Linux 除錯利器 - GDB 簡介

Tetralet | 22 九月, 2007 13:30

雖然我們不得不承認，Free Software 最大的問題是：比起同類的商業軟體來，Free Software 往往在功能上有所不足。但個人卻認為：Free Software 的最大優勢是：比起同類的商業軟體來，Free Software 往往穩定多了。原因無它：因為 Free Software 提供了 Source Code，而世界上可不乏願意替這些 Free Software 進行除錯的廣大使用者。

GDB 是 Linux 上最常見的除錯器。我們將以一個簡易的小程式來介紹如何使用 GDB 來替程式除錯。

範例程式：

這是一個非常簡單的小程式：我們使用亂數來取得 5 個四位數字的密碼，並把它們存在陣列裡：

```
1  #include<stdlib.h>
2  #include<stdio.h>
3
4  #define TOTAL 5
5
6  Password(int MAX) {
7          int password[TOTAL];
8          int i;
9
10         srand(time(NULL));
```

```
11          for (i=1;i<=TOTAL;i++) {
12                  password[i]=(rand()%MAX);
13  #ifdef DEBUG
14                  printf("No: %d, Password= %04d\n",i,password[i]);
15  #endif
16          }
17  }
18
19  main () {
20          Password(10000);
21  }
```

其中的 13 ~ 15 行是利用 `DEBUG` 來判斷是否要用 `printf` 來在螢幕上顯示**除錯資訊**，這是在程式寫作時的一個很不錯的小技巧。不過您必須在編譯時加上 `-DDEBUG` 才能看到這些除錯訊息。

接下來，讓我們將它編譯成應用程式，並指定**啟用除錯訊息**：

```
gcc -o Password -DDEBUG Password.c
```

以下是執行結果之一：

```
No: 1, Password= 7815
No: 2, Password= 6157
No: 3, Password= 7479
No: 4, Password= 9017
Segmentation fault
```

有一定的機率這個程式會當掉。問題是出在哪裡呢？該是 GDB 出場的時候了！

事前準備：

為了便於除錯，您必須在編譯軟體時加上 `-g` 參數才能讓 gcc 在編譯程式時，將除錯資訊加到程式裡。例：

```
gcc -g -o Password -DDEBUG Password.c
```

因為這些除錯訊息會增加應用程式的檔案大小，有時大小差異會高達 10 倍之多，所以一般在發佈應用程式時是不會以 `-g` 參數編譯的。您可以在事後利用 `strip` 指令清掉應用程式裡的除錯資訊。例：

```
strip Password
```

GDB 的 區塊(frame) 和堆疊(stack) 觀念

在 `gdb` 裡，為了方便除錯，它將程式碼以**副程式**為單位分成一個個的區塊 (frame)。比如說，在上例裡的 `main()` 會被視為一個 frame，而 `Password()` 則會被視為另一個 frame。因而在利用 `gdb` 除錯時，可以直接執行某個 frame、跳至上一個或下一個 frame、直接執行至該 frame 結束... 等等，而加快在使用 `gdb` 來 debug 的速度並降低 debug 的困難度。

在 frame 與 frame 之間，正在執行的區塊就是 `frame 0`。呼叫該區塊的就是 `frame 1`；而再上上一層的就叫 `frame 2`，以此類推。

在 gdb 進入另一個 frame 之前，它會將該 frame 的一些變數值之類的儲存至堆疊 (stack) 裡，等到從 frame 回來後再從這些 stack 裡把這些變數值取回來。

我們會在下文中看到非常多應用 `frame` 和 `stack` 的指令。

執行 GDB：

在確認程式在編譯時有加入除錯資訊後，就可以利用 GDB 來進行除錯了。例：

```
gdb ./Password
```

以下是 gdb 的常見指令（其中 () 內為簡短指令）：

`help (h)`：顯示指令簡短說明。例：`help` *`breakpoint`*

`file`：開啟檔案。等同於 `gdb` *`filename`*

`run (r)`：執行程式，或是從頭再執行程式。

kill：中止程式的執行。

backtrace (bt)：堆疊追蹤。會顯示出上層所有的 frame 的簡略資訊。

print (p)：印出變數內容。例：print i，印出變數 i 的內容。

list (l)：印出程式碼。若在編譯時沒有加上 -g 參數，list 指令將無作用。

whatis：印出變數的型態。例： whatis i，印出變數 i 的型態。

breakpoint (b, bre, break)：設定中斷點
使用 info breakpoint (info b) 來查看已設定了哪些中斷點。
在程式被中斷後，使用 info line 來查看正停在哪一行。

continue (c, cont)：繼續執行。和 breakpoint 搭配使用。

frame：顯示正在執行的行數、副程式名稱、及其所傳送的參數等等 frame 資訊。
frame 2：看到 #2，也就是上上一層的 frame 的資訊。

next (n)：單步執行，但遇到 frame 時不會進入 frame 中單步執行。

step (s)：單步執行。但遇到 frame 時則會進入 frame 中單步執行。

until：直接跑完一個 while 迴圈。

return：中止執行該 frame（視同該 frame 已執行完畢），
並返回上個 frame 的呼叫點。功用類似 C 裡的 return 指令。

finish：執行完這個 frame。當進入一個過深的 frame 時，如：C 函式庫，
可能必須下達多個 finish 才能回到原來的進入點。

up：直接回到上一層的 frame，並顯示其 stack 資訊，如進入點及傳入的參數等。
up 2：直接回到上三層的 frame，並顯示其 stack 資訊。

down：直接跳到下一層的 frame，並顯示其 stack 資訊。
必須使用 up 回到上層的 frame 後，才能用 down 回到該層來。

display：在遇到中斷點時，自動顯示某變數的內容。

undisplay：取消 display，取消自動顯示某變數功能。

commands：在遇到中斷點時要自動執行的指令。

info：顯示一些特定的資訊。如： info break，顯示中斷點，

`info share`，顯示共享函式庫資訊。

`disable`：暫時關閉某個 `breakpoint` 或 `display` 之功能。

`enable`：將被 `disable` 暫時關閉的功能再啟用。

`clear`/`delete`：刪除某個 `breakpoint`。

`set`：設定特定參數。如：`set env`，設定環境變數。也可以拿來修改變數的值。

`unset`：取消特定參數。如：`unset env`，刪除環境變數。

`show`：顯示特定參數。如：`show environment`，顯示環境變數。

`attach` *`PID`*：載入已執行中的程式以進行除錯。其中的 `PID` 可由 `ps` 指令取得。

`detach` *`PID`*：釋放已 `attach` 的程式。

`shell`：執行 Shell 指令。如：`shell ls`，呼叫 sh 以執行 ls 指令。

`quit`：離開 `gdb`。或是按下 `<Ctrl><C>` 也行。

`<Enter>`：直接執行上個指令

執行程式

在 GDB 啟動上面的範例程式 Password 後，並不會立即執行該程式。gdb 讓您能在此先指定某些中斷點或參數，在準備完畢後，鍵入 `run`，程式才會開始執行；使用 `kill` 來中斷正在執行中的程式。

在本例中，在鍵入 `run` 執行後，gdb 很快得就抓到了錯誤點了，真是一點都不含糊：（當然，大多數的程式的錯誤，尤其像是本例中的記憶體存取錯誤，不會那麼容易就抓到的）

```
(gdb) run
Starting program: /tmp/b/Password
No: 1, Password= 7815
No: 2, Password= 6157
No: 3, Password= 7479
No: 4, Password= 9017

Program received signal SIGSEGV, Segmentation fault.
```

```
0x08048443 in Password (MAX=10000) at Password.c:14
14                      printf("No: %d, Password= %04dn",i,password[i]);

(gdb) kill
Kill the program being debugged? (y or n) y
```

在上例中的除錯資訊包含了出錯的行號、所傳遞的參數，及原始程式碼。但如果您沒有在程式裡加入除錯資訊，那些除錯資訊就不會出現，這將會讓除錯變得更加困難。

您可以在 `run` 後面加上參數，它們會被視為命令列的參數傳遞給程式執行。

您也可以利用 `set args` 來設定命令列的參數，使用 `show args` 來顯示被設定的命令列參數。

堆疊追蹤

如果這個程式有使用副程式，您還可以利用 `backtrace (bt)` 指令來找到程式的進入點，了解到程式是如何執行到這個步驟的，以及在副程式間所傳遞的參數內容。一旦 gdb 捕捉到 `SIGSEGV` 資訊時，也可以使用 `bt` 來試圖找到程式的出錯點。例：

```
(gdb) bt
#0  0x08048443 in Password (MAX=10000) at Password.c:14
#1  0x0804848b in main () at Password.c:24
```

在上例中，我們可以知道：出錯的地方在 `Password (MAX=10000)`，它是由 `Password.c` 第 20 行的 `main ()` 所呼叫的。而 `MAX=10000` 則是傳遞給 `Password ()` 的參數內容。

而前頭的 #0 和 #1 就是上文中所討論的 **frame 的層數**。可以由此看出，`Password (MAX=10000) (frame 0)` 是由 `main () (frame 1)` 在 `Password.c` 的第 24 行呼叫的。

顯示變數內容：

在上例中，我們知道問題是出在 `Password.c` 的第 14 行，但也許我們還搞不懂問題是如何發生的。我們可以利用 `print (p)` 指令來印出變數的內容，利用 `print/x` 來以 16 進位列印變數內容。例：

```
(gdb) p i
$1 = 6078
```

結果是 `i = 6078`？這和我們預期的 `i=5` 有極大差距。

在上例中，`$1` 是執行 `print` 指令後，gdb 產生的一個**臨時變數**，方便我們在接下來的指令中採用。我們可以直接利用 `$1` 來取用這個變數，使用 `$` 來取用最後產生的臨時變數，使用 `$$` 來取用上一個產生的臨時變數。例：

```
(gdb) p password[$]
Cannot access memory at address 0xbff750f4
```

原本預期是 `i=5` 的，但如今 `i=6078`，也難怪存取 `password[i]` 會讓程式當掉了。

我們也可以利用相同的方式來顯示陣列（或其它變數）的內容。例：

```
(gdb) p password[5]
$2 = 6078
```

其中的 `$2` 是 gdb 替 `password[5]` 產生的另一個臨時變數。

在上例中，我們要求 gdb 顯示：若 `i=5` 時，`password[i]` 的內容其值為何？結果是 `6078`，而它卻剛好是 `i` 的數值。

我們也可以利用 `list` 一次印出陣列裡的值。例：

```
(gdb) p password[1]@5
$1 = {7815, 6157, 7479, 9017, 6078}
```

在上例中，我們印出了從 `password[1]` 之後 `5` 個陣列裡的值，也就是 `password[1]` ~ `password[5]` 的值。

或者，利用 `info locals` 一次列出所有區域變數的值：

```
(gdb) info locals
```

```
password = {-1209199346, 7815, 6157, 7479, 9017}
i = 6078
```

現在我們應該可以看出問題是出在哪裡了。

直接查看原始碼：

gdb 告訴我們，問題是出在 `Password.c` 的第 14 行。我們可以利用 `list (l)` 來直接印出當前位置的上下 5 行原始碼：

```
(gdb) l
9
10              srand(time(NULL));
11              for (i=1;i<=TOTAL;i++) {
12                      password[i]=(rand()%MAX);
13      #ifdef DEBUG
14                      printf("No: %d, Password= %04dn",i,password[i]);
15      #endif
16              }
17      }
18
```

再下達一次 `list` 會繼續印出之後的原始碼。我們也可以利用 `list 7` 要求印出原始程式第 7 行的上下 5 行；利用 `list 1,5` 要求印出原始程式的第 1 ~ 5 行。

設定中斷點：

如果這樣還是看不出問題出在哪裡，我們可以利用 `breakpoint (b, break)` 設定中斷點 (breakpoint)，讓程式執行到中斷點時能夠停下來讓我們進行其它的除錯工作。如果我們是在 debug 一個大型的專案，我們也可以針對某個檔案指定其中斷點，像是 `break Password.c:12`。或者是指定一個副程式 (frame)，像是 `break Password`。`break +3` 則是將中斷點設定為現行位置以下第 3 行。`break 12 if (i==5)` 則是說，若 i 等於 5 時在第 12 行設定中斷點。

在查看沒問題後，利用 continue (c, cont) 繼續執行。例：

```
# 設定中斷點為目前檔案的第 12 行。效果等同於 b Password.c:12。
(gdb) b 12
Breakpoint 1 at 0x8048428: file Password.c, line 12.

# 開始執行
(gdb) r
Starting program: /tmp/b/Password

# 被中斷了
Breakpoint 1, Password (MAX=10000) at Password.c:12
12                      password[i]=(rand()%MAX);

# 顯示 i 的內容
(gdb) p i
$1 = 1

# 顯示 password[1] 的內容
(gdb) p password[$]
$2 = -1208274956

# 繼續執行
(gdb) c
Continuing.
No: 1, Password= 7815

# 又被中斷了
Breakpoint 1, Password (MAX=10000) at Password.c:12
12                      password[i]=(rand()%MAX);
```

```
# 再次顯示 password[1] 的內容
(gdb) p password[1]
$3 = 7815
```

但如果每次遇到中斷點都得下指令顯示 `password[1]` 的內容，其實也不夠人性化。您可以利用 `display` 來指定每次遇到中斷點時要顯示的變數內容。例：

```
# 設定中斷點為第 12 行
(gdb) b 12
Breakpoint 1 at 0x8048428: file Password.c, line 12.

# 開始執行
(gdb) r
Starting program: /tmp/b/Password

# 被中斷了
Breakpoint 1, Password (MAX=10000) at Password.c:12
12                      password[i]=(rand()%MAX);

# 要求 gdb 持續顯示 password[1] 的內容
(gdb) display password[1]
1: password[1] = -1208127500

# 繼續執行
(gdb) c
Continuing.
No: 1, Password= 2318

# 又被中斷了。但這次 gdb 自動顯示 password[1] 的內容
Breakpoint 1, Password (MAX=10000) at Password.c:12
12                      password[i]=(rand()%MAX);
```

```
1: password[1] = 2318
```

我們也可以利用 Commands 指定在遇到中斷點時所要執行的指令。例：

```
# 設定中斷點為第 14 行
(gdb) b 14
Breakpoint 1 at 0x8048440: file Password.c, line 14.

# 指定遇到中斷點 #1 時要執行的指令。
# 若未指定中斷點，則為最後一個中斷點。
# 鍵入 end 來完成輸入。
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>p i                 # 顯示 i 的內容
>p password [i]      # 顯示 password [i]
>c                   # 然後續繼執行
>end                 # 輸入完成

# 開始執行
(gdb) r
Starting program: /tmp/b/Password

# 被中斷了，自動執行上文中所指定的指令。
# 但 gdb 沒有將這些指令顯示在螢幕上，只顯示執行的結果。
Breakpoint 1, Password (MAX=10000) at Password.c:14
14                      printf("No: %d, Password= %04dn",i,password[i]);
$1 = 1
$2 = 8279
No: 1, Password= 8279
```

```
# 又被中斷了
Breakpoint 1, Password (MAX=10000) at Password.c:14
14                      printf("No: %d, Password= %04dn",i,password[i]);
$3 = 2
$4 = 3080
No: 2, Password= 3080
```

我們還可以利用 info 指令來檢視我們設定了多少 breakpoint (及 commands) 和 display，並利用 disable 來暫時關閉它。例：

```
# 顯示 display。只有一項：password[1]
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
1:   y  password[1]

# 顯示 breakpoint 及 commands。其中的 Enb 是指 Enabled。
(gdb) info breakpoint
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048440 in Password at Password.c:14
        breakpoint already hit 1 time
        p i
        p password [i]
        c

# 利用 disable 來暫時關閉它。
(gdb) disable display 1
(gdb) disable breakpoint 1
```

disable 不加參數將會關閉所有的 breakpoint (及 commands) 和 display。使用 enable 來再次啟用。使用 clear 或 delete 來刪除中斷點。

單步執行：

也許我們得一步一步得執行程式才能看清楚程式到底是哪裡出錯。您可以在程式被中斷之後，利用 next (n) 和 step (s) 來一行一行執行程式。例：

```
# 列出第 24 行前後的原始程式碼
(gdb) l 24
19      main () {
20              Password(10000);
21      }

# 在第 20 行上設定中斷點
(gdb) b 20
Breakpoint 1 at 0x804847f: file Password.c, line 20.

# 執行
(gdb) r
Starting program: /tmp/b/Password

# 遇到中斷點了
Breakpoint 1, main () at Password.c:20
20              Password(10000);

# 單步執行。step 會進入副程式中一步一步執行。
(gdb) step
Password (MAX=10000) at Password.c:10
10              srand(time(NULL));

# 按 <Enter> 來直接執行上個指令，也就是繼續單步執行
(gdb)
11              for (i=1;i<=TOTAL;i++) {
```

```
# 繼續單步執行
(gdb)
12                      password[i]=(rand()%MAX);

# 嗯... 厭煩了。讓程式繼續執行吧！
(gdb) finish
Run till exit from #0  Password (MAX=10000) at Password.c:14
No: 1, Password= 6100
No: 2, Password= 2856
No: 3, Password= 3963
No: 4, Password= 0749

Program received signal SIGSEGV, Segmentation fault.
0x08048443 in Password (MAX=10000) at Password.c:14
14                      printf("No: %d, Password= %04dn",i,password[i]);

# 中止除錯
(gdb) kill
Kill the program being debugged? (y or n) y

# 重新執行
(gdb) r
Starting program: /tmp/b/Password

# 遇到中斷點了
Breakpoint 1, main () at Password.c:20
20              Password(10000);

# 單步執行。但 next 會直接跑完整個副程式，不會進入副程式中一步一步執行。
```

```
(gdb) next
No: 1, Password= 5672
No: 2, Password= 1951
No: 3, Password= 1921
No: 4, Password= 9885

Program received signal SIGSEGV, Segmentation fault.
0x08048443 in Password (MAX=10000) at Password.c:14
14                      printf("No: %d, Password= %04dn",i,password[i]);
```

其中，`step` 遇到副程式 (frame) 時會進入副程式中一步一步執行；而 `next` 則會直接跑完整個副程式，不會進入副程式中一步一步執行。

我們還可以利用 `up` 回到上一層，該副程式的呼叫點；然後利用 `down` 回到呼叫 `up` 的位置。

顯示資訊：

我們還可以利用以下指令顯示某些資訊以利於我們進行 debug。例：

`frame`：顯示正在執行的行數、副程式、及其所傳送的參數。

`info frame`：顯示更多的副程式資訊。

`info args`：顯示傳給副程式的參數值。上文已有介紹。

`info locals`：顯示該副程式內所有區域變數的值。

`info reg`：顯示暫存器的值。

`info all-reg`：顯示暫存器的值，包括數學運算暫存器。

以上所說明的是一般在利用 GDB 來進行除錯時常會使用的功能。和一些整合型的開發介面比較起來，其實 GDB 也提供了不遑多讓的強大功能，只要稍加熟悉這些工具，相信即使沒了那些整合型的開發介面，在文字介面之下要進行程式開發/除錯也非難事。

對於程式開發者而言，開發新的程式應該是非常愉悅且充滿成就感的工作，但一講到後續維護，像是功能補強、處理使用者需求或是

程式除錯，就變成極繁雜且容易令人生厭的麻煩事了。有多少有如慧星般光明燦爛的專案就是因為缺乏良好的後續維護而像流星般殞落了呢？

或者我們更應該說，一個軟體能否成功不光取決於程式設計師的功力，後續的服務和維護才是決勝點。一個沒有維護者的孤兒軟體往往很容易得就消失在歷史的洪流之中了。

Debian | 下一篇 | 上一篇 | 迴響 (20) | 引用 (3)

迴響

感謝你~

我是bestself-tw，寫程式一直是我的夢想，可是對我這個英文很爛的人來說，一直是個致命傷。 今天無聊又在網路上亂逛(放在老家的電腦，下禮拜能搬回來了，現在在網咖玩。喝了點啤酒:)，舒解一下生活上的壓力)。 改天有空我再來認真看這篇文章，喝點酒暫不宜看此文章，等頭腦清楚一點再來理解。 謝謝你對自由軟體的付出。 最後我想說~好想再去參加網聚喔~

愛與溫柔 @ 23/09/2007, 03:04 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

這篇真是太詳盡了...
作者真用心

c9s @ 23/09/2007, 18:32 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

Good Job!

---

lancetw @ 24/09/2007, 17:40 [回應]

很详细

带着例子的GDB介绍，很好，呵呵，很喜欢。

---

cocobear @ 26/09/2007, 22:41 [回應]

非常精簡

很好, 非常精簡. 不用花太多時間看全文十. 手上現有資料;

Red Hat Enterprise Linux 4
Debugging with gdb
http://linux.web.cern.ch/linux/scientific4/docs/rhel-gdb-en-4/index.html

是 2003-07-22-cvs 編, 彼太舊.

剛找到的是;

http://sourceware.org/gdb/current/onlinedocs/gdb_toc.html

是 2007 9 月17日 出品, 但還未詳細看, 未不知否適合 ?

無名氏 @ 30/09/2007, 16:27 [回應]

愚見

以我愚見, 沒有必要和對 Free Software 存有偏見的人氏理論. 他走他的陽關道,我過我的獨木橋.

如從自私的角度出發, 少些人對 Free Software 認識和能應用, 對 Free Software 熟習的人氏只有利而無害, 可以善價而沽. 若 Free Software 人材遍地泛濫 , 找工作可不易.

無名氏 @ 30/09/2007, 16:30 [回應]

請繼續加油！

非常受用的內容，讓我一窺 GDB的全貌，很快進入狀況。相當感謝！

---

qerter @ 06/10/2007, 00:47 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

首先謝謝你的文章, 但我有個問題就是
我執行Password, 並不會有當掉的情形..
我也嘗試把 TOTAL 改成 100 多跑幾次..但還是OK!
請問是哪裡出了錯?

殺手 @ 22/10/2007, 10:36 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

還有就是
printf("No: %d, Password= %04dn",i,password[i]);
好像漏掉了, 應該更改為
printf("No: %d, Password= %04dn",i,password[i]);
提供你做參考

殺手 @ 22/10/2007, 10:40 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

抱歉!上一篇回覆有點小錯..應該是

還有就是
printf("No: %d, Password= %04dn",i,password[i]);
好像漏掉了, 應該更改為
printf("No: %d, Password= %04d/n",i,password[i]);
提供你做參考

殺手 @ 22/10/2007, 11:09 [回應]

Re: 殺手

不會當？ XD

請把「Password(10000);」裡的數字改大一點試試看。

另外，"n" 誤植的原因是本 blog 在貼文時會把 "倒斜線" 給吃掉，所以囉...

感謝回報問題！

Tetralet @ 22/10/2007, 19:10 [回應]

target remote

感謝感謝，讓人一目了然

是不是用在target remote
有些指令就不適用了
ex: run,step,....

thanks

Cindy @ 14/11/2007, 17:38 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

在windows上用MinGW编译出来，不会当掉，用GDB查看变量地址并没有像预期的那样重叠，难道优化太差？感谢版主好文

yc @ 07/01/2008, 16:18 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

不錯不錯，介紹得很詳盡，有沒有考慮做一份手冊來給我們參考？

carl_tw @ 07/05/2008, 12:56 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

首先謝謝你的文章, 但我有個問題就是
我執行Password, 並不會有當掉的情形..
我也嘗試把 TOTAL 改成 100 多跑幾次..但還是OK!
請問是哪裡出了錯?

---

dfsa @ 05/06/2008, 18:31 [回應]

边界溢出错误

因为定义了int password[TOTAL]
然后for (i=0;i

---

pinong @ 10/09/2008, 14:08 [回應]

边界溢出错误

因为定义了int password[TOTAL]
然后for (i=0;i<=TOTAL;i++)
password[i]=(rand()%MAX);
那么password会被赋TOTAL+1次值，所以属边界溢出错误，执行的大部分时候都会报错

to Telralet: 评论忘了屏蔽<了 :)

---

pinong @ 10/09/2008, 14:11 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

我是gdb的爱好 者

元谷 @ 24/12/2011, 21:01 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

感謝分享 受益良多

小帥 @ 03/04/2012, 11:09 [回應]

Re: *Linux 除錯利器 - GDB 簡介*

帶有範例的 GDB教學真的很有用，太感謝了

60341006S @ 13/01/2015, 12:07 [回應]

發表迴響

標題

內容 (必填)

暱稱 (必填)

電子郵件

個人網頁

驗證碼前 4 碼

驗證碼皆為英文大寫字母。
僅輸入前 4 碼即可。後 2 碼是假的，欺敵用。
這是為了防制 Spam 而設計的。若造成您的不便還請見諒！

發表