

GDB 教程

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，这给程序开发提供了极大的便利，命令行软件的优势在于，他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 下的软件更能有机的结合，各自发挥各自的长处，组合成更为强筋的功能。而 windows 下的图形软件基本上各自为营，互相不能条用，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。

用 GDB 调试程序

GDB 概述

GDB 是 GNU 开源组织发布的一个强大的 UNIX 下调试程序工具。或许各位比较喜欢那种图形界面方式的，像 VC，BCB 等 IDE 的调试，但如果你是在 UNIX 平台下作软件，你会发现 GDB 这个调试工具有比 VC，BCB 的图形化调试器更强大的功能。所谓“寸有所长，尺有所短”就是这个道理。

一般来说，GDB 主要帮助你完成下面四个方面的功能：

- 1、启动你的程序，可以按照你自定义的要求随心所欲的运行程序。
- 2、可以让调试程序在你所指定的位置的断点处停止。
- 3、当程序停止时，可以检查此时你的程序中所发生的事情。
- 4、动态的改变你程序的执行环境。

从上面看来，GDB 和一般的调试工具没有什么两样，基本上也是完成这些功能，不过在细节上，你会发现 GDB 这个调试工具的强大，大家可能习惯图形化的调试工具，但有时候，命令行的调试工具却有着图形化工具所不能完成的功能。

让我们来看一个例子：

源程序：test.c

```
1 #include <stdio.h>
2
3 int func(int n)
4 {
5     int sum=0,i;
6     for(i=0; i<n; i++)
7     {
8         sum+=i;
9     }
10    return sum;
11 }
12
13
```

```

14 main()
15 {
16     int i;
17     long result = 0;
18     for(i=1; i<=100; i++)
19     {
20         result += i;
21     }
22
23     printf("result[1-100] = %d \n", result );
24     printf("result[1-250] = %d \n", func(250) );
25 }

```

编译生成执行文件:

```
<byb>gcc -g test.c -o test
```

使用 GDB 调试:

```
<byb>gdb test      —————启动 GDB
```

GNU gdb 5.1.1

Copyright 2002 Free Software Foundation, Inc.

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or 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.

This GDB was configured as "i386-suse-linux"...

(gdb) list —————从第一行列出源码

```

1      #include <stdio.h>
2
3      int func(int n)
4      {
5          int sum=0,i;
6          for(i=0; 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     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 0x8048496: file tst.c, line 16.
(gdb) break func ——设置断点，在函数 func()入口处。
Breakpoint 2 at 0x8048456: file tst.c, line 5.
(gdb) info break ——查看断点信息。
Num Type      Disp Enb Address  What
1  breakpoint keep y  0x08048496 in main at tst.c:16
2  breakpoint keep y  0x08048456 in func at tst.c:5
(gdb) run      ——运行程序
Starting program: /home/hchen/test/tst
Breakpoint 1, main () at tst.c:17 ——在断点处停住。
17      long result = 0;
(gdb) next      ——单条语句执行。
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) continue ——继续运行程序
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) print i ——打印变量 i 的值。
$1 = 134513808
(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 ——p 是 print 的缩写

```

```

$4 = 3
(gdb) bt      ——查看函数堆栈
#0 func (n=250) at tst.c:5
#1 0x080484e4 in main () at tst.c:24
#2 0x400409ed in __libc_start_main () from /lib/libc.so.6
(gdb) finish  ——推出函数
Run till exit from #0 func (n=250) at tst.c:5
0x080484e4 in main () at tst.c:24
24      printf("result[1-250] = %d \n", func(250) );
Value returned is $6 = 31375
(gdb) continue
Continuing.
result[1-250] = 31375 ——程序输出。
Program exited with code 027. ——程序退出，调试结束。
(gdb) quit    ——退出 gdb

```

好了，有了以上的感性认识，还是让我们来系统的认识一下 gdb 吧。

使用 GDB

一般来说 GDB 主要调试的是 C/C++ 程序。要调试 C/C++ 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：

```
$ cc -g hello.c -o hello
```

```
$ g++ -g hello.cpp -o hello
```

如果没有 -g，你将看不见程序的函数名，变量名，所代替的全是运行的内存地址。当你用 -g 把调试信息假如之后，并成功编译目标代码以后，让我们来看看如果用 GDB 调试它。

启动 GDB 的方法有以下几种：

1、gdb <program>

program 也就是你的执行文件，一般在当前目录下。

2、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。

3、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，并调试它。program 应该在 PATH 环境变量中搜索到。

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 来查看。下面只列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 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，就进入了 gdb 的调试环境，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) help

List of classes of commands:

aliases -- Aliases of other commands

breakpoints -- Making program stop at certain points

data -- Examining data

files -- Specifying and examining files

internals -- Maintenance commands

obscure -- Obscure features

running -- Running the program

stack -- Examining the stack

status -- Status inquiries

support -- Support facilities

tracepoints -- Tracing of program execution without stopping the program

user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.

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

Command name abbreviations are allowed if unambiguous.

(gdb)

gdb 的命令很多，gdb 把之分成很多种类。help 命令只是列出了 gdb 的命令种类，如果要看种类中的命令，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，命令的前几个字符要标志着一个唯一的命令，在 linux 下，可以敲击两次 TAB 键来补齐命令的全称，如果有重复的，gdb 会把其列出来。

示例一：在进入函数 func 时，设置一个断点。可以敲击 break func，或者直接就是 b func

(gdb) b func

Breakpoint 1 at 0x8048458: file hello.c, line 10.

示例二：敲入 b 按两次 TAB 键，你会看到所有 b 开头的命令：

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

示例三：只记得函数的前缀，可以这样：

```
(gdb) b make_<按 TAB 键>
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++ 程序，可以函数名一样。如：

```
(gdb) b 'bubble(<按两次 TAB 键>
bubble(double,double)    bubble(int,int)
```

```
(gdb) b 'bubble
你可以查看到 C++ 中所有的重载函数以及参数
```

要退出 GDB，只要 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 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

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“make make <make -args>”

在 GDB 中运行程序

1、 当以`

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 显示当前的工作目录。

4、程序的输入输出

`info terminal` 显示程序用到的终端的模式

使用重定向空值程序输出。如 `run>outfile`

`tty` 命令可以指定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` 来取消挂接的程序。

暂停/恢复程序运行

调试程序中，暂停进程运行时必须的，GDB 可以方便的暂停程序的运行。你可以设置程序在哪停住，在什么条件下停住，在收到什么信号时停住等等。你便于你查看运行的变量，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（watchpoint）、捕捉点（catchpoint）、信号（signals）、线程停止（thread stops）。如果要恢复程序运行，可以使用 `c` 或者 `continue` 命令。

一、设置断点（breakpoint）

我们用 `break` 命令设置断点。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者后面的 `offset` 行停住。Offset 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，可使用命令 **info** 命令，如下所示：（注：**n** 表示断点号）

info breakpoints[n]

info break[n]

二、 设置捕捉点（**catchpoint**）

我们可以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++ 的一场，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，当程序停住后，断点被自动删除。

三、 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** 这几个命令来进程维护。

clear

清楚所有已定义的停止点。

clear<function>

clear <filename:function>

清楚所有设置在函数上的停止点。

clear<linenum>

clear <filename:linenum>

清楚所有设定在指定行上的停止。

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 2-7），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。

```
enable[breakpoints][range...]
```

enable 所指定的停止点，breakpoints 为停止号。

```
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。

四、 停止条件维护

前面在说到设置断点时，我们提到过可设置一个条件，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，专门说说这个条件相关维护命令。一般来说，为断点设置一个条件，我们使用 if 关键字，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，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。

五、 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，我们可以让你自动运行一些别的命令，这很有利于自动化调试。对给予 GDB 的自动化调试是一个很强大的支持。

```
command[bnum]
```

```
...command -list...
```

```
end
```

为断点号 bnum 指定一个命令列表。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，如果程序符合条件被停住后，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。

如果要清楚断点上的命令序列，那么只要简单的执行以下 `commands` 命令，并直接再打个 `end` 就行了。

六、 断点菜单

在 C++ 中，可能会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，你也可以使用 `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，你可以选以下列表编号就行了。0 表示放弃设置断点，1 表示所有函数都设置断点。

七、 恢复程序运行和单步调试

当程序被停住后，你可以用 `continue` 命令恢复程序的运行直到程序结束，或下一个断点的到来。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，次函数被编译有 `debug` 信息。像 VC 等工具中的 `step in`。后面可以加 `count`，也可以不加，不加表示一条一条地执行，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，他不会进入函数。想 VC 等工具中的 **step over**。后面可以加 **count**，也可以不加。不加表示一条条的执行，加表示执行后面的 **count** 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，程序不会因为缺少 **debug** 信息而不停住。这个参数很有利于查看机器码。

set step-mode off

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信息。

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知道退出循环体。

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式 “**display/l \$pc**”，当运行完这个命令后，单步跟踪会在打出代码的同时打出机器指令（也就是汇编代码）

八、 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，比较重要的应用程序一般都会处理信号。UNIX 定义了很多信号，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。信号量编程时 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，我们可以告诉 GDB 需要处理哪一种信号。我们可以要求 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** 来表明要处理的所有信号。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。

九、 线程 (**thread stops**)

如果你的程序是多线程的话，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 指定了断点是在源程序的行号。**Threadno** 指定了线程 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的线程上面。还可以为某个线程指定断点条件，如：

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

当程序被 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。哪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了。当你的程序调用了函数，函数的地址，函数的参数，函数内的局部变量都会被压入“栈 (stack)”中。你可以使用 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 是一个负整数，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，你需要切换当前的栈，一般来说，程序停止时，最顶层的栈就是当前的栈，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。如果你不想让其打印出信息。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，函数所在的文件及行号，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。

比如函数的地址，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

打印出当前函数中的异常处理信息。

查看源代码

一、 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** 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，如果显示函数则是上 2 行和下 8 行，默认是 10 行，当然，也可以定制显示的范围，使用下面的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的参数：

<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>那个文件的那一行。

<function>函数名

<filename:function>那个文件中的那个函数。

<*address>程序运行时的语句在内存中的地址。

二、 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，也可以是一个字符串的匹配模式，关于正则表达式，我就不在这里说了，还请各位查看相关资料。

三、 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称，没有路径。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

显示定义的源文件搜索路径。

四、 源代码的内存

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代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 <func+13>.

还有一个命令（`disassemble`）你可以查看源程序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    $0x0,0xffffffff(%ebp)
0x804845d <func+13>:  movl    $0x1,0xffffffff8(%ebp)
0x8048464 <func+20>:  mov     0xffffffff8(%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     0xffffffff8(%ebp),%eax
0x8048473 <func+35>:  add     %eax,0xffffffffc(%ebp)
0x8048476 <func+38>:  incl    0xffffffff8(%ebp)
0x8048479 <func+41>:  jmp     0x8048464 <func+20>
0x804847b <func+43>:  nop
0x804847c <func+44>:  lea     0x0(%esi,1),%esi
0x8048480 <func+48>:  mov     0xffffffffc(%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，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。

一、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 调试其他语言的章节，将在后面介绍）

在表达式中，有几种 GDB 所支持的操作符，他们可以用在任何一种语言中。

@

是一个数组有关的操作符，在后面会有更详细的说明。

::

指定一个在文件中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。

二、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（也就是重命名），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重的局部变量同名时，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局部变量的值。如果此时你想查看局部变量的值，可以使用“::”操作符。

file:variable

function::variable

可以通过这种形式指定你想查看的变量，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“::”是否是 C++ 的操作符，所以不用担心在调试 C++ 程序时会出现异常。

另外，需要主要的是，如果你的程序编译时开启了优化选项，那么在 GDB 调试未优化过的程序时，可能会发生某些变量不能访问，或是取值错误的情况。这个是正常的，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的指令就有不一样的，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。

三、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的内存的长度。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，你可以用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：

```
(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。

四、 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形变量的位的情况。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
```

五、 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 表示显示格式，参加上面。如果地址所指的是字符串，那么格式是 **s**，如果地址是指令地址，那么格式也可以是 **i**。

u 表示从当前的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** 表示按十六进制显示。

六、 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。于是当程序停下来后，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示的编号。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 **enable** 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 会打印出一张表格，向你报告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**。

七、 设置显示选项

GDB 中关于显示的选项比较多，这里只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 会显示出函数的参数地址。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，就不多说了。

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 时如何显示结构体的。

set print sevenbit-strings<no/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，如果一个对象指针所指向其派生类，如果打开这个选项，GDB 会自动按照虚方法条用的规则显示输出，如果关闭这两个选项的话，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++ 对象中的内容时，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。其默认是关闭的。

show print vtbl

查看虚函数显示格式的选项。

八、历史记录

当你的用 GDB 的 print 查看程序运行的数据时，你每一个 print 都会被 GDB 记录一下。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。

九、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。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。

这时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。

十、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...>

查看所有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 `$` 符号就可以来看。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的在 GDB 中更改当前被调试程序的运行线路或是其变量的值，这个强大的功能能让你更好的调试你的程序，比如，你可以在你的程序的一次运行中走遍程序所有的分支。

一、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，可以使用 `set var` 命令来告诉 GDB，`width` 不是 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。

二、 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，可以让程序随意跳跃。这个功能可以由 GDB 得 `jump` 命令来完成：

`jump<linespec>`

指定下一条语句的运行点。`<linespec>` 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。表示下一条运行语句从哪里开始。

`jump<laddress>`

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，甚至产生 `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
```

三、 产生信号量

使用 `signal` 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，这种精确的在某处产生信号非常有利于程序的调试。

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

`single` 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 `single` 命令所发出的信号则是直接发给被调试程序的。

四、 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作为函数的返回值。

五、 强制调用函数

`call<expr>`

表达式中可以是函数，以此达到强制调用函数的目的。并显示函数的返回值，如果函数返回值是 `void`，那么就不显示。

另一个相似的命令也可以完成这一功能——`print`，`print`后面可以跟表达式，所以也可以用他来调用函数，`print`和`call`不同的是，如果函数返回 `void`，`call` 则不显示，而 `print` 显示函数返回值，并把该值存入历史数据中。

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 会根据你所调用的程序来确定当时的调试语言，比如：发现文件后缀名为“.c”的，GDB 会认为是 C 程序。文件后缀名为“.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，后缀名如果是“.s, .S”，GDB 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。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，在调试过程中，GDB 也能根据不同的语言自动的切换语言环境。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 不能识别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，那么我们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。