第 5 章 嵌入式 Linux 程序设计基础

本章主要介绍 Linux 系统下的开发环境的建立与开发工具的使用方法。

主要内容

1. 详细介绍 Linux 下的目录结构、基本命令、常用网络服务管理配置。
2. 熟练掌握 Linux 下编辑器(vi)、编译器 GCC、调试器 GDB。
3. 熟练掌握 Linux 下项目管理工具 Make 以及 Makefile 工作原理及其编写。
4. 熟练掌握 Linux 下 shell 脚本相关知识及其编写。

重点与难点

重点：编译器 GCC 的配置方法，GDB 使用方法。

难点：Makefile 工作原理及编写方法。

本章学习目标

1. Linux 编程风格。
2. Linux 下使用 GNU cc 开发应用程序。
3. Linux 程序的调试。

5.1 Linux 系统下的开发环境以及常用工具介绍

5.1.1 Linux 编程

Linux 软件开发一直在 Internet 环境下讲行。这个环境是全球性的，编程人员来自世界各地。只要能够访问 Web 站点，就可以启动一个以 Linux 为基础的软件项目。Linux 开发工作经常是在 Linux 用户决定共同完成一个项目时开始的。当开发工作完成后，该软件就被放到 Internet 站点上，任何用户都可以访问和下载它。由于这个活跃的开发环境，新的以 Linux 为基础的软件功能日益强大，而且呈现爆炸式的增长态势。

大多数 Linux 软件是经过自由软件基金会(Free Software Foundation)提供的 GNU(GNU 即 GNU's not UNIX)公开认证授权的，因而通常被称作 GNU 软件。GNU 软件免费提供给用户使用，并被证明是非常可靠和高效的。许多流行的 Linux 实用程序如 C 编译器、shell 和编辑器都是 GNU 软件应用程序。

Linux 程序需要首先转化为低级机器语言即所谓的二进制代码以后，才能被操作系统执行。例如编程时，先用普通的编程语言生成一系列指令，这些指令可被翻译为适当的可执行应用程序的二进制代码。这个翻译过程可由解释器一步步来完成，或者也可以立即由编译器明确地完成。shell 编程语言如 BASH、TCSH、GAWK、Perl、Tcl 和 Tk 都利用自己的解释器。用这些语言编制的程序尽管是应用程序文件，但可以直接运行。编译器则不同，它将生成一个独立的二进制代码文件然后才可以运行。

1. GNU 风格

（1）函数返回类型说明和函数名分两行放置，函数起始字符和函数开头左花括号放到最左边。

（2）尽量不要让两个不同优先级的操作符出现在相同的对齐方式中，应该附加额外的括号使得代码缩进可以表示出嵌套。

（3）按照规定方式排版 do-while 语句：

（4）每个程序都应该以一段简短的说明其功能的注释开头。

（5）请为每个函数书写注释，说明函数是做什么的，需要哪些入口参数，参数可能值的含义和用途。如果用了非常见的、非标准的东西，或者可能导致函数不能工作的任何可能的值，应该进行特殊说明。如果存在重要的返回值，也需要说明。

（6）不要声明多个变量时跨行，每一行都以一个新的声明开头。

（7）当一个 if 中嵌套了另一个 if-else 时，应用花括号把 if-else 括起来。

（8）要在同一个声明中同时说明结构标识和变量或者结构标识和类型定义（typedef）。先定义变量，再使用。

（9）尽量避免在 if 的条件中进行赋值。

（10）请在名字中使用下划线以分割单词，尽量使用小写；把大写字母留给宏和枚举常量，以及根据统一惯例使用的前缀。例如，应该使用类似 ignore_space_change_flag 的名字；不要使用类似 iCantReadThis 的名字。

2. Linux 内核编程风格

（1）Linux 内核缩进风格是 8 个字符。

（2）Linux 内核风格采用 K&R 标准，将开始的大括号放在一行的最后，而将结束的大括号放在一行的第一位。

（3）命名尽量简洁。不应该使用诸如 ThisVariableIsATemporaryCounter 之类的名字。应该命名为 tmp，这样容易书写，也不难理解。但是命名全局变量，就应该用描述性命名方式，例如应该命名“count_active_users( )”，而不是“cntusr( )”。本地变量应该避免过长。

5.2　嵌入式 Linux 编译器

vi 提供了一些功能强大的但容易记忆的命令供用户使用。类似这样的编辑任务在 vi 中可以轻松高效完成。

1. vi 的三种模式

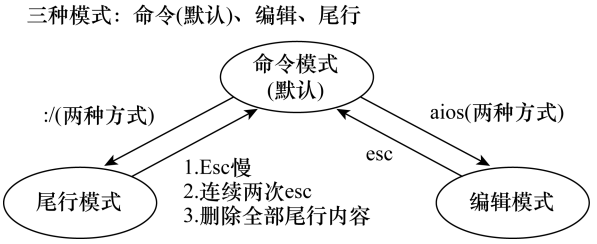

图 5.1　vi 三种模式示意图

2. 切换到编辑模式的四种方式，编辑模式可以输入任意内容

a. 光标向后移动一位。
i. 当前位置。
o. 另起新行。
s. 删除光标所在字符。
r. 替换光标所在字符。

3. 尾行模式，用于保存内容、查找替换、设置行号等功能性操作

```
: q                          //quit 退出 vi 编辑器
: w                          //write 保存修改的内容
: wq                         //保存并退出
: q!                         //强制退出，当对文本内容作了修改而不想要保存时
: w!                         //强制保存，当没有文本的写权限时
: set number  或  : set nu   //显示行号
: set nonumber  或  : set nonu //取消显示行号
: /内容/  或  /内容          //查找指定内容
                             //n 将光标移动到下一个目标
                             //N 上一个
: n                          //跳转到第 n 行
: s/targetContent/newContent //替换当前行第一个 targetContent 为 newContent
: s/targetContent/newContent/g //整行相应内容替换
: %s/targetContent/newContent <==>: 1, $ s/word1/word2/g
                             //整个文本相应内容替换
: n1, n2s/word1/word2/gc     //: 100, 200s/word1/word2/g 把 100 行到 200 行之
                               间的 word1 替换为 word2，并提示是否替换
                               c -> confirm
```

4. 命令模式

① 光标移动

a. 字符级

左(h)　下(j)　上(k)　右(l)

b. 单词级

w　word 移动到下个单词首字母

b　before 上个单词首字母

e　end 下个单词结尾

c. 行级

0 行首

$ 行尾

d. 段落级{ 上 } 下(没必要记忆)

e. 屏幕级　H 屏首　L 屏尾(没必要记忆)

f. 文档级

G　文档尾部

nG　文档第 n 行

gg　文档第一行

crtl + f <--> pagedown 向下翻页

crtl + b <--> pageup 向上翻页

n + enter　向下移动 n 行

② 内容删除

```
dd          //删除当前行
ndd         //自当前行向下删除 n 行
x           //删除当前字符
cw          //删除光标所在字母后面的字符
```

③ 内容复制

```
yy          //复制光标当前行
nyy         //自当前行复制 n 行
p           //对(删除)复制的内容进行粘贴
```

④ 相关快捷操作

```
u           //撤销
.           //重复上次操作
```

5.2.1　Linux 下 C 语言编译过程

当用 gcc 编译 C 代码时，它会试着用最少的时间完成编译并且使编译后的代码易于调试，易于调试意味着编译后的代码与源代码有同样的执行次序，编译后的代码没有经过优化。有很多选项可用于告诉 gcc，在耗费更多编译时间和牺牲易调试性的基础上，产生更小更快的可执行文件。这些选项中最典型的是 -O 和 -O2 选项。

-O 选项告诉 gcc 对源代码进行基本优化。这些优化在大多数情况下都会使程序执行的更快。

-O2 选项告诉 gcc 产生尽可能小和尽可能快的代码。-O2 选项将使编译的速度比使

用 -O 时慢。但通常产生的代码执行速度会更快。

5.2.2 GCC 编译器

1. 使用 GNU cc

gcc 可以使程序员灵活地控制编译过程。编译过程一般可以分为下面四个阶段，每个阶段分别调用不同的工具进行处理。

2. gcc 的版本信息

一般来说，系统安装后就已经安装和设定好了 gcc。在 shell 的提示符下键入 gcc v，屏幕上就会显示出目前正在使用的 gcc 的版本，同时这可以确定系统所支持的是 ELF 还是 a. out 可执行文件格式。

Linux 系统中可执行文件有两种格式。第一种格式是 a. out 格式，这种格式用于早期的 Linux 系统以及 Unix 系统的原始格式。a. out 来自于 Unix C 编译程序默认的可执行文件名。当使用共享库时，a. out 格式就会发生问题。把 a. out 格式调整为共享库是一种非常复杂的操作，由于这个原因，一种新的文件格式被引入 Unix 系统 5 的第四版本和 Solaris 系统中。它被称为可执行和连接的格式(ELF)。这种格式很容易实现共享库。

ELF 格式已经被 Linux 系统作为标准的格式采用。gcc 编译程序产生的所有的二进制文件都是 ELF 格式的文件(即使可执行文件的默认名仍然是 a. out)。较旧的 a. out 格式的程序仍然可以运行在支持 ELF 格式的系统上。

gcc 的使用格式如下：

```
$ gcc [options][filenames]
```

其中 filenames 为所要编译的程序源文件。

当使用 gcc 时，gcc 会完成预处理、编译、汇编和连接。前三步分别生成目标文件，连接时，把生成的目标文件链接成可执行文件。gcc 可以针对支持不同的源程序文件进行不同处理，文件格式以文件的后缀来识别。

GCC 支持数种调试和剖析选项。在这些选项里最常用的是 -g 和 -pg 选项。

-g 选项告诉 gcc 产生能被 GNU 调试器使用的调试信息以便调试程序。gcc 提供了一个很多其他 C 编译器里没有的特性，在 gcc 里能使 -g 和 -O(产生优化代码)连用。这一点非常有用，因为能在与最终产品尽可能相近的情况下调试代码。同时使用这两个选项时必须清楚所写的某些代码已经在优化时被 gcc 作了改动。

-pg 选项告诉 gcc 在程序里加入额外的代码，执行时，产生 gprof 用的剖析信息以显示程序的耗时情况。

5.2.3 GDB 调试技术

启动程序准备调试

gdb yourpram

或者

先输入 gdb

然后输入 file yourpram

然后使用 run 或者 r 命令开始程序的执行，也可以使用 run parameter 将参数传递给该程序表 5.1。

表 5.1　参数列表

命令	命令缩写	命令说明
list	l	显示多行源代码
break	b	设置断点，程序运行到断点的位置会停下来
info	i	描述程序的状态
run	r	开始运行程序
display	disp	跟踪查看某个变量，每次停下来都显示它的值
step	s	执行下一条语句，如果该语句为函数调用，则进入函数执行其中的第一条语句
next	n	执行下一条语句，如果该语句为函数调用，不会进入函数内部执行（即不会一步步地调试函数内部语句）
print	p	打印内部变量值
continue	c	继续程序的运行，直到遇到下一个断点
set var name = v		设置变量的值
start	st	开始执行程序，在 main 函数的第一条语句前面停下来
file		装入需要调试的程序
kill	k	终止正在调试的程序
watch		监视变量值的变化
backtrace	bt	产看函数调用信息(堆栈)
frame	f	查看栈帧
quit	q	退出 GDB 环境

gcc-g-o e e. c

调试 gdb e

或者输入 gdb

然后 file e

list 命令用法

list 命令显示多行源代码，从上次的位置开始显示，默认情况下，一次显示 10 行，第一次使用时，从代码其实位置显示。

```
(gdb) list
1       #include < stdio. h >
2       void debug(char*str)
```

```
3     {
4           printf("debug info: %s\n", str );
5     }
6           main(int argc,char*argv[ ]){
7           int i, j;
8           j = 0;
9           for(i = 0; i < 10; i ++ ){
10                j + = 5;
```

断点命令 break

break location：在 location 位置设置断点，改位置可以为某一行，某函数名或者其他结构的地址 gdb 会在执行该位置的代码之前停下来。

5.3 Makefile 使用

使用 Makefile 编译代码

```
$ make
```

运行 helloworld

```
$ ./helloworld
Hello, The World!
```

这样 helloworld 就编译出来了，你如果按上面的步骤来做的话，应该也会很容易地编译出正确的 helloworld 文件。你还可以试着使用一些其他的 make 命令，如 make clean，make install，make dist，看看它们会给你什么样的效果。

5.3.1 Makefile 基本原理

Makefile 是用于自动编译和链接的，一个工程由很多文件组成，每一个文件的改变都会导致工程的重新链接，但是不是所有的文件都需要重新编译，Makefile 中纪录有文件的信息，在 make 时会决定在链接的时候需要重新编译哪些文件。

Makefile 的宗旨就是：让编译器知道要编译一个文件需要依赖其他的哪些文件。当那些依赖文件有了改变，编译器会自动的发现最终的生成文件已经过时，而重新编译相应的模块。

Makefile 的基本结构不是很复杂，但当一个程序开发人员开始写 Makefile 时，经常会怀疑自己写的是否符合惯例，而且自己写的 Makefile 经常和自己的开发环境相关联，当系统环境变量或路径发生了变化后，Makefile 可能还要跟着修改。这样就造成了手工书写 Makefile 的诸多问题，automake 恰好能很好地帮助我们解决这些问题。

使用 automake，程序开发人员只需要写一些简单的含有预定义宏的文件，由 autoconf 根据一个宏文件生成 configure，由 automake 根据另一个宏文件生成 Makefile. in，再使用 configure 依据 Makefile. in 来生成一个符合惯例的 Makefile 。下面我们将详细介绍 Makefile

的 automake 生成方法。

5.3.2　Makefile 变量

Makefile 里的变量就像一个环境变量。事实上，环境变量在 make 中也被解释成 make 的变量。这些变量对大小写敏感，一般使用大写字母。几乎可以从任何地方引用定义的变量，变量的主要作用如下：

保存文件名列表。在前面的例子里，作为依赖文件的一些目标文件名出现在可执行文件的规则中，而在这个规则的命令行里同样包含这些文件并传递给 gcc 做为命令参数。如果使用一个变量来保存所有的目标文件名，则可以方便地加入新的目标文件而且不易出错。

保存可执行命令名，如编译器。在不同的 Linux 系统中存在着很多相似的编译器系统，这些系统在某些地方会有细微的差别，如果项目被用在一个非 gcc 的系统里，则必须将所有出现编译器名的地方改成用新的编译器名。但是如果使用一个变量来代替编译器名，那么只需要改变该变量的值。其他所有地方的命令名就都改变了。

保存编译器的参数。在很多源代码编译时，gcc 需要很长的参数选项，在很多情况下，所有的编译命令使用一组相同的选项，如果把这组选项使用一个变量代表，那么可以把这个变量放在所有引用编译器的地方。当要改变选项的时候，只需改变一次这个变量的内容即可。

Makefile 中的变量是用一个文本串在 Makefile 中定义的，这个文本串就是变量的值。只要在一行的开始写下这个变量的名字，后面跟一个“ = ”号，以及要设定这个变量的值即可定义变量，下面是定义变量的语法：

```
VARNAME = string
```

使用时，把变量用括号括起来，并在前面加上 $ 符号，就可以引用变量的值：

```
$ {VARNAME}
```

make 解释规则时，VARNAME 在等式右端展开为定义它的字符串。变量一般都在 Makefile 的头部定义。按照惯例，所有的 Makefile 变量都应该是大写。如果变量的值发生变化，就只需要在一个地方修改，从而简化了 Makefile 的维护。

现在利用变量把前面的 Makefile 重写一遍：

```
OBJS = prog. o code. o
CC = gcc
test: $ { OBJS }
    $ { CC }-o test  $ { OBJS }
prog. o:prog. c prog. h code. h
    $ { CC }-c prog. c-o prog. o
code. o:code. c code. h
    $ { CC }-c code. c-o code. o
clean:
    rm-f*. o
```

除用户自定义的变量外，make 还允许使用环境变量、自动变量和预定义变量。使用环境变量的方法很简单，在 make 启动时，make 读取系统当前已定义的环境变量，并且创建与之同名同值的变量，因此用户可以像在 shell 中一样在 Makefile 中方便的引用环境变量。需要注意的是，如果用户在 Makefile 中定义了同名的变量，用户自定义变量将覆盖同名的环境变量。此外，Makefile 中还有一些预定义变量和自动变量，但是看起来并不像自定义变量那样直观。

5.3.3 Makefile 规则

在上面的例子中，几个产生目标文件的命令都是从“.c”的 C 语言源文件和相关文件通过编译产生“.o”目标文件，这也是一般的步骤。实际上，make 可以使工作更加自动化，也就是说，make 知道一些默认的动作，它有一些称作隐含规则的内置的规则，这些规则告诉 make 当用户没有完整地给出某些命令的时候，应该怎样执行。

例如，把生成 prog.o 和 code.o 的命令从规则中删除，make 将会查找隐含规则，然后会找到并执行一个适当的命令。由于这些命令会使用一些变量，因此可以通过改变这些变量来定制 make。象在前面的例子中所定义的那样，make 使用变量 CC 来定义编译器，并且传递变量 CFLAGS(编译器参数)、CPPFLAGS(C 语言预处理器参数)、TARGET_ARCH(目标机器的结构定义)给编译器，然后加上参数 -c，后面跟变量 $ <(第一个依赖文件名)，然后是参数 -o 加变量 $ @(目标文件名)。综上所述，一个 C 编译的具体命令将会是：

$ {CC} $ {CFLAGS} $ {CPPFLAGS} $ {TARGET_ARCH} -c $ <- o $ @在上面的例子中,利用隐含规则,可以简化为：

```
OBJS = prog.o code.o
CC = gcc
test: ${OBJS}
	${CC}-o $@ $^
prog.o:prog.c prog.h code.h
code.o:code.c code.h
clean:
	rm-f*.o
```

5.3.4 Make 命令的使用

make install

将编译成功的可执行文件安装到系统目录中，一般为/usr/local/bin 目录。

make dist

产生发布软件包文件(即 distribution package)。这个命令将会将可执行文件及相关文件打包成一个 tar.gz 压缩的文件用来作为发布软件的软件包。

它会在当前目录下生成一个名字类似“PACKAGE-VERSION.tar.gz”的文件。

PACKAGE 和 VERSION，是我们在 configure. in 中定义的 AM _ INIT _ AUTOMAKE（PACKAGE，VERSION）。

make distcheck

生成发布软件包并对其进行测试检查，以确定发布包的正确性。这个操作将自动把压缩包文件解开，然后执行 configure 命令，并且执行 make，来确认编译不出现错误，最后提示你软件包已经准备好，可以发布了。

```
==========================================
helloworld - 1. 0. tar. gz is ready for distribution
==========================================
```

make distclean

类似 make clean，但同时也将 configure 生成的文件全部删除掉，包括 Makefile 。

通过上面的介绍，应该可以很容易地生成一个你自己的符合 GNU 惯例的 Makefile 文件及对应的项目文件。

如果想写出更复杂的且符合惯例的 Makefile，可以参考一些开放代码的项目中的 configure. in 和 Makefile. am 文件，比如：嵌入式数据库 sqlite，单元测试 cppunit 。

5. 3. 5　autoconf 和 automake 生成 Makefile

autoconf 是用来产生 configure 文件的。configure 是一个脚本，它能设置源程序来适应各种不同的操作系统平台，并且根据不同的系统来产生合适的 Makefile，从而可以使你的源代码能在不同的操作系统平台上被编译出来。

configure. in 文件的内容是一些宏，这些宏经过 autoconf 处理后会变成检查系统特性、环境变量、软件必须的参数的 shell 脚本。configure. in 文件中的宏的顺序并没有规定，但是你必须在所有宏的最前面和最后面分别加上 AC_INIT 宏和 AC_OUTPUT 宏。在 configure. ini 中：

号表示注释，这个宏后面的内容将被忽略。

AC_INIT(FILE)

这个宏用来检查源代码所在的路径。

AM_INIT_AUTOMAKE(PACKAGE，VERSION)

这个宏是必须的，它描述了我们将要生成的软件包的名字及其版本号：PACKAGE 是软件包的名字，VERSION 是版本号。当你使用 make dist 命令时，它会给你生成一个类似 helloworld - 1. 0. tar. gz 的软件发行包，其中就有对应的软件包的名字和版本号。

AC_PROG_CC

这个宏将检查系统所用的 C 编译器。

AC_OUTPUT(FILE)

这个宏是我们要输出的 Makefile 的名字。

我们在使用 automake 时，实际上还需要用到其他的一些宏，但我们可以用 aclocal 来帮我们自动产生。执行 aclocal 后我们会得到 aclocal. m4 文件。

产生了 configure. in 和 aclocal. m4 两个宏文件后，我们就可以使用 autoconf 来产生 configure 文件了。

automake

我们使用 automake --add-missing 来产生 Makefile. in 。

选项 --add-missing 的定义是“add missing standard files to package”，它会让 automake 加入一个标准的软件包所必须的一些文件。

我们用 automake 产生出来的 Makefile. in 文件是符合 GNU Makefile 惯例的，接下来我们只要执行 configure 这个 shell 脚本就可以产生合适的 Makefile 文件了。

本章小结

Make 是用于自动编译、链接程序的实用工具，使用 make 后就不需要手工的编译每个程序文件。要使用 make，首先要编写 makefile。

Makefile 描述程序文件之间的依赖关系，并提供更新文件的命令。在一个程序中，可执行文件依赖于目标文件，而目标文件依赖于源文件。如果 makefile 文件存在，每次修改完源程序后，用户通常所需要做的事情就是在命令行敲入“make”，然后所有的事情都由 make 来完成。

习　题

1. 什么是嵌入式操作系统？
2. 与通用计算机相比，嵌入式系统有哪些特点？
3. 根据嵌入式的复杂程度，嵌入式系统可分为哪 4 类？
4. 举例介绍嵌入式处理器有哪几类？
5. 从硬件系统来看，嵌入式系统由哪几个部分组成？画出简图。
6. 嵌入式系统的定义是什么？嵌入式系统具有哪些主要特点？
7. 嵌入式系统与传统的单片机系统在软件和硬件上有哪些主要的不同？
8. 常见的处理器有哪些类型，各有什么特点。各类处理器主要应用在哪些领域？
9. 在嵌入式系统中，操作系统具有怎样主要功能和特点？
10. 常见的实时操作系统有哪些，各有什么特点，具体应用在哪些领域？
11. 设计嵌入式系统时，在选择嵌入式处理器和实时操作系统时，分别考虑哪些主要因素？