Oracle® Developer Studio 12.5：使用 dbx 调试程序

文件号码 E71956
2016 年 6 月

ORACLE

文件号码 E71956

文档可访问性

有关 Oracle 对可访问性的承诺，请访问 Oracle Accessibility Program 网站 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc。

获得 Oracle 支持

购买了支持服务的 Oracle 客户可通过 My Oracle Support 获得电子支持。有关信息，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info；如果您听力受损，请访问 http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs。

目录

使用本文档

- **概述** – 介绍如何使用 `dbx` 命令行调试器（交互式源代码级调试工具）。
- **目标读者** – 应用程序开发者、系统开发者、架构师、支持工程师
- **必备知识** – 熟悉 Fortran、C、C++ 或 Java 编程语言，并对 Oracle Solaris 操作系统或 Linux 操作系统以及 UNIX® 命令有一定的了解。

产品文档库

有关该产品及相关产品的文档和资源，可从以下网址获得：`http://www.oracle.com/pls/topic/lookup?ctx=E71940`。

反馈

可以在 `http://www.oracle.com/goto/docfeedback` 上提供有关本文档的反馈。

♦ ♦ ♦ 第 1 章

1 dbx 入门

dbx 是一个交互式源代码级命令行调试工具。可以使用它来以可控方式运行程序以及检查已停止程序的状态。使用 dbx 可以完全控制程序的动态执行过程，包括收集性能和内存使用情况数据、监视内存访问及检测内存泄漏。

您可使用 dbx 调试使用 C、C++ （包括 C++11 和 C11 标准）或 Fortran 编译的应用程序。也可以调试使用 Java™ 代码和 C JNI（Java Native Interface，Java 本地接口）代码或 C++ JNI 代码混合编写的应用程序，但存在一些限制（请参见“使用 dbx 调试 Java 代码的限制” [199]）。

dbxtool 为 dbx 提供图形用户界面。

本章提供使用 dbx 调试应用程序的基本知识。其中包含以下各节：

编译调试代码

在使用 dbx 对程序进行源代码级调试前，必须使用 -g 选项（C 编译器、C++ 编译器、Fortran 编译器和 Java 编译器均接受此选项）编译程序。dbx 还支持使用 C++11 和 C11 标准编写的代码。有关更多信息，请参见“编译调试程序” [41]。

启动 dbx 或 dbxtool 和装入程序

要启动 dbx，请在 shell 提示符处键入 dbx 命令：

```
$ dbx
```

要启动 dbxtool，请在 shell 提示符处键入 dbxtool 命令：

```
$ dbxtool
```

要启动 dbx 并装入要调试的程序：

```
$ dbx program-name
```

要启动 dbxtool 并装入要调试的程序：

```
$ dbxtool program-name
```

要启动 dbx 并装入 Java 代码和 C JNI 代码或 C++ JNI 代码混编的程序：

```
$ dbx program-name {.class | .jar}
```

可使用 dbx 命令启动 dbx 并通过指定进程 ID 将其连接到运行中的进程。

```
$ dbx - process-ID
```

可使用 dbxtool 命令启动 dbxtool 并通过指定进程 ID 将其连接到运行中的进程。

```
$ dbxtool - process-ID
```

如果不知道进程的进程 ID，请在 dbx 命令中加入 pgrep 命令来查找并连接至进程。例如：

```
$ dbx - `pgrep Freeway`
Reading -
Reading ld.so.1
Reading libXm.so.4
Reading libgen.so.1
Reading libXt.so.4
Reading libX11.so.4
Reading libce.so.0
Reading libsocket.so.1
Reading libm.so.1
Reading libw.so.1
Reading libc.so.1
Reading libSM.so.6
Reading libICE.so.6
Reading libXext.so.0
Reading libnsl.so.1
Reading libdl.so.1
Reading libmp.so.2
Reading libc_psr.so.1
Attached to process 1855
stopped in _libc_poll at 0xfef9437c
0xfef9437c: _libc_poll+0x0004:   ta     0x8
Current function is main
   48   XtAppMainLoop(app_context);
```

```
(dbx)
```

有关 dbx 命令和启动选项的更多信息，请参见“dbx 命令” [289] 和 dbx(1) 手册页，或键入 dbx -h。

如果 dbx 已在运行，可使用 debug 命令装入要调试的程序，或从正在调试的程序切换到其他程序：

```
(dbx) debug program-name
```

要装入或切换到包含 Java 代码和 C JNI 代码或 C++ JNI 代码的程序：

```
(dbx) debug program-name{.class | .jar

   dbx debug program-name{.class | .jar

 }
```

如果 dbx 已在运行，还可以使用 debug 命令将 dbx 连接到运行中的进程：

```
(dbx) debug program-name process-ID
```

将 dbx 连接到包含 Java 代码和 C JNI（Java Native Interface，Java 本地接口）代码或 C++ JNI 代码的正在运行的进程：

```
(dbx) debug program-name{.class | .jar} process-ID
```

有关更多信息，请参见“debug 命令” [291]。

在 dbx 中运行程序

要在 dbx 中运行最近装入的程序，请使用 run 命令。如果最初键入 run 命令时没有使用参数，则程序便在没有参数的情况下运行。要传递参数或重定向程序的输入或输出，请使用下列语法：

```
run [ arguments ] [ < inputfile ] [ > output-file ]
```

例如：

```
(dbx) run -h -p < input > output
Running: a.out
(process id 1234)
execution completed, exit code is 0
(dbx)
```

运行包含 Java 代码的应用程序时，运行参数传递给 Java 应用程序而不是 JVM 软件。不要把主类名当作参数。

如果重复执行 run 命令时没有使用参数，程序重新启动时使用上一个 run 命令中的参数或重定向。可以使用 rerun 命令重置选项。有关 run 命令的更多信息，请参见“run 命令” [336]。有关 rerun 命令的更多信息，请参见“rerun 命令” [334]。

应用程序可能会运行完毕或正常终止。如果设置了断点，程序可能会在断点处停止。如果应用程序中有错误，会因内存故障或段故障而停止。

使用 dbx 调试程序

可能出于下列原因之一调试程序：

- 为了确定程序在何处以及为何导致崩溃。确定崩溃原因的方法包括：
  - 在 dbx 中运行程序。dbx 会报告崩溃的发生位置。
  - 检查信息转储文件并查看堆栈跟踪。请参见“检查信息转储文件” [28] 和“查看调用堆栈” [31]。
- 为了确定程序为何返回错误结果。其方法包括：
  - 设置用于停止执行的断点，以便可以检查程序的状态以及查看变量值。请参见“设置断点” [29] 和“检查变量” [32]。
  - 采用一次执行一行源代码的方式单步执行代码，监视程序状态的变化情况。请参见“单步执行程序” [30]。
- 为了查找内存泄漏或内存管理问题。执行运行时检查 (Runtime Checking, RTC) 可以检测运行时错误（例如，内存访问错误和内存泄漏错误），以及监视内存使用情况。请参见“查找内存访问问题和内存泄漏” [32]。

检查信息转储文件

要确定程序发生崩溃的位置，可能需要检查信息转储文件，即程序崩溃时的程序内存映像。可使用 where 命令确定程序在转储信息时的执行位置。请参见“where 命令” [370]

注 - dbx 无法像对待本机代码那样通过信息转储文件来指明 Java 应用程序的状态。

要调试信息转储文件，请键入：

```
$ dbx program-name core
```

或

```
$ dbx - core
```

在下面的示例中，程序因段故障和转储核心而崩溃。首先，dbx 从装入的信息转储文件开始。然后，where 命令显示堆栈跟踪，其中显示在 foo.c 文件的第 9 行发生崩溃。

```
% dbx a.out core
Reading a.out
core file header read successfully
Reading ld.so.1
Reading libc.so.1
Reading libdl.so.1
Reading libc_psr.so.1
program terminated by signal SEGV (no mapping at the fault address)
Current function is main
    9       printf("string '%s' is %d characters long\n", msg, strlen(msg));
(dbx) where
  [1] strlen(0x0, 0x0, 0xff337d24, 0x7efefeff, 0x81010100, 0xff0000), at
0xff2b6dec
=>[2] main(argc = 1, argv = 0xffbef39c), line 9 in "foo.c"
(dbx)
```

有关调试信息转储文件的更多信息，请参见“调试信息转储文件” [36]。有关使用调用堆栈的更多信息，请参见“查看调用堆栈” [31]。

注 - 如果程序与共享库动态链接，在创建信息转储文件的操作环境中调试该文件。有关如何调试在不同的操作环境中创建的信息转储文件的信息，请参见“调试不匹配的信息转储文件” [37]。

设置断点

断点是程序中要暂时停止程序的执行并让 dbx 进行控制的位置。在程序内怀疑存在错误之处设置断点。如果程序崩溃，请确定崩溃的发生位置，然后在这部分代码前设置断点。

程序在断点处停止时，便可以检查程序的状态和变量值。使用 dbx 可以设置多种类型的断点（请参见“使用 Ctrl+C 停止进程” [85]）。

最简单的断点类型就是停止断点。可以设置用于在函数或过程中停止的停止断点。例如，要在调用 main 函数时停止：

```
(dbx) stop in main
(2) stop in main
```

有关 stop in 命令的更多信息，请参见第 6 章 设置断点和跟踪和“stop 命令” [344]。

也可以设置用于在源代码的特定行处停止的停止断点。例如，要在源文件 t.c 中的第 13 行处停止：

```
(dbx) stop at t.c:13
(3) stop at "t.c":13
```

有关 stop at 命令的更多信息，请参见“在源代码行设置断点” [88]和“stop 命令” [344]。

可以使用 file 命令设置当前文件并使用 list 命令列出要在其中停止的函数来确定要停止在那里的行。然后使用 stop at 命令在源代码行设置断点：

```
(dbx) file t.c
(dbx) list main
10    main(int argc, char *argv[])
11    {
12        char *msg = "hello world\n";
13        printit(msg);
14    }
(dbx) stop at 13
(4) stop at “t.c”:13
```

要使程序在断点处停止后继续执行，请使用 cont 命令（请参见“继续执行程序” [82]和“cont 命令” [287]）。

要显示所有当前断点的列表，请使用 status 命令：

```
(dbx) status
(2) stop in main
(3) stop at "t.c":13
```

现在如果运行程序，程序将在第一个断点处停止：

```
(dbx) run
...
stopped in main at line 12 in file "t.c"
12        char *msg = "hello world\n";
```

单步执行程序

程序在断点处停止后，可能希望按一次执行一个源代码行的方式执行程序，在此时比较程序的实际状态与预期状态。可以使用 step 和 next 命令来执行此操作。这两个命令都是执行程序的一个源代码行，当执行完相应行时即停止。但在处理包含函数调用的源代码行时有所差别： step 命令步入函数，而 next 命令步过函数。

step up 命令会一直执行，直至当前函数将控制权返回给调用它的函数为止。

step to 命令会尝试步入当前源代码行中的指定函数；如果未指定任何函数，则尝试步入由当前源代码行的汇编代码确定调用的最后一个函数。

某些函数（特别是 printf 之类的库函数）可能未使用 -g 选项编译，因此 dbx 无法步入这些函数。在这种情况下，step 和 next 执行功能相似。

以下示例说明如何使用 step 和 next 命令以及在“设置断点” [29]中设置的断点。

```
(dbx) stop at 13
(3) stop at "t.c":13
(dbx) run
Running: a.out
```

```
stopped in main at line 13 in file "t.c"
   13           printit(msg);
(dbx) next
Hello world
stopped in main at line 14 in file "t.c"
   14   }

(dbx) run
Running: a.out
stopped in main at line 13 in file "t.c"
   13           printit(msg);
(dbx) step
stopped in printit at line 6 in file "t.c"
    6           printf("%s\n", msg);
(dbx) step up
Hello world
printit returns
stopped in main at line 13 in file "t.c"
   13           printit(msg);
(dbx)
```

有关单步执行程序的更多信息，请参见“单步执行程序” [81]。有关 `step` 和 `next` 命令的更多信息，请参见“step 命令” [341] 和“next 命令” [322]。

查看调用堆栈

调用堆栈代表当前处于活动状态的所有例程，即那些已调用但尚未返回各自调用方的例程。在该堆栈中，函数及其参数按其调用顺序存放。堆栈跟踪显示程序流中执行停止位置及执行到达此点的过程。它提供了有关程序状态的最简明的描述。

要显示堆栈跟踪，请使用 `where` 命令：

```
(dbx) stop in printf
(dbx) run
(dbx) where
  [1] printf(0x10938, 0x20a84, 0x0, 0x0, 0x0, 0x0), at 0xef763418
=>[2] printit(msg = 0x20a84 "hello world\n"), line 6 in "t.c"
  [3] main(argc = 1, argv = 0xefffe93c), line 13 in "t.c"
(dbx)
```

对于使用 `-g` 选项编译的函数，参数名及其类型是已知的，因此会显示精确的值。对于无调试信息的函数，显示的参数值是十六进制数。这些数字未必都有意义。例如，在上述堆栈跟踪中，帧 1 所示为 SPARC 输入寄存器 `$i0` 至 `$i5` 的内容，但仅寄存器 `$i0` 至 `$i1` 的内容有意义，因为只有两个参数传递到“单步执行程序” [30] 所示的示例中的 `printf`。

可以在未使用 `-g` 选项编译的函数中停止。在此类函数中停止时，`dbx` 在堆栈内向下搜索其函数是使用 `-g` 选项编译的第一帧（本例中为 `printit()`），并为其设置当前作用域。这用箭头符号 (`=>`) 表示。

有关调用堆栈的更多信息，请参见“效率考虑事项” [99]。有关当前作用域的更多信息，请参见“程序作用域” [63]。

检查变量

虽然堆栈跟踪可能包含足够的信息，可以完全表明程序的状态，但仍可能需要查看更多变量的值。print 命令可以求表达式的值，并根据表达式的类型输出值。以下示例中例举了几个简单的 C 表达式：

```
(dbx) print msg
msg = 0x20a84 "Hello world"
(dbx) print msg[0]
msg[0] = 'h'
(dbx) print *msg
*msg = 'h'
(dbx) print &msg
&msg = 0xefffe8b4
```

可以使用数据更改断点跟踪变量和表达式的值何时发生变化（请参见“设置数据更改断点（监视点）” [91]）。例如，要在变量计数值更改时停止执行，请键入：

```
(dbx) stop change count
```

查找内存访问问题和内存泄漏

运行时检查由两部分组成： 内存访问检查及内存使用和泄露检查。访问检查将检查被调试应用程序是否不当使用了内存。内存使用和泄露检查包括跟踪所有仍存在的堆空间，然后在需要时或程序终止时，扫描可用数据空间以及识别无引用的空间。

可以使用 check 命令启用内存访问检查及内存使用和泄露检查。要仅启用内存访问检查：

```
(dbx) check -access
```

要启用内存使用和内存泄漏检查：

```
(dbx) check -memuse
```

启用所需的运行时检查类型后，运行程序。程序正常运行，但速度很慢，因为每次进行内存访问前都要检查其有效性。如果 dbx 检测到无效访问，便会显示错误的类型和位置。此时，可以使用 dbx 命令（如 where 命令）显示当前堆栈跟踪，也可以使用 print 命令检查变量。

注 - 不能对使用 Java 代码和 C JNI 代码或 C++ JNI 代码混编的应用程序使用运行时检查。

有关使用运行时检查的详细信息，请参见第 9 章 使用运行时检查。

退出 dbx

dbx 会话在启动 dbx 之后将持续运行，直到退出 dbx 为止。在 dbx 会话期间，可以连续调试任意数量的程序。

要退出 dbx 会话，请在 dbx 提示符下键入 **quit**。

```
(dbx) quit
```

如果启动 dbx 时提供进程 ID 将其连接到正在运行的进程，退出调试会话，该进程仍存在并继续运行。dbx 在退出会话之前执行隐式 detach。

有关退出 dbx 的更多信息，请参见“退出调试” [47].。

访问 dbx 联机帮助

dbx 包含一个帮助文件，可使用 help 命令进行访问：

```
(dbx) help
```

♦ ♦ ♦ 第 2 章

2 启动 dbx

本章说明如何启动、执行、保存、恢复和退出 dbx 调试会话。其中包含以下各节：

启动调试会话

启用 dbx 的方式取决于您所调试的内容、您的位置、您需要 dbx 执行的操作、您对 dbx 的了解程度以及您是否设置了任何 dbxenv 变量。

可以完全通过终端窗口中的命令行使用 dbx，也可以运行 dbxtool（dbx 的图形用户界面）。有关 dbxtool 的信息，请参见 dbxtool 手册页和 dbxtool 中的联机帮助。

启动 dbx 会话的最简单方法是在 shell 提示符下键入 dbx 命令或 dbxtool 命令。

要从 shell 中启动 dbx 并装入要调试的程序，请键入：

```
$ dbx program-name
```

或

```
$ dbxtool program-name
```

要启动 dbx 并装入 Java 代码和 C JNI 代码或 C++ JNI 代码混编的程序：

```
$ dbx program-name{.class | .jar}
```

Oracle Developer Studio 软件包括两个 dbx 二进制文件，一个是只能调试 32 位程序的 32 位 dbx，另一个是既可调试 32 位程序也可调试 64 位程序的 64 位 dbx。启动 dbx 时，它会决定执行哪一个二进制文件。在 64 位操作系统上，64 位 dbx 是缺省值。

注 - 在 Linux OS 上，64 位 dbx 无法调试 32 位程序。要在 Linux OS 上调试 32 位程序，必须使用 dbx 命令选项 -xexec32 启动 32 位 dbx 或设置 DBX_EXEC_32 环境变量。

在 64 位 Linux OS 上使用 32 位 dbx 时，如果结果是将执行 64 位程序，则不要使用 debug 命令或将 follow_fork_mode 环境变量设置为 *child*。退出 dbx 并启动 64 位 dbx 以调试 64 位程序。

有关 dbx 命令和启动选项的更多信息，请参见“dbx 命令” [289] 和 dbx(1) 手册页。

调试信息转储文件

如果转储核心的程序与所有共享库动态链接，请在创建信息转储文件的相同操作环境中调试信息转储文件。dbx 对调试“不匹配的”信息转储文件（例如，运行其他版本或修补程序级别的 Oracle Solaris 操作系统的系统上生成的信息转储文件）具有有限的支持。

注 - dbx 无法像对待本机代码那样通过信息转储文件来指明 Java 应用程序的状态。

在相同的操作环境中调试信息转储文件

要调试信息转储文件，请使用以下命令：

```
$ dbx program-name core
```

或

```
$ dbxtool program-name core
```

如果发出以下命令，dbx 会通过信息转储文件确定程序名称：

```
$ dbx - core
```

或

```
$ dbxtool - core
```

如果 dbx 已在运行，也可以使用 debug 命令调试信息转储文件：

```
(dbx) debug -c core program-name
```

如果用 - 替换程序名，dbx 将尝试从信息转储文件中提取程序名。如果在信息转储文件中未提供可执行文件的全路径名，则 dbx 可能找不到可执行文件。如果 dbx 找不到可执行文件，请在指示 dbx 装入信息转储文件时指定二进制文件的完整路径名。

如果信息转储文件不在当前目录下，可指定它的路径名（例如，/tmp/core）。

使用 where 命令可确定程序在进行信息转储时的执行位置。

调试信息转储文件时，也可以求变量和表达式的值来查看程序崩溃时的值，但不能求调用函数的表达式的值。尽管不能单步执行，但是可以设置断点，然后重新运行程序。

如果信息转储文件被截断

如果装入信息转储文件时存在问题，请检查是否有被截断的信息转储文件。如果在创建信息转储文件时将其最大允许大小设得太低，那么 dbx 将无法读取最终被截断的信息转储文件。在 C shell 中，可使用 limit 命令来设置允许的最大信息转储文件大小（请参见 limit(1) 手册页）。在 Bourne shell 和 Korn shell 中，应使用 ulimit 命令（请参见 limit(1) 手册页）。可以在 shell 启动文件中更改信息转储文件大小的限制，重新寻找启动文件，然后重新运行生成该信息转储文件的程序，以生成完整的信息转储文件。

如果信息转储文件不完整并且缺少堆栈段，则堆栈跟踪信息不可用。如果缺少运行时链接程序信息，则装入对象列表不可用。在这种情况下，您会收到 librtld_db.so 未初始化的错误消息。如果缺少轻量级进程 (light weight processe, LWP) 的列表，则任何线程信息、LWP 信息或堆栈跟踪信息都将不可用。如果运行 where 命令，将会报错，说明程序未处于活动状态。

调试不匹配的信息转储文件

有时信息转储文件在一个系统（核心主机）上创建，而您想在另一台计算机（dbx 主机）上装入该信息转储文件以进行调试。但这样做可能会引起两个与库有关的问题：

- 核心主机上的程序所使用的共享库与 dbx 主机上的共享库可能不相同。要获取该库相关的正确堆栈跟踪，需要在 dbx 主机上提供这些原始库。
- dbx 使用 /usr/lib 中的系统库来帮助了解系统上运行时链接程序和线程库的实现详细信息。可能还需要通过核心主机提供这些系统库，以便 dbx 能够了解运行时链接程序数据结构和线程数据结构。

用户库和系统库可随着修补程序以及主要的 Oracle Solaris 操作系统升级而改变，因此如果在收集信息转储文件之后但在信息转储文件上运行 dbx 之前安装修补程序，此问题仍会出现在同一台主机上。

当装入不匹配的信息转储文件时，dbx 可能会显示以下一条或多条错误消息：

```
dbx: core file read error: address 0xff3dd1bc not available
dbx: warning: could not initialize librtld_db.so.1 -- trying libDP_rtld_db.so
dbx: cannot get thread info for 1 -- generic libthread_db.so error
dbx: attempt to fetch registers failed - stack corrupted
dbx: read of registers from (0xff363430) failed -- debugger service failed
```

调试不匹配的信息转储文件时应注意：

- pathmap 命令不能识别 "/" 路径映射，因此不能使用以下命令：

  ```
  pathmap / /net/core-host
  ```

- pathmap 命令的单参数模式不能与装入对象路径名同时使用，因此请使用双参数（来源路径和目标路径）模式。
- 如果 dbx 主机使用的 Oracle Solaris 操作系统版本与核心主机相同或比它更新，那么调试信息转储文件时效果可能会更好，虽然这并不总是必要的。
- 可能需要的系统库如下：
  - 对于运行时链接程序：

    ```
    /usr/lib/ld.so.1
    ```

    ```
    /usr/lib/librtld_db.so.1
    ```

    ```
    /usr/lib/64/ld.so.1
    ```

    ```
    /usr/lib/64/librtld_db.so.1
    ```

  - 对于线程库，取决于您所使用的 libthread 执行：

    ```
    /usr/lib/libthread_db.so.1
    ```

    ```
    /usr/lib/64/libthread_db.so.1
    ```

    如果 dbx 在支持 64 位的 Oracle Solaris OS 版本上运行，您将需要 64 位版本的 *xxx*_db.so 库，因为这些系统库是作为 dbx 的一部分（而不是目标程序的一部分）装入和使用的。

    与 libc.so 或任何其他库一样，ld.so.1 库是信息转储文件映像的一部分，因此需要与创建信息转储文件的程序相匹配的 32 位 ld.so.1 库或 64 位 ld.so.1 库。
- 如果正在查看来自某个线程程序的信息转储文件，并且 where 命令未显示堆栈，请尝试使用 lwp 命令。例如：

```
(dbx) where
current thread: t@0
[1] 0x0(), at 0xffffffff
(dbx) lwps
o>l@1 signal SIGSEGV in _sigfillset()
(dbx) lwp l@1
(dbx) where
=>[1] _sigfillset(), line 2 in "lo.c"
  [2] _liblwp_init(0xff36291c, 0xff2f9740, ...
  [3] _init(0x0, 0xff3e2658, 0x1, ...
```

...

如果 LWP 的帧指针 (fp) 被破坏，则 lwp 命令的 -setfp 和 -resetfp 选项会很有用。这些选项在调试信息转储文件时发挥作用，此时 assign $fp=... 不可用。

缺少线程堆栈表明 thread_db.so.1 有问题，因此，您可能还需要尝试从核心主机中复制适当的 libthread_db.so.1 库。

▼ 消除共享库问题和调试不匹配的信息转储文件

1. **将 dbxenv 变量 core_lo_pathmap 设置为 on。**
2. **使用 pathmap 命令可指示信息转储文件的正确库的位置。**
3. **使用 debug 命令可装入程序和信息转储文件。**

   例如，假定核心主机的根分区已通过 NFS 导出，并且可以通过 dbx 主机上的 /net/core-host/ 访问，应使用以下命令装入 prog 程序和 prog.core 信息转储文件来进行调试：

```
(dbx) dbxenv core_lo_pathmap on
(dbx) pathmap /usr /net/core-host/usr
(dbx) pathmap /appstuff /net/core-host/appstuff
(dbx) debug prog prog.core
```

   如果没有导出核心主机的根分区，则必须手动复制这些库。不需要重新创建符号链接。（例如，您不必建立从 libc.so 到 libc.so.1 的链接，只需确保 libc.so.1 可用即可）。

使用进程 ID

将进程 ID 用作 dbx 命令或 dbxtool 命令的参数，可以将正在运行的进程连接到 dbx。

```
$ dbx programname process-ID
```

或

```
dbxtool program-name processD
```

将 dbx 连接到包含 Java™ 代码和 C JNI（Java Native Interface，Java 本地接口）代码或 C++ JNI 代码的正在运行的进程：

```
$ dbx program-name{.class | .jar} process-ID
```

您也可以在不知道程序名的情况下，使用进程 ID 连接进程。

```
$ dbx - processID
```

或

```
$ dbxtool - processID
```

由于 dbx 仍然不知道程序名称，因此无法将参数传递到 run 命令中的进程。

有关更多信息，请参见“将 dbx 连接到正在运行的进程” [80]。

dbx 启动序列

启动 dbx 时，如果不指定 -S 选项，dbx 将在目录 */install-dir*/lib 中查找已安装的启动文件 dbxrc。在 Oracle Solaris 平台和 Linux 平台上，缺省安装目录分别是 /opt/solstudio12.4 和 /opt/oracle/solstudio12.4。如果 Oracle Developer Studio 软件未安装在缺省目录中，dbx 将根据 dbx 可执行文件的路径派生出 dbxrc 文件的路径。

然后，dbx 在当前目录中搜索 .dbxrc 文件，然后在 $HOME 中进行搜索。通过使用 -s 选项指定文件路径，可以显式指定与 .dbxrc 不同的启动文件。有关更多信息，请参见“使用 dbx 初始化文件” [53]。

启动文件可以包含任何 dbx 命令，通常包含 alias 命令、dbxenv 命令、pathmap 命令以及 Korn shell 函数定义。但某些命令要求已经装入程序或已经连接进程。所有启动文件均在装入程序或进程之前装入。启动文件也可以使用 source 或 .（句点）命令查找其他文件。您还可以使用启动文件设置其他 dbx 选项。

dbx 在装入程序信息的同时，将输出一系列的消息，如 Reading *filename*。

完成程序装入后，dbx 进入就绪状态，访问程序的 main 块（对于 C 或 C++ 而言：main()；对于 Fortran 而言：MAIN()）。一般来说，应设置断点（例如，stop in main），然后对 C 程序发出 run 命令。

设置启动属性

可使用 pathmap、dbxenv 和 alias 命令为 dbx 会话设置启动属性。

将编译时目录映射到调试时目录

缺省情况下，dbx 在编译程序的目录中查找与所调试的程序相关联的源文件。如果源文件或对象文件不在此目录下，或者所使用的计算机没有使用相同的路径名，您必须通知 dbx 这些文件的位置。

如果移动源文件或对象文件，可以将它们的新位置添加到搜索路径。pathmap 命令可创建从文件系统的当前视图到可执行映像中的名称的映射。该映射应用于源路径和对象文件路径。

向 .dbxrc 文件中添加公共 pathmap。

以下命令建立从目录 *from* 到目录 *to* 的新映射

(dbx) **pathmap [-c]** *from to*

如果使用 -c，该映射还将应用于当前工作目录。

pathmap 命令对于处理在不同主机上具有不同基路径的自动挂载和显式 NFS 挂载文件系统很有用。因为当前工作目录在自动挂载的文件系统中不准确，所以在尝试解决由自动挂载程序引起的问题时，请使用 -c。

缺省情况下，存在 /tmp_mnt 到 / 的映射。

有关更多信息，请参见“pathmap 命令” [327]。

设置 dbx 环境变量

可使用 dbxenv 命令列出或设置 dbx 定制变量。可以将 dbxenv 命令放置在 .dbxrc 文件中。

还可以设置 dbxenv 变量。有关 .dbxrc 文件以及设置这些变量的更多信息，请参见“使用 replay 保存和恢复” [51]。

有关更多信息，请参见“设置 dbxenv 变量” [54] 和“dbxenv 命令” [291]。

创建自己的 dbx 命令

可使用 kalias 或 dalias 命令创建自己的 dbx 命令。有关更多信息，请参见“dalias 命令” [288]。

编译调试程序

必须用 -g 或 -g0 选项对程序进行编译，以便为使用 dbx 进行调试做好准备。

使用 -g 选项进行编译

-g 选项指示编译器在编译期间生成调试信息。

例如，要使用 C++ 编译器进行编译：

```
% CC -g example_source.cc
```

对于 C++ 编译器：

- 单独使用 -g 选项（不指定优化级别）时，可以捕获调试信息并禁用函数的内联。
- -g 选项与 -O 选项或 -x*Olevel* 选项结合使用时，将打开调试信息，但不禁用函数的内联。该组选项将产生有限的调试信息和内联的函数。
- `--g0`（零）选项将打开调试信息，但不影响函数的内联。不能调试使用 `--g0` 选项编译的代码中的内联函数。`-g0` 选项可明显缩短链接时间和 `dbx` 启动时间（取决于程序对内联函数的使用）。

要编译将使用 `dbx` 调试的优化代码，请使用 -O（大写字母 O）和 -g 选项来编译源代码。

使用独立的调试文件

`dbx` 允许您在 `objcopy` 命令（在 Linux 平台上）和 `gobjcopy` 命令（在 Oracle Solaris 平台上）中使用选项，将调试信息从可执行文件复制到独立的调试文件、从可执行文件中删除该信息，以及在这两个文件之间创建链接。

`dbx` 按照以下顺序搜索独立的调试文件，并从找到的第一个文件中读取调试信息：

1. 包含可执行文件的目录。
2. 包含可执行文件的目录中名为 `debug` 的子目录。
3. 全局调试文件目录的子目录；如果 `dbxenv` 变量 `debug_file_directory` 设置为该目录的路径名，您可以查看或更改该目录。环境变量的缺省值为 `/usr/lib/debug`。

例如，以下过程介绍了如何为可执行文件 `a.out` 创建独立的调试文件。

▼ 如何创建独立的调试文件

1. **创建包含调试信息的、名为 `a.out.debug` 的独立调试文件。**

   ```
   objcopy --only-keep-debug a.out a.out.debug
   ```

2. **从 `a.out` 中删除调试信息：。**

```
objcopy --strip-debug a.out
```

3. 在两个文件之间建立链接。

```
objcopy --add-gnu-debuglink=a.out.debug a.out
```

在 Oracle Solaris 平台上，使用 `gobjcopy` 命令。在 Linux 平台上，使用 `objcopy` 命令。

在 Linux 平台上，可以使用 `objcopy -help` 命令来确定该平台是否支持 `-add-gnu-debuglink` 选项。可以使用 `cp a.out a.out.debug` 命令替换 `objcopy` 命令的 `-only-keep-debug` 选项，以便使 `a.out.debug` 成为完全可执行文件。

辅助文件（仅限 Oracle Solaris）

缺省情况下，装入对象包含可分配和不可分配节。可分配节是包含可执行代码和运行时该代码需要的数据的节。不可分配节包含补充信息，在运行时执行文件时这些信息不是必需的。这些节用于支持调试器和其他观察工具的操作。在运行时操作系统不会将对象中的不可分配节装入内存，因此，无论其大小如何，都不会影响内存使用或运行时性能的其他方面。

为方便起见，可分配节和不可分配节通常保留在同一文件中。但是，在某些情况下，将这些节分开会很有用。尤其是对高度优化的代码进行细粒度调试需要大量的调试数据。在现代系统中，调试数据可能很轻易地就大于其所描述的代码。32 位对象的大小限制为 4 GB。在非常大的 32 位对象中，调试数据可能会导致超出此限制，并阻止创建对象。

过去，为了解决这些问题，装入对象被剥离了若干不可分配节。剥离很有效，但是会销毁以后可能需要的数据。而 Oracle Solaris 链接编辑器可以将不可分配节写入辅助文件。此功能可通过 `-z ancillary` 命令行选项启用。

```
% ld ... -z ancillary[=outfile] ...
/* Your file is separated into a.out and b.out, where
a.out: ELF 32-bit LSB executable 80386 Version 1 [FPU], dynamically linked, not stripped,
 ancillary object b.out
b.out: ELF 32-bit LSB ancillary 80386 Version 1, primary object a.out */
```

缺省情况下，辅助文件和主输出对象同名，具有 `.anc` 文件扩展名。但是，可以将 *outfile* 值提供给 `-z ancillary` 选项来指定一个不同的名称。

注 - 辅助文件的 ELF 定义提供了单个主文件和任意数量的辅助文件。当前，Oracle Solaris 链接编辑器仅生成单个包含所有不可分配节的辅助文件。将来可能会有所更改。

指定了 `-z ancillary` 时，链接编辑器将执行以下操作。

- 将所有可分配节写入主文件。此外，将包含一个或多个设置了 `SHF_SUNW_PRIMARY` 节头标志的输入节的所有不可分配节写入主文件。

- 将所有其余的不可分配节写入辅助文件。
- 两个输出文件均收到以下已知的不可分配节完全相同的副本：

  .shstrtab　　节名称字符串表。

  .symtab　　完整非动态符号表。

  .symtab　　与 `.symtab` 关联的符号表扩展索引节。

  .strtab　　与 `.symtab` 关联的非动态字符串表。

  .SUNW_ancillary　　包含标识主对象、所有辅助对象和要检查的对象所需的信息。
- 主文件和所有辅助文件包含相同的节头数组。每个节在每个文件中具有相同的节索引。
- 尽管主文件和辅助文件均定义了相同的节头，但大多数节的数据将写入单个文件中，如上所述。如果给定文件中不存在某个节的数据，则会设置 `SHF_SUNW_ABSENT` 节头标志，且 `sh_size` 字段将为 0。

通过检查单个文件，该组织可获取节头的完整列表、完整的符号表以及主文件和辅助文件的完整列表。

然后，通过查找可执行文件中的辅助文件，`dbx` 可以如同 `dbx` 使用独立的调试文件一样使用这些辅助文件。按如下方式进行编译时，请使用 `-z ancillary` 选项：

```
%CC -g -z ancillary=a.out demo.cpp //"a.out" contains the ancillary object
```

主装入对象和所有相关的辅助文件包含一个允许标识所有装入对象并将其关联在一起的 .SUNW_ancillary 节。

有关更多信息，请参见《Oracle Solaris 11.3 链接程序和库指南》 中的 第 2 章, “链接编辑器”。

注 - 当前，此功能仅用于 Oracle Solaris 11.1。

压缩调试节（仅限 Oracle Solaris）

除了“辅助文件（仅限 Oracle Solaris）” [43]之外，`dbx` 还支持压缩调试节的调试。压缩调试节在压缩有时可能会比代码本身大的调试数据时很有用。此问题有时称为“DWARF 膨胀”问题。

压缩调试节是不可分配节，使用行业标准的 ZLIB 压缩库减小了大小。有关 ZLIB 的文档可以在 `http://www.zlib.net/` 上找到。调试器可识别输入对象中的压缩调试节，并自动解压缩这些节。此操作对调试器的用户是透明的，且无需特殊操作。

使用 -z compress-debug-sections 选项启用对输出文件中调试节的压缩。

```
$ cc -z compress-sections[=cmp-type] demo.cc
```

下面列出了 *cmp-type* 的可接受值：

zlib　　使用 ZLIB 压缩来压缩候选节。生成的输出节将设置 SHF_COMPRESSED 节标志来标识使用了压缩。如果未指定 *cmd-type*，则这是缺省值。有关 SHF_COMPRESSED 节标志的更多信息，请参见《Oracle Solaris 11.3 链接程序和库指南》 中的 “节压缩”。

zlib-gnu　　使用 ZLIB 压缩、使用 GNU 节压缩格式来压缩所有候选节。此格式要求候选节具有以 .debug 开头的名称。生成的输出节将重命名为以 .zdebug 开头来标识使用了压缩。

有关压缩调试节的更多信息以及使用压缩调试节的示例，请参见《Oracle Solaris 11.3 链接程序和库指南》 中的 “压缩调试节”。

调试优化代码

dbx 为优化代码提供部分调试支持。支持的程度主要取决于编译程序的方式。

在分析优化代码时，可以执行以下操作：

- 在任何函数的开始处停止执行（stop in *function* 命令）
- 计算、显示或修改参数
- 计算、显示或修改全局变量、局部变量或静态变量
- 从一行单步执行到另一行（next 或 step 命令）

如果在同时启用优化和调试的情况下（使用 -O 和 -g 选项）编译程序，dbx 将在限定模式下操作。

关于在什么情况下哪些编译器发出哪种符号信息的详细信息，随发行版的不同也会有所不同。

源代码行信息可用，但一个源代码行的代码可能会出现在优化程序的几个不同位置上，所以按源代码行在程序中单步执行会导致当前行位于源文件中的不同位置上，这取决于优化器如何调度代码。

当函数中的最后一个有效操作是调用另一个函数时，尾部调用优化会导致丢失堆栈帧。

对于 OpenMP 程序，使用 -xopenmp=noopt 选项进行编译即指示编译器不要应用任何优化。但是，为了实现 OpenMP 指令，优化器仍会处理代码，因此，使用 -xopenmp=noopt 编译的程序可能会出现所描述的一些问题。

参数与变量

通常，对于优化程序而言，参数、局部变量和全局变量的符号信息可用。结构、联合、C++ 类的类型信息，以及局部变量、全局变量和参数的类型和名称应该可用。

优化代码中有时会缺少有关参数和局部变量的位置的信息。如果 dbx 无法找到值，它会报告无法找到。有时该值可能会临时消失，因此请尝试再次单步执行和输出。

适用于基于 SPARC 的系统和基于 x86 的系统的 Oracle Developer Studio 12.2 编译器以及后来的 Oracle Developer Studio 更新版本将提供用于查找参数和局部变量的信息。较新的 GNU 编译器版本也提供此信息。

尽管在尚未发生最终寄存器至内存存储的情况下全局变量的值可能不准确，但您仍可以输出全局变量并为这些变量赋值。

内联函数

dbx 允许在内联函数上设置断点。当在调用方中执行内联函数的第一条指令时，控制将会停止。可以对内联函数执行的 dbx 操作（例如 step、next 和 list 命令）与非内联函数相同。

where 命令显示调用堆栈以及内联函数和参数（如果内联函数的位置信息可用）。

还支持对内联函数使用用于上下移动调用堆栈的 up 和 down 命令。

调用方的局部变量在内联框架中不可用。

寄存器（如果显示）来自调用方的窗口。

编译器可能进行内联的函数包括 C++ 内联函数、具有 C99 内联关键字的 C 函数以及编辑器认为可以提高性能的任何其他函数。

《Oracle Developer Studio 12.5：性能分析器》包含在调试优化程序时可能很有帮助的信息。

编译时未使用 -g 选项的代码

虽然大多数调试支持要求使用 -g 编译程序，但对未使用 -g 进行编译的代码，dbx 仍可以提供以下级别的支持：

- 栈回溯 （dbx where 命令）

- 调用函数（但没有参数检查）
- 检查全局变量

但请注意，除非用 -g 选项编译代码，否则 dbx 无法显示源代码。此限制也适用于使用 strip -x 的代码。

共享库要求使用 -g 选项以获得完全 dbx 支持

要获得完全支持，共享库也必须使用 -g 选项进行编译。如果利用没有使用 -g 选项进行编译的共享库模块来生成程序，则仍可以调试该程序。但由于未为这些库模块生成信息，所以无法获得完全 dbx 支持。

完全剥离的程序

dbx 能够调试已完全剥离的程序。这些程序包含一些可用来调试程序的信息，但只有外部可见函数可用。有些运行时检查对剥离的程序或装入对象有效。例如，内存使用检查与访问检查对使用 strip -x 剥离的代码有效，但对使用 strip 剥离的代码无效。

退出调试

dbx 会话在启动 dbx 之后将持续运行，直到退出 dbx 为止；在 dbx 会话期间，可以连续调试任意数量的程序。

要退出 dbx 会话，请在 dbx 提示符下键入 quit。

```
(dbx) quit
```

如果启动 dbx 时提供进程 ID 选项将其连接到正在运行的进程，退出调试会话时，该进程仍存在并继续运行。dbx 在退出会话之前执行隐式 detach。

停止进程执行

随时都可以通过按下 Ctrl+C 组合键停止执行进程，而无需退出 dbx。

从 dbx 中分离进程

如果已将 dbx 连接到一个进程，通过使用 detach 命令，无需中止进程或 dbx 会话便可从 dbx 中分离进程。

在临时应用其他基于 /proc 的调试工具（这些工具可能由于 dbx 独占访问而被阻止）时，可以分离进程并将其保留在停止状态。有关更多信息，请参见“从进程中分离 dbx” [81]。

有关更多信息，请参见“detach 命令” [294]。

中止程序而不终止会话

dbx kill 命令用于终止当前进程的调试和中止进程。但 kill 命令保留 dbx 会话，让 dbx 准备调试另一个程序。

中止程序是无需退出 dbx 即可消除正在调试的程序的剩余部分的好方法。有关更多信息，请参见“kill 命令” [311]。

保存和恢复调试运行

dbx 提供了三个命令，用于保存全部或部分调试运行并在稍后进行重放：

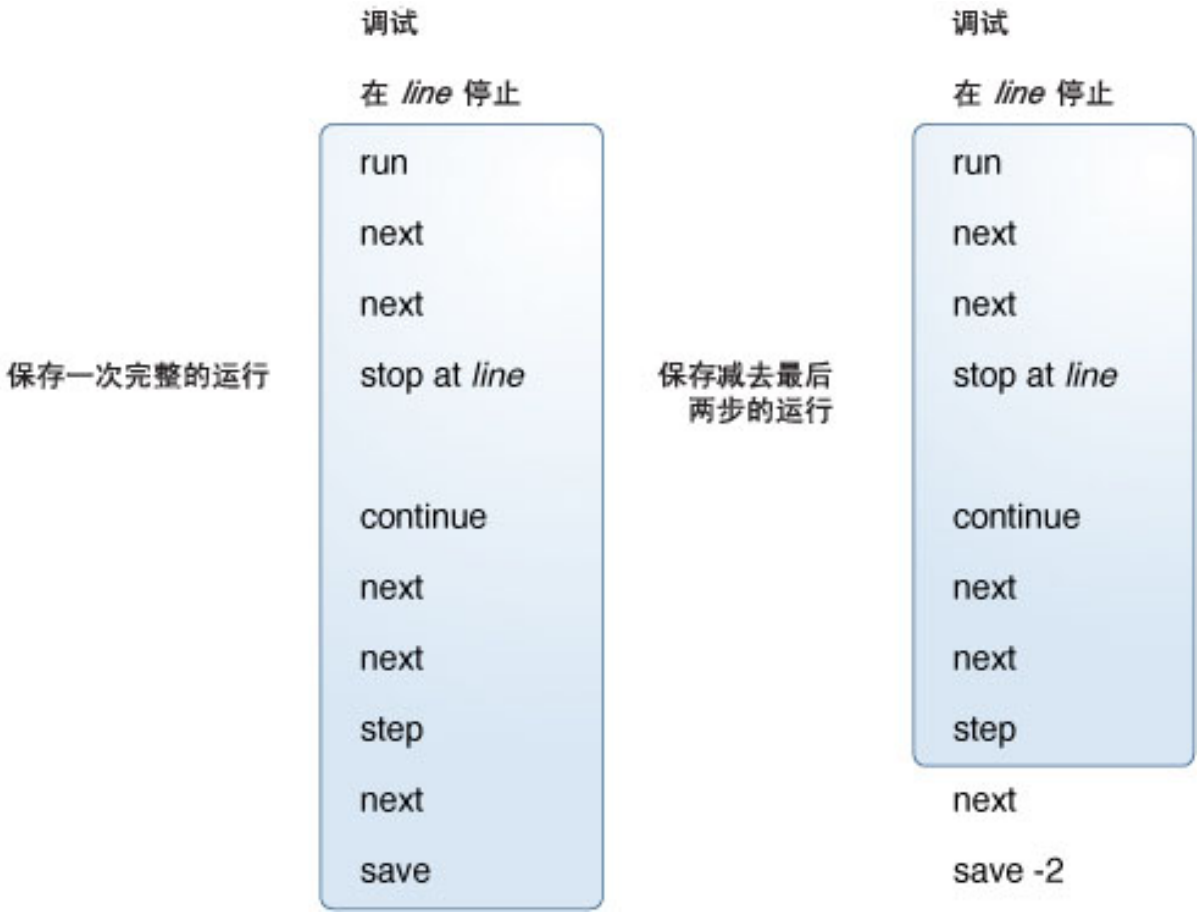
- save [*-number*] [*filename*]
- restore [*filename*]
- replay [*-number*]

使用 save 命令

save 命令将自上一 run 命令、rerun 命令或 debug 命令开始直到 save 命令期间所发出的所有调试命令保存到文件中。调试会话中的此段称为调试运行。

除了已发出的调试命令列表外，save 命令还保存运行开始时与程序状态相关联的调试信息，如断点、显示列表等等。当恢复已保存的运行时，dbx 将使用保存文件中的信息。

可以保存调试运行的一部分，即，整个运行从最后输入命令开始减去指定数目的命令。



如果不能确定要在何处结束正在保存的运行，可以使用 `history` 命令查看自会话开始以来发出的调试命令列表。

注 - 缺省情况下，`save` 命令将信息写入特定的特殊文件。如果要将调试运行保存到稍后可以恢复的文件，可使用 `save` 命令来指定文件名。请参见“将系列调试运行另存为检查点” [50]。

在要保存整个调试的点处发出 `save` 命令。

```
(dbx) save
```

要保存部分调试运行，请提供 *number* 选项，其中 *number* 是在 `save` 命令之前所不想保存的命令的个数。

```
(dbx) save -number
```

将系列调试运行另存为检查点

如果保存调试运行时没有指定文件名，dbx 将信息写入特定的特殊文件。每次保存时，dbx 都会覆盖此文件。但是，通过为 save 命令指定 *filename* 参数，可以将调试运行保存到文件中，这样，即使保存到 *filename* 之后又保存了其他调试运行，这个文件稍后也可以恢复。

保存一系列运行可为您提供一组检查点，从会话中较早的时间开始依次排列每个检查点。您可以恢复这些已保存运行中的任意一个运行，继续，然后将 dbx 重置到早期运行中所保存的程序位置和状态。

要将调试运行保存到非缺省文件，请提供文件名：

```
(dbx) save filename
```

恢复已保存的运行

保存运行后，可使用 restore 命令恢复该运行。dbx 使用保存文件中的信息。恢复运行时，dbx 首先将内部状态重置到运行开始时的状态，然后重新发出所保存的运行中的每个调试命令。

注 - source 命令还会重新发出存储在文件中的一组命令，但是不会重置 dbx 的状态。它只会从当前的程序位置重新发出命令列表。

要精确恢复已保存的调试运行，运行的所有输入必须完全相同： run 类型命令的参数、手动输入和文件输入。

注 - 如果在执行 restore 之前保存段，然后发出 run、rerun 或 debug 命令，restore 将使用第二个保存后 run、rerun 或 debug 命令的参数。如果这些参数不同，则不能进行精确恢复。

恢复已保存的调试运行

```
(dbx) restore
```

要恢复已保存到其他文件而不是缺省文件的调试运行：

```
(dbx) restore filename
```

使用 **replay** 保存和恢复

replay 命令是一个组合命令，相当于在 save -1 命令后立即发出一个 restore 命令。replay 命令带有负的 *number* 参数，此参数将传递给命令的 save 部分。缺省情况下，*-number* 的值为 -1，因此 replay 相当于撤消命令，可以恢复直到（但不包括）最后发出的命令为止的最后一个运行。

要重放当前的调试运行，但不包括最后发出的调试命令，请键入：

```
(dbx) replay
```

要重放当前调试运行并在特定命令前停止该运行，请使用 -number 选项，其中 *number* 是最后一个调试命令之前的命令个数。

```
(dbx) replay -number
```

♦ ♦ ♦ 第 3 章

3 定制 dbx

本章介绍可用来定制调试环境的某些属性的 dbxenv 变量，并说明如何使用初始化文件 .dbxrc 保留会话之间的更改和调整。

本章包含以下各节：

使用 dbx 初始化文件

dbx 初始化文件存储每次启动 dbx 时执行的 dbx 命令。通常，该文件包含定制调试环境的命令，但您可以将任何 dbx 命令放到该文件中。如果调试时是从命令行定制 dbx，这些设置将仅应用于当前调试会话。

.dbxrc 文件不应包含执行代码的命令。但您可以将此类命令放到一个文件中，然后使用 dbx source 命令执行该文件中的这些命令。

启动期间，搜索顺序为：

1. 安装目录（除非您将 -S 选项指定为 dbx 命令）*/install--dir*\/lib/dbxrc。在 Oracle Solaris 平台和 Linux 平台上，缺省安装目录分别是 /opt/solstudio12.4 和 /opt/oracle/solstudio12.4。如果 Oracle Developer Studio 软件未安装在缺省 *install-dir* 目录中，dbx 将根据 dbx 可执行文件的路径派生出 dbxrc 文件的路径。
2. 当前目录 ./.dbxrc
3. 起始目录 $HOME/.dbxrc

创建 .dbxrc 文件

创建包含常用定制和别名的 .dbxrc 文件

```
(dbx) help .dbxrc>$HOME/.dbxrc
```

然后，可使用文本编辑器对要执行的条目取消注释，来定制所生成的文件。

初始化文件样例

以下示例提供了一个样例 .dbxrc 文件：

```
dbxenv input_case_sensitive false
catch FPE
```

第一行更改区分大小写控制的缺省设置：

- dbxenv 是用来设置 dbxenv 变量的命令。有关 dbxenv 变量的完整列表，请参见“设置 dbxenv 变量” [54]。
- input_case_sensitive 是控制区分大小写的 dbxenv 变量。
- false 是 input_case_sensitive 的设置。

第二行是调试命令 catch，它用于将系统信号 FPE 添加到 dbx 可响应的一组缺省信号中，以停止程序。

设置 dbxenv 变量

可使用 dbxenv 命令设置用于定制 dbx 会话的 dbxenv 变量。

要显示特定变量的值：

(dbx) **dbxenv** *variable*

显示全部变量和变量值

(dbx) **dbxenv**

要显示变量的值：

(dbx) **dbxenv** *variable value*

表 1 “dbx 环境变量” 列出了可以设置的所有 dbxenv 变量。

表 1 dbx 环境变量

dbx 环境变量	变量功能说明
array_bounds_check on|off	如果设置为 on，dbx 将检查数组边界。缺省值：on。
c_array_op on | off	允许在 C 和 C++ 中进行数组操作。例如，如果 a 和 b 是数组，则可以使用 print a+b 命令。缺省值：off。
CLASSPATHX	为 dbx 指定由定制类装入程序装入的 Java 类文件的路径。

dbx 环境变量	变量功能说明
core_lo_pathmap on|off	控制 dbx 是否使用 pathmap 设置来定位 ismatchedcore 文件的正确库。缺省值：off。
debug_file_directory	设置全局调试文件目录。缺省值： /usr/lib/debug。
disassembler_versionautodetect|v8|v9|x86_32|x86_64	SPARC 平台： 设置适用于 SPARC V8 或 V9 的 dbx 的内置反汇编程序版本。缺省值是 autodetect，它根据运行 a.out 的计算机类型动态地设置模式。 x86 平台：设置适用于 x86_32 或 x86_64 的 dbx 的内置反汇编程序版本。缺省值是 autodetect，它根据运行 a.out 的计算机类型动态地设置模式。
event_safety on | off	保护 dbx 使其免受不安全的事件使用的影响。缺省值：on。
filter_max_length *num*	将通过美化输出过滤器转化为数组的序列最大长度设置为 *num*。
fix_verbose on|off	控制 fix 运行期间的编译行输出。缺省值：off。
follow_fork_inherit on|off	当跟随子进程时，确定是否继承断点。缺省值：off。
follow_fork_mode parent|child|both|ask	确定派生之后应跟随哪个进程；即，当前进程何时执行 fork、vfork 或 fork1。如果设置为 parent，则进程跟随父进程。如果设置为 child，则跟随子进程。如果设置为 both，则进程跟随子进程，但父进程保持活动状态。如果设置为 ask，则在检测到派生时询问应跟随哪个进程。缺省值：parent。
follow_fork_mode_inner unset|parent|child|both	如果将 follow_fork_mode 设置为 ask，且已选择 stop，则在检测到派生后设置此变量，无需使用 cont -follow。缺省值：unset。
input_case_sensitive autodetect| true|false	如果设置为 autodetect，dbx 将根据文件的语言自动选择区分大小写：对于 Fortran 文件，为 false；否则为 true。如果为 true，变量和函数名区分大小写；否则大小写无实际意义。缺省值：autodetect。
JAVASRCPATH	指定 dbx 查找 Java 源文件的目录。
jdbx_mode java| jni| native	存储当前 dbx 模式。有效设置为 java、jni 或 native。
jvm_invocation	通过 jvm_invocation 环境变量可以定制 JVM™ 软件的启动方式。（术语“Java 虚拟机”和 "JVM" 表示用于 Java™ 平台的虚拟机。）有关更多信息，请参见“定制 JVM 软件的启动” [204]。
language_mode autodetect|main|c| c++|fortran|fortran90	控制用于解析和计算表达式的语言。 ■ autodetect 将表达式语言设置为当前文件的语言。用于调试使用混合语言的程序（缺省）。 ■ main 将表达式语言设置为程序中主例程的语言。用于调试同类程序。 ■ c、c++、c++、fortran 或 fortran90 将表达式语言设置为选定语言。
macro_expand on | off	设置为 on 时，为选定表达式全局启用宏扩展。缺省值：on。
macro_source none | compiler | skim | skim_unless_compiler	管理 dbx 获得宏信息的位置。有关更多信息，请参见“略读 (skimming) 错误” [269]。缺省值：skim_unless_compiler。
mt_resume_one on | off | auto	如果设为 off，当使用 next 命令步过调用时将恢复所有线程以避免死锁。如果设为 on，当使用 next 命令步过调用时仅恢

dbx 环境变量	**变量功能说明**
	复当前线程。如果设为 auto，则与设为 off 时的行为相同，除非程序是一个事务管理应用程序并且您正在事务内执行单步操作，在这种情况下，仅恢复当前线程。缺省值：auto。
mt_scalable on|off	如果启用，dbx 在其资源使用方面更为保守，将能够使用 300 个以上的 LWP 调试进程。但是，此设置可能会导致速度明显减慢。缺省值：off。
mt_sync_tracking on | off	确定在 dbx 启动进程时是否启用对同步对象的跟踪。缺省值：off。
output_auto_flush on|off	每次调用后，自动调用 fflush()。缺省值：on
output_base 8|10|16|automatic	输出整型常量的缺省基数。缺省值： automatic（指针是十六进制字符，而其他都是十进制）。
output_class_prefix on | off	用于在输出类成员的值和声明时将一个或多个类名作为类成员的前缀。如果设置为 on，则为类成员添加前缀。缺省值：on。
output_data_member_only on|off	用于在输出类的定义时仅显示数据成员（dbx 命令 whatis -t -a）。如果设置为 on，则缺省值是 -a 选项（dbx 命令 whatis -t 将仅显示数据成员）。缺省值：off。
output_dynamic_type on|off	如果设置为 on，则输出监视和显示的缺省值是 -d。缺省值：off。
output_inherited_members on|off	如果设置为 on，则输出、显示和检查的缺省值是 -r。缺省值：off。
output_list_size *num*	控制 list 命令所输出的缺省行数。缺省值：10。
output_log_file_name *filename*	命令日志文件的名称。 缺省值：/tmp/dbx.log.*unique-ID* .
output_max_object_size *number*	设置输出变量的最大字节数。如果变量大小大于此数，则需要指定 -L 标志。此 dbxenv 变量适用于命令 print、display 和 watch。缺省值：4096。
output_max_string_length *number*	为 char *s 设置输出的字符数 (*number*)。缺省值：096。
output_no_literal on|off	如果启用，则当表达式是字符串 (char *) 时，仅输出地址，而不输出文字。缺省值：off。
output_pretty_print on|off	将 -p 设置为输出监视和显示的缺省值。缺省值：on。
output_pretty_print_fallback on|off	缺省情况下，出现问题时，美化输出会恢复为常规输出。如果希望诊断美化输出问题，请将此变量设置为 off 以防止出现这种回退。缺省值：on。
output_pretty_print_mode call | filter | filter_unless_call	确定使用何种美化输出机制。如果设置为 call，则使用调用样式的美化输出器。如果设置为 filter，则使用基于 python 的美化输出器。如果设置为 filter_unless_call，则先使用调用样式的美化输出器。
output_short_file_name on|off	显示文件的短路径名。缺省值：on。
overload_function on|off	对于 C++，如果设置为 on，则启用自动函数重载解析。缺省值：on。
overload_operator on|off	对于 C++，如果设置为 on，则启用自动运算符重载解析。缺省值：on。

dbx 环境变量	变量功能说明
pop_auto_destruct on|off	如果设置为 on，则当弹出帧时会自动为局部变量调用适当的析构函数。缺省值：on。
proc_exclusive_attach on|off	如果设置为 on，且已连接其他工具，将阻止 dbx 连接到进程。注意：如果多个工具连接到一个进程，当尝试对其进行控制时，可能会出现意外结果。缺省值：on。
rtc_auto_continue on|off	将错误记录到 rtc_error_log_file_name 并继续。缺省值：off。
rtc_auto_suppress on|off	如果设置为 on，则只报告一次指定位置的 RTC 错误。缺省值：on。
rtc_biu_at_exit on|off|verbose	在显式打开或由于 check -all 而打开内存使用检查时使用。如果值为 on，则在程序退出时生成一个非详细的内存使用（使用的块）报告。如果值为 verbose，则在程序退出时生成一个详细的内存使用报告。值为 off 时将不产生任何输出。缺省值：on。
rtc_error_limit *number*	要报告的 RTC 访问错误数。缺省值：1000。
rtc_error_log_file_name *filename*	记录 RTC 错误的文件名（如果设置了 rtc_auto_continue）。缺省值：/tmp/dbx.errlog。
rtc_error_stack on|off	如果设置为 on，堆栈跟踪将显示与 RTC 内部机制相对应的帧。缺省值：off。
rtc_inherit on|off	如果设置为 on，则对从调试程序执行的子进程启用运行时检查，并导致 LD_PRELOAD 环境变量被继承。缺省值：off。
rtc_mel_at_exit on|off|verbose	在内存泄露检查为 on 时使用。如果值为 on，则在程序退出时生成一个非详细的内存泄露报告。如果值为 verbose，则在程序退出时生成一个详细的内存泄漏报告。值为 off 时将不产生任何输出。缺省值：on。
run_autostart on|off	如果在没有活动程序时设置为 on，则 step、next、stepi 和 nexti 将隐式运行程序，并在语言相关的 main 例程处停止。如果设置为 on，则必要时 cont 表示 run。缺省值：off。
run_iostdio|pty	控制是否将用户程序的输入/输出重定向至 dbx 的 stdio 或特定 pty。pty 由 run_pty 提供。缺省值：stdio。
run_pty *ptyname*	当 run_io 设置为 pty 时，设置要使用的 pty 的名称。ptys 由图形用户界面包装器使用。
run_quick on|off	如果设置为 on，则不会装入任何符号信息。可使用 prog -readsysms 按需装入符号信息。在此之前，dbx 的行为如同所调试的程序被剥离。缺省值：off。
run_savetty on | off	dbx 与被调试程序之间的多路复用 TTY 设置、进程组和键盘设置（如果在命令行中使用了 -kbd）。用于调试编辑器和 shell。如果 dbx 获取了 SIGTTIN 或 SIGTTOU，并弹回到 shell，则将此变量设置为 on。将此变量设置为 off，可稍稍加快速度。如果 dbx 连接到了被调试的程序，或正在 Oracle Developer Studio IDE 中运行，则该设置无任何作用。缺省值：off。
run_setpgrp on | off	如果设置为 on，当程序运行时，setpgrp(2) 将在派生后立即被调用。缺省值：off。

dbx 环境变量	变量功能说明
scope_global_enums on | off	如果设置为 on，枚举器将被置于全局范围，而不是文件范围。请在处理调试信息 (~/.dbxrc) 之前进行设置。缺省值：off。
scope_look_aside on | off	如果设置为 on，则在当前范围之外查找文件静态符号。缺省值：on。
session_log_file_name *filename*	dbx 记录所有命令及其输出的文件名。输出将被附加至文件。缺省值："" （无会话记录）。
show_static_members	如果设置为 on，则用于输出、监视和显示的缺省值是 -S。缺省值：on。
stack_find_source on | off	如果设置为 on，当被调试程序在未使用 -g 编译的函数中停止时，dbx 将尝试查找并自动激活堆栈的第一帧。 缺省值：on。
stack_max_size *number*	设置 where 命令的缺省大小。缺省值：100。
stack_verbose on | off	控制 where 中参数和行信息的输出。缺省值：on。
step_abflow stop|ignore	如果设置为 stop，则在单步执行时，dbx 会在 longjmp()、siglongjmp() 和 throw 语句中停止。如果设置为 ignore，则 dbx 不检测 longjmp() 和 siglongjmp() 的异常控制流更改。缺省值：stop。
step_events on |off	如果设置为 on，则在使用 step 和 next 命令单步执行代码时允许断点。缺省值：off。
step_granularity statement | line	控制源代码行单步执行的粒度。如果设置为 statement，则以下代码： a(); b(); 执行两个 next 命令。如果设置为 line，将由一个 next 命令执行代码。在处理多行宏时，行的粒度是非常有用的。缺省值：statement。
suppress_startup_message *number*	设置版本级别，级别以下的启动信息不输出。缺省值：3.01。
symbol_info_compression on|off	如果设置为 on，则对于每个 include 文件，只读取一次调试信息。缺省值：on。
trace_speed *number*	设置跟踪执行的速度。其值是步骤之间暂停的秒数。缺省值：0.50。
track_process_cwd on|off	当设置为 on 且 GUI 连接到正在运行的进程时，当前工作目录将更改为正在运行的进程的工作目录。缺省值：off。
vdl_mode classic | lisp | xml	值描述语言 (Value Description Language, VDL) 用于将数据结构传达给 dbx 的图形用户界面 (graphical user interface, GUI)。classic 模式用于 Sun WorkShop™ IDE。Sun Studio 和 Oracle Developer Studio 发行版中的 IDE 使用 lisp 模式。xml 模式是实验性模式，不受支持。缺省值：值通过 GUI 设置。

dbxenv 变量和 Korn Shell

每个 dbxenv 变量就像 ksh 变量一样也可访问。通过在 dbxenv 变量前添加 DBX_ 前缀，可派生出 ksh 变量的名称。例如，dbxenv stack_verbose 和 echo $DBX_stack_verbose 可产生相同的输出。可以直接给变量赋值，也可以使用 dbxenv 命令给变量赋值。

♦ ♦ ♦ 第 4 章

4 查看和导航到代码

本章说明 dbx 如何导航到代码以及如何查找函数和符号。还说明如何使用命令导航到代码或查找标识符、类型和类的声明。

本章包含以下各节

导航到代码

每当正在调试的程序停止时，dbx 都会输出与停止位置关联的源代码行。在每个程序停止位置，dbx 会将当前函数的值重置为程序在其中停止执行的函数。在程序开始运行前及程序停止运行时，您可以移动或导航到程序中其他地方的函数和文件。可导航到任何函数或文件，只要它们是程序的一部分。导航会设置当前作用域（请参见“程序作用域” [63]）。这对于确定要在何时以及在哪一源代码行设置 stop at 断点非常有用。

导航到文件

可以导航到 dbx 将其识别为程序一部分的任何文件，即使模块或文件未使用 -g 选项进行编译也是如此。要导航到文件：

```
(dbx) file filename
```

使用不带参数的 file 命令将回显当前导航的文件名。

```
(dbx) file
```

如果不指定行号，dbx 会从文件的第一行开始显示文件。

```
(dbx) file filename ; list line-number
```

有关更多信息，请参见“在源代码行设置断点” [88]。

导航到函数

可以使用 func 命令导航到函数。键入命令 func，后跟函数名。例如：

```
(dbx) func adjust-speed
```

func 命令本身会回显当前函数。

有关更多信息，请参见“func 命令” [304]

从 C++ 二义函数名称列表中选择

如果尝试导航到的 C++ 成员函数具有二义性名称或重载函数名称，系统将显示一个列表，其中会列出具有重载名称的所有函数。键入要导航的函数的号码。如果您知道函数所属的具体类，则可以键入类名和函数名。例如：

```
(dbx) func block::block
```

在多个具体值中进行选择

如果可从同一作用域级别访问多个符号，则 dbx 会输出一条报告二义性的消息。

```
(dbx) func main
(dbx) which C::foo
More than one identifier 'foo'.
Select one of the following:
 0) Cancel
 1) "a.out"t.cc"C::foo(int)
 2) "a.out"t.cc"C::foo()
>1
"a.out"t.cc"C::foo(int)
```

在 which 命令的上下文中，从具体值列表中进行选择不会影响 dbx 或程序的状态。无论选择哪个具体值，dbx 都会回显名称。

输出源代码列表

使用 `list` 命令可输出文件或函数的源代码列表。在文件中导航后，`list` 会从第一行开始输出 *number* 行。缺省数量为 10 行。在函数中导航后，`list` 会输出其代码行。

有关详细信息，请参见“`list` 命令” [312]。

在调用堆栈中移动以导航到代码

如果存在活动进程，另一种导航到代码的方法是“在调用堆栈中移动”，即执行堆栈命令查看调用堆栈中当前存在的函数，这些函数代表当前处于活动状态的所有例程。在堆栈中移动会使当前函数和文件在您每次显示堆栈函数时发生变化。停止位置被视为位于堆栈的“底部”，因此，要离开该位置，请使用 `up` 命令，即向 `main` 或 `begin` 函数方向移动。使用 `down` 命令可向当前帧方向移动。

有关更多信息，请参见“堆栈中移动和返回起始位置” [102]。

程序位置的类型

`dbx` 使用三个全局位置来跟踪您正在检查的程序的各部分：

- 当前地址，由 `dis` 命令和 `examine` 命令使用和更新。
- 当前源代码行，由 `list` 命令使用和更新。该行号由可更改访问作用域的某些命令重置。有关更多信息，请参见“更改访问作用域” [65]。
- 当前访问作用域，这是一个复合变量，在“访问作用域” [64] 中有相关说明。对表达式求值期间使用访问作用域。它由 `line` 命令、`func` 命令、`file` 命令和 `list` 命令更新。

程序作用域

作用域是按变量或函数的可见性定义的程序子集。如果某个符号的名称在给定执行点是可见的，则称该符号“在作用域内”。在 C 语言中，函数可以具有全局或文件静态作用域；变量可以具有全局、文件静态、函数或块作用域。

反映当前作用域的变量

以下变量总是反映当前线程或 LWP 的当前程序计数器，而且不受更改访问作用域的各种命令的影响：

`$scope`	当前程序计数器的作用域
`$lineno`	当前行号
`$func`	当前函数
`$class`	`$func` 所属的类
`$file`	当前源文件
`$loadobj`	当前装入对象

这些变量仅在实时进程期间有用。

访问作用域

使用 `dbx` 检查程序的各种元素，需要修改访问作用域。`dbx` 在表达式求值期间使用访问作用域来实现解析二义符号等目的。例如，如果键入以下命令，`dbx` 会使用访问作用域来确定要输出哪个 `i`：

```
(dbx) print i
```

每个线程和 LWP 都有自己的访问作用域。在线程间切换时，每个线程都会返回其访问作用域。

访问作用域的组件

访问作用域的某些组件在以下预定义的 `ksh` 变量中是可见的：

`$vscope`	当前访问作用域
`$vloadobj`	当前访问装入对象
`$vfile`	当前访问文件
`$vlineno`	当前访问行号
`$vclass`	`$vfunc` 所属的类
`$vfunc`	当前访问函数

当前访问作用域的所有组件相互间保持兼容。例如，如果您访问不包含函数的文件，则当前访问源文件会更新为新的文件名，并且当前访问函数会更新为 NULL。

更改访问作用域

下列命令是更改访问作用域的最常用方法：

- `func`
- `file`
- `up`
- `down`
- `frame` *number*
- `pop`
- `list` *procedure*

`debug` 命令和 `attach` 命令将设置初始访问作用域。

遇到断点时，`dbx` 会将访问作用域设置为当前位置。如果将 `stack-find-source` 环境变量设置为 `on`，则 `dbx` 会尝试查找并激活有源代码的堆栈帧。

当您使用 `up` 命令、`down` 命令、`frame` 命令或 `pop` 命令更改当前堆栈帧时，`dbx` 会根据来自新堆栈帧的程序计数器设置访问作用域。

仅当使用 `list` 命令时，`list` 命令使用的行号位置才会更改访问作用域。在设置访问作用域后，`list` 命令的行号位置会设置为访问作用域的第一个行号。以后使用 `list` 命令时，`list` 命令的当前行号位置会更新，但只要是列出当前文件中的代码行，访问作用域就不会更改。例如，以下命令会导致 `dbx` 列出 `my-func` 源的开头，并将访问作用域更改为 `my-func`。

```
(dbx) list my-func
```

以下命令会导致 `dbx` 列出当前源文件的第 127 行，但不会更改访问作用域。

```
(dbx) list 127
```

使用 `file` 命令或 `func` 命令更改当前文件或当前函数时，访问作用域也会相应地更新。

使用作用域转换操作符限定符号

使用 `func` 或 `file` 命令时，可能需要使用作用域转换操作符来限定作为目标给出的函数的名称。

`dbx` 提供了三个用于限定符号的作用域转换操作符：反引号操作符 (`)、C++ 双冒号操作符 (`::`) 和块局部操作符 (*:lineno*)。应单独使用它们，在某些情况下可以一起使用。

除了在代码中导航时限定文件名和函数名外，输出和显示作用域外变量和表达式以及显示类型和类声明时还必须限定符号名（使用 whatis 命令）。

本节介绍了所有各类符号名称限定的规则。符号限定规则在所有情况下都是相同的。

反引号操作符

使用反引号字符 (`) 可查找全局作用域变量或函数：

```
(dbx) print `item
```

一个程序可以在两个不同的文件或编译模块中使用同一函数名。在这种情况下，还必须将函数名限定到 dbx，以便它记录您要导航的函数。要按相应的文件名限定函数名，请使用通用反引号 (`) 作用域转换操作符。

```
(dbx) func`filename`function-name
```

C++ 双冒号作用域转换操作符

使用双冒号操作符 (::) 可以用以下名称类型限定具有全局作用域的 C++ 成员函数、顶级函数或变量：

- 重载名（不同参数类型使用同一名称）
- 二义名（不同类中使用同一名称）

如果不限定重载函数名称，则 dbx 会显示一个重载列表，以便您从中选择要导航的函数。如果您知道函数类名，则可以将其与双冒号作用域转换操作符一起使用来限定名称。

```
(dbx) func class::function-name (args)
```

例如，如果 hand 是类名，而 draw 是函数名：

```
(dbx) func hand::draw
```

块局部操作符

使用块局部操作符 (:*line-number*)，您可以专门引用嵌套块中的变量。如果有遮蔽参数或成员名的局部变量，或如果有几个块，其中每个块都有其自己的局部变量版本，可能需要这样做。行号是相关变量所对应的块内第一行代码的号码。当 dbx 使用块局部操作符限定局部变量时，dbx 会使用第一个代码块的行号，但您可以在 dbx 表达式中使用作用域内的任意行号。

在下例中，块局部操作符 (:230) 与反引号操作符配合使用。

```
(dbx) stop in `animate.o`change-glyph:230`item
```

下例显示了当函数中有多个具体值时，dbx 如何对使用块局部操作符限定的变量名求值。

```
(dbx) list 1,$
    1   #include <stddef.h>
    2
    3   int main(int argc, char** argv) {
    4
    5   int i=1;
    6
    7       {
    8            int i=2;
    9            {
   10                   int j=4;
   11                   int i=3;
   12                   printf("hello");
   13           }
   14           printf("world\n");
   15       }
   16       printf("hi\n");
   17   }
   18
(dbx) whereis i
variable: `a.out`t.c`main`i
variable: `a.out`t.c`main:8`i
variable: `a.out`t.`main:10`i
(dbx) stop at 12 ; run
...
(dbx) print i
i = 3
(dbx) which i
`a.out`t.c`main:10`i
(dbx) print `main:7`i
`a.out`t.c`main`i = 1
(dbx) print `main:8`i
`a.out`t.c`main:8`i = 2
(dbx) print `main:10`i
`a.out`t.c`main:10`i = 3
(dbx) print `main:14`i
`a.out`t.c`main:8`i = 2
(dbx) print `main:15`i
`a.out`t.c`main`i = 1
```

链接程序名

dbx 提供了一种按链接程序名（在 C++ 中为改编名称）查找符号的特殊语法。使用 #（英镑标记）字符为符号名称加前缀。在任何 $（美元标记）字符前面使用 ksh 转义符 \（反斜杠）。

```
(dbx) stop in #.mul
(dbx) whatis #\$FEcopyPc
(dbx) print `foo.c`#staticvar
```

查找符号

在程序中，同一名称可能会引用不同类型的程序实体，并可能会在许多作用域中出现。dbx whereis 命令会列出全限定名称，即该名称的所有符号的位置。如果在表达式中给出该名称，则 dbx which 命令会告知 dbx 将使用符号的哪个具体值。

输出符号具体值列表

要输出指定符号的所有具体值的列表，请使用 whereis *symbol,*，其中 *symbol* 可以是用户定义的任何标识符。例如：

```
(dbx) whereis table
forward: `Blocks`block-draw.cc`table
function: `Blocks`block.cc`table::table(char*, int, int, const point&)
class: `Blocks`block.cc`table
class: `Blocks`main.cc`table
variable:       `libc.so.1`hsearch.c`table
```

输出内容包括程序在其中定义 *symbol* 的可装入对象的名称、及其实体类型：类、函数或变量。

由于 dbx 符号表中的信息是在需要时才读入，因此 whereis 命令只记录已装入符号的具体值。随着调试会话越来越长，具体值列表也会增长。有关更多信息，请参见“对象文件和可执行文件中的调试信息” [73]。

确定 dbx 使用哪个符号

如果在表达式中指定了名称（没有完全限定），则 which 命令会告知 dbx 使用哪个具有给定名称的符号。例如：

```
(dbx) func
wedge::wedge(char*, int, int, const point&, load-bearing-block*)
(dbx) which draw
`block-draw.cc`wedge::draw(unsigned long)
```

如果指定的符号名不在局部作用域中，则 which 命令会沿着作用域转换搜索路径搜索该符号的第一个具体值。如果 which 找到该名称，则它会报告全限定名称。

如果搜索操作在搜索路径中的任何位置找到了 *symbol* 的处于同一作用域级别的多个具体值，则 dbx 会在命令窗格中输出一条消息，报告这种不明确情况。

```
(dbx) which fid
More than one identifier `fid'.
Select one of the following:
 0) Cancel
 1) `example`file1.c`fid
 2) `example`file2.c`fid
```

dbx 会显示重载信息，并列出二义符号名。在 which 命令的上下文中，从具体值列表中进行选择不会影响 dbx 或程序的状态。无论选择哪个具体值，dbx 都会回显名称。

如果使 *symbol*（本例中为 block）成为必须对 *symbol* 运行的某个命令（例如，print 命令）的参数，则 which 命令会让您预览将发生的事情。如果有二义名，重载显示列表会指明 dbx 尚未记录它会使用两个或更多名称中的哪一个具体值。dbx 会列出可能的值，等待您从中选择一个。

作用域转换搜索路径

当您发出包含表达式的调试命令时，将按以下顺序查找表达式中的符号。dbx 会按编译器在当前访问作用域中进行操作的同样方式解析这些符号。

1. 在使用当前访问作用域的当前函数的作用域内，如果程序在嵌套块中停止，则 dbx 会在该块内搜索，然后在所有封装块的作用域中搜索。
2. 仅限于 C++： 当前函数的类及其基类的类成员。
3. 仅限于 C++： 当前名字空间。
4. 当前函数的参数。
5. 立即封装模块，通常为包含当前函数的文件。
6. 供此共享库或可执行文件专用的符号。可使用链接程序作用域来创建这些符号。
7. 主程序的全局符号，然后为共享库的全局符号。
8. 如果上述搜索都不成功，则 dbx 会假定您正在引用其他文件中的专用或文件静态变量或函数。dbx 可选择根据 dbxenv 设置 scope-look-aside 的值在每个编译单元中搜索文件静态符号。

无论 dbx 使用符号的哪个具体值，它都会先沿此搜索路径查找。如果 dbx 找不到符号，它会报告错误。

放宽作用域查找规则

要为静态符号和 C++ 成员函数放宽作用域查找规则，请将 dbxenv 变量 scope-look-aside 设置为 on：

```
dbxenv scope-look-aside on
```

您还可以使用“双反引号”前缀：

```
stop in ``func4           func4 may be static and not in scope
```

如果将 dbxenv 变量 scope-look-aside 设置为 on，则 dbx 查找以下内容：

- 在其他文件中定义的静态变量（如果在当前作用域中没有找到）。不会搜索 /usr/lib 下的库中的文件。
- 没有类限定的 C++ 成员函数。
- 其他文件中的 C++ 内联成员函数实例（如果某个成员函数在当前文件中未实例化）。

which 命令会告知您 dbx 选择哪个符号。如果有二义名，重载显示列表会指明 dbx 尚未确定它会使用两个或更多名称中的哪一个具体值。dbx 会列出可能的值，等待您从中选择一个。

查看变量、成员、类型和类

whatis 命令可输出标识符、结构、类型和 C++ 类的声明或定义，或者表达式类型。您可以查找的标识符包括变量、函数、字段、数组和枚举常量。

有关更多信息，请参见“whatis 命令” [366]。

查找变量、成员和函数的定义

使用 whatis 命令可输出标识符的声明：

(dbx) **whatis** *identifier*

根据需要使用文件和函数信息来限定标识符名。

对于 C++ 程序，whatis 会列出函数模板实例。可使用 whatis -t 显示模板定义（请参见“查找类型和类的定义” [71]）。

对于 Java 程序，whatis *identifier* 会列出类的声明、当前类中的方法、当前帧中的局部变量或当前类中的字段。

要输出成员函数，请键入以下命令：

```
(dbx) whatis block::draw
void block::draw(unsigned long pw);
(dbx) whatis table::draw
void table::draw(unsigned long pw);
(dbx) whatis block::pos
class point *block::pos();
(dbx) whatis table::pos
class point *block::pos();
:
```

要输出数据成员

```
(dbx) whatis block::movable
int movable;
```

对于变量，whatis 命令将提供变量的类型。

```
(dbx) whatis the-table
class table *the-table;
.
```

对于字段，whatis 命令将提供字段的类型。

```
(dbx) whatis the-table->draw
void table::draw(unsigned long pw);
```

当程序在成员函数中停止时，可以查找 this 指针。

```
(dbx) stop in brick::draw
(dbx) cont
(dbx) where 1
brick::draw(this = 0x48870, pw = 374752), line 124 in
     "block-draw.cc"
(dbx) whatis this
class brick *this;
```

查找类型和类的定义

whatis 命令的 -t 选项可显示类型的定义。对于 C++，whatis -t 显示的列表包括模板定义和类模板实例。

要输出类型或 C++ 类的声明：

```
(dbx) whatis -t class-name
```

要仅查看数据成员，请使用 whatis 命令以及 -a 选项。该选项仅输出特定类的数据成员的列表（而不输出成员的函数）。它显示该信息的顺序与 -r 选项相同，即先从基类开始。

```
(dbx) whatis -t -a class-name
```

要查看基类中的成员，可以在 whatis 命令中使用 -r 选项（用于递归）。这样将显示指定类的声明以及它从基类继承的成员。

```
(dbx) whatis -t -r  class-name
```

whatis -r 查询的输出可能会很长，具体取决于类分层结构以及类的大小。输出将显示继承的数据成员的列表，从最原始的类开始。插入的注释行将成员列表分到其各自的父类中。

要查看类的继承成员的根目录，可以在 whatis 命令中使用 -u 选项，以显示类型定义的根目录。如果不使用 -u 选项，则 whatis 命令将显示值历史记录中的最近值。这与 gdb 中使用的 ptype 命令类似。

在下面的两个示例中，使用了类 table，它是父类 load-bearing-block 的子类，而该父类又是 block 的子类。

如果不使用 -r，whatis 会报告在类 table 中声明的成员。

```
(dbx) whatis -t class table
class table : public load-bearing-block {
public:
    table::table(char *name, int w, int h, const class point &pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
```

以下示例显示了对子类使用 whatis -r 来查看它所继承的成员时的结果。

```
(dbx) whatis -t -r class table
class table : public load-bearing-block {
public:
  /* from base class table::load-bearing-block::block */
  block::block();
  block::block(char *name, int w, int h, const class point &pos, class load-bearing-block
 *blk);
    virtual char *block::type();
    char *block::name();
    int block::is-movable();
//  deleted several members from example protected:
    char *nm;
    int movable;
    int width;
    int height;
    class point  position;
    class load-bearing-block *supported-by;
    Panel-item panel-item;
    /* from base class table::load-bearing-block */
public:
    load-bearing-block::load-bearing-block();
```

```
    load-bearing-block::load-bearing-block(char *name, int w, int h,
        const class point &pos, class load-bearing-block *blk);
    virtual int load-bearing-block::is-load-bearing();
    virtual class list *load-bearing-block::supported-blocks();
    void load-bearing-block::add-supported-block(class block &b);
    void load-bearing-block::remove-supported-block(class block &b);
    virtual void load-bearing-block::print-supported-blocks();
    virtual void load-bearing-block::clear-top();
    virtual void load-bearing-block::put-on(class block &object);
    class point load-bearing-block::get-space(class block &object);
    class point load-bearing-block::find-space(class block &object);
    class point load-bearing-block::make-space(class block &object);
protected:
    class list *support-for;
    /* from class table */
public:
    table::table(char *name, int w, int h, const class point &pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
```

对象文件和可执行文件中的调试信息

为了获得最佳结果，您可使用 `-g` 选项编译源文件，以使程序的可调试性更好。`-g` 选项会使编译器将调试信息（采用 stabs 或 DWARF 格式）与程序的代码和数据一起记录到对象文件中。

需要调试信息时，`dbx` 会根据需要解析和装入每个对象文件（模块）的调试信息。可以使用 `module` 命令让 `dbx` 装入有关任何特定模块或所有模块的调试信息。另请参见“查找源文件和对象文件” [77]。

对象文件装入

将对象 (`.o`) 文件链接到一起后，链接程序可选择只将摘要信息存储到生成的装入对象中。`dbx` 可以在运行时使用此摘要信息从对象文件本身（而不是可执行文件）装入其余调试信息。生成的可执行文件占用的磁盘资源较小，但要求在 `dbx` 运行时能够使用对象文件。

使用 `-xs` 选项编译对象文件可覆盖此要求，从而使这些对象文件的所有调试信息在链接时都被放入可执行文件中。

如果使用对象文件创建归档库（`.a` 文件），并且在程序中使用归档库，则 `dbx` 会根据需要从归档库中提取对象文件。此时不需要原始对象文件。

将所有调试信息放入可执行文件的唯一缺点是会占用更多磁盘空间。由于运行时调试信息并未装入到进程映像中，因此程序运行速度不会降低。

使用 stabs 时的缺省行为是使编译器只将摘要信息放入可执行文件中。

通过使用 `-xs` 选项，您可利用 DWARF 创建对象文件。有关更多信息，请参见"索引 DWARF (`-xs[={yes|no}]`)" [75]。

注 - 记录相同的信息时，使用 DWARF 格式要比使用 stabs 格式紧凑得多。但是，由于将全部信息都复制到可执行文件中，因此 DWARF 信息所占的空间看上去要比 stabs 信息所占的空间大。

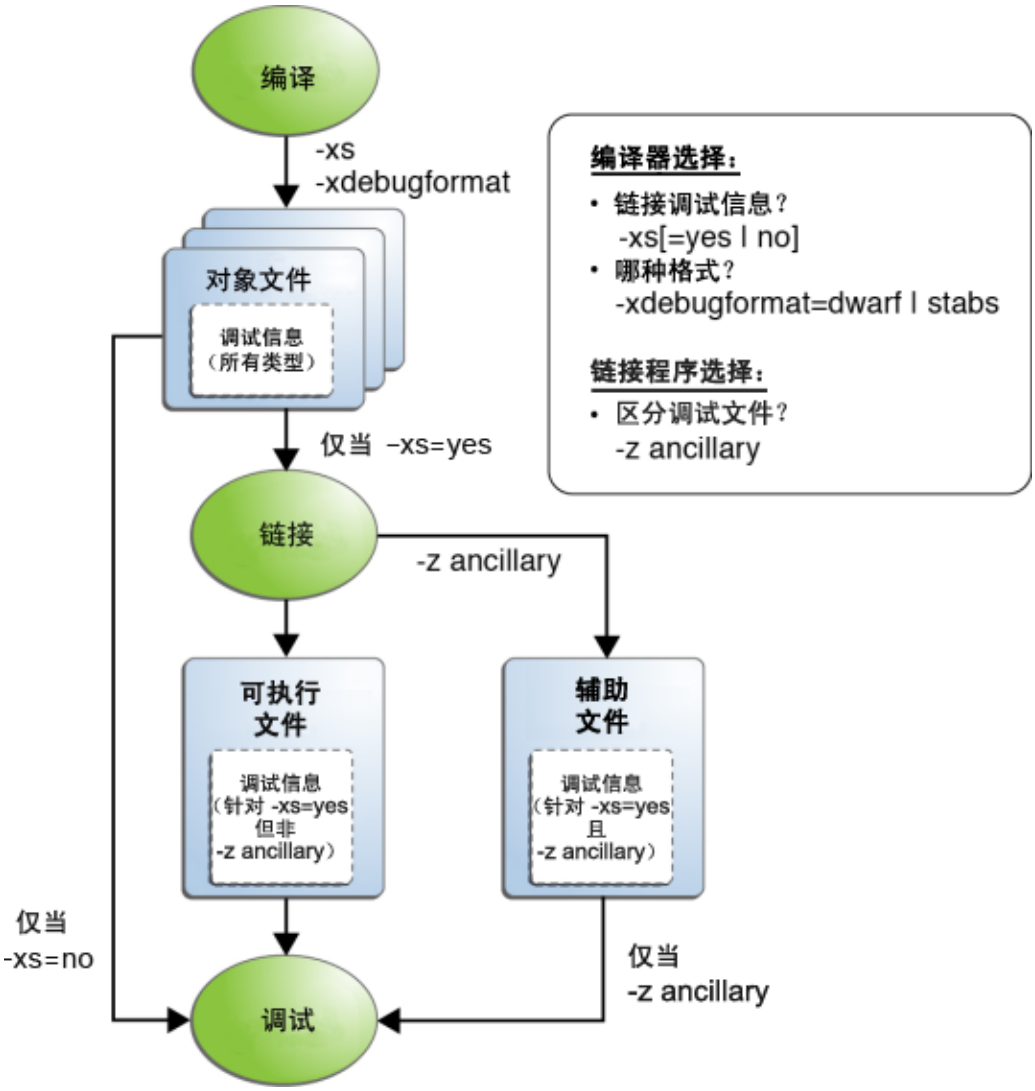
有关 stabs 索引的更多信息，请参见 Stab 接口指南（可在 *install-dir*`/solarisstudio12.4/READMEs/stabs.pdf` 路径中找到）。

用于支持调试的编译器和链接程序选项

使用编译器和链接程序选项，用户可更自由地生成和使用调试信息。编译器为 DWARF 生成索引，这与索引 stabs 类似。该索引始终存在，并可在使用 DWARF 进行调试时加快 `dbx` 启动速度并带来其他改进。

下图显示了不同类型的调试信息及其位置，特别突出显示了调试数据所在的位置：

图 1 调试信息流



索引 DWARF (-xs[={yes|no}])

缺省情况下，DWARF 已装入可执行文件。新索引可使用 -xs=no 选项将 DWARF 保留在对象文件中。这样可生成更小的可执行文件和更快的链接。对象文件必须保留，以便进行调试。这与 stabs 的工作原理类似。

独立的调试文件 (-z ancillary[=*outfile*])

Oracle Solaris 11.1 链接程序可以将调试信息发送到单独的辅助文件，同时生成可执行文件。在必须移动、安装或归档所有调试信息的环境中，独立的调试文件非常有用。可执行文件既可独立运行，也可由用户使用其独立的调试文件副本进行调试。

dbx 继续支持使用 GNU 实用程序 objcopy 将调试信息提取到独立的文件中，但与 objcopy 相比，使用 Oracle Solaris 链接程序具有以下优点：

- 独立的调试文件作为链接的副产物生成
- 由于过大而难以作为一个文件链接的程序可作为两个文件链接为两个文件

有关更多信息，请参见“辅助文件（仅限 Oracle Solaris）” [43]。

最大程度地减少调试信息

-g1 编译器选项旨在实现已部署应用程序的最小可调试性。使用此选项编译应用程序时，将生成文件、行号以及系统认为对事后调试至关重要的简单参数信息。有关更多信息，请参见编译器手册页和编译器用户指南。

列出模块的调试信息

module 命令及其选项有助于在调试会话期间跟踪程序模块。使用 module 命令可读入一个模块或所有模块的调试信息。一般情况下，dbx 会根据需要自动并且延后读入模块的调试信息。

要读入模块的调试信息：

```
(dbx) module [-f] [-q] name
```

要读入所有模块的调试信息：

```
(dbx) module [-f] [-q] -a
```

其中：

-a	指定所有模块
-f	强制读取调试信息，即使该文件比可执行文件新也是如此。
-q	指定静默模式。
-v	指定冗余模式，在该模式下会输出语言、文件名等信息。这是缺省值。

要输出当前模块的名称，请键入：

```
(dbx) module
```

列出模块

`modules` 命令通过列出模块名称可帮助您跟踪模块。

要列出包含已读入 `dbx` 的调试信息的模块的名称，请键入：

```
(dbx) modules [-v] -read
```

要列出所有程序模块（无论是否包含调试信息）的名称，请键入：

```
(dbx) modules [-v]
```

要列出包含调试信息的所有程序模块，请键入：

```
(dbx) modules [-v] -debug
```

其中：

-v　　指定冗余模式，在该模式下会输出语言、文件名等信息。

查找源文件和对象文件

`dbx` 必须知道与程序关联的源代码文件的位置。源文件的缺省目录是上次编译时它们所在的目录。如果移动源文件或将源文件复制到新位置，必须重新链接程序并在调试前更改为新位置，或者使用 `pathmap` 命令。

如果采用 Sun Studio 11 及早期发行版中 `dbx` 所用的 stabs 格式，`dbx` 中的调试信息有时会使用对象文件来载入其他调试信息。当 `dbx` 显示源代码时，会使用源文件。

包括源文件路径在内的符号信息包含在可执行文件中。当 `dbx` 需要显示源代码行时，将根据查找源文件的需要读取适量的符号信息，并读取和显示源文件中的代码行。

符号信息包括源文件的全路径名，但是当键入 `dbx` 命令时，通常只使用文件的基名。例如：

```
stop at test.cc:34
```

`dbx` 将在符号信息中搜索匹配的文件。

如果源文件已经删除，`dbx` 将无法显示这些文件中的源代码行，但您可以显示堆栈跟踪，输出变量值，甚至还可以确定现在所处的源代码行。

如果在编译和链接程序后移动了源文件，可将其新位置添加到搜索路径中。pathmap 命令可创建从文件系统的当前视图到可执行映像中的名称的映射。该映射应用于源路径和对象文件路径。

要建立从目录 *from* 到目录 *to* 的新映射：

(dbx) **pathmap [-c]** *from to*

如果使用 -c，该映射还将应用于当前工作目录。

pathmap 命令还可用于处理在不同主机上具有不同基路径的自动挂载或显式 NFS 挂载的文件系统。因为当前工作目录在自动挂载的文件系统中不准确，所以在尝试解决由自动挂载程序引起的问题时，请使用 -c。

缺省情况下，存在 /tmp-mnt 到 / 的映射。

♦ ♦ ♦ 第 5 章

5 控制程序执行

用于运行、单步执行和继续执行的命令（run、rerun、next、step 和 cont）称为进程控制命令。将这些命令与事件管理命令结合使用，可以控制程序在 dbx 下执行时的运行时行为。

本章包含以下各节：

运行程序

首次将程序装入 dbx 时，dbx 会导航到程序的 "main" 块（对 C、C++ 和 Fortran 90 而言是 main；对 Fortran 77 而言是 MAIN；对 Java 代码而言是 main 类）。dbx 会等待您发出进一步的命令；方法是在代码中导航或使用事件管理命令。

运行程序之前，可以在程序内设置断点。

注 - 调试使用 Java™ 代码和 C JNI（Java Native Interface，Java 本地接口）代码或 C++ JNI 代码混合编写的应用程序时，可能需要在尚未装入的代码中设置断点。有关更多信息，请参见“在本地 (JNI) 代码中设置断点” [204]。

可以使用 run 命令来启动程序执行。

使用 <（用于输入）和 > 或 >>（用于输出），您可以选择性地添加命令行参数并重定向输入和输出。使用 >> 会将内容附加到现有输出文件。

(dbx) **run** [*arguments*][< *input-file*] [> *output-file*]

注 - 无法重定向 Java 应用程序的输入和输出。

注 - 即使已为运行 dbx 的 shell 设置了 noclobber，run 命令的输出也会覆盖现有文件，除非使用 >> 指示命令将输出附加到现有文件。

不带参数的 run 命令将使用上次的参数和重定向来重新启动程序。rerun 命令重新启动程序并清除原始参数和重定向。

将 dbx 连接到正在运行的进程

可能需要调试已经运行的程序。在以下情况中，需要连接到正在运行的进程：

- 要调试正在运行的服务器，但又不想停止或中止服务器。
- 要调试正在运行的具有图形用户界面的程序，但又不想重新启动该程序。
- 程序处于无限循环，要调试该程序但又不想将其中止。

通过将程序的进程 ID 编号用作 dbx debug 命令的一个参数，可以将 dbx 连接到正在运行的程序。

程序调试完毕后，便可以使用 detach 命令来解除 dbx 对程序的控制，而无需终止进程。

如果在将 dbx 连接到正在运行的进程后退出，dbx 会在终止前隐式分离。

要将 dbx 连接到独立于 dbx 运行的程序，可以使用 attach 命令或 debug 命令：

```
(dbx) debug program-name process-ID
```

或

```
(dbx) attach process-ID
```

可以用一个 –（短划线）替换程序名称。dbx 将自动查找与进程 ID 相关联的程序并将其装入。

有关更多信息，请参见“debug 命令” [291] 和“attach 命令” [273]。

如果 dbx 未运行，可通过键入以下命令来启动 dbx：

```
% dbx program-name process-id
```

dbx 连接到程序后，程序便停止执行。将其他程序装入 dbx 时，可以检查该程序。可以使用任何事件管理或进程控制命令来调试该程序。

在调试现有进程的过程中将 dbx 连接到新进程时，会出现下列情况：

- 如果使用 run 命令启动当前正在调试的进程，那么 dbx 会在连接新进程之前终止此进程。

- 如果使用 attach 命令或通过在命令行中指定进程 ID 来开始调试当前进程，则 dbx 会在连接到新进程之前从当前进程中分离。

如果要将 dbx 连接到的进程由于 SIGSTOP 信号、SIGTSTOP 信号、SIGTTIN 信号或 SIGTTOUT 信号而停止，则连接会成功，并显示类似以下的消息：

```
dbx76: warning: Process is stopped due to signal SIGSTOP
```

该进程是可检查的，但是要恢复它，您需要使用 cont 命令向其发送 SIGCONT 信号：

```
(dbx) cont -sig cont
```

可以对具有某些异常的连接的进程使用运行时检查。请参见“对连接的进程使用运行时检查” [137]。

从进程中分离 dbx

程序调试完毕后，请使用 detach 命令从程序中分离 dbx。这时程序将恢复独立于 dbx 而运行，除非在分离时指定 -stop 选项。

在临时应用其他基于 /proc 的调试工具（这些工具可能由于 dbx 独占访问而被阻止）时，可以分离进程并将其保留在停止状态。例如：

```
(dbx) oproc=$proc          # Remember the old process ID
(dbx) detach -stop
(dbx) /usr/proc/bin/pwdx $oproc
(dbx) attach $oproc
```

有关更多信息，请参见“detach 命令” [294]。

单步执行程序

dbx 支持两个基本单步执行命令：next 和 step，外加 step 命令的两个变体（称为 step up 和 step to）。next 命令和 step 命令均可在再次停止前执行一个源代码行。

如果执行的行中包含函数调用，next 命令允许执行调用并于下一行停止（“步过”调用）。step 命令停止在被调用函数的第一行（“步入”调用）。

step up 命令会在步入函数之后，将程序返回到调用方函数。

step to 命令会尝试步入当前源代码行中的指定函数；如果未指定任何函数，则尝试步入由当前源代码行的汇编代码确定调用的最后一个函数。可能会因为条件分支或当前源

代码行内没有被调用的函数，而无法执行函数调用。在这些情况下，step to 命令将步过当前源代码行。

有关 next 和 step 命令的更多信息，请参见“next 命令” [322] 和“step 命令” [341]。

控制单步执行行为

要单步执行指定行数的代码，请使用 dbx 命令 next 或 step，并在命令后跟要执行代码的行数 [*n*]。

```
(dbx) next n
```

或

```
(dbx) step n
```

step_granularity dbxenv 变量确定 step 命令和 next 命令单步执行代码的单元。该单元可以是 statement 或 line。

step_events 环境变量控制单步执行过程中是否启用断点。

step_abflow 环境变量控制当 dbx 检测到将发生不寻常的控制流更改时，是否停止运行。对 siglongjmp() 或 longjmp() 的调用或异常抛出有可能导致此类型的控制流更改。

有关更多信息，请参见“设置 dbxenv 变量” [54]。

步入特定函数或最后一个函数

要步入从当前源代码行调用的函数，请使用 step to 命令。

```
(dbx) step to function
```

要步入最后调用的函数：

```
(dbx) step to
```

对于以下两个示例而言，使用 step to 本身将会步入 foo：

```
foo(bar(baz(4)));

baz()->bar()-> foo()
```

继续执行程序

要在程序遇到断点或某些事件后继续执行程序，请使用 cont 命令。

```
(dbx) cont
```

使用变体 `cont at` *line-number*，您可以指定除当前程序位置行以外的行来恢复程序执行。此选项允许您跳过已知引起问题的一行或多行代码，而无需重新编译。

要在指定的行继续执行程序，请键入：

```
(dbx) cont at 124
```

行号是相对于程序停止所在的文件求值的。给定的行号必须位于当前函数的作用域内。

将 `cont at` *line-number* `command` 命令和 `assign` 命令配合使用，可以避免执行包含某个函数调用（可能会错误地计算某一变量的值）的代码行。要快速调整错误计算的值，请使用 `assign` 命令为变量赋予一个正确的值。使用 `cont at` *line-number* 来跳过包含不正确地计算值之函数调用的行。

例如，假定程序停止于第 123 行。第 123 行调用函数 `how_fast()`，该函数将不正确地计算变量 `speed`。您知道 speed 变量应该取什么值，因此便为 `speed` 赋值。然后跳过对 `how_fast()` 的调用，于第 124 行处继续执行程序。

```
(dbx) assign speed = 180; cont at 124;
```

如果使用带有 `when` 断点命令的 `cont` 命令，则程序每次尝试执行第 123 行时，都会跳过对 `how_fast()` 的调用。

```
(dbx) when at 123 { assign speed = 180; cont at 124;}
```

有关更多信息，请参见以下主题：

- “在源代码行设置断点” [88]
- “在不同类的成员函数中设置断点” [90]
- “在类的所有成员函数中设置断点” [90]
- “在非成员函数中设置多个断点” [90]
- “`when` 命令” [367]

调用函数

程序被停止时，可以使用 `dbx call` 命令调用函数，以此接受必须传递给被调用函数的参数值。

要调用过程，请键入函数名称并提供其参数。例如：

```
(dbx) call change_glyph(1,3)
```

如果参数是可选项，则必须在参数名称后键入括号。例如：

```
(dbx) call type_vehicle()
```

您可以使用 call 命令显式调用函数；或者，通过对包含函数调用的表达式求值或使用 stop in glyph -if animate() 等条件修饰符来隐式调用函数。

您可以通过使用 print 命令或 call 命令或任何其他用于执行函数调用的命令来调用 C++ 虚拟函数，就像调用任何其他函数一样。

对 C++ 而言，dbx 可处理隐式 this 指针、缺省参数和函数重载。如有可能，C++ 重载函数将自动求解。如果存在任何二义性（例如，未使用 -g 编译函数），dbx 将显示重载名称列表。

如果定义函数的源文件已使用 –g 选项编译，或原型声明在当前作用域可见，dbx 会检查参数的个数和类型，如果存在不匹配便发出错误消息。否则，dbx 不会检查参数的个数并继续执行调用。

缺省情况下，在每个 call 命令之后，dbx 都会自动调用 fflush(stdout) 来确保 I/O 缓冲区内存储的所有信息全部被输出。要禁用自动刷新，请将 dbxenv 变量 output_auto_flush 设置为 off。

使用 call 命令时，dbx 的行为就好像您使用了 next 命令一样（从被调用的函数返回）。但是，如果程序在被调用的函数内遇到断点，dbx 将在断点处停止程序并发出消息。如果之后键入 where 命令，堆栈跟踪将显示调用源自 dbx 命令级。

如果继续执行，调用会正常返回。如果尝试中止、运行、重新运行或调试，当 dbx 试图从嵌套中恢复时，命令便会终止。然后，可以重新发出命令。另外，可以使用命令 pop -c 弹出调试器最近调用之前的所有帧。

调用安全性

通过使用 call 命令或通过输出包含调用的表达式来在要调试的进程中进行调用时，可能会导致不明显的严重中断。例如：

- 调用可能会陷入无限循环（您可以中断该循环），或导致段故障。在许多情况下，您可以使用 pop -c 命令返回到调用方。
- 在多线程应用程序中进行调用时，将会恢复所有线程以避免死锁，所以您可能会在进行调用的线程之外的其他线程上看到负面影响。
- 断点条件中使用的调用可能会搞乱事件管理（请参见“恢复执行” [154]）。

dbx 进行的一些调用会安全地执行。如果遇到问题（通常是段故障）、而非通常的 Stopped with call to ...，则 dbx 将执行以下操作之一：

- 忽略任何 stop 命令（包括那些因检测内存访问错误而使用的命令）
- 自动发出 pop -c 命令以返回到调用方
- 继续执行

dbx 将在以下情况下使用安全调用：

- 由 display 命令输出的表达式中发生的调用。失败的调用显示为：ic0->get _data() = <call failed>

  要诊断此类故障，请尝试使用 print 命令输出表达式。
- 对 db_pretty_print() 函数的调用（使用 print -p 命令时除外）。
- 事件条件表达式中使用的调用。包含失败调用的条件会计算为 false。
- 执行 pop 命令期间为调用析构函数所做的调用。
- 所有内部调用。

使用 Ctrl+C 停止进程

可以通过按 Ctrl+C (^C) 来停止 dbx 中正在运行的进程。使用 ^C 停止进程时，dbx 会忽略 ^C，但子进程会将其作为 SIGINT 予以接受并停止。然后，便可以检查进程，就像进程是通过断点被停止的一样。

若要在使用 ^C 停止程序后恢复执行，请使用 cont 命令。要恢复执行时，不需要使用 cont 可选修饰符 sig *signal-name*。取消待决信号之后，cont 命令将恢复子进程。

事件管理

事件是指在调试进程中发生的、导致系统向 dbx 发出通知的事件。事件管理是指 dbx 在所调试的程序中发生事件时执行操作的能力。发生事件时，可利用 dbx 停止进程、执行任意命令或输出信息。例如，断点便是最简单的事件。另外，故障、信号、系统调用、dlopen() 调用以及数据更改（请参见“使用调用方过滤器限定断点” [94]）等都是事件。

有关事件管理的更深入的信息，如事件处理程序、事件安全性、创建事件、事件规范和其他事件管理主题，请参见附录 B, 事件管理。

♦ ♦ ♦ 第 6 章

6 设置断点和跟踪

发生事件时，可利用 dbx 停止进程、执行任意命令或输出信息。例如，断点便是最简单的事件。另外，错误、信号、系统调用、dlopen() 调用以及数据更改等都是事件。

本章说明如何设置、清除和列出断点和跟踪。有关设置断点和跟踪时可以使用的事件规范的完整信息，请参见“设置事件规范” [244]。

本章包含以下各节：

设置断点

在 dbx 中，可以使用下列三个命令设置断点：

- stop – 程序执行到使用 stop 命令创建的断点处时将停止。直到发出其他调试命令（如 cont、step 或 next）后，程序才会恢复执行。
- when – 程序执行到使用 when 命令创建的断点处时将停止，且 dbx 执行一个或多个调试命令，然后程序继续执行（除非执行的命令之一是 stop）。
- trace – 跟踪可显示程序中事件的相关信息，如变量值变化。尽管跟踪的行为与断点的行为不同，但跟踪和断点的事件处理程序相似。程序执行到使用 trace 命令创建的断点处时将停止，此时会发送事件特定的 trace 信息行，然后程序继续执行。

stop、when 和 trace 命令都将事件规范（说明断点所基于的事件）当作参数。“设置事件规范” [244]详细讨论了事件规范。

要设置计算机级断点，请使用 `stopi`、`wheni` 和 `tracei` 命令。有关更多信息，请参见第 17 章 在计算机指令级调试。

注 - 调试使用 Java™ 代码和 C JNI（Java Native Interface，Java 本地接口）代码或 C++ JNI 代码混合编写的应用程序时，可能需要在尚未装入的代码中设置断点。有关在此类代码中设置断点的信息，请参见“在本地 (JNI) 代码中设置断点” [204]。

在源代码行设置断点

可以使用 `stop at` 命令在行号处设置断点，其中 *n* 是源代码行号，*filename* 是可选的程序文件名限定符。

```
(dbx) stop at filename:n
```

例如：

```
(dbx) stop at main.cc:3
```

如果指定的行不是可执行的源代码行，`dbx` 会在下一个可执行源代码行处设置断点。如果没有可执行源代码行，`dbx` 会发出错误。

可以使用 `file` 命令设置当前文件并使用 `list` 命令列出要在其中停止的函数来确定要停止在那里的行。然后使用 `stop at` 命令在源代码行设置断点，如以下示例所示。

```
(dbx) file t.c
(dbx) list main
10    main(int argc, char *argv[])
11    {
12        char *msg = "hello world\n";
13        printit(msg);
14    }
(dbx) stop at 13
```

有关指定 at 位置事件的更多信息，请参见“at 事件规范” [244]。

在函数中设置断点

可以使用 `stop in` 命令在函数中设置断点。

```
(dbx) stop in function
```

In Function 断点用于在过程或函数中第一个源代码行的开头处暂停程序执行。

dbx 应当能够确定您正在引用哪个函数，但下列情况除外：

- 只通过名称来引用一个重载的函数。
- 引用以 ` 开头的函数。
- 按函数的链接程序名称（在 C++ 中为改编名称）引用函数。在这种情况下，如果您在名称前添加 # 前缀，则 dbx 将接受该名称。有关更多信息，请参见“链接程序名” [67]。

假设有下面一组声明：

```
int foo(double);
int foo(int);
int bar();
class x {
   int bar();
};
```

要在非成员函数处停止，以下命令将在全局 foo(int) 处设置断点：

```
stop in foo(int)
```

要在成员函数中设置断点：

```
stop in x::bar()
```

在以下命令中，dbx 无法确定所指是全局函数 foo(int) 还是全局函数 foo(double)，因而会显示重载菜单以便确认。

```
stop in foo
```

如果键入：

```
stop in `bar
```

dbx 便无法确定所指是全局函数 bar() 还是成员函数 bar()，因而会显示重载菜单。

注 - 如果成员名称是唯一的（如 unique_member），则使用 stop in unique_member 便已足够。如果成员名称不具有唯一性，则可以使用 stop in 命令并响应重载菜单，以指定您所指的成员。

有关指定 in-function 事件的更多信息，请参见“in 事件规范” [244]。

在 C++ 程序中设置多个断点

可以检查与对不同类成员的调用、对给定类任何成员的调用或对重载的顶级函数的调用有关的问题。可以将关键字 inmember、inclass、infunction 或 inobject 与 stop、when 或 trace 命令一起使用，在 C++ 代码中设置多个断点。

在不同类的成员函数中设置断点

要在特定成员函数的每个类特定变体（成员函数名相同，类不同）中设置断点，请使用 `stop inmember`。

例如，如果在几个不同的类中定义了函数 `draw`，要在每个函数中设置断点，请键入：

```
(dbx) stop inmember draw
```

有关指定 `inmember` 或 `inmethod` 事件的更多信息，请参见“`inmember` 事件规范” [245]。

在类的所有成员函数中设置断点

要在特定类的所有成员函数中设置断点，请使用 `stop inclass` 命令。

缺省情况下，断点只插入类中定义的类成员函数中，而不会插入可能从基类继承的类成员函数中。如果还要在从基类继承的函数中插入断点，请指定 `-recurse` 选项。

以下命令可在类 `shape` 中定义的所有成员函数中设置断点：

```
(dbx) stop inclass shape
```

以下命令可在类中定义的所有成员函数中以及从该类继承的函数中设置断点：

```
(dbx) stop inclass shape -recurse
```

有关指定 `inclass` 事件的更多信息，请参见“`inclass` 事件规范” [246] 和“`stop` 命令” [344]。

由于 `stop inclass` 和其他断点选择可能会插入大量断点，因此应确保将 `dbxenv` 变量 `step_events` 设置为 `on` 以加快 `step` 和 `next` 命令的执行速度。有关更多信息，请参见“效率考虑事项” [99]。

在非成员函数中设置多个断点

要在具有重载名称（名称相同、参数的类型或数量不同）的非成员函数中设置多个断点，请使用 `stop infunction` 命令。

例如，如果 C++ 程序定义了两个名为 `sort()` 的函数版本（一个传递 `int` 类型参数，另一个传递 `float` 类型参数），以下命令将会在这两个函数中设置断点：

```
(dbx) stop infunction sort
```

有关指定 `infunction` 事件的更多信息，请参见“`infunction` 事件规范” [245]。

在对象中设置断点

可以设置 In Object 断点来检查应用于特定对象实例的操作。

使用 In Object 断点在对特定对象实例调用任何方法时停止程序执行。例如，以下代码仅在调用 `f1->printit（）` 时导致停止：

```
Foo *f1 = new Foo();
Foo *f2 = new Foo();
 f1->printit();
 f2->printit();
```

```
(dbx) stop inobject f1
```

`f1` 中存储的地址将标识您在其中放置断点的对象。这表示只能在 `f1` 中的对象实例化之后创建此断点。

缺省情况下，In Object 断点会在对象的类（包括继承的类）的所有非静态成员函数中暂停程序执行。要将断点局限于对象类，请指定 `-norecurse` 选项。

要在对象 `foo` 的基类中定义的所有非静态成员函数中以及在对象 `foo` 的继承类中定义的所有非静态成员函数中设置断点：

```
(dbx) stop inobject &foo
```

要在对象 `foo` 的类中定义的所有非静态成员函数中设置断点，但不在对象 `foo` 的继承类中定义的所有非静态成员函数中设置断点：

```
(dbx) stop inobject &foo -norecurse
```

有关指定 `inobject` 事件的更多信息，请参见“`inobject` 事件规范” [246] 和“`stop` 命令” [344]。

设置数据更改断点（监视点）

您可以使用数据更改断点（也称为“监视点”）在 `dbx` 中注明更改变量或表达式值的时间。

访问地址时停止执行

使用 `stop access` 命令可在已访问内存地址时停止执行：

```
(dbx) stop access mode address-expression [, byte-size-expression]
```

mode 指定内存访问模式。有效的模式选项包括：

`r` 已读取指定地址处的内存。

`w` 已写入内存。

`x` 已执行内存。

mode 还可以包含以下任一项：

`a` 访问后停止进程（缺省值）。

`b` 访问前停止进程。

在这两种情况下，程序计数器都将指向访问指令。“之前”和“之后”都具有副作用。

address-expression 是求值结果为地址的任何表达式。如果提供符号表达式，则会自动推导出要监视的区域大小。可以通过指定 *byte-size-expression* 覆盖它。也可以使用非符号、无类型地址表达式，在这种情况下，必须提供大小。

在以下示例中，该命令将在读取了内存地址 `0x4762` 之后在任意四个字节后停止执行。

```
(dbx) stop access r 0x4762, 4
```

在以下示例中，将在写入变量速度前停止执行：

```
(dbx) stop access wb &speed
```

使用 `stop access` 命令时请记住下列要点：

- 写入变量时（即使值不变），会发生事件。
- 缺省情况下，在执行了写入变量的指令后会发生事件。可以通过将模式指定为 `b` 来指示要在执行指令前发生事件。

有关指定 access 事件的更多信息，请参见“`access` 事件规范” [246] 和“`stop` 命令” [344]。

变量更改时停止执行

如果指定变量的值已更改，请使用 `stop change` 命令停止程序执行：

```
(dbx) stop change variable
```

使用 `stop change` 命令时请记住下列要点：

- `dbx` 会将程序停止在引起指定变量值更改的行后面的那一行。
- 如果 *variable* 是函数的局部变量，则第一次进入函数并分配 *variable* 的存储空间时，便认为该变量已更改。对于参数也是如此。
- 该命令无法与多线程应用程序配合使用。

有关指定 change 事件的更多信息，请参见“change 事件规范” [247] 和“stop 命令” [344]。

dbx 实现的 stop change 操作是自动单步执行且每一步都检查值。如果库未使用 -g 选项进行编译，则单步执行会跳过库调用。因此，如果控制按以下方式流动，dbx 便不会跟踪嵌套的 user_routine2，因为跟踪会跳过库调用和对 user_routine2 的嵌套调用。

```
user_routine calls
   library_routine, which calls
     user_routine2, which changes variable
```

variable 值的更改是在从库调用返回后便已发生，而不是在 user_routine2 中发生的。

dbx 无法为块局部变量（{} 中嵌套的变量）中的更改设置断点。如果尝试在块局部嵌套变量中设置断点或跟踪，dbx 会发出错误，说明无法执行此操作。

注 - 与使用 change 事件相比，使用 access 事件监视数据更改的速度更快。access 事件不是自动单步执行程序，而是使用速度更快的硬件或 OS 服务。

条件停止执行

如果条件语句的求值结果为 true，请使用 stop cond 命令停止程序执行：

(dbx) **stop cond** *condition*

条件发生时，程序便停止执行。

使用 stop cond 命令时请记住下列要点：

- dbx 会将程序停止在使条件的求值结果为 true 的行后面的那一行。
- 该命令无法与多线程应用程序配合使用。

有关指定条件事件的更多信息，请参见“cond 事件规范” [248] 和“stop 命令” [344]。

在断点上设置过滤器

在 dbx 中，大多数事件管理命令都支持可选的事件过滤器修饰符。最简单的过滤器是指示 dbx 在程序执行到断点或跟踪处理程序处后或出现数据更改断点后测试条件。

如果相应过滤器条件的求值结果为 true（非 0），则会应用事件命令，且程序在断点处停止执行。如果条件的求值结果为 false (0)，dbx 会继续执行程序，就好像从未发生过事件。

要设置包含过滤器的断点，请将可选的 - if *condition* 修饰符语句添加到 stop 或 trace 命令的末尾。

条件可以是任何有效的表达式（包括函数调用），其返回值是布尔值或输入命令时所用语言表示的整数值。

对于像 `in` 或 `at` 这样基于位置的断点，用来解析条件的作用域便是断点位置的作用域。否则，条件的作用域是输入时的作用域，而不是事件发生时的作用域。可能必须使用反引号操作符（请参见“反引号操作符” [66]）来精确指定作用域。

以下两个过滤器是不一样的：

```
stop in foo -if a>5
stop cond a>5
```

前者在 `foo` 处中断并测试条件。后者自动单步执行并测试条件。

使用条件过滤器限定断点

要设置包含过滤器的断点，请将可选的 `-if` *condition* 修饰符语句添加到 `stop` 或 `trace` 命令的末尾。*condition* 可以是任何有效的表达式（包括函数调用），其返回值是布尔值或输入命令时所用语言表示的整数值。

可以将函数调用用作断点过滤器。在以下示例中，如果字符串 `str` 中的值是 `abcde`，则在函数 `foo()` 中停止执行：

```
(dbx) stop in foo -if !strcmp(“abcde”,str)
```

您可以使用包含函数调用的 `-if` 选项：

```
stop in lookup -if strcmp(name, "troublesome")==0
```

下面是使用包含监视点的条件过滤器示例：

```
(dbx) stop access w &speed -if speed==fast_enough
```

使用调用方过滤器限定断点

没有经验的用户有时会将设置条件事件命令（监视类型命令）与使用过滤器混淆。从概念上来说，“监视”会创建在执行每行代码前必须检查的前提条件（在监视的作用域内）。但即便是有条件触发的断点命令也可以连接过滤器。

假设有这样一个示例：

```
(dbx) stop access w &speed -if speed==fast_enough
```

此命令指示 `dbx` 监视变量 *speed*；如果变量 *speed* 已写入（“监视”部分），则 `-if` 过滤器生效。`dbx` 检查 *speed* 的新值是否等于 `fast_enough`。如果不等，程序会继续执行，而“忽略”`stop` 命令。

在 `dbx` 语法中，过滤器以命令末尾处的 `[-if` *condition*`]` 形式表示。

```
stop in function [-if condition]
```

假设有一个包含以下代码的简单示例：

```
44:     if(open(filename, ...) == -1)
45:          return "Error";
```

您可以使用以下命令在出现某一特定故障时停止执行，例如，open（） 的 ENOENT：

```
(dbx) stop at 45 -if errno == 2
```

使用过滤器，可以非常方便地在局部变量上放置数据更改断点。在以下示例中，当前作用域处在函数 foo（） 中，而相关变量 index 处在函数 bar（） 中。

```
(dbx) stop access w &bar`index -in bar
```

bar`index 用于确保选取的是函数 bar（） 中的 index 变量，而不是函数 foo 中的 index 变量或名为 index 的全局变量。

-in bar 表示以下内容：

- 进入函数 bar（） 时自动启用断点。
- 在执行 bar（） （包括所调用的任何函数）期间，断点保持启用状态。
- 从 bar（） 返回时自动禁用断点。

某些其他函数的其他局部变量可能会重用与 index 对应的堆栈位置。-in 用于确保仅当访问 bar`index 时触发断点。

过滤器和多线程

如果在多线程程序中设置的断点中使用了包含多个函数调用的过滤器，dbx 将在到达断点时停止所有线程的执行，然后对条件求值。如果满足条件且调用了函数， dbx 便会恢复执行调用期间内的所有线程。

例如，可以在多线程应用程序中许多线程调用 lookup() 之处设置以下断点：

```
(dbx) stop in lookup -if strcmp(name, “troublesome”) == 0
```

dbx 会在线程 t@1 调用 lookup（） 时停止，并对条件求值，然后调用恢复所有线程的 strcmp（） 。如果在函数调用期间 dbx 在另一个线程中遇到断点，便会发出警告，例如：

```
event infinite loop causes missed events in the following handlers:
...

Event reentrancy
first event BPT(VID 6m TID 6, PC echo+0x8)
second event BPT*VID 10, TID 10, PC echo+0x8)
the following handlers will miss events:
...
```

在这种情况下，如果可以确定条件表达式中调用的函数不会抓取互斥锁，则可使用 -resumeone 事件规范修饰符强制 dbx 只恢复在其中遇到断点的第一个线程。例如，可设置以下断点：

```
(dbx) stop in lookup -resumeone -if strcmp(name, "troublesome") == 0
```

在有些情况下，-resumeone 修饰符并不能防止出现问题。例如，它在以下情况下不起作用：

- 由于条件以递归方式调用 lookup（），所以 lookup（）上的第二个断点与第一个断点出现在同一个线程中。
- 运行条件的线程放弃控制权，将其交给另一个线程。

有关详细信息，请参见"事件规范修饰符" [256]。

跟踪执行

跟踪会收集程序中发生情况的相关信息并显示这些信息。程序执行到使用 trace 命令创建的断点处时将停止，此时会发送事件特定的 trace 信息行，然后程序继续执行。

跟踪会显示即将执行的每个源代码行。除最简单的程序以外，此跟踪都会产生大量输出。

一种更加有用的跟踪是应用过滤器显示程序中事件的相关信息。例如，可以跟踪每个函数调用、给定名称的每个成员函数、类中的每个函数或每次从函数的退出。还可以跟踪对变量的更改。

设置跟踪

可以通过在命令行中键入 trace 命令设置跟踪。trace 命令的基本语法如下：

trace *event-specification* [*modifier*]

有关 trace 命令的完整语法，请参见"trace 命令" [355]。

跟踪提供的信息取决于与之关联的事件类型（请参见"设置事件规范" [244]）。

控制跟踪速度

通常，跟踪输出的显示速度太快。使用 dbxenv 变量 trace_speed 可以控制输出每个跟踪之后的延迟。缺省延迟时间为 0.5 秒。

要设置跟踪期间每个代码行执行间隔时间（秒）：

```
dbxenv trace_speed number
```

将跟踪输出定向到文件

可以使用 `-file` *filename* 选项将跟踪输出定向到文件。例如，以下命令可将跟踪输出定向到文件 `trace1`：

```
(dbx) trace -file trace1
```

要将跟踪输出恢复为标准输出，请使用 `-` 表示 *filename*。跟踪输出始终附加到 *filename* 后面。每当显示 `dbx` 提示以及应用程序退出时都会刷新。在执行新的运行或连接后继续运行时总会重新打开文件。

在行中执行 dbx 命令

`when` 断点命令接受其他 `dbx` 命令（如 `list`），这表示您可以编写自己的 `trace` 版本。

```
(dbx) when at 123 {list $lineno;}
```

`when` 命令隐含了 `cont` 命令。在该示例中，列出当前行的源代码后，程序会继续执行。如果在 `list` 命令后添加了 `stop` 命令，程序便不会继续执行。

有关 `trace` 命令的完整语法，请参见“when 命令” [367]。有关事件修饰符的详细信息，请参见“事件规范修饰符” [256]。

在动态装入的库中设置断点

`dbx` 与以下类型的共享库进行交互：

- 在程序开始执行时隐式装入的库。
- 使用 `dlopen(2)` 显式（动态）装入的库。只有在运行期间装入了库以后，此类库的名称才是已知的，所以您无法在使用 `debug` 或 `attach` 命令启动调试会话后在这些库中放置断点。
- 使用 `dlopen(2)` 显式装入的过滤器库。只有在装入库且调用其中的第一个函数之后，此类库的名称才是已知的。

可以采用以下两种方式在显式（动态）装入的库中设置断点：

- 如果有一个包含函数 `myfunc()` 的库（例如 `mylibrary.so`），可以将此库的符号表预装入 `dbx` 中并在此函数上设置断点，如下所示：

```
(dbx) loadobject -load fullpathto/mylibrary.so
(dbx) stop in myfunc
```

- 采用更简单的方式，在 dbx 下运行程序直至完成。dbx 将记录并记住使用 dlopen(2) 装入的所有共享库，即使这些库已通过 dlclose（） 关闭。所以在第一次运行程序之后，您将能够成功设置断点。

```
(dbx) run
execution completed, exit code is 0
(dbx) loadobject -list
u   myprogram (primary)
u   /lib/libc.so.1
u p /platform/sun4u-us3/lib/libc_psr.so.1
u   fullpathto/mylibrary.so
(dbx) stop in myfunc
```

列出和删除断点

通常，在调试会话期间会设置多个断点或跟踪处理程序。dbx 中有用于列出和清除它们的命令。

列出断点和跟踪

要显示所有活动断点的列表，请使用 status 命令显示 ID 号（用圆括号或方括号括住），以后其他命令可以使用该号。如果 ID 号用方括号括起来，则会禁用这些断点。此外，圆括号或方括号之前可能会出现星号 (*)，用于指示程序是否因该事件而停止。

对于使用关键字 inmember、inclass 和 infunction 设置的多个断点，dbx 将使用一个状态 ID 号将它们一起作为一组断点来报告。

使用处理程序 ID 号删除特定断点

使用 status 命令列出断点时，dbx 会显示创建每个断点时为其分配的 ID 号。可以使用 delete 命令按 ID 号删除断点，也可以使用关键字 all 来删除程序中当前设置的所有断点。

要按 ID 号删除断点（此例中为 3 和 5）：

```
(dbx) delete 3 5
```

要删除 dbx 中当前装入的程序中设置的所有断点：

```
(dbx) delete all
```

有关更多信息，请参见“delete 命令” [294]。

启用和禁用断点

用来设置断点的每个事件管理命令 (stop, trace, when) 都会创建一个事件处理程序。其中每个命令都会返回一个称为处理程序 ID (*hid*) 的编号。可将处理程序 ID 用作 handler 命令的参数来启用或禁用断点。例如：

```
(dbx) handler -disable 5
(dbx) handler -enable 5
```

有关更多信息，请参见“事件处理程序” [241]。

效率考虑事项

在所调试程序的执行时间方面，各种事件都有不同程度的开销。但某些事件（如最简单的断点）几乎没有开销。基于单个断点的事件的开销非常小。

多种可能会导致生成数百个断点的断点（如 inclass）仅在创建期间有开销。这是因为 dbx 使用永久性断点，这些断点一直保留在进程中，每次中断时并不会被移除，且每次执行 cont 命令时都会被置入。

就 step 和 next 而言，缺省情况下，恢复进程前会移除所有断点，完成相应步骤后会立即重新插入这些断点。如果使用大量断点或在多产类中使用多个断点，step 命令和 next 命令的执行速度会显著降低。可使用 dbx step_events 环境变量控制是否移除断点并在每次执行 step 命令或 next 命令后重新插入断点。

速度最慢的事件是利用自动单步执行功能的事件。在单步执行每个源代码行的 trace step 命令中，该进程可能非常明显。其他一些事件（如 stop change 或 trace cond 命令）不仅自动单步执行，而且还必须在执行每一步时对表达式或变量求值。

这些事件的速度非常慢，但通常可以使用 -in 修饰符将事件与函数绑定来克服速度慢这一问题。例如：

```
trace next -in mumble
stop change clobbered_variable -in lookup
```

请勿使用 trace -in main，这是因为 trace 在 main 调用的函数中也有效。如果怀疑 lookup（） 函数会损坏变量，请使用此修饰符。

♦ ♦ ♦ 第 7 章

7 使用调用堆栈

本章讨论 dbx 如何使用调用堆栈，以及在处理调用堆栈时如何使用 where 命令、hide 命令、unhide 命令和 pop 命令。

在多线程程序中，这些命令可对当前线程的调用堆栈进行操作。有关如何更改当前线程的信息，请参见“thread 命令” [352]。

本章包含以下各节：

调用堆栈代表当前处于活动状态的所有例程，即那些已调用但尚未返回各自调用方的例程。堆栈帧是分配供一个函数使用的调用堆栈的一段。

由于调用堆栈是从高端内存（较大地址）延伸到低端内存，因此向上意味着向调用方的帧移动（最终移动到 main（） 或该线程的起始函数），向下意味着向被调用函数的帧移动（最终移动到当前函数）。程序在断点处停止、单步执行后或出错并生成信息转储文件时，用于例程执行的帧位于低端内存中。调用方例程（如 main（））位于高端内存中。

确定在堆栈中的位置

使用 where 命令确定当前在堆栈中的位置。

```
where [-f] [-h] [-l] [-q] [-v] number-ID
```

调试使用 Java™ 代码和 C JNI（Java Native Interface，Java 本地接口）代码或 C++ JNI 代码混合编写的程序时，where 命令的语法为：

```
where [-f] [-q] [-v] [ thread_id ] number-ID
```

where 命令对于了解已崩溃并已生成信息转储文件的程序的状态也很有用。出现这种情况时，可将信息转储文件装入 dbx 中 （请参见“调试信息转储文件” [36]）。

有关更多信息，请参见“where 命令” [370]。

堆栈中移动和返回起始位置

在堆栈中上下移动称为“堆栈中移动”。当您通过在堆栈中上下移动的方式访问函数时，dbx 会显示当前函数和源代码行。起始位置起始是程序停止执行的点。可以使用 up 命令、down 命令或 frame 命令从起始处在堆栈中上下移动。

dbx 命令 up 和 down 均接受 *number* 参数，该参数指示 dbx 在堆栈中从当前帧向上或向下移动若干帧。如果未指定 *number*，则缺省值为 1。-h 选项将所有隐藏帧都包括在计数中。

在堆栈中上下移动

可以检查非当前函数中的局部变量。

在堆栈中上移

要在调用堆栈中上移（向 main 移动）*number* 级：

```
up [-h] [ number ]
```

如果未指定 *number*，则缺省值为一级。有关更多信息，请参见“up 命令” [364]。

在堆栈中下移

要在调用堆栈中下移（向当前停止点移动）*number* 级：

```
down [-h] [ number ]
```

如果未指定 *number*，则缺省值为一级。有关更多信息，请参见“down 命令” [297]。

移到特定帧

frame 命令与 up 命令和 down 命令类似。用于直接转到 where 命令显示的编号所指示的帧。

```
frame
frame -h
frame [-h] number
frame [-h] +[number]
frame [-h] -[number]
```

如果执行 `frame` 命令时不使用参数，将显示当前帧编号。如果使用 *number*，则直接转到编号所指示的帧。在命令中加入 +（加号）或 -（减号）可向上 (+) 或向下 (-) 移动一级。如果在 *number* 中包括加号或减号，可以向上或向下移动指定数量的级别。`-h` 选项将所有隐藏帧都包括在计数中。

也可以使用 `pop` 命令移到特定帧。

弹出调用堆栈

可以从调用堆栈中删除停止于函数，从而使调用函数成为新的停止于函数。

与在调用堆栈中上下移动不同，弹出堆栈会更改程序的执行。从堆栈中删除停止于函数后，该函数会使程序恢复到其先前状态，但对全局或静态变量、外部文件、共享成员及类似全局状态的更改不会恢复到先前状态。

`pop` 命令用于从调用堆栈中删除一个或多个帧。例如，要从堆栈中弹出五个帧：

```
pop 5
```

也可以弹到特定帧。要弹到第 5 帧，请键入：

```
pop -f 5
```

有关更多信息，请参见“pop 命令” [328]。

隐藏堆栈帧

使用 `hide` 命令可列出当前正在使用的堆栈帧过滤器。

要隐藏或删除所有与正则表达式匹配的堆栈帧：

```
hide [ regular-expression ]
```

正则表达式与函数名称或装入对象的名称匹配，并使用 `sh` 或 `ksh` 语法进行文件匹配。

可使用 `unhide` 命令删除所有堆栈帧过滤器。

```
unhide 0
```

由于 hide 命令列出过滤器时会列出相应号码，因此还可以在使用 unhide 命令时使用过滤器号码。

unhide [*number* | *regular-expression*]

显示和读取堆栈跟踪

堆栈跟踪显示程序流中执行停止位置及执行到达此点的过程。它提供了有关程序状态的最简明的描述。

要显示堆栈跟踪，请使用 where 命令。

对于使用 -g 选项编译的函数，由于参数的名称和类型已知，因此显示的是准确值。对于无调试信息的函数，显示的参数值是十六进制数。这些数字未必都有意义。通过函数指针 0 进行函数调用时，函数值显示为一个小的十六进制数，而非符号名。

可以在未使用 -g 选项编译的函数中停止。在此类函数中停止时，dbx 将在堆栈中向下搜索其函数是使用 -g 选项编译的第一个帧，并设置其当前作用域。此停止于函数用箭头符号 (=>) 表示。

在以下示例中，main（） 在编译时带有 -g 选项，因此会显示参数的符号名称以及值。main（） 调用的库函数编译时不带 -g 选项，因此虽然显示了函数的符号名称，但对参数显示的是 SPARC 输入寄存器 $i0 至 $i5 的十六进制内容。

在以下示例中，程序因段故障而崩溃。原因很可能是 SPARC 输入寄存器 $i0 中的 strlen() 的参数为空。

```
(dbx) run
Running: Cdlib
(process id 6723)

CD Library Statistics:

 Titles:         1

 Total time:     0:00:00
 Average time:   0:00:00

signal SEGV (no mapping at the fault address) in strlen at 0xff2b6c5c
0xff2b6c5c: strlen+0x0080:    ld      [%o1], %o2
Current function is main
(dbx) where
  [1] strlen(0x0, 0x0, 0x11795, 0x7efefeff, 0x81010100, 0xff339323), at 0xff2b6c5c
  [2] _doprnt(0x11799, 0x0, 0x0, 0x0, 0x0, 0xff00), at 0xff2fec18
  [3] printf(0x11784, 0xff336264, 0xff336274, 0xff339b94, 0xff331f98, 0xff00), at 0xff300780
=>[4] main(argc = 1, argv = 0xffbef894), line 133 in "Cdlib.c"
(dbx)
```

有关更多堆栈跟踪示例，请参见“查看调用堆栈” [31] 和“跟踪调用” [190]。

♦ ♦ ♦ 第 8 章

求值和显示数据

本章介绍两种类型的数据检查：求值数据和显示数据。
本章包含以下各节：

求变量和表达式的值

本节讨论如何使用 dbx 对变量和表达式求值。

验证 dbx 使用的变量

如果不确定 dbx 对其求值的变量，可使用 which 命令查看 dbx 使用的全限定名。

要查看在其中定义变量名的其他函数和文件，请使用 whereis 命令。

有关这些命令的信息，请参见“which 命令” [372] 和“whereis 命令” [371]。

当前函数作用域之外的变量

要对当前函数的作用域之外的变量进行求值或监视时，请执行下列操作之一：

- 限定函数名。请参见“使用作用域转换操作符限定符号” [65]。例如：

```
(dbx) print 'item
```

- 通过更改当前函数来访问该函数。请参见“导航到代码” [61]。

输出变量、表达式或标识符的值

表达式应遵循当前语言的语法，但 dbx 中引入以用来处理作用域和数组的元语法除外。

使用输出命令可对本地代码中的变量或表达式求值：

print *expression*

可以使用 print 命令对 Java 代码中的表达式、局部变量或参数求值。

有关更多信息，请参见“print 命令” [329]。

注 - dbx 支持 C++ dynamic_cast 和 typeid 操作符。使用这两个操作符求表达式的值时，dbx 会调用由编译器提供的一些运行时类型识别函数。如果源代码没有明确使用这些操作符，编译器便不会生成这些函数，因此 dbx 将无法求表达式的值。

输出 C++ 指针

在 C++ 中，对象指针有两种类型：静态类型（在源代码中定义）和动态类型（对象进行类型转换之前的状况）。dbx 有时可以提供有关对象的动态类型的信息。

通常情况下，如果对象有虚拟函数表（即 vtable），dbx 便可使用 vtable 中的信息正确地确定对象的类型。

您可以将 print 命令、display 命令或 watch 命令与 -r（递归）选项结合使用。dbx 将显示通过类直接定义的所有数据成员以及从基类继承的所有数据成员。

这些命令还可采用 -d 或 +d 选项，切换 dbxenv 变量 output_dynamic_type 的缺省行为。

使用 -d 标志或将 dbxenv 变量 output_dynamic_type 设置为 on 时，如果没有进程运行，则将生成 program is not active 错误消息。同样，当您正在调试信息转储文件时，如果没有进程，则无法访问动态信息。如果尝试通过虚拟继承来查找动态类型，将生成 illegal cast on class pointers 错误消息。在 C++ 中，从虚拟基类转换到派生类是非法的。

对 C++ 程序中未命名参数求值

在 C++ 中，可以定义带有未命名参数的函数。例如：

```
void tester(int)
```

```
{
};
main(int, char **)
{
   tester(1);
};
```

虽然未命名参数不能在程序中的其他地方使用，但是编译器会按某种格式对未命名参数进行编码，让您可以对其求值。该格式如下（这里，编译器为 %n 赋整数值）：

```
_ARG%n
```

要获取由编译器分配的函数名称，请使用 whatis 命令，并用函数名称作为其目标。

```
(dbx) whatis tester
void tester(int _ARG1);
(dbx) whatis main
int main(int _ARG1, char **_ARG2);
```

有关更多信息，请参见“whatis 命令” [366]。

要对未命名的函数参数求值（或显示）：

```
(dbx) print _ARG1
_ARG1 = 4
```

解除指针引用

解除指针引用时，请查看该指针指向的容器的内容。

解除指针引用时，dbx 会在命令窗格中显示求值；在下面的示例中，即由 t 指向的值：

```
(dbx) print *t
*t = {
a = 4
}
```

监视表达式

监视每次程序停止时表达式的值是一种了解特定表达式或变量的变化情况和变化时间的有效方法。display 命令可指示 dbx 监视一个或多个指定的表达式或变量。监视会一直持续进行，直至使用 undisplay 命令将其停止为止。watch 命令用于在每个停止点处以相应点的当前作用域对表达式求值并进行输出。

使用 display 命令显示每次程序停止时变量或表达式的值：

display *expression*, ...

一次可以监视不止一个变量。如果 display 命令不带选项，则将输出显示的所有表达式的列表。

有关更多信息，请参见“display 命令” [296]。

使用 watch 命令可监视每个停止点处的表达式的值：

watch *expression*, ...

有关更多信息，请参见“watch 命令” [365]。

停止显示（取消显示）

dbx 会一直显示监视的变量值，直至使用 undisplay 命令停止显示为止。可以停止指定的表达式的显示，也可以停止当前监视的所有表达式的显示。

要停止特定变量或表达式的显示：

undisplay *expression*

要停止显示当前监视的所有变量：

undisplay 0

有关更多信息，请参见“undisplay 命令” [361]。

为变量赋值

使用 assign 命令为变量赋值：

assign *variable* **=** *expression*

对数组求值

对数组求值的方法与对其他类型的变量求值的方法相同。

以下是 Fortran 数组样例的示例：

```
integer*4 arr(1:6, 4:7)
```

要对数组求值，可使用 `print` 命令。例如：

```
(dbx) print arr(2,4)
```

使用 `dbx print` 命令可以对大型数组的一部分求值。数组求值包括：

- 数组分片 – 输出多维数组的任意矩形形状的 *n* 维区块。
- 数组跨距 – 按固定模式仅输出指定分片（可以是整个数组）中的某些元素。

进行数组分片时，可以使用跨距，也可以不使用跨距。（缺省跨距值为 1，即表示输出每个元素。）

数组分片

C、C++ 和 Fortran 语言中的 `print`、`display` 和 `watch` 命令支持数组分片。

C 和 C++ 的数组分片语法

对于数组的每个维度而言，用于对数组分片的 `print` 命令的完整语法如下所示：

`print` *array-expression* `[`*first-expression* `..` *last-expression* `:` *stride-expression*`]`

其中：

array-expression	求值结果应为数组或指针类型的表达式。
first-expression	要输出的第一个元素。缺省值为0。
last-expression	要输出的最后一个元素。缺省值为数组上界。
stride-expression	跨距长度（跳过的元素个数为 *stride-expression*`-1`）。缺省值为 1。

第一个表达式、最后一个表达式和跨距表达式都是可选表达式，它们的求值结果应为整数。

例如：

```
(dbx) print arr[2..4]
arr[2..4] =
[2] = 2
[3] = 3
```

```
[4] = 4
(dbx) print arr[..2]
arr[0..2] =
[0] = 0
[1] = 1
[2] = 2

(dbx) print arr[2..6:2]
arr[2..6:2] =
[2] = 2
[4] = 4
[6] = 6
```

Fortran 数组分片语法

对于数组的每个维度而言，用于对数组分片的 `print` 命令的完整语法如下所示：

`print` *array-expression*（*first-expression* : *last-expression* : *stride-expression*）

其中：

array-expression 求值结果应为数组类型的表达式。

first-expression 某个范围内的第一个元素，也是要输出的第一个元素。缺省值为数组下界。

last-expression 某个范围内的最后一个元素，但如果跨距不等于 1，则可能不是要输出的最后一个元素。缺省值为数组上界。

stride-expression 跨距长度。缺省值为 1。

第一个表达式、最后一个表达式和跨距表达式都是可选表达式，它们的求值结果应为整数。对于 *n* 维分片，请用逗号分隔各分片的定义。

例如：

```
(dbx) print arr(2:6)
arr(2:6) =
(2) 2
(3) 3
(4) 4
(5) 5
(6) 6

(dbx) print arr(2:6:2)
arr(2:6:2) =
(2) 2
(4) 4
(6) 6
```

要指定行和列：

```
demo% f95 -g -silent ShoSli.f
demo% dbx a.out
Reading symbolic information for a.out
(dbx) list 1,12
    1         INTEGER*4 a(3,4), col, row
    2         DO row = 1,3
    3             DO col = 1,4
    4               a(row,col) = (row*10) + col
    5             END DO
    6         END DO
    7         DO row = 1, 3
    8                WRITE(*,'(4I3)') (a(row,col),col=1,4)
    9        END DO
    10        END
(dbx) stop at 7
(1) stop at "ShoSli.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "ShoSli.f"
    7          DO row = 1, 3
```

要输出第 3 行：

```
(dbx) print a(3:3,1:4)
'ShoSli'MAIN'a(3:3, 1:4) =
       (3,1)   31
       (3,2)   32
       (3,3)   33
       (3,4)   34
(dbx)
```

要输出第 4 列：

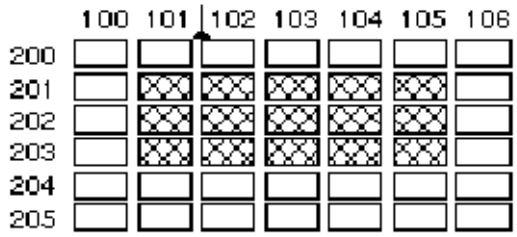
```
(dbx) print a(1:3,4:4)
'ShoSli'MAIN'a(1:3, 1:4) =
       (1,4)   14
       (2,4)   24
       (3,4)   34
(dbx)
```

使用分片

以下是一个二维矩形 C++ 数组分片示例（省略了缺省跨距 1）。

```
print arr[201..203][101..105]
```

此命令输出大型数组的一部分元素。请注意，该命令省略了 *stride-expression*，使用的是缺省跨距值 1。



如上所示，前两个表达式 (201:203) 指定该二维数组的第一个维度中的分片（三行一列）。分片从第 201 行开始，到第 203 行结束。第二组表达式（以逗号与第一组表达式分开）用于定义第二个维度的分片。该分片从第 101 列开始，到第 105 列结束。

使用跨距

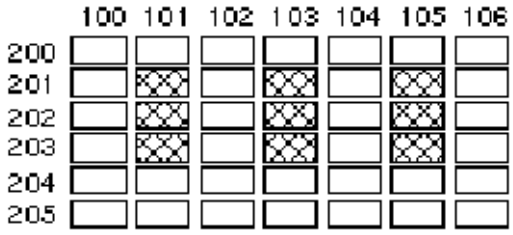
指示 print 跨越数组的分片时，dbx 只对该分片中的某些元素求值，而跳过对其求值的各个元素之间一定数量的元素。

数组分片语法中的第三个表达式 *stride-expression* 指定跨距长度。*stride-expression* 值指定要输出的元素。缺省跨距值为 1，即： 对指定分片中的所有元素求值。

以下示例是上一分片示例使用的数组。这一次，print 命令中，第二个维度中分片的跨距为 2。

```
print arr(201:203, 101:105:2)
```

如图所示，使用跨距 2 时，将输出所有第二个元素，而跳过所有其他元素。



对于省略的表达式，输出时取与数组的声明大小相等的缺省值。以下示例将说明如何使用简化语法。

对于一维数组，请使用下列命令：

`print arr`	使用缺省边界输出整个数组。
`print arr(:)`	使用缺省边界和缺省跨距 1 输出整个数组。
`print arr (::`*stride-expression*`)`	使用跨距 *stride-expression* 输出整个数组。

对于二维数组，可使用以下命令输出整个数组。

```
print arr
```

以下命令将输出二维数组中第二个维度的所有第三个元素：

```
print arr (:,::3)
```

使用美化输出

利用美化输出功能，程序可通过函数调用以自己的方式呈现表达式的值。dbx 支持两种美化输出机制：基于调用的美化输出和基于过滤器的美化输出。早期基于调用的机制的工作原理是调用调试对象中定义的符合特定模式的函数。当前版本的 dbx 现在支持基于 Python 的过滤器，从而允许用户创建可将值从一种形式转换为另一种形式的过滤器。

- “基于调用的美化输出” [114]
- “Python 美化输出过滤器 (Oracle Solaris)” [116]

dbx 将确定哪个机制结合使用 dbxenv 变量 output_pretty_print_mode。如果设置为 call，则将寻找基于调用的美化输出器。如果设置为 filter，则将寻找基于 Python 的美化输出器。如果设置为 filter_unless_call，则优先采用基于调用的美化输出器，而不是过滤器。

如果将 -p 选项指定为 print 命令、rprint 命令、display 命令或 watch 命令，则调用美化输出器（无论它属于何种类型）。有关调用美化输出器的更多信息，请参见“调用美化输出” [113]。

如果将 dbxenv 变量 output_pretty_print 设置为 on，则 -p 将作为缺省值传递给 print 命令、rprint 命令或 display 命令。可使用 +p 来覆盖此行为。此外，output_pretty_print 控制 IDE 局部变量、气球表达式求值和监视的美化输出。

调用美化输出

以下情况下会调用美化输出函数：

- 执行 print -p 或当 dbxenv 变量 output_pretty_print 设置为 on 时。

- 执行 display -p 或当 dbxenv 变量 output_pretty_print 设置为 on 时。
- 执行 watch -p 或当 dbxenv 变量 output_pretty_print 设置为 on 时。
- 使用气球表达式求值，当 dbxenv 变量 output_pretty_print 设置为 on 时。
- 使用局部变量，当 dbxenv 变量 output_pretty_print 设置为 on 时。

以下情况下不会调用美化输出函数：

- $[]. $[] 专门用于脚本中，因此脚本应当是可以预测的。
- 执行 dump 命令。dump 与 where 命令使用相同的简化格式，可能会在未来的版本中转为使用美化输出。IDE 的 "Local Variables"（局部变量）窗口不存在这一限制。

基于调用的美化输出

利用基于调用的美化输出功能，应用程序可通过函数调用以自己的方式呈现表达式值。如果在 print 命令、rprint 命令、 display 命令或 watch 命令中指定 -p 选项，则 dbx 会搜索格式为 const chars *db_pretty_print(const T *, int flags, const char *fmt) 的函数并调用它，从而替换 print 或 display 的返回值。

以该函数的 flags 参数传递的值是按位操作值或以下所列之一：

FVERBOSE	0x1	目前尚未实现，始终设置
FDYNAMIC	0x2	-d
FRECURSE	0x4	-r
FFORMAT	0x8	-f （如果设置，fmt 是格式部分）
FLITERAL	0x10	-l

db_pretty_print（） 函数可以是静态成员函数，也可以是独立函数。

美化输出时，还要考虑以下信息：

- “可能的故障” [115]
- “美化输出函数的注意事项” [115]
- 在 dbx 8.0 版本之前，美化输出基于 prettyprint 的 ksh 实现。虽然此 ksh 函数（及其预定义别名 pp）仍然存在， 但大部分语义已在 dbx 重新实现，结果如下：
  - 对于 IDE，可以对监视、局部变量和气球表达式求值使用美化输出。
  - 在 print 命令、display 命令和 watch 命令中，-p 选项表示使用本地路由。
  - 提高了可伸缩性，尤其是现在可频繁调用美化输出，特别是用于监视和局部变量的情况下。
  - 可以更好地从表达式中得到地址。
  - 具有更好的错误恢复功能。

- 对于嵌套值，无法进行美化输出，因为 dbx 没有用于计算嵌套字段的地址的基础结构。
- 缺省情况下，dbxenv 变量 output_pretty_print_fallback 设置为 on，这表示如果美化输出失败，dbx 将回退为采用常规格式。如果在该环境变量设置为 off 的情况下美化输出失败，dbx 将仍发出错误消息。

美化输出函数的注意事项

使用美化输出函数时，您需考虑以下内容：

- 一般来讲，对于 const/volatile 非限定类型，db_pretty_print(int *, ...（）) 和 db_pretty_print(const int *, ...)（） 等函数视为不同函数。dbx 的重载解析方法是识别性但非强制性的：
  - 识别性 – 如果定义了声明为 int 和 const int 的变量，各变量会传送到适当的函数中。
  - 非强制性 – 如果只定义了一个 int 或 const int 变量，它们会传送到两种函数中。此行为不是特定于美化输出的，而是适用于任何调用。
- 必须使用 -g 选项编译 db_pretty_print（） 函数，因为 dbx 需要能够访问参数签名。
- 允许 db_pretty_print（） 函数返回 NULL。
- 传递给 db_pretty_print（） 函数的主指针可以确保是非 NULL 的，但它仍可能指向初始化不当的对象。
- db_pretty_print（） 函数需要通过其第一个参数的类型来消除岐义。在 C 中，可通过将函数写为静态文件来重载它们。

可能的故障

如果出现以下可检测并可恢复的情况之一，美化输出可能失败：

- 未找到美化输出函数。
- 无法获取要美化输出的表达式的地址。
- 函数调用未立即返回，这可能表示由于遇到错误对象时美化输出函数不够强健而导致出现段错误。它也可能表示遇到一个用户断点。
- 美化输出函数返回了 NULL。
- 美化输出函数返回了 dbx 无法间接使用的指针。
- 正在调试信息转储文件。

所有情况（函数调用未立即返回除外）下，这些失败都不会有任何提示，dbx 会回退为采用常规格式。但是，如果该 output_pretty_print_fallback dbxenv 变量设置为 off，则当美化输出失败时，dbx 将发出错误消息。

如果您使用 `print -p` 命令，而没有将 `dbxenv` 变量 `output_pretty_print` 设置为 `on`，则 `dbx` 会在中断的函数中停止，允许您诊断故障的原因。然后可使用 `pop-c` 命令来清除此调用。

Python 美化输出过滤器 (Oracle Solaris)

利用美化输出过滤器功能，您可以通过 Python 编写过滤器，这些过滤器可将值从一种形式转换为另一种形式。基于 Python 的美化输出器仅在 Oracle Solaris 中可用。

注 - Python 美化输出过滤器仅可在 C 和 C++ 代码中使用，而无法在 Fortran 中使用。

系统已内置过滤器，用于在 C++ 标准模板库的 4 种实现中选择类。下表指定了库名和该库的编译器选项：

库的编译器选项	库名
`-library=Cstd`（缺省值）	`libCstd.so.1`
`-library=stlport4`	`libstlport.so.1`
`-library=stdcxx4`	`libstdcxx4.so.4.**`
`-library=stdcpp`（使用 `-std=c++11` 选项时的缺省值）	`libstdc++.so.6.*`

下表说明了美化输出过滤器在 C++ 标准模板库中可用于哪些类以及是否可以输出索引和分片：

类	是否输出索引和分片	C++ 兼容性
`string`	否	是
`pair`	否	是
`vector`	是	是
`list`	是	是
`set`	是	是
`bitset`	是	是
`map`	是	是
`stack`	是	是
`priority queue`	是	是
`queue`	是	是
`multimap`	是	是
`tuple`	否	仅限 C++11
`unique_ptr`	否	仅限 C++11

类	是否输出索引和分片	C++ 兼容性
forward_list	是	仅限 C++11
unordered_map	是	仅限 C++11
unordered_multimap	是	仅限 C++11
unordered_set	是	仅限 C++11
unordered_multiset	是	仅限 C++11
array	是	仅限 C++11
initializer_list	是	仅限 C++11

例 1 使用过滤器的美化输出

以下是一个在 dbx 中使用 print 命令时输出列表的输出示例：

```
(dbx) dbxenv output_pretty_print off
(dbx) print list10
list10 = {
   __buffer-size = 32U
   __buffer-list = {
     __data_ = 0x654a8
   }
   __free-list   = (nil)
   __next-avail  = 0x67334
   __last     = 0x67448
   __node     = 0x48830
   __length   = 10U
   }
```

以下是与 dbx 中输出的列表相同的列表，但该列表使用的是美化输出过滤器：

```
(dbx) print -p list10
list10 = (200, 201, 202, 203, 204, 205, 206, 207, 208, 209)

(dbx) print -p list10[5]
list10[5] = 205

(dbx) print -p list10[1..100:2]
list10[1..100:2] =
[1] = 202
[3] = 204
[5] = 206
[7] = 208
```

在 Oracle Solaris 上使用 Python

Python 美化输出过滤器和 python 命令仅在 Oracle Solaris 中可用。要启动内置的 Python 解释程序，请键入 python。要对 Python 代码求值，请键入 python *python-*

code。新生的 Python 插件 API 可用。但是，其主要用途是编写可以作为回调调用的美化输出器过滤器。因此 `python` 命令主要用于测试和诊断用途。

Python 美化输出 API 文档

要生成 Python 美化输出 API 文档，请使用 `python-docs` 命令。该命令仅在 Oracle Solaris 上可用。

♦ ♦ ♦ 第 9 章

9 使用运行时检查

利用运行时检查 (Runtime Checking, RTC) 可在开发阶段在本机代码应用程序中自动检测运行时错误（如内存访问错误和内存泄漏）。它还允许您监视内存使用情况。

本章阐述下列主题：

运行时检查功能

由于运行时检查是一种综合的调试功能，因此可在使用运行时检查功能时（使用收集器收集性能数据的情况除外）执行所有调试操作。

注 - 但不能对 Java 代码使用运行时检查。

运行时检查可提供以下功能：

- 检测内存访问错误
- 检测内存泄漏
- 收集内存使用情况数据

- 适用于所有语言环境
- 适用于多线程代码
- 无需重新编译、重新链接或进行 makefile 更改

使用 -g 标志进行编译，可提供运行时检查错误消息中的源代码行号相关性。运行时检查也可以检查使用优化 -O 标志进行编译的程序。对于不使用 -g 选项编译的程序，有一些特殊考虑事项。

可以通过 check 命令使用运行时检查功能。

何时使用运行时检查

要避免突然看到大量错误，请在开发周期的早期使用运行时检查，这是因为您正在开发的各个模块最终将组成一个程序。先编写一个单元测试来驱动每个模块，然后使用运行时检查以递增方式逐个检查模块。此方法意味着您可一次处理数量较少的错误。将所有模块集成为完整的程序时，遇到的新错误可能会很少。将错误数减少为零后，就只有在对模块进行了更改时，才需要再次使用运行时检查。

运行时检查要求

要使用运行时检查必须满足下列要求：

- 与 libc 动态链接。
- 使用标准 libc malloc、free 和 realloc 函数或基于这些函数的分配器。运行时检查提供了一个应用编程接口 (application programming interface, API) 来处理其他分配器。请参见“运行时检查应用编程接口” [140]。
- 可接受未完全剥离的程序以及使用 strip -x 剥离的程序。

有关运行时检查限制的信息，请参见“运行时检查限制” [142]。

使用运行时检查

要使用运行时检查，请在运行程序前启用要使用的检查类型。

启用内存使用和内存泄漏检查

使用以下命令可启用内存使用和内存泄漏检查：

```
(dbx) check -memuse
```

启用内存使用检查或内存泄漏检查后，showblock 命令将显示有关指定地址的堆块的详细信息。这些详细信息包括块分配的位置及其大小。有关更多信息，请参见“showblock 命令” [339]。

启用内存访问检查

使用以下命令仅启用内存访问检查：

```
(dbx) check -access
```

启用所有运行时检查

使用以下命令可启用内存泄漏、内存使用和内存访问检查：

```
(dbx) check -all
```

有关更多信息，请参见“check 命令” [276]。

禁用运行时检查

使用以下命令可完全禁用运行时检查：

```
(dbx) uncheck -all
```

有关详细信息，请参见“uncheck 命令” [360]。

运行程序

启用所需的运行时检查类型后，运行使用或未使用断点进行测试的程序。

程序正常运行，但速度很慢，因为每次进行内存访问前都要检查其有效性。如果 dbx 检测到无效访问，便会显示错误的类型和位置。控制权将交还给您，除非 dbxenv 变量 rtc_auto_continue 设置为 on。

然后便可发出 dbx 命令，例如执行 where 获取当前堆栈跟踪，或执行 print 检查变量。如果错误并非致命错误 则可使用 cont 命令继续执行程序。程序继续执行，直至遇到下一个错误或断点（无论先检测到哪一个）。有关详细信息，请参见“cont 命令” [287]。

如果 rtc_auto_continue dbxenv 变量设置为 on，则运行时检查会持续查找错误，并自动保持运行。它将错误重定向至 dbxenv 变量 rtc_error_log_file_name 命名的文件。缺省日志文件名称为 /tmp/dbx.errlog.*unique-ID*。

可以使用 suppress 命令限制报告运行时检查错误。有关详细信息，请参见“suppress 命令” [349]。

以下简单示例显示如何为名为 hello.c 的程序启用内存访问和内存使用检查。

```
% cat -n hello.c
     1 #include <stdio.h>
     2 #include <stdlib.h>
     3 #include <string.h>
     4
     5 char *hello1, *hello2;
     6
     7 void
     8 memory_use()
     9 {
    10      hello1 = (char *)malloc(32);
    11      strcpy(hello1, "hello world");
    12      hello2 = (char *)malloc(strlen(hello1)+1);
    13      strcpy(hello2, hello1);
    14 }
    15
    16 void
    17 memory_leak()
    18 {
    19      char *local;
    20      local = (char *)malloc(32);
    21      strcpy(local, "hello world");
    22 }
    23
    24 void
    25 access_error()
    26 {
    27      int i,j;
    28
    29      i = j;
    30 }
    31
    32 int
    33 main()
    34 {
    35      memory_use();
    36      access_error();
    37      memory_leak();
    38      printf("%s\n", hello2);
    39      return 0;
    40 }
% cc -g -o hello hello.c

% dbx -C hello
Reading ld.so.1
Reading librtc.so
Reading libc.so.1
Reading libdl.so.1
```

```
(dbx) check -access
access checking - ON
(dbx) check -memuse
memuse checking - ON
(dbx) run Running: hello
(process id 18306)
Enabling Error Checking... done
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff068
     which is 96 bytes above the current stack pointer
Variable is 'j'
Current function is access_error
    29       i = j;
(dbx) cont
hello world
Checking for memory leaks...
Actual leaks report    (actual leaks:         1 total size:      32 bytes)

 Total      Num of  Leaked     Allocation call stack
 Size       Blocks  Block
                    Address
==========  ====== ==========  =======================================
        32       1    0x21aa8  memory_leak < main

Possible leaks report  (possible leaks:       0  total size:      0 bytes)

Checking for memory use...
Blocks in use report   (blocks in use:        2  total size:      44 bytes

 Total     % of Num of  Avg    Allocation call stack
 Size      All  Blocks  Size
========== ==== ====== ======  =======================================
        32  72%      1     32  memory_use < main
        12  27%      1     12  memory_use < main

execution completed, exit code is 0
```

函数 `access_error()` 在变量 `j` 被初始化前便读取它。运行时检查将此访问错误报告为 "Read from uninitialized"（从未初始化项读取）(`rui`) 错误。

函数 `memory_leak()` 不会在返回前释放变量 `local`。`memory_leak()` 返回时，此变量超出作用域，在第 20 行分配的块变为泄漏。

程序使用全局变量 `hello1` 和 `hello2`，这两个变量始终处于作用域内。它们都指向动态分配的内存，这种情况报告为使用的块 (`biu`)。

使用内存访问

访问检查通过监视每个读取、写入、分配和释放操作，检查程序是否正确访问内存。

程序可能会以各种方式错误读取或写入内存，这称为内存访问错误。例如，程序可能通过 `free()` 调用堆块，引用已取消分配的内存块。或者，函数可能会将指针返回局部变量，而在访问该指针时，将产生错误。访问错误可能会导致程序中指针混乱，并可导致程序行为异常，包括输出错误和段违规。有些类型的内存访问错误很难发现。

运行时检查保存有一个跟踪程序使用的每个内存块状态的表。运行时检查会将每个内存操作与其涉及的内存块的状态进行对照，然后确定相应操作是否有效。可能的内存状态为：

- Unallocated, initial state（未分配，初始状态）。尚未分配内存。读取、写入或释放此内存是非法操作，因为它不归程序所有。
- Allocated, but uninitialized（已分配，但尚未初始化）。已为程序分配了内存，但尚未初始化此内存。写入或释放此内存是合法操作，但读取是非法操作，因为尚未初始化此内存。例如，输入函数时，已分配局部变量的堆栈内存，但未初始化此内存。
- Read-only（只读）。读取只读内存是合法操作，但写入或释放只读内存是非法操作。
- Allocated and initialized（已分配且已初始化）。读取、写入或释放已分配且已初始化的内存是合法操作。

使用运行时检查来查找内存访问错误与使用编译器查找程序中的语法错误没有什么不同。在这两种情况下，都会生成错误列表，并提供与每个错误对应的错误消息，说明出错原因和程序中的出错位置。在这两种情况下，应该从错误列表的顶部开始依次向下修复程序中的错误。在链锁反应下，一个错误可导致其他错误发生。因此，该链中的第一个错误是“首要原因”，修复该错误也可能会修复一些后续错误。

例如，从未初始化的内存区进行读取会创建不正确的指针，这样，取消其引用时，便会导致出现其他无效的读取或写入，而这又会导致出现另一个错误。

理解内存访问错误报告

运行时检查可为内存访问错误提供以下信息：

`type`	错误类型。
`access`	尝试访问的类型（读取或写入）。
`size`	尝试访问的大小。
`address`	尝试访问的地址。
`size`	泄漏块的大小。
`detail`	有关地址的更多详细信息。例如，如果地址在邻近堆栈中，便会提供其相对当前堆栈指针的位置。如果地址在堆中，便会提供最近堆块的地址、大小和相对位置。

`stack`	出错时调用堆栈（批处理模式）。
`allocation`	如果地址在堆中，便会提供最近堆块的分配跟踪。
`location`	出错位置。如果有行号信息，则此信息包括行号和函数。如果无行号，运行时检查会提供函数和地址。

以下示例显示的是一个典型的访问错误。

```
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xefffee50
    which is 96 bytes above the current stack pointer
Variable is ”j’
Current function is rui
   12           i = j;
```

内存访问错误

运行时检查可检测到以下内存访问错误：

- `rui` – 请参见“从未初始化的内存中读 (rui) 错误” [147]
- `rua` – 请参见“从未分配的内存中读 (rua) 错误” [147]
- `rob` – 请参见“从数组越界中读 (rob) 错误” [146]
- `wua` – 请参见“写入到未分配内存 (wua) 错误” [148]
- `wro` – 请参见“写入到只读内存 (wro) 错误” [147]
- `wob` – 请参见“写入到数组越界内存 (wob) 错误” [147]
- `mar` – 请参见“未对齐读 (mar) 错误” [146]
- `maw` – 请参见“未对齐写 (maw) 错误” [146]
- `duf` – 请参见“重复释放 (duf) 错误” [145]
- `baf` – 请参见“错误释放 (baf) 错误” [145]
- `maf` – 请参见“未对齐释放 (maf) 错误” [145]
- `oom` – 请参见“内存不足 (oom) 错误” [146]

注 - 在 SPARC 平台上，运行时检查不执行数组边界检查，因此不会将数组边界违规行为报告为访问错误。

使用内存泄漏检查

内存泄漏是动态分配的内存块，在程序的数据空间中任何位置都没有指向它的指针。这类块是孤立内存。由于任何指针均未指向块，因此程序无法引用这些块，也很少释放它们。运行时检查会查找并报告这类块。

内存泄漏会导致占用的虚拟内存增加，且通常会导致产生内存碎片。这可能会降低程序及整个系统的性能。

通常情况下，导致出现内存泄漏的原因是未释放分配的内存，而又丢失了指向分配块的指针。下面是一些内存泄漏示例：

```
void
foo()
{
    char *s;
    s = (char *) malloc(32);

    strcpy(s, "hello world");

    return; /* no free of s. Once foo returns, there is no     */
            /* pointer pointing to the malloc'ed block,         */
            /* so that block is leaked.                         */
}
```

API 使用不当会导致泄漏。

```
void
printcwd()
{

    printf("cwd = %s\n", getcwd(NULL, MAXPATHLEN));

    return; /* libc function getcwd() returns a pointer to     */
            /* malloc'ed area when the first argument is NULL, */
            /* program should remember to free this. In this   */
            /* case the block is not freed and results in leak.*/
}
```

总是在不再需要内存时便将其释放，并密切注意返回已分配内存的库函数，便可避免内存泄漏。如果使用这类函数，记得要适当地释放内存。

有时，内存泄漏一词用于指未释放的内存块。此定义用处不大，这是因为如果程序不久后即将终止，则常见的编程惯例是不释放内存。如果程序仍保留一个或多个指向块的指针，则运行时检查不会将该块报告为泄漏。

检测内存泄漏错误

运行时检查可检测到以下内存泄漏错误：

- mel – 请参见“内存泄漏 (mel) 错误” [149]
- air – 请参见“地址位于寄存器内 (air) 错误” [148]
- aib – 请参见“地址位于块内 (aib) 错误” [148]

注 - 运行时检查只查找 malloc 内存的泄漏。如果程序未使用 malloc，运行时检查便无法找到内存泄漏。

可能的泄漏

运行时检查可以在两种情况下报告“可能的”泄漏。第一种情况是任何指针均未指向块的开头处，但有一个指针指向块的内部。这种情况将报告为 "Address in block" 地址位于块内 (aib) 错误。指向该块的迷失指针是真正的内存泄漏。但是，某些程序会根据需要有意反复移动指向数组的唯一指针来访问其条目。这类情况不是内存泄漏。由于运行时检查无法区分这两种情况，因此会将这两种情况都按可能的泄漏来报告，由您来确定哪一种情况是真正的内存泄漏。

当数据空间中没有找到指向块的指针，但在寄存器中找到指针时，将发生第二种可能的泄漏。这种情况按“地址位于寄存器内 (air)”错误来报告。如果寄存器意外指向内存块或寄存器是后来丢失了的内存指针的旧副本，便是真正的泄漏。不过，编译器可以优化引用以及将指向内存块的唯一指针放入寄存器中，而不必将指针写入内存。这类情况便不是真正的泄漏。因此，如果程序已优化且报告是 showleaks 命令的结果，则这类情况很可能不是真正的泄漏。所有其他情况便可能是真正的泄漏。有关更多信息，请参见“showleaks 命令” [339]。

注 - 运行时泄漏检查要求使用标准 libc malloc/free/realloc 函数或基于这些函数的分配器。有关其他分配器，请参见“运行时检查应用编程接口” [140]。

检查泄漏

如果启用了内存泄漏检查，则会在所测试的程序即将退出之前，自动执行内存泄漏扫描。检测到的所有泄漏都会报告出来。不应使用 kill 命令中止程序。以下示例是一条典型的内存泄漏错误消息：

```
Memory leak (mel):
Found leaked block of size 6 at address 0x21718
At time of allocation, the call stack was:
```

```
[1] foo() at line 63 in test.c
[2] main() at line 47 in test.c
```

UNIX 程序具有 main 过程（在 f77 中称为 MAIN），该过程是程序的顶级用户函数。程序通常以两种方式终止：一种是调用 exit(3)，另一种是从 main 返回。在后一种情况下，main 的所有局部变量都会在返回后超出作用域，而它们指向的所有堆块都会按泄漏来报告（除非有全局变量也指向这些块）。

常见的编程惯例是不释放已分配给 main 中局部变量的堆块，这是因为程序即将终止，并在不调用 exit() 的情况下从 main 中返回。要防止运行时检查将这种块按内存泄漏来报告，请在 main 中最后一个可执行源代码行设置一个断点，以在 main 返回前停止程序。当程序在该处停止时，使用 showleaks 命令报告所有真正的泄漏，而忽略仅由 main 中的变量超出作用域导致的泄漏。

有关更多信息，请参见“showleaks 命令” [339]。

理解内存泄漏报告

启用泄漏检查之后，您将在程序退出时收到一份自动泄漏报告。所有可能的泄漏都会报告出来，但前提是程序不是使用 kill 命令中止的。报告中的详细程度由 dbxenv 变量 rtc_mel_at_exit 控制。缺省情况下，将生成非详细的泄漏报告。

报告按泄漏的合并大小排序。先报告真正的内存泄漏，然后报告可能的泄漏。详细报告包含详细的堆栈跟踪信息，其中包括行号和可用的源文件。

两种报告都包括内存泄漏错误的下列信息：

Size（大小）	泄漏块的大小
Location（位置）	泄漏块被分配到的位置
Address（地址）	泄漏块的地址
Stack（堆栈）	分配时调用堆栈，受 check -frames 限制

以下是对应的非详细内存泄漏报告。

```
Actual leaks report    (actual leaks:    3 total size:    2427 bytes)

 Total      Num of  Leaked      Allocation call stack
 Size       Blocks  Block
                    Address
==========  ======  ==========  =====================================
      1852       2      -      true_leak < true_leak
       575       1    0x22150  true_leak < main

Possible leaks report  (possible leaks:  1  total size:       8 bytes)
```

```
 Total      Num of  Leaked      Allocation call stack
 Size       Blocks  Block
                    Address
==========  ====== ==========  ======================================
         8       1    0x219b0  in_block < main
```

以下示例显示了一个典型的详细泄漏报告。

```
Actual leaks report    (actual leaks:         3  total size:    2427 bytes)

Memory Leak (mel):
Found 2 leaked blocks with total size 1852 bytes
At time of each allocation, the call stack was:
    [1] true_leak() at line 220 in "leaks.c"
    [2] true_leak() at line 224 in "leaks.c"

Memory Leak (mel):
Found leaked block of size 575 bytes at address 0x22150
At time of allocation, the call stack was:
    [1] true_leak() at line 220 in "leaks.c"
    [2] main() at line 87 in "leaks.c"

Possible leaks report  (possible leaks:       1  total size:       8 bytes)

Possible memory leak -- address in block (aib):
Found leaked block of size 8 bytes at address 0x219b0
At time of allocation, the call stack was:
    [1] in_block() at line 177 in "leaks.c"
    [2] main() at line 100 in "leaks.c"
```

生成泄漏报告

您可以使用 showleaks 命令随时查看泄漏报告，该报告将报告自上次执行 showleaks 命令后的新内存泄漏。有关更多信息，请参见“showleaks 命令” [339]。

组合泄漏

由于单个泄漏的数量可能会非常大，因此运行时检查会自动将同一位置分配的泄漏合并到一个合并泄漏报告中。决定是组合泄漏、还是分别报告，这是由 number-of-frames-to-match 参数控制的，您可在 check -leaks 上的 -match *m* 选项或 showleaks 命令的 -m 选项中指定该参数。如果两个或更多泄漏分配时的调用堆栈与 *m* 帧在严格程序计数器等级匹配，便会在一个合并泄漏报告中报告这些泄漏。

假设有下列三个调用序列：

块 1	块 2	块 3
[1] malloc	[1] malloc	[1] malloc

块 1	块 2	块 3
[2] d() at 0x20000	[2] d() at 0x20000	[2] d() at 0x20000
[3] c() at 0x30000	[3] c() at 0x30000	[3] c() at 0x31000
[4] b() at 0x40000	[4] b() at 0x41000	[4] b() at 0x40000
[5] a() at 0x50000	[5] a() at 0x50000	[5] a() at 0x50000

如果所有这些块均导致内存泄漏，则 *m* 值决定泄漏按单独泄漏还是一个重复泄漏来报告。如果 *m* 为 2，则块 1 和块 2 按一个重复泄漏来报告，因为两个调用序列 `malloc()` 上的 2 个堆栈帧相同。块 3 将按单独泄漏来报告，因为 `c()` 的跟踪与其他块不匹配。如果 *m* 大于 2，则运行时检查将所有泄漏报告为单独的泄漏。`malloc` 不会显示在泄漏报告中。

一般情况下，*m* 值越小，生成的单个泄漏报告越少，合并泄漏报告越多。*m* 值越大，生成的合并泄漏报告越少，单个泄漏报告越多。

修复内存泄漏

获得内存泄漏报告后，按下列指导修复内存泄漏：

- 最重要的是确定泄漏位置。泄漏报告会提供泄漏块的分配跟踪，即泄漏块的分配位置。
- 然后可以查看程序的执行流程，了解块的使用情况。如果指针丢失位置很明显，便很好解决；否则，可以使用 `showleaks` 来缩小泄漏时段。缺省情况下，`showleaks` 命令仅列出自上次执行 `showleaks` 命令后创建的新泄漏。可以在单步执行程序的同时重复运行 `showleaks` 以缩小内存块泄漏时段。

有关更多信息，请参见“`showleaks` 命令” [339]。

利用内存使用检查

使用内存使用检查，您可以查看所有使用的堆内存。可以通过此信息大致了解程序中内存的分配位置或程序的哪些部分在使用动态性最强的内存。另外，在减少程序占用的动态内存时，此信息很有用，并且在进行性能优化时，此信息也可能有用。

内存使用检查在性能优化过程中或对控制虚拟内存使用很有用。程序退出时，便可生成内存使用报告。您也可以使用 `showmemuse` 命令在程序执行期间随时获取内存使用情况信息，该命令可显示内存使用情况。有关信息，请参见“`showmemuse` 命令” [340]。

启用内存使用检查时，也会启用泄露检查。除了程序退出时的泄漏报告外，您还将获得一份使用的块 (`biu`) 报告。缺省情况下，系统会在程序退出时生成非详细的使用的块报告。内存使用报告中的详细程度由 `dbxenv` 变量 `rtc_biu_at_exit` 控制。

以下示例显示了典型的非详细内存使用报告。

```
Blocks in use report   (blocks in use: 5   total size:   40 bytes)

 Total     % of Num of  Avg     Allocation call stack
 Size       All Blocks  Size
========== ==== ====== ======  ====================================
        16  40%      2      8  nonleak < nonleak
         8  20%      1      8  nonleak < main
         8  20%      1      8  cyclic_leaks < main
         8  20%      1      8  cyclic_leaks < main

Blocks in use report   (blocks in use: 5   total size:   40 bytes)

Block in use (biu):
Found 2 blocks totaling 16 bytes (40.00% of total; avg block size 8)
At time of each allocation, the call stack was:
     [1] nonleak() at line 182 in "memuse.c"
     [2] nonleak() at line 185 in "memuse.c"

Block in use (biu):
Found block of size 8 bytes at address 0x21898 (20.00% of total)
At time of allocation, the call stack was:
     [1] nonleak() at line 182 in "memuse.c"
     [2] main() at line 74 in "memuse.c"

Block in use (biu):
Found block of size 8 bytes at address 0x21958 (20.00% of total)
At time of allocation, the call stack was:
     [1] cyclic_leaks() at line 154 in "memuse.c"
     [2] main() at line 118 in "memuse.c"

Block in use (biu):
Found block of size 8 bytes at address 0x21978 (20.00% of total)
At time of allocation, the call stack was:
     [1] cyclic_leaks() at line 155 in "memuse.c"
     [2] main() at line 118 in "memuse.c"
The following is the corresponding verbose memory use report:
```

可以随时使用 `showmemuse` 命令获得内存使用报告。

抑制错误

运行时检查包括一个强大的错误禁止工具，可极为灵活地限制报告的错误数量和类型。如果发生被抑制的错误，则不会生成任何报告，程序会继续执行，就像没有发生错误一样。

可以使用 `suppress` 命令禁止错误。

可以使用 `unsuppress` 命令撤消错误禁止。

抑制在同一调试会话期间内的各 run 命令中有效，但在各 debug 命令之间，抑制作用无关。

禁止的类型

本节介绍了可用的禁止类型：

按作用域和类型抑制

必须指定要抑制的错误类型。可以指定要抑制的程序部分。选项包括：

Global（全局）	缺省值，应用于整个程序
Load Object（装入对象）	应用于整个装入对象（如共享库）或主程序
File（文件）	应用于特定文件中的所有函数
Function（函数）	应用于特定函数
Line（行）	应用于特定源代码行
Address（地址）	应用于某地址处的特定指令

抑制上一错误

缺省情况下，运行时检查将禁止最近的错误，防止重复报告相同的错误。此设置由 dbx 变量 rtc_auto_suppress 控制。当 rtc_auto_suppress 设置为 on（缺省值）时，在特定位置出现的特定访问错误只在首次出现时报告，此后抑制报告。此设置非常有用，例如，执行多次的循环中出现错误时，可用于防止出现同一错误报告的多个副本。

限制报告的错误数

您可以使用 dbxenv 变量 rtc_error_limit 限制将报告的错误数。错误限制分别用于访问错误和泄漏错误。例如，如果错误限制设置为 5，则在运行结束和您发出每条 showleaks 命令时生成的泄漏报告中，最多显示 5 个访问错误和 5 个内存泄漏。缺省值为 1000。

禁止错误示例

在以下示例中，main.cc 是文件名，foo 和 bar 是函数，a.out 是可执行文件的名称。

不报告在函数 foo 中发生分配的内存泄漏：

```
suppress mel in foo
```

禁止报告从 libc.so.1 中分配的使用的块：

```
suppress biu in libc.so.1
```

禁止从 a.out 的所有函数中未初始化的内存中读取：

```
suppress rui in a.out
```

不报告从文件 main.cc 中未分配的内存中读取：

```
suppress rua in main.cc
```

禁止在 main.cc 第 10 行重复释放：

```
suppress duf at main.cc:10
```

抑制报告函数 bar 中的所有错误：

```
suppress all in bar
```

有关更多信息，请参见“suppress 命令” [349]。

缺省禁止

为了检测所有错误，运行时检查不要求使用 -g 选项（符号）编译程序。但是，为保证准确检测某些错误（主要是 rui 错误），有时会需要符号信息。为此，如果没有符号信息，缺省情况下，会抑制某些错误（a.out 的 rui 以及共享库的 rui、aib 和 air）。可以使用 suppress 命令和 unsuppress 命令的 -d 选项更改此行为。

如果使用以下命令，运行时检查将不再抑制在无符号信息（编译时未使用 -g）的代码中从未初始化的内存中读取 (rui)：

```
unsuppress -d rui
```

有关更多信息，请参见“unsuppress 命令” [363]。

使用抑制来管理错误

初次在大型程序上运行时，可能出现无法应付的大量错误。考虑采用分阶段的方法。为此，您可以使用 suppress 命令将报告的错误数减少至可管理的数量，仅修复这些错误并重复循环。这样一来，您可以通过每次迭代减少错误数。

例如，可以每次侧重处理几个类型的错误。通常遇到的最常见错误类型是 rui、rua 和 wua，而且通常是按该顺序出现。rui 错误最不严重（尽管它们可能会导致以后出现较严

重的错误）。通常，程序在遇到这些错误时可能仍会正常运行。rua 和 wua 错误比较严重，因为它们是通过无效内存地址进行的访问，而且总是指示编码错误。

可以先抑制 rui 和 rua 错误。修复出现的所有 wua 错误后，再次运行程序，仅抑制 rui 错误。修复出现的所有 rua 错误后，再次运行程序，不抑制错误。修复所有 rui 错误。最后，再一次运行程序，确保无残余错误。

如果要抑制上一次报告的错误，请使用 suppress -last。

对子进程使用运行时检查

要对子进程使用运行时检查，必须将 dbxenv 变量 rtc_inherit 设置为 on。缺省情况下，该变量设置为 off。

如果针对父进程启用了运行时检查，且 dbx 变量 follow_fork_mode 设置为 child，则 dbxenv 支持对子进程使用运行时检查。

发生派生时，dbx 会自动对子进程使用运行时检查。如果程序调用 exec()，则调用 exec() 的程序的运行时检查设置会传递给该程序。

在任一时刻，运行时检查只能控制一个进程，如以下示例所示。

```
% cat -n program1.c
     1 #include <sys/types.h>
     2 #include <unistd.h>
     3 #include <stdio.h>
     4
     5 int
     6 main()
     7 {
     8      pid_t child_pid;
     9      int parent_i, parent_j;
    10
    11      parent_i = parent_j;
    12
    13      child_pid = fork();
    14
    15      if (child_pid == -1) {
    16          printf("parent: Fork failed\n");
    17          return 1;
    18      } else if (child_pid == 0) {
    19          int child_i, child_j;
    20
    21          printf("child: In child\n");
    22          child_i = child_j;
    23          if (execl("./program2", NULL) == -1) {
    24              printf("child: exec of program2 failed\n");
    25              exit(1);
    26          }
```

```
    27      } else {
    28          printf("parent: child's pid = %d\n", child_pid);
    29      }
    30      return 0;
    31 }
```

```
 % cat -n program2.c
     1
     2 #include <stdio.h>
     3
     4 main()
     5 {
     6      int program2_i, program2_j;
     7
     8      printf ("program2: pid = %d\n", getpid());
     9      program2_i = program2_j;
    10
    11      malloc(8);
    12
    13      return 0;
    14 }
%
```

```
 % cc -g -o program1 program1.c
 % cc -g -o program2 program2.c
 % dbx -C program1
 Reading symbolic information for program1
 Reading symbolic information for rtld /usr/lib/ld.so.1
 Reading symbolic information for librtc.so
 Reading symbolic information for libc.so.1
 Reading symbolic information for libdl.so.1
 Reading symbolic information for libc_psr.so.1
 (dbx) check -all
 access checking - ON
 memuse checking - ON
 (dbx) dbxenv rtc_inherit on
 (dbx) dbxenv follow_fork_mode child
 (dbx) run
 Running: program1
 (process id 3885)
 Enabling Error Checking... done
```

RTC reports first error in the parent, `program1`

```
 Read from uninitialized (rui):
 Attempting to read 4 bytes at address 0xeffff110
     which is 104 bytes above the current stack pointer
 Variable is 'parent_j'
 Current function is main
   11       parent_i = parent_j;
(dbx) cont
 dbx: warning: Fork occurred; error checking disabled in parent
 detaching from process 3885
 Attached to process 3886
```

Because `follow_fork_mode` *is set to* `child`*, when the fork occurs error checking is switched from the parent*

to the child process

```
stopped in _fork at 0xef6b6040
0xef6b6040: _fork+0x0008:    bgeu    _fork+0x30
Current function is main
   13       child_pid = fork();
parent: child's pid = 3886
(dbx) cont
child: In child
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff108
    which is 96 bytes above the current stack pointer
```

RTC reports an error in the child

```
Variable is 'child_j'
Current function is main
   22       child_i = child_j;
(dbx) cont
dbx: process 3886 about to exec("./program2")
dbx: program "./program2" just exec'ed
dbx: to go back to the original program use "debug $oprog"
Reading symbolic information for program2
Skipping ld.so.1, already read
Skipping librtc.so, already read
Skipping libc.so.1, already read
Skipping libdl.so.1, already read
Skipping libc_psr.so.1, already read
```

When the exec of `program2` *occurs, the RTC settings are inherited by* `program2` *so access and memory use checking*
are enabled for that process

```
Enabling Error Checking... done
stopped in main at line 8 in file "program2.c"
    8       printf ("program2: pid = %d\n", getpid());
(dbx) cont
program2: pid = 3886
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff13c
    which is 100 bytes above the current stack pointer
```

RTC reports an access error in the executed program, `program2`

```
Variable is 'program2_j'
Current function is main
    9       program2_i = program2_j;
(dbx) cont
Checking for memory leaks...
```

RTC prints a memory use and memory leak report for the process that exited while under RTC control, `program2`

```
Actual leaks report   (actual leaks:      1  total size:   8
bytes)

Total      Num of  Leaked     Allocation call stack
Size       Blocks  Block
                   Address
==========  ====== ========== ===================================
         8       1    0x20c50  main
Possible leaks report  (possible leaks:   0  total size:   0
bytes)
```

```
execution completed, exit code is 0
```

对连接的进程使用运行时检查

除非因已分配受影响的内存而无法检测到 rui，否则可以对连接的进程使用运行时检查。

系统运行的 Oracle Solaris 上的连接进程

在运行 Oracle Solaris 操作系统的系统中，进程启动时必须预装入 rtcaudit.so。如果连接的进程是 64 位进程，请使用相应的 64 位 rtcaudit.so，该文件位于：

64 位 SPARC 平台：/*install-dir*/lib/dbx/sparcv9/runtime/rtcaudit.so

AMD64 平台：/*install-dir*/lib/dbx/amd64/runtime/rtcaudit.so

32 位平台：/*install-dir*/lib/dbx/runtime/rtcaudit.so

要预装入 rtcaudit.so：

```
% setenv LD_AUDIT path-to-rtcaudit/rtcaudit.so
```

将 LD_AUDIT 环境变量设置为仅在需要时预装入 rtcaudit.so。不要始终将其保持为装入状态。例如：

```
% setenv LD_AUDIT...
% start_your_application
% unsetenv LD_AUDIT
```

连接到进程后，便可以启用运行时检查。

如果要连接的程序从某一其他程序派生或执行，则必须为主程序（派生者）设置 LD_AUDIT。在派生和执行中会继承 LD_AUDIT 的设置。如果 32 位程序派生或执行 64 位程序，或 64 位程序派生或执行 32 位程序，则该解决方案将不起作用。

LC_AUDIT 环境变量既适用于 32 位程序也适用于 64 位程序，这导致很难为运行 64 位程序的 32 位程序或运行 32 位程序的 64 位程序选择正确的库。某些版本的 Oracle Solaris 操作系统支持 LD_AUDIT_32 环境变量和 LD_AUDIT_64 环境变量（它们分别仅影响 32 位程序和 64 位程序）。请参见针对您所运行 Oracle Solaris 版本的《链接程序和库指南》，确定是否支持这些变量。

运行 Linux 的系统上的连接进程

在运行 Linux 操作系统的系统中，进程启动时必须预装入 `librtc.so`。如果连接的进程是在 AMD64 处理器中运行的 64 位进程，请使用相应的 64 位 `librtc.so`，该文件位于：

64 位 AMD64 平台：`/`*install-dir*`/lib/dbx/amd64/runtime/librtc.so`

32 位 AMD64 平台 `/`*install-dir*`/lib/dbx/runtime/librtc.so`

要预装入 `librtc.so`：

```
% setenv LD_PRELOAD path-to-rtcaudit/librtc.so
```

将 `LD_PRELOAD` 环境变量设置为仅在需要时预装入 `librtc.so`。不要始终将其保持为装入状态。例如：

```
% setenv LD_PRELOAD...
% start_your_application
% unsetenv LD_PRELOAD
```

连接到进程后，便可以启用运行时检查。

如果要连接的程序从某一其他程序派生或执行，则必须为主程序（派生者）设置 `LD_PRELOAD`。在派生和执行中会继承 `LD_PRELOAD` 的设置。如果 32 位程序派生或执行 64 位程序，或 64 位程序派生或执行 32 位程序，则该解决方案将不起作用。

`LC_PRELOAD` 环境变量既适用于 32 位程序也适用于 64 位程序，这导致很难为运行 64 位程序的 32 位程序或运行 32 位程序的 64 位程序选择正确的库。某些版本的 Linux 支持 `LD_PRELOAD_32` 环境变量和 `LD_PRELOAD_64` 环境变量（它们分别仅影响 32 位程序和 64 位程序）。请参见针对您所运行 Linux 版本的《链接程序和库指南》，确定是否支持这些变量。

结合使用修复并继续功能与运行时检查

可以将运行时检查与 `fix` 和 `cont` 命令一起使用，以便快速查出并修复编程错误。修复并继续功能提供了强大的组合功能，可以为您节省大量调试时间。例如：

```
% cat -n bug.c
     1 #include stdio.h
     2 char *s = NULL;
     3
     4 void
     5 problem()
     6 {
     7      *s = 'c';
     8 }
     9
```

```
    10 main()
    11 {
    12      problem();
    13      return 0;
    14 }
% cat -n bug-fixed.c
     1 #include stdio.h
     2 char *s = NULL;
     3
     4 void
     5 problem()
     6 {
     7
     8      s = (char *)malloc(1);
     9      *s = 'c';
    10 }
    11
    12 main()
    13 {
    14      problem();
    15      return 0;
    16 }
yourmachine46: cc -g bug.c
yourmachine47: dbx -C a.out
Reading symbolic information for a.out
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for librtc.so
Reading symbolic information for libc.so.1
Reading symbolic information for libintl.so.1
Reading symbolic information for libdl.so.1
Reading symbolic information for libw.so.1
(dbx) check -access
access checking - ON
(dbx) run
Running: a.out
(process id 15052)
Enabling Error Checking... done
Write to unallocated (wua):
Attempting to write 1 byte through NULL pointer
Current function is problem
    7       *s = 'c';
(dbx) pop
stopped in main at line 12 in file "bug.c"
   12       problem();
(dbx) #at this time we would edit the file; in this example just copy
the correct version
(dbx) cp bug_fixed.c bug.c
(dbx) fix
fixing "bug.c" ......
pc moved to "bug.c":14
stopped in main at line 14 in file "bug.c"
   14       problem();
(dbx) cont
```

```
execution completed, exit code is 0
(dbx) quit
The following modules in \Qa.out' have been changed (fixed):
bug.c
Remember to remake program.
```

有关使用修复和继续功能的更多信息，请参见"内存泄漏 (mel) 错误" [149]。

运行时检查应用编程接口

泄漏检测和访问检查都要求使用共享库 libc.so 中的标准堆管理例程，这样，运行时检查便可跟踪程序中所有内存分配和释放情况。许多应用程序中都有在 malloc() 或 free() 函数的基础上或独立编写而成的自己的内存管理例程。当您使用自己的分配器（称为专用分配器）时，运行时检查便无法自动跟踪它们。因此，您无法了解因不当使用而导致的泄漏和内存访问错误。

不过，运行时检查提供了一个 API 以便使用专用分配器。使用此 API 可将专用分配器视为标准堆分配器。该 API 本身在头文件 rtc-api.h 中提供，并作为 Oracle Developer Studio 软件的一部分进行分发。手册页 rtc_api(3x) 详细介绍了运行时检查 API 入口点。

专用分配器不使用程序堆时，运行时检查访问错误报告可能会存在一些细小差别。发生有关标准堆块的内存访问错误时，错误报告通常包括堆块分配的位置。专用分配器不使用程序堆时，错误报告可能不包括分配项。

不需要使用运行时检查 API 来跟踪 libumem 中的内存分配器。运行时检查会插入 libumem 堆管理例程并将这些例程重定向至相应的 libc 函数。

在批处理模式下使用运行时检查

bcheck 实用程序是 dbx 的运行时检查的方便的批处理接口。它在 dbx 下运行程序，并且在缺省情况下，将运行时检查错误输出放在缺省文件 program.errs 中。

bcheck 实用程序可以分别或一起执行内存泄漏检查、内存访问检查和内存使用检查。其缺省操作是只执行泄漏检查。有关其使用的更多详细信息，请参见 bcheck(1) 手册页。

注 - 在运行 64 位 Linux 操作系统的系统上运行 bcheck 实用程序之前，必须设置 _DBX_EXEC_32 环境变量。

---

bcheck 语法

bcheck 的语法如下：

```
bcheck [-V] [-access | -all | -leaks | -memuse] [-xexec32] [-o logfile] [-q]
[-s script] program [args]
```

-o *logfile* 选项用于为日志文件指定另一个名称。-s *script* 选项在执行程序前使用，用于在 *script* 文件中包含的 dbx 命令中进行读取。*script* 文件通常包含 suppress 和 dbxenv 等这类命令，用于调整 bcheck 实用程序的错误输出。

-q 选项可使 bcheck 实用程序处于完全静默状态，返回时的状态与程序相同。要在脚本或 makefiles 中使用 bcheck 实用程序时，此选项很有用。

bcheck 示例

要对 hello 仅执行泄漏检查：

```
bcheck hello
```

要使用参数 5 对 mach 仅执行访问检查：

```
bcheck -access mach 5
```

要以静默方式对 cc 执行内存使用检查，并以正常退出状态退出：

```
bcheck -memuse -q cc -c prog.c
```

在批处理模式下检测到运行时错误时，程序不会停止。所有错误输出都会重定向到错误日志文件 logfile 中。遇到断点或程序被中断时，程序会停止。

在批处理模式下，会生成完整的堆栈回溯，且其会重定向到错误日志文件。可使用 dbxenv 变量 stack_max_size 控制堆栈帧数。

如果文件 logfile 已存在，则 bcheck 会清除该文件的内容，然后将批处理输出重定向到该文件。

直接在 dbx 中启用批处理模式

您也可以通过设置 dbxenv 变量 rtc_auto_continue 和 rtc_error_log_file_name，从 dbx 中直接启用类似批处理模式。

如果 rtc_auto_continue 设置为 on，则运行时检查会继续查找错误，并自动保持运行。它将错误重定向至 dbxenv 变量 rtc_error_log_file_name 命名的文件。缺省日

志文件名称为 /tmp/dbx.errlog.*unique-ID*。要将所有错误都重定向到终端，请将 rtc_error_log_file_name 环境变量设置为 /dev/tty。

缺省情况下，rtc_auto_continue 设置为 off。

故障排除提示

为程序启用了错误检查并运行程序后，可能会检测到下列错误之一：

```
librtc.so and dbx version mismatch; Error checking disabled
```

对连接的进程使用运行时检查时，如果 LD_AUDIT 未设置为 Oracle Developer Studio dbx 映像文件所附带的 rtcaudit.so 的版本，可能会出现此错误。要修复此错误，请更改 LD_AUDIT 的设置。

```
patch area too far (8mb limitation); Access checking disabled
```

运行时检查找不到距装入对象足够近的修补空间以便启用访问检查。请参见“运行时检查限制” [142]。

运行时检查限制

本节介绍了运行时检查的限制。

使用更多的符号和调试信息提高性能

访问检查需要装入对象的某些符号信息。当装入对象被完全剥离时，运行时检查可能无法捕获所有的错误。从未初始化的 (rui) 内存错误进行读取可能会出错，因而会被抑制。可以使用 unsuppress rui 命令覆盖抑制。要保留装入对象的符合表，请在剥离装入对象时使用 -x 选项。

运行时检查无法捕获所有数组越界错误。如果没有调试信息，针对静态内存和堆栈内存的边界检查不可用。

SIGSEGV 和 SIGALTSTACK 信号在 x86 平台上受限制

运行时检查为访问检查检测内存访问指令。这些指令由 SIGSEGV 处理程序在运行时处理。由于运行时检查需要其自己的 SIGSEGV 处理程序和信号备用堆栈，所以尝试安装 SIGSEGV 处理程序或 SIGALTSTACK 处理程序会导致发生 EINVAL 错误或忽略该尝试。

SIGSEGV 处理程序的调用不能被嵌套。如果嵌套，则会导致 terminating signal 11 SEGSEGV 错误。如果收到此错误，请使用 rtc skippatch 命令跳过受影响函数的检测过程。

当 8 MB 的所有现有代码中具有足够的修补程序区时性能将提高（仅限 SPARC 平台）。

如果 8 MB 的所有现有代码中没有足够的修补程序区，则可能会引发两个问题。

- 缓慢

  如果启用了访问检查，dbx 会用分支到一个修补程序区的分支指令来替换每一个装入和存储指令。此分支指令寻址范围为 8 MB。如果被调试程序用完了替换的特定装入或存储指令的所有 8 MB 地址空间，将没有空间来存放修补程序区。在这种情况下，dbx 会调用陷阱处理程序而不是使用分支。将控制权移交给陷阱处理程序的过程非常缓慢（最多会慢 10 倍），但不受 8 MB 限制。
- V8+ 模式中的输出寄存器覆盖问题

  如果同时符合以下两个条件，则陷阱处理程序限制会影响访问检查：

  - 被调试的进程使用陷阱进行检测。
  - 进程使用 V8+ 指令集。

  出现问题的原因是，V8+ 体系结构中输出寄存器和输入寄存器的大小不相同。输出寄存器的长度为 64 位，而输入寄存器仅为 32 位。当调用陷阱处理程序时，输出寄存器被复制到输入寄存器，而较高的 32 位会丢失。因此，如果被调试的进程使用输出寄存器的高 32 位，则启用访问检查时该进程可能会以不正确的方式运行。

  缺省情况下，创建基于 SPARC 的 32 位二进制文件时，编译器使用 V8+ 体系结构，但您可以使用 -xarch 选项来通知编译器使用 V8 体系结构。遗憾的是，重新编译应用程序不会影响系统运行时库。

  dbx 会自动跳过以下函数和库（已知使用陷阱进行检测时不能正常工作）的检测过程：

  - server/libjvm.so
  - client/libjvm.so
  - `libfsu_isa.so`__f_cvt_real
  - `libfsu_isa.so`__f90_slw_c4

  但是，跳过检测过程可能会导致不正确的 RTC 错误报告。

如果您的程序存在上述任何一种情况，并且在启用访问检查时程序的行为方式开始有所不同，则可能是陷阱处理程序限制影响到了您的程序。要变通克服这些限制，您可以执行以下操作：

- 使用 `rtc skippatch` 命令跳过对使用以上所列函数和库的程序代码的检测。通常，跟踪特定函数的问题很难，因此您可能希望跳过对整个装入对象的检测。`rtc showmap` 命令显示了按地址排序的检测类型映射。
- 尝试使用 64 位 SPARC-V9 而不使用 32 位 SPARC-V8。

  如果可能，重新编译 V9 体系结构的程序，在该体系结构中，所有的寄存器长度为 64 位。
- 尝试添加修补程序区对象文件。

  可以使用 `rtc_patch_area` shell 脚本创建特殊的 `.o` 文件，这些文件可以链接到大型可执行文件或共享库的中间，从而可以提供更多修补程序空间。有关更多信息，请参见 `rtc_patch_area`(1) 手册页。

  如果 `dbx` 达到 8 MB 限制，它会告知哪个装入对象过大（主程序还是共享库），并会显示该装入对象所需的修补程序空间总量。

  为了得到最佳结果，应在可执行文件或共享库中均匀分布这些特殊的修补程序对象文件，并应使用缺省大小 (8 MB) 或更小的大小。另外，添加的修补程序空间不要超过 `dbx` 指出的所需大小的 10-20%。例如，如果 dbx 指出需要 31 MB 用于 `a.out`，请添加使用 `rtc_patch_area` 脚本创建的四个对象文件，每个大小均为 8 MB，然后在可执行文件中大致平均地分布这些文件。

  如果 `dbx` 在可执行文件中找到显式修补程序区，则会输出这些修补程序区跨越的地址范围，这有助于将它们正确地置于链接行上。
- 尝试将较大的装入对象分解为较小的装入对象。

  将可执行文件或大型库中的对象文件分解成较小的多组对象文件，然后将这些文件链接至较小的部分。如果大型文件是可执行文件，则将其分解成较小的可执行文件和一系列共享库。如果大型文件是共享库，则将其重新安排成一组较小的库。

  `dbx` 可利用这一技术在不同的共享对象间为修补程序代码寻找空间。
- 尝试添加“填充”`.so` 文件。

  只有当要连接已启动的进程时，才需要采用此解决方案。

  运行时链接程序可能会将各库非常紧密地放置在一起，造成无法在库之间的间隙中创建修补程序空间。`dbx` 在启用了运行时检查的情况下启动可执行文件时，会要求运行时链接程序在共享库之间留出额外的间隙。但是，如果连接到的进程不是由 `dbx` 在启用了运行时检查的情况下所启动的，则这些库之间可能会靠得过近。

  如果运行时库之间太近，且如果无法使用 `dbx` 启动程序，则可以尝试使用 `rtc_patch_area` 脚本创建一个共享库，然后将其链接到其他共享库之间的程序。有关更多详细信息，请参见 `rtc_patch_area`(1) 手册页。

运行时检查错误

运行时检查报告的错误一般分两类：访问错误和泄漏。

访问错误

如果启用了访问检查，则运行时检查会检测并报告本节所述的各种类型的错误。

错误释放 (baf) 错误

问题： 尝试释放尚未分配的内存。

可能的原因： 将非堆数据指针传递给了 `free()` 或 `realloc()`。

示例：

```
char a[4];
char *b = &a[0];

free(b);                    /* Bad free (baf) */
```

重复释放 (duf) 错误

问题： 尝试释放已释放过的堆块。

可能的原因： 多次使用同一指针调用 `free()`。在 C++ 中，多次对同一指针使用删除操作符。

示例：

```
char *a = (char *)malloc(1);
free(a);
free(a);                    /* Duplicate free (duf) */
```

未对齐释放 (maf) 错误

问题： 尝试释放未对齐的堆块。

可能的原因：将未正确对齐的指针传递给了 `free()` 或 `realloc()`；更改了 `malloc` 返回的指针。

示例：

```
char *ptr = (char *)malloc(4);
ptr++;
free(ptr);                    /* Misaligned free */
```

未对齐读 (mar) 错误

问题： 尝试从未正确对齐的地址中读取数据。

可能的原因： 分别从那些没有半字对齐、字对齐或双字对齐的地址中读取 2 个、4 个或 8 个字节。

示例：

```
char *s = “hello world”;
int *i = (int *)&s[1];
int j;

j = *i;                    /* Misaligned read (mar) */
```

未对齐写 (maw) 错误

问题： 尝试将数据写入未正确对齐的地址。

可能的原因： 将 2 个、4 个或 8 个字节分别写入没有半字对齐、字对齐或双字对齐的地址。

示例：

```
char *s = “hello world”;
int *i = (int *)&s[1];

*i = 0;                    /* Misaligned write (maw) */
```

内存不足 (oom) 错误

问题： 尝试分配超出可用物理内存的内存。

原因： 程序无法从系统获得更多的内存。查找在未检查 `malloc()` 的返回值是否为 `NULL`（一个常见编程错误）时发生的问题时会有用。

示例：

```
char *ptr = (char *)malloc(0x7fffffff);
/* Out of Memory (oom), ptr == NULL */
```

从数组越界中读 (rob) 错误

问题：尝试从数组越界内存中进行读取。

可能的原因：溢出堆块边界的迷失指针。

示例：

```
char *cp = malloc (10);
char ch = cp[10];
```

从未分配的内存中读 (rua) 错误

问题： 尝试从不存在、未分配或未映射的内存中进行读取。

可能的原因： 溢出堆块边界或访问已被释放的堆块的迷失指针。

示例：

```
char *cp = malloc (10);
free (cp);
cp[0] = 0;
```

从未初始化的内存中读 (rui) 错误

问题： 尝试从未初始化的内存中进行读取。

可能的原因： 读取尚未初始化的局部数据或堆数据。

示例：

```
foo()
{   int i, j;
    j = i;    /* Read from uninitialized memory (rui) */
}
```

写入到数组越界内存 (wob) 错误

问题：尝试写入到数组越界内存。

可能的原因：溢出堆块边界的迷失指针。

示例：

```
char *cp = malloc (10);
cp[10] = 'a';
```

写入到只读内存 (wro) 错误

问题： 尝试写入到只读内存。

可能的原因： 向文本地址写入、向只读数据区 (.rodata) 写入或向已由 mmap 设置为只读的页写入。

示例：

```
foo()
{   int *foop = (int *) foo;
    *foop = 0;               /* Write to read-only memory (wro) */
}
```

写入到未分配内存 (wua) 错误

问题： 尝试写入到不存在、未分配或未映射的内存。

可能的原因： 溢出堆块边界或访问已被释放的堆块的迷失指针。

示例：

```
char *cp = malloc (10);
free (cp);
cp[0] = 0;
```

内存泄漏错误

启用了泄漏检查时，运行时检查会报告下列类型的错误。

地址位于块内 (aib) 错误

问题： 可能的内存泄漏。没有对已分配块开始处的引用，但至少有一个对块内地址的引用。

可能的原因： 指向块开始处的唯一指针增加。

示例：

```
char *ptr;
main()
{
   ptr = (char *)malloc(4);
   ptr++;    /* Address in Block */
}
```

地址位于寄存器内 (air) 错误

问题： 可能的内存泄漏。尚未释放已分配块，程序内存中不存在对块的引用，但寄存器中存在引用。

可能的原因：如果编译器只将程序变量保留在寄存器中，而不保留在内存中，自然会出现这种情况。编译器常常会在启用了优化功能的情况下这样处理局部变量和函数参数。如果在未启用优化功能的情况下出现这种错误，则可能是真正的内存泄漏。如果指向已分配块的唯一指针在块被释放前超出作用域，便会出现这种情况。

示例：

```
if (i == 0) {
       char *ptr = (char *)malloc(4);
       /* ptr is going out of scope */
}
  /* Memory Leak or Address in Register */
```

内存泄漏 (mel) 错误

问题：尚未释放已分配块，程序中不存在对块的引用。

可能的原因： 程序未能释放不再使用的块。

示例：

```
char *ptr;

    ptr = (char *)malloc(1);
    ptr = 0;
/* Memory leak (mel) */
```

♦ ♦ ♦ 第 10 章

10 调试多线程应用程序

dbx 可以调试使用 Oracle Solaris 线程或 POSIX 线程的多线程应用程序。可以使用 dbx 检查每个线程的堆栈跟踪，恢复所有线程，step 或 next 特定线程及在线程间导航。

本章说明如何查找线程的相关信息及如何使用 dbx 线程命令调试线程。其中包含以下各节：

了解多线程调试

dbx 通过检测程序是否利用 libthread.so 来识别多线程程序。程序对 libthread.so 的使用方法有两种：一是通过 -lthread 或 -mt 对其进行显式编译；二是通过 -lpthread 对其进行隐式编译。

检测到多线程程序时，dbx 会尝试装入 libthread_db.so，它是一个专门用于进行线程调试的系统库，位于 /usr/lib 目录下。

dbx 是同步式的；所以当某个线程或轻量级进程 (lightweight process, LWP) 停止时，所有其他线程和 LWP 也会相应停止。我们有时将这种行为称作 "stop the world" 模型。

注 - 有关多线程编程和 LWP 的信息，请参见 Oracle Solaris《多线程编程指南》。

线程信息

dbx 中提供有以下线程信息示例。

```
(dbx) threads
    t@1 a l@1  ?()  running   in main()
    t@2      ?() asleep on 0xef751450  in_swtch()
    t@3 b l@2  ?()  running in sigwait()
```

```
    t@4     consumer()  asleep on 0x22bb0 in _lwp_sema_wait()
  *>t@5 b l@4 consumer()  breakpoint     in Queue_dequeue()
    t@6 b l@5 producer()     running       in _thread_start()
(dbx)
```

本机代码的每行信息由以下内容组成：

- *（星号）表示该线程内发生了需要用户处理的事件。该事件通常是断点。

  如果不是星号，而是 "o"，则表示发生的是 dbx 内部事件。
- >（箭头）表示为当前线程。
- 线程 id t@*number* 指特定线程。*number* 是 thr_create 传回的 thread_t 值。
- b l@*number* 或 a l@*number* 表示线程绑定到指定的 LWP 或在其上处于活动状态（即操作系统可实际运行该线程）。
- 传递给 thr_create 的线程的“启动函数”。?() 表示启动函数未知。
- 线程状态。
- 线程当前执行的函数。

Java 代码的每行信息由以下内容组成：

- t@*number*，dbx 样式的线程 ID
- 线程状态
- 加单引号的线程名
- 说明线程优先级的数字

线程和 LWP 状态

suspended（挂起）	线程已被显式挂起。
runnable（可运行）	线程可运行，并正等待作为计算资源的 LWP。
zombie（僵停）	分离的线程退出 (thr_exit()) 后，在通过使用 thr_join(). THR_DETACHED (它是在线程创建时，(thr_create()) 指定的标志) 重新链接之前，将一直处于僵停状态。退出的非分离线程在被获得前将一直处于僵停状态。
asleep on *syncobj*（在 syncobj 上休眠）	线程在给定同步对象上被阻塞。视 libthread 和 libthread_db 所提供的支持级别，*syncobj* 可能会如十六进制地址一般简单，也可能会包含更多的信息内容。
active（活动）	线程在某 LWP 中处于活动状态，但 dbx 无法访问该 LWP。
unknown（未知）	dbx 无法确定状态。

lwpstate	绑定或活动线程状态具有与之关联的 LWP 的状态。
running（正在运行）	LWP 正在运行，但因响应另一 LWP 而同步停止。
syscall *num*	LWP 进入给定系统调用 # 时停止。
syscall return *num*（syscall return num）	LWP 退出给定系统调用 # 时停止。
job control（作业控制）	LWP 因作业控制而停止。
LWP suspended（LWP 挂起）	LWP 在内核中被阻塞。
single stepped（单步递阶）	LWP 刚刚完成一步。
breakpoint（断点）	LWP 刚刚遇到断点。
fault *num*（故障 num）	LWP 招致给定错误 #。
signal *name*（信号 name）	LWP 招致给定信号。
process sync（进程同步）	此 LWP 所属进程刚刚开始执行。
LWP death（LWP 停止）	LWP 正在退出。

查看另一线程的上下文

要将查看上下文切换到另一线程，请使用 `thread` 命令。语法是：

`thread [-blocks] [-blockedby] [-info] [-hide] [-unhide] [-suspend] [-resume]` *thread_id*

要显示当前线程：

`thread`

要切换到线程 *thread-ID*：

thread *thread-ID*

有关更多信息，请参见“thread 命令” [352]。

查看线程列表

要查看线程列表，请使用 threads 命令。语法是：

```
threads [-all] [-mode [all|filter] [auto|manual]]
```

要输出所有已知线程列表：

```
threads
```

要输出通常不输出的线程（僵停）：

```
threads -all
```

有关线程列表的说明，请参见“线程信息” [151]。

有关 threads 命令的更多信息，请参见“threads 命令” [354]。

恢复执行

可以使用 cont 命令来恢复程序执行。因为线程目前使用同步断点，所以所有线程都会恢复执行。不过，您可以使用带有 -resumeone 选项的 call 命令恢复单个线程。

在调试多线程应用程序（其中许多线程会调用函数 lookup（））时，请考虑以下两种情况：

- 您设置条件断点：

  ```
  stop in lookup -if strcmp(name, "troublesome") == 0
  ```

  当 t@1 在对 lookup（） 的调用处停止时，dbx 尝试对条件求值并调用 strcmp（）。
- 您设置断点：

  ```
  stop in lookup
  ```

  当 t@1 在对 lookup（） 的调用处停止时，发出以下命令：

  ```
  call strcmp(name, "troublesome")
  ```

调用 strcmp（） 时，dbx 将会恢复调用期间内的所有线程，这与使用 next 命令进行单步执行时 dbx 所执行的操作类似。之所以这样做的原因是，如果 strcmp（） 尝试抓取另一个线程拥有的锁，则仅恢复 t@1 可能会导致死锁。

在这种情况下恢复所有线程的缺点是，dbx 无法处理另一个线程（例如 t@2），从而会在调用 strcmp（） 时在 lookup（） 处遇到断点。它会发出类似以下之一的警告：

```
event infinite loop causes missed events in following handlers:

Event reentrancy
first event BPT(VID 6, TID 6, PC echo+0x8)
second event BPT(VID 10, TID 10, PC echo+0x8)
the following handlers will miss events:
```

在这些情况下，如果可以确定条件表达式中调用的函数不会抓取互斥锁，则可使用 -resumeone 事件修饰符强制 dbx 只恢复 t@1：

```
stop in lookup -resumeone -if strcmp(name, "troublesome") == 0
```

只恢复在 lookup（） 中遇到断点的线程，以便对 strcmp（） 进行求值。

此方法在诸如以下示例中会无能为力：

- 由于条件以递归方式调用 lookup（），lookup（） 上的第二个断点发生在同一个线程中
- 运行条件的线程放弃控制权、休眠或以某种方式将控制权交给另一个线程

了解线程创建活动

通过使用 thr_create 事件和 thr_exit 事件，您可以了解您的应用程序多久创建并销毁一次线程，如以下示例所示：

```
(dbx) trace thr_create
(dbx) trace thr_exit
(dbx) run

trace: thread created t@2 on l@2
trace: thread created t@3 on l@3
trace: thread created t@4 on l@4
trace: thr_exit t@4
trace: thr_exit t@3
trace: thr_exit t@2
```

该应用程序创建了三个线程。请注意线程是如何以相反顺序从其创建中退出来的。这可能表明，如果此应用程序中的线程偏多，就会导致线程累积并消耗资源。

要获得更广泛的信息，可以在不同会话中尝试以下示例：

```
(dbx) when thr_create { echo "XXX thread $newthread created by $thread"; }
XXX thread t@2 created by t@1
XXX thread t@3 created by t@1
```

```
XXX thread t@4 created by t@1
```

此输出显示三个线程全部是由公用多线程模式的 t@1 线程创建的。

假设您要从开头调试线程 t@3。可以在线程 t@3 创建为如下所示时停止应用程序。

```
(dbx) stop thr_create t@3
(dbx) run
t@1 (l@1) stopped in tdb_event_create at 0xff38409c
0xff38409c: tdb_event_create       :    retl
Current function is main
216       stat = (int) thr_create(NULL, 0, consumer, q, tflags, &tid_cons2);
(dbx)
```

如果您的应用程序有时从线程 t@5 而不是从线程 t@1 产生新的线程，则可以按如下所示捕获此事件：

```
(dbx) stop thr_create -thread t@5
```

了解 LWP 信息

通常无需注意 LWP。但是，有时无法完成线程级查询。在这些情况下，使用 lwps 命令来显示有关 LWP 的信息。

```
(dbx) lwps
    l@1 running in main()
    l@2 running in sigwait()
    l@3 running in _lwp_sema_wait()
  *>l@4 breakpoint in Queue_dequeue()
    l@5 running in _thread_start()
(dbx)
```

LWP 列表的每行都包含下列内容：

- *（星号）表示此 LWP 内发生了需要用户处理的事件。
- >（箭头）表示当前 LWP。
- l@*number* 指特定 LWP。
- LWP 状态。
- LWP 当前执行的函数名称。

使用 lwp 命令可列出或更改当前的 LWP。

♦ ♦ ♦ 第 11 章

11 调试子进程

本章介绍如何调试子进程。dbx 提供了若干种工具，可帮助调试使用 fork (2) 和 exec (2) 函数创建子进程的进程。

本章包含以下各节：

连接到子进程

可通过以下方法之一连接到正在运行的子进程。

- 启动 dbx 时：

  ```
  $ dbx program-name process-ID
  ```

- 从 dbx 命令行：

  ```
  (dbx) debug program-name process-ID
  ```

如果包括 -（减号）而非程序名称，则 dbx 会自动查找与给定进程 ID 相关联的可执行程序。使用 - 符号后，后面的 run 命令或 rerun 命令将不起作用，因为 dbx 不知道可执行程序的全路径名。

也可以在 Oracle Developer StudioIDE 中连接到正在运行的子进程。有关更多信息，请参见 IDE 和 dbxtool 联机帮助。

跟随 exec 函数

如果一个子进程使用 exec(2) 函数或其变体之一执行新程序，进程 ID 不会更改，但进程映像将更改。dbx 会自动记录对 exec() 函数的调用，并隐式重新装入新执行的程序。

可执行程序的原始名称保存在 $oprog 中。要返回原始名称，请使用 debug $oprog 命令。

跟随 fork 函数

如果一个子进程调用 vfork(2)、fork1(2) 或 fork(2) 函数，则进程 ID 会更改，但进程映像保持不变。dbx 的行为取决于 dbxenv 变量 follow_fork_mode 的设置方式。

parent	在传统行为中，dbx 将忽略派生而跟随父进程。
child	dbx 自动切换到使用新进程 ID 的派生子进程。到原始父进程的所有连接以及对该进程的所有认知均丢失。
both	此模式仅当通过 Oracle Developer Studio IDE 或 dbxtool 使用 dbx 时才可用。
ask	只要 dbx 检测到派生，便会提示您选择 parent、child、both 或 stop to investigate。如果选择 stop，则可检查程序的状态，然后键入 cont 继续。系统会再次提示您选择继续的方式。只有 Oracle Developer Studio IDE 和 dbxtool 支持 both。

与事件交互

对于任意 exec() 或 fork() 进程，都将删除所有断点和其他事件。通过将 dbxenv 变量 follow_fork_inherit 设置为 on 覆盖对派生进程的删除，或使用 -perm eventspec 修饰符使事件成为持久性事件。有关使用事件规范修饰符的更多信息，请参见“cont at 命令” [239]。

♦ ♦ ♦ 第 12 章

12 调试 OpenMP 程序

OpenMP™ 应用编程接口 (application programming interface, API) 是 Sun 与多家计算机供应商为共享内存多处理器体系结构合作联合开发的可移植并行程序设计模型。对使用 dbx 调试 Fortran、C++ 和 C OpenMP 程序的支持基于 dbx 的通用多线程调试功能。

本章包含以下各节：

有关组成 OpenMP 4.0 版应用程序接口（通过 Oracle Developer Studio Fortran 和 C 编译器实现）的指令、运行时库例程和环境变量的信息，请参见《Oracle Developer Studio 12.5：OpenMP API 用户指南》。

编译器如何转换 OpenMP 代码

要更好地介绍 OpenMP 调试，有必要了解 OpenMP 代码是如何由编辑器进行转换的。参见下列 Fortran 示例：

```
1    program example
2        integer i, n
3        parameter (n = 1000000)
4        real sum, a(n)
5
6        do i = 1, n
7        a(i) = i*i
8        end do
9
10        sum = 0
11
12    !$OMP PARALLEL DO DEFAULT(PRIVATE), SHARED(a, sum)
13
14        do i = 1, n
15        sum = sum + a(i)
16        end do
17
18    !$OMP END PARALLEL DO
```

```
19
20        print*, sum
21        end program example
```

第 12 至 18 行的代码是一个并行区域。f95 编译器将这部分代码转换成从 OpenMP 运行时库中调用的外联子例程。此外联子例程有一个内部生成的名称，在本例中为 _ _ $d1A12.MAIN_。然后，f95 编译器用一个 OpenMP 运行时库调用替换并行区域的代码，并将外联子例程作为其中一个参数进行传递。该 OpenMP 运行时库将处理线程相关的所有问题，并分配并行执行外联子例程的从属线程。C 编译器的工作原理与此相同。

调试 OpenMP 程序时，dbx 将外联子例程与其他任何函数一样对待，唯一的不同是您不能使用其内部生成的名称在该函数中显式设置断点。

可用于 OpenMP 代码的 dbx 功能

除了调试多线程程序这一常规功能外，dbx 还可以调试 OpenMP 程序。对线程和 LWP 进行操作的所有 dbx 命令都可用于 OpenMP 调试。dbx 不支持 OpenMP 调试中的异步线程控制。

单步步入并行区域

dbx 可以单步步入并行区域。因为并行区域是外联区域并从 OpenMP 运行时库中进行调用，所以单步执行实际涉及几个层次的运行时库调用（这些调用由为此目的而创建的线程执行）。单步步入并行区域时，到达断点的第一个线程引起程序停止。此线程可能是从属线程而不是启动单步执行的主线程。

例如，请参阅“编译器如何转换 OpenMP 代码” [159] 中的 Fortran 代码，并假定主线程 t@1 位于第 10 行。当您单步执行到第 12 行时，将创建从属线程 t@2、t@3 和 t@4 来执行运行时库调用。线程 t@3 首先到达断点并导致程序停止执行。因此，由线程 t@1 启动的单步执行结束于线程 t@3。该行为不同于常规单步执行，常规单步执行后您通常处于与之前相同的线程中。

输出变量和表达式

dbx 可以输出所有共享、专用和线程专用变量。如果尝试输出并行区域之外的线程专用变量，将输出主线程的副本。whatis 命令用于输出并行构造内共享变量和专用变量的数据共享属性。对于线程专用变量，无论这些变量是否在并行构造内，都将输出其数据共享属性。例如：

```
(dbx) whatis p_a
# OpenMP first and last private variable
int p_a;
```

`print -s` 命令用于输出当前 OpenMP 并行区域中每个线程的 *expression* 表达式的值（如果该表达式包含专用变量或线程专用变量）。例如：

```
(dbx) print -s p_a
thread t@3: p_a = 3
thread t@4: p_a = 3
```

如果该表达式不包含任何专用变量或线程专用变量，将仅输出一个值。

输出区域和线程信息

使用 `omp_pr` 命令可输出当前并行区域或指定并行区域的描述，其中包括父区域、并行区域 ID、组大小（线程数）以及程序位置（程序计数器地址）。例如：

```
(dbx) omp_pr
parallel region 127283434369843201
    team size = 4
    source location = test.c:103
    parent = 127283430568755201
```

您还可以输出所有并行区域的描述以及从当前并行区域或指定并行区域到其根目录的路径。例如：

```
(dbx) omp_pr -ancestors
parallel region 127283434369843201
    team size = 4
    source location = test.c:103
    parent = 127283430568755201

    parallel region 127283430568755201
        team size = 4
        source location = test.c:95
        parent = <no parent>
```

它也可以输出整个并行区域树。例如：

```
(dbx) omp_pr -tree
parallel region 127283430568755201
    team size = 4
    source location = test.c:95
    parent = <no parent>

    parallel region 127283434369843201
        team size = 4
        source location = test.c:103
        parent = 127283430568755201
```

有关更多信息，请参见“omp_pr 命令” [325]。

使用 `omp_tr` 命令可输出当前任务区域或指定任务区域的描述，其中包括任务区域 ID、状态（spawned、executing、waiting）、正在执行的线程、程序位置（程序计数器地址）、未完成子项以及父项。例如：

```
(dbx) omp_tr
task region 65540
    type = implicit
    state = executing
    executing thread = t@4
    source location == test.c:46
    unfinished children = 0
    parent = <no parent>
```

您还可以输出所有任务区域的描述以及从当前任务区域或指定任务区域到其根目录的路径。

```
(dbx) omp_tr -ancestors
task region 196611
    type = implicit
    state = executing
    executing thread = t@3
    source location - test.c:103
    unfinished children = 0
    parent = 131075

    task region 131075
        type = implicit
        state = executing
        executing thread = t@3
        unfinished children = 0
        parent = <no parent>
```

您也可以输出整个任务区域树。例如：

```
(dbx) omp_tr -tree
task region 10
    type = implicit
    state = executing
    executing thread = t@10
    source location = test.c:103
    unfinished children = 0
    parent = <no parent>
task region 7
    type = implicit
    state = executing
    executing thread = t@7
    source location = test.c:103
    unfinished children = 0
    parent = <no parent>
task region 6
    type implicit
    state = executing
    executing thread = t@6
    source location = test.c:103
    unfinished children = 0
    parent = <o parent>
task region 196609
    type = implicit
```

```
    state = executing
    executing thread = t@1
    source location = test.c:95
    unfinished children = 0
    parent = <no parent>

    task region 262145
        type = implicit
        state = executing
        executing thread = t@1
        source location = test.c:103
        unfinished children - 0
        parent = 196609
```

有关更多信息，请参见“omp_tr 命令” [326]。

使用 omp_loop 命令可输出当前循环的描述，其中包括调度类型（静态、动态、指导、自动或运行时）、有序性、界限、步幅或跨距以及迭代数。例如：

```
(dbx) omp_loop
    ordered loop: no
    lower bound: 0
    upper bound: 3
    step: 1
    chunk: 1
    schedule type: static
    source location: test.c:49
```

有关更多信息，请参见“omp_loop 命令” [325]。

使用 omp_team 命令可输出当前组或指定并行区域组上的所有线程。例如：

```
(dbx) omp_team
team members:
    0: t@1 state = in implicit barrier, task region = 262145
    1: t@6 state = in implicit barrier, task region = 6
    2: t@7 state = working, task region = 7
    3: t@10 state = in implicit barrier, task region = 10
```

有关更多信息，请参见“omp_team 命令” [326]。

调试 OpenMP 代码时，除了关于当前或指定线程的有用信息之外，thread -info 还将输出 OpenMP 线程 ID、并行区域 ID、任务区域 ID 及 OpenMP 线程状态。有关更多信息，请参见“thread 命令” [352]。

将并行区域的执行序列化

使用 omp_serialize 命令可将当前线程或当前组中所有线程遇到的下一个并行区域的执行进行序列化。有关更多信息，请参见“omp_serialize 命令” [325]。

使用堆栈跟踪

当执行在并行区域中停止时，where 命令会显示包含外联子例程的堆栈跟踪。

```
(dbx) where
current thread: t@4
=>[1] _$d1E48.main(), line 52 in "test.c"
  [2] _$p1I46.main(), line 48 in "test.c"

--- frames from parent thread ---
current thread: t@1
  [7] main(argc = 1, argv = 0xffffffff7fffec98), line 46 in "test.c"
```

堆栈的顶帧是外联函数帧。尽管代码是外联的，源代码行号仍映射回 15。

当执行在并行区域中停止时，如果相关帧仍处于活动状态，来自从属线程的 where 命令将输出主线程的堆栈跟踪。来自主线程的 where 命令具有完全的回溯。

也可以首先使用 omp_team 命令列出当前组中的所有线程，然后切换到主线程（OpenMP 线程 ID 为 0 的线程）并从该线程获得堆栈跟踪，以此来确定执行到达从属线程中断点的方式。

使用 dump 命令

当执行在并行区域中停止时，dump 命令可以输出专用变量的多个副本。在下例中，dump 命令输出变量 i 的两个副本：

```
[t@1 l@1]: dump
i = 1
sum = 0.0
a = ARRAY
i = 1000001
```

因为外联例程作为宿主例程的嵌套函数实现，而专用变量作为外联例程的局部变量实现，所以会输出变量 i 的两个副本。由于 dump 命令输出作用域内的所有变量，因此宿主例程中的 i 和外联例程中的 i 均会显示。

使用事件

dbx 提供了可以与 stop、when 和 trace 命令结合用于 OpenMP 代码的事件。有关将事件与这些命令结合使用的信息，请参见“设置事件规范” [244]。

同步事件

`omp_barrier` [*type*] [*state*]

跟踪线程进入屏障事件。

type 有效值为：

- `explicit` – 跟踪显式屏障
- `implicit` – 跟踪隐式屏障

如果不指定 *type*，则只跟踪显式屏障。

state 有效值为：

- `enter` – 在有线程进入屏障时报告该事件
- `exit` – 在有线程退出屏障时报告该事件
- `all_entered` – 在所有线程都进入屏障时报告该事件

如果不指定 *state*，缺省值为 `all_entered`。

如果指定 `enter` 或 `exit`，可以设置线程 ID 以指定仅跟踪该线程。

`omp_taskwait` [*state*]

跟踪线程进入任务等待事件。

state 有效值为：

- `enter` – 当线程进入任务等待时报告该事件
- `exit` – 在所有子任务都已完成时报告该事件

如果不指定 *state*，缺省值为 `exit`。

`omp_ordered` [*state*]

跟踪线程进入有序区域事件。

state 有效值为：

- `begin` – 在有序区域开始时报告该事件
- `enter` – 在线程进入有序区域时报告该事件
- `exit` – 在线程退出有序区域时报告该事件

如果不指定 *state*，则缺省值为 `enter`。

`omp_critical`

跟踪线程进入重要区域事件。

`omp_atomic` [*state*]

跟踪线程进入 atomic 区域事件。

state 有效值为：

- `begin` – 在原子区域开始时报告该事件
- `exit` – 在线程退出原子区域时报告该事件

如果不指定 *state*，缺省值为 `begin`。

`omp_flush`

跟踪线程执行刷新事件。

其他事件

`omp_task [`*state*`]` 跟踪任务的创建和终止。

state 有效值为：

- `create` – 在任务刚刚创建完毕且尚未开始执行时报告该事件
- `start` – 在任务开始执行时报告该事件
- `finish` – 在任务执行完毕即将终止时报告该事件

如果不指定 *state*，缺省值为 `start`。

`omp_master` 跟踪主线程进入主区域事件。

`omp_single` 跟踪线程进入单个区域事件。

OpenMP 代码的执行序列

当在 OpenMP 程序中的并行区域内单步执行时，执行序列与源代码序列可能会不同。产生这种序列差异的原因在于并行区域中的代码通常会由编译器进行转换和重新排列。OpenMP 代码中的单步执行与优化代码中的单步执行类似，其中优化器经常会移动代码。

♦ ♦ ♦ 第 13 章

13 处理信号

本章介绍如何使用 dbx 处理信号。

本章包含以下各节。

了解信号事件

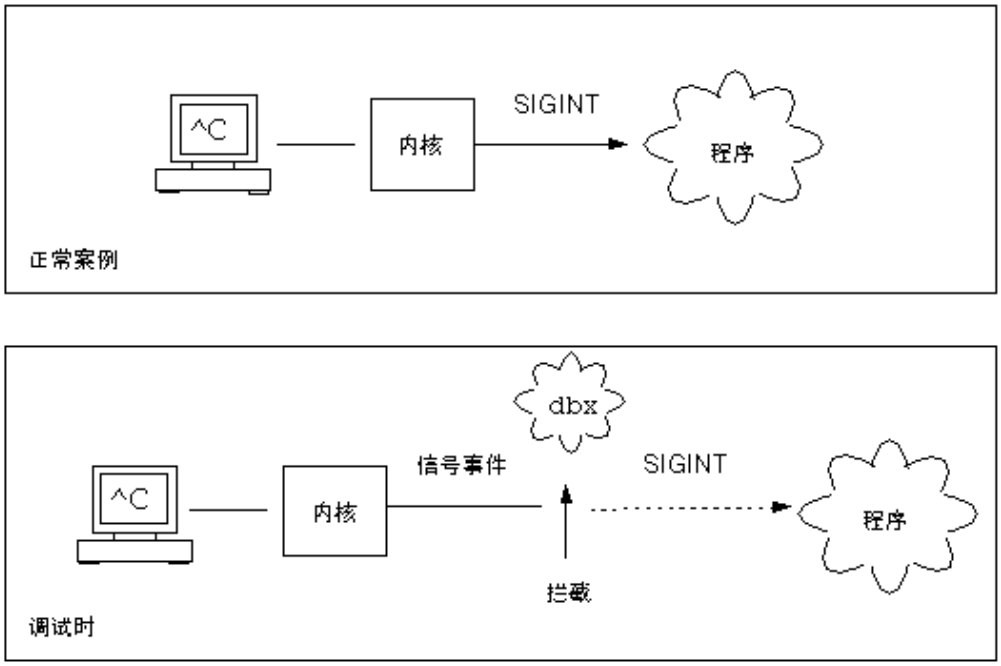
在要将信号传送给正在调试的进程时，该信号会由内核重定向到 dbx。这时，通常会收到一个提示。而您有两种选择：

- 在程序恢复后取消信号（这是 cont 命令的缺省行为），以方便使用 SIGINT (Ctrl-C) 中断和恢复程序，如图 2 中所示。
- 使用以下命令将信号转发给进程：

  ```
  cont -sig signal
  ```

  signal 可以是信号名或信号编号。

图 2 拦截和取消 SIGINT 信号



此外，如果频繁收到某一信号，但又不想显示它，可以安排 dbx 自动转发该信号：

`ignore` *signal*

但是，信号仍会转发给进程。一组缺省信号就是以这种方式自动转发的（请参见“ignore 命令” [307]）。

捕获信号

dbx 支持 catch 命令，该命令指示 dbx 在检测到捕获列表中出现的任何信号时停止程序。

缺省情况下，捕获列表包含 33 个以上可检测信号中的许多信号。（信号数目取决于操作系统和版本。）可通过从缺省捕获列表中添加或删除信号来更改缺省捕获列表。

注 - dbx 接受的信号名称列表中包括 dbx 支持的各种版本 Oracle Solaris 操作环境所支持的所有信号。因此，dbx 可能接受当前正在运行的 Oracle Solaris 操作环境版本所不支持的信号。例如，dbx 可能接受 Oracle Solaris 9 OS 所支持的信号，即使当前运行的是 Oracle Solaris 7 OS。有关当前正在运行的 Oracle Solaris 操作系统所支持的信号列表，请参见 signal(3head) 手册页。

要查看当前正在捕获的信号的列表，请键入不带 *signal* 参数的 **catch**。

```
(dbx) catch
```

要查看当前即使被程序检测到也会被 dbx 忽略的信号列表，请键入不带 *signal* 参数的 **ignore**。

```
(dbx) ignore
```

更改缺省信号列表

通过将信号名从一个列表移到另一个列表，可以控制由哪些信号导致程序停止。要移动信号名，将当前出现于某个列表的信号名作为参数提供给另一个列表即可。

例如，要将捕获列表中的 QUIT 和 ABRT 信号移到忽略列表中：

```
(dbx) ignore QUIT ABRT
```

捕获 FPE 信号（仅限 Oracle Solaris）

浮点和整数算术运算可能会导致异常，例如溢出和被零除。此类异常通常无提示，因此系统会返回合理答案（例如 NaN）作为导致异常的运算的结果。因此，这些异常对 dbx 不可见。

您可以进行设置使得出现异常时不再无提示，而是导致陷阱。然后，操作系统将陷阱转换为 SIGFPE 并将其传送到进程，dbx 可以拦截此信号传送。请注意以下几点：

- 缺省情况下，F77 不会在出现任何浮点异常时自陷。
- 缺省情况下，F95 会在出现无效操作数、除以零和溢出异常时自陷，但不会在出现下溢和不精确异常时自陷。
- 缺省情况下，C 和 C++ 不会在出现浮点异常时自陷。
- 没有针对整数溢出进行置备以隐式触发 SIGFPE。在 SPARC 上，您可以使用 TVS (trap-on-overflow-set) 汇编指令。在 SPARC 或 Intel 上，您可以使用类似 branch-on-overflow-set 的指令。

要查找导致异常的原因，需要在程序内设置陷阱处理程序，以使异常触发 SIGFPE 信号。

可使用下列方式启用陷阱：

- `fpsetmask` – 此函数严格控制陷阱的启用。请参见 `fpsetmask`(3C) 手册页。

  示例：

  ```
  #include <ieeefp.h>
      int main() {
      fpsetmask(FP_X_INV|FP_X_OFL|FP_X_UFL|FP_X_DZ|FP_X_IMP);
      ...
  ```

- `ieee_handler` – 不存在 Fortran 的 `psetmask(3c)` 精确模拟。相反，您可通过按如下所有建立缺省行为，以启用陷阱。

  示例：

  ```
      integer*4 ieeer
      ieeeer  = ieee_handler('set', 'common', SIGFPE_DEFAULT)
  ```

  有关更多信息，请参见 `ieee_environment(3f)` 和 `ieee_handler(3m)` 手册页。

- `-ftrap` 编译器标志 – 此标记类似于 `fpsetmask`（）()，它将严格控制陷阱的启用。有关 Fortran 95 的信息，请参见 `f95`(1) 手册页。

使用以上方法之一启用浮点陷阱时，系统将设置硬件浮点状态寄存器中的陷阱启用掩码。此陷阱启用掩码会导致异常在运行时引发 `SIGFPE` 信号。

插入对 `fpsetmask`（）() 或 `ieee_handler`（）() 的调用，或使用陷阱处理程序编译程序后，将程序装入到 `dbx` 中。从 Oracle Developer Studio 12.5 起，`SIGFPE` 在缺省情况下已捕获。使用旧版本的 `dbx` 时，请确保该信号仍在捕获列表中。

```
(dbx) catch FPE
```

您可以通过对 `fpsetmask`（） 和 `ieee_handler`（） 的参数稍作调整，并通过使用 dbx 捕获命令的替代命令，就如捕获 FPE 一样，进一步量身定制查看哪些特定异常。

```
(dbx) stop sig FPE
(dbx) ignore SIGFPE #don't catch it twice
```

可以使用以下代码优化控制：

```
stop sig FPE subcode
```

其中 *subcode* 可以是下列选项之一：

FPE_INTDIV	整数除以零。
FPE_INTOVF	整数溢出。
FPE_FLTDIV	浮点除以零。
FPE_FLTOVF	浮点溢出。

FPE_FLTUND	浮点下溢。
FPE_FLTRES	浮点不精确结果。
FPE_FLTINV	无效的浮点运算。
FPE_FLTSUB	下标超出范围。

确定发生异常的位置

将 `FPE` 添加到捕获列表后，在 `dbx` 中运行程序。出现要捕获的异常时，系统将引发 `SIGFPE` 信号且 `dbx` 会停止程序。然后，可以使用 `dbx where` 命令跟踪调用堆栈，以帮助查找程序中发生该异常的特定行号。

确定导致异常的原因

要确定导致 SPARC 异常的原因，请使用 `regs -f` 命令显示浮点状态寄存器 (floating point state register, FSR)。查看寄存器的已产生的异常 (`aexc`) 和当前异常 (`cexc`) 字段，这些字段包含下列浮点异常条件位：

- 无效操作数
- 溢出
- 下溢
- 除以零
- 不精确结果

在 Intel 中，x87 的浮点状态寄存器是 `fstat`，SSE 的浮点状态寄存器是 `mxcsr`。

有关浮点状态寄存器的更多信息，请参见《*The SPARC Architecture Manual*》的版本 8 (V8) 或版本 9 (V9)。有关更多讨论和示例，请参见《Oracle Developer Studio 12.5：数值计算指南》。

向程序发送信号

`dbx cont` 命令支持 `-sig` 选项，使用该选项可以恢复程序的执行，并使程序的行为像收到了系统信号 *signal* 一样。

例如，如果程序具有相应于 `SIGINT` (`^C`) 的中断处理程序，则可以键入 `^C` 来停止应用程序，并将控制返回到 `dbx`。如果您发出 `cont` 命令本身来继续执行程序，则永远不会执行该中断处理程序。要执行该中断处理程序，请将信号 `SIGINT` 发送到程序：

```
(dbx) cont -sig int
```

step 命令、next 命令和 detach 命令也接受 -sig 选项。

自动处理信号

事件管理命令也可以把信号作为事件进行处理。这两种命令具有相同的作用。

```
(dbx) stop sig signal
(dbx) catch signal
```

如果需要将一些预编程操作关联起来，那么信号事件将是非常有用的。

```
(dbx) when sig SIGCLD {echo Got $sig $signame;}
```

在这种情况下，请确保首先将 SIGCLD 移至忽略列表。

```
(dbx) ignore SIGCLD
```

♦ ♦ ♦ 第 14 章

使用 dbx 调试 C++

本章说明了 dbx 如何处理 C++ 异常及调试 C++ 模板，其中概括介绍了完成这些任务所使用的命令，并提供了代码样例示例。对于本章中说明的异常情况，您通常可使用 dbx 调试 C++。

本章包含以下各节：

有关编译 C++ 程序的信息，请参见“编译调试程序” [41]。

使用 dbx 调试 C++

虽然本章只重点介绍了调试 C++ 的两个特定方面，但您在调试 C++ 程序时可使用 dbx 的全部功能。您仍可使用 C++ 程序执行以下任务：

注 - 以下所有任务均已在前面的章节中做了探讨。

查找类和类型的定义	请参见“查找类型和类的定义” [71]
输出或显示继承的数据成员	请参见“输出 C++ 指针” [106]
查找有关对象指针的动态信息	请参见“输出 C++ 指针” [106]
调试虚拟函数	请参见“调用函数” [83]
使用运行时类型信息	请参见“输出变量、表达式或标识符的值” [106]
在某个类的所有成员函数上设置断点	请参见“在类的所有成员函数中设置断点” [90]
在所有重载的成员函数中设置断点	请参见“在不同类的成员函数中设置断点” [90]
在所有重载的非成员函数中设置断点	请参见“在非成员函数中设置多个断点” [90]

在特定对象的所有成员函数中设置断点	请参见“在对象中设置断点” [91]
处理重载函数或数据成员	请参见“在函数中设置断点” [88]

本章其余部分将重点介绍调试 C++ 的两个特定方面。

dbx 中的异常处理

如果出现异常，程序便停止运行。异常会发送编程异常信号，例如：被零除或数组上溢。可以设置块来捕获代码中其他位置的表达式引起的异常。

调试程序时，您可以使用 dbx 完成以下操作：

- 在堆栈解开之前捕获未处理的异常
- 捕获意外的异常
- 在堆栈解开之前捕获已处理或未处理的特定异常
- 确定程序中特定点处出现的异常的捕获位置

如果在程序停止于异常抛出点之后发出 step 命令，则将在堆栈解开期间开始执行第一个析构函数时返回控制。如果通过 step 命令步出堆栈解开期间执行的析构函数，则将在下一个析构函数的起始处返回控制。当所有析构函数都执行完毕后，step 命令将前进到处理异常抛出的 catch 块。

异常处理命令

本节介绍了处理异常的 dbx 命令。

exception 命令

exception 命令的语法如下所示：

```
exception [-d | +d]
```

在调试期间，可以随时使用 exception 命令显示异常的类型。如果使用 exception 命令时不带选项，则所示类型由 dbxenv 变量 output_dynamic_type: 的设置确定：

- 如果设置为 on，则显示派生类型。
- 如果设置为 off（缺省值），则显示静态类型。

如果指定 -d 或 +d 选项，则会覆盖环境变量的设置。

- 如果指定 -d，则显示派生类型。
- 如果指定 +d，则显示静态类型。

有关更多信息，请参见“exception 命令” [300]。

intercept 命令

intercept 命令的语法如下所示：

```
intercept [-all] [-x] [-set] [typename]
```

可以在堆栈解开之前拦截或捕获特定类型的异常。

- 使用不带参数的 intercept 命令可列出正在拦截的类型。
- 使用 -all 可拦截所有异常。使用 *typename* 可将类型添加到拦截列表中。
- 使用 -x 可从排除列表中排除特定类型，从而不拦截该类型。
- 使用 -set 可清除拦截列表和排除列表，并将列表设置为仅拦截或排除指定类型的抛出。

例如，要拦截除 int 之外的所有类型：

```
(dbx) intercept -all -x int
```

要拦截类型为 Error 的异常：

```
(dbx) intercept Error
```

如果使用以下命令拦截的 CommonError 异常过多：

```
(dbx) intercept -x CommonError
```

如果键入不带参数的 intercept 命令，则会发现拦截列表中包括未处理的异常和意外的异常（缺省情况下将拦截这些异常），以及 Error 类的异常（CommonError 类的异常除外）。

```
(dbx) intercept
-unhandled   -unexpected   class Error -x class CommonError
```

如果您随后意识到 Error 不是您需要的异常类，但是不知道要查找的异常类的名称，则可以通过键入以下内容来尝试拦截除 Error 类异常之外的所有异常：

```
(dbx) intercept -all -x Error
```

有关更多信息，请参见“intercept 命令” [308]。

unintercept 命令

unintercept 命令的语法如下所示：

```
unintercept [-all] [-x] [typename]
```

- 使用 unintercept 命令可从拦截列表或排除列表中删除异常类型。
- 使用不带参数的命令可列出正在拦截的类型（与 intercept 命令相同）。
- 使用 -all 可从拦截列表中删除所有类型。使用 *typename* 可从拦截列表中删除一个类型。使用 -x 可从排除列表中删除一个类型。

有关更多信息，请参见“unintercept 命令” [176]。

whocatches 命令

如果在当前执行点抛出异常，whocatches 命令会报告捕获类型为 *typename* 的异常的位置。使用此命令可查出从堆栈顶帧抛出异常时会发生什么情况。

捕获 *typename* 的 catch 子句的行号、函数名和帧号均会显示出来。如果捕获点所在函数与抛出异常的函数相同，则该命令返回 "*type* is unhandled"。

有关更多信息，请参见“whocatches 命令” [176]。

异常处理示例

此示例通过一个包含异常的样例程序说明 dbx 中的异常处理。在函数 bar 中抛出类型为 int 的异常，并在后面的 catch 块中捕获该异常。

```
1  #include <stdio.h>
2
3  class c {
4      int x;
5    public:
6      c(int i) { x = i; }
7      ~c() {
8                printf("destructor for c(%d)\n", x);
9           }
10 };
11
12 void bar() {
13     c c1(3);
14     throw(99);
15 }
16
```

```
17  int main() {
18      try {
19          c c2(5);
20          bar();
21          return 0;
22      }
23      catch (int i) {
24          printf("caught exception %d\n", i);
25      }
26  }
```

下面摘录的程序示例显示了 dbx 中的异常处理功能。

```
(dbx) intercept
-unhandled -unexpected
(dbx) intercept int
<dbx> intercept
-unhandled -unexpected int
(dbx) stop in bar
(2) stop in bar()
(dbx)run
Running: a.out
(process id 304)
Stopped in bar at line 13 in file “foo.cc”
   13       c c1(3);
(dbx) whocatches int
int is caught at line 24, in function main (frame number 2)
(dbx) whocatches c
dbx: no runtime type info for class c (never thrown or caught)
(dbx) cont
Exception of type int is caught at line 24, in function main (frame number 4)
stopped in _exdbg_notify_of_throw at 0xef731494
0xef731494: _exdbg_notify_of_throw          :        jmp     %o7 + 0x8
Current function is bar
   14        throw(99);
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8         printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(3)
stopped in c::~c at line 9 in file "foo.cc"
    9       }
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8        printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(5)
stopped in c::~c at line 9 in file "foo.cc"
    9       )
(dbx) step
stopped in main at line 24 in file "foo.cc"
   24           printf("caught exception %d\n", i);
(dbx) step
caught exception 99
```

```
stopped in main at line 26 in file "foo.cc"
   26   }
```

注 - 本节中使用的示例是使用 Oracle Developer Studio 编译器构建的。如果使用 gcc 编译代码，则示例可能有所不同。

使用 C++ 模板调试

dbx 支持 C++ 模板。可将包含类和函数模板的程序装入 dbx 中，并对要用于类或函数的模板调用任何 dbx 命令：

在类或函数模板实例中设置断点	请参见“stop inclass 命令” [181]、“stop inclass 命令” [181]和“stop in 命令” [182]。
输出所有类和函数模板实例列表	请参见“whereis 命令” [180]
显示模板和实例的定义	请参见“whatis 命令” [180]
调用成员模板函数和函数模板实例	请参见“call 命令” [182]
输出函数模板实例的值	请参见“print 表达式” [182]
显示函数模板实例的源代码	请参见“list 表达式” [183]

模板示例

以下代码示例中显示了类模板 Array 及其实例以及函数模板 square 及其实例。

```
 1        template<class C> void square(C num, C *result)
 2        {
 3             *result = num * num;
 4        }
 5
 6        template<class T> class Array
 7        {
 8        public:
 9              int getlength(void)
10             {
11                    return length;
12              }
13
14              T & operator[](int i)
15              {
16                    return array[i];
17             }
18
19              Array(int l)
```

```
20              {
21                    length = l;
22                    array = new T[length];
23              }
24
25              ~Array(void)
26              {
27                    delete [] array;
28              }
29
30        private:
31              int length;
32              T *array;
33        };
34
35        int main(void)
36        {
37              int i, j = 3;
38             square(j, &i);
39
40             double d, e = 4.1;
41              square(e, &d);
42
43              Array<int> iarray(5);
44              for (i = 0; i < iarray.getlength(); ++i)
45              {
46                   iarray[i] = i;
47             }
48
49              Array<double> darray(5);
50              for (i = 0; i < darray.getlength(); ++i)
51             {
52                    darray[i] = i * 2.1;
53              }
54
55              return 0;
56        }
```

该示例中：

- `Array` 是一个类模板
- `square` 是一个函数模板
- `Array<int>` 是类模板实例（模板类）
- `Array<int>::getlength` 是模板类的成员函数
- `square(int, int*)` 和 `square(double, double*)` 是函数模板实例（模板函数）

C++ 模板的命令

在模板和模板实例中使用这些命令。知道类或类型定义后，便可以输出值、显示源代码列表或设置断点。

whereis 命令

使用 whereis 命令可输出函数模板或类模板的函数或类实例的所有具体值列表。

对于类模板：

```
(dbx) whereis Array
member function: `Array<int>::Array(int)
member function: `Array<double>::Array(int)
class template instance: `Array<int>
class template instance: `Array<double>
class template: `a.out`template_doc_2.cc`Array
```

对于函数模板：

```
(dbx) whereis square
function template instance: `square<int>(__type_0,__type_0*)
function template instance: `square<double>(__type_0,__type_0*)
function template: `a.out`template_doc_2.cc`square
```

__type_0 参数引用 0 号模板参数。__type_1 引用下一个模板参数。

有关更多信息，请参见“whereis 命令” [180]。

whatis 命令

使用 whatis 命令可输出函数模板和类模板的定义以及实例化的函数和类。

对于类模板：

```
(dbx) whatis -t Array
template<class T> class Array
To get the full template declaration, try `whatis -t Array<int>';
```

对于类模板的构造函数：

```
(dbx) whatis Array
More than one identifier 'Array'.
Select one of the following:
 0) Cancel
 1) Array<int>::Array(int)
 2) Array<double>::Array(int)
> 1
Array<int>::Array(int 1);
```

对于函数模板：

```
(dbx) whatis square
More than one identifier 'square'.
Select one of the following:
 0) Cancel
 1) square<int(__type_0,__type_0*)
 2) square<double>(__type_0,__type_0*)
> 2
void square<double>(double num, double *result);
```

对于类模板实例：

```
(dbx) whatis -t Array<double>
class Array<double> {
public:
    int Array<double>::getlength()
    double &Array<double>::operator [](int i);
    Array<double>::Array<double>(int l);
    Array<double>::~Array();
private:
    int length;
    double *array;
};
```

对于函数模板实例：

```
(dbx) whatis square(int, int*)
void square(int num, int *result);
```

有关更多信息，请参见“whatis 命令” [180]。

stop inclass 命令

要在模板类的所有成员函数中停止：

```
(dbx)stop inclass Array
(2) stop inclass Array
```

使用 stop inclass 命令可在特定模板类的所有成员函数中设置断点：

```
(dbx) stop inclass Array<int>
(2) stop inclass Array<int>
```

有关更多信息，请参见“stop in 命令” [182]和“inclass 事件规范” [246]。

stop infunction 命令

使用 stop infunction 命令可在特定函数模板的所有实例中设置断点：

```
(dbx) stop infunction square
```

```
(9) stop infunction square
```

有关更多信息，请参见“stop infunction 命令” [181]和“infunction 事件规范” [245]。

stop in 命令

使用 stop in 命令可在模板类的成员函数或在模板函数中设置断点。

对于类模板实例的成员：

```
(dbx) stop in Array<int>::Array(int l)
(2) stop in Array<int>::Array(int)
```

对于函数实例：

```
(dbx) stop in square(double, double*)
(6) stop in square<double>(__type_0,__type_0*)
```

有关更多信息，请参见“stop in 命令” [182]和“in 事件规范” [244]。

call 命令

使用 call 命令可在作用域中停止时显式调用函数实例或类模板的成员函数。如果 dbx 无法确定正确的实例，它会显示一个带编号的实例列表，您可以从列表中进行选择。

```
(dbx) call square(j,&i)
```

有关更多信息，请参见“call 命令” [274]。

print 表达式

使用 print 命令可对函数实例或类模板的成员函数求值。

```
(dbx) print iarray.getlength()
iarray.getlength() = 5
```

使用 print 对 this 指针求值。

```
(dbx) whatis this
class Array<int> *this;
(dbx) print *this
*this = {
    length = 5
    array   = 0x21608
}
```

有关更多信息，请参见“print 命令” [329]。

list 表达式

使用 list 命令可输出指定函数实例的源代码列表。

```
(dbx) list square(int, int*)
```

有关更多信息，请参见“list 命令” [312]。

♦ ♦ ♦ 第 15 章

15 使用 dbx 调试 Fortran

本章介绍可能会用于 Fortran 的 dbx 功能。另外还提供了一些 dbx 请求样例，以便为您在使用 dbx 调试 Fortran 代码时提供帮助。

本章包括以下主题：

调试 Fortran

以下是一些有助于您调试 Fortran 程序的提示和一般性概念。有关使用 dbx 调试 Fortran OpenMP 代码的信息，请参见“与事件交互” [158]。

当前过程和文件

调试会话期间，dbx 会将过程和源文件定义为当前过程和源文件。设置断点及输出或设置变量的请求都相对于当前函数和文件来解释。因此，stop at 5 会根据哪一个文件是当前文件来设置不同的断点。

大写字母

程序的标识符中有大写字母时，dbx 会识别出它们。无需提供区分大小写或不区分大小写的命令，而某些早期版本中需要提供。

Fortran 和 dbx 必须都处于区分大小写模式或不区分大小写模式下：

- 在不区分大小写模式下，编译和调试时无需使用 -U 选项。这时，dbx 环境变量 input_case_sensitive 的缺省值为 false。

  如果源代码中有名为 LAST 的变量，则在 dbx 中，print LAST 或 print last 命令都有效。Fortran 和 dbx 按照要求将 LAST 和 last 视为相同。
- 在区分大小写模式下，编译和调试时使用 -U。这时，dbx 环境变量 input_case_sensitive 的缺省值为 true。

  如果源代码中有一个名为 LAST 和一个名为 last 的变量，则在 dbx 中，print last 有效，而 print LAST 无效。Fortran 和 dbx 按照要求区分 LAST 和 last。

注 - 在 dbx 中，文件或目录名始终区分大小写，即使将 dbx input_case_sensitive 环境变量设置为 false 时也是如此。

样例 dbx 会话

下面的示例中使用称为 my_program 的样例程序。

用于调试的主程序 a1.f：

```
    PARAMETER ( n=2 )
    REAL twobytwo(2,2) / 4 *-1 /
    CALL mkidentity( twobytwo, n )
    PRINT *, determinant( twobytwo )
    END
```

用于调试的子例程 a2.f：

```
    SUBROUTINE mkidentity ( array, m )
    REAL array(m,m)
    DO 90 i = 1, m
        DO 20 j = 1, m
            IF ( i .EQ. j ) THEN
            array(i,j) = 1.
            ELSE
            array(i,j) = 0.
            END IF
20        CONTINUE
90    CONTINUE
    RETURN
```

```
END
```

用于调试的函数 a3.f：

```
REAL FUNCTION determinant ( a )
REAL a(2,2)
determinant = a(1,1) * a(2,2) - a(1,2) * a(2,1)
RETURN
END
```

▼ 如何运行样例 dbx 会话

1. **使用 -g 选项进行编译和链接。**
   可以通过一两个步骤来完成此操作。

   - **要使用一个步骤完成编译和链接：**

     ```
     demo% f95 -o my_program -g a1.f a2.f a3.f
     ```

   - **要分步完成编译和链接：**

     ```
     demo% f95 -c -g a1.f a2.f a3.f
     demo% f95 -o my_program a1.o a2.o a3.o
     ```

2. **对名为 my_program 的可执行文件启动 dbx。**

   ```
   demo% dbx my_program
   Reading symbolic information…
   ```

3. **设置简单的断点。**
   在主程序的第一个可执行语句处停止。

   ```
   (dbx) stop in MAIN
   (2) stop in MAIN
   ```

   尽管主程序 MAIN 必须全部为大写，但子例程、函数或块数据子程序的名称可以为大写或小写。

4. **在启动 dbx 时指定的可执行文件中运行程序。**

   ```
   (dbx) run
   Running: my_program
   stopped in MAIN at line 3 in file "a1.f"
       3         call mkidentity( twobytwo, n )
   ```

   到达断点时，dbx 会显示一条消息，指出其停止位置（本例中为文件 a1.f 的第 3 行）。

5. **输出值。**
   输出 n 的值：

```
(dbx) print n
n = 2
```

要输出矩阵 twobytwo，格式可能会有所不同：

```
(dbx) print twobytwo
twobytwo =
   (1,1)       -1.0
   (2,1)       -1.0
   (1,2)       -1.0
   (2,2)       -1.0
```

请注意，不能输出矩阵数组的原因是此处未定义 array，而只是在 mkidentity 中进行了定义。

6. 继续执行到下一行。

```
(dbx) next
stopped in MAIN at line 4 in file "a1.f"
    4             print *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo =
    (1,1)       1.0
    (2,1)       0.0
    (1,2)       0.0
    (2,2)       1.0
(dbx) quit
demo%
```

next 命令会执行当前源代码行并在下一行处停止。它将各次子程序调用按独立的语句来计数。

7. 退出 **dbx**。

```
(dbx)quit
demo%
```

调试段故障

如果程序遇到段故障 (SIGSEGV)，便会引用其可用内存外的内存地址。

导致段故障的最常见原因：

- 数组索引超出声明的范围。
- 数组索引的名称拼写错误。
- 调用例程的一个参数是 REAL，而被调用例程对应的参数却是 INTEGER。
- 数组索引计算错误。
- 调用例程的参数数量不足。

- 使用未定义的指针。

使用 dbx 找出问题

可使用 dbx 找到出现段故障的源代码行。

使用程序来生成段故障。

```
demo% cat WhereSEGV.f
    INTEGER a(5)
    j = 2000000
    DO 9 i = 1,5
        a(j) = (i * 10)
9    CONTINUE
    PRINT *, a
    END
demo%
```

可使用 dbx 找到 dbx 段故障的行号。

```
demo% f95 -g -silent WhereSEGV.f
demo% a.out
Segmentation fault
demo% dbx a.out
Reading symbolic information for a.out
program terminated by signal SEGV (segmentation violation)
(dbx) run
Running: a.out
signal SEGV (no mapping at the fault address)
    in MAIN at line 4 in file "WhereSEGV.f"
    4                   a(j) = (i * 10)
(dbx)
```

定位异常

程序抛出异常的原因有很多。一种定位故障的方法是在源程序中找到发生异常的行号，然后在该处进行检查。

编译时使用 -ftrap=common 可强制捕获所有常见异常。

查找异常发生位置：

```
demo% cat wh.f
                call joe(r, s)
                print *, r/s
                end
                subroutine joe(r,s)
```

```
                r = 12.
                s = 0.
                return
                end
demo% f95 -g -o wh -ftrap=common wh.f
demo% dbx wh
Reading symbolic information for wh
(dbx) catch FPE
(dbx) run
Running: wh
(process id 17970)
signal FPE (floating point divide by zero) in MAIN at line 2 in file “wh.f”
   2                    print *, r/s
(dbx)
```

跟踪调用

有时程序会因信息转储而停止，此时便需要知道将程序引至该处的调用的序列。此序列称为堆栈跟踪。

where 命令显示程序流中执行停止位置以及执行到达此点的过程，即被调用例程的堆栈跟踪。

ShowTrace.f 是编写专门用于使信息转储深入调用序列中的若干层来显示堆栈跟踪的程序。

Note the reverse order:

```
demo% f77 -silent -g ShowTrace.f
demo% a.out
MAIN called calc, calc called calcb.
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quil 174% dbx a.out
Execution stopped, line 23
Reading symbolic information for a.out
...
(dbx) run
calcB called from calc, line 9
Running: a.out
(process id 1089)
calc called from MAIN, line 3
signal SEGV (no mapping at the fault address) in calcb at line 23 in file "ShowTrace.f"
   23                   v(j) = (i * 10)
(dbx) where -V
=>[1] calcb(v = ARRAY , m = 2), line 23 in "ShowTrace.f"
  [2] calc(a = ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f"
  [3] MAIN(), line 3 in "ShowTrace.f"
(dbx)
Show the sequence of calls, starting at where the execution stopped:
```

处理数组

dbx 可识别数组并输出它们。

```
demo% dbx a.out
Reading symbolic information…
(dbx) list 1,25
    1           DIMENSION IARR(4,4)
    2           DO 90 I = 1,4
    3                   DO 20 J = 1,4
    4                           IARR(I,J) = (I*10) + J
    5   20              CONTINUE
    6   90      CONTINUE
    7           END
(dbx)  stop at 7
(1) stop at "Arraysdbx.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "Arraysdbx.f"
    7           END
(dbx) print IARR
iarr =
    (1,1) 11
    (2,1) 21
    (3,1) 31
    (4,1) 41
    (1,2) 12
    (2,2) 22
    (3,2) 32
    (4,2) 42
    (1,3) 13
    (2,3) 23
    (3,3) 33
    (4,3) 43
    (1,4) 14
    (2,4) 24
    (3,4) 34
    (4,4) 44
(dbx) print IARR(2,3)
    iarr(2, 3) = 23  - Order of user-specified subscripts ok
(dbx) quit
```

有关更多信息，请参见“Fortran 数组分片语法” [110]。

Fortran 可分配数组

以下示例说明如何在 dbx 中处理对可分配数组所做的更改。

```
demo% f95 -g Alloc.f95
```

```
 demo% dbx a.out
 (dbx) list 1,99
     1   PROGRAM TestAllocate
     2   INTEGER n, status
     3   INTEGER, ALLOCATABLE :: buffer(:)
     4           PRINT *, 'Size?'
     5           READ *, n
     6           ALLOCATE( buffer(n), STAT=status )
     7           IF ( status /= 0 ) STOP 'cannot allocate buffer'
     8           buffer(n) = n
     9           PRINT *, buffer(n)
    10           DEALLOCATE( buffer, STAT=status)
    11   END
 (dbx) stop at 6
 (2) stop at "alloc.f95":6
 (dbx) stop at 9
 (3) stop at "alloc.f95":9
 (dbx) run
 Running: a.out
 (process id 10749)
  Size?
 1000
 stopped in main at line 6 in file "alloc.f95"
     6           ALLOCATE( buffer(n), STAT=status )
 (dbx) whatis buffer
 integer*4 , allocatable::buffer(:)
 (dbx) next
 continuing
 stopped in main at line 7 in file "alloc.f95"
     7           IF ( status /= 0 ) STOP 'cannot allocate buffer'
 (dbx) whatis buffer
 integer*4 buffer(1:1000)
 (dbx) cont
 stopped in main at line 9 in file "alloc.f95"
     9           PRINT *, buffer(n)
 (dbx) print n
buffer(1000) holds 1000
 n = 1000
 (dbx) print buffer(n)
 buffer(n) = 1000
```

显示内部函数

dbx 可识别 Fortran 内部函数（仅限于 SPARC 平台和 x86 平台）。

要在 dbx 中显示内部函数：

```
demo% cat ShowIntrinsic.f
    INTEGER i
    i = -2
    END
```

```
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
Running: shi
(process id 18019)
stopped in MAIN at line 2 in file "shi.f"
    2             i = -2
(dbx) whatis abs
Generic intrinsic function: "abs"
(dbx) print i
i = 0
(dbx) step
stopped in MAIN at line 3 in file "shi.f"
    3             end
(dbx) print i
i = -2
(dbx) print abs(i)
abs(i) = 2
(dbx)
```

显示复杂表达式

dbx 还可识别 Fortran 复杂表达式。

要在 dbx 中显示复杂表达式：

```
demo% cat ShowComplex.f
   COMPLEX z
    z = ( 2.0, 3.0 )
    END
demo% f95 -g ShowComplex.f
demo% dbx a.out
(dbx) stop in MAIN
(dbx) run
Running: a.out
(process id 10953)
stopped in MAIN at line 2 in file "ShowComplex.f"
    2       z = ( 2.0, 3.0 )
(dbx) whatis z
complex*8  z
(dbx) print z
z = (0.0,0.0)
(dbx) next
stopped in MAIN at line 3 in file "ShowComplex.f"
    3       END
(dbx) print z
z = (2.0,3.0)
(dbx) print z+(1.0,1.0)
z+(1,1) = (3.0,4.0)
(dbx) quit
```

```
demo%
```

显示区间表达式

要在 dbx 中显示区间表达式：

```
demo% cat ShowInterval.f95
   INTERVAL v
   v = [ 37.1, 38.6 ]
   END
demo% f95 -g -xia ShowInterval.f95
demo% dbx a.out
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
Running: a.out
(process id 5217)
stopped in MAIN at line 2 in file "ShowInterval.f95"
    2      v = [ 37.1, 38.6 ]
(dbx) whatis v
INTERVAL*16  v
(dbx) print v
v = [0.0,0.0]
(dbx) next
stopped in MAIN at line 3 in file "ShowInterval.f95"
    3      END
(dbx) print v
v = [37.1,38.6]
(dbx) print v+[0.99,1.01]
v+[0.99,1.01] = [38.09,39.61]
(dbx) quit
demo%
```

显示逻辑运算符

dbx 可以查找 Fortran 逻辑运算符并输出它们。

要在 dbx 中显示逻辑操作符：

```
demo% cat ShowLogical.f
        LOGICAL a, b, y, z
        a = .true.
        b = .false.
        y = .true.
        z = .false.
        END
```

```
demo% f95 -g ShowLogical.f
demo% dbx a.out
(dbx) list 1,9
    1           LOGICAL a, b, y, z
    2           a = .true.
    3           b = .false.
    4           y = .true.
    5           z = .false.
    6           END
(dbx) stop at 5
(2) stop at "ShowLogical.f":5
(dbx) run
Running: a.out
(process id 15394)
stopped in MAIN at line 5 in file "ShowLogical.f"
    5           z = .false.
(dbx) whatis y
logical*4 y
(dbx) print a .or. y
a.OR.y = true
(dbx) assign z = a .or. y
(dbx) print z
z = true
(dbx) quit
demo%
```

查看 Fortran 派生类型

可以使用 dbx 显示结构，即 Fortran 派生类型。

```
demo% f95 -g DebStruc.f95
demo% dbx a.out
(dbx) list 1,99
    1   PROGRAM Struct ! Debug a Structure
    2      TYPE product
    3         INTEGER        id
    4         CHARACTER*16   name
    5         CHARACTER*8    model
    6         REAL           cost
    7 REAL price
    8      END TYPE product
    9
   10      TYPE(product) :: prod1
   11
   12      prod1%id = 82
   13      prod1%name = "Coffee Cup"
   14      prod1%model = "XL"
   15      prod1%cost = 24.0
   16      prod1%price = 104.0
   17      WRITE ( *, * ) prod1%name
   18   END
```

```
(dbx) stop at 17
(2) stop at "Struct.f95":17
(dbx) run
Running: a.out
(process id 12326)
stopped in main at line 17 in file "Struct.f95"
   17      WRITE ( *, * ) prod1%name
(dbx) whatis prod1
product prod1
(dbx) whatis -t product
type product
    integer*4 id
    character*16 name
    character*8 model
    real*4 cost
    real*4 price
end type product
(dbx) n
(dbx) print prod1
    prod1 = (
    id    = 82
    name = 'Coffee Cup'
    model = 'XL'
    cost = 24.0
    price = 104.0
)
```

Fortran 派生类型的指针

可以使用 dbx 显示结构（Fortran 派生类型）和指针。

```
demo% f95 -o debstr -g DebStruc.f95
 demo% dbx debstr
 (dbx) stop in MAIN
 (2) stop in MAIN
 (dbx) list 1,99
     1   PROGRAM DebStruPtr! Debug structures & pointers
Declare a derived type.
     2      TYPE product
     3         INTEGER        id
     4         CHARACTER*16   name
     5         CHARACTER*8    model
     6         REAL           cost
     7         REAL           price
     8      END TYPE product
     9
Declare prod1 and prod2 targets.
    10      TYPE(product), TARGET :: prod1, prod2
Declare curr and prior pointers.
    11      TYPE(product), POINTER :: curr, prior
    12
```

```
Make curr point to prod2.
    13      curr => prod2
Make prior point to prod1.
    14      prior => prod1
Initialize prior.
    15      prior%id = 82
    16      prior%name = "Coffee Cup"
    17      prior%model = "XL"
    18      prior%cost = 24.0
    19      prior%price = 104.0
Set curr to prior.
    20      curr = prior
Print name from curr and prior.
    21      WRITE ( *, * ) curr%name, " ", prior%name
    22   END PROGRAM DebStruPtr
 (dbx) stop at 21
 (1) stop at "DebStruc.f95":21
 (dbx) run
 Running: debstr
(process id 10972)
stopped in main at line 21 in file "DebStruc.f95"
   21      WRITE ( *, * ) curr%name, " ", prior%name
(dbx) print prod1
 prod1 = (
    id = 82
    name = "Coffee Cup"
    model = "XL"
    cost = 24.0
    price = 104.0
)
```

在上一个示例中，dbx 显示派生类型的所有字段，包括字段名称。

可以使用结构并查询有关 Fortran 派生类型的项。

```
Ask about the variable
(dbx) whatis prod1
 product prod1
Ask about the type (-t)
 (dbx) whatis -t product
 type product
    integer*4 id
    character*16 name
    character*8 model
    real cost
    real price
 end type product
```

要输出指针：

```
dbx displays the contents of a pointer, which is an address. This address can be different with every
 run.
(dbx) print prior
 prior = (
```

```
    id    = 82
    name = 'Coffee Cup'
    model = 'XL'
    cost = 24.0
    price = 104.0
)
```

面向对象的 Fortran

dbx 中支持的面向对象的 Fortran 功能为类型扩展和多态指针，这与 C++ 支持一致。

dbxenv 变量 output_dynamic_type 和 output_inherited_members 可用于 Fortran。

可以在 print 和 whatis 命令中使用 -r、+r、-d 和 +d 选项，以获得关于面向对象的 Fortran 代码中继承（父）类型和动态类型的信息。

可分配的标量类型

dbx 支持 Fortran 可分配的标量类型。

♦ ♦ ♦ 第 16 章

16 使用 dbx 调试 Java 应用程序

本章介绍如何使用 dbx 调试使用 Java™ 代码和 C JNI（Java Native Interface，Java 本地接口）代码或 C++ JNI 代码混合编写的应用程序。

本章包含以下各节：

使用 dbx 调试 Java 代码

您可以使用 Oracle Developer Studio dbx 调试正在 Oracle Solaris™ OS 和 Linux OS 下运行的混合代码（Java 代码和 C 代码或 C++ 代码）。

使用 dbx 调试 Java 代码的功能

您可以使用 dbx 调试多种 Java 应用程序。在调试本地代码和 Java 代码时，大多数 dbx 命令的使用方式都类似。

使用 dbx 调试 Java 代码的限制

dbx 在调试 Java 代码时具有以下限制：

- dbx 无法像对待本机代码那样通过信息转储文件来指明 Java 应用程序的状态。
- 如果 Java 应用程序由于某种原因挂起，dbx 无法指示该应用程序的状态，且 dbx 也不能进行过程调用。

- 修复并继续以及运行时检查不适用于 Java 应用程序。

Java 调试的环境变量

以下 dbxenv 变量专门用于使用 dbx 调试 Java 应用程序。您可在启动 dbx 之前从 Shell 提示符或 dbx 命令行中设置 JAVASRCPATH、CLASSPATHX 和 jvm_invocation 环境变量。jdbx_mode 环境变量的设置会在调试应用程序的过程中发生更改。您可使用 jon 命令和 joff 命令更改其设置。

jdbx_mode	jdbx_mode dbxenv 变量可具有以下设置：java、jni 或 native。有关 Java、JNI 和本地模式的说明以及模式变化的方式和时机，请参见“调试 Java 代码的 dbx 模式” [207]。缺省值：java。
JAVASRCPATH	您可使用 JAVASRCPATH dbxenv 变量指定 dbx 应查找的 Java 源文件的目录。当 Java 源文件与 .class 或 .jar 文件不在同一目录中时，此变量十分有用。有关更多信息，请参见“指定 Java 源文件的位置” [203]。
CLASSPATHX	CLASSPATHX dbxenv 变量用于为 Java 类文件指定由定制类装入程序装入的 dbx 路径。有关更多信息，请参见“为使用定制类加载器的类文件指定路径” [204]。
jvm_invocation	通过 jvm_invocation dbxenv 变量可以定制 JVM™ 软件的启动方式。（术语“Java 虚拟机”和 "JVM" 表示用于Java平台的虚拟机。）有关更多信息，请参见“定制 JVM 软件的启动” [204]。

开始调试 Java 应用程序

您可使用 dbx 调试以下类型的 Java 应用程序：

- 文件名以 .class 结尾的文件
- 文件名以 .jar 结尾的文件
- 使用包装器启动的 Java 应用程序
- 在调试模式下启动并连接了 dbx 的运行的 Java 应用程序
- 使用 JNI_CreateJavaVM 接口嵌入 Java 应用程序的 C 应用程序或 C++ 应用程序

在上述所有情况下，dbx 均可识别其正在调试的是 Java 应用程序。

调试类文件

如果定义应用程序的类在包中定义，便需要如同在 JVM 软件上运行应用程序那样加入包的路径，如下例所示。

```
(dbx) debug java.pkg.Toy.class
```

您可使用 dbx 调试使用 .class 文件扩展名的文件。此外，您还可使用类文件的全路径名。通过查找 .class 文件并将全路径名的剩余部分添加到类路径，dbx 可自动确定类路径的软件包部分。例如，对于以下路径名，dbx 将确定 pkg/Toy.class 为主类名，并将 /home/user/java 添加到类路径。

```
(dbx) debug /home/user/java/pkg/Toy.class
```

调试 JAR 文件

Java 应用程序可以使用 JAR（Java 归档）文件打包。您可使用 dbx 调试 JAR 文件。当您开始调试文件名以 .jar 结尾的文件时，dbx 会使用在此 JAR 文件的清单中指定的 Main-Class 属性来确定主类。（主类是作为应用程序入口点的 JAR 文件内的类。）如果使用全路径名或相对路径名来指定 JAR 文件，dbx 会使用目录名，并在 Main-Class 属性中将其作为类路径的前缀。

如果调试不具有 Main-Class 属性的 JAR 文件，则可使用标准版 Java 2 平台的 JarURLConnection 类中指定的 JAR URL 语法 jar:<url>!/{entry} 来指定主类名，如以下示例所示。

```
(dbx) debug jar:myjar.jar!/myclass.class
(dbx) debug jar:/a/b/c/d/e.jar!/x/y/z.class
(dbx) debug jar:file:/a/b/c/d.jar!/myclass.class
```

对于这些示例中的每一个示例，dbx 都会执行以下操作：

- 在 ! 字符后指定的类路径将视为主类（例如，/myclass.class 或 /x/y/z.class）
- 将 JAR 文件的名称（./myjar.jar、/a/b/c/d/e.jar 或 /a/b/c/d.jar）添加到类路径
- 开始调试主类

注 - 如果使用 jvm_invocation 环境变量指定了 JVM 软件的定制启动（请参见“定制 JVM 软件的启动” [204]），则不会将 JAR 文件名自动添加到类路径中。在这种情况下，必须在开始调试时将 JAR 文件名添加到类路径中。

调试有包装器的 Java 应用程序

Java 应用程序通常具有一个可设置环境变量的包装器。如果 Java 应用程序具有包装器，则您需要通过设置环境变量 jvm_invocation 告知 dbx 正在使用包装器脚本。有关更多信息，请参见“定制 JVM 软件的启动” [204]。

将 dbx 连接到正在运行的 Java 应用程序

启动应用程序时，如果指定了以下示例中显示的选项，则可将 dbx 连接到运行的 Java 应用程序。启动应用程序后，您可将 dbx 命令与运行的 Java 进程的进程 ID 结合使用，以开始调试。

```
$ java -agentlib:dbx_agent myclass.class
$ dbx - 2345
```

为了使 JVM 软件能够找到 libdbx_agent.so，需要在运行 Java 应用程序前将适当路径添加到 LD_LIBRARY_PATH 中：

- 在运行 Oracle Solaris OS 的系统的 32 位版本的 JVM 软件上：添加 */install-dir*/SUNWspro/lib/libdbx_agent.so
- 在运行 Oracle Solaris OS 的系统（基于 SPARC）的 64 位版本的 JVM 软件上：将 */install-dir*/SUNWspro/lib/v9/libdbx_agent.so 添加到 LD_LIBRARY_PATH
- 在运行 Linux OS 的系统（基于 x64）的 64位 版本的 JVM 软件上：将 */install-dir*/sunstudio12/lib/amd64/libdbx_agent.so 添加到 LD_LIBRARY_PATH

install-dir 是 Oracle Developer Studio 安装的位置。

将 dbx 连接到运行的应用程序时，dbx 将以 Java 模式开始调试应用程序。

如果 Java 应用程序需要 64 位的对象库，请在启动应用程序时包括 -d64 选项。之后，将 dbx 连接到应用程序时，dbx 将使用运行该应用程序的 64 位 JVM 软件。

```
$ java -agentlib:dbx_agent
$ dbx - 2345
```

以下任务介绍如何使用进程 ID 将 dbx 连接到特定的 Java 进程。

▼ 连接到运行中的 Java 进程

1. **按照上一节中所述，通过将 libdbx_agent.so 添加到 LD_LIBRARY_PATH，确保 JVM™ 软件可找到 libdbx_agent.so。**

2. **通过键入以下命令启动 Java 应用程序：**

   java -agentlib:dbx_agent *myclass*.class

3. **然后，通过使用进程 ID 启动 dbx 便可连接到进程：**

   dbx -*process-ID*

调试内嵌 Java 应用程序的 C 应用程序或 C++ 应用程序

可以使用 JNI_CreateJavaVM 接口调试内嵌 Java 应用程序的 C 应用程序或 C++ 应用程序。C 应用程序或 C++ 应用程序必须通过将以下选项指定给 JVM 软件来启动 Java 应用程序：

```
-agentlib:dbx_agent
```

对于查找 libdbx_agent.so 的 JVM 软件，您需要在运行 Java 应用程序之前将正确的路径添加到 LD_LIBRARY_PATH。请参见“将 dbx 连接到正在运行的 Java 应用程序” [202]。

install-dir 是 Oracle Developer Studio 软件的安装位置。

将参数传递给 JVM 软件

在 Java 模式下使用 run 命令时，会将提供的参数传递给应用程序，而非 JVM 软件。要将参数传递给 JVM 软件，请参见“定制 JVM 软件的启动” [204]。

指定 Java 源文件的位置

有时，Java 源文件所在的目录可能与 .class 或 .jar 文件所在的目录不同。可以使用 $JAVASRCPATH 环境变量指定 dbx 查找 Java 源文件的目录。在以下示例中，dbx 将查找列出的源文件（与调试的类文件对应）目录。

```
JAVASRCPATH=.:/mydir/mysrc:/mydir/mylibsrc:/mydir/myutils
```

指定 C 源文件或 C++ 源文件的位置

dbx 可能无法在以下环境中找到 C 源文件或 C++ 源文件：

- 如果源文件不在编译时所在的位置
- 如果编译源文件的系统与运行 dbx 的系统不同，而且编译目录的路径名不同

在此类情况下，请使用 pathmap 命令（请参见“pathmap 命令” [327]）将一个路径名映射到另一个路径名，以便 dbx 可以找到您的文件。

为使用定制类加载器的类文件指定路径

应用程序可以有从可能不属于常规类路径的位置装入类文件的定制类加载器。在这种情况下，dbx 无法找到类文件。使用环境变量 CLASSPATHX 可以为 dbx 指定由定制类加载器装入的类文件的路径。例如，CLASSPATHX=.:/myloader/myclass:/mydir/mycustom 将导致 dbx 在尝试查找类文件时查找列出的目录。

在 Java 方法中设置断点

与本地应用程序不同，Java 应用程序不包含便于访问的名称索引。因此，例如，您不能仅指定方法名称：

```
(dbx) stop in myMethod #This will not work
```

而是需要使用该方法的完整路径。

```
(dbx) stop in com.any.library.MyClass.myMethod
```

当您因使用 MyClass 的某种方法（myMethod 原本应已足够）而被迫停止时，便会出现异常情况。

有一种方法可避免将完整路径包括在该方法中，即：使用 stop inmethod。

```
(dbx) stop inmethod myMethod
```

但是，此命令可能会导致停止多个方法名称 myMethod。

在本地 (JNI) 代码中设置断点

包含 JNI C 或 C++ 代码的共享库将由 JVM 动态装入，且在其中设置断点时，需要采取一些其他步骤。有关更多信息，请参见“在动态装入的库中设置断点” [97]。

定制 JVM 软件的启动

您可能需要从 dbx 中定制 JVM 软件的启动，才能执行某些任务。涉及定制的常见任务包括：

可以使用环境变量 jvm_invocation 定制 JVM 软件的启动。缺省情况下，环境变量 jvm_invocation 未定义时，dbx 将按以下方式启动 JVM 软件

```
java -agentlib:dbx_agent=sync=process-ID
```

定义环境变量 jvm_invocation 后，dbx 会使用该变量的值来启动 JVM 软件。

定义 jvm_invocation 环境变量时，必须包含 -Xdebug 选项。dbx 将 -Xdebug 扩展至内部选项 -Xdebug- Xnoagent -Xrundbxagent:sync。

如果不在定义中包括 -Xdebug 选项，如下例所示，dbx 会显示错误消息。

```
jvm_invocation="/set/java/javasoft/sparc-S2/jdk1.2/bin/java"

dbx: Value of `$jvm_invocation' must include an option to invoke the VM in debug mode
```

指定 JVM 软件的路径名

缺省情况下，如果不指定 JVM 软件的路径名，dbx 将在您的路径中启动 JVM 软件。

要指定 JVM 软件的路径名，请将 jvm_invocation 环境变量设置为正确的路径名，如以下示例所示。

```
jvm_invocation="/myjava/java -Xdebug"
```

此设置会使 dbx 按以下方式启动 JVM 软件：

```
/myjava/java -agentlib:dbx_agent=sync
```

将运行参数传递给 JVM 软件

要将运行参数传递给 JVM 软件，请对环境变量 jvm_invocation 进行设置，以使用那些参数启动 JVM 软件，如下例所示。

```
jvm_invocation="java -Xdebug -Xms512 -Xmx1024 -Xcheck:jni"
```

此示例将导致 dbx 按如下方式启动 JVM 软件：

```
java -agentlib:dbx_agent=sync= -Xms512 -Xmx1024 -Xcheck:jni
```

指定 Java 应用程序的定制包装器

Java 应用程序可使用定制包装器执行启动。如果应用程序使用定制包装器，则您可使用 jvm_invocation 环境变量指定要使用的包装器，如以下示例所示。

```
jvm_invocation="/export/siva-a/forte4j/bin/forte4j.sh -J-Xdebug"
```

此示例将导致 dbx 按如下方式启动 JVM 软件：

```
/export/siva-a/forte4j/bin/forte4j.sh - -agentlib:dbx_agent=sync=process-ID
```

使用接受命令行选项的定制包装器

以下包装器脚本 (xyz) 将设置几个环境变量并接受命令行选项。

```
#!/bin/sh
CPATH=/mydir/myclass:/mydir/myjar.jar; export CPATH
JARGS="-verbose:gc -verbose:jni -DXYZ=/mydir/xyz"
ARGS=
while [ $# -gt 0 ] ; do
    case "$1" in
        -userdir) shift; if [ $# -gt 0 ]
; then userdir=$1; fi;;
        -J*) jopt=`expr $1 : '-J<.*>'`
; JARGS="$JARGS '$jopt'";;
        *) ARGS="$ARGS '$1'" ;;
    esac
    shift
done
java $JARGS -cp $CPATH $ARGS
```

此脚本接受 JVM 软件和用户应用程序的一些命令行选项。对于这种形式的包装器脚本，可设置环境变量 jvm_invocation 并按以下方式启动 dbx：

```
% jvm_invocation="xyz -J-Xdebug -J other-java-options"
% dbx myclass.class -Dide=visual
```

使用不接受命令行选项的定制包装器

以下包装器脚本 (xyz) 将设置几个环境变量并启动 JVM 软件，但不接受任何命令行选项或类名。

```
#!/bin/sh
CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
ABC=/mydir/abc; export ABC
java <options> myclass
```

可以通过 dbx 并采用以下两种方法之一，使用此类脚本来调试包装器：

- 通过将 jvm_invocation 变量的定义添加到脚本并启动 dbx来修改脚本，以便从包装器脚本内部启动 dbx。

  ```
  #!/bin/sh
  CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
  ABC=/mydir/abc; export ABC
  jvm_invocation="java -Xdebug <options>"; export jvm_invocation
  dbx myclass.class
  ```

修改后，便可运行脚本来启动调试会话。

- 按如下所示稍微修改脚本，接受一些命令行选项：

```
#!/bin/sh
CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
ABC=/mydir/abc; export ABC
JAVA_OPTIONS="$1 <options>"
java $JAVA_OPTIONS $2
```

修改后，便可设置环境变量 jvm_invocation，并按以下方式启动 dbx：

```
% jvm_invocation="xyz -Xdebug"; export jvm_invocation
% dbx myclass.class
```

指定 64 位 JVM 软件

如果需要 dbx 启动 64 位 JVM 软件，以调试一个需要 64 位对象库的应用程序，请在设置 jvm_invocation 环境变量时包括 -d64 选项。

```
jvm_invocation="/myjava/java -Xdebug -d64"
```

调试 Java 代码的 dbx 模式

调试 Java 应用程序时，dbx 处于以下三种模式之一：

- Java 模式
- JNI 模式
- 本地模式

当 dbx 处于 Java 模式或 JNI（Java 本地接口）模式时，您可以检查 Java 应用程序的状态（其中包括 JNI 代码），并控制代码的执行。当 dbx 处于本地模式下时，可以检查 C 或 C++ JNI 代码的状态。当前模式（java、jni、native）存储在环境变量 jdbx_mode 中。

在 Java 模式下，可以使用 Java 语法与 dbx 交互，dbx 会使用 Java 语法显示信息。此模式用于调试纯 Java 代码或使用 Java 代码、C JNI 代码或 C++ JNI 代码混合编写的应用程序中的 Java 代码。

在 JNI 模式下，dbx 命令使用本地语法并会影响本地代码，但命令的输出既显示与 Java 有关的状态，也显示本地状态，所以 JNI 模式是一种“混合”模式。此模式用于调试使用 Java 代码、C JNI 代码或 C++ JNI 代码混合编写的应用程序的本地部分。

在本地模式下，dbx 命令仅影响本地程序，与 Java 相关的所有功能将被禁用。此模式用于调试与 Java 无关的程序。

在执行 Java 应用程序的过程中，dbx 会根据情况自动在 Java 模式和 JNI 模式间切换。例如，遇到 Java 断点时，dbx 会自动切换到 Java 模式，而当您从 Java 代码步入 JNI 代码时，它又会切换到 JNI 模式。

从 Java 或 JNI 模式切换到本地模式

dbx 不会自动切换到本地模式。可以使用 joff 命令显式从 Java 模式或 JNI 模式切换到本地模式，也可以使用 jon 命令从本地模式切换到 Java 模式。

中断执行时切换模式

如果中断执行 Java 应用程序（例如，通过键入 Ctrl-C），dbx 会尝试通过将应用程序置于安全状态并挂起所有线程将模式自动设置为 Java/JNI 模式。

如果 dbx 无法挂起应用程序并切换到 Java/JNI 模式，dbx 便会切换到本地模式。稍后，您可使用 jon 命令切换到 Java 模式，以便能够检查程序的状态。

在 Java 模式下使用 dbx 命令

使用 dbx 调试由 Java 代码和本地代码组成的混合型代码时，dbx 命令分为以下几类：

- 第一类命令：接受相同的参数，而且在 Java 模式或 JNI 模式下的运行方式与本地模式下相同。请参见“在 Java 模式和本地模式下具有完全相同语法和功能的命令” [209]。
- 第二类命令：具有仅在 Java 模式或 JNI 模式下有效的参数，或者仅在本地模式下有效的参数。请参见“在 Java 模式下有不同语法的命令” [210]。
- 仅在 Java 模式或 JNI 模式下有效的命令。请参见“只在 Java 模式下有效的命令” [211]。

未包括在上述任一类别的命令只在本地模式下有效。

dbx 命令中的 Java 表达式求值

大多数 dbx 命令中使用的 Java 表达式计算器可支持以下构造：

- 所有文字
- 所有名称和字段访问
- this 和 super

- 数组访问
- 类型转换
- 条件二进制运算
- 方法调用
- 其他一元/二进制运算
- 对变量或字段赋值
- `instanceof` 运算符
- 数组长度操作符

Java 表达式计算器不支持以下构造：

- 限定的 `this`，例如 <*ClassName*>`.this`
- 类实例创建表达式
- 数组创建表达式
- 字符串并置操作符
- 条件操作符 ?:
- 复合赋值操作符，例如 `x += 3`

一种特别有用的检查 Java 应用程序状态的方式是在 IDE 或 `dbxtool` 中使用监视功能。

不要依赖于表达式中的精确值语义，它们的工作远不仅限于检查数据。

dbx 命令使用的静态信息和动态信息

只有在 JVM 软件启动后，有关 Java 应用程序的许多信息才可正常使用，并且执行完 Java 应用程序后，这些信息将不再使用。但是，使用 `dbx` 调试 Java 应用程序时，`dbx` 会在启动 JVM 软件前从属于系统类路径和用户类路径的类文件和 JAR 文件中收集其需要的一些信息。此信息可使 `dbx` 在您运行应用程序之前，对断点执行更好的错误检查。

有些 Java 类及其属性可能无法通过类路径访问。`dbx` 可检查并单步执行这些类，表达式解析器则可在这些类于运行时装入后对它们进行访问。但它收集的信息是临时性的，JVM 软件终止后便不再可用。

`dbx` 调试 Java 应用程序所需的某些信息在任何地方均无记录，因此，`dbx` 会在调试代码期间浏览 Java 源文件来取得这些信息。

在 Java 模式和本地模式下具有完全相同语法和功能的命令

下表中列出的 `dbx` 命令在 Java 模式下和本地模式下具有相同的语法并执行相同的操作。

命令	功能
attach	将 dbx 连接到正在运行的进程，从而停止执行并将程序置于调试控制之下
cont	使进程继续执行
dbxenv	列出或设置 dbxenv 变量
delete	删除断点和其他事件
down	将调用堆栈下移（远离 main）
dump	输出过程或方法的所有局部变量
file	列出或更改当前文件
frame	列出或更改当前堆栈帧号
handler	修改事件处理程序（断点）
import	从 dbx 命令库中导入命令
line	列出或更改当前行号
list	显示源文件的行
next	单步执行一个源代码行（步过调用）
pathmap	将一个路径名映射到另一个路径名，以查找源文件等
proc	显示当前进程的状态
prog	管理正被调试的程序和它们的属性
quit	退出 dbx
rerun	不带参数运行程序
runargs	更改目标进程的参数
status	列出事件处理程序（断点）
step up	向上单步执行并步出当前函数或方法
stepi	单步执行一个计算机指令（步入调用）
up	将调用堆栈上移（靠近 main）
whereami	显示当前源代码行

在 Java 模式下有不同语法的命令

下表中列出的 dbx 命令在进行 Java 调试时所用的语法与进行本地代码调试时所用的语法不同，而且在 Java 模式下的运行方式也与本地模式下的运行方式不同。

命令	本地模式功能	Java 模式功能
assign	为程序变量赋新值	为局部变量或参数赋新值
call	调用过程	调用方法
dbx	启动 dbx	启动 dbx
debug	装入指定应用程序，然后开始调试该应用程序	装入指定 Java 应用程序，接着检查类文件是否存在，然后开始调试应用程序

命令	本地模式功能	Java 模式功能
detach	使目标进程脱离 dbx 的控制	使目标进程脱离 dbx 的控制
display	在每个停止点对表达式求值并输出	在每个停止点对表达式、局部变量或参数求值并输出
files	列出与某个正规表达式匹配的文件名	列出 dbx 已知的所有 Java 源文件
func	列出或更改当前函数	列出或更改当前方法
next	单步执行一个源代码行（步过调用）	单步执行一个源代码行（步过调用）
print	输出表达式的值	输出表达式、局部变量或参数的值
run	带参数运行程序	带参数运行程序
step	单步执行一个源代码行或语句（正在步入调用）	单步执行一个源代码行或语句（正在步入调用）
stop	设置源代码级断点	设置源代码级断点
thread	列出或更改当前线程	列出或更改当前线程
threads	列出所有线程	列出所有线程
trace	显示执行的源代码行、函数调用或变量更改	显示执行的源代码行、函数调用或变量更改
undisplay	撤消 display 命令	撤消 display 命令
whatis	输出表达式类型或类型声明	输出标识符声明
when	指定事件发生时执行命令	指定事件发生时执行命令
where	输出调用堆栈	输出调用堆栈

只在 Java 模式下有效的命令

下表中列出的 dbx 命令仅在 Java 模式或 JNI 模式下有效。

命令	功能
java	当 dbx 处于 JNI 模式下时，用于指示将执行的是 Java 版本的指定命令
jclasses	发出该命令后，输出 dbx 已知的所有 Java 类的名称
joff	将 dbx 从 Java 模式或 JNI 模式切换到本地模式
jon	将 dbx 从本地模式切换到 Java 模式
jpkgs	发出该命令后，输出 dbx 已知的所有 Java 程序包的名称
native	当 dbx 处于 Java 模式下时，用于指示将执行的是本地版本的指定命令

♦ ♦ ♦ 第 17 章

17 在计算机指令级调试

本章介绍了如何在计算机指令级使用事件管理和进程控制命令、如何显示指定地址的内存内容以及如何显示源代码行及其对应的计算机指令。

本章包含以下各节：

在计算机指令级使用 dbx

next 命令、step 命令、stop 命令和 trace 命令分别支持相应的计算机指令级变体：nexti 命令、stepi 命令、stopi 命令和 tracei 命令。使用 regs 命令输出计算机寄存器的内容，或者使用 print 命令输出单个寄存器。

检查内存的内容

可使用地址以及 examine 或 x 命令检查内存位置的内容及输出每个地址处的汇编语言指令。使用从 adb(1)（汇编语言调试器）派生的命令，可以查询以下内容：

- *address*（地址），使用 =（等号）字符
- 某地址处存储的 *contents*（内容），使用 /（斜线）字符

可使用 dis 命令和 listi 命令输出汇编命令。

examine 或 x 命令用法

可使用 examine 命令或其别名 x 显示内存内容或地址。

下列语法用于以 *format* 格式显示始于 *address* 的 *count* 项内存内容。缺省的 *address* 为先前显示的最后一个地址后的下一个地址。缺省 *count* 为 1。缺省 *format* 与在先前的 examine 命令中使用的相同；如果这是给出的第一个命令，则为 X。

examine 命令的语法如下：

```
examine [address] [/ [count] [format]]
```

要以 *format* 格式显示 *address1* 到 *address2*（首末地址包含在内）的内存内容：

```
examine address1, address2 [/ [format]]
```

要以指定格式显示地址（而不是地址的内容）：

```
examine address = [format]
```

要输出存储在 examine 显示的最后一个地址之后的下一个地址中的值：

```
examine +/ i
```

要输出表达式的值，请提供表达式作为地址。

```
examine address=format
examine address=
```

使用地址

address 是求值结果为地址或可用作地址的任何表达式。*address* 可以替换为 +（加号），该符号以缺省格式显示下一个地址的内容。

以下示例为有效地址：

`0xff00`	绝对地址
`main`	函数地址
`main+20`	与函数地址的偏移
`&errno`	变量地址
`str`	指向字符串的指针值变量

用于显示内存的符号地址的名称前有和号 (&)。使用函数名称时不必带有和符号；&main 等效于 main。寄存器名称前有美元符号 ($)。

使用格式

该格式为 dbx 显示查询结果的地址显示格式。生成的输出取决于当前显示格式。要更改显示格式，应提供不同的格式代码。

在每个 dbx 会话的开始处设置的缺省格式为 X，该格式将地址或值显示为十六进制 32 位字。以下内存显示格式合法：

i	显示为汇编指令
d	显示为十进制 16 位（2 字节）
D	显示为十进制 32 位（4 字节）
o	显示为八进制 16 位（2 字节）
O	显示为八进制 32 位（4 字节）
x	显示为十六进制 16 位（2 字节）
X	显示为十六进制 32 位（4 字节）（缺省格式）
b	显示为八进制字节
c	显示为字符
n	显示为十进制（1 字节）。
w	显示为宽字符
s	显示为以空字节终止的字符串
W	显示为宽字符串
f	显示为单精度浮点数
F, g	显示为双精度浮点数
E	显示为扩展精度浮点数
ld, lD	显示为十进制 32 位（4 字节）（与 D 相同）
lo, lO	显示为八进制 32 位（4 字节）（与 O 相同）
lx, LX	显示为十六进制 32 位（4 字节）（与 X 相同）
Ld, LD	显示为十进制 64 位（8 字节）
Lo, LO	显示为八进制 64 位（8 字节）
Lx, LX	显示为十六进制 64 位（8 字节）

使用计数

该计数是以十进制表示的重复计数。增量大小取决于内存显示格式。

使用地址的示例

以下示例说明如何使用地址和格式选项来显示从当前停止点开始的五个连续反汇编指令。

对于基于 SPARC 的系统：

```
(dbx) stepi
stopped in main at 0x108bc
0x000108bc: main+0x000c: st    %l0, [%fp - 0x14]
(dbx) x 0x108bc/5i
0x000108bc: main+0x000c: st    %l0, [%fp - 0x14]
0x000108c0: main+0x0010: mov   0x1,%l0
0x000108c4: main+0x0014: or    %l0,%g0, %o0
0x000108c8: main+0x0018: call  0x00020b90 [unresolved PLT 8: malloc]
0x000108cc: main+0x001c: nop
```

对于基于 x86 的系统：

```
(dbx) x &main/5i
0x08048988: main       :  pushl  %ebp
0x08048989: main+0x0001:  movl   %esp,%ebp
0x0804898b: main+0x0003:  subl   $0x28,%esp
0x0804898e: main+0x0006:  movl   0x8048ac0,%eax
0x08048993: main+0x000b:  movl   %eax,-8(%ebp)
```

dis 命令用法

dis 命令等同于以 i 作为缺省显示格式的 examine 命令。

dis 命令的语法为：

dis [<*address*>] [/<*count*>] | <*address1*>, <*address1*>

dis 命令操作如下：

- 不使用参数时显示以 *address* 开始的 10 个指令
- 只使用 *address* 参数时，反汇编始于 *address* 的 10 个指令
- 使用 *address* 参数和 *count*，反汇编 *count* 个始于 *address* 的指令
- 使用 *address1* 和 *address2* 参数，反汇编 *address1* 到 *address2* 的指令
- 仅使用 *count*，显示从 + 开始的 *count* 条指令
- 使用 -a 选项时，反汇编整个函数，如果使用时不带参数，则反汇编当前访问函数的剩余部分

listi 命令用法

要显示源代码行及其对应的汇编指令，请使用 listi 命令，该命令等效于 list -i 命令。请参见“输出源代码列表” [63] 中有关 list -i 的讨论。

基于 SPARC 的系统示例：

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x0001083c: main+0x0014:  ld      [%fp + 0x48], %l0
0x00010840: main+0x0018:  add     %l0, 0x4, %l0
0x00010844: main+0x001c:  ld      [%l0], %l0
0x00010848: main+0x0020:  or      %l0, %g0, %o0
0x0001084c: main+0x0024:  call    0x000209e8 [unresolved PLT 7: atoi]
0x00010850: main+0x0028:  nop
0x00010854: main+0x002c:  or      %o0, %g0, %l0
0x00010858: main+0x0030:  st      %l0, [%fp - 0x8]
   14       j = foo(i);
0x0001085c: main+0x0034:  ld      [%fp - 0x8], %l0
0x00010860: main+0x0038:  or      %l0, %g0, %o0
0x00010864: main+0x003c:  call    foo
0x00010868: main+0x0040:  nop
0x0001086c: main+0x0044:  or      %o0, %g0, %l0
0x00010870: main+0x0048:  st      %l0, [%fp - 0xc]
```

基于 x86 的系统示例：

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x080488fd: main+0x000d:  movl   12(%ebp),%eax
0x08048900: main+0x0010:  movl   4(%eax),%eax
0x08048903: main+0x0013:  pushl  %eax
0x08048904: main+0x0014:  call   atoi <0x8048798>
0x08048909: main+0x0019:  addl   $4,%esp
0x0804890c: main+0x001c:  movl   %eax,-8(%ebp)
   14       j = foo(i);
0x0804890f: main+0x001f:  movl   -8(%ebp),%eax
0x08048912: main+0x0022:  pushl  %eax
0x08048913: main+0x0023:  call   foo <0x80488c0>
0x08048918: main+0x0028:  addl   $4,%esp
0x0804891b: main+0x002b:  movl   %eax,-12(%ebp)
```

在计算机指令级单步执行和跟踪

计算机指令级命令与其对应的源代码级命令的功能相同，只不过它们在单步指令级而非源代码行级执行。

在计算机指令级单步执行

要从一个计算机指令到下一个计算机指令单步执行，请使用 `nexti` 命令或 `stepi` 命令。

`nexti` 命令和 `stepi` 命令的行为与其相对应的源代码级行为相同：`nexti` 命令步过函数，而 `stepi` 函数步入下一个指令调用的函数，停止于被调用函数中的第一个指令。命令形式也相同。

nexti 命令和 stepi 命令的输出与对应的源代码级命令在两个方面有所不同：

- 输出中包含程序停止处的指令地址（而非源代码行号）。
- 缺省输出中包含反汇编指令，而非源代码行。

例如：

```
(dbx) func
hand::ungrasp
(dbx) nexti
ungrasp +0x18:  call support
(dbx)
```

有关更多信息，请参见“nexti 命令” [324] 和“stepi 命令” [343]。

在计算机指令级跟踪

计算机级的跟踪技术与源代码级相同，只是要使用 tracei 命令。执行 tracei 命令时，dbx 只会在每次检查执行的地址或跟踪的变量值后，执行一个指令。tracei 命令会产生类似 stepi 的自动行为。程序一次前进一个指令来步入函数调用。

使用 tracei 命令时，它会使程序在执行每个指令后停止一会儿，这时，dbx 检查地址执行情况或跟踪的变量或表达式的值。使用 tracei 命令会显著降低执行速度。

有关 trace 及其事件规范和修饰符的更多信息，请参见“跟踪执行” [96] 和“tracei 命令” [359]。

tracei 命令的通用语法为：

tracei *event-specification* [*modifier*]

tracei 命令的常用形式为：

tracei step	跟踪每个指令
tracei next	跟踪每一指令，但跳过调用
tracei at *address*	跟踪给定代码地址。

有关更多信息，请参见“tracei 命令” [359]。

对于 SPARC：

```
(dbx) tracei next -in main
(dbx) cont
0x00010814: main+0x0004:  clr     %l0
0x00010818: main+0x0008:  st      %l0, [%fp - 0x8]
0x0001081c: main+0x000c:  call    foo
0x00010820: main+0x0010:  nop
```

```
0x00010824: main+0x0014:  clr     %l0
....
....
(dbx) (dbx) tracei step -in foo -if glob == 0
(dbx) cont
0x000107dc: foo+0x0004:  mov     0x2, %l1
0x000107e0: foo+0x0008:  sethi   %hi(0x20800), %l0
0x000107e4: foo+0x000c:  or      %l0, 0x1f4, %l0     ! glob
0x000107e8: foo+0x0010:  st      %l1, [%l0]
0x000107ec: foo+0x0014:  ba      foo+0x1c
....
....
```

在计算机指令级设置断点

要在计算机指令级设置断点，请使用 `stopi` 命令。该命令接受任何事件规范。`stopi` 命令的语法为：

`stopi` *event-specification* `[`*modifier*`]`

`stopi` 命令的常用形式如下：

`stopi [at` *address*`] [-if` *cond*`]`
`stopi in` *function* `[-if` *cond*`]`

有关更多信息，请参见“`stopi` 命令” [348]。

在地址处设置断点

使用 `stopi` 命令可在特定地址处设置断点：

`(dbx)` **`stopi at`** *address*

例如：

```
(dbx) nexti
stopped in hand::ungrasp at 0x12638
(dbx) stopi at &hand::ungrasp
(3) stopi at &hand::ungrasp
(dbx)
```

regs 命令用法

使用 `regs` 命令可以输出所有寄存器的值。

regs 命令的语法为：

```
regs [-f][-F]
```

-f 表示包括浮点寄存器（单精度）。-F 表示包括浮点寄存器（双精度）。

有关更多信息，请参见“regs 命令” [333]。

基于 SPARC 的系统示例：

```
dbx[13] regs -F
current thread: t@1
current frame:  [1]
g0-g3    0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7    0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3    0x00000003 0x00000014 0xef7562b4 0xeffff420
o4-o7    0xef752f80 0x00000003 0xeffff3d8 0x000109b8
l0-l3    0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7    0xeffff438 0x00000001 0x00000007 0xef74df54
i0-i3    0x00000001 0xeffff4a4 0xeffff4ac 0x00020c00
i4-i7    0x00000001 0x00000000 0xeffff440 0x000108c4
y        0x00000000
psr      0x40400086
pc       0x000109c0:main+0x4    mov     0x5, %l0
npc      0x000109c4:main+0x8    st      %l0, [%fp - 0x8]
f0f1     +0.00000000000000e+00
f2f3     +0.00000000000000e+00
f4f5     +0.00000000000000e+00
f6f7     +0.00000000000000e+00
...
```

对于基于 x64 的系统示例：

```
(dbx) regs
current frame:  [1]
r15     0x0000000000000000
r14     0x0000000000000000
r13     0x0000000000000000
r12     0x0000000000000000
r11     0x0000000000401b58
r10     0x0000000000000000
r9      0x0000000000401c30
r8      0x0000000000416cf0
rdi     0x0000000000416cf0
rsi     0x0000000000401c18
rbp     0xfffffd7fffdff820
rbx     0xfffffd7fff3fb190
rdx     0x0000000000401b50
rcx     0x0000000000401b54
rax     0x0000000000416cf0
trapno  0x0000000000000003
err     0x0000000000000000
rip     0x0000000000401709:main+0xf9    movl $0x0000000000000000,0xfffffffffffffffc(%rbp)
cs      0x000000000000004b
eflags  0x0000000000000206
```

```
rsp     0xfffffd7fffdff7b0
ss      0x0000000000000043
fs      0x00000000000001bb
gs      0x0000000000000000
es      0x0000000000000000
ds      0x0000000000000000
fs_base  0xfffffd7fff3a2000
gsbase  0xffffffff80000000
(dbx) regs -F
current frame:  [1]
r15     0x0000000000000000
r14     0x0000000000000000
r13     0x0000000000000000
r12     0x0000000000000000
r11     0x0000000000401b58
r10     0x0000000000000000
r9      0x0000000000401c30
r8      0x0000000000416cf0
rdi     0x0000000000416cf0
rsi     0x0000000000401c18
rbp     0xfffffd7fffdff820
rbx     0xfffffd7fff3fb190
rdx     0x0000000000401b50
rcx     0x0000000000401b54
rax     0x0000000000416cf0
trapno  0x0000000000000003
err     0x0000000000000000
rip     0x0000000000401709:main+0xf9    movl     $0x0000000000000000,0xfffffffffffffffc(%rbp)
cs      0x000000000000004b
eflags  0x0000000000000206
rsp     0xfffffd7fffdff7b0
ss      0x0000000000000043
fs      0x00000000000001bb
gs      0x0000000000000000
es      0x0000000000000000
ds      0x0000000000000000
fs_base  0xfffffd7fff3a2000
gsbase  0xffffffff80000000
st0     +0.00000000000000000000e+00
st1     +0.00000000000000000000e+00
st2     +0.00000000000000000000e+00
st3     +0.00000000000000000000e+00
st4     +0.00000000000000000000e+00
st5     +0.00000000000000000000e+00
st6     +0.00000000000000000000e+00
st7     +NaN
xmm0a-xmm0d     0x00000000 0xfff80000 0x00000000 0x00000000
xmm1a-xmm1d     0x00000000 0x00000000 0x00000000 0x00000000
xmm2a-xmm2d     0x00000000 0x00000000 0x00000000 0x00000000
xmm3a-xmm3d     0x00000000 0x00000000 0x00000000 0x00000000
xmm4a-xmm4d     0x00000000 0x00000000 0x00000000 0x00000000
xmm5a-xmm5d     0x00000000 0x00000000 0x00000000 0x00000000
xmm6a-xmm6d     0x00000000 0x00000000 0x00000000 0x00000000
xmm7a-xmm7d     0x00000000 0x00000000 0x00000000 0x00000000
```

```
xmm8a-xmm8d     0x00000000 0x00000000 0x00000000 0x00000000
xmm9a-xmm9d     0x00000000 0x00000000 0x00000000 0x00000000
xmm10a-xmm10d   0x00000000 0x00000000 0x00000000 0x00000000
xmm11a-xmm11d   0x00000000 0x00000000 0x00000000 0x00000000
xmm12a-xmm12d   0x00000000 0x00000000 0x00000000 0x00000000
xmm13a-xmm13d   0x00000000 0x00000000 0x00000000 0x00000000
xmm14a-xmm14d   0x00000000 0x00000000 0x00000000 0x00000000
xmm15a-xmm15d   0x00000000 0x00000000 0x00000000 0x00000000
fcw-fsw  0x137f 0x0000
fctw-fop        0x0000 0x0000
frip     0x0000000000000000
frdp     0x0000000000000000
mxcsr    0x00001f80
mxcr_mask       0x0000ffff
(dbx)
```

平台特定寄存器

本节中的表针对可在表达式中使用的 SPARC 体系结构、x86 体系结构和 AMD64 体系结构，列出了平台特定寄存器的名称。

SPARC 寄存器信息

下表列出了适用于 SPARC 体系结构的寄存器信息。

寄存器	说明
`$g0 through $g7`	全局寄存器
`$o0 through $o7`	“外部”寄存器
`$l0 through $l7`	“本地”寄存器
`$i0 through $i7`	“内部”寄存器
`$fp`	帧指针，等同于寄存器 $i6
`$sp`	堆栈指针，等同于寄存器 $o6
`$y`	Y 寄存器
`$psr`	处理器状态寄存器
`$wim`	窗口无效屏蔽寄存器
`$tbr`	捕获基址寄存器
`$pc`	程序计数器
`$npc`	下一程序计数器
`$f0 through $f31`	FPU “f” 寄存器
`$fsr`	FPU 状态寄存器
`$fq`	FPU 队列

浮点寄存器的 $f0f1 $f2f3 ... $f30f31 对被视为具有 C 双精度类型（通常 $fN 寄存器被视为 C 浮点类型）。这些对也称为 $d0 ... $d30。

以下四浮点寄存器可被视为具有 C 长双精度型，它们适用于 SPARC V9 硬件：

```
$q0 $q4 through $q60
```

以下寄存器对组合了两个寄存器的最低有效 32 位，可在 SPARC V8+ 硬件上使用：

```
$g0g1 through $g6g7
$o0o1 through $o6o7
```

SPARC V9 和 V8+ 硬件上另外还提供了以下这些寄存器：

```
$xg0 through $xg7
$xo0 through $xo7
$xfsr $tstate $gsr
$f32f33 $f34f35 through $f62f63 ($d32 ... $$d62)
```

有关 SPARC 寄存器和寻址的更多信息，请参见《*SPARC Architecture Reference Manual*》（《SPARC 体系结构参考手册》）和《*SPARC Assembly Language Reference Manual*》（《SPARC 汇编语言参考手册》）。

x86 寄存器信息

下表列出了适用于 x86 体系结构的寄存器信息。

寄存器	说明
$gs	交替数据段寄存器
$fs	交替数据段寄存器
$es	交替数据段寄存器
$ds	数据段寄存器
$edi	目标索引寄存器
$esi	源索引寄存器
$ebp	帧指针
$esp	堆栈指针
$ebx	通用寄存器
$edx	通用寄存器
$ecx	通用寄存器
$eax	通用寄存器
$trapno	异常向量数
$err	异常错误代码
$eip	指令指针
$cs	代码段寄存器

寄存器	说明
$eflags	标志
$ss	堆栈段寄存器

常用寄存器也使用其计算机无关名称作为别名。

寄存器	说明
$sp	堆栈指针，等同于 $uesp
$pc	程序计数器，等同于 $eip
$fp	帧指针，等同于 $ebp
$ps	处理器状态寄存器

下表列出了 80386 下半部分（16 位）的寄存器。

寄存器	说明
$ax	通用寄存器
$cx	通用寄存器
$dx	通用寄存器
$bx	通用寄存器
$si	源索引寄存器
$di	目标索引寄存器
$ip	指令指针，下 16 位
$flags	标志，下 16 位

80386 的前四个 16 位寄存器可分为多个 8 位部分，如下表所示：

寄存器	说明
$al	寄存器 $ax 的（右）下半部分
$ah	寄存器 $ax 的（左）上半部分
$cl	寄存器 $cx 的（右）下半部分
$ch	寄存器 $cx 的（左）上半部分
$dl	寄存器 $dx 的（右）下半部分
$dh	寄存器 $dx 的（左）上半部分
$bl	寄存器 $bx 的（右）下半部分
$bh	寄存器 $bx 的（左）上半部分

下表列出了 80387 部分的寄存器：

寄存器	说明
$fctrl	控制寄存器
$fstat	状态寄存器
$ftag	标记寄存器
$fip	指令指针偏移
$fcs	代码段选择器
$fopoff	操作数指针偏移
$fopsel	操作数指针选择器
$st0 through $st7	数据寄存器

AMD64 寄存器信息

下表列出了适用于 AMD64 体系结构的寄存器信息：

寄存器	说明
rax	通用寄存器 – 为函数调用传递参数
rbp	通用寄存器 – 堆栈管理/帧指针
rbx	通用寄存器 – 被调用方保存
rcx	通用寄存器 – 为函数调用传递参数
rdx	通用寄存器 – 为函数调用传递参数
rsi	通用寄存器 – 为函数调用传递参数
rdi	通用寄存器 – 为函数调用传递参数
rsp	通用寄存器 – 堆栈管理/堆栈指针
r8	通用寄存器 – 为函数调用传递参数
r9	通用寄存器 – 为函数调用传递参数
r10	通用寄存器 – 临时
r11	通用寄存器 – 临时
r12	通用寄存器 – 被调用方保存
r13	通用寄存器 – 被调用方保存
r14	通用寄存器 – 被调用方保存
r15	通用寄存器 – 被调用方保存
rflags	标志寄存器
rip	指令指针
mmx0/st0	64 位媒体和浮点寄存器
mmx1/st1	64 位媒体和浮点寄存器
mmx2/st2	64 位媒体和浮点寄存器
mmx3/st3	64 位媒体和浮点寄存器

寄存器	说明
mmx4/st4	64 位媒体和浮点寄存器
mmx5/st5	64 位媒体和浮点寄存器
mmx6/st6	64 位媒体和浮点寄存器
mmx7/st7	64 位媒体和浮点寄存器
xmm0	128 位媒体寄存器
xmm1	128 位媒体寄存器
xmm2	128 位媒体寄存器
xmm3	128 位媒体寄存器
xmm4	128 位媒体寄存器
xmm5	128 位媒体寄存器
xmm6	128 位媒体寄存器
xmm7	128 位媒体寄存器
xmm8	128 位媒体寄存器
xmm9	128 位媒体寄存器
xmm10	128 位媒体寄存器
xmm11	128 位媒体寄存器
xmm12	128 位媒体寄存器
xmm13	128 位媒体寄存器
xmm14	128 位媒体寄存器
xmm15	128 位媒体寄存器
cs	段寄存器
es	段寄存器
fs	段寄存器
gs	段寄存器
os	段寄存器
ss	段寄存器
fcw	fxsave 和 fxstor 内存映像控制字
fsw	fxsave 和 fxstor 内存映像状态字
fctw	fxsave 和 fxstor 内存映像标记字
fop	fxsave 和 fxstor 内存映像最后一个 x87 op 代码
frdp	fxsave 和 fxstor 内存映像 64 位日期段偏移
frip	fxsave 和 fxstor 内存映像 64 位代码段偏移
mxcsr_mask	mxcsr_mask 中的设置位指示 mxcsr 中支持的特征位
ymmo	256 位高级向量寄存器
ymm1	256 位高级向量寄存器
ymm2	256 位高级向量寄存器
ymm3	256 位高级向量寄存器

寄存器	说明
ymm4	256 位高级向量寄存器
ymm5	256 位高级向量寄存器
ymm6	256 位高级向量寄存器
ymm7	256 位高级向量寄存器
ymm8	256 位高级向量寄存器
ymm9	256 位高级向量寄存器
ymm10	256 位高级向量寄存器
ymm11	256 位高级向量寄存器
ymm12	256 位高级向量寄存器
ymm13	256 位高级向量寄存器
ymm14	256 位高级向量寄存器
ymm15	256 位高级向量寄存器

高级向量 (AVX) 寄存器（ymm0 至 ymm15）的域可被视为具有 C int、float 或 double 类型。

♦ ♦ ♦ 第 18 章

18 将 dbx 与 Korn Shell 配合使用

dbx 命令以 Korn Shell (ksh 88) 的语法为基础，其中包括 I/O 重定向、循环、内置算术、历史记录和命令行编辑。本章列出了 ksh-88 和 dbx 命令语言之间的差别。

如果启动时未找到 dbx 初始化文件，dbx 会采用 ksh 模式。

本章包含以下各节：

ksh-88 功能未实现

以下 ksh-88 功能尚未在 dbx 中实现：

- 用于给数组 *name* 赋值的 `set -A`
- `set -o` 选项：`allexport bgnice gmacs markdirs noclobber nolog privileged protected viraw`
- `typeset -l -u -L -R -H` 属性
- 用于命令替换的反引号 (`\Q…\Q`)（用 `$(...)` 代替）
- 用于表达式求值的 [[*expression*]] 复合命令
- `@(pattern[|pattern] …)` 扩展模式匹配
- 协同进程（与程序通信的在后台运行的命令或管道）

ksh-88 扩展

dbx 添加了以下功能作为扩展：

- `$[ p– > flags ]` 语言表达式
- `typeset -q` 为用户定义的函数启用特殊引用

- 类似 C shell 的 `history` 和 `alias` 参数
- `set +o path` 禁用路径搜索
- 用于八进制和十六进制数的 `0xabcd` C 语法
- `bind` 更改 Emacs 模式绑定
- `set -o hashall`
- `set -o ignore suspend`
- `print -e` 和 `read -e`（与 `-r`、raw 相对）
- 内置 `dbx` 命令

重命名的命令

某些 `dbx` 命令已重命名，以避免与 `ksh` 命令发生冲突。

- `dbx print` 命令保留了名称 `print`；`ksh print` 命令已被重命名为 `kprint`。
- ksh `kill` 命令已与 `dbxkill` 命令合并。
- `alias` 命令是 `ksh alias` 命令，在 `dbx` 兼容模式下除外。
- *address*`/`*format* 现为 `examine` *address*`/`*format*。
- `/`*pattern* 现为 `search` *pattern*。
- `?`*pattern* 现为 `bsearch` *pattern*。

编辑函数的再绑定

使用 `bind` 命令可以重新绑定编辑功能。可以使用命令显示或修改 EMacs 风格编辑器和 vi 风格编辑器的键绑定。`bind` 命令的语法为：

`bind`	显示当前编辑键绑定
`bind` *key*`=`*definition*	将 *key* 绑定到 *definition*
`bind` *key*	显示 *key* 的当前定义
`bind` *key*`=`	删除 *key* 绑定
`bind -m` *key*`=`*definition*	将 *key* 定义为具有 *definition* 的宏
`bind -m`	与 `bind` 相同

其中：

key 为键名。

definition 为要绑定到键的宏的定义。

EMacs 风格编辑器的其中一些更重要的缺省键绑定包括：

^A = 行开始
^B = 后一个字符
^D = 磁带结束符或删除
^E = 行结束
^F = 前一个字符
^G = 终止
^K = 删除到行末
^L = 刷新
^N = 下一个历史命令
^P = 上一个历史命令
^R = 搜索历史命令
^^ = 引号
^? = 向后删除字符
^H = 向后删除字符
^[b = 后退一个字
^[d = 向前删除字
^[f = 向前一个字
^[^H = 向后删除字
^[^[= 完成
^[? = 列出命令

vi 风格编辑器的其中一些更重要的缺省键绑定包括：

a = 追加
A = 行尾追加
c = 更改
d = 删除
G = 行跳转
h = 后一个字符
i = 插入
I = 行首插入
j = 后一行
k = 前一行
l = 行向前
n = 下一个匹配
N = 前一个匹配
p = 后置
p = 前置
r = 重复
R = 替换
s = 代替
u = 撤消
x = 删除字符
X = 删除前一字符
y = yank
~ = 格式调换
_ = 最后参数
* = 展开
= = 列出展开式
- = 前一行
+ = 后一行
sp = 前一字符
# = 注释掉命令
? = 从开始搜索历史命令
/ = 从当前开始搜索历史命令

在插入模式下，下列按键是特殊的：

^? = 删除字符
^H = 删除字符
^U = 中止行
^W = 删除字

♦ ♦ ♦ 第 19 章

19 调试共享库

dbx 为使用动态链接库、共享库的程序提供全面的调试支持，只要这些库是使用 -g 选项编译的。

本章包含以下各节：

动态链接程序

动态链接程序（亦称 rtld、运行时 ld 或 ld.so）安排将共享对象（装入对象）引入到正在执行的程序中。rtld 在两个主要区域处于活动状态：

- 程序启动 – 程序启动时，rtld 先运行，然后动态装入在链接时指定的所有共享对象。它们是预装入的共享对象，可能包括 libc.so、libC.so 或 libX.so。使用 ldd(1) 查明程序将装入哪些共享对象。
- 应用程序请求 – 应用程序使用函数调用 dlopen(3) 和 dlclose(3) 来动态装入和卸下共享对象或可执行文件。

dbx 使用术语装入对象来表示共享对象 (.so) 或可执行文件 (a.out)。可以使用 loadobject 命令列出和管理来自装入对象的符号信息。

链接映射

动态链接程序在称为链接映射的列表中保留有所有装入对象的列表。链接映射保留在正被调试程序的内存中，可通过 librtld_db.so 这一供调试器使用的特殊系统库来间接访问它。

启动序列和 .init 段

.init 段是一段属于共享对象的代码，装入共享对象时该代码将执行。例如，.init 段由 C++ 运行时系统用于调用 .so 文件中的所有静态初始化函数。

动态链接程序会先在所有共享对象中映射，从而将它们置于链接映射中。然后，动态链接程序将遍历链接映射并为每个共享对象执行 .init 段。syncrtld 事件发生在这两个阶段之间。有关更多信息，请参见“syncrtld 事件规范” [256]。

过程链接表

过程链接表 (Procedure Linkage Table, PLT) 是 rtld 为了实现跨共享对象边界调用所使用的结构。例如，对 printf 的调用便会通过这个间接表。有关详细信息，请参见通用的及特定于处理器的 SVR4 ABI 参考手册。

要使 dbx 能够在各 PLT 中处理 step 和 next 命令，它必须记录每个装入对象的 PLT 表。表信息的获得与 rtld 握手同时进行。

在共享库中设置断点

为了在共享库中设置断点，dbx 需要确认程序将在运行时使用该库，并且 dbx 需要为该库装入符号表。为了确定新装入的程序在运行时将使用哪些库，dbx 会将该程序执行足够长的时间，以便运行时链接程序装入所有启动库。然后，dbx 会读入已装入库的列表并终止进程。库会保持装入状态，这样便可以在重新运行程序进行调试前于库中设置断点。

无论程序是在命令行使用 dbx 命令装入，还是在 dbx 提示符处使用 debug 命令装入，抑或在 IDE 中装入，dbx 都会按相同的步骤装入库。

在显式装入的库中设置断点

dbx 会自动检测发生了 dlopen() 还是 dlclose()，然后装入装入对象的符号表。使用 dlopen() 装入共享对象后，即可在其中设置断点，然后像对待程序的任何部分一样进行调试。

如果共享对象是使用 dlclose() 卸下的，dbx 会记住其中设置的断点，并在使用 dlopen() 再次装入该共享对象时替换这些断点，即便应用程序再次运行也是如此。

但如果要在其中设置断点，或导航其函数和源代码，就不必等待使用 dlopen() 装入共享对象。如果知道正被调试的程序将使用 dlopen() 装入的共享对象的名称，可以使用 loadobject -load 命令请求 dbx 预装其符号表：

```
loadobject -load /usr/java1.1/lib/libjava_g.so
```

现在便可在此装入对象被 dlopen() 装入前在其中导航模块和函数及设置断点。装入对象由程序装入后，dbx 即会自动设置断点。

在动态链接库中设置断点受以下限制：

- 使用 dlopen() 装入的过滤器库中的第一个函数被调用后，才能在该库中设置断点。
- dlopen() 装入库后，名为 _init() 的初始化例程便会被调用。此例程可能会调用库中的其他例程。在此初始化完成之前，dbx 不能在已装入的库中设置断点。因此，您无法让 dbx 在 dlopen() 装入的库中的 _init() 处停止。

♦ ♦ ♦ 附 录 A

修改程序状态

本附录重点说明 dbx 的使用方法以及在 dbx 下运行程序时更改程序或更改程序行为的命令，并与不使用 dbx 运行程序进行了比较。了解哪些命令可能修改程序是很有必要的。

本章包含以下各节：

在 dbx 下运行程序的影响

可以使用 dbx 观察进程，并且不会对进程造成影响。但有时可能会彻底修改进程的状态。有时，普通的观察可能会影响执行，并导致出现间歇性错误症状。

在 dbx 下运行时，应用程序的行为方式可能会不同。尽管 dbx 尽量减小它对所调试的程序的影响，但您还是应该注意以下问题：

- 您可能忘记去掉 -C 或禁用 RTC。将 RTC 支持库 librtc.so 装入程序可能会导致程序的行为方式不同。
- dbx 初始化脚本中可能设置了一些您已忘记的环境变量。在 dbx 下运行时，堆栈基址从不同的地址开始。该地址也可能因环境及 argv[] 内容的不同而异，从而强制局部变量以不同方式分配。如果变量未初始化，则将生成不同的随机编号。可以使用运行时检查来检测此问题。
- 程序不会在使用前对通过 malloc（） 分配的内存进行初始化。可以使用运行时检查来检测此问题。
- dbx 必须捕捉 LWP 创建和 dlopen 事件，这可能会影响与计时有关的多线程应用程序。
- dbx 会在收到信号时执行上下文切换，因此，如果应用程序大量使用信号，则系统工作情况可能会有所不同。
- 程序可能期望 mmap（） 始终为映射的段返回相同的基址。在 dbx 下运行会严重影响地址空间，致使 mmap（） 返回的地址不可能与在不使用 dbx 运行该程序时所返回的地址相同。要确定这是否构成问题，请查看 mmap（） 的所有使用情况，确保返回的地址是程序使用的地址，而非固定编码地址。

- 如果程序是多线程程序，则它可能包含数据争用或以其他方式依赖线程调度。在 dbx 下运行会干扰线程调度，并可能导致程序以异常顺序执行线程。要检测此类情况，请使用 lock_lint。

否则，请确定使用 adb 或 truss 运行是否会导致相同的问题。

要尽可能减小 dbx 带来的干扰，请尝试当应用程序在其自然环境中运行时与其连接。

更改程序状态的命令

本节介绍的命令可修改程序。

assign 命令

assign 命令将 *expression* 的值赋给 *variable*。在 dbx 中使用它会永久更改 *variable* 的值。

```
assign variable = expression
```

pop 命令

pop 命令从堆栈中弹出一个或多个帧：

`pop`	弹出当前帧。
`pop` *number*	弹出 *number* 个帧。
`pop -f` *number*	弹出帧，直至帧号达到指定帧 *number*。

任何弹出的调用在恢复后将重新执行，这可能导致不必要的程序更改。pop 还为所弹出函数的本地对象调用析构函数。

有关更多信息，请参见“pop 命令” [328]。

call 命令

在 dbx 中使用 call 命令时，会调用一个过程，而且该过程将按指定方式执行：

```
call proc([params])
```

该过程可能会修改程序中的某些内容。dbx 执行该调用的方式好像您已将其写入程序源代码中一样。

有关更多信息，请参见“call 命令” [274]。

print 命令

要输出表达式的值，请键入：

```
print expression, ...
```

如果表达式含有函数调用，则请输出表达式以执行调用命令。因此，应遵循与使用“call 命令” [274] 命令时相同的注意事项。对于 C++，还应注意由重载运算符导致的意外的副作用。

有关更多信息，请参见“print 命令” [329]。

when 命令

when 命令的通用语法如下所示：

```
when event-specification [modifier] {command; ... }
```

发生事件时，执行上述命令。此操作是否会更改程序的状态取决于发出的命令。

有关更多信息，请参见“when 命令” [367]。

fix 命令

可以使用 fix 命令对程序进行即时更改。

尽管 fix 命令是一个非常有用的工具，但它会重新编译修改过的源文件，并将修改过的函数动态链接到应用程序中。

请注意，修复并继续功能在 Intel Linux 或 SPARC Linux 上不受支持。确保检查对修复并继续功能的其他限制。请参见“内存泄漏 (mel) 错误” [149]。

有关更多信息，请参见“fix 命令” [302]。

cont at 命令

cont at 命令将更改运行程序的顺序。在行 *line* 处继续执行。如果程序是多线程程序，则 ID 是必需的。

`cont at` *line* `[` *ID* `]`

该命令可能会更改程序的结果。

♦ ♦ ♦ 附 录 B

事件管理

事件管理是指 dbx 在所调试的程序中发生事件时执行操作的能力。

本附录包含以下各节：

事件处理程序

事件管理以处理程序的概念为基础。该名称的命名源自硬件中断处理程序。每个事件管理命令通常都会创建一个处理程序，它由事件规范和一系列副作用操作组成。（请参见“设置事件规范” [244]。）事件规范指定将触发处理程序的事件。

发生事件并触发了处理程序时，处理程序会根据事件规范中包括的修饰符来评估事件。（请参见“事件规范修饰符” [256]。）如果事件满足修饰符施加的条件，则将执行处理程序的副作用操作（即处理程序“触发”）。

在特定行设置断点便是一个将程序事件与 dbx 操作关联的示例。

创建处理程序最通用的形式是使用 when 命令。

```
when event-specification {action; ... }
```

本章中的示例将向您显示如何针对 when 编写命令（如 stop、step 或 ignore）。这些示例的目的在于说明 when 命令与底层处理程序机制的灵活性，但实际应用中并不一定要严格遵循这些示例。

创建事件处理程序

使用 when 命令、stop 命令和 trace 命令均可创建事件处理程序。（有关详细信息，请参见“when 命令” [367]、“stop 命令” [344] 和“trace 命令” [355]。）

stop 是常见术语 when 的简略表达方式。

```
when event-specification { stop -update; whereami; }
```

事件规范由事件管理命令 stop、when 和 trace 用于指定相关事件。（请参见“设置事件规范” [244]）。

大多数 trace 命令都可以使用 when 命令、ksh 功能和事件变量来手动创建。 在需要特定格式的跟踪输出时，这特别有用。

每个命令返回一个称为处理程序 ID (*hid*) 的编号。可以使用预定义变量 $newhandlerid 访问此编号。

操作事件处理程序

您可以使用以下命令操作事件处理程序。有关各命令的更多信息，请参见说明的相应一节。

表 2 操作事件处理程序

命令	说明	有关更多信息
status	列出处理程序	请参见“status 命令” [341]
delete	删除所有处理程序，其中包括临时处理程序	请参见“delete 命令” [294]
clear	根据断点位置删除处理程序	请参见“clear 命令” [279]
handler -enable	启用处理程序	请参见“handler 命令” [306]
handler -disable	禁用处理程序	请参见“handler 命令” [306]
cancel	取消信号并启用该流程以继续	请参见“cancel 命令” [275]

使用事件计数器

事件处理程序有一个行程计数器，它具有计数限制。每当发生指定事件时，计数器的计数便会增加。只有当计数达到计数限制时，才会执行与处理程序关联的操作，此时计数器自动重置为 0。缺省限制为 1。每当重新运行进程时，所有事件计数器都会重置。

您可以随 stop 命令、when 命令或 trace 命令使用 -count 修饰符来设置计数限制。另外也可以使用 handler 命令单独操作事件处理程序。

```
handler [ -count | -reset ] hid new-count new-count-limit
```

事件安全

尽管 dbx 向您提供了通过事件机制实现的一组丰富的断点类型，但它还会在内部使用许多事件。通过在发生其中一些内部事件时停止，可以很容易地中止 dbx 的内部操作。如果在这些情况下修改进程状态，中止的可能性会更高。请参见附录 A, 修改程序状态和“调用安全性” [84]。

dbx 可以保护自身在某些情况下（但不是所有情况下）中止操作。一些事件按照低级别事件来实现。例如，所有步进操作都是基于 fault FLTTRACE 事件。因此，发出 stop fault FLTTRACE 命令会中止步进操作。

在以下调试阶段，dbx 不能处理用户事件，因为这些事件会干扰某些需要非常谨慎的内部协调操作。这些阶段包括：

- 程序启动时rtld 运行阶段（请参见“动态链接程序” [233]）
- 进程开始和结束阶段
- 跟随 fork（） 函数和 exec（） 函数（请参见“跟随 fork 函数” [158] 和“跟随 exec 函数” [157]）
- dbx 需要在用户进程中初始化头文件 (proc_heap_init（）) 的调用期间
- dbx 需要确保堆栈上映射页的可用性 (ensure_stack_memory（）) 的调用期间

在许多情况下，可以使用 when 命令而非 stop 命令，以及回显以其他方式交互获得的信息。

dbx 通过以下方式保护自身：

- 禁止对 sync、syncrtld 和 prog_new 事件使用 stop 命令
- 在 rtld 握手期间和上面提到的其他阶段中，忽略 stop 命令

例如：

```
...SolBook linebreakstopped in munmap at 0xff3d503c 0xff3d503c: munmap+0x0004: ta %icc,
0x00000008SolBook linebreak dbx76: warning: 'stop' ignored -- while doing rtld handshake
```

仅忽略中断影响（其中包括 $firedhandlers 变量中的记录）。计数器或过滤器仍发挥作用。要在这种情况下停止，请将环境变量 event_safety 设置为 off。

设置事件规范

stop 命令、stopi 命令、when 命令、wheni 命令、trace 命令和 tracei 命令使用事件规范表示事件类型和参数。此格式由一个表示事件类型和可选参数的关键字组成。对于所有三种命令，事件规范的含义通常相同。在附录 D 的命令说明中介绍了各种异常。

断点事件规范

断点是操作的发生位置，程序在该位置停止执行。本节介绍了断点事件的事件规范。

in 事件规范

in 事件规范的语法如下：

in *function*

函数已输入，即将执行第一行。序进程后的第一个可执行代码用作实际断点位置。这一行可能会初始化局部变量。如果是 C++ 构造函数，在执行了所有基类构造函数后，将停止执行。如果使用 -instr 修饰符，则它是即将执行的第一个函数指令。*function* 规范可以接受形式参数签名，以支持重载的函数名或模板实例规范。例如：

```
stop in mumble(int, float, struct Node *)
```

注 - 不要将 in *function* 与 -in *function* 修饰符混淆。

at 事件规范

at 事件规范的语法如下：

at [*filename:*]*line-number*

指定行即将被执行。如果指定 *filename*，则将执行指定文件中的指定行。文件名可以是源文件名或对象文件名。尽管引号并非必需，但如果文件名中包含特殊字符，则引号必不可少。如果指定行在模板代码中，则会在该模板的所有实例上放置断点。

您还可以使用指定特定地址：

`at` *address-expression*

给定地址处的指令即将被执行。此事件仅可与 `stopi` 命令或 `-instr` 事件修饰符结合使用

`infile` 事件规范

`infile` 事件规范的语法如下：

`infile` *filename*

该事件在文件中定义的每个函数中放置断点。`stop infile` 命令循环访问与 `funcs -f` *filename* 命令相同的函数列表。

`.h` 文件中的方法定义、模板文件或 `.h` 文件中的纯 C 代码（如 `regexp` 命令使用的那种）可能会为文件提供函数定义，但这些定义会被排除。

如果指定的文件名是对象文件的名称（即，该文件名以 `.o` 结尾），则会在出现在该对象文件中的每个函数上放置断点。

`stop infile list.h` 命令不会在 `list.h` 文件中定义的所有方法实例中放置断点。请使用诸如 `inclass` 或 `inmethod` 之类的事件来执行此操作。

`fix` 命令可以删除文件中的函数或向文件中添加函数。`stop infile` 命令可在文件中的所有旧版本函数中，以及未来可能添加的任何函数中放置断点。

不会在 Fortran 文件的嵌套函数或子例程中放置断点。

您可以使用 `clear` 命令禁用 `infile` 事件创建的集合中的单个断点。

`infunction` 事件规范

`infunction` 事件规范的语法如下：

`infunction` *function*

对于所有名为 *function* 的重载函数或其所有模板实例，此规范等效于 `in` *function*。

`inmember` 事件规范

`inmember` 事件规范的语法如下：

`inmember` *function*

此规范是 inmethod 事件规范的别名。

inmethod 事件规范

inmethod 事件规范的语法如下：

inmethod *function*

此规范等效于 in *function* 或每个类命名为 *function* 的成员方法。

inclass 事件规范

inclass 事件规范的语法如下：

inclass *classname* [-recurse | -norecurse]

对于所有属于 *classname* 成员、但并非任何 *classname* 基础的成员函数，此规范等效于 in *function*。-norecurse 为缺省值。如果指定 -recurse，则包括基类。

inobject 事件规范

inobject 事件规范的语法如下：

inobject *object-expression* [-recurse | -norecurse]

已调用针对 *object-expression* 表示的地址中的特定对象所调用的成员函数。stop inobject *ox* 大致等效于以下内容，但与 inclass 不同，它包括 *ox* 的动态类型基础。-recurse 为缺省值。如果指定 -norecurse，则不包括基类。

stop inclass dynamic_type(*ox*) -if this==*ox*

数据更改事件规范

本节介绍了涉及访问或更改内存地址内容的事件的事件规范。

access 事件规范

access 事件规范的语法如下：

access *mode address-expression* [,*byte-size-expression*]

访问了由 *address-expression* 指定的内存。

mode 指定内存访问模式。有效值为以下一个字母或全部字母：

`r` 已读取指定地址处的内存。

`w` 已写入内存。

`x` 已执行内存。

mode 还可以包含以下任一项：

`a` 访问后停止进程（缺省值）。

`b` 访问前停止进程。

在这两种情况下，程序计数器都将指向违例指令。“之前”和“之后”都具有副作用。

address-expression 是求值结果为地址的任何表达式。如果提供符号表达式，则会自动推导出要监视的区域大小。可以通过指定 *byte-size-expression* 覆盖它。也可以使用非符号、无类型地址表达式，在这种情况下，必须提供大小。例如：

```
stop access w 0x5678, sizeof(Complex)
```

`access` 命令具有限制，即：两个匹配区域不可重叠。

注 - `access` 事件规范取代了 `modify` 事件规范

`change` 事件规范

`change` 事件规范的语法如下：

`change` *variable*

variable 的值已更改。`change` 事件大致等效于：

```
when step { if [ $last_value !=$[variable]]
            then
                 stop
            else
                 last_value=$[variable]
            fi
          }
```

此事件使用单步执行来执行。要提高速度，可使用 `access` 事件。

即使未检测到更改，第一次检查 *variable* 时也会触发一个事件。这第一个事件是访问 *variable* 的初始值。以后检测到 *variable* 值的更改时会触发其他事件。

cond 事件规范

cond 事件规范的语法如下：

```
cond condition-expression
```

由 *condition-expression* 表示的条件的求值结果为 true。可以为 *condition-expression* 指定任何表达式，但其求值结果必须为整型。cond 事件大致等同于以下 stop 命令：

```
stop step -if conditional-expression
```

系统事件规范

本节介绍了系统事件的事件规范。

dlopen 和 dlclose 事件规范

dlopen（） 和 dlopen（） 事件规范的语法如下：

```
dlopen [ lib-path ]

dlclose [ lib-path ]
```

成功调用 dlopen（） 或 dlclose（） 之后，会发生系统事件。调用 dlopen（） 或调用 dlclose（） 可能导致装入多个库。这些库列表始终可用于预定义的变量 $dllist 中。$dllist 中的第一个 Shell 字是 +（加号）或 -（减号），表示是在添加、还是删除库列表。

lib-path 是共享库的名称。如果指定该名称，则只在装入或卸载了给定库时才会发生事件。在这种情况下，$dlobj 包含库的名称。$dllist 仍然可用。

如果 *lib-path* 以 / 开头，则将执行完整字符串匹配。否则，将只比较路径的尾部。

如果未指定 *lib-path*，则只要出现 dl 活动，都会发生事件。在这种情况下，$dlobj 为空，但 $dllist 有效。

fault 事件规范

fault 事件规范的语法如下：

```
fault fault
```

遇到指定的故障时，会发生 fault 事件。这些错误为体系结构相关式错误。dbx 的已知故障集在以下列表中列出，且在 proc(4) 手册页中已定义。

FLTILL	非法指令
FLTPRIV	特权指令
FLTBPT*	断点陷阱
FLTTRACE*	跟踪陷阱（单步）
FLTACCESS	内存访问（如对齐）
FLTBOUNDS	内存边界（无效地址）
FLTIOVF	整数溢出
FLTIZDIV	整数除以零
FLTPE	浮点异常
FLTSTACK	无法恢复的堆栈错误
FLTPAGE	可恢复的缺页
FLTWATCH*	监视点陷阱
FLTCPCOVF	CPU 性能计数器溢出

注 - FLTBPT、FLTTRACE 和 FLTWATCH 未处理，因为它们已由 dbx 用来实现断点、单步步入和监视点。

上述错误摘自 /usr/include/sys/fault.h。*fault* 可以是上面所列错误中的任何一种（大小写、有无 FLT 前缀均可），也可以是实际的数字代码。

注 - 在 Linux 平台上不能使用 fault 事件。

lwp_exit 事件规范

lwp_exit 事件规范的语法如下：

```
lwp_exit
```

退出 lwp 时，会发生 lwp_exit 事件。对于事件处理程序的持续时间，$lwp 包含退出的 LWP（轻量级进程）的 ID。

注 - Linux 平台上没有 lwp_exit 事件。

sig 事件规范

sig 事件规范的语法如下：

sig *signal*

信号第一次传递给调试的程序时，会发生 sig *signal* 事件。*signal* 既可以是十进制数，也可以是大写或小写的信号名称。前缀是可选的。尽管 catch 命令可按如下所示实现，但此事件完全独立于 catch 命令和 ignore 命令：

```
function simple_catch {
    when sig $1 {
            stop;
            echo Stopped due to $sigstr $sig
            whereami
    }
}
```

注 - 收到 sig 事件时，进程尚未检测到它。只有在使用指定的信号继续执行进程时，信号才会传送给它。

或者，您可以使用子代码指定信号。sig 事件规范的此选项的语法如下：

sig *signal sub-code*

首次将具有指定 *sub-code* 的指定信号传送到子进程时，会发生 sig *signal sub-code* 事件。对于信号，您可以十进制数、大写或小写字母的形式提供 *sub-code*。前缀是可选的。

sysin 事件规范

sysin 事件规范的语法如下：

sysin *code*|*name*

刚启动了指定的系统调用，且进程已进入内核模式。

dbx 支持的系统调用概念，是指由进入内核（/usr/include/sys/syscall.h 中所枚举）中的陷阱所提供的概念。

此概念不同于系统调用的 ABI 概念。一些 ABI 系统调用在用户模式下得到部分实现，并且使用非 ABI 内核陷阱。但是，对于大多数普通系统调用（主要异常是信号处理），syscall.h 和 ABI 之间没有区别。

注 - 在 Linux 平台上不能使用 sysin 事件。

/usr/include/sys/syscall.h 中的内核系统调用陷阱列表是 Oracle Solaris OS 中的专用接口的一部分，它因版本而异。dbx 接受的陷阱名（代码）和陷阱编号列表包括 dbx 支持的任何 Oracle Solaris OS 版本支持的所有陷阱名（代码）和陷阱编号。dbx 支持的名称不可能与任何特定版本 Oracle Solaris OS 的名称完全匹配，且 syscall.h 中的某些名称可能不可用。任何陷阱编号（代码）均可为 dbx 接受，并可正常使用，但是，如果它与已知的系统调用陷阱不对应，系统会发出警告。

sysout 事件规范

sysout 事件规范的语法如下：

sysout *code*|*name*

已完成指定的系统调用，进程即将返回到用户模式。

注 - 在 Linux 平台上不能使用 sysout 事件。

---

sysin | sysout 事件规范

如果不使用参数，所有系统调用都会被跟踪。某些 dbx 功能（例如 modify 事件和运行时检查）会导致子进程为其自身目的执行系统调用，并且在被跟踪时显示出来。

执行进度事件规范

本节介绍了有关于执行进度的事件的事件规范。

exit 事件规范

exit 事件规范的语法如下：

exit *exitcode*

进程退出时，会发生 exit 事件。

next 事件规范

next 事件类似于 step 事件，除了函数尚未步入以外。

returns 事件规范

returns 事件是当前已访问的函数返回点中的断点。使用访问的函数是为了可以在提供大量 step up 命令后使用 returns 事件规范。returns 事件始终为 -temp，并且只有当存在活动进程时才能创建该事件。

returns 事件规范的语法如下：

```
returns function
```

每当给定的函数返回到其调用点时，都会执行 returns *function* 事件。这不是临时事件。执行此操作时，并不提供返回值，但可以访问以下寄存器查找整型返回值：

- 基于 SPARC 的系统 – $o0
- 基于 x86 的系统 – $eax
- 基于 x64 的系统 – $rax、$rdx

该事件大致等效于：

```
when in func { stop returns; }
```

step 事件规范

执行源代码行的第一个指令时，会发生 step 事件。例如，您可使用以下命令进行简单跟踪：

```
when step { echo $lineno: $line; }; cont
```

启用了 step 事件后，即指示 dbx 在下次使用 cont 命令时自动单步执行。

注 - step 命令终止时，不会发生 step（和 next）事件。step 命令针对 step 事件大致按如下所示实现：alias step="when step -temp { whereami; stop; }; cont"

---

throw 事件规范

throw 事件的语法如下：

```
throw [type | -unhandled | -unexpected]
```

每当应用程序抛出任何未处理的或意外的异常时，均会发生 throw 事件。

如果异常类型使用 throw 事件指定，则只有该类型的异常才会导致 throw 事件的发生。

如果指定了 -unhandled 选项，则对于没有处理程序的情况，系统将抛出特殊异常类型，表明出现异常。

指定了 -unexpected 选项，该选项是一种特殊的异常类型，表示异常不符合抛出它的函数的异常规范。

跟踪线程事件规范

下面这一节介绍跟踪线程的事件规范。

omp_barrier 事件规范

当跟踪线程进入或退出屏障时，会涉及 omp_barrier 事件规范。您可以指定 *type*（它可以是 explicit 或 implicit）和 *state*（它可以是 enter、exit 或 all_entered）。缺省值为 explicit all_entered。

omp_taskwait 事件规范

当跟踪线程进入或退出任务等待时，会涉及 omp_taskwait 事件规范。您可以指定 *state*（它可以是 enter 或 exit）。缺省值为 exit。

omp_ordered 事件规范

当跟踪线程进入或退出有序区域时，会涉及 omp_ordered 事件规范。您可以指定 *state*（它可以是 begin、enter 或 exit）。缺省值为 enter。

omp_critical 事件规范

当跟踪线程进入重要区域时，会涉及 omp_critical 事件规范。

omp_atomic 事件规范

当跟踪线程进入或退出原子区域时，会涉及 omp_atomic 事件规范。您可以指定 *state*（它可以是 begin 或 exit）。缺省值为 begin。

omp_flush 事件规范

当跟踪线程进入显式刷新区域时，会涉及 omp_flush 事件规范。

omp_task 事件规范

当跟踪线程进入或退出任务区域时，会涉及 omp_task 事件规范。您可以指定 *state*（它可以是 create、start 或 finish）。缺省值为 start。

omp_master 事件规范

当跟踪线程进入主要区域时，会涉及 omp_master 事件规范。

omp_single 事件规范

当跟踪线程进入单个区域时，会涉及 omp_single 事件规范。

其他事件规范

本节介绍了其他类型事件的事件规范。

attach 事件规范

attach 事件是 dbx 成功连接到进程。

detach 事件规范

detach 事件是 dbx 成功与所调试的程序分离。

lastrites 事件规范

lastrites 事件是所调试的进程即将过期，原因可能如下：

- _exit(2) 系统调用已通过显式调用发生，或在 main（） 返回时发生。

- 即将传送终止信号。
- 进程正由 kill 命令中止。

触发此事件时，通常（但并非总是）会有进程的最终状态，从而提供了检查进程状态的最后机会。在此事件后恢复执行会终止进程。

注 - 在 Linux 平台上不能使用 lastrites 事件。

proc_gone 事件规范

当 dbx 不再与调试进程关联时，会发生 proc_gone 事件。预定义变量 $reason 可以是 signal、exit、kill 或 detach。

prog_new 事件规范

当新程序作为 follow exec 的结果装入时，会发生 prog_new 事件。

注 - 此事件的处理程序始终永久性存在。

stop 事件规范

每当进程停止以便用户收到提示（尤其是为了响应 stop 处理程序）时，都会发生 stop 事件。例如，以下命令是等效的：

```
display x
when stop {print x;}
```

sync 事件规范

正在调试的进程正好已使用 exec（） 执行时，会发生 sync 事件。在 a.out 中指定的所有内存均有效且存在，但预装入的共享库尚未装入。例如，尽管 printf 可供 dbx 使用，但尚未映射到内存中。

对此事件执行 stop 不起作用，但可以将 sync 事件与 when 命令一起使用。

注 - 在 Linux 平台上不能使用 sync 事件。

`syncrtld` 事件规范

如果正在调试的进程尚未处理共享库，则在 `sync` 或 `attach` 之后会发生 `syncrtld` 事件。它在动态链接程序启动代码已经执行且所有预装入共享库的符号表都已装入之后、`.init` 段中的任何代码运行之前执行。

针对此事件执行 `stop` 不起作用，但可以将 `syncrtld` 事件与 `when` 命令一起使用。

`thr_create [`*thread-ID*`]` 事件规范

创建线程或具有指定线程 ID 的线程时，会发生 `thr_create` 事件。例如，在以下 `stop` 命令中，线程 ID `t@1` 是指执行创建线程，而线程 ID `t@5` 是指被创建线程。

```
stop thr_create t@5 -thread t@1
```

`thr_exit` 事件规范

线程退出时，会发生 `thr_exit` 事件。要捕获特定线程的退出，请使用 `stop` 命令的 `-thread` 选项，如下所示：

```
stop thr_exit -thread t@5
```

`timer` 事件规范

`timer` 事件的语法如下：

`timer` *seconds*

正在调试的程序已针对 *seconds* 运行时，会发生 `timer` 事件。用于此事件的计时器与 `collector` 命令共享。精度为毫秒，因此，*seconds* 可以为浮点值（例如 `0.001`）。

事件规范修饰符

事件规范修饰符设置处理程序的其他属性，最常见的一种是事件过滤器。修饰符必须位于事件规范的关键字部分之后。修饰符以短划线 (`-`) 开头。以下是有效的事件规范修饰符。

`-if` 修饰符

`-if` 修饰符的语法为：

`-if` *condition*

发生事件规范指定的事件时，便会对条件进行求值。仅当条件的求值结果为非零时，处理程序才会出现副作用。

如果 `-if` 修饰符用于具有关联的单一源位置（如 `in` 或 `at`）的事件，则在与该位置对应的作用域对 *condition* 进行求值。否则，请使用所需作用域来限定它。

根据与 `print` 命令相同的约定对条件执行宏扩展。

`-resumeone` 修饰符

`-resumeone` 修饰符可以与 `-if` 修饰符一起用于多线程程序的事件规范中，这样，条件中包含函数调用时，只恢复一个线程。有关更多信息，请参见“使用条件过滤器限定断点” [94]。

`-in` 修饰符

`-in` 修饰符的语法为：

`-in` *function*

仅当事件在达到给定函数的第一个指令之后、函数返回之前的这段时间发生时，才会触发事件。函数的递归会被忽略。

`-disable` 修饰符

`-disable` 修饰符用于创建处于禁用状态的处理程序。

`-count` *n*、`-count infinity` 修饰符

`-count` 修饰符的语法为：

`-count` *n*

或

`-count infinity`

`-count` *n* 和 `-count infinity` 修饰符用于使处理程序从 0 开始计数（请参见“使用事件计数器” [243]）。每次发生事件时，计数便会增加，直至达到 *n*。达到 n 后，处理程序便会启动，且计数器重置为零。

程序运行或重新运行时，所有启用的处理程序的计数都会被重置。更确切地讲，就是在发生 sync 事件时重置计数。

在使用 debug -r 命令（请参见“debug 命令” [291]）或 attach -r 命令（请参见“attach 命令” [273]）开始调试新程序时会重置计数。

-temp 修饰符

-temp 修饰符创建临时处理程序。发生事件后，会立即自动将其删除。缺省情况下，处理程序不是临时性的。如果处理程序是计数处理程序，则只有在计数达到 0（零）时，才会自动将其删除。

可使用 delete -temp 命令删除所有临时处理程序。

-instr 修饰符

-instr 修饰符使处理程序在指令级操作。此事件取代了大多数命令的传统后缀 "i"。它通常修改事件处理程序的下列两个方面：

- 任何消息均输出汇编级信息，而非源代码级信息。
- 事件的粒度变为指令级。例如，step -instr 意味着指令级单步执行。

-thread 修饰符

-thread 修饰符的语法为：

-thread *thread-ID*

-thread 修饰符意味着只有在导致事件的线程与另一个线程 ID 相符时，才会执行该操作。对于您考虑的特定线程，在程序的不同次执行中可能分配不同的线程 ID。

-lwp 修饰符

-lwp 修饰符的语法为：

-lwp *lwp-ID*

-lwp 修饰符意味着只有在导致事件的 -lwp 与 *lwp-ID* 相符时，才会执行该操作。对于您考虑的特定 -lwp，在程序的不同次执行中可能分配有不同的 *lwp-ID*。

-hidden 修饰符

-hidden 修饰符在常规 status 命令中隐藏处理程序。可使用 status -h 查看隐藏的处理程序。

-perm 修饰符

正常情况下，装入新程序时，所有处理程序都会被抛弃。使用 -perm 修饰符可在多个调试会话中保留处理程序。不带参数的 delete 命令不会删除永久性处理程序。可使用 delete -p 删除永久性处理程序。

解析和二义性

事件规范和修饰符的语法由关键字驱动，且以 ksh 约定为基础。所有内容均拆分为用空格分隔的单词。

表达式可能会有内嵌空格，这便会导致不明确情况发生。例如，假设有以下两个命令：

```
when a -temp
when a-temp
```

在第一个示例中，尽管应用程序可能有一个名为 *temp* 的变量，但 dbx 解析器仍会优先将 -temp 作为修饰符来解析事件规范。在第二个示例中，a-temp 作为整体传递给语言特定的表达式解析器。如果不存在名为 *a* 和 *temp* 的变量，则会发生错误。可使用括号强制解析。

使用预定义变量

有一些只读的 ksh 预定义变量。下表中列出的变量始终有效。

变量	定义
$ins	反汇编当前指令。
$lineno	以十进制数表示的当前行号。
$vlineno	以十进制数表示的当前“访问”行号。
$line	当前行的内容。
$func	当前函数名。
$vfunc	当前“访问”函数的名称。
$class	$func 所属类的名称。

变量	定义
$vclass	$vfunc 所属类的名称。
$file	当前文件名。
$vfile	被访问的当前文件的名称。
$loadobj	当前可装入对象的名称。
$vloadobj	被访问的当前可装入对象的名称。
$scope	当前 PC 在反引用符号中的作用域。
$vscope	被访问 PC 在反引用符号中的作用域。
$funcaddr	以十六进制数表示的 $func 地址。
$caller	调用 $func 的函数的名称。
$dllist	在发生 dlopen 或 dlclose 事件后，包含刚装入或卸载的装入对象的列表。dllist 的第一个字是 + （加号）或 - （减号），具体取决于是否已发生 dlopen 或 dlclose。
$newhandlerid	最近创建的处理程序的 ID。在任何删除处理程序的命令后，此变量有一个未定义的值。创建处理程序后立即使用该变量。dbx 不能针对创建多个处理程序的命令捕获所有处理程序 ID。
$firedhandlers	导致最近中断的处理程序 ID 列表。在 status 命令的输出中，该列表中的处理程序都标记有 *（星号）。
$proc	正被调试的当前进程的“进程 ID”。
$lwp	当前 LWP 的 ID。
$thread	当前线程的“线程 ID”。
$newlwp	新建 LWP 的 ID。
$newthread	新建线程的 ID。
$prog	正被调试程序的全路径名。
$oprog	$prog 的上一个值，当程序的全路径名恢复为 - （短划线）时，使用此值返回您在 exec() 之后调试的内容。$prog 扩展为全路径名时，$oprog 包含命令行中指定的程序路径或指定给 debug 命令的程序路径。如果调用 exec() 多次，则无法返回到原始程序。
$exec32	如果 dbx 二进制数为 32 位，则为 True。
$exitcode	最后运行程序的退出状态。如果进程尚未退出，则该值为空字符串。
$booting	如果在引导过程中发生事件，则会设置为 true。每当调试新程序时，它会先引导以便可以确定共享库的列表和位置。然后，进程将被中止。这一序列被称为“引导”。 引导过程中，所有事件仍然可用。例如，使用此变量来区分在调试运行期间发生的 sync 和 syncrtld 事件以及在正常运行期间发生的这些事件。
$machtype	如果已装入程序，则返回计算机类型：sparcv8、sparcv8+、sparcv9、x86 或 x86_64。否则，返回 unknown。

$datamodel	如果已装入程序，则返回数据模型：ilp32 或 lp64。否则，返回 unknown。要查找刚才装入的程序模型，请使用 .dbxrc 文件中的以下内容：
	
	`when prog_new -perm {`
	`  echo machine: $machtype $datamodel;`
	`}`

以下示例说明可以实现 whereami：

```
function whereami {
  echo Stopped in $func at line $lineno in file $(basename $file)
  echo "$lineno\t$line"
}
```

对 when 命令有效的变量

本节介绍的变量仅在 when 命令主体内有效。

$handlerid

在执行主体过程中，$handlerid 是主体所属 when 命令的 ID。以下命令是等效的：

```
when X -temp { do_stuff; }
when X  { do_stuff; delete $handlerid; }
```

对 when 命令和特定事件有效的变量

某些变量仅在 when 命令主体中以及对于下表中所示的特定事件有效。

表 3 对 sig 事件有效的变量

变量	说明
$sig	触发事件的信号数
$sigstr	$sig 的名称
$sigcode	$sig 的子代码（如果适用）
$sigcodestr	$sigcode 的名称
$sigsender	信号发出方的进程 ID（如果适用）

表 4 对 exit 事件有效的变量

变量	说明
$exitcode	传递给 _exit(2) 或 exit(3) 的参数的值或 main 的返回值

表 5 对 dlopen 和 dlclose 事件有效的变量

变量	说明
$dlobj	装入对象 dlopened 或 dlclosed 的路径名

表 6 对 sysin 和 sysout 事件有效的变量

变量	说明
$syscode	系统调用号
$sysname	系统调用名

表 7 对 proc_gone 事件有效的变量

变量	说明
$reason	signal、exit、kill 或 detach 之一

表 8 对 thr_create 事件有效的变量

变量	说明
$newthread	新建线程的 ID（例如 t@5）
$newlwp	新建 LWP 的 ID（例如 l@4）

表 9 对 access 事件有效的变量

变量	说明
$watchaddr	正在写入、读取或执行的地址
$watchmode	以下项之一：r 代表读取，w 代表写入，x 代表执行；后跟以下项之一：a 代表之后，b 代表之前

事件处理程序示例

本节提供了设置事件处理程序的一些示例。

为存储到数组成员设置断点

本示例显示如何在 array[99] 中设置数据更改断点：

```
(dbx) stop access w &array[99]
(2) stop access w &array[99], 4
(dbx) run
Running: watch.x2
watchpoint array[99] (0x2ca88[4]) at line 22 in file "watch.c"
   22    array[i] = i;
```

执行简单跟踪

本示例显示如何实现简单跟踪：

```
(dbx) when step { echo at line $lineno; }
```

在函数内时启用处理程序

以下示例显示如何在函数内时启用处理程序：

```
<dbx> trace step -in foo
```

此命令等效于下面的命令：

```
    # create handler in disabled state
    when step -disable { echo Stepped to $line; }
    t=$newhandlerid    # remember handler id
    when in foo {
    # when entered foo enable the trace
    handler -enable "$t"
    # arrange so that upon returning from foo,
    # the trace is disabled.
    when returns { handler -disable "$t"; };
    }
```

确定已执行的行数

本示例显示如何查看小程序中已执行多少行，请键入：

```
(dbx) stop step -count infinity     # step and stop when count=inf
(2) stop step -count 0/infinity
(dbx) run
...
(dbx) status
(2) stop step -count 133/infinity
```

程序永远不会停止，程序只会终止。执行的行数为 133。此进程速度非常慢。它对多次调用的函数中的断点用处最大。

确定源代码行执行的指令数

本示例显示如何计算一行代码执行多少个指令：

```
(dbx) ...                       # get to the line in question
(dbx) stop step -instr -count infinity
(dbx) step ...
(dbx) status
(3) stop step -count 48/infinity # 48 instructions were executed
```

如果步过的行进行函数调用，则该函数中的行也计入在内。可以使用 `next` 事件而非 `step` 计算指令数（不包括被调用函数）。

事件发生后启用断点

只在另一事件发生后启用断点。例如，如果程序在 `hash` 函数中开始执行出错，则您可使用以下断点（但必须在完成 1300 次符号查找之后）。

```
(dbx) when in lookup -count 1300 {
    stop in hash
    hash_bpt=$newhandlerid
    when proc_gone -temp { delete $hash_bpt; }
}
```

注 - `$newhandlerid` 是指刚执行的 `stop in` 命令。

为重放重置应用程序文件

在本例中，如果应用程序处理需要在 `replay` 期间重置的文件，可以编写一个处理程序，在每次运行程序时执行该操作。

```
(dbx) when sync { sh regen ./database; }
(dbx) run < ./database...    # during which database gets clobbered
(dbx) save
...             # implies a RUN, which implies the SYNC event which
(dbx) restore       # causes regen to run
```

检查程序状态

本示例显示如何快速查看程序运行时的位置，请键入：

```
(dbx) ignore sigint
(dbx) when sig sigint { where; cancel; }
```

然后，键入 `^C` 即可在不停止程序的情况下查看程序的堆栈跟踪。

该示例基本上是收集器人工示例模式的任务（还有更多）。可使用 SIGQUIT (^\) 在 ^C 已用尽的情况下中断程序。

捕获浮点异常

本示例显示如何仅捕获特定的浮点异常（如 IEEE 下溢）：

```
(dbx) ignore FPE              # disable default handler
(dbx) help signals | grep FPE  # can't remember the subcode name
...
(dbx) stop sig fpe FPE_FLTUND
...
```

有关启用 ieee 处理程序的更多信息，请参见“捕获 FPE 信号（仅限 Oracle Solaris）” [169]。

♦ ♦ ♦ 附录 C

宏

缺省情况下，选定的表达式在求值前扩展宏，包括使用以下项指定的表达式：print、display 和 watch 命令，stop、trace 和 when 命令的 -if 选项，以及 $[] 构造。宏扩展还适用于 IDE 或 dbxtool 中的气球表达式求值和监视。

宏扩展的其他用途

宏扩展适用于 assign 命令中的变量和表达式。

在 call 命令中，宏扩展适用于正被调用的函数的名称以及正在传递的参数。

macro 命令可以采用任意表达式和宏，并可以扩展宏。例如：

```
(dbx) macro D(1, 2)
    Expansion of: D(1, 2)
              is: d(1,2)
```

如果赋予 whatis 命令一个宏，则它显示宏的定义。例如：

```
(dbx) whatis B
    #define B(x) b(x)
```

如果赋予 which 命令一个宏，则它显示作用域中当前活动的宏在何处定义。例如：

```
(dbx) which B2
    `a.out`macro_wh.c`B2    # defined at defs2.h:3
            # included from defs1.h:3
            # included from macro_wh.c:23
```

如果赋予 whereis 命令一个宏，则它显示已定义宏的所有位置。该列表只限于 dbx 已读取调试信息的模块。例如：

```
(dbx) whereis U
    macro:          U       # defined at macro_wh.c:21
    macro:          U       # undefined at defs1.h:5
```

dbxenv 变量 macro_expand 控制这些命令是否扩展宏。缺省情况下，它设置为 on。

一般情况下，dbx 命令中的 +m 选项会导致命令跳过宏扩展。-m 选项强制进行宏扩展，即使 dbxenv 变量 macro_expand 设置为 off 也是如此。$[] 构造中的 -m 选项是一个例外，其中 -m 仅使宏被扩展，不进行求值。该例外有助于 shell 脚本中的宏扩展。

宏定义

dbx 可以采用两种方法识别宏定义：

- 如果使用缺省 DWARF 格式调试信息，在以 -g3 选项进行编译时，编译器会提供定义。如果编译时指定 -xdebugformat=stabs 选项，则不提供定义。
- dbx 可以通过略读源文件及其包含文件来重新创建定义。准确的重新创建取决于对原始源文件及包含文件的访问情况。这还取决于所使用编译器的路径名的可用性以及编译器选项（如 -D 和 -I）。可以从 Oracle Developer Studio 编译器（但不能从 GNU 编译器）以 DWARF 和 stabs 格式提供该信息。有关确保成功略读的信息，请参见“略读 (skimming) 错误” [269] 和“使用 pathmap 命令改进略读 (skimming)” [270]。

dbxenv 变量macro_source（请参见第 3 章 定制 dbx中的表 1 “dbx 环境变量”）可控制 dbx 使用两种方法中的哪一种来识别宏定义。

在选择希望 dbx 使用的方法时，需要考虑多种因素。

编译器和编译器选项

选择宏定义方法时要考虑的一个因素是不同类型信息的可用性，这些信息取决于用于生成代码的编译器和编译器选项。下表显示了可根据编译器和调试信息选项选择的方法。

表 10　可用于各种生成选项的宏定义方法

编译器	-g 选项	调试信息格式	起作用的方法
Oracle Developer Studio	-g	DWARF	略读
Oracle Developer Studio	-g	stabs	略读
Oracle Developer Studio	-g3	DWARF	略读和通过编译器
Oracle Developer Studio	-g3	stabs	略读（不支持 -g3 选项与 -xdebugformat=stabs 选项）
GNU	-g	DWARF	两者均无
GNU	-g	stabs	N/A
GNU	-g3	DWARF	通过编译器
GNU	-g3	stabs	N/A

功能方面的权衡

选择宏定义方法时另一个要考虑的因素是根据选择的方法在功能方面进行权衡：

- **可执行文件的大小**。略读方法的主要优点是不需要使用 -g3 选项进行编译，因为该方法适用于使用 -g 选项编译而生成的较小可执行文件。
- **调试格式**。略读适用于 DWARF 和 stabs。使用 -g3 选项进行编译以通过编译器获得定义仅适用于 DWARF。
- **速度**。第一次针对 dbx 尚未读取调试信息的模块对表达式求值时，略读最多需要一秒钟。
- **精确度**。使用 -g3 选项进行编译时，编译器提供的信息比略读提供的信息更稳定、更准确。
- **生成环境的可用性**。略读要求编译器、源代码文件以及包含文件在调试过程中可用。dbx 不会检查这些过期的项，因此，如果这些项有可能发生更改，准确性可能会下降，所以使用 -g3 选项进行编译可能好于依靠略读。
- **在与编译代码的系统不同的系统上进行调试**。如果在系统 A 上编译代码而在系统 B 上进行调试，则 dbx 使用 NFS 并借助 pathmap 命令访问系统 A 上的文件。

  pathmap 命令还有助于在略读期间访问文件。尽管，它适用于程序的源文件和包含文件，但它可能不适用于系统包含文件，这是因为 /usr/include 通常不通过 NFS 来提供。因此，宏定义将从调试系统（而不是生成系统）的 /usr/include 中提取。

  您可以选择了解并容许系统包含文件之间可能存在的差异，或者选择使用 -g3 选项进行编译。

限制

- 虽然 Fortran 编译器通过 cpp(1) 函数或 fpp(1) 函数支持宏，但 dbx 不支持 Fortran 的宏扩展。
- dbx 会忽略使用 -g3 选项和 -xdebugformat=stabs 选项进行编译而生成的宏信息。

  有关 stabs 索引的更多信息，请参见 Stab 接口指南（可在 *install-dir*/solarisstudio12.4/READMEs/stabs.pdf 路径中找到）。
- 略读适用于使用 -g 选项和 -xdebugformat=stabs 选项编译的代码。

略读 (skimming) 错误

如果不使用 -g3 选项编译代码，而且将 macro_source dbxenv 变量设置为 skim_unless_compiler 或 skim，则您将依赖宏略读。

若要针对模块成功进行略读，需要满足以下条件：

- 必须使用 -g 选项通过 Oracle Developer Studio 编译器编译了模块。
- 用于编译模块的编译器必须可由 dbx 访问。
- 模块的源文件必须可由 dbx 访问。
- 模块源代码所含的文件必须可用，即，编译模块时指定给 -I 选项的路径必须可由 dbx 访问。
- 源代码的语法必须合理。例如，代码的注释不能包含未结束的字符串，或是代码不能缺少 #endif。

如果源代码或包含文件不可由 dbx 访问，可以使用 pathmap 命令使其可以访问。

使用 pathmap 命令改进略读 (skimming)

如果在编译后移动源文件，或是在一台计算机上生成而在另一台计算机上进行调试，或是发生“查找源文件和对象文件” [77]中所述的其他情况之一，则宏略读可能无法在其略读的文件中找到包含文件。与无法找到文件的其他情况一样，解决方案是使用 pathmap 命令帮助宏略读器定位包含目录。例如，假设使用选项 -I/export/home/proj1/include 进行编译，而且代码中包含语句 #include "module1/api.h"。然后，如果将 proj1 重命名为 proj2，则以下 pathmap 命令将帮助宏略读器定位文件：

```
pathmap /export/home/proj1 /export/home/proj2
```

pathmap 不适用于用来编译原始代码的编译器。

处理宏时，必须重新装入应用程序，以使路径映射生效，与找不到文件时的其他情况不同，可以使用 pathmap 命令在路径映射过程中进行更改，这些更改会立即生效。

在一台计算机上生成而在另一台计算机上进行调试时，pathmap 命令可帮助 dbx 找到正确的文件。然而，系统包含文件（如 /usr/include/stdio.h）通常不会从生成计算机中导出，因此宏略读器可能会使用调试计算机中的文件。在某些情况下，系统包含文件可能在调试计算机上不可用。而且，特定于系统的宏和与系统相关的宏的值在调试计算机和生成计算机中也可能不相同。

如果 pathmap 命令不能解决略读问题，则考虑使用 -g3 选项编译代码，并将 macro_source dbxenv 变量设置为 skim_unless_compiler 或 compiler。

♦ ♦ ♦ 附 录 D

命令参考

本附录提供了所有 `dbx` 命令的详细语法和功能说明。

`adi assign` 命令

`adi assign` 命令用于为地址分配新的 ADI 版本。在 Oracle Solaris SPARC 系统上，只在本地模式下支持该命令。

本地模式语法

`adi assign` <*addr*> [/ <*count*>] = <*ver*>	为地址分配新的 ADI 版本，从 *addr* 开始，共 *count* 个地址。缺省 count 为 1。*ver* 必须介于 0 和 15 之间。
`adi assign` <*&object*> [/ <*count*>] = <*ver*>	为对象的地址范围或 *count* 个字节（以较小者为准）分配新的 ADI 版本。缺省 count 为整个对象。
`adi assign` <*addr1*>, <*addr2*> = <*ver*>	为从 *addr1* 到 *addr2* 的地址分配新的 ADI 版本。*ver* 必须介于 0 和 15 之间。

其中：

addr 是调试对象的地址空间中的地址。

count 是字节数。

ver 是新分配的 ADI 版本。

adi examine 命令

adi examine 命令用于在每个高速缓存行中显示一个 ADI 版本。与紧接在前面的输出行相同的全部输出行组（字节偏移除外）将替换为仅包含星号 (*) 的行。始终输出最后一行。在 Oracle Solaris SPARC 系统上，只在本地模式下支持该命令。

本地模式语法

adi examine | x <addr> [/ <count>]	每个高速缓存行显示一个 ADI 版本，从 *addr* 开始，共 *count* 个地址。缺省 count 为 1。
adi examine | x *&object* [/ <*count*>]	显示对象的 ADI 版本，共 *count* 个字节，但受到对象地址范围的限制。缺省 count 为整个对象。每个高速缓存行仅输出一个版本。
adi examine | x <*addr1*> , <*addr2*>	每个高速缓存行显示一个 ADI 版本，从 *addr1* 到 *addr2*（包含这两个值）。

其中：

addr 是调试对象的地址空间中的地址。

count 是字节数。

assign 命令

在本地模式中，assign 命令用于为程序变量赋新值。在 Java 模式中，assign 命令用于为局部变量或参数赋新值。

本地模式语法

assign *variable* = *expression*

其中：

expression 是分配给 *variable* 的值。

Java 模式语法

`assign` *identifier* `=` *expression*

其中：

expression 是有效的 Java 表达式，可包括以下任何内容：

- *class-name* 是 Java 类。可使用下列任意值：
  - 使用句点 (.) 作为限定符的软件包路径；例如，`test1.extra.T1.Inner`
  - 前面带有英镑标记 (#) 并使用斜杠 (/) 和美元标记 ($) 作为限定符的全路径名。例如，`#test1/extra/T1$Inner`。如果使用 `$` 限定符，请使用引号将 *class-name* 引起来。
- *field-name* 是类中字段的名称。
- *identifier* 是一个局部变量或参数，包括 `this`、当前类实例变量 (*object-name*.*field-name*) 或类（静态）变量 (*class-name*.*field-name*)。
- *object-name* 是 Java 对象的名称。

`attach` 命令

`attach` 命令用于将 `dbx` 连接到运行中的进程，停止执行并将程序置于调试控制下。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`attach` *process-ID*	使用进程 ID *process-ID* 开始调试程序。`dbx` 使用 `/proc` 查找程序。
`attach -p` *process-ID program-name*	使用进程 ID *process-ID* 开始调试 *program-name*。
`attach` *program-name process-ID*	使用进程 ID *process-ID* 开始调试 *program-name*。*program-name* 可为 `-`。`dbx` 使用 `/proc` 查找它。
`attach -r ...`	使用 `-r` 选项时，`dbx` 保留所有 `watch` 命令、`display` 命令、`when` 命令以及 `stop` 命令。如果不使用 `-r` 选项，则执行隐式 `delete all` 命令和 `undisplay 0` 命令。

其中：

process-ID 是运行中的进程的进程 ID。

program-name 是正在运行的程序的路径名。

有关如何将 dbx 连接到正在运行的 Java 进程的信息，请参见“将 dbx 连接到正在运行的 Java 应用程序” [202]。

bsearch 命令

bsearch 命令用于在当前源文件中向后搜索。仅在本地模式中有效。

语法

`bsearch` *string*	在当前文件中向后搜索 *string*。
`bsearch`	使用上一搜索字符串重复搜索。

其中：

string 是字符串。

call 命令

在本地模式中，call 命令用于调用过程。在 Java 模式中，call 命令用于调用方法。

您还可以使用 call 命令调用函数。要显示返回值，请使用 print 命令。

有时，调用的函数会到达断点。您可以选择使用 cont 命令继续，或使用 pop -c 终止调用。如果调用的函数导致段故障，后一种方法也很有用。

本地模式语法

`call [-lang` *language*`] [-resumeone] [-m] [+m]` *procedure* `([`*parameters*`])`

其中：

language 是被调用过程的语言。

procedure 是过程的名称。

parameters 是过程的参数。

-lang 指定被调用过程的语言，并指示 dbx 使用指定语言的调用转换。要被调用的过程先前编译时未显示调试信息并且 dbx 不知道如何传递参数的情况下，此选项很有用。

-resumeone 在调用过程时仅恢复一个线程。有关更多信息，请参见“恢复执行” [154]。

当 dbxenv 变量 macro_expand 设置为 off 时，-m 指定宏扩展应用于过程和参数。

当 dbxenv 变量 macro_expand 设置为 on 时，+m 指定跳过宏扩展。

Java 模式语法

call [*class-name*.|*object-name*.] *method-name* ([*parameters*])

其中：

class-name 是 Java 类。可使用下列任意值：

- 使用句点 (.) 作为限定符的软件包路径；例如，test1.extra.T1.Inner
- 前面带有英镑标记 (#) 并使用斜杠 (/) 和美元标记 ($) 作为限定符的全路径名。例如，#test1/extra/T1$Inner。如果使用 $ 限定符，请使用引号将 *class-name* 引起来。

object-name 是 Java 对象的名称。

method-name 是 Java 方法的名称。

parameters 是方法的参数。

cancel 命令

cancel 命令用于取消当前信号。它主要用在 when 命令的主体中（请参见“when 命令” [367]）。仅在本地模式中有效。

当 dbx 由于某个信号而停止时，信号通常会被取消。如果 when 命令已与信号事件连接，则信号不会被自动取消。可以使用 cancel 命令显式取消信号。

catch 命令

catch 命令用于捕获指定的信号。仅在本地模式中有效。

如果进程收到指定信号，则捕获该信号将导致 dbx 停止运行程序。如果在该点继续运行程序，则程序不会处理该信号。

语法

`catch`	输出已捕获信号的列表。
`catch` *number* *number* `...`	捕获编号为 *number* 的信号。
`catch` *signal* *signal* `...`	捕获名为 *signal* 的信号。但无法捕获或忽略 `SIGKILL`。
`catch $(ignore)`	捕获所有信号。

其中：

number 是信号的编号。

signal 是信号名。

check 命令

check 命令用于启用检查内存访问、泄漏或使用，并输出运行时检查 (runtime checking, RTC) 的当前状态。仅在本地模式中有效。

由此命令启用的运行时检查功能可由 debug 命令重置为其初始状态。

语法

本节提供有关 check 命令选项的信息。

`check [`*functions*`] [`*files*`] [`*loadobjects*`]`

等效于 *functions*、*files* 和 *loadobjects* 中的 `check -all`、`suppress all` 或 `unsuppress all`

其中：

functions 是一个或多个函数名。

files 是一个或多个文件名。

loadobjects 是一个或多个装入对象名。

您可使用此命令将运行时检查用于需要的地方。

注 - 为了检测所有错误，RTC 不要求使用 -g 编译程序。但是，有时为了保证准确检测某些错误（通常是从未初始化的内存中读取），需要使用符号 (-g) 信息。为此，如果没有符号信息，会禁止某些错误（a.out 的 rui，以及共享库的 rui + aib + air）。可通过使用 suppress 和 unsuppress 更改此行为。

-access 选项

-access 选项启用检查。RTC 报告下列错误：

baf	错误释放
duf	重复释放
maf	未对齐释放
mar	未对齐读
maw	未对齐写
oom	内存不足
rob	从数组越界内存中读
rua	从未分配的内存中读
rui	从未初始化的内存中读
wob	写入到数组越界内存
wro	写入到只读内存
wua	写入到未分配内存

缺省行为是在检测到每个访问错误之后停止进程，可以使用 rtc_auto_continue dbxenv 变量更改此行为。如果设置为 on，则访问错误将记录到文件。日志文件名由 dbxenv 变量 rtc_error_log_file_name 控制。

缺省情况下，每个独特访问错误只在第一次 发生时进行报告。可以使用 dbxenv 变量 rtc_auto_suppress 更改此行为。此变量的缺省设置为 on。

-leaks 选项

-leaks 选项的语法如下：

```
check -leaks [-frames n] [-match m]
```

启用泄漏检查。RTC 报告下列错误：

aib	可能发生内存泄漏 – 唯一指针指向块中间位置
air	可能发生内存泄漏 – 指向块的指针仅存在于寄存器中
mel	内存泄漏 – 无指向块的指针

启用泄漏检查之后，将在程序退出时自动生成泄漏报告。届时将报告包括可能的泄漏在内的所有泄漏。缺省情况下，会生成非详细报告，可以通过 dbxenv 变量 rtc_mel_at_exit 更改此行为。但可以随时获取泄漏报告（请参见“showleaks 命令” [339]）。

-frames *n* 表示在报告泄漏时最多显示 *n* 个不同的堆栈帧。-match *m* 用于组合泄漏；如果两个或更多泄漏分配时的调用堆栈与 *n* 帧匹配，则在一个合并泄漏报告中报告这些泄漏。

n 的缺省值为 8 或 *m* 的值（取较大值）。*n* 的最大值是 16。*m* 的缺省值是 8。

-memuse 选项

-memuse 选项的语法为：

```
check -memuse [-frames n] [-match m]
```

-memuse 选项与 -leaks 选项的作用类似，同时还可在程序退出时启用使用的块报告 (biu)。缺省情况下，会生成非详细的使用的块报告，可以通过 dbxenv 变量 rtc_biu_at_exit 更改此行为。在程序执行期间，您随时可以查看程序中已分配的内存位置（请参见“showmemuse 命令” [340]）。

-frames *n* 表示报告内存使用和泄漏时最多显示 *n* 个不同的堆栈帧。使用 -match *m* 可组合这些报告。如果两个或更多泄漏分配时的调用堆栈与 *m* 帧匹配，则在一个合并的内存泄漏报告中报告这些泄漏。

n 的缺省值为 8 或 *m* 的值，以较大者为准。*n* 的最大值为 16。*m* 的缺省值是 8。

-all 选项

-all 选项的语法为：

check -all [-frames *n*] [-match *m*]

等效于：

check -access 和 check -memuse [-frames *n*] [-match *m*]

dbxenv 变量 rtc_biu_at_exit 的值不随 check -all 变化，因此在缺省情况下，退出时不生成内存使用报告。有关 rtc_biu_at_exit 环境变量的说明，请参见“dbx 命令” [289]。

clear 命令

clear 命令用于清除断点。仅在本地模式中有效。

使用 stop 命令、trace 命令或者 when 命令以及 inclass 参数、inmethod 参数、infile 参数或 infunction 参数创建的事件处理程序将创建多组断点。如果您在 clear 命令中指定的 *line* 与这些断点之一匹配，则仅清除该断点。一旦使用此方法清除了属于某个集的单个断点，则无法再启用该断点。但是，禁用并再次启用相关的事件处理程序将重建所有断点。

语法

clear [*filename: line*]

其中：

line 是源代码行的编号，以便清除指定行中的所有断点。

filename 是源代码文件的名称，以便清除指定文件中行 *line* 的所有断点。

如果未指定文件或行，则清除当前停止点的所有断点。

collector 命令

collector 命令用于收集性能数据，以供性能分析器进行分析。仅在本地模式中有效。

本节列出了收集器命令，并提供相关的详细信息。

语法

`collector archive` *options*	指定实验终止时归档实验的模式。
`collector dbxsample` *options*	控制 dbx 停止目标进程时样例的收集。
`collector disable`	停止数据收集并关闭当前实验。
`collector enable`	启用收集器并打开新实验。
`collector heaptrace` *options*	启用或禁用堆跟踪数据收集。
`collector hwprofile` *options*	指定硬件计数器分析设置。
`collector limit` *options*	限制已记录的分析数据量。
`collector pause`	停止收集性能数据，但让实验保持打开状态。
`collector profile` *options*	指定用于收集调用堆栈分析数据的设置。
`collector resume`	在暂停后开始收集性能数据。
`collector sample` *options*	指定抽样设置。
`collector show` *options*	显示当前收集器设置。

`collector status`	查询有关当前实验的状态。
`collector store` *options*	实验文件控制和设置。
`collector synctrace` *options*	指定用于收集线程同步等待跟踪数据的设置。
`collector tha` *options*	指定用于收集线程分析程序数据的设置。
`collector version`	报告将用于收集数据的 `libcollector.so` 的版本。

其中：

要开始收集数据，请键入 `collector enable`。

要停止数据收集，请键入 `collector disable`。

`collector archive` 命令

`collector archive` 命令指定实验终止时使用的归档模式。

语法

`collector archive on|off|copy`	缺省情况下，使用正常归档。如果不归档，请指定 `off`。要将装入对象复制到实验中以提高可移植性，请指定 `copy`。

`collector dbxsample` 命令

`collector dbxsample` 命令指定 `dbx` 停止进程时是否记录样例。

语法

`collector dbxsample on|off`	缺省情况下，当 `dbx` 停止进程时，收集抽样数据。要指示此时不收集样例，请指定 `off`。

collector disable 命令

collector disable 命令用于停止收集数据，并关闭当前实验。

collector enable 命令

collector enable 命令用于启用收集器，并打开新实验。

collector heaptrace 命令

collector heaptrace 命令指定用于收集堆跟踪（内存分配）数据的选项。

语法

collector heaptrace on|off	缺省情况下，不收集堆跟踪数据。要收集此数据，请指定 on。

collector hwprofile 命令

collector hwprofile 命令指定用于收集硬件计数器溢出分析数据的选项。

语法

collector hwprofile on|off	缺省情况下，不收集硬件计数器溢出分析数据。要收集此数据，请指定 on。
collector hwprofile list	输出可用计数器列表。
collector hwprofile counter on|hi|high|lo|low|off	缺省情况下，不收集硬件计数器溢出分析数据。要收集此数据，请指定 on。您可以将计数器的分辨率设置为 high 或 low。如果未指定分辨率，则设置为正常。这些选项类似于 collect 命令选项。有关更多信息，请参见 collect(1) 手册页。
collector hwprofile	为硬件计数器溢出分析添加其他计数器。

`addcounter on|off`

`collector hwprofile counter` *name interval* `[`*name2 interval2*`]` 指定硬件计数器名称和时间间隔。

其中：

name 是硬件计数器的名称。

interval 是收集时间间隔（毫秒）。

name2 是另一个硬件计数器的名称。

interval2 是收集时间间隔（毫秒）。

硬件计数器特定于系统，因此可用的计数器选项取决于您使用的系统。许多系统都不支持硬件计数器溢出分析。在这些计算机中，这一功能被禁用。

`collector limit` 命令

`collector limit` 命令指定实验文件大小限制。

语法

`collector limit` *value* `| unlimited | none`

其中：

value（以 MB 为单位）是记录的分析数据量的限制值，必须为正数。当达到限制时，不会再记录分析数据，但是实验会保持打开状态，继续记录样本点。缺省情况下，记录的数据量不存在限制。

如果设置了限制，则指定 `unlimited` 或 `none` 来删除该限制。

`collector pause` 命令

`collector pause` 命令用于停止收集数据，但让当前实验保持打开状态。收集器暂停时不记录样本点。在暂停之前会生成一个样本，在恢复之后会立即生成另一个样本。数据收集可以使用 `collector resume` 命令加以恢复。

collector profile 命令

collector profile 命令指定用于收集分析数据的选项。

语法

collector profile on|off	指定分析数据收集模式。
collector profile timer *interval*	指定分析计时器周期（定点或浮点），可在后面添加 m 表示毫秒或添加 u 表示微秒。

collector resume 命令

collector resume 命令用于在使用 collector pause 命令（请参见“collector pause 命令” [283]）暂停数据收集后恢复数据收集。

collector sample 命令

collector sample 命令指定抽样模式和抽样时间间隔。

语法

collector sample periodic|manual	指定样例模式。
collector sample period *seconds*	指定抽样时间间隔（以 *seconds* 为单位）。
collector sample record [*name*]	使用可选的 *name* 记录抽样数据。

其中：

seconds 是抽样时间间隔长度。

name 是抽样的名称。

collector show 命令

collector show 命令用于显示一种或多种选项设置。

语法

collector show	显示所有设置
collector show all	显示所有设置
collector show archive	显示归档设置
collector show duration	显示持续时间设置
collector show hwprofile	显示硬件计数器数据设置
collector show heaptrace	显示堆跟踪数据设置
collector show limit	显示实验大小限制
collector show pausesig	显示暂停和恢复信号
collector show profile	显示调用堆栈分析设置
collector show sample	显示抽样设置
collector show samplesig	显示抽样信号
collector show store	显示存储设置
collector show synctrace	显示线程同步等待跟踪设置

`collector show tha`	显示线程分析程序数据设置

collector status 命令

`collector status` 命令用于查询当前实验的状态。它将返回工作目录和实验名称。

collector store 命令

`collector store` 命令用于指定存储实验的目录和文件名。

语法

`collector store {-directory` *pathname* `| -filename` *filename* `| -group` *string*`}`

其中：

pathname 是要存储实验的目录的路径名。

filename 是实验文件名。

string 是实验组名。

collector synctrace 命令

`collector synctrace` 命令指定用于收集同步等待跟踪数据的选项。

语法

`collector synctrace on|off`	缺省情况下，不收集线程同步等待跟踪数据。要收集此数据，请指定 `on`。
`collector synctrace threshold`	以微秒为单位指定阈值。缺省值是 1000。如果指定 `calibrate`，则将自动计算阈值。

{*microseconds*|
calibrate}

其中：

microseconds 是阈值，低于此阈值的同步等待事件都将被放弃。

collector tha 命令

collector tha 命令指定用于收集线程分析器数据的选项。

语法

collector tha on|off	缺省情况下，不收集线程分析程序数据。要收集此数据，请指定 on。

collector version 命令

collector version 命令用于报告将用于收集数据的 libcollector.so 的版本。

语法

collector version

cont 命令

cont 命令用于使进程继续执行。在本地模式和 Java 模式中，它的语法和功能相同。

语法

cont	继续执行。多线程进程中的所有线程都会被恢复。使用 Ctrl-C 停止执行程序。
cont ... -sig *signal*	使用信号 *signal* 继续执行。

`cont ... ID`	*id* 指定要继续的线程或 LWP。
`cont at line [ID]`	在行 *line* 处继续执行。如果应用程序为多线程，则 *ID* 是必需的。
`cont ... -follow parent|child|both`	如果 `dbx` `follow_fork_mode` 环境变量设置为 `ask`，并且您选择了 `stop`，请使用此选项选择要后跟的进程。`both` 只能在 Oracle Developer Studio IDE 中使用。

dalias 命令

`dalias` 命令用于定义 `dbx` 样式（`csh` 样式）的别名。仅在本地模式中有效。

语法

`dalias [name [definition]]`	（`dbx` 别名）列出所有当前定义的别名。 如果指定了名称，则列出别名 *name* 的定义（如有）。 如果还指定了定义，则定义 *name* 作为 *definition* 的别名。*definition* 可以包含空格。一个分号或新行即可结束定义。

其中：

name 是别名的名称。

definition 是别名的定义。

`dbx` 接受别名中常用的以下 `csh` 历史替代元语法：

```
!:<n>

!-<n>

!^

!$

!*
```

`!` 前通常需要加上反斜杠。例如：

```
dalias goto "stop at \!:1; cont; clear"
```

有关更多信息，请参见 `csh`(1) 手册页。

dbx 命令

dbx 命令用于启动 dbx。

本地模式语法

`dbx` *options program-name* [*core* | *process-ID*]	调试 *program-name*。 如果指定了 *core*，则使用信息转储文件 *core* 调试 *program-name*。 如果指定了 *process-ID*，则使用进程 ID *process-ID* 调试 *program-name*。
`dbx` *options* `-` {*process-ID*|*core*}	如果指定了 *process ID*，则调试进程 ID *process-ID*；`dbx` 使用 `/proc` 查找程序。 如果指定了 *core*，则使用信息转储文件 *core* 进行调试。
`dbx` *options* `-` *core*	使用信息转储文件 *core* 进行调试。
`dbx` *options* `-r` *program-name arguments*	使用参数 *arguments* 运行 *program-name*。如果异常终止，请开始调试 *program-name*，否则仅退出即可。

其中：

program-name 是要调试的程序的名称。

process-ID 是运行中的进程的进程 ID。

arguments 是要传递给程序的参数。

options 在“选项” [290]中进行了介绍。

Java 模式语法

`dbx` *options program-name*{`.class` | `.jar`}	调试 *program-name*。
`dbx` *options program-name*{`.class` | `.jar`} *process-ID*	使用进程 ID *process ID* 调试 *program-name*。

`dbx` *options* *process-ID*	调试进程 ID *process ID*；`dbx` 使用 `/proc` 查找程序。
`dbx` *options* { `-r` | `-a`} *program-name*`{.class | .jar}` *arguments*	使用参数 *arguments* 运行 *program-name*。如果异常终止，请开始调试 *program-name*，否则仅退出即可。

其中：

program-name 是要调试的程序的名称。

process-id 是运行中的进程的进程 ID。

arguments 是要传递给程序（而不是 JVM 软件）的参数。

options 在“选项” [290]中进行了介绍。

选项

下表列出了适用于本地模式和 Java 模式的 `dbx` 命令选项：

`--a` *arguments*	为程序装入程序参数 *arguments*。
`--B`	抑制所有消息；返回时带有正在被调试的程序的退出代码。
`-c` *commands*	执行 *commands* 后，提示输入。
`-C`	预装入运行时检查库（请参见“check 命令” [276]）。
`-d`	与 `-s` 一起使用，读取后删除 *file*。
`-e`	回显输入命令。
`-f`	强制装入信息转储文件，即使该文件不匹配。
`-h`	输出有关 `dbx` 的用法帮助。
`-I` *dir*	将 *dir* 添加至 `pathmap` 设置（请参见“pathmap 命令” [327]）。
`-k`	保存并恢复键盘转换状态。
`-q`	禁止关于读取 stabs 的消息。
`-r`	运行程序；如果程序正常退出，则退出。
`-R`	输出 `dbx` 的自述文件。
`-s` *file*	使用 *file* 而非 */current-dir*`/.dbxrc` 或 `$HOME/ .dbxrc` 作为启动文件
`-S`	禁止读取初始化文件 */install-dir*`/lib/dbxrc`。
`-V`	输出 `dbx` 的版本信息。
`-w` *n*	执行 `where` 命令时跳过 *n* 帧。
`-x exec32`	运行 32 位 `dbx` 二进制文件，而不是运行 64 位操作系统的系统上缺省情况下运行的 64 位 `dbx` 二进制文件。

`--`	标记选项列表的末尾；如果程序名以短划线开头，则使用此选项。

dbxenv 命令

`dbxenv` 命令用于列出或设置 `dbxenv` 变量。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`dbxenv` [*environment-variable setting*]	显示 `dbxenv` 变量的当前设置。如果指定了 `dbxenv` 变量，则将 `dbxenv` 变量设置为 *setting*。

其中：

environment-variable 是 `dbxenv` 变量。

setting 是相应变量的有效设置。

debug 命令

`debug` 命令用于列出或更改所调试的程序。在本地模式中，它装入指定的应用程序，然后开始调试该应用程序。在 Java 模式中，它装入指定的 Java 应用程序，接着检查是否存在类文件，然后开始调试该应用程序。

本地模式语法

`debug`	输出被调试程序的名称和参数。
`debug` *program-name*	在不使用进程或信息转储文件的情况下，开始调试 *program-name*。
`debug -c` *core program-name*	使用信息转储文件 *core* 开始调试 *program-name*。
`debug -p` *process-ID program-name*	使用进程 ID *process-ID* 开始调试 *program-name*。

`debug` *program-name core*	使用信息转储文件 *core* 开始调试 *program*。*program-name* 可以为 `-`。`dbx` 将尝试从信息转储文件中提取可执行文件的名称。有关详细信息，请参见“调试信息转储文件” [36]。
`debug` *program-name process-ID*	使用进程 ID *process-ID* 开始调试 *program-name*。*program-name* 可为 `-`；`dbx` 使用 `/proc` 查找程序。
`debug -f ...`	强制装入信息转储文件，即使该文件不匹配。
`debug -r ...`	使用 `-r` 选项时，`dbx` 保留所有 `display`、`trace`、`when` 和 `stop` 命令。如果不使用 `-r` 选项，则执行隐式 `delete all` 和 `undisplay 0`。
`debug -clone ...`	`-clone` 选项可使另一个 `dbx` 进程开始执行，从而允许一次调试多个进程。仅当在 Oracle Developer Studio IDE 中运行时才有效。
`debug -clone`	启动另一个 `dbx` 进程，但不进行任何调试。仅当在 Oracle Developer Studio IDE 中运行时才有效。
`debug [`*options*`] -- `*program-name*	开始调试 *program-name*，即使 *program-name* 以短划线开头。

其中：

core 是信息转储文件的名称。

options 在“选项” [293]中进行了介绍。

process-ID 是运行中的进程的进程 ID。

program-name 是程序的路径名。

使用 `debug` 命令装入程序时，禁用泄漏检查和访问检查。可以使用 `check` 命令启用它们。

Java 模式语法

`debug`	输出被调试程序的名称和参数。
`debug` *program-name* `[.class | .jar]`	在不使用进程的情况下，开始调试 *program-name*。
`debug -p` *process-ID program-name* `[.class | .jar]`	使用进程 ID *process-ID* 开始调试 *program-name*。

`debug` *program-name* `[.class | .jar]` *process-ID*	使用进程 ID *process-ID* 开始调试 *program-name*。*program-name* 可为 `-`；`dbx` 使用 `/proc` 查找程序。
`debug -r`	使用 `-r` 选项时，`dbx` 保留所有 `watch` 命令、`display` 命令、`trace` 命令、`when` 命令以及 `stop` 命令。如果不使用 `-r` 选项，则执行隐式 `delete all` 命令和 `undisplay 0` 命令。
`debug -clone ...`	`-clone` 选项可使另一个 `dbx` 进程开始执行，从而允许一次调试多个进程。仅当在 Oracle Developer Studio IDE 中运行时才有效。
`debug -clone`	启动另一个 `dbx` 进程，但不进行任何调试。仅当在 Oracle Developer Studio IDE 中运行时才有效。
`debug [`*options*`] -- `*program-name*`{.class | .jar}`	开始调试 *program-name*，即使 *program-name* 以短划线开头。

其中：

options 在“选项” [293]中进行了介绍。

process-ID 是运行中的进程的进程 ID。

program-name 是程序的路径名。

选项

`-c` *commands*	执行 *commands* 后，提示输入。
`-d`	与 `-s` 一起使用，读取后删除 file。
`-e`	回显输入命令。
`-I` *directory_name*	将 *directory_name* 添加至 `pathmap` 设置（请参见“`pathmap` 命令” [327]）。
`-k`	保存并恢复键盘转换状态。
`-q`	禁止关于读取 stabs 的消息。
`-r`	运行程序；如果程序正常退出，则退出。

-R	输出 dbx 的自述文件。
-s *file*	使用 *file* 而非 *current_directory*/.dbxrc 或 $HOME/.dbxrc 作为启动文件
-S	禁止读取初始化文件 /*install-dir*/lib/dbxrc。
-V	输出 dbx 的版本信息。
-w *n*	执行 where 命令时跳过 *n* 帧。
--	标记选项列表的末尾；如果程序名以短划线开头，则使用此选项。

delete 命令

delete 命令用于删除断点和其他事件。在本地模式和 Java 模式中，它的语法和功能相同。

语法

delete [-h] *handler-ID* ...	删除给定 *handler-ID* 的 trace 命令、when 命令或 stop 命令。要删除隐藏的处理程序，必须包括 -h 选项。
delete [-h] O | all | -all	删除所有 trace 命令、when 命令和 stop 命令，但不包括永久性和隐藏的处理程序。指定 -h 也可以删除隐藏的处理程序。
delete -temp	删除所有临时处理程序。
delete $firedhandlers	删除所有导致最近停止的处理程序。

其中：

handler-ID 是处理程序的标识符。

detach 命令

detach 命令用于取消 dbx 对目标进程的控制。

本地模式语法

`detach` [`-sig` *signal* | `-stop`]

使 `dbx` 与目标分离，并取消所有待处理信号。

如果指定了 `-sig` 选项，则在转发指定的 *signal* 时分离。

如果指定了 `-stop` 选项，则使 dbx 与目标分离，并让进程处于停止状态。使用此选项可以临时应用其他可能因独占访问而被阻止的基于 `/proc` 的调试工具。有关示例，请参见“从进程中分离 dbx” [81]。

其中：

signal 是信号名。

Java 模式语法

`detach`

使 `dbx` 与目标分离，并取消所有待处理信号。

dis 命令

`dis` 命令用于反汇编计算机指令。仅在本地模式中有效。

语法

`dis [ -a ]` *address* `[/`*count*`]`

反汇编 *count* 个始于地址 *address* 的指令（缺省值为 10）。

`dis` *address1*, *address2*

反汇编 *address1* 到 *address2* 的指令。

`dis`

反汇编 10 条指令，从 + 开始。

其中：

address 是开始反汇编的地址。*address* 的缺省值为先前汇编的最后一个地址的下一个地址。此值由 `examine` 命令共享。

address1 是开始反汇编的地址。

address2 是停止反汇编的地址。

count 是反汇编的指令数。*count* 的缺省值是 10。

选项

`-a`	如果与函数地址一起使用，则反汇编整个函数。如果使用时不带参数，则反汇编当前访问函数的剩余部分（如果有）。

display 命令

在本地模式中，`display` 命令用于在每个停止点处对表达式重新求值并进行输出。在 Java 模式中，`display` 命令用于在每个停止点处对表达式、局部变量或参数求值并进行输出。对象引用将扩展为一层，而数组将逐条输出。

键入命令时，会针对当前作用域解析表达式，并在每个停止点处对表达式重新求值。因为在到达入口点时会解析表达式，所以可以立即检验表达式是否正确。

如果在 IDE 中运行 `dbx` 或在 Sun Studio 12 发行版、Sun Studio 12 Update 1 发行版、Oracle Solaris Studio 12.2 发行版或后来的更新版本中运行 `dbxtool`，`display` *expression* 命令的行为实际上如同 `watch $(which` *expression*`)` 命令一样。

本地模式语法

`display`	输出所显示的表达式列表。
`display` *expression*`, ...`	显示每个停止点处表达式 *expression*, ... 的值。由于在到达入口点时会解析 *expression*，因此可以立即检验表达式是否正确。
`display [-r|+r|-d|+d|-S|+S|-p|+p|-L|-f`*format*`|-F`*format*`|-m|+m|--]` *expression*`, ...`	有关这些标志的含义，请参见“`print` 命令” [329]。

其中：

expression 是有效的表达式。

format 是输出表达式时要使用的输出格式。有关有效格式的信息，请参见“`print` 命令” [329]。

Java 模式语法

`display`	输出所显示的变量和参数列表。
`display` *expression*|*identifier*, ...	显示每个停止点处 *identifier*、*...* 的变量和参数的值。
`display [-r|+r|-d|+d|-p|+p|-f`*format*`|-F`*format*`|-F`*format*`|--]` *expression*|*identifier*, ...	有关这些标志的含义，请参见“`print` 命令” [329]。

其中：

class-name 是 Java 类。可使用下列任意值：

- 使用句点 (.) 作为限定符的软件包路径；例如，`test1.extra.T1.Inner`
- 前面带有英镑标记 (#) 并使用斜杠 (/) 和美元标记 ($) 作为限定符的全路径名。例如，`#test1/extra/T1$Inner`。如果使用 `$` 限定符，请使用引号将 *class-name* 引起来。

expression 是有效的 Java 表达式。

field-name 是类中字段的名称。

format 是输出表达式时要使用的输出格式。有关有效格式的信息，请参见“`print` 命令” [329]。

identifier 是一个局部变量或参数，包括 `this`、当前类实例变量 (*object-name*.*field-name*) 或类（静态）变量 (*class-name*.*field-name*)。

object-name 是 Java 对象的名称。

down 命令

`down` 命令用于下移调用堆栈（远离 `main`）。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`down`	调用堆栈下移一级。

`down` *number*	将调用堆栈下移 *number* 级。
`down -h` [*number*]	下移调用堆栈，但不跳过隐藏的帧。

其中：

number 是调用堆栈级数。

dump 命令

`dump` 命令用于输出某个过程的所有局部变量。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`dump` [*procedure*]	输出当前过程的所有局部变量。 如果指定了过程，则输出 *procedure* 的所有局部变量。

其中：

procedure 是过程名。

edit 命令

`edit` 命令用于对源文件调用 `$EDITOR`。仅在本地模式中有效。

如果 `dbx` 不是在 Oracle Developer Studio IDE 中运行，`edit` 命令将使用 `$EDITOR`。否则，它将向 IDE 发送消息，显示相应文件。

语法

`edit` [*filename* | *procedure*]	编辑当前文件。 如果指定了文件名，则编辑指定的文件 *filename*。 如果指定了过程，则编辑包含函数或过程 *procedure* 的文件。

其中：

filename 是文件名。

procedure 是函数名或过程名。

examine 命令

examine 命令用于显示内存内容。仅在本地模式中有效。

x 命令是 examine 命令的别名。

语法

examine [*address*] [/ [*count*] [*format*]]	以 *format* 格式显示始于 *address* 的 *count* 项内存内容。
examine *address1*, *address2* [/ [*format*]]	以 *format* 格式显示 *address1* 到 *address2*（首末地址包含在内）的内存内容。
examine *address*= [*format*]	以指定格式显示地址（而不是地址的内容）。 *address* 可以是 +，表示先前显示的最后一个地址的下一个地址（如同忽略了先前显示的最后一个地址）。 x 是 examine 的预定义别名。

其中：

address 是开始显示内存内容的地址。*address* 的缺省值是最后显示其内容的地址的下一个地址。此值由 dis 命令共享。

address1 是开始显示内存内容的地址。

address2 是停止显示内存内容的地址。

count 是从中显示内存内容的地址数。*count* 的缺省值是 1。

format 是内存地址内容的显示格式。第一个 examine 命令的缺省格式是 X（十六进制），后续 examine 命令的缺省格式是前一个 examine 命令中指定的格式。以下所示为 *format* 的有效值：

o,O	八进制（2 或 4 字节）

x,X	十六进制（2 或 4 字节）
b	八进制（1 字节）
c	字符
w	宽字符
s	字符串
W	宽字符串
f	十六进制和浮点（4 字节，6 位精度）
F	十六进制和浮点（8 字节，14 位精度）
g	与 F 相同
E	十六进制和浮点（16 字节，14 位精度）
ld,lD	十进制（4 字节，与 D 相同）
lo,lO	八进制（94 字节，与 O 相同）
lx,lX	十六进制（4 字节，与 X 相同）
Ld,LD	十进制（8 字节）
Lo,LO	八进制（8 字节）
Lx,LX	十六进制（8 字节）

exception 命令

exception 命令用于输出当前 C++ 异常的值。仅在本地模式中有效。

语法

exception [-d | +d]	输出当前 C++ 异常的值（如果有）。

其中：

`-d` 启用显示动态异常。

`+d` 禁用显示动态异常。

exists 命令

`exists` 命令用于检查符号名是否存在。仅在本地模式中有效。

语法

`exists` *name*	如果在当前程序中找到 *name*，则返回 0；如果没找到 *name*，则返回 1。

其中：

name 是符号名。

file 命令

`file` 命令用于列出或更改当前文件。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`file` *filename*	输出当前文件的名称。 如果指定了文件名，则更改当前文件。

其中：

filename 是文件名。

files 命令

在本地模式中，`files` 命令用于列出与正则表达式匹配的文件名。在 Java 模式中，`files` 命令用于列出 `dbx` 已知的所有 Java 源文件。如果 Java 源文件与 `.class`

或 .jar 文件不在同一目录中，dbx 可能找不到它们，除非您已经设置了 $JAVASRCPATH 环境变量。有关更多信息，请参见“指定 Java 源文件的位置” [203]。

本地模式语法

`files`	列出提供当前程序调试信息的所有文件（使用 -g 编译的文件）的名称。
`files` *regular-expression*	列出所有使用 -g 编译且与指定正则表达式匹配的文件的名称。

其中：

regular-expression 是正则表达式。

例如：

```
(dbx) files ^r
myprog:
retregs.cc
reg_sorts.cc
reg_errmsgs.cc
rhosts.cc
```

Java 模式语法

`files`	列出 dbx 已知的所有 Java 源文件的名称。

fix 命令

fix 命令用于重新编译修改过的源文件，并动态将修改过的函数链接至应用程序中。仅在本地模式中有效。该功能在 Linux 平台上无效。

语法

`fix [`*file-name file-name ...*`] [-options]`	修复当前文件。 如果列出了文件名，将修复列表中的文件。

其中：

-options 是指以下有效选项。

-f	强制修复文件，即使源文件未经过修改。
-a	修复所有已修改的文件。
-g	清除 -O 标志并添加 -g 标志。
-c	输出编译行（可以包含某些在内部添加以供 dbx 使用的选项）。
-n	不执行编译/链接命令（与 -v 一起使用）。
v	详细模式（覆盖 dbx 环境变量 fix_verbose 的设置）。
+v	非详细模式（覆盖 dbx 环境变量 fix_verbose 的设置）。

fixed 命令

fixed 命令用于列出所有已修复文件的名称。仅在本地模式中有效。

fortran_modules 命令

fortran_modules 命令用于列出当前程序中的 Fortran 模块或者其中一个模块中的函数或变量。

语法

fortran_modules	用于列出当前程序中的 Fortran 模块。
[-f *module-name* | -v *module-name*]	如果指定了 -f 选项，则列出指定模块中的所有函数。 如果指定了 -v 选项，则列出指定模块中的所有变量。

frame 命令

frame 命令用于列出或更改当前堆栈帧号。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`frame`	显示当前帧的帧号。
`frame [-h]` *number*	将当前帧设置为帧 *number*。
`frame [-h]` `+[`*number*`]`	在堆栈中上移 *number* 帧；缺省值为 1。
`frame [-h] -[`*number*`]`	在堆栈中下移 *number* 帧；缺省值为 1。
`-h`	即使帧隐藏，也转至帧。

其中：

number 是调用堆栈中帧的编号。

func 命令

在本地模式中，`func` 命令用于列出或更改当前函数。在 Java 模式中，`func` 命令用于列出或更改当前方法。

本地模式语法

`func [`*procedure*`]`	输出当前函数的名称。 如果指定了过程，则将当前函数更改为函数或过程 *procedure*。

其中：

procedure 是函数名或过程名。

Java 模式语法

`func`	输出当前方法的名称。
`func [`*class-name*`.]`*method-*	将当前函数更改为方法 *method-name*。

name
[(*parameters*)]

其中：

class-name 是 Java 类。可使用下列任意值：

- 使用句点 (.) 作为限定符的软件包路径；例如，test1.extra.T1.Inner
- 前面带有英镑标记 (#) 并使用斜杠 (/) 和美元标记 ($) 作为限定符的全路径名。例如，#test1/extra/T1$Inner。如果使用 $ 限定符，请使用引号将 *class-name* 引起来。

method-name 是 Java 方法的名称。

parameters 是方法的参数。

funcs 命令

funcs 命令用于列出与某个正则表达式匹配的所有函数名。仅在本地模式中有效。

语法

funcs [-f *filename*] [-g] [*regular-expression*]	列出当前程序中的所有函数， 如果指定了 -f *filename*，则列出文件中的所有函数。如果指定了 -g，则列出具有调试信息的所有函数。如果 *filename* 以 .o 结尾，则会列出所有函数，包括那些编译器自动创建的函数。否则，只会列出出现在源代码中的函数。 如果指定了 *regular-expression*，则列出与正则表达式匹配的所有函数。

其中：

filename 是要列出其所有函数的文件的名称。

regular-expression 是要列出的所有匹配函数所匹配的正则表达式。

例如：

```
(dbx) funcs [vs]print
”libc.so.1”isprint
”libc.so.1”wsprintf
```

```
"libc.so.1"sprintf
"libc.so.1"vprintf
"libc.so.1"vsprintf
```

gdb 命令

gdb 命令用于支持 GDB 命令集。仅在本地模式中有效。

语法

`gdb on | off` 使用 `gbd on` 可进入 GDB 命令模式，在该模式下，`dbx` 理解并接受 GDB 命令。要退出 GDB 命令模式并返回到 `dbx` 命令模式，请键入 `gdb off`。在 GDB 命令模式下不接受 `dbx` 命令；在 `dbx` 模式下不接受 GDB 命令。所有调试设置（例如，断点）在不同的命令模式中均保留。

此发行版本不支持下列 GDB 命令：

- `commands`
- `define`
- `handle`
- `hbreak`
- `interrupt`
- `maintenance`
- `printf`
- `rbreak`
- `return`
- `signal`
- `tcatch`
- `until`

handler 命令

`handler` 命令用于修改事件处理程序（启用、禁用等）。在本地模式和 Java 模式中，它的语法和功能相同。

对于每个需要在调试会话中管理的事件，都会创建一个处理程序。命令 `trace`、`stop` 和 `when` 均可创建处理程序。其中每个命令均会返回一个称为处理程序 *ID* (*handler-ID*) 的数字。`handler`、`status` 和 `delete` 命令以一般方式处理或提供有关处理程序的信息。

语法

`handler [-enable | -disable]` *handler-ID* `...`	启用或禁用指定处理程序；要启用或禁用所有处理程序，可将 *handler-ID* 指定为 all。要禁用导致最近停止的处理程序，请使用 `$firedhandlers` 而不是 *handler-ID*。
`handler -count` *handler-ID new-limit*	输出指定处理程序的行程计数器的值。 如果指定了新限制参数，将为指定事件设置新的计数限制。
`handler -reset` *handler-ID*	使指定处理程序的行程计数器复位。

其中：

handler-ID 是处理程序的标识符。

hide 命令

`hide` 命令用于隐藏与某个正则表达式匹配的堆栈帧。仅在本地模式中有效。

语法

`hide` *regular-expression*	列出当前正在使用的堆栈帧过滤器。 如果指定了正则表达式，则隐藏与 *regular-expression* 匹配的堆栈帧。正则表达式将匹配函数名称或装入对象的名称，是 `sh` 或 `ksh` 文件匹配样式正则表达式。

其中：

regular-expression 是正则表达式。

ignore 命令

`ignore` 命令用于告知 `dbx` 进程不要捕获指定信号。仅在本地模式中有效。

忽略信号会导致进程收到这种信号时 dbx 不会停止。

语法

`ignore [`*number* `... | `*signal* `...]`	输出已忽略信号的列表。 如果指定了信号编号，则忽略编号为 *number* 的信号。 如果指定了信号，则忽略名为 *signal* 的信号。但无法捕获或忽略 SIGKILL。

其中：

number 是信号的编号。

signal 是信号名。

import 命令

import 命令用于从 dbx 命令库中导入命令。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`import` *path-name*	从 dbx 命令库 *path-name* 中导入命令。

其中：

path-name 是 dbx 命令库的路径名。

intercept 命令

intercept 命令用于抛出指定类型的 (C++) 异常（仅限于 C++）。仅在本地模式中有效。

当所抛出异常的类型与拦截列表中的某一类型匹配时，dbx 将停止，除非该异常的类型也与排除列表中的某一类型匹配。没有匹配捕获的抛出异常称为“未处理”抛出。与从中抛出异常的函数的异常规范不匹配的抛出异常称为“意外”抛出。

缺省情况下，会拦截未处理的抛出和意外抛出。

语法

`intercept -x` *excluded-typename* `[,` *excluded-typename* `...]`	将 *excluded-typename* 的抛出添加到排除列表。
`intercept -a[ll] -x` *excluded-typename* `[,` *excluded-typename*`...]`	将 *excluded-typename* 之外的所有类型添加到拦截列表。
`intercept -s[et] [`*intercepted-typename* `[,` *intercepted-typename* `...]] [-x` *excluded-typename* `[,` *excluded-typename*`]]`	清除拦截列表和排除列表，并设置列表以仅拦截或排除指定类型的抛出。
`intercept`	列出被拦截的类型。

其中：

included-typename 和 *excluded-typename* 是异常类型规范，如 `List <int>` 或 `unsigned short`。

java 命令

当 `dbx` 处于 JNI 模式时，使用 `java` 命令表示将要执行指定命令的 Java 版本。这将使指定命令使用 Java 表达式计算器，并在关联时显示 Java 线程和堆栈帧。

语法

`java` *command*

其中：

command 是要执行的命令的名称和参数。

jclasses 命令

`jclasses` 命令用于在您发出命令时输出 `dbx` 已知的所有 Java 类名。仅在 Java 模式中有效。

不输出程序中尚未装入的类。

语法

`jclasses [-a]` 输出 `dbx` 已知的所有 Java 类名。
如果指定了 `-a` 选项，则输出系统类和其他已知的 Java 类。

joff 命令

`joff` 命令用于将 `dbx` 从 Java 模式或 JNI 模式切换到本地模式。

jon 命令

`jon` 命令用于将 `dbx` 从本地模式切换到 Java 模式。

jpkgs 命令

`jpkgs` 命令用于在您发出命令时输出 `dbx` 已知的所有 Java 包名。仅在 Java 模式中有效。

不输出程序中尚未装入的包。

kill 命令

kill 命令用于向进程发送信号。仅在本地模式中有效。

语法

kill -l	列出所有已知信号编号、名称和说明。
kill	中止受控进程。
kill [*signal*]*job* ...	向列出的作业发送 SIGTERM 信号。 如果指定了 -signal 选项，则向列出的作业发送指定信号。

其中：

job 可以是进程 ID，也可以按下列任一方法进行指定：

%+	中止当前作业。
%-	中止上一作业。
%*number*	中止编号为 *number* 的作业。
%*string*	中止以 *string* 开头的作业。
%?*string*	中止包含 *string* 的作业。

其中：

signal 是信号名。

language 命令

language 命令用于列出或更改当前源语言。仅在本地模式中有效。

语法

language	输出通过 dbx 环境变量 language_mode 设置的当前语言模式。如果语言模式设置为 autodetect 或 main，该命令还将输出用于解析和求表达式值的当前语言的名称。

其中：

language 是 c、c++、fortran 或 fortran90。

注 - c 是 ansic 的别名。

line 命令

line 命令用于列出或更改当前行号。在本地模式和 Java 模式中，它的语法和功能相同。

语法

line [["*file-name*":] [*number*]]	显示当前行号。 如果指定了编号，则将当前行号设置为 *number*。 如果指定了文件名，则将当前行号设置为 *filename* 中的第 1 行。 如果同时指定了这两者，则将当前行号设置为 *file-name* 中的第 *number* 行。

其中：

filename 是要更改行号的文件名。文件名两边的引号 "" 是可选的。如果文件名包含空格，则需要使用引号。

number 是文件中的行号。

示例

```
line 100
line "/root/test/test.cc":100
```

list 命令

list 命令用于显示源文件的行。在本地模式和 Java 模式中，它的语法和功能相同。

列出的缺省行数 N 由 dbx 环境变量 output_list_size 控制。

语法

`list`	列出 N 行。
`list` *number*	列出行号为 *number* 的行。
`list +`	列出后 N 行。
`list +`*n*	列出后 *n* 行。
`list -`	列出前 N 行。
`list -`*n*	列出前 *n* 行。
`list` *n1*`,` *n2*	列出第 *n1* 行到第 *n2* 行。
`list` *n1*`, +`	列出第 *n1* 行到第 *n1* + N 行。
`list` *n1*`, +`*n2*	列出第 *n1* 行到第 *n1* + *n2* 行。
`list` *n1*`, -`	列出第 *n1*-N 行到第 *n1* 行。
`list` *n1*`, -`*n2*	列出第 *n1*-*n2* 行到第 *n1* 行。
`list` *function*	列出 *function* 源的开头。`list` *function* 将更改当前范围。有关更多信息，请参见“程序作用域” [63]。
`list` *filename*	列出文件 *filename* 的开头。
`list` *filename*`:`*n*	列出文件 *filename* 中从第 *n* 行开始的内容。

其中：

filename 是源代码文件的文件名。

function 是要显示的函数名。

number 是源文件中的行号。

n 是要显示的行数。

n1 是要显示的第一行的编号。

n2 是要显示的最后一行的编号。适当时，行号可能为 "$"，表示文件的最后一行。逗号是可选的。

选项

-i 或 -instr	混合源代码行和汇编代码。
-w 或 -w*n*	列出行或函数前后 N（或 *n*）行（范围）。此选项不能与加号 (+) 或减号 (-) 语法一起使用，指定了两个行号时，也不允许使用此选项。
-a	如果与函数一起使用，则列出整个函数。如果使用时不带参数，则列出当前访问函数的剩余部分（如果有）。

示例

```
list                      // list N lines starting at current line
list +5                      // list next 5 lines starting at current line
list -                    // list previous N lines
list -20                  // list previous 20 lines
list 1000                 // list line 1000
list 1000,$               // list from line 1000 to last line
list 2737 +24             // list line 2737 and next 24 lines
list 1000 -20             // list line 980 to 1000
list test.cc:33           // list source line 33 in file test.cc
list -w                   // list N lines around current line
list -w8 ”test.cc”func1   // list 8 lines around function func1
list -i 500 +10                // list source and assembly code for line
                500 to line 510
```

listi 命令

`listi` 命令用于显示源指令和反汇编指令。仅在本地模式中有效。此命令与使用 `list -i` 相同。

有关详细信息，请参见“`list` 命令” [312]。

loadobject 命令

`loadobject` 命令用于列出和管理装入对象中的符号信息。仅在本地模式中有效。

本节列出了 loadobject 选项，并提供相关的详细信息。

语法

`loadobject -list [`*regexp*`] [-a]`	显示当前装入的装入对象。
`loadobject -load` *loadobject*	为指定的装入对象装入符号。
`loadobject -unload [`*regexp*`]`	卸载指定的装入对象。
`loadobject -hide [`*regexp*`]`	从 dbx 的搜索算法中删除装入对象。
`loadobject -use [`*regexp*`]`	将装入对象添加到 dbx 的搜索算法中。
`loadobject -dumpelf [`*regexp*`]`	显示装入对象的各种 ELF 详细信息。
`loadobject -exclude` *ex-regexp*	不自动装入与 *ex-regexp* 匹配的装入对象。
`loadobject exclude -clear`	清除模式排除列表。

其中：

regexp 是正则表达式。如果未指定，则该命令应用于所有装入对象。

ex-regexp 不是可选的，必须指定。

该命令的缺省别名是 lo。

loadobject -dumpelf 命令

loadobject -dumpelf 命令用于显示装入对象的各种 ELF 详细信息。仅在本地模式中有效。

语法

`loadobject -dumpelf [`*regexp*`]`

其中：

regexp 是正则表达式。如果未指定，则该命令应用于所有装入对象。

该命令可转储出磁盘上装入对象文件的 ELF 结构的相关信息。该输出的详细信息随时变化。如果要解析此输出，请使用 Oracle Solaris OS 命令 `dump` 或 `elfdump`。

`loadobject -exclude` 命令

`loadobject -exclude` 命令用于告知 `dbx` 不要自动装入与指定正则表达式匹配的装入对象。

语法

`loadobject -exclude` *ex-regexp* `[-clear]`

其中：

ex-regexp 是正则表达式。

此命令可防止 `dbx` 自动为与指定正则表达式匹配的装入对象装入符号。与其他 `loadobject` 子命令中的 *regexp* 不同，如果未指定 *ex-regexp*，并不表示缺省情况下应用于所有装入对象。如果未指定 *ex-regexp*，该命令将列出先前的 `loadobject -exclude` 命令指定的排除模式。

如果指定 `-clear`，则删除排除模式列表。

目前这种功能不能用于防止装入 主程序或运行时链接程序。另外，如果使用它防止装入 C++ 运行时库，可能会导致某些 C++ 功能无法正常使用。

此选项不能与运行时检查 (RTC) 一起使用。

`loadobject -hide` 命令

`loadobject -hide` 命令用于从 `dbx` 的搜索算法中删除装入对象。

语法

```
loadobject -hide [regexp]
```

其中：

regexp 是正则表达式。如果未指定，则该命令应用于所有装入对象。

该命令从程序范围中删除装入对象，并隐藏其函数和符号，不让 dbx 得知。该命令也重置“预装入”位。有关更多信息，请在 dbx 提示符处键入以下命令，参阅 dbx 帮助文件。

```
(dbx) help loadobject preloading
```

loadobject -list 命令

loadobject -list 命令用于显示当前装入的装入对象。仅在本地模式中有效。

语法

```
loadobject -list [regexp] [-a]
```

其中：

regexp 是正则表达式。如果未指定，则该命令应用于所有装入对象。

显示每个装入对象的全路径名，并在旁边显示表示状态的字母。仅当指定了 -a 选项时，才会列出隐藏的装入对象。

h	此字母表示“隐藏”（whatis 或 stop in 之类的符号查询不会找到符号）。
u	如果有活动进程，u 表示“已取消映射”。
p	此字母表示预装入的装入对象，即程序中 loadobject -load 命令或 dlopen 事件的结果。

例如：

```
(dbx) lo -list libm
/usr/lib/64/libm.so.1
/usr/lib/64/libmp.so.2
(dbx) lo -list ld.so
h  /usr/lib/sparcv9/ld.so.1 (rtld)
```

上述示例表明，缺省情况下，运行时链接程序的符号处于隐藏状态。要在 dbx 命令中使用这些符号，请参见“loadobject -use 命令” [318]。

loadobject -load 命令

loadobject -load 命令用于为指定装入对象装入符号。仅在本地模式中有效。

语法

loadobject -load *load-object*

其中：

load-object 可以是全路径名，也可以是 /usr/lib、/usr/lib/sparcv9 或 /usr/lib/amd64 中的库。如果在调试某个程序，将仅搜索适当的 ABI 库目录。

loadobject -unload 命令

loadobject -unload 命令用于卸载指定的装入对象。仅在本地模式中有效。

语法

loadobject -unload [*regexp*]

其中：

regexp 是正则表达式。如果未指定，则该命令应用于所有装入对象。

该命令卸载与命令行中提供的 *regexp* 匹配的所有装入对象的符号。不能卸载使用 debug 命令装入的主程序。另外，dbx 可能会拒绝卸载当前正在使用或对 dbx 的正确运行至关重要的其他装入对象。

loadobject -use 命令

loadobject -use 命令用于向 dbx 的搜索算法中添加装入对象。仅在本地模式中有效。

语法

`loadobject -use [`*regexp*`]`

其中：

regexp 是正则表达式。如果未指定，则该命令应用于所有装入对象。

lwp 命令

`lwp` 命令用于列出或更改当前 LWP（lightweight process，轻量级进程）。仅在本地模式中有效。

注 - 只能在 Oracle Solaris 平台上执行 `lwp` 命令。

语法

`lwp`	显示当前 LWP。
`lwp` *lwp-ID*	切换到 LWP *lwp-ID*。
`lwp -info`	显示当前 LWP 的名称、位置和屏蔽信号。
`lwp [`*lwp-ID*`] -setfp` *address-expression*	向 `dbx` 告知 `fp` 寄存器具有 *address-expression* 值。不会更改正在调试的程序的状态。恢复执行时，使用 `-setfp` 选项设置的帧指针将被重置为其初始值。
`lwp [`*lwp-ID*`] -resetfp`	根据当前进程或信息转储文件中的寄存器值设置帧指针逻辑值，从而撤消前一个 `lwp -setfp` 命令的作用。

其中：

lwp-ID 是轻量级进程的标识符。

如果该命令与 LWP ID 和选项一起使用，则会对 *lwp-ID* 指定的 LWP 执行相应的操作，但是不会更改当前的 LWP。

当 LWP 的帧指针 (`fp`) 已损坏时，`-setfp` 和 `-resetfp` 选项很有用。这种情况下，`dbx` 无法正常重建调用堆栈以及对局部变量求值。这些选项在调试信息转储文件时发挥作用，此时 `assign $fp=...` 不可用。

要更改正在调试的应用程序可访问的 fp 寄存器，请使用 `assign $fp=`*address-expression* 命令。

lwps 命令

lwps 命令用于列出进程中的所有 LWP（lightweight process，轻量级进程）。仅在本地模式中有效。

注 - 只能在 Oracle Solaris 平台上执行 lwps 命令。

macro 命令

宏命令输出表达式的宏扩展。

语法

macro *expression*, ...

mmapfile 命令

mmapfile 命令用于查看核心转储中丢失的内存映射文件内容。仅在本地模式中有效。

Oracle Solaris 信息转储文件不包含只读内存段。可执行只读段（即文本）自动进行处理，dbx 通过查看可执行文件和相关共享对象对照这些段解决内存访问问题。

语法

mmapfile *mmapped-file address offset length*	查看从核心转储中丢失的内存映射文件的内容。

其中：

mmapped-file 是信息转储期间映射内存的文件的文件名。

address 是进程地址空间的起始地址。

length 是要查看的地址空间的字节长度。

offset 是以字节数表示的距离 *mmapped-file* 中起始地址的偏移量。

示例

只读数据段通常在应用程序内存映射数据库时出现。例如：

```
caddr_t vaddr = NULL;
off_t offset = 0;
size_t = 10 * 1024;
int fd;
fd = open("../DATABASE", ...)
vaddr = mmap(vaddr, size, PROT_READ, MAP_SHARED, fd, offset);
index = (DBIndex *) vaddr;
```

以下命令允许通过作为内存的调试器访问数据库：

```
mmapfile ../DATABASE $[vaddr] $[offset] $[size]
```

然后，按结构化方式查看数据库内容：

```
print *index
```

module 命令

`module` 命令用于读取一个或多个模块的调试信息。仅在本地模式中有效。

语法

`module [-v]`	输出当前模块的名称。
`module [-f] [-v] [-q] {`*name*` | -a}`	如果指定了 *name*，则读入称为 *name* 的模块的调试信息。如果指定了 `-a`，则读入所有模块的调试信息。

其中：

name 是要读取其调试信息的模块的名称。

`-a` 表示指定所有模块。

`-f` 强制读取调试信息，即使该文件比可执行文件新也是如此。请谨慎使用此选项！

`-v` 表示指定详细模式，用于输出语言、文件名等。

`-q` 表示指定静默模式。

modules 命令

modules 命令用于列出模块名。仅在本地模式中有效。

语法

modules [-v] [-debug |-read]	列出所有模块。 如果指定了 -debug，则列出包含调试信息的所有模块。 如果指定了 -read，则列出包含已读入的调试信息的模块的名称。

其中：

-v 表示指定详细模式，用于输出语言、文件名等。

native 命令

当 dbx 处于 Java 模式时，使用 native 命令表示将要执行指定命令的本机版本。如果在命令前加上 native，dbx 将以本地模式执行命令。这意味着按照 C 表达式或 C++ 表达式解释和显示表达式，且某些其他命令在该模式下生成的输出与在 Java 模式下的输出不同。

如果在调试 Java 代码的过程中要检查本机环境，此命令很有用。

语法

native *command*

其中：

command 是要执行的命令的名称和参数。

next 命令

next 命令用于单步执行一个源代码行（步过调用）。

dbx 环境变量 step_events（请参见“设置 dbxenv 变量” [54]）控制在单步执行期间是否启用断点。

本地模式语法

next	单步执行一行（步过调用）。对于多线程程序，步过函数调用时，为了避免死锁，会在该函数调用期间隐式恢复所有 LWP（lightweight process，轻量级进程）。非活动线程无法单步执行。
next *n*	单步执行 *n* 行（步过调用）。
next ... -sig *signal*	单步执行时传递指定信号。
next ... *thread-ID*	单步执行指定线程。
next ... *lwp-ID*	单步执行指定 LWP。步过函数时，不隐式恢复所有 LWP。

其中：

n 是要单步执行的行数。

signal 是信号名。

thread-ID 是线程 ID。

lwp-ID 是 LWP ID。

如果提供显式 *thread-id* 或 *lwp-ID*，则通用 next 命令的死锁避免措施不起作用。

有关计算机级步过调用，另请参见“nexti 命令” [324]。

注 - 有关轻量级进程 (lightweight process, LWP) 的信息，请参见 Oracle Solaris《多线程编程指南》。

Java 模式语法

next	单步执行一行（步过调用）。对于多线程程序，步过函数调用时，为了避免死锁，会在该函数调用期间隐式恢复所有 LWP（lightweight process，轻量级进程）。非活动线程无法单步执行。
next *n*	单步执行 *n* 行（步过调用）。

`next ...` *thread-ID*	单步执行指定线程。
`next ...` *lwp-ID*	单步执行指定 LWP。步过函数时，不隐式恢复所有 LWP。

其中：

n 是要单步执行的行数。

thread-ID 是线程标识符。

lwp-ID 是 LWP 标识符。

如果提供显式 *thread-ID* 或 *lwp-ID*，则通用 `next` 命令的死锁避免措施不起作用。

注 - 有关轻量级进程 (lightweight process, LWP) 的信息，请参见 Oracle Solaris《多线程编程指南》。

`nexti` 命令

`nexti` 命令单步执行一个计算机指令（步过调用）。仅在本地模式中有效。

语法

`nexti`	单步执行一个计算机指令（步过调用）。
`nexti` *n*	单步执行 *n* 个计算机指令（步过调用）。
`nexti -sig` *signal*	单步执行时传递指定信号。
`nexti ...` *lwp-ID*	单步执行指定 LWP。
`nexti ...` *thread-ID*	单步执行指定线程在其中处于活动状态的 LWP。步过函数时，不隐式恢复所有 LWP。

where:

n 是要单步执行的指令个数。

signal 是信号名。

thread-ID 是线程 ID。

lwp-ID 是 LWP ID。

omp_loop 命令

omp_loop 命令用于输出当前循环的描述，其中包括调度（静态、动态、指导、自动或运行时）、有序性、界限、步幅或跨距以及迭代数。只能从当前正执行循环的线程发出该命令。

omp_pr 命令

omp_pr 命令用于输出当前或指定并行区域的描述，其中包括父区域、并行区域 ID、组大小（线程数）及程序位置（程序计数器地址）。

语法

omp_pr	输出当前并行区域的描述。
omp_pr *parallel-region-ID*	输出指定并行区域的描述。此命令不会使 dbx 将当前并行区域切换为指定区域。
omp_pr -ancestors	输出所有并行区域的描述以及从当前并行区域到当前并行区域树根目录的路径。
omp_pr *parallel-region-ID* -ancestors	输出所有并行区域的描述以及从指定并行区域到其根目录的路径。
omp_pr -tree	输出整个并行区域树的描述。
omp_pr -v	输出当前并行区域的描述以及组成员信息。

omp_serialize 命令

omp_serialize 命令用于将当前线程或当前组中所有线程遇到的下一个并行区域的执行进行序列化。序列化仅适用于并行区域的此次行程，以后不会继续有效。

使用此命令时，请确保处于程序中的正确位置。合理的位置是指执行并行指令之前的点。

语法

`omp_serialize [-team]`	将当前线程遇到的下一个并行区域的执行序列化。 如果指定了 `-team`，则对当前组中的所有线程执行此操作。

omp_team 命令

`omp_team` 命令用于输出当前组中的所有线程。

语法

`omp_team [`*parallel-region-ID*`]`	输出当前组中的所有线程。 如果指定了并行区域 ID，则输出指定并行区域的组中的所有线程。

omp_tr 命令

`omp_tr` 命令用于输出当前任务区域的描述，其中包括任务区域 ID、类型（隐式或显式）、状态（已产生、正在执行或正在等待）、正在执行的线程、程序位置（程序计数器地址）、未完成的子项以及父项。

语法

`omp_tr`	输出当前任务区域的描述。
`omp_tr` *task-region-ID*	输出指定任务区域的描述。此命令不会使 `dbx` 将当前任务区域切换为指定任务区域。
`omp_tr -ancestors`	输出所有任务区域的描述以及从当前任务区域到当前任务区域树根目录的路径。
`omp_tr` *task-region-ID* `-ancestors`	输出所有任务区域的描述以及从指定任务区域到其根目录的路径。
`omp_tr -tree`	输出整个任务区域树的描述。

pathmap 命令

pathmap 命令用于将一个路径名映射到另一个路径名，以查找源文件等。该映射应用于源路径、对象文件路径和当前工作目录（如果指定 -c）。在宏略读过程中，也会应用映射来包括目录路径。在本地模式和 Java 模式中，pathmap 命令的语法和功能相同。

pathmap 命令对于处理在不同主机上具有不同路径的自动挂载和显式 NFS 挂载文件系统很有用。当前工作目录在自动挂载的文件系统中不准确。尝试解决由自动挂载程序引起的问题时，请指定 -c。如果原始树或构建树移动了，pathmap 命令也很有用。

缺省情况下，存在 pathmap /tmp_mnt /。

dbxenv 变量 core_lo_pathmap 设置为 on 时，可使用 pathmap 命令查找信息转储文件的装入对象。除此之外，pathmap 命令对查找装入对象（共享库）无效。有关更多信息，请参见“调试不匹配的信息转储文件” [37]。

语法

pathmap [-c] [-*index*] *from* *to*	建立从 *from* 到 *to* 的新映射。
pathmap [-c] [-*index*] *to*	将所有路径都映射到 *to*。
pathmap	列出所有现有路径映射（按索引）。
pathmap -s	相同，但 dbx 可以读取输出。
pathmap -d *from1* *from2* ...	删除指定映射（按路径）。
pathmap -d *index1 index2* ...	删除指定映射（按索引）。

其中：

from 和 *to* 是路径前缀。*from* 是指编译到可执行文件或对象文件中的路径，*to* 是指调试时的路径。

from1 是要删除的第一个映射的路径。

from2 是要删除的最后一个映射的路径。

index 指定映射插入列表中时采用的索引。如果未指定索引，映射将添加到列表末尾。

index1 是要删除的第一个映射的索引。

index2 是要删除的最后一个映射的索引。

如果指定 `-c`，映射也适用于当前工作目录。

如果指定 `-s`，则以 `dbx` 可以读取的输出格式列出现有映射。

如果指定 `-d`，则删除指定映射。

示例

```
(dbx) pathmap /export/home/work1 /net/mmm/export/home/work2
  # maps /export/home/work1/abc/test.c to /net/mmm/export/home/work2/abc/test.c
(dbx) pathmap /export/home/newproject
  # maps /export/home/work1/abc/test.c to /export/home/newproject/test.c
(dbx) pathmap
(1) -c /tmp_mnt /
(2) /export/home/work1 /net/mmm/export/home/work2
(3) /export/home/newproject
```

pop 命令

`pop` 命令用于从调用堆栈中删除一个或多个帧。仅在本地模式中有效。

弹出过程只能到达使用 `-g` 编译的函数的帧。程序计数器会重置为调用点的源代码行开头。无法弹出调试器调用的函数之后的帧；但必须使用 `pop -c`。

通常，`pop` 命令将调用与弹出帧关联的所有 C++ 析构函数。您可以通过将 `dbx` 环境变量 `pop_auto_destruct` 设置为 `off` 来覆盖此行为。

语法

`pop`	从堆栈中弹出当前顶部帧。
`pop` *number*	从堆栈中弹出 *number* 帧
`pop -f` *number*	从堆栈中弹出帧，直至达到指定的帧号 *number*。
`pop -c`	弹出从调试器中进行的最后一个调用。

其中：

number 是要从堆栈中弹出的帧数。

print 命令

在本地模式中，print 命令用于输出表达式的值。在 Java 模式中，print 命令输出表达式、局部变量或参数的值。

本地模式语法

`print` *expression*`, ...`	输出表达式 *expression*, ... 的值。
`print -r` *expression*	输出表达式 *expression* 的值，包括其继承成员。
`print +r` *expression*	`dbx` 环境变量 `output_inherited_members` 设置为 `on` 时，不输出继承成员。
`print -d [-r]` *expression*	显示动态类型而不是静态类型的表达式 *expression*。
`print +d [-r]` *expression*	`dbx` 环境变量 `output_dynamic_type` 为 `on` 时，不使用动态类型的表达式 *expression*。
`print -s` *expression*	输出当前 OpenMP 并列区域中每个线程的 *expression* 表达式的值（如果该表达式包含专用变量或线程专用变量）。
`print -S [-r] [-d]` *expression*	输出 *expression* 表达式的值，包括其静态成员（仅限于 C++）。
`print +S [-r] [-d]` *expression*	`dbxenv` 的变量 `show_static_members` 设置为 `on` 时不输出静态成员（仅限于 C++）。
`print -p` *expression*	调用 `prettyprint` 函数。
`print +p` *expression*	`dbx output_pretty_print` 环境变量设置为 on 时，不调用 `prettyprint` 函数。
`print -L` *expression*	如果输出对象 *expression* 大于 4K，强制输出。

`print +l` *expression*	如果表达式是一个字符串 (`char *`)，则只输出地址，不输出字符。
`print -l` *expression*	('Literal') 不输出左侧内容。如果表达式是一个字符串 (`char *`)，则不输出地址，只输出字符串的原始字符，且不带引号。
`print -f`*format* *expression*	使用 *format* 作为整数、字符串或浮点表达式的格式。
`print -F`*format* *expression*	使用指定格式但不输出左侧内容（变量名或表达式）。
`print -o` *expression*	以序数值形式输出必须是枚举的 *expression* 值。此处也可以使用格式字符串 (`-f`*format*)。对于非枚举型表达式，忽略此选项。
`print –m` *expression*	当 `dbxenv` 变量 `macro_expand` 设置为 `off` 时，将宏扩展应用于 *expression*。
`print +m` *expression*	当 `dbxenv` 变量 `macro_expand` 设置为 `on` 时，跳过表达式的宏扩展。
`print --` *expression*	"--" 表示标志参数的结尾。如果 *expression* 可以加号或减号开头，这很有用。有关作用域转换规则，请参见“程序作用域” [63]。

其中：

expression 是要输出其值的表达式。

format 是输出表达式时要使用的输出格式。如果该格式不适用于指定类型，则会默认忽略该格式字符串，`dbx` 将使用其内置输出机制。

允许的格式是 `printf`(3S) 命令使用的格式的子集。遵循以下约束：

- 没有 `n` 转换。
- 没有将 `*` 用于字段宽度和精度。
- 没有 `%<digits>$` 参数选择。
- 每个格式字符串只有一项转换规范说明。

  下列简单语法定义了允许的格式：

```
FORMAT ::= CHARS % FLAGS WIDTH PREC MOD SPEC CHARS

CHARS ::= <any character sequence not containing a %>

| %%

| <empty>

| CHARS CHARS

FLAGS ::= + | - | <space> | # | 0 | <empty>

WIDTH ::= <decimal_number> | <empty>
```

```
PREC ::= . | . <decimal_number> | <empty>

MOD ::= h | l | L | ll | <empty>

SPEC ::= d | i | o | u | x | X | f | e | E | g | G |

c | wc | s | ws | p
```

如果指定的格式字符串不包含 `%`，dbx 会自动预置一个。如果该格式字符串包含空格、分号或制表符，则整个格式字符串必须加上双引号。

Java 模式语法

`print` *expression*`, ...` `|` *...*	输出表达式 *expression*、. . . 或标识符 *identifier*, ... 的值
`print -r` *expression* `|` *identifier*	输出 *expression* 或 *identifier* 的值，包括其继承成员。
`print +r` *expression* `|` *identifier*	dbx 环境变量 `output_inherited_members` 设置为 `on` 时，不输出继承成员。
`print -d [-r]` *expression* `|` *identifier*	显示动态类型而不是静态类型 *expression* 或 *identifier*。
`print +d [-r]` *expression* `|` *identifier*	dbx 环境变量 `output_dynamic_type` 设置为 `on` 时，不使用动态类型的 *expression* 或 *identifier*。
`print --` *expression* `|` *identifier*	"--" 表示标志参数的结尾。如果 *expression* 可以加号或减号开头，这很有用。有关作用域转换规则，请参见“程序作用域” [63]。

其中：

class-name 是 Java 类。可使用下列任意值：

- 使用句点 (.) 作为限定符的软件包路径；例如，`test1.extra.T1.Inner`
- 前面带有英镑标记 (#) 并使用斜杠 (/) 和美元标记 ($) 作为限定符的全路径名。例如，`#test1/extra/T1$Inner`。如果使用 `$` 限定符，请使用引号将 *class-name* 引起来。

expression 是要输出其值的 Java 表达式。

field-name 是类中字段的名称。

identifier 是一个局部变量或参数，包括 `this`、当前类实例变量 (*object-name*.*field-name*) 或类（静态）变量 (*class-name*.*field-name*)。

object-name 是 Java 对象的名称。

proc 命令

`proc` 命令用于显示当前进程的状态。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`proc {-cwd | -map | -pid}`	如果指定了 `-cwd`，则显示当前进程的当前工作目录。 如果指定了 `-map`，则显示装入对象和地址列表。 如果指定了 `-process-ID`，则显示当前进程 ID（进程 ID）。

prog 命令

`prog` 命令用于管理调试的程序及其属性。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`prog -readsyms`	读取由于将 `dbx run_quick` 环境变量设置为 `on` 而延迟的符号信息。
`prog -executable`	输出可执行文件的完整路径，如果使用 `-` 连接了程序，则使用 `-`。
`prog -argv`	输出全部 `argv`，包括 `argv[0]`。
`prog -args`	输出 `argv`，不包括 `argv[0]`。
`prog -stdin`	输出 < *filename*；如果使用 `stdin`，则为空。
`prog -stdout`	输出 > *filename* 或 >> *filename*；如果使用 `stdout`，则为空。`-args`、`-stdin`、`-stdout` 的输出有特定格式，以便可以组合字符串并在 `run` 命令中重用。

quit 命令

quit 命令用于退出 dbx。在本地模式和 Java 模式中，它的语法和功能相同。

如果 dbx 连接到某个进程，在退出之前会分离该进程。如果有待决信号，它们将被取消。使用 detach 命令可进行更精细的控制。

语法

quit	退出 dbx 且返回代码 0。与 exit 相同。
quit *n*	退出且返回代码为 *n*。与 exit *n* 相同。

其中：

n 是返回代码。

regs 命令

regs 命令用于输出寄存器的当前值。仅在本地模式中有效。

语法

```
regs [-f] [-F]
```

其中：

-f 表示包括浮点寄存器（单精度）（仅限于 SPARC 平台）

-F 表示包括浮点寄存器（双精度）（仅限于 SPARC 平台）

示例（SPARC 平台）

```
dbx[13] regs -F
current thread: t@1
current frame:  [1]
g0-g3              0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7              0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3              0x00000003 0x00000014 0xef7562b4 0xeffff420
```

```
o4-o7             0xef752f80 0x00000003 0xefffff3d8 0x000109b8
l0-l3             0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7             0xefffff438 0x00000001 0x00000007 0xef74df54
i0-i3             0x00000001 0xefffff4a4 0xefffff4ac 0x00020c00
i4-i7             0x00000001 0x00000000 0xefffff440 0x000108c4
y                 0x00000000
psr               0x40400086
pc                0x000109c0:main+0x4    mov     0x5, %l0
npc               0x000109c4:main+0x8    st      %l0, [%fp - 0x8]
f0f1              +0.00000000000000e+00
f2f3              +0.00000000000000e+00
f4f5              +0.00000000000000e+00
f6f7              +0.00000000000000e+00
```

replay 命令

replay 命令用于重放自上次执行 run、rerun 或 debug 命令之后执行的调试命令。仅在本地模式中有效。

语法

replay [-*number*]	重放自上次执行 run 命令、rerun 命令或 debug 命令之后执行的所有命令或所有命令减去 *number* 个命令后所剩的命令。

其中：

number 是不重放的命令个数。

rerun 命令

rerun 命令用于在不使用参数的情况下运行程序。在本地模式和 Java 模式中，它的语法和功能相同。

语法

rerun	在不使用参数的情况下开始执行程序。
rerun *arguments*	使用 save 命令（请参见“save 命令” [338]）保存的新参数开始执行程序。

restore 命令

restore 命令用于将 dbx 恢复到先前保存的状态。仅在本地模式中有效。

语法

restore [*filename*]

其中：

filename 是文件名，该文件保存了自上次执行 run、rerun 或 debug 命令之后执行的 dbx 命令。

rprint 命令

rprint 命令用于采用 shell 引用规则输出表达式。仅在本地模式中有效。

语法

rprint [-r|+r|-d|+d|-S|+S|-p|+p|-L|-l|-f *format* | -F*format* | --] *expression*	输出表达式的值。没有应用特殊的引用规则，因此 rprint a > b 将 a 的值（如果存在）放入文件 b 中。有关这些标志的含义，请参见“print 命令” [329]。

其中：

expression 是要输出其值的表达式。

format 是输出表达式时要使用的输出格式。有关有效格式的信息，请参见“print 命令” [329]。

rtc showmap 命令

rtc showmap 命令用于报告按检测类型（分支和陷阱）分类的程序文本的地址范围。仅在本地模式中有效。

此命令适用于专家用户。运行时检查为访问检查检测程序文本。根据可用资源的情况，该检测类型可以是分支或陷阱指令。`rtc showmap` 命令用于报告按检测类型分类的程序文本的地址范围。此映射可用于查找最优位置以便添加修补程序区对象文件，以及用于避免自动使用陷阱。有关详细信息，请参见“运行时检查限制” [142]。

rtc skippatch 命令

`rtc skippatch` 命令用于使装入对象、对象文件和函数免受运行时检查对其进行检测。除非显式将装入对象卸载，否则该命令就会对每个 `dbx` 会话产生永久性的影响。

由于 `dbx` 不对该命令影响的装入对象、对象文件和函数中的内存访问进行跟踪，因此对于没有跳过的函数，会报告不正确的 `rui` 错误。`dbx` 无法确定 `rui` 错误是否由该命令引起，因此不会自动禁止此类错误。

语法

`rtc skippatch` *load-object* `[-o` *object-file* `...]` `[-f` *function* `...]`	使指定装入对象中的指定对象文件和函数免受检测。

其中：

load-object 是装入对象的名称或装入对象名称的路径。

object-file 是对象文件的名称。

function 是函数名。

run 命令

`run` 命令用于在使用参数的情况下运行程序。

使用 Ctrl-C 停止执行程序。

本地模式语法

`run`	使用当前参数开始执行程序。
`run` *arguments*	使用新参数开始执行程序。

`run ... >|>> output-file` 设置输出重定向。

`run ... < input-file` 设置输入重定向。

其中：

arguments 是运行目标进程时要使用的参数。

input-file 是要从中重定向输入的文件的文件名。

output-file 是要将输出重定向至的文件的文件名。

注 - 目前无法使用 `run` 或 `runargs` 命令重定向 `stderr`。

Java 模式语法

`run` 使用当前参数开始执行程序。

`run` *arguments* 使用新参数开始执行程序。

其中：

arguments 是运行目标进程时要使用的参数。这些参数传递给 Java 应用程序，而不是 JVM 软件。不要把主类名当作参数。

不能使用 `run` 命令重定向 Java 应用程序的输入或输出。

在某一运行中设置的断点将在后续运行中持续。

`runargs` 命令

`runargs` 命令用于更改目标进程的参数。在本地模式和 Java 模式中，它的语法和功能相同。

可以在不使用参数的情况下执行 `debug` 命令检查目标进程的当前参数。

语法

`runargs` *arguments* 设置要由 `run` 命令使用的当前参数（请参见“run 命令” [336]）。

`runargs ... >|` `>>`*file*	设置要由 `run` 命令使用的输出重定向。
`runargs ... <`*file*	设置要由 `run` 命令使用的输入重定向。
`runargs`	清除当前参数。

其中：

arguments 是运行目标进程时要使用的参数。

file 是目标进程的输出重定向到或从中重定向目标进程的输入的文件。

save 命令

`save` 命令由于将命令保存到文件中。仅在本地模式中有效。

语法

`save [ -`*number* `] [` *filename* `]`	将自上次执行 `run` 命令、`rerun` 命令或 `debug` 命令之后执行的所有命令或所有命令减去 *number* 个命令所剩的命令保存到缺省文件或 *filename* 中。

其中：

number 是不保存的命令个数。

filename 是文件名，该文件保存了自上次执行 `run`、`rerun` 或 `debug` 命令之后执行的 `dbx` 命令。

scopes 命令

`scopes` 命令用于输出活动作用域列表。仅在本地模式中有效。

search 命令

`search` 命令用于在当前源文件中向前搜索。仅在本地模式中有效。

语法

`search` *string*　　在当前文件中向前搜索 *string*。

`search`　　使用上一个搜索字符串重复搜索。

其中：

string 是要搜索的字符串。

showblock 命令

`showblock` 命令用于显示进行运行时检查时分配特定堆块的位置。仅在本地模式中有效。

启用了运行时检查时，`showblock` 命令将显示有关指定地址处堆块的详细信息。详细信息包括块分配的位置及其大小。

语法

`showblock -a` *address*

其中：

address 是堆块的地址。

showleaks 命令

注 - 只能在 Oracle Solaris 平台上执行 `showleaks` 命令。

在缺省的非冗余情况下，对于每个泄漏记录输出一行报告。先报告实际泄漏，然后再报告可能的泄漏。报告按泄漏的合并大小排序。

语法

`showleaks [-a] [-m` *m*`] [-n` *number*`] [-v]`

其中：

-a 表示显示迄今为止生成的所有泄漏（不仅仅是上次执行 showleaks 命令之后出现的泄漏）。

-m *m* 表示组合泄漏；如果两个或更多泄漏分配时的调用堆栈与 *m* 帧匹配，则在一个组合泄漏报告中报告这些泄漏。如果指定 -m 选项，它将覆盖通过 check 命令指定的 *m* 的全局值。

-n *number* 表示在报告中最多显示 *number* 个记录。缺省值为显示所有记录。

-v 表示生成详细输出。缺省值为显示非冗余输出。

showmemuse 命令

对于每个“使用中的块”记录均输出一行报告。命令将按块的合并大小排序报告。报告中还包括自上次执行 showleaks 命令以来的所有泄漏块。

语法

showmemuse [-a] [-m *m*] [-n *number*] [-v]

其中：

-a 显示所有使用的块（不仅仅是上次执行 showmemuse 命令之后使用的块）。

-m *m* 表示合并使用的块报告。*m* 的缺省值为 8 或上次通过 check 命令指定的全局值。如果两个或更多块分配时的调用堆栈与 *m* 帧匹配，则在一个合并报告中报告这些块。如果指定 -m 选项，则它将覆盖 *m* 的全局值。

-n *number* 表示在报告中最多显示 *number* 个记录。缺省值为 20。

-v 表示生成详细输出。缺省值为显示非冗余输出。

source 命令

source 命令用于执行指定文件中的命令。仅在本地模式中有效。

语法

`source` *filename*	执行文件 *filename* 中的命令。不搜索 `$PATH`。

status 命令

`status` 命令用于列出事件处理程序（断点等）。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`status`	输出正在使用的 `trace`、`when` 和 `stop` 断点。
`status` *handler-ID*	输出处理程序 *handler-ID* 的状态。
`status -h`	输出正在使用的 `trace`、`when` 和 `stop` 断点，其中包括隐藏的断点。
`status -s`	相同，但 `dbx` 可以读取输出。

其中：

handler-ID 是事件处理程序的标识符。

示例

```
(dbx) status -s > bpts
...
(dbx) source bpts
```

step 命令

`step` 命令用于单步执行源代码或语句（步入使用 `-g` 选项编译的调用）。

`dbx` 环境变量 `step_events` 控制单步执行过程中是否启用断点。

`dbx step_granularity` 环境变量控制源代码行单步执行的粒度。

dbx step_abflow 环境变量控制在检测到将要发生异常控制流更改时 dbx 是否停止。对 siglongjmp() 或 longjmp() 的调用或异常抛出有可能导致此类型的控制流更改。

本地模式语法

`step`	单步执行一行（步入调用）。对于多线程程序，步过函数调用时，为了避免死锁，会在该函数调用期间隐式恢复所有线程。非活动线程无法单步执行。
`step` *n*	单步执行 *n* 行（步入调用）。
`step up`	向下单步执行并步出当前函数。
`step ... -sig` *signal*	单步执行时传递指定信号。如果该信号存在信号处理程序，并且使用 `-g` 选项编译了该信号处理程序，则步入该信号处理程序。
`step ...`*thread-ID*	单步执行指定线程。不适用于 `step up`。
`step ...`*lwp-ID*	单步执行指定 LWP。步过函数时，不隐式恢复所有 LWP。

`step to [` *function* `]`	尝试步入从当前源代码行调用的 *function*。如果未指定 *function*，则步入上一个调用的函数，这有助于避免出现过长的 `step` 命令和 `step up` 命令序列。例如，上一个函数为：
	`f()->s()-t()->last();`
	`last(a() + b(c()->d()));`

其中：

n 是要单步执行的行数。

signal 是信号名。

thread-ID 是线程 ID。

lip-ID 是 LWP ID。

function 是函数名。

只有指定显式 *lwp*ID 时，通用 `step` 命令的死锁避免措施才不起作用。

在执行 `step to` 命令时，如果尝试步入上一个汇编调用指令或步入当前源代码行中的函数（如果指定），则调用可能因条件分支而无法进行。在不执行调用或当前源代码行中没有函数调用的情况下，`step to` 命令会步过当前源代码行。使用 `step to` 命令时，需要特别注意用户定义的运算符。

有关计算机级步进信息，另请参见“stepi 命令” [343]。

Java 模式语法

`step`	单步执行一行（步入调用）。对于多线程程序，步过方法调用时，为了避免死锁，会在该方法调用期间隐式恢复所有线程。非活动线程无法单步执行。
`step` *n*	单步执行 *n* 行（步入调用）。
`step up`	向下单步执行并步出当前方法。
`step ...`*thread-ID*	单步执行指定线程。不适用于 `step up`。
`step ...`*lwp-ID*	单步执行指定 LWP。步过方法时，不隐式恢复所有 LWP。

stepi 命令

`stepi` 命令用于单步执行一个计算机指令（步入调用）。仅在本地模式中有效。

语法

`stepi`	单步执行一个计算机指令（步入调用）。
`stepi` *n*	单步执行 *n* 个计算机指令（步入调用）。
`stepi -sig` *signal*	单步执行并传递指定信号。
`stepi ...`*lwp-ID*	单步执行指定 LWP。
`stepi ...`*thread-ID*	单步执行指定线程在其中处于活动状态的 LWP。

其中：

n 是要单步执行的指令个数。

signal 是信号名。

lwp-ID 是 LWP ID。

thread-ID 是线程 ID。

stop 命令

stop 命令用于设置源代码级断点。

语法

stop 命令的通用语法如下：

stop *event-specification* [*modifier*]

发生指定事件时，停止进程。

本地模式语法

本节介绍在本地模式下有效的其中一些较重要的语法。有关其他事件的信息，请参见“设置事件规范” [244]。

stop [-update]	立刻停止执行。仅在 when 命令的主体中有效。
stop -noupdate	立刻停止执行，但不更新 Oracle Developer Studio IDE 调试器窗口。
stop access *mode address-expression* [,*byte-size-expression*]	在访问了通过 *address-expression* 指定的内存时停止执行。另请参见“访问地址时停止执行” [91]。
stop at *line-number*	在 *line-number* 处停止执行。请参见“在源代码行设置断点” [88]。
stop change *variable*	*variable* 的值发生了更改时停止执行。
stop cond *condition-expression*	*condition-expression* 表示的条件的求值结果为 true 时停止执行。
stop in *function*	调用 *function* 时停止执行。请参见“在函数中设置断点” [88]。

`stop inclass` *class-name* `[ -recurse | -norecurse ]`	仅限于 C++：对类、结构、联合或模板类的所有成员函数设置断点。`-norecurse` 是缺省值。如果指定 `-recurse`，则包括基类。另请参见“在类的所有成员函数中设置断点” [90]。
`stop infile` *file-name*	调用 *filename* 中的任何函数时停止执行。
`stop infunction` *name*	仅限于 C++：对所有非成员函数 *name* 设置断点。
`stop inmember` *name*	仅限于 C++：对所有成员函数 *name.* 设置断点。请参见“在不同类的成员函数中设置断点” [90]。
`stop inobject` *object-expression* `[ -recurse | -norecurse ]`	仅限于 C++：在从对象 *object-expression* 调用类及其所有基类的任何非静态方法的入口处设置断点。`-recurse` 是缺省值。如果指定 `-norecurse`，则不包括基类。请参见“在对象中设置断点” [91]。

line-number 是源代码行的编号。

function 是函数名。

class-name 是 C++ 类、结构、联合或模板类的名称。

mode 指定内存访问模式。可由以下一个或所有字母组成：

`r`	已读取指定地址处的内存。
`w`	已写入内存。
`x`	已执行内存。

mode 也可以包含：

`a`	访问后停止进程（缺省值）。
`b`	访问前停止进程。

name 是 C++ 函数的名称。

object-expression 标识的是 C++ 对象。

variable 是变量名。

下列修饰符在本地模式中有效。

`-if` *condition-expression*	只有在 *condition-expression* 的求值结果为 true 时，才会发生指定的事件。

`-in` *function*	只有在 *function* 范围内发生指定事件时，执行才会停止。
`-count` *number*	从 0 开始，每次发生事件时，计数器的计数便会增加。达到 *number* 时，执行便停止，且计数器重置为 0。
`-count infinity`	从 0 开始，每次发生事件时，计数器的计数便会增加。执行没有停止。
`-temp`	创建发生事件时会删除的临时断点。
`-disable`	创建禁用状态的断点。
`-instr`	执行指令级变量。例如，`step` 变为指令级单步执行，而 `at` 使用文本地址而不是行号作为参数。
`-perm`	使相应事件在 `debug` 中保持持久性。有些事件（如断点）不适合保持持久性。`delete all` 不会删除持久性的处理程序。要删除持久性的处理程序，请使用 `delete` *hid*。
`-hidden`	通过 `status` 命令隐藏事件。一些导入模块可能会选择使用它。可使用 `status -h` 查看它们。
`-lwp` *lwp-ID*	只有在给定的 LWP 中发生指定的事件时，执行才会停止。
`-thread` *thread-ID*	只有在给定的线程中发生指定的事件时，执行才会停止。

Java 模式语法

下列特定语法在 Java 模式中有效。

`stop access` *mode class-name.field-name*	访问了 *class-name.field-name* 指定的内存时停止执行。
`stop at` *line-number*	在 *line-number* 处停止执行。
`stop at` *filename:line-number*	在 *filename* 中的 *line-number* 处停止执行。
`stop change` *class-name.field-name*	*class-name* 中的 *field-name* 值发生了更改时停止执行。
`stop classload`	在装入任何类时停止执行。

`stop classload` *class-name*	装入 *class-name* 时停止执行。
`stop classunload`	在卸载任何类时停止执行。
`stop classunload` *class-name*	卸载 *class-name* 时停止执行。
`stop cond` *condition-expression*	*condition-expression* 表示的条件的求值结果为 true 时停止执行。
`stop in` *class-name.method-name*	输入了 *class-name*.*method-name* 且将要执行第一行时停止执行。如果未指定参数且重载方法，则会显示方法列表。
`stop in` *class-name.method-name*`(`[*parameters*]`)`	输入了 *class-name*.*method-name* 且将要执行第一行时停止执行。
`stop inmethod` *class-name.method-name*	对所有非成员方法 *class-name*.*method-name* 设置断点。
`stop inmethod` *class-name.method-name* `(`[*parameters*]`)`	对所有非成员方法 *class-name*.*method-name* 设置断点。
`stop throw`	抛出了 Java 异常时停止执行。
`stop throw` *type*	抛出了 *type* 类型的 Java 异常时停止执行。

其中：

class-name 是 Java 类。可使用下列任意值：

- 使用句点 (.) 作为限定符的软件包路径；例如，`test1.extra.T1.Inner`
- 前面带有英镑标记 (#) 并使用斜杠 (/) 和美元标记 ($) 作为限定符的全路径名。例如，`#test1/extra/T1$Inner`。如果使用 `$` 限定符，请使用引号将 *class-name* 引起来。

condition-expression 可以是任何表达式，但其求值结果必须为整型。

field-name 是类中字段的名称。

filename 是文件名。

line-number 是源代码行的编号。

method-name 是 Java 方法的名称。

mode 指定内存访问模式。可由以下一个或所有字母组成：

`r`	已读取指定地址处的内存。
`w`	已写入内存。

mode 也可以包含：

`b`	访问前停止进程。

程序计数器将指向违例指令。

parameters 是方法的参数。

type 是 Java 异常的类型。`-unhandled` 或 `-unexpected` 是 *type* 的有效值。

下列修饰符在 Java 模式中有效：

`-if` *condition-expression*	只有在 *condition-expression* 的求值结果为 true 时，才会发生指定的事件。
`-count` *number*	从 0 开始，每次发生事件时，计数器的计数便会增加。达到 *number* 时，执行便停止，且计数器重置为 0。
`-count infinity`	从 0 开始，每次发生事件时，计数器的计数便会增加。执行没有停止。
`-temp`	创建发生事件时会删除的临时断点。
`-disable`	创建禁用状态的断点。

有关设置计算机级断点的信息，请参见“`stopi` 命令” [348]。

有关所有事件的列表和语法，请参见“设置事件规范” [244]。

`stopi` 命令

`stopi` 命令用于设置计算机级断点。仅在本地模式中有效。

语法

`stop` 命令的通用语法如下：

`stopi` *event-specification* [*modifier*]

发生指定事件时，停止进程。

下列特定语法有效：

`stopi at` *address-expression*	在 *address-expression* 位置处停止执行。
`stopi in` *function*	调用 *function* 时停止执行。

其中：

address-expression 是可产生地址或可用作地址的任何表达式。

function 是函数名。

有关所有事件的列表和语法，请参见“设置事件规范” [244]。

suppress 命令

`suppress` 命令用于禁止在运行时检查期间报告内存错误。仅在本地模式中有效。

如果 `dbx` 环境变量 `rtc_auto_suppress` 设置为 `on`，则只报告一次指定位置处的内存错误。

语法

`suppress`	`suppress` 和 `unsuppress` 命令（不包括指定 `-d` 和 `-reset` 选项的那些命令）的历史记录。
`suppress -d`	在未被编译用于调试的函数中禁止的错误列表（缺省禁止）。每个装入对象都有这样一个列表。只能通过使用带 `-d` 选项的 `unsuppress` 命令来取消禁止这些错误。
`suppress -d` *errors*	通过进一步抑制 *errors* 来修改所有装入对象的缺省禁止。
`suppress -d` *errors* `in` *load-objects*	通过进一步禁止 *errors* 来修改 *load-objects* 中的缺省禁止。

`suppress -last`	在错误位置禁止当前错误。
`suppress -reset`	将缺省禁止设置为初始值（启动时）。
`suppress -r` *ID*`...`	删除按 ID 指定的取消禁止事件，可通过 `unsuppress` 命令获取这些事件。
`suppress -r 0 | all | -all`	删除由 `unsuppress` 命令指定的所有取消禁止事件。
`suppress` *errors*	禁止各处的 *errors*。
`suppress` *errors* `in [` *functions* `] [` *files* `] [` *load-objects* `]`	禁止 *functions* 列表、*files,* 列表和 *load-objects.* 列表中的 *errors*。
`suppress` *errors* `at` *line*	禁止 *line* 处的 *errors*。
`suppress` *errors* `at "`*file*`":`*line*	禁止 *file* 中 *line* 处的 *errors*。
`suppress` *errors* `addr` *address*	禁止 *address* 位置处的 *errors*。

其中：

address 是内存地址。

errors 由空格分隔开，可以是下列各项的任一组合：

all	所有错误
`aib`	可能的内存泄漏 - 地址位于块内
`air`	可能的内存泄漏 - 地址位于寄存器内
`baf`	错误释放
`duf`	重复释放
`mel`	内存泄漏
`maf`	未对齐释放
`mar`	未对齐读

maw	未对齐写
oom	内存不足
rob	从数组越界内存中读
rua	从未分配的内存中读
rui	从未初始化的内存中读
wob	写入到数组越界内存
wro	写入到只读内存
wua	写入到未分配内存
biu	使用的块（分配的内存）。尽管不是错误，但可以在 suppress 命令中使用 biu，就像使用 *errors* 一样。

file 是文件名。

files 是一个或多个文件名。

functions 是一个或多个函数名。

line 是源代码行的编号。

load-objects 是一个或多个装入对象名。

有关抑制错误的更多信息，请参见“抑制错误” [131]。

有关取消禁止错误的信息，请参见“unsuppress 命令” [363]。

sync 命令

sync 命令用于显示有关指定同步对象的信息。仅在本地模式中有效。

注 - 只能在 Oracle Solaris 平台上执行 sync 命令。

语法

sync -info *address*	显示有关 *address* 的同步对象的信息。

其中：

address 是同步对象的地址。

syncs 命令

`syncs` 命令用于列出所有同步对象（锁）。仅在本地模式中有效。

注 - 只能在 Oracle Solaris 平台上执行 `syncs` 命令。

thread 命令

`thread` 命令用于列出或更改当前线程。

本地模式语法

`thread` 显示当前线程。

`thread` *thread-ID* 切换到线程 *thread-ID*。

在下列变量中，如果未指定线程 ID，则恢复当前线程。

`thread -info` [*thread-ID*] 输出有关指定线程的所有信息。对于 OpenMP 线程，这些信息包括 OpenMP 线程 ID、并行区域 ID、任务区域 ID 及线程状态。

`thread -hide` [*thread-ID*] 隐藏指定（或当前）线程。它将不会显示在通用线程列表中。

`thread -unhide` [*thread-ID*] 隐藏指定（或当前）线程。

`thread -unhide all` 取消隐藏所有线程。

`thread -suspend` *thread-ID* 阻止指定线程运行。在线程列表中挂起的线程标有 "S"。

`thread -resume` *thread-ID* 撤消使用 `-suspend` 执行的操作。

`thread -blocks [thread-ID]` 列出阻塞其他线程的指定线程所控制的所有锁。

`thread -blockedby [thread-ID]` 显示哪个同步对象阻塞了指定线程（如果有）。

其中：

thread-ID 是线程 ID。

Java 模式语法

`thread` 显示当前线程。

`thread thread-ID` 切换到线程 *thread-ID*。

在下列变量中，如果未指定线程 ID，则恢复当前线程。

`thread -info [thread-ID]` 输出有关指定线程的所有信息。

`thread -hide [thread-ID]` 隐藏指定（或当前）线程。它将不会显示在通用线程列表中。

`thread -unhide [thread-ID]` 隐藏指定（或当前）线程。

`thread -unhide all` 取消隐藏所有线程。

`thread -suspend thread-ID` 阻止指定线程运行。在线程列表中挂起的线程标有 "S"。

`thread -resume thread-ID` 撤消使用 `-suspend` 执行的操作。

`thread -blocks [thread-ID]` 列出 *thread-ID* 拥有的 Java 监视器。

`thread -blockedby [thread-id]` 列出阻塞的 *thread-ID* 所在的 Java 监视器。

其中：

thread-ID 是格式为 `t@`*number* 的 `dbx` 样式的线程 ID 或为线程指定的 Java 线程名称。

threads 命令

threads 命令用于列出所有线程。

本地模式语法

`threads`	输出所有已知线程列表。
`threads -all`	输出通常不输出的线程（僵停）。
`threads -mode all|filter`	控制是输出所有线程还是过滤线程。缺省值为过滤线程。启用了过滤时，不会列出已使用 `thread -hide` 命令隐藏的线程。
`threads -mode auto|manual`	在 IDE 下，启用自动更新线程列表。
`threads -mode`	回显当前模式。

每行信息由以下内容组成：

- *（星号），表示该线程内发生了需要用户注意的事件。该事件通常是断点。

  如果不是星号，而是 "o"，则表示发生的是 `dbx` 内部事件。
- >（箭头），表示当前线程。
- 线程 id `t@`*num*，指特定线程。*number* 是 `thr_create` 传回的 `thread_t` 值。
- `b l@`*num*，表示线程是绑定的（当前已分配给指定的 LWP）；或 `a l@`*num*，表示线程是活动的（当前已计划运行）。
- 传递给 `thr_create` 的线程的“启动函数”。`?()` 表示启动函数未知。
- 线程状态，可为下列之一：
  - `monitor`
  - `running`
  - `sleeping`
  - `unknown`
  - `wait`
  - `zombie`

  线程当前执行的函数。

Java 模式语法

`threads`	输出所有已知线程列表。

`threads -all`	输出通常不输出的线程（僵停）。
`threads -mode all|filter`	控制是输出所有线程还是过滤线程。缺省值为过滤线程。
`threads -mode auto|manual`	在 IDE 下，启用自动更新线程列表。
`threads -mode`	回显当前模式。

列表中的每行信息由以下内容组成：

- >（箭头），表示当前线程。
- `t@`*number*，`dbx` 样式的线程 ID
- 线程状态，可为下列之一：
  - `monitor`
  - `running`
  - `sleeping`
  - `unknown`
  - `wait`
  - `zombie`
- 加单引号的线程名
- 说明线程优先级的数字

trace 命令

`trace` 命令用于显示执行的源代码行、函数调用或变量更改。

可使用 `dbx` 环境变量 `trace_speed` 设置跟踪速度。

如果 `dbx` 处于 Java 模式下，而您要在本地模式中设置 trace 断点，请使用 `joff` 命令切换到本地模式，或为 `trace` 命令添加前缀 `native`。

如果 `dbx` 处于 JNI 模式下，而您要在 Java 代码中设置 trace 断点，请为 `trace` 命令添加前缀 `java`。

语法

`trace` 命令的通用语法如下：

`trace` *event-specification* [*modifier*]

发生指定事件时，跟踪即被输出。

本地模式语法

下列特定语法在本地模式中有效：

`trace -file` *filename*	将所有跟踪输出定向到指定的文件名。要将跟踪输出恢复为标准输出，请使用 - 表示 *filename*。跟踪输出始终附加到 *filename* 后面。每当显示 `dbx` 提示以及应用程序退出时都会刷新。在执行新的运行或连接后继续运行时总会重新打开文件。
`trace step`	跟踪每个源代码行、函数调用并返回。
`trace next -in` *function*	跟踪指定函数中每个源代码行。
`trace at` *line-number*	跟踪指定源代码行 *line*。
`trace in` *function*	跟踪对指定函数的调用及其返回。
`trace infile` *filename*	跟踪对 *filename* 中所有函数的调用及其返回。
`trace inmember` *function*	跟踪对所有名为 *function* 的成员函数的调用。
`trace infunction` *function*	调用任何名为 *function* 的函数时跟踪。
`trace inclass` *class*	跟踪对所有 *class* 的成员函数的调用。
`trace change` *variable*	跟踪对 *variable* 的更改。

其中：

filename 是跟踪输出要发送到的文件的名称。

function 是函数名。

line-number 是源代码行的编号。

class 是类名。

variable 是变量名。

下列修饰符在本地模式中有效。

`-if` *condition-expression*	只有在 *condition-expression* 的求值结果为 true 时，才会发生指定的事件。
`-in` *function*	只有在 *function* 中发生指定的事件时，执行才会停止。
`-count` *number*	从 0 开始，每次发生事件时，计数器的计数便会增加。达到 *number* 时，执行便停止，且计数器重置为 0。
`-count infinity`	从 0 开始，每次发生事件时，计数器的计数便会增加。执行没有停止。
`-temp`	创建发生事件时会删除的临时断点。
`-disable`	创建禁用状态的断点。
`-instr`	执行指令级变量。例如，`step` 变为指令级单步执行，而 `at` 使用文本地址而不是行号作为参数。
`-perm`	使相应事件在 `debug` 中保持持久性。有些事件（如断点）不适合保持持久性。`delete all` 不会删除持久性的处理程序。要删除持久性的处理程序，请使用 `delete` *hid*。
`-hidden`	通过 `status` 命令隐藏事件。一些导入模块可能会选择使用它。可使用 `status -h` 查看它们。
`-lwp` *lwp-ID*	只有在给定的 LWP 中发生指定的事件时，执行才会停止。
`-thread` *thread-ID*	只有在给定的线程中发生指定的事件时，执行才会停止。

Java 模式语法

下列特定语法在 Java 模式中有效。

`trace -file` *filename*	将所有跟踪输出定向到指定的 *filename*。要将跟踪输出恢复为标准输出，请使用 - 表示 *filename*。跟踪输出始终附加到 *filename* 后面。每当显示 `dbx` 提示以及应用程序退出时都会刷新。在执行新的运行或连接后继续运行时总会重新打开文件。
`trace at` *line-number*	跟踪 *line-number*。
`trace at` *filename.line-number*	跟踪指定的源代码 *filename.line-number*。

`trace in` *class-name.method-name*	跟踪对 *class-name. method-name* 的调用及其返回。
`trace in` *class-name.method-name*`([`*parameters*`]).`	跟踪对 *class-name.method-name*([*parameters*]) 的调用及其返回。
`trace inmethod` *class-name. method-name*	在调用任何名为 *class-name.method-name* 的方法时跟踪。
`trace inmethod` *class-name. method-name*`[(`*parameters*`)]`	在调用任何名为 *class-name.method-name* [(*parameters*)] 的方法时跟踪。

其中：

class_name 是 Java 类。可使用下列任意值：

- 使用句点 (.) 作为限定符的软件包路径；例如，`test1.extra.T1.Inner`
- 前面带有英镑标记 (#) 并使用斜杠 (/) 和美元标记 ($) 作为限定符的全路径名。例如，`#test1/extra/T1$Inner`。如果使用 `$` 限定符，请使用引号将 *class-name* 引起来。

filename 是文件名。

line-number 是源代码行的编号。

method-name 是 Java 方法的名称。

parameters 是方法的参数。

下列修饰符在 Java 模式中有效。

`-if` *condition-expression*	只有在 *condition-expression* 的求值求值结果为 true 时，才会发生指定事件并输出跟踪。
`-count` *number*	从 0 开始，每次发生事件时，计数器的计数便会增加。达到 *number* 时，便输出跟踪，且计数器重置为 0。
`-count infinity`	从 0 开始，每次发生事件时，计数器的计数便会增加。执行没有停止。
`-temp`	创建发生了事件且输出了跟踪时会删除的临时断点。如果 `-temp` 与 `-count` 一起使用，则仅当计数器重置为 0 时才删除断点。
`-disable`	创建禁用状态的断点。

有关所有事件的列表和语法，请参见“设置事件规范” [244]。

tracei 命令

tracei 命令用于显示计算机指令、函数调用或变量更改。仅在本地模式中有效。

tracei 实际上是 trace *event-specification* -instr 的简称，其中 -instr 修饰符表示在指令粒度而非源代码行粒度上进行跟踪。发生事件时，信息以反汇编格式而非源代码行格式输出。

语法

tracei step	跟踪每个计算机指令。
tracei next -in *function*	在指定函数中跟踪每个指令。
tracei at *address*	跟踪 *address* 处的指令。
tracei in *function*	跟踪对指定函数的调用及其返回。
tracei inmember *function*	跟踪对所有名为 *function* 的成员函数的调用。
tracei infunction *function*	调用任何名为 *function* 的函数时跟踪。
tracei inclass *class*	跟踪对所有 *class* 的成员函数的调用。
tracei change *variable*	跟踪对变量的更改。

其中：

address 是可产生地址或可用作地址的任何表达式。

filename 是跟踪输出要发送到的文件的名称。

function 是函数名。

line 是源代码行的编号。

class 是类名。

variable 是变量名。

有关更多信息，请参见“trace 命令” [355]。

uncheck 命令

uncheck 命令用于禁用检查内存访问、泄漏或使用。仅在本地模式中有效。

语法

`uncheck`	输出当前检查状态。
`uncheck -access`	禁用访问检查。
`uncheck -leaks`	禁用泄漏检查。
`uncheck -memuse`	禁用内存使用检查（同时禁用泄漏检查）。
`uncheck -all`	等效于 `uncheck -access`; `uncheck -memuse`。
`uncheck [`*functions*`] [`*files*`] [`*load-objects*`]`	等效于 *functions files load-objects.* 中的 `suppress all`。

其中：

functions 是一个或多个函数名。

files 是一个或多个文件名。

load-objects 是一个或多个装入对象名称。

有关开启检查的信息，请参见“check 命令” [276]。

有关抑制错误的信息，请参见“suppress 命令” [349]。

有关运行时检查的简介，请参见“运行时检查功能” [119]。

undisplay 命令

undisplay 命令用于撤消 display 命令。

本地模式语法

undisplay {*expression*, ... | *n* ...}	撤消 display *expression* 命令或编号为 *n*, ... 的 display 所有命令。如果 *n* 设置为零 (0)，则撤消所有显示命令。

其中：

expression 是有效的表达式。

Java 模式语法

undisplay *expression*, ... | *identifier*, ...	撤消 display *expression*, ... 或 display *identifier*, ... 命令。
undisplay *n*, ...	撤消编号为 *n*、...的 display 命令。
undisplay 0 执行所有 display 命令。	撤消所有 display 命令。

其中：

expression 是有效的 Java 表达式。

field-name 是类中字段的名称。

identifier 是一个局部变量或参数，包括 this、当前类实例变量 (*object-name*.*field-name*) 或类（静态）变量 (*class-name*.*field-name*)。

unhide 命令

unhide 命令用于撤消 hide 命令。仅在本地模式中有效。

语法

`unhide {`*regular-expression* `|` *number*`}`	删除堆栈帧过滤器 *regular-expression* 或删除编号为 *number* 的堆栈帧过滤器。 如果 *number* 设置为零 (0)，则删除所有堆栈帧过滤器。

其中：

regular-expression 是正则表达式。

number 是堆栈帧过滤器的编号。

`hide` 命令可列出具有编号的过滤器。

`unintercept` 命令

`unintercept` 命令用于撤消 `intercept` 命令（仅限于 C++）。仅在本地模式中有效。

语法

`unintercept` *intercepted-typename* `[,` *intercepted-typename* `... ]`	从 `intercept` 列表中删除类型为 *intercepted-typename* 的抛出。
`unintercept -a [ll]`	从 `intercept` 列表中删除所有类型的所有抛出。
`unintercept -x` *excluded-typename* `[,` *excluded-typename* `... ]`	从 `excluded` 列表中删除 *excluded-typename*。
`unintercept -x -a[ll]`	从 `excluded` 列表中删除所有类型的所有抛出。
`unintercept`	列出被拦截的类型。

其中：

included-typename 和 *excluded-typename* 是异常类型规范，如 `List <int>` 或 `unsigned short`。

unsuppress 命令

`unsuppress` 命令用于撤消 `suppress` 命令。仅在本地模式中有效。

语法

`unsuppress`	`suppress` 和 `unsuppress` 命令（不包括指定 `-d` 和 `-reset` 选项的那些命令）的历史记录。
`unsuppress -d`	未被编译用于调试的函数中取消禁止的错误列表。每个装入对象都有这样一个列表。任何其他错误都只能使用带 `-d` 选项的 suppress 命令来禁止。
`unsuppress -d` *errors*	通过进一步取消禁止 *errors* 来修改所有装入对象的缺省禁止。
`unsuppress -d` *errors* `in` *load-objects*	通过进一步取消禁止 *errors* 来修改 *load-objects* 中的缺省禁止。
`unsuppress -last`	在错误位置启用当前错误。
`unsuppress -reset`	将缺省禁止屏蔽设置为初始值（启动时）。
`unsuppress` *errors*	取消禁止各处的 *errors*。
`unsuppress` *errors* `in [`*functions*`] [`*filename ...*`] [`*load-objects*`]`	禁止函数列表、文件列表和装入对象列表中的 *errors*。
`unsuppress` *errors* `at` *line*	取消禁止 *line* 处的 *errors*。
`unsuppress` *errors* `at "`*filenames*`"` *line*	取消禁止 *filenames* 中 *line* 处的 *errors*。

`unsuppress` *errors* `addr` *address*	取消禁止 *address* 位置处的 *errors*。

其中：

errors 是一个或多个错误名。

functions 是一个或多个函数名。

filenames 是一个或多个文件名。

line 是行号。

load-objects 是一个或多个装入对象名称。

unwatch 命令

`unwatch` 命令用于撤消 `watch` 命令。仅在本地模式中有效。

语法

`unwatch` {*expression* `|` *n*`}`	撤消 `watch` *expression* 命令或编号为 *n* 的 `watch` 命令。 如果 *n* 设置为零 (0)，则撤消所有 `watch` 命令。

其中：

expression 是有效的表达式。

up 命令

`up` 命令用于上移调用堆栈（靠近 `main`）。在本地模式和 Java 模式中，它的语法和功能相同。

语法

`up [-h` [*number*`]]`	调用堆栈上移一级。 如果指定 *number*，则将调用堆栈上移 *number* 级。

如果指定 -h，则上移调用堆栈，但不跳过隐藏的帧。

其中：

number 是调用堆栈级数。

use 命令

use 命令用于列出或更改目录搜索路径。仅在本地模式中有效。

此命令已过时，该命令的用途已由下列 pathmap 命令实现：

use 等效于 pathmap -s

use *directory* 等效于 pathmap *directory*。

watch 命令

watch 命令用于在每个停止点处以相应点的当前作用域对表达式求值并进行输出。由于在到达入口点时不会解析表达式，因此无法立即检验表达式是否正确。watch 命令仅在本地模式中有效。

语法

watch	输出所显示的表达式列表。
watch [-r|+r|-d|+d|-S|+S|-p|+p|-L|-f*format*|-F*format*|-m|+m|--] *expression*	监视每个停止点处表达式 *expression* 的值。有关这些标志的含义，请参见“print 命令” [329]。

其中：

expression 是有效的表达式。

format 是输出表达式时要使用的输出格式。有关有效格式的信息，请参见“print 命令” [329]。

whatis 命令

在本地模式中，whatis 命令用于输出表达式的类型或类型声明或宏定义。如果适用，它还输出 OpenMP 数据共享属性信息。

在 Java 模式中，whatis 命令用于输出标识符声明。如果标识符是类，则它将输出类的方法信息（包括所有继承方法）。

本地模式语法

`whatis [-n] [-r] [-m] [+m]` *name*	输出非类型 *name* 的声明，或定义（如果 *name* 为宏）。
`whatis -t [-a] [-r] [-u]` *type*	输出类型为 *type* 的声明。
`whatis -e [-r] [-u] [-d]` *expression*	输出表达式 *expression* 的类型。

其中：

name 是非类型或宏的名称。

type 是类型名。

expression 是有效的表达式。

macro 是宏的名称。

-a 仅输出指定类的数据成员。

-d 显示动态类型而非静态类型。

-e 表示显示表达式的类型。

-n 表示显示非类型的声明。不必指定 -n；如果键入不带选项的 whatis 命令，缺省使用该值。

-r 输出有关基类和类型的信息。

-t 表示显示类型的声明。

-u 显示类型的根定义。

-m 强制进行宏扩展，即使 dbxenv 变量 macro_expand 设置为 off 也是如此。

+m 使宏查找失效，从而能找到可能被宏投影的任何符号。

对 C++ 类或结构运行 whatis 命令时，它将提供一个列表，其中列出所有已定义的成员函数、静态数据成员、类友元以及在该类中显式定义的数据成员。未定义的成员函数不列出。

如果指定 -r（递归）选项，将添加来自继承类的信息。

-d 标志与 -e 标志一起使用时表示使用动态类型的表达式。

对于 C++，模板相关标识符显示如下：

- 所有模板定义通过 whatis -t 列出。
- 函数模板实例通过 whatis 列出。
- 类模板实例通过 whatis -t 列出。

Java 模式语法

whatis *identifier*　　输出 *identifier* 的声明。

其中：

identifier 可以是类、当前类中的方法、当前帧中的局部变量或当前类中的字段。

when 命令

when 命令用于在发生指定事件时执行命令。

如果 dbx 处于 Java 模式下，而您要在本地代码中设置 when 断点，请使用 joff 命令切换到本地模式，或为 when 命令添加前缀 native。

如果 dbx 处于 JNI 模式下，而您要在 Java 代码中设置 when 断点，请为 when 命令添加前缀 java。

语法

when 命令的通用语法如下：

when *event-specification* [*modifier*]{*command*; ... }

发生指定事件时，执行上述命令。在 when 命令中禁止使用以下命令：

- attach

- debug
- next
- replay
- rerun
- restore
- run
- save
- step

如果 cont 命令不带选项，则会忽略该命令。

本地模式语法

下列特定语法在本地模式中有效：

when at *line-number* { *command;* }	达到 *line-number* 时，执行 *command*。
when in *procedure* { *command;* }	调用 *procedure* 时，执行 *command*。

其中：

line-number 是源代码行的编号。

command 是命令名。

procedure 是过程名。

Java 模式语法

下列特定语法在 Java 模式中有效。

when at *line-number*	达到源代码 *line-number* 时，执行命令。
when at *filename.line-number*	达到 *filename*.*line-number* 时，执行命令。
when in *class-name.method-name*	调用 *class-name*.*method-name* 时，执行命令。

`when in` *class-name.method-name*`([`*parameters*`])` 调用 *class-name*.*method-name*([*parameters*]) 时，执行命令。

class-name 是 Java 类。可使用下列任意值：

- 使用句点 (.) 作为限定符的软件包路径；例如，`test1.extra.T1.Inner`
- 前面带有英镑标记 (#) 并使用斜杠 (/) 和美元标记 ($) 作为限定符的全路径名。例如，`#test1/extra/T1$Inner`。如果使用 `$` 限定符，请使用引号将 *class-name* 引起来。

filename 是文件名。

line-number 是源代码行的编号。

method-name 是 Java 方法的名称。

parameters 是方法的参数。

有关所有事件的列表和语法，请参见“设置事件规范” [244]。

有关对指定的低级别事件执行命令的信息，请参见“wheni 命令” [369]。

wheni 命令

`wheni` 命令用于在发生指定的低级别事件时执行命令。仅在本地模式中有效。

语法

`wheni` *event-specification* `[`*modifier*`]{`*command*`... ; }`

发生指定事件时，执行上述命令。

下列特定语法有效：

`wheni at` *address* `{` *command;* `}` 达到 *address* 时，执行 *command*。

其中：

address 是可产生地址或可用作地址的任何表达式。

command 是命令名。

有关所有事件的列表和语法，请参见“设置事件规范” [244]。

where 命令

where 命令用于输出调用堆栈。对于 OpenMP 从属线程，如果相关帧仍处于活动状态，该命令还会输出主线程的堆栈跟踪。

本地模式语法

`where`	输出过程回溯。
`where` *number*	输出回溯中顶部的 *number* 帧。
`where -f` *number*	从第 *number* 帧开始回溯。
`where -fp` *address-expression*	认为 `fp` 寄存器中有 *address-expression* 值，并输出回溯。
`where -h`	包括隐藏的帧。
`where -l`	包括库名和函数名。
`where -q`	快速回溯（仅限于函数名）。
`where -v`	冗余回溯（包括函数参数和行信息）。

其中：

address-expression 是可产生地址或可用作地址的任何表达式。

number 是调用堆栈帧数。

上述任一选项都可以与线程 ID 或 LWP ID 一起使用，以便获得指定实体的回溯。

当 `fp`（帧指针）寄存器已损坏时（在这种情况下，`dbx` 无法正常重建调用堆栈），`-fp` 选项很有用。此选项为测试某值是否为正确的 `fp` 寄存器值提供了方便。一旦确定了此值为正确的值，就可以使用 `assign` 命令或 `lwp` 命令对其进行设置。

Java 模式语法

`where [`*thread-ID*`]`	输出方法回溯。

`where -f [`*thread-ID*`] `*number*	输出回溯中顶部的 *number* 帧。 如果指定了 f，则从第 *number* 帧开始回溯。
`where -q [`*thread-ID*`]`	快速回溯（仅限于方法名）。
`where -v [`*thread-ID*`]`	冗余回溯（包括方法参数和行信息）。

其中：

number 是调用堆栈帧数。

thread-ID 是 `dbx` 样式的线程 ID 或为线程指定的 Java 线程名称。

whereami 命令

`whereami` 命令用于显示当前源代码行。仅在本地模式中有效。

语法

`whereami`	显示与当前位置（堆栈顶部）对应的源代码行以及与当前帧对应的源代码行（如果不同）。
`whereami -instr`	与之前相同，只不过输出的是当前反汇编指令而不是源代码行。

whereis 命令

`whereis` 命令用于输出指定名称的所有使用情况或地址的符号名称。仅在本地模式中有效。

如果地址来自堆或堆栈，则 `whereis -a` 命令可以输出 address-expression 的位置。请注意，如果进程未处于活动状态或者 dbx 处理的是信息转储文件，则 dbx 无法输出位置。

语法

`whereis` *name*	输出 *name* 的所有声明。

`whereis -a` *address-expression*	输出 *address-expression* 的位置。

其中：

name 是作用域内可装入对象的名称，例如，变量、函数、类模板或函数模板。

address 是可产生地址或可用作地址的任何表达式。

`which` 命令

`which` 命令用于输出指定名称的全部限定。仅在本地模式中有效。

语法

`which [-n] [-m] [+m]` *name*	输出 *name* 的全部限定。
`which -t` *type*	输出 *type* 的全部限定。

其中：

name 是作用域内可装入对象的名称，例如，变量、函数、类模板或函数模板。

type 是类型名。

`-n` 表示显示非类型的全部限定。不必指定 `-n`；如果键入不带选项的 `which` 命令，缺省使用该值。

`-t` 表示显示类型的全部限定。

`-m` 强制进行宏查找，即使 `dbxenv` 变量 `macro_expand` 设置为 `off` 也是如此。

`+m` 使宏查找失效，从而能找到可能被宏投影的任何符号。

`whocatches` 命令

`whocatches` 命令用于指明捕获 C++ 异常的位置。仅在本地模式中有效。

语法

`whocatches` *type*

指明在当前执行点抛出类型为 *type* 的异常时捕获该异常的位置（如果无论如何都会捕获异常）。假定要执行的下一个语句是 `throw` *x*（其中 *x* 的类型为 *type*），将显示捕获它的 `catch` 子句的行号、函数名称和帧号。

如果捕获点位于执行抛出操作的同一个函数内，将返回 "*type* `is unhandled`"。

其中：

type 是异常的类型。

索引

D

E

F

J

M

N

O

P

Q

T

U

V

W

X

Y

Z