

本章主要内容

- Linux 编程环境介绍
- VIM 及 Emacs 的使用
- GNU make 使用及 Makefile 编程
- GDB 的使用方法
- 本章小结

3.1 Linux 编程环境介绍

众所周知,程序的运行需要操作系统支持,程序的优化必然也要考虑到操作系统的特性。创建一个进程要比创建一个线程耗用更多的时间和空间资源,这也使得在 Windows 中开发多任务程序时,往往采用多线程来实现。在 Linux 下创建进程的 fork 函数非常高效,因此在 Linux 下频繁使用它。如果要想编写高效、安全、稳定、易维护的软件,不考虑特定的系统平台几乎是不可能的。

3.1.1 开发工具环境

现在的程序员们早已习惯了可视化的编程工具,习惯使用“向导”。系统将进行校对,提醒代码是否有错,完成后,单击“编译”按钮,就可以生成可执行程序。也可以画出用户界面,生成基本的程序框架,然后根据需要加以修改,就完成了程序。这就是在 Windows 环境下的程序员所享受的生活。而 Linux 世界却是另一个世界,一切都显得那样的原始、古朴、原汁原味。一定会勾起那些从 DOS 世界,或更早的世界中走出来的程序员对往事的回忆,那些来自 UNIX 世界的程序员都会感到无比的亲切。

Windows 一面以友好的界面展现给程序员,但却严格限制程序员对其透彻地研究,将自己用华丽的外表包装起来。而 Linux 则一直以真面目示人——神秘、费解,而内心是对刻苦者敞开的。

一套完整的开发工具应包括编辑工具、编译工具和调试工具,大型项目还要有配置工具和项目管理工具。开发环境分为基于文本的开发平台(vim/emacs+gcc+gdb)和集成开发平台(Eclipse+CDT 插件)。近年来,Linux 受到越来越多的开发者和普通用户的喜爱,Linux 上的集成开发工具也越来越多,比较常用的有 Kylix、Eclipse 等。



3.1.2 基于文本模式的开发平台

(1) 编辑工具。早些时候,Linux 中尚未拥有集成化环境,开发者们使用经典的 vi 来编辑源程序,更复杂一点的有 joe、emacs 等。

(2) 编译工具。Linux 支持大量的语言有 C、C++、Java、Pascal 等,在本书中以 C 语言为主。一般用命令行的方式使用编译工具,先用编辑工具输入源程序,然后再执行一长串包括参数的命令进行编译,最后还需要为其赋予可执行的权限,这样才完成了整个工作。

(3) 调试工具。程序运行中,发现存在缺陷,就需要确定出错的位置、出错的原因以及一些运行时的数据。这时,就需要 gdb 的帮助,通过 gdb 调试程序,可以查看程序运行中某一变量值,支持断点调试等功能。

如果程序分成很多源文件,就不得不先把每个源文件都编译成目标代码,最后再链接成可执行文件。为了完成这种繁重的重复性劳动,可以使用 make 的工具。make 依据一个 Makefile 文档来工作,而一个简单的 Makefile 文档其实就是 gcc 命令的集合。编辑好之后,输入 make 就可以让它自动运行这些编译命令。Makefile 文档中一个重要的概念就是目标,它告诉 make 要完成什么工作,后续章节将详细论述 make。

对于稍微大一点的程序,大家都不会自己去写 Makefile 文档,而是用 automake 的工具来自动生成 Makefile,当然也会用到其他一些如 autoscan、autolocal、autoconf 等工具。虽然标准的 GNU 软件推荐使用这些工具,但初学者完全可以跳过它们,只需对 make 的工作方式有基本的了解就可以了。为了供不同的开发人员之间的分工合作,Linux 还提供了 CVS,用于版本控制、软件配置管理。

3.1.3 集成开发平台 Eclipse+CDT

Eclipse 是一个由 IBM、Borland 等公司资助的开源开发环境,其功能可以通过插件方式进行扩展。尽管 Eclipse 主要用于 Java 程序开发,但其体系结构确保了它对其他程序语言的支持。Eclipse CDT(Eclipse C/C++ Development Tool)是用于 C/C++ 程序开发的一组插件,CDT 项目致力于为 Eclipse 平台提供功能完整的 C/C++ 集成开发环境,该项目重点面向 Linux 平台。

在下载和安装 CDT 之前,首先必须确保 GNU C 编译器 GCC 及所有附带的工具(make、binutil 和 GDB)都是可用的。如果正在运行 Linux,只要通过使用适于你分发版的软件包管理器来安装开发软件包。Eclipse 可从官方网站(<http://www.eclipse.org/>)下载,网站上也有 CDT 的下载链接。需注意的是,不同版本的 Eclipse 需要特定版本的 CDT 插件的支持。

下一步是下载 CDT 二进制文件。由于 CDT 与平台有关,选择下载相关的操作系统的 CDT,然后,将归档文件解压到临时目录中,从临时目录将所有插件目录内容都移到 Eclipse plugins 子目录下。还需要将 features 目录内容移到 Eclipse features 子目录中。现在,重新启动 Eclipse。Eclipse 再次启动之后,更新管理器将告诉你它发现了更改,并询问是否确认这些更改。现在你将能够看到两个可用的新项目:C 和 C++。

在 Eclipse 中安装 CDT 之后,浏览至 File|New|Project,可看到 3 个新的可用项目类型:C(Standard C Make Project)、C++(Standard C++ Make Project) 和 Convert to C or

C++ Projects。从 Standard Make C++ Project 开始,可创建源代码。在 C/C++ Projects 视图中,单击鼠标右键,然后选择 New | Simple | File,对文件命名并保存。你可能会用这种方法创建许多头文件及 C/C++ 实现代码文件。最后是 Makefile,GNU make 将使用它来构建二进制文件。对该 Makefile 使用常见的 GNU make 语法。

通常会将现有的源代码导入 Eclipse,可使用 Import 向导,将文件从文件系统目录复制到工作台。转至主菜单栏,选择 File | Import | File System。单击 Next 按钮,打开源目录,选择要添加文件的目录。单击 Select All 按钮以选择目录中的所有资源,然后从头到尾检查,取消不需要添加的资源,指定将作为导入目标的工作台项目或文件夹。还可以通过从文件系统拖动文件夹和文件并将它们放入 Navigatoin 视图中,或者通过复制和粘贴来导入文件夹和文件。

CDT 依赖于三个 GNU 工具:GCC、GDB 和 Make。大多数 Linux 源代码软件包使用 autoconf 脚本来检查构建环境,所以必须运行 configure 命令,该命令在编译之前创建 Makefile。CDT 没有提供编辑 autoconf 脚本的方法,所以必须手动编写。然而,你可以配置构建选项以在编译之前调用 configure 命令。

可以使用默认设置通过调用 make 命令来构建项目。如果要使用更复杂的方法进行构建,则必须在 Build Command 文本框中输入适当的命令(例如,make -f make_it_all)。

接下来,在 C/C++ Project 视图中,选择 C/C++ Project,然后单击鼠标右键并选择 Rebuild Project。所有来自 make、编译器和链接程序的编译消息都重定向到控制台窗口。

编译成功之后,就可以运行应用程序,用于运行和调试的选项都位于 Eclipse 主菜单的 Run 菜单下。必须预先定义好用于运行项目的选项。通过转至主菜单中的 Run 选项来完成这一步。例如,可以将一个概要文件用于测试目的,而将另一个概要文件用于运行最终版本。另外,可以定义希望要传递给应用程序的参数,或者设置环境变量。其他选项用于设置调试选项,例如使用哪个调试器(GNU GDB 或 Cygwin GDB)。

3.1.4 文档帮助环境

Windows 程序开发人员如果遇到不太熟悉的函数,可以求助于 MSDN。在 Linux 下,虽然最初的文档很不齐全且管理混乱,给开发人员带来了不少困难。但随着 Linux 文档计划(见网址 <http://www.tldp.org/>)的开展,以及对原版英文文档的翻译,Linux 文档逐渐丰富起来。

在 Linux 下开发应用程序时,手册页(manpage)是主要的参考信息来源。手册页中存放的是参考信息,对于每一条 shell 命令、系统调用、库函数、配置文件和系统的守护程序,都有相关的一页对其进行说明。手册页分为 8 个部分:

- 第 1 部分: shell 命令和用户级程序;
- 第 2 部分: 系统调用相关文档;
- 第 3 部分: C 和 C++ 库函数和宏调用相关文档;
- 第 4 部分: 在内核模块、/dev 目录、/proc 等目录中的特殊文件和设备的相关文档;
- 第 5 部分: 系统的不同文件格式;
- 第 6 部分: 因历史原因而包含的游戏相关文档;
- 第 7 部分: 有关语言或小语言的文档;

- 第8部分：守护程序或者其他系统管理员命令的相关文档。

可以在 shell 提示下输入命令 man 和命令的名称,来阅读有关的说明书页。例如,要阅读关于 ls 命令的说明书页,输入以下命令:

```
[root@localhost ~]# man ls
```

会显示如下信息:

```
LS(1)      User Commands      LS(1)
NAME
ls - list directory contents
SYNOPSIS
ls [OPTION] ... [FILE] ...
DESCRIPTION
List information about the FILES (the current directory by default).
Sort entries alphabetically if none of -cftuSUX nor --sort is specified

Mandatory arguments to long options are mandatory for short options
too.
a, --all
    do not hide entries starting with -
A, --almost-all
    do not list implied. and..
--author
    with -l; print the author of each file
...
```

其中,NAME 字段显示了可执行文件的名称和对其功能的简短解释;SYNOPSIS 字段显示了可执行文件的常用方法,如要声明的选项和它支持的输入类型(文件或数值);DESCRIPTION 字段显示了和文件或可执行文件相关的可用选项和数值;See Also 显示了相关的术语、文件和程序。

如果要翻阅说明书,可以使用 Page Down 和 Page Up 键,或使用空格键来向后翻一页,使用字符“b”来向前翻。要退出说明书页,输入字符“q”。要在说明书中搜索关键字,输入“/”和要搜索的关键字或短语,然后按 Enter 键。所有出现在说明书中的关键字都会被突出显示,并允许快速地阅读上下文中的关键字。打印说明书页是给常用命令归档的便利方法,可以把它们装订起来以便使用。如果你有一台打印机,并在 Linux 中配置了,就可以在 shell 提示下输入以下命令来打印说明书:

```
[root@localhost ~]# man ls | col -b | lpr
```

以上命令把分开的命令并入一个独特的功能。man ls 会把 ls 的说明书的内容输出给 col,这个命令会格式化内容,使其适合打印页的大小。lpr 命令把格式化后的内容发送给打印机。

3.2 常用编辑器

常用的编辑器有:图形模式下的 gedit、kwrite、OpenOffice,文本模式下的 VIM、Emacs 和 nano。VIM 和 Emacs 是在 Linux 下最常用的两个编辑器。Emacs 的功能比 VIM 多。下面简单介绍一下两者的常用功能,更多的功能可以查阅帮助文档。

3.2.1 VIM 编辑器

VIM 是 Linux 最基本的文本编辑工具, 虽然没有类似于图形界面编辑器中单击鼠标的简单操作, 但在系统管理、服务器管理中, 永远不是图形界面的编辑器能比的。当没有安装 Windows 桌面环境或桌面环境崩溃时, 字符模式下的编辑器 VIM 就派上用场了。另外, VIM 编辑器是创建和编辑简单文档最高效的工具。

1. VIM 的模式

VIM 的模式有 6 种, 为避免初学者搞混, 一般分成三种:

(1) 一般模式。进入 VIM 就处于一般模式, 只能通过按键向编辑器发送命令, 不能输入文字。这些命令可能是移动光标的命令, 也可能是编辑命令或寻找替换命令。

(2) 编辑模式。在一般模式下按 i 键就会进入编辑模式(也称插入模式), 此时可以输入文字、写文章, 按 Esc 键就又回到一般模式。

(3) 命令模式。在一般模式下按冒号键“:”就会进入命令模式, 屏幕左下角会有一个冒号出现, 此时可输入命令并执行。同样地, 按 Esc 键回到一般模式。

2. VIM 的启动保存和退出

(1) 在命令行中指定打开文件。例如打开 test.txt 文件, 输入 vim test.txt 即可。此时 VIM 处于一般模式, 也是其默认模式。

(2) 先进入 VIM 后打开文件。进入 VIM 后, 进入命令模式, 使用冒号命令: e test.txt, 就可以编辑 test.txt 这个文件。使用以上两种打开文件方式时, 如果 test.txt 不存在, 就会打开一个新的以 test.txt 命名的文件。

(3) 编写文件。进入 VIM 后, 按 i 键进入编辑模式, 就可以编写文件了。通过方向键控制光标的移动, 退格键可消去光标前一个字母, 若是中文则消去一个字。Delete 键可删除光标所在处的字母(或汉字)。

(4) 保存文件和退出。如果写好了文件, 就可以先按 Esc 键回到一般模式, 然后使用冒号命令:w, 就会保存文件, 但这时还没有离开 VIM, 如果要离开 VIM, 则可以通过冒号命令:q!。另外, 也可以把以上两个命令合起来使用, 如: wq, 这样就会存盘并退出。

3. 光标快速移动

当 VIM 处于一般模式下时, 可以用下列快捷键位来快速移动光标:

j	向下移动一行;
k	向上移动一行;
h	向左移动一个字符;
l	向右移动一个字符;
ctrl+b	向上移动一屏;
ctrl+f	向下移动一屏;
向上箭头	向上移动;
向下箭头	向下移动;
向左箭头	向左移动;
向右箭头	向右移动。

对于 j、k、l 和 h 键, 还能在这些动作命令前面加上数字, 例如 3j, 表示向下移动 3 行。

4. 文本的插入

进行此类操作前, VIM 应处于一般模式, 操作后 VIM 处于编辑模式:

- i 在光标之前插入;
- a 在光标之后插入;
- I 在光标所在行的行首插入;
- A 在光标所在行的行末插入;
- o 在光标所在行的下面插入一行;
- O 在光标所在行的上面插入一行;
- s 删除光标后的一个字符, 然后进入插入模式;
- S 删除光标所在的行, 然后进入插入模式。

5. 文本内容的删除操作

进行此类操作前, VIM 应处于一般模式, 操作后 VIM 仍处于一般模式:

- x 删除一个字符;
- # x 删除几个字符, # 表示数字, 例如 3x;
- dw 删除一个单词;
- # dw 删除几个单词, 使用数字表示, 例如 3dw 表示删除三个单词;
- dd 删除一行;
- # dd 删除多行, # 表示数字, 例如 3dd 表示删除光标行及光标的下两行;
- d \$ 删除光标到行尾的内容;
- J 清除光标所处的行与下一行之间的空格, 把光标行和下一行接在一起。

6. 恢复修改及恢复删除操作

- u 恢复修改及恢复删除操作。

进行此操作前, VIM 应处于一般模式, 操作后 VIM 仍处于一般模式。在一般模式下按 u 键来撤销以前的删除或修改; 如果希望撤销多个以前的修改或删除操作, 需要多按几次 u 键。这和 Word 的撤销操作没有太大的区别。

7. 复制和粘贴的操作

“删除”有“剪切”的含义, 当删除文字后, 按下 Shift+p 组合键就把内容贴在原处, 然后再移动光标到某处, 按 p 键或 Shift+p 组合键就能粘贴上了。

- p 在光标之后粘贴;
- Shift+p 在光标之前粘贴。

例如, 希望把一个文档的第三行复制下来, 然后粘贴到第五行的后面, 先把光标移动到第三行处, 然后用 dd 动作, 接着再按一下 Shift+p 组合键, 这样就把刚才删除的第三行粘贴在原处了。接着再用 k 键移动光标到第五行, 然后再按一下 p 键, 这样就把第三行的内容又粘贴到第五行的后面了。要注意的是: 此处所有的操作都在一般模式下完成。

8. 查找

首先进入命令模式; 再输入 / 或 ? 就可以使用查找功能了。

- / 要查找的单词 正向查找, 按 n 键把光标移动到下一个符合条件的地方;
- ? 要查找的单词 反向查找, 按 Shift+n 组合键, 把光标移动到下一个符合条件的地方。

9. 替换

按 Esc 键进入命令模式。

:s/SEARCH/REPLACE/g 注：把当前光标所处的行中的 SEARCH 单词，替换成 REPLACE，并把所有 SEARCH 高亮显示；

:%s SEARCH/REPLACE 注：把文档中所有 SEARCH 替换成 REPLACE；

:#, #s/SEARCH/REPLACE/g 注：#号表示数字，表示从多少行到多少行，把 SEARCH 替换成 REPLACE 在这里，g 表示全局查找；注意到，即使没有替换的地方，也会把 SEARCH 高亮显示。

10. 关于行号

当编译程序出现错误或警告时，就需要转到对应的行号去编辑源文件，VIM 可以方便地显示行号。

在命令模式下，输入 set number，就会在每行的行首显示出行号。如下所示：

```
27 安装 ethtool - 3 - 1. i386.  
28 安装 expat - 1. 95. 8 - 6. i386.  
29 安装 gdbm - 1. 8. 0 - 25. i386.
```

以上介绍的只是 VIM 创建、编辑和修改文件使用的最基本的命令格式，也没有涉及更为高级的 VIM 用法。它包含的命令还很多，如果把 VIM 所有的功能都一一列出来，足够单独出一本书了。如果想了解更多，请查找 man 或 help：

`man vim` 或 `vim -help`

3.2.2 Emacs 编辑器

作为一个文本编辑器，Emacs 的确是太庞大了（有 70 多兆字节）。但是，如果把 Emacs 视为一个环境，则是非常优秀的，70MB 的体积也就不算什么了。作为普通用户，不推荐使用 Emacs，用 VIM 就可以了。但是如果你是一个程序员或系统管理员，你所关心的就不会是绚丽的界面，而是强大的功能和工作的效率——而这就是 Emacs 能带给你的。Emacs 并不比惯用的其他编辑器，如 UltraEdit、TextPad、EmEditor 等更难使用，只是在使用 Emacs 的时候，需要重新适应 Emacs 定义的快捷键。一旦你熟悉了它的快捷键，就能像用其他软件一样自如。

1. 概念介绍

为了方便以后的学习，在介绍 Emacs 使用方法之前，先介绍以下几个概念：

(1) 缓冲区(buffer)。缓冲区是一块用来保存输入的数据的内存区域。在 Emacs 里，一切都是在内存中进行的，直到按下 C-x, C-s 键来保存，文件才会被改变，几乎所有的文本编辑器都是这样工作的。

(2) 窗口(frame)。指所编辑的文本被显示的区域。这一点类似于在 UltraEdit 里打开的各个文件所在的小窗口。

(3) 模式(mode)。模式是 Emacs 里最重要的概念，Emacs 的强大功能基本上都是由各种模式提供的。常用的有 C/C++ 模式、shell 模式、Perl 模式、SGML/HTML 模式等。



2. 文件缓冲区和窗口操作

首先,可以在 Emacs 里同时编辑多个文件。随时可以使用 C-x 和 C-f(即 Ctrl+x 和 Ctrl+f)来打开(或者创建)文件。默认情况下,编辑器自动进入到新的文件窗口中。如果希望同时看到两个文件,就必须首先对窗口进行分割。使用 C-x2 键对窗口进行水平分割。分割完毕的两个窗口里的内容竟然完全一样。只是分割了窗口,但是并没有切换缓冲区,因此依旧是显示原来缓冲区的内容。使用 C-xo 键切换到想去看的窗口,然后在缓冲区列表里选择目标文件。这样就可以在同一屏中审视两个文件了。也可以用 C-x3 键将屏幕垂直分割成左右两个区域。理论上来说,窗口可以无限分割,因此完全可以将屏幕分割成倒“品”字形,只需依次按下 C-x2、C-x3 键即可。

窗口和缓冲区的概念是完全不同的,因此可以“关闭”窗口,而非“关闭”缓冲区,让它暂时从我们的视线里消失,这相当于图形环境下的“最小化窗口”。使用 C-x0 键关闭当前窗口,使用 C-x1 键关闭当前窗口以外的其他窗口。

下面的内容很直观地显示了对窗口和缓冲区的键盘操作。

窗口操作如下:

功能键	功能
C-x0	删除当前窗口,对缓冲区无影响。注意:这里是数字(最小化当前窗口)。
C-x1	删除当前以外的所有窗口,对缓冲区无影响(最小化其他窗口)。
C-x2	水平分割当前窗口。
C-x3	垂直分割当前窗口。
C-xo	切换窗口(当且仅当有一个以上的窗口存在)。注意:这里是字母 o。

缓冲区操作如下:

功能键	功能
C-x C-f	打开(创建)文件,创建一个新的缓冲区。
C-x C-s	保存当前缓冲区到文件。
C-x C-w	保存当前缓冲区到其他文件(文件另存为)。
C-x k	关闭当前缓冲区。
C-x C-b	缓冲区列表。你可以用方向键来选择要切换的缓冲区。
C-x C-c	关闭所有的缓冲区,退出 Emacs。

3. 简单的光标移动操作

从当前位置出发,移动光标的命令如图 3.1 所示(其中 C 表示 Ctrl 键,M 表示 Alt 键)。

4. 插入与删除

如果要插入文字,输入它即可。可以看到的字符,像是 A、7、* 等被 Emacs 视为文字并且可以直接插入。输入<Return>(carriage-return 键)以插入一个 Newline 字符。输入<Delete>键以删除光标处的字符。当一行文字变得比“在窗格中的一行”长时,这一行文字会“接续”到下一行。这时,一个小小的弯弯的箭头会出现在其右边界,以指出此行未完。如果连续输入几个相同的字符(例如 *****),可以利用命令 C-u 8 * 实现。

现在已经学到了输入字符到 Emacs 及修正错误的大部分基本方法。当然也可以以字或行为单位进行删除。这里有份关于删除操作的摘要:

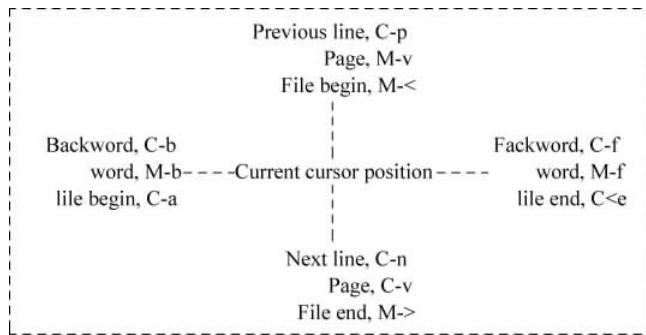


图 3.1 移动光标的命令

- < BackSpace > 删除光标所在的前一个字符。
- C-d 删除光标所在的后一个字符。
- M-< BackSpace > 删除光标所在的前一个字。
- M-d 删除光标所在的后一个字。
- C-k 删除从光标所在位置到“行尾”间的字符。
- M-k 删除从光标所在位置到“句尾”间的字符。

也可以只以一种方法来删除缓冲区内的任何部分,先移动到要删除的部分的一端,然后输入 Ctrl+空格组合键,再移到那部分的另一端,接着输入 C-w。这样就会把介于这两个位置间的所有文字删除。

5. 取消动作

如果对文字做了一些改变,后来觉得这些改变是错误的,就可以用 C-x u 命令来取消这一改变。通常 C-x u 会把一个命令所造成的改变取消掉;如果在一行中重复了许多次 C-x u,每一个重复都会取消额外的命令。但是有两个例外:

- (1) 当行中没有发生过改变文本的操作时,例如光标移动命令。
- (2) 自行输入的字符以一群一群(每群最多 20 个)来进行处理。

这是为了减少在取消“输入文字动作”所必须输入 C-x u 的次数。

6. 查找和替换

以下列出的是 Emac 中的查找和替换操作命令:

功能键	功能
C-s	向前搜索;连续按 C-s 键,跳到下一个搜索到的目标。
C-s RET	普通搜索。
C-r	向前搜索。
C-s RET C-w	按单词查询。
M-%	查询替换,也就是替换前会询问一下。
M-x replace-string	普通替换。

除了作为文本编辑器外,Emacs 还集成了万维网浏览器、邮件阅读器、FTP、Telnet、新闻组阅读器、版本控制系统等。Emacs 的集成功能实在太多了,这里不再讨论。

3.3 gcc 编译器的使用

gcc 是 GNU 项目的编译器套件,能够编译用 C、C++ 和 Objective C 等编写的程序,通过使用 gcc,程序员能够对编译过程有更多的控制。编译过程分为四个阶段:预处理、编译、汇编、链接。程序员可以在编译的任何阶段结束后停止整个编译过程,以检查编译器在该阶段的输出信息。gcc 可以在生成的二进制执行文件中加入不同数量和种类的调试码,也能优化代码。gcc 能够在生成调试信息的同时对代码进行优化,但不建议使用该特性,因为优化后的代码中,静态变量可能被取消,循环也可能被展开,优化后的代码与源代码已经不是行行对应了。

gcc 的软件包如表 3-1 所示。

表 3-1 gcc 的软件包

名 称	功 能 描 述
cpp	C 预处理器
g++	C++ 编译器
gcc	C 编译器
gccbug	创建出错(bug)报告的 Shell 脚本
gcov	覆盖测试工具,用来分析在哪处程序进行优化的效果最好
libgcc	gcc 的运行库
libstdc++	标准 C++ 库,包含许多常用的函数
libsupc++	提供支持 C++ 语言的函数库

gcc 有 30 多个警告和 3 个“call-all”警告级。同时,gcc 是一个交叉平台编译器,所以能够在当前 CPU 平台上为不同体系结构的硬件系统开发软件。最后,gcc 对 C 和 C++ 作了大量扩展,这些扩展大部分能够提高程序执行效率,或有助于编译器进行代码优化,或使编程更加容易,但不建议使用,因为这是以降低移植性为代价的。

gcc 的基本用法: `gcc[options][filenames]`,该命令行使 gcc 在给定文件(filenames)上执行编译选项(options)指定的操作。

一个基本实例,如源码清单 3-1:

源码清单 3-1

```
# include <stdio.h>
void main()
{
    printf("Hello world!\n");
}
```

使用 gcc 编译器编译一下:

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

用户将会在同一目录下得到一个名为 a.out 的文件,然后在命令提示符下执行:

```
$ ./a
```

就可以显示结果 Hello world!。要注意的是：a.out 是默认生成的目标文件名，如果在同一个目录下，编译另外一个源程序且没有指明生成的目标文件名，原先的 a.out 文件会被覆盖。我们可以使用-o 选项指定生成的目标文件名，如：

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

这样在同一目录下会生成名为 test.o 的目标文件，然后执行 \$./test 即可，会得到同样的输出（Hello World!）。

3.3.1 gcc 的主要选项

gcc 有一百多个编译选项，并支持多个选项同时使用，只要它们不互相冲突即可。现在介绍经常使用的主要选项，如表 3-2 所示。如果需要了解选项的具体说明和完整列表，可以参考 gcc 联机帮助，即在命令行上输入：man gcc。

表 3-2 gcc 选项列表

选 项	说 明
-werror	把所有警告转换为错误，以在警告发生时中止编译过程
-w	关闭所有警告，不建议使用
-W	允许发出 gcc 能提供的所有有用的警告，也可以用-W{warning} 来标记指定的警告
-V	显示在编译过程中每一步用到的命令
-traditional	支持 Kernighan&Ritchie C 语法
-static	链接静态库
-pedantic-errors	允许发出 ANSI/ISO C 标准所列出的所有错误
-pedantic	允许发出 ANSI/ISO C 标准所列出的所有警告
-O N	指定代码优化级别为 N, 0≤N≤3
-o FILE	指定输出文件名，在编译为目标代码时该项不是必须的，如果 FILE 没有指定，默认文件名是 a.out
-O	优化编译过的代码
-MM	输出一个 make 兼容的相关列表
-l FOO	链接名为 libFOO 的函数库
-L Dirname	将 Dirname 加入到库文件的搜索目录列表中，默认情况下 gcc 只链接共享库
-I Dirname	将 Dirname 加入到头文件搜索目录列表中
-ggdb	在可执行程序中包含只有 GNU debugger 才能识别的大量调试信息
-g	在可执行程序中包含标准调试信息
-DFOO=BAR	在命令行定义预处理宏 FOO，其值为 BAR
-C	只编译不链接
-ansi	支持 ANSI/ISO C 的标准语法，取消 GNU 的语法中与该标准有冲突的部分（但该选项并不能保证生成 ANSI 兼容代码）
-S	只对文件进行预处理，但不编译汇编和链接
-p	产生 prof 所需的信息
-pg	产生 gprof 所使用的信息

现在讨论-o 选项的使用，不论是否生成输出数据，-o FILE 告诉 gcc 把输出定向到 FILE 文件。如果不指定-o 选项，对于名为 FILE.SUFFIX 的输入文件，其生成可执行程序

名为 a.out，目标文件代码是 FILE.o，汇编代码在 FILE.s 中。

下面简单介绍几个选项的使用。

1. 指定函数库和包含文件的查找路径

如果需要链接函数库或不在标准位置下的包含文件，可以使用-L{Dirname} 和-I{Dirname} 选项指定文件所在目录，以确保该目录的搜索顺序在标准目录之前。例如，要编译的程序要包含 paint.h，而该文件位于/usr/include/sam 目录下，不在默认路径下，可以使用如下命令：

```
gcc -Wall -I/usr/include/sam -o paint -c paint.c
```

预处理器就能找到需要的 paint.h 文件。

类似地，如果使用/home/sst/目录下的库文件和头文件编译自己的程序，可以使用如下命令：

```
gcctest.c -L/home/sst/lib -I/home/sst/include -o test -lnew
```

使用该命令编译时，gcc 首先分别在/home/sst/lib 和 /home/sst/include 下寻找所需要的库文件和头文件。该命令中的选项-I 表示要链接的库文件名为 libnew.so。gcc 在默认情况下只链接动态共享库，如果要链接静态库，需要使用-static 选项，如果上例中库文件 libnew 为静态库，则应该使用如下命令：

```
gcc test.c -L/home/sst/lib -I/home/sst/include -o test -lnew -static
```

命令中-static 选项表明要链接的库文件为静态库 libnew.a。链接了静态库的可执行程序比链接动态库时要大很多，不过链接静态库时，程序在任何情况下都可以执行，而链接动态共享库的程序只有在系统中包含了所需的共享库时才可以运行。有的程序如 Netscape 浏览器生成两类可执行程序，Netscape 需要使用付费 Motif，但多数 Linux 用户不能承担在系统中安装 Motif 的费用，所以 Netscape 在用户系统中实际上安装了两个版本的浏览器，一个是链接共享库(netscape -dynMotif)，另一个是链接静态库(netscape -statMotif)，而“可执行”netscape 实际上是一个 shell 脚本。可以检查系统中是否安装 Motif 共享库，以决定需要启动哪一个二进制程序。

2. 出错检查及警告

从上面的选项中可以看到 gcc 的检查出错和生成警告功能。其中，选项-pedantic 表示 gcc 只发出 ANSI/ISO C 标准所列出的所有警告，-pedantic-errors 与-pedantic 相近，但它针对的是错误。选项-ansi 支持 ANSI/ISO C 的标准语法，取消 GNU 的语法中与该标准有冲突的部分，但该选项并不能保证生成 ANSI 兼容代码。下面看一下具体用法，见源码清单 3-2：

源码清单 3-2

```
# include <stdio.h>
void main()
{
    long long i = 1;
    printf("a simple sample");
    return i;
}
```

这段代码中有两处明显缺陷,一是在定义变量 i 时使用了 gcc 扩展语法,二是在返回值时与函数声明中的无返回类型不符。如使用 `gcc -c test2.c -o test` 或 `gcc -ansi test.c -o test` 编译此文件,不会出现编译错误。原因是使用第一个命令时,gcc 略去了返回值的错误,第二个命令中虽有`-ansi`命令,但是 gcc 生成标准语法所要求的诊断信息,并不保证没有警告的程序就遵循 ANSI C 标准。

现在使用`-pedantic`选项。使用`-pedantic`选项来编译会得到警告信息,但编译可以成功;如果使用选项`-pedantic -errors`会得到出错信息,编译不能成功,如下所示:

```
$ gcc -pedantic -errors test.c -o test
test.c: In function 'main':
test.c: 9: warning: ANSI C does not support 'long long'
```

以上实例说明`-ansi`、`-pedantic`、`-pedantic -errors`并不能保证被编译程序的 ANSI/ISO 的兼容性。

3. 优化选项

gcc 的命令行选项中的代码优化选项可以改进程序的执行效率,但代价是需要更长的编译时间和在编译期间需要更多的内存。`-O` 选项与`-O1` 等价,告诉 gcc 对代码长度和执行时间进行优化。在这个级别上执行的优化类型取决于目标处理器,但一般都包括线程跳转和延迟退栈这两种优化。线程跳转优化的目的是减少跳转操作的次数。而延迟退栈是指在嵌套函数调用时推迟退出堆栈的时间,如不做这种优化,每次函数调用完成后都要弹出堆栈中的函数参数,做这种优化后,在栈中保留了参数,直到完成所有的递归调用后才同时弹出栈中积累的函数参数。`-O2` 选项包含`-O1` 级所做的优化,还包括安排处理器指令时序。使用`-O2` 优化时,编译器保证处理器在等待其他指令的结果或来自高速缓存或主存的数据延迟时仍然有可执行的指令,它的实现与处理器密切相关。`-O3` 级优化包含所有`-O2` 级优化,同时还包含循环展开及其他一些与处理器结构有关的优化内容。

依据对目标处理器系列相关信息的多少,还可以使用`-f{<flag>}`选项来指定具体优化类型。常用的类型有 3 种:`-ffastmath`、`-finline-function` 和`-funroll-loops`。`-ffastmath` 对浮点数学运算进行优化以提高速度,但是这种优化违反 IEEE 和 ANSI 标准;`-finline-functions` 选项把所有简单函数像预处理宏一样展开,当然是由预处理器决定什么是简单函数;`-funroll-loops` 选项用于让编译器展开在编译期间就可以确定循环次数的循环。使用以上选项后,因为展开函数和确定性的循环避免了部分变量的查找和函数调用工作,因而理论上可以提高执行速度,但是,上述展开可能会导致目标文件或可执行代码的急剧增长,因而必须通过试验才能决定使用以上选项是否合适。用户可以使用 `time` 命令来查看程序执行时间。优化带来的问题如下:

(1) 优化级别越高,程序执行得越快,但程序编译时间也将越长。这提醒程序员在集中开发时不要使用优化选项,而在版本发行时或者开发即将结束时再优化。

(2) 优化级别越高,程序量越大,特别是`-O3` 级在优化时需要有一定的交换空间,它和程序对 RAM 的要求成正比,在一定程度上,过于庞大的程序会带来负面效果。

(3) 在使用优化时,程序调试会变得困难。因为优化器会删除最终版本中不用的代码,或者重组更多声明,跟踪可执行文件变得困难,所以在调试代码时不要使用优化选项。

在一般情况下,使用`-O2` 优化已经足够。



4. 调试选项

程序员开发程序总免不了错误,所以需要进行程序的调试以排除错误。如果想在编译后的程序中插入调试信息以便于调试,可以使用 gcc 的-g 和-ggdb 选项。调试选项-gN 中,N 可以取 1、2 或 3,N 越大,加入的调试信息就越多。使用级别 1 时仅生成必要的信息以创建回退和堆栈转储,而不包含与局部变量和行号有关的调试信息。选项 2 是默认级别,调试信息会包括扩展的符号表、行号及局部或外部变量信息。3 级选项除包含 2 级的所有信息以外,还包含了宏定义信息。

如果所使用的调试器是 GNU Debugger、gdb,需要使用-ggdb 选项来生成额外的调试信息以方便 gdb 的使用。但这样做也使得程序不能被其他调试器调试。-ggdb 能接受的调试级别和-g 一样,它们对调试输出有同样的影响。但使用任何选项都会使程序大小急剧增长,这样文件中大部分是调试信息,而不是代码,用其他调试器已失去意义。使用标准调试选项(-g)不会使文件大小增加太多,且生成的调试信息已经足够使用。

除了以上两个选项以外,还有-P、-pg、-a 和-save-temps,它们将统计信息添加到二进制文件中,利用这些信息,程序员可以找到性能瓶颈。-p 选项在代码中加入 prof 能够读取的统计信息,而-pg 选项在代码中加入的符号则只能被 GNU 的 prof 所解释。此外,-a 选项在代码中加入记录代码块执行次数的计数器,-save-temps 选项用于保存在编译过程中生成的中间文件,如目标文件和汇编文件。

最后,gcc 虽然允许在优化代码的同时插入调试信息,但是代码优化后,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等,所以最好优化之前彻底调试好程序。在本节中提到的优化只是指编译器所能做的那部分优化,同时注意,不要因为编译器的优化能力而忽略程序设计阶段的优化工作,高效的算法对程序效率的影响比编译器优化的影响大得多。

3.3.2 GNU C 扩展简介

GNU C 在很多方面扩展了 ANSI 标准,如果不介意编写非标准代码,其中一些扩展会很有用。下面只介绍在 Linux 系统头文件和源代码中常见的那些 GNU 扩展。

- (1) gcc 使用 long long 数据类型来提供 64 位存储单元,如 long long long_int_var。
- (2) 内联函数。只要足够短小,内联函数就能像宏一样在代码中展开,从而减少函数调用的开销,同时编译器在编译时对内联函数进行类型检查,所以使用内联函数比宏安全。但是,在编译时至少要使用-o 选项才能够使用内联函数。
- (3) 使用 attribute 关键字指明代码相关信息以方便优化。很多标准库函数没有返回值,如果编译器知道某些函数没有返回值,可以生成较为优化的代码。自定义函数也可以没有返回值,gcc 提供 noreturn 属性来标识这些属性,以提示 gcc 在编译时对其进行优化。如: void exit_on_error(void) __attribute__((noreturn)),但定义和平常一样。

```
void exit_on_error(void)
{
    exit(1);
}
```

也可以对变量指定属性。Aligned 属性指定编译器在为变量分配内存空间时按指定字

节边界对齐。如 `int int_var __attribute__((aligned 16)) = 0;` 使 `int_var` 变量的边界按 16 字节对齐。`packed` 属性使 `gcc` 为变量或结构分配最小空间, 定义结构时, `packed` 属性使得 `gcc` 不会为了对齐不同变量的边界而插入额外的字节。

(4) `gcc` 还对 `case` 做了扩展。在 ANSI C 中, `case` 语句只能对应于单个值的情况, 扩展后 `case` 语句可以对应于一个范围。使用语法是: 在 `case` 关键字后列出范围的两个边界值, 边界值之间使用空格-省略号-空格分开, 即 `case LOWVAL ... HIGHVAL:` 在这里省略号前后的空格是必需的, 如伪代码:

```
switch (i){
    case 0 ... 10:
        /* 事务处理代码 */
        break;
    case 11 ... 100:
        /* 事务处理代码 */
        break;
    default:
        /* 默认情况下的事务处理代码 */
}
```

3.4 GNU make 管理项目

3.4.1 make 简介

当使用 GNU 中的编译语言如 `gcc`、`GNU C++` 编程开发应用时, 绝大多数情况下要使用 `make` 管理项目。为什么要使用 `make` 呢? 首先, 包含多个源文件的项目在编译时都有长而复杂的命令行, 使用 `make` 可以通过把这些命令行保存在 `Makefile` 文件中而简化这个工作; 其次, 使用 `make` 可以减少重新编译所需要的时间, 因为它可以识别出那些被修改的文件, 并且只编译这些文件; 最后, `make` 在一个数据库中维护了当前项目中各文件的相互关系, 从而可以在编译前检查是否可以找到所有需要的文件。

要使用 `make` 进行项目管理必须编写 `Makefile` 文件。`Makefile` 文件是一个文本形式的数据库文件, 其中包含的规则指明 `make` 编译哪些文件及怎样编译这些文件。一条规则包含 3 方面内容: `make` 要创建的目标文件(`target`), 编译目标文件所需的依赖文件列表(`dependencies`), 通过依赖文件创建目标文件所需要执行的命令组(`commands`)。`make` 在执行时按序查找名为 `GNUmakefile`、`makefile` 和 `Makefile` 的文件进行编译, 为保持与 Linux 操作系统源代码开发的一致性, 建议用户使用 `Makefile`。

`Makefile` 规则通用形式如下:

```
target: dependency file1 dependency file2[ ... ]
    command1
    command2
    [ ... ]
```

每一个命令行的首字符必须是 Tab 制表符, 仅使用 8 个空格是不够的。除非特别指定, 否则 `make` 的工作目录就是当前目录。

下面介绍一个简单的 Makefile 实例,见源码清单 3-3:

源码清单 3-3

```
printer : printer.o shape.o op_lib.o
          gcc -o printer printer.o shape.o op_lib.o
printer.o : printer.c printer.h shape.h op_lib.h
          gcc -c printer.c
shape.o : shape.c shape.h
          gcc -c shape.c
op_lib.o : op-lib.c op-lib.h
          gcc -c op-lib.c
clean:
rm printer *.o
```

第一行中的 3 个依赖文件并不存在,如果是在 shell 上用命令行编译则会出错并退出,但在 make 管理项目中,在执行 gcc 时会先检查依赖文件是否存在,若不存在,则会先执行别的规则以生成缺少的依赖文件,最后生成相关的目标文件。如果依赖文件已经存在,则并不急于执行后面的命令重新得到它们,而是比较这些依赖文件及与其对应的源文件的生成时间,如果有一个或多个源文件比相应的依赖文件更新,则重新编译这些文件以反映相关源文件的变化,否则,使用旧的依赖文件生成目标文件。

3.4.2 编写 Makefile 文件的规则

1. 伪目标

在编写 Makefile 文件时,既可以建立普通目标,也可以建立伪目标,伪目标不对应实际文件,如上面的 clean 就是一个伪目标。由于伪目标没有依赖文件,它不会自动执行,原因是: make 在执行到目标 clean 时,make 先检查它的依赖文件是否存在,由于 clean 没有依赖文件,make 就认为该目标 clean 是最新版本,不需要重新创建。要想启动伪目标,必须使用如下命令: make[virtual target],在上例中是执行 make clean。

实际上可以使用特殊的 make 目标.PHONY,目标.PHONY 的依赖文件含义与通常一样,但 make 不会检查是否存在它的依赖文件而直接执行.PHONY 所对应规则的命令,见源码清单 3-4:

源码清单 3-4

```
printer : printer.o shape.o op_lib.o
          gcc -o printer printer.o shape.o op_lib.o
printer.o : printer.c printer.h shape.h op_lib.h
          gcc -c printer.c
shape.o : shape.c shape.h
          gcc -c shape.c
op_lib.o : op-lib.c op-lib.h
          gcc -c op-lib.c
.PHONY : clean
clean:
rm printer *.o
```

2. 变量

编写 Makefile 时也可以使用变量, 所谓变量就是用指定文本串在 Makefile 中定义的一个名字, Makefile 中变量一般用大写, 并用等号给它赋值。引用时只需用括号将变量名括起来并在括号前加上 \$ 符号, 如 VARNAME=some-text。

当编写大型应用程序的 Makefile 时, 其中涉及的依赖文件和规则繁多, 如果使用变量表示某些依赖文件的路径, 则会大大简化 Makefile。一般在 Makefile 文件开始就定义文件中所需的所有变量, 这样使 Makefile 文件清晰且便于修改, 见源码清单 3-5:

源码清单 3-5

```
一个简单的 Makefile 文件
OBJECT = printer.o shape.o op_lib.o
HEADER = printer.h shape.h op_lib.h
printer : $(OBJECT)
        gcc -o printer $(OBJECT)
printer.o : printer.c $(HEADER)
        gcc -c printer.c
shape.o : shape.c shape.h
        gcc -c shape.c
op_lib.o : op_lib.c op_lib.h
        gcc -c op_lib.c
.PHONY : clean
clean:
        rm printer*.o
```

3. make 变量

以上所说的变量都是自定义变量, make 管理项目也允许在 Makefile 中使用 make 变量。make 变量包括环境变量、自动变量和预定义变量。

环境变量是系统环境变量, make 命令执行时会读取系统环境变量并创建与其同名的变量, 但是如果 Makefile 有同名变量, 则用户定义变量会覆盖系统的环境变量值。

make 管理项目允许使用的自动变量全部以符号“\$”开头, 以下是部分自动变量:

- \$@ 扩展为 Makefile 文件中规则的目标文件名。
- \$< 扩展为 Makefile 文件中规则的第一个依赖文件名。
- \$^ 扩展为 Makefile 文件中规则的目标所对应的所有依赖文件的列表, 以空格分隔。
- \$? 扩展为 Makefile 文件中规则的目标所对应的依赖文件中新于目标的文件列表, 以空格分隔。
- \$(@D) 扩展为 Makefile 文件中规则的目标文件的目录部分(如果目标在子目录中)。
- \$(@F) 扩展为 Makefile 文件中规则的目标文件的文件名部分(如果目标在子目录中)。

make 管理项目支持的预定义变量主要用于定义程序名, 以及传给这些程序的参数和标志值。变量含义如下:

- | | |
|----|------------------|
| AR | 归档维护程序, 默认值为 ar。 |
|----|------------------|



AS	汇编程序,默认值为 as。
CC	C 语言编译程序,默认值为 CC。
CPP	C 语言预处理程序,默认值为 cpp。
RM	文件删除程序,默认值为 rm -f。
ARFLAGS	传给归档维护程序的标志,默认值为 rv。
ASFLAGS	传给汇编程序的标志,无默认值。
CFLAGS	传给 C 语言编译程序的标志,无默认值。
CPPFLAGS	传给 C 语言预处理程序的标志,无默认值。
LDFLAGS	传给连接程序的标志,无默认值。

4. 隐式规则

以上介绍的 Makefile 规则都是用户自己定义的,make 还有一系列隐式规则集。如有下面一个 Makefile 文件源码清单 3-6:

源码清单 3-6

```
# a simple Makefile
OBJECT = printer.o shape.o op_lib.o
printer : $(OBJECT)
    gcc -o printer $(OBJECT)
.PHONY : clean
clean:
    rm printer*.o
```

默认目标 printer 的依赖文件是 printer.o shape.o op_lib.o,但在 Makefile 中并没有提及如何生成这些目标的规则,这时,make 使用所谓的隐式规则。实际上,对每一个名为 somefile.o 的目标文件,make 先寻找与之对应的 somefile.c 文件,并用 gcc -c somefile.c -o somefile.o 编译生成这个目标文件。目标文件(.o)可以从 C、Pascal 等源代码中生成,所以 make 符合实际情况的文件。例如,如果在该目录下有 printer.p shape.p op_lib.p(Pascal 源文件),make 就会激活 Pascal 编译器来编译它们。注意:如果在项目中使用多种语言时,不要使用隐式规则,因为使用隐式规则得到的结果可能与预期结果不同。

5. 模式规则

模式规则是指用户自定义的隐式规则。隐式规则和普通规则格式一致,但是目标和依赖文件必须带有百分号符号“%”。该符号可以和任何非空字符串匹配,例如: %.o %.c。实际上 make 已经对一些模式规则进行了定义,如:

```
% .o : % .c
$ (CC) -c $ (CFLAGS) $ (CPPFLAGS) $ < -o $ @
```

此规则表示所有的 Object 文件都由 C 源码生成,使用该规则时,它利用自动变量 \$< 和 \$@ 替代第一个依赖文件和 Object 文件,变量 CC、CFLAGS、CPPFLAGS 使用系统预定值。

6. make 命令

建立 Makefile 文件后,就可以使用 make 命令生成和维护目标文件了。命令格式为:

```
make [options] [macrodef] [target]
```

选项 options 指定 make 的工作行为；选项 macrodef(宏定义)指定执行 Makerfile 时的宏值；目标(target)是要更新的文件列表。这些参数都是可选的，参数之间用空格隔开。

通过在命令行中指定 make 命令的选项(options)，可以使 make 以不同方式运行。现在将一些常用选项列举如表 3-3 所示。

7. 宏

在 make 中使用宏，首先要定义宏，在 makefile 中引用宏的定义格式为：

宏名 赋值符号 宏值

宏名由用户指定，可以使用字母、数字、下画线(_)的任意组合，不过不能以数字开头，习惯上一般使用大写字母，并使名字有意义，便于阅读和维护。

赋值符号有如下 3 种：

- (1) = 直接将后面的字符串赋给宏。
- (2) := 后面跟字符串常量，将它的内容赋给宏。
- (3) += 宏原来的值加上一个空格，再加上后面的字符串，作为新的宏值。

一般常用的是第一种。除了空格，赋值符号的前面不能有制表符或其他任何分隔符号，否则都会被 make 当作宏名的一部分，从而引起语法错误。

宏的引用有如下两种格式：

`$ (宏名)` 或 `$ {宏名}`

当宏名只有单个字符时，可以省略括号，如 `$ A` 就等于 `$ (A)`。由于 make 将美元符号 \$ 作为宏引用的开始，因此要表示 \$ 符号需要用两个 \$(即 \$ \$)。

make 处理时会先扫描一遍整个 makefile，确定所有宏的值。因此，注意宏的引用可以在定义之后，而且使用的是最后一次赋予的值。例如源码清单 3-7：

源码清单 3-7

```
All : print1 print2
var1 = hello
print1 : ; @echo $ (var1)
var1 += world
print2 : ; @echo $ (var1)
```

则两次打印的都是 hello world。

表 3-3 给出了 make 的选项列表。

表 3-3 make 的选项列表

选项	说 明
-C dir	make 开始运行之后的工作目录为指定目录。如果有多个-C，后面的 dir 指定的是相对于前一个的目录，如-C/-C etc 等价于-c/etc
-d	打印除一般处理信息之外的调试信息，例如进行比较的文件的时间、真正被重新构建的文件等
-e	不允许在 makefile 中对环境的宏赋新值
-f file	使用指定的文件为 makefile
-i	忽略运行 makefile 文件时命令行产生的错误，不退出 make

续表

选项	说 明
-I dir	指定搜索被包含的 makefile 的目录。如果命令行中有多个-I 选项,按出现的顺序依次搜索。与 make 的其他选项不同,允许 dir 紧跟在-I 之后(一般的选项和参数之间一定要加空格),这是为了与 c 预处理器兼容
-k	执行命令出错时放弃当前的目标文件,尽可能地维护其他目标
-n	按实际运行时的执行顺序显示命令,包括以@开头的命令,但不真正执行
-o file	不维护指定文件,即使它比其依赖文件更旧。如果认为某个文件太陈旧了,没有必要再加以维护,可以使用该选项忽略该文件以及与之有关的规则
-p	显示 makefile 中所有宏定义和描述内部规则的规则,然后按一般情况执行。如果只想打印这些信息而不真正进行维护,可以使用 make - p - f /dev/null
-q	“问题模式”。如果指定的目标目前没有过期,就返回 0,否则返回一个非零值。不运行任何命令或打印任何信息
-r	忽略内部规则,同时清除默认的后缀规则
-s	执行但不显示执行的命令
-S	执行 makefile 命令菜单时出错即退出 make。这是 make 的默认工作方式,所以一般不必指定
-t	修改每个目标文件的创建日期,但不真正重新创建文件
-v	打印 make 的版本号,然后正常执行。如果希望只打印信息而不真正维护,使用 make -v - f /dev/null

宏还允许嵌套使用,处理时依次展开,例如:

```
HEADFILE = myfile.h
HEADFILE1 = myfile2.h
INDEX = 1
```

现在引用 \$(HEADFILE \$(INDEX)),首先展开宏 INDEX,得到 \$(HEADFILE1),最后的结果是 myfile2.h。

宏的定义可以出现在三个地方。一是在 makefile 中定义,二是在 mde 命令行中指明,三是载入环境中的宏定义。在 makefile 中定义只需直接书写上面的定义式,也可以用前面介绍过的伪目标. include 从其他文件中获得宏定义。在 make 命令行中定义时,应放在“属性”之后,“目标文件”之前。

3.5 GDB 调试

3.5.1 GDB 命令介绍

Linux 包含了一个 GNU 调试程序 GDB,即 GNU Debugger。GDB 主要做 4 件事,包括为了完成这些事而附加的功能,帮助找出程序中的错误:

- (1) 运行程序,设置所有能影响程序运行的选项。
 - (2) 保证程序在指定的条件下停止。
 - (3) 当程序停止时,进行检查。
 - (4) 改变程序。可以试着修正某个缺陷引起的问题,然后继续查找另一个缺陷。
- 当然,可以用 GDB 来调试用 C 和 C++ 写的程序。在 shell 命令行输入 gdb 然后按

Enter 键启动 GDB 调试器,如果正常启动,则在屏幕上会看到:

```
GDB is free software and you are welcome to 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
GDB 4.14(i486 - slakware - Linux), Copyright 1995 Free Software Foundation,
Inc.
(gdb)
```

也可以在 shell 命令上输入 `gdb filename`,告诉 GDB 装入名为 `filename` 的可执行文件,并对其调试,但在使用该命令前,要求源代码在编译时使用`-g` 选项,用以生成增强的符号表。也可以用 GDB 去检查一个因程序异常终止而产生的 core 文件,或与一个正在运行的程序相连。可以通过参考页或者在命令行上输入 `gdb -h`,得到一个有关 GDB 选项的简单列表,或在 `gdb` 下输入 `help` 来查看如何使用 GDB。

载入程序后,接下来就是要进行断点的设置,以及要监视的变量的添加等工作,下面对在这个过程中常会用到的命令逐一进行介绍。

(1) `list`: 显示程序中的代码,常用的格式有:

<code>list</code>	输出从上次调用 <code>list</code> 命令开始往后的 10 行程序代码
<code>list -</code>	输出从上次调用 <code>list</code> 命令开始往前的 10 行程序代码
<code>list n</code>	输出第 <code>n</code> 行附近的 10 行程序代码
<code>list function</code>	输出函数 <code>function</code> 前后的 10 行程序代码

(2) `forward/search`: 从当前行向后查找匹配某个字符串的程序行。使用格式:

`forward/search` 字符串

查找到的行号将保存在 `$_` 变量中,可以用 `print $_` 命令来查看。

(3) `reverse-search`: 和 `forward/search` 相反,向前查找字符串。使用格式同上。

(4) `break`: 在程序中设置断点,当程序运行到指定行上时,会暂停执行。使用格式为:

`break` 要设置断点的行号

(5) `tbreak`: 设置临时断点,在设置之后只起一次作用。使用格式为:

`tbreak` 要设置临时断点的行号

(6) `clear`: 和 `break` 相反,`clear` 用于清除断点。使用格式为:

`clear` 要清除的断点所在的行号

(7) `run`: 启动程序,在 `run` 后面带上参数可以传递给正在调试的程序。

(8) `awatch`: 用来增加一个观察点(`add watch`),使用格式为:

`awatch` 变量或表达式

当表达式的值发生改变或表达式的值被读取时,程序就会停止运行。

(9) `watch`: 与 `awatch` 类似,用来设置观察点,但程序只有当表达式的值发生改变时才会停止运行。使用格式为:

`watch` 变量或表达式

需要注意的是,awatch 和 watch 都必须在程序运行的过程中设置观察点,即在运行 run 之后才能设置。

(10) commands: 设置在遇到断点后执行特定的指令。使用格式有:

commands 设置遇到最后一个断点时要执行的命令

commands n 设置遇到断点号 n 时要执行的命令

注意,commands 后面跟的是断点号,而不是断点所在的行号。

在输入命令后,就可以输入遇到断点后要执行的命令,每行一条命令,输入最后一条命令后再输入 end 就可以结束输入。

(11) delete: 清除断点或自动显示的表达式。使用格式为:

delete 断点号

(12) disable: 让指定断点失效。使用格式为:

disable 断点号列表

断点号之间用空格间隔开。

(13) enable: 和 disable 相反,恢复失效的断点。使用格式为:

enable 断点编号列表

(14) ignore: 忽略断点。使用格式为:

ignore 断点号忽略次数

(15) condition: 设置断点在一定条件下才能生效。使用格式为:

condition 断点号条件表达式

(16) cont/continue: 使程序在暂停在断点之后继续运行。使用格式为:

cont 跳过当前断点继续运行

cont n 跳过 n 次断点,继续运行

当 n 为 1 时,cont 1 即为 cont。

(17) jump: 让程序跳到指定行开始调试。使用格式为:

jump 行号

(18) next: 继续执行语句,但是跳过子程序的调用。使用格式为:

next 执行一条语句

next n 执行 n 条语句

(19) nexti: 单步执行语句,和 next 不同的是,它会跟踪到子程序的内部,但不打印出子程序内部的语句,使用格式同上。

(20) step: 与 next 类似,它会跟踪到子程序的内部,而且会显示子程序内部的执行情况。使用格式同上。

(21) stepi: 与 step 类似,但是比 step 更详细,是 nexti 和 step 的结合。使用格式同上。

(22) whatis: 显示某个变量或表达式的数据类型。使用格式为:

`whatis` 变量或表达式

(23) `ptype`: 和 `whatis` 类似, 用于显示数据类型, 它还可以显示 `typedef` 定义的类型等。

使用格式为:

`ptype` 变量或表达式

(24) `set`: 设置程序中变量的值。使用格式为:

`set` 变量 = 表达式

`set` 变量: = 表达式

(25) `display`: 增加要显示值的表达式。使用格式为:

`display` 表达式

(26) `info display`: 显示当前所有的要显示值的表达式。

(27) `delete display/undisplay`: 删除要显示值的表达式。使用格式为:

`delete display/undisplay` 表达式编号

(28) `disable display`: 暂时不显示一个表达式的值。使用格式为:

`disable display` 表达式编号

(29) `enable display`: 与 `disable display` 相反, 使用表达式恢复显示。使用格式为:

`enable display` 表达式编号

(30) `print`: 打印变量或表达式的值。使用格式为:

`print` 变量或表达式

表达式中有两个符号有特殊含义: `$` 和 `$$`。`$` 表示给定序号的前一个序号, `$$` 表示给定序号的前两个序号。

如果 `$` 和 `$$` 后面不带数字, 则给定序号为当前序号。

(31) `backtrace`: 打印指定个数的栈帧(stack frame)。使用格式为:

`backtrace` 栈帧个数

(32) `frame`: 打印栈帧。使用格式:

`frame` 栈帧号

(33) `info frame`: 显示当前栈帧的详细信息。

(34) `select -frame`: 选择栈帧, 选择后可以用 `info frame` 来显示栈帧信息。使用格式:

`select -frame` 栈帧号

(35) `kill`: 结束当前程序的调试。

(36) `quit`: 退出 GDB。

如要查看所有的 GDB 命令, 可以在 GDB 下输入两次 Tab(制表符), 运行“`help command`”可以查看命令 `command` 的详细使用格式。

3.5.2 GDB 调试例程

下面用一个实例介绍怎样用 GDB 调试程序。该程序称为 greeting, 它显示一个简单的问候, 再用反序将它打印出来, 见源码清单 3-8:

源码清单 3-8

```
# include < stdio.h >
main()
{
    char my_string[] = "hello there";
    my_print(my_string);
    my_print2(my_string);
}
void my_print(char * string)
{
    printf("The string is %s\n",string);
}
void my_print2(char * string)
{
    char * string2;
    int size,i;
    size = strlen(string);
    string2 = (char *)malloc(size+1);
    for(i=0;i<size;i++)
        string2[size-i] = string[i];
    string2[size+1] = '\0';
    printf("The string Iprinted backward is %s\n",string2);
}
```

用 gcc 编译它:

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

程序执行时显示如下结果:

```
The string is hello there
The string printed backward is
```

输出第一行是正确的, 但第二行打印出的字符并不是所期望的。期望的输出应该是:

```
The string printed backward is olleh ereht
```

毫无疑问, my_print2 函数没有正常工作。现在, 用 GDB 看看问题究竟出在哪儿, 先输入如下命令:

```
gdb greeting
```

如果输入命令时忘了把要调试的程序作为参数传给 GDB, 可以在 GDB 提示符下用 file 命令加载它:

```
(gdb) file greeting
```

这个命令加载 greeting 可执行文件,就像在 GDB 命令行里加载它一样。

现在可以用 GDB 的 run 命令来运行 greeting 了。当它运行在 GDB 中时,结果会像这样:

```
(gdb)run
starting program:/root/greeting
The string is hello there
The string printed backward is
Program exited with code 041
```

这个输出和在 GDB 外面运行的结果一样。可是,为什么反序打印没有工作呢?为了找出问题所在,可以在 my_print2 函数的 for 语句后设一个断点。具体的做法是在 gdb 提示符下执行三次 list 命令,列出源代码:

每次执行 list 命令会列出 10 行代码。

第一次执行 list 命令的输出如下:

```
1 # include<stdio.h>
2
3 main()
4 {
5     char my_string[ ] = "hello there";
6
7     my_print(my_string);
8     my_print2(my_string);
9 }
10
```

如果按下 Enter 键,GDB 将再执行一次 list 命令,输出下列代码:

```
11 my_print(char * string)
12 {
13     printf("The string is % s\n", string);
14 }
15
16 my_print2(char' string)
17 {
18     char * string2;
19     int size, i;
20 }
```

再按一次 Enter 键将列出 greeting 程序的剩余部分:

```
21     size = strlen(string);
22     string2 = (char *)malloc(size + 1);
23     for(i = 0;i < size;i++)
24         string2[size - i] = string[i];
25     string2[size + 1] = '\0';
26     printf ("The string printed backward is % s\n", string2);
27 }
```

根据列出的源程序,可以看到应该将断点设在第 24 行,在 GDB 命令行提示符下输入如

下设置断点：

```
(gdb) break 24
```

gdb 将做出如下响应：

```
Breakpoint 1 at 0x139: file greeting.c, line 24
(gdb)
```

现在再执行 run 命令,将产生如下的输出：

```
Starting program:/root/greeting
The string is hello there
Breakpoint 1, my_print2(string = 0xbffffdc4"hello there")at greeting.c, 24
24  string2[size - i] = string[i]
```

可以通过设置一个观察 string2[size-i]变量值的观察点来找出错误的产生原因,做法如下语句：

```
(gdb) watch string2[size - i]
```

GDB 将做出如下响应：

```
Watchpoint 2: string2[size - i]
```

现在可以用 next 命令来一步步地执行 for 循环了：

```
(gdb)next
```

经过第一次循环后,GDB 告诉我们 string2[size-i]的值是'h'。GDB 显示如下信息：

```
Watchpoint 2, string2[size - i]
Old value = 0 '\000'.
New value = 104'h'
my_print2(string = 0xbffffdc4"hello there")at greeting.c: 23
23  for(i = 0;i < size;i++)
```

这个值正是期望的。后来的数次循环的结果也都是正确的。当 i=10 时,表达式 string2[size-i]的值等于'e',size-i 的值等于 1,最后一个字符已经复制到新的字符串中了。

如果再把循环执行下去,会看到已经没有值分配给 string2[0]了,而它是新字符串的第一个字符,因为 malloc 数在分配内存时把它们初始化为空(null)字符。所以 string2 的第一个字符是空字符。这解释了为什么在打印 string2 时没有任何输出。

找出了问题的所在,修正这个错误也就会变得很容易。可以把代码里写入 string2 的第一个字符的偏移量改为 size-1 而不是 size。这是因为 string2 的大小为 12,但起始偏移量是 0,串内的字符从偏移量 0 偏移到 10,偏移量 11 为空字符保留。

为了使代码正常工作,有很多种修改办法。一种是另设一个变量,它比字符串的实际大小要小 1。

下面是这种解决办法的代码,见源码清单 3-9:

源码清单 3-9

```
# include <stdio.h>
```

```

main()
{
    char my_string[ ] = "hello there";
    my_print(my_string);
    my_print2(my_string);
}
my_print(char string)
{
    printf("The string is %s\n", string);
}
my_print2(char * string)
{
    char * string2;
    int size, size2, i;
    size = strlen(string);
    size2 = size - 1;
    string2 = (char *) malloc(size + 1);
    for(i = 0; i < size; i++)
        string2[size2 - i] = string[i];
    string2[size] = '\0';
    printf("The string printed backward is %s\n", string2);
}

```

3.5.3 基于 GDB 的图形界面调试工具

1. xxgdb

xxgdb 是早期的 GDB 前端,用法与 GDB 类似,最后的版本号是 1.12。xxgdb 是 GDB 的一个基于 X Windows 系统的图形界面。xxgdb 包括了命令行版的 GDB 上的所有特性。xxgdb 使你能通过按钮来执行常用的命令。设置了断点的地方也用图形来显示。

可以在一个 Xterm 窗口里输入下面的命令来运行它:

```
xxgdb
```

可以用 GDB 里任何有效的命令行选项来初始化 xxgdb。

2. DDD

GNU DDD 是命令行调试程序,如 GDB、DBX、WDB、Ladebug、JDB、XDB、Perl Debugger 或 Python Debugger 的可视化图形前端。它特有的图形数据显示功能(graphical datadisplay)可以把数据结构按照图形的方式显示出来。DDD 最初源于 1990 年 Andreas Zeller 编写的 VSL 结构化语言,后来经过一些程序员的努力,演化成今天的模样。DDD 的功能非常强大,可以调试用 C/C++、Ada、FORTRAN、Pascal、Modula-2 和 Modula-3 编写的程序;可以超文本方式浏览源代码;能够进行断点设置、回溯调试和历史记录编辑;具有程序在终端运行的仿真窗口,并在远程主机上进行调试的能力。图形数据显示功能是创建该调试器的初衷之一,能够显示各种数据结构之间的关系,并将数据结构以图形化形式显示;它具有 GDB/DBX/XDB 的命令行界面,包括完全的文本编辑、历史记录、搜寻引擎。

3. kdbg

kdbg 是基于 KDE 的调试工具,是采用 KDE GUI 的 GDB 前端,对 KDE/QT 等 C++ 程



序支持得较好,与集成开发环境 kdevelop 集成得比较好。使用习惯类似于 Visual C++ 等现代调试工具。

3.6 本章小结

本章详细介绍了 Linux 的编程环境,包括:集成开发平台 Eclipse+CDT、编辑工具 VIM 和 Emacs、gcc 的编辑工具、GNU make 管理工具以及 GDB 调试器,这些是今后进行嵌入式 Linux 系统开发的具体基础。

3.7 思考题

- (1) GNU 的 gcc 的软件包包括哪几大部分?
- (2) 如何使用 vi 编辑器进行查找、替换、复制和粘贴等操作?
- (3) 给下列 makefile 源代码作注释:

```
1 INCLUDES = -I /home/nie/mysrc/include \
2   -I /home/nie/mysrc/extern/include \
3   -I /home/nie/mysrc/src \
4   -I /home/nie/mysrc/libsrc \
5   -I. \
6   -I..
7 EXT_CC_OPTS = -DEXT_MODE
8 CPP_REQ_DEFINES = -DMODEL=tunel -DRT -DNUMST=2 \
9   -DTID01EQ=1 -DNCSTATES=0 \
10  -DMT=0 -DHAVESTDIO
11 RTM_CC_OPTS = -DUSE_RTMODEL
12 CFLAGS = -O -g
13 CFLAGS += $(CPP_REQ_DEFINES)
14 CFLAGS += $(EXT_CC_OPTS)
15 CFLAGS += $(RTM_CC_OPTS)
16 SRCS = tunel.c rt_sim.c rt_nonfinite.c grt_main.c rt_logging.c \
17   ext_svr.c updown.c ext_svr_transport.c ext_work.c
18 OBJS = $(SRCS:.c=.o)
19 RM = rm -f
20 CC = gcc
21 LD = gcc
22 all: tunel
23 %.o : %.c
24   $(CC) -c -o $@ $(CFLAGS) $(INCLUDES) $<
25 tunel : $(OBJS)
26   $(LD) -o $@ $(OBJS) -lm
27 clean :
28   $(RM) $(OBJS)
```

- (4) 请在 Linux 平台下对本章介绍的 greeting 例程进行编译,并按书中的调试流程对原来的错误代码进行调试,以熟悉 GDB 的调试方法。