



使用 dbx 调试程序

Sun™ Studio 11

Sun Microsystems, Inc.
www.sun.com

文件号码 819-4755-10
2005 年 11 月, 修订版 A

请将有关本文档的意见和建议提交至: <http://www.sun.com/hwdocs/feedback>

版权所有 © 2005 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. 保留所有权利。

美国政府权利 - 商业用途。政府用户应遵循 Sun Microsystems, Inc. 的标准许可协议，以及 FAR（Federal Acquisition Regulations，即“联邦政府采购法规”）的适用条款及其补充条款。必须依据许可证条款使用。

本发行版可能包含由第三方开发的内容。

本产品的某些部分可能是从 Berkeley BSD 系统衍生出来的，并获得了加利福尼亚大学的许可。UNIX 是 X/Open Company, Ltd. 在美国和其他国家/地区独家许可的注册商标。

Sun、Sun Microsystems、Sun 徽标、Java 和 JavaHelp 是 Sun Microsystems, Inc. 在美国和其他国家/地区的商标或注册商标。所有的 SPARC 商标的使用均已获得许可，它们是 SPARC International, Inc. 在美国和其他国家/地区的商标或注册商标。标有 SPARC 商标的产品均基于由 Sun Microsystems, Inc. 开发的体系结构。

本服务手册所介绍的产品以及所包含的信息受美国出口控制法制约，并应遵守其他国家/地区的进出口法律。严禁将本产品直接或间接地用于核设施、导弹、生化武器或海上核设施，也不能直接或间接地出口给核设施、导弹、生化武器或海上核设施的最终用户。严禁出口或转口到美国禁运的国家/地区以及美国禁止出口清单中所包含的实体，包括但不限于被禁止的个人以及特别指定的国家/地区的公民。

本文档按“原样”提供，对于所有明示或默示的条件、陈述和担保，包括对适销性、适用性或非侵权性的默示保证，均不承担任何责任，除非此免责声明的适用范围在法律上无效。



Adobe PostScript

目录

- 阅读本书之前 25
 - 本书的结构 25
 - 印刷约定 26
 - Shell 提示符 27
 - 支持的平台 27
 - 访问 Sun Studio 软件和手册页 28
 - 访问 Sun Studio 文档 30
 - 访问相关的 Solaris 文档 33
 - 开发者资源 33
 - 联系 Sun 技术支持 34
 - Sun 欢迎您提出意见 34
- 1. dbx 入门 35
 - 编译调试代码 35
 - 启动 dbx 和加载程序 36
 - 在 dbx 中运行程序 38
 - 使用 dbx 调试程序 39
 - 检查核心文件 39
 - 设置断点 40
 - 单步执行程序 42

- 查看调用栈 43
- 检查变量 43
- 查找内存访问问题和内存泄漏 44
- 退出 dbx 44
- 访问 dbx 联机帮助 45
- 2. 启动 dbx 47**
 - 启动调试会话 47
 - 调试核心文件 48
 - 在相同的操作环境中调试核心文件 48
 - 如果核心文件被截断 49
 - 调试不匹配的核心文件 49
 - 使用进程 ID 52
 - dbx 启动序列 52
 - 设置启动属性 53
 - 将编译时目录映射到调试时目录 53
 - 设置 dbx 环境变量 54
 - 创建自己的 dbx 命令 54
 - 编译调试程序 54
 - 调试优化代码 55
 - 编译时未使用 -g 选项的代码 55
 - 共享库要求 -g 选项以获得完全 dbx 支持 56
 - 完全剥离的程序 56
 - 退出调试 56
 - 停止进程执行 56
 - 从 dbx 中分离进程 56
 - 中止程序而不终止会话 57
 - 保存和恢复调试运行 57
 - 使用 save 命令 57

- 将系列调试运行另存为检查点 59
- 恢复已保存的运行 59
- 使用 replay 恢复和保存 60

- 3. 自定义 dbx 61**
 - 使用 dbx 初始化文件 61
 - 创建 .dbxrc 文件 62
 - 初始化文件示例 62
 - 设置 dbx 环境变量 62
 - dbx 环境变量和 Korn Shell 67

- 4. 查看和导航到代码 69**
 - 导航到代码 69
 - 导航到文件 70
 - 导航到函数 70
 - 打印源码列表 71
 - 在调用栈中移动以导航到代码 71
 - 程序位置的类型 72
 - 程序作用域 72
 - 反映当前作用域的变量 72
 - 访问作用域 73
 - 使用作用域转换操作符限定符号 74
 - 反引号操作符 74
 - C++ 双冒号作用域转换操作符 75
 - 块局部操作符 75
 - 链接程序名 77
 - 查找符号 77
 - 打印符号具体值列表 77
 - 确定 dbx 使用哪个符号 78

- 作用域转换搜索路径 78
- 放松作用域查找规则 79
- 查看变量、成员、类型和类 79
 - 查找变量、成员和函数的定义 80
 - 查找类型和类的定义 81
- 在目标文件和可执行文件中调试信息 83
 - 目标文件装入 83
 - 列出模块的调试信息 84
 - 列出模块 85
- 查找源文件和目标文件 85
- 5. 控制程序执行 87**
 - 运行程序 87
 - 将 dbx 连接到正在运行的进程 88
 - 从进程中分离 dbx 89
 - 单步执行程序 90
 - 单步执行 90
 - 继续执行程序 91
 - 调用函数 92
 - 使用 Ctrl+C 停止进程 93
- 6. 设置断点和跟踪 95**
 - 设置断点 95
 - 在源代码行设置 stop 断点 96
 - 在函数中设置 stop 断点 97
 - 在 C++ 程序中设置多个断点 98
 - 设置数据更改断点 99
 - 在断点上设置过滤器 102
 - 跟踪执行 104

- 设置跟踪 104
- 控制跟踪速度 105
- 将跟踪输出定向到文件 105
- 在行中设置 when 断点 105
- 在共享库中设置断点 106
- 列出和清除断点 106
 - 列出断点和跟踪 106
 - 使用处理程序 ID 号删除特定断点 106
- 启用和禁用断点 107
- 效率方面的考虑 107
- 7. 使用调用栈 109**
 - 确定在栈中的位置 109
 - 栈中移动和返回起始位置 110
 - 在栈中上下移动 110
 - 栈中上移 110
 - 栈中下移 111
 - 移到特定帧 111
 - 弹出调用栈 111
 - 隐藏栈帧 112
 - 显示和读取栈跟踪 112
- 8. 求值和显示数据 115**
 - 求变量和表达式的值 115
 - 验证 dbx 使用的变量 115
 - 当前函数作用域之外的变量 116
 - 打印变量、表达式或标识符的值 116
 - 打印 C++ 116
 - 非关联化指针 117

- 监视表达式 118
- 关闭显示（取消显示） 118
- 给变量赋值 118
- 求数组的值 119
 - 数组分片 119
 - 数组片 122
 - 跨距 123
- 9. 使用运行时检查 125**
 - 运行时检查功能 126
 - 使用运行时检查的时机 126
 - 运行时检查要求 126
 - 局限性 127
 - 使用运行时检查 127
 - 打开内存使用和内存泄漏检查 127
 - 打开内存访问检查 127
 - 打开所有运行时检查 128
 - 关闭运行时检查 128
 - 运行程序 129
 - 使用访问检查 132
 - 理解内存访问错误报告 133
 - 内存访问错误 133
 - 使用内存泄漏检查 134
 - 检测内存泄漏错误 135
 - 可能的泄漏 135
 - 检查泄漏 136
 - 理解内存泄漏报告 136
 - 修复内存泄漏 139
 - 使用内存使用检查 139

禁止错误	140
禁止的类型	141
禁止错误示例	141
默认禁止	142
使用禁止来管理错误	143
对子进程使用运行时检查	143
对连接的进程使用运行时检查	147
同时使用修复并继续与运行时检查	148
运行时检查应用编程接口	149
在批处理模式下使用运行时检查	150
bcheck 语法	150
bcheck 示例	151
直接在 dbx 中启用批处理模式	151
疑难解答提示	152
运行时检查的 8 M 字节限制	152
运行时检查错误	153
访问错误	153
内存泄漏错误	156
10. 修复并继续	159
使用修复并继续	159
修复并继续如何操作	160
使用修复并继续修改源码	160
修复程序	160
修复后继续	161
修复后更改变量	162
修改头文件	164
修复 C++ 模板定义	164

- 11. 调试多线程应用程序 165**
 - 了解多线程调试 165
 - 线程信息 166
 - 查看另一线程的上下文 167
 - 查看线程列表 168
 - 恢复执行 168
 - 了解线程创建活动 169
 - 理解 LWP 信息 170

- 12. 调试子进程 171**
 - 连接到子进程 171
 - 跟随 exec 函数 172
 - 跟随 fork 函数 172
 - 与事件交互 172

- 13. 调试 OpenMP 程序 173**
 - 编译器如何转换 OpenMP 代码 174
 - OpenMP 代码可用的 dbx 功能 175
 - 使用带 OpenMP 代码的栈跟踪 175
 - 在 OpenMP 代码上使用 dump 命令 176
 - OpenMP 代码的执行序列 177

- 14. 处理信号 179**
 - 了解信号事件 179
 - 捕获信号 180
 - 更改默认信号列表 181
 - 捕获 FPE 信号（仅限 Solaris 平台） 181
 - 向程序发送信号 182
 - 自动处理信号 183

- 15. 使用 dbx 调试 C++ 185
 - 使用 dbx 调试 C++ 185
 - dbx 中的异常处理 186
 - 异常处理命令 186
 - 异常处理示例 188
 - 使用 C++ 模板调试 190
 - 模板示例 190
 - C++ 模板的命令 192

- 16. 使用 dbx 调试 Fortran 197
 - 调试 Fortran 197
 - 当前过程和文件 197
 - 大写字母 198
 - dbx 会话示例 198
 - 调试段故障 201
 - 使用 dbx 来找到故障 202
 - 定位异常 203
 - 跟踪调用 204
 - 处理数组 205
 - Fortran 95 可分配数组 206
 - 显示内在函数 207
 - 显示复数表达式 208
 - 显示区间表达式 209
 - 显示逻辑操作符 210
 - 查看 Fortran 95 派生类型 211
 - 指向 Fortran 95 派生类型的指针 212

- 17. 使用 dbx 调试 Java 应用程序 215
 - 使用 dbx 调试 Java 代码 215

- 使用 dbx 调试 Java 代码的功能 215
- 使用 dbx 调试 Java 代码的限制 216
- Java 调试的环境变量 216
- 开始调试 Java 应用程序 216
 - 调试类文件 217
 - 调试 JAR 文件 217
 - 调试有包装器的 Java 应用程序 218
 - 将 dbx 连接到正在运行的 Java 应用程序 218
 - 调试内嵌 Java 应用程序的 C 应用程序或 C++ 应用程序 219
 - 将参数传递给 JVM 软件 220
 - 指定 Java 源文件的位置 220
 - 指定 C 源文件或 C++ 源文件的位置 220
 - 为使用自定义类加载器的类文件指定路径 220
 - 在 JVM 软件尚未装入的代码上设置断点 220
- 自定义 JVM 软件的启动 221
 - 指定 JVM 软件的路径名 222
 - 将运行参数传递给 JVM 软件 222
 - 指定 Java 应用程序的自定义包装器 222
 - 指定 64 位 JVM 软件 224
- 调试 Java 代码的 dbx 模式 225
 - 从 Java 或 JNI 模式切换到本地模式 225
 - 中断执行时切换模式 225
- 在 Java 模式下使用 dbx 命令 226
 - dbx 命令中的 Java 表达式求值 226
 - dbx 命令使用的静态和动态信息 227
 - 在 Java 模式和本地模式下具有完全相同语法和功能的命令 227
 - 在 Java 模式下有不同语法的命令 228
 - 只在 Java 模式下有效的命令 229

- 18. 在机器指令级调试 231
 - 检查内存的内容 231
 - 使用 examine 或 x 命令 231
 - 使用 dis 命令 234
 - 使用 listi 命令 235
 - 在机器指令级单步执行和跟踪 236
 - 在机器指令级单步执行 236
 - 在机器指令级跟踪 236
 - 在机器指令级设置断点 238
 - 在地址处设置断点 238
 - 使用 adb 命令 239
 - 使用 regs 命令 239
 - 平台特定寄存器 242
 - x86 寄存器信息 243
 - AMD64 寄存器信息 245

- 19. 将 dbx 与 Korn Shell 配合使用 247
 - 未实现的 ksh-88 功能 247
 - ksh-88 的扩展 248
 - 重命名命令 248
 - 编辑函数的再绑定 249

- 20. 调试共享库 251
 - 动态链接程序 251
 - 链接映射 251
 - 启动序列和 .init 段 252
 - 过程链接表 252
 - 修复并继续 252
 - 在共享库中设置断点 252

在显式装入的库中设置断点 252

A. 修改程序状态 255

在 dbx 下运行程序的影响 255

更改程序状态的命令 256

assign 命令 256

pop 命令 256

call 命令 257

print 命令 257

when 命令 257

fix 命令 258

cont at 命令 258

B. 事件管理 259

事件处理程序 259

创建事件处理程序 260

操作事件处理程序 260

使用事件计数器 261

设置事件规范 261

断点事件规范 261

数据更改事件规范 263

系统事件规范 264

执行进度事件规范 267

其他事件规范 268

事件规范修饰符 271

分析和二义性 273

使用预定义变量 274

对 when 命令有效的变量 275

对特定事件有效的变量 276

设置事件处理程序示例	277
为存储到数组成员设置断点	277
执行简单跟踪	277
在函数内时启用处理程序 (<i>in function</i>)	277
确定已执行的行数	278
确定源代码行执行的指令数	278
事件发生后启用断点	279
为 <code>replay</code> 而重置应用程序文件	279
检查程序状态	279
捕获浮点异常	280

C. 命令参考 281

adb 命令	281
assign 命令	281
attach 命令	282
bsearch 命令	283
call 命令	284
cancel 命令	285
catch 命令	285
check 命令	286
clear 命令	288
collector 命令	288
collector archive 命令	290
collector dbxsample 命令	290
collector disable 命令	290
collector enable 命令	290
collector heaptrace 命令	291
collector hwprofile 命令	291
collector limit 命令	292

collector mpitrace 命令 292
collector pause 命令 292
collector profile 命令 292
collector resume 命令 293
collector sample 命令 293
collector show 命令 294
collector status 命令 294
collector store 命令 294
collector synctrace 命令 295
collector version 命令 295
cont 命令 296
dalias 命令 296
dbx 命令 297
dbxenv 命令 299
debug 命令 300
delete 命令 302
detach 命令 303
dis 命令 303
display 命令 304
down 命令 305
dump 命令 306
edit 命令 306
examine 命令 307
exception 命令 308
exists 命令 308
file 命令 309
files 命令 309
fix 命令 310

fixed 命令 312
frame 命令 312
func 命令 313
funcs 命令 314
gdb 命令 315
handler 命令 315
hide 命令 316
ignore 命令 317
import 命令 317
intercept 命令 318
java 命令 319
jclasses 命令 319
joff 命令 319
jon 命令 320
jpkgs 命令 320
kill 命令 320
language 命令 321
line 命令 322
list 命令 323
listi 命令 325
loadobject 命令 325
 loadobject -dumpelf 命令 326
 loadobject -exclude 命令 326
 loadobject -hide 命令 327
 loadobject -list 命令 327
 loadobject -load 命令 328
 loadobject -unload 命令 328
 loadobject -use 命令 328

lwp 命令 329
lwps 命令 329
mmapfile 命令 330
module 命令 331
modules 命令 332
native 命令 332
next 命令 333
nexti 命令 334
pathmap 命令 335
pop 命令 336
print 命令 337
proc 命令 340
prog 命令 340
quit 命令 341
regs 命令 341
replay 命令 342
rerun 命令 343
restore 命令 343
rprint 命令 343
rtc -showmap 命令 344
run 命令 345
runargs 命令 346
save 命令 346
scopes 命令 347
search 命令 347
showblock 命令 347
showleaks 命令 348
showmemuse 命令 349

source 命令 349
status 命令 350
step 命令 350
stepi 命令 352
stop 命令 353
stopi 命令 358
suppress 命令 358
sync 命令 361
syncs 命令 361
thread 命令 361
threads 命令 363
trace 命令 365
tracei 命令 368
uncheck 命令 369
undisplay 命令 370
unhide 命令 371
unintercept 命令 371
unsuppress 命令 372
up 命令 373
use 命令 373
whatis 命令 373
when 命令 375
wheni 命令 376
where 命令 377
whereami 命令 378
whereis 命令 379
which 命令 379
whocatches 命令 380

索引 381

图

-
- 图 8-1 跨距为 1 的二维矩形数组片示例 122
- 图 8-2 跨距为 2 的二维矩形数组片示例 123
- 图 14-1 截取和取消 SIGINT 信号 180

表

表 3-1	dbx 环境变量	63
表 11-1	线程和 LWP 状态	166
表 B-1	对 sig 事件有效的变量	276
表 B-2	对 exit 事件有效的变量	276
表 B-3	对 dlopen 和 dlclose 事件有效的变量	276
表 B-4	对 sysin 和 sysout 事件有效的变量	276
表 B-5	对 proc_gone 事件有效的变量	276

阅读本书之前

dbx 命令行调试器是一种交叉源码级调试工具。《使用 dbx 调试程序》适用于希望使用 dbx 命令调试应用程序的编程人员。使用 dbx 的用户需要具备 Fortran、C 或 C++ 的使用经验，并对 Solaris™ 操作系统 (Solaris Operating System, Solaris OS) 或 Linux 操作系统以及 UNIX® 命令都有一定了解。

本书的结构

《使用 dbx 调试程序》包含以下章节和附录：

第 1 章提供了使用 dbx 调试应用程序的基本知识。

第 2 章介绍了如何编译用于调试的程序以及如何启动调试会话。

第 3 章介绍了如何自定义调试环境。

第 4 章介绍如何访问源文件和函数、如何查找符号，以及如何查找变量、成员、类型和类。

第 5 章介绍了如何在 dbx 下运行程序，其中包括如何连接正在运行的程序。

第 6 章介绍了如何设置、清除及列出断点和跟踪。

第 7 章介绍如何检查调用栈。

第 8 章说明如何求数据的值及如何显示表达式、变量和其他数据结构的值，以及如何赋值给变量。

第 9 章介绍如何使用运行时检查功能自动检测程序中的内存泄漏和内存访问错误。

第 10 章介绍 dbx 的“修复并继续”功能。

第 11 章介绍如何查找有关线程的信息。

第 12 章介绍几个帮助调试子进程的 dbx 工具。

第 13 章介绍如何使用 `dbx` 来调试 OpenMP™ 代码。

第 14 章介绍如何使用 `dbx` 处理信号。

第 15 章介绍 C++ 模板的 `dbx` 支持、用于处理 C++ 异常的命令以及 `dbx` 如何处理这些异常。

第 16 章介绍一些可用来调试 Fortran 程序的 `dbx` 工具。

第 17 章介绍如何使用 `dbx` 来调试由 Java™ 代码和 C JNI (Java™ 本地接口) 代码或 C++ JNI 代码混和而成的应用程序。

第 18 章介绍如何在机器指令级使用 `dbx`

第 19 章说明 `ksh-88` 与 `dbx` 命令之间的区别。

第 20 章介绍 `dbx` 对使用动态链接共享库的程序的的支持。

附录 A 着重介绍在 `dbx` 下运行程序时能够更改程序或其行为的 `dbx` 命令。

附录 B 介绍如何在调试的程序中管理事件。

附录 C 提供所有 `dbx` 命令的详细语法和功能说明。

印刷约定

表 P-1 字体约定

字体*	含义	示例
AaBbCc123	命令、文件和目录的名称；计算机屏幕输出。	编辑 <code>.login</code> 文件。 使用 <code>ls -a</code> 列出所有文件。 % You have mail.
AaBbCc123	用户键入的内容，与计算机屏幕输出的显示不同。	% su Password:
AaBbCc123	保留未译的新词或术语以及要强调的词。要使用实名或值替换的命令行变量。	这些称为 <i>class</i> 选项。 要删除文件，请键入 rm filename 。
新词术语强调	新词或术语以及要强调的词。	您 必须 成为超级用户才能执行此操作。
《书名》	书名	阅读《用户指南》的第 6 章。

* 浏览器的设置可能会与这些设置不同。

表 P-2 代码约定

代码符号	含义	表示法	代码示例
[]	方括号中包含可选参数。	O[n]	-O4, -O
{ }	花括号中包含所需选项的选项集合。	d{y n}	-dy
	分隔变量的“ ”或“-”符号，只能选择其一。	B{dynamic static}	-Bstatic
:	与逗号一样，分号有时可用于分隔参数。	Rdir[:dir]	-R/local/libs:/U/a
...	省略号表示一系列的省略。	-xinline=f1[,...fn]	-xinline=alpha,dos

Shell 提示符

Shell	提示符
C shell	<i>machine-name%</i>
C shell 超级用户	<i>machine-name#</i>
Bourne shell、Korn shell 和 GNU Bourne-Again shell	\$
Bourne shell、Korn shell 和 GNU Bourne-Again shell 的超级用户	#

支持的平台

此 Sun Studio 发行版本支持使用 SPARC® 和 x86 系列处理器体系结构（UltraSPARC®、SPARC64、AMD64、Pentium 和 Xeon EM64T）的系统。通过访问 <http://www.sun.com/bigadmin/hcl> 中的硬件兼容性列表，可以了解您在使用的 Solaris 操作系统版本的支持系统。这些文档列出了实现各个平台类型的所有差别。

在本文档中，这些与 x86 有关的术语具有以下含义：

- “x86”是指较大的 64 位和 32 位 x86 兼容产品系列。
- “x64”表示有关 AMD64 或 EM64T 系统的特定 64 位信息。

- “32 位 x86”表示有关基于 x86 的系统的特定 32 位信息。
有关所支持的系统，请参见硬件兼容性列表。

访问 Sun Studio 软件和手册页

Sun Studio 软件及其手册页未安装到 `/usr/bin/` 和 `/usr/share/man` 标准目录中。要访问软件，必须正确设置 `PATH` 环境变量（请参见第 28 页的“访问软件”）。要访问手册页，必须正确设置 `MANPATH` 环境变量（请参见第 29 页的“访问手册页”）。

有关 `PATH` 变量的详细信息，请参见 `csh(1)`、`sh(1)`、`ksh(1)` 和 `bash(1)` 手册页。有关 `MANPATH` 变量的详细信息，请参见 `man(1)` 手册页。有关设置 `PATH` 变量和 `MANPATH` 变量以访问此发行版本的详细信息，请参见安装指南或询问系统管理员。

注 – 本节中的信息假设 Sun Studio 软件安装在 Solaris 平台上的 `/opt` 目录中和 Linux 平台上的 `/opt/sun` 目录中。如果未将软件安装在默认目录中，请咨询系统管理员以获取系统中的相应路径。

访问软件

使用以下步骤决定是否需要更改 `PATH` 变量以访问该软件。

决定是否需要设置 `PATH` 环境变量

1. 通过在命令提示符后键入以下内容以显示 `PATH` 变量的当前值。

```
% echo $PATH
```

2. 在 Solaris 平台上，查看输出中是否包含有 `/opt/SUNWspro/bin` 的路径字符串。在 Linux 平台上，查看输出中是否包含有 `/opt/sun/sunstudio11/bin` 的路径字符串。

如果找到该路径，则说明已设置了访问该软件的 `PATH` 变量。如果没有找到该路径，则需要按照下一步的说明来设置 `PATH` 环境变量。

设置 PATH 环境变量以实现软件的访问

- 在 **Solaris** 平台上, 将以下路径添加到 PATH 环境变量中。如果以前安装了 **Forte Developer** 软件、**Sun ONE Studio** 软件, 或其他发行版本的 **Sun Studio** 软件, 则将以下路径添加到这些安装路径之前。

```
/opt/SUNWspro/bin
```

- 在 **Linux** 平台上, 将以下路径添加到 PATH 环境变量中。

```
/opt/sun/sunstudio11/bin
```

访问手册页

使用以下步骤决定是否更改 MANPATH 变量以访问手册页。

决定是否设置 MANPATH 环境变量

1. 通过在命令提示符后输入以下内容以请求 dbx 手册页。

```
% man dbx
```

2. 请查看输出 (如果有)。

如果找不到 dbx(1) 手册页或者显示的手册页不是软件当前版本的手册页, 请按照下一步的说明来设置 MANPATH 环境变量。

设置 MANPATH 环境变量以实现对手册页的访问

- 在 **Solaris** 平台上, 将以下路径添加到 MANPATH 环境变量中。

```
/opt/SUNWspro/man
```

- 在 **Linux** 平台上, 将以下路径添加到 MANPATH 环境变量中。

```
/opt/sun/sunstudio11/man
```

访问集成开发环境

Sun Studio 集成开发环境 (integrated development environment, IDE) 提供了创建、编辑、生成、调试 C、C++ 或 Fortran 应用程序并分析其性能模块。

启动 IDE 的命令是 `sunstudio`。有关该命令的详细信息, 请参见 `sunstudio(1)` 手册页。

IDE 是否可以正确操作取决于 IDE 能否找到核心平台。sunstudio 命令查找两个位置的核心平台：

- 该命令首先查找 Solaris 平台上的默认安装目录 /opt/netbeans/3.5V11 和 Linux 平台上的默认安装目录 /opt/sun/netbeans/3.5V11。
- 如果该命令在默认目录找不到核心平台，则它会假设包含 IDE 的目录和包含核心平台的目录均安装在同一位置上。例如，在 Solaris 平台上，如果包含 IDE 的目录的路径是 /foo/SUNWspr0，则该命令会在 /foo/netbeans/3.5V11 中查找核心平台。在 Linux 平台上，如果包含 IDE 的目录的路径是 /foo/sunstudio11，则该命令会在 /foo/netbeans/3.5V11 中查找核心平台。

如果核心平台未安装在 sunstudio 命令查找它的任一位置上，则客户端系统上的每个用户必须将环境变量 SPRO_NETBEANS_HOME 设置为安装核心平台的位置 (/installation_directory/netbeans/3.5V11)。

在 Solaris 平台上，IDE 的每个用户也必须将 /installation_directory/SUNWspr0/bin 添加到其他任何 Forte Developer 软件、Sun ONE Studio 软件或 Sun Studio 软件发行版本路径前面的 \$PATH 中。在 Linux 平台上，IDE 的每个用户也必须将 /installation_directory/sunstudio11/bin 添加到其他任何发行版本的 Sun Studio 软件路径前面的 \$PATH 中。

路径 /installation_directory/netbeans/3.5V11/bin 不能添加到用户的 \$PATH 中。

访问 Sun Studio 文档

您可以访问以下位置的文档：

- 可以通过随软件一起安装在本地系统或网络上的文档索引获取文档，位置为 Solaris 平台上的 file:/opt/SUNWspr0/docs/zh/index.html 和 Linux 平台上的 file:/opt/sun/sunstudio11/docs/zh/index.html。

如果未将软件安装在 Solaris 平台上的 /opt 目录中或 Linux 平台上的 /opt/sun 目录中，请咨询系统管理员以获取系统中的相应路径。

- 大多数的手册都可以从 docs.sun.comsm Web 站点获取。以下书目只能从 Solaris 平台上安装的软件中找到：
 - 《标准 C++ 库类参考》
 - 《标准 C++ 库用户指南》
 - 《Tools.h++ 类库参考》
 - 《Tools.h++ 用户指南》
- 适用于 Solaris 平台和 Linux 平台的发行说明可以通过 docs.sun.com Web 站点获取。
- 在 IDE 中通过“帮助”菜单以及许多窗口和对话框上的“帮助”按钮，可以访问 IDE 所有组件的联机帮助。

您可以通过 Internet 访问 docs.sun.com Web 站点 (<http://docs.sun.com>) 以阅读、打印和购买 Sun Microsystems 的各种手册。如果找不到手册，请参见与软件一起安装在本地系统或网络中的文档索引。

注 – Sun 对本文档中提到的第三方 Web 站点的可用性不承担任何责任。对于此类站点或资源中的（或通过它们获得的）任何内容、广告、产品或其他资料，Sun 并不表示认可，也不承担任何责任。对于因使用或依靠此类站点或资源中的（或通过它们获得的）任何内容、物品或服务而造成的或连带产生的实际或名义损坏或损失，Sun 概不负责，也不承担任何责任。

使用易读格式的文档

该文档以易读格式提供，以方便残障用户使用辅助技术进行阅读。您还可以按照下表所述，找到文档的易读版本。如果未将软件安装在 /opt 目录中，请咨询系统管理员以获取系统中的相应路径。

文档类型	易读版本的格式和位置
手册（第三方手册除外）	HTML，位于 http://docs.sun.com
第三方手册： <ul style="list-style-type: none">• 《标准 C++ 库类参考》• 《标准 C++ 库用户指南》• 《Tools.h++ 类库参考》• 《Tools.h++ 用户指南》	HTML，位于 Solaris 平台上所安装软件中的文档索引 <code>file:/opt/SUNWspro/docs/zh/index.html</code>
自述文件	HTML，位于开发者门户网站 http://developers.sun.com/prodtech/cc/documentation/ss11/cn/docs/mr/READMEs 中
手册页	HTML，位于安装的软件上的文档索引，位置为 Solaris 平台上的 <code>file:/opt/SUNWspro/docs/zh/index.html</code> 和 Linux 平台上的 <code>file:/opt/sun/sunstudio11/docs/zh/index.html</code> 。
联机帮助	HTML，可通过 IDE 中的“帮助”菜单和“帮助”按钮访问
发行说明	HTML，位于 http://docs.sun.com

相关文档

对于 Solaris 平台，下表描述的相关文档可以在 [/opt/SUNWspr/docs/zh/index.html](http://opt/SUNWspr/docs/zh/index.html) 和 <http://docs.sun.com> 上获取。如果未将软件安装在 `/opt` 目录中，请咨询系统管理员以获取系统中的相应路径。

文档标题	描述
dbx 自述文件	列出 dbx 的新增功能、已知问题、限制和不兼容性。
dbx(1) 手册页	介绍 dbx 命令。
《C 用户指南》	介绍 Sun Studio 11 C 编程语言编译器和 ANSI C 编译器特定信息。
《C++ 用户指南》	指导您使用 Sun Studio 11 C++ 编译器，并提供有关命令行编译器选项的详细信息。
《Fortran 用户指南》	介绍 Sun Studio 11 Fortran 编译器的编译时环境以及命令行选项。
《OpenMP API 用户指南》	概述生成多重处理应用程序的 OpenMP Fortran 95、C 和 C++ 应用程序接口 (application program interface, API)。Sun Studio 编译器支持 OpenMP API。
《性能分析器》	介绍随 Sun Studio 11 提供的性能分析工具

对于 Linux 平台，下表描述的相关文档可以在 file:/opt/sun/sunstudio10u1/docs/index.html 和 <http://docs.sun.com> 上获取。如果未将软件安装在 `/opt/sun` 目录中，请咨询系统管理员以获取系统中的相应路径。

文档标题	描述
dbx 自述文件	列出 dbx 的新增功能、已知问题、限制和不兼容性。
dbx(1) 手册页	介绍 dbx 命令。
《性能分析器》	介绍随 Sun Studio 11 提供的性能分析工具。

访问相关的 Solaris 文档

下表描述了可从 docs.sun.com Web 站点上获取的相关文档。

文档集合	文档标题	描述
Solaris 参考手册集合	请参见手册页部分的标题。	提供有关 Solaris 操作系统的信息。
Solaris 软件开发者集合	《链接程序和库指南》	介绍了 Solaris 链接编辑器和运行时链接程序的操作。
Solaris 软件开发者集合	《多线程编程指南》	涵盖 POSIX 和 Solaris 线程 API、使用同步对象进行程序设计、编译多线程程序和多线程程序的查找工具。
Solaris 软件开发者集合	SPARC 汇编语言参考手册	介绍运行在 SPARC 体系结构中，并将汇编语言格式的源文件转换成链接格式目标文件的汇编程序。

开发者资源

访问 <http://developers.sun.com/prodtech/cc> 以查找以下经常更新的资源：

- 有关编程技术和最佳实例的文章
- 有关编程小技巧的知识库
- 软件的文档，以及随软件一同安装的文档的更正信息
- 有关支持级别的信息
- 用户论坛
- 可下载的代码样例
- 新技术预览

您可以通过访问 <http://developers.sun.com> 找到其他开发者资源。

联系 Sun 技术支持

如果您遇到通过本文档无法解决的技术问题，请访问以下网址：

<http://www.sun.com/service/contacting>

Sun 欢迎您提出意见

Sun 致力于提高其文档的质量，并十分乐意收到您的意见和建议。您可以通过以下网址提交您的意见和建议：

<http://www.sun.com/hwdocs/feedback>

请在电子邮件的主题行中注明文档的文件号码。例如，本文档的文件号码是 819-4755-10。

第1章

dbx 入门

dbx 是一个交叉源码级命令行调试工具。可用它以可控方式运行程序并检查已停止程序的状态。dbx 使您能够完全控制程序的动态执行过程，包括收集性能和内存使用数据、监视内存访问及检测内存泄漏。

可使用 dbx 调试由 C、C++ 或 Fortran 语言编写的应用程序。在某些限制内（请参见第 216 页的“使用 dbx 调试 Java 代码的限制”），也可以调试结合使用 Java™ 代码和 C JNI（Java™ 本地接口）代码或 C++ JNI 代码编写的应用程序。

本章将提供使用 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
- 启动 dbx 和加载程序
- 在 dbx 中运行程序
- 使用 dbx 调试程序
- 退出 dbx
- 访问 dbx 联机帮助

编译调试代码

在利用 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 95 编译器和 Java 编译器均接受此选项）编译程序。有关更多信息，请参见第 54 页的“编译调试程序”。

启动 dbx 和加载程序

要启动 dbx，在 shell 提示符处键入 dbx 命令：

```
$ dbx
```

要启动 dbx 并加载要调试的程序：

```
$ dbx program_name
```

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

```
$ dbx program_name{.class | .jar}
```

通过指定进程 ID，可使用 dbx 命令启动 dbx 并将其连接到正在运行的进程。

```
$ dbx - process_id
```

如果不知道该进程的进程 ID，请在 dbx 命令中包含 pgrep 命令以查找并连接该进程。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 命令和启动选项的详细信息，请参见第 297 页的“dbx 命令”和 dbx(1) 手册页，或键入 dbx -h。

如果 dbx 已经运行，可使用 debug 命令加载要调试的程序，或从正在调试的程序切换到其他程序：

```
(dbx) debug program_name
```

要加载或切换到包含 Java 代码和 C JNI 代码或 C++ JNI 代码的程序：

```
(dbx> debug program_name{.class | .jar}
```

如果 dbx 已经运行，也可以使用 debug 命令将 dbx 连接到正在运行的进程：

```
(dbx) debug program_name process_id
```

要将 dbx 连接到包含 Java™ 代码和 C JNI (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 debug 命令的详细信息, 请参见第 300 页的 “debug 命令”。

在 dbx 中运行程序

要在 dbx 中运行最近加载的程序, 请使用 run 命令。如果最初键入不带参数的 run 命令, 程序便会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, 请使用以下语法:

```
run [ arguments ] [ < input_file ] [ > 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。

如果重复不带参数的 run 命令, 程序将使用上一个 run 命令中的参数或重定向来重新启动。您可以使用 rerun 命令重置选项。有关 run 命令的详细信息, 请参见第 345 页的 “run 命令”。有关 rerun 命令的详细信息, 请参见第 343 页的 “rerun 命令”。

应用程序可能会运行完毕或正常终止。如果已经设置了断点, 程序可能会在断点处停止。如果应用程序包含错误, 程序会因内存故障或段故障而停止。

使用 dbx 调试程序

可能基于以下原因之一而调试程序：

- 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。当程序崩溃时，dbx 会报告发生崩溃的位置。
 - 检查核心文件并查看栈跟踪（请参见第 39 页的“检查核心文件”和第 43 页的“查看调用栈”）。
- 为了确定程序缘何得出错误结果。其方法包括：
 - 设置断点以停止运行程序，这样可以检查程序的状态和查看变量值（请参见第 40 页的“设置断点”和第 43 页的“检查变量”）。
 - 单步执行代码，每次执行一个源代码行，以监视程序状态如何变化（请参见第 42 页的“单步执行程序”）。
- 为了查找内存泄漏或内存管理问题。运行时检查允许检测运行时错误（如：内存访问错误和内存泄漏错误）并监视内存使用（请参见第 44 页的“查找内存访问问题和内存泄漏”）。

检查核心文件

为确定程序崩溃的位置，您可能要检查程序崩溃时的核心文件和程序的内存图像。可使用 `where` 命令（请参见第 377 页的“`where` 命令”）确定转储核心时程序执行的位置。

注 – dbx 无法像对待本机代码那样通过内核文件来指示 Java 应用程序的状态。

要调试核心文件，请键入：

```
$ dbx program_name core
```

或

```
$ dbx - core
```

在下例中，程序因段故障和转储核心崩溃。用户启动 `dbx` 并加载核心文件。然后使用 `where` 命令来显示栈跟踪，其中显示在 `foo.c` 文件的第 9 行出现崩溃。

```
% 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)
```

有关调试核心文件的详细信息，请参见第 48 页的“调试核心文件”。有关使用调用栈的详细信息，请参见第 43 页的“查看调用栈”。

注 - 如果程序与所有共享库动态链接，最好在创建核心文件的相同操作环境中调试核心文件。有关调试在不同的操作环境中创建的核心文件的信息，请参见第 49 页的“调试不匹配的核心文件”。

设置断点

断点是程序中要暂停程序执行的位置，它为 `dbx` 提供控制。在程序内怀疑存在错误之处设置断点。如果程序崩溃，确定崩溃发生的位置，然后在这部分代码前设置断点。

当程序在断点处停止时，便可以检查程序的状态和变量值。`dbx` 允许设置多种类型的断点（请参见第 6 章）。

最简单的断点类型就是停止断点。可以设置停止断点，使之在函数或过程中停止。例如，要在调用 `main` 函数时停止：

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


有关 `stop in` 命令的详细信息，请参见第 97 页的“在函数中设置 `stop` 断点”和第 353 页的“`stop` 命令”。

或者在特定源代码行内设置停止断点。例如，要在 `t.c` 源文件第 13 行停止：

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

有关 `stop at` 命令的详细信息，请参见第 96 页的“在源代码行设置 `stop` 断点”和第 353 页的“`stop` 命令”。

可以使用 `file` 命令设置当前文件来确定欲停止在的行；使用 `list` 命令列出欲停止在其中的函数。然后使用 `stop at` 命令在源代码行中设置断点：

```
(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
```

程序在断点处停止后，要继续执行程序，应使用 `cont` 命令（请参见第 91 页的“继续执行程序”和第 296 页的“`cont` 命令”）。

要获取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，在比较程序的实际状态和预期状态时，可能需要单步执行程序，一次执行一个源代码行。您可以使用 `step` 和 `next` 命令来执行。这两个命令都执行程序的一个源代码行，当此行执行完毕时即停止。但在处理包含函数调用的源代码行时，则有所差别：`step` 命令步入函数，而 `next` 命令步过函数。

`step up` 命令将一直执行，直至当前函数将控制权返回到调用它的函数为止。

`step to` 命令试图步入当前源代码行中的指定函数；如果未指定任何函数，则试图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 `step` 和 `next` 命令的使用及第 40 页的“设置断点”中的断点设置。

```
(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详细信息，请参见第 90 页的“单步执行程序”。有关 `step` 和 `next` 命令的详细信息，请参见第 350 页的“`step` 命令”和第 333 页的“`next` 命令”。

查看调用栈

调用栈代表那些已被调用但尚未返回各自调用程序的所有当前活动例程。在栈中，函数及其参数按调用的顺序进行存放。栈跟踪显示执行在程序流中停止的位置及执行到达此点的过程。它提供对程序状态的最简明描述。

要显示栈跟踪，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 = 0xefff93c), line 13 in "t.c"
(dbx)
```

对于已用 `-g` 选项编译的函数，参数名和参数类型是已知的，所以显示出准确的值。对于无调试信息的函数，其参数显示为十六进制数字。这些数字未必都有意义。例如，在上述栈跟踪中，帧 1 显示 SPARC 输入寄存器 `$i0 through $i5` 的内容；仅 `$i0 through $i1` 寄存器的内容有意义，因为只有两个参数被传递到第 42 页的“单步执行程序”页示例中的 `printf`。

可以停止于不使用 `-g` 选项编译的函数中。在此类函数中停止时，`dbx` 在栈内向下搜索其函数已使用 `-g` 选项编译的第一帧 - 此情况是 `printit()` - ，并将当前范围（请参见第 72 页的“程序作用域”）设置给它。这用箭头符号 (`=>`) 表示。

有关调用栈的详细信息，请参见第 7 章。

检查变量

虽然栈跟踪可能包含足够的信息以完全表示程序的状态，仍可能需要查看更多变量的值。`print` 命令可以求表达式的值，并能根据表达式的类型打印表达式的值。下面的示例显示几个简单的 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 = 0xefff93c
```

当变量和表达式的值更改时，可使用数据更改断点（请参见第 99 页的“设置数据更改断点”）进行跟踪。例如，要在变量计数值更改时停止执行，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：内存访问检查和内存使用及泄露检查。访问检查将检查被调试应用程序对内存的不正确使用。内存使用和泄露检查包括对所有显著堆空间的跟踪，然后在需要时或程序终止时扫描可用数据空间并识别未引用的空间。

注 - 只能在 Solaris 平台上进行运行时检查。

使用 `check` 命令启用内存访问检查和内存使用及泄露检查。若只打开内存访问检查，请键入：

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

要打开内存使用和内存泄漏检查，请键入：

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

打开所需的运行时检查类型后，运行程序。程序正常运行，但很缓慢，因为每次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。然后，您可以使用 `dbx` 命令，如使用 `where` 获取当前栈跟踪，或使用 `print` 命令检查变量。

注 - 由 Java 代码和 C JNI 代码或 C++ JNI 代码混编的程序不能使用运行时检查。

有关使用运行时检查的详细信息，请参见第 9 章。

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，可以连续调试任何数目的程序。

要退出 **dbx** 会话，请在 **dbx** 提示符处键入 `quit`。

```
(dbx) quit
```

如果使用 `process_id` 选项启动 **dbx** 并将其连接到进程，当退出调试会话时，进程将存在并继续运行。在退出会话之前，**dbx** 执行隐式 `detach`。

有关退出 **dbx** 的详细信息，请参见第 56 页的“退出调试”。

访问 **dbx** 联机帮助

dbx 包含帮助文件，可使用 `help` 命令进行访问：

```
(dbx) help
```


第2章

启动 dbx

本章解释如何启动、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
 - 设置启动属性
 - 调试优化代码
 - 退出调试
 - 保存和恢复调试运行
-

启动调试会话

dbx 的启动方式取决于：调试的对象、所在的位置、dbx 需要执行的任务、您对 dbx 的熟悉程度，以及是否设置了 dbx 环境变量。

要启动 dbx 会话，最简单的方法是在 shell 提示符处键入 dbx 命令。

```
$ dbx
```

要从 shell 中启动 dbx 并加载要调试的程序，请键入：

```
$ dbx program_name
```

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

```
$ dbx program_name{.class | .jar}
```

Sun Studio 软件包括两个 dbx 二进制文件：32 位 dbx 和 64 位 dbx；前者只能调试 32 位程序，后者既可以调试 32 位程序也可以调试 64 位程序。启动 dbx 时，该软件决定执行哪一个二进制文件。在 64 位操作系统上，默认值是 64 位 dbx。要在 64 位 OS 上启动 32 位 dbx，请将 dbx 命令与 `-x exec32` 选项（请参见第 298 页的“选项”）一起使用，或者设置 `_DBX_EXEC_32` 环境变量。

注 – 在 Linux OS 上，64 位 dbx 不能调试 32 位程序。要在 Linux OS 上调试 32 位程序，必须启动 32 位 dbx。

注 – 在 64 位 Linux OS 上使用 32 位 dbx 时，如果是执行 64 位程序，则不要使用 `debug` 命令或者将 `follow_fork_mode` 环境变量设置为 `child`。请退出 dbx 并启动 64 位 dbx 以调试 64 位程序。

有关 dbx 命令和启动选项的详细信息，请参见第 297 页的“dbx 命令”和 dbx(1) 手册页。

调试核心文件

如果转储核心的程序与任何共享库动态链接，则最好在创建核心文件的相同操作环境中调试核心文件。dbx 只是有限地支持“不匹配”核心文件（例如，由运行不同版本或 Solaris 操作系统修补程序级别的系统所生成的核心文件）的调试。

注 – dbx 无法像对待本机代码那样通过内核文件来指示 Java 应用程序的状态。

在相同的操作环境中调试核心文件

要调试核心文件，请键入：

```
$ dbx program_name core
```

当已经运行 dbx 时，也可以使用 `debug` 命令来调试核心文件：

```
(dbx) debug -c core program_name
```


您可以用 `-` 替代程序名，`dbx` 会尝试从核心文件中提取程序名。如果核心文件中未提供可执行程序完整路径名，`dbx` 可能找不到该文件。如果出现这种情况，只要在通知 `dbx` 加载该核心文件时指定二进制的完整路径名即可。

如果核心文件不在当前目录中可指定它的路径名（例如，`/tmp/core`）。

使用 `where` 命令（请参见第 377 页的“`where` 命令”）确定转储核心时程序执行的位置。

调试核心文件时，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。无法单步执行或设置断点。

如果核心文件被截断

如果加载核心文件时存在问题，请检查是否有被截断的核心文件。如果在核心文件创建时将其最大允许大小设得太低，那么 `dbx` 无法读取最终被截断的核心文件。在 `C shell` 中，可使用 `limit` 命令来设置允许的最大核心文件大小（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` 启动文件中更改核心文件大小的限制，重新寻找启动文件，然后重新运行生成该核心文件的程序，以生成完整的核心文件。

如果核心文件不完整并且缺少栈段，则栈跟踪信息不可用。如果缺少运行时链接程序信息，则加载对象列表不可用。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 `LWP` 列表，则没有线程信息、`lwp` 信息或栈跟踪信息。如果运行 `where` 命令，则会收到程序未“激活”的错误消息。

调试不匹配的核心文件

有时核心文件在一个系统（核心主机）上创建，而您想在另一台机器（`dbx` 主机）上加载该核心文件。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内有关运行时链接程序和线程库实现的详细信息。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补丁以及主要的 `Solaris` 操作环境升级而改变，因此如果在收集核心文件之后，但在核心文件上运行 `dbx` 之前安装补丁，此问题仍会出现在同一台主机上。

当加载“不匹配”的核心文件时，dbx 可能会显示如下的一个或多个错误消息：

```
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
```

消除共享库问题

要消除库问题并用 dbx 调试“不匹配”的核心文件，您可以执行以下操作：

1. 将 dbx 环境变量 `core_lo_pathmap` 设置为 on。
2. 使用 `pathmap` 命令告知 dbx 核心文件的正确库的位置。
3. 使用 `debug` 命令加载程序和核心文件。

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，应使用下面的命令加载 `prog` 程序和 `prog.core` 核心文件来进行调试：

```
(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` 的连接，只要确保 `libc.so.1` 可用。）

注意事项

调试不匹配的核心文件时应注意：

- `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加载对象路径名同时使用，因此请使用二元模式 `from-path to-path`。
- 如果 dbx 主机使用的 Solaris 操作环境版本与核心主机相同或更新，那么调试核心文件时效果可能会更好，虽然这并不总是必要的。

- 可能需要的系统库是:

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

/usr/lib/lwp 文件仅适用于在 Solaris 8 操作环境中运行 dbx 的情况, 并且仅在您使用交替 libthread 库时适用。

如果 dbx 在支持 64 位的 Solaris OS 版本上运行, 则需要 xxx_db.so 库的 64 位版本, 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。

ld.so.1 库是核心文件映像的一部分, 与 libc.so 或其他任何库一样, 因此需要与创建该核心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核心文件, 并且 where 命令未显示栈, 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, ...
...

```

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

使用进程 ID

您可以把进程 ID 作为 dbx 命令的一个参数，将正在运行的进程连接到 dbx。

```
$ dbx program_name process_id
```

要将 dbx 连接到正在运行的进程，该进程包含 Java™ 代码和 C JNI（Java™ 本地接口）代码或 C++ JNI 代码：

```
$ 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 - process_id
```

由于 dbx 尚不知道程序名，所以无法在 run 命令中将参数传递给进程。

有关详细信息，请参见第 88 页的“将 dbx 连接到正在运行的进程”。

dbx 启动序列

启动 dbx 时，如果未指定 -S 选项，则 dbx 在目录 */installation_directory/lib* 中查找已安装的启动文件 *dbxrc*。（在 Solaris 平台上，默认 *installation_directory* 为 */opt/SUNWsprow*；在 Linux 平台上，默认 *installation_directory* 为 */opt/sun/sun/sunstudio10u1*。）如果 Sun Studio 软件没有安装在默认目录中，dbx 会从 dbx 可执行文件路径派生出 *.dbxrc* 文件路径。

然后，dbx 依次在当前目录和 \$HOME 中搜索 *.dbxrc* 文件。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。有关详细信息，请参见第 61 页的“使用 dbx 初始化文件”。

启动文件可以包含任何 dbx 命令，通常包含 *alias*、*dbxenv*、*pathmap* 和 Korn shell 函数定义。但某些命令要求已经加载程序或已经连接进程。所有启动文件均在加载程序或进程之前加载。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 在加载程序信息的同时，将打印一系列的消息，例如：读取 *filename*。

完成程序加载后，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 95 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

默认情况下，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。如果源文件或目标文件不在此目录中或者所使用的机器没有使用相同的路径名，您必须通知 dbx 这些文件的位置。

如果移动源文件或目标文件，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目标文件路径。

向 .dbxrc 文件中添加公共 pathmap。

要建立从目录 *from* 到目录 *to* 的新映射，键入：

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

如果使用 -c，映射还将应用于当前工作目录。

pathmap 命令对于处理不同主机上具有不同基路径的自动安装或显式 NFS 安装文件系统非常有用。因为当前工作目录在自动安装的文件系统中不准确，所以当您解决由自动安装程序引起的问题时，请使用 -c。

默认情况下，存在 /tmp_mnt 到 / 的映射。

有关详细信息，请参见第 335 页的“pathmap 命令”。

设置 dbx 环境变量

可使用 `dbxenv` 命令列出或设置 dbx 自定义变量。可以将 `dbxenv` 命令放置在 `.dbxrc` 文件中。要列出变量，请键入：

```
$ dbxenv
```

您也可以设置 dbx 环境变量。有关 `.dbxrc` 文件和设置这些变量的详细信息，请参见第 3 章。

有关详细信息，请参见第 62 页的“设置 dbx 环境变量”和第 299 页的“`dbxenv` 命令”。

创建自己的 dbx 命令

您可以使用 `kalias` 或 `dalias` 命令创建自己的 dbx 命令。有关详细信息，请参见第 296 页的“`dalias` 命令”。

编译调试程序

在利用 dbx 调试程序前，必须使用 `-g` 或 `-g0` 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，要用 C++ 进行编译，请键入：

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

在 C++ 中，`-g` 选项打开调试并关闭函数的内联。`-g0`（零）选项打开调试但并不影响函数的内联。不能用 `-g0` 选项调试内联函数。`-g0` 选项可大大减少链接时间和 dbx 启动时间（取决于程序所使用的内联函数）。

要编译优化代码以便与 dbx 一起使用，应使用 `-O`（大写字母 O）和 `-g` 选项编译源代码。

调试优化代码

dbx 工具为优化代码提供部分调试支持。支持的程度很大程度取决于程序是如何进行编译的。

当分析优化代码时，可以：

- 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、显示或修改全局或静态变量
- 从一行到另一行单步执行（`next` 或 `step` 命令）

但对于优化代码，dbx 无法计算、显示或修改局部变量。

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

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通常被认为是不稳定的接口，可能随版本的变化而不同。

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”在源文件中跳转，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栈帧。

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。但对于优化程序而言，有关这些项目在程序中的位置的完整信息不可用。C++ 编译器不提供局部变量的符号类型信息；C 编译器则提供。

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` 选项编译程序，dbx 仍为未使用 `-g` 选项进行编译的代码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 `-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` 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` 工具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程序或加载对象有效：内存使用检查有效，访问检查对使用 `strip -x` 剥离的代码有效，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，可以连续调试任何数目的程序。

要退出 `dbx` 会话，请在 `dbx` 提示符处键入 `quit`。

```
(dbx) quit
```

如果使用 `process_id` 选项启动 `dbx` 并将其连接到进程，当退出调试会话时，进程将存在并继续运行。在退出会话之前，`dbx` 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。

要想不中止进程而从 dbx 中分离，键入：

```
(dbx) detach
```

要临时应用其他基于 /proc 的调试工具，这些工具可能由于 dbx 专用访问而被阻止，可以分离进程，同时将进程保留在停止状态。有关详细信息，请参见第 89 页的“从进程中分离 dbx”。

有关 detach 命令的详细信息，请参见第 303 页的“detach 命令”。

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 保持 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。

要中止 dbx 中正在执行的程序，键入：

```
(dbx) kill
```

有关详细信息，请参见第 320 页的“kill 命令”。

保存和恢复调试运行

dbx 工具提供三个保存和稍后重新运行全部或部分调试运行的命令：

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

使用 save 命令

save 命令把自上一 run、rerun 或 debug 命令直到 save 命令以来所发出的所有调试命令保存到文件中。调试会话中的此段被称为**调试运行**。

save 命令不仅仅保存已发出的调试命令列表。它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。示例 A 显示完整的保存运行。示例 B 显示减去最后两步保存的相同运行。

Example A: Saving a complete run§	Example B: Saving a run minus the last two steps§
<code>debug§</code>	<code>debug§</code>
<code>stop at line§</code>	<code>stop at line§</code>
<code>run§</code>	<code>run§</code>
<code>next§</code>	<code>next§</code>
<code>next§</code>	<code>next§</code>
<code>stop at line§</code>	<code>stop at line§</code>
<code>continue§</code>	<code>continue§</code>
<code>next§</code>	<code>next§</code>
<code>next§</code>	<code>next§</code>
<code>step§</code>	<code>step§</code>
<code>next §</code>	<code>next §</code>
<code>save§</code>	<code>save -2§</code>

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 – 默认情况下，`save` 命令将信息写入特定的保存文件。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。请参见第 59 页的“将系列调试运行另存为检查点”。

要保存直到 `save` 命令的整个调试运行，键入：

```
(dbx) save
```

要保存部分调试运行，可使用 `save number` 命令，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` 将信息写入特定的保存文件。每次保存时，`dbx` 都会覆盖此保存文件。但可通过为 `save` 命令指定 `filename` 参数，将调试运行保存到稍后可以恢复的文件，因为信息已保存到 `filename`，所以即使保存了其他调试运行，这个文件今后也能恢复。

保存一系列运行可提供**检查点集**，每个检查点都将在今后会话中启动。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其他文件而不是默认的保存文件：

```
(dbx) save filename
```

恢复已保存的运行

保存完运行之后，可使用 `restore` 命令恢复运行。`dbx` 将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` 命令也可以重新发出存储在文件中的命令集，但不能重置 `dbx` 的状态；它只能从当前程序位置重新发出命令列表。

已保存运行精确恢复的必要条件

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：运行 - 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` 将使用第二个且后保存 `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。

要恢复已保存的调试运行，键入：

```
(dbx) restore
```

要恢复已保存到其他文件而不是默认的保存文件的调试运行，键入：

```
(dbx) restore filename
```

使用 `replay` 恢复和保存

`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。默认情况下，*-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。

要重新运行减去最后发出的调试命令的当前调试运行，键入：

```
(dbx) replay
```

要重新运行当前调试运行并在特定命令前停止该运行，使用 `dbx replay` 命令，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

自定义 dbx

本章说明可用来自定义调试环境的某些属性的 dbx 环境变量，以及如何使用初始化文件 .dbxrc 保持会话之间的更改和调整。

本章由以下部分组成：

- 使用 dbx 初始化文件
- dbx 环境变量和 Korn Shell
- 设置 dbx 环境变量

使用 dbx 初始化文件

dbx 初始化文件存储每次启动 dbx 时所执行的 dbx 命令。通常，该文件包含自定义调试环境的命令，但您可以把任何 dbx 命令放置在该文件中。如果调试时是从命令行自定义 dbx，这些设置将仅适用于当前调试会话。

注 - .dbxrc 文件内不应含有执行代码的命令。但您可以将此类命令放到一个文件中，然后使用 dbx source 命令在此文件中执行这些命令。

启动期间，搜索顺序为：

1. 安装目录（除非对 dbx 命令指定 -s 选项）*/installation_directory/lib/dbxrc*（Solaris 平台和 Linux 平台上的默认 *installation_directory* 分别是 */opt/SUNWspr* 和 */opt/sun/sunstudio9*）。如果 Sun Studio 软件没有安装在默认 *installation_directory* 目录中 dbx 将从 dbx 可执行文件路径派生出 dbxrc 文件路径。
2. 当前目录 *./dbxrc*
3. 起始目录 *\$HOME/dbxrc*

创建 .dbxrc 文件

要创建包含常用自定义和别名的 .dbxrc 文件，请键入：

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

然后，可通过使用文本编辑器取消注释要执行的入口来自定义所产生的文件。

初始化文件示例

以下是示例 .dbxrc 文件：

```
dbxenv input_case_sensitive false
catch FPE
```

第一行更改区分大小写控制的默认设置：

- dbxenv 是用来设置 dbx 环境变量的命令。（有关 dbx 环境变量的完整列表，请参见第 62 页的“设置 dbx 环境变量”。）
- input_case_sensitive 是控制区分大小写的 dbx 环境变量。
- false 是为 input_case_sensitive 设置的值。

下一行是调试命令 catch，此命令将系统信号 FPE 添加到 dbx 响应的默认信号列表，以停止程序。

设置 dbx 环境变量

可使用 dbxenv 命令设置自定义 dbx 会话的 dbx 环境变量。

要显示特定变量的值，键入：

```
(dbx) dbxenv variable
```

要显示全部变量和变量值，键入：

```
(dbx) dbxenv
```

要设置变量值，键入：

```
(dbx) dbxenv variable value
```

表 3-1 显示所有可以设置的 dbx 环境变量：

表 3-1 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。 默认值：on。
CLASSPATHX	允许给 dbx 指定由自定义类加载器加载的 Java 类文件路径
core_lo_pathmap on off	控制 dbx 是否使用 pathmap 设置来定位“不匹配”核心文件的正确库。默认值：off。
disassembler_version autodetect v8 v9 v9vis	SPARC 平台：为 SPARC V8、V9 或具有可视化指令集的 V9 设置 dbx 内建反汇编程序的版本。默认值是 autodetect，可根据运行 a.out 的机器类型动态地设置模式。 IA 平台：有效选项为 autodetect。
fix_verbose on off	控制 fix 期间的编译行打印。默认值：off
follow_fork_inherit on off	当跟随子进程时，继承或不继承断点。默认值：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，当检测到派生时，将询问应跟随哪个进程。默认值：parent。
follow_fork_mode_inner unset parent child both	在检测到派生后，将 follow_fork_mode 设置为 ask，并选择了 stop 时适用。设置此变量后，无需使用 cont -follow。
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；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。 默认值：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。它可具有以下设置：java、jni 或 native。

表 3-1 dbx 环境变量 (续)

dbx 环境变量	变量功能说明
jvm_invocation	jvm_invocation 环境变量允许自定义 JVM™ 软件的启动方式。(术语“Java 虚拟机”和“JVM”表示 Java™ 平台的虚拟机。)有关详细信息,请参见第 221 页的“自定义 JVM 软件的启动”。
language_mode autodetect main c c++ fortran fortran90	控制用于分析和计算表达式的语言。 <ul style="list-style-type: none"> • 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(默认)。 • 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。
mt_scalable on off	如果启用, dbx 将限制资源的使用,并可使用 300 个以上的 LWP 调试进程。下方速度将明显减慢。默认值: off。
output_auto_flush on off	每次调用后,自动调用 fflush()。默认值: on
output_base 8 10 16 automatic	打印整型常量的默认基数。默认值: automatic (指针是十六进制字符,而其他都是十进制)。
output_class_prefix on off	在打印类成员的值和声明时,用于将类名作为类成员的前缀。如果设置为 on,便给类成员添加前缀。默认值: on。
output_dynamic_type on off	如果设置为 on,打印、显示和检查的默认值是 -d。默认值: off。
output_inherited_members on off	如果设置为 on,打印、显示和检查的默认值是 -r。默认值: off。
output_list_size num	控制 list 命令打印的默认行数。默认值: 10.
output_log_file_name filename	命令日志文件的名称。 默认值: /tmp/dbx.log.uniqueID
output_max_string_length number	为 char *s 设置打印字符串的 number。默认值: 512.
output_pretty_print on off	将 -p 设置为打印、显示和检查的默认值。默认值 off。
output_short_file_name on off	显示文件的短路径名。默认值: on。
overload_function on off	对于 C++, 如果设置为 on, 则启用自动函数重载方案。默认值: on。
overload_operator on off	对于 C++, 如果设置为 on, 则启用自动运算符重载方案。默认值: on。
pop_auto_destruct on off	如果设置为 on, 当弹出一个帧时, 自动为本地调用适当的析构函数。默认值: on。

表 3-1 dbx 环境变量 (续)

dbx 环境变量	变量功能说明
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。Warning: 请注意, 如果多个工具连接到某个进程并试图对其进行控制, 则会出现混乱。默认值: on。
rtc_auto_continue on off	将错误记录到 rtc_error_log_file_name 并继续。默认值: 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。默认值: off。
rtc_biu_at_exit on off verbose	在显式或通过 check -all 打开内存使用检查时使用。如果值为 on, 在退出程序时会生成一个非冗余内存使用(使用的块)报告。如果值为 verbose, 则会在程序退出时生成一个冗余内存使用报告。值为 off 时将不产生任何输出。默认值: on。
rtc_error_limit <i>number</i>	要报告的 RTC 错误数目。默认值: 1000。
rtc_error_log_file_name <i>filename</i>	记录 RTC 错误的文件名 (如果设置了 rtc_auto_continue)。默认值: /tmp/dbx.errlog. <i>uniqueID</i>
rtc_error_stack on off	如果设置为 on, 栈跟踪将显示与 RTC 内部机制相对应的帧。默认值: off。
rtc_inherit on off	如果设置为 on, 则启用从调试程序执行的子进程的运行时检查, 并导致 LD_PRELOAD 环境变量被继承。默认值: 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, 在退出程序时将生成一个非冗余内存泄露报告。如果值为 verbose, 则会在程序退出时生成一个冗余内存泄露报告。值为 off 时将不产生任何输出。默认值: on。
run_autostart on off	如果在没有活动程序时设置为 on, step、next、stepi 和 nexti 将隐式运行程序, 并在语言相关的主例程处停止。如果设置为 on, 需要时可通过 cont 开始 run。默认值: off。
run_io stdio 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_ptty 提供。默认值: stdio。
run_ptty <i>ptyname</i>	当 run_io 设置为 pty 时, 设置 pty 名称以供使用。Pty 供图形用户界面包装器使用。
run_quick on off	如果设置为 on, 则不会加载任何符号信息。符号信息可使用 prog -readsysms 按需加载。直到 dbx 的行为如同所调试的程序被剥离。默认值: off。

表 3-1 dbx 环境变量 (续)

dbx 环境变量	变量功能说明
<code>run_savetty</code> 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 (如果命令行中使用了 <code>-kbd</code>)。用于调试编辑器和 shell。如果 dbx 获取了 <code>SIGTTIN</code> 或 <code>SIGTTOU</code> , 并弹回到 shell, 则将其设置为 on。将其设置为 off 可稍稍加快速度。如果 dbx 连接到被调试对象, 或正在 Sun Studio IDE 中运行, 则该设置无关。默认值: on。
<code>run_setpgrp</code> on off	如果设置为 on, 当程序运行时, <code>setpgrp(2)</code> 将在派生后立即执行。默认值: off。
<code>scope_global_enums</code> 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。在处理调试信息前设置 (<code>~/ .dbxrc</code>)。默认值: off。
<code>scope_look_aside</code> on off	如果设置为 on, 则在当前范围之外查找文件静态符号。默认值: on。
<code>session_log_file_name</code> <i>filename</i>	dbx 记录所有命令及其输出的文件名。输出将被附加至文件。默认值: "" (无会话记录)。
<code>stack_find_source</code> on off	如果设置为 on, 当被调试程序在未使用 <code>-g</code> 编译的函数中停止时, dbx 将尝试查找并自动激活栈的第一帧。默认值: on。
<code>stack_max_size</code> <i>number</i>	设置 <code>where</code> 命令的大小默认值。默认值: 100。
<code>stack_verbose</code> on off	控制 <code>where</code> 中参数和行信息的打印。默认值: on。
<code>step_abflow</code> stop ignore	如果设置为 stop, 则在单步执行时 dbx 在 <code>longjmp()</code> 、 <code>siglongjmp()</code> 和 <code>throw</code> 语句中停止。如果设置为 ignore, 则 dbx 不检测 <code>longjmp()</code> 和 <code>siglongjmp()</code> 的异常控制流更改。
<code>step_events</code> on off	如果设置为 on, 在使用 <code>step</code> 和 <code>next</code> 命令单步执行代码时允许断点。默认值: off。
<code>step_granularity</code> 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: <code>a(); b();</code> 执行两个 next 命令。如果设置为 line, 将由单个 next 命令执行代码。在处理多行宏时, 行的粒度是非常有用的。默认值: statement。
<code>suppress_startup_message</code> <i>number</i>	设置版本级别, 级别以下的启动信息不打印。默认值: 3.01。
<code>symbol_info_compression</code> 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。默认值: on。
<code>trace_speed</code> <i>number</i>	设置跟踪执行的速度。其值是步骤之间暂停的秒数。默认值: 0.50。

dbx 环境变量和 Korn Shell

每个 dbx 环境变量如同 ksh 变量一样可访问。通过在 dbx 环境变量前添加 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

查看和导航到代码

每次正在调试的程序停止时，dbx 便会打印与**停止位置**关联的源代码行。dbx 会在每个程序停止位置将**当前函数**的值重置为程序停止在的函数。您可以在程序开始运行前及程序被停止时，移动或导航到程序中其他地方的函数和文件。

本章说明 dbx 如何导航到代码及如何查找函数和符号。还说明如何使用命令来导航到代码或查看标识符、类型、类的声明。

本章由以下部分组成

- 导航到代码
- 程序位置的类型
- 程序作用域
- 使用作用域转换操作符限定符号
- 查找符号
- 查看变量、成员、类型和类
- 在目标文件和可执行文件中调试信息
- 查找源文件和目标文件

导航到代码

程序被停止时，可导航到程序中其他位置的代码。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（请参见第 72 页的“程序作用域”）。这对于确定要在何时及在哪一源代码行设置 `stop at` 断点非常有用。

导航到文件

可导航到 dbx 识别为程序一部分的任何文件（即使模块或文件未使用 `-g` 选项编译）。要导航到文件：

```
(dbx) file filename
```

使用不带参数的 `file` 命令显示当前正在导航的文件的文件名。

```
(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
```

有关在源代码行的断点处设置停止的信息，请参见第 96 页的“在源代码行设置 `stop` 断点”。

导航到函数

可使用 `func` 命令导航到函数。要导航到函数，请键入命令 `func`，后跟函数名。例如：

```
(dbx) func adjust_speed
```

`func` 命令本身会回显当前函数。

有关详细信息，请参见第 313 页的“`func` 命令”。

从 C++ 二义函数名称列表中选择

如要尝试导航到使用二义名称或重载函数名的 C++ 成员函数，会显示一个列表，其中列出了有重载名称的所有函数。键入要导航的函数的号码。如果知道函数所属的具体类，可键入类名和函数名。例如：

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

在多个具体值中选择

如果可在同一作用域级访问多个符号，dbx 会打印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，从具体值列表中选择不会影响 dbx 或程序状态的值。无论选择哪个具体值，dbx 都会回显名称。

打印源码列表

使用 `list` 命令打印文件或函数的源码列表。在文件中导航后，`list` 会从顶部开始打印 *number* 行。在函数中导航后，`list` 会打印其行。

有关 `list` 命令的详细信息，请参见第 323 页的“`list` 命令”。

在调用栈中移动以导航到代码

活动进程存在时，另一种导航到代码的方法是“在调用栈中移动”，使用栈命令来查看调用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。在栈中移动会使当前函数和文件在每次显示栈函数时均发生变化。停止位置被认为位于栈的“底部”，因此要从其中移出，请使用 `up` 命令，即向 `main` 或 `begin` 函数方向移动。使用 `down` 命令向当前帧方向移动。

有关在调用栈中移动的更多信息，请参见第 110 页的“栈中移动和返回起始位置”。

程序位置的类型

dbx 使用三个全局位置来跟踪正在检查的程序部分：

- 当前地址，由 `dis` 命令（请参见第 303 页的“`dis` 命令”）和 `examine` 命令（请参见第 307 页的“`examine` 命令”）使用和更新。
- 当前源代码行，由 `list` 命令（请参见第 323 页的“`list` 命令”）使用和更新。此行号由一些改变访问作用域的命令重置（请参见第 73 页的“更改访问作用域”）。
- 当前访问作用域，在第 73 页的“访问作用域”中有对该复合变量的说明。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、`list func` 命令及 `list file` 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上可见的，则称为“在作用域内”。在 C 语言中，函数可以有全局或文件静态作用域；变量可以有全局、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，且不受更改访问作用域的各种命令的影响：

<code>\$scope</code>	当前程序计数器的作用域
<code>\$lineno</code>	当前行号
<code>\$func</code>	当前函数
<code>\$class</code>	<code>\$func</code> 所属类
<code>\$file</code>	当前源文件
<code>\$loadobj</code>	当前装入对象

访问作用域

使用 `dbx` 检查程序的各种元素时，便要修改访问作用域。`dbx` 会在对表达式求值期间使用访问作用域来解决诸如二义符号等问题。例如，如果键入以下命令，`dbx` 会使用访问作用域来确定打印哪个 `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，每个线程都会记住其自己的访问作用域。

访问作用域的组件

访问作用域的其中一些组件在以下预定义 `ksh` 变量中可见：

<code>\$vscope</code>	语言作用域
<code>\$vloadobj</code>	当前访问装入对象
<code>\$vfile</code>	当前访问源文件
<code>\$vfunc</code>	当前访问函数
<code>\$vlineno</code>	当前访问行号
<code>\$vclass</code>	C++ 类

当前访问作用域的所有组件相互间保持兼容。例如，如果访问不包含函数的文件，当前访问源文件会被更新为新文件名，当前访问函数会被更新为 `NULL`。

更改访问作用域

以下命令是更改访问作用域的最常见方法：

- `func`
- 文件
- `up`
- `down`
- `frame`
- `list procedure`

`debug` 命令和 `attach` 命令设置初始访问作用域。

遇到断点时，`dbx` 会将访问作用域设置到当前位置。如果 `stack_find_source` 环境变量（请参见第 62 页的“设置 `dbx` 环境变量”）被设置为 `ON`，`dbx` 会尝试查找并激活有源代码的栈帧。

使用 `up` 命令（请参见第 373 页的“`up` 命令”）、`down` 命令（第 305 页的“`down` 命令”）、`frame number` 命令（请参见第 312 页的“`frame` 命令”）或 `pop` 命令（请参见第 336 页的“`pop` 命令”）更改当前栈帧时，`dbx` 会根据新栈帧的程序计数器设置访问作用域。

只有在使用 `list function` 或 `list file` 命令时，`list`（请参见第 323 页的“`list` 命令”）命令使用的行号位置才会更改访问作用域。设置访问作用域后，`list` 命令的行号位置会被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便会被更新，但只要是在当前文件中列出各行，访问作用域就不会改变。例如，如果键入以下内容，`dbx` 便会列出 `my_func` 源的开头，并将访问作用域更改为 `my_func`。

```
(dbx) list my_func
```

如果键入以下内容，`dbx` 会在当前源文件中列出第 127 行，而不会更改访问作用域。

```
(dbx) list 127
```

使用 `file` 命令或 `func` 命令来更改当前文件或当前函数时，访问作用域也会被相应更新。

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给定目标函数的名称。

`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 (`::`)、块局部操作符 (`:lineno`)。单独使用它们，在某些情况下一起使用。

除在代码中导航时限定文件和函数名外，符号名限定也是打印和显示作用域外变量和表达式及显示类型和类声明所必需的（使用 `whatis` 命令）。在所有情况下，符号限定规则都是相同的；本节论述所有符号名限定类型的规则。

反引号操作符

使用反引号字符 (```) 来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，以便其记录将导航的函数。要按文件名限定函数名，请使用通用反引号 (`) 作用域转换操作符。

```
(dbx) func `file_name`function_name
```

C++ 双冒号作用域转换操作符

使用双冒号操作符 (::) 来限定 C++ 成员函数、顶级函数或有以下名称、具有全局作用域的变量：

- 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）

可能需要限定重载函数名。如果不限定它，dbx 会显示一个重载列表，以便从中选择要导航的函数。如果知道函数类名，可使用它与双冒号作用域转换操作符配合来限定名称。

```
(dbx) func class::function_name (args)
```

例如，如果 hand 是类名，而 draw 是函数名，请键入：

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

块局部操作符

利用块局部操作符 (:line_number) 可专门引用嵌套块中的变量。如果有跟踪参数或成员名的局部变量，或如果有几个块，每一块都有其自己的局部变量版本，则可能需要这样做。line_number 是感兴趣变量在块内第一行代码的号码。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，但您可以使用 dbx 表达式作用域内的任意行号。

在下例中，块局部操作符 (:230) 与反引号操作符合并使用。

```
(dbx) stop in `animate.o`change_glyph:230`item
```

下例显示了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.c`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 提供了一种特殊语法来按链接程序名（在 C++ 中为损坏名称）查找符号。使用 #（磅符号）字符作为符号名的前缀（在任何 \$（美元符号）字符前使用 ksh 换码符 \（反斜线）），如以下这些示例中所示：

```
(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程序实体的不同类型，并可能会在许多作用域中出现。dbx whereis 命令会列出完全限定的名称，即该名称全部符号的位置。如果在表达式中给出该名称，dbx which 命令会告知 dbx 会使用符号的哪个具体值（请参见第 379 页的“which 命令”）。

打印符号具体值列表

要打印指定符号所有具体值的列表，请使用 whereis 符号，其中 *symbol* 可以是任何用户定义标识符。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
```

输出包括程序定义 *symbol* 所在的可装入对象的名称以及每个对象的实体类型：类、函数或变量。

由于来自 dbx 符号表的信息需要时才会被读入，因此 whereis 命令只会记录已装入的符号的具体值。随着调试会话变长，具体值列表也会增长（请参见第 83 页的“在目标文件和可执行文件中调试信息”）。

有关详细信息，请参见第 379 页的“whereis 命令”。

确定 dbx 使用哪个符号

`which` 命令会告知 `dbx` 会使用哪个有给定名称的符号，前提是在表达式中指定了该名称（没有完全限定）。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，`which` 命令便会按作用域转换搜索路径搜索符号的第一个具体值。如果 `which` 找到该名称，会报告完全限定的名称。

如果搜索在搜索路径的任何位置找到了同一作用域级的 *symbol* 的多个具体值，`dbx` 便会在命令窗格中打印一条消息，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` 显示重载显示，列出二义符号名。在 `which` 命令的上下文中，从具体值列表中选择不会影响 `dbx` 或程序状态的值。无论选择哪个具体值，`dbx` 都会回显名称。

如果将 *symbol*（本例中为 `block`）作为必须在 *symbol* 上运行的某个命令（例如，`print` 命令）的一个参数，`which` 命令会提供预览将发生情况的功能。如果有二义名，重载显示列表会指示 `dbx` 尚未记录它使用的是两个或更多名称中的哪个具体值。`dbx` 会列出各种可能的值，等待您从中选择一个。有关 `which` 命令的详细信息，请参见第 379 页的“`which` 命令”。

作用域转换搜索路径

发出包含表达式的调试命令时，会按以下顺序查找表达式中的符号。`dbx` 会像编译器在当前访问作用域的同样方式解析符号。

1. 在使用当前访问作用域的当前函数的作用域内（请参见第 73 页的“访问作用域”）。如果程序停止在嵌套块中，`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，一般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. 如果以上搜索都不成功，`dbx` 会假定正在引用另一文件中的专用或文件静态的变量或函数。`dbx` 还可选择根据 `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，都会先沿此搜索路径查找。如果 `dbx` 无法找到符号，会报告错误。

放松作用域查找规则

要为静态符号和 C++ 成员函数放松作用域查找规则，请将 `dbx` 环境变量 `scope_look_aside` 设置为 `on`：

```
dbxenv scope_look_aside on
```

或使用“双反引号”前缀：

在 `func4` 处停止 `func4` 可以是静态的，也可以不在作用域中

如果 `dbx` 环境变量 `scope_look_aside` 被设置为 `on`，`dbx` 会查找：

- 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）。`/usr/lib` 的库中的文件不会被搜索。
- 没有类限定的 C++ 成员函数
- 其他文件中的 C++ 内联成员函数实例化（如果当前文件中的某个成员函数未实例化）。

`which` 命令会告知 `dbx` 会选择哪个符号。如果有二义名，重载显示列表会指示 `dbx` 尚未确定它使用的是两个或更多名称中的哪个具体值。`dbx` 会列出各种可能的值，等待您从中选择一个。

有关详细信息，请参见第 313 页的“`func` 命令”。

查看变量、成员、类型和类

`whatis` 命令会打印标识符、结构、类型和 C++ 类的声明或定义，或表达式类型。可查找的标识符包括变量、函数、字段、数组和枚举常量。

有关详细信息，请参见第 373 页的“`whatis` 命令”。

查找变量、成员和函数的定义

要打印输出标识符的声明，请键入：

```
(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** 标识符会列出函数模板实例化。模板定义显示时有 **whatis -t identifier**。请参见第 81 页的“查找类型和类的定义”。

对于 Java 程序，**whatisidentifier** 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

要打印输出成员函数，请键入：

```
(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();
```

要打印输出数据成员，请键入：

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

在变量上，**whatis** 指示变量的类型。

```
(dbx) whatis the_table  
class table *the_table;
```

在字段上，**whatis** 提供字段类型。

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


停止在一个成员函数中时，可查找 `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化。

要打印某类型或 C++ 类的声明，请键入：

```
(dbx) whatis -t type_or_class_name
```

要查看继承成员，`whatis` 命令带有一个 `-r` 选项（代表递归），它显示指定类的声明和从基类中继承的成员。

```
(dbx) whatis -t -r class_name
```

`whatis -r` 查询中的输出可能会很长，是否会很长取决于类分层结构和类的大小。输出以从最原始的类继承的成员列表开头。插入的注释行将成员列表分为其各自的父类。

以下是使用类 `table` 的两个示例，父类 `load_bearing_block` 的子类同时又是 `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);
};
```

以下是在子类上使用 `whatis -r` 来查看它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);
};
```

在目标文件和可执行文件中调试信息

通常想使用 `-g` 选项来编译源文件，以使程序的可调试性更佳。`-g` 选项会使编译器将调试信息（以 `stabs` 或 `Dwarf` 格式）连同程序的代码和数据记录到目标文件中。

`dbx` 会根据需要为每个目标文件（模块）分析和装入调试信息。可以使用 `module` 命令让 `dbx` 为任何特定模块或所有模块装入调试信息。另请参见第 85 页的“查找源文件和目标文件”。

目标文件装入

目标（`.o`）文件被链接到一起后，链接程序可选择只将汇总信息存储到生成的装入对象中。`dbx` 可以在运行时使用此汇总信息直接从目标文件（而不是从可执行文件）装入其余调试信息。生成的可执行文件的磁盘覆盖区较小，但 `dbx` 运行时需要有目标文件。

使用 `-xs` 选项编译目标文件便可忽略此要求，使那些目标文件的所有调试信息在链接时都被置入可执行文件中。

如果使用目标文件创建归档库（`.a` 文件），并在程序中使用归档库，则 `dbx` 会根据需要从归档库中提取目标文件。此时不需要原始目标文件。

将所有调试信息置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`（调试信息的默认格式）时的默认行为是使编译器只将汇总信息置入可执行文件中。

DWARF 格式尚不支持目标文件装入。

注 – 记录相同的信息，使用 DWARF 格式要比使用 `stabs` 格式占用的空间小。但由于全部信息都被复制到可执行文件中，DWARF 信息看上去比 `stabs` 信息的体积要大。

列出模块的调试信息

`module` 命令及其选项有助于在调试会话过程中记录程序模块。使用 `module` 命令读入一个或多个模块的调试信息。正常情况下，`dbx` 会根据需要自动、“懒洋洋地”读入模块的调试信息。

要读入模块 *name* 的调试信息，请键入：

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

要读入所有模块的调试信息，请键入：

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

其中：

- a 指定所有模块。
- f 强制读取调试信息，即使文件比可执行文件新。
- q 指定安静模式。
- v 指定冗余模式，在该模式下会打印语言、文件名等信息。这是默认设置。

要打印当前模块的名称，请键入：

```
(dbx) module
```

列出模块

`modules` 命令列出模块名称来协助记录模块。

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的名称（无论其是否包含调试信息），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，在该模式下会打印语言、文件名等信息。

查找源文件和目标文件

`dbx` 必须知道与程序关联的源文件和目标代码文件的位置。目标文件的默认目录是程序上次链接时文件所在的目录。源文件的默认目录是上次编译时它们所在的目录。如果移动源文件或目标文件，或将它们复制到另一位置，必须要么重新链接程序，在调试前更改到新位置；要么使用 `pathmap` 命令

`dbx` 有时会使用目标文件来装入附加调试信息。`dbx` 显示源代码时，会使用源文件。

如果编译和链接程序后移动了源文件或目标文件，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目标文件路径。

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`，映射还将应用于当前工作目录。

`pathmap` 命令对处理在不同主机上具有不同基路径的自动安装或显式 NFS 安装文件系统同样有用。因为当前工作目录在自动安装的文件系统中不准确，所以当您解决由自动安装程序引起的问题时，请使用 `-c`。

默认情况下，存在 `/tmp_mnt` 到 `/` 的映射。

有关详细信息，请参见第 335 页的“`pathmap` 命令”。

第5章

控制程序执行

用于运行、单步执行和继续进行的命令（`run`、`rerun`、`next`、`step` 和 `cont`）称为**进程控制命令**。与附录 B 中描述的事件管理命令一起使用，当程序在 `dbx` 下执行时，可以控制程序的运行时行为。

本章由以下部分组成：

- 运行程序
- 将 `dbx` 连接到正在运行的进程
- 从进程中分离 `dbx`
- 单步执行程序
- 使用 `Ctrl+C` 停止进程

运行程序

首次将程序载入 `dbx` 时，`dbx` 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；您可以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™ 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，请参见第 220 页的“在 JVM 软件尚未装入的代码上设置断点”。

使用 `run` 命令开始执行程序。

要在 `dbx` 中不带参数运行程序，请键入：

```
(dbx) run
```

可以选择添加命令行参数和输入、输出重定向。

```
(dbx) run [arguments] [ < input_file] [ > output_file]
```

注 – 无法重定向 Java 应用程序的输入和输出。

`run` 命令的输出将覆盖现有文件，即使已为正在运行 `dbx` 的 shell 设置了 `noclobber`。

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。有关详细信息，请参见第 345 页的“`run` 命令”。`rerun` 命令重新启动程序并清除原始参数和重定向。有关详细信息，请参见第 343 页的“`rerun` 命令”。

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。如果属于下列情况，便需要连接正在运行的进程：

- 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，要调试但又不想中止程序。

可以通过把程序的 `process_id` 编号作为 `dbx debug` 命令的一个参数，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。

如果在将其连接到正在运行的进程后退出 `dbx`，`dbx` 会在终止前隐式分离。

要将 `dbx` 连接到独立于 `dbx` 而运行的某个程序，可选择使用 `attach` 命令或 `debug` 命令。

要将 `dbx` 连接到已经运行的进程，请键入：

```
(dbx) debug program_name process_id
```

或

```
(dbx) attach process_id
```

可以用 `-`（短线）替代 `program_name`；`dbx` 将自动查找与进程 ID 相关联的程序并予以加载。

有关详细信息，请参见第 300 页的“debug 命令”和第 282 页的“attach 命令”。

如果 dbx 尚未运行，可通过键入下列命令来启动 dbx：

```
% dbx program_name process_id
```

dbx 连接到程序后，程序便停止执行。当其他程序载入 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。

如果在调试现有进程时将 dbx 连接到一个新进程，会出现以下情况：

- 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 命令或通过命令行中指定进程 ID 来开始调试当前进程，那么 dbx 会在连接新进程之前从当前进程中分离。

可以对具有某些异常的连接的进程使用运行时检查。请参见第 147 页的“对连接的进程使用运行时检查”。

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。

要分离 dbx 控制下正在运行的进程：

```
(dbx) detach
```

要临时应用其他基于 /proc 的调试工具，这些工具可能由于 dbx 专用访问而被阻止，可以分离进程，同时将进程保留在停止状态。例如：

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

有关详细信息，请参见第 303 页的“detach 命令”。

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` 的两个变体，称为 `step up` 和 `step to`。`next` 和 `step` 这两个命令均使程序于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` 命令在步入函数之后将程序返回调用程序函数。

`step to` 命令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。可能会因为条件转移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。在这种情况下，`step to` 命令将步过当前源代码行。

单步执行

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

```
(dbx) next n
```

或

```
(dbx) step n
```

有关上述命令的详细信息，请参见第 333 页的“`next` 命令”和第 350 页的“`step` 命令”。

`step_granularity` 环境变量确定 `step` 命令和 `next` 命令用于单步执行代码的单元（请参见第 62 页的“设置 dbx 环境变量”）。其单元可以是语句或行。

`step_events` 环境变量控制步骤中是否启用断点（请参见第 62 页的“设置 dbx 环境变量”）。

`step_abflow` 环境变量控制 dbx 在检测到即将发生异常的控制流更改时是否停止（请参见第 62 页的“设置 dbx 环境变量”）。对 `siglongjmp()` 或 `longjmp()` 的调用或异常抛出可以导致控制流更改。

继续执行程序

要继续一个程序，请使用 `cont` 命令。

```
(dbx) cont
```

`cont` 命令有一个变体 `cont at line_number`，它允许指定恢复程序执行的当前程序位置之外的行。这样可以跳过已知引起问题的一行或多行代码，而无需重新编译。

要于指定行继续程序，请键入：

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

行号是相对于程序停止的文件求出的；给定的行号必须位于当前函数的作用域内。

使用 `cont at line_number` 和 `assign`，可以避免执行包含调用某个函数（该函数可能会不正确地计算某一变量的值）的代码行。

要于指定行恢复程序执行，请键入：

1. 使用 `assign` 给变量赋正确的值。
2. 使用 `cont at line_number` 来跳过包含导致错误地计算值的函数调用的行。

假定程序停止于第 123 行。第 123 行调用函数 `how_fast()`，该函数不正确地计算变量 `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 行继续程序执行。

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

有关详细信息，请参见第 296 页的“`cont` 命令”。

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 `when` 命令的详细信息，请参见：

- 第 96 页的“在源代码行设置 `stop` 断点”
- 第 98 页的“在不同类的成员函数中设置断点”
- 第 98 页的“在相同类的成员函数中设置断点”
- 第 99 页的“在非成员函数中设置多个断点”
- 第 375 页的“`when` 命令”

调用函数

程序停止时，可以使用 `dbx call` 命令来调用函数，此命令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 `function_name` 后键入括号。例如：

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

可以使用 `call` 命令来显式调用函数；或者通过求包含函数调用的表达式的值或使用 `stop in glyph -if animate()` 之类的条件修饰符来隐式调用函数。

C++ 虚函数与其他函数一样，可以使用 `print` 命令、`call` 命令（请参见第 337 页的“`print` 命令”或第 284 页的“`call` 命令”）或其他执行函数调用的命令进行调用。

如果定义函数的源文件使用 `-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。

默认情况下，在每个 `call` 命令之后，`dbx` 都会自动调用 `fflush(stdout)` 来确保 I/O 缓冲区内存存储的所有信息全部被打印。要关闭自动刷新，请将 `dbx` 环境变量 `output_auto_flush` 设置为 `off`。

对 C++ 而言，`dbx` 可处理隐式 `this` 指针、默认参数和函数重载。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` 编译的函数），`dbx` 将显示重载名称列表。

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。如果现在键入 `where` 命令，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 `pop -c` 命令弹出直到最近调用的所有帧。

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，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。

第6章

设置断点和跟踪

发生事件时，可利用 `dbx` 停止进程、执行任意命令或打印信息。一个最简单的事件示例就是断点。其他事件的示例包括错误、信号、系统调用、对 `dlopen()` 的调用以及数据更改。

跟踪显示程序中事件的有关信息，如某变量值的更改。尽管跟踪的行为与断点的行为不同，但跟踪和断点共享类似的事件处理程序（请参见第 259 页的“事件处理程序”）。

本章说明如何设置、清除和列出断点和跟踪。有关在设置断点和跟踪时可以使用的事件规范的详尽信息，请参见第 261 页的“设置事件规范”。

本章由以下部分组成：

- 设置断点
- 在断点上设置过滤器
- 跟踪执行
- 在行中设置 `when` 断点
- 在共享库中设置断点
- 列出和清除断点
- 启用和禁用断点
- 效率方面的考虑

设置断点

在 `dbx` 中，可以使用三个命令来设置断点：

- `stop` 断点 – 如果程序执行到使用 `stop` 命令创建的断点处时便会停止。您发出另一调试命令（如 `cont`、`step` 或 `next`）后，程序才会继续执行。
- `when` 断点 – 如果程序执行到使用 `when` 命令创建的断点处时便会停止，`dbx` 会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` 断点 – 程序执行到使用 `trace` 命令创建的断点处时便会停止，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。在第 261 页的“设置事件规范”中有对事件规范的详细论述。

要设置计算机级断点，请使用 `stopi`、`wheni` 和 `tracei` 命令（请参见第 18 章）。

注 - 调试使用 Java™ 代码和 C JNI（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，请参见第 220 页的“在 JVM 软件尚未装入的代码上设置断点”。

在源代码行设置 `stop` 断点

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

```
(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` 会在下一可执行行设置断点。如果没有可执行行，`dbx` 会发出错误。

可以使用 `file` 命令设置当前文件来确定欲停止在的行；使用 `list` 命令列出欲停止在其中的函数。然后使用 `stop at` 命令在源代码行中设置断点：

```
(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
```

有关在位置事件指定的详细信息，请参见第 262 页的“`at [filename:] line_number`”。

在函数中设置 stop 断点

可以使用 `stop in` 命令在函数中设置断点:

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

In Function 断点会在过程或函数第一个源代码行的开头处暂停程序执行。

`dbx` 应能确定所引用的变量或函数,但在下列情况下除外:

- 只通过名称来引用一个重载的函数。
- 引用 ``` 开头的函数或变量。

假设有下面一组声明:

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

在非成员函数处停止时,可以键入:

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

在全局 `foo(int)` 处设置断点。

要在成员函数上设置断点,可以使用以下命令:

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

如果键入:

```
stop in foo
```

`dbx` 便无法确定所指为全局函数 `foo(int)` 还是全局函数 `foo(double)`,而被迫显示一个重载菜单,要求澄清。

如果键入:

```
stop in `bar
```

dbx 便无法确定所指的是全局函数 `bar()` 还是成员函数 `bar()`，而显示一个重载菜单。
有关指定 `in function` 事件的详细信息，请参见第 261 页的“`in function`”。

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` 命令连用，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（相同成员函数名，不同类）的每个对象特定变量中设置断点，请使用 `stop inmember`。

例如，如果函数 `draw` 定义在几个不同的类中，要在每个函数中置入断点，请键入：

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

有关指定 `inmember` 或 `inmethod` 事件的详细信息，请参见第 262 页的“`inmember function inmethod function`”。

在相同类的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。

默认情况下，断点只插入到类中定义的类型成员函数中，而不会插入到可能从基类继承的类成员函数中。要同样在从基类继承的函数中插入断点，请指定 `-recurse` 选项

要在类 `shape` 中定义的所有成员函数中设置断点，请键入：

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

要在类 `shape` 中定义的所有成员函数中以及从类继承的函数中都设置断点，请键入：

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

有关指定 `inclass` 事件的详细信息，请参见第 262 页的“`inclass classname [-recurse | -norecurse]`”和第 353 页的“`stop` 命令”。

由于 `stop inclass` 和其他断点选择可能会插入大量断点，应该确保将 `dbx` 环境变量 `step_events` 设置为 `on` 以加速 `step` 和 `next` 命令（请参见第 107 页的“效率方面的考虑”）。

在非成员函数中设置多个断点

要在带重载名称（相同名、不同类型或参数数量不同）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++ 程序已定义了以 `sort()` 命名的函数的两种版本（一种传递 `int` 类型参数，另一种传递 `float` 类型参数），要在这两个函数中都设置断点，请键入：

```
(dbx) stop infunction sort [command;]
```

有关指定 `infunction` 事件的详细信息，请参见第 262 页的“`infunction function`”。

在对象中设置断点

设置 `In Object` 断点来检查应用于特定对象实例的操作。

默认情况下，`In Object` 断点会在对象类（包括从对象调用时的继承类）的所有非静态成员函数中暂停程序执行。要只在对象类而不在继承类中定义的非静态成员函数中设置断点来暂停程序执行，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和在对象 `foo` 的继承类中定义的所有非静态成员函数中设置断点，请键入：

```
(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，请键入：

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

有关指定 `inobject` 事件的详细信息，请参见第 262 页的“`inobject object-expression [-recurse | -norecurse]`”和第 353 页的“`stop` 命令”

设置数据更改断点

变量或表达式的值更改时，可以在 `dbx` 中使用数据更改断点来进行记录。

地址被访问时停止执行

要在内存地址被访问时停止执行，请键入：

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

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 也可以包含以下两个字母之一：

- a 访问后停止进程（默认值）。
- 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”或“之后”指副作用。

address-expression 是可对其求值来生成地址的任何表达式。如果给出符号表达式，则可以自动推导出要监视的区域大小，可以指定 *byte-size-expression* 来将其覆盖。也可以使用非符号、无类型地址表达式。使用该表达式时，必须提供大小。

在下例中，执行将在内存地址 0x4762 被读取后停止：

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

本例中，执行将在变量速度被写入前停止：

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

使用 `stop access` 命令时请记住以下这些要点：

- 变量被写入时（即使值不变），事件便会发生。
- 默认情况下，事件发生在写入变量的指令执行后。可以将模式指定为 **b** 来指示想让事件发生在指令执行前。

有关指定访问事件的详细信息，请参见第 263 页的“`access mode address-expression [, byte-size-expression]`”和第 353 页的“`stop` 命令”。

变量更改时停止执行

要在指定变量的值更改时停止程序执行，请键入：

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

使用 `stop change` 命令时请记住以下这些要点：

- `dbx` 会将程序停止在引起指定变量值更改的行之后的一行。
- 如果 *variable* 是函数的局部变量，则第一次输入函数并分配了 *variable* 的存储空间时，便将该变量视为已更改。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。

有关指定更改事件的详细信息，请参见第 264 页的“change variable”和第 353 页的“stop 命令”。

`dbx` 通过使自动单步执行与每一步的值检查一并来进行来执行 `stop change`。如果库未使用 `-g` 选项编译，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，通知您它无法执行此操作。

注 - 使用 `access` 事件比使用 `change` 事件能更快地发现数据更改。`access` 事件不自动单步执行程序，而是使用速度快得多的页保护方案。

条件停止执行

要在条件语句的求值为真时停止程序执行，请键入：

```
(dbx) stop cond condition
```

condition 发生时，程序便会停止执行。

使用 `stop cond` 命令时请记住以下这些要点：

- dbx 会将程序停止在使条件的求值为真的行之后的一行。
- 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详细信息，请参见第 264 页的“cond condition-expression”和第 353 页的“stop 命令”。

在断点上设置过滤器

在 dbx 中，大多数事件管理命令同样支持可选事件过滤器修饰符。最简单的过滤器在程序执行到断点或跟踪处理程序或监视条件发生后指示 dbx 为条件进行测试。

如果此过滤器条件的求值为真（非 0），会应用事件命令，程序会在断点处停止执行。如果条件的求值为假 (0)，dbx 会继续执行程序，好像事件从未发生过。

要在行处或函数中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` 命令的尾部。

条件可以是任意有效的表达式，包括函数调用、输入命令时以当前语言返回的布尔值或整数值。

对于像 `in` 或 `at` 这样基于位置的断点，作用域便是断点位置的作用域。否则，条件的作用域便是输入时的作用域，而不是事件发生时的作用域。可能必须使用反引号操作符（请参见第 74 页的“反引号操作符”）精确指定作用域。

例如，以下这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：测试条件。后者自动单步执行：测试条件。

将函数调用的返回值用作过滤器

可以将函数调用用作断点过滤器。本例中，如果字符串 `str` 中的值是 `abcde`，则执行会在函数 `foo()` 中停止：

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

将变量作用域用作过滤器

变量作用域可以用于设置断点过滤器。本例中，当前作用域在 `function foo()` 中，而 `local` 是在 `main()` 中定义的局部变量：

```
(dbx) stop access w &main\local -if pr(main\local) -in main
```

将过滤器与条件事件配合使用

新用户有时会将设置条件事件命令（监视类型命令）和使用过滤器混淆。从概念上来说，“监视”会创建在每行代码执行前必须要检查的前提（在监视作用域内）。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` 被写入（“监视”部分），`-if` 过滤器便生效。`dbx` 便会检查 `speed` 的新值是否与 `fast_enough` 相等。如果不相等，程序会继续执行，而“忽略”`stop`。

在 `dbx` 语法中，过滤器在命令尾部以 `[-if condition]` 形式来表示。

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

如果在多线程程序中使用包含函数调用的过滤器设置断点，`dbx` 便会在遇到断点时停止执行所有线程，然后对条件求值。如果满足条件且调用了函数，`dbx` 便会继续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
```

线程 `t@1` 调用 `lookup()` 时，`dbx` 便会停止，并对条件求值，然后调用 `strcmp()` 便会继续执行所有线程。如果在函数调用期间 `dbx` 在另一线程中遇到断点，它会发出如下之一的警告：

```
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，如果可确定在条件表达式中调用的函数不会去争夺互斥，则可以使用 `-resumeone` 事件规范修饰符来强制 `dbx` 只继续执行它遇到断点时所在的第一个线程。例如，可能设置以下断点：

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

`-resumeone` 修饰符无法在所有情况下都可防止问题发生。例如，它在下列情况下便无能为力：

- `lookup()` 中的第二个断点与第一个断点都在同一线程中发生，因为条件会以递归方式调用 `lookup()`。
- 条件运行于的线程将控制权放弃，交给另一线程。

有关事件修饰符的详细信息，请参见第 271 页的“事件规范修饰符”。

跟踪执行

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

跟踪会显示每行即将被执行的源代码。除最简单的程序以外，此跟踪都会产生大量输出。

更有用的跟踪会应用过滤器来显示程序中事件的相关信息。例如，可以跟踪对函数的每个调用、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

在命令行键入 `trace` 命令来设置跟踪。`trace` 命令的基本语法为：

```
trace event-specification [ modifier ]
```


有关跟踪命令的完整语法，请参见第 365 页的“trace 命令”。

跟踪提供的信息取决于与它关联的事件的类型（请参见第 261 页的“设置事件规范”）。

控制跟踪速度

跟踪输出往往瞬间闪过。利用 dbx 环境变量 `trace_speed` 能够控制每个跟踪打印后的延迟。默认延迟为 0.5 秒。

要设置跟踪期间每行代码执行间的间隔（秒），请键入：

```
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
```

要将跟踪输出还原为标准输出，请使用 `-` 代表 `filename`。跟踪输出始终附加到 `filename` 后。并在显示 dbx 提示和退出应用程序时刷新。`filename` 在新运行或连接后继续运行时总会被重新打开。

在行中设置 when 断点

`when` 断点命令接受其他 dbx 命令，如 `list`，从而可以利用它编写自己的 `trace` 版本。

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

`when` 命令与隐含的 `cont` 命令一并执行。上例中，在当前行列出源代码后，程序会继续执行。如果在 `list` 命令后加入了 `stop` 命令，程序便不会继续执行。

有关 `when` 命令的完整语法，请参见第 375 页的“when 命令”。有关事件修饰符的详细信息，请参见第 271 页的“事件规范修饰符”。

在共享库中设置断点

dbx 对使用连接运行时链接程序的编程接口的代码提供全面的调试支持：调用 `dlopen()`、`dlclose()` 及其关联函数的代码。运行时链接程序在程序执行期间会绑定和解开共享库。利用对 `dlopen()` 和 `dlclose()` 的调试支持可步入函数，或在动态共享库的函数中以在程序启动时在链接的库中设置断点的同样方式来设置断点。

但有几个例外。dbx 无法在尚未装入的装入对象（例如，使用 `dlopen()` 装入的对象）中放置断点：

- 无法在 `dlopen()` 装入某库前在 `dlopen()` 装入的库中设置断点。
- `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。

可以使用 `loadobject` 命令将此类装入对象的名称置于预装列表中（请参见第 325 页的“`loadobject` 命令”）。

dbx 会考虑使用 `dlopen()` 装入的装入对象。例如，在刚装入的装入对象中设置的断点会一直保持到下一次运行命令执行时，即便使用 `dlclose()` 卸下了装入对象，也可随后使用 `dlopen()` 再次装入。

列出和清除断点

在调试会话期间往往会设置不止一个断点或跟踪处理程序。dbx 支持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` 命令在括号中显示 ID 号，其他命令随后可使用该号。

dbx 将使用 `inmember`、`inclass` 和 `infunction` 关键字设置的多个断点以使用一个状态 ID 号的一组断点来报告。

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 会显示创建时分配给每个断点的 ID 号。可以使用 `delete` 命令按 ID 号删除断点，也可使用关键字 `all` 来删除当前在程序中设置的所有断点。

要按 ID 号删除断点（此例中为 3 和 5），请键入：

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

要删除 dbx 中当前装入的程序中设置的所有断点，请键入：

```
(dbx) delete all
```

有关详细信息，请参见第 302 页的“delete 命令”。

启用和禁用断点

用于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（请参见第 259 页的“事件处理程序”）。这些命令中的每一个均会返回一个称为处理程序 ID (*hid*) 的数。可以将处理程序 ID 用作 `handler` 命令的一个参数（请参见第 315 页的“handler 命令”）来启用或禁用断点。

效率方面的考虑

各种事件在被调试程序的执行时间方面有不同程度的开销。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只有最小量的开销。

像 `inclass` 这样的多个断点可能会产生成百上千的断点，它在创建期间会有开销。这是因为 dbx 使用永久性断点，这些断点永久保留在进程中，每次中断时不会被移除，每次执行 `cont` 命令时都会被置入。

注 – 就 `step` 和 `next` 而言，默认情况下，进程继续执行前所有断点都会被移除，步骤完成后又会被重新插入。如果使用大量断点或在多个类中使用多个断点，`step` 和 `next` 的执行速度会显著降低。使用 dbx `step_events` 环境变量来控制是否移除断点并在每次执行 `step` 或 `next` 后重新插入。

速度最慢的事件是利用自动单步执行功能的事件。这可能与在 `trace step` 命令中一样明显，它单步执行每个源代码行。像 `stop change expression` 或 `trace cond variable` 等其他事件不但自动单步执行，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，但往往可以使用 `-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
```

不要使用 `trace -in main`，因为 `trace` 在 `main` 调用的函数中也有效。怀疑 `lookup()` 函数正在破坏变量时，一定要使用它。

第7章

使用调用栈

本章论述 dbx 如何使用调用栈，及处理调用栈时如何使用 where、hide、unhide 和 pop 命令。

调用栈代表那些已被调用但尚未返回各自调用程序的所有当前活动例程。栈帧是分配供一个函数使用的调用栈的一段。

由于调用栈是从高端内存（大地址）延伸到低端内存，因此向上意味着向调用函数的帧移动（最终移动到 main()），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、执行一步后或出错并产生内核文件时，用于例程执行的帧位于低端内存中。调用程序例程，如 main()，位于高端内存中。

本章由以下部分组成：

- 确定在栈中的位置
- 栈中移动和返回起始位置
- 在栈中上下移动
- 弹出调用栈
- 隐藏栈帧
- 显示和读取栈跟踪

确定在栈中的位置

使用 where 命令确定当前在栈中的位置。

```
where [-f] [-h] [-l] [-q] [-v] number_id
```

调试使用 Java™ 代码和 C JNI（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 命令在了解崩溃并生成内核文件的程序的状态时也很有用。出现这种情况时，可将内核文件装入到 dbx 中（请参见第 48 页的“调试核心文件”）

有关 where 命令的详细信息，请参见第 377 页的“where 命令”。

栈中移动和返回起始位置

在栈中上下移动称为“栈中移动”。通过在栈中上下移动来访问函数时，dbx 会显示当前函数和源代码行。起始位置 *home* 是程序停止执行的点。可以从起始使用 up、down 或 frame 命令在栈中上下移动。

dbx 命令 up 和 down 均接受 *number* 参数，该参数指示 dbx 在栈中从当前帧向上或向下移动若干帧。如果未指定 *number*，默认值为 1。-h 选项将所有隐藏帧都加入到计数中。

在栈中上下移动

可以检查非当前函数中的局部变量。

栈中上移

要在调用栈中上移（向 main 移动）*number* 级：

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

如果未指定 *number*，默认值为一级。有关详细信息，请参见第 373 页的“up 命令”。

栈中下移

要在调用栈中下移（向当前停止点移动）*number* 级：

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

如果未指定 *number*，默认值为一。有关详细信息，请参见第 305 页的“down 命令”。

移到特定帧

`frame` 命令与 `up` 和 `down` 命令类似。利用它可直接转到 `where` 命令显示的给定编号的帧。

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

不使用参数的 `frame` 命令显示当前帧编号。该命令可利用 *number* 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的级别数。-h 选项将所有隐藏帧都包括在计数中。

也可以使用 `pop` 命令移至特定帧（请参见第 111 页的“弹出调用栈”）。

弹出调用栈

可以从调用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。

与在调用栈中上下移动不同，弹出栈会更改程序的执行。从栈中删除停止于函数后，该函数会使程序恢复先前状态，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` 命令从调用栈中删除一个或多个帧。例如，要从栈中弹出五个帧，请键入：

```
pop 5
```

也可以弹到特定帧。要弹到第 5 帧，请键入：

```
pop -f 5
```

有关详细信息，请参见第 336 页的“pop 命令”。

隐藏栈帧

使用 `hide` 命令列出当前正在使用的栈帧过滤器。

要隐藏或删除与所有的正则表达式匹配的所有栈帧，请键入：

```
hide [ regular_expression ]
```

regular_expression 与函数名或装入对象的名称匹配，并使用 `sh` 或 `ksh` 语法进行文件匹配。

使用 `unhide` 来删除所有栈帧过滤器。

```
unhide 0
```

由于 `hide` 命令会列出带有号码的过滤器，因此还可以使用带有过滤器号码的 `unhide` 命令。

```
unhide [ number | regular_expression ]
```

显示和读取栈跟踪

栈跟踪显示执行在程序流中停止的位置及执行到达此点的过程。它提供对程序状态的最简明描述。

要显示栈跟踪，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，其参数显示为十六进制数字。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。

可以停止于不使用 `-g` 选项编译的函数中。在此类函数中停止时，`dbx` 会在栈中向下搜索其函数使用 `-g` 选项编译的第一个帧，并将当前作用域（请参见第 72 页的“程序作用域”）设置为此值。这用箭头符号 (`=>`) 表示。

在下例中，`main()` 使用 `-g` 选项编译，因此会显示符号名以及参数值。`main()` 调用的库函数不是使用 `-g` 来编译的，因此会显示函数的符号名，但参数便只会显示 SPARC 输入寄存器 `$i0` 至 `$i5` 的十六进制内容：

```
(dbx) where
  [1] _libc_poll(0xffbef3b0, 0x1, 0xffffffff, 0x0, 0x10,
0xffbef604), at 0xfef9437c
  [2] _select(0xffbef3b8, 0xffbef580, 0xffbef500, 0xffbef584,
0xffbef504, 0x4), at 0xfef4e3dc
  [3] _XtWaitForSomething(0x5a418, 0x0, 0x0, 0xf4240, 0x0, 0x1),
at 0xff0bdb6c
  [4] XtAppNextEvent(0x5a418, 0x2, 0x2, 0x0, 0xffbef708, 0x1), at
0xff0bd5ec
  [5] XtAppMainLoop(0x5a418, 0x0, 0x1, 0x5532d, 0x3, 0x1), at
0xff0bd424
=>[6] main(argc = 1, argv = 0xffbef83c), line 48 in "main.cc"
```

本例中，程序因段故障而崩溃。同时只有 `main()` 是使用 `-g` 选项进行编译，所以库函数的参数以十六进制显示，不带符号名称。可能性最大的崩溃原因是 SPARC 输入寄存器 `$i0` 和 `$i1` 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
当前函数是 main
(dbx) where
  [1] strlen(0x0, 0x0, 0x11795, 0x7efefeff, 0x81010100,
0xff339323), 位于 0xff2b6c5c
  [2] _doprnt(0x11799, 0x0, 0x0, 0x0, 0x0, 0xff00), 位于 0xff2fec18
  [3] printf(0x11784, 0xff336264, 0xff336274, 0xff339b94,
0xff331f98, 0xff00), 位于 0xff300780
=>[4] main(argc = 1, argv = 0xffbef894), line 133 in "Cdlib.c"
(dbx)
```

要查看栈跟踪的更多示例，请参见第 43 页的“查看调用栈”和第 204 页的“跟踪调用”。

第8章

求值和显示数据

在 dbx 中，可以执行两种类型的数据检查：

- 求数据的值 (`print`) – 抽样检查表达式的值
- 显示数据 (`display`) – 每次程序停止时监视表达式的值

本章由以下部分组成：

- 求变量和表达式的值
 - 给变量赋值
 - 求数组的值
-

求变量和表达式的值

本节讨论如何使用 dbx 求变量和表达式的值。

验证 dbx 使用的变量

如果不能确定 dbx 正在求哪个变量的值，使用 `which` 命令可查看 dbx 正在使用的完全限定的名称。

要查看定义变量名的其他函数和文件，请使用 `whereis` 命令。

有关上述命令的详细信息，请参见第 379 页的“`which` 命令”和第 379 页的“`whereis` 命令”。

当前函数作用域之外的变量

需要求值或监视当前函数作用域之外的变量时：

- 限定函数名。请参见第 74 页的“使用作用域转换操作符限定符号”。
- 或
- 通过更改当前函数来访问该函数。请参见第 69 页的“导航到代码”。

打印变量、表达式或标识符的值

除 dbx 中引入的用来处理作用域和数组的元语法之外，表达式应遵循当前语言的语法。

要使用本机代码求变量或表达式的值，请键入：

```
print expression
```

可以使用 `print` 命令在 Java 代码中求表达式、局部变量或参数的值。

有关详细信息，请参见第 337 页的“`print` 命令”。

注 - dbx 支持 C++ `dynamic_cast` 和 `typeid` 运算符。当使用这两种运算符求表达式的值时，dbx 会调用由编译器提供的一些 `rtti` 函数。如果源码没有显式使用这些运算符，编译器便不会生成这些函数，而且 dbx 将无法求表达式的值。

打印 C++

在 C++ 中，一个对象指针具有两种类型：**静态类型**（在源代码中定义的类型）和**动态类型**（对象在进行任何类型转换之前的状态）。dbx 有时会提供有关对象动态类型的信息。

一般而言，如果对象含有虚函数表（即 `vtable`），dbx 便可使用 `vtable` 中的信息来正确指明对象的类型。

可以使用带有 `-r`（递归）选项的 `print` 或 `display` 命令。dbx 会显示所有由类直接定义的数据成员和从基类继承的数据成员。

这些命令也具有 `-d` 或 `+d` 选项，用来切换 dbx 环境变量 `output_derived_type` 的默认行为。

如果在没有运行进程时使用 `-d` 标志或将 dbx 环境变量 `output_dynamic_type` 设置为 `on`，则会生成“程序未被激活”的错误消息，因为没有进程便无法访问动态信息。如果尝试通过虚拟继承来查找动态类型，将生成“类指针非法类型转换”的错误消息。（在 C++ 中，从虚拟基类转换到派生类是非法的。）

求 C++ 程序中未命名参数的值

C++ 允许使用未命名的参数来定义函数。例如：

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

虽然未命名参数不能在程序的其他地方使用，但是编译器会按某种格式对未命名参数进行编码，使您可以对其求值。格式如下，其中编译器会将一个整数赋值给 %n：

```
 _ARG%n
```

要获取编译器所赋给的名称，请键入以函数名作为其目标的 `whatis` 命令。

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

有关详细信息，请参见第 373 页的“`whatis` 命令”。

要求值（或显示）未命名的函数参数，请键入：

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

非关联化指针

在非关联化某一指针时，请查看该指针所指向容器的内容。

为了非关联化指针，`dbx` 会在命令窗格中显示求出的值；在这种情况下，指的是 `t` 所指向的值：

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

监视表达式

每次程序停止时监视表达式的值是了解特定表达式或变量如何和何时更改的有效方法。`display` 命令指示 `dbx` 监视一个或多个指定表达式或变量。监视会一直持续下去，直至使用 `undisplay` 命令将其关闭为止。

要在每次程序停止时显示变量或表达式的值，请键入：

```
display expression, ...
```

一次可以监视多个变量。不带选项的 `display` 命令打印所显示的所有表达式的列表。

有关详细信息，请参见第 304 页的“`display` 命令”。

关闭显示（取消显示）

`dbx` 会一直显示所监视变量的值，直至使用 `undisplay` 命令关闭显示为止。可以关闭指定表达式的显示，也可以关闭当前被监视的所有表达式的显示。

要关闭特定变量或表达式的显示，请键入：

```
undisplay expression
```

要关闭所有当前被监视的变量的显示，请键入：

```
undisplay 0
```

有关详细信息，请参见第 370 页的“`undisplay` 命令”。

给变量赋值

要给变量赋值，请键入：

```
assign variable = expression
```

求数组的值

数组与其他类型的变量求值方法相同。

以下是 Fortran 数组示例：

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

要求数组的值，请使用 `print` 命令。例如：

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

`dbx print` 命令允许求大型数组的某部分的值。数组求值包括：

- 数组分片 – 打印多维数组的任意矩形阵列、 n 维阵列。
- 数组跨距 – 仅打印指定数组片内固定阵列中的某些元素（可以是整个数组）。

进行数组分片时，可以使用跨距，也可以不使用跨距。（默认跨距值为 1，即表示打印每个元素。）

数组分片

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

C 和 C++ 的数组分片语法

对各维数组而言，使用 `print` 命令进行数组分片的完整语法为：

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

其中：

<i>array-expression</i>	求得的值应为数组或指针类型的表达式。
<i>first-expression</i>	要打印的第一个元素。默认值为 0。
<i>last-expression</i>	要打印的最后一个元素。默认值为数组上界。
<i>stride-expression</i>	跨距长度（所跳过的元素个数 $stride-expression - 1$ ）。默认值为 1。

第一个表达式、最后一个表达式和跨距表达式是可选表达式，它们求得的值应为整数。

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

对各维数组而言，使用 `print` 命令进行数组分片的完整语法为：

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

其中：

<i>array-expression</i>	求得的值应为数组类型的表达式。
<i>first-expression</i>	某个范围内的第一个元素，也是要打印的第一个元素。默认值为数组下界。
<i>last-expression</i>	某个范围内的最后一个元素，但如果跨距不等于 1，则可能不是要打印的最后一个元素。默认值为数组上界。
<i>stride-expression</i>	跨距长度。默认值为 1。

第一个表达式、最后一个表达式和跨距表达式是可选表达式，它们求得的值应为整数。对于 n 维数组片，请用逗号分隔各片的定义。

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, 请键入:

```
demo% f77 -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
```

要打印第 3 行，请键入：

```
(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)
```

要打印第 4 列，请键入：

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

数组片

以下是 C++ 数组的一个二维矩形片示例（省略默认跨距为 1）。

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

此命令打印大型数组的一部分元素。请注意，该命令省略了 *stride-expression*，而使用默认跨距值 1。

	100	101	102	103	104	105	106
200	□	□	□	□	□	□	□
201	□	▣	▣	▣	▣	▣	□
202	□	▣	▣	▣	▣	▣	□
203	□	▣	▣	▣	▣	▣	□
204	□	□	□	□	□	□	□
205	□	□	□	□	□	□	□

图 8-1 跨距为 1 的二维矩形数组片示例

如图 8-1 所示，前两个表达式 (201:203) 指定这个二维数组中第一维的数组片（三行一列）。数组片从第 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）用于定义第二维的数组片。数组片从第 101 列开始，到第 105 列结束。

跨距

当指示 `print` 跨越数组片时，`dbx` 将只求该片中某些元素的值，而在求值元素之间跳过固定数目的元素。

在数组分片语法中，第三个表达式 *stride-expression* 指定跨距长度。 *stride-expression* 值指定要打印的元素。默认跨距值为 1，意即：求指定数组片中的所有元素值。

以下数组与上一数组片示例使用的数组相同。这一次打印命令在第二维包含数组片跨距 2。

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

如图 8-2 所示，使用跨距 2 将跳过所有其他元素打印所有的第二个元素。

	100	101	102	103	104	105	106
200							
201		▣		▣		▣	
202		▣		▣		▣	
203		▣		▣		▣	
204							
205							

图 8-2 跨距为 2 的二维矩形数组片示例

对于省略的所有表达式，打印时取与数组声明的大小相等的默认值。以下是如何使用简化语法的示例。

对于一维数组，请使用以下命令：

```
print arr                使用默认边界打印整个数组。  
print arr(:)            使用默认边界和默认跨距 1 打印整个数组。  
print arr(::stride-expression) 使用跨距 stride-expression 打印整个数组。
```

对于二维数组，可使用以下命令打印整个数组。

```
print arr
```

要打印二维数组中第二维的所有第三个元素，请键入：

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


第9章

使用运行时检查

利用运行时检查 (RTC) 可在开发阶段于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露）。还可利用它监控内存使用情况。无法对 Java 代码使用运行时检查。

本章阐述以下主题：

- 运行时检查功能
- 使用运行时检查
- 使用访问检查
- 使用内存泄漏检查
- 使用内存使用检查
- 禁止错误
- 对子进程使用运行时检查
- 对连接的进程使用运行时检查
- 同时使用修复并继续与运行时检查
- 运行时检查应用编程接口
- 在批处理模式下使用运行时检查
- 疑难解答提示
- 运行时检查的 8 M 字节限制
- 运行时检查错误

注 - 运行时检查功能仅在运行 Solaris OS 的平台上可用。该功能在运行 Linux OS 的平台上不可用。

注 - 内存访问检查功能仅在运行 Solaris OS 的 SPARC 平台上可用。该功能在运行 Solaris OS 的 x86 平台上不可用。

运行时检查功能

由于运行时检查是一种整型调试功能，因此可在使用运行时检查时（使用“收集器”收集性能数据的情况除外）执行所有调试操作。

运行时检查：

- 检测内存访问错误
- 检测内存泄露
- 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` 标志编译在运行时检查错误消息中提供了源代码行号关联。运行时检查还可以检查用优化 `-O` 标志编译的程序。对不使用 `-g` 选项编译的程序有某些特别注意事项。

可以通过 `check` 命令来使用运行时检查功能。

使用运行时检查的时机

避免同时出现大量错误的一种方法是在开发周期的组成模块开发阶段便提早使用运行时检查功能。编写一个单元测试来驱动每个模块，然后使用运行时检查，以递增方式每次检查一个模块。这样每次需要处理的错误数量便会减少。将所有模块集成为一个完整的程序时，便可能遇不到什么新错误。将错误数减少到零后，就只有在对模块进行更改时，才需要再次运行运行时检查。

运行时检查要求

使用运行时检查必须满足以下要求：

- 程序使用 Sun 编译器编译。
- 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 (API) 来处理其他分配器。请参见第 149 页的“运行时检查应用编程接口”。
- 可接受未完全剥离的程序和使用 `strip -x` 剥离的程序。

局限性

对不基于 UltraSPARC® 处理器的硬件，运行时检查无法处理大于 8 M 字节的程序文本区和数据区。有关详细信息，请参见第 152 页的“运行时检查的 8 M 字节限制”。

一种可行的解决办法是在可执行映像中插入一些特殊文件来处理大于 8 M 字节的程序文本区和数据区。

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。

打开内存使用和内存泄漏检查

要打开内存使用和内存泄漏检查，请键入：

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

打开内存使用检查或内存泄漏检查时，`showblock` 命令会显示给定地址堆块的详情。详情包括块分配的位置及其大小。有关详细信息，请参见第 347 页的“`showblock` 命令”。

打开内存访问检查

若只打开内存访问检查，请键入：

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

注 – 内存访问检查功能在运行 Solaris OS 的 x86 平台上不可用。

打开所有运行时检查

要打开内存泄漏、内存使用和内存访问检查，请键入：

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

有关详细信息，请参见第 286 页的“check 命令”。

关闭运行时检查

要完全关闭运行时检查，请键入：

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

有关详细信息，请参见第 369 页的“uncheck 命令”。

运行程序

打开所需运行时检查类型后，使用或不使用断点运行被测程序。

程序正常运行，但很缓慢，因为每次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。控制会向您返回（除非 `dbx` 环境变量 `rtc_auto_continue` 设置为 `on`（请参见第 62 页的“设置 `dbx` 环境变量”。）

然后便可发出 `dbx` 命令，如发出 `where` 命令来获取当前栈跟踪，或发出 `print` 命令来检查变量。如果错误并非致命错误，则可以使用 `cont` 命令继续执行程序。程序继续执行到下一错误或断点（先检测到者优先）。有关详细信息，请参见第 296 页的“`cont` 命令”。

如果将 `rtc_auto_continue` 设置为 `on`，运行时检查会继续查找错误，并自动保持运行。它会将错误重定向到以 `dbx` 环境变量 `rtc_error_log_file_name` 命名的文件。（请参见第 62 页的“设置 `dbx` 环境变量”。）默认日志文件名为 `/tmp/dbx.errlog.uniqueid`。

可以使用 `suppress` 命令限制报告运行时检查错误。有关详细信息，请参见第 358 页的“`suppress` 命令”。

以下是一个说明如何为名为 `hello.c` 的程序打开内存访问和内存使用检查的简单示例。

```
% 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 librttc.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。运行时检查会将此访问错误报告为从未初始化项读取 (rui)。

函数 `memory_leak()` 返回前不释放变量 `local`。`memory_leak()` 返回时，此变量超出作用域，分配于第 20 行的块变为泄漏。

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，这种情况会被报告为使用的块 (biu)。

使用访问检查

访问检查功能通过监控每个读取、写入、分配及释放操作来检查程序访问内存的方式是否正确。

注 – 内存访问检查功能在运行 Solaris OS 的 x86 平台上不可用。

程序可能会以各种不正确的方式读取或写入内存，这些都被称为内存访问错误。例如，程序可能会通过 `free()` 调用堆块来引用已释放的内存块。函数也可能会将指针返回给局部变量，该指针被访问时，便会产生错误。访问错误可能会导致指针在程序中乱跳，并可能导致程序行为异常，包括输出错误和段故障。某些类型的内存访问错误很难跟踪。

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所涉及的内存块的状态进行比对，然后确定该操作是否有效。可能的内存状态有：

- 未分配，初始状态 – 内存尚未分配。读取、写入或释放此内存是非法操作，因为它不归程序所有。
- 已分配，但未初始化 – 内存已分配给程序，但未初始化。写入或释放此内存为合法操作，但读取为非法操作，因为内存尚未初始化。例如，输入函数时，局部变量的栈内存已分配，但未初始化。
- 只读 – 读取为合法操作，但不能写入或释放只读内存。
- 已分配和初始化 – 读取、写入或释放已分配和初始化的内存为合法操作。

使用运行时检查查找内存访问错误与使用编译器在程序中查找语法错误没有什么不同。在这两种情况下，都会产生错误列表，并有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，链中的第一个错误是“首要原因”，修复该错误后，其引发的一些错误可能便会被修复。

例如，读取内存的未初始化区会创建不正确的指针。非关联化该指针时，便会导致另一无效读取或写入发生，而这一错误又会导致另一错误发生。

理解内存访问错误报告

运行时检查会打印内存访问错误的以下信息：

错误	信息
类型	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。
addr	尝试访问的地址。
详细信息	有关 addr 的更多详细信息。例如，如果 addr 在邻近栈，便会提供其相对于当前栈指针的位置。如果 addr 在堆中，便提供最近堆块的地址、大小和相对位置。
stack	出错时调用栈（批处理模式）。
分配	如果 addr 在堆中，则提供最近堆块的分配跟踪。
位置	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 0xefff50
    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

运行时检查检测到以下内存访问错误：

- rui（请参见第 156 页的“从未初始化的内存中读 (rui) 错误”）
- rua（请参见第 155 页的“从未分配的内存中读 (rua) 错误”）
- wua（请参见第 156 页的“写入到未分配内存 (wua) 错误”）
- wro（请参见第 156 页的“写入到只读内存 (wro) 错误”）
- mar（请参见第 154 页的“未对齐读 (mar) 错误”）
- maw（请参见第 155 页的“未对齐写 (maw) 错误”）
- duf（请参见第 154 页的“重复释放 (duf) 错误”）
- baf（请参见第 154 页的“错误释放 (baf) 错误”）
- maf（请参见第 154 页的“未对齐释放 (maf) 错误”）
- oom（请参见第 155 页的“内存不足 (oom) 错误”）

注 - 运行时检查不执行数组边界检查，因此不会将数组边界超出按访问错误报告。

使用内存泄漏检查

内存泄漏是一个动态分配的内存块，在程序的数据空间中没有指向它的指针。这样的块是孤立内存。由于没有指针指向这些块，程序无法引用它们，更谈不上释放它们。运行时检查会查找和报告这样的块。

内存泄漏会导致虚拟内存消耗增加，且通常会产生内存碎片。这可能会降低程序及整个系统的执行速度。

导致内存泄漏的通常原因是未释放分配的内存，而又丢失了指向分配块的指针。以下是一些内存泄漏的示例：

```
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.                        */
}
```

不正确使用 API 可导致泄漏。

```
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.*/
}
```

不再需要时便释放内存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尚未释放的内存块。内存泄漏的这一定义使用的几率要小得多，因为常见的编程惯例是，如果程序不久便会终止，就不释放内存。如果程序仍然保留一个或多个指针指向内存块，运行时检查便不会将内存块按泄漏来报告。

检测内存泄漏错误

运行时检查检测到以下内存泄漏错误：

- mel（请参见第 157 页的“内存泄漏 (mel) 错误”）
- air（请参见第 157 页的“地址位于寄存器内 (air) 错误”）
- aib（请参见第 157 页的“地址位于块内 (aib) 错误”）

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

可能的泄漏

在两种情况下，运行时检查会报告“可能的”泄漏。第一种情况是没有指针指向块开始处，但有指针指向块内部时。程序将这种情况按“地址位于块内 (aib)”错误来报告。如果指针是指向块内部的迷失指针，便是真正的内存泄漏。不过，某些程序会根据需要有意将唯一指针反复指向数组来访问其条目。这种情况便不是内存泄漏。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，让您来确定哪一种情况是真正的内存泄漏。

在数据空间中找不到指向内存块的指针，但可在寄存器中找到指针时，便会发生第二种类型的可能泄漏。程序将这种情况按“地址位于寄存器内 (air)”错误来报告。如果寄存器意外地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，编译器可以不必将指针写入内存就能优化引用和将指向内存块的唯一指针置入寄存器中。这种情况便不是真正的泄漏。因此，如果程序已得到优化，而报告是 showleaks 命令的执行结果，便可能不是真正的泄漏。所有其他情况便可能是真正的泄漏。有关详细信息，请参见第 348 页的“showleaks 命令”。

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，请参见第 149 页的“运行时检查应用编程接口”。

检查泄漏

如果打开了内存泄漏检查，则被测程序退出前，会自动执行内存泄漏扫描。程序会报告所有检测到的泄漏。不应使用 `kill` 命令中止程序。以下是一个典型的内存泄漏错误消息：

```
Memory leak (mle):
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 程序有一个 `main` 过程（在 `f77` 中称为 `MAIN`），它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被按泄漏来报告（除非全局变量也指向这些块）。

常见的编程惯例是不释放分配给 `main` 中的局部变量的堆块，因为程序即将在不调用 `exit()` 的情况下终止，并从 `main` 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` 的最后一个可执行源代码行上设置一个断点，以在 `main` 返回前停止程序。程序停止在该处后，使用 `showleaks` 命令报告所有真正的泄漏，忽略仅由 `main` 中的变量超出作用域而致的泄漏。

有关更多信息，请参见第 348 页的“`showleaks` 命令”。

理解内存泄漏报告

打开泄漏检查时，如果程序退出，便会收到自动产生的泄漏报告。程序会报告所有可能的泄漏 - 使用 `kill` 命令中止程序。报告的详细程度由 `dbx` 环境变量 `rtc_mle_at_exit`（请参见第 62 页的“设置 `dbx` 环境变量”）控制。默认情况下生成简短的泄漏报告。

报告按泄漏的合并大小排序。先报告实际内存泄漏，然后报告可能的泄漏。详细报告包含详细的栈跟踪信息，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以下信息：

信息	描述
位置	泄漏块被分配到的位置。
addr	泄漏块的地址。
size	泄漏块的大小。
stack	分配时调用栈，如 <code>check -frames</code> 所限。

以下是相应的简短内存泄漏报告。

```
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
```

以下是一个典型的详细泄漏报告。

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

Memory Leak (m1):
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 (m1):
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

可以使用 `showleaks` 命令随时查看泄漏报告，该报告会报告上次执行 `showleaks` 命令以来的新内存泄漏。有关详细信息，请参见第 348 页的“`showleaks` 命令”。

合并泄漏

由于单个泄漏的数量可能会很大，运行时检查会自动将分配在同一位置的泄漏合并成一个合并泄漏报告。合并或单独报告泄漏的决定由 `showleaks` 命令的 `check -leaks` 或 `-m` 选项上的 `-match m` 选项所指定的 `number-of-frames-to-match` 参数控制。如果两个或更多泄漏分配时的调用栈与 `m` 帧在严格程序计数器等级匹配，这些泄漏便会在一个合并泄漏报告中报告。

假设有以下三个调用序列：

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

如果所有这些块均导致内存泄漏，则 `m` 的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 2 按一个重复泄漏来报告，因为两个调用序列 `malloc()` 上的 2 个栈帧相同。块 3 将按单独泄漏来报告，因为 `c()` 的跟踪与其他块不匹配。`m` 大于 2，运行时检查会将所有泄漏按单独泄漏来报告。（`malloc` 不显示在泄漏报告中。）

一般来将，`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

得到内存泄漏报告后，请按以下指导修复内存泄漏。

- 最重要的是确定泄漏位置。泄漏报告会告知泄漏块的分配跟踪，即泄漏块被分配到的位置。
- 然后可以查看程序的执行流程，了解块的使用情况。如果显然指针已经丢失，便很好解决；否则，可以使用 `showleaks` 来限定泄漏时段。默认情况下，`showleaks` 命令提供上次执行 `showleaks` 命令以来产生的新泄漏。可以在程序中单步执行的同时重复运行 `showleaks` 来缩小内存块泄漏的时段。

有关详细信息，请参见第 348 页的“`showleaks` 命令”。

使用内存使用检查

利用内存使用检查可了解所有使用中的堆内存。可以使用此信息对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有一个大致了解。此信息对减少程序的动态内存消耗也有用处，并可能有助于性能优化

内存使用检查在性能优化过程中或对虚拟内存使用的控制有用处。程序退出时，便可生成内存使用报告。也可在程序执行过程中随时使用 `showmemuse` 命令来获得内存使用信息，使用该命令可以显示内存使用情况。有关信息，请参见第 349 页的“`showmemuse` 命令”。

打开内存使用检查便同时打开了泄漏检查。除程序退出时的泄漏报告外，还会获得使用的块 (`biu`) 报告。默认情况下，程序退出时会生成简短的使用的块报告。内存使用报告的详细程度由 `dbx` 环境变量 `rtc_biu_at_exit`（请参见第 62 页的“设置 `dbx` 环境变量”）。

以下是一个典型的简短内存使用报告。

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

Total Size	% of All	Num of Blocks	Avg Size	Allocation call stack
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

以下是相应的详细内存使用报告：

```
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"
```

可以随时使用 `showmemuse` 命令查看内存使用报告。

禁止错误

运行时检查提供了一个强大的错误禁止工具，利用它可以非常灵活地限制所报告错误的数量和类型。如果发生被禁止的错误，则不会提供报告，程序会像未出现错误一样继续执行。

可以使用 `suppress` 命令禁止错误（请参见第 358 页的“`suppress` 命令”）。

可以使用 `unsuppress` 命令撤消错误禁止（请参见第 372 页的“`unsuppress` 命令”）。

禁止在同一调试会话期间内的各 `run` 命令中持续存在，但在各 `debug` 命令中不持续存在。

禁止的类型

可使用以下禁止类型：

按作用域和类型禁止

必须指定要禁止的错误类型。可以指定要禁止的程序部分。选项有：

选项	描述
全局	默认值，应用于整个程序。
装入对象	应用于整个装入对象（如共享库）或主程序。
文件	应用于特定文件中的所有函数。
功能	应用于特定函数。
Line	应用于特定源代码行。
地址	应用于某地址处的特定指令。

禁止上一错误

默认情况下，运行时检查禁止最近的错误，以防重复产生同一错误的报告。这种行为由 `dbx` 环境变量 `rtc_auto_suppress` 控制。`rtc_auto_suppress` 设置为 `on`（默认值）时，在特定位置发生的特定访问错误只在首次出现时报告，此后会被禁止报告。例如，因多次执行的循环中出现错误而要防止产生同一错误报告的多个副本时，便要用到这一设置。

限制报告的错误数

可以使用 `dbx` 环境变量 `rtc_error_limit` 限制将报告的错误数。访问错误和泄漏错误的错误限制分别使用。例如，如果将错误限制设置为 5，则在运行结束时的泄漏报告和发出每个 `showleaks` 命令时，都会最多显示五个访问错误和五个内存泄漏。默认值为 1000。

禁止错误示例

下例中，`main.cc` 是文件名，`foo` 和 `bar` 是函数，`a.out` 是可执行文件名。

不报告分配在函数 `foo` 中发生的内存泄漏。

```
suppress mel in foo
```

禁止报告从 `libc.so.1` 分配的使用的块。

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

禁止在 `a.out` 的所有函数中从未初始化项读取。

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

不报告文件 `main.cc` 中的从未分配项读取。

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

在 `main.cc` 的第 10 行禁止重复释放。

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

禁止报告函数 `bar` 中的所有错误。

```
suppress all in bar
```

有关详细信息，请参见第 358 页的“`suppress` 命令”。

默认禁止

要检测到所有错误，运行时检查不要求程序使用 `-g` 选项（符号）编译。不过，为保证准确检测某些错误（大部分是 `rui` 错误），有时需要符号信息。为此，如果未提供符号信息，某些错误（`a.out` 的 `rui`、共享库的 `rui`、`aib` 和 `air`）在默认情况下会被禁止。这种行为可以使用 `suppress` 和 `unsuppress` 命令的 `-d` 选项来更改。

执行以下命令会使运行时检查不再于无符号信息（不使用 `-g` 编译）的代码内禁止从未初始化内存读取（`rui`）：

```
unsuppress -d rui
```

有关详细信息，请参见第 372 页的“`unsuppress` 命令”。

使用禁止来管理错误

在大程序上初次运行时，可能无法应付出现的大量错误。采取分阶段的方法可能会更好。使用 `suppress` 命令将报告的错误减少到可以管理的数量，只修复这些错误，然后重复该周期；随着重复次数的增加，需要禁止的错误会越来越少，这样便实现了对错误的管理。

例如，可以每次侧重处理几个类型的错误。最常见的错误类型有 `rui`、`rua` 和 `wua`，通常是这一顺序。`rui` 错误较不严重（尽管它们会导致后来发生的更严重错误）。尽管存在这些错误，程序可能往往仍然能够正确运行。`rua` 和 `wua` 错误较严重，因为它们会访问无效内存地址或从无效内存地址访问，并始终指示编码错误。

可以先禁止 `rui` 和 `rua` 错误。修复所有出现的 `wua` 错误后，再次运行程序，这一次只禁止 `rui` 错误。修复所有出现的 `rua` 错误后，再次运行程序，这一次不禁止错误。修复所有 `rui` 错误。最后，最后一次运行程序，确保无残余错误。

如果想禁止报告的上一错误，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 `dbx` 环境变量 `rtc_inherit` 设置为 `on`。默认情况下，它被设置为 `off`。（请参见第 62 页的“设置 `dbx` 环境变量”。）

如果为父进程启用了运行时检查且 `dbx` 环境变量 `follow_fork_mode` 被设置为 `child`（请参见第 62 页的“设置 `dbx` 环境变量”），则 `dbx` 支持对子进程进行运行时检查。

发生派生时，`dbx` 会对子进程自动执行运行时检查。如果程序调用 `exec()`，则调用 `exec()` 的程序运行时检查设置会被传递给程序。

在任意时刻，运行时检查只能控制一个进程。下面是一个示例。

```
% 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 librtdc.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 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 $oprogram"
Reading symbolic information for program2
Skipping ld.so.1, already read
Skipping librttc.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
=====
      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执行运行时检查。但是进程启动时必须预装 `rtcaudit.so`。如果要连接的进程是 64 位进程，请使用相应的 64 位 `rtcaudit.so`。如果该产品安装在 `/opt` 目录中，`rtcaudit.so` 便位于：

`/opt/SUNWspro/lib/v9/rtcaudit.so`（对于 64 位 SPARC 平台）

`/opt/SUNWspro/lib/amd64/rtcaudit.so`（对于 AMD 64 平台）

`/opt/SUNWspro/lib`（对于 32 位平台）

预装 `rtcaudit.so`：

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

请仅在需要时再设置 `LD_AUDIT` 环境变量来预装 `rtcaudit.so`；不要将其一直装入。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。

如果想连接的程序从某一其他程序派生或执行，则需要为主程序（派生者）设置 `LD_AUDIT`。`LD_AUDIT` 的设置 在派生和执行中全盘继承。

`LD_AUDIT` 环境变量同时适用于 32 位程序和 64 位程序，这使得很难为 32 位程序选择运行 64 位程序的正确的库，或者为 64 位程序选择运行 32 位程序的正确的库。某些 Solaris OS 版本支持 `LD_AUDIT_32` 环境变量和 `LD_AUDIT_64` 环境变量，它们分别只影响 32 位程序和 64 位程序。有关正在运行的 Solaris OS 的版本，请参见《链接程序和库指南》，以确定是否支持这些变量。

同时使用修复并继续与运行时检查

可以将运行时检查与修复并继续一起使用，以快速查出和修复编程错误。修复并继续提供了强大的组合功能，可以为您节省大量的调试时间。以下是一个示例：

```
% 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 rtd /usr/lib/ld.so.1
Reading symbolic information for librt.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 `a.out' have been changed (fixed):
bug.c
Remember to remake program.

```

有关使用修复并继续的详细信息，请参见第 10 章。

运行时检查应用编程接口

泄露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明了程序中的所有内存分配和释放情况。许多应用程序在 `malloc()` 或 `free()` 函数的基础上或独立编写自己的内存管理例程。如果您使用自己的分配器（称为**专用分配器**），运行时检查便无法自动跟踪它们，这样您就无从知晓由于不当使用而导致的泄露和内存访问错误。

不过，运行时检查提供了一个 API 以便使用专用分配器。利用此 API 可将专用分配器视同标准堆分配器来对待。API 自身在头文件 `rtc_api.h` 中提供，并作为 Sun Studio 软件的一部分提供。手册页 `rtc_api(3x)` 详细说明了运行时检查 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。错误报告将不包括分配项。

在批处理模式下使用运行时检查

`bcheck` 实用程序是 `dbx` 的运行时检查功能的一个方便的批处理接口。它在 `dbx` 下运行程序，并在默认情况下，将运行时检查错误输出置于默认文件 `program.errs` 中。

`bcheck` 实用程序可以单独或一并执行内存泄漏检查、内存访问检查、内存使用检查这三种检查。其默认操作为只执行泄漏检查。有关其使用方法的详细信息，请参见 `bcheck(1)` 手册页。

注 - 在运行 64 位 Linux OS 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` 的语法为：

```
bcheck [-V] [-access | -all | -leaks | -memuse] [-o logfile] [-q]
[-s script] program [args]
```

使用 `-o logfile` 选项为日志文件指定另一名称。在执行程序前使用 `-s script` 选项在包含于文件 `script` 中的 `dbx` 命令中读取。`script` 文件通常包含像 `suppress` 和 `dbxenv` 这样的命令来处理 `bcheck` 实用程序的错误输出。

`-q` 选项可使 `bcheck` 实用程序完全安静，返回时具有与程序相同的状态。想在脚本或 `makefiles` 中使用 `bcheck` 实用程序时，此选项很有用。

bcheck 示例

要对 `hello` 只执行泄漏检查，请键入：

```
bcheck hello
```

要对带参数 5 的 `mach` 只执行访问检查，请键入：

```
bcheck -access mach 5
```

要对 `cc` 安静地执行内存使用检查，并以正常退出状态退出，请键入：

```
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。所有错误输出会被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，会生成完整的栈回溯，并会重定向到错误日志文件。可使用 `dbx` 环境变量 `stack_max_size` 来控制栈帧数。

如果文件 `logfile` 已存在，则在将批处理输出重定向到该文件前，`bcheck` 会清除该文件的内容。

直接在 `dbx` 中启用批处理模式

也可以直接在 `dbx` 中通过设置 `dbx` 环境变量 `rtc_auto_continue` 和 `rtc_error_log_file_name` 来启用批处理类模式（请参见第 62 页的“设置 `dbx` 环境变量”）。

如果将 `rtc_auto_continue` 设置为 `on`，运行时检查会继续查找错误，并自动保持运行。它会将错误重定向到以 `dbx` 环境变量 `rtc_error_log_file_name` 命名的文件。（请参见第 62 页的“设置 `dbx` 环境变量”。）默认日志文件名为 `/tmp/dbx.errlog.uniqueid`。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`。

默认情况下，`rtc_auto_continue` 会被设置为 `off`。

疑难解答提示

启用程序的错误检查功能并运行程序后，可能会检测到以下错误之一：

librtc.so 和 dbx 版本不匹配；错误检查被禁用

如果正在对连接的进程使用运行时检查，并将 LD_AUDIT 设置为 rtcaudit.so 版本，而不是随 Sun Studio dbx 映像提供的版本，便可能会发生此错误。要修复此错误，请更改 LD_AUDIT 的设置。

修补区域太远（8mb 限制）；访问检查被禁用

运行时检查无法找到距启用访问检查所必需的装入对象足够近的修补空间。请参见下面的“运行时检查的 8 M 字节限制”。

运行时检查的 8 M 字节限制

下述 8 M 字节限制在基于 UltraSPARC 处理器的硬件上不会再发生，因为其中的 dbx 具有不使用分支便可调用陷阱处理程序的功能。将控制权移交给陷阱处理程序的过程非常缓慢（最多需要 10 倍时间），但不受 8 M 字节限制。只要硬件基于 UltraSPARC 处理器，便会在需要时自动使用陷阱。可以使用系统命令 isalist 检查硬件，同时检查结果中是否包含字符串 sparcv8plus。rtc -showmap 命令（请参见第 344 页的“rtc -showmap 命令”）显示按地址排序的工具类型图表。

如果启用了访问检查，dbx 会用分支到一个补丁区域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 M 字节。这意味着如果被调试程序用完了替换特定装入或存储指令的所有 8 M 字节地址空间，将没有空间来存放补丁区域。

如果运行时检查不能截取对内存的所有装入和存储，便无法提供准确的信息，从而会完全禁用访问检查。泄漏检查不受影响。

dbx 遇到此限制时会在内部应用一些策略，并在能够纠正此问题时继续。在某些情况下，dbx 无法继续；出现这种情况时，它会在打印错误消息后关闭访问检查。

如果遇到此 8 M 字节限制，请尝试以下解决方法。

1. 尝试使用 32 位 SPARC-V8 代替 64 位 SPARC-V9

如果使用 -xarch=v9 选项编译的应用程序遇到 8 M 字节问题，请尝试对该应用程序的 32 位版本进行内存测试。由于 64 位地址需要更长的补丁指令序列，因此使用 32 位地址可以缓解 8 M 字节问题。如果这种解决方法不太奏效，则可以在 32 位和 64 位程序上使用以下方法。

2. 尝试添加补丁区域对象文件。

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一个大型可执行文件或共享库的中间，可以提供更多补丁空间。请参见 `rtc_patch_area(1)` 手册页。

如果 `dbx` 达到 8 M 字节限制，它会告知哪一个装入对象过大（主程序还是共享库），并且会输出该装入对象所需的总补丁空间。

为得到最优结果，特殊的补丁对象文件应当在可执行文件或共享库中均匀分布，并应使用默认大小（8 M 字节）或更小的大小。同样，添加的补丁空间不要超过 `dbx` 所要求的 10-20%。例如，如果 `dbx` 称需要 31 兆字节用于 `a.out`，请添加使用 `rtc_patch_area` 脚本创建的四个对象文件，每个大小为 8 M 字节，在可执行文件中近似平均地进行分布。

如果 `dbx` 在可执行文件中找到显式补丁区域，则会打印补丁区域所跨越的地址范围，这有助于将它们正确地置于链接行上。

3. 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对象文件分组。然后将它们链接成较小的部分。如果大文件是可执行文件，则将其分解成较小的可执行文件和一系列共享库。如果大文件是共享库，则将它分解成一系列较小的库。

`dbx` 可利用这一技术在不同的共享对象间为补丁代码寻找空间。

4. 尝试添加“填充” `.so` 文件。

要连接已启动的进程时，才有必要这样做。

运行时链接程序可能会将各库非常紧密地放置在一起，使在库的间隙中创建补丁空间无法实现。如果 `dbx` 在打开运行时检查的情况下启动可执行文件，它会要求运行时链接程序在共享库间增加一个间隙；但如果在启用运行时检查的情况下连接不是由 `dbx` 启动的进程，这些库的间隙可能会过小。

如果各运行时库间隙过小，（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，然后将它链接到其他共享库间的程序。有关详细信息，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。访问错误和泄露。

访问错误

打开访问检查时，运行时检查会检测并报告以下类型的错误。

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：使用同一指针调用 `free()` 一次以上。在 C++ 中，对同一指针使用删除操作符一次以上。

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 `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：从未半字对齐、字对齐或双字对齐的地址中分别读取 2、4 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、4 或 8 个字节分别写入未半字对齐、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。查找在未对 `malloc()` 的返回值检查 `NULL` (一种常见编程错误) 的情况下发生的问题时会有用。

示例:

```
char *ptr = (char *)malloc(0x7fffffff);
/* Out of Memory (oom), ptr == NULL */
```

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读。

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 c, *a = (char *)malloc(1);
c = a[1];                                   /* Read from unallocated memory (rua) */
```

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读。

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) */
}
```

写入到只读内存 (wro) 错误

问题：尝试写入到只读内存。

可能的原因：写入到文本地址、写入到只读数据区 (.rodata) 或写入到已由 mmap 设置为只读的页。

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 *a = (char *)malloc(1);
a[1] = '\0';          /* Write to unallocated memory (wua) */
```

内存泄漏错误

打开泄漏检查时，运行时检查会报告以下类型的错误。

地址位于块内 (aib) 错误

问题：可能的内存泄漏。缺少对已分配内存块开始处的引用，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发生增值。

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。尚未释放已分配的块，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，而不保存在内存中，自然会出现这种错误。编译器往往会在打开优化的情况下对局部变量和函数参数这样处理。如果在未打开优化的情况下出现这种错误，则可能是真正的内存泄漏。如果指向已分配内存块的唯一指针在块被释放前超出作用域，便会发生这种情况。

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：尚未释放已分配内存块，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

修复并继续

使用 `fix` 命令可以快速重新编译编辑过的本地源代码而不用停止调试进程。无法使用 `fix` 命令重新编译 Java 代码。

注 - 在 Linux 平台上不能使用 `fix` 命令。

本章由以下部分组成：

- 使用修复并继续
- 修复程序
- 修复后更改变量
- 修改头文件
- 修复 C++ 模板定义

使用修复并继续

修复并继续功能允许修改和重新编译本地源文件并继续执行，而无需重新生成整个程序。通过更新 `.o` 文件并将其拼接到程序中，不需要进行重新链接。

使用修复并继续的优点是：

- 不必重新链接程序。
- 不必重新装入调试程序。
- 可以从修复位置恢复运行程序。

注 - 请不要在生成过程中使用 `fix` 命令。

修复并继续如何操作

使用 `fix` 命令之前，必须先 在编辑器窗口中编辑源码。（有关修改代码的各种方法，请参见第 160 页的“使用修复并继续修改源码”）。保存更改后，请键入 `fix`。有关 `fix` 命令的信息，请参见第 310 页的“`fix` 命令”。

调用 `fix` 命令后，`dbx` 会使用适当的编译器选项来调用编译器。编译已修改的文件并创建共享对象（`.so`）文件。通过比较新旧文件进行语义测试。

使用运行时链接程序将新目标文件链接到正在运行的进程中。如果正在修复栈顶部的函数，则新的停止于函数便是新函数中同一行的开始。旧文件中的所有断点均被移动到新文件中。

可以在使用或未使用调试信息进行编译的文件上使用修复并继续，但是 `fix` 命令和 `cont` 命令在功能上对于最初未使用调试信息编译的文件有些限制。有关详细信息，请参见第 310 页的“`fix` 命令”中的 `-g` 选项说明。

可以修复共享对象（`.so`）文件，但是必须在特定模式中打开它们。在调用 `dlopen` 函数时，可以使用 `RTLD_NOW|RTLD_GLOBAL` 或 `RTLD_LAZY|RTLD_GLOBAL`。

使用修复并继续修改源码

使用修复并继续时，可以通过以下方式修改源代码：

- 添加、删除或更改函数中的各代码行
- 添加或删除函数
- 添加或删除全局变量和静态变量

将函数从旧文件映射到新文件时，可能会出现 问题。在编辑源文件时，要尽可能减少此类问题：

- 不要更改函数的名称。
- 不要添加、删除参数或将其类型更改为函数。
- 不要添加、删除当前在栈中处于活动状态的函数中的局部变量或更改其类型。
- 不要对模板声明或模板实例进行修改。只有 C++ 模板函数定义的函数体可以被更改。

如果作出以上任何更改，请重新生成整个程序而非使用修复并继续。

修复程序

可以在作出更改后使用 `fix` 命令重新链接源文件，而无需重新编译整个程序。然后可继续执行程序。

要修复文件：

1. 将所作更改保存到源文件。
2. 在 dbx 提示符处键入 **fix**。

虽然修复的次数不受限制，但是如果在一行中进行了多次修复，则需要考虑重新生成程序。fix 命令更改内存中的程序映像，但不更改磁盘上的程序映像。当您进行更多的修复时，内存映像将和磁盘中的映像变得不同步。

fix 命令不会在可执行文件中进行更改，而是仅更改 .o 文件和内存映像。完成程序调试后，必须重新生成程序，以将所作更改合并到可执行文件中。退出调试时，会出现一条消息提醒您重新生成程序。

如果调用 fix 命令时使用 -a 之外的某个选项且不带文件名参数，则只修复当前修改的源文件。

调用 fix 时，会在执行编译日之前搜索当前正在编译的文件的当前工作目录。由于文件系统结构从编译时间到调试时间发生更改，可能会在查找正确目录时出现问题。为避免此问题，请使用 pathmap 命令，它会创建从一个路径名到另一个路径名的映射。该映射被应用于源路径和目标文件路径。

修复后继续

可以使用 cont 命令继续执行（请参见第 296 页的“cont 命令”）。

在恢复程序执行之前，需要注意决定更改结果的以下条件。

更改已执行的函数

如果在已执行的函数中作出更改，此更改将无效，直至：

- 重新运行程序
- 下次调用此函数

如果所作修改并不仅仅只涉及到对变量的简单更改，请使用 fix 命令，然后使用 run 命令。使用 run 命令比较快，因为它并不重新链接程序。

更改一个尚未被调用的函数

如果在未被调用的函数中作出更改，所作更改将在调用此函数时生效。

更改当前正在执行的函数

如果更改当前正在执行的函数，`fix` 命令的影响取决于更改相对于停止于函数的位置：

- 如果在已执行的代码中更改，则不会重新执行该代码。通过将当前函数弹出栈（请参见第 336 页的“`pop` 命令”）并从更改的函数被调用的地方继续来执行代码。您需要充分了解代码，以便确定函数是否具有无法撤消的副作用（例如，打开一个文件）。
- 如果在将要执行的代码中更改，则运行新代码。

更改当前位于栈中的函数

如果已更改当前位于栈中的函数，但未更改停止于函数，则此函数的当前调用不使用更改的代码。停止于函数返回时，执行栈中函数的旧版本。

下列几种方法可以解决此问题：

- 使用 `pop` 命令弹出栈，直至所有更改的函数均从栈中删除。您需要了解代码，以确保不会发生问题。
- 使用 `cont at line_number` 命令从另一行继续。
- 在继续前手动修复数据结构（使用 `assign` 命令）。
- 使用 `run` 命令重新运行程序。

如果栈中修改的函数内存在断点，断点将被移动到函数的新版本中。如果执行旧版本，程序就不会停止在这些函数中。

修复后更改变量

`pop` 命令或 `fix` 命令无法撤消对全局变量所作的更改。要手动为全局变量重新指定正确的值，请使用 `assign` 命令。（请参见第 281 页的“`assign` 命令”。）

下例显示如何修复简单的错误。在尝试非关联化 NULL 指针时，应用程序于第 6 行发生段故障。

```
dbx[1] list 1,$
1  #include <stdio.h>
2
3  char *from = "ships";
4  void copy(char *to)
5  {
6      while ((*to++ = *from++) != '\0');
7      *to = '\0';
8  }
9
10 main()
11 {
12     char buf[100];
13
14     copy(0);
15     printf("%s\n", buf);
16     return 0;
17 }
(dbx) run
Running: testfix
(process id 4842)
signal SEGV (no mapping at the fault address) in copy at line 6
in file "testfix.cc"
6      while ((*to++ = *from++) != '\0');
```

将第 14 行更改为 `copy(buf);` 而不是 `0` 并保存文件，然后进行修复：

```
14     copy(buf); <=== modified line
(dbx) fix
fixing "testfix.cc" .....
pc moved to "testfix.cc":6
stopped in copy at line 6 in file "testfix.cc"
6      while ((*to++ = *from++) != '\0');
```

如果程序从这里继续，仍会发生段故障，因为零指针仍然被置于栈中。使用 `pop` 命令弹出一个栈帧：

```
(dbx) pop
stopped in main at line 14 in file "testfix.cc"
14 copy(buf);
```

如果程序从这里继续，则运行程序但不打印正确的值，因为全局变量 `from` 已经执行加一操作。程序将打印 `hips` 而不是 `ships`。使用 `assign` 命令恢复全局变量，然后使用 `cont` 命令。现在程序打印正确的字符串：

```
(dbx) assign from = from-1
(dbx) cont
ships
```

修改头文件

有时可能有必要修改头 (`.h`) 文件和源文件。为确保修改的头文件可被包含它的程序中的所有源文件访问，必须作为 `fix` 命令的一个参数给出包含此头文件的所有源文件列表。如果不给出源文件列表，则只重新编译主（当前）源文件，并且只有它包含头文件的修改版本。程序中的其他源文件继续包含此头文件的原始版本。

修复 C++ 模板定义

无法直接修复 C++ 模板定义。改用模板实例修复文件。如果未更改模板定义文件，则可以使用 `-f` 选项覆盖日期检查。对于使用 Sun Studio C 编译器编译的程序，`dbx` 会在默认系统信息库目录 `SunWS_cache` 中查找模板定义 `.o` 文件。在 `dbx` 中，`fix` 命令不支持 `-ptr` 编辑器选项。

第11章

调试多线程应用程序

dbx 可以调试使用 Solaris 或 POSIX 线程的多线程应用程序。利用 dbx 可以检查每个线程的栈跟踪、恢复所有线程、step 或 next 特定线程及在线程间导航。

dbx 通过检测程序是否利用 libthread.so 来识别多线程程序。程序使用 libthread.so 的方法有两种：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。

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。

本章由以下部分组成：

- 了解多线程调试
- 理解 LWP 信息

了解多线程调试

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 目录中。

dbx 是同步式的；当某个线程或轻量进程 (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 Solaris 《多线程编程指南》。

线程信息

dbx 中提供有以下线程信息:

```
(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。其中的 `number` 为 `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 (请参见表 11-1 中有关线程状态的说明。)
- 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`, 一种 dbx 类型线程 ID
- 线程状态 (请参见表 11-1 中有关线程状态的说明。)
- 加单引号的线程名
- 说明线程优先级的数字

表 11-1 线程和 LWP 状态

线程和 LWP 状态	描述
挂起	线程已被显式挂起。
可运行	线程可运行, 并正等待作为计算资源的 LWP。
僵停	分离的线程退出 (<code>thr_exit</code>) 后, 在使用 <code>thr_join</code> () 重新连接前, 将一直处于僵停状态。 <code>THR_DETACHED</code> 是在线程创建 (<code>thr_create</code> ()) 时指定的标志。退出的非分离线程在被获得前将一直处于僵停状态。

表 11-1 线程和 LWP 状态（续）

线程和 LWP 状态	描述
在 <i>syncobj</i> 上休眠	线程在给定同步对象上被阻塞。视 <code>libthread</code> 和 <code>libthread_db</code> 所提供的支持级别， <i>syncobj</i> 可能会如十六进制地址一般简单，也可能包含更多的信息内容。
活动	线程在某 LWP 中处于活动状态，但 <code>dbx</code> 无法访问该 LWP。
未知	<code>dbx</code> 无法确定状态。
<i>lwpsstate</i>	绑定或活动线程状态具有与之关联的 LWP 的状态。
运行	LWP 正在运行，但因响应另一 LWP 而同步停止。
syscall <i>num</i>	LWP 进入给定系统调用 # 时停止。
syscall 返回 <i>num</i>	LWP 退出给定系统调用 # 时停止。
作业控制	LWP 因作业控制而停止。
LWP 挂起	LWP 在内核中被阻塞。
单步递阶	LWP 刚刚完成一步。
断点	LWP 刚刚遇到断点。
fault <i>num</i>	LWP 招致给定错误 #。
signal <i>name</i>	LWP 招致给定信号。
进程同步	此 LWP 所属进程刚刚开始执行。
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。语法为：

```
thread [-blocks] [-blockedby] [-info] [-hide] [--unhide] [-suspend] [-resume] thread_id
```

要显示当前线程，请键入：

```
thread
```

要切换到线程 *thread_id*，请键入：

```
thread thread_id
```

有关 thread 命令的详细信息，请参见第 361 页的“thread 命令”。

查看线程列表

要查看线程列表，请使用 threads 命令。语法为：

```
threads [-all] [-mode [all|filter] [auto|manual]]
```

要打印所有已知线程的列表，请键入：

```
线程
```

要打印通常不输出的线程（僵停），请键入：

```
threads -all
```

有关线程列表的说明，请参见第 166 页的“线程信息”。

有关 threads 命令的详细信息，请参见第 363 页的“threads 命令”。

恢复执行

使用 cont 命令恢复程序执行。因为线程目前使用同步断点，所以所有线程都会恢复执行。

了解线程创建活动

通过使用 `thr_create` 事件和 `thr_exit` 事件，可以了解应用程序创建和销毁线程的频率，如下例所示：

```
(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
```

这里应用程序创建了三个线程。请注意，线程退出的顺序与创建它们时的相反，这可能指示应用程序具有更多的线程，线程将累积并消耗资源。

要获取更重要的信息，可以在不同会话中尝试以下内容：

```
(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
```

输出显示所有三个线程都是由线程 `t@1`（它是通用的多线程模式）创建的。

假定要从其开头调试线程 `t@3`。您可以在创建线程 `t@3` 时停止应用程序，如下所示：

```
(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)
```

如果应用程序偶尔从线程 t@5（而不是线程 t@1）产生新线程，则可以捕获该事件，如下所示：

```
(dbx) stop thr_create -thread t@5
```

理解 LWP 信息

通常无需注意 LWP。但有时无法完成线程级查询。在这些情况下，请使用 `lwps` 命令来显示 LWP 的相关信息。

```
(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 列表的每行都包含以下内容：

- *（星号）表示此 LWP 内发生了需要用户处理的事件。
- 箭头表示为当前 LWP。
- l@number 指特定 LWP。
- 下一项代表 LWP 状态。
- in function_name() 标识 LWP 当前正在执行的函数。

第12章

调试子进程

本章说明如何调试子进程。dbx 提供了几种工具，可帮助您调试使用 `fork(2)` 和 `exec(2)` 函数创建子进程的进程。

本章由以下部分组成：

- 连接到子进程
- 跟随 `exec` 函数
- 跟随 `fork` 函数
- 与事件交互

连接到子进程

可以通过以下方法之一连接某个正在运行的子进程。

- 启动 dbx 时：

```
$ dbx program_name process_id
```

- 从 dbx 命令行中：

```
(dbx) debug program_name process_id
```

可以将 `program_name` 替换成 `name -`（减号），这样 dbx 便会查找与给定进程 ID (`process_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完整路径名。

也可以使用 Sun Studio IDE 中的“调试器”窗口连接正在运行的子进程。（请参见 IDE 联机帮助中的 dbx 调试器帮助集中的“连接进程”。）

跟随 exec 函数

如果子进程使用 `exec(2)` 函数或其中一个变量来执行新程序，则进程 `id` 保持不变，但进程映像发生更改。dbx 自动记录对 `exec()` 函数的调用并隐式重新加载最近执行的程序。

可执行程序的原名称保存在 `$oprog` 中。要返回原名称，请使用 `debug $oprog` 命令。

跟随 fork 函数

如果子进程调用 `vfork()`、`fork(1)` 或 `fork(2)` 函数，则进程 `id` 发生更改，但进程映像保持不变。根据 dbx 环境变量 `follow_fork_mode` 的设置情况，dbx 将执行以下操作之一。

Parent	在传统行为中，dbx 将忽略派生而跟随父进程。
Child	dbx 使用新进程 ID 自动切换到派生的子进程。原始父进程的所有连接和认识均丢失。
Both	此模式只有在通过 Sun Studio IDE 使用 dbx 时才有效。
Ask	只要 dbx 检测到派生，便会提示您选择 <code>parent</code> 、 <code>child</code> 、 <code>both</code> 或 <code>stop to investigate</code> 。如果选择 <code>stop</code> ，便可以检查程序的状态，然后键入 <code>cont</code> 继续执行；这时会提示选择哪种方式继续。

与事件交互

为所有 `exec()` 或 `fork()` 进程删除全部断点和其他事件。可以通过将 dbx 环境变量 `follow_fork_inherit` 设置为 `on` 来覆盖派生进程的删除，或者使用 `-perm eventspec` 修饰符来保持事件的持久性。有关使用事件规范修饰符的详细信息，请参见附录 B。

第13章

调试 OpenMP 程序

OpenMP™ 应用编程接口 (API) 是多家计算机供应商为共享内存多处理器体系结构联合开发的可移植并行程序设计模型。它基于 dbx 通用多线程调试功能，支持使用 dbx 调试 Fortran 和 C OpenMP 程序。在线程和 LWP 上运行的所有 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。

本章由以下部分组成：

- 编译器如何转换 OpenMP 代码
- OpenMP 代码可用的 dbx 功能
- 使用带 OpenMP 代码的栈跟踪
- 在 OpenMP 代码上使用 dump 命令
- OpenMP 代码的执行序列

有关组成“OpenMP 2.0 版本应用程序接口”并通过 Sun Studio Fortran 95 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《OpenMP API 用户指南》。

注 – OpenMP 调试仅在运行 Solaris OS 的平台上可用。该功能在 Linux 平台上不适用。

编译器如何转换 OpenMP 代码

为了更好地说明 OpenMP 调试，了解编译器如何转换 OpenMP 代码是非常有帮助的。请参见以下 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，在本例中其名称是 `_$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 将外联子例程视作其他函数进行处理，但是无法使用内部生成的名称在函数中显式设置断点。

OpenMP 代码可用的 dbx 功能

除了调试多线程程序的常用功能外，dbx 还允许在 OpenMP 程序中进行如下操作：

- **单步步入并行区域。**因为并行区域是外联的并且从 OpenMP 运行时库中进行调用，所以每一步执行实际涉及几个层的由为此目的而创建的从属线程执行的运行时库调用。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，请参见第 174 页的“编译器如何转换 OpenMP 代码”中的 Fortran 代码，假定主线程 t@1 位于第 10 行。您已单步步入第 12 行，并为执行运行时库调用而创建从属线程 t@2、t@3 和 t@4。线程 t@3 首先到达断点并引起程序停止执行。因此，由线程 t@1 启动的单步执行在线程 t@3 上结束。此行为不同于正常的单步执行（单步执行后通常仍处在与此之前相同的线程上）。

- **打印共享、专用和线程专用变量。**dbx 可以打印所有共享、专用和线程专用变量。如果尝试打印并行区域之外的线程专用变量，则打印主线程的副本。whatis 命令不能判断一个变量是共享、专用还是线程专用变量。

使用带 OpenMP 代码的栈跟踪

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和几个运行时库调用的栈跟踪。使用第 174 页的“编译器如何转换 OpenMP 代码”中的 Fortran 示例，在第 15 行停止执行，where 命令将生成以下栈跟踪。

```
[t@4 l@4]: where
current thread: t@4
=>[1] _$d1A12.MAIN_(), line 15 in "example.f90"
[2] __mt_run_my_job_(0x45720, 0xff82ee48, 0x0, 0xff82ee58, 0x0,
0x0), at 0x16860
[3] __mt_SlaveFunction_(0x45720, 0x0, 0xff82ee48, 0x0, 0x455e0,
0x1), at 0x1aaf0
```

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。其他两帧用于运行时库调用。

当执行在并行区域中停止时，来自从属线程的 `where` 命令并未使栈回溯至其父线程（如上例所示）。但来自主线程的 `where` 命令却具有完全回溯：

```
[t@4 l@4]: thread t@1
t@1 (l@1) stopped in _$d1A12.MAIN_ at line 15 in file "example.f90"
15          sum = sum + a(i)
[t@1 l@1]: where
current thread: t@1
=>[1] _$d1A12.MAIN_(), line 15 in "example.f90"
[2] __mt_run_my_job_(0x41568, 0xff82ee48, 0x0, 0xff82ee58, 0x0,
0x0), at 0x16860
[3] __mt_MasterFunction_(0x1, 0x0, 0x6, 0x0, 0x0, 0x40d78), at
0x16150
[4] MAIN(), line 12 in "example.f90"
```

如果线程为数不多，可以通过使用 `threads` 命令（请参见第 363 页的“`threads` 命令”）列出所有线程，然后切换到各个线程以确定哪个是主线程来确定执行如何到达从属线程中的断点。

在 OpenMP 代码上使用 `dump` 命令

当执行在并行区域中停止时，`dump` 命令可以打印多个专用变量副本。下例中，`dump` 命令打印变量 `i` 的两个副本：

```
[t@1 l@1]: dump
i = 1
sum = 0.0
a = ARRAY
i = 1000001
```

因为外联例程作为主机例程的嵌套函数执行，而专用变量作为外联例程的局部变量执行，所以会打印变量 `i` 的两个副本。由于 `dump` 命令打印作用域内的所有变量，所以主机例程中的 `i` 和外联例程中的 `i` 均会被打印。

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14章

处理信号

本章说明如何使用 `dbx` 处理信号。`dbx` 支持 `catch` 命令，当 `dbx` 检测到捕获列表中出现的任何信号时，该命令会指示 `dbx` 停止程序。

`dbx` 命令 `cont`、`step` 和 `next` 均支持 `-sig signal_name` 选项，该选项可以恢复程序的执行，使程序的行为像收到了在 `cont -sig` 命令中指定的信号一样。

本章由以下部分组成：

- 了解信号事件
- 捕获信号
- 向程序发送信号
- 自动处理信号

了解信号事件

当信号要传送给调试中的进程时，该信号会通过 `kernel` 被重定向到 `dbx`。这时，通常会收到一个提示。而您有两种选择：

- 恢复程序时“取消”信号 — `cont` 命令的默认行为，便于使用图 14-1 所示的 `SIGINT` (Control-C) 进行中断和恢复。

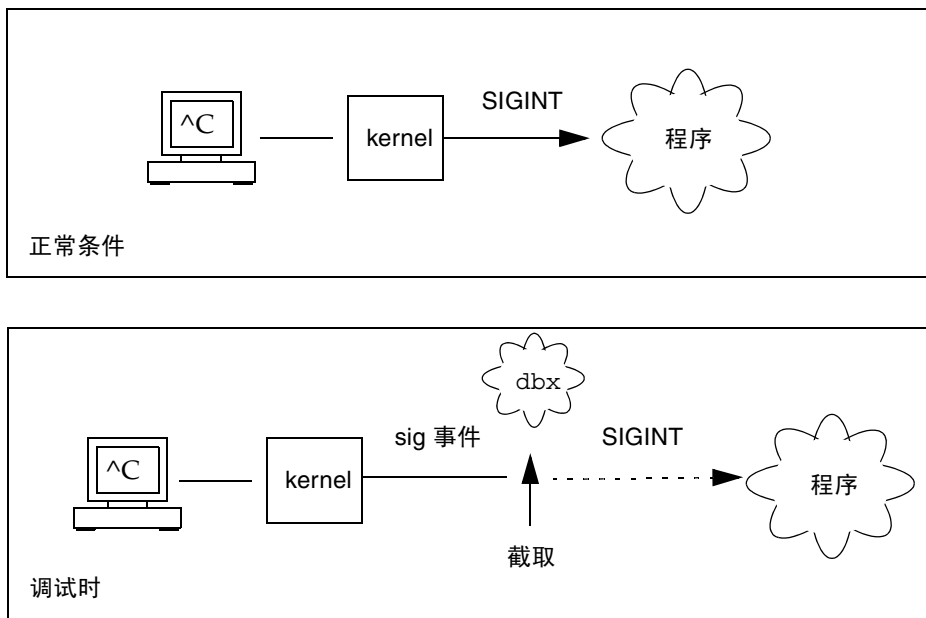


图 14-1 截取和取消 SIGINT 信号

- 使用下列命令将信号“转发”到进程：

```
cont -sig signal
```

signal 可以是信号名或信号编号。

此外，如果频繁接收某一信号，但又不想显示它，可以让 dbx 来自动转发该信号：

```
ignore signal # "ignore"
```

但是，*signal* 仍将转发到进程。信号的默认设置即以此种方式自动转发（请参见第 317 页的“ignore 命令”）。

捕获信号

默认情况下，捕获列表包含多达 33 个可检测信号。（信号数目取决于操作系统和版本。）可通过从默认捕获列表中添加或删除信号来更改默认捕获列表。

注 - dbx 所接受的信号名列表包含 dbx 支持的 Solaris 操作环境各版本的所有信号。因此，dbx 可能会接受当前正在运行的 Solaris 操作环境版本所不支持的信号。例如，即使正在运行 Solaris 7 OS，dbx 也可能会接受 Solaris 9 OS 所支持的信号。有关正在运行的 Solaris OS 所支持的信号列表，请参见 `signal` 手册页（在第三部分的开头）。

要查看当前正在捕获的信号列表，请键入不带 `signal` 参数的 `catch` 命令。

```
(dbx) catch
```

要查看当程序检测到信号时被 dbx 忽略的信号列表，请键入不带 `signal` 参数的 `ignore` 命令。

```
(dbx) ignore
```

更改默认信号列表

可通过在列表之间移动信号名来控制究竟哪个信号将导致程序停止。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 Solaris 平台）

使用要求进行浮点计算的代码编程的编程人员经常需要调试程序中生成的异常。当出现被零除或上溢之类的浮点异常时，系统会返回一个合理的答案，作为导致异常的操作的结果。返回合理答案后，程序会继续安静地执行。Solaris OS 执行异常合理答案之“二进制浮点运算 IEEE 标准”定义。

由于返回浮点异常的合理答案，所以异常不会自动触发信号 `SIGFPE`。默认情况下，某些整数异常（如整数被零除和整数上溢）会触发信号 `SIGFPE`。

要查找导致异常的原因，需要在程序内设置陷阱处理程序，这样异常便会触发 `SIGFPE` 信号。（请参见 `ieee_handler(3m)` 手册页中的陷阱处理程序示例。）

可使用下列方式启用陷阱：

- `ieee_handler`
- `fpsetmask`（请参见 `fpsetmask(3c)` 手册页）

- `-ftrap` 编译器标志（对于 Fortran 95 而言，请参见 f95(1) 手册页）

使用 `ieee_handler` 命令设置陷阱处理程序时，将设置硬件浮点状态寄存器中的陷阱启用屏蔽。此陷阱启用屏蔽会导致异常在运行时引发 `SIGFPE` 信号。

使用陷阱处理程序编译程序后，将其载入 `dbx`。在捕获 `SIGFPE` 信号之前，必须将 `FPE` 添加到 `dbx` 信号捕获列表。

```
(dbx) catch FPE
```

默认情况下，`FPE` 位于忽略列表。

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。当出现要捕获的异常时，便引发 `SIGFPE` 信号，同时 `dbx` 停止程序。然后，可以使用 `dbx where` 命令跟踪调用栈来帮助查找程序中出现异常的特定行号（请参见第 377 页的“`where` 命令”）。

确定导致异常的原因

要确定导致异常的原因，请使用 `regs -f` 命令显示浮点状态寄存器 (FSR)。查看寄存器内产生的异常 (`aexc`) 和当前异常 (`cexc`) 字段，其中包含以下浮点异常条件位：

- 无效操作数
- 溢出
- 下溢
- 被零除
- 不精确结果

有关浮点状态寄存器的更多信息，请参见《SPARC 体系结构手册》的第 8 版 (V8) 或第 9 版 (V9)。要查看更多讨论和示例，请参见《数值计算指南》。

向程序发送信号

`dbx cont` 命令支持 `-sig signal` 选项，该选项可以恢复执行程序，而程序的行为就如同收到系统信号 `signal` 一样。

例如，如果程序具有 SIGINT (^C) 中断处理程序，则可以键入 ^C 来停止应用程序，并使控制返回 dbx。如果发出 cont 命令本身来继续执行程序，便永远不会执行中断处理程序。要执行中断处理程序，请将信号 SIGINT 发送给程序：

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

step、next 和 detach 命令均亦接受 -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15章

使用 dbx 调试 C++

本章说明 dbx 如何处理 C++ 异常和调试 C++ 模板，包括完成这些任务所使用的命令摘要以及代码样例示例。

本章由以下部分组成：

- 使用 dbx 调试 C++
- dbx 中的异常处理
- 使用 C++ 模板调试

有关编译 C++ 程序的详细信息，请参见第 54 页的“编译调试程序”。

使用 dbx 调试 C++

虽然本章内容集中在调试 C++ 的两个特定方面，但是 dbx 允许您在调试 C++ 程序时使用全部功能。可以：

- 查找类和类型的相关定义（请参见第 81 页的“查找类型和类的定义”）
- 打印或显示继承的数据成员（请参见第 116 页的“打印 C++”）
- 查找有关对象指针的动态信息（请参见第 116 页的“打印 C++”）
- 调试虚函数（请参见第 92 页的“调用函数”）
- 使用运行时类型信息（请参见第 116 页的“打印变量、表达式或标识符的值”）
- 在某个类的所有成员函数上设置断点（请参见第 98 页的“在相同类的成员函数中设置断点”）
- 在所有重载的成员函数上设置断点（请参见第 98 页的“在不同类的成员函数中设置断点”）
- 在所有重载的非成员函数上设置断点（请参见第 99 页的“在非成员函数中设置多个断点”）
- 在特定对象的所有成员函数上设置断点（请参见第 99 页的“在对象中设置断点”）

- 处理重载的函数或数据成员（请参见第 97 页的“在函数中设置 stop 断点”）

dbx 中的异常处理

如果出现异常，程序便停止运行。异常向编程异常发送信号，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，dbx 允许您：

- 在堆栈解退之前捕获未处理的异常
- 捕获意外的异常
- 在堆栈解退之前捕获已处理的或未处理的特定异常
- 当异常出现在程序的特定点时确定其捕获位置

如果在程序停止于异常抛出点后给出 `step` 命令，则将于堆栈解退期间执行的第一个析构函数的起始点处返回控制。如果步出于堆栈解退期间执行的析构函数，则将于下一个析构函数的起始点处返回控制。当所有析构函数都执行完毕时，`step` 命令可带您进入处理异常抛出的 `catch` 块。

异常处理命令

`exception [-d | +d]` 命令

在调试期间，可以随时使用 `exception` 命令来显示异常的类型。如果使用不带选项的 `exception` 命令，所显示的类型将取决于 dbx 环境变量 `output_dynamic_type` 的设置：

- 如果设置为 `on`，则显示派生类型。
- 如果设置为 `off`（默认值），则显示静态类型。

指定 `-d` 或 `+d` 选项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，请参见第 308 页的“`exception` 命令”。

`intercept [-all] [-x] [-set] [typename] 命令`

可以在堆栈解退之前截取或捕获特定类型的异常。使用不带参数的 `intercept` 命令可列出正在截取的类型。使用 `-all` 可截取所有异常。使用 `typename` 可将类型添加到截取列表中。使用 `-x` 可将特定类型添加到排除列表中，以防止它被截取。使用 `-set` 可同时清除截取列表和排除列表，并将列表设置为仅截取或排除指定类型的抛出。

例如，要截取除 `int` 之外的所有类型，请键入：

```
(dbx) intercept -all -x int
```

要截取 `Error` 类型的异常，请键入：

```
(dbx) intercept Error
```

截取过多 `CommonError` 异常后，可以通过键入以下内容排除这些异常：

```
(dbx) intercept -x CommonError
```

键入不带参数的 `intercept` 命令，则会显示截取列表中包括未处理的异常和未预料到的异常（默认情况下，它们将被截取）以及 `Error` 类的异常（`CommonError` 类的异常除外）。

```
(dbx) intercept  
-unhandled -unexpected class Error -x class CommonError
```

如果后来意识到 `Error` 不是您所关注的异常类，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尝试截取除 `Error` 类的异常之外的所有异常：

```
(dbx) intercept -all -x Error
```

有关详细信息，请参见第 318 页的“`intercept` 命令”。

`unintercept [-all] [-x] [typename] 命令`

使用 `unintercept` 命令可从截取列表或排除列表中删除异常类型。使用不带参数的命令列出正在截取的类型（与 `intercept` 命令相同）。使用 `-all` 可从截取列表中删除所有类型。使用 `typename` 可从截取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。

有关详细信息，请参见第 371 页的“`unintercept` 命令”。

whocatches *typename* 命令

如果在当前执行点抛出，whocatches 命令会报告捕获 *typename* 类型异常的位置。使用此命令可查出如果异常自栈的顶帧抛出将会发生什么情况。

显示捕获 *typename* 的 *catch* 子句的行号、函数名和帧号。如果捕获点所在函数与执行抛出的函数相同，则命令返回“*type* 未处理”。

有关详细信息，请参见第 380 页的“whocatches 命令”。

异常处理示例

本例演示如何在 dbx 中使用包含异常的示例程序进行异常处理。int 类型的异常将在函数 bar 中被抛出，并在以下的 catch 块中被捕获。

```
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 }
```

以下示例程序抄本显示的是 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         }
```

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载入 dbx，并调用将在类和函数上使用的模板上的任何 dbx 命令，例如：

- 在类和函数模板实例化中设置断点（请参见第 194 页的“stop inclass classname 命令”、第 194 页的“stop infunction name 命令”和第 194 页的“stop in function 命令”）
- 打印所有类和函数模板实例化列表（请参见第 192 页的“whereis name 命令”）
- 显示模板和实例的定义（请参见第 192 页的“whatis name 命令”）
- 调用成员模板函数和函数模板实例化（请参见第 195 页的“call function_name (参数) 命令”）
- 打印函数模板实例化的值（请参见第 195 页的“print 表达式”）
- 显示函数模板实例化的源代码（请参见第 196 页的“list 表达式”）

模板示例

以下代码示例显示类模板 Array 及其实例化和函数模板 square 及其实例化。

```
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>` 是类模板实例化（模板类）
- `Array<int>::getlength` 是模板类的成员函数
- `square(int, int*)` 和 `square(double, double*)` 是函数模板实例化（模板函数）

C++ 模板的命令

在模板和模板实例化中使用这些命令。知道类或类型定义后，便可以打印值、显示源码列表或设置断点。

`whereis name` 命令

使用 `whereis` 命令打印函数或类模板的函数或类实例化的所有具体值列表。

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*)
```

`__type_0` 参数引用 0 号模板参数。`__type_1` 将引用下一模板参数。

有关详细信息，请参见第 379 页的“`whereis` 命令”。

`whatis name` 命令

使用 `whatis` 命令打印函数和类模板以及实例化的函数和类的定义。

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);
```

对于类模板实例化:

```
(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<double>();  
private:  
    int length;  
    double *array;  
};
```

对于函数模板实例化:

```
(dbx) whatis square(int, int*)  
void square(int num, int *result);
```

有关详细信息, 请参见第 373 页的 “whatis 命令”。

stop inclass *classname* 命令

要停止在模板类的所有成员函数中:

```
(dbx) stop inclass Array  
(2) stop inclass Array
```

使用 stop inclass 命令, 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>
```

有关详细信息, 请参见第 353 页的 “stop 命令” 和第 262 页的 “inclass *classname* [-recurse | -norecurse]”。

stop infunction *name* 命令

使用 stop infunction 命令, 在指定函数模板的所有实例中设置断点:

```
(dbx) stop infunction square  
(9) stop infunction square
```

有关详细信息, 请参见第 353 页的 “stop 命令” 和第 262 页的 “infunction *function*”。

stop in *function* 命令

使用 stop in 命令, 在模板类的成员函数或模板函数中设置断点。

对于类模板实例化的成员：

```
(dbx) stop in Array<int>::Array(int l)
(2) stop in Array<int>::Array(int)
```

对于函数实例化：

```
(dbx) stop in square(double, double*)
(6) stop in square(double, double*)
```

有关详细信息，请参见第 353 页的“stop 命令”和第 261 页的“in function”。

call function_name (参数) 命令

当停止在作用域中时，使用 call 命令可显式调用函数实例化和类模板的成员函数。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)
```

有关详细信息，请参见第 284 页的“call 命令”。

print 表达式

使用 print 命令求函数实例化或类模板成员函数的值。

```
(dbx) print iarray.getlength()
iarray.getlength() = 5
```

使用 print 求 this 指针的值。

```
(dbx) whatis this
class Array<int> *this;
(dbx) print *this
*this = {
    length = 5
    array   = 0x21608
}
```

有关详细信息，请参见第 337 页的“print 命令”。

list 表达式

使用 list 命令打印指定函数实例化的源码列表。

```
(dbx) list square(int, int*)
```

有关详细信息，请参见第 323 页的“list 命令”。

第 16 章

使用 dbx 调试 Fortran

本章介绍可能会用于 Fortran 的 dbx 功能。另外还给出了一些 dbx 请求示例，以便为您提供使用 dbx 调试 Fortran 代码时提供帮助。

本章包括以下主题：

- 调试 Fortran
- 调试段故障
- 定位异常
- 跟踪调用
- 处理数组
- 显示内在函数
- 显示复数表达式
- 显示逻辑操作符
- 查看 Fortran 95 派生类型
- 指向 Fortran 95 派生类型的指针

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，请参见第 13 章。

当前过程和文件

调试会话期间，dbx 会定义过程和源文件为当前过程和源文件。设置断点及打印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

如果程序的任何标识符中有大写字母，dbx 便会识别出它们。如同某些早期版本一样，无需提供区分大小写或不区分大小写的命令。

Fortran 95 和 dbx 必须都处于区分大小写或不区分大小写模式下：

- 在不区分大小写模式下不使用 -U 选项进行编译和调试。dbx input_case_sensitive 环境变量的默认值便为 false。

如果源码中有名为 LAST 的变量，则在 dbx 中，print LAST 或 print last 命令都有效。Fortran 95 和 dbx 会依照请求将 LAST 和 last 视为相同的命令。

- 在区分大小写模式下使用 -U 编译和调试。dbx input_case_sensitive 环境变量的默认值便为 true。

如果源码中有一个名为 LAST 和一个名为 last 的变量，则在 dbx 中，print last 有效，但 print LAST 无效。Fortran 95 和 dbx 会依照请求区分 LAST 和 last。

注 - 在 dbx 中，文件和目录名始终区分大小写，即便将 dbx input_case_sensitive 环境变量设置为 false，也是如此。

dbx 会话示例

下例使用称作 my_program 的示例程序。

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
```

1. 使用 -g 选项编译和链接。

可以通过一或两个步骤来完成此操作。

使用 -g 一步完成编译和链接:

```
demo% f95 -o my_program -g a1.f a2.f a3.f
```

或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. 在名为 my_program 的可执行文件中启动 dbx。

```
demo% dbx my_program
Reading symbolic information...
```

- 键入 `stop in subnam` 来设置简单断点，其中 *subnam* 是子例程、函数或块数据子程序的名称。

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 MAIN 必须全部为大写字母，但 *subnam* 大小写均可。

- 键入 `run` 命令，它会运行启动 dbx 时命名的可执行文件中的程序。

```
(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，以指出程序停止的位置（本例中为文件 `a1.f` 的第 3 行）。

- 要打印值，请键入 `print` 命令。

打印 `n` 的值：

```
(dbx) print n
n = 2
```

打印矩阵 `twobytwo`，格式可能会不同：

```
(dbx) print twobytwo
twobytwo =
      (1,1)      -1.0
      (2,1)      -1.0
      (1,2)      -1.0
      (2,2)      -1.0
```

打印矩阵 `array`：

```
(dbx) print array
dbx: "array" is not defined in the current scope
(dbx)
```

打印失败的原因：`array` 未在此处定义，只是在 `mkidentity` 中进行了定义。

6. 要将执行前进到下一行，请键入 `next` 命令。

将执行前进到下一行：

```
(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停止于下一行。它将各次子程序调用按独立的语句来计数。

比较 `next` 命令和 `step` 命令。`step` 命令会执行下一源代码行或进入子程序的下一步。如果下一个可执行源码语句是一个子例程或函数调用，则：

- `step` 命令会在子程序的第一个源码语句处设置断点。
- `next` 命令将在调用后的第一个源码语句处，但仍在调用程序内的地方设置断点。

7. 要退出 `dbx`，请键入 `quit` 命令。

```
(dbx) quit
demo%
```

调试段故障

如果程序出现段故障 (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：

- 有超出声明范围的数组索引。
- 数组索引的名称拼写错误。
- 调用例程有一个 `REAL` 参数，而被调用例程的参数却为 `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 来找到故障

使用 dbx 来找到发生段故障的源代码行。

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%
```

使用 dbx 来找到发生 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

有许多原因可导致程序异常。定位故障的一种方法是在源程序中找到发生异常的行号，然后在该处寻找线索。

使用 `-ftrap=common` 编译将对所有常见异常进行强制捕获。

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

有时程序会因核心转储而停止，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**栈跟踪**。

`where` 命令显示执行在程序流中停止的位置及执行到达此点的过程 — 被调用例程的**栈跟踪**。

`ShowTrace.f` 是专门用于使核心转储在调用序列中深入几级 — 显示栈跟踪的程序。

自执行停止处开始显示调用序列：

```
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)
quilt 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)
```

处理数组

dbx 会识别出数组并打印它们。

```
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 Fortran 中的数组分片的信息，请参见第 120 页的“Fortran 数组分片语法”。

Fortran 95 可分配数组

下例显示在 dbx 中如何处理分配的数组。

```
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
Unknown size at line 6
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)
Known size at line 9
(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
```

显示内在函数

dbx 会识别出 Fortran 内在函数（仅限 SPARC™ 平台和 x86 平台）。

要在 dbx 中显示内在函数，请键入：

```
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(1)
abs(i) = 2
(dbx)
```

显示复数表达式

dbx 也会识别出 Fortran 复数表达式。

要在 dbx 中显示复数表达式，请键入：

```
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，请键入：

```
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%
```

注 - 区间表达式仅受以下程序的支持：编译为在基于 SPARC 的平台上运行，使用 `-xarch={sse|sse2}` 编译为在 Solaris x86 SSE/SSE2 Pentium 4 兼容的平台上运行，或使用 `-xarch=amd64` 编译为在 x64 平台上运行。

显示逻辑操作符

dbx 可以定位 Fortran 逻辑操作符并打印它们。

要在 dbx 中显示逻辑操作符，请键入：

```
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 95 派生类型

可以使用 dbx 显示结构 — Fortran 95 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
      )

```

指向 Fortran 95 派生类型的指针

可以使用 dbx 显示结构 — Fortran 95 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
      2 Declare a derived type.
      3 TYPE product
      4     INTEGER id
      5     CHARACTER*16 name
      6     CHARACTER*8 model
      7     REAL cost
      8     REAL price
      9     END TYPE product
     10
     11 Declare prod1 and prod2 targets.
     12 TYPE(product), TARGET :: prod1, prod2
     13
     14 Declare curr and prior pointers.
     15 TYPE(product), POINTER :: curr, prior
     16
     17 Make curr point to prod2.
     18 curr => prod2
     19
     20 Make prior point to prod1.
     21 prior => prod1
     22
     23 Initialize prior.
     24 prior%id = 82
     25 prior%name = "Coffee Cup"
     26 prior%model = "XL"
     27 prior%cost = 24.0
     28 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
)

```

上例中，dbx 显示了派生类型的所有字段，包括字段名。

可以使用结构 — 查询某 Fortran 95 派生类型的某项。

```

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

```

要打印指针，请键入：

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
)
```

第17章

使用 dbx 调试 Java 应用程序

本章说明如何使用 dbx 来调试由 Java™ 代码和 C JNI（Java™ 本地接口）代码或 C++ JNI 代码混和编写的应用程序。

本章由以下部分组成：

- 使用 dbx 调试 Java 代码
- Java 调试的环境变量
- 开始调试 Java 应用程序
- 自定义 JVM 软件的启动
- 调试 Java 代码的 dbx 模式
- 在 Java 模式下使用 dbx 命令

使用 dbx 调试 Java 代码

可以使用 Sun Studio dbx 调试运行在 Solaris™ OS 和 Linux OS 下的混合代码（Java 代码和 C 代码或 C++ 代码）。

使用 dbx 调试 Java 代码的功能

使用 dbx 可以调试几种类型的 Java 应用程序（请参见第 216 页的“开始调试 Java 应用程序”）。在调试本机代码和 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

调试 Java 代码时，dbx 有以下限制：

- dbx 无法像对待本机代码那样通过内核文件来指示 Java 应用程序的状态。
- 如果 Java 应用程序由于某种原因被挂起，dbx 无法指示该应用程序的状态，且 dbx 也不能进行过程调用。
- 修复并继续和运行时检查功能不适用于 Java 应用程序。

Java 调试的环境变量

以下环境变量专门用于用 dbx 来调试 Java 应用程序。可以在启动 dbx 前在 shell 提示符处设置 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变化。可以使用 jon 命令（第 320 页的“jon 命令”）和 joff 命令更改其设置（请参见第 319 页的“joff 命令”）。

jdbx_mode	jdbx_mode 环境变量可有以下设置：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，请参见第 225 页的“调试 Java 代码的 dbx 模式”。默认值：java。
JAVASRCPATH	可以使用 JAVASRCPATH 环境变量指定 dbx 查找 Java 源文件的目录。Java 源文件与 .class 或 .jar 文件不在同一目录中时，此变量很有用。有关详细信息，请参见第 220 页的“指定 Java 源文件的位置”。
CLASSPATHX	利用 CLASSPATHX 环境变量可以给 dbx 指定自定义类加载器装入的 Java 类文件的路径。有关详细信息，请参见第 220 页的“为使用自定义类加载器的类文件指定路径”。
jvm_invocation	利用 jvm_invocation 环境变量可以自定义 JVM™ 软件启动的方式。（术语“Java 虚拟机”和“JVM”的意思是 Java™ 平台的虚拟机。）有关详细信息，请参见第 221 页的“自定义 JVM 软件的启动”。

开始调试 Java 应用程序

可以使用 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 的正在运行的 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
在所有上述情况下，`dbx` 均可识别其正在调试的是 Java 应用程序。

调试类文件

可以如下例中所示的那样使用 `dbx` 调试文件扩展名为 `.class` 的文件。

```
(dbx) debug myclass.class
```

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
```

也可以使用类文件的完整路径名。`dbx` 会在 `.class` 文件中搜索来自动确定类路径的包部分，然后将完整路径名的其余部分添加到类路径中。例如，假定有以下路径名，`dbx` 会确定 `pkg/Toy.class` 是主类名，然后将 `/home/user/java` 添加到类路径中。

```
(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。可以使用 `dbx` 按下例所示调试 JAR 文件。

```
(dbx) debug myjar.jar
```

开始调试文件名以 `.jar` 结尾的文件时，`dbx` 会使用在此 JAR 文件的标明中指定的 `Main Class` 属性来确定主类。(主类是 JAR 文件内作为应用程序入口点的类。如果使用完整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。

如果调试无 Main-Class 属性的 JAR 文件，可以使用在 Java™ 2 平台标准版的 JarURLConnection 类中指定的 JAR URL 语法 `jar:<url>!/{entry}` 来指定主类的名称，如下例所示。

```
(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，dbx 都会执行以下操作：

- 将 ! 字符后指定的类路径作为主类（例如，`/myclass.class` 或 `/x/y/z.class`）来处理
- 将 JAR 文件名 `./myjar.jar`、`/a/b/c/d/e.jar` 或 `/a/b/c/d.jar` 添加到类路径中
- 开始调试主类

注 – 如果使用 `jvm_invocation` 环境变量（请参见第 221 页的“自定义 JVM 软件的启动”）指定 JVM 软件的自定义启动，则不会将 JAR 文件的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 文件的文件名添加到类路径中。

调试有包装器的 Java 应用程序

Java 应用程序通常有用于设置环境变量的包装器。如果 Java 应用程序有包装器，需要设置 `jvm_invocation` 环境变量来告知 dbx 正在使用包装器脚本（请参见第 221 页的“自定义 JVM 软件的启动”）。

将 dbx 连接到正在运行的 Java 应用程序

如果启动应用程序时按下例所示指定了各选项，便可将 dbx 连接到正在运行的 Java 应用程序。启动应用程序后，便可以一起使用正在运行的 Java 进程的进程 ID 和 dbx 命令（请参见第 297 页的“dbx 命令”）来开始调试。

```
$ java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrunjdbi
myclass.class
$ dbx - 2345
```

为了使 JVM 软件能够找到 `libjdbi_agent.so`，需要在运行 Java 应用程序之前将相应路径添加到 `LD_LIBRARY_PATH`：

- 如果在运行 Solaris OS 的系统上使用 JVM 软件的 32 位版本，请将 `/installation_directory/prod/lib/dbxruntime` 添加到 `LD_LIBRARY_PATH`。
- 如果在运行 Solaris OS、基于 SPARC 的系统上使用 JVM 软件的 64 位版本，请将 `/installation_directory/prod/lib/v9/dbxruntime` 添加到 `LD_LIBRARY_PATH`。
- 如果在运行 Linux OS、基于 x64 的系统上使用 JVM 软件的 64 位版本，请将 `/installation_directory/prod/lib/amd64/dbxruntime` 添加到 `LD_LIBRARY_PATH`。

`installation_directory` 是 Sun Studio 软件的安装位置。

将 dbx 连接到正在运行的应用程序时，dbx 会在 Java 模式下开始调试应用程序。

如果 Java 应用程序要求 64 位目标库，请在启动应用程序时加入 `-d64` 选项。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 -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent -d64
myclass.class
$ dbx - 2345
```

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++ 应用程序必须通过为 JVM 软件指定以下选项，才能启动 Java 应用程序：

```
-Xdebug -Xnoagent -Xrundbx_agent
```

为了使 JVM 软件能够找到 `libdbx_agent.so`，需要在运行 Java 应用程序之前将相应路径添加到 `LD_LIBRARY_PATH`：

- 如果在运行 Solaris OS 的系统上使用 JVM 软件的 32 位版本，请将 `/installation_directory/prod/lib/dbxruntime` 添加到 `LD_LIBRARY_PATH`。
- 如果在运行 Solaris OS、基于 SPARC 的系统上使用 JVM 软件的 64 位版本，请将 `/installation_directory/prod/lib/v9/dbxruntime` 添加到 `LD_LIBRARY_PATH`。
- 如果在运行 Linux OS、基于 x64 的系统上使用 JVM 软件的 64 位版本，请将 `/installation_directory/prod/lib/amd64/dbxruntime` 添加到 `LD_LIBRARY_PATH`。

`installation_directory` 是 Sun 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，所提供的参数会被传递给应用程序，而非 JVM 软件。要将参数传递给 JVM 软件，请参见第 221 页的“自定义 JVM 软件的启动”。

指定 Java 源文件的位置

Java 源文件有时与 `.class` 或 `.jar` 文件不在同一目录中。可以使用 `$JAVASRCPATH` 环境变量指定 `dbx` 查找 Java 源文件的目录。例如，`JAVASRCPATH=./mydir/mysrc:/mydir/mylibsrc:/mydir/myutils` 会使 `dbx` 在与正被调试的类文件对应的源文件的列出目录中查找。

指定 C 源文件或 C++ 源文件的位置

在以下情况下，`dbx` 可能无法找到 C 源文件或 C++ 源文件：

- 如果编译时源文件不在原来所在的位置
- 如果编译源文件的系统与运行 `dbx` 的系统不同，编译目录将不会有相同的路径名

在此类情况下，请使用 `pathmap` 命令（请参见第 335 页的“`pathmap` 命令”）将一个路径名映射到另一个路径名，以便 `dbx` 能够找到文件。

为使用自定义类加载器的类文件指定路径

应用程序可以有从可能不是常规类路径一部分的位置装入类文件的自定义类加载器。在此类情况下，`dbx` 无法找到类文件。利用 `CLASSPATHX` 环境变量可以给 `dbx` 指定类文件的自定义类加载器装入的类文件的路径。例如，`CLASSPATHX=./myloader/myclass:/mydir/mycustom` 会使 `dbx` 在尝试查找类文件时到列出的目录中查找。

在 JVM 软件尚未装入的代码上设置断点

要在 JVM 软件尚未装入的类文件中的 Java 方法上设置停止断点，请使用带 `stop in` 命令的类的全名，或带 `stop inmethod` 命令的类名。请参见下例。

```
(dbx) stop in Java.Pkg.Toy.myclass.class.mymethod
(dbx) stop inmethod myclass.class.mymethod
```

要在 JVM 软件尚未装入的共享库中的 C 或 C++ 函数上设置停止断点，请在设置断点前预装共享库的符号表。例如，如果有名为 `mylibrary.so` 的库，其中包含名为 `myfunc` 的函数，便可按以下方式预装库并在函数上设置断点：

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

还可以在开始使用 `dbx` 调试前运行一次应用程序，以装入所有动态装入的共享目标的符号表。

自定义 JVM 软件的启动

可能需要在 `dbx` 中自定义 JVM 软件的启动，以完成以下工作：

- 指定 JVM 软件的路径名（请参见第 222 页的“指定 JVM 软件的路径名”）
- 将某些运行参数传递给 JVM 软件（请参见第 222 页的“将运行参数传递给 JVM 软件”）
- 指定自定义包装器取代默认 Java 包装器来运行 Java 应用程序（请参见第 222 页的“指定 Java 应用程序的自定义包装器”）
- 指定 64 位 JVM 软件（请参见第 224 页的“指定 64 位 JVM 软件”）

可以使用 `jvm_invocation` 环境变量自定义 JVM 软件的启动。默认情况下，`jvm_invocation` 环境变量未定义时，`dbx` 将按以下方式启动 JVM 软件：

```
java -Xdebug -Xnoagent -Xrun:dbx_agent:syncpid
```

定义 `jvm_invocation` 环境变量时，`dbx` 会使用变量的值来启动 JVM 软件。

必须在 `jvm_invocation` 环境变量的定义中包括 `-Xdebug` 选项。`dbx` 会将 `-Xdebug` 扩展到内部选项 `-Xdebug -Xnoagent -Xrun:dbx_agent::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

默认情况下，如果不指定 JVM 软件的路径名，则 dbx 会启动路径中的 JVM 软件。

要指定 JVM 软件的路径名，请将 `jvm_invocation` 环境变量设置为适当的路径名，如下例所示。

```
jvm_invocation="/myjava/java -Xdebug"
```

此设置将使 dbx 按以下方式启动 JVM 软件：

```
/myjava/java -Djava.compiler=NONE -Xdebug -Xnoagent -  
Xrundbx_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，请设置 `jvm_invocation` 环境变量来带这些参数启动 JVM 软件，如下例所示。

```
jvm_invocation="java -Xdebug -Xms512 -Xmx1024 -Xcheck:jni"
```

这将使 dbx 按以下方式启动 JVM 软件：

```
java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent:sync=  
-Xms512 -Xmx1024 -Xcheck:jni
```

指定 Java 应用程序的自定义包装器

Java 应用程序可以使用自定义包装器来启动。如果应用程序使用自定义包装器，便可以使用 `jvm_invocation` 环境变量来指定要使用的包装器，如下例所示。

```
jvm_invocation="/export/siva-a/forte4j/bin/forte4j.sh -J-Xdebug"
```

这将使 dbx 按以下方式启动 JVM 软件：

```
/export/siva-a/forte4j/bin/forte4j.sh - -J-Xdebug -J-Xnoagent -J-  
Xrundbxagent:sync=process_id
```

使用接受命令行选项的自定义包装器

以下包装器脚本 (xyz) 设置了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 and 用户应用程序的某些命令行选项。对于此形式的包装器脚本, 可设置 `jvm_invocation` 环境变量并按以下方式启动 `dbx`:

```
% jvm_invocation="xyz -J-Xdebug -Jany other java options"
% dbx myclass.class -Dide=visual
```

使用不接受命令行选项的自定义包装器

以下包装器脚本 (xyz) 会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` 按以下两种方式之一, 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中，并启动 `dbx`，可以将脚本修改为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。

- 可以略微修改脚本来接受某些命令行选项，如下所示：

```
#!/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 `jvm_invocation` 环境变量，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

如果希望 `dbx` 启动 64 位 JVM 软件来调试需要 64 位目标库的应用程序，请在设置 `jvm_invocation` 环境变量时加入 `-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

dbx 处于 Java 模式或 JNI（Java 本地接口）模式下时，可以检查 Java 应用程序（包括 JNI 代码）的状态，并控制代码的执行。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机代码，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 命令只会影响本地程序，所有与 Java 有关的功能都会被禁用。此模式用于调试与 Java 无关的程序。

执行 Java 应用程序过程中，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` 命令显式地从 Java 或 JNI 模式切换到本地模式，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，使用 `control-C`），dbx 会尝试将应用程序置于安全状态并挂起所有线程来将模式自动设置为 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。然后便可使用 `jon` 命令切换到 Java 模式来检查程序的状态。

在 Java 模式下使用 dbx 命令

使用 dbx 调试混合 Java 代码和本机代码时，dbx 命令分为以下几类：

- 接受相同参数，且在 Java 模式或 JNI 模式下的运行方式与在本地模式下相同的命令（请参见第 227 页的“在 Java 模式和本地模式下具有完全相同语法和功能的命令”）。
- 有只在 Java 模式或 JNI 模式下有效的参数，也有只在本地模式下有效的参数的命令（请参见第 228 页的“在 Java 模式下有不同语法的命令”）。
- 只在 Java 模式或 JNI 模式下有效的命令（请参见第 229 页的“只在 Java 模式下有效的命令”）。

未包括在任一以上类别中的命令只在本地模式下有效。

dbx 命令中的 Java 表达式求值

在大多数 dbx 命令中使用的 Java 表达式算子支持以下构造：

- 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 操作符
- 数组长度操作符

Java 表达式算子不支持以下构造：

- 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

特别有用的一种检查 Java 应用程序状态的方式是在 dbx 调试器中使用显示功能。

不建议依靠表达式中作用不限于检查数据的精确值语义。

dbx 命令使用的静态和动态信息

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，Java 应用程序执行完成后这些信息将无法使用。不过，使用 dbx 调试 Java 应用程序时，dbx 会在启动 JVM 软件前从作为系统类路径一部分的类文件和 JAR 文件中收集其需要的一部分信息。这样 dbx 便可在您运行应用程序前更好地完成对断点的错误检查。

某些 Java 类及其属性可能无法通过类路径来访问。dbx 可以检查并单步执行这些类，这些类被装入后，表达式分析器便可访问它们。但它收集的信息是临时性的，JVM 软件终止后便无法再使用。

dbx 调试 Java 应用程序所需的某些信息在任何地方均无记录，在这种情况下，dbx 会在调试代码期间浏览 Java 源文件来取得该信息。

在 Java 模式和本地模式下具有完全相同语法和功能的命令

以下 dbx 命令在 Java 模式和本地模式下具有相同的语法，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 dbx 环境变量
delete	删除断点和其他事件
down	将调用栈下移（远离 main）
dump	打印过程或方法的所有局部变量
文件	列出或更改当前文件
frame	列出或更改当前栈帧号
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	列出或更改当前行号
next	单步执行一个源代码行（步过调用）
pathmap	将一个路径名映射至另一个路径名，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

命令	功能
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	单步执行一个机器指令（步入调用）
up	将调用栈上移（靠近 main）
wherami	显示当前源代码行

在 Java 模式下有不同语法的命令

以下 dbx 命令在进行 Java 调试和本机代码调试时使用的语法不同，在 Java 模式和本地模式下的运行方式也不同。

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
detach	将目标进程从 dbx 的控制下释放出来	将目标进程从 dbx 的控制下释放出来
显示	在每个停止点对表达式求值并打印	在每个停止点对表达式、局部变量或参数求值并打印
文件	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）
打印	打印表达式的值	打印表达式、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	设置源码级断点	设置源码级断点

命令	本地模式功能	Java 模式功能
线程	列出或更改当前线程	列出或更改当前线程
线程	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	打印表达式类型或类型声明	打印标识符声明
when	指定事件发生时执行命令	指定事件发生时执行命令
其中	打印调用栈	打印调用栈

只在 Java 模式下有效的命令

以下 dbx 命令只在 Java 模式或 JNI 模式下有效。

命令	功能
java	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
javaclasses	发出该命令时打印 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
jpgks	发出该命令时打印 dbx 已知的所有 Java 包的名称
native	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18章

在机器指令级调试

本章介绍如何在机器指令级使用事件管理和进程控制命令，如何显示指定地址处的内存内容以及如何将源代码行连同其对应的机器指令一并显示。`next`、`step`、`stop` 和 `trace` 命令中的每一个命令都支持一个机器指令级变量：`nexti`、`stepi`、`stopi` 和 `tracei`。使用 `regs` 命令输出机器寄存器中的内容，或使用 `print` 命令分别输出单个寄存器中的内容。

本章由以下部分组成：

- 检查内存的内容
- 在机器指令级单步执行和跟踪
- 在机器指令级设置断点
- 使用 `adb` 命令
- 使用 `regs` 命令

检查内存的内容

可使用 `addresses` 和 `examine` 或 `x` 命令检查内存位置的内容及输出每个地址的汇编语言指令。可使用派生自 `adb(1)` 的命令，即汇编语言调试器来查询：

- `address`，使用 =（等号）字符，或
- 保存在地址处的 `contents`，使用 /（斜线）字符。

可使用 `dis` 和 `listi` 命令打印汇编命令。（请参见第 234 页的“使用 `dis` 命令”和第 235 页的“使用 `listi` 命令”。）

使用 `examine` 或 `x` 命令

使用 `examine` 命令或其别名 `x` 显示内存内容或地址。

使用以下语法以 `format` 格式显示始于 `count` 项的 `address` 的内存内容。默认的 `address` 为先前显示的最后一个地址后的下一个地址。默认 `count` 为 1。默认 `format` 和在先前的 `examine` 命令中使用的相同；如果是给定的第一个命令，则为 `x`。

examine 命令的语法为：

```
examine [address] [/ [count] [format]]
```

要按 *format* 格式显示 *address1* 到 *address2*（首末地址包含在内）的内存内容，请键入：

```
examine address1, address2 [/ [format]]
```

要显示地址，而不是给定格式的地址内容，请键入：

```
examine address = [format]
```

要打印保存在 examine 最后显示的地址后的下一个地址处的值，请键入：

```
examine +/ i
```

要打印表达式的值，请以地址形式输入表达式：

```
examine address=format  
examine address=
```

地址

address 是可产生地址或可用作地址的任何表达式。可用 +（加号）替换 *address*，它以默认格式显示下一地址的内容。

例如，以下为有效地址：

0xff99	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通过在名称前加和号 (&) 来指定。使用函数名称时可以不使用和符号；&main 等同于 main。通过在名称前加美元符号 (\$) 来表示寄存器。

格式

format 是 `dbx` 显示查询结果所采用的地址显示格式。产生的输出取决于当前显示 *format*。要更改显示格式，请提供另一 *format* 代码。

被设置在每个 `dbx` 会话起始处的默认格式为 `x`，它以十六进制的 32 位字形式显示地址或值。以下内存显示格式为合法格式。

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）。（默认格式）
b	显示为八进制字节。
c	显示为字符。
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。
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）。

Count

count 为十进制的重复计数。增量大小取决于内存显示格式。

使用地址的示例

以下示例显示如何使用带有 *count* 和 *format* 选项的地址来显示始于当前停止点的五个连续的反汇编指令。

对于基于 SPARC 的系统:

```
(dbx) stepi
停止于 main 中的 0x108bc 处
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 [未解析的 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)
```

使用 dis 命令

dis 命令等同于以 i 作为默认显示格式的 examine 命令。

以下是 dis 命令的语法。

```
dis [address] [address1, address2] [/count]
```

dis 命令:

- 不使用参数时显示以 + 开始的 10 个指令。
- 只使用 *address* 参数时, 反汇编 10 个始于 *address* 的指令。
- 使用 *address* 参数和 *count* 时, 反汇编始于 *address* 的 *count* 指令。
- 使用 *address1* 和 *address2* 参数时, 反汇编 *address1* 到 *address2* 的指令。
- 只使用 *count* 时, 显示始于 + 的 *count* 指令。

使用 listi 命令

要将源代码行连同其对应的汇编指令一并显示，请使用 listi 命令，此命令等同于命令 list -i。请参见第 71 页的“打印源码列表”中对 list -i 的论述。

对于基于 SPARC 的系统：

```
(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]
```

对于基于 x86 的系统：

```
(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)
```

在机器指令级单步执行和跟踪

机器指令级命令与其源码级副本的功能相同，只不过它们在单步指令级，而非源代码行级执行。

在机器指令级单步执行

要从一个机器指令单步执行到下一机器指令，请使用 `nexti` 命令或 `stepi` 命令

`nexti` 命令和 `stepi` 命令与其源代码级副本的功能相同：`nexti` 命令步出函数，`stepi` 命令步入由下一指令（停止于被调用函数中的第一个指令）调用的函数。命令形式也相同。请参见第 333 页的“`next` 命令”和第 350 页的“`step` 命令”中的说明。

`nexti` 命令和 `stepi` 命令与对应的源码级命令的输出在以下两个方面不同：

- 输出包含程序停止在的指令地址（而非源代码行号）。
- 默认输出包含反汇编指令，而非源代码行。

例如：

```
(dbx) func
hand::ungrasp
(dbx) nexti
ungrasp +0x18: call support
(dbx)
```

有关详细信息，请参见第 334 页的“`nexti` 命令”和第 352 页的“`stepi` 命令”。

在机器指令级跟踪

除使用 `tracei` 命令外，机器指令级与源代码级的跟踪技术的功能相同。对于 `tracei` 命令，`dbx` 在执行每一地址检查或跟踪每一变量的值后才会执行一条指令。`tracei` 命令会产生自动的、与 `stepi` 类似的行为：程序一次前进一个指令来步入函数调用。

使用 `tracei` 命令时，执行每一指令后，它会使程序停止片刻，让 `dbx` 检查地址执行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。

有关跟踪及其事件说明和修饰符的详细信息，请参见第 104 页的“跟踪执行”和第 368 页的“`tracei` 命令”。

以下是 `tracei` 的一般语法:

```
tracei event-specification [modifier]
```

`tracei` 的常用形式为:

```
tracei step           跟踪每一指令。
tracei next           跟踪每一指令, 但跳过调用。
tracei at address   跟踪给定代码地址。
```

有关详细信息, 请参见第 368 页的“`tracei` 命令”。

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
....
....
```

在机器指令级设置断点

要在机器指令级设置断点，请使用 `stopi` 命令。该命令接受任何 *event specification*，使用以下语法：

```
stopi event-specification [modifier]
```

`stopi` 命令的常用形式为：

```
stopi [at address] [-if cond]  
stopi in function [-if cond]
```

有关详细信息，请参见第 358 页的“`stopi` 命令”。

在地址处设置断点

要在特定地址设置断点，请键入：

```
(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)
```

使用 adb 命令

利用 adb 命令可按 adb(1) 语法输入命令。也可输入将每条命令解释为 adb 语法的 adb 模式。支持大多数 adb 命令。

有关详细信息，请参见第 281 页的“adb 命令”。

使用 regs 命令

利用 regs 命令可打印所有寄存器的值。

以下是 regs 命令的语法：

```
regs [-f] [-F]
```

-f 包括浮点寄存器（单精度）。-F 包括浮点寄存器（双精度）。

有关详细信息，请参见第 341 页的“regs 命令”。

对于基于 SPARC 的系统：

```
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 0xffff420
o4-o7      0xef752f80 0x00000003 0xffff3d8 0x000109b8
l0-l3      0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7      0xffff438 0x00000001 0x00000007 0xef74df54
i0-i3      0x00000001 0xffff4a4 0xffff4ac 0x00020c00
i4-i7      0x00000001 0x00000000 0xffff440 0x000108c4
y          0x00000000
psr        0x40400086
pc         0x000109c0:main+0x4      mov      0x5, %l0
npc        0x000109c4:main+0x8      st       %l0, [%fp - 0x8]
f0f1      +0.0000000000000000e+00
f2f3      +0.0000000000000000e+00
f4f5      +0.0000000000000000e+00
f6f7      +0.0000000000000000e+00
...
```

对于基于 x64 的系统:

```
(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      0xfffffd7ffdfdf820
rbx      0xfffffd7fff3fb190
rdx      0x0000000000401b50
rcx      0x0000000000401b54
rax      0x0000000000416cf0
trapno   0x0000000000000003
err      0x0000000000000000
rip      0x0000000000401709:main+0xf9movl
$0x0000000000000000,0xffffffffffffffffc(%rbp)
cs       0x000000000000004b
eflags   0x0000000000000206
rsp      0xfffffd7ffdfdf7b0
ss       0x0000000000000043
fs       0x00000000000001bb
gs       0x0000000000000000
es       0x0000000000000000
ds       0x0000000000000000
fsbase   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      0xfffffd7ffdfdf820
rbx      0xfffffd7fff3fb190
rdx      0x0000000000401b50
```



```

rcx      0x0000000000401b54
rax      0x0000000000416cf0
trapno   0x0000000000000003
err      0x0000000000000000
rip      0x0000000000401709:main+0xf9movl
$0x0000000000000000,0xffffffffffffffffc(%rbp)
cs       0x000000000000004b
eflags   0x0000000000000206
rsp      0xffffffff7fffdf7b0
ss       0x0000000000000043
fs       0x00000000000001bb
gs       0x0000000000000000
es       0x0000000000000000
ds       0x0000000000000000
fsbase   0xffffffff7fff3a200
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 0xffff8000 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

以下各表列出了可在表达式中使用的 SPARC 体系结构、x86 体系结构和 AMD64 体系结构的平台特定寄存器名称。

SPARC 寄存器信息

以下是 SPARC 体系结构的寄存器信息。

寄存器	描述
\$g0 到 \$g7	全局寄存器
\$o0 到 \$o7	“外”寄存器
\$l0 到 \$l7	“局部”寄存器
\$i0 到 \$i7	“内部”寄存器
\$fp	帧指针，等同于寄存器 \$i6
\$sp	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 到 \$f31	FPU “f” 寄存器
\$fsr	FPU 状态寄存器
\$fq	FPU 队列

\$f0f1 \$f2f3 ...\$f30f31 对浮点型寄存器被视为具有 C 的 “double” 类型（通常 \$fN 寄存器被视为 C 的 “float” 类型）。也可将这些对称作 \$d0 ...\$d30。

SPARC V9 和 V8+ 硬件上提供了以下这些额外寄存器：

```
$g0g1 through $g6g7
$o0o1 through $o6o7
$xfsr $tstate $gsr
$f32f33 $f34f35 through $f62f63 ($d32 ... $d62)
```

有关 SPARC 寄存器和寻址的详细信息，请参见《SPARC 体系结构参考手册》和《SPARC 汇编语言参考手册》。

x86 寄存器信息

以下是 x86 体系结构的寄存器信息。

寄存器	描述
\$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	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
\$eflags	标志
\$uesp	用户栈指针
\$ss	堆栈段寄存器

常用寄存器也使用其机器无关名称作为别名。

寄存器	描述
\$SP	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

80386 下半部（16 位）寄存器为：

寄存器	描述
\$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：

寄存器	描述
\$al	寄存器下（右）半部 \$ax
\$ah	寄存器上（左）半部 \$ax
\$cl	寄存器下（右）半部 \$cx
\$ch	寄存器上（左）半部 \$cx
\$dl	寄存器下（右）半部 \$dx
\$dh	寄存器上（左）半部 \$dx
\$bl	寄存器下（右）半部 \$bx
\$bh	寄存器上（左）半部 \$bx

80387 的寄存器为：

寄存器	描述
\$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	代码段选择符
\$fopoff	操作数指针偏移
\$fopsel	操作数指针选择符
\$st0 到 \$st7	数据寄存器

AMD64 寄存器信息

以下是 AMD64 体系结构的寄存器信息：

寄存器	描述
rax	通用寄存器 - 为函数调用传递参数
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
rbp	通用寄存器 - 栈管理 / 帧指针
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 - 栈管理 / 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
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

寄存器	描述
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
os	段寄存器
es	段寄存器
fs	段寄存器
gs	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
ftw	fxsave 和 fxstor 内存映像标记字
fop	fxsave 和 fxstor 内存映像最后 x87 op 代码
frfp	代码段中的 fxsave 和 fxstor 内存映像 64 位偏移
frdp	日期段中的 fxsave 和 fxstor 内存映像 64 位偏移
mxcsr	fxsave 和 fxstor 内存映像 128 媒体指令控制和状态寄存器
mxcsr_mask	mxcsr_mask 中的设置位指示 mxcsr 中支持的功能位

第19章

将 dbx 与 Korn Shell 配合使用

dbx 命令语言基于 Korn Shell (ksh 88) 语法，包括 I/O 重定向、循环、内置运算、历史及命令行编辑。本章列出了 ksh-88 与 dbx 命令语言间的区别。

如果启动时未找到 dbx 初始化文件，dbx 会采用 ksh 模式。

本章由以下部分组成：

- 未实现的 ksh-88 功能
- ksh-88 的扩展
- 重命名命令

未实现的 ksh-88 功能

以下 ksh-88 功能尚未在 dbx 中实现：

- 用于给数组 *name* 赋值的 `set -A name`
- `set -o` 特定选项: `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 (``...``) (使用 `$(...)` 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 (与程序通信的后台运行的命令或管线)

ksh-88 的扩展

dbx 添加以下功能作为扩展：

- `$(p > flags]` 语言表达式
- `typeset -q` 为用户定义函数启用特殊引用
- `csh` 类 `history` 和 `alias` 参数
- `set +o path` 禁用路径搜索
- 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`（相反者为：`-r, raw`）
- 内置 `dbx` 命令

重命名命令

某些 `dbx` 命令被重命名，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 `dbx kill` 命令合并。
- `alias` 命令是 `ksh alias`，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

利用 `bind` 可再绑定编辑函数。可以使用该命令来显示或修改 EMacs 风格编辑器和 vi 风格编辑器的键绑定。`bind` 命令的语法为：

<code>bind</code>	显示当前编辑键绑定
<code>bind key=definition</code>	将 <i>key</i> 绑定到 <i>definition</i>
<code>bind key</code>	显示 <i>key</i> 的当前定义
<code>bind key=</code>	删除 <i>key</i> 绑定
<code>bind -m key=definition</code>	将 <i>key</i> 定义为使用 <i>definition</i> 的宏
<code>bind -m</code>	与 <code>bind</code> 相同

其中：

key 为键名。

definition 为将绑定到键上的宏的定义。

以下是 EMacs 风格编辑器的其中一些更重要的默认键绑定：

<code>^A</code> = 行开始	<code>^B</code> = 后一个字符
<code>^D</code> = 磁带结束符或删除	<code>^E</code> = 行结束
<code>^F</code> = 前一个字符	<code>^G</code> = 终止
<code>^K</code> = 删除到行末	<code>^L</code> = 刷新
<code>^N</code> = 下一个历史命令	<code>^P</code> = 上一个历史命令
<code>^R</code> = 搜索历史命令	<code>^^</code> = 引号
<code>^?</code> = 向后删除字符	<code>^H</code> = 向后删除字符
<code>^[b</code> = 后退一个字	<code>^[d</code> = 向前删除字
<code>^[f</code> = 向前一个字	<code>^[^H</code> = 向后删除字
<code>^[^</code> = 完成	<code>^[?</code> = 列出命令

以下是 vi 风格编辑器的其中一些更重要的默认键绑定：

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 = 取消操作	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，以下按键是特殊的：

^? = 删除字符	^H = 删除字符
^U = 删除行	^W = 删除字

第20章

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。

本章由以下部分组成：

- 动态链接程序
- 修复并继续
- 在共享库中设置断点
- 在显式装入的库中设置断点

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**的共享对象，可包括 `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 `dldclose(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**来指称共享对象（`.so`）或可执行文件（`a.out`）。可以使用 `loadobject` 命令（请参见第 325 页的“`loadobject` 命令”）列出和管理装入对象中的符号信息。

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。链接映射保留在正在被调试程序的内存中，可间接通过 `librtld_db.so` 这一供调试器使用的特殊系统库来访问它。

启动序列和 .init 段

.init 段是属于装入共享对象时执行的共享对象的一段代码。例如，.init 由 C++ 运行时系统用于调用 .so 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。动态链接程序随后便会遍历该链接映射，并为每个共享对象执行 .init 段。syncrtld 事件（请参见第 270 页的“syncrtld”）在这两个阶段之间发生。

过程链接表

过程链接表 (PLT) 是 rtld 用于便利跨共享对象调用的结构。例如，对 printf 的调用便会通过这个间接表。可以在通用及处理器特定 SVR4 ABI 参考手册中找到对这一过程的详细说明。

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。

修复并继续

对使用 dlopen() 装入的共享对象使用修复并继续需要更改打开它们的方式，修复并继续才能正常工作。使用模式 RTLD_NOW|RTLD_GLOBAL 或 RTLD_LAZY|RTLD_GLOBAL。

在共享库中设置断点

要在共享库中设置断点，dbx 运行时需要知道程序将使用该库，dbx 需要为该库装入符号表。要确定新装入的程序运行时将使用哪些库，dbx 会执行程序至运行时链接程序装入所有启动库为止。dbx 随后会读取装入库的列表，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，还是在 dbx 提示符处使用 debug 命令，抑或是从 IDE 中的 dbx 调试器中装入，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。

如果共享对象使用 `dlclose()` 卸下，dbx 会记住其中设置的断点，如果使用 `dlopen()` 再次装入该共享对象，即便应用程序再次运行，也会替换它们。

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
```

现在便可在装入对象被 `dlopen()` 装入前在其中导航模块和函数及设置断点。程序装入装入对象后，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，dbx 在此初始化完成后才能在装入的库中设置断点。确切地讲，这意味着无法让 dbx 停止在 `dlopen()` 装入的库中的 `_init()` 处。

附录 A

修改程序状态

本附录重点说明 dbx 的使用方法及在 dbx 下运行程序时更改程序或更改程序运行状态的命令，并与不使用 dbx 运行程序进行了比较。重要的是了解哪些命令可能会修改程序。

本章由以下部分组成：

- 在 dbx 下运行程序的影响
- 更改程序状态的命令

在 dbx 下运行程序的影响

在 dbx 下运行时，应用程序的行为方式可能会不同。尽管 dbx 努力尽量减小对被调试程序的影响，但您还是应该注意以下这些问题：

- 可能忘记了去除 `-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 初始化脚本中可能有您忘记的曾设置过的一些环境变量。在 dbx 下运行时，栈基从另一地址处启动。它视环境及 `argv[]` 的内容的不同而异，从而强制局部变量以不同方式分配。如果它们未被初始化，将获得不同的随机编号。可以使用运行时检查来检测此问题。
- 程序不对使用之前用 `malloc()` 分配的内存进行初始化，与前者情况类似。可以使用运行时检查来检测此问题。
- dbx 必须捕获 LWP 创建和 `dlopen` 事件，这些事件可能会影响对计时敏感的多线程应用程序。
- dbx 会对信号进行上下文切换，因此，如果应用程序大量使用信号，情况可能会有所不同。

- 程序可能期望 `mmap()` 对映射段总是返回相同的基地址。在 `dbx` 下运行会严重影响地址空间，使 `mmap()` 无法像在不使用 `dbx` 运行程序时那样返回相同的地址。要确定这是否有问题，请查看 `mmap()` 的所有使用情况，确保返回的地址为程序所使用，而非硬编码地址。
- 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计划。在 `dbx` 下运行会扰乱线程计划，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` 运行是否会导致同样问题发生。

为尽可能减少 `dbx` 施加的影响，请尝试当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

`assign` 命令

`assign` 命令将表达式的值赋给变量。在 `dbx` 中使用它会永久性更改变量的值。

```
assign variable = expression
```

`pop` 命令

`pop` 命令会从栈中弹出一个或多个帧：

<code>pop</code>	弹出当前帧。
<code>pop number</code>	弹出 <code>number</code> 帧。
<code>pop -f number</code>	弹出帧直至指定帧号 <code>number</code> 。

继续执行时所有弹出的调用即会被重新执行，这可能会导致不希望发生的程序更改。`pop` 还会为弹出函数的局部对象调用析构函数。

有关详细信息，请参见第 336 页的“`pop` 命令”。

call 命令

在 dbx 中使用 call 命令时，如果调用过程，过程便会按指定方式执行：

```
call proc ([params])
```

过程可能会修改程序中的某些内容。dbx 会像您已将其写入到程序源码中那样进行调用。

有关详细信息，请参见第 284 页的“call 命令”。

print 命令

要打印表达式的值，请键入：

```
print expression, ...
```

如果表达式有函数调用，便应遵循使用 call 命令时相同的注意事项。对于 C++，还应注意由重载操作符导致的意外的副作用。

有关详细信息，请参见第 337 页的“print 命令”。

when 命令

when 命令的一般语法如下：

```
when event-specification [modifier] {command; ... }
```

事件发生时，其中的命令便会被执行。

到达某行或过程时，便会执行命令。程序的状态是否会被更改取决于发出的命令。

有关详细信息，请参见第 375 页的“when 命令”。

fix 命令

可以使用 `fix` 命令直接更改程序：

```
fix
```

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地链接到应用程序中。

确保查看修复并继续的限制。请参见第 10 章。

有关详细信息，请参见第 310 页的“`fix` 命令”。

cont at 命令

`cont at` 命令会更改程序运行的顺序。如果程序为多线程程序，则必须在行 `line.id` 继续执行。

```
cont at line [ id ]
```

这可能会更改程序的结果。

事件管理

事件管理是指被调试程序中发生事件时，dbx 执行操作的能力。发生事件时，可利用 dbx 停止进程、执行任意命令或打印信息。一个最简单的事件示例就是断点（请参见第 6 章）。其他事件的示例包括错误、信号、系统调用、对 `dlopen()` 的调用以及数据更改（请参见第 99 页的“设置数据更改断点”）。

本附录由以下部分组成：

- 事件处理程序
- 创建事件处理程序
- 操作事件处理程序
- 使用事件计数器
- 设置事件规范
- 事件规范修饰符
- 分析和二义性
- 使用预定义变量
- 设置事件处理程序示例

事件处理程序

事件管理建立在**处理程序**概念的基础上。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。（请参见第 261 页的“设置事件规范”。）事件规范指定将触发处理程序的事件。

事件发生并触发处理程序时，处理程序会根据事件规范中包括的修饰符来评估事件。（请参见第 271 页的“事件规范修饰符”。）如果事件满足修饰符施加的条件，便会执行处理程序的副作用动作（即处理程序“发生”）。

在特定行设置断点便是程序事件与 dbx 操作关联的一个示例。

创建处理程序的最普通形式是使用 `when` 命令来创建。

```
when event-specification {action; ... }
```

本章中的示例显示如何使用 `when` 语句编写命令（如 `stop`、`step` 或 `ignore`）。这些示例的目的在于说明 `when` 及其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按示例中所示。

创建事件处理程序

使用命令 `when`、`stop` 和 `trace` 来创建事件处理程序。（有关详细信息，请参见第 375 页的“`when` 命令”、第 353 页的“`stop` 命令”和第 365 页的“`trace` 命令”。）

`stop` 是常见的术语 `when` 的简称。

```
when event-specification { stop -update; whereami; }
```

事件规范由事件管理命令 `stop`、`when` 和 `trace` 用于指定感兴趣的事件。（请参见第 261 页的“设置事件规范”）。

大多数 `trace` 命令可以使用 `when` 命令、`ksh` 功能和事件变量来手动建立。需要固定格式的跟踪输出时，这种方法特别有用。

每个命令都会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` 来访问此编号。

操作事件处理程序

可以使用以下命令来操作事件处理程序。有关每个命令的详细信息，请参见各说明部分。

- `status` - 列出处理程序（请参见第 350 页的“`status` 命令”）。
- `delete` - 删除包括临时处理程序在内的所有处理程序（请参见第 302 页的“`delete` 命令”）。
- `clear` - 删除基于断点位置的处理程序（请参见第 288 页的“`clear` 命令”）。
- `handler -enable` - 启用处理程序（请参见第 315 页的“`handler` 命令”）。
- `handler -disable` - 禁用处理程序。
- `cancel` - 取消信号并使进程继续（请参见第 285 页的“`cancel` 命令”）。

使用事件计数器

每个事件处理程序均有一个行程计数器，它有计数限制。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。达到限制时，计数器会自动重置为 0。默认限制是 1。每当重新运行进程时，所有事件计数器都会被重置。

可以使用带 `stop`、`when` 或 `trace` 命令的 `-count` 修饰符来设置计数限制（请参见第 272 页的“`-count n -count infinity`”）。否则，请使用 `handler` 命令单独操作事件处理程序：

```
handler [ -count | -reset ] hid new-count new-count-limit
```

设置事件规范

`stop`、`stopi`、`when`、`wheni`、`trace` 和 `tracei` 命令使用事件规范来表示事件类型和参数。其格式由一个代表事件类型和可选参数的关键字组成。所有三个命令的事件规范的含义通常完全相同；例外情况在命令说明中有介绍（请参见第 353 页的“`stop` 命令”、第 365 页的“`trace` 命令”和第 375 页的“`when` 命令”）。

断点事件规范

断点是操作发生的位置，程序在该点处停止执行。以下是断点事件的事件规范。

in function

已输入函数，并即将执行第一行。序进程后的第一个可执行代码将用作实际断点位置。它可以是要初始化的局部变量所在的那一行。如果是 C++ 构造函数，执行所有基类构造函数后，执行便会停止。如果使用了 `-instr` 修饰符（请参见第 272 页的“`-instr`”），则它为即将执行函数的第一个指令。*function* 规范可以接受正式的参数签名，以协助重载的函数名或模板实例规范。例如：

```
停止于 mumble(int, float, struct Node *) 中
```

注 – 不要混淆 `in function` 与 `-infunction` 修饰符。

`at [filename:] line_number`

指定行即将被执行。如果指定了 `filename`，则即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。尽管不要求使用引号，但文件名中包含特殊字符时可能有使用的必要。如果指定行在模板代码中，断点会被置于该模板的所有实例上。

`at address_expression`

给定地址处的指令即将被执行。此事件只能与 `stopi` 命令（请参见第 352 页的“`stopi` 命令”）或 `-instr` 事件修饰符（请参见第 272 页的“`-instr`”）一起使用。

`infunction function`

等价于所有以 `function` 命名的重载函数的 `in function` 或其所有模板实例化。

`inmember function`

`inmethod function`

等价于 `in function` 或每一类以 `function` 命名的成员函数。

`inclass classname [-recurse | -norecurse]`

对所有是 `classname` 成员，但不是 `classname` 的任何基类的成员函数而言，等价于 `in function`。 `-norecurse` 是默认值。如果指定 `-recurse`，则基类被包括进来。

`inobject object-expression [-recurse | -norecurse]`

在 `object-expression` 表示的地址处的特定对象上调用的成员函数已被调用。 `stop inobject ox` 大致等价于以下形式，但与 `inclass` 不同的是，它包括 `ox` 的动态类型的基类。 `-recurse` 是默认值。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

以下是涉及访问或更改内存地址内容的事件的事件规范。

access mode address-expression [, byte-size-expression]

已访问 *address-expression* 指定的内存。

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 也可以包含以下两个字母之一：

- a 访问后停止进程（默认值）。
- 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”或“之后”指副作用。

address-expression 是可对其求值来生成地址的任何表达式。如果给出符号表达式，则可以自动推导出要监视的区域大小，可以指定 *byte-size-expression* 来将其覆盖。也可以使用非符号、无类型地址表达式。使用该表达式时，必须提供大小。例如：

```
stop access w 0x5678, sizeof(Complex)
```

`access` 命令有这样一个限制，即两个匹配区不能重叠。

注 - `access` 事件规范是 `modify` 事件规范的替代品。

change *variable*

variable 的值已更改。change 事件大致等价于：

```
when step { if [ $last_value !=${variable}] then
              stop
            else
              last_value=${variable}
            }
```

此事件使用单步执行来执行。要提高执行速度，请使用 access 事件（请参见第 263 页的“access mode address-expression [, byte-size-expression]”）。

即使未检测到更改，第一次检查 *variable* 时也会触发一个事件。这第一个事件提供对 *variable* 初始值的访问。以后检测到 *variable* 值的变化时会触发其他事件。

cond *condition-expression*

condition-expression 表示的条件的求值为真。可以为 *condition-expression* 指定任何表达式，但其求得的值必须为整型。cond 事件大致等价于：

```
stop step -if conditional_expression
```

系统事件规范

以下是系统事件的事件规范。

dlopen [*lib-path*] | dlclose [*lib-path*]

这些事件在成功调用 dlopen() 或 dlclose() 后发生。一次 dlopen() 或 dlclose() 调用可触发装入一个以上库。任何时候均可在预定义变量 `$dllib` 中得到这些库的列表。`$dllib` 中的第一个 shell 字为“+”或“-”，指示是正在添加还是正在删除的库列表。

lib-path 是共享库的名称。如果指定了该名称，则只有在装入或卸下给定库时才会发生事件。在该情况下，`$dlobj` 包含库名称。`$dllib` 仍然可用。

如果 *lib-path* 以 / 开头，则会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*，则无论是否有 dl- 活动，事件都会发生。在这种情况下，`$dlobj` 为空，但 `$dllib` 有效。

fault *fault*

遇到指定错误时，`fault` 事件便会发生。这些错误为体系结构相关式错误。以下这组 `dbx` 已知的错误在 `proc(4)` 手册页中有定义。

错误	描述
FLTILL	非法指令
FLTPRIV	特权指令
FLTBPT*	断点陷阱
FLTRACE*	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	无法恢复的栈错误
FLTPAGE	可恢复的页错误
FLTWATCH*	监视点陷阱
FLTCPCOVF	CPU 性能计数器溢出

注 – `dbx` 使用 `BPT`、`TRACE` 和 `BOUNDS` 来执行断点和单步执行。处理它们也许会干扰 `dbx` 的运行方式。

这些错误来自 `/sys/fault.h`。*fault* 可以是上列中的任何一种，可以使用大写或小写字母，带或不带 `FLT-` 前缀或实际数字代码。

注 – 在 Linux 平台上不能使用 `fault` 事件。

`lwp_exit`

退出 `lwp` 时，`lwp_exit` 事件便会发生。在事件处理程序期间内，`$lwp` 包含退出 LWP（轻量进程）的 `id`。

注 – 在 Linux 平台上不能使用 `lwpexit` 事件。

sig signal

信号首次传送给调试中的程序时，*sig signal* 事件便会发生。*signal* 可以是小数，信号名可以使用大写或小写字母；前缀为可选项。它与 *catch* 和 *ignore* 命令毫不相关，尽管 *catch* 命令可按如下方式执行：

```
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。只有在指定信号为传送给它的信号时，才能继续进程。

sig signal sub-code

具有指定 *sub-code* 的指定信号首次传送到子进程时，*sig signal sub-code* 事件便会发生。与信号相同，可以以小数形式键入 *sub-code*，使用大写或小写字母；前缀为可选项。

sysin code | name

刚启动指定的系统调用，且进程已进入内核模式。

dbx 所支持的系统调用的概念是由陷阱以在 `/usr/include/sys/syscall.h` 中枚举的形式提供到内核。

这与系统调用的 *ABI* 概念不同。一些 *ABI* 系统调用在用户模式下被部分执行，并使用非 *ABI* 内核陷阱。但大多数普通系统调用（主要差异是信号处理）在 `syscall.h` 和 *ABI* 间是相同的。

注 - 在 *Linux* 平台上不能使用 *sysin* 事件。

注 - `/usr/include/sys/syscall.h` 中的内核系统调用陷阱列表各版本均不相同的 Solaris 操作环境中的专用界面的一部分。dbx 接受的陷阱名（代码）和陷阱编号列表包括 dbx 支持的 Solaris 操作环境所有版本所支持的所有陷阱名（代码）和陷阱编号。dbx 支持的名称与 Solaris 操作环境的任何特定版本所支持的名称完全相同不太可能，而且 `syscall.h` 中的某些名称也许会不可用。任何陷阱编号（代码）均可被 dbx 所接受，并会正常运行。不过，如果它与已知系统调用陷阱不对应，系统会发出警告。

`sysout code | name`

指定系统调用完成，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`

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）会触发子进程为其自身目的执行系统调用，并会在被跟踪时现身。

执行进度事件规范

以下是与执行进度有关的事件的事件规范。

`exit exitcode`

退出时，`exit` 事件便会发生。

`next`

`next` 事件与 `step` 事件类似，只不过它不步入函数。

`returns`

`returns` 事件是一个断点，它位于当前被访问函数的返回点上。使用被访问函数的目的是可以在提供大量 `step up` 命令后使用 `returns` 事件规范。`returns` 事件始终为 `-temp`，只有在存在活动进程时，才可以创建它。

returns *function*

`returns function` 事件每当给定函数返回到其调用处时执行。这不是临时事件。未提供返回值，但可以访问以下寄存器来找到整型返回值：

基于 SPARC 的系统	\$o0
基于 x86 的系统	\$eax
基于 x64 的系统	\$rax、\$rdx

该事件大致等价于：

```
when in func { stop returns; }
```

step

源代码行的第一个指令执行时，`step` 事件便会发生。例如，可以使用以下语句获得简单跟踪：

```
when step { echo $lineno:$line; }; cont
```

启用 `step` 事件时，便是指示 `dbx` 在下次使用 `cont` 命令时自动单步执行。

注 - `step` 命令终止时，`step`（和 `next`）事件才会发生。`step` 命令按 `step` 事件大致以如下方式执行：

```
alias step="when step -temp { whereami; stop; }; cont"
```

其他事件规范

以下是其他类型事件的事件规范。

attach

`dbx` 已成功连接到进程。

detach

dbx 已成功地从正被调试的程序中分离。

lastrites

正被调试的进程即将过期，以下原因可导致发生这种情况：

- 已调用 `_exit(2)` 系统调用。（通过显式调用或 `main()` 返回时，便会发生这种情况。）
- 即将传送终止信号。
- 进程正由 `kill` 命令终止。

触发此事件时，进程的最终状态通常（并非一直）处于可用状态，从而给您提供检查进程状态的最后机会。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

dbx 不再与被调试的进程关联时，`proc_gone` 事件便会发生。预定义变量 `$reason` 可以是 `signal`、`exit`、`kill` 或 `detach`。

prog_new

因 `follow exec` 而装入新程序时，`prog_new` 事件便会发生。

注 – 此事件的处理程序始终永久性存在。

stop

进程已停止。每当进程以这种方式停止：用户得到提示（尤其是在响应 `stop` 处理程序时），`stop` 事件便会发生。例如，以下命令等价：

```
display x
when stop {print x;}
```

sync

刚使用 `exec()` 执行了正被调试的进程。在 `a.out` 中指定的所有内存均有效并存在，但尚未装入预装的共享库。例如，`printf` 尽管可供 `dbx` 使用，但尚未映射到内存中。

对此事件执行 `stop` 无效；不过，可以使用带 `when` 命令的 `sync` 事件。

注 – 在 Linux 平台上不能使用 `sync` 事件。

syncrtld

`syncrtld` 事件在 `sync`（或 `attach`，如果正被调试的进程尚未处理共享库）后发生。它在动态链接程序启动代码执行且所有预装共享库的符号表都已装入后，`.init` 部分中的任何代码运行前执行。

对此事件执行 `stop` 无效；不过，可以使用带 `when` 命令的 `syncrtld` 事件。

thr_create [*thread_id*]

在创建线程或创建具有指定 `thread_id` 的线程后，便会发生 `thr_create` 事件。例如，在下面的 `stop` 命令中，线程 ID `t@1` 是指正在创建的线程，而线程 ID `t@5` 是指已创建的线程。

```
stop thr_create t@5 -thread t@1
```

thr_exit

线程退出后，便会发生 `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
```

throw

每当应用程序抛出不是未处理过的或意外的异常时，`throw` 事件便会发生。

注 – 在 Linux 平台上不能使用 `throw` 事件。

throw type

如果为 `throw` 事件指定了异常 *type*，则只有该类型的异常会触发 `throw` 事件。

`throw -unhandled`

`-unhandled` 是一种特殊的异常类型，用来表示一个被抛出但没有相应处理程序的异常。

`throw -unexpected`

`-unexpected` 是一种特殊的异常类型，用于表示不符合抛出它的函数的异常规范的异常。

timer seconds

正在调试的程序持续运行 *seconds* 后 `timer` 事件便会发生。用于此事件的计时器与 `collector` 命令共享。精度为毫秒，因此例如 0.001 这一浮点值对 *seconds* 是可以接受的。

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，最常见的类型是事件过滤器。修饰符必须位于事件规范的关键字部分之后。修饰符以短线 (-) 开头。以下是有效的事件规范修饰符。

-if condition

事件规范指定的事件发生时，便会对条件求值。只有在对条件的求值不是零时，才允许处理程序的副作用。

如果 `-if` 修饰符用于的事件有与之关联的单个源位置，如 `in` 或 `at`，则会在与该位置对应的作用域中对 *condition* 求值。否则，请使用所需作用域来限定它。

`-resumeone`

可以在多线程程序的事件规范中将 `-resumeone` 修饰符与 `-if` 修饰符一并使用，如果条件中包含函数调用，它只会触发一个线程恢复。有关详细信息，请参见第 103 页的“将过滤器与条件事件配合使用”。

`-in function`

只有在事件的发生时间介于达到给定 *function* 的第一条指令的时间和函数返回时间之间时，事件才会触发。函数的递归会被忽略。

`-disable`

`-disable` 修饰符会在禁用状态下创建处理程序。

`-count n`

`-count infinity`

`-count n` 和 `-count infinity` 修饰符让处理程序从 0 开始计数（请参见第 261 页的“使用事件计数器”）。每次事件发生时，计数便会增加，直至达到 *n*。达到 *n* 后，处理程序便会启动，计数器会被重置为零。

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，这些计数是在 `sync` 事件发生时被重置。

使用 `debug -r` 命令（请参见第 300 页的“`debug` 命令”）或 `attach -r` 命令（请参见第 282 页的“`attach` 命令”）调试新程序时，计数便会被重置。

`-temp`

创建临时处理程序。事件发生后，它会被自动删除。默认情况下，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，它才会被自动删除。

使用 `delete -temp` 命令删除所有临时处理程序。

`-instr`

使处理程序在指令级起作用。此事件将取代大多数命令传统的“`i`”后缀。它通常会修改事件处理程序的两个方面：

- 任何消息均打印汇编级信息，而非源码级信息。
- 事件的粒度变为指令级。例如，`step -instr` 意指指令级单步执行。

`-thread thread_id`

只有在触发事件的线程与 *thread_id* 匹配时，才会执行该操作。您印象中的特定线程在程序的不同次执行时可能被分配不同的 *thread_id*。

`-lwp lwp_id`

只有在触发事件的线程与 `lwp_id` 匹配时，才会执行该操作。只有在触发事件的线程与 `lwp_id` 匹配时，才会执行该操作。您印象中的特定线程在程序的不同次执行时可能被分配不同的 `lwp_id`。

`-hidden`

在常规 `status` 命令中隐藏处理程序。使用 `status -h` 查看隐藏的处理程序。

`-perm`

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` 修饰符在所有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。使用 `delete -p` 来删除永久性处理程序。

分析和二义性

事件规范和修饰符的语法为：

- 受驱动的关键字
- 根据 `ksh` 惯例，所有关键字均拆分为由空格分隔的多个字

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
```

在第一个例子中，尽管应用程序可能有名为 `temp` 的变量，但 `dbx` 分析器仍会优先按 `-temp` 作为修饰符来解析事件规范。在第二个例子中，`a-temp` 被整体传递给语言特定表达式分析器。一定存在名为 `a` 和 `temp` 的变量，不然就是发生了错误。使用括号来强制分析。

使用预定义变量

提供了一些只读 ksh 预定义变量。以下变量一直有效：

变量	定义
<code>\$ins</code>	反汇编当前指令。
<code>\$lineno</code>	以十进制数表示的当前行号。
<code>\$vlineno</code>	以十进制数表示的当前“访问”行号。
<code>\$line</code>	当前行的内容。
<code>\$func</code>	当前函数名。
<code>\$vfunc</code>	当前“访问”函数的名称。
<code>\$class</code>	<code>\$func</code> 所属类的名称。
<code>\$vclass</code>	<code>\$vfunc</code> 所属类的名称。
<code>\$file</code>	当前文件名。
<code>\$vfile</code>	被访问的当前文件的名称。
<code>\$loadobj</code>	当前可装入对象的名称。
<code>\$vloadobj</code>	被访问的当前可装入对象的名称。
<code>\$范围</code>	当前 PC 在反引用符号中的作用域。
<code>\$scope</code>	被访问 PC 在反引用符号中的作用域。
<code>\$funcaddr</code>	以十六进制数表示的 <code>\$func</code> 地址。
<code>\$caller</code>	调用 <code>\$func</code> 的函数的名称。
<code>\$dllist</code>	<code>dlopen</code> 或 <code>dlclose</code> 事件后，包含刚装入或卸下的装入对象的列表。根据发生的是 <code>dlopen</code> 还是 <code>dlclose</code> ， <code>dllist</code> 的第一个字将为“+”或“-”。
<code>\$newhandlerid</code>	最近创建的处理程序的 ID。在任何删除处理程序的命令后，此变量具有一个未定义的值。创建处理程序后立即使用变量。dbx 无法为创建多个处理程序的命令捕获所有处理程序 ID。
<code>\$firedhandlers</code>	导致最近中断的处理程序 id 列表。该列表中的处理程序在 <code>status</code> 命令的输出中以“*”标记。
<code>\$proc</code>	正被调试的当前进程的“进程 ID”。
<code>\$lwp</code>	当前 LWP 的 Lwp ID。
<code>\$thread</code>	当前线程的“线程 ID”。
<code>\$newlwp</code>	新创建的 LWP 的 lwp ID。
<code>\$newthread</code>	新创建的线程的线程 ID。

变量	定义
\$prog	正被调试程序的完整路径名。
\$oprog	\$prog 的上一个值，当 \$prog 还原为“-”时，它被用在 exec() 之后来返回到上一调试处。\$prog 扩展为完整路径名时，\$oprog 包含在命令行上指定的程序路径或到 debug 命令的程序路径。如果调用 exec() 一次以上，则无法返回到原始程序。
\$exitcode	最后运行程序的退出状态。如果没有退出进程，则该值为空字符串。

举一个例子，假设 whereami 可以这样执行：

```
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

以下变量只有在 when 命令体内有效。

\$handlerid

在体执行过程中，\$handlerid 是体所属 when 命令的 id。这些命令是等价的：

```
when X -temp { do_stuff; }
when X { do_stuff; delete $handlerid; }
```

\$booting

如果事件在 boot 进程期间发生，则 \$booting 会被设置为 true。每当调试新程序时，它会在用户不知晓的情况下先运行，以便确定共享库的列表和位置。然后，进程将被终止。这一序列被称为引导。

引导过程中，所有事件仍然可用。使用此变量来区分 debug 期间发生的 sync 和 syncrtld 事件以及正常 run 期间发生的事件。

将 \$newlwp 设置为新创建的 LWP 的 lwp-id。

\$newthread

对特定事件有效的变量

某些变量只对以下各表中显示的特定事件有效。

表 B-1 对 sig 事件有效的变量

变量	描述
\$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 B-2 对 exit 事件有效的变量

变量	描述
\$exitcode	传递给 <code>_exit(2)</code> 或 <code>exit(3)</code> 的参数的值或 <code>main</code> 的返回值

表 B-3 对 `dlopen` 和 `dlclose` 事件有效的变量

变量	描述
\$dlobj	装入对象 <code>dlopened</code> 或 <code>dlclosed</code> 的路径名

表 B-4 对 `sysin` 和 `sysout` 事件有效的变量

变量	描述
\$syscode	系统调用号
\$sysname	系统调用名

表 B-5 对 `proc_gone` 事件有效的变量

变量	描述
\$reason	signal、exit、kill 或 detach 之一

设置事件处理程序示例

以下是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

要在 `array[99]` 中设置断点，请键入：

```
(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

要执行简单跟踪，请键入：

```
(dbx) when step { echo at line $lineno; }
```

在函数内时启用处理程序 (*in function*)

要在函数内时启用处理程序，请键入：

```
<dbx> trace step -in foo
```

它等价于：

```
# 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

要了解在小程序中已执行了多少行，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
```

程序从不会停止 — 程序只会终止。执行的行数为 133。此进程速度很慢。它对被多次调用的函数中的断点的用处最大。

确定源代码行执行的指令数

要计算一行代码执行了多少条指令，请键入：

```
(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
```

如果步出的行进行了函数调用，则被调用函数中的行也会被计算在内。可以使用 `next` 事件代替 `step` 来计算不包括被调用函数的指令数。

事件发生后启用断点

只在另一事件发生后启用断点。例如，如果程序开始在函数 `hash` 中不正确地执行，但这种情况只发生在第 1300 个符号查找后，便可键入：

```
(dbx) when in lookup -count 1300 {
    stop in hash
    hash_bpt=$newhandlerid
    when proc_gone -temp { delete $hash_bpt; }
}
```

注 - `$newhandlerid` 指刚执行的 `stop in` 命令。

为 replay 而重置应用程序文件

如果应用程序处理需在 `replay` 期间重置的文件，可以编写一个处理程序，以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

要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` 来查看运行中程序的栈跟踪。

这基本上是收集器人工示例模式的任务（还有更多）。使用 `SIGQUIT` (`^\`) 来中断程序，因为 `^c` 现已用尽。

捕获浮点异常

要只捕获特定浮点异常，例如 IEEE 下溢，请键入：

```
(dbx) ignore FPE # turn off default handler
(dbx) help signals | grep FPE # can't remember the subcode name
...
(dbx) stop sig fpe FPE_FLTUND
...
```


附录 C

命令参考

本附录提供所有 dbx 命令的详细语法和功能说明。

adb 命令

adb 命令执行一个 adb 样式的命令或设置 adb 模式。仅在本地模式中有效。

语法

adb <i>adb-command</i>	执行一个 adb 样式的命令。
adb	设置 adb 模式；使用 \$q 退出 adb 模式。

assign 命令

在本地模式中，assign 命令为程序变量分配一个新值。在 Java 模式中，assign 命令为局部变量或参数分配一个新值。

本地模式语法

```
assign variable = expression
```

其中：

expression 是赋给 *variable* 的值。

Java 模式语法

assign identifier = expression

其中：

class_name 是 Java 类的名称，可以使用包路径（使用句点 (.) 作为限定符；例如 `test1.extra.T1.Inner`），也可以使用完整路径名（前面加磅符号 (#) 并使用斜线 (/) 和美元符号 (\$) 作为限定符；例如 `#test1/extra/T1$Inner`）。如果使用 \$ 限定符，请用引号将 *class_name* 引起来。

expression 是一个有效的 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*)。

object_name 是 Java 对象名。

attach 命令

`attach` 命令将 `dbx` 连接到正在运行的进程，停止执行并将程序置于调试控制下。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

<code>attach process_id</code>	使用进程 ID <i>process_id</i> 开始调试程序。 <code>dbx</code> 使用 <code>/proc</code> 查找程序。
<code>attach -p process_id program_name</code>	使用进程 ID <i>process_id</i> 开始调试 <i>program</i> 。
<code>attach program_name process_id</code>	使用进程 ID <i>process_id</i> 开始调试 <i>program</i> 。 <i>program</i> 可以是 -； <code>dbx</code> 使用 <code>/proc</code> 查找它。
<code>attach -r ...</code>	-r 选项可使 <code>dbx</code> 保留所有的 <code>display</code> 、 <code>trace</code> 、 <code>when</code> 和 <code>stop</code> 命令。如果没有 -r 选项，则执行隐式 <code>delete all</code> 和 <code>undisplay 0</code> 。

其中：

process_id 是正在运行的进程的进程 ID。

program_name 是运行程序的路径名。

要连接一个运行中的 Java 进程：

1. 通过将 `libdbx_agent.so` 添加到 `LD_LIBRARY_PATH`。 `libdbx_agent.so`，确保 JVM™ 软件可以找到 `libdbx_agent.so`：
 - 如果在运行 Solaris OS 的系统上使用 JVM 软件的 32 位版本，请将 `/installation_directory/prod/lib/dbxruntime` 添加到 `LD_LIBRARY_PATH`。
 - 如果在运行 Solaris OS、基于 SPARC 的系统上使用 JVM 软件的 64 位版本，请将 `/installation_directory/prod/lib/v9/dbxruntime` 添加到 `LD_LIBRARY_PATH`。
 - 如果在运行 Linux OS、基于 x64 的系统上使用 JVM 软件的 64 位版本，请将 `/installation_directory/prod/lib/amd64/dbxruntime` 添加到 `LD_LIBRARY_PATH`。

`installation_directory` 是 Sun Studio 软件的安装位置。

2. 通过键入以下命令启动 Java 应用程序：

```
java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrun dbx_agent  
myclass.class
```

3. 然后，可以通过使用进程 `id` 启动 `dbx` 来连接进程：

```
dbx - process_id
```

bsearch 命令

`bsearch` 命令在当前源文件中向后搜索。仅在本地模式中有效。

语法

<code>bsearch string</code>	在当前文件中向后搜索 <code>string</code> 。
-----------------------------	----------------------------------

<code>bsearch</code>	使用上一搜索字符串重复搜索。
----------------------	----------------

其中：

`string` 是一个字符串。

call 命令

在本地模式中，call 命令调用一个过程。在 Java 模式中，call 命令调用一个方法。

本地模式语法

```
call procedure([parameters])
```

其中：

procedure 是过程名。

parameters 是过程的参数。

您也可以使用 call 命令调用函数；使用 print 命令查看函数的返回值（请参见第 337 页的“print 命令”）。

有时，调用的函数会到达断点。您可以使用 cont 命令（请参见第 296 页的“cont 命令”）选择继续，或通过使用 pop -c（请参见第 336 页的“pop 命令”）终止调用。如果调用的函数引发段错误，则可以使用后者。

Java 模式语法

```
call [class_name.|object_name.]method_name([parameters])
```

其中：

class_name 是 Java 类的名称，可以使用包路径（使用句点(.)作为限定符；例如 test1.extra.T1.Inner），也可以使用完整路径名（前面加磅符号(#)并使用斜线(/)和美元符号(\$)作为限定符；例如 #test1/extra/T1\$Inner）。如果使用 \$ 限定符，请用引号将 *class_name* 引起来。

object_name 是 Java 对象名。

method_name 是 Java 方法名。

parameters 是方法的参数。

有时，调用的方法会到达断点。您可以使用 cont 命令（请参见第 296 页的“cont 命令”）选择继续，或通过使用 pop -c（请参见第 336 页的“pop 命令”）终止调用。如果调用的方法引发段错误，则可以使用后者。

cancel 命令

`cancel` 命令取消当前信号。它主要用于 `when` 命令体内（请参见第 375 页的“`when` 命令”）。仅在本地模式中有效。

当 `dbx` 由于某个信号而导致停止时，信号通常会被取消。如果 `when` 命令已与信号事件连接，则该信号将不会被自动取消。`cancel` 命令可用于显式取消信号。

语法

```
cancel
```

catch 命令

`catch` 命令捕获指定的信号。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

<code>catch</code>	打印已捕获信号的列表。
<code>catch number number...</code>	捕获编号为 <i>number</i> 的信号。
<code>catch signal signal...</code>	捕获名为 <i>signal</i> 的信号。无法捕获或忽略 SIGKILL。
<code>catch \$(ignore)</code>	捕获所有信号。

其中：

number 是信号的编号。

signal 是信号名。

check 命令

check 命令可启用内存访问、泄漏或使用检查，并打印出运行时检查 (RTC) 的当前状态。仅在本地模式中有效。

注 – 只能在 Solaris 平台上执行 check 命令。

由此命令启用的运行时检查功能将被 debug 命令重置为初始状态。

语法

check -access

打开访问检查。RTC 报告以下错误：

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
rua	从未分配的内存中读
rui	从未初始化的内存中读
wro	写入到只读内存
wua	写入到未分配内存

默认行为是在检测到每个访问错误后停止进程，可使用 `rtc_auto_continue` dbx 环境变量进行更改。如果设置为 `on`，访问错误将被记录到一个文件中（文件名由 `dbx` 环境变量 `rtc_error_log_file_name` 控制）。请参见第 299 页的“`dbxenv` 命令”。

默认情况下，每个独特访问错误只在第一次发生时报告。可使用 `dbx` 环境变量 `rtc_auto_suppress` 更改此行为（此变量默认设置为 `on`）。请参见第 299 页的“`dbxenv` 命令”。

`check -leaks [-frames n] [-match m]`

打开泄漏检查。RTC 报告以下错误：

<code>aib</code>	可能的内存泄漏 - 仅指针指向块中央
<code>air</code>	可能的内存泄漏 - 指向块的指针仅存在于寄存器中
<code>mel</code>	内存泄漏 - 没有指针指向块

打开泄漏检查时，如果程序退出，便会收到自动产生的泄漏报告。届时将报告包括可能的泄漏在内的所有的泄漏。默认情况下，将生成一个非冗余报告（由 `dbx` 环境变量 `rtc_mel_at_exit` 控制）。但是，您也可以随时查看泄漏报告（请参见第 348 页的“`showleaks` 命令”）。

`-frames n` 表明在报告泄漏时，最多将显示 *n* 个不同栈帧。`-match m` 用于合并泄漏；如果两个或更多泄漏分配时的调用栈与 *n* 帧匹配，则这些泄漏将记录在单个合并泄漏报告中。

n 的默认值为 8 或 *m* 的值（取较大值）。*n* 最大值为 16。对于 C++，*m* 的默认值为 3，其他为 2。

`check -memuse [-frames n] [-match m]`

打开内存使用 (`memuse`) 检查。`check -memuse` 也表明检查泄漏。除退出程序时的泄漏报告外，您还可获取块使用的情况 (`biu`) 报告。默认情况下，将生成一个非冗余块使用情况报告（由 `dbx` 环境变量 `rtc_biu_at_exit` 控制）。运行程序时，您可以随时查看程序中内存的分配位置（请参见第 349 页的“`showmemuse` 命令”）。

`-frames n` 表明在报告内存使用和泄漏时，最多将显示 *n* 个不同的栈帧。`-match m` 用于合并这些报告；如果两个或更多泄漏分配时的调用栈与 *m* 帧匹配，则这些泄漏将记录在单个合并内存泄漏报告中。

n 的默认值为 8 或 *m* 的值（取较大值）。*n* 最大值为 16。对于 C++，*m* 的默认值为 3，其他为 2。另请参见 `check -leaks`。

`check -all [-frames n] [-match m]`

等价于 `check -access` 或 `check -memuse [-frames n] [-match m]`

`dbx` 环境变量 `rtc_biu_at_exit` 的值没有用 `check -all` 更改，因此在默认情况下，退出时没有生成内存使用报告。有关 `rtc_biu_at_exit` 环境变量的说明，请参见第 297 页的“`dbx` 命令”。

`check [functions] [files] [loadobjects]`

等价于 `functions`、`files` 和 `loadobjects` 中的 `check -all`、`suppress all`、`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 是一个或多个加载对象名

您可使用此命令将运行时检查用于需要的地方。

注 – 要检测所有错误，RTC 不要求程序使用 `-g` 编译。但是，为了保证准确检测某些错误（通常从未初始化的内存中读取），需要使用符号 (`-g`) 信息。为此，如果未提供符号信息，某些错误（`a.out` 的 `rui`，共享库的 `rui + aib + air`）会被禁止。此行为可通过使用 `suppress` 和 `unsuppress` 更改。

clear 命令

`clear` 命令清除断点。仅在本地模式中有效。

使用具有 `inclass`、`inmethod` 或 `infunction` 参数的 `stop`、`trace` 或 `when` 命令创建的事件处理程序可创建多个断点集。如果您在 `clear` 命令中指定的 *line* 与这些断点之一匹配，则只清除该断点。一旦以此方法清除，属于该集合的单个断点将无法再启用。但是，禁用并再次启用相关的事件处理程序将重建所有的断点。

语法

<code>clear</code>	清除当前停止点的所有断点。
<code>clear <i>line</i></code>	清除 <i>line</i> 中的所有断点。
<code>clear <i>filename:line</i></code>	清除 <i>filename</i> 中 <i>line</i> 处的所有断点。

其中：

line 是源代码行的编号。

filename 是源代码文件的名称。

collector 命令

`collector` 命令收集性能数据用于性能分析器进行分析。仅在本地模式中有效。

注 – 只能在 Solaris 平台上执行 `collector` 命令。

语法

<code>collector <i>command_list</i></code>	指定一个或多个收集器命令
<code>archive <i>options</i></code>	指定实验终止时的归档模式（请参见第 290 页的“collector archive 命令”）。
<code>dbxsample <i>options</i></code>	控制 dbx 停止目标进程时对样例的收集（请参见第 290 页的“collector dbxsample 命令”）。
<code>disable</code>	停止收集数据并关闭当前实验（请参见第 290 页的“collector disable 命令”）。
<code>enable</code>	启用收集器并打开新实验（请参见第 290 页的“collector enable 命令”）。
<code>heaptrace <i>options</i></code>	启用或禁用收集堆跟踪数据（请参见第 291 页的“collector heaptrace 命令”）
<code>hwprofile <i>options</i></code>	指定硬件计数器配置设置（请参见第 291 页的“collector hwprofile 命令”）。
<code>limit <i>options</i></code>	限制记录的分析数据的数量（请参见第 292 页的“collector limit 命令”）。
<code>mpitrace <i>options</i></code>	启用或禁用收集 MPI 跟踪数据（请参见第 292 页的“collector mpitrace 命令”）。
<code>pause</code>	停止收集性能数据但不关闭实验（请参见第 292 页的“collector pause 命令”）。
<code>profile <i>options</i></code>	指定收集调用栈配置数据的设置（请参见第 292 页的“collector profile 命令”）。
<code>resume</code>	暂停后继续性能数据收集（请参见第 293 页的“collector resume 命令”）。
<code>sample <i>options</i></code>	指定抽样设置（请参见第 293 页的“collector sample 命令”）。
<code>show <i>options</i></code>	显示当前收集器设置（请参见第 294 页的“collector show 命令”）。
<code>status</code>	查询当前实验的状态（请参见第 294 页的“collector status 命令”）。
<code>store <i>options</i></code>	实验文件控制和设置（请参见第 294 页的“collector store 命令”）。
<code>synctrace <i>options</i></code>	指定收集线程同步等待跟踪数据的设置（请参见第 295 页的“collector synctrace 命令”）。
版本	报告将用于收集数据的 libcollector.so 版本（请参见）。

其中：

options 是可以用每个命令指定的设置。

要开始收集数据，键入 `collector enable`。

要结束数据收集，键入 `collector disable`。

collector archive 命令

`collector archive` 命令指定实验终止时所使用的归档模式。

语法

<code>collector archive</code>	默认情况下，使用正常归档。如果不归档，请指定 <code>off</code> 。要将加载对象复制到试验，以方便移植，请指定 <code>copy</code> 。
--------------------------------	--

collector dbxsample 命令

`collector dbxsample` 命令指定当进程由 `dbx` 停止时是否记录抽样。

语法

<code>collector dbxsample</code>	默认情况下，当进程由 <code>dbx</code> 停止时，收集抽样。若本次不收集抽样，请指定 <code>off</code> 。
----------------------------------	--

collector disable 命令

`collector disable` 命令导致数据收集停止，并关闭当前实验。

语法

`collector disable`

collector enable 命令

`collector enable` 命令启用收集器并打开一个新实验。

语法

```
collector enable
```

collector heaptrace 命令

collector heaptrace 命令为收集堆跟踪（内存分配）数据指定选项。

语法

```
collector heaptrace on|off
```

默认情况下，不收集堆跟踪数据。要收集此数据，请指定 on。

collector hwprofile 命令

collector hwprofile 命令为收集硬件计数器溢出配置数据指定选项。

语法

```
collector hwprofile on|off
```

默认情况下，不收集计数器溢出性能分析数据。要收集此数据，请指定 on。

```
collector hwprofile list
```

打印可用计数器列表。

```
collector hwprofile counter name interval [ name2 interval2 ]
```

其中：

name 是硬件计数器的名称。

interval 是收集时间间隔，以毫秒计。

name2 是第二个硬件计数器的名称。

interval2 是收集时间间隔，以毫秒计。

硬件计数器由系统指定，因此可供选择的计数器取决于您所使用的系统。许多系统不支持硬件计数器溢出配置。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
```

其中：

value（单位 MB）限制所记录的配置数据总量。当达到限制时，不再记录分析数据，但实验会保持打开状态，且继续记录样例点。对记录数据的默认限制为 2000 MB。

collector mpitrace 命令

collector heaptrace 命令为收集 MPI 跟踪数据指定选项。

语法

collector mpitrace	默认情况下，不收集 MPI 跟踪数据。要收集此数据，请指定
on off	on。

collector pause 命令

collector pause 命令导致数据收集停止，但不关闭当前实验。暂停收集器时，不记录样例点。样例是在暂停之前生成的，另一个样例则是在恢复之后立即生成的。可使用 collector resume 命令恢复数据收集（请参见第 293 页的“collector resume 命令”）。

语法

```
collector pause
```

collector profile 命令

收集器性能分析命令为收集性能分析数据指定选项。

语法

<code>collector profile on off</code>	指定性能分析数据收集模式
<code>collector profile timer <i>interval</i></code>	指定定点或浮点性能分析计时器周期，可在后面添加 <code>m</code> 表示毫秒或添加 <code>u</code> 表示微秒。

collector resume 命令

`collector resume` 命令可在由 `collector pause` 命令创建的暂停后恢复数据收集（请参见第 292 页的“`collector pause` 命令”）。

语法

```
collector resume
```

collector sample 命令

`collector sample` 命令指定抽样模式和抽样时间间隔。

语法

<code>collector sample periodic manual</code>	指定抽样模式。
<code>collector sample period <i>seconds</i></code>	以 <i>seconds</i> 为单位指定抽样时间间隔。
<code>collector sample record [<i>name</i>]</code>	使用可选 <i>name</i> 记录抽样。

其中：

seconds 是抽样时间间隔长度。

name 是抽样名称。

collector show 命令

collector show 命令显示一种或多种选项设置。

语法

collector show all	显示所有设置。
collector show archive	显示归档设置。
collector show profile	显示调用栈配置设置。
collector show synctrace	显示线程同步等待跟踪设置。
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 mpitrace	显示 MPI 跟踪数据设置。
collector show sample	显示抽样设置。
collector show store	显示存储设置。

collector status 命令

collector status 命令查询当前实验的状态。

语法

```
collector status
```

collector store 命令

collector store 指定存储实验的目录和文件名。

语法

<code>collector store</code> <code>directory <i>pathname</i></code>	指定存储实验的目录。
<code>collector store</code> <code>filename <i>filename</i></code>	指定实验文件名。
<code>collector store group</code> <code><i>string</i></code>	指定实验组名

其中：

pathname 是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` 命令为收集同步等待跟踪数据指定选项。

语法

<code>collector synctrace</code> <code>on off</code>	默认情况下，不收集线程同步等待跟踪数据。要收集此数据，请指定 <code>on</code> 。
<code>collector threshold</code> <code><i>microseconds</i></code>	以微秒为单位指定阈值。默认值是 1000。
<code>collector threshold</code> <code>calibrate</code>	自动计算出阈值

其中：

microseconds 是阈值，在此阈值以下，将丢弃同步等待事件。

collector version 命令

`collector version` 命令报告将用于收集数据的 `libcollector.so` 版本。

语法

collector version

cont 命令

cont 命令使进程继续执行。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

cont	继续执行。MT 进程中的所有线程都会被恢复。使用 Control-C 停止执行程序。
cont ...-sig <i>signal</i>	使用信号 <i>signal</i> 继续执行。
cont ... <i>id</i>	<i>id</i> 指定要继续的线程或 LWP。
cont at <i>line</i> [<i>id</i>]	如果应用程序是多线程的，则要求在行 <i>line.id</i> 处继续执行。
cont ...-follow parent child both	如果将 dbx follow_fork_mode 环境变量设置为 ask，并且已经选择了 stop，可使用此选项选择要跟随的进程。both 仅在 Sun Studio IDE 中适用。

dalias 命令

dalias 命令定义一个 dbx 样式（csh 样式）的别名。仅在本地模式中有效。

语法

dalias	(dbx 别名) 列出所有当前已定义的别名
dalias <i>name</i>	列出别名 <i>name</i> 的定义（如果有）。
dalias <i>name definition</i>	将 <i>name</i> 定义为 <i>definition</i> 的一个别名。 <i>definition</i> 可以包含空白。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 接受在别名中普遍使用的以下 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"
```

有关详细信息，请参见 `cs(1)` 手册页。

dbx 命令

dbx 命令以 dbx 开头。

本地模式语法

<code>dbx options program_name</code>	调试 <code>program_name</code> 。
<code>dbx options program_name core</code>	使用核心文件 <code>core</code> 调试 <code>program_name</code> 。
<code>dbx options program_name process_id</code>	使用进程 ID <code>process_id</code> 调试 <code>program_name</code> 。
<code>dbx options - process_id</code>	调试进程 ID <code>process_id</code> ；dbx 通过 <code>/proc</code> 查找程序。
<code>dbx options - core</code>	使用核心文件 <code>core</code> 调试；另请参见第 300 页的“debug 命令”。
<code>dbx options -r program_name arguments</code>	使用参数 <code>arguments</code> 运行 <code>program_name</code> ；如果异常终止，则开始调试 <code>program_name</code> ，否则退出。

其中：

program_name 是要调试的程序名。

process_id 是正在运行的进程的进程 ID。

arguments 是要传递给程序的参数。

options 是在第 298 页的“选项”中列出的选项。

Java 模式语法

<code>dbx options</code>	调试 <i>program_name</i> 。
<code>program_name{.class .jar}</code>	
<code>dbx options</code>	使用进程 ID <i>process_id</i> 调试 <i>program_name</i> 。
<code>program_name{.class .jar} process_id</code>	
<code>dbx options - process_id</code>	调试进程 ID <i>process_id</i> ；dbx 使用 <code>/proc</code> 查找程序。
<code>dbx options -r</code>	使用参数 <i>arguments</i> 运行 <i>program_name</i> ；如果异常终止，则开始调试 <i>program_name</i> ，否则退出。
<code>program_name{.class .jar} arguments</code>	

其中：

program_name 是要调试的程序名。

process_id 是正在运行的进程的进程 ID。

arguments 是要传递给程序（而不是 JVM 软件）的参数。

options 是在第 298 页的“选项”中列出的选项。

选项

对于本地模式和 Java 模式来说，*options* 有以下选项：

<code>-c commands</code>	在提示输入前执行 <i>commands</i> 。
<code>-C</code>	预加载运行时检查库（请参见第 286 页的“check 命令”）。
<code>-d</code>	与 <code>-s</code> 一起使用，读取后删除 <i>file</i> 。
<code>-e</code>	回显输入命令。
<code>-f</code>	强制加载核心文件，即使它不匹配。
<code>-h</code>	打印 dbx 的使用帮助。

-I <i>dir</i>	添加 <i>dir</i> 到 pathmap 设置（请参见第 335 页的“pathmap 命令”）。
-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	打印 dbx 的自述文件。
-s <i>file</i>	使用 <i>file</i> 代替 <i>/current_directory/.dbxrc</i> 或 <i>\$HOME/.dbxrc</i> 作为启动文件
-S	禁止读取初始化文件 <i>/installation_directory/lib/dbxrc</i> 。
-V	打印 dbx 的版本信息。
-w <i>n</i>	在 where 命令上跳过 <i>n</i> 帧。
-x <i>exec32</i>	运行 32 位 dbx 二进制文件，而不是 64 位 dbx 二进制文件（默认情况下，它在运行 64 位 OS 的系统上运行）。
--	标记选项列表的末尾；如果程序名以短线开头，则使用此选项。

dbxenv 命令

dbxenv 用于列出或设置 dbx 环境变量。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

dbxenv	显示 dbx 环境变量的当前设置。
dbxenv <i>environment_variable</i> <i>setting</i>	将 <i>environment_variable</i> 设置为 <i>setting</i> 。

其中：

environment_variable 是一个 dbx 环境变量。

setting 是此变量的有效设置。

debug 命令

`debug` 命令列出或更改正在被调试的程序。在本地模式中，它加载指定应用程序，然后开始调试该应用程序。在 Java 模式中，它加载指定 Java 应用程序，接着检查类文件是否存在，然后开始调试程序。

本地模式语法

<code>debug</code>	打印被调试程序的名称和参数。
<code>debug program_name</code>	无需进程或核心，开始调试 <code>program_name</code> 。
<code>debug -c core program_name</code>	使用核心文件 <code>core</code> 开始调试 <code>program_name</code> 。
<code>debug -p process_id program_name</code>	使用进程 ID <code>process_id</code> 开始调试 <code>program_name</code> 。
<code>debug program_name core</code>	使用核心文件 <code>core</code> 开始调试 <code>program</code> 。 <code>program_name</code> 可以是 -。 <code>dbx</code> 将尝试从核心文件中提取可执行程序的名称。详细信息请参见第 48 页的“调试核心文件”。
<code>debug program_name process_id</code>	使用进程 ID <code>process_id</code> 开始调试 <code>program_name</code> 。 <code>program_name</code> 可以是 -； <code>dbx</code> 使用 <code>/proc</code> 查找它。
<code>debug -f ...</code>	强制加载核心文件，即使它不匹配。
<code>debug -r ...</code>	<code>-r</code> 选项可使 <code>dbx</code> 保留所有的 <code>display</code> 、 <code>trace</code> 、 <code>when</code> 和 <code>stop</code> 命令。如果没有 <code>-r</code> 选项，则执行隐式 <code>delete all</code> 和 <code>undisplay 0</code> 。
<code>debug -clone ...</code>	<code>-clone</code> 选项可使另一个 <code>dbx</code> 进程开始执行，允许一次调试多个进程。只有在 <code>dbx</code> 调试器下运行才有效。
<code>debug -clone</code>	不调试任何程序，启动另一个 <code>dbx</code> 进程。仅当在 Sun Studio IDE 中运行时才有效。
<code>debug [options] -- program_name</code>	开始调试 <code>program</code> ，即使 <code>program_name</code> 以短线开头。

其中：

`core` 是核心文件的名称。

`options` 是在第 301 页的“选项”中列出的选项。

`pid` 是正在运行的进程的进程 ID。

`program_name` 是程序的路径名。

当用 `debug` 命令加载程序时，泄漏检查和访问检查被关闭。您可以使用 `check` 命令启用它们（请参见第 286 页的“`check` 命令”）。

Java 模式语法

<code>debug</code>	打印被调试程序的名称和参数。
<code>debug program_name{.class .jar}</code>	无需进程，开始调试 <i>program_name</i> 。
<code>debug -p process_id program_name{.class .jar}</code>	使用进程 ID <i>process_id</i> 开始调试 <i>program_name</i> 。
<code>debug program_name{.class .jar} process_id</code>	使用进程 ID <i>process_id</i> 开始调试 <i>program_name</i> 。 <i>program_name</i> 可以是 -； <code>dbx</code> 使用 <code>/proc</code> 查找它。
<code>debug -r ...</code>	<code>-r</code> 选项可使 <code>dbx</code> 保留所有的 <code>display</code> 、 <code>trace</code> 、 <code>when</code> 和 <code>stop</code> 命令。如果没有 <code>-r</code> 选项，则执行隐式 <code>delete all</code> 和 <code>undisplay 0</code> 。
<code>debug -clone ...</code>	<code>-clone</code> 选项可使另一个 <code>dbx</code> 进程开始执行，允许一次调试多个进程。只有运行在 <code>dbx</code> 调试器窗口中才有效。
<code>debug -clone</code>	不调试任何程序，启动另一个 <code>dbx</code> 进程。仅当在 Sun Studio IDE 中运行时才有效。
<code>debug [options] -- program_name{.class .jar}</code>	开始调试 <i>program_name</i> ，即使 <i>program_name</i> 以短线开头。

其中：

file_name 是文件名。

options 是在第 301 页的“选项”中列出的选项。

process_id 是正在运行的进程的进程 ID。

program_name 是程序的路径名。

选项

<code>-c commands</code>	在提示输入前执行 <i>commands</i> 。
<code>-d</code>	与 <code>-s</code> 一起使用，读取后删除 <i>file_name</i> 。
<code>-e</code>	回显输入命令。

-h	打印 dbx 的使用帮助。
-I <i>directory_name</i>	添加 <i>directory_name</i> 到 pathmap 设置（请参见第 335 页的“pathmap 命令”）。
-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	打印 dbx 的自述文件。
-s <i>file</i>	使用 <i>file</i> 代替 <i>/current_directory/.dbxrc</i> 或 <i>\$HOME/.dbxrc</i> 作为启动文件
-S	禁止读取初始化文件 <i>/installation_directory/lib/dbxrc</i> 。
-V	打印 dbx 的版本信息。
-w <i>n</i>	在 where 命令上跳过 <i>n</i> 帧。
--	标记选项列表的末尾；如果程序名以短线开头，则使用此选项。

delete 命令

delete 命令删除断点和其他事件。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

delete [-h] <i>handler_id</i> ...	删除指定 <i>handler_id</i> (s) 的 trace 命令、when 命令或 stop 命令。要删除隐藏的处理程序，必须指定 -h 选项。
delete [-h] 0 all -all	删除所有的 trace 命令、when 命令和 stop 命令，不包括持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 命令将目标进程从 dbx 的控制中释放出来。

本地模式语法

detach	从目标中分离 dbx，并取消所有待决信号。
detach -sig <i>signal</i>	当转发指定的 <i>signal</i> 时分离。
detach -stop	从目标中分离 dbx，并使进程处于停止状态。此选项允许临时使用其他基于 /proc 的调试工具，这些工具可能由于专用访问而被阻止。有关示例，请参见第 89 页的“从进程中分离 dbx”。

其中：

signal 是信号名。

Java 模式语法

detach	从目标中分离 dbx，并取消所有待决信号。
--------	-----------------------

dis 命令

dis 命令反汇编机器指令。仅在本地模式中有效。

语法

dis <i>address</i> [/ <i>count</i>]	反汇编 <i>count</i> 个始于地址 <i>address</i> 的指令（默认值为 10）。
--------------------------------------	---

<code>dis address1, address2</code>	反汇编 <i>address1</i> 到 <i>address2</i> 的指令。
<code>dis</code>	反汇编 10 个指令，始于值 +（请参见第 307 页的“ <code>examine</code> 命令”）。
<code>dis /count</code>	反汇编 <i>count</i> 个指令，始于 +。

其中：

address 是开始反汇编的地址。*address* 的默认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（请参见第 307 页的“`examine` 命令”）。

address1 是开始反汇编的地址。

address2 是停止反汇编的地址。

count 是反汇编的指令个数。*count* 的默认值是 10。

display 命令

在本地模式中，`display` 命令于每个停止点处计算并打印表达式。在 Java 模式中，`display` 命令于每个停止点处计算并打印表达式、局部变量或参数。对象引用扩展为一级，数组被逐条打印出来。

本地模式语法

显示	打印所显示的表达式列表。
<code>display expression, ...</code>	于每个停止点处显示表达式 <i>expression, ...</i> 的值。
<code>display</code> <code>[-r +r -d +d -p +p -L </code> <code>-fformat -Fformat --]</code> <code>expression, ...</code>	请参见第 337 页的“ <code>print</code> 命令”以了解这些标志的含义。

其中：

expression 是一个有效表达式。

format 是用于打印表达式的输出格式。有关有效格式的信息，请参见第 337 页的“`print` 命令”。

Java 模式语法

显示	打印所显示的变量和参数列表。
<code>display expression identifier, ...</code>	于每个停止点处显示变量和参数 <i>identifier, ...</i> 的值。
<code>display [-r +r -d +d -p +p -fformat -Fformat --] expression identifier, ...</code>	请参见第 337 页的“print 命令”以了解这些标志的含义。

其中：

class_name 是 Java 类的名称，可以使用包路径（使用句点 (.) 作为限定符；例如 `test1.extra.T1.Inner`），也可以使用完整路径名（前面加磅符号 (#) 并使用斜线 (/) 和美元符号 (\$) 作为限定符；例如 `#test1/extra/T1$Inner`）。如果使用 \$ 限定符，请用引号将 *class_name* 引起来。

expression 是一个有效的 Java 表达式。

field_name 是类中字段的名称。

format 是用于打印表达式的输出格式。有关有效格式的信息，请参见第 337 页的“print 命令”。

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 对象名。

down 命令

`down` 命令可使调用栈下移（远离 `main`）。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

<code>down</code>	调用栈下移一级。
<code>down number</code>	调用栈下移 <i>number</i> 级。
<code>down -h [number]</code>	下移调用栈，但不跳过隐藏的帧。

其中：

number 是调用栈级数。

dump 命令

`dump` 命令打印某个过程的所有局部变量。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

<code>dump</code>	打印当前过程的所有局部变量。
<code>dump <i>procedure</i></code>	打印 <i>procedure</i> 的所有局部变量。

其中：

procedure 是过程名。

edit 命令

`edit` 命令在一个源文件上调用 `$EDITOR`。仅在本地模式中有效。

如果 `dbx` 没有在 Sun Studio IDE 中运行，则 `edit` 命令使用 `$EDITOR`。否则，它会向 IDE 发送消息以显示相应的文件。

语法

<code>edit</code>	编辑当前文件
<code>edit <i>filename</i></code>	编辑指定文件 <i>filename</i> 。
<code>edit <i>procedure</i></code>	编辑包含函数或过程 <i>procedure</i> 的文件。

其中：

filename 是文件名。

procedure 是一个函数名或过程名。

examine 命令

examine 命令显示内存内容。仅在本地模式中有效。

语法

examine [<i>address</i>]	以 <i>format</i> 格式显示始于 <i>address</i> 的 <i>count</i> 项的内存内容。
[/ [<i>count</i>] [<i>format</i>]]	
examine <i>address1</i> ,	按 <i>format</i> 格式显示 <i>address1</i> 到 <i>address2</i> （首末地址包含在内）的内存内容。
<i>address2</i> [/ [<i>format</i>]]	
examine <i>address</i> =	以指定格式显示地址（而不是地址的内容）。
[<i>format</i>]	<i>address</i> 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 是为 examine 预定义的别名。

其中：

address 是开始显示内存内容的地址。*address* 默认值为最后一个显示其内容的地址之后的地址。此值由 dis 命令共享（请参见第 303 页的“dis 命令”）。

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* 的默认值是 1。

format 是显示内存地址内容的格式。第一个 examine 命令的默认格式是 X（十六进制），后续 examine 命令的默认格式是在前一个 examine 命令中指定的格式。以下为 *format* 的有效值：

d,D	十进制（2 或 4 字节）
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	string
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（4 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

异常命令打印当前 C++ 异常的值。仅在本地模式中有效。

语法

```
exception [-d | +d]    打印当前 C++ 异常的值（如果有）。
```

有关 -d 标志的含义，请参见第 337 页的“print 命令”。

exists 命令

exists 命令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，否则返回 1。

其中：

`name` 是一个符号名。

file 命令

`file` 命令列出或更改当前文件。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

文件 打印当前文件的名称。

`file filename` 更改当前文件。

其中：

`filename` 是文件名。

files 命令

在本地模式中，`files` 命令列出与正规表达式匹配的文件名。在 Java 模式中，`files` 命令列出 `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，除非您已经设置了 `$JAVASRCPATH` 环境变量，否则 `dbx` 不可能找到它们（请参见第 220 页的“指定 Java 源文件的位置”）。

本地模式语法

文件	列出提供当前程序调试信息的所有文件（已使用 <code>-g</code> 编译的文件）的名称。
<code>files</code> <code>regular_expression</code>	列出所有使用 <code>-g</code> 编译，且与指定正则表达式相匹配之文件的名称。

其中：

`regular_expression` 是一个正则表达式。

例如：

```
(dbx) files ^r
myprog:
retregs.cc
reg_sorts.cc
reg_errmsgs.cc
rhosts.cc
```

Java 模式语法

文件	列出 <code>dbx</code> 已知的所有 Java 源文件名称。
----	---------------------------------------

fix 命令

`fix` 命令重新编译修改的源文件并动态地将修改的函数链接至应用程序。仅在本地模式中有效。该功能在 Linux 平台上无效。

语法

<code>fix</code>	修复当前文件。
<code>fix filename filename ...</code>	修复 <code>filename</code> 。
<code>fix -f</code>	强制修复文件，即使源文件未被修改。
<code>fix -a</code>	修复所有已修改的文件。

<code>fix -g</code>	剥离 -O 标志并添加 -g 标志。
<code>fix -c</code>	打印编译行（可能包含某些内部添加以供 <code>dbx</code> 使用的选项）。
<code>fix -n</code>	不执行编译/链接命令（与 <code>-v</code> 一起使用）。
<code>fix -v</code>	冗余模式（覆盖 <code>dbx fix_verbose</code> 环境变量设置）。
<code>fix +v</code>	非冗余模式（覆盖 <code>dbx fix_verbose</code> 环境变量设置）。

fixed 命令

`fixed` 命令列出所有已修复文件的名称。仅在本地模式中有效。

语法

`fixed`

frame 命令

`frame` 命令列出或更改当前栈帧编号。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

<code>frame</code>	显示当前帧的帧号。
<code>frame [-h] <i>number</i></code>	将当前帧设置为帧 <i>number</i> 。
<code>frame [-h] +<i>[number]</i></code>	在栈中上移 <i>number</i> 帧；默认值为 1。
<code>frame [-h] -<i>[number]</i></code>	在栈中下移 <i>number</i> 帧；默认值为 1。
<code>-h</code>	即使帧隐藏，也转至帧。

其中：

number 是调用栈中帧的编号。

func 命令

在本地模式中，`func` 命令列出或更改当前函数。在 Java 模式中，`func` 命令列出或更改当前方法。

本地模式语法

<code>func</code>	打印当前函数的名称。
<code>func procedure</code>	将当前函数更改为函数或过程 <i>procedure</i> 。

其中：

procedure 是一个函数名或过程名。

Java 模式语法

<code>func</code>	打印当前方法的名称。
<code>func [<i>class_name</i>.]<i>method_name</i> [(<i>parameters</i>)]</code>	将当前函数更改为方法 <i>method_name</i> 。

其中：

class_name 是 Java 类的名称，可以使用包路径（使用句点 (.) 作为限定符；例如 `test1.extra.T1.Inner`），也可以使用完整路径名（前面加磅符号 (#) 并使用斜线 (/) 和美元符号 (\$) 作为限定符；例如 `#test1/extra/T1$Inner`）。如果使用 \$ 限定符，请用引号将 *class_name* 引起来。

method_name 是 Java 方法名。

parameters 是方法的参数。

funcs 命令

funcs 命令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	列出当前程序中的所有函数。
funcs [-f <i>filename</i>] [-g] [<i>regular_expression</i>]	如果指定了 -f <i>filename</i> ，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。如果指定了 <i>regular_expression</i> ，则列出与正则表达式匹配的所有函数。

其中：

filename 是您希望列出其所有函数的文件名。

regular_expression 是您希望列出所有与之匹配的函数的正则表达式。

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 命令支持 gdb 命令集。仅在本地模式中有效。

语法

<code>gdb on off</code>	使用 <code>gdb on</code> 进入 gdb 命令模式，在此模式下， <code>dbx</code> 理解并接受 <code>gdb</code> 命令。要退出 <code>gdb</code> 命令模式并返回到 <code>dbx</code> 命令模式，请输入 <code>gdb off</code> 。当在 <code>gdb</code> 命令模式中时，不接受 <code>dbx</code> 命令，反之亦然。在不同的命令模式中保持所有的调试设置（例如：断点）。
---------------------------	---

此版本不支持以下 gdb 命令：

- 命令
- `define`
- `handle`
- `hbreak`
- 中断
- `maintenance`
- `printf`
- `rbreak`
- `return`
- 信号
- `tcatch`
- `until`

handler 命令

`handler` 命令修改事件处理程序（启用、禁用等）。它在本地模式和 Java 模式中具有相同的语法和功能。

对于每个需要在调试会话中管理的事件，创建一个处理程序。命令 `trace`、`stop` 和 `when` 创建处理程序。这些命令中的每一个均会返回一个称为处理程序 ID (`handler_id`) 的数字。`handler`、`status` 和 `delete` 命令以通用方式处理或提供处理程序信息。

语法

<code>handler -enable handler_id ...</code>	启用指定处理程序，要启用所有处理程序，请将 <i>handler_id</i> 指定为 <code>all</code> 。
<code>handler -disable handler_id ...</code>	禁用指定处理程序，要禁用所有处理程序，请将 <i>handler_id</i> 指定为 <code>all</code> 。使用 <code>\$firedhandlers</code> 而不是 <i>handler_id</i> 来禁用导致最近停止的处理程序。
<code>handler -count handler_id</code>	打印指定处理程序的行程计数器的值。
<code>handler -count handler_id newlimit</code>	设置指定事件的新计数限制。
<code>handler -reset handler_id</code>	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` 命令隐藏与某个正规表达式匹配的栈帧。仅在本地模式中有效。

语法

<code>hide</code>	列出当前正在使用的栈帧过滤器。
<code>hide regular_expression</code>	隐藏与 <i>regular_expression</i> 匹配的栈帧。正规表达式与函数名或加载对象的名称匹配，并且是与样式正规表达式匹配的 <code>sh</code> 或 <code>ksh</code> 文件。

其中：

regular_expression 是一个正规表达式。

ignore 命令

ignore 命令告知 dbx 进程不要捕获指定信号。仅在本地模式中有效。

当进程收到这种信号时，忽略信号会导致 dbx 无法停止。

语法

ignore	打印已忽略信号的列表。
ignore <i>number</i> ...	忽略编号为 <i>number</i> 的信号。
ignore <i>signal</i> ...	忽略名为 <i>signal</i> 的信号。无法捕获或忽略 SIGKILL。
ignore \$(catch)	忽略所有信号。

其中：

number 是信号的编号。

signal 是信号名。

import 命令

import 命令从 dbx 命令库中导入命令。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

import <i>pathname</i>	从 dbx 命令库 <i>pathname</i> 中导入命令。
------------------------	----------------------------------

其中：

pathname 是 dbx 命令库的路径名。

intercept 命令

`intercept` 命令抛出指定类型的 (C++ 异常), 仅适用于 C++。仅在本地模式中有效。

当抛出异常的类型与截取列表上的类型匹配时, 除非异常类型也与排除列表上的类型匹配, 否则 `dbx` 将停止。无匹配 `catch` 的抛出异常称为“未处理” `throw`。与抛出函数的异常规范不匹配的抛出异常称为“意外” `throw`。

默认情况下, 截取未处理和意外 `throw`。

语法

<code>intercept <i>intercepted_typename</i> [, <i>intercepted_typename</i> ...]</code>	将 <i>included_typename</i> 类型的 <code>throw</code> 添加到截取列表。
<code>intercept -a[ll]</code>	将所有类型的 <code>throw</code> 添加到截取列表。
<code>intercept -x <i>excluded_typename</i> [, <i>excluded_typename</i> ...]</code>	将 <i>excluded_typename</i> 类型的 <code>throw</code> 添加到排除列表。
<code>intercept -a[ll] -x <i>excluded_typename</i> [, <i>excluded_typename</i> ...]</code>	将除 <i>excluded_typename</i> 之外的所有类型添加到截取列表。
<code>intercept -s[et] [<i>intercepted_typename</i> [, <i>intercepted_typename</i> ...]] [-x <i>excluded_typename</i> [, <i>excluded_typename</i>]]</code>	同时清除截取列表和排除列表, 并将列表设置为仅截取或排除指定类型的 <code>throw</code> 。
<code>intercept</code>	列出被截取的类型。

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的 Java 版本的命令。这将使指定命令应用 Java 表达式算子，并在关联时显示 Java 线程和栈帧。

语法

`java command`

其中：

`command` 是要执行的命令的名称和参数。

jclasses 命令

`jclasses` 命令用于打印 dbx 已知的所有 Java 类名。仅在 Java 模式中有效。

不打印程序中尚未加载的类。

语法

<code>jclasses</code>	打印 dbx 已知的所有 Java 类名。
<code>jclasses -a</code>	打印系统类和其他已知的 Java 类。

joff 命令

`joff` 命令将 dbx 从 Java 模式或 JNI 模式切换到本地模式。

语法

`joff`

jon 命令

jon 命令将 dbx 从本地模式切换到 Java 模式。

语法

jon

jpkgs 命令

jpkgs 命令用于打印 dbx 已知的所有 Java 包名。仅在 Java 模式中有效。

不打印程序中尚未加载的包。

语法

jpkgs

kill 命令

kill 命令将信号发送到进程。仅在本地模式中有效。

语法

kill -l	列出所有已知信号编号、名称和描述。
kill	中止受控进程。
kill <i>job...</i>	向列出的作业发送 SIGTERM 信号。
kill - <i>signal job...</i>	向列出的作业发送指定信号。

其中：

job 可以是进程 ID，或者以以下任一方法指定：

%+	中止当前作业。
%-	中止上一作业。
% 编号	中止编号为 <i>number</i> 的作业。
% <i>string</i>	中止以 <i>string</i> 开头的作业。
%? <i>string</i>	中止包含 <i>string</i> 的作业。

signal 是信号名。

language 命令

language 命令列出或更改当前源语言。仅在本地模式中有效。

语法

语言	打印由 dbx language_mode 环境变量设置的当前语言模式（请参见第 62 页的“设置 dbx 环境变量”）。如果语言模式设置为 autodetect 或 main，该命令还将打印用于分析和计算表达式的当前语言名称。
----	--

其中：

language 可为 c、c++、fortran 或 fortran90。

注 - c 是 ansic 的别名。

line 命令

line 命令列出或更改当前行号。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

line	显示当前行号。
line <i>number</i>	将当前行号设置为 <i>number</i> 。
line " <i>filename</i> "	将当前行号设置为 <i>filename</i> 中的第 1 行。
line " <i>filename</i> ": <i>number</i>	将当前行号设置为 <i>filename</i> 中的第 <i>number</i> 行。

其中：

filename 是要更改行号的文件名。带 "" 的文件名是可选的。

number 是文件中的行号。

示例

```
line 100  
line "/root/test/test.cc":100
```

list 命令

`list` 命令显示源文件的行。它在本地模式和 `Java` 模式中具有相同的语法和功能。

列出的默认行数 `N` 受 `dbx output_list_size` 环境变量控制。

语法

<code>list</code>	列出 <code>N</code> 行。
<code>list number</code>	列出行号为 <code>number</code> 的行。
<code>list +</code>	列出后 <code>N</code> 行。
<code>list +n</code>	列出后 <code>n</code> 行。
<code>list -</code>	列出前 <code>N</code> 行。
<code>list -n</code>	列出前 <code>n</code> 行。
<code>list n1,n2</code>	列出第 <code>n1</code> 行到第 <code>n2</code> 行。
<code>list n1,+</code>	列出第 <code>n1</code> 行到第 <code>n1 + N</code> 行。
<code>list n1,+n2</code>	列出第 <code>n1</code> 行到第 <code>n1 + n2</code> 行。
<code>list n1,-</code>	列出第 <code>n1-N</code> 行到第 <code>n1</code> 行。
<code>list n1,-n2</code>	列出第 <code>n1-n2</code> 行到第 <code>n1</code> 行。
<code>list function</code>	列出 <code>function</code> 源函数的开头。 <code>list function</code> 更改当前范围。更多信息请参见第 72 页的“程序作用域”。
<code>list filename</code>	列出文件 <code>filename</code> 的开头。
<code>list filename:n</code>	从行 <code>n</code> 开始显示文件 <code>filename</code> 。在适当的地方，行号可能为“\$”，表示文件最后一行。逗号是可选的。

其中：

filename 是源代码文件的名称。

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。

选项

<code>-i</code> 或 <code>-instr</code>	混合源代码行和汇编代码
<code>-w</code> 或 <code>-wn</code>	围绕行或函数列出 <i>N</i> （或 <i>n</i> ）行（窗口）。如果使用“+”或“-”语法进行组合，或者指定了两个行号时，不允许使用此选项。

示例

```
list                // 从当前行开始列出 N 行
list +5            // 从当前行开始列出后 5 行
list -             // 列出前 N 行
list -20           // 列出前 20 行
list 1000          // 列出第 1000 行
list 1000,$        // 列出第 1000 行到最后一行
list 2737 +24      // 列出第 2737 行及其后 24 行
list 1000 -20      // 列出第 980 行到第 1000 行
list test.cc:33    // 列出文件 test.cc 中第 33 行
list -w           // 列出当前行周围的 N 行
list -w8 `test.cc`func1 // 列出函数 func1 周围的 8 行
list -i 500 +10    // 列出从第 500 行到第 510 行的源代码和汇编代码
```

listi 命令

listi 命令显示源指令和反汇编指令。仅在本地模式中有效。

详细信息，请参见第 323 页的“list 命令”。

loadobject 命令

loadobject 命令列出和管理来自加载对象的符号信息。仅在本地模式中有效。

语法

loadobject *command_list*

- | | |
|-------------------------------|---|
| -list [<i>regex</i>] [-a] | 显示当前加载的加载对象（请参见第 327 页的“loadobject -list 命令”） |
| -load <i>loadobject</i> | 加载指定加载对象的符号（请参见第 328 页的“loadobject -load 命令”） |
| -unload [<i>regex</i>] | 卸载指定的加载对象（请参见第 328 页的“loadobject -unload 命令”） |
| -hide [<i>regex</i>] | 从 dbx 的搜索算法中删除加载对象（请参见第 327 页的“loadobject -hide 命令”） |
| -use [<i>regex</i>] | 将加载对象添加至 dbx 的搜索算法（请参见第 328 页的“loadobject -use 命令”） |
| -dumpelf [<i>regex</i>] | 显示加载对象的各种 ELF 细节（请参见第 326 页的“loadobject -dumpelf 命令”） |
| -exclude <i>ex-regex</i> | 不自动加载与 <i>ex-regex</i> 匹配的加载对象（请参见第 326 页的“loadobject -exclude 命令”） |
| -exclude | 显示排除模式的列表（请参见第 326 页的“loadobject -exclude 命令”） |
| -exclude -clear | 清除模式的“排除”列表（请参见第 326 页的“loadobject -exclude 命令”） |
-

其中：

regex 是一个正规表达式。如果未加指定，该命令将应用于所有加载对象。

ex-regex 不是可选的，必须指定。

此命令有一个默认别名 `lo`。

loadobject -dumpelf 命令

`loadobject -dumpelf` 命令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 [ regex ]
```

其中：

regex 是一个正规表达式。如果未加指定，该命令将应用于所有加载对象。

此命令转储磁盘上与加载对象文件之 ELF 结构相关的信息。该输出的细节极易被更改。如果要分析此输出，请使用 Solaris OS 命令 `dump` 或 `elfdump`。

loadobject -exclude 命令

`loadobject -exclude` 命令告知 `dbx` 不要自动加载与指定正规表达式匹配的加载对象。

语法

```
loadobject -exclude ex-regex [ -clear ]
```

其中：

ex-regex 是一个正规表达式。

此命令防止 `dbx` 自动加载与指定正规表达式相匹配的加载对象的符号。与其他 `loadobject` 子命令中的 *regex* 不同，如果不指定 *ex-regex*，并非默认为全部。如果不指定 *ex-regex*，该命令列出由前一 `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加载主程序或运行时链接程序。另外，使用它来防止加载 C++ 运行时库可能导致某些 C++ 功能失效。

此选项不能与运行时检查 (RTC) 一起使用。

loadobject -hide 命令

loadobject -hide 命令从 dbx 的搜索算法中删除加载对象。

语法

```
loadobject -hide [ regex ]
```

其中：

regex 是一个正规表达式。如果未加指定，该命令将应用于所有加载对象。

此命令将一个加载对象从程序范围中删除，并在 dbx 中隐藏它的函数和符号。此命令也将重置“预加载”位

loadobject -list 命令

loadobject -list 命令显示当前加载的加载对象。仅在本地模式中有效。

语法

```
loadobject -list [ regex ] [ -a ]
```

其中：

regex 是一个正规表达式。如果未加指定，该命令将应用于所有加载对象。

显示每个加载对象的完整路径名，并在页边显示表示状态的字母。只有指定 -a 选项，才

h	它表示“隐藏”（符号不会被 <code>whatis</code> 或 <code>stop in</code> 之类的符号查询找到）。
u	如果有活动进程，u 表示“已取消映射”。
p	这个字母表示预加载的加载对象，即程序中“loadobject -load”命令或 <code>dlopen</code> 事件的结果。（请参见“ <code>help loadobject preloading</code> ”）

会列出隐藏的加载对象。

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)
```

最后这个示例显示，运行时链接程序的符号默认情况下被隐藏。要在 `dbx` 命令中使用这些符号，请参见下面的“`lo -use`”命令。

loadobject -load 命令

`loadobject -load` 命令加载指定加载对象的符号。仅在本地模式中有效。

语法

```
loadobject -list [ regex ] [ -a ]
```

其中：

loadobject 可以是一个完整路径名，也可以是 `/usr/lib` 或 `/usr/lib/sparcv9` 中的库。如果某个程序正在调试，将仅搜索正确的 ABI 库目录。

loadobject -unload 命令

`loadobject -unload` 命令卸载指定的加载对象。仅在本地模式中有效。

语法

```
loadobject -unload [ regex ]
```

其中：

regex 是一个正规表达式。如果未加指定，该命令将应用于所有加载对象。

此命令卸载与命令行中 *regex* 相匹配的所有加载对象的符号。无法卸载使用 `debug` 命令加载的主程序。`dbx` 也可能拒绝卸载当前正在使用的或者会影响到 `dbx` 正常运行的加载对象。

loadobject -use 命令

`loadobject -use` 命令从 `dbx` 的搜索算法中添加加载对象。仅在本地模式中有效。

语法

```
loadobject -use [ regex ]
```

其中：

regex 是一个正规表达式。如果未加指定，该命令将应用于所有加载对象。

lwp 命令

`lwp` 命令列出或更改当前 LWP（轻量进程）。仅在本地模式中有效。

注 – 只能在 Solaris 平台上执行 `lwp` 命令。

语法

<code>lwp</code>	显示当前 LWP。
<code>lwp <i>lwp_id</i></code>	切换到 LWP <i>lwp_id</i> 。
<code>lwp -info</code>	显示当前 <code>lwp</code> 的名称、位置和屏蔽信号。

其中：

lwp_id 是轻量进程的标识符。

lwps 命令

`lwps` 命令列出进程中的所有 LWP（轻量进程）。仅在本地模式中有效。

注 – 只能在 Solaris 平台上执行 `lwps` 命令。

语法

`lwps`

列出当前进程中所有 LWP

mmapfile 命令

`mmapfile` 命令查看从核心转储中丢失的内存映射文件的内容。仅在本地模式中有效。

Solaris 核心文件不包含只读内存段。可执行只读段（即文本）被自动处理，`dbx` 通过浏览可执行文件和相关共享对象来解决内存访问问题。

语法

`mmapfile` *mmapped_file* 查看从核心转储中丢失的内存映射文件的内容。
address *offset* *length*

其中：

mmapped_file 是核心转储期间映射内存文件的文件名。

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 是对 *mmapped_file* 中的起始地址的字节偏移。

module 命令

module 命令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]	打印当前模块的名称。
module [-f] [-v] [-q] <i>name</i>	读入名为 <i>name</i> 的模块的调试信息。
module [-f] [-v] [-q] -a	读入所有模块的调试信息。

其中：

- name* 是为其读取调试信息的模块名。
- 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更新（使用时需谨慎！）。
- v 指定冗余模式，用于打印语言、文件名等。
- q 指定安静模式。

示例

只读数据段通常在应用程序内存映射某个数据库时出现。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;
```

要通过调试器像访问内存一样访问数据库，您应该输入：

```
mmapfile ../DATABASE ${vaddr} ${offset} ${size}
```

然后可以使用结构方式来查看数据库内容，请输入：

```
print *index
```

modules 命令

modules 命令列出模块名。仅在本地模式中有效。

语法

modules [-v]	列出所有模块
modules [-v] -debug	列出包含调试信息的所有模块。
modules [-v] -read	列出包含已读入的调试信息的模块名称。

其中：

-v 指定冗余模式，用于打印语言、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的本地版本命令。在命令前加上 “native”，可使 dbx 在本地模式中执行该命令。这意味着按 C 表达式或 C++ 表达式的方式解释和显示表达式，而其他命令会产生与 Java 模式不同的输出。

如果在调试 Java 代码的过程中想要检查本地环境，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` 命令单步执行一个源代码行（步过调用）。

`dbx step_events` 环境变量（请参见第 62 页的“设置 `dbx` 环境变量”）控制步骤中是否启用断点。

本地模式语法

<code>next</code>	单步执行一行（步过调用）。对于多线程程序，当步过函数调用时，为了避免死锁，所有 LWP（轻量进程）会在此函数调用期间隐式恢复。非活动线程无法单步执行。
<code>next n</code>	单步执行 <i>n</i> 行（步过调用）。
<code>next ...-sig signal</code>	单步执行时传递指定信号。
<code>next ... thread_id</code>	单步执行指定线程。
<code>next ... lwp_id</code>	单步执行指定 LWP。当步过一个函数时，并不隐式恢复所有 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 是轻量进程 ID。

如果指定显式 *thread_id* 或 *lwp_id*，则通用 `next` 命令的死锁避免措施不起作用。

另请参见第 334 页的“`nexti` 命令”以了解计算机级步过调用。

注 – 有关轻量进程 (LWP) 的信息，请参见 Solaris 《多线程编程指南》。

Java 模式语法

<code>next</code>	单步执行一行（步过调用）。对于多线程程序，当步过函数调用时，为了避免死锁，所有 LWP（轻量进程）会在此函数调用期间隐式恢复。非活动线程无法单步执行。
<code>next n</code>	单步执行 n 行（步过调用）。
<code>next ... thread_id</code>	单步执行指定线程。
<code>next ... lwpid</code>	单步执行指定 LWP。当步过一个函数时，并不隐式恢复所有 LWP。

其中：

n 是要单步执行的行数。

`thread_id` 是线程标识符。

`lwpid` 是 LWP 标识符。

如果指定显式 `thread_id` 或 `lwpid`，则通用 `next` 命令的死锁避免措施不起作用。

注 – 有关轻量进程 (LWP) 的信息，请参见 Solaris 《多线程编程指南》。《多线程编程指南》。

nexti 命令

`nexti` 命令单步执行一个机器指令（步过调用）。仅在本地模式中有效。

语法

<code>nexti</code>	单步执行一个机器指令（步过调用）。
<code>nexti n</code>	单步执行 n 个机器指令（步过调用）。
<code>nexti -sig signal</code>	单步执行时传递指定信号。
<code>nexti ... lwp_id</code>	单步执行指定 LWP。
<code>nexti ... thread_id</code>	单步执行指定线程在其中处于活动状态的 LWP。当步过一个函数时，并不隐式恢复所有 LWP。

其中:

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 是轻量进程 ID。

pathmap 命令

`pathmap` 命令将一个路径名映射到另一个路径名，以便查找源文件等。该映射被应用于源路径、目标文件路径以及当前工作目录（如果指定 `-c`）。`pathmap` 命令在本地模式和 Java 模式中具有相同的语法和功能。

`pathmap` 命令对于处理不同主机上具有不同路径的自动安装或显式 NFS 安装文件系统非常有用。因为 CWD 在自动安装的文件系统上不准确，所以当您解决由于自动安装而引起的问题时，请指定 `-c`。如果原始树或构建树被移动，使用 `pathmap` 命令也非常有用。

默认情况下，`pathmap /tmp_mnt /` 存在。

当 `dbx` 环境变量 `core_lo_pathmap` 设置为 `on` 时，使用 `pathmap` 命令可查找核心文件的加载对象。除此之外，`pathmap` 命令对查找加载对象（共享库）无效。请参见第 49 页的“调试不匹配的核心文件”。

语法

<code>pathmap [-c] [-index]</code>	建立一个从 <i>from</i> 到 <i>to</i> 的新映射。
<code>from to</code>	
<code>pathmap [-c] [-index]</code>	将所有路径映射到 <i>to</i> 。
<code>to</code>	
<code>pathmap</code>	列出所有现存的路径映射（使用索引）
<code>pathmap -s</code>	相同，但 <code>dbx</code> 可以读取输出。
<code>pathmap -d from1</code>	通过路径删除指定映射。
<code>from2...</code>	
<code>pathmap -d index1 index2</code>	通过索引删除指定映射。
<code>...</code>	

其中：

from 和 *to* 是文件路径前缀。*from* 表示被编译成可执行文件或目标文件的文件路径；*to* 表示调试时的文件路径。

from1 是要删除的第一个映射的文件路径。

from2 是要删除的最后一个映射的文件路径。

index 指定索引，映射连同该索引一起插入到列表中。如果不指定索引，映射将被添加到列表结尾。

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*，映射还将应用于当前工作目录。

如果指定 *-s*，则以 *dbx* 可以读取的输出格式列出现存映射。

如果指定 *-d*，则删除指定映射。

示例

```
(dbx) pathmap /export/home/work1 /net/mmm/export/home/work2
# 将 /export/home/work1/abc/test.c 映射至
/net/mmm/export/home/work2/abc/test.c
(dbx) pathmap /export/home/newproject
# 将 /export/home/work1/abc/test.c 映射至
/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 命令从调用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* 编译过的函数的帧。程序计数器将重置为调用点的源代码行开头。您无法弹出在调试器调用的函数之后的帧；请使用 *pop -c*。

通常 *pop* 命令调用与弹出的帧相关联的所有 C++ 析构函数；您可以通过将 *dbx pop_auto_destruct* 环境变量设置为 *off* 来覆盖此行为（请参见第 62 页的“设置 *dbx* 环境变量”）。

语法

<code>pop</code>	从栈中弹出当前顶部帧
<code>pop number</code>	从栈中弹出 <i>number</i> 个帧
<code>pop -f number</code>	从栈中弹出帧，直至到达指定的帧号 <i>number</i>
<code>pop -c</code>	弹出从调试器中进行的最后一个调用。

其中：

number 是从栈中弹出的帧数。

print 命令

在本地模式中，`print` 命令打印表达式的值。在 Java 模式中，`print` 命令打印表达式、局部变量或参数的值。

本地模式语法

<code>print expression, ...</code>	打印表达式 <i>expression, ...</i> 的值
<code>print -r expression</code>	打印表达式 <i>expression</i> 的值，包括其继承成员（仅限于 C++）。
<code>print +r expression</code>	当 <code>dbx output_inherited_members</code> 环境变量为 <code>on</code> 时，不打印继承成员（仅限于 C++）。
<code>print -d [-r] 表达式</code>	显示动态类型而不是静态类型的表达式 <i>expression</i> （仅限于 C++）。
<code>print +d [-r] 表达式</code>	当 <code>dbx output_dynamic_type</code> 环境变量为 <code>on</code> 时，不使用表达式 <i>expression</i> 的动态类型（仅限于 C++）。
<code>print -p expression</code>	调用 <code>prettyprint</code> 函数。
<code>print +p expression</code>	当 <code>dbx output_pretty_print</code> 环境变量为 <code>on</code> 时，不调用 <code>prettyprint</code> 函数。
<code>print -L expression</code>	如果打印对象 <i>expression</i> 大于 4K，则强制打印。
<code>print -l expression</code>	（'Literal'）不打印左侧。如果表达式是一个字符串（ <code>char *</code> ），不打印地址，只打印字符串不带引号的原始字符。
<code>print -fformat expression</code>	使用 <i>format</i> 作为整数、字符串或浮点表达式的格式。

<code>print -Fformat expression</code>	使用指定格式但不打印左侧（变量名或表达式）。
<code>print -o expression</code>	以序数值的形式打印必须作为枚举的 <i>expression</i> 的值。此处也可以使用格式字符串 (<i>-format</i>)。对于非枚举型表达式，忽略此选项。
<code>print -- expression</code>	'--' 代表标志参数的结尾。如果 <i>expression</i> 以加号或减号开头，这会很有用（请参见第 72 页的“程序作用域”以了解范围转换规则。

其中：

expression 是要打印其值的表达式。

format 是用于打印表达式的输出格式。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，并且 `dbx` 会使用其内建打印机制。

允许的格式是 `printf(3S)` 命令所使用的格式的子集。遵循以下约束：

- 没有 `n` 转换。
- 没有将 `*` 用于字段宽度或精度。
- 没有 `%<digits>$` 参数选择。
- 每个格式字符串只有一项转换规范说明。

以下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

<code>print expression, ... identifier, ...</code>	打印表达式 <i>expression, ...</i> 或标识符 <i>identifier, ...</i> 的值
<code>print -r expression identifier</code>	打印 <i>expression</i> 或 <i>identifier</i> 的值, 包括其继承成员。
<code>print +r expression identifier</code>	当 <code>dbx output_inherited_members</code> 环境变量为 <code>on</code> 时, 不打印继承成员。
<code>print -d [-r] expression identifier</code>	显示动态类型而不是静态类型的 <i>expression</i> 或 <i>identifier</i> 。
<code>print +d [-r] expression identifier</code>	当 <code>dbx output_dynamic_type</code> 环境变量为 <code>on</code> 时, 不使用 <i>expression</i> 或 <i>identifier</i> 的动态类型。
<code>print -- expression identifier</code>	'-' 代表标志参数的结尾。如果 <i>expression</i> 以加号或减号开头, 这会很有用 (请参见《使用 <code>dbx</code> 调试程序》中的“范围”, 以了解范围转换规则。

其中:

class_name 是 Java 类的名称, 可以使用包路径 (使用句点 (.) 作为限定符; 例如 `test1.extra.T1.Inner`), 也可以使用完整路径名 (前面加磅符号 (#) 并使用斜线 (/) 和美元符号 (\$) 作为限定符; 例如 `#test1/extra/T1$Inner`)。如果使用 \$ 限定符, 请用引号将 *class_name* 引起来。

expression 是要打印其值的 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 对象名。

proc 命令

proc 命令显示当前进程的状态。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

proc -map	显示带地址的加载对象列表
proc -pid	显示当前进程 ID (pid)

prog 命令

prog 命令管理被调试的程序及其属性。它在本地模式和 Java 模式中具有相同的语法和功能。

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	打印可执行文件的完整路径, 如果程序需要使用 -, 则打印 -。
prog -argv	打印全部 argv, 包括 argv[0]。
prog -args	打印 argv, 不包括 argv[0]。
prog -stdin	打印 < filename ; 如果使用 stdin, 则为空。
prog -stdout	打印 > filename 或 >> filename ; 如果使用 stdout, 则为空。设计 -args、-stdin、-stdout 输出的目的在于: 字符串可合并以及通过 run 命令重新使用 (请参见第 345 页的 “run 命令”)。

quit 命令

quit 命令退出 dbx。它在本地模式和 Java 模式中具有相同的语法和功能。

如果 dbx 连接到某个进程，该进程在退出之前被分离。如果有待决信号，它们将被取消。使用 detach 命令进行严格控制（请参见第 303 页的“detach 命令”）。

语法

quit	退出 dbx 并返回代码 0。与 <i>exit</i> 相同。
quit <i>n</i>	退出并返回代码 <i>n</i> 。与 <i>exit n</i> 相同。

其中：

n 是返回代码。

regs 命令

regs 命令打印寄存器的当前值。仅在本地模式中有效。

语法

regs [-f] [-F]

其中：

-f 包括浮点寄存器（单精度）（仅限于 SPARC 平台）

-F 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
```

```
当前线程: t@1
```

```
当前帧: [1]
```

```
g0-g3          0x00000000 0x0011d000 0x00000000 0x00000000
```

```

g4-g7          0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3          0x00000003 0x00000014 0xef7562b4 0xffff420
o4-o7          0xef752f80 0x00000003 0xffff3d8 0x000109b8
l0-l3          0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7          0xffff438 0x00000001 0x00000007 0xef74df54
i0-i3          0x00000001 0xffff4a4 0xffff4ac 0x00020c00
i4-i7          0x00000001 0x00000000 0xffff440 0x000108c4
y              0x00000000
psr            0x40400086
pc             0x000109c0:main+0x4mov      0x5, %l0
npc            0x000109c4:main+0x8st      %l0, [%fp - 0x8]
f0f1          +0.0000000000000000e+00
f2f3          +0.0000000000000000e+00
f4f5          +0.0000000000000000e+00
f6f7          +0.0000000000000000e+00

```

replay 命令

replay 命令重新运行自上次执行 run、rerun 或 debug 命令之后的调试命令。仅在本地模式中有效。

语法

```
replay [-number] 重新运行自上次执行 run 命令、rerun 命令或 debug 命令之后
                  的所有命令，或所有命令减去 number 个命令。
```

其中：

number 是不重新运行的命令个数。

rerun 命令

rerun 命令可以不带参数运行程序。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

rerun	不带参数开始执行程序
rerun <i>arguments</i>	通过 save 命令使用新参数开始执行程序（请参见第 346 页的“save 命令”）。

restore 命令

restore 命令将 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 命令使用 shell 引用规则打印表达式。仅在本地模式中有效。

语法

<code>rprint</code>	打印表达式的值。未应用特殊引用规则，因此 <code>rprint a ></code>
<code>[-r +r -d +d -p +p -L </code>	<code>b</code> 将 <code>a</code> 值（如果存在）放入 <code>b</code> 文件中（请参见第 337 页的
<code>-l -fformat -Fformat --]</code>	“ <code>print</code> 命令”以了解标志的含义）。

`expression`

其中：

`expression` 是要打印其值的表达式。

`format` 是用于打印表达式的输出格式。有关有效格式的信息，请参见第 337 页的“`print` 命令”。

rtc -showmap 命令

`rtc -showmap` 命令报告按工具类型（分支和捕获）分类的程序文本的地址范围。仅在本地模式中有效。

注 – 只能在 Solaris 平台上执行 `rtc -showmap` 命令。

语法

`rtc -showmap`

此命令供专业用户使用，用于内部调试 `dbx`。运行时检查为访问检查提供程序文本。视可用资源而定，该工具类型可以是分支或捕获指令。`rtc -showmap` 命令报告按工具类型分类的程序文本的地址范围。此映射可用于查找添加补丁区目标文件的最优位置，并避免自动使用捕获。详细信息，请参见第 152 页的“运行时检查的 8 M 字节限制”。

run 命令

run 命令使用参数运行程序。

使用 Control-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 <i>arguments</i>	使用新参数开始执行程序。
run ...> >> <i>input_file</i>	设置输出重定向。
run ... < <i>output_file</i>	设置输入重定向。

其中：

arguments 是运行目标进程时所要使用的参数。

input_file 是要将输入重定向自该文件的文件名。

output_file 是要将输出重定向至该文件的文件名。

注 – 目前无法使用 run 或 runargs 命令来重定向 stderr。

Java 模式语法

run	使用当前参数开始执行程序
run <i>arguments</i>	使用新参数开始执行程序。

其中：

arguments 是运行目标进程时所要使用的参数。这些参数被传递给 Java 应用程序，而不是 JVM 软件。不要把主类名当作参数。

无法使用 run 命令来重定向 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 命令更改目标进程的参数。它在本地模式和 Java 模式中具有相同的语法和功能。

使用不带参数的 debug 命令（请参见第 300 页的“debug 命令”）检查目标进程的当前参数。

语法

runargs <i>arguments</i>	设置当前参数，由运行命令使用（请参见第 345 页的“run 命令”）。
runargs ...> >> <i>file</i>	设置由 run 命令使用的输出重定向。
runargs ... < <i>file</i>	设置由 run 命令使用的输入重定向。
runargs	清除当前参数。

其中：

arguments 是运行目标进程时所要使用的参数。

file 是来自目标进程的输出或向目标进程的输入被重定向到的文件。

save 命令

save 命令将命令保存到一个文件中。仅在本地模式中有效。

语法

save [- <i>number</i>] [<i>filename</i>]	将自上次执行 run 命令、rerun 命令或 debug 命令之后的所有命令，或所有命令减去 <i>number</i> 个命令保存到默认文件或 <i>filename</i> 中。
--	--

其中：

number 是不保存的命令个数。

filename 是保存自上次执行 run、rerun 或 debug 命令之后执行的 dbx 命令的文件名。

scopes 命令

scopes 命令打印活动范围列表。仅在本地模式中有效。

语法

scopes

search 命令

search 命令在当前源文件中向前搜索。仅在本地模式中有效

语法

search <i>string</i>	在当前文件中向前搜索 <i>string</i> 。
search	使用上一搜索字符串重复搜索

其中：

string 是要搜索的字符串。

showblock 命令

showblock 命令显示自运行时进程分配特定堆块的位置。仅在本地模式中有效。

注 – 只能在 Solaris 平台上执行 showblock 命令。

打开内存使用检查或内存泄漏检查时，showblock 命令会显示指定地址堆块的详情。详情包括块分配的位置及其大小。请参见第 286 页的“check 命令”。

语法

```
showblock -a address
```

其中：

address 是堆块的地址。

showleaks 命令

`showleaks` 命令报告自上次执行 `showleaks` 命令以来的新内存泄漏。仅在本地模式中有效。

注 – 只能在 Solaris 平台上执行 `showleaks` 命令。

在默认的非冗余情况下，将打印每个泄漏记录的单行报告。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 显示所有已经产生的泄漏（不仅仅是上次执行 `showleaks` 命令之后的泄漏）。
- m *m* 合并泄漏；如果两个或更多泄漏分配时的调用栈与 *m* 帧匹配，则这些泄漏将记录在单个合并泄漏报告中。如果指定 -m 选项，它会覆盖使用 `check` 命令指定的 *m* 的全局值（请参见第 286 页的“`check` 命令”）。
- n *number* 表示在报告中最多显示 *number* 个记录。默认值为显示所有记录。
- v 生成冗余输出。默认值为显示非冗余输出。

showmemuse 命令

showmemuse 命令显示自上次执行 showmemuse 命令以来所使用的内存。仅在本地模式中有效。

注 – 只能在 Solaris 平台上执行 showmemuse 命令。

每条“使用的块”记录打印一个单行报告。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（请参见第 348 页的“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 命令之后的块）。

- m *m* 合并使用的块报告。*m* 的默认值是 2 或上次使用 check 命令（请参见第 286 页的“check 命令”）指定的全局值。如果两个或更多块分配时的调用栈与 *m* 帧匹配，则这些块将记录在单个合并报告中。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。默认值为 20。-v 生成冗余输出。默认值为显示非冗余输出。

source 命令

source 命令在指定文件中执行命令。仅在本地模式中有效。

语法

```
source filename           在文件 filename 中执行命令。未搜索到 $PATH。
```

status 命令

`status` 命令列出事件处理程序（断点等）。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

<code>status</code>	打印正在使用的 <code>trace</code> 、 <code>when</code> 和 <code>stop</code> 断点。
<code>status handler_id</code>	打印处理程序 <code>handler_id</code> 的状态。
<code>status -h</code>	输出正在使用的 <code>trace</code> 、 <code>when</code> 和 <code>stop</code> 断点，包括隐藏部分。
<code>status -s</code>	相同，但 <code>dbx</code> 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` 命令单步执行一个源代码行或语句（步入使用 `-g` 选项编译的调用）。

`dbx step_events` 环境变量控制步骤中是否启用断点。

`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 `dbx` 在检测到即将发生“异常”控制流更改时是否停止。对 `siglongjmp()` 或 `longjmp()` 的调用或异常抛出可以导致这样的控制流更改。

本地模式语法

<code>step</code>	单步执行一行（步入调用）。对于多线程程序，当步过函数调用时，为了避免死锁，所有线程会在此函数调用期间隐式恢复。非活动线程无法单步执行。
<code>step n</code>	单步执行 n 行（步入调用）。
<code>step up</code>	向下单步执行并步出当前函数。
<code>step ...-sig signal</code>	单步执行时传递指定信号。如果存在该信号的信号处理程序，而且信号处理器是使用 <code>-g</code> 选项编译的，则步入它。
<code>step ... thread_id</code>	单步执行指定线程。不适用于 <code>step up</code> 。
<code>step ... lwp_id</code>	单步执行指定 LWP。当步过一个函数时，并不隐式恢复所有 LWP。
<code>step to [function]</code>	试图步入从当前源代码行调用的 <i>function</i> 。如果未指定 <i>function</i> ，则试图步入 <code>last</code> 函数，以帮助避免 <code>step</code> 命令和 <code>step up</code> 命令的长序列。 <code>last</code> 函数的示例如下： <pre>f()->s()-t()->last(); last(a() + b(c()->d()));</pre>

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 是轻量进程 ID。

function 是函数名。

仅当指定显式 *lwp_id* 时，通用 `step` 命令的死锁避免措施才不起作用。

执行 `step to` 命令时，在当前源代码行中尝试步入最后一个汇编调用指令或步入函数（如果指定）时，可能会因为条件转移而无法执行调用。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。

另请参见第 352 页的“`stepi` 命令”以了解机器级单步执行。

Java 模式语法

<code>step</code>	单步执行一行（步入调用）。对于多线程程序，当步过方法调用时，为了避免死锁，所有线程会在该方法调用期间隐式恢复。非活动线程无法单步执行。
<code>step n</code>	单步执行 n 行（步入调用）。
<code>step up</code>	向下单步执行并步出当前方法。
<code>step ... tid</code>	单步执行指定线程。不适用于 <code>step up</code> 。
<code>step ... lwpid</code>	单步执行指定 LWP。当步过一个方法时，并不隐式恢复所有 LWP。

stepi 命令

`stepi` 命令单步执行一个机器指令（步入调用）。仅在本地模式中有效。

语法

<code>stepi</code>	单步执行一个机器指令（步入调用）。
<code>stepi n</code>	单步执行 n 个机器指令（步入调用）。
<code>stepi -sig signal</code>	单步执行并传递指定信号。
<code>stepi ... lwp_id</code>	单步执行指定 LWP。
<code>stepi ... thread_id</code>	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` 是轻量进程 ID。

`thread_id` 是线程 ID。

stop 命令

stop 命令设置源码级断点。

语法

stop 命令有以下一般语法：

```
stop event-specification [ modifier ]
```

发生指定事件时，停止进程。

本地模式语法

以下特定语法是在本地模式下有效的一些更重要的语法。有关其他事件，请参见第 261 页的“设置事件规范”。

stop [-update]	立刻停止执行。仅在 when 命令体内有效。
stop -noupdate	立即停止执行，但不更新 Sun Studio IDE 调试器窗口。
stop access <i>mode</i> <i>address_expression</i> [<i>byte_size_expression</i>]	在访问由 <i>address_expression</i> 指定的内存后停止执行。另请参见第 100 页的“地址被访问时停止执行”。
stop at <i>line_number</i>	在 <i>line_number</i> 处停止执行。另请参见第 96 页的“在源代码行设置 stop 断点”。
stop change <i>variable</i>	当 <i>variable</i> 的值发生改变时停止执行。
stop cond <i>condition_expression</i>	当 <i>condition_expression</i> 表示的条件求值为真时停止执行。
stop in <i>function</i>	调用 <i>function</i> 时停止执行。另请参见第 97 页的“在函数中设置 stop 断点”。
stop inclass <i>class_name</i> [-recurse -norecurse]	仅限于 C++：在类、结构、联合或模板类的所有成员函数上设置断点。-norecurse 是默认值。如果指定 -recurse，则基类被包括进来。另请参见第 98 页的“在相同类的成员函数中设置断点”。

<code>stop infunction <i>name</i></code>	仅限于 C++：在所有非成员函数 <i>name</i> 上设置断点。
<code>stop inmember <i>name</i></code>	仅限于 C++：在所有成员函数 <i>name</i> 上设置断点。请参见第 98 页的“在不同类的成员函数中设置断点”。
<code>stop inobject <i>object_expression</i> [-recurse -norecurse]</code>	仅限于 C++：从对象 <i>object_expression</i> 调用时，在任何非静态类方法及其所有基类的入口处设置断点。-recurse 是默认值。如果指定 -norecurse，则不包括基类。另请参见第 99 页的“在对象中设置断点”。

其中：

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（默认值）。
b	访问前停止进程。

name 是 C++ 函数名。

object_expression 标识 C++ 对象。

variable 是变量名。

以下修饰符在本地模式中有效。

<code>-if <i>condition_expression</i> on</code>	只有在 <i>condition_expression</i> 求值为真时，指定的事件才会发生。
<code>-in <i>function</i></code>	仅当指定的事件在 <i>function</i> 的范围内发生时，才会停止执行。
<code>-count <i>number</i></code>	从 0 开始，每次发生事件时，计数器的计数便会增加。当达到 <i>number</i> 时，执行停止，计数器重置为 0。
<code>-count infinity</code>	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。
<code>-temp</code>	创建一个临时断点，当事件发生时即删除该断点。
<code>-disable</code>	创建禁用状态的断点。
<code>-instr</code>	执行指令级变量。例如， <code>step</code> 变为指令级单步执行，而 <code>at</code> 使用参数的文本地址而不是行号。
<code>-perm</code>	使此事件在 <code>debug</code> 中保持持久性。某些事件（例如断点）不适宜保持持久性。 <code>delete all</code> 不会删除持久性处理程序，应使用 <code>delete hid</code> 。
<code>-hidden</code>	通过 <code>status</code> 命令隐藏事件。一些导入模块可能会选择使用它。可使用 <code>status -h</code> 查看它们。
<code>-lwp <i>lwpid</i></code>	只有当指定的事件在给定的 LWP 中发生时，执行才会停止。
<code>-thread <i>tid</i></code>	只有当指定的事件在给定的线程中发生时，执行才会停止。

Java 模式语法

以下特定语法在 Java 模式中有效。

<code>stop access mode <i>class_name.field_name</i></code>	在访问由 <i>class_name.field_name</i> 指定的内存后停止执行。
<code>stop at <i>line_number</i></code>	在 <i>line_number</i> 处停止执行。
<code>stop at <i>file_name:line_number</i></code>	在 <i>file_name</i> 中的 <i>line_number</i> 处停止执行。
<code>stop change <i>class_name.field_name</i></code>	当 <i>class_name</i> 中的 <i>field_name</i> 值发生更改时停止执行。
<code>stop classload</code>	在加载任何类时停止执行。

<code>stop classload <i>class_name</i></code>	在加载 <i>class_name</i> 类时停止执行。
<code>stop classunload</code>	在卸载任何类时停止执行。
<code>stop classunload <i>class_name</i></code>	在卸载 <i>class_name</i> 类时停止执行。
<code>stop cond <i>condition_expression</i></code>	当 <i>condition_expression</i> 表示的条件求值为真时停止执行。
<code>stop in <i>class_name.method_name</i></code>	当已输入 <i>class_name.method_name</i> 并即将执行第一行时停止执行。如果未指定参数且方法是重载的，则会显示方法列表。
<code>stop in <i>class_name.method_name</i> ([<i>parameters</i>])</code>	当已输入 <i>class_name.method_name</i> 并即将执行第一行时停止执行。
<code>stop inmethod <i>class_name.method_name</i></code>	在所有非成员方法 <i>class_name.method_name</i> 上设置断点。
<code>stop inmethod <i>class_name.method_name</i> ([<i>parameters</i>])</code>	在所有非成员方法 <i>class_name.method_name</i> 上设置断点。
<code>stop throw</code>	在抛出一个 Java 异常后停止执行。
<code>stop throw <i>类型</i></code>	在抛出一个 <i>type</i> 类型的 Java 异常后停止执行。

其中：

class_name 是 Java 类的名称，可以使用包路径（使用句点 (.) 作为限定符；例如 `test1.extra.T1.Inner`），也可以使用完整路径名（前面加磅符号 (#) 并使用斜线 (/) 和美元符号 (\$) 作为限定符；例如 `#test1/extra/T1$Inner`）。如果使用 \$ 限定符，请用引号将 *class_name* 引起来。

condition_expression 可以是任何表达式，但其求得的值必须为整型。

field_name 是类中字段的名称。

file_name 是文件名。

line_number 是源代码行的编号。

method_name 是 Java 方法名。

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。类型可以是 `-unhandled` 或 `-unexpected`。

以下修饰符在 Java 模式中有效：

<code>-if condition_expression</code>	只有在 <i>condition_expression</i> 求值为真时，指定的事件才会发生。
<code>-count number</code>	从 0 开始，每次发生事件时，计数器的计数便会增加。当达到 <i>number</i> 时，执行停止，计数器重置为 0。
<code>-count infinity</code>	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。
<code>-temp</code>	创建一个临时断点，当事件发生时即删除该断点。
<code>-disable</code>	创建禁用状态的断点。

有关设置计算机级断点的信息，另请参见第 358 页的“`stopi` 命令”。

有关所有事件的列表和语法，请参见第 261 页的“设置事件规范”。

stopi 命令

stopi 命令设置计算机级断点。仅在本地模式中有效。

语法

stopi 命令有以下一般语法：

```
stopi event-specification [ modifier ]
```

发生指定事件时，停止进程。

以下特定语法有效：

```
stopi at address                在 address 位置处停止执行。
```

```
stopi in function              调用 function 时停止执行。
```

其中：

address 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，请参见第 261 页的“设置事件规范”。

suppress 命令

suppress 命令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。

注 – 只能在 Solaris 平台上执行 suppress 命令。

如果 dbx rtc_auto_suppress 环境变量为 on，则只报告一次指定位置的内存错误。

语法

禁止	<code>suppress</code> 和 <code>unsuppress</code> 命令的历史记录（不包括指定 <code>-d</code> 和 <code>-reset</code> 选项的那些命令）。
<code>suppress -d</code>	未被编译用于调试的函数中禁止的错误列表（默认禁止）。每个加载对象都有这样一个列表。只能通过使用带 <code>-d</code> 选项的启用命令来启用这些错误。
<code>suppress -d errors</code>	通过进一步禁止 <i>errors</i> 来修改所有加载对象的默认禁止。
<code>suppress -d errors in loadobjects</code>	通过进一步禁止 <i>errors</i> 来修改 <i>loadobjects</i> 中的默认禁止。
<code>suppress -last</code>	在错误位置禁止当前错误。
<code>suppress -reset</code>	将默认禁止设置为初始值（启动时）
<code>suppress -r <id> ...</code>	删除 <i>id</i> 指定的启用事件， <i>id</i> 可使用 <code>unsuppress</code> 命令获得（请参见第 372 页的“ <code>unsuppress</code> 命令”）。
<code>suppress -r 0 all -all</code>	删除所有由 <code>unsuppress</code> 命令指定的启用事件（请参见第 372 页的“ <code>unsuppress</code> 命令”）
<code>suppress errors</code>	禁止任何地方的 <i>errors</i>
<code>suppress errors in [functions] [files] [loadobjects]</code>	禁止 <i>functions</i> 列表、 <i>files</i> 列表和 <i>loadobjects</i> 列表中的 <i>errors</i> 。
<code>suppress errors at line</code>	禁止 <i>line</i> 处的 <i>errors</i> 。
<code>suppress errors at "file":line</code>	禁止 <i>file</i> 中 <i>line</i> 处的 <i>errors</i> 。
<code>suppress errors addr address</code>	禁止 <i>address</i> 位置的 <i>errors</i> 。

其中：

address 是内存地址。

errors 由空格分隔，可以是以下项目的任一组合：

<i>all</i>	所有错误
<i>aib</i>	可能的内存泄漏 - 地址位于块内
<i>air</i>	可能的内存泄漏 - 地址位于寄存器内
<i>baf</i>	错误释放
<i>duf</i>	重复释放
<i>mel</i>	内存泄漏
<i>maf</i>	未对齐释放
<i>mar</i>	未对齐读
<i>maw</i>	未对齐写
<i>oom</i>	内存不足
<i>rua</i>	从未分配的内存中读
<i>rui</i>	从未初始化的内存中读
<i>wro</i>	写入到只读内存
<i>wua</i>	写入到未分配内存
<i>biu</i>	使用的块（分配的内存）。尽管不是错误，您可以在禁止命令中使用 <i>biu</i> ，就像使用 <i>errors</i> 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。

loadobjects 是一个或多个加载对象名。

有关禁止错误的详细信息，请参见第 140 页的“禁止错误”。

有关启用错误的信息，请参见第 372 页的“*unsuppress* 命令”。

sync 命令

sync 命令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 Solaris 平台上执行 sync 命令。

语法

sync -info *address* 显示有关指定地址的同步对象的信息。

其中：

address 是同步对象的地址。

syncs 命令

syncs 命令列出所有同步对象（锁）。仅在本地模式中有效。

注 - 只能在 Solaris 平台上执行 syncs 命令。

语法

syncs

thread 命令

thread 命令列出或更改当前线程。

本地模式语法

线程	显示当前线程
<code>thread <i>thread_id</i></code>	切换到线程 <i>thread_id</i> 。

以下变量中，如果没有 *thread_id*，则表示当前线程。

<code>thread -info [<i>thread_id</i>]</code>	打印有关指定线程的所有信息。
<code>thread -hide [<i>thread_id</i>]</code>	隐藏指定（或当前）线程。它将不会显示在通用线程列表中。
<code>thread -unhide [<i>tid</i>]</code>	取消隐藏指定（或当前）线程。
<code>thread -unhide all</code>	取消隐藏所有线程。
<code>thread -suspend <i>thread_id</i></code>	阻止指定线程运行。挂起的线程在线程列表中标有一个“S”。
<code>thread -resume <i>thread_id</i></code>	撤消 <code>-suspend</code> 执行的操作。
<code>thread -blocks [<i>thread_id</i>]</code>	列出阻塞其他线程的指定线程所控制的所有锁。
<code>thread -blockedby [<i>thread_id</i>]</code>	显示指定线程被哪个同步对象阻塞（如果有）。

其中：

thread_id 是线程 ID。

Java 模式语法

线程	显示当前线程
<code>thread <i>thread_id</i></code>	切换到线程 <i>thread_id</i> 。

以下变量中，如果没有 *thread_id*，则表示当前线程。

<code>thread -info [<i>thread_id</i>]</code>	打印有关指定线程的所有信息。
<code>thread -hide [<i>thread_id</i>]</code>	隐藏指定（或当前）线程。它将不会显示在通用线程列表中。
<code>thread -unhide [<i>thread_id</i>]</code>	取消隐藏指定（或当前）线程。

<code>thread -unhide all</code>	取消隐藏所有线程。
<code>thread -suspend <i>thread_id</i></code>	阻止指定线程运行。挂起的线程在线程列表中标有一个“S”。
<code>thread -resume <i>thread_id</i></code>	撤消 <code>-suspend</code> 执行的操作。
<code>thread -blocks [<i>thread_id</i>]</code>	列出 <i>thread_id</i> 占用的 Java 监视器。
<code>thread -blockedby [<i>thread_id</i>]</code>	列出阻塞 <i>thread_id</i> 的 Java 监视器。

其中：

thread_id 是具有 `t@number` 格式的 dbx 样式线程 ID 或为线程指定的 Java 线程名。

threads 命令

`threads` 命令列出所有线程。

本地模式语法

线程	打印所有已知线程列表。
<code>threads -all</code>	打印通常不输出的线程（僵停）。
<code>threads -mode all filter</code>	控制是打印所有线程还是过滤线程。默认值为过滤线程。启用过滤时，不会列出已使用 <code>thread -hide</code> 命令隐藏的线程。
<code>threads -mode auto manual</code>	在 dbx 调试器下，启用自动更新线程列表。
<code>threads -mode</code>	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@num` 指特定线程。其中的 *number* 为 `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），或者 `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，可为以下之一：
 - 监视
 - 运行
 - 休眠
 - 等待
 - 未知
 - 僵停
- 线程当前执行的函数。

Java 模式语法

线程	打印所有已知线程列表。
<code>threads -all</code>	打印通常不输出的线程（僵停）。
<code>threads -mode all filter</code>	控制是打印所有线程还是过滤线程。默认值为过滤线程。
<code>threads -mode auto manual</code>	在 dbx 调试器下，启用自动更新线程列表。
<code>threads -mode</code>	回显当前模式

列表中的每行信息由以下内容组成：

- >（箭头）表示为当前线程
- `t@number`，一种 dbx 类型线程 ID
- 线程状态，可为以下之一：
 - 监视
 - 运行
 - 休眠
 - 等待
 - 未知
 - 僵停
- 加单引号的线程名
- 说明线程优先级的数字

trace 命令

跟踪命令显示执行的源代码行、函数调用或变量更改。

使用 `dbx trace_speed` 环境变量可设置跟踪速度。

如果 `dbx` 处于 Java 模式，而您想在本机代码中设置 `trace` 断点，请使用 `joff` 命令（请参见第 319 页的“`joff` 命令”）将其切换到“本地”模式，或在 `trace` 命令前添加前缀 `native`（请参见第 332 页的“`native` 命令”）。

如果 `dbx` 处于 JNI 模式，而您想在 Java 代码中设置 `trace` 断点，请在 `trace` 命令前添加前缀 `java`（请参见第 319 页的“`java` 命令”）。

语法

跟踪命令有以下一般语法：

```
trace event-specification [ modifier]
```

发生指定事件时，跟踪即被打印。

本地模式语法

以下特定语法在本地模式中有效：

<code>trace -file file_name</code>	将所有跟踪输出指定给 <code>file_name</code> 。要将跟踪输出还原为标准输出，请将 <code>-</code> 用于 <code>file_name</code> 。跟踪输出始终被添加到 <code>file_name</code> 中。并在显示 <code>dbx</code> 提示和退出应用程序时刷新。 <code>filename</code> 在新运行或连接后继续运行时总会被重新打开。
<code>trace step</code>	跟踪每个源代码行、函数调用并返回。
<code>trace next -in 函数</code>	在指定 <code>function</code> 中跟踪每个源代码行
<code>trace at line_number</code>	跟踪指定源码 <code>line</code> 。
<code>trace in function</code>	跟踪对指定 <code>function</code> 的调用和返回。
<code>trace inmember 函数</code>	跟踪对所有名为 <code>function</code> 的成员函数的调用。

<code>trace infunction <i>function</i></code>	调用任何名为 <i>function</i> 的函数时进行跟踪。
<code>trace inclass <i>class</i></code>	跟踪对所有 <i>class</i> 的成员函数的调用。
<code>trace change <i>variable</i></code>	跟踪对 <i>variable</i> 的更改。

其中：

file_name 是跟踪输出所要发送到的文件名。

function 是函数名。

line_number 是源代码行的编号。

class 是类名。

variable 是变量名。

以下修饰符在本地模式中有效。

<code>-if <i>condition_expression</i></code>	只有在 <i>condition_expression</i> 求值为真时，指定的事件才会发生。
<code>-in <i>function</i></code>	只有当指定的事件在 <i>function</i> 中发生时，执行才会停止。
<code>-count <i>number</i></code>	从 0 开始，每次发生事件时，计数器的计数便会增加。当达到 <i>number</i> 时，执行停止，计数器重置为 0。
<code>-count infinity</code>	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。
<code>-temp</code>	创建一个临时断点，当事件发生时即删除该断点。
<code>-disable</code>	创建禁用状态的断点。
<code>-instr</code>	执行指令级变量。例如， <code>step</code> 变为指令级单步执行，而 <code>at</code> 使用参数的文本地址而不是行号。
<code>-perm</code>	使此事件在 <code>debug</code> 中保持持久性。某些事件（例如断点）不适宜保持持久性。 <code>delete all</code> 不会删除持久性处理程序，应使用 <code>delete hid</code> 。
<code>-hidden</code>	通过 <code>status</code> 命令隐藏事件。一些导入模块可能会选择使用它。可使用 <code>status -h</code> 查看它们。
<code>-lwp <i>lwpid</i></code>	只有当指定的事件在给定的 LWP 中发生时，执行才会停止。
<code>-thread <i>thread_id</i></code>	只有当指定的事件在给定的线程中发生时，执行才会停止。

Java 模式语法

以下特定语法在 Java 模式中有效。

<code>trace -file <i>file_name</i></code>	将所有跟踪输出指定给 <i>file_name</i> 。要将跟踪输出还原为标准输出，请将 - 用于 <i>file_name</i> 。跟踪输出始终被添加到 <i>file_name</i> 中。并在显示 dbx 提示和退出应用程序时刷新。 <i>file_name</i> 在新运行或连接后继续运行时总会被重新打开。
<code>trace at <i>line_number</i></code>	跟踪 <i>line_number</i> 。
<code>trace at <i>file_name.line_number</i></code>	跟踪指定源码 <i>file_name.line_number</i> 。
<code>trace in <i>class_name.method_name</i></code>	跟踪对 <i>class_name.method_name</i> 的调用和返回。
<code>trace in <i>class_name.method_name</i> ([<i>parameters</i>])</code>	跟踪对 <i>class_name.method_name</i> ([<i>parameters</i>]) 的调用和返回。
<code>trace inmethod <i>class_name.method_name</i></code>	跟踪对所有名为 <i>class_name.method_name</i> 的方法的调用和返回。
<code>trace inmethod <i>class_name.method_name</i> ([<i>parameters</i>])</code>	跟踪对所有名为 <i>class_name.method_name</i> ([<i>parameters</i>]) 的方法的调用和返回。

其中：

class_name 是 Java 类的名称，可以使用包路径（使用句点 (.) 作为限定符，例如 `test1.extra.T1.Inner`），也可以使用完整路径名（前面加磅符号 (#) 并使用斜线 (/) 和美元符号 (\$) 作为限定符，例如 `#test1/extra/T1$Inner`）。如果使用 \$ 限定符，请用引号将 *class_name* 引起来。

file_name 是文件名。

line_number 是源代码行的编号。

method_name 是 Java 方法名。

parameters 是方法的参数

以下修饰符在 Java 模式中有效。

<code>-if <i>condition_expression</i></code>	只有在 <i>condition_expression</i> 求值为真时，才会发生指定事件并打印跟踪。
<code>-count <i>number</i></code>	从 0 开始，每次发生事件时，计数器的计数便会增加。当达到 <i>number</i> 时，便打印跟踪，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	创建一个临时断点，当事件发生并打印跟踪时即删除该断点。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，请参见第 261 页的“设置事件规范”。

tracei 命令

tracei 命令显示机器指令、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 修饰符可使跟踪在指令粒度上而非源代码行粒度上进行。发生事件时，信息将以反汇编格式而非源代码行格式打印。

语法

tracei step	跟踪每个机器指令。
tracei next -in <i>函数</i>	在指定 <i>function</i> 中跟踪每个指令。
tracei at <i>address</i>	跟踪 <i>address</i> 处的指令。
tracei in <i>function</i>	跟踪对指定 <i>function</i> 的调用和返回。
tracei inmember <i>function</i>	跟踪对所有名为 <i>function</i> 的成员函数的调用。
tracei infunction <i>function</i>	调用任何名为 <i>function</i> 的函数时进行跟踪。
tracei inclass <i>class</i>	跟踪对所有 <i>class</i> 的成员函数的调用。
tracei change <i>variable</i>	跟踪对 <i>variable</i> 的更改。

其中：

filename 是跟踪输出所要发送到的文件名。

function 是函数名。

line 是源代码行的编号。

class 是类名。

variable 是变量名。

更多信息请参见第 365 页的“trace 命令”。

unchecked 命令

unchecked 命令禁用内存访问、泄露或使用检查。仅在本地模式中有效。

注 - 只能在 Solaris 平台上执行 unchecked 命令。

语法

unchecked	打印当前检查状态。
unchecked -access	关闭访问检查。
unchecked -leaks	关闭泄漏检查。
unchecked -memuse	关闭内存使用检查（同时关闭泄漏检查）。
unchecked -all	等价于 unchecked -access; unchecked -memuse。
unchecked [<i>functions</i>] [<i>files</i>] [<i>loadobjects</i>]	等价于 <i>functions files loadobjects</i> 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。

loadobjects 是一个或多个加载对象名

有关打开检查的信息，请参见第 286 页的“check 命令”。

有关禁止错误的信息，请参见第 358 页的“suppress 命令”。

有关运行时检查的简介，请参见第 126 页的“运行时检查功能”。

undisplay 命令

undisplay 命令撤消 display 命令。

本地模式语法

undisplay <i>expression</i> ,	撤消 display <i>expression</i> 命令。
...	
undisplay <i>n</i> , ...	撤消编号为 <i>n...</i> 的 display 命令。
undisplay 0	撤消所有 display 命令。

其中：

expression 是一个有效表达式。

Java 模式语法

undisplay <i>expression</i> ,	撤消 display <i>expression</i> , ... 或 display <i>identifier</i> ,
... <i>identifier</i> , 命令。
undisplay <i>n</i> , ...	撤消编号为 <i>n...</i> 的 display 命令。
undisplay 0	撤消所有 display 命令。

其中：

expression 是一个有效的 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 命令撤消 hide 命令。仅在本地模式中有效。

语法

unhide 0	删除所有栈帧过滤器
unhide <i>regular_expression</i>	删除栈帧过滤器 <i>regular_expression</i> 。
unhide <i>number</i>	删除编号为 <i>number</i> 的栈帧过滤器。

其中：

regular_expression 是一个正规表达式。

number 是栈帧过滤器的编号。

隐藏命令（请参见第 316 页的“hide 命令”）可列出带有编号的过滤器。

unintercept 命令

unintercept 命令撤消 intercept 命令（仅限于 C++）。仅在本地模式中有效。

语法

unintercept <i>intercepted_typename</i> [, <i>intercepted_typename</i> ...]	从 intercept 列表中删除 <i>intercepted_typename</i> 类型的 throw。
unintercept -a[ll]	从 intercept 列表中删除所有类型的所有 throw。
unintercept -x <i>excluded_typename</i> [, <i>excluded_typename</i> ...]	从 excluded 列表中删除 <i>excluded_typename</i> 。
unintercept -x -a[ll]	从 excluded 列表中删除所有类型的所有 throw。
unintercept	列出被截取的类型

其中：

included_typename 和 *excluded_typename* 是异常类型规范，如 `List <int>` 或 `unsigned short`。

unsuppress 命令

unsuppress 命令撤消 suppress 命令。仅在本地模式中有效。

注 - 只能在 Solaris 平台上执行 unsuppress 命令。

语法

<code>unsuppress</code>	禁止和启用命令的历史记录（不包括指定 <code>-d</code> 和 <code>-reset</code> 选项的那些命令）
<code>unsuppress -d</code>	未被编译用于调试的函数中启用的错误列表。每个加载对象都有这样一个列表。其他错误只能使用带有 <code>-d</code> 选项的禁止命令（请参见第 358 页的“suppress 命令”）来禁止。
<code>unsuppress -d errors</code>	通过进一步启用 <i>errors</i> 来修改所有加载对象的默认禁止。
<code>unsuppress -d errors in loadobjects</code>	通过进一步启用 <i>errors</i> 来修改 <i>loadobjects</i> 中的默认禁止。
<code>unsuppress -last</code>	在错误位置启用当前错误。
<code>unsuppress -reset</code>	将默认禁止屏蔽设置为初始值（启动时）。
<code>unsuppress errors</code>	启用任何地方的 <i>errors</i> 。
<code>unsuppress errors in [functions] [files] [loadobjects]</code>	禁止 <i>functions</i> 列表、 <i>files</i> 列表和 <i>loadobjects</i> 列表中的 <i>errors</i> 。
<code>unsuppress errors at line</code>	启用 <i>line</i> 处的 <i>errors</i> 。
<code>unsuppress errors at "file":line</code>	启用 <i>file</i> 中 <i>line</i> 处的 <i>errors</i> 。
<code>unsuppress errors addr address</code>	启用 <i>address</i> 处的 <i>errors</i> 。

up 命令

up 命令可使调用栈上移（移至 main）。它在本地模式和 Java 模式中具有相同的语法和功能。

语法

up	调用栈上移一级。
up <i>number</i>	调用栈上移 <i>number</i> 级。
up -h [<i>number</i>]	上移调用栈，但不跳过隐藏的帧。

其中：

number 是调用栈级数。

use 命令

use 命令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，其使用已映射至以下 pathmap 命令：

use 等价于 pathmap -s

use *directory* 等价于 pathmap *directory*。

whatis 命令

在本地模式中，whatis 命令可打印表达式类型或类型的声明。在 Java 模式中，whatis 命令可打印标识符的声明。如果标识符是一个类，则打印类的方法信息（包括所有继承方法）。

本地模式语法

<code>whatIs [-n] [-r] name</code>	打印无类型 <i>name</i> 的声明。
<code>whatIs -t [-r] type</code>	打印类型 <i>type</i> 的声明
<code>whatIs -e [-r] [-d] expression</code>	打印表达式 <i>expression</i> 的类型。

其中：

name 是无类型名称。

type 是类型名。

expression 是一个有效表达式。

-d 显示动态类型，而非静态类型（仅限于 C++）。

-e 显示表达式的类型。

-n 显示无类型声明。不必指定 -n；如果键入不带选项的 `whatIs` 命令，即表示使用这一默认值。

-r 打印有关基类的信息（仅限于 C++）。

-t 显示类型的声明。

当运行在一个 C++ 类或结构上时，`whatIs` 命令将提供一个列表，其中列出所有已定义的成员函数（未定义的成员函数不会列出）、静态数据成员、类友元以及在该类中显式定义的数据成员。

指定 -r（递归）选项将添加来自继承类的信息。

与 -e 标志一起使用时，-d 标志使用表达式的动态类型。

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。
- 函数模板实例化通过 `whatIs` 列出。
- 类模板实例化通过 `whatIs -t` 列出。

Java 模式语法

<code>whatIs identifier</code>	打印 <i>identifier</i> 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 命令在指定事件发生时执行命令。

如果 dbx 处于 Java 模式，而您想在本机代码中设置 when 断点，请使用 joff 命令（请参见第 319 页的“joff 命令”）将其切换到“本地”模式，或在 when 命令前添加前缀 native（请参见第 332 页的“native 命令”）。

如果 dbx 处于 JNI 模式，而您想在 Java 代码中设置 when 断点，请在 when 命令前添加前缀 java（请参见第 319 页的“java 命令”）。

语法

when 命令有以下一般语法：

```
when event-specification [ modifier ] { command; ... }
```

发生指定事件时，执行上述命令。

本地模式语法

以下特定语法在本地模式中有效：

```
when at line_number { command; } 当到达 line_number 时，便执行 command。
```

```
when in procedure { command; } 当调用 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

以下特定语法在 Java 模式中有效。

<code>when at <i>line_number</i></code>	当到达源码 <i>line_number</i> 时，便执行命令。
<code>when at <i>file_name.line_number</i></code>	当到达 <i>file_name.line_number</i> 时，便执行命令。
<code>when in <i>class_name.method_name</i></code>	当调用 <i>class_name.method_name</i> 时，执行命令。
<code>when in <i>class_name.method_name</i> ([<i>parameters</i>])</code>	当调用 <i>class_name.method_name</i> (<i>[parameters]</i>) 时，执行命令。

class_name 是 Java 类的名称，可以使用包路径（使用句点 (.) 作为限定符；例如 `test1.extra.T1.Inner`），也可以使用完整路径名（前面加磅符号 (#) 并使用斜线 (/) 和美元符号 (\$) 作为限定符；例如 `#test1/extra/T1$Inner`）。如果使用 \$ 限定符，请用引号将 *class_name* 引起来。

file_name 是文件名。

line_number 是源代码行的编号。

method_name 是 Java 方法名。

parameters 是方法的参数。

有关所有事件的列表和语法，请参见第 261 页的“设置事件规范”。

有关在指定低级事件上执行命令的信息，请参见第 376 页的“wheni 命令”。

wheni 命令

wheni 命令在指定的低级事件发生时执行命令。仅在本地模式中有效。

wheni 命令有以下一般语法：

语法

```
wheni event-specification [ modifier ] { command ... ; }
```

发生指定事件时，执行上述命令。

以下特定语法有效：

`wheni at address { command; }` 当到达 *address* 时，便执行 *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，请参见第 261 页的“设置事件规范”。

where 命令

`where` 命令打印调用栈。

本地模式语法

其中	打印过程回溯。
<code>where <i>number</i></code>	打印回溯中的 <i>number</i> 个顶帧。
<code>where -f <i>number</i></code>	从帧 <i>number</i> 开始回溯。
<code>where -h</code>	包括隐藏的帧。
<code>where -l</code>	包括库名和函数名
<code>where -q</code>	快速回溯（仅限于函数名）。
<code>where -v</code>	冗余回溯（包括函数参数和行信息）。

其中：

number 是调用栈帧数。

线程或 LWP ID 可能均遵循以上任一格式，以便获得指定实体的回溯。

Java 模式语法

<code>where [thread_id]</code>	打印方法回溯。
<code>where [thread_id] number</code>	打印回溯中的 <i>number</i> 个顶帧。
<code>where -f [thread_id] number</code>	从帧 <i>number</i> 开始回溯。
<code>where -q [thread_id]</code>	快速回溯（仅限于方法名）。
<code>where -v [thread_id]</code>	冗余回溯（包括方法参数和行信息）。

其中：

number 是调用栈帧数。

thread_id 是 dbx 样式的线程 ID 或为线程指定的 Java 线程名。

whereami 命令

whereami 命令显示当前源代码行。仅在本地模式中有效。

语法

<code>whereami</code>	显示与当前位置（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）。
<code>whereami -instr</code>	同上，只不过打印的是当前反汇编指令而不是源代码行。

whereis 命令

`whereis` 命令打印指定名称的所有用法，或某个地址的符号名称。仅在本地模式中有效。

语法

<code>whereis name</code>	打印 <i>name</i> 的所有声明。
<code>whereis -a address</code>	打印某个 <i>address</i> 表达式的位置。

其中：

name 是作用域内可加载对象的名称；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` 命令打印某个指定名称的全部限定。仅在本地模式中有效。

语法

<code>which [-n] name</code>	打印 <i>name</i> 的全部限定。
<code>which -t type</code>	打印 <i>type</i> 的全部限定。

其中：

name 是作用域内某些对象的名称；例如，变量、函数、类模板或函数模板。

type 是类型名。

`-n` 显示无类型的全部限定。不必指定 `-n`；如果键入不带选项的 `which` 命令，即表示使用这一默认值。

`-t` 显示某个类型的全部限定。

whocatches 命令

`whocatches` 命令指明捕获 C++ 异常的位置。仅在本地模式中有效。

语法

`whocatches type`

如果在当前执行点抛出，则指明捕获 *type* 类型异常的位置。假定要执行的下一个语句是 `throw x`，其中 *x* 的类型为 *type*，并显示捕获它的 `catch` 子句的行号、函数名和帧号。如果捕获点位于执行抛出操作的同一个函数内，则返回“*type* 未处理”。

其中：

type 是异常的类型。

索引

符号

:: (双冒号) C++ 操作符, 75

A

access 事件, 263

adb 命令, 239, 281

adb 模式, 239

alias 命令, 54

AMD64 寄存器, 245

array_bounds_check 环境变量, 63

assign 命令, 118, 162, 164, 256, 281

attach 命令, 73, 88, 282

attach 事件, 268

B

bcheck 命令, 150

 示例, 151

 语法, 150

bind 命令, 249

bsearch 命令, 283

保存

 调试运行到文件, 57, 59

 系列调试运行为检查点, 59

报告异常类型被捕获的位置, 188

编辑器的键绑定, 显示或修改, 249

变量

 查看, 79

 查找定义, 80

 查找声明, 79

 打印值, 116

 赋值给, 118, 256

 关闭显示, 118

 检查, 43

 监视更改, 118

 确定 dbx 正在求哪个变量的值, 115

 声明, 查找, 79

 事件特定, 275, 276

 限定名, 74

 显示定义变量的函数和文件, 115

 修复后更改, 162

 作用域之外, 116

变量类型, 显示, 80

编译

 调试代码, 35

 使用 -g 选项, 54

 使用 -O 选项, 54

 优化代码, 54

编译器, 访问, 28

编译时未使用 -g 选项的代码, 55

表达式

 打印值, 116, 257

 复数, Fortran, 208

 关闭显示, 118

 监视更改, 118

 监视值, 118

 区间, Fortran, 209

 显示, 118

剥离的程序, 56

捕获特定类型的异常, 187

捕获信号列表, 181

C

C 源文件, 指定位置, 220

C++

打印, 116

对象指针类型, 116

二义或重载函数, 70

反引号操作符, 74

跟踪成员函数, 104

函数模板实例化, 列出, 80

继承的成员, 82

类

查看, 79

查看继承的成员, 82

打印声明, 81

定义, 查找, 81

声明, 查找, 79

显示所有继承的数据成员, 116

显示所有直接定义的数据成员, 116

模板调试, 190

模板定义

显示, 80

修复, 164

设置多个断点, 98

使用 dbx, 185

使用 -g 选项编译, 54

使用 -g0 选项编译, 54

双冒号作用域转换操作符, 75

损坏名称, 77

未命名的参数, 117

异常处理, 186

C++ 源文件, 指定位置, 220

call 命令, 92, 195, 257, 284

cancel 命令, 285

catch 块, 186

catch 命令, 181, 182, 285

change 事件, 264

check 命令, 44, 127, 128, 286

CLASSPATHX 环境变量, 63, 216

clear 命令, 288

collector archive 命令, 290

collector dbxsample 命令, 290

collector disable 命令, 290

collector enable 命令, 290

collector heaptrace 命令, 291

collector hw_profile 命令, 291

collector limit 命令, 292

collector mpitrace 命令, 292

collector 命令, 288

collector pause 命令, 292

collector profile 命令, 292

collector resume 命令, 293

collector sample 命令, 293

collector show 命令, 294

collector status 命令, 294

collector store 命令, 294

collector synctrace 命令, 295

collector version 命令, 295

cond 事件, 264

cont 命令, 91, 129, 161, 162, 164, 168, 258, 296

对未使用调试信息编译的文件的限制, 160

core_lo_pathmap 环境变量, 63

-count 事件规范修饰符, 272

操作事件处理程序, 260

查看

变量, 79

成员, 79

类, 79

类型, 79

另一线程的上下文, 167

线程列表, 168

查找

变量定义, 80

成员定义, 80

函数定义, 80

类定义, 81

类型定义, 81

this 指针, 81

程序

剥离, 56

单步执行, 90

多线程

- 调试, 165
 - 恢复执行, 168
 - 恢复执行于指定行, 91
 - 继续执行, 91
 - 修复后, 161
 - 在指定行, 258
 - 停止执行
 - 如果条件语句的求值为真, 101
 - 如果指定变量的值已更改, 101
 - 修复, 160, 258
 - 运行, 87
 - 打开运行时检查, 129
 - 在 dbx 下, 影响, 255
 - 中止, 57
 - 状态, 检查, 279
 - 成员
 - 查看, 79
 - 查找定义, 80
 - 查找声明, 79
 - 声明, 查找, 79
 - 成员函数
 - 打印, 80
 - 跟踪, 104
 - 设置多个断点于, 98
 - 成员模板函数, 190
 - 重新运行已保存的调试运行, 60
 - 处理程序, 259
 - 创建, 260
 - 在函数内时启用, 277
 - 处理程序 id, 定义的, 260
 - 创建
 - .dbxrc 文件, 62
 - 事件处理程序, 260
 - 从 C++ 二义函数名称列表中选择, 70
 - 错误禁止, 140, 141
 - 类型, 141
 - 默认, 142
 - 示例, 141
- D**
- dalias 命令, 296
 - dbx 环境变量, 63
 - array_bounds_check, 63
 - CLASSPATHX, 63, 216
 - core_lo_pathmap, 63
 - disassembler_version, 63
 - fix_verbose, 63
 - follow_fork_inherit, 63, 172
 - follow_fork_mode, 63, 143, 172
 - follow_fork_mode_inner, 63
 - 和 Korn shell, 67
 - Java 调试, 216
 - JAVASRCPATH, 63, 216
 - jdbx_mode, 63, 216
 - input_case_sensitive, 63, 198
 - jvm_invocation, 64, 216
 - language_mode, 64
 - mt_scalable, 64
 - overload_function, 64
 - overload_operator, 64
 - output_auto_flush, 64
 - output_base, 64
 - output_class_prefix, 64
 - output_derived_type, 116
 - output_dynamic_type, 64, 186
 - output_inherited_members, 64
 - output_list_size, 64
 - output_log_file_name, 64
 - output_max_string_length, 64
 - output_pretty_print, 64
 - output_short_file_name, 64
 - pop_auto_destruct, 64
 - proc_exclusive_attach, 65
 - rtc_auto_continue, 65, 129, 151
 - rtc_auto_suppress, 65, 141
 - rtc_biu_at_exit, 65, 139
 - rtc_error_limit, 65, 141
 - rtc_error_log_file_name, 65, 129, 151
 - rtc_error_stack, 65
 - rtc_inherit, 65
 - rtc_mel_at_exit, 65
 - run_autostart, 65
 - run_io, 65
 - run_pty, 65
 - run_quick, 65
 - run_savetty, 66
 - run_setpgrp, 66
 - scope_global_enums, 66
 - scope_look_aside, 66, 79
 - session_log_file_name, 66

- stack_find_source, 66, 73
- stack_max_size, 66
- stack_verbose, 66
- step_abflow, 66
- step_events, 66, 107
- step_granularity, 66, 90
- suppress_startup_message, 66
- symbol_info_compression, 66
- 使用 dbxenv 命令设置, 62
- trace_speed, 66, 105
- dbx 命令, 47, 52, 297
 - 调试 Java 代码时使用的静态和动态信息, 227
 - Java 表达式求值, 226
 - 在 Java 模式和本地模式下具有完全相同的语法和功能, 227
 - 在 Java 模式下使用, 226
 - 在 Java 模式下有不同语法, 228
 - 只在 Java 模式下有效, 229
- dbx, 启动, 47
 - 带核心文件名, 48
 - 仅进程 ID, 52
 - 启动选项, 298
- dbxenv 命令, 54, 62, 299
- .dbxrc 文件, 61
 - 创建, 62
 - sample, 62
 - 用于 dbx 启动, 52, 61
- dbxrc 文件在 dbx 启动时使用, 52, 61
- debug 命令, 48, 73, 88, 171, 300
- delete 命令, 302
- detach 命令, 56, 89, 303
- detach 事件, 269
- dis 命令, 72, 234, 303
- disable 事件规范修饰符, 272
- disassembler_version 环境变量, 63
- display 命令, 118, 304
- 打开
 - 内存访问检查, 44, 127, 128
 - 内存使用检查, 44, 127, 128
 - 内存泄漏检查, 128
- dlopen 事件, 264
- dlopen()
 - 断点限制, 106
 - 设置断点, 106
- down 命令, 74, 111, 305
- dump 命令, 306
 - 在 OpenMP 代码上使用, 176
- 打印
 - 变量或表达式的值, 116
 - 变量类型, 80
 - 表达式的值, 257
 - 成员函数, 80
 - 当前模块的名称, 84
 - 符号具体值列表, 77
 - 函数模板实例化的值, 190
 - 含所有已知线程的列表, 168
 - 某类型或 C++ 类的声明, 81
 - OpenMP 代码中的共享、专用和线程专用变量, 175
 - 数据成员, 80
 - 数组, 119
 - 所有机器级寄存器的值, 239
 - 所有类和函数模板实例化列表, 190, 192
 - 通常不输出的线程（僵停）列表, 168
 - 源码列表, 71
 - 指定函数实例化的源码列表, 196
 - 指针, 214
 - 字段类型, 80
- 单步执行
 - 程序, 90
 - 在机器指令级, 236
- 单步执行程序, 42, 90
- 当前地址, 72
- 当前过程和文件, 197
- 导航
 - 到函数, 70
 - 到文件, 70
 - 通过在调用栈中移动, 71
- 地址
 - 当前, 72
 - 检查内容于, 231
 - 显示格式, 233
- 调用
 - 成员模板函数, 190
 - 过程, 257
 - 函数, 92

- 函数实例化或类模板的成员函数, 195
- 调用选项, 298
- 调用栈, 109
 - 查看, 43
 - 弹出, 111, 162, 256
 - 一个帧, 163
 - 定义, 109
 - 确定位置, 109
 - 删除
 - 所有帧过滤器, 112
 - 帧, 112
 - 删除停止于函数, 111
 - 上下移动, 71, 110
 - 移动
 - down, 111
 - 到特定帧, 111
 - up, 110
 - 隐藏帧, 112
 - 帧, 定义的, 109
- 动态链接程序, 251
- 读取栈跟踪, 113
- 读入
 - 模块的调试信息, 84
 - 所有模块的调试信息, 84
- 断点
 - 定义, 40, 95
 - 多个, 在非成员函数中设置, 99
 - 概述, 95
 - In Function, 97
 - In Object, 99
 - 禁用, 107
 - 列出, 106
 - 启用, 107
 - 清除, 106
 - stop 类型, 95
 - 确定何时设置, 69
 - 删除, 使用处理程序 ID, 106
 - 设置
 - C++ 代码中的多个断点, 98
 - 函数模板实例化, 190, 194
 - 计算机级, 238
 - 类模板实例化, 190, 194
 - 使用包含函数调用的过滤器, 103
 - 在 JVM 软件尚未装入的代码上, 220

- 在不同类的成员函数中, 98
- 在地址处, 238
- 在对象中, 99
- 在共享库中, 106, 252
- 在函数模板的所有实例中, 194
- 在函数中, 40, 97
- 在模板类的成员函数或模板函数中, 194
- 在显式装入的库中, 252
- 在相同类的成员函数中, 98
 - 在行内, 41, 96
- 设置过滤器于, 102
- 事件发生后启用, 279
- 事件规范, 261
- 事件效率, 107
- trace 类型, 95
- when 类型, 95
 - 在行中设置, 105
- 限制, 106
- 值更改时, 101

段故障

- Fortran, 原因, 201
- 生成, 202
- 找到行号, 202

对象文件

- 确定, 53, 85

对象指针类型, 116

多线程程序, 调试, 165

E

- edit 命令, 306
- examine 命令, 72, 231, 307
- exception 命令, 186, 308
- exec 函数, 跟随, 172
- exists 命令, 308
- exit 事件, 267

F

- fault 事件, 265
- fflush(stdout), dbx 调用后, 92
- file 命令, 70, 72, 74, 309
- files 命令, 309
- fix 命令, 160, 258, 310

- 对未使用调试信息编译的文件的限制, 160
- 结果, 161
- fix_verbose 环境变量, 63
- fixed 命令, 312
- follow_fork_inherit 环境变量, 63, 172
- follow_fork_mode 环境变量, 63, 143, 172
- follow_fork_mode_inner 环境变量, 63
- fork 函数, 跟随, 172
- Fortran
 - 复数表达式, 208
 - 结构, 211
 - 可分配数组, 206
 - 逻辑操作符, 210
 - 内函数, 207
 - 派生类型, 211
 - 区分大小写, 198
 - 区间表达式, 209
 - 数组分片语法, 120
- FPE 信号, 捕获, 181
- frame 命令, 74, 111, 312
- func 命令, 70, 72, 74, 313
- funcs 命令, 314
- 范围
 - 查找规则, 放松, 79
 - 当前, 69, 72
 - 定义, 72
 - 访问, 73
 - 更改, 73
 - 组件, 73
 - 更改访问, 73
- 反引号操作符, 74
- 访问检查, 132
- 访问作用域, 73
 - 更改, 73
 - 组件, 73
- 非关联化指针, 117
- 分离
 - dbx 中的进程, 56, 89
 - dbx 中的进程并将其保留在停止状态, 89
- 分片
 - C 和 C++ 数组, 119
 - Fortran 数组, 120

- 数组, 122
- 浮点异常 (FPE)
 - 捕获, 280
 - 确定位置, 182
 - 确定原因, 182
- 符号
 - 打印具体值列表, 77
 - 确定 dbx 使用哪个, 78
 - 在多个具体值中选择, 71
- 符号名, 限定作用域, 74

G

- g 编译器选项, 54
- gdb 命令, 315
- 给变量赋值, 118, 256
- 跟随
 - exec 函数, 172
 - fork 函数, 172
- 跟踪
 - 控制速度, 105
 - 列出, 106
 - 设置, 104
 - 实现, 277
- 跟踪输出, 定向到文件, 105
- 更改
 - 当前位于栈中的函数, 162
 - 当前正在执行的函数, 162
 - 默认信号列表, 181
 - 未被调用的函数, 161
 - 修复后变量, 162
 - 已执行的函数, 161
- 共享对象
 - 使用修复并继续, 252
 - 修复, 160
- 共享库
 - 设置断点于, 252
 - 为 dbx 编译, 56
- 关闭
 - 所有当前被监视的变量的显示, 118
 - 特定变量或表达式的显示, 118
 - 运行时检查, 128
- 过程, 调用, 257
- 过程链接表, 252

H

handler 命令, 261, 315

-hidden 事件规范修饰符, 273

hide 命令, 112, 316

函数

查找定义, 80

当前位于栈中, 更改, 162

当前正在执行的, 更改, 162

导航到, 70

调用, 92

二义或重载, 70

获取编译器所赋给的名称, 117

类模板成员, 调用, 195

类模板成员, 求值, 195

内在, Fortran, 207

设置断点于, 97

实例化

打印源码列表, 196

调用, 195

求值, 195

未被调用, 更改, 161

限定名, 74

已执行的, 更改, 161

在 C++ 代码中设置断点, 98

函数参数, 未命名

求值, 117

显示, 117

函数模板实例化

打印列表, 190, 192

打印值, 190

显示源代码, 190

行中的 when 断点, 设置, 105

核心文件

调试, 39, 48

调试不匹配, 49

检查, 39

忽略信号列表, 181

汇编语言调试, 231

恢复

程序执行于指定行, 91

执行多线程程序, 168

恢复已保存的调试运行, 59

会话, dbx

启动, 47

退出, 56

获取编译器所赋给的函数名, 117

I

-if 事件规范修饰符, 271

ignore 命令, 180, 181, 317

import 命令, 317

In Function 断点, 97

In Object 断点, 99

in 事件, 261

-in 事件规范修饰符, 272

inclass 事件, 262

infunction 事件, 262

inmember 事件, 262

inmethod 事件, 262

inobject 事件, 262

input_case_sensitive 环境变量, 63, 198

-instr 事件规范修饰符, 272

Intel 寄存器, 243

intercept 命令, 187, 318

J

JAR 文件, 调试, 217

Java 代码

dbx 的功能, 215

dbx 的限制, 216

调试的 dbx 模式, 224

使用 dbx, 215

Java 调试, 环境变量, 216

Java 类文件, 调试, 217

java 命令, 319

Java 应用程序

开始调试, 216

可使用 dbx 调试的类型, 216

连接 dbx 到, 218

使用包装器, 调试, 218

要求 64 位库, 219

指定自定义包装器, 222

Java 源文件, 指定位置, 220

JAVASRCPATH 环境变量, 63, 216

jclasses 命令, 319
继承的成员
 查看, 82
 显示, 81
寄存器
 AM64 体系结构, 245
 打印值, 239
 Intel 体系结构, 243
 SPARC 体系结构, 242
jdbx_mode 环境变量, 63, 216

joff 命令, 319
jon 命令, 320
jpkgs 命令, 320
机器指令级
 AMD64 寄存器, 245
 打印所有寄存器的值, 239
 单步执行, 236
 地址, 设置断点, 238
 调试, 231
 跟踪, 236
 Intel 寄存器, 243
 SPARC 寄存器, 242
 在地址设置断点, 238

JVM 软件
 将运行参数传递给, 220, 222
 指定 64 位, 224
 指定路径名, 222
 自定义启动, 221
jvm_invocation 环境变量, 64, 216

继续执行程序, 91
 修复后, 161
 在指定行, 91, 258

加载程序, 36

检查点, 将系列调试运行另存为, 59

检查内存的内容, 231

监视表达式的值, 118

进程
 从 dbx 中分离, 56, 89
 从 dbx 中分离并保留在停止状态, 89
 连接的, 使用运行时检查, 147
 使用 Ctrl+C 停止, 93
 停止执行, 56

运行, 连接 dbx 到, 88, 89
子

 连接 dbx 到, 171
 使用运行时检查, 143

进程控制命令, 定义, 87

禁止上一错误, 141

K

kill 命令, 57, 136, 320

Korn shell

 扩展, 248
 未实现的功能, 247
 与 dbx 的区别, 247
 重命名命令, 248

Korn shell 与 dbx 命令间的区别, 247

控制跟踪速度, 105

库

 动态链接, 设置断点于, 106
 共享, 为 dbx 编译, 56

跨越数组片, 123

块局部操作符, 75

L

language 命令, 321

language_mode 环境变量, 64

lastrites 事件, 269

LD_AUDIT, 147

librtc.so, 预装, 147

librtld_db.so, 251

libthread.so, 165

libthread_db.so, 165

line 命令, 72, 322

list 命令, 71, 72, 74, 196, 323

listi 命令, 235, 325

loadobject -dumpelf 命令, 326

loadobject -exclude 命令, 326

loadobject -hide 命令, 327

loadobject -list 命令, 327

loadobject -load 命令, 328

loadobject 命令, 325

loadobject -unload 命令, 328

- loadobject -use 命令, 328
- lwp 命令, 329
 - lwp 事件规范修饰符, 273
- lwp_exit 事件, 265
- lwps 命令, 170, 329
- LWP (轻量进程), 165
 - 显示相关信息, 170
 - 信息显示, 170
- 类
 - 查看, 79
 - 查看继承的成员, 82
 - 查找定义, 81
 - 查找声明, 79
 - 打印声明, 81
 - 显示所有继承的数据成员, 116
 - 显示所有直接定义的数据成员, 116
- 类模板实例化, 打印列表, 190, 192
- 类型
 - 查看, 79
 - 查找定义, 81
 - 查找声明, 79
 - 打印声明, 81
 - 派生, Fortran, 211
 - 声明, 查找, 79
- 连接
 - dbx 到正在运行的进程, 52, 88
 - 当 dbx 尚未运行时, 89
 - dbx 到正在运行的子进程, 171
 - 在调试现有进程时将 dbx 连接到新的进程, 89
- 链接程序名, 77
- 连接的进程, 使用运行时检查, 147
- 链接映射, 251
- 列出
 - 包含调试信息的所有程序模块, 85
 - 包含已读入 dbx 的调试信息的模块的名称, 85
 - C++ 函数模板实例化, 80
 - 当前被忽略的信号, 181
 - 当前正在捕获的信号, 181
 - 断点, 106
 - 跟踪, 106
 - 模块的调试信息, 84
 - 所有程序模块的名称, 85

M

- MANPATH 环境变量, 设置, 29
- memory
 - 地址显示格式, 233
 - 检查内容于地址, 231
 - 显示模式, 231
 - 状态, 132
- mmapfile 命令, 330
- module 命令, 84, 331
- modules 命令, 84, 85, 332
- mt_scalable 环境变量, 64
- 命令
 - adb, 239, 281
 - alias, 54
 - assign, 118, 162, 164, 256, 281
 - attach, 88, 282
 - 按 adb(1) 语法输入, 239
 - bcheck, 150
 - bind, 249
 - bsearch, 283
 - call, 92, 195, 257, 284
 - cancel, 285
 - catch, 181, 182, 285
 - check, 44, 127, 128, 286
 - clear, 288
 - collector, 288
 - collector archive, 290
 - collector dbxsample, 290
 - collector disable, 290
 - collector enable, 290
 - collector heaptrace, 291
 - collector hw_profile, 291
 - collector limit, 292
 - collector mpitrace, 292
 - collector pause, 292
 - collector profile, 292
 - collector resume, 293
 - collector sample, 293
 - collector show, 294
 - collector status, 294
 - collector store, 294
 - collector synctrace, 295
 - collector version, 295
 - cont, 91, 129, 161, 162, 164, 168, 258, 296
 - 对未使用调试信息编译的文件的限制, 160
 - dalias, 296

dbx, 47, 52, 297
 dbxenv, 54, 62, 299
 debug, 48, 88, 171, 300
 delete, 302
 detach, 56, 89, 303
 dis, 72, 234, 303
 down, 111, 305
 dump, 306
 在 OpenMP 代码上使用, 176
 打印, 116, 117, 119, 120, 195, 257, 337
 edit, 306
 examine, 72, 231, 307
 exception, 186, 308
 exists, 308
 fix, 160, 258, 310
 对未使用调试信息编译的文件的限制, 160
 结果, 161
 fixed, 312
 frame, 111, 312
 func, 70, 72, 313
 funcs, 314
 gdb, 315
 更改程序的状态, 256
 handler, 261, 315
 hide, 112, 316
 java, 319
 jclassses, 319
 ignore, 180, 181, 317
 import, 317
 intercept, 187, 318
 joff, 319
 jon, 320
 jpkgs, 320
 进程控制, 87
 禁止, 129, 140, 142, 143, 358
 kill, 57, 136, 320
 line, 72, 322
 list, 71, 72, 196, 323
 listi, 235, 325
 loadobject, 325
 loadobject -dumpelf, 326
 loadobject -exclude, 326
 loadobject -hide, 327
 loadobject -list, 327
 loadobject -load, 328
 loadobject -unload, 328
 loadobject -use, 328
 lwp, 329
 lwps, 170, 329
 mmapfile, 330
 module, 84, 331
 模块, 84, 85, 332
 native, 332
 next, 90, 333
 nexti, 236, 334
 pathmap, 53, 85, 161, 335
 pop, 74, 111, 163, 256, 336
 proc, 340
 prog, 340
 quit, 341
 其中, 110, 204, 377
 regs, 239, 341
 replay, 57, 60, 342
 rerun, 343
 restore, 57, 59, 343
 rtc -showmap, 344
 run, 87, 345
 runargs, 346
 save, 57, 346
 scopes, 347
 search, 347
 showblock, 127, 347
 showleaks, 135, 138, 139, 141, 348
 showmemuse, 139, 349
 status, 350
 step, 90, 186, 350
 step to, 42, 90, 351
 step up, 90, 351
 stepi, 236, 352
 stop, 194, 353
 stop change, 101
 stop inclass, 98
 stop inmember, 98
 stopi, 238, 358
 stop<Default Para Font, 194
 sync, 361
 syncs, 361
 trace, 104, 365
 tracei, 236, 368
 whatis, 80, 81, 117, 192, 373
 when, 105, 257, 260, 375
 wheni, 376

- whereami, 378
- whereis, 77, 115, 192, 379
- which, 71, 78, 115, 379
- whocatches, 188, 380
- uncheck, 128, 369
- undisplay, 118, 370
- unhide, 112, 371
- unintercept, 187, 371
- unsuppress, 140, 143, 372
- up, 110, 373
- use, 373
- 文件, 70, 72, 309
- x, 231
- 线程, 167, 168, 361, 363
- 显示, 118, 304
- 语言, 321
- 源文件, 349

模板

- 查找声明, 81
- 函数, 190
- 类, 190
 - 停止在所有成员函数中, 194
- 实例化, 190
 - 打印列表, 190, 192
 - 显示定义, 190, 192

模块

- 包含调试信息, 列出, 85
- 包含已读入 dbx 的调试信息, 列出, 85
- 当前, 打印名称, 84
- 列出调试信息, 84
- 所有程序, 列表, 85

N

- native 命令, 332
- next 命令, 90, 333
- next 事件, 267
- nexti 命令, 236, 334
- 内存访问
 - 错误, 133, 153
 - 错误报告, 133
 - 检查, 132
 - 打开, 44, 127, 128
- 内存使用检查, 139
 - 打开, 44, 127, 128

- 内存泄漏
 - 报告, 136
 - 错误, 135, 156
 - 检查, 134, 136
 - 打开, 44, 127, 128
 - 修复, 139
- 内嵌 Java 应用程序的 C 应用程序, 调试, 219
- 内嵌 Java 应用程序的 C++ 应用程序
 - 调试, 219

O

OpenMP 代码

- 编译器转换, 174
- dbx 可用功能, 175
- 打印共享、专用和线程专用变量, 175
- 单步执行, 175
- 使用 dump 命令, 176
- 使用栈跟踪, 175
- 执行序列, 177

OpenMP 应用编程接口, 173

- overload_function 环境变量, 64
- overload_operator 环境变量, 64
- output_auto_flush 环境变量, 64
- output_base 环境变量, 64
- output_class_prefix 环境变量, 64
- output_derived_type 环境变量, 116
- output_dynamic_type 环境变量, 64, 186
- output_inherited_members 环境变量, 64
- output_list_size 环境变量, 64
- output_log_file_name 环境变量, 64
- output_max_string_length 环境变量, 64
- output_pretty_print 环境变量, 64
- output_short_file_name 环境变量, 64

P

- PATH 环境变量, 设置, 29
- pathmap 命令, 53, 85, 161, 335
 - perm 事件规范修饰符, 273
- pop 命令, 74, 111, 163, 256, 336
- pop_auto_destruct 环境变量, 64
- print 命令, 116, 117, 119, 120, 195, 257, 337

proc 命令, 340
proc_exclusive_attach 环境变量, 65
proc_gone 事件, 269
prog 命令, 340
prog_new 事件, 269

Q

启动 dbx, 36
启动选项, 298
quit 命令, 341
清除断点, 106
求值
 函数实例化或类模板的成员函数, 195
 数组, 119
 未命名的函数参数, 117
区分大小写, Fortran, 198

确定

 程序崩溃的位置, 39
 dbx 使用哪个符号, 78
 对象文件, 53, 85
 浮点异常 (FPE) 位置, 182
 浮点异常 (FPE) 原因, 182
 已执行的行数, 278
 已执行的指令数, 278
 源代码行单步执行的粒度, 90
 源文件, 53, 85
 在调用栈中的位置, 109

R

regs 命令, 239, 341
replay 命令, 57, 60, 342
rerun 命令, 343
restore 命令, 57, 59, 343
-resumeone 事件规范修饰符, 104, 271
returns 事件, 267, 268
rtc -showmap 命令, 344
rtc_auto_continue 环境变量, 65, 129, 151
rtc_auto_suppress 环境变量, 65, 141
rtc_biu_at_exit 环境变量, 65
rtc_error_limit 环境变量, 65, 141

rtc_error_log_file_name 环境变量, 65, 129, 151
rtc_error_stack 环境变量, 65
rtc_inherit 环境变量, 65
rtc_mel_at_exit 环境变量, 65
rtld, 251
run 命令, 87, 345
run_autostart 环境变量, 65
run_io 环境变量, 65
run_pty 环境变量, 65
run_quick 环境变量, 65
run_savetty 环境变量, 66
run_setpgrp 环境变量, 66
runargs 命令, 346

S

save 命令, 57, 346
scope_global_enums 环境变量, 66
scope_look_aside 环境变量, 66, 79
scopes 命令, 347
search 命令, 347
session_log_file_name 环境变量, 66
Shell 提示符, 27
showblock 命令, 127, 347
showleaks 命令, 135, 138, 139, 141, 348
showmemuse 命令, 139, 349
sig 事件, 266
source 命令, 349
SPARC 寄存器, 242
stack_find_source 环境变量, 66, 73
stack_max_size 环境变量, 66
stack_verbose 环境变量, 66
status 命令, 350
step 命令, 90, 186, 350
step 事件, 268
step to 命令, 42, 90, 351
step up 命令, 90, 351
step_abflow 环境变量, 66
step_events 环境变量, 66, 107

- step_granularity 环境变量, 66, 90
- stepi 命令, 236, 352
- stop at 命令, 96, 97
- stop change 命令, 101
- stop inclass 命令, 98
- stop inmember 命令, 98
- stop 命令, 194, 353
- stop 事件, 269
- stopi 命令, 238, 358
- suppress 命令, 129, 140, 142, 143, 358
- suppress_startup_message 环境变量, 66
- symbol_info_compression 环境变量, 66
- sync 命令, 361
- sync 事件, 270
- syncrtld 事件, 270
- syncs 命令, 361
- sysin 事件, 266
- sysout 事件, 267

删除

- 调用栈帧, 112
- 调用栈中的停止于函数, 111
- 截取列表中的异常类型, 187
- 使用处理程序 ID 的特定断点, 106
- 所有调用栈帧过滤器, 112

设置

断点

- 使用包含函数调用的过滤器, 103
- 在 JVM 软件尚未装入的代码上, 220
- 在不同类的成员函数中, 98
- 在对象中, 99
- 在函数模板的所有实例中, 194
- 在模板类的成员函数或模板函数中, 194
- 在相同类的成员函数中, 98
- 非成员函数中的多个断点, 99
- 跟踪, 104
- 过滤器于断点上, 102
- 使用 dbxenv 命令设置 dbx 环境变量, 62

声明, 查找 (显示), 79

事件

- 二义性, 273
- 分析, 273

- 子进程交互, 172
- 事件处理程序
 - 操作, 260
 - 创建, 260
 - 设置, 示例, 277
 - 隐藏, 273
 - 在所有会话中保留, 273
- 事件发生后启用断点, 279
- 事件规范, 238, 259, 260, 261
 - access, 263
 - at, 262
 - attach, 268
 - cond, 264
 - detach, 269
 - dlopen, 264
 - 断点事件, 261
 - exit, 267
 - fault, 265
 - 更改, 264
 - 关键字, 定义的, 261
 - in, 261
 - inclass, 262
 - infunction, 262
 - inmember, 262
 - inmethod, 262
 - inobject, 262
 - lastrites, 269
 - lwp_exit, 265
 - next, 267
 - prog_gone, 269
 - prog_new, 269
 - 其他类型事件, 268
 - returns, 267, 268
 - sig, 266
 - step, 268
 - stop, 269
 - sync, 270
 - syncrtld, 270
 - sysin, 266
 - sysout, 267
 - 设置, 261
 - 使用预定义变量, 274
 - 数据更改事件, 263
 - thr_create, 169, 270
 - thr_exit, 169, 270

- throw, 270, 271
- timer, 271
 - 系统事件, 264
 - 修饰符, 271
 - 执行进度事件, 267
- 事件规范的预定义变量, 274
- 事件规范修饰符
 - count, 272
 - disable, 272
 - hidden, 273
 - if, 271
 - in, 272
 - instr, 272
 - lwp, 273
 - perm, 273
 - resumeone, 104, 271
 - temp, 272
 - thread, 272
- 事件计数器, 261
- 事件特定变量, 275
- 示例 .dbxrc 文件, 62
- 实例, 显示定义, 190, 192
- 实验
 - 限制其大小, 292
- 手册页, 访问, 28
- 数据成员, 打印, 80
- 数据更改事件规范, 263
- 数组
 - 边界, 超出, 201
 - Fortran, 205
 - Fortran 95 可分配, 206
 - 分片, 119, 122
 - C 和 C++ 语法, 119
 - Fortran 语法, 120
 - 分片和跨距的语法, 119
 - 跨距, 119, 123
 - 求值, 119

T

- temp 事件规范修饰符, 272
- thr_create 事件, 169, 270
- thr_exit 事件, 169, 270
- thread 命令, 167, 361

- thread 事件规范修饰符, 272
- threads 命令, 168, 363
- throw 事件, 270, 271
- timer 事件, 271
- trace 命令, 104, 365
- trace_speed 环境变量, 66, 105
- tracei 命令, 236, 368

弹出

- 调用栈, 111, 162, 256
- 调用栈的一个帧, 163

调试

- 编译时未使用 -g 选项的代码, 55
- 不匹配的核心文件, 49
- 多线程程序, 165
- 核心文件, 39, 48
- 汇编语言, 231
- 机器指令级, 231, 236
- 优化代码, 55
- 子进程, 171

- 调试 Java 代码的 dbx 模式, 224

调试信息

- 模块, 读入, 84
- 所有模块, 读入, 84

调试运行

- 保存, 57
- 已保存
 - 恢复, 59
 - 重新运行, 60

- 调整默认 dbx 设置, 61

停止

- 程序执行
 - 如果条件语句的求值为真, 101
 - 如果指定变量的值已更改, 101
- 进程, 使用 Ctrl+C, 93
- 进程执行, 56
- 在模板类的所有成员函数中, 194

- 头文件, 修改, 164

- 退出 dbx, 44

- 退出 dbx 会话, 56

U

- unchecked 命令, 128, 369

undisplay 命令, 118, 370
unhide 命令, 112, 371
unintercept 命令, 187, 371
unsuppress 命令, 140, 143, 372
up 命令, 74, 110, 373
use 命令, 373

W

whatis 命令, 80, 81, 117, 192, 373
when 命令, 105, 257, 260, 375
wheni 命令, 376
where 命令, 110, 204, 377
whereami 命令, 378
whereis 命令, 77, 115, 192, 379
which 命令, 71, 78, 115, 379
whocatches 命令, 188, 380
为使用自定义类加载器的类文件指定路径, 220
为重新运行而重置应用程序文件, 279
文档, 访问, 30 至 32
文档索引, 30
文件
 导航到, 70
 确定, 53, 85
 位置, 85
 限定名, 74

X

x 命令, 231
系统事件规范, 264
线程
 打印含所有已知的列表, 168
 打印含通常不输出的 (僵停) 列表, 168
 当前, 显示, 167
 列表, 查看, 168
 其他, 切换查看上下文到, 167
 通过线程 id 切换, 167
 信息显示, 166
 只继续执行遇到断点时所在的第一个线程, 104
线程创建, 了解, 169
限定符号名, 74
显示

变量和表达式, 118
变量类型, 80
符号, 具体值, 77
函数模板实例化的源代码, 190
继承的成员, 81
模板定义, 80
模板和实例的定义, 190, 192
声明, 79
所有从基类继承的数据成员, 116
所有由类直接定义的数据成员, 116
未命名的函数参数, 117
异常的类型, 186
栈跟踪, 112

限制实验大小, 292

信号

捕获, 180
dbx 接受的名称, 181
FPE, 捕获, 181
更改默认列表, 181
忽略, 181
列出当前被忽略的信号, 181
列出当前正在捕获的信号, 181
取消, 179
在程序中发送, 182
转发, 180
自动处理, 183

行程计数器, 261

修复

C++ 模板定义, 164
程序, 160, 258
共享对象, 160
修复并继续, 159
 对共享对象使用, 252
 如何操作, 160
 修改源代码, 160
 与运行时检查一并使用, 148
 约束, 160

修改头文件, 164

Y

验证 dbx 正在求哪个变量的值, 115
异常
 报告类型被捕获的位置, 188

- 从截取列表中删除类型, 187
- 浮点, 确定位置, 182
- 浮点, 确定原因, 182
- 类型, 显示, 186
- 特定类型, 捕获, 187
- 在 Fortran 程序中, 定位, 203
- 异常处理, 186
 - 示例, 188
- 移动
 - 到调用栈中的特定帧, 111
 - 调用栈向上, 110
 - 调用栈向下, 111
- 易读文档, 31
- 疑难解答提示, 运行时检查, 152
- 隐藏调用栈帧, 112
- 印刷约定, 26
- 优化代码
 - 编译, 54
 - 调试, 55
- 预装 librt.c.so, 147
- 源码列表, 打印, 71
- 源文件, 查找, 53, 85
- 运算符
 - C++ 双冒号作用域转换, 75
 - 反引号, 74
 - 块局部, 75
- 运行程序, 38, 87
 - 打开运行时检查, 129
 - 在 dbx 中, 不带参数, 38, 87
- 运行时检查
 - 错误, 153
 - 错误禁止, 140
 - 错误禁止类型, 141
 - 访问检查, 132
 - 非 UltraSPARC 上的 8 M 字节限制, 152
 - 关闭, 128
 - 禁止错误, 140
 - 默认, 142
 - 示例, 141
 - 禁止上一错误, 141
 - 可能的泄漏, 135
 - 连接的进程, 147
 - 内存访问
 - 错误, 133, 153
 - 错误报告, 133
 - 检查, 132
 - 内存使用检查, 139
 - 内存泄漏
 - 错误, 135, 156
 - 错误报告, 136
 - 检查, 134, 136
 - 使用时机, 126
 - 使用修复并继续, 148
 - 限制, 127
 - 修复内存泄漏, 139
 - 要求, 126
 - 疑难解答提示, 152
 - 应用编程接口, 149
 - 在批处理模式下使用, 150
 - 直接在 dbx, 151
 - 子进程, 143
- Z**
 - 在调用栈中移动, 71, 110
 - 在符号的多个具体值中选择, 71
 - 在机器指令级跟踪, 236
 - 在目录之间建立新映射, 53, 85
 - 在事件, 262
 - 栈跟踪, 204
 - 读取, 113
 - 示例, 113, 114
 - 显示, 112
 - 在 OpenMP 代码上使用, 175
 - 栈帧, 定义的, 109
 - 帧, 定义的, 109
 - 执行进度事件规范, 267
 - 指针
 - 打印, 214
 - 非关联化, 117
 - 中止
 - 程序, 57
 - 仅程序, 57
 - 装入对象, 定义的, 251
 - 自定义 dbx, 61
 - 字段类型
 - 打印, 80

- 显示, 80
- 子进程
 - 调试, 171
 - 连接 dbx 到, 171
 - 使用运行时检查, 143
 - 与事件交互, 172
- 作用域转换操作符, 74
- 作用域转换搜索路径, 78

