用**GDB**调试程序

来自**Ubuntu**中文

- 用GDB调试程序(zz)
- 作者：haoel (QQ

目录

是：753640，MSN是：haoel@hotmail.com)
- 来源：http://blog.csdn.net/haoel/archive/2003/07/02/2879.aspx

使用GDB

一般来说GDB主要调试的是C/C++的程序。要调试C/C++的程序，首先在编译时，我们必须要把调试信息加到可执行文件中。使用编译器（cc/gcc /g++）的 -g 参数可以做到这一点。如：

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

如果没有-g，你将看不见程序的函数名、变量名，所代替的全是运行时的内存地址。当你用-g把调试信息加入之后，并成功编译目标代码以后，让我们来看看如何用gdb来调试他。

启动GDB的方法有以下几种：

- gdb <program>

  program也就是你的执行文件，一般在当前目录下。

- gdb <program> core

  用gdb同时调试一个运行程序和core文件，core是程序非法执行后core dump后产生的文件。

- gdb <program> <PID>

  如果你的程序是一个服务程序，那么你可以指定这个服务程序运行时的进程ID。gdb会自动attach上去，并调试他。program应该在PATH环境变量中搜索得到。

以上三种都是进入gdb环境和加载被调试程序同时进行的。也可以先进入gdb环境，在加载被调试程序，方法如下：

```
*在终端输入：gdb
*在gdb环境中：file <program>
这两步等价于：gdb <program>
```

GDB启动时，可以加上一些GDB的启动开关，详细的开关可以用gdb -help查看。我在下面只例举一些比较常用的参数：

-symbols <file>
-s <file>
从指定文件中读取符号表。

-se file
从指定文件中读取符号表信息，并把他用在可执行文件中。

-core <file>
-c <file>
调试时core dump的core文件。

-directory <directory>
-d <directory>
加入一个源文件的搜索路径。默认搜索路径是环境变量中PATH所定义的路径。

GDB的命令概貌

启动gdb后，你就被带入gdb的调试环境中，就可以使用gdb的命令开始调试程序了，gdb的命令可以使用help命令来查看，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把之分成许多个种类。help命令只是例出gdb的命令种类，如果要看种类中的命令，可以使用help <class> 命令，如：help breakpoints，查看设置断点的所有命令。也可以直接help <command>来查看命令的帮助。

gdb中，输入命令时，可以不用打全命令，只用打命令的前几个字符就可以了，当然，命令的前几个字符应该要标志着一个唯一的命令，在Linux下，你可以敲击两次TAB键来补齐命令的全称，如果有重复的，那么gdb会把其列出来。

示例一：在进入函数func时，设置一个断点。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按两次TAB键，你会看到所有b打头的命令：

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

示例三：只记得函数的前缀，可以这样：

```
(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开头的函数全部列出来给你查看。
```

示例四：调试C++的程序时，有可以函数名一样。如：

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

你可以查看到C++中的所有的重载函数及参数。（注：M-?和“按两次TAB键”是一个意思）

要退出gdb时，只用发quit或命令简称q就行了。

GDB中运行UNIX的shell程序

在gdb环境中，你可以执行UNIX的shell的命令，使用gdb的shell命令来完成：

```
shell <command string>
```

调用UNIX的shell来执行<command string>，环境变量SHELL中定义的UNIX的shell将会被用来执行<command string>，如果SHELL没有定义，那就使用UNIX的标准shell：/bin/sh。（在Windows中使用Command.com或cmd.exe）

还有一个gdb命令是make：

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

可以在gdb中执行make命令来重新build自己的程序。这个命令等价于“shell make <make-args>”。

在**GDB**中运行程序

当以gdb <program>方式启动gdb后，gdb会在PATH路径和当前目录中搜索<program>的源文件。如要确认gdb是否读到源文件，可使用l或list命令，看看gdb是否能列出源代码。

在gdb中，运行程序使用r或是run命令。程序的运行，你有可能需要设置下面四方面的事。

1、程序运行参数。

set args 可指定运行时参数。（如：set args 10 20 30 40 50）
show args 命令可以查看设置好的运行参数。

2、运行环境。

path <dir> 可设定程序的运行路径。
show paths 查看程序的运行路径。
set environment varname [=value] 设置环境变量。如：set env USER=hchen
show environment [varname] 查看环境变量。

3、工作目录。

cd <dir> 相当于shell的cd命令。
pwd 显示当前的所在目录。

4、程序的输入输出。

info terminal 显示你程序用到的终端的模式。
使用重定向控制程序输出。如：run > outfile
tty命令可以指写输入输出的终端设备。如：tty /dev/ttyb

调试已运行的程序

两种方法：

1. 在UNIX下用ps查看正在运行的程序的PID（进程ID），然后用gdb <program> PID格式挂接正在运行的程序。
2. 先用gdb <program>关联上源代码，并进行gdb，在gdb中用attach命令来挂接进程的PID。并用detach来取消挂接的进程。

暂停/恢复程序运行

调试程序中，暂停程序运行是必须的，GDB可以方便地暂停程序的运行。你可以设置程序的在哪行停住，在什么条件下停住，在收到什么信号时停往等等。以便于你查看运行时的变量，以及运行时的流程。

当进程被gdb停住时，你可以使用info program 来查看程序的是否在运行，进程号，被暂停的原因。

在gdb中，我们可以有以下几种暂停方式：断点（BreakPoint）、观察点（Watch Point）、捕捉点（Catch Point）、信号（Signals）、线程停止

（Thread Stops）。如果要恢复程序运行，可以使用c或是 continue命令。

设置断点（**Break Points**）

我们用break命令来设置断点。下面有几点设置断点的方法：

break <function>
在进入指定函数时停住。C++中可以使用class::function或function(type,type)格式来指定函数名。

break <linenum>
在指定行号停住。

break +offset
break -offset
在当前行号的前面或后面的offset行停住。offiset为自然数。

break filename：**linenum**
在源文件filename的linenum行处停住。

break filename：**function**
在源文件filename的function函数的入口处停住。

break *address
在程序运行的内存地址处停住。

break
break命令没有参数时，表示在下一条指令处停住。

break ... if <condition>
...可以是上述的参数，condition表示条件，在条件成立时停住。比如在循环体中，可以设置break if i==100，表示当i为100时停住程序。

查看断点时，可使用info命令，如下所示：（注：n表示断点号）

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

设置观察点（**WatchPoint**）

观察点一般来观察某个表达式（变量也是一种表达式）的值是否有变化了，如果有变化，马上停住程序。我们有下面的几种方法来设置观察点：

watch <expr>
为表达式（变量）expr设置一个观察点。一表达式值有变化时，马上停住程序。

rwatch <expr>
当表达式（变量）expr被读时，停住程序。

awatch <expr>
当表达式（变量）的值被读或被写时，停住程序。

info watchpoints
列出当前所设置了的所有观察点。

设置捕捉点（**CatchPoint**）

你可设置捕捉点来捕捉程序运行时的一些事件。如：载入共享库（动态链接库）或是C++的异常。设置捕捉点的格式为：

- catch <event>

当event发生时，停住程序。event可以是下面的内容：

1. throw 一个C++抛出的异常。（throw为关键字）
2. catch 一个C++捕捉到的异常。（catch为关键字）
3. exec 调用系统调用exec时。（exec为关键字，目前此功能只在HP-UX下有用）
4. fork 调用系统调用fork时。（fork为关键字，目前此功能只在HP-UX下有用）
5. vfork 调用系统调用vfork时。（vfork为关键字，目前此功能只在HP-UX下有用）
6. load 或 load <libname> 载入共享库（动态链接库）时。（load为关键字，目前此功能只在HP-UX下有用）
7. unload 或 unload <libname> 卸载共享库（动态链接库）时。（unload为关键字，目前此功能只在HP-UX下有用）

- tcatch <event>

只设置一次捕捉点，当程序停住以後，该点被自动删除。

维护停止点

上面说了如何设置程序的停止点，GDB中的停止点也就是上述的三类。在GDB中，如果你觉得已定义好的停止点没有用了，你可以使用delete、clear、disable、enable这几个命令来进行维护。

clear
清除所有的已定义的停止点。

clear <function>
clear <filename：**function>**
清除所有设置在函数上的停止点。

clear <linenum>
clear <filename：**linenum>**
清除所有设置在指定行上的停止点。

delete [breakpoints] [range...]
删除指定的断点，breakpoints为断点号。如果不指定断点号，则表示删除所有的断点。range 表示断点号的范围（如：3-7）。其简写命令为d。

比删除更好的一种方法是disable停止点，disable了的停止点，GDB不会删除，当你还需要时，enable即可，就好像回收站一样。

disable [breakpoints] [range...]

disable所指定的停止点，breakpoints为停止点号。如果什么都不指定，表示disable所有的停止点。简写命令是dis.

enable [breakpoints] [range...]
enable所指定的停止点，breakpoints为停止点号。

enable [breakpoints] once range...
enable所指定的停止点一次，当程序停止后，该停止点马上被GDB自动disable。

enable [breakpoints] delete range...
enable所指定的停止点一次，当程序停止后，该停止点马上被GDB自动删除。

停止条件维护

前面在说到设置断点时，我们提到过可以设置一个条件，当条件成立时，程序自动停止，这是一个非常强大的功能，这里，我想专门说说这个条件的相关维护命令。一般来说，为断点设置一个条件，我们使用if关键词，后面跟其断点条件。并且，条件设置好后，我们可以用condition命令来修改断点的条件。（只有break和watch命令支持if，catch目前暂不支持if）

condition <bnum> <expression>
修改断点号为bnum的停止条件为expression。

condition <bnum>
清除断点号为bnum的停止条件。

还有一个比较特殊的维护命令ignore，你可以指定程序运行时，忽略停止条件几次。

ignore <bnum> <count>
表示忽略断点号为bnum的停止条件count次。

为停止点设定运行命令

我们可以使用GDB提供的command命令来设置停止点的运行命令。也就是说，当运行的程序在被停止住时，我们可以让其自动运行一些别的命令，这很有利行自动化调试。对基于GDB的自动化调试是一个强大的支持。

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

为断点号bnum指写一个命令列表。当程序被该断点停住时，gdb会依次运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
```

断点设置在函数foo中，断点条件是x>0，如果程序被断住后，也就是，一旦x的值在foo函数中大于0，GDB会自动打印出x的值，并继续运行程序。

如果你要清除断点上的命令序列，那么只要简单的执行一下commands命令，并直接再打个end就行了。

断点菜单

在C++中，可能会重复出现同一个名字的函数若干次（函数重载），在这种情况下，break <function>不能告诉GDB要停在哪个函数的入口。当然，你可以使用break <function(type)>也就是把函数的参数类型告诉GDB，以指定一个函数。否则的话，GDB会给你列出一个断点菜单供你选择你所需要的断点。你只要输入你菜单列表中的编号就可以了。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)
```

可见，GDB列出了所有after的重载函数，你可以选一下列表编号就行了。0表示放弃设置断点，1表示所有函数都设置断点。

恢复程序运行和单步调试

当程序被停住了，你可以用continue命令恢复程序的运行直到程序结束，或下一个断点到来。也可以使用step或next命令单步跟踪程序。

continue [ignore-count]
c [ignore-count]
fg [ignore-count]

恢复程序运行，直到程序结束，或是下一个断点到来。ignore-count表示忽略其后的断点次数。continue，c，fg三个命令都是一样的意思。

step <count>

单步跟踪，如果有函数调用，他会进入该函数。进入函数的前提是，此函数被编译有debug信息。很像VC等工具中的step in。后面可以加count也可以不加，不加表示一条条地执行，加表示执行后面的count条指令，然后再停住。

next <count>

同样单步跟踪，如果有函数调用，他不会进入该函数。很像VC等工具中的step over。后面可以加count也可以不加，不加表示一条条地执行，加表示执行后面的count条指令，然后再停住。

set step-mode
set step-mode on
打开step-mode模式，于是，在进行单步跟踪时，程序不会因为没有debug信息而不停住。这个参数很有利于查看机器码。

set step-mode off
关闭step-mode模式。

finish
运行程序，直到当前函数完成返回。并打印函数返回时的堆栈地址和返回值及参数值等信息。

until 或 **u**
当你厌倦了在一个循环体内单步跟踪时，这个命令可以运行程序直到退出循环体。

stepi 或 **si**
nexti 或 **ni**
单步跟踪一条机器指令！一条程序代码有可能由数条机器指令完成，stepi和nexti可以单步执行机器指令。与之一样有相同功能的命令是“display/i $pc”，当运行完这个命令后，单步跟踪会在打出程序代码的同时打出机器指令（也就是汇编代码）

信号（**Signals**）

信号是一种软中断，是一种处理异步事件的方法。一般来说，操作系统都支持许多信号。尤其是 UNIX，比较重要应用程序一般都会处理信号。UNIX定义了许多信号，比如SIGINT表示中断字符信号，也就是Ctrl+C的信号，SIGBUS表示硬件故障的信号；SIGCHLD表示子进程状态改变信号；SIGKILL表示终止程序运行的信号，等等。信号量编程是UNIX下非常重要的一种技术。

GDB有能力在你调试程序的时候处理任何一种信号，你可以告诉GDB需要处理哪一种信号。你可以要求GDB收到你所指定的信号时，马上停住正在运行的程序，以供你进行调试。你可以用GDB的handle命令来完成这一功能。

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

在GDB中定义一个信号处理。信号<signal>可以以SIG开头或不以SIG开头，可以用定义一个要处理信号的范围（如：SIGIO- SIGKILL，表示处理从SIGIO信号到SIGKILL的信号，其中包括SIGIO，SIGIOT，SIGKILL三个信号），也可以使用关键字 all来标明要处理所有的信号。一旦被调试的程序接收到信号，运行程序马上会被GDB停住，以供调试。其<keywords>可以是以下几种关键字的一个或多个。

nostop
当被调试的程序收到信号时，GDB不会停住程序的运行，但会打出消息告诉你收到这种信号。
stop
当被调试的程序收到信号时，GDB会停住你的程序。
print
当被调试的程序收到信号时，GDB会显示出一条信息。
noprint

当被调试的程序收到信号时，GDB不会告诉你收到信号的信息。

pass
noignore

当被调试的程序收到信号时，GDB不处理信号。这表示，GDB会把这个信号交给被调试程序处理。

nopass
ignore

当被调试的程序收到信号时，GDB不会让被调试程序来处理这个信号。

info signals
info handle

查看有哪些信号在被GDB检测中。

线程（**Thread Stops**）

如果你程序是多线程的话，你可以定义你的断点是否在所有的线程上，或是在某个特定的线程。GDB很容易帮你完成这一工作。

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

linespec指定了断点设置在的源程序的行号。threadno指定了线程的ID，注意，这个ID是GDB分配的，你可以通过“info threads”命令来查看正在运行程序中的线程信息。如果你不指定thread <threadno>则表示你的断点设在所有线程上面。你还可以为某线程指定断点条件。如：

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

当你的程序被GDB停住时，所有的运行线程都会被停住。这方便你查看运行程序的总体情况。而在你恢复程序运行时，所有的线程也会被恢复运行。那怕是主进程在被单步调试时。

查看栈信息

当程序被停住了，你需要做的第一件事就是查看程序是在哪里停住的。当你的程序调用了一个函数，函数的地址，函数参数，函数内的局部变量都会被压入“栈”（Stack）中。你可以用GDB命令来查看当前的栈中的信息。

下面是一些查看函数调用栈信息的GDB命令：

backtrace
bt

打印当前的函数调用栈的所有信息。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
```

从上可以看出函数的调用栈信息：__libc_start_main --> main() --> func()

backtrace <n>
bt <n>
n是一个正整数，表示只打印栈顶上n层的栈信息。

backtrace <-n>
bt <-n>
-n表一个负整数，表示只打印栈底下n层的栈信息。

如果你要查看某一层的信息，你需要切换当前栈，一般来说，程序停止时，最顶层的栈就是当前栈，如果你要查看栈下面层的详细信息，首先要做的是切换当前栈。

frame <n>
f <n>
n是一个从0开始的整数，是栈中的层编号。比如：frame 0，表示栈顶，frame 1，表示栈的第二层。

up <n>
表示向栈的上面移动n层，可以不打n，表示向上移动一层。

down <n>
表示向栈的下面移动n层，可以不打n，表示向下移动一层。

上面的命令，都会打印出移动到的栈层的信息。如果你不想让其打出信息。你可以使用这三个命令：

```
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。
```

查看当前栈层的信息，你可以用以下GDB命令：

frame 或 **f**
会打印出这些信息：栈的层编号，当前的函数名，函数参数值，函数所在文件及行号，函数执行到的语句。

info frame
info f
这个命令会打印出更为详细的当前栈层的信息，只不过，大多数都是运行时的内存地址。比如：函数地址，调用函数的地址，被调用函数的地址，目前的函数是由什么样的程序语言写成的、函数参数地址及值、局部变量的地址等等。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
打印出当前函数的参数名及其值。

info locals
打印出当前函数中所有局部变量及其值。

info catch
打印出当前的函数中的异常处理信息。

查看源程序

显示源代码

GDB 可以打印出所调试程序的源代码，当然，在程序编译时一定要加上-g的参数，把源程序信息编译到执行文件中。不然就看不到源程序了。当程序停下来以后，GDB会报告程序停在了那个文件的第几行上。你可以用list命令来打印程序的源代码。还是来看一看查看源代码的GDB命令吧。

list <linenum>
显示程序第linenum行的周围的源程序。
list <function>
显示函数名为function的函数的源程序。
list
显示当前行后面的源程序。
list -
显示当前行前面的源程序。

一般是打印当前行的上5行和下5行，如果显示函数是是上2行下8行，默认是10行，当然，你也可以定制显示的范围，使用下面命令可以设置一次显示源程序的行数。

set listsize <count>
设置一次显示源代码的行数。
show listsize
查看当前listsize的设置。

list命令还有下面的用法：

list <first>, <last>
显示从first行到last行之间的源代码。

list , <last>
显示从当前行到last行之间的源代码。

list +
往后显示源代码。

一般来说在list后面可以跟以下这些参数：

```
<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。
```

搜索源代码

不仅如此，GDB还提供了源代码搜索的命令：

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

其中，<regexp>就是正则表达式，也主一个字符串的匹配模式，关于正则表达式，我就不在这里讲了，还请各位查看相关资料。

指定源文件的路径

某些时候，用-g编译过后的执行程序中只是包括了源文件的名字，没有路径名。GDB提供了可以让你指定源文件的路径的命令，以便GDB进行搜索。

directory <dirname ... >
dir <dirname ... >
加一个源文件路径到当前路径的前面。如果你要指定多个路径，UNIX下你可以使用“:”，Windows下你可以使用“;”。
directory
清除所有的自定义的源文件搜索路径信息。
show directories
显示定义了的源文件搜索路径。

源代码的内存

你可以使用info line命令来查看源代码在内存中的地址。info line后面可以跟“行号”，“函数名”，“文件名:行号”，“文件名:函数名”，这个命令会打印出所指定的源码在运行时的内存地址，如：

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

还有一个命令（disassemble）你可以查看源程序的当前执行时的机器码，这个命令会把目前内存中的指令dump出来。如下面的示例表示查看函数func的汇编代码。

```
(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.
```

查看运行时数据

在你调试程序时，当程序被停住时，你可以使用print命令（简写命令为p），或是同义命令inspect来查看当前程序的运行数据。print命令的格式是：

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

<expr>是表达式，是你所调试的程序的语言的表达式（GDB可以调试多种编程语言），<f>是输出的格式，比如，如果要把表达式按16进制的格式输出，那么就是/x。

表达式

print和许多GDB的命令一样，可以接受一个表达式，GDB会根据当前的程序运行的数据来计算这个表达式，既然是表达式，那么就可以是当前程序运行中的const常量、变量、函数等内容。可惜的是GDB不能使用你在程序中所定义的宏。

表达式的语法应该是当前所调试的语言的语法，由于C/C++是一种大众型的语言，所以，本文中的例子都是关于C/C++的。（而关于用GDB调试其它语言的章节，我将在后面介绍）

在表达式中，有几种GDB所支持的操作符，它们可以用在任何一种语言中。

@

是一个和数组有关的操作符，在后面会有更详细的说明。

::

指定一个在文件或是一个函数中的变量。

{<type>} <addr>

表示一个指向内存地址<addr>的类型为type的一个对象。

程序变量

在GDB中，你可以随时查看以下三种变量的值：

1. 全局变量（所有文件可见的）
2. 静态全局变量（当前文件可见的）
3. 局部变量（当前Scope可见的）

如果你的局部变量和全局变量发生冲突（也就是重名），一般情况下是局部变量会隐藏全局变量，也就是说，如果一个全局变量和一个函数中的局部变量同名时，如果当前停止点在函数中，用print显示出的变量的值会是函数中的局部变量的值。如果此时你想查看全局变量的值时，你可以使用“::”操作符：

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

可以通过这种形式指定你所想查看的变量，是哪个文件中的或是哪个函数中的。例如，查看文件f2.c中的全局变量x的值：

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

当然，“::”操作符会和C++中的发生冲突，GDB能自动识别“::”是否C++的操作符，所以你不必担心在调试C++程序时会出现异常。

另外，需要注意的是，如果你的程序编译时开启了优化选项，那么在用GDB调试被优化过的程序时，可能会发生某些变量不能访问，或是取值错误码的情况。这个是很正常的，因为优化程序会删改你的程序，整理你程序的语句顺序，剔除一些无意义的变量等，所以在GDB调试这种程序时，运行时的指令和你所编写指令就有不一样，也就会出现你所想象不到的结果。对付这种情况时，需要在编译程序时关闭编译优化。一般来说，几乎所有的编译器都支持编译优化的开关，例如，GNU 的C/C++编译器GCC，你可以使用“-gstabs”选项来解决这个问题。关于编译器的参数，还请查看编译器的使用说明文档。

数组

有时候，你需要查看一段连续的内存空间的值。比如数组的一段，或是动态分配的数据的大小。你可以使用GDB的“@”操作符，“@”的左边是第一个内存的地址的值，“@”的右边则是你想查看内存的长度。例如，你的程序中有这样的语句：

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

于是，在GDB调试过程中，你可以以如下命令显示出这个动态数组的取值：

```
p *array@len
```

@的左边是数组的首地址的值，也就是变量array所指向的内容，右边则是数据的长度，其保存在变量len中，其输出结果，大约是下面这个样子的：

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

如果是静态数组的话，可以直接用print数组名，就可以显示数组中所有数据的内容了。

输出格式

一般来说，GDB会根据变量的类型输出变量的值。但你也可以自定义GDB的输出的格式。例如，你想输出一个整数的十六进制，或是二进制来查看这个整型变量的中的位的情况。要做到这样，你可以使用GDB的数据显示格式：

```
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
```

查看内存

你可以使用examine命令（简写是x）来查看内存地址中的值。x命令的语法如下所示：

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

n、f、u是可选的参数。

- n 是一个正整数，表示显示内存的长度，也就是说从当前地址向后显示几个地址的内容。
- f 表示显示的格式，参见上面。如果地址所指的是字符串，那么格式可以是s，如果地址是指令地址，那么格式可以是i。
- u 表示从当前地址往后请求的字节数，如果不指定的话，GDB默认是4个bytes。u参数可以用下面的字符来代替，b表示单字节，h表示双字节，w表示四字节，g表示八字节。当我们指定了字节长度后，GDB会从指定的内存地址开始，读写指定字节，并把其当作一个值取出来。

<addr>表示一个内存地址。

n/f/u三个参数可以一起使用。例如：

命令：x/3uh 0x54320 表示，从内存地址0x54320读取内容，h表示以双字节为一个单位，3表示三个单位，u表示按十进制显示。

自动显示

你可以设置一些自动显示的变量，当程序停住时，或是在你单步跟踪时，这些变量会自动显示。相关的GDB命令是display。

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

expr是一个表达式，fmt表示显示的格式，addr表示内存地址，当你用display设定好了一个或多个表达式后，只要你的程序被停下来，GDB会自动显示你所设置的这些表达式的值。

格式i和s同样被display支持，一个非常有用的命令是：

```
display/i $pc
```

$pc是GDB的环境变量，表示着指令的地址，/i则表示输出格式为机器指令码，也就是汇编。于是当程序停下后，就会出现源代码和机器指令码相对应的情形，这是一个很有意思的功能。

下面是一些和display相关的GDB命令：

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

删除自动显示，dnums意为所设置好了的自动显式的编号。如果要同时删除几个，编号可以用空格分隔，如果要删除一个范围内的编号，可以用减号表示（如：2-5）

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

disable和enalbe不删除自动显示的设置，而只是让其失效和恢复。

```
info display
```

查看display设置的自动显示的信息。GDB会打出一张表格，向你报告当前调试中设置了多少个自动显示设置，其中包括，设置的编号，表达式，是否enable。

设置显示选项

GDB中关于显示的选项比较多，这里我只例举大多数常用的选项。

set print address
set print address on
打开地址输出，当程序显示函数信息时，GDB会显出函数的参数地址。

系统默认为打开的，如：

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

set print address off

关闭函数的参数地址显示，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

查看当前地址显示选项是否打开。

set print array
set print array on

打开数组显示，打开后当数组显示时，每个元素占一行，如果不打开的话，每个元素则以逗号分隔。这个选项默认是关闭的。与之相关的两个命令如下，我就不再多说了。

set print array off
show print array
set print elements <number-of-elements>

这个选项主要是设置数组的，如果你的数组太大了，那么就可以指定一个<number-of-elements>来指定数据显示的最大长度，当到达这个长度时，GDB就不再往下显示了。如果设置为0，则表示不限制。

show print elements

查看print elements的选项信息。

set print null-stop <on/off>

如果打开了这个选项，那么当显示字符串时，遇到结束符则停止显示。这个选项默认为off。

set print pretty on

如果打开printf pretty这个选项，那么当GDB显示结构体时会比较漂亮。如：

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

set print pretty off

关闭printf pretty这个选项，GDB显示结构体时会如下显示：

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

show print pretty

查看GDB是如何显示结构体的。

set print sevenbit-strings <on/off>

设置字符显示，是否按“\nnn”的格式显示，如果打开，则字符串或字符数据按\nnn显示，如“\065”。

show print sevenbit-strings

查看字符显示开关是否打开。

set print union <on/off>

设置显示结构体时，是否显式其内的联合体数据。例如有以下数据结构：

```
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}};
```

当打开这个开关时，执行 p foo 命令后，会如下显示：

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

当关闭这个开关时，执行 p foo 命令后，会如下显示：

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

show print union

查看联合体数据的显示方式

set print object <on/off>:在C++中，如果一个对象指针指向其派生类，如果打开这个选项，GDB会自动按照虚方法调用的规则显示输出，如果关闭这个选项的话，GDB就不管虚函数表了。这个选项默认是off。

show print object

查看对象选项的设置。

set print static-members <on/off>

这个选项表示，当显示一个C++对象中的内容是，是否显示其中的静态数据成员。默认是on。

show print static-members
查看静态数据成员选项设置。

set print vtbl <on/off>
当此选项打开时，GDB将用比较规整的格式来显示虚函数表时。其默认是关闭的。

show print vtbl
查看虚函数显示格式的选项。

历史记录

当你用GDB的print查看程序运行时的数据时，你每一个print都会被GDB记录下来。GDB会以$1, $2, $3 .....这样的方式为你每一个print命令编上号。于是，你可以使用这个编号访问以前的表达式，如$1。这个功能所带来的好处是，如果你先前输入了一个比较长的表达式，如果你还想查看这个表达式的值，你可以使用历史记录来访问，省去了重复输入。

GDB环境变量

你可以在GDB的调试环境中定义自己的变量，用来保存一些调试程序中的运行数据。要定义一个GDB的变量很简单只需使用GDB的set命令。GDB的环境变量和UNIX一样，也是以$起头。如：

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

使用环境变量时，GDB会在你第一次使用时创建这个变量，而在以后的使用中，则直接对其赋值。环境变量没有类型，你可以给环境变量定义任一的类型。包括结构体和数组。

show convenience
该命令查看当前所设置的所有的环境变量。

这是一个比较强大的功能，环境变量和程序变量的交互使用，将使得程序调试更为灵活便捷。例如：

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

于是，当你就不必，print bar[0]->contents, print bar[1]->contents地输入命令了。输入这样的命令后，只用敲回车，重复执行上一条语句，环境变量会自动累加，从而完成逐个输出的功能。

查看寄存器

要查看寄存器的值，很简单，可以使用如下命令：

info registers
查看寄存器的情况。（除了浮点寄存器）

info all-registers
查看所有寄存器的情况。（包括浮点寄存器）

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

寄存器中放置了程序运行时的数据，比如程序当前运行的指令地址（ip），程序的当前堆栈地址（sp）等等。你同样可以使用print命令来访问寄存器的情况，只需要在寄存器名字前加一个$符号就可以了。如：p $eip。

改变程序的执行

一旦使用GDB挂上被调试程序，当程序运行起来后，你可以根据自己的调试思路来动态地在GDB中更改当前被调试程序的运行线路或是其变量的值，这个强大的功能能够让你更好的调试你的程序，比如，你可以在程序的一次运行中走遍程序的所有分支。

修改变量值

修改被调试程序运行时的变量值，在GDB中很容易实现，使用GDB的 print 命令即可完成。如：

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

x=4这个表达式是C/C++的语法，意为把变量x的值修改为4，如果你当前调试的语言是Pascal，那么你可以使用Pascal的语法：x:=4。

在某些时候，很有可能你的变量和GDB中的参数冲突，如：

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

因为，set width是GDB的命令，所以，出现了“Invalid syntax in expression”的设置错误，此时，你可以使用set var命令来告诉GDB，width不是你GDB的参数，而是程序的变量名，如：

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

另外，还可能有些情况，GDB并不报告这种错误，所以保险起见，在你改变程序变量取值时，最好都使用set var格式的GDB命令。

跳转执行

一般来说，被调试程序会按照程序代码的运行顺序依次执行。GDB提供了乱序执行的功能，也就是说，GDB可以修改程序的执行顺序，可以让程序执行随意跳跃。这个功能可以由GDB的jump命令来完：

jump <linespec>

指定下一条语句的运行点。<linespce>可以是文件的行号，可以是file:line格式，可以是+num这种偏移量格式。表示着下一条运行语句从哪里开始。

jump <address>

这里的<address>是代码行的内存地址。

注意，jump命令不会改变当前的程序栈中的内容，所以，当你从一个函数跳到另一个函数时，当函数运行完返回时进行弹栈操作时必然会发生错误，可能结果还是非常奇怪的，甚至于产生程序Core Dump。所以最好是同一个函数中进行跳转。

熟悉汇编的人都知道，程序运行时，有一个寄存器用于保存当前代码所在的内存地址。所以，jump命令也就是改变了这个寄存器中的值。于是，你可以使用“set $pc”来更改跳转执行的地址。如：

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

产生信号量

使用singal命令，可以产生一个信号量给被调试的程序。如：中断信号Ctrl+C。这非常方便于程序的调试，可以在程序运行的任意位置设置断点，并在该断点用GDB产生一个信号量，这种精确地在某处产生信号非常有利程序的调试。

语法是：signal <singal>，UNIX的系统信号量通常从1到15。所以<singal>取值也在这个范围。

signal命令和shell的kill命令不同，系统的kill命令发信号给被调试程序时，是由GDB截获的，而signal命令所发出一信号则是直接发给被调试程序的。

强制函数返回

如果你的调试断点在某个函数中，并还有语句没有执行完。你可以使用return命令强制函数忽略还没有执行的语句并返回。

```
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
```

在不同语言中使用**GDB**

GDB支持下列语言：C, C++, Fortran, PASCAL, Java, Chill, assembly, 和 Modula-2。一般说来，GDB会根据你所调试的程序来确定当前的调试语言，比如：发现文件名后缀为“.c”的，GDB会认为是C程序。文件名后缀为“.C, .cc, .cp, .cpp, .cxx, .c++”的，GDB会认为是C++程序。而后缀是“.f, .F”的，GDB会认为是Fortran程序，还有，后缀为如果是“.s, .S”的会认为是汇编语言。

也就是说，GDB会根据你所调试的程序的语言，来设置自己的语言环境，并让GDB的命令跟着语言环境的改变而改变。比如一些GDB命令需要用到表达式或变量时，这些表达式或变量的语法，完全是根据当前的语言环境而改变的。

例如C/C++中对指针的语法是*p，而在Modula-2中则是p^。并且，如果你当前的程序是由几种不同语言一同编译成的，那到在调试过程中，GDB也能根据不同的语言自动地切换语言环境。这种跟着语言环境而改变的功能，真是体贴开发人员的一种设计。

下面是几个相关于GDB语言环境的命令：

show language
查看当前的语言环境。如果GDB不能识为你所调试的编程语言，那么，C语言被认为是默认的环境。

info frame
查看当前函数的程序语言。

info source
查看当前文件的程序语言。

如果GDB没有检测出当前的程序语言，那么你也可以手动设置当前的程序语言。使用set language命令即可做到。

当set language命令后什么也不跟的话，你可以查看GDB所支持的语言种类：

```
(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
```

于是你可以在set language后跟上被列出来的程序语言名，来设置当前的语言环境。

后记

GDB是一个强大的命令行调试工具。大家知道命令行的强大就是在于，其可以形成执行序列，形成脚本。UNIX下的软件全是命令行的，这给程序开发提代供了极大的便利，命令行软件的优势在于，它们可以非常容易的集成在一起，使用几个简单的已有工具的命令，就可以做出一个非常强大的功能。

于是UNIX下的软件比Windows下的软件更能有机地结合，各自发挥各自的长处，组合成更为强劲的功能。而Windows下的图形软件基本上是各自为营，互相不能调用，很不利于各种软件的相互集成。在这里并不是要和Windows做个什么比较，所谓“寸有所长，尺有所短”，图形化工具还是有不如命令行的地方。（看到这句话时，希望各位千万再也不要认为我就是“鄙视图形界面”，和我抬杠了）

我是根据版本为5.1.1的GDB所写的这篇文章，所以可能有些功能已被修改，

或是又有更为强劲的功能。而且，我写得非常仓促，写得比较简略，并且，其中我已经看到有许多错别字了（我用五笔，所以错字让你看不懂），所以，我在这里对我文中的差错表示万分的歉意。

文中所罗列的GDB的功能时，我只是罗列了一些常用的GDB的命令和使用方法，其实，我这里只讲述的功能大约只占GDB所有功能的60%吧，详细的文档，还是请查看GDB的帮助和使用手册吧，或许，过段时间，如果我有空，我再写一篇GDB的高级使用。

我个人非常喜欢GDB的自动调试的功能，这个功能真的很强大，试想，我在UNIX下写个脚本，让脚本自动编译我的程序，被自动调试，并把结果报告出来，调试成功，自动checkin源码库。一个命令，编译带着调试带着checkin，多爽啊。只是GDB对自动化调试目前支持还不是很成熟，只能实现半自动化，真心期望着GDB的自动化调试功能的成熟。

如果各位对GDB或是别的技术问题有兴趣的话，欢迎和我讨论交流。本人目前主要在UNIX下做产品软件的开发，所以，对UNIX下的软件开发比较熟悉，当然，不单单是技术，对软件工程实施，软件设计，系统分析，项目管理我也略有心得。欢迎大家找我交流，（QQ是：753640，MSN是：haoel@hotmail.com）

相关词条

- GCC新手入门
- C/C++ IDE简介
- 用**GDB**调试程序
- Gtk与Qt编译环境安装与配置
- 跟我一起写Makefile
- C编译初步
- C++编译初步
- Fortran编译初步
- C和C++混合编译初步
- C和Fortran混合编译初步