GDB 用户手册

目录

摘要

象 GDB 这样的调试程序，目的就是让你可以查看其它程序的内部运行过程，或者是在它崩溃的那一时刻它在做什么。

GDB 能做 4 件事（这些还需附加其他的一些事），帮助你捕获在场的错误：

• 启动程序，设定任何可以影响它行为的东西。

• 在特定的条件下使程序停止。

• 当程序停止时，分析发生了什么。

• 改变程序里的一些东西，进行一个由于 bug 所导致的结果的矫正性试验，同时继续了解另外一个 bug。

可以使用 GDB 调试用 C 和 C++编写的程序，更多信息参见支持的语言，及 C 与 C++。部分支持 Modula-2，Modula-2 的更多信息参见 Modula-2。

在调试使用 sets、subranges、file variables 或嵌套函数的 Pascal 程序时，目前不能工作。GDB 不支持 entering expressions、printing values 或者类似特性的 Pascal 语法。

GDB 可以调试 Fortran 写的程序，尽管那必然会涉及到带有下划线后缀的一些变量。

GDB 可以调试 Objective-C 写的程序，既可以使用 Apple/NeXT 运行时库，也可以使用 GNU Objective-C 运行时库。

自由软件

GDB 是自由软件，受 GNU 公共许可证（GPL）保护。GPL 给予了你自由复制或改编程序的许可——就是说获得拷贝的人也就获得了自由修改它的权利（这意味着他们必须有权访问源代码），而且可以自由的发布更多的拷贝。大部分软件公司所使用的版权限制了你的自由。自由软件基金会利用 GLP 保护了这些自由。

基本上来说，公共许可证是一个说明你拥有这些自由的许可证，而且你不能把这些自由从任何人那里占为己有。

自由软件急需自由文档

当今的自由软件社区所存在的最大缺憾不在于软件——而在于没有我们可以随同自由

软件包含在一起的好文档。好多我们十分重要的程序没有一同提供自由的参考指南和介绍性文本。对任何一个软件包来说，文档是最基本的部分。当一个重要的自由软件包没有与一个自由手册或指南一同提供时，那就是一个极大的缺憾。如今，我们拥有太多这样的缺憾了！

拿 Perl 来说，人们日常所使用的指导手册就不是免费的。为什么会这样呢？因为这些手册的作者们在发表它们的时候伴有很多限制项目——不能复制、不能修改、不能得到源文件——把它们从自由软件世界中驱逐出去了。

这类的事情已经不只发生过一次了，而且今后还会陆续发生。我们经常听到某位热心的 GNU 用户说他正在编写的一个手册，他打算把它捐献给社区，可没想到他签署了出版合同而使这个手册不自由了，所有的期望全都破灭。

自由文档，就像自由软件一样，是自由的，不需要付费的东西。非自由手册的问题不在于发行商为印刷拷贝所要承担的费用——只要它本身很好就行（自由软件基金会也出售印刷拷贝），而在于这个问题会约束手册的利用。自由手册可以以源代码的方式获得，允许复制与修改。非自由手册是不允许这么做的。

自由文档自由度的标准，一般来说与自由软件差不多。再发布（包括很多常规的商业再发布）必须被允许，不管是以在线形式还以书面形式，以便手册可以伴随着程序的每一份拷贝。

允许有关技术性方面的内容的更正也是至关重要的。当人们更改软件，添加或改变其某些特性时，如果他们负责任的话，也将会修改相应的手册——因而，他们能够为修改过的程序提供准确而清晰的文档。某个手册的页数你是无法决定的，但是为某个程序的变更版本写一份全新的手册，对于我们的社区来说，那真是没有必要。

在改进过程中所运用的某些限制是合理的。例如，要求保持原作者的版权通告、发布条款、以及作者名单，是没有问题的。在修正版本中包含是他们更正的通告也是没有问题的。只要论述的是非技术性的话题（就像这一章），可以接受连续完整的章节不可删除或被更改。能够接受这些限制，是因为它们不会妨碍社区对手册的正常使用。

无论如何，必须允许对手册中所有关技术性方面的内容进行修改，然后通过所有正常的通道，利用所有常规的媒质，发布这个结果。否则，这些限制就妨碍了对手册的使用，那么它就是非自由的了，我们就得需要一个新的手册来代替它了。

请散布有关这一论点的言辞。我们的社区仍然在遗失好多手册，这些手册都在成为私有出版物。如果我们趁早散布自由软件急需自由参考手册和指南这样的言辞的话，也许下一个投稿人就会意识到，只有少数的手册投稿给了自由软件社区。

如果你正在撰写文档，请坚持在GNU的自由文档许可证或其他的自由许可证下出版它。别忘了，这个决策是需要争得你的赞同的——你不用理会出版社的决策。只要你坚持，某些出版社会使用自由许可证的，但是他们不能奢求有买卖的特权；那需要由你自己来发行，并且坚定地说：这就是你想要的。如果这个出版社拒绝了你的生意，那就再换一家。如果你不

能确定某个被提议的许可证是自由的，就写信给licensing@gnu.org。

你可以使用购买的方式来鼓励商业出版社出售更多的免费的，非赢利版权的手册与指南，尤其是购买那些来自于出版社的拷贝，付给他们撰写或作重大改进的费用。同时，尽量完全避免购买非自由的文档。在购买之前，先查看一下发布条款，不管谁要做你的生意都必须尊重你的自由。查看书的历史，设法奖励支付了作者们工资的那些出版社。

自由软件基金会在http://www.fsf.org/doc/other-free-books.html维护了一个已经由其他一些出版社出版了的文档的列表。

GDB的贡献者们

Richard Stallman 是 GDB 的原作者，也是其他好多 GNU 程序的原作者。好多人已经对它的开发作了贡献。谨以此节来表彰那些主要的贡献者们。自由软件的一个优点就是每个人都无偿的为它作贡献。遗憾的是，我们无法逐一向他们表示感谢。在 GDB 的发布中，有一个“ChangeLog”文件，做了极为详尽的说明。

2.0 版本以前的大量变化已湮灭在时间的迷雾中。

恳请：极力欢迎对本节的补充。如果您或您的朋友（或者是敌人，为了公平），不公平地在这个列表中被遗漏了，我们愿意加入您的名字。

为了使那些可能被遗忘的人们的工作不至于徒劳无功，在此特别感谢那些带领 GDB 走过各个重要发布版的那些人：Andrew Cagney（发布了 6.1, 6.0, 5.3, 5.2, 5.1 和 5.0 版）；Jim Blandy（发布了 4.18 版）；Jason Molenda（发布了 4.17 版）；Stan Shebs（发布了 4.14 版）；Fred Fish （发布了 4.16，4.15，4.13，4.12，4.11，4.10 和 4.9）；Stu Grossman 和 John Gilmore（发布了 4.8，4.7，4.6，4.5 和 4.4 版)；John Gilmore（发布了 4.3，4.2，4.1，4.0 和 3.9 版）；Jim Kingdon（发布了 3.5，3.4 和 3.3 版）；以及 Randy Smith（发布了 3.2，3.1 和 3.0）。

Richard Stallman，在 Peter TerMaat、Chris Hanson、和 Richard Mlynarik 的多次协助下，完成到了 2.8 版的发布。

Michael Tiemann 是 GDB 中大部分 GNU C++支持的作者，得益于来自 Per Bothner 和 Daniel Berlin 的其他的一些重要贡献。James Clark 编写了 GNU C++反签名编码器（demangler）。早期在 C++方面的工作是由 Peter TerMaat 做的（他也做了大量的到 3.0 发布版的常规更新工作）。

GDB 是使用 BFD 子程序库来分析多种目标文件格式的，BFD 是 David V. Henkel-Wallace、Rich Pixley、Steve Chamberlain 和 John Gilmore 的一个合作项目。

David Johnson 编写了最初的 COFF 支持。Pace Willison 做了最初的压缩的 COFF 支持。

哈里斯计算机系统（Harris Computer Systems）的 Brent Benson 贡献了 DWARF 2 的支持。

Adam de Boor 和 Bradley Davis 贡献了 ISI Optimum V 的支持。Per Bothner、Noboyuki Hikichi 和 Alessandro Forin 贡献了 MIPS 的支持。Jean-Daniel Fekete 贡献了 Sun 386i 的支持。Chris Hanson 改良了 HP9000 的支持。Noboyuki Hikichi 和 Tomoyuki Hasei 贡献了 Sony/News OS 3 的支持。David Johnson 贡献了 Encore Umax 的支持。Jyrki Kuoppala 贡献了 Altos 3068 的支持。Jeff Law 贡献了 HP PA 和 SOM 的支持。Keith Packard 贡献了 NS32k 的支持。Doug Rabson 贡献了 Acorn Risc Machine 的支持。Bob Rusk 贡献了 Harris Nighthawk CX-UX 的支持。Chris Smith 贡献了 Convex 的支持（还有 Fortran 的调试）。Jonathan Stone 贡献了 Pyramid 的支持。Michael Tiemann 贡献了 SPARC 的支持。Tim Tucker 贡献了对 Gould NP1 和 Gould Powernode 的支持。Pace Willison 贡献了 Intel 386 的支持。Jay Vosburgh 贡献了 Symmetry 的支持。Marko Mlinar 贡献了 OpenRISC 1000 的支持。

Andreas Schwab 贡献了 M68k GNU/Linux 的支持。

Rich Schaefer 和 Peter Schauer 为支持 SunOS 的共享库提供了帮助。

Jay Fenlason 和 Roland McGrath 保证了 GDB 和 GAS 适用于若干机器指令集。

Patrick Duval、Ted Goldstein、Vikram Koka 和 Glenn Engel 帮助开发了远程调试。Intel 公司、风河系统（Wind River Systems）、AMD、以及 ARM 分别贡献了 i960、VxWorks、A29K UDI 和 RDI targets 的远程调试模块。

Brian Fox，readline 库的作者，正在提供命令行编辑与命令历史功能。

SUNY Buffalo 的 Andrew Beers 编写了语言切换代码、Modula-2 的支持，并且贡献了此手册的语言一章。

Fred Fish 做了支持 Unix System Vr4 的大部分编写工作。他也增强了 command-completion 的支持，使其覆盖到了 C++的过载符号。

Hitachi America (现在是 Renesas America), Ltd。负责了对 H8/300、H8/500 和 Super-H 处理器的支持。

NEC 负责了对 v850、Vr4xxx 和 Vr5xxx 处理器的支持。

Mitsubishi（现在是 Renesas）负责了对 D10V、D30V 和 M32R/D 处理器的支持。

Toshiba 负责了对 TX39 Mips 处理器的支持。

Matsushita 负责了对 MN10200 和 MN10300 处理器的支持。

Fujitsu 负责了对 SPARClite 和 FR30 处理器的支持。

Kung Hsu、Jeff Law 和 Rick Sladkey 添加了对硬件监视点（hardware watchpoints）的支持。

Michael Snyder 添加了对跟踪点（tracepoints）的支持。

Stu Grossman 编写了 gdbserver。

Jim Kingdon、Peter Schauer、Ian Taylor、及 Stu Grossman，修复了几乎数不清的 bug，并且对整个 GDB 做了清理。

惠普公司（Hewlett-Packard Company）的一些人贡献了对 PA-RISC 2.0 体系、HP-UX 10.20、10.30 和 11.0 (窄模式)、HP 的内核执行线程、HP 的 aC++编译器、以及文本用户界面（旧称终端用户界面）的支持。他们是：Ben Krepp、Richard Title、John Bishop、Susan Macchia、 Kathy Mann、Satish Pai、India Paul、Steve Rehrauer 和 Elena Zannoni。Kim Haase 提供了此手册中的 HP-specific 信息。

DJ Delorie 为 DJGPP 项目，把 GDB 移植到了 MS-DOS 上。Robert Hoehne 对 DJGPP 的移植作了重大的贡献。

Cygnus Solutions 已负责起 GDB 的维护，自 1991 年以来已做了大量的开发工作。Cygnus 为 GDB 做全职工作的工程师有：Mark Alexander、Jim Blandy、 Per Bothner、Kevin Buettner、Edith Epstein、Chris Faylor、Fred Fish、Martin Hunt、Jim Ingham、John Gilmore、Stu Grossman、Kung Hsu、Jim Kingdon、John Metzler、Fernando Nasser、Geoffrey Noer、Dawn Perchik、Rich Pixley、Zdenek Radouch、Keith Seitz、Stan Shebs、David Taylor 和 Elena Zannoni. 另外还有：Dave Brolley、Ian Carmichael、Steve Chamberlain、Nick Clifton、JT Conklin、Stan Cox、DJ Delorie、Ulrich Drepper、Frank Eigler、Doug Evans、Sean Fagan、David Henkel-Wallace、Richard Henderson、Jeff Holcomb、Jeff Law、Jim Lemke、Tom Lord、Bob Manson、Michael Meissner、Jason Merrill、Catherine Moore、Drew Moseley、Ken Raeburn、Gavin Romig-Koch、Rob Savoye、Jamie Smith、Mike Stump、Ian Taylor、Angela Thomas、Michael Tiemann、Tom Tromey、Ron Unrau、Jim Wilson 和 David Zuhn，他们做了大大小小不同的贡献。

Andrew Cagney、Fernando Nasser 和 Elena Zannoni，他们在 Cygnus Solutions 工作时，实现了最初 GDB/MI 接口。

Jim Blandy 添加了预处理宏的支持，那时他在 Red Hat 工作。

Andrew Cagney 设计了 GDB 的结构向量。包括 Andrew Cagney、Stephane Carrez、Randolph Chung、Nick Duffek、Richard Henderson、Mark Kettenis、Grace Sainsbury、Kei

Sakamoto、Yoshinori Sato、Michael Snyder、Andreas Schwab、Jason Thorpe、Corinna Vinschen、Ulrich Weigand 和 Elena Zannoni 的很多人，为把旧有的体系结构移植到这个新的框架上提供了帮助。

本文是由 GDB 的管理员于 2005 年 7 月 16 日，使用 text2html 生成的。

1. 一个简单的GDB会话

你可以在你的业余时间利用本手册了解有关 GDB 的一切。可是，少数几个命令，就足以开始使用调试器了。本章就阐明了那些命令。

GNU m4（一个普通的宏处理器）的一个初级版本表现出下列 bug：有些时候，当我们改变它默认的引证串（quote string，译者注：也就是我们常说的引号）时，用于捕获一个在别处定义的宏的命令停止工作。在下列简短的 m4 会话中，我们定一个可扩展为 0000 的宏；然后我们利用 m4 内建的 defx 定义一个相同的东西 bar。可是当我们把左引证串(open quote string，译者注：英文直译为开引证串)改为<QUOTE>，右引证串（close quote string）改为<UNQUOTE>时，相同的程序不能定义新的替代名 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
C-d
m4: End of input: 0: fatal error: EOF in string
```

让我们利用 GDB 设法看一下发生了什么事。

```
$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 6.3.50.20050716, Copyright 1999 Free Software Foundation, Inc...
(gdb)
```

GDB 仅读取足够查找所需的符号数据，余下的按需读取。结果是，第一提示很快就出现了。我们告诉 GDB 使用一个比平常要窄的显示宽度，以使样例与本手册相适应。

```
（gdb）set width 70
```

我们要看一下 m4 内建的 changequote 是如何工作的。通过查看源代码，我们了解到与其相关的子程序是 m4_changequote，因此我们使用 GDB 的 break 命令在那里设置了一个断点。

```
（gdb）break m4_changequote
Breakpoint 1 at 0x62f4: file builtin.c, line 879.
```

使用 run 命令，在 GDB 的控制下，我们开始了 m4 的运行。只要控制没有到达 m4_changequote 子程序，程序就会像平常那样运行。

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

foo
0000
```

为了引发断点，我们调用 changequote。GDB 暂停执行 m4，显示中断处的上下文信息。

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

Breakpoint 1, m4_changequote (argc=3, argv=0x33c70)
    at builtin.c:879
879         if (bad_argc(TOKEN_DATA_TEXT(argv[0]),argc,1,3))
```

现在，我们使用命令 n（next）向前执行到当前函数的下一行。

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

set_quotes 像是一个有希望的子程序。我们可以使用命令 s（step）代替 next 进入到他的里面。Step 转到任意子程序中要被执行的下一行，因此他进入了 set_quotes。

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

显示器显示了目前 m4 所停在子程序（还有它的参数）的被调堆栈帧的展示。它显示了一个堆栈的摘要信息。我们可以使用 backtrace 命令（也可以被拼写为 bt）,查看总体上我们在堆栈上的位置：Backtrace 显示了每一个活动子程序的堆栈帧。

```
(gdb) bt
#0  set_quotes (lq=0x34c78 "<QUOTE>", rq=0x34c88 "")
    at input.c:530
#1  0x6344 in m4_changequote (argc=3, argv=0x33c70)
    at builtin.c:882
#2  0x8174 in expand_macro (sym=0x33320) at macro.c:242
#3  0x7a88 in expand_token (obs=0x0, t=209696, td=0xf7fffa30)
    at macro.c:71
#4  0x79dc in expand_input () at macro.c:40
#5  0x2930 in main (argc=0, argv=0xf7fffb20) at m4.c:195
```

我们向前单步调试更多的行，看看都发生了什么。前两次我们使用“s”；后两次为了避免陷入 xstrdup 子程序我们使用“n”。

```
(gdb) s
0x3b5c  532         if (rquote != def_rquote)
(gdb) s
0x3b80  535         lquote = (lq == nil || *lq == '\0') ?  \
def_lquote : xstrdup(lq);
(gdb) n
536         rquote = (rq == nil || *rq == '\0') ? def_rquote\
 : xstrdup(rq);
(gdb) n
538         len_lquote = strlen(rquote);
```

最后以行看起来有点古怪；我们能够检查变量 lquote 和 rquote，看看他们现在是不是我们新指定的左右引号。我们使用命令 p 来查看它们的值。

```
（gdb）p lquote
$1 = 0x35d40 "<QUOTE>"
（gdb）p rquote
$2 = 0x35d50 "<UNQUOTE>"
```

Lquote 和 rquote 的确是新的左右引号。为了能够查看上下文，我们可以使用 l（list）命令，围绕着当前行，显示十行源代码。

```
(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
```

让我们单步执行，通过设置 len_lquote 和 len_rquote 的那两行，然后查看一下它们的值。

```
（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_lquote 和 len_rquote 分别代表 lquote 和 rquote 的长度的话，那显然就是错

的。我们能够利用 p 命令给它们设定更好的值，因为它可以打印任何表达式的值——表达式中还可以有子程序调用并赋值。

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

那足以修复与 m4 内建的 defn 一同使用新引号的问题吗？我们可以使用 c（continue）命令使 m4 继续执行，然后试试刚开始时那个有问题的例子：

```
(gdb) c
Continuing.

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

baz
0000
```

成功了！新的引号已经可以象默认的一样工作了。问题看来就是两个输入错误导致了错误的长度计算造成的。我们输入一个 EOF 让 m4 退出。

```
C-d
Program exited normally.
```

“Program exited normally”是 GDB 发出的。它表明 m4 已经结束执行了。我们可以使用 GDB 的 quit 命令结束我们的 GDB 会话。

```
(gdb) quit
```

2. 征服GDB的进与出

本章讨论如何启动 GDB，并且如何离开 GDB。要点为：

- 输入“gdb”启动 GDB
- 输入 quit 或 C-d 退出 GDB

2.1 调用GDB

通过运行 gdb 程序来调用 GDB。一旦启动，GDB 就开始从终端读取命令，直到你让它退出为止。

你也可以使用多种参数与选项来运行 gdb，在一开始就能更多地为您定义调试环境。

对于命令行选项的描述，这里打算覆盖多种情况。在一些环境中，某些选项可能会不起作用。

启动 GDB 最常用的方法是使用一个参数，指定一个可执行程序：

gdb *program*

也可以指定一个可执行程序和一个 core 文件一起启动 gdb：

gdb *program core*

如果你想调试一个运行中的程序，作为代替，你可以指定一个进程 ID 为第二个参数：

gdb *program* 1234

将把 GDB 附着到进程 1234（除非你也有一个名叫“1234”的文件；GDB 总是先检查 core 文件）。

运动第二个命令行参数需要一个相当完整的操作系统；当我们利用 GDB 作为远程调试器附着到一个裸板上时，那里可能没有任何“进程”的概念，那也就常常无法获得 core dump。如果 GDB 没有能力附着或读取 core dumps 时，它会发出警告。

可以使用--args 选项，给 gdb 要调试的程序传递参数。这一选项使 gdb 的选项处理停止。

gdb –args gcc -O2 -c foo.c

这使得 gdb 调试 gcc，并给 gcc 传递命令行参数（见 4.3 程序的参数一节）“-O2 -c foo.c”。

可以通过指定-silent 选项，运行 gdb 而不打印前面的资料，这些资料说明 GDB 不作任何担保。

gdb –silent

你可以通过设定命令行选项，更多地控制 GDB 的启动。GDB 自身就能够给你各种可用选项的提示。

输入

gdb –help

显示所有可用选项，并对它们的用法作了简短的描述（可以简写成 gdb –h，作用相同）。

所有的选项和命令行参数按照你所给的顺序进行处理。当使用“-x”选项时，顺序就就比较重要了。

2.1.1 选择文件

在 GDB 启动时，除了选项之外，它把所有的参数都看作是可执行文件和 core 文件（或者是进程 ID）来读取。就如同这些参数已分别被“-se”和“-c”（或者是“-p”）选项所修

饰过一样。（GDB 读取的第一个参数，如果没有关联的选项标志，就等同于“-se”选项后面参数；如果有第二个这样的参数的话，就等同于“-c”/“-p”选项后的参数）如果第二个参数是以一个 10 进制数开头的话，GDB 首先会尝试把它作为一个进程去附着，如果失败了的话，就尝试着把它作为 core 文件打开。如果有一个文件名以数字开头的 core 文件的话，可用通过附加./前缀来避免 GDB 把它当成 pid，如./12345。

如果 GDB 已经被配置为不支持 core 文件，比如大部分的嵌入式目标，它将拒绝第二个参数而忽略它。

大部分的选项都有长格式和短格式；都被展示在下表中。GDB 也能识别不完整的长格式，只要与给出的各个选项之间没有歧义就行。（如果你愿意，你可以使用“--”来标记选项参数，而不用“-”，虽然我们展示的是更常用的约定）。

-symbols *file*
-s *file*
从文件 *file* 中读取符号表。

-exec *file*
-e *file*
把文件 ***file*** 当作可执行文件来使用，在适当的时候执行，连同 core dump，分析单纯的数据。

-se ***file***
从文件 ***file*** 中读取符号表，并把它当作可执行文件来使用。

-core *file*
-c *file*
把文件 *file* 当成 core 文件使用，进行分析。

-c *number*
-pid *number*
-p *number*
同附着命令一样，连接到一个进程 ID *number*。如果没有这个进程，GDB 就尝试着打开一个名为 *number* 的 core 文件。

-command *file*
-x *file*

执行文件 *file* 中的 GDB 命令。参见命令文件一章

-directory *directory*
-d *directory*
添加 *directory* 到原文件搜索路径。

-m
-mapped
警告：这个选项依赖于操作系统的功能，并不被所有的操作系统所支持。
如果你的操作系统可以通过 mmap 系统调用使用内存映射文件（memory-mapped files），你就可以利用这个选项，让 GDB 把你程序的符号写到当前目录下的一个可重用文件（reusable file）中。如果你正调试程序是“/tmp/fred”，那么对应的符号文件就是“/tmp/fred.syms”。之后的 GDB 调试会话会注意这个文件的存在，并且快速的映射它的符号信息，而不需要从可执行程序的符号表中读取。
“.syms”文件特属于 GDB 所运行的主机。它保存着 GDB 内部符号表的精确映像。它不能被多个主机平台交叉共享。

-r
-readnow
立即读取每一个符号文件的整个符号表，不同于默认的根据需要而逐步读取。这使得启动会更慢，但之后的操作会更快。

为了创建一个包换完整符号信息的“.syms”文件，典型的做法是-mapped 与-readnow 联合使用（参见文件指定命令一章，有关“.syms”文件的资料）。一个简单的创建一个“.symes”文件以便以后使用的 GDB 调用是：

```
gdb -batch -nx -mapped -readnow programname
```

2.1.2 选择模式

可以以两种模式中的一种运行 GDB——批处理模式或单调模式。

-nx
-n
不执行在任何初始化文件中找到的命令。默认情况下，GDB会在处理完所有命令选项与参数之后执行这些文件里的命令。参见命令文件一章。

-quiet
-silent
-q

“单调”。不打印介绍性信息和版权信息。这些信息在批处理模式下也是不打印的。

-batch

以批处理模式运行。处理完所有由“-x”所指定的命令文件（如果不使用“-n”来约束，还有所有的初始化文件）后以状态 0 退出。在执行命令文件中的命令过程中，如果发生错误，则以非 0 状态退出。

GDB 作为一个过滤器运行时，批处理模式可能会有用，例如，在另外一台计算机上下载并运行一段程序；为了使这个更有用，消息

Program exited normally.

(只要程序是在 GDB 的控制下运行终止的，一般都会使这个结果)说明批处理模式下的运行是没有问题。

-nowindows
-nw

“无窗口”。倘若 GDB 伴随有一个图形用户界面（GUI），这时这个选项就告诉 GDB 仅使用命令行界面。如果没有 GUI 可用，这个选项也就没有作用。

-windows
-w

如果 GDB 含有一个 GUI 的话，那么这个选项就是请求：如果可能的话，就使用它。

-cd *directory*

用 *directory* 代替当前的目录，作为 GDB 的运行工作目录运行。

-fullname
-f

当 GDB 作为 GNU Emacs 的子进程运行时，设置这个选项。他告诉 GDB 以某一标准输出完整的文件名和行号，以便每次都能以大家公认方式显示栈结构（这也包括每次程序停止时）。这个公认的格式为：两个“\032”字符后面跟着一个文件名，行号和字符位置以颜色区分，外加一个换行。Emacs-to-GDB 的接口程序利用两个“\032”字符作为显示栈结构源代码的信号。

-epoch

当 GDB 作为 Epoch Emacs-GDB 接口的子进程运行时，设置这个选项。它告诉 GDB 修改它的打印程序，以便 Epoch 可以在分割窗口中显示表达式的值。

`-annotate` *level*

这个选项设置GDB内部的注释级别。它的作用和使用“set annotate *leave*”相同（参见25.GDB的注释一章）。注释级别控制着应有多少消息同GDB的提示符一起打印，这些信息有：表达式的值、源代码行数以及其他类型的一些输出。级别 0 是默认级别，级别 1 用于GDB作为Emacs的子进程时，级别 3 是最大量的在GDB控制下的程序的相关信息，级别 2 现在已经不被建议使用了。

这个注释机制在很大程度上已被GDB/MI所代替（参见24.GDB/IM接口一章）。

`--args`

改变对命令行的解释，以使可执行文件的参数可以被作为命令行参数传给它。这个选项阻止选项处理。

`-baud` *bps*
`-b` *bps*

设置任一用于 GDB 远程调试的串行接口的线速度。

`-l` *timeout*

设置任一用于 GDB 远程调试的通讯的超时值（以秒为单位）。

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

把 *device* 作为程序的标准输入输出运行。

`-tui`

启动时激活文本用户界面。文本用户界面在终端上管理几个文本窗口，显示源代码、汇编、寄存器和GDB的命令输出（参见GDB文本用户界面一章）。最为选择，通过调用“gdbtui”程序可以启动文本用户界面。如果你在Emacs中运行GDB，不要使用这个选项（参见在GNU Emacs下使用GDB一章）。

`-interpreter` *interp*

使用解释器interp与控制程序或设备一起作为接口。这个选项是想让与GDB通讯的

程序作为它的一个后端程序。参见命令解释器一章。

“--interpreter=mi”（或者“--interpreter=mi2“）让GDB使用其 6.0 版以后包含的GDB/MI接口（参见GDB/MI接口一章）。之前包含在GDB 5.3 中的GDB/MI接口使用“--interpreter=mi1”选择，但不赞成使用。早期的GDB/MI接口已经不提供支持了。

-write

打开可执行文件和Core文件，可读可写。这相当于GDB里面的“set write on”命令。（参见14.6 修补程序一节）

-statistics

这个选项让 GDB 在完成每个命令并返回到提示符后，打印有关时间和内存使用率的统计。

-version

这个选项让 GDB 打印它的版本号和“无担保”内容后退出。

2.1.3 启动期间，GDB做了什么

这里是 GDB 在会话启动期间做了什么的描述：

1. 按照命令行的规定，准备命令解释器（参见解释器一章）。
2. 在你的 HOME 目录中读取初始化文件(如果有)，这行那个文件里的所有命令。
3. 处理命令行选项和操作对象。
4. 读取并执行当前工作目录中初始化文件（如果有）中的命令。仅在工作目录与 HOME 目录不同时这样做。因此，可以拥有不止一个初始化文件。在 HOME 目录下的是一个普通的，另外一个，在调用 GDB 的目录下的，是专用于程序调试的。
5. 读取由“-x”选项所指定的命令文件。要获得有关GDB命令文件的详细信息，请参考20.3 命令文件一节。

6. 读取历史文件中的命令历史记录。要获得有关命令历史以及GDB用于记录它的文件的详细信息，请参考19.3 命令历史一节。

初始化文件与命令文件使用相同的语法（见20.3 命令文件一节），而且在GDB中也是以相同的方法处理。在HOME目录下的初始化文件可以设置那些能够影响后续的命令行选项处理的选项（比如“set complaints”）。如果使用了“-nx”选项（参见选择模式一节），初始化文件就不会被执行。

GDB 的初始化文件通常被命名为“.gdbinit”。在某些 GDB 的配置中，初始化文件可以是其它的名字（有些典型的环形，有一个 GDB 的专有形式需要与其他的形式并存，因此，给专有版本的初始文件一个不同的名字）。这些使用特殊初始化文件名的环境有：

- GDB 的 DJGPP 移植使用的是“gdb.ini”，这是由于受到 DOS 文件系统对文件名的强制限制。GDB 的 Windows 移植采用的是标准名，而且一旦发现“gdb.ini”文件，它会发出警告，建议把这个文件更改为标准名。
- VxWorks(风河 Systems 的实时操作系统)：“.vxgdbinit”
- OS68K（Enea Data Systems 的实时操作系统）：“.os68gdbinit”
- ES-1800（爱立信电信 AB M68000 仿真器）：“.esgdbinit”
- CISCO 68k：“.cisco-gdbinit”

2.2 退出GDB

quit [*expression*]
q

使用 quit（缩写 q）命令退出 GDB，或者键入一个文件结束符（通常是 C-d）。如果你不提供 *expression*，GDB 会正常的结束；否则，它会以 *expression* 的结果作为错误代码结束。

中断（通常是 C-c）不会使 GDB 退出，而是终止了在进程中所有 GDB 命令的动作，并返回到 GDB 的命令层。在任何时候键入中断符都是安全的，因为 GDB 在不安全的时候是不会使它生效的。

如果你曾用GDB控制一个被附着的进程或设备的话，你可以使用detach命令释放它（参见调试一个早以运行的进程一章）。

2.3 Shell命令

要在调试会话中需要执行一个应时的 shell 命令，是不需要离开或挂起 GDB 的；只需使用“shell”命令即可。

shell command string

调用标准 shell，执行 command string。倘若有环境变量 SHELL，那么它就决定应该运行哪个 shell。否则，GDB 就使用默认的 shell（UNIX 系统是/bin/sh，MS-DOS 是 COMMAND.COM，等等）。

make 工具一般是开发环境所需要的。最好不要在 GDB 中使用它。

make make-args

以指定的参数执行 make 程序。等同于“shell make *make-args*”。

2.4 Loging输出

你可能希望将 GDB 的命令输出保存到一个文件中去。这里有几个指令控制 GDB 的 Loging。

set logging on

开启 logging。

set logging off

关闭 logging。

set logging file *file*

更改当前 logfile 的名字。默认的 logfile 是“gdb.txt”。

set logging overwrite [on|off]

通常，GDB 是追加 logfile。如果设置 overwrite 为 on 的话，GDB 覆盖 logfile。

set logging redirect [on|off]

通常，GDB 同时向终端和 logfile 输出。要是想让 GDB 只输入到 logfile，就设置 redirect 为 on。

show logging

显示当前的 logging 设置值。

3. GDB命令

可以使用命令名的头几个字母简写 GDB 的命令，只要这个简写不会产生二义性；还以通过键入 RET 重复某些命令。也可用利用 TAB 键，让 GDB 自己填写命令单词中剩余的部分（或者是可用的备选方案，只要有多于 1 个的可能性）。

3.1 命令语法

一个 GDB 命令是一个单行输入，对于长度没有限制。以命令名开始，后跟一些参数，这些参数的意义取决于命令名。例如，step 命令，它接受一个表明步进次数的参数，如在“step 5”中。step 命令也可以不带参数。有些命令根本就不允许有参数。

GDB的命令总是可以被缩短的，只要这个简写不会产生二义性。其他可能的命令简写都已被列在个专用令文档中了。在某些情况下，甚至二义性的简写也是被允许的；例如：s就是被特别定义为等同于step的，即使还有其他的一些名字以s开头的命令。可以把它们作为help命令的参数来测试一个简写命令。向GDB输入一个空白行（仅键入RET）意味着重复先前的命令。这种方法是不能重复某些命令的（比如run）。有些命令，不小心的重复可能会导致一些问题，况且你也不太可能要重复。自定义命令可以关闭这一特性，见禁止重复一节。

list 和 x 命令，当你使用 RET 重复它们时，构造新的参数，而不是原来的。这允许轻松地扫描源代码或内存。

GDB也可以以另外一种方式使用RET：以一种类似通用工具more的方式，分屏冗长的输出（见屏幕尺寸一章）。由于在这种情形下，按一次RET后容易产生太多的重复，因此，GDB在任何能够产生那种显示的命令以后都关闭了命令重复。

任何从一个#到行尾的文本都是一个注释；它什么也不做；它主要用在命令文件中（见命令文件一章）。

C-o 组合有重复一个复杂命令序列的作用。这个命令接受当前行，如同 RET，然后从编辑历史记录中读取与当前行相关的下一行。

3.2 命令完成

如果只有一种可能性的话，GDB 可以为你填写命令单词中剩余的部分；也可以把所有有效的可能性的命令单词显示给你，而且任何时候都可以。这个功能对 GDB 命令，GDB 子命令以及程序中的符号名起作用。

只要你想让 GDB 填写一个词的剩余的部分，按 TAB 键即可。如果只有一种可能性，GDB 在填写完后就等待你去完成这个命令（或者说按 RET 输入它）。例如，如果你键入：

（gdb） info bre TAB

GDB 填写“breakpoints”单词的剩余部分，因为 info 的子命令以 bre 开头的只有它：

（gdb） info breakpoints

既可以在这个位置按下 RET 运行这个 info breakpoints 命令，也可以按下控各，输入其他的东西，如果“info breakpoints”不是你所期待的命令的话（如果“info breakpoints”确实就是你想要的，还是在“info bre”后立即按下 RET 为好，与其使用命令完成，还不如使用命令缩写）。

当你按 TAB 时，如果对于下一个单词有多于一个的可能性的话，GDB 会发出一个蜂鸣。你既可以补充更多的字母然后重试，也可以在按一下 TAB，GDB 显示对于那个单词的所有可能的完成。例如，你可能想在一个名字以“make_”开头的子程序中设置一个断点，可是当你键入 b make_TAB 时，GDB 只是发出了一个蜂鸣。再按一次 TAB，显示出程序中所有以这些字母开头的子程序名。

(gdb) b make_ TAB

GDB 发出蜂鸣；再按一次 TAB，看到:

```
make_a_section_from_file     make_environ
make_abs_section             make_function_type
make_blockvector             make_pointer_type
make_cleanup                 make_reference_type
```

```
make_command                    make_symbol_completion_list
(gdb) b make_
```

显示完可用的可能性后，GDB 复制你的部分输入（例子中是“b make_”），便于你完成命令。

如果先前你只是想看一下备选方案列表，与其按 TAB 两次，还不如按 M-?。M-？意思是 META ?。即可以通过按住键盘上代表 META 的键（如果有，译者注：在 PC 上一般是 ALT），再按下？输入，也可以按住 ESC 后再按？输入。

有些时候，你需要的字符串，虽然逻辑上是一个“词”，但可能包含圆括号或者是其他的字符，被 GDB 正常地从它的词的概念上排除了。为了“词完成”可以在这种情况下工作，可以在 GDB 命令中用’（单引号）把这些词包围起来。

这最适合在键入 C++函数名时使用这一功能。这是因为 C++允许函数重载（相同的函数有多个定义，以参数类型来区分）。例如，当你想要设置一个断点时，你可能需要区分是否是你想要的那个带有一个 int 型参数的 name 版本 name(int)，还是另外一个带有一个 float 型参数的 name 版本 name(float)。为了在这种情况下使用“词完成”功能。在函数名的开头敲一个单引号’。这通知 GDB，当按下 TAB 或 M-?请求“词完成”时，需要考虑比平时要多的信息。

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

在某些情况下，GDB 能够知道正完成的名字需要使用引号。当这发生时，如果你先前没有输入引号，GDB 会为你插入引号（尽其所能地）。

```
(gdb) b bub TAB
```

GDB 通知你，你的输入行变成下面这样，并发出蜂鸣：

```
(gdb) b 'bubble(
```

通常，当你请求一个关于重载符号的“完成”时，如果你还没有敲入参数表，GDB 都

会知道那需要一个引号。

要获得更多有关重载函数的信息，请参见C++表达式。可以使用命令set overload-resolution off 关闭重载转换。参见适用于C++的GDB特性。

3.3 获得帮助

总是可以向 GDB 本身询问有关命令的信息，使用命令 help。

help
h

可以使用没有参数的 help（简写 h）显示一个简短的命令分类的命名列表。

```
(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*

使用一个分类作为参数，可以获得这个分类中的单独命令的列表。这是一个分类

的帮助显示情况。

```
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*

以一个命令名作为帮助的参数。GDB 显示一个简单的如何使用这个命令的短评。

apropos *args*

apropos 命令详细地 GDB 的所有命令以及它们的文档进行搜索，并与在 *args* 指定的正则表达式进行比配，将所有比配到的信息打印输出。例如：

apropos reload

结果是：

```
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*

complete *args* 命令列出所有适合于某一个命令开头可能的“完成”。使用 *args* 指定你想要“完成”的命令开头。例如：

complete i

结果是：

if
ignore
info
inspect

这个打算由 GNU Emacs 使用。

除了help之外，可以使用GDB命令info和show了解关于程序的状况，或者是GDB本身的状况。每一个命令都支持很多的询问主题。本手册会在适当的情况下一一介绍它们。在索引中，info和show下的列表，指向了所有的子命令。参见索引一章。

info

这个命令（简写 i）适用于描述程序的状况。例如：info args 可以列出赋给程序的参数，info registers 列出当前使用的寄存器，info breakpoints 可以列出已设置的断点。使用 help info 可以获得一个有关 info 子命令的完整列表。

set

使用 set 可以把表达式的结果设置给环境变量。例如：用 set prompt $ 把 GDB 的提示符设置为$符。

show

与 info 正好相反，它显示的是 GDB 本身的状况。利用相关的命令 set，可以更改大部分可以显示的东西。例如：使用 set radix 控制使用什么数制用于显示，或者使用 show radix 简单地询问当前所使用的数制。

为了显示所有可置位的参数以及它们的值，可以使用不带参数的 show，也可以使用 info set。两个命令产生相同的显示。

这里有三个其他的 show 子命令，它们特别的是都没有对应的 set 命令：

show version

显示正在运行的 GDB 的版本。应该在 GDB 的 bug 报告中包含这个信息。如果你

的地方用了多个版本的 GDB，你可能需要确定你正在运行的是哪一个版本。由于 GDB 的进展，新命令的引用，旧命令可能已经幻灭。同样，好多系统供应商配备有不同的 GDB 版本，而且这些不同的 GDB 版本也是发布在 GNU/Linux 中。在启动 GDB 时，版本号同样通告一次。

show copy

info copy

显示 GDB 的复制许可信息。

show warranty

info warranty

显示 GNU“无担保”信息，或者一个担保，要是你的 GDB 版本有一个的话。

4. 在GDB下运行程序

在 GDB 下运行程序时，首先必须在编译时生成调试信息。

在你选择的一个环境中，如果有的话，你可以使用参数启动 GDB。如果进行的是本地调试，可以重定向程序的输入输出，调试某个已经运行的进程，或者杀掉某个子进程。

4.1 适合调试的编译

为了有效的调试某个程序，需要在编译的时候生成调试信息。这个调试信息被存储在目标文件中；它描述了各个变量或函数的数据类型，以及可执行代码与源代码行号之间的对应关系。

在运行编译器时，指定“-g”选项，要求编译器产生调试信息。配备给客户的程序是使用最优化编译的，使用的是“-O”选项。可是，好多编译器不能把“-g”和“-O”选项放在一起处理。使用这些编译器，不能产生包含编译信息的最优化可执行代码。

GCC，GNU C/C++编译器，支持附带或不附带“-O”的“-g”，这使得它能够调试优化后的代码。我们建议，只要编译程序，就使用“-g”。你可能认为你的程序是正确的，但不要期望好运会持续。

在调试某个使用“-g –O”编译的程序时，要想到优化程序已经重排了代码，调试器显示给你的是真实的代码。执行路径与源文件不符时，不要太惊讶。一个极端的例子：假如你

定义了一个变量，但是从来没有使用过，GDB 根本就看不到到那个变量——因为编译器已经把他优化掉了（译者注：不存在了）。

仅使用“-g”，某些东西跟使用“-g –O”工作的不一样，特别是在有指令调度的机器上。要是不信的话，就单独使用“-g”重新编译，这要是校准这个问题，请把他作为一个bug发送给我们（包括测试条件）。要得到更多有关调试优化代码的信息，参见8.2 程序变量一章。

老版本的 GCC 编译器允许有一个不同的选项“-gg”用于生成调试信息。GDB 现已不再支持这个格式。要是你的 GCC 有这个选项，请不要使用。

GDB了解有关的预处理宏，而且可以把它展开后显示给你（参见9.C的预处理宏一章）。单独指定“-g”标志，大部分编译器不会在调试信息中包含有关预处理宏的信息，因为这些信息是相当庞大的。3.1 及其以后版本的GCC中的GNU C编译器，如果指定了“-gdwarf-2”和“-g3”选项，可以提供宏的信息。前一个选项要求产生Dwarf 2 格式的调试信息，后一个选项要求产生“非常信息（extra information）”。今后，我们希望找到些更紧凑的方式来描述宏信息，以使它单独使用“-g”就可以被包含。

4.2 启动程序

run
r

利用run命令在GDB下启动程序。首先必须使用一个参数给GDB指定程序的名称（除非是在VxWorks上），或者使用file或exe-file命令指定（见指定文件的命令一节）

如果在一个支持进程的环境下运行程序，run 创建一个次级进程，让这个进程来运行程序。（在没有进程的环境下，run 跳转的程序的开始处）。

某个程序的执行，受到从它上级接收到的某些信息的影响。GDB 提供了指定这些信息的方式，但必须在程序启动之前指定。（虽然能够在程序启动之后更改它，但是所作的更改只能在程序下次启动后才有作用。）这些信息可以被划分为四类：

参数

把参数最为运行命令参数指定给程序。如果在你的目标上有一个shell可用，那么这个shell通常用于传递这个参数，以便于可以利用常规描述参数的约定（如通配符展开或变量置换）。在UNIX系统中，可以利用SHELL环境变量控制使用哪一个Shell。参见程序的参数一节。

环境

程序一般从GDB继承它的环境，但是也可以使用GDB命令set environment与unset environment更改某些影响程序的局部环境。参见程序的环境一节。

工作目录

程序从GDB继承它的工作目录，在GDB中可以使用cd命令设置GDB的工作目录。参见程序的工作目录一节。

标准输入输出

程序一般与GDB所使用的相同的设备作为标准输入输出。可以在run的命令行里从定向输入输出，或者使用tty命令给程序设定一个不同的设备。见程序的输入输出一节。

警告：当输入输出从定向生效时，不能使用管道传递正调试的程序的输入给另外一个程序。试图这么做的话，GDB 可能终止调试错误的程序。

下达run命令时，程序会立即开始执行。有关讨论怎样安排程序的停止，见停止与继续一节。一旦程序已经停止，就可以使用print或call命令调用程序中的函数。参见检验数据一章。

自上一次 GDB 读取符号后，符号文件的修改时间已更改的话，GDB 会重读它。当做这件事的时候，GDB 会努力保持当前的断点。

start

不同的语言，主过程名也不相同。C 或 C++的主过程永远是 main，但是像 Ada 就不要求为它们的主过程起个特殊的名字。调试器提供了一个方便的方法开始程序的执行，并在主过程的起始位置停住，这依赖于所使用的语言。

“start”命令相当于在程序主过程的起始位置设置一个临时断点之后调用“run”命令。

有些程序会包含一个细化（*elaboration*）阶段，有些代码在主过程被调用之前执行。这取决于编写程序的具体语言。例如，在 C++中，静态的与全局的对象的构造函数会在 main 被调用之前执行。这使得调试器在达到主过程之前停止程序成为可能。不管怎样，临时断点仍然会停止程序的执行。

程序的参数可以作为“start”命令的参数指定给程序。这些参数会精确地传递给次级的“run”命令。要注意，后面调用的“start”或“run”没有提供给参数的话，会重用前面提供的那个参数。

在细化期间调试程序，有些时候是必要的。在这种情况下，使用 start 命令对于停止程序的执行来说，那太晚了，因为这时程序早已完成了细化阶段。在这种情况下，在运行程序之前在细化代码中插入断点。

4.3 程序的参数

程序的参数可由“run”命令的参数指定。它们被传递给一个 shell，展开通配符并重定向标 I/O，最后传递给程序。SEHLL 环境变量（如果有）规定了 GDB 使用什么 shell。如果没有定义 SHELL，GDB 会使用默认的 shell（在 UNIX 上是“/bin/sh”）。

在非 UNIX 系统上，程序一般由 GDB 直接调用，利用适当的系统调用来模拟 I/O 重定向，同时，通配符由程序的启动代码展开，而不是 shell。

没有参数的 run 通常使用前一个 run 的参数，或者是 set args 命令指定的那些。

set args

指定下次程序运行用到的参数。如果 set args 没有参数，run 不传递参数而执行程序。一旦使用参数 run 了程序，那么在下次 run 程序之前使用 set args，是再次不使用参数 run 程序的唯一方法。

show args

显示启动时传给程序的参数。

4.4 程序的环境

“环境”由一组环境变量和它们对应的值构成。环境变量通常记录着如：用户名、HOME 目录、终端类型以及程序运行的搜索目录这些信息。通常你用 shell 设置的环境变量会被所有你运行的其他程序所继承。调试时，不需要反复启动 GDB 就可以使用一个修改的环境变

量试着运行程序，是很有用的。

path *directory*

在传递给程序的 PATH 环境变量（可执行程序的搜索路径）前，添加 *directory*。GDB 所使用的 PATH 值不会被更改。可以指定若干的目录名，使用空格或系统相关的分隔符（UNIX 上是“:”，MS-DOS 和 MS-Windows 上是“;”）分隔它们。如果 *directory* 已经在 PATH 中了，就把它移到前面，以使它可以被立即搜索到。

可以使用字符串“$cwd”，在 GDB 搜索路径时，引用当前工作目录。如果使用“.”代替的话，它引用的是由执行 path 命令指定的目录。在添加 *directory* 到搜索路径之前，GDB 在 *directory* 参数中替换“.”（使用当前路径）。

show path

显示可执行文件的搜索路径列表（PATH 环境变量）。

show environment [*varname*]

打印程序启动时惯于它的环境变量 *varname* 的值。要是没有给出 *varname*，就打印所有惯于程序的环境变量的名和它的值。可以用 env 简写 environment。

set environment *varname* [=value]

给环境变量 *varname* 赋值为 *value*。变量值的更改仅针对于程序，不会影响 GDB 本身。*value* 可以是任何字符串——环境变量值只能是字符串，对它的任何解释都由程序本身提供。*value* 参数是可选的。如果去除的话，变量会被赋予一个 null 值。例如这个命令：

set env USER = foo

告诉 GDB，在随后的 run 时，它的用户是一个名叫“foo”的（“=”前后使用的空格是为了清晰，实际上是不需要的）。

unset environment *varname*

从传递给程序的环境中移除变量 *varname*。这与“set environment varname=”不同，它是从环境中移出变量，而不是给它赋一个空值。

警告：在 UNIX 系统中，如果有 SHELL 环境变量的话，GDB 使用的是它指定的 shell（没有的话使用/bin/sh）。如果你的 SHELL 环境变量指定了一个运行初始化文件（如 C-Shell 的“.cshrc”或者是 BASH 的“.bashrc”）的 shell，你设置在那个文件中的任何变量，都会对你的程序有所影响。你可能希望把这些环境变量的设置移到仅在你注册时运行的文件中去，这样的文件有“.login”或“.profile”。

4.5 程序的工作目录

每次使用 run 启动的程序，都 GDB 的当前工作目录继承为它的工作目录。GDB 的工作目录最初是继承自它的父进程（通常是 shell），不过，你可以在 GDB 中使用 cd 命令指定一个新的工作目录。

GDB的工作目录也担当着GDB操作的指定文件命令的默认目录。参见指定文件的命令一节。

cd *directory*

设置 GDB 的工作目录为 *directory*。

pwd

打印 GDB 的当前工作目录。

找到正被调试的进程的当前工作目录，通常是做不到的（因为一个程序可以在它运行的时候改变它的目录）。如果你工作在GDB可以被配置为带有“/proc”支持的系统上的话，你可以利用info proc命令（见18.1.3 SVR4 进程信息一节）找到debuggee的当前工作目录。

4.6 程序的输入输出

默认情况下，GDB 下运行的程序作输入输出的终端，与 GDB 所使用的是相同的。GDB 把终端转换为它自己的终端方式与你交互，不过，它记录你的应用程序所使用的终端方式，当你继续运行你的程序时，它再切换回来。

info terminal

显示由 GDB 记录下来的程序所使用的终端模式的信息。

可以使用 run 命令，利用 shell 的重定向，重定向程序的输入和/或输出。

run > outfile

启动程序，驱使他的输出到文件“outfile”。

另外一种指定程序在何处做输入输出的方式是使用 tty 命令。这个命令接受一个作为参数的文件名，并使这个文件成为之后 run 命令的默认重定向目标。它也为子进程、之后的 run 命令重置控制终端。例如：

tty /dev/ttyb

指示后续用 run 命令启动的进程默认在终端“/dev/ttyb”上作输入输出，并且拿这个作为它们的控制终端。

在 run 中明确的重定向，优先于 tty 对输入/输出设备的影响，但是并不对控制终端有影响。

当使用 tty 命令或在 run 命令中重定向输入时，只有程序的输入会受到影响。GDB 的输入依然来自你的终端。tty 是 set inferior-tty 的别名。

可以使用 show inferior-tty 命令告诉 GDB，显示将来程序运行会被使用的终端的名字。

set inferior-tty /dev/ttyb

设置正被调试的程序的 tty 为/dev/ttyb。

show inferior-tty

显示正被调试的程序的当前 tty。

4.7 调试某个已运行的进程

attached *process-id*

这个命令附着到一个正在 GDB 以外运行的进程。（info 文件显示活动目标）这个命令带有一个 *process-id* 参数。要找到某个 UNIX 进程的 *process-id*，通常的方式是使用 ps 工具，或者使用“jobs –l”shell 命令。

attached 命令执行之后，第二次按 RET 的话，是不会被重复的。

为了使用 attached，程序必须运行在一个支持进程的环境下。例如：对于在缺乏操作的 bare-board 目标上的程序，attached 是不会工作的。也必须得有发送信号的权限。

当使用attached的时候，调试器首先定位当前目录中正运行在进程中的程序，然后（如果没有发现这个程序）查看元文件搜索路径（参见指定源文件目录一节）。你也可以利用file命令装载程序。参见指定文件的命令一节。

准备好要调试的指定进程后，所做的第一件事就是停止进程。检查与修改被附着的进程，可以使用平常使用 run 启动进程时能够用到的所有命令。可以插入断点；可以单步调试并继续；可以修改存储器。如果你愿意让进程继续运行的话，你可以在将 GDB 附着到进程之后使用 continue 命令。

detach

当对被附着的进程的调试完成时，可以使用 detach 命令，把它从 GDB 的控制下释放出来。分离进程而继续执行。detach 命令之后，那个进程与 GDB 再一次变得完全独立，并且准备附着另外一个进程，或者使用 run 命令启动一个。detach 命令执行之后，再按 RET 的话，是不会被重复的。

当已附着到一个进程时，退出GDB或使用run命令，会杀掉那个进程。默认情况下，GDB

会询是否尝试这么做的确认，可以使用set confirm命令控制是否需要确认。（参见可选的警告与消息一节）。

4.8 杀掉子进程

kill 杀掉运行在 GDB 下的程序的子进程。

这个命令在你想要调试一个 core dump，而不是一个进程时很有用。程序运行时，GDB 会忽略所有的 core dump。

在某些操作系统上，当你在 GDB 中给某个程序设置了断点，那么这个程序就不能在 GDB 以外被执行了。可以在这里使用 kill 命令，准许在 GDB 以外运行程序。

kill 在想要重新编译或重新连接程序的时候也很有用，因为在大多数系统上是不允许修改正在运行的进程的可执行文件的。在这种情况下，当你下次敲入 run 时，GDB 会通知那个文件已经被修改了，并重新读取符号文件（会尽量维持当前的断点设置）。

4.9 多线程程序的调试

在某些操作系统中，例如 HP-UX 和 Solaris，一个单独的程序可能拥有不止一个线程。一个操作系统对于另外一个操作系统，现程的精确语义是不相同的，但是大体上，单个程序的线程，除了共享同一个地址空间之外（更确切地说，它们都可以访问并修改同一个变量），与多进程近似。从另外一个角度讲，每一个线程拥有自己的寄存器和执行栈，而且也可能有自己专用的存储器。

GDB 为调试多线程程序提供了如下功能：

- 新线程的自动通知。
- “thread *threadno*”，在线程间进行切换的命令。
- “info threads”，查询现有线程的命令。
- “thread apply [*threadno*] [*all*] *args*”，让一列线程应用某个命令的命令
- 线程特有的断点。

警告：这些功能并不是在每一个操作系统支持线程的 GDB 配置中都可用。你的 GDB 不支持线程的话，这些命令是不会起作用的。例如，有一个不支持线程的系统，来自“info

threads”命令的输入没有显示，而且总是拒线程命令，就像下面这样：

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

GDB 的线程调试工具可以让你观察运行在你的程序中的所有线程——有个特殊的线程总会是调试的焦点所在，而且任何时候 GDB 都要获得对它的控制权。这个线程就被称作当前线程。调试命令透过对当前线程的观察来显示程序的信息。

只要 GDB 检测到程序中有新线程，它就使用“[New *systag*]”格式的一个消息显示目标系统对于这个线程的标识。*systag* 是一个线程表示符，各式会因具体的系统而不同。例如，在 LynxOS 上，当 GDB 通知有一个新线程时，你看到的可能会是这样的消息：

```
[new process 35 thread 27]
```

相对的，在一个 SGI 系统上，*systag* 就很简单，如“process 3 68”，没有过多的限定词。

为了调试的目的，GDB 使用它自己的线程号——总是一个单独的整数——对应于程序中的各个线程。

info thread

显示目前程序中所有线程的一个摘要。GDB 对于每一个线程的显示（按这种顺序）：

1. 由 GDB 分配的线程号
2. 目标系统的线程标识符（*systag*）
3. 对应线程当前的堆栈帧的摘要信息

GDB 线程号左边有一个“*”号，表明这是当前线程。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 系统上：

为了调试的目的，GDB 使用它自己的线程号——按照线程创建顺序分配的一个小整数——对应于程序中的各个线程。

只要 GDB 检测到了程序中有新的线程，它就使用“[New *systag*]”格式的一个消息显示目标系统对于这个线程的标识。*systag* 是一个线程表示符，各式会因具体的系统而不同。例如，在 HP-UX 系统上，当 GDB 通知有一个新线程时，你看到的可能会是这样的消息：

[New thread 2 (system thread 26594)]

info thread

显示目前程序中所有线程的一个摘要。GDB 对于每一个线程的显示（按这种顺序）：

1. 由 GDB 分配的线程号
2. 目标系统的线程标识符（*systag*）
3. 对应线程当前的堆栈帧的摘要信息

GDB 线程号左边有一个“*”号，表明这是当前线程。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 系统上，使用一个 Solaris 专有命令，能够显示更多有关用户线程的信息：

maint info sol-threads

显示 Solaris 上的用户线程信息。

thread *threadno*

使线程号为 *threadno* 线程成为当前线程。这个命令的参数 *threadno* 是 GDB 内部线程编号，也就是上述“info thread”所显示的那个。GDB 以显示你所选择的线程的标识符，以及它目前的堆栈帧的摘要作为应答：

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

同样带有“[New …]”消息，Switching to 后面的文字格式依赖于你的操作系统所规定的线程标识。

thread apply [*threadno*] [*all*] *args*

thread apply 命令使你可以给一个或多个线程应用某个命令。利用命令参数 *threadno* 指定你想要影响的线程的编号。*threadno* 是 GDB 内部线程编号，也就是上述“info thread”所显示的那个。要对所有线程应用某个命令，使用 thread apply all *args*。

只要 GDB 停止了你的程序，不管是因为某个断点还是某个信号，它都会自动地选择那个发生断点或信号的线程。GDB 用一个“[Switching to systag]”格式消息提醒你注意上下文切换，鉴别线程。

参见停止与启动多线程程序一节，了解更多有关停止和启动某个带有多个线程的程序时，GDB是如果工作的信息。

参见设置观察点一节，了解有关带有多个线程的程序中的观察点信息。

4.10 多进程程序的调试

在大多数系统上，GDB 没有对调试那些利用 fork 函数创建额外进程的程序提供特殊支持。当某个程序 forks 时，GDB 会继续调试父进程，而不会对子进程的运行有所妨碍。如果你已在子进程能够执行到的任意代码中设置了一个断点话，那么子进程将会获得一个 SIGTRAP 信号，这（除非它获得了这个信号）会导致它的中断。

不管怎样，你要是想调试子进程，还是有一个不算太费力的工作区的。在子进程fork之后要执行的代码中放置一个sleep调用。只有当某个环境变量被设置了，或者已存在某个文件，那也许会对sleep有作用，所以，你不想在子进程上运行GDB时，那个延迟就没有必要发生。当子进程正睡眠时，利用ps程序获得它的进程ID。然后告诉GDB（要是还想调试父

进程的话，就启动一个新的GDB）附着到这个子进程上（参见4.7 调试某个已运行的进程一节）。从那一点开始，你就可以像调试其他附着到的进程一样，调试这个子进程了。

在某些系统上，GDB 对那些利用 fork 或者是 vfork 函数创建额外进程的程序提供了支持。目前，仅有的带有这一特性的几个平台是：HP-UX（仅 11.x 及后续版本？）和 GNU/Linux（2.5.60 及后续版本）。

一般情况下，当程序 forks 时，GDB 会继续调试父进程，而不会对子进程的运行有所妨碍。

如果你想跟随子进程而不是父进程的话，就使用命令 set follow-fork-mode。

set follow-fork-mode

设置调试器响应程序对 fork 或 vfork 的调用。一个 fork 或 vfork 的调用创建一个新的进程。*mode* 参数可以是：

parent

fork 之后的原有进程被调试。子进程的运行不受妨碍。这也是默认的。

child

fork 之后的新进程被调试。父进程的运行不受妨碍。

show follow-fork-mode

显示调试器目前对 fork 或 vfork 调用的响应。

假如你要求调试子进程，同时 vfork 之后又跟着一个 exec 的话，GDB 执行那个新的目标直到遇到目标中的第一个断点。如果在原有程序的 main 中有一个断点的话，那么这个断点也会被设置在子进程的 main 中。

当子进程被 vfork 产生时，直到 exec 调用完成之前，既不能对子进程调试也不能对父进程调试。

如果在exec调用执行之后，你对GDB发出了一个run命令，这个新的目标重新启动。要重新启动父进程，须利用以父进程可执行文件名为参数的file命令。可以使用catch命令，只要有fork，vfork或exec调用产生，就让GDB停止。参见设置捕获点一节。

5.0 停止与继续

使用调试器的首要目的是为了在程序终止之前停止程序；抑或为了程序运行有问题时，可以探查并寻找原因。

在 GDB 内部，程序可以因为多种因素而停止，如信号、断点，或者是使用 step 这样的命令后到达了新的一行。然后可以查看和更改变量，设置新的断点或去除老的断点，然后继续执行。通常，GDB 所显示的消息对程序的状态都提供了充分的说明——也可以在任时候显示的请求这些信息。

info program

显示有关程序状态的信息：是否在运行、它的进程是什么，还有他为什么停止了。

5.1 断点、观察点和捕捉点

只要到达了程序中的某一断点，程序就会停止。对于每一个断点，都可以附加一些条件，在细节上控制程序的停止与否。可以利用break命令设置断点和它的变体（参见设置断点一节），用行号、函数名或者是程序中的确切地址，指定程序应在何处停止。

在有些系统上，在可执行文件运行以前，就可以在共享库中设置断点。在 HP-UX 系统上会有一点限制：要想在那些不被程序直接调用的共享库例程（例如，作为 pthread_create 调用的参数的例程）中设置断点，必须等到可执行文件运行才行。

观察点是一种特殊的断点，当某个表达式的值改变时，停止程序。必须使用一个不同的命令来设置观察点（参见设置观察点一节），但除此以外，就可以向管理断点那样管理观察点：可以利用相同的命令启用，停用和删除观察点和断点。

可以安排程序中的某些值，只要GDB停在了某个断点上，就可以自动地被显示出来（参

见自动显示一节）。

捕获点是另外一种特殊的断点，当有某些事件发生时，停止程序，例如抛出C++异常，或者某个库的加载。与观察点一样，使用不同的命令设置捕获点，但除此以外，可以像管理断点那样管理捕获点。(要使程序在接收到某个信号时停止，使用handle命令;参见信号一节）。

当创建一个断点、观察点或者捕获点时，GDB 会为他们分配一个编号；这些编号是从 1 开始的连续整数。在许多的控制断点不同特性的命令中，使用编号说明要改变哪一个断点。要是被停用了，直到再次启用前是不会对程序有任何影响的。

有些 GDB 命令可以接受某一范围的断点，对其操作。一个断点范围即可以使一个单独的编号，像“5”，也可以是两个编号，递增顺序，中间用横线连接，像“5-7”。当某个断点范围指定给了某个命令时，在那范围以内的所有断点都会受到影响。

5.1.1 设置断点

断点由break（简写b）命令设置。调试器便利变量（debugger convenience variable）“$bpnum”记录了最近设置的断点的数量；参见便利变量一节，讨论了使用便利变量都可以做什么。

有多种方法说明断点的去向：

```
break function
```

在函数*function*的入口处设置断点。当使用允许符号重载的源语言时，如C++，*function*可能会涉及到多个可能的中断位置。参见断点菜单一节，对于这种情形的讨论。

break +*offset*

break -*offset*

从当前选取的堆栈帧的执行停止处，向前或向后若干行处设置一个断点。参见帧一节，对于堆栈帧的描述。

break *linenum*

在当前源文件的第 *linenum* 行处设置一个断点。当前源文件即最后那个被打印出源程序文本的文件。断点会正好在执行那以行的任何代码之前停止程序。

break *filename*:*linenum*

在源文件 *filename* 的 *linenum* 行处设置一个断点。

break *filename*:*function*

在文件 *filename* 中找到的函数 *function* 的入口处设置一个断点。即指定文件名又指定函数名是多余的，除非是有多个文件中包含了相同名称的函数。

break **address*

在地址 *address* 处设置一个断点。可以使用这个命令，在程序中那些没有调试信息或源文件的部分中设置断点。

break

当不使用任何参数调用的时候，break在被选取的堆栈帧内的下一个要执行的指令处设置一个断点（参见检查堆栈一节）。在被选取的任何堆栈帧内除最深处的以外，这会使得一旦控制回到那一帧，程序就会停止。这有些类似于被选择帧的内部帧内的finish命令的作用——除了finish还没有离开活动断点之外。要是在最内部的帧内使用不带参数的break的话，GDB会在下一次到达当前位置时停止。这或许

会对内部循环有用。

break ... if *cond*

设置一个带有条件*cond*的断点；每次达到断点时计算表达式*cond*的值，只有是非 0 值时才停止，即条件为“真”时。“...”代表上述的指定在什么位置中断的参数之一（或者无参数）。要获得更多有关断点条件的信息，参见中断条件一节。

tbreak *args*

设置仅允许停止一次的断点。*args*与break命令相同，而且断点设置的方法也相同，只不过程序首次在那里停止以后，这个断点就会被删除。参见停用断点一节。

hbreak *args*

设置一个硬件辅助的（hardware-assisted）断点。Args与break命令相同，而且断点设置的方法也相同，只不过断点需要硬件的支持，而且好多目标硬件可能没有这方面的支持。这个命令的主要用途是EPROM/ROM代码的调试，因为可以在一个指令上设置断点而不用更改这个指令。这个命令可以被用于带有由SPARClite DSU和基于x86 所提供的新型trap-generation（陷阱发生）的目标。这些目标会在程序访问某些已分配给调试寄存器的数据或指令地址时产生陷阱。不过，硬件断点寄存器只能持有有限数量的断点。例如，在DSU上，一次只能有两个数据断点被设置，同时，如果超过了两个，GDB会拒绝这个命令。在设置新断点之前，删除或停用不用的断点（参见停用一节）。参见中断条件一节。对于远程目标，可以限定GDB要使用的硬件断点数量，参见设置远程硬件断点限制。

thbreak *args*

设置一个仅允许停止一次的硬件辅助断点。Args与hbreak命令相同，而且使用方法也相同。不过，像tbreak命令一样，程序首次在那里停止后，这个断点就会被删除。同样，也像hbreak命令，需要硬件的支持，而且有些硬件可能没有这方面的支持。参见停用断点一节，及中断条件一节。

rbreak *regex*

在所有由正则表达式 *regex* 所匹配到的函数上设置断点。这个命令在所有匹配上设置一个无条件断点，打印它所设置的所有断点的一个列表。一旦设置了这些断点，它们仅仅会像由 break 命令设置的断点那样被处理。可以删除它们，停用它们，或者给它们限定条件，这些都跟操作所有其他断点的方法是一样的。

正则表达式的语法与“grep”这样的工具所使用的是同一标准。注意这与 shell 所使用的语法不同，例如：foo* 配置包含一个 fo 后面跟着 0 个或多个 o 的函数。你所提供的是一个隐含了.*开头与结尾的正则表达式，因此，仅匹配以 foo 开头的函数，使用^foo。