# 用GDB調試程序

出自Ubuntu中文

- 用GDB調試程序(zz)
- 作者：haoel (QQ是：753640，MSN是：haoel@hotmail.com)
- 來源：

目錄

http://blog.csdn.net/haoel/archive/2003/07/02/2879.aspx

# GDB概述

GDB 是GNU開源組織發布的一個強大的UNIX下的程序調試工具。或許，各位比較喜歡那種圖形界面方式的，像VC、BCB等IDE的調試，但如果你是在 UNIX平台下做軟件，你會發現GDB這個調試工具有比VC、BCB的圖形化調試器更強大的功能。所謂“寸有所長，尺有所短”就是這個道理。

一般來說，GDB主要幫忙你完成下面四個方面的功能：

1. 啟動你的程序，可以按照你的自定義的要求隨心所欲的運行程序。

2. 可讓被調試的程序在你所指定的調置的斷點處停住。（斷點可以是條件表達式）
3. 當程序被停住時，可以檢查此時你的程序中所發生的事。
4. 動態的改變你程序的執行環境。

從上面看來，GDB和一般的調試工具沒有什麼兩樣，基本上也是完成這些功能，不過在細節上，你會發現GDB這個調試工具的強大，大家可能比較習慣了圖形化的調試工具，但有時候，命令行的調試工具卻有着圖形化工具所不能完成的功能。讓我們一一看來。

## 一個調試示例

源程序：tst.c

```
#include <stdio.h>

int func(int n)
{
    int sum=0,i;
    for(i=1; i<=n; i++)
    {
        sum+=i;
    }
    return sum;
}


int main()
{
    int i;
    long result = 0;
    for(i=1; i<=100; i++)
    {
        result += i;
    }

    printf("result[1-100] =  %ld\n", result );
    printf("result[1-250] =  %d\n", func(250) );
    return 0;
}
```

編譯生成執行文件：

```
$gcc -g -Wall tst.c -o tst
```

使用GDB調試：

```
$ gdb tst   <---------- 启动GDB
GNU gdb 6.7.1-debian
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb) l 1 <-------------------- l命令相当于list，从第一行开始列出源码。
1       #include <stdio.h>
2
3       int func(int n)
4       {
5               int sum=0,i;
6               for(i=1; i<=n; i++)
7               {
8                       sum+=i;
9               }
10              return sum;
(gdb)    <-------------------- 直接回车表示，重复上一次命令
11      }
12
13
14      int main()
15      {
16              int i;
17              long result = 0;
18              for(i=1; i<=100; i++)
19              {
20                      result += i;
(gdb) break 16    <-------------------- 设置断点，在源程序第16行处。
Breakpoint 1 at 0x80483b2: file tst.c, line 16.
(gdb) break func   <-------------------- 设置断点，在函数func()入口处。
Breakpoint 2 at 0x804837a: file tst.c, line 5.
(gdb) info break   <-------------------- 查看断点信息。
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483b2 in main at tst.c:16
2   breakpoint     keep y   0x0804837a in func at tst.c:5
(gdb) r    <-------------------- 运行程序，run命令简写
Starting program: /home/dbzhang/tst

Breakpoint 1, main () at tst.c:17
17                      long result = 0;
(gdb) n     <---------------------- 单条语句执行，next命令简写。
18                      for(i=1; i<=100; i++)
(gdb) n
20                              result += i;
(gdb) n
18                      for(i=1; i<=100; i++)
(gdb) n
20                              result += i;
(gdb) c    <---------------------- 继续运行程序，continue命令简写。
Continuing.
result[1-100] = 5050    <----------程序输出。

Breakpoint 2, func (n=250) at tst.c:5
5                       int sum=0,i;
(gdb) n
6                       for(i=1; i<=n; i++)
(gdb) p i   <---------------------- 打印变量i的值，print命令简写。
$1 = -1074568236
(gdb) n
8                               sum+=i;
(gdb) n
6                       for(i=1; i<=n; i++)
(gdb) p sum
$2 = 1
(gdb) n
8                               sum+=i;
(gdb) p i
$3 = 2
(gdb) n
6                       for(i=1; i<=n; i++)
(gdb) p sum
$4 = 3
(gdb) bt   <---------------------- 查看函数堆栈。
#0  func (n=250) at tst.c:6
#1  0x080483f1 in main () at tst.c:24
(gdb) finish   <---------------------- 退出函数。
Run till exit from #0  func (n=250) at tst.c:6
0x080483f1 in main () at tst.c:24
24                      printf("result[1-250] =  %d \n", func(250) );
Value returned is $5 = 31375
(gdb) c   <---------------------- 继续运行。
Continuing.
result[1-250] = 31375    <----------程序输出。
```

好了，有了以上的感性認識，還是讓我們來系統地認識一下gdb吧。

## 使用GDB

一般來說GDB主要調試的是C/C++的程序。要調試C/C++的程序，首先在編譯時，我們必須要把調試信息加到可執行文件中。使用編譯器（cc/gcc/g++）的 -g 參數可以做到這一點。如：

```
$gcc -g -Wall hello.c -o hello
$g++ -g -Wall hello.cpp -o hello
```

如果沒有-g，你將看不見程序的函數名、變量名，所代替的全是運行時的內存地址。當你用-g把調試信息加入之後，並成功編譯目標代碼以後，讓我們來看看如何用gdb來調試他。

啟動GDB的方法有以下幾種：

- gdb <program>

  program也就是你的執行文件，一般在當前目錄下。

- gdb <program> core

  用gdb同時調試一個運行程序和core文件，core是程序非法執行后core dump后產生的文件。

- gdb <program> <PID>

  如果你的程序是一個服務程序，那麼你可以指定這個服務程序運行時的進程ID。gdb會自動attach上去，並調試他。program應該在PATH環境變量中搜索得到。

GDB啟動時，可以加上一些GDB的啟動開關，詳細的開關可以用gdb -help查看。我在下面只例舉一些比較常用的參數：

-symbols <file>
-s <file>
　　從指定文件中讀取符號表。

-se file
　　從指定文件中讀取符號表信息，並把他用在可執行文件中。

-core <file>
-c <file>
　　調試時core dump的core文件。

-directory <directory>
-d <directory>
　　加入一個源文件的搜索路徑。默認搜索路徑是環境變量中PATH所定義的路徑。

## GDB的命令概貌

啟動gdb后，就你被帶入gdb的調試環境中，就可以使用gdb的命令開始調試程序了，gdb的命令可以使用help命令來查看，如下所示：

```
$ gdb
GNU gdb 6.7.1-debian
Copyright (C) 2007 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
(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 all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb)
```

gdb 的命令很多，gdb把之分成許多個種類。help命令只是例出gdb的命令種類，如果要看種類中的命令，可以使用help <class> 命令，如：help breakpoints，查看設置斷點的所有命令。也可以直接help <command>來查看命令的幫助。

gdb中，輸入命令時，可以不用打全命令，只用打命令的前幾個字符就可以了，當然，命令的前幾個字符應該要標志著一個唯一的命令，在Linux下，你可以敲擊兩次TAB鍵來補齊命令的全稱，如果有重複的，那麼gdb會把其例出來。

示例一：在進入函數func時，設置一個斷點。可以敲入break func，或是直接就是b func

```
(gdb) b func
Breakpoint 1 at 0x804837a: file tst.c, line 5.
```

示例二：敲入b按兩次TAB鍵，你會看到所有b打頭的命令：

```
(gdb) b
backtrace break bt
(gdb)
```

示例三：只記得函數的前綴，可以這樣：

```
(gdb) b make_ <按TAB键>
（再按下一次TAB键，你会看到：）
make_a_section_from_file make_environ
make_abs_section make_function_type
make_blockvector make_pointer_type
make_cleanup make_reference_type
make_command make_symbol_completion_list
(gdb) b make_
GDB把所有make开头的函数全部例出来给你查看。
```

示例四：調試C++的程序時，有可以函數名一樣。如：

```
(gdb) b 'bubble( M-?
bubble(double,double) bubble(int,int)
(gdb) b 'bubble(
```

你可以查看到C++中的所有的重載函數及參數。（注：M-?和“按兩次TAB鍵”是一個意思）

要退出gdb時，只用發quit或命令簡稱q就行了。

## GDB中運行UNIX的shell程序

在gdb環境中，你可以執行UNIX的shell的命令，使用gdb的shell命令來完成：

```
shell <command string>
```

調用UNIX的shell來執行<command string>，環境變量SHELL中定義的UNIX的shell將會被用來執行<command string>，如果SHELL沒有定義，那就使用UNIX的標準shell：/bin/sh。（在Windows中使用Command.com或cmd.exe）

還有一個gdb命令是make：

```
make <make-args>
```

可以在gdb中執行make命令來重新build自己的程序。這個命令等價于“shell make <make-args>”。

## 在GDB中運行程序

當以gdb <program>方式啟動gdb后，gdb會在PATH路徑和當前目錄中搜索<program>的源文件。如要確認gdb是否讀到源文件，可使用l或list命令，看看gdb是否能列出源代碼。

在gdb中，運行程序使用r或是run命令。程序的運行，你有可能需要設置下面四方面的事。

1、程序運行參數。

set args 可指定運行時參數。（如：set args 10 20 30 40 50）
show args 命令可以查看設置好的運行參數。

2、運行環境。

path <dir> 可設定程序的運行路徑。
show paths 查看程序的運行路徑。
set environment varname [=value] 設置環境變量。如：set env USER=hchen
show environment [varname] 查看環境變量。

3、工作目錄。

cd <dir> 相當於shell的cd命令。
pwd 顯示當前的所在目錄。

4、程序的輸入輸出。

info terminal 顯示你程序用到的終端的模式。
使用重定向控製程序輸出。如：run > outfile
tty命令可以指寫輸入輸出的終端設備。如：tty /dev/ttyb

## 調試已運行的程序

兩種方法：

1. 在UNIX下用ps查看正在運行的程序的PID（進程ID），然後用gdb <program> PID格式掛接正在運行的程序。
2. 先用gdb <program>關聯上源代碼，並進行gdb，在gdb中用attach命令來掛接進程的PID。並用detach來取消掛接的進程。

## 暫停/恢復程序運行

調試程序中，暫停程序運行是必須的，GDB可以方便地暫停程序的運行。你可以設置程序的在哪行停住，在什麼條件下停住，在收到什麼信號時停往等等。以便於你查看運行時的變量，以及運行時的流程。

當進程被gdb停住時，你可以使用info program 來查看程序的是否在運行，進程號，被暫停的原因。

在gdb中，我們可以有以下幾種暫停方式：斷點（BreakPoint）、觀察點（Watch Point）、捕捉點（Catch Point）、信號（Signals）、線程停止（Thread Stops）。如果要恢復程序運行，可以使用c或是 continue命令。

## 設置斷點（**Break Points**）

我們用break命令來設置斷點。下面有幾點設置斷點的方法：

break <function>
　　在進入指定函數時停住。C++中可以使用class::function或function(type,type)格式來指定函數名。

break <linenum>
　　在指定行號停住。

break +offset
break -offset
　　在當前行號的前面或後面的offset行停住。offiset為自然數。

break filename：linenum
　　在源文件filename的linenum行處停住。

break filename：function
在源文件filename的function函數的入口處停住。

break *address
在程序運行的內存地址處停住。

break
break命令沒有參數時，表示在下一條指令處停住。

break ... if <condition>
...可以是上述的參數，condition表示條件，在條件成立時停住。比如在循環體中，可以設置break if i=100，表示當i為100時停住程序。

查看斷點時，可使用info命令，如下所示：（注：n表示斷點號）

- info breakpoints [n]
- info break [n]

## 設置觀察點（**WatchPoint**）

觀察點一般來觀察某個表達式（變量也是一種表達式）的值是否有變化了，如果有變化，馬上停住程序。我們有下面的幾種方法來設置觀察點：

watch <expr>
為表達式（變量）expr設置一個觀察點。一表達式值有變化時，馬上停住程序。

rwatch <expr>
當表達式（變量）expr被讀時，停住程序。

awatch <expr>
當表達式（變量）的值被讀或被寫時，停住程序。

info watchpoints
列出當前所設置了的所有觀察點。

## 設置捕捉點（**CatchPoint**）

你可設置捕捉點來捕捉程序運行時的一些事件。如：載入共享庫（動態鏈接庫）或是C++的異常。設置捕捉點的格式為：

- catch <event>

當event發生時，停住程序。event可以是下面的內容：

1. throw 一個C++拋出的異常。（throw為關鍵字）
2. catch 一個C++捕捉到的異常。（catch為關鍵字）
3. exec 調用系統調用exec時。（exec為關鍵字，目前此功能只在HP-UX下有用）
4. fork 調用系統調用fork時。（fork為關鍵字，目前此功能只在HP-UX下有用）
5. vfork 調用系統調用vfork時。（vfork為關鍵字，目前此功能只在HP-UX下有用）
6. load 或 load <libname> 載入共享庫（動態鏈接庫）時。（load為關鍵字，目前此功能只在HP-UX下有用）

7. unload 或 unload <libname> 卸載共享庫（動態鏈接庫）時。（unload為關鍵字，目前此功能只在HP-UX下有用）

- tcatch <event>

只設置一次捕捉點，當程序停住以後，應點被自動刪除。

## 維護停止點

上面說了如何設置程序的停止點，GDB中的停止點也就是上述的三類。在GDB中，如果你覺得已定義好的停止點沒有用了，你可以使用delete、clear、disable、enable這幾個命令來進行維護。

clear
　　清除所有的已定義的停止點。

clear <function>
clear <filename：function>
　　清除所有設置在函數上的停止點。

clear <linenum>
clear <filename：linenum>
　　清除所有設置在指定行上的停止點。

delete [breakpoints] [range...]
　　刪除指定的斷點，breakpoints為斷點號。如果不指定斷點號，則表示刪除所有的斷點。range 表示斷點號的範圍（如：3-7）。其簡寫命令為d。

比刪除更好的一種方法是disable停止點，disable了的停止點，GDB不會刪除，當你還需要時，enable即可，就好像回收站一樣。

disable [breakpoints] [range...]
　　disable所指定的停止點，breakpoints為停止點號。如果什麼都不指定，表示disable所有的停止點。簡寫命令是dis.

enable [breakpoints] [range...]
　　enable所指定的停止點，breakpoints為停止點號。

enable [breakpoints] once range...
　　enable所指定的停止點一次，當程序停止后，該停止點馬上被GDB自動disable。

enable [breakpoints] delete range...
　　enable所指定的停止點一次，當程序停止后，該停止點馬上被GDB自動刪除。

## 停止條件維護

前面在說到設置斷點時，我們提到過可以設置一個條件，當條件成立時，程序自動停止，這是一個非常強大的功能，這裏，我想專門說說這個條件的相關維護命令。一般來說，為斷點設置一個條件，我們使用if關鍵詞，後面跟其斷點條件。並且，條件設置好后，我們可以用condition命令來修改斷點的條件。（只有break和watch命令支持if，catch目前暫不支持if）

condition <bnum> <expression>
修改斷點號為bnum的停止條件為expression。

condition <bnum>
清除斷點號為bnum的停止條件。

還有一個比較特殊的維護命令ignore，你可以指定程序運行時，忽略停止條件幾次。

ignore <bnum> <count>
表示忽略斷點號為bnum的停止條件count次。

## 為停止點設定運行命令

我們可以使用GDB提供的command命令來設置停止點的運行命令。也就是說，當運行的程序在被停止住時，我們可以讓其自動運行一些別的命令，這很有利行自動化調試。對基於GDB的自動化調試是一個強大的支持。

```
commands [bnum]
... command-list ...
end
```

為斷點號bnum指寫一個命令列表。當程序被該斷點停住時，gdb會依次運行命令列表中的命令。例如：

```
break foo if x>0
commands
printf "x is  %d\n",x
continue
end
```

斷點設置在函數foo中，斷點條件是x>0，如果程序被斷住后，也就是，一旦x的值在foo函數中大於0，GDB會自動打印出x的值，並繼續運行程序。

如果你要清除斷點上的命令序列，那麼只要簡單的執行一下commands命令，並直接在打個end就行了。

## 斷點菜單

在C++中，可能會重複出現同一個名字的函數若干次（函數重載），在這種情況下，break <function>不能告訴GDB要停在哪個函數的入口。當然，你可以使用break <function(type)>也就是把函數的參數類型告訴GDB，以指定一個函數。否則的話，GDB會給你列出一個斷點菜單供你選擇你所需要的斷點。你只要輸入你菜單列表中的編號就可以了。如：

```
(gdb) b String::after
[0] cancel
[1] all
[2] file:String.cc; line number:867
[3] file:String.cc; line number:860
[4] file:String.cc; line number:875
[5] file:String.cc; line number:853
[6] file:String.cc; line number:846
[7] file:String.cc; line number:735
> 2 4 6
Breakpoint 1 at 0xb26c: file String.cc, line 867.
Breakpoint 2 at 0xb344: file String.cc, line 875.
Breakpoint 3 at 0xafcc: file String.cc, line 846.
Multiple breakpoints were set.
Use the "delete" command to delete unwanted
breakpoints.
(gdb)
```

可見，GDB列出了所有after的重載函數，你可以選一下列表編號就行了。0表示放棄設置斷點，1表示所有函數都設置斷點。

## 恢復程序運行和單步調試

當程序被停住了，你可以用continue命令恢復程序的運行直到程序結束，或下一個斷點到來。也可以使用step或next命令單步跟蹤程序。

continue [ignore-count]
c [ignore-count]
fg [ignore-count]
　　恢復程序運行，直到程序結束，或是下一個斷點到來。ignore-count表示忽略其後的斷點次數。continue，c，fg三個命令都是一樣的意思。

step <count>
　　單步跟蹤，如果有函數調用，他會進入該函數。進入函數的前提是，此函數被編譯有debug信息。很像VC等工具中的step in。後面可以加count也可以不加，不加表示一條條地執行，加表示執行後面的count條指令，然後再停住。

next <count>
　　同樣單步跟蹤，如果有函數調用，他不會進入該函數。很像VC等工具中的step over。後面可以加count也可以不加，不加表示一條條地執行，加表示執行後面的count條指令，然後再停住。

set step-mode
set step-mode on
　　打開step-mode模式，於是，在進行單步跟蹤時，程序不會因為沒有debug信息而不停住。這個參數有很利於查看機器碼。

set step-mod off
　　關閉step-mode模式。

finish
　　運行程序，直到當前函數完成返回。並打印函數返回時的堆棧地址和返回值及參數值等信息。

until 或 u
　　當你厭倦了在一個循環體內單步跟蹤時，這個命令可以運行程序直到退出循環體。

stepi 或 si
nexti 或 ni
: 單步跟蹤一條機器指令！一條程序代碼有可能由數條機器指令完成，stepi和nexti可以單步執行機器指令。與之一樣有相同功能的命令是 “display/i $pc” ，當運行完這個命令后，單步跟蹤會在打出程序代碼的同時打出機器指令（也就是彙編代碼）

## 信號（**Signals**）

信號是一種軟中斷，是一種處理異步事件的方法。一般來說，操作系統都支持許多信號。尤其是 UNIX，比較重要應用程序一般都會處理信號。UNIX定義了許多信號，比如SIGINT表示中斷字符信號，也就是Ctrl+C的信號，SIGBUS表示硬件故障的信號；SIGCHLD表示子進程狀態改變信號；SIGKILL表示終止程序運行的信號，等等。信號量編程是UNIX下非常重要的一種技術。

GDB有能力在你調試程序的時候處理任何一種信號，你可以告訴GDB需要處理哪一種信號。你可以要求GDB收到你所指定的信號時，馬上停住正在運行的程序，以供你進行調試。你可以用GDB的handle命令來完成這一功能。

```
handle <signal> <keywords...>
```

在GDB中定義一個信號處理。信號<signal>可以以SIG開頭或不以SIG開頭，可以用定義一個要處理信號的範圍（如：SIGIO- SIGKILL，表示處理從SIGIO信號到SIGKILL的信號，其中包括SIGIO，SIGIOT，SIGKILL三個信號），也可以使用關鍵字 all來標明要處理所有的信號。一旦被調試的程序接收到信號，運行程序馬上會被GDB停住，以供調試。其<keywords>可以是以下幾種關鍵字的一個或多個。

nostop
: 當被調試的程序收到信號時，GDB不會停住程序的運行，但會打出消息告訴你收到這種信號。

stop
: 當被調試的程序收到信號時，GDB會停住你的程序。

print
: 當被調試的程序收到信號時，GDB會顯示出一條信息。

noprint
: 當被調試的程序收到信號時，GDB不會告訴你收到信號的信息。

pass
noignore
: 當被調試的程序收到信號時，GDB不處理信號。這表示，GDB會把這個信號交給被調試程序處理。

nopass
ignore
: 當被調試的程序收到信號時，GDB不會讓被調試程序來處理這個信號。

info signals
info handle
: 查看有哪些信號在被GDB檢測中。

## 線程（**Thread Stops**）

如果你程序是多線程的話，你可以定義你的斷點是否在所有的線程上，或是在某個特定的線

程。GDB很容易幫你完成這一工作。

```
break <linespec> thread <threadno>
break <linespec> thread <threadno> if ...
```

linespec指定了斷點設置在的源程序的行號。threadno指定了線程的ID，注意，這個ID是GDB分配的，你可以通過“info threads”命令來查看正在運行程序中的線程信息。如果你不指定thread <threadno>則表示你的斷點設在所有線程上面。你還可以為某線程指定斷點條件。如：

```
(gdb) break frik.c:13 thread 28 if bartab > lim
```

當你的程序被GDB停住時，所有的運行線程都會被停住。這方便你你查看運行程序的總體情況。而在你恢復程序運行時，所有的線程也會被恢復運行。那怕是主進程在被單步調試時。

## 查看栈信息

當程序被停住了，你需要做的第一件事就是查看程序是在哪裡停住的。當你的程序調用了一個函數，函數的地址，函數參數，函數內的局部變量都會被壓入“栈”（Stack）中。你可以用GDB命令來查看當前的栈中的信息。

下面是一些查看函數調用栈信息的GDB命令：

backtrace
bt
　　打印當前的函數調用栈的所有信息。如：

```
(gdb) bt
#0  func (n=250) at tst.c:6
#1  0x08048524 in main (argc=1, argv=0xbffff674) at tst.c:30
#2  0x400409ed in __libc_start_main () from /lib/libc.so.6
```

從上可以看出函數的調用栈信息：__libc_start_main --> main() --> func()

backtrace <n>
bt <n>
　　n是一個正整數，表示只打印栈頂上n層的栈信息。

backtrace <-n>
bt <-n>
　　-n表一個負整數，表示只打印栈底下n層的栈信息。

如果你要查看某一層的信息，你需要切換當前栈，一般來說，程序停止時，最頂層的栈就是當前栈，如果你要查看栈下面層的詳細信息，首先要做的是切換當前栈。

frame <n>
f <n>
　　n是一個從0開始的整數，是栈中的層編號。比如：frame 0，表示栈頂，frame 1，表示栈的第二層。

up <n>
表示向棧的上面移動n層，可以不打n，表示向上移動一層。

down <n>
表示向棧的下面移動n層，可以不打n，表示向下移動一層。

上面的命令，都會打印出移動到的棧層的信息。如果你不想讓其打出信息。你可以使用這三個命令：

```
select-frame <n> 对应于 frame 命令。
up-silently <n> 对应于 up 命令。
down-silently <n> 对应于 down 命令。
```

查看當前棧層的信息，你可以用以下GDB命令：

frame 或 f
會打印出這些信息：棧的層編號，當前的函數名，函數參數值，函數所在文件及行號，函數執行到的語句。

info frame
info f
這個命令會打印出更為詳細的當前棧層的信息，只不過，大多數都是運行時的內存地址。比如：函數地址，調用函數的地址，被調用函數的地址，目前的函數是由什麼樣的程序語言寫成的、函數參數地址及值、局部變量的地址等等。如：

```
(gdb) info f
Stack level 0, frame at 0xbffff5d4:
eip = 0x804845d in func (tst.c:6); saved eip 0x8048524
called by frame at 0xbffff60c
source language c.
Arglist at 0xbffff5d4, args: n=250
Locals at 0xbffff5d4, Previous frame's sp is 0x0
Saved registers:
ebp at 0xbffff5d4, eip at 0xbffff5d8
```

info args
打印出當前函數的參數名及其值。

info locals
打印出當前函數中所有局部變量及其值。

info catch
打印出當前的函數中的異常處理信息。

## 查看源程序

### 顯示源代碼

GDB 可以打印出所調試程序的源代碼，當然，在程序編譯時一定要加上-g的參數，把源程序信息編譯到執行文件中。不然就看不到源程序了。當程序停下來以後，GDB會報告程序停在了那個文件的第幾行上。你可以用list命令來打印程序的源代碼。還是來看一看查看源代碼的GDB命令吧。

list <linenum>
　　顯示程序第linenum行的周圍的源程序。
list <function>
　　顯示函數名為function的函數的源程序。
list
　　顯示當前行後面的源程序。
list -
　　顯示當前行前面的源程序。

一般是打印當前行的上5行和下5行，如果顯示函數是是上2行下8行，默認是10行，當然，你也可以定製顯示的範圍，使用下面命令可以設置一次顯示源程序的行數。

set listsize <count>
　　設置一次顯示源代碼的行數。
show listsize
　　查看當前listsize的設置。

list命令還有下面的用法：

list <first>, <last>
　　顯示從first行到last行之間的源代碼。

list , <last>
　　顯示從當前行到last行之間的源代碼。

list +
　　往後顯示源代碼。

一般來說在list後面可以跟以下這些參數：

```
<linenum>    行号。
<+offset>    当前行号的正偏移量。
<-offset>    当前行号的负偏移量。
<filename:linenum>   哪个文件的哪一行。
<function>   函数名。
<filename:function> 哪个文件中的哪个函数。
<*address>   程序运行时的语句在内存中的地址。
```

## 搜索源代碼

不僅如此，GDB還提供了源代碼搜索的命令：

forward-search <regexp>
search <regexp>
　　向前面搜索。
reverse-search <regexp>
　　全部搜索。

其中，<regexp>就是正則表達式，也主一個字符串的匹配模式，關於正則表達式，我就不在這裏講了，還請各位查看相關資料。

## 指定源文件的路徑

某些時候，用-g編譯過後的執行程序中只是包括了源文件的名字，沒有路徑名。GDB提供了可以讓你指定源文件的路徑的命令，以便GDB進行搜索。

directory <dirname ... >
dir <dirname ... >
　　加一個源文件路徑到當前路徑的前面。如果你要指定多個路徑，UNIX下你可以使用“:”，Windows下你可以使用“;”。
directory
　　清除所有的自定義的源文件搜索路徑信息。
show directories
　　顯示定義了的源文件搜索路徑。

## 源代碼的內存

你可以使用info line命令來查看源代碼在內存中的地址。info line後面可以跟“行號”，“函數名”，“文件名:行號”，“文件名:函數名”，這個命令會打印出所指定的源碼在運行時的內存地址，如：

```
(gdb) info line tst.c:func
Line 5 of "tst.c" starts at address 0x8048456 <func+6> and ends at 0x804845d <func+13>.
```

還有一個命令（disassemble）你可以查看源程序的當前執行時的機器碼，這個命令會把目前內存中的指令dump出來。如下面的示例表示查看函數func的彙編代碼。

```
(gdb) disassemble func
Dump of assembler code for function func:
0x8048450 <func>:       push   %ebp
0x8048451 <func+1>:     mov    %esp,%ebp
0x8048453 <func+3>:     sub    $0x18,%esp
0x8048456 <func+6>:     movl   $0x0,0xfffffffc(%ebp)
0x804845d <func+13>:    movl   $0x1,0xfffffff8(%ebp)
0x8048464 <func+20>:    mov    0xfffffff8(%ebp),%eax
0x8048467 <func+23>:    cmp    0x8(%ebp),%eax
0x804846a <func+26>:    jle    0x8048470 <func+32>
0x804846c <func+28>:    jmp    0x8048480 <func+48>
0x804846e <func+30>:    mov    %esi,%esi
0x8048470 <func+32>:    mov    0xfffffff8(%ebp),%eax
0x8048473 <func+35>:    add    %eax,0xfffffffc(%ebp)
0x8048476 <func+38>:    incl   0xfffffff8(%ebp)
0x8048479 <func+41>:    jmp    0x8048464 <func+20>
0x804847b <func+43>:    nop
0x804847c <func+44>:    lea    0x0(%esi,1),%esi
0x8048480 <func+48>:    mov    0xfffffffc(%ebp),%edx
0x8048483 <func+51>:    mov    %edx,%eax
0x8048485 <func+53>:    jmp    0x8048487 <func+55>
0x8048487 <func+55>:    mov    %ebp,%esp
0x8048489 <func+57>:    pop    %ebp
0x804848a <func+58>:    ret
End of assembler dump.
```

## 查看運行時數據

在你調試程序時，當程序被停住時，你可以使用print命令（簡寫命令為p），或是同義命令inspect來查看當前程序的運行數據。print命令的格式是：

```
print <expr>
print /<f> <expr>
```

<expr>是表達式，是你所調試的程序的語言的表達式（GDB可以調試多種編程語言），<f>是輸出的格式，比如，如果要把表達式按16進制的格式輸出，那麼就是/x。

## 表達式

print和許多GDB的命令一樣，可以接受一個表達式，GDB會根據當前的程序運行的數據來計算這個表達式，既然是表達式，那麼就可以是當前程序運行中的const常量、變量、函數等內容。可惜的是GDB不能使用你在程序中所定義的宏。

表達式的語法應該是當前所調試的語言的語法，由於C/C++是一種大眾型的語言，所以，本文中的例子都是關於C/C++的。（而關於用GDB調試其它語言的章節，我將在後面介紹）

在表達式中，有幾種GDB所支持的操作符，它們可以用在任何一種語言中。

**@**

是一個和數組有關的操作符，在後面會有更詳細的說明。

**::**

指定一個在文件或是一個函數中的變量。

**{<type>} <addr>**

表示一個指向內存地址<addr>的類型為type的一個對象。

## 程序變量

在GDB中，你可以隨時查看以下三種變量的值：

1. 全局變量（所有文件可見的）
2. 靜態全局變量（當前文件可見的）
3. 局部變量（當前Scope可見的）

如果你的局部變量和全局變量發生衝突（也就是重名），一般情況下是局部變量會隱藏全局變量，也就是說，如果一個全局變量和一個函數中的局部變量同名時，如果當前停止點在函數中，用print顯示出的變量的值會是函數中的局部變量的值。如果此時你想查看全局變量的值時，你可以使用“::”操作符：

```
file::variable
function::variable
```

可以通過這種形式指定你所想查看的變量，是哪個文件中的或是哪個函數中的。例如，查看文件f2.c中的全局變量x的值：

```
(gdb) p 'f2.c'::x
```

當然，“::”操作符會和C++中的發生衝突，GDB能自動識別“::” 是否C++的操作符，所以你不必擔心在調試C++程序時會出現異常。

另外，需要注意的是，如果你的程序編譯時開啟了優化選項，那麼在用GDB調試被優化過的程序時，可能會發生某些變量不能訪問，或是取值錯誤碼的情況。這個是很正常的，因為優化程序會刪改你的程序，整理你程序的語句順序，剔除一些無意義的變量等，所以在GDB調試這種程序時，運行時的指令和你所編寫指令就有不一樣，也就會出現你所想象不到的結果。對付這種情況時，需要在編譯程序時關閉編譯優化。一般來說，幾乎所有的編譯器都支持編譯優化的開關，例如，GNU 的C/C++編譯器GCC，你可以使用“-gstabs”選項來解決這個問題。關於編譯器的參數，還請查看編譯器的使用說明文檔。

## 数組

有時候，你需要查看一段連續的內存空間的值。比如數組的一段，或是動態分配的數據的大小。你可以使用GDB的“@”操作符，“@”的左邊是第一個內存的地址的值，“@”的右邊則你你想查看內存的長度。例如，你的程序中有這樣的語句：

```
int *array = (int *) malloc (len * sizeof (int));
```

於是，在GDB調試過程中，你可以以如下命令顯示出這個動態數組的取值：

```
p *array@len
```

@的左邊是數組的首地址的值，也就是變量array所指向的內容，右邊則是數據的長度，其保存在變量len中，其輸出結果，大約是下面這個樣子的：

```
(gdb) p *array@len
$1 = {2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40}
```

如果是靜態數組的話，可以直接用print數組名，就可以顯示數組中所有數據的內容了。

## 輸出格式

一般來說，GDB會根據變量的類型輸出變量的值。但你也可以自定義GDB的輸出的格式。例如，你想輸出一個整數的十六進制，或是二進制來查看這個整型變量的中的位的情況。要做到這樣，你可以使用GDB的數據顯示格式：

```
x   按十六进制格式显示变量。
d   按十进制格式显示变量。
u   按十六进制格式显示无符号整型。
o   按八进制格式显示变量。
t   按二进制格式显示变量。
a   按十六进制格式显示变量。
c   按字符格式显示变量。
f   按浮点数格式显示变量。
```

```
(gdb) p i
$21 = 101
(gdb) p/a i
$22 = 0x65
(gdb) p/c i
$23 = 101 'e'
(gdb) p/f i
$24 = 1.41531145e-43
(gdb) p/x i
$25 = 0x65
(gdb) p/t i
$26 = 1100101
```

## 查看內存

你可以使用examine命令（簡寫是x）來查看內存地址中的值。x命令的語法如下所示：

```
x/<n/f/u> <addr>
```

n、f、u是可選的參數。

- n 是一個正整數，表示顯示內存的長度，也就是說從當前地址向後顯示幾個地址的內容。
- f 表示顯示的格式，參見上面。如果地址所指的是字符串，那麼格式可以是s，如果地址是指令地址，那麼格式可以是i。
- u 表示從當前地址往後請求的字節數，如果不指定的話，GDB默認是4個bytes。u參數可以用下面的字符來代替，b表示單字節，h表示雙字節，w表示四字節，g表示八字節。當我們指定了字節長度后，GDB會從指內存定的內存地址開始，讀寫指定字節，並把其當作一個值取出來。

<addr>表示一個內存地址。

n/f/u三個參數可以一起使用。例如：

命令：x/3uh 0x54320 表示，從內存地址0x54320讀取內容，h表示以雙字節為一個單位，3表示三個單位，u表示按十六進制顯示。

## 自動顯示

你可以設置一些自動顯示的變量，當程序停住時，或是在你單步跟蹤時，這些變量會自動顯示。相關的GDB命令是display。

```
display <expr>
display/<fmt> <expr>
display/<fmt> <addr>
```

expr是一個表達式，fmt表示顯示的格式，addr表示內存地址，當你用display設定好了一個或多個表達式后，只要你的程序被停下來，GDB會自動顯示你所設置的這些表達式的值。

格式i和s同樣被display支持，一個非常有用的命令是：

```
display/i $pc
```

$pc是GDB的環境變量，表示着指令的地址，/i則表示輸出格式為機器指令碼，也就是彙編。於是當程序停下后，就會出現源代碼和機器指令碼相對應的情形，這是一個很有意思的功能。

下面是一些和display相關的GDB命令：

```
undisplay <dnums...>
delete display <dnums...>
```

刪除自動顯示，dnums意為所設置好了的自動顯式的編號。如果要同時刪除幾個，編號可以用空格分隔，如果要刪除一個範圍內的編號，可以用減號表示（如：2-5）

```
disable display <dnums...>
enable display <dnums...>
```

disable和enalbe不刪除自動顯示的設置，而只是讓其失效和恢復。

```
info display
```

查看display設置的自動顯示的信息。GDB會打出一張表格，向你報告當然調試中設置了多少個自動顯示設置，其中包括，設置的編號，表達式，是否enable。

## 設置顯示選項

GDB中關於顯示的選項比較多，這裏我只例舉大多數常用的選項。

set print address
set print address on
　　打開地址輸出，當程序顯示函數信息時，GDB會顯出函數的參數地址。系統默認為打開的，如：

```
(gdb) f
#0  set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")
    at input.c:530
530         if (lquote  != def_lquote)
```

set print address off
　　關閉函數的參數地址顯示，如：

```
(gdb) set print addr off
(gdb) f
#0  set_quotes (lq="<<", rq=">>") at input.c:530
530         if (lquote  != def_lquote)
```

show print address
　　查看當前地址顯示選項是否打開。

set print array
set print array on
　　打開數組顯示，打開后當數組顯示時，每個元素佔一行，如果不打開的話，每個元素則

以逗號分隔。這個選項默認是關閉的。與之相關的兩個命令如下，我就不再多說了。

set print array off
show print array
set print elements <number-of-elements>
這個選項主要是設置數組的，如果你的數組太大了，那麼就可以指定一個<number-of-elements>來指定數據顯示的最大長度，當到達這個長度時，GDB就不再往下顯示了。如果設置為0，則表示不限制。

show print elements
查看print elements的選項信息。

set print null-stop <on/off>
如果打開了這個選項，那麼當顯示字符串時，遇到結束符則停止顯示。這個選項默認為off。

set print pretty on
如果打開printf pretty這個選項，那麼當GDB顯示結構體時會比較漂亮。如：

```
$1 = {
  next = 0x0,
  flags = {
    sweet = 1,
    sour = 1
  },
  meat = 0x54 "Pork"
}
```

set print pretty off
關閉printf pretty這個選項，GDB顯示結構體時會如下顯示：

```
$1 = {next = 0x0, flags = {sweet = 1, sour = 1}, meat = 0x54 "Pork"}
```

show print pretty
查看GDB是如何顯示結構體的。

set print sevenbit-strings <on/off>
設置字符顯示，是否按“\nnn”的格式顯示，如果打開，則字符串或字符數據按\nnn顯示，如“\065”。

show print sevenbit-strings
查看字符顯示開關是否打開。

set print union <on/off>
設置顯示結構體時，是否顯式其內的聯合體數據。例如有以下數據結構：

```
typedef enum {Tree, Bug} Species;
typedef enum {Big_tree, Acorn, Seedling} Tree_forms;
typedef enum {Caterpillar, Cocoon, Butterfly}
             Bug_forms;
```

```
struct thing {
  Species it;
  union {
    Tree_forms tree;
    Bug_forms bug;
  } form;
};
```

```
struct thing foo = {Tree, {Acorn}};
```

當打開這個開關時，執行 p foo 命令后，會如下顯示：

```
$1 = {it = Tree, form = {tree = Acorn, bug = Cocoon}}
```

當關閉這個開關時，執行 p foo 命令后，會如下顯示：

```
$1 = {it = Tree, form = {...}}
```

show print union
查看聯合體數據的顯示方式

set print object <on/off>:在C++中，如果一個對象指針指向其派生類，如果打開這個選項，GDB會自動按照虛方法調用的規則顯示輸出，如果關閉這個選項的話，GDB就不管虛函數表了。這個選項默認是off。

show print object
查看對象選項的設置。

set print static-members <on/off>
這個選項表示，當顯示一個C++對象中的內容是，是否顯示其中的靜態數據成員。默認是on。

show print static-members
查看靜態數據成員選項設置。

set print vtbl <on/off>
當此選項打開時，GDB將用比較規整的格式來顯示虛函數表時。其默認是關閉的。

show print vtbl
查看虛函數顯示格式的選項。

## 歷史記錄

當你用GDB的print查看程序運行時的數據時，你每一個print都會被GDB記錄下來。GDB會以$1, $2, $3 .....這樣的方式為你每一個print命令編上號。於是，你可以使用這個編號訪問以前的表達式，如$1。這個功能所帶來的好處是，如果你先前輸入了一個比較長的表達式，如果你還想查看這個表達式的值，你可以使用歷史記錄來訪問，省去了重複輸入。

## GDB環境變量

你可以在GDB的調試環境中定義自己的變量，用來保存一些調試程序中的運行數據。要定義一個GDB的變量很簡單隻需。使用GDB的set命令。GDB的環境變量和UNIX一樣，也是以$起頭。如：

```
set $foo = *object_ptr
```

使用環境變量時，GDB會在你第一次使用時創建這個變量，而在以後的使用中，則直接對其賦值。環境變量沒有類型，你可以給環境變量定義任一的類型。包括結構體和數組。

show convenience
　　該命令查看當前所設置的所有的環境變量。

這是一個比較強大的功能，環境變量和程序變量的交互使用，將使得程序調試更為靈活便捷。例如：

```
set $i = 0
print bar[$i++]->contents
```

於是，當你就不必，print bar[0]->contents, print bar[1]->contents地輸入命令了。輸入這樣的命令后，只用敲回車，重複執行上一條語句，環境變量會自動累加，從而完成逐個輸出的功能。

### 查看寄存器

要查看寄存器的值，很簡單，可以使用如下命令：

info registers
　　查看寄存器的情況。（除了浮點寄存器）

info all-registers
　　查看所有寄存器的情況。（包括浮點寄存器）

info registers <regname ...>
　　查看所指定的寄存器的情況。

寄存器中放置了程序運行時的數據，比如程序當前運行的指令地址（ip），程序的當前堆棧地址（sp）等等。你同樣可以使用print命令來訪問寄存器的情況，只需要在寄存器名字前加一個$符號就可以了。如：p $eip。

## 改變程序的執行

一旦使用GDB掛上被調試程序，當程序運行起來后，你可以根據自己的調試思路來動態地在GDB中更改當前被調試程序的運行線路或是其變量的值，這個強大的功能能夠讓你更好的調試你的程序，比如，你可以在程序的一次運行中走遍程序的所有分支。

### 修改變量值

修改被調試程序運行時的變量值，在GDB中很容易實現，使用GDB的print命令即可完成。

如：

```
(gdb) print x=4
```

x=4這個表達式是C/C++的語法，意為把變量x的值修改為4，如果你當前調試的語言是Pascal，那麼你可以使用Pascal的語法：x:=4。

在某些時候，很有可能你的變量和GDB中的參數衝突，如：

```
(gdb) whatis width
type = double
(gdb) p width
$4 = 13
(gdb) set width=47
Invalid syntax in expression.
```

因為，set width是GDB的命令，所以，出現了“Invalid syntax in expression”的設置錯誤，此時，你可以使用set var命令來告訴GDB，width不是你GDB的參數，而是程序的變量名，如：

```
(gdb) set var width=47
```

另外，還可能有些情況，GDB並不報告這種錯誤，所以保險起見，在你改變程序變量取值時，最好都使用set var格式的GDB命令。

## 跳轉執行

一般來說，被調試程序會按照程序代碼的運行順序依次執行。GDB提供了亂序執行的功能，也就是說，GDB可以修改程序的執行順序，可以讓程序執行隨意跳躍。這個功能可以由GDB的jump命令來完：

jump <linespec>
: 指定下一條語句的運行點。<linespce>可以是文件的行號，可以是file:line格式，可以是+num這種偏移量格式。表示着下一條運行語句從哪裡開始。

jump <address>
: 這裏的<address>是代碼行的內存地址。

注意，jump命令不會改變當前的程序棧中的內容，所以，當你從一個函數跳到另一個函數時，當函數運行完返回時進行彈棧操作時必然會發生錯誤，可能結果還是非常奇怪的，甚至於產生程序Core Dump。所以最好是同一個函數中進行跳轉。

熟悉彙編的人都知道，程序運行時，有一個寄存器用於保存當前代碼所在的內存地址。所以，jump命令也就是改變了這個寄存器中的值。於是，你可以使用“set $pc”來更改跳轉執行的地址。如：

```
set $pc = 0x485
```

## 產生信號量

使用singal命令，可以產生一個信號量給被調試的程序。如：中斷信號Ctrl+C。這非常方便於

程序的調試，可以在程序運行的任意位置設置斷點，並在該斷點用GDB產生一個信號量，這種精確地在某處產生信號非常有利程序的調試。

語法是：signal <singal>，UNIX的系統信號量通常從1到15。所以<singal>取值也在這個範圍。

single命令和shell的kill命令不同，系統的kill命令發信號給被調試程序時，是由GDB截獲的，而single命令所發出一信號則是直接發給被調試程序的。

### 強制函数返回

如果你的調試斷點在某個函數中，並還有語句沒有執行完。你可以使用return命令強制函數忽略還沒有執行的語句並返回。

```
return
return <expression>
   使用return命令取消当前函数的执行，并立即返回，如果指定了<expression>，那么该表达式的值会被认作函数的返回值。
```

### 強制調用函数

```
call <expr>
```

表達式中可以一是函數，以此達到強制調用函數的目的。並顯示函數的返回值，如果函數返回值是void，那麼就不顯示。

另一個相似的命令也可以完成這一功能——print，print後面可以跟表達式，所以也可以用他來調用函數，print和call的不同是，如果函數返回void，call則不顯示，print則顯示函數返回值，並把該值存入歷史數據中。

## 在不同語言中使用**GDB**

GDB支持下列語言：C, C++, Fortran, PASCAL, Java, Chill, assembly, 和 Modula-2。一般說來，GDB會根據你所調試的程序來確定當前的調試語言，比如：發現文件名後綴為“.c”的，GDB會認為是C程序。文件名後綴為 “.C, .cc, .cp, .cpp, .cxx, .c++”的，GDB會認為是C++程序。而後綴是“.f, .F”的，GDB會認為是Fortran程序，還有，後綴為如果是“.s, .S”的會認為是彙編語言。

也就是說，GDB會根據你所調試的程序的語言，來設置自己的語言環境，並讓GDB的命令跟着語言環境的改變而改變。比如一些GDB命令需要用到表達式或變量時，這些表達式或變量的語法，完全是根據當前的語言環境而改變的。例如C/C++中對指針的語法是*p，而在Modula-2中則是p^。並且，如果你當前的程序是由幾種不同語言一同編譯成的，那到在調試過程中，GDB也能根據不同的語言自動地切換語言環境。這種跟着語言環境而改變的功能，真是體貼開發人員的一種設計。

下面是幾個相關於GDB語言環境的命令：

show language
: 查看當前的語言環境。如果GDB不能識為你所調試的編程語言，那麼，C語言被認為是默認的環境。

info frame
　　查看當前函數的程序語言。

info source
　　查看當前文件的程序語言。

如果GDB沒有檢測出當前的程序語言，那麼你也可以手動設置當前的程序語言。使用set language命令即可做到。

當set language命令后什麼也不跟的話，你可以查看GDB所支持的語言種類：

```
(gdb) set language
The currently understood settings are:
```

```
local or auto    Automatic setting based on source file
c                Use the C language
c++              Use the C++ language
asm              Use the Asm language
chill            Use the Chill language
fortran          Use the Fortran language
java             Use the Java language
modula-2         Use the Modula-2 language
pascal           Use the Pascal language
scheme           Use the Scheme language
```

於是你可以在set language後跟上被列出來的程序語言名，來設置當前的語言環境。

## 後記

GDB是一個強大的命令行調試工具。大家知道命令行的強大就是在於，其可以形成執行序列，形成腳本。UNIX下的軟件全是命令行的，這給程序開發提代供了極大的便利，命令行軟件的優勢在於，它們可以非常容易的集成在一起，使用幾個簡單的已有工具的命令，就可以做出一個非常強大的功能。

於是UNIX下的軟件比Windows下的軟件更能有機地結合，各自發揮各自的長處，組合成更為強勁的功能。而Windows下的圖形軟件基本上是各自為營，互相不能調用，很不利於各種軟件的相互集成。在這裏並不是要和Windows做個什麼比較，所謂“寸有所長，尺有所短”，圖形化工具還是有不如命令行的地方。（看到這句話時，希望各位千萬再也不要認為我就是“鄙視圖形界面”，和我抬杠了 ）

我是根據版本為5.1.1的GDB所寫的這篇文章，所以可能有些功能已被修改，或是又有更為強勁的功能。而且，我寫得非常倉促，寫得比較簡略，並且，其中我已經看到有許多錯別字了（我用五筆，所以錯字讓你看不懂），所以，我在這裏對我文中的差錯表示萬分的歉意。

文中所羅列的GDB的功能時，我只是羅列了一些常用的GDB的命令和使用方法，其實，我這裏只講述的功能大約只佔GDB所有功能的60%吧，詳細的文檔，還是請查看GDB的幫助和使用手冊吧，或許，過段時間，如果我有空，我再寫一篇GDB的高級使用。

我個人非常喜歡GDB的自動調試的功能，這個功能真的很強大，試想，我在UNIX下寫個腳本，讓腳本自動編譯我的程序，被自動調試，並把結果報告出來，調試成功，自動checkin源碼庫。一個命令，編譯帶着調試帶着checkin，多爽啊。只是GDB對自動化調試目前支持還不是很成熟，只能實現半自動化，真心期望着GDB的自動化調試功能的成熟。

如果各位對GDB或是別的技術問題有興趣的話，歡迎和我討論交流。本人目前主要在UNIX下做產品軟件的開發，所以，對UNIX下的軟件開發比較熟悉，當然，不單單是技術，對軟件工程實施，軟件設計，系統分析，項目管理我也略有心得。歡迎大家找我交流，（QQ是：753640，MSN 是：haoel@hotmail.com）

## 相關條

- GCC新手入門
- C/C++ IDE簡介
- 用**GDB**調試程序
- Gtk與Qt編譯環境安裝與配置
- 跟我一起寫Makefile
- C編譯初步
- C++編譯初步
- Fortran編譯初步
- C和C++混合編譯初步
- C和Fortran混合編譯初步