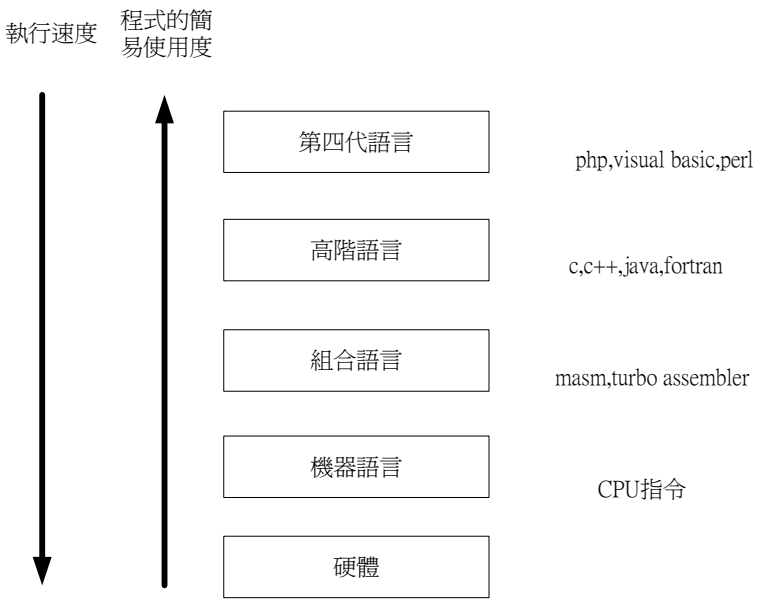
第 17 章
Linux 上的
程式軟體工具

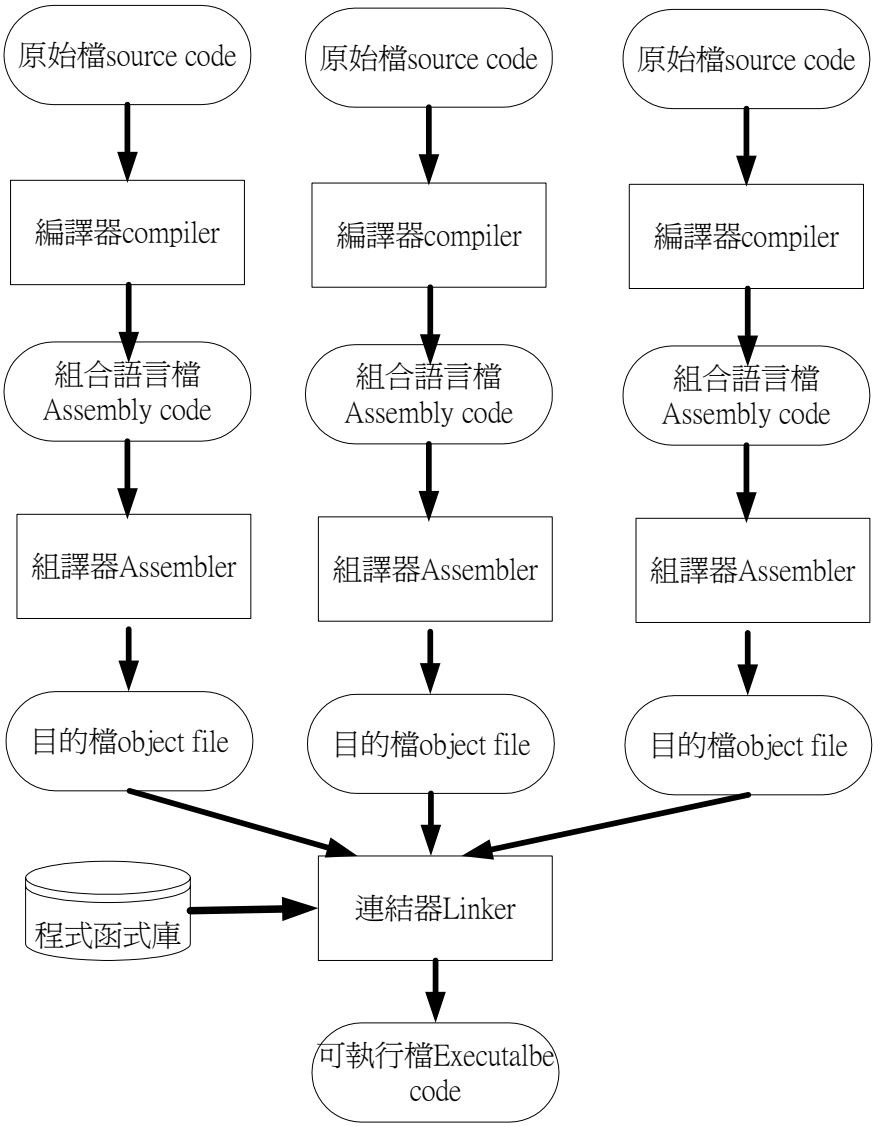
第 17 章 Linux 上的程式軟體工具

越高階的程式語言越容易撰寫，但是卻可能造成執行速度的降低。雖然有時我們使用組合語言寫的程式，組譯過後，盡然發現比用 C 語言寫的還要跑得慢，這是因為 C 語言的編譯器把我們的程式最佳化了。因此，要用什麼軟體來解決問題或寫程式就要看我們自己的需求了。



程式語言執行的速度

我們所撰寫的程式碼(原始檔)，經過編譯器編譯成組合語言，再經由組譯器組譯成目的檔，然後再經過連結器，將數個目的檔和函式庫的函式組成可執行檔(執行檔可在機器上 CPU 執行)。



將高階語言轉換成可執行檔

17-1 編寫程式的工具

我們編寫程式可以用 Linux 文字編輯器(pico 編輯器、vi 編輯器、emacs 編輯器和 xemacs 編輯器)。

我們可以使用 vi 編輯器來編輯 second.c，這是一個 C 語言的檔案。

```
[root@flash /]# vi second.c
```

我們可以輸入下列程式碼。

```
main( )
{
int i,j;
for(i=0,j=10;i<j;i++)
{
printf("大家歡喜快樂");
}
}
```

我們使用 more 指令來顯示我們所輸入的程式碼。

```
[root@flash /]# more second.c
main( )
{
int i,j;
for(i=0,j=10;i<j;i++)
{
printf("大家歡喜快樂");
}
}
```

我們使用 indent 指令來將 second.c 的檔案給標準格式化，並且將它備份成 second.c~。

```
[root@flash /]# indent second.c
[root@flash /]# ls
bin   dev  home    lib         misc  opt   root  second.c   tftpboot  usr
boot  etc  initrd  lost+found  mnt   proc  sbin  second.c~  tmp       var
```

我們使用 more second.c 來顯示我們已經標準格式的 C 語言檔。

```
[root@flash /]# more second.c
main ()
{
  int i, j;
  for (i = 0, j = 10; i < j; i++)
    {
      printf ("大家歡喜快樂");
    }
}
```

這是我們備份的 second.c~檔。

```
[root@flash /]# more second.c~
main( )
{
int i,j;
for(i=0,j=10;i<j;i++)
{
printf("大家歡喜快樂");
}
}
```

Indent 指令從我們輸入的檔案將它按照 C 的語法來將內容給標準格式化，然後將原來的檔案給取代成標準格式的檔案，再將原來輸入的檔案備份起來。

語法：

指令：indent [參數]　[輸入的檔案]

參數：

-bad　：在宣告變數後空一行

-bap：在函式主體後空一行

-bl：以 Pascal 語法格式

-kr：以 Kernighan &Ritchie coding 方式格式

-orig：以 Berkeley coding 方式格式

-st ：以標準格式輸出

我們使用 indent –st second.c 將 second.c 以標準格式輸出。

```
[root@flash /]# indent -st second.c
main ()
{
  int i, j;
  for (i = 0, j = 10; i < j; i++)
    {
      printf ("大家歡喜快樂");
    }
}
```

我們使用 indent -kr –st second.c 將 second.c 以以 Kernighan &Ritchie coding 方式格式輸出。

```
[root@flash /]# indent -kr -st second.c
main()
{
    int i, j;
    for (i = 0, j = 10; i < j; i++) {
        printf("大家歡喜快樂");
    }
}
```

我們使用 indent –bad –st second.c 將 second.c 在宣告變數後空一行。

```
[root@flash /]# indent -bad -st second.c
main ()
{
  int i, j;

  for (i = 0, j = 10; i < j; i++)
    {
      printf ("大家歡喜快樂");
    }
}
```

17-2 編譯 C 語言程式

17-2-1 檔案與路徑名稱

當我們新增一個目錄時，會自動產生兩個檔案名稱：.(稱為點)及..(稱為點點)。.點為表示目前的目錄，點點表示上一層的目錄。路徑名稱是以斜線字元/相隔，然後由多個檔案名稱連接起來的結果。如果以斜線字元/當作路徑的開頭則我們稱為絕對路徑，如果不是以斜線字元為開頭，我們稱為相對路徑。

我們使用 echo $PATH 來顯示系統預設的路徑。

```
[root@flash chaiyen]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
:.:/home/chaiyen/bin
```

我們也可以使用 PATH="$PATH:/home"來設定/home 為我們預設的路徑。這些都是作臨時的更改路徑，如果使用者每次在登錄時，就會預設我們的路徑，則需修改使用者的.bashrc 檔案。

```
[root@flash chaiyen]# PATH="$PATH:/home"
[root@flash chaiyen]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
:.:/home/chaiyen/bin:/home
```

我們也可以使用 export 指令來設定臨時的環境路徑。

```
[root@flash /]# export PATH="$PATH:/var"
[root@flash /]# echo $PATH
/usr/kerberos/sbin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin
:.:/home/chaiyen/bin:/home:/var
```

每個使用者都有他自己的.bashrc 檔，因此我們使用 vi 編輯我們自己的.bashrc 檔。~/.bashrc 指的是使用者家目錄下的 bashrc 檔案。

#vi ~/.bashrc

我們在第四行加入 PATH="$PATH：."這樣就可以設定我們目前的目錄為系統設定 PATH 環境變數設定的路徑去找檔案。

```
1 # .bashrc
2
3 # User specific aliases and functions
4 PATH="$PATH:."
5 # Source global definitions
6 if [ -f /etc/bashrc ]; then
7         . /etc/bashrc
8 fi
```

我們使用 gcc(GNU C/C++ 編譯器)來編譯我們剛才的程式 second.c。

```
[root@flash /]# gcc second.c
```

我們然後直接打上程式的名子就可以執行了。

```
[root@flash /]# ls
a.out  dev   initrd      misc  proc  second     tftpboot  var
bin    etc   lib         mnt   root  second.c   tmp
boot   home  lost+found  opt   sbin  second.c~  usr
[root@flash /]# second
大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜
快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂[root@flash /]#
```

我們也可以打上．/second 來執行。．/這是表示在所在的目錄底下。

```
[root@flash /]# ./second
大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜
快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂[root@flash /]#
```

我們也可以輸入 a.out 來執行我們的 second 程式。

```
[root@flash /]# a.out
大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜
快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂[root@flash /]# _
```

我們也可以使用 cc 指令來編譯。

```
[root@flash /]# cc -o second1 second.c
[root@flash /]# ls
a.out  bab.c  bb.s  dev   initrd      misc  proc  second    second.c~  usr
ba     ba.c   bin   etc   lib         mnt   root  second1   tftpboot   var
bab    ba.cc  boot  home  lost+found  opt   sbin  second.c  tmp
```

gcc 這個指令可以啟動 C 編譯系統，當我們執行時，它就會編譯、組譯、和連接函式庫，來產生可執行的檔案。這個可執行的檔案預設是放在 a.out 的地方。gcc 指令接受存成．a 的延伸檔(靜態函式庫)、．so 或．sa(共享函式庫)、C 的原始檔(．c 延伸檔)、C++原始檔(．C、．cc 或．cpp 的延伸檔)、組合語言檔(．s 延伸檔)或物件

檔(. o 延伸檔)。當不是我們預設的延伸檔時，指令會它當作物件檔或函式檔，而我們這些被編譯的檔案是放到檔案串列中。

語法：

指令：gcc [參數] 檔案串列

參數：

-ansi：以 ANSI 的標準。

-c：阻止連接階段而保留目的檔。

-g：給 GNU 除錯器建立符號表、檔案資訊或除錯資訊。

-llib：連接到函式庫。

-mconfig：針對中央處理器將程式碼最佳化。包含 X86 或 MIPS 中央處理器。

-o 檔案名稱：建立可執行的檔案名稱，而不是預設的 a.out。

-O [level]：程式最佳化的層度。0 到 3，數字越高，最佳化的層度越高。0 則表示程式無最佳化。

-pg：提供給 gprof 使用的檔案資訊

-s：沒有組譯或連接 . c 的檔案

-v：顯示所有的指令

-w：沒有警告

我們使用 cat good.c 來顯示我們的檔案。

```
[root@flash chaiyen]# cat good.c
main()
{
  int i, j;

  for (i = 0, j = 10; i < j; i++)
    {
      printf ("大家歡喜快樂");
    }
}

[root@flash chaiyen]# ls
fol.txt  fo2.txt  fool.txt  foo.paths  foo.txt  good.c  -print  xpilot.motd
```

我們使用 gcc good.c 來編譯 good.c 並且將可執行檔放到 a.out。

```
[root@flash chaiyen]# gcc good.c
[root@flash chaiyen]# ls
a.out     fo2.txt   foo.paths  good.c  xpilot.motd
fol.txt   fool.txt  foo.txt    -print
[root@flash chaiyen]# a.out
大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜
快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂[root@flash chaiyen]# _
```

我們使用 gcc –o good good.c 來將 good.c 編譯成 good 可執行檔。

```
[root@flash chaiyen]# gcc -o good good.c
[root@flash chaiyen]# ls
a.out     fo2.txt   foo.paths  good     -print
fol.txt   fool.txt  foo.txt    good.c   xpilot.motd
[root@flash chaiyen]# good
大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂大家歡喜
快樂大家歡喜快樂大家歡喜快樂大家歡喜快樂[root@flash chaiyen]#
```

17-2-2 靜態函式庫

簡單的函式庫就是將單純可用的物件放在一起，當需要時就到指定的路徑去呼叫它們，它包含一個標頭檔讓我們可以呼叫函式，靜態的函式它為 . a 的延伸檔。我們使用 ar(archive)來建立與維護我們的靜態函式庫，而我們組譯的方式是用 gcc –c。

我們可以建立靜態函式庫分別為 good 與 lucky 的函數其原始資源檔為 good.c 與 lucky.c。

我們使用 vi 編輯 good.c 的檔案。

```
[root@flash chaiyen]# vi good.c
```

我們輸入這五行，用來顯示我們主程式 total 所要顯示的字串。這是將當作函數的檔案。

```
 1 #include <stdio.h>
 2 void good(char *argu)
 3 {
 4     printf("good:大家好 %d \n", argu);
 5 }
```

我們使用 vi 編輯 lucky.c 的檔案。

```
[root@flash chaiyen]# vi lucky.c
```

我們輸入這五行，用來顯示我們主程式 total 所要顯示的字串。這是將當作函數的檔案。

```
1 #include <stdio.h>
2 void good(char *argu)
3 {
4    printf("good:大家好 %s \n", argu);
5 }
```

我們使用 vi 編輯標頭檔 lib.h。這是宣告將函式庫之函數與將會用到這些函數的程式包含 include 起來。

```
[root@flash chaiyen]# vi lib.h
```

這是我們在 lib.h 宣告之函數。

```
1 void good(char *);
2 void lucky(char *);
```

我們使用 vi 編輯我們的主程式 total.c。

```
[root@flash chaiyen]# vi total.c
```

第一行它包括了我們的標頭檔 lib.h，第五行和第六行它呼叫了 good( )函數和 lucky( )函數，並且將參數字串給代入。

```
1 #include "lib.h"
2
3 int main()
4 {
5   good("歡喜快樂");
6   lucky("天天美好");
7   exit(0);
8 }
```

我們組譯了 total.c 而且產生了 total.o 目的檔。

```
[root@flash chaiyen]# gcc -c total.c
[root@flash chaiyen]# ls
a.out    fo2.txt   foo.paths  good.c  lucky.c  total.c  xpilot.motd
fol.txt  fool.txt  foo.txt    lib.h   -print   total.o
```

我們將 total.o、good.c 和 lucky.c 一起組譯成 total。然後我們就可以執行 total 了。

```
[root@flash chaiyen]# gcc -o total total.o good.c lucky.c
[root@flash chaiyen]# ls
a.out    fo2.txt   foo.paths  good.c  lucky.c  total    total.o
fol.txt  fool.txt  foo.txt    lib.h   -print   total.c  xpilot.motd
[root@flash chaiyen]# total
good:大家好 歡喜快樂
lucky: 大家歡喜 天天美好
```

我們將 gcc –c good.c lucky.c 將 good.c 與 lucky.c 組譯成目的檔。我們然後建立並使用函式庫, 我們使用 ar 來建立 archive 來將目的檔 good.o 與 lucky.o 加入到 lib.a 中。

```
[root@flash chaiyen]# ls
a.out    fo2.txt   foo.paths  good.c  lucky.c  total    total.o
fol.txt  fool.txt  foo.txt    lib.h   -print   total.c  xpilot.motd
[root@flash chaiyen]# gcc -c good.c lucky.c
[root@flash chaiyen]# ls
a.out    fo2.txt   foo.paths  good.c  lib.h    lucky.o  total    total.o
fol.txt  fool.txt  foo.txt    good.o  lucky.c  -print   total.c  xpilot.motd
[root@flash chaiyen]# ar crv lib.a good.o lucky.o
a - good.o
a - lucky.o
```

我們的函式庫 lib.a 已經建立，然後我們再次組譯 total.o，然後執行 total2。

```
[root@flash chaiyen]# gcc -o total2 total.o lib.a
[root@flash chaiyen]# total2
good:大家好 歡喜快樂
lucky: 大家歡喜 天天美好
```

17-2-3 連接函式

```
[root@flash chaiyen]# vi rectangle.c
```

我們可以使用 printf( )函數來列印字串，我們也可以使用 scanf( )函數來掃描我們在終端機所輸入的參數。這個程式為簡單的矩行面積求取。

```
 1 #include<math.h>
 2 main ()
 3 {
 4   float x, y;
 5   printf ("請輸入長和寬以求出矩形的面積\n");
 6   printf ("請輸入長x:");
 7   scanf ("%f", &x);
 8   printf ("請輸入寬y:");
 9   scanf ("%f", &y);
10   printf ("矩形面積是:%f\n", x * y);
11 }
```

我們使用 gcc 組譯。

```
[root@flash chaiyen]# gcc -o rect rectangle.c
```

這是我們執行的情況。

```
[root@flash chaiyen]# rect
請輸入長和寬以求出矩形的面積
請輸入長x:5
請輸入寬y:6
矩形面積是:30.000000
```

我們現在要求的是 x 的 y 次方，它的主程式 power.c 將包含在/lib/libm.a 的數學函數。我們使用 vi 來編輯 power.c。

```
[root@flash chaiyen]# vi power.c
```

我們在第一行表示要包含數學函數 math.h。我們在第十行使用了 pow(x,y)求取 x

的 y 次方函數。

```
#include<math.h>
main ()
{
  float x, y;
  printf ("請輸入x的y次方，以便求x的y次方的值\n");
  printf ("請輸入x:");
  scanf ("%f", &x);
  printf ("請輸入y:");
  scanf ("%f", &y);
  printf ("x的y次方: %f\n",pow((double)x,(double)y));
}
```

當我們使用 gcc –o power power.c 來組譯發生 pow 函式找不到，這是因為在編譯時，需將函數 math 的路徑給連接。

```
[root@flash chaiyen]# gcc -o power power.c
/tmp/ccotxJe2.o: In function `main':
/tmp/ccotxJe2.o(.text+0x79): undefined reference to `pow'
collect2: ld returned 1 exit status
```

因此我們使用 gcc –o power –lm power.c 來連接數學函數 math。-lm 參數指的是 libm．a 數學函數，-l 是參數的選項。

```
[root@flash chaiyen]# gcc -o power -lm power.c
```

這樣就可以執行了。

```
[root@flash chaiyen]# power
請輸入x的y次方，以便求x的y次方的值
請輸入x:3
請輸入y:2
x的y次方: 9.000000
```

17-3 模組化的設計工具—make

大部份 C 語言開發的軟體都是由多種原始檔(．c 和．h)所組成。模組化的軟體可以形成較小的程式就可以作到很多的事，而且程式開發者也不用花費太多時間在編

輯、編譯、測試和除錯。模組化讓我們可以只需編譯那些只經過修改的模組，而不是整個檔案都需要重新編譯過，而且模組化可以讓我們將需要隱藏的資料隱藏，這就是物件導向資料隱藏的特色。

然而，模組化軟體的時作也有它方便之處。例如我們需了解所有構成系統的檔案, 和這些檔案相依的關係, 而且我們在最後執行時需了解哪一些檔案已經被修改了。

我們的 Linux 有個很強大的工具 make，它可以管理我們多個模組的編譯成可執行檔。Make 工具提供我們有彈性的機制來建立大型的軟體計畫。Make 工具讀取 makefile，而 makefile 會描述軟體系統模組的相依情況。Make 工具使用 makefile 模組的相依和各個元件被修改的時間來降低重新編譯的時間。下面是 make 指令。

make 工具是依據 makefile 來更新檔案間的相依性。

語法：

指令：make [參數] [目標] [巨集定義]

參數：

-d：顯示除錯資訊。

-f 檔案：這個參數選項允許我們指示 make 工具從檔案來讀取和從檔案來指定內部的相依情況。如果我們沒有指定則檔案會被設定成 makefile 或 Makefile。

-h：顯示所有選項的簡略描素。

-n：沒有執行 makefile 的指令，而只是顯示它們。

-s：執行但沒有顯示任何訊息。

這是 make 的規則，它指示我們相依的檔案、建立執行檔與目的檔的指令。目標串列是以空隔隔開的的目標檔案串列，相依檔案串列也是以空隔隔開而且每一個檔案皆有相依性質。每一個在指令串列中的指令都有<Tab>字元在前面。

語法：

目標串列：相依檔案串列

<Tab> 指令串列

makefile 是由數個 make 的規則所組成，來建立可執行檔。Make 工具使用在

makefile 的規則來決定哪一個檔案需被重新編譯與連接來重新建立可執行檔。例如我們修改了標頭檔(. h)make 工具將重新編譯所有包含此標頭檔的檔案。

我們使用 vi 來編輯 makefile。

```
[root@flash chaiyen]# vi makefile
```

#這在 makefile 是當作註解。我們在第四行一開始為<Tab>字元(要空這麼多格)。第三行和第四行為 make 規則的語法。

```
1 #這是makefile的解說
2 #每一個指令開始前爲<Tab>
3 power: power.c
4         gcc power.c -o power -lm
```

我們執行 make，但顯示了 power.c 並沒有被更改 power is up to date。

```
[root@flash chaiyen]# make
make: `power' is up to date.
```

power 和 power.c 被製造出的時間都一樣。

```
-rwxr-xr-x    1 root     root        13980  8月  8 22:48 power
-rw-r--r--    1 root     root          235  8月  8 22:48 power.c
```

為了要能從新建立可執行的檔案，我們將改變 power.c 所更新的時間。因此我們始用 touch 來修改目前的 power.c 的時間。

```
[root@flash chaiyen]# touch power.c
```

```
-rwxr-xr-x    1 root     root        13980  8月  8 22:48 power
-rw-r--r--    1 root     root          235  8月  8 23:39 power.c
```

我們使用 touch power.c 將 power.c 已的更新時間改為現在，這時就可使用 make 了。

```
[root@flash chaiyen]# make
gcc power.c -o power -lm
```

我們將剛才的 power.c 分成兩個檔案，分別是 power1.c 和 compute.c。compute.c 內含 compute 函數。而 power1.c 為主程式，它會呼叫 compute.c 裏面的 compute 函數。為了產生 power1 的執行檔，我們需要分別獨立的編譯它們，再用 link 連結它們。

這是我們的主程式 power1.c，我們在最後一行使用 compute(x,y)函數來呼叫 compute.c 的函數。

```
[root@flash chaiyen]# cat power1.c
double compute(double x,double y);
main ()
{
  float x, y;
  printf ("請輸入x的y次方，以便求x的y次方的值\n");
  printf ("請輸入x:");
  scanf ("%f", &x);
  printf ("請輸入y:");
  scanf ("%f", &y);
  printf ("x的y次方: %f\n",compute(x,y));
}
```

這是 compute.c 的內容。Compute( )函數會計算 x 的 y 次方。

```
[root@flash chaiyen]# cat compute.c
#include<math.h>
double compute(double x,double y)
{
   return (pow((double)x,(double)y));
}
```

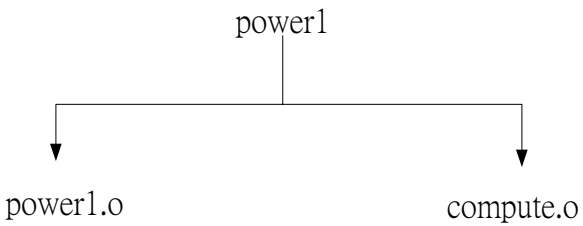
我們使用 gcc –c compute.c power1.c 來編譯成目的檔 compute.o 和 power1.o。

```
[root@flash chaiyen]# gcc -c compute.c power1.c
```

這是我們將 compute.o、power1.o 和數學函數/lib/libm.a 作連結，形成執行檔 power1。

```
[root@flash chaiyen]# gcc compute.o power1.o -o power1 -lm
```

power1.o 和 compute.o 相依的關係是很簡單的，當我們要建立 power1 執行檔時，就需要它們。假如 power1.c 或 compute.c 被更新或修改時，我們就需要重新建立 power1 執行檔而下列就是我們檔案的相依關係。



make 的檔案相依樹

我們知到 pwer1 是由 power.o、compute.o 和數學函數/usr/lib/libm.a 所組成。

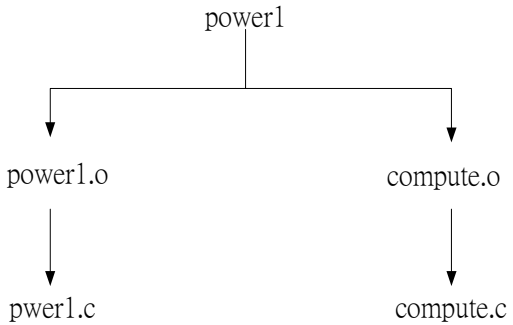
所以 make 的規則和相依的關係為下列。

```
power1: power1.o compute.o
        gcc power1.o compute.o -o power1 -lm
```

我們也知道目的檔 power.o 是由 power.c 所建立，而 compute.o 是由 compute.c 所建立所以它們 make 的規則和相依關係為下列。

```
power1.o:power1.c
        gcc -c power.c

compute.o:compute.c
        gcc -c compute.c
```



make 的檔案相依樹2

這是最後的 makefile，我們在這裏 make 的規則是以空一行來顯示，讓讀者可以更清楚了解。

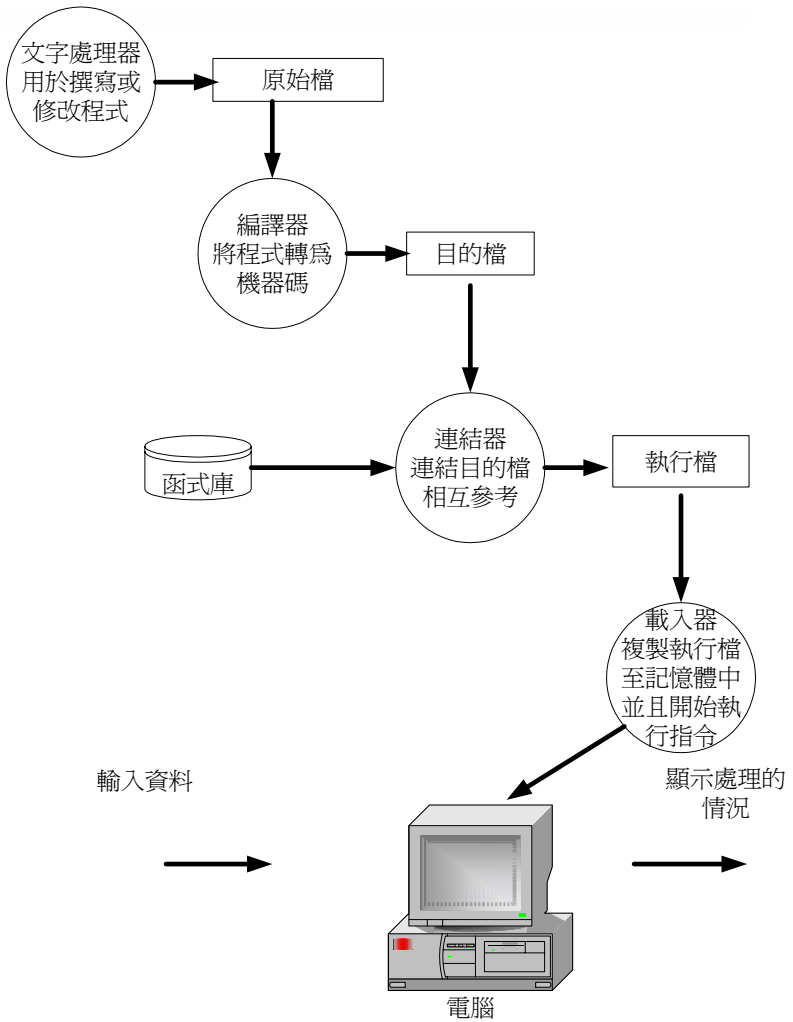
```
power1: power1.o compute.o
        gcc power1.o compute.o -o power1 -lm

power1.o:power1.c
        gcc -c power.c

compute.o:compute.c
        gcc -c compute.c
```

我們依據我們的 makefile 來執行 make 來產生 power1 的執行檔。

```
[root@flash chaiyen]# make
gcc -c power.c
gcc -c compute.c
gcc power1.o compute.o -o power1 -lm
```



我們也可以將數個檔案給組成一個大型的軟體，在這裏有五個標頭檔和三個 C 語言的原始檔，讓我們練習如何來始用 make，以及如何來撰寫 make 的規則 makefile。

這是 main.c 的檔案，它包含了 main.h、compute.h、input.h 的標頭檔，在第十一行它呼叫了 compute( )函數。

```
#include "main.h"
#include "compute.h"
#include "input.h"

main()
{
  double x, y;
  printf ("請輸入x的y次方，以便求x的y次方的值\n");
  x=input(prompt1);
  y=input(prompt2);
  printf ("x的y次方: %f\n",compute(x,y));
}
```

這是我們 main.h 的標頭檔。在第二行它定義了 prompt1 為提示"請輸入 x 的值。在第三行它定義了 prompt2 為提示"請輸入 y 次方的值："。

```
/*使用者的提示*/
#define prompt1 "請輸入x的值:"
#define prompt2 "請輸入y次方的值:"
```

這是 compute.h 的標頭檔。它宣告了 compute( )函數的原型為 double。

```
/* 宣告compute( )函數的原型 */

double compute(double, double);
```

這是 input.c。它在第一行包含了 input.h 標頭檔。Input( )函數會列印出所輸入的字串 s，並且會掃描我們從鍵盤所輸入的值，然後再回傳我們所輸入的值。

```
#include "input.h"

double input(char *s)
{
  float x;

  printf ("%s",s);
  scanf ("%f", &x);
  return (x);
}
```

這是 input.h 的標頭檔，它宣告了 input( )函數的原型。

```
/*宣告input( )函數的型態*/
double input (char *);
```

這是 compute.c，它包含了 compute.h 檔。

```
#include<math.h>
#include "compute.h"

double compute(double x,double y)
{
   return (pow((double)x,(double)y));
}
```

這是 compute.h 的標頭檔，它宣告了 compute( )函數的原型。

```
/* 宣告compute( )函數的原型 */

double compute(double, double);
```

這是 power2 的 makefile。為了產生可以執行的程式，我們需要將 main.o、compute.o、input.o、/usr/lib/libm 組成連結成 power2 可執行檔。而要編譯成 main.o 目的檔，我們需要 main.c、main.h、input.h、和 compute.h 的檔案來組成編譯。而要編譯成 input.o 目的檔，我們需要 input.c、input.h 來組成編譯。而要編譯成 compute.o 目的檔，我們需要 compute.c、compute.h 來組成編譯。

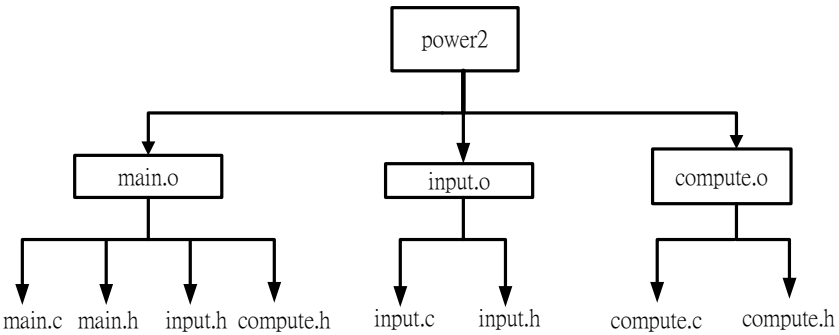
```
power2: main.o compute.o input.o
        gcc main.o compute.o input.o -o power2 -lm

main.o: main.c main.h input.h compute.h
        gcc -c main.c

input.o: input.c input.h
        gcc -c input.c

compute.o: compute.c compute.h
        gcc -c compute.c
```

我們 makefile 的檔案相依情況，而這是我們 make 的規則。



make 的檔案相依樹3

這是我們使用 make 指令執行的情況。

```
[root@flash 1-3]# make
gcc -c main.c
gcc -c compute.c
gcc -c input.c
gcc main.o compute.o input.o -o power2 -lm
```

我們可以指定我們 make 的規則是放到我們所指定的檔案, 在這裏我們要把 make 的規則放到 my.makefile 中。我們可以使用 make –f my.makefile 的指令。my.makefile 就是放置 make 的規則。

```
[root@flash my.makefile]# make
make: *** No targets specified and no makefile found.  Stop.
[root@flash my.makefile]# make -f my.makefile
gcc -c main.c
gcc -c compute.c
gcc -c input.c
gcc main.o compute.o input.o -o power2 -lm
[root@flash my.makefile]# ls
compute.c  compute.o  input.h  main.c  main.o       power2
compute.h  input.c    input.o  main.h  my.makefile
```

Make 工具支援我們使用簡單的巨集來代替文字內容。我們在使用巨集前，需要先定義它，它們巨集通常都是放在 makefile 的最前面。我們可以用下列的語法來使用巨集。

語法；

指令：巨集名稱= 特定文字內容

或者

define 巨集名稱

特定文字內容

endef

$(巨集名稱)取代了我們所指定的特定文字內容。Make 工具也有內定的巨集，像是 CFLAGS 巨集。CFLAGS 巨集是代表最佳化的參數-O。我們將介紹使用者自訂的巨集，而且 makefile 將顯示我們可以使用 shell 的其它指令。

這是我們 my.makefile 的檔案。我們在第一行使用 cc 代替 gcc。使用 options 代替參數 o 和參數 O3(是英文大寫 O)。我們在第三行使用 objects 代替 main.o input.o compute.o 的文字內容。我們在第四行使用 sources 代替 main.c input.c compute.c 的文字內容。我們在第五行使用 headers 代替 main.h input.h compute.h。我們在第七行使用$(objects)代替 main.o input.o compute.o 的文字內容。我們在第八行使用 $(cc)代替 gcc。第十七行是將檔案壓縮成 all.tar 檔，第 20 行是將目的檔給清除。

```
1 cc=gcc
2 options = -O3 -o
3 objects = main.o input.o compute.o
4 sources = main.c input.c compute.c
5 headers = main.h input.h compute.h
6
7 power3 : $(objects)
8         $(cc) $(options) power3 $(objects) -lm
9
10 main.o : main.h input.h compute.h
11
12 input.o : input.h
13
14 compute.o : compute.h
15
16 all.tar : $(sources) $(headers) my.makefile
17         tar -cvf - $(sources) $(headers) my.makefile > all.tar
18
19 clean :
20         rm *.o
```

我們使用 make –f my.makefile 來執行，就可以編譯出 power3 的執行檔。在執行前我們使用 touch my.makefile 來改變和更新日期。

```
[root@flash macro]# make -f my.makefile
cc     -c -o main.o main.c
cc     -c -o input.o input.c
cc     -c -o compute.o compute.c
gcc -O3 -o power3 main.o input.o compute.o -lm
```

這時會顯是 compute.o、input.o 和 mani.o 的檔案。我們第十九行 clean 的規則 rm.*因為和其它的規則並沒有相依，所以並不會執行。

```
[root@flash macro]# ls
compute.c  compute.o  input.c  input.o  main.h  my.makefile      power3
compute.h  good       input.h  main.c   main.o  my.makefile.bak
```

我們輸入 power3 就可以執行程式。

```
[root@flash macro]# power3
請輸入x的y次方，以便求x的y次方的值
請輸入x的值:2
請輸入y次方的值:3
x的y次方: 8.000000
```

我們可以使用 make –f my.makefile clean 來執行 my.makefile 的 clean 規則。這時就會執行第二十行的 rm *.o 的指令了。

```
[root@flash macro]# make -f my.makefile clean
rm *.o
```

這時所有目的檔已經被清除了。

```
[root@flash macro]# ls
compute.c  good     input.h  main.h       my.makefile.bak
compute.h  input.c  main.c   my.makefile  power3
```

我們也可以使用 make –f my.makefile all.tar 來執行第十六 all.tar 的規則。all.tar 會將檔案給包裝成 all.tar 檔。tar –cvf 為包裝及檔案備份的指令及參數。

```
[root@flash macro]# make -f my.makefile all.tar
tar -cvf - main.c input.c compute.c main.h input.h compute.h my.makefile > all.t
ar
main.c
input.c
compute.c
main.h
input.h
compute.h
my.makefile
[root@flash macro]# ls
all.tar    compute.h  input.c  main.c  my.makefile      power3
compute.c  good       input.h  main.h  my.makefile.bak
```

make 有幾個特別的內建巨集可以使 makefile 更為好用，下面就是幾個內部的巨集。

內部巨集	說明
$@	目的檔名稱
$?	比目的檔還需要經常更改的檔名清單
$<	檔名(不包含副檔名)
$*	沒有副檔名之檔名

這是我們的 my.makefile。我們在第一到第三行使用 define 來命名巨集 cc 為 gcc。我們在第十行使用@echo 來顯示字串，echo 是 shell 的指令。@是表示不顯示指令。我們在第十五行使用 rm *.o 來將所有目的檔清除。我們在第十六行使用 ls 來顯示目錄的檔案。

```
 1 define cc
 2 gcc
 3 endef
 4 options = -O3 -o
 5 objects = main.o input.o compute.o
 6 sources = main.c input.c compute.c
 7 headers = main.h input.h compute.h
 8
 9 complete : power3
10         @echo "建立完成"
11 power3 : $(objects)
12         $(cc) $(options) power3 $(objects) -lm
13         @ls
14         @echo "建立完成"
15         @rm *.o
16         @ls
17 main.o : main.h input.h compute.h
18
19 input.o : input.h
20
21 compute.o : compute.h
```

我們使用 make –f my.makefile 來執行 make。一開始第十三行的 ls 會顯示目的檔，但經過第十五行的 rm *.o 就可以將目的檔給清除。第十六行的 ls 就不會顯示出目的檔了*.o 了。

```
[root@flash macro1]# make -f my.makefile
cc     -c -o main.o main.c
cc     -c -o input.o input.c
cc     -c -o compute.o compute.c
gcc -O3 -o power3 main.o input.o compute.o -lm
all.tar    compute.h  good     input.h  main.c  main.o       my.makefile.bak
compute.c  compute.o  input.c  input.o  main.h  my.makefile  power3
建立完成
all.tar    compute.h  input.c  main.c  my.makefile      power3
compute.c  good       input.h  main.h  my.makefile.bak
建立完成
[root@flash macro1]# power3
請輸入x的y次方，以便求x的y次方的值
請輸入x的值:2
請輸入y次方的值:2
x的y次方: 4.000000
```

17-4 從函式庫建立、修改或抽取

Linux 作業系統允許我們將許多的目的檔存入單一的函式庫檔，而且它允許我們使用這函式庫檔而不是只有目地檔而已。

ar 工具可以建立或修改備存檔，或者從備存檔中抽取檔案出來。

語法：

指令：ar 參數 [成員檔案串列]

-d：刪除備份檔中的成員檔案。

-m：變更成員檔案在備份檔中的順序。

-p：顯示備份檔中的成員檔案內容。

-q：將檔案附加在備份檔後面，而不檢查備份檔是否有重複的成員檔案。

-r：將檔案插入備份檔中。

-t：顯示備份檔中所包含的檔案。

-x：自備份檔中取出成員檔案。

a 成員檔案：將檔案插入備份檔中指定的成員檔案後。

b 成員檔案：將檔案插入備份檔中指定的成員檔案前。

c：建立備份檔。

o：保留備份檔中的日期。

s：建立備份檔中的符號表。

v：顯示執行時的詳細資訊。

我們現在要把 compute.o 和 input.o 包裝成 mathlib.a 一個包裝檔案。

```
[root@flash 1-3]# ls
compute.c  compute.o  input.h  macro1  main.h  makefile     power2
compute.h  input.c    input.o  main.c  main.o  my.makefile
```

我們使用 ar –rv mathlib.a compute.o input.o 將 compute.o 和 input.o 包裝成 mathlib.a 一個檔案。

```
[root@flash 1-3]# ar -rv mathlib.a compute.o input.o
a - compute.o
a - input.o
[root@flash 1-3]# ls
compute.c  compute.o  input.h  macro1  main.h  makefile    my.makefile
compute.h  input.c    input.o  main.c  main.o  mathlib.a   power2
```

我們使用 ar –rv lib.a input.h main.h compute.h 將三個 input.h main.h compute.h 包裝成一個 lib.a 的檔案。

```
[root@flash 1-3]# ar -rv lib.a input.h main.h compute.h
a - input.h
a - main.h
a - compute.h
```

我們使用 ar –p libh.a 來顯示我們 lib.a 的內容。

```
[root@flash 1-3]# ar -p lib.a
#include "input.h"

double input(char *s)
{
  float x;

  printf ("%s",s);
  scanf ("%f", &x);
  return (x);
}
/*使用者的提示*/
#define prompt1 "請輸入x的值:"
#define prompt2 "請輸入y次方的值:"
/* 宣告compute( )函數的原型 */

double compute(double, double);
```

我們使用ar –rbv main.h lib.a input.c將input.c的檔案插入到main.h的檔案之前。

```
[root@flash 1-3]# ar -rbv main.h lib.a input.c
a - input.c
```

我們使用 ar –t lib.a 來顯示 lib.a 備份檔中所包含的內容。

```
[root@flash 1-3]# ar -t lib.a
input.h
input.c
main.h
compute.h
```

我們使用 ar –dv lib.a 將備份檔中的 input.h 的檔案給刪除。

```
[root@flash 1-3]# ar -dv lib.a input.h
d - input.h
[root@flash 1-3]# ar -t lib.a
input.c
main.h
compute.h
```

我們使用 ar r mathlib.a input.o compute.o 將 input.o 和 compute.o 包裝成 mahtlib.a 檔。

```
[root@flash 1-3]# ar  r mathlib.a input.o compute.o
```

假如我們的 lib.a 已經存在我們的目錄中，我們編譯時就可以將它連結。

```
[root@flash 1-3]# gcc -o power5 main.c -lm mathlib.a
```

我們使用 power5 就可以來執行我們的程式。

```
[root@flash 1-3]# ls
compute.c  compute.o  input.h  lib.a   macro1  main.h  makefile   my.makefile
compute.h  input.c    input.o  libh.a  main.c  main.o  mathlib.a  power5
[root@flash 1-3]# power5
請輸入x的y次方，以便求x的y次方的值
請輸入x的值:2
請輸入y次方的值:3
x的y次方: 8.000000
```

我們使用 ar q mathlib.a input.o compute.o 就可以將 input.o 和 compute.o 加到 mathlib.a 的檔案中。

```
[root@flash 1-3]# ar q mathlib.a input.o compute.o
```

我們使用 ar r newlib.a `ls *.o`就可以將目錄中所有的目的檔加入到 newlib.a 的檔案中。 . a 的副檔名通常為函式檔。

```
[root@flash 1-3]# ar r newlib.a `ls *.o`
```

我們使用 ar t mathlib.a 就可以顯示 mathlib.a 所包裝的檔案。

```
[root@flash 1-3]# ar t  mathlib.a
compute.o
input.o
input.o
compute.o
```

我們使用 ar d mathlib.a input.o 就可以將 input.o 給刪除。

```
[root@flash 1-3]# ar d mathlib.a input.o
[root@flash 1-3]# ar t mathlib.a
compute.o
input.o
compute.o
```

我們可以使用 ar x mathlib.a compute.o 將 compute.o 目的檔從 mahtlib.a 的檔案中抽取出來。

```
[root@flash 1-3]# ls
compute.c  input.c  lib.a   macro1  main.h    mathlib.a    newlib.a
compute.h  input.h  libh.a  main.c  makefile  my.makefile  power5
[root@flash 1-3]# ar x mathlib.a compute.o
[root@flash 1-3]# ls
compute.c  compute.o  input.h  libh.a  main.c  makefile   my.makefile  power5
compute.h  input.c    lib.a    macro1  main.h  mathlib.a  newlib.a
```

這是我們的 makefile。我們在第三行使用 ar 指令將 compute.o 和 input.o 包裝到 mathlib.a 的函式檔。

```
power2: main.o compute.o input.o
        gcc main.o compute.o input.o -o power2 -lm
        ar qv mathlib.a compute.o input.o
        rm *.o
        @echo "目的檔已經全部包裝成mathlib.a檔"
        @echo "已經將目的檔清除"
main.o: main.c main.h input.h compute.h
        gcc -c main.c

input.o: input.c input.h
        gcc -c input.c

compute.o: compute.c compute.h
        gcc -c compute.c
```

這是我們執行 make 的情況。我們先使用 touch makefile 來將 makefile 給更新。

```
[root@flash 1-3]# touch makefile
[root@flash 1-3]# make
gcc -c main.c
gcc -c input.c
gcc main.o compute.o input.o -o power2 -lm
ar qv mathlib.a compute.o input.o
a - compute.o
a - input.o
rm *.o
目的檔已經全部包裝成mathlib.a檔
已經將目的檔清除
[root@flash 1-3]# ls
compute.c  input.c  lib.a   macro1  main.h    makefile.bak  my.makefile  power2
compute.h  input.h  libh.a  main.c  makefile  mathlib.a     newlib.a     power5
```

17-4-1 ranlib 指令

ranlib 指令可以增加或產生順序內容的表格在我們的包裝備份檔中。

語法：

指令：ranlib [-vV] 包裝檔串列

ranlib 指令產生內容的表格在我們的包裝備份檔 mathlib.a 中。

```
[root@flash ranlib]# ranlib mathlib.a
```

我們可以顯示 mathlib.a 的版本編號。

```
[root@flash ranlib]# ranlib -v mathlib.a
GNU ranlib 2.11.93.0.2 20020207
Copyright 2002 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License.  This program has absolutely no warranty.
```

17-4-2 顯示函式資訊

我們可以使用 nm 工具來顯示目的檔和函數的符號表(如名稱、形態、檔案大小、和進入點)。nm 指令在目的檔或函式以每一行顯示每一個函數或變數。這讓我們了解函數和變數相依的情況。這讓我們可以方便的除錯。

nm 可來顯示目的檔和函數的符號。

語法：

指令：nm　[參數]　[目的檔串列]

-D：顯示動態符號(當有使用動態函式庫時)。

-V：顯示 nm 的版本編號。

-f 格式：顯示以 bsd、sysv 或 posix 的格式輸出。

-g：只有顯示外部符號。

-l：找尋和顯示每個符號表的檔名和編號。

-n,-v：以位址排序外部符號表。

-u：顯示沒有定義的符號。

我們使用 nm –V 來顯示 nm 指令的版本。

```
[root@flash 1-4]# nm -V
GNU nm 2.11.93.0.2 20020207
Copyright 2002 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License.  This program has absolutely no warranty.
```

我們使用 nm input.o 來顯示 input.o 的資訊。

```
[root@flash 1-4]# nm input.o
00000000 t gcc2_compiled.
00000000 T input
         U printf
         U scanf
```

我們使用 nm –n imput.o 來顯示以位址排序外部的符號表。

```
[root@flash 1-4]# nm -n input.o
         U printf
         U scanf
00000000 t gcc2_compiled.
00000000 T input
```

我們使用 nm –f sysv –l input.o 來以 System V 的格式顯示。在 Class 欄位中，U 代表未定義的符號，通常代表函式庫中的函式、外部變數、或在物件模組中的函式。

```
[root@flash 1-4]# nm -f sysv -l input.o


Symbols from input.o:

Name                  Value   Class        Type         Size   Line  Section

gcc2_compiled.      |00000000|   t  |                  |      |     |
input               |00000000|   T  |                  |      |     |
printf              |        |   U  |                  |      |     |   (null):0
scanf               |        |   U  |                  |      |     |   (null):0
```

17-5 軟體的版本控制

版本的控制包含當一個使用者在編輯檔案時，要將檔案給鎖住，以免其他使用者使用、建立不同的檔案版本、幫助辨別檔案的修正處、儲存和解析出不同的檔案版本、合併多個相同的檔案來建立新的最後檔案、可以接觸到軟體所有檔案的最早版本、而且限制在系統上的其他使用者的使用權力。

17-5-1 修正控制系統(RCS)

有好幾個 Linux 軟體允許控制我們的檔案，而很多的工具都是使用 RCS 來當作它們的主要核心。RCS 修正控制系統來讓我們作修改和控制的工作。所有 RCS 的工

具都是放在/usr/bin 的目錄上。

ci 的指令通常用作建立和管理 RCS 的檔案。

語法：

指令：ci [參數] [檔案名稱]

參數：

-f [rev]：檢查修正版是否不同於先前的版本。

-l [rev]：檢查和鎖住修正版。

-r 版本：檢查被當作修正的 ver 版檔案。

-u：記錄檔案但保留可讀版本。

我們使用 ci 來建立 RCS 的歷史檔，稱為 RCS/input.c,v。當我們有新的檔案修正版時，ci 提示我們給予檔案記錄。當我們要終止 ci 時，我們可以使用<Ctrl-D>或(.)。

當我們建立 RCS/input.c,v 的檔案後，ci 指令刪除原始的檔案 input.c。

```
[root@flash 1-5]# ci input.c
input.c,v  <--  input.c
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> "吳佳諺的input.c檔案"
>> .
initial revision: 1.1
done
```

我們建立了 input.c,v。

```
[root@flash 1-5]# ls -l
total 108
-r-xr--r--    1 root     root          122  8月 10 15:57 compute.c
-r-xr--r--    1 root     root          325  8月 10 15:53 compute.c,v
-rwxr--r--    1 nobody   nobody         67  8月  9 17:05 compute.h
-rw-r--r--    1 root     root          852  8月 10 15:57 compute.o
-r-xr--r--    1 root     root          324  8月 10 16:47 input.c,v
-rwxr--r--    1 nobody   nobody         54  8月  9 17:06 input.h
```

我們使用 ci –u compute.c 來記錄檔案和保留只可讀寫的的備份檔。

```
[root@flash 1-5]# ci -u compute.c
compute.c,v  <--  compute.c
ci: compute.c,v: no lock set by chaiyen
```

我們使用 co 指令來檢查 RCS 的檔案。co 工具經常使用來檢查檔案和將它儲存在相關的工作檔中。

語法：

指令：co [參數] [檔案串列]

參數：

-l：在編輯鎖住的情況下檢查檔案。

-r 版本：檢查指定檔案的 ver 版本。

-u [版本]：檢查指定檔案的唯讀版本。

-v：顯示 RCS 的版本編號。

我們使用 co 指令檢查唯讀的備份檔。關於檔案的內容與控制的資訊都放在 RCs 檔，compute.c,v。

```
[root@flash 1-5]# co -l compute.c
compute.c,v  -->  compute.c
revision 1.1 (locked)
done
```

這是我們再次編輯的第 1.2 版本。

我們使用 chmod 700 改變權限 compute.c 再來編輯改變我們的檔案。

```
[root@flash 1-3]# ci -u compute.c
compute.c,v  <--  compute.c
enter description, terminated with single '.' or end of file:
NOTE: This is NOT the log message!
>> '這是第一版'.
>> .
initial revision: 1.1
done
[root@flash 1-3]# chmod 700 compute.c
[root@flash 1-3]# vi compute.c
```

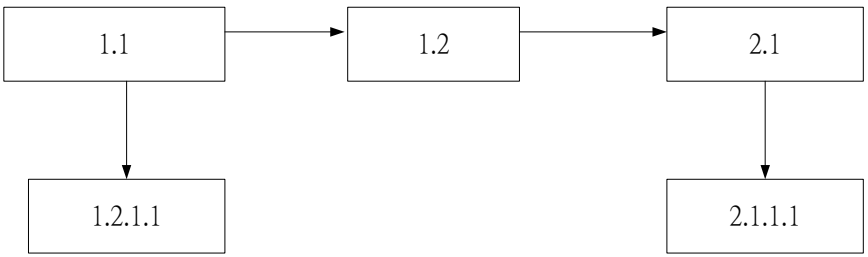
我們使用 rcs –l 指令來鎖住 compute.c 這樣才能使用 ci 指令來編輯。

```
[root@flash 1-3]# ci -u compute.c
compute.c,v  <--  compute.c
ci: compute.c,v: no lock set by chaiyen
[root@flash 1-3]# rcs -l compute.c
RCS file: compute.c,v
1.1 locked
done
[root@flash 1-3]# ci -u compute.c
compute.c,v  <--  compute.c
new revision: 1.2; previous revision: 1.1
enter log message, terminated with single '.' or end of file:
>> '這是1.2版'
>> .
done
```

我們使用 rcs –l compute.c 指令來鎖住 compute.c 的檔案。我們使用 ci –r2 指令來建立 compute.c 第二版。

```
[root@flash 1-3]# rcs -l compute.c
RCS file: compute.c,v
1.2 locked
done
[root@flash 1-3]# ci -r2 compute.c
compute.c,v  <--  compute.c
new revision: 2.1; previous revision: 1.2
enter log message, terminated with single '.' or end of file:
>> '這是第二版'
>> .
done
```

這是 compute.c 的版本樹。



compute.c的版本樹

rcs 工具允許我們 RCS 檔案的控制，它允許我們更改 RCS 檔案的屬性。

語法：

指令：rcs [參數] 檔案串列

參數：

-alogins：允許在登錄情況下的使用者檢查和記錄檔案串列中可編輯的版本。

-e[logins]：不允許在登錄情況下的使用者檢查檔案串列中可編輯的版本。

-l[rev]：檢查檔案修正的 rev 版，且鎖住它，但沒有覆寫它。並沒有指定使用最新的 rev 版本。

-o 範圍：移除指定範圍的版本，範圍為 rev1：rev2。

-u[rev]：沒有鎖住 rev 版本。禁止檔案的改變。

當我們使用 co –l 檢查檔案時，它會鎖住，然後我們可以使用 ci 編輯和改變這個檔案。假如檔案沒有被鎖住，ci 指令沒有辦法安裝和改變這個檔案。我們可以使用 rcs –u 的指令來檢查編輯檔案而不用鎖住它。我們將使用 co –l 檢查 input.c 而且儲存備份檔 input.c.bak。然後我們使用 rcs –u input.c 不鎖住而檢查和改變 input.c。然後我們使用 ci 指令安裝和改變。

```
[root@flash macro1]# rcs -u input.c
RCS file: input.c,v
1.1 unlocked
done
[root@flash macro1]# ci input.c
input.c,v  <--  input.c
ci: input.c,v: no lock set by chaiyen
[root@flash macro1]# co -l input.c
input.c,v  -->  input.c
revision 1.1 (locked)
writable input.c exists; remove it? [ny](n): y
done
[root@flash macro1]# diff input.c input.c.bak
```

我們可以使用 rcs –l 指令來鎖住檔案，而不用從 RCS 目錄檢查或複寫目前的檔案。

我們使用 co –u 來檢視版本。我們編輯改變 compute.c，再來改變版本。

```
[root@flash 1-3]# co -u compute.c
compute.c,v  -->  compute.c
revision 1.2 (unlocked)
done
[root@flash 1-3]# chmod 700 compute.c
[root@flash 1-3]# vi compute.c
#include<math.h>
#include "compute.h"

double compute(double x,double y)
{
   return (pow((double)x,(double)y));
}
/**/
```

我們使用 rcs –l 來所住 compute.c，然後，我們使用 ci –r2 來將 compute.c 修改成第二版。

```
[root@flash 1-3]# clear
[root@flash 1-3]# rcs -l compute.c
RCS file: compute.c,v
1.2 locked
done
[root@flash 1-3]# ci -r2 compute.c
compute.c,v  <--  compute.c
new revision: 2.1; previous revision: 1.2
enter log message, terminated with single '.' or end of file:
>> '這是第二版'
>> .
done
```

我們可以使用 rcs –o1.1:1.2 將 1.1 到 1.2 的 compute.c 的版本刪除。

```
[root@flash 1-3]# rcs -o1.1:1.2 compute.c
RCS file: compute.c,v
deleting revision 1.2
deleting revision 1.1
done
```

我們使用 chmod 700 來改變 compute.c 的權限，我們使用 vi 來編輯改變 compute.c 檔。

```
[root@flash 1-3]# chmod 700 compute.c
[root@flash 1-3]# vi compute.c
#include<math.h>
#include "compute.h"

double compute(double x,double y)
{
   return (pow((double)x,(double)y));
}
/*123*/
```

我們使用 rcs –l compute.c 來鎖住檔案。並且使用 ci –u compute.c 來修正版本。

```
[root@flash 1-3]# rcs -l compute.c
RCS file: compute.c,v
2.1 locked
done
[root@flash 1-3]# ci -u compute.c
compute.c,v  <--  compute.c
new revision: 2.2; previous revision: 2.1
enter log message, terminated with single '.' or end of file:
>> '這是2.2版'
>> .
done
```

我們使用 rcs –o2.1 將舊版的 compute.c 給刪除。

```
[root@flash 1-3]# rcs -o2.1 compute.c
RCS file: compute.c,v
deleting revision 2.1
done
```

我們可以使用 rlog 指令來顯示檔案修正控制系統 RCS 的版本。

語法：

指令：rlog [參數] 檔案串列

參數：

-L：檢查檔案被修改的資訊。

-R：顯示檔案的名稱。

-l[users]：顯示被使用者鎖住的檔案資訊。

-r[revs]：顯示修正版本的資訊。

rlog 指令顯示我們 compute.c 所有修正控制系統 RCS 的版本資訊。

```
[root@flash 1-3]# rlog compute.c

RCS file: compute.c,v
Working file: compute.c
head: 2.2
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 1;     selected revisions: 1
description:
'這是第一版'.
----------------------------
revision 2.2
date: 2002/08/11 02:38:04;  author: chaiyen;  state: Exp;
'這是2.2版'
```

我們可以使用 rcsdiff 指令來顯示出相同檔案不同版本的不同之處。

語法：

指令：rcsdiff [參數] 檔案串列

參數：

-r 版本：指定比較的版本。

我們使用 rcsdiff –r2.2 –r2.3 來比較 compute.c 檔案 2.2 修正版和 2.3 修正版的不同。下面顯示了不同的第方。

```
[root@flash 1-3]# rcsdiff -r2.2 -r2.3 compute.c
=============================================================
RCS file: compute.c,v
retrieving revision 2.2
retrieving revision 2.3
diff -r2.2 -r2.3
8c8
< /*13*/
---
> /*12`2`3*/
```

我們使用 rcsmerge 指令來合併不同的版本。

語法：

指令：rcsmerge [參數] 檔案

參數：

-r 版本：指定的版本。

-p：輸出改變到標準輸出，而不是目前的版本。

我們使用 rcsmerge –r.2.2 –p compute.c >merged_compute.c 將目前所檢查的版本和 compute.c 檔案 2.2 的版本作合併相異的地方並且將它輸出到 merged_compute.c。

```
[root@flash 1-3]# rcsmerge -r2.2 -p compute.c > merged_compute.c
RCS file: compute.c,v
retrieving revision 2.2
retrieving revision 2.3
Merging differences between 2.2 and 2.3 into compute.c; result to stdout
```

我們使用 rcsmerge –r.2.2 –r2.3 –p compute.c > merged2_compute.c 將 compute.c 檔案 2.2 版本和 compute.c 檔案 2.3 的版本作合併相異的地方並且將它輸出到 merged2_compute.c。

```
[root@flash 1-3]# rcsmerge -r2.2 -r2.3 -p compute.c > merged2_compute.c
RCS file: compute.c,v
retrieving revision 2.2
retrieving revision 2.3
Merging differences between 2.2 and 2.3 into compute.c; result to stdout
```

我們使用 rcsmerge –r.2.2 –r2.3 –p compute.c > merged2_compute.c 將 compute.c 檔案 2.2 版本和 compute.c 檔案 2.3 的版本作合併相異的地方並且將它輸出到 compute.c。。

```
[root@flash 1-3]# rcsmerge -r2.2 -r2.3  compute.c
RCS file: compute.c,v
retrieving revision 2.2
retrieving revision 2.3
Merging differences between 2.2 and 2.3 into compute.c
```

使用 RCS 檔案的權限。

我們使用 rcs –a 使用者，來讓指定的使用者可以有權限使用我們的 RCS 版本修正控制系統。我們使用 rcs –achaiyen,justin，來讓指定的使用者 chaiyen 和 justin 可以有權限使用我們的 RCS 版本修正控制系統來修改 compute.c 的版本。

我們使用 rlog –h compute.c 來顯示 compute.c 檔案的版本資訊。

```
[root@flash 1-3]# rcs -achaiyen,justin compute.c
RCS file: compute.c,v
done
[root@flash 1-3]# rlog -h compute.c

RCS file: compute.c,v
Working file: compute.c
head: 2.3
branch:
locks: strict
access list:
        chaiyen
        justin
symbolic names:
keyword substitution: kv
total revisions: 2
```

我們使用 rcs –e 使用者，來刪除用我們的 RCS 版本修正控制系統的使用者。我們使用 rcs –ejustin compute.c，來刪除 justin 的使用者修改 compute.c 的權限。

```
[root@flash 1-3]# rcs -ejustin compute.c
RCS file: compute.c,v
done
[root@flash 1-3]# rlog -h compute.c

RCS file: compute.c,v
Working file: compute.c
head: 2.3
branch:
locks: strict
access list:
        chaiyen
symbolic names:
keyword substitution: kv
total revisions: 2
```

15-5-2 同步版本系統(CVS)

CVS 允許我們在目錄階層的模式組織原始碼，而且檢查所有修改的模組。CVS 允許團體中多個開發者同步的檢查與修改原始碼模組。在 RCS(修正控制系統)下，檔案的修正作業是只允許一位開發者連續的檢查和編輯，而其他的使用者只能以唯讀得方式檢查檔案，CVS 允許多位開發者同步的來修改檔案，而且保證檔案改變的無衝突性。CVS 允許我們以標記在發展和維持軟體的階段，以任何時間點標記軟體的出版和檢查這個版本。

cvs 指令允許我們啟動 CVS 來完成各式的軟體版本控制的工作。

語法：

指令：CVS [cvs-參數] 指令 [指令參數]

參數：

-H：顯示 cvs-指令的使用資訊；假如'cvs-指令'沒有指定，則顯示簡短的 cvs 指令。

-d CVS-root-dir：使用'CVS-root-dir'當作原始碼容器的絕對路徑；覆寫環境變數 CVSROOT 的設定。

-e 編輯器：使用編輯器來編輯，預設為 vi。

-n：嘗式執行所給的 cvs-指令，但不作更改。

-t：幫助我們了解 cvs-指令的語法。

我們可以使用 cvs –H 的指令來顯示簡單的 cvs 指令。cvs * -H 可以顯示所有 cvs 指令的目的。

```
[root@flash chaiyen]# cvs -H
Usage: cvs [cvs-options] command [command-options-and-arguments]
  where cvs-options are -q, -n, etc.
    (specify --help-options for a list of options)
  where command is add, admin, etc.
    (specify --help-commands for a list of commands
     or --help-synonyms for a list of command synonyms)
  where command-options-and-arguments depend on the specific command
    (specify -H followed by a command name for command-specific help)
  Specify --help to receive this message

The Concurrent Versions System (CVS) is a tool for version control.
For CVS updates and additional information, see
    the CVS home page at http://www.cvshome.org/ or
    Pascal Molli's CVS site at http://www.loria.fr/~molli/cvs-index.html
```

cvs * -H 可以顯示所有 cvs 指令的目的。

```
[root@flash chaiyen]# cvs * -H

CVS commands are:
        add             Add a new file/directory to the repository
        admin           Administration front end for rcs
        annotate        Show last revision where each line was modified
        checkout        Checkout sources for editing
        commit          Check files into the repository
        diff            Show differences between revisions
        edit            Get ready to edit a watched file
        editors         See who is editing a watched file
        export          Export sources from CVS, similar to checkout
        history         Show repository access history
        import          Import sources into CVS, using vendor branches
        init            Create a CVS repository if it doesn't exist
        kserver         Kerberos server mode
        log             Print out history information for files
        login           Prompt for password for authenticating server
        logout          Removes entry in .cvspass for remote repository
        pserver         Password server mode
        rannotate       Show last revision where each line of module was modified
        rdiff           Create 'patch' format diffs between releases
        release         Indicate that a Module is no longer in use
        remove          Remove an entry from the repository
        rlog            Print out history information for a module
        rtag            Add a symbolic tag to a module
```

我們可以使用 cvs –H 指令來了解 CVS 指令，我們使用指令名稱當作參數。這是 cvs add 指令的參數說明。

```
[root@flash chaiyen]# cvs -H add
Usage: cvs add [-k rcs-kflag] [-m message] files...
        -k      Use "rcs-kflag" to add the file with the specified kflag.
        -m      Use "message" for the creation log.
(Specify the --help global option for a list of other help options)
```

這是 cvs commit 指令的參數說明。

```
[root@flash chaiyen]# cvs -H commit
Usage: cvs commit [-nRlf] [-m msg | -F logfile] [-r rev] files...
    -n          Do not run the module program (if any).
    -R          Process directories recursively.
    -l          Local directory only (not recursive).
    -f          Force the file to be committed; disables recursion.
    -F logfile  Read the log message from file.
    -m msg      Log message.
    -r rev      Commit to this branch or trunk revision.
(Specify the --help global option for a list of other help options)
```

我們可以建立 CVS 原始碼的容器。我們將建立 CVSROOT 變數到開始檔案的路

徑中。在 Bash 中我們要修改~/．bashrc 或~/．profile 的檔案。在 TC shell 中我們要修改~/．chsrc 或~/．tcshrc 的檔案。我們使用 CVSAREA 變數來儲存我們 CVS 工作區的位置。

我們可以修改 Bash 中的~/．bashrc。

#vi　~/.bashrc

我們加入 CVSAREA=/home/projects 和 CVSROOT=/home/cvsroot 再加上輸出變數 export CVSAREA CVSROOT。

```
# .bashrc

# User specific aliases and functions
PATH="$PATH:."
CVSAREA=/home/projects
CVSROOT=/home/cvsroot
export CVSAREA CVSROOT
# Source global definitions
        . /etc/bashrc
fi
```

當我們從新啟動後，來顯示變術$CVSROOT 就可以看到/home/cvsroot 的目錄了。

```
[root@flash chaiyen]# echo $CVSROOT
/home/cvsroot
```

如果我們使用的 shell 是 TC shell，則我們也要設定我們 TC shell 的路徑。我們顯示目前的 shell，然後我們使用 chsh 來更改 shell 成 TC shell。

```
[root@flash chaiyen]# echo $SHELL
/bin/bash
[root@flash chaiyen]# chsh
Changing shell for root.
New shell [/bin/bash]: /bin/tcsh
Shell changed.
```

我們使用 vi 修改~/．bashrc 檔。

```
[root@flash chaiyen]# vi ~/.bashrc
```

我們加入 setenv CVSATEA ~projects 和 setenv CVSROOT ~/cvsroot。

```
# .tcshrc

# User specific aliases and functions
setenv CVSAREA ~/projects
setenv CVSROOT ~/CVSROOT
alias rm 'rm -i'
alias cp 'cp -i'
alias mv 'mv -i'

setenv PATH "/usr/local/sbin:/usr/sbin:/sbin:${PATH}:${HOME}/bin"

set prompt='[%n@%m %c]# '
```

我們使用 cvs init 來建立原始碼的容器。

```
[root@flash chaiyen]# cvs init
```

當 cvs init 指令順利執行的時後，CVSROOT 目錄就會在 cvsroot 目錄容器中被建立。這個目錄包含 CVS 維持資訊與管理檔案的設定。我們可以到我們所建立的 cvsroot 目錄中。

```
[root@flash home]# ls -l cvsroot
total 4
drwxrwxr-x    3 root     root         4096  8月 11 17:40 CVSROOT
[root@flash home]# cd cvsroot
[root@flash cvsroot]# ls CVSROOT/
checkoutlist    config,v        Emptydir    modules,v   taginfo
checkoutlist,v  cvswrappers     history     notify      taginfo,v
commitinfo      cvswrappers,v   loginfo     notify,v    val-tags
commitinfo,v    editinfo        loginfo,v   rcsinfo     verifymsg
config          editinfo,v      modules     rcsinfo,v   verifymsg,v
```

我們使用 CVS 管理我們的軟體計劃，在我們建立我們的目錄容器之後，我們需要輸入我們的計劃資源檔案。我們可以使用 cvs import 指令。cvs import 指令用來輸入我們的資源到目錄容器中。這-m 參數是用來支援 cvs 資訊來顯示檔案的歷史資訊。

語法：

指令：CVS　import [參數]

參數：

-b branch：設定分支的 ID 到 branch。

-d　：使用檔案的修正時間當作是輸入時間。

-m 資訊：記錄我們想要的歷史資訊。

這是我們要輸入的資源。

```
[root@flash home]# cd $CVSAREA/project1
[root@flash project1]# ls
compute.c     compute.o  input.h  lib.a,v   main.h     makefile.bak  power2
compute.c,v   input.c    input.o  libh.a    main.h,v   mathlib.a     power5
compute.h     input.c,v  lib.a    main.c,v  makefile   newlib.a
```

我們使用 cvs import 指令輸入我們 project1 到目錄容器中。

```
[root@flash project1]# cvs import -m "輸入測試檔" project1 DemoCVS start
I project1/makefile.bak
N project1/compute.c,v
N project1/compute.h
I project1/compute.o
N project1/input.c
N project1/input.c,v
N project1/input.h
I project1/input.o
I project1/lib.a
N project1/lib.a,v
I project1/libh.a
N project1/main.c,v
N project1/main.h
N project1/main.h,v
N project1/makefile
N project1/compute.c
I project1/mathlib.a
I project1/newlib.a
N project1/power2
N project1/power5

No conflicts created by this import
```

在輸入資源檔到目錄容器和設定存取得權限後，再來我們就要檢查這資源檔。我們可以使用 cvs checkout 指令來檢查資源模組。這個指令允許我們檢查許多一起的資源檔。假如我們要檢查計劃的模組，我們需要在 CVS 資源目錄中檢查目錄的指定名稱。這個指令檢查編輯的資源模組。

語法：

指令：cvs checkout　[參數] 模組

參數：

-D date：以日期檢查版本。

-P：刪除空的目錄。

-R：以遞迴的方式處理目錄。

-d 目錄：以目錄而不是以模組名稱檢查。

-j 版本：合併目前版本和指定版本。

-r 版本：檢查版本。

我們可以使用 cvs checkout 指令來檢查資源模組。

```
[root@flash projects]# cvs checkout project1/
cvs checkout: Updating project1
U project1/compute.c
U project1/compute.c,v
U project1/compute.h
U project1/input.c
U project1/input.c,v
U project1/input.h
U project1/lib.a,v
U project1/main.c,v
U project1/main.h
U project1/main.h,v
U project1/makefile
U project1/power2
U project1/power5
```

```
[root@flash projects]# cd project1/
[root@flash project1]# ls -l
total 112
-rwxr-xr-x    1 root     root          137  8月 11 21:38 compute.c
-rwxr-xr-x    1 root     root          325  8月 11 21:38 compute.c,v
-rwxr-xr-x    1 root     root           67  8月 11 21:38 compute.h
-rwxr--r--    1 nobody   nobody        852  8月 10 15:57 compute.o
drwxr-xr-x    2 root     root         4096  8月 11 22:19 CVS
-rwxr-xr-x    1 root     root          121  8月 11 21:38 input.c
-rwxr-xr-x    1 root     root          334  8月 11 21:38 input.c,v
-rwxr-xr-x    1 root     root           54  8月 11 21:38 input.h
-rwxr--r--    1 nobody   nobody        964  8月 10 11:38 input.o
-r-xr--r--    1 nobody   nobody        464  8月 10 17:22 lib.a
-rwxr-xr-x    1 root     root          652  8月 11 21:38 lib.a,v
-rwxr--r--    1 nobody   nobody        282  8月  9 23:11 libh.a
-rwxr-xr-x    1 root     root          426  8月 11 21:38 main.c,v
-rwxr-xr-x    1 root     root           86  8月 11 21:38 main.h
-rwxr-xr-x    1 root     root          274  8月 11 21:38 main.h,v
-rwxr-xr-x    1 root     root          375  8月 11 21:38 makefile
-rwxr--r--    1 nobody   nobody        258  8月  9 17:06 makefile.bak
-rwxr--r--    1 nobody   nobody       6870  8月 10 11:38 mathlib.a
-rwxr--r--    1 nobody   nobody       3288  8月 10 10:03 newlib.a
```

```
[root@flash project1]# cd CVS
[root@flash CVS]# ls -l
total 12
-rw-r--r--    1 root     root          584  8月 11 22:19 Entries
-rw-r--r--    1 root     root            9  8月 11 22:19 Repository
-rw-r--r--    1 root     root           14  8月 11 22:19 Root
[root@flash CVS]# more Entries
/compute.c/1.1.1.1/Sun Aug 11 13:38:18 2002//
/compute.c,v/1.1.1.1/Sun Aug 11 13:38:18 2002//
/compute.h/1.1.1.1/Sun Aug 11 13:38:18 2002//
/input.c/1.1.1.1/Sun Aug 11 13:38:18 2002//
/input.c,v/1.1.1.1/Sun Aug 11 13:38:18 2002//
/input.h/1.1.1.1/Sun Aug 11 13:38:18 2002//
/lib.a,v/1.1.1.1/Sun Aug 11 13:38:18 2002//
/main.c,v/1.1.1.1/Sun Aug 11 13:38:18 2002//
/main.h/1.1.1.1/Sun Aug 11 13:38:18 2002//
/main.h,v/1.1.1.1/Sun Aug 11 13:38:18 2002//
/makefile/1.1.1.1/Sun Aug 11 13:38:18 2002//
/power2/1.1.1.1/Sun Aug 11 13:38:18 2002//
/power5/1.1.1.1/Sun Aug 11 13:38:18 2002//
```

我們顯示目錄容器。

```
[root@flash CVS]# more Repository
project1
```

我們顯示 cvsroot 的目錄。

```
[root@flash CVS]# more Root
/home/cvsroot
```

我們可以使用 cvs commit 指令來執行資源檔的改變。

語法：

指令：cvs commit　[參數] 檔案串列

參數：

-F 檔案：從檔案讀取記錄資訊。

-R：以遞迴的方式處理目錄。

-f：強制檔案串列中的檔案被執行，非遞迴的方式。

-m message：記錄訊息。

-r 版本：執行這個版本。

我們修改 input.c 的檔案。

```
[root@flash project1]# vi input.c
#include "input.h"

double input(char *s)
{
  float x;

  printf ("%s",s);
  scanf ("%f", &x);
  return (x);
}
```

我們可以使用 cvs commit 指令來執行資源檔的改變，這時它成為 1.2 版。

```
[root@flash project1]# cvs commit -m"加強使用者界面" input.c
Checking in input.c;
/home/cvsroot/project1/input.c,v  <--  input.c
new revision: 1.2; previous revision: 1.1
done
```

我們可以使用 cvs rtag 指令來加入符號標記到模組串列中的模組。

語法：

指令：cvs rtag　[參數] tag　模組串列

參數：

-D：使用存在的日期當作標記。

-F：假如標記已存在就移動標記。

-R：以遞迴的方式處理目錄。

-b：設定標記一個分支標記，來同步發展。

-d：刪除標記

-r　版本：分配版本當作標記。

我們可以使用 cvs export 指令來輸出在模組上特別標記的資源。

語法：

指令：cvs export　[參數]　模組

參數：

-D date：輸出某日期的版本。

-d　目錄：輸出到目錄，而不是到模組名稱。

-r　版本：輸出某版本標記的資源。

17-6 靜態分析工具

我們有時後想要知到程式花多少時間在執行每一個函數。我們可以使用 Linux 工具 gprof 來顯示我們程示中每一個函數執行時間的百分比。gprof 工具是非常有效率

的，它可以辨別出主要花費我們最多執行時間的函數。我們可以使用這個資訊來改善效率，而且把函數和程式最佳化。

gprof 指令可以顯示從 gmon.out 圖表顯示 C 語言程式執行的情況。

語法：

指令：gprof [參數] [目的檔[gmon.out]]

參數：

-b：不顯示在圖表中每一欄位的描述。

-e 函式名稱：不顯示指定函式名稱的圖表。

-E 函式名稱：不包括指定函數花費的時間。

-f 函式名稱：顯示指定函數和其衍生的函數圖表資訊。

-F 函式名稱：只有顯示指定函數和其衍生函數的圖表資訊。

-z 函式名稱：顯示那些沒有被使用的函式名稱。

我們使用 gcc –p 來產生顯示我們 gprof 指令所需的資訊 gmon.out。

```
[root@flash 1-6]# gcc -p input.c compute.c main.c -lm
[root@flash 1-6]# a.out
請輸入x的y次方，以便求x的y次方的值
請輸入x的值:2
請輸入y次方的值:2
x的y次方: 4.000000
[root@flash 1-6]# ls -l gmon.out
-rw-r--r--    1 root     root          452  8月 12 08:30 gmon.out
```

我們使用 gprof –b 來顯示圖表中每一欄位的描述。圖中顯示 input 函數被呼叫兩次，而 compute 函數被呼叫一次，而所占用的時間非常短暫，幾乎為零。

```
[root@flash 1-6]# gprof -b
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00        2     0.00     0.00  input
  0.00      0.00     0.00        1     0.00     0.00  compute

                        Call graph


granularity: each sample hit covers 4 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00       2/2           main [12]
[1]      0.0    0.00    0.00       2         input [1]
-----------------------------------------------
                0.00    0.00       1/1           main [12]
[2]      0.0    0.00    0.00       1         compute [2]
-----------------------------------------------

Index by function name

   [2] compute                 [1] input
```

gprof 指令輸出欄位第一列說明。

欄位	說明
%time	函數使用時間占所有執行時間的百分比。
Cumulative seconds	函數和上列函數累計所執行的時間。
Self seconds	函數本身所執行的時間。
Calls	函數被呼叫的所有時間。
Self us/call	每一次呼叫，花費在函數的時間 microseconds。
Total us/call	每一次呼叫，花費在函數及其衍生函數的時間 microseconds。
name	函數的名稱

gprof 指令輸出欄位第二列說明。

欄位	說明
Index	編號
%time	花費在函數及其衍生函數的所有時間百分比。
Self	花費在函數的時間。
Children	被衍生函數所花費的時間。一般指遞迴函數。
Called	呼叫函數所花費的時間。
name	目前函數的名稱。

17-7 動態分析工具

除錯的能力包括執行程式、設定中斷點、一步一步執行、列出原始碼、編輯原始碼、修改和了解變數、追蹤程式的執行、尋找函數和變數。

gdb 指令允許 execprog 程式的執行被追蹤，來決定發生了什麼，因此幫助我們辨別在程式中 bug 錯誤的位置。我們可以指定核心檔案，也可以指定正在執行的行程編號。

語法：

指令：gdb [參數] [execprog [core|PID]]

參數：

-c core：使用 core 當作核心檔案來檢查。

-h：使用簡單的解釋列出指令參數。

-n：在處理所有指令參數後，不執行~/.gdbinit 檔案的指令。

-q：不顯示和不簡介版權資訊。

-s 檔案：從檔案使用符號表。

這是我們要除錯的程式。

```
#include <stdio.h>

#define PROMPT "請輸入字串:"
#define size 255

char *get_input(char *);
void null_function1();
void null_function2();

int main()
{
        char *input;

        null_function1();
        null_function2();
        input=get_input(PROMPT);
        (void) printf("你輸入了:%s.\n", input);
        (void) printf("結束\n");
        return(0);
}

void null_function1()
{}

void null_function2()
{}

char *get_input(char *prompt)
{
        char *str;

        (void) printf("%s",prompt);
        for(*str = getchar();*str != '\n';*str = getchar())
        str++;
        *str ='\0';
        return(str);
}
```

當我們使用 c 編譯器來編輯，然後執行，就會發生程式記憶體區段錯誤的發生。

```
[root@flash 1-7]# cc bugged.c -o bugged
[root@flash 1-7]# bugged
請輸入字串:大家歡喜快樂
程式記憶體區段錯誤
```

我們執行程式時，程式提示我們輸入字串，但確沒有執行結果。這是我們沒有將指標初始化，這通常是剛學習 C 的人會發生的情況，這時我們就可以使用 gdb 了。我們可以在編譯時使用-g 的參數來進入 gdb 的環境。-g 這個參數建立可執行的檔案，包含了符號表、除錯、配置記憶體、和數據表的資訊。

我們使用 gcc –g bugged.c –o bugged 來建立 gdb 的除錯環境。gdb 指令允許程式的執行被追蹤，來決定發生了什麼，因此幫助我們辨別在程式中 bug 錯誤的位置。

我們使用 gdb –q bugged 來進入 gdb。

```
[root@flash 1-7]# gcc -g bugged.c -o bugged
[root@flash 1-7]# gdb -q bugged
(gdb)
```

在 gdb 環境中，我們輸入 help 就可以看到所有的 gdb 指令。Help running 指令顯示執行這程式的指令的簡單描述。Help tracepoints 指令顯示追蹤程式的執行而沒有停止程式的簡單描述。Help trace 顯示簡單的追蹤指令的描述。

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
```

這是我們在 gdb 環境中輸入 help running 的指令。

```
(gdb) help running
Running the program.

List of commands:

attach -- Attach to a process or file outside of GDB
continue -- Continue program being debugged
detach -- Detach a process or file previously attached
finish -- Execute until selected stack frame returns
handle -- Specify how to handle a signal
info handle -- What debugger does when program gets various signals
interrupt -- Interrupt the execution of the debugged program
jump -- Continue program being debugged at specified line or address
kill -- Kill execution of program being debugged
next -- Step program
nexti -- Step one instruction
run -- Start debugged program
set args -- Set argument list to give program being debugged when it is started
set environment -- Set environment variable value to give the program
set follow-fork-mode -- Set debugger response to a program call of fork or vfork
set scheduler-locking -- Set mode for locking scheduler during execution
set step-mode -- Set mode of the step operation
show args -- Show argument list to give program being debugged when it is starte


show follow-fork-mode -- Show debugger response to a program call of fork or vfo
rk
show scheduler-locking -- Show mode for locking scheduler during execution
show step-mode -- Show mode of the step operation
signal -- Continue program giving it signal specified by the argument
step -- Step program until it reaches a different source line
stepi -- Step one instruction exactly
target -- Connect to a target machine or process
thread -- Use this command to switch between threads
thread apply -- Apply a command to a list of threads
apply all -- Apply a command to all threads
tty -- Set terminal for future runs of program being debugged
unset environment -- Cancel environment variable VAR for the program
until -- Execute until the program reaches a source line greater than the curren
t

Type "help" followed by command name for full documentation.
```

這是我們在 gdb 環境中輸入 help tracepoints 的指令。

```
(gdb) help tracepoints
Tracing of program execution without stopping the program.

List of commands:

actions -- Specify the actions to be taken at a tracepoint
collect -- Specify one or more data items to be collected at a tracepoint
delete tracepoints -- Delete specified tracepoints
disable tracepoints -- Disable specified tracepoints
enable tracepoints -- Enable specified tracepoints
end -- Ends a list of commands or actions
passcount -- Set the passcount for a tracepoint
save-tracepoints -- Save current tracepoint definitions as a script
tdump -- Print everything collected at the current tracepoint
tfind -- Select a trace frame;
tfind end -- Synonym for 'none'
tfind line -- Select a trace frame by source line
tfind none -- De-select any trace frame and resume 'live' debugging
tfind outside -- Select a trace frame whose PC is outside the given range
tfind pc -- Select a trace frame by PC
tfind range -- Select a trace frame whose PC is in the given range
tfind start -- Select the first trace frame in the trace buffer
tfind tracepoint -- Select a trace frame by tracepoint number
trace -- Set a tracepoint at a specified line or function or address
```

這是我們在 gdb 環境中輸入 help trace 的指令。

```
[root@flash 1-7]# gdb -q bugged
(gdb) help trace
Set a tracepoint at a specified line or function or address.
Argument may be a line number, function name, or '*' plus an address.
For a line number or function, trace at the start of its code.
If an address is specified, trace at that exact address.

Do "help tracepoints" for info on other tracepoint commands.
```

我們使用指令 run 可以執行程式。Segmentation fault 為分斷錯誤。

```
(gdb) run
Starting program: /home/chaiyen/1-7/bugged
請輸入字串:大家歡喜

Program received signal SIGSEGV, Segmentation fault.
0x4000b230 in _dl_runtime_resolve () from /lib/ld-linux.so.2
(gdb)
```

我們可以使用 backtrace 指令，就可以看到程式如何執行到這個位址。

```
(gdb) backtrace
#0  0x4000b230 in _dl_runtime_resolve () from /lib/ld-linux.so.2
#1  0xbffff4f8 in ?? ()
#2  0x4205aldc in printf () from /lib/i686/libc.so.6
#3  0x08048475 in main () at bugged.c:17
#4  0x42017499 in __libc_start_main () from /lib/i686/libc.so.6
(gdb)
```

我們在 gdb 環境下執行 list 來顯示我們的程式原始碼。

```
(gdb) list
```

```
6  char *get_input(char *);
7  void null_function1();
8  void null_function2();
9
10 int main()
11 {
12         char *input;
13
14         null_function1();
15         null_function2();
16         input=get_input(prompt);
17         (void) printf("你輸入了:%s.\n", input);
18         (void) printf("結束\n");
19         return(0);
20 }
21
22 void null_function1()
23 {}
24 void null_function2()
25 {}
26
27 char *get_input(char *prompt)
28 {
29         char *str;
30
31         (void) printf("%s",prompt);
32         for(*str=getchar();*str!='\n';*str=getchar())
33         str++;
34         *str ='\0';
35         return(str);
36 }
```

我們使用 break 32，在第 32 行設定中斷點，並且執行程式。

```
(gdb) break 32
Breakpoint 1 at 0x80484a6: file bugged.c, line 32.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/chaiyen/1-7/bugged

Breakpoint 1, get_input (prompt=0x8048568 "請輸入字串:") at bugged.c:32
32              (void) printf("%s",prompt);
```

我們使用 print *str 列印出變數*str 的值。

```
(gdb) print *str
$1 = 90 'Z'
```

我們使用 step 指令一步一步執行每一行，然後看看變數的變化。

```
(gdb) step
33              for(*str = getchar();*str != '\n';*str = getchar())
(gdb) step
請輸入字串:step
34              str++;
(gdb) step
33              for(*str = getchar();*str != '\n';*str = getchar())
(gdb) step
34              str++;
(gdb) step
33              for(*str = getchar();*str != '\n';*str = getchar())
(gdb) step
34              str++;
(gdb) step
33              for(*str = getchar();*str != '\n';*str = getchar())
(gdb)
34              str++;
(gdb)
33              for(*str = getchar();*str != '\n';*str = getchar())
(gdb)
35              *str ='\0';
```

我們在先前的程式發現，因為我們在事先沒有配置記憶體給變數*str，因此造成程式的錯誤。我們在第 26 行使用 str = (char *) malloc (SIZE *(sizeof (char)));來動態分配記憶體給 str 變數。

```
1  #include <stdio.h>
2
3  #define PROMPT "請輸入字串:"
4  #define SIZE 255
5
6  char *get_input(char *);
7  void null_function1();
8  void null_function2();
9
10 int main()
11 {
12         char *input;
13
14         null_function1();
15         null_function2();
16         input=get_input(PROMPT);
17         (void) printf("你輸入了:%s.\n", input);
18         (void) printf("結束\n");
19         return(0);
20 }
21
22 char *get_input(char *prompt)
23 {
24         char *str,*temp;
25
26         str=(char *) malloc (SIZE * ( sizeof (char)));
27         temp=str;
28         (void) printf("%s",prompt);
29         for(*str = getchar();*str != '\n';*str = getchar())
30            str++;
31         *str ='\0';
32         return(temp);
33 }
```

我們使用 quit 來離開 gdb 環境。

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

我們從新編譯就可以執行，且不會發生錯誤。

```
[root@flash 1-7]# gcc -g rebugged.c -o rebugged
[root@flash 1-7]# rebugged
請輸入字串:5
你輸入了:5.
結束
```

我們在除錯後, 將留下很多 gdb 的資訊到 bugged 程式上, 因此我們可以使用 strip 指令將這些資訊給除去。在使用 strip 指令除去後，我們的檔案就變小了。

```
[root@flash 1-7]# ls -l bugged
-rwxr-xr-x    1 root     root        21816  8月 12 10:11 bugged
[root@flash 1-7]# strip bugged
[root@flash 1-7]# ls -l bugged
-rwxr-xr-x    1 root     root         3336  8月 12 15:19 bugged
```

17-8 執行的性能

我們可以使用 time 指令測量任何 shell 指令或程式的執行性能。這個指令以時：分：秒的格式報告系統時間，使用者的時間，和消逝的時間。消逝的時間是程式完成執行的時間。程式執行時系統所占的時間稱為系統時間。使用者時間是執行程式的時間。

我們使用 time 指令 來報告指令的執行時間。

語法：

指令：time 指令

我們在 TC shell 中可以直接使用 time 指令。我們也可以在 Bash 中使用 /usr/bin/time 指令。我們使用/usr/bin/time find 來看看指令 find 找尋 socket.h 檔案所花費的時間。

```
[root@flash 1-7]# /usr/bin/time find /usr  -name socket.h -print
/usr/lib/bcc/include/sys/socket.h
/usr/lib/dietlibc/include/sys/socket.h
/usr/include/isc/socket.h
/usr/include/asm/socket.h
/usr/include/linux/socket.h
/usr/include/bits/socket.h
/usr/include/sys/socket.h
/usr/include/libguile/socket.h
/usr/include/ptlib/unix/ptlib/socket.h
/usr/include/ptlib/socket.h
/usr/src/linux-2.4.18-3/include/abi/util/socket.h
/usr/src/linux-2.4.18-3/include/asm-i386/socket.h
/usr/src/linux-2.4.18-3/include/linux/socket.h
/usr/i386-glibc21-linux/include/asm/socket.h
/usr/i386-glibc21-linux/include/bits/socket.h
/usr/i386-glibc21-linux/include/linux/socket.h
/usr/i386-glibc21-linux/include/sys/socket.h
0.68user 1.54system 1:40.69elapsed 2%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (158major+70minor)pagefaults 0swaps
```