The Linux GCC HOWTO中譯版V0.2

作者: Daniel Barlow <dan@detached.demon.co.uk>
譯者: 陳建勳(Frank J.S. Chen)<frank63@ms5.hinet.net>
v1.17, 28 February 1996

本文闡述安裝GNU C編譯器和程式館的方法，同時概觀地說明程式的編譯、連結、執行、除錯的過程以及可能面臨的諸多問題。寫作的參考資料泰半來自於Mitch D'Souza先生所收集的GCC-FAQ；而另一個來源是ELF-HOWTO。此份HOWTO可以說已代替了GCC-FAQ，而且即將要永久替代ELF-HOWTO了。此乃GCC-HOWTO第一份公開發行的版本（不須理會版本序號；那是RCS的傑作），有任何指正與建議的，本人都很歡迎。

目錄

1 行遠必自邇！

1.1 譯者的話

- 這份譯文為Linux document projects(*LDP*)中文翻譯計畫系列之一。目前之網址為 *http://www.linux.org.tw/CLDP/*，歡迎各位網友踴躍投入此一計畫。
- 我並沒有完全按照原文逐字翻譯。為了力求譯文通暢可讀，我會稍稍的重組一部份的文字，加油添醋，或是精簡原文；這樣做的話，可以彌補中英文間語法結構的差異性，且語氣可以貫通無礙。
- 一些關鍵字與專業詞彙等，會附加上原文單字。
- 遇有轉譯困難，唯原文常見的字彙如bugs、shadow password、padding、image之類的，則保留原文不變。若閣下對這些字彙有適當譯辭的，請不吝指教。
- 內文中若遇有"[譯者註:**]"之標記，則為本人額外之註解。
- 對這篇譯文有任何建議與疑問的，請email至*frank63@ms5.hinet.net*。
- WWW Home Page： *http://linux.ntcic.edu.tw/~jsfrank/*。
- 此中譯文件之翻譯權已取得英籍之原作者Daniel Barlow 先生之同意；另，陳建勳先生保有此份中譯版文件所有的權利，你可以任意的拷貝，以各種媒體散佈這份中譯文件，唯此節補充說明需原封不動附上，且不可任意更動譯文。
- v0.1版的譯文相當粗糙，連文句的語法結構都嫌太過於鬆散，v0.2版針對中文的用字習慣來修正，並將上一版譯的不妥當的地方修正過來，例如either這個字，英國人常把either當名詞用，指兩者中任意一個；這份HOWTO就充份反應出這個用字習性，跟美語有基本上的差異。

- 文中有幾句話沒有譯出來，一方面是看不懂，另方面是直譯也譯不出來，所以只好保留原文了，要是閣下有新的領悟的，無論如何請告訴我。
- v0.1版翻譯起始日期為：11/7/97；截止日期為：11/19/97。
- v0.2版修正起始日期：5/13/98；截止日期為：6/3/98。

1.2 動與靜

目前Linux的發展正波濤洶湧的進行著。簡單一點講，Linux有兩種執行檔的格式可用，取決於你的系統是怎麼整合起來的；你的Linux應該是其中一種吧！閱讀這份文件，可以幫助你釐清執行檔的類別。

要如何區別呢？執行公用程式‘file’（例如，`file /bin/bash`）就對了。就ELF的程式而言，螢幕上顯示出來的訊息會含有ELF的字眼；如果說是a.out的，訊息內會箱有 `Linux/i386`的字樣。

ELF與a.out格式的差異之處會在後續的章節中討論（很廣泛喔！）。ELF是比較新的格式，一般而言，接受的程度較佳。

1.3 作者的私語

版權說明與合法的行逕規定，就擺在這份文件的尾端。除此之外，我還有一些不得不提醒你的話要講。就算你閒著沒事幹，也不要在Usenet上丟一些呆瓜問的問題；還有，不要老以為自己C語言的功力深厚，專門發表一些不是bugs的bugs出來丟人現眼，這不就等於告訴別人你不學無術，在關老爺面前耍大刀了嗎？所以說自以為是的英雄主義是得不償失的。

1.4 印刷與排版

如果你現在讀的是Postscript、dvi或是html格式，那麼你所看到的字型變化就會比只讀純文字格式的人多一些。特別是檔案名稱、命令、命令的輸出與摘錄出來的原始碼等，統統都是打字機的字型。這樣做的話，對於某些需要強調的‘變數’還有那些沒有固定結果的範例而言，就可以達到強調的效果了。

讀這份文件的同時，你也會得到一套蠻有用的索引。假若是dvi、 postscript之類的版本，索引的數字就是章節(section)的編號；如果是HTML的話，這些數字會按順序排列，你可以用滑鼠左鍵去連結相對的索引內容；如果你看的是純文字版本的話, 數字就只是數字，沒別的含意；建議你趕快昇級為妙哩！

我用的shell是Bourne shell（不是C shell），舉的例子自然是Bourne shell的語法。如果你用的是C shell的話，設定環境變數的語法會像下面這樣：

```
% setenv FOO bar
```

要是用Bourne shell的話，我會這樣子寫：

```
$ FOO=bar; export FOO
```

如果提示符號顯示的是井字符號#而不是錢字符號 $的話，很有可能這個命令是只適用於root而已。當然啦，要是你試了這些範例，結果弄得你的系統發生災變，我可是一點責任也不會負的喔！祝好運！:-)

11/8/97譯. 5/13/98修訂

2 東東在哪兒？

2.1 GCC-HOWTO在哪兒？

這份文件是Linux HOWTO系列之一，換句話說，你可以在所有存放Linux HOWTO文件的網站上面找到她的芳蹤，例如 <http://sunsite.unc.edu/pub/linux/docs/HOWTO/>。HTML的版本（可能會是較新的版本）可以從 <http://ftp.linux.org.uk/~barlow/howto/gcc-howto.html>上面抓下來。

2.2 GCC相關的資料又在哪兒？

標準的gcc說明文件是隨附在發行的原始碼(source distribution)內（往下看就有了！），裡頭有textinfo與.info兩種檔案。要是你的網路連接速率夠快，或是有一片cdrom，不然，有高度的耐心也成，你可以自己把它untar，然後再把相對應的位元一一拷貝到/usr/info的目錄底下。假如你的條件與上述的不符，不妨到 *tsx-11* <ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/>站上去參觀參觀。不過，我想，沒有必要老是惦記著最新的版本吧！

libc的文件說明有兩種來源。一種是GNU libc，以.info的格式儲存，除了stdio之外，其餘Linux libc的說明都相當的詳盡精確。另一種可以在Linux的archive*manpages* <ftp://sunsite.unc.edu/pub/Linux/docs/>上找到系統呼叫（system call）（第2節）與libc函數（function）（第3節）的文件說明。

2.3 GCC

解答有二:

(a)你可以在 <ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/>的網站上找到 正式的Linux GCC發行系統(distribution)，而且是已經編譯好的可執行檔。當我在寫這份文件時，2.7.2(gcc-2.7.2.bin.tar.gz)是最新的版本。

(b)自由軟體基金會（Free Software Foundation）所發佈的GCC最新原始碼可以從網站 *GNU archives* <ftp://prep.ai.mit.edu/pub/gnu/>上取得。沒有必要非得與上述的版本一致才行，不過這個版本的確是目前最新的。Linux GCC的維護網友(maintainers)讓你可以很輕鬆的自行編譯這個最新的版本。configure命令稿(script)會幫你自動設定好所有該做的事情。建議你有空不妨到 *tsx-11* <ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/>看看，說不定會有修正的版本是你會想要用的。

如果想要編寫出一些有用的軟體（不是我囉唆，還是有不少沒啥用途的軟體在網路上四處流竄。），下面這一小節所談的也是你需要的：

2.4 C程式館與標頭檔

該選哪一套程式館是取決於(i)你的系統是ELF的或是a.out的；(ii)你希望你的系統變成哪一種？如果你是從libc 4昇級到libc 5，那麼給你一個良心的建議，先去看看ELF-HOWTO再說。你一定會問，在ELF文件的哪兒呢？嘿！嘿！不偏不倚，就差不多跟這份文件相同的位置。網站 *tsx-11* <ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/>上面可以找到你想要的。

`libc-5.2.18.bin.tar.gz`

— ELF共享程式館（ELF shared library images），靜態程式館（static libraries）與標頭檔（針對C語言與數學程式館）。

`libc-5.2.18.tar.gz`

—libc-5.2.18.bin.tar.gz的原始碼。這個檔案你也需要，因爲`.bin.`套件(package)含有必需的標頭檔。如果此時你正猶豫不決，不曉得是老身親自下海，動手編譯C程式庫比較好；還是直接用人家編譯好的二進位檔(binaries)就可以了。有這種困擾的人，來，看我的嘴形：用人家編譯好的二進位檔不就解決了嘛！只有在你想要NYS或是shadow password的情況下，才需要自己的手來推動搖籃。

`libc-4.7.5.bin.tar.gz`

— 這個檔案是a.out的共享程式庫(shared library images)與靜態程式庫，用途是爲了與前述的libc 5套件共存共榮而設計的，不過除非你想要繼續使用a.out的程式或是繼續發展a.out的程式，不然的話，是不需要它的。

2.5 有關聯的工具 (as, ld, ar, strings etc)

到目前爲止，與之前所談的都一樣，從網站*tsx-11* `<ftp://tsx-11.mit.edu:/pub/linux/packages/GCC/>`上，就可以找到這些工具程式。目前的版本是`binutils-2.6.0.2.bin.tar.gz`。

需要注意的是binutils只適用於ELF，而且目前libc的版本也都是屬於ELF的；當然啦，習慣a.out的人如果有個ELF的libc與a.out的libc聯合起來一起使用，這對他們來講是再好不過的美事了。不可否認的，C程式館的發展正以堅決的腳步邁向ELF的格式，除非你眞的有很好的理由，需要a.out的東東，不然啊，大家都會鼓勵你勇於突破，趁早加入銳不可擋的大潮流。

11/9/97譯

3 GCC的安裝與GCC的設定

3.1 GCC的版本

在shell的提示符號下鍵入`gcc -v`，螢幕上就會顯示出你目前正在使用的GCC的版本。同時這也是一個相當可靠的方法，可以確定你現在所用的是ELF或是a.out。在我的系統上，執行gcc -v的結果是：

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

上面的訊息指出了幾件重要的事情：

- `i486` 這是指明你現在正在用的gcc是爲了486的微處理器而寫的—你的電腦可能是386或是586。這3種微處理器的晶片所編譯而成的程式碼，彼此間是可以相容使用的。差別之處是486的程式碼在某些地方有加上padding的功能，所以可以在486上面跑得比較快。這對386的機器而言，執行程式的效能並不會有什麼不良的影響，只不過眞的會讓程式碼變得稍稍的大了一些。
- `box` 這可以說一點也不重要；不過也可能另有所指（像是`slackware`或者是`debian`），或者根本什麼也不是（所以囉！完整的目錄名稱是`i486-linux`）。假如你是實務派的佼佼者，親自動手建立屬於自己的gcc，那麼你可以在建立的過程中設定這一項，以裝點門面。就像我做的一樣`:-)`。
- `linux` 其實這是指`linuxelf`或是`linuxaout`。這一項會令人引起不必要的困惑，究竟是指哪一種會根據你所用的版本而異。

  - `linux` 意指ELF若版本序號是2.7.0.（或是更新的版本）；否則的話，就是a.out的了。
  - `linuxaout` 意指a.out的格式。當`linux`的定義從a.out更換到ELF時，linuxaout就會順水推舟，搖身一變，成了一個目標物。因此，你不會看到任何版本新於2.7.0.的gcc有linuxaout的。
  - `linuxelf` 已經過時了。通常這是指2.6.3版的gcc，而且這個版本也可以用來產生ELF的可執行檔。要注意的是，gcc 2.6.3版在產生ELF程式碼時會有bugs，所以如果你目前用的恰好是這個版本，建議你趕快昇級。

- `2.7.2` 版本的序號。

所以，總結起來，我有2.7.2版的gcc，可以產生ELF格式的程式碼。就這麼簡單，驚訝吧！eh？

3.2 東東裝好後都到哪兒去了?

如果安裝gcc時沒有仔細的看著螢幕，或者你是從一個完整的發行系統裡把gcc單獨抓出來安裝的話，那麼也許你會想知道到底這些東東裝好後是住在整個檔案系統的那些地方。幾個重點如下：

- `/usr/lib/gcc-lib/`*target*`/`*version*`/`（與子目錄）大部份的編譯器就是住在這個地方的。在這兒有可執行的程式，實際在做編譯的工作；另外，還有一些特定版本的程式庫與標頭檔等也會儲存在此。
- `/usr/bin/gcc` 指的是編譯器的驅動程式—也就是你實際在命令列(command line)上執行的程式。這個目錄可供各種版本的gcc使用，只要你用不同的編譯器目錄（如上所述）來安裝就可以了。要知道內定的版本是那一個，在shell提示符號下打`gcc -v`。要是想強迫執行某個版本，就換打`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
```

- `/usr/`*target*`/(bin|lib|include)/` 如果你裝了數種的目標物件，例如a.out與elf，或是某一種的交叉編譯器(cross-compiler)等等，那些屬於非主流目標物件（non-native target(s)）的程式庫，binutils（`as`、`ld`等等）工具與標頭檔等都可以在這兒找到。即使你只安裝了一種gcc，還是可以在這兒找到這些原本就是替它們準備的東東。如果不是在這兒，那麼就應該是在`/usr/(bin|lib|include)`了。
- `/lib/`，`/usr/lib` 與其它的目錄等都是主流系統（native-system）的程式館目錄。許多的應用程式都會用到`/lib/cpp`，因此你也需要它—作法上，不是從`/usr/lib/gcc-lib/`*target*`/`*version*`/` 目錄裡拷貝，就是弄個符號連結（symlink）指向那兒。 [譯者註:所謂的native，是指目前你的系統是以a.out或elf的格式爲主，或者內定的gcc是哪一種版本等等。native的意思是‘本土的’、‘本國的’與‘天生的’……等等；當你拿到一片CD-ROM重頭至尾將Linux安裝完成，讓Linux出生，成爲你個人特色濃厚的作業平台後，如果再加裝一些不一樣的目標物件，自然就有‘本土’與‘外省’（無關政治），‘本國’與‘外國’、‘天生’與‘人爲’等等的區別，同時也含有內定（default）的意思在。假若再附加上你個人的價值觀判斷和喜好，我想用主流（native）與非主流（non-native）來翻譯應該還算恰當。]

3.3 標頭檔ㄋㄟ?標頭檔ㄋㄟ?

假如把你自行安裝在/usr/local/include目錄底下的標頭檔排除在外的話，Linux還有另外3種主要的標頭檔：

- /usr/include/與其子目錄底下的標頭檔，大部份都是由H.J.Lu發展的libc套件(libc binary package)所提供的。我會只說‘大部份’的原因，是因為你可能有其它來源的標頭檔（像是curses與dbm程式庫等等）擺在這兒；尤其是，如果你現在用的是最新的libc發行系統的話（新版本不像舊版那樣，已經不再支援curses或dbm了。），那東東之多是人人為之咋舌的！ [譯者註:libc binary package意指以二進位形式(machine code)儲存之套件，並非原始碼（text），若要以中文全稱譯出，則成‘libc二進位檔套件’，似有聱牙之嫌，故略去binary，以libc套件通稱。]

- 在核心原始碼的發行系統內(kernel source distribution)，/usr/include/linux 與 /usr/include/asm（裡頭有這些檔案：<linux/*.h> 與 <asm/*.h>）應該有符號連結（symbolic links）可以連結至目錄linux/include/linux 與 linux/include/asm。如果你有鴻鵠之志的話，安裝這些東東後，就不應該只是拿來編譯核心（kernel）而已。

  把原始碼解壓縮（unpacking）後，可能你也會發現，需要在核心的目錄（kernel directory）底下做make config的動作。很多的檔案都會依賴<linux/autoconf.h>的幫忙，可是這個檔案卻有可能因版本不同而不存在。若干核心版本裡，asm就只是它自己的一個符號連結，僅僅是在make config時才建立出來而已。 [譯者註:原文提及autoconf.h時是‘Many files depend on <linux/autoconf.h>,which otherwise may not exist,*’。此處之otherwise之詞性應為形容詞(adj)，指‘另一情況’、‘另一種’、‘不同的’之意，將原文形容詞子句拆開來應為：(i) Many files depend on <linux/autoconf.h>. (ii)<linux/autoconf.h> *of other condition* may not exist. 與下一句互相比對，此處應同指在不同版本之情況下。]

  所以，當你在目錄/usr/src/linux底下，解開核心的程式碼時，就照著下面指示的做吧！

```
$ cd /usr/src/linux
$ su
# make config
(回答接下來的問題。通常回答得正不正確並不重要，除非你打算繼續築造核心。)
# cd /usr/include
# ln -s ../src/linux/include/linux .
# ln -s ../src/linux/include/asm .
```

- 諸如 <float.h>、<limits.h>、<varargs.h>、<stdarg.h> 與 <stddef.h>之類的檔案，會隨著不同的編譯器版本而異，屬於你‘個人的’檔案，可以在/usr/lib/gcc-lib/i486-box-linux/2.7.2/include/與其它相類似（相同）的目錄名稱的地方找到。

  11/11/97譯 5/14/98修正

3.4 建立交叉編譯器(Building cross compilers)

3.4.1 將Linux當作標的作業平台(target platform)

假設你已經拿到gcc的原始碼，通常你只要依循INSTALL檔的指示便可完成一切的設定。 make後面再接configure -target=i486-linux -host=XXX on platform XXX，就能幫你變把戲了。要注意的是，你會需要Linux還有核心的標頭檔；同時也需要建立交叉組譯器(cross assembler)與交叉連結器(cross linker)，來源是 <ftp://tsx-11.mit.edu/pub/linux/packages/GCC/>。

3.4.2 Linux當成原始作業平台(source platform)而MSDOS作爲標的作業平台

Ugh。很明顯的，這個大概需要用到套件“emx”或是延伸套件“go”。請自行去 <ftp://sunsite.unc.edu/pub/Linux/devel/msdos>看看。我並沒有測試過這些個東西，所以沒有辦法保證什麼。

4 移植程式與編譯程式

4.1 gcc自行定義的符號

只要執行gcc時，附加-v這個參數，就能找出你所用的這版gcc，自動幫你定義了什麼符號。例如，我的機器看起來會像這樣：

```
$ 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__ -
```

假若你正在寫的程式碼會用到一些Linux獨有的特性，那麼把那些無法移植的程式碼，以條件式編譯的前置命令封括起來，可是個不錯的主意呢！如下所示：

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

用__linux__就可以達成目的；看仔細一點，不是linux喔。儘管linux也有定義，畢竟，這個仍然不是POSIX的標準。

4.2 線上求助說明

gcc編譯器參數的說明文件是gcc info page（在Emacs內，按下C-h i，然後選‘gcc’的選項）。要是弄不出來，不是賣你CD-ROM的人沒把這個東東壓給你，不然就是你現在用的是舊版的。遇到這種情況，最好的方法是移動尊臀到archive <ftp://prep.ai.mit.edu/pub/gnu>或是它的mirrors站台，去把gcc的原始檔案抓回家，重新烹飪一番。

gcc manual page（gcc.1） 可以說是已經過時了，要是你吃飽了撐著沒事幹硬是想看，它就會告訴你說別無聊了。

4.2.1 旗正飄飄

在命令列上執行gcc時，只要在它的屁股後面加上-On的選項，就能讓gcc乖乖的替你生出最佳編碼的機器碼。這裡的n是一個可有可無的小整數，不同版本的gcc，n的意義與其正確的功效都不一樣，不過，典型的範圍是從0（不要雞婆，我不要最佳編碼。）變化到2（最佳編碼要多一點。），再昇級到3（最佳編碼要再多一點，多一點）。

gcc在其內部會將這些數字轉譯成一系列的`-f`與`-m`的選項。執行gcc時帶上旗號`-v`與`-Q`，你就能很清楚的看出每一種等級的`-O`是對應到那些選項。好比說，就`-O2`來講，我的gcc告訴會我說：

```
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
```

要是你用的最佳編碼等級高於你的編譯器所能支援的（e.g. `-O6`），那麼它的效果就跟你用你的編譯器所能提供的最高等級的效果是一樣的。說實在的，發行出去的gcc程式碼，用在編譯時竟是如此處理這等問題，眞的不是什麼好的構想。日後若是有更進步的最佳編碼方法具體整合到新的版本裡，而你（或是你的users）還是試著這樣做的話，可能就會發現gcc會中斷你的程式了。

從gcc 2.7.0昇級到2.7.2的users應該注意一點，使用`-O2`時會有一個bug。更糟糕的是，強度折減參數(strength reduction)居然沒有用！要是你喜歡重新編譯gcc的話，是有那麼一個修正的版本可以更正這項錯誤；不然的話，一定要確定每次編譯時都有加上`-fno-strength-reduce`喔！

11/12/97譯

有個性的微處理器 一些`-m`的旗號十分有用處，但是卻無法藉由各種等級的`-O`打開來使用。這之中最重要的有是`-m386`和`-m486`這兩種，用來告訴gcc該把正在編譯的程式碼視作專爲386或是486機器所寫的。不論是用哪一種`-m`來編譯程式碼，都可以在彼此的機器上執行，-m486編譯出來的碼會比較大，不過拿來在386的機器上跑也不會比較慢就是了。

目前尚無`-mpentium`或是`-m586`的旗號。Linus建議我們可以用`-m486 -malign-loops=2 -malign-jumps=2 -malign-functions=2`來得到最佳編碼的486程式碼，這樣做正好就可以避免alignment（Pentium並不需要）有過大的gaps發生。Michael Meissner說：

> 我的第六感告訴我，`-mno-strength-reduce`（嘿！要曉得我可不是在談強度折減參數的bug呀，那已經是另外一個爭論的戰場了。）一樣也可以在x86的機器上產生較快的程式碼，這是因爲x86的機器對暫存器有著不可磨滅的饑渴在，而且GCC's method of grouping registers into spill registers vs. other registers doesn't help either。傳統上，強度折減的結果會使得編譯器去利用加法暫存器以加法運算來取代乘法運算。事實上，我在懷疑`-fcaller-saves`可能也只是個漏洞也說不定。
>
> 而我的第七感則再度的告訴我說，`-fomit-frame-pointer`可能會也可能不會有任何的賺頭。從這點來看，就是意謂著有另一個暫存器可以用來處理記憶體分配的問題。另方面，若純粹從x86的機器在轉換它的指令集成爲機器碼的方法上來看，便意謂著堆疊所用到的記憶體空間要比frame所用到的還要來得多；換句話說，Icache對程式碼而言並沒有實質上的幫助，若是閣下用了`-fomit-frame-pointer`的話，同時也是告訴編譯器在每次呼叫函數之後，就必須修正堆疊的指標；然而，就frame來講，若呼叫的次數不多的話，則允許堆疊暫時堆積起來。

有關這方面主題的最後一段話仍是來自於Linus：

> 要注意的是，如果你想要得到最佳狀況的執行效能，可千萬別相信我的話。無論如何，一定要進行測試。gcc編譯器還有許多的參數可用，其中可能就有一種最特別的組合，可以給你最佳編碼的結果。

11/14/97譯 5/15/98修正

4.2.2 Internal compiler error: cc1 got fatal signal 11

Signal 11是指 SIGSEGV，或者 ‘segmentation violation’。通常這是指 說gcc對自己所用的指標感到困惑，而且還嘗試著把資料寫入不屬於它的記憶體裡。所以，這可能是一個gcc的bug。 然而，大體而言，gcc是一支經過嚴密測試且可靠度良好的軟體佳作。它也用了大量複雜的資料結構與驚人的指標數量。簡言之，若是要評選本世紀最挑惕與最一絲不苟的RAM測試程式，gcc絕對可以一摘后冠。假如你無法重新複製這隻*bug*—當你重新開始編譯時，錯誤的訊息並沒有一直出現在同一個地方—那幾乎可以確定,是你的硬體本身有問題(CPU,記憶體,主機板或是快取記憶體).千萬不要因爲你的電腦可以通過開機程序的測試、或是Windows可以跑得很順、或者其它什麼的，就回過頭來大肆宣傳說這是gcc的一個bug；你所做的這些測試動作，通常沒有什麼實際上的價值，這是很合理的結論。另外，也不要因爲編譯核心時，總是停留在 ‘`make zImage`’的階段，就要大罵這是gcc的bug—當然它會停在那兒啊！做 ‘`make zImage`’時，需要編譯的檔案可能就超過200檔案；我們正在研擬一個替代的方案。

如果你可以重覆產生這個bug，而且（最好是這樣啦！）可以寫一個短小的程式來展示這隻bug的話，你就可以把它做成bug報告，然後email給FSF，或者是linux-gcc通信論壇。你可以去參考gcc的說明文件，看看有什麼詳細的資訊,是他們所需要的。

4.3 移植能力

據報，近日來許多正面的消息指出，若是有某件東東到現在都還沒移植到Linux上去，那麼可以肯定的是，它一定一點價值也沒有。:-)

嗯！正經一點。一般而言，原始碼只需要做一些局部的修改，就可以克服Linux 100%與POSIX相容的特質。如果你做了任何的修改，而將此部份傳回給原作者，會是很有建設性的舉動。這樣日後就只需要用到‘make’，就能得到一個可執行的檔案了。

4.3.1 BSD教徒 (有 `bsd_ioctl`、`daemon` 與 `<sgtty.h>`)

編譯程式時，可以配合`-I/usr/include/bsd`與連結`-lbsd`的程式庫。（例如：在你的Makefile檔內，把`-I/usr/include/bsd`加到`CFLAGS`那一行；把`-lbsd`加到`LDFLAGS`那一行）。如果你真的那麼想要BSD型態的信號行爲，也不需要再加上`-D__USE_BSD_SIGNAL`了。那是因爲當你用了`-I/usr/include/bsd`與含括了標頭檔`<signal.h>`之後，make時就會自動加入了。

4.3.2 失落的封印(`SIGBUS`, `SIGEMT`, `SIGIOT`, `SIGTRAP`, `SIGSYS` etc)

Linux與POSIX是完全相容的。不過，有些信號並不是POSIX定義的—ISO/IEC 9945-1:1990 (IEEE Std 1003.1-1990), paragraph B.3.3.1.1 sez:

> “在POSIX.1中省略了SIGBUS、SIGEMT、SIGIOT、SIGTRAP與SIGSYS信號，那是因爲它們的行爲與實作的方式息息相關，而且也無法進行適當的分類。確認實作方式後，便可以發送這些信號，可是必須以文件說明它們是在什麼樣的環境底下發送出來的，以及指出任何與它們的發展相關的限制。”

想要修正這個問題，最簡單也是最笨的方法就是用`SIGUNUSED`重新定義這些信號。正確的方法應該是以條件式的編譯`#ifdef`來處理這些問題才對：

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

11/15/97譯 5/22/98修正

4.3.3 K & R

gcc是一個與ANSI相容的編譯器；奇怪的是，目前大多數的程式碼都不符合ANSI所定的標準。如果你熱愛ANSI，喜歡用ANSI提供的標準來撰寫C程式，似乎除了加上`-traditional`的旗號之外，就沒有其它什麼可以多談的了。There is a certain amount of finer-grained control over which varieties of brain damage to emulate;請自行查閱gcc info page。

要注意的是，儘管你用了`-traditional`來改變語言的特性，它的效果也僅侷限於gcc所能夠接受的範圍。例如, `-traditional`會打開`-fwritable-strings`，使得字串常數移至資料記憶體空間內(從程式碼記憶體空間移出來，這個地方是不能任意寫入的)。這樣做會讓程式碼的記憶體空間無形中增加的。

4.3.4 前置處理器的符號卯上函數原型宣告

最常見的問題是，如眾所皆知，Linux中有許多常用的函數都定義成巨集存放在標頭檔內，此時若有相似的函數原型宣告出現在程式碼內，前置處理器會拒絕進行語法分析的前置作業。常見的有`atoi()`與`atol()`。

4.3.5 sprintf()

在大部份的Unix系統上，`sprintf(string, fmt, ...)`傳回的是`string`的指標，然而，這方面Linux（遵循ANSI）傳回的卻是放入string內的字元數目.進行移植時，尤其是針對SunOS，需有警覺的心。

4.3.6 fcntl 與相關的函數；FD_*家族的定義到底擺在哪裡?

就在`<sys/time.h>`裡頭。為了真正的原型宣告，當你用了`fcntl`，可能你也想含括標頭檔`<unistd.h>`進來。

一般而言，函數的manual page會在SYNOPSIS章節內列出需要的標頭檔。

4.3.7 select()的計時—程式執行時會處於忙碌-等待的狀態

很久很久以前,，`select()`的計時參數只有唯讀的性而已。即使到了最近，manual pages仍然有下面這段的警告：

> select()應該是藉由修正時間的數值（如果有的話），再傳回自原始計時開始後所剩餘的時間。未來的版本可能會使這項功能實現。因此，就目前而言，若以爲呼叫select()之後，計時指標仍然不會被修正過，可是一種非常不明智的想法喔！

未來就在我們的眼前了！至少，在這兒你絕對可以看到。函數`select()`傳回的，是扣除等待尚未到達的資料所耗費的時間後，其剩餘的時間數值。如果在計時結束時，都沒有資料傳送進來，計時引數便會設爲0；如果接著還有任何的select()，以同樣的計時structure來呼叫，那麼select()便會立刻結束。

若要修正這項問題，只要每次呼叫select()前，都把計時數值放到計時 structure內，就沒有問題了。把下面的程式碼，

```
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);
}
```

這個問題，在有些版本的Mosaic裡是相當著名的，只要一次的等待，Mosaic就掛在那裡了。Mosaic的螢幕右上角，是不是有個圓圓的、會旋轉的地球動畫。那顆球轉得愈快，就表示資料從網路上傳送過來的速率愈慢！

4.3.8 產生中斷的系統呼叫

特徵： 一支程式以Ctrl-Z中止、然後再重新執行時—或者是其它可以產生Ctrl-C中斷信號的情況，如子程序的終結等—系統就會抱怨說"interrupted system call"或是"write: unknown error"，或者諸如此類的訊息。

問題點： POSIX的系統檢查信號的次數，比起一些舊版的Unix是要多那麼一點。如果是Linux，可能就會執行signal handlers了—

- 非同步地(計時器的滴答聲)
- 系統呼叫的傳回值
- 在下列系統呼叫的執行期間： `select()`, `pause()`, `connect()`,`accept()`, `read()` on terminals, sockets, pipes or files in `/proc`, `write()` on terminals, sockets, pipes or the line printer, `open()` on FIFOs, PTYs or serial lines,`ioctl()` on terminals, `fcntl()` with command `F_SETLKW`, `wait4()`, `syslog()`, any TCP or NFS operations.

就其它的作業系統而言，你需要的可能就是下面這些系統呼叫了： `creat()`, `close()`, `getmsg()`, `putmsg()`, `msgrcv()`, `msgsnd()`, `recv()`, `send()`, `wait()`, `waitpid()`, `wait3()`, `tcdrain()`, `sigpause()`, `semop()` to this list.

在系統呼叫期間，若有一信號（那支程式本身應準備好handler因應了）產生，handler就會被呼叫。當handler將控制權轉移回系統呼叫時，它會偵測出它已經產生中斷，而且傳回值會立刻設定成-1，而`errno`設定成`EINTR`。程式並沒有想到會發生這種事，所以就掛了。

有兩種修正的方法可以選擇：

(1) 對每個你自行安裝的signal handler，都須在sigaction的旗號加上`SA_RESTART`。例如，把下列的程式，

```
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，當這部份的變更大量應用到系統呼叫之後，呼叫read()、write()、ioctl()、 select()、 pause() 與 connect()時，你仍然得自行檢查EINTR。如下所示：

(2) 你自己得很明確地檢查EINTR：

這裡有兩個針對read()與ioctl()的例子。

原始的程式片段，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; }
}
```

原始的程式片段，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));
```

注意一點，有些版本的BSD Unix，其內定的行爲是重新執行系統呼叫。若要讓系統呼叫中斷，得使用SV_INTERRUPT或SA_INTERRUPT旗號。

4.3.9 可以寫入的字串

gcc對其users總懷抱著樂觀的想法，相信當他們打算讓某個字串當作常數來用時—那它就真的只是字串常數而已。因此，這種字串常數會儲存在程式碼的記憶體區段內。這塊區域可以page到磁碟機的image上，避免耗掉swap的記憶體空間，而且任何嘗試寫入的舉動都會造成分頁的錯誤(segmentation fault)。這可是一種特色呢！

對老舊一點的程式而言，這可能會產生一個問題。例如，呼叫`mktemp()`，傳遞的引數(arguments)是字串常數。 `mktemp()`會嘗試著在*適當的位置*重新寫入它的引數。

修正的方法不外乎(a)以`-fwritable-strings`編譯，迫使gcc將此常數置放在資料記憶體空間內；或者(b)將侵犯地權的部份重新改寫，配置一個不為常數的字串，在呼叫前，先以strcpy()將資料拷貝進去。

4.3.10 為什麼呼叫`execl()`會失敗？

那是因為你呼叫的方式不對。`execl`的第一個引數是你想要執行的程式名.第二個與接續的引數會變成你所呼叫的程式的`argv`陣列。記住：傳統上，`argv[0]`是只有當程式沒有帶著引數執行時，才會有設定值。所以囉，你應該這樣寫：

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

而不是只有，

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

執行程式而不帶任何引數，可解釋成是一種邀請函，目的是把此程式的動態程式庫獨立的特性印出來。至少，a.out是這樣的。就ELF而言。事情就不是這樣了.

（如果你想得知此程式庫的資訊，有一些更簡單的介面可用；參考動態載入那一章節，或是`ldd`的manual page。）

11/16/97譯 6/2/98修正

5 除錯與監管

5.1 預防重於治療（lint）

lint對Linux而言並沒有很廣泛的用途，主要是因為大部份的人都能滿足於gcc所提供的警告訊息。可能最有用的就是`-Wall`參數了—這個參數的用途是要求gcc將所有的警告訊息顯現出來；but probably has more mnemonic value if thought of as the thing you bang your head against.

網路上有一個實用的public domain lint，位於 `<ftp://larch.lcs.mit.edu/pub/Larch/lclint>`。我並不知道這個站到底有多好就是了。

5.2 除錯

5.2.1 我要怎樣做才能將除錯資訊放到一支程式裡頭？

你需要添加-g的參數來編譯與連結程式，而且不可以用`-fomit-frame-pointer`參數。事實上，你不需要重新編譯所有的程式，只需重新編譯目前你正在除錯的部份即可。

就a.out的組態而言，共享程式庫是以-fomit-frame-pointer編譯而成，這個時候，gdb就變得英雄無用武之地了。連結時給定-g的選項，應該就隱含著靜態連結的意義了；這就是爲什麼要加-g的原因了。

果連結器連結失敗，告訴你找不到libg.a，那就是在/usr/lib/的目錄底下，少了libg.a。libg.a是一個C語言很特別的偵錯程式庫。一般在libc的套件內就會提供libg.a；不然的話（新版是這樣的），你可能需要拿libc的原始碼自己設置了，不過，實際上你應該不需要才對。不管是什麼目的，大部份的情況下，只需將libg.a連結到/usr/lib/libc.a，你就能得到足夠的資訊了。

那，能不能把除錯資訊給拿掉？ 多的GNU軟體在編譯連結時，都會設定-g的選項；這樣做會造成執行檔過大的問題（通常是靜態的連結）。實際上，這並不是一個很熱門的想法。

如果程式本身有autoconf，產生了configure命令稿，通常你就可以用./configure CFLAGS=或是./configure CFLAGS=-O2來關掉除錯資訊。不然的話，你得檢查檢查Makefile了。當然啦，假如你用的是ELF，程式便會以動態的方式來連結，不論是否有-g的設定；因此你可以平常心把-g拿掉。

5.2.2 實用的軟體

據瞭解，大部份的人都是用**gdb**來除錯。你可以從*GNU archive sites* <ftp://prep.ai.mit.edu/pub/gnu>拿到原始程式；或者是到*tsx-11* <ftp://tsx-11.mit.edu/pub/linux/packages/GCC>拿可執行檔。**xxgdb**是一個X介面的除錯程式，植基於gdb（也就是說你得先安裝好gdb，才能再裝xxgdb）。xxgdb的原始碼可以在 <ftp://ftp.x.org/contrib/xxgdb-1.08.tar.gz>找到。

另外，**UPS**除錯程式已由Rick Sladkey移植成功。UPS可以在X底下活得很好，不像xxgdb那樣—僅僅是gdb的X前端介面（X front end）。這支除錯程式有一大堆優良的特點，而且如果你得花時間去除一支破爛的程式，建議你考慮考慮xxgdb。事先編譯好的Linux版與修正版的原始碼可以在 <ftp://sunsite.unc.edu/pub/Linux/devel/debuggers/>找到。而最初的原始程式則放在 <ftp://ftp.x.org/contrib/ups-2.45.2.tar.Z>。

你可能會發現另一個用來除錯的工具**strace**，也是相當的有用。它可以顯示出由程序所產生的系統呼叫，而且還擁有其它眾多繁複的功能，像是如果你手邊沒有原始碼的話，strace可以幫你找出有那些路徑名稱（path-names）已編譯進執行檔內； exacerbating race conditions in programs that you suspect contain them;還有，strace可拿來學習程式是怎麼在電腦中執行的。最新的版本（目前是3.0.8）可在找到 <ftp://ftp.std.com/pub/jrs/>。

5.2.3 背景程式（常駐程式）

早期典型的常駐程式(daemon programs)是執行fork()，然後終止父程序。這樣的做法使得除錯的時間減短了。

瞭解這點的最簡單的方法就是替fork()設一個中斷點（breakpoint）。當程式停止時，強迫fork()傳回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       }
```

5.2.4 核心檔案

當Linux開機時，通常組態會設定成不要產生核心檔案。要是你那麼喜歡它們的話，可以用shell的builtin命令使其重新生效：就C-shell相容的shell（如tcsh）而言，會是下面這樣：

```
% limit core unlimited
```

而類似Bourne shell的shell（sh, bash, zsh, pdksh）則使用下面的語法：

```
$ ulimit -c unlimited
```

如果你想要有個多才多藝的核心檔命名（core file naming）（for example, if you're trying to conduct a post-mortem using a debugger that's buggy itself），那麼你可以對你的核心程式做一點小小的更動。找一找fs/binfmt_aout.c與fs/binfmt_elf.c檔中與下列相符的程式片段(in newer kernels, you'll have to grep around a little in older ones)：

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

將0換成1.

5.3 監管

監管（Profiling）是用來檢核一支程式中那些部份是最常呼叫或是執行的時間最久的方法。這對程式的最佳化與找出何時時間是浪費掉的而言，是相當好的方式。你必須就你所要的時程資訊（timing information）的目的檔加上-p來編譯，而且如果要讓輸出的檔案有意義，你也會需要gprof（來自binutils套件的命令）。參閱gprof的manual page，可得知其細節。

11/18/97譯

6 連結

由於靜態與共享程式庫兩者間不相容的格式的差異性與動詞*link*過量使用於指稱*編譯完成後的事情*與*當編譯好的程式使用時所發生的事情*這兩件事上頭，使得這一章節變得複雜了許多。（ and, actually, the overloading of the word ‘load’ in a comparable but opposite sense）不過，再複雜也就是這樣了，所以閣下不必過於擔心。

爲了稍微減輕讀者的困惑，我們稱執行期間所發生的事爲*動態載入*，這一主題會在下一章節中談到。你也會在別的地方看到我把動態載入描述成*動態連結*，不過不會是在這一章節中。換句話說，這一章節所談的，全部是指發生在編譯結束後的連結。

6.1 共享程式庫 vs靜態程式庫

建立程式的最後一個步驟便是連結；也就是將所有分散的小程式組合起來，看看是否遺漏了些什麼。顯然，有一些事情是很多程式都會想做的—例如，開啓檔案，接著所有與開檔有關的小程式就會將儲存程式庫的相關檔案提供給你的程式使用。在一般的Linux系統上，這些小程式可以在`/lib`與`/usr/lib/`目錄底下找到。

當你用的是靜態的程式庫時，連結器會找出程式所需的模組，然後實際將它們拷貝到執行檔內。然而，對共享程式庫而言，就不是這樣了。共享程式庫會在執行檔內留下一個記號，指明*當程式執行時，首先必須載入這個程式庫*。顯然，共享程式庫是試圖使執行檔變得更小，等同於使用更少的記憶體與磁碟空間。Linux內定的行爲是連結共享程式庫，只要Linux能找到這些共享程式庫的話，就沒什麼問題；不然,Linux就會連結靜態的了。如果你想要共享程式庫的話，檢查這些程式庫（`*.sa` for a.out, `*.so` for ELF）是否住在它們該在的地方，而且是可讀取的。

在Linux上，靜態程式庫會有類似`libname.a`這樣的名稱；而共享程式庫則稱爲`libname.so.x.y.z`，此處的`x.y.z`是指版本序號的樣式。共享程式庫通常都會有連結符號指向靜態程式庫（很重要的）與相關聯的`.sa`檔案。標準的程式庫會包含共享與靜態程式庫兩種格式。

你可以用`ldd`（List Dynamic Dependencies）來查出某支程式需要哪些共享程式庫。

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

這是說在我的系統上，WWW瀏覽器*lynx*會依賴`libc.so.5` (the C library)與`libncurses.so.1`（終端機螢幕的控制）的存在。若某支程式缺乏獨立性，`ldd`就會說‘`statically linked`’或是‘`statically linked (ELF)`’。

6.2 終極審判（‘`sin()` 在哪個程式庫裡？’）

nm 程式庫名稱應該會列出此程式庫名稱所參考到的所有符號。這個指令可以應用在靜態與共享程式庫上。假設你想知道`tcgetattr()`是在哪兒定義的：你可以如此做，

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

*U*指出*未定義*—也就是說ncurses程式庫有用到tegetattr()，但是並沒有定義它。你也可以這樣做，

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

*W*說明了*弱態(weak)*，意指符號雖已定義，但可由不同程式庫中的另一定義所替代。最簡單的*正常*定義（像是tcgetpgrp）是由*T*所標示：

標題所談的問題，最簡明的答案便是libm.(so|a)了。所有定義在<math.h>的函數都保留在maths程式庫內；因此，當你用到其中任何一個函數時，都需要以-lm的參數連結此程式庫。

6.3 X檔案？

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

ld與其相類似的命令在搜尋檔案的策略上，會依據版本的差異而有所不同，但是唯一一個你可以合理假設的內定目錄便是/usr/lib了。如果你希望身處它處的程式庫也列入搜尋的行列中，那麼你就必須以-L選項告知gcc或是ld。

要是你發現一點效果也沒有，就趕緊察看看那檔案是不是還乖乖的躺在原地。就a.out而言，以-lfoo參數來連結，會驅使ld去尋找libfoo.sa（shared stubs）；如果沒有成功，就會換成尋找libfoo.a（static）。就ELF而言， ld會先找libfoo.so，然後是libfoo.a。libfoo.so通常是一個連結符號，連結至libfoo.so.x。

6.4 建立你自己的程式庫

6.4.1 控制版本

與其它任何的程式一樣，程式庫也有修正不完的bugs的問題存在。它們也可能產生出一些新的特點，更改目前存在的模組的功效，或是將舊的移除掉。這對正在使用它們的程式而言，可能會是一個大問題。如果有一支程式是根據那些舊的特點來執行的話，那怎麼辦？

所以，我們引進了程式庫版本編號的觀念。我們將程式庫*次要*與*主要*的變更分門別類，同時規定*次要*的變更是不允許用到這程式庫的舊程式發生中斷的現象。你可以從程式庫的檔名分辨出它的版本（實際上，嚴格來講，對ELF而言僅僅是一場天大的謊言；繼續讀將下去，便可明白為什麼了）：libfoo.so.1.2的主要版本是1，次要版本是2。次要版本的編號可能真有其事，也可能什麼都沒有—libc在這一點上用了*修正程度*的觀念，而訂出了像libc.so.5.2.18這樣的程式庫名稱。次要版本的編號內若是放一些字母、底線、或是任何可以列印的ASCII字元，也是很合理的。

ELF與a.out格式最主要的差別之一就是在設置共享程式庫這件事上；我們先看ELF，因為它比較簡單一些。

6.4.2 ELF？它到底是什麼東東ㄋㄟ？

ELF（Executable and Linking Format）最初是由USL（UNIX System Laboratories）發展而成的二進位格式，目前正應用於Solaris與System V Release 4上。由於ELF所增漲的彈性遠遠超過Linux過去所用的a.out格式，因此GCC與C程式庫的發展人士於1995年決定改用ELF為Linux標準的二進位格式。

怎麼又來了？　一節是來自於‘/news-archives/comp.sys.sun.misc’的文件。

> ELF（“Executable Linking Format”）是於SVR4所引進的新式改良目的檔格式。ELF比起COFF可是多出了不少的功能。以ELF而言，它*是*可由使用者自行延伸的。ELF視一目的檔爲節區（sections），如串列般的組合；而且此串列可爲任意的長度（而不是一固定大小的陣列）。這些節區與COFF的不一樣，並不需要固定在某個地方，也不需要以某種順序排列。如果使用者希望補捉到新的資料，便可以加入新的節區到目的檔內。ELF也有一個更強而有力的除錯法式，稱爲DWARF（Debugging With Attribute Record Format）—目前Linux並不完全支援。DWARF DIEs（Debugging Information Entries）的連結串列會在ELF內形成 .debug的節區。DWARF DIEs的每一個 .debug節區並非一些少量且固定大小的資訊記錄的集合，而是一任意長度的串列，擁有複雜的屬性，而且程式的資料會以有範圍限制的樹狀資料結構寫出來。DIEs所能補捉到的大量資訊是COFF的 .debug節區無法望其項背的。（像是C++的繼承圖。）
>
> ELF檔案是從SVR4（Solaris 2.0？）ELF存取程式庫（ELF access library）內存取的。此程式庫可提供一簡便快速的介面予ELF。使用ELF存取程式庫最主要的恩惠之一便是，你不再需要去察看一個ELF檔的qua了。就UNIX的檔案而言，它是以Elf*的型式來存取；呼叫elf_open()之後，從此時開始，你只需呼叫elf_foobar()來處理檔案的某一部份即可，並不需要把檔案實際在磁碟上的image搞得一團亂。

ELF的優缺點與昇級至ELF等級所需經歷的種種痛苦，已在ELF-HOWTO內論及；我並不打算在這兒塗漿糊。ELF HOWTO應該與這份文件有同樣的主題才是。

ELF共享程式庫　想讓`libfoo.so`成爲共享程式庫，基本的步驟會像下面這樣：

```
$ 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
```

這會產生一個名爲`libfoo.so.1.0`的共享程式庫，以及給予ld適當的連結（`libfoo.so`）還有使得動態載入程式（dynamic loader）能找到它（`libfoo.so.1`）。爲了進行測試，我們將目前的目錄加到`LD_LIBRARY_PATH`裡。

當你津津樂道於程式庫製做成功之時，別忘了把它移到如`/usr/local/lib`的目錄底下，並且重新設定正確的連結路徑。`libfoo.so.1`與`libfoo.so.1.0`的連結會由`ldconfig`依日期不斷的更新，就大部份的系統來說，ldconfig會在開機過程中執行。`libfoo.so`的連結必須由手動方式更新。如果你對程式庫所有組成份子（如標頭檔等）的昇級，總是抱持著一絲不苟的態度，那麼最簡單的方法就是讓`libfoo.so -> libfoo.so.1`；如此一來，ldconfig便會替你同時保留最新的連結。要是你沒有這麼做，你自行設定的東東就會在數日後造成千奇百怪的問題出現。到時候，可別說我沒提醒你啊！

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

版本編號、soname與符號連結 一個程式庫都有一個*soname*。當連結器發現它正在搜尋的程式庫中有這樣的一個名稱，連結器便會將soname箝入連結中的二進位檔內，而不是它正在運作的實際的檔名。在程式執行期間，動態載入程式會搜尋擁有soname這樣的檔名的檔案，而不是程式庫的檔名。因此，一個名爲`libfoo.so`的程式庫，就可以有一個`libbar.so`的soname了。而且所有連結到`libbar.so`的程式，當程式開始執行時，會尋找的便是`libbar.so`了。

這聽起來好像一點意義也沒有，但是這一點，對於瞭解數個不同版本的同一個程式庫是如何在單一系統上共存的原因，卻是關鍵之鑰。Linux程式庫標準的命名方式，比如說是`libfoo.so.1.2`，而且給這個程式庫一個`libfoo.so.1`的soname。如果此程式庫是加到標準程式庫的目錄底下（e.g. `/usr/lib`），`ldconfig`會建立符號連結`libfoo.so.1 -> libfoo.so.1.2`，使其正確的image能於執行期間找到。你也需要連結`libfoo.so -> libfoo.so.1`，使ld能於連結期間找到正確的soname。

所以囉，當你修正程式庫內的bugs，或是添加了新的函數進去（任何不會對現存的程式造成不利的影響的改變），你會重建此程式庫，保留原本已有的soname，然後更改程式庫檔名。當你對程式庫的變更會使得現有的程式中斷，那麼你只需增加soname中的編號—此例中，稱新版本爲`libfoo.so.2.0`，而soname變成`libfoo.so.2`。緊接著，再將`libfoo.so`的連結轉向新的版本；至此，世界又再度恢復了和平！

其實你不須要以此種方式來替程式庫命名，不過這的確是個好的傳統。ELF賦予你在程式庫命名上的彈性，會使得人氣喘呼呼的搞不清楚狀況；有這樣的彈性在，也並不表示你就得去用它。

ELF總結：假設經由你睿智的觀察發現有個慣例說：程式庫主要的昇級會破壞相容性；而次要的昇級則可能不會；那麼以下面的方式來連結，所有的一切就都會相安無事了。

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

6.4.3 a.out—舊舊的格式~~~

建立共享程式庫的便利性是昇級至ELF的主要原因之一。那也是說,a.out可能還是有用處在的。上ftp站去抓 `<ftp://tsx-11.mit.edu/pub/linux/packages/GCC/src/tools-2.17.tar.gz>`；解壓縮後你會發現有20頁的文件可以慢慢的讀哩。我很不喜歡自己黨派的偏見表現得那麼的淋璃盡致，可是從上下文間，應該也可以很清楚的嗅出我從來不拿石頭砸自己的腳的脾氣吧！:-)

ZMAGIC vs QMAGIC QMAGIC是一種類似舊格式的a.out（亦稱爲ZMAGIC）的可執行檔 格式，這種格式會使得第一個分頁無法map。當0-4096的範圍內沒有mapping存在時，則可允許NULL dereference trapping更加的容易。所產生的邊界效應是你的執行檔會比較小（大約少1K左右）。

只有即將作廢的連結器有支援ZMAGIC，一半已埋入棺材的連結器有支援這兩種格式；而目前的版本僅支援QMAGIC而已。事實上，這並沒有多大的影響，那是因爲目前的核心兩種格式都能執行。

*file*命令應該可以確認程式是不是QMAGIC的格式的。

檔案配置 a.out(DLL)的共享程式庫包含兩個眞實的檔案與一個連結符號。就*foo*這個用於整份文件做爲範例的程式庫而言，這些檔案會是`libfoo.sa`與`libfoo.so.1.2`；連結符號會是`libfoo.so.1`，而且會指向`libfoo.so.1.2`。這些是做什麼用的？

在編譯時，`ld`會尋找`libfoo.sa`。這是程式庫的*stub*檔案。而且含有所有執行期間連結所需的exported的資料與指向函數的指標。

執行期間，動態載入程式會尋找`libfoo.so.1`。這僅僅是一個符號連結，而不是眞實的檔案。故程式庫可更新成較新的且已修正錯誤的版本，而不會損毀任何此時正在使用此程式庫的應用程式。在新版

—比如說`libfoo.so.1.3`—已完整呈現時，ldconfig會以一極微小的操作，將連結指向新的版本，使得任何原本使用舊版的程式不會感到絲毫的不悅。

DLL程式庫（我知道這是無謂的反覆—所以對我提出訴訟吧！）通常會比它們的靜態副本要來得大多。它們是以*洞（holes）*的形式來保留空間以便日後的擴充。這種*洞*可以不佔用任何的磁碟空間。一個簡單的`cp`呼叫，或是使用`makehole`程式，就可以達到這樣效果。因爲它們的位址是固定在同一位置上，所以在建立程式庫後，你可以把它們拿掉。不過，千萬不要試著拿掉**ELF**的程式庫。

"libc-lite"? libc-lite是輕量級的libc版本。可用來存放在磁碟片上，也可以替大部份低微的UNIX任務收尾。它沒有包含curses, dbm, termcap等等的程式碼。如果你的`/lib/libc.so.4`是連結到一個lite的libc，那麼建議你以完整的版本取代它。

6.4.4 連結：常見的問題

把你連結時所遭遇的問題寄給我！我可能什麼事也不會做，但是只要累積了足夠的數量，我會把它們寫起來*。

你想共享，偏偏程式卻連結成靜態的！

檢查你提供給`ld`的連結是否正確，使ld能找到每一個對應的共享程式庫，就ELF而言，這是指一個符號連結`libfoo.so`，連結至image；就a.out而言，就是`libfoo.sa`檔了。很多人將ELF binutils 2.5昇級至2.6之後，就產生了這個問題—早期的版本搜尋共享程式庫時較有智慧，所以並沒有將所有的連結建立起來。後來，爲了與其它的架構相容，這項充滿智慧的行爲被人給刪除掉了，另外，這樣的*智慧*判斷錯誤的機率相當高，所造成的麻煩比它所解決的問題還多，所以留著也是害人精；不如歸去兮！

DLL的工具程式‘**mkimage**’找不到**libgcc**？

自`libc.so.4.5.x`之後，libgcc已不再是共享的格式。因此，你必須在*`-lgcc`*出現之處以‘`gcc -print-libgcc-file-name`‘替代（完整的倒單引號（back-quotes））。另外，刪除所有`/usr/lib/libgcc*`的檔案。這點很重要哩。

`__NEEDS_SHRLIB_libc_4 multiply defined` **messages**

是同樣的問題所造成的另一種結果。

"Assertion failure" message when rebuilding a DLL ?

這一條神秘的訊息最有可能的原因是，在原始的`jump.vars`檔案內，由於保留的空間太少，以致於造成其中一個jump table slots溢滿。你可以執行工具程式—由2.17.tar.gz套件所提供的‘`getsize`’命令，定出所有嫌疑犯的蹤跡。可能唯一的解決方法是，解除此程式庫主要的版本編號，強迫它回到不相容的年代。

`ld: output file needs shared library libc.so.4`

通常這是發生在當你連結的程式庫不是libc（如X程式庫），而且在命令列用了-g的參數，卻沒有一併使用`-static`，所發出的錯誤訊息。

共享程式庫的`.sa` stubs通常有一個未定義的符號`_NEEDS_SHRLIB_libc_4`；這一點可藉由`libc.sa` stub來解決，然而，以-g來編譯時，會使得連結以`libg.a`或`libc.a`來結束；因此這個符號一直就沒有解決，也就會導致上面的錯誤訊息了。

總之，以-g的旗號編譯時別忘了加上`-static`，不然就別用-g來連結。通常，以-g編譯各個獨立的檔案時，所獲得的除錯資訊已經足夠，連結時就可以不需要它了。

7 動態載入

這一章節目前是簡短了一點；當我掠盡*ELF HOWTO*時，就是這部份再度擴展的時候了。

7.1 基本概念

Linux有共享程式庫，如果之前你已坐著讀完上一章節，想必現在一聽到像這樣的說詞，便會立刻感到頭昏。有一些照慣例而言是在連結時期便該完成的工作，必須延遲到載入時期才能完成。

7.2 錯誤訊息

把你連結的錯誤寄給我！我不會做任何的事，不過我可以把它們寫起來**

`can't load library: /lib/libxxx.so, Incompatible version`

(a. out only) 這是指你沒有xxx程式庫的正確的主要版本。可別以爲隨隨 便便弄個連結到你目前擁有的版本就可以了，如果幸運的話，就只會造成你的程式分頁錯誤而已。去抓新的版本.ELF類似的情況會造成像下面這樣的訊息：

```
ftp: can't load library 'libreadline.so.2'
```

`warning using incompatible library version xxx`

(a. out only)你的程式庫的次要版本比起這支程式用來編譯的還要舊。程式依然可以執行。只是可能啦！我想，昇個級應該沒什麼傷害吧！

7.3 控制動態載入器的運作

有一組環境變數會讓動態載入器有所反應。大部份的環境變數對`ldd`的用途要比起對一般users的還要來得更多。而且可以很方便的設定成由ldd配合各種參數來執行。這些變數包括，

- `LD_BIND_NOW` — 正常來講，函數在呼叫之前是不會讓程式尋找的。設定這個旗號會使得程式庫一載入，所有的尋找便會發生，同時也造成起始的時間較慢。當你想測試程式，確定所有的連結都沒有問題時，這項旗號就變得很有用。

- `LD_PRELOAD`可以設定一個檔案，使其具有*覆蓋*函數定義的能力。例如，如果你要測試記憶體分配的方略，而且還想置換*malloc*，那麼你可以寫好準備替換的副程式，並把它編譯成`mallolc.`，然後：

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

  `LD_ELF_PRELOAD` 與 `LD_AOUT_PRELOAD` 很類似，但是僅適用於正確的二進位型態。如果設定了 `LD_`*something*`_PRELOAD` 與 `LD_PRELOAD`，比較明確的那一個會被用到。

- `LD_LIBRARY_PATH`是一連串以分號隔離的目錄名稱，用來搜尋共享程式庫。對ld而言，並沒有任何的影響；這項只有在執行期間才有影響。另外，對執行setuid與setgid的程式而言，這一項是無效的。而`LD_ELF_LIBRARY_PATH`與`LD_AOUT_LIBRARY_PATH`這兩種旗號可根據各別的二進位型式分別導向不同的搜尋路徑。一般正常的運作下，不應該會用到`LD_LIBRARY_PATH`；把需要搜尋的目錄加到`/etc/ld.so.conf/`裡；然後重新執行ldconfig。

- LD_NOWARN 僅適用於a.out。一旦設定了這一項（LD_NOWARN=true; export LD_NOWARN），它會告訴載入器必須處理fatal-warnings（像是次要版本不相容等）的警告訊息。
- LD_WARN僅適用於ELF。設定這一項時，它會將通常是致命訊息的"Can*t find library"轉換成警告訊息。對正常的操作而言，這並沒有多大的用處，可是對ldd就很重要了。
- LD_TRACE_LOADED_OBJECTS僅適用於ELF。而且會使得程式以爲它們是由ldd所執行的：

```
$ 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
```

7.4 以動態載入撰寫程式

如果你很熟悉Solaris 2.x所支援的動態載入的工作的話，你會發現Linux在這點上與其非常的相近。這一部份在H.J.Lu的ELF程式設計文件內與dlopen(3)的manual page（可以在ld.so的套件上找到）上有廣泛的討論。這裡有個不錯的簡單範例：以-ldl連結。

```
#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");
  }

}
```

8 與發展人士聯絡

8.1 Bug報表

把問題寫下來。這是針對Linux的，亦或是gcc在其它系統上所發生的問題。與kernel的版本相關嗎？或者是程式庫的版本？如果改用靜態方式連結，問題是不是就消失了？你可以節錄一小段程式來展示這隻bug嗎？

當你做了這些事情之後，你將會知道程式內的bugs是什麼。就gcc而言，bug報表程序是以info檔來說明的。如果是ld.so或是C、maths程式庫，將email寄到linux-gcc@vger.rutgers.edu。如果可能的話，包含一支自己自足的小程式以展示這個bug，而且附上說明，描述你想要讓這支程式做些什麼與實際上它又做了些什麼。

8.2 協助發展

如果你想要幫忙發展gcc或是C程式庫，第一件事就是加入`linux-gcc@vger.rutgers.edu`通信論壇。如果你只是想看看通信論壇在討論些什麼，這裡有一個論壇的archives，位於 `<http://homer.ncm.com/linux-gcc/>`。接下來的事，就看你想做什麼了。

9 結語

9.1 名人榜

> Only presidents, editors, and people with tapeworms have the right to use the editorial “we”.

(Mark Twain)

這份HOWTO文件幾乎完全根植於Mitchum Dsouza的GCC-FAQ； 文件中大部份的資訊是直接來自於GCC-FAQ。這份文件用到的第一人稱代名詞，可視爲我們兩人其中一個；可以說，要是其中有一個人說“我還沒有測試過這些；如果它燒了你的硬碟、毀了你的系統或是搞得你妻離子散的，可別說我沒提醒你啊！"，像這樣的話適用於我倆身上。

對這份文件有貢獻的名人雅士如下所列（以ASCII碼的順序列出）： Andrew Tefft, Axel Boldt, Bill Metzenthen, Bruce Evans, Bruno Haible, Daniel Barlow, Daniel Quinlan, David Engel, Dirk Hohndel, Eric Youngdale, Fergus Henderson, H.J. Lu, Jens Schweikhardt, Kai Petzke, Michael Meissner, Mitchum DSouza, Olaf Flebbe, Paul Gortmaker, Rik Faith, Steven S. Dick, Tuomas J Lukka, 當然還有Linux Torvalds，沒有了他，這整個運動就變得一點意義也沒有了，所以不可能讓他孤單的。:-)

請不要覺得有任何的冒犯之處，如果您的名字沒有出現在這兒。如果您對這份文件（HOWTO或是FAQ）曾經有過貢獻的話，請email給我，我會改正過來的。

9.2 翻譯

到目前爲止，這份文件還沒有已知的翻譯版本出現。如果你希望生一個出來，請儘管去做，不過一定得告訴我相關的事宜。我講的語言會是你想要翻的語言，那機率（很遺憾）只有好幾百分之一，不過還是先把悲傷擺一旁吧，不管是什麼樣問題，我都會很樂意幫忙的。

9.3 歡迎任何的回饋

寄信給我*dan@detached.demon.co.uk*。我的PGP public key (ID 5F263625) 可在我的烘培雞*web pages* `<http://ftp.linux.org.uk/~barlow/>`上使用，如果你覺得事情有必要保密的話。

9.4 合法的行逕規定

本文採用過的商標分屬於個別的持有者。

6/3/98增修

10 索引

開頭如果不是文數字的字元，排列就按照ASCII碼的順序。