GDB 使用说明书

编 写	vincent.cai	编写 时间	2010-04-16
文档版本	V1_0_0		

目录

前言

本翻译遵从 GPL。参见：

gdb is free software, protected by the gnu General Public License (GPL). The GPL gives you the freedom to copy or adapt a licensed program—but every person getting a copy also gets with it the freedom to modify that copy (which means that they must get access to the source code), and the freedom to distribute further copies. Typical software companies use copyrights to limit your freedoms; the Free Software Foundation uses the GPL to preserve these freedoms.

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

自由软件需要自由文档。

自由属于人民。

第1章 一个 GDB 会话样例

一个 GDB 会话样例

你可以随意用这部手册来了解有关 GDB 的一切。然而，一些趁手的命令就足以开始使用调试器。这一章介绍了这些命令。

在这个简单的会话里，我们强调用户输入用黑体来显示，这样可以和环境输出明确的区分开来。

GNU m4（通用宏处理器）的以前版本有以下的一个 bug:有时候，在我们改变了宏默认的引号字符串的时候，用来在别的宏里捕获宏定义的命令就失效了。在接下来简短的 m4 例子里，我们定义了一个展开是“0000”的宏 foo；我们接着用 m4 内建的 defn 来定义宏 bar，bar 的值也是“0000”。然而，在我们用<QUOTE>来替代开引号字符和用<UNQUOTE>替代闭引号字符的后，定义一个同义词 baz 的相同的过程却失败了。

```
baz:
$ 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
```

让我们试着用 GDB 来看看发生了什么。

```
$ gdb m4
gdb is free software and you are welcome to distribute copies
of it under certain conditions; type “show copying” to see
the conditions.
There is absolutely no warranty for gdb; type “show warranty”
for details.
gdb 6.8.50.20080307, Copyright 1999 Free Software Foundation, Inc…
(gdb)
```

GDB 只是读入仅够在有需要的时候用来发现哪里能够找到后续内容的数据；这将是 GDB 的第一个提

示很开出现。现在我们让 GDB 用一个比通常窄的显示区域，这样可以让本书的例子显示的更好。

```
(gdb) set width 70
```

我们需要探查 m4 内建函数 changequote 是如何工作的。因为已经看过了源代码，我们知道相关的子函数是 m4_changequote，所以我们用 GDB break 命令在这个函数上设置一个断点。

```
(gdb) break m4 changequote
Breakpoint 1 at 0×62f4: file builtin.c, line 879.
```

用 run 命令，m4 就在 GDB 的控制下运行了。只要还没有运行到 m4_changequote 子函数，程序就如同往常一样运行：

```
(gdb) run
Starting program: /work/Editorial/gdb/gnu/m4/m4
define(foo,0000)
foo
0000
```

为了出发断点，我们要调用 changequote.GDB 挂起 m4，显示出在它停止执行处的上下文信息。

```
changequote(<QUOTE>,<UNQUOTE>)
Breakpoint 1, m4_changequote (argc=3, argv=0×33c70)
at builtin.c:879
879 if (bad_argc(TOKEN_DATA_TEXT(argv[0]),argc,1,3))
```

现在我们可以用命令 n 接着运行当前函数的下一行指令。

```
(gdb) n
882 set_quotes((argc >= 2) ? TOKEN_DATA_TEXT(argv[1])\
: nil,
```

set_quotes 看上去像有可能失败的子函数。为了进入子函数，我们可以用命令 s(step)而不是用 next。step 进入被调用的子函数的第一行，所以 step 进入 set_qoutes.

```
(gdb) s
set_quotes (lq=0×34c78 “<QUOTE>”, rq=0×34c88 “<UNQUOTE>”)
at input.c:530
530 if (lquote != def_lquote)
```

这些输出显示了在 m4 挂起时的调用函数栈。它显示了栈的概要。我们可以用命令 backtrace(也可以用缩写 bt)来看当前在哪里：

backtrace 命令显示了每一个活动子程序的栈。

```
gdb) bt
#0 set_quotes (lq=0×34c78 “<QUOTE>”, rq=0×34c88 “<UNQUOTE>”)
at input.c:530
#1 0×6344 in m4_changequote (argc=3, argv=0×33c70)
at builtin.c:882
#2 0×8174 in expand_macro (sym=0×33320) at macro.c:242
#3 0×7a88 in expand_token (obs=0×0, t=209696, td=0xf7fffa30)
```

```
at macro.c:71
#4 0×79dc in expand_input () at macro.c:40
#5 0×2930 in main (argc=0, argv=0xf7fffb20) at m4.c:195
```

我们 step 几行来观察会发生什么。在开始的两次为了避免进入 xstrdup 子程序，我们用’s’;接下来的两次我们用’n'。

```
(gdb) s
0×3b5c 532 if (rquote != def_rquote)
(gdb) s
0×3b80 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);
```

最后一行输出看上去有点怪异；我们可以检查变量 lquote 和 rquote 的值来看看它们是否是真的是我们所期待的新的左右引号。我们用命令 p(print)

```
(gdb) p lquote
$1 = 0×35d40 “<QUOTE>”
(gdb) p rquote
$2 = 0×35d50 “<UNQUOTE>”
```

lquote 和 rquote 确实是新的左右引号。为了查看一些相关的信息，我们可以用命令 l(list)显示当前代码附近的 10 行代码。

```
(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
```

让我们 step 两行（设置 len_lquote 和 len_rquote），然后检查这两个变量的值。

```
(gdb) n
539 len_rquote = strlen(lquote);
(gdb) n
```

```
540 }
(gdb) p len lquote
$3 = 9
(gdb) p len rquote
$4 = 7
```

假设 len_lqoute 和 len_rquote 分别代表 lqoute 和 rquote 的长度，很显然这个结果是错误的，我们可以用命令 p 来设置较合理的值，p 命令不仅可以打印表达式的值，而且表达式也可以调用子函数，也可以给表达式赋值。

```
(gdb) p len lquote=strlen(lquote)
$5 = 7
(gdb) p len rquote=strlen(rquote)
$6 = 9
```

是不是这样就足以修正 m4 内建的 defn 关于使用新的引号的错误了呢？我们可以用命令 c 让 m4 继续执行,接着试这个刚刚过引起问题的例子：

```
(gdb) c
Continuing.
define(baz,defn(<QUOTE>foo<UNQUOTE>))
baz
0000
```

成功了！这个新的引用现在和默认的一样正确了。这个问题看来是两个形参搞错了。我们输入一个 EOF 让 m4 退出：

```
Ctrl-d
Program exited normally.
```

消息’Program exited normally.’是 GDB 输出的；它显示了 m4 已经执行完了。我们可以用 quit 命令来结束 GDB 会话。

第2章 进入和离开 GDB

这章讨论了如何开始和离开 GDB。

提要：

1. 输入’gdb’开始 GDB
2. 输入 quit or Ctrl-d 来退出

2.1 调用 GDB

运行 gdb 程序调用 GDB。一旦开始执行，GDB 会一直从终端读入命令，直到你告诉它结束为止。

在需要制定一些调试环境的时候，你也可以在开始的时候就用可变长参数和选项来运行 GDB。

命令行参数描述了 GDB 可以适合多种情况；在某些环境下，某些选项不可以用。

最常用的启动 GDB 的方式是使用一个制定要调试程序的参数：

```
gdb program
```

你也可以用可执行程序和 core 文件来启动:

```
gdb program core
```

如果你要调试一个进程，你可以用用进程 ID 来替代第二个参数：

```
gdb program 1234
```

这样可以 attach GDB 到进程 1234（除非你同时有一个文件叫“1234”，GDB 首先会检查是否有一个 core 文件）。

要利用第二个命令行参数需要操作系统的支持；当你用 GDB 作为远程调试器去 attach 到一个裸机上时，很有可能得不到任何有关进程的信息，也不大可能得到 core dump 文件。如果不能 attach 到进程或读入 core dump 文件，GDB 会给出警告。 你可以用参数－args 来让 gdb 传递参数给被调试的可执行程序.这个选项可以停止参数的执行。

```
gdb –args gcc -O2 -c foo.c
```

这将让 gdb 调试 gcc，设置 gcc 的命令行参数(参见 4.3 节[参数]，27 页)为‘-O2 -c foo.c’.

你可以运行 gdb 而不输出开头信息，开头信息描述了 GDB 的非保障性，用-silent 参数指定：

```
gdb -silent
```

你可以用命令行参数来进一步控制 GDB 的启动。GDB 本身可以提示参数信息。

输入

```
gdb -help
```

来显示所有可选项和简短参数用法描述（’gdb -h’是对等的缩写）。

所有的选项和命令行参数将以先后顺序处理。’-x’选项将会改变一规则。

2.1.1 选择文件

当 GDB 启动时它读入选项和参数，参数是固定指定为可执行程序和 core 文件的（或者进程 ID）。这和分别用选项‘-se’和’-c’(‘-p’)指定的参数一样。（GDB 读入的第一个没有选项相联系的参数将和用’-se’选项相关的参数一样；如果有，第二个没有选项相关的参数和用’-c’/'p’选项相关的参数一样）如果第二个参数用十进制数表示，GDB 首先会尝试着 attach 到一个进程，如果失败了，GDB 会尝试着将它作为 core 文件打开。如果你有一个用十进制数字命名的 core 文件，你可以用前缀’./’(例如‘./12345′)来防止 GDB 将它误作为进程 ID.

如果 GDB 没有配置为支持 core 文件，就像大多数嵌入式目标一样，那么 GDB 将会视第二个参数为多余而忽略它。

很多选项都有长短形式；长短形式都将在下表中列出。如果你截断了长选项，只要这个选项参数足以明确的表达，GDB 也可以认识。(如果你喜欢，你可以用’－’标记选项参数而不是我们更都使用的’-')

```
-symbols file
```

-s file 从 file 读入符号表.

```
-exec file
```

-e file 用文件 file 作为可执行文件来执行，或者在和 core dump 连接的时候用来检查出数据.

```
-se file
```

从文件中读入符号表，而且将文件作为可执行程序.

```
-core file
```

-c 文件 file 将作为 core dump 来检查.

```
-pid number
```

-p number 连接到以 pid 为 number 的进程，作为命令 attach 的参数.

```
-command file
-x file
```

从文件 file 里执行 GDB 命令.参见 20.3 节[命令文件，221 页]

```
-eval-command command
-ex command
```

执行单一的 GDB 命令。这个选项可以多次调用来执行多个命令。它也可以用’-command’交叉.

```
gdb -ex ’target sim’ -ex ’load’ \
-x setbreakpoints -ex ’run’ a.out
-directory directory
-d directory
```

加入路径 directory 作为源代码和脚本文件的搜索路径.

```
-r
```

-readnow 立即读入每一个符号文件的符号表，而不是默认的那种在需要时才渐次读入的方式。这将是初始阶段慢一点，而以后执行将更快。

2.1.2 选择模式

你可以用多种模式运行 GDB－例如，批处理模式和安静模式。

```
-nx
```

-n 不执行任何初始化文件里的命令。通常，在处理完所有的命令选项和参数之后，GDB 会执行这些文件里的命令。

```
-quiet
-silent
```

-q“安静”。不打印介绍和版权信息。在批处理模式下这些信息也不打印。

```
-batch
```

以批处理模式运行。处理完所有命令文件（用’-x’指定）后以 0 状态退出（如果没有用‘-x’选项，需要执行完在初始化文件里的所有命令）。批处理模式在将 GDB 作为过滤器运行的时候很有用，例如下载和运行一个远程计算机上的程序；为了使这个模式更有用，信息‘Program exited normally.’将不输出（通常在 GDB 控制下会输出的）

```
-batch-silent
```

类似于批处理模式运行，但是完全的安静。GDB 所有的输出到 stdout 的信息都将禁止（stderr 不受影响）。这个模式比’-silent’更安静而将是交互式的会话失效。这个模式在使用给出‘Loading section’信息的目标是特别有用。

注意，通过 GDB 输出的那些目标也将变哑。

```
-return-child-result
```

GDB 的返回值是子进程（别调试的进程）的返回值，但是有以下的例外：

1. GDB 异常退出。例如，由于不正确的参数或者内部错误。在此情况退出码和没有‘-return-child-result’一样。

2. 用户用明确的值退出。例如,’quit 1′

3. 子进程没有运行，或者不可结束，这种情况下推出码是-1.

这个选项在和’-batch’/'-batch-silent’联用，GDB 作为远程程序加载器或者仿真接口时很有用。

```
-nowindows
-nw
```

“无窗口”。如果 GDB 内建图形用户接口，那么这个选项将让 GDB 只以命令行接口运行。如果 GUI 不可用，这个选项将不起效。

```
-cd directory
```

GDB 用 directory 作为它的工作目录，而不是当前目录

```
-fullname
```

-fGNU Emacs 在把 GDB 运行为子进程的时候设置这个选项。这个选项告诉 GDB 在每次栈显示的时候以标准且可识别的方式输出完整的文件名和行号（包括每次程序中断的时候）。可识别的形式看上去像两个’\032′字符开始，接下来是文件名，行号和字符位置和新行，他们用冒号分隔。Emacs-to-gdb 接口程序用两个’\032′字符作为信号来在一帧上显示源代码。

```
-epochEpoch
```

在运行 GDB 作为子进程时 Epoch Emacs-GDB 接口设置这个选项.它让 GDB 修改打印例程以此来让 Epoch 在单独的窗口里显示表达式的值。

```
-annotate level
```

这个选项设置 GDB 的注释级别。这个选项和‘set annotate level’命令相同（参见 25 张[注释]，291 页）。注释级别控制着 GDB 打印多少信息，提示，表达式的值，代码行和其它种类的输出。通常是 0 级，1 级为 GNU Emacs 运行 GDB 而用，3 级是给控制 GDB 的程序的最高可用级别，2 级已经不再使用了。GDB/MI 可以极大的取代注释机制（参见第 24 章[GDB/MI],235 页）。

```
－args
```

改变命令行的转译，一边把可执行文件参数后面的参数传递给它。这个选项将阻止选项的处理。

```
-baud bps
```

-b bps 设置被 GDB 用来远程调试的串口行速率（波特率或者 bits/每秒）。

```
-l timeout
```

设置被 GDB 用来远程调试的链接超时（秒）。

```
-tty device
-t device
```

将设备作为你的程序的标准输入输出。

```
-tui
```

在启动时激活文本用户接口。文本用户接口在终端上管理多种文本窗口，用来显示代码，汇编，寄存

器和 GDB 命令的输出（参见第 22 章[GDB 文本用户接口]，227 页）。作为选择，文本用户接口可以用程序’gdbtui’激活。如果你在 Emacs 时不要使用这个选项（参见第 23 张[在 GNU Emacs 里使用 GDB]）。

```
-interpreter interp
```

把解释器 interp 作为控制程序或设备的接口。这个选项由和 GDB 通讯的程序设置，并以此作为后台的。参见第 21 张[命令解释器]，225 页。

从 GDB6.0 版以后’－interpreter=mi’(或者’－interpreter=mi2′)导致 GDB 使用 GDB/MI 接口（参见第 24 章[GDB/MI 接口]，235 页）。

以前的 GDB/MI 接口，包括 GDB5.3 版本和选择了‘－interpreter=mi1’都已经废止了。更早的 GDB?MI 接口也不再支持了。

```
-write
```

以可读可写的方式打开可执行程序和 core 文件。和‘set write on’命令相同。（参见 14.6 节[补丁]，152 页）

```
-statistics
```

在每次完成命令和回到提示符的时候，此选项可让 GDB 打印时间和内存使用统计信息。

```
-version
```

此选项可让 GDB 打印版本号和非保障性声明然后退出。

2.1.3 GDB 在启动阶段的活动

下文描述了 GDB 在启动阶段时的活动：

1. 启动命令行解释器（由命令行制定）（参见 2.1.2 节[模式选项]，13 页）

2. 读入在你的 home 目录下的初始化文件（如果有的话）然后执行里面的所有命令。

3. 处理命令行选项和参数。

4 读入和执行在当前工作目录下的初始话文件（如果有的话）里的命令。只有在当前目录和你的 home 目录不同时才会执行。

因此，在你启动 GDB 的目录下你可以有不止一个的初始化文件。

5. 读入命令文件（用’-x’选项指定）。更多详细信息请请参见 20.3 节[命令文件]，221 页。

6. 读入记录在历史文件里的命令历史。更多详细信息请参见 19.3 节[命令历史]。

初始化文件和命令文件使用相同的语法，并且 GDB 用相同的方式处理它们。你的 home 目录下初始化文件可以设置选项(

像’set complaints’), 这样可以影响此后的命令行选项和参数的处理。如果你用了’-nx’选项，初

始化文件将不会被执行（参见 2.1.2 节

[选择模式],13 页）。

GDB 初始化文件通常称为’.gdbinit’．由于 DOS 文件系统的文件名限制，GDB DJGPP 口使用’gdb.ini’这个名字。GDB 的 Windows

口使用标准名称，但是如果发现’gdb.ini’文件，它会警告你并建议你重命名为标准名称。

2.2 退出 GDB

```
quit [expression]
```

q 要退出 GDB,可以用 quit 命令（缩写为 q),或者敲入文件结束符(通常是 Ctrl-d).如果你不提供表达式，GDB 会正常结束；否则 GDB 会用表达式的结果作为错误码结束。

中断（常常是 Ctrl-c)并不从 GDB 里退出，而是结束正在处理中的 GDB 命令然后回到 GDB 命令级。任何时候敲入中断都是安全的，因为 GDB 知道安全的时候才会让这个中断起效。

如果你用 GDB attach 过一个进程，你可以用 detach 命令释放进程（参见 4.7 节[调试已经运行的进程]，30 页）。

2.3 Shell 命令

在调试期间，如果你需要偶尔执行 shell 命令，不需要离开或者刮起 GDB;你可以直接使用 shell 命令。

shell command string

启动标准 shell 执行 command string.如果环境变量 SHELL 存在，环境变量 SHELL 决定哪一个 shell 来运行。否则 GDB 将用默认的 shell(Unix 系统’/bin/sh’，MS-DOS 用’COMMAND.COM’).在编译环境里 make 工具通常都是必需的。在 GDB 里你不需要用 shell 命令调用 make:

make make-args

用指定参数执行 make 程序。和’shell make make-args’相同。

2.4 日志输出

你可能想要保存 GDB 命令的输出到一个文件里。有多个命令可以控制 GDB 的日志。

```
set logging on
```

激活日志功能.

```
set logging off
```

关闭日志功能.

```
set logging file file
```

改变当前的 logfile 名字 l. 默认的 logfile 是‘gdb.txt’.

```
set logging overwrite [on|off]
```

默认的，gdb 会以附加的方式保存日志。如果你想改为覆盖方式保存的话，可以设置为覆盖方式。

```
set logging redirect [on|off]
```

默认的，gdb 输出会打印到终端和 logfile。可以将终端重定向到 logfile 里，如果你只要它输出到 logfile 里。

```
show logging
```

显示当前日志设置

第3章 GDB 命令

假如缩写是无歧义的话，你可以将一个 GDB 命令缩写为开头的几个字母；你也可以用回车键来重复一些 GDB 命令。你也可以用 TAP 键来让 GDB 补全一个命令的剩余部分（或者告诉你可供选择的命令，假如不止一个命令可选的话）。

3.1 命令语法

一个 GDB 命令是单独的输入行。没有长度限制。命令由一个命令名开始，接着是提供给命令的参数。例如，命令 step 接收一个代表步长的参数，就像"step 5".你也可以用不带参数的 step 命令。某些命令不允许参数。

GDB 命令名总是在没有歧义的情况下允许截短。在某些情况下，即使是有歧义的缩写也是允许的；比如，s 是特别为 step 而定义的缩写，即使有其他的命令也是以 s 开头。你可以用这些缩写作为 help 命令的参数测试他们。

一个空白行的输入（敲入回车键）对 GDB 而言意味着重复此前的命令。有些命令（例如 run)不能用这种方式重复；这些命令不经意的重复可能导致麻烦或者你不大希望重复他们。用户定义命令可以关闭这些 feature;参见 20.1.1 节[定义],227 页。

list 和 x 命令，在你用回车键重复他们的时候，会建构新的而不是重复此前输入的参数。这个特性可以很便捷扫描代码和内存。

GDB 也可以以另外一种方式使用回车键：和通用工具 more 相似的方式来区分长输出（参见 19.4 节[屏幕大小],219 页）。因为在这种情况下很容易按下过多的回车键，在产生长输出时 GDB 关闭命令重复的功能。

从#开始到行结束的文本都是注释；这些文本什么也不干。他们主要是在命令文件里起帮助理解的作用（参见 20.1.3 节[命令文件],229 页）。

```
Ctrl-o
```

绑定对于重复复杂的命令序列很有帮助。这个命令接受一个当前行，例如一个回车，接着从命令历史里取得相对于当前行的下一行来编辑。

3.2 命令补全

GDB 可以为你补全命令的剩余部分，如果有且只有一个可能的命令；它也可以在任何时间为你显示一个命令里的下一个词的有效可能值。命令补全功能对 GDB 命令，子命令和你的程序里的符号都有效。

无论何时你想要 GDB 补全一个单词的时候，按下 TAB 键就可以了。如果只有一个可能，GDB 会补全这个词，接着等待你去完成这个命令（按下回车键）。例如，如果你敲入

```
(gdb) info bre <TABi>
```

GDB 补全'breakpoints'的剩余部分，因为只有 info 子命令以'bre'开头：

```
(gdb) info breakpoints
```

现在你可以敲入回车键来运行 info breakpoints 命令；假如'breakpoints'看上去不像你期待的，你可以用回退键删除之，然后敲入别的。，假如'breakpoints'看上去不像你期待的。（如果在开头你就确信你要的就是 info breakpoints，你就可以用缩写的形式来立即回车运行'info bre'，而不必等命令补全再回车）。

如果在你按下 TAB 键的时候有过个候选项的话，GDB 会发出一个铃声。你可以多敲入几个字符后再试一下，或者再按一次 TAB 键；GDB 会为你显示所有可能补全的候选项。例如，你可能想要在一个名字开头是'make_'子函数里设置一个断点,而在你敲入 b make_<TAB>的时候，GDB 会发出一声响。再次敲入<TAB>键会显示所有以 make_开头的函数，例如：

```
(gdb) b make_ <TAB>
gdb sounds bell; press hTABi again, to see:
make_a_section_from_file make_environ
make_abs_section make_function_type
make_blockvector make_pointer_type
make_cleanup make_reference_type
make_command make_symbol_completion_list
(gdb) b make_
```

显示完所有可能的候选项之后，GDB 会复制你刚才的输入（在这个例子里是'b make_'）以便你完成这个命令。

如果你只是想要在开始的时候看看候选列表，你可以按下 M-?而不是按下<TAB>两次。M-?是<META>?.你可以在敲入?的时候按住<META>键（假如键盘上有这个键的话），假如没有这个键，你可以按下<ESC>再按下?来代替。有时候你需要的字符串可能含有圆括号，或者 GDB 认为这个字符串是不一个字。为了让补全功能在这种情况下生效，你可以用'(单括号)封起来。

这种情况最有可能出现在你敲入一个 C++函数名的时候。这是因为 C++允许函数重载（同一个函数名多次定义，以参数类型来区分）。例如，在一个名为 name 的函数设置断点的时候，你需要区分是在参数

为 int 的函数 name 上还是参数为 float 的函数 name 设置断点的。为了在这时用词补全功能，在函数名之前敲入一个单引号'。这样 GDB 就可以知道需要考虑比通常只按下<TAB>或者 M-?更多的信息：

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

在某些需要补全的情况下，GDB 可以提示你需要引号。这时，如果你开始的时候没有敲入引号，GDB 会为你插入一个引号：

```
(gdb) b bub <TAB>
```

GDB 会以下面的输出提醒你，然后响一声：

```
(gdb) b ’bubble(
```

通常的，在有重载符号情况下，在你还没有开始敲入参数列表的时候就用补全功能的时候，GDB 提示需要一个引号然后插入它。

更多有关重载函数信息，参见 12.4.1.3 节[C++表达式],126 页。你可以用 set overload-resolution off 命令关闭重载解决方案，参见 12.4.1.7 节，[GDB 的 C++功能],128 页。

3.3 帮助

用 help 功能，你可以获得 GDB 的命令信息。

3.3.1 help

h 你可以用 help(缩写 h)不带参数来显示一个命令分类的简短列表。

```
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without
stopping the program
```

```
user-defined -- User-defined commands
Type "help" followed by a class name for a list of
commands in that class.
Type "help" followed by command name for full
documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
help class
```

用 help 分类作为参数，你可以得到这个分类里命令列表。比如，下面是 status 分类的帮助显示：

```
(gdb) help status
Status inquiries.
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
```

用命令名作参数，GDB 会显示一段如何使用这个命令的信息。

3.3.2 apropos args

apropos 命令会在命令和文档里文档搜索这个 args 指定的正则表达式。这个命令会打印所有符合的结果。例如：

```
apropos reload
```

结果：

```
set symbol-reloading -- Set dynamic symbol table reloading multiple times in one
run
show symbol-reloading -- Show dynamic symbol table reloading multiple times in
one run
complete args
```

3.3.3 complete args

complete 命令列出所有可能的补全结果。用 args 指定你想要的命令的开头字母。例如：

```
complete i
```

结果：

```
if
ignore
info
inspect
```

这个是为 GNU Emacs 设计的。

更进一步的，你可以用 GDB 命令 info 和 show 来查询你程序的状态或者 GDB 本身的状态。这两个命令都支持多个主题的查询；这本手册会在恰当的时候介绍这两个命令。索引里的 info 和 show 下的列表列出了所有的子命令。参见[索引]，407 页。

3.3.4 info

这个命令（缩写 i）可以描述程序的状态。例如，你可以用 info args 显示传递给函数的参数,用 info registers 来列出寄存器数据，用 info breakpoints 列出你设置的断点。你可以用 help info 来取得 info 的所有子命令。

3.3.5 set

你可以用 set 命令把一个表达式的值来设置一个环境变量。例如，你可以用 set prompt $来设置 GDB 提示符。

3.3.6 show

show 和 info 不同，show 描述的 GDB 本身的状态。你可以用 set 命令改变大多数你可以用 show 显示的内容。例如，你可以用 set radix 来设置显示的数值进制系统，或者用 show radix 来显示数值进制。

你可以用不带参数的 show 命令来显示所有可以设置的参数和它们的值；你也可以用 info set。这两个命令是一样的。

还有其余 3 种 show 子命令，这 3 中命令缺乏对应的 set 命令：

```
show version
```

显示当前 GDB 的版本。你应该在 GDB bug 报告中包含版本信息。如果你的机器上有多个版本的 GDB, 你可能需要知道哪个版本是你正在运行的；随着 GDB 的发展，新的命令会引入，而一些旧的将废弃。同

时，许多系统供应商移植了不同版本的 GDB,在 GNU/Linux 发行版也存在着多种版本的 GDB.版本号和你启动时显示一样。

```
show copying
info copying
```

显示 GDB 版权信息。

```
show warranty
info warranty
```

显示 GNU 免责声明，或者保证（如果你的 GDB 版本有的话）。

第4章 在 GDB 里运行程序

在你开始在 GDB 里运行程序前，你需要在编译的时候产生调试信息。

你可以在你选定的环境里带参数（如果有的话）的启动 GDB。如果你是在本地调试，你可以重定向输入输出，调试一个已运行的进程，或者结束一个进程。

4.1 为调试而编译

为了有效的调试程序，你需要在编译的时候产生调试信息。调试信息存储在目标文件里；调试信息描述了数据和函数的类型，源代码和可执行代码的对应关系。

编译时指定编译器的'-g'选项可以产生调试信息。

在编译给你的客户发布的程序时，可以用'-O'选项指定编译器进行优化。然而，许多编译器不能同时处理'-g'和'-o'选项。如果用的是这些编译器，你得不到带有调试信息的优化过的可执行程序。

GCC,GNU C/C++编译器，带有或不带'-O'选项都可以用'-g'选项，因此可以让 GDB 调试优化过的代码。我们推荐你在编译程序时总是用'-g'。也许你认为你的程序是正确的，但决不要去碰运气。

请记住在你调试一个用'-g -o'编译的程序时，优化器已经重排了你的代码；调试器显示的是真正编译成的代码。在执行路径和你的源代码不一致时，不要太惊讶！一个极端例子：如果定义了一个变量，但从来也没用过，GDB 发现不了它----因为优化器已经把它优化掉了。

用'-g -o'和只用'-g'编译的程序有时候不大一样，特别是在有些具有指令调度的机器上。如有疑问，再用只带'-g'编译，如果这个版本修正了问题，请给我们发送一个报告（包含一个测试用例）。更多调试优化过的代码，参见 8.2 节[变量],76 页。

较早前的 GNU C 编译器允许一个'-gg'变体选项来产生调试信息。GDB 不再支持这个格式；如果你的 GNU C 编译器有这个选项，别再用了。

GDB 知道预编译宏，可以显示宏的展开式（参见第九章[宏],101 页）。由于'-g'选项产生的调试信息过多，大多数编译器不会产生与编译宏的信息。GCC3.1 版本以及此后的可以提供宏信息，如果在编译的时候指定了'-gdwarf-2'和'-g3'选项;前一个选项产生 Dwarf

2 格式的调试信息，后者产生"额外信息"。我们希望在将来能够找到一个精简的方式来表示宏信息，这样只要一个'-g'选项就可以了。

4.2 开始程序

run

r 在 GDB 里用 run 命令开始你的程序。你在启动 GDB 时指定要调试的程序名（VxWorks 例外）（参见第二章[进入和离开 GDB]），或者用 file 或者 exec-file 命令（参见 15.1 节[指定文件的命令]，155 页）。

如果你是在一个支持多进程的环境里运行 GDB 的话，run 命令创建一个子进程来运行你的程序。在某系不支持多进程的环境下，run 跳到被调试程序的开头。其他的目标，比如'remote',总是在运行的。如果你的到一个类似如下的错误信息：

The "remote" target does not support "run".

Try "help target" or "continue".

接着用'continue'继续运行你的程序。你可能需要先 load（参见[加载]，169 页）。

程序的执行总会受某些从它的上级那里得到的信息的影响。GDB 可以指定那些虚啊哟你在启动程序之前就要设置的信息（你可以在启动程序之后改变，但是只有在下一次运行的时候才起效）。这些信息可以划分为 4 类：

The arguments.

将你的程序的参数作为 run 命令的参数。如果在你的环境里有 shell，shell 可以用来传递参数，那样的话你就可以用普通的方式（比如 wildcard 展开和变量替换）来描述参数了。在 Unix 系统里，你可以用环境变量 SHELL 来控制使用那个 shell。参见 4.3 节[程序参数]，27 页。

The environment.

你的程序会从 GDB 里继承环境变量，而你也可以用 GDB 命令 set environment 和 unset environment 改变某些影响你的程序的环境变量。参见 4.4 节[程序的环境]，28 页。

The working directory.

你的程序从 GDB 里继承工作目录。你可以用 GDB 命令 cd 来改变工作目录。参见 4.5 节[程序的工作目录]，29 页。

The standard input and output.

你的程序使用和 GDB 一样的标准输入输出。你可以在 run 命令行里重定向输入和输出，或者你可以用 tty 命令来为你的程序设置不同的设备。参见 4.6 节[程序的输入输出]，29 页。

警告：输入输出重定向起效之后，你不能再用管道将程序的产生的输出传递到另外一个程序里去；如果你要这样试的话，GDB 很有可能结束调试这个错误的程序。

在你执行 run 命令后，你的程序马上就开始执行。参见第五章[停止和继续]，39 页，那里讨论了如何筹划中断你的程序。一 旦你的程序中断下来，你就可以在你的程序里调用函数，用 print 或者 call 命令。参见第八章[检验数据]，75 页。

如果在最近一次 GDB 读入符号表之后符号文件的修改时间发生了改变，GDB 会丢弃现有的符号表然后重新读入。重新读入符号表的时候，GDB 会试图保留你当前的断点设置。

start 不同的语言可能有不同的主函数的名称。对于 C/C++来说，主函数的名称一直都是"main",而有些语言例如 Ada 就不需要为他们的主函数指定一个特定的名称。依赖于编程语言，调试器可以方便的开始执行程序并且在主函数的入口点中断。

'start'命令的功能和在主函数入口点里设置一个临时断点后执行'run'命令相当。有些程序包含一个在主程序之前执行的加工期，加工期会执行一些启动代码。这依赖于你使用哪种语言写程序。例如，C++,构造函数会在 main 之前为静态和全局变量调用。因此调试器有可能在主函数之前就中断程序。而临时断点还会保留来中断程序的执行。

'start'的参数将会传递给你的程序。这些参数会以字符形式给'run'命令。注意要是下次不带参数调用'start'或'run'的时候，这些参数将会被重用。

有些时候必须在加工期调试程序。这样的话，start 命令在主函数入口点的中断就太迟了，唯一此时已经过了加工期。在这种情况下，在运行情在加工期代码上设置断点就可以了。

4.3 程序参数

参数可以藉由 run 命令的参数来指定。参数由 shell 传递给你的程序，shell 会扩展通配符和执行重定向。你的 SHELL 环境变量（如果有的话）会决定 GDB 使用哪种 shell.如果你没有定义 SHELL 的话，GDB 使用默认的 shell(Unix 下'bin/sh')。

在非 Unix 系统里，程序通常都是由 GDB 直接调用的，GDB 会用相应的系统调用来模拟 I/O 重定向，通配符的宽展由程序的加工期代码完成，不是由 shell 来做的。

不带参数的 run 命令使用前一次的 run 命令的参数，或者由 set args 命令设置的参数。

set args

为你的下一次执行程序设置参数。如果 set args 不带参数的话，run 就不带参数的执行程序。一旦你带参数的 run 程序的话，要想下一次不带参数的执行程序就只有用不带参数的 set args 命令了。

show args

显示在启动的时候传递给程序的参数。

4.4 程序的环境

环境由一系列的环境变量和环境变量的值的组合组成。环境变量可以很方便地记录一些东西，例如你的用户名，home 目录，终端类型，程序执行的搜索路径等等。通常通过 shell 来设置环境变量，这些变量就可以被别的程序继承了。这在调试的时候很有用，GDB 就不必重新启动来试验一个改变了的环境。

path directory

在 PATH 环境变量的前头加上 diretory(可执行程序的搜索路径)。GDB 用的 PATH 将不会改变。你可以指定多个路径名称，用空格符或者系统依赖（Unix':',MS_DOS、MS-Windows';'）的分隔符类分割如果 directory 已经在 PATH 里了，这个 directory 会移到 PATH 的前面以加快搜索。

在 GDB 搜索路径的时候，你可以用'$cwd'来代指当前工作目录。如果你用'.'代替，它代表在你执行 path 命令时的目录。GDB 在把路径加入 PATH 前用当前路径替代'.'.

show paths

显示可执行程序的搜索路径列表（PATH 环境变量）。

show environment [varname]

打印名为 varname 的环境变量的值。如果你没提供 varname，打印所有的环境变量的名称和值。你可以缩写 environment 为 env.

set environment varname [=value]

设置环境变量 varname 的为 value.这个值只为你的程序改变，GDB 本身不改变。value 可以使任意字符串；环境变量都是字符串，你的程序负责转译；如果被删除了，变量的值就被设置为 null 值。例如，命令：

set env USER = foo

告诉被调试的程序，下一次执行 run 的时候，它的用户是'foo'.(‘=’附近的空格是为了清楚些；空格不是必须的)

unset environment varname

删除传递给程序的环境变量。和‘set env varname =’不同，unset environment 是从环境变量里删除变量，而不是为它设置一个空值。

警告：在 Unix 系统，GDB 用 shell 执行程序，shell 由 SHELL 决定（如果有的话，否则用'/bin/sh'）。

如果 SHELL 环境变量指定的 shell 有初始化文件的话--例如 C-shell 的'.cshrc'，BASH 的'.bashrc'---这些文件里定义的变量都将影响你的程序。

你也可以将那些只在登录时用到的变量移到别的文件里，例如'.login'或者'.profile'。

4.5 程序的工作目录

每次你用 run 启动程序时，程序都从 GDB 的当前工作目录继承工作目录。GDB 从它的父进程（通常是 shell）那里继承工作目录,而你可以用 cd 在 GDB 里指定一个新的工作目录。

GDB 的工作目录是 GDB 操作文件时的默认目录。参加按 15.1 节[文件命令]，155 页。

cd directory

指定 GDB 的工作目录为 directory

pwd 打印 GDB 的工作目录

通常很难找到被调试程序的当前工作路径（因为它可以在运行的时候改变）。如果你是在支持'/proc'文件系统的系统下运行 GDB 的，你可以用 info proc 命令来查找当前被调试程序的工作目录。(参见 18.1.3 节，[SVR4 进程信息]，183 页）

4.6 程序的输入输出

缺省情况下，GDB 里运行的程序在与 GDB 相同的终端上输入输出。GDB 会在和你交互的时候切换到它自己的终端模式，不过它会记住你的程序的终端模式然后在继续运行程序切换到那个模式上。

info terminal

显示 GDB 记录的你的程序使用的终端模式。

你可以用 run 命令重定向程序的输入/输出。例如：

run > outfile

开始运行程序，将打印输出到文件'outfile'。

另外一个指定程序输入输出的命令是 tty 命令。这个命令接受一个文件名作为参数，然后将这个文件作为接下来的 run 命令的缺省值。它也可以为子进程重置控制终端。例如：

tty /dev/ttyb

将接下来 run 命令运行的进程的输入输出定向到'/dev/ttyb',并将此作为控制终端。

run 命令将改变 tty 命令对于输入输出的设备的设置，但不改变其控制终端。

用 tty 命令或者在 run 命令里重定向输入只会影响你调试的程序。GDB 的输入仍然来自于你的终端。tty 是 set inferior-tty 的别名。

你可以用 show inferior-tty 命令来趟 GDB 显示程序将要使用终端名。

set inferior-tty /dev/ttyb

将被调试程序的 tty 设备设置为/dev/ttyb

show inferior-tty

显示被调试程序目前的 tty 设备名

4.7 调试一个已经在运行的进程

attach process-id

这个命令 attach 到一个从 GDB 外启动的进程上。（info files 显示你当前活跃的目标）这个命令需要一个进程 id 作为参数。通常用 ps 工具来找到一个 Unix 进程的 ID,或者用'jobs -l'shell 命令。

在你执行 attach 命令之后，按下回车键 attach 将不会再次执行。

只有在支持进程的环境下，attach 命令才有效；例如，attach 在没有操作系统的裸机上市无效的。你必须有发给进程送信号的权限。

在你执行 attach 命令的时候，调试器首先在当前工作目录下查找进程的可执行程序，如果没有找到，接着会用源代码文件搜索路径（参见 7.5 节[指定源代码目录]，70 页）。你也可以用 file 命令来加载可执行文件。参见 15.1 节[指定文件的命令]，155 页。

GDB 在准备好要调试的进程后第一件事就是中断这个进程。可以在 run 启动的进程上的使用的命令也可以用在你 attach 的进程上，你可以检查，修改这个进程。你可以插入一个断点；你可以 step 和 continue; 你可以修改存储器。

如果你希望进程继续执行，你可以在 attach 之后用 continue 命令来继续。

detach

在完成了调试之后，可以用 detach 来释放 GDB 对进程的控制。detach 进程后，进程继续执行。detach 命令之后，进程和 GDB 就没有关系了，你还可以 attach 到另外一个进程或者用 run 启动一个程序。detach 执行之后，按下回车键不会再重复。

如果你 attach 过一个进程，退出 GDB 会 detach 这个进程。如果你是用 run 命令启动的话，你将 kill

这个进程。缺省的，GDB 会要求得到你的确认；你可以用 set confirm 命令来控制是否需要确认（参见 19.7 节[可选的警告和消息]，213 页）。

4.8 杀死子进程

kill 杀死在 GDB 里运行的子进程

在你希望调试一个 core dump 而不是进程的时候，这个命令很有用。在程序运行期间的时候，GDB 会忽略 core dump。

在某系操作系统，如果你在 GDB 里为这个程序设置了断点，这个程序就不能在 GDB 外运行了。你可以用 kill 命令来

让程序在 GDB 外运行。

在你运行程序的时候，kill 命令也有助于重编和重新连接程序，而有些系统是不可能做到这个的。在这种情况下，在下次执行 run 命令的时候，GDB 可以知道程序已经发生变化了，就会重新读取符号表（同时也会保留你目前的断点设置）。

4.9 调试多线程进程

在某些操作系统里，例如 HP-UX 和 Solaris，一个程序可能有多个线程。线程精确概念随着各个操作系统而不一样，但大体上，一个有多个线程的进程和多进程相似，除了多线程共享一个地址空间（就是说，他们可以检查和修改同一个变量）。另一方面，每个线程有它自己的寄存器和执行栈，也可能有自己私有的存储空间。

GDB 提供了多个调试多线程的工具：

新线程的自动通知

'thread threadno'，切换线程

'info threads'，查询线程

'thread apply [threadno] [all] args'，对线程列表执行命令

线程特定断点

'set print thread-events'，控制线程开始和结束时打印消息

警告：在各个支持线程的操作系统里，不是所有的 GDB 配置都支持这些工具的。如果你的 GDB 不支

持线程，这个命令就无效。例如，不支持线程的系统里'info threads'命令就不能输出信息，也会拒绝 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 线程调试工具可以观察进程的所有线程，而一旦 GDB 控制线程的话，这个线程就总是调试的焦点了。这个线程称为当前线程。调试命令从当前线程的角度来显示进程的信息。

一旦 GDB 察觉到进程的新线程，GDB 就会用‘[New systag]’的方式显示目标系统的标识。systag 是线程的标识，各个系统不一样。例如，当 GDB 发现一个新线程的时候，在 GNU/Linux 你可能看到

[New Thread 46912507313328 (LWP 25582)]。

而在 SGI 系统里，systag 就简单的形如‘process 368’，没有更多信息。

出于调试的目的，GDB 自己会给线程一个编号--总是一个整数。

info threads

显示当前进程里的线程的总概要。GDB 显示每个线程（以此为序）：

1.GDB 分配的线程号

2.目标系统的线程标识（systag）

3.线程当前栈的概要

线程号左边的星号'*'代表此线程是当前线程。例如：

(gdb) info threads

3 process 35 thread 27 0x34e5 in sigpause ()

2 process 35 thread 23 0x34e5 in sigpause ()

* 1 process 35 thread 13 main (argc=1, argv=0x7ffffff8)

at threadtest.c:68

在 HP-UX 系统里：

出于调试目的， GDB 为进程里每个线程分配一个线程号（以线程创建顺序分配小整数）。

无论何时 GDB 察觉到一个新线程，它会用‘[New systag]’的形式显示 GDB 自己的线程号和目标系统的线程标志 。

systag 是线程标识，各个系统下可能不同。例如，GDB 察觉到新线程，在 HP-UX，你能看到

[New thread 2 (system thread 26594)]。

info threads

显示所有线程的概要。GDB 显示每一个线程（以此为序）：

1.GDB 分配的线程号

2.目标系统的线程标识（systag)

3.线程当前栈的概要

线程号左边的星号'*'代表此线程是当前线程。例如：

(gdb) info threads

* 3 system thread 26607 worker (wptr=0x7b09c318 "@") \

at quicksort.c:137

2 system thread 26606 0x7b0030d8 in __ksleep () \

from /usr/lib/libc.2

1 system thread 27905 0x7b003498 in _brk () \

from /usr/lib/libc.2

在 Solaris 系统，那你可以用一个 Solaris 特有的命令来显示更多的信息：

maint info sol-threads

显示 Solaris 用户线程的信息。

thread threadno

将 threadno 指向的线程设置为当前线程。这个命令的参数 threadno 是 GDB 内部的线程号，就是'info threads'

命令显示第一列。

GDB 会显示你选择的线程的系统标识和它当前栈的概要：

(gdb) thread 2

[Switching to process 35 thread 23]

0x34e5 in sigpause ()

伴随着'[New...]'消息，'Switching to'之后的文本形式由你的系统线程标识表示方式决定。

thread apply [threadno] [all] command

thread apply 命令可以让你在一个或多个线程上执行名为 command 命令。用参数 threadno 指定你希望操作的线程数目。可以是单个线程号，'info threads'显示的第一列；或者可以是线程范围，像 2-4.要操

作所有线程，敲入 thread apply all command。

set print thread-events

set print thread-events on

set print thread-events off

GDB 察觉到新线程启动或线程结束的时候，set print thread-events 命令可以开启或关闭打印信息。缺省下，如果目标系统支持的话，这些事件发生的时候，这些信息会打印出来。注意，这些信息不一定在所有目标系统里都可以关闭的。

show print thread-events

显示是否在 GDB 察觉线程启动或结束时打印信息。

由断点或者信号决定，无论何时 GDB 停止程序，它都会选择断点或信号发生的线程。GDB 会用‘[Switching to systag]’形式标识线程提示线程上下文的切换。

更多关于 GDB 在停止启动多线程程序的行为的信息，参见 5.4 节[停止核启动多线程程序]，59 页。更多多线程程序观察点的信息，参见 5.1.2[设置观察点]，44 页。

4.10 调试多个程序

在多数系统下，GDB 没有为能用 fork 调用创建附加进程的程序提供特殊的支持。程序创建子进程时，GDB 会继续调试父进程，而子进程则不受影响。如果你此前在子进程的代码上设置了一个断点，则子进程会被 SIGTRAP 信号结束。

不过，如果你想调试子进程的话，有一个不那么麻烦的替代方案。在 fork 调用之后的子进程代码里调用 sleep 调用。如果 sleep 代码的调用由某些环境变量或者某个文件的存在与否来决定，那就很方便了：如果你不想调试子进程，不设置这些变量或者删除文件就可以了。在子进程休眠的时候，用 ps 程序来得到进程 id.接着用 GDB attach 到这个子进程上，如果你正在调试父进程，你需要新启动一个 GDB 实例，参见 4.7[Attach]，30 页。你就可以如同 attach 到别的进程那样开始调试子进程了。

在某些系统上，GDB 提供调试用 fork 或 vfork 调用创建子进程的程序的支持。目前，只有 HP-UX(11.x 和以后版本)和 GNU/Linux（2.5.60 内核版本及后续）提供这个功能的支持。

缺省的，在创建子进程的时候，GDB 会继续调试父进程，而子进程不受影响。

如果你要调试子进程，用命令

set follow-fork-mode.

set follow-fork-mode mode

设置调试器对于 fork 或 vfork 调用的反应。fork 或 vfork 创建一个子进程。mode 参数可以是：

parentfork 之后调试原进程。子进程不受影响。这是缺省方式。

childfork 之后调试新的进程。父进程不受影响。

show follow-fork-mode

显示当前调试器对于 fork/vfork 调用的反应。

在 Linux 下，如果你要调试父进程和子进程，用命令

set detach-on-fork.

set detach-on-fork mode

设置 GDB 在 fork 之后是否 detach 进程中的其中一个，或者继续保留控制这两个进程。

on 子进程（或者父进程，依赖于 follow-fork-mode 的值）会被 detach 然后独立运行。这是缺省 mode。

off 两个进程都由 GDB 控制。一个进程（子进程或者父进程，依赖于 follow-fork-mode）被调试，另外一个则被挂起。

show detach-on-fork

显示 detach-on-fork mode

如果你选择了设置‘detach-on-fork’为 off,那么 GDB 会保持控制所有被创建的子程序（包括被嵌套创建的）。你可以用 info forks 命令来显示在 GDB 里创建的子进程，然后用 fork 命令来从一个进程切换到另一个。

info forks

打印在 GDB 控制下被创建的子进程列表。这个表包括 fork id, 进程 id 和当前进程的位置（程序计数器）。

fork fork-id

切换到 fork-id 指定的进程。参数 fork-id 是 GDB 内部为 fork 分配的，如命令'info forks'所显示列表的第一列。

process process-id

切换到 process-id 指定的进程。参数 process-id 必须是'info forks'输出的。

要想结束调试一个被创建的进程，可以用 detach fork 命令（允许这个进程独立的运行），或者用删除（也杀死）的方法 delete fork 命令。

detach fork fork-id

detach 一个由 GDB 标识的 fork-id 指定的进程，然后从 fork 列表里删除。这个进程会被允许继续独立运行。

delete fork fork-id

杀死一个由 GDB 标识的 fork-id 指定的进程，然后从 fork 列表里删除。

如果你要调试一个 vfork 创建接着 exec 的进程的话，GDB 会在这个新的目标上执行到底一个断点。如果你在原来程序的主函数上设置了一个断点，子进程上的主函数上也有一个同样的断点。

如果子进程正在执行 vfork 调用，你不能调试子进程或者父进程。

如果你在 exec 调用执行之后运行 GDB 的 run 命令，新目标会重新启动。要重启父进程，运行 file 命令，父进程可执行程序名作参数。你可以用 catch 命令来在 fork,vfork 或者 exec 调用的时候让 GDB 中断。

4.11 为跳转设置书签

在某些操作系统（目前只在 GNU/Linux 上），GDB 可以保存一个程序状态的快照，称为检查点，以后可以跳回。

跳回到检查点会撤销所有在检查点之后的变化。这些变化包括内存，寄存器，甚至系统状态（有些限制）。这样可以有效的及时回到在检查点设置的状态。

因此，如果你单步调试到你认为你接近到快要发生错误的地方，你就可以保存一个检查点。接着，如果你不经意的走的太远错过了关键的状态，你可以回到检查点后再从那里开始，而不需要从头启动程序。

检查点对于需要很长时间或者单步调试里 bug 发生地方很远的情况下很有帮助。

用 checkpoint/restart 方法调试：

checkpoint

保存被调试程序当前执行状态的快照。checkpoint 命令不需要参数，但每个检查点都分配一个小整数标识，如同 breakpoint 标识一样。

info checkpoints

列出在当前被调试会话的检查点。对于每个检查点，信息显示如下：

Checkpoint ID

Process ID

Code Address

Source line, or label

restart checkpoint-id

在检查点号的状态上重新启动。所有程序变量，寄存器，栈帧等等都恢复到检查点上保存的状态。本质上将，gdb 会把时钟回拨到检查点所记录的时间。

注意，断点，GDB 变量，命令历史等不受检查点重置的影响。通常，检查点只重置被调试程序内部的状态，不影响调试器本省的状态。

delete checkpoint checkpoint-id

删除以前保存的检查点

回到以前保存的检查点，会重置程序的用户状态，加上相当数量的系统状态，包括文件指针。重置不会撤销已写入文件的数据，但是会把文件指针指向以前的文职，因此以前写入的数据就可以被覆盖。对于以只读模式打开的文件，指针也会回到以前的位置，因此可以重新读取数据。

当然，送到打印机（或者其它外设）的字符不能收回，而从别的设备（比如串口设备）里读取的数据可以从内部程序缓冲里撤销，但是不能被塞回到串行管道里去，然后再读取他们。相似的是文件的内如如果被改变了，也不能被重置。

然而，在这些约束条件下，你可以重新回到以前保存的程序状态去，重新调试--然后你可以改变事件的过程来执行一个不同的路径调试。

最后，在你回到检查点的时候，有些内部程序状态会不一样---程序的进程 id。每个检查点会有一个独立的进程 id(pid),每个都和原来的 pid 不一样。如果你的程序保存了一个进程 id 的本地副本，这会有一个潜在的问题。

4.11.1 使用检查点的隐含好处

在某些系统里（例如 GNU/Linux）,出于安全考虑，每个新进程的地址空间都要随机确定。这就很难或者说不可能在一个绝对地址上设置一个断点或者观察点，因为一个符号的绝对位置每次执行都不一样。

然而，一个检查点是一个进程的相同的副本。因此如果你在主函数的入口点创建了一个检查点，你可以避开地址空间的随机化的影响，而且符号也会呆在相同的位置。

第5章 中断和继续

使用调试器的主要目的是在程序结束之前可以中断它；或者是在程序出现问题的时候，你可以调查为什么出问题。

在 GDB 里，有多个原因可以让程序中断，例如信号，断点或者一个 GDB 命令之后（例如 step）执行新一行代码前。你可以检查和改变变量，设置一个新的断点，或者删除一个旧的断点，再接着执行。通常 GDB 提供的消息可以显示程序大量的状态---但你也可以在任何时候显式的请求这些信息。

info program

显示程序状态信息：是否在执行，是什么进程，为什么中断。

5.1 断点，监视点，捕获点

断点可以让程序在执行到某个点上停止下来。对于每个断点，你可以加上条件来更详细地控制程序是否中断。你可以用 break 命令（带变量）来设置断点（参见 5.1.1 节[设置断点]，40 页），变量用来指定程序在什么地方中断（以行号，函数名或者程序的绝对地址的方式）。

在某些系统里，你可以在可执行程序运行前，在共享库里设置断点。在 HP-UX 系统里有些小小的限制：你必须等到程序运行才能在那些被程序间接调用的共享库例程上设置断点，例如，例程是 pthread_create 调用的参数。

监视点是特殊的断点，在表达式的值改变的时候中断程序。表达式可以是是一个变量的值，或者是由操作符绑定的一个或多个变量，例如'a+b'.有时这种断点也称为数据断点。你必须用一个不同的命令来设置监视点（参见 5.1.2 节[设置监视点]，44 页），除此之外，你看原因两断点一样管理监视点：用相同的命令激活，禁用，删除断点和监视点。

在任何 GDB 中断程序的时候，你可以安排自动显示程序的数值。参见 8.6 节[自动显示]，81 页。

捕获点是另一种特殊类型的断点，用来在某些事件发生时中断程序，例如在抛出 C++异常或者加载库的时候。和监视点一样，你需要用不同的命令来设置捕获点（参见 5.1.3 节[设置捕获点]，47 页），除此之外，你可以类似断点来管理捕获点。(在程序接到一个信号时停止程序，用 handle 命令；参见 5.3 节[信号]，57 页）

GDB 会在你创建断点，监视点，捕获点的时候分配一个数字给它们；这些数字是从 1 开始的连续整

数。在很多用来控制断点多种功能的命令里，你可以用断点号来指明是操作哪一个断点。每个断点可以激活或者禁用；如果被禁用了，他就不再影响程序的运行，除非你再激活它。

5.1.1 设置断点

断点设置用 break 命令（缩写 b). 调试器用'$bpnum'变量记录你最近设置的断点号；关于便利变量用途的讨论，参见 8.9 节[便利变量]，89 页。

break location

在给定的位置(location)设置断点，位置可以是函数名，行号，或者是一个指令的地址。(参见 7.2 节[指定位置]，68 页，所有可能指定位置的方式)。断点可能在程序执行指定位置前的代码前中断程序。

在可以重载符号的源代码语言里，例如 C++，一个函数名可以涉及到多于一个可能中断的位置。参见 5.1.8 节[断点菜单]，52 页，讨论了这种情况。

break

在不带参数的情况下，break 命令在当前栈里的下一条指令里设置断点（参见第六章[检查栈]，61 页）。在当前栈的最低端，这可以让程序在控制返回到帧的时候立即中断。这和一个栈帧里的 finish 命令的效果相似--除了 finish 不留下一个有效的断点。如果你在栈帧的最低端用 break 而不带参数，GDB 在下次到达当前位置时中断程序；这在循环内很有帮助。

GDB 通常在继续执行时忽略断点，直到最少一条指令执行为止。如果没有这样做，你不禁用断点，你将不能通过断点。这个股则在程序中断的时候，不论断点是否存在，都可以生效。

break ... if cond

带参数设置断点；在每次断点到达时计算 cond 表达式，并且当且仅当表达式的值不为零的时候中断---就是说，如果 cond 表达式为真。

'...'代表可能的指定中断位置的参数（上面描述过的）。更多中断条件的信息，参见 5.1.6 节[中断条件，50 页]。

tbreak args

设置一个只中断一次的断点。args 和 break 命令里的参数一样，断点设置也一样，但断点在第一次程序中断后自动删除。参见 5.1.5 节[关闭断点]，49 页。

hbreak args

设置一个硬件支持的断点。args 和 break 命令的一样，设置也一样，但断点需要硬件支持，某些目标

硬件可能不支持。这个命令的主要目的是为了调试 EPROM/ROM 代码，所以你可以不改变指令而在这个指令上设置一个断点。这个指令可以用在 SPARClite DSU 支持的新的陷阱-产生和多数基于 X86 的目标。这些目标可以在程序访问某些数据或指令地址的时候产生陷阱，这些陷阱是设计用来调试寄存器的。然而硬件断点寄存器有断点数的限制。例如，在 DSU 上，一次只可以设置两个数据断点，如果多于两个的话 GDB 会拒绝的。在设置新的断点前删除或禁用不用的硬件断点（参见 5.1.5 节[禁用断点]，49 页）。参见 5.1.6 节[中断条件]，50 页。更多远程目标，你可以限制硬件断点的数量，见[设置远程硬件断点限制]，177 页。

thbreak args

设置一个只中断一次的硬件支持断点。args 和 hbreak 的参数一样，设置方式也一样。不过，和 tbreak 命令相似，断点会在程序第一次中断后自动删除。和 hbreak 命令相似，断点需要硬件支持，某些硬件可能不支持。参见 5.1.5[禁用断点]，49 页。参见 5.1.6 节[中断条件]，50 页。

rbreak regex

在所有匹配正则表达式 regex 的函数上设置断点。这个命令会在所有匹配的函数上设置无条件的断点，也打印设置的断点列表。一旦这些断点被设置上，它们就和用 break 命令设置的一样了。你可以删除，禁用它们，或者可以和别的断点一样为他们设置条件。

正则表达式的语法是标准的，就如'grep'工具用的一样。注意，和 shells 用的不一样，例如 foo*匹配开头是 fo，接下来有 0 或者多个 o 的函数。在你的正则表达式的开头和结尾有个隐含的.*,所以要想只匹配 foo 开头的函数，用^foo.在调试 C++程序，在非特定类的成员函数的重载函数的设置断点上，rbreak 很有用。

可以用 rbreak 命令在一个程序里的所有函数上设置断点，如下：

(gdb) rbreak .

info breakpoints [n]

info break [n]

info watchpoints [n]

打印断点，监视点和捕获点表。可选参数 n 代表打印特定断点的信息（或者监视点，捕获点）。对于每个断点，打印下列信息：

Breakpoint Numbers

Type 断点，监视点，或捕获点

Disposition

断点是否标记为禁用或删除

Enabled or Disabled

用'y'标记激活断点，用'n'标记断点禁用。

Address

程序里的断点位置，内存里的位置。对于一个挂起的断点，它的位置是未知的，这个域会包含'<PENDING>'。这类断点在共享库没被加载前世不会起作用的。详细说明见下面。一个断点对应多个位置的话，这个域会包含'<MULTPLE>'--详细说明见下面。

What

断点位置在程序源代码，文件和行号。对于一个挂起的断点，由于在对应的共享库未被加载前不能解释，此时断点命令会显示一个初始的字符串。

如果断点是有条件的，info break 会显示被断点影响的行上的条件；断点命令，如果有的话，会在这行后显示。一个挂起的断点可以有条件的指定。这个条件会在共享库加载后分析有效性，以此来确定一个有效的位置。

info break 带有一个断点号 n 的参数将只显示此断点。变量$_和 x 命令缺省的检查地址用来显示最近位置的断点（参见 8.5 节[查看内存]，79 页）。

info break 断点被执行过的次数。这个命令在和 ignore 命令和用的时候特别有用。你可以忽略大部分的断点执行，查看断点信息来看断点总共有多少次执行，然后再次运行，忽略这个总数少一次的端点执行。这将可以让你快速的到达断点的最后一次执行。

GDB 可以在程序的同一位置设置任意数量的断点。这不是愚蠢或毫无意义的。特别是在断点是条件性的情况下，更为有用（参见 5.1.6 节[断点条件]，50 页）。

一个断点可能对应于多个位置。这种情况的例子如下：

对于 C++构造函数，GCC 编译器产生函数体多个的实例，用于不同的重载场景。

对于 C++模板函数，函数里一个给定的行可以对应于任意数量的实例。

对于内联函数，一个给定的源代码行可以对应于多个内联的地址。

在这些情况下，GDB 会在这些相关的位置插入断点。

一个对应于多个位置的断点可能会用多行来显示断点信息表--一个表头行，接下来是每一行对应于每一断点位置。表头行在地址列里有'<MULTIPLE>'。每个位置有单独的行，这一行包含位置的实际地址，和那个位置对应的函数名。断点号列的形式是断点号.位置号。

例如：

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

将断点号.位置号作为参数传递给 enable 何 disable 命令，每个位置就可以被单独的激活或者禁用。注意，不能从列表里删除一个单独的位置，只能删除从属于父断点的整个位置列表（用 delete num 命令，num 是父断点的编号，上面例子里是 1）.禁用或者激活父断点（参见 5.1.5[禁用]，49 页）影响所有属于这个断点的位置。

在共享库里设置断点是很平常的事。程序运行的时候共享库可以显式加载/卸载，还可以多次重复。为了支持这个用例，GDB 会在共享库加载/卸载的时候更新断点位置。典型地，在库尚未加载或库的符号还不可用的时候，你可以在调试会话的开始在库里设置一个断点。在设置此类断点的时，GDB 会问你是否想要设置一个所谓的挂起断点--断点的地址还不能解释。

程序运行后，当一个新共享库加载以后，GDB 会重新计算所有断点。当一个新加载的共享库包含挂起断点引用到的符号或者行时，这个断点就变为已解析和普通断点了。在共享库卸载时，所有引用到它的符号或行的断点都成为挂起断点。

这个逻辑也适用于对应于多个位置的断点。例如，如果你在一个 C++模板函数里设置了一个断点，一个新加载的共享库有此模板的一个实例，一个新的位置会加到断点的位置列表里。

除了位置未解析外，挂起断点和常规断点没有区别。你可以设置条件或者命令，激活或者禁用和执行别的断点操作。

在'break'命令不能解析断点地址时，GDB 提供了附加的命令来控制解析此地址：

set breakpoint pending auto

这个命令是缺省行为。GDB 不能找到断点位置时，它会向你询问是否该创建一个刮起断点。

set breakpoint pending on

设置未识别断点位置应该自动的创建一个刮起断点。

set breakpoint pending off

挂起断点不创建。任何未识别断点位置都将导致一个错误。这个设置不影响此前创建的挂起断点。

show breakpoint pending

显示目前关于创建挂起断点的行为模式。

上面的设置只影响 break 命令和它的参数。一旦断点被设置了，共享库加载/卸载时会被自动的更新。

上面的设置只影响 break 命令和它的参数。一旦断点被设置了，共享库加载/卸载时会被自动的更新。

对于某些目标，断点所在的位置可以在只读或是可读写的，GDB 可以根据断点位置来自行决定用硬件或者软件断点。这个规则应用于用 break 命令来设置的断点，也用于那些用 next 和 finish 之类的命令设置的内部断点。对于用 hbreak 命令设置的断点，GDB 总会用硬件断点。

用下列命令控制这个自动行为：

set breakpoint auto-hw on

这是缺省行为。GDB 设置断点时，它会尝试用目标内存映射来决定是用软件还是用硬件断点。

set breakpoint auto-hw off

设置 GDB 不自动选择断点类型。如果目标提供了内存映射，GDB 会在试图在只读地址上设置软件断点的时候警告。

GDB 本身会在程序里为某些特殊目的设置断点，例如为了恰当的处理 longjmp（在 C 程序里）。这些内部断点分配负值编号，从-1 开始；'info breakpoints'不显示它们。可以用 GDB 维持命令'maint info breakpoints'来查看它们（参见[maint infobreakpoints],331 页）。

5.1.2 设置监视点

在一个表达式改变时，可以用监视点来中断程序运行，而不需要预先知道表达式在哪里发生变化。(这类断点有时称为数据断点)。表达式可以简单如单个变量的值，也可以复杂如多个变量用操作符结合起来。例如：

1.单个变量值的引用

2.一个地址转换为一个恰当的数据类型。比如，‘*(int *)0x12345678’会在指定的地址监视一个 4 字节长的区域（认为是一个整形)。

3.任意复杂的表达式，例如‘a*b + c/d’。表达式可以程序语言的任何正确的操作符（参见 12 章[语言]，119 页)。

可以在一个表达式上设置一个断点，即使这个表达式尚不能解析。例如，可以在‘*global_ptr’初始化前设置一个监视点。在程序设置了‘*global_ptr’和表达式用一个有效值时，GDB 会中断程序。如果表达式通过其他方式变有效，而不是通过改变

变量的方式的话（例如，malloc 调用而使‘*global_ptr’指向的内存可用），GDB 会在下次表达式改变时中断程序。

监视点可以用硬件或硬件的方式实现，这依赖于你的系统。如果是软件断点的话，GDB 单步跟踪程序并且每一次都测试变量的值，这将使得比普通的执行慢几百倍。（但这是值得的，在你没有线索去哪里找 bug 的时候）

在某些系统里，比如 HP-UX, PowerPC, gnu/Linux 和大多数基于 x86 的系统里，GDB 支持硬件监视点，而这不会降低程序执行速度。

watch expr [thread threadnum

设置一个表达式监视点。在表达式 expr 被被改写和值改变时 GDB 会中断程序。最简单（也是最常用的）的用法是监视一个变量：

(gdb) watch foo

如果命令包含[thread threadnum]参数，GDB 只会在 threadnum 标识的线程改变表达式 expr 的值时中断。注意，只在硬件监视点上 GDB 才起作用。

rwatch expr [thread threadnum]

程序读表达式的值时中断。

awatch expr [thread threadnum]

读或写表达式时中断。

info watchpoints

打印监视点，断点和捕获点列表；和 info break 相同（参见 5.1.1 节[设置断点]，40 页）

如果可能，GDB 就设置一个硬件监视点。硬件监视点执行的非常快，调试器会在指令产生监视值改变的时候报告这个变化。如果 GDB 不能设置一个硬件监视点，那么它会设置一个软件监视点，软件监视点会相对慢得多，而且是在变化发生之后的下一个指令才报告，并不是在发生改变的时候就报告。

用 set can-use-hw-watchpoints 0 命令可以迫使 GDB 只设置软件监视点。由于参数设置为 0 了，GDB 就再也不会试图去设置硬件监视点了，即使这个系统支持硬件监视点（注意，此前设置的硬件-协助的监视点仍将使用硬件机制来监视表达式的值）。

set can-use-hw-watchpoints

设置是否用硬件监视点。

show can-use-hw-watchpoints

显示硬件监视点的当前模式。

对于远程系统，你可是限制硬件监视点的数量，参见[设置远程 硬件-断点-限制]，177 页。

在执行 watch 命令时，GDB 报告：

Hardware watchpoint num: expr

如果成功设置了一个硬件监视点的话。

目前，awatch 和 rwatch 命令只能设置硬件监视点，因为不改变被监视的表达式的值的数据的访问不在每个执行到它的指令上都加以检查的话，就没法探测到；目前 GDB 也没有那样做。如果 GDB 发现用 awatch 或者 rwatch 不能设置硬件监视点的话，会打印类似如下信息：

Expression cannot be implemented with read/access watchpoint.

有时候，由于被监视的表达式的数据类型长度超出目标系统上支持的硬件监视点，GDB 可能不能设置硬件监视点。例如,某些系统只能监视最多 4 个字节宽的硬件监视点；在这些系统里，不能为带有双精度的浮点指针数据（通常是 8 字节长）的表达式设置一个硬件断点。一个可能的替代方案是，把一个长区域分成一系列小的监视点。

如果设置了太多的硬件监视点，GDB 有可能不能再继续执行程序时插入所有的监视点。由于准确的有效监视点的数量在程序重新执行后才能知道，GDB 可能不能在设置监视点的时候给出警告，警告信息会在程序重新执行后发出：

Hardware watchpoint num: Could not insert watchpoint

如果发出了这个警告，需要删除或禁用一些监视点。

监视复杂的引用了大量的变量的表达式可能会消耗光硬件断点可用的资源。这是因为 GDB 需要分别为监视表达式里引用的每个变量分配资源。

如果调用了一个交互使用打印或者调用的函数，任何监视点都只能等到 GDB 遇到另一种断点或调用结束之后才有效。

GDB 会在离开生存区的时候，自动删除监视本地（自动）变量的监视点，或者是引用到这种变量的表达式的监视点；

也就是说在离开这些变量定义区的时候。尤其是，在调试程序要结束时，所有的本地变量都要离开生存区的时候，只有全局变量的监视点才保留。如果重新执行程序，需要重设这些监视点。一个方法是在 main 函数的入口点设置代码断点，在执行断点的时候设置这些监视点。

在多线程程序里，监视点会从每个线程里发现被监视的表达式值的变化。

警告：多线程程序里，软件监视点只有有限的用处。如果 GDB 创建了一个软件监视点，它只能在一个线程里监视这个表达式的值。如果你你确信只有当前的线程活动会导致表达式的值的改变的话（并且你

确信没有别的线程会变成当前线程的话），那么你可以像通常那样使用软件监视点。然而，GDB 不能察觉非当前线程改变表达式值。（于此相反，硬件监视点能从所有线程中监视断点）参见[设置远程硬件-监视点-限制]，177 页。

5.1.3 设置捕获点

用捕获点可以让调试器为某些程序事件中断程序执行，例如 C++异常或者是共享库的加载。用 catch 命令来设置一个捕获点。

catch event

在 event 发生时中断。事件可以是下列的：

throwC++异常的抛出。

catchC++异常的捕获。

execption

Ada 异常。如果在命令的结尾指定了异常名（比如 catch execption Program_Error）,调试器就只会在这个异常发生时中断。否则，调试器会在任意的 Ada 异常发生时中断。

exception unhandled

没有被程序处理的异常。

assert 一个失败的 Ada 断言。

execexec 调用。目前只在 HP-UX 和 GNU/Linux 上可用。

forkfork 调用。目前只在 HP-UX 和 GNU/Linux 上可用。

vforkvfork 调用。目前只在 HP-UX 和 GNU/Linux 上可用。

load

load libname

动态加载共享库，或者加载库 libname。目前只在 HP-UX 上可用。

unload

unload libname

卸载动态加载的共享库，或者卸载库 libname。目前只在 HP-UX 上可用。

tcatch event

设置只中断一次的捕获点。捕获点会在事件第一次捕获之后自动删除。

用 info break 命令列出目前的捕获点。

目前 GDB 对 C++异常处理有一些限制（catch throw 和 catch catch）：

1.如果交互调用函数，GDB 通常会在这个函数结束时将控制权交还给你。如果这个调用产生了一个异常，这个调用可能越过交还控制权的机制，让程序结束或仅仅只是继续执行，直到它碰到一断点，捕获到一个 GDB 检测的信号，或者退出。即使你为异常设置了捕获点也是一样的；异常的捕获点在交互式的调用里是禁用的。

2.不能交互的产生一个异常。

3.不能交互的注册异常处理函数。

某些时候 catch 并不是最好的异常处理调试技术：如果你需要异常发生的精确位置的话，在异常处理函数前中断是更好的选择，因为那样的话可以看到还未退绕的栈。相反，在异常处理函数里设置断点的话，就不太容易找出在哪里发生了异常。

要在异常处理函数调用前中断程序，你需要了解一些实现的知识。就 GNU C++而言，异常是调用一个名为__raise_exception 的库函数而引发的，这个函数有下面的 ANSI C 接口：

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

要让调试器在栈没有退绕前捕获异常，在函数__raise_exception 上设置断点就可以了（参见 5.1 节[断点；监视点；和异常]，39 页）。

用一个条件断点（参见 5.1.6 节[断点条件]，50 页），可以在一个指定异常发生时中断。可以用多个条件断点来在任何一个异常发生的时候中断程序。

5.1.4 删除断点

常常有必要在断点完成任务之后删除断点，监视点或者捕获点，因为不再需要它们再次中断了。这就是删除断点。一个断点删除之后就不存在了；它被遗忘了。

用 clear 命令可以从程序中删除所有断点。用 delete 命令指定断点号可以删除单独的断点，监视点或者捕获点。

用删除断点来继续执行并不是必须的。只要不改变执行地址的执行程序，GDB 会自动忽略在将要执行的第一个指令上的断点。

clear 在选定的栈帧上的下一个指令上删除所有的断点（参见 6.3 节[选择一个帧]，64 页）。只要选择最内层的帧的话，这是一个在程序中断地方删除断点的好方法。

clear location

在指定的位置删除所有的断点。参见 7.2 节[制定位置]，68 页，更多关于位置的形式；最有用的形式如下表：

clear function

clear filename:function

删除在名为 function 入口点上的断点。

clear linenum

clear filename:linenum

删除在指定的行或文件名上的断点。

delete [breakpoints] [range...]

在指定的范围内删除断点，监视点或者捕获点。如果没有制定参数，就删除所有的断点（GDB 会要求确定，除非你设置了 set confirm off）。delete 命令可以缩写为 d.

5.1.5 禁用断点

相比删除断点，监视点或者捕获点，你也许更愿意禁用它们。这将让断点不起作用就如它被删除了一样，但会记住断点的信息，可以在以后激活。

可以用 enable 和 disable 命令禁用和激活断点，监视点和捕获点，可选择指定一个或多个断点号作为参数。如果不知道用哪个断点号，用 info break 或 info watch 来打印断点，监视点，捕获点的信息。

禁用和激活有多个位置的断点会影响其所有的位置。

断点，监视点，捕获点有四种激活状态中的任意一种：

1.已激活的（Enabled）。断点会中断程序。这个状态是用 break 命令发起的。

2.已禁用的（Disabled）。断点不再影响程序。

3.激活一次（Enabled once）。断点会中断程序，但会变成 disabled 状态。

4.激活并删除（Enabled for deletion）。断点中断程序，但中断后立即就永久删除。这个状态是用 tbreak 命令发起的。

可以用下列命令来激活或禁用断点，监视点，捕获点：

disable [breakpoints] [range...]

禁用指定断点--或者所有的断点，如果没有参数的话。已禁用的断点不再起效，但没有被删除。所有的选项如

ignore-counts,conditions 和 commands 会记住，以便将来再次激活。disable 缩写 dis。

enable [breakpoints] [range...]

激活指定的断点（或者是所有的断点）。重新可以中断程序。

enable [breakpoints] once range...

临时激活指定的断点。GDB 会在这个断点中断程序之后立即禁用它。

enable [breakpoints] delete range...

激活断点中断一次，然后删除之。GDB 在程序中断后立即删除这类断点。这类断点用 tbreak 命令发起。

除了用 tbreak 命令设置的断点（参见 5.1.1[设置断点]，40 页），断点会自动激活；因此，它们用上述的命令转换状态。（命令 util 可以设置和删除它自己的断点，但不改变其它的断点的状态；参见[继续和单步跟踪]，54 页）

5.1.6 中断条件

最简单的断点会在程序执行到指定位置时中断程序。也可以为断点指定条件。条件只是程序语言的一个 Boolean 型的表达式（参见 8.1 节[表达式]，75 页）。带条件的断点会在程序执行到底时候计算表达式的值，只有在条件为真的时候才中断。

这是和程序断言的用法是相反的；这这个情况里，你希望在断言失败是中断--就是说，条件为假。在 C 语言里，如果要测试由叫天 assert 表示的断言，应该在相应的断点上设置条件为‘! assert’。

监视点也可以用条件；其实监视点不太需要条件，因为监视点总在检测表达式的值--但用条件可能更简单，在一个变量名上设置一个监视点，并制定一个条件来测试新值是否是你期望的。

断点条件可能有边际效应，甚至可能调用程序里的函数。这可能很有用，比如，在激活程序进程日志函数，或者用你自己的打印函数来格式化特殊数据结构。除非在相同的位置还有另外已激活断点，否则效应是完全可预测的。（假使那样的话，GDB 会首先检查另外的断点来中断程序，不检查此断点的条件）注意，要在断点到达时实现边际效应，断点命令通常比断点条件更方便更灵活。（参见 5.1.7[断点命令列表]，51 页）

在 break 命令里用'if'可以在设置断点时指定断点条件。参见 5.1.1 节[设置断点]，40 页。它们也可以

在任何时候用 condition 命令改变。

你也可以在用 watch 时带 if 关键字。catch 命令不识别 if 关键字。只有 condition 可以用来对捕获点进一步的设置。

condition bnum expression

为断点，监视点，捕获点指定断点条件表达式。一旦你设置了条件，断点 bnum 就只在表达式为真中断程序（非零，在 C 里）。当用 condition，GDB 立即检查语义的正确性，然后判断它用到的符号是否用在了断点上下文里。如果表达式用到的符号不在断点上下文里，GDB 会打印一个错误消息：

No symbol "foo" in current context.

然而，GDB 不会真的在用 condition 命令的同时就计算表达式（或者在一个命令带条件的设置断点，像 break if...）.参见 8.1 节[表达式]，75 页。

condition bnum

从断点 bnum 里删除条件。这个断点就成为普通的无条件的断点。

一个特殊的条件断点的例子是在断点达到一定次数时才中断。这非常有用，因此有一个特别的方式来实现：用断点的忽略计数。每个断点都有一个忽略计数，忽略计数是个整型数。大多数情况下，忽略计数是 0，因此没有什么效用。但是程序执行到一个忽略计数为正数的的断点的时候，那么就会中断执行，忽略计数减 1，接着执行程序。结果是，如果忽略计数是 n，断点不会在接下来的 n 次中断程序。

ignore bnum count

将断点 bnum 的忽略计数设置为 count。接下来的 count 次碰到断点，程序执行不会中断；除了忽略计数减 1，GDB 不做别的。

要在下一次断点执行到的时候中断程序，设置忽略计数为 0。在用 continue 在中断后来重新执行，可以直接指定忽略计数为 continue 的参数，而不需要用 ignore.参见 5.2 节[继续和单步跟踪]，54 页。如果一个断点有一个正当忽略计数和条件，不会检查条件。一旦忽略计数变为 0，GDB 会重新检查条件。

要达到和忽略计数一样的效果，可以用每次自减 1 的调试器方便变量作为断点条件来做，比如‘$foo--<= 0’。

忽略计数可以用于断点，监视点和捕获点。

5.1.7 断点命令列表

可以为断点（或监视点，捕获点）设置一系列命令来让断点中断时执行之。例如，你可能想要打印某

些表达式的值，或者激活别的断点。

commands [bnum]

... command-list ...

end 为断点 bnum 指定命令列表。commands 后面接命令行。最后输入一行 end 阿里结束命令。

要删除一个断点的命令，输入 commands 后接一行 end；就是说，不带命令。

不带 bnum 参数的话，commands 指定最后的断点，监视点或者捕获点（不是最近执行到的断点）。

在命令列表里，按下回车键意味着重复上一个命令的功能将被禁用。

可以用断点命令来重新启动程序。仅仅只用 continue 命令，或者 step，或者是其它重新执行的命令就可以了。

在重新执行的命令之后的命令将会被忽略不执行。这是因为每次重新执行的时候（即使只是 next 或 step）,可能会碰到别的断点--这个断点可能有自己的命令列表，会产生执行哪一个命令列表的歧义 。

如果你指定的命令列表的的哥命令是 silent，通常断点中断的所产生的消息就不会打印。这个在打印一个特定的消息，然后接着继续执行的中断而言是可取的。如果剩下的命令都不打印消息，你将看不到什么中断的迹象。silent 只在断点命令开头有意义。

命令 echo，output 和 printf 可以打印精确控制的输出，并常常在哑断点里很有用。参见 20.4 节[控制输出的命令]，222 页。

举个例子，下面的例子可以让你用断点命令来在 foo 的入口点，当 x>0 时打印 x 的值。

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

断点命令的一个应用是应付一个 bug（就是说先放下此 bug）来测试另一个。在出错行后的代码上设置一个断点，设置条件来检查出错情况，用命令来给需要用到的变量设置正确的值。用 continue 命令来结束命令列表，因此程序就不会停止，用 silent 命令来禁止输出信息。下面是一个例子：

```
break 403
commands
silent
```

```
set x = y + 4
cont
end
```

5.1.8 断点菜单

某些编程语言（特别是 C++和 Objective-C）允许一个函数名可以定义好几次，应用于不同的上下文，称之为重载。

当一个函数名重载时，‘break function’就不足与让 GDB 明白你需要在哪里设置断点。可以用函数的准确特征来指明要设置的是哪个函数版本，例如用‘break function(types)’。否则，GDB 会提供一个数字标识的菜单供你选择不同的断点，并用提示符'>'等待你的选择。开头的两个选项是‘[0] cancel’和‘[1] all’。输入 1 会在每个函数上设置一个断点，输入 0 取消 break 命令设置新的断点。

例如，下面的会话摘录显示的是在重载的符号 String::after 上设置断点。选择了 3 个特别的函数定义体：

```
(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)

5.1.9 “不能插入断点”

在某些操作系统里， 如果有其它的进程运行了这个程序的话，断点就不能在程序里设置了。在此情况下，用一个断点试图运行或继续执行程序会引起输出一个错误信息：

Cannot insert breakpoints.

The same program may be running in another process.

这个情况发生时，有三种继续的方式：

1.删除或关闭断点，接着继续。

2.挂起 GDB，复制程序文件并命名为另一个名字。重新执行 GDB，用 exec-file 命令来指定 GDB 要执行的程序文件名。接着重启程序。

3.用链接器选项'-N'重新链接程序，以使文本段是非共享的.操作系统的限制可能不应用于非共享模式的可执行程序。

如果你要求设置太多有效的硬件支持的断点和监视点的话，类似的信息也会输出：

Stopped; cannot insert breakpoints.

You may have requested too many hardware breakpoints and watchpoints.

这个消息在你试图继续执行程序是打印，因为 GDB 只有在此时才能准确的知道有多少硬件断点和监视点需要插入。

在这个消息打印的时候，你需要禁用或者删除一些硬件支持断点和监视点，然后接着继续执行。

5.1.10 “断点地址已调整...”

某些处理器架构对断点所在的地址有限制。对于那些有限制的架构，GDB 会试图调整断点的地址来符合它的限制。

这类架构的一个例子是富士的 FR-V。FR-V 是 VLIW 架构，有着类 RISC 的指令集，它的一组指令可能集合在一起以便于并行运算。FR-V 架构限制断点指令的位置在一组指令集的最低的地址空间。GDB 会主动调整断点位置，将断点指令设置在这组指令集的开头来适应 FR-V 的限制。

由于优化代码的原因，得到一组指令集常常和源代码的表述是不一样的，因此断点位置可能被调整到和源代码不一致的地方。由于这样的调整使得 GDB 断点的行为可能和用户的预期有很大的不同，在设置

断点的时候会打印出一个警告信息，并且断点执行到时也会打印这样的信息。

断点在受到调整时，类似下面的警告信息会打印出来：

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

对于用户可设置的和 GDB 内部的断点，这样的警告都可能出现。如果你看到一个这样的警告，你应该确认调整后的位置上设置的断点是否具有你所期望的效果。如果不是，问题断点要删除，应该设置一个有效的断点。例如，可能在后面的指令上设置断点就够了。在某些情况下，条件断点在防治断点出发调整方面也很有用。

GDB 也会在调整后的断点上中断时打印一个警告：

warning: Breakpoint 1 address previously adjusted from 0x00010414

to 0x00010410.

遇到这个警告时，很可能要采取补救措施已经晚了，除非断点中断早于所期望的或者中断的频率比预期的高。

5.2 继续和单步跟踪

继续意味着重新执行程序知道程序正常结束。相反，单步跟踪意味着只执行程序“一步”，“一步”可能代表着一行源代码，或者一条机器指令（依赖于你用的特定命令）。不论继续或是单步跟踪，程序可能很快就由于断点或信号中断执行。（如果由于信号中断，你可能希望用 handle,或者用'signal 0'来继续执行。参见 5.3 节[信号]，57 页）

continue [ignore-count]

c [ignore-count]

fg [ignore-count]

在程序最近中断的地方重新执行；任何设置在这个地址上的断点都会被绕过。可选参数 ignore-count 可以让你进一步指定在这个位置上忽略断点次数；它的效果就如 ignore（参见 5.1.6[断点条件]，50 页）。参数 ignore-count 只有在程序由于断点中断时才有意义。其他时候，continue 的 ignore-count 参数会被忽略。

同义词 c 和 fg（表示 foreground，因为被调试的程序总被认为是前台程序）只是为方便而提供的，其行为就如 continue 一样。

要在不同的位置重新执行程序，可以用 return（参见 14.4 节[从函数里返回]）来回到调用函数；或者

jump（参见 14.2[在不同的地址继续]，150 页）到程序里的绝对地址。

使用单步跟踪的典型的技术是在函数的入口点设置一个断点，或者在可能发生错误的程序段上设置，运行程序知道它在断点上中断，再在这个嫌疑区域单步执行，检测感兴趣的变量，直到你发现错误发生。

step 继续执行程序直到另一行代码，再中断，把控制权交换 GDB。可以缩写为 s。

警告：如果在没有调试信息的函数里使用 step 命令，程序执行会继续执行直到有调试信息的一个函数。类似的，不会单步进一个没有调试信息的函数。要跳过没有调试信息的函数，用 stepi 命令，下面会介绍。

step 命令只在一行源代码的第一个指令上中断。这就阻止了在类似 switch 语句，for 循环上的多次中断。如果在这行代码上调用的函数有调试信息，step 会继续中断。换句话说是，step 会进入这行代码所调用的所有函数。

如果函数有行号信息，step 命令也只进入这个函数。否则 step 命令就如 next 命令。这个规则可以在 MIPS 机器上用 cc -gl 编译时避免错误。在以前，step 会进入有调试信息的子函数。

step count

继续单步执行，不过执行 count 次。如果执行到一个断点，或者一个与单步跟踪无关的信号在 count 次单步执行发生时，单步执行会立即中断。

next [count]

在当前栈帧（最内层）上继续执行到下一行源码行。和 step 相似，但在这行代码上调用的函数将不会中断。

程序执行到在原栈层里的另一行代码时会中断。缩写为 n。

参数 count 是重复次数，和 step 的一样。

命令 next 只在一行源代码的第一个指令上中断。这就阻止了在类似 switch 语句，for 循环上的多次中断。如果在这行代码上调用的函数有调试信息，step 会继续中断。

set step-mode

set step-mode on

set step-mode on 命令会导致 step 命令在没有调试行信息的函数的入口点中断，而不是越过这个函数。

如果你对一个不带符号信息的函数的机器指令很感兴趣，又不想 GDB 越个这个函数的话，那么这个命令就很有用了。

set step-mode off

导致 step 命令越过不带调试信息的函数。这是缺省的。

show step-mode

显示 GDB 是否中断或越过不带源代码行调试信息的函数。

finish

继续执行直到当前选定栈帧上的函数返回。打印返回值（如果有）。

和 return 命令相反（参见 14.4 节[从一个函数里返回]，151 页）。

until

u 在当前栈帧上继续执行直到越过当前行的源代码行。这个命令用来避免多次单步执行一个循环。和 next 命令，除了

until 在遇到一个跳转时，until 会自动继续执行直到程序计数器比跳转地址大的时候。

这就意味着在你用单步执行达到循环结束的时候，until 会使程序继续执行直到循环结束。相反，在循环结束时，

next 命令只是简单的又从循环开始单步执行，这就使得你要进行下次单步执行循环。

until 总是在程序试图从当前栈帧中退出的时候中断执行。

如果机器指令的顺序和源代码的不相符合的话，until 可能会产生和预期不大一样的结果。例如，下列调试会话摘录里，f(frame)命令显示了程序执行在杭 206 中断；但用 until，得到行 195：

(gdb) f

#0 main (argc=4, argv=0xf7fffae8) at m4.c:206

206 expand_input();

(gdb) until

195 for (; argc > 0; NEXTARG) {

这是因为执行效率的缘故，编译器在循环尾产生结束测试的代码，而不是在循环开头--即使 C for 循环的测试实在循环体前。until 命令看上去就像会单步执行回循环的开头；然而，它不是真的要回到前面的代码里去--不是依照实际的机器代码。

不带参数的 until 命令将使用单个指令的单步执行方式，因此比带参数的 until 要慢。

until location

u location

继续执行直到程序执行到指定的位置，或者从当前栈帧返回。location 可以是在 7.2 节[指定位置]里讨论的任意一种形式。这个命令形式使用临时断点，因此比不带参数的 until 命令要快。只有在这个指定的位

置在当前帧上时，它才会真的被执行到。这就意味着 until 可以用来跳过函数嵌套调用。如下面函数所示，如果当前位置是 96 行，执行命令 until 99 会执行程序到 99 行，在内从调用返回的时候。

```
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位置 location。参数是必须的，可以是可以是在 7.2 节[指定位置]里讨论的任意一种形式。执行会在从当前栈帧上退出时中断。advance 和 until 相似，但 advance 不会跳出函数嵌套调用，而且目标位置不必要在当前帧上。

stepi

stepi arg

si 执行一个极其指令，接着中断然后返回到调试器。

在单步执行一个机器指令的时候，执行‘display/i $pc’很有用。这使得每次程序中断的时候，GDB 在下一个指令要执行的时候自动显示这个指令。参见 8.6 节[自动显示]，81 页。

参数是重复次数，和 step 一样。

nexti

nexti arg

ni 执行下一个机器指令，但如果是一个函数调用的话，执行到这个函数返回。

参数是重复次数，和 next 一样。

5.3 信号

信号是程序里发生的异步事件。操作系统定义了信号种类，并命名这些信号，给这些信号编号。例如，在 Unix 里 SIGINT 是程序在你输入一个中断字符（通常是 Ctrl-c）的时候得到的信号;SIGSEGV 是程序在程序引用了离当前所有用到的内存区域都很远的内存时得到的信号；SIGALRM 在定时器超时的时候发生

（只有在程序定义了定时器的时候才发生）。

某些信号，包括 SIGALRM，是程序的普通组成部分。其它的，像 SIGSEGV，指示发生了错误；这些信号是致命的（它们会立即结束你的程序），如果你没有预先定义好信号处理函数的话。SIGINT 不表示程序发生错误，但常常也是致命的，所以可以用来执行中断的目的：杀死进程。

GDB 能够察觉程序里出现的任何信号。可以预先设定 GDB 如何应对每一种信号。

通常，GDB 会将非错误性的信号悄悄的传进程序（因此不会干涉它们在程序里的功能角色），但在一个错误信号发生时立即中断程序。可以用 handle 命令来改变这些设置。

info signals

info handle

打印所有信号和 GDB 被设置如何设置来处理它们。可以用这个命令来查看定义好的信号的编号。

info signals sig

和 info handle 相似，但只打印指定信号的信息。info handle 是 info signals 的别名。

handle signal [keywords...]

改变 GDB 处理信号的方式。signal 可以是一个信号的编号或则会名字（开头带或不带'SIG'）；信号编号形式“log-high”列表；或者是'all'，代表所有的信号。可选参数 keywords，如下所述，表示要设置什么。

handle 命令带有的参数 keywords 可以缩写。它们是：

nostopGDB 在信号发生的时候不中断程序。它可以打印一个信息告诉你有信号出现了。

stopGDB 在信号出现的时候中断程序。意味着 print 关键字也同样。

printGDB 在信号出现的时候打印一个信息。

noprintGDB 在信号出现的时候不打印任何信息。意味着和 nostop 关键字一样。

pass

noignore

GDB 让程序能得到这个信号。程序可以处理信号，或者如果信号是致命而又没有处理的话可以终止信号。pass 和 noignore 同义。

nopass

ignoreGDB 不允许程序得到信号。nopass 和 ignore 同义。

在信号中断程序时，在程序继续执行前信号都是不可见的。如果在这个时间点上 pass 对此信号起作用了的话，程序就能看到这个信号。换言之，GDB 报告信号后，你可以用 handle 命令带 pass 或者 nopass

来控制程序在继续执行的时候是否可见到信号。

缺省是设置为 nostop，noprint，非错误性信号如 SIGALRM 是 pass，错误性信号是 SIGWINCH 和 SIGCHLD，stop，print。

用 signal 命令可以防止程序看见一个信号，或者看到通常不可见的信号，或者在任意时间给程序一个信号。例如，如果程序由于某种内存引用错误导致的中断，你可能把正确的值存储进错误的变量然后继续执行，希望看见更多的执行；

但程序可能由于看到了致命的信号的缘故，立即就终止了；要防止这样的情况，用'signal 0'继续执行。参见 14.3 节[向程序发一个信号]，151 页。

5.4 中断和开始多线程程序

如果程序具有多个线程（参见 4.9 节[调试多线程程序]，31 页），可以选择是否在所有线程上设置断点，或者在特定的线程上设置断点。

break linespec thread threadno

break linespec thread threadno if ...

linespec 指定源代码行；有多种方式来指定（参见 7.2 节[指定位置]，68 页），不过其效果都是指定某些源代码行。

在断点命令里使用限定语'thread threadno'可以指定 GDB 只在特定线程执行到这个断点时中断程序。threadno 是 GDB 分配的线程标识号，'info threads'显示的第一列就是。

如果设定断点时没有指定'thread threadno'的话，那么断点就适用于程序里所有的线程。

thread 限定词也可用于条件断点；这种情况下，把'thread threadno'放在断点条件前，像这样：

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

无论何时程序在 GDB 里因为何种原因而中断，所有的线程执行都将中断，而非仅仅是当前线程。这就可以让你检查程序所有的状态，包括线程切换，而不用担心事情会悄无声息的发生改变。

然而有一个不幸的边际效应。如果一个线程由于断点而中断，或者因为别的缘故，而另一个线程由于一个系统调用而阻塞住了，那么这个系统调用可能会过早的返回。这是多线程和某些信号相互影响的结果，这些信号是 GDB 用来实现断点和别的用来中断执行的事件。

要处理这个问题，程序应当检查每个系统调用的返回值并作适当的反应。这总是好的编程风格。

例如，不要写这样的代码：

sleep (10);

sleep 调用可能过早返回，由于其它的线程中断或别的原因的缘故。

相反，要这样写：

int unslept = 10;

while (unslept > 0)

unslept = sleep (unslept);

一个系统调用可过早返回，而系统还在执行这个调用。但 GDB 真的会导致程序和不在 GDB 里的行为不一样。

而且，GDB 在线程库里用内部断点来监视某些事件，比如线程创建和线程销毁。这些事件发生时，另一个线程里系统调用可能过早返回，即使程序没有中断。

相反，重启程序时，所有线程都会开始执行。即使用单步命令如 step 或 next 来执行程序时都是这样的。

特别的，GDB 不能在锁步下单步执行所有线程。因为线程调度由调试操作系统决定（不是由 GDB 控制），其它的线程可能比当前线程完成一次单步执行多执行一些。更进一步说，在程序中断的时候，通常其它的线程中断在一个语句的中间，而不是在一个语句的边界上。

在继续或者单步执行之后，你甚至可能发现程序在另一个线程里中断。在其它线程执行到一个断点，信号，或者一个异常发生,或者当前线程执行完成的时候，都可能发生这个现象。

在某些操作系统里，可以锁住操作系统的调度器来直让一个线程运行。

set scheduler-locking mode

设置调度器锁定模式。 如果是 off,那么就不锁定，任何线程可以在任何时候执行。如果是 on，那么只有当前线程可以运行。step 模式优化单步执行。在单步调试时，step 用抢占当前线程的方式来阻止其他线程抢占控制权。单步调试的时候，其它线程很少（或者从不）有机会来执行。用'next'来执行一个函数调用的话，它们有更有可能运行，并且在用'continue'、'until'或者'finish'之类的命令的话，它们就完全自由的运行了。不过，除非另一个线程在它自己的时间片里执行到了一个断点，它们从不从你正在调试的线程抢夺 GDB 的控制权。

show scheduler-locking

显示当前调度器的锁定模式。

第6章 检查栈

程序中断后，首先需要知道在哪里中断的和如和到此的。

每次程序执行了一个函数调用，调用信息就产生了。这些信息包含了程序里的调用位置，调用的参数和被调用函数的本地比昂两。这些信息被存储在一块被称为堆栈帧的数据里。堆栈帧是在成为调用栈的内存区域里分配的。

程序中断的时候，GDB 提供的堆栈检查命令可以看见所有此类信息。

GDB 会选择其中的一个堆栈帧，GDB 的很多命令也隐式地引用这个帧。特别的，在查看程序里的变量值的时候，这个值就在这个选定的帧里。GDB 提供了特殊的命令来要选择你感兴趣的堆栈帧。参见 6.3 节[选择一个帧]，64 页。

在程序中断时，GDB 自动选择当前执行的堆栈帧，并用和 frame 命令那样类似的描述这个帧，只不过简明一点。

6.1 堆栈帧

调用栈划分为连续的区块，称为堆栈帧，或者简称为栈；每个帧是一个函数调用另一个函数的相关数据。帧包含了传递给被调用函数的参数，这个函数的本地变量和这个函数执行的地址。

在程序开始的时候，堆栈只有一个帧，那是主函数 main 的。这个帧称为初始帧或者最外层的帧。每当一个函数被调用了，就产生一个新的堆栈帧。函数返回时，这个调用所属的帧就被销毁了。如果是递归函数，那么同一个函数就可能有多个帧。当前正在执行的函数调用的帧称为最内层的帧。这是最近创建的帧，同时还有别的帧存在。

程序内部，堆栈帧用地址来标识。一个堆栈帧有许多字节组成，每个字节都有自己的地址；每种计算机都有自己的方式选择作为堆栈帧的地址的一个字节。程序在这个帧里执行时，通常这个地址保存在成为帧指针寄存器里（参见 8.10 节[寄存器]，90 页）。

GDB 为所有现存的堆栈帧编号，从最内层帧 0 开始，1 是这个函数调用的帧，以此类推。这些编号并不真正在存在于程序里；它们是由 GDB 分配用于 GDB 命令来区分堆栈帧的。

某些编译器提供编译不带函数堆栈帧的方法（例如，GCC 选项‘-fomit-frame-pointer’产生不带帧的函数）。在频繁调用库函数时，可以用这个技术来节省建立帧的时间。对于这类函数调用的处理，GDB 的

能力有限。如果最内层的函数调用没有堆栈帧的话，GDB 仍然认为它是由一个单独的堆栈帧，通常编号为 0，因此可以正确的追踪函数调用链。然而，GDB 不在堆栈的其它地方提供无帧函数。

frame args

frame 命令可以从一个堆栈帧转到另一个，并打印所选的堆栈帧。args 可以是帧地址或是帧号。如果不带参数，frame 打印当前堆栈帧。

select-frame

select-frame 命令可以从一个堆栈帧转到另一个而不打印这个帧。是 frame 的安静版本。

6.2 回溯

回溯是关于程序如何达到所处位置的概要。每行显示一个帧，对于多个帧的情况，从当前执行的帧（0 帧）开始，接下来是它的调用者（1 帧），以此类推。

backtrace

bt 打印整个堆栈的回溯：每帧一行。

可以用输入系统中断字符在任意时间中断回溯，通常是 Ctrl-c.

backtrace n

bt n 类似的，但只打印最内 n 层帧。

backtrace full

bt full

bt full n

bt full -n

也打印本地变量。n 是要打印的帧的数量，如上所述。

where 和 info stack 是 backtrace 的别名。

在多线程程序里，GDB 默认只显示当前线程的回溯。要显示多个或所有线程的回溯，用 thread apply 命令（参见 4.9 节[线程]，31 页）。例如，输入 thread apply all backtrace，GDB 会显示所有线程的回溯；在调试一个多线程程序的 core dump 的时候是很方便的。

回溯里的每一行显示了帧号和函数名。程序计数器的值也会显示--除非用了 set print address off。回溯也显示源代码文件名和行号，而且函数的参数也会显示。如果是在那行代码的开头，程序计数器的值会被忽略不显示。

下面是一个回溯的例子。用命令'bt 3'产生的，因此显示最内 3 层的堆栈帧。

#0 m4_traceon (obs=0x24eb0, argc=1, argv=0x2b8c8)

at builtin.c:993

#1 0x6e38 in expand_macro (sym=0x2b600) at macro.c:242

#2 0x6840 in expand_token (obs=0x0, t=177664, td=0xf7fffb08)

at macro.c:71

(More stack frames follow...)

没有在堆栈帧里存储的参数，其值用‘<value optimized out>’来替代。

如果需要显示这些优化掉的参数值，或者用其它依赖于这个参数的变量来推论，或者不带优化的重新编译程序。

大多数程序都有标准的用户入口点--在此处系统库和启动代码转化为用户代码。对于 C 是 main 函数（注 1）。GDB 要是发现回溯里有入口函数，就会结束回溯，避免跟踪到高度系统相关（或通常不关心的）的代码。

要是需要检查启动代码，或者限制回溯到层次数量，可以改变其行为：

set backtrace past-main

set backtrace past-main on

回溯会在入口点处继续跟踪下去。

set backtrace past-main off

回溯在入口点处终止。默认行为。

show backtrace past-main

显示当前用户入口点回溯规则。

（注 1，嵌入式程序（所谓的“自举”环境）不要求在入口点有 main 函数。甚至可以有多个入口点）

set backtrace past-entry

set backtrace past-entry on

回溯会在应用程序的内部入口点继续跟踪下去。这个入口点是由连接器在链接程序的时候

编码的，很可能是在用户入口点 main 函数（或者是相应的）之前被调用。

set backtrace past-entry off

回溯会在程序的内部入口点处终止。默认行为。

show backtrace past-entry

显示当前内部入口点回溯规则。

set backtrace limit n

set backtrace limit 0

显示回溯层为 n。0 代表不限制。

show backtrace limit

显示当前回溯层限制。

6.3 选择堆栈帧

大多数用来检查堆栈和数据的命令作用于此时所选择的堆栈帧上。有多个命令来选择堆栈帧；都以打印一个堆栈帧的简明描述作为结束。

frame n

f n 选择编号为 n 的帧。回忆一下，0 帧时最内层（当前执行）的帧，1 帧时最内层帧所调用的，以此类推。最高编号的帧时 main 函数的。

frame addr

f addr 选择在地址 addr 上的帧。这个命令在堆栈帧链被 bug 损坏的时候很有用，使得 GDB 可以为所有帧编号。另外，要是程序有多个堆栈的时候，要在它们之间切换就很有用了。

在 SPARC 架构里，frame 需要恋歌地址来选定一个绝对帧：一个帧指针和一个堆栈指针。

在 MIPS 和 ALPHA 架构里，需要两个地址：一个堆栈指针和一个程序计数器。

在 29k 架构里，需要 3 个地址：一个寄存器堆栈指针，一个程序计数器和一个内存堆栈指针。

up n 在堆栈里上移 n 帧。对于正数 n，向外层的帧移动，更高编号的帧，存在更长时间的帧。n 默认是 1.

down n 在堆栈里下移 n 帧。对于正数 n，向内层的帧移动，更低编号的帧，新近创建的帧。缩写 down 为 do。

这些命令都已打印两行描述堆栈帧的输出结束。第一行显示帧号，函数名，参数，源代码文件和和行号。第二行显示源代码文本。

例如：

(gdb) up

#1 0x22f0 in main (argc=1, argv=0xf7fffbf4, env=0xf7fffbfc)

at env.c:10

10 read_input_file (argv[i]);

在此输出后，不带参数的 list 命令会以此帧的执行点为中心，显示 10 行代码。可以用 edit 命令来编辑此执行点的代码。详情参见 7.1 节[打印源代码行]，67 页。

up-silently n

down-silently n

这两个命令分别是 up 和 down 的变体；不同之处在于它们悄悄的处理，不输出新帧的信息。

主要是为 GDB 命令脚本用的，脚本的输出可能是不必要和恼人的。

6.4 堆栈帧信息

还有其它几个命令可以打印选定堆栈帧的信息。

fram

f 要是不带参数，命令不改变帧，但是打印当前选定堆栈帧的简明描述。可以缩写为 f。带参数的话用来选择堆栈帧。参见 6.3 节[选择堆栈帧]，64 页。

info frame

info f 这个命令打印选定帧的文本描述，包括：

- 帧地址
- 下一个帧的地址（被这个帧调用的）
- 上一个帧的地址（这个帧的调用者）
- 这个帧对应的源代码的语言
- 帧参数的地址
- 帧的本地变量的地址
- 帧里的程序计数器（调用者帧的执行地址）
- 在帧里保存的计数器

要是发生了导致堆栈格式不能以通常的方式查看的错误的话，文本描述就很有用了。

info frame addr

info f addr

打印在地址 addr 上的帧的文本描述，但不选择此帧。此帧不会被此命令选中。要求和 frame 命令相

同的地址（对于某些架构，可能要求多个）。参见 6.3 节[选择堆栈帧]，64 页。

info args

打印选定帧的参数，每个参数一行。

info locals

打印选定帧上的本地变量，每个一行。选定帧的执行点上可见的所有变量（声明为静态或自动的均可）。

info catch

打印当前帧的执行点上的异常处理函数的列表。要查看其它异常处理函数，访问相应的帧（用 up,down 或者 frame 命令）；接着输入 info catch。参见 5.1.3 节[设置捕获点]，47 页。

第7章 检查源文件

由于程序里记录的调试信息告诉 GDB 程序是由哪些文件编译的，GDB 可以打印程序各部分源文件。程序中断时，GDB 同时自动打印是在哪一行上中断的。同样，当选择一个堆栈帧时（参见 6.3 节[选择帧]，64 页），GDB 也打印那个帧上的执行是在哪一行里中断的。用显式的命令可以打印源文件别的信息。

如果通过 GNU Emacs 接口来用 GDB，可以优先选择 Emacs 工具来查看源代码；参见 23 章[在 GNU Emacs 里用 GDB]，233 页。

7.1 打印源代码行

要打印源文件里的代码行，用 list 命令（缩写为 l）。缺省的，一次打印十行。有几个方式可以指定你希望打印文件的哪部分；完整列表，参见 7.2 节[指定位置]，68 页。

下面是命令 list 最常用的形式：

list linenum

以当前行为中心，打印当前源文件 linenum 行。

list function

以函数 function 开头为中心打印源文件。

list 打印更多行。如果用 list 命令打印过源文件行，list 命令将会在打印的最后一行代码下接着打印；否则，如果最近打印的源代码行是单独的一行，作为显示堆栈帧的一部分的话，list 命令将已这一行为中心开始打印文件代码。

list -只打印最近已打印的代码行前的代码。

缺省的，用上述形式的 list 命令，GDB 打印十行源代码。可以用 set listsize 改变：

set listsize count

设置 list 命令显示 count 行源代码（除非 list 命令明确指定其它的数字）。

show listsize

显示 list 打印行的数量。

用回车键省略参数的重复 list 命令，和只输入 list 相等。和列出相同的代码行相比更有用。参数是'-'的 list 命

令是例外；这个参数是用来重复执行的，每次执行都会在源文件里向上打印。

总之，list 命令期待用户提供 0，1 或者 2 的行定义。行定义指定源代码行；有多种方式来制定（参见 7.2 节[指定位置]，68 页），不过其效果都是制定某些源代码行。

下面是 list 命令的完整的参数描述：

list linespec

以 linespec 为中心打印源代码。

list first,last

从 first 开始打印到 last。两个参数都是行定义。当 list 命令有两个行定义时，并且第二个行定义的源文件省略了的话，两个行定义指的是在同一个文件里的代码。

list ,last

打印到 last 行。

list first,

从 first 开始打印

list +只打印最近打印行后的代码

list -只打印最近打印行前的代码

list 如前所述。

7.2 指定位置

有多个 GDB 命令接受指定程序代码位置的参数。由于 GDB 是源代码级的调试器，位置通常指的是源代码里的某一行；因此，位置通常是指行定义。

下面是 GDB 所能识别的代码位置的指定方式：

linenum 指定当前源文件的行数量 linenum

-offset

+offset 指定从当前行偏移 offset。对于 list 命令，当前行是最经打印过的；对于断点命令，是当前选定的堆栈帧上的执行中断处（堆栈帧的描述，参见 6.1 节[帧]，61 页）。要是用做 list 命令的两个行定义的第二个参数，指定的是从第一个行定义开始的向上或向下的偏移。

filename:linenum

指定源文件 filename 的行号。

function

指定行从函数 function 为开始。例如，在 C 里，行是开括号{。

filename:function

指定行从文件 filename 里的函数 function 开始。要是有多个文件里有相同名字的函数的话，只需要用文件名加上函数名就可以避免混淆了。

*address

指定程序地址 address。对于行导向的命令，例如 list 和 edit，这个参数指定位置为 address 的源代码行。

对于 break 和其它断点导向的命令，这个参数可以在不带调试信息或没有源文件的程序部分里设置断点。

这里 address 可以是当前工作语言（参见 12 章[语言]，119 页）的任意有效的指定代码位置的表达式。另外，GDB 扩展了用于位置的表达式的语义，以此包含在调试过程中经常碰到的情况。下面是 address 的各种形式：

expression

当前工作语言的任意有效表达式。

funcaddr

函数的地址或源自名字的过程的地址。在 C，C++，Java，Object-C，Fortran，微指令和汇编里这个参数是函数名 function（或者是一个有效表达式的一个特例）。在 Pascal 和 Modula-2，是&function。在 Ada 里，是 function'Address（虽然 Pascal 形式也可用）。

这中形式指定函数第一个指令的位置，这个位置是在堆栈帧和参数建立前的。

'filename'::funcaddr

和上面的 funcaddr 类似，不过还指定了源文件名。这个形式在只指定函数名会造成歧义的时候很有用，例如，如果在不同的文件里有多个函数都具有相同的名字时。

7.3 编辑源文件

要编辑源文件里的内容，用 edit 命令。你选择的编辑程序将被调用，并将程序里的当前行设置为激活行。另外，有几种方式可以指定你希望打印文件的哪部分，如果你要想看程序的其它部分的话。

edit location

编辑指定为 location 的源文件。编辑从 location 开始，例如，在指定文件的指定行赏。参见 7.2 节[指定位置]，68 页，所有位置参数的可能形式；下面是 edit 命令最常用的形式：

edit number

将行 number 作为激活行编辑文件。

edit function

编辑包含函数 function 的文件，在其定义的开头开始编辑。

7.3.1 选择编辑器

GDB 可以定制你所期望的任意编辑器（注 1）。缺省的，是'/bin/ex'，但你可以在运行 GDB 前，设置环境变量 EDITOR 来改变。例如，要配置 GDB 实用 vi 编辑器，在 sh shell 里用下面的命令：

EDITOR=/usr/bin/vi

export EDITOR

gdb ...

（注 1，唯一的限制在于，你的编辑器要识别如下命令行语义：ex +number file 可选的数值+number 指定文件的行号。）

或者在 csh shell 里，

setenv EDITOR /usr/bin/vi

gdb ...

7.4 搜索源文件

有两个命令可以用正则表达式在当前文件里搜索。

forward-search regexp

search regexp

命令'forward-search regexp'检查每一行，从最近列出的行开始，查找正则表达式 regexp 的匹配项。命令列出找到的行。也可以用同义词'search regexp'或缩写命令为 fo.

reverse-search regexp

命令'reverse-search regexp'检查每一行，从最近列出的行往后，查找正则表达式 regexp 的匹配项。命令列出找到的行。缩写为 rev。

7.5 指定源文件目录

可执行程序有时并不记录源文件编译的目录，只记录名字。即使它们记录了，目录也可能在编译和调试期间被移动了。GDB 一个目录列表来搜索源文件；这些目录称为源代码路径。每次 GDB 需要源文件时，它都会尝试在所有的目录列表里搜索，以它们在列表里的顺序进行，直到 GDB 找到所查询的文件。

例如，假设一个可执行文件引用了文件'/usr/src/foo-1.0/lib/foo.c'并且我们的源代码路径是'/mnt/cross'。

GDB 首先会从字面上去查找；如果失败了，会在'/mnt/cross/usr/src/foo-1.0/lib/foo.c'里查找；如果还是失败，会查找'/mnt/cross/foo.c'；如果再失败的话，会打印一个错误消息。GDB 不会查找部分的路径名，例如'/mnt/cross/src/foo-1.0/lib/foo.c'。类似的，源代码路径的子目录也不会搜索：如果源代码路径是'/mnt/cross',并且二进制文件引用了'foo.c'，GDB 不会去'/mnt/cross/usr/src/foo-1.0/lib'路径下查找。

简单文件名，带先导路径的相对文件名，包含点的文件名（点文件）等等，都如上所述查找；比如，如果源代码路径是'/mnt/cross'，并且像'../lib/foo.c'这样记录的话，GDB 会首先尝试'../lib/foo.c'，接着'/mnt/cross/../lib/foo.c'，最后是'/mnt/cross/foo.c'。

注意，可执行文件搜索路径不用于定位源代码文件。

无论何时你重新设置或组织了源代码路径，GDB 会清除任何缓存的关于路径和文件行的信息。

当你启动 GDB 时，它的源代码路径只包含'cdir'和'cwd'，且以此顺序排列。要增加其它目录，用 directory 命令。

搜索路径用于查找程序源文件和 GDB 脚本文件（读用'-command'选项和'source'命令）。

除了源文件路径，GDB 提供了一些命令来管理源代码路径列表替换的规则。替换规则说明了在编译和调试期间目录被移动的情况下，如何来重写存储于程序调试信息里的源代码目录。一个规则是由两个参数组成的，第一个参数指定需要重写的路径，第二个参数指定这个路径如何重写。在[设置替换路径]，72 页里，我们将这两个部分命名为 from 和 to。GDB 简单的用 to 将 from 的源文件名的开头目录部分替换掉，并用替换后的结果来搜索源文件。

还用前例，假设'foo-1.0'目录树已从'/usr/src'移动到'/mnt/cross'，然后让 GDB 用'/mnt/cross'替换所有路径的'/usr/src'。首先在路径'/mnt/cross/foo-1.0/lib/foo.c'查询，替代原来的路径'/usr/src/foo-1.0/lib/foo.c'。

要定义一个源代码路径替换规则，用 set substitute-path 命令（参见[设置替换路径]，72 页）。

要避免意外的替换结果，规则只在目录名的 from 部分以目录分隔符结尾的情况下才应用。例如，用'/usr/source'替换'/mnt/cross'的话，会得到'/usr/source/foo-1.0'，而不是'/usr/sourceware/foo-2.0'。并且替换规则只应用于目录名的开头部分，这个规则不会得到'/root/usr/source/baz.c'的结果。

在很多情况下，可以用 directory 命令来达到相同的效果。然而，在源文件分布于一个复杂的，带有多个子目录

的目录树时，set substitute-path 会更有效率。用 directory 命令的话，用户需要添加项目的每一个子目录。如果将整个目录移动且保持内部代码组织的话，那么 set substitute-path 允许你只用一个命令就可以将调试器导向到所有的源文件。

set substitute-path 也不仅仅只是一个快捷命令。源代码路径只用于原目录下的文件不再存在的情况下。另一方面，set substitute-path 修改调试器行为模式来在重写的位置上搜索文件。所以，如果有任何原因导致源代码文件相对于可执行程序的位置发生了改变，替换规则是唯一可以通知 GDB 新位置的方法。

directory dirname ...

dir dirname ...

在源代码路径前加上目录 dirname。可以将几个目录名传递给这个命令，用':'分隔（在 MS-DOS 和

MS-Windows 是';'，':'通常是一个绝对路径名的一部分），或者用空格分隔。可以指定已存在的目录；这会将此目录前移，GDB 就能更快的搜索到它了。

可以用字符串'$cdir'来指代编译目录（如果已记录的话），'$cwd'指代当前工作目录。'$cwd'和'.'不相

同---前者记录在 GDB 调试会话期间变动的当前工作目录，后者则在你将一个目录添加进源代码路径时立即展开为当前目录。

directory

将源代码路径重置为默认值（Unix 系统下是'$cdir:$cwd'）。这个命令要求确认。

show directories

打印源代码路径：显示包含那个目录。

set substitute-path from to

定义一个源代码路径替换规则，并将其填加到当前替换规则列表的尾部。如果有相同的 from 规则存在的话，那么旧的规则就会被删除。

例如，如果文件'/foo/bar/baz.c'移动到'/mnt/cross/baz.c',那么命令

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

告诉 GDB 用'/mnt/cross'替换'/usr/src',这就可以让 GDB 查找到'baz.c',即使它已经移走了。

如果定义了多个替换规则，那么 GDB 会以规则定义的顺序一个接一个的计算它们。如果有的话，第一个匹配项就会进行替换。

例如，如果我们输入了下列命令：

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

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

GDB 会用第一个规则将'/usr/src/include/defs.h'用'/mnt/include/defs.h'替换。不过，它会用第二个规则将'/usr/src/lib/foo.c'用'/mnt/src/lib/foo.c'替换。

unset substitute-path [path]

如果指定了 path 的话，在当前替换规则列表里搜索要重置的规则。如果找到的话就删除之。如果没有找到的话，调试器会输出一个警告信息。

如果没有指定 path 的话，那么所有的替换规则都将被删除。

show substitute-path [path]

如果指定了 path，那么打印打印源代码路径替换规则，如果有的话。

如果没有指定 path，那么打印所有的替换规则。

如果源代码路径混杂着一些不再有用的目录的话，GDB 可能在某些情况下造成错误的代码版本的混淆。可以用下列命令来纠正这种错误：

1.用不带参数的 directory 命令来重置源代码路径为默认值。

2.用带正确参数的 directory 来添加你需要的目录。可以用一个命令将所有的路径添加。

7.6 源代码和机器代码

可以用命令 info line 将源代码映射到程序地址上（反过来一样），命令 disassemble 可以显示一定范围的机器指令。在 GNU Emacs 模式下运行时，info line 命令会引起箭头指向指定的行。而且，info line 打印符号形式的地址，也打印 16 进制的地址。

info line linespec

打印指定的源代码行的编译代码，从开始到结尾的地址。可以指定源代码行，7.2 节[指定位置]，68 页里讨论的任意形式。

例如，我们可以用 info line 来查找函数 m4_chagequote 的第一行的目标代码：

(gdb) info line m4_changequote

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

我们也可以查询（用*addr 作为 linespec 的形式）哪一行源代码对应一个特定的地址：

(gdb) info line *0x63ff

Line 926 of "builtin.c" starts at pc 0x63e4 and ends at 0x6404.

在 info line 后，x 命令的缺省地址就变为这行的开头地址，所以'x/i'足以开始检查机器代码（参见 8.5 节[检查内存]，79 页）。而且，这个地址也存储与变量$_里（参见 8.9 节[便利的变量]，89 页）。

disassemble

这个命令将一段内存作为机器指令转储。缺省的内存范围是选定堆栈帧的程序计数器所代表的函数。单个参数的话是程序计数器的值；GDB 转储值个值附近的函数。两个参数指定要转储的地址范围（第一个是开始，第二个是结束）。

下面的例子显示了反汇编一个 HP PA-RISC 2.0 上的一段代码：

(gdb) disas 0x32c4 0x32e4

Dump of assembler code from 0x32c4 to 0x32e4:

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

0x32c8 <main+208>: ldw 0x22c(sr0,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 assembler dump.

某些架构有多个通用的指令助记符或者同义词。

对于动态链接的和使用共享库的程序，调用函数或位于共享库的分支位置的指令可能显示伪地址--这个地址是重定位表的位置。在某些架构里，GDB 可以将这些伪地址映射到函数名上。

set disassembly-flavor instruction-set

用 disassemble 或 x/i 命令反汇编程序时，选择指令集。

目前这个命令只在 Intel x86 族平台上定义。可以设置指令集为 intel 或 att。默认是 att，基于 x86 系统的 Unix 汇编器默认使用 AT&T 风格。

show disassembly-flavor

显示当前反汇编风格的设置。

第8章 第八章 查看数据

在程序里查看数据的常用方式是用 print 命令（缩写为 p），或者用它的同义命令 inspect。print 命令会计算和打印用程序语言写的表达式的值（参见 12 章[不同语言下使用 GDB]，127 页）。

print expr

print /f expr

expr 是表达式（用程序语言写的）。缺省情况下，expr 的值会以它的数据类型相近的格式打印；也可以用'/f'选择不同的格式，f 是格式描述符；参见 8.5 节[输出形式]，85 页。

print

print /f

如果省略了 expr，GDB 将显示最后一次的值（从值的历史记录里；参见 8.9 节[值历史记录]，96 页）。这个命令允许用户方便的换一种格式查看相同值。

用 x 命令可以在更低层次上查看数据。x 命令检验一个指定位置上的数据，并且以指定格式打印出来。参见 8.6 节[查看内存]，87 页。

如果对数据类型感兴趣的话，或者想知道一个结构体或类里的一个域是如何声明的话，用 ptype expr 命令取代 print 命令就可以做到了。

参见 13 章[查看符号表]，151 页。

8.1 表达式

print 很许多其他 GDB 命令一样接受一个表达式作为参数并且计算它的值。在 GDB 里，程序里定义的任意类型的常量，变量或者操作符都是有效的表达式。包括条件表达式，函数调用，类型转换和字符串常量。也包括预定义宏，如果在编译程序时包含了的话；参见 4.1 节[编译]，25 页。

GDB 支持用户输入的表达式里包含数组常量。语法是{element,element...}。例如，可以用命令 print {1,2,3}来在内存里建立一个数组，而这个数组是目标系统里在自由堆里分配的。

由于 C 应用的如此广泛，因此本手册里的大多数例子里的表达式都是用 C 写的。关于在其它语言里如何使用表达式的信息，参见 12 章[不同语言下使用 GDB]，119 页。

在这节里，讨论可用于 GDB 表达式的与编程语言无关的操作符。

GDB 支持下列操作符，另外还有其它编程语言可以通用的操作符：

@'@'是二进制操作符，将一块内存作为数组。更多信息，参见 8.3 节[伪数组]，77 页。

::'::'可以指定一个文件或函数里定义的变量。参见 8.2 节[程序变量]，76 页。

{type} addr

引用存储于 addr 位置上的 type 类型的对象。addr 可能是一个表达式，其值是一个整型或是指针（但需要圆括号来进行类型转换）。

无论在 addr 地址上存储的数据是何种类型，这个用法都是允许的。

8.2 程序变量

在程序里，最常用的表达式类型是使用变量的名字。

表达式里的变量应理解为存储与一个特定的堆栈帧里（参见 6.3 节[选择一个帧]，64 页）；变量可以是下列两种：

- 全局变量（或是文件里的静态变量 file-static）
- 根据编程语言的变量生存规则，在当前执行的堆栈帧上是可见的变量

这意味着在函数

```
foo (a)
int a;
{
bar (a);
{
int b = test ();
bar (b);
}
}
```

程序在函数 foo 里执行时，用户可以检查和使用变量 a，但只能在程序执行于 b 声明的块内（函数 bar）使用或检查变量 b。

有一个例外：可以引用生存期是单个文件的变量或函数，即使当前执行点不在这个文件上。不过，很有可能有多个变量或函数有相同的名字（在不同的源文件）。如果那样的话，引用重名的变量可能会有预想不到的效果。如果需要，可以指定一个特定函数或文件的静态变量，用双冒号（::）标记：

file::variable

function::variable

这里的 file 或 function 是这个静态变量的上下文的名字。对于文件名，可以用两个单引号将文件名包起以让 GDB 将其作为一单个词分析--例如，要打印'f2.c'文件里定义的全局变量 x:

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

与在 C++相同符号的用法相比，C 的'::' 的用法很少有冲突。GDB 也支持 C++生存期操作符。

警告：偶尔的，一个本地变量可能在函数的某些点上显示错误的值--在进入一个新的生存期后和在离开前。

在用机器指令单步跟踪程序的时候，可能碰到这个问题。这是因为，在大多数机器里，需要多于一个的指令才能建立一个堆栈帧（包括本定变量定义）；如果你是用机器指令来单步跟踪的话，变量就可能显示错误的值，直到堆栈帧完全建立为止。

在退出时，通常也需要多个机器指令才能销毁堆栈帧；在你开始单步执行通过这组指令的过程中，本地变量的定义可能已经消失了。

这个问题也可能在编译器做了显著优化的时候碰到。要确保总是得到精确的值的话，在编译时关闭优化选项。

编译器优化结果所带来的另一个潜在影响是将没有用到的变量优化掉了，或者叫爱那个变量存储于寄存器（而不是内存地址上）。由于依赖于编译器提供的调试信息格式对于此类问题的支持，GDB 可能不能显示这些本地变量的值。如果真的这样，GDB 会打印类似如下的消息：

No symbol "foo" in current context.

要解决这些问题，要么不带优化选项重新编译，要么使用一个不同的调试信息格式，如果编译器支持多种格式的话。例如，GCC，GNU C/C++编译器，通常支持'-gstabs+'选项。'-gstabs+'产生优于普通调格式（如 COFF）的调试信息。可以用 DWARF 2（'-gdwarf-2'）,这也是一个有效的调试信息格式。参见节"调试程序或 GCC 的选项"。参见 12.4.1 节[C 和 C++],123 页，更多关于最佳 C++程序调试信息格式的信息。

如果需要打印一个 GDB 未知其内容的对象，例如，由于调试信息没有完全说明它的数据类型，GDB 会打印'<incomplete type>'。

更多信息，参见 13 章[符号]，143 页。

字符串是定义为无符号的 char 数组。signed char 或 unsigned char 会以 1 字节宽度的整形数组打印。由于 GDB 定义字符串类型 type'char'为无符号类型，-fsigned-char 或-funsigned-charGCC 选项不会起效。对于程序代码

```
char var0[] = "A";
signed char var1[] = "A";
```

可以在调试时得到下列信息

```
(gdb) print var0
$1 = "A"
(gdb) print var1
$2 = {65 'A', 0 '\0'}
```

8.3 伪数组

打印几个在内存里连续的相同类型的对象，常常是很有用的；数组的一节，或一个动态决定大小的数组，在程序里只有一个指针存在。

用二进制操作符'@'将一个连续的内存区域作为伪数组,可以达到这个目的。'@'的左操作数应该是目标数组的第一个元素，并且是一个单独的对象。右操作数应该是目标数组的长度。结果是整个数组的值，其元素都是左操作数的类型。第一个元素是左操作数；第二个元素在内存中数紧邻着第一个元素，依此类推。下面是一个例子。如果程序

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

可以用下面命令打印数组的内容

p *array@len

'@'的左操作数必须存在于内存中。用'@'打印的数组值和用其它下标索引得到的值一样，并且会在表达式里强制转换为指针。伪数组常常通过值历史在表达式里出现（参见 8.8 节[值历史]，88 页），在打印过后。

另一种创建伪数组的方法是使用强制转化。转化会将一个值作为一个数组对待。这个值不一定在内存里：

(gdb) p/x (short[2])0x12345678

$1 = {0x1234, 0x5678}

为方便起见，如果不指定数组长度（如'type[]value'）,GDB 会计算这个伪数组合适的长度（如'sizeof(value)/sizeof(type)'）：

(gdb) p/x (short[])0x12345678

$2 = {0x1234, 0x5678}

有时伪数组机制还不够；在相当复杂的数据结构里，结构体里的元素可能不是真的相邻--例如，如果你需要的元素在结构体里声明为指针。一个有用的变通方法（参见 8.9 节[惯用变量]，89 页）是在表达式里使用惯用变量作为计数器，记录第一个值，然后通过回车键重复执行表达式。例如，假设有一个结构体

数组名为 dtab，其结构体定义了一个指针 fv。下面是一个使用 dtab 的例子：

```
set $i = 0
p dtab[$i++]->fv
<RET>
<RET>
...
```

8.4 输出格式

缺省的，GDB 根据数据类型打印数据值。不过，有时这种方式可能不是你想要的。例如，你可能要以 16 进制打印一个数值，或者以 10 进制打印一个指针。或者你想要查看内存中某个地址上的数据，作为字符串或者一个指令。要做到这些，在打印值的时候指定输出格式就可以了。

最简单的输出格式的用法是指定如何打印一个已计算过的值。要达到这个目的，在命令 print 后加上反斜杠'/'和一个格式符号就可以。

格式符号如下：

x 将数值作为整型数据，并以 16 进制打印。

d 打印带符号整型数据

u 打印以无符号整型数据

o 以 8 进制打印整形数据

t 以 2 进制打印整形。't'代表'two'(注一).

注一：'b'不能用，因为这个格式在 x 命令里也用到了，x 命令里'b'代表'byte';参见 8.5 节[查看内存]，79 页。

a 打印地址，打印 16 进制的绝对地址和最近符号的偏移量。可以用这个格式找出一个未知地址的位于何处（在哪个函数里）：

```
(gdb) p/a 0x54320
$3 = 0x54320 <_initialize_vx+396>
```

命令 info symbol 0x54320 也能得到相似的结果。参见 13 章[符号]143 页。

c 将一个整型以字符常量打印。会打印一个数值和它表示的字符。超出 7 位的 ASCII 的数值（大于 127）的字符用 8 进制的数字替代打印。

不用这个格式的话，GDB 将 char，unsigned char 和 signed char 数据作为字符常量打印。单字节的向量成员以整型数据打印。

f 将数据以浮点类型打印。

s 如果可能的话，以字符串形式打印。用这个格式，指向单字节数据的指针将以 nll 结尾的字符串打印，单字节数据的数组则会以定长字符串打印。其它的值以它们原本类型打印。

不用这个格式的话，GDB 将指向 char，unsigned char 和 signed char 作为字符串打印，这些类型的数组也同样处理。单字节向量的成员以整型数组打印。

例如，要以 16 进制打印程序计数器（参见 8.10 节[寄存器]，90 页），输入 p/x $pc

注意，在反斜杠前不需要空格；这是由于 GDB 里的命令名不能包含一个反斜杠。

要用其它格式打印最近值历史里的值，可以用 print 命令带一个格式就可以了，不用指定表达式。例如，'p/x'会以 16 进制打印最近的值。

8.5 查看内存

可以用命令 x（表示"examine"）以多种格式查看内存，而和程序数据类型无关。

x/nfu addr

x addr

x 用 x 命令查看内存。

n，f 和 u 都是可选的参数，指定打印多长的内存数据和以何种格式打印之；addr 是需要打印的内存的地址表达式。如果用默认的 nfu，不需要输入反斜杠'/'。有几个命令可以方便的设置 addr 的默认值。

n，重复次数

10 进制整数；默认是 1。指定显示多长的内存（需要和单元长度 u 一起计算得到）。

f，显示格式

显示格式和 print 命令的格式一样（'x','d','u','o','t','a','c','f','s'），外加'i'（表示机器指令格式）。默认是'x'（16 进制）。默认格式在用 x 或 print 命令的时候都会改变。

u，单元大小，单元大小如下：

b 字节

h2 节节

w4 字节。默认值。

g8 字节。

每次用 x 指定单元长度，这个长度就成为默认值，知道下一次用 x 再设置。（对于's'和'i'格式，单元长度会被忽略而不会改写）

addr，要打印的起始位置

addr 是要 GDB 开始打印的内存起始位置。表达式不需要指针值（虽然其可能是一个指针值）；这个表达式总会翻译为内存中一个字节的整型地址。参见 8.1 节[表达式]，75 页，更多关于表达式的信息。默认的 addr 通常是紧随最近查看地址之后--但其它几个命令也可以设置默认地址：info breakpoints（设置为最近断点的地址），info line（设置一行代码的起始地址），和 print（如果用了 print 来显示内存中的一个值）。

例如，'x/3uh 0x54320'打印 3 个半字（6 字节）的内存数据，以 10 进制整型格式打印（'u'），从地址 0x54320 开始。'x/4xw $sp'打印 4 个字（'w'）的内存数据，以 16 进制从堆栈指针指向的内存开始（这里'$sp'；参见 8.10 节[寄存器]，90 页）打印。

由于制定单元长度的字符和制定输出格式的字符是截然不同的，因此不需要记住单元长度和格式字符的先后顺序；无论哪个在先都可以。输出格式声明'4xw'和'4wx'是一样的。（不过，次数 n 必须在先；'wx4'就是无效的。）

即使对于格式's'和'i'来说单元长度 u 是忽略不计的，用户也可能要用一个计数 n；例如，'3i'指定要看 3 个机器指令，包括操作数。为方便起见，特别是在用 display 命令时，'i'格式会超过计数所知定的长度打印延迟转移槽指令，如果有的话，这个转移指令就紧接在计数长度的指令之后。disassemble 命令提供了另一种查看机器指令的方式；参见 7.6 节[源代码和机器代码]，72 页。

x 命令的全部缺省参数都为方便的继续扫描内存而设计的，这样每次继续使用 x 命令的时候就只需要很少的指定了。例如，用'x/3i addr'命令查看机器指令后，可以只用'x/7'来查看下 7 个指令。如果用回车键来重复 x 命令的话，就会重复 n 次;其它参数就成为接下来的 x 命令的缺省值。

由于 x 命令打印的地址和内容通常是非常多而且可能会变成瓶颈，因此不会在值历史里保留。 相反，GDB 将那些在后续表达式里用到的值形成惯用变量$_和$__。一个 x 命令之后，最后被查看的地址可以用惯用变量$_来在表达式里引用。这个地址的内容，如前所查，可以用变量$__来引用。如果 x 命令带有重复次数参数，地址和内容会保存最后打印的内存单元；如果有多个单元在最后一行打印的话，就不是记录最后打印的地址。如果调试一个在远程机器上运行的程序（参见 17 章[远程调试]，179 页），你可能希望确认和下载到远程机器上的可执行文件相比的内存中的程序文件。compare-sections 命令提供了这样的功能。

compare-sections [section-name]

用可执行文件里的名为 section-name 的可加载段数据和远程机器内存中相同的段数据比较，并且打印不匹配的数据。如果不带参数，比 较所有的可加载段数据。这个命令的可用性依赖于系统对于"qCRC"远程请求的支持与否。

8.6 自动显示

如果你觉得需要频繁打印一个表达式的值（来查看其如何改变的），可能要把它加到自动显示列表里让 GDB 在每次程序中断时打印这个表达式的值。每个加入列表的表达式会有一个编号来标识；要将一个表达式从列表里删除，可以用这个编号。自动显示如下所示：

2: foo = 38

3: bar[5] = (struct hack *) 0x3804

这个输出显示了条目编号，表达式和它们目前的值。如同你手工用 x 或 print 命令那样打印输出那样，可以指定你喜欢的输出格式；事实上 display 命令决定是用 print 还是用 x 命令，这取决于你指定的格式--如果你指定了'i'或者's'格式的话，或者有一个单元长度的话，就用 x；

否则就用 print。

display expr

将表达式 expr 加入表达式列表，每次程序中断时自动显示。参见 8.1 节[表达式]，75 页。

在时候此命令后再按回车键时，display 不会重复执行。

display/fmt expr

fmt 只指定显示格式，不指定大小和次数，将表达式 expr 加入自动显示列表；每次用指定格式 fmt 输出。参见 8.4 节[输出格式]，78 页。

display/fmt addr

对于'i'或者's'格式，或者包含一个单元长度或一个单元数量的话，将表达式 addr 作为一个要查看的内存地址加入列表，每次程序中断的时候打印。"查看"用'x/fmt addr'命令来实现。参见 8.5 节[查看内存]，79 页。

例如，要在每次执行中断时查看机器指令，'display/i $pc'就很有用的（'$pc'是程序计数器的通用名；参见 8.10 节[寄存器]，90 页）。

undisplay dnums...

delete display dnums...

从显示列表中删除编号为 dnums 的显示项。

在执行 undisplay 后再回车的话，undisplay 不会重复。（否则你会得到'No display number....'的错误信息）

disable display dnums

禁用编号为 dnums 的显示项。禁用的显示项不会自动输出，但系统仍会记录它。可以再次激活。

enable display dnums...

激活编号为 dnums 的显示项。会再次自动显示其表达式的值，直到你禁用它。

display

显示当前列表中的变大是的值，就如同程序中断那样输出。

info display

打印此前设置的自动显示列表里的表达式，每个表达式带一个编号，但不显示其值。包括禁用的表达式，这类表达式会标明为禁用。也包括目前不能显示的表达式，这类表达式引用了当前不可用的自动变量。

如果显示表达式引用了本地变量，那么在其设置范围外是没有意义的。在执行到其变量无定义的上下文时，这类表达式会被禁用。

例如，如果你在一个函数内执行了命令 display last_char，last_char 是此函数的参数，GDB 会在程序再次执行到这个函数并在此函数中断

时自动显示此参数。要是在别的位置中断的话--那里没有变量 last_char--就会将此显示项自动禁用。下次程序在 last_char 有意义的位置中断时，可以再次激活这个显示表达式。

8.7 打印设置

GDB 提供如下数组，结构体和符号的打印设置方法。在任何语言里下列设置对于调试都是很有用的：

set print address

set print address on

GDB 打印内存地址，显示堆栈回溯的位置，结构体的值，指针值，断点等等，甚至在其也显示哪些地址上的内容时。缺省是 on。

例如，下面是用 set print address on 来设置后，堆栈帧的输出：

(gdb) f

#0 set_quotes (lq=0x34c78 "<<", rq=0x34c88 ">>")

at input.c:530

530 if (lquote != def_lquote)

set print address off

在现实其值的时候不打印地址。例如，下面是用 set print address off 设置后相同的堆栈帧的输出：

(gdb) set print addr off

(gdb) f

#0 set_quotes (lq="<<", rq=">>") at input.c:530

530 if (lquote != def_lquote)

可以用'set print address off'来从 GDB 接口中取消所有机器相关的显示。例如，使用 print address off，可以在所有机器上得到相同的内容的回溯--不论时候牵涉到指针参数。

show print address

显示是否打印地址。

GDB 打印符号的地址时，通常会打印离此地址最近且位置靠前的符号，外加打印偏移。如果符号不能指定位置的地址（例如，在一个文件里的一个名字），你就需要确认它。一个确认的方法是用 info line，例如'info line *0x4537'。另外一方法是，在打印一个符号的地址时设置 GDB 要打印的源文件和行号：

set print symbol-filename on

设置 GDB 要打印的符号的源文件名和行号。

set print symbol-filename off

不打印符号的源文件名和行号。默认方式。

show print symbo-filename

显示是否打印符号的源文件名和行号。

显示符号文件名和行号的另外一种有用的情况是，在反汇编代码时打印文件名和行号。GDB 会显示相应于每个指令的行号和源文件。

另外，你可能希望确认被打印地址的符号形式是否是离得最近且位置考前的符号：

set print max-symbolic-offset max-offset

设置 GDB 只显示地址的符号形式，如果偏移是在最近且靠前的符号和比 max-offset 低的地址区间。缺省值是 0，GDB 总是打印比此地址靠前的符号。

show print max-symbolic-offset

显示 GDB 打印一个符号地址的最大偏移量。

如果有一个指针但不能确定它指向何处，可以用'set print symbol-filename on'来尝试。然后用'p/a pointer'来确认此指针指向的名字和源文件的位置。这个命令可以将地址转换为符号形式。例如，下面是 GDB 显示的变量 ptt 指向另一个变量 t,于'hi2.c'定义：

(gdb) set print symbol-filename on

(gdb) p/a ptt

$4 = 0xe008 <t in hi2.c>

警告：对于执行一个本地变量的指针，'p/a'不显示涉及到符号名和文件名，即使是把相关的 set print 选项打开也如此。

其他设置控制如何打印不同类型的对象的方法如下：

set print array

set print array on

以习惯方式打印数组。这个格式更便于阅读，但要更多空间。默认是关闭的。

set print array off

返回到压缩方式打印数组。

show print array

显示何种方式显示数组。

set print array-indexes

set print array-indexes on

在打印数组的时候打印每个数组成员的下标。可以方便的找到一个给定的数组成员的位置，或者查找一个给定成员的下标。缺省是关闭的。

set print array-indexes off

在现实数组时，不打印数组成员下标。

show print array-indexes

显示在打印数组时是否输出成员下标。

set print elements number-of-elements

设置 GDB 打印的数组成员的数量。如果 GDB 打印一个大数组，在打印完 set print elements 命令设置的限制之后就不再继续打印此

数组成员。这个限制也会应用于字符串打印。GDB 启动时，这个限制设置为 200。将 number-of-element 设置为 0 意味着打印数组时没有长度限制。

show print elements

显示 GDB 打印大数组的长度。如果是 0，那么没有限制。

set print frame-arguments value

这个命令允许控制在调试器打印一个堆栈帧的时候，如何打印参数的值（参见 6.1 节[帧]，61 页）。可能的值：

all 打印所有的参数值。缺省的。

scalars 只打印非向量参数。复杂的参数如数组，结构体，联合等，用...替代。下面是非向量参数的打印例子：

#1 0x08048361 in call_me (i=3, s=..., ss=0xbf8d508c, u=..., e=green)

at frame-args.c:23

none 不打印参数。所有的参数都用...替代。下面是这样的例子：

#1 0x08048361 in call_me (i=..., s=..., ss=..., u=..., e=...)

at frame-args.c:23

缺省的，总是打印所有的参数。不过这个命令在好几情况下是非常有用的。例如，在打印每个帧时可以用来减少输出的信息，使得回溯更加可读。而且，在打印 Ada 帧时这个命令可以提高执行效率，因为有时大参数的计算可能是 CPU 密集型的，特别是在大程序里。设置 print frame-arguments 为 scalars 或 none 可以避免这类运算，因此可以加速打印 Ada 帧。

show print frame-arguments

显示打印帧时如何显示参数。

set print repeats

设置打印数组的长度上限值。如果数组中连续相同的成员的数量超过这个上限，GDB 会打印字符串 "<repeats n times>"，这里 n 是同样的重复次数，而不是重复打印这些相同的成员。将这个上限设置为 0 的话，打印所有的成员。默认上限时 10。

show print repeats

显示打印重复相同成员的上限数量。

set print null-stop

设置 GDB 在初次遇到 NULL 字符是终止打印字符串。如果大数组里包含短字符串时很有用。默认关闭。

show print null-stop

显示 GDB 是否在初次遇到 NULL 字符串时停止打印。

set print pretty on

设置 GDB 在打印结构体时，以缩进的格式每行打印一个结构体成员，如下：

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

设置 GDB 以紧凑方式打印结构体，如下所示：

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

默认方式。

set print sevenbit-strings on

只打印 7bit 的字符串。如果设置了这个选项，GDB 用\nnn 来显示 8bit 字符。如果在英语（ASCII）环境下执行或者将高位作为标记位或作为元数据位的话，这个设置是非常有用的。

set print sevenbit-strings off

打印 8bit 字符。允许使用国际化字符集，缺省的。

show print sevenbit-strings

显示 GDB 是否打印 7bit 字符。

set print union on

设置 GDB 打印包含结构体或其他联合的联合。缺省设置。

set print union off

设置 GDB 不打印包含结构体或其他联合的联合。GDB 用"{...}"替代之。

show print union

显示 GDB 是否打印包含结构体和其它联合的联合。

例如，假设下列声明

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

```
struct thing {
Species it;
union {
Tree_forms tree;
Bug_forms bug;
} form;
};
struct thing foo = {Tree, {Acorn}};
```

设置 set print union on，'p foo'会打印如下输出：

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

设置 set print union off，'p foo'会打印如下输出：

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

set print union 对于类似 C 和 Pascal 的程序有效。

下面这些设置对于调试 C++程序很有用：

set print demangle

set print demangle on

用源代码的形式打印 C++名，而不是用编码后传递给汇编器和连接器的形式。默认打开。

show print demangle

显示是否用源码形式打印 C++名。

set print asm-demangle

set print asm-demangle on

用源代码的形式打印 C++名，而不用编码形式，即使在汇编代码的输出如指令反汇编。默认关闭。

show print asm-demangle

显示是否用源码形式打印汇编代码。

set demangle-style style

从编码方式中选择一种编码体系，解析 C++名。style 的选择有如下几种：

auto 设置 GDB 通过探测程序来选择一种解码方式。

gnu 基于 GNU C++编译器(g++)的编码算法的解码方式。缺省方式。

hp 基于 HP ANSI C++（aCC）编码算法的解码方式。

lucid 基于 Lucid C++编译器编码算法的解码方式。

arm 用 C++ Annotated Reference Manual 里定义的算法解码。警告：这个选项不足与调试 cfront 产生的可执行程序。GDB 要

改进才能调试此类程序。

如果忽略了 style，可能会看多多种格式的输出。

show demangle-style

显示目前 C++符号解析使用的编码方式。

set print object

set print object on

在打印一个指针指向的对象时，使用虚函数表来标明对象真实的类型，而不是其声明的类型。

set print object off

只显示对象所声明的类型，不引用虚函数表。默认选项。

show print object

显示是否打印真实的，或声明的对象类型。

set print static-members

set print static-members on

打印 C++对象的静态成员。选项缺省打开。

set print static-members off

打印 C++对象时不打印静态成员。

show print static-members

显示是否打印 C++静态成员。

set print pascal_static-members

set print pascal_static-members on

打印 Pascal 对象的静态成员。选项默认打开。

set print pascal_static-members off

不打印 Pascal 对象的静态成员。

show print pascal_static-members

显示是否打印 Pascal 静态成员。

set print vtbl

set print vtbl on

以习惯方式打印 C++虚函数表。默认关闭。（vtbl 命令在 HP ANSI C++编译器（aCC）编译的程序上无效。）

set print vtbl off

不以习惯方式打印 C++虚函数表。

show print vtbl

显示是否以习惯方式打印 C++虚函数表。

8.8 值历史

以 print 命令打印的值保存于 GDB 值历史里。这种方式使得在其它表达式里引用这些值。值会保留到符号重载或者被丢弃（例如用 file 或者 symbol-file 命令）。在符号表改变时，值历史会被丢弃，因为值可能包含指向符号表里定义的类型。

打印过的值在历史表里都有编号，可以用此编号来引用值。这个编号是从 1 开始的连续整数。print 在打印值的时候会在值前打印'$num='，num 是历史表里的编号。

要引用此前的值，用'$'后接值历史编号就可以了。print 打印标记'$'就是提醒你这个的。只有一个$指最近的值，$$指倒数第二近的值。$$n 指倒数 n 近的值；$$2 是比$$靠前一个的值，$$1 和$$相等，$$0 和$相等。

例如，假设刚打印过一个指针指向的结构体，想要查看这个结构体的内容。输入下列命令就可以了：

p *$

如果有一个结构体链表，其结构体里有一个成员 next 指向下一个结构体，可以用下面的命令来打印下一个结构体的内容：

p *$.next

可以重复执行这个命令来连续打印这个链表--只需要输入回车键。

注意，值历史记录值，不记录表达式。如果 x 的值是 4，输入如下命令：

print x

set x=5

那么值历史里记录的值是 4，即使 x 的值已经变为 5 了。

show values

打印值历史表里最近的 10 个值，带其编号打印。和'p $$9'重复类似，除了 show values 不改变值历史。

show values n

打印值历史表里的 10 个记录，以编号 n 为中心打印。

show values +

打印最近打印过的值前的 10 个值。如果没有记录，show values +不产生输出。

输入回车键来重复 show values n 和'show values +'效果一样。

8.9 惯用变量

GDB 提供了惯用变量，用户可以在 GDB 里用来存储数据，在以后引用。这类变量在 GDB 里全程存在；它们不是被调试程序的组成部分，设置惯用变量对程序执行没有直接影响。这就是为什么可以自由使用这类变量的原因。

惯用变量有个前缀'$'。任何前导'$'的名字都可以用作惯用变量，除非是已经定义好的，系统指定的寄存器名（参见 8.10 节[寄存器]，90 页）。（相反，引用值历史，是在数字前加'$'。参见 8.8 节[值历史]，88 页）

可以将一个表达式的值保存在惯用变量里，就如同在程序里设置一个变量一样。例如：

set $foo = *object_ptr

可以将指针 object_ptr 指向的值保存在$foo 里。第一次使用惯用变量的时候会创建此变量，但其值是 void，直到设置一个值为止。可以在任何时间改变此值。

惯用变量没有固定的类型。可以为惯用变量指定任意类型的值，包括结构体和数组，即使此变量已经有一个不同类型的值。惯用变量，在一个表达式里用时，其类型是当前值的类型。

show convenience

打印目前为止的惯用变量列表和它们的值。缩写为 show conv。

init-if-undefined $variable=expression

设置一个惯用变量，如果其尚未设置的话。对于用户定义的保持某些状态的命令而言很有用。在概念上和在 C 语言里带初始化的使用本地静态变量很相似（除了惯用变量是全局的）。也可以用于覆盖在命令脚本里设置的缺省值。

如果变量已经定义了，那么不计算表达式，因此也就没有边际效应。

惯用变量的一种使用方式是作为一个增量计数器或者是一个指针。例如，要打印一个结构体数组成员里的域：

set $i = 0

print bar[$i++]->contents

用回车键重复命令。

有些惯用变量是 GDB 自动创建的，记录有用的值。

$_变量$_是 x 命令自动设置的，记录最近查看的地址（参见 8.5 节[查看内存]，79 页）。其它提供一个默认地址给 x 来查看的命令也会设置这个变量；这些命令包括 info line 和 info breakpoint。$_的类型是 void *；在 x 命令设置之后，其类型是$__的指针。

$__变量$__是 x 命令自动设置的，记录最近查看地址上的值。其类型是打印的数据所匹配的类型。

$_exitcode

变量$_exitcode 在程序调试结束时自动记录退出码。

在 HP-UX 系统里，如果被引用的函数或变量名前有一个$符号，GDB 会先搜索用户或系统名，其后再搜索惯用变量。

8.10 寄存器

在表达式里可以引用系统寄存器内容，将在寄存器名前置$符作为变量来用。寄存器名对于各系统可能不一样；使用 info registers 可以查看系统的寄存器名。

info registers

打印所有的寄存器名和值，除了浮点和向量寄存器（在选定的堆栈帧里）。

info all-registers

打印所有的寄存器名和值，包括浮点和向量寄存器（在选定的堆栈帧里）。

info registers regname ...

打印所有指定寄存器对应的值。如下面详细讨论的那样，寄存器值通常是相对于选定的堆栈帧的。regname 可能是系统里可用的，任意的寄存器名带不带'$'都可以。

GDB 有 4 个在多数系统里可用的（在表达式里）"标准" 寄存器名--他们和系统标准的寄存器名总是不冲突的。寄存器名$pc 和$sp 由于记录程序计数器寄存器和堆栈指针。$fp 用于记录当前堆栈帧的指针，$ps 用于记录处理器状态的寄存器。例如，可以用 16 进制打印程序计数器：

p/x $pc

或者打印下一条要执行的指令

x/i $pc

或者将堆栈指针加 4(注)：

set $sp += 4

只要有可能，这 4 个标准寄存器名在系统里都是可用的，即使系统有不同的命名，只要没有冲突就行。info registers 命令显示标准寄存器名。例如，在 SPARC 系统里，info registers 显示处理器状态寄存器为$psr，但也可应用$ps 来引用；在基于 x86 的系统里$ps 是 EFLAGS 寄存器的别名。

GDB 总是将一个普通寄存器的内容作为一个整型值，寄存器也是这样查看的。某些系统具有只能存储浮点数据的特殊寄存器；这类寄存器因此应该认为具有浮点类型的值。没有方法将普通寄存器作为浮点型来引用（虽然可以用用 print 将此寄存器作为浮点数打印，'print/f $regname'）。

某些寄存器具有不同的'原始'和'虚拟'的数据格式。这意味着由操作系统设置的寄存器值的数据格式和程序通常得到的不一样。例如，68881 浮点协处理器的寄存器总是保存"扩展"（原始）的数据格式，但所有 C 程序会以"double"(虚拟)的格式处理。在这些情况下，GDB 通常以虚拟格式处理（程序能够处理的格式），但 info registers 命令会以两种格式打印这个数据。

注：这是从堆栈里删除一个字的方法，在系统内存中堆栈是向下增长的（现代多数系统）。这种方法假设选定的是最内层的堆栈帧；在选定别的堆栈帧时不允许设置$sp，使用 return；参见 14.4 节[从函数返回]，151 页。

某些系统具有一些其它特殊的寄存器，它们的内容可以有多种转换方式。例如，现代基于 x86 的系统有 SSE 何 MMX 寄存器，它们以不同的格式保存打包在一起的多个值。GDB 用结构体方式来引用这些寄存器：

```
(gdb) print $xmm1
$1 = {
v4_float = {0, 3.43859137e-038, 1.54142831e-044, 1.821688e-044},
v2_double = {9.92129282474342e-303, 2.7585945287983262e-313},
v16_int8 = "\000\000\000\000\3706;\001\v\000\000\000\r\000\000",
v8_int16 = {0, 0, 14072, 315, 11, 0, 13, 0},
v4_int32 = {0, 20657912, 11, 13},
v2_int64 = {88725056443645952, 55834574859},
uint128 = 0x0000000d0000000b013b36f800000000
}
```

要设置这类寄存器，需要告诉 GDB 要设置寄存器的哪部分，就如设置一个结构体的域那样：

(gdb) set $xmm1.uint128 = 0x00000000000000000000000FFFFFFFF

通常，寄存器值和选定堆栈帧是相对应的（参见 6.3 节[选择帧]，64 页）。这就是说，如果已经退出所有更深层的堆栈帧并且已经保存了它们的寄存器值的话，你就可以得到这些寄存器的值。要查看硬件寄存

器里的真实内容，必须选择最内层的堆栈帧（用'frame 0'）。

然而，GDB 需要从编译器产生的机器代码里推导出寄存器保存于何处。如果某些寄存器没有保存的话，或者 GDB 不能定位已保存的寄存器的话，那么选定的堆栈帧就没法区分了。

8.11 浮点硬件

取决于是如何配置的，GDB 可以提供关于浮点硬件的状态信息。

info float

显示挂于浮点单元的硬件相关的信息。确切内容和布局随浮点芯片不同而有所差异。目前，'info float'在 ARM 和 x86 平台上支持。

8.12 向量单元

取决于是如何配置的，GDB 可以提供关于向量单元的状态信息。

info vector

显示向量单元的信息。确切内容和布局随硬件不同而有所差异。

8.13 操作系统辅助信息

GDB 提供操作系统有用的工具的接口，用户可以用来帮助调试程序。

如果运行于 Posix 系统（如 GNU 或者 Unix 系统），GDB 通过系统调用 ptrace 和操作系统内部进行通信。操作系统为这个接口创建了特殊的数据结构，称为 struct user。可以用命令 info udot 来显示这个数据结构的内容。

info udot

显示操作系统内核维护的 struct user 结构体的内容。GDB 用类似于 examine 命令的形式来显示 struct user 内容，打印出 16 进制数据的列表。

某些操作系统在程序启动时提供了一个辅助向量。这个向量等效于为程序指定的参数、环境变量，包含一些系统相关的二进制的值，让系统库得到关于硬件，操作系统和进程的重要的细节。每个值的目的都由一个整数标签指定；其含义是众所周知而又系统相关的。取决于配置和操作系统的工具，GDB 可以显示这些信息。对于远程系统，这个功能可能进步依赖于远程存根对于'qXfer:suxv:read'包的支持，参见[qXfer

辅助向量读]，354 页。

info auxv

显示内部辅助向量，此向量可以是一个正在执行的进程或者是一个 core dump 文件。GDB 以数值形式打印每个标签，并显示名称和对可识别的标签显示文字描述。向量里的某些值是数字，某些位(bit)是掩码，某些是指向字符串或其它数据的指针。GDB 会以适当的方式显示每个可识别的标签，对于不可识别的标签则以 16 进制的形式显示。

8.14 内存区域属性

内存区域属性提供了描述由系统内存请求的特殊处理的功能。GDB 使用属性来判断是否允许某些类型的内存访问；是否使用明确的访问宽度；时候缓存系统内存。缺省的，内存区域的描述取自系统（如果当前系统支持的话），但用户可以覆盖被取的区域。

已定义的内存区域可以单独的激活或禁用。如果禁用一个内存区域，GDB 会在访问这个区域时使用缺省属性。类似的，如果没有定义内存区域的话，GDB 在访问任何内存时都使用缺省属性。

如果定义了内存区域，会有一个整数来标识；要激活，禁用或删除它的话，应该用这个编号。

mem lower upper attributes...

定义一个内存区域，从 lower 到 upper，属性是 attribute...，并将其加入由 GDB 监控的区域列表。注意，upper==0 是个特殊例子：

当作系统最大内存地址。（16 位系统里是 0xffff，32 位系统是 0xffffffff）

mem auto

放弃用户对内存区域的改变，并使用系统提供的区域，如果有的话，如果系统不提供的话就不适用内存区域。

delete mem nums...

从 GDB 监控的内存列表里删除内存区域 nums...。

disable mem nums...

禁止监视内存区域 nums...被禁用的内存区域不会被遗忘。可以在此激活之。

enable mem nums...

激活监控内存区域 nums...

info mem

打印所有定义的内存区域列表，每个区域都有下面的列：

Memory Region Number

Enabled or Disabled.

以激活的内存区域标记为'y'。已禁用的内存区域标记为'n'。

Lo Address

内存区域最低地址。

Hi Address

内存区域最高地址。

Attributes

内存区域的属性集。

8.14.1 属性

8.14.1.1 内存访问模式

访问模式属性决定 GDB 是否可以对一个内存区域进行读写访问。

要是这些属性阻止 GDB 进行非法内存访问的话，它们将组织系统 I/O，DMA 之类的内存访问。

ro 内存只读。

wo 内存只写。

rw 内存可读写。默认属性。

8.14.1.2 内存访问的尺寸

访问尺寸属性告诉 GDB 使用指定大小的内存访问。通常内存和设备寄存器要求的指定大小的访问匹配。如果不指定访问尺寸属性，GDB 可能使用任意大小的访问。

8 使用 8 位内存访问。

16 使用 16 位内存访问。

32 使用 32 位内存访问。

64 使用 64 位内存访问。

8.14.1.3 数据缓冲 4

数据缓冲属性设置 GDB 是否缓冲系统内存。由于减少了调试协议的开销，这个属性可以改善性能，与此同时也可能导致错误的结果，因为 GDB 不知道 volatile 变量和内存映射寄存器。

cache 激活缓存系统内存。

nocache 禁用缓冲系统内存。默认属性。

8.14.2 内存访问检查

GDB 可以设置拒绝访问没有明确描述的内存。如果在某个系统下，访问这些内存区域存在不能预料的效果的话，要预防这种状况，或者要提供一个更好的错误检查，都是很大帮助的。下列命令控制这种行为。

set mem inaccessible-by-default [on|off]

如果设置 on，设置 GDB 将未明确描述范围的内存当作不存在的并拒绝对此内存的访问。只有在至少有一个已定义的内存范围的情况下才会进行检查。如果设置了 off，设置 GDB 将此未明确描述范围的内存作为 RAM。默认值是 on。

show mem inaccessible-by-default

显示当前对于未知内存访问的设置。

8.15 在内存和文件之间复制数据

可以用命令 dump，append 和 restore 来在目标内存和文件直线复制数据。dump 和 append 命令将数据写入文件，restore 命令将文件数据读入到内存中。文件可以是二进制，Motorola S-record，Inetl16 进制，或着 Tekrronix16 进制格式的；不过，GDB 只支持将数据附加到二进制文件。

dump [format] memory filename start_addr end_addr

dump [format] value filename expr

将内存从 start_addr 开始到 end_addr 的内容，或表达式 expr 的值，以指定格式转储到文件。

格式参数可以是下面类型之一：

binary 原始二进制形式

ihexIntel 16 进制格式

srecMotorola S-record 格式

tekhexTektronix 16 进制格式

GDB 使用的格式和 GNU 二进制工具所使用的一样，比如'objdump'和'objcopy'。如果省略 format，GDB 用原始二进制格式转储数据。

append [binary] memory filename start_addr end_addr

append [binary] value filename expr

将内存从 start_addr 开始到 end_addr 的内容，或表达式 expr 的值，以原始二进制格式附加到文件。（GDB 只能用原始二进制格式附加数据到文件。）

restore filename [binary] bias start end

将文件 filename 的内容恢复到内存中。restore 命令可以自动识别任何已知的 BFD 文件格式，除了原始二进制文件。要恢复原始二进制文件，必须在文件名后指定可选关键字 binary。

如果 bias 非零，它是指从文件开头的偏移量。二进制文件总是荣地址 0 开始，所以会从地址 bias 开始恢复。其他 bfd 文件有一个内置位置；可以从那个位置再偏移 bias 开始恢复。

如果 start 和/或 end 是非零的话，那么只有在文件偏移 start 和文件偏移 end 之间的数据会恢复。这些偏移是相对于文件内的位置的，且是在 bias 参数相加之前的偏移。

8.16 如何从程序里产生 Core 文件

core 文件或者 core dump 记录执行中的进程的内存镜像和状态（例如寄存器值）。它的主用作用是对崩溃的程序事后调试。发生崩溃的程序会自动产生 core 文件，除非这个功能被用户禁用了。关于事后调试模式的信息，参见 15.1 节[文件]，155 页。

偶尔的，可能希望在调试程序期间产生 core 文件来保存进程的状态快照。GDB 为此提供了一个特殊的命令。

generate-core-file [file]

gcore [file]

为调试进程产生 core dump。可选参数 file 指定存储 core dump 的文件名。如果没有指定，文件名那个默认是'core.pid'，这里 pid 是被调试进程的进程 ID。

注意，这个命令只在某些系统上实现（到手册编写时，GNU/Linux，FreeBSD，Unixware 和 S390）。

8.17 字符集

如果调试中的程序使用的字符集和 GDB 使用的不一样，GDB 可以自动为用户转换字符集。GDB 使用的字符集我们称为宿主字符集；调试程序使用的称为目标字符集。

例如，如果在 GNU/Linux 系统上运行 GDB，GNU/Linux 系统使用 ISO Latin 1 字符集，而用户用 GDB 远程协议（参见 17 章[远程调试]，171 页）来调试在 IBM 框架下运行的程序，其字符集是 EBCDIC 字符集，那么宿主字符集是 Latin-1，而目标字符集是 EBCDIC。如果用命令 set target-charset EBCDIC-US 设置 GDB，那么在输入字符或字符串或在表达式里使用字符和字符串时，GDB 会在 EBCDIC 和 Latin 1 之间转换。

GDB 不能自动识别调试中的程序所使用的字符集；用户必须告诉它，使用 set target-charset 命令，如下所述。

下面是控制 GDB 字符集的命令：

set target-charset charset

将当前目标字符集设置为 charset。下面会列举 GDB 能识别的字符集名称，不过如果输入 set target-charset 接着再敲两次 TAB 键的话，GDB 会列出它所能识别的字符集。

set host-charset charset

设置当前的宿主字符集为 charset。

缺省的，GDB 使用的宿主字符集和系统的相近；可以用 set host-charset 命令覆盖默认值。

GDB 只能使用某些字符集作为宿主字符集。下面会列举 GDB 能识别的字符集名称，并指明哪种可以用作宿主字符集，不过如果输入 set target-charset 接着再敲两次 TAB 键的话，GDB 会列出它所支持的宿主字符集。

set charset charset

涩会址当前的宿主和目标字符集为 charset。如前所述，如果输入 set charset 接着再敲两次 TAB 键的话，GDB 会列出它所支持的宿主/目标字符集。

show charset

显示当前宿主和目标字符集。

show host-charset

显示当前宿主字符集名。

show target-charset

显示当前目标字符集名。

GDB 目前支持下列字符集：

ASCIIU.S. ASCII 7-bit。GDB 可使之为其宿主字符集。

ISO-8859-1

ISO Latin 1 字符集。为法语，德语和西班牙语的重音符号而扩展到字符集。GDB 可使之为其宿主字符集。

EBCDIC-US

IBM1047

EBCDIC 字符集的变体，用于某些 IBM 架构的操作系统。（S/390 上的 GNU/Linux 使用 U.S. ASCII）GDB 不可使之为其宿主字符集。

注意，这些都是单字节字符集。GDB 里的很多处理都需要支持多字节或可变长度字符编码，例如 Unicode 的 UTF-8 和 UCS-2 编码方法。下面是 GDB 字符集支持的实际例子。假设下面的源代码在文件'charset-test.c'里：

```
#include <stdio.h>
char ascii_hello[]
= {72, 101, 108, 108, 111, 44, 32, 119,
111, 114, 108, 100, 33, 10, 0};
char ibm1047_hello[]
= {200, 133, 147, 147, 150, 107, 64, 166,
150, 153, 147, 132, 90, 37, 0};
main ()
{
printf ("Hello, world!\n");
}
```

在此程序里，ascii_hello 和 ibm1047_hello 是包含字符串"hello,world!"的数组，用 ASCII 和 IBM1047 字符集编码。

编译此程序并开始调试之：

```
$ gcc -g charset-test.c -o charset-test
$ gdb -nw charset-test
```

```
GNU gdb 2001-12-19-cvs
Copyright 2001 Free Software Foundation, Inc.
...
(gdb)
```

用 show charset 命令来查看 GDB 目前使用哪中字符集来转换和显示字符和字符串：

```
(gdb) show charset
The current host and target character set is ‘ISO-8859-1’.
(gdb)
```

出于打印手册的缘故，让我们用 ASCII 作为初始字符集：

```
(gdb) set charset ASCII
(gdb) show charset
The current host and target character set is ‘ASCII’.
(gdb)
```

假设 ASCII 是目前宿主系统的真正字符集--换句话说，假设 GDB 用 ASCII 字符集打印字符，终端会正确的显示字符。由于当前的目标字符集

也是 ASCII，ascii_hello 的内容会以可读的形式打印：

```
(gdb) print ascii_hello
$1 = 0x401698 "Hello, world!\n"
(gdb) print ascii_hello[0]
$2 = 72 ’H’
(gdb)
```

GDB 使用目标字符集来打印字符和字符串常量：

```
(gdb) print ’+’
$3 = 43 ’+’
gdb)
```

ASCII 字符集使用数字 43 来编码字符'+'。

GDB 依赖用户告知目标程序使用的是何种字符集。如果目标字符集还是 ASCII，我们试图打印 ibm1047_hello 就会得到乱码：

```
(gdb) print ibm1047_hello
$4 = 0x4016a8 "\310\205\223\223\226k@\246\226\231\223\204Z%"
(gdb) print ibm1047_hello[0]
$5 = 200 ’\310’
(gdb)
```

如果输入 set target-charset 接着再敲两次 TAB 键，GDB 会告知其所支持的字符集：

(gdb) set target-charset

ASCII EBCDIC-US IBM1047 ISO-8859-1

(gdb) set target-charset

我们可以选择 IBM1047 作为目标字符集，并再次检查程序字符串。现在 ASCII 字符串就是错误的了，但 GDB 会将 ibm1047_hello 从目标字符集转换到宿主字符集 ASCII，这样就可以正确显示了：

```
(gdb) set target-charset IBM1047
(gdb) show charset
The current host character set is ‘ASCII’.
The current target character set is ‘IBM1047’.
(gdb) print ascii_hello
$6 = 0x401698 "\110\145%%?\054\040\167?\162%\144\041\012"
(gdb) print ascii_hello[0]
$7 = 72 ’\110’
(gdb) print ibm1047_hello
$8 = 0x4016a8 "Hello, world!\n"
(gdb) print ibm1047_hello[0]
$9 = 200 ’H’
(gdb)
```

如前所示，GDB 用目标字符集来打印字符和字符串常量：

```
(gdb) print ’+’
$10 = 78 ’+’
(gdb)
```

IBM1047 字符集使用数字 78 来编码字符'+'。

8.18 缓存远程目标的数据

GDB 可以缓存在调试器和远程目标之间交换的数据（参见 17 章[远程调试]，171 页）。这种缓存通常可以改善性能，因为其可减少由于内存读写所带来的远程协议的开销。不幸的是，目前 GDB 对 volatile 寄存器无能为力，因此如果使用了 volatile 寄存器的时候，数据缓存就会带来错误的结果。

set remotecache on

set remotecache off

为远程目标设置缓存状态。如果是 on 的话，使用数据缓存。缺省的，此选项是 off。

show remotecache

显示目前远程目标的数据缓存状态。

ifo dcache

打印数据缓存性能的信息。显示的信息包括：dcache 的宽度和深度；对于每个缓存行，其被多少次引用到了，其数据和状态（脏，坏，好，等等）。对于调试数据缓存操作，这个命令很有帮助。

第9章 第九章 C 预处理宏

某些语言，例如 C 和 C++，提供定义和引用“预处理宏”的方法，这些宏可以展开为符号串。GDB 可以计算包含宏的表达式，显示宏展开的结果，并且显示宏的定义，包括在何处定义的。

可能需要特别编译程序来给 GDB 提供预处理宏的信息。大多数编译器在调试信息中不包括宏，即使编译时使用'-g'选项。参见 4.1 节[编译]，25 页。

程序可能在某个点定义一个宏，在后面删除这个定义，然后在此后给这个宏提供另外的定义 。因此，在程序里不同点上，同一个宏可能有不同的定义，或者根本就没有定义。如果在当前堆栈帧上，GDB 使用这个帧源代码行范围的宏。否则，GDB 使用当前位置范围的宏；参见[打印源代码行]，67 页。

同时，GDB 不支持##符号剪接操作符，#宏字符串常量替换操作符，或者可变长宏。

无论何时 GDB 计算表达式，总会将在表达式里引用的宏展开。GDB 也提供下列命令来明确地识别宏。

macro expand expression

macro exp expression

显示表达式里引用的所有预处理宏的展开结果。由于 GDB 只简单地展开宏，不会去解析结果，因此表达式不必是有效的；可以是包含任意符号的的字符串。

macro expand-once expression

macro exp1 expression

（此命令尚未实现。）显示在表达式里引用的预处理宏的展开结果。表达式里的所引用的宏不会改变。这个命令可以更清楚的查看一个特殊宏，而不会被更多的展开所迷惑。由于 GDB 只是简单展开宏，而不解析结果，表达式不必是有效的；可是是任意符号的字符串。

info macro macro

显示名为 macro 的宏的定义，并显示这个定义是在代码的何处设立的。

macro define macro replacement-list

macro define macro(arglist) replacement-list

（此命令尚未实现。）为名为 macro 的宏引入一个定义，对其的引用将被给定的 replacement-list 所替换。此命令的第一中形式定义了“对象式”的宏，不带参数；第二种形式定义了一个“函数式”的宏，用给定的 arglist 作为参数的。

用此命令引入的定义，其范围在 GDB 所计算的所有表达式中，知道用命令 macro undef 命令删除之，

如下所述。此定义会覆盖调试程序里所有名为 macro 的宏定义，并且也包括任何用户提供的定义。

macro undef macro

（此命令尚未实现。）删除所有用户提供的名为 macro 宏定义。此命令只影响用命令 macro define 所定义的宏。如前所述；不能删除调试程序里的定义宏。

macro list

（此命令尚未实现。）列举所有用 macro define 命令定义的宏。

下面的例子展示了如何使用上述命令。首先，看一下源代码：

```
$ cat sample.c
#include <stdio.h>
#include "sample.h"
#define M 42
#define ADD(x) (M + x)
main ()
{
#define N 28
printf ("Hello, world!\n");
#undef N
printf ("We’re so creative.\n");
#define N 1729
printf ("Goodbye, world!\n");
}
$ cat sample.h
#defineQ <
$
```

现在，让我们用 GNU C 编译器 GCC 来编译此程序。我用在编译的时候指定'-gdwarf-2'和'-g3'参数来产生预处理宏的调试信息。

$ gcc -gdwarf-2 -g3 sample.c -o sample

$

接着，我们就可以启动 GDB 来调试此例子程序了：

```
$ gdb -nw sample
GNU gdb 2002-05-06-cvs
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, ...
(gdb)
```

我们可以展开宏并检查其定义，甚至是在程序尚未运行时。GDB 使用当前代码位置来判断哪个宏的定义处于此范围之内：

```
(gdb) list main
```

```
3
4 #define M 42
5 #define ADD(x) (M + x)
6
7 main ()
8 {
9 #define N 28
10 printf ("Hello, world!\n");
11 #undef N
12 printf ("We're so creative.\n");
(gdb) info macro ADD
Defined at /home/jimb/gdb/macros/play/sample.c:5
#define ADD(x) (M + x)
(gdb) info macro Q
Defined at /home/jimb/gdb/macros/play/sample.h:1
included at /home/jimb/gdb/macros/play/sample.c:2
#define Q <
(gdb) macro expand ADD(1)
expands to: (42 + 1)
(gdb) macro expand-once ADD(1)
expands to: once (M + 1)
(gdb)
```

注意，在上面的这个例子里，macro expand-once 只把原文本引用的宏展开--ADD 的引用--但不展开宏 M，此宏由 ADD 所引用。

一旦程序运行起来后，在当前堆栈帧的源代码上，GDB 使用有效的宏定义：

```
(gdb) break main
Breakpoint 1 at 0x8048370: file sample.c, line 10.
(gdb) run
Starting program: /home/jimb/gdb/macros/play/sample
Breakpoint 1, main () at sample.c:10
10 printf ("Hello, world!\n");
(gdb)
```

在第 10 行，宏 N 的有效定义是在第 9 行：

```
(gdb) info macro N
Defined at /home/jimb/gdb/macros/play/sample.c:9
#define N 28
(gdb) macro expand N Q M
expands to: 28 < 42
(gdb) print N Q M
$1 = 1
(gdb)
```

如果我们单步执行到删除 N 的定义之后，并给其新的定义，GDB 会在每个点上找到有效的定义（或

者没有定义）：

```
(gdb) next
Hello, world!
12 printf ("We’re so creative.\n");
(gdb) info macro N
The symbol ‘N’ has no definition as a C/C++ preprocessor macro
at /home/jimb/gdb/macros/play/sample.c:12
(gdb) next
We’re so creative.
14 printf ("Goodbye, world!\n");
(gdb) info macro N
Defined at /home/jimb/gdb/macros/play/sample.c:13
#define N 1729
(gdb) macro expand N Q M
expands to: 1729 < 42
(gdb) print N Q M
$2 = 0
(gdb)
```

第10章 跟踪点

在某些应用程序里，调试器不大可能因为开发者要了解此程序的行为，长时间的中断程序的执行。如果程序的正确性依赖于实时行为，调试器造成的延迟会导致程序根本改变其行为，甚至在代码本省正确的情况下也可能导致失败。不中断程序的执行来观察其行为是非常有用的功能。

使用 GDB 的 trace 或者 collect 命令，可以指定程序里的位置，称为跟踪点，和在跟踪点执行到的时候要计算的任意表达式。稍后在跟踪点执行到的时候，可以用 tfind 命令来查看表达式的值。表达式也可以引用内存里的对象--结构体或者数组，例如---GDB 应该记录的值；在访问一个特殊的跟踪点时，可以查看那些对象，如果在这个时间点上这些对象在内存里的话。不过，由于 GDB 不需要和用户交互就可以记录这些值，所以 GDB 可以迅速优雅的记录而不干扰调试程序运行。

目前，跟踪点功能只在远程系统上实现。参见第 16 章[目标],167 页。另外，远程系统必须了解如何收集跟踪数据。这个功能实现于远程存根；不过，到这个手册编写为止，没有存根支持 GDB 提供的跟踪点。实现跟踪点的远程数据包格式参见 D.6 节[跟踪点数据包]，356 页。

本章说明跟踪点命令和功能。

10.1 设置跟踪点的命令

在运行跟踪尝试前，可以设置任意跟踪点号都。如同断点那样（参见 5.1.1 节[设置断点]，40 页），跟踪点由 GDB 分配编号。和断点一样，跟踪点编号是从 1 开始连续的整数。跟踪点相关的命令，大多需要跟踪点号作为其参数来指定跟踪点。

可以为各个跟踪点指定要系统收集的任意的数据集，此数据存储于跟踪缓冲区里。收集的数据包括寄存器，本地变量和全局数据。接下来，可以用 GDB 命令来查看这些数据的值。

本节说明设置跟踪点的命令，并设置相关条件和操作。

10.1.1 创建和删除跟踪点

tracetrace 命令和 break 命令非常相似。其参数可以是源代码行，函数名或者目标程序的某个地址。参见 5.1.1 节[设置断点]，40 页。trace 命令创建跟踪点，程序在此点上短暂中断，收集数据，然后程序继续往下执行。设置跟踪点或者改变跟踪点命令直到下个 tstart 命令才会生效；因此，不能在跟踪会话过程

中改变跟踪点的属性。

下面是使用 trace 命令的一些例子：

(gdb) trace foo.c:121 // 源文件和行号

(gdb) trace +2 // 当前行的下两行

(gdb) trace my function // 函数的第一行代码

(gdb) trace *my function // 函数的真正开始的地方

(gdb) trace *0x2117c4 // 某个地址

trace 可以简写为 tr。

惯用变量$tpnum 记录最近设置的跟踪点号。

delete tracepoint [num]

永久删除一个或多个跟踪点。不带参数的话，默认删除所有的跟踪点。

例如：

(gdb) delete trace 1 2 3 // 删除三个跟踪点

(gdb) delete trace // 删除所有跟踪点

此命令可以简写为 del tr。

10.1.2 激活和禁用跟踪点

disable tracepoint [num]

禁用跟踪点 num，或者所有跟踪点，如果不指定参数的话。已禁用的跟踪点在下一次跟踪会话期将不再有效，但不会被系统遗忘。

enable tracepoint 命令可以再次激活已禁用的跟踪点。

enable tracepoint [num]

激活跟踪点 num，或者所有的跟踪点。已激活的跟踪点将在下一次跟踪会话期起效。

10.1.3 跟踪点通过计数

passcount [n [num]]

设置跟踪点的通过计数。使用通过计数可以自动中止跟踪会话。如果跟踪点通过计数是 n，那么跟踪会话会在跟踪点第 n 次执行到的时候自动中止。如果没有指定跟踪点号 num，通过计数命令将设置最近创

建的跟踪点。如果没有指定通过计数，那么跟踪会话会在一直执行，直到用户手动终止为止。

例如：

(gdb) passcount 5 2 // 跟踪点 2 在第 5 次执行时中止

(gdb) passcount 12 // 最近创建的跟踪点，在第 12 次执行时中断

(gdb) trace foo

(gdb) pass 3

(gdb) trace bar

(gdb) pass 2

(gdb) trace baz

(gdb) pass 1 // 在 foo 执行过 3 次，或者 bar 执行过 2 次，或者 baz 执行过 1 次时，中止跟踪

10.1.4 跟踪点操作列表

action [num]

此命令设置在跟踪点执行到时执行的操作。如果没有指定跟踪点号 num，此命令会为最近创建的跟踪点设置操作（因此创建跟踪点后执行 actions 命令就不必要再费事指定跟踪点号了）。接下来指明要执行的操作，每个操作一行，只有包含 end 的最后一行用来结束操作列表。到目前为止，只定义了 collect 和 while-stepping 操作。

要删除跟踪点的所有操作，输入'actions num'，接下来立即输入 end。

(gdb) collect data // 收集某些数据

(gdb) while-stepping 5 // 单步执行 5 次，收集数据

(gdb) end // 操作结束

下面的例子里，操作列表从 collect 命令开始，在跟踪点执行到时收集数据。那么，要单步执行和收集额外的数据，就要在设置单步执行时收集数据之后执行 while-stepping 命令。需要用 end 命令来终结 while-stepping 命令。最后，用 end 命令来终结操作列表。

```
(gdb) trace foo
(gdb) actions
Enter actions for tracepoint 1, one per line:
> collect bar,baz
> collect $regs
> while-stepping 12
> collect $fp, $sp
```

```
> end
end
collect expr1, expr2, ...
```

在跟踪点执行到时，收集指定表达式的结果。此命令可以接受以逗号分隔的任意有效表达式作为参数。另外，全局，静态或本地变量以外的，也支持下列特殊的参数：

$regs 收集所有寄存器

$args 收集函数的所有参数

$locals 收集所有本地变量

可以连续使用 collect 命令，每个 collect 都可以有单独的参数，或者一个 collect 命令带多个参数，以逗号分隔：效果是相同的。

命令 info scope（参见 13 章[符号]，143 页）非常适合查出哪些数据是要收集的。

while-steppingn

在跟踪点后执行 n 次单步跟踪，每执行一步都收集一次数据。while-stepping 命令后接要收集的数据列表（最后再接 end 结束 while-stepping 命令）：

> while-stepping 12

> collect $regs, myglobal

> end

>

while-stepping 可以缩写为 ws 或者 stepping。

10.1.5 跟踪点列表

info tracepoints [num]

打印跟踪点 num 的信息。如果不指定跟踪点号，将显示所有跟踪点的信息。每个跟踪点都包含下列信息：

- 跟踪点编号
- 是否激活或禁用
- 跟踪点地址
- 用 passcount n 命令设置的通过计数
- 用 while-stepping n 命令设置的单步执行次数

• 跟踪点设置于源代码的何处

• 用 actions 命令设置的操作列表

(gdb) info trace

Num Enb Address PassC StepC What

1 y 0x002117c4 0 0 <gdb_asm>

2 y 0x0020dc64 0 0 in g_test at g_test.c:1375

3 y 0x0020b1f4 0 0 in get_data at ../foo.c:41

(gdb)

本命令可缩写为 info tp。

10.1.6 开始和中止跟踪会话

tstart 此命令不需要参数。开始一次跟踪会话，并开始收集数据。如果不保留上一次跟踪会话期间收集的数据的话，可能会带来一些副作用。

tstop 此命令不需要参数。结束一次跟踪会话，并停止收集数据。

注意：一次跟踪会话和数据收集可能在达到跟踪点通过计数时自动终止（参见 10.1.3 节[跟踪点通过计数]，106 页），或者在跟踪缓冲区满的时候也可能导致终止。

tstatus 此命令显示当前跟踪数据收集的状态。

下面是关于目前为止我们介绍的命令的例子：

```
(gdb) trace gdb c test
(gdb) actions
Enter actions for tracepoint #1, one per line.
> collect $regs,$locals,$args
> while-stepping 11
> collect $regs
> end
> end
(gdb) tstart
[time passes ...]
(gdb) tstop
```

10.2 使用已收集的数据

跟踪会话结束以后，可以使用 GDB 命令来检查跟踪数据。基本的概念是，在达到跟踪点的时候每次收集一个跟踪快照，此外每次单步跟踪的时候都收集一次快照。所有这些快照都保存与跟踪缓冲区里，并且是从 0 开始连续编号的，所以可以供用户在以后来查看。要查看这些快照，需要明确指定是哪一个。如果远程代理指定了某个跟踪快照的话，在接到 GDB 的请求时，它会从缓冲区里读取此快照相应的内存和寄存器，而不是从实际的内存或寄存器里读取内容，反馈给 GDB。这就意味着 GDB 所有命令（print,info registers,backtrace 等等）都会像正在调试程序期间一样工作，就如同在跟踪点发生时那样。如果请求的数据不在缓冲区里，此请求将会失败。

10.2.1 tfind n

从缓冲区里选择一个跟踪快照的基本命令是 tfind n，tfind 查找编号为 n 的跟踪快照，从 0 开始。如果没有指定参数，那么会选择下一个快照。

下面是 tfind 命令各种形式。

tfind start

查找第一个快照。tfind 0 的同义词（因为 0 是第一个快照的编号）。

tfind none

停止调试跟踪快照，重新开始现场调试。

tfind end

和'tfind none'相同。

tfind 不带参数代表查找下一个跟踪快照。

tfind -查找当前快照的前调试过的快照。这就允许再次跟踪此前的步骤。

tfind tracepoint num

查找跟踪点编号 num 相对应的下一个快照。搜索从最近查看的跟踪点快照开始。如果不带参数 num 的，查找当前跟踪点的下一个快照

。

tfind pc addr

查找程序计数器地址 addr 对应的下一个快照。搜索从最近查看的跟踪点快照开始。如果不带参数的

话，查找当前快照的 PC 对应的下

一个快照。

tfind outside addr1,addr2

查找指定范围之外的 PC 对应的下一个快照。

tfind range addr1,addr2

查找指定范围内的 PC 对应的下一个快照。

tfind line [file:]n

查找源代码行 n 对应的下一个快照。如果指定了可选参数 file，那么指定是此源文件的代码行 n。搜索从最近查看的跟踪点快照开始。如果没有指定参数 n，代表查找下一行，而不是当前检查的这一行；因此，重复 tfind line 就好象在现场调试期间的单步跟踪一样。

tfind 命令的默认参数是特别设计的，使得它方便的在跟踪缓冲区里搜索。例如，不带参数的 tfind 命令选择下一个跟踪快照，而不带参数的 tfind -命令选择前一个跟踪快照。所以，用一个 tfind 命令，再按回车键重复就可以依次检查所有的跟踪快照。或者，用 tfind -再接着重复按回车键就可以反向检查快照了。不带参数的 tfind line 命令选择下一行代码对应的快照。不带参数的 tfind pc 命令选择当前堆栈帧上保存程序计数器 PC 对应的下一个快照。不带参数的 tfind tracepoint 命令选择当前跟踪点上收集的下一个快照。

除了让用户可以手动在跟踪缓冲区里搜索之外，这些命令也能方便的构建 GDB 脚本，搜索跟踪缓冲区并打印用户感兴趣的数据。因此，如果我们想要检查缓冲区里的每个跟踪帧里的 PC，FP 和 SP 寄存器，我们可以用下面这些命令：

```
(gdb) tfind start
(gdb) while ($trace frame != -1)
> printf "Frame %d, PC = %08X, SP = %08X, FP = %08X\n", \
$trace_frame, $pc, $sp, $fp
> tfind
> end
Frame 0, PC = 0020DC64, SP = 0030BF3C, FP = 0030BF44
Frame 1, PC = 0020DC6C, SP = 0030BF38, FP = 0030BF44
Frame 2, PC = 0020DC70, SP = 0030BF34, FP = 0030BF44
Frame 3, PC = 0020DC74, SP = 0030BF30, FP = 0030BF44
Frame 4, PC = 0020DC78, SP = 0030BF2C, FP = 0030BF44
Frame 5, PC = 0020DC7C, SP = 0030BF28, FP = 0030BF44
Frame 6, PC = 0020DC80, SP = 0030BF24, FP = 0030BF44
Frame 7, PC = 0020DC84, SP = 0030BF20, FP = 0030BF44
Frame 8, PC = 0020DC88, SP = 0030BF1C, FP = 0030BF44
Frame 9, PC = 0020DC8E, SP = 0030BF18, FP = 0030BF44
Frame 10, PC = 00203F6C, SP = 0030BE3C, FP = 0030BF14
```

或者，如果想要检查缓冲区里每行代码里的变量 X：

```
(gdb) tfind start
(gdb) while ($trace frame != -1)
> printf "Frame %d, X == %d\n", $trace_frame, X
> tfind line
> end
Frame 0, X = 1
Frame 7, X = 2
Frame 13, X = 255
```

10.2.2 tdump

本命令没有参数。tdump 打印在当前跟踪快照里所有收集到的数据。

```
(gdb) trace 444
(gdb) actions
Enter actions for tracepoint #2, one per line:
> collect $regs, $locals, $args, gdb_long_test
> end
(gdb) tstart
(gdb) tfind line 444
#0 gdb_test (p1=0x11, p2=0x22, p3=0x33, p4=0x44, p5=0x55, p6=0x66)
at gdb_test.c:444
444 printp( "%s: arguments = 0x%X 0x%X 0x%X 0x%X 0x%X 0x%X\n", )
(gdb) tdump
Data collected at tracepoint 2, trace frame 1:
d0 0xc4aa0085 -995491707
d1 0x18 24
d2 0x80 128
d3 0x33 51
d4 0x71aea3d 119204413
d5 0x22 34
d6 0xe0 224
d7 0x380035 3670069
a0 0x19e24a 1696330
a1 0x3000668 50333288
a2 0x100 256
a3 0x322000 3284992
a4 0x3000698 50333336
a5 0x1ad3cc 1758156
fp 0x30bf3c 0x30bf3c
sp 0x30bf34 0x30bf34
ps 0x0 0
```

```
pc 0x20b2c8 0x20b2c8
fpcontrol 0x0 0
fpstatus 0x0 0
fpiaddr 0x0 0
p = 0x20e5b4 "gdb-test"
p1 = (void *) 0x11
p2 = (void *) 0x22
p3 = (void *) 0x33
p4 = (void *) 0x44
p5 = (void *) 0x55
p6 = (void *) 0x66
gdb_long_test = 17 '\021'
(gdb)
```

10.2.3 save-tracepoints filename

本命令将当前所有跟踪点的定义以及它们的操作和通过计数保存到文件'filename'里， 以便以后的调试会话里使用。要读取保存的跟踪点定义，使用 sourcce 命令（参见 20.3 节[命令文件]，221 页）。

10.3 跟踪点的惯用变量

(int) $trace_frame

当前跟踪快照（也称为帧）编号，或者-1，如果没有选择快照的话。

(int) $tracepoint

当前跟踪快照的跟踪点。

(int) $trace_line

当前跟踪快照的行号。

(char []) $trace_file

当前跟踪快照的源文件。

(char []) $trace_func

包含$tracepoint 的函数名。

注意：$trace_file 不适于用 printf，用 output 打印。

下面是使用这些惯用变量的例子，单步执行跟踪快照并打印数据。

(gdb) tfind start

```
(gdb) while $trace frame != -1
> output $trace_file
> printf ", line %d (tracepoint #%d)\n", $trace_line, $tracepoint
> tfind
> end
```

译注：

stub，翻译成"存根"还是"代理"，"存根"感觉有点拗口，对应于中文的感觉不是很舒服；"代理"又觉得不能完整表达原文的意义。

第11章 调试使用覆盖技术的程序

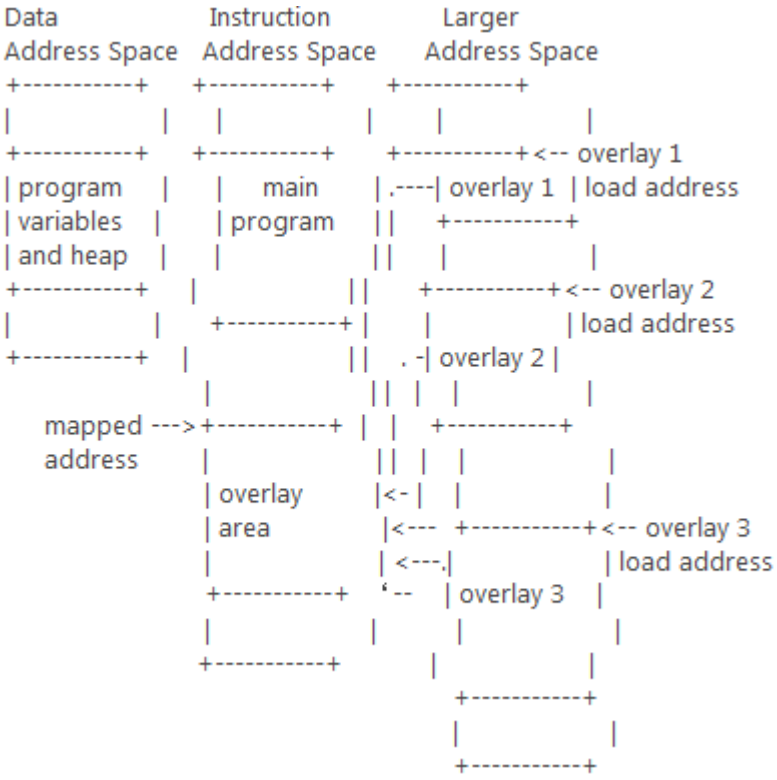
如果程序大到不能放到系统的内存里，你可以使用覆盖技术来规避这个问题。GDB 提供了调试使用覆盖技术的程序的支持。

11.1 覆盖是如何工作的

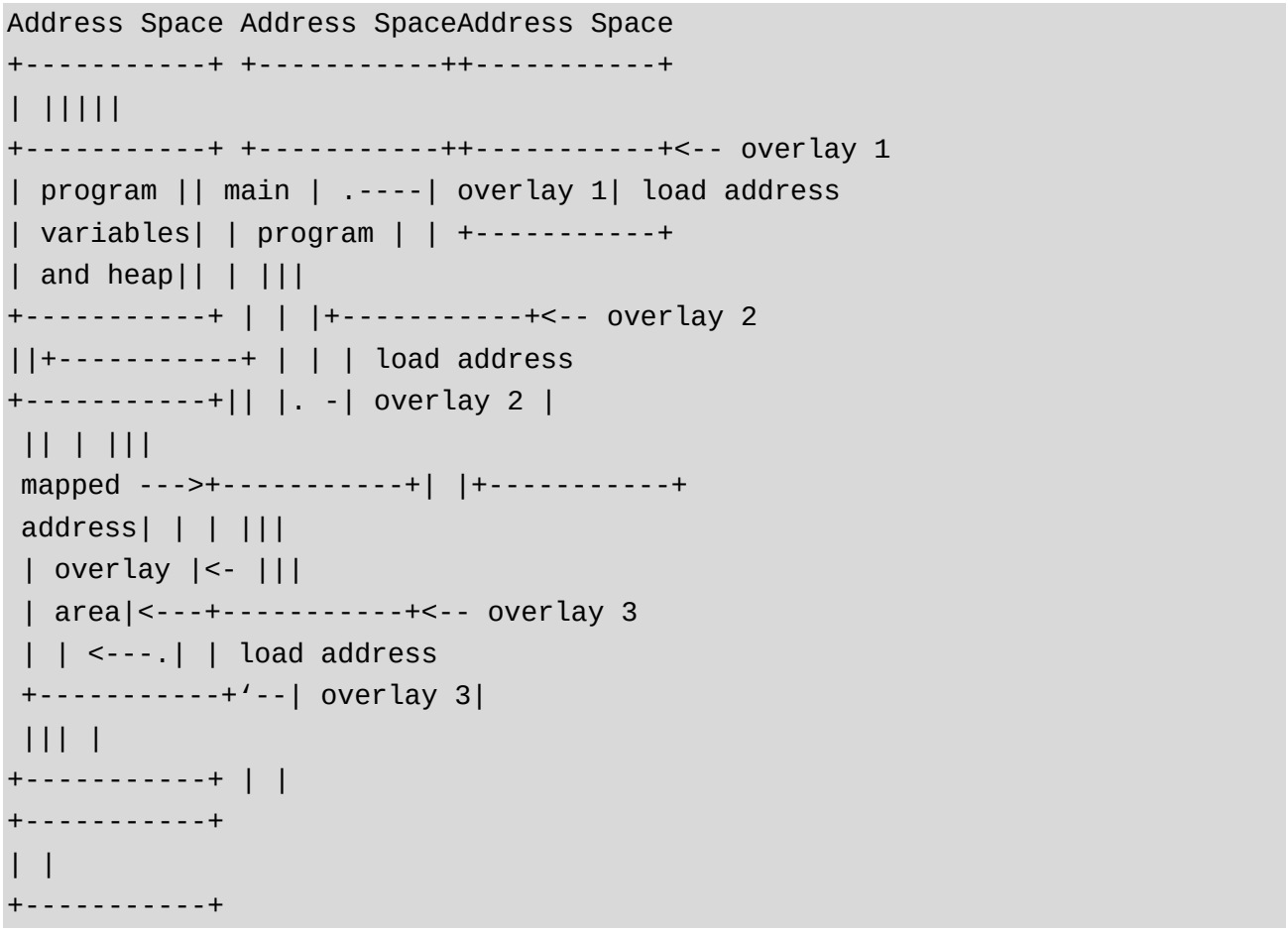
假设你的计算机的指令地址空间只有 64KB 长，而有超过此长度的内存可供别的方式访问：例如：特殊指令，段寄存器，或者内存管理硬件。进一步假设，你要将一个超过 64KB 长的程序加载到此计算机上运行。

一个解决方法是，将相对独立且相互之间没有直接调用的程序模块单独标识；这些模块称为覆盖段。将这些覆盖段从主程序里隔离，将它们的机器码放在更大的内存中。将主程序放在指令内存中，而同时保留至少足够大的空间来存放最大的覆盖段。

现在，要调用一个位于覆盖段上的函数，首先必须从大内存里将此覆盖段的机器代码复制到指令内存中来，并跳转到其入口点。




```
Data InstructionLarger
```



覆盖代码

这个图（参见[覆盖代码]，113 页）展示了将数据和指令地址空间分割的系统。要映射一个覆盖段，程序需要从大地址空间将代码复制到指令空间。由于上面介绍的所有复占位段使用相同的映射地址，所以在同一时刻只有一个可以映射。对于数据和指令使用同一个地址空间的系统，原理是相同的，只是程序变量，堆，主程序和覆盖段共享一个地址空间。

一个已经加载进指令内存且准备好使用的覆盖段称为已映射的覆盖段；其映射到地址是指令内存中的地址。指令空间中未加载（或部分加载）的覆盖段称为未映射覆盖段；其加载地址是在大内存中的地址。已映射地址也称为虚拟内存地址，或 VMA；加载地址也称为加载内存地址，或者 LMA。

不幸的是，对于将程序放置在有限指令内存的方法，覆盖段不完全是透明的方式。它们引进了一些新的全局行限制，在设计程序时必须时刻牢记在心：

• 在调用前或返回到一个覆盖段里的函数时，程序必须保证此段是已映射的。否则，调用或返回会将控制交给此实际地址，但错误的段，程序很可能会崩溃。

• 如果系统里映射覆盖段的处理很昂贵，那就需要仔细的选择覆盖段来减少其对程序性能的影响。

• 加载进系统的可执行文件必须包含每个覆盖段的指令，就是说覆盖段的加载地址，不是它们的已映

射地址。不过，每个重复占位的指令必须重新重新定位，其符号就如同段在它的已映射地址上。可以用 GNU 连接器脚本来为程序的各个段指定不同的加载和重定位地址；参见使用 ld:GNU 连接器，节"覆盖段描述"。

• 系统加载可执行文件的程序不但要能够将可执行文件的内容加载进大地址空间，也要能加载进指令和数据空间。

上面介绍的覆盖段系统是相对简单的，还可以有许多改进的方式：

• 如果系统有合适的体开关寄存器或者存储器管理硬件的话，可以用这些工具来让覆盖段的加载区域内容简单的映射到在经过映射后的指令地址空间里。这个方式很可能比复制覆盖段的内容到映射区域去要快得多。

• 如果覆盖段足够小，可以一次设置多个段，并且可以同时映射多个段。

• 可以用覆盖段来管理数据，就如同指令那样。通常来说，数据覆盖段比代码覆盖段更加不透明：只有在调用或返回到代码覆盖段的时候，才需要关心，数据覆盖段却时刻需要注意你所访问的数据。而且，如果改变了一个数据覆盖段的内容，在复制其它的数据覆盖段到此映射区域前，必须将其内容复制回其加载地址。

11.2 覆盖命令

要使用 GDB 对覆盖的支持，程序里每个覆盖必须对应可执行文件里的一个单独段。段虚拟内存地址和加载内存地址必须是覆盖段的映射和加载地址。将覆盖段和程序段等价使得 GDB 可以判断函数或变量的恰当的地址，由此覆盖段是否映射就可以推测。

所有 GDB 覆盖段命令都以 overlay 开头；可以缩写为 ov 获 ovly。这些命令是：

overlay off

禁用 GDB 对覆盖段的支持。禁用覆盖的支持之后，GDB 会假设所有的函数和变量都位于其映射地址上。缺省的，GDB 禁用对覆盖的支持。

overlay manual

激活手动覆盖调试。在此模式下，GDB 由用户告知映射的是哪个段，哪个段未映射，使用下面介绍的 overlay map-overlay 和 overlay unmap-overlay 命令。

overlay map-overlay overlay

overlay map overlay

通知 GDB 段已映射；overlay 必须是目标文件包含此覆盖段的名字。段映射后，GDB 假定可以在映射地址上查找到覆盖段的函数和变量。GDB 假定其他会重叠在此映射地址上的段都未映射。

overlay unmap-overlay overlay

overlay unmap overlay

通知 GDB 段解除映射；overlay 必须是目标文件包含此覆盖段的名字。段解除映射之后，GDB 假定段的函数和变量都位于其加载地址上。

overlay auto

激活自动覆盖调试。在此模式下，GDB 会查询一个由内部覆盖管理器维护的数据结构，以此得知映射的哪一个段。更多细节，参见 11.3 节[自动覆盖调试]，446 页。

overlay load-target

overlay load

从内部重读覆盖段表。通常，每次 GDB 在内部中断时都会自动重读此表，因此这个命令只在手动该表段映射时才需要使用。此命令只在使用自动段调试时才有用。

overlay list-overlays

overlay list

打印当前已映射段的列表，输出映射地址，加载地址和大小。

通常，GDB 在打印一个代码地址时，会包含函数名，和地址：

(gdb) print main

$3 = {int ()} 0x11a0 <main>

当覆盖段调试启用后，GDB 能够识别未映射段上的代码，并用*符号来标识次未映射函数名。例如，如果 foo 是位于未映射段上的函数，GDB 会如下打印：

(gdb) overlay list

No sections are mapped.

(gdb) print foo

$5 = {int (int)} 0x100000 <*foo*>

foo 对应的段映射后，GDB 会正常方式打印函数名：

(gdb) overlay list

Section .ov.foo.text, loaded at 0x100000 - 0x100034,

mapped at 0x1016 - 0x104a

(gdb) print foo

$6 = {int (int)} 0x1016 <foo>

覆盖调试启用后，GDB 就可以在覆盖段上正确地查找函数和变量的地址了。因此，大多数 GDB 命令，如 break 何 disassemble，就可以正常使用了，即使是在未映射段上也可以。然而，GDB 断点支持还是有一些限制：

• 只要 GDB 可以往在加载地址上的段上写入数据，就可以在此未映射段上的函数里设置断点。

• GDB 不能在未映射段上设置硬件和基于模拟器的断点。不过，如果在覆盖管理器的尾部上设置断点的话（并告诉 GDB 映射的是哪个段，如果手动管理重覆盖段的话），GDB 恰当地会重新设置其断点。

11.3 自动覆盖调试

只要覆盖管理器提供一些简单的协作，GDB 就可以自动跟踪段的映射情况。如果用 overlay auto 命令启用了自动段调试（参见 11.2 节[覆盖命令]，114 页），GDB 会从内部存储器里查找某些变量来，就可以得到当前段的状态。

要支持 GDB 自动覆盖调试，下面是覆盖管理器必须定义的变量：

_ovly_table:

这个变量必须是下面结构体的数组：

```
struct
{
/* The overlay's mapped address. */
unsigned long vma;
/* The size of the overlay, in bytes. */
unsigned long size;
/* The overlay's load address. */
unsigned long lma;
/* Non-zero if the overlay is currently mapped;
zero otherwise. */
unsigned long mapped;
}
```

_novlys:

这个变量必须是一个 4 字节的有符号整数，存储_ovly_table 的数量。

要判断一个特殊段映射与否，GDB 会从_ovly_table 里查找一个节点，比较其 vma 和 lma 是否和此段在可执行文件上的 VMA 和 LMA 相等。GDB 查到匹配的节点时，就会通过结构体成员 mapped 来判断此段是否已经映射了。

另外，段管理器也可能定义一个函数，函数名为_ovly_debug_event。如果定义了此函数，GDB 可以悄悄地在此函数上设置一个断点。如果段管理器在其改变了段表时调用这个函数，就可以使得 GDB 能够准确的跟踪段的映射情况，并且更新其上设置的断点。即使段还在 ROM 或者其

它不可写内存上，还没有执行的时候，这个功能也能让断点能够执行。

11.4 覆盖示例程序

在链接使用了覆盖的程序时，必须将段置于其加载地址上，而在运行时再将其重定位于映射地址上。要达到这个目的，必须写一个连接器脚本（参见节"覆盖介绍"使用 ld:GNU 连接器）。不幸的是，由于连接器脚本是和宿主系统，目标架构和目标内存布局高度相关的，本手册不能提供可移植的示例代码来演示 GDB 的覆盖支持。

不过，作为 GDB 源代码的测试套件的一部分，GDB 源代码发布版包含了一个覆盖程序，带有支持一些系统的连接器脚本。程序由'gdb/testsuite/gdb.base'目录下的文件组成：

‘overlays.c’

主程序文件。

‘ovlymgr.c’

覆盖管理器，由'overlays.c'调用。

‘foo.c’

‘bar.c’

‘baz.c’

‘grbx.c’

覆盖模块，由'overlays.c'加载和使用。

‘d10v.ld’

‘m32r.ld’

连接器脚本，链接生成测试程序的目标 d10v-els 和 m32r-elf 目标。

用 d10v-elf GCC 交叉编译器来生成测试程序：

```
$ d10v-elf-gcc -g -c overlays.c

$ d10v-elf-gcc -g -c ovlymgr.c

$ d10v-elf-gcc -g -c foo.c

$ d10v-elf-gcc -g -c bar.c

$ d10v-elf-gcc -g -c baz.c

$ d10v-elf-gcc -g -c grbx.c

$ d10v-elf-gcc -g overlays.o ovlymgr.o foo.o bar.o \

baz.o grbx.o -Wl,-Td10v.ld -o overlays
```

各个架构的编译的过程都一样，除了必须将相应的编译器和链接器替换为相应目标系统的编译器和链接器。

第12章 用 GDB 调试不同语言编写的程序

虽然编程语言通常都有一些共同的东西，但是事实上它们却很少能够用相同的形式来表述。例如，在 ANSI C 里，对指针 p 取值是用*p 来表示的，而在 Modula-2 里，用 p^。值的表示（或者显示）也可能不相同。C 里的 16 进制数的表示形如"0x1ae"，而在 Modula-2 里表示如"1AEH"。

GDB 里内置了某些语言的与其相关的信息，可以用程序原语言来执行类似上述的操作，也使得 GDB 可以和程序元语言一致的形式输出值。用来建立表达式的语言成为工作语言。

12.1 切换源代码语言

有两种方式可以控制工作语言--一种是让 GDB 自动设置，一种是用户手动设置。可以用 set language 命令来达到意图。GDB 在启动时会自动设置语言。工作语言用来决定如何翻译用户输入的表达式，如何打印这些结果，等等。

除了工作语言之外，GDB 能识别的源文件都有其自己的工作语言。对于某些目标文件格式，编译器可能会提示此特定源文件是那种语言。不过，大多数时候 GDB 都是从文件名里推断语言种类的。源文件的语言控制C++名字是否去修饰符--这种方式是backtrace以其自身语言来恰当地显示每一个帧。参见12.2节[显示语言]，120 页。

使用程序时，例如 cfront 或 f2c，这类程序产生 C 的目标代码，但其本身是用其它语言编写的，通常都会碰到问题。在此情况下，让程序在其 C 输出里使用#line 指令；这种方式可以让 GDB 知道原程序代码正确的语言，并能显示源代码，而不是显示其所产生的 C 代码。

12.1.1 文件扩展名和语言列表

如果源文件的扩展名是下列名称之一，那么 GDB 就会推断其语言。

‘.ada’

‘.ads’

‘.adb’

‘.a’ Ada 源文件.

‘.c’ C 源文件

‘.C’

‘.cc’

‘.cp’

‘.cpp’

‘.cxx’

‘.c++’ C++ 源文件

‘.m’ Objective-C 源文件

‘.f’

‘.F’ Fortran 源文件

‘.mod’ Modula-2 源文件

‘.s’

‘.S’汇编源文件。和 C 一样执行，但在单步跟踪时 GDB 不会跳过函数的序言。

另外，可以设置和文件扩展名相关的语言。参见 12.2 节[显示语言]，120 页。

12.1.2 设置工作语言

如果允许 GDB 自动设置语言的话，那么在程序和调试会话里表达式就以相同的方式转换。

如果用户期望的话，也可以手动设置。要手动设置，执行命令'set language lang'，lang 是语言的名称，例如 c 和 modula-3。要得到 GDB 支持的语言列表，执行命令'set language'。

手动设置语言将妨碍 GDB 自动更新工作语言。在调试工作语言和源代码语言不想同时，如果两种语言都可以接受表达式---但表示不同的事情的话，就可能导致一些困惑了。例如，如果当前的源文件是用 C 写的，GDB 分析 Modula-2，如下的命令：

print a = b + c

可能不会得到你所期望的那个结果。在 C 里，这个命令表示得失将 b 和 c 相加，并将其结果赋值给 a。结果是打印出 a 的值。在 Modula-2 里，这个命令表示比较 a 和 b+c 的结果，产生一个布尔值。

12.1.3 让 GDB 推断源语言

要让 GDB 自动设置工作语言，用'set language local'或者'set language auto'。GDB 就会推断工作语言了。也就是说，程序在一个堆栈帧里中断的时候（通常是遇到了一个断点），GDB 就会为此栈上的函数

设置已记录的工作语言。如果此帧的语言未知（那就是说，此帧上的函数或代码块定义于某个源文件，此文件的扩展名却是未知的），当前工作语言不会改变，GDB 会发出一个警告。

这个命令对于多数只用同一种源代码语言编写的程序来说不是必须的。不过，用同一个源代码语言编写的程序模块和库可以被不同源语言编写的主程序调用。使用'set language auto'可以免除用户手动设置工作语言之苦。

12.2 显示语言

下面的命令可以助你找出何种工作语言，并找出程序是哪种源代码语言编写的。

show language

显示当前工作语言。可以用这个语言来执行命令（如 print）来建立和计算表达式，表达式里可以引用程序里的变量。

info frame

显示此帧的源代码语言。如果从此帧上使用了一个标志符的话，这个语言就变成工作语言。要识别列于此处的其他信息，参见 6.4 节

[堆栈帧的信息]，65 页。

info source

显示此源代码的源代码语言。要识别列于此处的其他信息，参见 13 章[检验符号表]，143 页。

在某些异常情况下，源文件的扩展名可能不在标准列表里。可以明确将语言和扩展名关联起来。

set extension-language ext language

设置 GDB 将扩展名 ext 的文件认为是由源代码语言 language 编写的。

info extensions

列出所有的文件扩展名和相关连的语言。

12.3 类型和域检查

警告：在本版手册里，包括了 GDB 类型和域检查命令，但还没有效用。本节文档介绍了预期的功能。

通过编译期和运行时检查，某些语言的设计可以防止程序员犯一些常见的错误。这些检查包括函数和操作符的参数的类型检查，保证在运行时捕获算术溢出。一旦通过排除类型不匹配的错误，程序编译好之

后，检查这些有助于保证程序的正确性，并能在程序运行时提供对域错误的实时检查。

GDB 可以检查如上述条件的检查。虽然不检查程序里的状态，GDB 可以检查直接传递给它的表达式，例如 print 命令可以计算表达式的值。如同工作语言一样，GDB 也可以基于程序的源语言来决定是否自动检查。关于缺省的语言支持设置，参见 12.4 节[语言支持]。

12.3.1 类型检查概述

某些语言，如 Modula-2，是强类型的，也就是说操作符和函数的参数必须是正确的类型，否则就会发生错误。这个检查防止类型不匹配的错误引起运行时错误。例如，

1 + 2 => 3

但

[error] 1 + 2.3

第二个例子的错误是因为序数（CARDINAL）1 和实数（REAL）2.3 类型不相容。

对用于 GDB 命令的表达式，用户可以设置让 GDB 类型检查器不检查；将任意不匹配作为错误并拒绝表达式；或者在遇到类型不匹配时只发出警告，但会计算表达式。如果选择了最后一项，GDB 会计算如上述第二个例子的表达式，但会发出一个警告。

即使关闭了类型检查，也还有其他和类型有关的理由来防止 GDB 去对表达式进行计算。例如，GDB 不知道如何将一个 int 型和 struct foo 型相加。这些特殊的类型错误和使用的语言无关，常常在表达式里引起，例如前面介绍的那个例子，其错误是不知道如何计算。

每种语言都定义了其对类型检查的严格程度。例如，Modula-2 和 C 要求算数操作符的参数都是数字。在 C 里，枚举类型和指针可以转换为数值型，所以他们对于算数操作符也是有效的。关于特定语言的更多细节，参见 12.4 节[语言支持]。

GDB 提供了附加命令来控制类型检查器：

set check type auto

设置自动类型检查，由当前工作语言决定。各种语言的缺省设置，参见 12.4 节[语言支持]。

set check type on

set check type off

设置类型检查启用或关闭，覆盖掉当前工作语言的缺省设置。如果设置不匹配语言的缺省值就会发出一个警告。如果启用了类型检查，在计算表达式的时候发现类型不匹配的话，GDB 会打印警告信息并放

弃计算表达式。

set check range warn

在 GDB 域检查器发现域错误的时候输出警告信息，但会尝试计算表达式的值。由于其他缘故，表达式计算可能会失败，例如访问不属于进程的内存（许多 Unix 系统的典型例子）。

show range

显示当前域检查器的设置，以及 GDB 是否自动设置了域检查器。

12.4 语言支持

GDB 支持 C，C++，Objective-C，Fortran，Java，Pascal，汇编，Modula-2 和 Ada 语言。GDB 的某些功能可以用于和语言无关的表达式：GDB@，::操作符和'{type}addr'指令（参见 8.1 节[表达式]，75 页）可以和任何 GDB 支持的语言的指令一起用。

接下来的章节详细的介绍了 GDB 对于每种语言的支持程度。这些章节不是语言导论和参考，而只是 GDB 表达式解析器能接受何种表达式的参考指引，以及对于各种不同语言的输入输出格式。有许多关于这些语言很好的书；请参考这些书的参考或导论。

12.4.1 C 和 C++

由于 C 和 C++关系很近，GDB 的很多功能都能在这两种语言上使用。在这些情况下，我们一起讨论这两种语言。

C++调试工具是由 C++编译器和 GDB 共同实现的。因此，要有效调试 C++代码，必须用支持 C++的编译器上编译 C++程序，例如 GNU g++，或 HP ANSI C++编译器（aCC）。

使用 GNU C++时要得到最佳效果，应使用 DWARF 2 调试格式；如果此格式不能在你的系统里执行，请尝试用 stabs+调试格式。你可以用 g++ 命令行选项'-gdwarf-2'和'-gstabs+'选择格式。“Options for Debugging Your Program or GCC” in Using the gnu Compiler Collection (GCC)。

12.4.1.1 C 和 C++操作符

操作符必须定义于明确类型的值。例如，+定义于数值，但不是结构体。操作符常常定义与类型组。

下列定义表述了 C 和 C++的目的：

• 整型类型包括 int 和其存储类型限定符；char；enum；以及 C++的 bool。

• 浮点类型包括 float，double 和 long double（如果目标平台支持的话）。

• 指针类型包括所有定义为(type*)的类型。

• 标量类型包括所有上述类型。

GDB 支持下列操作符。下面将以递增的次序列举：

，逗号和顺序操作符。用逗号分隔的表达式列表将从左到右计算，最后计算整个表达式的结果。

=赋值。赋值表达式的值是被赋值的值。定义于标量类型。

op=用于形如 a op=b 的表达式里，并转换到 a = a op b。op=和=具有相同的优先顺序。op 是下面的任意操作符之一：|, ^, &, <<, >>,

+, -, *, /, %。

?:三进制操作符。a?b:C 可以如此认为：如果 a 那么 b，否则 c。a 必须是整形类型。

|| 逻辑或。定义于整形类型。

&&逻辑与。定义于整形类型。

| 位或。定义于整形类型。

^位异或。定义于整形类型。

& 位与。整形类型。

==,!=相等和不相等。表达式的值 0 代表假，非 0 代表真。

<, >, <=, >=

小于，大于，小于等于，大于等于。定义与标量类型。表达式的值 0 代表假，非 0 代表真。

<<, >>左移，右移。定义于整形类型。

@ GDB"人工数组"操作符（参见 8.1 节[表达式]，75 页）。

+,- 加和减。定义于整形类型，浮点类型和指针类型。

*, /, % 乘，除和求余。乘和除定义于整形和浮点类型。求余定义与整形类型。

++，--自增和自减。位于变量前面，此操作符会于变量使用前执行；位于变量后面，操作符会在变量使用后执行。

* 对指针取值。定义于指针类型。和++的优先级相同。

&地址操作符。定义于变量。和++的优先级相同。

对于调试 C++，除了在 C++语言里的用法，GDB 还实现了'&'的另一种用法：可以用'&(&ref)'来查看

C++引用变量（用'&ref'声

明）的存储地址。

- 负。定义于整形和浮点类型。和++的优先级相同。

! 逻辑非。定义于整形类型。和++的优先级相同。

~按位求补运算符。定义与整形类型。按位求补运算符。

.,-> 结构体成员，结构体指针成员。为方便起见，GDB 视这两种操作符为等同，基于存储类型信息来选择是否对一个指针取值。定

义于结构体和联合数据。

.*,->*对指针指向的成员取值。

[]数组索引。a[i]定义如*(a+i)。和->的优先级相同。

()函数参数列表。和->的优先级相同。

:: C++范围解析操作符。定义于结构体，联合和类。

:: 双冒号也表示 GDB 范围解析操作符（参见 8.1 节[表达式]，75 页）。和::的优先级相同，如上。

如果在用户代码里重新定义了操作符，通常 GDB 会尝试用重新定义的版本来执行，而不用预定义的方式。

12.4.1.2 C 和 C++常量

GDB 允许用户用下列方式表示 C 和 C++的常量：

• 整形常量是数字系列。8 进制常量的由'0'(即 0)开头，16 进制常量由'0x'或'0X'开头。常量也可以用字符'l'结尾，指明次常量应该视为 long(长整型)类型。

• 浮点指正常量是数字系列，接着一个十进制小数点，再接着数字系列，还可以接着一个指数。指数形如'e[[+]|-]nnn'，这里 nnn 是另一个数字系列。如是正的指数，'+'可选用。浮点常量也可以用字符'f'或者'F'结尾，指明此常量应视为 float（而不是默认的 double）类型；或者用字符'l'或者'L'来结尾，指明是 long double 常量。

• 枚举常量有枚举变量，或者是枚举变量对应的整数组成。

•字符常量是由单引号(')括起来的单个字符，或者一个数字--字符对应的原始数值(通常是其 ASCII 值)组成。在引号里，单字符可以用一个词或者转义序列表示，形如'\nnn'，这里 nnn 是字符序数的八进制表现形式；或者形如'\x'，这里'x'是预定义的特殊字符---例如，'\n' 表示新行（newline）。

• 字符串常量由双引号(")括起来的字符常量系列组成。任何有效的字符常量（如上所述）都可以。字

符串里的双引号必须跟在反斜杠后面，例如'"a\"b'c"'是 5 字符的字符串。

• 指针常量是整形值。也可以用 C 操作符'&'将指针写道常量里。

• 数组常量用花括号'{'和'}'括起来，用逗号分隔；例如，'{1,2,3}'是 3 元素的整形数组，‘{{1,2}, {3,4}, {5,6}}’是 3 行 2 列的数组， 而‘{&"hi", &"there", &"fred"}’是 3 元素的指针数组。

12.4.1.3 C++表达式

GDB 可以处理大多数 C++表达式。

注意：只有在使用正确的编译器和调试格式的情况下，GDB 才能调试 C++代码。目前，在用 GCC 2.95.3 或者 GCC 3.1 或者更新的版本上，使用选项‘-gdwarf-2’或者‘-gstabs+’的 C++代码上，GDB 工作的最好。DWARF 2 优于 stabs+。大多数 GCC 的配置选用 DWARF 2 或者 stabs+作为其默认调试格式，所以通常不需要手动指定调试格式。用其它编译器编译的和/或者调试格式的代码，GDB 可能工作的不太好，也可能根本就不起效。

1.允许调用成员函数；调用表达式如下：

count = aml->GetOriginal(x, y)

2.当成员函数可用时（在选定的堆栈帧上），表达式和成语函数具有相同的可用的名字空间；那就是说，用 C++相同的规则，GDB 允许 隐式引用类实例指针 this。

3.可以调用重载函数；GDB 会在某些限制下解决函数调用中函数的正确的定义。涉及到用户定义类型的转换，构造函数的调用，不在程 序里的模板实例，GDB 都不会去进行重载解析。GDB 也不能处理省略参数列表或者默认参数的情况。

GDB 可以执行整形转换和提升（例如在函数参数 char 提升到 int），浮点提升，算术转换，指针转换，类对象转换到基类，以及标准转 换如函数和数组到指针；GDB 需要精确匹配函数参数的数量。

总是执行重载解析，除非指定了 set overload-resolution off。参见节 12.4.1.7[GDB 的 C++功能]，128 页。

要使用明确的函数签名来调用一个重载函数，必须指定 set overload-resolution off，如 p ’foo(char,int)’(’x’, 13)

GDB 命令补全功能可以简化这些；参见节 3.2[命令补全]，19 页。

4.GDB 能理解定义为 C++引用的变量；可以在表达式里使用这些变量，如同在 C++源代码里一样--它们是自动取值的。

在 GDB 显示堆栈帧的时候，引用变量的值不睡显示在参数列表里（和其它类型的变量不同）；这样能

避免聚集，因为引用变量常常用 于大的结构体。总是显示引用变量的地址，除非指定了'set print address off'。

5.GDB 支持 C++名字解析操作符::--表达式可以使用此操作符就如同在程序里那样。由于一个范围可以定义于另外一个，如果需要可以 重复使用::，例如表达式如'scope1::scope2::name'。

在 C 和 C++调试里，GDB 也可以参考源文件来解决名字空间（参见 8.2 节[程序变量]，76 页）。

另外，要是用 HP 的 C++编译器，GDB 支持虚函数调用，打印对象的虚基类，调用基类子对象的函数，对象转换，以及调用用户定义的操作符。

12.4.1.4 C 和 C++缺省值

如果允许 GDB 自动设置类型和域检查，工作语言转换到 C 或者 C++时，这两个设置的缺省设置都是 off。不论用户或 GDB 是否选择工作语言，GDB 都是这样处理的。

如果允许 GDB 自动设置语言，GDB 会识别文件名以'.c'，'.C'或者'.cc'等等结尾的文件，在 GDB 进入以这些文件编译的代码时，GDB 会将工作语言设置为 C 或者 C++。更多细节，参加节 12.1.3[设置 GDB 推断源文件语言]，120 页。

12.4.1.5 C 和 C++类型和域检查

在 GDB 分析 C 或者 C++表达式，缺省的，不使用类型检查。不过，如果你将类型检查打开，GDB 认为这两种变量类型相等，如果：

- 这两种变量是结构化的且具有相同的结构，联合，或者枚举标签。
- 这两种变量具有相同的类型名，或者用是用 typedef 声明的相等的类型。

域检查，如果打开的话，用于数学操作。由于下标常常用于索引一个指针，而指针本身并不是数组，数组下标不检查。

12.4.1.6 GDB 和 C

set print union 和 show print union 命令应用于 union 类型。要是设置为'on'，任何 struct 或者 class 里的 union 都会打印。否则，会用'{...}'替代。

@操作符有助于调试用指针和内存分配函数分配的动态数组。参见节 8.1[表达式]，75 页。

12.4.1.7 GDB 的 C++功能

GDB 某些命令对于 C++特别有用，而有些是特殊设计用于 C++。下面是这些命令的概述：

breakpoint menus

要想在一个重载函数里设置断点，GDB 断点菜单可以助你指定哪个函数定义上设置。参见节 5.1.8[断点菜单]，52 页。

rbreak regex

用正则表达式设置断点，正则表达式断点对于在不是任何特殊类的成员重载函数上设置断点，很有帮助。参见节 5.1.1[设置断点]，40 页。

catch throw

catch catch

使用这两个命令，调试 C++异常处理。参见节 5.1.3[设置捕获点]，47 页。

ptype typename

打印继承关系和其它有关类型 typename 的信息。参见 13 章[检验符号表]，143 页。

set print demangle

show print demangle

set print asm-demangle

show print asm-demangle

控制是否以源代码形式显示 C++符号，在现实 C++源代码和汇编的时候。参见节 8.7[打印设置]，82 页。

set print object

show print object

选择是否打印派生（实际的）或者声明的对象类型。参见节 8.7[打印设置]，82 页。

set print vtbl

show print vtbl

控制打印虚函数表的格式。参见节 8.7[打印设置]，82 页。（vtbl 命令在用 HP ANSI C++编译器编译（aCC）的程序上不起效。）

set overload-resolution on

激活 C++表达式计算的重载解析。缺省打开。对于重载函数，GDB 使用 C++转换规则计算参数并搜

索签名匹配参数类型的函数（更多细节，参见节 12.4.1.3[C++表达式]）。如果不能找到匹配项，将打印一个消息。

set overload-resolution off

关闭 C++表达式计算的重载解析。对于不是类成员函数的重载函数，GDB 选择在符号表里找到的此名字的第一个函数，不论其参数是否是正确的类型。对于是类成员函数的重载函数，GDB 搜索签名精确匹配参数类型的函数。

show overload-resolution

显示当前重载解析的设置。

Overloaded symbol names

使用相同的用于声明 C++符号的标识，可知指定重载符号的特殊定义：type symbol(types)而不仅仅是 symbol。也可以用 GDB 命令行字补全功能来列出可用的选择，或者来补全类型列表。参见节 3.2[命令补全]，关于如何完成的更多细节。

12.4.1.8 十进制浮点格式

GDB 可以十进制浮点格式来检验，设置和执行数字计算，在 C 语言里相应的用_Decimal32，_Decimal64 和 _Decimal128 类型的扩展支持十进制浮点算术。

在实际应用中有两种编码方式，依赖于系统架构：x86 和 x86-64 平台下是 BID(Binary Integer Decimal)，PowerPC 是 DPD(Densely PackedDecimal)。GDB 为已配置的系统使用恰当的编码格式。

由于'libdecnumber'的限制，此库用于操作十进制浮点数，不能将宽度超过 32 位的整形转换到十进制浮点。

另外，要模仿 GDB 的二进制浮点计算行为，十进制浮点操作里的错误检查将忽略下溢，上溢和 0 除异常。

PowePC 架构，GDB 提供了伪寄存器集来检查存储于浮点寄存器里的_Decimal128 值。更多细节，参见节 18.4.7[PowerPC]，207 页。

12.4.2 Objective-C

本节提供的命令和命令选项的信息有助于调试 Objective-C 代码。参见 13 章[符号]，143 页，和 13 章[符号]，143 页，更多 Objective-C 相关的命令。

12.4.2.1 命令里的方法名

下面是接受 Objective-C 方法名的扩展命令，

- clear
- break
- info line
- jump
- list

完全合格的的 Objective-C 方法名应声明如下：

-[Class methodName]

这里减号用来表示对象实例方法，加号（没显示出来）用来表示一个类方法。类名 Class 和方法名 methodName 用括号括起来，和 Objective-C 源代码里消息表示方法相似。例如，要在调试程序里的类 Fruit 的实例 create 方法上设置一个断点，输入：

break -[Fruit create]

要显示类方法 initialize 附近的 10 行程序代码，输入：

list +[NSText initialize]

在目前的 GDB 版本上，需要输入加号和减号。以后的 GDB 版本，加号和减号会是可选的，但可以用来缩小搜索范围。也可能只需要指定一个方法名：

break create

必须指定完整的方法名，包括冒号。如果程序源文件包含多个 create 方法的话，将以编号方式打印实现了此方法的类列表。用编号指定所选择的方法，或者输入'0'来退出，如果不选择的话。

另一个例子是清除在 NSWindow 类的 makeKeyAndOrderFront:方法上的断点，输入：

clear -[NSWindow makeKeyAndOrderFront:]

12.4.2.2 和 Objective-C 协作的 Print 命令

print 命令也扩展接受方法。例如：

print -[object hash]

让 GDB 发送 hash 消息给对象 object 并打印结果。而且，也增加了附加命令，print-object 或缩写为 po，此命令打印对象的描述。不过，此命令依赖于 Objective-C 库实现了特殊的钩子函数，

_NSPrintForDebugger。

12.4.3 Fortran

GDB 可以调试用 Fortran 写的程序，但目前只支持 Fortran 77 语言的特征。

某些 Fortran 编译器（GNU Fortran 77 和 Fortran 95 编译器）在变量和函数上添加一个下划线。在调试用这些编译器的编译的程序时，引用变量和函数时需要加上下划线。

12.4.3.1 Fortran 操作符和表达式

操作符必须定义与特定类型上。例如，+定义于数字，但不能定义于字符或者其它非算术类型。操作符通常定义于类型组。

**求幂操作符。将第一个操作数作为第二个操作数的乘方。

: 域操作符。常用于数组形式，表示数组的一部分。

12.4.3.2 Fortran 的缺省值

Fortran 符号通常是不区分大小写的，所以 GDB 缺省使用大小写不区分地匹配 Fortran 符号。用'set case-insensitive'命令改变此设置，更多细节，参见 13 章[符号]，143 页。

12.4.3.3 Fortran 的特殊命令

GDB 实现了一些支持 Fortran 相关特征的命令，例如显示公共块。

info common [common-name]

此命令打印名为 common-name 的公共块里的数据值。不带参数的话，打印当前程序所有可访问的共块的名字。

12.4.4 Pascal

目前不能调试使用集合，子域，文件变量和嵌套函数的 Pascal 程序。GDB 不支持进入表达式，打印数值，或者相似的使用 Pascal 句法。

Pascal 特定的命令 set print pascal_static-members 控制是否显示静态 Pascal 对象。参见 8.7 节[打印设置]，82 页。

12.4.4.1 Modula-2

GDB 对 Modula-2 的扩展支持智能支持由 GNU Modula-2 编译器（目前尚处于开发中）编译的程序。目前尚未支持其他 Modula-2 编译器，试图调试此类编译器产生的可执行程序的话，GDB 会在读入可执行程序的符号表时打印一个出错信息。

12.4.4.2 操作符

操作符必须定义域特定类型的数据上。例如，+定义于数字，但不能在结构体上。操作符通常定义于类型组。就 Modula-2 而言，有下列定义：

- 整形类型由 INTEGER，CARDINAL，以及它们的子域组成。
- 字符类型由 CHAR 和其子域组成。
- 浮点类型由 REAL 组成。
- 指针类型由任意声明为 POINTER TO 类型组成。
- 向量类型由上述所有类型组成。
- 集合类型由 SET 和 BITSET 类型组成。
- 布尔类型由 BOOLEAN 组成。

下面的介绍支持的操作符，以增序排列：

，函数参数或数组索引分割符。

:= 赋值。var := value 的值是 value。

<, >小于，大于，定义于整型，浮点型和枚举型。

<=, >=小于等于，大于等于，定义于整型，浮点型和枚举型，集包含和集合类型。和<优先顺序相同。

=, <>, #

相等和两种不相等的表示，向量类型也起效。和<优先顺序相同。在 GDB 脚本里，只有<>可以表示不相等，因为#和脚本注释

符冲突。

IN 集合成员。定义与集合类型和其成员的类型。和<优先顺序相同。

OR 布尔析取。定义于布尔型。

AND,& 布尔合取。定义于布尔型。

@ GDB“伪数组”操作符（参见 8.1 节[表达式]，75 页）。

+, - 加和减，定义于整型和浮点型，联合和集合差类型。

*乘，定义于整型和浮点型，或者集合交集。

/ 除，定义于浮点类型，或者集合类型的对称集合差。和*优先顺序相同。

DIV, MOD

整型出和求余。定义于整型。和*优先顺序相同。

- 负。定义于 INTEGER 和 REAL 数据。

^ 对指针取值。定义于指针类型。

NOT 布尔非。定义于布尔型。和^有限顺序相同。

.RECORD(记录)域选择符。定义于 RECORD 数据。和^优先顺序相同。

[]数组索引。定义于 ARRAY 数据。和^优先顺序相同。

()过程参数列表。定义于 PROCEDURE 对象。和^优先顺序相同。

::, .GDB 和 Modula-2 域操作符。

警告：设置表达式和它们的操作还不支持，GDB 在集合上使用操作符 IN，+， -， *， /， =， ,， <>，#，<=，>=会引起错误。

12.4.4.3 内建函数和过程

Modula-2 也提供了一些内建的过程和函数。在介绍之前，先介绍下面用到的元变量：

a 表示一个数组变量。

c 表示一个 CHAR 常量或变量。

i 表示一个变量或者整型常量。

m 表示属于一个集合的标识符。通常和原变量 s 一起用于同一个函数。s 的类型英格是 SET OF mtype（这里 mtype 是 m 的类型）。

n 表示一个整型或浮点类型的变量或者常量

r 表示一个浮点类型的变量或常量。

t 表示一个类型。

v 表示一个变量。

x 表示一个变量或常量，其类型是多种类型中的其一。详细参见函数的解释。

所有 Modula-2 内建过程都返回一个结果，见下。

ABS(n)返回 n 的绝对值。

CAP(c)如果 c 是小写字符，返回它对应的大写字符，否则返回 c。

CHR(i)返回值是 i 的字符。

DEC(v)将变量 v 的值减 1.返回新的值。

DEC(v,i)

将变量 v 的值减 i。返回新的值。

EXCL(m,s)

将元素 m 从集合 s 里删除。返回新的集合。

FLOAT(i)

返回整形变量 i 对应的浮点数。

HIGH(a)返回数组 a 的最后一个元素的索引。

INC(v) 将变量 v 增加 1。返回新的值。

INC(v,i)

将变量 v 的值增加 i。返回新的值。

INCL(m,s)

如果 m 不在集合 s 的话，将元素 m 加入集合 s。返回新的集合。

MAX(t)返回类型 t 的最大值。

MIN(t)返回类型 t 的最小值。

ODD(i)如果 i 是一个奇数，返回 TRUE。

ORD(x)返回参数的原始值。例如，一个字符的原始值是其 ASCII 值（机器支持的 ASCII 字符集）。x 必须是一个序数类型，

包括整形，字符和枚举类型。

SIZE(x) 返回参数的大小。x 可以是变量或类型。

TRUNC(r)

返回 r 的实数部分。

TSIZE(x)

返回参数的大小。x 可以是变量或类型。

VAL(t,i)

返回类型 t 的成员，其值是 i。

警告：集合和它们的操作还未支持，所以使用过程 INCL 和 EXCL 的话，GDB 会视为错误。

12.4.4.4 常量

GDB 允许用下列方式表示 Modula-2 的常量：

• 整型常量，此类型常量是数据序列。在用于表达式中时，常量会转换为和表达式其它部分相容的类型。16 进制整数应该后接'H'，二进 制整数后接'B'。

• 浮点常量也是数据序列，后接一个小数点和另外一个数据序列。可以指定一个可选的指数，其形式是'E[+|-]nnn'，这里'[+|-]nnn'表示指数。浮点常量的所有数据都必须是有效的 10 进制数据。

• 字符常量由用一对单引号或双引号括起来的单个字符组成。用 C 风格的转义序列也是可以的。转义序列的简明介绍，参见节 12.4.1.2[C 和 C++常量]，125 页。

• 字符串常量由一对单引号或双引号括起来的字符序列组成。用 C 风格的转义序列也是可以的。转义序列的简明介绍，参见节 12.4.1.2[C 和 C++常量]，125 页。

• 枚举常量由由枚举标志符组成。

• 布尔常量由标志符 TRUE 和 FALSE 表示。

• 指针常量由整数值组成。

• 集合常量目前尚未支持。

12.4.4.5 Modula-2 类型

目前，GDB 可以在 Modula-2 上下文中打印下列数据类型：数组类型，记录类型，集合类型，指针类型，过程类型，枚举类型，子域类型和基类。还可以打印用这些类型定义的变量内容。本节所示的 GDB 调试例子，包括几个源代码的样例。

第一个例子包含下列的代码段：

```
VAR
 s: SET OF CHAR ;
 r: [20..40] ;
```

可以用 GDB 查询 r 和 s 的类型和值。

(gdb) print s

{'A'..'C', 'Z'}

(gdb) ptype s

SET OF CHAR

(gdb) print r

21

(gdb) ptype r

[20..40]

类似的，如果源代码声明 s 为：

VAR

  s: SET ['A'..'Z'] ;

那么可以查询 s 的类型：

(gdb) ptype s

type = SET ['A'..'Z']

注意，目前还不能用调试器交互的操作集合表达式。

下面的例子演示在 Modula-2 里如何生命数组和怎样用 GDB 来打印其类型和内容：

VAR

  s: ARRAY [-10..10] OF CHAR ;

(gdb) ptype s

ARRAY [-10..10] OF CHAR

注意，数组处理尚未完全实现，虽然类型可以正确打印，表达式处理仍然会假设所有的数组的下标是0，本例中将不会认为是-10。

下面还有一些 Modula-2 类型相关的例子：

TYPE

  colour = (blue, red, yellow, green) ;

  t = [blue..yellow] ;

VAR

  s: t ;

```
BEGIN
 s := blue ;
```

GDB 交互显示如何查询数据类型和变量的值。

```
(gdb) print s
$1 = blue
(gdb) ptype t
type = [blue..yellow]
```

在这个例子中声明了一个 Modula-2 的数组，打印其内容。观察到数组的内容以如同 C 对应的数据那样的相同方式写入。

```
VAR
s: ARRAY [1..5] OF CARDINAL ;
BEGIN
s[1] := 1 ;
(gdb) print s
$1 = {1, 0, 0, 0, 0}
(gdb) ptype s
type = ARRAY [1..5] OF CARDINAL
```

Modula-2 语言对于 GDB 的接口也理解指针类型，如下例所示：

```
VAR
s: POINTER TO ARRAY [1..5] OF CARDINAL ;
BEGIN
NEW(s) ;
s^[1] := 1 ;
```

可以要求 GDB 描述 s 的类型。

```
(gdb) ptype s
type = POINTER TO ARRAY [1..5] OF CARDINAL
```

GDB 可以处理复合类型，正如本例所示。下面将数组类型，记录类型，指针类型和子域类型结合起来：

```
TYPE
```

```
 foo = RECORD
 f1: CARDINAL ;
f2: CHAR ;
f3: myarray ;
END ;
myarray = ARRAY myrange OF CARDINAL ;
myrange = [-2..2] ;
VAR
s: POINTER TO ARRAY myrange OF foo ;
```

可以让 GDB 描述 s 的类型，如下所示。

```
(gdb) ptype s
type = POINTER TO ARRAY [-2..2] OF foo = RECORD
f1 : CARDINAL;
f2 : CHAR;
f3 : ARRAY [-2..2] OF CARDINAL;
END
```

12.4.4.6 Modula-2 的缺省设置

如果 GDB 自动设置了类型和域检查的话，那么在工作语言切换为 Modula-2 的时候这两项设置将是激活的。不论用户或 GDB 选择工作语言与否，都是这么设置的。

如果允许 GDB 自动设置语言，那么在进入文件名以'.mod'结尾的文件编译成代码时，工作语言会设置为 Modula-2。更多细节，参见 12.1.3 节[让 GDB 推断源代码语言]。

12.4.4.7 与标准 Modula-2 的差异

为了更容易调试 Modula-2 程序，做了一些修改。主要是放宽了类型限制。

• 与标准 Modula-2 不同，指针常量可以用整数来构成。这就允许在调试期修改指针变量。（在标准 Modul

a-2 里，指针变量的实际地址是不可见的；其地址只能用另外一个指针变量或者返回一个指针的表达式

来直接赋值去改变。）

• C 转义序列可用于字符串和字符来表示不可打印字符。GDB 用内嵌的转移序列来输出这些字符串。单个

不可打印字符用'CHR(nnn)'的形式输出。

• 赋值操作符(:=)返回右值参数的值。

• 所有内置的过程都可以修改和返回参数。

12.4.4.8 Modula-2 类型和域检查

警告：在此版本中，GDB 不执行类型和域检查。GDB 将两个 Modula-2 变量的类型视为相等，如果

• 用 TYPE t1 = t2 声明的变量，这两个变量的类型就是相等的。

• 在同一行中声明的变量。（注意：在 GNU Modual-2 编译器里是正确的，但在其他编译器里就未必。）

只要打开了类型检查，任何试图将不同类型的变量联合的操作都是错误的。

域检查用于所有的算术操作，赋值，数据索引边界和所有内置函数和过程。

12.4.4.9 范围操作符::和.

Modula-2 的范围操作符(.)和 GDB 范围操作符(::)有一些微妙的差异。这两个操作符都有相似的语法：

module . id

scope :: id

这里的 scope 是模块或过程的名字，module 是模块的名字，id 是除了另外模块的名字外，程序里定义的标识符。

使用::操作符可以让 GDB 在 scope 指定的范围里搜索标志符 id。如果在指定的范围里没有查找到，那么 GDB 会在包含字符串 scope 的所有范围里搜索。

使用.操作符可以放 GDB 在 module 指定的模块里搜索标志符 id。如果模块 module 定义里没有引进标志符 id，或者 id 不是 moudle 里定义的标志符，使用这个操作符将引发错误。

12.4.4.10 GDB 和 Modula-2

GDB 的某些命令很少应用于调试 Modula-2 程序。set print 和 show print 的 5 个子命令'vtbl' ,'demangle','asm-demangle','object','union'是专门为 C 和 C++设计的。前 4 个命令用于 C++，第 5 个用于 C union

类型，Modula-2 没有 union 相似的类型。

@操作符（参见 8.1 节[表达式]，75 页），虽然在所有语言里都可用，但对于 Modula-2 而言并不很有用。其设计意图是帮助调试动态数组，而 Modula-2 不能如同 C 和 C++那样创建动态数据。不过，由于可以用整型常量指定一个地址，指令'{type}adrexp'还是很用用的。

在 GDB 脚本里，Modula-2 不相等操作符#会转译为注释的开始。用<>替代#。

12.4.5 Ada

GDB 对 Ada 的扩展只支持 GNU Ada(GNAT)编译器产生的程序。其他 Ada 编译器目前尚未支持，并且要调试它们所产生的可执行程序很困难。

12.4.5.1 介绍

GDB 的扩展 Ada 模式支持相当大的 Ada 表达式语法子集。这个子集的设计背后的哲学是：

• GDB 应该提供基本的文字和为算法，取值，域选择，索引，子程序调用的操作访问，将更复杂的计算留给程序的子程序（子程序可供 GDB 调用）。

• Ada 语言的类型安全和限制对 GDB 用户不是特别重要。

• 简介对 GDB 用户很重要。

因此，要简介，调试器对于所有用户包好像隐含 with 和 use 子句，使得不必要用它们的包来完全限定大多数名字，而不用管上下文。在产生歧义的地方，GDB 要询问用户的目的。如同其它语言一样，如果程序中断于从 Ada 语言转换的程序上时，调试器会进入 Ada 模式。

在 Ada 模式里，可以用'--'来注释。这个功能在为命令文件做文档工作的时候最为有用。GDB 标准注释('#')在 Ada 模式里出现在行开头仍然有效，但在行的中部则不然（出于允许基准常量的缘故）。

调试器支持有限的重载。假设在某个程序点上，某个子程序有多个定义，在此处调用此子程序，调试器会使用实参的个数和参数的类型的某些信息来试图缩小子程序定义的子集。调试器使用非常有限的上下

文，在 call 命令上下文里倾向于使用过程而非函数，在别的地方可能是函数而非过程。

12.4.5.2 Ada 里的遗漏

下面是 Ada 里显著的遗漏子集：

• 只支持属性的子集：

-在数组对象里（非类型和子类型），支持'First,'Last,和'Length

-'Min 和'Max

-'Pos 和'Val

-'Tag

-在数组对象里（非子类型）支持'Range，但只在成员操作符(in)的右操作数上有效

-'Access，'Unchecked_Access，和'Unrestricted_Access（GNAT 扩展）

-'Address

• 不支持 Characters.Latin_1 字符集且字符串连接操作没有实现。因此，字符串里的转移字符目前不可用

• 数组相等测试（'='和'/='）也可以测试表达式的位相等。通常他们在以对元素是整型和枚举类型的数组上正确测试。但对数组成员的类型是用户自定义的则有可能不能正确运行，而在实数类型的数字（特别的，IEEE 标准的浮点数，因为负 0 和 NaNs），和含有未使用的位的不确定值的数组成员的数组上也可能不能正确测试。

• 其余的组件对组件的数据操作（and,or,xor,not 和关系测试）没有实现。

• 对数组和记录的总计的有限支持。只允许在右侧的赋值数上，如下所示：

```
set An_Array := (1, 2, 3, 4, 5, 6)
set An_Array := (1, others => 0)
set An_Array := (0|4 => 1, 1..3 => 2, 5 => 6)
set A_2D_Array := ((1, 2, 3), (4, 5, 6), (7, 8, 9))
set A_Record := (1, "Peter", True);
set A_Record := (Name => "Peter", Id => 1, Alive => True)
```

用总计来复制给判别式来改变其值可能导致未定义的后果，如果此判别式用于记录的话。不过，可以通过直接赋值给判别式来首先改变判别式值（在 Ada 里这个操作通常不允许），接着在执行总计赋值。例如，假设一个变量 A_Rec 声明其类型如下：

```
type Rec (Len : Small_Integer := 0) is record
  Id : Integer;
  Vals : IntArray (1 .. Len);
end record;
```

可以用两次赋值来将不同大小的 Vals 赋值：

```
set A_Rec.Len := 4
set A_Rec := (Id => 42, Vals => (1, 2, 3, 4))
```

12.4.5.3 对 Ada 的扩展

如同对其他语言那样，GDB 也对 Ada 提供了一些通用扩展（参见节 8.1[表达式]，75 页）：

• 如果表达式 E 是驻于内存的变量（典型地是一个本地变量或者数组成员）且 N 是一个正整数，那么 E@N 现实 E 的值和 E 后面相邻的 N-1 个数组成员。在 Ada 里，此操作通常不需要，因为其主要用途是显示数组部分，在 Ada 里通常用切割（slicing）实现此目的。不过，在调试某些调试信息被优化掉的程序时，还是偶尔有些使用的。

• B::var 表示“在函数或文件 B 里的名为 var 的变量”。如果 B 是一个文件名，通常必须用单引号将其括起来。

• 表达式{type} addr 表示“在地址 addr 上的类型为 type 的变量”

• 以'$'开头的名字是惯用变量（参见节 8.9[惯用变量]，89 页）或者是其寄存器（参见节 8.10[寄存器]，90 页）。

另外，GDB 提供了一些和 Ada 相关的其他的快捷方式和直接的附加方式：

• 赋值语句可以是一个表达式，返回其右边的操作数作为其值。因此，可以输入

```
set x := y + 3
print A(tmp := y + 1)
```

• 可以使用分号作为操作符，返回右边操作数的值。例如，可以允许如下的复杂条件断点：

```
break f
condition 1 (report(i); k += 1; A(k) > 100)
```

• 与其使用字符串连接和符号字符名称来将某些特殊的字符引入字符串，用户可以用特殊括号标注来替代，这个方法也用于打印字符串。假设有一些字符序列在字符串里或字符里，其形如"XX"，那么代表这个字符是以 XX 为编码的十六进制数值。在字符串中，字符序列'[""]'也代表单个引号。例如，

"One line.["0a"]Next line.["0a"]"

• 由于属性'Pos，'Min，'Max 的子类型是可选的（且在任何情况下都将被忽略）。例如，可以输入如下形式 rint 'max(x, y)

• 打印数组时，如果数组的下界是 1，GDB 使用位置数值记数法， 否则使用修改过的技术命名。例如，一个具有 3 个整数的一维数组的下界是 3 的话，将打印如下

(3 => 10, 17, 1)

也就是说，和标准的 Ada 不一样，只有第一个元素具有=>前缀。

• 可以用属性的独特的表达方式或者是其名字的子序列的多个字符（精确的匹配优先）来缩写属性。例如，可以用 a'len，a'gth，或者 a'lh 来替代 a'length。

• 由于 Ada 是不区分大小写的，通常调试器将用户输入的标志符映射为小写。GNAT 编译器为其内部标志符使用大写字符，通常这根用户无关。对于少数用户需要检查的，用半角括号围起来避免小写映射。例如，gdb print <JMPBUF_SAVE>[0]

• 打印一个全类范围类型的对象时和对一个访问到全类范围的变量取值时，会打印此对象相关类型的所有组件（如其运行时标签所示）。类似的，在此对象上的组件选择需要操作其的相关类型。

12.4.5.4 在开头处停止

某些时候有必要在阐述代码期和在主过程前调试代码。Ada 参考手册里定义次阐述代码由名为 adainit 的过程调用。

要在阐述代码前中断执行，只需要简单的使用两个命令：tbreak adainit 和 run。

12.4.5.5

除了前述的遗漏外（参见节 12.4.6.2[Ada 的遗漏]，138 页），在 GDB Ada 模式中有几个缺陷和限制，其中的一些会在今后计划发行的调试器发行版本和 GNU Ada 编译器里修复。

• 目前，调试器不能提供足够的信息来判断某些指针是指向对象的指针还是指针就是对象本身。因此，要打印一个表达式，用户需要在此表达式后边附加额外的.all。

• 编译器选择不实例化为对象的静态常量对于调试器是不可见的。

• 函数参数列表里的命名参数联合将被忽略（参数列表是位置相关的）。

• 目前很多有用的库文件包对于调试器不可见。

• 定点算术运算，转换，输入和输出使用浮点算术运算，并将得到和主机相近的计算结果。

• 属性'Address 可能不是 System.Address。

• GNAT 编译器从不为 Ada 语言定义的标准符号产生标准前缀。GDB 明白：它会在你使用的时候将此前缀从名字里删除，且不会在本地符号里查找其名字，也不会再其他包和子程序里去匹配符号的。如果在程序的任意地方已定义了非参数和本地变量的符号，这些符号的名字匹配在“标准（Standard）”里的话，GNAT 缺少鉴定的条件而可能导致混淆。如果混淆发生了，可以将用 Standard 包里的名字来明确地鉴定这些有问题的名字。

12.5 未支持的语言

除了完全支持的编程语言，GDB 还支持称为 minimal 的伪语言。minimal 不是一种真正的编程语言，提供接近 C 和汇编语言所能提供的功能子集。在调试 GDB 尚未支持的语言编写的程序时，应该允许执行大多数简单的操作。

如果将语言设置为 auto，当前架构要是未支持的语言，GDB 会自动选择语言

第13章 第十三章 查看符号表

本章介绍的命令可以查询在程序里定义的符号（变量名，函数名，类型名）。这些信息位于程序文本段，不会在程序执行时改变的。GDB 在程序符号表里查找这些符号，或者启动 GDB 时指示的文件（参见节 2.1.1[选择文件]，12 页），或者由文件管理命令指定的文件里查找（参见节 15.1[指定文件的命令]，155 页）。

有时可能需要引用含有奇怪字符的符号，通常 GDB 将这些符号视为字定义符。最常见的例子是引用到其他源文件里定义的静态变量（参见节 8.2[程序变量]，76 页）。对象文件会记录文件名作为调试符号，不过 GDB 可能解析文件名为 3 个部分，例如'foo.c'，会分解为'foo'，'.'，'c'三个字。要让 GDB 将'foo.c'视为单个符号，只要将其用单引号括起来；例如：

p 'foo.c'::x

在文件'foo.c'里查找 x 的值。

set case-sensitive on

set case-sensitive off

set case-sensitive auto

通常，GDB 在查找符号时，会用当前源代码语言决定的大小写相关与否来匹配它们的名字。偶尔用户也可能希望能自由选择。命令 set case-sensitive 用 on 参数指定大小写相关匹配，off 参数指定大小写无关匹配。如果指定了 auto，大小写相关将由此源代码语言缺省的设置决定。除了 Fortran，对于大部分语言缺省的都是大小写相关的，Fortran 是大小写无关的匹配。

show case-sensitive

此命令显示当前符号查找的大小写相关性的设置。

info address symbol

显示符号 symbol 数据在何处存储。对于寄存器变量，是说此变量在哪个寄存器里存储。对于非寄存器的本地变量，此命令打印此变量对于堆栈帧的偏移。

注意'print &symbol'，不是在所有寄存器变量上都有效，且对于栈本地变量而言，会打印此变量的当前实例的实际地址。

info symbol addr

打印存储于地址 addr 上的符号名。如果在此地址上没有存储符号，GDB 打印最近的符号和偏移量：

(gdb) info symbol 0x54320

_initialize_vx + 396 in section .text

这是 info address 命令的输出。可以用它来查找变量名或者用地址来查找函数名。

whatis [arg]

打印 arg 的数据类型，arg 可以是表达式或数据类型。如果不带参数，打印$的数据类型，值历史里最近的值。如果 arg 是表达式，不会真的计算，且此表达式里的任何边际效应（例如赋值或函数调用）操作都不会执行。如果 arg 是类型名，可能是一个类型的名字或者 typedef，或者对于 C 代码而言，可以有'class class-name'，'struct struct-tag'，'union union-tag'或者'enum enum-tag'形式.参见节 8.1[表达式]，81 页。

例如，对于下面变量声明：

struct complex {double real; double imag;} v;

这两个命令会输出如下：

(gdb) whatis v

type = struct complex

(gdb) ptype v

type = struct complex {

double real;

double imag;

}

如同 whatis 一样，使用 ptype 但不带参数的话会打印$的类型，值历史里最近的值。

有时候，程序使用模糊的数据类型或者不完整声明的复杂数据类型。如果程序里调试信息不足于让 GDB 显示数据类型的完整声明，GDB 会输出'<incomplete type>'。例如，假设下面声明：

struct foo;

struct foo *fooptr;

但没有 struct foo 自身的定义，GDB 会输出：

(gdb) ptype foo

$1 = <incomplete type>

“Incomplete type”是 C 描述没有完全声明的数据类型的术语。

info types regexp

info types

打印所有名字匹配正则表达式 regexp（如果没有参数的话，打印程序里所有的类型）的的类型的简短介绍。会匹配每个完整类型名，如同这个命令是一个个单独完整的命令行；因此，'i type value'显示程序里所有名字中包含字符串 value 的类型的信息，但'i type ^value$'显示那些完整名字是 value 的类型的信息。

本命令有两个方面和 ptype 不同：首先，和 whatis 类似，它不会打印详细描述；第二，它列出所有定义了此类型的源代码文件。

info scope location

列出在指定范围内的所有变量。此命令那个接受位置参数-函数名，源代码行，前面一个'*'的地址，且打印所有此范围内的变量。（更多关于支持位置形式的细节，参见节 7.2[指定位置]，74 页。）例如：

(gdb) info scope command line handler

Scope for command_line_handler:

Symbol rl is an argument at stack/frame offset 8, length 4.

Symbol linebuffer is in static storage at address 0x150a18, length 4.

Symbol linelength is in static storage at address 0x150a1c, length 4.

Symbol p is a local variable in register $esi, length 4.

Symbol p1 is a local variable in register $ebx, length 4.

Symbol nline is a local variable in register $edx, length 4.

Symbol repeat is a local variable at frame offset -8, length 4.

此命令在跟踪实践中判断要收集何种数据的时候特别有用，参见节 10.1.4[跟踪点操作]，115 页。

info source

显示当前源文件的信息--就是说，包含当前函数执行点的源文件：

- 源文件名，包含此文件的目录，
- 编译此文件的目录，
- 文件长度，以行为计量单位，
- 何种编程语言编写的，
- 可执行程序中是否包含有此文件的调试信息，如果有的话，此信息是何种格式（例如，STABS,Dwarf 2），且
- 调试信息中是否包含预处理宏。

info sources

打印程序中所有有调试信息的源文件名，用两个列表打印：一个是其符号已经读过的，一个是在需要

是将被度的。

info functions

打印所有已定义函数的名字和数据类型。

info functions regexp

打印所有名字包含匹配正则表达式 regexp 的函数的名字和数据类型。因此，'info fun step'查找所有名字中包含 step 的函数；'info fun ^step'查找名字以 step 开头的函数。如果函数名包含的字符和正则表达式语言冲突的话（例如'operator*()'）,可以用反斜线符号括起来解决此问题。

info variables

打印所有声明于函数外的变量的名字和数据类型（例如，除了本地变量）。

info variables regexp

打印所有名字包含匹配正则表达式 regexp 的变量的名字和数据类型（除了本地变量）。

info classes

info classes regexp

打印程序里所有 Objective-C 的类，或者（带参数 regexp）所有匹配正则表达式 regexp 的类。

info selectors

info selectors regexp

显示程序里所有 Objective-C 选择器，或者（有正则表达式参数的话）所有匹配某个正则表达式的选择器。

有些系统允许替换程序里单独的目标文件，而不需要中断和重启程序。例如，在 VxWorks 里，可以简单地重新编译一个有缺陷的目标文件，且同时保持程序运行。如果运行于这样的系统的话，可以让 GDB 重新加载这些自动再链接

的模块的符号：

set symbol-reloading on

在重新检视某个目标文件里的符号时，将此文件里符号定义用此替换。

set symbol-reloading off

再次检视目标文件里的相同名称的符号时，不替换其定义。缺省状态时不替换；如果在不允许自动模块链接的系统里运行的话，应该让 symbol-reloading 保持 off 状态，否则 DB 可能在链接大程序的时候丢弃符号；大程序可能包含几个具有相同符号的模块（在不同的目录或库里）。

show symbol-reloading

显示当前 on 或者 off 设置状态。

set opaque-type-resolution on

设置让 GDB 解决模糊类型问题。模糊类型是指声明为一个指向 struct，class，或者 union 的指针类型---例如,struct MyType*--用于一个源文件，而 struct MyType 的完全声明在另一个文件里。缺省是 on。

此子命令的设置改变会在下一次符号文件加载时生效。

set opaque-type-resolution off

设置让 GDB 不解决模糊类型问题。在这种情况下，打印这种数据类型如下：

{<no data fields>}

show opaque-type-resolution

显示是否解决模糊类型。

maint print symbols filename

maint print psymbols filename

maint print msymbols filename

将调试符号数据转储到文件 filename。这几个命令能够用于调试 GDB 读符号的代码。只包括调试数据的符号。如果使用'maint print symbols'，GDB 会包含那些它已经得到完整信息的符号：也就是说，filename 反应的是那些 GDB 已经读入符号的文件。可以用命令 Info sources 查找文件在哪里。如果用'maint print psymbols'的话，转储的文件显示 GDB 只部分了解的符号的信息。--也就是，符号定义于 GDB 略过的文件中，GDB 还没有完全读入。最后，对于目标文件需求的符号信息，'maint print msymbols'只从 GDB 已经读入的文件中转储最少的符号信息。关于 GDB 如何读取符号的讨论在（在 symbolfile 介绍中），参见节 15.1[指定文件的命令]，163 页。

maint info symtabs [regexp]

maint info psymtabs [regexp]

列出名字匹配 regexp 的 struct symtab 和 struct partial_symtab 的结构体。如果不带 regexp 的话，列出所有的结构体。输出包括可以拷贝进 GDB 里的表达式，调试这些表达式可以更详细的查看特定结构体。例如：

(gdb) maint info psymtabs dwarf2read

{ objfile /home/gnu/build/gdb/gdb

((struct objfile *) 0x82e69d0)

{ psymtab /home/gnu/src/gdb/dwarf2read.c

((struct partial_symtab *) 0x8474b10)

readin no

fullname (null)

text addresses 0x814d3c8 -- 0x8158074

globals (* (struct partial_symbol **) 0x8507a08 @ 9)

statics (* (struct partial_symbol **) 0x40e95b78 @ 2882)

dependencies (none)

}

}

(gdb) maint info symtabs

(gdb)

可以看到，有一个部分符号表，其名称包含字符串'dwarf2read',此表属于'gdb'可执行程序；并且，可以看到 GDB 根本没有读入任何符号表。如果在一个函数里设置断点，那么会让 GDB 读入包含此函数的编译单元的符号表：

(gdb) break dwarf2_psymtab_to_symtab

Breakpoint 1 at 0x814e5da: file /home/gnu/src/gdb/dwarf2read.c,

line 1574.

(gdb) maint info symtabs

{ objfile /home/gnu/build/gdb/gdb

((struct objfile *) 0x82e69d0)

{ symtab /home/gnu/src/gdb/dwarf2read.c

((struct symtab *) 0x86c1f38)

dirname (null)

fullname (null)

blockvector ((struct blockvector *) 0x86c1bd0) (primary)

linetable ((struct linetable *) 0x8370fa0)

debugformat DWARF 2

}

```
}
(gdb)
```

第14章 改变执行

如果认为在程序中发现了错误，也许你想知道，纠正当前的错误是否能让剩余的运算得到正确的结果。使用 GDB 的执行中改变的功能，通过实践就可以找到答案。

例如，将新值存入变量或内存地址，给程序一个信号，在不同的地址上重新执行程序，甚至从函数提前返回。

14.1 给变量赋值

通过赋值表达式可以改变变量的值。参见节 8.1[表达式]，75 页。例如：

print x=4

将 4 赋值给变量 x，接着打印这个赋值表达式的值（此值是 4）。更多关于已支持语言的操作符信息，参见第 12 章[用 GDB 调试不同语言编写的程序]。

如果对赋值操作的结果不感兴趣的话，使用 set 命令替代 print 命令就可以了。set 和 print 一样赋值，只是不打印赋值的结果，不将此值加入到值历史里（参见节 8.8[值历史]，88 页）。此表达式只用于其作用效果中。

如果 set 命令的参数字符串的开头和 set 子命令一样的话，使用 set variable 命令，而不要只用 set 命令。此命令和 set 命令一样，除了它没有子命令外。例如，如果程序有个变量叫 width，如果试图用'set width=13'来设置一个新值的的话，就会导致一个错误，因为 GDB 有个命令叫'set width':

(gdb) whatis width

type = double

(gdb) p width

$4 = 13

(gdb) set width=47

Invalid syntax in expression.

当然，错误在于表达式'=47'。要想设置程序变量 width，用

(gdb) set var width=47

由于 set 命令有许多子命令，很容易就和程序变量名冲突，用 set variable 命令替代 set 命令就能很好

的解决这个问题。例如，如果程序有个变量叫 g，由于 GDB 有个命令叫 set gnutarget，缩写为 set g：

```
(gdb) whatis g
type = double
(gdb) p g
$1 = 1
(gdb) set g=4
(gdb) p g
$2 = 1
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/smith/cc_progs/a.out
"/home/smith/cc_progs/a.out": can't open to read symbols:
Invalid bfd target.
(gdb) show g
The current BFD target is "=4".
```

在赋值时，GDB 允许比 C 更多的隐式转换；你可以方便地将一个整数存进一个指针变量，反过来也是一样地，并且可以具有相同长度的结构体转换到其它结构体，或者将其转换到长度短一些的结构体。

要将值存进内存中的绝对地址，使用'{...}'指令在指定的地址上存入指定类型的值（参见节 8.1[表达式]，81 页）。例如，{int}0x83040 把内存地址 0x8340 作为一个整型（这就意味着其在内存中有某种固定的大小和表示方式），且 set {int}0x83040 = 4 将值 4 存入此内存位置。

14.2 在不同的位置上继续执行

通常，继续执行程序时，用 continue 命令中断的地方上继续执行。不过，用下列命令，可以在自己选定的位置上继续执行程序：

jump linespec

jump location

在给定的行 linespec 或者 location 上继续执行。如果那个位置上有断点的话，执行会立即中断。关于

linespec 和 location 的不同形式，参见 7.2 节[指定位置]，74 页。jump 命令惯常会和 tbreak 命令联合使用。参见 5.1.1 节[设置断点]，43 页。

jump 命令除了改变程序计数器之外，不会改变当前堆栈帧，堆栈指针，也不改变任何内存位置和任何寄存器。如果行 linespec 是在当前执行的函数之外的函数里，如果这两个函数的参数模式或者本地变量不一样的话，那么结果就可能很诡异。由于这个原因，如果指定的行不在当前执行的函数里的话，jump 命令需要用户确认。不过，如果用户很熟悉程序的机器语言的话，这个诡异的结构也是可以预测的。

在很多系统里，把一个新的值存入寄存器$pc 里和 jump 命令的效果一样。不同之处在于，这样做不会让程序开始执行；这样做只会改变程序继续执行的地址。例如，

set $pc = 0x485

设置下一个 continue 命令或者单步执行命令的执行地址为 0x485，而不是在程序中断处继续执行。参见 5.2 节[继续和单步执行]，54 页。

最常使用 jump 命令的场景在于回退已经执行过的程序--中间可能有几个断点，可以更详细的检查程序的执行。

14.3 为程序设置信号

signal signal

在中断处立即继续执行程序，但立即给程序一个信号 signal。signal 可以是信号名字或信号的编号。例如，在很多系统里，signal 2 和 signal SIGINT 都可以设置中断信号。

另外，如果 signal 是 0，不设置信号继续执行。在程序中断于一个信号之后，通常会在用 continue 命令继续执行程序时会看到此信号；'signal 0'设置继续执行而不会遇到信号。、

再次按下回车键后，不会再执行 signal 命令。

引用 signal 命令和在 shell 里引用 kill 工具不一样。kill 设置信号的话，由信号处理表决定，GDB 选择如何处理此信号（参见 5.3 节[信号]，57 页）。signal 命令将信号直接传递给被调试程序。

14.4 从函数里返回

return

return expression

用 return 命令可以取消函数调用的执行。如果指定一个 expression 表达式参数，其值将是此函数的返回值。

要是运用 return，GDB 将销毁当前选定的堆栈帧（包括此栈内的所有帧）。可以认为是从销毁的堆栈帧中永久返回。如果希望设定一个返回值，指定 return 的参数。

return 命令弹出选定的堆栈帧（参见 6.3 节[选择堆栈帧]，70 页），和在此堆栈帧里所有的其他帧，使得其调用函数为最内层的堆栈帧。这个调用函数将变成为当前选定的堆栈帧。设定的值会存储于寄存器，用作函数的返回值。

return 命令不会继续执行程序；它会让程序中断地像函数刚刚返回的状态那样。与此相反，finish 命令（参见 5.2 节[继续和单步执行]，56 页）会继续执行直到此堆栈帧自然返回。

14.5 调用程序函数

print expr

计算表达式 expr 和打印结果。expr 可以包含调用被调试程序里的函数。

call expr

计算表达式 expr 而不打印空返回值。

call 是 print 命令的变体，使用它可以用来执行无返回值的函数（也称为 void 函数），但不会将 void 返回值打印，GDB 会负责打印。如果结果

不是 void，GDB 会打印且存入值历史。

可以为用 print 和 call 命令调用的函数产生一个信号（例如，如果此函数里有个 bug，或者给这个函数传递了错误的参数）。set unwindonsignal

命令决定在此情况下的行为。

set unwindonsignal

设置如果在 GDB 调用被调试程序里的函数的时候接收到一个信号时，堆栈退绕。如果设置为打开的话，GDB 会退绕到被创建的堆栈上，并且恢复到被调用前的上下文。如果设置为关闭（缺省的），GDB 会在信号接收到的地方上中断。

show unwindonsignal

显示当前堆栈退绕的设置。

有时候，你想调用的函数对于其他函数来说是个模糊的别名而已。在此情况下，GDB 不能得到类型

信息，包括函数参数的类型，这回导致 GDB 错误调用内部函数。结果是，被调用函数可能不正确工作，而且可能会导致崩溃。一个解决方案是，使用此别名函数的真名字。

14.6 为程序打补丁

缺省地，GDB 以只读方式打开包含程序可执行码的文件（或者 core 文件）。这样可以防止意外修改机器码；不过，这也组织了你主动对程序的二进制文件进行打补丁修改。

如果希望能对二进制文件进行补丁，可以用 set write 命令明确指定。例如，可能希望打开内部调试标志，或者紧急修改。

set write on

set write off

如果指定了'set write on'，GDB 以可读可写的方式打开可执行文件和 core 文件；如果指定了'set write off'（缺省方式），GDB 以只读方式打开。

如果已经加载了文件，要是让新的 set write 设置产生效果，必须在改变了设置之后，重新加载它（使用 exec-file 或者 core-file 命令）。

show write

显示可执行文件和 core 文件的读写的方式。

第15章 GDB 文件

GDB 需要了解被调试程序的文件名，来读取其符号表和开始执行程序。要调试一个 core dump，也必须告诉 GDB 此 core dump 文件的名字。

15.1 设置文件的命令

用户可能希望设置可执行程序和 core dump 文件名。通常可以在启动的时候，使用 GDB 启动命令的参数来设置（参见第二章[进入和离开 GDB],11 页）。

不过，在 GDB 调试期间偶尔也有可能需要改变文件。或者，在运行 GDB 的时候忘记设置需要用到的文件。或者，通过 gdbserver 调试远程目标（参见 17.3 节[使用 gdbserver 程序]，181 页）。在这些情况下，GDB 设置文件的命令就很有用了。

file filename

设置 filename 作为要调试的程序。GDB 从 filename 文件里读取符号和纯存储器的内容。此文件也是 run 命令用以执行的程序。如果

没有设置目录且在 GDB 工作目录里没有发现这个文件的话，GDB 使用环境变量 PATH 作为目录列表来搜索，就好象 shell 搜索程序来运行的那样。使用 path 命令，可以为 GDB 和程序改变这个变量的值。

可以用 file 命令将没有被链接的'.o'目标文件加载进 GDB。虽然不能“运行”目标文件，但可以反汇编和检查变量。而且，如果相关的 BFD 功能支持的话，可以用 gdb-write 来为目标文件打补丁。注意，在此用例中，GDB 不能转换湖州哦和修改相对位置，否则，分支和某些初始化变量就会出现在错误的地址上。这个功能有时还是很方便的。

file

不带参数的 file 命令将使得 GDB 丢弃从可执行文件和符号表里读取的所有信息。

exec-file [filename]

设置要执行的程序（但不是符号表）。如果要定位程序的话，GDB 会搜索环境变量 PATH。不带 filename 的话意味着丢弃可执行程序里的信息。

symbol-file [filename]

从 filename 里读取符号表信息。如果有必要，会搜索 PATH。使用 file 命令的话，将从同一个文件里取得符号表和可执行程序。

不带参数的 symbol-file 命令，将清除程序关于符号表的信息。

symbol-file 命令会引起 GDB 丢弃某些断点的内容和自动显示的表达式。这是由于它们可能包含指向内部数据的指针，这些内部数据记录这符号和数据类型的信息；这些数据是 GDB 里的旧符号表数据的一部分，执行此命令后会被丢弃。

在执行之后 symbol-file 命令之后，如果按下回车键的话，不会再重复执行 symbol-file 命令。

如果为某个特殊的环境而配置 GDB 的话，GDB 能够理解何种格式的调试信息是此环境的标准格式；既可以用 GNU 的编译器，也可以使用遵循本地习惯编译器；通常使用 GNU 编译器能够得到最好的效果；例如，使用 GCC 可以为优化后的代码产生调试信息。

对于大多数目标文件，除了使用 COFF 的 SVR3 系统外，symbol-file 命令通常不能立即读取符号表。相反，GDB 会快速扫描符号表来查找当前的原文件和符号。稍后会读取详细的信息，如果需要的话，每次读取一个源文件。

分两阶段读取的策略是为了加速 GDB 启动。对于大多数情况而言，在读取某个特定源文件的符号表的详细信息的时候，GDB 是非常快的，除了偶尔有一些停顿外。（set verbose 命令可以为这些停顿打印消息，如果设置了的话。参见 19.7 节[可选的警告和消息]，213 页。）

我们还没有为 COFF 实现两阶段策略。如果符号表用 COFF 格式存储，symbol-file 会立即读取符号表数据。注意，"stabs-in-COFF"仍然会以两阶段策略读取，因为调试信息实际上是以 stabs 格式存储的。

symbol-file filename [-readnow]

file filename [-readnow]

如果想让 GDB 真正读取完整的符号表信息的话，用'-readnow'选项取消 GDB 取符号表的两阶段读策略，此选项也适用于任何加载符号表信息的命令。

core-file [filename]

core

设置 core dump 文件，“内存映像”。通常，core 文件只包含产生这些 core 文件的进程的地址空间的某些内容；对于其他部分，GDB 可以访问可执行程序本身。

不带参数的 core-file 命令不设置 core 文件。

注意，程序还在 GDB 里运行的话，GDB 会忽略 core 文件。所以，如果你正在执行程序，而又想调试 core 的话，必须先结束当前正在运行的子进程。想这样做的话，使用 kill 命令（参见 4.8 节[结束子进程],

31 页）。

add-symbol-file filename address

add-symbol-file filename address [-readnow]

add-symbol-file filename -ssection address ...

add-symbol-file 命令从文件 filename 里读取附加符号表信息。如果文件 filename 是（用其他方式）动态加载进运行中的程序的话，可以使用此命令。address 是开始加载文件的内存地址；GDB 自身不能计算此地址。另外，指定段名和此段的基址的话，可以设置任意多的' ‘-ssection address'对。可以用任意表达式来设置 address。

文件 filename 的符号表会加入原来用 symbol-file 命令读入的符号表里。可以多次重复使用 add-symbol-file 命令；新读入的符号数据会加入到旧的符号表里。要销毁所有的旧符号数据，使用不带参数的 symbol-file 命令就可以了。

虽然 filename 通常是动态链接库文件，可执行文件，或者是其他要加载到进程里的，已经完全重定位过的目标文件，也可以从可重定位的'.o'文件爱你里加载符号信息，只要满足：

- 此文件的符号信息只引用定义于此文件的连接器符号，不包括定义于其他目标文件的符号，
- 此文件的符号信息指向的各段都已经完全载入，如同其在文件里一样，并且
- 可以决定各段数据加载的地址，并且将其作为 add-symbol-file 命令的参数。

某些嵌入式操作系统，例如 Sun Chorus 和 VxWorks，可以将可重定位的文件加载进已运行的程序里；这类操作系统很明显将需求弄得很难满足。不过，许多原系统使用交叉链接程序也会让 GDB 难于符合其需求，认识到这一点很重要（例如，.linkonce 段分解和 C++构造函数表组建）。总之，不能假定使用 add-symbol-file 读取的可重定位目标文件符号信息具有和链接进程序里的效果。

add-symbol-file 在使用后，按下回车键不会再重复执行。

add-symbol-file-from-memory address

从动态加载的目标文件里的指定地址 address 里加载符号，此文件已经映射进内存中。例如，Linux 内核已经将系统调用 DSO 映射进每个进程的地址空间；DSO 为某些系统调用提供了内核相关的代码。任意能够得到文件的共享目标文件头地址的表达式，都可为此命令作参数。要让此命令起效，必须先运行 symbol-file 或者 exec-file

命令。

add-shared-symbol-files library-file

assf library-file

add-shared-symbol-files 命令目前只能用于 MS-Windows 操作系统的 Cygwin 版的 GDB，其是 dll-symbos 命令的别名 （参见 18.15 节[Cygwin 原生]，187 页）。GDB 自动查找共享库，不过，如果 GDB 没有找到，你可以调用

add-shared-symbol-files。此命令需要一个参数：共享库的文件名。assf 是 add-shared-symbol-files 命令的简写别名。

section section addr

section 命令将可执行程序的名为 section 的段的基址改变到 addr。此命令可以用于可执行文件不包含段地址的情况下，（例如在 a.out 格式里），或者在已设置的地址是错误的情况下。各段必须单独修改。下面介绍的 info files 命令列出所有的段和它们的地址。

info files

info target

info files 和 info target 是同义命令；这两个命令都可以打印当前目标（参见第 16 章[设置调试目标]，167 页），包括可执行文件名和当前当前 GDB 使用的 core dump 文件，并打印当前的符号是从哪个文件里加载的。help target 命令可以列出除当前目标外，所有可能的目标。

maint info sections

另一个能查看程序段的额外信息的命令是 maint info sections。除了用 info files 命令显示的段信息之外，这个 sections 命令还能显示可执行程序和 core dump 文件里每个段的标记和文件偏移。另外，maint info sections 还提供如下命令选项（可以任意组合）：

ALLOBJ 显示所有已加载目标文件的段，包括共享库。

sections

只显示名为 sections 的段的信息。

section-flags

显示 section-flags 标记为 true 的段的信息。当前 GDB 能识别的段标记如下：

ALLOC 加载后在程序里会分配空间的段。出包含调试信息的段外，会对其余所有的段设置。

LOAD 从文件中加载进子进程存储器里的段。对初始化前代码和数据设置，将.bss 段清除。

RELOC 在加载前需要重新定位的段。

READONLY

子进程不能修改的段。

CODE 只包含可执行代码的段。

DATA 只包含数据的段（无可执行代码）。

ROM 位于 ROM 里的代码。

CONSTRUCTOR

包含构造器/析构器数据的段。

HAS_CONTENTS

非空的段。

NEVER_LOAD

链接器的指令，设置不输出段。

COFF_SHARED_LIBRARY

告知链接器此段包含 COFF 共享库信息。

IS_COMMON

包含公共符号的段。

set trust-readonly-sections on

告知 GDB 目标文件里的只读段真的是只读的（例如，它们的内容不会改变）。那样的话，GDB 从目标文件里的段获取信息，而不是从目标程序。对于某些目标系统（典型的是嵌入式系统），这个设置可以显著的提高调试性能。缺省关闭。

set trust-readonly-sections off

告知 GDB 不要信任只读段。这个设置表明此段的内容在程序运行时可能发生改变，因此在需要的时候应该从系统里再次获取。

show trust-readonly-sections

显示当前只读段的信任设置。

所有文件设置相关的命令都能接受绝对\相对文件名作为参数。GDB 总是将文件名转换成绝对文件名，并以此记录之。

GDB 支持 GNU/Linux，MS-Windows，HP-UX，SunOS，SVr4，Irix 和 IBM RS/6000 AIX 共享库。

在 MS-Windows 系统里，GDB 必须链接 Expat 库，才能支持共享库。参见[Expat],325 页。

用 run 命令或者检验 core 文件的话，GDB 会自动从共享库里加载符号定义。（在执行 run 命令前，GDB 无法理解对共享库里函数的引用，除非是在调试一个 core 文件）。

在 HP-UX 系统里，如果程序显式加载一个库，GDB 在 shl_load 调用时自动加载符号。

不过，也有很多情况下用户也许不希望 GDB 自动从共享库里加载符号定义，例如符号定义特别多或者有很多符号定义的时候。要控制共享库符号自动加载的话，使用下列命令：

set auto-solib-add mode

如果 mode 是 on，程序开始执行，或者在动态连接器通知 GDB 加载了一个新的共享库的时候，共享库里的符号都将自动加载。如果 mode 是 off，必须用 sharedlibrary 命令手动加载符号。缺省值是 on。

如果程序使用了大量调试信息的共享库的话，而这些调试信息占用很多存储器，避免自动从共享库中加载符号将可以减少 GDB 内存占用。为此，在运行程序前输入 set auto-solib-add off,接着用 sharedlibrary regexp 分别加载确实需要调试信息的共享库，这里 regexp 匹配共享库的正则表达式。

show auto-solib-add

显示当前自动加载模式。

要显式加载共享库符号，使用 sharedlibrary 命令：

info share

info sharedlibrary

打印已经加载的共享库的名字。

sharedlibrary regex

share regex

加载名字匹配 Unix 正则表达式 regex 的共享目标库的符号。由于会自动加载，此命令只会在程序需要加载共享库的时候，或者输入了 run 命令之后执行。如果不带 regex 参数，会加载程序需要的所有共享库的符号。

nosharedlibrary

卸载所有共享目标库的符号。此命令将销毁所有已加载的共享库的符号。不过，用户明确加载的共享库符号不会销毁。

在发生共享库事件的时候，用户可能希望将 GDB 中断并将控制权交给用户。要达到此目的，使用 set stop-on-solib-events 命令。

set stop-on-solib-events

此命令决定在收到动态连接器通知共享库事件时，GDB 是否把控制权交给用户。通常，人们最关注的时间是加载和卸载共享库。

show stop-on-solib-events

显示，共享库事件出现时，GDB 是否中断且交出控制权。

在很多交叉和远程调试配置里也支持共享库。在本地宿主系统里必须有一份目标库的拷贝；这些拷贝应该和目标库一样，不过，目标系统上的拷贝可以删除掉调试信息，而本地系统上的可以不删除。

对于远程调试，要告知 GDB 目标库在何处，GDB 才能够正确加载拷贝--否则，GDB 可能加载本地宿主系统里的库。GDB 有两个变量可以设置目标库的搜索路径。

set sysroot path

设置 path 作为要调试的程序的系统根目录。所有绝对共享库的路径都将以 path 为先导路径；很多运行时加载器将共享库的绝对路径存储在目标程序里的存储器里。如果使用 set sysroot 查找共享库，共享库应该和目标系统那样存储，例如，path 下的'/lib'和'/usr/lib'结构。

set solib-absolute-prefix 命令是 set sysroot 的别名。

用配置'--with-sysroot'选项设置默认系统根目录。如果 GDB 配置了 prefix（用'--prefix'或者'--exec-prefix'设置）系统根目录，那么默认系统根目录会在 GDB 移到另外一个目录下的时候自动更新。

show sysroot

显示当前共享库的前缀（prefix）。

set solib-search-path path

path 应该以分号分隔，如果设置了此变量，将用以搜索共享库。要是'sysroot'不能找到共享库，或者是共享库的相对路径的话，用'solib-search-path'再搜索共享库。要只用'solib-search-path'，不使用'sysroot'的话，一定要将'sysroot'设置为一个不存在的目录，防止 GDB 查找本地宿主系统上的库。'sysroot'优先使用；将其设为不存在的目录可以防止自动加载共享库符号。

show solib-search-path

显示当前共享库搜索路径。

15.2 调试信息位于不同文件中

GDB 支持用户将程序调试信息放在一个独立的文件里，不必要在可执行文件中，GDB 可以某种方式来查找和自动加载调试信息。由于调试信息可能非常大--有时可能比可执行代码自身还要大--某些系统将其可执行程序的调试信息以单独的文件发布，在需要调试问题的时候，用户可以再安装这些文件。

GDB 支持两种设置单独调试信息文件的方式：

• 可执行程序里包含了一个调试链接，此链接指定了单独的调试信息文件名。单独调试文件名通常是'executable.debug'，

executable 是相应的可执行程序名，不带路径（例如，'ls.debug'是'/usr/bin/ls'的调试信息文件）。此外，调试链接为调试文件设置了 CRC32 的校验和，GDB 用此校验和来确保可执行文件和调试文件是同一个版本的。

• 可执行文件包含一个版本 ID 号和一个唯一的 bit 串，而相应的调试信息文件也包含此 bit 串。（此方式只在某些系统上支持，特别是那些在二进制文件里使用 ELF 格式和 GNU Binutils 的系统。）更多关于此功能的细节，参见'--build-id'命令行选项的介绍，在 GNU 连接器的“命令行选项”节中。虽然版本 ID 号没有执行指出调试信息文件名，但是可以从版本 ID 号里计算出来，参见下面。

由于有两种方法可以设置调试信息文件，GDB 也用两种不同的方式查找调试信息文件：

• 对于“调试链接”方式，GDB 在可执行文件的目录里查找对应名字的文件，接着在此目录下的子目录'.debug'下查找，最后在全局调试目录下的一个子目录里查找，此子目录的名字和可执行文件的绝对文件名的先导目录名相同。

• 对于“版本 ID”方式，GDB 在全局调试目录下的'.build-id'子目录下查找名为'nn/nnnnnnnn.debug'的文件，这里 nn 是版本 ID 字符 串的头两个 16 进制字符，nnnnnnnn 是余下的字符。（真正的版本 ID 字符串是 32 个或更多的 16 进制字符，不是 10 个。）

因此，举个例子，假设你要调试'/usr/bin/ls',此程序有个调试链接指定了调试文件'ls.debug'，且其版本 ID 是是 16 进制的 abcdef1234。如果全局调试目录是'/usr/lib/debug'，那么 GDB 按顺序查找下列调试信息文件：

--‘/usr/lib/debug/.build-id/ab/cdef1234.debug’

--‘/usr/bin/ls.debug’

--‘/usr/bin/.debug/ls.debug’

--‘/usr/lib/debug/usr/bin/ls.debug’.

你可以设置全局调试信息目录的名称，并查看当前 GDB 所使用的名称。

set debug-file-directory directory

将 directory 设置为 GDB 搜索单独调试信息文件的目录。

show debug-file-directory

显示搜索单独调试信息文件的目录。

调试链接是可执行文件的一个特殊段，称为.gnu_debuglink。这个段必须包含：

• 一个文件名，不带任何目录，以 0 为结尾（一个字节，其值是 0），

• 0 到 3 字节的填充，填充的 4 个 0 值字节作为一个边界----假若在这个段里还有下一个条目的话，且

• 4 字节 CRC 校验和，用可执行文件一样的编码格式存储。此校验和用下面给出的函数计算调试信息文件的全部内容得到，crc 的参数 0 为 0.任何可执行文件格式都可以带有调试链接，只要它能包含一个名为.gnu_debuglink 的段，且其内容如上所述。

版本 ID 是可执行文件（或者在其它 GDB 可以识别的 ELF 二进制文件）里的特殊段。此段常常称为.note.gnu.build-id，但名称不是强制性的。此 ID 包含编译文件的唯一标志--此 ID 也能包含同一版本树的多个版本的相同的部分。缺省的算法是 SHA1，能产生 160 bit（40 个 16 进制的字符）的版本 ID 字符串。带调试符号的原本二进制文件里，去除调试符号的版本里和单独的调试信息文件里，都有相同的段，并且段里的内容也是相同的。

调试信息文件本身应该是普通的可执行程序，包含链接器符号的全集，段和调试信息。调试信息文件的各段应该有和原文件相同的段名称，地址和大小，但不必要包含数据--很像普通可执行文件里的.bss 段。

GNU 二进制工具包（Binutils），有'objcopy'工具，用此工具可以产生单独的可执行文件/调试信息文件对，如下命令：

objcopy --only-keep-debug foo foo.debug

strip -g foo

这些命令将调试信息从可执行文件'foo'里移除，并将其存储于文件'foo.debug'。可以用第一种，第二种，或者两种方式来将两个文件链接起来：

• 调试链接方式需要下面额外的命令来将调试链接放进文件'foo'里：

objcopy --add-gnu-debuglink=foo.debug foo

Ulrich Drepper 的‘elfutils’包，从 0.53 版开始，带有一个特殊的版本的 strip，这个版本的 strip 的命令 strip foo -f foo.debug，具有和上面两个命令 objcopy 和 ln -s 一起用同样的功能。

•用 ld --build-id 和 GCC 对应的 gcc -Wl,--build-id 命令，可以将版本 ID 存储进主可执行文件。从 2.18 版开始，GNU 二进制工具包（Binutils）支持版本 ID 添加调试文件的兼容修复。

由于有很多方法可以计算调试链接的 CRC 值（多项式，反转，字节序等等），要介绍计算用于.gnu_debuglink 段的 CRC 的方法，给出完整的计算 CRC 的函数代码是最简单的：

```
unsigned long
gnu_debuglink_crc32 (unsigned long crc,
unsigned char *buf, size_t len)
{
static const unsigned long crc32_table[256] =
```

```
{
0x00000000, 0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419,
0x706af48f, 0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4,
0xe0d5e91e, 0x97d2d988, 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07,
0x90bf1d91, 0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de,
0x1adad47d, 0x6ddde4eb, 0xf4d4b551, 0x83d385c7, 0x136c9856,
0x646ba8c0, 0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9,
0xfa0f3d63, 0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4,
0xa2677172, 0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b,
0x35b5a8fa, 0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3,
0x45df5c75, 0xdcd60dcf, 0xabd13d59, 0x26d930ac, 0x51de003a,
0xc8d75180, 0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599,
0xb8bda50f, 0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924,
0x2f6f7c87, 0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190,
0x01db7106, 0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f,
0x9fbfe4a5, 0xe8b8d433, 0x7807c9a2, 0x0f00f934, 0x9609a88e,
0xe10e9818, 0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01,
0x6b6b51f4, 0x1c6c6162, 0x856530d8, 0xf262004e, 0x6c0695ed,
0x1b01a57b, 0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950,
0x8bbeb8ea, 0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3,
0xfbd44c65, 0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2,
0x4adfa541, 0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a,
0x346ed9fc, 0xad678846, 0xda60b8d0, 0x44042d73, 0x33031de5,
0xaa0a4c5f, 0xdd0d7cc9, 0x5005713c, 0x270241aa, 0xbe0b1010,
0xc90c2086, 0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f,
0x5edef90e, 0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17,
0x2eb40d81, 0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6,
0x03b6e20c, 0x74b1d29a, 0xead54739, 0x9dd277af, 0x04db2615,
0x73dc1683, 0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8,
0xe40ecf0b, 0x9309ff9d, 0x0a00ae27, 0x7d079eb1, 0xf00f9344,
0x8708a3d2, 0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb,
0x196c3671, 0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a,
0x67dd4acc, 0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5,
0xd6d6a3e8, 0xa1d1937e, 0x38d8c2c4, 0x4fdff252, 0xd1bb67f1,
0xa6bc5767, 0x3fb506dd, 0x48b2364b, 0xd80d2bda, 0xaf0a1b4c,
0x36034af6, 0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef,
0x4669be79, 0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236,
0xcc0c7795, 0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe,
0xb2bd0b28, 0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31,
0x2cd99e8b, 0x5bdeae1d, 0x9b64c2b0, 0xec63f226, 0x756aa39c,
0x026d930a, 0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713,
0x95bf4a82, 0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x92d28e9b,
0xe5d5be0d, 0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242,
0x68ddb3f8, 0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1,
```

```
0x18b74777, 0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c,
0x8f659eff, 0xf862ae69, 0x616bffd3, 0x166ccf45, 0xa00ae278,
0xd70dd2ee, 0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7,
0x4969474d, 0x3e6e77db, 0xaed16a4a, 0xd9d65adc, 0x40df0b66,
0x37d83bf0, 0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9,
0xbdbdf21c, 0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605,
0xcdd70693, 0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8,
0x5d681b02, 0x2a6f2b94, 0xb40bbe37, 0xc30c8ea1, 0x5a05df1b,
0x2d02ef8d
};
unsigned char *end;
crc = ~crc & 0xffffffff;
for (end = buf + len; buf < end; ++buf)
crc = crc32_table[(crc ^ *buf) & 0xff] ^ (crc >> 8);
return ~crc & 0xffffffff;
}
```

这个计算不用于"版本 ID"方式。

15.3 读取符号文件的错误

在读取符号温家时，GDB 偶尔会碰到问题，例如不能识别符号类型，或者已知的编译器输出的 bug。缺省的，GDB 不会告知用户这类问题，因为这些问题相对来说只会对调试编译器的人才会感兴趣的。如果你有兴趣查看关于符号表的建构错误的信息，可以让 GDB 每种问题只打印一个消息；要查看问题发生过多少次，用命令 set complaints（参见 19.7 节[可选的警告和消息]，221 页）。

当前能打印的消息和其意义，包括：

inner block not inside outer block in symbol

此符号信息显示符号范围的起止（例如，函数或者一块声明的开始地址）。这个错误表明，外部范围块没有完全包含内部范围块。

GDB 会避开这个问题，将此内部范围块认为和外部块具有一样的范围。在此错误消息中，如果外部块不是函数的话，symbol 可能显示为"(don't know)"。

block at address out of order

符号范围块的信息应该以增序排列。这个错误表明，符号信息没有按增序排列。

GDB 不会避开这个错误，在读源文件的符号时，定位此文件的符号会发生问题。（设置 set verbose on 可以知道哪些源文件有这个问题。参见 19.7 节[可选警告和消息]，221 页。）

bad block start address patched

符号范围块的信息，表明符号块的起始地址比先前读入的源代码行的地址小。此错误见于 SunOS 4.1.1（和更早版本）C 编译器。

GDB 会避开此错误，将符号范围块认为和先前读入的源代码行具有相同的地址。

bad string table offset in symbol n

编号为 n 的符号有一个指向字符串表的指针，此指针指向的地址超出字符串表的地址。

GDB 会避开此问题，认为此符号的名字是 foo，不过，如果有很多符号的名字也是 foo 结尾的话，可能会导致其它很多问题。

unknown symbol type 0xnn

符号信息包含 GDB 不知道如何读入的新数据类型。0xnn 是不能识别的符号类型，16 进制数。

GDB 忽略此信息，避开此类错误。即使某些符号不可用的时候，这也能让用户调试自己的程序。如果遇到了这个错误，又想调试这个它的话，可以调试 gdb 自身，complain 设置断点，接着执行到函数 read_dbx_symtab，examine *bufp 查看符号。

stub type has NULL name

GDB 没有找到结构体或类的完整定义。

const/volatile indicator missing (ok if using g++ v1.x), got...

C++成员函数缺失一些最新版本编译器应该输出的符号信息。

info mismatch between compiler and debugger

GDB 不能分析编译器产生的类型声明。

第16章 设置调试目标

目标是指被调试程序所拥有的执行环境。

通常，GDB 运行于被调试程序所在的相同宿主环境里；在此类环境里，file 或者 core 命令执行的同时，也顺带设置了调试目标。要是需要更灵活的话--例如，在物理上隔离的宿主上运行 GDB，或者通过串口控制单独的系统，甚或通过 TCP/IP 链接实时系统--可以使用 target

命令为 GDB 设置目标类型（参见 16.2 节[管理目标的命令]，176 页）。

GDB 可以编译为支持多种不同的目标架构。如果如此编译 GDB 的话，用 set architecture 命令，可以选择可用架构中的其一。

set architecture arch

此命令设置当前目标架构为 arch。除了支持的架构之外，arch 的值也可以是"auto"。

show architecture

显示当前目标架构。

set processor

processor

这两个命令分别是 set architecture 和 show architecture 的别名。

16.1 有效目标

有三类目标：进程，core 文件和可执行文件。GDB 可以同时工作于 3 个不同类型的有效目标。这意味着用户（举例来说）可以启动一个进程并监控其状态，并不需要放弃 core 文件的调试。

例如，如果执行'gdb a.out'，那么可执行文件 a.out 就是唯一的有效目标。如果同时指定了一个 core 文件--假定先前有过崩溃和转储的话--那么 GDB 就有个连个有效的目标并串联起来使用，首先从 core 文件里查找，然后再可执行文件里查找，得到存储器位置。（典型地，这两类目标是互补的，因为 core 文件只包含程序的读-写存储器--变量之类的--加上机器状态，而可执行文件只包含程序文本和初始化数据。）

在输入 run 命令后，可执行文件也就成为有效进程目标了。如果进程目标是有效的，所有需要存储器地址的 GDB 命令都指向此目标；进程目标起效之后，有效 core 文件和可执行文件里的地址就被隐藏了。

用 core-file 和 exec-file 命令可以设置新的 core 文件或可执行目标（参见 15.1 节[设置文件的命令]，

155 页）。要将一个已运行的进程设置为目标，使用 attach 命令（参见 4.7 节[调试已运行的进程]，30 页）。

16.2 管理目标的命令

target type parameters

连接 GDB 宿主环境到一个目标机器或进程。典型地，目标是与调试工具对话的协议。使用参数 type 来设置类型或者目标机器的协议。

目标协议解析参数 parameters，一般包括设备名、要连接的主机名，进程号和波特率。

执行 target 命令之后，再按下回车键的话，不会重复执行 target 命令。

help target

显示所有可用目标的名称。要显示当前设置的目标，可以用 info target，也可以用 info files（参见 15.1 节[设置文件的命令，155 页]）。

help target name

介绍指定的目标，包括设置它所需要的所有参数。

set gnutarget args

GDB 使用自有的库 BFD 来读取用户的文件。GDB 清楚是在读取可执行程序，还是 core 文件，或者是一个.o 文件;不过,可以用 set gnutarget 命令设置文件格式。和大多数 target 命令不同的是,用 gnutarget 的话，目标指向一个程序，而不是机器。

警告：要用 set gnutarget 设置文件格式，必须知道实际的 BFD 的名称。

参见爱你 15.1 节[设置文件的命令]，155 页。

show gnutarget

show gnutarget 命令可以显示为 gnutarget 设置的是何种文件格式。如果还未设置 gnutarget，GDB 会自行判断每个文件的文件格式，show gnutarget 显示"当前 BDF 目标"是"auto"。

下面是一些常用的目标（可用的或不可用，由 GDB 的配置决定）：

target exec program

可执行文件。'target exec program'和'exec-file program'相同。

target core filename

core dump 文件。'target core filename'和'core-file filename'相同。

target remote medium

通过串口线或网络连接到 GDB 的远程系统。此命令设置 GDB 用其自有的协议通过媒介 medium 来调试。

参见第 17 章[远程调试]，171 页。

例如，如果有个板子连接到机器上的'/dev/ttya'，可以看到：

target remote /dev/ttya

target remote 支持 load 命令。此功能只在有其它方式得到目标系统的代理是才有用，并且可以将其存储与存储器里，这样就不至于在下载时导致崩溃。

target sim

内置的 CPU 仿真器。GDB 包含有大多数架构的仿真器。一般来说，

target sim

load

run

是可以正常工作的；不过，不能假定某个确定的内存映射，设备驱动，或者基本的 I/O 是可用的，虽然某些仿真器确实提供了。更多处理器相关的仿真器细节，参见 18.3 节[嵌入处理器]的相关内容，194 页。

某些配置可能也包括下面的目标：

target nrom dev

NetROM ROM 枚举器。这个目标只支持下载。

GDB 不同的配置上具有不同的目标；你的配置有或多或少的目标。

一旦成功建立连接，很多远程目标需要用户下载可执行程序的代码。用户可能希望控制此过程的不同方面。

set hash

此命令决定在下载一个文件到远程监视器的时候，是否打印 hash 标记'#'。如果 on，每个 S-record 记录成功下

载到监视器之后，都会打印一个 hash 标记。

show hash

显示当前 hash 标记显示的状态。

set debug monitor

设置或者关闭显示 GDB 和远程监视器之间的通信消息的。

show debug monitor

打印当前是否显示 GDB 和远程监视器之间的通信消息。

load filename

load 命令是否可用，取决于 GDB 集成了那些远程调试工具。如果可用的话，就意味着在远程系统上文件 filename（可执行文件）在调试中是可用的--例如，通过下载，或者动态链接的方式。在 GDB 里，load 记录文件 filename 符号表，如同 add-symbol-file 命令一样。

如果你的 GDB 没有 load 命令，试图执行 load 命令的话，GDB 会打印错误消息"You can't do that when your target is ..."

文件加载的地址是由可执行程序指定。对于某些目标文件格式，你可以在链接程序的时候指定加载地址；对于其它格式，例如 a.out，目标文件格式指定的是固定地址。GDB 是否能够将程序加载到闪存里，是依赖于对端的功能的。

执行 load 之后，再次按下回车键的话，load 不会再次执行。

16.3 选择目标字节序

某些处理器，例如 MIPS，PowerPC 和 Renesas SH，在 big-endian（大端）和 little-endian（小端）字节序上都可以工作。

一般来说，可执行程序或符号有一个 bit 指示字节序，用户通常都不需要关心使用何种字节序。不过，有时手动改变处理器的字节序，可能很有用。

set endian big

设置 GDB 假定目标是大端序。

set endian little

设置 GDB 假定目标是小端序。

set endian auto

设置 GDB 用可执行程序的字节序。

show endian

显示 GDB 当前系统字节序。

注意，这些命令只调整宿主系统上对符号数据的转换，因此对于目标系统是毫无作用的。

第17章 调试远程程序

如果要调试运行于一个不能运行 GDB 的机器上的程序时，使用远程调试就很有帮助了。例如，可以用远程调试操作系统内核，或者在小型的，没有足够的操作系统能力来支持运行完全功能的调试器的系统里调试。

GDB 可以配置特殊串口和 TCP/IP 接口来支持远程调试特殊的调试目标。另外，GDB 还有通用串口协议（特指 GDB，不是指任何特殊目标系统），如果你可以用它实现远程代理--远程代理的代码运行于远程系统，用来和 GDB 通讯。

可能还有其它的远程目标在你的 GDB 配置中也可用；可用 help target 来查看。

17.1 连接到远程目标

在 GDB 的宿主系统里，需要一个被调试程序的未去除符号的拷贝，因为 GDB 需要符号和调试信息。如常启动 GDB，使用程序的本地拷贝的名字作为第一个参数。

GDB 可以通过串口线和目标通讯，或使用 TCP 和 UDP 在 IP 网络和目标连接。在所有的用例下，GDB 使用相同的协议来调试程序；只是传输调试包的媒介改变而已。target remote 命令建立一个到目标的连接。其参数指定使用何种传输媒介：

target remote serial-device

使用 serial-device 来和目标通讯。例如，使用串口线连接的名为"/dev/ttyb"的设备：

target remote /dev/ttyb

如果使用的是串口线，你可能想要设置 GDB 的'--baud'选项，或者在 target 命令之前执行 set remotebaud 命令（参见 17.4[远程配置]，184 页）。

target remote host:port

target remote tcp:host:port

使用 TCP 连接到主机端口 port 去调试。host 可以是主机名，也可以是一个数字 IP 地址；port 必须是十进制的数字。host 可以是目标机器本身--如果目标是直接联网的话，或者可以是有串口线连接到目标的终端服务器。

例如，要连接到名为 manyfarms 的终端服务器上的端口 2828：

target remote manyfarms:2828

如果远程目标真的运行于和你的调试会话相同的机器上（例如，运行于相同那个主机上的目标仿真器），你可以省略主机名。

例如，连接到本机上的端口 1234：

target remote :1234

注意，冒号仍然是必须的。

target remote udp:host:port

使用 UDP 包在主机的端口 port 上调试程序。例如，连接终端服务器 manyfarms 的 UDP 端口 2828：

target remote udp:manyfarms:2828

使用 UDP 连接来调试远程程序时，必须牢记'U'代表'不可靠'。由于繁忙或者不可靠的网络，UDP 可能无声无息的丢包，这也会

导致调试会话遭到破坏。

target remote | command

在后台执行 command 并用管道与其通讯。command 是 shell 命令，系统命令 shell(/bin/sh)会解析并展开此命令；它将远程协议包作为其标准输入，将其回复作为标准输出。可以用这个命令来运行能提供远程调试协议的单独的仿真器，如用 ssh 来建立网络连接，或者用此命令来提供其它相似的技巧。

如果 command 关闭其标准输出（可能是由于推出）时，GDB 会尝试发一个 SIGTERM 信号。（如果程序已经推出，此信号无效。）

一旦连接建立了，可以用所有常用的命令来检验和改变数据。远程程序已经执行；可以用 step 和 continue，且不必用 run。

GDB 在等待远程程序时，如果输入中断字符（通常是 Ctrl-c），GDB 会试图去中断此程序。中断可能也可能不成功，这部分地取决于硬件和远程系统使用的串口驱动。如果再次输入中断字符，GDB 打印如下提示：

Interrupted while waiting for the program.

Give up (and stop debugging it)? (y or n)

如果输入 y，GDB 会丢弃远程调试会话。（如果想在以后再次尝试的话，使用'target remote'可以再次连接。）如果输入 n，GDB 会接着等待。

detach

在完成调试远程程序时，可以用 detach 命令来将其从 GDB 中释放。程序从目标中分离时，通常会继

续执行，但实际上依赖于特定的远程代理。detach 命令之后，GDB 就可以连接另外的目标了。

disconnect

disconnect 命令的行为和 detach 类似，除了目标通常不继续执行。此命令会等待 GDB（此 GDB 实例或另外一个）来连接，然后继续调试。在 disconnect 命令之后，GDBB 就可以连接另外的目标了。

monitor cmd

此命令那个允许用户给远程监视器发送直接命令。由于 GDB 不关心命令它所发送的此类命令，此类命令也是扩展 GDB 的方法--可以添加只有外部监视器能识别和实现的新命令。

17.2 给远程系统发送文件

在和 GDB 通讯的链路上，某些远程目标还具有发送文件的功能。这个功能方便了用其他方式来访问目标，例如，GNU/Linux 系统在网络接口之上运行 gdbserver。对于其他目标，例如，只有一个串口的嵌入式设备，只有此功能此能上传、下载文件。

并不是所有的远程目标都支持这些命令。

remote put hostfile targetfile

从主机（运行着 GDB 的机器）拷贝文件 hostfile 到目标系统，保存为 targetfile。

remote get targetfile hostfile

从目标系统拷贝文件 targetfile 到主机，保存为 hostfile。

remote delete targetfile

删除目标系统上的文件 targetfile。

17.3 使用 gdbserver 程序

gdbserver 是类 Unix 系统上的控制程序，可以允许用户通过用 target remote 将自己的程序和远程 GDB 通讯--不需要通过常用的调试代理来连接。

gdbserver 不是完全的调试代理的替代物，因为它和 GDB 需要相同的操作系统功能。事实上，能够运行 gdbserver 连接到远程 GDB 的系统同样能够在本地运行 GDB！不论如何，某些时候 gdbserver 会很有用，因为它比 GDB 小很多。移植 gdbserver 比移植全部 GDB 要容易的多，因此在新系统上用 gdbserver 可以更快的开始工作。最后，如果为实时系统写代码的话，实时操作的权衡会让 gdbserver 更方便，就如

在其它系统上的开发工作一样多，例如使用交叉编译。使用 gdbserver 可以选择相似的调试。

GDB 和 gdbserver 在串行线或者 TCP 连接上，使用标准的 GDB 远程串行协议相互通讯。

警告：gdbserver 没有任何内置安全性。不要在连接到任何公共网络上的系统上运行 gdbserver；GDB 和 gdbserver 的连接具有和运行 gdbserver 用户相同的权限访问目标系统。

17.3.1 运行 gdbserver

在目标系统上运行 gdbserver。需要被调试程序的一份拷贝，包括它所需的所有链接库。gdbserver 不需要程序的符号表，所以为了节省空间，可以 strip 程序。主机系统上的 GDB 负责处理符号。

要使用服务器，必须告诉它如何和 GDB 通讯；程序名；和程序的参数。常用的命令是：

target> gdbserver comm program [args ...]

comm 可以是设备名（使用串行线），也可以是 TCP 主机名和端口号。例如，要调试带参数'foo.txt'的 Emacs，其通过串口'/dev/com1'和

GDB 通讯：

target> gdbserver /dev/com1 emacs foo.txt

gdbserver 会被动的等待主机上的 GDB 和自己通讯。

不使用串行线，而要使用 TCP 连接的话：

target> gdbserver host:2345 emacs foo.txt

和前面的例子相比，唯一的不同之处是第一个参数，它指定通过 TCP 来和主机上的 GDB 通讯。参数'host:2345'表明 gdbserver 期待的连接是从主机'host'到本地 TCP 端口 2345 的连接。（目前，忽略'host'部分。）可以选择任意端口号，只要这个号码和目标系统上已经使用的不冲突（例如，23 保留给 telnet）。（注释，如果使用冲突的端口号，gdbserver 会打印出错信息。）必须和主机 GDB 的 target remote 命令使用相同的端口号。

17.3.1.1 附着到运行着的程序

在某些目标上，gdbserver 也可以附着到已经运行的程序上。此功能通过--attach 参数实现。语法是：

target> gdbserver --attach comm pid

pid 是当前运行进程的进程号。不必要用二进制方式告知 gdbserver 运行着的进程。

如果目标上有 pidof 工具的话，就可以用进程名替代进程号：

target> gdbserver --attach comm ‘pidof program‘

如果有多个程序的副本在运行的话，或者程序有多个线程，大多数版本的 pidof 支持-s 选项来只返回第一个进程号。

17.3.1.2 gdbserver 的多进程模式

如果用 target remote 连接 gdbserver，gdbserver 只会调试指定的程序。要是程序退出，或者你从程序里脱离的话，GDB 会断开连接，然后 gdbserver 就会退出。

如果用 target extended-remote 来连接，gdbserver 会进入多进程模式。要是被调试的程序退出的话，或者你从此程序脱离的话，GDB 会保持和 gdbserver 的连接，即使没有程序运行。run 和 attach 命令指示 gdbserver 运行或者附着到新的程序。run 命令使用 set remote exec-file（参见[set remote exec-file]，185 页）来选择运行程序。支持命令行参数，除了通配符扩展和 I/O 重定向（参见 4.3 节[参数]，28 页）。

要启动 gdbserver 而不带初始命令来运行或进程号以附着的话，使用'--multi'命令行选项。接着用 target extended-remote 连接和启动你想要调试的程序。

在多进程模式下，gdbserver 不会自动退出。要结束它的话，可以用 monitor exit（参见[gdbserver 监视器命令]，183 页）。

17.3.1.3 其它 gdbserver 命令行参数

gdbserver 命令行里可以包含'--debug'。gdbserver 会额外显示调试进程的状态信息。此选项是为 gdbserver 开发和 bug 报告而设的。

'--wrapper'选项指定一个适配器来启动要调试的程序。此选项应该后接适配器名字，接下来是传递给适配器的命令行参数，再接着--，表示适配器参数结束。

gdbserver 用组合命令行和适配器参数来执行指定的适配器程序，接着是被调试程序名，接着是被调试程序的参数。适配器会一直运行到它执行被调试程序为止，接着 GDB 就会获得控制权。

只要最终调用 execve 的程序，都可以作为适配器。有几个标准 Unix 工具可以做到这个，例如，env 和 nohup。任何以 exec "$@"结尾的 Unix shell 脚本也都可以做到这个。

例如，可以用 env 来将环境变量传递给被调试程序，而不需要设置 gdbserver 的环境变量：

$ gdbserver --wrapper env LD_PRELOAD=libtest.so -- :2222 ./testprog

17.3.2 连接 gdbserver

在主机上执行 GDB.

首先确认有必须的符号文件。在连接前，用 file 命令为应用程序加载符号。用 set sysroot 来定位目标库（除非你的 GDB 是用--with-sysroot 和 sysroot 正确编译的）。

符号文件和目标库和目标上的可执行程序和库必须是完全一致的，除了一个例外：主机系统上的文件不能是精简过的，而目标系统上的文件可以精简。不匹配或者文件缺失的话，在调试过程中可能会导致让人迷惑的结果。在 GNU/Linux 目标上，不匹配或者文件缺失也可能使得 gdbserver 不能调试多线程程序。

来接到目标(参见 17.1 节[连接到远程目标]，179 页)。对于 TCP 连接，必须用 target remote 命令先启动 gdbserver。否则会得到一个错误，此错误的文本信息依赖于主机系统，不过通常类似于"Connection refused"。当使用 gdbserver 的时候，不要在 GDB 里用 load 命令，因为此程序已经在目标系统上了。

17.3.3 gdbserver 的监视命令

在使用 gdbserver 的 GDB 会话期间，可以用 monitor 命令来给 gdbserver 发送特殊的请求。下面是可用的命令。

monitor help

列出可用监视命令。

monitor set debug 0

monitor set debug 1

关闭或打开普通调试信息。

monitor set remote-debug 0

monitor set remote-debug 1

关闭或打开和远程协议相关的调试信息（参见附录 D[远程协议]，347 页）。

monitor exit

通知 gdbserver 立即退出。此命令后应该接着 disconnect 来结束调试会话。gdbserver 会将附加的进程剥离，并将其创建的进程杀死。在多进程调试会话的结尾，用 monitor exit 结束 gdbserver。

17.4 远程配置

本节介绍调试远程程序的可用配置选项。关于远程协议的文件 I/O 扩展相关的选项，参见[系统]，381 页。

set remoteaddresssize bits

将存储器包里的地址最大值存储到指定数字的位中。为远程目标设置地址之后，GDB 会将此数据上的地址位屏蔽掉。缺省值是目标

地址里的位总数。

show remoteaddresssize

显示当前远程地址大小。

set remotebaud n

设置远程串口 I/O 波特率为 n 波特率。此值用于设置串口的速率。

show remotebaud

显示当前远程连接的速率。

set remotebreak

如果设置为 on，在输入 Ctrl-c 来终止在远程目标上运行的程序时，GDB 发送一个 BREAK 信号给远程目标。如果设置为 off，GDB 会发送

一个'Ctrl-C'。缺省是 off，因为多数远程系统会认为'Ctrl-c'是中断信号。

show remotebreak

显示 GDB 是发送 BREAK 还是'Ctrl-C'来中断远程程序。

set remoteflow on

set remoteflow off

打开或关闭和远程目标通讯的串口硬件流控（RTS/CTS）。

show remoteflow

显示当前硬件流控的设置。

set remotelogbase base

将登陆串口通讯协议的基数（也称为数制）设置为 base。支持的数制为：ascii,octal 和 hex。缺省是 ascii。

show remotelogbase

显示当前登陆的远程串口协议的数制。

set remotelogfile file

将远程串口通讯记录于文件 file。缺省是不记录。

show remotelogfile.

显示当前存储串口通讯的文件名。

set remotetimeout num

设置等待远程目标回应的超时为 num 秒。缺省是 2 秒。

show remotetimeout

显示当前远程目标回应的超时。

set remote hardware-watchpoint-limit limit

set remote hardware-breakpoint-limit limit

限制 GDB 使 u 用 limit 个远程硬件断点或监视点。缺省是-1，没有限制。

set remote exec-file filename

show remote exec-file

选择可执行文件，用于在 target extended-remote 上 run。应该设置在目标系统上有效的文件名。如果没有设置，目标会使用缺省文件名（例如，最后一次运行的程序）。

GDB 远程协议自动探测调试代理支持的数据包。如果要去除自动探测，可以使用命令打开或关闭单独的数据包。可以为每种包设置'on'（远程目标支持此包），也可以设置为'off'（远程目标不支持此包），或者'auto'（探测远程目标支持此包与否）。这些设置的缺省值都是'auto'。更多关于每种数据包的信息，参见附录 D[远程协议]，347 页。

通常的调试中，应该不需要使用这些命令。如果确实需要，可能是你的远程调试代理有 bug，或者 GDB 有 bug。你可能要向 GDB 开发者报告问题。

对于每种数据包名，打开或关闭这种包的命令是 set remote name-packet。可用的设置是：

命令名 远程数据包 相关的功能

fetch-register pinfo registers

set-register Pset

binary-downloadXload, set

read-aux-vectorqXfer:auxv:readinfo auxv

symbol-lookupqSymbolDetecting multiple threads

attach vAttachattach

verbose-resume vContStepping or resuming multiple threads

runvRun run

software-breakpointZ0 break

hardware-breakpointZ1 hbreak

write-watchpoint Z2 watch

read-watchpointZ3 rwatch

access-watchpointZ4 awatch

target-featuresqXfer:features:readset architecture

library-info qXfer:libraries:read info sharedlibrary

memory-map qXfer:memory-map:readinfo mem

read-spu-objectqXfer:spu:read info spu

write-spu-object qXfer:spu:writeinfo spu

get-thread-localstorageqGetTLSAddrDisplaying__thread variables

-address

search-memoryqSearch:memory find

supported-packetsqSupported Remote communications parameters

pass-signals QPassSignals handle signal

hostio-close-packetvFile:closeremote get, remote put

hostio-open-packet vFile:open remote get, remote put

hostio-pread-packetvFile:preadremote get, remote put

hostio-pwrite-packet vFile:pwrite remote get, remote put

hostio-unlink-packet vFile:unlink remote delete

noack-packet QStartNoAckModePacket acknowledgment

17.5 实现远程代理

代理文件提供了目标端的通讯协议的实现，GDB 端的实现在 GDB 源代码文件'remote.c'里。通常，只

要简单的让子程序通讯就可以了，而不需要关心细节。（如果实现自己的代理代码，还是可以忽略细节：启动一个已有的代理程序。'sparc-stub.c'是组织的最好，而最容易读的代码。）

要提示运行于另一机器上的程序（也就是调试目标机器），首先你必须要让程序能自己准备好所有的先决步骤。例如，对于 C 程序来说，需要：

1.一个启动例程以设置 C 运行时环境；此例程名通常是'crt0'。启动例程可能是硬件供应商提供的，或者需要你自己实现。

2.一个 C 子例程库以支持程序的子例程调用，主要是是管理输入和输出。

3.将程序放到别的机器上的方法--例如，下载程序。通常由硬件制造商提供，不过也可能需要你根据硬件文档自己实现。

下一步是让程序通过串口和 GDB 所在的机器通讯（主机）。总体上来说，框架如下：

在主机上：

GDB 已知道如何使用此协议；所有事情都设置好后，就可以用'target remote'命令了（参见 16 张[指定调试目标]，175 页）。

在目标上：

必须让程序和一些特殊的子例程一起链接，这些子例程实现了 GDB 远程串口协议。包含这些子例程的代码称为调试代理。

在某些远程目标上，可以用辅助程序 gdbserver 而不用将代理连接进你的程序里。详细说明，参见 17.3 节[使用 gdbserver 程序]，181 页。

调试代理和远程机器的架构密切相关的；例如，要用'sparc-stub.c'来在 SPARC 板子上调试程序。

下面的远程代理随 GDB 发布：

i386-stub.c

Intel 386 和兼容系统。

m68k-stub.c

Motorola 68x0 架构。

sh-stub.c

Renesas SH 架构。

sparcl-stub.c

Fujitsu sparclite 架构。

GDB 发行版里的'README'文件里列出最近添加的代理，如果有的话。

17.5.1 代理能为你做什么

调试代理为用户的系统架构提供 3 个子例程：

set_debu_traps

此例程在程序中断的时候调用 handle_exception。必须在程序开始运行时显式调用此例程。

handle_exception

此例程是代理的中心，但程序从不会显式调用它---设置代码会在触发一个陷阱（也即是中断之类）的时候调用 handle_exception。在程序中断执行的时候，handle_exception 将取得执行的控制权（例如，执行到一个中断的时候），并且和主机上的 GDB 通讯。

通讯协议也在此例程里实现；handle_exception 是目标机器上的 GDB 代表。开始的时候，它会发送程序状态的简要信息，接着会执行程序，获取并传送 GDB 需要的信息，直到用户执行 GDB 命令来重新执行程序；在此时，handle_exception 将控制权交换给目标机器上的程序。

breakpoint

此辅助例程可以让程序得到断点。在某些特殊的环境里，GDB 肯能只能用此方式得到控制权。例如，如果目标系统有某种中断按钮，你可能不需要调用此例程；按下中断按钮可以将控制权交给 handle_exception---实际上是给 GDB。在某些系统上，仅仅从串口上获取字符就可能引发陷阱；再有，在那样的环境里，用户不必从用户自己的程序里调用 breakpoint---仅从主机 GDB 里运行'target remote'就可以获得控制权。

如果以上限制都没有的话，或者只是想让程序在预先设定的地方中断来开始调试的话，调用 breakpoint。

17.5.2 你必须为代理做什么

随同 GDB 发布的调试代理可以设置特殊的芯片架构，但它们不知道调试目标系统余下的信息。

首先，必须告诉代理如何和串口进行通讯。

int getDebugChar()

此例程从串口读取一个字符。此例程可能和目标系统上 getchar 相同；不同的名字可以区分它们，如

果你想如此的话。

void putDebugChar(int)

此例程将一个字符写到串口去。此例程可能和目标系统上 putchar 相同;；不同的名字可以区分它们，如果你想如此的话。

如果你想让 GDB 能够在程序运行的时候中断它的话，需要使用中断驱动的串口驱动，并设法让它在接受到'^C'（'\003',control-C 字符）的时候中断。GDB 用词字符通知远程系统中断。

要让调试目标将正确的状态返回给 GDB，可能需要修改标准代理；一个快捷但有害的方式是仅仅执行断点指令（"有害"是指 GDB 报告 SIGTRAP 而不是 SIGINT）。

其它需要提供的例程是：

void exceptionHandler (int exception_number, void *exception_address)

此函数用来在异常处理表注册 exception_address。由于代理没有任何途径了解目标系统的异常处理表是如何工作的（例如，处理器的异常表可能在 ROM，其包含了指向 RAM 里的表的入口），因此必须实现此函数。exception number 是要修改的异常编号；它的内涵是架构相关的（例如，不同的编号可能表示被零除，没有对齐的访问，等等）。在异常发生的时候，控制可以直接交给 exception_address，并且处理器状态（栈，寄存器等）应该和处理器异常发生一样。因此如果要用跳转指令来跳转到 exception_address，此跳转应该只是一个跳转，而不是跳转到一个子例程。

对于 386 而言，exception_address 应该安装为一个中断门，在中断处理函数运行的时候可以被屏蔽。此门应该处于 0 级特权（最高权限）。SPARC 和 68k 代理不需要 exceptionHandler 的帮助就可以屏蔽中断自身。

void flush_i_cache()

在 SAPRC 和 SPARCLITE 架构上，此子例程清空目标机器上的指令缓存，如果有的话。如果没有指令缓存，此子例程可以是空操作。

在目标系统上有指令缓存的话，GDB 需要此例程来确保程序状态的稳定。

必须确保有以下的库函数：

void *memset(void *, int, int)

标准库函数 memset 将一块存储器设置为一个既定值。如果有一个自由版的 libc.a，代理可以在那找到 memset；否则，必须从硬件制造商那里得到，或者自己实现。

如果不使用 GNU C 编译器，那就可能需要其它标准库函数；这样的话，代理的实现之间可能存在差异，但通常代理可能使用公共的库函数，而 GCC 将这些库函数链接成为内联代码。

17.5.3 集成

概括起来，在准备好调试程序时，必须符合如下步骤。

1.确保已经定义了低级例程（参见 17.5.2 节[你必须为代理做什么]，189 页）：

getDebugChar, putDebugChar,

flush_i_cache, memset, exceptionHandler.

2.在程序的开头插入下面的两行代码：

set_debug_traps();

breakpoint();

3.对于 680X0 代理，需啊哟你提供一个名为 exceptionHook 的变量。通常可以使用：

void (*exceptionHook)() = 0;

不过，要是在调用 set_debug_traps，设置其为指向你的程序里的一个函数的话，那么这个函数就会在 GDB 从一个陷阱（例如，总线错误）中断后继续执行的时候被调用。exceptionHook 激发的函数带有一个参数：一个整型参数，代表一个异常编号。

4.编译并链接到一起：程序，GDB 调试代理和支持例程。

5.确保在目标机器和 GDB 主机之间有一个串口连接，并且指定主机上的串口号。

6.下载程序到目标机器上（或者用制造商提供的方法将其放到机器上），然后开始运行程序。

7.在主机上开始运行 GDB，然后连接到目标（参见 17.1 节[连接到远程目标]，179 页）。

第18章 配置相关的信息

虽然差不多全部 GDB 命令对于本地和交叉版本的调试器都是可用的，但还是有些例外。本章将介绍只有在某些配置下才可用的命令。

有三种主要的配置：本地配置，此配置的宿主和目标都是同一个机器，嵌入式操作系统配置，此配置通常和几个不同处理器架构一样，裸嵌入式处理器，此配置的差异非常大。

18.1 本地

本节介绍和特殊的本地配置相关的细节。

18.1.1 HP-UX

在 HP-HX 系统里，如果如果引用到以符号$开头的函数或变量名，GDB 会首先搜索用户和系统名，然后再搜索惯用变量。

18.1.1 BSD libkvm 接口

BSD 类系统（FreeBSD/NetBSD/OpenBSD）提供一个内核存储器接口，可以用统一的接口来访问内核虚拟存储映像，包括系统和崩溃转储。GDB 使用此接口来在很多本地 BSD 配置上提供调试内核和内核崩溃转储。要调试活动的系统，加载当前运行着的内核到 GDB 并连接到 kvm 目标：

(gdb) target kvm

要调试崩溃转储，要用崩溃转储的文件名作为参数：

(gdb) target kvm /var/crash/bsd.0

一旦连接到 kvm 目标，用下面的命令：

kvm pcb 将 PCB（进程控制块）地址设置为当前上下文。

kvm proc 将 proc 地址设置为当前上下文。此命令在现代 FreeBSD 系统下不可用。

18.1.2 SVR4 进程信息

SVR4 的许多版本和相容的系统都提供名为'/proc'的工具，可以用于检查运行着的进程所使用的文件系统子例程的映像。如果 GDB 在带有这个工具的系统上配置的话，命令 info proc 就能收集运行程序的进程的信息，或者你的系统上的任意进程的信息。info proc 只能在那些包含 procfs 代码上的 SVR4 系统上工作。包括，gnu/Linux, OSF/1 (Digital Unix), Solaris, Irix, 和 Unixware，但不包括 HP-UX，举例来说。

info proc

info proc process-id

概述进程的可用信息。如果指定了进程 id，只显示此进程的信息；否则显示被调试进程的信息。概述信息包括调试进程 ID，命令行，当前工作目录，可执行文件的绝对文件名。

某些系统里，进程号可以是'[pid]/tid'的形式，此形式标明了一个进程内的某个线程号。如果缺失可选的 pid 部分，意味着此进程的线程正处于调试状态（前导的'/'还是必须的，否则 GDB 会将此数字认为是进程号，而不是线程号）。

info proc mappings

报告此进程内可用的内存地址空间范围，还包括此进程的各部分是否具有可读，写，可执行的权利。在 GNU/Linux 系统里，每个存储器范围包含映射到此范围的目标文件，而不是此范围的存储器方位权。

info proc stat

info proc status

这两个字命令是 GNU/Linux 系统特有的。它们显示进程相关的信息，包括 user id 和 group id；进程内有多少线程；虚拟内存的使用；挂起的信号，阻塞的信号，忽略的信号；TTY；消耗的系统和用户时间；堆栈大小；'nice'值；等等。更多信息，参见

'proc'手册（在 shell 上输入 man 5 proc）。

info proc all

显示上面 info proc 子命令描述的所有的信息。

set procfs-trace

此命令打开和关闭 procfs API 调用的跟踪。

show procfs-trace

此命令显示当前 procfs API 调用的跟踪状态。

set procfs-file file

让 GDB 将 procfs API 的跟踪信息写入 file 文件。GDB 会将跟踪信息附加到文件里。缺省是将跟踪信息显示到标准输出里。

show procfs-file

显示保存 procfs API 跟踪信息的文件。

proc-trace-entry

proc-trace-exit

proc-untrace-entry

proc-untrace-exit

这些命令打开或关闭 syscall 接口的跟踪。

info pidlist

此命令是 QNX Neutrino 特有，显示所有进程的列表和每个进程里的所有线程。

info meminfo

此命令是 QNX Neutrino 特有，显示所有映射信息的列表。

18.1.3 调试 DJGPP 程序的功能

DJGPP 是 MS-DOS 和 MS-Windows 下的 GNU 开发工具的一部分。DJGPP 程序是 32 位保护模式程序，使用 DPMI(DOS Protected-Mode Interface，DOS 保护模式接口)API，运行于实模式 DOS 系统和其模拟器之上。

GDB 支持本地调试 DJGPP 程序，并定义了一些 DJGPP 口特有的命令。本节将介绍这些命令。

info dos

本命令用于打印目标系统和重要的 OS 结构的信息。

info dos sysinfo

本命令打印系统的分类信息：CPU 类型和功能，OS 版本号和， DPMI 版本，可用的常用和 DPMI 存储器。

info dos gdt

info dos ldt

info dos idt

这 3 个命令分别打印全局，本地和中断描述符表(GDT,LDT 和 IDT)。描述符表是存储描述符的数据结

构，每个描述符代表一个当前使用的段。段选择器是指向一个描述符表的索引；这个索引所代表的表条目保存着描述符的基址和边界，其属性和访问权限。

典型的 DJGPP 程序使用 3 个段：代码段，数据段（用于数据和栈）和 DOS 段（此段允许访问 DOS/BIOS 数据结构和在常用存储器里绝对地址）。不过，DJGPP 主机通常会定义附加段来支持 DJGPP 环境。

这些命令允许打印描述符表的条目。不带参数的话，打印指定描述符表所有的条目。一个参数的话，并且这个参数必须是一个整数表达式，将会打印这个参数指定的表的条目。例如，下面是打印调试程序数据段的常用方式：

(gdb) info dos ldt $ds

0x13f: base=0x11970000 limit=0x0009ffff 32-Bit Data (Read/Write, Exp-up)

用这个方式可以很方便的查看一个指针是否越出数据段（例如被篡改了）。

info dos pde

info dos pte

这两个命令分别打印页目录和页表的条目。页目录和页表是存放控制虚拟存储器的地址和所映射的物理地址的数据结构。页表包含一个条目，这个条目里指向一个映射到程序地址空间的存储器页；可能有几个页表，每个最多包含 4096 个条目。页目录最多有 4096 个条目，每个条目指向当前正在使用的页表。

不带参数的话，info dos pde 打印整个页目录，info dos pte 打印页表里所有的条目。一个整数表达式的参数的话，info dos pde 命令打印此参数代表的页目录表。info dos pte 带一个参数的话，打印单个页表的一个条目，此页表号由页目录所指定。

要是程序使用了 DMA(Direct Memory Access，直接存储器访问)的话，那这两个命令就很有用，因为 DMA 需要物理地址来编 DMA 控制器。

这些命令只支持某些 DPMI 服务器。

info dos address-pte addr

这个命令显示指定地址所在的页表条目。参数 addr 是一个线性地址，其应该是已经和正确的段基址相加的，因为此命令接受的地址参数允许属于各个段。例如，下面是显示变量 i 所在的页的页表条目：

(gdb) info dos address-pte __djgpp_base_address + (char *)&i

Page Table entry for address 0x11a00d30:

Base=0x02698000 Dirty Acc. Not-Cached Write-Back Usr Read-Write +0xd30

这是说 i 存储于相对页基址的偏移地址地址 0xd30 上，此页的物理地址基址是 0x02698000，并且显示此页的所有属性。

注意，必须将变量的地址转化为 char *，因为否则的话，__djgpp_base_address 的值--DJGPP 里所有变量和函数的基址

会用 C 指针算法来累加：如果 i 声明为一个整形 int 变量，那么 GDB 会加__djgpp_base_address 4 次到 i 的地址上。

下面是另外一个例子，显示转换缓冲区的页表条目：

(gdb) info dos address-pte *((unsigned *)&_go32_info_block + 3)

Page Table entry for address 0x29110:

Base=0x00029000 Dirty Acc. Not-Cached Write-Back Usr Read-Write +0x110

（加偏移 3 是因为转换缓冲区的地址是_go32_info_block 结构的第三个成员。）输出很清楚的告诉我们 DPMI 服务器将其地址以 1:1 的比例映射到惯用存储器上，例如，物理（0x00029000+0x110）和线性（0x29220）地址是相同的。

除了本地调试，DJGPP 口支持通过串口连接远程调试。下列命令专门支持 GDB 的 GJGPP 口远程串口调试。

set com1base addr

本命令设置串口'COM1'的 I/O 口基址。

set com1irq irq

本命令设置'COM1'串口使用的中断（IRQ）线。

还有些类似的命令可以设置端口地址和 IRQ 线，如"set com2base", "set com3irq"，等等。

有相关的命令来显示当前的基址和 IRQ 线的设置，如"show com1base"，"show com1irq"，等等。

info serial

此命令打印 4 个 DOS 串口的状态。对于每一个端口，打印其是激活与否，其 I/O 基址和 IRQ 号，是否使用 16550-格式的 FIFO，其波特率和到目前为止的各种错误的总和。

18.1.4 调试 MS Windows PE 格式可执行程序的功能

GDB 支持本地调试 MS Windows 程序，还可以有调试符号信息 DLL，也可以调试不带调试符号信息的 DLL。本节会介绍各种 Cygwin 相关的命令。18.5.1 节[没有调试符号的 DLL]介绍调试不带调试符号的 DLL。

info w32

MS Windows 相关命令的前缀，其会打印目标系统的信息和重要的 OS 结构。

info w32 selector

本命令显示 Win32 API 函数 GetThreadSelectorEntry 的返回信息。可以带一个可选的长整型参数，来指定选择器。不带参数的话，此命令会打印六个段寄存器的信息。

info dll

此命令是 Cygwin 特有的命令 info shared 的别名。

dll-symbols

此命令从 dll 里加载符号，和 add-sym 命令相似，但不需要指定基址。

set cygwin-exceptions mode

如果 mode 是 on，GDB 会 Cygwin DLL 里出现异常的时候中断。如果 mode 是 off，GDB 会延迟识别异常，而且可能会忽略某些可能是内部

Cygwin DLL"bookkeeping"引发的异常。这个选项主要是为了调试 Cygwin DLL 本省而设的；默认值是 off，以此来避开恼人的伪 SIGSEGV 信号。

show cygwin-exceptions

显示 GDB 在 Cygwin DLL 本身发生异常时，是否中断。

set new-console mode

如果 mode 为 on，被调试程序会在下一次调试开始时在新的控制台上开始。如果 mode 是 off，被调试程序还在原来的控制台上。

show new-console

显示被调试程序是否在新的控制台上开始。

set new-group mode

此布尔值参数控制被调试程序是启动一个新的组还是和 GDB 在同一个组里。此设置会影响 Windows OS 处理"Ctrl-C"的行为。

show new-group

显示当前 new-group 的布尔值。

set debugevents

此布尔值可以设置和让调试器得到与被调试程序相关的内核事件的调试信息。这些事件包括线程信号，进程创建和结束，DLL 加载和卸载，控制台中断，Windows OutputDebugString API 调用产生的调试信息。

set debugexec

此布尔值可以让调试器得到与执行事件（例如线程继续执行）相关的调试输出。

set debugexecptions

此布尔值可以让调试器得到被调试程序里发生异常的调试信息。

set debugmemory

此布尔值可以让调试器得到被调试程序读写存储器的调试输出信息。

set shell

此布尔值设置被调试程序是从 shell 执行还是直接调用的（缺省是 on）。

show shell

显示被调试程序是否是从 shell 里执行的。

18.1.4.1 支持不带调试符号的 DLLs

程序运行所需要的 DLL 不带调试符号信息的情况，在 windows 里很常见（例如'kernel32.dll'）。如果 GDB 不能识别 DLL 里的调试符号时，它就会依赖 DLL 的输出表里包含的很少的符号信息。本节描述 GDB 如何利用这些称为“最小符号”符号。

注意，在被调试程序开始运行前，DLL 不会被加载。绕开这个问题最简单的方法是执行程序---无论是设置一个断点还是让程序一次执行到结束。也可以让 GDB 在执行程序之前强制加载一个特殊的 DLL--参见共享库信息，15.1 节[文件]，155 页，或者 dll-symbos 命令，在 18.1.5 节[Cygwin Native],187 页。目前，从不带调试信息的 DLL 里显式加载符号，会导致在 GDB 的查找表里复制一份符号名，而这会显著的影响符号查找性能。

18.1.4.2 DLL 名的前缀

为了和 Microsoft 调试工具的命名习惯保持一致，DLL 输出符号会有基于 DLL 名的前缀，例如 KERNEL32!CreateFileA. 符号表里存储了全名，所以 CreateFileA 通常都能满足要求。程序里有时候会有命名冲突的情况（特别是可执行程序自身包含调试符号的情况下），需要使用完整的名字来引用 DLL 的内容。在名字前后使用单引号来避免声明符（"!"）转译为语言操作符。

注意，DLL 的内部名可以全部大写，即使 DLL 的文件名是小写的，反之亦然。由于 GDB 内的符号是大小写敏感的，这样就会让人迷惑。如果有疑问，试试用 info functions 和 info variables 命令或者用 maint

jprint msymbols（参见 13 章[符号]，143 页）。下面是一个例子：

```
(gdb) info function CreateFileA
All functions matching regular expression "CreateFileA":
Non-debugging symbols:
0x77e885f4 CreateFileA
0x77e885f4 KERNEL32!CreateFileA
(gdb) info function !
All functions matching regular expression "!":
Non-debugging symbols:
0x6100114c cygwin1!__assert
0x61004034 cygwin1!_dll_crt0@0
0x61004240 cygwin1!dll_crt0(per_process *)
[etc...]
```

18.1.4.3 调试最小符号 DLL

从 DLL 的输出表里取得的符号不会包含很多类型信息。GDB 能做的就只能猜测符号是指向函数还是变量，这取决于连接器段是否包含符号。还要注意的是，DLL 里包含的存储器里的内容只有在程序运行时才可用。这意味着在程序没有运行的状态下，不能检查 DLL 变量的内容或者反汇编函数。

通常变量会被当作指针并自动间接取值。由于这个缘故，通常在命令中需要在变量名前加上一个地址符("&")并且需要明确的类型性息。下面是这类问题的例子：

```
(gdb) print 'cygwin1!__argv'
$1 = 268572168
(gdb) x 'cygwin1!__argv'
0x10021610: "\230y\""
```

下面是两种可能的解决方案：

```
(gdb) print ((char **)'cygwin1!__argv')[0]
$2 = 0x22fd98 "/cygdrive/c/mydirectory/myprogram"
(gdb) x/2x &'cygwin1!__argv'
0x610c0aa8 <cygwin1!__argv>: 0x10021608 0x00000000
```

(gdb) x/x 0x10021608

0x10021608: 0x0022fd98

(gdb) x/s 0x0022fd98

0x22fd98: "/cygdrive/c/mydirectory/myprogram"