化之后，原先在源程序中声明和使用的变量很可能不再使用，控制流也可能会突然跳转到意外的地方，循环语句也有可能因为循环展开而变得到处都有，所有这些对调试来讲都是不好的。所以在调试的时候最好不要使用任何的优化选项，只有当程序在最终发行的时候才考虑对其进行优化。

通常用的是-O2

详细可以看 https://blog.csdn.net/gatieme/article/details/48898261

3.2 make 工程管理器

3.2.1 为什么要使用 Makefile

可以试想一下，有一个上百个文件的代码构成的项目，如果其中只有一个活少数几个文件进行了修改，如果再从头到尾将每一个文件都重新编译是个比较繁琐的过程。为此，引入了 Make 工程管理器的概念，工程管理器指管理较多的文件，它是自动管理器能根据文件时间自动发现更新过的文件而减少编译的工作量，同时通过读入 Makefile 文件来执行大量的编译工作

Makefile 格式

target: dependency_files　//目标项:依赖项

<　TAB　>command　//必须以 tab 开头，command 编译命令

注意点：在写 command 命令行的时候，必须要在前面按 TAB 键

例如，有 Makefile 文件，内容如下：

```
main.exe:main.o func.o
    g++ -o main.exe main.o func.o
main.o:main.cpp
    g++ -c main.cpp
func.o:func.cpp
    g++ -c func.cpp
```

使用 make 编译

对于该 Makefile 文件,程序 make 处理过程如下:

make 程序首先读到第 1 行的目标文件 main.exe 和它的两个依赖文件 main.o 和 func.o;然后比较文件 main.exe 和 main.o/func.o 的产生时间，

如果 main.exe 比 main.o/func.o 旧的话，则执行第 2 条命令，以产生目标文件 main.exe。

在执行第 2 行的命令前,它首先会查看 makefile 中的其他定义，看有没有以第 1 行 main.o 和 func.o 为目标文件的依赖文件，如果有的话，继续按照(1)、(2)的方式匹配下去。

根据(2)的匹配过程，make 程序发现第 3 行有目标文件 main.o 依赖于 main.cpp，则比较目 main.o 与它的依赖文件 main.cpp 的文件新旧,如果 main.o 比 main.cpp 旧,则执行第 4 行的命令以产生目标文件 main.o.

在执行第 4 条命令时,main.cpp 在文件 makefile 不再有依赖文件的定义,make 程序不再继续往下匹配,而是执行第 4 条命令,产生目标文件 main.o

目标文件 func.o 按照上面的同样方式判断产生.

执行(3)、(4)产生完 main.o 和 func.o 以后，则第 2 行的命令可以顺利地执行了，最终产生了第 1 行的目标文件 main.exe。

特殊处理与伪目标

.PHONY 是 Makefile 文件的关键字，表示它后面列表中的目标均为伪目标

```
.PHONY:b
b:
    echo  ‘b’                  //通常用@echo “hello”
```

伪目标通常用在清理文件、强制重新编译等情况下。

Example1:main.c 函数，func.c 函数为前面计算+,-,*,/运算的程序

#vi **Makefile**　　//系统默认的文件名为 Makefile

main.exe:main.o func.o　　//表示要想生成 main.exe 文件，要依赖于 main.o 和 func.o 文件

　　gcc -o main.exe main.o func.o//如果 main.o,func.o 已经存在了，就链接成 main.exe

main.o:main.c　　//表示 main.o 文件依赖于 main.c 文件

　　gcc -c main.c　　//编译 main.c，默认生成 main.o。可写为：gcc –c main.c –o main.o

func.o:func.c　　//表示 func.o 文件依赖于 func.c 文件

　　gcc -c func.c　　//如果 func.c 存在，则编译 func.c ,生成 func.o

.PHONY:rebuild clean //表示后面的是**伪目标**，通常用在清理文件、强制重新编译等情况下

rebuild:clean main.exe //先执行清理，在执行 main.exe

clean:

　　rm –rf main.o func.o main.exe　　//最后删除.o 和.exe 的文件

按 ESC 键之后，:wq 保存退出

再执行下面的命令：

#make　　//直接 make,即从默认文件名（Makefile）的第一行开始执行

#make clean　//表示执行 clean: 开始的命令段

#make func.o　//表示执行 func.o: 开始的命令段

#make rebuild //则先执行清除，再重新编译连接

如果不用系统默认的文件名 Makefile，而是用户随便起的一个名字，如：

#vi Makefile11

则 make 后面必须要加上 -f Makefile11 ，如：

#make –f Makefile11 clean //表示执行 clean: 开始的命令段

#make –f Makefile11 main.exe //表示执行 main.exe: 开始的命令段

3.2.2 变量、函数与规则

随着软件项目的变大、变复杂，源文件也越来越多，如果采用前面的方式写 makefile 文件，将会使 makefile 也变得复杂而难于维护。通过 make 支持的变量定义、规则和内置函数，可以写出通用性较强的 makefile 文件，使得同一个 makefile 文件能够适应不能的项目。

变量：用来代替一个文本字符串

定义变量的 2 种方法：

变量名=变量值 递规变量展开(几个变量共享一个值) //不常用

变量名:=变量值 简单变量展开(类似于 C++的赋值) //通常采用这种形式

使用变量的一般方法：$(变量名)=??? 赋值

???=$(变量名) 引用

例：将以前的那个可以写为：

```
OBJS:=main.o func.o//相当于 main.o func.o
EXE:=main.exe
$(EXE):$(OBJS)
    g++ -o $(EXE) $(OBJS)
main.o:main.cpp
    g++ -c main.cpp  -o main.o
func.o:func.cpp
    g++ -c func.cpp  -o func.o
clean:
    rm –rf $(EXE) $(OBJS)
```

变量分为：**用户自定义变量，预定义变量（CFLAGS），自动变量，环境变量**

自动变量：指在使用的时候，自动用特定的值替换。

常用的有：

变量	说明
$@	当前规则的目标文件(重点)
$<	当前规则的第一个依赖文件
$^	当前规则的所有依赖文件，以空格分隔(重点)
$?	规则中日期新于目标文件的所有相关文件列表，逗号分隔
$(@D)	目标文件的目录名部分
$(@F)	目标文件的文件名部分

Examp：用自动变量：

CFLAGS:= -Wall -O2 –fpic

预定义变量：内部事先定义好的变量，但是它的值是固定的，并且有些的值是为空的。

AR：库文件打包程序默认为 ar

AS：汇编程序，默认为 as

CC：c 编译器默认为 cc

CPP：c 预编译器，默认为$(CC) –E

CXX：c++编译器，默认为 g++

RM：删除，默认为 rm –f

ARFLAGS：库选项，无默认

ASFLAGS：汇编选项，无默认

CFLAGS：c 编译器选项，无默认

CPPFLAGS：c 预编译器选项，无默认

CXXFLAGS：c++编译器选项

根据内部变量，可以将 makefile 改写为：

```
OBJS:=main.o func.o
CC:=g++
main.exe:$(OBJS)
    $(CC) -o $@ $^
main.o:main.cpp
    $(CC) -o $@ -c $^
func.o:func.cpp
    $(CC) -o $@ -c $^
```

规则分为：普通规则，隐含规则，模式规则

隐含规则：//*.o 文件自动依赖*.c 或*.cc 文件，所以可以省略

main.o:main.cpp 等

```
OBJS := main.o fun.o
CFLAGS := -Wall –O2 -g
main.exe: $(OBJS)
        gcc $^ -o $@
```

模式规则：通过匹配模式找字符串， %匹配 1 或多个任意字符串

%.o: %.cpp 任何目标文件的依赖文件是与目标文件同名的并且扩展名为.cpp 的文件

```
OBJS := main.o fun.o
CFLAGS := -Wall –O2 –g
main.exe : $(OBJS)
        gcc $^ -o $@
%.o: %.cpp    //模式通配
gcc -o $@ -c $^
```

另外还可以指定将*.o、*.exe、*.a、*.so 等编译到指定的目录中：

```
DIR:=./Debug/
EXE:=main.exe
OBJS:=main.o
LIBFUNCSO:=libfunc.so
CFLAGS:= -fpic
$(DIR)$(EXE):$(DIR)$(OBJS) $(DIR)$(LIBFUNCSO)
```

```
	gcc -o $@ $< -L./ -lfunc
$(DIR)$(LIBFUNCSO):$(DIR)func.o
	gcc -shared -o $@ $^
$(DIR)main.o:main.c
	gcc -o $@ -c $^
$(DIR)func.o:func.c
	gcc $(CFLAGS) -c $^ -o $@
.PHONY:rebuild clean
rebuild:clean $(DIR)$(EXE)
clean:
rm -rf $(DIR)*.o $(DIR)*.exe $(DIR)*.so
```

注意：当 OBJS 里面有多项的时候，此时(DIR)(OBJS)只能影响到 OBJS 中第一个，后面的全部无效，因此需要全部列出来。

函数：

1. wildcard 搜索当前目录下的文件名，展开成一列所有符合由其参数描述的文件名，文件间以空格间隔。SOURCES = $(wildcard *.cpp)把当前目录下所有'.cpp'文件存入变量 SOURCES 里。

2. 字符串替换函数：$(patsubst 要查找的子串,替换后的目标子串，源字符串)。将源字符串(以空格分隔)中的所有要查找的子串替换成目标子串。如 OBJS = $(patsubst %.cpp,%.o,$(SOURCES))

把 SOURCES 中'.cpp' 替换为'.o' 。

3. $(addprefix 前缀，源字符串)函数把第二个参数列表的每一项前缀

上第一个参数值

下面是一个较为通用的 makefile:

```
DIR    := ./debug
EXE    := $(DIR)/Main.exe
CC     := g++
LIBS   :=
SRCS   := $(wildcard *.cpp) $(wildcard *.c) $(wildcard *.cc)
OCPP   := $(patsubst %.cpp, $(DIR)/%.o, $(wildcard *.cpp))
OC     := $(patsubst %.c, $(DIR)/%.co, $(wildcard *.c))
OCC    := $(patsubst %.cc, $(DIR)/%.cco, $(wildcard *.cc))
OBJS   := $(OC) $(OCC) $(OCPP)
RM     := rm -rf
CXXFLAGS := -Wall -g
start : mkdebug $(EXE)
mkdebug :
   @if [ ! -d $(DIR) ]; then mkdir $(DIR); fi;
$(EXE)    : $(OBJS)
   $(CC) -o $@ $(OBJS) $(addprefix -l,$(LIBS))
$(DIR)/%.o : %.cpp
   $(CC) -c $(CXXFLAGS) $< -o $@
$(DIR)/%.co : %.c
   $(CC) -c $(CXXFLAGS) $< -o $@
```

```
$(DIR)/%.cco : %.cc
    $(CC) -c $(CXXFLAGS) $< -o $@
.PHONY   : clean rebuild
clean :
    @$(RM) $(DIR)/*.exe $(DIR)/*.o $(DIR)/*.co $(DIR)/*.cco
rebuild: clean start
```

（注意 gcc 和 g++的区别：当 main.c 中调用了其他源文件的程序时，gcc -o main.o -c main.c 或 gcc -o Debug/main.o -c main.c 都没有问题，而 g++ -o main.o -c main.c 没问题，但 g++ -o Debug/main.o -c main.c 有问题，如果想解决这个问题，需要在 main.c 中添加对于函数调用的函数原型声明即可）

make 的命令行选项：

命 令 格 式	含　义
-C dir	读入指定目录下的 makefile
-f file	读入当前目录下的 file 文件作为 makefile
-i	忽略所有的命令执行错误
-I dir	指定被包含的 makefile 所在目录
-n	只打印要执行的命令，但不执行这些命令
-p	显示 make 变量数据库和隐含规则
-s	在执行命令时不显示命令
-w	如果 make 在执行过程中改变目录，则打印当前目录名

3.3.3 同时编译出多个可执行二进制

采用 for 循环编译多个目标文件的 Makefile 写法

```
SRCS = open1.c open2.c

OBJECTS = $(SRCS:%.c=%.o)

TARGETS = $(SRCS:%.c=%)

all : $(TARGETS)

        @for i in $(TARGETS);do gcc -o $${i} $${i}.c;done

.PHONY:clean

clean:

        rm $(TARGETS)
```

3.3 程序调试

3.3.1 gdb 常用命令

Linux 包含了一个叫 gdb 的调试程序。gdb 可以用来调试 C 和 C++ 程序。

在程序编译时用 -g 选项可打开调试选项.

常见的调试程序的步骤如下：

gcc –o filename –Wall filename.c –g //**编译一定要加-g**

gdb filename //进入调试

l //显示代码 (list)

b 4 //在第四行设置断点 相当于 Windows 的 F9 (break)

r //运行 相当于 Windows 的 F5 (run)

n　　//下一步不进入函数　相当于 Windows 的 F10 (next)

s　　//表示单步进入函数，相当于 Windows 的 F11　(step)

p I　　//打印变量 I　相当于 Windows 的 Watch 窗口 (print)

c　　//运行到最后 (continue)

q　　//退出　相当于 Windows 的 Shift+F5(quit)

3.3.2 gdb 调试命令列表

命 令 格 式	含　义
set args 运行时的参数	指定运行时参数，如 set args 2
show args	查看设置好的运行参数
path dir	设定程序的运行路径
show paths	查看程序的运行路径
set environment var [=value]	设置环境变量
show environment [var]	查看环境变量
cd dir	进入到 dir 目录，相当于 shell 中的 cd 命令
pwd	显示当前工作目录
shell command	运行 shell 的 command 命令

info b	查看所设断点
break [文件名:]行号或函数名 <条件表达式>	设置断点
tbreak [文件名:]行号或函数名 <条件表达式>	设置临时断点，到达后被自动删除
delete [断点号]	删除指定断点，其断点号为“info b”中的第一栏。若缺省断点号则删除所有断点
disable [断点号]]	停止指定断点，使用“info b”仍能查看此断点。同 delete 一样，省断点号则停止所有断点
enable [断点号]	激活指定断点，即激活被 disable 停止的断点
condition [断点号] <条件表达式>	修改对应断点的条件
ignore [断点号]<num>	在程序执行中，忽略对应断点 num 次
Step	单步恢复程序运行，且进入函数调用
Next	单步恢复程序运行，但不进入函数调用
Finish	运行程序，直到当前函数完成返回
c	继续执行函数，直到函数结束或遇到新的断点

命 令 格 式	含　义
list <行号>|<函数名>	查看指定位置代码
file [文件名]	加载指定文件
forward-search 正则表达式	源代码前向搜索
reverse-search 正则表达式	源代码后向搜索
dir dir	停止路径名
show directories	显示定义了的源文件搜索路径
info line	显示加载到 gdb 内存中的代码

print 表达式|变量	查看程序运行时对应表达式和变量的值
x <n/f/u>	查看内存变量内容。其中 n 为整数表示显示内存的长度，f 表示显示的格式，u 表示从当前地址往后请求显示的字节数
display 表达式	设定在单步运行或其他情况中，自动显示的对应表达式的内容
backtrace	查看当前栈的情况，即可以查到调用哪些函数尚未返回。

按 Tab 键补齐命令,用光标键上下翻动历史命令. 用 help up 看帮助

3.3.3 gdb 调试段错误

小知识：

1. 在 linux 中利用 system(“clear”);实现类似于 windows 里面的清屏函数 system(“cls”);

2. LINUX 中可以通过下面的方式可以实现 system("pause");功能：

printf(“Press any key to continue…”);

getchar();

getchar(); //要用两个 getchar()函数

3. linux 中如何刷新输入缓冲区，利用 getchar()函数即可。输出缓冲区可以利用 fflush(stdout);

4.命令 x 是用来检查内存情况，英文是 examine 含义，使用方法

x /20xb 变量首地址，其中 20x 代表 16 进制的长度，b 代表字节的含义

5.针对**段错误**，可以通过 ulimit -c unlimited 设置 core file size 为不限制大小，设置完毕后，可以通过 ulimit -a 进行查看是否设置 ok，这时候再次运行程序，会产生 core 文件，通过 gdb 可执行程序 core 文件，进行调试。直接通过 bt 可以看到程序段错误时的现场，通过 f 1 可以直接切换到程序现场。

gdb ./test2 core

6.调试正在运行的程序，通过 attach 进程 ID，调试正在运行的程序