第 3 章　Linux 系统编程基础

本章首先介绍 GCC 编译器的编译过程及常用选项的使用，通过实例讲述 GDB 调试器的使用方法，然后介绍 Make 工具的使用，最后介绍文件操作、时间获取和创建线程等任务的编程方法。

3.1　GCC 编译器

3.1.1　GCC 概述

GCC(GNU C Compiler)是 GUN 项目的 C 编译器套件，也是 GNU 软件家族中具有代表性的产品之一。GCC 目前支持的体系结构有四十余种，如 x86、ARM、PowerPC 等系列处理器；能运行在不同的操作系统上，如 Linux、Solaris、Windows CE 等操作系统；可完成 C、C++、Objective C 等源文件向运行在特定 CPU 硬件上的目标代码的转换。GCC 的执行效率与一般的编译器相比平均效率要高 20%～30%。GCC 是 Linux 平台下最常用的编译器之一，它也是 Linux 平台编译器事实上的标准。同时，在使用 Linux 操作系统的嵌入式开发领域，GCC 也是使用最普遍的编译器之一。

GCC 编译器与 GUN Binutils 工具包是紧密集成的，如果没有 Binutils 工具，GCC 也不能正常工作。Binutils 是一系列开发工具，包括连接器、汇编器和其他用于目标文件和档案的工具。Binutils 工具集里主要包含以下一系列程序：addr2line、ar、as、c++、gprof、ld、nm、objcopy、objdump、ranlib、readelf、size、strings 和 strip，它包含的库文件有：libiberty. a、libbfd. a、libbfd. so、libopcodes. a 和 libopcodes. so。

在 Linux 操作系统中，文件的后缀名不代表文件的类型，但为了提高工作效率，通常会给每种文件定义一个后缀名。GCC 支持的文件类型比较多，具体如表 3.1 所示。

表 3.1　GCC 支持的文件类型

后　缀	说　明	后　缀	说　明
. c	C 源程序	. ii	经过预处理的 C++程序
. a	由目标文件构成的档案文件(库文件)	. m	Objective C 源程序
. C　. cc	C++源程序	. o	编译后的目标程序
. h	头文件	. s	汇编语言源程序
. i	经过预处理的 C 程序	. S	经过预编译的汇编程序

3.1.2　GCC 编译过程

下面通过一个常用的例子来说明 GCC 的编译过程。

利用文本编辑器创建 hello. c 文件，程序内容如下。

```
#include<stdio.h>
void main()
{
    char msg[80]="Hello,world!";
    printf("%s\n",msg);
}
```

编写完后,执行以下编译指令。

```
#gcc hello.c
```

因为编译时没有加任何选项,所以会默认生成一个名为 a.out 的可执行文件。执行该文件的命令及结果如下。

```
#./a.out
    Hello,world!
```

使用 GCC 由 C 语言源代码程序生成可执行文件要经历 4 个过程,如图 3.1 所示。

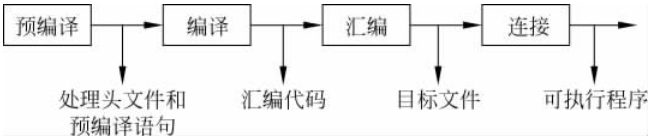

图 3.1　GCC 编译过程

1. 预编译

预编译(preprocessing)的主要功能是读取源程序,并对头文件(include)、预编译语句(如 define 等)和一些特殊符号进行分析和处理。如把头文件复制到源文件中,并将输出的内容送到系统的标准输出。源代码中的预编译指示以“#”为前缀。通过在 GCC 后加上-E 选项完成对代码的预编译。命令如下。

```
# gcc -E hello.c
```

执行命令时,控制台上会有数千行的输出,其中大多数来自 stdio.h 头文件,也有部分是声明。预编译主要完成以下 3 个具体任务。

(1) 把 include 中的头文件复制到要编译的源文件中。

(2) 用实际值替代 define 文本。

(3) 在调用宏的地方进行宏替换。

下面通过实例 test.c 来理解预编译完成的工作。

test.c 的代码如下。

```
#define number (1+2*3)
int main()
{
    int n;
    n=number+3;
    return 0;
}
```

对 test.c 文件进行预编译，输入以下命令。

```
#gcc -E test.c
```

执行命令后会显示如下内容。

```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "test.c"

main()
{
    int n;
    n=(1+2*3)+3;
    return 0;
}
```

如果要将预编译结果保存在 test.i 文件中，可以输入以下命令。

```
#gcc -E test.c -o test.i
```

2. 编译

编译(compilation)的主要功能包括两部分，第一部分是检查代码的语法，如果出现语法错误，则给出错误提示代码，并结束编译；只有在代码无语法错误的情况下，才能进入第二部分。第二部分是将预编译后的文件转换成汇编语言，并自动生成后缀为.s 的文件。编译的命令如下。

```
#gcc -S test.c
```

执行命令后会生成一个名为 test.s 的汇编程序，文件内容如下。

```
    .file"test.c"
    .text
    .align 2
.globl main
    .typemain,@function
main:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $8, %esp
    andl    $-16, %esp
    movl    $0, %eax
    subl    %eax, %esp
    movl    $10,-4(%ebp)
    movl    $0, %eax
    leave
    ret
.Lfe1:
    .size   main,.Lfe1-main
    .ident  "GCC: (GNU) 3.2 20020903 (Red Hat Linux 8.0 3.2-7)"
```

3. 汇编

汇编(assembly)的主要功能是将汇编语言代码变成目标代码(机器代码)。汇编只是将

汇编语言代码转换成目标代码,但不进行连接,目标代码不能在 CPU 上运行。汇编使用选项为-c,它会自动生成一个后缀名为.o 的目标程序。汇编的命令如下。

```
#gcc -c test.c
```

执行命令后会生成一个名为 test.o 的目标文件,目标文件是一个二进制文件,所以不能用文本编辑器来查看它的内容。

4. 连接

连接(linking)的主要功能是连接目标代码,并生成可执行文件。连接的命令如下。

```
#gcc -o test test.o
```

也可以执行以下命令。

```
#gcc -o test test.c
```

执行命令后会生成一个名为 test 的可执行文件。通过执行./test 命令,就可以运行指定的程序。命令中的“./”是指在当前目录下执行程序。

3.1.3 GCC 选项

GCC 编译器提供了较多的选项。选项必须以“-”开始,常用的选项如表 3.2 所示。

表 3.2　GCC 的常用选项

选　　项	说　　明
-c	编译生成目标文件,后缀为.o
-E	只进行预编译,不做其他处理
-g	在执行程序中包括标准调试信息
-I DirName	将 DirName 加入到头文件的搜索目录列表中
-L DirName	将 DirName 加入到库文件的搜索目录列表中,在默认情况下 gcc 只链接共享库
-l FOO	链接名为 libFOO 的函数库
-O	整个源代码会在编译、连接过程中进行优化处理,可执行文件的执行效率可以提高,但是编译、连接的速度就相应的要慢些
-O2	比-O 有更好的优化能力,但编译连接速度就更慢
-o FileName	指定输出文件名,如果没有指定,默认文件名是 a.out
-pipe	在编译过程的不同阶段间使用管道
-S	只编译不汇编,生成汇编代码
-static	链接静态库
-wall	指定产生全部的警告信息

1. 输出文件选项

如果不使用任何选项进行编译,生成的可执行文件都是 a.out。如果要指定输出的文件名,可以使用选项-o。例如将源文件 hello.c 编译成可执行文件 hello。命令格式如下。

```
#gcc -o hello hello.c
```

2. 链接库文件选项

Linux 操作系统下的库文件包括两种格式，一种是动态链接库，另一种是静态链接库。动态链接库的后缀为.so，静态链接库的后缀为.a。动态链接库是在程序运行过程中进行动态加载，静态链接库是在编译过程中完成静态加载。

使用 GCC 编译时，编译器会自动调用 C 标准库文件，但当要使用到标准库以外的库文件时，一定要使用选项-l 来指定具体库的文件名，否则会报编译错误，如报 undefined reference to 'xxxx'错误。Linux 操作系统下的库文件都是以 lib 三个字母开头的，因此在使用-l 选项指定链接的库文件名时可以省去 l、i、b 三个字母。例如，有一个多线程程序 pthread.c，需要用到 libpthread.a 或 libpthread.so 库文件（文件保存在/usr/lib 目录）。编译生成一个名为 pthread 的可执行程序的命令格式如下。

```
#gcc pthread.c -lpthread -o pthread
```

GCC 在默认情况下，优先使用动态链接库，当需要强制使用静态链接库时，需要加上-static 选项。使用静态链接库，编译生成一个名为 pthread-s 的可执行程序的命令格式如下。

```
#gcc pthread.c -static -lpthread -o pthread-s
```

可以使用 ls -l 命令查看文件的大小，会发现 pthread-s 比 pthread 文件大很多。

```
-rwxr-xr-x 1 root root 12014 12 月 28 22:22 pthread
-rwxr-xr-x 1 root root 589856 12 月 28 22:22 pthread-s
```

3. 指定头文件目录选项

编译时，编译器会自动到默认目录（一般为/usr/include）寻找头文件，但当文件中的头文件不在默认目录时，就需要使用-I 选项来指定头文件所在的目录（或称文件所在的路径）。如果不指定头文件所在的目录，编译时会报 xxx.h：No such file or directory 错误。假设 someapp.c 程序中有一个自定义的头文件放置在/usr/local/include/someapp 目录下，则命令格式如下。

```
#gcc -I /usr/local/include/someapp -o someapp someapp.c
```

4. 指定库文件目录选项

编译时，编译器会自动到默认目录（一般为/usr/lib）寻找库文件，但当编译时所用的库文件不在默认目录时，就需要使用-L 选项来指定库文件所在的目录。如果不指定库文件所在的目录，编译时会报 cannot find lxxx 错误。假设程序 my.c 需要使用 libnew.so 库文件，且该库文件保存在/home/someuser/lib 目录，则命令格式如下。

```
#gcc my.c -L/home/someuser/lib -lnew -o my
```

5. 警告选项

在编译过程中，编译器的警告信息对于程序员来说是非常重要的，GCC 包含完整的警告提示功能，以便确定代码是否正确，尽可能实现可移植性。GCC 的编译器警告选项如表 3.3 所示。

表 3.3　GCC 的警告选项

类　　型	说　　明
-Wall	启用所有警告信息
-Werror	在发生警告时取消编译操作,即将警告看作是错误
-w	禁用所有警告信息

下面通过一实例来了解如何在编译时产生警告信息。example.c 的代码如下。

```
#include<stdio.h>
int main ()
{
  int x,y;
  for(x=1;x<=5;x++)
      printf("x=%d\n",x);
}
```

使用以下命令进行编译。

```
#gcc example.c
```

编译过程没有任何提示信息,生成一个 a.out 可执行文件。

如果加入-Wall 选项进行编译,命令如下。

```
# gcc -Wall example.c -o example
```

编译过程将会出现下面的警告信息。

```
example.c: In function 'main':
example.c:4: warning: unused variable 'y'
example.c:7: warning: control reaches end of non-void function
```

第 1 条警告信息的意思是:在 main 函数有警告信息。

第 2 条警告信息的意思是:指出变量 y 在程序中未使用。

第 3 条警告信息的意思是:main 函数的返回类型是 int,但在程序中没有 return 语句。

GCC 给出的警告从严格意义上不算错误,但是可能会成为错误的栖息之地。所以在嵌入式软件开发时,需要重视警告信息,最好根据警告信息对源程序进行修改,直至编译时没有任何警告信息。

-Werror 选项会要求 GCC 将所有的警告信息当成错误进行处理,需要将所有的警告信息都修改后才能生成可执行文件。命令如下。

```
#gcc -Wall -Werror example.c -o example
```

当需要忽略警告信息时,可以使用-w 选项,命令如下。

```
#gcc -w example.c -o example
```

6. 调试选项

代码通过了编译并不代表能正常工作。可以通过调试器检查代码,以便更好地找到程序中的问题。Linux 下主要采用的是 GDB 调试器。在使用 GDB 之前,在执行程序中要包

括标准调试信息，加入的方法是采用调试选项-g。具体的命令如下。

```
#gcc -g -c hello.c
#gcc -g -o hello hello.o
```

7. 优化选项

优化选项的作用在于缩减代码规模和提高代码执行效率，常用的选项有以下几个。

(1) -O、-O1：整个源代码会在编译、连接过程中进行优化处理，可执行文件的执行效率可以提高，但是编译、连接的速度会相应慢些。对于复杂函数，优化编译会占用较多的时间和相当大的内存。在-O1 下，编译会尽量减少代码的体积和代码的运行时间，但是并不执行会花费大量时间的优化操作。

(2) -O2：除了不涉及空间和速度交换的优化选项，执行几乎所有的优化工作。比-O 有更好的优化效果，但编译连接速度更慢。-O2 将会花费更多的编译时间同时也会生成性能更好的代码。但并不执行循环展开和函数“内联”优化操作。

(3) -O3：在-O2 的基础上加入函数内联、循环展开和其他一些与处理器特性相关的优化工作。

下面通过 optimize.c 程序观察一下优化前后的效果。

```
#include <stdio.h>
int main(void)
{
    double counter;
    double result;
    double temp;
    for (counter=0;counter<2000.0*2000.0*2000.0/20.0+2020; counter += (5-1)/4)
    {
        temp = counter / 1979;
        result = counter;
    }
    printf("Result is %lf\\n", result);
    return 0;
}
```

不加优化选项进行编译，程序执行耗时如下。

```
# gcc optimize.c -o optimize
# time ./optimize
Result is 400002019.000000\n
real    0m4.203s
user    0m4.190s
sys     0m0.020s
```

增加优化选项进行编译，程序执行耗时如下。

```
# gcc -O1 optimize.c -o optimize1
# time ./optimize1
Result is 400002019.000000\n
real    0m1.064s
user    0m1.060s
sys     0m0.010s
```

3.2　GDB 调试器

应用程序的调试是开发过程中必不可少的环节之一。Linux 下的 GNU 的调试器称为 GDB(GUN Debugger),该软件最早由 Richard Stallman 编写。GDB 是一个用来调试 C 和 C++语言程序的调试器,它能使开发者在程序运行时观察程序的内部结构和内存的使用情况。GDB 主要可以完成下面 4 个方面的功能。

(1) 启动程序,按照程序员自定义的要求运行程序。

(2) 单步执行、设置断点,可以让被调试的程序在所指定的断点处停住。

(3) 监视程序中变量的值。

(4) 动态地改变程序的执行环境。

3.2.1　GDB 的基本使用方法

下面通过一个例子 test.c 介绍 GDB 的基本使用方法,test.c 文件的代码如下。

```
#include<stdio.h>
int sum(int n);
main()
{
    int s=0;
    int i,n;
    for(i=0;i<=50;i++)
    {
        s=i+s;
    }
    s=s+sum(20);
    printf("the result is %d\n",s);
}
int sum(int n)
{
    int total=0;
    int i;
    for(i=0;i<=n;i++)
        total=total+i;
    return (total);
}
```

使用 GDB 调试器,必须在编译时加入调试选项-g,命令如下。

```
#gcc -g test.c -o test
#gdb test                    <-------进入 gdb 调试环境
GNU gdb Red Hat Linux(5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) l                    <-------相当于 list,查看源代码
1   #include<stdio.h>
2   int sum(int n);
3   main()
4   {
5       int s=0;
6       int i,n;
7       for(i=0;i<=50;i++)
8       {
9           s=i+s;
10      }
(gdb) l
11      s=s+sum(20);
12      printf("the result is %d\n",s);
13  }
14  int sum(int n)
15  {
16    int total=0;
17    int i;
18    for(i=0;i<=n;i++)
19          total=total+i;
20  return (total);
(gdb) l
21  }
(gdb) break 7              <-------在源代码第 7 行设置断点
Breakpoint 1 at 0x804833f: file test.c,line 7.
(gdb) break sum            <-------在源代码 sum 函数处设置断点
Breakpoint 2 at 0x804838a: file test.c,line 16.
(gdb) info break           <-------显示断点信息
Num Type         Disp Enb Address        What
1   breakpoint   keep y   0x0804833f    in main at test.c:7
2   breakpoint   keep y   0x0804838a    in main at test.c:16
(gdb) r                    <-------运行程序
Staring program: /lvli/test
Breakpoint 1, main () at test.c:7
7                   for(i=0;i<=50;i++)
(gdb) n                    <-------在第一个断点处停止,n 相当于 next,单步执行
9                           s=i+s;
(gdb) n
7                   for(i=0;i<=50;i++)
(gdb) print s              <-------输出变量 s 的值
$1 = 0
(gdb) c                    <-------相当于 continue,继续执行
Continuing.
Breakpoin 2 ,sum(n=20) at test.c:16
16                   int total=0;
(gdb) c
Continuing.
the result is 1485
```

```
Program exited with code 024.
(gdb) q                    <-------退出 gdb
```

3.2.2　GDB 基本命令

GDB 命令很多,可以通过 help 来帮助,方法是:在启动 GDB 后,输入 help 命令。

```
(gdb)help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

因为 GDB 命令有很多,所以将它们分成许多种类。help 命令只列出了 GDB 的命令种类,如果要查看某一种类下的具体命令,可以在 help 命令后加类名,具体格式如下。

```
help <class>
```

例如想了解 running 类下的具体命令,可以输入以下命令。

```
help running
```

常用的 GDB 命令如表 3.4 所示。

表 3.4　GDB 常用命令描述

命　　令	描　　述
backtrace	显示程序中的当前位置和表示如何到达当前位置的栈跟踪
break	设置断点
cd	改变当前工作目录
clear	清除停止处的断点
continue	从断点处开始继续执行
delete	删除一个断点或监测点
display	程序停止时显示变量或表达式
file	装入要调试的可执行文件
info	查看程序的各种信息
kill	终止正在调试的程序
list	列出源文件内容

续表

命　令	描　述
make	使用户不退出 GDB 就可以重新产生可执行文件
next	执行一行代码，从而执行其整体的一个函数
print	显示变量或表达式的值
pwd	显示当前工作目录
quit	退出 GDB
run	执行当前被调试的程序
set	给变量赋值
shell	不退出 GDB 就执行 UNIX shell 命令
step	执行一行代码且进入函数内部
watch	设置监视点，使用户能监视一个变量或表达式的值而不管它何时变化

3.2.3　GDB 典型实例

下面的程序中植入了错误，通过这个存在错误的程序掌握如何利用 GDB 进行程序调试。

有一个 bug.c 程序，它的功能是将输入的字符串逆序显示在屏幕上，源代码如下。

```
#include<stdio.h>
#include<string.h>
int main(void)
{
    int i,len;
    char str[]="hello";
    char *rev_string;
    len=strlen(str);
    rev_string=(char *)malloc(len+1);
    printf("%s\n",str);
    for(i=0;i<len;i++)
        rev_string[len-i]=str[i];
    rev_string[len+1]='\0';
    printf("the reverse string is%s\n",rev_string);
}
```

程序的编译和运行结果如下。

```
# gcc -o bug bug.c
# ./bug
hello
the reverse string is
```

以上运行的结果是错误的，正确结果如下。

```
hello
the reverse string is olleh
```

这时，可以使用 GDB 调试器来查看问题在哪儿。具体步骤是：编译时加上-g 调试选

项,然后再对可执行程序进行调试,命令如下。

```
# gcc -g -o bug bug.c
# gdb bug
```

执行命令后,进入调试环境,显示如下。

```
(gdb) l                          <-------列出源文件内容
1          #include<stdio.h>
2          #include<string.h>
3          int main(void)
4          {
5                    int i,len;
6                    char str[]="hello";
7                    char *rev_string;
8                    len=strlen(str);
9                    rev_string=(char *)malloc(len+1);
10                   printf("%s\n",str);
(gdb) l
11                   for(i=0;i<len;i++)
12                            rev_string[len-i]=str[i];
13                   rev_string[len+1]='\0';
14                   printf("the reverse string is%s\n",rev_string);
15         }
(gdb) break 8                    <-------在源代码第 8 行设置断点
Breakpoint 1 at 0x80483b2: file example.c, line 8.
(gdb) r                          <-------运行程序
Starting program: /lvli/program/bugg/bug
Breakpoint 1, main () at bug.c:8
8                    len=strlen(str);
(gdb) n                          <-------在第一个断点处停止,n 相当于 next,单步执行
9                    rev_string=(char *)malloc(len+1);
(gdb) print len                  <-------输出变量 len 的值
$1 = 5
(gdb) n                          <-------单步执行
10                   printf("%s\n",str);
(gdb) n
hello
11                   for(i=0;i<len;i++)
(gdb) n
12                            rev_string[len-i]=str[i];
(gdb) n
11                   for(i=0;i<len;i++)
(gdb) n
12                            rev_string[len-i]=str[i];
(gdb) n
11                   for(i=0;i<len;i++)
(gdb) print rev_string[5]        <-------输出 rev_string[5]的值
$2 = 104 'h'
(gdb) print rev_string[4]        <-------输出 rev_string[4]的值
$3 = 101 'e'
```

```
(gdb) n
12                          rev_string[len-i]=str[i];
(gdb) n
11                  for(i=0;i<len;i++)
(gdb) print rev_string[3]
$4 = 108 'l'
(gdb) n
12                          rev_string[len-i]=str[i];
(gdb) n
11                  for(i=0;i<len;i++)
(gdb) print rev_string[2]
$5 = 108 'l'
(gdb) n
12                          rev_string[len-i]=str[i];
(gdb) n
11                  for(i=0;i<len;i++)
(gdb) print rev_string[1]
$6 = 111 'o'
(gdb) n
13                  rev_string[len+1]='\0';
(gdb) n
14                  printf("the reverse string is%s\n",rev_string);
(gdb) print rev_string[0]      <-------输出 rev_string[0]的值
$7 = 0 '\0'
(gdb) c                        <-------相当于 continue,继续执行
Continuing.
the reverse string is
```

通过以上调试过程可见,错误的根源在于没有给 rev_string[0]赋值,所以 rev_string[0]为'\0',导致字符串输出为空。可以将 rev_string[len-i]改成 rev_string[len-1-i],这样结果就是期待的结果。

3.3　Make 工具的使用

在大型软件项目的开发过程中,通常有成百上千个源文件,如 Linux 内核源文件。如果每次都通过手工输入 GCC 命令进行编译,非常不方便,所以引入了 Make 工具来解决这个问题。Make 工具可以将大型的开发项目分解成为多个更易于管理的模块,简洁明了地理顺各个源文件之间纷繁复杂的相互依赖关系,最后自动完成编译工作。

Make 工具最主要的作用是通过 Makefile 文件来描述源程序之间的相互关系,并自动完成维护编译工作。Makefile 文件需要严格按照语法进行编写,文件中需要说明如何编译各个源文件并连接生成可执行文件,并定义源文件之间的依赖关系等。

3.3.1　Makefile 的基础知识

1. Makefile 文件

Makefile 是描述文件依赖关系的说明,它由若干个规则组成,每个规则的格式如下。

目标：依赖关系

＜tab 键＞ 命令

其中：目标是指 make 最终需要创建的东西。另外，目标也可以是一个 make 执行的动作名称，如目标 clean，可以称这样的目标为“伪目标”。

依赖关系是指编译目标体要依赖的一个或多个文件列表。

命令是指为了从指定的依赖体创建出目标体所需执行的命令。

一个规则可以有多个命令行，每一条命令占一行。注意：每一个命令的第一个字符必须是制表符 Tab，如果使用空格会导致错误，make 会在执行过程中显示 Missing Separator(缺少分隔符)并停止。

在 3.1 节中创建了一个名为 hello.c 的文件，并使用命令 gcc -o hello hello.c 生成了一个可执行文件 hello。如果要利用 make 工具生成可执行程序，则首先要在 hello.c 所在的目录下编写一个 Makefile 文件，文件内容如下。

```
all: hello.o
	gcc hello.o -o hello
hello.o:hello.c
	gcc -c hello.c -o hello.o
clean:
	rm *.o hello
```

以上共有 3 个规则，第一个规则是生成 hello 可执行程序，第二个规则是生成 hello.o 目标文件，第三个规则是删除 hello 和后缀为.o 的所有文件。

上面例子中的编译器是 GCC，而在嵌入式项目开发中经常要使用交叉编译器，如本书采用的交叉编译器是 arm-linux-gcc。如果要交叉编译 hello.c 程序，就要将 Makefile 文件中所有 gcc 替换成 arm-linux-gcc，如果一个一个修改会非常麻烦，所以在 Makefile 中引进变量来解决。

使用变量将上面的 Makefile 文件改写为如下形式。

```
CC=gcc
OBJECT=hello.o
all: $(OBJECT)
	$(CC) $(OBJECT) -o hello
$(OBJECT):hello.c
	$(CC) -c hello.c -o $(OBJECT)
clean:
	rm *.o hello
```

在文件中，CC 和 OBJECT 是定义的两个变量，它们的值分别是 gcc 和 hello.o。变量的引用方法是：把变量用括号括起来，并在前面加上“$”。例如引用变量 CC，就可以写成 $(CC)。

变量一般在 Makefile 文件的头部进行定义，按照惯例，变量名一般使用大写字母。变量的内容可以是命令、文件、目录、变量、文件列表、参数列表、常量、目标名等。

如果要对上面的 hello.c 进行交叉编译，可以将 Makefile 文件改写为如下形式。

```
CROSS=arm-linux-
CC= $(CROSS)gcc
OBJECT=hello.o
all: $(OBJECT)
    $(CC) $(OBJECT) -o hello
$(OBJECT):hello.c
    $(CC) -c hello.c -o $(OBJECT)
clean:
    rm *.o hello
```

2. Make 工具的使用

Makefile 文件编写完成以后，需要通过 make 工具来执行，命令格式如下。

```
#make [target]
```

参数 target 是指要处理的目标名。make 命令会自动查找当前目录下的 Makefile 或 makefile 文件，如果文件存在就执行，否则报错。如果 make 命令后面没有任何参数，则表示处理 Makefile 文件中的第一个目标。

例如，如果使用前面编写好的 Makefile 文件，执行 make 命令或 make all 命令，都表示执行 all 目标，即生成 hello 文件；执行 make clean 命令，表示执行 clean 目标，即删除 hello 和后缀为.o 的所有文件。

GUN Make 工具在当前工作目录中按照 GNUmakefile、makefile、Makefile 的顺序搜索 Makefile 文件，也可以通过-f 参数指定描述文件。如果编写的 Makefile 文件名为 zhs，则可以通过 make -f zhs 命令来执行。Make 工具的选项很多，读者可以到 make 工具参考书上查找。

3.3.2　Makefile 的应用

3.3.1 节只介绍了 Makefile 的简单编写和使用方法，本节通过实例来详细讲解 Makefile 的应用。

1. 所有文件均在一个目录下的 Makefile 的编写

现有 7 个文件分别是 m.c、m.h、study.c、listen.c、visit.c、play.c、watch.c。

m.c 文件的内容如下。

```
#include<stdio.h>
main()
{
    int i;
    printf("please input the value of i from 1 to  5:\n");
    scanf("%d",&i);
    if(i==1)
            visit();
    else if(i==2)
            study();
    else if(i==3)
            play();
    else if(i==4)
```

```
            watch();
    else if(i==5)
            listen();
    else
            printf("nothing to do\n");
    printf("This is a woderful day\n");
}
```

study.c 文件的内容如下。

```
void study()
{
    printf("study embedded system today\n");
}
```

listen.c 文件的内容如下。

```
#include<stdio.h>
void listen()
{
    printf("listen english today\n");
}
```

play.c 文件的内容如下。

```
#include<stdio.h>
void play()
{
    printf("play football today\n");
}
```

visit.c 文件的内容如下。

```
#include<stdio.h>
void visit()
{
    printf("visit friend today\n");
}
```

watch.c 文件的内容如下。

```
#include<stdio.h>
void watch()
{
    printf("watch TV today\n");
}
```

m.h 文件的内容如下。

```
void visit();
void listen();
void watch();
void study();
void play();
```

从上面的代码可以看出这些文件之间的相互依赖关系，如图 3.2 所示。

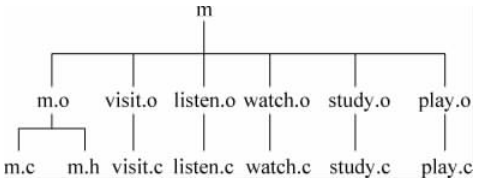

图 3.2　文件之间的依赖关系

现在利用这 7 个程序生成一个名为 m 的可执行程序，Makefile 文件可编写如下。

```
CC=gcc
TARGET=All
OBJECTS= m.o visit.o listen.o watch.o study.o play.o
$(TARGET):$(OBJECTS)
    $(CC) $(OBJECTS) -o m
m.o:m.c m.h
    $(CC) -c m.c -o m.o
visit.o:visit.c
    $(CC) -c visit.c -o visit.o
listen.o:listen.c
    $(CC) -c listen.c -o listen.o
watch.o:watch.c
    $(CC) -c watch.c -o watch.o
study.o:study.c
    $(CC) -c study.c -o study.o
play.o:play.c
    $(CC) -c play.c -o play.o
clean:
    rm *.o
```

这个 Makefile 文件可以通过预定义变量来简化。常见预定义变量如表 3.5 所示。

表 3.5　Makefile 预定义变量

变　量	说　明
$@	规则的目标所对应的文件名
$*	不包含扩展名的目标文件名称
$+	所有的依赖文件，以空格分开，并以出现的先后为序，可能包含重复的依赖文件
$%	如果目标是归档成员，则该变量表示目标的归档成员名称
$<	规则中的第一个依赖文件名
$^	规则中所有依赖的列表，以空格为分隔符
$?	规则中日期新于目标的所有依赖文件的列表，以空格为分隔符
$(@D)	目标文件的目录部分(如果目标在子目录中)
$(@F)	目标文件的文件名部分(如果目标在子目录中)

现用 $@、$<、$^ 来改写上述 Makefile 文件。

```
CC=gcc
TARGET=All
```

```
OBJECTS= m.o visit.o listen.o watch.o study.o play.o
$(TARGET):$(OBJECTS)
    $(CC) $^ -o m
m.o:m.c m.h
    $(CC) -c $< -o $@
visit.o:visit.c
    $(CC) -c $< -o $@
listen.o:listen.c
    $(CC) -c $< -o $@
watch.o:watch.c
    $(CC) -c $< -o $@
study.o:study.c
    $(CC) -c $< -o $@
play.o:play.c
    $(CC) -c $< -o $@
clean:
    rm *.o
```

从修改后的 Makefile 文件可以看出,各个文件的编译命令几乎没有区别,所以进一步用%和 * 两个通配符来简化。

```
CC=gcc
TARGET=All
OBJECTS= m.o visit.o listen.o watch.o study.o play.o
$(TARGET):$(OBJECTS)
    $(CC) $^ -o m
*.o:*.c
    $(CC) -c $< -o $@
clean:
    rm *.o
```

2. 编写文件在不同目录下的 Makefile

假设程序的目录结构为：源文件、可执行文件和 Makefile 在 src 目录中,头文件在 include 目录中,obj 存放.o 文件。就需要指定文件和头文件路径。仍以上面的程序为例。Makefile 文件如下。

```
CC=gcc
SRC_DIR=./
OBJ_DIR=../obj/
INC_DIR=../include/
TARGET=all
$(TARGET):$(OBJ_DIR)m.o $(OBJ_DIR)visit.o $(OBJ_DIR)listen.o $(OBJ_DIR)watch.o\
          $(OBJ_DIR)study.o $(OBJ_DIR)play.o
          $(CC) $^ -o $(SRC_DIR)m
$(OBJ_DIR)m.o:$(SRC_DIR)m.c $(INC_DIR)m.h
          $(CC) -I$(INC_DIR) -c -o $@ $<
$(OBJ_DIR)visit.o:$(SRC_DIR)visit.c
          $(CC) -c $< -o -@
$(OBJ_DIR)listen.o:$(SRC_DIR)listen.c
          $(CC) -c $< -o -@
```

```
$(OBJ_DIR)watch.o: $(SRC_DIR)watch.c
        $(CC) -c $< -o -@
$(OBJ_DIR)study.o: $(SRC_DIR)study.c
        $(CC) -c $< -o -@
$(OBJ_DIR)play.o: $(SRC_DIR)play.c
        $(CC) -c $< -o -@
clean:
        rm $(OBJ_DIR)*.o
```

3.3.3　自动生成 Makefile

编写 Makefile 确实不是一件轻松的事，尤其对于一个较大的项目而言更是如此。本节要讲的 autoTools 系列工具正是为此而设的，它只需用户输入简单的目标文件、依赖文件、文件目录等就可以轻松地生成 Makefile。另外，这些工具还可以完成系统配置信息的收集，方便地处理各种移植性的问题。

autoTools 是系列工具，它包含了 aclocal、autoscan、autoconf、autoheader 和 automake 工具，使用 autoTools 主要就是利用各个工具的脚本文件来生成最后的 Makefile 文件。其总体流程如图 3.3 所示。

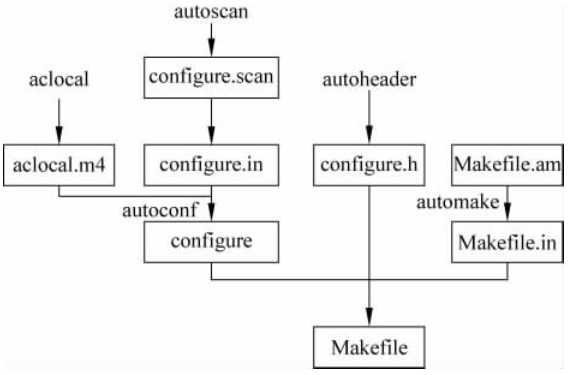

图 3.3　自动生成 Makefile 的流程图

以 3.1.2 节中的 hello.c 为例介绍自动生成 Makefile 的过程。

1. autoscan

```
# ls
hello.c
# autoscan
# ls
autoscan.log configure.scan hello.c
```

2. 创建 configure.in 文件

configure.in 是 autoconf 的脚本配置文件，是在 configure.scan 基础上修改的。修改如下。

```
# vi configure.scan
#-*-Autoconf-*-                    //以"#"号开始的行为注释
AC_PREREQ(2.59)                    //本文件要求的 autoconf 版本
```

```
AC_INIT(hello,1.0)                    // AC_INIT 宏用来定义软件的名称和版本等信息
AM_INIT_AUTOMAKE(hello,1.0)           //是 automake 所必备的宏,软件名称和版本号
AC_CONFIG_SRCDIR([hello.c])           //用来侦测所指定的源码文件是否存在
AC_CONFIG_HEADER([config.h])          //用于生成 config.h 文件,以便 autoheader 使用
AC_PROG_CC
AC_CONFIG_FILES([Makefile])           //用于生成相应的 Makefile 文件
AC_OUTPUT
```

最后用命令 mv configure.scan configure.in 将 configure.scan 改成 configure.in。

3. 运行 aclocal 生成 aclocal.m4 文件

```
# aclocal
# ls
aclocal.m4 autoscan.log configure.in hello.c
```

4. 运行 autoconf,生成 configure 可执行文件

```
# autoconf
# ls
aclocal.m4 autom4te.cache autoscan.log configure configure.in hello.c
```

5. 使用 autoheader 生成 config.h.in

```
# autoheader
```

6. 创建 Makefile.am 文件

automake 用的脚本配置文件是 Makefile.am,需要先创建相应的文件。

```
# vi Makefile.am
```

内容为：AUTOMAKE_OPTIONS=foreign

```
bin_PROGRAMS= hello
hello_SOURCES= hello.c
```

接下来用 automake 生成 Makefile.in,使用选项 adding-missing 可以让 automake 自动添加一些必需的脚本文件,命令如下。

```
# automake --add-missing
configure.in: installing './install-sh'
configure.in: installing './missing'
Makefile.am: installing 'depcomp
# ls
aclocal.m4 autoscan.log configure depcomp install-sh Makefile.in
autom4te.cache config.h.in configure.in hello.c Makefile.am missing
```

7. configure

通过运行自动配置设置文件 configure,Makefile.in 变成了最终的 Makefile。

```
# ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for gawk... gawk
```

```
checking whether make sets $(MAKE)... yes
checking for gcc... gcc
checking for C compiler default output... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts-g... yes
checking for gcc option to accept ANSI C... none needed
checking for style of include used by make... GNU
checking dependency style of gcc... gcc3
configure: creating ./config.status
config.status: creating Makefile
config.status: creating config.h
config.status: executing depfiles commands
```

8. 执行 make 命令，生成可执行文件 hello

```
# make
cd . && /bin/sh ./config.status config.h
config.status: creating config.h
config.status: config.h is unchanged
make all-am
make[1]: Entering directory '/lvli/12'
source='hello.c' object='hello.o' libtool=no \
depfile='.deps/hello.Po' tmpdepfile='.deps/hello.TPo' \
depmode=gcc3 /bin/sh ./depcomp \
gcc -DHAVE_CONFIG_H -I. -I. -I.      -g -O2 -c 'test-f 'hello.c' || echo './''hello.c
gcc -g -O2      -o hello hello.o
cd . && /bin/sh ./config.status config.h
config.status: creating config.h
config.status: config.h is unchanged
make[1]: Leaving directory '/lvli/12
```

9. 运行 hello

```
# ./hello
hello,world!
```

3.4　Linux 应用程序设计

虽然 Linux 操作系统下的 C 语言编程与 Windows 操作系统下的 C 语言编程方法基本相同，但是也有细微的差别。下面通过文件操作、时间获取和创建线程等任务来了解 Linux 应用程序设计。

3.4.1　文件操作编程

在 Linux 操作系统下，实现文件操作可以采用两种方法，一种是通过 Linux 系统调用来

实现,另一种是通过 C 语言库函数调用来实现。前者依赖于 Linux 操作系统,后者独立于具体操作系统,即在任何操作系统下,使用 C 语言库函数操作文件的方法都相同。Linux 的系统调用在 5.1.3 节介绍,本节只介绍利用 C 语言库函数来操作文件。下面介绍一些常用的文件操作函数,这些函数的说明包含在 stdio.h 头文件中。

1. 打开和关闭文件函数

打开文件可通过 fopen 函数来完成,关闭文件可通过 fclose 函数来完成,格式如下。

```
FILE * fopen(const char * filename, const char * mode);
int fclose(FILE * stream);
```

其中:参数 filename 表示需要打开的文件名(包括路径,默认为当前路径)。mode 为文件打开模式,常见模式如表 3.6 所示。若成功打开文件,fopen 函数返回值是文件指针,若文件打开失败则返回 NULL,并把错误代码存在 errno 中。

表 3.6　常见模式

模　式	含　义
r, rb	只读方式打开文件,该文件必须存在
r+, rb+	读写方式打开文件,若文件不存在则自动创建
w, wb	只写方式打开文件,若文件不存在则自动创建
w+, wb+	读写方式打开文件,若文件不存在则自动创建
a, ab	追加方式打开文件,若文件不存在则自动创建
a+,ab+	读和追加方式打开文件,若文件不存在则自动创建

模式中的 b 用于区分文本文件和二进制文件。在 Windows 操作系统下有区分,但在 Linux 下不需要区分。

2. 读取文件数据函数

读取文件数据可通过 fread、fgetc 和 fgets 等函数实现,格式如下。

```
size_t fread(void * ptr, size_t size, size_t n, FILE * stream);
int fgetc(FILE * stream);
char * fgets(char * s,int size,FILE * stream);
```

fread 函数的功能是从 stream 指向的文件中,读取长度为 n * size 个字节的字符串,并将读取的数据保存到 ptr 缓存中,返回值是实际读出数据的字节数。fgetc 函数的功能是从 stream 指向的文件中读取一个字符,若读到文件尾,则返回 EOF。fgets 函数的功能是从 stream 指向的文件中读取一串字符,并存到 s 缓存中,直到出现换行字符、文件尾或已读了 size−1 个字符时结束,最后会加上 NULL 作为字符串结束符。

3. 向文件写数据函数

向文件写数据可通过 fwrite、fputc 和 fputs 等函数实现,格式如下。

```
size_t fwrite(const void * ptr,size_t size,size_t n,FILE * stream);
int fputc(int c,FILE * stream);
int fputs(const char * s,FILE * stream);
```

fwrite 函数的功能是将 ptr 缓存中的数据写到 stream 指向的文件中,写入长度为 n * size 个字节,返回值是实际写入的字节数。fputc 函数的功能是向 stream 指向的文件中写入一

个字符。fputs 函数的功能是将 s 缓存中的字符串写入到 stream 指向的文件中。

下面通过一个实际应用加深对文件操作的理解。

【程序 3.1】 文件复制程序 file_copy.c。

```
#include <stdio.h>
#include <stdlib.h>
#define BUFFER_SIZE 1024
int main(int argc, char ** argv )
{
    FILE * fileFrom, * fileTo;
    char buffer[BUFFER_SIZE]={0};
    int length=0;
    /* 检查输入命令格式是否正确 */
    if(argc!=3)
        {    printf("Usage: %s fileFrom fileTo\n", argv[0]);
             exit(0);
        }
    /* 打开源文件 */
    fileFrom = fopen(argv[1],"rb+");
    if(fileFrom==NULL)
        {
            printf(" Open File %s Failed\n", argv[1]);
            exit(0);
        }
    /* 打开或创建目标文件 */
    fileTo = fopen(argv[2],"wb+");
    if(fileTo==NULL)
        {
            printf(" Open File %s Failed\n", argv[2]);
            exit(0);
        }
    /* 复制文件内容 */
    while ((length =fread(buffer,1,BUFFER_SIZE,fileFrom))>0 )
        {
            fwrite(buffer,1,length,fileTo);
        }
    /* 关闭文件 */
    fclose(fileFrom);
    fclose(fileTo);
    return 0;
}
```

编译源程序并生成可执行程序 file_copy，然后执行 file_copy 程序将 hello.c 复制成 zhs.c，则编译和运行命令如下。

```
#gcc file_copy.c -o file_copy
#./file_copy hello.c zhs.c
```

3.4.2　时间编程

在编程中经常要使用到时间，如获取系统时间、计算事件耗时等。下面介绍一些常用的

时间函数,这些函数的说明包含在 time.h 头文件中。

1. time 函数

函数格式:time_t　time(time_t　*tloc);

函数功能:获取日历时间,即从 1970 年 1 月 1 日 0 点到现在所经历的秒数,结果保存在 tloc 中。如果操作成功,则返回值为经历的秒数;若操作失败,则返回值为((time_t)－1),错误原因存于 errno 中。

2. gmtime 函数

函数格式:struct tm　*gmtime(const time_t　*timep);

函数功能:将日历时间转化为格林威治标准时间,并将数据保存在 tm 结构中。tm 结构的定义如下。

```
struct tm
{
int tm_sec;          //秒
int tm_min;          //分
int tm_hour;         //时
int tm_mday;         //日
int tm_mon;          //月
int tm_year;         //年
int tm_wday;         //本周第几日
int tm_yday;         //本月第几日
int tm_isdst;        //日光节约时间
};
```

3. gettimeofday 函数

函数格式:int gettimeofday(struct timeval *tv,struct timezone *tz);

函数功能:获取从今日凌晨到现在的时间差,并存放在 tv 中,然后将当地时区的信息存放到 tz 中。两个结构的定义如下。

```
strut timeval {
long tv_sec;              /* 秒数 */
long tv_usec;             /* 微秒数 */
};
struct timezone{
int tz_minuteswest;       /* 和 GMT 时间差 */
int tz_dsttime;
};
```

4. sleep 和 usleep 函数

函数格式:unsigned int sleep(unsigned int sec);

　　void usleep(unsigned long usec);

函数功能:sleep 函数的功能是使程序睡眠 sec 秒,usleep 函数的功能是使程序睡眠 usec 微秒。

下面通过一个实际应用加深对获取时间方法的理解。

【程序 3.2】 算法分析程序 test_time.c。

```
#include <sys/time.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

/* 算法 */
void function()
{
    unsigned int i,j;
    double y;
    for(i=0;i<100;i++)
        for(j=0;j<100;j++)
            {usleep(10);y++;}
}

main()
{
    struct timeval tpstart,tpend;
    float timeuse;

    gettimeofday(&tpstart,NULL);        // 获取开始运行时间
    function();
    gettimeofday(&tpend,NULL);          // 获取运行结束时间
    /* 计算算法执行时间 */
    timeuse=1000000*(tpend.tv_sec-tpstart.tv_sec)+tpend.tv_usec-tpstart.tv_usec;
    timeuse/=1000000;
    printf("Used Time:%f sec.\n",timeuse);
    exit(0);
}
```

程序编译及运行结果如下。

```
#gcc test_time.c  -o  test_time
#./test_time
Used Time : 39.432861 sec.
```

3.4.3　多线程编程

采用多线程可以提高程序的运行效率，目前绝大多数嵌入式操作系统和中间件都支持多线程。Linux 操作系统的多线程遵循 POSIX 线程接口，称为 pthread。在 Linux 操作系统进行多线程编程时，需要使用到 pthread.h 头文件和 libpthread.a 库文件。下面介绍几个常用的线程函数。

1. pthread_create 函数

函数格式：int pthread_create(pthread_t *tid,const pthread_attr_t *attr, void *(*start_rtn)(void), void *arg);

函数功能：创建一个新的线程。参数 tid 为线程 id；attr 为线程属性，通常设置为 NULL；start_rtn 是线程要执行的函数；arg 是执行函数 start_rtn 的参数。当创建线程成

功时,函数返回值为 0;若返回值为 EAGAIN,则表示系统限制创建新的线程,例如线程数目过多;若返回值为 EINVAL,则表示第二个参数代表的线程属性值非法。创建线程成功后,新创建的线程则运行参数三和参数四确定的函数,原来的线程则继续运行下一行代码。

2. pthread_exit 函数

函数格式:int pthread_exit(void * rval_ptr);

函数功能:退出当前线程,返回值保存在 rval_ptr 中。

3. pthread_join 函数

函数格式:int pthread_join(pthread_t tid, void ** rval_ptr);

函数功能:阻塞调用线程,直到指定的线程终止。参数 tid 是指定的线程,rval_ptr 是线程退出的返回值。

下面通过一个实际应用加深对多线程编程的理解。

【程序 3.3】 创建线程程序 pthread_create.c。

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
/*子线程执行的函数*/
void *thread(void *str)
{
    int i;
    for (i = 0; i < 6; ++i)
    {
        sleep(2);
        printf( "This in the thread : %d\n" , i );
    }
    return NULL;
}

int main()
{
    pthread_t pth;
    int i;
    int ret;
    ret=pthread_create(&pth, NULL, thread, (void *)(i));        //创建线程

    printf("Test start\n");
    for (i = 0; i < 6; ++i)
    {
        sleep(1);
        printf( "This in the main : %d\n" , i );
    }

  pthread_join(pth, NULL);                                      //等待线程结束
  return 0;
}
```

程序编译及运行结果如下。

```
# gcc pthread_create.c -lpthread -o pthread_create
# ./pthread_create
Test start
This in the main : 0          //主线程上的输出
This in the thread : 0        //子线程上的输出
This in the main : 1          //主线程上的输出
This in the main : 2          //主线程上的输出
This in the thread : 1        //子线程上的输出
This in the main : 3          //主线程上的输出
This in the main : 4          //主线程上的输出
This in the thread : 2        //子线程上的输出
This in the main : 5          //主线程上的输出
This in the thread : 3        //子线程上的输出
This in the thread : 4        //子线程上的输出
This in the thread : 5        //子线程上的输出
```

3.5 练　习　题

1. 选择题

(1) GCC 软件是(　　)。

A. 调试器　　B. 编译器　　C. 文本编辑器　　D. 连接器

(2) GDB 软件是(　　)。

A. 调试器　　B. 编译器　　C. 文本编辑器　　D. 连接器

(3) 如果生成通用计算机上(系统是 Linux 操作系统)能够执行的程序,则使用的 C 编译是(　　)。

A. TC　　B. VC　　C. GCC　　D. arm-linux-gcc

(4) GCC 用于指定头文件目录的选项是(　　)。

A. -o　　B. -L　　C. -g　　D. -I

(5) make 有许多预定义变量,表示“目标完整名称”的是(　　)。

A. $@　　B. $^　　C. $<　　D. $>

2. 填空题

(1) Linux 下,动态链接库文件是以_____结尾的,静态链接库文件是以_____结尾的。动态链接库是在_____动态加载的,静态链接库是在_____静态加载的。

(2) GCC 指定库文件目录选项的字母是_____。指定头文件目录选项的字母是_____。指定输出文件名选项的字母是_____。

(3) 为了方便文件的编辑,在编辑 Makefile 时,可以使用变量。引用变量时,只需在变量前面加上_____符。

(4) Makefile 文件预定定义变量有很多,列举 3 个预定定义变量:_____、_____和_____。

(5) Makefile 文件预定定义变量“$@”表示_____,“$^”表示_____,“$<”表示_____。

3. 简答题

(1) make 和 Makefile 之间的关系?

(2) Makefile 的普通变量和预定义变量有什么不同?预定义变量有哪些?它们分别表示什么意思?

(3) GCC 编译器的常用参数有哪些?它们的功能分别是什么?

4. 编程及调试题

(1) 根据要求编写 Makefile 文件。有 5 个文件分别是 main. c、visit. h、study. h、visit. c、study. c,具体代码如下。

main. c 文件

```
#include<stdio.h>
main()
{
    int i;
    printf("please input the value of i from 1 to  5:\n");
    scanf("%d",&i);
    if(i==1)
        visit();
    if(i==2)
        study();
}
```

visit. h 文件

```
void visit();
```

study. h 文件

```
void study();
```

visit. c 文件

```
#include "visit.h"
void visit()
{
    printf("visit friend today\n");
}
```

study. c 文件

```
#include "study.h"
void study()
{
    printf("study embedded system today\n");
}
```

① 如果上述文件在同一目录,请编写 Makefile 文件,用于生成可执行程序 zhs。

② 如果按照下面的目录结构存放文件,请改写 Makefile 文件。

bin:存放生成的可执行文件;

obj:存放. o 文件;

include：存放 visit.h、study.h；

src：存放 main.c、visit.c、study.c 和 Makefile。

③ 如果按照下面的目录结构存放文件，请改写 Makefile 文件。

bin：存放生成的可执行文件；

obj：存放.o 文件；

include：存放 visit.h 和 study.h；

src：存放 main.c 和 Makefile；

src1：存放 visit.c；

src2：存放 study.c。

(2) 按照要求完成以下操作。

① 用 vi 编辑 test.c 文件，其内容如下。

```
#include<stdio.h>
int main()
{
    int s=0,i;
    for(i=1;i<=15;i++)
    {
        s=s+i;
        printf("the value of s is %d \n",s);
    }
    return 0;
}
```

② 使用 gcc -o test.o test.c 编译，生成 test.o。

③ 使用 gcc -g -o test1.o test.c 编译，生成 test1.o。

④ 比较 test.o 和 test1.o 文件的大小，思考为什么？

(3) 使用 GDB 调试上面的程序。

① 带调试参数-g 进行编译。

```
#gcc -g test.c -o test
```

② 启动 GDB 调试，开始调试。

```
#gdb Gtest
```

③ 使用 gdb 的命令进行调试。

(4) 编写一个程序，将系统时间以 year-month-day hour：minute：second 格式显示在屏幕上，并将它保存在 time.txt 文件中。