重庆理工大学/电气学院
CHONGQING UNIVERSITY OF TECHNOLOGY

嵌入式Linux系统开发教程

—基于ARM处理器通用平台（arm9-arm11-contexA系列）

丛超

2025年4月

1 VI编辑器

2 程序编译与调试

3 Makefile

4 服务器配置

01

VI编辑器

Linux系统提供了一个完整的编辑器家族系列。

按功能它们可以分为两大类：行编辑器(Ed、Ex)和全屏幕编辑器(vi、vim、emacs)。

行编辑器每次只能对一行进行操作，使用起来很不方便。而全屏幕编辑器可以对整个屏幕进行编辑，用户编辑的文件直接显示在屏幕上，从而克服了行编辑的那种不直观的操作方式，便于用户学习和使用，具有强大的功能。

本节主要介绍vi编辑器的使用，vim相当于vi的增强版，使用方法一致。emacs留给读者自学，Ed、Ex编辑器使用较少不再累述。

vi是Linux系统的第一个全屏幕交互式编辑程序，它从诞生至今一直得到广大用户的青睐，历经数十年仍然是人们主要使用的文本编辑工具，足以见其生命力之强，而强大的生命力是其强大的功能带来的。

vi有3种模式，分别为命令行模式、插入模式及命令行模式各模式的功能，下面具体进行介绍。

(1)命令行模式

用户在用vi编辑文件时，最初进入的为一般模式。在该模式中可以通过上下移动光标进行“删除字符”或“整行删除”等操作，也可以进行“复制”、“粘贴”等操作，但无法编辑文字。

(2)插入模式

在该模式下，用户才能进行文字编辑输入，用户可按ESC键回到命令行模式。

(3)底行模式

在该模式下，光标位于屏幕的底行。用户可以进行文件保存或退出操作，也可以设置编辑环境，如寻找字符串、列出行号等。

(1)命令行模式常见功能键如表 2-1 所示。

表 2-1 vi 命令行模式功能键

标号	含义
i	切换到插入模式，此时光标当于开始输入文件处
a	切换到插入模式，并从目前光标所在位置的下一个位置开始输入文字
O	切换到插入模式，且从行首开始插入新的一行
[ctrl]+[b]	屏幕往“后”翻动一页
[ctrl]+[f]	屏幕往“前”翻动一页
[ctrl]+[u]	屏幕往“后”翻动半页
[ctrl]+[d]	屏幕往“前”翻动半页
0(数字 0)	光标移到本行的开头
G	光标移动到文章的最后
nG	光标移动到第 n 行

$	移动到光标所在行的“行尾”
n<Enter>	光标向下移动 n 行
/name	在光标之后查找一个名为 name 的字符串
?name	在光标之前查找一个名为 name 的字符串
x	删除光标所在位置的“后面”一个字符
dd	删除光标所在行
ndd	从光标所在行开始向下删除 n 行
yy	复制光标所在行
nyy	复制光标所在行开始的向下 n 行
p	将缓冲区内的字符粘贴到光标所在位置(与 yy 搭配)
u	恢复前一个动作

(2)插入模式的功能键只有一个 i，按 Esc 键退出到命令行模式。

02

程序编译与调试

嵌入式系统开发常用编译工具是gcc，调试工具使用gdb，下面将一一介绍。

01

gcc是GNU项目中符合ANSI C标准的编译系统，能够编译用C、C++和Object C等语言编写的程序。

gcc是一个交叉平台编译器，它能够在当前CPU平台上为多种不同体系结构的硬件平台开发软件，因此尤其适合在嵌入式领域的开发编译。本节中的示例，采用gcc版本为7.3.0。

表 2-3 gcc 所支持后缀名解释

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		

gcc演示

```
# 安装软件必须要有管理员权限
# ubuntu
$ sudo apt update                    # 更新本地的软件下载列表，得到最新的下载地址
$ sudo apt install gcc g++           # 通过下载列表中提供的地址下载安装包，并安装
```

gcc安装完毕之后，可以查看版本：

SHELL

```
# 查看 gcc 版本
$ gcc -v
$ gcc --version
```

```
congchao@ubuntu:~/cc$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.4.0-1ubuntu1~20.04.2' --with-bugurl=file:///usr/share/doc/gcc-9/README.Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,gm2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-9 --program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker-build-id --libexecdir=/usr/lib --without-included-gettext --enable-threads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu --enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-verify --enable-plugin --enable-default-pie --with-system-zlib --with-target-system-zlib=auto --enable-objc-gc=auto --enable-multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=generic --enable-offload-targets=nvptx-none=/build/gcc-9-9QDOt0/gcc-9-9.4.0/debian/tmp-nvptx/usr,hsa --without-cuda-driver --enable-checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu --target=x86_64-linux-gnu
Thread model: posix
gcc version 9.4.0 (Ubuntu 9.4.0-1ubuntu1~20.04.2)
```

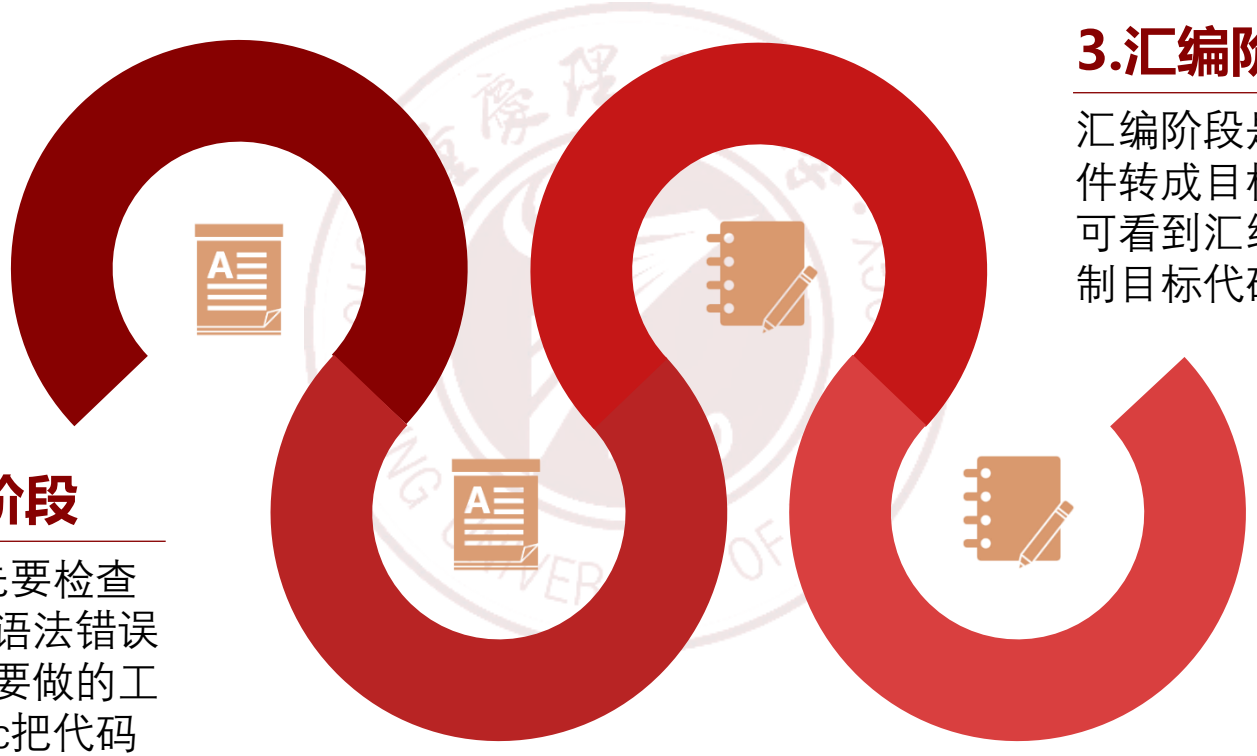
gcc演示：课本P43 2.2.2-1

```
congchao@ubuntu:~/cc$ gcc hello.c -o hello
congchao@ubuntu:~/cc$ ls
hello  hello.c  test1.c
congchao@ubuntu:~/cc$ ./hello
hello world!
congchao@ubuntu:~/cc$
```

gcc的编译流程分为了4个步骤依次为：预处理→编译→汇编→链接。

1. 预处理阶段

编译器将hello.c代码中的stdio.h编译进来，并且用户可以使用gcc的选项“-E”进行查看，该选项的作用是让gcc在预处理结束后停止编译过程。



2. 编译阶段

在这个阶段中，gcc首先要检查代码的规范性、是否有语法错误等，以确定代码的实际要做的工作，在检查无误后，gcc把代码翻译成汇编语言。用户可以使用“-S”选项来进行查看，该选项只进行编译而不进行汇编，生成汇编代码。

3.汇编阶段

汇编阶段是把编译阶段生成的“.s”文件转成目标文件，可使用选项“-c”就可看到汇编代码已转化为“.o”的二进制目标代码了。

4.链接阶段

在成功编译之后，就进入了链接阶段。函数库一般分为静态库和动态库两种。静态库是指编译链接时，把库文件的代码全部加入到可执行文件中，因此生成的文件比较大，但在运行时也就不再需要库文件了。其后缀名一般为“.a”。动态库与之相反，在编译链接时并没有把库文件的代码加入到可执行文件中，而是在程序执行时由运行时链接文件

表 2-4 gcc 总体选项列表

后缀名	所对应的语言
-c	只是编译不链接，生成目标文件“.o”
-S	只是编译不汇编，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	把输出文件输出到 file 里
-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
-L dir	在库文件的搜索路径列表中添加 dir 目录
-static	链接静态库
-llibrary	连接名为 library 的库文件

gcc 的告警和出错选项如表 2-5 所示。

表 2-5 gcc 总体选项列表

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

优化选项

- gcc可以对代码进行优化，它通过编译选项“-On”来控制优化代码的生成，其中n是一个代表优化级别的整数。对于不同版本的gcc来讲，n的取值范围及其对应的优化效果可能并不完全相同，比较典型的范围是从0变化到2或3。

- 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“-O”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，同时还要进行一些额外的调整工作，如处理器指令调度等。选项“-O3”则还包括循环展开和其他一些与处理器特性相关的优化工作。

优化选项

gcc 的体系结构相关选项如表 2-6 所示。

表 2-6 gcc 体系结构相关选项列表

选项	含义
-mieee-fp/-mno-ieee-fp	使用/不使用 IEEE 标准进行浮点数的比较
-msoft-float	输出包含浮点库调用的目标代码
-mshort	把 int 类型作为 16 位处理，相当于 short int
-mrtd	强行将函数参数个数固定的函数用 ret NUM 返回，节省调用函数的一条指令
-mcpu=type	针对不同的 CPU 使用相应的 CPU 指令。可选择的 type 有 i386、i486、pentium 及 i686 等

嵌入式系统开发常用编译工具是gcc，调试工具使用gdb，下面将一一介绍。

02

在软件开发过程中，调试是其中最重要的一环，很多时候，调试程序的时间比实际编写代码的时间要长的多。

gdb作为GNU开发组织发布的一个强大的UNIX/Linux下的程序调试工具，提供了强大的调试功能。gdb调试基本命令如表2-7所示。

1.调试准备

- 项目程序如果是为了进行调试而编译时， 必须要打开调试选项(-g)。
- 另外还有一些可选项，比如: 在尽量不影响程序行为的情况下关掉编译器的优化选项(-O0)，-Wall选项打开所有 warning，也可以发现许多问题，避免一些不必要的 bug。
- -g选项的作用是在可执行文件中加入源代码的信息，比如可执行文件中第几条机器指令对应源代码的第几行，但并不是把整个源文件嵌入到可执行文件中，所以在调试时必须保证gdb能找到源文件。
- 习惯上如果是c程序就使用gcc编译, 如果是 c++程序就使用g++编译, 编译命令中添加上边提到的参数即可。

表 2-7 gdb 体系结构相关选项列表

命令	缩写	用法	作 用
help	h	h command	显示命令的帮助
run	r	r [args]	运行要调试的程序 args 为要运行程序的参数
step	s	s [n]	步进,n 为步进次数。如果调用了某个函数，会跳入函数内部。
next	n	n [n]	下一步,n 为下一步的次数
continue	c	c	继续执行程序
list	l	l / l+ / l-	列出源码
break	b	b address	在地址 address 上设置断点
		b function	此命令用来在某个函数上设置断点。
		b linenum	在行号为 linenum 的行上设置断点。程序在运行到此行之前停止
		b +offset b -offset	在当前程序运行到的前几行或后几行设置断点。offset 为行号
watch	w	w exp	监视表达式的值
kill	k	k	结束当前调试的程序
print	p	p exp	打印表达式的值
output	o	o exp	同 print,但是不输出下一行的语句
ptype		ptype struct	输出一个 struct 结构的定义
whatis		whatis var	命令可以显示某个变量的类型
pwd		pwd	显示当前路径
delete	d	d num	删除编号为 num 的断点和监视

display		display expr	暂停，步进时自动显示表达式的值
finish			执行直到函数返回 执行直到当前 stack 返回
return			强制从当前函数返回
where			命令用来查看执行的代码在什么地方中止
backtrace	bt		显示函数调用得所有栈框架(stack frames)的 踪迹和当前函数的参数的值。
quit	q		退出调试程序
shell		shell ls	执行 shell 命令
make			不退出 gdb 而重新编译生成可执行文件
disassemble			显示反汇编代码
thread		thread thread_no	用来在线程之间的切换
set		set width 70	就是把标准屏幕设为 70 列
		set var=54	设置变量的值。
forward/search		search string	从当前行向后查找匹配某个字符串的程序行
reverse-search			forward/search 相反，向前查找字符串。使用格式同上
up/down			上移/下移栈帧，使另一函数成为当前函数
info	i	i breakpoint	显示当前断点列表
		i reg[ister]	显示寄存器信息
		i threads	显示线程信息
		i func	显示所有的函数名
		info proc all	显示上面 proc 命令这些命令返回的所有信息
x	x/(length)(format)(size) addr x/6(o/d/x/u/c/t)(b/h/w)		按一定格式显示内存地址或变量的值

2. 演示用程序：

```
// args.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define NUM 10

// argc, argv 是命令行参数
// 启动应用程序的时候
int main(int argc, char* argv[])
{
    printf("参数个数: %d\n", argc);
    for(int i=0; i<argc; ++i)
    {
        printf("%d\n", NUM);
        printf("参数 %d: %s\n", i, argv[i]);
    }
    return 0;
}
```

3. 演示gdb:

第一步: 编译出带条信息的可执行程序

gcc args.c -o app -g

第二步: 启动gdb进程, 指定需要gdb调试的应用程序名称

gdb app

第三步: 在启动应用程序 app之前设置命令行参数。gdb中设置参数的命令叫做set args ...， 查看设置的命令行参数命令是 show args。 语法格式如下:

```
# 设置的时机: 启动gdb之后, 在应用程序启动之前
(gdb) set args 参数1 参数2 .... ...
# 查看设置的命令行参数
(gdb) show args
```

第四步： 用run命令或start命令启动

run: 可以缩写为 r, 如果程序中设置了断点会停在第一个断点的位置, 如果没有设置断点, 程序就执行完了

start: 启动程序, 最终会阻塞在main函数的第一行， 等待输入后续其它 gdb 指令

第五步： 如果想让程序start之后继续运行, 或者在断点处继续运行,

可以使用： continue命令, 可以简写为 c

也可以用： step（步进）； next步进（不进函数）， until（跳出循环）

第六步： 在调试时查看代码： list命令

最后： 退出gdb调试, 就是终止 gdb 进程, 需要使用 quit命令, 可以缩写为 q

4. 文件代码查看：

- **当前文件：**

一个项目中一般是有很多源文件的，默认情况下通过list查看到代码信息位于程序入口函数main对应的的那个文件中。因此如果不进行文件切换main函数所在的文件就是当前文件，如果进行了文件切换，切换到哪个文件哪个文件就是当前文件。当前文件的查看用list命令

- **切换文件：**

在查看文件内容的时候，很多情况下需要进行文件切换，我们只需要在list命令后边将要查看的文件名指定出来就可以了，切换命令执行完毕之后，这个文件就变成了当前文件。文件切换方式如下：

```
# 切换到指定的文件，并列出这行号对应的上下文代码，默认情况下只显示10行内容
(gdb) l 文件名:行号

# 切换到指定的文件，并显示这个函数的上下文内容，默认显示10行
(gdb) l 文件名:函数名
```

默认通过list只能一次查看10行代码，如果想显示更多，可以通过set listsize设置，同样如果想查看当前显示的行数可以通过 show listsize查看，这里的listsize可以简写为 list。

5. 断点：

- **设置断点：**

想要通过gdb调试某一行或者得到某个变量在运行状态下的实际值，就需要在在这一行设置断点，程序指定到断点的位置就会阻塞，我们就可以通过gdb的调试命令得到我们想要的信息了。

断点的设置有两种方式一种是常规断点，程序只要运行到这个位置就会被阻塞，还有一种叫条件断点，只有指定的条件被满足了程序才会在断点处阻塞。调试程序的断点可以设置到某个具体的行，也可以设置到某个函数上，具体的设置方式如下：

```
(gdb) b 行号
(gdb) b 函数名              # 停止在函数的第一行
```

如果要在非当前文件的某一行上设置断点：

```
(gdb) b 文件名:行号
(gdb) b 文件名:函数名              # 停止在函数的第一行
```

设置条件断点：

```
# 必须要满足某个条件，程序才会停在这个断点的位置上
# 通常情况下，在循环中条件断点用的比较多
(gdb) b 行数 if 变量名==某个值
```

5. 断点：

- **查看断点：**
  断点设置完毕之后，可以通过 info break命令查看设置的断点信息，其中info可以缩写为i

```
(gdb) i b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x0000000000400cb5 in main() at test.cpp:12
2       breakpoint     keep y   0x0000000000400cbd in main() at test.cpp:13
3       breakpoint     keep y   0x0000000000400cec in main() at test.cpp:18
4       breakpoint     keep y   0x00000000004009a5 in insertionSort(int*, int)
                                                   at insert.cpp:8
5       breakpoint     keep y   0x0000000000400cdd in main() at test.cpp:16
6       breakpoint     keep y   0x00000000004009e5 in insertionSort(int*, int)
                                                   at insert.cpp:16
```

在显示的断点信息中有一些属性需要在其他操作中被使用，
Num: 断点的编号，删除断点或者设置断点状态的时候都需要使用
Enb: 当前断点的状态，y表示断点可用，n表示断点不可用
What: 描述断点被设置在了哪个文件的哪一行或者哪个函数上

5. 断点：

- **删除断点：**

如果确定设置的某个断点不再被使用了，可用将其删除，删除命令是 delete 断点编号，这个delete可以简写为del也可以再简写为d。

删除断点的方式有两种：删除(一个或者多个)指定断点或者删除一个连续的断点区间

```
(gdb) d 1           # 删除第1个断点
(gdb) d 2 4 6       # 删除第2,4,6个断点

# 删除一个范围，断点编号 num1 - numN 是一个连续区间
(gdb) d num1-numN
# 举例，删除第1到第5个断点
(gdb) d 1-5
```

03

Makefile

- 使用 GCC 的命令行进行程序编译在单个文件下是比较方便的，当工程中的文件逐渐增多，甚至变得十分庞大的时候，使用 GCC 命令编译就会变得力不从心。
- 这种情况下我们需要借助项目构造工具 make 帮助我们完成任务。 make是一个命令工具，是一个解释makefile中指令的命令工具，一般来说，大多数的IDE都有这个命令，比如：Visual C++的nmake，QtCreator的qmake等。
- makefile带来的好处就是——“自动化编译”，一旦写好，只需要一个make命令，整个工程完全自动编译，极大的提高了软件开发的效率。
- make工具在构造项目的时候需要加载一个叫做makefile的文件，makefile关系到了整个工程的编译规则。一个工程中的源文件不计数，其按类型、功能、模块分别放在若干个目录中，makefile定义了一系列的规则来指定哪些文件需要先编译，哪些文件需要后编译，哪些文件需要重新编译，甚至于进行更复杂的功能操作，因为makefile就像一个Shell脚本一样，其中也可以执行操作系统的命令。
- 构建项目的时候在哪个目录下执行构建命令 make这个目录下的 makefile 文件就会别加载，因此在一个项目中可以有多个 makefile 文件，分别位于不同的项目目录中。

Makefile里面的规则由“目标：依赖命令”组成。例如，以一个最简单Makefile文件为例，有一个源程序hello.c文件，编写Makefile文件编译生成可执行文件hello，Makefile内容如下：

```
hello:hello.c
	gcc hello.c -o hello
clean:
	rm -f hello
```

hello是我们要产生的目标文件，后面的hello.c是它的依赖文件，下面为将依赖文件生成目标文件所执行的命令。

当执行make命令后，make会在当前目录下找名字叫“Makefile”或“makefile”的文件。如果找到，它会找文件中的第一个目标文件（hello），并把这个文件作为最终的目标文件。如果hello文件不存在，或是hello所依赖的文件的文件修改时间要比hello这个文件新，那么，他就会执行后面所定义的命令来生成hello这个文件。

像clean这种，没有被第一个目标文件直接或间接关联，那么它后面所定义的命令将不会被自动执行，不过，我们可以显示要make执行。即命令——“make clean”，以此来清除所有的目标文件，以便重编译。

注意：命令前面是以一个Tab键缩进，不能使用空格代替，所以gcc hello.c -o hello和rm -f hello前面是一个Tab键。

Makefile中变量的使用

当有很多源文件，头文件需要包含的时候，依赖项和编译命令的书写变得很麻烦，尤其当需要修改的时候，写的到处都是，很容易出错，这时候用户可以使用变量来方便书写。

示例 2.3.1-1 将有源文件test1.c、test2.c、test3.c编译生成可执行程序test，Makefile内容如下：

```
objects = test1.c test2.c test3.c
test: $(objects)
	gcc -o test $(objects)
clean:
	rm -f test
```

如例子中所写的，我们创建了一个objects变量，它的值是我们的依赖文件，这样在下面用到依赖文件的时候，可以用$(objects)来替代，这样做的好处就是，当我们需要修改依赖文件的时候，只需要修改objects的值即可，便于维护。

另，Makefile有三个非常有用的变量。分别是$@，$^，$<，代表的意义分别是：

$@：目标文件；

$^：所有的依赖文件；

$<：第一个依赖文件。

我们可以直接使用这几个变量，来方便我们Makefile的书写。

例如，上面的makefile例子可以改写为：

```
objects = test1.c test2.c test3.c
test: $(objects)
	gcc -o $@ $^
clean:
	rm -f test
```

Makefile复杂文件的编写，读者可以参考相关的书籍，本书掌握这些内容，能够满足程序调试的需求。

演示用程序：

- 假设我们有如下的C语言项目：

```
my_project/
├── Makefile
├── main.c
├── func.c
└── func.h
```

```c
#include <stdio.h>
#include "func.h"

int main() {
    say_hello();
    return 0;
}
```

```c
#include <stdio.h>
#include "func.h"

void say_hello() {
    printf("Hello, Makefile!\n");
}
```

```c
#ifndef FUNC_H
#define FUNC_H

void say_hello();

#endif
```

演示用程序：

- Makefile应该这么写：

```
# 定义编译器
CC = gcc

# 编译参数（-Wall 显示全部警告，-g 保留调试信息）
CFLAGS = -Wall -g

# 目标程序名
TARGET = my_program

# 默认目标：生成可执行文件
all: $(TARGET)

# 链接生成目标程序
$(TARGET): main.o func.o
	$(CC) $(CFLAGS) -o $(TARGET) main.o func.o

# 编译main.c生成main.o
main.o: main.c func.h
	$(CC) $(CFLAGS) -c main.c

# 编译func.c生成func.o
func.o: func.c func.h
	$(CC) $(CFLAGS) -c func.c

# 清理生成的文件
clean:
	rm -f $(TARGET) *.o

.PHONY: all clean
```

04

服务器配置

在嵌入式系统应用开发中，需要用到文件传输工具

- tftp和nfs是在嵌入式linux开发环境中经常要用到的传输工具
- samba则是在linux和windows之间的文件传输工具。

SAMBA服务器

- Samba是在Linux/Unix系统上实现SMB(Session Message Block)协议的一个免费软件，以实现文件共享和打印机服务共享，它的工作原理与Windows网上邻居类似。
- 为了能让使用Linux操作系统的计算机和使用Wmdows操作系统的计算机共享资源，需要使用Samba工具。

NFS服务器

- NFS是网络文件系统(Network File System)的简称，是分布式计算系统的一个组成部分，可实现在多种网络上共享和装配远程文件系统。
- NFS由Sun公司开发，目前已经成为文件服务的一种标准。其最大的功能就是可以通过网络，让不同操作系统的计算机可以共享数据，所以也可以将它看做是一个文件服务器。
- NFS提供了除Samba之外，Windows与Linux及Unix与Linux之间通信的方法。

TFTP服务器

- TFTP(Trivial File Transfer Protocol，简单文件传输协议)是TCP/IP协议族中的一个用来在客户机与服务器之间进行简单文件传输的协议，提供不复杂、开销不大的文件传输服务。
- 当前TFTP有3种传输模式：netASCII模式即8位ASCII；八位组模式(替代了以前版本的二进制模式)，如原始八位字节；邮件模式，在这种模式中，传输给用户的不是文件而是字符。主机双方可以自己定义其它模式。

作业

1. 根据P35~36页的程序和Makefile文件，执行make，将结果截图。
2. 根据题1进行修改：
   - 新增两个文件：calc.c 和 calc.h；
   - 在calc.c中实现一个函数int add(int a, int b)，返回两个整数的和；
   - 在主程序main.c中调用add函数，并打印结果；
   - 修改Makefile，使得新增文件可以被正确编译并链接。
   - 最终结果截图
3. 根据课本P56 示例2.4.1-1，完成samba服务器的搭建，测试其文件共享功能并截图。

重庆理工大学/电气学院

CHONGQING UNIVERSITY OF TECHNOLOGY

嵌入式Linux系统开发教程

—基于ARM处理器通用平台（arm9-arm11-contexA系列）