GDB 用戶手冊

目錄

摘要

象 GDB 這樣的調試程式，目的就是讓你可以查看其他程式的內部運行過程，或者是在它崩潰的那一時刻它在做什麼。

GDB 能做 4 件事（這些還需附加其他的一些事），幫助你捕獲在場的錯誤：

- 啟動程式，設定任何可以影響它行為的東西。
- 在特定的條件下使程式停止。
- 當程式停止時，分析發生了什麼。
- 改變程式裏的一些東西，進行一個由於 bug 所導致的結果的矯正性試驗，同時繼續瞭解另外一個 bug。

可以使用 GDB 調試用 C 和 C++編寫的程式，更多資訊參見支援的語言，及 C 與 C++。部分支援 Modula-2，Modula-2 的更多資訊參見 Modula-2。

在調試使用 sets、subranges、file variables 或嵌套函數的 Pascal 程式時，目前不能工作。GDB 不支援 entering expressions、printing values 或者類似特性的 Pascal 語法。

GDB 可以調試 Fortran 寫的程式，儘管那必然會涉及到帶有下劃線尾碼的一些變數。

GDB 可以調試 Objective-C 寫的程式，既可以使用 Apple/NeXT 運行時庫，也可以使用 GNU Objective-C 運行時庫。

自由軟體

GDB 是自由軟體，受 GNU 公共許可證（GPL）保護。GPL 給予了你自由複製或改編程式的許可——就是說獲得拷貝的人也就獲得了自由修改它的權利（這意味著他們必須有權訪問源代碼），而且可以自由的發佈更多的拷貝。大部分軟體公司所使用的版權限制了你的自由。自由軟體基金會利用 GLP 保護了這些自由。

基本上來說，公共許可證是一個說明你擁有這些自由的許可證，而且你不能把這些自由從任何人那裏占為己有。

自由軟體急需自由文檔

當今的自由軟體社區所存在的最大缺憾不在於軟體——而在於沒有我們可以隨同自由

套裝軟體含在一起的好文檔。好多我們十分重要的程式沒有一同提供自由的參考指南和介紹性文本。對任何一個套裝軟體來說，文檔是最基本的部分。當一個重要的自由套裝軟體沒有與一個自由手冊或指南一同提供時，那就是一個極大的缺憾。如今，我們擁有太多這樣的缺憾了！

拿 Perl 來說，人們日常所使用的指導手冊就不是免費的。為什麼會這樣呢？因為這些手冊的作者們在發表它們的時候伴有很多限制項目——不能複製、不能修改、不能得到原始檔案——把它們從自由軟體世界中驅逐出去了。

這類的事情已經不只發生過一次了，而且今後還會陸續發生。我們經常聽到某位熱心的 GNU 用戶說他正在編寫的一個手冊，他打算把它捐獻給社區，可沒想到他簽署了出版合同而使這個手冊不自由了，所有的期望全都破滅。

自由文檔，就像自由軟體一樣，是自由的，不需要付費的東西。非自由手冊的問題不在於發行商為印刷拷貝所要承擔的費用——只要它本身很好就行（自由軟體基金會也出售印刷拷貝），而在於這個問題會約束手冊的利用。自由手冊可以以源代碼的方式獲得，允許複製與修改。非自由手冊是不允許這麼做的。

自由文檔自由度的標準，一般來說與自由軟體差不多。再發佈（包括很多常規的商業再發佈）必須被允許，不管是以線上形式還以書面形式，以便手冊可以伴隨著程式的每一份拷貝。

允許有關技術性方面的內容的更正也是至關重要的。當人們更改軟體，添加或改變其某些特性時，如果他們負責任的話，也將會修改相應的手冊——因而，他們能夠為修改過的程式提供準確而清晰的文檔。某個手冊的頁數你是無法決定的，但是為某個程式的變更版本寫一份全新的手冊，對於我們的社區來說，那真是沒有必要。

在改進過程中所運用的某些限制是合理的。例如，要求保持原作者的版權通告、發佈條款、以及作者名單，是沒有問題的。在修正版本中包含是他們更正的通告也是沒有問題的。只要論述的是非技術性的話題（就像這一章），可以接受連續完整的章節不可刪除或被更改。能夠接受這些限制，是因為它們不會妨礙社區對手冊的正常使用。

無論如何，必須允許對手冊中所有關技術性方面的內容進行修改，然後通過所有正常的通道，利用所有常規的媒質，發佈這個結果。否則，這些限制就妨礙了對手冊的使用，那麼它就是非自由的了，我們就得需要一個新的手冊來代替它了。

請散佈有關這一論點的言辭。我們的社區仍然在遺失好多手冊，這些手冊都在成為私有出版物。如果我們趁早散佈自由軟體急需自由參考手冊和指南這樣的言辭的話，也許下一個投稿人就會意識到，只有少數的手冊投稿給了自由軟體社區。

如果你正在撰寫文檔，請堅持在GNU的自由文檔許可證或其他的自由許可證下出版它。別忘了，這個決策是需要爭得你的贊同的——你不用理會出版社的決策。只要你堅持，某些出版社會使用自由許可證的，但是他們不能奢求有買賣的特權；那需要由你自己來發行，並且堅定地說：這就是你想要的。如果這個出版社拒絕了你的生意，那就再換一家。如果你不

能確定某個被提議的許可證是自由的，就寫信給licensing@gnu.org。

你可以使用購買的方式來鼓勵商業出版社出售更多的免費的，非贏利版權的手冊與指南，尤其是購買那些來自於出版社的拷貝，付給他們撰寫或作重大改進的費用。同時，儘量完全避免購買非自由的文檔。在購買之前，先查看一下發佈條款，不管誰要做你的生意都必須尊重你的自由。查看書的歷史，設法獎勵支付了作者們工資的那些出版社。

自由軟體基金會在http://www.fsf.org/doc/other-free-books.html維護了一個已經由其他一些出版社出版了的文檔的列表。

GDB的貢獻者們

Richard Stallman 是 GDB 的原作者，也是其他好多 GNU 程式的原作者。好多人已經對它的開發作了貢獻。謹以此節來表彰那些主要的貢獻者們。自由軟體的一個優點就是每個人都無償的為它作貢獻。遺憾的是，我們無法逐一向他們表示感謝。在 GDB 的發佈中，有一個“ChangeLog”檔，做了極為詳盡的說明。

2.0 版本以前的大量變化已湮滅在時間的迷霧中。

懇請：極力歡迎對本節的補充。如果您或您的朋友（或者是敵人，為了公平），不公平地在這個列表中被遺漏了，我們願意加入您的名字。

為了使那些可能被遺忘的人們的工作不至於徒勞無功，在此特別感謝那些帶領 GDB 走過各個重要發佈版的那些人：Andrew Cagney（發佈了 6.1, 6.0, 5.3, 5.2, 5.1 和 5.0 版）；Jim Blandy（發佈了 4.18 版）；Jason Molenda（發佈了 4.17 版）；Stan Shebs（發佈了 4.14 版）；Fred Fish （發佈了 4.16，4.15，4.13，4.12，4.11，4.10 和 4.9）；Stu Grossman 和 John Gilmore（發佈了 4.8，4.7，4.6，4.5 和 4.4 版)；John Gilmore（發佈了 4.3，4.2，4.1，4.0 和 3.9 版）；Jim Kingdon（發佈了 3.5，3.4 和 3.3 版）；以及 Randy Smith（發佈了 3.2，3.1 和 3.0）。

Richard Stallman，在 Peter TerMaat、Chris Hanson、和 Richard Mlynarik 的多次協助下，完成到了 2.8 版的發佈。

Michael Tiemann 是 GDB 中大部分 GNU C++支持的作者，得益于來自 Per Bothner 和 Daniel Berlin 的其他的一些重要貢獻。James Clark 編寫了 GNU C++ 反簽名編碼器（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 發佈版的常規更新工作）。

GDB 是使用 BFD 副程式庫來分析多種目標檔格式的，BFD 是 David V. Henkel-Wallace、Rich Pixley、Steve Chamberlain 和 John Gilmore 的一個合作專案。

David Johnson 編寫了最初的 COFF 支援。Pace Willison 做了最初的壓縮的 COFF 支援。

哈裏斯電腦系統（Harris Computer Systems）的 Brent Benson 貢獻了 DWARF 2 的支持。

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

Andreas Schwab 貢獻了 M68k GNU/Linux 的支持。

Rich Schaefer 和 Peter Schauer 為支持 SunOS 的共用庫提供了幫助。

Jay Fenlason 和 Roland McGrath 保證了 GDB 和 GAS 適用於若干機器指令集。

Patrick Duval、Ted Goldstein、Vikram Koka 和 Glenn Engel 幫助開發了遠端調試。Intel 公司、風河系統（Wind River Systems）、AMD、以及 ARM 分別貢獻了 i960、VxWorks、A29K UDI 和 RDI targets 的遠端調試模組。

Brian Fox，readline 庫的作者，正在提供命令行編輯與命令歷史功能。

SUNY Buffalo 的 Andrew Beers 編寫了語言切換代碼、Modula-2 的支援，並且貢獻了此手冊的語言一章。

Fred Fish 做了支持Unix System Vr4 的大部分編寫工作。他也增強了command-completion 的支持，使其覆蓋到了 C++的超載符號。

Hitachi America (現在是 Renesas America), Ltd。負責了對 H8/300、H8/500 和 Super-H 處理器的支援。

NEC 負責了對 v850、Vr4xxx 和 Vr5xxx 處理器的支援。

Mitsubishi（現在是 Renesas）負責了對 D10V、D30V 和 M32R/D 處理器的支援。

Toshiba 負責了對 TX39 Mips 處理器的支援。

Matsushita 負責了對 MN10200 和 MN10300 處理器的支援。

Fujitsu 負責了對 SPARClite 和 FR30 處理器的支援。

Kung Hsu、Jeff Law 和 Rick Sladkey 添加了對硬體監視點（hardware watchpoints）的支持。

Michael Snyder 添加了對跟蹤點（tracepoints）的支持。

Stu Grossman 編寫了 gdbserver。

Jim Kingdon、Peter Schauer、Ian Taylor、及 Stu Grossman，修復了幾乎數不清的 bug，並且對整個 GDB 做了清理。

惠普公司（Hewlett-Packard Company）的一些人貢獻了對 PA-RISC 2.0 體系、HP-UX 10.20、10.30 和 11.0 (窄模式)、HP 的內核執行線程、HP 的 aC++編譯器、以及文本用戶介面（舊稱終端用戶介面）的支援。他們是：Ben Krepp、Richard Title、John Bishop、Susan Macchia、 Kathy Mann、Satish Pai、India Paul、Steve Rehrauer 和 Elena Zannoni。Kim Haase 提供了此手冊中的 HP-specific 資訊。

DJ Delorie 為 DJGPP 專案，把 GDB 移植到了 MS-DOS 上。Robert Hoehne 對 DJGPP 的移植作了重大的貢獻。

Cygnus Solutions 已負責起 GDB 的維護，自 1991 年以來已做了大量的開發工作。Cygnus 為 GDB 做全職工作的工程師有：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. 另外還有：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，他們做了大大小小不同的貢獻。

Andrew Cagney、Fernando Nasser 和 Elena Zannoni，他們在 Cygnus Solutions 工作時，實現了最初 GDB/MI 介面。

Jim Blandy 添加了預處理宏的支援，那時他在 Red Hat 工作。

Andrew Cagney 設計了 GDB 的結構向量。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，為把舊有的體系結構移植到這個新的框架上提供了幫助。

請發送FSF和GNU的疑問和問題到gnu@gnu.org。這也有一些其他的方式聯繫FSF。

這些頁面是由GDB的開發者們維護的。

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

1・一個簡單的GDB會話

你可以在你的業餘時間利用本手冊瞭解有關 GDB 的一切。可是，少數幾個命令，就足以開始使用調試器了。本章就闡明了那些命令。

GNU m4（一個普通的宏處理器）的一個初級版本表現出下列 bug：有些時候，當我們改變它默認的引證串（quote string，譯者注：也就是我們常說的引號）時，用於捕獲一個在別處定義的宏的命令停止工作。在下列簡短的 m4 會話中，我們定一個可擴展為 0000 的宏；然後我們利用 m4 內建的 defx 定義一個相同的東西 bar。可是當我們把左引證串(open quote string，譯者注：英文直譯為開引證串)改為<QUOTE>，右引證串（close quote string）改為<UNQUOTE>時，相同的程式不能定義新的替代名 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
```

讓我們利用 GDB 設法看一下發生了什麼事。

```
$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 僅讀取足夠查找所需的符號資料，餘下的按需讀取。結果是，第一提示很快就出現了。我們告訴 GDB 使用一個比平常要窄的顯示寬度，以使樣例與本手冊相適應。

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

我們要看一下 m4 內建的 changequote 是如何工作的。通過查看源代碼，我們瞭解到與其相關的副程式是 m4_changequote，因此我們使用 GDB 的 break 命令在那裏設置了一個中斷點。

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

使用 run 命令，在 GDB 的控制下，我們開始了 m4 的運行。只要控制沒有到達 m4_changequote 副程式，程式就會像平常那樣運行。

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

foo
0000
```

為了引發中斷點，我們調用 changequote。GDB 暫停執行 m4，顯示中斷處的上下文資訊。

```
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))
```

現在，我們使用命令 n（next）向前執行到當前函數的下一行。

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

set_quotes 像是一個有希望的副程式。我們可以使用命令 s（step）代替 next 進入到他的裏面。Step 轉到任意副程式中要被執行的下一行，因此他進入了 set_quotes。

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

顯示器顯示了目前 m4 所停在副程式（還有它的參數）的被調堆疊幀的展示。它顯示了一個堆疊的摘要資訊。我們可以使用 backtrace 命令（也可以被拼寫為 bt）,查看總體上我們在堆疊上的位置：Backtrace 顯示了每一個活動副程式的堆疊幀。

```
(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
```

我們向前單步調試更多的行，看看都發生了什麼。前兩次我們使用“s”；後兩次為了避免陷入 xstrdup 副程式我們使用“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);
```

最後以行看起來有點古怪；我們能夠檢查變數 lquote 和 rquote，看看他們現在是不是我們新指定的左右引號。我們使用命令 p 來查看它們的值。

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

Lquote 和 rquote 的確是新的左右引號。為了能夠查看上下文，我們可以使用 l（list）命令，圍繞著當前行，顯示十行源代碼。

```
(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
```

讓我們單步執行，通過設置 len_lquote 和 len_rquote 的那兩行，然後查看一下它們的值。

```
(gdb) n
539         len_rquote = strlen(lquote);
(gdb) n
540     }
(gdb) p len_lquote
$3 = 9
(gdb) p len_rquote
$4 = 7
```

假設 len_lquote 和 len_rquote 分別代表 lquote 和 rquote 的長度的話，那顯然就是錯

的。我們能夠利用 p 命令給它們設定更好的值，因為它可以列印任何運算式的值——運算式中還可以有副程式調用並賦值。

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

那足以修復與 m4 內建的 defn 一同使用新引號的問題嗎？我們可以使用 c（continue）命令使 m4 繼續執行，然後試試剛開始時那個有問題的例子：

```
(gdb) c
Continuing.

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

baz
0000
```

成功了！新的引號已經可以象默認的一樣工作了。問題看來就是兩個輸入錯誤導致了錯誤的長度計算造成的。我們輸入一個 EOF 讓 m4 退出。

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

“Program exited normally”是 GDB 發出的。它表明 m4 已經結束執行了。我們可以使用 GDB 的 quit 命令結束我們的 GDB 會話。

```
(gdb) quit
```

2．征服GDB的進與出

本章討論如何啟動 GDB，並且如何離開 GDB。要點為：

輸入“gdb”啟動 GDB

輸入 quit 或 C-d 退出 GDB

2.1 調用GDB

通過運行 gdb 程式來調用 GDB。一旦啟動，GDB 就開始從終端讀取命令，直到你讓它退出為止。

你也可以使用多種參數與選項來運行 gdb，在一開始就能更多地為您定義調試環境。

對於命令行選項的描述，這裏打算覆蓋多種情況。在一些環境中，某些選項可能會不起作用。

啟動 GDB 最常用的方法是使用一個參數，指定一個可執行程式：

gdb *program*

也可以指定一個可執行程式和一個 core 檔一起啟動 gdb：

gdb *program core*

如果你想調試一個運行中的程式，作為代替，你可以指定一個進程 ID 為第二個參數：

gdb *program* 1234

將把 GDB 附著到進程 1234（除非你也有一個名叫“1234”的檔；GDB 總是先檢查 core 文件）。

運動第二個命令行參數需要一個相當完整的作業系統；當我們利用 GDB 作為遠端調試器附著到一個裸板上時，那裏可能沒有任何“進程”的概念，那也就常常無法獲得 core dump。如果 GDB 沒有能力附著或讀取 core dumps 時，它會發出警告。

可以使用--args 選項，給 gdb 要調試的程式傳遞參數。這一選項使 gdb 的選項處理停止。

gdb –args gcc -O2 -c foo.c

這使得 gdb 調試 gcc，並給 gcc 傳遞命令行參數（見 4.3 程式的參數一節）“-O2 -c foo.c”。

可以通過指定-silent 選項，運行 gdb 而不列印前面的資料，這些資料說明 GDB 不作任何擔保。

gdb –silent

你可以通過設定命令行選項，更多地控制 GDB 的啟動。GDB 自身就能夠給你各種可用選項的提示。

輸入

gdb –help

顯示所有可用選項，並對它們的用法作了簡短的描述（可以簡寫成 gdb –h，作用相同）。

所有的選項和命令行參數按照你所給的順序進行處理。當使用“-x”選項時，順序就就比較重要了。

2.1.1 選擇檔

在 GDB 啟動時，除了選項之外，它把所有的參數都看作是可執行檔和 core 檔（或者是進程 ID）來讀取。就如同這些參數已分別被“-se”和“-c”（或者是“-p”）選項所修

飾過一樣。（GDB 讀取的第一個參數，如果沒有關聯的選項標誌，就等同於“-se”選項後面參數；如果有第二個這樣的參數的話，就等同於“-c”/“-p”選項後的參數）如果第二個參數是以一個 10 進制數開頭的話，GDB 首先會嘗試把它作為一個進程去附著，如果失敗了的話，就嘗試著把它作為 core 檔打開。如果有一個檔案名以數字開頭的 core 檔的話，可用通過附加./首碼來避免 GDB 把它當成 pid，如./12345。

如果 GDB 已經被配置為不支援 core 檔，比如大部分的嵌入式目標，它將拒絕第二個參數而忽略它。

大部分的選項都有長格式和短格式；都被展示在下表中。GDB 也能識別不完整的長格式，只要與給出的各個選項之間沒有歧義就行。（如果你願意，你可以使用“--”來標記選項參數，而不用“-”，雖然我們展示的是更常用的約定）。

-symbols file
-s file
　　從檔 file中讀取符號表。

-exec file
-e file
　　把檔 *file* 當作可執行檔來使用，在適當的時候執行，連同 core dump，分析單純的資料。

-se *file*
　　從檔 *file* 中讀取符號表，並把它當作可執行檔來使用。

-core file
-c file
　　把檔 file當成 core 檔使用，進行分析。

-c number
-pid number
-p number
　　同附著命令一樣，連接到一個進程 ID number。如果沒有這個進程，GDB 就嘗試著打開一個名為 number的 core 檔。

-command file
-x file

執行檔 file中的 GDB 命令。參見命令文件一章

-directory directory
-d directory
添加 directory到原文件搜索路徑。

-m
-mapped
警告：這個選項依賴於作業系統的功能，並不被所有的作業系統所支援。
如果你的作業系統可以通過 mmap 系統調用使用記憶體映射檔（memory-mapped files），你就可以利用這個選項，讓 GDB 把你程式的符號寫到當前目錄下的一個可重用檔（reusable file）中。如果你正調試程式是“/tmp/fred”，那麼對應的符號檔就是“/tmp/fred.syms”。之後的 GDB 調試會話會注意這個檔的存在，並且快速的映射它的符號資訊，而不需要從可執行程式的符號表中讀取。“.syms”檔特屬於 GDB 所運行的主機。它保存著 GDB 內部符號表的精確映射。它不能被多個主機平臺交叉共用。

-r
-readnow
立即讀取每一個符號檔的整個符號表，不同於默認的根據需要而逐步讀取。這使得啟動會更慢，但之後的操作會更快。

為了創建一個包換完整符號資訊的“.syms”檔，典型的做法是-mapped 與-readnow聯合使用（參見文件指定命令一章，有關“.syms”檔的資料）。一個簡單的創建一個“.symes”檔以便以後使用的 GDB 調用是：

gdb -batch -nx -mapped -readnow programname

2.1.2 選擇模式

可以以兩種模式中的一種運行 GDB——批次處理模式或單調模式。

-nx
-n
不執行在任何初始化檔中找到的命令。默認情況下，GDB會在處理完所有命令選項與參數之後執行這些檔裏的命令。參見命令文件一章。

-quiet
-silent
-q

“單調”。不列印介紹性資訊和版權資訊。這些資訊在批次處理模式下也是不列印的。

-batch

以批次處理模式運行。處理完所有由“-x”所指定的命令檔（如果不使用“-n”來約束，還有所有的初始化檔）後以狀態 0 退出。在執行命令檔中的命令過程中，如果發生錯誤，則以非 0 狀態退出。

GDB 作為一個篩檢程式運行時，批次處理模式可能會有用，例如，在另外一台電腦上下載並運行一段程式；為了使這個更有用，消息

Program exited normally.

(只要程式是在 GDB 的控制下運行終止的，一般都會使這個結果)說明批次處理模式下的運行是沒有問題。

-nowindows
-nw

“無窗口”。倘若 GDB 伴隨有一個圖形用戶介面（GUI），這時這個選項就告訴 GDB 僅使用命令行介面。如果沒有 GUI 可用，這個選項也就沒有作用。

-windows
-w

如果 GDB 含有一個 GUI 的話，那麼這個選項就是請求：如果可能的話，就使用它。

-cd *directory*

用 directory代替當前的目錄，作為 GDB 的運行工作目錄運行。

-fullname
-f

當 GDB 作為 GNU Emacs 的子進程運行時，設置這個選項。他告訴 GDB 以某一標準輸出完整的檔案名和行號，以便每次都能以大家公認方式顯示棧結構（這也包括每次程式停止時）。這個公認的格式為：兩個“\032”字元後面跟著一個檔案名，行號和字元位置以顏色區分，外加一個換行。Emacs-to-GDB 的介面程式利用兩個“\032”字元作為顯示棧結構源代碼的信號。

-epoch

當 GDB 作為 Epoch Emacs-GDB 介面的子進程運行時，設置這個選項。它告訴 GDB 修改它的列印程式，以便 Epoch 可以在分割視窗中顯示運算式的值。

`-annotate` *level*

這個選項設置GDB內部的注釋級別。它的作用和使用“set annotate *leave*”相同（參見25.GDB的注釋一章）。注釋級別控制著應有多少消息同GDB的提示符一起列印，這些資訊有：運算式的值、源代碼行數以及其他類型的一些輸出。級別 0 是默認級別，級別 1 用於GDB作為Emacs的子進程時，級別 3 是最大量的在GDB控制下的程式的相關資訊，級別 2 現在已經不被建議使用了。

這個注釋機制在很大程度上已被GDB/MI所代替（參見24.GDB/IM介面一章）。

`--args`

改變對命令行的解釋，以使可執行檔的參數可以被作為命令行參數傳給它。這個選項阻止選項處理。

`-baud bps`
`-b bps`

設置任一用於 GDB 遠端調試的串列介面的線速度。

`-l` *timeout*

設置任一用於 GDB 遠端調試的通訊的超時值（以秒為單位）。

`-tty device`
`-t device`

把 device作為程式的標準輸入輸出運行。

`-tui`

啟動時啟動文本用戶介面。文本用戶介面在終端上管理幾個文本視窗，顯示源代碼、彙編、寄存器和GDB的命令輸出（參見GDB文本用戶介面一章）。最為選擇，通過調用“gdbtui”程式可以啟動文本用戶介面。如果你在Emacs中運行GDB，不要使用這個選項（參見在GNU Emacs下使用GDB一章）。

`-interpreter` *interp*

使用解釋器interp與控制程式或設備一起作為介面。這個選項是想讓與GDB通訊的

程式作為它的一個後端程式。參見命令解釋器一章。

“--interpreter=mi”（或者“--interpreter=mi2“）讓GDB使用其 6.0 版以後包含的GDB/MI介面（參見GDB/MI介面一章）。之前包含在GDB 5.3 中的GDB/MI介面使用“--interpreter=mi1”選擇，但不贊成使用。早期的GDB/MI介面已經不提供支援了。

-write

打開可執行檔和Core檔，可讀可寫。這相當於GDB裏面的“set write on”命令。（參見14.6 修補程式一節）

-statistics

這個選項讓 GDB 在完成每個命令並返回到提示符後，列印有關時間和記憶體使用率的統計。

-version

這個選項讓 GDB 列印它的版本號和“無擔保”內容後退出。

2.1.3 啟動期間，GDB做了什麼

這裏是 GDB 在會話啟動期間做了什麼的描述：

1． 按照命令行的規定，準備命令解釋器（參見解釋器一章）。

2． 在你的 HOME 目錄中讀取初始化檔(如果有)，這行那個檔裏的所有命令。3．處理命令行選項和操作物件。

4． 讀取並執行當前工作目錄中初始化檔（如果有）中的命令。僅在工作目錄與 HOME 目錄不同時這樣做。因此，可以擁有不止一個初始化檔。在 HOME 目錄下的是一個普通的，另外一個，在調用 GDB 的目錄下的，是專用於程式調試的。

5． 讀取由“-x”選項所指定的命令檔。要獲得有關GDB命令檔的詳細資訊，請參考20.3 命令文件一節。

6． 讀取歷史檔中的命令歷史記錄。要獲得有關命令歷史以及GDB用於記錄它的檔的詳細資訊，請參考19.3 命令歷史一節。

初始化檔與命令檔使用相同的語法（見20.3 命令檔一節），而且在GDB中也是以相同的方法處理。在HOME目錄下的初始化檔可以設置那些能夠影響後續的命令行選項處理的選項（比如“set complaints”）。如果使用了“-nx”選項（參見選擇模式一節），初始化檔就不會被執行。

GDB 的初始化檔通常被命名為“.gdbinit”。在某些 GDB 的配置中，初始化檔可以是其他的名字（有些典型的環形，有一個 GDB 的專有形式需要與其他的形式並存，因此，給專有版本的初始檔一個不同的名字）。這些使用特殊初始化檔案名的環境有：

GDB 的 DJGPP 移植使用的是“gdb.ini”，這是由於受到 DOS 檔系統對檔案名的強制限制。GDB 的 Windows 移植採用的是標準名，而且一旦發現“gdb.ini”檔，它會發出警告，建議把這個檔更改為標準名。

VxWorks(風河 Systems 的即時操作系統)：“.vxgdbinit” OS68K（Enea Data Systems 的即時操作系統）：“.os68gdbinit” ES-1800（愛立信電信 AB M68000 模擬器）：“.esgdbinit” CISCO 68k：“.cisco-gdbinit”

2.2 退出GDB

```
quit [expression]
q
```

使用 quit（縮寫 q）命令退出 GDB，或者鍵入一個檔結束符（通常是 C-d）。如果你不提供 expression，GDB 會正常的結束；否則，它會以 expression的結果作為錯誤代碼結束。

中斷（通常是 C-c）不會使 GDB 退出，而是終止了在進程中所有 GDB 命令的動作，並返回到 GDB 的命令層。在任何時候鍵入中斷符都是安全的，因為 GDB 在不安全的時候是不會使它生效的。

如果你曾用GDB控制一個被附著的進程或設備的話，你可以使用detach命令釋放它（參見調試一個早以運行的進程一章）。

2.3 Shell命令

要在調試會話中需要執行一個應時的 shell 命令，是不需要離開或掛起 GDB 的；只需使用“shell”命令即可。

shell command string

調用標準 shell，執行 command string。倘若有環境變數 SHELL，那麼它就決定應該運行哪個 shell。否則，GDB 就使用默認的 shell（UNIX 系統是/bin/sh，MS-DOS 是 COMMAND.COM，等等）。

make 工具一般是開發環境所需要的。最好不要在 GDB 中使用它。

make make-args

以指定的參數執行 make 程式。等同於“shell make *make-args*”。

2.4 Loging輸出

你可能希望將 GDB 的命令輸出保存到一個檔中去。這裏有幾個指令控制 GDB 的 Loging。

set logging on

開啟 logging。

set logging off

關閉 logging。

set logging file file

更改當前 logfile 的名字。默認的 logfile 是“gdb.txt”。

set logging overwrite [on|off]

通常，GDB 是追加 logfile。如果設置 overwrite 為 on 的話，GDB 覆蓋 logfile。

set logging redirect [on|off]

通常，GDB 同時向終端和 logfile 輸出。要是想讓 GDB 只輸入到 logfile，就設置 redirect 為 on。

show logging

顯示當前的 logging 設置值。

3．GDB命令

可以使用命令名的頭幾個字母簡寫 GDB 的命令，只要這個簡寫不會產生二義性；還以通過鍵入 RET 重複某些命令。也可用利用 TAB 鍵，讓 GDB 自己填寫命令單詞中剩餘的部分（或者是可用的備選方案，只要有多於 1 個的可能性）。

3.1 命令語法

一個 GDB 命令是一個單行輸入，對於長度沒有限制。以命令名開始，後跟一些參數，這些參數的意義取決於命令名。例如，step命令，它接受一個表明步進次數的參數，如在“step 5”中。step 命令也可以不帶參數。有些命令根本就不允許有參數。

GDB的命令總是可以被縮短的，只要這個簡寫不會產生二義性。其他可能的命令簡寫都已被列在個專用令文檔中了。在某些情況下，甚至二義性的簡寫也是被允許的；例如：s就是被特別定義為等同於step的，即使還有其他的一些名字以s開頭的命令。可以把它們作為help命令的參數來測試一個簡寫命令。向GDB輸入一個空白行（僅鍵入RET）意味著重複先前的命令。這種方法是不能重複某些命令的（比如run）。有些命令，不小心的重複可能會導致一些問題，況且你也不太可能要重複。自定義命令可以關閉這一特性，見禁止重複一節。

list 和 x 命令，當你使用 RET 重複它們時，構造新的參數，而不是原來的。這允許輕鬆地掃描源代碼或記憶體。

GDB也可以以另外一種方式使用RET：以一種類似通用工具more的方式，分屏冗長的輸出（見螢幕尺寸一章）。由於在這種情形下，按一次RET後容易產生太多的重複，因此，GDB在任何能夠產生那種顯示的命令以後都關閉了命令重複。

任何從一個#到行尾的文本都是一個注釋；它什麼也不做；它主要用在命令檔中（見命令檔一章）。

C-o 組合有重複一個複雜命令序列的作用。這個命令接受當前行，如同 RET，然後從編輯歷史記錄中讀取與當前行相關的下一行。

3.2 命令完成

如果只有一種可能性的話，GDB 可以為你填寫命令單詞中剩餘的部分；也可以把所有有效的可能性的命令單詞顯示給你，而且任何時候都可以。這個功能對 GDB 命令，GDB 子命令以及程式中的符號名起作用。

只要你想讓 GDB 填寫一個詞的剩餘的部分，按 TAB 鍵即可。如果只有一種可能性，GDB 在填寫完後就等待你去完成這個命令（或者說按 RET 輸入它）。例如，如果你鍵入：

（gdb） info bre TAB

GDB 填寫“breakpoints”單詞的剩餘部分，因為 info 的子命令以 bre 開頭的只有它：

（gdb） info breakpoints

既可以在這個位置按下 RET 運行這個 info breakpoints 命令，也可以按下控各，輸入其他的東西，如果“info breakpoints”不是你所期待的命令的話（如果“info breakpoints”確實就是你想要的，還是在“info bre”後立即按下 RET 為好，與其使用命令完成，還不如使用命令縮寫）。

當你按 TAB 時，如果對於下一個單詞有多於一個的可能性的話，GDB 會發出一個蜂鳴。你既可以補充更多的字母然後重試，也可以在按一下 TAB，GDB 顯示對於那個單詞的所有可能的完成。例如，你可能想在一個名字以“make_”開頭的副程式中設置一個中斷點，可是當你鍵入 b make_TAB 時，GDB 只是發出了一個蜂鳴。再按一次 TAB，顯示出程式中所有以這些字母開頭的副程式名。

(gdb) b make_ TAB

GDB 發出蜂鳴；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_
```

顯示完可用的可能性後，GDB 複製你的部分輸入（例子中是“b make_”），便於你完成命令。

如果先前你只是想看一下備選方案列表，與其按 TAB 兩次，還不如按 M-?。M-？意思是 META？。即可以通過按住鍵盤上代表 META 的鍵（如果有，譯者注：在 PC 上一般是 ALT），再按下？輸入，也可以按住 ESC 後再按？輸入。

有些時候，你需要的字串，雖然邏輯上是一個“詞”，但可能包含圓括號或者是其他的字元，被 GDB 正常地從它的詞的概念上排除了。為了“詞完成”可以在這種情況下工作，可以在 GDB 命令中用’（單引號）把這些詞包圍起來。

這最適合在鍵入 C++函數名時使用這一功能。這是因為 C++允許函數重載（相同的函數有多個定義，以參數類型來區分）。例如，當你想要設置一個中斷點時，你可能需要區分是否是你想要的那個帶有一個 int 型參數的 name 版本 name(int)，還是另外一個帶有一個 float型參數的 name 版本 name(float)。為了在這種情況下使用“詞完成”功能。在函數名的開頭敲一個單引號’。這通知 GDB，當按下 TAB 或 M-?請求“詞完成”時，需要考慮比平時要多的資訊。

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

在某些情況下，GDB 能夠知道正完成的名字需要使用引號。當這發生時，如果你先前沒有輸入引號，GDB 會為你插入引號（盡其所能地）。

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

GDB 通知你，你的輸入行變成下面這樣，並發出蜂鳴：

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

通常，當你請求一個關於重載符號的“完成”時，如果你還沒有敲入參數表，GDB 都

會知道那需要一個引號。

要獲得更多有關重載函數的信息，請參見C++表達式。可以使用命令 set overload-resolution off 關閉重載轉換。參見適用於C++的GDB特性。

3.3 獲得幫助

總是可以向 GDB 本身詢問有關命令的資訊，使用命令 help。

help
h

可以使用沒有參數的 help（簡寫 h）顯示一個簡短的命令分類的命名列表。

```
(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*

使用一個分類作為參數，可以獲得這個分類中的單獨命令的列表。這是一個分類

的幫助顯示情況。

```
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*

以一個命令名作為幫助的參數。GDB 顯示一個簡單的如何使用這個命令的短評。

apropos *args*

apropos 命令詳細地 GDB 的所有命令以及它們的文檔進行搜索，並與在 *args* 指定的正則運算式進行比配，將所有比配到的資訊列印輸出。例如：

apropos reload

結果是：

```
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* 命令列出所有適合於某一個命令開頭可能的“完成”。使用 *args* 指定你想要“完成”的命令開頭。例如：

complete i

結果是：

if
ignore
info
inspect

這個打算由 GNU Emacs 使用。

除了help之外，可以使用GDB命令info和show瞭解關於程式的狀況，或者是GDB本身的狀況。每一個命令都支援很多的詢問主題。本手冊會在適當的情況下一一介紹它們。在索引中，info和show下的列表，指向了所有的子命令。參見索引一章。

info

這個命令（簡寫 i）適用於描述程式的狀況。例如：info args 可以列出賦給程式的參數，info registers 列出當前使用的寄存器，info breakpoints 可以列出已設置的中斷點。使用 help info 可以獲得一個有關 info 子命令的完整列表。

set

使用 set 可以把運算式的結果設置給環境變數。例如：用 set prompt $ 把 GDB 的提示符設置為$符。

show

與 info 正好相反，它顯示的是 GDB 本身的狀況。利用相關的命令 set，可以更改大部分可以顯示的東西。例如：使用 set radix 控制使用什麼數制用於顯示，或者使用 show radix 簡單地詢問當前所使用的數制。

為了顯示所有可置位元的參數以及它們的值，可以使用不帶參數的 show，也可以使用 info set。兩個命令產生相同的顯示。

這裏有三個其他的 show 子命令，它們特別的是都沒有對應的 set 命令：

show version

顯示正在運行的 GDB 的版本。應該在 GDB 的 bug 報告中包含這個資訊。如果你

的地方用了多個版本的 GDB，你可能需要確定你正在運行的是哪一個版本。由於 GDB 的進展，新命令的引用，舊命令可能已經幻滅。同樣，好多系統供應商配備有不同的 GDB 版本，而且這些不同的 GDB 版本也是發佈在 GNU/Linux 中。在啟動 GDB 時，版本號同樣通告一次。

show copy

info copy

顯示 GDB 的複製許可資訊。

show warranty

info warranty

顯示 GNU“無擔保”資訊，或者一個擔保，要是你的 GDB 版本有一個的話。

4．在GDB下運行程式

在 GDB 下運行程式時，首先必須在編譯時生成調試資訊。

在你選擇的一個環境中，如果有的話，你可以使用參數啟動 GDB。如果進行的是本地調試，可以重定向程式的輸入輸出，調試某個已經運行的進程，或者殺掉某個子進程。

4.1 適合調試的編譯

為了有效的調試某個程式，需要在編譯的時候生成調試資訊。這個調試資訊被存儲在目標檔中；它描述了各個變數或函數的資料類型，以及可執行代碼與源代碼行號之間的對應關係。

在運行編譯器時，指定“-g”選項，要求編譯器產生調試資訊。配備給客戶的程式是使用最優化編譯的，使用的是“-O”選項。可是，好多編譯器不能把“-g”和“-O”選項放在一起處理。使用這些編譯器，不能產生包含編譯資訊的最優化可執行代碼。

GCC，GNU C/C++編譯器，支持附帶或不附帶“-O”的“-g”，這使得它能夠調試優化後的代碼。我們建議，只要編譯程序，就使用“-g”。你可能認為你的程式是正確的，但不要期望好運會持續。

在調試某個使用“-g –O”編譯的程式時，要想到優化程式已經重排了代碼，調試器顯示給你的是真實的代碼。執行路徑與原始檔案不符時，不要太驚訝。一個極端的例子：假如你

定義了一個變數，但是從來沒有使用過，GDB 根本就看不到到那個變數——因為編譯器已經把他優化掉了（譯者注：不存在了）。

僅使用“-g”，某些東西跟使用“-g –O”工作的不一樣，特別是在有指令調度的機器上。要是不信的話，就單獨使用“-g”重新編譯，這要是校準這個問題，請把他作為一個bug發送給我們（包括測試條件）。要得到更多有關調試優化代碼的資訊，參見8.2 程式變數一章。

老版本的 GCC 編譯器允許有一個不同的選項“-gg”用於生成調試資訊。GDB 現已不再支持這個格式。要是你的 GCC 有這個選項，請不要使用。

GDB瞭解有關的預處理宏，而且可以把它展開後顯示給你（參見9.C的預處理宏一章）。單獨指定“-g”標誌，大部分編譯器不會在調試資訊中包含有關預處理巨集的資訊，因為這些資訊是相當龐大的。3.1 及其以後版本的GCC中的GNU C編譯器，如果指定了“-gdwarf-2”和“-g3”選項，可以提供巨集的資訊。前一個選項要求產生Dwarf 2 格式的調試資訊，後一個選項要求產生“非常資訊（extra information）”。今後，我們希望找到些更緊湊的方式來描述巨集資訊，以使它單獨使用“-g”就可以被包含。

4.2 啟動程式

run
r

利用run命令在GDB下啟動程式。首先必須使用一個參數給GDB指定程式的名稱（除非是在VxWorks上），或者使用file或exe-file命令指定（見指定文件的命令一節）

如果在一個支援進程的環境下運行程式，run 創建一個次級進程，讓這個進程來運行程序。（在沒有進程的環境下，run 跳轉的程式的開始處）。

某個程式的執行，受到從它上級接收到的某些資訊的影響。GDB 提供了指定這些資訊的方式，但必須在程式啟動之前指定。（雖然能夠在程式啟動之後更改它，但是所作的更改只能在程式下次啟動後才有作用。）這些資訊可以被劃分為四類：

參數

把參數最為運行命令參數指定給程式。如果在你的目標上有一個shell可用，那麼這個shell通常用於傳遞這個參數，以便於可以利用常規描述參數的約定（如通配符展開或變數置換）。在UNIX系統中，可以利用SHELL環境變數控制使用哪一個Shell。參見程式的參數一節。

環境

程式一般從GDB繼承它的環境，但是也可以使用GDB命令set environment與unset environment更改某些影響程式的局部環境。參見程式的環境一節。

工作目錄

程式從GDB繼承它的工作目錄，在GDB中可以使用cd命令設置GDB的工作目錄。參見程式的工作目錄一節。

標準輸入輸出

程式一般與GDB所使用的相同的設備作為標準輸入輸出。可以在run的命令行裏從定向輸入輸出，或者使用tty命令給程式設定一個不同的設備。見程式的輸入輸出一節。

警告：當輸入輸出從定向生效時，不能使用管道傳遞正調試的程式的輸入給另外一個程式。試圖這麼做的話，GDB 可能終止調試錯誤的程式。

下達run命令時，程式會立即開始執行。有關討論怎樣安排程式的停止，見停止與繼續一節。一旦程式已經停止，就可以使用print或call命令調用程式中的函數。參見檢驗資料一章。

自上一次 GDB 讀取符號後，符號檔的修改時間已更改的話，GDB 會重讀它。當做這件事的時候，GDB 會努力保持當前的中斷點。

start

不同的語言，主過程名也不相同。C 或 C++的主過程永遠是 main，但是像 Ada 就不要求為它們的主過程起個特殊的名字。調試器提供了一個方便的方法開始程式的執行，並在主過程的起始位置停住，這依賴於所使用的語言。

“start”命令相當於在程式主過程的起始位置設置一個臨時中斷點之後調用“run”命令。

有些程式會包含一個細化（*elaboration*）階段，有些代碼在主過程被調用之前執行。這取決於編寫程式的具體語言。例如，在 C++中，靜態的與全局的物件的構造函數會在main被調用之前執行。這使得調試器在達到主過程之前停止程式成為可能。不管怎樣，臨時中斷點仍然會停止程式的執行。

程式的參數可以作為“start”命令的參數指定給程式。這些參數會精確地傳遞給次級的“run”命令。要注意，後面調用的“start”或“run”沒有提供給參數的話，會重用前面提供的那個參數。

在細化期間調試程式，有些時候是必要的。在這種情況下，使用 start 命令對於停止程式的執行來說，那太晚了，因為這時程式早已完成了細化階段。在這種情況下，在運行程式之前在細化代碼中插入中斷點。

4.3 程式的參數

程式的參數可由“run”命令的參數指定。它們被傳遞給一個 shell，展開通配符並重定向標 I/O，最後傳遞給程式。SEHLL 環境變數（如果有）規定了 GDB 使用什麼 shell。如果沒有定義 SHELL，GDB 會使用默認的 shell（在 UNIX 上是“/bin/sh”）。

在非 UNIX 系統上，程式一般由 GDB 直接調用，利用適當的系統調用來類比 I/O 重定向，同時，通配符由程式的啟動代碼展開，而不是 shell。

沒有參數的 run 通常使用前一個 run 的參數，或者是 set args 命令指定的那些。

set args

指定下次程式運行用到的參數。如果 set args 沒有參數，run 不傳遞參數而執行程式。一旦使用參數 run 了程式，那麼在下次 run 程式之前使用 set args，是再次不使用參數 run 程式的唯一方法。

show args

顯示啟動時傳給程式的參數。

4.4 程式的環境

“環境”由一組環境變數和它們對應的值構成。環境變數通常記錄著如：用戶名、HOME 目錄、終端類型以及程式運行的搜索目錄這些資訊。通常你用 shell 設置的環境變數會被所有你運行的其他程式所繼承。調試時，不需要反復啟動 GDB 就可以使用一個修改的環境變

量試著運行程式，是很有用的。

path *directory*

在傳遞給程式的 PATH 環境變數（可執行程式的搜索路徑）前，添加 *directory*。GDB 所使用的 PATH 值不會被更改。可以指定若干的目錄名，使用空格或系統相關的分隔符號（UNIX 上是“:”，MS-DOS 和 MS-Windows 上是“;”）分隔它們。如果 *directory* 已經在 PATH 中了，就把它移到前面，以使它可以被立即搜索到。可以使用字串“$cwd”，在 GDB 搜索路徑時，引用當前工作目錄。如果使用“.”代替的話，它引用的是由執行 path 命令指定的目錄。在添加 *directory* 到搜索路徑之前，GDB 在 *directory* 參數中替換“.”（使用當前路徑）。

show path

顯示可執行檔的搜索路徑列表（PATH 環境變數）。

show environment [*varname*]

列印程式啟動時慣於它的環境變數 *varname* 的值。要是沒有給出 *varname*，就列印所有慣於程式的環境變數的名和它的值。可以用 env 簡寫 environment。

set environment *varname* [=value]

給環境變數 *varname* 賦值為 *value*。變數值的更改僅針對於程式，不會影響 GDB 本身。*value* 可以是任何字串——環境變數值只能是字串，對它的任何解釋都由程式本身提供。*value* 參數是可選的。如果去除的話，變數會被賦予一個 null 值。例如這個命令：

set env USER = foo

告訴 GDB，在隨後的 run 時，它的用戶是一個名叫“foo”的（“=”前後使用的空格是為了清晰，實際上是不需要的）。

unset environment *varname*

從傳遞給程式的環境中移除變數 *varname*。這與“set environment varname=”不同，它是從環境中移出變數，而不是給它賦一個空值。

警告：在 UNIX 系統中，如果有 SHELL 環境變數的話，GDB 使用的是它指定的 shell（沒有的話使用/bin/sh）。如果你的 SHELL 環境變數指定了一個運行初始化檔（如 C-Shell的“.cshrc”或者是 BASH 的“.bashrc”）的 shell，你設置在那個檔中的任何變數，都會對你的程式有所影響。你可能希望把這些環境變數的設置移到僅在你註冊時運行的檔中去，這樣的檔有“.login”或“.profile”。

4.5 程式的工作目錄

每次使用 run 啟動的程式，都 GDB 的當前工作目錄繼承為它的工作目錄。GDB 的工作目錄最初是繼承自它的父進程（通常是 shell），不過，你可以在 GDB 中使用 cd 命令指定一個新的工作目錄。

GDB的工作目錄也擔當著GDB操作的指定檔命令的默認目錄。參見指定文件的命令一節。

cd *directory*

設置 GDB 的工作目錄為 *directory*。

pwd

列印 GDB 的當前工作目錄。

找到正被調試的進程的當前工作目錄，通常是做不到的（因為一個程式可以在它運行的時候改變它的目錄）。如果你工作在GDB可以被配置為帶有“/proc”支援的系統上的話，你可以利用info proc命令（見18.1.3 SVR4 進程資訊一節）找到debuggee的當前工作目錄。

4.6 程式的輸入輸出

默認情況下，GDB 下運行的程式作輸入輸出的終端，與 GDB 所使用的是相同的。GDB 把終端轉換為它自己的終端方式與你交互，不過，它記錄你的應用程式所使用的終端方式，當你繼續運行你的程式時，它再切換回來。

info terminal

顯示由 GDB 記錄下來的程式所使用的終端模式的資訊。

可以使用 run 命令，利用 shell 的重定向，重定向程式的輸入和/或輸出。

run > outfile

啟動程式，驅使他的輸出到檔“outfile”。

另外一種指定程式在何處做輸入輸出的方式是使用 tty 命令。這個命令接受一個作為參數的檔案名，並使這個檔成為之後 run 命令的默認重定向目標。它也為子進程、之後的 run命令重置控制終端。例如：

tty /dev/ttyb

指示後續用 run 命令啟動的進程默認在終端“/dev/ttyb”上作輸入輸出，並且拿這個作為它們的控制終端。

在 run 中明確的重定向，優先於 tty 對輸入/輸出設備的影響，但是並不對控制終端有影響。

當使用 tty 命令或在 run 命令中重定向輸入時，只有程式的輸入會受到影響。GDB 的輸入依然來自你的終端。tty 是 set inferior-tty 的別名。

可以使用 show inferior-tty 命令告訴 GDB，顯示將來程式運行會被使用的終端的名字。

set inferior-tty /dev/ttyb

設置正被調試的程式的 tty 為/dev/ttyb。

show inferior-tty

顯示正被調試的程式的當前 tty。

4.7 調試某個已運行的進程

attached *process-id*

這個命令附著到一個正在 GDB 以外運行的進程。（info 檔顯示活動目標）這個命令帶有一個 *process-id* 參數。要找到某個 UNIX 進程的 *process-id*，通常的方式是使用 ps 工具，或者使用“jobs –l”shell 命令。

attached 命令執行之後，第二次按 RET 的話，是不會被重複的。

為了使用 attached，程式必須運行在一個支援進程的環境下。例如：對於在缺乏操作的 bare-board 目標上的程式，attached 是不會工作的。也必須得有發送信號的許可權。

當使用attached的時候，調試器首先定位當前目錄中正運行在進程中的程式，然後（如果沒有發現這個程式）查看元檔搜索路徑（參見指定原始檔案目錄一節）。你也可以利用file命令裝載程式。參見指定文件的命令一節。

準備好要調試的指定進程後，所做的第一件事就是停止進程。檢查與修改被附著的進程，可以使用平常使用 run 啟動進程時能夠用到的所有命令。可以插入中斷點；可以單步調試並繼續；可以修改記憶體。如果你願意讓進程繼續運行的話，你可以在將 GDB 附著到進程之後使用 continue 命令。

detach

當對被附著的進程的調試完成時，可以使用 detach 命令，把它從 GDB 的控制下釋放出來。分離進程而繼續執行。detach 命令之後，那個進程與 GDB 再一次變得完全獨立，並且準備附著另外一個進程，或者使用 run 命令啟動一個。detach 命令執行之後，再按 RET 的話，是不會被重複的。

當已附著到一個進程時，退出GDB或使用run命令，會殺掉那個進程。默認情況下，GDB

會詢是否嘗試這麼做的確認，可以使用set confirm命令控制是否需要確認。（參見可選的警告與消息一節）。

4.8 殺掉子進程

kill 殺掉運行在 GDB 下的程式的子進程。

這個命令在你想要調試一個 core dump，而不是一個進程時很有用。程式運行時，GDB 會忽略所有的 core dump。

在某些作業系統上，當你在 GDB 中給某個程式設置了中斷點，那麼這個程式就不能在 GDB 以外被執行了。可以在這裏使用 kill 命令，准許在 GDB 以外運行程式。

kill 在想要重新編譯或重新連接程式的時候也很有用，因為在大多數系統上是不允許修改正在運行的進程的可執行檔的。在這種情況下，當你下次敲入 run 時，GDB 會通知那個檔已經被修改了，並重新讀取符號檔（會儘量維持當前的中斷點設置）。

4.9 多線程程式的調試

在某些作業系統中，例如 HP-UX 和 Solaris，一個單獨的程式可能擁有不止一個線程。一個作業系統對於另外一個作業系統，現程的精確語義是不相同的，但是大體上，單個程式的線程，除了共用同一個位址空間之外（更確切地說，它們都可以訪問並修改同一個變數），與多進程近似。從另外一個角度講，每一個線程擁有自己的寄存器和執行棧，而且也可能有自己專用的記憶體。

GDB 為調試多線程程式提供了如下功能：

新線程的自動通知。

“thread *threadno*”，線上程間進行切換的命令。

“info threads”，查詢現有線程的命令。

“thread apply [*threadno*] [*all*] *args*”，讓一列線程應用某個命令的命令

線程特有的中斷點。

警告：這些功能並不是在每一個作業系統支援線程的 GDB 配置中都可用。你的 GDB 不支持線程的話，這些命令是不會起作用的。例如，有一個不支援線程的系統，來自“info

threads”命令的輸入沒有顯示，而且總是拒線程命令，就像下面這樣：

```
(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 的線程調試工具可以讓你觀察運行在你的程式中的所有線程——有個特殊的線程總會是調試的焦點所在，而且任何時候 GDB 都要獲得對它的控制權。這個線程就被稱作當前線程。調試命令透過對當前線程的觀察來顯示程式的資訊。

只要 GDB 檢測到程式中有新線程，它就使用“[New *systag*]”格式的一個消息顯示目標系統對於這個線程的標識。*systag* 是一個線程表示符，各式會因具體的系統而不同。例如，在 LynxOS 上，當 GDB 通知有一個新線程時，你看到的可能會是這樣的消息：

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

相對的，在一個 SGI 系統上，*systag* 就很簡單，如“process 3 68”，沒有過多的限定詞。

為了調試的目的，GDB 使用它自己的線程號——總是一個單獨的整數——對應於程式中的各個線程。

info thread

顯示目前程式中所有線程的一個摘要。GDB 對於每一個線程的顯示（按這種順序）：

1． 由 GDB 分配的線程號

2． 目標系統的線程識別字（*systag*）

3． 對應線程當前的堆疊幀的摘要資訊

GDB 線程號左邊有一個“*”號，表明這是當前線程。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 系統上：

為了調試的目的，GDB 使用它自己的線程號——按照線程創建順序分配的一個小整數——對應於程式中的各個線程。

只要 GDB 檢測到了程式中有新的線程，它就使用“[New *systag*]”格式的一個消息顯示目標系統對於這個線程的標識。*systag* 是一個線程表示符，各式會因具體的系統而不同。例如，在 HP-UX 系統上，當 GDB 通知有一個新線程時，你看到的可能會是這樣的消息：

[New thread 2 (system thread 26594)]

info thread

顯示目前程式中所有線程的一個摘要。GDB 對於每一個線程的顯示（按這種順序）：

1． 由 GDB 分配的線程號

2． 目標系統的線程識別字（*systag*）

3． 對應線程當前的堆疊幀的摘要資訊

GDB 線程號左邊有一個“*”號，表明這是當前線程。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 系統上，使用一個 Solaris 專有命令，能夠顯示更多有關用戶線程的資訊：

maint info sol-threads

顯示 Solaris 上的用戶線程資訊。

thread *threadno*

使線程號為 *threadno* 線程成為當前線程。這個命令的參數 *threadno* 是 GDB 內部線程編號，也就是上述“info thread”所顯示的那個。GDB 以顯示你所選擇的線程的識別字，以及它目前的堆疊幀的摘要作為應答：

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

同樣帶有“[New …]”消息，Switching to 後面的文字格式依賴於你的作業系統所規定的線程標識。

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

thread apply 命令使你可以給一個或多個線程應用某個命令。利用命令參數 *threadno* 指定你想要影響的線程的編號。*threadno* 是 GDB 內部線程編號，也就是上述“info thread”所顯示的那個。要對所有線程應用某個命令，使用 thread apply all *args*。

只要 GDB 停止了你的程式，不管是因為某個中斷點還是某個信號，它都會自動地選擇那個發生中斷點或信號的線程。GDB 用一個“[Switching to systag]”格式消息提醒你注意上下文切換，鑒別線程。

參見停止與啟動多線程程式一節，瞭解更多有關停止和啟動某個帶有多個線程的程式時，GDB是如果工作的資訊。

參見設置觀察點一節，瞭解有關帶有多個線程的程式中的觀察點資訊。

4.10 多進程程式的調試

在大多數系統上，GDB 沒有對調試那些利用 fork 函數創建額外進程的程式提供特殊支援。當某個程式 forks 時，GDB 會繼續調試父進程，而不會對子進程的運行有所妨礙。如果你已在子進程能夠執行到的任意代碼中設置了一個中斷點話，那麼子進程將會獲得一個SIGTRAP 信號，這（除非它獲得了這個信號）會導致它的中斷。

不管怎樣，你要是想調試子進程，還是有一個不算太費力的工作區的。在子進程fork之後要執行的代碼中放置一個sleep調用。只有當某個環境變數被設置了，或者已存在某個檔，那也許會對sleep有作用，所以，你不想在子進程上運行GDB時，那個延遲就沒有必要發生。當子進程正睡眠時，利用ps程式獲得它的進程ID。然後告訴GDB（要是還想調試父

進程的話，就啟動一個新的GDB）附著到這個子進程上（參見4.7 調試某個已運行的進程一節）。從那一點開始，你就可以像調試其他附著到的進程一樣，調試這個子進程了。

在某些系統上，GDB 對那些利用 fork 或者是 vfork 函數創建額外進程的程式提供了支援。目前，僅有的帶有這一特性的幾個平臺是：HP-UX（僅 11.x 及後續版本？）和 GNU/Linux（2.5.60 及後續版本）。

一般情況下，當程式 forks 時，GDB 會繼續調試父進程，而不會對子進程的運行有所妨礙。

如果你想跟隨子進程而不是父進程的話，就使用命令 set follow-fork-mode。

set follow-fork-mode

設置調試器回應程式對 fork 或 vfork 的調用。一個 fork 或 vfork 的調用創建一個新的進程。*mode* 參數可以是：

parent

fork 之後的原有進程被調試。子進程的運行不受妨礙。這也是默認的。

child

fork 之後的新進程被調試。父進程的運行不受妨礙。

show follow-fork-mode

顯示調試器目前對 fork 或 vfork 調用的響應。

假如你要求調試子進程，同時 vfork 之後又跟著一個 exec 的話，GDB 執行那個新的目標直到遇到目標中的第一個中斷點。如果在原有程式的 main 中有一個中斷點的話，那麼這個中斷點也會被設置在子進程的 main 中。

當子進程被 vfork 產生時，直到 exec 調用完成之前，既不能對子進程調試也不能對父進程調試。

如果在exec調用執行之後，你對GDB發出了一個run命令，這個新的目標重新啟動。要重新啟動父進程，須利用以父進程可執行檔案名為參數的file命令。可以使用catch命令，只要有fork，vfork或exec調用產生，就讓GDB停止。參見設置捕獲點一節。

5.0 停止與繼續

使用調試器的首要目的是為了在程式終止之前停止程式；抑或為了程式運行有問題時，可以探查並尋找原因。

在 GDB 內部，程式可以因為多種因素而停止，如信號、中斷點，或者是使用 step 這樣的命令後到達了新的一行。然後可以查看和更改變數，設置新的中斷點或去除老的中斷點，然後繼續執行。通常，GDB 所顯示的消息對程式的狀態都提供了充分的說明——也可以在任時候顯示的請求這些資訊。

info program

顯示有關程式狀態的資訊：是否在運行、它的進程是什麼，還有他為什麼停止了。

5.1 中斷點、觀察點和捕捉點

只要到達了程式中的某一中斷點，程式就會停止。對於每一個中斷點，都可以附加一些條件，在細節上控制程式的停止與否。可以利用break命令設置中斷點和它的變體（參見設置中斷點一節），用行號、函數名或者是程式中的確切位址，指定程式應在何處停止。

在有些系統上，在可執行檔運行以前，就可以在共用庫中設置中斷點。在 HP-UX 系統上會有一點限制：要想在那些不被程式直接調用的共用庫常式（例如，作為 pthread_create 調用的參數的常式）中設置中斷點，必須等到可執行檔運行才行。

觀察點是一種特殊的中斷點，當某個運算式的值改變時，停止程式。必須使用一個不同的命令來設置觀察點（參見設置觀察點一節），但除此以外，就可以向管理中斷點那樣管理觀察點：可以利用相同的命令啟用，停用和刪除觀察點和中斷點。

可以安排程式中的某些值，只要GDB停在了某個中斷點上，就可以自動地被顯示出來（參

見自動顯示一節）。

捕獲點是另外一種特殊的中斷點，當有某些事件發生時，停止程式，例如拋出C++異常，或者某個庫的載入。與觀察點一樣，使用不同的命令設置捕獲點，但除此以外，可以像管理中斷點那樣管理捕獲點。（要使程式在接收到某個信號時停止，使用handle命令;參見信號一節）。

當創建一個中斷點、觀察點或者捕獲點時，GDB 會為他們分配一個編號；這些編號是從 1 開始的連續整數。在許多的控制中斷點不同特性的命令中，使用編號說明要改變哪一個中斷點。要是被停用了，直到再次啟用前是不會對程式有任何影響的。

有些 GDB 命令可以接受某一範圍的中斷點，對其操作。一個中斷點範圍即可以使一個單獨的編號，像“5”，也可以是兩個編號，遞增順序，中間用橫線連接，像“5-7”。當某個中斷點範圍指定給了某個命令時，在那範圍以內的所有中斷點都會受到影響。

5.1.1 設置中斷點

中斷點由break（簡寫b）命令設置。調試器便利變數（debugger convenience variable）“$bpnum”記錄了最近設置的中斷點的數量；參見便利變數一節，討論了使用便利變數都可以做什麼。

有多種方法說明中斷點的去向：

```
break function
```

在函數*function*的入口處設置中斷點。當使用允許符號重載的源語言時，如C++，*function*可能會涉及到多個可能的中斷位置。參見中斷點菜單一節，對於這種情形的討論。

break +offset

break -offset

從當前選取的堆疊幀的執行停止處，向前或向後若干行處設置一個中斷點。參見幀一節，對於堆疊幀的描述。

break linenum

在當前原始檔案的第 *linenum* 行處設置一個中斷點。當前原始檔案即最後那個被列印出根源程式文本的檔。中斷點會正好在執行那以行的任何代碼之前停止程式。

break filename:linenum

在原始檔案 *filename* 的 *linenum* 行處設置一個中斷點。

break filename:function

在檔 filename中找到的函數 function的入口處設置一個中斷點。即指定檔案名又指定函數名是多餘的，除非是有多個檔中包含了相同名稱的函數。

break *address

在位址 address處設置一個中斷點。可以使用這個命令，在程式中那些沒有調試資訊或原始檔案的部分中設置中斷點。

break

當不使用任何參數調用的時候，break在被選取的堆疊幀內的下一個要執行的指令處設置一個中斷點（參見檢查堆疊一節）。在被選取的任何堆疊幀內除最深處的以外，這會使得一旦控制回到那一幀，程式就會停止。這有些類似於被選擇幀的內部幀內的finish命令的作用——除了finish還沒有離開活動中斷點之外。要是在最內部的幀內使用不帶參數的break的話，GDB會在下一次到達當前位置時停止。這或許

會對內部迴圈有用。

break ... if cond

設置一個帶有條件cond的中斷點；每次達到中斷點時計算運算式cond的值，只有是非 0值時才停止，即條件為“真”時。“...”代表上述的指定在什麼位置中斷的參數之一（或者無參數）。要獲得更多有關中斷點條件的資訊，參見中斷條件一節。

tbreak args

設置僅允許停止一次的中斷點。args與break命令相同，而且中斷點設置的方法也相同，只不過程式首次在那裏停止以後，這個中斷點就會被刪除。參見停用中斷點一節。

hbreak args

設置一個硬體輔助的（hardware-assisted）中斷點。Args與break命令相同，而且中斷點設置的方法也相同，只不過中斷點需要硬體的支援，而且好多目標硬體可能沒有這方面的支持。這個命令的主要用途是EPROM/ROM代碼的調試，因為可以在一個指令上設置中斷點而不用更改這個指令。這個命令可以被用於帶有由SPARClite DSU和基於x86 所提供的新型trap-generation（陷阱發生）的目標。這些目標會在程式訪問某些已分配給調試寄存器的資料或指令位址時產生陷阱。不過，硬體中斷點寄存器只能持有有限數量的中斷點。例如，在DSU上，一次只能有兩個資料中斷點被設置，同時，如果超過了兩個，GDB會拒絕這個命令。在設置新中斷點之前，刪除或停用不用的中斷點（參見停用一節）。參見中斷條件一節。對於遠端目標，可以限定GDB要使用的硬體中斷點數量，參見設置遠端硬體中斷點限制。

thbreak *args*

設置一個僅允許停止一次的硬體輔助中斷點。Args與hbreak命令相同，而且使用方法也相同。不過，像tbreak命令一樣，程式首次在那裏停止後，這個中斷點就會被刪除。同樣，也像hbreak命令，需要硬體的支援，而且有些硬體可能沒有這方面的支持。參見停用中斷點一節，及中斷條件一節。

rbreak *regex*

在所有由正則運算式 *regex* 所匹配到的函數上設置中斷點。這個命令在所有匹配上設置一個無條件中斷點，列印它所設置的所有中斷點的一個列表。一旦設置了這些中斷點，它們僅僅會像由 break 命令設置的中斷點那樣被處理。可以刪除它們，停用它們，或者給它們限定條件，這些都跟操作所有其他中斷點的方法是一樣的。正則運算式的語法與“grep”這樣的工具所使用的是同一標準。注意這與 shell 所使用的語法不同，例如：foo* 配置包含一個 fo 後面跟著 0 個或多個 o 的函數。你所提供的是一個隱含了.*開頭與結尾的正則運算式，因此，僅匹配以 foo 開頭的函數，使用^foo。