# 《嵌入式 Linux 系统开发标准教程》

作者：华清远见



## 第 3 章 Linux 编程环境

本章目标

本章内容包括常用的 Linux 开发工具使用技巧和 Linux 编程技术。本章内容比 Linux 编程方面的书籍简略得多，重点介绍常用的 Linux 编程工具和技巧。通过本章学习可以使读者快速掌握基本的 Linux 开发工具，为后续的嵌入式 Linux 开发打下基础。

- 常用 Linux 编程工具
- GNU 工具链的使用技巧
- Linux 编程库的 API 介绍

# 3.1 Linux 常用工具

## 3.1.1 Shell 简介

在 Linux 系统开发过程中，开发者或者用户与 Linux 系统（内核）进行交互的时候需要一个平台，这就是 Shell，有了它，用户就能通过键盘输入与系统进行交互了。Shell 会执行用户输入的命令，并且在屏幕上显示执行结果。这种交互的全过程都是基于文本方式的，这种面向命令行的用户界面被称为 CLI（Command Line Interface），在图形化用户界面（GUI）出现之前，人们一直是通过命令行界面来操作计算机的。Linux 的图形化环境最近这几年有很大改进，在 X 窗口系统下，只需打开 Shell 提示来完成极少量的任务。然而，许多 Linux 功能在 Shell 提示下要比在图形化用户界面（GUI）下完成得更加高效，况且一些应用程序并不支持图形界面。

单从字面意思上理解，Shell 的本意是“壳”的意思，通俗地讲就是内部核心与外部使用者发生联系的介质。当用户希望与系统内核（Kernel）发生联系进而控制硬件设备时，用户不会也不允许直接与内核交互，而必须通过 Shell 来下达命令使系统来控制硬件，同时内核也会通过 Shell 来反馈执行情况，这里的 Shell 就是一个桥梁。图 3.1 形象地说明了这一过程。

Shell 提供了用户与操作系统之间通信的方式。这种通信可以以交互方式（从键盘输入，并且可以立即得到响应）或者以 Shell script（非交互）方式执行。Shell script 是放在文件中的一串 Shell 和操作系统命令，它们可以被重复使用。本质上，Shell script 是把命令行的命令简单地组合到一个文件中。

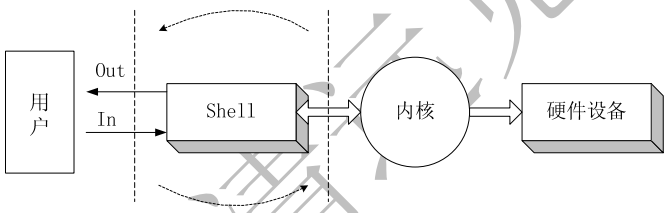


图 3.1 Shell 工作示意图

Shell 本身又是一个解释型的程序，也是一种编程语言，Shell 程序设计语言支持绝大多数在高级语言中能见到的程序元素，如函数、变量、数组和程序控制结构。Shell 编程语言简单而且易于掌握，任何在提示符中能键入的命令都能放到一个可执行的 Shell 程序中。作为操作系统的外壳，如果把 Linux 内核想像成一个系统的中心部分，那么 Shell 就是围绕内核的外层。当从 Shell 或其他程序向 Linux 传递命令时，内核会做出相应的反应。

历史上第一个真正的 Unix shell 称为“sh”，是 Stephen R. Bourne 于 20 世纪 70 年代中期在新泽西的 AT&T 贝尔实验室编写的，为了纪念他，亦称为“Bourne shell”，Bourne Shell 是一个交换式的命令解释器和命令编程语言。在 20 世纪 80 年代早期，在美国 Berkeley 的加利福尼亚大学开发了 C shell（csh 和 tcsh），它主要是为了让用户更容易地使用交互式功能。C shell 是一种比 Bourne Shell 更适于编程的 shell，它的语法与 C 语言很相似。

Bash（Bourne Again Shell）是目前大多数 Linux（Red Hat，Slackware 等）系统默认使用的 Shell，它由 Brian Fox 和 Chet Ramey 共同完成，内部命令一共有 40 个，它是 Bourne Shell 的扩展，与 Bourne Shell 完全向后兼容，并且在 Bourne Shell 的基础上增加了很多特性。Bash 是 GNU 计划的一部分，用来替代 Bourne Shell。Linux 使用它作为默认的 Shell 是因为它有以下的特点。

- 可以使用类似 DOS 下面的 doskey 的功能，用上下方向键查阅和快速输入并修改命令。
- 自动通过查找匹配的方式，给出以某字串开头的命令。
- 包含了自身的帮助功能，只要在提示符下面键入 help 就可以得到相关的帮助。

Linux 下使用 Shell 非常简单，打开终端就可以看到 Shell 的提示符了，登录系统之后，系统将执行一个称为 Shell 的程序，正是 Shell 进程提供了命令行提示符。作为 Linux 默认的 Bash，对于普通用户用“$”作为 Shell 提示符，而对于根用户（root）用“#”作提示符。如图 3.2 所示。

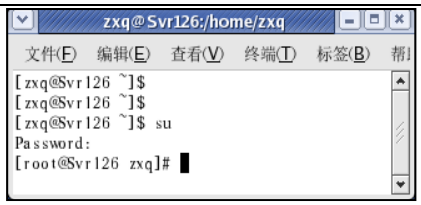


图 3.2 shell 提示符

从上面的界面中可以看到，当前用户是普通用户“zxq”时，Shell 提示符是‘$’；而当切换为根用户 root 时，Shell 提示符是‘#’。一旦出现了 Shell 提示符，就可以键入命令名称及命令所需要的参数了。用户键入有关命令行后，如果 Shell 找不到以其中的命令名为名字的程序，就会给出错误信息。例如，如果用户键入：

```
$mypfile
bash:myfile:command not found
$
```

可以看到，用户得到了一个没有找到该命令的错误信息。

## 3.1.2 常用 Shell 命令

目前，Linux 下基于图形界面的工具越来越多，许多工作都不必使用 Shell 就可以完成了。然而，专业的 Linux 使用者还是认为 Shell 是一个非常必要的工具，使用 Linux 时一定要熟悉 Sheel 的使用，至少要掌握一些基础知识和基本的命令。

由于 Bash 是 Linux 上缺省的 Shell，本章主要介绍 Bash 及其相关知识，Shell 命令可以分为两种。

- 包含于 Shell 内部的命令，如 cd 命令。
- 存在于系统文件内部的某个应用程序，如 ls 命令。

对用户使用 Shell 来说，不必关心一个命令是建立在 Shell 内部还是一个单独的程序。在实际执行的时候，Shell 会首先检查输入的命令是否是 Shell 的内部命令，如果不是，再检查是否是一个内部的应用程序。然后 Shell 在搜索路径里寻找这些应用程序（搜索路径就是一个能找到可执行程序的目录列表）。如果键入的命令不是一个内部命令并且在路径里没有找到这个可执行文件，将会显示一条错误信息。如果能够成功找到命令，该内部命令或应用程序将被分解为系统调用并传给 Linux 内核。

Shell 命令的一般格式如下。

命令名 【选项】 【参数 1】 【参数 2】…

用户登录时，实际就进入了 Shell，它遵循一定的语法将输入的命令加以解释并传给系统。命令行中输入的第一个部分必须是一个命令的名字，第二个部分是命令的选项或参数，命令行中的每个部分必须由空格或 Tab 键隔开，注意，这里的选项和参数都用【】标注，这是说明它们都是可选的，因为有的命令不需要选项和参数就可以执行。

### 1．对于选项和参数的说明

【选项】是包括一个或多个字母的代码，它前面有一个减号（–），Linux 用它来区别选项和参数，【选项】可用于改变命令执行的动作的类型。多个【选项】可以用一个减号（–）连起来，例如‘ls–l–a’ 与‘ls–la’相同。

以常用的 ls 命令为例，ls 命令可以查看当前目录的内容，加入选项–l 可以以长格式查看当前目录内容，如图 3.3 所示.。

加入–l 选项，将会为每个文件列出一行信息，诸如数据大小和数据最后被修改的时间。

使用该指令可以查看文件的权限位，如上图中的“–rw–r– –r– –”符号，它表示的是 3 组不同用户对该文件的使用权限，每组有 3 个权限位，如下所示。

- rw- 用户权限
- r-- 同组用户权限
- r-- 其他用户权限

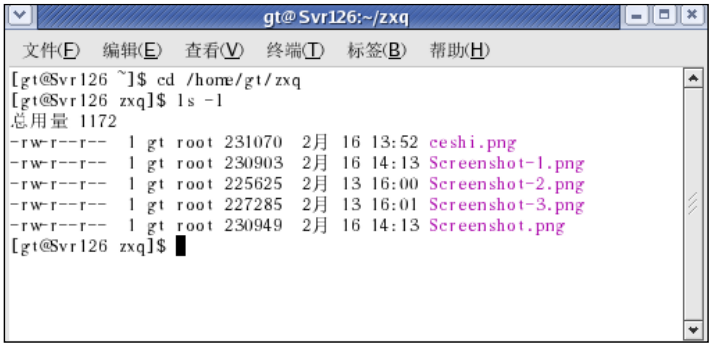


图 3.3 ls 命令

【参数】提供命令运行的信息，或者是命令执行过程中所使用的文件名。使用分号（;）可以将两个命令隔开，这样可以实现一行中输入多个命令。命令的执行顺序和输入的顺序相同。当然，ls 命令也可以加入参数，例如“ls -l /home/zxq”命令会将/home/zxq 目录的内容详细地列出。

## 2．命令行输入

命令行输入实际上是可以编辑的一个文本缓冲区，在命令行中就可以输入 Shell 命令了。在按“回车键”以确认当前操作之前，可以对输入的内容进行编辑。比如删除、复制、粘贴等，还可以插入字符，使得用户在输入命令，尤其是复杂命令时，若出现键入错误，无须重新输入整个命令，只要利用编辑操作，即可改正错误。

Bash 可以保存以前键入命令的列表，这一列表被称为命令历史表。按向上箭头键，便可以在命令行上逐次显示各条命令。同样，按向下箭头键可以在命令列表中向下移动，这样可以将以前的各条命令显示在命令行上，用户可以修改并执行这些命令，这样可以不用重复输入以前执行的命令。

## 3．常用 Shell 命令介绍

Shell 命令种类很多，功能也很复杂，下面就几种常用的 Shell 命令来介绍。

（1）输入命令行自动补齐（automatic command line completion）功能。

在 Linux 下有的文件名或文件夹的名称可能会很长，完全逐字输入比较麻烦，在输入命令的任何时刻可以按<Tab>键，当这样做时，系统将试图补齐此时已输入的命令。例如，假设当前目录下有一文件 Busybox-pre-1.00.tar.gz，现在想要解压该文件，而该文件是当前目录下惟一以 B 开头的文件名，此时就可以如下操作。

```
# tar zxvf  B<Tab>usy-pre-1.00.tar.gz
```

此时，系统会自动补齐该文件名后面的部分，这样用起来就会非常方便。

使用命令行自动补齐功能，对于使用长命令或操作较长名字的文件或文件夹都是非常有意义的。

（2）对目录和文件的操作。

- 改变当前目录。

```
# cd [目的目录名]
```

这里的目的目录名可用相对路径表示，也可以用绝对路径表示。如果要切换到上一级目录，可以采用下面的命令。

```
# cd ..
```

- 显示当前所在目录。

Linux 下 pwd 命令是最常用的命令之一，用于显示用户当前所在的目录。例如：

```
# pwd
/home/TH
```

执行 pwd 指令后，系统提示当前所在的目录是/home/TH。

- 创建目录。

在 Linux 下可以使用 mkdir 指令来创建一个目录。

```
# mkdir [新目录名]
```

例如：mkdir /home/TH，该命令的功能就是在/home/目录下创建 TH 子目录。

- 删除一个目录/文件。

rm [选项] 被删除的文件/目录

对于选项的说明如下。

–r：完全删除目录，其下的目录和文件也一并删除。

–i：在删除目录之前需要经过使用者的确认才能被删除。

–f：不需要确认就可以删除，也不会产生任何错误信息。

例如：rm –rf /home/TH/tmp，就是不必经过确认就把/home/TH/tmp/下的目录和文件全部删除。

- 复制文件/目录。

```
# cp [ 选项] [源文件/目录] [指定文件/目录]
```

对于选项的说明如下。

-i:采用-i 选项时，当指定目录下已存在被复制的文件时，会在复制之前要求确认是否要覆盖，如使用者的回答是 y（yes）才执行复制的动作。

-p：保留权限模式和更改时间。

-r：此参数是用来将一目录下的所有文件都复制到另一个指定目录中。

例如：cp /etc/ld.conf ～/，复制/etc/目录下的 ld.conf 文件到系统的主目录中；

cp -r dir1 dir2，将目录 dir1 的全部内容全部复制到目录 dir2 里面。

- 建立文件的符号链接。

建立文件的符号链接是 Linux 中一个很重要的命令，它的基本功能是为某一个文件在另外一个位置建立一个不同的链接，这个命令最常用的选项是-s，具体用法如下。

```
# ln [-s] [源文件] [目标文件]
```

在实际的操作过程当中，有时在不同的目录中要用到相同的文件，我们不需要在每一个需要的目录下都放一个相同的文件，而是使用 ln 命令链接（link）它就可以（相当于建立了一个快捷方式），这样可以避免重复地占用磁盘空间。例如：ln –s /bin/test /usr/local/bin/test，这就为/bin 下的 test 文件在/usr/local/bin 目录下建立了一个符号链接。

**注意** ln 命令会保持每一处链接文件的同步性，也即是说如果改动了某一文件，其他的符号链接文件都会发生相应的变化；其次，ln 命令的链接方式又有软链接和硬链接两种，上面提到的用法就是软链接，它只会在你选定的位置上生成一个文件的镜像，不会占用磁盘空间，硬链接没有选项-s，它会在指定的位置上生成一个和源文件大小相同的文件，无论是软链接还是硬链接，文件都保持同步变化。

- 改变文件/目录访问权限。

在 Linux 系统下面，一个文件有可读（r）、可写（w）、可执行（x）3 种模式，chmod 可以用数字来表示该文件的使用权限，其语法如下。

```
# chmod [XYZ] 文件
```

其中 X、Y、Z 各为一个数字，分别表示 User（用户）、Group（同组用户）及 Other（其他用户）对于该文件的使用权限。对于文件的属性，r（可读）=4，w（可写）=2，x（可执行）=1。对于每一位用户来说，若要具有 rwx 属性则对应的位应为 4+2+1=7，若要 rw-属性则为 4+2=6，若要 r-x 属性则为 4+1=5。比如下面的例子：

```
# chmod 751 /home/TH/test
```

其执行结果就是使程序 test 对于用户可读、写、执行，对于同组用户可读、执行，对于其他用户可执行。Chmod 还有一种用法就是使用包含字母和操作符表达式的字符设定法（相对权限设定），通过参数-r、-w、-x 来设定权限，这里不再详细地介绍。

- 改变文件/目录的所有权。

```
chown [-R] 用户名 文件/目录
```

例如

```
# chown TH File1
```

将当前目录下的文件 File 改为用户 TH 所有。

```
# chown -R TH Dir1
```

将当前目录 Dir1 改为用户 TH 所有。

（3）用户管理。

- 添加/删除用户。

# adduser user1，由具有 root 权限的用户添加用户 user1

# userdel user2，由具有 root 权限的用户删除用户 user2

- 设置用户口令。

为了更好地保护用户账号的安全，Linux 允许用户随时修改自己的口令。修改口令的命令是 passwd，它将提示用户输入旧口令和新口令，之后还要求用户再次确认新口令，以避免用户无意中按错键。

（4）文件的打包和压缩。

先来看一下 Linux 下打包命令。Linux 下最常用的打包程序就是 tar（tape archive-磁带存档），使用 tar 程序打出来的包都是以.tar 结尾的。tar 命令可以为文件和目录创建档案(备份文件)，也可以在档案中改变文件，或者向档案中加入新的文件。使用 tar 命令，可以把一大堆的文件和目录全部打包成一个文件，这对于备份文件或将几个文件组合成为一个文件以便于传输是非常有用的。其语法如下。

```
# tar [选项]f targetfile.tar 文件/目录
```

**注意** 选项后面的 f 是必须的，通常用来指定包的文件名。

选项说明如下。

c：创建新的档案文件。如果用户想备份一个目录或是一些文件，就要选择这个选项。例如：

```
# tar -cf test.tar /home/tmp          ---
```

将/home/tmp 目录下的文件打包为 test.tar。

r：增加文件到已有的包，如果发现还有一个目录或是一些文件忘记备份了，这时可以使用该选项，将还需要的目录或文件添加到包文件中，例如：

```
# tar -rf test.tar *.jpg
```

该命令将所有的 jpg 文件添加到 test.tar 包里面去。-r 是表示增加文件的意思。

t：列出包文件的所有内容，查看已经备份了哪些文件。例如：

```
# tar -tf test.tar
```

x：从 tar 包文件中恢复所有文件，事实上是一个解包的过程。例如：

```
# tar  -xf  test.tar
```

k：保存已经存在的文件。例如把某个文件还原，在还原的过程中，遇到相同的文件，不会进行覆盖。
w：每一步都要求确认。
tar 命令还有一个非常重要的用法，这就是 tar 可以在打包或解包的同时调用其他的压缩程序（如 gzip、bzip2）来压缩文件。

**注意** 打包和压缩是两个不同的概念。

Linux 下的压缩文件主要有以下几种格式。
.Z-compress 程序的压缩格式；
.bz2-bzip2 程序的压缩格式；
.gz-gzip 程序的压缩格式；
.tar.gz-由 tar 程序打包，并且经过 gzip 程序的压缩，是 Linux 下常见的压缩文件格式；
.tar.bz2-由 tar 程序打包，并且经过 bzip2 程序的压缩。
以下就几种常用的情况进行说明。

- 调用 gzip 程序来压缩文件。

gzip 是 GNU 组织开发的一个压缩程序，gzip 压缩文件的后缀是.gz，与 gzip 相对的解压程序是 gunzip。tar 中使用-z 这个参数来调用 gzip。下面举例说明。

```
# tar -czf  test.tar.gz  *.jpg
```

这条命令是将当前目录下的所有.jpg 文件打成一个 tar 包，并且将其用 gzip 程序压缩，生成一个 gzip 压缩过的包，压缩包名为 test.tar.gz，解开该压缩包的用法如下。

```
# tar -xzf  test.tar.gz
```

- 调用 bzip2 程序来压缩文件。

bzip2 是 Linux 下的一个压缩能力更强的压缩程序，bzip2 压缩文件的后缀是.bz2，与 bzip2 相对应的解压程序是 bunzip2。tar 中使用-j 这个参数来调用 gzip 压缩程序。例如：

```
# tar  -cjf  test.tar.bz2  *.jpg
```

该命令是将当前目录下所有.jpg 的文件打成一个 tar 包，并且将其用 bzip2 程序压缩，生成一个 bzip2 压缩过的包，压缩包名为 test.tar.bz2，解开该压缩包的用法如下。

```
# tar  -xjf  test.tar.bz2
```

（5）rpm 软件包的安装。
在使用任何操作系统的过程中，安装和卸载软件是必须的操作。Linux 中有一套软件包管理器，最初由 Red Hat 公司推出，称为 rpm（Red Hat Package Manager），它可以用来安装、查询、校验、删除、更新 rpm 格式的软件包。rpm 软件包包含可执行的二进制程序和该程序运行时所需要的文件，rpm 格式的软件包文件使用.rpm 为扩展名。与直接从源代码安装相比，软件包管理易于安装、更新和卸载软件，也易于保护配置文和跟踪已安装文件。
安装 rpm 软件包的主要格式如下。

```
#rpm -i[options] software.rpm
```

rpm 命令主要有以下参数。

- i：安装 rpm 软件包。
- t：测试安装。
- h：安装时输出 hash 记号（"#"），可以显示安装进度。

- f：忽略安装过程中的任何错误。
- U：升级安装 rpm 软件包。
- e：卸载已安装的软件包。
- V：检测软件包件是否正确安装。

以安装 develop-devel-0.9.2-2.4.5.i386.rpm 软件包为例，图 3.4 显示了它的安装过程。

如图 3.4 所示，系统提示的‘#’号就表示软件安装进度，当后面的的百分比数字为 100%时表示软件安装完成。

（6）源码维护基本命令。

diff 命令。

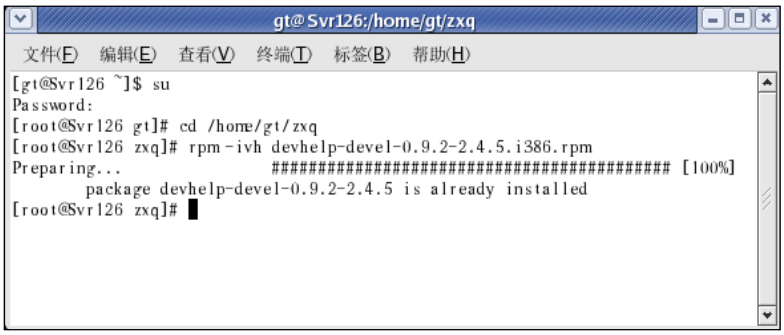


图 3.4　rpm 软件包安装示例

diff 命令是生成源代码补丁的必备工具，其命令格式如下。

```
diff  [命令行选项]  源文件  新文件
```

diff 命令常用选项如下。

- -r：递归处理相应目录。
- -N：包含新文件到 patch。
- -u：输出统一格式（unified format），这种格式比缺省格式更紧凑些。
- -a：可以包含二进制文件到 patch。

通常可以使用 diff 命令加参数-ruN 来比较两个文件并生成一个补丁文件。这个补丁文件会列出这两个不同版本文件的差异。比如有两个文本文件：text1 和 text2，两者内容不尽相同，现在来创建补丁文件。

```
[root@localhost]# diff -ruN test1.txt  test2.txt > test.patch
```

这样就创建好了补丁文件 test.patch，补丁创建好以后需要给相应文件/程序打好补丁，这里就要用到 patch 命令。

```
patch  [命令行选项]  [patch 文件 ]
```

patch 的详细使用方法可参见 patch 的 man help，常用的命令行选项是-p*n*（*n* 是自然数），例如采用下面的指令来打好补丁。

```
[root@localhost]patch -p1 <test.patch
```

-p1 选项代表 patch 文件名左边目录的层数，考虑到顶层目录在不同的系统上可能有所不同。要使用这个选项，就要把 patch 文件放在要被打补丁的目录下，然后在这个目录中运行 path -p1 < [patchfile]命令。

（7）配置、编译、安装源码包软件。

所谓源码包软件就是源代码的可见的软件包，在 Linux 系统下也经常需要用到源码包软件。

大多数的源码软件包是以 tar.gz 或 tar.bz2 的形式得到的，所以在配置和编译之前需要将软件包解压缩，具体的做法已经在前面提到过。配置、编译、安装的过程大多如下所示。

```
#./configure
# make
# make install
```

./configure 用来配置软件的功能，./configure 比较重要的一个参数是--prefix，通过使用--prefix 参数，可以指定软件的安装目录；比如可以指定软件安装到/homet/tmp 目录中，可以执行如下的指令。

```
#./configure --prefix=/home/tmp
# make
# make install
```

（8）中断 Shell 命令执行的方法。

在 Linux 系统下，一旦出现了 Shell 提示符，就可以键入命令名称及命令所需要的参数。Shell 将执行这些命令。如果在执行过程当中想终止命令执行，可以从键盘上按 Ctrl+C 发出中断信号来中断它。

（9）模块管理指令。

Linux 内核采用模块化管理方式，这是 Linux 内核的一大特点，这也使得 Linux 整体结构非常灵活，编于精简。

- insmod（添加模块）指令。

Linux 有许多功能是通过模块的方式，在需要时才载入 kernel。如此可使 kernel 较为精简，进而提高效率，以及保有较大的弹性。这些可动态加载的模块，通常是系统的设备驱动程序。加载模块采用 insmod 指令，其常用语法如下。

```
insmod [-fkmpsvxX] [-o<模块名称>] [模块文件]
```

其中的参数解释如下。

-f：不检查目前 kernel 版本与模块编译时的 kernel 版本是否一致，强制将模块载入。

-k：将模块设置为自动卸载。

-m：输出模块的载入信息。

-p：测试模块是否能正确地载入 kernel。

-s：将所有信息记录在系统记录文件中。

-v：执行时显示详细的信息。

-x：要汇出模块的外部符号。

-X：汇出模块所有的外部符号，此为预设置。

- rmmod（卸载模块）指令。

Linux 把系统的许多功能编译成一个个单独的模块，待有需要时再分别加载它们，如果不再需要这些模块的时候，就可以使用 rmmod 命令来卸载这些模块。其语法如下。

rmmod [-as] [模块名称…]

其使用参数说明如下。

-a：删除所有目前不需要的模块。

-s：把信息输出至 syslog 常驻服务，而非终端机界面。

## 3.1.3 编写 Shell 脚本

在 Linux 系统中，虽然有各种各样的图形化接口工具，但是 Shell 仍然是一个非常灵活的工具。Shell 不仅仅是命令的执行，而且是一种编程语言，它提供了定义变量和参数的手段以及丰富的程序控制结构。由于 Shell 特别擅长系统管理任务，尤其适合那些易用性、可维护性和便携性比效率更重要的任务，所以用户可以通过使用 Shell 使大量的任务自动化，就像使用 DOS 操作系统的过程当中，会执行一些重复性的命令。因此常将这些大量的重复性命令写成批处理命令，通过执行这个批处理命令来代替执行重复性的命令。在 Linux 系统中也有类似的批处理命令，被称作是 Shell 脚本（Script）。前面已经提到 Shell 也是一种解释性的语言，而解释执性的语言的与编译型语言（如 C 语言）的最大不同就在于它们编写起来很方便、快捷，可以说，使用 Shell 脚本来完成一些特定的常用的任务是一个不错的选择。

### 1．建立脚本

编辑 Shell 脚本文件使用 Linux 下的普通编辑器如 vi、Emacs 等即可。Linux 下的 Shell 默认采用 Bash，所以本书也主要以 Bash 脚本为例介绍，在建立 Shell 脚本程序的开始首先应指明使用哪种 Shell 来解释所写的脚本，一般来说 Bash 脚本以"#!"开头（文件的首行），而"#!"后面同时要将所使用 Shell 的路径明确指出，比如 Bourne Shell 的路径为/bin/sh，而 C Shell 的路径则为/bin/csh，Linux 下默认采用 Bash，所以本书也主要以 Bash 脚本为例介绍，下面的语句就是指定 Bash 来解释脚本。

```
#! /bin/sh
```

该语句说明该脚本文件是一个 Bash 程序，需要由/bin 目录下的 Bash 程序来解释执行。除了在脚本内指定所使用的 Shell 类型以外，使用过程中也可以在命令行中强制指定。比如想用 C Shell 执行某个脚本，就可以使用以下命令。

```
# csh  Myscript
```

为了增加程序的可读性，Shell 脚本语句也可以像高级语言那样加注释，在 Bash 脚本程序中从“#”号开始到行尾的部分均被看作是程序的注释语句。

## 2．Shell 变量

Shell 编程中可以使用变量，这充分体现了它的灵活性。对 Shell 来讲，所有变量的取值都是一个字串。Shell 脚本中主要有以下几种变量：系统变量、环境变量、用户变量。其中用户变量在编程过程中使用频繁；系统变量在对参数判断和命令返回值判断会使用；环境变量主要是在程序运行的时候需要设置。此外，Shell 脚本的执行并不需要编译，所以也就不需用检查脚本中变量的类型，因此在 Shell 脚本中使用变量不必像高级语言那样事先对变量进行定义。

- Shell 系统变量。

以下是一些常用到的 Shell 系统变量及其含义。

$ #：保存程序命令行参数的数目。

$ ?：保存前一个命令的返回值。

**注意** 在 Linux 中，命令退出状态为 0 表示该命令正确执行，任何非 0 值表示命令出错。

$ 0：当前程序名。

$ *：以（"$1 $2…"）的形式保存所有输入的命令行参数。

$ @：以（"$1""$2"…）的形式保存所有输入的命令行参数。

$ n：$1 为命令行的第一个参数，$1 为命令行的第二个参数，依次类推。

举一个针对以上系统变量使用的例子，使用 vi 编辑一个脚本文件，文件名为 Example Script，其内容如下。

```
#! /bin/sh
# Script name: Example Script
echo "The No. of parameter is:  $#";
echo "The script name is: $0";
echo "The parameters in the script are: $*";
```

在命令行中执行该脚本程序：

```
# ./Example Script  Hello Linux
```

命令行中的 Hello Linux 是其参数，该程序执行结果如下。

```
The No. of parameter is: 2
The script name is: ./ Example Script
The parameters in the script are: Hello Linux
```

- Shell 环境变量。

Shell 环境变量是所有 Shell 程序都会接受的参数。Shell 程序运行时，都会接收一组变量，这组变量就是环境变量，常用的 Shell 环境变量如下。

PATH：决定了 Shell 将到哪些目录中寻找命令或程序。

HOME：当前用户主目录的完全路径名。

HISTSIZE：历史记录数。

LOGNAME：当前用户的登录名。

HOSTNAME：指主机的名称。

SHELL：Shell 路径名。

LANGUGE：语言相关的环境变量，多语言可以修改此环境变量。

MAIL：当前用户的邮件存放目录。

PS1：主提示符，对于 root 用户是#，对于普通用户是$。

PS2：辅助提示符，默认是“>”。

TERM：终端的类型。

PWD：当前工作目录的绝对路径名。

- Shell 用户变量。

Shell 用户变量是最常使用的变量，可以使用任何不包含空格字符的字串来当做变量名称，在 Linux 支持的所有 Shell 中，都可以用赋值符号（=）为变量赋值，在使用 Shell 用户变量的时候，通常是按照下面的语法规则来定义用户变量。

变量名=变量值

例如：

```
A=9
B="Hello World"
```

**注意** 在定义变量时，变量名前不应加符号$，等号两边一定不能留空格。

变量的引用，要在变量前加$，例如：

```
S="string"
echo $S
```

下面举一个非常简单的例子来说明。

```
#! /bin/bash
# This is a example
SR="Hello World"
echo $STR
```

上面的例子很简单，定义了一个变量 SR，并且赋值给 SR，然后在终端输出 SR 的值。

## 3．流程控制

同传统的编程语言一样，Shell 提供了很多特性，如数据变量、参数传递、判断、流程控制、数据输入和输出、子程序及以中断处理等。

（1）条件语句。

同其他高级语言程序一样，复杂的 Shell 程序中经常使用到分支和循环控制结构，主要有两种不同形式的条件语句：if 语句和 case 语句。

- if 语句。

if 语句的语法格式如下。

```
if  [expression]
then
commands1        // expression 为 True 时的动作
else
commands2            // expression 为 False 时的动作
fi
.
.
```

- case 语句。

case 语句的语法格式如下。

```
case 字符串 in
 模式 1） command;;
 模式 2） command;;
……
esac
```

case 语句是多分支语句，它按“）”左边的模式对字符串值的匹配来执行相应的命令，匹配总是由上而下地进行，总是执行首先匹配到的模式对应的命令表，如果模式中的每个都匹配不到，则什么也不执行，所以一般会在最后放一个*），代表以上都不匹配的任意字符串。";;"表示该模式对应的命令部分程序。

（2）循环语句。

- while 循环语句。

在 while 循环语句中，当某一条件为真时，执行指定的命令。语句的结构如下。

```
while expression
do
command
……
done
```

- for 循环语句。

for 循环语句对一个变量的可能的值都执行一个命令序列。赋给变量的几个数值既可以在程序内以数值列表的形式提供，也可以在程序以外以位置参数的形式提供。for 循环语句的一般格式如下。

```
for    变量名 [in 列表]
do
   command1
   command2
  ……
done
```

## 4．Shell 脚本的执行

Shell 脚本是以文本方式存储的，而非二进制文件。所以 Shell 脚本必须在 Linux 系统的 Shell 下解释执行。如果已经写好 Shell 脚本，运行该脚本可以有以下的几种方法。

（1）设置好脚本的执行权限之后再执行脚本。

可以使用下列方式设置脚本的执行权限。

- chmod u+x Scriptname 只有自己可以执行，其他人不能执行；
- chmod ug+x Scriptname 只有自己以及同一群可以执行，其他人不能执行；
- chmod +x Scriptname 所有人都可以执行。

设置好执行权限之后就可以执行脚本程序了，例如，编辑好一个脚本程序 MyScript 之后，可按下面的方式来执行。

```
[localhost@zxq]# chmod  +x  MyScript
[localhost@zxq]# ./Myscript
```

（2）使用 Bash 内部指令 source。
例如，下面的执行过程：

```
[localhost@zxq]# source  Myscript
```

（3）直接使用 sh 命令来执行
例如：

```
[localhost@zxq]# sh  Myscript
```

**注意** 后面的两种情况不必设置权限即可执行。

## 3.1.4 正则表达式

正则表达式源于人类神经系统如何工作的早期研究。在 19 世纪 60 年代，一位叫 Stephen Kleene 的数学家发表了一篇标题为“神经网事件的表示法”的论文，正式引入了正则表达式的概念。正则表达式就是用来描述他称为“正则集的代数”的表达式，因此采用“正则表达式”这个术语，此后，正则表达式的第一个实用应用程序就是 UNIX 中的 qed 编辑器。
在 Shell 编程中经常会用到正则表达式（regular expression），简单地讲，正则表达式是一种可以用于模式匹配和替换的有效工具。正则表达式描述了一种字符串匹配的模式，可以用来检查一个串是否含有某种子串、将匹配的子串做替换或者从某个串中取出符合某个条件的子串等。使用 Shell 时，从一个文件中抽取多于一个字符串有时会很不方便，而使用正则表达式可以方便、快捷地解决这一问题。
正则表达式由普通字符（例如字符 a～z）以及特殊字符（称为特殊字符）组成特定文字模式。当从一个文件或命令中抽取或者过滤文本时，使用正则表达式可以简化命令中的匹配表达。Linux 系统自带的所有文本过滤工具在某种模式下都支持正则表达式，正则表达式可以匹配行首与行尾、数据集、字母和数字以及一定范围内的字符串集合，在进行匹配时，正则表达式有一组基本特殊字符，其基本的特殊字符及其含义如表 3.1 所示。

**表 3.1　　正则表达式特殊字符及其含义**

| 特 殊 字 符 | 代 表 含 义 |
|---|---|
| ^ | 只匹配行首 |
| $ | 只匹配行尾 |
| * | 单字符后跟*将匹配 0 个或者多个此字符 |
| [ ] | 匹配[ ]内的字符，可以是单个字符也可以是字符序列 |
| \ | 转义字符，用来屏蔽一个字符的特殊含义 |
| . | 用来匹配任意的单字符 |
| Pattern\{\n} | 用来匹配 pattern 在前面出现的次数，n 即为次数 |
| Pattern\{n,\} | 用来匹配前面 pattern 出现的次数，次数最少为 n |
| Pattern\{n,m\} | 用来匹配前面 pattern 出现的次数，次数在 n 和 m 之间 |

下面举几个简单的例子来说明。

### 1．行首和行尾的匹配

在 Bash 中使用正则表达式时，可以使用^和$来分别匹配行首和行尾的字符或字符串，比如下面的正则表达式。

```
^....abc..
```

该表达式的含义是在每行开始任意匹配 4 个字符，之后必须是字符 abc，行尾匹配任意的 3 个字符，那么该表达式与下面各个字符串的匹配结果如下。

```
Gyftabc12345          不匹配（行尾不匹配）
7853abcpoi            匹配
85fabc0k8             不匹配（行首不匹配）
```

## 2．[ ]和指定次数的匹配

括号[ ]用来匹配特定字符串和字符串集合，可以用逗号将要匹配的不同字符串分开，用“-”符号表示匹配字符串的范围，例如，想要匹配任意的字母和数字，可以使用下面的正则表达式。

```
[A-Z，a-z，0-9]
```

*号可以匹配单字符 0 次或多次，例如下面的字符串都可以与表达式 Des*k 匹配。

```
Desk
Dessk
Dessskl
```

使用*可匹配所有匹配结果任意次，如果要指定匹配的次数，就应使用\{\}用法，使用有 3 种模式。

- pattern\{n\} 匹配模式出现 n 次。
- pattern\{n,\} 匹配模式出现至少 n 次。
- pattern\{n, m} 匹配模式出现次数在 n 到 m 次之间，n、m 为 0～255 中的任意整数。

例如表达式 G\{2\}H、G\{2,\}H、G\{2,3}的匹配结果分别如下。

```
GGH
GG（...,多个G）H
GGH,GGGH
```

## 3．使用反斜杠\来屏蔽一个特殊字符的含义

有时在进行文本过滤或抽取的时候，所要匹配的字符本身就是特殊字符，但并没有特殊的含义，为了将二者区分开来，就需要用到反斜杠来转义该字符（也称转义符）。比如要匹配包含“*”的字符串，而“*”是一个特殊字符，因此需要屏蔽它的特殊含义，就可以如下操作。

*

这样的表示方式就认为*是一个特殊的字符，再比如要匹配包含“^”的语句，可以如下表示。

\^

反斜杠\将^的特殊含义屏蔽，在这里只是代表一个普通字符^。

构造正则表达式的方法和创建数学表达式的方法一样，采用多种元字符与操作符将一些基本的表达式组合成为功能更复杂的正则表达式，其组成元素可以是单个的字符、字符集、字符或数字的范围、字符间的选择或者所有这些元素的任意组合。表 3.2 是常用到的一些正则表达式。

**表 3.2　　常用正则表达式特及其含义**

| 表　达　式 | 代 表 含 义 |
| --- | --- |
| ^ | 只匹配行首 |

| | |
|---|---|
| $ | 只匹配行尾 |
| ^[ STR] | 匹配以 STR 作为行的开头 |
| [Ss]igna[lL] | 匹配单词 s i g n a l、Signal、s i g n a L、SignaL |
| ^USER$ | 匹配只包含 USER 的行 |
| ^d..x..x..x | 匹配对用户、用户组和其他用户组成员都有可执行权限的目录 |
| [.*0] | 用来匹配 0 之前或之后加任意字符 |
| [^$] | 用来匹配空行 |
| [^.*$] | 用来匹配行中任意字符串 |
| [a-z][a-z]* | 至少一个小写字母 |
| [^0-0A-Za-z] | 匹配非数字或字母（大小写均可） |
| [i I] [n N] | 匹配大写或小写的 i/n |
| \. | 匹配带句点的行 |
| [ 0 0 0 * ] | 匹配 0 0 0 或更多个 0 |
| ^.* | 匹配只有一个字符的行 |

### 3.1.5 Linux 程序编辑器

编辑器是系统的重要工具之一，在各种操作系统中，编辑器都是必不可少的部件。Linux 系统提供了一个完整的编辑器家族系列，如 Ed、Ex、Vi 和 Emacs 等，按功能它们可以分为两大类。

- 行编辑器（如 Ed、Ex）。
- 全屏幕编辑器（如 Vi、Emacs）。

行编辑器每次只能对一行进行操作，使用起来不是很方便。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，修改的结果可以立即看出来，克服了行编辑方式存在的一些缺点，便于用户学习和使用。Vi（Visual Interface）和 Emacs（Editing with MACroS）是 Linux 下主要的两个编辑器，下面的内容主要就 Vi 的使用做详细的介绍。

Vi 编辑器最初是由 Sun Microsystems 公司的 Bill Joy 在 1976 年开发的。一开始 Bill 开发了 Ex 编辑器，后来开发了 Vi 作为 Ex 的 visual interface，也就是说 Vi 允许一次能看到一屏的文本而非一行，Vi 也因此得名。随之技术的不断进步，基于 Vi 的各种变种版本不断出现，其中，移植特性最好，使用最广泛的当属 Vim 编辑器，相比早期的 Vi，Vim 编辑器增加的一项最重要的功能便是多级撤销，Vi 只支持一级撤销。

目前，Vi/Vim 已经是 Linux 下用的最普遍的文本处理器之一，Vi 也是 Linux 下的第一个全屏幕交互式编辑程序，使用非常普遍，Vi 没有菜单，只有命令，且命令繁多，但是一旦掌握了 Vi 的用法，就可以体会到它的强大功能。它可以执行输出、删除、查找、替换、块操作等众多文本操作，而且用户可以根据自己的需要对其进行定制，这是其他编辑程序所没有的。在终端下输入 Vim 命令就可以看到 Vi 的界面了，如图 3.5 所示。


图 3.5 Vi 界面

Vi 有 3 种基本工作模式：指令行模式、文本输入模式、末行模式，它们的相互关系如图 3.6 所示。

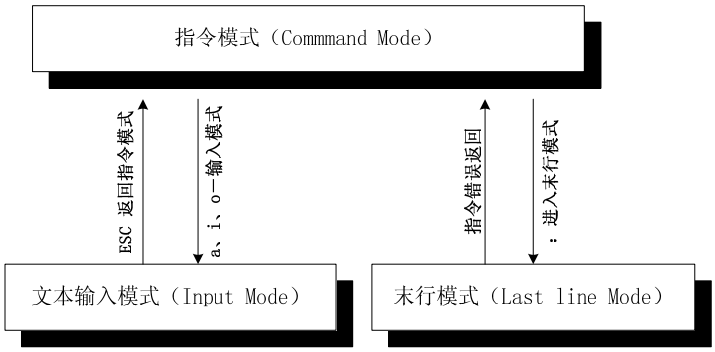


图 3.6　Vi 的模式切换关系

下面分别来介绍这 3 种模式。

## 1．指令模式（command mode）

指令模式主要使用方向键移动光标位置以进行文字的编辑，在输入模式下按【Esc】键或是在末行模式输入了错误命令，都会回到指令模式，表 3.3 列出了其常用操作命令及含义。

**表 3.3　　vi 指令模式命令及其含义**

| 操 作 命 令 | 实 现 功 能 |
|---|---|
| 0 | 光标移至行首 |
| h | 光标左移一格 |
| l | 光标右移一格 |
| j | 光标向下移一行 |
| k | 光标向上移一行 |
| $ + A | 将光标移到该行最后 |
| PageDn | 向下滚动一页 |
| PageUp | 向上滚动一页 |
| d+方向键 | 删除文字 |
| dd | 删除整行 |
| pp | 整行复制 |
| r | 修改光标所在字符 |
| S | 删除光标所在的列，并进入输入模式 |

## 2．文本输入模式

在 vim 下编辑文字，不能直接插入、替代或删除文字，而必须先进入输入模式。要进入输入模式，可以在指令模式下按【a/A】键、【i/I】键或【o/O】键，它们的命令及其含义如表 3.4 所示。

**表 3.4　　文本输入模式命令及其含义**

| 操 作 命 令 | 实 现 功 能 |
|---|---|
| a | 在光标后开始插入 |
| A | 在行尾开始插入 |

| | |
|---|---|
| i | 从光标所在位置前面开始插入 |
| I | 从光标所在列的第一个非空白字元前面开始插入 |
| o | 在光标所在列下新增一列并进入输入模式 |
| O | 在光标所在列上方新增一列并进入输入模式 |
| Esc | 返回命令行模式 |

**注意** 结束文本输入模式必须用【Esc】键。

## 3．末行模式

末行模式主要用来进行一些文字编辑辅助功能，比如字串搜寻、替代、保存文件等，表 3.5 介绍一些常用的命令。

表 3.5 末行模式命令及其含义

| 操 作 命 令 | 实 现 功 能 |
|---|---|
| ：q | 结束 Vi 程序，如果文件有过修改，先保存文件 |
| ：q! | 强制退出 Vi 程序 |
| ：wq | 保存修改并退出程序 |
| ：set nu | 设置行号 |

大多数时候，可用命令如：Vi filename 来打开文件 filename，Vim 以编辑或打开某个文件。下面以编辑一个简单脚本程序为例介绍 Vi 的简单使用方法，其主要流程如下。

- 在终端输入命令用 Vi 建立文件（可以是文本文件、C\C++程序等）。

```
# vi  Script_edit
```

输入该命令之后就进入了 Vi 的编辑界面，如图 3.7 所示。

此时的 Vi 是指令模式，输入“：set nu”来设置行号，此时属于末行模式，末行模式不能直接切换到文本输入模式，需要先切换到指令模式，按【Esc】键进入指令模式。

- 输入“i”进入输入模式。

在指令模式下输入“i”进入文本输入模式，并编辑文本内容，如图 3.8 所示。

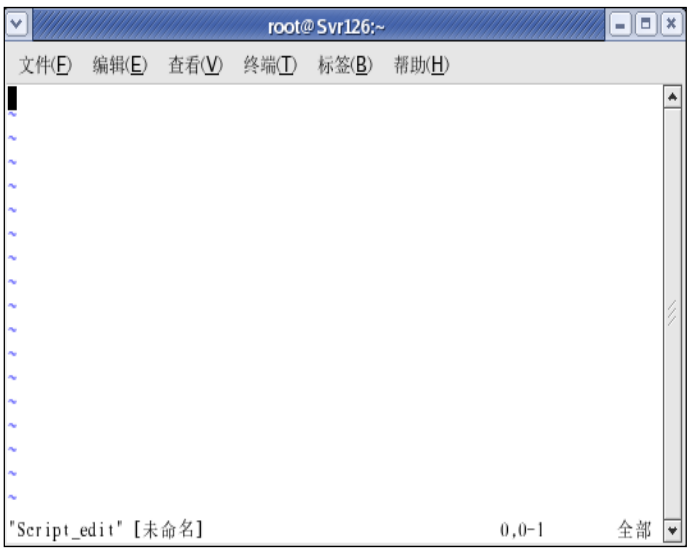


图 3.7 Vi 编辑界面


图 3.8 Vi 文本输入模式界面

- 保存、修改编辑内容并退出 Vi 程序。

在输入模式下编辑并修改相应内容，编辑好之后需要再返回到指令模式（Esc），之后输入“：wq”就可以保存并且退出刚才的编辑程序了。

## 3.2 Makefile 简介

### 3.2.1 GNU make

GNU make 最初是 UNIX 系统下的一个工具，设计之初是为了维护 C 程序文件不必要的重新编译，它是一个自动生成和维护目标程序的工具。在使用 GNU 的编译工具进行开发时，经常要用到 GNU make 工具。使用 make 工具，我们可以将大型的开发项目分解成为多个更易于管理的模块，对于一个包括几百个源文件的应用程序，使用 make 和 Makefile 工具就可以高效地处理各个源文件之间复杂的相互关系，进而取代了复杂的命令行操作，也大大提高了应用程序的开发效率，可以想到的是如果一个工程具有上百个源文件时，但是采用命令行逐个编译那将是多么大的工作量。

使用 make 工具管理具有多个源文件的工程，其优势是显而易见的，举一个简单的例子，如果多个源文件中的某个文件被修改，而有其他多个源文件依赖该文件，采用手工编译的方法需要对所有与该文件有关的源文件进行重新编译，这显然是一件费时费力的事情，而如果采用 make 工具则可以避免这种繁杂的重复编译工作，大大地提高了工作效率。

make 是一个解释 Makefile 文件中指令的命令工具，其最基本的功能就是通过 Makefile 文件来描述源程序之间的相互关系并自动维护编译工作，它会告知系统以何种方式编译和链接程序。一旦正确完成 Makefile 文件，剩下的工作就只是在 Linux 终端下输入 make 这样的一个命令，就可以自动完成所有编译任务，并且生成目标程序。通常状况之下 GNU make 的工作流程如下。

① 查找当前目录下的 Makefile 文件。
② 初始化文件中的变量。
③ 分析 Makefile 中的所有规则。
④ 为所有的目标文件创建依赖关系。
⑤ 根据依赖关系，决定哪些目标文件要重新生成。
⑥ 执行生成命令。

为了比较形象地说明 make 工具的工作原理，举一个简单的例子来介绍。假定一个项目中有以下一些文件。

- 源程序：Main.c、test1.c、test.c。
- 包含的头文件：head1.h、head2.h、head3.h。
- 由源程序和头文件编译生成的目标文件：Main.o、test1.o、test2.o。
- 由目标文件链接生成的可执行文件：test。

这些不同组成部分的相互依赖关系如图 3.9 所示。

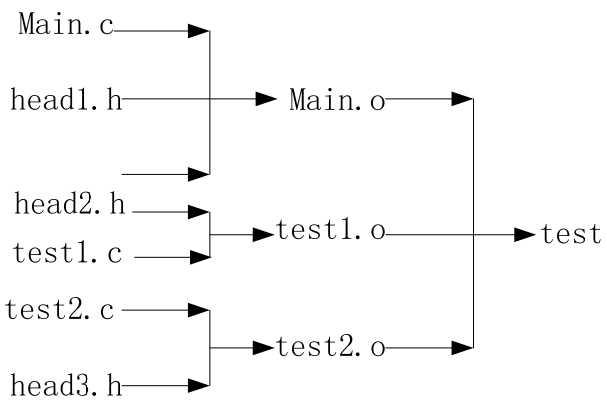


图 3.9 依赖关系

在该项目的所有文件当中，目标文件 Main.o 的依赖文件是 Main.c、head1.h、head2.h；test1.o 的依赖文件是 head2.h、test1.c；目标文件 test2.o 的依赖文件是 head3.h、test2.c；最终的可执行文件的依赖文件是 Main.o、test1.o 和 test2.o。执行 make 命令时，会首先处理 test 程序的所有依赖文件（.o 文件）的更新规则，对于.o 文件，会检查每个依赖程序（.c 和.h 文件）是否有更新，判断有无更新的依据主要看依赖文件的建立时间是否比所生成的目标文件要晚，如果是，那么会按规则重新编译生成相应的目标文件，接下来对于最终的可执行程序，同样会检查其依赖文件（.o 文件）是否有更新，如果有任何一个目标文件要比最终可执行的目标程序新，则重新链接生成新的可执行程序，所以，make 工具管理项目的过程是从最底层开始的，是一个逆序遍历的过程。从以上的说明就能够比较容易理解使用 make 工具的优势了，事实上，任何一个源文件的改变都会导致重新编译、链接生成可执行程序，使用者不必关心哪个程序改变、或者依赖哪个文件，make 工具会自动完成程序的重新编译和链接工作。

执行 make 命令时，只需在 Makefile 文件所在目录输入 make 指令即可，事实上，make 命本身可带有这样的一些参数：【选项】、【宏定义】、【目标文件】。其标准形式如下。

Make [选项] [宏定义] [目标文件]

Make 命令的一些常用选项及其含义如下。

- -f file：指定 Makefile 的文件名。
- -n：打印出所有执行命令，但事实上并不执行这些命令。
- -s：在执行时不打印命令名。
- -w：如果在 make 执行时要改变目录，则打印当前的执行目录。
- -d：打印调试信息。
- -I<dirname>：指定所用 Makefile 所在的目录。
- -h：help 文档，显示 Makefile 的 help 信息。

举例来讲，在使用 make 工具的时候，习惯把 makefile 文件命名为 Makefile，当然也可以采用其他的名字来命名 makefile 文件，如果要使用其他文件作为 Makefile，则可利用带-f 选项的 make 命令来指定 Makefile 文件。

```
# make –f  Makefilename
```

参数【目标文件】对于 make 命令来说也是一个可选项，如果在执行 make 命令时带有该参数，可以输入如下的命令。

```
# make  target
```

target 是用户 Makefile 文件中定义的目标文件之一，如果省略参数 target，make 就将生成 Makefile 文件中定义的第一个目标文件。因此，常见的用法就是经常把用户最终想要的目标文件（可执行程序）放在 Makefile 文件中首要的位置，这样用户直接执行 make 命令即可。

## 3.2.2 Makefile 规则语法

简单地讲，Makefile 的作用就是让编译器知道要编译一个文件需要依赖哪些文件，同时当那些依赖文件有了改变，编译器会自动发现最终的生成文件已经过时，而重新编译相应的模块。Makefile 的内容规定了整个工程的编译规则。一个工程中的许多源文件按其类型、功能、模块可能分别被放在不同的目录中，Makefile 定义了一系列的规则来指定，比如哪些文件是有依赖性的，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。

Makefile 有其自身特定的编写格式并且遵循一定的语法规则。

```
#注释
目标文件：依赖文件列表
..............
<Tab>命令列表
..............
```

格式的说明如下。

- 注释：和 Shell 脚本一样，Makefile 语句行的注释采用“#”符号。
- 目标：目标文件的列表，通常指程序编译过程中生成的目标文件（.o 文件）或最终的可执行程序，有时也可以是要执行的动作，如“clean”这样的目标。
- 依赖文件：目标文件所依赖的文件，一个目标文件可以依赖一个或多个文件。
- “:”符号，分隔符，介于目标文件和依赖文件之间。
- 命令列表：make 程序执行的动作，也是创建目标文件的命令，一个规则可以有多条命令，每一行只能有一条命令。

**注意** 每一个命令行必须以[Tab]键开始，[Tab]告诉 make 程序该行是一个命令行，make 按照命令完成相应的动作。

从上面的分析可以看出，Makefile 文件的规则其实主要有两个方面，一个是说明文件之间的依赖关系，另一个是告诉 make 工具如何生成目标文件的命令。下面是一个简单的 makefile 文件例子。

```
#Makefile Example
test: main.o test1.o test2.o
     gcc -o test main.o test1.o test2.o
main.o: main.c head1.h head2.h
     gcc -c main.c
test1.o: test1.c  head2.h
     gcc -c test1.c
test2.o: test2.c head3.h
     gcc -c test2.c
install:
     cp  test  /home/tmp
clean:
     rm -f *.o
```

在这个 makefile 文件中，目标文件（target）即为：最终的可执行文件 test 和中间目标文件 main.o、test1.o、test2.o，每个目标文件和它的依赖文件中间用“:”隔开，依赖文件的列表之间用空格隔开。每一个.o 文件都有一组依赖文件，而这些.o 文件又是最终的可执行文件 test 的依赖文件。依赖关系实质上就是说明了目标文件是由哪些文件生成的。

在定义好依赖关系后，在命令列表中定义了如何生成目标文件的命令，命令行以 Tab 键开始。Make 工具会比较目标文件和其依赖文件的创建或修改日期，如果所依赖文件比目标文件要新，或者目标文件不存在的话，那么，make 就会执行命令行列表中的命令来生成目标文件。

## 3.2.3 Makefile 文件中变量的使用

Makefile 文件中除了一系列的规则，对于变量的使用也是一个很重要的内容。Linux 下的 Makefile 文件中可能会使用很多的变量，定义一个变量（也常称为宏定义），只要在一行的开始定义这个变量（一般使用大写，而且放在 Makefile 文件的顶部来定义），后面跟一个=号，=号后面即为设定的变量值。如果要引用该变量，用一个$符号来引用变量，变量名需要放在$后的()里。

make 工具还有一些特殊的内部变量，它们根据每一个规则内容定义。

- $@：指代当前规则下的目标文件列表。
- $<：指代依赖文件列表中的第一个依赖文件。
- $^：指代依赖文件列表中所有依赖文件。
- $?：指代依赖文件列表中新于对应目标文件的文件列表。

变量的定义可以简化 makefile 的书写，方便对程序的维护。例如前面的 Makefile 例程就可以如下书写。

```
#Makefile Example
```

```
OBJ=main.o test1.o test2.o
CC=gcc
test: $(OBJ)
      $(CC) -o test  $(OBJ)
main.o: main.c head1.h head2.h
     $(CC)-c main.c
test1.o: test1.c  head2.h
     $(CC) -c test1.c
test2.o: test2.c head3.h
     $(CC) -c test2.c
install:
     cp  test  /home/tmp
clean:
     rm  -f  *.o
```

从上面修改的例子可以看到，引入了变量 OBJ 和 CC，这样可以简化 makefile 文件的编写，增加了文件的可读性，而且便于修改。举个例子来说，假定项目文件中还需要加入另外一个新的目标文件 test3.o，那么在该 Makefile 中有两处需要分别添加 test3.o；而如果使用变量的话只需在 OBJ 变量的列表中添加一次即可，这对于更复杂的 Makefile 程序来说，会是一个不小的工作量，但是，这样可以降低因为编辑过程中的疏漏而导致出错的可能。

一般来说，Makefile 文件中变量的应用主要有以下几个方面。

## 1．代表一个文件列表

Makefile 文件中的变量常常存储一些目标文件甚至是目标文件的依赖文件，引用这些文件的时候引用存储这些文件的变量即可，这给 Makefile 编写和维护者带来了很大的方便。

## 2．代表编译命令选项

当所有编译命令都带有相同编译选项时（比如-Wall -O2 等），可以将该编译选项赋给一个变量，这样方便了引用。同时，如果改变编译选项的时候，只需改变该变量值即可，而不必在每处用到编译选项的地方都做改动。

在上面的 Makefile 例子中，还定义了一个伪目标 clean，它规定了 make 应该执行的命令，即删除所有编译过程中产生的中间目标文件。当 make 处理到伪目标 clean 时，会先查看其对应的依赖对象。由于伪目标 clean 没有任何依赖文件，所以 make 命令会认为该目标是最新的而不会执行任何操作。为了编译这个目标体，必须手工执行如下命令。

```
# make  clean
```

此时，系统会有提示信息：

```
rm -f *.o
```

另一个经常用到的伪目标是 install。它通常是将编译完成的可执行文件或程序运行所需的其他文件复制到指定的安装目录中，并设置相应的保护。例如在上面的例子中，如果用户执行命令：

```
# make  install
```

系统会有提示信息：

```
cp testl /home/tmp
```

也即是将可执行程序 test1 复制到系统/home/tmp 下。事实上，许多应用程序的 Makefile 文件也正是这样写的，这样便于程序在正确编译后可以被安装到正确的目录。

## 3.3 二进制代码工具的使用

### 3.3.1 GNU Binutils 工具介绍

在 Linux 下建立嵌入式交叉编译环境要用到一系列的工具链（tool-chain），主要有比如 GNU Binutils、Gcc、Glibc、Gdb 等，它们都属于 GNU 的工具集。其中，GNU Binutils 是一套用来构造和使用二进制所需的工具集。建立嵌入式交叉编译环境，Binutils 工具包是必不可少的，而且 Binutils 与 GNU 的 C 编译器 gcc 是紧密相集成的，没有 binutils，gcc 也不能正常工作的。Binutils 的官方下载地址是：ftp://ftp.gnu.org/gnu/binutils/，在这里可以下载不同版本的 Binutils 工具包。GNU Binutils 工具集里主要有以下一系列的部件。

- as：GNU 的汇编器。

作为 GNU Binutils 工具集中最重要的工具之一。as 工具主要用来将汇编语言编写的源程序转换成二进制形式的目标代码。Linux 平台的标准汇编器是 GAS，它是 GNU GCC 编译器所依赖的后台汇编工具，通常包含在 Binutils 软件包中。

- ld：GNU 的链接器。

同 as 一样，ld 也是 GNU Binutils 工具集中重要的工具，Linux 使用 ld 作为标准的链接程序，由汇编器产生的目标代码是不能直接在计算机上运行的，它必须经过链接器的处理才能生成可执行代码，链接是创建一个可执行程序的最后一个步骤，ld 可以将多个目标文件链接成为可执行程序，同时指定了程序在运行时是如何执行的。

- add2line：将地址转换成文件名或行号对，以便调试程序。
- ar：从文件中创建、修改、扩展文件。
- gasp：汇编宏处理器。
- nm：从目标代码文件中列举所有变量（包括变量值和变量类型），如果没有指定目标文件，则默认是 a.out 文件。
- objcopy：objcopy 工具使用 GNU BSD 库，它可以把目标文件的内容从一种文件格式复制到另一种格式的目标文件中。

在默认的情况下，GNU 编译器生成的目标文件格式为 elf 格式，elf 文件由若干段（section）组成，如果不作特殊指明，由 C 源程序生成的目标代码中包含如下段：.text（正文段）包含程序的指令代码；.data（数据段）包含固定的数据，如常量、字符串；.bss（未初始化数据段）包含未初始化的变量、数组等。C++源程序生成的目标代码中还包括.fini（析构函数代码）和.init（构造函数代码）等。链接生成的 elf 格式文件还不能直接下载到目标平台来运行执行，需要通过 objcopy 工具生成最终的二进制文件。连接器的任务就是将多个目标文件的.text、.data 和.bss 等段连接在一起，而连接脚本文件是告诉连接器从什么地址开始放置这些段。

- add2line：把程序地址转换为文件名和行号。

在命令行中带一个地址和一个可执行文件名，它就会使用这个可执行文件的调试信息指出在给出的地址上是哪个文件以及行号。

- objdump：显示目标文件信息。

objdump 工具可以反编译二进制文件，也可以对对象文件进行反汇编，并查看机器代码。

- readelf：显示 elf 文件信息。

readelf 命令可以显示符号、段信息、二进制文件格式的信息等，这在分析编译器如何从源代码创建二进制文件时非常有用。

- ranlib：生成索引以加快对归档文件的访问，并将其保存到这个归档文件中。

在索引中列出了归档文件各成员所定义的可重分配目标文件。

- size：列出目标模块或文件的代码尺寸。

size 命令可以列出目标文件每一段的大小以及总体的大小。默认情况下，对于每个目标文件或者一个归档文件中的每个模块只产生一行输出。

- strings：打印可打印的目标代码字符（至少 4 个字符），打印字符多少可以控制。

对于其他格式的文件，打印字符串。打印某个文件的可打印字符串，这些字符串最少 4 个字符长，也可以使用选项“-n”设置字符串的最小长度。默认情况下，它只打印目标文件初始化和可加载段中的可打印字符；对于其他类型的文件它打印整个文件的可打印字符，这个程序对于了解非文本文件的内容很有帮助。

- strip：放弃所有符号连接。　　删除目标文件中的全部或者特定符号。
- c++filt：链接器 ld 使用该命令可以过滤 C++符号和 Java 符号，防止重载函数冲突。
- gprof：显示程序调用段的各种数据。

## 3.3.2 Binutils 工具软件使用

就 Binutils 工具软件的使用问题，以下以 Binutils 工具包中两个常用的工具的使用进行简单的说明。

### 1．汇编器

Linux 平台的标准汇编器是 GAS，它是 GCC 所依赖的后台汇编工具，通常包含在 binutils 软件包中。GAS 使用标准的 AT&T 汇编语法可以用来汇编用 AT&T 格式编写的程序，例如可以这样来编译用汇编语言编写的源程序 test.s。

```
[root@localhost]# as  -o test.o  test.s
```

### 2．链接器

GNU 链接器使用一个命令语言脚本来控制链接过程。默认情况下，ld 是由一组内部命令进行控制的，这些命令可以进行扩展或覆盖。强调可移植性和灵活性在 GCC 的功能中是非常明显的一条，它可以为很多不同的编译环境生成链接脚本，并向 ld 传递定制过的链接脚本，而不用手工进行干预。
需要注意的是，在 Linux 下编写应用程序（假定采用 gcc 编译器）时，gcc 编译器内置缺省的连接脚本。如果采用缺省脚本，则生成的目标代码需要操作系统才能加载运行。
就像前面讲到的，由汇编器产生的目标代码是不能直接在计算机上运行的，它必须经过链接器的处理才能生成可执行代码。Linux 使用 ld 作为标准的链接程序，比如我们可以用下面的方法来链接上述编译的程序。

```
[root@localhost]# ld -s –o  test  test.o
```

这样就生成了最终的可执行程序 test。

## 3.4 编译器 GCC 的使用

## 3.4.1 GCC 编译器介绍

GCC 是 GNU 项目的编译器组件之一，也是 GNU 软件产品家族具有代表性的作品。在 GCC 设计之初，仅仅是作为一个 C 语言的编译器，可是经过十多年的发展，GCC 已经不仅仅能支持 C 语言；它现在还支持 Ada 语言、C++语言、Java 语言、Objective C 语言、Pascal 语言、COBOL 语言，以及支持函数式编程和逻辑编程的 Mercury 语言等。而 GCC 也不再单只是 GNU C Compiler 的意思了，而是变成了 GNU Compiler Collection 即 GNU 编译器家族的意思了，目前已成为 Linux 下最重要的软件开发工具之一。GCC 的发展大体经历了下面的几个阶段。

① 1987 年，第一版的 GCC 发布。
② 2001.6.18，GCC3.0 正式发布。
③ 2004.4.20，GCC 3.4.0 版本发布。
④ 2005.4.22，最新版本的 GCC 4.0 发布，官方网站：http://gcc.gnu.org。
⑤ 2008.8.27，GCC4.3.2 发布。

GCC 是一个交叉平台的编译器，目前支持几乎所有主流 CPU 处理器平台，它可以完成从 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 不仅功能非常强大，结构也异常灵活，便携性（portable）与跨平台支持（cross-platform support）特性是 GCC 的显著优点，目前，GCC 编译器所能够支持的源程序的格式如表 3.6 所示。

表 3.6　　GCC 所支持的源程序格式

| 后 缀 格 式 | 说　　明 |
|---|---|
| .c | C 语言源程序 |
| .a | 由目标文件构成的档案库文件 |
| .C;；cc；.cxx | C++源程序 |
| .h | 源程序包含的头文件 |
| .i | 经过预处理的 C 程序 |
| .ii | 经过预处理的 C++程序 |
| .m | Objective-C 源程序 |
| .o | 编译后的目标文件 |
| .s | 汇编语言源程序 |
| .S | 经过预编译的汇编程序 |

GCC 是一组编译工具的总称，其软件包里包含众多的工具，按其类型主要有以下的分类。
① C 编译器；cc、cc1、cc1plus、gcc。
② C++编译器：c++、cc1plus、g++。
③ 源码预处理程序：cpp、cpp0。
④ 库文件：libgcc.a、libgcc_eh.a、libgcc_s.so、libiberty.a、libstdc++.[a,so]、libsupc++.a。

用 GCC 编译程序生成可执行文件有时候看起来似乎仅通过编译一步就完成了，但事实上，使用 GCC 编译工具由 C 语言源程序生成可执行文件的过程并不单单是一个编译的过程，而要经过下面的几个过程。

- 预处理（Pre-Processing）。
- 编译（Compiling）。
- 汇编（Assembling）。
- 链接（Linking）。

在实际编译的时候，GCC 首先调用 cpp 命令进行预处理，主要实现对源代码编译前的预处理，比如将源代码中指定的头文件包含进来。接着调用 cc1 命令进行编译，作为整个编译过程的一个中间步骤，该过程会将源代码翻译生成汇编代码。汇编过程是针对汇编语言的步骤，调用 as 命令进行工作，生成扩展名为.o 的目标文件，当所有的目标文件都生成之后，GCC 就调用链接器 ld 来完成最后的关键性工作——链接。

## 3.4.2　GCC 编译选项解析

GCC 是 Linux 下基于命令行的 c 语言编译器，其基本的使用语法如下。

```
gcc [option | filename ]…
```

对于编译 C++的源程序，其基本的语法如下。

```
g++ [ option | filename ]…
```

其中 option 为 GCC 使用时的选项（后面会再详述），而 filename 为需要用 GCC 作编译处理的文件名。就 GCC 来说，其本身是一个十分复杂的命令，合理地使用其命令选项可以有效提高程序的编译效率、优化代码，GCC 拥有众多的命令选项，有超过 100 个的编译选项可用，按其应用有如下的分类。

## 1．常用编译选项

- -c 选项：这是 GCC 命令的常用选项。-c 选项告诉 GCC 仅把源程序编译为目标代码而并不做链接的工作，所以采用该选项的编译指令不会生成最终的可执行程序，而是生成一个与源程序文件名相同的以.o 为后缀的目标文件。例如一个 Test1.c 的源程序经过下面的编译之后会生成一个 Test1.o 的文件。

```
# gcc  -c  Test1.c
```

- -S 选项：使用该选项会生成一个后缀名为.s 的汇编语言文件，但是同样不会生成可执行的程序。
- -e 选项：-e 选项只对文件进行预处理，预处理的输出结果被送到标准输出（比如显示器）。
- -v 选项：在 Shell 的提示符号下键入 gcc -v，屏幕上就会显示出目前正在使用的 GCC 的版本信息。例如：

```
# gcc  -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux7/2.96/specs
gcc version 2.96 20000731（Redhat linux 7.3 2.96-128）
```

上面的系统信息指出了 GCC 的版本：gcc 2.96。

- -x language：强制编译器用指定的语言编译器来编译某个源程序。

例如下面的指令：

```
# gcc  -x  c++  P1.c
```

该指令表示强制采用 C++编译器来编译 C 程序 P1.c。

- -I<DIR>选项：库依赖选项，指定库及头文件路径。

在 Linux 下开发程序的时候，通常来讲都需要借助一个或多个函数库的支持才能够完成相应的功能。一般情况下，Linux 下的大多数函数都将头文件放到系统/usr/include/目录下，而库文件则放到/usr/lib/目录下。但在有些情况下并不是这样的，在这些情况下，使用 GCC 编译时必须指定所需要的头文件和库文件所在的路径。-I 选项可以向 GCC 的头文件搜索路径中添加新的目录<DIR>。例如，一个源程序所依赖的头文件在用户/home/include/目录下，此时就应该使用-I 选项来指定。

```
# gcc -I /home/include -o  Test  Test.c
```

- -L<DIR>：类似上面的情况，用来特别指定所依赖库所在的路径。

如果使用了不在标准位置的库，那么可以通过-L 选项向 GCC 的库文件搜索路径中添加新的目录。例如，一个程序要用到的库 libapp.so 在/home/zxq/lib/目录下，为了能让 GCC 能够顺利地链接该库，可以使用下面的命令：

```
#gcc –Test.c –L /home/zxq/lib –lapp –o Test
```

这里的-L 选项表示 GCC 去连接库文件 libapp.so。Linux 下的库文件在命名时有一个约定，那就是应该以 lib 三个字母开头，由于所有的库文件都遵循了同样的规范，因此在用-L 选项指定链接的库文件名时可以省去 lib 三个字母，也就是说 GCC 在对-lapp 进行处理时，会自动去链接名为 libapp.so 的文件。

- -static 选项：GCC 在默认情况下链接的是动态库，有时为了把一些函数静态编译到程序中，而无需链接动态库就采用-static 选项，它会强制程序链接静态库。
- -o 选项：在默认的状态下，如果 GCC 指令没有指定编译选项的情况下会在当前目录下生成一个名为 a.out 的可执行程序，例如：执行# gcc Test.c 命令之后会生成一个 a.out 的可执行程序。因此，为了指定生成的可执行程序的文件名，就可以采用-o 选项，比如下面的指令：

```
# gcc –o  Test  Test.c
```

执行该指令会在当前目录下生成一个名为 Test 的可执行文件。

**注意** 使用-o 选项时，-o 后面必须带有可执行文件的文件名（可以任意指定）。

## 2．出错检查和警告提示选项

GCC 编译器包含完整的出错检查和警告提示功能，比如 GCC 提供了 30 多条警告信息和 3 个警告级别，使用这些选项有助于增强程序的稳定性和更加完善程序代码的设计，此类选项常用的如下。

- -pedantic：以 ANSI/ISO C 标准列出的所有警告。

当 GCC 在编译不符合 ANSI/ISO C 语言标准的源代码时，如果在编译指令中加上了-pedantic 选项，那么源程序中使用了扩展语法的地方将产生相应的警告信息。

- -w：禁止输出警告消息。
- -Werror：将所有警告转换为错误。

Werror 选项要求 GCC 将所有的警告当成错误进行处理，这在使用自动编译工具（如 Make 等）时非常有用。如果编译时带上-Werror 选项，那么 GCC 会在所有产生警告的地方停止编译，只有程序员对源代码进行修改并且相应的警告信息消除时，才可能继续完成后续的编译工作。

- -Wall：显示所有的警告消息。

-Wall 选项可以打开所有类型的语法警告，以便于确定程序源代码是否是正确的，并且尽可能实现可移植性。

对 Linux 程序开发人员来讲，GCC 给出的警告信息是很有价值的，它们不仅可以帮助程序员写出更加健壮的程序，而且还是跟踪和调试程序的有力工具。建议在用 GCC 编译源代码时始终带上-Wall 选项，养成良好的习惯。

## 3．代码优化选项

代码优化指的是编译器通过分析源代码找出其中尚未达到最优的部分，然后对其重新进行组合，进而改善代码的执行性能。GCC 通过提供编译选项-On 来控制优化代码的生成，对于大型程序来说，使用代码优化选项可以大幅度提高代码的运行速度。

- -O 选项：编译时使用选项-O 可以告诉 GCC 同时减小代码的长度和执行时间，其效果等价于-O1。
- -O2 选项：选项-O2 告诉 GCC 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。

## 4．调试分析选项

- -g 选项：生成调试信息，GNU 调试器可利用该信息。GCC 编译器使用该选项进行编译时，将调试信息加入到目标文件当中，这样 gdb 调试器就可以根据这些调试信息来跟踪程序的执行状态。
- -pg 选项：编译完成之后，额外产生一个性能分析所需的信息。

**注意** 需要注意的是，使用调试选项都会使最终生成的二进制文件的大小急剧增加，同时增加程序在执行时的开销，因此调试选项通常推荐仅在程序的开发和调试阶段中使用。

下面举一个简单的例子来说明 GCC 的编译过程。首先用 vi 编辑器来编辑一个简单的 C 程序 test.c，程序清单如下。

```
#include <stdio.h>
int main()
{
 printf("Hello,this is a test!\n");
 return 0;
}
```

根据前面讲到的内容，使用 gcc 命令来编译该程序。

```
[root@localhost]# gcc –o test test.c
[root@localhost]#./test
Hello,this is a test!
```

可以从上面的编译过程看到，编译一个这样的程序非常简单，一条指令即可完成，事实上，这一条指令掩盖了很多细节。我们可以从编译器的角度来看上述的编译过程，这对于更好理解 GCC 编译工作原理有很好的帮助。

GCC 编译器首先做的工作是预处理：调用-E 参数可以让 GCC 在预处理结束后停止编译过程。

```
# gcc –E test.c -o test.i
```

编译器在这一步调用 cpp 工具来对源程序进行预处理，此时会生成 test.i 文件，下面部分列出了 test.i 文件中的内容。

```
# 1 "test.c"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "test.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 28 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 314 "/usr/include/features.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 315 "/usr/include/features.h" 2 3 4
# 337 "/usr/include/features.h" 3 4
# 1 "/usr/include/gnu/stubs.h" 1 3 4
# 338 "/usr/include/features.h" 2 3 4
# 29 "/usr/include/stdio.h" 2 3 4
# 1 "/usr/lib/gcc/i386-redhat-linux/3.4.4/include/stddef.h" 1 3 4
# 213 "/usr/lib/gcc/i386-redhat-linux/3.4.4/include/stddef.h" 3 4
typedef unsigned int size_t;
# 35 "/usr/include/stdio.h" 2 3 4
# 1 "/usr/include/bits/types.h" 1 3 4
# 28 "/usr/include/bits/types.h" 3 4
# 1 "/usr/include/bits/wordsize.h" 1 3 4
# 29 "/usr/include/bits/types.h" 2 3 4
# 1 "/usr/lib/gcc/i386-redhat-linux/3.4.4/include/stddef.h" 1 3 4
# 32 "/usr/include/bits/types.h" 2 3 4
typedef unsigned char __u_char;
typedef unsigned short int __u_short;
typedef unsigned int __u_int;
typedef unsigned long int __u_long;
typedef signed char __int8_t;
```

```
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;
__extension__ typedef signed long long int __int64_t;
__extension__ typedef unsigned long long int __uint64_t;
```

查看代码会发现 stdio.h 的内容都被加入到该文件里去了，而且被预处理的宏定义也都作了相应的处理。
下一步是将 test.i 编译为目标代码，这可以通过使用-c 参数来完成。

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

GCC 默认将.i 文件看成是预处理后的 C 语言源代码，因此上述命令将自动跳过预处理步骤而开始执行编译过程，也可以使用-x 参数让 GCC 从指定的步骤开始编译。
编译的最后一步是将上一步所生成的目标文件链接成最终的可执行文件。

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

## 3.5 调试器 GDB 的使用技巧

### 3.5.1 GDB 调试器介绍

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（GNU Debugger），该软件最早由 Richard Stallman 编写，GDB 是一个用来调试 C 和 C++程序的调试器（Debugger）。使用者能在程序运行时观察程序的内部结构和内存的使用情况，GDB 是一种基于命令行工作模式下的程序，工作在字符模式，由多个不同的图形用户界面前端予以支持，每个前端都能以多种方式提供调试控制功能，它的功能非常丰富，适用于修复程序代码中的问题，在 X Window 系统中，基于图形界面的调试工具称为 xxgdb。目前比较新的版本是 GDB 6.8（2008 年 4 月 7 日发布），其官方网站是 http://www.gnu.org/software/gdb/。
以下是 GDB 所提供的一些功能。

- 启动程序，并且可以设置运行环境和参数来运行指定程序。
- 让程序在指定断点处停止执行。
- 对程序做出相应的调整，这样就能纠正一个错误后继续调试。

需要注意的是，GDB 调试的是可执行文件，而不是源程序，如果想让 GDB 调试编译后生成的可执行文件，在使用 GDB 工具调试程序之前，必须使用带有-g 或-gdb 编译选项的 gcc 命令来编译源程序，例如：

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

只有这样会在目标文件中产生相应的调试信息。调试信息包含源程序的每个变量的类型和在可执行文件里的地址映射以及源代码的行号，GDB 利用这些信息使源代码和机器码相关联。
使用 gdb 命令的语法如下。

```
# gdb  [参数]  Filename
```

下面列举一些常用的参数。

- -help：列出所有参数，并作简要说明。
- -symbols=file

-s file：读出文件（file）的所有符号。

- -core

-c
这里的 core 是程序非法执行后 core dump 后产生的文件。

- -directory

-d

加入一个源文件的搜索路径。默认搜索路径是环境变量中 PATH 所定义的路径。

- -quiet

-q

使用该参数不显示 gdb 的介绍和版权信息等。

## 3.5.2 GDB 调试命令

运行 GDB 调试程序通常使用如下的命令。

# gdb Filename

```
Copyright 2004 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)
```

之后就可以在系统的（gdb）提示符后面输入相应的调试命令了，如果不希望出现 gdb 的系统信息提示，可以输入下面的指令：

```
# gdb   -q  Filename
```

表 3.7 列举了一些常用到的 GDB 调试命令。

**表 3.7　　常用 GDB 命令**

| 命　令 | 说　明 |
|---|---|
| file | 指定要调试的可执行程序 |
| kill | 终止正在调试的可执行程序 |
| next | 执行一行源代码但并不进入函数内部 |
| list | 部分列出源代码 |
| step | 执行一行源代码并不进入函数内部 |
| run | 执行当前的可执行程序 |
| quit | 结束 gdb 调试任务 |
| watch | 可以检查一个变量的值，而不管它何时被改变 |
| print | 打印表达式的值到标准输出 |
| break N | 在指定的第 N 行源代码设置断点 |
| info break | 显示当前断点清单，包括到达断点处的次数等 |
| info files | 显示被调试文件的详细信息 |
| info func | 显示所有的函数名 |
| info local | 显示当函数中的局部变量信息 |
| info prog | 显示被调试程序的执行状态 |
| info var | 显示所有的全局和静态变量名称 |
| make | 在不退出 gdb 的情况下运行 make 工具 |
| shell | 在不退出 gdb 的情况下运行 shell 命令 |
| continue | 继续执行正在调试的程序 |

下面举一个简单的例子来说明 GDB 调试命令的使用方法，下面的程序很简单，即通过用户输入一个圆的半径值来求得圆面积，其源代码如下。

```
#include<stdio.h>
#include<math.h>
int main(void)
{
  float Pi=3.1415926;
  float R;
  float S=0;
  printf("Please input your Ridus:\n");
  scanf("%f",&R);
  if (R>=0)
    {
      S=Pi*R*R;
      printf("The value of S is:%f\n",S);
    }
  else
    printf("Sorry,Wrong input!!\n");
  return 0;
}
```

为了方便调试可执行程序，可以用下面的语句来编译该程序。

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

开始调试：

```
# gdb -q new
Using host libthread_db library "/lib/tls/libthread_db.so.1"
(gdb)
```

出现了（gdb）提示符以后，就可以输入相应的调试命令了。

（1）查看源代码，使用 list 命令。

```
(gdb) list
1   #include<stdio.h>
2   int main(void)
3   {
4   float Pi=3.1415926;
5   float R;
6   float S=0;
7   printf("Please input your Ridus:\n");
(gdb)
```

如上所示，使用 list 命令之后部分地列出了源代码，而且每行都有相应的标号，如果想列出更多的源代码，可以继续输入 list 命令（或者直接回车即可）。

（2）运行该程序，使用 run 命令。

```
(gdb) run
Starting program /home/zxq/new
Please input  your Radius:
10
The value of S is: 314.15926
Program exited normally
(gdb)
```

如上所示，使用 run 命令会执行编译后生成的可执行程序 new。

（3）设置断点。

gdb 可以使用 break N 命令来设置断点，N 表示在源代码的第 N 行处设置断点，例如：

```
（gdb）break 13
Breakpoint 1 at 0x804840a: file new.c,line 13.
```

这样程序执行到第 13 行语句处就会停止执行，

```
(gdb) run
Starting program: /home/zxq/new
Please input your Ridus:
9.5
Breakpoint 1, main () at new.c:13    /*指出程序执行停止的位置*/
13   printf("The value of S is:%f\n",S);
（gdb）
```

如果想看到程序中设置断点的数量或断点位置，可以使用 info break 命令来查看：

```
(gdb) info break
Num    Type     Disp Enb  Address  What
1   breakpoint       keep y  0x0804839c in main at new.c:4
2   breakpoint       keep y  0x08048426  in main at new.c:14
(gdb)
```

从上面的信息可以看到程序分别在第 4、14 行处设置了断点。

（4）清除断点。

c1ear 是一条用来清除断点的命令，在程序调试过程中，如果确定设置断点的语句处没有必要再暂停运行，就可以用 c1ear 命令来清除设置的断点。它的使用格式是：

```
（gdb） clear  n
```

在上述例子中，清除第 6 行处的断点的做法如下：

```
(gdb)  clear  13
Deleted breakpoint  l
```

事实上，比删除更好的一种方法是 disable 命令，关闭了断点，它并不会被删除，它只是让所设断点暂时失效，当还需要改断点时，enable 即可。

（5）查看变量的值。

当程序执行到断点处停止以后，往往要查看某些变量的值，进而观察程序的执行状态，gdb 采用 print 命令来查看指定变量的值，例如：

```
（gdb）break 13
Breakpoint 1 at 0x804840a: file new.c, line 13.
(gdb) run
Starting program: /home/zxq/new
Please input your Ridus:
9.5
Breakpoint 1, main () at new.c:13
13   printf("The value of S is:%f\n",S);
(gdb) print S          /*查看变量 S 的值*/
$1 = 283.528717
（gdb）
```

如果想看到变量的类型，可使用 whatis 命令，如：

```
(gdb) whatis S
type = float              /*变量 S 类型为 float*/
(gdb)
```

（6）单步执行。

gdb 提供以下两种方式。

step 指令，单步进入，可以跟踪到函数内部。命令是 step 或 s。

next 指令，单步，只是简单的单步执行，不会进入函数内部。

以上只是部分地列出了一些 GDB 调试指令的用法，事实上 GDB 具有非常的调试指令，具体详细的使用可参见 GNU GDB 使用手册。

（7）搜索源代码。

gdb 还提供了源代码搜索的命令。

向前搜索：

```
(gdb) forward-search  <regexp>
(gdb) search <regexp>
```

全部搜索：

```
(gdb) reverse-search <regexp>
```

其中，<regexp>就是正则表达式。

（8）指定源文件的路径。

某些时候，用-g 编译过后的执行程序中只是包括了源文件的名字，没有路径名。GDB 提供了可以指定源文件的路径的命令，以便 GDB 进行搜索要调试的源程序。

```
(gdb) dir <dirname ... >
```

（9）结束当前程序的调试。

kill 命令用来结束当前程序的调试。在 gdb 下直接输入下面这条命令即可结束程序的调试过程。

```
(gdb)  kill
Kill  programm being  debugged(y or n)
```

确认即可结束调试。

## 3.6 Linux 编程库

### 3.6.1 Linux 编程库介绍

所谓编程库就是指始终可以被多个 Linux 软件项目重复使用的代码集。以 C 语言为例，它包含了几百个可以重复使用的例程和调试程序的工具代码，其中包括函数。如果每次编写新程序都要重新写这些函数会非常不方便。使用编程库有两个主要优点。

- 可以简化编程，实现代码重复使用，进而减小应用程序的大小；
- 可以直接使用比较稳定的代码。

Linux 下的库文件分为共享库和静态库两大类，它们两者的差别仅在程序执行时所需的代码是在运行时动态加载的，还是在编译时静态加载的。此外，通常共享库以.so(Shared Object)结尾，静态链接库通常以.a 结尾（Archive）。在终端下查看库的内容，通常共享库为绿色，而静态库为黑色。

Linux 的库一般在/lib 或/usr/lib 目录下。它主要存放系统的链接库文件，没有该目录则系统无法正常运行。/lib 目录中存储着程序运行时使用的共享库。通过共享库，许多程序可以重复使用相同的代码，因此可以有效减小应用程序的大小。表 3.8 部分列出了一些 Linux 下常用到的编程库。

**表 3.8　　常用到的 Linux 编程库**

| 库 名 称 | 说 明 |
|---|---|
| libc.so | 标准的 C 库 |
| libdl.so | 可以使用库的源代码而无需静态编译库 |
| libglib.so | Glib 库 |
| libm.so | 标准数学库 |
| libGL.so | OpenGL 的接口 |
| libcom_err.so | 常用出错例程集合 |
| libdb.so | 创建和操作数据库 |
| libgthread.so | Glib 线程支持 |
| libgtk.so | GIMP 下的 X 库 |
| libz.so | 压缩例程库 |
| libvga.so | Linux 的 VGA 和 SVGA 图形库 |
| libresolve.so | 提供使用因特网域名服务器接口 |
| libpthread.so | Linux 多线程库 |
| libgdm.so | GNU 数据库管理器 |

## 3.6.2 Linux 系统调用

从字面意思上理解，系统调用说的是操作系统提供给用户程序调用的一组“特殊”接口。Linux 中用于创建进程的 fork()函数本身就是一个系统调用，使用系统主要目的是使得用户可以使用操作系统提供的有关设备管理、输入/输出系统、文件系统和进程控制、通信以及存储管理等方面的功能，而不必了解系统程序的内部结构和有关硬件细节，从而起到减轻用户负担和保护系统以及提高资源利用率的作用。

Linux 的运行空间划分为用户空间和内核空间，它们各自运行在不同的级别中，所以用户进程在通常情况下不允许访问内核，也无法使用内核函数，它们只能在用户空间操作用户数据，调用用户空间函数。这样做的目的是为了对系统作必要的“保护”措施，但是使用系统调用可以最大程度地解决这一问题。其具体的措施是进程先用适当的值填充寄存器，然后调用一个特殊的指令，这个指令会跳到一个事先定义的内核中的一个位置（当然，这个位置是用户进程可读但是不可写的）。硬件知道一旦用户进程跳到这个位置，则认为该用户就不是在限制模式下运行的用户，而是作为操作系统的内核。当然，用户访问内核的路径是事先规定好的，只能从规定位置进入内核，而不允许任意跳入内核。

Linux 系统有 200 多个系统调用，这些系统调用按照功能分类大致可分为以下几个方面。

- 进程控制。
- 文件系统控制。
- 系统控制。
- 内存管理。
- 网络管理。
- socket 控制。
- 用户管理。
- 进程间通信。

类似于在 Windows 下面进行过 Win32 编程，Windows 会提供 API（Application Programming Interface）接口函数作为 Windows 操作系统提供给程序员的系统调用接口。同样的，Linux 作为一个操作系统也有它自己的系统调用，用户可以根据特定的方法来添加需要的系统调用。Linux 的 API 接口遵循 POSIX 标准，这套标准定义了一系列 API。在 Linux 中，这些 API 主要是通过 C 库（libc）实现的。下面通过举例来说明在 Linux 下添加新的系统调用的几个步骤。

（1）修改 kernel/sys.c，增加服务例程代码。首先编写添加到内核中的源程序，即要添加的服务，所用函数的名称应该是新的系统调用名称前面加上 sys_标志。例如新加的系统调用为 mysyscall（int number），那么就应该在系统的/usr/src/linux/kernel/sys.c 文件中添加相应的源代码，如下所示。

```
asmlinkage int sys_mysyscall(int number)
{
    printk("This is a example of systemcall \n ");
  return number;
}
```

为了说明问题，仅仅是一个返回一个值的简单例子。

**注意** 系统调用函数通常在成功时返回 0 值，不成功时返回非零值。

（2）添加新的系统调用后，为了从已有的内核程序中增加到新的函数的连接，需要编辑以下两个文件。

① /usr/src/linux/include/asm-i386/unistd.h

② /usr/src/linux/arch/i386/kernel/syscall_table.S

第 1 个文件中定义了每个系统调用的中断号，可以打开文件/usr/src/linux/include/ asm-i386/unistd.h 来查看，该文件中包含了系统调用清单，用来给每个系统调用分配一个唯一的号码，部分内容如下。

```
.............
#define __NR_add_key                    286
#define __NR_request_key                287
#define __NR_keyctl                     288
#define __NR_ioprio_set                 289
#define __NR_ioprio_get                 290
#define __NR_inotify_init          291
#define __NR_inotify_add_watch     292
#define __NR_inotify_rm_watch      293
#define    NR_syscalls                  294
.............
```

文件中每一行的格式如下。

```
# define __NR_syscallname  N
```

syscallname 为系统调用名，而 N 则是该系统调用对应的中断号，每个系统调用都有唯一的中断号。应该将新的系统调用名称加到清单的最后，并给它分配号码序列中下一个可用的系统调用号。在该文件中的最后一句：#define NR_syscalls 294

NR_syscalls 表示系统调用的个数，294 表示有 294 个系统调用，标号从 0 开始，所以最后一个系统调用号是 293，那么如果新添加一个系统调用其中断号就应该是 294。

例如可以在该文件中如下定义一个系统调用：

```
# define   __NR __mysyscall   294
```

如果还需要添加另外的系统调用，可以此类推将中断号依次递增。此外需要注意的是，重新添加系统调用之后，应该在/usr/src/linux/include/asm-i386/unistd.h 文件中的#define NR_syscalls 语句重新指定的编号 n，例如在上面添加一个新的系统调用之后，该语句应该为：

```
# define       NR_syscalls     295
```

第 2 个要编辑的文件是：/usr/src/linux/arch/i386/kernel/syscall_table.S。该文件中定义了系统调用列表。在该文件中有以下类似的内容。

```
.data
ENTRY(sys_call_table)
    .long sys_restart_syscall  /* 0 - old "setup()" system call, used for restarting */
    .long sys_exit
    .long sys_fork
```

```
    .long sys_read
    .long sys_write
    .long sys_open              /* 5 */
    .long sys_close
    .long sys_waitpid
    .long sys_creat
    .long sys_link
    .long sys_unlink        /* 10 */
.............
```

在该文件中添加新的系统调用：

```
    .long sys_mysyscall
```

（3）重新编译内核。添加好系统调用之后，需要重新编译内核，并且用新的内核来启动，此时，系统调用就添加好了，重新编译内核的过程在这里不做详细介绍。

（4）测试新的系统调用，编辑程序 test_call.c 如下。

```
#include <linux/unistd.h>
#include <stdio.h>
#include <errno.h>
_syscall1(int, mysyscall, int, num);  /*系统调用宏定义*/
int main(void)
{
          int n;
          n=mysyscall(10);           /*执行系统调用*/
          printf("n=%d\n",n);
          return 0;
}
```

编译并执行该程序。

```
# gcc -o  test_call  -I/usr/src/linux/include  test_call.c
# ./test_call
n=10
```

输出值正确，说明添加系统调用就成功了。

## 3.6.3 Linux 线程库

简单地讲，进程是资源管理的最小单位，线程是程序执行的最小单位。一个进程至少需要一个线程作为它的指令执行体，进程管理着资源（比如 CPU、内存、文件等），而将线程分配到某个 CPU 上执行。一个进程当然可以拥有多个线程。

Linux 是一个多用户多任务的操作系统。多用户是指多个用户可以在同一时间使用计算机系统；多任务是指 Linux 可以同时执行几个任务，它可以在还未执行完一个任务时又执行另一项任务。在操作系统设计上，从进程演化出线程，最主要的目的就是更好地支持多处理器以及减小（进程/线程）上下文切换开销。

现在，多线程技术已经被许多操作系统所支持，包括 Windows/Linux。现在有 3 种不同标准的线程库：WIN32、OS/2 和 POSIX。其中前两种只能用在它们各自的平台上（WIN32 线程仅能运行于 Windows 平台上，OS/2 线程运行于 OS/2 平台上）。POSIX（Portable Operating System Interface Standard，可移植操作系统接口标准）规范则是适用于各种平台，而且已经或正在所有主要的 Unix/Linux 系统上实现。

Linux 系统下的多线程遵循 POSIX 接口，称为 pthread。POSIX 标准由 IEEE 制定，并由国际标准化组织接受为国际标准。在 Linux 2.6 内核版本之前，LinuxThreads 是现有 Linux 平台上使用最为广泛的线程库，它由 Xavier Leroy 负责开发完成，并已绑定在 Glibc 中发行。LinuxThreads 是一种面向 Linux 的 POSIX

1003.1c-pthread 标准接口。它所实现的就是基于核心轻量级进程的“一对一”线程模型，一个线程实体对应一个核心轻量级进程，而线程之间的管理在核外函数库中实现。使用 LinuxThreads 线程库创建和管理线程常用到下面几个函数。

- pthread_create()：创建新的线程。

pthread_create()函数类似 fork()函数，完整的函数形式如下。

```
int pthread_create（pthread_t thread，const pthread_attr_t*attr，void*（*func）（void*），void *arg）
```

第 1 个参数是一个 pthread_t 型的指针用于保存线程 ID，以后对该线程的操作都要用 ID 来标示。每个 LinuxThreads 线程都同时具有线程 ID 和进程 ID，其中进程 ID 就是内核所维护的进程号，而线程 ID 则由 LinuxThreads 分配和维护。

第 2 个参数是一个 pthread_attr_t 的指针用于说明要创建的线程的属性，使用 NULL 表示要使用缺省的属性。

第 3 个参数指明了线程运行函数的起始地址，是一个只有一个（void *）参数的函数。

第 4 个参数指明了运行函数的参数，参数 arg 一般指向一个结构。

函数返回值类型为整数，当创建线程成功时，函数返回 0，若不为 0 则说明创建线程失败。创建线程成功后，新创建的线程则运行参数 3 和参数 4 确定的函数，原来的线程则继续运行下一行代码。

- pthread_join()：等待线程结束。

pthread_join()函数用来挂起当前线程直到由参数 thread 指定的线程终止为止，完整的函数形式如下。

```
int pthread_join（pthread_t thread，void* *status ）
```

第 1 个参数为被等待的线程标识符，第 2 个参数为一个用户定义的指针，它可以用来存储被等待线程的返回值。

函数返回值类型为整数，成功返回 0，错误返回非零值。

- pthread_self()：获取线程 ID。

函数原型

```
pthread_t pthread_self(void)
```

函数返回本线程的 ID。

- pthread_detach()：用于让线程脱离。

pthread_detach()函数用于将处于连接状态的线程变为脱离状态，函数完整的形式如下。

```
int pthread_detach（pthread_t thread）
```

函数返回值类型为整数，如果成功将线程转换为脱离态时返回 0，否则返回非零值。

- pthread_exit()：终止线程。

pthread_exit()函数用来终止线程，函数完整形式如下。

```
pthread_exit（void *status）
```

参数 status 是指向线程返回值的指针。

下面通过一个简单的例子来介绍基于 POSIX 线程接口的 Linux 多线程编程，编写 Linux 下的多线程程序，需要使用头文件 pthread.h，连接时需要使用库 libpthread.a。以下是一个简单的例子。

```
/*mypthread.c*/
#include <pthread.h>
#include <stdlib.h>
#include <unistd.h>
void *thread_function(void *arg)
{
  int i;
  for ( i=0; i<20; i++)
 {
```

```
        printf("This is a thread!\n");
    }
     return NULL;
   }
   int main(void)
    {
       pthread_t mythread;
       if ( pthread_create( &mythread, NULL, thread_function, NULL) )
   {
        printf("error creating thread.");
        abort();
     }

      printf("This is main process!\n");
      if ( pthread_join ( mythread, NULL ) )
    {
       printf("error joining thread.");
       abort();
     }
     exit(0);
   }
```

编译并执行该程序：

```
#gcc –lpthread –o mypthread mypthread.c
./mypthread
This is main process!
This is a thread!
```

一个线程实际上就是一个函数，创建后，该线程立即被执行。在上面的例程中，系统创建了一个主线程，又用 pthread_create 创建了一个新的子线程。

事实上，在 Linux 2.6 内核以前，Linux 把进程当作其调度实体，内核并不真正支持线程（轻量线程实现）。它提供了一个 clone()系统调用来创建一个调用进程的拷贝，这个拷贝与调用者共享地址空间。LinuxThreads 项目就是利用这个系统调用，完全在用户级模拟了线程。Linuxthread 线程库目前存在一些不足之处，比如在信号处理、任务调度以及进程间同步原语等方面。在 Linux 2.6.x 内核中，Linux 内核的调度性能得到了很大改进。Linux 重写了其线程库，使用 NPTL（Native Posix Thread Library）来取代受争议的 LinuxThreads 线程库，成为 glibc 的首选线程库，与此同时，最新发布的 glibc 2.4 版本正式采用 NTPL 作为 pthread 实现。

## 联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路 250 号银海大厦 11 层 B 区，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-25590506

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218

广州地址：广州市天河区中山大道 268 号天河广场 3 层，电话：020-28916067