|  |
| --- |
|  |
| 使用GDB进行调试 |
| GNU代码级调试工具 |
| 第十版，GDB版本7.6.50.20130513 |
| **lnwdl** |
| **2013/5/14** |

|  |
| --- |
| Richard Stallman, Roland Pesch, Stan Shebs, et al. |

# GDB概述

调试工具，比如GDB用来让你了解当一个程序执行的时候程序内部的信息，或者当一个程序崩溃的时候它在干什么。

GDB可以完成四类事情（加上支持这些的其它一些东西）来帮助你在运行过程中查找bugs：

* 启动程序，指定任何可以影响它的行为的事情。
* 使程序在指定的条件下暂停。
* 当程序暂停的时候，检查发生了什么。
* 改变程序的内容，这样可以暂时修正一个bug，然后继续调试。

GDB可以调试使用C和C++写的程序。详细内容，参考15.4章。详细内容，参考15.4.1。

部分支持D语言。详细信息，参考15.4.2。

部分支持OpenCL C。关于OpenCL C的详细信息，参考15.4.5。

暂不支持使用sets，subranges，文件变量或者内嵌函数调试Pascal程序。GDB不支持Pascal语法的外部表达式，变量打印，和一些相似的特性。

GDB可以用来调试Fortran程序，尽管可能需要使用尾部的下划线来引用一些变量。

GDB可以用来调试Objective-C程序，无论是使用Apple/NeXT或者GNU Objective-C。

## 开源软件

略

## 开源软件需要开源的文档

略

## GDB的贡献者

略

# GDB会话的一个例子

你可以在空闲的时候通过本手册详细了解GDB。但是，少数的一些命令足够可以使用GDB了。本章介绍这些命令。

在示例会话中，我们使用粗体字强调输入，将它从环境中突出出来：**输入**。

我们使用GNU m4（一个通用的宏处理器）程序来展示下面的调试过程：有时，但我们将引用的字符串从默认的进行改变时，指令会在另一个终止符之内截取宏定义。在下面简短的m4会话中，我们定义个名称为foo的宏，展开为0000；之后使用m4内置的defn命令来定义bar宏。然而，当我们将起始的引用字符串改变为<QUOTE>，结束字符串改变为<UNQUOTE>时，使用同样的方式定义新的宏会失败：

$ **cd gnu/m4**

$ **./m4**

**define(foo, 0000)**

foo

0000

**define(bar, defn(‘foo’))**

bar

0000

**changequote(<QUOTE>, <UNQUOTE>)**

**define(baz, defn(<QUOTE>foo<UNQUOTE>))**

baz

**Ctrl-d**

m4: End of input: 0: fatal error : EOF in string

让我们使用GDB来观察一下发生了什么。

$ **gdb m4**

(gdb)

GDB只是读取如何去查找剩余符号的一些信息；所以，第一个提示符很快就出现。现在我们告诉GDB使用比较狭窄的显示宽度，使显示信息能够在本手册中合理一些。

(gdb) **set width 70**

我们需要查看m4内置的changequote是如何工作的，通过对源码的查看，我们知道相关的子函数是m4\_changequote，所以我们在这个函数上面打一个断点。

(gdb) break m4\_changequote

Breakpoint 1 at 0x62f4: file builtin.c, line 879

使用run命令，我们使m4在GDB的控制下运行；只要还没有运行到m4\_changequote子函数，程序像往常一样运行：

(gdb) **run**

Starting program: /work/Editorial/gdb/gnu/m4/m4

**define(foo, 0000)**

foo

0000

为了触发断点，我们调用changequote。GDB将m4暂停，显示此时的上下文。

**changequote(<QUOTE>, <UNQUOTE>)**

Breakpoint 1, m4\_changequote(argc=3, argv=0x33c70)

at builtin.c: 879

879 if (bad\_argc(TOKEN\_DATA\_TEXT(argv[0], argc, 1, 3))

现在我们使用命令n（next）来向前执行一步，达到此函数的下一行。

(gdb) **n**

882 set\_quotes((argc >= 2) ? TOKEN\_DATA\_TEXT(argv[1]) \

: nil,

set\_quotes函数是一个字程序。我们可以使用命令s（step）而不是next来进入此函数。Step会进入下一行中将要执行的函数，所以它进入函数set\_quotes。

(gdb) **s**

set\_quotes (lq=0x34c78 “<QUOTE>”, rq=0x34c88 ”<UNQOUTE>”)

at input.c: 530

530 if (lquote != def\_lquote)

上面显示m4暂停的子函数的调用栈。它显示栈的总体信息。我们可以使用backtrace命令（还可以写为bt），来查看我们在整体调用栈的位置：backtrace命令显示每个有效子程序的调用栈。

(gdb) **bt**

#0 set\_quotes (lq=0x34c78 “<QUOTE>”, rq=0x34c88 “<UNQUOTE>”)

at input.c: 530

#1 0x6344 in m4\_changequote(argc=3, argv=0x33c70)

at builtin.c: 882

#2 0x8174 in expand\_macro (sym=0x33320) at macro.c: 242

#3 0x7a88 in expand\_token (obs=0x0, t=209696, td=0xf7fffa30)

at macro.c: 71

#4 0x70dc in expand\_input () at macro.c: 40

#5 0x2930 in main (argc=0, argv=0xf7fffb20) at m4.c: 195

我们向前执行几步来看看发生了什么。前两步，我们使用命令s；后面两次我们使用命令n来避免进入函数xstrdump内部。

(gdb) **s**

0x3b5c 532 if (rquote != def\_rquote)

(gdb) **s**

0x3b80 535 lquote = (lq == nil || \*lq == ‘\0’) ? \

def\_lquote: xstrdup(lq);

(gdb) **n**

536 rquote = (rq == nil || \*rq == ‘\0’) ? def\_rquote \

: xstrdup(rq);

(gdb) **n**

538 len\_lquote = strlen(rquote);

最后一行看起来有点奇怪；我们检查一下变量lquote和rquote，看看它们是否是我们所指定的内容。我们使用命令p（print）来查看它们的数值。

(gdb) **p lquote**

$1 = 0x35d40 “<QUOTE>”

(gdb) **p rquote**

$2 = 0x35d50 “<UNQUOTE>”

lquote和rquote确实是正确的。为了查看一下上下文，我们使用l（list）命令显示当前行周围10行的源代码内容。

(gdb) **l**

533 xfree(rquote);

534

535 lquote = (lq == nil || \*lq == ‘\0’) ? def\_lquote \

: xstrdup(lq);

536 rquote = (rq == nil || \*rq == ‘\0’) ? def\_rquote \

: xstrdup(rq);

537

538 len\_lquote = strlen(rquote);

539 len\_rquote = strlen(lquote);

540 }

541

542 void

我们执行越过设置len\_lquote和len\_rquote的两行，然后检查一下这两个变量的数值。

(gdb) **n**

539 len\_rquote = strlen(lquote);

(gdb) **n**

540 }

(gdb) **p len\_lquote**

$3 = 9

(gdb) **p len\_rquote**

$4 = 7

如果len\_lquote和len\_rquote分别代表lquote和rquote字符串的长度，那么上面的数值就是错误的。我们可以使用p命令来重新设置一下这俩个值。p命令打印出表达式的结果，表达式可以包含子程序调用和赋值语句。

(gdb) **p len\_lquote = strlen(lquote)**

$5 = 7

(gdb) **p len\_rquote = strlen(rquote)**

$6 = 9

上面修改就足够了么？我们使用c（continue）命令让m4继续执行，然后检查一下前面引起错误的命令：

(gdb) **c**

Continuing.

**define(baz, defn(<QUOTE>foo<UNQUOTE>))**

baz

0000

正确了！新的引用像默认的一样工作了。问题就是长度赋值的错误。我们输入EOF让m4退出：

**Ctrl-d**

Program exited normally.

信息“Program exited normally”来自于GDB；它表明m4已经退出。我们使用quit命令退出GDB的会话。

(gdb) **quit**

# 进入和退出GDB

本章讨论如何开启GDB，如何退出GDB。概要是：

* 输入‘gdb’来开启GDB。
* 输入*quit*或者*Ctrl-d*来退出。

## 调用GDB

通过执行程序gdb来调用GDB。一旦开启，GDB从终端读取命令，直到你通知它退出。

可以在运行GDB的时候指定一些参数和选项，用来指定外部的调试环境。

这里描述的命令行选项用来覆盖一些条件；在一些环境下面，一些选项可能会无效。

经常使用的的一个启动GDB的参数是，指定可运行的程序：

*gdb program*

另外还可以在指定程序的同时指定core file文件：

*gdb program core*

另外，第二个参数还可以是正在运行的程序的进程ID：

*gdb program 1234*

这样会将GDB绑定到进程1234上。（除非有个文件名称为‘1234’；GDB会首先检查core file。）

使用GDB的第二个参数需要一个完整的系统环境；当使用GDB作为裸板的远程调试工具时，不可能获得进程，也经常不会存在core dump。GDB在不能绑定到一个进程或者不能读取core dumps时会发出警告信息。

另外还可以在可执行程序后面的—args后面跟任意的参数，来指定下一级的参数。这个选项终止GDB对选项的处理。

*gdb –args gcc –O2 –c foo.c*

这个命令是用gdb调试gcc，gcc的参数为‘-O2 –c foo.c’（参考4.3）。

可以在运行gdb时通过使用-silent参数使gdb不打那些非保证的信息：

*gdb -silent*

还可以通过命令行选项进一步控制GDB的启动。GDB本身会提供这些提示信息。

输入

*gdb -help*

来显示所有可选的选项，包括功能的简短描述（简写为‘gdb -h’）。

### 选择文件

当GDB启动时，它读取除了由选项指定的可执行文件和corefile外的所有参数。这就好像这些参数通过独立的‘-se’和‘-c’（或者‘-p’）选项来指定一样。（GDB把第一个没有选项标志的参数当做‘-se’选项的参数；把第二个没有选项标志的参数当做‘-c’/‘-p’选项的参数。）如果第二个参数前面有减号，GDB首先尝试将它作为进程，如果失败，尝试把它当做corefile而打开。如果你的corefile文件名字是数字，可以通过在文件名前面加上‘./’前缀，比如‘./12345’，以防止GDB将它当做进程id。

如果GDB没有配置为支持corefile，比如大多数的嵌入式目标，那么它会发出警告并且忽略这个参数。

很多选项都有长和短两种形式；下面的列表中会列出。如果你将参数截短，GDB仍然可以识别它的长形式，所以参数应该达到无歧义的长度。（如果你喜欢，可以将参数使用‘--’而不是‘-’前缀，尽管我们只介绍了常用的习惯。）

-symbols *file*

-s file 从文件*file*中读取符号表。

-exec *file*

-e *file* 适当的时候使用文件*file*当做可执行文件去执行，结合core dump来检查纯数据。

-se *file* 从文件*file*中读取符号表，并且把它当做可执行文件。

-core *file*

-c *file* 使用文件*file*当做core dump来检查。

-pid *number*

-p *number* 连接进程ID *number*，就像attach命令一样。

-command *file*

-x *file* 从文件*file*中执行命令。文件的内容就像原命令一样去执行。参考23.1.3[命令文件]。

-eval-command *command*

-ex *command* 执行单个GDB命令。这里命令可以多次执行以调用多个命令。必要的时候可以穿插使用。

gdb –ex ‘target sim’ –ex ‘load’ \

-x setbreakpoints –ex ‘run’ a.out

-init-command *file*

-ix *file* 在加载下一级之前执行文件*file*中的命令（但是在加载gdbinit文件之后）。参考2.1.3章[启动]。

-init-eval-command *command*

-iex *command* 在加载下一级之前执行单个命令（但是在加载gdbinit文件之后）。参考2.1.3章[启动]。

-directory *directory*

-d *directory* 向查找源文件和脚本的路径中添加新的目录。

-r

-readnow 立即读取每个符号表文件的所有符号表，而不是默认的。默认的行为是在需要的时候增量的读取符号表。这个选项使启动变得更慢，但是后续的操作会更快。

### 选择模式

你可以使用各种可选的不同模式运行GDB，比如，批量（batch）模式和安静（quiet）模式。

-nx

-n 不执行任何初始化文件中的命令。有三个初始化文件，按照下面的顺序加载：

‘system.gdbinit’

这是系统级的初始化文件。它的位置通过-with-systme-gdbinit配置命令指定（参考C.6章[系统级的配置]）。当GDB启动的时候它在命令行选项执行之前先加载。

‘~/.gdbinit’

这是在你的home目录下的初始化文件。它在‘system.gdbinit’加载之后加载，在命令选项执行之前。

‘./.gdbinit’

这是当前目录下的初始化文件。它最后加载，在除了-x和-ex选项的命令选项执行之后。命令选项-x和-ex在最后执行，在加载‘./.gdb.init’文件之后。

关于启动过程的更多信息，参考[2.1.3章[启动]](#_GDB在启动时做了些什么)。关于如何写命令文件，参考23.1.3章[命令文件]。

-nh 不执行‘~/.gdbinit’文件中的命令，此文件在你的home目录下。参考[2.1.3章[启动]](#_GDB在启动时做了些什么)。

-quiet

-silent

-q “安静”。不打印介绍性的和产权相关的信息。这些信息在批量（batch）模式中也不会出现。

-batch 在批量模式下运行。在执行完所有由-x选项指定的命令后返回0。如果在执行过程中出错返回非0值。批量模式禁用分页功能（pagination），设置终端的宽度和高度为无限，参考22.4[屏幕大小]，就像执行了*set confirm off*指令一样（参考22.8章[信息/警告]）。

批量模式在将GDB当做一个过滤器的时候有用，比如在另一台机器上下载并且运行一个程序；为了使这种模式更加有用，信息

Progam exited normally.

（当一个程序在GDB的控制下运行的时候会显示）在批量模式下不会显示。

-batch-silent 运行像‘-batch’模式一样的批量模式，但是完全安静的。所有GDB的标准输出stdout被禁止（标准错误输出stderr不受影响）。这个模式比‘-silent’更加安静，并且对于交互的会话是无效的。

在使用输出‘Load section’信息的目标机上，这种模式非常有用。

目标机将通过GDB进行输出，所以直接输出到stdout的信息也都不存在了。

-return-child-result

GDB的返回值是子程序的返回值（被调试的程序），下列情况除外：

GDB异常退出。比如，由于一个不正确的参数或者内部错误。这种情况下的返回值与通常的一样，而不是‘-return-child-result’。

用户使用显示的数值退出。比如，‘quit 1’。

子程序没有运行，或者不允许结束，这种情况会返回-1。

这个选项在与‘-batch’或者‘-batch-silent’选项配合，将GDB用来远程调试或者当成仿真接口的时候非常有用。

-nowindows

-nw “无窗口”。如果GDB携带内置的图形界面（GUI），那么这个选项将禁止图形界面的运行。如果没有内置图形界面，这个选项没有影响。

-cd *directory* 使用*directory*作为GDB的工作目录，而不是当前的目录。

-data-directory *direcotory*

使用*directory*作为GDB的数据目录。数据目录是GDB查找相关辅助文件的目录。参考18.6章[数据文件]。

-fullname

-f GNU的Emaces当使用GDB作为子程序的时候设置这个选项。它告诉GDB在每次显示堆栈信息的时候（包括程序的每次暂停），使用全文件名和行号进行输出。这种可识别的格式为两个‘0x32’字符，后面跟着使用冒号分割的文件名，行号和字符位置，并且新起一行。Emacs-to-GDB接口程序使用两个‘0x32’字符作为显示源代码的信号。

-annotate *level*

这个选项设置GDB内部的注释级别。它的效果和使用‘set annotate *level*’一样（参考28章[注释]）。注释的级别控制着GDB与它的提示符，表达式，源代码和其他类型的输出一起打印多少信息。0级别是常用的，1级别在GDB作为GNU Emacs的子程序时使用，3级别是程序控制GDB的最大注释，2级别已经不用了。

注释的级别功能在GDB/MI（参考27章[GDB/MI]）下被很大的限制住。

--args 改变命令行解释行为，用以将后面的参数当做下一级的参数去执行。这个选项终止选项的处理。

-baud *bps*

-b *bps* 设置GDB在远程调试使用穿行接口的线速（波特率或者每秒的bit数目）。

-l *timeout* 在使用GDB的远程调试时设置通信的超时时间（以秒为单位）。

-tty *device*

-t *device* 使用设备*device*作为程序的标准输入和输出。

-tui 在启动的时候使能文本用户界面（*Text User Interface*）。文本用户界面管理终端的一些文本窗口，显示源代码，汇编，寄存器和GDB命令的输出（参考25章[GDB文本用户界面]）。不要在Emacs下调用GDB时使用这个参数（参考26章[在GNU Emacs下使用GDB]）。

-interpreter *interp*

使用解释器*interp*作为和控制程序或者设备的接口。这个选项是通过程序设置GDB的后台。参考24章[命令解释器]。

‘—interpreter=mi’（或者‘—interpreter=mi2’）使GDB使用自版本6.0包含的GDB/MI的接口（参考27章[GDB/MI接口]）。之前的在版本5.3中包含的并且使用命令‘—interpreter=mi1’来选择的GDB/MI接口，已经不再使用了。再早的GDB/MI接口已经不再支持了。

-write 以读和写的方式打开可执行的core文件。这个相当于在GDB的命令行内执行‘set write on’命令（参考17.6节[补丁]）。

-statistics 这个命令使GDB在完成每一条命令，回到命令提示符之后打印出关于时间和内存使用的一些统计信息。

-version 这个选项使GDB打印出它的版本号和免责声明，并且退出。

-configuration

这个选项使GDB详细打印出在编译时的配置参数信息，然后退出。这些信息在报告GDB的bugs的非常有用（参考31章[GDB Bugs]）。

### GDB在启动时做了些什么

下面描述了GDB在启动一个会话的过程中都做了什么：

1. 通过命令行的参数设置命令解释器（参考[2.1.2[选择模式]](#_选择模式)）。
2. 读取系统级的初始化文件（如果在编译GDB时使用了‘—with-system-gdbinit’；参考C.6[系统级配置和设置]）并且执行此文件中的所有指令。
3. 读取home目录下的配置文件（如果存在）并且执行文件中的所有命令。
4. 按照指定的顺序执行通过‘-iex’和‘-ix’设置的命令和命令文件。一般应该使用‘-ex’和‘-x’选项替代，但是通过前面这种方式你可以在GDB初始化文件（当前工作目录下的）执行之前，并且在下一级加载之前应用设置。
5. 处理命令行选项和操作数。
6. 如果‘set auto-load local-gdbinit’设置为‘on’，读取当前工作目录下的初始化文件（如果存在）并且执行（参考22.7.1节[当前目录下的配置文件]）。这一步只有在当前的目录与你的home目录不同时才会执行。这样，你可以有多于一个初始化文件，一个通用的在你的home目录，另一个，通过你调试的程序来指定，在调用GDB的工作目录下。
7. 如果命令行执行一个调试的程序，或者绑定一个程序，或者一个core文件，GDB加载一些自动加载的程序脚本或者共享库。参考22.7节[自动加载]。

如果想要禁用在启动时自动加载，必须执行如下形式的指令：

$**gdb –iex “set auto-load python-scripts off” myprogram**

选项‘-ex’不会起作用，因为自动加载禁用的太晚了。

1. 按照指定的顺序执行通过‘-ex’和‘-x’指定的命令和命令文件。参考23.1.3节[命令文件]，得到更多了GDB命令文件信息。
2. 读取在history文件中的命令历史记录。参考22.3节[命令历史]，得到命令历史的更多信息和GDB在哪里记录它。

初始化文件和命令文件的语法相同（参考23.1.3章[命令文件]），并且GDB处理的方式也一样。在home目录下的初始化文件可以设置影响后续的命令行选项和操作的处理。如果使用‘-nx’选项那么初始化文件不会执行（参考[2.1.2[选择模式]](#_选择模式)）。

显示gdb在启动的时候加载初始化文件的列表，使*gdb –help。*

GDB的初始化文件经常称为‘.gdbinit’。GDB的DJGPP端口由于受DOS文件系统对文件名的限制，使用的文件名为‘gdb.ini’。GDB的windows端口使用标准的名字，但是如果在home目录下存在‘gdb.ini’文件，系统给你发出一个重命名文件为标准文件的警告。

## 退出GDB

quit [*expression*]

q 使用quit（简写为q）命令退出GDB，或者输入文件结束字符（一般为*Ctrl-d*）。如果不提供表达式（即*expression*），GDB会正常退出；否则以表达式的值作为错误码返回。

中断（经常为*Ctrl-c*）并不能退出GDB，而是退出GDB正在处理的命令然后回到GDB的命令行。在任何时候输入中断都是安全的，因为GDB只有在它安全的时候才会受到影响。

如果使用GDB控制一个绑定的程序或者设备，可以使用detach命令代替它（参考[4.7[调试正在运行的程序]](#_调试一个正在运行的程序)）。

## Shell命令

如果在调试程序的时候偶然需要执行shell命令，不要退出或者挂起GDB；可以使用shell命令。

shell *command-string*

!*command-string*

调用标准shell去执行命令*command-string*。在!和*command-string*之间不需要空格。如果存在，由环境变量SHELL决定运行那个shell。否则，GDB使用默认的shell（Unix系统为‘/bin/sh’，MS-DOS系统为‘COMMAND.COM’，等等）。

make工具在开发环境下经常被使用。在GDB中不需要使用shell去调用这个命令：

make *make-args*

使用指定的参数执行make程序。与命令‘shell make *make-args*’作用是一样的。

输出日志

可能需要将GDB的命令保存到一个文件中。下面是控制GDB的日志的几条命令。

set logging on

使能日志记录。

set logging off

禁用日志记录。

set logging file *file*

改变当前日志文件的名称。默认的日志文件为‘gdb.txt’。

set logging overwrite [on | off]

默认情况下，GDB会追加日志文件。如果需要覆盖那么就设置为overwrite on。

set logging redirect [on | off]

默认的情况下，GDB的输出会进入终端和日志文件。如果需要只记录日志文件，那么就设置redirect on。

show logging

显示当前日志设置的信息。

# GDB命令

如果简写是无歧义的，你可以将GDB的命令简写为开始的几个字母；可以通过键入RET（即回车符）来重复上一个命令。还可以使用TAB键让GDB实现命令的自动补全功能（如果不是唯一的，则显示可选的命令的列表）。

## 命令语法

GDB的命令是单行的输入。对于长度没有限制。命令以一个命令名开始，接着是依赖于命令的参数。比如，step命令接收一个指定单步执行次数的参数，如‘step 5’。也可以使用不加参数的step命令。一些命令是没有参数的。

如果命令的简写是无歧义的那么这个命令就可以截短。其它的一些命令的简写在每个命令中介绍。一些情况下，含糊不清的简写是允许的；比如s是step命令的简写，尽管还有其他一些命令是以s开头的。可以通过将命令名作为help命令的参数来了解命令的简写。

一个空行（键入回车符）在GDB中意味着重复前一个命令。一些命令（比如run）在这种情况下是不会重复的；这些命令一旦重复执行会引起不必要的麻烦。用户定义的命令可以不受这一点的限制；参考23.1.1[定义]。

list命令和x命令，在使用RET进行重复的时候，不使用上一次的参数而是使用新的参数。这样有利于查看源代码和内存。

在GDB中RET还有其他的用法：为了将冗长的输出分区，这个用法和常用的more工具相似（参考22.4章[屏幕大小]）。由于这种情况下很容易按下多个RET，GDB在这种情况下禁止命令重复。

以#开始直到这一行的末尾的文本是注释；它什么也不做。这一点在命令文件中很有用（参考23.1.3[命令文件]）。

Ctrl-o用来绑定重复复杂的命令序列。这个命令接受当前行，比如RET，之后获取从历史到下一行以便编辑。

## 命令补全

如果只有一种可能，GDB会补全命令的后面的字符。它还可以在任意时刻向你展示下一个可能的单词。这个功能在GDB命令，GDB子命令和程序的符号表名称中都有效。

当你在任何时候需要GDB进行补全的时候就按TAB键。如果只有一个可能，GDB补全这个单词，然后等待你完成命令（或者按下RET以完成）。比如，如果你键入

（gdb）**info bre TAB**

GDB会补全单词‘breakpoints’的剩余部分，因为这个以‘bre’开头的单词是info子命令的唯一选择：

（gdb）infobreakpoints

这个时候，如果需要执行info breakpoints命令，就按下RET键。如果‘breakpoints’不是你想要的命令，就按backspace键进行修改。（如果你确认要使用info breakpoints命令，你也可以在键入‘info brea’后直接键入RET，这样是使用GDB的简写命令）。

当你按下TAB键时如果有多重选择，GDB会发出响铃。你可以键入更多的单词然后再进行匹配；或者再次按下TAB键，GDB会显示所有可能的补全命令。比如，你可能想要在一个以‘make\_’开头的函数上面设置断点，但是当你键入*b make\_TAB*时听到一声响铃。再次键入TAB会显示所有以make\_开头的函数，比如：

（gdb）**b make\_TAB**

GDB响一次铃；再次按下TAB键，显示为：

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

在显示完所有可用的之后，GDB拷贝你输入的部分（‘本例中为b make\_’）以便你完成命令。

如果你只想查看可用的列表，直接键入*M-?*而不是两次TAB。*M-?*是*META?*。你可以在按住META键（如果键盘上有）的同时输入？，或者在？后输入ESC。

有时你需要的字符串，逻辑上是一个单词，可能包含括号或者其它GDB不识别的单词符号。为了防止这种情况，可以使用‘’’（单个引号）将单词包含起来。

更普遍的情况是你需要键入C++的函数名。由于C++的函数支持重载（相同的函数名但是不同的参数类型）。比如，当你需要设置一个断点的时候需要区分是函数name(int)还是name(float)。在这种情况下使用补全功能，在函数名称前面加上一个单引号‘’’。这样告诉GDB这个函数可能包含除了名称补全之外的其他信息：

（gdb）**b ‘bubble(*M-?***

bubble(double, double) bubble(int, int)

（gdb）**b ‘bubble(**

在一些情况下，GDB可以使用引号提示你需要键入完整的名称。这种情况下，如果你不以引号开始的话GDB会插入引号：

（gdb）**b bubTAB**

GDB将你的输出改为如下，并且响铃：

（gdb）b ‘bubble(

通常来说，GDB在你需要补全一个重载函数的时候会通知你（插入引号）需要输入引号。

更多的关于重载函数的问题，参考15.4.1.3节，可以使用命令set overload-resolution off来禁止重载的处理，参考15.4.1.7节。

当需要补全一个结构体里面的成员的时候，GDB也会将类型限制在已经键入的结构之内：

（gdb）**p gdb\_stdout.*M-?***

magic to\_fputs to\_rewind

to\_data to\_isatty to\_write

to\_delete to\_put to\_write\_async\_safe

to\_flush to\_read

这是由于gdb\_stdout是一个在源代码中定义为struct ui\_file类型的变量：

struct ui\_file

{

int \*magic;

ui\_file\_flush\_ftype \*to\_flush;

ui\_file\_write\_ftype \*to\_write;

ui\_file\_write\_async\_safe\_ftpye \*to\_write\_async\_safe;

ui\_file\_fputs\_ftype \*to\_fputs;

ui\_file\_read\_ftype \*to\_read;

ui\_file\_delete\_ftype \*to\_delete;

ui\_file\_isatty\_ftype \*to\_isatty;

ui\_file\_rewind\_ftype \*to\_rewrind;

ui\_file\_put\_ftype \*to\_put;

void \*to\_data;

}

## 获得帮助

可以在任何的时候使用help命令，来获得命令的帮助信息。

help

h 可以使用不带任何参数的help（简写为h）命令来显示分类的命令列表：

(gdb) **help**

List of classes of command:

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)

help *class* 使用通用的帮助类作为参，可以得到此类中的命令列表。比如，下面是status类帮助的输出：

(gdb) **help status**

Status inquires.

List of commands:

info – Generic command for showing things

About the program being debugged

show – Generic command for showing things

about the debugger

Type “help” followed by command name for full

documentation.

Command name abbreviations are allowed if unambiguous.

(gdb)

help *command*

以一个命令名称当做参数，GDB显示此命令的用法的概述。

apropos *args*

apropos命令以参数的正则表达式查找GDB的所有命令，和它们的文档。显示出所有符合条件的信息。比如：

apropos alias

显示出：

alias -- | Define a new command that is an allias of an existing command

aliases – Aliases of other commands

d – Delete some breakpoints or auto-display expressions

del – Delete some breakpoints or auto-display expressions

delete – Delete some breakpoints or auto-display expressions

complete *args*

这个命令列出所有以*args*开头的命令。比如：

complete i

显示出：

if

ignore

info

inspect

这个命令在GNU Emacs下使用。

除了help，还可以使用info和show命令查询程序的状态信息，或者GDB本身的信息。每个命令都支持许多查询的主题；本文会在合适的上下文介绍它们。这两个命令可以显示的信息包括所有子命令的命令，变量，函数的索引指针。参考[命令和变量索引]。

info 这个命令（简写为i）用来显示你的程序的状态。比如，可以使用info args来显示传递给函数的参数信息，用info registers显示寄存器中当前的信息，或者用info breakpoints来显示设置的断点的信息。可以使用help info命令或者所有info 子命令的列表。

set 使用set可以将环境变量设置为一个表达式的结果。比如，可以使用命令set prompt $将GDB的命令提示符设置为$。

show 与info对比，show命令显示GDB本身的状态信息。可以使用set命令改变大部分可以show的信息；比如，你可以使用set radix改变系统使用的显示数字的进制，或者简单的使用命令show radix进行查询。

可以使用不带参数的show命令显示所有可以设置的参数的当前值；还可以使用info set。这两个命令显示的结果一样。

下面是show子命令的一些杂项，这些项没有对应的set命令：

show version

显示正在运行的GDB的版本。在GDB的bug报告中应该包含这个信息。如果在你的站点上运行多个GDB的版本，你可能需要检查正在运行的是哪个版本；在GDB的发展过程中，引入了新的命令，并且老的命令可能不存在了。并且，很多制造商发行变种的GDB，在GNU/Linux的发行版中也存在着变种版本的GDB。这个版本号与启动GDB时的版本号是一致的。

show copying

info copying

显示复制GDB的限制信息。

show warranty

info warranty

显示GNU的免责声明（如果你的GDB版本携带的话）。

show configuration

显示GDB编译时的配置信息。这个命令显示调用‘configure’脚本的选项参数和脚本自动检测到的配置参数。当报告一个GDB的bug的时候，在报告中包含这些信息很重要。

# 在GDB下运行程序

当需要在GDB下运行程序时，首先在编译的时候需要产生调试信息。

依赖于你的环境，在启动GDB的是时候可能携带参数。如果是本地调试，可能需要重定向程序的输入和输出，调试一个正在运行的进程，或者杀死子进程。

## 编译调试信息

为了有效地调试一个程序，在程序编译的时候需要生成调试信息。调试信息存储在目标文件（object file）中；它们描述每个变量和函数的的类型，和相应的源代码的行号与可执行文件的地址。

为了生成调试信息，在编译的时候使用‘-g’选项。

程序通过‘-O’选项进行编译优化后传递给你的客户。然而，一些编译器不能同时处理‘-g’和‘-O’选项。使用这种类型的编译器，你不能得到携带调试信息并且优化的可执行程序。

GNU的C/C++编译器GCC，支持‘-g’选项同时配合或者不配合‘-O’选项，可以调试优化的代码。我们推荐你编译程序的时候一直使用‘-g’选项。可能你认为你的程序是正确的，但现实往往不是如此。更多的信息，参考11章[优化代码]。

老版本的GNU C编译器允使用许变种的选项‘-gg’来生成调试信息。GDB现在不支持这种格式；如果你的GNU C编译器有这个选项，请不要使用。

GDB识别预处理宏并且显示它们的扩展（参考[12章[宏]](#_C预处理宏)）。很多编译器上，如果你只是指定了‘-g’选项，不会包含预处理宏的调试信息。GNU C编译器的3.1以及之后的版本，如果使用DWARF调试格式并且指定了选项‘-g3’，就会提供宏的信息。

更多关于GCC选项影响调试信息的内容，参考“Options for Debugging Your Program or GCC” in Using the gnu Compiler Collection (GCC)。

如果使用你的编译器支持的最新的DWARF格式的调试信息，你将获得最佳的调试体验。DWARF是目前GDB支持的最好的调试格式。

## 启动程序

run

r 使用run命令启动GDB下的程序。首先你必须将程序当做GDB的参数来执行（除了VxWorks），参考[第2章[进入和退出GDB]](#_进入和退出GDB)，或者使用文件或可执行文件命令（参考18.1节[指定文件的命令]）。

如果你运行程序的环境支持多进程，run命令创建一个下级进程，这个下级进程执行你的程序。在一些不允许多进程的环境中，run命令跳转到你的程序的开始处。其它的情况，比如远程的，经常是已经运行的。如果你得到如下的错误信息：

The “remote” target does not support “run”.

Try “help target” or “continue”.

那么使用continue来运行程序。可能需要先load（参考[load]）。

执行的程序会受到上级程序发送的一些信息的影响。GDB提供指定此信息的方式，但是必须在启动程序之前执行。（可以通过启动后来改变这些信息，但是这些改变只有在下次启动的时候才会生效。）这种信息可以分为四类：

参数 通过将run命令的参数将这些参数传递给你的程序。如果你的目标机支持shell，使用shell传递参数，这样你可以使用常用的惯例（比如通配符表达式和变量替换）来描述参数。在Unix系统下，可以通过SHELL环境变量控制使用哪个shell。参考[4.3节[程序的参数]](#_程序的参数)。

环境 你的程序会继承GDB的环境。可以通过GDB的命令set environment和unset environment来改变影响你的程序的环境变量。参考[4.4.节[程序的环境]](#_程序的环境)。

工作目录 你的程序继承GDB的工作目录。可以通过cd命令改变GDB的工作目录。参考[4.5节[程序的工作目录]](#_程序的工作目录)。

标准输入和输出

你的程序使用和GDB相同的标志输入和标准输出设备。可以才run命令中重定向输入和输出，或者使用tty命令来为程序设置不同的设备。参考[4.6节[程序的输入和输出]](#_程序的输入和输出)。

警告：当输入和输出重定向工作的时候，不可以使用管道将程序的输出传递给另一个程序；如果你这么做了，GDB可能会以调试错误的程序终止。

如果你执行了run命令，你的程序就会立即执行。参考[第5章[暂停和继续]](#_停止和继续)，来了解如何安排程序的暂停。一旦你的程序暂停，可以在程序里使用print或者call命令调用函数，参考[第10章[检查数据]](#_检查数据)。

如果子GDB上次读取符号的时候你的符号表文件的修改时间有了改动，GDB会丢弃所有的符号表，并且重新加载。如果这件事发生了，GDB试着维持你当前的断点。

start 主程序的名称根据语言的不同而不同。在C或C++中，主程序的名称固定为main，但是其它的语言比如Ada并不要求主程序的名称固定。调试器为启动程序并且在主程序的入口处暂停的方便的方法，依赖于使用的语言。

‘start’命令相当于在主程序的开始处设置一个临时断点并且运行‘run’命令。一些程序在主程序调用之前有一个执行启动代码的解释时期（elaboration phase）。这依赖于写程序时所使用的语言。比如在C++中，在main函数执行之前，静态变量和全局对象的构造函数被执行。所以在到达主函数之前就暂停是可能的。然而，临时断点依然挂起执行的进行。

将程序指定的参数通过给‘start’命令来进行传递。这些参数会逐字的传递给‘run’命令。如果在后续的调用‘start’或者‘run’命令是不携带参数，那么相同的参数会重新使用。

有时调试程序的解释时期（elaboration）是必要的。如果是这样的话，使用start命令就暂停的太晚了，因为程序已经过了解释时期了。在这种情况下，需要在执行程序之前将断点插入到解释时期的代码。

set exec-wrapper *wrapper*

show exec-wrapper

unset exec-wrapper

当设置了‘exec-wrapper’的时候，为调试而加载程序时使用指定的封装。GDB使用*exec wrapper program*的shell命令来启动你的程序。在程序和它的参数上会加上引用，但是封装上不会，如果适合你的shell，你更改加上引用。封装一直执行到你的程序，然后GDB开始接管。

你可以使用任何最终调用携带参数的execve程序作为一个封装。一些Unix工具是这样做的，比如env和nohup。任何以exec “$@”结尾的Unix的shell脚本也是可以的。

比如，你可以使用env来将一个环境变量传递给调试器，而不是在shell的环境中设置这个变量：

(gdb) **set exec-wrapper env ‘LD\_PRELOAD=libtest.so’**

(gdb) **run**

这个命令在调试本地的程序的时候是可用的，除了DJGPP，Cygwin，MS Windows和QNX Neutrino。

set disable-randomization

set disable-randomization on

这个选项（GDB中默认是使能的）将会关闭启动程序的虚拟地址空间的本地随机化。这个选项在多个调试会话中使可执行程序更好的可再生和调试会话中地址空间的重用是有用的。

这个特性只在特定的目标机（包括GNU/Linux）上面实现。在GNU/Linux上还可以通过下面的命令来达到相同的目的：

(gdb) **set exec-wrapper setarch ‘uname –m’ –R**

set disable-randomization off

是启动的可执行程序的行为不改变。一些bug只有程序在指定的地址加载的时候才会出现。如果在GDB下执行程序的时候bug不见了，可能是由于GDB默认的在平台（比如GNU/Linux）上调试单个程序的时候禁止了地址随机。使用*set disable-randomization off*来使这样的bug重现。

在一些可能的目标机上，虚拟地址空间随机分布保护程序防止一些安全方面的攻击。这种情况下攻击者需要知道具体的可执行代码的实际地址。随机分布地址使在代码的指定地址注入跳转这种滥用方式变得不可能。

预连接共享库提高了启动的性能，但是它使目标机上的特权进程通过非特权的访问就能预见共享库的地址。读取二进制的共享库足够用来装配恶意代码从而滥用它。一个预连接的共享库仅仅需要在启动时候的正常的程序重定位就能够加载在一个新的地址。还没有预连接的共享库经常加载在一个随机选择的地址。

位置独立的可执行程序（PIE，Position independent executable）包括与共享库相似的位置独立的代码，并且这个可执行程序在启动时能够加载在一个随机选择的地址。PIE可执行程序经常加载在随机的地址。使用gcc –fPIE –pie来编译这种可执行的程序。

堆（Heap），栈和定制的mmap区域也是随机放置的（只要随机化是使能的）。

show disable-randomization

显示当前的启动的程序的虚拟地址空间的本地随机化的设置。

## 程序的参数

传递给程序的参数可以通过run命令的参数指定。它们传递给shell，展开通配符，执行I/O重定向，然后传递给执行的程序。环境变量SHELL（如果存在）指定GDB使用那个shell。如果没有定义SHELL，GDB使用默认的shell（Unix下的‘/bin/sh’）。

在非Unix的系统，程序经常直接被GDB调用，GDB将I/O重定向仿真为相应的系统调用，使用程序的启动代码（startup code）将通配符展开，而不是shell。

使用不带参数的run命令使用上一次run命令的参数，或者是set args命令指定的参数。

set args 指定程序下次运行时使用的参数。如果你需要携带参数运行程序，在下一次的程序执行之前使用set args命令是不携带参数而再次执行的唯一方式。

show args 显示程序启动的参数。

## 程序的环境

环境包括环境变量和它们的值的集合。环境变量通常记录如下的信息，比如用户名，home目录，终端类型，程序运行时的查找路径。通常使用shell设置环境变量，你运行的程序会继承这些环境变量。当调试的时候，使用修改的环境来运行程序而不是重新启动GDB是有用的。

path *directory*

将*directory*添加到将要传递给程序的PATH环境变量（可执行文件的查找路径）的前面。GDB使用的PATH变量的数值是不变的。可能需要指定一些目录的名称，使用空白或者依赖于系统的分隔符（在Unix下为‘:’，在MS-DOS和MS-Windows下为‘;’）进行分隔。如果*directory*已经在路径之中了，它将会被提前，这样查找时能够更快。

在GDB查找路径的是时候可以使用字符串‘$cwd’来指定当前目录。如果使用‘.’，它指定为你执行path命令的目录。GDB在将*directory*添加到查找路径之前把在*directory*变量中的‘.’替换为当前的目录。

show paths 显示查找可执行文件的目录（PATH环境变量）。

show environment [*varname*]

打印在程序启动时将要传递给它的环境变量*varname*的内容。如果不指定*varname*，将会打印所有将要传递给程序的环境变量。可以将environment简写为env。

set environment *varname*[=*value*]

将环境变量*varname*的内容设置为*value*。这个数值只在你的程序中改变，在GDB中不变。*value*可以是任何的字符串；环境变量的数值只有字符串，依赖于你的程序进行解释。参数*value*是可选的；如果它不存在，变量的内容为空（null）值。

比如，下面的命令：

set env USER = foo

告诉被调试的程序，后续的执行的时候，它的用户是‘foo’。（‘=’前后的空白在这里为了更加清晰，它们并不是必须存在的。）

unset environment *varname*

从传递给程序的环境变量中删除*varname*变量。这与‘set env *varname=*’不同，unset environment将变量从环境中删除，而不是赋予它空值。

*警告*：在Unix系统中，GDB使用SHELL环境变量指定的shell（如果SHELL不存在使用‘/bin/sh’）。如果SHELL变量指定的shell运行初始化文件，比如C-shell的‘.cshrc’，或者BASH的‘.bashrc’，此文件中设置的变量也会影响你的程序。你可能需要删除由这些文件设置的环境变量，只需要登录时的环境变量，比如‘.login’或者‘.profile’。

## 程序的工作目录

每次使用run启动程序的时候，它继承GDB使用的当前的工作目录。GDB的工作目录初始的时候从它的父进（典型的为shell）程继承而来，可以在GDB中使用cd命令改变工作目录。

GDB的工作目录也是文件操作的默认目录。参考18.1节[指定文件的命令]。

cd [*directory*]

设置GDB的工作目录为*directory*。如果没有参数，参数为‘~’。

pwd 打印GDB的工作目录。

由于程序在运行过程中可能改变它的目录，所以不可能找到被调试的程序的当前的工作目录。如果运行的GDB配置了支持‘/proc’，可以使用info proc命令（参考21.1.3节[SVR4 程序信息]）找出被调试程序的当前工作目录。

## 程序的输入和输出

默认情况下，在GDB下运行的程序使用与GDB相同的输入和输出。GDB切换它自己的终端来与用户交互，它记录程序使用的终端模式，当继续运行程序的时候切换给程序。

info terminal

显示GDB记录的关于程序使用的终端模式的信息。

可以使用run命令携带shell的重定向的参数使程序的输入/输出重定向。比如：

run > outfile

启动程序，将输出分流到文件‘outfile’中。

另一种指定程序输入输出的方式是使用tty命令。这个命令以一个文件名作为参数，使这个文件是后来的run命令的默认文件。对于后来的run命令来说，它也重置子进程的控制终端。比如，

tty /dev/ttyb

将后来的run命令的输入和输出重定向到‘/dev/ttyb’终端上，使之成为控制终端。

显式的run命令的重定向会覆盖tty命令所设置的输入/输出设备，但是不会影响控制终端。

当你使用tty命令或者run命令的输入重定向，只有所运行的程序的输入被影响。GDB的输入还是来自原来的终端。tty是set inferior-tty命令的别名。

可以使用show inferior-tty命令使GDB显示出程序未来运行时所使用的终端。

set inferior-tty */dev/ttby*

设置被调试的程序的tty为*/dev/ttyb*。

show inferior-tty

显示被调试的程序所使用的tty。

## 调试一个正在运行的程序

attach *process-id*

这个命令绑定一个已经运行的程序，这个程序在GDB之外被启动。（info files显示显示你活动的目标机。）这个命令以进程ID作为参数。经常使用的查找Unix进程的进程ID的方式是使用ps工具，或者使用‘jobs -l’shell命令。

attach命令执行后再按下RET不会被重复执行。

使用attach命令，你的程序必须运行在支持进程的环境下；比如，attach不会在没有操作系统的裸板上有效。还需要有权限对这个进程发信号。

当使用attach时，调试器首先在当前的工作目录查找运行的程序，然后（如果程序没有找到）使用源代码的查找路径（参考9.5[指定源代码目录]）。还可以使用file命令加载程序。参考18.1节[指定文件的命令]。

GDB对安排到它调试的程序首先要做的是暂停这个程序。可以使用run命令开始时的GDB的命令检查和修改绑定的程序。可以插入断点；可以单步执行和继续执行；可以修改内存。如果想要程序继续执行，可以在GDB绑定程序后使用continue命令。

detach 当完成对绑定程序的调试时，可以使用detach命令使GDB放弃对它的控制。分离程序后继续程序的执行。在detach命令后，程序和GDB再次成为各自独立的程序，可以绑定其他的程序或者使用run命令开始一个程序。detach命令在运行后不会随着键入RET而重复执行。

如果当你绑定一个程序后退出GDB，GDB会分离这个程序。如果使用run命令开始的程序，GDB会杀死这个程序。默认的情况下，GDB会请求确认信息；可以通过set confirm命令来配置是否需要确认的步骤（参考22.8节[警告和通知选项]。）。

## 杀死子进程

kill 杀死GDB下运行的程序。

如果需要调试一个core dump而不是正在运行的进程，这个命令是有用的。GDB在程序运行时忽略任何的core dump文件。

在一些操作系统中，在GDB中设置程序的一些断点后，程序就不能在GDB之外执行了。这种情况下可以使用kill命令允许程序在调试器之外运行。

kill命令在想要重新编译和重新连接程序的时候也很有用，因为很多系统不允许在程序运行的时候修改程序的可执行文件。这种情况下，在执行下一次的run的时候，GDB发现文件已经改变，就会再一次读取符号表（同时试着保存当前的断点设置）。

## 调试多个子级（inferior）和程序

GDB允许在一个会话中执行和调试多个程序。另外，一些系统上的GDB运行同时运行几个程序（否则你需要退出一个然后再启动另一个）。通常情况下，允许启动多个可执行文件，每个程序允许多线程。

GDB使用一个称为子级（inferior）的对象代表每个可执行程序的状态。一个子级典型的对应一个进程，但是它更加通用，还可以应用于没有进程的目标机。子级可以在程序运行前被创建，在程序不存在时保留。子级有与进程ID不同的标识符。通常情况下，每个子集有自己的独占的地址空间，尽管一些嵌入式的目标机可能存在一些子级运行在一个地址空间的不同部分。每个子级可以有多个线程。

查看当前的子级信息，使用info inferiors命令：

info inferiors

打印GDB管理的当前的所有inferiors的列表。

GDB显示每个子级（按照下面的顺序）：

1. GDB分配的子级数值
2. 目标机系统的子级标识符
3. 正在运行的子级的可执行文件名称

前面带有星号‘\*’的是当前的子级。

比如：

(gdb) **info inferiors**

Num Discription Executable

2 process 2307 hello

\*1 process 3401 goodbye

切换当前的子级，使用inferior命令：

inferior *infno*

使数字为*infno*的子级成为当前的子级。参数*infno*是GDB分配的子级数值，在‘info inferiors’命令显示的第一个成员。

可以通过add-inferior和clone-inferior命令将多个可执行文件加入到一个调试的会话中。在一些系统中GDB可以在后续的调用fork和exec的时候自动将子级加入到当前的调试会话中。从调试的会话中删除子级使用remove-inferiors命令。

add-inferior [ -copies *n*] [ -exec *executable* ]

添加*n*个使用可执行程序*executable*去执行的子级。*n*默认为1。如果不指定可执行的程序，子级初始为空，没有任何程序。还可以在任何时候赋予或者改变子级中赋予的程序，使用携带可执行程序名称为参数的file命令。

clone-inferior [ -copies *n* ] [ *infno* ]

添加*n*个准备运行同一个程序的子集*infno*。*n*默认为1。*infno*默认为当前子级的数值。这个命令在想要运行程序的另一个实例的时候很方便。

(gdb) **info inferiors**

Num Description Executable

\*1 process 29964 helloworld

(gdb) **clone-inferior**

Added inferior 2.

1 inferiors added.

(gdb) info inferiors

Num Description Executable

2 <null> helloworld

\*1 process 29964 helloworld

现在可以切换到子级2并且运行。

remove-inferiors *infno…*

删除子集或者子集*infono…*这个命令不会删除正在运行的子级。对于这样的来说，先使用kill或者detach命令。

detach inferior *infno…*

从子级或者由GDB子级标志*infno*指定的子级中分离。子级依然会在info inferiors命令中显示，但是它的描述会变成‘<null>’。

kill inferiors *infno…*

杀死子级或者由GDB子级标志*infno*指定的子集。子级依然会在info inferiors中显示，但是它的描述变成了‘<null>’。

在命令detach，detach inferiors，kill，kill inferiors或者正常的程序退出执行完毕之后，子级仍然有效，并且在info inferiors命令中显示。

在GDB的控制下当子级开始或者退出时发出通知，使用set print inferior-events命令：

set print inferior-events

set print inferior-events on

set print inferior-events off

set print inferior-events命令使GDB在子级开始或者退出或者分离的时候发出通知。默认的情况下，这些信息不回打印出来。

show print inferior-events

显示当GDB检测到子级开始，退出或者分离时是否发出信息。

很多命令在调试多个程序时像调试一个程序时那样有效：比如，print myglobal会显示当前子级中的myglobal变量的值。

偶尔，当调试GDB本身的时候，可能需要得到一个会话内子级，程序，地址空间的关系。可使用maint info program-spaces命令来完成这个功能。

maint info program-spaces

打印GDB管理的所有的程序空间。

GDB显示每个程序空间（按照下面的顺序）：

1. GDB分配的程序空间号
2. 在程序空间加载的可执行程序的名称。

星号‘\*’开头的表示当前的程序空间。

另外，在每个程序空间行下面，GDB打印出不适合列成表格形式的额外信息。比如，绑定程序空间的子级列表。

(gdb) **maint info program-spaces**

Id Executable

2 goodbye

Bound inferiors: ID 1 (process 21561)

\*1 hello

这里我们可以看出在hello程序中没有运行的子集，同时进程21561正在运行goodbye程序。在一些目标机上，多个子集绑定一个程序空间是可能的。常用的例子是同时调试通过调用vfork形成的父子进程。比如，

(gdb) **maint info program-spaces**

Id Executable

\*1 vfork-test

Bound inferiors: ID 2 (process 18050), ID 1 (process 18045)

在这里，子级2和子集1都在运行同一个进程空间，因为子级1调用了vfork。

## 调试多线程程序

在一些应用系统中，比如HP-UX和Solaris，单个进程可以有多于一个的可执行线程。精确的线程语义在不同的操作系统上是不同的，但是通用的单程序线程和多进程是相似的，除了它们共享一个地址空间（意味着，它们可以检查和修改同一个变量）。换句话说，每个线程都有它自己的寄存器和执行栈，可能还会有私有的空间。

GDB为调试多线程提供如下的机制：

* 新线程的自动通知
* ‘thread *threadno*’，在线程之间相互切换的命令
* ‘info threads’，查询存在的线程的命令
* ‘thread apply [*threadno*] [*all*] args’，将一个命令应用于线程列表的命令
* 线程级别的断点
* ‘set print thread-events’，控制线程开始和结束的提示信息。
* ‘set libthread-db-search-path *path*’，如果默认的选择不适合程序，用户可以指定使用哪个libthread\_db。

*警告：*这些机制并不是在所有支持线程的系统上的GDB配置中都支持。如果你的GDB不支持线程，这些命令没有效果。比如，不支持线程的系统中在使用‘info thread’命令是什么也不显示，并且拒绝thread命令，如下：

(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的线程调试机制允许在程序运行的时候查看所有的线程，但是在GDB控制的时候，只有一个特殊的线程是关注的焦点。这个线程叫做当前线程（current thread）。显示程序信息的调试命令只关注当前线程。

当GDB检测到程序开启一个新的线程的时候，它以‘[New *systag*]’的信息格式显示目标机系统关于此线程的标识符。*systag*的形式是依赖于特定系统的线程标识符。比如，在GNU/Linux上，当GDB发现一个新线程的时候，可以看到

[New Thead 0x41e02940 (LWP 25582)]

与之对比，在SGI系统上，*systag*是形如‘process 368’的简单信息，没有其它的限定词。

为了调试，GDB为它分配一个线程数字，这个数值经常是一个整数，程序的每个线程都有。

info threads [*id…*]

显示程序中当前所有线程的统计信息。可选的选项*id…*，是使用空格分隔的线程标志，如果指定，只显示指定的线程。GDB显示每个线程（按照如下的顺序）：

1. GDB分配的线程数字
2. 系统的线程标志（*systag*）
3. 线程的名字，如果知道的话。线程可以由用户命名（参考下面），或者在某种情况下，由程序指定。
4. 此线程的当前栈信息。

左边有星号‘\*’标志的是当前的线程。比如，

(gdb) **info threads**

Id Target Id Frame

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

在Solaris上，可以使用Solaris指定的命令查看更多的用户线程信息。

maint info sol-threads

显示Solaris用户线程的信息。

thread *threadno*

使由*threadno*指定的线程为当前的线程。命令参数*threadno*是GDB分配的内部线程号，为‘info thread’命令显示的第一个成员。GDB显示选定的线程的系统标识符，和当前的栈信息：

(gdb) **thread 2**

[Switching to thread 2 (Thread 0xb7fdab70 (LWP 12747))]

#0 some\_function (ignore=0x0) at example.c:8

8 printf(“hello\n“)；

正如‘[New …]’信息，在‘Switching to’命令后显示的文本格式依赖于系统的关于线程的约定。

调试器的变量‘$\_thread’包含当前的线程数目。在写断点的条件表达式，命令脚本等时它很有用。参考10.11章[常用的变量]，获得常用变量的信息。

thread apply [*threadno* | all] *command*

这个命令允许用户将命令*command*作用于一个或多个线程上。通过指定参数*threadno*来决定将受影响的线程。可以是单个的线程数字，即在‘info threas’命令显示结果的第一列的数字；还可以是数字序列，比如2-4。将命令应用于所有的线程，键入命令*thread apply all commandI*。

thread name [*name*]

这个命令给当前的线程命名。如果不含有任何参数，那么任何用户指定的名称将会删除。线程的名称在执行‘info threads’的结果中显示。

在一些系统中，比如GNU/Linux，GDB可以通过操作系统给定的名称决定线程的名称。在这些系统，使用命令‘thread name’会覆盖系统指定的线程名称，当删除用户指定的名称时GDB还会显示线程的系统名称。

thread find [*regexp*]

查找并且显示与匹配表达式相匹配的线程的名称或者线程的系统名称。作为‘thread name’的补充，这个命令允许通过目标机的系统标签来定位线程。比如，在GNU/Linux上，目标机的系统标签是LWP id。

(GDB) **thread find 26688**

Thread 4 has target id ‘Thread 0x41e02940 (LWP 26688)’

(GDB) info thread 4

Id Target Id Frame

4 Thread 0x41e02940 (LWP 26688) 0x00000031ca6cd372 in select()

set print thread-events

set print thread-events on

set print thread-events off

这个命令允许你控制在GDB检测到新开一个线程或者线程退出的时候打印信息。默认的情况下，如果目标机能够获得到事件，是会打印信息的。不是所有的目标机都能禁止打印信息。

show print thread-events

显示当GDB检测到线程开启或退出的是否打印信息。

参考[5.5[暂停和开始多线程程序]](#_停止和开始多线程程序)，关于GDB在用户暂停或者开始多线程的程序的时候做些什么。

参考[5.1.2[设置观察点]](#_设置观察点)，关于多线程程序的观察点的信息。

set libthread-db-search-path [*path*]

如果设置了这个变量，*path*是使用冒号分隔的目录列表，GDB将在这些路径中查找libthread\_db库。如果省略了*paht*参数，‘libthread-db-search-path’变量将会重新置为默认的数值（在GNU/Linux和Solaris系统上为$sdir:$pdir）。内在的，默认的数值来自LIBTHREAD\_DB\_SEARCH\_PATH宏。

在GNU/Linux和Solaris系统上，GDB使用“帮助者”libthread\_db库来获得子级进程的线程信息。GDB将使用‘libthread-db-search-path’来查找libthread\_db。如果子级进程通过‘set auto-load libthread-db’命令设置了线程调试的库，GDB会首先采用。

‘libthread-search-path’中使用的变量‘$sdir’是查找并加载共享库的默认的系统路径。‘$sdir’是唯一一个不需要通过‘set auto-load libthread-db’使能的（参考22.8.2节[libthread\_db.so.1文件]）。

‘libthread-db-search-path’中的‘$pdir’是子级进程中加载libpthread的路径。

通过上述路径找到的libthread\_db库，GDB尝试使用当前的子级进程来初始化。如果初始化失败（由于libthread\_db和libpthread的版本不一致），GDB卸载libthread\_db，继续下一个目录的查找。如果都失败了，GDB会发出警告并且禁止线程调试。

目前只在一些平台上实现了设置libthread-db-search-path。

show libthread-db-search-path

显示查找libthread\_db的路径。

set debug libthread-db

打开或者关闭显示libthread\_db相关的事件。使用1来开启，0来关闭。

## 调试forks

在大多的系统中，GDB并没有为使用fork建立一个新的进程的调试提供特殊的支持。当fork一个程序时，GDB继续调试父进程，而子进程无阻的继续执行。如果在子进程将要执行的代码上设置了断点，子进程会获得一个SIGTRAP信号并且结束（除非它扑捉这个信号）。

然而，如果你需要调试子进程，需要做一些简单的工作。在子进程fork之后调用sleep。只在存在某个环境变量或者某个文件的时候才会休眠是很有用的，这样防止不想在子进程上运行GDB的而引入的延迟。之后告知GDB（如果希望同时调试父进程就再启动一个GDB）绑定到这个子进程上面（参考[4.7[绑定]](#_调试一个正在运行的程序)）。之后就可以像调试其他的绑定程序一样调试子进程了。

在一些系统上，GDB提供对使用fork和vfork产生的额外的进程的调试支持。当前，仅有HP-UX（11.x和之后的版本？）和GNU/Linux（2.5.60版本之后的内核）。

默认的情况下，当一个程序fork的时候，GDB仍然继续调试父进程，并且子进程无阻的继续执行。

如果你想要调试子进程而不是父进程，使用命令set follow-fork-mode。

set follow-fork-mode *mode*

设置程序调用fork或vfork时的调试模式。调用fork或vfork产生一个新的进程，参数*mode*可以为：

parent 原始的程序在fork之后继续进行调试。子进程无阻的执行。这是默认的。

child 在调用fork之后调试新的进程。父进程无阻的执行。

show follow-fork-mode

显示当前的调用fork或vfork的调试模式。

在Linux下，如果希望同时调试父进程和子进程，使用命令set detach-on-fork。

set detach-on-fork *mode*

告诉GDB在fork之后是否需要分离进程中的一个，或者在两个进程中都维持调试状态。

on 子进程（或者父进程，依赖于follow-fork-mode的设置）会被分离，并且独立的运行。这是默认的情况。

off 两个进程都在GDB的控制之下。一个进程（子进程或父进程，依赖于follow-fork-mode的设置）像平时一样进行调试，同时另一个暂停。

show detach-on-fork

显示detach-on-fork模式是否开启/关闭。

如果设置了‘detach-on-fork’模式为off，那么GDB会保持所有forked进程（包括内嵌的forks）。可以使用info inferiors命令显示GDB控制下的所有forked的进程，使用inferior命令在各个进程间切换（参考[4.9章[调试多个子级和程序]](#_调试多个子级（inferior）和程序)）。

退出调试的一个forked的程序，可以使用detach inferiors命令（使程序继续独立的运行），或者使用kill inferiors命令杀死程序。（参考[4.9章[调试多个子级和程序]](#_调试多个子级（inferior）和程序)）。

如果请求调试子程序，并且在vfork后面调用exec，GDB执行新的目标，并在新目标的第一个断点处暂停。如果在你的原始程序的main函数中设置了断点，那么在子程序中的main函数中也存在着相同的断点。

在一些系统中，当使用vfork产生一个子程序的时候，在exec调用完成之前不能进行调试父进程和子进程。

如果在一个exec调用执行完毕后向GDB输入run命令，新的调试目标重启。如果需要重启父进程，使用将父进程的可执行文件作为参数的文件命令。默认的情况是，在一个exec调用执行完毕后，GDB丢弃之前运行程序的映像。可以通过set follow-exec-mode命令改变这个模式。

set follow-exec-mod *mode*

设置GDB对程序调用exec的处理模式。一个exec调用替换原来进程的映像。

follow-exec-mode可能为：

new GDB创建一个新的子级，将新的进程绑定到新的子级上面。调用exec的程序可以通过重启之前的子级而重启。

比如：

(gdb) **info inferiors**

(gdb) **info inferior**

Id Description Executable

\*1 <null< prog1

(gdb) **run**

Process 12020 is executing new program: prog2

Program exited normally.

(gdb) **info inferiors**

Id Description Executable

\*2 <null> Prog2

1 <null> Prog1

same GDB将程序绑定在同一个子级中。在子级中新的可执行程序的映像代替之前的可执行程序的映像。在exec调用完毕后重启子级。这是默认的设置。

比如：

(gdb) **info inferiors**

Id Description Executable

\*1 <null> prog1

(gdb) **run**

Process 12020 is executing new program: prog2

Program exited normally.

(gdb) **info inferiors**

Id Description Executable

\*1 <null> prog2

可以使用catch命令使GDB在fork，vfork，或exec执行后暂停下来。参考[5.1.3章[设置捕捉点]](#_设置捕捉点)。

## 设置将来要返回的书签

在指定的操作系统上（目前只在GNU/Linux上有效），GDB可以保存程序状态的快照，叫做检查点（*checkpoint*），稍后可以回来。

返回一个检查点是非常高效的，不需要做任何事情，因为检查点已经被保存了。包括内存的改变，寄存器，甚至（在一些限制内）是系统状态。高效地，就像回到了保存检查点的那个时间。

这样，如果你正在单步调试一个程序，并且觉得就要到达发生错误的点了，可以保存一个检查点。之后，如果你错过了关键的点，那么不必重新启动程序而从头开始，只需要回到这个检查点，然后从这个检查点重新开始。

这个功能在需要花很多时间并执行很多的单步步骤才能到达想要的bug点的情况下是特别有用的。

在调试的过程中使用checkpont/restart方法：

checkpoint

保存调试程序当前状态的一个快照。checkpoint命令没有参数，但是每个检查点都会被赋予一个整数的id，和断点的id很像。

info checkpoints

列出在当前的调试会话中保存的检查点。对于每个检查点，会列出下面的信息：

Checkpoint ID

Process ID

Code Address

Source line, or label

restart *checkpoint-id*

使程序回到由*checkpoint-id*指定的检查点的状态。所有的程序变量，寄存器，栈帧空间等等都会回到保存的检查点当时的状态。本质上来说，gdb会“回滚时钟”到保存检查点的时间。

断点，GDB变量，命令历史等等不会受到恢复一个检查点的影响。通常来说，一个检查点只是恢复被调试程序内部状态，而不是调试者的状态。

delete checkpoint *checkpoint-id*

删除由*checkpoint-id*指定的之前存贮的检查点。

回到之前存储的检查点会重置被调试程序的用户状态，和系统状态的一个子集，包括文件指针。它不会向文件中“不写”数据，而是将文件指针回滚到之前的位置，这样之前写过的数据就会被覆盖掉。对于之前以读的方式打开的文件，指针也会回滚，这样之前读入的数据可以再次被读入。

当然，已经发送给打印机（或者其他外设）的字符不能“快照回来”，同时从串行设备接收的字符可以从程序的buffer中擦除，但是不可能“回退”给串行的管线，然后准备下一读取。相似地，已经被改变的文件内容也不会再恢复了（在这个时候）。

但是，尽管有这些限制，你仍然可以“回滚”你的程序到之前时间存贮的点，并且再次调试，同时可以改变程序的路径从而这次调试一个新的程序路径。

最后，当你回到一个检查点的时候程序的内部状态会有一点不同，即程序的进程id。每个检查点都有一个特别的进程id（或者PID），并且与原始程序的进程id不同。如果你的程序保存了进程id的副本，这可能会引起一个潜在的错误。

### 使用检查点的一个不明显的优点

在一些比如GNU/Linux的系统中，为了安全的因素，地址空间随机化会在新的程序中使用。这使得在重启程序时在绝对地址上设置断点或者观察点变得非常困难或者是不可能的，因为符号的绝对地址在这次运行的时候和下一运行的时候是不一样的。

一个检查点，是一个进程的完整拷贝。所以，如果你在main函数中设置了一个检查点，通过回到这个检查点而不是重启进程，就可以避免由于地址随机化带来的影响，符号会存在于相同的位置。

# 暂停和继续

使用调试器的主要目的是可以在程序退出之前暂停程序；或者是程序运行出错，可以找出原因。

在GDB中，程序可以因为一些原因而暂停，比如，信号，断点，或者在GDB命令比如step后遇到新的一行。之后你可以检查和修改变量，设置新的断点或者删除旧的断点，之后继续执行。通常，GDB显示的信息能够充分的解释程序的状态，可以在任何时候请求这些信息。

info program

显示程序的状态信息：它是否正在运行，哪个进程，为什么暂停。

## 断点，观察点，和捕捉点

断点（breakpoint）使程序在到达程序中某个指定的点的时候暂停。对于每个断点，可以添加条件来更加细微的控制程序是否暂停。可以使用break命令和它的变种来设置断点（参考[5.1.1节，[断点]](#_设置断点)），通过指定行号，函数名或者程序中的实际地址来控制程序暂停的位置。

在一些系统中，在可执行程序运行之前可以在共享库中设置断点。在HP-UX系统中有小幅的限制：必须在可执行程序运行后才能在不被程序直接调用的共享库例程中设置断点（比如，例程是pthread\_create调用的参数）。

观察点（watchpoint）是特别的断点，当表达式的值改变的时候使程序暂停。表达式可以是变量的值，或者是使用操作符连接的一个或者多个变量，比如‘a+b’。有时称之为数据断点（data breakpoints）。必须使用不同的命令来设置观察点（参考[5.1.2节[设置观察点]](#_设置观察点)），但是除此之外，可以想管理其他断点一样管理观察点：使能，禁止，和删除断点和观察点使用相同的命令。

可以安排在GDB暂停在一个断点时自动显示程序中的数值。参考10.7[自动显示]。

捕捉点（catchpoint）是另一种特殊的断点，它使程序在某种事件发生的时候暂停下来，比如抛出C++的异常或者加载一个库。和观察点一样，我们使用不同的命令来设置捕捉点（参考[5.1.3[设置捕捉点]](#_设置捕捉点)），但是除了这些，可以像管理其它断点一样管理捕捉点。（当程序收到一个信号的时候暂停，使用handle命令，参考[5.4章[信号]](#_信号)。）

GDB为每个断点，观察点，或者捕捉点在产生的时候分配一个数字；这些数字是从1开始的连续整数。在其他控制断点的各种特新的命令中使用断点数字来表明相应的断点。每个断点可能是使能或者禁止；如果被禁止了，它在使能之前对程序没有任何影响。

一些GDB命令接受断点数字序列进行操作。断点序列可以是单个断点数字，比如‘5’，或者这样的两个数字，按照升序，以连字符进行分隔，比如‘5-7’。当断点序列传递给一个命令时，序列中的每个断点都会被操作。

### 设置断点

使用break命令（简写为b）设置断点。调试器的常用变量‘$bpnum’记录最新设置的断点的数字；参考10.1章[常用变量]，关于使用常用变量能做些什么。

break *location*

在指定的*location*处设置断点，指定的位置可以是函数名，行号，或者指令的地址。（参考9.2[指定位置]，阐述所有指定位置的方式。）断点会使你的程序在指定位置的前面暂停。

当使用可以重载的源语言的时候，比如C++，一个函数名可能涉及到多个地方。参考10.2[含糊的表达式]，讨论这种情形。

还可以插入一个在指定的线程（参考5.4.4节[指定线程的断点]）或者指定的任务（参考15.4.9.5[Ada任务]）中匹配断点而使程序暂停下来的断点。

break 当不携带参数执行此命令时，在选择的栈帧中的下一条将要执行的指令上设置一个断点（参考第8章[检查栈]）。在选择的除了最里层的栈帧，将使你的程序在返回到此栈帧中的时候立即暂停。这与在选择的栈帧内执行finish命令的效果相同，除了finish命令不留下任何有效的断点。如果在最里层的栈帧中使用不带参数的break命令，GDB在下次到达此处时暂停；这种情况在内部循环的时候非常有用。

GDB在恢复执行的时候一般会忽略断点，直到执行了一个命令之后。如果不这样的话，就不能够在不首先禁止断点的时候继续向下执行。这条规则无论在你的程序中是否有断点都适合。

break … if *cond*

设置一个带条件*cond*的断点；当到达断点的时候计算表达式*cond*的数值，在数值非零的时候（也即是说为真的时候）暂停程序。‘…’代表上面描述的用来指定断点位置的参数（或者不带参数）。参考[5.1.6节[断点条件]](#_断点条件)，获得更多关于断点条件的信息。

tbreak *args*

设置一个一次性有效的断点。参数*args*和前面描述的break命令的参数一样，断点的设置方式也和break命令一样，但是这个断点在程序到达的时候会自动被删除。参考[5.1.5章[禁止断点]](#_禁止断点)。

hbreak *args*

设置一个硬件辅助（hardware-assisted）的断点。参数*args*和break命令的参数一样，断点的设置和break命令的也一样，但是这个断点需要硬件的支持，并且一些目标机的硬件可能不支持。这个命令的主要用于EPROM/ROM代码的调试，这样可以不用改变这个指令而给这个指令设置一个断点。这个命令可以用在有SPARClite DSU和大多数的x86系列所提供的新的陷阱产生。在这些目标机上，当程序访问分配给调试器寄存器的数据或者指令地址的时候会产生陷阱。硬件断点寄存器能够支持有限数量的断点。比如，在DSU上，一次只能设置两个数据断点，当多于两个的时候GDB会拒绝这个指令。在设置一个新的断点之前删掉或者禁止不用的硬件断点（参考[5.1.5章，[禁止断点]](#_禁止断点)）。参考[5.1.6章，[断点条件]](#_断点条件)。对于远程目标机来说，可以限定GDB将要使用的硬件断点的数量，参考[set remote hardware-breakpoint-limit]。

thbreak *args*

设置一个硬件辅助的临时断点。参数*args*和断点的设置都与hbreak命令相同。然而，与tbreak命令相似，这个断点在程序到达的时候将被删除。并且，与hbreak命令一样，这种断点需要硬件的支持，而且一些目标机的硬件不支持这个断点。参考[5.1.5[禁止断点]](#_禁止断点)。参考[5.1.6[断点条件]](#_断点条件)。

rbreak *regex*

在所有匹配表达式*regex*的函数上设置断点。这个命令在所有匹配的地方设置无条件的断点，并且列出所有通过它设置的断点。一旦这些断点被设置，它们就像使用break命令设置的断点一样被对待。可以像其它断点一样删除，禁止，或者使它们具有条件。

正则表达式的语法是标准的比如‘grep’工具使用的语法。这个语法与shell使用的语法不一样，比如foo\*匹配所有包含一个fo后面跟随0个或者多个o的函数。表达式的前面和后面都有一个隐式的.\*，如果只想匹配以foo开始的函数，那么用^foo。

当调试C++程序的时候，rbreak命令在没有指定任何特别类型的重载函数上设置断点是非常有用的。

rbreak命令可以给程序中的所有函数设置断点，比如：

(gdb) rbreak .

rbreak *file:regex*

当使用文件名限定rbreak命令的时候，它限制了表达式查找的函数只在指定的文件中。可以用来在给定文件中的所有函数都设置上断点，比如：

(gdb) rbreak file.c:.

使用冒号分隔的文件名和表达式之间可以使用空格。

info breakpoints [n …]

info break [n …]

打印所有的没有被删除的断点，观察点和捕捉点的集合。参数*n*代表着打印指定的断点（或者观察点，或者捕捉点）。对于每个断点，会显示下面的列信息：

*Breakpoint Numbers*

*Type* 断点，观察点或者捕捉点

*Disposition*

标志着当断点到达的时候是继续保持还是删除掉此断点。

*Enabled or Disabled*

使能的断点标志为‘y’。‘n’标志断点被禁止。

*Adress* 程序中的断点的内存地址。对于待定的断点，其地址是目前尚不知道的，这个域为‘<PENDING>’。这样的断点直到加载了其符号表的共享库或者此断点指定的行号被加载后才被激活。下面会详细解释。有多个位置的断点的此域为‘<MULTIPLE>’，下面会详细解释。

*What* 断点在源文件中的位置，由文件名和行号指定。对于一个待定的断点，传递给break命令的原始字符串将被显示，因为在未来加载合适的动态库之前不能解析函数的地址。

如果一个断点含有条件，有两种计算的模式：“宿主机”和“目标机”。如果模式是“宿主机”，断点的条件由在宿主机的GDB进行计算。如果是“目标机”，条件将由目标机进行计算。info break命令在紧接着断点的下一行显示条件，在圆括号内显示条件的模式。如果还存在断点命令，接着它后面（下一行）显示。待定的断点允许有指定的条件。条件在没有加载共享库而能够计算之前是不会被解析的。

携带参数*n*的info break命令只显示相应断点的信息。常用的变量$\_和默认的检查地址的命令x设置为最后一个设置断点的位置（参考10.6章，[检查内存]）。

info break命令显示此断点命中的次数。这与ignore命令联合起来使用特别有用。你可以忽略大量的命中的断点，只观察断点的信息而获得断点命中的次数，之后再次执行，忽略被那个数字少一次的次数。这样可以使你很快找到最后一次命中的断点。如果一个使能的断点被引用超过了一次，info break命令也会进行显示。

GDB允许在程序的同一个位置设置多个断点。这样做并不是由于太傻或者是毫无意义的。当一个断点含有条件的时候，就会非常有用了（参考[5.1.6章，[断点条件]](#_断点条件)）。

一个断点对应程序中的多个位置也是可能的。比如下面的条件：

* 程序中的多个函数可能会有重名的。
* 对于C++构造函数，GCC编译器产生几个函数体的实例，在不同的条件下使用。
* 对于C++的模板函数，指定的一行函数能够对应任意个初始化实例。
* 对于inline函数，在指定的源代码行可能对应多个被内联的函数的多个位置。

基于上述的条件，GDB会在每个相关的位置都插入断点。

对应于多个位置的断点会在断点表中使用几行来显示，一行显示头，其它的每行显示每个位置的断点。显示头的一行中的地址列显示‘<MULTIPLE>’。独立的位置的每一行包含位置的实际地址，并且显示此位置属于哪个函数。数字列的格式为断点数值.位置数值。比如：

Num Type Disp Enb Address What

1 breakpoint keep y <MULTIPLE>

stop only if i == 1

breakpoint already hit 1 time

1.1 y 0x080486a2 in void foo<int>() at t.cc:8

1.2 y 0x080486ca in void foo<double>() at t.cc:8

每个位置的断点都可以独立的使用“断点数字.位置数字”作为enable和disable命令的参数进行使能和禁止。但是不能从列表中单独的删除某一个位置的断点，只能同时删除全部属于同一个父断点的所有子断点（使用*delete num*命令，*num*是父断点的数字）。禁止或者使能父断点（参考[5.1.5[禁止]](#_禁止断点)）会影响所有的子断点。

经常会在一个共享库中包含断点。共享库可以显式的加载和卸载，当程序执行的时候还可能重复的加/卸载。为了支持这种情况，GDB会在任意一个共享库加载或者卸载的时候更新断点位置。典型地，你会在开始调试会话的时候设置一个共享库中的断点信息，此时共享库并没有被加载，共享库中的符号表还不可用。当你设置这样一个断点时，GDB将会向你询问是否设置一个待定的断点，此断点的地址尚不知道。

当程序运行的时候，当新加载一个共享库的时候，GDB重新计算所有的断点。当新加载的共享库包含待定断点的符号或者行号的时候，这个断点就会被解析并且成为一个正常的断点。当一个库卸载的时候，所有指向它的符号表或者源代码行的断点变成待定的断点。

断点的这种工作机制对多位置的断点也实用。比如，如果有一个C++的模板函数的断点，并且新加载的共享库中包含此模板函数的实例，那么会在此多位置的断点中再增加一个位置。

除了没有解析的地址，待定的断点与正常的断点没有区别。都可以设置条件或者命令，使能或者禁止等其它断点操作。

GDB为break命令不能进行断点地址解析的时候提供了额外的命令：

set breakpoint pending auto

这是默认的行为。当GDB不能找到断点的位置的时候，它想你寻味是否要建立一个待定的断点。

set breakpoint pending on

这个选项使一个不能解析位置的断点自动的成为一个待定的断点。

set breakpoint pending off

这个选项表明不会创建一个待定的断点。任何不能识别位置的断点会导致一个错误。这个设置不会影响任何之前设置的待定断点。

show breakpoint pending

显示当前对于待定断点的处理设置。

上面的设置只会影响break命令和它的变种。一旦已经设置了一个断点，它会自动的在共享库加载和卸载的时候进行更新（译者注：也就是说不会影响之前已经设置的断点的默认行为。）。

在一些目标机上，GDB能够自动得决定使用硬件的或者软件的断点，这依赖于断点地址是只读的还是可读写的。这个原则适用于使用break命令设置的断点，也适用于使用next或者finish等命令设置的内部断点。使用hbreak命令设置的断点，GDB会一直使用硬件断点。

可以使用下面的命令控制这个自动的行为：

set breakpoint auto-hw on

这是默认的行为。当GDB设置一个断点时，它会试着用目标机的内存地图来决定是必须使用软件的还是硬件的断点。

set breakpoint auto-hw off

这个选项使GDB不会自动选择断点的类型。如果目标机提供内存地图，GDB在只读的地址中设置软件断点时会发出警告。

GDB一般通过在断点处通过替换程序代码为指定的指令来实现断点。替换后的指定的指令一旦执行，就将控制权交给调试器。默认的情况下，只有在程序重新开始的时候，GDB会这样修改程序代码。一旦程序暂停，GDB会恢复原始的指令。这样防止GDB在突然断开的时候将断点信息留在源程序中。然而，如果使用慢速的目标机，插入和删除断点会降低性能。这个行为可以使用下面的命令进行控制：

set breakpoint always-inserted off

所有的断点，包括用户新添加的，只在目标重新开始的时候才会插入。所有的断点在程序暂停的时候都被删除。

set breakpoint always-inserted on

在任何时候都将所有的断点插入到目标程序中。如果用户添加一个新的断点，或者改变一个已经存在的断点，目标中的断点会立即更新。只有在断点被删除的时候才会从目标中真正删除。

set breakpoint always-instered auto

这是默认的模式。如果GDB使用非暂停模式控制子级（参考[5.5.2章，[非暂停模式]](#_非停止模式)），gdb就使用breakpoint always-inserted mode on的模式；如果GDB使用全暂停的模式控制子级，GDB就使用breakpoint always-insterted mode is off的模式。

GDB处理带条件的断点的方式是在到达断点的时候计算断点的条件。如果条件为真，那么被调试的程序暂停下来，否则继续执行。

如果目标机支持在后端计算条件，GDB会携带着断点的条件将断点下载到目标机。这个特性可以通过下面的命令进行控制：

set breakpoint condition-evaluation host

这个选项使GDB在宿主机一端计算断点的条件。无条件的断点会发送到目标机上面，反过来，目标机接收条件断点的触发事件并将它们发回给宿主机。这是标准的计算模式。

set breakpoint condition-evaluation target

这个命令选项使GDB将条件断点在插入的时候下载到目标机上面。目标机负责计算条件的表达式，并且在条件为真的时候将断点的事件发回给宿主机。由于目标机一段计算上的限制，有一些条件不能被计算，比如依赖于只有宿主机知道的局部变量的条件。还包括涉及到常用变量的条件表达式，太复杂以至于代理的表达式解析器不能解析，和太长的以至于不能发送到目标机上的表达式，尤其是目标机是远程的系统。这些情况下，这些条件将由GDB来计算。

set breakpoint condition-evaluation auto

这是默认的模式。如果目标机支持在后端计算条件断点，GDB会将断点的条件下载到目标机（应用前面讨论过的限制）。如果目标机不支持条件断点，GDB会退而求其次地在宿主机一端计算所有的条件。

GDB本身有时由于特殊的目的会在程序中设置断点，比如合理的处理longjmp（在C程序中）。这些断点被分配负的数字，从-1开始；‘info breakpoont’命令不会显示它们。可以通过GDB的管理命令‘maint info breakpoints’（参考[maint info breakpoints]）命令来查看它们。

### 设置观察点

可以使用观察点使程序在一个表达式的数值改变的时候暂停执行，而不必预知这个事件会发生的具体位置。（有时候也称为数据断点（*data breakpoint*）。）表达式可以简单的是单个变量，或者复杂的由操作符连接的多个变量。比如，包括：

* 对于单个变量的引用。
* 将一个地址强转为一个合适的数据类型。比如，‘\*(int \*)0x12345678’会观察指定地址开始的4-byte区域的数值（假设一个int类型占用4bytes）。
* 一个任意复杂的表达式，比如‘a\*b + c/d’。表达式可以使用程序原始语言支持的任何有效的操作符（参考15章，[语言]）。

可以给观察点的表达式设置为当前还不能被计算的表达式。比如，可以设置‘\*global\_ptr’的观察点，尽管‘global\_ptr’尚未初始化。GDB会在程序设置‘global\_ptr’并且表达式为真的时候暂停下来。如果表达式通过其他的方式而不是变量改变的方式变为真的话（比如，由于‘\*global\_ptr’所指向的地址里面的数据由于使用了malloc系统调用而变成可读的），GDB在表达式下次改变之前不会使程序暂停。

依赖于你的系统，观察点可能是软件的或者硬件的。GDB通过单步执行程序加每次都测试变量的数值的方式来实现软件的观察点，这样会使程序比正常执行慢上百倍。（但是这样仍然值得去做，用来在你不知道程序的那部分出现问题的时候观察错误。）

在一些系统上，比如HP-UX，PowerPC，GNU/Linux等大部分其它基于x86的目标上，GDB包含硬件观察点的支持，使得你的程序执行时不会变慢。

watch [-l | -location] *expr* [thread *threadnum*] [mask *maskvalue*]

设置一个表达式的观察点。GDB在这个表达式*expr*在程序中变成可写的并且它的值改变的时候使程序暂停下来。这个指令最简单的（也是最常用的）应用是观察单个变量的数值：

(gdb) watch foo

如果命令中包含一个[thread *threadnum*]参数，GDB会在由*threadnum*指定的线程上面观察表达式*expr*的数值的改变并使这个线程暂停下来。如果其他的线程也改变了这个表达式的数值，GDB不会使它们暂停下来。这种限制在单个线程上的观察点的方式只在硬件观察点下有效。

通常一个观察点代表表达式中变量的范围（参考下面）。

选项-location告诉GDB观察去观察表达式*expr*代表的内存。这种情况下，GDB计算表达式，使用结果的地址，并且观察这个地址的内存。结果的类型决定观察的内存的大小。如果表达式的结果没有地址，GDB会打印一个错误。

如果当前的架构支持这个特性（比如PowerPC的嵌入式架构，参考21.3.6节，[PowerPC嵌入式]），[mask *maskvalue*]选项允许创建带掩码的观察点。带掩码的观察点对一个观察的地址指定一个掩码。掩码指定了地址的一些bits（在掩码中复位的bits）在子级的观察的地址中应该被忽略。这样，带掩码的观察点同时观察很多地址，这些地址中没有被掩的位是与观察点中没有被掩的的位是相同的。maks选项意味着-location。比如：

(gdb) watch foo mask 0xffff00ff

(gdb) watch \*0xdeadbeef mask 0xffffff00

rwatch [-l | -location] *expr* [thread *threadnum*] [mask *maskvalue*]

设置一个观察点，使程序在表达式*expr*的值可读的时候暂停下来。

awatch [-l | -location] *expr* [thread *threadnum*] [mask *maskvalue*]

设置一个观察点，使程序在表达式*expr*既不可读又不可写的时候暂停下来。

info watchpoints [n …]

这个命令打印观察点的列表，与info break（参考[5.1.1节，[设置断点]](#_设置断点)）命令的格式一样。

如果需要观察一个数值输入的地址需要对它进行解引用，因为地址本身是一个永远不会改变的常量。GDB拒绝创建一个对永远不变的数值的观察点：

(gdb) watch 0x600850

Cannot watch constant value 0x600850

(gdb) watch \*(int \*)0x600850

Watchpoint 1: \*(int \*)6293584

如果可能的话GDB设置一个硬件断点。硬件断点执行起来非常快，并且调试在精准的在数值改变的地方进行报告。如果GDB不能设置硬件断点，它会设置软件断点，这样执行起来非常慢，而且在下一条语句中（改变已经发生了）报告改变，而不是语句中。

可以使用*set can-use-hw-watchpoints 0*命令强制GDB使用软件观察点。将这个变量设置成0，GDB将永远不会使用硬件观察点，无论底层的系统是否支持硬件观察点。（如果在将can-use-hw-watchpoints设置为0之前设置了hardware-assisted观察点，GDB仍然会使用观察点表达式数值的硬件机制。）

set can-use-hw-watchpoints

设置是否使用硬件观察点。

show can-use-hw-watchpoints

显示当前硬件观察点使用的模式。

对于远程的目标机来说，可以显示GDB使用硬件观察点的数量，参考[set remote hardware-breakpoint-limit]。

当你执行了watch命令，如果可以设置硬件观察点，GDB报告

Hardware watchpoint *num: expr*

当前，awatch和rwatch命令只能设置硬件观察点，因为对观察点的表达式不进行更改的数据访问不能在不检查每个正在执行的指令的情况下被GDB检测到，GDB目前还没有实现。如果GDB检测到不能使用awatch或者rwatch命令设置一个硬件观察点，它会打印如下的信息：

Expression cannot be implemented with read/access watchpoint.

有时，由于观察点表达式的类型长度比目标机能够处理的观察点的长度要长，GDB因此而不能设置观察点。比如，一些系统最多能够观察4bytes宽的区域；在这样的系统上，你不能设置一个表达式的类型为双精度的浮点类型数字（典型的为8bytes宽）的观察点。替代的方式是，将大的区域分为连续的小区域，并且在每个小区域上面分别设置观察点。

如果设置了太多的硬件观察点，GDB在恢复执行程序的时候不能把它们全部都插入进去。因为在程序被恢复执行之前，准确的活动的观察点的数目是不知道的，GDB在你设置观察点的是时候是不能检测到的，所以警告信息会在程序恢复执行的时候被打印出来：

Hardware watchpoint *num*: Could not insert watchpoint

如果这种情况发生了，删除或者禁止一些观察点。

涉及到很多变量的复杂表达式的观察点也会消耗可用的硬件观察点的资源。因为GDB需要观察表达式中的每个独立分配资源的变量。

如果调用使用交的使用print或者call的函数，在GDB到达另一类断点或者调用完成之前你设置的任何观察点都会变成非活跃的。

当局部（automatic）变量，或者包含这种变量的表达式不在作用域的时候，也就是说当执行的位置离开这些变量定义的块的时候，GDB会自动删除它们的观察点。特别的，当调试的程序终止的时候，所有的局部变量都无效了，只有那些观察全局变量的观察点还存在。如果你回到程序，需要再次设置观察点。一种方式是在进入main函数的时候设置代码断点，当到达代码断点的时候，再设置观察点。

在多线程的程序中，观察点会检测每个线程里面观察点表达式的变化。

*警告：*在多线程的程序中，软件观察点只有有限的有效性。如果GDB创建一个软件的观察点，它只在单个的线程中观察表达式的数值。如果你确认表达式只在当前活动的线程中改变（并且确认没有其它的线程会成为当前的），那么可以向平时一样使用软件的观察点。然而，GDB不会由于非当前的活动线程改变表达式而发出通知。（相反，硬件观察点，观察所有线程中的表达式。）

参考[set remote hardware-watchpoint-limit]。

### 设置捕捉点

通过设置捕捉点，可以使程序在某种事件，比如C++的异常或者加载一个共享库的时候暂停下来。使用catch命令设置捕捉点。

catch *event*

在事件*event*发生的时候暂停。事件*event*可以下面之一：

throw [*regexp*]

rethrow [*regexp*]

catch [*regexp*]

C++异常的throwing，re-throwing，或者catching。

如果提供了正则表达式*regexp*，那么只有匹配正则表达式的异常才会被捕获。

在一些系统中，常用变量$\_exception在一个异常的捕捉点是可用的。它保存抛出的异常。

当前在GDB中对处理C++的异常有一些限制：

* 对这些命令的支持是依赖于系统的。当前，只支持系统使用‘gnu-v3’的C++ ABI（参考22.6章，[ABI]）。
* 正则表达式的特性和$\_exption常用变量依赖于一些在libstdc++中的现有的SDT探针。如果这些探针不存在，那么这个特性不能被应用。这些探针在GCC4.8 release版本中首次可用，但是在你的GCC中它们是否可用依赖于它是如何构建的。
* 常用变量$\_exception只有在设置异常相关的捕捉点的时候才可用。
* 当命中了异常相关的捕捉点，GDB暂停在实现C++实时异常支持的系统库，经常为libstdc++。可以使用up命令（参考8.4节，[选择]）到你的代码中。
* 如果调用一个交互的函数，这个函数完成执行之后GDB一般将控制权交给你。如果这个调用引起一个异常，那么会绕过将控制权教过你的步骤，而是使程序abort后者简单得继续执行直到命中一个断点，或者捕捉一个GDB监听的信号，或者退出。这种情况下尽管为异常设置了一个捕捉点；针对异常的捕捉点在交互的调用里面是被禁止的。参考17.5章，[调用]，使用set unwind-on-terminating-exception来控制这种情况。
* 不能抛出一个交互的异常。
* 不能注册一个交互的异常处理函数。

exception

抛出一个Ada的异常。如果在命令最后指定异常的名称（比如catch exception Program\_Error），调试器会在指定的异常抛出的时候暂停。否则，调试器会在抛出任何Ada异常的时候暂停。

当在一个用户定义的异常上面插入一个异常捕捉点的时候，这个用户定义的名称是使用语言的异常定义语句定义的，必须使用完整的授权名称作为异常的名称。否则，GDB会假设在预定义的异常而不是已经定义的异常上面暂停。比如，加入在Pck包中定义一个叫做Constraint\_Error的异常，那么设置捕获此异常的命令为*catch exception Pck.Constraint\_Error*。

exception unhandled

抛出了一个异常但是还没有被程序处理。

assert

一个失败的Ada断言。

exec

对exec的调用。目前只在HP-UX和GNU/Linux上面有效。

syscall

syscall [*name* | *number*] …

对一个系统调用（system call，a.k.a. *syscall*）的调用或者返回。一个系统调用时应用程序向操作系统或者操作系统服务请求一个服务。GDB可以捕获被调试者中的一些或者全部的系统调用，并且显示每个系统调用的相关信息。如果没有指定参数，对每个系统调用的调用和返回都会被捕获。

参数*name*可以是任何底层的操作系统中有效的系统调用的名称。那些系统调用是有效的依赖于操作系统。在GNU和Uinx系统中，可以在‘/usr/include/asm/unistd.h’中找到全部有效的系统调用的名称。

通常情况下，GDB事先知道每个操作系统的有效的系统调用，所以可以使用GDB的命令行的命令补全机制（参考[3.2节，[命令补全]](#_命令补全)）来显示可用的选项的列表。

还可以指定系统调用的数字。系统调用的数字是传递给操作系统的系统调用调度器用以确认请求的服务。当指定系统调用的名称的时候，GDB使用自己的系统调用的数据库将名称转换为响应的数字码，但是当GDB的数据库中没有你的系统中全部的系统调用的列表的时候，直接使用数字是非常有用的（比如，GDB比操作系统的更新要迟很多。）。

下面的例子显示在不提供参数的时候它是如何工作的：

(gdb) catch syscall

Catchpoint 1 (syscall)

(gdb) r

Staring program: /tmp/catch-syscall

Catchpoint 1 (call to syscall ‘close’), \

0xffffe424 in \_\_kernel\_vsyscall()

(gdb) c

Continue.

Catchpoint 1 (returned from syscall ‘close’), \

0xffffe424 in \_\_kernel\_vsyscall()

下面是使用带名称的系统调用的一个例子：

(gdb) catch syscall chroot

Catchpoint 1 (syscall ‘chroot’ [61])

(gdb) r

Starting program: /tmp/catch-syscall

Catchpoint 1 (call to syscall ‘chroot’), \

0xffffe424 in \_\_kernel\_vsyscall()

(gdb) c

Continuing.

Catchpoint 1 (returned from syscall ‘chroot’), \

0xffffe424 in \_\_kernel\_vsyscall()

(gdb)

一个使用系统调用数字的例子。在下面的情况中，系统调用的数字在XML文件中有相应的实体，所以GDB找到他的名称并打印了出来：

(gdb) catch syscall 252

Catchpoint 1 (syscall(s) ‘exi\_group’)

(gdb) r

Starting program: /tmp/catch-syscall

Catchpoint 1 (call to syscall ‘exit\_group’), \

0xffffe424 in \_\_kernel\_vsyscall()

(gdb) c

Continuing.

Program exited normally.

然而，存在在XML文件中没有对应的系统调用的数字的情况。这种情况下，GDB打印一个说明不能找到系统调用名称的警告，但是捕捉点无论如何都会被设置。参考下面的例子：

(gdb) catch syscall 764

Warngin: The number ‘764’ does not represent a known syscall.

Catchpoint 2 (syscall 764)

(gdb)

如果使用‘—without-expat’选项配置了GDB，它不会显示系统调用的名称。并且，如果你的架构中没有描述系统调用的XML文件，你也不会看到系统调用的名称。这两点是用于访问系统调用名称的数据库的。否则，你会看到如下的警告信息：

(gdb) catch syscall

waring: Could not open “syscalls/i386-linux.xml”

waring: Could not load the syscall XML file ‘syscalls/i386-linux.xml’.

GDB will not be able to display syscall names.

Catchpoint 1 (syscall)

(gdb)

当然，这个文件名会依赖你的架构和系统而改变。

仍然使用上面的例子，可以使用系统调用的数字设置捕捉点。在这种情况下，你会看到如似下面的信息：

(gdb) catch syscall 252

Catchpoint 1 (syscall(s) 252)

再次地，这种情况下GDB不能显示系统调用的名称。

fork

调用fork。目前只在HP-UX和GNU/Linux上可用。

vfork

调用vfork。目前只在HP-UX和GNU/Linux上可用。

load [*regexp*]

unload [*regexp*]

在加载和卸载共享库的时候。如果正则表达式*regexp存在，*那么只有在遇到匹配正则表达式的库的捕捉点的时候才会暂停下来。

signal [*signal …* | ‘all’]

在传送一个信号的时候。

如果不携带任何参数，这个捕获点会捕获任何GDB内部不使用的信号，也就是所，除了‘SIGTRAP’和‘SIGINT’之外的所有信号。

使用参数‘all’，所有的信号，包括GDB使用的都会被捕获。这个参数不能和其它信号名称一起使用。

否则，参数是需要处理的信号名称列表（参考5.4节，[信号]）。只有在此列表中的信号才会被捕获。

捕获信号比处理程序更加有用的一点是可以绑定命令和条件到捕获点上面。

当一个信号被捕捉点捕获的时候，信号暂停下来并且打印设置信息，像处理程序是被忽略的。然而，信号是否仍然传递给子级依赖于pass的设置；这个可以在捕捉点的命令中改变。

tcatch *event*

设置一个只暂停一次的捕捉点。这个捕捉点在事件首次被捕获的时候被自动的删除掉。

使用info break命令产看当前的捕捉点列表。

### 删除断点

当断点，观察点或者捕捉点完成了它们的任务并且你不想程序再在它们上面暂停的时候，删除它们是必要的。这就是删除断点。一个被删除的断点不再存在；它被遗忘掉。

使用clear命令可以根据在程序中的位置删除断点。使用delete命令加上断点的数字可以独立地删除断点，观察点或者捕捉点。

没有必要为了越过这个断点继续执行而删除这个断点。GDB在你不改变执行地址而继续执行的时候自动地忽略将要执行的断点的第一条指令。

clear

在选中的栈帧中删除任何下一条将要执行的指令的断点。如果选中了最内的栈帧，这是删除你所暂停处的断点的好方法。

clear *location*

删除指定位置*location*处的任何断点。参考9.2节，[指定位置]来获得位置的变量格式；最常用的是如下方法：

clear *function*

clear *filename:function*

删除在函数名*function*实体里面的任何断点。

clear *linenum*

clear *filename:linenum*

删除由*linenum*指定的处的任何断点。

delete [breakpoints] [*range …*]

删除由参数指定的序列的断点，观察点，或者捕捉点。如果没有任何参数，删除所有断点（GDB会请求确认，除非设置了set confirm off）。这个命令可以简写为d。

### 禁止断点

除了删除断点，观察点，或者捕捉点，还可以禁止它们。这样是断点不起作用，就好像它已经被删除一样，但是请记住断点的信息以便在之后使能它。

使能，禁止断点，观察点，和捕捉点使用enable和disable命令，可选的使用一个或者多个断点数字作为参数。如果你不知道使用那些数字，使用info break命令来打印所有断点，观察点和捕捉点的信息。

禁止和使能含有多个位置的断点会影响所有的位置。

一个断点，观察点或者捕捉点可以有几种不同的使能状态：

* + - Enabled。这个断点使程序暂停。使用break命令设置的断点一开始就在这个状态。
    - Disabled。这个断点对程序没有任何影响。
    - Enabled once。这个断点使你的程序暂停，然后变成disabled的状态。
    - Enabled for a count。这个断点使你的程序在下个N次暂停，然后变成disabled状态。
    - Enabled for deletion。这个断点使你的程序暂停下来，但是之后这个断点被删除。使用tbreak命令设置的断点会进入这个状态。

可以使用如下的命令使能或者禁止断点，观察点和捕捉点。

disable [breakpoints] [*range …*]

禁止指定的断点——或者所有的断点，如果没有列出断点。一个被禁止的断点没有影响但是没有被遗忘。所有的选项比如ignore-counts，conditions和命令一旦这个断点被再次使能就会被记住。这个命令可以简写为dis。

enable [breakpoints] [*range …*]

使能指定的断点（或者所有定义的断点）。它们会再次有效的暂停你的程序。

enable [breakpoints] once *ragne …*

临时的使能指定的断点。GDB在这些断点使程序暂停下来后立即禁止它们。

enable [breakpoints] count *count range …*

临时的使能指定的断点。GDB记录每个指定的断点的计数，当它们命中的时候是计数减一。当计数为0的时候，GDB禁止这个断点。如果一个断点有忽略的计数（参考[5.1.6，[断点条件]](#_断点条件)），在忽略计数减到0之后使能计数才受影响。

enable [breakpoints] delete *range …*

是指定的断点有效一次，然后删除这个断点。GDB在暂停到这个断点上的时候删除这个断点。使用tbreak命令设置的断点一开始进入这个状态。

除了使用tbreak命令设置的断点（参考[5.1.1，[设置断点]](#_设置断点)），被设置的断点的初始状态都是enabled；后来，它们只能通过上面的命令变成disabled或者enabled。（until命令能够设置和删除一个断点本身，但是它不会改变断点的状态；参考[5.2节，[继续和单步执行]](#_继续和单步执行)。）

### 断点的条件

最简单的断点是每次你的程序运行到指定的位置的时候暂停下来。还可以为一个断点指定条件（*condition*）。一个条件就是使用你的程序的语言的布尔表达式（参考10.1节，[表达式]）。带有条件的断点在程序到达此断点时计算表达式，如果表达式的值为true就使程序暂停下来。

这就是使用程序确认的声明的逻辑。在声明的逻辑中，你需要在声明表达式为假的情况下使程序暂停。在C中，如果你需要测试一个声明的表达式*assert*，你需要将指定断点的条件置为‘!*assert*’。

条件对观察点也有效；你可能不要它，因为一个观察点已经在检查一个表达式——但是这个表达式可能很简单，比如仅仅一个变量的名字，还可以在加上一个测试新的数值是否是感兴趣的数值的条件。

断点条件会有副作用，并且可能会在你程序中调用函数。这一点是有用的，比如，激活记录程序进程的程序，或者使用你自己的打印函数来格式化指定的数据结构。这个副作用是完全可预见的，除非在相同的地址中含有另一个断点（在这种情况下，GDB会首先查看另一个断点而不会去检查这个断点的条件。）。breakpoint命令比由于到达一个断点而引起副作用的断点条件经常更加的方便和灵活（参考[5.1.7，[断点命令列表]](#_断点命令列表)）。

如果目标机支持，断点条件可以在目标机一段进行计算。与在本地计算条件相比，GDB独立的将表达式编码为更适合目标机执行的代理表达式（参考附录F，[代理表达式]）。全局变量成为原始的内存位置，局部的变成栈访问的，等等。

在这种情况下，GDB将只在断点的条件变成真的时候被通知断点被触发了。这个机制可能会提供更快的响应时间，依赖于目标机的性能特性，因为它不再使GDB被告知每一个被触发的断点，尤其是是那些条件为假的断点。

断点条件可以在断点设置的时候被指定，使用带‘if’参数的break命令。参考5.1.1节，[设置断点]。它们可以使用condition命令在任何时候被改变。

还可以在watch命令中使用if关键字。catch命令不识别if关键字；condition命令是在一个捕捉点上添加条件的唯一方式。

condition *bnum expression*

为由数字*bnum*指定的断点，观察点，或者捕捉点设置一个指定的条件*expression*。在设置一个条件之后，断点*bnum*只有在表达式*expression*为真（在C中，非零即为真）的时候才会使程序暂停下来。当使用condition命令时，GDB立刻检查表达式*expression*的语法正确性，并且检测里面的符号是否引用了断点的上下文内容。如果表达式使用了不在断点上下文中的符号，GDB打印一个错误信息：

No symbol “foo” in current context.

GDB在执行condition命令的时候并不计算表达式的数值（或者设置断点条件的命令，比如break if …）。参考10.1节，[表达式]。

condition *bnum*

删除数字为*bnum*的断点的条件。使它成为无条件的普通断点。

一种特殊的情况是断点的条件只在断点命中了指定次数之后才会使程序暂停下来。这个应用非常有用，这里有一个专门的命令来实现它，即断点的ignore *count*命令。每个断点都有一个忽略记录，它的类型为整数。大多情况下，忽略计数为0，即没有任何影响。但是如果你的程序到达了一个忽略计数为正的断点，程序不会暂停，只会使这个断点的忽略计数减一然后继续执行。这样，如果忽略计数为n，在程序到达此断点的下n次都不会使程序暂停下来。

ignore *bnum count*

将由*bnum*指定的断点的忽略计数设置为*count*。在这个断点到达的下*count*次，程序不会暂停；而是减少断点的忽略计数，GDB不采取任何行动。

使一个断点在下次到达的时候让程序暂停，将忽略计数*count*置为0。

当使用continue命令使程序从断点处回复执行的时候，可以直接的使用忽略计数作为continue命令的参数。参考[5.2节，[继续和单步执行]](#_继续和单步执行)。

如果一个断点含有正的忽略计数和条件，断点的条件不会被检查。一旦断点的忽略计数变为0，GDB恢复对断点条件的检查。

可以使用condition命令，比如‘$foo-- <= 0’，使用调试器的常用变量的每次递减来实现ignore count命令。参考10.11节，[常用变量]。

忽略计数应用与断点，观察点和捕捉点。

### 断点的命令列表

可以在程序由于任何一个断点（或者观察点，或者捕捉点）而暂停的时候执行一系列的命令。比如，可能想要打印某个表达的的数值，或者使能另一个断点。

Commands [*range …*]

… *command-list …*

end 对指定的短短指定一系列的命令。命令本身在下一行出现。键入包含end的一行来结束命令。

删除一个断点上的命令，输入后面紧跟着end的commands命令，也就说，给这个断点一个空的命令列表。

如果不携带断点参数，这个命令作用在最后设置的一个断点，观察点或者捕捉点（而不是最后一个遇到的断点）上面。如果在最后设置的断点上面设置了单个命令（也就是不携带断点参数），那么commands命令会在所有已经设置的断点上面应用这个命令。这个适用于使用rbreak设置的断点，也适用于单个break命令创建了多个断点的命令（参考10.2[泛指表达式]）。

在命令列表中禁止输入回车会重复执行上一个命令的特性。

可以使用断点的命令列表中的命令来使程序继续执行。简单的使用continue命令，或者step命令，或者其他的可以使程序恢复执行的命令。

在断点的命令列表中，那些使程序恢复执行的命令后面的命令是被忽略的。这是因为在使程序恢复执行后（尽管是简单的next或者step命令），任何时间都可能遇到另一个断点。新的断点可能有它自己的命令列表，这样就不知道应该继续执行那个命令了。

如果在命令列表中的第一个命令是silent，在到达一个断点时候的信息是不打印的。这样可以用来在到达一个断点后打印出自己想要的信息，然后继续执行。如果后面的命令都不打印任何信息，那么在到达这个断点的时候没有任何信息。silent命令只在一个断点的命令列表的开始出有意义。

命令echo，outpu，和printf允许输出的精确控制，在silent的断点上非常有用。参考23.1.4节，[控制输出的命令]。

比如，下面的例子，说明了可以在x是整数的时候，到达foo实体的时候打印出x的数值：

break foo if x > 0

commands

silent

printf “x is %d\n”, x

cont

end

断点的命令列表的另一个应用是临时修复一个bug，继续测试另一个bug。将断点设置在错误代码的下一行，给断点设置一个检测一些错误的发生的条件，之后给这个断点设置一个临时修正错误的命令列表。命令列表的最后一个命令是continue，这样就不会使程序停下来，如果不需要任何输出还需要使用silent命令。下面是一个例子：

break 408

commands

silent

set x = y + 4

cont

end

### 动态Printf

动态的printf命令dprintf，将一个断点和格式化打印程序数据起来，可以在程序中实时的插入printf函数调用，而不用重新编译程序。

在常用的基础模式中，输出到达GDB的控制台。然而，可以通过设置dprintf-style变量实现替换的处理。比如，可以调用程序的printf函数来进行格式输出。这样可以将输出输出到程序的输出设备中，可以进行文件的重定向等等。

如果你使用stub（打桩？？）或者代理进行远程调试，也可以让远程代理处理printf。除了使输出和其它程序可能产生的输出能够到达远程程序的设备中之外，还可以在断开远程目标机的情况下使dprintf依然有效。使用stub/agent经常更加有效，因为它在不与GDB通信的情况下做了一切。

dprintf *location, template, expression* [*, expression …*]

当程序执行到*location*的时候，使用字符串*template*的格式控制打印出一个或者多个表达式*expressions*的数值。打印多个数值，使用逗号分隔。

set dprintf-style *style*

使用下面列举的几个参数来进行dprintf输出的处理。对这个变量的改变立即影响所有已经存在的动态printf。（如果独立的控制print命令，简单的方法是使用携带命令列表的常用断点。）

gdb 使用GDB的printf命令处理输出。

call 使用程序中的函数（经常为printf）处理输出。

agent 让远程的调试代理（比如gdbserver）自己处理输出。只针对支持命令的目标机上面的代理有效。

set dprintf-function *function*

设置调用dprintf函数的调用函数。默认的情况下是printf。可以将它设置为任何GDB可以根据调用的命令进行计算出一个函数的表达式。

set dprintf-channel *channel*

为dprintf设置一个“通道”。如果设置为非空值，GDB将会作为表达式对它进行计算，并且将它作为dprintf函数的第一个参数，就像是按照fprintf相似的函数的调用方式。否则，会按照printf的方式，将格式化字符串作为第一个参数。

举例来说，如果想要把dprintf的输出输出到一个有变量mylog指定的标准I/O流的日志文件，可以如下：

(gdb) set dprintf-function fprintf

(gdb) set dprintf-channel mylog

(gdb) dprintf 25, “at line 25, glob=%d\n”, glob

Dprintf 1 at 0x123456: file main.c line 25.

(gdb) info break

1 dprintf keep y 0x00123456 in main at main.c:25

call (void)fprintf(mylog, “at line 25, glob=%d\n”, glob)

continue

(gdb)

Info命令像正常的断点一样显示动态printf；可以直观的看出对变量设置的效果。

set disconnected-dprintf on

set disconnected-dprintf off

在GDB与目标机断开连接的时候，设置dprintf命令是否依然有效。这个命令只有在dprintf-style是agent的情况下有效。

show disconnected-dprintf off

显示当前断开连接的dprintf的模式。

GDB不会检查函数和通道的有效性，完全依赖于用户在使用的上下文中选取有意义的数值。比如，函数和通道可能是局部变量的数值，如果是这样的话，那么所有的动态打印函数都必须在这个局部范围之内。如果计算失败，GDB会报告一个错误。

### 如何将断点保存到文件

使用save breakpoints命令将已经定义的断点保存到文件中。

save breakpoints [*filename*]

这个命令将所有当前定义的断点，包括它们的命令列表，忽略计数保存到一个名为‘*filename*’的文件中，可能在后续的调试会话中使用。保存的断点包括所有的断点（断点，观察点，追踪点）。读取存贮的已定义断点，使用source命令（参考23.1.3章，[命令文件]）。携带局部变量表达式的观察点可能无法重复创建，因为不可能再访问观察点有效的上下文。由于保存的已定义的断点仅仅是重复创建断点的GDB命令序列，可以在你喜欢的文本编辑器中进行编辑，删除那些不在关心的，或者不能被重建的断点。

### 静态侦测点

GDB在代码中支持SDT（Statically Defined Tracing）探针。SDT指的是静态定义的追踪，探针被设计为具有微小运行时的代码和数据足迹，并且不会动态重定向。它们在汇编，C和C++语言中很有用。参考<http://sourceware.org/systemtap/wiki/UserSpaceProbeImplementation>，了解SDT探针是如何实现的。

当前，SystemTap（<http://sourceware.org/systemtap/）SDT>探针在ELF兼容的系统上面已经支持。参考<http://sourceware.org/systemtap/wiki/AddingUserSpaceProbingToApps>，获得如何向应用中添加SystemTapSDT探针的方法。

一些探针包含相关的信号量变量；比如，这种情况在使用DTrace-style‘.d’文件定义探针的时候自动发生。如果你的探针包含信号量，当使用‘-probe-stap’符号指定一个断点的时候，GDB自动使它使能。但是，如果你使用其它的方式（比如，break file:line）将一个断点置在一个探针的位置，那么GDB不会自动的设置信号量。

可以使用带参数的info probes命令来检查可用的静态探针。

info probes stap [*provider* [*name* [*objfile*]]]

如果存在，*provider*是在选择列出的探针的时候匹配提供者名称的正则表达式。如果忽略，所有提供者的探针都会列举出来。

如果存在，*name*是在悬着列出的探针的时候匹配探针名称的正则表达式。如果忽略，在决定是否显示的时候不会考虑探针的名称。

如果存在，*objfile*是选择检查的目标文件（可执行文件或者共享库）名称的正则表达式。如果忽略，所有的目标文件都会被考虑。

info probes all

列举所有的可用的静态探针。

一个探针可以指定多大12个参数。这在探针定义的时候是可用的，也就是说，在当前的PC指针处在这个探针的位置的时候。可以使用常用变量列举变量（参考10.11节，[常用变量]），$\_probe\_arg0 … $\_probe\_arg11。么个探针参数是适当宽度的整数；类型不被保留。常用变量$\_probe\_argc代表当前的探针点的参数的数目。

这些变量经常是有效的，但是在一个探针点之外的位置访问它们的时候会是GDB给出一个错误信息。

### “无法插入断点”

如果请求了太多的活动的硬件辅助断点和观察点，可以看到如下的错误信息：

Stopped: cannot insert breakpoints.

You may have requested too many hardware breakpoints and watchpoints.

这个信息在恢复程序执行的时候被打印出来，因为只有在这个时候GDB才能精确的知道需要插入多少个硬件断点和观察点。

当打印出这个信息的时候，需要禁止或者删除一些硬件断点和观察点，然后再继续。

### “断点地址调整…”

一些处理器架构在设置断点的位置上有限制。在这样的架构限制下，GDB会尝试调整断点的地址，来遵从架构表述的限制。

这样架构的一个例子是Fujitsu FR-V。FR-V是VLIW（Very long instruction word）结构，肯能会将一些类似RISC（Reduced Instruction-Set Computer）指令绑定起来并行执行。FR-V架构限制断点的地址必须为一些指令中最低地址指令的地址。GDB将断点的地址调整到首个绑定起来的一些指定的第一个指令，这样遵从这个限制。

常用的优化代码会将源代码中不同位置的一些指令绑定在一起，这样的话也会发生将一个断点的位置调整到另一个位置。由于GDB的这个特性会改变用户设置的断点的位置，所以在这样的断点首次设置和命中的时候会打印一个警告信息。

在设置一个断点时候，如果GDB进行了位置调整，会打印如下的警告信息：

warnging: Breakpoint address adjusted from 0x00010414 to 0x00010410.

无论是用户设置的断点还是GDB内部的断点，都会打印这样的信息。如果你看到了这样的一个信息，应该确认在修正后的位置的断点具有想要的影响。如果不是，这个断点应该被删除，另一个具有想要影响的断点应该被设置。比如，在后面的一条指令上面设置断点应可能会有想要的效果。条件断点能够防止这个断点被太频繁的触发。

GDB在程序暂停在一个修正的断点上的时候也会打印一个警告信息：

waring: Breakpoint 1 address previously adjused from 0x00010414 to 0x00010410.

当遇到这样的警告的时候，除了这个断点在预期的位置之前和比预期更频繁的命中的时候，再采取补救措施可能太晚了。

## 继续执行和单步执行

继续执行（continue）意味着恢复程序的执行直到程序正常退出。与之相对应，单步执行（step）意味着仅仅执行程序中的“一步”，“一步”或者代表源代码中的一行，或者代表一条机器指令（依赖于使用的特定指令）。无论是继续执行还是单步执行，程序由于断点或者信号可能很快暂停。（如果由于信号而暂停，可能想要使用处理函数，或者使用‘signal 0’来恢复执行。参考[5.4，[信号]](#_信号)。）

continue [*ignore-count*]

c [*ignore-count*]

fg [*ignore-count*]

在程序上次暂停的地方，恢复程序的执行；在此处设置的任何断点将被跳过。参数*ignore-count*允许指定后续执行时忽略此处的断点的次数；它的影响与ignore相似（参考[5.1.6，[断点条件]](#_断点的条件)）。

参数*ignore-count*只有在程序是由于一个断点而暂停的时候才有意义。其它的情况下，continue的参数被忽略。

同义词c和fg（代表前台，就像被调试的程序被视为前台程序）完全为了方便而提供，它们和continue具有相同的行为。

在不同的地点恢复执行，可以使用return（参考17.4，[从一个函数返回]）回到调用程序；或者jump（参考17.2，[从不同的地址继续执行]）跳转到程序的任意位置。

使用单步执行的典型应用是，在出错的函数或者程序的指定位置处设置一个断点，执行程序，直到到达这个断点，然后单步执行可疑的区域，检查关心的变量，直到找出错误的产生。

step 继续执行程序，直到到达不同的源代码行，然后暂停程序，将控制权交给GDB。这个命令简写为s。

*警告：*如果在一个没有调试信息的函数中执行step命令，直到执行到含有调试信息的函数时才会暂停。同样地，也不会step进入一个没有调试信息的函数。要逐句调试一个没有调试信息的函数，使用stepi命令，下面会描述。

step命令只会暂停在源代码行的第一条指令。这样能够防止在switch语句中的多处暂停，for循环等等。如果一个含有调试信息的函数在此行被调用，step继续暂停。也就是说，step单步进入在此行中调用的函数。

同时，step命令只会进入一个含有行号信息的函数。否则它的行为像next命令。这样在MIPS机器上使用cc –gl时防止出现问题。如前，step进入一个含有调试信息的函数。

step *count*

像使用step一样继续执行，但是这样执行*count*次。如果在执行*count*此之前到达一个断点，或者一个与step不相关的信号，单步执行会立即暂停。

next [*count*]

继续执行直到当前（最内层的）栈帧里面的下一条源代码。这个指令与step相似，但是在代码此行内出现的函数调用被会执行，而不是单步进去。Next命令在当前执行的，原始的栈级别中执行到代码的下一行，然后暂停。这个命令简写为n。

参数*count*是重复的计数，就像step命令一样。

next命令只会暂停在源代码的第一条指令。这样防止在switch语句，for循环等等出发生的多处暂停。

set step-mode

set step-mode on

命令set step-mode on使step命令暂停在一个不包含调试行信息的函数的第一个指令，而不是跳过它。

这个在关心一个没有符号表信息的函数的机器指令，并且不想GDB自动跳过这个函数的情况下非常有用。

set step-mode off

使step命令跳过任何没有调试信息的函数。这是默认的设置。

show step-mode

显示GDB是进入还是跳过没有源代码行调试信息的函数。

finish 继续执行，直到当前栈帧的函数返回。如果存在返回值，打印返回值。这个命令简写为fin。这个命令与return命令的对比（参考17.4，[从一个函数中返回]）。

until

u 继续执行，直到当前栈帧中源代码行超过当前行。这个命令用来避免单步执行一个超过一次的循环。它和next命令很像，但是当until遇到一个跳转的时候，它自动的继续执行，直到程序计数比跳转的地址要大。

这意味着，在单步调试到达一个循环的底部的时候，until使程序继续执行，直到退出循环。然而，在一个循环的底部执行next命令简单的回到循环的开始，然后开始执行下一次迭代。

until命令在想要退出当前的栈帧的时候一般会使程序暂停。

如果机器码的顺序和源代码的顺序不匹配的话，until会产生一些违反直觉的结果。比如，下面来自一段调试会话的引用，f（frame）命令显示执行暂停在206行；当使用until的时候，我们到了195行：

(gdb) f

#0 main(argc=4, argv=0xf7ffae8) at m4.c:206

206 expand\_input();

(gdb) until

195 for ( ; argc > 0; NEXTARG) {

这种情况的发生是由于，为了是执行高效，编译器将循环的退出测试放在了后面，而不是开始处，尽管C的for循环写在循环体的前面。until命令在进入这个表达式的时候好像进入了循环的开始；然而，它并没有真正进入前面的一条语句，并不是按照实际的机器代码。

不带参数的until命令意味着单个指令的前进，所以比带参数的until要慢一些。

until *location*

u *location*

继续执行程序，直到指定的位置到达，或者当前栈帧退出。*location*是在9.2节[指定位置]中描述的任何形式的位置。这种形式的命令使用临时断电，所以比不带参数的until命令要快一些。只有指定的位置在当前的栈帧的时候才会到达。比如下面的代码，如果当前的位置为96行，执行until 99这个命令将使程序暂停在相同调用的factorial函数的99行，在内部的递归调用返回之后。

94 int factorial (int value)

95 {

96 if (value > 1) {

97 value \*= factorial(value – 1);

98 }

99 return (value);

100 }

advance *location*

继续执行程序，直到给定的*location*。需要指定一个参数，参数的形式在9.2节[指定位置]中描述。执行也是也是在基于本栈帧内部，如果退出本栈帧，程序将暂停。这个命令和until命令很相似，但是advance不会跳过递归函数的调用，并且目标位置不必和当前的位置在同一个栈帧中。

stepi

stepi *arg*

si 执行一条机器指令，然后暂停并且回到调试器。

在单步执行机器指令的时候，使用‘display/i $pc’是很有用的。这个指令在程序暂停的时候，使GDB显示下一条将要执行的指令。参考10.7节[自动显示]。

指定的参数为重复的次数，和step命令的参数一样。

nexti

nexit *arg*

ni 执行一条机器指令，但如果为函数调用，那么直到函数返回时才暂停。

指定的参数为重复的次数，和next命令的参数一样。

## 跳过函数和文件

被调试的程序可能包含一些不想要调试的函数。skip命令在调试过程中告诉GDB跳过一个函数或者在一个文件中的所有函数。

比如，下面的C函数：

101 int func()

102 {

103 foo(boring());

104 bar(boring());

105 }

假设我们想要单步进入函数foo和bar，但是不想计入函数boring。如果在103行运行step命令，就会进入boring函数，但是如果运行next命令，那么会跳过foo和boring函数。

一个解决方法是是使用step命令，进入boring函数，然后使用finish命令立即退出它。但是如果boring函数在很多地方被调用，这样做太烦了。

另一个更加灵活的方法是使用*skip boring*。这个命令高速GDB从不单步进入boring函数。现在如果在103行运行step命令，将会跳过boring直接单步进入foo函数。

还可以让GDB跳过在一个文件中的所有函数，比如，skip file boring.c。

skip [*linespec*]

skip function [*linespec*]

运行这个命令之后，通过*linespec*指定的名称的函数，或者包含由*linespec*指定的行号的函数在单步执行的时候会被跳过。参考9.2节[指定位置]。

如果不指定*linespec*，当前调试的函数会被跳过。

（如果希望跳过一个名称为file的函数，使用*skip function file*命令。）

skip file [*filename*]

在运行此命令之后，在源文件*filename*中的任何函数在单步执行的时候都会被跳过。

如果不指定*filename*，正在调试的函数所在的源文件中的所有函数都将会被跳过。

跳过点可以被列出，删除，使能和禁止，和断点很像。下面是管理跳过点的命令：

info skip [*range*]

显示指定的跳过点的详细信息。如果没有指定*range*参数，列出一个关于所有被跳过的函数和文件的详细列表。这个命令列出每个跳过点的下列信息：

*Identifier* 这个跳过点被分配的数字id。

*Type* 跳过点的类型，‘function’或者‘file’。

*Enabled or Disabled*

使能的跳过点标记为‘y’。禁止的跳过点标记为‘n’。

*Address* 对于函数的跳过点，这一列显示被跳过的函数在内存中的地址。如果在一个还没有加载的函数上面设置了跳过点，这个位置上会显示‘<PENDING>’。一旦包含这个函数的共享库被加载，这个函数的地址就会被显示。

*What* 对于文件跳过点，这个位置包含被跳过的文件名。对于函数跳过点，这个位置包含函数的名称和在源文件中的行号。

skip delete [*range*]

删除指定的跳过点（们）。如果没有指定*range*参数，删除所有的跳过点。

skip enable [*range*]

使能指定的跳过点（们）。如果没有指定*range*参数，使能所有的跳过点。

skip disable [*range*]

禁止指定的跳过点（们）。如果没有指定*range*参数，禁止所有的跳过点。

## 信号

信号是在程序中可能发生的异步事件。操作系统定义可能的信号集合，为每个信号分配一个名称和数字。比如，Unix的SIGINT信号是程序在用户键入中断字符（经常为*Ctrl-c*）的时候收到的信号。SIGSEGV是程序在引用远离使用区间的地址时候收到的信号；SIGALRM信号在定时器时间到达的时候产生（只有程序在使用定时器的情况下才会产生）。

一些信号，包括SIGALRM，是程序正常功能的一部分。另外一些，比如SIGSEGV，表明错误；如果程没有提前设置信号的处理回调，这些信号将是致命的（它们会立即杀死程序）。SIGINT信号并不代表程序中的错误，但是它经常是致命的，它的目的是发出中断：用来杀死程序。

GDB能够检测到程序中任何信号的产生。可以提前告知GDB对每种信号执行什么操作。

通常情况下，GDB使非错误的信号，比如SIGALRM，默默地传递给调试的程序（以免干扰它们在程序中的工作），但是当产生一个错误的信号时，会使程序立即停止下来。可以使用handle命令来改变这一设置。

info signals

info handle

打印所有信号的列表，包含GDB对每个信号的处理方式。可以使用这个命令来查看所有已经定义的信号的信号数字。

info signal *sig*

相似的，但是只打印由数字指定的信号的信息。

info handler是info signals命令的别名。

catch signal [*signal…* | ‘all’]

为指定的信号设置一个捕捉点。此命令的详细信息请参考[5.1.3[设置捕捉点]](#_设置捕捉点)。

handle *signal* [*keywords…*]

改变GDB处理信号*signal*的方式。*signal*可以是信号的数字或者信号的名称（带或者不带‘SIG’前缀）；信号列表的形式为‘从低到高’；或者是‘all’，代表所有已知的信号。可选的参数*keywords*，在下面进行描述，代表进行什么样的改变。

Handle命令的keyword可以简写。他们的全名为：

nostop 当信号产生的时候GDB不会使程序暂停。它会打印一条信号来到的消息。

stop 当信号产生的时候GDB使程序暂停。这同时意味着print关键字。

print 当信号产生的时候打印一段信息。

noprint GDB不关系信号是否产生。这同时意味这nostop关键字。

pass

noignore GDB使程序能够感知到这个信号；程序可以处理这个信号，或者如果信号是致命的且不能处理的时候终止进程。pass和noignore是同义词。

nopass

ignore GDB使程序不感知这个信号。nopass和ignore是同义词。

当一个信号使程序暂停，这个信号对程序是不可见的，直到继续执行。如果pass关键词在那个时候起作用，程序就能感知到信号。换句话说，在GDB报告一个信号的时候，可以使用携带pass或者nopass参数的handle命令来控制程序在继续执行的时候是否可见信号。

默认的设置是，针对如SIGALRM，SIGWINCH和SIGCHLD等非错误的信号，设置为nostop，noprint，pass；针对错误的信号，设置为stop，print，pass。

还可以使用signal命令来阻止程序感知到一个信号，或者使它感知到一个它平常不能感知的信号，或者在任何时候传递给它任何信号。比如，如果程序由于一些内存引用的错误而暂停，可以修正错误的变量然后继续执行，预期看到更多的执行；但是程序可能会因为感知到了致命的信号而立即终止。为了防止这样的事情发生，可以使用‘signal 0’来继续执行。参考17.3[给程序一个信号]。

在一些目标机上，GDB可以检查中断信号相关的额外信息，在它传送给被调试的程序之前。这个信息导出到常用变量$\_siginfo中，由内核传递给信号处理程序的数据内容在收到一个信号的时候生成。数据信息的类型本身依赖于目标机器。可以使用ptype $\_siginfo命令查看数据的类型。在Unix系统上面，它典型的对应与标准的siginfo\_t结构，在‘signal.h’系统文件中定义。

下面是一个例子，在GNU/Linux系统上面，打印一个野地址会引发一个段错误。

(gdb) **continue**

Program received signal SIGSEGV, Segmentation fault.

0x0000000000400766 in main()

69 \*(int \*)p = 0;

(gdb) **ptype $\_siginfo**

type = struct {

int si\_signo;

int si\_errno;

int si\_code;

union {

int \_pad[28];

struct {…} \_kill;

struct {…} \_timer;

struct {…} \_rt;

struct {…} \_sigchld;

struct {…} \_sigfault;

struct {…} \_sigpoll;

} \_sifields;

}

(gdb) **ptype $\_siginfo.\_sifields.\_sigfault**

type = struct {

void \*si\_addr;

}

(gdb) **p $\_siginfo.\_sifields.\_sigfault.si\_addr**

$1 = (void \*) 0x7ffff7ff7000

依赖于目标机的支持，$\_siginfo可能是可写的。

## 暂停和开始多线程程序

GDB支持调试多线程的程序（参考[4.10[调试多线程的程序]](#_调试多线程程序)）。在调试器内部有两种控制程序执行的more是。默认的模式为，全暂停模式，当程序中的任何一个线程暂停（比如，到达一个断点或者正在单步执行），程序中的其它线程也会被GDB暂停下来。在一些目标机上，GDB还支持非暂停模式，这种模式下在调试器中检查暂停的线程时，其它的线程可以自由的继续执行。

### 全暂停模式

在全暂停模式下，在GDB控制下的程序由于任何原因而暂停的时候，所有的正在执行的线程都暂停下来，而不只是当前的线程。这样可以检查程序的全部的状态，包括线程之间的转换，而不必担心当下的某些东西被改变。

相反的，当你重新开始程序时，所有的线程开始执行。这个原则即使在适用step或者next命令单步执行的时候依然适用。

特别的，GDB不能步伐一致的单步执行所有线程。因为线程的调度依赖于调试所在的目标机的操作系统（而不是GDB的控制），在当前的线程只执行了一步的时候其它的线程可能执行多于一步。而且，当程序暂停的时候，其它的线程通常会暂停在一个语句的中间，而不是一个语句的边界。

甚至在继续执行或者单步执行的时候发现程序在另一个线程上暂停了。在当前的线程完成指定的操作之前，其它的线程可能会执行到了一个断点，收到一个信号或者抛出一个异常。

在GDB使程序暂停的任何时候，无论是由于断点还是信号，它会自动的选择到遇到断点或者信号发生的线程上面。GDB使用‘[Switching to Thread n]’来通知你进行了上下文的切换。

在一些操作系统中，可以通过锁住操作系统的调度来改变GDB的默认行为，这样允许只运行一个线程。

set scheduler-locking *mode*

设置调度器锁的模式。如果为off的模式，那么将没有锁，并且任何的线程可以在任意的时间被调用。如果为on的状态，只有当前的线程会执行，直到子级消耗完毕。step模式为单步执行而优化；它在单步执行的时候防止其它线程抢占当前的线程，所以调试的重点不会意外的被改变。在单步执行的时候其它的线程很少（不可能）或得运行的机会。它们可能在‘next’过一个函数调用时执行，或者在使用‘continue’，‘until’或者‘finish’的时候完全自由的执行。然而，除非其它的线程在执行时遇到了断点，否则GDB不会将当前正在调试的线程替换掉。

show scheduler-locking

显示当前的调度锁的模式。

默认情况下，当发出了一个执行的命令，比如continue，next或者step，GDB只允许当前线程的子级执行。比如，如果GDB绑定了两个子级，每个子集中有两个线程，continue命令只会回复当前子级中的两个线程的执行。这样是很有用的，比如，当调试的程序调用fork，你想要父进程一直暂停（比如，这样它就不会退出。），同时你调试子程序。在其它的情况下，可能不关心GDB当前绑定的程序感兴趣，可能需要让它们所有都执行起来，直到到达某个断点。在后面的情况中，可以使用set schedule-multiple命令，让GDB允许所有子级的所有线程都执行起来。

set schedule-multiple

设置在运行了执行的命令的时候，多个进程的线程恢复执行的模式。当它为on的时候，所有进程的所有线程线程都会允许执行。当它为off的时候，只有当前的进程的线程们会执行。默认的是off。在设置为on的时候，或者使用step单步执行的时候，scheduler-locking模式优先级更高。

show schedule-multiple

显示当前恢复多个进程的多个线程恢复执行的模式。

### 非暂停模式

对于一些多线程的目标，GDB支持一个可选的操作模式，在这种模式下，可以在调试器中检查暂停的程序线程，同时其它的线程继续自由地执行。在调试实时的系统时，比如在一些有线程的实时限制的程序中，或者必须相应外部事件的程序中，将入侵减到了最小。我们称它为非暂停模式。

在非暂停模式中，当一个线程暂停下来并且报告一个调试的事件时，只有这个线程暂停；与全暂停模式相对比，GDB不会使其它的线程也暂停下来。另外，执行的命令，比如continue和step默认只在非暂停模式下的当前线程上生效，而不是像在全暂停模式中的那样。这样可以显示的控制线程，而在全暂停模式下是做不到的。比如，单步执行一个线程，同时允许其它的线程自由的执行；单步执行一个线程，同时其它的线程全部暂停；或者同时单步执行几个独立的线程。

进入非暂停模式，需要在执行run或者attach命令之前执行如下的命令：

# Enable the async interface.

**set target-async 1**

# If using the CLI, pagination breaks non-stop.

**set pagination off**

# Finlally, turn it on!

**set non-stop on**

可以使用如下的命令来管理非暂停模式的设置：

set no-stop on

使能非暂停模式。

set no-stop off

关闭非暂停模式。

show no-stop

显示当前非暂停模式的设置。

需要注意的是，这些命令只会影响非暂停模式是否使能，而不是当前处在非暂停模式下正在执行的程序。特别地，set non-stop命令只在GDB开始或者在链接目标的时候有效，一旦调试已经开始，通常是不会生效的。进一步来说，由于不是所有的目标都支持非暂停模式，尽管你已经使能了非暂停模式，GDB仍然可能会进入全暂停模式。

在非暂停模式中，所有执行的命令默认情况下只会应用于当期的线程。也就是说，continue命令只会使一个线程继续执行。如果要是所有的线程都继续执行，使用continue –a或者c –a命令。

可以使用GDB的后台执行命令（参考[5.5.3 [后台执行]](#_后台执行)）来在后台运行一些线程，同时继续或者检查，或者单步执行其它的线程。MI执行命令（参考27.15节，[GDB/MI程序执行]）经常在非暂停模式下异步执行。

在后台执行时输入interrupt命令来暂停执行，或者使用*Ctrl-c*。在全暂停模式下，会使整个进程暂停下来；但是在非暂停模式，只在当前的线程中应用interrupt名。如果要使整个进程停止下来，使用interrupt –a命令。

其它的执行命令尚且不支持-a选项。

在非暂停模式下，当一个线程暂停时，GDB不会自动的将这个线程变成当前的线程，就像在全暂停模式中一样。这是因为线程暂停的通知与GDB的命令解释器是异步的，如果GDB在键入一个命令后意外的改变到一个不同的线程上，那么这个指令是作用于哪个线程就会混淆。

### 后台执行

GDB的可执行命令有两个变种：前台（同步）行为，和后台（异步）行为。在前台的执行中，GDB在提示另一个命令之前等待程序报告一些线程已经暂停。在后台的执行中，GDB立即给出命令提示，这样可以在程序运行的时候执行其它命令。

在能够使用后台执行命令前需要显示的使能异步模式，可以使用下面的命令管理异步模式的设置：

set target-async on

使能异步模式。

set target-async off

禁止异步模式。

show target-async

显示当前异步模式的设置。

如果目标机不支持异步模式，GDB会在想要设置后台执行命令的时候发出一个错误信息。

为了指定后台执行，在命令后面加一个‘&’符号。比如，continue命令的后台执行模式为continue&，或者仅仅是c&。可以接受后台执行的命令为：

run 参考[4.2节，[启动程序]](#_启动程序)。

attatch 参考[4.7节，[调试已经运行的程序]](#_调试一个正在运行的程序)。

step 参考[5.2节，[继续执行和单步执行]](#_继续执行和单步执行)。

stepi 参考[5.2节，[继续执行和单步执行]](#_继续执行和单步执行)。

next 参考[5.2节，[继续执行和单步执行]](#_继续执行和单步执行)。

nexti 参考[5.2节，[继续执行和单步执行]](#_继续执行和单步执行)。

continue 参考[5.2节，[继续执行和单步执行]](#_继续执行和单步执行)。

finish 参考[5.2节，[继续执行和单步执行]](#_继续执行和单步执行)。

until 参考[5.2节，[继续执行和单步执行]](#_继续执行和单步执行)。

### 指定线程的断点

当程序包含多个线程的时候（参考[4.10 [调试多线程程序]](#_调试多线程程序)），可以选择是在所有线程上面，或者指定的线程上面设置断点。

break *linespec* thread *threadno*

break *linespec* thread *threadno* if …

*linespec*指定源代码行号；有几种表示方法（参考9.2 [指定位置]），但是有效的经常是指定一些源代码行。

在设置断点的命令中使用‘thread *threadno*’限定符，用来指定在特定的线程到达这个断点的时候使GDB暂停下来。*Threadno*是GDB分配的数字标识，在‘info threads’中的第一列显示的数字。

如果在设置断点的时候没有指定‘thread *threadno*’，这个断点在程序的所有线程上面都有效。

同时还可以使用thread的条件断点；在这种情况下，将‘thread *threadno*’放在条件断点的前面或者后面，比如：

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

### 打断系统调用

不幸的是，在使用GDB吊事多线程的时候会有副作用。如果一个线程在一个断点上暂停，或者由于其它的某种原因，其它的线程阻塞在一个系统调用上面，然后这个系统调用会提前地返回。在GDB使用多线程的交互和信号来实现断点和其它事件来暂停程序的时候会发生这种情况。

为了解决这个问题，你的程序需要检查系统调用的返回值并且作出适当的反应。这也是优秀的变成风格。

比如，不要这样写程序：

**sleep (10);**

调用sleep会在另一个不同的线程在一个断点上暂停或者由于其它原因会提前返回。替代的，这样写：

int unslept = 10;

while (unslept > 0)

unslept = sleep(unslept);

系统调用允许提前返回，这样系统依然遵从它的规格。但是GDB会导致多线程的程序和不带GDB的程序运行不一致。

同时，GDB使用库的内置断点来某些事件，比如线程创建和线程销毁。当到达这样的一个事件时，其它线程的系统调用会立即返回，尽管你的程序好像并没有暂停。

### 观察者模式

如果想要建立非暂停模式，并且不让GDB破坏程序而观察程序的行为，可以将变量设置为禁止调试器试图改变的模式，无论是写内存，或者插入断点等等。这个操作在底层，会拦截所有命令的操作。

当所有这些都被关闭的时候，GDB进入了观察者模式。作为一个方便，变量**observer**可以被设置来禁止这些，加上使能非暂停模式。

GDB不会阻止你进行一些荒谬的组合设置。比如，如果你使能了**may-insert-breakpoints**但是禁止了**may-write-memory**，那么通过代码六中的写陷阱指令而工作的断点将不会被设置。

set observer on

set observer off

当设置为on的时候，禁止下面所有的变量的许可（除了**insert-fast-tracepoints**），加上使能非暂停调试。设置为off的时候回到正常的设置，虽然还在非暂停模式。

show observer

显示观察者模式是开启还是关闭。

set may-wirte-registers on

set may-write-registers off

用来控制是否允许GDB改变寄存器的值，比如使用带表达式的print，或者jump命令。默认的是no。

show may-write-registers

显示当前对写寄存器的限制。

set may-write-memory on

set may-write-memory off

用来控制是否允许GDB改变内存的值，比如通过携带表达式的print命令。默认的为on。

show may-write-memory

显示当前对写内存的限制。

set may-insert-breakpoints on

set may-insert-breakpoints off

用来控制是否允许GDB插入断点。它会影响所有的断点，包括GDB的内部断点。默认的为on。

show may-insert-breakpoints

显示当前对插入断点的限制。

set may-insert-tracepoints on

set may-insert-tracepoints off

用来控制是否允许GDB在追踪实验的开始处插入（正常的）追踪点。它只影响非快速追踪点，快速追踪点使用**may-insert-fast-tracepoints**来控制。默认的为on。

show may-insert-tracepoints

显示当前对于插入追踪点的限制。

set may-insert-fast-tracepoints on

set may-insert-fast-tracepoints off

用来控制是否允许GDB在追踪实验的开始处插入快速追踪点。它只影响快速追踪点，正常的（非快速的）追踪点由**may-insert-tracepoints**来控制。默认的为on。

show may-insert-fast-tracepoints

显示当前对于插入快速追踪点的限制。

set may-interrupt on

set may-interrupt off

用来控制是否允许GDB打断或者暂停程序的执行。当这个变量设置为off是，interrupt命令不会起作用，包括*Ctrl-c*。默认为on。

show may-interrupt

显示当前对于打断或者暂停程序的限制。

# 反向执行程序

在调试一个程序的时候，经常会发现已经走的太远了，一些插入的事件已经发生过了。如果模板环境支持，GDB允许通过反向执行来使程序“回绕”。

支持反向执行的目标环境应该能够像程序顺序执行一样“收回”机器状态已经发生的改变。变量，寄存器等等应该回到之前的数值。显然这要求在目标环境上有许多复杂的技术；不是所有的目标环境都支持反向执行。

当程序反向执行的时候，经常被使用的指令是“un-executed”，以反向的顺序。在“un-executed”中的灭个指令，内存和寄存器的值回到这个指令之前的状态。在反向执行一段源代码后，这段代码的副作用都会失效，所有的变量应该回到之前的状态。

如果下载支持反向执行的目标环境中调试程序，GDB提供如下的命令。

reverse-continue [*ignore-count*]

rc [*ignore-count*]

从程序上次暂停的位置开始，开始反向执行。反向执行会在断点和同步异常（信号）处暂停，就像正常执行一样。异步信号的行为依赖于目标机环境。

reverse-step [*count*]

反向执行程序，直到到达不同的源代码行；然后暂停，将控制权交给GDB。

像step命令一样，reverse-stop只会在源代码行的开始处暂停。它“不执行”之前已经执行过的源代码行。如果之前的源代码行包含对调试函数的调用，reverse-step会单步（返回）到调用的函数，在调用的函数的最后一条语句的开始处暂停（典型的是一个return语句）。

同时，就像step命令，如果非调试的函数被调用，reverse-stop不会暂停而是执行过它们。

reverse-stepi [*count*]

反向执行一条机器指令。在反向执行中的指令并不是由程序计数器指定的一条，而是之前执行过的一条指令。比如，如果最后一条指令是jump，那么reverse-stepi指令会使你回到jump指令本身这条指令。

reverse-next [*count*]

返回到之前执行的在当前（最内层）栈帧中的一行的开始处。如果此行包含函数调用，会“不执行”而不是暂停。在一个函数的第一行处，reverse-next会回到调用此函数的位置，在函数调用之前，就像正常的next命令会使你在函数的最后一行回到函数的调用者一样。

reverse-nexti [*count*]

像nexti指令，reverse-nexti反向执行一个指令，除了被调用的函数自动得“不执行”。也就是说，如果之前执行的指令是从另一个函数返回，reverse-nexti会反向执行到这个函数被调用处（在当前的栈帧中）。

reverse-finish

就像finish命令会将你带到当前的函数返回的点，reverse-finish将你带到函数调用的点。不是在当前函数调用的结尾处结束，而是在开始处结束。

set exec-direction

这是目标执行的方向。

set exec-direction reverse

GDB会反向执行所有的命令，知道exec-direction模式回到“向前（forwward）”。受影响的命令包括step，stepi，next，nexti，continue，和finish。return命令不能再反向执行模式中使用。

set exec-direction forward

GDB以正常的方式执行命令。这是默认的模式。

# 记录子级的执行和重放

在一些平台上，GDB提供一个特殊的进程记录和重放功能，它可以记录进程执行的日志，在后面将正向的和反向的执行命令重放。

当在目标上使用它的时候，如果执行的日志包含对next指令的记录，GDB会在重放模式中调试。在重放模式中，子级并不真正的执行代码指令。而是，在代码执行过程中所有发生的事件都从执行日志中获得。虽然代码在重放模式下并不真正的执行，但是寄存器的数值（包括程序计数器寄存器）和子级的内容仍然会像正常执行的那样改变。它们的内容从执行日志中获得。

如果next指令的记录中不在执行日志中包含，GDB会在记录模式下调试。在这个模式下，子级正常的执行，同时GDB为未来的重放会记录执行日志。

进程的记录和重放目标支持反向执行（参考第[6章 [反向执行]](#_反向执行程序)），尽管子级支持的平台不支持反向执行。然而，反向执行只限于在执行日志中记录的范围。也就是说，在不支持反向执行的平台的上面只有通过重放模式来达到反向执行。

当在反向模式调试的时候，只要在执行日志中包含之前指令的记录，GDB就会进入回放模式；否则，如果平台支持反向执行，它进入记录模式，如果不支持，GDB会停止。

在支持进程记录和重放的架构环境中，GDB提供下面的命令：

record *method*

这个命令开启程序记录和重放模式。记录的方式作为参数。如果不带参数，此命令使用全记录方式。支持下面的记录方式:

full 全记录/重放使用GDB的软件记录和重放功能。这个方式允许重放和反向执行。

btrace 硬件支持的指令记录。这个方式不支持重放和反向执行。

这个记录方式可能不是在所有的处理器上都有效。

进程记录和重放功能指定调试一个已经运行的程序。也就说，你需要先使用**run**或者**start**命令启动程序，然后使用**record mothod**命令开始记录。

**record method**和**rec method**都是**target record-method**命令的别名。置换的单步（参考附录D [置换的单步执行]）在进程记录和重放开始的时候会自动的被禁止。这是因为进程记录和重放功能不支持置换的单步。

record stop

暂停进程的记录和重放功能。当进程的记录和重放功能暂停时，整个执行日志会被删除，并且子级会停止，或者停留在最终的状态。

当在记录的模式下暂停进程的记录和重放的功能时（在执行日志的最后），暂停在下一条指令的子级会被记录。换句话说，如果你记录了一段时间，然后暂停记录，子级会停留在好像记录从来没有发生的状态。

另一方面，如果在重放的模式下暂停进程的记录和重放功能（也就是，不在执行日志的最后，而是前面的某一点），子级进程会“活”在之前的状态，并且可以从这个状态继续调试。

当子级进程退出的时候，或者GDB从它分离，进程记录和重放功能会自动的暂停。

record save *filename*

将执行日志保存到文件‘*filename*’中。默认的文件名是‘gdb\_record.process\_id’，process\_id是子级的进程ID。这个命令不是对所有的记录方法都有效。

record restore *filename*

从文件‘*filename*’中恢复执行日志。文件必须是由record save创建的。

set record full insn-number-max *limit*

set record full insn-number-max unlimited

设置在全记录方式下记录的指令的限制。默认的为200000。

如果*limit*是一个正数，那么当记录的指令大于*limit* 的时候，GDB开始从日志中删除指令。对于每个新的指令，GDB会删除最早的指令，以保持记录的指令数目维持在限制之内。（由于删除记录的指令会导致信息丢失，可以在到达限制时通过**stop-at-limit**选项来进行设置如何去做，下面会进行描述。）

如果*limit*是unlimited或者为0，GDB将不会从执行日志中删除指令。记录指令的数目只会收到可用存储的限制。

show record full insn-number-max

显示在全记录模式中记录指令的限制值。

set record full stop-at-limit

当记录的指令数目到达限制值时，控制全记录模式的行为。如果为ON（默认情况），GDB在首次到达限制值的时候会暂停下来，然后询问你是想要暂停子级还是继续运行且记录执行日志。如果你要继续记录，每个新的指令都会导致最老的指令被删除。如果选项为OFF，GDB会自动的为每个新的指令通过删除老的指令来腾出空间，不会发出询问。

show record full stop-at-limit

显示当前**stop-at-limit**的设置。

set record full memory-query

当GDB不能记录由全记录模式下的指令导致的内存改变时，控制GDB的行为。如果为ON，GDB会询问是否要暂停子级。

如果选项为OFF（默认的），GDB会自动的忽略这样的指令对内存的影响。之后，当GDB重放执行日志的时候，它会将这条指令的日志标记为不可访问，不会影响重放的结果。

show record full memory-query

显示当前的**memory-query**的设置。

info record

显示基于当前记录方式的统计信息：

full 对于全记录模式，显示进程记录的状态和在内存中的执行日志buffer，包括：

* 是在记录模式还是重放模式。
* 最低的记录的指令的数目（从当前的可执行日志开始记录指令开始）。
* 最高的记录的指令的数目。
* 将要重放的当前指令（如果在重放模式下）。
* 在执行日志中包含的指令的数目。
* 在执行日志中可以包含的最大的指令的数目。

btrace 在btrace记录模式下，显示已经记录的指令的数目和由记录指令形成的连续的控制流组成的块的数目。

record delete

如果目标机运行在重放模式，删除后续的执行日志并且从当前地址开始记录新的日志。也就是说，将要删除之前的“未来”部分而重新录制新的“未来”部分。

record instruction-history

从记录的执行日志中反汇编指令。默认的，十个指令被反汇编。可以通过**set record instruction-history-size**来控制大小。指令以执行的顺序被打印。下面有几种指定执行日志的哪部分需要进行汇编的方式：

record instruction-history *insn*

从指令数字*insn*开始反汇编十条指令。

record instruction-history *insn, +/-n*

在指令数字*insn*范围内反汇编*n*条指令。如果*n*有前缀+，反汇编指令数字*insn*后*n*条指令。如果*n*有前缀-，反汇编指令数字*insn*前*n*条指令。

record instruction-history

在上次反汇编的指令之后再反汇编十条指令。

record instruction-history -

在上次反汇编的指令之前再反汇编十条指令。

Record instruction-history *begin end*

反汇编从指令数字*begin*开始到*end*的指令。指令数字*end*的指令将不被包含。

这个指令不是对所有的记录模式都有效。

set record instruction-history-size *size*

set record instruction-history-size unlimited

设置在**record instruction-history**指令中汇编多少条指令。默认的值是10。Unlimited意味着无限制的指令。

show record instruction-history-size

显示在**record instruction-history**指令中汇编多少条指令。

record function-call-history

打印在函数的粒度上的历史。它打印包含在同一个函数内的所有指令为一行，通过函数名进行显示，指令序列的行号（如果使用 /l 修饰符），和指令数字组成的序列（如果指定 /i 修饰符）。

(gdb) **list 1, 10**

1. void foo (void)
2. {
3. }
4. void bar (void)
5. {
6. …
7. foo();
8. …
9. }

(gdb) **record function-call-history /l**

1 foo.c:6-8 bar

2 foo.c:2-3 foo

3 foo.c:9-10 bar

默认情况下，会打印出10行。可以通过**set record function-call-history-size**来改变。函数以执行的顺序打印。有几种控制打印信息的方式：

record function-call-history *func*

从函数*func*开始打印10个函数。

record function-call-history *func, +/- n*

在函数*func*周围打印*n*个函数。如果*n*包含+前缀，在函数*func*之后打印*n*个函数。如果*n*包含-前缀，在函数*func*之前打印*n*个函数。

record function-call-history

在上次打印的十行之后再打印十行函数。

record function-call-history -

在上次打印的十行之前在打印十行函数。

record function-call-history *begin end*

打印从函数数字*begin*到*end*的函数。函数数字*end*不包含。

这个指令不是对所有的记录方式都有效。

set record function-call-history-size *size*

set record function-call-history-size unlimited

定义在**record function-call-history**中打印多少行。默认的值是10。Unlimited意味着无限制。

show record function-call-history-size

显示在**record function-call-hisoty**命令中打印多少行。

# 检查栈

当程序暂停的时候，首先要知道的事情是暂停的地点和程序是如何到达这里的。

程序每次调用函数时，就会生成关于调用的信息。这个信息包括调用在程序中的位置，调用的参数，和被调用的函数的局部变量。这个信息保存在一块成为栈帧（*stack frame*）的结构里面。栈帧在称为调用栈（*call stack*）的存储区上面进行分配空间。

当程序暂停的时候，GDB提供的检查栈的命令允许你查看所有的这些信息。

GDB选择一个栈帧，并且许多的GDB命令都与这个被选择的栈帧相关。特别的，无论何时通过GDB查看程序中变量的数值的时候，GDB在当前选中的栈帧中查找。有一些特别的关于选取感兴趣的栈的GDB命令。参考8.4 [选择一个栈]。

当程序暂停时，GDB自动的选择当前的正在执行的栈，并且显示它的简要信息，与frame命令很像（参考8.5节[栈的信息]）。

## 栈帧

调用栈被分为连续的称为栈帧，或者简称为帧的片段；每个栈帧包含对一个函数调用的相关信息。包括传递给函数的参数，函数的局部变量和函数执行的地址。

当程序开始的时候，栈上面只有一个帧，也就是main函数的帧。它叫做初始帧，或者最外层的帧。每次函数调用的时候，生成一个新的帧。每次函数返回的时候，这个函数调用的帧结束。如果函数是递归的，存在一个函数的多个帧。正在执行的函数的帧称为最内层的帧，它是所有现存的栈帧中最后一个生成的帧。

在程序内部，栈帧通过它们的地址来标识。一个栈帧包含很多拥有自己的地址的byte；每种类型的计算机都有将一个byte的地址作为帧的地址的约定。这个地址通常保存在寄存器里面，称为栈帧指针寄存器（*frame pointer register*）（参考10.13[寄存器]），同时执行在此帧中进行。

GDB为每个存在的栈帧分配数字，从最内层的帧的数字为0开始，调用这个栈帧的栈帧数字为1，这样增加下去。这些数字在程序中实际上并不存在；GDB赋予它们这些数值只是为了在GDB命令中标明栈帧。

一些编译器提供一种编译函数的方式，让函数不使用栈帧。（比如，GCC选项‘-fomit-frame-point’产生不带栈帧的函数）。偶尔会在大量使用的库函数中为了节省栈帧建立时间而使用。GDB在这种函数上面有一些限制。如果最内层的调用函数没有栈帧，GDB仍然将它看为含有独立的栈帧，像平常一样为它分配数字0，允许追溯调用栈链。然而，GDB不会为在栈的其它地方的不含有栈帧的函数提供这个功能。

frame *args*

frame命令允许你从一个栈帧移动到另一个栈帧，并且打印出选择的栈帧。*args*参数可以是栈帧的地址，或者栈帧的数字。如果不带参数，此命令将打印当前栈帧的信息。

select-frame

select-frame命令允许从一个栈帧移动到另一个栈帧，而不打印出栈帧。它是安静版本的frame。

## 回溯

回溯是程序如何到达此处的总结信息。它为每个栈帧显示一行，对于多个栈帧，从当前的正在执行的栈帧（栈帧0）开始，接着是它的调用者（栈帧1），如此继续下去。

backtrace

bt 打印整个栈的回溯信息；栈中的每个帧占用一行。

可以在任何时候键入系统中断字符来停止回溯，通常为***Ctrl-c***。

backtrace *n*

bt *n* 与上面相似，但是只显示最内层的*n*个帧。

backtrace *–n*

bt *–n* 与上面相似，但是只显示最外层的*n*个帧。

backtrace full

bt full

bt full *n*

bt full ­*-n*

同时打印出局部变量。参数*n*指定打印帧的数目，就像上面提到的。

backtrace no-filters

bt no-filters

bt no-filters *n*

bt no-filters ­­*–n*

bt no-filters full

bt no-filters full *n*

bt no-filters full *–n*

在回溯中不运行Python帧过滤。参考23.2.2.9 [帧过滤API]，获得更多信息。另外使用[disable filter all]关闭所有的帧过滤。这一项只有在GDB配置为Python支持的时候才有效。

名称where和info stack（简写为info s）是backtrace的别名。

在多线程程序中，GDB默认的只显示当前线程的回溯信息。如果要显示一些或者所有线程的回溯信息，使用命令thread apply（参考[4.10节，[线程]](#_调试多线程程序)）。例如，如果键入**thread apply all backtrace**，GDB会显示所有线程的回溯信息；这在调试多线程程序的core dump时是很方便的。

回溯信息的没一行显示栈帧的数字和函数名称。程序计数器的数值也显示出来——除非设置了set print address off。回溯信息还包括源文件名称和行号，还有函数的参数。如果在这一行的代码开始处，程序计数器的数值不显示。

下面是一个回溯信息的例子。它使用命令‘bt 3’，所以显示最内层的三个栈帧。

#0 m4\_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)

at builtin.c:993

#1 0x6e38 in expand\_macro (sym=0x2b600, data=…) at macro.c:242

#2 0x6840 in expand\_token (obs=0x0, t=177664, td=0xf7fffb08)

at macro.c:71

(More stack frames follow…)

在frame 0开始显示的内容中并不以程序计数器数值开始，表明程序暂停在文件builton.c的993行的开始处。参数data的数值我用…进行了代替，默认情况下，GDB只打印参数是标量（整数，指针，枚举等）的参数内容。参考***set print frame-arguments***，在10.8节，[打印设置]，来或得配置如何打印参数的更加详细的信息。

如果程序在编译过程中使用了优化选项，一些编译器会优化掉传递给函数但从不使用的参数。这样的优化产生从寄存器传递参数的代码，而不是在栈帧中存储参数。GDB不可能显示除了最内层的其它栈帧上面的参数信息。下面是这样的栈帧显示的示例：

#0 m4\_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)

at builtin.c:993

#1 0x6e38 in expand\_macro (sys=<optimized out>) at macro.c:242

#2 0x6840 in expand\_token (obs=0x0, t=<optimized out>, td=0xf7ff7b08)

(More stack frames follow …)

不在栈帧中保存的参数的数值显示为‘<optimized out>’。

如果你需要显示优化掉的参数的数值，或者通过它依赖的其它变量推到出来，或者重新使用不带优化选项的编译。

很多程序有一个标准的用户进入点——系统库和启动代码从这一点进入用户代码。对于C语言来说是main函数。当GDB找到进入点函数的时候结束回溯，这样避免回溯到高级的系统指定的代码（经常是不关心的）。

如果需要检查启动代码，或者限制回溯的级别数字，可以改变这个行为：

set backtrace past-main

set backtrace past-main on

回溯会在到达用户进入点的时候继续进行。

set backtrace past-main off

回溯会在到达用户进入点的时候停止。这是默认的设置。

show backtrace past-main

显示当前的基于用户进入点的回溯设置。

set backtrace past-entry

set backtrace past-entry on

回溯会在通过应用的内置进入点的时候继续。这个进入点在应用简历的时候由编译器编码，并且可能早于（或者等于）用户进入点main被调用的时候。

set backtrace past-entry

回溯会在遇到应用的内置的进入点的时候暂停。这是默认的设置。

show backtrace past-entry

显示当前的基于应用的内置进入点的回溯设置。

set backtrace limit *n*

set backtrace limit 0

set backtrace limit unlimited

限制回溯的级别为*n*。数值0和unlimited意味这无限制的级别。

show backtrace limit

显示当前回溯级别的限制。

可以控制如何显示文件名。

set filename-display

set filename-display relative

基于编辑的目录显示文件名。这是默认的设置。

set filename-display basename

只显示文件名的基础名称。

set filename-display absolute

显示绝对的文件名。

show filename-display

显示当前的显示文件名的方式。

## 管理帧过滤

帧过滤是一个基于Python的工具集，用来管理和装饰帧的输出。更多信息参考23.3.3.9节，[帧过滤API]。

管理帧过滤是通过GDB内置的几个命令来实现的，它们如下。

info frame-filter

打印所有从字典中安装的帧过滤，显示它们的名字，优先级和使能状态。

disable frame-filter *filter-dictionary filter-name*

禁止在有*filter-dictionary*匹配的字典，或者所有的字典中的由*filter-name*指定的帧过滤。过滤字典可为all，global，progspace或者帧过滤字典所在的位置的字典名称。当指定为all的时候，所有字典中的帧过滤都被禁止。*Filter-name*是帧过滤的名称，当*filter-dictorionary*不是all的时候使用它。被禁止的帧过滤并不是被删除，它还可以在后面被重新使能。

enable frame-filter *filter-dictorionary filter-name*

使能在有*filter-dictionary*匹配的字典，或者所有的字典中的由*filter-name*指定的帧过滤。过滤字典可为all，global，progspace或者帧过滤字典所在的位置的字典名称。当指定为all的时候，所有字典中的帧过滤都被使能。*Filter-name*是帧过滤的名称，当*filter-dictorionary*不是all的时候使用它。

比如：

(gdb) **info frame-filter**

global frame-filters:

Priority Enabled Name

1000 No PrimaryFunctionFilter

100 Yes Reverse

progspace /build/test frame-filters:

Priority Enabled Name

100 Yes ProgspaceFilter

objfile /build/test frame-filters:

Priority Enabled Name

999 Yes BuildProgramFilter

(gdb) **disable frame-filter /build/test BuildProgramFilter**

(gdb) **info frame-filter**

global frame-filters:

Priority Enabled Name

1000 No PrimaryFunctionFilter

100 Yes Reverse

progspace /build/test frame-filters:

Priority Enabled Name

100 Yes ProgspaceFilter

objfile /build/test frame-filters:

Priority Enabled Name

999 No BuildProgramFilter

(gdb) **enable frame-filter global PrimaryFunctionFilter**

(gdb) **info frame-filter**

global frame-filters:

Priority Enabled Name

1000 Yes PrimaryFunctionFilter

100 Yes Reverse

progspace /build/test frame-filters:

Priority Enabled Name

100 Yes ProgspaceFilter

objfile /build/test frame-filters:

Priority Enabled Name

999 No BuildProgramFilter

show frame-filter priority *filter-dictionary filter-name*

显示匹配*filter-dictionary*的字典，匹配*filter-name*名称的过滤的优先级。参数*filter-dictionary*可以是global，progspace或者帧过滤字典所在的位置的字典名称。

例如：

(gdb) **info frame-filter**

global frame-filters:

Priority Enabled Name

1000 Yes PrimaryFunctionFilter

100 Yes Reverse

progspace /build/test frame-filters:

Priority Enabled Name

100 Yes ProgspaceFilter

objfile /build/test frame-filters:

Priority Enabled Name

999 No BuildProgramFilter

(gdb) **set frame-filter priority global Reverse 50**

(gdb) **info frame-filter**

global frame-filters:

Priority Enabled Name

1000 Yes PrimaryFunctionFilter

50 Yes Reverse

progspace /build/test frame-filters:

Priority Enabled Name

100 Yes ProgspaceFilter

objfile /build/test frame-filters:

Priority Enabled Name

999 No BuildProgramFilter

## 选择一个帧

很多检查栈和程序工作的其它信息基于当时所选择的栈帧。下面是关于选择栈帧的命令；它们都已打印一个剪短的关于选择的栈帧的信息而结束。

frame *n*

f *n* 选择数字为*n*的栈帧。再次调用frame 0是最内层的（当前正在执行的）栈帧，frame 1是调用最内层栈帧的栈帧，如此下去。最大数字的栈帧是函数main的栈帧。

frame *addr*

f *addr* 选择在地址*addr*处的栈帧。主要的应用是如果栈帧链被一个bug破坏了，那么GDB不可能对所有的栈帧分配数字的时候。另外，在程序有多个栈，在这些栈之间切换的时候也很有用。

在SPARC架构上面，选择一个任意的栈帧需要两个地址：一个帧指针和一个栈指针。

在MIPS和Alpha架构上面，需要两个地址：一个栈指针和一个程序计数器。

在29k架构上面，需要三个地址：一个寄存器栈指针，一个程序计数器，和一个内存栈指针。

up *n* 在栈上向上移动*n*个帧。如果*n*为正数，方向向帧的外面，也就是更大数字的帧，存在时间更长的帧。*n*默认值是1。

down *n* 在栈上向下系统*n*个帧。如果*n*为正数，方向向帧的里面，也就是更小数字的帧，最新生成的帧。*n*默认值是1。down可以简写为do。

这些命令都以打印两行描述帧的信息而结束。第一行显示帧的数字，函数名称，参数，和执行此栈帧的源文件和行号。第二行显示源代码行的文本。

比如：

(gdb) **up**

#1 0x22f0 in main (argc=1, argv=0xf7fffbf4, env=0xf7fffbfc) at env.c:10

10 read\_input\_file(argv[i]);

在打印完此信息后，不带参数的list命令显示以此为中心的10行文本。输入edit，还可以使用你喜欢的编辑器进行编辑。参考9.1节，[打印源代码行]。

up-silently *n*

down-silently *n*

这两个命令分别是up和down命令的变种；不同的是它们安静的工作，不会显示新的帧的信息。它们主要用于GDB的命令脚本，在脚本里面输出是不必要的。

## 帧的信息

有一些显示选中的栈帧的一些信息的命令。

frame

f 当不带参数时，这个命令不会改变选中的栈帧，二是打印当前栈帧的简要描述信息。可以简写为f。如果带参数，这个命令用来选择栈帧，参考[8.4节，[选择栈帧]](#_选择一个帧)。

info frame

info f 此命令打印选中栈帧的详细信息，包括：

帧的地址

* 下一个帧的地址（这个帧调用的帧）
* 上一个帧的地址（调用这个帧的帧）
* 和此帧相对应的源代码的语言
* 帧的参数的地址
* 帧的局部变量的地址
* 帧保存的程序计数器（调用函数的执行地址）
* 帧中保存的寄存器

详细的描述信息在出错导致栈的格式和常用的惯例不匹配的时候很有用。

info frame *addr*

info f *addr*

打印在*addr*地址处的栈帧的详细信息，而不是选择栈帧。已经选择的栈帧不会因为此命令而改变。这里的参数和指定栈帧的命令frame的参数是一样的。参考[8.4节，[选择栈帧]](#_选择一个帧)。

# 检查源代码

GDB能够部分打印程序的源代码，因为程序中的调试信息（GCC编译时加‘-g’选项）告诉GDB编译程序时使用的源文件。在程序暂停的时候，GDB自动的打印暂停处的行。同样的，当选择了一个栈帧（参考[8.4节，[选择栈帧]](#_选择一个帧)）的时候，GDB打印在此栈帧中暂停的位置。可以使用显示的命令打印源代码的其它部分。

如果是通过GNU的Emacs接口调用GDB，可以使用Emacs来查看源代码；参考[26章，[在GNU Emacs下使用GDB]](#_在GNU_Emacs下使用GDB)。

## 打印源代码行

打印源代码，使用list命令（简写为l）。默认情况下，会打印10行。有一些指定打印文件什么位置的方法，参考[9.2节，[指定位置]](#_指定位置)。

下面是list命令的常用形式：

list *linenum*

打印以行号*linenum*为中心的的几行源文件。

list *function*

打印以函数*function*的开始为中心的几行源文件。

list 打印更多的行。如果上一行是使用list命令打印的，会接着上次打印的信息继续打印；如果上一行是显示栈帧的信息（参考[第8章，[检查栈]](#_检查栈)），会以这一行为中心进行打印。

list - 打印上次打印的行之前的信息。

默认情况下，无论是那种list的形式，GDB打印行源代码。可以通过set listsize来改变这一行为：

set listsize *count*

set listsieze unlimited

使list命令显示*count*行源代码（除非list的参数显示的指定了其它的数字）。*count*为unlimited或者0代表不进行限制。

show listsize

显示list命令打印的行数。

使用RET来进行重复list命令会丢掉命令的参数，也就是等于仅仅使用list。这样比显示相同的一行要实用的多。一个例外是使用参数‘-’；这个参数是保存重复的，在源文件中每次重复动作。

通常，list命令期望得到0个，一个或者两个*linespec*参数。*Linespec*指定源代码行；它有几种表示方法（参考[9.2节，[指定位置]](#_指定位置)），经常是指定一些源代码行。

下面是list命令的参数的完整描述：

list *linespec*

打印由*linespec*指定的行为中心的几行。

list *first, last*

打印从*first*到*last*的行。两个参数都是linespec。当list命令含有两个linespec，并且第二个linespec中省略了源文件，代表和第一个参数使用同一个源文件。

list *, last*

打印以*last*结尾的源代码行。

list *first,*

打印以*first*开始的源代码行。

list + 打印上次打印的行后面的行。

list - 打印上次打印的行之前的行。

list 见上表中的描述。

## 指定位置

一些命令的参数是指定程序代码中的位置。由于GDB是源代码级的调试器，位置通常指的是源代码中的某行；因此，位置就是常常说的指定的行（*linespec*）。

下面是GDB所能理解的所有不同的指定代码位置的方式：

*linenum* 指定当前源文件中的行号为*linenum*的行。

*-offset*

*+offset* 指定基于当前行的偏移行。对于list命令来说，当前的行就是最后以此打印的行；对于断点命令来说，当前行就是在当前的栈帧中执行暂停的行（参考[8.1节，[帧]](#_栈帧)）。当在list命令中作为第二个参数的时候，它特指基于第一个参数的上或者下的偏移。

*filename：linenum*

指定源代码文件*filename*中的*linenum*行。如果*filename*是相对的名字，它将匹配所有相同尾部元素相同的源文件。比如，如果*filename*是‘gcc/expr.c’，那么它将匹配‘/build/trunk/gcc/expr.c’的源代码文件名，而不是‘/build/trunk/libcpp/expr.c’或者‘/build/trunk/gcc/x-expr.c’。

*function* 指定*function*函数的函数体开始的地方。比如，在C代码中，就是以左大括弧开始的行。

*function:label*

指定函数*function*中的*label*标签的那一行。

*filename:function*

指定在原文件*fialename*中函数*function*的函数体开始的那行。使用文件名加函数名的方式是为了在不同的文件中有同名的函数的时候避免含糊不清。

*label* 指定标签*label*出现的那一行。GDB在当前选中的栈帧中的函数中搜索这个标签。如果当前没有已经选择的栈帧（比如，子级现在没有运行），那么GDB不会进行查找标签。

*\*address* 指定程序的*address*地址处。对于面向行的指令，比如list和edit，这个指定包含地址*address*的源代码行。对于break和其它面向断点的命令，这个用于设置程序中没有调试信息或者源代码信息部分的断点。

在这里，*address*可以为当前工作语言（参考[15章，[语言]](#_用不同的语言使用GDB)）的有效表达式指定的地址。另外，为了方便，GDB扩展了位置中的表达式的语义来包含在调试中经常发生的情况。下面是*address*的变量形式：

*expression*

使用当前语言的有效的表达式。

*funcaddr*

用名字来获得的函数或者程序的地址。在C，C++，Java，Objective-C，Fortran，minimal，和汇编中，就是函数的名称*function*（实际上就是一种特别的有效表达式）。在Pascal和Modula-2中，它是&*function*。在Ada中，它是函数的地址（尽管Pascal的形式也是可以的）。

这种形式指明了函数首个指令的地址，在栈帧和参数建立起来之前。

*‘filename’::funcaddr*

和上面的*funcaddr*很相似，但是明确指定源代码文件。这在仅由一个函数名不能确认具体函数的时候很有用，比如，多个文件中有同名的函数。

-pstap | -probe-stap [*objfile:*[*provider:*]]*name*

GNU/Linux的SystemTap工具提供向应用中嵌入静态探针的方式。参考[5.1.10，[静态侦测点]](#_静态侦测点)。这种形式指定一个静态探针的位置。

如果给定了*objfile*，只有正则匹配这个*objfile*的共享库或者可执行文件才会进行查找。如果给定了*provider*参数，只有来自这个提供者的探针才会进行查找。如果多个探针匹配了这个位置，GDB会在每个探针中都加入断点。

## 编辑源文件

编辑源代码文件的某一行，使用edit命令。你选择的编辑程序会被调用，将当前行设置为活动的行。如果想要看到程序的其它部分，有几种指定文件的那一部分需要打印的方式：

edit *location*

编辑源文件的指定的*location*行。从*location*处开始编辑，也就是源文件的指定行。参数*location*的形式，参考[9.2节，[指定位置]](#_指定位置)；下面是edit命令常用的形式：

edit *number*

通过指定的行号来编辑文件的某一行。

edit *function*

通过指定函数名来编辑文件的某一行。

### 选择编辑器

可以通过GDB来定制使用的文件编辑器。默认情况下，为‘/bin/ex’，但是可以在使用GDB之前通过设置环境变量EDITOR来进行改变。比如，配置GDB使用vi编辑器，需要在sh shell中使用下面的命令：

EDITOR=/usr/bin/vi

export EDITOR

gdb …

或者在csh shell中，

setenv EDITOR /usr/bin/vi

gdb …

## 查找源文件

有两种使用正则表达式在当前源文件中进行查找的命令。

forward-search *regexp*

search *regexp*

‘forward-search *regexp*’命令使用*regexp*进行匹配，自上次的列表行中从前到后检查每一行。列出找到匹配到的行。还可以使用‘search *regexp*’或者命令的简写fo。

reverse-search *regexp*

‘reverse-search *regexp*’命令使用*regexp*进行匹配，自上次的列表行中从后到前检查每一行。列出找到匹配到的行。可以使用命令的简写rev。

## 指定源代码目录

可执行文件有时不会记录编译时使用的源文件的目录，而只是文件名。即使它记录了目录，在编译后到使用GDB进行调试的期间目录也可能被移动。GDB含有查找源文件路径的列表；它被称为源路径（*source path*）。当GDB需要源代码的时候，它在这个列表中进行顺序查找，直到找到指定名称的文件。

比如，假设可执行文件的源文件为‘/usr/src/foo-1.0/lib/foo.c’，并且源路径为‘/mnt/cross’。首先，按照可执行文件中的引用路径进行逐字的查找；如果失败，会尝试‘/mnt/cross/usr/src/foo-1.0/lib/foo.c’；如果失败，会尝试‘/mnt/cross/foo.c’；如果失败，打印出一个错误信息。GDB不会查找文件的部分名称，比如‘/mnt/cross/src/foo-1.0/lib/foo.c’。同时，源路径的子目录也不会进行递归查找：如果源路径为‘/mnt/cross’，并且二进制文件的引用为‘foo.c’，GDB不会在‘/mnt/cross/usr/src/foo-1.0/lib’下进行查找。

明文的文件名，具有前导目录的相对文件明，包含点号的文件名等等都按照上面的描述进行处理；比如，如果源路径为‘/mnt/cross’，（可执行文件中记录的）源代码为‘../lib/foo.c’，GDB首先尝试‘../lib/foo.c’，然后为‘/mnt/cross/../lib/foo.c’，之后为‘/mnt/cross/foo.c’。

可执行文件的查找路径不会用来进行源代码文件的定位。

当重置或者重新整理源路径的时候，GDB会清除它缓存的关于源代码文件查找到的路径和文件的行信息等所有信息。

查找的路径用来查找程序的源文件和GDB的脚本文件（使用‘source’命令的‘-command’选项进行读取）。

除了源路径，GDB提供一系列用来管理一些源路径的替代规则的命令。一个替代规则指明了如果源代码在编译和调试期间被移动到了不同的目录如何在程序的调试信息上面重写源代码目录。一个规则包含两条字符串，第一个指明在路径中什么需要被重写，第二个指明如何被重写。在[set substitute-path]中，我们将它们分别称为来（from）和去（to）。GDB在源代码文件名的前一部分进行简单的用‘去’替换‘来’的字符串，然后使用替换后的替代原来的进行源代码的查找。

使用前面的例子，加入‘foo-1.0’的目录树从‘/usr/src’被移动到了‘/mnt/cross’，那么可以告知GDB在所有的源路径名称中使用‘/mnt/cross’来替换‘/usr/src’。首次查找会变成‘/mnt/cross/foo-1.0/lib/foo.c’，而不是‘/usr/src/foo-1.0/lib/foo.c’。定义一个源路径的替代规则，使用set substitute-path命令（参考[set substitute-path]）。

为了避免意想不到的替代规则，只有目录名称的‘来’的部分以目录分隔符为结尾的规则才被应用。比如，一个将‘/usr/source’替换为‘/mnt/cross’的替换规则会应用于‘/usr/source/foo-1.0’，但不会应用于‘/usr/sourceware/foo-2.0’。并且由于替代规则只应用于目录名称的开始部分，这个规则不会被应用与‘/root/usr/source/baz.c’。

在许多条件下，可以通过directory命令达到相同的结果。然而，set substitute-path可以在源代码的组织是一个复杂的树的条件下更加高效。使用directory命令，必须将每个子目录都加入到你的工程之中。如果你移动整个目录并且保留它的内部组织结构，那么set substitute-path允许只使用一个命令就直接进行调试所有的源文件。

set substitute-path不仅是一个快速有效的命令。源路径只有在原始的位置不存在的时候才会被应用。也就是说，set substitute-path命令修改调试器的行为为查找重写的位置。所以，如果为了某种原因，源代码与可执行程序中原始的定位不相符，一个替代规则是唯一的改变GDB指向新的位置的方式。

可以通过‘—with-relocated-sources=*dir*’选项来配置默认的源代码替代规则。*dir*应该为在GDB的配置前缀目录（通过‘--prefix’或者‘—exec-prefix’进行设置）下的目录，当安装的GDB被移动到新的位置的时候在*dir*下调试信息的目录名称会自动调整。在GDB，库或者包含调试信息的可执行文件和相应的源代码整体移动到新的位置的时候很有用。

directory *dirname …*

dir *dirname …*

将目录*dirname*添加到源路径的前面。此命令可以给定多个目录，使用‘：’（在MS-DOS和MS-Windows中使用‘；’，‘：’经常作为绝对文件名的一部分）或空格进行分隔。还可以指定已经在源路径中的目录；这样将这个目录提前，GDB可以更快的查找到它。

可以使用字符串‘$cdir’来引用编译的目录（如果被记录），‘$cwd’引用当前的工作目录。‘$cwd’和‘.’不一样，前者在GDB的回话中追踪当前工作目录的改变，后者在添加一个实体到源路径的时候立即展开为当前的目录。

directory

重置源路径为它的默认值（在Unix系统上为‘$cdir:$cwd’）。这个命令需要进行确认。

set directories *path-list*

将源路径设置为*path-list*。如果忽略了‘$cdir:$cwd’，会自动加上。

show directories

打印源路径，显示它包含的目录。

set substitute-path *from to*

定义一个路径替换规则，并且将它加入到当前已经存在的替换规则列表的后面。如果新规则的*from*参数已经在旧的规则中存在，那么旧的规则将被替换。

比如，如果文件‘/foo/bar/baz.c’被移动到了‘/mnt/cross/baz.c’，那么命令

(gdb) set substitute-path /usr/src /mnt/cross

会高速GDB使用‘/mnt/cross’替换‘/usr/src’，这样尽管文件‘baz.c’被移动了，GDB仍然能够找到它。

在定义了多个替换规则的时候，那么会按照它们定义的顺序一条一条的计算。第一个匹配到的（如果存在），就按照它的规则进行替换。

比如，如果我们键入了下面的命令：

(gdb) set substitute-path /usr/src/include /mnt/include

(gdb) set substitute-path /usr/src /mnt/src

GDB会按照第一条规则将‘/usr/src/include /defs.h’重写为‘/mnt/include/defs.h’。然而，它将应用第二条规则将‘/usr/src/lib/foo.c’替换为‘/mnt/src/lib/foo.c’。

unset substitute-path [path]

如果指定了path，查找当前的替换规则的列表中匹配要重写的目录的项。如果找到，将此项规则删掉。如果没有找到任何规则将会发出警告。

如果没有指定path，那么所有的替换规则都将被删除。

show substitute-path [path]

如果指定了path，那么打印出匹配path的替换规则。如果没有指定path，那么打印出所有存在的源路径的替换规则。

如果元路径中堆满了不在关心的目录，GDB可能有时会由于错误的源代码版本而导致混乱。可以使用下面的方式进行修正。

1. 使用不带参数的directory命令来重置源路径为默认值。
2. 使用携带正确参数的directory命令来重新定义源路径的目录。可以在一个命令中加入全部的路径。

## 源代码和机器码

可以使用命令info line来将源代码行映射到程序地址（反之亦然），命令disassemble用来显示一段地址的机器指令。可以使用命令set disassemble-next-line在程序暂停的时候是否反汇编下一行的代码。在GNU Emacs模式下运行的时候，命令info line使箭头指向指定的行。并且，info line以16进制打印出符号表中的地址。

info line *linespec*

打印出源代码行*linespec*的开始和结束地址出的编译代码。可以通过[9.2节[指定位置]](#_指定位置)的形式指明源代码行。

比如，我们可以使用info line来查看可重定位代码中函数m4\_changeqoute的第一行：

(gdb) info line m4\_changeqoute

Line 895 of “builtin.c” starts at pc 0x634c and ends at 0x6350.

还可以询问（使用*\*addr*作为*linespec*的形式）源代码包含特定的地址：

(gdb) **info line \*0x63ff**

Line 926 of “builtin.c” starts at pc 0x63e4 and ends at 6404.

在info line之后，x命令的默认地址被改变为此行的开始地址，所以‘x/i’命令足可以开始检查机器代码（参考10.6节 [检查内存]）。并且，这个地址被保存在常用变量$\_里面（参考10.11节 [常用变量]）。

disassemble

disassemble /m

disassemble /r

这个命令将一段地址展示为机器指令。它可以通过指定/m参数打印混合的源代码+汇编，通过指定/r参数打印原始指令在符号表中的十六进制信息。默认的内存范围是程序计数器中的当前栈帧中的函数。此命令的单个参数是程序计数器的数值；GDB将会显示此值附近的函数。当给定两个参数的时候，参数应该通过逗号进行分隔，可以使用空白在前后填充。这两个参数指定地址的范围，可以为下面的形式：

*start, end*

地址范围是从*start*（包含）到*end*（不包含）。

*start, +length*

地址范围是从*start*（包含）到*start+length*（不包含）。

当指定两个参数的时候，函数的名称也会答应出来（因为在给定的范围内可能有多个函数）。

参数可以是产生一个数值的任何表达式，比如‘0x32c4’，‘&main+10’或者‘$pc - 8’。

如果反汇编的内存范围包含当前的程序计数器，在这个位置的指令会使用=>进行标记。

下面的例子显示了在HP PA-RISC 2.0代码中反汇编一段地址：

(gdb) **disas 0x32c4, 0x32e4**

Dump of assembler code from 0x32c4 to 0x32e4

0x32c4 <main+204>: addil 0, dp

0x32c8 <main+208>: ldw 0x22c(str0, r1), r26

0x32cc <main+212>: ldil 0x3000, r31

0x32d0 <main+216>: ble 0x3f8(sr4, r31)

0x32d4 <main+220>: ldo 0(r31), rp

0x32d8 <main+224>: addil -0x800, dp

0x32dc <main+228>: ldo 0x588(r1), r26

0x32e0 <main+232>: ldil 0x3000, r31

End of assermber dump.

下面的例子是在Intel x86上的混合源代码+汇编，程序在函数开始处暂停：

(gdb) **disas /m main**

Dump of assembler code for function main:

5 {

0x08048330 <+0>: push %ebp

0x08048330 <+1>: mov %esp, %ebp

0x08048330 <+3>: sub $0x8, %esp

0x08048330 <+6>: and $0xfffffff0, %esp

0x08048330 <+9>: sub $0x10, %esp

6 printf(“Hello.\n”);

=> 0x08048330 <+12>: movl $0x8048440, (%esp)

0x08048330 <+19>: call 0x8048284 <puts@plt>

7 return 0;

8 }

0x08048330 <+24>: mov $0x0, %eax

0x08048330 <+29>: leave

0x08048330 <+30>: ret

End of assembler dump.

下面的例子显示的是在AMDx86-64上的原始的十六进制指令：

(gdb) **disas /r 0x400281, +10**

Dump of assembler code from 0x400281 to 0x40028b:

0x0000000000400281: 38 36 cmp %dh, ($rsi)

0x0000000000400283: 2d 36 34 2e 73 sub $0x732e3436, %eax

0x0000000000400288: 6f outsl %ds: (%rsi), (%dx)

0x0000000000400289: 2e 32 00 xor %cs: (%rax), %al

End of assembler dump.

地址不能指定为linespec的形式（参考[9.2节，[指定地址]](#_指定位置)）。所以，比如，如果想要反汇编文件‘foo.c’中的函数bar，必须键入‘disassemble ‘foo.c’::bar’而不是‘disassemble foo.c:bar’。

一些架构有多于一个常用集合的指令助记符或者其它语法。

使用动态链接，用于共享库的程序，调用函数或者分支的指令在共享库中的位置看起来像是假的位置——它实际是重定向表的地址。在一些架构上，GDB可以解析这些为实际的函数名称。

set disassembly-flavor *instruction-set*

选取在使用disassemble或者x/i命令时使用的反汇编指令集。

当前这个命令只在x86家族中定义了。可以将指令集设为intel或者att。默认的是att，Unix在基于x86架构上所使用的默认的AT&T风格。

show disassembly-flavor

显示当前的反汇编风格的设置。

set disassemble-next-line

show disassemble-next-line

控制GDB在执行暂停的时候是否反汇编下一行源代码。如果为ON，GDB在被调试的程序暂停的时候会反汇编下一条源代码行。这个会附加到GDB默认的**显示源代码行的后面。如果由于某些原因源代码行不能显示（比如，如果GDB**不能找到源文件，或者在调试信息中不包含行信息），GDB会反汇编下一行而不显示下一行的源代码。如果为AUTO，GDB只有在不能显示源代码行的时候才会反汇编下一行。这个设置使你在单步调试不带行信息或者源文件不可用的函数的时候使GDB显示一些反馈信息。默认的为OFF，意味着永远不会反汇编下一行。

# 检查数据

在程序中检查数据的常用方式是使用print命令（简写为p），或者它的同义词inspect。它使用程序的代码语言计算并且打印表达式的值（参考15章 [用不同的语言使用GDB]）。还可以使用基于Python的优美打印方式打印结果（参考[10.9节 [优美打印]](#_优美打印)）。

print *expr*

print */f* *expr*

*expr*是一个表达式（使用源代码的语言）。默认情况下表达式的值以它相应的类型打印出来；可以通过‘/f’选项改变打印的格式，其中，f是指定格式的字符；参考[10.5节 [输出格式]](#_输出格式)。

print

print */f*

如果省略了表达式*expr*，GDB会再次显示最近一次的数值（从数值的历史中；参考[10.10节 [数值历史]](#_数值历史)）。这可以方便的使用不同的格式检查相同的数值。

更加底层的检测数据的方式是使用x命令。它检查内存中指定地址的数据并且按照指定的格式进行打印。参考[10.6节 [检查内存]](#_检查内存)。

如果关系类型的信息，或者一个结构体中的成员是如何声明的，使用ptype *exp*命令而不是print。参考第16章 [检查符号表]。

另一种检查表达式的数值和类型信息的方式是通过Python扩展的命令explore（只有在编译GDB的时候加上—with-python选项才可用）。它从表达式的数据类型的最高层（或者说，最抽象的层次）开始，向下探寻内嵌的叶子标量的数值，并且提供交互的界面。

explore *arg*

*arg*是一个表达式（以源代码的语言），或者在当前被调试的程序的上下文中是可见的一个类型。

explore命令的工作方式可以用一个例子阐明。如果在C程序中一个类型为struct ComplexStruct的结构定义为：

struct SimpleStruct

{

int i;

double d;

};

struct ComplexStruct

{

struct SimpleStruct \*ss\_p;

int arr[10];

}

紧接着变量声明为

struct SimpleStruct ss = { 10, 1.11 };

struct ComplexStruct cs = { &ss, { 1, 2, 3, 4, 5, 6, 7, 8, 9 } };

然后，变量cs可以使用如下的explore命令探测数值。

(gdb) **explore cs**

The value of ‘cs’ is a struct/class of type ‘struct ComplexStruct’ with the following fields:

ss\_p = <Enter 0 to explore this field of type ‘struct SimpleStruct \*’>

attr = <Enter 1 to explore this field of type ‘int [10]’>

Enter the field number of choice:

由于cs的att成员并不是标量，被提示选择需要探测的成员。比如通过键入0选择了成员ss\_p。然后，由于这个成员是一个指针，你会被问道它是否指向一个单个数值。从上面的cs的声明中看到，它确实指向一个单个数值，所以键入y。如果键入n，那么你会被问道它是否指向一个数组，这样这个成员会当做数组进行解释。

‘cs.ss\_p’ is a pointer to a value of type ‘struct SimpleStruct’

Continue exploring it as a pointer to a single value [y/n]: **y**

The value of ‘\*(cs.ss\_p)’ is a struct/class of type ‘struct

SimpleStruct’ with the following fields:

i = 10 .. (Value of type ‘int’)

d = 1.1100000000000001 .. (Value of type ‘double’)

Press enter to return to parent value:

如果之前通过键入1选择了cs的arr成员，那么由于它是一个数组，你会被提示输入数组的下标来确定这个值。

‘cs.arr’ is an array of ‘int’.

Enter the index of the element you want to explore in ‘cs.arr’: **5**

‘(cs.arr)[5]’ is a scalar value of type ‘int’.

(cs.arr)[5] = 4

Press enter to return parent value:

通常，在探索数值的任何时期，可以根据提示更深一步的探索叶子成员，或者直接加入回车回到包裹的数据结构（更高一层的数据结构）。

与探索数值相似，可以使用explore命令来探索类型。不是指定一个数值（在当前被调试程序的上下文中有效的变量名或者表达式），而是通过指定一个名称。那上面的例子来说，可以通过传入参数为struct ComplexStruct的explore命令来探索它的类型。

(gdb) **explore struct ComplexStruct**

通过在下面出现的相应提示信息，可以按照上面的显示cs的数值的方式探索类型struct ComplexStruct。

explore命令有两个子命令，explore value和explore type。第一个子命令显示的指明需要探索的数值，后面的子命令指明需要探索的类型。

explore value *expr*

这个explore的子命令探索表达式*expr*的数值（*expr*是在当前被调试的程序的上下文中有效的表达式）。这个命令的行为与使用携带参数*expr*的explore命令的行为相似。

explore type *arg*

这个explore的子命令探索*arg*的类型（*arg*是在当前被调试的程序的上下文中有效的类型），或者数值/表达式*arg*的数值的类型（*arg*是在当前被调试的程序的上下文中有效的表达式）。如果*arg*是类型，那么此命令的行为与携带参数*arg*的explore的命令相似。如果*arg*是一个表达式，那么此命令的行为和携带*arg*的类型为参数的explore的命令相似。

## 表达式

print命令和许多其它的GDB命令接受一个表达式的参数，并且计算表达式的数值。任何在使用的语言中定义的常量，变量或者操作符在GDB的表达式中使用都是有效的。这些包括条件表达式，函数调用，强制转换和字符串常量。如果编译程序时包含了宏的信息，那么还包含预处理宏；参考[4.1节 [编译]](#_编译调试信息)。

GDB在用户输入时接受数组常量。语法为{*element, element …*}。比如，可以使用命令print {1, 2, 3}产生三个整数的数组。如果将一个数组传给一个函数或者将它赋值给一个程序变量，GDB会将数组的内存值拷贝到目标程序中。

由于C的广泛应用，本手册中的大部分例子都使用C语言。参考第15章 [用不同的语言使用GDB]，来了解如何用其它语言来使用表达式。

在本章中，我们讨论可以在GDB表达式中可以使用的操作符，而不关心程序语言。

所有的语言中都支持强转，而不仅是在C中，因为这在将一个成员强转为一个地址而查看此地址的内存的时候非常有用。

GDB除了支持通用的语言中的操作付，还支持这些操作符：

@ ‘@’是二进制操作符，将内存的一部分转换为数组。详细信息参考[10.4节[人造数组]](#_人造数组)。

:: ‘::’用来指定一个变量是在文件或者函数的位置中被定义的。参考[10.3节 [程序变量]](#_程序变量)。

{*type*} *addr* 引用一个在内存地址为*addr*处的类型为*type*的对象。*addr*可以是值为整数或者指针的表达式（但是二进制操作符需要由括号包裹，就像是在强转）。无论在地址*addr*的数据是什么类型，这种操作都是有效的。

## 模糊表达式

表达式有时可以包含一些模糊的元素。比如，一些程序语言（尤其是Ada，C++和Objective-C）允许一个函数名由于不同的上下文应用而被多次定义。这叫做重载（*overloading*）。另一个包含在Ada中的例子是泛型（generics）。一个泛型包和C++的模板相似，典型的初始化多次，导致相同的函数名在不同的上下文被多次定义。

依赖于语言的一些情况下，可以修正表达式以消除模糊性。比如在C++中，可以指定希望添加断点的函数的签名，比如break function(types)。在Ada中，使用函数的全部的限制名称也行消除模糊性。

当检测到需要解析一个模糊表达式的时候，GDB能够列出一个可选择的数字列表，之后显示提示符‘>’，等待输入。第一个选项一直是‘[0] cancle’，键入0 回车退出当前的命令。如果使用的命令的表达式有多个选项，下一个选项是‘[1] all’，键入 1 回车选择所有可选的选项。

比如，下面的回话尝试在重载函数String::after上面设置一个断点。我们选择了三个特定的函数名定义：

(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)

set multiple-symbols *mode*

这个选项调整当遇到模糊表达式的时候调试器的行为。

默认情况下，*mode*设置为all。如果遇到多于一个选项的模糊表达式，那么GDB自动的选择所有的选项。比如，在一个模糊表达式所代表的函数上面添加一个断点会导致在每个函数上都添加一个断点。然而，如果必须做一个特殊的选择，那么GDB使用菜单来帮助你排除表达式的歧义。比如，打印一个重载函数的地址会使用到这个菜单。

当*mode*设置为ask的时候，调试器在遇到模糊表达式的时候始终使用菜单。

最后，当*mode*设置为cancel的时候，调试器在遇到一个模糊表达式的时候会报错并且退出。

show multiple-symbols

显示当前的multiple-symbols的设置。

## 程序变量

最常用的表达式的形式就是使用程序中的变量名。

表达式中的变量只在选择的栈帧（参考[8.4节[选择栈帧]](#_选择一个帧)）中进行理解；它们必须为：

* global（或者文件中的静态类型）
* 在当前执行的栈帧中通过程序语言的作用域是可见的。

这意味这在这个函数中

foo (a) int a;

{

bar (a);

{

int b = test();

bar (b);

}

}

咱函数foo的内部执行时，都可以检查和使用变量a，但是只能在b声明的代码块中使用变量b。

有一个例外：可以引用作用域在一个源文件中的变量和函数，尽管当前的执行点不在这个文件中。但是同一个名称可能有多个变量或者函数（在不同的源文件中）。如果这样做了，引用这个名称会有意想不到的结果。如果你愿意，可以使用双冒号（::）来指定静态变量所属的函数或者文件：

file::variable

function::variable

这里的file和function是静态变量variable的上下文。在使用文件名的时候，可以将文件名用单引号包含起来用以确保GDB将它看成一个文件名——比如，打印在文件‘f2.c’中的全局变量x：

(gdb) **p ‘f2.c’::x**

双冒号（::）标记经常用来引用静态变量，因为你经常通过选取适当的栈帧和简单的变量名区分函数的局部变量。然而，还可以使用这个标记来引用包含当前栈帧的帧里面的局部变量。

void foo (int a)

{

if (a < 0)

bar (a);

else

process (a); /\* Stop here \*/

}

int bar (int a)

{

foo (a +5);

}

比如，在注释的一行有一个断点，在执行bar(0)的过程中的情形如下：

(gdb) **p a**

$1 = 10

(gdb) **p bar::a**

$2 = 5

(gdb) **up 2**

#2 0x080483d0 in foo(a=5) at foobar.c: 12

(gdb) **p a**

$3 = 5

(gdb) **p bar::a**

$4 = 0

这种关于‘::’的用法与在C++中的此符号的用法很少存在冲突。GDB也支持在GDB的表达式中使用C++的域操作付。

*警告：*偶尔，一个局部变量在函数中的某一点会有错误的数值——在进入一个新的作用域，和在没有退出之前。

在单步调试机器指令的时候可能会遇到这个问题。这是由于，在大多数的机器上，建立一个栈帧需要多于一条指令（包括局部变量的声明）；如果你在单步调试机器指令，在栈帧完全建立起来之前变量可能会呈现错误的值。在退出的时候，销毁一个栈帧也需要多于一条的指令；如果单步进入了这样的一组指令中，局部变量的定义可能已经消失了。

这在编译器进行一些重要优化的时候也可能发生。为了能够保证一直能够真实的看到数值，在编译的时候将优化关闭。

编译器优化的另一个可能的影响是将没有使用的变量删除，或者将变量赋值给寄存器（与其相反的是内存地址）。依赖于这种情况发生时编译器提供的调试信息，GDB可能不能显示这种变量的数值，GDB会打印如下的信息：

No symbol “foo” in current context.

要解决这些问题，去掉优化重新编译，或者如果编译器致贺词多个格式就使用不同的调试信息格式。参考[4.1节[编译]](#_编译调试信息)，获得关于编译选项的更多信息。参考15.4.1节[C和C++]，获得最适合C++的调试信息的格式。

如果打印一个对于GDB来说内容是未知的对象，由于它的数据类型不能有调试信息来确认，GDB会打印‘<incomplete type>’。参考第[16章[符号]](#_检查符号表)。

如果在变量后边加上@entry字符串，那么会得到当刚调用函数时这个变量的数值。如果这个变量不可用会打印出错误信息。进入时的数值只有在一些编译器中可用。进入时的数值也经常在通过[set print entry-values]设置函数参数列表中打印出来。

Breakpoint 1, d (i = 30) at gdb.base/entry-value.c:29

29 i++

(gdb) **next**

30 e (i);

(gdb) **print i**

$1 = 31

(gdb) **print i@entry**

$2 = 30

字符串被认为是没有指定符号的char类型的数组。signed char和unsigned char的数组当成1byte大小的整数数组来打印。GCC的-fsigned-char或者-funsigned-char选项对于GDB定义文字字符串类型为无符号的char没有任何影响。在程序代码中：

char var0[] = “A”;

signed char var1[] = “A”;

在调试的时候会得到如下的信息：

(gdb) print var0

$1 = “A”

(gdb) print var1

$2 = {65, ‘A’, 0, ‘\0’}

## 人造数组

打印出内存中连续的相同类型的对象有时是非常有用的；一个数组的一段，或者在程序中仅存在一个指针的动态大小成员的数组。

可以通过人造数组来引用连续跨度的内存，使用二进制操作符‘@’。‘@’操作符的左边的操作数为所需求数组的第一个独立的对象元素。右边的操作数是需求的数组的长度。结果是一个所有成员都是左边操作数类型的数组。第一个元素就是左边的操作数；第二个元素来源于紧随于第一个元素的内存地址的内存，如此继续。下面是一个例子。如果一个程序中写到

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

可以通过下面打印出数组array的内容

p \*array@len

‘@’符号左边的操作数必须子啊内存中存在。使用‘@’按这种方式组成的数组就像脚本形式的数组，并在表达式中强制的使用指针。人造数组经常在打印出一个之后，通过数值历史（参考[10.10节 [数值历史]](#_数值历史)）出现在表达式中。

(gdb) p/x (short[2])0x12345678

$1 = {0x1234, 0x5678}

为了方便，如果不写出数组长度（写为‘(type[])value’）GDB计算数值的大小（通过‘sizeof(value)/sizeof(type)’）来填充这个位置：

(gdb) p/x (short[])0x12345678

$2 = {0x1234, 0x5678}

有时人造数组的机制不是十分充足；在适当复杂的数据结构中，所关心的元素可能不是连续的——比如，如果你关心数组中的指针。这种情况下使用常用变量（参考[10.11节 [常用变量]](#_常用变量)）作为表达式的计数器打印出第一个关心的数值，然后通过RET来进行重复。比如，假设有一个指向结构的dtab数组，我们关心每个结构中的fv成员。下面是输入的示例：

set $i = 0

p dtab[$i++]->fv

RET

RET

…

## 输出格式

默认情况下，GDB根据数据的类型打印出它的数值。有时这并不是你想要的。比如，想要以16进制打印一个数字，或者以十进制打印一个指针。或者可能先要以字符串或者一个指令的形式查看指定内存中数据。要实现这些，需要在打印数值的时候指定输出格式（*output format*）。

最简单的输出格式的应用是说明如何打印一个已经计算过的数值。通过print的第一个带斜线（/）和一个格式字符串组成的参数完成。支持的格式字符有：

x 将数值看为整数，并以十六进制形式打印。

d 以有符号十进制打印整数。

u 以无符号十进制打印整数。

o 以八进制打印整数。

t 以二进制打印整数。字母‘t’代表“two”。

a 作为地址而打印，绝对的十六进制地址加上前导符号的偏移。可以用来进行未知地址（在一个函数中）的定位：

(gdb) p/a 0x54320

$3 = 0x54320 <\_initialize\_vx+396>

这与命令info symbol 0x54320具有相似的结果。参考第16章 [符号]。

c 作为整数并且打印为字符常量。会同时打印打印它的数值和所代表的字符。超出7bit的ASCII字符的字符会使用转义的8禁止数值来表示，比如‘\nnn’。

如果不使用这个格式，GDB将**char**，**unsigned char**，和**signed char**数据显示为字符常量。向量的单字节成员显示为整数数据。

f 将bit数值作为浮点型数值，并且使用典型的浮点型格式来打印它。

s 如果可以，作为字符串。使用这个格式后，指向单个字节数据的指针作为以NULL结尾的字符串，单个字节数据的数组作为定长的字符串。其它的数值使用它们固有的类型。

不使用这个格式，GDB将指向**char**，**unsigned char**，和**signed char**的指针和数组显示为字符串。单字节成员组成的向量显示为整数数组。

r 使用‘原始’格式进行打印。默认情况下，GDB会使用有一个可以使用的基于Python的优美打印（参考[10.9章[优美打印]](#_优美打印)）。这会使用内容的高级显示功能。‘r’格式绕过可能存在的任何优美打印。

比如，使用十六进制打印程序的计数器（参考[10.13章[寄存器]](#_寄存器)），输入

p/x $pc

需要注意，在斜线前面不需要使用空格；这是因为GDB中的命令不含有斜线。

需要以不同的格式打印历史数值，可以使用仅仅带有格式的**print**命令，不需要指定表达式。比如，‘p/x’以十六进制打印上一个数值。

## 检查内存

可以使用x命令（examine）来以不用的格式检查任何内存，而不依赖于程序数据的类型。

x/*nfu addr*

x *addr*

x 使用**x**命令检查内存。

*n*，*f*，和*u*是可选的参数，用于指定显示多少内存和显示的格式；*addr*是用于代表查看内存的起始地址的表达式。如果使用默认的*nfu*，不用键入斜线‘/’。一些命令为了方便设置*addr*为默认。

*n*，重复的次数

重复的次数是一个十进制的整数；默认的为1。它指定了需要显示多少内存。

*f*，显示格式

显示格式是**print**命令使用的所有格式（‘x’，‘d’，‘u’，‘o’，‘t’，‘a’，‘c’，‘f’，‘s’）之一，还有‘i’（针对机器指令）。最初默认的是‘x’（十六进制）。默认格式在每次使用**x**或者**print**命令的时候会改变。

*u*，单位大小

单位大小是下列之一

b 字节。

h 半字（两个字节）。

w 字（四个字节）。这是默认的初始情况。

g 大字（八个字节）。

每次在使用携带指定单位大小参数的**x**命令时，这个大小就成为下一次使用**x**命令的默认单位大小。对于‘i’格式，单位大小被忽略且不会被改写。对于‘s’格式，单位大小默认为‘b’，除非明确指定。使用x /*hs*来显示16bit字符的字符串，使用x /*ws*来显示32bit字符的字符串。下次使用x /*s*命令会再次显示8bit字符的字符串。注意，结果依赖于使用的程序的语言和编译器。如果是C语言，‘s’修饰后会使用UTF-16编码，‘w’会使用UTF-32编码。编码由程序语言进行设置且不能更改。

*addr*，起始显示地址

*addr*是想要GDB显示内存的起始地址。表达式不必含有指针数据（尽管可能含有）；它代表这内存的一个字节的整数地址。详细请参考[10.1章[表达式]](#_表达式)。默认的地址是上次查看的地址的下一个地址——但是其它的一些命令也会设置默认地址：**info breakpoints**（显示的上一个断点的地址），**info line**（一行的起始地址），和**print**（如果使用它显示内存的内容）。

比如，‘x/3uh 0x54320’要求显示三个半字（‘**h**’）大小的内存，使用无符号的十六进制（‘**u**’）格式，起始地址为**0x54320**。‘x/4xw $sp’使用十六进制（‘**x**’）打印栈帧上面的4个字（‘**w**’）的内存（这里的‘**$sp**’，参考[10.13章[寄存器]](#_寄存器)）。

由于代表单元大小的字符和输出格式的字符都不相同，所以不用区分是先指定单元大小还是先指定输出格式；那个顺序都可以。使用‘**4xw**’和‘**4wx**’的输出是完全一样的。（然而，重复次数*n*必须首先指定；‘**wx4**’是错误的。）

尽管单位大小*u*在格式‘**s**’和‘**i**’中被忽略，仍然可以使用重复次数*n*；比如，‘**3i**’指定想要查看3条机器指令，包括任何操作数。为了方便，尤其是使用**display**命令时，‘**i**’格式也会打印出存在的分支指令延迟槽（branch delay slot instructions），超过由重复次数指定的，在计数器内的会立即执行下一条指令的指令。**disassemble**命令提供另一种可选的检查机器指令的方式；参考[9.6章[源代码和机器码]](#_源代码和机器码)。

**x**命令的所有默认参数被设计为让后续的使用**x**命令查看内存时使用最少的参数。比如，在用命令‘**x/3i *addr***’查看了三个机器指令后，可以使用命令‘**x/7**’继续查看下面7个。如果使用回车来重复**x**命令，重复次数*n*会被再次使用；其它的默认参数就像连续使用**x**。

当查看机器指令时，位于程序计数器处的指令会使用=>进行标记。比如：

(gdb) **x/5i $pc – 6**

0x804837f <main+11>: mov %esp, %ebp

0x8048381 <main+13>: push %ecx

0x8048382 <main+14>: sub %0x4, %esp

=> 0x8048385 <main+17>: movl $0x8048460, (%esp)

0x804838c <main+24>: call 0x80482d4 <puts@plt>

由**x**命令显示出的地址和内容不在内容历史中保存因为这些内容太多且会产生阻碍。而是，GDB让这些数值在后续的表达式中可用，使用变量**$\_**和**$\_\_**。在一个**x**命令后，上次检查的地址可以在表达式中通过常用变量**$\_**来引用。这个地址处的内容，查看到的东西，由常用变量**$\_\_**来保存。

如果**x**命令包含重复次数，那么保存的地址和内容从上次打印的最后一个内存单元中获得；这与输出中的一行中显示出多个单元的地址不同。

如果在远程目标机上面调试程序（参考20章，[远程调试]），可能想要验证远程机器的程序映像和下载到目标机上的可执行文件的差别。命令**compare-sections**就是为此而产生的。

compare-sections [*section-name*]

对比在被调试的可执行程序中的一个可加载的*section-name*段的数据和远程机器的内存，并报告任何不匹配的地方。如果不带参数，那么对比所有的可加载的段。这个命令是否可用依赖于目标机是否支持“qCRC”远程请求。

## 自动显示

如果发现需要经常查看一个表达式的值（观察它是如何变化的），可以将它加入自动显示列表，这样GDB在程序每次停止的时候都会打印它的值。每个加入自动显示列表的表达式都使用一个数字来标识它；从列表中删除一个表达式时，使用这个数字。自动显示格式如下：

2: foo = 38

3: bar[5] = (struct hack \*)0x3804

显示的内容为数字标识，表达式和它当前的值。

## 打印设置

## 优美打印

### 优美打印介绍

### 优美打印示例

### 优美打印命令

## 数值历史

## 常用变量

## 常用函数

## 寄存器

## 浮点型硬件

## 矢量单元

## 操作系统辅助信息

## 内存区域属性

### 属性

#### 内存访问模式

#### 内存访问大小

#### 数据缓存

### 内存访问检查

## 在内存和文件之间拷贝

## 如何从程序中产生core文件

## 字符集

## 缓存远程目标的数据

## 查找内存

# 调试优化代码

# C预处理宏

# 跟踪点

# 调试使用覆盖层的程序

# 用不同的语言使用GDB

# 检查符号表

# 改变执行

# GDB文件

# 指定调试目标

# 调试远程程序

# 指定配置的信息

# 控制GDB

# 扩展GDB

# 命令解释器

# GDB文本用户界面

# 在GNU Emacs下使用GDB

# GDB/MI接口

# GDB注释

# JIT编译接口

# 进程内部代理

# 报告GDB的bugs

# 命令行编辑

# 使用交互的历史

# 译者注：

针对原文中的某些词语的翻译，有些与我之前遇到过的翻译不同，这里我不得不解释一下：

stop

我们通常意为停止。但这个停止是具有阻塞意味的，并不是终止，更加确切的来说，应该为暂停。所以原文中的相应stop我都翻译为了暂停。

next和step

很多情况下我们都称它们为单步执行。但二者的区别还是很大的，next指令会跳过函数调用；step不会跳过函数而会进入函数内部。为了进行区分，我将next译为下一步，将step译为单步。

enable和disable

在某个点被enalbe（使能）和disable（禁止）操作之前，首先这个点需要存在。也就是说使能不是创建，禁止不是删除。

skippoint

为了与breakpoint对称，这里我译为跳过点。

stack和frame

stack，译为栈。在程序的虚拟内存镜像级别上来看，它是一片连续的内存空间。与堆，代码段等处于同一个级别。在数据结构的观点上来开，它是一种先入先出的队列。在程序的执行过程中，是动态增减的。

frame，译为帧。它是栈的一个子集。它对应于一个函数，包括函数的参数，地址，局部变量等等信息。从数据结构上，它继承栈的特性。

在函数的级别来看，stack里面存贮的元素就是frame，stack的增减对应着frame的压栈和出栈，对应着函数的调用和返回。

在本文中，单独说‘栈’，即指的是上面的stack。

无论是‘帧’还是‘栈帧’，都是指上面的frame。