第3章
Linux C 开发工具

本章主要内容

- 编辑器vi的使用
- 编译器gcc的使用
- 调试器gdb的使用
- 工程管理器make的使用

2.1 文本编辑器vi的使用

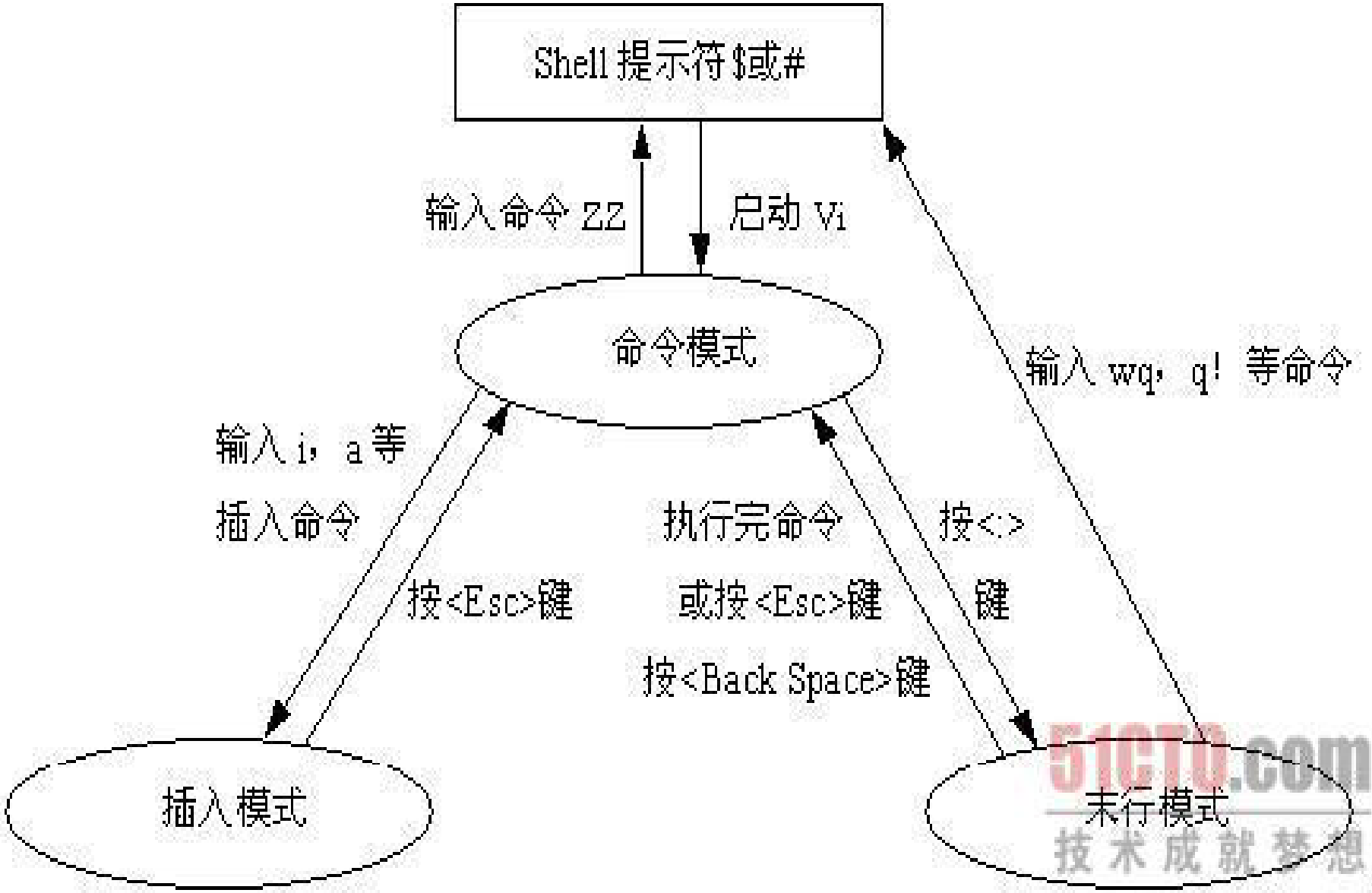
vi编辑器

- vi编辑器是Linux和Unix上最基本的文本编辑器，工作在字符模式下。由于不需要图形界面，vi是效率很高的文本编辑器。尽管在Linux上也有很多图形界面的编辑器可用，但vi在系统和服务器管理中的功能是那些图形编辑器所无法比拟的。
- vi编辑器可以执行输出、删除、查找、替换、块操作等众多文本操作，而且用户可以根据自己的需要对其进行定制，这是其他编辑程序所没有的。

vi的工作模式

- 命令行模式（Command Mode）
  - 在该模式下用户可以输入命令来控制屏幕光标的移动，字符、单词或行的删除，移动复制某区段，也可以进入到底行模式或者插入模式下。
- 插入模式（Insert Mode）
  - 用户只有在插入模式下才可以进行字符输入，用户按［Esc］键可回到命令行模式下。
- 底行（末行）模式（Last Line Mode）
  - 在该模式下，用户可以将文件保存或退出vi，也可以设置编辑环境，如寻找字符串、显示行号等。这一模式下的命令都是以“：”开始。

模式之间的转换

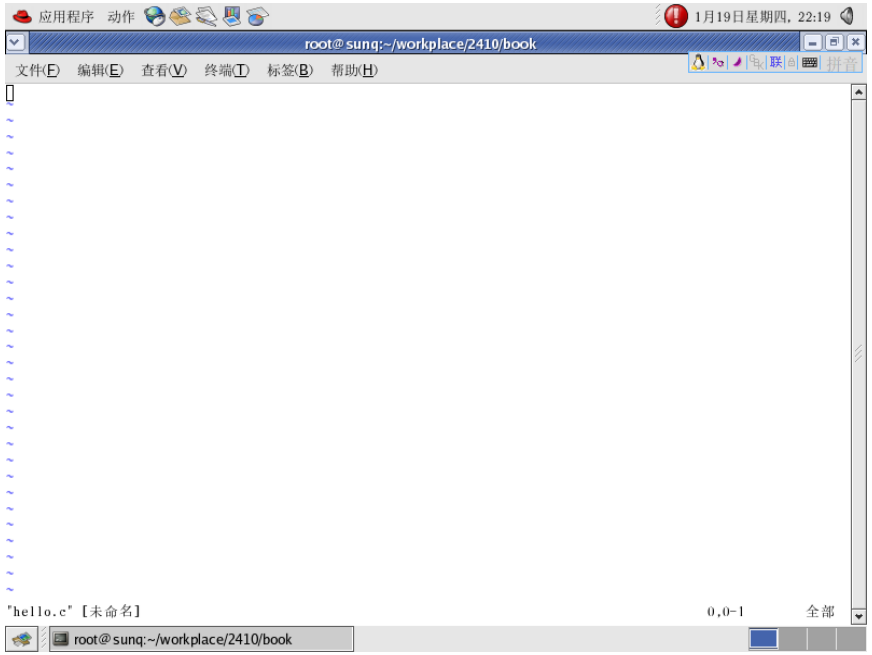

Shell 提示符 $或#
输入命令 ZZ
启动 Vi
命令模式
输入 wq，q! 等命令
输入 i，a 等
插入命令
执行完命令
按<:>
按<Esc>键
或按 <Esc>键
键
按<Back Space>键
插入模式
末行模式
51CTO.com
技术成就梦想

vi的基本操作

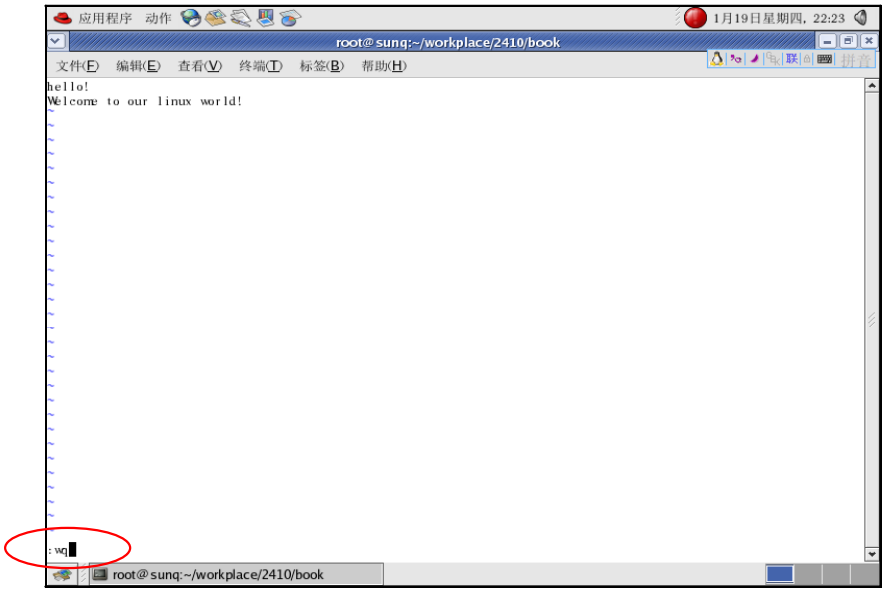
进入与离开

- 进入vi可以直接在系统提示符下键入vi ＜文档名称＞，vi可以自动载入所要编辑的文档或是创建一个新的文档。如在shell中键入vi hello.c（新建文档）即可进入vi画面.
- 进入vi后屏幕最左边会出现波浪符号，凡是有该符号就代表该行目前是空的。此时进入的是命令行模式。
- 要离开vi可以在底行模式下键入“:q”（不保存离开），“:wq”（保存离开）则是存档后再离开（注意冒号）。

vi的基本操作



在vi中打开/新建文档



在vi中退出文档

vi的常用命令（1）

特　征	命令	作　用
新增	a	从光标所在位置后面开始新增资料，光标后的资料随新增资料向后移动，进入插入模式
	A	从光标所在列最后面的地方开始新增资料
插入	i	从光标所在位置前面开始插入资料，游标后的资料随新增资料向后移动，进入插入模式
	I	从光标所在列的第一个非空白字元前面开始插入资料
开始	o	在光标所在列下新增一列，并进入插入模式
	O	在光标所在列上方新增一列，并进入插入模式

vi的常用命令（2）

特征	ARM	作　用
删除	x	删除光标所在的字符，可以5x，删除5个字符
	dd	删除光标所在的行，3dd，删除3行
	s	删除光标所在的字符，并进入输入模式
	S	删除光标所在的行，并进入输入模式
修改	r 待修改字符	修改光标所在的字符，键入r后直接键入待修改字符，例如ra，将光标处字符替换为a
	R	进入取代状态，可移动光标键入所指位置的修改字符，该取代状态直到按［Esc］才结束 例如Rabcdef，将光标开始的字符一次替换为abcdef
复制	yy	复制光标所在的行，5yy：复制5行
	nyy	复制光标所在的行向下n行
	p	将缓冲区内的字符粘贴到光标所在位置

vi的常用命令（3）

指　令	作　用
0	移动到光标所在行的最前面
$	移动到光标所在行的最后面
［Ctrl］d	光标向下移动半页
［Ctrl］f	光标向下移动一页
［Ctrl］u	光标向上移动半页
［Ctrl］b	光标向上移动一页
h	光标左移一列
H	光标移动到当前屏幕的第一行第一列
M	光标移动到当前屏幕的中间行第一列
L	光标移动到当前屏幕的最后行第一列

vi的常用命令（4）

指　令	作　用
b	移动到上一个字的第一个字母
w	移动到下一个字的第一个字母
e	移动到下一个字的最后一个字母
^	移动到光标所在行的第一个非空白字符
n-	向上移动n行
n+	向下移动n行
nG	移动到第n行
u	取消上一次的编辑命令

vi的常用命令（5）

vi的查找与替换

特　征	ARM	作　用
查找	/<要查找的字符>	向下查找要查找的字符
	?<要查找的字符>	向上查找要查找的字符
重复查找	*n	向文件头方向重复前一个查找命令
重复查找	*N	向文件尾方向重复前一个查找命令

vi的常用命令（5）

vi的查找与替换

特　征	ARM	作　用
替换	:s/oldstr/newstr	当前行的第一个oldstr用newstr替换
	:s/oldstr/newstr/g	当前行的所有oldstr用newstr替换
替换	:1,10s/string1/string2/g	1,n：替换范围从第1行到第10行 s：转入替换模式 string1/string2:把所有string1替换为string2 g：强制替换而不提示

vi的基本操作

vi的文件操作指令

指　令	作　用
：q	结束编辑，退出vi
：q!	不保存编辑过的文档
：w	保存文档，其后可加要保存的文件名
：wq	保存文档并退出
zz	功能与“：wq”相同
：x	功能与“：wq”相同
:set number	显示行号
:n	将光标定位到第n行

vi的使用实例分析

vi使用实例内容

（1）在/root目录下建一个名为vi的目录。
（2）进入vi目录。
（3）将文件/etc/inittab复制到当前目录下。
（4）使用vi编辑当前目录下的inittab。
（5）将光标移到第17行。
（6）复制该行内容。
（7）将光标移到最后一行行首。
（8）粘贴复制行的内容。
（9）撤销第9步的动作。
（10）将光标移动到最后一行的行尾。
（11）粘贴复制行的内容。
（12）光标移到“si::sysinit:/etc/rc.d/rc.sysinit”。
（13）删除该行。
（14）存盘但不退出。
（15）将光标移到首行。
（16）插入模式下输入“Hello,this is vi world!”。
（17）返回命令行模式。
（18）向下查找字符串“0:wait”。
（19）再向上查找字符串“halt”。
（20）强制退出vi，不存盘。

1.2.3 vi的使用实例分析

vi使用实例解析

（1）mkdir /root/vi
（2）cd /root/vi
（3）cp /etc/inittab ./
（4）vi ./inittab
（5）17<enter>（命令行模式）
（6）yy
（7）G
（8）p
（9）u
（10）$
（11）p
（12）21G
（13）dd
（14）:w（底行模式）
（15）1G
（16）i 并输入“Hello,this is vi world!”（插入模式）
（17）Esc
（18）/0:wait（命令行模式）
（19）?halt
（20）:q!（底行模式）

2.2 编译器GCC的使用

- 2.2.1 GCC概述
- 2.2.2 GCC编译流程分析
- 2.2.3 GCC警告提示
- 2.2.4 GCC使用库函数
- 2.2.5 GCC代码优化

2.2.1 GCC概述

- GCC除了能支持C语言外，目前还支持Ada语言、C++语言、Java语言、Objective C语言、PASCAL语言、COBOL语言，以及支持函数式编程和逻辑编程的Mercury语言等。

2.2.2 GCC编译流程分析

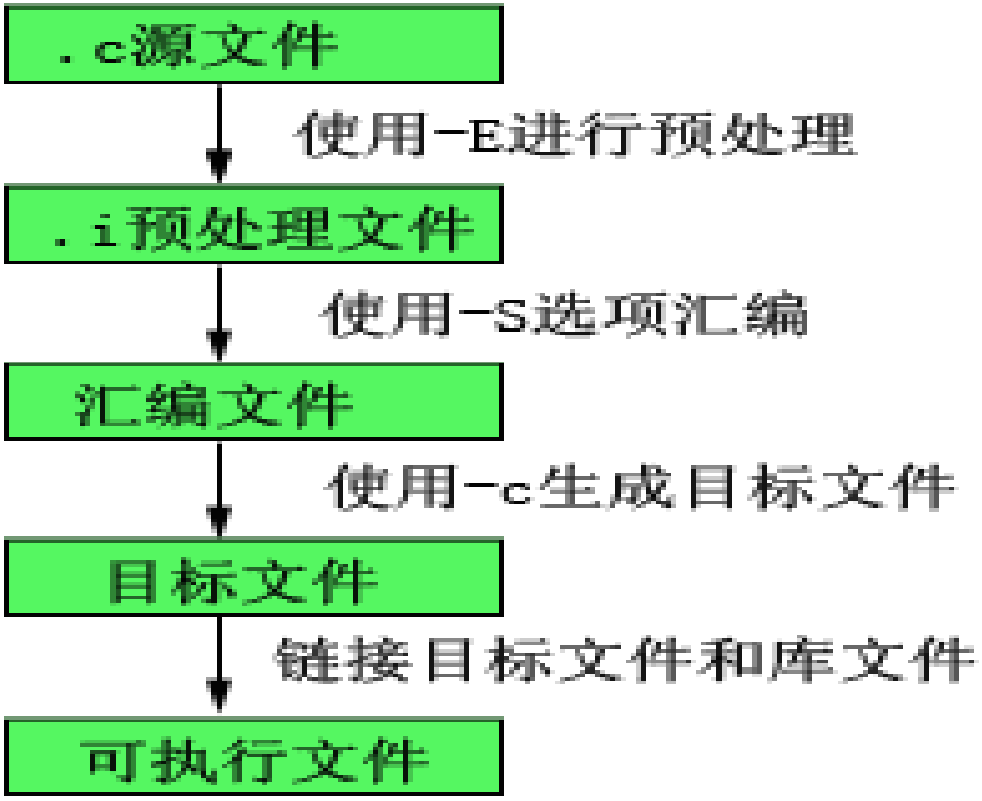
- GCC使用的基本语法为：

  gcc [option | filename]

参数	作用
-E	只运行C预编译器，配合-o可指定得到预处理过的.i文件
-S	配合-o将预处理输出文件.i文件汇编成扩展名为.s的汇编语言源代码文件
-c	只编译并生成后缀名为.o的目标文件，不连接成为可执行文件
-o	指定可执行文件的名称，如果不加该参数，可执行文件默认名为a.out
-g	产生调试工具Gdb所必要的符号信息，要调试程序，必须加入该选项
-l	将该参数后跟的目录加入到程序头文件列表中
-L	首先到该参数后跟的目录中寻找所需要的库文件
-Wall	生成所有警告信息

2.2.2 GCC编译流程分析

使用gcc进行的编译过程是一个相对复杂的过程，可分为预处理（Pre-Processing）、 编译（Compiling）、 汇编（Assembling）和链接（Linking）四个阶段。



2.2.2 GCC编译流程分析

- **GCC使用的基本语法为：**

  gcc [option | filename]

- **预处理阶段**
  - 输入：C语言的源文件
  - 输出：生成*.i得中间文件
  - 功能：处理文件中的#ifdef，#include和#define等预处理命令。
  - 格式：gcc –E –o [目标文件] [编译文件]

    或　gcc –E [编译文件]　-o　[目标文件]

  选项“-E”可以使编译器在预处理结束时就停止编译

  选项“-o”是指定GCC输出的结果。

  编译文件一般以.c为后缀名，目标文件以.i为后缀名

预处理阶段实例

- 已知test.c文件内容如下

```
#include <stdo.h>
Int main( )
{ int a;
   scanf(“%d”,a);
   printf(“a=%d”,a);
 }
```

- 编译命令：gcc –E test.c –o test.i
- 提示错误：stdo.h：没有那个文件或目录
  scanf语句的错误不提示

2.2.2 GCC编译流程分析

编译阶段

- 输入：中间文件*.i
- 输出：汇编语言文件*.s
- 功能：此时检查语法错误。
- 格式：gcc –s –o [目标文件] [编译文件]

  或 gcc –s [编译文件] -o [目标文件]

  选项“-s”可以使编译器完成编译阶段就停止

  选项“-o”是指定GCC输出的结果。
- 实例：gcc –s test.i –o test.s
- 提示：format ‘%d’ expects type ‘int *’ but argument 2 has type ‘int’

2.2.2 GCC编译流程分析

汇编阶段

- 输入：汇编文件*.s
- 输出：二进制机器代码*.o
- 格式：gcc –c –o [目标文件] [编译文件]

  或　gcc –c [编译文件]　-o　[目标文件]

  选项“-c”可以使编译器完成汇编阶段就停止

  选项“-o”是指定GCC输出的结果。
- 实例：gcc –c　test.s –o test.o

2.2.2 GCC编译流程分析

链接阶段

- 输入：二进制机器代码文件*.o
- 输出：可执行的二进制代码文件
- 格式：gcc –o [目标文件] [编译文件]

  或　gcc [编译文件]　-o　[目标文件]
- 实例：gcc test.o –o test
- $./test　　//执行可执行程序

2.2.2 GCC编译流程分析

- 多文件编译
  - 格式1：多文件同时编译
  - gcc 1.c 2.c 3.c –o test
  - $./test
  - 格式2：每个文件分别进行编译，然后链接成可执行文件
  - gcc –c 1.c –o 1.o
  - gcc –c 2.c –o 2.o
  - gcc –c 3.c –o 3.o
  - gcc 1.o 2.o 3.o –o test
  - $./test　　//执行可执行程序

多文件编译实例

- 建立文件夹duofile，里面建立三个文件
- other1.c

```
void welcome()
{
 printf("Welcome to compile multiple files!\n");
}
```

- Other2.c

```
int add(int x, int y)
{  return x+y;}
int sub(int x, int y)
{  return x-y;}
```

多文件编译实例

- 建立文件夹duofile，里面建立三个文件
- maintest.c

```
#include <stdio.h>
void main()
{  int a=15,b=3,c;
   printf(“test multiple file compile!”);
   welcome();
  c=add(a,b);
  printf("%d+%d=%d\n",a,b,c);
  c=sub(a,b);
  printf("%d-%d=%d\n",a,b,c);
}
```

多文件编译实例

```
li@li-desktop:~/gaojiOS/linshi$ gcc other2.c other1.c maintest1.c -o b
other1.c: In function 'welcome':
other1.c:3: warning: incompatible implicit declaration of built-in function '
printf'
li@li-desktop:~/gaojiOS/linshi$ ls
b  maintest1.c  other1.c  other2.c
li@li-desktop:~/gaojiOS/linshi$ ./b
test multiple file compile!
Welcome to compile multiple files!
15+3=18
15-3=12
li@li-desktop:~/gaojiOS/linshi$
```

为other1.c和other2.c增加头文件

- other1.h

```
#ifndef other1_h
#define other1_h
#include <stdio.h>
void welcome();
#endif
```

- Other2.h

```
#ifndef other2_h
#define other2_h
#include <stdio.h>
int add(int, int);
int sub(int, int);
#endif
```

修改other1.c和other2.c

- other1.c

```
#include “other1.h”
void welcome()
{
 printf("Welcome to compile multiple files!\n");
}
```

- Other2.c

```
#include “other2.h”
int add(int x, int y)
{  return x+y;}
int sub(int x, int y)
{  return x-y;}
```

修够maintest.c内容

- maintest.c

```
#include <stdio.h>
#include “other1.h”
#include “other2.h”
void main()
{  int a=15,b=3,c;
   printf(“test multiple file compile!”);
   welcome();
  c=add(a,b);
  printf("%d+%d=%d\n",a,b,c);
  c=sub(a,b);
  printf("%d-%d=%d\n",a,b,c);
}
```

编译结果

```
li@li-desktop:~/gaojiOS/example1$ ls
maintest1.c  other1.c  other1.h  other2.c  other2.h
li@li-desktop:~/gaojiOS/example1$ gcc maintest1.c other1.c other2.c -o b
li@li-desktop:~/gaojiOS/example1$ ./b
test multiple file compile!
Welcome to compile multiple files!
15+3=18
15-3=12
li@li-desktop:~/gaojiOS/example1$
```

头文件与源文件在一个目录下，可以正确编译执行。

将头文件放到一个指定子目录head下

```
li@li-desktop:~/gaojiOS/example1$ ls
maintest1.c  other1.c  other1.h  other2.c  other2.h
li@li-desktop:~/gaojiOS/example1$ mkdir head
li@li-desktop:~/gaojiOS/example1$ mv *.h head
li@li-desktop:~/gaojiOS/example1$ ls
head  maintest1.c  other1.c  other2.c
li@li-desktop:~/gaojiOS/example1$ gcc maintest1.c other1.c other2.c -o b
maintest1.c:1:20: error: other1.h: 没有那个文件或目录
maintest1.c:2:20: error: other2.h: 没有那个文件或目录
other1.c:1:20: error: other1.h: 没有那个文件或目录
other1.c: In function ‘welcome’:
other1.c:4: warning: incompatible implicit declaration of built-in function
printf’
other2.c:1:20: error: other2.h: 没有那个文件或目录
li@li-desktop:~/gaojiOS/example1$
```

编译报错，找不到头文件

报错原因

- 编译器gcc默认在当前目录和/usr/include目录下查找相应的头文件，如果在这两个目录里找不到相应的头文件则报错。
- 上例中，头文件都放在了当前目录的子目录head里，因此找不到，报错。

GCC指定头文件目录

- GCC使用缺省的路径来搜索头文件，如果想要改变搜索路径，用户可以使用“-I”选项。“-I<dir>”选项可以在头文件的搜索路径列表中添加<dir>目录。这样，GCC就会到指定的目录去查找相应的头文件。

```
li@li-desktop:~/gaojiOS/example1$ ls
head  maintest1.c  other1.c  other2.c
li@li-desktop:~/gaojiOS/example1$ gcc -I./head maintest1.c other1.c other2.c
-o b
li@li-desktop:~/gaojiOS/example1$ ./b
test multiple file compile!
Welcome to compile multiple files!
15+3=18
15-3=12
li@li-desktop:~/gaojiOS/example1$
```

Linux库的创建与使用-1

- 库：事先已经编号的代码，经过编译后可以直接调用的文件，本质上来说是一种可执行代码的二进制形式，可以被操作系统载入内存执行。
- 系统提供的库所在目录：/usr/lib
- Linux库文件名得组成
- 前缀lib库名后缀（.a（静态库）或.so（动态库））
- 例：libmm.a是库名为mm的静态库
- libnn.so是库名为nn的动态库

Linux库的创建与使用-2

- 静态库与动态库的载入顺序是不一样的
  - 静态库的代码在编译时就拷贝到应用程序中，因此当有多个程序同时引用一个静态库函数时，内存中将会有调用函数的多个副本。由于是完全拷贝，因此一旦连接成功，静态程序库就不再需要了，代码体积大。
  - 动态库：在执行程序内留下一个标记，指明当程序执行时，首先必须载入这个库。在程序开始运行后调用库函数时才被载入，被调用函数在内存中只有一个副本，代码体积小。

静态库的创建-1

- 创建步骤
  - 在一个头文件中声明静态库所导出的函数
  - 在一个源文件中实现静态库所导出的函数
  - 编译源文件，生成目标文件（*.o)
  - 通过命令ar将目标文件加入到某个静态库中

    ar　rcs　静态库名　目标文件列表
  - 将静态库拷贝到系统默认的存放库文件的目录或指定目录下

静态库创建实例-2

- 还是使用前面讲解的other1.h, other1.c, other2.h, other2.c, maintest.c
- 编译生成目标代码

  gcc –o other1.o –c other1.c

  gcc –o other2.o –c other2.c
- 将目标文件加入到静态库中

  ar rcs libother1.a other1.o

  ar rcs libother2.a other2.o
- 将静态库拷贝到系统默认的存放库文件的目录

  sudo cp libother1.a libother2.a /usr/lib/

静态库的使用-3

- 准备工作：在上述文件夹中删除other1.h,other1.c, other2.h, other2.c
- gcc 使用库文件的参数
  - -lname：指示编译器，在链接时，装载名为libname.a的函数库，该函数库位于系统定义的目录或由-L选项选项指定的目录下。例如，-lm表示链接名为libm.a的数学函数库。
  - -L <dir>”

    选项“-L <dir>”的功能是用于指明库文件的路径。
- gcc –o maintest maintest.c –lother1 –lother2
- ./maintest //可以正常执行

静态库的创建-4

- 可以将多个*.o文件创建一个静态库
- ar rcs libmylib.a other1.o other2.o
- sudo cp libmylib.a /usr/lib/
- gcc –o maintest maintest.c –lmylib
- ./maintest

静态库的创建-5

- 将生成的静态库放在自己建立的文件夹lib中
- mkdir lib
- ar rcs libmylib.a other1.o other2.o
- cp libmylib.a ./lib
- gcc –o maintest maintest.c –lmylib –L./lib
- ./maintest

动态库的创建

- 创建：gcc –fPIC –shared –o lib**.so ***.c
- 实例：gcc –fPIC –shared –o libtt.so other1.c other2.c
- 使用：gcc –o *** **.c
- 实例：gcc –o maintest maintest.c ./libtt.so
- ./maintest　　//正确执行
- 当删除libtt.so后执行./maintest将出错

2.3 调试器GDB的使用

- gdb是GNU开源组织发布的一个强大的Linux下的程序调试工具。
- gdb和其他调试器一样，可以在程序中设置断点、查看变量值，一步一步地跟踪程序的执行过程。
- 利用调试器的这些功能可以方便地找出程序中存在的非语法错误。

启动和退出gdb

- gdb调试对象：可执行程序，并且在gcc编译时必须加上-g选项。
- 实例：编写一个用于调试test.c文件

```
1 #include <stdio.h>
2
3 int get_sum(int n)
4 {
5   int sum=0,i;
6   for(i=0; i<n; i++)
7      sum+=i;
8   return sum;
9 }
10
11 int main( )
12 {
13   int i=100,result;
14   result=get_sum(i);
15   printf("1+2+...+%d=%d\n",i,result);
16   return 0;
17 }
```

编译并运行该程序

$ gcc –g test.c –o test

$ ./test

输出：1+2+…+100=4950

程序有错误，正确结果是5050

gdb的启用和退出

- gdb 程序名

  例如：gdb test

  启动gdb后，首先是一段版权说明，然后是gdb的提示符：(gdb)，可以在(gdb)之后输入调试命令

  若不想显示版权说明，可以加一个-q选项。

- 首先启动gdb –g，然后用file命令加载要调试的程序

  gdb –g

  file test

- gdb的退出

  (gdb)quit

显示程序源代码List

- list
- 输出从上次调用list命令开始往后的10行程序代码
- 显示10行代码，若再次运行该命令则显示接下来的10行代码
- list -（减号）：输出从上次调用list命令开始往前的10行代码
- list n：输出n行附近的10行代码
- List function：输出函数funciton附近的10行代码
- List 5，10：显示第5到第10行的代码
- List test.c：5，10：显示源文件test.c中的第5到第10行代码
- List test.c: function:显示源文件test.c中funciton附近的代码。

搜索字符串-1

- forward/search：从当前行向后查找某个字符串的程序行，查找时不包括当前行，可以用list n,n,将当前行设置为n。
- forward 字符串或search 字符串
- 例: (gdb) list 1,1

  (gdb) search get_sum

  结果：3 int get_sum(int n)

  (gdb) list 3,3

  (gdb) forward get_sum

  结果：14 result=get_sum(i);

搜索字符串-2

- reverse-search 字符串：从当前行向前查找第一个匹配的字符串。
- 例: (gdb) list 3,3

  (gdb) reverse-search get_sum

  结果：Expression not found

  (gdb) list 4,4

  (gdb) reverse-search get_sum

  结果：3 int get_sum(int n)

执行程序和获得帮助

- 使用gdb test或file test 只是装入程序，程序并没有运行
- 运行
- (gdb) run

  结果：starting program: 文件路径

  1+2+…+100=4950

  program exited normally
- 帮助命令：help 命令名

例：help list

3.3.4 设置和管理断点

- 1、以行号设置断点
- 格式：break n
- 功能：当程序运行到指定行时，会暂停执行，指定行的代码不执行。
- 例：(gdb) break 15

反馈：Breakpoint 1 at ox8048432: file test.c, line 15

- (gdb) run

结果：starting program: /home/lyj/test

Breakpoint 1， main( ) at test.c 15

15 printf(“1+2+…+%d=%d\n”,i,result);

3.3.4 设置和管理断点

- 2、以函数名设置断点
- 格式：break 函数名
- 例：(gdb) break get_sum

反馈：Breakpoint 1 at ox8048432: file test.c, line 5

- (gdb) run

3.3.4 设置和管理断点

- 3、以条件表达式设置断点
- 格式：break 行号或函数名 if 条件
- 功能：程序在运行过程中，当某个条件满足时，程序在某行中断暂停执行
- 例：(gdb) break 7 if i==99

含义：当程序执行到第7行时，判断条件i==99是否成立，若成立则暂停。

3.3.4 设置和管理断点

- 4、以条件表达式变化设置断点
- 格式：watch 条件表达式
- 功能：程序在运行过程中，当某个条件满足时，程序在某行中断暂停执行
- 注意：watch必须在程序运行的过程中设置观察点，即run之后才能设置，并且要保证条件表达式中的变量已经使用过。
- 例：(gdb) break 7 if i==99

含义：当程序执行到第7行时，判断条件i==99是否成立，若成立则暂停。

3.3.4 设置和管理断点

- 4、以条件表达式变化设置断点
- 格式：watch 条件表达式
- 例1：

```
(gdb) break 13
(gdb) run
(gdb) watch sum==3
No symbol “sum” in current context
```

- 例2

```
(gdb) break 5
(gdb) run
(gdb) watch sum==3
```

3.3.4 设置和管理断点

- 5、查看当前设置的断点
- 格式：info breakpoints
- 例1：

```
(gdb) break 7
(gdb) break 15 if result==5050
(gdb) info breakpoints
```

结果：

```
1 breakpoint keep y 0x080483fa  in get_sum at test.c:7
2 breakpoint keep y 0x08939324 in main at test.c 15 stop only
  if result==5050
```

keep:生效一次后不失效

y：是否有效

3.3.4 设置和管理断点

6、使中断失效或有效

（1）失效：disable 断点编号

（2）有效：enable 断点编号

7、删除断点

（1）clear：删除程序中所有断点

（2）clear 行号：删除此行断点

（3）delete 断点编号：删除指定编号的断点，若一次要删除多个断点，各个断点编号以空格隔开。

实例

```
(gdb) break 6
(gdb) break 7
(gdb) break 8 if sum==5050
(gdb) clear
```

3.3.5 查看和设置变量的值

- 当程序执行到中断点暂停执行时，往往要查看变量或表达式的值，借此了解程序的执行状态，进而发现问题所在
- 1、print命令
  - 功能：打印变量或表达式的值，还可以用来对某个变量进行赋值。
  - print 变量或表达式：打印变量或表达式的值
  - print 变量=值：对变量进行赋值

3.3.5 查看和设置变量的值

- 1、print命令
- 实例：
  - (gdb) break 7
  - (gdb) run
  - (gdb) print i<n　$1=1
  - (gdb) print i　$2=0
  - (gdb) print sum　$3=0
  - (gdb) print i=200　$4=200
  - (gdb)continue

continuting

1+2+…+100=200

program exited normally

3.3.5 查看和设置变量的值

2、what is 命令

- 功能：用于显示某个变量或表达式的数据类型。
- 格式：what is 变量或表达式
- 实例

```
(gdb) break 7
(gdb) run
(gdb) what is I
(gdb) type=int
(gdb) what is sum+0.5
(gdb)type=double
```

3.3.5 查看和设置变量的值

3、set 命令

- 功能：给变量赋值。
- 格式：set variable 变量=值

3.3.6 控制程序的执行

- 当程序执行到指定的中断点，查看了变量或表达式的值后，可以让程序一直运行下去直到下一个断点或运行完为止。
- 1、continue

程序继续运行，直到下一个断点或运行完为止

- 2、kill

结束当前程序的调试

- 3、next和step

功能：一次一条执行该程序代码

区别：next把函数调用当作一条语句来执行

step跟踪进入函数，一次一条地执行函数内的代码

3.4 make的使用和makefile文件的编写

- make：用来维护程序模块关系和生成可执行程序的工具。
- make是一个命令，需要一个makefile或Makefile文件。

3.4.1 makefile文件的基本构成

- 一个简单的makefile文件

```
main: main.o module1.o module2.o
        gcc main.0 module1.o module2.o –o main
main.o: main.c head1.h head2.h common_head.h
        gcc –c main.c
module1.o: module1.c head1.h
        gcc –c module1.c
module2.o: module2.c head2.h
        gcc –c module2.c
```

3.4.2 make执行过程

(1)第一次执行

$make

执行后输出：

gcc –c main.c

gcc –c module1.c

gcc –c module2.c

gcc main.o module1.o module2.o –o main

3.4.2 make执行过程

(1)部分修改后执行

假设修改了头文件head1.h

$make

执行后输出：

gcc –c main.c

gcc –c module1.c

gcc main.o module1.o module2.o –o main

3.4.3 其他说明

- 命令之间可以插入任意多个空行，空行也要按tab键开头
- 若某一行过长，可以在达到这一行行末前插入一个反斜杠(\)。由反斜杠连接起来的多行都被当作一行来处理。
- makefile可以命名为Makefile。
- 可以用其他文件名代替makefile，按如下方式执行

  $make –f othername

- 把最后要生成的文件放在第一条规则的目标文件列表的第一个位置上