GDB 使用手册

GDB 调试器使用手册（一）

START-INFO-DIR-ENTRY
* Gdb: (gdb). The GNU debugger.
END-INFO-DIR-ENTRY

使用 GDB:

本文描述 GDB,GNU 的原代码调试器。(这是 4.12 版 1994 年一月，GDB 版本 4。16)

* 索引：
GDB 简介：
****
调试器(比如象 GDB)能让你观察另一个程序在执行时的内部活动，或程序出错时发生了什么。

GDB 主要能为你做四件事(包括为了完成这些事而附加的功能)，帮助你找出程序中的错误。

* 运行你的程序，设置所有的能影响程序运行的东西。
* 保证你的程序在指定的条件下停止。
* 当你程序停止时，让你检查发生了什么。
* 改变你的程序。那样你可以试着修正某个 bug 引起的问题，然后继续查找另一个 bug.

你可以用 GDB 来调试 C 和 C++写的程序。(参考 *C 和 C++) 部分支持 Modula-2 和 chill，但现在还没有这方面的文档。

调试 Pascal 程序时，有一些功能还不能使用。

GDB 还可以用来调试 FORTRAN 程序，尽管现在还不支持表达式的输入，输出变量，或类 FORTRAN 的词法。

* GDB 是"free software"，大家都可以免费拷贝。也可以为 GDB 增加新的功能，不过可要遵守 GNU 的许可协议幺。反正我认为 GNU 还是比较不错的：-)

就这句话：

Fundamentally, the General Public License is a license which says that you have these freedoms and that you cannot take these freedoms away from anyone else.

GDB 的作者：

Richard Stallman 是 GDB 的始作俑者,另外还有许多别的 GNU 的成员。许多人为此作出了贡献。(都是老外不提也罢，但愿他们不要来找我麻烦：-))

GDB 调试器使用手册（二）

这里是 GDB 的一个例子：

原文中是使用一个叫 m4 的程序。但很遗憾我找不到这个程序的原代码，所以没有办法来按照原文来说明。不过反正是个例子，我就拿一个操作系统的进程调度原码来说明把，原代码我会附在后面。

首先这个程序叫 os.c 是一个模拟进程调度的原程序(也许是个老古董了：-))。先说明一下如何取得包括原代码符号的可执行代码。大家有心的话可以去看一下 gcc 的 man 文件(在 shell 下打 man gcc)。gcc -g <原文件.c> -o <要生成的文件名>

-g 的意思是生成带原代码调试符号的可执行文件。

-o 的意思是指定可执行文件名。

(gcc 的命令行参数有一大堆，有兴趣可以自己去看看。)

反正在 linux 下把 os.c 用以上方法编译连接以后就产生了可供 gdb 使用的可执行文件。

我用 gcc -g os.c -o os，产生的可执行文档叫 os.

然后打 gdb os,就可进入 gdb，屏幕提示：

GDB is free software and you are welcome to distribute copies
of it under certain conditions; type "show copying" to see
the conditions.

There is absolutely no warranty for GDB; type "show warranty"
for details.
GDB 4.16, Copyright 1995 Free Software Foundation, Inc...
(gdb)

(gdb)是提示符，在这提示符下可以输入命令，直到退出。(退出命令是 q/Q)
为了尽量和原文档说明的命令相符，即使在本例子中没用的命令我也将演示。
首先我们可以设置 gdb 的屏幕大小。键入：
(gdb)set width 70
就是把标准屏幕设为 70 列。
然后让我们来设置断点。设置方法很简单:break 或简单打 b 后面加行号或函数名
比如我们可以在 main 函数上设断点：
(gdb)break main
或(gdb)b main
系统提示：Breakpoint 1 at 0x8049552: file os.c, line 455.
然后我们可以运行这个程序，当程序运行到 main 函数时程序就会停止返回到 gdb 的提示符下。运行的命令是 run 或 r(gdb 中有不少 alias,可以看一下 help,在 gdb 下打 help)
run 后面可以跟参数，就是为程序指定命令行参数。
比如 r abcd，则程序就会 abcd 以作为参数。(这里要说明的是可以用 set args 来指定参数)。打入 r 或 run 后，程序就开始运行直到进入 main 的入口停止，显示：
Starting program: <路径>/os
Breakpoint 1, main () at os.c:455
455 Initial();
这里 455 Initial();是将要执行的命令或函数。
gdb 提供两种方式：1.单步进入,step into 就是跟踪到函数内啦。命令是 step 或 s
2.单步，next，就是简单的单步，不会进入函数。命令是 next 或 n
这两个命令还有别的用法以后再说。
我们用 n 命令，键入:
(gdb)n
Success forking process# 1 ,pid is 31474
Success forking process# 2 ,pid is 31475
Success forking process# 3 ,pid is 31476
Success forking process# 4 ,pid is 31477
Success forking process# 5 ,pid is 31478
Success forking process# 6 ,pid is 31479
Dispatching Algorithm : FIFO
****

PCB#	PID	Priority	PC	State
1	31474	24	0	WAITING
2	31475	19	0	WAITING
3	31476	16	0	WAITING
4	31477	23	0	WAITING
5	31478	22	0	WAITING
6	31479	20	0	WAITING

****
CPU : NO process running

```
IO :    No process
Waiting CPU!!!    31474     31475     31476     31477     31478     31479
Waiting    IO       NONE
456                  State=WAITING;
```
最后的一行就是下一句要执行的命令。我们现在在另一个函数上加断点。注意我们可以用 l/list 命令来显示原代码。这里我们键入

```
(gdb)l
451         main()
452         {
453                    int message;
454
455                  Initial();
456                  State=WAITING;
457                  printf("Use Control-C to halt \n");
458                  signal(SIGALRM,AlarmMessage);
459                  signal(SIGINT,InteruptMessage);
460                  signal(SIGUSR2,IoMessage);
(gdb) l
461                  alarm(TimeSlot);
462                  for(;;)
463                    {
464                    message=GetMessage();
465                              switch(message)
466                                 {
467                                             case INTERRUPT :              printf("Use Control-C t;

468                                                                           break;
469                                             case CHILD_IO:                WaitingIo();
470                                                                           break;
```

显示了原代码，现在在 AlarmMessage 上加断点。

```
(gdb) b AlarmMessage
Breakpoint 2 at 0x8048ee3: file os.c, line 259.
(gdb)
```

然后我们继续运行程序。

```
(gdb)c
```

c 或 continue 命令让我们继续被中断的程序。 显示：

```
Continuing.
Use Control-C to halt

Breakpoint 2, AlarmMessage () at os.c:259
259                  ClearSignal();
```

注意我们下一句语句就是 ClearSignal();

我们用 s/step 跟踪进入这个函数看看它是干什么的。

```
(gdb) s
ClearSignal () at os.c:227
227                  signal(SIGINT,SIG_IGN);
```

用 l 命令列出原代码：

```
(gdb) l
222         }
223
224
225         void ClearSignal()        /* Clear other signals */
226         {
227                     signal(SIGINT,SIG_IGN);
228                     signal(SIGALRM,SIG_IGN);
229                     signal(SIGUSR2,SIG_IGN);
230         }
231
(gdb)
```

我们可以用 s 命令继续跟踪。现在让我们来试试 bt 或 backtrace 命令。这个命令可以显示栈中的内容。

```
(gdb) bt
#0    ClearSignal () at os.c:227
#1    0x8048ee8 in AlarmMessage () at os.c:259
#2    0xbffffaec in ?? ()
#3    0x80486ae in ___crt_dummy__ ()
(gdb)
```

大家一定能看懂显示的意思。栈顶是 AlarmMessage，接下来的函数没有名字--就是没有原代码符号。这显示了函数调用的嵌套。

好了，我们跟踪了半天还没有检查过变量的值呢。检查表达式的值的命令是 p 或 print 格式是 p <表达式>

444444 让我们来找一个变量来看看。：-)

```
(gdb)l 1
```

还记得 l 的作用吗？l 或 list 显示原代码符号，l 或 list 加<行号>就显示从<行号>开始的原代码。好了找到一个让我们来看看 WaitingQueue 的内容

```
(gdb) p WaitingQueue
$1 = {1, 2, 3, 4, 5, 6, 0}
(gdb)
```

WaitingQueue 是一个数组，gdb 还支持结构的显示，

```
(gdb) p Pcb
$2 = {{Pid = 0, State = 0, Prior = 0, pc = 0}, {Pid = 31474, State = 2,
      Prior = 24, pc = 0}, {Pid = 31475, State = 2, Prior = 19, pc = 0}, {
      Pid = 31476, State = 2, Prior = 16, pc = 0}, {Pid = 31477, State = 2,
      Prior = 23, pc = 0}, {Pid = 31478, State = 2, Prior = 22, pc = 0}, {
      Pid = 31479, State = 2, Prior = 20, pc = 0}}
(gdb)
```

这里可以对照原程序看看。

原文档里是一个调试过程，不过我想这里我已经把 gdb 的常用功能介绍了一遍，基本上可以用来调试程序了。：-)

GDB 调试器使用手册（三）

运行 GDB(一些详细的说明)：

前面已经提到过如何运行 GDB 了，现在让我们来看一些更有趣的东西。你可以在运行 GDB 时通过许多命令行参数指定大量的参数和选项，通过这个你可以在一开始就设置好

程序运行的环境。
这里将要描述的命令行参数覆盖了大多数的情况，事实上在一定环境下有的并没有什么大用处。最通常的命令就是使用一个参数：
$gdb <可执行文档名>
你还可以同时为你的执行文件指定一个 core 文件：
$gdb <可执行文件名> core
你也可以为你要执行的文件指定一个进程号：
$gdb <可执行文件名> <进程号> 如：&gdb os 1234 将使 gdb 与进程 1234 相联系(attach)
除非你还有一个文件叫 1234 的。gdb 首先检查一个 core 文件。
如果你是使用一个远程终端进行远程调试的话，那如果你的终端不支持的话，你将无法使用第二个参数甚至没有 core dump。如果你觉得开头的提示信息比较碍眼的话，你可以用 gdb -silent。你还可以用命令行参数更加详细的控制 GDB 的行为。
打入 gdb -help 或-h 可以得到这方面的提示。所有的参数都被按照排列的顺序传给 gdb 除非你用了-x 参数。
当 gdb 开始运行时，它把任何一个不带选项前缀的参数都当作为一个可执行文件或 core 文件(或进程号)。就象在前面加了-se 或-c 选项。gdb 把第一个前面没有选项说明的参数看作前面加了-se 选项，而第二个(如果有的话)看作是跟着-c 选项后面的。
许多选项有缩写，用 gdb -h 可以看到。在 gdb 中你也可以任意的把选项名掐头去尾，只要保证 gdb 能判断唯一的一个参数就行。
在这里我们说明一些最常用的参数选项
-symbols <文件名>(-s <文件名>)------从<文件名>中读去符号。
-exec <文件名>(-e <文件名>)----在合适的时候执行<文件名>来做用正确的数据与 core dump 的作比较。
-se <文件名>------从<文件名>中读取符号并把它作为可执行文件。
-core <文件名>(-c <文件名>)--指定<文件名>为一个 core dump 文件。
-c <数字>----连接到进程号为<数字>，与 attach 命令相似。
-command <文件名>
-x <文件名>-----执行 gdb 命令，在<文件名>指定的文件中存放着一序列的 gdb 命令，就象一个批处理。
-directory(-d) <路径>---指定路径。把<路径>加入到搜索原文件的路径中。
-m
-mapped----
注意这个命令不是在所有的系统上都能用。如果你可以通过 mmap 系统调用来获得内存映象文件，你可以用这个命令来使 gdb 把你当前文件里的符号写入一个文件中，这个文件将存放在你的当前路径中。如果你调试的程序叫/temp/fred 那么 map 文件就叫./fred.syms 这样当你以后再调试这个程序时，gdb 会认识到这个文件的存在，从而从这个文件中读取符号，而不是从可执行文件中读取。.syms 与主机有关不能共享。
-r
-readnow---马上从符号文件中读取整个符号表，而不是使用缺省的。缺省的符号表是调入一部分符号，当需要时再读入一部分。这会使开始进入 gdb 慢一些，但可以加快以后的调试速度。
-m 和-r 一般在一起使用来建立.syms 文件
接下来再谈谈模式的设置(请听下回分解 :-))

附：在 gdb 文档里使用的调试例子我找到了在 minix 下有这个程序,叫 m4 有兴趣的
可以自己去看看模式的选择

现在我们来聊聊 gdb 运行模式的选择。我们可以用许多模式来运行 gdb，例如在“批模式”
或“安静模式”。这些模式都是在 gdb 运行时在命令行作为选项指定的。
`-nx'
`-n'
不执行任何初始化文件中的命令。(一般初始化文件叫做`.gdbinit').一般情况下在
这些文件中的命令会在所有的命令行参数都被传给 gdb 后执行。
`-quiet'
`-q'
“安静模式”。不输出介绍和版权信息。这些信息在“批模式”中也被跳过。
`-batch'
“批模式”。在“批模式”下运行。当在命令文件中的所有命令都被成功的执行后
gdb 返回状态“0”，如果在执行过程中出错，gdb 返回一个非零值。
“批模式”在把 gdb 作为一个过滤器运行时很有用。比如在一台远程计算机上下载且
执行一个程序。信息“ Program exited normally”(一般是当运行的程序正常结束
时出现)不会在这种模式中出现。
`-cd DIRECTORY'
把 DIRECTORY 作为 gdb 的工作目录，而非当前目录(一般 gdb 缺省把当前目录作为工
作目 录)。
`-fullname'
`-f'
GNU Emacs 设置这个选项，当我们在 Emacs 下，把 gdb 作为它的一个子进程来运行
时，
Emacs 告诉 gdb 按标准输出完整的文件名和行号，一个可视的栈内容。这个格式跟在
文件名的后面。行号和字符重新按列排，Emacs-to-GDB 界面使用\032 字符作为一个
显示一页原文件的信号。
`-b BPS'
为远程调试设置波特率。
`-tty DEVICE'
使用 DEVICE 来作为你程序的标准输入输出。
GDB 调试器使用手册（四）
退出 gdb
==========
`quit'
使用'quit'命令来退出 gdb,或打一个文件结束符(通常是' CTROL-D')。如果
你没有使用表达式，gdb 会正常退出，否则它会把表达式的至作为 error code
返回。
一个中断(通常是'CTROL-c)不会导致从 gdb 中退出，而是结束任何一个 gdb 的命
令，返回 gdb 的命令输入模式。一般在任何时候使用'CTROL-C'是安全的，因为
gdb 会截获它，只有当安全时，中断才会起作用。
如果你正在用 gdb 控制一个被连接的进程或设备，你可以用'detach'命令来释放
它。

Shell 命令
==========
当你偶尔要运行一些 shell 命令时，你不必退出调试过程，也不需要挂起它；你可以使用'shell'命令。
`shell COMMAND STRING'
调用标准 shell 来执行'COMMAND STRING'.环境变量'SHELL'决定了那个 shell 被运行。否则 gdb 使用'/bin/sh'.
'make'工具经常在开发环境中使用，所以你可以不用'shell'命令而直接打'make'
`make MAKE-ARGS'
用指定的命令行变量来运行'make'程序，这等于使用'shell make MAKE-ARGS'
GDB 命令
****
我们可以把一个 gdb 命令缩写成开头几个字母，如果这没有二意性你可以直接回车来运行。你还可以使用 TAB 键让 gdb 给你完成接下来的键入，或向你显示可选择的命令，如果有不止一个选择的话。
Command 语法
==========
一个 gdb 命令是一个单行的输入。长度没有限制。它一个命令开头，后面可以跟参量。比如命令'step'接受一个参量表示单步执行多少步。你也可以不用参量。有的命令不接受任何参量。

gdb 命令只要没有二意性的话就可以被缩写。另外一些缩写作为一个命令列出。在某些情况下二意也是允许的。比如's'是指定'step'的缩写，但还有命令'start'。你可以把这些缩写作为'help'命令的参量来测试它们。
空行(直接回车)表示重复上一个命令。但有些命令不能重复比如象'run'，就不会以这种方式重复，另外一些当不小心重复会产生严重后果的命令也不能用这种方法重复。
'list'和'x'命令当你简单的打回车时，会建立新的变量，而不是简单的重复上一个命令。这样你可以方便的浏览原代码和内存。
gdb 还有一种解释 RET 的方法：分割长输出。这种方法就和'more'命令相似。由于这时经常会不小心多打回车，gdb 将禁止重复当一个命令产生很长的输出时。
任何用'#'开头一直到行尾的命令行被看作是注释。主要在命令文件中使用。
GDB 调试器使用手册（五）
输入命令的技巧
==========
前面已经提到过 TAB 键的使用。使用 TAB 键能让你方便的得到所要的命令。比如在 gdb 中：
(gdb)info bre <TAB>(键入 info bre,后按 TAB 键)
gdb 能为你完成剩下的输入。它还能萎蔫提供选择的可能性。如果有两个以上可能的话，第一次按<TAB>键，gdb 会响铃提示，第二次则显示可能的选择。同样 gdb 也可以为一些子命令提供快速的访问。用法与上相同。
上例中显示
(gdb)info breakepoints
你也可以直接打回车，gdb 就将你输入的作为命令的可能的缩写。来判断执行。

如果你打入的缩写不足以判断，那么 gdb 会显示一个列表，列出可能的命令。同样的情况对于命令的参数。在显示完后 gdb 把你的输入拷贝到当前行以便让你继续输入。

如果你只想看看命令的列表或选项，你可以在命令行下打 M-?(就是按着 ESC 键同时按 SHIFT 和？键)。你可以直接在命令行下打试试。

(gdb)<M-?>

gdb 会响铃并显示所有的命令。不过这种方式好象在远程调试是不行。当有的命令使用一个字符串时，你可以用" ' "将其括起来。这种方法在调试 C++程序时特别有用。因为 C++支持函数的重载。当你要在某个有重载函数上设断点时，不得不给出函数参数以区分不同的重载函数。这时你就应该把整个函数用" ' "括起来。比如,你要在一个叫 name 的函数上设断点，而这个函数被重载了(name(int)和 name(float))。你将不得不给出参变量以区分不同的函数。使用'name(int)'和'name(float)'。这里有个技巧，你可以在函数名前加一个" ' "符号。然后打 M-?.

GDB 调试器使用手册（六-1）

得到帮助

==========

你可以使用 help 命令来得到 gdb 的在线帮助。

`help'

`h'

你可以使用 help 或 h 后面不加任何参数来得到一个 gdb 命令类的列表。

```
(gdb) help
List of classes of commands:
running -- Running the program
stack -- Examining the stack
data -- Examining data
breakpoints -- Making program stop at certain points
files -- Specifying and examining files
status -- Status inquiries
support -- Support facilities
user-defined -- User-defined commands
aliases -- Aliases of other commands
obscure -- Obscure features
 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.
(gdb)
```

`help CLASS'

使用上面列出的 help class 作为 help 或 h 的参量，你可以得到单一的命令列表。例如显示一个'status'类的列表。

```
(gdb) help status
Status inquiries.
List of commands:
show -- Generic command for showing things set
 with "set"
```

info -- Generic command for printing status
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)

`help COMMAND'
详细列出单个命令的资料。

`complete ARGS'
列出所有以 ARGS 开头的命令。例如：

complete i

results in:

info
inspect
ignore

This is intended for use by GNU Emacs.

除了使用'help'你还可以使用 gdb 的命令'info'和'show'来查询你程序的状态，每个命令可以查询一系列的状态。这些命令以恰当的方式显示所有的子命令。

`info'
此命令(可以缩写为'i')用来显示你程序的状态。比如，你可以使用 info args 列出你程序所接受的命令行参数。使用 info registers 列出寄存器的状态。或用 info breakpoint 列出在程序中设的断点。要获得详细的关于 info 的信息打 help info.

`set'
这个命令用来为你的程序设置一个运行环境(使用一个表达式)。比如你可以用 set prompt $来把 gdb 的提示符设为$.

`show'
与'info'相反，'show'命令用来显示 gdb 自身的状态。你使用'set'命令来可以改变绝大多数由'show'显示的信息。比如使用 show radix 命令来显示基数。用不带任何参变量的'set'命令你可以显示所有你可以设置的变量的值。
有三个变量是不可以用'set'命令来设置的。

`show version'
显示 gdb 的版本号。如果你发现 gdb 有 bug 的话你应该在 bug-reports 里加入 gdb 的版本号。

`show copying'
显示版权信息。

`show warranty'
显示担保信息。

在 gdb 下运行你的程序

****

当你在 gdb 下运行程序时，你必须先为 gdb 准备好带有调试信息的可执行文档。还可以在 gdb 中为你的程序设置参变量，重定向你程序的输入/输出，设置环境变量，调试一个已经执行的程序或 kill 掉一个子进程。

这里许多内容在早先的例子中都已经用到过，可以参见 gdb(二)。

目录：

* 编译:: 为调试编译带调试信息的代码
* 运行:: 运行你的程序
* 参变量:: 为你的程序设置参变量
* 运行环境:: 为你的程序设置运行时环境
* 设置工作目录:: 在 gdb 中设置程序的工作目录。
* 输入/输出:: 设定你程序的输入和输出
* 连接:: 调试一个已经运行的程序
* 结束子进程:: Kill 子进程
* 进程信息:: 附加的进程信息
* 线程:: 调试带多线程的程序
* 多进程:: 调试带多进程的程序

为调试准备带调试信息的代码

==========

为了高效的调试一个程序，你需要使用编译器来产生附带调试信息的可执行代码这些调试信息存储在目标文件中；描述了变量数据类型和函数声明，在原文件代码行和执行代码之间建立联系。

为产生调试信息，当你使用编译器时指定'-g'选项，就可以为你的程序产生带有调试信息的可执行代码。

有些 c 编译器不支持'-g'选项和'-O'选项，那你就有麻烦了，或者有别的方法产生带调试信息的可执行代码，要不就没办法了。

gcc，GNU 的 c 语言编译器支持'-g'和'-O'选项。这样你就可以产生带调试信息的且优化过的可执行代码.

当你使用 gdb 来调试一个使用'-g','-O'选项产生的程序时，千万记住编译器为了优化你的程序重新安排了你的程序。不要为运行次序与你原来设想的不同，最简单的例子就是当你定义了一个变量但从未使用过它时，gdb 中是看不到这个变量的--因为它已经被优化掉了。

所以有时你不要使用'-O'选项，如果当你不用优化时产生的程序是正确的，而优化过后变的不正确了，那么这是编译器的 bug 你可以向开发者提供 bug-reports(包括出错的例子)。

早期的 GUN C 语言编译器允许'-gg'选项，也用来产生调试信息，gdb 不再支持这种格式的调试信息，如果你的编译器支持'-gg'选项，请不要使用它。

开运行你的程序

==========

`run'

`r'

使用'run'命令在 gdb 下启动你的程序。你必须先指定你程序的名字(用 gdb 的命令行参数)或使用'file'命令，来指定文件名。如果你在一个支持多进程的环境下运行你的程序'run'命令创建一个子进程然后加载你的程序。如果环境不支持进程，则 gdb 直接调到程序的第一条命令。

一些父进程设置的参量可以决定程序的运行。gdb 提供了指定参量的途径，但你必须在程序执行前设置好他们。你也可以在运行过程中改变它们，但每次改变只有在下一次

运行中才会体现出来。这些参量可以分为四类：
---参数
　　你可以在使用'run'命令时设置，如果 shell 支持的话，你还可以使用通配符，或变量代换。在 UNIX 系统中你可以使用'shell 环境变量'来控制 shell。
---环境：
　　你的程序一般直接从 gdb 那里继承环境变量。但是你可以使用'set environment'命令来设置专门的环境变量。
---工作目录
　　你的程序还同时从 gdb 那里继承了工作目录,你可以使用'cd'命令在 gdb 中改变工作目录。
---标准输入/输出
　　你的程序一般使用与 gdb 所用的相似的设备来输入/输出。不过你可以为你的程序的输入/输出进行重定向。使用'run'或'tty'命令来设置于 gdb 所用不同的设备。
*注意：当你使用输入/输出重定向时，你将不能使用无名管道来把你所调试的程序的输出传给另一个程序。这样 gdb 会认为调试程序出错。
　当你发出'run'命令后，你的程序就开始运行。
　如果你的符号文件的时间与 gdb 上一次读入的不同，gdb 会废弃原来的符号表并重新读入。当前的断点不变。
 GDB 调试器使用手册（六-2）程序环境
==========
　“环境”包括了一系列的环境变量和它们的值。环境变量一般记录了一些常用的信息，比如你的用户名，主目录，你的终端型号和你的运行程序的搜索路径。一般你可以在 shell 下设置环境变量，然后这些变量被所有你所运行的程序所共享。在调试中，可以设置恰当的环境变量而不用退出 gdb.
`path DIRECTORY'
　　在'PATH'环境变量前加入新的内容('PATH'提供了搜索执行文件的路径)。对于 gdb 和你的程序来说你也许要设置一些专门的路径。使用':'或空格来分隔。如果 DIRECTORY 已经在路径中了，这个操作将会把它移到前面。
　　你可以使用串'$cmd'来代表当前路径，如果你用'.'的话，它代表你使用'path'命令时的路径,gdb 将在把 DIRECTORY 加入搜索路径前用'.'代替当前路径
`show paths'
　　显示当前路径变量的设置情况。
`show environment [VARNAME]'
　　显示某个环境变量的值。如果你不指明变量名，则 gdb 会显示所有的变量名和它们的内容。environment 可以被缩写成'env'

`set environment VARNAME [=] VALUE'
　　设置某个环境变量的值。不过只对你所调试的程序有效。对 gdb 本身是不起作用的。值可以是任何串。如果未指定值，则该变量值将被设为 NULL.
看一个例子：
　　　　set env USER = foo
　　告诉一个 linux 程序，当它下一次运行是用户名将是'foo'
`unset environment VARNAME'

删除某环境变量。

注意：gdb 使用'shell'环境变量所指定的 shell 来运行你的程序。

工作路径

==========

当你每次用'run'命令来运行你的程序时，你的程序将继承 gdb 的当前工作目录。而 gdb 的工作目录是从它的父进程继承而来的(一般是 shell)。但你可以自己使用'cd'命令指定工作目录。

gdb 的工作目录就是它去寻找某些文件或信息的途径。

`cd DIRECTORY'

把 gdb 的工作目录设为 DIRECTORY

`pwd'

打印输出当前目录。

你程序的输入/输出

==========

缺省时，你的程序的输入/输出和 gdb 的输入/输出使用同一个终端。gdb 在它自己和你的程序之间切换来和你交互，但这会引起混乱。

`info terminal'

显示你当前所使用的终端的类型信息。

你可以把你程序的输入/输出重定向。

例如：

run > outfile

运行你的程序并把你程序的标准输出写入文件 outfile 中。 另一个为你程序指定输入/输出的方法是使用'tty'命令，这个命令接受一个文件名作为参量把这个文件作为以后使用'run'命令的缺省命 令文件。它还重新为子进程设置控制终端。

例如：

tty /dev/ttyb

指定以后用'run'命令启动的进程使用终端'/dev/ttyb'作为程序的输入/输出，而且把这个终端设为你进程的控制终端。 一个清楚的使用'run'命令的重定向将重新设置'tty'所设置的内容，但不影响控制终端。 当你使用'tty'命令或在'run'命令中对输入/输出进行重定向时，只有你当前调试的程序的输入/输出被改变了，并不会影响到别的程序。

调试一个已经运行的程序：

==========

`attach PROCESS-ID'

这个命令把一个已经运行的进程(在 gdb 外启动)连接入 gdb,以便调试。PROCESS-ID 是进程号。(UNIX 中使用'ps'或'jobs -l'来查看进程)

'attach'一般不重复。(当你打了一个以上的回车时)

当然要使用'attach'命令的话，你的操作系统环境必须支持进程。

另外你还要有向此进程发信号的权力。

当使用'attach'命令时，你应该先使用'file'命令来指定进程所联系的程序源代码和符号表。当 gdb 接到'attach'命令后第一件事就是停止进程的运行，你可以使用所有 gdb 的命令来调试一个“连接”的进程，就象你用'run'命令在 gdb 中启动它一样。如果你要进程继续运行，使用'continue'或'c'命令就行了。

`detach'

当你结束调试后可以使用此命令来断开进程和 gdb 的连接。(解除 gdb 对它的控制)在这

个命令执行后进程将继续执行。

如果你在用'attach'连接一个进程后退出了 gdb，或使用'run'命令执行了另一个进程，这个被'attach'的进程将被 kill 掉。但缺省时，gdb 会要求你确认你是否要退出或执行一个新的进程。

GDB 调试器使用手册（七）

结束子进程

==========

`kill'

Kill 命令结束你程序在 gdb 下开的子进程

这个命令当你想要调试(检查)一个 core dump 文件时更有用。gdb 在调试过程中会忽略所有的 core dump。

在一些操作系统上，一个程序当你在上面加了断点以后就不能离开 gdb 独立运行。你可以用 kill 命令来解决这个问题。

'kill'命令当你想重新编译和连接你的程序时也很有用。因为有些系统不允许修改正变了，那么 gdb 会重新 load 新的符号。(而且尽量保持你当前的断点设置。

附加的进程信息

==========

一些操作系统提供了一个设备目录叫做'/proc'的，供检查进程映象。如果 gdb 被在这样的操作系统下运行，你可以使用命令'info proc'来查询进程的信息。('info proc'命令只在支持'procfs'的 SVR4 系统上有用。

`info proc'

显示进程的概要信息。

`info proc mappings'

报告你进程所能访问的地址范围。

`info proc times'

你进程和子进程的开始时间，用户时间(user CPU time),和系统 CPU 时间。

`info proc id'

报告有关进程 id 的信息。

`info proc status'

报告你进程的一般状态信息。如果进程停止了。这个报告还包括停止的原因和收到的信号。

`info proc all'

显示上面这些命令返回的所有信息。

对多线程程序的调试

==========

一些操作系统中，一个单独的程序可以有一个以上的线程在运行。线程和进程精确的定?nbsp;

?nbsp;

?nbsp;

?nbsp;

有自己的寄存器，运行时堆栈或许还会有私有内存。

gdb 提供了以下供调试多线程的进程的功能：

* 自动通告新线程。

* 'thread THREADNO'，一个用来在线程之间切换的命令。
* 'info threads'，一个用来查询现存线程的命令。
* 'thread apply [THREADNO] [ALL] ARGS',一个用来向线程提供命令的命令。
* 线程有关的断点设置。

注意：这些特性不是在所有 gdb 版本都能使用，归根结底要看操作系统是否支持。

如果你的 gdb 不支持这些命令，会显示出错信息：

```
(gdb) info threads
(gdb) thread 1
Thread ID 1 not known.  Use the "info threads" command to
see the IDs of currently known threads.
```

gdb 的线程级调试功能允许你观察你程序运行中所有的线程，但无论什么时候 gdb 控制，总有一个“当前”线程。调试命令对“当前”进程起作用。

一旦 gdb 发现了你程序中的一个新的线程，它会自动显示有关此线程的系统信息。比如：

[New process 35 thread 27]

不过格式和操作系统有关。

为了调试的目的，gdb 自己设置线程号。

`info threads'

显示进程中所有的线程的概要信息。gdb 按顺序显示：

1.线程号(gdb 设置)

2.目标系统的线程标识。

3.此线程的当前堆栈。

一前面打'*'的线程表示是当前线程。

例如：

```
(gdb) info threads
  3 process 35 thread 27  0x34e5 in sigpause ()
  2 process 35 thread 23  0x34e5 in sigpause ()
* 1 process 35 thread 13  main (argc=1, argv=0x7ffffff8)
      at threadtest.c:68
```

`thread THREADNO'

把线程号为 THREADNO 的线程设为当前线程。命令行参数 THREADNO 是 gdb 内定的

线程号。你可以用'info threads'命令来查看 gdb 内设置的线程号。gdb 显示该线程的系统定义的标识号和线程对应的堆栈。比如：

```
(gdb) thread 2
[Switching to process 35 thread 23]
0x34e5 in sigpause ()
```

"Switching 后的内容取决于你的操作系统对线程标识的定义。

`thread apply [THREADNO] [ALL] ARGS'

此命令让你对一个以上的线程发出相同的命令"ARGS",[THREADNO]的含义同上。如果你要向你进程中的所有的线程发出命令使用[ALL]选项。

无论 gdb 何时中断了你的程序(因为一个断点或是一个信号)，它自动选择信号或断点发生的线程为当前线程。gdb 将用一个格式为'[Switching to SYSTAG]'的消息来向你报告。

*参见：运行和停止多线程程序。

*参见：设置观察点

调试多进程的程序
==========
gdb 对调试使用'fork'系统调用产生新进程的程序没有很多支持。当一个程序开始一个新进程时，gdb 将继续对父进程进行调试，子进程将不受影响的运行。如果你在子进程可能会执行到的地方设了断点，那么子进程将收到'SIGTRAP'信号，如果子进程没有对这个信号进行处理的话那么缺省的处理就是使子进程终止。

然而，如果你要一定要调试子进程的话，这儿有一个不是很麻烦的折衷的办法。在子进程被运行起来的开头几句语句前加上一个'sleep'命令。这在调试过程中并不会引起程序中很大的麻烦(不过你要自己注意例外的情况幺：-))。然后再使用'ps'命令列出新开的子进程号，最后使用'attach'命令。这样就没有问题了。

关于这一段，本人觉得实际使用上并不全是这样。我在调试程中就试过，好象不一定能起作用，要看 gdb 的版本和你所使用的操作系统了。

停止和继续
****
调试器的基本功能就是让你能够在程序运行时在终止之前在某些条件下停止下来，然后再继续运行，这样的话你就可以检查当你的程序出错时你的程序究竟做了些什么。

在 gdb 内部，你的程序会由于各种原因而暂时停止，比如一个信号，一个断点，或是由于你用了'step'命令。在程序停止的时候你就可以检查和改变变量的值，设置或去掉断点，然后继续你程序的运行。一般当程序停下来时 gdb 都会显示一些有关程序状态的信息。比如象程序停止的原因，堆栈等等。如果你要了解更详细的信息，你可以使用'info program'命令。另外，在任何时候你输入这条命令，gdb 都会显示当前程序运行的状态信息。

`info program'

显示有关你程序状态的信息：你的程序是在运行还是停止，是什么进程，为什么停止。

断点，观察点和异常
==========
断点的作用是当你程序运行到断点时，无论它在做什么都会被停止下来。对于每个断点你都可以设置一些更高级的信息以决定断点在什么时候起作用。你可以使用'break’命令来在你的程序中设置断点，在前面的例子中我们已经提到过一些这个命令的使用方法了。你可以在行上，函数上，甚至在确切的地址上设置断点。在含有异常处理的语言(比如象 c++)中，你还可以在异常发生的地方设置断点。

在 SunOS 4.x,SVR4 和 Alpha OSF/1 的设置中，你还可以在共享库中设置断点。 观察点是一种特殊的断点。它们在你程序中某个表达式的值发生变化时起作用。你必 须使用另外一些命令来设置观察点。除了这个特性以外，你可以象对普通断点一样对观察 点进行操作--使用和普通断点操作一样的命令来对观察点使能，使不能，删除。

你可以安排当你程序被中断时显示的程序变量。

当你在程序中设置断点或观察点时 gdb 为每个断点或观察点赋一个数值.在许多对断点操作的命令中都要使用这个数值。

GDB 调试器使用手册（八）设置断点
==========
使用'break'或简写成'b'来设置断点。gdb 使用环境变量$bpnum 来记录你最新设置的断点。

你有不少方法来设置断点。

`break FUNCTION'
此命令用来在某个函数上设置断点。当你使用允许函数重载的语言比如 C++时，有可能同时在几个重载的函数上设置了断点。
`break +OFFSET'
`break -OFFSET'
在当前程序运行到的前几行或后几行设置断点。OFFSET 为行号。
`break LINENUM'
在行号为 LINENUM 的行上设置断点。程序在运行到此行之前停止。
`break FILENAME:LINENUM'
在文件名为 FILENAME 的原文件的第 LINENUM 行设置断点。
`break FILENAME:FUNCTION'
在文件名为 FILENAME 的原文件的名为 FUNCTION 的函数上设置断点。当你的多个文件中可能含有相同的函数名时必须给出文件名。
`break *ADDRESS'
在地址 ADDRESS 上设置断点，这个命令允许你在没有调试信息的程序中设置断点。
`break'
当'break'命令不包含任何参数时，'break'命令在当前执行到的程序运行栈中的下一条指令上设置一个断点。除了栈底以外，这个命令使程序在一旦从当前函数返回时停止。相似的命令是'finish'，但'finish'并不设置断点。这一点在循环语句中很有用。
gdb 在恢复执行时，至少执行一条指令。
`break ... if COND'
这个命令设置一个条件断点，条件由 COND 指定；在 gdb 每次执行到此
断点时 COND 都被计算当 COND 的值为非零时，程序在断点处停止。这意味着
COND 的值为真时程序停止。...可以为下面所说的一些参量。
`tbreak ARGS'
设置断点为只有效一次。ARGS 的使用同'break'中的参量的使用。
`hbreak ARGS'
设置一个由硬件支持的断点。ARGS 同'break'命令，设置方法也和'break'相同。但这种断点需要由硬件支持，所以不是所有的系统上这个命令都有效。这个命令的主要目的是用于对 EPROM/ROM 程序的调试。因为这条命令可以在不改变代码的情况下设置断点。这可以同 SPARCLite DSU 一起使用。当程序访问某些变量和代码时，DSU 将设置“陷井”。注意：你只能一次使用一个断点，在新设置断点时，先删除原断点。
`thbreak ARGS'
设置只有一次作用的硬件支持断点。ARGS 用法同'hbreak'命令。这个命令和'tbreak'命令相似，它所设置的断点只起一次作用，然后就被自动的删除。这个命令所设置的断点需要有硬件支持。

`rbreak REGEX'
在所有满足表达式 REGEX 的函数上设置断点。这个命令在所有相匹配的函数上设置无条件断点，当这个命令完成时显示所有被设置的断点信息。这个命令设置的断点和'break'命令设置的没有什么不同。这样你可以象操作一般的断点一样对这个命令设置的断点进行删除，使能，使不能等操作。当调试 C++程序时这个命令在重载函数上设置断点时非常有用。
`info breakpoints [N]'
`info break [N]'

`info watchpoints [N]'
显示所有的断点和观察点的设置表，有下列一些列
Breakpoint Numbers----断点号
Type----断点类型(断点或是观察点)
Disposition---显示断点的状态。
Enabled or Disabled---使能或不使能。'y'表示使能，'n'表示不使能。
Address----地址，断点在你程序中的地址(内存地址)
What---地址，断点在你程序中的行号。
如果断点是条件断点，此命令还显示断点所需要的条件。
带参数 N 的'info break'命令只显示由 N 指定的断点的信息。
此命令还显示断点的运行信息(被执行过几次),这个功能在使用'ignore' 命令时很有用。你可以'ignore'一个断点许多次。使用这个命令可以查看断点被执行了多少次。这样可以更快的找到错误。
gdb 允许你在一个地方设置多个断点。但设置相同的断点无疑是弱智的。不过你可以使用条件断点，这样就非常有用。
gdb 有时会自动在你的程序中加入断点。这主要是 gdb 自己的需要。比如为了正确的处理 C 语言中的'longjmp'。这些内部断点都是负值，以'-1'开始。'info breakpoints'不会显示它们。
不过你可以使用命令’maint info breakpoints'来查看这些断点。
`maint info breakpoints'
使用格式和'info breakpoints'相同，显示所有的断点，无论是你设置的还是 gdb 自动设置的。
以下列的含义：
`breakpoint'
断点，普通断点。
`watchpoint'
普通观察点。
`longjmp'
内部断点，用于处理'longjmp'调用。
`longjmp resume'
内部断点，设置在'longjmp'调用的目标上。
`until'
'until'命令所使用的内部断点。
`finish'
'finish'命令所使用的内部断点。
设置观察点
==========
你可以使用观察点来停止一个程序，当某个表达式的值改变时，观察点会将程序停止。而不需要先指定在某个地方设置一个断点。
由于观察点的这个特性，使观察点的使用时开销比较大，但在捕捉错误时非常有用。特别是你不知道你的程序什么地方出了问题时。
`watch EXPR'
这个命令使用 EXPR 作为表达式设置一个观察点。GDB 将把表达式加入到程序中并监视程序的运行，当表达式的值被改变时 GDB 就使程序停止。这个也可以被用在 SPARClite

DSU 提供的新的自陷工具中。当程序存取某个地址或某条指令时(这个地址在调试寄存器中指定),DSU 将产生自陷。对于数据地址 DSU 支持'watch'命令,然而硬件断点寄存器只能存储两个断点地址，而且断点的类型必须相同。就是两个'rwatch'型断点，或是两个'awatch'型断点。

`rwatch EXPR'

设置一个观察点，当 EXPR 被程序读时，程序被暂停。

`awatch EXPR'

设置一个观察点,当 EXPR 被读出然后被写入时程序被暂停。这个命令和'awatch'命令合用。

`info watchpoints'

显示所设置的观察点的列表，和'info break'命令相似。

*注意：*在多线程的程序中，观察点的作用很有限，GDB 只能观察在一个线程中的表达式的值如果你确信表达式只被当前线程所存取，那么使用观察点才有效。GDB 不能注意一个非当前线程对表达式值的改变。

断点和异常

==========

在一些语言中比如象 GNU C++，实现了异常处理。你可以使用 GDB 来检查异常发生的原因。而且 GDB 还可以列出在某个点上异常处理的所有过程。

`catch EXCEPTIONS'

你可以使用这个命令来在一个被激活的异常处理句柄中设置断点。EXCEPTIONS 是一个你要抓住的异常。

你一样可以使用'info catch'命令来列出活跃的异常处理句柄。

现在 GDB 中对于异常处理由以下情况不能处理。

* 如果你使用一个交互的函数，当函数运行结束时，GDB 将象普通情况一样把控制返回给你。如果在调用中发生了异常，这个函数将继续运行直到遇到一个断点，一个信号或是退出运行。

* 你不能手工产生一个异常(即异常只能由程序运行中产生)

* 你不能手工设置一个异常处理句柄。

有时'catch'命令不一定是调试异常处理的最好的方法。如果你需要知道异常产生的确切位置，最好在异常处理句柄被调用以前设置一个断点，这样你可以检查栈的内容。如果你在一个异常处理句柄上设置断点，那么你就不容易知道异常发生的位置和原因。 要仅仅只在异常处理句柄被唤醒之前设置断点，你必须了解一些语言的实现细节。比如在 GNU C++中异常被一个叫'__raise_exception'的库函数所调用。这个函数的原型是：

```
/* ADDR is where the exception identifier is stored.
    ID is the exception identifier.  */
void __raise_exception (void **ADDR, void *ID);
```

要使 GDB 在栈展开之前抓住所有的句柄，你可以在函数'__raise_exception'上设置断点。

对于一个条件断点，由于它取决于 ID 的值，你可以在你程序中设置断点，当某个特别的异常被唤醒。当有一系列异常被唤醒时，你可以使用多重条件断点来停止你的程序。

GDB 调试器使用手册（九）

断点条件

==========

最简单的断点就是当你的程序每次执行到的时候就简单将程序挂起。你也可以为断点

设置“条件”。条件只是你所使用的编程语言的一个布尔表达式，带有条件表达式的断点 在每次执行时判断计算表达式的值，当表达式值为真时才挂起程序。

这是使用“断言”的一中形式，在这种形式中你只有在断言为真时才挂起程序。如果 在 C 语言中你要使断言为假时挂起程序则使用：“!表达式”。

条件表达式对观察点也同样有效，但你并不需要它，因为观察点本身就计算一个表达式?nbsp;
?nbsp;
但它也许会简单一些。比如只在一个变量名上设置观察点然后设置一个条件来测试新的赋值。

断点条件可能有副作用(side effects)会影响程序的运行。这一点有时也是很有用的比如来激活一个显示程序完成情况的的函数，或使用你自己的打印函数来格式化特殊的数据结构。当在同一位置没有另一个断点设置时，结果是可预见的。(在 gdb 中如果在同一个地方使用了一个断点和一个条件断点则普通断点可能先被激活。)在条件断点的应用上有很多技巧。

断点条件可以在设置断点的同时被设置。使用'if'命令作为'break'命令的参数。断点条件也可以在任何时候使用'condition'命令来设置。'watch'命令不能以'if'作为参数
所以使用'condition'命令是在观察点上设置条件的唯一方法。

`condition BNUM EXPRESSION'

把'EXPRESSIN'作为断点条件。断点用'BNUM'来指定。在你为 BNUM 号断点设置了条件后，只有在条件为真时程序才被暂停。当你使用'condition'命令 GDB 马上同步的检查'EXPRESSION'的值判断表达式中的符号在断点处是否有效，但 GDB 并不真正计算表达式的值。

`condition BNUM'

删除在'BNUM'号断点处的条件。使之成为一个普通断点。

一个条件断点的特殊例子是时一个程序在执行了某句语句若干次后停止。由于这个功能非常常用，你可以使用一个命令来直接设置它那就是'ignore count'。每个断点都有'ignore count'，缺省是零。如果'ignore count'是正的那么你的程序在运行过断点处'count'次后被暂停。

`ignore BNUM COUNT'

设置第 BNUM 号断点的'ignore count'为'COUNT'。

如果要让断点在下次执行到时就暂停程序，那么把'COUNT'设为 0.

当你使用'continue'命令来继续你程序的执行时，你可以直接把'ignore count' 作为'continue'的参数使用。你只要直接在'continue'命令后直接跟要"ignore"的次数就行。

如果一个断点同时有一个 ignore count 和一个条件时，条件不被检查。只有当'ignore count'为零时 GDB 才开始检查条件的真假。

另外你可以用'condition'命令来获得与用‘ignore count'同样效果的断点。用法是用类似于'$foo--<=0'的参量作为'condition'命令的参数(使用一个不停减量的变量作为条件表达式的成员)。

断点命令列表
==========

你可以为任一个断点或观察点指定一系列命令，当你程序执行到断点时，GDB 自动执行这些命令。例如：你可以打印一些表达式的值，或使能其他的断点。

`commands [BNUM]'
`... COMMAND-LIST ...'
`end'

为断点号为 BNUM 的断点设置一个命令列表。这些命令在'...COMMAND-LIST...'中列出使用'end'命令来表示列表的结束。

要删除断点上设置的命令序列，你只需在'command'命令后直接跟'end'命令就可以了。

当不指定 BNUM 时，GDB 缺省为最近遇到的断点或是观察点设置命令列表。

使用回车来表示重复使用命令的特性在'...command list...'中不能使用。

你可以使用命令列表中的命令来再次使你的程序进入运行状态。简单的在命令列表中使用'continue'命令，或'step'命令。

在使程序恢复执行的命令后的命令都被忽略。这是因为一旦你的程序重新运行就可能遇到新的命令列表，那么就应该执行新的命令。防止了二义。

如果你在命令列表中使用了'silent'命令，那么你程序在断点处停止的信息将不被显示。这对于用一个断点然后显示一些信息，接着再继续执行很有用。但'silent'命令

只有在命令列表的开头有效。

命令'echo','output'和'printf'允许你精确的控制显示信息，这些命令在"silent"
断点中很有用。

例如：这个例子演示了使用断点命令列表来打印'x'的值.

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

断点命令列表的一个应用是在遇到一个 buf 之后改正数据然后继续调试的过程。
使用命令来修改含有错误值的变量，然后使用'continue'命令继续程序的运行。

使用'silent'命令屏蔽输出：

```
break 403
commands
silent
set x = y + 4
cont
end
```

断点菜单
==========

一些编程语言(比如象 C++)允许一个函数名被多次使用(重载)，以方便应用的使用。当一个函数名被重载时，'break FUNCITON'命令向 GDB 提供的信息不够 GDB 了解你要设置
断点的确切位置。如果你了解到这个问题，你可以使用'break FUNCITONS(TYPES)'命令来指定断点的确切位置。否则 GDB 会提供一个函数的选择的菜单供你选择。使用提示符'>'来等待你的输入。开始的两个选择一般是'[0] cancel'和'[1] all'输入 1 则在所有
同名函数上加入断点。输入 0 则退出选择。

下例为企图在重载的函数符号'String::after'上设置断点。

```
(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)
```