附录A　GCC使用介绍

A.1　获得GCC的方法

GCC是一种C++语言的编译器，它可以免费获得，其功能十分强大。你可以在 URL:ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/ 的网站上找到正式的Linux GCC发布系统，而且是已经编译好的可执行文件。

自由软件基金会（Free Software Foundation）所发布的GCC最新源代码可以从网站 GNU archives3上取得。没有必要非得使用上述的版本，不过这个版本的确是目前最新的。 Linux GCC的维护网友让你可以很轻松的自行编译这个最新的版本。 configure命令脚本会帮你自动设置好所有该做的事情。

A.2　C程序库与头文件

该选哪一套程序库取决于你的系统是ELF格式的还是a.out格式的，以及你希望系统变成哪一种格式。

• libc-5.2.18.bin.tar.gz

ELF共享程序库，静态程序库与头文件。

• libc-5.2.18.tar.gz

libc-5.2.18.bin.tar.gz的源代码。你也需要这个文件，因为.bin.套件中含有必需的头文件。

• libc-4.7.5.bin.tar.gz

这个文件是a.out的共享程序库与静态程序库，是为了与前述的libc5套件兼容而设计的。除非你想要继续使用a.out的程序或者继续开发a.out的程序，否则，是不需要它的。

A.3　一些有用的工具 (as、ld、 ar、 strings 等)

你也可以从网站tsx-116上找到这些工具程序。目前的版本是binutils-2.6.0.2.bin.tar.gz。

需要注意的是binutils只适用于ELF格式，而且目前libc的版本也都是属于ELF格式的；当然，习惯a.out格式的人如果有个ELF格式的libc与a.out格式的libc联合起来一起使用，是再好不过的事了。不可否认， C程序库的发展正以坚定的脚步迈向 ELF格式，除非你真的有很好的理由，否则应该放弃a.out格式。

A.4　GCC的安装与GCC的设置

A.4.1　GCC的版本

在外壳提示符号下键入gcc -v，屏幕上就会显示出你目前正在使用的 GCC的版本。同时也可确定你现在所用的是ELF格式或是a.out格式。在我的系统上，执行gcc -v的结果是：

```
$ gcc -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
```

上面的信息指出了几件重要的事情：

1) i486 这是指明你现在正在用的gcc是为486微处理器写的(你的电脑可能是386或是586)。这3种微处理器的芯片所编译而成的程序代码，彼此间是可以兼容使用的。差别之处是 486的程序代码在某些地方可加上 padding的功能，所以在486上面可以运行得比较快。对 386的计算机而言，执行程序的效能并不会有什么不良的影响，只不过程序代码变得稍稍的大了一些。

2) box 可以说没有什么用处。

3) linux 其实这是指linuxelf或是linuxaout。这一项会令人引起不必要的困惑，究竟是指哪一种会根据你所用的版本而异。

linux 若版本序号是2.7.0（或者更新）就是指ELF格式；否则，就是指a.out格式。

linuxaout 意指a.out格式。当linux的定义从a.out更换到ELF时，linuxaout就成为一个目标。因此，你不会看到任何新于2.7.0.的版本gcc有linuxaout。

linuxelf 已经过时了。通常这是指2.6.3版的gcc，而且这个版本也可以用来产生 ELF的可执行文件。要注意的是， gcc2.6.3版在产生ELF程序代码时会有错误，所以如果你目前用的恰好是这个版本，建议你赶快升级。

4) 版本的序号。

所以，总结起来，我的系统用的是2.7.2版的gcc，可以产生ELF格式的程序代码。

A.4.2 Gcc的安装目录

如果在安装gcc时没有仔细看着屏幕，或者你是从一个完整的发行系统里把 gcc单独提出来安装的话，那么，也许你会想知道到底这些东西装好后是在整个文件系统的哪些地方。几个重点目录如下：

1) /usr/lib/gcc-lib/target/version/（与子目录）

大部分的编译器就是在这个地方。在这里有可执行的程序，实际在做编译的工作；另外，还有一些特定版本的程序库与头文件等也会保存在此。

2) /usr/bin/gcc

指的是编译器的驱动程序，也就是你实际在命令行上执行的程序。这个目录可供各种版本的gcc使用，只要你用不同的编译器目录（如上所述）来安装就可以了。要知道内定的版本是哪一个，在外壳提示符号下键入 gcc -v。如果想强迫执行某个版本，就键入 gcc -V version。例如：

```
# gcc -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
# gcc -V 2.6.3 -v
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.6.3/specs
gcc driver version 2.7.2 executing gcc version 2.6.3
```

3) /usr/target/(bin|lib|include)/

如果你装了几种目标对象，例如a.out与elf，或者某种交叉编译器，那些属于非主流目标对象的程序库 —— binutils（as、ld等等），工具与头文件等都可以在这里找到。即使你只安装了一种gcc，还是可以在这里找到这些原本就是替它们准备的东西。如果不在这个目录中，那么就应该是在/usr/(bin|lib|include)中了。

4) /lib/、/usr/lib

与其他的相似的目录都是主流系统的程序库目录。许多的应用程序都会用到/lib/cpp，因此，

你也需要它，或者从/usr/lib/gcc-lib/target/version/ 目录里拷贝，或者使用符号链接指向那里。

A.4.3 头文件

假如把你自行安装在/usr/local/include目录底下的头文件排除在外的话， Linux还有下面目录下的另外3种主要的头文件：

/usr/include/

与其子目录底下的头文件，大部分都是由 H.J.Lu开发的libc套件所提供的。说“ 大部分 ”的原因是因为你可能有其他来源的头文件放在这里；尤其是，如果你现在用的是最新的 libc发行系统的话，那东西之多是人人为之咋舌的！

在内核源代码的发行系统内，应该有 /usr/include/linux与 /usr/include/asm （里头有这些文件：<linux/*.h> 与 <asm/*.h>）的符号链接。把源代码解压缩后，可能你也会发现，需要在内核的目录底下做make config的动作。很多的文件都会依赖<linux/autoconf.h>的帮忙，可是这个文件却有可能因版本不同而不存在。若干内核版本里， asm只是它自己的一个符号链接，仅仅在make config时才建立出来而已。

所以，当你在目录/usr/src/linux底下，解开内核的程序代码时，可以参照下面的命令：

```
$ cd /usr/src/linux
$ su
# make config
```

回答接下来的问题正不正确并不重要，除非你打算继续构造内核。

```
# cd /usr/include
# ln -s ../src/linux/include/linux .
# ln -s ../src/linux/include/asm .
```

诸如<float.h>、<limits.h>、<varargs.h>、<stdarg.h> 与<stddef.h>之类的文件，会随着不同的编译器版本而不同，属于你个人的文件，可以在 /usr/lib/gcc-lib/i486-box- linux/2.7.2/include/与其他相类似（相同）的目录名称的地方找到。

A.5 建立交叉编译器

假设你已经拿到gcc的源代码，通常你只要依循INSTALL文件的指示便可完成一切的设置。make后面再接configure --target=i486-linux --host=XXX on platform XXX，就能帮你完成操作。要注意的是，你会需要Linux内核的头文件；同时也需要建立交叉汇编器与交叉链接器，你可以在 URL:ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ 中找到它们。

A.6 移植程序与编译程序

A.6.1 gcc自行定义的符号

在执行gcc时，附加-v这个参数，就能找出你所用的这版 gcc自动帮你定义了什么符号。例如，我的计算机输出结果是：

```
$ echo 'main(){printf("hello world\n");}'| gcc -E -v -
Reading specs from /usr/lib/gcc-lib/i486-box-linux/2.7.2/specs
gcc version 2.7.2
 /usr/lib/gcc-lib/i486-box-linux/2.7.2/cpp -lang-c -v -undef
-D__GNUC__=2 -D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux
-D__ELF__ -D__unix__ -D__i386__ -D__linux__ -D__unix -D__i386
```

```
-D__linux -Asystem(unix) -Asystem(posix) -Acpu(i386)
-Amachine(i386) -D__i486__ -
```

假若你正在编写的程序代码会用到一些 Linux独有的特性，那么把那些无法移植的程序代码，以条件式编译的前置命令封括起来。如下所示：

```
#ifdef __linux__
/* ... funky stuff ... */
#endif /* linux */
```

用__linux__就可以完成任务；看仔细一点，不是 linux。尽管linux也有定义，毕竟，这个不是POSIX的标准。

A.6.2 调用编译程序

在命令行上执行gcc时，只要在它的后面加上 -On的选项，就能让gcc编译出最优化的计算机编码。这里的n是一个可以省略的小整数，不同版本的 gcc，n的意义与其作用都不一样，不过，典型的范围是从0变化到2，再升级到3。

gcc在其内部会将这些数字转换成一系列的 -f与-m的选项。执行gcc时带上标志-v与-Q，你就能很清楚的看出每一种等级的-O对应到哪些选项。好比说，就-O2来讲，我的gcc会显示：

```
enabled: -fdefer-pop -fcse-follow-jumps -fcse-skip-blocks
-fexpensive-optimizations
    -fthread-jumps -fpeephole -fforce-mem -ffunction-cse -finline
    -fcaller-saves -fpcc-struct-return -frerun-cse-after-loop
    -fcommon -fgnu-linker -m80387 -mhard-float -mno-soft-float
    -mno-386 -m486 -mieee-fp -mfp-ret-in-387
```

要是你用的最优化编码等级高于你的编译器所能支持的等级（例如 -O6），那么它的效果就跟你用你的编译器所能提供的最高等级的效果是一样的。

A.6.3 和特定的微处理器相关

有一些-m的标志十分有用，但是却无法根据各种等级的 -O来使用。这之中最重要的是 -m386和-m486这两种。它们用来告诉gcc该把正在编译的程序代码视作专为 386或是486计算机所写的代码。不论是用哪一种-m来编译程序代码，都可以在彼此的计算机上执行。但 -m486编译出来的代码会比较大，不过拿来在386的计算机上运行也不会比较慢就是了。

目前尚无-mpentium或是-m586的标志。Linux建议我们可以用 -m486 -malign-loops=2-malign-jumps=2 -malign-functions=2来得到最佳的486程序代码。

A.6.4 Internal compiler error: cc1 got fatal signal 11

Signal 11是指 SIGSEGV，或者“ segmentation violation ”。通常这是说gcc对自己所用的指标感到困惑，而且还试图把信息写入不属于它的内存里。所以，这可能是一个 gcc的错误。然而，大体而言， gcc是一个经过严密测试且可靠度良好的软件。它也使用了大量复杂的信息结构与惊人数量的指标。假如你无法再一次重复这个错误 (当你重新开始编译时，错误的信息并没有一直出现在同一个地方)那几乎可以确定是你的硬件(CPU、内存、主机板或是高速缓存)本身有问题。千万不要因为你的电脑可以通过开机程序的测试、或者 Windows可以运行得好、或者其他什么程序可以运行得好，就回过头来说这是 gcc的一个错误；你所做的这些开机测试动作，通常没有什么实际上的价值。

A.6.5 移植能力

1. BSD的用户 (有 bsd_ioctl、守护进程和 <sgtty.h>)

你可以使用 -I/usr/include/bsd来编译程序，同时使用 -lbsd来链接程序。例如：在你的Makefile文件内，把-I/usr/include/bsd加到CFLAGS那一行；把-lbsd加到LDFLAGS那一行。如果你确实需要BSD类型的信号，也不需要再加上-D__USE_BSD_SIGNAL了。那是因为当你用了-I/usr/include/bsd，并且包括了头文件<sig?nal.h>之后，建立时就会自动加入这些信号。

2. 丢失的信号(SIGBUS、SIGEMT、SIGIOT、SIGTRAP、SIGSYS 等)

Linux与POSIX是完全相容的。不过，有些信号并不是POSIX中定义的。

在POSIX.1中省略了SIGBUS、SIGEMT、SIGIOT、SIGTRAP与SIGSYS信号，那是因为它们的行为与实际运行的方式息息相关，而且也无法进行适当的分类。确认实际运行方式后，便可以发送这些信号，可是必须以文件形式说明它们是在什么样的环境下发送出来的，以及指出任何与它们的发展相关的限制。

想要修正这个问题，最简单也是最笨的方法就是用 SIGUNUSED重新定义这些信号。正确的方法应该是以条件式的编译#ifdef的形式来处理这些问题：

```
#ifdef SIGSYS
/* ... non-posix SIGSYS code here .... */
#endif
```

3. K & R代码

gcc是一个与ANSI相容的编译器；奇怪的是，目前大多数的程序代码都不符合 ANSI所定的标准。如果你喜欢用ANSI提供的标准来编写C程序，似乎除了加上-traditional的标志之外，就没有其他什么可以多谈的了

要注意的是，尽管你用了-traditional来改变语言的特性，它的效果也仅局限于gcc所能够接受的范围。例如，-traditional会打开-fwritable-strings，使得字符串常数移至数据内存空间内(从程序代码内存空间移出来，这个地方是不能任意写入的)。这样做会让程序代码的内存空间无形中增加了。

4. 预处理符号和函数原型的冲突

最常见的问题是Linux中有许多常用的函数都定义成宏存放在头文件内。此时若有相似的函数原型出现在程序代码内，预处理程序会拒绝进行语法分析的预处理。常见的函数有 atoi()与atol()。

5. sprintf()函数

在大部分的Unix系统上， sprintf(string, fmt,...)传回的是string的指针，然而， Linux（遵循ANSI）传回的却是放入string内的字符数目。

6. select()函数 —— 程序执行时会处于忙碌/等待的状态

很久以前， select()的计时参数还只是只读的。即使到了最近，操作联机帮助中仍然有下面这段的警告：

select()应该是根据修正时间的数值（如果有的话），再传回自原始计时开始后所剩余的时间。未来的版本可能会使这项功能实现。因此，就目前而言，若以为调用 select()之后，计时指针仍然不会被修正过，是不正确的。

函数select()传回的是扣除等待尚未到达的信息所耗费的时间后，其剩余的时间数值。如果在计时结束时，都没有信息传送进来，计时参数便会设为 0；如果接着还有select()函数，以同样的计时数据结构来调用，那么select()便会立刻结束。

若要修正这项问题，只要每次调用 select()前，都把计时数值放到计时数据结构内，这样就没有问题了。把下面的程序代码，

```
struct timeval timeout;
timeout.tv_sec = 1; timeout.tv_usec = 0;
while (some_condition)
 select(n,readfds,writefds,exceptfds,&timeout);
```

改成，

```
struct timeval timeout;
while (some_condition) {
      timeout.tv_sec = 1; timeout.tv_usec = 0;
      select(n,readfds,writefds,exceptfds,&timeout);
  }
```

这个问题，在有些版本的Mosaic里是相当著名的，只要有一次的等待， Mosaic就停止在那里了。Mosaic的屏幕右上角，有个圆圆的、会旋转的地球动画。那个球转得愈快，就表示信息从网络上传送过来的速率愈慢！

7. 产生中断的系统调用

当一个程序以Ctrl-Z中止、然后再重新执行时，或者有其他可以产生Ctrl-C中断信号的情况，如子程序的终结等，系统就会显示“ interrupted system call ”或者“ write: unknown error ”，或者诸如此类的信息。

比起一些旧版的Unix ， POSIX的系统检查信号的次数是多一点。而Linux可能会执行signal处理程序。

8. 系统调用的返回值

在下列系统调用的执行期间：

select()、pause()、connect()、accept()、read()（终端上）、套接字、管道或文件（/proc中）、write()（终端上）、套接字、管道或行式打印机、 open()（FIFO上）、PTY或串行线路、 ioctl()（终端上）、fcntl()（具有命令F_SETLKW）、wait4()、syslog()、任何TCP或NFS操作。

就其他的操作系统而言，你需要的可能就是下面这些系统调用了： creat()、close()、getmsg()、putmsg()、msgrcv()、msgsnd()、recv()、send()、wait()、waitpid()、wait3()、tcdrain()、sigpause()、semop() 。

在系统调用期间，若有一信号产生，处理程序就会被调用。当处理程序将控制权转移回系统调用时，它会侦测出已经产生中断，而且返回值会立刻设置成 -1，而errno设置成EINTR。程序并没有想到会发生这种事，所以就停止了。

有两种修正的方法可以选择：

1) 对每个你自行安装的信号处理程序，都须在 sigaction的标志加上SA_RESTART。例如，把下列的程序，

```
  signal (sig_nr, my_signal_handler);
```

改成：

```
  signal (sig_nr, my_signal_handler);
  { struct sigaction sa;
      sigaction (sig_nr, (struct sigaction *)0, &sa);
#ifdef SA_RESTART
      sa.sa_flags |= SA_RESTART;
#endif
#ifdef SA_INTERRUPT
```

```
        sa.sa_flags &= ~ SA_INTERRUPT;
#endif
        sigaction (sig_nr, &sa, (struct sigaction *)0);
    }
```

要注意的是，当这部分的更改大量应用到系统调用之后，调用 read()、write()、ioctl()、select()、 pause() 与 connect()时，你仍然得自行检查EINTR。如下所示：

2) 你自己明确地检查EINTR：

这里有两个针对read()与ioctl()的例子。

原始的程序使用read()：

```
int result;
while (len > 0) {
  result = read(fd,buffer,len);
  if (result < 0) break;
  buffer += result; len -= result;
}
```

修改成，

```
int result;
while (len > 0) {
  result = read(fd,buffer,len);
  if (result < 0) { if (errno != EINTR) break; }
  else { buffer += result; len -= result; }
}
```

原始的程序使用ioctl()：

```
int result;
result = ioctl(fd,cmd,addr);
```

修改成，

```
int result;
do { result = ioctl(fd,cmd,addr); }
while ((result == -1) && (errno == EINTR));
```

注意一点，有些版本的BSD Unix，其内定的行为是重新执行系统调用。若要让系统调用中断，得使用 SV_INTER?RUPT或SA_INTERRUPT标志。

9. 可以写入的字符串

gcc认为当用户打算让某个字符串当作常数来使用时，它就只是字符串常数而已。因此，这种字符串常数会保存在程序代码的内存区段内。这块区域可以交换到磁盘的 image文件上，避免耗掉交换区占用的内存空间，而且任何尝试写入的举动都会造成分页的错误。

对旧一点的程序而言，这可能会产生一个问题。例如，调用 mktemp()，传递的参数是字符串常数。mktemp()会试图在“适当的位置”重新写入它的参数。

修正的方法不外乎如下：

- 以-fwritable-strings编译，迫使gcc将此常数置放在数据内存空间内。
- 将侵犯地址的部分重新改写，配置一个不为常数的字符串，在调用前，先以 strcpy()将拷贝进去。

10. 为什么调用execl()会失败？

那是因为调用的方式不对。 execl的第一个参数是想要执行的程序名。第二个与后续的参数会变成所调用的程序的argv数组。记住：传统上， argv[0]是只有当程序没有带着参数执行时才会有的设置值。所以，你应该这样写：

```
execl("/bin/ls","ls",NULL);
```

而不是只有，

```
execl("/bin/ls", NULL);
```

执行程序而不带任何参数，可解释成是一种邀请，目的是把此程序的动态程序库独立的特性显示出来。至少，a.out是这样的。就ELF而言，事情就不是这样了。

A.7 除错与监管

A.7.1 预防

lint对Linux而言并没有很广泛的用途，主要是因为大部分的人都能满足于 gcc所提供的警告信息。可能最有用的就是 -Wall参数了。这个参数的用途是要求 gcc将所有的警告信息显现出来。

A.7.2 除错

怎样做才能将除错信息放到一个程序里？你需要添加 -g的参数来编译与链接程序，而且不可以用-fomit-frame-pointer参数。事实上，你不需要重新编译所有的程序，只需重新编译目前你正在除错的部分即可。

就a.out格式而言，共享程序库是以-fomit-frame-pointer编译而成，这个时候，gdb就变得没有用处了。链接时给定 -g的选项，应该就隐含着静态链接的意义了；这就是为什么要加 -g的原因了。

如果链接器链接失败，告诉你找不到 libg.a，那就是在/usr/lib/的目录底下少了 libg.a文件。libg.a是一个C语言使用的一个特殊的侦错程序库。一般在 libc的套件内就会提供libg.a；不然的话（新版是这样的），你可能需要拿libc的源代码自己设置了。不过，实际上你应该不需要才对。不管是什么目的，大部分的情况下，只需将 libg.a链接到/usr/lib/libc.a，你就能得到足够的信息了。

很多的GNU软件在编译链接时，都会设置 -g的选项；这样做会造成执行文件过大的问题（通常是静态的链接）。实际上，这并不是一个很好的做法。

如果程序本身有 autoconf，产生了 configure命令脚本，通常你就可以用 ./configure CFLAGS=或是./configure CFLAGS=-O2来关掉除错信息。否则，你得检查 Makefile了。当然，假如你用的是ELF格式，程序便会以动态的方式来链接，而不管是否有 -g的设置。因此，你可以把-g参数去掉。

1. 实用的软件

据了解，大部分人都是用gdb来除错的。可以从GNU archive sites7得到原始程序，或者到tsx-118得到可执行文件。xxgdb是一个X界面的除错程序，它是基于gdb的（也就是说你得先安装好 gdb程序，才能再安装 xxgdb程序）。xxgdb的源程序代码可以在 URL:ftp://ftp.x.org/con?trib/xxgdb-1.08.tar.gz 中找到。

另外，UPS除错程序已由 Rick Sladkey移植成功。UPS可以在X窗口环境下运行，不像xxgdb那样仅是gdb的X前端界面。这个除错程序有一大堆的优点，而且如果你得花时间去除一个错误很多的程序，建议你考虑使用 xxgdb。事先编译好的 Linux版与修正版的源代码可以在 URL:ftp://sun?site.unc.edu/pub/Linux/devel/debuggers/ 找到。而最初的原始程序则放在 URL:ftp://ftp.x.org/contrib/ups-2.45.2.tar.Z 中 。

另一个用来除错的工具 strace也是相当的有用。它可以显示出由程序所产生的系统调用，而且还拥有其他众多繁复的功能。如果你手边没有源代码的话， strace可以帮你找出有那些已编译进执行文件内路经名称，另外， strace可指导学习程序是怎么在电脑中执行的。最新的版本（目前是3.0.8）可在URL:ftp://ftp.std.com/pub/jrs/ 找到。

2. 守护程序

早期典型的守护程序是执行fork()，然后终止父程序。这样的做法使得除错的时间减短了。了解这点的最简单的方法就是替fork()设一个中断点。当程序停止时，强迫fork()传回0。

```
(gdb) list
1    #include <stdio.h>
2
3    main()
4    {
5      if(fork()==0) printf("child\n");
6      else printf("parent\n");
7    }
(gdb) break fork
Breakpoint 1 at 0x80003b8
(gdb) run
Starting program: /home/dan/src/hello/./fork
Breakpoint 1 at 0x400177c4
Breakpoint 1, 0x400177c4 in fork ()
(gdb) return 0
Make selected stack frame return now? (y or n) y
#0  0x80004a8 in main ()
      at fork.c:5
5      if(fork()==0) printf("child\n");
(gdb) next
Single stepping until exit from function fork,
which has no line number information.
child
7    }
```

3. 内核文件

当Linux开机时，通常状态会设置成不要产生内核文件。要是你那么喜欢内核文件的话，可以用外壳的builtin命令使其重新生效：就C-shell相容的外壳程序（如 tcsh）而言，会是下面这样：

```
% limit core unlimited
```

而类似Bourne shell的外壳（sh, bash, zsh, pdksh）则使用下面的语法：

```
$ ulimit -c unlimited
```

如果你想要有个多样化的内核文件名，那么你可以对你的内核程序做一点小小的更改。找一找fs/binfmt_aout.c与fs/binfmt_elf.c文件中与下列相符的程序段：

```
      memcpy(corefile,"core.",5);
#if 0
      memcpy(corefile+5,current->comm,sizeof(current->comm));
#else
      corefile[4] ='\0 ' ;
#endif
```

将0换成1。

A.7.3 监管

监管是用来检核一个程序中哪些部分是最常调用，或者执行的时间最久的方法。这对程序的最佳化与找出时间是如何浪费，是相当好的方式。你必须就你所要的时间信息的目的文件加上-p来编译，而且如果要让输出的文件有意义，你也会需要 gprof（来自binutils套件的命令）。参阅gprof的联机帮助，可得知其细节。

A.8 链接

我们称执行期间所发生的事为动态加载，这一主题会在下一节中谈到。你也会在别的地方看到我把动态加载描述成动态链接。换句话说，这一节所谈的，全部是指发生在编译结束后的链接。

A.8.1 共享程序库和静态程序库

建立程序的最后一个步骤便是链接，也就是将所有分散的小程序组合起来，看看是否遗漏了些什么。显然，有一些事情是很多程序都会想做的，例如打开文件。接着所有与打开文件有关的小程序就会将保存程序库的相关文件提供给你的程序使用。在一般的 Linux系统上，这些小程序可以在/lib与/usr/lib/目录底下找到。

当你用的是静态程序库时，链接器会找出程序所需的模块，然后实际将它们拷贝到执行文件内。然而，对共享程序库而言，就不是这样了。共享程序库会在执行文件内留下一个记号，指明当程序执行时，首先必须加载这个程序库。显然，共享程序库是试图使执行文件变得更小，等同于使用更少的内存与磁盘空间。 Linux内定的行为是链接共享程序库，只要Linux能找到这些共享程序库的话，就没什么问题。不然 ,Linux就会链接静态程序库了。如果你想要共享程序库的话，检查这些程序库（*.sa对于 a.out格式, *.so 对于ELF格式）是否在它们该在的地方，而且是可以读取的。

在Linux上，静态程序库会有类似 libname.a这样的名称；而共享程序库则称为libname.so.x.y.z，此处的x.y.z是指版本序号。共享程序库通常都会有链接符号指向静态程序库（很重要的）与相关联的.sa文件。标准的程序库会包含共享与静态程序库两种格式。

你可以用ldd（List Dynamic Dependencies）来查出某支程序需要哪些共享程序库。

```
$ ldd /usr/bin/lynx
    libncurses.so.1 => /usr/lib/libncurses.so.1.9.6
    libc.so.5 => /lib/libc.so.5.2.18
```

这是说在我的系统上， WWW浏览器lynx会依赖libc.so.5 (the C library)与libncurses.so.1（终端机屏幕的控制）的存在。若某个程序缺乏独立性， ldd就会说“ statically linked ”或者“ statically linked (ELF) ”。

A.8.2 sin() 在哪个程序库里

nm 程序库名称

此命令应该会列出程序库名称所指向的所有符号。这个指令可以应用在静态与共享程序库上。假设你想知道tcgetattr()是在哪儿定义的：你可以这样做，

```
$ nm libncurses.so.1 | grep tcget
U tcgetattr
```

U指出未定义 —— 也就是说ncurses程序库用到了tegetattr()，但是并没有定义它。你也可以

这样做：

```
$ nm libc.so.5 | grep tcget
00010fe8 T __tcgetattr
00010fe8 W tcgetattr
00068718 T tcgetpgrp
```

W说明了弱态，意指符号虽已定义，但可由不同程序库中的另一定义所替代。最简单的正常定义（像是tcgetpgrp）是由T所标示的。

标题所谈的问题，最简明的答案便是 libm.(so|a)了。所有定义在 <math.h>的函数都保留在maths程序库内。因此，当你用到其中任何一个函数时，都需要以 -lm的参数链接此程序库。

A.8.3 X文件

```
ld: Output file requires shared library `libfoo.so.1`
```

ld与其相类似的命令在搜索文件的方法上，会依据版本的差异而有所不同，但是你可以合理假设的唯一内定目录便是/usr/lib了。如果你希望它所在的程序库也列入搜索的行列中，那么你就必须以-L选项告知gcc或是ld。

要是你发现一点效果也没有，就赶紧察看此文件是不是还在原来的位置。就 a.out而言，以-lfoo参数来链接，会使得ld去寻找libfoo.sa；如果没有成功，就会换成寻找 libfoo.a。就ELF而言，ld会先找libfoo.so，然后是libfoo.a。libfoo.so通常是一个链接符号，链接至libfoo.so.x。

A.8.4 建立自己的程序库

1. 控制版本

与其他任何程序一样，程序库也有修正不完的错误。它们也可能产生出一些新的特点，更改目前存在的模块的功效，或者将旧的移除掉。这对正在使用它们的程序而言，可能会是一个大问题。如果有一个程序是根据那些旧的特点来执行的话，那怎么办？

所以，我们引进了程序库版本编号的观念。我们将程序库次要与主要的更改分类，同时规定次要的更改是不允许用到这程序库的旧程序发生中断的现象。你可以从程序库的文件名分辨出它的版本。

libfoo.so.1.2的主要版本是1，次要版本是2。次要版本的编号可能真有其事，也可能什么都没有。libc在这一点上用了修正程度的观念，而订出了像 libc.so.5.2.18这样的程序库名称。次要版本的编号内若是放一些字母、底线、或者任何可以显示的 ASCII字符，也是很合理的。

ELF与a.out格式最主要的差别之一就是在设置共享程序库这件事上；我们先看 ELF，因为它比较简单一些。

2. ELF

ELF（Executable and Linking Format）最初是由USL（UNIX System Laboratories）开发的二进位格式，目前正应用于Solaris与System V Release4上。由于ELF所具有的灵活性远远超过Linux过去所用的a.out格式，因此GCC与C程序库的发展人士于1995年决定改用ELF为Linux标准的二进位格式。

ELF是由SVR4所引进的新式改良目标文件格式。 ELF比COFF多出了不少功能。 ELF可由用户自行延伸。ELF视一个文件为节区，如串行般的组合；而且此串行可为任意的长度（而不是一固定大小的阵列）。这些节区与COFF的不一样，并不需要固定在某个地方，也不需要以某种顺序排列。如果用户希望捕捉到新的数据，便可以加入新的节区到目标文件内。 ELF也有一个更强有力的除错方式，称为 DWARF（Debugging With Attribute Record Format），但目前

Linux并不完全支持。DWARF DIE（Debugging Information Entries）的链接串行会在ELF内形成.debug的节区。DWARF DIE的每一个.debug节区并非一些少量且固定大小的信息记录的集合，而是一任意长度的串行组合，拥有复杂的属性，而且程序的数据会以有范围限制的树状数据结构写出来。DIE所能捕捉到的大量信息是COFF的.debug节区无法比拟的。

ELF文件是从SVR4（Solaris 2.0 ）ELF存取程序库（ELF access library）内存取的。此程序库可为ELF提供一简便快速的界面。使用ELF存取程序库最主要的好处之一便是，你不再需要去察看一个ELF文件的qua了。UNIX文件以Elf的型式来存取；调用elf_open()之后，从此时开始，你只需调用elf_foobar()来处理文件的某一部分即可，并不需要把文件实际在磁盘上的镜像搞得一团乱。

3. ELF共享程序库

如果想让libfoo.so成为共享程序库，基本的步骤如下：

```
$ gcc -fPIC -c *.c
$ gcc -shared -Wl,-soname,libfoo.so.1 -o libfoo.so.1.0 *.o
$ ln -s libfoo.so.1.0 libfoo.so.1
$ ln -s libfoo.so.1 libfoo.so
$ LD_LIBRARY_PATH=`pwd`:$LD_LIBRARY_PATH ; export LD_LIBRARY_PATH
```

这会产生一个名为libfoo.so.1.0的共享程序库，以及给予ld适当的链接（libfoo.so）还有使得动态加载程序（dynamic loader）能找到它（libfoo.so.1）。为了进行测试，我们将目前的目录加到LD_LIBRARY_PATH里。

当你的程序库制做成功时，别忘了把它移到/usr/local/lib的目录底下，并且重新设置正确的链接路径。libfoo.so.1与libfoo.so.1.0的链接会由ldconfig不断地更新。就大部分的系统来说，ldconfig会在开机过程中执行。libfoo.so的链接必须由手动方式更新。如果你对程序库所有组成（如头文件等）的升级，总是抱持着认真的态度，那么最简单的方法就是让 libfoo.so 符号链接到 libfoo.so.1；这样一来，ldconfig便会替你同时保留最新的链接。要是你没有这么做，你自行设置的东西就会在数日后有千奇百怪的问题出现。

```
$ su
# cp libfoo.so.1.0 /usr/local/lib
# /sbin/ldconfig
# ( cd /usr/local/lib ; ln -s libfoo.so.1 libfoo.so )
```

4. 版本编号、soname与符号链接

每一个程序库都有一个soname。当链接器发现它正在搜索的程序库中有这样的一个名称，链接器便会将soname箝入链接中的二进位文件内，而不是它正在运行的实际的文件名。在程序执行期间，动态加载程序会搜索拥有soname这样的文件名的文件，而不是程序库的文件名。因此，一个名为libfoo.so的程序库，就可以有一个libbar.so的soname了。而且所有链接到libbar.so的程序，当程序开始执行时，会寻找的便是libbar.so了。

这听起来好像一点意义也没有，但是这一点，对于了解多个不同版本的同一个程序库是如何在单一系统上共存却是十分关键。Linux程序库标准的命名方式，比如说是libfoo.so.1.2，而且给这个程序库一个libfoo.so.1的soname。如果此程序库是加到标准程序库的目录底下（ e.g. /usr/lib），ldconfig会建立符号链接libfoo.so.1 -> libfoo.so.1.2，使其正确的镜像能于执行期间找到。你也需要链接libfoo.so -> libfoo.so.1，使ld能于链接期间找到正确的soname。

所以，当你修正程序库内的错误时，或是是添加了新的函数进去时（任何不会对现存的程序造成不利的影响的改变），你会重建此程序库，保留原本已有的 soname，然后更改程序库文件名。当你对程序库的更改会使得现有的程序中断，那么你只需增加soname中的编号。此例中，

称新版本为libfoo.so.2.0，而soname变成libfoo.so.2。紧接着，再将libfoo.so的链接转向新的版本；至此，一切又恢复了正常。

其实你不需要以此种方式来为程序库命名，不过这的确是个好的传统。 ELF赋予你在程序库命名上的灵活性，会使得人不清楚状况；有这样的灵活性在，也并不表示你就得去用它。

假设你发现有个惯例：程序库主要的升级会破坏相容性，而次要的升级则可能不会。那么以下面的方式来链接，所有的一切就都会相安无事了。

```
gcc -shared -Wl,-soname,libfoo.so.major -o libfoo.so.major.minor
```

5. 文件配置

a.out(DLL)的共享程序库包含两个真正的文件与一个链接符号。就 foo这个用于整份文件做为样例的程序库而言，这些文件会是libfoo.sa与libfoo.so.1.2；链接符号会是libfoo.so.1，而且会指向libfoo.so.1.2。

在编译时，ld会寻找libfoo.sa。这是程序库的stub文件。而且含有所有执行期间链接所需的exported的资料与指向函数的指针。

执行期间，动态加载程序会寻找libfoo.so.1。这仅仅是一个符号链接，而不是真正的文件。故程序库可更新成较新的且已修正错误的版本，而不会损毁任何此时正在使用此程序库的应用程序。在新版(比如说libfoo.so.1.3)已完整呈现时， ldconfig会以一极微小的操作，将链接指向新的版本，使得任何原本使用旧版的程序不会有任何得冲突。

DLL程序库通常会比它们的静态副本要大得多。它们是以洞的形式来保留空间以便日后的扩充。这种洞可以不占用任何的磁盘空间。一个简单的 cp调用，或者使用makehole程序，就可以达到这样效果。因为它们的位址是固定在同一位置上的，所以在建立程序库后，你可以把它们拿掉。不过，千万不要试着拿掉ELF的程序库。

6.“ libc-lite ”

libc-lite是简单的libc版本。可用来存放在磁盘上，也可以结束大部分低级的 UNIX任务。它没有包含curses, dbm, termcap等等的程序代码。如果你的/lib/libc.so.4是链接到一个lite的libc，那么建议你以完整的版本取代它。

A.9 动态加载

A.9.1 基本概念

Linux有共享程序库，所以有一些照惯例是在链接时期便该完成的工作，必须延迟到加载时期才能完成。

A.9.2 控制动态加载器的运作

有一组环境变量会让动态加载器有所反应。大部分的环境变量对 ldd的用途要比对一般users的还要多。而且可以很方便地设置成由 ldd配合各种参数来执行。这些变量包括 LD_BIND_NOW 。正常来讲，函数在调用之前是不会让程序寻找的。设置这个标志会使得程序库一加载，所有的寻找便会发生，同时也造成起始的时间较慢。当你想测试程序，确定所有的链接都没有问题时，这项标志就变得很有用。

LD_PRELOAD可以设置一个文件，使其具有覆盖函数定义的能力。例如，如果你要测试内存分配的策略，而且还想置换 malloc，那么你可以写好准备替换的副程序，并把它编译成mallolc。然后：

```
$ LD_PRELOAD=malloc.o; export LD_PRELOAD
$ some_test_program
```

LD_ELF_PRELOAD 与 LD_AOUT_PRELOAD很类似，但是仅适用于正确的二进位类型。如果设置了 LD_something_PRELOAD 与LD_PRELOAD，比较明确的那一个会被用到。

LD_LIBRARY_PATH是一连串以分号隔离的目录名称，用来搜索共享程序库。对 ld而言，并没有任何的影响；这项只有在执行期间才有影响。另外，对执行 setuid与set gid的程序而言，这一项是无效的。而LD_ELF_LIBRARY_PATH与LD_AOUT_LIBRARY_PATH这两种标志可根据各别的二进位型式分别导向不同的搜索路径。一般正常的运作下，不应该会用到LD_LIBRARY_PATH；把需要搜索的目录加到/etc/ld.so.conf/里；然后重新执行ldconfig。

LD_NOWARN 仅适用于 a.out。一旦设置了这一项（ LD_NOWARN=true; export LD_NOWARN），它会告诉加载器必须处理致命错误（像是次要版本不相容等）的警告。LD_WARN仅适用于ELF。设置这一项时，它会将致命信息“ Can ’ t find library ”转换成警告信息。对正常的操作而言，这并没有多大的用处，可是对 ldd就很重要了。

LD_TRACE_LOADED_OBJECTS仅适用于ELF。而且会使得程序以为它们是由ldd执行的：

```
$ LD_TRACE_LOADED_OBJECTS=true /usr/bin/lynx
        libncurses.so.1 => /usr/lib/libncurses.so.1.9.6
        libc.so.5 => /lib/libc.so.5.2.18
```

A.9.3 以动态加载编写程序

如果你很熟悉Solaris 2.x所支持的动态加载功能的话，你会发现Linux在这点上与其非常相近。这一部分在H.J.Lu的ELF程序设计文件内与dlopen(3)的操作联机帮助（可以在ld.so的套件上找到）上有广泛的讨论。下面是一个简单样例(以-ldl链接)：

```
#include <dlfcn.h>
#include <stdio.h>
main()
{
  void *libc;
  void (*printf_call)();
  if(libc=dlopen("/lib/libc.so.5",RTLD_LAZY))
  {
      printf_call=dlsym(libc,"printf");
      (*printf_call)("hello, world\n");
  }
}
```