



Sun Studio 12: dbx コマンドによるデバッグ



Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 820-1199-10

Sun Microsystems, Inc. (以下米国 Sun Microsystems 社とします) は、本書に記述されている製品に含まれる技術に関連する知的財産権を所有します。特に、この知的財産権はひとつかそれ以上の米国における特許、あるいは米国およびその他の国において申請中の特許を含んでいることがあります。それらに限定されるものではありません。

U.S. Government Rights Commercial software. Government users are subject to the Sun Microsystems, Inc. standard license agreement and applicable provisions of the FAR and its supplements.

この配布には、第三者によって開発された素材を含んでいることがあります。

本製品の一部は、カリフォルニア大学からライセンスされている Berkeley BSD システムに基づいていることがあります。UNIX は、X/Open Company, Ltd. が独占的にライセンスしている米国ならびに他の国における登録商標です。フォント技術を含む第三者のソフトウェアは、著作権により保護されており、提供者からライセンスを受けているものです。

Sun、Sun Microsystems、Sun のロゴマーク、Solaris のロゴマーク、Java Coffee Cup のロゴマーク、docs.sun.com、Java および Solaris は、米国およびその他の国における米国 Sun Microsystems 社の商標、登録商標もしくは、サービスマークです。

すべての SPARC 商標は、米国 SPARC International, Inc. のライセンスを受けて使用している同社の米国およびその他の国における商標または登録商標です。SPARC 商標が付いた製品は、米国 Sun Microsystems 社が開発したアーキテクチャに基づくものです。

OPEN LOOK および Sun Graphical User Interface は、米国 Sun Microsystems 社が自社のユーザおよびライセンス実施権者向けに開発しました。米国 Sun Microsystems 社は、コンピュータ産業用のビジュアルまたはグラフィカル・ユーザインタフェースの概念の研究開発における米国 Xerox 社の先駆者としての成果を認めるものです。米国 Sun Microsystems 社は米国 Xerox 社から Xerox Graphical User Interface の非独占的ライセンスを取得しており、このライセンスは、OPEN LOOK のグラフィカル・ユーザインタフェースを実装するか、またはその他の方法で米国 Sun Microsystems 社との書面によるライセンス契約を遵守する、米国 Sun Microsystems 社のライセンス実施権者にも適用されます。

本書で言及されている製品や含まれている情報は、米国輸出規制法で規制されるものであり、その他の国の輸出入に関する法律の対象となることがあります。核、ミサイル、化学あるいは生物兵器、原子力の海洋輸送手段への使用は、直接および間接を問わず厳しく禁止されています。米国が禁輸の対象としている国や、限定はされませんが、取引禁止顧客や特別指定国民のリストを含む米国輸出排除リストで指定されているものへの輸出および再輸出は厳しく禁止されています。

本書は、「現状のまま」をベースとして提供され、商品性、特定目的への適合性または第三者の権利の非侵害の黙示の保証を含みそれに限定されない、明示的であるか黙示的であるかを問わない、なんらの保証も行われぬものとします。

本製品が、外国為替および外国貿易管理法(外為法)に定められる戦略物資等(貨物または役務)に該当する場合、本製品を輸出または日本国外へ持ち出す際には、サン・マイクロシステムズ株式会社の事前の書面による承諾を得ることのほか、外為法および関連法規に基づく輸出手続き、また場合によっては、米国商務省または米国所轄官庁の許可を得ることが必要です。

原典: Sun Studio 12: Debugging a Program With dbx

Part No: 819-5257

目次

はじめに	23
1 dbx の概要	31
デバッグを目的としてコードをコンパイルする	31
dbx を起動してプログラムを読み込む	32
プログラムを dbx で実行する	33
dbx を使用してプログラムをデバッグする	34
コアファイルをチェックする	34
ブレークポイントを設定する	36
プログラムをステップ実行する	37
呼び出しスタックを確認する	38
変数を調べる	39
メモリアクセス問題とメモリーリークを検出する	39
dbx を終了する	40
dbx オンラインヘルプにアクセスする	41
2 dbx の起動	43
デバッグセッションを開始する	43
既存のコアファイルのデバッグ	44
同じオペレーティング環境でのコアファイルのデバッグ	44
コアファイルが切り捨てられている場合	45
一致しないコアファイルのデバッグ	46
プロセス ID の使用	48
dbx 起動時シーケンス	49
起動属性の設定	50
デバッグ時ディレクトリへのコンパイル時ディレクトリのマッピング	50
dbx 環境変数の設定	50

ユーザー自身の dbx コマンドを作成	51
デバッグのためのプログラムのコンパイル	51
-g オプションでコンパイル	51
別のデバッグファイルの使用	51
▼別のデバッグファイルの作成	52
最適化コードのデバッグ	52
-g オプションを使用しないでコンパイルされたコード	54
dbx を完全にサポートするために -g オプションを必要とする共有ライブラリ ..	54
完全にストリップされたプログラム	54
デバッグセッションを終了する	55
プロセス実行の停止	55
dbx からのプロセスの切り離し	55
セッションを終了せずにプログラムを終了する	55
デバッグ実行の保存と復元	56
save コマンドの使用	56
一連のデバッグ実行をチェックポイントとして保存する	57
保存された実行の復元	58
replay を使用した保存と復元	58
3 dbx のカスタマイズ	61
dbx 初期化ファイルの使用	61
.dbxrc ファイルの作成	62
初期化ファイル	62
dbx 環境変数の設定	62
dbx 環境変数および Korn シェル	68
4 コードの表示とコードへの移動	69
コードへの移動	69
ファイルの内容を表示する	70
関数を表示する	70
ソースリストの出力	71
呼び出しスタックの操作によってコードを表示する	71
プログラム位置のタイプ	72
プログラムスコープ	72
現在のスコープを反映する変数	72

表示スコープ	73
スコープ決定演算子を使用してシンボルを特定する	74
逆引用符演算子	75
コロンを重ねたスコープ決定演算子 (C++)	75
ブロックローカル演算子	75
リンカー名	77
シンボルを検索する	77
シンボルの出現を出力する	77
実際に使用されるシンボルを調べる	78
スコープ決定検索パス	78
スコープ検索規則の緩和	79
変数、メンバー、型、クラスを調べる	80
変数、メンバー、関数の定義を調べる	80
型およびクラスの定義を調べる	81
オブジェクトファイルおよび実行可能ファイル内のデバッグ情報	83
オブジェクトファイルの読み込み	83
モジュールについてのデバッグ情報	84
モジュールのリスト	85
ソースファイルおよびオブジェクトファイルの検索	85
5 プログラムの実行制御	87
dbx でプログラムを実行する	87
動作中のプロセスに dbx を接続する	88
プロセスから dbx を切り離す	90
プログラムのステップ実行	90
シングルステップ	91
プログラムを継続する	91
▼ 特定の行からプログラムの実行を再開する	92
関数を呼び出す	92
安全な呼び出し	93
Control+C によってプロセスを停止する	94
6 ブレークポイントとトレースの設定	95
ブレークポイントを設定する	96
ソースコードの特定の行に stop ブレークポイントを設定する	96

関数に stop ブレークポイントを設定する	97
C++ プログラムに複数のブレークポイントを設定する	98
データ変更ブレークポイントを設定する	100
ブレークポイントのフィルタの設定	103
関数の戻り値をフィルタとして使用	103
トレースの実行	105
トレースを設定する	105
トレース速度を制御する	106
ファイルにトレース出力を転送する	106
ソース行で when ブレークポイントを設定する	106
共有ライブラリでブレークポイントを設定する	107
ブレークポイントをリストおよびクリアする	107
ブレークポイントとトレースポイントの表示	107
ハンドラ ID を使用して特定のブレークポイントを削除	108
ブレークポイントを有効および無効にする	108
イベント効率	108
7 呼び出しスタックの使用	111
スタック上での現在位置の検索	112
スタックを移動してホームに戻る	112
スタックを上下に移動する	112
スタックの上方向への移動	112
スタックの下方向への移動	113
特定フレームへの移動	113
呼び出しスタックのポップ	113
スタックフレームを隠す	114
スタックトレースを表示して確認する	114
8 データの評価と表示	117
変数と式の評価	117
実際に使用される変数を確認する	117
現在の関数のスコープ外にある変数	118
変数、式または識別子の値を出力する	118
pretty-print の使用	118
C++ での表示	121

ポインタを間接参照する	122
式を監視する	122
表示を取り消す (非表示)	123
変数に値を代入する	123
配列を評価する	123
配列の断面化	124
配列の断面	126
刻み幅	127
9 実行時検査	129
概要	130
RTCを使用する場合	130
RTCの必要条件	130
実行時検査	131
メモリー使用状況とメモリーリーク検査を有効化	131
メモリーアクセス検査を有効化	131
すべてのRTCを有効化	131
RTCを無効化	131
プログラムを実行	132
アクセス検査の使用	134
メモリーアクセスエラーの報告	135
メモリーアクセスエラー	136
メモリーリークの検査	137
メモリーリーク検査の使用	138
リークの可能性	138
リークの検査	139
メモリーリークの報告を理解する	139
メモリーリークの修正	142
メモリー使用状況検査の使用	142
エラーの抑止	144
抑止のタイプ	144
エラー抑止の例	145
デフォルトの抑止	145
抑止によるエラーの制御	146
子プロセスにおけるRTCの実行	146

接続されたプロセスへの RTC の使用	149
RTC での修正継続機能の使用	150
実行時検査アプリケーションプログラミングインタフェース	152
バッチモードでの RTC の使用	152
bcheck 構文	153
bcheck の例	153
dbx からバッチモードを直接有効化	154
トラブルシューティングのヒント	154
実行時検査の制限	155
より高い効果を得るにはより多くのシンボルおよびデバッグ情報が必要にな る	155
x86 プラットフォームでは SIGSEGV シグナルと SIGALTSTACK シグナルが制限され る	155
より高い効果を得るには、十分なパッチ領域を設け、すべての既存コードを含め て 8M バイト以内にする (SPARC プラットフォームのみ)	155
RTC エラー	158
アクセスエラー	158
メモリーリークエラー	160
10 修正継続機能 (fix と cont)	163
修正継続機能の使用	163
fix と cont の働き	164
fix と cont によるソースの変更	164
プログラムの修正	165
▼ ファイルを修正する	165
修正後の続行	166
修正後の変数の変更	167
ヘッダファイルの変更	168
C++ テンプレート定義の修正	168
11 マルチスレッドアプリケーションのデバッグ	169
マルチスレッドデバッグについて	169
スレッド情報	170
別のスレッドのコンテキストの表示	171
スレッドリストの表示	172

実行の再開	172
スレッド作成動作について	174
LWP 情報について	175
12 子プロセスのデバッグ	177
単純な接続の方法	177
exec 機能後のプロセス追跡	178
fork 機能後のプロセス追跡	178
イベントとの対話	178
13 OpenMP プログラムのデバッグ	179
コンパイラによる OpenMP コードの変換	180
OpenMP コードで利用可能な dbx の機能	181
OpenMP コードにおけるスタックトレースの使用	181
OpenMP コードにおける dump コマンドの使用	182
OpenMP コードの実行シーケンス	182
14 シグナルの処理	183
シグナルイベントについて	183
システムシグナルを捕獲する	184
デフォルトの catch リストと ignore リストを変更する	185
FPE シグナルをトラップする (Solaris プラットフォームのみ)	185
プログラムにシグナルを送信する	187
シグナルの自動処理	187
15 dbx を使用してプログラムをデバッグする	189
C++ での dbx の使用	189
dbx での例外処理	190
例外処理コマンド	190
例外処理の例	192
C++ テンプレートでのデバッグ	194
テンプレートの例	194
C++ テンプレートのコマンド	196

16	dbxを使用した Fortran のデバッグ	201
	Fortranのデバッグ	201
	カレントプロシージャとカレントファイル	201
	大文字	202
	dbxのサンプルセッション	202
	▼ dbxのサンプルセッションの実行	203
	セグメント不正のデバッグ	205
	dbxにより問題を見つける方法	205
	例外の検出	206
	呼び出しのトレース	207
	配列の操作	208
	Fortran 95 割り当て可能配列	209
	組み込み関数	210
	複合式	210
	間隔式の表示	211
	論理演算子	212
	Fortran 95 構造型の表示	213
	Fortran 95 構造型へのポインタ	214
17	dbxによる Java アプリケーションのデバッグ	217
	dbxとJavaコード	217
	Javaコードに対するdbxの機能	217
	Javaコードのデバッグにおけるdbxの制限事項	217
	Javaデバッグ用の環境変数	218
	Javaアプリケーションのデバッグの開始	218
	クラスファイルのデバッグ	219
	JARファイルのデバッグ	219
	ラッパーを持つJavaアプリケーションのデバッグ	220
	動作中のJavaアプリケーションへのdbxの接続	220
	Javaアプリケーションを埋め込むC/C++アプリケーションのデバッグ	221
	JVMソフトウェアへの引数の引き渡し	222
	Javaソースファイルの格納場所の指定	222
	C/C++ソースファイルの格納場所の指定	222
	独自のクラスローダーを使用するクラスファイルのパスの指定	222
	JVMソフトウェアによって読み込まれていないコードに対するブレークポイント	

トの設定	223
JVM ソフトウェアの起動方法のカスタマイズ	223
JVM ソフトウェアのパス名の指定	224
JVM ソフトウェアへの実行引数の引き渡し	224
Java アプリケーション用の独自のラッパーの指定	225
64 ビット JVM ソフトウェアの指定	226
dbx の Java コードデバッグモード	227
Java または JNI モードからネイティブモードへの切り替え	227
実行中断時のモードの切り替え	227
Java モードにおける dbx コマンドの使用法	228
dbx コマンドにおける Java の式の評価	228
dbx コマンドが利用する静的および動的情報	229
構文と機能が Java モードとネイティブモードで完全に同じコマンド	229
Java モードで構文が異なる dbx コマンド	230
Java モードでのみ有効なコマンド	232
18 機械命令レベルでのデバッグ	233
メモリーの内容を調べる	233
examine または x コマンドの使用	234
dis コマンドの使用	236
listi コマンドの使用	237
機械命令レベルでのステップ実行とトレース	238
機械命令レベルでステップ実行する	238
機械命令レベルでトレースする	239
機械命令レベルでブレークポイントを設定する	240
あるアドレスにブレークポイントを設定する	240
regs コマンドの使用	240
プラットフォーム固有のレジスタ	243
x86 レジスタ情報	244
AMD64 レジスタ情報	247
19 dbx の Korn シェル機能	251
実装されていない ksh-88 の機能	251
ksh-88 から拡張された機能	252
名前が変更されたコマンド	252

編集機能のキーバインドの変更	252
20 共有ライブラリのデバッグ	255
動的リンカー	255
リンクマップ	256
起動手順と .init セクション	256
プロシージャリンケージテーブル	256
修正と継続	256
共有ライブラリにおけるブレークポイントの設定	257
明示的に読み込まれたライブラリにブレークポイントを設定する	257
A プログラム状態の変更	259
dbx 下でプログラムを実行することの影響	259
プログラムの状態を変更するコマンドの使用	260
assign コマンド	260
pop コマンド	260
call コマンド	261
print コマンド	261
when コマンド	261
fix コマンド	262
cont at コマンド	262
B イベント管理	263
イベントハンドラ	263
イベントの安全性	264
イベントハンドラの作成	265
イベントハンドラを操作するコマンド	265
イベントカウンタ	266
イベント指定の設定	266
ブレークポイントイベント仕様	266
データ変更イベント指定	268
システムイベント指定	269
実行進行状況イベント仕様	272
その他のイベント仕様	273

イベント指定のための修飾子	276
- <i>if condition</i>	276
- <i>resumeone</i>	277
- <i>in function</i>	277
- <i>disable</i>	277
- <i>count n -count infinity</i>	277
- <i>temp</i>	277
- <i>instr</i>	278
- <i>thread thread_id</i>	278
- <i>lwp lwp_id</i>	278
- <i>hidden</i>	278
- <i>perm</i>	278
解析とあいまいさに関する注意	279
事前定義済み変数	279
when コマンドに対して有効な変数	281
イベント別の有効変数	281
イベントハンドラの設定例	282
配列メンバーへのストアに対するブレークポイントを設定する	282
単純なトレースを実行する	283
関数の中だけハンドラを有効にする (<i>in function</i>)	283
実行された行の数を調べる	283
実行された命令の数をソース行で調べる	284
イベント発生後にブレークポイントを有効にする	284
replay時にアプリケーションファイルをリセットする	284
プログラムの状態を調べる	285
浮動小数点例外を捕捉する	285
C コマンドリファレンス	287
assign コマンド	287
ネイティブモードの構文	287
Java モードの構文	287
attach コマンド	288
構文	288
▼実行中の Java プロセスに接続する	288
bsearch コマンド	289

構文	289
call コマンド	289
ネイティブモードの構文	290
Java モードの構文	290
cancel コマンド	291
構文	291
catch コマンド	291
構文	291
check コマンド	292
構文	292
clear コマンド	294
構文	295
collector コマンド	295
構文	295
collector archive コマンド	297
collector dbxsample コマンド	297
collector disable コマンド	297
collector enable コマンド	298
collector heaptrace コマンド	298
collector hwprofile コマンド	298
collector limit コマンド	299
collector mpitrace コマンド	299
collector pause コマンド	299
collector profile コマンド	300
collector resume コマンド	300
collector sample コマンド	300
collector show コマンド	301
collector status コマンド	301
collector store コマンド	301
collector synctrace コマンド	302
collector tha コマンド	302
collector version コマンド	303
cont コマンド	303
構文	303
dalias コマンド	303
構文	304

dbx コマンド	304
ネイティブモードの構文	304
Java モードの構文	305
オプション	306
dbxenv コマンド	307
構文	307
debug コマンド	307
ネイティブモードの構文	307
Java モードの構文	309
オプション	309
delete コマンド	310
構文	310
detach コマンド	311
ネイティブモードの構文	311
Java モードの構文	311
dis コマンド	311
構文	311
オプション	312
display コマンド	312
ネイティブモードの構文	312
Java モードの構文	313
down コマンド	314
構文	314
dump コマンド	314
構文	314
edit コマンド	314
構文	315
examine コマンド	315
構文	315
exception コマンド	316
構文	317
exists コマンド	317
構文	317
file コマンド	317
構文	317
files コマンド	318

ネイティブモードの構文	318
Java モードの構文	318
fix コマンド	318
構文	319
fixed コマンド	319
構文	319
fortran_module コマンド	319
構文	319
frame コマンド	320
構文	320
func コマンド	320
ネイティブモードの構文	320
Java モードの構文	321
funcs コマンド	321
構文	321
gdb コマンド	322
構文	322
handler コマンド	323
構文	323
hide コマンド	323
構文	324
ignore コマンド	324
構文	324
import コマンド	324
構文	325
intercept コマンド	325
構文	325
java コマンド	326
構文	326
jclasses コマンド	326
構文	326
joff コマンド	326
構文	326
jon コマンド	327
構文	327
jpkgs コマンド	327

構文	327
kill コマンド	327
構文	327
language コマンド	328
構文	328
line コマンド	328
構文	328
list コマンド	329
構文	329
listi コマンド	331
loadobject コマンド	331
構文	331
loadobject -dumpelf コマンド	332
loadobject -exclude コマンド	332
loadobject -hide コマンド	333
loadobject -list コマンド	333
loadobject -load コマンド	334
loadobject -unload コマンド	334
loadobject -use コマンド	335
lwp コマンド	335
構文	335
lwps コマンド	336
構文	337
mmapfile コマンド	337
構文	337
module コマンド	337
構文	337
modules コマンド	338
構文	339
native コマンド	339
構文	339
next コマンド	339
ネイティブモードの構文	340
Java モードの構文	340
nexti コマンド	341
構文	341

pathmap コマンド	342
構文	342
pop コマンド	343
構文	344
print コマンド	344
ネイティブモードの構文	344
Java モードの構文	346
proc コマンド	347
構文	347
prog コマンド	347
構文	347
quit コマンド	348
構文	348
regs コマンド	348
構文	348
replay コマンド	349
構文	349
rerun コマンド	349
構文	349
restore コマンド	350
構文	350
rprint コマンド	350
構文	350
rtc showmap コマンド	351
構文	351
rtc skipatch コマンド	351
構文	351
run コマンド	352
ネイティブモードの構文	352
Java モードの構文	352
runargs コマンド	353
構文	353
save コマンド	353
構文	354
scopes コマンド	354
構文	354

search コマンド	354
構文	354
showblock コマンド	355
構文	355
showleaks コマンド	355
構文	355
showmemuse コマンド	356
構文	356
source コマンド	357
構文	357
status コマンド	357
構文	357
step コマンド	358
ネイティブモードの構文	358
Java モードの構文	359
stepi コマンド	360
構文	360
stop コマンド	360
構文	360
stopi コマンド	365
構文	365
suppress コマンド	366
構文	366
sync コマンド	368
構文	368
syncs コマンド	369
構文	369
thread コマンド	369
ネイティブモードの構文	369
Java モードの構文	370
threads コマンド	371
ネイティブモードの構文	371
Java モードの構文	372
trace コマンド	372
構文	373
tracei コマンド	376

構文	376
unchecked コマンド	377
構文	377
undisplay コマンド	378
ネイティブモードの構文	378
Java モードの構文	378
unhide コマンド	379
構文	379
unintercept コマンド	379
構文	379
unsuppress コマンド	380
構文	380
unwatch コマンド	381
構文	381
up コマンド	381
構文	381
use コマンド	382
watch コマンド	382
構文	382
whatis コマンド	383
ネイティブモードの構文	383
Java モードの構文	384
when コマンド	384
構文	384
wheni コマンド	386
構文	386
where コマンド	386
ネイティブモードの構文	386
Java モードの構文	387
whereami コマンド	388
構文	388
whereis コマンド	388
構文	388
which コマンド	388
構文	389
whocatches コマンド	389

構文 389

索引 391

はじめに

dbx コマンド行デバッガは、対話型でソースレベルのデバッグツールです。

対象読者

このマニュアルは、dbx コマンドを使用してアプリケーションのデバッグを行うプログラマを対象にしています。dbx のユーザーには、Fortran、C、または C++ による開発経験を持ち、Solaris™ または Linux オペレーティングシステムと UNIX® コマンドについてある程度の知識が必要です。

表記上の規則

このマニュアルでは、次のような字体や記号を特別な意味を持つものとして使用します。

表 P-1 表記上の規則

字体または記号	意味	例
AaBbCc123	コマンド名、ファイル名、ディレクトリ名、画面上のコンピュータ出力、コード例を示します。	.login ファイルを編集します。 ls -a を使用してすべてのファイルを表示します。 system%
AaBbCc123	ユーザーが入力する文字を、画面上のコンピュータ出力と区別して示します。	system% su password:
<i>AaBbCc123</i>	変数を示します。実際に使用する特定の名前または値で置き換えます。	ファイルを削除するには、rm <i>filename</i> と入力します。
『 』	参照する書名を示します。	『コードマネージャ・ユーザーズガイド』を参照してください。

表 P-1 表記上の規則 (続き)

字体または記号	意味	例
「」	参照する章、節、ボタンやメニュー名、強調する単語を示します。	第5章「衝突の回避」を参照してください。 この操作ができるのは、「スーパーユーザー」だけです。
\	枠で囲まれたコード例で、テキストがページ幅を超える場合に、継続を示します。	<pre>sun% grep '^#define \ XV_VERSION_STRING'</pre>

このマニュアルでは、次のようなコードを特別な意味を持つものとして使用します。

表 P-2 コードについて

コードの記号	意味	記法	コード例
[]	角括弧には、オプションの引数が含まれます。	x0[n]	x04,0
{}	中括弧には、必須オプションの選択肢が含まれます。	d{y n}	dy
	「パイプ」または「バー」と呼ばれる記号は、その中から1つだけを選択可能な複数の引数を区切ります。	B{dynamic static}	Bstatic
:	コロンは、コンマ同様に複数の引数を区切るために使用されることがあります。	Rdir[:dir]	R/local/libs:/U/a
...	省略記号は、連続するものの一部が省略されていることを示します。	xinline=fl[...fn]	xinline=alpha,dos

コマンド例のシェルプロンプト

以下の表に、C シェル、Bourne シェル、および Korn シェルのデフォルトの UNIX システムプロンプト、およびスーパーユーザーのプロンプトを紹介します。

表 P-3 シェルプロンプト

シェル	プロンプト
C シェル	machine_name%
C シェルのスーパーユーザー	machine_name#
Bourne シェルおよび Korn シェル	\$
Bourne シェルおよび Korn シェルのスーパーユーザー	#

サポートされるプラットフォーム

この Sun™ Studio のリリースは、SPARC® および x86 ファミリ (UltraSPARC®, SPARC64, AMD64, Pentium, Xeon EM64T) プロセッサアーキテクチャーを使用するシステムをサポートしています。サポートされるシステムの、Solaris オペレーティングシステムのバージョンごとの情報については、<http://www.sun.com/bigadmin/hcl> にあるハードウェアの互換性に関するリストで参照することができます。ここには、すべてのプラットフォームごとの実装の違いについて説明されています。

このドキュメントでは、x86 関連の用語は次のものを指します。

- 「x86」は、64-ビットおよび 32-ビットの、x86 と互換性のある製品を指します。
- 「x64」は、AMD64 または EM64T システムで、特定の 64-ビット情報を指します。
- 「32-ビット x86」は、x86 ベースシステムで特定の 32 ビット情報を指します。

サポートされるシステムについては、ハードウェアの互換性に関するリストを参照してください。

Sun Studio マニュアルへのアクセス方法

マニュアルには、次の場所からアクセスできます。

- 製品マニュアルは、ご使用のローカルシステムまたはネットワークにこのソフトウェアとともにインストールされるマニュアルの索引からアクセスできます。Solaris プラットフォームの場合は、file:/opt/SUNWspro/docs/ja/index.html、Linux プラットフォームの場合は、file:/opt/sun/sunstudio12/docs/ja/index.html です。

製品ソフトウェアが Solaris プラットフォームで /opt、Linux プラットフォームで /opt/sun 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

- マニュアルは、docs.sun.comSM の Web サイトで入手できます。次に示すマニュアルは、Solaris プラットフォームにインストールされているソフトウェアからのみアクセスできます。
 - 『Standard C++ Library Class Reference』
 - 『標準 C++ ライブラリ・ユーザズガイド』
 - 『Tools.h++ クラスライブラリ・リファレンスマニュアル』
 - 『Tools.h++ ユーザズガイド』

リリースノートは docs.sun.com の Web サイトから入手できます。

- IDE の全コンポーネントのオンラインヘルプは、IDE 内の「ヘルプ」メニューだけでなく、多くのウィンドウおよびダイアログにある「ヘルプ」ボタンを使用してアクセスできます。

インターネットの Web サイト (<http://docs.sun.com>) から、Sun のマニュアルを参照したり、印刷したり、購入したりすることができます。マニュアルが見つからない場合はローカルシステムまたはネットワークの製品とともにインストールされているマニュアルの索引を参照してください。

注 - このマニュアルで紹介する Sun 以外の Web サイトが使用可能かどうかについては、Sun は責任を負いません。このようなサイトやリソース上、またはこれらを経由して利用できるコンテンツ、広告、製品、またはその他の資料についても、Sun は保証しておらず、法的責任を負いません。また、このようなサイトやリソース上、またはこれらを経由して利用できるコンテンツ、商品、サービスの使用や、それらへの依存に関連して発生した実際の損害や損失、またはその申し立てについても、Sun は一切の責任を負いません。

アクセシブルな製品マニュアル

マニュアルは、技術的な補足をすることで、ご不自由なユーザーの方々にとって読みやすい形式のマニュアルを提供しております。アクセシブルなマニュアルは次の表に示す場所から参照することができます。製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

表P-4 アクセシブルな製品マニュアル

マニュアルの種類	アクセシブルな形式と格納場所
マニュアル (Sun 以外のマニュアルは除く)	形式: HTML 場所: http://docs.sun.com
他社製マニュアル: <ul style="list-style-type: none"> ■ 『Standard C++ Library Class Reference』 ■ 『標準 C++ ライブラリ・ユーザーズガイド』 ■ 『Tools.h++ クラスライブラリ・リファレンスマニュアル』 ■ 『Tools.h++ ユーザーズガイド』 	形式: HTML 場所: file:/opt/SUNWspr/docs/ja/index.html (Solaris プラットフォーム) のマニュアル索引
Readme	形式: HTML 場所: http://developers.sun.com/sunstudio/documentation/ss12/mr の Sun Developer Network ポータル
マニュアルページ	形式: HTML 場所: file:/opt/SUNWspr/docs/ja/index.html (Solaris プラットフォーム) および file:/opt/sun/sunstudio12/docs/ja/index.html (Linux プラットフォーム) のマニュアル索引
オンラインヘルプ	形式: HTML 場所: IDE 内の「ヘルプ」メニューおよび「ヘルプ」ボタン
リリースノート	形式: HTML 場所: http://docs.sun.com

Sun Studio 関連マニュアル

次の表は、file:/opt/SUNWspr/docs/ja/index.html および <http://docs.sun.com> から参照できる関連マニュアルの一覧です。製品ソフトウェアが /opt 以外のディレクトリにインストールされている場合は、システム管理者に実際のパスをお尋ねください。

表P-5 Sun Studio 関連マニュアル

マニュアル	内容の説明
dbx Readme	dbx の新機能、既知の問題点、制限事項、および互換性の問題について説明しています。

表 P-5 Sun Studio 関連マニュアル (続き)

マニュアル	内容の説明
dbx (1) マニュアルページ	dbx コマンドの詳細について説明しています。
C ユーザーズガイド	Sun Studio 12 C プログラミング言語コンパイラについて説明しています。また、ANSI C コンパイラの詳細情報も記載されています。
C++ ユーザーズガイド	Sun Studio 12 C++ コンパイラの使用方法を説明しています。また、コマンド行コンパイラオプションの詳細情報も記載されています。
Fortran ユーザーズガイド	Sun Studio 12 Fortran コンパイラのコンパイル時環境およびコマンド行オプションについて説明しています。
OpenMP API ユーザーズガイド	多重処理アプリケーションのための OpenMP Fortran 95、C、C++ アプリケーションプログラムインタフェース (API) の概要を説明しています。Sun Studio コンパイラは、OpenMP API をサポートしています。
プログラムのパフォーマンス解析	Sun Studio 12 で利用可能なパフォーマンス解析ツールについて説明しています。

関連する Solaris マニュアル

次の表では、docs.sun.com の Web サイトで参照できる関連マニュアルについて説明します。

表 P-6 Solaris 関連マニュアル

マニュアルコレクション	マニュアルタイトル	内容の説明
Solaris Reference Manual Collection	マニュアルページのセクションのタイトルを参照。	Solaris OS に関する情報を提供しています。
Solaris Software Developer Collection	リンカーとライブラリ	Solaris のリンクエディタと実行時リンカーの操作について説明しています。
Solaris Software Developer Collection	マルチスレッドのプログラミング	POSIX と Solaris スレッド API、同期オブジェクトのプログラミング、マルチスレッド化したプログラムのコンパイル、およびマルチスレッド化したプログラムのツール検索について説明しています。

表 P-6 Solaris 関連マニュアル (続き)

マニュアルコレクション	マニュアルタイトル	内容の説明
Solaris Software Developer Collection	SPARC Assembly Language Reference Manual	SPARC アーキテクチャーで動作し、アセンブリ言語形式をリンク形式のオブジェクトファイルに翻訳するアセンブラについて説明しています。

開発者向けのリソース

Sun Developer Network の Sun Studio ポータル (<http://developers.sun.com/sunstudio>) にアクセスし、次のようなリソースを利用できます。リソースは頻繁に更新されます。

- プログラミング技術と最適な演習に関する技術文書
- ソフトウェアのマニュアル、およびソフトウェアとともにインストールされる一連のマニュアル
- サポートレベルに関する情報
- ユーザーフォーラム
- ダウンロード可能なサンプルコード
- 新しい技術の紹介

Sun Studio ポータルは、Sun Developer Network Web サイト (<http://developers.sun.com>) に多数ある開発者向け追加リソースの 1 つです。

技術サポートへの問い合わせ

製品についての技術的なご質問がございましたら、次のサイトからお問い合わせください(このマニュアルで回答されていないものに限りです)。

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

ご意見の送付先

米国 Sun Microsystems, Inc. では、マニュアルの向上に力を注いでおり、ユーザーのご意見やご提案をお待ちしております。コメントは下記よりお送りください。 <http://www.sun.com/hwdocs/feedback>

ご意見をお寄せいただく際には、電子メールの件名にマニュアルの Part No. を記載してください。たとえば、このマニュアルの Part No. は 820-1199-10 です。

dbx の概要

dbx は、対話型でソースレベルの、コマンド行ベースのデバッグツールです。dbx を使用して、プログラムを制御下に置いた状態で実行し、停止したプログラムの状態を調べることができます。このツールにより、プログラムの動的な実行を完璧に制御できるほか、パフォーマンスデータとメモリーの使用状況の収集、メモリーアクセスの監視、およびメモリーリークの検出も行えます。

dbx は、C、C++、または Fortran で記述されたアプリケーションのデバッグに使用できます。また、多少の制限はありますが(217 ページの「Java コードのデバッグにおける dbx の制限事項」を参照)、Java™ コードおよび C JNI (Java Native Interface) コードまたは C++ JNI コードの混在するアプリケーションをデバッグすることも可能です。

この章では、dbx によるアプリケーションのデバッグの基礎について説明します。この章の内容は次のとおりです。

- 31 ページの「デバッグを目的としてコードをコンパイルする」
- 32 ページの「dbx を起動してプログラムを読み込む」
- 33 ページの「プログラムを dbx で実行する」
- 34 ページの「dbx を使用してプログラムをデバッグする」
- 40 ページの「dbx を終了する」
- 41 ページの「dbx オンラインヘルプにアクセスする」

デバッグを目的としてコードをコンパイルする

dbx でソースレベルのデバッグを行えるようにプログラムを作成するには、-g オプションを付けてプログラムをコンパイルする必要があります。このオプションは、C、C++、Fortran 95、および Java の各コンパイラで利用できます。詳細については、51 ページの「デバッグのためのプログラムのコンパイル」を参照してください。

dbx を起動してプログラムを読み込む

dbx を起動するには、シェルプロンプトで dbx を入力します。

```
$ dbx
```

dbx を起動してデバッグ対象プログラムを読み込むには、次を入力します。

```
$ dbx program_name
```

dbx を起動して、Java コードおよび JNI コードまたは C++ JNI コードが混在するプログラムを読み込むには、次を入力します。

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

dbx コマンドを使用すると、dbx を起動し、プロセス ID で指定した実行中プロセスに接続できます。

```
$ dbx - process_id
```

プロセスの ID がわからない場合、dbx コマンドに `pgrep` コマンドを含めることで、ID を調べてプロセスに接続します。たとえば、次のようにします。

```
$ dbx - 'pgrep Freeway'
```

```
の読み込み中
ld.so.1 の読み込み中
libXm.so.4 の読み込み中
libgen.so.1 の読み込み中
libXt.so.4 の読み込み中
libX11.so.4 の読み込み中
libce.so.0 の読み込み中
libsocket.so.1 の読み込み中
libCrun.so.1 の読み込み中
libm.so.1 の読み込み中
libw.so.1 の読み込み中
libSM.so.6 の読み込み中
libICE.so.6 の読み込み中
libXext.so.0 の読み込み中
libnsl.so.1 の読み込み中
libdl.so.1 の読み込み中
libmp.so.2 の読み込み中
libc_psr.so.1 の読み込み中
プロセス 1855 に接続しました。
_libc_poll at 0xfef9437c で停止しました。
0xfef9437c: _libc_poll+0x0004:ta 0x8
現関数: main
48 XtAppMainLoop(app_context);
(dbx)
```

dbx コマンドと起動オプションの詳細については、[304 ページの「dbx コマンド」](#) および dbx(1) マニュアルページを参照するか、`dbx -h` と入力してください。

すでに dbx を実行している場合、`debug` コマンドにより、デバッグ対象プログラムを読み込むか、デバッグしているプログラムを別のプログラムに切り替えることができます。

```
(dbx) debug program_name
```

Java コードおよび C JNI コードまたは C++ JNI コードを含むプログラムを読み込むかそれに切り替える場合は、次を入力します。

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

すでに dbx を実行している場合、`debug` コマンドにより、dbx を実行中プロセスに接続することもできます。

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

Java コードと C JNI (Java Native Interface) コードまたは C++ JNI コードの混在する動作中のプロセスに dbx を接続するには、次のように入力します。

```
(dbx) debug program_name{.class | .jar} process_id
```

`debug` コマンドの詳細については、[307 ページの「debug コマンド」](#) を参照してください。

プログラムを dbx で実行する

dbx に最後に読み込んだプログラムを実行するには、`run` コマンドを使用します。引数を付けずに `run` コマンドを最初に入力すると、引数なしでプログラムが実行されます。引数を引き渡したりプログラムの入出力先を切り替えたりするには、次の構文を使用します。

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

たとえば、次のようにします。

```
(dbx) run -h -p < input > output
```

```
実行中: a.out
```

```
(プロセス id 1234)
```

```
実行完了。終了コードは 0 です。
```

```
(dbx)
```

Java コードを含むアプリケーションを実行する場合は、実行引数は、JVM ソフトウェアに渡されるのではなく、Java アプリケーションに渡されます。main クラス名を引数として含めないでください。

引数を付けずに `run` コマンドを繰り返し使用した場合、プログラムは前回の `run` コマンドの引数や入力先を使用します。 `rerun` コマンドを使用すれば、オプションをリセットできます。 `run` コマンドの詳細については、[352 ページの「run コマンド」](#)を参照してください。 `rerun` コマンドの詳細については、[349 ページの「rerun コマンド」](#)を参照してください。

アプリケーションは、最後まで実行され、正常に終了するかもしれません。ブレークポイントが設定されている場合には、ブレークポイントでアプリケーションが停止するはずですが、アプリケーションにバグが存在する場合は、メモリーフォルトまたはセグメント例外のため停止することがあります。

dbx を使用してプログラムをデバッグする

プログラムをデバッグする理由としては、次が考えられます。

- クラッシュする場所と理由をつきとめるため、クラッシュの原因をつきとめる方法としては、次があります。
 - プログラムを `dbx` で実行します。 `dbx` はクラッシュの発生場所をレポートします。
 - コアファイルを調べ、スタックトレースをチェックします ([34 ページの「コアファイルをチェックする」](#) および [38 ページの「呼び出しスタックを確認する」](#) を参照)。
- 次の方法で、プログラムが不正な実行結果を出力する原因を判定します。
 - ブレークポイントを設定して実行を停止することにより、プログラムの状態をチェックして変数の値を調べます ([36 ページの「ブレークポイントを設定する」](#) および [39 ページの「変数を調べる」](#) 参照)。
 - ソースコードを 1 行ずつステップ実行することによって、プログラムの状態がどのように変わっていくかを監視します ([37 ページの「プログラムをステップ実行する」](#) 参照)。
- メモリーリークやメモリー管理問題を見つける方法としては、次があります。実行時検査を行えば、メモリーアクセスエラーやメモリーリークエラーといった実行時エラーを確認できるとともに、メモリー使用状況を監視できます ([39 ページの「メモリーアクセス問題とメモリーリークを検出する」](#) を参照)。

コアファイルをチェックする

プログラムがどこでクラッシュするかをつきとめるには、プログラムがクラッシュしたときのメモリーイメージであるコアファイルを調べるとよいでしょう。 `where` コマンドを使用すれば ([386 ページの「where コマンド」](#) を参照)、コアをダンプしたときのプログラムの実行場所がわかります。

注-ネイティブコードのときと異なり、コアファイルから Java アプリケーションの状態情報を入手することはできません。

コアファイルをデバッグするには、次を入力します。

```
$ dbx program_name core
```

または

```
$ dbx - core
```

次の例では、プログラムがセグメント例外でクラッシュし、コアダンプが作成されています。ユーザーは dbx を起動し、コアファイルを読み込みます。次に、where コマンドを使用してスタックトレースを表示させます。これによって、ファイル foo.c の 9 行目でクラッシュが発生したことがわかります。

```
% dbx a.out core
a.out の読み込み中
core ファイルハンドラの読み込みに成功しました
ld.so.1 の読み込み中
libw.so.1 の読み込み中
libdl.so.1 の読み込み中
libc_psr.so.1 の読み込み中
プログラムはシグナル SEGV (フォルトのアドレスにマッピングしていません)
現関数: main
    9      printf("string '%s' is %d characters long\n", msg, strlen(msg));
(dbx) where
    [1] strlen(0x0, 0x0, 0xff337d24, 0x7efefeff, 0x81010100, 0xff0000)、アドレス
0xff2b6dec
=>[2] main(argc = 1, argv = 0xffbef39c)、"foo.c" の 9 行目
(dbx)
```

コアファイルのデバッグの詳細については、44 ページの「既存のコアファイルのデバッグ」を参照してください。呼び出しスタックの詳しい使い方については、38 ページの「呼び出しスタックを確認する」を参照してください。

注-プログラムが共有ライブラリと動的にリンクされている場合、できれば、コアファイルが作成されたオペレーティング環境でコアファイルをデバッグしてください。別のオペレーティング環境で作成されたコアファイルをデバッグする方法については、46 ページの「一致しないコアファイルのデバッグ」を参照してください。

ブレークポイントを設定する

ブレークポイントとは、一時的にプログラムの実行を停止し、コントロールを dbx に渡す場所のことです。バグが存在するのではないかとされるプログラム領域にブレークポイントを設定します。プログラムがクラッシュした場合、クラッシュが発生した個所をつきとめ、その部分の直前のコードにブレークポイントを設定します。

プログラムがブレークポイントで停止したとき、プログラムの状態と変数の値を調べることができます。dbx では、さまざまな種類のブレークポイントを設定できます(94 ページの「Control+C によってプロセスを停止する」を参照)。

もっとも単純なブレークポイントは、停止ブレークポイントです。停止ブレークポイントを使用すれば、関数や手続きの中で停止させることができます。たとえば、main 関数が呼び出されたときに停止させる方法は次のとおりです。

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

stop in コマンドの詳細については、97 ページの「関数に stop ブレークポイントを設定する」および 360 ページの「stop コマンド」を参照してください。

また、特定のソースコード行で停止するようにブレークポイントを設定することもできます。たとえば、ソースファイル t.c の 13 行目で停止させる方法は次のとおりです。

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

stop at コマンドの詳細については、96 ページの「ソースコードの特定の行に stop ブレークポイントを設定する」および 360 ページの「stop コマンド」を参照してください。

停止場所を確認するには、file コマンドで現在のファイルを設定し、list コマンドで停止場所とする関数を表示させます。次に、stop at コマンドを使用してソース行にブレークポイントを設定します。

```
(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
```

ブレークポイントで停止したプログラムの実行を続行するには、`cont` コマンドを使用します (91 ページの「プログラムを継続する」および 303 ページの「`cont` コマンド」を参照)。

現在のブレークポイントのリストを表示するには、`status` コマンドを使用します。

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

ここでプログラムを実行すれば、最初のブレークポイントでプログラムが停止します。

```
(dbx) run
...
main で停止しました 行番号 12 ファイル "t.c"
12      char *msg = "hello world\n";
```

プログラムをステップ実行する

ブレークポイントで停止したあと、プログラムを1ソース行ずつステップ実行すれば、あるべき正しい状態と実際の状態とを比較できます。それには、`step` コマンドと `next` コマンドを使用します。いずれのコマンドもプログラムのソース行を1行実行し、その行の実行が終了すると停止します。この2つのコマンドは、関数呼び出しが含まれているソース行の取り扱い方が違います。`step` コマンドは関数にステップインし、`next` コマンドは関数をステップオーバーします。

`step up` コマンドは、現在実行している関数が、自身を呼び出した関数に制御を戻すまで実行され続けます。

`step to` コマンドは、現在のソース行で指定されている関数にステップするか、関数が指定されていない場合は、現在のソース行のアセンブリコードにより最後に呼び出される関数にステップします。

`printf` のようなライブラリ関数をはじめとする一部の関数は `-g` を使ってコンパイルされていないことがあります。`dbx` は、このような関数にはステップインできません。このような場合、`step` と `next` は同じような動作を示します。

次は、`step` コマンドと `next` コマンド、および 36 ページの「ブレークポイントを設定する」に設定されたブレークポイントの使用例です。

```
(dbx) stop at 13
(3) stop at "t.c":13
(dbx) run
実行中: a.out
main で停止しました 行番号 13 ファイル "t.c"
```

```

    13          printit(msg);
(dbx) next
Hello world
main で停止しました 行番号 14 ファイル "t.c"
    14      }

(dbx) run
実行中: a.out
main で停止しました 行番号 13 ファイル "t.c"
    13          printit(msg);
(dbx) step
printit で停止しました 行番号 6 ファイル "t.c"
    6          printf("%s\n", msg);
(dbx) step up
Hello world
printit 戻り値 134524
main で停止しました 行番号 13 ファイル "t.c"
    13          printit(msg);
(dbx)

```

プログラムのステップ実行の詳細については、90 ページの「プログラムのステップ実行」を参照してください。step および next コマンドの詳細については、358 ページの「step コマンド」および 339 ページの「next コマンド」を参照してください。

呼び出しスタックを確認する

呼び出しスタックは、呼び出されたあと呼び出し側にまだ戻っていない、現在活動状態にあるルーチンすべてを示します。呼び出しスタックには、呼び出された順序で関数とその引数が一覧表示されます。プログラムフローのどこで実行が停止し、この地点までどのように実行が到達したのかが、スタックトレースに示されます。スタックトレースは、プログラムの状態を、もっとも簡潔に記述したものです。

スタックトレースを表示するには、where コマンドを使用します。

```

(dbx) stop in printf
(dbx) run
(dbx) where
[1] printf(0x10938, 0x20a84, 0x0, 0x0, 0x0, 0x0)、アドレス 0xef763418
=>[2] printit(msg = 0x20a84 "hello world\n"), "t.c" の 6 行目
[3] main(argc = 1, argv = 0xefff93c), "t.c" の 13 行目
(dbx)

```

-g オプションを使ってコンパイルされた関数の場合は引数の名前と型がわかっているので、正確な値が表示されます。デバッグ情報を持たない関数の場合、16 進数が引数として表示されます。これらの数字に意味があるとはかぎりません。たとえば、前述のスタックトレースのフレーム 1 は、\$i0 から \$i5 の SPARC 入力レジスタの

内容を示しています。37 ページの「プログラムをステップ実行する」の例の `printf` に引き渡された引数は 2 つだけなので、内容に意味があるレジスタは `$i0` から `$i1` までだけです。

`-g` オプションを使ってコンパイルされなかった関数の中でも停止することができます。このような関数の中で停止する場合、`dbx` は `-g` オプションを使ってコンパイルされた関数を持つフレームの中で最初のをスタック内で検索し (前述の例では `printit()`)、これに現在のスコープを設定します (72 ページの「プログラムスコープ」を参照)。これは、矢印記号 (`=>`) によって示されます。

呼び出しスタックの詳細については、108 ページの「イベント効率」を参照してください。

変数を調べる

プログラムの状態に関する十分な情報がスタックトレースに含まれているかもしれませんが、ほかの変数の値を調べる必要が生じることも考えられます。`print` コマンドは式を評価し、式の型に基づいて値を印刷します。次は、単純な C 式の例です。

```
(dbx) print msg
msg = 0x20a84 "Hello world"
(dbx) print msg[0]
msg[0] = 'h'
(dbx) print *msg
*msg = 'h'
(dbx) print &msg
&msg = 0xeffe8b4
```

データ変更ブレークポイントを使用すれば、変数と式の値を追跡できます (100 ページの「データ変更ブレークポイントを設定する」を参照)。たとえば、変数 `count` の値が変更されたときに実行を停止するには、次を入力します。

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

メモリーアクセス問題とメモリーリークを検出する

実行時検査は、メモリーアクセス検査、およびメモリー使用状況とリーク検査の 2 部で構成されます。アクセス検査は、デバッグ対象アプリケーションによるメモリーの使用がまちがっていないかどうかをチェックします。メモリー使用状況とメモリーリークの検査では、未処理のヒープ空間すべてを記録し、必要に応じて、またはプログラム終了時に、利用できるデータ空間の走査および参照なしの空間の確認を行います。

注- 実行時検査が実行できるのは、Solaris プラットフォームのみです。

メモリアクセス検査、およびメモリー使用状況とメモリーリークの検査は、`check` コマンドによって使用可能にします。メモリアクセス検査をオンにするには、次を入力します。

(dbx) **check -access**

メモリー使用状況とメモリーリークの検査をオンにするには、次を入力します。

(dbx) **check -memuse**

実行時検査をオンにしたら、プログラムを実行します。プログラムは正常に動作しますが、それぞれのメモリアクセスが発生する直前にその妥当性チェックが行われるため、動作速度は遅くなります。無効なアクセスを検出すると、dbxはそのエラーの種類と場所を表示します。現在のスタックトレースを取り出すには`where`などのdbxコマンド、変数を調べるには`print`コマンドを使用します。

注- Java コードおよびC JNI コードまたはC++ JNI コードが混在するアプリケーションには、実行時検査を使用できません。

実行時検査の詳細については、[第9章](#)を参照してください。

dbxを終了する

dbxセッションは、dbxを起動してから終了するまで継続されます。dbxセッションのあいだ、任意の数のプログラムを連続してデバッグできます。

dbxセッションを終了するには、dbxプロンプトで**quit**と入力します。

(dbx) **quit**

起動時にプロセスIDオプションを使用してデバッガを動作中のプロセスに接続した場合、デバッグセッションを終了しても、そのプロセスは終了しないで動作を続けます。すなわち、dbxはセッションを終了する前に自動的に`detach`コマンドを実行します。

dbxの終了の詳細については、55ページの「[デバッグセッションを終了する](#)」を参照してください。

dbx オンラインヘルプにアクセスする

dbx には、`help` コマンドでアクセスできるヘルプファイルが含まれています。

(dbx) **help**

dbx の起動

この章では、dbx デバッグセッションを開始、実行、保存、復元、および終了する方法について説明します。この章の内容は次のとおりです。

- 43 ページの「デバッグセッションを開始する」
- 44 ページの「既存のコアファイルのデバッグ」
- 48 ページの「プロセス ID の使用」
- 49 ページの「dbx 起動時シーケンス」
- 50 ページの「起動属性の設定」
- 51 ページの「デバッグのためのプログラムのコンパイル」
- 52 ページの「最適化コードのデバッグ」
- 55 ページの「デバッグセッションを終了する」
- 56 ページの「デバッグ実行の保存と復元」

デバッグセッションを開始する

dbx の起動方法は、デバッグの対象、現在の作業ディレクトリ、dbx で必要な実行内容、dbx の習熟度、および dbx 環境変数を設定したかどうかによって異なります。

dbx セッションを開始するもっとも簡単な方法は、dbx コマンドをシェルプロンプトで入力する方法です。

```
$ dbx
```

シェルから dbx を起動し、デバッグするプログラムを読み込むには、次のように入力します。

```
$ dbx program_name
```

dbx を起動して、Java コードおよび C JNI コードまたは C++ JNI コードが混在するプログラムを読み込むには、次のように入力します。

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

Sun Studio ソフトウェアには、32 ビットプログラムのみをデバッグ可能な 32 ビット dbx、および 32 ビットプログラムと 64 ビットプログラムの両方をデバッグ可能な 64 ビット dbx の、2 つの dbx バイナリが付属しています。dbx を起動すると、どちらのバイナリを実行すべきか自動的に判定されます。64 ビット OS では、デフォルトは 64 ビット dbx です。64 ビット OS で 32 ビット dbx を起動するには、dbx コマンドを `-x exec32` オプション付きで使用するか(306 ページの「オプション」参照)、環境変数 `_DBX_EXEC_32` を設定します。

注-Linux OS では、64 ビットの dbx で 32 ビットプログラムをデバッグできません。Linux OS で 32 ビットプログラムをデバッグするには、32 ビットの dbx を起動します。

注-64 ビット Linux OS で 32 ビット dbx を使用する場合は、`debug` コマンドを使用しないでください。デバッグによって 64 ビットプログラムが実行される場合は、環境変数 `follow_fork_mode` を子に設定します。64 ビットプログラムをデバッグするには、dbx を終了してから 64 ビット dbx を起動します。

dbx コマンドおよび起動オプションについての詳細は、304 ページの「dbx コマンド」および dbx(1) のマニュアルページを参照してください。

既存のコアファイルのデバッグ

コアダンプしたプログラムが共有ライブラリと動的にリンクしている場合、それが作成された同じオペレーティング環境でコアファイルをデバッグすることが重要です。dbx では、一致しないコアファイル(たとえば、バージョンまたはパッチレベルの異なる Solaris オペレーティングシステムで生成されたコアファイル)のデバッグに対しサポートが制限されます。

注-ネイティブコードのときと異なり、コアファイルから Java アプリケーションの状態情報を入手することはできません。

同じオペレーティング環境でのコアファイルのデバッグ

コアファイルをデバッグするには、次のように入力します。

```
$ dbx program_name core
```

dbx がすでに起動していれば、`debug` コマンドを使用してコアファイルをデバッグすることもできます。

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

プログラム名として `-` を指定すると、`dbx` はコアファイルからプログラム名を抽出します。実行可能ファイルのフルパス名をコアファイルから抽出できない場合は、実行可能ファイルを特定できないことがあります。この場合は、`dbx` でコアファイルを読み込むときに、バイナリの完全なパス名を指定します。

コアファイルが現在のディレクトリに存在しない場合、パス名を指定できます (`/tmp/core` など)。

プログラムがコアをダンプしたときにどこで実行されていたかを確認するには、`where` コマンド (386 ページの「[where コマンド](#)」を参照) を使用してください。

コアファイルをデバッグする場合、変数と式を評価して、プログラムがクラッシュした時点での値を確認することもできますが、関数呼び出しを行なった式を評価することはできません。ステップ実行したりブレークポイントを設定することはできません。

コアファイルが切り捨てられている場合

コアファイルの読み込みに問題がある場合は、コアファイルが切り捨てられているかどうかを確認してください。コアファイルの生成時に、コアファイルの最大サイズの設定が小さすぎる場合は、コアファイルが切り捨てられ、`dbx` で読み込めないことがあります。C シェルでは、`limit` コマンドを使用して、コアファイルの最大サイズを設定することができます (`limit(1)` マニュアルページを参照)。Bourne シェルおよび Korn シェルでは、`ulimit` コマンドを使用します (`limit(1)` マニュアルページを参照)。シェルの起動ファイルでコアファイルのサイズの上限を変更してその設定を有効にし、コアファイルを生成したプログラムを再実行すれば、完全なコアファイルが生成されます。

コアファイルが不完全で、スタックセグメントが欠落している場合、スタックのトレース情報は利用できません。実行時リンカー情報が欠落している場合、ロードオブジェクトのリストは利用できません。この場合は、`librtld_db.so` が初期化されていないというエラーメッセージが表示されます。LWP のリストが欠落している場合、スレッド情報、`lwp` 情報、またはスタックトレース情報は利用できません。`where` コマンドを実行すると、プログラムがアクティブでないというエラーメッセージが表示されます。

一致しないコアファイルのデバッグ

特定のシステム(コアホスト)で作成されたコアファイルを、デバッグのためにそのファイルを別のマシン(dbx ホスト)に読み込む場合があります。この場合、ライブラリに関する2つの問題が発生する可能性があります。

- コアホストのプログラムで使用される共有ライブラリが dbx ホストのライブラリと異なる場合があります。ライブラリに関して正しいスタックトレースを取得するには、dbx ホストでもオリジナルのライブラリを利用できなくてはなりません。
- dbx は、システム上の実行時リンカーとスレッドのライブラリについて実装詳細をわかりやすくするために、`/usr/lib`に配置されているライブラリを使用します。また、dbx が実行時リンカーのデータ構造とスレッドのデータ構造を理解できるように、コアホストからそれらのシステムライブラリを提供する必要性が出てくることもあります。

ユーザーライブラリとシステムライブラリは、パッチや主要な Solaris オペレーティング環境のアップグレードで変更できるため、収集したコアファイルで dbx を実行する前にパッチをインストールした場合など、この問題が同一ホストでも発生する可能性があります。

dbx は、一致しないコアファイルを読み込むと、次のエラーメッセージを1つ以上表示することがあります。

```
dbx: コアファイル読み取りエラー: アドレス 0xff3dd1bc は利用できません
dbx: 警告: librtld_db.so.1 を初期化できませんでした -- trying libDP_rtld_db.so
dbx: 1 のスレッド情報を取得できません -- 一般的な libthread_db.so エラー
dbx: レジスタをフェッチしようとして失敗しました - スタックが破壊されました
dbx: (0xff363430) からのレジスタの読み取りに失敗しました -- デバッガは失敗しました
```

共有ライブラリ問題の回避

▼ ライブラリ問題を回避し、一致しないコアファイルを dbx でデバッグするには、次の手順を実行します。

- 1 dbx 環境変数 `core_lo_pathmap` を on に設定します。
- 2 `pathmap` コマンドを使用して、コアファイルの正しいライブラリの配置場所を dbx に伝えます。

- 3 debug コマンドを使用して、プログラムとコアファイルを読み込みます。たとえば、コアホストのルートパーティションが NFS を介してエクスポートされており、dbx ホストマシンの `/net/core-host/` からアクセスできると仮定した場合、次のコマンドを使用して、プログラム `prog` とコアファイル `prog.core` をデバッグのために読み込みます。

```
(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
```

コアホストのルートパーティションをエクスポートしていない場合、手動でライブラリをコピーする必要があります。シンボリックリンクを再作成する必要はありません(たとえば、`libc.so` から `libc.so.1` へのリンクを作成する必要はありません。`libc.so.1` が利用可能であることだけを確認してください)。

注意点

一致しないコアファイルをデバッグする際に、次の点に注意してください。

- `pathmap` コマンドは `/` のパスマップを認識しないため、次のコマンドを使用できません。

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

- `pathmap` コマンドの単一引数モードは、ロードオブジェクトのパス名を使用すると機能しません。そのため、2つの引数をとる `form-path to-path` モードを使用してください。
- dbx ホストがコアホストと同一のバージョンまたはコアホストより最近のバージョンの Solaris オペレーティング環境を有している場合、コアファイルのデバッグが良好に機能する傾向にあります。ただし、これは必須ではありません。
- 必要となるシステムライブラリを次に示します。

- 実行時リンカーの場合:

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

- スレッドライブラリの場合 (使用しているスレッドの実装に依存します):

```
/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` ファイルは、Solaris 8 オペレーティング環境で dbx を実行していて、別の `libthread` ライブラリを使用している場合にだけ適用されます。

dbx を 64 ビット対応バージョンの Solaris OS で実行している場合は、これらのシステムライブラリはターゲットプログラムではなく dbx の一部として読み込まれて使用されるため、64 ビットバージョンの `xxx_db.so` ライブラリが必要です。

`ld.so.1` ライブラリは、`libc.so` などのライブラリのコアファイルイメージの一部であるため、コアファイルを作成したプログラムに一致する 32 ビットまたは 64 ビットの `ld.so.1` ライブラリが必要です。

- スレッド化されたプログラムからコアファイルを調べていて、および `where` コマンドがスタックを表示しない場合、`lwp` コマンドを使用してみてください。次に例を示します。

```
(dbx) where
現スレッド: t@0
[1] 0x0(), at 0xffffffff
(dbx) lwps
o>l@1 シグナル SIGSEGV 現在の関数 _sigfillset()
(dbx) lwp l@1
(dbx) where
=>[1] _sigfillset(), "lo.c" の 2 行目
    [2] _liblwp_init(0xff36291c, 0xff2f9740, ...
    [3] _init(0x0, 0xff3e2658, 0x1, ...
...

```

`lwp` コマンドの `-setfp` および `-resetfp` オプションは、LWP のフレームポインタ (fp) が壊れているときに便利です。これらのオプションは、`assign $fp=...` が利用できないコアファイルのデバッグ時に機能します。

スレッドスタックの欠如は、`thread_db.so.1` に問題があることを示している場合があります。そのため、コアホストから正しい `libthread_db.so.1` ライブラリをコピーしてください。

プロセスIDの使用

動作中のプロセスを dbx に接続できます。dbx コマンドに引数としてプロセス ID を指定します。

```
$ dbx program_name process_id
```

Java™ コードと C JNI (Java Native Interface) コードまたは C++ JNI コードの混在する動作中のプロセスに dbx を接続するには、次のように入力します。

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

プログラムの名前を知らなくても、その ID を使用してプロセスに接続できます。

```
$ dbx - process_id
```

この場合、dbx はプログラムの名前を認識できないため、run コマンドの中でそのプロセスに引数を渡すことはできません。

詳細については、88 ページの「動作中のプロセスに dbx を接続する」を参照してください。

dbx 起動時シーケンス

dbx を起動するときに、-S オプションを指定していない場合は、dbx は `installation_directory/lib` ディレクトリでインストール時の起動ファイル `.dbxrc` を検索します (デフォルトの `installation_directory` は、Solaris プラットフォームでは `/opt/SUNWspro`、Linux プラットフォームでは `/opt/sun/sunstudio10u1` です)。Sun Studio ソフトウェアがデフォルトのディレクトリ `installation_directory` にインストールされていない場合、`.dbxrc` ファイルへのパスは、dbx 実行可能ファイルへのパスから取得します。

dbx は、`.dbxrc` ファイルを現在のディレクトリ、`$HOME` の順で検索します。-s を使用して、別の起動ファイルを明示的に指定することもできます。詳細については、61 ページの「dbx 初期化ファイルの使用」を参照してください。

起動ファイルには、任意の dbx コマンドが含まれることがあり、一般に `alias` コマンド、`dbxenv` コマンド、`pathmap` コマンド、および Korn シェル関数定義が含まれます。ただし、特定のコマンドは、プログラムが読み込まれていること、またはプロセスが接続されていることを要求します。すべての起動ファイルは、プログラムまたはプロセスが読み込まれる前に読み込まれます。さらに起動ファイルは、`source` または `.` (ピリオド) コマンドを使用することにより、その他のファイルのソースとなることもできます。起動ファイルを使用して、ほかの dbx オプションを設定することもできます。

dbx がプログラム情報を読み込むと、`Filename` の読み込み中などの一連のメッセージを出力します。

プログラムが読み込みを終了すると、dbx は準備状態となり、プログラム (C、C++ については、`main()`、Fortran 95 については、`MAIN()`) のメインブロックを表示します。一般に、ブレークポイントを設定し (例: `stop in main`)、C プログラムに対し `run` コマンドを実行します。

起動属性の設定

`pathmap` コマンド、`dbxenv` コマンド、および `alias` コマンドを使用して、`dbx` セッションに対する起動プロパティーを設定できます。

デバッグ時ディレクトリへのコンパイル時ディレクトリのマッピング

デフォルトでは、`dbx` はプログラムがコンパイルされたディレクトリに、デバッグ中のプログラムに関連するソースファイルがないかを探します。ソースファイルまたはオブジェクトファイルがそのディレクトリにないか、または使用中のマシンが同じパス名を使用していない場合は、`dbx` にその場所を知らせる必要があります。

ソースファイルまたはオブジェクトファイルを移動した場合、その新しい位置を検索パスに追加できます。`pathmap` コマンドは、ファイルシステムの現在のディレクトリと実行可能イメージ内の名前とのマッピングを作成します。このマッピングは、ソースパスとオブジェクトファイルパスに適用されます。

一般的なパスマップは、各自の `.dbxrc` ファイルに追加する必要があります。

ディレクトリ *from* からディレクトリ *to* への新しいマッピングを確立するには、次のように入力します。

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

`-c` を使用すると、このマッピングは、現在の作業ディレクトリにも適用されます。

`pathmap` コマンドは、ホストによってベースパスの異なる、自動マウントされた明示的な NFS マウントファイルシステムを扱う場合にも役立ちます。`-c` は、現在の作業ディレクトリが自動マウントされたファイルシステム上で不正確なオートマウンタが原因で起こる問題を解決する場合に使用してください。

`/tmp_mnt` と `/` のマッピングはデフォルトで存在します。

詳細については、[342 ページの「pathmap コマンド」](#)を参照してください。

dbx 環境変数の設定

`dbxenv` コマンドを使用すると、`dbx` カスタマイズ変数を表示または設定できます。`dbxenv` コマンドは、各自の `.dbxrc` ファイルに入れることができます。変数を表示するには、次のように入力します。

```
$ dbxenv
```

dbx 環境変数は設定することもできます。 .dbxrc ファイルおよびこれら変数の設定方法については、58 ページの「[replay を使用した保存と復元](#)」を参照してください。

詳細については、62 ページの「[dbx 環境変数の設定](#)」および 307 ページの「[dbxenv コマンド](#)」を参照してください。

ユーザー自身の dbx コマンドを作成

kalias または dalias コマンドを使用して、ユーザー自身の dbx コマンドを作成することができます。詳細については、303 ページの「[dalias コマンド](#)」を参照してください。

デバッグのためのプログラムのコンパイル

プログラムは -g または -g0 オプションでコンパイルし、dbx でデバッグする準備をします。

-g オプションでコンパイル

-g オプションは、コンパイル時にデバッグ情報を生成するよう、コンパイラに指示します。

たとえば、C++ を使用してコンパイルするには、次のように入力します。

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

C++ では、-g オプションは、デバッグをオンにし、関数のインライン化をオフにします。-g0 (ゼロ) オプションは、デバッグをオンにし、関数のインライン化には影響を与えません。-g0 オプションでインライン関数をデバッグすることはできません。-g0 オプションは、リンクタイムおよび dbx の起動時間を大幅に削減します (プログラムによるインライン関数の使用に依存)。

dbx で使用するため、最適化コードをコンパイルするには、-O (大文字 O) と -g オプションの両方でソースコードをコンパイルします。

別のデバッグファイルの使用

dbx により、実行可能ファイルから別のデバッグファイルにデバッグ情報をコピーし、実行可能ファイルからその情報をストリップし、これらの 2 ファイル間にリンクを作成するために、Linux プラットフォームでは objcopy コマンド、Solaris プラットフォームでは gobjcopy コマンドのオプションを使用できます。

dbx は、次の順序で別のデバッグファイルを検索し、最初に見つかったファイルからデバッグ情報を読み取ります。

- 実行可能ファイルを含むディレクトリ。
- 実行可能ファイルを含むディレクトリ内の `debug` という名前のサブディレクトリ。
- グローバルデバッグファイルディレクトリのサブディレクトリ。dbx 環境変数 `debug_file_directory` がディレクトリのパス名に設定されている場合は、このサブディレクトリを表示したり変更したりできます。環境変数のデフォルト値は、`/usr/lib/debug` です。

たとえば、実行可能ファイル `a.out` に対して別のデバッグファイルを作成するには、次のことを行います。

▼ 別のデバッグファイルの作成

- 1 デバッグ情報を含む、`a.out.debug` という名前の別のデバッグファイルを作成します。
- 2 `a.out` からデバッグ情報をストリップします。
- 3 2つのファイル間にリンクを作成します。**Solaris** プラットフォームの場合、`gobjcopy` コマンドを使用します。**Linux** プラットフォームの場合、`objcopy` コマンドを使用します。

Linux プラットフォームでは、`objcopy` コマンドの `-help` オプションを使用して、プラットフォームで `-add-gnu-debuglink` オプションがサポートされているかどうかを調べることができます。`objcopy` コマンドの `-only-keep-debug` オプションは、`a.out.debug` を完全な実行可能ファイルにすることができる `cp a.out a.out.debug` コマンドに置き換えることができます。

最適化コードのデバッグ

dbx は、最適化コードのデバッグを部分的にサポートしています。サポートの範囲は、プログラムのコンパイル方法によって大幅に異なります。

最適化コードを分析する場合、次のことができます。

- 関数起動時に実行を停止する (`stop in function` コマンド)
- 引数を評価、表示、または変更する
- 大域変数、局所変数、または静的変数を、評価、表示、または変更する
- ある行から別の行へシングルステップする (`next` または `step` コマンド)

最適化によりプログラムがコンパイルされ、同時に (`-O -g` オプションを使用して) デバッグが有効になると、dbx は制限されたモードで操作します。

どのような環境下でどのコンパイラがどの種類のシンボリック情報を発行したかについての詳細は、不安定なインタフェースとみなされ、リリース移行時に変更される可能性があります。

ソース行についての情報が提供されます。ただし最適化プログラムについては、1つのソース行に対するコードが複数の異なる場所で表示される場合があります。そのため、ソース行ごとにプログラムをステップすると、オプティマイザによってどのようにコードがスケジュールされたかに依存して、ソースファイルの周りで現在の行のジャンプが発生します。

末尾呼び出しを最適化すると、関数の最後の有効な操作が別の関数への呼び出しである場合、スタックフレームがなくなります。

通常、パラメータ、局所変数、および大域変数のシンボリック情報は、最適化プログラムで利用できます。構造体、共用体、およびC++クラスの型情報と局所変数、大域変数、およびパラメータの型と名前を利用できるはずですが、C++コンパイラは、局所変数のシンボリック型情報を提供しません。ただし、Cコンパイラは、これらの情報を提供します。

パラメータと局所変数の位置に関する情報は、最適化コード内で欠落していることがよくあります。コンパイラがDWRP位置リストを生成する場合、dbxはこの情報を使用して局所変数とパラメータの値を出力します。最適化関数の最初の命令で停止する場合、パラメータの値はABI適合レジスタまたはスタックの位置に格納されるため、dbxはその値を出力できます。局所変数の値を調べるには、dbxの環境変数`optim_local_vars`の設定が必要な場合もあります。

デバッグ情報内の単純位置は、1つのレジスタまたはスタックの位置を示していません。位置リストは、コード内の異なるところにある変数の別の位置を記述しており、最適化コードを記述するのにより適しています。

Sun Studio 12以降のリリースのSun Studioコンパイラは、位置リストを生成しませんが、一部の最適化コード用に単純位置を生成します。新しいバージョンのGNUコンパイラは、フレームポインタおよび一部のパラメータや局所変数を記述するのに位置リストを使用します。コンパイラがこの情報を記録する方法は、リリースによって異なる可能性があります。

最後のレジスタからメモリーへのストアがまだ発生していない場合、値は正確でない可能性があります。大域変数は表示したり、値を割り当てたりできます。

Sun Studio 11パフォーマンスアナライザのマニュアルの8章には、最適化プログラムのデバッグ時に役立つ、コンパイラの最適化に関する情報が含まれています。

OpenMPプログラムの場合、`-xopenmp=noopt` オプションを使用してコンパイルすると、コンパイラは最適化を適用しないように指示されます。ただし、オプティマイザはOpenMp指令を実装するために引き続きコードを処理するので、記述された問題のいくつかは、`-xopenmp=noopt` を使用してコンパイルされたプログラムで発生する可能性があります。

-g オプションを使用しないでコンパイルされたコード

ほとんどのデバッグサポートでは、-g を使用してプログラムをコンパイルすることを要求していますが、dbx では、-g を使用しないでコンパイルされたコードに対し、次のレベルのサポートを提供しています。

- バックトレース (dbx where コマンド)
- 関数の呼び出し (ただし、パラメータチェックなし)
- 大域変数のチェック

ただし、dbx では、-g オプションでコンパイルされたコードを除いては、ソースコードを表示できません。これは、strip -x が適用されたコードについてもあてはまります。

dbx を完全にサポートするために -g オプションを必要とする共有ライブラリ

完全なサポートを提供するためには、共有ライブラリも -g オプションを使用してコンパイルする必要があります。-g オプションを使用してコンパイルされていない共有ライブラリモジュールを使用してプログラムを作成した場合でも、そのプログラムをデバッグすることはできます。ただし、これらのライブラリモジュールに関する情報が生成されていないため、dbx の機能を完全に使用することはできません。

完全にストリップされたプログラム

dbx は、完全にストリップされた (制御データなどが取り除かれた) プログラムをデバッグすることができます。これらのプログラムには、プログラムをデバッグするために使用できる情報がいくつか含まれますが、外部から識別できる関数しか使用できません。一部の実行時検査は、ストリップされたプログラムまたはロードオブジェクトに対して動作します。メモリー使用状況検査およびアクセス検査は、strip -x でストリップされたコードに対して動作します。ただし、strip でストリップされたコードに対しては動作しません。

デバッグセッションを終了する

dbx の起動から終了までが 1 つの dbx セッションになります。1 つの dbx セッション中に、任意の数のプログラムを連続してデバッグできます。

dbx セッションを終了するには、dbx プロンプトで `quit` と入力します。

(dbx) **quit**

起動時にプロセス ID オプションを使用してデバッガを動作中のプロセスに接続した場合、デバッグセッションを終了しても、そのプロセスは終了しないで動作を続けます。すなわち、dbx はセッションを終了する前に自動的に `detach` コマンドを実行します。

プロセス実行の停止

Ctrl+C を使用すると、dbx を終了しないでいつでもプロセスの実行を停止できます。

dbx からのプロセスの切り離し

dbx をあるプロセスに接続した場合、`detach` コマンドを使用すると、そのプロセスおよび dbx セッションを終了せずに、そのプロセスを dbx から切り離すことができます。

プロセスを終了せずに dbx から切り離すには、次のように入力します。

(dbx) **detach**

dbx が占有アクセスしているときにブロックされるほかの /proc ベースのデバッグツールを一時的に適用している間に、プロセスを切り離して停止状態にすることができます。詳細については、[90 ページの「プロセスから dbx を切り離す」](#) を参照してください。

`detach` コマンドの詳細については、[311 ページの「detach コマンド」](#) を参照してください。

セッションを終了せずにプログラムを終了する

dbx の `kill` コマンドは、プロセスを終了するとともに、現在のプロセスのデバッグも終了します。ただし、`kill` コマンドは、dbx セッション自体を維持したまま、dbx で別のプログラムをデバッグできる状態にします。

プログラムを終了すると、dbx を終了しないで、デバッグ中のプログラムの残りを除去することができます。

dbx で実行中のプログラムを終了するには、次のように入力します。

```
(dbx) kill
```

詳細については、[327 ページの「kill コマンド」](#)を参照してください。

デバッグ実行の保存と復元

dbx には、コマンドデバッグ実行の全部または一部を保存して、それをあとで再現するためのコマンドが3つあります。

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

save コマンドの使用

save コマンドは、直前に実行された run コマンド、rerun コマンド、または debug コマンドから save コマンドまでに発行されたデバッグコマンドをすべてファイルに保存します。このデバッグセッションのセグメントは、「デバッグ実行」と呼ばれます。

save コマンドは、発行されたデバッグコマンドのリスト以外のものも保存します。実行開始時のプログラムの状態に関するデバッグ情報、つまり、ブレークポイント、表示リストなども保存されます。保存された実行を復元するとき、dbx は、保存ファイル内にあるこれらの情報を使用します。

デバッグ実行の一部、つまり、入力されたコマンドのうち指定する数だけ最後から除いたものを保存することもできます。



continue	最後の2ステップを除く	continue
next		next
next		next
step		step
next		next
save		save-2

保存する実行の終了位置がわからない場合は、`history` コマンドを使用して、セッション開始以降に発行されたデバッグコマンドのリストを確認してください。

注-デフォルトで、`save` コマンドは特別な保存ファイルへ情報を書き込みます。デバッグ実行後に復元可能なファイルへ保存する場合は、`save` コマンドでファイル名を指定することができます。57 ページの「一連のデバッグ実行をチェックポイントとして保存する」を参照してください。

`save` コマンドまでのデバッグ実行のすべてを保存するには、次のように入力します。

```
(dbx) save
```

デバッグ実行の一部を保存するには、`save number` コマンドを使用します。`number` は、`save` コマンドの直前の、保存しないコマンドの数を示します。

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

一連のデバッグ実行をチェックポイントとして保存する

ファイル名を指定しないでデバッグ実行を保存すると、情報は特殊な保存ファイルに書き込まれます。保存のたびに、`dbx` はこの保存ファイルを上書きします。しかし、ファイル名引数を `save` コマンドに指定すると、あるデバッグ実行をこのファイル名に保存後、別のデバッグ実行を保存しても、前の内容を復元することができます。

一連の実行を保存すると、1組のチェックポイントが与えられます。各チェックポイントは、セッションのさらにとから始まります。保存されたこれらの実行は任意に復元して続行し、さらに、以前の実行で保存されたプログラム位置と状態に `dbx` をリセットすることができます。

デバッグ実行を、デフォルトの保存ファイル以外のファイルに保存するには、次のように入力します。

```
(dbx) save filename
```

保存された実行の復元

実行を保存したら、`restore` コマンドを使用して実行を復元できます。dbx は、保存ファイル内の情報を使用します。実行を復元すると、dbx は、まず内部状態をその実行の開始時の状態にリセットしてから、保存された実行内の各デバッグコマンドを再発行します。

注 - `source` コマンドは、ファイル内に保存された一連のコマンドを再発行しますが、dbx の状態をリセットはしません。これは、現在のプログラム位置からコマンドの一覧を再発行するだけです。

保存された実行の正確な復元に必要な条件

保存されたデバッグ実行を正確に復元するには、`run` タイプコマンドへの引数、手動入力、およびファイル入力などの、実行での入力すべてが正確に同じである必要があります。

注 - セグメントを保存してから、`restore` を実行する前に `run`、`rerun`、または `debug` コマンドを発行すると、`restore` は 2 番目の引数を使用して、`run`、`rerun`、または `debug` コマンドをあとで保存します。これらの引数が異なる場合、正確な復元が得られない可能性があります。

保存されたデバッグ実行を復元するには、次のように入力します。

```
(dbx) restore
```

デバッグ実行を、デフォルトの保存ファイル以外のファイルに保存するには、次のように入力します。

```
(dbx) restore filename
```

replay を使用した保存と復元

`replay` コマンドは組み合わせのコマンドで、`save -1` に続けて `restore` を発行するのと同じです。`replay` コマンドは負の `number` 引数をとります。これは、コマンドの `save`

部分に渡されるものです。デフォルトで、*-number* の値は *-1* になるため、`replay` は取り消しコマンドとして働き、直前に発行されたコマンドまで (ただしこのコマンドは除く) の前回の実行を復元します。

現在のデバッグ実行から、最後に発行されたデバッグコマンドを除くものを再現するには、次のように入力します。

(dbx) **replay**

現在のデバッグ実行を再現して、指定のコマンドの前で実行を停止するには、`dbx` の `replay` コマンドを使用します。ここで、*number* は、最後のデバッグコマンドから数えていくつめのコマンドで停止するかその数を示します。

(dbx) **replay** *-number*

dbx のカスタマイズ

この章では、デバッグ環境の特定の属性をカスタマイズするために使用できる dbx 環境変数と、初期化ファイル `.dbxrc` を使用してカスタマイズの内容をセッション間で保存する方法について説明します。

この章の内容は次のとおりです。

- 61 ページの「dbx 初期化ファイルの使用」
- 62 ページの「dbx 環境変数の設定」
- 68 ページの「dbx 環境変数および Korn シェル」

dbx 初期化ファイルの使用

dbx の起動時に実行される dbx コマンドは、すべて初期化ファイルに保存されます。通常このファイルには、デバッグ環境をカスタマイズするコマンドを記述しますが、任意の dbx コマンドを記述することもできます。デバッグ中に dbx をコマンド行からカスタマイズする場合、これらの設定値は、現在デバッグ中のセッションにしか適用されないことに注意してください。

注-.dbxrc ファイルは、コードを実行するコマンドを含むことはできません。ただし、それらのコマンドをファイルに置き、dbx source コマンドを使用して、そのファイルでコマンドを実行することは可能です。

dbx 起動時の検索順序は次のとおりです。

1. インストールディレクトリ (-s オプションを dbx コマンドに指定しない場合)
`/installation_directory/lib/dbxrc` (デフォルトの `installation_directory` は Solaris プラットフォームでは `/opt/SUNWspro`、Linux プラットフォームでは `/opt/sun/sunstudio12` となります)。Sun Studio ソフトウェアがデフォルトの `installation_directory` にインストールされていない場合、`.dbxrc` ファイルへのパスは、dbx 実行可能ファイルへのパスから取得します。

2. 現在のディレクトリ `./dbxrc`
3. ホームディレクトリ `$HOME/.dbxrc`

.dbxrc ファイルの作成

共通のカスタマイズおよびエイリアスを含む `.dbxrc` ファイルを作成するには、次のように入力します。

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

テキストエディタを使用して、結果的にできたファイルをカスタマイズすることにより、実行したいエントリをコメント解除することができます。

初期化ファイル

次に `.dbxrc` ファイルの例を示します。

```
dbxenv input_case_sensitive false
catch FPE
```

最初の行は、大文字/小文字区別の制御のデフォルト設定を変更するものです。

- `dbxenv` は、`dbx` 環境変数の設定に使用するコマンドです (`dbx` 環境変数の種類については、62 ページの「[dbx 環境変数の設定](#)」を参照してください)。
- `input_case_sensitive` は、大文字/小文字の区別を制御するための `dbx` 環境変数です。
- `false` は `input_case_sensitive` の設定値です。

次の行はデバッグコマンドの `catch` です。このコマンドは `dbx` が応答するデフォルトのシグナルの一覧に、システムシグナル `FPE` を追加して、プログラムを停止します。

dbx 環境変数の設定

`dbxenv` コマンドを使用して `dbx` 環境変数を設定することにより、`dbx` セッションをカスタマイズすることができます。

特定の変数の値を表示するには、次のように入力します。

```
(dbx) dbxenv variable
```

すべての変数とその値を表示するには、次のように入力します。

(dbx) **dbxenv**

変数の値を設定するには、次のように入力します。

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

表 3-1 に、設定可能なすべての dbx 環境変数を示します。

表 3-1 dbx 環境変数

dbx 環境変数	dbx 環境変数の機能
array_bounds_check on off	パラメータを on に設定すると、配列の上下限を検査します。 デフォルト値は on です。
CLASSPATHX	独自のクラスローダーを使用する場合に、そのローダーが読み込む Java クラスファイルのパスを指定することができます。
core_lo_pathmap on off	dbx が一致しないコアファイルの正しいライブラリを検索するためにパスマップ設定を使用するかどうかを制御します。デフォルト値は off です。
debug_file_directory	大域デバッグファイルディレクトリを設定します。デフォルト値は /usr/lib/debug です。
disassembler_version autodetect v8 v9 x86_32 x86-64	SPARC プラットフォームの SPARC V8 または V9 の dbx の組み込み逆アセンブラのバージョンを設定します。デフォルト値は autodetect で、a.out が実行されているマシンのタイプに従って、動的にモードを設定します。 x86 プラットフォームでは autodetect だけが有効です。
event_safety	dbx を安全でないイベント使用に対して保護します。
fix_verbose on off	fix 中のコンパイル行出力を制御します。デフォルト値は off です。
follow_fork_inherit on off	子プロセスを生成したあと、ブレークポイントを継承するかどうかを設定します。デフォルト値は off です。
follow_fork_mode parent child both ask	現在のプロセスが fork、vfork、fork1 を実行しフォークした場合、どのプロセスを追跡するかを決定します。parent に設定すると親を追跡します。child に設定すると子を追跡します。both に設定すると、親プロセスをアクティブ状態にして子を追跡します。ask に設定すると、フォークが検出されるたびに、追跡するプロセスを尋ねます。デフォルト値は parent です。

表 3-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
follow_fork_mode_inner unset parent child both	フォークが検出されたあと、follow_fork_mode が ask に設定されていて、停止を選んだときの設定です。この変数を設定すると、cont -follow を使用する必要はありません。
input_case_sensitive autodetect true false	autodetect に設定すると、デバッグ対象の言語に従って大文字/小文字の区別が自動的に選択されます。Fortran 95 ファイルの場合は false、そうでない場合は true です。true の場合は、変数と関数名では大文字/小文字が区別されます。変数と関数名以外では、大文字/小文字は区別されません。 デフォルト値は autodetect です。
JAVASRCPATH	dbx が Java ソースファイルを検索するディレクトリを指定します。
jdbx_mode java jni native	現在の dbx モードを設定します。次に設定できます。 java, jni, native。
jvm_invocation	jvm_invocation 環境変数を使って、JVM™ ソフトウェアの起動方法をカスタマイズすることができます (JVM は Java virtual machine の略語で、Java™ プラットフォーム用の仮想マシンを意味する)。詳細については、223 ページの「JVM ソフトウェアの起動方法のカスタマイズ」を参照してください。
language_mode autodetect main c c++ fortran fortran90	式の解析と評価に使用する言語を制御します。 <ul style="list-style-type: none"> ■ autodetect は、式の言語を現在のファイルの言語に設定します。複数の言語が混在するプログラムをデバッグする場合に有用です (デフォルト)。 ■ main は、式の言語をプログラム内の主ルーチンの言語に指定します。単一言語のデバッグをする場合に有用です。 ■ c、c++、fortran、または fortran90 は、式の言語を選択した言語に設定します。
mt_scalable on off	有効の場合、dbx はリソースの使用方法において保守的となり、300 個以上の LWP を持つプロセスのデバッグが可能ですが、下方サイドは大幅に速度が減少します。デフォルト値は off です。
optim_local_vars on off	有効にすると、最適化された関数で局所変数を評価できます。デフォルト値は off です。
output_auto_flush on off	呼び出しが行われるたびに、fflush() を自動的に呼び出します。デフォルト値は on です。

表 3-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
<code>output_base 8 10 16 automatic</code>	整数の定数を出力するためのデフォルト基数。デフォルト値は <code>automatic</code> です (ポインタは 16 進文字、その他すべては 10 進)。
<code>output_class_prefix on off</code>	クラスメンバーの値または宣言を表示するとき、その前にクラス名を付けるかどうかを制御します。on の場合は、クラスメンバーの前にクラス名が付けられます。デフォルト値は on です。
<code>output_dynamic_type on off</code>	on の場合、ウォッチポイントの出力および表示のデフォルトを <code>-d</code> にします。デフォルト値は off です。
<code>output_inherited_members on off</code>	on の場合、出力、表示、および検査のデフォルト出力を <code>-r</code> にします。デフォルト値は off です。
<code>output_list_size num</code>	<code>list</code> コマンドで出力する行のデフォルト数を指定します。デフォルト値は 10 です。
<code>output_log_file_name filename</code>	コマンドログファイルの名前。 デフォルト値は <code>/tmp/dbx.log.uniqueID</code> です。
<code>output_max_string_length number</code>	<code>char *s</code> で出力される文字数を設定します。デフォルト値は 512 です。
<code>output_no_literal</code>	有効にすると、式が文字列 (<code>char *</code>) の場合は、アドレスのみが出力され、文字は出力されません。デフォルト値は off です。
<code>output_pretty_print on off</code>	ウォッチポイントの出力および表示のデフォルトを <code>-p</code> に設定します。デフォルト値は off です。
<code>output_pretty_print_fallback</code>	デフォルトで、問題が発生した場合、 <code>pretty-print</code> は標準出力に戻されます。 <code>pretty-print</code> の問題を診断する場合は、この変数を off に設定して、フォールバックを回避します。デフォルト値は on です。
<code>output_short_file_name on off</code>	ファイル名を表示するときに短形式で表示します。デフォルト値は on です。
<code>overload_function on off</code>	C++ の場合、on に設定すると、自動で多重定義された関数の解決を行います。デフォルト値は on です。
<code>overload_operator on off</code>	C++ の場合、on に設定すると、自動で多重定義された演算子の解決を行います。デフォルト値は on です。
<code>pop_auto_destruct on off</code>	on に設定すると、フレームをポップするときに、ローカルの適切なデストラクタを自動的に呼び出します。デフォルト値は on です。

表 3-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
<code>proc_exclusive_attach on off</code>	<code>on</code> に設定すると、別のツールがすでに接続されている場合、dbx をプロセスへ接続しないようにします。警告: 複数のツールが 1 つのプロセスに接続している状態でプロセスを制御しようとする、混乱が生じるので注意してください。デフォルト値は <code>on</code> です。
<code>rtc_auto_continue on off</code>	<code>rtc_error_log_file_name</code> にエラーを記録して続行します。デフォルト値は <code>off</code> です。
<code>rtc_auto_suppress on off</code>	<code>on</code> に設定すると、特定の位置の RTC エラーが一回だけ報告されます。デフォルト値は <code>off</code> です。
<code>rtc_biu_at_exit on off verbose</code>	メモリ使用検査が明示的に、または <code>check -all</code> によって <code>on</code> になっている場合に使用されます。この値が <code>on</code> だと、簡易メモリー使用状況 (使用中ブロック) レポートがプログラムの終了時に作成されます。値が <code>verbose</code> の場合は、詳細メモリー使用状況レポートがプログラムの終了時に作成されます。 <code>off</code> の場合は出力は生成されません。デフォルト値は <code>on</code> です。
<code>rtc_error_limit number</code>	報告される RTC エラーの数。デフォルト値は 1000 です。
<code>rtc_error_log_file_name filename</code>	<code>rtc_auto_continue</code> が設定されている場合に、RTC エラーが記録されるファイル名。デフォルト値は <code>/tmp/dbx.errlog.uniqueID</code> です。
<code>rtc_error_stack on off</code>	<code>on</code> に設定すると、スタックトレースは、RTC 内部機構へ対応するフレームを示します。デフォルト値は <code>off</code> です。
<code>rtc_inherit on off</code>	<code>on</code> に設定すると、デバッグプログラムから実行される子プロセスでランタイムチェックを有効にし、環境変数 <code>LD_PRELOAD</code> が継承されます。デフォルト値は <code>off</code> です。
<code>rtc_mel_at_exit on off verbose</code>	リーク検査がオンの場合に使用されます。この値が <code>on</code> の場合は、簡易メモリーリークレポートがプログラムの終了時に作成されます。値が <code>verbose</code> の場合は、詳細メモリーリークレポートがプログラムの終了時に作成されます。 <code>off</code> の場合は出力は生成されません。デフォルト値は <code>on</code> です。
<code>run_autostart on off</code>	dbx で実行中でないプログラムで <code>on</code> の場合、 <code>step</code> 、 <code>next</code> 、 <code>stepi</code> 、および <code>nexti</code> を実行した場合、暗黙指定で <code>run</code> を実行し、言語依存のメインルーチンで停止します。 <code>on</code> の場合、 <code>cont</code> は必要に応じて <code>run</code> を暗黙指定します。デフォルト値は <code>off</code> です。

表 3-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
<code>run_io_stdio pty</code>	ユーザープログラムの入出力が、dbx の <code>stdio</code> か、または特定の <code>pty</code> にリダイレクトされるかどうかを指定します。pty は、 <code>run_pty</code> によって指定します。デフォルト値は <code>stdio</code> です。
<code>run_pty ptyname</code>	<code>run_io</code> が <code>pty</code> に設定されているときに使用する <code>pty</code> の名前を設定します。pty は GUI のラッパで使用されます。
<code>run_quick on off</code>	<code>on</code> の場合、シンボリック情報は読み込まれません。シンボリック情報は、 <code>prog -readsysms</code> を使用して要求に応じて読み込むことができます。それまで、dbx は、デバッグ中のプログラムがストリップされているかのように動作します。デフォルト値は <code>off</code> です。
<code>run_savetty on off</code>	dbx とデバッグ対象の間で、tty 設定、プロセスグループ、およびキーボード設定 (<code>-kbd</code> がコマンド行で使用されている場合) を多重化します。エディタやシェルをデバッグする際に便利です。dbx が <code>SIGTTIN</code> または <code>SIGTTOU</code> を取得しシェルに戻る場合は、 <code>on</code> に設定します。速度を多少上げるには <code>off</code> に設定します。dbx がデバッグ対象プログラムに接続されているのか、Sun Studio IDE のもとで動作しているのかということには無関係です。デフォルト値は <code>off</code> です。
<code>run_setpgrp on off</code>	<code>on</code> の場合プログラムが実行時に、フォークの直後に <code>setpgrp(2)</code> が呼び出されます。デフォルト値は <code>off</code> です。
<code>scope_global_enums on off</code>	<code>on</code> の場合、列挙子の有効範囲はファイルスコープではなく大域スコープになります。デバッグ情報を処理する前に設定する必要があります (<code>~/ .dbxrc</code>)。デフォルト値は <code>off</code> です。
<code>scope_look_aside on off</code>	<code>on</code> に設定した場合、ファイルの静的シンボルが、現在のファイルスコープにない場合でもそれを検出します。デフォルト値は <code>on</code> です。
<code>session_log_file_name filename</code>	dbx がすべてのコマンドとその出力を記録するファイルの名前。出力はこのファイルに追加されます。デフォルト値は "" (セッション記録なし) です。
<code>stack_find_source on off</code>	<code>on</code> に設定した場合、デバッグ中のプログラムが <code>-g</code> オプションなしでコンパイルされた指定の関数で停止したとき、dbx はソースを持つ最初のスタックフレームを検索し、自動的にアクティブにします。 デフォルト値は <code>on</code> です。

表 3-1 dbx 環境変数 (続き)

dbx 環境変数	dbx 環境変数の機能
<code>stack_max_size number</code>	<code>where</code> コマンドにデフォルトサイズを設定します。デフォルト値は 100 です。
<code>stack_verbose on off</code>	<code>where</code> コマンドでの引数と行情報の出力を指定します。デフォルト値は <code>on</code> です。
<code>step_abflow stop ignore</code>	<code>stop</code> に設定されていると、シングルステップ実行時に <code>dbx</code> が <code>longjmp()</code> 、 <code>siglongjmp()</code> で停止し、文を送出します。 <code>ignore</code> に設定されていると、 <code>dbx</code> は <code>longjmp()</code> および <code>siglongjmp()</code> の異常制御フロー変更を検出しません。
<code>step_events on off</code>	<code>on</code> に設定すると、ブレークポイントを許可する一方で、 <code>step</code> および <code>next</code> コマンドを使用してコードをステップ実行できます。デフォルト値は <code>off</code> です。
<code>step_granularity statement line</code>	ソース行ステップの細分性を制御します。 <code>statement</code> に設定すると、次のコード <pre>a(); b();</pre> を、実行するための 2 つの <code>next</code> コマンドが必要です。 <code>line</code> に設定すると、1 つの <code>next</code> コマンドでコードを実行します。複数行のマクロを処理する場合、行の細分化は特に有用です。デフォルト値は <code>statement</code> です。
<code>suppress_startup_message number</code>	リリースレベルを設定して、それより下のレベルでは起動メッセージが表示されないようにします。デフォルト値は 3.01 です。
<code>symbol_info_compression on off</code>	<code>on</code> に設定した場合、各 <code>include</code> ファイルのデバッグ情報を 1 回だけ読み取ります。デフォルト値は <code>on</code> です。
<code>trace_speed number</code>	トレース実行の速度を設定します。値は、ステップ間の休止秒数になります。デフォルト値は 0.50 です。

dbx 環境変数および Korn シェル

各 dbx 環境変数は、ksh 変数としてもアクセス可能です。ksh 変数名は dbx 環境変数から取られ、DBX_ という接頭辞が付けられます。たとえば、`dbxenv stack_verbose` および `echo $DBX_stack_verbose` は同じ出力を抑制します。変数の値は直接または `dbxenv` コマンドで割り当てることができます。

コードの表示とコードへの移動

プログラムが停止するたびに dbx が表示するソースコードは、その停止位置に対応するコードです。また、プログラムが停止するたびに、dbx は現在の関数の値をプログラムが停止した関数の値に再設定します。プログラムの停止後、その停止場所以外の関数やファイルを一時的に表示することができます。

この章では、デバッグセッション中に dbx がどのようにコードを参照し、関数やシンボルを検索するかを説明します。また、コマンドを使用して、プログラムの停止位置とは別の場所のコードを一時的に表示したり、識別子、型、クラスの宣言を調べたりする方法も説明します。

この章は、次の各節から構成されています。

- 69 ページの「コードへの移動」
- 72 ページの「プログラム位置のタイプ」
- 72 ページの「プログラムスコープ」
- 74 ページの「スコープ決定演算子を使用してシンボルを特定する」
- 77 ページの「シンボルを検索する」
- 80 ページの「変数、メンバー、型、クラスを調べる」
- 83 ページの「オブジェクトファイルおよび実行可能ファイル内のデバッグ情報」
- 85 ページの「ソースファイルおよびオブジェクトファイルの検索」

コードへの移動

プログラムを実行していないときはいつでも、プログラム内の関数やファイルに移動できます。プログラムに含まれるすべての関数またはファイルを表示できます。現在のスコープはプログラムの停止位置に設定されます (72 ページの「プログラムスコープ」を参照してください)。この機能は、`stopat` ブレークポイントを設定し、停止したときにソース行を決定する際に便利です。

ファイルの内容を表示する

dbx がプログラムの一部として認識していれば、どのようなファイルでもその内容を表示できます (モジュールまたはファイルが `-g` オプションでコンパイルされていない場合でも可能です)。ファイルの内容を表示するためには、次のように入力します。

```
(dbx) file filename
```

`file` コマンドを引数を指定しないで使用すると、現在表示中のファイル名が表示されます。

```
(dbx) file
```

dbx は、行番号を指定しないと、最初の行からファイルを表示します。

```
(dbx) file filename ; list line_number
```

ソースコードの行でブレークポイントを設定する詳細については、[96 ページ](#)の「ソースコードの特定の行に `stop` ブレークポイントを設定する」を参照してください。

関数を表示する

`func` コマンドを使用すると、関数を表示できます。コマンド `func` に続けて、関数名を入力します。たとえば、次のようにします。

```
(dbx) func adjust_speed
```

`func` コマンドを引数なしで使用すると、現在表示中の関数が表示されます。

詳細については、[320 ページ](#)の「`func` コマンド」を参照してください。

あいまいな関数名をリストから選択する (C++)

C++ の場合、あいまいな名前または多重定義されている関数名を指定してメンバー関数を表示しようとする、多重定義されているというメッセージが表示され、指定された名前を持つ関数のリストが示されます。表示したい関数の番号を入力します。関数が属している特定クラスを知っている場合は、クラス名と関数名を入力できます。たとえば、次のようにします。

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

複数存在する場合の選択

同じスコープレベルから複数のシンボルにアクセスできる場合、dbx は、あいまいさについて報告するメッセージを出力します。

```
(dbx) func main
(dbx) which C::foo
識別子 'foo' が複数あります
次の名前から 1 つ選択してください:
 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)
```

which コマンドのコンテキストでシンボル名のリストから特定のシンボルを選んでも、dbx またはプログラムの状態には影響しません。ほとんどの場合、どのシンボルを選んでも名前が表示されるだけです。

ソースリストの出力

list コマンドは、ファイルまたは関数のソースリストを出力するために使用します。filename を指定した場合は filename の先頭を表示、number を指定した場合は現在のファイルの number 行目を表示、function を指定した場合はその関数を表示します。

list コマンドの詳細については、329 ページの「list コマンド」を参照してください。

呼び出しスタックの操作によってコードを表示する

プロセスが存在するときにコードを表示する方法としては、さらに「呼び出しスタックを操作する」方法があります。この方法では、スタック操作コマンドを使用して現在スタック上にある関数を表示します。その結果、現時点でアクティブなすべてのルーチンが表示されます。スタックを操作すると、現在の関数とファイルは、スタック関数を表示するたびに変更されます。停止位置は、スタックの「底」にあるものと考えられます。したがって、そこから離れるには up コマンドを使用します。つまり、main 関数または begin 関数に向かって移動します。現在のフレーム方向へ移動するには、down コマンドを使用します。

呼び出しスタックの移動についての詳細は、112 ページの「スタックを移動してホームに戻る」を参照してください。

プログラム位置のタイプ

dbx は、3つのグローバル位置を使用して検査しているプログラムの部分を追跡します。

- `dis` コマンド (311 ページの「`dis` コマンド」参照) および `examine` コマンド (315 ページの「`examine` コマンド」参照) によって使用され更新される現在のアドレス。
- `list` コマンド (329 ページの「`list` コマンド」参照) によって使用され更新される現在のソースコード行。この行番号は表示スコープを変更するいくつかのコマンドによってリセットされます (73 ページの「表示スコープの変更」を参照)。
- 現在の表示スコープ。複合変数である表示スコープについては、73 ページの「表示スコープ」を参照してください。表示スコープは式の評価中に使用されます。スコープは `line` コマンド、`func` コマンド、`file` コマンド、`list func` コマンド、および `list file` コマンドによって更新されます。

プログラムスコープ

スコープとは、変数または関数の可視性について定義されたプログラムのサブセットです。あるシンボルの名前が特定の実行地点において可視となる場合、そのシンボルは「スコープ範囲内にある」こととなります。C 言語では、関数はグローバルまたはファイル固有のスコープを保持します。変数は、グローバル、ファイル固有、関数、またはブロックのスコープを保持します。

現在のスコープを反映する変数

次の変数は現在のスレッドまたは LWP の現在のプログラムカウンタを常に反映し、表示スコープを変更するコマンドには影響されません。

<code>\$scope</code>	現在のプログラムカウンタのスコープ
<code>\$lineno</code>	現在の行番号
<code>\$func</code>	現在の関数
<code>\$class</code>	<code>\$func</code> が所属するクラス
<code>\$file</code>	現在のソースファイル
<code>\$loadobj</code>	現在のロードオブジェクト

表示スコープ

プログラムのさまざまな要素を `dbx` を使用して検査する場合、表示スコープを変更します。`dbx` は、式の評価中にあいまいなシンボルを解析するなどの目的で表示スコープを使用します。たとえば、次のコマンドを入力すると、`dbx` は表示スコープを使用して印刷する `i` を判断します。

```
(dbx) print i
```

各スレッドまたは LWP は独自の表示スコープを持っています。スレッド間を切り替えるときに、各スレッドはそれぞれの表示スコープを記憶します。

表示スコープのコンポーネント

表示スコープのいくつかのコンポーネントは、次の事前定義済み `ksh` 変数内で可視になります。

```
$vscope      言語スコープ
$vloadobj    現在の表示ロードオブジェクト
$file        現在の表示ソースファイル
$vlineno     現在の表示行番号
$class       C++ クラス
```

現在の表示スコープのすべてのコンポーネントは、相互互換性があります。たとえば、関数を含まないファイルを表示する場合、現在の表示ソースファイルが新しいファイル名に更新され、現在の表示関数が `NULL` に更新されます。

表示スコープの変更

次のコマンドは表示スコープを変更するもっとも一般的な方法です。

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

`debug` コマンドおよび `attach` コマンドは最初の表示スコープを設定します。

ブレークポイントに達すると、`dbx` によって表示スコープが現在の位置に設定されます。`stack_find_source` 環境変数 (62 ページの「[dbx 環境変数の設定](#)」参照) が `ON` に設定されている場合、`dbx` はソースコードを持っているスタックフレームを検索してアクティブにします。

up コマンド (381 ページの「[up コマンド](#)」参照)、down コマンド (314 ページの「[down コマンド](#)」参照)、frame number コマンド (320 ページの「[frame コマンド](#)」参照)、または pop コマンド (343 ページの「[pop コマンド](#)」参照) を使用して現在のスタックフレームを変更すると、新しいスタックフレームからのプログラムカウンタに従って dbx によって表示スコープが設定されます。

list コマンド (329 ページの「[list コマンド](#)」参照) によって使用される行番号位置は、list function または list file を使用した場合にのみ表示スコープを変更します。表示スコープが設定されると、list コマンド用の行番号位置が表示スコープの最初の行番号に設定されます。続けて list コマンドを使用すると、list コマンド用の現在の行番号位置が更新されますが、現在のファイル内で行をリストしているかぎり表示スコープは変更されません。たとえば、次のように入力すると、dbx によって my_func のソースの開始位置がリストされ、表示スコープが my_func に変更されます。

```
(dbx) list my_func
```

次のように入力すると、dbx によって現在のソースファイル内の行 127 がリストされ、表示スコープは変更されません。

```
(dbx) list 127
```

file コマンドまたは func コマンドを使用して現在のファイルまたは現在の関数を変更すると、表示スコープも更新されます。

スコープ決定演算子を使用してシンボルを特定する

func または file を使用する場合、「スコープ決定演算子」を使用して、ターゲットとして指定する関数の名前を特定することができます。

dbx では、シンボルを特定するためのスコープ決定演算子として、逆引用符演算子 (^)、C++ 逆引用符演算子 (:::), およびブロックローカル演算子 (:lineno) を使用することができます。これらの演算子は別々に、あるいは同時に使用します。

停止位置以外の部分のコードを表示するためにファイルや関数の名前を特定するだけでなく、スコープ外の変数や式の出力や表示を行ったり、型やクラスの宣言を表示したり (whatis コマンドを使用) する場合にも、シンボルを特定することが必要です。シンボルの特定規則はすべての場合で同じです。この節で示す規則は、あらゆる種類のシンボル名の特定に適用されます。

逆引用符演算子

逆引用符演算子 (`'`) は、大域スコープの変数あるいは関数を検索するために使用できます。

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

プログラムでは、同じ関数名を2つの異なるファイル(またはコンパイルモジュール)で使用できます。この場合、`dbx` に対して関数名を特定して、表示する関数を認識させる必要があります。ファイル名に関連して関数名を特定するには、汎用逆引用符 (`'`) スコープ決定演算子を使用してください。

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

コロンを重ねたスコープ決定演算子 (C++)

次のような名前を持つ C++ のメンバー関数、トップレベル関数、またはグローバルスコープを伴う変数を特定するときは、コロンを2つ重ねた演算子 (`::`) を使用します。

- 多重定義されている名前(複数の異なる引数型で同じ名前が使用されている)
- あいまいな名前(複数の異なるクラスで同じ名前が使用されている)

多重定義された関数名を特定することができます。多重定義された関数名を特定しないと、`dbx` は多重定義表示リストを自動的に表示して、表示する関数を選択するよう要求します。関数のクラス名がわかっている場合は、それを二重コロンのスコープ決定演算子とともに使用して、名前を特定できます。

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

たとえば、`hand` がクラス名で `draw` が関数名の場合は、次のようになります。

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

ブロックローカル演算子

ブロックローカル演算子 (`:line_number`) を使用すると、ネストされたブロック内にある変数を参照することができます。これを行う必要があるのはパラメータまたはメンバー名を隠蔽している局所変数がある場合、またはそれぞれが個別の局所変数を持っている複数のブロックがある場合です。`line_number` は、対象となる変数に対するブロック内のコードの最初の行番号です。`dbx` が局所変数をブロックローカル演算子で特定した場合、`dbx` は最初のコードブロックの行番号を使用しますが、`dbx` の式ではスコープ内の任意の行番号を使用することができます。

次の例では、ブロックローカル演算子 (:230) が逆引用符演算子と組み合わせられています。

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

次の例は、関数内で複数存在する変数名が、ブロックローカル演算子によって特定され、dbx がその変数の内容を評価している様子を示しています。

```
(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
```

リンカー名

dbx は、(C++ のようにさまざまな名前が混在するため) リンカー名ごとにシンボルを探そう特別な構文を使用します。シンボル名の接頭辞として # 記号を付け、Korn シェルで \$ 記号の前にエスケープ文字 \ を使用します。たとえば、次のようにします。

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

シンボルを検索する

同じ名前が多くのある場所で使用されたり、プログラム内の異なる種類の構成要素を参照したりすることがあります。dbx コマンド `whereis` は、特定の名前を持つすべてのシンボルの完全修飾名(すなわち位置)のリストを表示します。一方、dbx コマンド `which` は、特定の名前を式に指定したときに、実際に使用されるシンボルを示します(388 ページの「`which` コマンド」を参照)。

シンボルの出現を出力する

指定シンボルの出現すべてのリストを出力するには、`whereis symbol` を使用します。ここで、`symbol` は任意のユーザー定義識別子にすることができます。たとえば、次のようにします。

```
(dbx) whereis table
前方: 'Blocks'block_draw.cc'table
関数: 'Blocks'block.cc'table::table(char*, int, int, const point&)
クラス: 'Blocks'block.cc'table
クラス: 'Blocks'main.cc'table
変数:      'libc.so.1'hsearch.c'table
```

この出力には、プログラムが `symbol` を定義する読み込み可能オブジェクトの名前が、各オブジェクトの構成要素の種類(クラス、関数、または変数)とともに示されます。

dbx シンボルテーブルの情報は必要に応じて読み取られるため、`whereis` コマンドは、すでに読み込まれているシンボルの出現についてしか出力しません。デバッグセッションが長くなると、出現のリストは大きくなります(83 ページの「`オブジェクトファイルおよび実行可能ファイル内のデバッグ情報`」参照)。

詳細については、388 ページの「`whereis` コマンド」を参照してください。

実際に使用されるシンボルを調べる

`which` コマンドにより、特定の名前を (完全に修飾しないで) 式に指定したときにどのシンボルが使用されるかを前もって調べることができます。たとえば、次のようにします。

```
(dbx) func
wedge::wedge(char*, int, int, const point&, load_bearing_block*)
(dbx) which draw
'block_draw.cc'wedge::draw(unsigned long)
```

`which` コマンドに指定したシンボル名が局所的スコープにない場合、スコープ決定パスで検索が行われます。決定パスで最初に見つかった名前の完全修飾名が表示されます。

決定パスに含まれる任意の場所で、同じスコープの該当するシンボルが複数見つかった場合、あいまいであることを示すメッセージが表示されます。

```
(dbx) which fid
識別子 'fid' が複数あります
次の名前から 1 つ選択してください:
 0) Cancel
 1) 'example'file1.c'fid
 2) 'example'file2.c'fid
```

`dbx` は、あいまいなシンボル名をリストで示し、多重定義であることを表示します。`which` コマンドのコンテキストでシンボル名のリストから特定のシンボルを選んでも、`dbx` またはプログラムの状態には影響しません。ほとんどの場合、どのシンボルを選んでも名前が表示されるだけです。

`which` コマンドは、あるシンボル (この例の場合は `block`) をコマンド (たとえば、`print` コマンド) のターゲットにした場合に何が起るかを前もって示すものです。あいまいな名前を指定して、多重定義が表示された場合は、該当する複数の名前の中のどれを使用するかがまだ特定されていません。`dbx` は該当する名前を列挙し、ユーザーがそのうちの1つを選択するまで待機します。`dbx` は該当する名前を列挙し、ユーザーがそのうちの1つを選択するまで待機します。`which` コマンドの詳細については、[388 ページ](#)の「`which` コマンド」を参照してください。

スコープ決定検索パス

式を含むデバッグコマンドを発行すると、式内のシンボルが次の順序で調べられます。`dbx` はシンボルをコンパイラが現在の表示スコープにあるとして決定します。

1. 現在の表示スコープを使用する現在の関数のスコープ内 (73 ページの「表示スコープ」参照)。プログラムが、入れ子になったブロックで停止した場合はそのブロック内で検索したあと、その関数によって宣言されている外側のすべてのブロックのスコープ内で検索します。
2. C++ の場合のみ: 現在の関数クラスのクラスメンバーとその基底クラス。
3. C++ の場合のみ: 現在のネームスペース。
4. 現在の関数のパラメータ。
5. すぐ外側にあるモジュールで、一般に、現在の関数が含まれているファイル。
6. この共有ライブラリまたは実行可能ファイル専用で作成されたシンボル。これらのシンボルはリンカースコープを使用して作成できます。
7. メインプログラム用で、その次に共有ライブラリ用のグローバルシンボル。
8. 前述のすべてで該当するシンボルが見つからなかった場合は非公開変数、すなわち別のファイル内で「静的」な変数または関数と見なします。dbxenv による `scope_look_aside` の設定値によっては、コンパイル単位ごとにファイル静的シンボルを検索することもできます。

dbx はこの検索パスで最初に見つけたシンボルを使用します。変数が見つからなかった場合はエラーを報告します。

スコープ検索規則の緩和

静的シンボルおよび C++ メンバー関数のスコープ検索規則を緩和するには、dbx 環境変数 `scope_look_aside` を on に設定します。

```
dbxenv scope_look_aside on
```

または、「二重逆引用符」接頭辞を使用します。

```
stop in ``func4          func4 (静的スコープにない場合)
```

dbx 環境変数 `scope_look_aside` が on に設定されている場合、dbx は次を検索します。

- その他のファイルで定義されている静的変数 (現在のスコープで見つからなかった場合)。/usr/lib に位置するライブラリのファイルは検索されません。
- クラス修飾子のない C++ メンバー関数
- その他のファイルの C++ インラインメンバー関数のインスタンス (メンバー関数が現在のファイルでインスタンス化されていない場合)

which コマンドは、dbx がどのシンボルを検索するかを前もって示すものです。あいまいな名前を指定して、多重定義が表示された場合は、該当する複数の名前のうち

のどれを使用するかがまだ特定されていません。dbxは該当する名前を列挙し、ユーザーがそのうちの1つを選択するまで待機します。

詳細については、[320 ページの「func コマンド」](#)を参照してください。

変数、メンバー、型、クラスを調べる

dbx コマンド `whatis` は、識別子、構造体、型、C++ のクラス、式の型の宣言または定義を出力します。検査できる識別子には、変数、関数、フィールド、配列、列挙定数が含まれます。

詳細については、[383 ページの「whatis コマンド」](#)を参照してください。

変数、メンバー、関数の定義を調べる

識別子の宣言を出力するには、次のように入力します。

```
(dbx) whatis identifier
```

識別名は、必要に応じてファイルおよび関数情報によって修飾します。

C++ プログラムについては `whatis identifier` 関数テンプレート例示をリストします。テンプレート定義は、`whatis -t identifier` を付けて表示されます。[81 ページの「型およびクラスの定義を調べる」](#)を参照してください。

Java プログラムについては、`whatis identifier` は、クラスの宣言、現在のクラスのメソッド、現在のフレームの局所変数、または現在のクラスのフィールドをリストします。

メンバー関数を出力するには、次のように入力します。

```
(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();  
:
```

データメンバーを出力するには、次のように入力します。

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

変数を指定すると、whatis コマンドによってその変数の型が示されます。

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

フィールドを指定すると、whatis コマンドによってそのフィールドの型が示されます。

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

メンバー関数で停止したときは、this ポインタを調べることができます。

```
(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;
```

型およびクラスの定義を調べる

whatis コマンドの `-t` オプションは、型の定義を表示します。C++ については、whatis `-t` で表示されるリストは、テンプレート定義およびクラステンプレート例示を含みます。

型または C++ のクラスの宣言を出力するには次のようにします。

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

whatis コマンドには、継承されたメンバーを表示するための `-r` (再帰) オプションが用意されています。このオプションを指定すると、指定したクラスの宣言とともに、そのクラスが基となるクラスから継承したメンバーが表示されます。

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

whatis `-r` による出力は、クラス階層と各クラスのサイズによって長くなる場合があります。出力の先頭には、階層のもっとも上にあるクラスから継承されたメンバーのリストが示されます。メンバーのリストは、コメント行によって親クラスごとに分けられます。

ここに、2つの例を示します。tableクラスは、load_bearing_blockクラスの子クラスの1つです。また、load_bearing_blockクラスは、blockの子クラスです。

-rを指定しないと、tableクラスで宣言されているメンバーが示されます。

```
(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);
};
```

次に、子クラスが継承するメンバーを表示するために `whatis -r` がその子クラスで使用された場合の結果を示します。

```
(dbx) whatis -t -r class table
class table : public load_bearing_block {
public:
    /* 基底 class table::load_bearing_block::block から */
    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();
// protected: までのいくつかのメンバー省略
    char *nm;
    int movable;
    int width;
    int height;
    class point position;
    class load_bearing_block *supported_by;
    Panel_item panel_item;
    /* 基底 class table::load_bearing_block から */
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;
/* class table から */
public:
table::table(char *name, int w, int h, const class point &pos);
virtual char *table::type();
virtual void table::draw(unsigned long pw);
};
```

オブジェクトファイルおよび実行可能ファイル内のデバッグ情報

ソースファイルを `-g` オプションを使用してコンパイルして、プログラムをよりデバッグしやすくすることができます。`-g` オプションを使用すると、コンパイラがデバッグ情報 (スタブまたは DWARF 形式) をプログラム用のコードおよびデータとともにオブジェクトファイルに記録します。

`dbx` は、必要なときに要求に応じて各オブジェクトファイル (モジュール) のデバッグ情報を解析して読み込みます。`module` コマンドを使用することによって `dbx` に特定のモジュール、またはすべてのモジュールのデバッグ情報を読み込むように要求することができます。[85 ページの「ソースファイルおよびオブジェクトファイルの検索」](#) も参照してください。

オブジェクトファイルの読み込み

オブジェクト (`.o`) ファイルがリンクされると、リンカーは任意で要約情報のみを結果ロードオブジェクトに保存することができます。この要約情報は実行時に `dbx` で使用して、実行可能ファイルからではなくオブジェクトファイル自体から残りのデバッグ情報を読み込むことができます。作成された実行可能ファイルの容量は小さいですが、`dbx` を実行するときにオブジェクトファイルが必要になります。

この要件は、オブジェクトファイルを `-xs` オプションを使用してコンパイルし、オブジェクトファイルのすべてのデバッグ情報をリンク時に実行可能ファイルに入れることによって変更することができます。

アーカイブライブラリ (`.a` ファイル) をオブジェクトファイルとともに作成して、そのアーカイブライブラリをプログラムで使用した場合、`dbx` は必要に応じてアーカイブライブラリからオブジェクトファイルを抽出します。ここではオリジナルのオブジェクトファイルは必要ありません。

ただし、すべてのデバッグ情報を実行可能ファイルに入れると、追加のディスク容量が必要になります。デバッグ情報は実行時にプロセスイメージに読み込まれないため、プログラムが遅くなることはありません。

スタブ (情報をデバッグするためのデフォルトの形式) を使用する場合のデフォルトの動作は、コンパイラが要約情報のみを実行可能ファイルに配置します。

DWARF 形式では、オブジェクトファイルの読み込みをサポートしていません。

注 - DWARF 形式は、同じ情報をスタブ形式で記録するよりも大幅にサイズが小さくなります。ただし、すべての情報が実行可能ファイルにコピーされるため、DWARF 情報はスタブ情報よりもサイズが大きくなって見えてしまいます。

モジュールについてのデバッグ情報

`module` コマンドおよびそのオプションは、デバッグセッション中、プログラムモジュールを追跡するのに役立ちます。`module` コマンドを使用して、1つまたはすべてのモジュールについてのデバッグ情報を読み込みます。通常 `dbx` は、必要に応じて、自動的にゆっくりとモジュールについてのデバッグ情報を読み込みます。

1つのモジュール *name* についてのデバッグ情報を読み込むには、次のように入力します。

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

すべてのモジュールについてのデバッグ情報を読み込むには、次のように入力します。

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

ここで

- a すべてのモジュールを指定します。
- f ファイルが実行可能より新しい場合でも、デバッグ情報を強制的に読み込みます。
- q 静止モードを指定します。
- v 言語、ファイル名などを出力する冗長モードを指定します。これはデフォルト値です。

現在のモジュール名を出力するには、次のように入力します。

```
(dbx) module
```

モジュールのリスト

`modules` コマンドは、モジュール名をリストすることにより、モジュールを追跡することができます。

すでに `dbx` に読み取られたデバッグ情報を含むモジュールの名前をリスト表示するには、次のように入力します。

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

すべてのプログラムモジュール名(デバッグ情報付き、またはなし)をリスト表示するには、次のように入力します。

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

デバッグ情報付きのすべてのプログラムモジュール名をリスト表示するには、次のように入力します。

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

ここで

-v	言語、ファイル名などを出力する冗長モードを指定します。
----	-----------------------------

ソースファイルおよびオブジェクトファイルの検索

`dbx` には、プログラムに関連するソースファイルおよびオブジェクトコードファイルの位置を認識させる必要があります。オブジェクトファイルのデフォルトディレクトリは、プログラムが最後にリンクされたときにオブジェクトファイルがあったディレクトリです。ソースファイルのデフォルトディレクトリは、最後のコンパイル時にそれらが存在したディレクトリです。ソースファイルまたはオブジェクトファイルを移動したか、またはそれらを新しい位置にコピーした場合は、プログラムを再リンクするか、新しい位置に変更してからデバッグを行うか、または `pathmap` コマンドを使用します。

`dbx` では、オブジェクトファイルを使用して追加のデバッグ情報を読み込む場合があります。ソースファイルは、`dbx` がソースコードを表示するときに使用されます。

プログラムをコンパイルしてリンクしたためにソースファイルまたはオブジェクトファイルを移動した場合、その新しい位置を検索パスに追加できます。`pathmap` コマンドは、ファイルシステムの現在のディレクトリと実行可能イメージ内の名前とのマッピングを作成します。このマッピングは、ソースパスとオブジェクトファイルパスに適用されます。

ディレクトリ *from* からディレクトリ *to* への新しいマッピングを確立するには、次のように入力します。

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

-c を使用すると、このマッピングは、現在の作業ディレクトリにも適用されます。

pathmap コマンドは、ホストによってベースパスの異なる、自動マウントされた明示的な NFS マウントファイルシステムを扱う場合でも便利です。-c は、現在の作業ディレクトリが自動マウントされたファイルシステム上で不正確なオートマウンタが原因で起こる問題を解決する場合に使用してください。

/tmp_mnt と / のマッピングはデフォルトで存在します。

詳細については、[342 ページの「pathmap コマンド」](#)を参照してください。

プログラムの実行制御

実行、ステップ、および続行に使用されるコマンド (run、rerun、next、step、および cont) は、プロセス制御コマンドと呼ばれます。262 ページの「cont at コマンド」で説明するイベント管理コマンドとともに使用すると、プログラムが dbx のもとで実行されるときに、その実行時の動作を管理できます。

この章の内容は次のとおりです。

- 87 ページの「dbx でプログラムを実行する」
- 88 ページの「動作中のプロセスに dbx を接続する」
- 90 ページの「プロセスから dbx を切り離す」
- 90 ページの「プログラムのステップ実行」
- 94 ページの「Control+C によってプロセスを停止する」

dbx でプログラムを実行する

プログラムを初めて dbx に読み込むと、dbx はそのプログラムの「メイン」ブロック (C、C++、および Fortran 90 の場合は main、Fortran 77 の場合は MAIN、Java コードの場合は main クラス) に移動します。dbx は続いて、ユーザーから出されるコマンドを待機します。ユーザーは、コード上を移動するか、イベント管理コマンドを使用できます。

プログラムを実行する前に、そのプログラムにブレークポイントを設定することもできます。

注 - Java™ コードと C JNI (Java Native Interface) コードまたは C++ JNI コードの混在するアプリケーションをデバッグする場合に、まだ読み込まれていないコードでブレークポイントを設定することができます。このようなコードでブレークポイントを設定する方法については、[223 ページの「JVM ソフトウェアによって読み込まれていないコードに対するブレークポイントの設定」](#)を参照してください。

プログラムの実行を開始するには、`run` コマンドを使用します。

`dbx` で引数を指定しないでプログラムを実行するには、次のように入力します。

```
(dbx) run
```

任意でコマンド行の引数と入出力の切り替えを追加できます。この場合は、次のように入力します。

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

注 - Java アプリケーションの入力および出力をリダイレクトすることはできません。

`run` コマンドの出力は、`dbx` を実行しているシェルに `noclobber` を設定した場合でも、既存ファイルを上書きします。

`run` コマンドそのものは、前の引数とリダイレクトを使用して、プログラムを実行します。詳細については、[352 ページの「run コマンド」](#)を参照してください。`rerun` コマンドは、元の引数とリダイレクトなしでプログラムを実行します。詳細については、[349 ページの「rerun コマンド」](#)を参照してください。

動作中のプロセスに dbx を接続する

すでに動作中のプログラムをデバッグしなければならないことがあります。動作中のプロセスにデバッグ機能を接続しなければならないのは、次のような場合です。

- 動作中のサーバーをデバッグしたいが、停止させたくない
- 動作中の GUI プログラムをデバッグしたいが、再起動したくない
- プログラムが無限ループに入っているかもしれないので、プログラムを停止させずにデバッグしたい

このような場合は、動作中のプログラムのプロセス ID (`process_id`) を引数として `dbx debug` コマンドに渡せば、そのプログラムに `dbx` を接続することができます。

デバッグを終了すると、`detach` コマンドが使用され、プロセスを終了することなく `dbx` の制御からプログラムを解放することができます。

動作中のプロセスに接続されているときに dbx を終了すると、dbx は終了前に自動的に切り離しを行います。

dbx とは関係なく実行されるプログラムへ dbx を接続するには、attach コマンドまたは debug コマンドを使用します。

すでに実行中のプロセスへ dbx を接続するには、次のように入力します。

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

または

```
(dbx) attach process_id
```

program_name を - (ダッシュ) で置換することができます。dbx は、プロセス ID と関連するプログラムを自動的に検索し、ロードします。

詳細については、307 ページの「debug コマンド」と 288 ページの「attach コマンド」を参照してください。

dbx が実行中でない場合は、次のように入力して dbx を開始します。

```
% dbx program_name process_id
```

プログラムに dbx を接続すると、そのプログラムは実行を停止します。このプログラムは、dbx に読み込んだプログラムの場合と同様にして調べることができます。任意のイベント管理コマンドまたはプロセス制御コマンドを使用してデバッグできます。

既存のプロセスのデバッグ中に dbx を新規のプロセスに接続すると、次のようになります。

- 現在デバッグ中のプロセスを run コマンドを使用して開始すると、新規のプロセスに接続する前にプロセスが終了します。
- 現在のプロセスを attach コマンドを使用するか、またはコマンド行でプロセス ID を指定することによってデバッグを開始すると、新規のプロセスに接続する前に現在のプロセスから切り離されます。

dbx を接続しようとしているプロセスが、SIGSTOP、SIGTSTP、SIGTTIN、SIGTTOU シグナルによって停止した場合、接続が成功し、次のようなメッセージが表示されます。

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

プロセスは検査可能ですが、プロセスを再開するためには、cont コマンドでプロセスに SIGCONT シグナルを送信する必要があります。

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

特定の例外がある接続済みプロセスで実行時チェック機能を使用できます。149 ページの「[接続されたプロセスへの RTC の使用](#)」を参照してください。

プロセスから dbx を切り離す

プログラムのデバッグが終了したら、`detach` コマンドを使用して dbx をプログラムから切り離してください。プログラムは切り離すときに `-stop` オプションを指定しないかぎり、dbx とは独立して実行を再開します。

dbx の制御のもとで、プロセスを実行から切り離すには、次のように入力します。

```
(dbx) detach
```

dbx が占有アクセスしているときにブロックされるほかの `/proc` ベースのデバッグツールを一時的に適用している間に、プロセスを切り離して停止状態にすることができます。たとえば、次のようにします。

```
(dbx) oproc=$proc          # 古いプロセス ID を覚えておく
(dbx) detach -stop
(dbx) /usr/proc/bin/pwdx $oproc
(dbx) attach $oproc
```

詳細については、311 ページの「[detach コマンド](#)」を参照してください。

プログラムのステップ実行

dbx は、`next` と `step` というステップ実行のための基本コマンドに加え、`step` コマンドの 2 つの変形である `step up` と `step to` をサポートします。`next` と `step` はともに、プログラムにソースの 1 行を実行させ、停止します。

実行される行に関数呼び出しが含まれる場合、`next` コマンドにより、呼び出しは実行され、次の行で停止します(呼び出しを「ステップオーバー」)。 `step` コマンドは、呼び出された関数の最初の行で停止します(呼び出しへの「ステップ」)。

`step up` コマンドは、関数をステップ実行したあと、呼び出し元の関数へプログラムを戻します。

`step to` コマンドは、現在のソース行で指定されている関数にステップするか、関数が指定されていない場合は、現在のソース行のアセンブリコードにより最後に呼び出される関数にステップします。条件付の分岐により、関数の呼び出しが発生しないことがあります。また、現在のソース行で関数が呼び出されない場合もあります。このような場合、`step to` は現在のソース行をステップオーバーします。

シングルステップ

指定された数のコード行をシングルステップするには、実行したいコードの行数 $[n]$ を付けた `dbx` コマンド、`next` または `step` を使用します。

```
(dbx) next n
```

または

```
(dbx) step n
```

コマンドの詳細については、339 ページの「`next` コマンド」と 358 ページの「`step` コマンド」を参照してください。

`step_granularity` 環境変数は、`step` コマンドおよび `next` コマンドにより、コードに対する単位を決定します (62 ページの「`dbx` 環境変数の設定」を参照)。単位は文か行のどちらかです。

環境変数 `step_events` (62 ページの「`dbx` 環境変数の設定」を参照) は、ステップ実行中にブレークポイントが使用可能であるかどうかを制御します。

環境変数 `step_abflow` は、`dbx` が異常制御フロー変更が発生しそうになっていることを検出したときに停止するかどうかを制御します (62 ページの「`dbx` 環境変数の設定」を参照)。このような制御フロー変更は、`siglongjmp()` または `longjmp()` の呼び出し、あるいは例外の送出しが原因で発生することがあります。

プログラムを継続する

プログラムを継続するには、`cont` コマンドを使用します。

```
(dbx) cont
```

`cont` コマンドには、派生関数の `cont at line_number` があります。これを使用すると、現在のプログラム位置の行以外の行を指定して、プログラムの実行を再開することができます。これにより、再コンパイルすることなく、問題を起こすことがわかっている 1 行または複数行のコードをスキップできます。

指定の行でプログラムを継続するには、次のように入力します。

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

行番号は、プログラムが停止しているファイルから計算される点に注意してください。指定した行番号は、関数のスコープ内になければなりません。

`cont at line_number` と `assign` とを組み合わせると、ある変数の値を正しく計算できない関数の呼び出しが含まれている行を実行しないようにすることができます。

▼ 特定の行からプログラムの実行を再開する

- 1 `assign` コマンドを使用して変数に正しい値を代入します。
- 2 `cont at line_number` で、その値を正しく計算できない関数の呼び出しが含まれている行を飛ばします。

プログラムが 123 行目で停止しているものとします。その行では、関数 `how_fast()` を呼び出しています。この関数は、変数 `speed` を正しく計算しません。`speed` の正しい値がわかっているため、`speed` に値を代入することができます。そのあと、`how_fast()` の呼び出しを飛ばして、プログラムの実行を 124 行目から継続します。

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

詳細については、303 ページの「`cont` コマンド」を参照してください。

`cont` コマンドを `when` ブレークポイントコマンドとともに使用すると、プログラムは 123 行目の実行を試みるたびに `how_fast()` の呼び出しを飛ばします。

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

`when` コマンドについての詳細は、次の節を参照してください。

- 96 ページの「ソースコードの特定の行に `stop` ブレークポイントを設定する」
- 98 ページの「異なるクラスのメンバー関数にブレークポイントを設定する」
- 99 ページの「同じクラスのメンバー関数にブレークポイントを設定する」
- 99 ページの「非メンバー関数に複数のブレークポイントを設定する」
- 384 ページの「`when` コマンド」

関数を呼び出す

プログラムが停止しているとき、`dbx` コマンド `call` を使用して関数を呼び出すことができます。`call` コマンドには、被呼び出し側関数に渡す必要のあるパラメータの値を指定することもできます。

関数(手続き)を呼び出すには、関数の名前を入力し、その引数を指定します。たとえば、次のようにします。

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

パラメータは省略できますが、関数名 `function_name` のあとには必ず括弧を入力してください。たとえば、次のようにします。

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

`call` コマンドを使用して関数を明示的に呼び出したり、関数呼び出しを含む式を評価するか、`stop in glyph -if animate()` などの条件付修飾子を使用して、関数を暗黙的に呼び出すことができます。

C++ 仮想関数は、`print` コマンドや `call` コマンド (344 ページの「`print` コマンド」または 289 ページの「`call` コマンド」を参照) を使用するその他の関数、または関数呼び出しを実行するその他のコマンドと同様に呼び出すことができます。

関数が定義されているソースファイルが `-g` フラグでコンパイルされたものであるか、プロトタイプ宣言が現在のスコープで可視であれば、`dbx` は引数の数と型をチェックし、不一致があったときはエラーメッセージを出します。それ以外の場合、`dbx` は引数の数をチェックしません。

デフォルトでは、`call` コマンドが実行されるたびに、`dbx` は `fflush(stdout)` を自動的に呼び出し、入出力バッファに格納されているすべての情報を出力します。自動的なフラッシュをオフにするには、`dbx` 環境変数 `output_auto_flush` を `off` に設定してください。

C++ の場合、`dbx` はデフォルト引数と関数の多重定義も処理します。可能であれば、C++ 多重定義関数の自動解析が行われます。関数を特定できない場合は (関数が `-g` でコンパイルされていない場合など)、多重定義名のリストが表示されます。

`call` を使用すると、`dbx` は `next` のように動作し、被呼び出し側から戻ります。しかし、プログラムが被呼び出し側関数でブレークポイントにあたると、`dbx` はそのブレークポイントでプログラムを停止し、メッセージを表示します。ここで `where` コマンドを実行すると、`dbx` コマンドのレベルを起点として呼び出しが行われたことが示されます。

実行を継続すると、呼び出しは正常に戻ります。強制終了、実行、再実行、デバッグを行おうとすると、`dbx` は入れ子になったインタプリタから回復しようとするので、コマンドが異常終了します。異常終了したコマンドは再発行することができます。また、`pop -c` コマンドを使用して、すべてのフレームを最後の呼び出しまでポップ (解放) することもできます。

安全な呼び出し

`call` または呼び出しを含む式を出力することによって、デバッグしているプロセスを呼び出すと、明示されない重大な障害が発生する可能性があります。次に、注意すべきいくつかの状況と、それらを回避する方法を示します。

- 呼び出しが無限ループに入る可能性があります。これは中断できますが、中断しないと、セグメント例外が発生します。多くの場合、`pop -c` コマンドを使用して、呼び出しの場所に戻ることができます。
- マルチスレッドアプリケーションで呼び出しを行う場合、デッドロックを回避するためにすべてのスレッドが再開されるため、呼び出したスレッド以外のスレッドに影響がおよぶ可能性があります。
- ブレークポイント条件で使用した呼び出しによって、イベント管理が混乱する可能性があります (172 ページの「実行の再開」を参照)。

dbxによって呼び出される一部の呼び出しは「安全」に実行されます。通常の「Stopped with call to ...」ではなく、主にセグメント例外などの問題が発生した場合、dbxは次のように動作します。

- すべての `stop` コマンドは、メモリーアクセスエラーの検出によって実行されたコマンドも含め、無視されます。
- `pop -c` コマンドを自動的に発行して、呼び出しの場所に戻ります。
- 実行を継続します。

dbxが安全な呼び出しを実行するのは、次のような呼び出しです。

- `display` コマンドによって出力される式の中で実行する呼び出し。失敗した呼び出しは、次のように表示されます。`ic0->get_data() = <call failed>`
このような失敗を診断するには、`print` コマンドを使用して、式を出力します。
- `print -p` コマンドを使用する場合を除く、`db_pretty_print()` 関数の呼び出し。
- イベント条件式で使用する呼び出し。呼び出しが失敗した条件は `false` と評価されます。
- `pop` コマンドの実行時にデストラクタを呼び出すための呼び出し。
- すべての内部呼び出し。

Control+Cによってプロセスを停止する

dbxで実行中のプロセスは、Control+C(^C)を使用して停止できます。^Cによってプロセスを停止すると、dbxは^Cを無視しますが、子プロセスはそれをSIGINTと見なして停止します。このプロセスは、それがブレークポイントによって停止しているときと同じように検査することができます。

^Cによってプログラムを停止したあとに実行を再開するには、コマンド `cont` を使用します。実行を再開する場合、`cont` に修飾語 `sig signal_name` は必要ありません。`cont` コマンドは、保留シグナルをキャンセルしたあとで子プロセスを再開します。

ブレイクポイントとトレースの設定

dbx を使用すると、イベント発生時に、プロセスの停止、任意のコマンドの発行、または情報を表示することができます。イベントのもっとも簡単な例はブレイクポイントです。その他のイベントの例として、障害、シグナル、システムコール、`dlopen()` の呼び出し、データ変更などがあります。

トレースは、変数の値の変更など、プログラム内のイベントに関する情報を表示します。トレースの動作はブレイクポイントと異なりますが、トレースとブレイクポイントは類似したイベントハンドラを共有します (263 ページの「イベントハンドラ」を参照)。

この章では、ブレイクポイントとトレースを設定、クリア、およびリストする方法について説明します。ブレイクポイントおよびトレースの設定に使用できるイベント仕様の完全な詳細については、266 ページの「イベント指定の設定」を参照してください。

この章は次の節で構成されています。

- 96 ページの「ブレイクポイントを設定する」
- 103 ページの「ブレイクポイントのフィルタの設定」
- 105 ページの「トレースの実行」
- 106 ページの「ソース行で when ブレイクポイントを設定する」
- 107 ページの「共有ライブラリでブレイクポイントを設定する」
- 107 ページの「ブレイクポイントをリストおよびクリアする」
- 108 ページの「ブレイクポイントを有効および無効にする」
- 108 ページの「イベント効率」

ブレークポイントを設定する

dbx では、ブレークポイントを設定するため、3 種類のコマンドを使用することができます。

- **stop** ブレークポイント。stop コマンドによって作成されたブレークポイントに到達すると、プログラムは停止します。停止したプログラムはほかの dbx コマンドを実行するまで再開されません。
- **when** ブレークポイント。プログラムは、when コマンドで作成されたブレークポイントに到達すると処理を停止し、1 つまたは複数のデバッグコマンドの実行後に処理を再開します。プログラムは、実行コマンドに stop が含まれていないかぎり処理を継続します。
- **trace** ブレークポイント。プログラムは、trace コマンドで作成されたブレークポイントに到達すると処理を停止し、イベント固有のトレース情報行を出力したあと、処理を再開します。

stop、when、および trace コマンドはすべて、イベントの指定を引数として取りまします。イベントの指定は、ブレークポイントのベースとなるイベントを説明しています。イベント指定の詳細については、[266 ページの「イベント指定の設定」](#)を参照してください。

マシンレベルのブレークポイントを設定するには、stopi、wheni、および tracei コマンドを使用します ([第 18 章](#) を参照)。

注 - Java™ コードと C JNI (Java Native Interface) コードまたは C++ JNI コードの混在するアプリケーションをデバッグする場合に、まだ読み込まれていないコードでブレークポイントを設定することができます。このようなコードでブレークポイントを設定する方法については、[223 ページの「JVM ソフトウェアによって読み込まれていないコードに対するブレークポイントの設定」](#)を参照してください。

ソースコードの特定の行に stop ブレークポイントを設定する

stop at コマンドを使用して、行番号にブレークポイントを設定します。ここで、*n* はソースコードの行番号、*filename* は任意のプログラムファイル名修飾子です。

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

次に例を示します。

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

指定された行が、ソースコードの実行可能行ではない場合、dbx は次の有効な実行可能行にブレークポイントを設定します。実行可能な行がない場合、dbx はエラーを出します。

停止場所を確認するには、`file` コマンドで現在のファイルを設定し、`list` コマンドで停止場所とする関数を表示させます。次に、`stop at` コマンドを使用してソース行にブレークポイントを設定します。

```
(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
```

`at an location` イベントを指定する詳細については、[267 ページの「at \[filename: \]line_number」](#) を参照してください。

関数に stop ブレークポイントを設定する

`stop in` コマンドを使用して、関数にブレークポイントを設定します。

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

指定関数中で停止するブレークポイントは、プロシージャまたは関数の最初のソース行の冒頭でプログラムの実行を中断します。

dbx は、次の場合を除いては、ユーザーが参照している変数または関数を決定します。

- 名前のみで、オーバーロードした関数を参照する場合
- 先頭に `'` が付く関数または変数を参照する場合

次の宣言を考えてみましょう。

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

メンバーでない関数で停止する場合、次のように入力して、

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

グローバル関数 `foo(int)` にブレークポイントを設定します。

メンバー関数にブレークポイントを設定するには、次のコマンドを使用します。

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

次のように入力すると、

```
stop in foo
```

`dbx` は、ユーザーがグローバル関数 `foo(int)`、グローバル関数 `foo(double)` のどちらを意味しているのかを判断することができず、明確にするため、オーバーロードしたメニューを表示する場合があります。

次のように入力すると、

```
stop in 'bar
```

`dbx` は、ユーザーがグローバル関数 `bar()`、メンバー関数 `bar()` のどちらを意味しているのかを判断することができないため、オーバーロードしたメニューを表示します。

`in function` イベントを指定する詳細については、[266 ページの「`in function`」](#)を参照してください。

C++ プログラムに複数のブレークポイントを設定する

異なるクラスのメンバー関数の呼び出し、特定のクラスのすべてのメンバー関数の呼び出し、または多重定義されたトップレベル関数の呼び出しに関連する問題が発生する可能性があります。このような場合に対処するために、`inmember`、`inclass`、`infunction` または `inobject` のキーワードのうちの1つを `stop`、`when`、または `trace` コマンドとともに使用することにより、1回のコマンドで C++ コードに複数のブレークポイントを挿入できます。

異なるクラスのメンバー関数にブレークポイントを設定する

特定のメンバー関数のオブジェクト固有のもの (同じメンバー関数名でクラスの異なるもの) それぞれにブレークポイントを設定するには、`stop inmember` を使用します。

たとえば、関数 `draw` が複数の異なるクラスに定義されている場合は、それぞれの関数ごとにブレークポイントを設定します。

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

`inmember` または `inmethod` イベントを指定する詳細については、[267 ページの「`inmember function inmethod function`」](#)を参照してください。

同じクラスのメンバー関数にブレークポイントを設定する

特定のクラスのすべてのメンバー関数にブレークポイントを設定するには、`stop inclass` コマンドを使用します。

デフォルトでは、ブレークポイントはクラスで定義されたクラスメンバー関数だけに挿入され、ベースクラスから継承した関数には挿入されません。ベースクラスから継承した関数にもブレークポイントを挿入するには、`-recurse` オプションを指定します。

クラス `shape` で定義されたすべてのメンバー関数にブレークポイントを設定するには、次のように入力します。

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

クラス `shape` で定義されたすべてのメンバー関数およびクラスから継承する関数にブレークポイントを設定するには、次のように入力します。

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

`inclass` イベントを指定する詳細については、[267 ページの「`inclass classname \[-recurse | -norecurse\]`」](#) および [360 ページの「`stop` コマンド」](#) を参照してください。

`stop inclass` およびその他のブレークポイントを選択することにより、大量のブレークポイントが挿入される場合があるため、`dbx` 環境変数 `step_events` を必ず `on` に設定し、`step` および `next` コマンドの実行速度を上げるようにしてください ([108 ページの「イベント効率」](#) 参照)。

非メンバー関数に複数のブレークポイントを設定する

多重定義された名前を持つ非メンバー関数 (同じ名前を持ち、引数の型または数の異なるもの) に複数のブレークポイントを設定するには、`stop infunction` コマンドを使用します。

たとえば、C++ プログラムで `sort()` という名前の関数が 2 種類定義されていて、一方が `int` 型の引数、もう一方が `float` 型の引数をとる場合に、両方の関数にブレークポイントを置くためには、次のように入力します。

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

`infunction` イベントを指定する詳細については、[267 ページの「`infunction function`」](#) を参照してください。

オブジェクトにブレークポイントを設定する

In Object ブレークポイントを設定し、特定のオブジェクトインスタンスに適用する操作をチェックします。

デフォルトでは、In Object ブレークポイントは、オブジェクトからの呼び出し時に、オブジェクトのクラス(継承されたクラスも含む)のすべての非静的メンバー関数でプログラムを中断します。継承クラスを除くオブジェクトのクラスで定義された非静的メンバー関数だけでプログラムの実行を中断するには、`-norecurse` オプションを指定します。

オブジェクト `foo` のベースクラスで定義されたすべての非静的メンバー関数と、オブジェクト `foo` の継承クラスで定義されたすべての非静的メンバー関数にブレークポイントを設定するには、次のように入力します。

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

オブジェクト `foo` の継承クラスを除く、オブジェクト `foo` のクラスで定義されたすべての非静的メンバー関数だけにブレークポイントを設定するには、次のように入力します。

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

`inobject` イベントの指定方法の詳細については、[267 ページの「`inobject object-expression \[-recurse | -norecurse\]`」](#) および [360 ページの「`stop` コマンド」](#) を参照してください。

データ変更ブレークポイントを設定する

`dbx` でデータ変更ブレークポイントを使用すると、変数値や式がいつ変更されたかをメモしておくことができます。

特定アドレスへのアクセス時にプログラムを停止する

特定のメモリーアドレスがアクセスされたときにプログラムを停止するには、次のように入力します。

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

`mode` はメモリーのアクセス方法を指定します。次の文字(複数可)で構成されます。

- `r` 指定したアドレスのメモリーが読み取られたことを示します。
- `w` メモリーへの書き込みが実行されたことを示します。
- `x` メモリーが実行されたことを示します。

さらに `mode` には、次のいずれかの文字も指定することができます。

- `a` アクセス後にプロセスを停止します(デフォルト)。
- `b` アクセス前にプロセスを停止します。

いずれの場合も、プログラムカウンタは副作用アクションの前後で違反している命令をポイントします。「前」と「後」は副作用を指しています。

address-expression は、その評価によりアドレスを生成できる任意の式です。シンボル式を使用すると、監視される領域のサイズが自動的に推定されます。このサイズは、*byte-size-expression* を指定することにより、上書されます。シンボルを使用しない、型を持たないアドレス式を使用することもできますが、その場合はサイズを指定する必要があります。

次の例では、メモリーアドレス `0x4762` が読み取られたあとにプログラムが停止します。

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

次の例では、変数 `speed` に書き込みが行われる前にプログラムが停止します。

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

`stop access` コマンドを使用する場合、次の点に注意してください。

- 変数に同じ値が書き込まれてもイベントが発生します。
- デフォルトにより、変数に書き込まれた命令の実行後にイベントが発生します。命令が実行される前にイベントを発生させるには、モードを `b` に指定します。

`access` イベントを指定する詳細については、[268 ページの「access mode address-expression \[, byte-size-expression \]」](#) および [360 ページの「stop コマンド」](#) を参照してください。

変数の変更時にプログラムを停止する

指定した変数の値が変更された場合にプログラム実行を停止するには、次のように入力します。

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

`stop change` コマンドを使用する場合、次の点に注意してください。

- `dbx` は、指定の変数の値に変更が発生した行の次の行でプログラムを停止します。
- *variable* が関数に対しローカルである場合、関数が初めて呼び出されて *variable* の記憶領域が割り当てられた時点で、変数に変更が生じたものとみなされます。パラメータについても同じことが言えます。
- このコマンドは、マルチスレッドのアプリケーションに対し機能しません。

change イベントを指定する詳細については、268 ページの「[change variable](#)」および 360 ページの「[stop コマンド](#)」を参照してください。

dbx は、自動シングルステップを実行し、各ステップで値をチェックすることにより、stop change を実装します。ライブラリが -g オプションでコンパイルされていない場合、ステップ実行においてライブラリの呼び出しが省略されます。そのため、制御が次のように流れていく場合、dbx はネストされた user_routine2 をトレースしません。トレースにおいて、ライブラリの呼び出しとネストされた user_routine2 の呼び出しが省略されるからです。

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

variable の値の変更は、user_routine2 が実行されている最中ではなく、ライブラリが呼び出しから戻ったあとに発生したように見えます。

dbx は、ブロック局所変数 ({} でネストされている変数) の変更に対しブレークポイントを設定できません。「ネスト」されたブロック局所変数でブレークポイントまたはトレースを設定しようとすると、その操作を実行できない旨を伝えるエラーメッセージが表示されます。

注 - change イベントよりも access イベントを使用した方が、迅速にデータ変更をチェックできます。access イベントは、自動的にプログラムをシングルステップする代わりに、はるかに迅速なページ保護スキーマを使用するからです。

条件付きでプログラムを停止する

条件文が真と評価された場合にプログラムを停止するには、次のように入力します。

(dbx) **stop cond condition**

condition が発生すると、プログラムは処理を停止します。

stop cond コマンドを使用する場合、次の点に注意してください。

- dbx は、条件が真と評価された行の次の行でプログラムを停止します。
- このコマンドは、マルチスレッドのアプリケーションに対し機能しません。

condition イベントを指定する詳細については、269 ページの「[cond condition-expression](#)」および 360 ページの「[stop コマンド](#)」を参照してください。

ブレイクポイントのフィルタの設定

dbx では、ほとんどのイベント管理コマンドが *event filter* 修飾子をオプションでサポートします。もっとも単純なフィルタは、プログラムがブレイクポイントかトレースハンドラに到達したあと、またはウォッチ条件の発生したあとに、dbx に対してある特定の条件をテストするように指示します。

このフィルタの条件が真 (非 0) と評価された場合、イベントコマンドが適用され、プログラムはブレイクポイントで停止します。条件が偽 (0) と評価された場合、dbx は、イベントが発生しなかったかのようにプログラムの実行を継続します。

フィルタを含む行または関数にブレイクポイントを設定するには、オプションの `-if condition` 修飾文を `stop` コマンドまたは `trace` コマンドの末尾に追加します。

`condition` には、任意の有効な式を指定できます。コマンドの入力時に有効だった言語で書かれた、ブール値または整数値を返す関数呼び出しも有効な式に含まれます。

`in` や `at` など位置に基づくブレイクポイントでは、スコープはブレイクポイント位置のスコープになります。それ以外の場合、イベントではなくエントリ発生時のスコープになります。スコープを正確に指定するために逆引用符演算子 (75 ページの「逆引用符演算子」を参照) を使用する必要があることがあります。

たとえば、次の 2 つのフィルタは異なります。

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

前者は `foo` にブレイクポイントが設定され、条件を検査します。後者は自動的に条件を検査します。

関数の戻り値をフィルタとして使用

関数呼び出しをブレイクポイントフィルタとして使用できます。次の例では、文字列 `str` の値が `abcde` の場合、プログラムが関数 `foo()` で停止します。

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

変数スコープをフィルタとして使用

ブレイクポイントフィルタの設定に変数スコープを使用できます。この例で、現在のスコープは関数 `foo()` にあり、`local` は `main()` で定義された局所変数です。

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

条件付イベントでのフィルタの使用

最初のうちは、条件付イベントコマンド (watch タイプのコマンド) の設定と、フィルタの使用とを混同してしまうかもしれません。概念的には、watch タイプのコマンドは、各行の実行前に検査される「前提条件」を作成します (watch のスコープ内で)。ただし、条件付トリガーのあるブレイクポイントコマンドでも、それに接続するフィルタを持つことができます。

次に具体的な例を示します。

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

このコマンドは、変数 *speed* を監視するように dbx に指令します。speed に書き込みが行われると (watch 部分)、-if フィルタが有効になります。dbx は *speed* の新しい値が *fast_enough* と等しいかどうかチェックします。等しくない場合、プログラムは実行を継続し、stop を「無視」します。

dbx 構文では、フィルタはブレイクの「事後」、構文の最後で [-if condition] 文の形式で指定されます。

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

マルチスレッドプログラムでブレイクポイントに関数呼び出しを含むフィルタを設定すると、dbx がブレイクポイントに達するとすべてのスレッドの実行が停止し、条件が評価されます。条件が合致して関数が呼び出されると、dbx がその呼び出し中すべてのスレッドを再開します。

たとえば、次のブレイクポイントを、多くのスレッドが lookup() を呼び出すマルチスレッドアプリケーションで設定する場合があります。

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

dbx は、スレッド t@1 が lookup() を呼び出して条件を評価すると停止し、strcmp() を呼び出してすべてのスレッドを再開します。dbx が関数呼び出し中に別のスレッドでブレイクポイントに達すると、次のいずれかの警告が表示されます。

イベント無限ループにより次のハンドラ中でイベントの取りこぼしが発生します。

...

イベントの再入

最初のイベント BPT(VID 6m TID 6, PC echo+0x8)

2 番目のイベント BPT*VID 10, TID 10, PC echo+0x8)

以下のハンドラはイベントを処理しません:

...

そのような場合、条件式内で呼び出された関数が `mutex` を取得しないことを確認できる場合は、`-resumeone` イベント指定修飾子を使用して、`dbx` がブレークポイントに達した最初のスレッドのみを再開させることができます。たとえば、次のブレークポイントを設定する場合があります。

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

`-resumeone` 修飾子はすべての場合において問題を防ぐことはしません。たとえば、次の場合にも何も行いません。

- 条件が再帰的に `lookup()` を呼び出すことにより、最初のスレッドと同じスレッドで `lookup()` で 2 回目のブレークポイントに達した場合。
- 条件実行が別のスレッドへの制御を放棄するスレッド。

イベント修飾子の詳細については、[276 ページ](#)の「イベント指定のための修飾子」を参照してください。

トレースの実行

トレースは、プログラムの処理状況に関する情報を収集して表示します。プログラムが、`trace` コマンドで作成されたブレークポイントに到達すると、プログラムの処理が停止され、イベント固有のトレース情報行が出力されたあと、処理が再開されます。

トレースは、ソースコードの各行を実行直前に表示します。極めて単純なプログラムを除くすべてのプログラムで、このトレースは大量の出力を生成します。

さらに便利なトレースは、フィルタを利用してプログラムのイベント情報を表示します。たとえば、関数の各呼び出し、特定の名前前のすべてのメンバー関数、クラス内のすべての関数、または関数の各 `exit` をトレースできます。また、変数の変更もトレースできます。

トレースを設定する

コマンド行に `trace` コマンドを入力することにより、トレースを設定します。`trace` コマンドの基本構文は次のとおりです。

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

`trace` コマンドの完全な構文については、[372 ページ](#)の「`trace` コマンド」を参照してください。

トレースで提供される情報は、トレースに関連する *event* の型に依存します (266 ページの「イベント指定の設定」を参照)。

トレース速度を制御する

トレースの出力が速すぎる場合がよくあります。dbx 環境変数 `trace_speed` を使用すると、各トレースの出力後の遅延を制御できます。デフォルトの遅延は 0.5 秒です。

トレース時の各行の実行間隔を秒単位で設定するには、次のように入力します。

```
dbxenv trace_speed number
```

ファイルにトレース出力を転送する

`-file filename` オプションを使用すると、トレース出力をファイルに転送できます。たとえば、次のコマンドはトレース出力をファイル `trace1` に転送します。

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

トレース出力を標準出力に戻すには、*filename* の代わりに `-` を使用します。トレース出力は常に *filename* に追加されます。トレース出力は、dbx がプロンプト表示するたび、またアプリケーションが終了するたびにフラッシュされます。dbx 接続後にプログラムの実行を再開するか新たに実行を開始すると、*filename* が常に開きます。

ソース行で when ブレークポイントを設定する

when ブレークポイントコマンドは `list` などその他の dbx コマンドを受け付けるため、ユーザーは独自のトレースを作成できます。

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

when コマンドは暗黙の `cont` コマンドとともに機能します。前述の例では、現在の行のソースコードをリストしたあと、プログラムが実行を継続します。list コマンドのあとに `stop` コマンドが含まれていた場合、プログラムの実行は継続されません。

when コマンドの完全な構文については、384 ページの「when コマンド」を参照してください。イベント修飾子の詳細については、276 ページの「イベント指定のための修飾子」を参照してください。

共有ライブラリでブレークポイントを設定する

dbx は、実行時リンカーにおけるプログラムを呼び出すインタフェースを使用しているコード (`dlopen()`、`dlclose()`、および関連関数を呼び出すコード) について完全なデバッグサポートを提供します。実行時リンカーは、プログラム実行の最中、共有ライブラリを結合および結合解除します。`dlopen()`/`dlclose()` デバッグサポートが提供されているため、ユーザーは関数内部を操作したり、プログラムの起動時にリンクされたライブラリの場合と同様に、動的な共有ライブラリの関数にブレークポイントを設定することができます。

ただし、例外もあります。dbx は、`dlopen()` などを読み込まれていないロードオブジェクトにブレークポイントを配置できません。

- `dlopen()` で読み込まれる前のライブラリにブレークポイントを設定できません。
- ライブラリの最初の関数が呼び出されるまで、`dlopen()` で読み込まれるフィルタライブラリにブレークポイントを設定できません。

`loadobject` コマンドを使用すると、事前読み込みリストにそれらのロードオブジェクトの名前を配置できます (331 ページの「`loadobject` コマンド」を参照)。

dbx は、`dlopen()` で読み込まれたロードオブジェクトを確実に管理します。たとえば、新たに読み込まれたロードオブジェクトに設定されたブレークポイントは、次の `run` コマンドが実行されるまで維持されます。これは、ロードオブジェクトが `dlclose()` でロード解除されたあと、再度 `dlopen()` で読み込まれた場合も同様です。

ブレークポイントをリストおよびクリアする

dbx セッション中にブレークポイントやトレースポイントを複数設定することがよくあります。dbx には、それらのポイントを表示したりクリアしたりするためのコマンドが用意されています。

ブレークポイントとトレースポイントの表示

すべての有効なブレークポイントのリストを表示するには、`status` コマンドを使用します。ブレークポイントは ID 番号付きで表示され、この番号はほかのコマンドで使用できます。

C++ の多重ブレークポイントのところでも説明したように、dbx はキーワード `inmember`、`inclass`、`infunction` で設定された多重ブレークポイントを、1 つのステータス ID 番号を使用してまとめて報告します。

ハンドラ ID を使用して特定のブレークポイントを削除

`status` コマンドを使用してブレークポイントをリスト表示した場合、`dbx` は、各ブレークポイントの作成時に割り当てられた ID 番号を表示します。`delete` コマンドを使用することで、ID 番号によってブレークポイントを削除したり、キーワード `all` により、プログラム内のあらゆる場所に現在設定されているブレークポイントをすべて削除することができます。

ブレークポイントを ID 番号 `ID_number` (この場合 3 と 5) によって削除するには、次のように入力します。

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

`dbx` に現在読み込まれているプログラムに設定されているすべてのブレークポイントを削除するには、次のように入力します。

```
(dbx) delete all
```

詳細については、[310 ページの「delete コマンド」](#)を参照してください。

ブレークポイントを有効および無効にする

ブレークポイントの設定に使用するイベント管理コマンド (`stop`、`trace`、`when`) は、イベントハンドラを作成します ([263 ページの「イベントハンドラ」](#)を参照)。これらの各コマンドは、ハンドラ ID (`hid`) として認識される番号を返します。ハンドラ ID は、ブレークポイントを有効または無効にする `handler` コマンド ([323 ページの「handler コマンド」](#)を参照) の引数として利用できます。

イベント効率

デバッグ中のプログラムの実行時間に関するオーバーヘッドの量はイベントの種類によって異なります。もっとも単純なブレークポイントのように、実際はオーバーヘッドが何もないイベントもあります。1つのブレークポイントしかないイベントも、オーバーヘッドは最小です。

実際のブレークポイントがときには何百にもなることのある多重ブレークポイント (`inclass` など) は、コマンド発行時のみオーバーヘッドがあります。これは、`dbx` が永続的ブレークポイントを使用するためです。永続的ブレークポイントは、プロセスに常に保持され、停止するたびに取り除かれたり、`cont` コマンドのたびに置かれたりすることはありません。

注 - `step` コマンドおよび `next` コマンドの場合、デフォルトでは、プロセスが再開される前にすべてのブレークポイントが取り除かれ、ステップが完了するとそれらは再び挿入されます。したがって、多くのブレークポイントを使用したり、多くのクラスで多重ブレークポイントを使用したりしているとき、`step` コマンドおよび `next` コマンドの速度は大幅に低下します。dbxenv 変数 `step_events` を使用して、`step` や `next` のたびにブレークポイントを取り除いたり、挿入し直したりするかどうかを制御することができます。

自動ステップ実行を利用するイベントはもっとも低速です。これは、各ソース行をステップ実行する単純な `trace step` コマンドの場合と同様にはっきりしています。一方、`stop change expression` や `trace cond variable` のようなイベントは、自動的にステップ実行するだけでなく、各ステップで式や変数を評価する必要があります。

これらのイベントは非常に低速ですが、イベントと修飾語 `-in` を使用した関数とを結び付けることで、効率が上がることがよくあります。たとえば、次のようにします。

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

`trace -in main` を使用しないでください。これは `main` によって呼び出された関数の中でも、トレースが有効になるためです。関数 `lookup()` が変数の値を頻繁に変更すると思われる場合には、この方法を使用してください。

呼び出しスタックの使用

この章では、dbxによる呼び出しスタックの使用方法と、呼び出しスタックを処理するときの `where`、`hide`、`unhide`、および `pop` コマンドの使用方法について説明します。

呼び出しスタックは、呼び出されたあと呼び出し側にまだ戻っていない、現在活動状態にあるルーチンすべてを示します。スタックフレームは、単一関数に割り当てられる呼び出しスタックのセクションです。

呼び出しスタックがメモリー上位(上位アドレス)からメモリー下位に成長することから、*up*は呼び出し側(最終的には`main()`)のフレームに向かうこと、そして*down*は呼び出された関数(最終的には現在の関数)のフレームに向かうことを意味します。プログラムの現在位置(ブレークポイント、ステップ実行のあと、プログラムが異常終了してコアファイルが作成された、いずれかの時点で実行されていたルーチン)はメモリー上位に存在しますが、`main()`のような呼び出し側ルーチンはメモリー下位に位置します。

この章の内容は次のとおりです。

- 112 ページの「スタック上での現在位置の検索」
- 112 ページの「スタックを移動してホームに戻る」
- 112 ページの「スタックを上下に移動する」
- 113 ページの「呼び出しスタックのポップ」
- 114 ページの「スタックフレームを隠す」
- 114 ページの「スタックトレースを表示して確認する」

スタック上での現在位置の検索

where コマンドを使用すると、スタックでの現在位置を検索できます。

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

Java™ コードおよび C JNI (Java Native Interface) コードまたは C++ JNI コードが混在するアプリケーションをデバッグする場合、where コマンドの構文は次のとおりです。

```
where [-f] [-q] [-v] [ thread_id ] number_id
```

where コマンドは、クラッシュしてコアファイルを作成したプログラムの状態を知る場合にも役立ちます。プログラムがクラッシュしてコアファイルを作成した場合、そのコアファイルを dbx に読み込むことができます (44 ページの「既存のコアファイルのデバッグ」を参照)。

where コマンドの完全な構文については、386 ページの「where コマンド」を参照してください。

スタックを移動してホームに戻る

スタックを上下に移動することを「スタックの移動」といいます。スタックを上下に移動して関数を表示すると、dbx はスタックの状態を表示し、矢印でその関数を示します。プログラムが停止している位置を「ホーム」と呼び、このホームを起点にし、up コマンド、down コマンド、または frame コマンドを使用してスタックを上下に移動できます。

dbx コマンドの up および down は、ともに引数として、スタック内で現在のフレームから上下に移動するフレームの数を指定する値 (*number*) を受け付けます。*number* が指定されなかった場合のデフォルトは 1 です。-h オプションを付けると、隠されたフレームもすべてカウントされます。

スタックを上下に移動する

現在の関数以外の関数にある局所変数を調べることができます。

スタックの上方向への移動

呼び出しスタックを *number* で指定されたレベル分、上に (main に向かって) 移動するには、次のように入力します。

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

number を指定しない場合、デフォルトは1レベルになります。詳細については、[381 ページの「up コマンド」](#)を参照してください。

スタックの下方方向への移動

呼び出しスタックを *number* で指定されたレベル分、下に (現在の停止点に向かって) 移動するには、次のように入力します。

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

number を指定しない場合、デフォルトは1レベルになります。詳細については、[314 ページの「down コマンド」](#)を参照してください。

特定フレームへの移動

`frame` コマンドは、`up` コマンドや `down` コマンドと同じような働きをします。`where` コマンドで得た番号を指定すると、その番号によって特定されるフレームに直接移動できます。

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

引数なしの `frame` コマンドは、現在のフレーム番号を出力します。*number* を指定すると、その番号によって示されるフレームに直接移動できます。"+" または "-" だけを指定すると、現在のフレームから1レベルだけ上 (+) または下 (-) に移動できます。また、正負の符号と *number* をともに指定すると、指定した数のレベルだけ上または下に移動できます。-h オプションを付けると、隠されたフレームもカウントされます。

`pop` コマンドを使用して特定のフレームに移動できます ([113 ページの「呼び出しスタックのポップ」](#)参照)。

呼び出しスタックのポップ

呼び出しスタックから、停止した関数を削除し、呼び出し中の関数を新たに指定関数で停止する関数にすることができます。

呼び出しスタックの上下方向への移動とは異なり、スタックのポップは、プログラムの実行を変更します。スタックから停止した関数が削除されると、プログラムは以前の状態に戻ります。ただし、大域または静的変数、外部ファイル、共有メンバー、および同様のグローバル状態への変更は対象外です。

`pop` コマンドは、1 個または複数のフレームを呼び出しスタックから削除します。たとえば、スタックから 5 つのフレームをポップするには、次のように入力します。

```
pop 5
```

指定のフレームへポップすることもできます。フレーム 5 へポップするには、次のように入力します。

```
pop -f 5
```

詳細については、[343 ページの「pop コマンド」](#)を参照してください。

スタックフレームを隠す

`hide` コマンドを使用して、現在有効なスタックフレームフィルタをリスト表示します。

正則表現に一致するすべてのスタックフレームを隠すか、または削除するには、次のように入力します。

```
hide [ regular_expression ]
```

regular_expression は、関数名、またはロードオブジェクト名のいずれかを表し、ファイルの照合に `sh` または `ksh` の構文を使用します。

すべてのスタックフレームフィルタを削除するには、`unhide` を使用します。

```
unhide 0
```

`hide` コマンドは、番号とともにフィルタをリスト表示するため、このフィルタ番号を使用して `unhide` コマンドを使用することもできます。

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

スタックトレースを表示して確認する

プログラムフローのどこで実行が停止し、この地点までどのように実行が到達したのかが、スタックトレースに示されます。スタックトレースは、プログラムの状態を、もっとも簡潔に記述したものです。

スタックトレースを表示するには、`where` コマンドを使用します。

`-g` オプションでコンパイルされた関数の場合、引数の名前と種類が既知であるため、正確な値が表示されます。デバッグ情報を持たない関数の場合、16 進数が引数

として表示されます。これらの数字に意味があるとはかぎりません。関数ポインタ 0 を介して関数が呼び出される場合、記号名の代わりに関数の値が下位 16 進数として示されます。

-g オプションを使ってコンパイルされなかった関数の中でも停止することができません。このような関数でトレースを停止すると、dbx はスタックを検索し、関数が -g オプションでコンパイルされている最初のフレームを探し、現在の適用範囲 (72 ページの「プログラマースコープ」を参照) そのフレームに設定します。これは、矢印記号 (=>) によって示されます。

次の例で、main() は -g オプションでコンパイルされているため、記号名と引数の値が表示されます。main() によって呼び出されたライブラリ関数は、-g でコンパイルされていないため、関数の記号名は表示されますが、引数については \$i0 から \$i5 までの SPARC 入力レジスタの 16 進数の内容が示されます。

(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), "main.cc" の 48 行目
:
```

次の例で、プログラムはセグメント例外によりクラッシュしています。今回も main() だけが -g でコンパイルされているため、ライブラリ関数の引数が記号名ではなく 16 進数で表示されています。クラッシュの原因は、SPARC 入力レジスタ \$i0 および \$i1 において strlen() にヌルの引数が指定されたことにあると考えられます。

(dbx) run

```
Running: Cdlib
(プロセス id 6723)
```

CD Library Statistics:

```
Titles:          1

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

シグナル SEGV (フォルトのアドレスにマッピングしていません) 関数 strlen
0xff2b6c5c で

```
0xff2b6c5c: strlen+0x0080:    ld        [%o1], %o2
```

現関数: main

(dbx) where

```
[1] strlen(0x0, 0x0, 0x11795, 0x7efefeff, 0x81010100, 0xff339323), アドレス 0xff2b6c5c
[2] _doprnt(0x11799, 0x0, 0x0, 0x0, 0x0, 0xff00), アドレス 0xff2fec18
```

```
[3] printf(0x11784, 0xff336264, 0xff336274, 0xff339b94, 0xff331f98, 0xff00), アドレス 0xff300780  
=>[4] main(argc = 1, argv = 0xffbef894), "C:\lib.c" の 133 行目  
(dbx)
```

スタックトレースの例については、[38 ページの「呼び出しスタックを確認する」](#) および [207 ページの「呼び出しのトレース」](#) を参照してください。

データの評価と表示

dbx では、次の 2 通りの方法でデータをチェックすることができます。

- データの評価 (print) - 任意の式の値を検査します。
- データの表示 (display) - プログラムが停止するたびに式の値を検査し監視することができます。

この章の内容は次のとおりです。

- 117 ページの「変数と式の評価」
- 123 ページの「変数に値を代入する」
- 123 ページの「配列を評価する」

変数と式の評価

この節は、dbx を使用して変数および式を評価する方法について説明します。

実際に使用される変数を確認する

dbx がどの変数を実行するか確かでないときは、which コマンドを使用して dbx が使用する完全修飾名を調べてください。

変数名が定義されているほかの関数やファイルを調べるには、whereis コマンドを使用します。

コマンドについては、388 ページの「which コマンド」と 388 ページの「whereis コマンド」を参照してください。

現在の関数のスコープ外にある変数

現在の関数のスコープ外にある変数（監視）したい場合は、次のようにします。

- 関数の名前を特定します。74 ページの「スコープ決定演算子を使用してシンボルを特定する」を参照してください。
あるいは
- 現在の関数を変更することにより、関数を表示します。69 ページの「コードへの移動」を参照してください。

変数、式または識別子の値を出力する

式はすべて、現在の言語構文に従う必要がありますが、`dbx` がスコープおよび配列を処理するために導入したメタ構文は除きます。

ネイティブコードの変数または式を評価するには、次のように入力します。

print *expression*

Java コードの式、局所変数、またはパラメータを評価するには、`print` コマンドを使用できます。

詳細については、344 ページの「`print` コマンド」を参照してください。

注 - `dbx` は、C++ の `dynamic_cast` および `typeid` 演算子をサポートしています。これらの 2 つの演算子で式を評価すると、`dbx` は、コンパイラで提供された特定の `rtti` 関数へ呼び出しを行います。ソースが明示的に演算子を使用しない場合、これらの関数はコンパイラで生成されない場合があります、`dbx` は式を評価することができません。

pretty-print の使用

`pretty-print` を使用すると、プログラムで関数呼び出しにより、式の値を独自に整形出力できます。`print`、`rprint`、`display`、または `watch` コマンドに `-p` オプションを指定すると、`dbx` は `const char *db_pretty_print (const T *, int flags, const char *fmt)` 形式の関数を検索して呼び出し、出力または表示用の戻り値を変換します。

この関数の `flags` 引数で渡される値は、次のいずれかのビット単位の論理和です。

`FVERBOSE` `0x1` 現在実装されておらず、常に設定される

FDYNAMIC	0x2	-d
FRECURSE	0x4	-r
FFORMAT	0x8	-f (設定されている場合、fmt はフォーマット部分)
FLITERAL	0x10	-l

`db_pretty_print()` 関数は、静的メンバー関数かスタンドアロン関数に指定できません。

`dbx` の環境変数 `output_pretty_print` を `on` に設定した場合、`print`、`rprint`、または `display` コマンドにデフォルトとして `-p` が渡されます。この動作を上書きするには、`+p` を使用します。

次のことも考慮してください。

- `version 7.6` 以前の `pretty-print` は、`prettyprint` の `ksh` 実装に基づいていました。この `ksh` 関数 (およびその定義済みのエイリアス `pp`) はまだ存在しますが、その動作の大半は `dbx` 内に再実装され、次のような結果になりました。
 - IDE の場合、ウォッチポイント、局所変数、およびバルーン評価で `pretty-print` を利用できる。
 - `print`、`display`、および `watch` コマンドの `-p` オプションでネイティブルートを使用。
 - 特に、ウォッチポイントおよび局所変数に `pretty-print` を頻繁に呼び出すことができるようになったため、スケーラビリティが向上。
 - 式からアドレスを取得できる機会が増加。
 - エラー回復処理の向上。
- `const/volatile` が非修飾型の場合、一般に `db_pretty_print(int *, ...())` と `db_pretty_print(const int *, ...())()` のような関数は区別されます。`dbx` の多重定義解決機能では、識別は行いますが、強制はしません。
 - 識別可能。定義した変数が `int` と `const int` の両方で宣言されている場合、それぞれが適切な関数にルーティングされます。
 - 強制不可。`int` または `const int` 変数が 1 つだけ定義されている場合、それらは両方の関数に一致します。この動作は `pretty-print` に固有ではなく、すべての呼び出しに適用します。
- `pretty-print` 関数は次のような場合に起動されます。
 - `print -p` または `dbx` 環境変数 `output_pretty_print` が `on` に設定されている場合。
 - `display -p` または `dbx` 環境変数 `output_pretty_print` が `on` に設定されている場合。
 - `watch -p` または `dbx` 環境変数 `output_pretty_print` が `on` に設定されている場合。

- dbx 環境変数 `output_pretty_print` が `on` に設定されている場合のバールン評価。
- dbx 環境変数 `output_pretty_print` が `on` に設定されている場合の局所変数。
- `pretty-print` 関数は次の場合に呼び出されません。
 - `$[]`。 `$[]` はスクリプトで使用することを意図しており、予測できる必要があるためです。
 - `dump` コマンド。 `dump` は、 `where` コマンドと同じ簡単なフォーマットを使用します。これは将来、 `pretty-print` を使用するように変換される可能性があります。この制限は IDE の局所変数ウィンドウには適用されません。
- 入れ子の値は整形出力されません。 dbx には入れ子のフィールドのアドレスを計算するインフラストラクチャーがありません。
- `db_pretty_print()` は `-g` オプションを使用してコンパイルする必要があります。 dbx はパラメータシングニチャーにアクセスする必要があります。
- `db_pretty_print()` 関数では `NULL` を返すことができます。
- `db_pretty_print()` 関数に渡されるメインポインタは `NULL` 以外であることが保証されていますが、そうでない場合は、完全に初期化されていないオブジェクトを指したままになる可能性があります。
- dbx 環境変数 `output_pretty_print_fallback` はデフォルトで `on` に設定され、 `pretty-print` が失敗した場合、 dbx は標準フォーマットを使用することを意味します。この環境変数が `off` に設定されている場合、 `pretty-print` が失敗すると dbx はエラーメッセージを発行します。
- 次のいずれかの理由により、 `pretty-print` が失敗する可能性があります。これらは検出および回復が可能です。
 - `pretty-print` 関数が見つからない。
 - 整形出力する式のアドレスを取得できない。
 - 関数呼び出しが直ちに戻らない。これは、不正なオブジェクトが検出されたときに、 `pretty-print` 関数が堅牢でない場合に発生するセグメント例外を示している可能性があります。ユーザーブレークポイントを示している可能性もあります。
 - `pretty-print` 関数が `NULL` を返した。
 - `pretty-print` 関数が、 dbx が間接参照できないポインタを返した。
 - コアファイルがデバッグ中である。

前述のような失敗では、関数呼び出しが直ちに戻らない場合を除くすべての状況で、メッセージは表示されず、 dbx は標準のフォーマットを使用します。ただし、 `output_pretty_print_fallback` 環境変数が `off` に設定されている場合、 `pretty-print` が失敗すると、 dbx はエラーメッセージを発行します。

ただし、dbx 環境変数 `output_pretty_print` を `on` に設定せずに、`print -p` コマンドを使用した場合、dbx は壊れている関数で停止するため、失敗の原因を診断できません。次に、`pop -c` コマンドを使用すると、呼び出しを取り消すことができます。

- `db_pretty_print()` 関数は、先頭のパラメータの型に基づいて、明確にする必要があります。C では、関数をファイルスタティックとして記述することで、関数を多重定義できます。

C++ での表示

C++ では、オブジェクトポインタに 2 つの型があります。1 つは「静的な型」で、ソースコードに定義されています。もう 1 つは「動的な型」です。dbx は、動的な型のオブジェクトに関する情報を提供できる場合があります。

通常、オブジェクトに仮想関数テーブルの `vtable` が含まれる場合、dbx はこの `vtable` 内の情報を使用して、オブジェクトの型を正しく知ることができます。

`print`、`display`、または `watch` コマンドは、`-r` (再帰的) オプション付きで使用できます。その場合、dbx はクラスによって直接定義されたデータメンバーすべてと、基底クラスから継承されたものを表示することができます。

これらのコマンドには、`-d` または `+d` オプションも使用できます。これは、dbx 環境変数 `output_derived_type` でデフォルト動作を切り替えることができます。

プロセスが何も実行されていないときに、`-d` フラグを使用するか、または dbx 環境変数 `output_dynamic_type` を `on` に設定すると、プロセスがないときに動的情報にアクセスすることは不可能なため、プログラムが実行可能な状態ではないことを表すエラーメッセージが出されます。仮想継承から動的な型の検索を試みると、クラスポインタの不正なキャストを表すエラーメッセージが生成されます (仮想基底クラスから派生クラスへのキャストは C++ では無効です)。

C++ プログラムにおける無名引数を評価する

C++ では、無名の引数を持つ関数を定義できます。たとえば、次のようにします。

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

無名の引数はプログラム内のほかの場所では使用できませんが、dbx は無名引数を評価できる形式にコード化します。その形式は次のとおりです。ここで、dbx は `%n` に整数を割り当てます。

```
_ARG%n
```

コンパイラによって割り当てられた引数名を入手するには、対象の関数名を指定した `whatis` コマンドを実行します。

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

詳細については、383 ページの「`whatis` コマンド」を参照してください。

無名の関数引数を評価 (表示) するには、次のようにします。

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

ポインタを間接参照する

ポインタを間接参照すると、ポインタが指している内容に格納された値を参照できます。

ポインタを間接参照すると、`dbx` は、コマンド区画に評価結果を表示します。次の例では、`t` の指す値が表示されます。

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

式を監視する

プログラムが停止するたびに式の値を監視することにより、特定の式または変数がいつどのように変化するかを効果的に知ることができます。`display` コマンドは、指定されている 1 つまたは複数の式または変数を監視するように `dbx` に命令します。監視は、`undisplay` コマンドによって取り消されるまで続けられます。

プログラムが停止するたびに変数または式の値を表示するには、次のようにします。

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

一度に複数の変数を監視できます。オプションを指定しないで `display` コマンドを使用すると、監視対象のすべての式が表示されます。

詳細については、312 ページの「`display` コマンド」を参照してください。

表示を取り消す (非表示)

監視している変数の値の表示は、`undisplay` コマンドで「表示」を取り消すまで続けられます。特定の式だけを表示しないようにすることも、現在監視しているすべての式の表示を中止することも可能です。

特定の変数または式の表示をオフにするには、次のようにします。

```
undisplay expression
```

現在監視しているすべての変数の表示をオフにするには、次のようにします。

```
undisplay 0
```

詳細については、[378 ページの「undisplay コマンド」](#)を参照してください。

変数に値を代入する

変数に値を代入するには、次のように入力します。

```
assign variable = expression
```

配列を評価する

配列の評価は、ほかの種類の変数进行评估する場合と同じ方法で行います。

Fortran の配列の例:

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

配列を評価するには、`print` コマンドを使用します。たとえば、次のようにします。

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

`dbx` コマンドの `print` を使用して、大型の配列の一部を評価することができます。配列を評価するには、次の操作を行います。

- 配列の断面化 - 多次元配列から任意の矩形ブロックまたは n 次元の領域を取り出して出力します。
- 配列の刻み - 指定された配列の断面 (配列全体のこともあります) から決まったパターンで特定の要素だけを取り出して出力します。

刻みは配列の断面化を行うときに必要に応じて指定することができます (刻みのデフォルト値は 1 で、その場合は各要素を出力します)。

配列の断面化

C、C++、Fortran では、`print`、`display`、および `watch` コマンドによって、配列の断面化を行うことができます。

C と C++ での配列の断面化の構文

配列の各次元を断面化するための `print` コマンドの完全な構文は次のとおりです。

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

ここで

<i>array-expression</i>	配列またはポインタ型に評価されるべき式
<i>first-expression</i>	印刷される最初の要素。デフォルトは 0
<i>last-expression</i>	印刷される最後の要素。その上限にデフォルト設定
<i>stride-expression</i>	刻み幅の長さ (スキップされる要素の数は <i>stride-expression</i> -1)。デフォルトは 1

最初、最後、および刻み幅の各式は、整数に評価されなければならない任意の式です。

たとえば、次のようにします。

```
(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 のための配列断面化構文

配列の各次元を断面化するための `print` コマンドの完全な構文は次のとおりです。

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

ここで

<i>array-expression</i>	配列型に評価される式
<i>first-expression</i>	範囲内の最初の要素は、出力される最初の要素下限にデフォルト設定
<i>last-expression</i>	範囲内の最後の要素。ただし刻み幅が1でない場合、出力される最後の要素とはなりません。その上限にデフォルト設定
<i>stride-expression</i>	刻み幅。デフォルトは1

最初、最後、および刻み幅の各式は、整数に評価されなければならない任意の式です。 n 次元の断面については、カンマで各断面の定義を区切ります。

たとえば、次のようにします。

```
(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
```

行と列を指定するには、次のように入力します。

```
demo% f95 -g -silent ShoSli.f
```

```
demo% dbx a.out
```

```
a.out のシンボル情報を読んでいます
```

```
(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
実行中: a.out
MAIN で停止しました。行位置: 7 ファイル "ShoSli.f"
7      DO row = 1, 3
```

行 3 を印刷するには、次のように入力します。

```
(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)
```

列 4 を印刷するには、次のように入力します。

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

配列の断面

2次元の C++ の矩形配列の断面の例を示します。ここでは、刻み値が省略され、デフォルト値の 1 が使用されます。

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

このコマンドは、大型配列の要素のブロックを出力します。*stride-expression* が省略され、デフォルトの刻み値である 1 が使用されていることに注意してください。

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

最初の2つの式(201:203)は、この2次元配列の第1次元(3行で構成される列)を指定します。配列の断面は行201から始まり、行203で終わります。次の2つの式(101:105)は最初の組とコンマで区切られ、第2次元の配列の断面を定義します。配列の断面は列101から始まり、列105で終わります。

刻み幅

print コマンドで刻み幅を指定すると、配列の断面に含まれる特定の要素だけが評価されます。

配列の断面のための構文の3番目の式(*stride-expression*)は、刻み幅の長さを指定します。*stride-expression*の値は印刷する要素を指定します。刻み幅のデフォルト値は1です。このとき、指定された配列の断面のすべての要素が評価されます。

ここに、前述の例で使用したものと同一配列があります。今度は、print コマンドの第2次元の配列の断面の定義に刻み幅の値として2を加えます。

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

図で示すとおり、刻み値として2を指定すると、各行を構成する要素が1つおきに出力されます。

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

print コマンドの配列の断面の定義を構成する式を省略すると、配列の宣言されたサイズに等しいデフォルト値が使用されます。このような簡易構文を使用した例を以下に示します。

1次元配列の場合

```
print arr
```

デフォルトの境界で配列全体を出力します。

```
print arr(:)
```

デフォルトの境界とデフォルトの刻み(1)で、配列全体を出力します。

```
print arr::stride-expression)
```

配列全体を *stride-expression* で指定された刻み幅で出力します。

2次元配列の場合、次のコマンドは配列全体を出力します。

```
print arr
```

2次元配列の第2次元を構成する要素を2つおきに出力します。

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

実行時検査

実行時検査 (RTC) を行うと、開発段階においてネイティブコードアプリケーションの実行時エラー (メモリアクセスエラー、メモリーリークなど) を自動的に検出できます。メモリーの使用状況も監視できます。Java コードでは、実行時検査を行うことはできません。

この章は次の各節から構成されています。

- 130 ページの「概要」
- 131 ページの「実行時検査」
- 134 ページの「アクセス検査の使用」
- 137 ページの「メモリーリークの検査」
- 142 ページの「メモリー使用状況検査の使用」
- 144 ページの「エラーの抑止」
- 146 ページの「子プロセスにおける RTC の実行」
- 149 ページの「接続されたプロセスへの RTC の使用」
- 150 ページの「RTC での修正継続機能の使用」
- 152 ページの「実行時検査アプリケーションプログラミングインタフェース」
- 152 ページの「バッチモードでの RTC の使用」
- 154 ページの「トラブルシューティングのヒント」
- 155 ページの「実行時検査の制限」
- 158 ページの「RTC エラー」

注 - 実行時検査が行えるのは Solaris OS の動作しているプラットフォームのみです。Linux OS の動作しているプラットフォームでは行えません。

概要

RTCは、統合的なデバッグ機能であり、コレクタによるパフォーマンスデータの収集時を除けば、実行時にあらゆるデバッグ機能を利用できます。

次に、RTCの機能を簡単に説明します。

- メモリアクセスエラーを検出する
- メモリーリークを検出する
- メモリー使用に関するデータを収集する
- すべての言語で動作する
- マルチスレッドコードで動作する
- 再コンパイル、再リンク、またはメイクファイルの変更が不要である

-gフラグを付けてコンパイルすると、RTCエラーメッセージでのソース行番号の関連性が与えられます。RTCは、最適化 -Oフラグによってコンパイルされたプログラムを検査することもできます。-gオプションによってコンパイルされていないプログラムについては、特殊な考慮事項があります。

RTCを実行するには、checkコマンドを使用します。

RTCを使用する場合

大量のエラーが一度に検出されないようにするには、RTCを開発サイクルの初期の段階で使用します。この段階では、プログラムの構成要素となる個々のモジュールを開発します。この各モジュールを実行する単位テストを作成し、RTCを各モジュールごとに1回ずつ使用して検査を行います。これにより、一度に処理するエラーの数が減ります。すべてのモジュールを統合して完全なプログラムにした場合、新しいエラーはほとんど検出されません。エラー数をゼロにしたあとでモジュールに変更を加えた場合にのみ、RTCを再度実行してください。

RTCの必要条件

RTCを使用するには、次の要件を満たす必要があります。

- Sunのコンパイラを使用してコンパイルされたプログラム。
- libcを動的にリンクしている。
- libcの標準関数 malloc、free、realloc を利用するか、これらの関数を基にアロケータを使用する。RTCでは、ほかのアロケータはアプリケーションプログラミングインタフェース (API) で操作します。[152 ページの「実行時検査アプリケーションプログラミングインタフェース」](#)を参照してください。
- 完全にストリップされていないプログラム。strip -x によってストリップされたプログラムは使用できません。

実行時検査の制限については、155 ページの「[実行時検査の制限](#)」を参照してください。

実行時検査

実行時検査を使用するには、使用したい検査の種類を指定します。

メモリー使用状況とメモリーリーク検査を有効化

メモリー使用状況とメモリーリークの検査をオンにするには、次を入力します。

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

MUCかMLCがオンになっている場合、`showblock` コマンドを実行する、所定のアドレスにおけるヒープブロックに関する詳細情報を表示できます。この詳細情報では、ブロックの割り当て場所とサイズを知ることができます。詳細については、355 ページの「[showblock コマンド](#)」を参照してください。

メモリーアクセス検査を有効化

メモリーアクセス検査をオンにするには、次を入力します。

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

すべての RTC を有効化

メモリーリーク、メモリー使用状況、およびメモリーアクセスの各検査をオンにするには、次のように入力します。

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

詳細については、292 ページの「[check コマンド](#)」を参照してください。

RTC を無効化

RTC をすべて無効にするには、次のように入力します。

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

詳細については、377 ページの「[uncheck コマンド](#)」を参照してください。

プログラムを実行

目的のタイプの RTC を有効にしてテストするプログラムを実行します。この場合、ブレークポイントを設定してもしなくてもかまいません。

プログラムは正常に動作しますが、それぞれのメモリアクセスが発生する直前にその妥当性チェックが行われるため、動作速度は遅くなります。無効なアクセスを検出すると、dbx はそのエラーの種類と場所を表示します。制御はユーザーに戻ります (dbx 環境変数 `rct_auto_continue` が on になっている場合を除く (62 ページの「dbx 環境変数の設定」を参照))。

次に、dbx コマンドを実行します。where コマンドでは現在のスタックトレースを呼び出すことができます。また print を実行すれば変数を確認できます。エラーが致命的でなければ、cont コマンドでプログラムの処理を続行します。プログラムは次のエラーまたはブレークポイントまで、どちらか先に検出される場所まで実行されます。詳細については、303 ページの「cont コマンド」を参照してください。

`rct_auto_continue` 環境変数が on に設定されている場合、RTC はそのままエラーを求めて自動的に続行されます。検出したエラーは、dbx 環境変数 `rct_error_log_name` で指定したファイルにリダイレクトされます (62 ページの「dbx 環境変数の設定」を参照)。デフォルトログファイル名は、`/tmp/dbx.errlog.uniqueid` です。

RTC エラーの報告が不要な場合は、`suppress` コマンドを使用します。詳細については、366 ページの「suppress コマンド」を参照してください。

次の例は、`hello.c` と呼ばれるプログラムのメモリアクセス検査とメモリー使用状況検査をオンにする方法を示しています。

```
% 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
ld.so.1 の読み込み中
librt.so のシンボル情報を読んでいます
libw.so.1 の読み込み中
libdl.so.1 の読み込み中

(dbx) check -access
アクセス検査 - ON
(dbx) check -memuse
メモリー使用状況検査 - ON
(dbx) run Running: hello
(プロセス id 18306)
実行時検査を有効にしています...done
非初期化領域からの読み取り (rui):
4 バイト読み取りをアドレス 0xeffff068 でしようとした
それは 96 バイト現スタックポインタより上で
変数は 'j' です。
現関数: access_error
    29     i = j;
(dbx) cont
hello world
メモリーリーク検査中...
実際のリークの報告      (実際のリーク:      1 合計サイズ:      32 バイト)

```

```

合計      ブロック   リーク      割り当て呼び出しスタック
サイズ    数           ブロック
           アドレス
=====
          32      1      0x21aa8 memory_leak < main

起り得るリークの報告 (起り得るリーク:      0 合計サイズ:      0 バイト)
メモリー使用状況検査中...
ブロック使用量の報告 (ブロック使用量:      2 合計サイズ:      44 バイト)

合計      割合   ブロック   平均   割り当て呼び出しスタック
サイズ   %     数       サイズ
=====
          32  72%     1       32 memory_use < main
          12  27%     1       12 memory_use < main

```

実行完了。終了コードは 0 です。

関数 `access_error()` は、初期化される前の変数 `j` を読み取ります。RTC は、このアクセスエラーを非初期化領域からの読み取り (`rui`) として報告します。

関数 `memory_leak()` は、終了する前に `local` を解放 (`free()`) しません。`memory_leak()` が終了してしまうと、`local` がスコープ外になり、行 20 で確保したブロックがリークになります。

プログラムは、常にスコープ内にある大域変数 `hello1` と `hello2` を使用します。これらの変数はいずれも、使用中ブロック (`biu`) として報告される割り当て済みメモリーを動的に指します。

アクセス検査の使用

アクセス検査では、読み取り、書き込み、割り当て、解放の各操作を監視することによって、プログラムがメモリーに正しくアクセスするかどうかを検査します。

プログラムは、さまざまな方法で間違っってメモリーを読み取ったり、メモリーに書き込んだりすることがあります。このようなエラーをメモリーアクセスエラーといいます。たとえば、ヒープブロックの `free()` 呼び出しを使用したり、または関数が局所変数にポインタを返したために、プログラムが参照するメモリーブロックの割り当てが解放されている可能性があります。アクセスエラーはプログラムでワイルドポインタの原因になり、間違っった出力やセグメント不正など、プログラムの異常な動作を引き起こす可能性があります。メモリーアクセスエラーには、検出が非常に困難なものもあります。

RTCは、プログラムによって使用されているメモリーの各ブロックの情報を追跡するテーブルを管理します。プログラムがメモリー操作を行うと、RTCは関係するメモリーブロックの状態に対してその操作が有効かどうかを判断します。メモリーの状態として次のものがあります。

- 未割り当て(初期)状態。メモリーは割り当てられていません。この状態のメモリーはプログラムが所有していないため、読み取り、書き込み、解放のすべての操作が無効です。
- 割り当て済み/未初期化。メモリーはプログラムに割り当てられていますが、初期化されていません。書き込み操作と解放操作は有効ですが、初期化されていないので読み取りは無効です。たとえば、関数に入るときに、スタック上にメモリーが割り当てられますが、初期化はされません。
- 読み取り専用。読み取りは有効ですが、書き込みと解放は無効です。
- 割り当て済み/初期化済み。割り当てられ、初期化されたメモリーに対しては、読み取り、書き込み、解放のすべての操作が有効です。

RTCを使用してメモリーアクセスエラーを見つける方法は、コンパイラがプログラム中の構文エラーを見つける方法と似ています。いずれの場合でも、プログラム中のエラーが発生した位置と、その原因についてのメッセージとともにエラーのリストが生成され、リストの先頭から順に修正していかなければなりません。これは、あるエラーがほかのエラーと関連して連結されたような作用があるためです。連結の最初のエラーが先頭の原因となり、そのエラーを修正することにより、そのエラーから派生したほかの問題も解決されることがあります。

たとえば、初期化されていないメモリーの読み取りにより、不正なポインタが作成されるとします。すると、これが原因となって不正な読み取りと書き込みのエラーが発生し、それがまた原因となってさらに別の問題が発生するというようなことになる場合があります。

メモリーアクセスエラーの報告

メモリーアクセスエラーを検出するとRTCは次の情報を出力します。

エラー	情報
type	エラーの種類。
access	試みられたアクセスの種類(読み取りまたは書き込み)。
size	試みられたアクセスのサイズ。
address	試みられたアクセスのアドレス
size	リークしたブロックのサイズ

エラー	情報
detail	アドレスについてのさらに詳しい情報。たとえば、アドレスがスタックの近くに存在する場合、現在のスタックポインタからの相対位置が与えられます。アドレスが複数存在する場合、一番近いブロックのアドレス、サイズ、相対位置が与えられます。
stack	エラー時の呼び出しスタック (バッチモード)。
allocation	addr がヒープにある場合、もっとも近いヒープブロックの割り当てトレースが与えられます。
location	エラーが発生した位置。行が特定できる場合には、ファイル名、行番号、関数が示されます。行番号がわからないときは関数とアドレスが示されます。

代表的なアクセスエラーは次のとおりです。

非初期化領域からの読み取り (rui):

4 バイト読み取りをアドレス 0xeffffee50 でしようとしてしました

それは 96 バイト現スタックポインタより上です

変数は 'j' です。

現関数: rui

```
12          i = j;
```

メモリーアクセスエラー

RTC は、次のメモリーアクセスエラーを検出します。

- rui (160 ページの「非初期化メモリーからの読み取り (rui) エラー」を参照)
- rua (159 ページの「非割り当てメモリーからの読み取り (rua) エラー」を参照)
- wua (160 ページの「非割り当てメモリーへの書き込み (wua) エラー」を参照)
- wro (160 ページの「読み取り専用メモリーへの書き込み (wro) エラー」を参照)
- mar (159 ページの「境界整列を誤った読み取り (mar) エラー」を参照)
- maw (159 ページの「境界整列を誤った書き込み (maw) エラー」を参照)
- duf (158 ページの「重複解放 (duf) エラー」を参照)
- baf (158 ページの「不正解放 (baf) エラー」を参照)
- maf (158 ページの「境界整列を誤った解放 (maf) エラー」を参照)
- oom (159 ページの「メモリー不足 (oom) エラー」を参照)

注-RTCでは、配列境界チェックは行いません。したがって、配列境界侵害はアクセスエラーにはなりません。

メモリーリークの検査

メモリーリークとは、プログラムで使用するために割り当てられているが、プログラムのデータ領域中のいずれも指していないポインタを持つ、動的に割り当てられたメモリーブロックを言います。そのようなブロックは、メモリーのどこに存在しているかプログラムにわからないため、プログラムに割り当てられていても使用することも解放することもできません。RTCはこのようなブロックを検知し、報告します。

メモリーリークは仮想メモリーの使用を増やし、一般的にメモリーの断片化を招きます。その結果、プログラムやシステム全体のパフォーマンスが低下する可能性があります。

メモリーリークは、通常、割り当てメモリーを解放しないで、割り当てブロックへのポインタを失うと発生します。メモリーリークの例を次に示します。

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

    strcpy(s, "hello world");

    return; /* s が解放されていない。foo が戻るとき、          */
           /* malloc されたブロックを指しているポインタが存在しないため、          */
           /* ブロックはリークする                                                  */
}
```

リークは、APIの不正な使用が原因で起こる可能性があります。

```
void
printcwd()
{

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

    return; /* libc の関数 getcwd() は、最初の引数が          */
           /* NULL の場合 malloc された領域へのポインタを返す */
           /* プログラムは、これを解放する必要がある。この場合、          */
           /* ブロックが解放されていないため、結果的にリークになる。*/
}
```

メモリーリークを防ぐには、必要のないメモリーは必ず解放します。また、メモリーを確保するライブラリ関数を使用する場合は、メモリーを解放することを忘れないでください。

解放されていないブロックを「メモリーリーク」と呼ぶこともあります。ただし、この定義はあまり使用されません。プログラムが短時間で終了する場合でも、通常のプログラミングではメモリーを解放しないからです。プログラムにそのブロックに対するポインタがある場合、RTCはそのようなブロックはメモリーリークとして報告しません。

メモリーリーク検査の使用

RTCでは、次のメモリーリークエラーを検出します。

- [mel \(161 ページの「メモリーリーク \(mel\) エラー」を参照\)](#)
- [air \(161 ページの「レジスタ中のアドレス \(air\)」を参照\)](#)
- [aib \(161 ページの「ブロック中のアドレス \(aib\)」を参照\)](#)

注-RTCにおけるリーク検出の対象はmallocメモリーのみです。mallocを使用していないプログラムでRTCを行なってもメモリーリークは検出されません。

リークの可能性

RTCが「リークの可能性」として報告するエラーには2種類あります。1つは、ブロックの先頭を指すポインタが検知されず、ブロックの内部を指しているポインタが見つかった場合です。これは、ブロック中のアドレス(aib)エラーとして報告されます。このようなブロック内部を指すポインタが見つかった場合は、プログラムに実際にメモリーリークが発生しています。ただし、プログラムによってはポインタに対して故意にそのような動作をさせている場合があり、これは当然メモリーリークではありません。RTCはこの違いを判別できないため、本当にリークが発生しているかどうかはユーザー自身の判断で行う必要があります。

もう1つのリークの種類は、レジスタ中のアドレス(air)エラーとして報告されるリークです。これは、ある領域を指すポインタがデータ空間中には存在せず、レジスタ内に存在する場合です。レジスタがブロックを不正に指していたり、古いメモリーポインタが残っている場合には、実際にメモリーリークが発生しています。ただし、コンパイラが最適化のために、ポインタをメモリーに書き込むことなく、レジスタのブロックに対して参照させることがあります。この場合はメモリーリークではありません。プログラムが最適化され、showleaksコマンドでエラーが報告された場合のみ、リークでない可能性があります。詳細については、[355 ページの「showleaks コマンド」](#)を参照してください。

注-RTCリーク検査では、標準のlibcのmalloc/free/realloc関数またはアロケータをこれらの関数に基づいて使用する必要があります。ほかのアロケータについては、152ページの「実行時検査アプリケーションプログラミングインタフェース」を参照してください。

リークの検査

メモリーリーク検査がオンの場合、メモリーリークの走査は、テスト中のプログラムが終了する直前に自動的に実行されます。検出されたリークはすべて報告されます。プログラムを、killコマンドによって強制的に終了しないでください。次に、典型的なメモリーリークエラーによるメッセージを示します。

```
メモリーリーク (mel):  
大きさ 6 バイトのリークのあるブロックをアドレス 0x21718 に発見  
割り当て時のスタックの状態:  
[1] foo() 行番号 63 test.c  
[2] main() 行番号 47 test.c
```

プログラムには通常main (FORTRAN 77ではMAIN)手続きが存在します。プログラムはexit(3)が呼び出されるか、mainから返った時点で終了します。いずれの場合でも、mainのすべての局所変数はプログラムが停止するまでスコープから出ず、それらを指す特定のヒープブロックはすべてメモリーリークとして報告されます。

main()に割り当てられているヒープブロックはプログラムでは解放しないのが一般的です。これらのヒープブロックはプログラムが停止するまでスコープ内に残り、プログラムの停止後オペレーティングシステムによって自動的に解放されるためです。main()に割り当てられたブロックがメモリーリークとして報告されないようにするには、main()が終了する直前にブレークポイントを設定しておきます。プログラムがそこで停止したとき、RTCのshowleaksコマンドを実行すれば、main()とそこで呼び出されるすべての手続きで参照されなくなったヒープブロックのすべてが表示されます。

詳細については、355ページの「showleaks コマンド」を参照してください。

メモリーリークの報告を理解する

リーク検査を有効にすると、プログラムの終了時にリークレポートが自動的に生成されます。killコマンドでプログラムを終了した場合を除き、リークの可能性がすべて報告されます。レポートの詳細レベルは、dbx環境変数rtc_mel_at_exit (62ページの「dbx環境変数の設定」を参照)で制御します。デフォルトで、冗長リークレポートが生成されます。

レポートは、リークのサイズによってソートされます。実際のメモリーリークが最初に報告され、次に可能性のあるリークが報告されます。詳細レポートには、スタックトレース情報の詳細が示されます。行番号とソースファイルが使用可能であれば、これらも必ず含まれます。

次のメモリーリークエラー情報が、2種類の報告のどちらにも含まれます。

情報	内容の説明
サイズ	リークしたブロックのサイズ
場所	リークしたブロックが割り当てられた場所
アドレス	リークしたブロックのアドレス
スタック	割り当て時の呼び出しスタック。check -frames によって制約される

次に、対応する簡易メモリーリークレポートを示します。

実際のリークの報告 (実際のリーク: 3 合計サイズ: 2427 バイト)

合計 サイズ	ブロック 数	リーク ブロック	割り当て呼び出しスタック アドレス
1852	2	-	true_leak < true_leak
575	1	0x22150	true_leak < main

起こり得るリークの報告 (起こり得るリーク: 1 合計サイズ: 8 バイト)

合計 サイズ	ブロック 数	リーク ブロック	割り当て呼び出しスタック アドレス
8	1	0x219b0	in_block < main

次に、典型的な詳細リークレポートを示します。

実際のリークの報告 (実際のリーク: 3 合計サイズ: 2427 バイト)

メモリーリーク (me1):

大きさ 1 バイトのリークのあるブロックをアドレス 0x20f18 に発見
割り当て時のスタックの状態:

```
[1] true_leak() 行番号 220 "leaks.c"
[2] true_leak() 行番号 224 "leaks.c"
```

メモリーリーク (me1):

大きさ 575 バイトのリークのあるブロックをアドレス 0x22150 に発見

割り当て時のスタックの状態:

```
[1] true_leak() 行番号 220 "leaks.c"
[2] main()      行番号 87  "leaks.c"
```

起こり得るリークの報告 (起こり得るリーク: 1 合計サイズ: 8 バイト)

メモリーリークの可能性 -- ブロック中のアドレス (aib):
 大きさ 4 バイトのリークのあるブロックをアドレス 0x219b0 に発見
 割り当て時のスタックの状態:

```
[1] in_block() 行番号 177 "leaks.c"
[2] main()     行番号 100 "leaks.c"
```

リークレポートの生成

showleaks コマンドを使用すると、いつでもリークレポートを要求することができます。このコマンドは、前回の showleaks コマンド以降の新しいメモリーリークを報告するものです。詳細については、[355 ページの「showleaks コマンド」](#)を参照してください。

リークレポート

リークレポートの数が多くなるのを避けるため、RTCは同じ場所で割り当てられたリークを自動的に1つにまとめて報告します。1つにまとめるか、それぞれ各リークごとに報告するかは、一致フレーム数引数によって決まります。この引数は、check-leaks コマンドを実行する際は -match *m* オプション、showleaks コマンドを実行する際は -m オプションで指定します。呼び出しスタックが2つ以上のリークを割り当てる際に *m* 個のフレームと一致した場合は、リークは1つにまとめて報告されます。

次の3つの呼び出しシーケンスを考えてみます。

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

これらのブロックがすべてメモリーリークを起こす場合、*m* の値によって、これらのリークを別々に報告するか、1つのリークが繰り返されたものとして報告するかが決まります。*m* が2のとき、ブロック1とブロック2のリークは1つのリークが繰り返されたものとして報告されます。これは、malloc() の上にある2つのフレームが共通しているためです。ブロック3のリークは、c() のトレースがほかのブロックと

一致しないので別々に報告されます。 m が2よりも大きい場合、RTCはすべてのリークを別々に報告します (mallocはリークレポートでは表示されません)。

一般に、 m の値が小さければリークのレポートもまとめられ、 m の値が大きければまとめられたリークレポートが減り、別々のリークレポートが生成されます。

メモリーリークの修正

RTCからメモリーリーク報告を受けた場合にメモリーリークを修正する方法についてのガイドラインを次に示します。

- リークの修正でもっとも重要なことは、リークがどこで発生したかを判断することです。作成されるリーク報告は、リークが発生したブロックの割り当てトレースを示します。リークが発生したブロックは、ここから割り当てられたことになります。
- 次に、プログラムの実行フローを見て、どのようにそのブロックを使用したかを調べます。ポインタが失われた箇所が明らか場合は簡単ですが、それ以外の場合は `showleaks` コマンドを使用してリークの検索範囲を狭くすることができます。 `showleaks` コマンドは、デフォルトでは前回このコマンドを実行したあとに検出されたリークのみを報告するため、 `showleaks` を繰り返し実行することにより、ブロックがリークを起こした可能性のある範囲が狭まります。

詳細については、[355 ページの「showleaks コマンド」](#)を参照してください。

メモリー使用状況検査の使用

メモリー使用状況検査は、使用中のヒープメモリーすべてを確認することができます。この情報によって、プログラムのどこでメモリーが割り当てられたか、またはどのプログラムセクションが大半の動的メモリーを使用しているかを知ることができます。この情報は、プログラムの動的メモリー消費を削減するためにも有効であり、パフォーマンスの向上に役立ちます。

メモリー使用状況検査は、パフォーマンス向上または仮想メモリーの使用制御に役立ちます。プログラムが終了したら、メモリー使用状況レポートを生成できます。メモリー使用情報は、メモリーの使用状況を表示させるコマンド (`showmemuse`) を使用して、プログラムの実行中に随時取得することもできます。詳細については、[356 ページの「showmemuse コマンド」](#)を参照してください。

メモリー使用状況検査をオンにすると、リーク検査もオンになります。プログラム終了時のリークレポートに加えて、使用中ブロック (biu) レポートも得ることができます。デフォルトでは、使用中ブロックの簡易レポートがプログラムの終了時に生成されます。メモリー使用状況レポートの詳細を制御するには、`dbx` 環境変数 `rtc_biu_at_exit` ([62 ページの「dbx 環境変数の設定」](#)を参照) を使用します。

次に、典型的な簡易メモリー使用状況レポートを示します。

ブロック使用量の報告 (ブロック使用量: 5 合計サイズ: 40 バイト)

合計 サイズ	割合	ブロック %	平均 数	割り当て呼び出しスタック サイズ
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

次に、対応する詳細メモリー使用状況レポートを示します。

ブロック使用量の報告 (ブロック使用量: 5 合計サイズ: 40 バイト)

ブロック使用状況(biu):

2 個のブロックを見つけました。合計 16 bytes (合計 40.00%; 平均ブロックサイズ 8)
割り当て時のスタックの状態:

- [1] nonleaks() 行番号 182 "memuse.c"
- [2] nonleaks() 行番号 185 "memuse.c"

ブロック使用状況(biu):

サイズ 8 bytes のブロックをアドレス 0x21898 で見つめました (合計 20.00%)
割り当て時のスタックの状態:

- [1] nonleaks() 行番号 182 "memuse.c"
- [2] main() 行番号 74 "main.c"

ブロック使用状況(biu):

サイズ 8 bytes のブロックをアドレス 0x21958 で見つめました (合計 20.00%)
割り当て時のスタックの状態:

- [1] cycle_leaks() 行番号 154 "memuse.c"
- [2] main() 行番号 118 "main.c"

ブロック使用状況(biu):

サイズ 8 bytes のブロックをアドレス 0x21978 で見つめました (合計 20.00%)
割り当て時のスタックの状態:

- [1] cycle_leaks() 行番号 155 "memuse.c"
- [2] main() 行番号 118 "main.c"

showmemuse コマンドを使用すると、メモリー使用状況レポートをいつでも要求できます。

エラーの抑止

RTCはエラーレポートの数や種類を限定するよう、エラーの抑制機能を備えています。エラーが発生してもそれが抑制されている場合は、エラーは無視され、報告されずにプログラムは継続します。

エラーは `suppress` コマンド (366 ページの「`suppress` コマンド」を参照) で抑止できます。

エラー抑止を取り消すには、`unsuppress` コマンド (380 ページの「`unsuppress` コマンド」を参照) を使用します。

抑止機能は同じデバッグ節内の `run` コマンドの実行期間中は有効ですが、`debug` コマンドを実行すると無効になります。

抑止のタイプ

次の抑制機能があります。

スコープと種類による抑制

どのエラーを抑止するかを指定する必要があります。次のように、プログラムのどの部分に抑制を適用するかを指定できます。

オプション	内容の説明
大域	スコープが指定されていないと全体のスコープが対象になり、すべてのプログラムに適用されます。
ロードオブジェクト	共有ライブラリなど、すべてのロードオブジェクトが対象になります。
ファイル	特定のファイルのすべての関数が対象になります。
関数	特定の関数が対象になります。
行	特定のソース行が対象になります。
アドレス	特定のアドレスが対象になります。

最新エラーの抑止

デフォルトでRTCを実行すると、最新のエラーで同じエラーが繰り返し報告されることがなくなります。この機能は、`dbx` 環境変数 `rtc_auto_suppress` で制御します。`rtc_auto_suppress` が `on` のとき (デフォルト)、特定箇所の特定エラーは最初の発生時にだけ報告され、そのあと同じエラーが同じ場所で発生しても報告が繰り返されることはありません。最新エラーを抑止すると、繰り返し実行するループに1つのエラーがあっても、それが何度も報告されることがなく、便利です。

エラー報告回数の制限

dbx 環境変数 `rtc_error_limit` では、報告されるエラーの回数を制限します。エラー制限は、アクセスエラーとリークエラーに別々に設定します。たとえば、エラー制限を 5 に設定すると、プログラムの終了時のリークレポートと、`showleaks` コマンドの実行ごとに、アクセスエラーとリークエラーがそれぞれ最高で 5 回報告されます。デフォルトは 1000 です。

エラー抑止の例

次の例では、`main.cc` はファイル名、`foo` と `bar` は関数を示し、`a.out` は実行可能ファイルの名前を示します。

割り当てが関数 `foo` で起こったメモリーリークは報告しません。

```
suppress mel in foo
```

`libc.so.1` から割り当てられた使用中のブロック報告を抑止します。

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

`a.out` の非初期化機能からの読み取りを抑止します。

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

ファイル `main.cc` の非割り当てメモリーからの読み取りを報告しません。

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

`main.cc` の行番号 10 での重複解放を抑止します。

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

関数 `bar` のすべてのエラー報告を抑止します。

```
suppress all in bar
```

詳細については、[366 ページの「suppress コマンド」](#)を参照してください。

デフォルトの抑止

RTC では、`-g` オプション (記号) を指定してコンパイルを行わなくてもすべてのエラーを検出できます。しかし、非初期化メモリーからの読み取りなど、正確さを保証するのに記号 (`-g`) 情報が必要な特定のエラーもあります。このため、`a.out` の `rui` や共有ライブラリの `rui`、`aib`、`air` など特定のエラーは、記号情報が取得できない場合は、デフォルトで抑制されます。この動作は、`suppress` や `unsuppress` コマンドの `-d` オプションを使用することで変更できます。

たとえば、次を実行すると、RTC は記号情報が存在しない (-g オプションを指定しないでコンパイルした) コードについて「非初期化メモリーからの読み取り (rui)」を抑制しません。

```
unsuppress -d rui
```

詳細については、[380 ページの「unsuppress コマンド」](#)を参照してください。

抑止によるエラーの制御

プログラムが大きい場合、エラーの数もそれに従って多くなることが予想されます。このような場合は、このような場合は、`suppress` コマンドを使用することにより、エラーレポートの数を管理しやすい大きさまで抑制し、一度で修正するエラーを制限します。

たとえば、一度で検出するエラーをタイプによって制限できます。一般的によくあるエラーのタイプは `rui`、`rua`、`wua` に関連したもので、この順序で検出されます。`rui` エラーはそれほど致命的なエラーではなく、このエラーが検出されてもたいていの場合プログラムは問題なく実行終了します。それに比べて `rua` と `wua` エラーは不正なメモリーアドレスにアクセスし、ある種のコーディングエラーを引き起こすため、問題は深刻です。

まず `rui` と `rua` エラーを抑制し、`wua` エラーをすべて修正したあと、もう一度プログラムを実行します。次に `rui` エラーだけを抑制し、`rua` エラーをすべて修正したあと、もう一度プログラムを実行します。さらにエラーの抑制をせずに、すべての `rui` エラーを修正します。最後にプログラムを実行し、エラーがすべて修正されたことを確認してください。

最新のエラー報告を抑止するには、「`suppress -last`」を実行します。

子プロセスにおける RTC の実行

子プロセスで RTC を実行するには、`dbx` 環境変数 `rtc_inherit` を `on` に設定します。デフォルトでは `off` になります ([62 ページの「dbx 環境変数の設定」](#)を参照)。

親で RTC が有効になっていて、`dbx` 環境変数 `follow_fork_mode` が `child` に設定されているときに `dbx` を実行すると子プロセスの RTC を実行できます ([62 ページの「dbx 環境変数の設定」](#)を参照)。

分岐が発生すると、`dbx` は子に RTC を自動的に実行します。プログラムが `exec()` を呼び出すと、`exec()` を呼び出すプログラムの RTC 設定がそのプログラムに渡ります。

特定の時間に RTC の制御下におくことができるプロセスは 1 つだけです。次に例を示します。

```
% 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
program1 のシンボル情報を読んでいます
rtld /usr/lib/ld.so.1 のシンボル情報を読んでいます
librt.so のシンボル情報を読んでいます
libc.so.1 のシンボル情報を読んでいます
libdl.so.1 のシンボル情報を読んでいます
libc_psr.so.1 のシンボル情報を読んでいます
(dbx) check -all
アクセス検査 - ON
メモリー使用状況検査 - ON
(dbx) dbxenv follow_fork_mode child
(dbx) run
実行中: program1
(プロセス id 3885)
実行時検査を有効にしています...done
RTC は親プロセス、program1 の最初のエラーを報告します。
非初期化領域からの読み取り (rui):
4 バイト読み取りをアドレス 0xeffff110 でしようとした
それは 104 バイト現スタックポインタより上です
変数は 'parent_j' です。
現関数: main
    11     parent_i = parent_j;
(dbx) cont
dbx 警告: フォークしました。親プロセス内での実行時検査機能は休止します
プロセス 3885 から切り離し中
プロセス 3886 に接続しました。
follow_fork_mode が child に設定されているため、フォークが起ると、エ
ラー検査が親プロセスから子プロセスに切り替えられます。
_libc_fork で停止しました アドレス 0xef6b6040
0xef6b6040: _fork+0x0008:bgeu      _fork+0x30
現関数: main
    13     child_pid = fork();
親: child's pid = 3886
(dbx) cont
子: In child
非初期化領域からの読み取り (rui):
4 バイト読み取りをアドレス 0xeffff108 でしようとした
それは 96 バイト現スタックポインタより上です
RTC は子プロセスのエラーを報告します。
変数は 'child_j' です。
現関数: main
    22     child_i = child_j;
(dbx) cont
dbx: プロセス 3886 は exec("./program2") をするところです
dbx: プログラム "./program2" が今 exec されました
dbx: オリジナルプログラムに戻るには "debug $prog" を使用します

```

program2 のシンボル情報を読んでいます
 すでに読んでいるので、ld.so.1 を飛ばします
 すでに読んでいるので、librttc.so を飛ばします
 すでに読んでいるので、libc.so.1 を飛ばします
 すでに読んでいるので、libdl.so.1 を飛ばします
 すでに読んでいるので、libc_psr.so.1 を飛ばします
 program2 の実行が起こると、RTC 設定値は program2 から継承されるため、アクセスおよびメモリー使用状況の検査がそのプロセスに対して有効になります。

実行時検査を有効にしています...done

```
main で停止しました 行番号 8 ファイル "program2.c"
      8      printf ("program2: pid = %d\n", getpid());
```

(dbx) cont

```
program2: pid = 3886
```

非初期化領域からの読み取り (rui):

4 バイト読み取りをアドレス 0xeffff13c でしようとした

それは 100 バイト現スタックポインタより上です

RTC は、実行されたプログラム、program2 のアクセスエラーを報告します。

変数は 'program2_j' です。

現関数: main

```
      9      program2_i = program2_j;
```

(dbx) cont

メモリーリーク検査中...

RTC は、RTC 制御下にある間に終了したプロセス、program2 に関するメモリー使用状況レポートとメモリーリークレポートを出力します。

実際のリークの報告 (実際のリーク: 1 合計サイズ: 8 バイト)

合計 サイズ	ブロック 数	リーク ブロック アドレス	割り当て呼び出しスタック
8	1	0x20c50	main
起り得るリークの報告 (起り得るリーク: 0 合計サイズ: 0 バイト)			

実行完了。終了コードは 0 です。

接続されたプロセスへの RTC の使用

実行時検査は、影響を受けるメモリーがすでに割り当てられている場合に RUI が検出できなかった例外を伴う接続済みプロセスで機能します。ただし、実行時検査を開始する際、rtcaudit.so を事前に読み込んでおく必要があります。接続先のプロセスが 64 ビットプロセスである場合、64 ビットの rtcaudit.so を使用します。製品が /opt ディレクトリにインストールされている場合、rtcaudit.so は次の場所にあります。

/opt/SUNWspro/lib/v9/rtcaudit.so (64 ビット SPARC プラットフォーム)

```
/opt/SUNWspro/lib/amd64/rtcaudit.so (AMD64 プラットフォーム)
```

32 ビットプラットフォームの場合は /opt/SUNWspro/lib

rtcaudit.so を事前に読み込むには、次のように入力します。

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

rtcaudit.so を常時読み込んだ状態にせず、必要なときにだけ読み込まれるように環境変数 LD_AUDIT を設定してください。たとえば、次のようにします。

```
% setenv LD_AUDIT...
```

```
% アプリケーションの実行
```

```
% unsetenv LD_AUDIT
```

プロセスに接続したら、RTC を有効にすることができます。

接続したいプログラムがフォークされるか、または別のプログラムによって実行された場合は、LD_AUDIT をフォークを行うメインプログラムに設定する必要があります。LD_AUDIT の設定値は、フォーク先および実行主体を問わず継承されます。

環境変数 LD_AUDIT は 32 ビットプログラムと 64 ビットプログラムの両方に適用されるため、64 ビットプログラムを実行する 32 ビットプログラム用、または 32 ビットプログラムを実行する 64 ビットプログラム用に正しいライブラリを選択することが困難です。Solaris OS のバージョンによっては、環境変数 LD_AUDIT_32 をサポートしているものと環境変数 LD_AUDIT_64 をサポートしているものがあり、それぞれ 32 ビットプログラムと 64 ビットプログラムのみを対象としています。実行している Solaris OS のバージョンで、これらの変数がサポートされているかどうか確認するには、『リンカーとライブラリ』を参照してください。

RTCでの修正継続機能の使用

RTC を修正継続機能とともに使用すると、プログラミングエラーを簡単に分離して修正することができます。修正継続機能を組み合わせて使用すると、デバッグに要する時間を大幅に削減することができます。次に例を示します。

```
% 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
a.out のシンボル情報を読んでいます
rtld /usr/lib/ld.so.1 のシンボル情報を読んでいます
librtc.so のシンボル情報を読んでいます
libc.so.1 のシンボル情報を読んでいます
libintl.so.1 のシンボル情報を読んでいます
libdl.so.1 のシンボル情報を読んでいます
libw.so.1 のシンボル情報を読んでいます
(dbx) check -access
アクセス検査 - ON
(dbx) run
実行中: a.out
(プロセス id 15052)
実行時検査を有効にしています...done
非割り当て領域への書き込み (wua):
1 バイト書き込みを NULL ポインタを通してしようとしました。
現関数: problem
  7     *s = 'c';
(dbx) pop
main で停止しました 行番号 12 ファイル "bug.c"
  12     problem();
(dbx) #ここでファイルを編集します。この例では正しいバージョンをコピーしま
す。
(dbx) cp bug-fixed.c bug.c
(dbx) fix

```

```
修正中 "bug.c" .....  
pc は "bug.c":14 に移動しました  
main で停止しました 行番号 14 ファイル "bug.c"  
14     problem();  
(dbx) cont
```

実行完了。終了コードは 0 です。

```
(dbx) quit  
'a.out' の次のモジュールは変更されました (修正済み):  
bug.c  
プログラムの再構築が必要です。
```

修正と継続についての詳細は、161 ページの「メモリーリーク (mem) エラー」を参照してください。

実行時検査アプリケーションプログラミングインタフェース

リーク検出およびアクセスの両方の検査では、共有ライブラリ `libc.so` 内の標準ヒープ管理ルーチンを使用する必要があります。これは、RTC がプログラム内のすべての割り当てと解放を追跡できるためです。アプリケーションの多くは、独自のメモリー管理ルーチンを `malloc/free` にかぶせて作成したり、または最初から作成します。独自のアロケータ (専用アロケータと呼ばれる) を使用すると、RTC はそれらを自動的に追跡できません。したがって、それらの不正な使用によるリークエラーとメモリーアクセスエラーを知ることができません。

ただし、RTC には専用アロケータを使用するための API があります。この API を使用すると、専用アロケータを、標準ヒープアロケータと同様に扱うことができます。API 自体はヘッダーファイル `rtc_api.h` に入っており、Sun Studio ソフトウェアの一部として配布されます。マニュアルページの `rtc_api(3x)` には、RTC API 入口の詳細が記載されています。

専用アロケータがプログラムヒープを使用しない場合の RTC アクセスエラーレポートには小さな違いがいくつかあります。エラーレポートに、割り当て項目は含まれません。

バッチモードでの RTC の使用

`bcheck(1)` は、dbx の RTC 機能の便利なバッチインタフェースです。これは、dbx のもとでプログラムを実行し、デフォルトにより RTC エラー出力をデフォルトファイルの `program.errs` に入れます。

bcheck は、メモリーリーク検査、メモリーアクセス検査、メモリー使用状況検査のいずれか、またはこのすべてを実行できます。デフォルトでは、リーク検査だけが実行されます。この使用方法の詳細については、bcheck(1) のマニュアルページを参照してください。

注-64 ビット Linux OS を実行しているシステムで bcheck ユーティリティを実行するには、その前に環境変数 `_DBX_EXEC_32` を設定する必要があります。

bcheck 構文

bcheck の構文は次のとおりです。

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

`-o logfile` オプションを使用すると、ログファイルに別の名前を指定することができます。`-s script` オプションはプログラムの実行前に `script` を実行します。ファイル `script` に含まれる `dbx` コマンドを読み取ることができます。`script` ファイルには通常、`suppress` や `dbxenv` などのコマンドが含まれていて、bcheck によるエラー出力を調整します。

`-q` オプションは、bcheck を完全な静止状態にして、プログラムと同じ状況になります。これは、スクリプトまたはメイクファイルで bcheck を使用したい場合に便利です。

bcheck の例

hello に対してリーク検査だけを実行します。

```
bcheck hello
```

mach に引数 5 を付けてアクセス検査だけを実行します。

```
bcheck -access mach 5
```

cc に対してメモリー使用状況検査だけを静止状態で実行し、通常の終了状態で終了します。

```
bcheck -memuse -q cc -c prog.c
```

プログラムは、実行時エラーがバッチモードで検出されても停止しません。すべてのエラー出力がエラーログファイル `logfile` にリダイレクトされます。しかしプログラムは、ブレイクポイントを検出するか、またはプログラムが割り込みを受けると停止します。

バッチモードでは、完全なスタックバックトレースが生成されて、エラーログファイルにリダイレクトされます。スタックフレームの数は、`dbxenv` 変数 `stack_max_size` によって指定できます。

ファイル `logfile` がすでに存在する場合、`bcheck` はそのファイルの内容を消去してから、そこに出力をリダイレクトします。

dbx からバッチモードを直接有効化

バッチモードに似たモードを、直接 `dbx` から有効にすることもできます。具体的には、`dbx` 環境変数 `rtc_auto_continue` および `rtc_error_log_file_name` を設定します (62 ページの「`dbx` 環境変数の設定」を参照)。

`rtc_auto_continue` が `on` に設定されていると、RTC はそのままエラーを求めて自動的に実行されます。検出したエラーは、`dbx` 環境変数 `rtc_error_log_name` で指定したファイルにリダイレクトされます (62 ページの「`dbx` 環境変数の設定」を参照)。デフォルトログファイル名は、`/tmp/dbx.errlog.uniqueid` です。すべてのエラーを端末にリダイレクトするには、`rtc_error_log_file_name` 環境変数を `/dev/tty` に設定します。

`rtc_auto_continue` はデフォルト値は、`off` です。

トラブルシューティングのヒント

プログラム中でエラー検査がオンになっていて、プログラムが実行中の場合、次のエラーが検出されることがあります。

`librtc.so` と `dbx` とのバージョンが合いません。エラー検査を休止状態にしました

これは、RTC を接続されたプロセスに使用していて、`LD_AUDIT` を、各自の Sun Studio `dbx` に添付されたもの以外の `rtcaudit.so` バージョンに設定した場合に起こる可能性があります。これを修正するには、`LD_AUDIT` の設定値を変更してください。

パッチエリアが遠すぎます (8M バイトの制限); アクセス検査を休止状態にしました

RTC は、アクセス検査を有効にするためにロードオブジェクトに十分に近いパッチスペースを検出できませんでした。次の「実行時検査の制限」を参照してください。

実行時検査の制限

実行時検査には次の制限があります。

より高い効果を得るにはより多くのシンボルおよびデバッグ情報が必要になる

アクセス検査では、ロードオブジェクトにいくつかのシンボル情報が必要です。ロードオブジェクトが完全に削除されている場合、実行時検査ですべてのエラーをキャッチできないことがあります。初期化されていないメモリーからの読み取りエラーは正しくない可能性があるため、抑止されます。この抑止は `unsuppress rui` コマンドを使用して上書きできます。シンボルオブジェクトのシンボルテーブルを取得するには、ロードオブジェクトを削除する際に `-x` オプションを使用します。

x86 プラットフォームでは SIGSEGV シグナルと SIGALTSTACK シグナルが制限される

実行時検査では、メモリーアクセス命令を計測してアクセス検査をします。これらの命令は、実行時に SIGSEGV ハンドラによって処理されます。実行時検査には、独自の SIGSEGV ハンドラとシグナル代替スタックが必要なため、SIGSEGV ハンドラまたは SIGALTSTACK ハンドラをインストールしようとしても無視されるか、EINVAL エラーが生成されます。

SIGSEGV ハンドラの呼び出しは入れ子にできません。入れ子にすると、エラー `terminating signal 11 SIGSEGV` が生成されます。このエラーが表示された場合は、`rte skippatch` コマンドを使用して、影響のある関数の計測機構を飛ばします。

より高い効果を得るには、十分なパッチ領域を設け、すべての既存コードを含めて 8M バイト以内にする (SPARC プラットフォームのみ)

既存のすべてのコードを含め、8M バイト以内に十分なパッチ領域がない場合、2つの問題が発生する可能性があります。

- 遅延

アクセス検査を有効にすると、`dbx` は各ロードおよびストア命令をパッチ領域に分岐する分岐命令に置き換えます。この分岐命令の有効範囲は 8M バイトです。デバッグされたプログラムが、置き換えられた特定のロード/ストア命令の 8M バイトのアドレス空間をすべて使いきってしまった場合、パッチ領域を保存する場

所がなくなります。この場合、dbxは分岐を使用する代わりにトラップハンドラを呼び出します。トラップハンドラに制御を移行すると、実行速度が著しく(最大10倍)遅くなりますが、8Mバイトの制限に悩まされることはなくなります。

- V8+ モードでの出力レジスタの上書きの問題

トラップハンドラの制限は、次の両方の状況に該当する場合に、アクセス検査に影響します。

 - デバッグするプロセスがトラップを使用して検査される。
 - プロセスがV8+ 命令セットを使用する。

この問題は、V8+ アーキテクチャーでの出力レジスタのサイズと入力レジスタのサイズが異なるために発生します。出力レジスタは64ビット長ですが、入力レジスタは32ビット長しかありません。トラップハンドラが呼び出されると、出力レジスタが入力レジスタにコピーされ、上位32ビットが失われます。そのため、デバッグするプロセスで、出力レジスタの上位32ビットを利用する場合に、アクセス検査が有効になると、プロセスが正常に実行しない可能性があります。

32ビット SPARC ベースのバイナリの作成時に、デフォルトでコンパイラはV8+ アーキテクチャーを使用しますが、`-xarch` オプションで、V8 アーキテクチャーを使用するように指示することができます。アプリケーションを再コンパイルしてもシステム実行時ライブラリは影響を受けません。

dbxは、トラップを使用して検査すると正常に動作しない次の関数とライブラリは、計測機構を自動的に飛ばします。

- `server/libjvm.so`
- `client/libjvm.so`
- `'libfsu_isa.so'__f_cvt_real`
- `'libfsu_isa.so'__f90_slw_c4`

ただし、計測機構を飛ばすと、不正なRTCエラーが生成されることがあります。

使用しているプログラムに前述のどちらかの状況があてはまり、アクセス検査を有効にするとプログラムの動作が異なってくるようであれば、そのプログラムはトラップハンドラの制限の影響を受けている可能性があります。この制限を回避するには、次の操作を実行します。

- `rtc skippatch` コマンド (351 ページの「`rtc skippatch` コマンド」を参照) を使用して、前述の関数とライブラリを使用するプログラム内のコードの計測機構を飛ばします。一般に、特定の関数の問題を追跡するのは難しいため、ロードオブジェクト全体の計測機構を飛ばす場合は次の操作を行ってください。
- 64ビット SPARC V9の代わりに32ビット SPARC V8を使用します。

可能であれば、すべてのレジスタが64ビット長のV9アーキテクチャーでプログラムを再コンパイルします。
- パッチ領域オブジェクトファイルを追加します。

`rtc_patch_area` シェルスクリプトを使用し、大きな実行可能ファイルや共有ライブラリの中にリンクできる特別な `.o` ファイルを作成すれば、パッチ領域を拡大できます。`rtc_patch_area(1)` マニュアルページを参照してください。

`dbx` の実行時に 8M バイト制限に達すると、大きすぎる読み込みオブジェクト (メインプログラムや共有ライブラリ) が報告され、その読み込みプロジェクトに必要なパッチ領域値が出力されます。

最適な結果を得るには、実行可能ファイルや共有ライブラリ全体に特別なパッチオブジェクトファイルを均等に分散させ、デフォルトサイズ (8M バイト) かそれよりも小さいサイズを使用します。`dbx` が必要とする必要値の 10% から 20% の範囲を超えてパッチ領域を追加しないでください。たとえば、`dbx` が `.out` に 31M バイトを要求する場合は、`rtc_patch_area` スクリプトで作成したそれぞれのサイズが 8M バイトのオブジェクトファイルを 4 つ追加し、実行可能ファイル内でそれらをほぼ均等に分割します。

`dbx` の実行時に、実行可能ファイルに明示的なパッチ領域が見つかり、パッチ領域になっているアドレス範囲が出力されるので、リンク回線に正しく指定することができます。

- 読み込みオブジェクトが大きい場合は、小さい読み込みオブジェクトに分割します。

実行ファイルや大きなライブラリ内のオブジェクトファイルを小さいオブジェクトファイルグループに分割します。それらを小さいパーツにリンクします。大きいファイルが実行可能ファイルの場合、小さい実行可能ファイルと共有ライブラリに分割します。大きいファイルが共有ライブラリの場合、複数の小さいライブラリのセットに再編します。

この方法では、`dbx` により、異なる共有オブジェクト間でパッチコード用の領域を探することができます。

- パッド `.so` ファイルを追加します。

この解決方法は、プロセスの起動後に接続する場合にのみ必要です。

実行時リンカーによるライブラリの配置間隔が狭すぎてライブラリ間にパッチ領域を作成できない場合があります。RTC を `on` にして `dbx` が実行可能ファイルを起動すると、`dbx` は実行時リンカーに対して共有ライブラリ間に新たなギャップを挿入するよう指示しますが、実行時検査を有効にして `dbx` で起動されていないプロセスに接続しても、ライブラリ間が狭すぎて対応できません。

実行時ライブラリ間が狭すぎる場合 (そしてプログラムを `dbx` で起動できない場合) は、`rtc_patch_area` スクリプトで共有ライブラリを作成し、ほかの共有ライブラリ間でプログラムにリンクしてください。詳細については、`rtc_patch_area(1)` マニュアルページを参照してください。

RTCエラー

RTCで報告されるエラーは、通常はアクセスエラーとリークの2種類があります。

アクセスエラー

アクセスチェックがオンのとき、RTCによる検出と報告の対象になるのは次のタイプのエラーです。

不正解放 (baf) エラー

意味: 割り当てられたことのないメモリーを解放しようとした。

考えられる原因: `free()` または `realloc()` にヒープデータ以外のポインタを渡した。

例:

```
char a[4];
char *b = &a[0];

free(b);                /* 不正開放 (baf) */
```

重複解放 (duf) エラー

意味: すでに解放されているヒープブロックを解放しようとした。

考えられる原因: 同じポインタを使用して `free()` を2回以上呼び出した。C++ では、同じポインタに対して "delete" 演算子を2回以上使用した。

例:

```
char *a = (char *)malloc(1);
free(a);
free(a);                /* 重複開放 (duf) */
```

境界整列を誤った解放 (maf) エラー

意味: 境界合わせされていないヒープブロックを解放しようとした。

考えられる原因: `free()` または `realloc()` に正しく境界合わせされていないポインタを渡した。 `malloc` によって返されたポインタを変更した。

例:

```
char *ptr = (char *)malloc(4);
ptr++;
free(ptr);              /* 境界整列を誤った開放 (maf) */
```

境界整列を誤った読み取り (mar) エラー

意味: 適切に境界合わせされていないアドレスからデータを読み取ろうとした。

考えられる原因: ハーフワード、ワード、ダブルワードの境界に合わせられていないアドレスから、それぞれ2バイト、4バイト、8バイトを読み取った。

例:

```
char *s = "hello world";
int *i = (int *)&s[1];
int j;

j = *i;                                /* 境界整列を誤った読み取り (mar) */
```

境界整列を誤った書き込み (maw) エラー

意味: 適切に境界合わせされていないアドレスにデータを書き込もうとした。

考えられる原因: ハーフワード、ワード、ダブルワードの境界に合わせられていないアドレスに、それぞれ2バイト、4バイト、8バイトを書き込んだ。

例:

```
char *s = "hello world";
int *i = (int *)&s[1];

*i = 0;                                /* 境界整列を誤った書き込み (maw) */
```

メモリー不足 (oom) エラー

意味: 利用可能な物理メモリーより多くのメモリーを割り当てようとした。

考えられる原因: プログラムがこれ以上システムからメモリーを入手できない。oomエラーは、`malloc()`からの戻り値がNULLかどうか検査していない(プログラミングでよく起きる誤り)ために発生する問題の追跡に役立ちます。

例:

```
char *ptr = (char *)malloc(0x7fffffff);
/* メモリー不足 (oom), ptr == NULL */
```

非割り当てメモリーからの読み取り (rua) エラー

意味: 存在しないメモリー、割り当てられていないメモリー、マップされていないメモリーからデータを読み取ろうとした。

考えられる原因: ストレイポインタ(不正な値を持つポインタ)、ヒープブロック境界のオーバーフロー、すでに解放されたヒープブロックへのアクセス。

例:

```
char c, *a = (char *)malloc(1);
c = a[1]; /* 非割り当てメモリからの読み取り (rua) */
```

非初期化メモリからの読み取り (rui) エラー

意味: 初期化されていないメモリからデータを読み取ろうとした。

考えられる原因: 初期化されていない局所データまたはヒープデータの読み取り。

例:

```
foo()
{ int i, j;
  j = i; /* 非初期化メモリからの読み取り (rui) */
}
```

読み取り専用メモリへの書き込み (wro) エラー

意味: 読み取り専用メモリにデータを書き込もうとした。

考えられる原因: テキストアドレスへの書き込み、読み取り専用データセクション (.rodata) への書き込み、読み取り専用として mmap されているページへの書き込み。

例:

```
foo()
{ int *foop = (int *) foo;
  *foop = 0; /* 読み取り専用メモリへの書き込み (wro) */
}
```

非割り当てメモリへの書き込み (wua) エラー

意味: 存在しないメモリ、割り当てられていないメモリ、マップされていないメモリにデータを書き込もうとした。

考えられる原因: ストレイポインタ (不正な値を持つポインタ)、ヒープブロック境界のオーバーフロー、すでに解放されたヒープブロックへのアクセス。

例:

```
char *a = (char *)malloc(1);
a[1] = '\0'; /* 非割り当てメモリへの書き込み (wua) */
```

メモリーリークエラー

リーク検査をオンにしておくと、RTC では次のエラーが報告されます。

ブロック中のアドレス (aib)

意味: メモリーリークの可能性はある。割り当てたブロックの先頭に対する参照はないが、そのブロック内のアドレスに対する参照が少なくとも1つある。

考えられる原因: そのブロックの先頭を示す唯一のポインタが増分された。

例:

```
char *ptr;
main()
{
    ptr = (char *)malloc(4);
    ptr++; /* ブロック中のアドレス */
}
```

レジスタ中のアドレス (air)

意味: メモリーリークの可能性はある。割り当てられたブロックが解放されておらず、そのブロックに対する参照がプログラムのどこにもないが、レジスタには参照がある。

考えられる原因: コンパイラがプログラム変数をメモリーではなくレジスタにだけ保存している場合にこのエラーになる。最適化をオンにしてコンパイラを実行すると、局所変数や関数パラメータにこのような状況がよく発生する。最適化をオンにしていないのにこのエラーが発生する場合は、メモリーリークが疑われる。ブロックを解放する前に、割り当てられたブロックに対する唯一のポインタが範囲外を指定するとメモリーリークになる。

例:

```
if (i == 0) {
    char *ptr = (char *)malloc(4);
    /* ptr は範囲外になる */
}
/* レジスタ中のメモリーリーク */
```

メモリーリーク (mel) エラー

意味: 割り当てられたブロックが解放されておらず、そのブロックへの参照がプログラム内のどこにも存在しない。

考えられる原因: プログラムが使用されなくなったブロックを解放しなかった。

例:

```
char *ptr;
```

```
    ptr = (char *)malloc(1);  
    ptr = 0;  
/* メモリーリーク (me1) */
```

修正継続機能 (fix と cont)

fix を使用すると、デバッグプロセスを停止しないで、編集されたネイティブソースコードを簡単に再コンパイルすることができます。fix コマンドを使用して Java コードを再コンパイルすることはできません。

注 - fix コマンドは、Linux プラットフォームでは使用できません。

この章の内容は次のとおりです。

- 163 ページの「修正継続機能の使用」
- 165 ページの「プログラムの修正」
- 167 ページの「修正後の変数の変更」
- 168 ページの「ヘッダファイルの変更」
- 168 ページの「C++ テンプレート定義の修正」

修正継続機能の使用

fix と cont の各機能を使用すると、ソースファイルを修正して再コンパイルし、プログラム全体を作成し直すことなく実行を続けることができます。.o ファイルを更新して、それらをデバッグ中のプログラムに組み込むことにより、再リンクの必要がなくなります。

この機能を使用する利点は次のとおりです。

- プログラムをリンクし直す必要がない。
- プログラムを dbx に再読み込みする必要がない。
- 修正した位置からプログラムの実行を再開できる。

注 - 構築が進行中の場合は、`fix` コマンドを使用しないでください。

fix と cont の働き

`fix` コマンドを使用するには、エディタウィンドウでソースを編集する必要があります(コードの変更方法については、164 ページの「`fix` と `cont` によるソースの変更」を参照)。変更結果を保存して `fix` と入力します。`fix` コマンドについては、318 ページの「`fix` コマンド」を参照してください。

`fix` が実行されると、`dbx` は適切なコンパイラオプションでコンパイラを呼び出します。変更後のファイルがコンパイルされ、一時共有オブジェクト (`.so`) ファイルが作成されます。古いファイルと新しいファイルとを比較することによって、修正の安全性を検査する意味上のテストが行われます。

実行時リンカーを使用して新しいオブジェクトファイルが動作中のプロセスにリンクされ、プログラムカウンタが古い関数から新しい関数の同じ行に移動します(その関数が修正中のスタックの一番上にある場合)。さらに、古いファイルのブレイクポイントがすべて新しいファイルに移動します。

対象となるファイルがデバッグ情報付きでコンパイルされているかどうかにかかわらず、`fix` と `cont` を実行できます。ただし、デバッグ情報なしでコンパイルされているファイルの場合には多少の機能制限があります。318 ページの「`fix` コマンド」の `-g` オプションの解説を参照してください。

共有オブジェクト (`.so`) ファイルの修正は可能ですが、その場合、そのファイルを特別なモードでオープンする必要があります。`dlopen` 関数の呼び出しで、`RTLD_NOW|RTLD_GLOBAL` または `RTLD_LAZY|RTLD_GLOBAL` のどちらかを使用します。

Sun Studio C および C++ コンパイラのプリコンパイル済みヘッダー機能では、再コンパイル時にコンパイラオプションが同じである必要があります。`fix` コマンドによって、コンパイラオプションがわずかに変更されるため、プリコンパイル済みヘッダーを使用して作成されたオブジェクトファイルでは `fix` コマンドを使用しないでください。

fix と cont によるソースの変更

`fix` と `cont` を使用すると、ソースを次の方法で変更できます。

- 関数の各行を追加、削除、変更する。
- 関数を追加または削除する。
- 大域変数および静的変数を追加または削除する。

古いファイルから新しいファイルに関数をマップすると問題が起きることがあります。ソースファイルの編集時にこのような問題の発生を防ぐには、次のことを守ってください。

- 関数の名前を変更しない。
- 関数に渡す引数の型を追加、削除、または変更しない。
- スタック上で現在アクティブな関数の局所変数の型を追加、削除、または変更しない。
- テンプレートの宣言やテンプレートインスタンスを変更しない。C++ テンプレート関数定義の本体でのみ修正可能です。

前述の変更を行う場合は、`fix` と `cont` で処理するよりプログラム全体を作り直す方が簡単です。

プログラムの修正

変更後にソースファイルを再リンクするとき `fix` コマンドを使用すればプログラム全体を再コンパイルしなくて済みます。引き続きプログラムの実行を続けることができます。

▼ ファイルを修正する

- 1 変更をソースファイルに保存します。
- 2 `dbx` プロンプトで `fix` と入力します。

修正は無制限に行うことができますが、1つの行でいくつかの修正を行なった場合は、プログラムを作成し直すことを考えてください。`fix` コマンドは、メモリー内のプログラムのイメージを変更しますが、ディスク上のイメージは変更しません。また修正を行うと、メモリーのイメージは、ディスク上のイメージと同期しなくなります。

`fix` コマンドは、実行可能ファイル内での変更ではなく、`.o` ファイルとメモリーイメージの変更だけを行います。プログラムのデバッグを終了したら、プログラムを作成し直して、変更内容を実行可能ファイルにマージする必要があります。デバッグを終了すると、プログラムを作成し直すように指示するメッセージが出されます。

`-a` 以外のオプションを指定し、ファイル名引数なしで `fix` コマンドを実行すると、現在変更を行なったソースファイルだけが修正されます。

`fix` を実行すると、コンパイル時にカレントであったファイルの現在の作業ディレクトリが検索されてからコンパイル行が実行されます。したがってコンパイル時とデバッグ時とでファイルシステム構造が変化すると正しいディレクトリが見つからなくなることがあります。これを防ぐには、`pathmap` コマンドを使用します。これは1つのパス名から別のパス名までのマッピングを作成するコマンドです。マッピングはソースパスとオブジェクトファイルパスに適用されます。

修正後の続行

プログラムの実行を継続するには、`cont` コマンドを使用します (303 ページの「`cont` コマンド」を参照)。

プログラムの実行を再開するには、変更による影響を判断するための次の条件に注意してください。

実行された関数への変更

すでに実行された関数に変更を加えた場合、その変更内容は次のことが起こるまで無効です。

- プログラムが再び実行される
- その関数が次に呼び出される

変数への単純な変更以上のことを修正した場合は、`fix` コマンドに続けて `run` コマンドを使用してください。 `run` コマンドを使用すると、プログラムの再リンクが行われないため処理が速くなります。

呼び出されていない関数への変更

呼び出されていない関数に変更を加えた場合、変更内容は、その関数が呼び出されたときに有効になります。

現在実行中の関数への変更

現在実行中の関数に変更を加えた場合、`fix` コマンドの影響は、変更内容が停止した関数のどの場所に関連しているかによって異なります。

- 実行済みのコードを変更しても、そのコードは再実行されません。コードを実行するには、現在の関数をスタックからポップし (343 ページの「`pop` コマンド」を参照)、変更した関数を呼び出した位置から処理を続けます。取り消すことのできない副作用 (ファイルのオープンなど) が発生しないか、コードの内容をよく理解しておく必要があります。
- 変更内容がまだ実行されていないコードにある場合は、新しいコードが実行されます。

現在スタック上にある関数への変更

停止された関数ではなく、現在スタック上にある関数に変更を加えた場合、変更されたコードは、その関数の現在の呼び出しでは使用されません。停止した関数から戻ると、スタック上の古いバージョンの関数が実行されます。

この問題を解決する方法はいくつかあります。

- 変更したすべての関数がスタックから削除されるまで `pop` コマンドを実行する。コードを実行して問題が発生しないか確認する。

- `cont at line_number` コマンドを使用して、別の行から実行を続ける。
- データ構造を手作業で修正してから (`assign` コマンドを使用)、実行を続ける。
- `run` コマンドを使用してプログラムを再び実行する。

スタック上の修正された関数にブレークポイントがある場合、このブレークポイントは、新しいバージョンの関数に移動します。古いバージョンが実行される場合、プログラムはこれらの関数で停止しません。

修正後の変数の変更

大域変数への変更は、`pop` コマンドでも `fix` コマンドでも取り消されません。大域変数に正しい値を手作業で再び割り当てるには、`assign` コマンドを使用してください(287ページの「`assign` コマンド」を参照)。

次の例は、修正継続機能を使用して簡単なバグを修正する方法を示しています。6行目で `NULL` ポインタを逆参照しようとしたときに、セグメンテーションエラーが発生します。

```
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

実行中: testfix

(プロセス id 4842)

シグナル SEGV (フォルトのアドレスにマッピングしていません) 関数 copy 行番号 6 ファイル "testfix.c"

```
6      while ((*to++ = *from++) != '\0');
```

14行目を `0` ではなく `buf` をコピー (`copy`) するように変更し、`fix` を実行します。

```
14     copy(buf);      <=== 変更後
```

(dbx) fix

```
fixing "testfix.cc" .....
pc は "testfix.c":6 に移動しました
copy で停止しました 行番号 6 ファイル "testfix.c"
    6         while ((*to++ = *from++) != '\0')
```

ここでプログラムを続行しても、NULLポインタがスタックをプッシュしているためセグメント例外が返されます。pop コマンドを使用して、スタックフレームを1つ上がってください。

```
(dbx) pop
main で停止しました 行番号 14 ファイル "testfix.cc"
    14 copy(buf);
```

ここでプログラムを続行すると、プログラムは実行されますが、大域変数 from がすでに増分されているため正しい値が出力されません。assign コマンドを使用しないと、プログラムは ships と表示すべきところを hips と表示します。assign コマンドを使用して大域変数を復元し、次に cont コマンドを使用してください。プログラムは次のように正しい値を表示します。

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

ヘッダファイルの変更

場合によってはソースファイルだけでなくヘッダ(.h)ファイルも変更することがあります。変更したヘッダファイルをインクルードしている、プログラム内のすべてのソースファイルから、それらのヘッダファイルをアクセスするには、そのヘッダファイルをインクルードしているすべてのソースファイルのリストを引数として fix コマンドに渡す必要があります。ソースファイルのリストを指定しなければ、主要(現在の)ソースファイルだけが再コンパイルされ、変更したヘッダファイルは主要ソースファイルにしかインクルードされず、プログラムのほかのソースには変更前のヘッダファイルがインクルードされたままになります。

C++ テンプレート定義の修正

C++ テンプレート定義は直接修正できないので、これらのファイルはテンプレートインスタンスで修正します。テンプレート定義ファイルを変更しなかった場合に日付チェックを上書きするには、-f オプションを使用します。Sun Studio C コンパイラでコンパイルされたプログラムでは、dbx によるテンプレート定義 .o ファイルの検索範囲は、デフォルトのリポジトリディレクトリ SunWS_cache です。dbx の fix コマンドは -ptr コンパイラスイッチをサポートしていません。

マルチスレッドアプリケーションのデバッグ

dbx では Solaris スレッドや POSIX スレッドを使用するマルチスレッドアプリケーションをデバッグできます。dbx には、各スレッドのスタックトレースの確認、全スレッドの再実行、特定のスレッドに対する `step` や `next` の実行、スレッド間の移動をする機能があります。

dbx は、`libthread.so` が使用されているかどうかを検出することによって、マルチスレッドプログラムかどうかを認識します。プログラムは、`-lthread` または `-mt` を使用してコンパイルすることによって明示的に、あるいは `-lpthread` を使用してコンパイルすることによって暗黙的に `libthread.so` を使用します。

この章では dbx の `thread` コマンドを使用して、スレッドに関する情報を入手したり、デバッグを行う方法について説明します。

この章の内容は次のとおりです。

- 169 ページの「マルチスレッドデバッグについて」
- 174 ページの「スレッド作成動作について」
- 175 ページの「LWP 情報について」

マルチスレッドデバッグについて

dbx は、マルチスレッドプログラムを検出すると、`libthread_db.so` の `dlopen` を試行します。これは、`/usr/lib` にあるスレッドデバッグ用の特別なシステムライブラリです。

dbx は同期的に動作します。つまり、スレッドか軽量プロセス (LWP) のいずれかが停止すると、ほかのスレッドおよび LWP もすべて自動的に停止します。この動作は、「世界停止 (stop the world)」モデルと呼ばれる場合があります。

注-マルチスレッドプログラミングとLWPについては、『Solaris マルチスレッドのプログラミング』を参照してください。

スレッド情報

dbx では、次のスレッド情報を入手できます。

```
(dbx) threads
t@1 a l@1 ?() 実行中 : 現在の関数 main()
t@2      ?() 0xef751450 でスリープ : 現在の関数 in_swch()
t@3 b l@2 ?() 実行中 : 現在の関数 sigwait()
t@4      consumer() 0x22bb0 でスリープ : 現在の関数 in_lwp_sema_wait()
*>t@5 b l@4 consumer() ブレークポイント : 現在の関数 Queue_dequeue()
t@6 b l@5 producer()      実行中 : 現在の関数 in_thread_start()
(dbx)
```

ネイティブコードに対して、情報の各行の内容は次のとおりです。

- *(アスタリスク)は、ユーザーの注意を必要とするイベントがこのスレッドで発生したことを示します。通常は、ブレークポイントに付けられます。アスタリスクの代わりに'o'が示される場合は、dbx 内部イベントが発生していません。
- >(矢印)は現在のスレッドを示します。
- t@number はスレッド ID であり、特定のスレッドを指します。number は、thr_create が返す thread_t の値になります。
- b l@number はそのスレッドが指定の LWP に結合されていることを表し、a l@number はそのスレッドがアクティブであることを表します。すなわちそのスレッドはオペレーティングシステムにて実行可能です。
- thr_create に渡されたスレッドの開始関数。?() は開始関数が不明であることを示します。
- スレッド状態(スレッド状態の詳細については、表 11-1 を参照)
- スレッドが現在実行している関数

Java コードでは、情報の各行は次で構成されています。

- t@number は dbx スタイルスレッド ID を示します。
- スレッド状態(スレッド状態の詳細については、表 11-1 を参照)
- 単一引用符内のスレッド名
- スレッドの優先順位を示す番号

表 11-1 スレッドの状態とLWPの状態

スレッドの状態とLWPの状態	内容の説明
中断	スレッドは明示的に中断されています。
実行可能	スレッドは実行可能であり、コンピューティング可能なリソースとしてLWPを待機しています。
ゾンビ	結合されていないスレッド (<code>thr_exit</code>) がある場合、 <code>thr_join()</code> で再結合するまでゾンビ状態になります。THR_DETACHED は、スレッド作成時に指定するフラグです (<code>thr_create()</code>)。非結合のスレッドは、再実行されるまでゾンビ状態です。
<code>syncobj</code> でスリープ中	スレッドは所定の同期オブジェクトでブロックされています。 <code>libthread</code> と <code>libthread_db</code> によるサポートレベルにより、 <code>syncobj</code> が伝える情報は単純な 16 進アドレスになったり、より詳細な内容になります。
アクティブ	LWP でスレッドがアクティブですが、 <code>dbx</code> は LWP をアクセスできません。
未知	<code>dbx</code> では状態を判定できません。
<code>lwpstate</code>	結合スレッドやアクティブスレッドの状態に、LWP の状態が関連付けられています。
実行中	LWP が実行中でしたが、ほかの LWP と同期して停止しました。
システムコール <code>num</code>	所定のシステムコール番号の入口で LWP が停止しました。
システムコール <code>num</code> 戻り	所定のシステムコール番号の出口で LWP が停止しました。
ジョブコントロール	ジョブコントロールにより、LWP が停止しました。
LWP 中断	LWP がカーネルでブロックされています。
シングル中断	LWP により、1 ステップが終了しました。
ブレークポイント	LWP がブレークポイントに達しました。
障害 <code>num</code>	LWP に所定の障害番号が発生しました。
シグナル <code>name</code>	LWP に所定のシグナルが発生しました。
プロセス <code>sync</code>	この LWP が所属するプロセスの実行が開始しました。
LWP 終了	LWP は終了プロセス中です。

別のスレッドのコンテキストの表示

表示コンテキストを別のスレッドに切り替えるには、`thread` コマンドを使用します。この構文は次のとおりです。

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

現在のスレッドを表示するには、次のように入力します。

```
thread
```

スレッド *thread_id* に切り替えるには、次のように入力します。

```
thread thread_id
```

`thread` コマンドの詳細については、[369 ページ](#)の「`thread` コマンド」を参照してください。

スレッドリストの表示

スレッドリストを表示するには、`threads` コマンドを使用します。この構文は次のとおりです。

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

既知のスレッドすべてのリストを表示するには、次のように入力します。

```
threads
```

通常は表示されないスレッド(ゾンビ)などを表示するには、次のように入力します。

```
threads -all
```

スレッドリストについては、[170 ページ](#)の「スレッド情報」を参照してください。

`threads` コマンドの詳細については、[371 ページ](#)の「`threads` コマンド」を参照してください。

実行の再開

プログラムの実行を再開するには、`cont` コマンドを使用します。プログラム現在、スレッドは同期ブレークポイントを使用して、すべてのスレッドが実行を再開するようにしています。

ただし、シングルスレッドは、`call` コマンドに `-resumeone` オプションを付けて使用することにより再開できます([289 ページ](#)の「`call` コマンド」を参照)。

多数のスレッドが関数 `lookup()` を呼び出すマルチスレッドアプリケーションをデバッグする場合の2つのシナリオを次に示します。

- 条件付きブレークポイントを設定します。

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

t@1 が lookup() の呼び出しで停止すると、dbx は条件の評価を試み、strcmp() を呼び出します。

- ブレークポイントを設定します。

```
stop in lookup
```

t@1 が lookup() の呼び出しで停止したら、次のコマンドを発行します。

```
call strcmp(name, "troublesome")
```

strcmp() を呼び出すと、dbx は呼び出しの間、すべてのスレッドを再開することがありますが、これは next コマンドを使用して、シングルステップ実行をする場合の dbx の動作に似ています。この動作は、t@1 のみを再開すると、strcmp() が別のスレッドによって所有されているロックを奪取しようと試みた場合に、デッドロックが発生する可能性があるためです。

この場合にすべてのスレッドを再開することの欠点は、strcmp() の呼び出し中に lookup() のブレークポイントにヒットして、dbx が t@2 などのほかのスレッドを処理できないことです。次のような警告が表示されます。

イベント無限ループにより次のハンドラ中でイベントの取りこぼしが起きます。

Event reentrancy

first event BPT(VID 6, TID 6, PC echo+0x8)

second event BPT(VID 10, TID 10, PC echo+0x8)

the following handlers will miss events:

そのような場合は、条件式で呼び出された関数が相互排他ロックを奪取しないことが確実であれば、-resumeone イベント修飾子を使用して、dbx に t@1 のみを再開させることができます。

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

strcmp() を評価するために、lookup() のブレークポイントをヒットしたスレッドのみが再開されます。

この方法は、次のような状況では無効です。

- 条件で再帰的に lookup() を呼び出すため、同じスレッドで lookup() の 2 つ目のブレークポイントが発生した場合
- 条件を実行するスレッドが生成するか、スリープさせるか、または何らかの方法で、別のスレッドに制御を放棄する場合

スレッド作成動作について

次の例に示すように、アプリケーションが `thr_create` イベントおよび `thr_exit` イベントを使用して、どれくらい頻繁にスレッドを作成および終了しているかを知ることができます。

```
(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
```

ここでは、アプリケーションが3つのスレッドを作成します。スレッドは作成されたのとは逆の順序で終了し、アプリケーションにそれ以上のスレッドがある場合は、スレッドが累積されてリソースを消費します。

有用な情報を得るため、別のセッションで次のコマンドを実行してみてください。

```
(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
```

出力を見ると、3つのスレッドすべてがスレッド `t@1` によって作成されていることがわかります。これは、一般的なマルチスレッド化のパターンです。

スレッド `t@3` を、その出力セットからデバッグする場合を考えます。次のようにすると、スレッド `t@3` が作成されたポイントでアプリケーションを停止できます。

```
(dbx) stop thr_create t@3
(dbx) run
t@1 (l@1) stopped in tdb_event_create at 0xff38409c
0xff38409c: tdb_event_create      :   retl
現関数: main
216      stat = (int) thr_create(NULL, 0, consumer, q, tflags, &tid_cons2);
(dbx)
```

アプリケーションで新しいスレッドが発生しますが、それがスレッド `t@1` ではなくスレッド `t@5` から発生することがある場合は、次のようにするとそのイベントを獲得できます。

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

LWP 情報について

通常は LWP を意識する必要はありません。ただし、スレッドレベルでの問い合わせが完全にできない場合には、`lwps` コマンドを使用して、LWP に関する情報を入手できます。

```
(dbx) lwps
  l@1 実行中 : 現在の関数 main()
  l@2 実行中 : 現在の関数 sigwait()
  l@3 実行中 : 現在の関数 _lwp_sema_wait()
 *>l@4 ブレークポイント : 現在の関数 Queue_dequeue()
  l@5 実行中 : 現在の関数 _thread_start()
(dbx)
```

LWP リストの各行の内容は、次のとおりです。

- `*` (アスタリスク) は、ユーザーの注意を要するイベントがこの LWP で起こったことを示します。
- 矢印は現在の LWP を表します。
- `l@number` は特定の LWP を示します。
- 次の項目で詳しい LWP の状態を説明しています。
- `function_name()` は、LWP が現在実行している関数を示します。

現在の LWP を表示または変更するには、[335 ページの「lwp コマンド」](#) を使用してください。

◆◆◆ 第 12 章

子プロセスのデバッグ

この章では、子プロセスのデバッグ方法について説明します。dbx は、fork(2) および exec(2) を介して子を作成するプロセスのデバッグに役立つ機能をいくつか備えています。

この章の内容は次のとおりです。

- 177 ページの「単純な接続の方法」
- 178 ページの「exec 機能後のプロセス追跡」
- 178 ページの「fork 機能後のプロセス追跡」
- 178 ページの「イベントとの対話」

単純な接続の方法

子プロセスがすでに作成されている場合は、次のいずれかの方法でそのプロセスに接続できます。

- dbx 起動時、シェルから次のように入力します。

```
$ dbx program_name process_id
```

- コマンド行からは次のように入力します。

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

どちらの場合も *program_name* を "-" に置き換えることができます。そうすると dbx は指定されたプロセス ID (*process_id*) に対応する実行可能ファイルを自動的に見つけ出します。 "-" を使用すると、それ以後 run コマンドおよび rerun コマンドは機能しません。これは、dbx が実行可能ファイルの絶対パス名を知らないためです。

Sun Studio IDE の「デバッガ」ウィンドウからは、実行中の子プロセスにも結合できます (IDE オンラインヘルプの「実行中のプロセスへデバッガの接続」を参照してください)。

exec 機能後のプロセス追跡

子プロセスが新しいプログラムを `exec(2)` 関数を用いて実行すると、そのプロセス ID は変わりませんが、プロセスイメージは変化します。dbx は `exec()` の呼び出しを自動的に検知し、新しく実行されたプログラムを自動的に再読み込みします。

実行可能ファイルの元の名前は、`$oprog` に保存されます。この名前に復帰するには、`debug $oprog` を使用します。

fork 機能後のプロセス追跡

子プロセスが、関数 `vfork()`、`fork(1)`、または `fork(2)` を呼び出すと、プロセス ID が変化しますが、プロセスイメージは変化しません。dbx 環境変数 `follow_fork_mode` の設定値に従って、dbx は次のように動作します。

parent (親プロセス)	従来の動作です。dbx は <code>fork</code> を無視し、親プロセスを追跡します。
child (子プロセス)	dbx は、新しいプロセス ID で、分岐先の子に自動的に切り替わります。元の親のすべての接続と認識が失われています。
both (両方)	このモードは、Sun Studio IDE から dbx を使用する場合は利用できません。
ask (質問)	dbx が <code>fork</code> を検出するたびにプロンプトが表示され、 <code>parent</code> 、 <code>child</code> 、 <code>both</code> のどのモードを使用するか問い合わせできます。 <code>stop</code> を選択すると、プログラムの状態を調べてから、 <code>cont</code> を使用して実行を続けることができます。プロンプトに従って次の処理を選択します。

イベントとの対話

`exec()` 関数や `fork()` 関数では、ブレークポイントやほかのイベントが、すべて削除されます。しかし、dbx 環境変数で `follow_fork_inherit` を on に設定するか、`-perm eventspec` 修飾子でイベントを持続イベントにすれば、ブレークポイントやほかのイベントは削除されません。イベント仕様修飾子の使用方法の詳細については、[262 ページの「cont at コマンド」](#)を参照してください。

OpenMP プログラムのデバッグ

OpenMP™ アプリケーションプログラミングインタフェース (API) は、共用メモリーマルチプロセッサアーキテクチャー用に複数のコンピュータベンダーと共同で開発された並列プログラミングモデルです。Fortran、C++ および C の OpenMP プログラムを dbx を使用してデバッグするためのサポートは、dbx の汎用マルチスレッドデバッグ機能に基づいています。スレッドおよび LWP 上で動作するすべての dbx コマンドは OpenMP デバッグに使用できます。dbx は、OpenMP デバッグでの非同期スレッド制御はサポートしていません。

この章の内容は次のとおりです。

- 180 ページの「コンパイラによる OpenMP コードの変換」
- 181 ページの「OpenMP コードで利用可能な dbx の機能」
- 181 ページの「OpenMP コードにおけるスタックトレースの使用」
- 182 ページの「OpenMP コードにおける dump コマンドの使用」
- 182 ページの「OpenMP コードの実行シーケンス」

Sun Studio Fortran 95 および C コンパイラによって実装される指示、実行時ライブラリルーチン、および OpenMP Version 2.0 アプリケーションプログラムインタフェースの環境変数については、『OpenMP API ユーザーズガイド』を参照してください。

注 - OpenMP デバッグが行えるのは Solaris OS の動作しているプラットフォームのみです。Linux プラットフォームで行うことはできません。

コンパイラによる OpenMP コードの変換

OpenMP デバッグの詳細については、OpenMP コードがコンパイラによってどのように変換されるかを理解することが役立ちます。次に Fortran の例を示します。

```
1   プログラムの例
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  プログラムの例、終わり
```

行 12～18 のコードは並列領域です。f95 コンパイラは、コードのこのセクションを、OpenMP 実行時ライブラリから呼び出されるアウトラインサブルーチンに変換します。このアウトラインサブルーチンには、内部で生成された名前が付きます。この場合は `_sd1A12.MAIN_` です。次に f95 コンパイラは、OpenMP 実行時ライブラリへの呼び出しによって並列領域用にコードを置換して、アウトラインサブルーチンを引数の 1 つとして渡します。OpenMP 実行時ライブラリはすべてのスレッド関連実行を処理し、アウトラインサブルーチンを並列で実行するスレーブスレッドをディスパッチします。C コンパイラも同様に動作します。

OpenMP プログラムをデバッグするときには、アウトラインサブルーチンは dbx によって別の関数として扱われますが、内部生成された名前を使用して関数内のブレークポイントを明示的に設定することはできません。

OpenMP コードで利用可能な dbx の機能

マルチスレッドプログラムのデバッグ用の機能に加えて、OpenMP プログラム内で dbx を使用して次のことが実行できます。

- 並列領域へのシングルステップ。並列領域は OpenMP 実行時ライブラリから呼び出されるため、実行のシングルステップは実際には、この目的のために作成されたスレーブスレッドによって実行される複数の実行ライブラリ呼び出しレイヤーがかかわってきます。並列領域にシングルステップ実行すると、最初にブレークポイントに到達したスレッドによってプログラムが停止します。このスレッドは、ステップを開始したマスターステップではなく、スレーブスレッドになります。

たとえば、180 ページの「コンパイラによる OpenMP コードの変換」の Fortran を参照して、マスタースレッド t@1 が行 10 にあるとします。行 12 にシングルステップすると、実行時ライブラリ呼び出しを実行するためのスレーブスレッド t@2、t@3、および t@4 が作成されます。スレッド t@3 が最初にブレークポイントに到達し、プログラムの実行が停止します。したがって、スレッド t@1 によって開始されたシングルステップはスレッド t@3 で終了します。この動作は、シングルステップのあとに同じスレッドで行う通常のステップ実行とは異なります。

- *shared*、*private*、および *threadprivate* 変数の出力。dbx はすべての *shared*、*private*、および *threadprivate* 変数を出力できます。並列領域外で *threadprivate* 変数を出力しようとする、マスタースレッドのコピーが出力されます。whatis コマンドは変数が *shared*、*private*、または *threadprivate* のいずれであるかを通知しません。

OpenMP コードにおけるスタックトレースの使用

並列領域で実行が停止されると、where コマンドによってアウトラインサブルーチンを含むスタックトレースと複数の実行時ライブラリ呼び出しが表示されます。180 ページの「コンパイラによる OpenMP コードの変換」の Fortran の例を使用して実行を行 15 で停止すると、where コマンドによって次のスタックトレースが生成されます。

```
[t@4 l@4]: where
現スレッド: t@4
=>[1] _$d1A12.MAIN_(), 行番号 15 "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
```

スタックの上位フレームはアウトライン関数のフレームです。コードがアウトラインされているにもかかわらず、ソース行番号は 15 にマップされたままです。ほかの 2 つのフレームは実行時ライブラリルーチン用です。

並列領域で実行が停止されると、前述の例のようにスレーブスレッドの where コマンドはスタックトレースを親スレッドに戻しません。ただし、マスタースレッドの where コマンドは完全トレースバックを行います。

```
[t@4 l@4]: thread t@1
t@1 (l@1) で停止しました _$d1A12.MAIN_ 行番号 15 ファイル "example.f90"
15      sum = sum + a(i)
[t@1 l@1]: where
現スレッド: t@1
=>[1] _$d1A12.MAIN_(), 行番号 15 "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(), 行番号 12 "example.f90"
```

いくつかのスレッドが大きくない場合、threads コマンド (371 ページの「threads コマンド」参照) を使用してすべてのスレッドをリスト表示し、スレーブスレッド内で実行がどのようにブレークポイントに到達したかを判別し、各スレッドに切り替えてマスタースレッドを判別することができます。

OpenMP コードにおける dump コマンドの使用

並列領域で実行が停止すると、dump コマンドによって private 変数の複数のコピーが出力されます。次の例では、dump c コマンドが変数 i の 2 つのコピーを出力します。

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

変数 i の 2 つのコピーが出力されるのは、アウトラインルーチンがホストルーチンのネストされた関数として実装され、private 変数がアウトラインルーチンの局所変数として実装されます。dump コマンドがスコープ内のすべての変数を出力するため、ホストルーチン内の i およびアウトラインルーチン内の i の両方が表示されます。

OpenMP コードの実行シーケンス

OpenMP プログラム内の並列領域の内部にシングルステップするときの実行シーケンスは、ソースコードシーケンスとは同じではありません。シーケンスが異なるのは、並列領域内のコードが通常はコンパイラによって変換され再配置されるためです。OpenMP コード内でのシングルステップは、オプティマイザがコードを移動する最適化コード内でのシングルステップと似ています。

シグナルの処理

この章では、`dbx` を使用してシグナルを処理する方法を説明します。`dbx` は、`catch` というブレークポイントコマンドをサポートします。`catch` コマンドは、`catch` リストに登録されているシステムシグナルのいずれかが検出された場合にプログラムを停止するよう `dbx` に指示します。

また、`dbx` コマンド `cont`、`step`、`next` は、オプション `-sig signal_name` をサポートします。このオプションを使用すると、実行を再開したプログラムに対し、`cont -sig` コマンドで指定したシグナルを受信した場合の動作をさせることができます。

この章は次の各節から構成されています。

- 183 ページの「シグナルイベントについて」
- 184 ページの「システムシグナルを捕獲する」
- 187 ページの「プログラムにシグナルを送信する」
- 187 ページの「シグナルの自動処理」

シグナルイベントについて

デバッグ中のプロセスにシグナルが送信されると、そのシグナルはカーネルによって `dbx` に送られます。通常、このことはプロンプトによって示されますが、ここでは次の2つの操作から1つを選択してください。

- プログラムを再開するときにそのシグナルを「取り消し」ます。これは、`cont` コマンドのデフォルトの動作です。これにより、次の図 14-1 のような `SIGINT` (Control-C) を使用した割り込みと再開が容易になります。
- 次のコマンドを使用して、シグナルをプロセスに「転送」します。

```
cont -sig signal
```

`signal` は、シグナル名またはシグナル番号です。

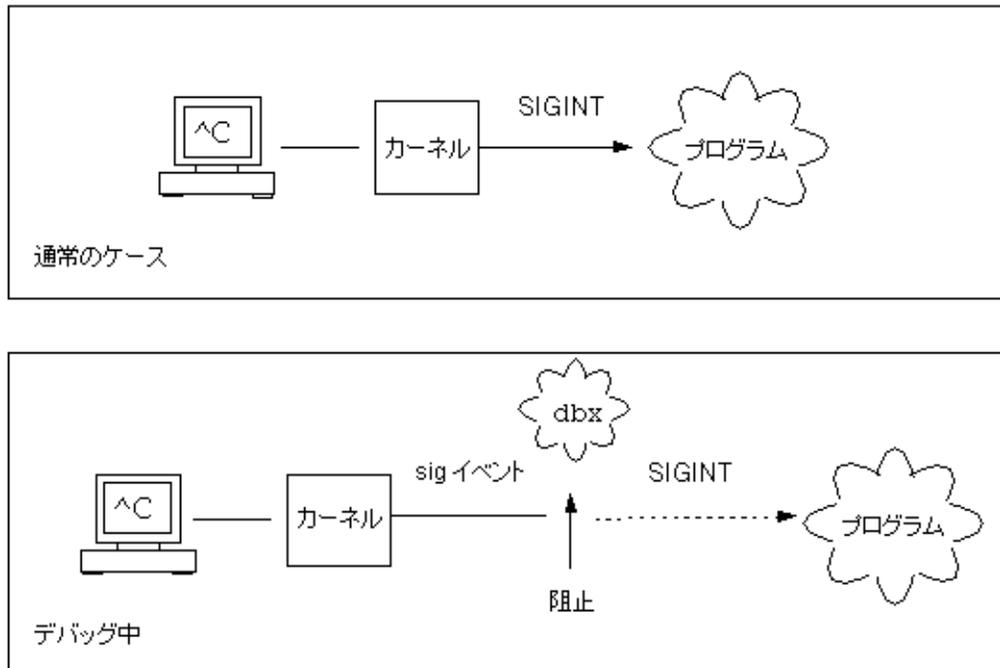


図 14-1 SIGINT シグナルの阻止と取り消し

さらに、特定のシグナルを頻繁に受信する場合、そのシグナルを表示せずに受信したシグナルを dbx が自動的に転送するように設定できます。次のように入力します。

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

前述の操作をしてもシグナルはプロセスに送信されます。シグナルがデフォルト設定で、このように自動送信されるようになっているからです (324 ページの「[ignore コマンド](#)」を参照)。

システムシグナルを捕獲する

デフォルトのシグナル捕獲リスト (catch リスト) には、33 種類の検出可能なシグナルのうち 22 種類が含まれています (これらの数はオペレーティングシステムとそのバージョンによって異なります)。デフォルトの catch リストは、リストにシグナルを追加したり削除したりすることによって変更できます。

注 - `dbx` が受け付けるシグナル名のリストは、`dbx` がサポートするバージョンの Solaris オペレーティング環境によってサポートされているすべてを含みます。したがって、`dbx` は、ユーザーが実行している Solaris オペレーティング環境のバージョンでサポートされていないシグナルを受け付ける場合があります。たとえば、`dbx` は、ユーザーが Solaris 7 OS を実行していても、Solaris 9 OS によってサポートされているシグナルを受け付けます。実行している Solaris OS でサポートされているシグナルのリストについては、`signal(3head)` マニュアルページを参照してください。

現在捕獲されているシグナルのリストを調べるには、シグナルの引数を指定せずに、次のように入力します。

```
(dbx) catch
```

プログラムで検出された場合でも、現在無視されているシグナルのリスト (`ignore` リスト) を調べるには、シグナル名の引数を指定せずに、次のように入力します。

```
(dbx) ignore
```

デフォルトの **catch** リストと **ignore** リストを変更する

どのシグナルでプログラムを停止するかは、2つのリストの間でシグナル名を移動することによって制御します。シグナル名を移動するには、一方のリストに現在表示されているシグナル名を、もう一方のリストに引数として渡します。

たとえば、`QUIT` シグナルと `ABRT` シグナルを `catch` リストから `ignore` リストに移動するには、次のように入力します。

```
(dbx) ignore QUIT ABRT
```

FPE シグナルをトラップする (Solaris プラットフォームのみ)

浮動小数点の計算が必要なコードを扱っている場合には、プログラム内で発生した例外をデバッグしなければならないことがよくあります。オーバーフローやゼロ除算などの浮動小数点例外が発生すると、例外を起こした演算の結果としてシステムが「適正な」答えを返します。適正な答えが返されることで、プログラムは正常に実行を続けることができます (Solaris OS は、IEEE 標準のバイナリ浮動小数点演算定義の、例外に対する「適正 (reasonable) な」答えを実装しています)。

浮動小数点例外に対して適正な答えを返すため、例外によって自動的に SIGFPE シグナルが生成されることはありません。例外の場合 (ゼロで整数を割ると整数がオーバーフローする場合など) は、デフォルトでは SIGFPE シグナルをトリガーします。

例外の原因を見つけ出すためには、例外によって SIGFPE シグナルが生成されるように、トラップハンドラをプログラム内で設定する必要があります (トラップハンドラの例については、`ieee_handler(3m)` コマンドのマニュアルページを参照)。

トラップを有効にするには、次のコマンド等を利用します。

- `ieee_handler`
- `fdsetmask(fdsetmask(3c))` マニュアルページ参照)
- `-ftrap` コンパイラフラグ (Fortran 95 については、マニュアルページ `f95(1)` を参照)

`ieee_handler` コマンドを使用してトラップハンドラを設定すると、ハードウェア浮動小数点状態レジスタ内のトラップ許可マスクがセットされます。このトラップ許可マスクにより、実行中に例外が発生すると SIGFPE シグナルが生成されます。

トラップハンドラ付きのプログラムをコンパイルしたあと、そのプログラムを `dbx` に読み込んでください。ここで、SIGFPE シグナルが捕獲されるようにするには、`dbx` のシグナル捕獲リスト (catch リスト) に FPE を追加する必要があります。

```
(dbx)
catch FPE
```

FPE はデフォルトでは `ignore` リストに含まれています。

例外の発生場所の判定

FPE を catch リストに追加後、`dbx` でプログラムを実行します。トラップしている例外が発生すると SIGFPE シグナルが生成され、`dbx` はプログラムを停止します。ここで、呼び出しスタックを (`dbx` コマンド `where` を使用して) トレースすることにより、プログラムの何行目で例外が発生したかを調べることができます ([386 ページ](#) の「`where` コマンド」参照)。

例外処理の原因追求

例外処理の原因を調べるには、`regs -f` コマンドを実行して浮動小数点状態レジスタ (FSR) を表示します。このレジスタで、発生した例外処理 (`aexc`) フィールドと現在の例外処理 (`cexc`) フィールドの内容を確認します。このフィールドには次のような浮動小数点例外条件が格納されています。

- 無効なオペランド
- オーバーフロー
- アンダーフロー
- ゼロによる除算

- 不正確な結果

浮動小数点状態レジスタの詳細については、『SPARCアーキテクチャマニュアルバージョン8』(V9の場合はバージョン9)を参照してください。説明と例については、『数値演算ガイド』を参照してください。

プログラムにシグナルを送信する

dbx コマンド `cont` は、オプション `-sig signal` をサポートします。このオプションを使用すると、実行を再開したプログラムに対し、指定したシステムシグナル `signal` を受信した場合の動作をさせることができます。

たとえば、プログラムに `SIGINT (^C)` の割り込みハンドラが含まれている場合、`^c` を入力することによって、アプリケーションを停止し、`dbx` に制御を返すことができます。ここで、プログラムの実行を継続するときにオプションなしの `cont` コマンドを使用すると、割り込みハンドラは実行されません。割り込みハンドラを実行するためには、プログラムに `SIGINT` シグナルを送信する必要があります。次のコマンドを使用します。

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

`step`、`next`、`detach` コマンドも、`-sig` オプションを指定できます。

シグナルの自動処理

イベント管理コマンドでは、シグナルをイベントとして処理することもできます。次の2つのコマンドの結果は同じになります。

```
(dbx) stop sig signal  
(dbx) catch signal
```

プログラミング済みのアクションを関連付ける必要がある場合、シグナルイベントがあると便利です。

```
(dbx) when sig SIGCLD {echo Got $sig $signature;}
```

この場合は、まず `SIGCLD` を `ignore` リストに必ず移動してください。

```
(dbx) ignore SIGCLD
```


dbx を使用してプログラムをデバッグする

この章では、dbx による C++ の例外の処理方法と C++ テンプレートのデバッグについて説明します。これらの作業を実行するために使用するコマンドの要約とコード例も示します。

この章の内容は次のとおりです。

- 189 ページの「C++ での dbx の使用」
- 190 ページの「dbx での例外処理」
- 194 ページの「C++ テンプレートでのデバッグ」

C++ プログラムのコンパイルの詳細については、51 ページの「デバッグのためのプログラムのコンパイル」を参照してください。

C++ での dbx の使用

この章では C++ デバッグの 2 つの特殊な点を中心に説明しますが、dbx を使用すると、C++ プログラムのデバッグに次の機能を利用することができます。

- クラスと型定義の検索 (81 ページの「型およびクラスの定義を調べる」参照)
- 継承されたデータメンバーの出力または表示 (121 ページの「C++ での表示」参照)
- オブジェクトポインタに関する動的情報の検索 (121 ページの「C++ での表示」参照)
- 仮想関数のデバッグ (92 ページの「関数を呼び出す」参照)
- 実行時型情報の使用 (118 ページの「変数、式または識別子の値を出力する」参照)
- クラスのすべてのメンバー関数に対するブレークポイントの設定 (99 ページの「同じクラスのメンバー関数にブレークポイントを設定する」参照)
- 多重定義されたすべてのメンバー関数に対するブレークポイントの設定 (98 ページの「異なるクラスのメンバー関数にブレークポイントを設定する」参照)
- 多重定義されたすべての非メンバー関数に対するブレークポイントの設定 (99 ページの「非メンバー関数に複数のブレークポイントを設定する」参照)

- 特定オブジェクトのすべてのメンバー関数に対するブレークポイントの設定 (99 ページの「オブジェクトにブレークポイントを設定する」参照)
- 多重定義された関数またはデータメンバーの処理 (97 ページの「関数に stop ブレークポイントを設定する」参照)

dbxでの例外処理

プログラムは例外が発生すると実行を停止します。例外は、ゼロによる除算や配列のオーバーフローといったプログラムの障害を知らせるものです。ブロックを設定して、コードのどこかほかの場所で起こった式による例外を捕獲できます。

プログラムのデバッグ中、dbxを使用すると次のことが可能になります。

- スタックを解放する前に処理されていない例外を捕獲する
- 予期できない例外を捕獲する
- スタックを解放する前に、特定の例外が処理されたかどうかに関係なく捕獲する
- 特定の例外がプログラム内の特定の位置で起こった場合、それが捕獲される場所を決める

例外処理の発生箇所では `step` コマンドを実行すると、スタックの開放時に実行された最初のデストラクタの先頭に制御が戻ります。step を実行して、スタックの解放時に実行されたデストラクタを終了すると、制御は次のデストラクタの先頭に移ります。こうしてすべてのデストラクタが終了したあとに `step` コマンドを実行すると、例外処理の原因を扱う捕獲ブロックに制御が移ります。

例外処理コマンド

exception [-d | +d] コマンド

exception コマンドでは、デバッグ時にいつでも例外処理の型を確認できます。オプションなしで exception コマンドを実行するときに表示される型は、dbx 環境変数 `output_dynamic_type` の設定で制御できます。

- この変数を on に設定すると、派生型が表示されます。
- この変数を off (デフォルト) に設定すると、静的な型が表示されます。

-d オプションや +d オプションを指定すると、環境変数の設定が無効になります。

- -d を設定すると、派生型が表示されます。
- +d を設定すると、静的な型が表示されます。

詳細については、316 ページの「exception コマンド」を参照してください。

intercept [-all] [-x] [-set] [typename] コマンド

スタックを解放する前に、特定の型の例外を阻止または捕獲できます。intercept コマンドを引数を付けずに使用すると、阻止される型がリストで示されます。-all を使用すると、すべての例外が阻止されます。阻止リストに型を追加するには *typename* を使用します。-x を使用すると、特定の型を除外リストに格納し、阻止から除外することができます。-set を使用すると、阻止リストと除外リストの両方をクリアし、リストを指定した型のみをスローするインターセプトまたは除外に設定できます。

たとえば、int を除くすべての型を阻止するには、次のように入力します。

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

Error 型の例外を阻止するには、次のように入力します。

```
(dbx) intercept Error
```

CommonError 例外の阻止が多すぎた場合は、次のように入力してその除外を実行することができます。

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

intercept コマンド引数なしで入力すると、処理されていない例外および予期できない例外を含んだ阻止リストが表示されます。これらの例外はデフォルトで阻止され、それに加えてクラス CommonError を除くクラス Error の例外が阻止されます。

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

Error が例外クラスのものではなく、探している例外クラスの名前が分からない場合は、次のように入力すると、クラス Error 以外のすべての例外を阻止できます。

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

詳細については、[325 ページの「intercept コマンド」](#)を参照してください。

unintercept [-all] [-x] [typename] コマンド

unintercept コマンドは、阻止リストまたは除外リストから例外の型を削除するために使用します。引数を付けずにこのコマンドを使用すると、阻止されている型のリストが示されます (intercept コマンドに同じ)。-all を使用すると、阻止リストからすべての型を削除することができます。typename を使用すると、阻止リストから 1 つの型を削除することができます。-x を使用すると、除外リストから 1 つの型を削除することができます。

詳細については、[379 ページの「unintercept コマンド」](#)を参照してください。

whocatches *typename* コマンド

whocatches コマンドは、*typename* の例外が実行の現時点で送出された場合に、どこで捕獲されるかを報告するものです。このコマンドは、例外がスタックのトップフレームから送出された場合に何が起こるかを検出する場合に使用します。

typename を捕獲した元の送出の行番号、関数名、およびフレーム数が表示されません。捕獲ポイントがスルーを行なっている関数と同じ関数内にあると、このコマンドは、「型にはハンドルがありません」というメッセージを表示します。

詳細については、[389 ページの「whocatches コマンド」](#)を参照してください。

例外処理の例

次の例は、例外を含むサンプルプログラムを使用して、dbxで例外処理がどのように実行されるかを示しています。型 `int` の例外が、関数 `bar` で送出されて、次の捕獲ブロックで捕獲されています。

```
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 }
```

サンプルプログラムからの次のトランスクリプトは、dbxの例外処理機能を示しています。

```

(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
(プロセス id 304)
bar で停止しました 行番号 13 ファイル "foo.cc"
    13      c c1(3);
(dbx) whocatches int
int が行番号 24 で捕獲されました、関数 main (フレーム番号 2)
(dbx) whocatches c
dbx: class c の実行時型情報がありません (送出手も捕獲もされていない)
(dbx) cont
例外の型 int が行番号 24 で捕獲されました、関数 main (フレーム番号 4)
_exdbg_notify_of_throw で停止しました アドレス 0xef731494
0xef731494: _exdbg_notify_of_throw      :      jmp      %o7 + 0x8
現関数 : bar
    14      throw(99);
(dbx) step
c::~c で停止しました 行番号 8 ファイル "foo.cc"
     8      printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(3)
c::~c で停止しました 行番号 9 ファイル "foo.cc"
     9      }
(dbx) step
c::~c で停止しました 行番号 8 ファイル "foo.cc"
     8      printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(5)
c::~c で停止しました 行番号 9 ファイル "foo.cc"
     9      )
(dbx) step
main で停止しました 行番号 24 ファイル "foo.cc"
    24      printf("caught exception %d\n", i);
(dbx) step
caught exception 99
main で停止しました 行番号 26 ファイル "foo.cc"
    26  }
```

C++ テンプレートでのデバッグ

dbx は C++ テンプレートをサポートしています。クラスおよび関数テンプレートを
含むプログラムを dbx に読み込み、クラスや関数に対して使用する任意の dbx コマ
ンドをテンプレートに対して次のように呼び出すことができます。

- クラスまたは関数テンプレートのインスタンス化にブレークポイントを設定する
(197 ページの「`stop inclass classname` コマンド」、198 ページの「`stop infunction
name` コマンド」、198 ページの「`stop infunction` コマンド」参照)
- すべてのクラスおよび関数テンプレートのインスタンス化のリストを出力する (196
ページの「`whereis name` コマンド」参照)
- テンプレートおよびインスタンスの定義を表示する (196 ページの「`whatis name`
コマンド」参照)
- メンバーテンプレート関数と関数テンプレートのインスタンス化を呼び出す (198
ページの「`call function_name(parameters)` コマンド」参照)
- 関数テンプレートのインスタンス化の値を出力する (199 ページの「`print` コマ
ンド」参照)
- 関数テンプレートのインスタンス化のソースコードを表示する (199 ページ
の「`list` コマンド」参照)

テンプレートの例

次のコード例は、クラステンプレート `Array` とそのインスタンス化、および関数テン
プレート `square` とそのインスタンス化を示しています。

```
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     }
```

この例の内容は次のとおりです。

- Array はクラステンプレート
- square は関数テンプレート
- Array<int> はクラステンプレートインスタンス化(テンプレートクラス)
- Array<int>::getLength はテンプレートクラスのメンバー関数
- square(int, int*) と square(double, double*) は関数テンプレートのインスタンス化(テンプレート関数)

C++ テンプレートのコマンド

次に示すコマンドは、テンプレートおよびインスタンス化されたテンプレートに使用します。クラスまたは型定義がわかったら、値の出力、ソースリストの表示、またはブレークポイントの設定を行うことができます。

whereis *name* コマンド

whereis コマンドは、関数テンプレートまたはクラステンプレートの、インスタンス化された関数やクラスの出現すべてのリストを出力するために使用します。

クラステンプレートの場合は、次のように入力します。

(dbx) **whereis Array**

```
メンバー関数: 'Array<int>::Array(int)
メンバー関数: 'Array<double>::Array(int)
クラステンプレートインスタンス: 'Array<int>
クラステンプレートインスタンス: 'Array<double>
クラステンプレート: 'a.out 'template_doc_2.cc' Array
```

関数テンプレートの場合は、次のように入力します。

(dbx) **whereis square**

```
関数テンプレートインスタンス: 'square<int>(__type_0,__type_0*)
関数テンプレートインスタンス: 'square<double>(__type_0,__type_0*)
```

`__type_0` パラメータは、0 番目のパラメータを表します。`__type_1` パラメータは、次のパラメータを表します。

詳細については、[388 ページの「whereis コマンド」](#)を参照してください。

whatis *name* コマンド

関数テンプレートおよびクラステンプレートと、インスタンス化された関数やクラスの定義を出力するために使用します。

クラステンプレートの場合は、次のように入力します。

(dbx) **whatis -t Array**

```
template<class T> class Array
完全なテンプレート宣言を得るために次を実行してください: 'whatis -t Array<int>';
```

クラステンプレートの構造については次のように実行します。

(dbx) **whatis Array**

```
識別子 'Array' が複数あります
次のうち 1 つ選択してください:
```

```

0) 取り消し
1) Array<int>::Array(int)
2) Array<double>::Array(int)
> 1
Array<int>::Array(int 1);

```

関数テンプレートの場合は、次のように入力します。

```

(dbx) whatis square
識別子 'square' が複数あります
次のうち 1 つ選択してください:
0) 取り消し
1) square<int(__type_0, __type_0*)
2) square<double>(__type_0, __type_0*)
> 2
void square<double>(double num, double *result);

```

クラステンプレートのインスタンス化の場合は、次のように入力します。

```

(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;
};

```

関数テンプレートのインスタンス化の場合は、次のように入力します。

```

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

```

詳細については、[383 ページの「whatis コマンド」](#)を参照してください。

stop inclass *classname* コマンド

テンプレートクラスのすべてのメンバー関数を停止するには、次のように入力します。

```

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

```

stop inclass コマンドを使用して、特定のテンプレートクラスのメンバー関数すべてにブレークポイントを設定します。

```
(dbx) stop inclass Array<int>
(2) stop inclass Array<int>
```

詳細については、[360 ページの「stop コマンド」](#) と [267 ページの「inclass classname \[-recurse | -norecurse\]」](#) を参照してください。

stop infunction *name* コマンド

stop infunction コマンドを利用して、指定した関数テンプレートのインスタンスにブレークポイントを設定します。

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

詳細については、[360 ページの「stop コマンド」](#) と [267 ページの「infunction function」](#) を参照してください。

stop in function コマンド

stop in コマンドを使用して、テンプレートクラスのメンバー関数、またはテンプレート関数にブレークポイントを設定します。

クラスインスタンス化のメンバーの場合は、次のとおりです。

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

関数インスタンス化の場合は、次のように入力します。

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

詳細については、[360 ページの「stop コマンド」](#) と [266 ページの「in function」](#) を参照してください。

call function_name(parameters) コマンド

スコープ内で停止した場合に、インスタンス化された関数やクラステンプレートのメンバー関数を明示的に呼び出すには、call コマンドを使用します。dbx で正しいインスタンスを決定できない場合、選択肢となる番号が付いたインスタンスのリストが表示されます。

```
(dbx) call square(j,&i)
```

詳細については、[289 ページの「call コマンド」](#) を参照してください。

print コマンド

print コマンドを使用して、インスタンス化された関数またはクラステンプレートメンバー関数を評価します。

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

printを使用して this ポインタを評価します。

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

詳細については、[344 ページの「print コマンド」](#)を参照してください。

list コマンド

list コマンドを使用して、指定のインスタンス化された関数のソースリストを出力します。

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

詳細については、[329 ページの「list コマンド」](#)を参照してください。

dbx を使用した Fortran のデバッグ

この章では、Fortran で使用されることが多いいくつかの dbx 機能を紹介します。dbx を使用して Fortran コードをデバッグするときの助けになる、dbx に対する要求の例も示してあります。

この章は次の各節から構成されています。

- 201 ページの「Fortran のデバッグ」
- 205 ページの「セグメント不正のデバッグ」
- 206 ページの「例外の検出」
- 207 ページの「呼び出しのトレース」
- 208 ページの「配列の操作」
- 210 ページの「組み込み関数」
- 210 ページの「複合式」
- 212 ページの「論理演算子」
- 213 ページの「Fortran 95 構造型の表示」
- 214 ページの「Fortran 95 構造型へのポインタ」

Fortran のデバッグ

次のアドバイスと概要は、Fortran プログラムをデバッグするときに役立ちます。dbx を使用した Fortran OpenMP コードのデバッグについては、178 ページの「イベントとの対話」を参照してください。

カレントプロシージャとカレントファイル

デバッグセッション中、dbx は、1つのプロシージャと1つのソースファイルをカレントとして定義します。ブレークポイントの設定要求と変数の出力または設定要求は、カレントの関数とファイルに関連付けて解釈されます。したがって、stop at 5 は、カレントファイルがどれであるかによって、3つの異なるブレークポイントのうち1つを設定します。

大文字

プログラムのいずれかの識別子に大文字が含まれる場合、dbx はそれらを認識します。いくつかの旧バージョンの場合のように、大文字/小文字を区別するコマンド、または区別しないコマンドを指定する必要はありません。

Fortran 95 と dbx は、大文字/小文字を区別するモードまたは区別しないモードのいずれかに統一する必要があります。

- 大文字/小文字を区別しないモードでコンパイルとデバッグを行うには、`-U` オプションを付けずにこれらの処理を行います。その場合、dbx `input_case_sensitive` 環境変数のデフォルト値は `false` になります。
ソースに `LAST` という変数がある場合、dbx では、`print LAST` コマンドおよび `print last` コマンドはいずれも要求どおりに動作します。Fortran 95 と dbx は、`LAST` と `last` を要求どおり同じものとして扱います。
- 大文字/小文字を区別するモードでコンパイルとデバッグを行うには、`-U` オプションを付けます。その場合、dbx `input_case_sensitive` 環境変数のデフォルト値は `true` になります。
ソースに `LAST` という変数と `last` という変数がある場合、dbx では、`print last` コマンドは動作しますが、`print LAST` コマンドは動作しません。Fortran 95 と dbx はいずれも、`LAST` と `last` を要求どおりに区別します。

注 - dbx `input_case_sensitive` 環境属性の環境変数を `false` に設定しても、dbx ではファイル名またはディレクトリ名について、大文字/小文字を常に区別します。

dbx のサンプルセッション

次の例では、サンプルプログラム `my_program` を使用します。

デバッグのための主プログラム `a1.f`

```
PARAMETER ( n=2 )
REAL twobytwo(2,2) / 4 *-1 /
CALL mkidentity( twobytwo, n )
PRINT *, determinant( twobytwo )
END
```

デバッグのためのサブルーチン `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

```

デバッグのための関数 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

```

▼ dbx のサンプルセッションの実行

- 1 -g オプションでコンパイルとリンクをします。

この処理は、まとめて1回または2回に分けて実行することができます。

-g フラグ付きコンパイルとリンクを1度にまとめて行います。

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

コンパイルとリンクを分けて行います。

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

- 2 実行可能ファイル my_program について dbx を起動します。

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

- 3 stop in subnam と入力して、最初の実行可能文の前にブレークポイントを設定する。subnam は、サブルーチン、関数、ブロックデータサブプログラムを示します。main プログラム中の最初の実行可能文で停止します。

```
(dbx) stop in MAIN
(2) stop in MAIN
```

通常 MAIN は大文字ですが、subnam は大文字でも小文字でもかまいません。

- 4 run コマンドを入力して、dbx からプログラムを実行します。dbx の起動時に指定された実行可能ファイルの中で、プログラムが実行されます。

```
(dbx) run
実行中: my_program
MAIN で停止しました 行位置 3 ファイル "a1.f"
    3      call mkidentity( twobytwo, n )
```

ブレークポイントに到達すると、dbx はどこで停止したかを示すメッセージを表示します。前述の例では、a1.f ファイルの行番号 3 で停止しています。

- 5 print コマンドを使用して、値を出力します。
n の値を出力します。

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

マトリックス twobytwo を出力します。

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

マトリックス array を出力します。

```
(dbx) print array
dbx: "array" が現在のスコープに定義されていません。
(dbx)
```

ここで array は定義されていないため、出力は失敗します (mkidentity 内でのみ有効)。

- 6 next コマンドを使用して、次の行に実行を進めます。
次の行に実行を進めます。

```
(dbx) next
MAIN で停止しました 行番号 4 ファイル "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 コマンドは現在のソース行を実行し、次のソース行で停止します。これは副プログラムの呼び出しを1つの文として数えます。

next コマンドと step コマンドを比較します。step コマンドは、ソースの次の行または副プログラムの次のステップを実行します。通常、次の実行可能ソース文がサブルーチンまたは関数呼び出しの場合、各コマンドは次の処理を行います。

- step コマンドは、副プログラムのソースの最初の文にブレークポイントを設定します。
- next コマンドは、呼び出し元のプログラム中で、呼び出しのあとの最初の文にブレークポイントを設定します。

7 quit コマンドを入力して、dbx を終了します。

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

セグメント不正のデバッグ

プログラムでセグメント不正 (SIGSEGV) が発生するのは、プログラムが使用可能なメモリー範囲外のメモリーアドレスを参照したことを示します。

セグメント不正の主な原因を次に示します。

- 配列インデックスが宣言された範囲外にある。
- 配列インデックス名のつづりが間違っている。
- 呼び出し元のルーチンでは引数に REAL を使用しているが、呼び出し先のルーチンでは INTEGER が使われている。
- 配列インデックスの計算が間違っている。
- 呼び出し元ルーチンの引数が足りない。
- ポインタを定義しないで使用している。

dbx により問題を見つける方法

問題のあるソース行を見つけるには、dbx を使用してセグメント例外が発生したソースコード行を検出します。

プログラムを使ってセグメント例外を生成します。

```
demo% cat WhereSEGV.f
      INTEGER a(5)
      j = 2000000
      DO 9 i = 1,5
```

```

        a(j) = (i * 10)
9    CONTINUE
    PRINT *, a
    END
demo%
```

dbx を使用してセグメント例外が発生した行番号を検出します。

```

demo% f95 -g -silent WhereSEGV.f
demo% a.out
セグメント例外
demo% dbx a.out
a.out のシンボル情報を読んでいます
シグナル SEGV でプログラムが停止しました (セグメント侵害)
(dbx) run
実行中: a.out
シグナル SEGV (障害アドレスにマッピングがありません)
ファイル "WhereSEGV.f" の行番号 4 の MAIN で
    4                a(j) = (i * 10)
(dbx)
```

例外の検出

プログラムが例外を受け取る原因は数多く考えられます。問題を見つける方法の1つとして、ソースプログラムで例外が発生した行番号を検出して調べる方法があります。

-ftrap=common によってコンパイルすると、すべての例外に対してトラップが強制的に行われます。

例外が発生した箇所を検索します。

```

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
wh の記号情報を読み込み中
(dbx) catch FPE
(dbx) run
```

```

実行中: wh
(プロセス id 17970)
シグナル FPE (ゼロによる浮動小数点除算) 関数 MAIN 行番号 2 ファイル "wh.f"
2          print *, r/s
(dbx)

```

呼び出しのトレース

プログラムがコアダンプで終了したため、終了するまでの呼び出しシーケンスが必要な場合があります。このシーケンスをスタックトレースといいます。

where コマンドは、プログラムフローの実行が停止した位置、およびどのようにその位置に達したかを表示します。これを呼び出し先ルーチンのスタックトレースといいます。

ShowTrace.f は、呼び出しシーケンスでコアダンプを数レベル深くする、つまりスタックトレースを示すために考えられたプログラムです。

逆順での注意:

```

demo% f77 -silent -g ShowTrace.f
demo% a.out
MAIN が calc を呼び出し、calc が calcb を呼び出します。
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quilt 174% dbx a.out
23 行目で実行が停止します。
a.out のシンボル情報を読んでいます
...
(dbx) run
calcb が calc の 9 行目で呼び出されました。
実行中: a.out
(プロセス id 1089)
calc が MAIN の 3 行目で呼び出されました。
シグナル SEGV (フォルトのアドレスにマッピングしていません) 関数
calcb 行番号 23 ファイル "ShowTrace.f"
23          v(j) = (i * 10)
(dbx) where -V
=>[1] calcb(v = ARRAY , m = 2) 行番号 23 ファイル "ShowTrace.f"
[2] calc(a = ARRAY , m = 2, d = 0) 行番号 9 ファイル "ShowTrace.f"
[3] MAIN() 行番号 3 ファイル "ShowTrace.f"
(dbx)
実行が停止した時点から呼び出しシーケンスを表示します。

```

配列の操作

dbx が配列を認識し、配列を出力します。

```
demo% dbx a.out
シンボル情報を読んでいます...
(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
実行中: a.out
MAIN で停止しました 行番号 7 ファイル "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 - 配列の要素を指定することができます
(dbx) quit
```

Fortran の配列のスライスについては、125 ページの「Fortran のための配列断面化構文」を参照してください。

Fortran 95 割り当て可能配列

次の例は、dbx で割り当て済み配列を処理する方法を示しています。

```
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
実行中: a.out
(プロセス id 10749)
Size?
1000
行番号 6 ではサイズは未知
MAIN で停止しました 行番号 6 ファイル "alloc.f95"
6 ALLOCATE( buffer(n), STAT=status )
(dbx) whatis buffer
integer*4 , allocatable::buffer(:)
(dbx) next
続き
MAIN で停止しました 行番号 7 ファイル "alloc.f95"
7 IF ( status /= 0 ) STOP 'cannot allocate buffer'
(dbx) whatis buffer
integer*4 buffer(1:1000)
行番号 9 ではサイズは既知
(dbx) cont
MAIN で停止しました 行番号 9 ファイル "alloc.f95"
9 PRINT *, buffer(n)
(dbx) print n
バッファ (1000) に 1000 を格納
n = 1000
(dbx) print buffer(n)
buffer(n) = 1000
```

組み込み関数

dbx は、Fortran の組み込み関数 (SPARC™ プラットフォームおよび x86 プラットフォームのみ) を認識します。

dbx での組み込み関数を示します。

```
demo% cat ShowIntrinsic.f
      INTEGER i
      i = -2
      END

(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
実行中: shi
(プロセス id 18019)
MAIN で停止しました 行番号 2 ファイル "shi.f"
      2          i = -2

(dbx) whatis abs
Generic intrinsic function: "abs"
(dbx) print i
i = 0
(dbx) step
MAIN で停止しました 行番号 3 ファイル "shi.f"
      3          end

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

複合式

dbx は、Fortran 複合式も認識します。

dbx での複合式を示します。

```
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
実行中: a.out
```

```

(プロセス id 10953)
MAIN で停止しました 行番号 2 ファイル "ShowComplex.f"
  2      z = ( 2.0, 3.0 )
(dbx) whatis z
complex*8 z
(dbx) print z
z = (0.0,0.0)
(dbx) next
MAIN で停止しました 行番号 3 ファイル "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%

```

間隔式の表示

dbx で間隔式を表示するには、次のように入力します。

```

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
実行中: a.out
(プロセス id 5217)
MAIN で停止しました 行番号 2 ファイル "ShowInterval.f95"
  2      v = [ 37.1, 38.6 ]
(dbx) whatis v
INTERVAL*16 v
(dbx) print v
v = [0.0,0.0]
(dbx) next
MAIN で停止しました 行番号 3 ファイル "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-間隔式は、SPARCプラットフォームで実行するよう、Solaris x86 SSE/SSE2 Pentium 4 互換プラットフォームでは `-xarch={sse|sse2}`、x64 プラットフォームでは `-xarch=amd64` を付けてコンパイルされたプログラムに対してのみサポートされま
す。

論理演算子

dbx は、Fortran の論理演算子を配置し、出力することができます。

dbx での論理演算子を示します。

```
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
実行中: a.out
(プロセス id 15394)
MAIN で停止しました 行番号 5 ファイル "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%
```

Fortran 95 構造型の表示

構造型、Fortran 95 構造型を dbx で表示できます。

```
demo% f95 -g DebStruct.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
実行中: a.out
(プロセス id 12326)
MAIN で停止しました 行番号 17 ファイル "DebStruct.f95"
 17 WRITE ( *, * ) prod1%name
(dbx) whatis prod1
product prod1
(dbx) whatis -t product
type product
  integer*4 id
  character*16 name
  character*8 model
  real*4 cost
  real*4 price
end type product
(dbx) n
(dbx) print prod1
prod1 = (
  id      = 82
  name    = 'Coffee Cup'
  model   = 'XL'
```

```

        cost = 24.0
        price = 104.0
    )

```

Fortran 95 構造型へのポインタ

構造型、Fortran 95 構造型およびポインタを dbx で表示できます。

```

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
   構造型を宣言します。

```

```

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

```

prod1 および prod2 ターゲットを宣言します。

```

10     TYPE(product), TARGET :: prod1, prod2
curr および prior ポインタを宣言します。

```

```

11     TYPE(product), POINTER :: curr, prior
12

```

curr が prod2 を指すようにします。

```

13     curr => prod2

```

prior が prod1 を指すようにします。

```

14     prior => prod1

```

prior を初期化します。

```

15     prior%id = 82
16     prior%name = "Coffee Cup"
17     prior%model = "XL"
18     prior%cost = 24.0
19     prior%price = 104.0

```

curr を prior に設定します。

```

20     curr = prior

```

curr および prior から名前を出力します。

```

21     WRITE ( *, * ) curr%name, " ", prior%name

```

```

22     END PROGRAM DebStruPtr

```

```

(dbx) stop at 21

```

```

(1) stop at "DebStruc.f95":21

```

```

(dbx) run
実行中: debstr
(プロセス id 10972)
MAIN で停止しました 行番号 21 ファイル "DebStruct.f95"
21      WRITE ( *, * ) curr%name, " ", prior%name
(dbx) print prod1
prod1 = (
      id = 82
      name = "Coffee Cup"
      model = "XL"
      cost = 24.0
      price = 104.0
)

```

前述において dbx は、構造型のすべての要素を表示します。

構造体を使用して、Fortran 95 構造型の項目について照会できます。

変数について尋ねます。

```

(dbx) whatis prod1
product prod1
型 (-t) について尋ねます。
(dbx) whatis -t product
type product
      integer*4 id
      character*16 name
      character*8 model
      real cost
      real price
end type product

```

ポインタを出力するには、次のようにします。

dbx は、アドレスであるポインタの内容を表示します。このアドレスは、実行のたびに異なる場合があります。

```

(dbx) print prior
prior = (
      id      = 82
      name   = 'Coffee Cup'
      model  = 'XL'
      cost   = 24.0
      price  = 104.0
)

```


◆◆◆ 17

第 17 章

dbx による Java アプリケーションのデバッグ

この章では、dbx を使い、Java™ コードと C JNI (Java Native Interface) コードまたは C++ JNI コードが混在するアプリケーションをデバッグする方法を説明します。

この章は次の節で構成されています。

- 217 ページの「dbx と Java コード」
- 218 ページの「Java デバッグ用の環境変数」
- 218 ページの「Java アプリケーションのデバッグの開始」
- 223 ページの「JVM ソフトウェアの起動方法のカスタマイズ」
- 227 ページの「dbx の Java コードデバッグモード」
- 228 ページの「Java モードにおける dbx コマンドの使用法」

dbx と Java コード

Sun Studio の dbx を使い、Solaris™ OS および Linux OS で動作する混在コード (Java コードと C コードまたは C++ コード) をデバッグすることができます。

Java コードに対する dbx の機能

dbx で数種類の Java アプリケーションをデバッグすることができます (218 ページの「Java アプリケーションのデバッグの開始」を参照)。大部分の dbx コマンドは、ネイティブコードと Java コードのどちらにも同様の働きをします。

Java コードのデバッグにおける dbx の制限事項

Java コードのデバッグでは、dbx に次の制限事項があります。

- ネイティブコードのときと異なり、コアファイルから Java アプリケーションの状態情報を入手することはできません。

- Java アプリケーションが何らかの理由で停止し、dbx が手続きを呼び出せない場合、Java アプリケーションの状態情報を入手することはできません。
- Java アプリケーションに、fix と cont、および実行時検査は使用できません。

Java デバッグ用の環境変数

ここでは、dbx を使った Java アプリケーションデバッグの専用の環境変数を説明します。JAVASRCPATH、CLASSPATHX、jvm_invocation 環境変数は、dbx を起動する前にシェルプロンプトから設定することができます。jdbx_mode 環境変数の値はアプリケーションのデバッグ中に変化します。ただし、jon コマンド (327 ページの「jon コマンド」) と joff コマンド (326 ページの「joff コマンド」) を使って変更することもできます。

jdbx_mode	jdbx_mode 環境変数の設定は次のとおりです。java、jni、または native。Java、JNI、ネイティブモードと、モードの変化の仕方および変化のタイミングについては、227 ページの「dbx の Java コードデバッグモード」を参照してください。デフォルトのモードは java です。
JAVASRCPATH	JAVASRCPATH 環境変数を使って、dbx が Java ソースファイルを探すディレクトリを指定することができます。この変数は、Java ソースファイルが .class や .jar ファイルと同じディレクトリにない場合に役立ちます。詳細については、222 ページの「Java ソースファイルの格納場所の指定」を参照してください。
CLASSPATHX	CLASSPATHX 環境変数を使って、独自のクラスローダーが読み込む Java クラスファイルのパスを指定することができます。詳細については、222 ページの「独自のクラスローダーを使用するクラスファイルのパスの指定」を参照してください。
jvm_invocation	jvm_invocation 環境変数を使って、JVM™ ソフトウェアの起動方法をカスタマイズすることができます (JVM は Java virtual machine の略語で、Java プラットフォーム用の仮想マシンを意味します)。詳細については、223 ページの「JVM ソフトウェアの起動方法のカスタマイズ」を参照してください。

Java アプリケーションのデバッグの開始

dbx では、次の種類の Java アプリケーションをデバッグすることができます。

- .class で終わるファイル名を持つファイル
- .jar で終わるファイル名を持つファイル
- ラッパーを使って起動する Java アプリケーション
- デバッグモードで起動した実行中の Java アプリケーションを dbx で接続 (アタッチ) する

- JNI_CreateJavaVM インタフェースを使って Java アプリケーションを埋め込む C および C++ アプリケーション

dbx は、これらのどの場合もデバッグ対象が Java アプリケーションであることを認識します。

クラスファイルのデバッグ

次の例に示すように dbx を使用することによって、ファイル名拡張子が `.class` のファイルをデバッグすることができます。

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

アプリケーションを定義しているクラスがパッケージに定義されている場合は、JVM ソフトウェアの制御下でアプリケーションを実行するときと同じで、次の例に示すように、パッケージのパスを指定する必要があります。

```
(dbx) debug java.pkg.Toy.class
```

クラスファイルのフルパス名を使用することもできます。この場合、dbx は `.class` ファイル内を調べることによってクラスパスのパッケージ部分を自動的に特定し、フルパス名の残りの部分をクラスパスに追加します。たとえば次のパス名の場合、dbx は `pkg/Toy.class` を主クラス名と判断し、クラスパスに `/home/user/java` を追加します。

```
(dbx) debug /home/user/java/pkg/Toy.class
```

JAR ファイルのデバッグ

Java アプリケーションは、JAR (Java Archive) ファイルにバンドルすることができます。JAR ファイルは、次の例に示すように dbx を使用することによってデバッグすることができます。

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

ファイル名が `.jar` で終わるファイルのデバッグを開始すると、dbx は、その JAR ファイルのマニフェストに指定されている `Main-Class` 属性を使って主クラスを特定します (主クラスは、アプリケーションのエントリーポイントになっている、JAR ファイル内のクラスです)。フルパス名または相対パス名を使って JAR ファイルが指定された場合、dbx は `Main-Class` 属性のクラスパスの前にそのディレクトリ名を追加します。

JAR ファイルに `Main-Class` 属性がない場合は、次の例に示すように Java 2 Platform, Standard Edition の `JarURLConnection` クラスに指定されている JAR の URL 構文、`jar:<url>!/{entry}` を使って、主クラスの名前を指定することができます。

```
(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
```

これらの例のどの場合も、dbx は次のことを行います。

- 文字 ! のあとに指定されたクラスパスを主クラスとみなします (例: /myclass.class または /x/y/z.class)。
- クラスパスに JAR ファイル名 (./myjar.jar、/a/b/c/d/e.jar、/a/b/c/d.jar) を追加します。
- 主クラスのデバッグを開始します。

注-jvm_invocation 環境変数を使って JVM ソフトウェアの起動方法をカスタマイズした場合は (223 ページの「JVM ソフトウェアの起動方法のカスタマイズ」を参照)、JAR ファイルのファイル名がクラスパスに追加されません。この場合は、デバッグを開始するときに JAR ファイルのファイル名をクラスパスに手動で追加する必要があります。

ラッパーを持つ Java アプリケーションのデバッグ

通常 Java アプリケーションには、環境変数を設定するためのラッパーがあります。Java アプリケーションにラッパーがある場合は、jvm_invocation 環境変数を設定することによって、ラッパースクリプトを使用することを dbx に知らせる必要があります (223 ページの「JVM ソフトウェアの起動方法のカスタマイズ」を参照)。

動作中の Java アプリケーションへの dbx の接続

Java アプリケーションを起動するときに次の例に示すオプションを指定することによって、動作中の Java アプリケーションに dbx を接続することができます。アプリケーションが起動すると、動作中のプロセスのプロセス ID を指定して dbx コマンドを実行することによって、デバッグを開始することができます (304 ページの「dbx コマンド」を参照)。

```
$ java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrunldb_agent myclass.class
$ dbx - 2345
```

JVM ソフトウェアが libldb_agent.so を見つけられるようにするには、Java アプリケーションを実行する前に正しいパスを LD_LIBRARY_PATH に追加する必要があります。

- SolarisOS を実行しているシステムで 32 ビットの JVM ソフトウェアを使用している場合は、/installation_directory/SUNWspro/lib/libldb_agent.so を追加します。

- Solaris OS を実行している SPARC システムで 64 ビットの JVM ソフトウェアを使用している場合は、`/installation_directory/SUNWspro/lib/v9/libdbx_agent.so` を `LD_LIBRARY_PATH` に追加します。
- Linux OS を実行している x64 システムで 64 ビットの JVM ソフトウェアを使用している場合は、`/installation_directory/sunstudio12/lib/amd64/libdbx_agent.so` を `LD_LIBRARY_PATH` に追加します。

`installation_directory` は Sun Studio ソフトウェアがインストールされている場所です。

動作中のアプリケーションに `dbx` を接続すると、`dbx` は Java モードでアプリケーションのデバッグを開始します。

Java アプリケーションが 64 ビットのオブジェクトライブラリを必要とする場合は、アプリケーションを起動するときに `-d64` オプションを追加してください。この場合、`dbx` はアプリケーションが動作している 64 ビットの JVM ソフトウェアを使用します。

```
$ java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrundbx_agent -d64 myclass.class
$ dbx - 2345
```

Java アプリケーションを埋め込む C/C++ アプリケーションのデバッグ

JNI `CreateJavaVM` インタフェースを使って Java アプリケーションを埋め込む C あるいは C++ アプリケーションをデバッグすることができます。この場合、C/C++ アプリケーションは、JVM ソフトウェアに次のオプションを指定することによって Java アプリケーションを起動することができます。

```
-Xdebug -Xnoagent -Xrundbx_agent
```

JVM ソフトウェアが `libdbx_agent.so` を見つけられるようにするには、Java アプリケーションを実行する前に正しいパスを `LD_LIBRARY_PATH` に追加する必要があります。

- Solaris OS を実行しているシステムで 32 ビットの JVM ソフトウェアを使用している場合は、`/installation_directory/SUNWspro/lib/libdbx_agent.so` を `LD_LIBRARY_PATH` に追加します。
- Solaris OS を実行している SPARC システムで 64 ビットの JVM ソフトウェアを使用している場合は、`/installation_directory/SUNWspro/lib/v9/libdbx_agent.so` を `LD_LIBRARY_PATH` に追加します。
- Linux OS を実行している x64 システムで 64 ビットの JVM ソフトウェアを使用している場合は、`/installation_directory/sunstudio12/lib/amd64/libdbx_agent.so` を `LD_LIBRARY_PATH` に追加します。

`installation_directory` は Sun Studio ソフトウェアがインストールされている場所です。

JVM ソフトウェアへの引数の引き渡し

Java モードで `run` コマンドを使用した場合、指定した引数は、JVM ソフトウェアではなく、アプリケーションに渡されます。JVM ソフトウェアに引数を渡す方法については、223 ページの「[JVM ソフトウェアの起動方法のカスタマイズ](#)」を参照してください。

Java ソースファイルの格納場所の指定

Java ソースファイルが、`.class` や `.jar` ファイルと異なるディレクトリに置かれていることがあります。その場合は、`$JAVASRCPATH` 環境変数を使って、`dbx` が Java ソースファイルを探すディレクトリを指定することができます。たとえば `JAVASRCPATH=./mydir/mysrc:/mydir/mylibsrc:/mydir/myutils` の場合、`dbx` は指定されたディレクトリで、デバッグ対象のクラスファイルに対応するソースファイルを探します。

C/C++ ソースファイルの格納場所の指定

次の場合は、`dbx` が C/C++ ソースファイルを見つけられないことがあります。

- ソースファイルの現在の格納場所がコンパイルしたときにあった場所と異なる場合
- `dbx` を実行しているシステムとは異なるシステムでソースファイルをコンパイルし、コンパイルディレクトリのパス名が異なる場合

このような場合、`dbx` がファイルを見つけられるよう、`pathmap` コマンドを使ってパス名を別のパス名に対応づけてください(342 ページの「[pathmap コマンド](#)」を参照)。

独自のクラスローダーを使用するクラスファイルのパスの指定

通常のクラスパスに含まれてない場所からクラスファイルを読み込む独自のクラスローダーが、アプリケーションに存在することがあります。そのような場合、`dbx` はクラスファイルを見つけられません。`CLASSPATHX` 環境変数を使って、独自のクラスローダーが読み込む Java クラスファイルのパスを指定することができます。たとえば `CLASSPATHX=./myloader/myclass:/mydir/mycustom` の場合、`dbx` は指定されたディレクトリでクラスファイルを探そうとします。

JVMソフトウェアによって読み込まれていないコードに対するブレークポイントの設定

JVMソフトウェアによって読み込まれていないクラスファイル内のJavaメソッドに停止ブレークポイントを設定するには、`stop in` コマンドでクラスのフル名を使用するか、`stop inmethod` コマンドでクラス名を使用します。次はその例です。

```
(dbx) stop in Java.Pkg.Toy.myclass.class.mymethod
(dbx) stop inmethod myclass.class.mymethod
```

JVMソフトウェアによって読み込まれていない共有ライブラリ内のC/C++関数に停止ブレークポイントを設定するには、ブレークポイントを設定する前に共有ライブラリのシンボルテーブルを事前に読み込みます。たとえば`myfunc`という関数を含む`mylibrary.so`というライブラリがある場合は、次のように入力することによって、ライブラリを事前に読み込み、関数にブレークポイントを設定することができます。

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

dbxでデバッグを開始する前に1回アプリケーションを実行することによって、動的に読み込まれたすべての共有オブジェクトのシンボルテーブルを読み込むことができます。

JVMソフトウェアの起動方法のカスタマイズ

次のことを行うために、dbxからのJVMソフトウェアの起動方法のカスタマイズが必要になることがあります。

- JVMソフトウェアのパス名を指定します(224ページの「JVMソフトウェアのパス名の指定」を参照)。
- JVMソフトウェアに`run`の引数を渡します(224ページの「JVMソフトウェアへの実行引数の引き渡し」を参照)。
- Javaアプリケーションの実行に際してデフォルトのJavaランパーではなく独自のランパーを指定します(225ページの「Javaアプリケーション用の独自のランパーの指定」を参照)。
- 64ビットのJVMソフトウェアを指定します(226ページの「64ビットJVMソフトウェアの指定」を参照)。

JVMソフトウェアの起動方法のカスタマイズは、`jvm_invocation`環境変数を使って行うことができます。この環境変数が定義されていない場合、デフォルトではdbxは次の設定でJVMソフトウェアを起動します。

```
java -Xdebug -Xnoagent -XrunDbx_agent:syncpid  
:
```

`jvm_invocation` 環境変数が定義されている場合は、その変数の値を使って JVM ソフトウェアを起動します。

`jvm_invocation` 環境変数の定義には、`-Xdebug` オプションを含める必要があります。`dbx` は、`-Xdebug` を内部オプションの `-Xdebug Xnoagent -XrunDbxagent::sync` に展開します。

次の例に示すように `-Xdebug` オプションが定義に含まれていない場合は、`dbx` からエラーメッセージが発行されます。

```
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
```

JVMソフトウェアのパス名の指定

デフォルトでは、JVMソフトウェアにパス名を指定しなかった場合、`dbx` はパス内の JVMソフトウェアを起動します。

JVMソフトウェアのパス名を指定するには、次の例に示すように、`jvm_invocation` 環境変数に適切なパス名を設定します。

```
jvm_invocation="/myjava/java -Xdebug"
```

この設定の場合、`dbx` は次の設定で JVMソフトウェアを起動します。

```
/myjava/java -Djava.compiler=NONE -Xdebug -Xnoagent -XrunDbx_agent:sync
```

JVMソフトウェアへの実行引数の引き渡し

JVMソフトウェアに実行引数を渡すには、次の例に示すように `jvm_invocation` 環境変数を設定することによって、それらの引数を付けて JVMソフトウェアを起動します。

```
jvm_invocation="java -Xdebug -Xms512 -Xmx1024 -Xcheck:jni"
```

この場合、`dbx` は次の設定で JVMソフトウェアを起動します。

```
java -Djava.compiler=NONE -Xdebug -Xnoagent -XrunDbx_agent:sync= -Xms512 -Xmx1024 -Xcheck:jni
```

Java アプリケーション用の独自のラッパーの指定

Java アプリケーションは起動時に独自のラッパーを使用することができます。その場合は、次の例に示すように `jvm_invocation` 環境変数を使って、利用するラッパーを指定します。

```
jvm_invocation="/export/siva-a/forte4j/bin/forte4j.sh -J-Xdebug"
```

この場合、`dbx` は次の設定で JVM ソフトウェアを起動します。

```
/export/siva-a/forte4j/bin/forte4j.sh - -J-Xdebug -J-Xnoagent -J-Xrundbxagent:sync=process_id
```

コマンド行オプションを受け付ける独自のラッパーの利用

次のラッパースクリプト (`xyz`) は複数の環境変数を設定して、コマンド行オプションを受け付けます。

```
#!/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
```

このスクリプトは、JVM ソフトウェアとユーザーアプリケーション用のコマンド行オプションを受け付けます。この形式のラッパースクリプトに対しては、次のように `jvm_invocation` 環境変数を設定して、`dbx` を起動します。

```
% jvm_invocation="xyz -J-Xdebug -Jany other java options"
% dbx myclass.class -Dide=visual
```

コマンド行オプションを受け付けない独自のラッパーの利用

次のラッパースクリプト (`xyz`) は複数の環境変数を設定して、JVM ソフトウェアを起動しますが、コマンド行オプションやクラス名を受け付けません。

```
#!/bin/sh
CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
```

```
ABC=/mydir/abc; export ABC
java <options> myclass
```

このようなスクリプトを次のいずれかの方法で利用し、dbxを使ってラッパーをデバッグすることもできます。

- `jvm_invocation` 変数の定義をスクリプトに追加することによって、ラッパースクリプトそのものからdbxが起動されるようにスクリプトを変更する。

```
#!/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
```

この変更を行うと、スクリプトを実行することによってデバッグセッションを開始することができます。

- 次に示すようにスクリプトを少し変更して、コマンド行オプションを受け付けられるようにする。

```
#!/bin/sh
CLASSPATH=/mydir/myclass:/mydir/myjar.jar; export CLASSPATH
ABC=/mydir/abc; export ABC
JAVA_OPTIONS="$1 <options>"
java $JAVA_OPTIONS $2
```

この変更を行なったら、次のように `jvm_invocation` 環境変数を設定して、dbxを起動します。

```
% jvm_invocation="xyz -Xdebug"; export jvm_invocation
% dbx myclass.class
```

64ビットJVMソフトウェアの指定

dbxで64ビットのJVMソフトウェアを起動して、64ビットのオブジェクトライブラリを必要とするアプリケーションをデバッグするには、`jvm_invocation` 環境変数の定義に `-d64` オプションを含めます。

```
jvm_invocation="/myjava/java -Xdebug -d64"
```

dbx の Java コードデバッグモード

Java アプリケーションのデバッグの場合、dbx は次の 3 つのモードのいずれかで動作します。

- Java モード
- JNI モード
- ネイティブモード

Java または JNI (Java Native Interface) モードでは、JNI コードを含めて Java アプリケーションの状態を調べ、コードの実行を制御することができます。ネイティブモードでは、C または C++ JNI コードの状態を調べることができます。現在のモード (java、jni、native) は、`jdbx_mode` 環境変数に記憶されます。

Java モードでは、Java 構文を使って dbx と対話します。dbx も Java 構文を使って情報を提供します。このモードは、純粋な Java コードか、Java コードと C JNI または C++ JNI コードが混在するアプリケーション内の Java コードのデバッグに使用します。

JNI モードでは、dbx はネイティブの構文を使用して、ネイティブコードにだけ作用しますが、コマンドの出力には、ネイティブの状態ばかりでなく、Java 関係の状態も示されるため、JNI モードは「混在」モードです。このモードは、Java コードと C JNI または C++ JNI コードが混在するアプリケーションのネイティブ部分のデバッグに使用します。

ネイティブモードでは、dbx コマンドはネイティブのプログラムにだけ作用し、Java 関係の機能はすべて無効になります。このモードは Java が関係しないプログラムのデバッグに使用します。

Java アプリケーションを実行すると、dbx は状況に応じて Java モードと JNI モードを自動的に切り替えます。たとえば、Java ブレークポイントを検出すると、dbx は Java モードに切り替わり、Java コードから JNI コードに入ると、JNI モードに切り替わります。

Java または JNI モードからネイティブモードへの切り替え

dbx は、自動的にネイティブモードに切り替わりません。Java または JNI モードからネイティブモードへは `joff` コマンド、ネイティブモードから Java モードへは `jon` コマンドを使って明示的に切り替えることができます。

実行中断時のモードの切り替え

たとえば Control-C を使って Java アプリケーションの実行が中断された場合、dbx はアプリケーションを安全な状態にして、すべてのスレッドを一時停止することによって、自動的にモードを Java/JNI モードに切り替えようとします。

アプリケーションを一時停止して Java/JNI モードに切り替えることができない場合、dbx はネイティブモードに切り替わります。この場合でも、jon コマンドを使用して、Java モードに切り替え、プログラムの状態を調べることができます。

Java モードにおける dbx コマンドの使用法

Java コードとネイティブコードが混在するアプリケーションのデバッグに使用する dbx コマンドは、次のように分類することができます。

- 受け付ける引数と機能が Java/JNI モードとネイティブモードで完全に同じコマンド (229 ページの「[構文と機能が Java モードとネイティブモードで完全に同じコマンド](#)」を参照)。
- Java または JNI モードとネイティブモードの間で有効な引数が異なるコマンド (230 ページの「[Java モードで構文が異なる dbx コマンド](#)」を参照)。
- Java または JNI モードでのみ有効なコマンド (232 ページの「[Java モードでのみ有効なコマンド](#)」を参照)。

どの分類にも属さないコマンドはすべてネイティブモードでのみ動作します。

dbx コマンドにおける Java の式の評価

大部分の dbx コマンドで使用される Java の式の評価機能は次の構造をサポートしています。

- すべてのリテラル
- すべての名前とフィールドアクセス
- `this` および `super`
- 配列アクセス
- キャスト
- 条件付きの二項演算
- メソッドの呼び出し
- その他の単項/二項演算
- 変数またはフィールドへの値の代入
- `instanceof` 演算子
- 配列の長さ演算子

サポートされていない構造は次のとおりです。

- 修飾付きの `this` (例: `<ClassName>.this`)
- クラスのインスタンス作成式
- 配列の作成式
- 文字列連結演算子
- 条件演算子 `?:`

- 複合代入演算子 (例: `x += 3`)

Java アプリケーションの情報を調べるうえで特に有用な方法は、dbx デバッガの表示 (`display`) 機能を利用する方法です。

データを調べる以上の操作を行う式に対して正確な値解釈を依存します。

dbx コマンドが利用する静的および動的情報

通常、Java アプリケーションに関する情報の多くは、JVM ソフトウェアが起動してからのみ利用でき、終了すると利用できなくなります。ただし、Java アプリケーションのデバッグでは、dbx は、JVM ソフトウェアを起動する前にシステムクラスパスとユーザークラスパスに含まれているクラスファイルと JAR ファイルから必要な情報の一部を収集します。この情報のおかげで dbx は、アプリケーションの実行前にブレークポイントで綿密なエラー検査を行うことができます。

一部 Java クラスとその属性に、クラスパスからアクセスできないことがあります。dbx はそうしたクラスを調べて、ステップ実行することができ、式パーサーはそれらが読み込まれてからアクセスできるようになります。ただし、dbx が収集する情報は一時的な情報であり、JVM ソフトウェアが終了すると利用できなくなります。

Java アプリケーションのデバッグに dbx が必要とする情報はどこにも記録されません。このため dbx は、Java のソースファイルを読み取り、コードをデバッグしながらその情報を取得しようとします。

構文と機能が Java モードとネイティブモードで完全に同じコマンド

ここでは、構文と行う処理が Java モードとネイティブモードで完全に同じ dbx コマンドをまとめています。

コマンド	機能
<code>attach</code>	動作中のプロセスに dbx を接続します。プログラムは停止して、デバッグの制御下に置かれます。
<code>cont</code>	プロセスが実行を再開します。
<code>dbxenv</code>	dbx 環境変数を一覧表示するか、設定します。
<code>delete</code>	ブレークポイントとその他のイベントを削除します。
<code>down</code>	呼び出しスタックを下方方向に移動します (main の逆方向)。

コマンド	機能
dump	プロシージャまたはメソッドにローカルなすべての変数を表示します。
file	現在のファイルを表示するか、変更します。
frame	現在のスタックフレーム番号を表示するか、変更します。
handler	イベントハンドラ(ブレークポイント)を変更します。
import	dbx コマンドライブラリからコマンドをインポートします。
line	現在の行番号を表示するか、変更します。
list	現在の行番号を表示するか、変更します。
next	ソース行を1行ステップ実行します(呼び出しをステップオーバー)。
pathmap	ソースファイルなどの検索でパス名を別のパス名に対応づけます。
proc	現在のプロセスの状態を表示します。
prog	デバッグ対象のプログラムとその属性を管理します。
quit	dbx を終了します。
rerun	引数なしでプログラムを実行します。
runargs	ターゲットプロセスの引数を変更します。
status	イベントハンドラ(ブレークポイント)を一覧表示します。
step up	ステップアップして、現在の関数またはメソッドを出ます。
stepi	機械命令を1つステップ実行します(呼び出しにステップイン)。
up	呼び出し方向を上方向に移動します(main 方向)
whereami	現在のソース行を表示します。

Java モードで構文が異なる dbx コマンド

ここでは、Java のデバッグとネイティブコードのデバッグで構文が異なる dbx コマンドをまとめています。これらのコマンドは、Java モードとネイティブモードで動作が異なります。

コマンド	ネイティブモードでの機能	Java モードでの機能
assign	プログラム変数に新しい値を代入します。	局所変数またはパラメータに新しい値を代入します。
call	手続きを呼び出します。	メソッドを呼び出します。
dbx	dbx を起動します。	dbx を起動します。
debug	指定されたアプリケーションを読み込んで、アプリケーションのデバッグを開始します。	指定された Java アプリケーションを読み込んで、クラスファイルの有無を調べ、アプリケーションのデバッグを開始します。
detach	dbx の制御下にあるターゲットプロセスを解放します。	dbx の制御下にあるターゲットプロセスを解放します。
display	あらゆる停止点で式を評価して表示します。	あらゆる停止点で式か局所変数、パラメータを評価して表示します。
files	正規表現に一致するファイル名を一覧表示します。	dbx が認識しているすべての Java ソースファイルを一覧表示します。
func	現在の関数を表示するか、変更します。	現在のメソッドを表示するか、変更します。
next	ソースを 1 行ステップ実行します (呼び出しをステップオーバー)。	ソースを 1 行ステップ実行します (呼び出しをステップオーバー)。
print	式の値を表示します。	式か局所変数、パラメータの値を表示します。
run	引数を付けてプログラムを実行します。	引数を付けてプログラムを実行します。
step	ソースを 1 行か 1 文ステップ実行します (呼び出しにステップイン)。	ソースを 1 行か 1 文ステップ実行します (呼び出しにステップイン)。
stop	ソースレベルのブレークポイントを設定します。	ソースレベルのブレークポイントを設定します。
thread	現在のスレッドを表示するか、変更します。	現在のスレッドを表示するか、変更します。
threads	すべてのスレッドを一覧表示します。	すべてのスレッドを一覧表示します。

コマンド	ネイティブモードでの機能	Java モードでの機能
trace	実行されたソース行か関数呼び出し、変数の変更を表示します。	実行されたソース行か関数呼び出し、変数の変更を表示します。
undisplay	display コマンドを取り消します。	display コマンドを取り消します。
whatis	式の型または型の宣言を表示します。	識別子の宣言を表示します。
when	指定されたイベントが発生したときにコマンドを実行します。	指定されたイベントが発生したときにコマンドを実行します。
where	呼び出しスタックを表示します。	呼び出しスタックを表示します。

Java モードでのみ有効なコマンド

ここでは、Java または JNI モードでのみ有効な dbx コマンドをまとめています。

コマンド	機能
java	JNI モードのときに、指定したコマンドの Java 版を実行するよう指示するときに使用します。
javaclasses	コマンドが入力された時点で dbx が認識しているすべての Java クラス名を表示します。
joff	Java または JNI モードからネイティブモードに dbx を切り替えます。
jon	ネイティブモードから Java モードに dbx を切り替えます。
jpgks	コマンドが入力された時点で dbx が認識しているすべての Java パッケージ名を表示します。
native	Java モードのときに、指定したコマンドのネイティブ版を実行するよう指示するときに使用します。

機械命令レベルでのデバッグ

この章は、イベント管理コマンドやプロセス制御コマンドを機械命令レベルで使用方法と、特定のアドレスにおけるメモリーの内容を表示する方法、対応する機械命令とともにソース行を表示する方法を説明します。コマンド `next`、`step`、`stop`、`trace` のそれぞれに、対応する機械命令レベルのコマンド `nexti`、`stepi`、`stopi`、`tracei` が用意されています。regs コマンドは、機械語レジスタを出力するために使用できます。また、`print` コマンドは、個々のレジスタを出力するために使用できます。

この章の内容は次のとおりです。

- 233 ページの「メモリーの内容を調べる」
- 238 ページの「機械命令レベルでのステップ実行とトレース」
- 240 ページの「機械命令レベルでブレークポイントを設定する」
- 240 ページの「regs コマンドの使用」

メモリーの内容を調べる

アドレスと `examine` または `x` コマンドを使用して、メモリーロケーションの内容を調べたり、各アドレスでアセンブリ言語命令を出力したりすることができます。アセンブリ言語のデバッガである `adb(1)` から派生したコマンドを使用して、次の項目について問い合わせることができます。

- アドレス - "=" (等号) を使用。
- あるアドレスに格納されている内容 - "/" (スラッシュ) を使用。

`dis`、`listi` コマンドを使用して、アセンブリ命令とメモリーの内容を調べることができます (236 ページの「`dis` コマンドの使用」と 237 ページの「`listi` コマンドの使用」を参照)。

examine または x コマンドの使用

examine コマンドまたはその別名 x を使用すると、メモリーの内容やアドレスを表示することができます。

あるメモリーの内容を表示するには、書式 *format* の *count* 項目の *address* で表される次の構文を使用します。デフォルトの *address* は、前に表示された最後のアドレスの次のアドレスになります。デフォルトの *count* は 1 です。デフォルト *format* は、前の examine または x コマンドで使用されたものと同じです。

examine コマンドの構文は次のとおりです。

```
examine [address] [/ [count] [format]]
```

address1 から *address2* までのメモリー内容を書式 *format* で表示するには、次のように入力します。

```
examine address1, address2 [/ [format]]
```

アドレスの内容ではなくアドレスを指定の書式で表示するには、次のように入力します。

```
examine address = [format]
```

examine によって最後に表示されたアドレスの次のアドレスに格納された値を出力するには、次のように入力します。

```
examine +/ i
```

式の結果を出力するには、式をアドレスとして入力します。

```
examine address=format
```

```
examine address=
```

アドレス (address)

address はアドレスの絶対値、またはアドレスとして使用できる任意の式です。+(プラス記号) はデフォルトのアドレスの次のアドレスを表します。

たとえば、次のアドレスは有効です。

0xff99	絶対アドレス
main	関数のアドレス
main+20	関数アドレス+オフセット

<code>&errno</code>	変数のアドレス
<code>str</code>	文字列を指すポインタ変数

メモリーを表示するためのアドレス表現は、名前の前にアンパサンド `&` を付けて指定します。関数名はアンパサンドなしで使用できます。`&main` は `main` と同じです。レジスタは、名前の前にドル記号 `$` を付けることによって表します。

書式 (format)

`format` は、`dbx` がアドレスの問い合わせ結果を表示するときの書式です。生成される出力は、現在の表示書式 `format` によって異なります。表示書式を変更する場合は、異なる `format` コードを使用してください。

各 `dbx` セッションの初めに設定されるデフォルトの書式は `x` です。このとき、16 進表記のアドレスと値が 1 ワード (32 ビット) で表示されます。次の表は、表示書式の一覧です。

<code>i</code>	アセンブラ命令として表示
<code>d</code>	10 進表記の 16 ビット (2 バイト) で表示
<code>D</code>	10 進表記の 32 ビット (4 バイト) で表示
<code>o</code>	8 進表記の 16 ビット (2 バイト) で表示
<code>O</code>	8 進表記の 32 ビット (4 バイト) で表示
<code>x</code>	16 進表記の 16 ビット (2 バイト) で表示
<code>X</code>	16 進表記の 32 ビット (4 バイト) で表示 (デフォルト書式)
<code>b</code>	8 進表記のバイトで表示
<code>c</code>	1 バイトの文字で表示
<code>w</code>	ワイド文字列で表示
<code>-s</code>	NULL バイトで終わる文字列で表示
<code>W</code>	ワイド文字列で表示
<code>f</code>	単精度浮動小数点数として表示
<code>F,g</code>	倍精度浮動小数点数として表示
<code>E</code>	拡張精度浮動小数点数として表示
<code>ld,LD</code>	10 進数として 32 ビット (4 バイト) で表示 (D と同じ)
<code>lo,LO</code>	8 進数として 32 ビット (4 バイト) で表示 (O と同じ)

lx, LX	16進数として32ビット(4バイト)で表示(Xと同じ)
Ld, LD	10進数として64ビット(8バイト)で表示
Lo, LO	8進数として64ビット(8バイト)で表示
Lx, LX	16進数として64ビット(8バイト)で表示

繰り返し (count)

count は、10進法での反復カウントを示します。増分サイズは、メモリーの表示書式によって異なります。

アドレスの使用例

次の例は、*count* および *format* の各オプションを付けてアドレスを使用して、現在の停止点から始まる5つの連続する分解された命令を表示する方法を示しています。

SPARC システムの場合:

```
(dbx) stepi
stopped in main at 0x108bc
0x000108bc: main+0x000c: st    %l0, [%fp - 0x14]
(dbx) x 0x108bc/5i
0x000108bc: main+0x000c: st    %l0, [%fp - 0x14]
0x000108c0: main+0x0010: mov   0x1,%l0
0x000108c4: main+0x0014: or    %l0,%g0, %o0
0x000108c8: main+0x0018: call  0x00020b90 [unresolved PLT 8: malloc]
0x000108cc: main+0x001c: nop
```

x86 システムの場合:

```
(dbx) x &main/5i
0x08048988: main      : pushl %ebp
0x08048989: main+0x0001: movl  %esp,%ebp
0x0804898b: main+0x0003: subl  $0x28,%esp
0x0804898e: main+0x0006: movl  0x8048ac0,%eax
0x08048993: main+0x000b: movl  %eax,-8(%ebp)
```

dis コマンドの使用

このコマンドは、表示書式を *i* として指定した *examine* コマンドと同じです。

dis コマンドの構文は次のようになります。

```
dis [address] [address1, address2] [/count]
```

dis コマンドの動作は次のとおりです。

- 引数なしで実行すると、+で始まる 10 の命令を表示します。
- 引数 *address* だけを指定して実行すると、*address* で始まる 10 の命令を逆アセンブルします。
- 引数 *address* と *count* を指定して実行すると、*address* で始まる *count* 命令を逆アセンブルします。
- 引数 *address1* と *address2* を指定して実行すると、*address1* から *address2* までの命令を逆アセンブルします。
- *count* だけを指定して実行すると、+で始まる *count* 命令を表示します。

listi コマンドの使用

対応するアセンブリ命令とともにソース行を表示するには listi コマンドを使用します。これは list -i と同じです。71 ページの「ソースリストの出力」の list -i についての説明を参照してください。

SPARC システムの場合:

```
(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]
```

x86 システムの場合:

```
(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)
```

機械命令レベルでのステップ実行とトレース

機械命令レベルの各コマンドは、対応するソースレベルのコマンドと同じように動作します。ただし、動作の単位はソース行ではなく、単一の命令です。

機械命令レベルでステップ実行する

ある機械命令から次の機械命令に1つだけステップ実行するには、`nexti` コマンドまたは `stepi` コマンドを使用します。

`nexti` コマンドと `stepi` コマンドは、それぞれに対応するソースコードレベルのコマンドと同じように動作します。すなわち、`nexti` コマンドは `over` 関数を実行し、`stepi` は次の命令が呼び出した関数をステップ実行します (呼び出された関数の最初の命令で停止します)。コマンドの書式も同じです。詳細については、[339 ページの「next コマンド」](#)と [358 ページの「step コマンド」](#)を参照してください。

`nexti` と `stepi` の出力は、対応するソースレベルのコマンドの場合と次の2つの違いがあります。

- ソースコードの行番号の代わりに、プログラムが停止したアドレスが出力に含まれる。
- ソースコード行の代わりに、デフォルトの出力に逆アセンブルされた命令が示される。

たとえば、次のようにします。

```
(dbx) func
hand::ungrasp
(dbx) nexti
ungrasp +0x18: call support
(dbx)
```

詳細については、[341 ページの「nexti コマンド」](#)と [360 ページの「stepi コマンド」](#)を参照してください。

機械命令レベルでトレースする

機械命令レベルでのトレースは、ソースコードレベルでのトレースと同じように行われます。ただし、`tracei` コマンドを使用する場合は例外です。`tracei` コマンドでは、実行中のアドレスまたはトレース対象の変数の値がチェックされた場合にだけ、単一の命令が実行されます。`tracei` コマンドは、`stepi` のような動作を自動的に行います。すなわち、プログラムは1度に1つの命令だけ進み、関数呼び出しに入ります。

`tracei` コマンドを使用すると、各命令が実行され、アドレスの実行またはトレース中の変数または式の値を `dbx` が調べている間、プログラムは一瞬停止します。このように `tracei` コマンドの場合、実行速度がかなり低下します。

トレースとそのイベント仕様および修飾子については、[105 ページの「トレースの実行」](#)と [376 ページの「`tracei` コマンド」](#) を参照してください。

`tracei` コマンドの一般的な構文は次のとおりです。

```
tracei event-specification [modifier]
```

一般的に使用される `tracei` コマンドの書式は次のとおりです。

<code>tracei step</code>	各命令をトレース
<code>tracei next</code>	各命令をトレースするが、呼び出しを飛び越します。
<code>tracei at address</code>	指定のコードアドレスをトレース

詳細については、[376 ページの「`tracei` コマンド」](#) を参照してください。

SPARC の場合は次のようになります。

```
(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
....
....
```

機械命令レベルでブレークポイントを設定する

機械命令レベルでブレークポイントを設定するには、`stopi` コマンドを使用します。`stopi` は次の構文を使用して *event specification* を受け入れます。

```
stopi event-specification [modifier]
```

一般的に使用される `stopi` コマンドの書式は次のとおりです。

```
stopi [at address] [-if cond]
stopi in function [-if cond]
```

詳細については、[365 ページの「stopi コマンド」](#) を参照してください。

あるアドレスにブレークポイントを設定する

特定のアドレスにブレークポイントを設定するには、コマンドラインで次のように入力します。

```
(dbx) stopi at address
```

たとえば、次のようにします。

```
(dbx) nexti
hand::ungrasp で停止しました 0x12638
(dbx) stopi at &hand::ungrasp
(3) stopi at &hand::ungrasp
(dbx)
```

regs コマンドの使用

`regs` コマンドを使用すると、すべてのレジスタの値を表示することができます。

次に、`regs` コマンドの構文を示します。

```
regs [-f][-F]
```

`-f` には、浮動小数点レジスタ (単精度) が含まれます。`-F` には、浮動小数点レジスタ (倍精度) が含まれます。

詳細については、348 ページの「regs コマンド」を参照してください。

SPARC システムの場合：

```
dbx[13] regs -F
現スレッド: t@1
現フレーム: [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
...
```

x64 システムの場合：

```
(dbx) regs
current frame: [1]
r15      0x0000000000000000
r14      0x0000000000000000
r13      0x0000000000000000
r12      0x0000000000000000
r11      0x000000000401b58
r10      0x0000000000000000
r9       0x000000000401c30
r8       0x000000000416cf0
rdi      0x000000000416cf0
rsi      0x000000000401c18
rbp      0xfffffd7ffdf820
rbx      0xfffffd7fff3fb190
rdx      0x000000000401b50
rcx      0x000000000401b54
rax      0x000000000416cf0
trapno   0x0000000000000003
err      0x0000000000000000
rip      0x000000000401709:main+0xf9    movl $0x0000000000000000,0xfffffffffffc(%rbp)
cs       0x000000000000004b
```

```
eflags 0x0000000000000206
rsp    0xfffffd7ffdf7b0
ss     0x000000000000043
fs     0x0000000000001bb
gs     0x000000000000000
es     0x000000000000000
ds     0x000000000000000
fsbase 0xfffffd7ff3a2000
gsbase 0xffffffff8000000
(dbx) regs -F
current frame: [1]
r15    0x000000000000000
r14    0x000000000000000
r13    0x000000000000000
r12    0x000000000000000
r11    0x000000000401b58
r10    0x000000000000000
r9     0x000000000401c30
r8     0x000000000416cf0
rdi    0x000000000416cf0
rsi    0x000000000401c18
rbp    0xfffffd7ffdf820
rbx    0xfffffd7ff3fb190
rdx    0x000000000401b50
rcx    0x000000000401b54
rax    0x000000000416cf0
trapno 0x000000000000003
err    0x000000000000000
rip    0x000000000401709:main+0xf9    movl    $0x000000000000000,0xffffffffffffc(%rbp)
cs     0x000000000000004b
eflags 0x0000000000000206
rsp    0xfffffd7ffdf7b0
ss     0x000000000000043
fs     0x0000000000001bb
gs     0x000000000000000
es     0x000000000000000
ds     0x000000000000000
fsbase 0xfffffd7ff3a2000
gsbase 0xffffffff8000000
st0    +0.0000000000000000e+00
st1    +0.0000000000000000e+00
st2    +0.0000000000000000e+00
st3    +0.0000000000000000e+00
st4    +0.0000000000000000e+00
st5    +0.0000000000000000e+00
st6    +0.0000000000000000e+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)

```

プラットフォーム固有のレジスタ

次の表は、式で使用できる SPARC、x86、および AMD64 の各アーキテクチャーのプラットフォームに固有のレジスタ名を示しています。

SPARC レジスタ情報

[Please define the SPARCsans text entity] アーキテクチャーのレジスタ情報は次のとおりです。

レジスタ	内容の説明
\$g0 through \$g7	「大域」レジスタ
\$o0 through \$o7	「出力」レジスタ
\$l0 through \$l7	「局所」レジスタ
\$i0 through \$i7	「入力」レジスタ
\$fp	フレームポインタ (レジスタ \$i6 と等価)
\$sp	スタックポインタ (レジスタ \$o6 と等価)

レジスタ	内容の説明
\$y	Y レジスタ
\$psr	プロセッサ状態レジスタ
\$wim	ウィンドウ無効マスクレジスタ
\$tbr	トラップベースレジスタ
\$pc	プログラムカウンタ
\$npc	次のプログラムカウンタ
\$f0 through \$f31	FPU "F" レジスタ
\$fsr	FPU 状態レジスタ
\$fq	FPU キュー

\$f0f1 \$f2f3 ... \$f30f31 のような浮動小数点レジスタのペアは、C の "double" 型とみなされます。通常、\$fN レジスタはC の "float" 型とみなされます。これらのペアは、\$d0 ... \$d30。

次の追加レジスタは、SPARC V9 および V8+ ハードウェアで使用できます。

```
$g0g1 through $g6g7
$o0o1 through $o6o7
$xfsr $tstate $gsr
$f32f33 $f34f35 through $f62f63 ($d32 ... $$d62)
```

SPARC のレジスタとアドレッシングの詳細については、『SPARC アーキテクチャー マニュアルバージョン 8』(トッパン刊) および『SPARC Assembly Language Reference Manual』を参照してください。

x86 レジスタ情報

x86 アーキテクチャーのレジスタ情報は次のとおりです。

レジスタ	内容の説明
\$gs	代替データセグメントレジスタ
\$fs	代替データセグメントレジスタ
\$es	代替データセグメントレジスタ
\$ds	データセグメントレジスタ

レジスタ	内容の説明
\$edi	デスティネーションインデックスレジスタ
\$esi	ソースインデックスレジスタ
\$ebp	フレームポインタ
\$esp	スタックポインタ
\$ebx	汎用レジスタ
\$edx	汎用レジスタ
\$ecx	汎用レジスタ
\$eax	汎用レジスタ
\$trapno	例外ベクトル番号
\$err	例外を示すエラーコード
\$eip	命令ポインタ
\$cs	コードセグメントレジスタ
\$eflags	フラグ
\$uesp	ユーザースタックポインタ
\$ss	スタックセグメントレジスタ

一般的に使用されるレジスタには、マシンに依存しない名前が別名として指定されます。

レジスタ	内容の説明
\$SP	スタックポインタ (\$uesp と同じ)。
\$pc	プログラムカウンタ (\$eip と同じ)。
\$fp	フレームポインタ (\$ebp と同じ)。

80386 用の下位 16 ビットのレジスタは次のとおりです。

レジスタ	内容の説明
\$ax	汎用レジスタ
\$cx	汎用レジスタ

レジスタ	内容の説明
\$dx	汎用レジスタ
\$bx	汎用レジスタ
\$si	ソースインデックスレジスタ
\$di	デスティネーションインデックスレジスタ
\$ip	命令ポインタ(下位 16 ビット)
\$flags	フラグ(下位 16 ビット)

上記のうち最初の 4 つの 80386 用 16 ビットレジスタは、8 ビットずつに分割できます。

レジスタ	内容の説明
\$al	レジスタの下位(右)部分 \$ax
\$ah	レジスタの上位(左)部分 \$ax
\$cl	レジスタの下位(右)部分 \$cx
\$ch	レジスタの上位(左)部分 \$cx
\$dl	レジスタの下位(右)部分 \$dx
\$dh	レジスタの上位(左)部分 \$dx
\$bl	レジスタの下位(右)部分 \$bx
\$bh	レジスタの上位(左)部分 \$bx

80387 用レジスタは次のとおりです。

レジスタ	内容の説明
\$fctrl	コントロールレジスタ
\$fstat	状態レジスタ
\$ftag	タグレジスタ
\$fip	命令ポインタオフセット
\$fcs	コードセグメントセクタ
\$fopoff	オペランドポインタオフセット

レジスタ	内容の説明
\$fopse1	オペランドポインタセクタ
\$st0 through \$st7	データレジスタ

AMD64 レジスタ情報

AMD64 アーキテクチャのレジスタ情報は次のとおりです。

レジスタ	内容の説明
rax	汎用レジスタ - 関数呼び出しの引数の引き渡し
rbx	汎用レジスタ - 呼び出し先保存
rcx	汎用レジスタ - 関数呼び出しの引数の引き渡し
rdx	汎用レジスタ - 関数呼び出しの引数の引き渡し
rbp	汎用レジスタ - スタック管理/フレームポインタ
rsi	汎用レジスタ - 関数呼び出しの引数の引き渡し
rdi	汎用レジスタ - 関数呼び出しの引数の引き渡し
rsp	汎用レジスタ - スタック管理/スタックポインタ
r8	汎用レジスタ - 関数呼び出しの引数の引き渡し
r9	汎用レジスタ - 関数呼び出しの引数の引き渡し
r10	汎用レジスタ - 一時レジスタ
r11	汎用レジスタ - 一時レジスタ
r12	汎用レジスタ - 呼び出し先保存
r13	汎用レジスタ - 呼び出し先保存
r14	汎用レジスタ - 呼び出し先保存
r15	汎用レジスタ - 呼び出し先保存
rflags	フラグレジスタ
rip	命令ポインタ
mmx0/st0	64ビットメディアおよび浮動小数点レジスタ
mmx1/st1	64ビットメディアおよび浮動小数点レジスタ
mmx2/st2	64ビットメディアおよび浮動小数点レジスタ

レジスタ	内容の説明
mmx3/st3	64 ビットメディアおよび浮動小数点レジスタ
mmx4/st4	64 ビットメディアおよび浮動小数点レジスタ
mmx5/st5	64 ビットメディアおよび浮動小数点レジスタ
mmx6/st6	64 ビットメディアおよび浮動小数点レジスタ
mmx7/st7	64 ビットメディアおよび浮動小数点レジスタ
xmm0	128 ビットメディアレジスタ
xmm1	128 ビットメディアレジスタ
xmm2	128 ビットメディアレジスタ
xmm3	128 ビットメディアレジスタ
xmm4	128 ビットメディアレジスタ
xmm5	128 ビットメディアレジスタ
xmm6	128 ビットメディアレジスタ
xmm7	128 ビットメディアレジスタ
xmm8	128 ビットメディアレジスタ
xmm9	128 ビットメディアレジスタ
xmm10	128 ビットメディアレジスタ
xmm11	128 ビットメディアレジスタ
xmm12	128 ビットメディアレジスタ
xmm13	128 ビットメディアレジスタ
xmm14	128 ビットメディアレジスタ
xmm15	128 ビットメディアレジスタ
cs	セグメントレジスタ
os	セグメントレジスタ
es	セグメントレジスタ
fs	セグメントレジスタ
gs	セグメントレジスタ
ss	セグメントレジスタ
fcw	fxsave および fxstor メモリーイメージ制御ワード

レジスタ	内容の説明
fsw	fxsave および fxstor メモリーイメージステータスワード
ftw	fxsave および fxstor メモリーイメージタグワード
fop	fxsave および fxstor メモリーイメージ最終 x87 オペコード
frip	fxsave および fxstor メモリーイメージ 64 ビットオフセットからコードセグメントへ
frdp	fxsave および fxstor メモリーイメージ 64 ビットオフセットからデータセグメントへ
mxcsr	fxsave および fxstor メモリーイメージ 128 メディア命令制御およびステータスレジスタ
mxcsr_mask	mxcsr_mask のビットを設定し、mxcsr でサポートされる機能ビットを示す

◆◆◆ 第 19 章

dbx の Korn シェル機能

dbx コマンド言語は Korn シェル (ksh 88) の構文に基づいており、入出力ダイレクション、ループ、組み込み算術演算、ヒストリ、コマンド行編集 (コマンド行モードのみで、dbx からは利用不可能) といった機能を持っています。この章では、ksh-88 と dbx コマンド言語の違いをまとめています。

dbx 初期化ファイルが起動時に見つからない場合、dbx は ksh モードを想定します。

この章の内容は次のとおりです。

- 251 ページの「実装されていない ksh-88 の機能」
- 252 ページの「ksh-88 から拡張された機能」
- 252 ページの「名前が変更されたコマンド」

実装されていない ksh-88 の機能

ksh-88 の次の機能は dbx では実装されていません。

- `set -A name` による配列 `name` への値の代入
- `set -o` の次のオプション: `allexport bgnice gmacs markdirs noclobber nolog privileged protected viraw`
- `typeset` の次の属性: `-l -u -L -R -H`
- バッククォート ('...') によるコマンドの置き換え (代わりに `$(...)` を使用)
- 複合コマンド `[[式]]` による式の評価
- `@(<パターン>[|<パターン>] ...)` による拡張パターン照合
- 子プロセス (バックグラウンドで動作し、プログラム交信するコマンドまたはパイプライン)

ksh-88 から拡張された機能

dbx では、次の機能が追加されました。

- 言語式 `$(p- > flags)`
- `typeset -q` (ユーザ一定義関数のための特殊な引用を可能にする)
- `cs` のような `history` および `alias` の引数
- `set +o path` (パス検索を無効にする)
- `0xabcd` (8 進数および 16 進数を示す C の構文)
- `bind` による `emacs` モードバインディングの変更
- `set -o hashall`
- `set -o ignore suspend`
- `print -e` および `read -e (-r (raw) の逆の働きをする)`
- `dbx` コマンドが組み込まれている

名前が変更されたコマンド

ksh コマンドとの衝突を避けるために `dbx` コマンドの一部の名前が変更されています。

- `dbx` の `print` コマンドはそのまま、`ksh` の `print` コマンドが `kprint` という名前に変更されました。
- `ksh` の `kill` コマンドが `dbx` の `kill` コマンドにマージされました。
- `alias` コマンドは、`dbx` 互換モードでないかぎり `ksh` の `alias` コマンドとして機能します。
- `address/format` は現在 `examine address/format` です。
- `/pattern` は現在 `search pattern` です。
- `?pattern` は現在 `bsearch pattern` です。

編集機能のキーバインドの変更

`bind` コマンドを使い、編集機能のキーバインドを変更できます。EMacs 風のエディタや `vi` 風のエディタのキーバインドを表示したり、変更したりすることができます。 `bind` コマンドの構文は次のとおりです。

<code>bind</code>	現在の編集機能のキーバインドを表示します。
<code>bindkey=definition</code>	<code>key</code> を <code>definition</code> にバインドします。
<code>bind key</code>	<code>key</code> の現在の定義を表示します。

<code>bind key=</code>	<code>key</code> をバインド解除します。
<code>bind -m key=definition</code>	<code>key</code> を <code>definition</code> のマクロとして定義します。
<code>bind -m</code>	<code>bind</code> と同じです。

ここで

`key` はキーの名前です。

`definition` はキーにバインドするマクロの定義です。

次は、Emacs 風のエディタ用の主なデフォルトのキーバインドを示しています。

<code>^A = beginning-of-line</code>	<code>^B = backward-char</code>
<code>^D = eot-or-delete</code>	<code>^E = end-of-line</code>
<code>^F = forward-char</code>	<code>^G = abort</code>
<code>^K = kill-to-eo</code>	<code>^L = redraw</code>
<code>^N = down-history</code>	<code>^P = up-history</code>
<code>^R = search-history</code>	<code>^^ = quote</code>
<code>^? = delete-char-backward</code>	<code>^H = delete-char-backward</code>
<code>^[b = backward-word</code>	<code>^[d = delete-word-forward</code>
<code>^[f = forward-word</code>	<code>^[^H = delete-word-backward</code>
<code>^[^[= complete</code>	<code>^[? = list-command</code>

次は、vi 風のエディタ用の主なデフォルトのキーバインドを示しています。

<code>a = append</code>	<code>A = append at EOL</code>
<code>c = change</code>	<code>d = delete</code>
<code>G = go to line</code>	<code>h = backward character</code>
<code>i = insert</code>	<code>I = insert at BOL</code>
<code>j = next line</code>	<code>k = previous line</code>
<code>l = forward line</code>	<code>n = next match</code>
<code>N = prev match</code>	<code>p = put after</code>

P = put before	r = repeat
R = replace	s = substitute
u = undo	x = delete character
X = delete previous character	y = yank
~ = transpose case	_ = last argument
* = expand	= = list expansion
- = previous line	+ = next line
sp = forward char	# = comment out command
? = search history from beginning	
/ = search history from current	

挿入モードでは、次のキーストロークが特別な働きをします。

^? = delete character	^H = delete character
^U = kill line	^W = delete word

共有ライブラリのデバッグ

dbx は動的にリンクされた共有ライブラリのデバッグを完全にサポートしています。ただし、これらのライブラリが `-g` オプションを使用してインストールされていることが前提になります。

この章の内容は次のとおりです。

- 255 ページの「動的リンカー」
- 256 ページの「修正と継続」
- 257 ページの「共有ライブラリにおけるブレークポイントの設定」
- 257 ページの「明示的に読み込まれたライブラリにブレークポイントを設定する」

動的リンカー

動的リンカーは `rtld`、`"実行時 ld"`、または `ld.so` と呼ばれ、実行中のアプリケーションに共有オブジェクト (ロードオブジェクト) を組み込むように準備します。rtld が稼働状態になるのは主に次の 2 つの場合です。

- プログラムの起動時 - プログラムの起動時、rtld はまずリンク時に指定されたすべての共有オブジェクトを動的に読み込みます。これらは「あらかじめ読み込まれた」共有オブジェクトで、一般に `libc.so`、`libC.so`、`libX.so` などがあります。ldd (1) を使用すれば、プログラムによって読み込まれる共有オブジェクトを調べることができます。
- アプリケーションから呼び出しがあった場合 - アプリケーションでは、関数呼び出し `dlopen(3)` と `dlclose(3)` を使用して共有オブジェクトやプレーンな実行可能ファイルの読み込みや読み込みの取り消しを行います。

共有オブジェクト (`.so`) や通常の実行可能ファイル (`a.out`) のことを、dbx では「ロードオブジェクト」といいます。loadobject コマンド (331 ページの「loadobject コマンド」参照) を使用して、loadobject からの記号情報をリストおよび管理できます。

リンクマップ

動的リンカーは、読み込んだすべてのオブジェクトのリストを、*link map* というリストで管理します。このリストは、デバッグするプログラムのメモリーに保存され、`librtld_db.so` で間接的にアクセスできます。これはデバッガ用に用意された特別なシステムライブラリです。

起動手順と .init セクション

A .init セクションは、共有オブジェクトの読み込み時に実行される、その共有オブジェクトのコードの一部です。たとえば、.init セクションは、C++ 実行時システムがすべての静的初期化関数を呼び出すときに使用します。

動的リンカーは最初にすべての共有オブジェクトにマップインし、それらのオブジェクトをリンクマップに登録します。そのあと、動的リンカーはリンクマップに含まれる各オブジェクトの .init セクションを順に実行します。syncrtld イベント (275 ページの「syncrtld」参照) は、これら 2 つの動作の間に発生します。

プロシージャリンケージテーブル

PLT は、共有オブジェクトの境界間の呼び出しを容易にするために `rtld` によって使用される構造体です。たとえば、`printf` の呼び出しはこの間接テーブルによって行います。その方法の詳細については、SVR4 ABI に関する汎用リファレンスマニュアルおよびプロセッサ固有のリファレンスマニュアルを参照してください。

複数の PLT 間で `step` コマンドと `next` コマンドを操作するために、`dbx` は各ロードオブジェクトの PLT テーブルを追跡する必要があります。テーブル情報は `rtld` ハンドシェイクと同時に入手されます。

修正と継続

`dlopen()` で読み込んだ共有オブジェクトに `fix` と `cont` を使用する場合、開き方を変更しないと `fix` と `cont` が正しく機能しません。モード `RTLD_NOW | RTLD_GLOBAL` または `RTLD_LAZY | RTLD_GLOBAL` を使用します。

共有ライブラリにおけるブレークポイントの設定

共有ライブラリにブレークポイントを設定する場合、dbx はプログラムの実行時にそのライブラリが使用されることを知っている必要があります。また、そのライブラリのシンボルテーブルを読み込む必要もあります。新しく読み込まれたプログラムが実行時に使用するライブラリを調べる際、dbx は実行時リンカーが起動時のライブラリのすべてを読み込むのに十分な時間を使い、プログラムを実行します。そして、読み込まれたライブラリのリストを読み取ってプロセスを終了します。このとき、ライブラリは読み込まれたままであるため、デバッグ対象としてプログラムを再実行する前にそれらのライブラリにブレークポイントを設定することができます。

dbx は、3つあるうちのどの方法(コマンド行から dbx コマンドを使用、dbx プロンプトで debug コマンドを使用、IDE で dbx デバッガを使用)でプログラムが読み込まれたかに関係なく、同じ手順に従ってライブラリを読み込みます。

明示的に読み込まれたライブラリにブレークポイントを設定する

dbx は `dlopen()` または `dlopen()` の発生を自動的に検出し、読み込まれたオブジェクトの記号テーブルを読み込みます。`dlopen()` で共有オブジェクトを読み込むと、そのオブジェクトにブレークポイントを設定できます。またプログラムのその他の任意の場所で行う場合と同様にデバッグも可能です。

共有オブジェクトを `dlopen()` で読み込み解除しても、dbx はそのオブジェクトに設定されていたブレークポイントを記憶しているため、たとえアプリケーションを再実行しても、共有オブジェクトが `dlopen()` で再び読み込まれれば再びそのブレークポイントを設定し直します。

ただし、`dlopen()` で共有オブジェクトが読み込まれるのを待たなくても共有オブジェクトにブレークポイントを設定したり、その関数やソースコードを検索することはできます。デバッグするプログラムが `dlopen()` で読み込む共有オブジェクトの名前がわかっている場合は、`loadobject -load` コマンドを使用してその記号テーブルをあらかじめ dbx に読み込んでおくことができます。

```
loadobject -load /usr/java1.1/lib/libjava_g.so
```

これで、`dlopen()` で読み込む前でも、この読み込みオブジェクト内でモジュールと関数を検索してその中にブレークポイントを設定できます。読み込みオブジェクトの読み込みが済んだら、dbx はブレークポイントを自動的に設定します。

動的にリンクしたライブラリにブレークポイントを設定する場合、次の制約があります。

- `dlopen()` で読み込んだ「フィルタ」ライブラリには、その中の最初の関数が呼び出されるまでブレークポイントは設定できません。
- `dlopen()` でライブラリを読み込むと、初期化ルーチン `_init()` が呼び出されます。このルーチンがライブラリ内のほかのルーチンを呼び出すこともあります。この初期化が終了するまで、`dbx` は読み込んだライブラリにブレークポイントを設定できません。具体的には、`dbx` は、`dlopen` で読み込んだライブラリ内の `_init()` では停止できません。

プログラム状態の変更

ここでは、dbx を使用しないでプログラムを実行する場合と比べながら、dbx で実行する際のプログラムまたはプログラムの動作を変更する dbx の使用法とコマンドについて説明します。プログラムに変更を加えるコマンドがどれかを理解する必要があります。

この付録は、次の各節から構成されています。

- 259 ページの「dbx 下でプログラムを実行することの影響」
- 260 ページの「プログラムの状態を変更するコマンドの使用」

dbx 下でプログラムを実行することの影響

dbx を使用してプロセスを監視しますが、監視によってプロセスが妨げられるべきではありません。しかし、時によって、プロセスの状態を大幅に変わる可能性があります。さらに、簡単な監視によって、実行が妨げられ、原因不明のバグ症状が現れることもときどきあります。

アプリケーションは、dbx のもとで実行される場合、本来と動作が異なることがあります。dbx は被デバッグプログラムに対する影響を最小限に抑えようとはしますが、次の点に注意する必要があります。

- `-c` オプション付きで起動しないでください。また、RTC は無効にしてください。RTC のライブラリの `librtc.so` をプログラムに読み込むと、プログラムの動作が変わる可能性があります。
- dbx 初期化スクリプトで環境変数が設定されていることを忘れないでください。スタックベースは、dbx のもとで実行する場合、異なるアドレスから始まります。これは、各自の環境と `argv[]` の内容によっても異なり、局所変数の割り当てが若干異なります。これらが初期化されていないと、異なる乱数を受け取ります。この問題は、実行時検査によって検出できます。
- プログラムは、使用前に `malloc()` されたメモリーを初期化しません。これは、前述の状態と似ています。この問題は、実行時検査によって検出できます。

- dbx は LWP 作成イベントと dlopen イベントを捕獲しなければならず、これによって、タイミングに左右されやすいマルチスレッドアプリケーションが影響を受ける可能性があります。
- dbx は、シグナルに対するコンテキスト切り替えを実行するため、タイミングに影響を受けるシグナルを多用する場合、動作が異なってしまうおそれがあります。
- プログラムは、mmap() が、マップされたセグメントについて常に同じベースアドレスを返すことを期待します。dbx のもとで実行すると、アドレス空間が混乱して、mmap() は、dbx を使用しないでプログラムを実行したときと同じアドレスを返せなくなります。プログラムでこのことが問題になるかどうかを判断するには、mmap() の使用場所をすべて調べて、返される値がハードコードされたアドレスではなく、プログラムによって実際に使用されることを確認してください。
- プログラムがマルチスレッド化されている場合、データの競合が存在するか、またはスレッドスケジュールに依存する可能性があります。dbx のもとで実行すると、スレッドスケジュールが混乱して、プログラムが通常の順序とは異なる順序でスレッドを実行するおそれがあります。このような状態を検出するには、lock lint を使用してください。

あるいは、adb または truss を使用して実行した場合に同じ問題が起こるか確認してください。

dbx によって強いらられる混乱を最小限に抑えるには、アプリケーションが自然な環境で実行されているときに dbx を接続するようにしてください。

プログラムの状態を変更するコマンドの使用

assign コマンド

assign コマンドは、*expression* の値を *variable* に割り当てます。dbx 内で使用すると *variable* の値が永久に変更されます。

```
assign variable = expression
```

pop コマンド

pop コマンドは、スタックから 1 つまたは複数のフレームをポップ (解放) します。

pop カレントフレームをポップ

pop *number* *number* 個のフレームをポップ

pop -f *number* 指定のフレーム数までフレームをポップ

ポップされた呼び出しはすべて、再開時に再び実行されて、プログラムに望ましくない変更が加えられる可能性があります。pop は、ポップされた関数にローカルなオブジェクトのデストラクタも呼び出します。

詳細については、[343 ページの「pop コマンド」](#)を参照してください。

call コマンド

call コマンドを dbx で使用すると、ある手続きが呼び出されて、その手続きは指定どおりに実行されます。

```
call proc([params])
```

この手続きは、プログラムの一部を変更する可能性があります。dbx は、プログラムソースに呼び出しを組み込んだ場合と同様に、実際に呼び出しを行います。

詳細については、[289 ページの「call コマンド」](#)を参照してください。

print コマンド

式の値を出力するには、次のように入力します。

```
print expression, ...
```

式に関数呼び出しがある場合は、[289 ページの「call コマンド」](#)と同じ考慮事項が適用されます。C++ では、多重定義演算子による予期しない副作用にも注意する必要があります。

詳細については、[344 ページの「print コマンド」](#)を参照してください。

when コマンド

when コマンドの一般的な構文は次のとおりです。

```
when event-specification [modifier] {command; ... }
```

イベントが発生すると、*command* が実行されます。

ある行または手続きに到達すると、コマンドが実行されます。どのコマンドを出したかによって、プログラムの状態が変わる可能性があります。

詳細については、[384 ページの「when コマンド」](#)を参照してください。

fix コマンド

fix コマンドを使用すると、プログラムに対して、実行中の変更を加えることができます。

fix

これは非常に便利なツールですが、fix は変更されたソースファイルを再コンパイルして、変更された関数をアプリケーションに動的にリンクすることに注意してください。

fix と cont の制限事項を必ず確認してください。161 ページの「メモリーリーク (mel) エラー」を参照してください。

詳細については、318 ページの「fix コマンド」を参照してください。

cont at コマンド

cont at コマンドは、プログラムが実行される順序を変更します。実行を *line.id* で指定した行で続けられます。プログラムがマルチスレッド化されている場合は ID が必要です。

cont at *line* [*id*]

これにより、プログラムの結果が変更される可能性があります。

イベント管理

イベント管理は、デバッグ中のプログラムで特定のイベントが発生したときに特定のアクションを実行する、dbxの一般的な機能です。dbxを使用すると、イベント発生時に、プロセスの停止、任意のコマンドの発行、または情報を表示することができます。イベントのもっとも簡単な例はブレークポイントです(94ページの「Control+Cによってプロセスを停止する」を参照)。その他のイベントの例として、障害、信号、システムコール、dlopen()の呼び出し、およびデータの変更などがあります(100ページの「データ変更ブレークポイントを設定する」を参照)。

この付録の内容は次のとおりです。

- 263ページの「イベントハンドラ」
- 264ページの「イベントの安全性」
- 265ページの「イベントハンドラの作成」
- 265ページの「イベントハンドラを操作するコマンド」
- 266ページの「イベントカウンタ」
- 266ページの「イベント指定の設定」
- 276ページの「イベント指定のための修飾子」
- 279ページの「解析とあいまいさに関する注意」
- 279ページの「事前定義済み変数」
- 282ページの「イベントハンドラの設定例」

イベントハンドラ

イベント管理は「ハンドラ」の概念に基づくもので、この名前はハードウェアの割り込みハンドラからきたものです。通常、ハンドラは各イベント管理コマンドによって作成されます。これらのコマンドは、「イベント指定」と関連する一連のアクションで構成されます(266ページの「イベント指定の設定」参照)。イベント指定は、ハンドラを発生させるイベントを指定します。

イベントが発生し、ハンドラが引き起こされると、イベント指定に含まれる任意の修飾子に従って、ハンドラはイベントを評価します(276ページの「イベント指定の

ための修飾子」参照)。修飾子によって課された条件にイベントが適合すると、ハンドラの関連アクションが実行されます(つまり、ハンドラが起動します)。

プログラムイベントを dbx アクションに対応付ける例は、特定の行にブレークポイントを設定するものです。

ハンドラを作成するもっとも一般的な形は、when コマンドを使用するものです。

```
when event-specification {action; ... }
```

この章の例は、when を使用した表現でコマンド(stop、step、ignoreなど)を記述する方法を示します。これらの例は、when とその配下にある「ハンドラ」機構の柔軟性を示すものですが、常に同じ働きをすることはできません。

イベントの安全性

dbx では、イベント機構によって、豊富な種類のブレークポイントが用意されていますが、内部でも多くのイベントが使用されています。これらの内部イベントのいくつかで停止することによって、dbx の内部の動作を簡単に中断することができます。さらに、これらの場合の処理状態を変更すると、中断できる機会が増えます。付録 A と 93 ページの「安全な呼び出し」を参照してください。

場合によっては、dbx は自身を保護して中断を妨げることがありますが、すべての場合ではありません。一部のイベントは下位レベルのイベントという観点で実装されています。たとえば、すべてのステップ実行は fault FLTRACE イベントに基づきます。そのため、コマンド stop fault FLTRACE を発行すると、ステップ実行が停止します。

デバッグに続く段階では、dbx はユーザーイベントを処理できません。これは、ユーザーイベントにより精密な内部統合が妨げられるからです。このような段階には次のものが含まれます。

- プログラムの起動時に rtdld が実行された場合 (255 ページの「動的リンカー」を参照)
- プロセスの開始と終了時
- fork() () 関数と exec() () 関数のあと (178 ページの「fork 機能後のプロセス追跡」と 178 ページの「exec 機能後のプロセス追跡」を参照)
- dbx がユーザープロセスのヘッドを初期化する必要がある場合の呼び出し時 (proc_heap_init())
- dbx がスタックのマッピングされたページを確実に利用できるようにする場合の呼び出し時 (ensure_stack_memory())

多くの場合、stop コマンドの代わりに when コマンドを使用して、情報を表示することができます。このコマンドを使用しない場合は、対話によって情報を取得する必要があります。

dbx は次のようにして自身を保護します。

- `sync`、`syncrtld`、および `prog_new` イベントに `stop` コマンドを許可しない
- `rtld` ハンドシェイク時および前述の段階で `stop` コマンドを無視する

たとえば、次のようにします。

```
stopped in munmap at 0xff3d503c 0xff3d503c: munmap+0x0004: ta %icc,0x00000008
dbx76: warning: 'stop' ignored -- while doing rtld handshake
```

`$firedhandlers` 変数での記録を含む停止効果のみが無視されます。カウントやフィルタはアクティブなままになります。このような場合で停止させるには、`event_safety` 環境変数を `off` に設定します。

イベントハンドラの作成

イベントハンドラを作成するには、`when`、`stop`、`trace` の各コマンドを使用します (詳細については、[384 ページの「when コマンド」](#)、[360 ページの「stop コマンド」](#)、および [372 ページの「trace コマンド」](#) を参照)。

共通の `when` 構文は、`stop` を使用して簡単に表現できます。

```
when event-specification { stop -update; whereami; }
```

`event-specification` は、イベント管理コマンド `stop`、`when`、`trace` にて使用され、関心のあるイベントを指定します ([266 ページの「イベント指定の設定」](#) 参照)。

`trace` コマンドのほとんどは、`when` コマンド、`ksh` 機能、イベント変数を使用して手動で作成することができます。これは、スタイル化されたトレーシング出力を希望する場合、特に有益です。

コマンドが実行される度に、ハンドラ `id` (`hid`) 番号を返します。事前定義変数 `$newhandlerid` を介してこの番号にアクセスすることができます。

イベントハンドラを操作するコマンド

次のコマンドを使用して、イベントハンドラを操作することができます。各コマンドの詳細については、それぞれの節を参照してください。

- `status` - ハンドラを表示します ([357 ページの「status コマンド」](#) 参照)。
- `delete` - 一時ハンドラを含むすべてのハンドラを削除します ([310 ページの「delete コマンド」](#) 参照)。
- `clear` - ブレークポイントの位置に基づいてハンドラを削除します ([294 ページの「clear コマンド」](#) 参照)。
- `handler -enable` - ハンドラを有効にします ([323 ページの「handler コマンド」](#) 参照)。

- `handler -disable` - ハンドラを無効にします。
- `cancel` - 信号を取り消し、プロセスを継続させます (291 ページの「`cancel` コマンド」参照)。

イベントカウンタ

イベントハンドラはカウンタを備えており、制限値と実際のカウンタを保持します。イベントが発生するたびにカウンタをインクリメント (1つ増加) し、その値が制限値に達すると、ハンドラに対応するアクションを起動してカウンタをゼロにリセットします。デフォルトの制限値は1です。プロセスが再実行されるたびに、すべてのイベントカウンタがリセットされます。

`stop`、`when`、`trace` コマンドで `-count` 修飾子を使用して、カウント制限を設定することができます (277 ページの「`-count n -count infinity`」参照)。このほか、`handler` コマンドを使用して、個々のイベントハンドラを操作できます。

```
handler [ -count | -reset ] hid new-count new-count-limit
```

イベント指定の設定

イベント指定は、`stop`、`stopi`、`when`、`wheni`、`trace`、`tracei` コマンドで、イベントタイプとパラメータを表すために使用します。その書式は、イベントタイプを表すキーワードと省略可能なパラメータで構成されます。指定子の意味は、一般的にすべてのコマンドで同一です。例外については、コマンドの説明 (360 ページの「`stop` コマンド」、372 ページの「`trace` コマンド」、384 ページの「`when` コマンド」を参照) に記載されています。

ブレークポイントイベント仕様

ブレークポイントとは、アクションが発生する位置であり、その位置でプログラムは実行を停止します。次に、ブレークポイントイベントに対するイベント仕様を説明します。

in function

関数が入力され、最初の行が実行される直前。先行ログ後の最初の実行可能コードは、実際のブレークポイントの位置として使用されます。この行は、局所変数を初期化する行になることがあります。C++ のコンストラクターの場合、すべてのベースクラスのコンストラクターの実行後に実行されます。`-instr` 修飾子が使用される場合 (278 ページの「`-instr`」参照) は、関数の最初の命令が実行される直前です。*function* 仕様は、仮パラメータを含むことができるため、多重定義関数名、またはテンプレートインスタンスの指定に役立ちます。たとえば、次のようにします。

```
stop in mumble(int, float, struct Node *)
```

注 - *infunction* と *-infunction* 修飾子とを混同しないでください。

at [*filename:*]*line_number*

指定の行が実行される直前。*filename* を指定した場合は、指定ファイルの指定の行が実行される直前。ファイル名には、ソースファイル名またはオブジェクトファイル名を指定します。引用符は不要ですが、ファイル名に特殊文字が含まれる場合は、必要な場合もあります。指定の行がテンプレートコードに含まれる場合、ブレークポイントは、そのテンプレートのすべてのインスタンス上に置かれます。

at *address_expression*

指定のアドレスの指示が実行される直前。このイベントは `stopi` コマンド (365 ページの「[stopi コマンド](#)」参照) または `-instr` イベント修飾子 (278 ページの「[-instr](#)」参照) のみ利用可能です。

infunction *function*

function と名付けられたすべての多重定義関数、およびテンプレートインスタンスのすべてに対し、*infunction* と同じ働きをします。

inmember *function* inmethod *function*

すべてのクラスの *function* と名付けられたメンバー関数に対し、*infunction* と同じ働きをします。

inclass *classname* [-recurse | -norecurse]

classname のベースではなく、*classname* のメンバーであるすべてのメンバー関数に対し、*infunction* と同じ働きをします。`-norecurse` はデフォルトです。`-recurse` が指定された場合、基底クラスが含まれます。

inobject *object-expression* [-recurse | -norecurse]

object-expression に指定されているアドレスのオブジェクトを呼び出したメンバー関数が呼び出されているとき。`stop inobject ox` は次のコードとほとんど同じ働きをしますが、`inclass` と異なり、動的な *ox* のベースが含まれます。`-recurse` はデフォルトです。`-norecurse` が指定された場合、基底クラスが含まれます。

```
stop inclass dynamic_type(ox) -if this==ox
```

データ変更イベント指定

メモリアドレスへのアクセスまたは変更が必要なイベントのイベント指定の例を示します。

access mode address-expression [, byte-size-expression]

address-expression で指定されたメモリーがアクセスされたとき。

mode はメモリーのアクセス方法を指定します。次の文字 (複数可) で構成されます。

- r 指定したアドレスのメモリーが読み取られたことを示します。
- w メモリーへの書き込みが実行されたことを示します。
- x メモリーが実行されたことを示します。

さらに *mode* には、次のいずれかの文字も指定することができます。

- a アクセス後にプロセスを停止します (デフォルト)。
- b アクセス前にプロセスを停止します。

いずれの場合も、プログラムカウンタは副作用アクションの前後で違反している命令をポイントします。「前」と「後」は副作用を指しています。

address-expression は、その評価によりアドレスを生成できる任意の式です。シンボル式を使用すると、監視される領域のサイズが自動的に推定されます。このサイズは、*byte-size-expression* を指定することにより、上書されます。シンボルを使用しない、型を持たないアドレス式を使用することもできますが、その場合はサイズを指定する必要があります。たとえば、次のようにします。

```
stop access w 0x5678, sizeof(Complex)
```

access コマンドには、2つの一致する範囲が重複しない、という制限があります。

注 - *access* イベント仕様は、*modify* イベント仕様の代替です。

change *variable*

variable の値は変更されました。change イベントは、次のコードとほとんど同じ働きをします。

```
when step { if [ $last_value !=${variable}] then
    stop
    else
        last_value=${variable}
    }
```

このイベントはシングルステップを使用して実装されます。パフォーマンス速度を上げるには、`access` イベント (268 ページの「[access mode address-expression \[, byte-size-expression \]](#)」参照) を使用します。

最初に `variable` がチェックされると、変更が検出されない場合でも1つのイベントが発生します。この最初のイベントによって `variable` の最初の値にアクセスできるようになります。あとから検出された `variable` の値への変更によって別のイベントが発生します。

cond condition-expression

condition-expression によって示される条件が真と評価されます。*condition-expression* には任意の式を使用できますが、整数型に評価されなければなりません。`cond` イベントは、次のコードとほとんど同じ働きをします。

```
stop step -if conditional_expression
```

システムイベント指定

次に、システムイベントに対するイベント指定について説明します。

`dlopen [lib-path] | dlclose [lib-path]`

これらのイベントは、`dlopen()` または `dlclose()` の呼び出しが正常終了したあとに発生します。`dlopen()` または `dlclose()` の呼び出しにより、複数のライブラリが読み込まれることがあります。これらのライブラリのリストは、事前定義済み変数 `$dllib` で常に入手できます。`$dllib` 中の最初のシェルの単語は実際には「+」または「-」で、それぞれライブラリが追加されているか、削除されているかを示します。

lib-path は、該当する共有ライブラリの名前です。これを指定した場合、そのライブラリが読み込まれたり、読み込みが取り消されたりした場合にだけイベントが起動します。その場合、`$dlobj` にライブラリの名前が格納されます。また、`$dllib` も利用できます。

lib-path が / で始まる場合は、パス名全体が比較されます。それ以外の場合は、パス名のベースだけが比較されます。

lib-path を指定しない場合、イベントは任意の `dl` 動作があるときに必ず起動します。`$dlobj` は空になりますが、`$dllib` は有効です。

fault *fault*

fault イベントは、指定の障害に遭遇したとき、発生します。障害は、アーキテクチャ依存です。dbx に対して知られる次の一連の障害は、proc(4) マニュアルページで定義されています。

障害	内容の説明
FLTILL	不正命令
FLTPRIV	特権付き命令
FLTBPT*	ブレークポイントトラップ
FLTRACE*	トレーストラップ(ステップ実行)
FLTACCESS	メモリアクセス(境界合わせなど)
FLTBOUNDS	メモリー境界(無効なアドレス)
FLTIOVF	整数オーバーフロー
FLTIZDIV	整数ゼロ除算
FLTPE	浮動小数点例外
FLTSTACK	修復不可能なスタックフォルト
FLTPAGE	修復可能なページフォルト
FLTWATCH*	ウォッチポイントトラップ
FLTCPCOVF	CPU パフォーマンスカウンタオーバーフロー

注 - BPT、TRACE、BOUNDS は、ブレークポイントとステップ実行を実現するため、dbx で使用されます。これらを実行すると、dbx の動作に影響を及ぼす場合があります。

注 - FLTBPT および FLTRACE は、ブレークポイントやシングルステップ実行などの dbx の基本機能を妨げることがあるため、無視されます(264 ページの「イベントの安全性」を参照)。

これらの障害は、/sys/fault.h から抜粋されています。fault には前述の名前を大文字または小文字で指定できるほか、実際のコードも指定できます。また、コードの名前には、接頭辞 FLT- を付けることがあります。

注 - `fault` イベントは、Linux プラットフォームでは使用できません。

`lwp_exit`

`lwp_exit` イベントは、`lwp` が終了したとき、発生します。`$lwp` には、イベントハンドラを維持している間に終了した LWP (軽量プロセス) の id が含まれます。

注 - `lwpexit` イベントは、Linux プラットフォームでは使用できません。

`sig signal`

`sig signal` イベントは、デバッグ中のプログラムに信号が初めて送られたとき、発生します。`signal` は、10 進数、または大文字、小文字の信号名のいずれかです。接頭辞は任意です。このイベントは、`catch` および `ignore` コマンドからは完全に独立しています。ただし、`catch` コマンドは次のように実現することができます。

```
function simple_catch {
    when sig $1 {
        stop;
        echo Stopped due to $sigstr $sig
        whereami
    }
}
```

注 - `sig` イベントを受け取った時点では、プロセスはまだそれを見ることができません。指定の信号を持つプロセスを継続する場合のみ、その信号が転送されます。

`sig signal sub-code`

指定の `sub-code` を持つ指定の信号が `child` に初めて送られたとき、`sig signal sub-code` イベントが発生します。信号同様、`sub-code` は 10 進数として、大文字または小文字で入力することができます。接頭辞は任意です。

`sysin code | name`

指定されたシステムコールが起動された直後で、プロセスがカーネルモードに入ったとき。

`dbx` の認識するシステムコールは `procfs(4)` の認識するものに限られます。これらのシステムコールはカーネルでトラップされ、`/usr/include/sys/syscall.h` に列挙されます。

これは、ABIの言うところのシステムコールとは違います。ABIのシステムコールの一部は部分的にユーザーモードで実装され、非ABIのカーネルトラップを使用します。ただし、一般的なシステムコールのほとんど(シグナル関係は除く)は `syscall.h` とABIで共通です。

注 - `sysin` イベントは、Linux プラットフォームでは使用できません。

注 - `/usr/include/sys/syscall.h` 内のカーネルシステムコールトラップのリストは、Solaris OS のプライベートインタフェースの一部です。これはリリースによって異なります。dbx が受け付けるトラップ名(コード)およびトラップ番号のリストは、dbx がサポートするバージョンの Solaris OS によってサポートされているすべてを含みます。dbx によってサポートされている名前が特定のリリースの Solaris OS でサポートされている名前と性格に一致することはありえないため、`syscall.h` 内のいくつかの名前は利用可能でない場合があります。すべてのトラップ番号(コード)は dbx で受け入れられ、予測どおりに動作しますが、既知のシステムコールトラップに対応しない場合は、警告が発行されます。

`sysout code | name`

指定されたシステムコールが終了し、プロセスがユーザーモードに戻る直前。

注 - `sysout` イベントは、Linux プラットフォームでは使用できません。

`sysin | sysout`

引数がないときは、すべてのシステムコールがトレースされます。ここで、`modify` イベントや RTC (実行時検査) などの特定の dbx は、子プロセスにその目的でシステムコールを引き起こすことがあることに注意してください。トレースした場合にそのシステムコールの内容が示されることがあります。

実行進行状況イベント仕様

次に、実行進行状況に関するイベントのイベント仕様について説明します。

`exit exitcode`

`exit` イベントは、プロセスが終了したときに発生します。

`next`

`next` イベントは、関数がステップされないことを除いては、`step` イベントと同様です。

returns

このイベントは、現在表示されている関数の戻りのブレークポイントです。表示されている関数を使用するのは、いくつかの `up` を行なったあとに `returns` イベント指定を使用できるようにするためです。通常の `returns` イベントは常に一時イベント (`-temp`) で、動作中のプロセスが存在する場合にだけ作成できます。

returns function

特定の関数がある呼び出し場所に戻るたびに発生します。これは一時イベントではありません。戻り値は示されませんが、SPARC プラットフォームでは `$o0`、Intel プラットフォームでは `$eax` を使用して、必須戻り値を調べることができます。

SPARC システム	<code>\$o0</code>
x86 システム	<code>\$eax</code>
x64 システム	<code>\$rax, \$rdx</code>

このイベントは、次のコードとほとんど同じ働きをします。

```
when in func { stop returns; }
```

step

`step` イベントは、ソース行の先頭の命令が実行されると発生します。たとえば、次のようにシンプルに表現することができます。

```
when step { echo $lineno: $line; }; cont
```

`step` イベントを有効にするということは、次に `cont` コマンドが使用されるときに自動的にステップ実行できるように `dbx` に命令することと同じです。

注 `-step` (および `next`) イベントは一般的なステップコマンド終了時に発生しません。`step` コマンドは `step` イベントで次のように実装されます。`alias step="when step -temp { whereami; stop; }; cont"`

その他のイベント仕様

次に、その他のタイプのイベントに対するイベント仕様を説明します。

attach

`dbx` がプロセスを正常に接続した直後。

detach

dbx がプロセスを切り離す直前。

lastrites

デバッグ中のプロセスが終了しようとしています。これは次の理由によって発生します。

- システムコール `_exit(2)` が呼び出し中 (これは、明示的に呼び出されたとき、または `main()` のリターン時に発生します)。
- 終了シグナルが送信されようとするとき。
- dbx コマンド `kill` によってプロセスが強制終了されつつあるとき。

プロセスの最終段階は、必ずではありませんが通常はこのイベントが発生したときに利用可能になり、プロセスの状態を確認することができます。このイベントのあとにプログラムの実行を再開すると、プロセスは終了します。

注 - lastrites イベントは、Linux プラットフォームでは使用できません。

proc_gone

dbx がデバッグ中のプロセスと関連しなくなるとき。事前定義済み変数 `$reason` に、`signal`、`exit`、`kill`、または `detach` のいずれかが設定されます。

prog_new

`follow exec` の結果、新規のプログラムがロードされると、`prog_new` イベントが発生します。

注 - このイベントのハンドラは常に存在しています。

stop

プロセスが停止したとき。特に `stop` ハンドラによりユーザーがプロンプトを受け取るようにプロセスが停止すると、このイベントが起動します。次に例を示します。

```
display x
when stop {print x;}
```

sync

デバッグ対象のプロセスが `exec()` で実行された直後。 `a.out` で指定されたメモリはすべて有効で存在しますが、あらかじめ読み込まれるべき共有ライブラリはまだ読み込まれていません。たとえば `printf` は `dbx` に認識されていますが、まだメモリにはマップされていません。

`stop` コマンドにこのイベントを指定しても期待した結果は得られません。 `when` コマンドに指定してください。

注 - `sync` イベントは、Linux プラットフォームでは使用できません。

syncrtld

このイベントは、`sync` (被デバッグ側が共有ライブラリをまだ処理していない場合は `attach`) のあとに発生します。すなわち、動的リンカーの起動時コードが実行され、あらかじめ読み込まれている共有ライブラリすべてのシンボルテーブルが読み込まれたあと、ただし、`.init` セクション内のコードがすべて実行される前に発生します。

`stop` コマンドにこのイベントを指定しても期待した結果は得られません。 `when` コマンドに指定してください。

thr_create [*thread_id*]

`thr_create` イベントは、スレッドまたは `thread_id` の指定されたスレッドが作成されたときに発生します。たとえば、次の `stop` コマンドでスレッド ID `t@1` はスレッド作成を示しますが、スレッド ID `t@5` は作成済みスレッドを示しています。

```
stop thr_create t@5 -thread t@1
```

thr_exit

`thr_exit` イベントは、スレッドが終了したときに発生します。指定したスレッドの終了を取り込むには、次のように `stop` コマンドで `-thread` オプションを使用します。

```
stop thr_exit -thread t@5
```

throw

処理されない、または予期されない例外がアプリケーションから投げ出されると、`throw` イベントが発生します。

注 - `throw` イベントは、Linux プラットフォームでは使用できません。

`throw type`

例外 `type` が `throw` イベントで指定されると、そのタイプの例外のみが `throw` イベントを発生させます。

`throw -unhandled`

`-unhandled` は、投げ出されたが、それに対するハンドラがない例外を示す、特別な例外タイプです。

`throw -unexpected`

`-unexpected` は、それを投げ出した関数の例外仕様を満たさない例外を示す、特別な例外タイプです。

`timer seconds`

デバッグ中のプログラムが `seconds` 間実行されると、`timer` イベントが発生します。このイベントで使用されるタイマーは、`collector` コマンドで共有されます。解像度はミリ秒であるため、秒の浮動小数点値 (`0.001` など) が使用可能です。

イベント指定のための修飾子

イベント指定のため修飾子は、ハンドラの追加属性を設定します。もっとも一般的な種類はイベントフィルタです。修飾子はイベント指定のキーワードのあとに指定しなければなりません。修飾語はすべて `'.'` で始まります (その前に空白が置かれます)。各修飾子の構成は次のとおりです。

`-if condition`

`event-spec` で指定されたイベントが発生したとき、条件が評価されます。イベントは、条件が非ゼロと評価された場合にだけ発生すると考えられます。

`-if` が、`in` または `at` などの単独のソース位置に基づくイベントで使用された場合、`cond` はその位置に対応するスコープで評価されます。そうでない場合は、必要なスコープによって正しく修飾する必要があります。

- resumeone

-resumeone 修飾子は、-if 修飾子とともにイベント仕様内でマルチスレッドプログラムに対して使用して、条件に関数呼び出しが含まれている場合に1つのスレッドのみを再開することができます。詳細については、104 ページの「条件付イベントでのフィルタの使用」を参照してください。

- in function

イベントは、最初の指定 *function* の命令に達したときから関数が戻るまでの間に発生した場合にのみ開始されます。関数の再帰は無視されます。

-disable

無効な状態にしてイベントを作成します。-

-count *n* -count infinity

-count *n* および -count infinity 修飾子は、0 からのハンドラカウントを持ちます (266 ページの「イベントカウンタ」参照)。イベントが発生するたび、*n* に達するまでカウントはインクリメントします。一度それが生じると、ハンドラはファイアし、カウンタはゼロにリセットされます。

プログラムが実行または再実行されると、すべてのイベントのカウントがリセットされます。より具体的に言えば、カウントは sync イベントが発生するとリセットされます。

カウントは debug -r コマンド (307 ページの「debug コマンド」参照) または attach -r コマンド (288 ページの「attach コマンド」参照) を使用して新しいプログラムのデバッグを開始したときにリセットされます。

-temp

一時ハンドラを作成します。イベントが発生すると、一時イベントは削除されます。デフォルトではハンドラは、一時イベントではありません。ハンドラが計数ハンドラ (-count が指定されたイベント) の場合はゼロに達すると自動的に破棄されません。

一時ハンドラをすべて削除するには delete -temp を実行します。

-instr

イベントを命令レベルで動作させます。これにより、ほとんどの'i'で始まるコマンドは不要となります。この修飾子は、イベントハンドラの2つの面を修飾します。

- 出力されるどのメッセージもソースレベルの情報ではなく、アセンブリレベルを示す。
- イベントの細分性が命令レベルになる。たとえば `step -instr` は、命令レベルのステップ実行を意味する。

-thread *thread_id*

イベントを引き起こしたスレッドが *thread_id* と一致する場合にかぎり、アクションが実行されます。プログラムの実行を繰り返すうちに特定スレッドの *thread_id* が変わってしまうことがあります。

-lwp *lwp_id*

イベントを引き起こしたスレッドが *lwp_id* と一致する場合にかぎり、アクションが実行されます。イベントを引き起こしたスレッドが *lwp_id* と一致する場合にかぎり、アクションが実行されます。プログラムの実行を繰り返すうちに特定スレッドの *lwp_id* が変わってしまうことがあります。

-hidden

ハンドラが正規の `status` コマンドに示されないようにします。隠されたハンドラを表示するには、`status -h` を使用してください。

-perm

通常、すべてのハンドラは、新しいプログラムが読み込まれると廃棄されます。`-perm` 修飾子を使用すると、ハンドラはデバッグセッションが終わっても保存されます。`delete` コマンド単独では、永続ハンドラは削除されません。永続ハンドラを削除するには、`delete -p` を使用してください。

解析とあいまいさに関する注意

イベント指定と修飾子のための構文の特徴は次のとおりです。

- キーワード駆動型である。
- 主に、空白によって区切られた「単語」に分割される点など、すべて ksh の規約に基づいている。

下位互換性のため、式の中には空白を含むことができます。そのため、式の内容があいまいになることがあります。たとえば、次の2つのコマンドがあるとします。

```
when a -temp
when a-temp
```

前述の例では、アプリケーションで *temp* という名前の変数が使用されていても、dbx は *-temp* を修飾子としてイベント指定を解釈します。下の例では、*a-temp* がまとめて言語固有の式解析プログラムに渡され、*a* および *temp* という変数が存在しなければ、エラーになります。オプションを括弧で囲むことにより、解析を強制できます。

事前定義済み変数

読み取り専用の ksh 事前定義済み変数がいくつか用意されています。次に示す変数は常に有効です。

変数	定義
<code>\$ins</code>	現在の命令の逆アセンブル
<code>\$lineno</code>	現在の行番号 (10 進数)
<code>\$vlineno</code>	現在の表示行番号 (10 進数)
<code>\$line</code>	現在の行の内容
<code>\$func</code>	現在の関数の名前
<code>\$vfunc</code>	現在の表示関数の名前
<code>\$class</code>	<code>\$func</code> が所属するクラスの名前
<code>\$vclass</code>	<code>\$vfunc</code> が所属するクラスの名前
<code>\$file</code>	現在のファイルの名前
<code>\$vfile</code>	現在表示しているファイルの名前

変数	定義
<code>\$loadobj</code>	現在のロードオブジェクトの名前
<code>\$vloadobj</code>	現在表示している現在のロードオブジェクトの名前
<code>\$scope</code>	逆引用符表記での現在の PC のスコープ
<code>\$vscope</code>	現在表示している逆引用符表記での PC のスコープ
<code>\$funcaddr</code>	<code>\$func</code> のアドレス (16 進数)
<code>\$caller</code>	<code>\$func</code> を呼び出している関数の名前
<code>\$dllist</code>	<code>dlopen</code> イベントまたは <code>dlclose</code> イベントのあと、 <code>dlopen</code> または <code>dlclose</code> された直後のロードオブジェクトのリストが格納されます。 <code>\$dllist</code> の中の先頭の単語は実際には「+」または「-」です。これは、 <code>dlopen</code> と <code>dlclose</code> のどちらが発生したかを示します。
<code>\$newhandlerid</code>	最後に作成されたハンドラの ID。この変数は、ハンドラを削除するコマンドのあとの未定義の値です。ハンドラを作成した直後に変数を使用します。 <code>dbx</code> では、複数のハンドラを作成する 1 つのコマンドに対してすべてのハンドラ ID を取り込むことはできません。
<code>\$firedhandlers</code>	停止の原因となった最近のハンドラ ID のリストです。リストにあるハンドラには、 <code>status</code> コマンドの出力時に「*」が付きます。
<code>\$proc</code>	現在デバッグ中のプロセスの ID
<code>\$lwp</code>	現在の LWP の ID
<code>\$thread</code>	現在のスレッドの ID
<code>\$newlwp</code>	新しく作成した LWP の <code>lwp</code> ID
<code>\$newthread</code>	新しく作成したスレッドのスレッド ID
<code>\$prog</code>	デバッグ中のプログラムの絶対パス名
<code>\$oprog</code>	<code>\$prog</code> の前の値は、 <code>\$prog</code> が「-」に戻るときに <code>exec()</code> に続いて、デバッグしていたものに戻る場合に便利です。 <code>\$prog</code> がフルパス名に展開され、 <code>\$oprog</code> がコマンド行または <code>debug</code> コマンドに指定されているプログラムパスを含みます。 <code>exec()</code> が 2 回以上呼び出されると、オリジナルのプログラムには戻りません。
<code>\$exec32</code>	<code>dbx</code> バイナリが 32 ビットの場合は <code>true</code> です。
<code>\$exitcode</code>	プログラムの最後の実行状態を終了します。この値は、プロセスが実際には終了していない場合、空文字列になります。

たとえば、`whereami` は次のように実装できます。

```
function whereami {
    echo Stopped in $func at line $lineno in file $(basename $file)
```

```
echo "${lineno}\t$line"
}
```

when コマンドに対して有効な変数

次の変数は、when コマンドの本体内でのみ有効です。

\$handlerid

本体の実行中、\$handlerid にはそれが属する when コマンドの ID が格納されます。次のコマンドは同じ結果になります。

```
when X -temp { do_stuff; }
when X { do_stuff; delete $handlerid; }
```

\$booting

イベントがブートプロセス中に起こると、true(真)に設定されます。新しいプログラムは、デバッグされるたびに、まず共有ライブラリのリストと位置を確認できるよう、ユーザーに通知されないまま実行されます。プロセスはそのあと終了します。ブートはこのようなシーケンスで行われます。

ブートが起こっても、イベントはすべて使用可能です。この変数は、デバッグ中に起こる sync および syncrtld のイベントと、通常の実行中に起こるこれらのイベントを区別するとき使用してください。

イベント別の有効変数

次の表で指定されている、特定のイベントの場合のみ有効な変数があります。

表 B-1 sig イベントに固有の変数

変数	説明
\$sig	イベントを発生させたシグナル番号
\$sigstr	\$sig の名前
\$sigcode	適用可能な場合、\$sig のサブコード
\$sigcodestr	\$sigcode の名前
\$sigsender	必要であれば、シグナルの送信者のプロセス ID

表B-2 exit イベントに固有の変数

変数	説明
\$exitcode	_exit(2) または exit(3) に渡された引数の値、または main の戻り値

表B-3 dlopen および dlclose イベントに固有の変数

変数	説明
\$dlobj	dlopen または dlclose されたロードオブジェクトのパス名

表B-4 sysin および sysout イベントに固有の変数

変数	説明
\$syscode	システムコールの番号
\$sysname	システムコールの名前

表B-5 proc_gone イベントに固有の変数

変数	説明
\$reason	シグナル、終了、強制終了、または切り離しのいずれか。

表B-6 thr_create イベントに固有の変数

変数	説明
\$newthread	新しく作成されるスレッドの ID (t@5 など)
\$newlwp	新しく作成される LWP の ID (l@4 など)

イベントハンドラの設定例

次に、イベントハンドラの設定例をあげます。

配列メンバーへのストアに対するブレークポイントを設定する

array[99] でブレークポイントを設定するには、次のように入力します。

```
(dbx) stop access w &array[99]
(2) stop access w &array[99], 4
(dbx) run
実行中: watch.x2
```

```
ウォッチポイント array[99] (0x20b68[4]) 行番号 12 ファイル "watch.c"
22   array[i] = i;
```

単純なトレースを実行する

単純なトレースの例:

```
(dbx) when step { echo at line $lineno; }
```

関数の中だけハンドラを有効にする (*in function*)

たとえば、

```
<dbx> trace step -in foo
```

は、次のようなスクリプトと等価です。

```
# 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"; };
}
```

実行された行の数を調べる

小規模なプログラムで何行実行されたかを調べます。

```
(dbx) stop step -count infinity # ステップ実行し、count=inf (関数が
無限大) になったところで停止する
(2) stop step -count 0/infinity
(dbx) run
...
(dbx) status
(2) stop step -count 133/infinity
```

ここでは、プログラムを停止させているのではなく、明らかにプログラムが終了しています。133は実行された行数です。ただし、このプロセスは非常に低速です。この方法が有効なのは、何度も呼び出される関数にブレークポイントを設定している場合です。

実行された命令の数をソース行で調べる

特定の行で実行された命令の数を数えます。

```
(dbx) ... # 調べたい行まで移動する
(dbx) stop step -instr -count infinity
(dbx) step ...
(dbx) status
(3) stop step -count 48/infinity # 48 個の命令が実行された
```

ステップ実行している行で関数呼び出しが行われる場合、最終的にそれらの呼び出しもカウントされます。step イベントの代わりに next イベントを使用すれば、そのような呼び出しはカウントされません。

イベント発生後にブレークポイントを有効にする

別のイベントが発生した場合のみ、ブレークポイントを有効にします。たとえば、プログラムで関数 hash が 1300 番目のシンボル検索以後に正しく動作しなくなるとします。次のように入力します。

```
(dbx) when in lookup -count 1300 {
    stop in hash
    hash_bpt=$newhandlerid
    when proc_gone -temp { delete $hash_bpt; }
}
```

注 - \$newhandlerid が、実行された直後の stop in コマンドを参照している点に注意してください。

replay 時にアプリケーションファイルをリセットする

アプリケーションが処理するファイルを replay 中にリセットする必要がある場合、プログラムを実行するたびに自動的にリセットを行うハンドラを書くことができます。

```
(dbx) when sync { sh regen ./database; }
(dbx) run < ./database... # この間にデータベースファイルが壊れた場合
(dbx) save
... # irun が自動的に行われ、sync イベントが
(dbx) restore # 発生し、regen が実行される。
```

プログラムの状態を調べる

プログラムの実行中にその状態をすばやく調べます。

```
(dbx) ignore sigint
(dbx) when sig sigint { where; cancel; }
```

プログラムを停止しないでそのスタックトレースを調べるためには、ここで`^c`を押します。

コレクタはこれ以外のことも実行できますが、基本的にコレクタの手動標本収集モードが実行する機能は、このように状態を調べます。ここではすでに`^c`を使用したため、プログラムに割り込むには`SIGQUIT (^)`を使用します。

浮動小数点例外を捕捉する

特定の浮動小数点例外を捕捉します。ここでは、IEEE オーバーフローだけを捕捉しています。

```
(dbx) ignore FPE # デフォルトのハンドラをオフにする
(dbx) help signals | grep FPE # サブコードの名前を思い出せない
...
(dbx) stop sig fpe FPE_FLTUND
...
```


コマンドリファレンス

この付録では、dbx コマンドの構文と機能について詳しく説明します。

assign コマンド

ネイティブモードでは、`assign` コマンドは新しい値をプログラムの変数に代入します。Java モードでは、`assign` コマンドは新しい値を局所変数またはパラメータに代入します。

ネイティブモードの構文

```
assign variable = expression
```

ここで

expression は、*variable* に代入される値です。

Java モードの構文

```
assign identifier = expression
```

ここで

class_name は、Java クラス名で、パッケージのパス (". "(ピリオド) を修飾子として使用。たとえば `test1.extra.T1.Inner`) またはフルパス名 (# 記号で始まり、"/" (スラッシュ) や \$ 記号を修飾子として使用。たとえば `#test1/extra/T1$Inner`) のいずれかで指定します。修飾子 \$ を使用する場合は、*class_name* を引用符で囲みます。

expression は、有効な Java の式です。

field_name は、クラス内のフィールド名です。

identifier は、局所変数またはパラメータです。これには、`this`、現在のクラスのインスタンス変数 (*object_name.field_name*)、クラス (static) 変数 (*class_name.field_name*) が含まれます。

object_name は、Java オブジェクトの名前です。

attach コマンド

`attach` コマンドは実行中プロセスに `dbx` を接続し、実行を停止してプログラムをデバッグ制御下に入れます。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

`attach process_id`

プロセス ID *process_id* を持つプログラムのデバッグを開始します。dbx は、`/proc` を使用してプログラムを見つけます。

`attach -p process_id program_name`

プロセス ID *process_id* を持つ *program* のデバッグを開始します。

`attach program_name process_id`

プロセス ID *process_id* を持つプログラム名のデバッグを開始します。*program* には `-` を指定できます。dbx は `/proc` を使用してプログラムを見つけます。

`attach -r ...`

`-r` オプションを使用すると、dbx は、`watch`、`display`、`when`、`stop` のコマンドをすべて保持します。`-r` オプションを使用しない場合は、`delete all` と `undisplay 0` コマンドが暗黙に実行されます。

ここで

process_id は、動作中のプロセスのプロセス ID です。

program_name は、実行中プログラムのパス名です。

▼ 実行中の Java プロセスに接続する

- 1 JVM™ ソフトウェアで `libdbx_agent.so` を認識できるように、`libdbx_agent.so` を `LD_LIBRARY_PATH` に追加します。`libdbx_agent.so` は次のようにして追加します。

- Solaris OS を実行しているシステムで 32 ビットの JVM ソフトウェアを使用している場合は、`/installation_directory/SUNWspro/libdbx_agent.so` を `LD_LIBRARY_PATH` に追加します。
- Solaris OS を実行している SPARC システムで 64 ビットの JVM ソフトウェアを使用している場合は、`/installation_directory/SUNWspro/lib/v9/libdbx_agent.so` を `LD_LIBRARY_PATH` に追加します。
- Linux OS を実行している x64 システムで 64 ビットの JVM ソフトウェアを使用している場合は、`/installation_directory/sunstudio12/lib/amd64/libdbx_agent.so` を `LD_LIBRARY_PATH` に追加します。

`installation_directory` は Sun Studio ソフトウェアがインストールされている場所です。

- 2 次のように入力して、Java アプリケーションを起動します。

```
java -Djava.compiler=NONE -Xdebug -Xnoagent -Xrun dbx_agent myclass.class
```

- 3 次のようにプロセス ID を指定して dbx を起動し、プロセスに dbx を接続します。

```
dbx - process_id
```

bsearch コマンド

`bsearch` コマンドは、現在のソースファイルにおいて逆方向検索を行います。ネイティブモードでだけ有効です。

構文

`bsearch string` 現在のファイルの中で、`string` を逆方向で検索します。

`bsearch` 最後の検索文字列を使用して検索を繰り返します。

ここで

`string` は、文字列です。

call コマンド

ネイティブモードでは、`call` コマンドは手続きを呼び出します。Java モードでは、`call` コマンドはメソッドを呼び出します。

ネイティブモードの構文

```
call procedure ([parameters ]) [-lang language] [-resumeone]
```

ここで

language は呼び出す手続きの言語です。

procedure は、手続きの名前です。

parameters は、手続きのパラメータです。

call コマンドによって関数を呼び出すこともできます。戻り値を調べるには、print コマンドを使用します (344 ページの「[print コマンド](#)」参照)。

呼び出されたメソッドがブレークポイントに達することがあります。cont コマンド (303 ページの「[cont コマンド](#)」を参照) を使用して実行を継続するか、pop -c (343 ページの「[pop コマンド](#)」参照) を使用して呼び出しを中止するかを選択できます。呼び出しの中止は、呼び出されたメソッドがセグメント例外を引き起こした場合にも便利です。

-lang オプションは呼び出す手続きの言語で、指定した言語の呼び出し規則を使用するよう dbx に指示します。このオプションは、呼び出された手続きがデバッグ情報なしでコンパイルされ、dbx がパラメータを渡す方法が不明な場合に役立ちます。

-resumeone オプションは手続きが呼び出されたときにスレッドを1つだけ再開します。詳細は、172 ページの「[実行の再開](#)」を参照してください。

Java モードの構文

```
call [class_name.|object_name.] method_name ([parameters])
```

ここで

class_name は、Java クラス名で、パッケージのパス ("." (ピリオド) を修飾子として使用。たとえば test1.extra.T1.Inner) またはフルパス名 (# 記号で始まり、"/" (スラッシュ) や \$ 記号を修飾子として使用。たとえば #test1/extra/T1\$Inner) のいずれかで指定します。修飾子 \$ を使用する場合は、*class_name* を引用符で囲みます。

object_name は、Java オブジェクトの名前です。

method_name は、Java メソッドの名前です。

parameters は、メソッドのパラメータです。

呼び出されたメソッドがブレークポイントに達することがあります。cont コマンド (303 ページの「cont コマンド」を参照) を使用して実行を継続するか、pop -c (343 ページの「pop コマンド」参照) を使用して呼び出しを中止するかを選択できます。呼び出しの中止は、呼び出されたメソッドがセグメント例外を引き起こした場合にも便利です。

cancel コマンド

cancel コマンドは、現在のシグナルを取り消します。このコマンドは、主として when コマンドの本体内で使用します (384 ページの「when コマンド」参照)。ネイティブモードでだけ有効です。

通常、シグナルが取り消されるのは、dbx がシグナルのため停止した場合です。when コマンドがシグナルイベントに接続されている場合、そのシグナルが自動的に取り消されることはありません。cancel コマンドを使用すれば、シグナルを明示的に取り消せます。

構文

```
cancel
```

catch コマンド

catch コマンドは、指定のシグナルを捕獲します。ネイティブモードでだけ有効です。

シグナルを捕獲すると、プロセスがそのシグナルを受信したときに dbx がプログラムを停止します。その時点でプログラムを続行しても、シグナルがプログラムによって処理されることはありません。

構文

catch	捕獲するシグナルのリストを出力します。
catch <i>number number ...</i>	<i>number</i> の番号のシグナルを捕獲します。
catch <i>signal signal ...</i>	<i>signal</i> という名前のシグナルを捕獲します。SIGKILL を捕獲したり無視したりすることはできません。
catch \$(ignore)	すべてのシグナルを捕獲します。

ここで

number は、シグナルの番号です。

signal はシグナル名です。

check コマンド

check コマンドは、メモリーへのアクセス、メモリーリーク、メモリー使用状況をチェックし、実行時検査 (RTC) の現在状態を出力します。ネイティブモードでだけ有効です。

注 - check コマンドが使用できるのは、Solaris プラットフォームのみです。

このコマンドによる実行時検査機能は、debug コマンドによって初期状態にリセットされます。

構文

check -access

アクセス検査を起動します。RTC は、次のエラーを報告します。

baf	不正解放
duf	重複解放
maf	境界整列を誤った解放
mar	境界整列を誤った読み取り
maw	境界整列を誤った書き込み
oom	メモリー不足
rua	非割り当てメモリーからの読み取り
rui	非初期化メモリーからの読み取り
wro	読み取り専用メモリーへの書き込み
wua	非割り当てメモリーへの書き込み

デフォルトの場合、各アクセスエラーが検出されるとプロセスが停止されます。このデフォルト動作を変更するには、dbx 環境変数 `rtc_auto_continue` を使用します。

onが設定されている場合、アクセスエラーはファイルに記録されます(ファイル名はdbx環境変数 `rtc_error_log_file_name` によって制御します)。307ページの「[dbxenv コマンド](#)」を参照してください。

デフォルトの場合、それぞれのアクセスエラーが報告されるのは、最初に発生したときだけです。この動作を変更するには、dbx環境変数 `rtc_auto_suppress` を使用します(この変数のデフォルト値はonです)。307ページの「[dbxenv コマンド](#)」を参照してください。

```
check -leaks [-frames n] [-match m]
```

リーク検査をオンにします。RTCは、次のエラーを報告します。

aib メモリーリークの可能性-唯一のポインタがブロック中央を指す。

air メモリーリークの可能性-ブロックを指すポインタがレジスタ内のみ存在する。

mel メモリーリーク-ブロックを指すポインタがない。

リーク検査がオンの場合、プログラムが存在していれば自動リークレポートが作成されます。このとき、可能性のあるリークを含むすべてのリークが報告されます。デフォルトの場合、簡易レポートが作成されます(dbx環境変数 `rtc_mel_at_exit` によって制御します)。ただし、リークレポートをいつでも要求することができます(355ページの「[showLeaks コマンド](#)」を参照)。

-frames *n* は、リーク報告時に最大 *n* 個のスタックフレームが表示されることを意味します。-match *m* は、複数のリークをまとめます。2個以上のリークに対する割り当て時の呼び出しスタックが *n* 個のフレームに一致するとき、これらのリークは1つのリークレポートにまとめて報告されます。

n のデフォルト値は、8または *m* の値です(どちらか大きい方)。 *n* の最大値は16です。*m* のデフォルト値は、C++では3、それ以外では2です。

```
check -memuse [-frames n] [-match m]
```

-memuse オプションは -leaks オプションと同じような動作をし、プログラム終了時、使用中のブロックのレポート(biu)も有効にします。デフォルトの場合、簡易使用中レポートが生成されます(dbx環境変数 `rtc_biu_at_exit` によって制御します)。プログラム実行中、プログラムの中でメモリーが割り当てられた場所をいつでも調べることができます(356ページの「[showmemuse コマンド](#)」参照)。

-frames *n* は、メモリーの使用状況とリークを報告するときに最大 *n* 個のスタックフレームが表示されることを意味します。-match *m* は、複数のリークをまとめます。2個以上のリークに対する割り当て時の呼び出しスタックが *n* 個のフレームに一致するとき、これらのリークは1つのリークレポートにまとめて報告されます。

n のデフォルト値は、8 または m の値です (どちらか大きい方)。 n の最大値は 16 です。 m のデフォルト値は、C++ では 3、それ以外では 2 です。 `check -leaks` も参照してください。

```
check -all [-frames  $n$ ] [-match  $m$ ]
```

`check -access` または `check -memuse [-frames n] [-match m]` と同じです。

dbx 環境変数 `rtc_biu_at_exit` の値は `check -all` によって変更されないで、デフォルトの場合、終了時にメモリー使用状況レポートは生成されません。

`rtc_biu_at_exit` 環境変数については、304 ページの「dbx コマンド」を参照してください。

```
check [ $functions$ ] [ $files$ ] [ $loadobjects$ ]
```

`functions`、`files`、`loadobjects` における `check -all`、`suppress all`、`unsuppress all` と同じです。

ここで

`functions` は、1 個または複数の関数名です。

`files` は、1 個または複数のファイル名です。

`loadobjects` は、1 個または複数のロードオブジェクト名です。

これを使用することにより、特定の場所を対象として実行時検査を行えます。

注 - RTC ですべてのエラーを検出する際、`-g` を付けてプログラムをコンパイルする必要はありません。ただし、特定のエラー (ほとんどは非初期化メモリーから読み取られるもの) の正確さを保証するには、シンボリック (`-g`) 情報が必要となりますことがあります。このため、一部のエラー (`a.out` の `ru` と共有ライブラリの `ru + aib + air`) は、シンボリック情報を利用できないときには抑止されます。この動作は、`suppress` と `unsuppress` によって変更できます。

clear コマンド

`clear` コマンドは、ブレークポイントをクリアします。ネイティブモードでだけ有効です。

引数 `inclass`、`inmethod`、または `infunction` を付けた `stop`、`trace`、または `when` コマンドを使用して作成したイベントハンドラは、ブレークポイントセットを作成します。`clear` コマンドで指定した `line` がこれらのブレークポイントのどれかに一致した場合、そのブレークポイントだけがクリアされます。特定のセットに属するブレー

クポイントをこの方法でクリアしたあと、そのブレイクポイントを再び使用可能にすることはできません。ただし、関連するイベントハンドラをいったん使用不可にしたあと使用可能にすると、すべてのブレイクポイントが再設定されます。

構文

`clear` 現在の停止点にあるブレイクポイントをすべてクリアします。

`clear line` `line` にあるブレイクポイントすべてをクリアします。

`clear filename:line` `filename` の `line` にあるブレイクポイントをすべてクリアします。

ここで

`line` は、ソースコード行の番号です。

`filename` は、ソースコードファイルの名前です。

collector コマンド

`collector` コマンドは、パフォーマンスアナライザによって分析するパフォーマンスデータを収集します。ネイティブモードでだけ有効です。

注 - `collector` コマンドが使用できるのは、Solaris プラットフォームのみです。

構文

`collector command_list` 1 個または複数の `collector` コマンドを指定します。

`collector archive options` 終了したときに実験をアーカイブ化するモードを指定します (297 ページの「[collector archive コマンド](#)」参照)。

`collector dbxsample options` `dbx` がターゲットプロセスを停止したときのサンプルの収集を制御します (297 ページの「[collector dbxsample コマンド](#)」参照)。

`collector disable` データ収集を停止して現在の実験をクローズします (297 ページの「[collector disable コマンド](#)」参照)。

<code>collector enable</code>	コレクタを使用可能にして新規の実験をオープンします (298 ページの「 collector enable コマンド 」参照)。
<code>collector heaptrace options</code>	ヒープトレースデータの収集を有効または無効にします (298 ページの「 collector heaptrace コマンド 」参照)。
<code>collector hwprofile options</code>	ハードウェアカウンタプロファイル設定値を指定します (298 ページの「 collector hwprofile コマンド 」参照)。
<code>collector limit options</code>	記録されているプロファイルデータの量を制限します (299 ページの「 collector limit コマンド 」参照)。
<code>collector mpitrace options</code>	MPI トレースデータの収集を有効または無効にします (299 ページの「 collector mpitrace コマンド 」参照)。
<code>collector pause</code>	パフォーマンスデータの収集は停止しますが、実験はオープン状態のままとします (299 ページの「 collector pause コマンド 」参照)。
<code>collector profile options</code>	呼び出しスタックプロファイルデータを収集するための設定値を指定します (300 ページの「 collector profile コマンド 」参照)。
<code>collector resume</code>	一時停止後、パフォーマンスデータの収集を開始します (300 ページの「 collector resume コマンド 」参照)。
<code>collector sample options</code>	標本設定値を指定します (300 ページの「 collector sample コマンド 」参照)。
<code>collector show options</code>	現在のコレクタ設定値を表示します (301 ページの「 collector show コマンド 」参照)。
<code>collector status</code>	現在の実験に関するステータスを照会します (301 ページの「 collector status コマンド 」参照)。
<code>collector store options</code>	ファイルの制御と設定値を実験します (301 ページの「 collector store コマンド 」参照)。
<code>collector synctrace options</code>	スレッド同期待ちトレースデータの設定値を指定します (302 ページの「 collector synctrace コマンド 」参照)。
<code>collector tha options</code>	スレッドアナライザデータ収集の設定値を指定します (302 ページの「 collector tha コマンド 」参照)。

`collector version` データ収集に使用される `libcollector.so` のバージョンを報告します(303ページの「`collector version` コマンド」参照)。

ここで

`options` は、各コマンドで指定できる設定値です。

データの収集を開始するには、`collector enable` と入力します。

データ収集を停止するには、`collector disable` と入力します。

collector archive コマンド

`collector archive` コマンドは、実験が終了したときに使用するアーカイブモードを指定します。

構文

`collector archive on|off|copy` デフォルトでは通常のアーカイブが使用されます。アーカイブしない場合は、`off` を指定します。ロードオブジェクトを実験にコピーするには、`copy` を指定します。

collector dbxsample コマンド

`collector dbxsample` コマンドは、プロセスが `dbx` によって停止された場合に、標本を記録するかどうかを指定します。

構文

`collector dbxsample on|off` デフォルトでは、プロセスが `dbx` によって停止された場合に標本を収集します。収集しない場合は、`off` を指定します。

collector disable コマンド

`collector disable` コマンドは、データ収集を停止して現在の実験をクローズします。

構文

```
collector disable
```

collector enable コマンド

collector enable コマンドは、コレクタを使用可能にして新規の実験をオープンします。

構文

```
collector enable
```

collector heaptrace コマンド

collector heaptrace コマンドは、ヒープのトレース (メモリーの割り当て) データの収集オプションを指定します。

構文

```
collector heaptrace on|off
```

デフォルトでは、ヒープのトレースデータは収集されません。このデータを収集するには、on を指定します。

collector hwprofile コマンド

collector hwprofile コマンドは、ハードウェアカウンタオーバーフロープロファイルデータ収集のオプションを指定します。

構文

```
collector hwprofile on|off
```

デフォルトの場合、ハードウェアカウンタオーバーフロープロファイルデータは収集されません。このデータを収集するには、on を指定します。

```
collector hwprofile list
```

利用できるカウンタのリストを出力します。

```
collector hwprofile counter name interval [name2 interval2]
```

ハードウェアカウンタ名と間隔を指定します。

ここで

name は、ハードウェアカウンタの名前です。

interval は、ミリ秒単位による収集間隔です。

name2 は、第2ハードウェアカウンタの名前です。

interval2 は、ミリ秒単位による収集間隔です。

ハードウェアカウンタはシステム固有であるため、どのようなカウンタを利用できるかはご使用のシステムによって異なります。多くのシステムでは、ハードウェアカウンタオーバーフロープロファイル機能をサポートしていません。こういったマシンの場合、この機能は使用不可になっています。

collector limit コマンド

collector limit コマンドは、実験ファイルのサイズの上限を指定します。

構文

collector limit *value*

ここで

value - メガバイト単位。記録されているプロファイルデータの量を制限します。制限に達すると、それ以上のプロファイルデータは記録されませんが、実験はオープンのまままで標本ポイントの記録は継続します。記録されるレコードのデフォルトの制限値は2000M バイトです。

collector mpitrace コマンド

collector mpitrace コマンドは、MPI のトレースデータの収集オプションを指定します。

構文

collector mpitrace on|off デフォルトでは、MPI のトレースデータは収集されません。このデータを収集するには、on を指定します。

collector pause コマンド

collector pause コマンドはデータ収集を停止しますが、現在の実験はオープン状態のままとします。コレクタが一時停止している間、サンプルポイントは記録されません。サンプルは一時停止の前に生成され、再開直後に別のサンプルが生成されます。collector resume コマンドを使用すると、データ収集を再開できます ([300 ページの「collector resume コマンド」](#) 参照)。

構文

collector pause

collector profile コマンド

collector profile コマンドは、プロファイルデータ収集のオプションを指定します。

構文

collector profile on|off プロファイルデータ収集モードを指定します。

collector profile timer *interval* プロファイルタイマー時間を固定ポイントまたは浮動小数点で、オプションの m (ミリ秒の場合) または u (マイクロ秒の場合) を付けて指定します。

collector resume コマンド

collector resume コマンドは、collector pause コマンドによる一時停止のあと、データ収集を再開します (299 ページの「[collector pause コマンド](#)」参照)。

構文

collector resume

collector sample コマンド

collector sample コマンドは、標本モードと標本間隔を指定します。

構文

collector sample periodic|manual 標本モードを指定します。

collector sample period *seconds* 標本間隔を *seconds* 単位で指定します。

collector sample record [*name*] *name* (オプション) を指定して標本を記録します。

ここで

seconds は、標本間隔の長さです。

name は、標本の名前です。

collector show コマンド

collector show コマンドは、1 個または複数のオプションカテゴリの設定値を表示します。

構文

collector show all	すべての設定値を表示します。
collector show archive	すべての設定値を表示します。
collector show profile	呼び出しスタックプロファイル設定値を表示します。
collector show synctrace	スレッド同期待ちトレース設定値を表示します。
collector show hwprofile	ハードウェアカウンタデータ設定値を表示します。
collector show heaptrace	ヒープトレースデータ設定値を表示します。
collector show limit	実験サイズの上限を表示します。
collector show mpitrace	MPI トレースデータ設定値を表示します。
collector show sample	標本設定値を表示します。
collector show store	ストア設定値を表示します。
collector show tha	スレッドアナライザのデータ設定値を表示します。

collector status コマンド

collector status コマンドは、現在の実験のステータスについて照会します。

構文

```
collector status
```

collector store コマンド

collector store コマンドは、実験が保存されているディレクトリとファイルの名前を指定します。

構文

`collector store directory pathname` 実験が保存されているディレクトリを指定します。

`collector store filename filename` 実験ファイル名を指定します。

`collector store group string` 実験グループ名を指定します。

ここで

pathname は、実験を保存するディレクトリのパス名です。

filename は、実験ファイルの名前です。

string は、実験グループの名前です。

collector synctrace コマンド

`collector synctrace` コマンドは、同期待ちトレースデータの収集オプションを指定します。

構文

`collector synctrace on|off`

デフォルトの場合、スレッド同期待ちトレースデータは収集されません。このデータを収集するには、`on` を指定します。

`collector synctrace threshold milliseconds`

しきい値をマイクロ秒単位で指定します。デフォルト値は 100 です。

`collector synctrace threshold calibrate`

しきい値は、自動的に算出されます。

ここで

microseconds は、この値未満であるときに同期待ちイベントが破棄されるしきい値です。

collector tha コマンド

構文

`collector tha on|off` デフォルトでは、スレッドアナライザのデータは収集されません。このデータを収集するには、`on` を指定します。

collector version コマンド

collector version コマンドは、データ収集に使用される libcollector.so のバージョンを報告します。

構文

```
collector version
```

cont コマンド

cont コマンドは、プロセスの実行を継続します。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

cont	実行を継続します。MT プロセスのすべてのスレッドが再開します。Control-Cを使用すると、プログラムの実行が停止します。
cont ... -sig <i>signal</i>	シグナル <i>signal</i> で実行を継続します。
cont ... <i>id</i>	継続するスレッドまたはLWPを <i>id</i> で指定します。
cont at <i>line</i> [<i>id</i>]	行 <i>line</i> で実行を継続します。アプリケーションがマルチスレッドの場合には <i>id</i> が必須です。
cont ... -follow parent child both	dbx の <code>follow_fork_mode</code> 環境変数を stop に設定した場合、このオプションを使用して後続のプロセスを選択します。both は Sun Studio IDE でのみ有効です。

dalias コマンド

dalias コマンドは、dbx 形式の (csh 形式) 別名を定義します。ネイティブモードでだけ有効です。

構文

<code>dalias</code>	(dbx alias) 現在定義されている別名をすべて一覧表示します。
<code>dalias name</code>	別名 <i>name</i> の定義がある場合には、それを表示します。
<code>dalias name definition</code>	<i>name</i> を <i>definition</i> の別名として定義します。 <i>definition</i> には空白を含めることができます。セミコロンまたは改行によって定義を終端させます。

ここで

name は、別名の名前です。

definition は、別名の定義です。

dbx は、別名に通常使用される次の csh 履歴置換メタ構文を受け付けます。

!:<n>

!-<n>

!^

!\$

!*

通常、!の前にはバックスラッシュを付ける必要があります。たとえば、次のようにします。

```
dalias goto "stop at \!:1; cont; clear"
```

詳細については、csh(1) マニュアルページを参照してください。

dbx コマンド

dbx コマンドは dbx を起動します。

ネイティブモードの構文

`dbx options program_name` *program_name* をデバッグします。

<code>dbx options program_name core</code>	コアファイル <code>core</code> によって <code>program_name</code> をデバッグします。
<code>dbx options program_name process_id</code>	プロセス ID <code>process_id</code> を持つ <code>program_name</code> をデバッグします。
<code>dbx options - process_id</code>	プロセス ID <code>process_id</code> をデバッグします。dbx は、 <code>/proc</code> によってプログラムを見つけてみます。
<code>dbx options - core</code>	コアファイル <code>core</code> を使用してデバッグします。「 307 ページの「debug コマンド」 」も参照してください。
<code>dbx options -r program_name arguments</code>	引数 <code>arguments</code> を付けて <code>program_name</code> を実行します。異常終了した場合は <code>program_name</code> のデバッグを開始します。そうでない場合はそのまま終了します。

ここで

`program_name` は、デバッグ対象プログラムの名前です。

`process_id` は、動作中のプロセスのプロセス ID です。

`arguments` は、プログラムに渡す引数です。

`options` は、[306 ページの「オプション」](#) に挙げられているオプションです。

Java モードの構文

`dbx options program_name{.class | .jar}`
`program_name` をデバッグします。

`dbx options program_name{.class | .jar} process_id`
 プロセス ID `process_id` を持つ `program_name` をデバッグします。

`dbx options - process_id`
 プロセス ID `process_id` をデバッグします。dbx は、`/proc` によってプログラムを見つけてみます。

`dbx options -r program_name{.class | .jar} arguments`
 引数 `arguments` を付けて `program_name` を実行します。異常終了した場合は `program_name` のデバッグを開始します。そうでない場合はそのまま終了します。

ここで

`program_name` は、デバッグ対象プログラムの名前です。

process_id は、動作中のプロセスのプロセス ID です。

arguments は、プログラム (JVM ソフトウェアではない) に渡す引数です。

options は、306 ページの「オプション」に挙げられているオプションです。

オプション

ネイティブモード、Java モードともに、*options* には次を使用できます。

-B	すべてのメッセージを抑制します。デバッグするプログラムの exit コードを返します。
-c <i>commands</i>	コマンドを実行してから入力を要求します。
-C	実行時検査ライブラリをあらかじめ読み込みます (292 ページの「check コマンド」参照)。
-d	-s を付けて使用した場合、読み取った <i>file</i> を削除します。
-e	入力コマンドを表示します。
-f	コアファイルが一致しない場合でも、コアファイルの読み込みを強制します。
-h	dbx のヘルプを出力します。
-I <i>dir</i>	<i>dir</i> を pathmap セットに追加します (342 ページの「pathmap コマンド」参照)。
-k	キーボードの変換状態を保存および復元します。
-q	スタブの読み込みについてのメッセージの出力を抑制します。
-r	プログラムを実行します。プログラムが正常に終了した場合は、そのまま終了します。
-R	dbx の README ファイルを出力します。
-s <i>file</i>	<i>/current_directory/.dbxrc</i> または <i>\$HOME/.dbxrc</i> の代わりに <i>file</i> を起動ファイルとして使用します。
-S	初期設定ファイル <i>/installation_directory/lib/dbxrc</i> の読み込みを抑制します。
-V	dbx のバージョンを出力します。
-w <i>n</i>	where コマンドで <i>n</i> 個のフレームをスキップします。
-x <i>exec32</i>	64 ビット OS の実行されているシステムで、64 ビット dbx バイナリではなく 32 ビット dbx バイナリを実行します。

--	オプションのリストの最後を示します。プログラム名がダッシュで始まる場合は、これを使用します。
----	--

dbxenv コマンド

dbxenv コマンドは、dbx 環境変数の表示や設定を行います。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

dbxenv dbx 環境変数の現在の設定値を表示します。

dbxenv *environment_variable* *setting* *environment_variable* に *setting* を設定します。

ここで

environment_variable は dbx 環境変数です。

setting は、その変数の有効な設定値です。

debug コマンド

debug コマンドは、デバッグ対象プログラムの表示や変更を行います。ネイティブモードでは、指定したアプリケーションを読み込み、アプリケーションのデバッグを開始します。Java モードでは、指定したアプリケーションを読み込み、クラスファイルが存在するかどうかを確認し、アプリケーションのデバッグを開始します。

ネイティブモードの構文

debug デバッグ対象プログラムの名前と引数を出力します。

debug *program_name* プロセスやコアなしで *program_name* のデバッグを開始します。

debug -c *core* *program_name* コアファイル *core* による *program_name* のデバッグを開始します。

debug -p *process_id* *program_name* プロセス ID *process_id* を持つ *program_name* のデバッグを開始します。

<code>debug program_name core</code>	コアファイル <i>core</i> による <i>program</i> のデバッグを開始します。 <i>program_name</i> には - を指定できません。 <i>dbx</i> は、コアファイルから実行可能ファイルの名前を取り出そうとします。詳細については、 44 ページの「既存のコアファイルのデバッグ」 を参照してください。
<code>debug program_name process_id</code>	プロセス ID <i>process_id</i> を持つ <i>program_name</i> のデバッグを開始します。 <i>program_name</i> には - を指定できません。 <i>dbx</i> は <i>/proc</i> を使用してプログラムを見つけます。
<code>debug -f ...</code>	コアファイルが一致しない場合でも、コアファイルの読み込みを強制します。
<code>debug -r ...</code>	-r オプションを使用すると、 <i>dbx</i> は <i>display</i> 、 <i>trace</i> 、 <i>when</i> 、 <i>stop</i> のコマンドをすべて保持します。-r オプションを使用しない場合は、 <i>delete all</i> と <i>undisplay 0</i> が暗黙に実行されます。
<code>debug -clone ...</code>	-clone オプションは新たな <i>dbx</i> プロセスの実行を開始するので、複数のプロセスを同時にデバッグできます。Sun Studio IDE で使用する場合にのみ有効です。
<code>debug -clone</code>	何もデバッグしない <i>dbx</i> プロセスを新たに開始します。Sun Studio IDE で使用する場合にのみ有効です。
<code>debug [options] -- program_name</code>	<i>program_name</i> がダッシュで始まる場合でも、 <i>program_name</i> のデバッグを開始します。

ここで

core は、コアファイルの名前です。

options は、[309 ページの「オプション」](#)に挙げられているオプションです。

process_id は、実行中プロセスのプロセス ID です。

program_name は、プログラムのパス名です。

`debug` でプログラムを読み込むと、リーク検査とアクセス検査はオフになります。`check` コマンドを使用すれば、これらの検査を使用可能にできます ([292 ページの「check コマンド」](#)参照)。

Java モードの構文

`debug`

デバッグ対象プログラムの名前と引数を出力します。

`debug program_name [.class | .jar]`

プロセスなしで `program_name` のデバッグを開始します。

`debug -p process_id program_name [.class | .jar]`

プロセス ID `process_id` を持つ `program_name` のデバッグを開始します。

`debug program_name [.class | .jar] process_id`

プロセス ID `process_id` を持つ `program_name` のデバッグを開始します。

`program_name` には `-` を指定できます。dbx は `/proc` を使用してプログラムを見つけます。

`debug -r ...`

`-r` オプションを使用すると、dbx は `watch`、`display`、`trace`、`when`、`stop` のコマンドをすべて保持します。`-r` オプションを使用しない場合は、`delete all` と `undisplay 0` が暗黙に実行されます。

`debug -clone ...`

`-clone` オプションは新たな dbx プロセスの実行を開始するので、複数のプロセスを同時にデバッグできます。Sun Studio IDE で使用する場合にのみ有効です。

`debug -clone`

何もデバッグしない dbx プロセスを新たに開始します。Sun Studio IDE で使用する場合にのみ有効です。

`debug [options] -- program_name{.class | .jar}`

`program_name` がダッシュで始まる場合でも、`program_name` のデバッグを開始します。

ここで

`file_name` は、ファイルの名前です。

`options` は、[309 ページの「オプション」](#) に挙げられているオプションです。

`process_id` は、動作中のプロセスのプロセス ID です。

`program_name` は、プログラムのパス名です。

オプション

`-c commands` コマンドを実行してから入力を要求します。

`-d` `-s` と併せて指定した場合に、読み込み後に `file_name` で指定したファイルを削除します。

-e	入力コマンドを表示します。
-I <i>directory_name</i>	<i>directory_name</i> を pathmap セットに追加します (342 ページの「pathmap コマンド」参照)。
-k	キーボードの変換状態を保存および復元します。
-q	スタブの読み込みについてのメッセージの出力を抑止します。
-r	プログラムを実行します。プログラムが正常に終了した場合は、そのまま終了します。
-R	dbx の README ファイルを出力します。
-s <i>file</i>	<i>/current_directory/.dbxrc</i> または <i>\$HOME/.dbxrc</i> の代わりに <i>file</i> を起動ファイルとして使用します。
-S	初期設定ファイル <i>/installation_directory/lib/dbxrc</i> の読み込みを抑止します。
-V	dbx のバージョンを出力します。
-w <i>n</i>	where コマンドで <i>n</i> 個のフレームをスキップします。
--	オプションのリストの最後を示します。プログラム名がダッシュで始まる場合は、これを使用します。

delete コマンド

delete コマンドは、ブレークポイントなどのイベントを削除します。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

delete [-h] <i>handler_id</i> ...	指定の <i>handler_id</i> を持つ trace コマンド、when コマンド、または stop コマンドを削除します。隠しハンドラを削除するには、-h オプションを使用する必要があります。
delete [-h] 0 all -all	常時隠しハンドラを除き、trace コマンド、when コマンド、stop コマンドをすべて削除します。-h を指定すると、隠しハンドラも削除されます。
delete -temp	一時ハンドラをすべて削除します。
delete \$firedhandlers	最後の停止を引き起こしたハンドラすべてを削除します。

ここで

handler_id は、ハンドラの識別子です。

detach コマンド

detach コマンドは、dbx の制御からターゲットプロセスを解放します。

ネイティブモードの構文

detach	ターゲットから dbx を切り離し、保留状態のシグナルがある場合はそれらのシグナルを取り消します。
detach -sig <i>signal</i>	指定の <i>signal</i> を転送している間、切り離します。
detach -stop	dbx をターゲットから切り離してプロセスを停止状態にします。このオプションを使用すると、占有アクセスによってブロックされるほかの /proc ベースのデバッグツールを一時的に適用することができます。例については、 90 ページの「プロセスから dbx を切り離す」 を参照してください。

ここで

signal はシグナル名です。

Java モードの構文

detach	ターゲットから dbx を切り離し、保留状態のシグナルがある場合はそれらのシグナルを取り消します。
--------	---

dis コマンド

dis コマンドは、マシン命令を逆アセンブルします。ネイティブモードでだけ有効です。

構文

dis [-a] <i>address</i> [/count]	アドレス <i>address</i> を始点とし、 <i>count</i> 命令 (デフォルトは 10) を逆アセンブルします。
----------------------------------	--

`dis address1, address2` `address1` から `address2` までの命令を逆アセンブルします。

`dis` `+` の値を視点とし、10 個の命令を逆アセンブルします (315 ページの「`examine` コマンド」参照)。

ここで

`address` は、逆アセンブルを開始するアドレスです。デフォルトの `address` 値は、前にアセンブルされた最後のアドレスの次のアドレスになります。この値は、`examine` コマンド (315 ページの「`examine` コマンド」参照) によって共有されます。

`address1` は、逆アセンブルを開始するアドレスです。

`address2` は、逆アセンブルを停止するアドレスです。

`count` は、逆アセンブル対象命令の数です。`count` のデフォルト値は 10 です。

オプション

- a 関数のアドレスと使用した場合、関数全体を逆アセンブルします。パラメータなしで使用した場合、現在の関数に残りがあると、その残りを逆アセンブルします。

display コマンド

ネイティブモードでは、`display` コマンドはすべての停止ポイントで式を再評価して出力します。Java モードでは、`display` コマンドはすべての停止ポイントで式、局所変数、パラメータを評価して出力します。オブジェクト参照は、1つのレベルに展開され、配列は項目と同様に出力されます。

式はコマンドを入力したときに現在のスコープで構文分析され、すべての停止ポイントで再評価されます。式は入力時に分析されるため、式の正確さをすぐに確認できます。

ネイティブモードの構文

`display`
表示されている式のリストを表示します。

`display expression, ...`
すべての停止点で式 `expression, ...` の値を表示します。

`display [-r|+r|-d|+d|-p|+p|-L|-fformat|-F format|--] expression, ...`
 フラグの意味については、344 ページの「print コマンド」を参照してください。

ここで

expression は、有効な式です。

format は、式の出力時に使用する形式です。詳細については、344 ページの「print コマンド」を参照してください。

Java モードの構文

`display`

表示される変数およびパラメータのリストを出力します。

`display expression| identifier, ...`

すべての停止ポイントで、表示される変数およびパラメータ *identifier, ...* の値を表示します。

`display [-r|+r|-d|+d|-p|+p|-fformat|-F format|-Fformat|--] expression |identifier,`

...

フラグの意味については、344 ページの「print コマンド」を参照してください。

ここで

class_name は、Java クラス名で、パッケージのパス (". "(ピリオド) を修飾子として使用。たとえば `test1.extra.T1.Inner`) またはフルパス名 (# 記号で始まり、"/" (スラッシュ) や \$ を修飾子として使用。たとえば `#test1/extra/T1$Inner`) のいずれかで指定します。修飾子 \$ を使用する場合は、*class_name* を引用符で囲みます。

expression は、有効な Java の式です。

field_name は、クラス内のフィールド名です。

format は、式の出力時に使用する形式です。詳細については、344 ページの「print コマンド」を参照してください。

identifier は、局所変数またはパラメータです。これには、`this`、現在のクラスのインスタンス変数 (*object_name.field_name*)、クラス (static) 変数 (*class_name.field_name*) が含まれます。

object_name は、Java オブジェクトの名前です。

構文

`edit` 現在のファイルを編集します。

`edit file_name` 指定のファイル `file_name` を編集します。

`edit procedure` 関数または手続き `procedure` が入っているファイルを編集します。

ここで

`file_name` は、ファイルの名前です。

`procedure` は、関数または手続きの名前です。

examine コマンド

`examine` コマンドは、メモリーの内容を表示します。ネイティブモードでだけ有効です。

構文

`examine [address] [/[count] [format]]` `address` を始点とし、`count` 個の項目のメモリー内容を形式 `format` で表示します。

`examine address1, address2 [/[format]]` `address1` から `address2` までのメモリー内容 (`address1`、`address2` を含む) を形式 `format` で表示します。

`examine address= [format]` アドレスを (アドレスの内容ではなく) 指定の形式で表示します。

直前に表示された最後のアドレスを示す + (省略した場合と同じ) を `address` として使用できます。

`x` は、`examine` の事前定義別名です。

ここで

`address` は、メモリーの内容の表示を開始するアドレスです。デフォルトの `address` 値は、内容が最後に表示されたアドレスの次のアドレスになります。この値は `dis` コマンド (311 ページの「`dis` コマンド」参照) によって共有されます。

`address1` は、メモリーの内容の表示を開始するアドレスです。

*address2*は、メモリーの内容の表示を停止するアドレスです。

*count*は、メモリーの内容を表示するアドレスの数です。*count*のデフォルト値は1です。

*format*は、メモリーアドレスの内容を表示する形式です。最初の *examine* コマンドのデフォルトの形式は X (16進数) で、後続の *examine* コマンドに対して前の *examine* コマンドに指定されている形式です。次に示す値は *format* に対して常に有効です。

<i>o</i> , <i>O</i>	8進数 (2 または 4 バイト)
<i>x</i> , <i>X</i>	16進数 (2 または 4 バイト)
<i>b</i>	8進数 (1 バイト)
<i>c</i>	文字
<i>w</i>	ワイド文字
<i>s</i>	文字列
<i>W</i>	ワイド文字列
<i>f</i>	16進浮動小数点数 (4 バイト、6 桁の精度)
<i>F</i>	16進浮動小数点数 (8 バイト、14 桁の精度)
<i>g</i>	Fと同じです。
<i>E</i>	16進浮動小数点数 (16 バイト、14 桁の精度)
<i>ld</i> , <i>LD</i>	10進数 (4 バイト、 <i>D</i> と同じ)
<i>lo</i> , <i>LO</i>	8進数 (4 バイト、 <i>o</i> と同じ)
<i>lx</i> , <i>LX</i>	16進数 (4 バイト、 <i>x</i> と同じ)
<i>Ld</i> , <i>LD</i>	10進数 (8 バイト)
<i>Lo</i> , <i>LO</i>	8進数 (8 バイト)
<i>Lx</i> , <i>LX</i>	16進数 (8 バイト)

exception コマンド

exception コマンドは、現在の C++ 例外の値を出力します。ネイティブモードでだけ有効です。

構文

`exception [-d | +d]` 現在の C++ 例外がある場合、その値を出力します。

`-d` フラグの意味については、[344 ページ](#)の「`print` コマンド」を参照してください。

exists コマンド

`exists` コマンドは、シンボルの有無をチェックします。ネイティブモードでだけ有効です。

構文

`exists name` 現在のプログラム内で `name` が見つかった場合は 0、`name` が見つからなかった場合は 1 を返します。

ここで

`name` は、シンボルの名前です。

file コマンド

`file` コマンドは、現在のファイルの表示や変更を行います。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

`file` 現在のファイルの名前を出力します。

`file file_name` 現在のファイルを変更します。

ここで

`file_name` は、ファイルの名前です。

files コマンド

ネイティブモードでは、files コマンドは正規表現に一致したファイル名を表示します。Java モードでは、files コマンドは dbx で認識されているすべての Java ソースファイルのリストを表示します。Java ソースファイルが .class または .jar ファイルと同一のディレクトリにない場合は、\$JAVASRCPATH 環境変数を設定しないと、dbx でファイルが認識されないことがあります (222 ページの「Java ソースファイルの格納場所の指定」を参照)。

ネイティブモードの構文

`files` 現在のプログラムに対してデバッグ情報を提供したファイルすべての名前を一覧表示します (-g によってコンパイルされたもの)。

`files regular_expression` 指定の正規表現に一致し -g によってコンパイルされたファイルすべての名前を一覧表示します。

ここで

regular_expression は、正規表現です。

次に例を示します。

```
(dbx) files ^r
myprog:
retregs.cc
reg_sorts.cc
reg_errmsgs.cc
rhosts.cc
```

Java モードの構文

`files dbx` で認識されているすべての Java ソースファイルの名前を表示します。

fix コマンド

fix コマンドは、修正されたソースファイルを再コンパイルし、修正された関数をアプリケーションに動的にリンクします。ネイティブモードでだけ有効です。Linux プラットフォームでは有効ではありません。

<code>fortran_modules -f <i>module_name</i></code>	指定したモジュールのすべての関数を一覧表示します。
<code>fortran_modules -v <i>module_name</i></code>	指定したモジュールのすべての変数を一覧表示します。

frame コマンド

frame コマンドは、現在のスタックフレーム番号の表示や変更を行います。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

<code>frame</code>	現在のフレームのフレーム番号を表示します。
<code>frame [-h] <i>number</i></code>	現在のフレームとしてフレーム <i>number</i> を設定します。
<code>frame [-h] +[<i>number</i>]</code>	<i>number</i> 個のフレームだけスタックを上方向に移動します。デフォルトは1です。
<code>frame [-h] -[<i>number</i>]</code>	<i>number</i> 個のフレームだけスタックを下方向に移動します。デフォルトは1です。
<code>-h</code>	フレームが隠されている場合でもフレームに進みます。

ここで

number は、呼び出しスタック内のフレームの番号です。

func コマンド

ネイティブモードでは、func コマンドは現在の関数を表示または変更します。Java モードでは、func コマンドは現在のメソッドを表示または変更します。

ネイティブモードの構文

<code>func</code>	現在の関数の名前を出力します。
<code>func <i>procedure</i></code>	現在の関数を関数または手続き <i>procedure</i> に変更します。

ここで

procedure は、関数または手続きの名前です。

Java モードの構文

`func`

現在の関数の名前を出力します。

`func [class_name.] method_name [(parameters)]`

現在の関数をメソッド `method_name` に変更します。

ここで

`class_name` は、Java クラス名で、パッケージのパス (". "(ピリオド) を修飾子として使用。たとえば `test1.extra.T1.Inner` またはフルパス名 (# 記号で始まり、"/" (スラッシュ) や \$ を修飾子として使用。たとえば `#test1/extra/T1$Inner`) のいずれかで指定します。修飾子 \$ を使用する場合は、`class_name` を引用符で囲みます。

`method_name` は、Java メソッドの名前です。

`parameters` は、メソッドのパラメータです。

funcs コマンド

`funcs` コマンドは、特定の正規表現に一致する関数名をすべて一覧表示します。ネイティブモードでだけ有効です。

構文

`funcs`

現在のプログラム内の関数すべてを一覧表示します。

`funcs [-f file_name] [-g] [regular_expression]`

`-f file_name` を指定すると、ファイル内の関数すべてが表示されます。 `-g` を指定すると、デバッグ情報を持つ関数すべてが表示されます。 `regular_expression` を指定すると、この正規表現に一致する関数すべてが表示されます。

ここで

`file_name` は、一覧表示対象の関数が入っているファイルの名前です。

`regular_expression` は、一覧表示対象の関数が一致する正規表現です。

たとえば、次のようにします。

```
(dbx) funcs [vs]print
```

```
"libc.so.1"ispri
```

```
"libc.so.1"wsprintf
```

```
"libc.so.1"sprintf  
"libc.so.1"vprintf  
"libc.so.1"vsprintf
```

gdb コマンド

gdb コマンドは、gdb コマンドセットをサポートします。ネイティブモードでだけ有効です。

構文

`gdb on | off` `gdb on` を使用すると、`dbx` が `gdb` コマンドを理解し受け付ける `gdb` コマンドモードに入ります。`gdb` コマンドモードを終了し `dbx` コマンドモードに戻るには、`gdb off` と入力します。`gdb` コマンドモードでは `dbx` コマンドは受け付けられず、その逆に `dbx` コマンドモードでは `gdb` コマンドが受け付けられません。ブレークポイントなどのデバッグ設定は、コマンドモードの種類にかかわらず保持されます。

<code>gdb on off</code>	
---------------------------	--

このリリースでは、次の `gdb` コマンドをサポートしていません。

- `commands`
- `define`
- `handle`
- `hbreak`
- `interrupt`
- `maintenance`
- `printf`
- `rbreak`
- `return`
- `signal`
- `tcatch`
- `until`

handler コマンド

handler コマンドは、イベントハンドラを変更します (使用可能や使用不可にするなど)。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

ハンドラは、デバッグセッションで管理する必要があるイベントそれぞれについて作成されます。trace、stop、when の各コマンドは、ハンドラを作成します。これらのコマンドはそれぞれ、ハンドラ ID と呼ばれる番号を返します (*handler_id*)。

handler、status、delete の各コマンドは、一般的な方法でハンドラの操作やハンドラ情報の提供を行います。

構文

handler -enable <i>handler_id</i> ...	特定のハンドラを使用可能にし、全ハンドラを示す all を <i>handler_id</i> として指定します。
handler -disable <i>handler_id</i> ...	特定のハンドラを使用不可にし、全ハンドラを示す all を <i>handler_id</i> として指定します。 <i>handler_id</i> の代わりに \$firedhandlers を使用すると、最後の停止を引き起こしたハンドラが使用不可となります。
handler -count <i>handler_id</i>	特定のハンドラのトリップカウンタの値を出力します。
handler -count <i>handler_id</i> <i>new_limit</i>	特定のイベントに対し、新たなカウント制限値を設定します。
handler -reset <i>handler_id</i>	特定のハンドラのトリップカウンタをリセットします。

ここで

handler_id は、ハンドラの識別子です。

hide コマンド

hide コマンドは、特定の正規表現に一致するスタックフレームを隠します。ネイティブモードでだけ有効です。

構文

`hide` 現在有効であるスタックフレームフィルタを一覧表示します。

`hide regular_expression` *regular_expression* に一致するスタックフレームを隠します。正規表現は関数名またはロードオブジェクトの名前を表し、sh または ksh の正規表現スタイルをとります。

ここで

regular_expression は、正規表現です。

ignore コマンド

`ignore` コマンドは、指定のシグナルを捕獲しないことを `dbx` プロセスに指示します。ネイティブモードでだけ有効です。

シグナルを無視すると、プロセスがそのシグナルを受信しても `dbx` が停止しなくなります。

構文

`ignore` 無視するシグナルのリストを出力します。

`ignore number ...` *number* の番号のシグナルを無視します。

`ignore signal ...` *signal* という名前のシグナルを無視します。SIGKILL を捕獲したり無視したりすることはできません。

ここで

number は、シグナルの番号です。

signal はシグナル名です。

import コマンド

`import` コマンドは、`dbx` コマンドライブラリからコマンドをインポートします。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

`import pathname` dbx コマンドライブラリ *pathname* からコマンドをインポートします。

ここで

pathname は、dbx コマンドライブラリのパス名です。

intercept コマンド

`intercept` コマンドは、指定タイプ (C++ のみ) の (C++ 例外) を送出します。ネイティブモードでだけ有効です。

送出された例外の種類が阻止リストの種類と一致した場合、その例外の種類が除外リストの種類とも一致した場合を除いて、dbx は停止します。一致するものがない送出例外は、「処理されない」送出と呼ばれます。送出元関数の例外仕様に一致しない送出例外は、「予期されない」送出と呼ばれます。

処理されない送出と予期されない送出は、デフォルト時に阻止されます。

構文

`intercept -x excluded_typename [, excluded_typename ...]`
種類が *excluded_typename* の送出を阻止リストに追加します。

`intercept -a[ll] -x excluded_typename [, excluded_typename ...]`
excluded_typename 以外のすべての種類の送出を阻止リストに追加します。

`intercept -s[et] [intercepted_typename [, intercepted_typename ...]] [-x excluded_typename [, excluded_typename]]`
インターセプトリストと除外リストの両方をクリアし、リストを指定した種類のみを送出する阻止または除外に設定します。

`intercept`
阻止対象の型を一覧表示します。

ここで

included_typename および *excluded_typename* は、`List<int>` や `unsigned short` などの例外仕様です。

java コマンド

java コマンドは、dbx が JNI モードの場合に、指定したコマンドの Java バージョンを実行するように指定します。java コマンドは、指定したコマンドで Java の式の評価を実行するように設定します。また、該当する場合には、Java スレッドおよびスタックフレームを表示します。

構文

`java command`

ここで

`command` は、実行対象コマンドの名前および引数です。

jclasses コマンド

jclasses コマンドは、コマンド実行時に dbx で認識されているすべての Java クラスの名前を出力します。Java モードでだけ有効です。

プログラム内のまだ読み込まれていないクラスは出力されません。

構文

`jclasses` dbx で認識されているすべての Java クラスの名前を出力します。

`jclasses -a` システムクラスおよびその他の認識されている Java クラスを出力します。

joff コマンド

joff コマンドは、Java モードまたは JNI モードからネイティブモードに dbx を切り替えます。

構文

`joff`

jon コマンド

jon コマンドは、ネイティブモードから Java モードに dbx を切り替えます。

構文

```
jon
```

jpgks コマンド

jpgks コマンドは、コマンド実行時に dbx で認識されているすべての Java パッケージの名前を出力します。Java モードでだけ有効です。

プログラム内のまだ読み込まれていないパッケージは出力されません。

構文

```
jpgks
```

kill コマンド

kill コマンドはプロセスにシグナルを送ります。ネイティブモードでだけ有効です。

構文

kill -l	既知の全シグナルの番号、名前、説明を一覧表示します。
kill	制御対象プロセスを終了します。
kill job ...	一覧表示されているジョブに SIGTERM シグナルを送ります。
kill -signal job ...	一覧表示されているジョブに指定のシグナルを送ります。

ここで

job としてプロセス ID を指定するか、または次のいずれかの方法で指定します。

%+	現在のジョブを終了します。
%-	直前のジョブを終了します。

`%number` `number` の番号を持つジョブを終了します。

`%string` `string` で始まるジョブを終了します。

`/?string` `string` を含んでいるジョブを終了します。

`signal` はシグナル名です。

language コマンド

language コマンドは、現在のソース言語の表示や変更を行います。ネイティブモードでだけ有効です。

構文

language dbx language_mode 環境変数 (62 ページの「dbx 環境変数の設定」参照) によって設定される現在の言語モードを出力します。言語モードが `autodetect` または `main` に設定されている場合は、式の解析と評価に使用されている現在の言語の名前も出力されます。

ここで

`language` は、`c`、`c++`、`fortran`、または `fortran90` です。

注 `-c` は、`ansic` の別名です。

line コマンド

line コマンドは、現在の行番号の表示や変更を行います。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

line 現在の行番号を表示します。

line *number* 現在の行番号として *number* を設定します。

line "*file_name*" 現在の行番号として行 1 を *file_name* に設定します。

line "*file_name*": *number* 現在の行番号として行 *number* を *file_name* に設定します。

ここで

file_name は、変更対象の行番号があるファイルの名前です。ファイル名を囲んでいる "" は省略可能です。

number は、ファイル内の行の番号です。

例

```
line 100
line "/root/test/test.cc":100
```

list コマンド

list コマンドは、ソースファイルの行を表示します。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

デフォルト表示行数 *N* は、`dbx` 環境変数 `output_list_size` によって制御されます。

構文

list	<i>N</i> 行を一覧表示します。
list <i>number</i>	行番号 <i>number</i> を表示します。
list +	次の <i>N</i> 行を一覧表示します。
list + <i>n</i>	次の <i>n</i> 行を一覧表示します。
list -	直前の <i>N</i> 行を一覧表示します。
list - <i>n</i>	直前の <i>n</i> 行を一覧表示します。
list <i>n1</i> , <i>n2</i>	<i>n1</i> から <i>n2</i> までの行を一覧表示します。
list <i>n1</i> , +	<i>n1</i> から <i>n1</i> + <i>N</i> までを一覧表示します。
list <i>n1</i> , + <i>n2</i>	<i>n1</i> から <i>n1</i> + <i>n2</i> までを一覧表示します。
list <i>n1</i> , -	<i>n1</i> - <i>N</i> から <i>n1</i> . までを一覧表示します。
list <i>n1</i> , - <i>n2</i>	<i>n1</i> - <i>n2</i> から <i>n1</i> までを一覧表示します。
list <i>function</i>	<i>function</i> のソースの先頭を表示します。list <i>function</i> は、現在のスコープを変更します。詳細については、 72 ページの「プログラマースコープ」 を参照してください。
list <i>file_name</i>	ファイル <i>file_name</i> の先頭を表示します。

`list file_name: n` ファイル *filename* を行 *n* から表示します。ファイルの末尾行を示す '\$' を行番号の代わりに使用できます。コンマは省略可能です。

ここで

file_name は、ソースコードファイルの名前です。

function は、表示対象の関数の名前です。

number は、ソースファイル内の行の番号です。

n は、表示対象の行数です。

n1 は、最初に表示する行の番号です。

n2 は、最後に表示する行の番号です。

オプション

`-i` または `-instr` ソース行とアセンブリコードを混合します。

`-w` または `-wn` 行または関数のまわりの *N* (または *n*) 行を一覧表示します。このオプションを '+' 構文または '-' 構文と併用したり 2 つの行番号が指定されているときに使用したりすることはできません。

`-a` 関数名と使用した場合、関数全体を一覧表示します。パラメータなしで使用した場合、現在の関数に残りがあると、その残りを一覧表示します。

例

```
list           // 現在の行を先頭とする N 行を一覧表示する
list +5       // 現在の行を先頭とする 5 行を一覧表示する
list -        // 直前の N 行を一覧表示する
list -20      // 直前の 20 行を一覧表示する
list 1000     // 行 1000 を表示する
list 1000,$   // 行 1000 から末尾行までを一覧表示する
list 2737 +24 // 行 2737 と次の 24 行を一覧表示する
list 1000 -20 // 行 980 から 1000 までを一覧表示する
list test.cc:33 // ファイル test.cc のソース行 33 を表示する
list -w      // 現在行のまわりの N 行を一覧表示する
list -w8 'test.cc'func1 // 関数 func1 のまわりの 8 行を一覧表示する
list -i 500 +10 // 行 500 から 510 までを一覧表示する
                (ソースとアセンブリコード)
```

listi コマンド

listi コマンドは、ソース命令と逆アセンブリされた命令を表示します。ネイティブモードでだけ有効です。

詳細については、329 ページの「list コマンド」を参照してください。

loadobject コマンド

loadobject コマンドは、現在のロードオブジェクトの名前を出力します。ネイティブモードでだけ有効です。

構文

loadobject -list [<i>regex</i>] [-a]	読み込まれているロードオブジェクトを表示します (333 ページの「loadobject -list コマンド」参照)。
load -load <i>loadobject</i>	指定したロードオブジェクトのシンボルを読み込みます (334 ページの「loadobject -load コマンド」参照)。
load -unload [<i>regex</i>]	指定したロードオブジェクトの読み込みを解除します (334 ページの「loadobject -unload コマンド」参照)。
load -hide [<i>regex</i>]	dbx の検索アルゴリズムからロードオブジェクトを削除します (333 ページの「loadobject -hide コマンド」参照)。
loadobject -use [<i>regex</i>]	dbx の検索アルゴリズムにロードオブジェクトを追加します (335 ページの「loadobject -use コマンド」参照)。
load -dumpelf [<i>regex</i>]	ロードオブジェクトの ELF 情報を表示します (332 ページの「loadobject -dumpelf コマンド」参照)。
loadobject -exclude <i>ex_regex</i>	<i>ex_regex</i> に一致するロードオブジェクトを自動的に読み込まないように指定します (332 ページの「loadobject -exclude コマンド」参照)。
loadobject exclude -clear	除外パターンのリストをクリアします (332 ページの「loadobject -exclude コマンド」参照)。

ここで

regex は正規表現です。指定していない場合は、コマンドがすべてのロードオブジェクトに適用されます。

ex-regex は省略できません。

このコマンドには、別名 *lo* がデフォルトで設定されています。

loadobject -dumpelf コマンド

`loadobject -dumpelf` コマンドは、ロードオブジェクトのさまざまな ELF の詳細情報を表示します。ネイティブモードでだけ有効です。

構文

```
loadobject -dumpelf [regex]
```

ここで

regex は正規表現です。指定していない場合は、コマンドがすべてのロードオブジェクトに適用されます。

このコマンドは、ディスク上のロードオブジェクトの ELF 構造に関する情報をダンプします。この出力の詳細は、今後変更される可能性があります。この出力を解析する場合は、Solaris OS のコマンドである `dump` または `elfdump` を使用してください。

loadobject -exclude コマンド

`loadobject -exclude` コマンドは、指定した正規表現に一致するロードオブジェクトを自動的に読み込まないように指定します。

構文

```
loadobject -exclude ex_regex [-clear]
```

ここで

ex_regex は正規表現です。

このコマンドは、指定した正規表現に一致するロードオブジェクトのシンボルを `dbx` で自動的に読み込まないように指定します。ほかの `loadobject` のサブコマンドでの *regex* とは異なり、*ex_regex* を指定しない場合は、すべてのロードオブジェクトを

対象に処理が実行されることはありません。 *ex-regexp* を指定しない場合は、このコマンドは前の `loadobject -exclude` コマンドで指定した除外パターンを表示します。

`-clear` を指定した場合は、除外パターンのリストが削除されます。

現時点では、この機能を使用してメインプログラムや実行時リンカーを読み込まないように指定することはできません。また、このコマンドを使用して C++ 実行時ライブラリを読み込まないように指定すると、C++ の一部の機能が正常に機能しなくなります。

このオプションは、実行時チェック (RTC) では使用しないでください。

loadobject -hide コマンド

`loadobject -hide` コマンドは、`dbx` の検索アルゴリズムからロードオブジェクトを削除します。

構文

```
loadobject -hide [regexp]
```

ここで

regexp は正規表現です。指定していない場合は、コマンドがすべてのロードオブジェクトに適用されます。

このコマンドは、プログラムのスコープからロードオブジェクトを削除し、その関数およびシンボルを `dbx` で認識しないように設定します。また、このコマンドは、`'preload'` ビットをリセットします。

loadobject -list コマンド

`loadobject -list` コマンドは、読み込まれているロードオブジェクトを表示します。ネイティブモードでだけ有効です。

構文

```
loadobject -list [regexp] [-a]
```

ここで

regexp は正規表現です。指定していない場合は、コマンドがすべてのロードオブジェクトに適用されます。

各ロードオブジェクトのフルパス名が表示されます。また、余白部分には状態を示す文字が表示されます。隠されたロードオブジェクトは、`-a` オプションを指定した場合のみリスト表示されます。

`h` "hidden" を意味します (シンボルは、`whatis` や `stop in` などのシンボル照会では検出されません)。

`u` 有効なプロセスがある場合、`u` は "unmapped" を意味します。

`p` この文字は、事前に読み込まれた LO、つまり `'loadobject -load'` コマンドまたはプログラムの `dlopen` イベントの結果を示します。

たとえば、次のようにします。

```
(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)
```

最後の例は、実行時リンカーのシンボルがデフォルトでは隠されていることを示します。これらのシンボルを `dbx` コマンドで使用するには、次の [335 ページ](#) の「`loadobject -use` コマンド」を使用します。

loadobject -load コマンド

`loadobject -load` コマンドは、指定したロードオブジェクトのシンボルを読み込みます。ネイティブモードでだけ有効です。

構文

```
loadobject -list [regexp] [-a]
```

ここで

`loadobject` には、フルパス名または `/usr/lib` または `/usr/lib/sparcv9` 内のライブラリを指定します。デバッグ中のプログラムがある場合は、該当する ABI ライブラリのディレクトリだけが検索されます。

loadobject -unload コマンド

`loadobject -unload` コマンドは、指定したロードオブジェクトを読み込み解除します。ネイティブモードでだけ有効です。

構文

```
loadobject -unload [regex]
```

ここで

regex は正規表現です。指定していない場合は、コマンドがすべてのロードオブジェクトに適用されます。

このコマンドは、コマンド行で指定した *regex* に一致するすべてのロードオブジェクトのシンボルを読み込み解除します。debug コマンドで読み込んだ主プログラムは読み込み解除できません。また、使用中のロードオブジェクトや、dbx が正常に動作するために必要なロードオブジェクトの読み込み解除もできない場合があります。

loadobject -use コマンド

loadobject -use コマンドは、dbx の検索アルゴリズムにロードオブジェクトを追加します。ネイティブモードでだけ有効です。

構文

```
loadobject -use [regex]
```

ここで

regex は正規表現です。指定していない場合は、コマンドがすべてのロードオブジェクトに適用されます。

lwp コマンド

lwp コマンドは、現在の LWP (軽量プロセス) の表示や変更を行います。ネイティブモードでだけ有効です。

注 - lwp コマンドは Solaris プラットフォームでのみ利用可能です。

構文

lwp	現在の LWP を表示します。
lwp <i>lwp_id</i>	LWP <i>lwp_id</i> に切り替えます。

<code>lwp -info</code>	現在の LWP の名前、ホーム、およびマスクシグナルを表示します。
<code>lwp [lwp_id] -setfp address_expression</code>	fp レジスタに <i>address_expression</i> の値が入っていることを dbx に伝えます。デバッグするプログラムの状態は変更されません。assign \$fp=... が利用できないコアファイルをデバッグします。-setfp オプションで設定されたフレームポインタは、実行を再開するときに元の値にリセットされます。
<code>lwp [lwp_id] -resetfp</code>	このコマンドより前で使用された lwp -setfp コマンドの設定を元に戻して、フレームポインタの論理値を現在のプロセスまたはコアファイルのレジスタ値から設定します。

ここで

lwp_id は軽量プロセスの識別子です。

コマンドが *lwp_id* とオプションの両方と使用された場合、対応するアクションは *lwp_id* によって指定された LWP に適用されますが、現在の LWP は変更されません。

-setfp と -resetfp オプションは、LWP のフレームポインタ (fp) が破損した場合に便利です。このイベントでは、dbx は呼び出しスタックを適切に再構築できず、局所変数を評価できません。これらのオプションは assign \$fp=... が利用できないコアファイルのデバッグ時に機能します。

fp への変更を、デバッグするアプリケーションに見えるようにするには、assign \$fp=*address_expression* コマンドを使用します。

lwps コマンド

lwps コマンドは、プロセス内の LWP (軽量プロセス) すべてを一覧表示します。ネイティブモードでだけ有効です。

注 - lwps コマンドは Solaris プラットフォームでのみ利用可能です。

構文

`lwps` 現在のプロセス内の LWP すべてを一覧表示します。

mmapfile コマンド

`mmapfile` コマンドは、コアダンプに存在しないメモリーマップファイルの内容を表示します。ネイティブモードでだけ有効です。

Solaris コアファイルには、読み取り専用のメモリーセグメントは含まれていません。実行可能な読み取り専用セグメント(つまりテキスト)は自動的に処理され、`dbx` は、実行可能ファイルと関連する共有オブジェクトを調べることによってこれらのセグメントに対するメモリーアクセスを解釈処理します。

構文

`mmapfile mmapmed_file address offset length` コアダンプに存在しないメモリーマップファイルの内容を表示します。

ここで

`mmapmed_file` は、コアダンプ中にメモリーマップされたファイルのファイル名です。

`address` は、プロセスのアドレス空間の開始アドレスです。

`length` は、表示対象アドレス空間のバイト単位による長さです。

`offset` は、`mmapmed_file` の開始アドレスまでのバイト単位によるオフセットです。

module コマンド

`module` コマンドは、1個または複数のモジュールのデバッグ情報を読み込みます。ネイティブモードでだけ有効です。

構文

`module [-v]` 現在のモジュールの名前を出力します。

`module [-f] [-v] [-q] name` `name` というモジュールのデバッグ情報を読み込みます。

`module [-f] [-v] [-q] -a` 全モジュールのデバッグ情報を読み込みます。

ここで

name は、読み込み対象のデバッグ情報が関係するモジュールの名前です。

`-a` は、すべてのモジュールを指定します。

`-f` は、実行可能ファイルより新しいファイルの場合でもデバッグ情報の読み込みを強制します (使用にあたっては十分に注意してください)。

`-v` は、言語、ファイル名などを出力する冗長モードを指定します。

`-q` は、静止モードを指定します。

例

読み取り専用データセグメントは、アプリケーションメモリーがデータベースをマップしたときに通常発生します。たとえば、次のようにします。

```
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;
```

デバッガによってメモリーとしてデータベースにアクセスできるようにするには、次を入力します。

```
mmapfile ../DATABASE $[vaddr] $[offset] $[size]
```

ここで、次を入力すれば、データベースの内容を構造的に表示させることができます。

```
print *index
```

modules コマンド

`modules` コマンドは、モジュール名を一覧表示します。ネイティブモードでだけ有効です。

構文

<code>modules [-v]</code>	すべてのモジュールを一覧表示します。
<code>modules [-v] -debug</code>	デバッグ情報が入っているモジュールすべてを一覧表示します。
<code>modules [-v] -read</code>	すでに読み込まれたデバッグ情報が入っているモジュールの名前を表示します。

ここで

`-v` は、言語、ファイル名などを出力する冗長モードを指定します。

native コマンド

`native` コマンドは、`dbx` が Java モードの場合に、指定したコマンドのネイティブバージョンを実行するように指定します。コマンドの前に `"native"` を指定すると、`dbx` はそのコマンドをネイティブモードで実行します。つまり、式が C または C++ の式として解釈および表示され、一部のコマンドでは Java モードの場合と異なる出力が生成されます。

このコマンドは、Java コードをデバッグしていて、ネイティブ環境を調べる必要があるときに便利です。

構文

`native command`

ここで

`command` は、実行対象コマンドの名前および引数です。

next コマンド

`next` コマンドは、1 ソース行をステップ実行します (呼び出しをステップオーバー)。

`dbx` の環境変数 `step_events` (62 ページの「`dbx` 環境変数の設定」参照) は、ステップ実行中にブレークポイントが使用可能であるかどうかを制御します。

ネイティブモードの構文

<code>next</code>	1行をステップ実行します (呼び出しをステップオーバー)。関数呼び出しがステップオーバーされるマルチスレッドプログラムの場合、デッドロック状態を避けるため、その関数呼び出し中は全 LWP (軽量プロセス) が暗黙に再開されます。非活動状態のスレッドをステップ実行することはできません。
<code>next n</code>	n 行をステップ実行します (呼び出しをステップオーバー)。
<code>next ... -sig signal</code>	ステップ実行中に指定のシグナルを引き渡します。
<code>next ... thread_id</code>	指定のスレッドをステップ実行します。
<code>next ... lwp_id</code>	指定の LWP をステップ実行します。関数をステップオーバーしたときに全 LWP を暗黙に再開しません。

ここで

n は、ステップ実行対象の行数です。

`signal` はシグナル名です。

`thread_id` は、スレッド ID です。

`lwp_id` は、LWP ID です。

明示的な `thread_id` または `lwp_id` が指定されている場合、`next` コマンドによる汎用のデッドロック回避は無効となります。

マシンレベルの呼び出しステップオーバーについては、[341 ページの「nexti コマンド」](#) も参照してください。

注 - 軽量プロセス (LWP) の詳細については、Solaris の『マルチスレッドのプログラミング』を参照してください。

Java モードの構文

<code>next</code>	1行をステップ実行します (呼び出しをステップオーバー)。関数呼び出しがステップオーバーされるマルチスレッドプログラムの場合、デッドロック状態を避けるため、その関数呼び出し中は全 LWP (軽量プロセス) が暗黙に再開されます。非活動状態のスレッドをステップ実行することはできません。
<code>next n</code>	n 行をステップ実行します (呼び出しをステップオーバー)。

`next ... thread_id` 指定のスレッドをステップ実行します。

`next ... lwp_id` 指定のLWPをステップ実行します。関数をステップオーバーしたときに全LWPを暗黙に再開しません。

ここで

n は、ステップ実行対象の行数です。

thread_id は、スレッド識別子です。

lwp_id は、LWP 識別子です。

明示的な *thread_id* または *lwpid* が指定されている場合、`next` コマンドによる汎用のデッドロック回避は無効となります。

注 - 軽量プロセス (LWP) の詳細については、Solaris の『マルチスレッドのプログラミング』を参照してください。

nexti コマンド

`nexti` コマンドは、1 マシン命令をステップ実行します (呼び出しをステップオーバー)。ネイティブモードでだけ有効です。

構文

`nexti` マシン命令 1 個をステップ実行します (呼び出しをステップオーバー)。

`nexti n` *n* 行をステップ実行します (呼び出しをステップオーバー)。

`nexti -sig signal` ステップ実行中に指定のシグナルを引き渡します。

`nexti ... lwp_id` 指定の LWP をステップ実行します。

`nexti ... thread_id` 指定のスレッドが活動状態である LWP をステップ実行します。関数をステップオーバーしたときに全 LWP を暗黙に再開しません。

ここで、

n は、ステップ実行対象の命令数です。

signal はシグナル名です。

thread_id は、スレッド ID です。

lwp_id は、LWP ID です。

pathmap コマンド

pathmap コマンドは、ソースファイルを検索する場合などに1つのパス名を別のパス名にマッピングします。マッピングは、ソースパス、オブジェクトファイルパス、および現在の作業用ディレクトリ (-c を指定した場合) に適用されます。pathmap コマンドの構文および機能は、ネイティブモードと Java モードで同一です。

pathmap コマンドは、さまざまなホスト上に存在するさまざまなパスを持つ、オートマウントされた明示的な NFS マウント済みファイルシステムを取り扱うときに便利です。オートマウントされたファイルシステムにおける CWD も不正確であるため、オートマウントが原因である問題を解決する際には、-c を指定します。pathmap コマンドは、ソースツリーやビルドツリーを移動した場合にも便利です。

デフォルトの場合、pathmap /tmp_mnt / が存在します。

pathmap コマンドは、dbx 環境変数 `core_lo_pathmap` が on に設定されているときにロードオブジェクトを検索します。前述の場合以外では、pathmap コマンドはロードオブジェクト (共有ライブラリ) の検索に対して効果がありません。46 ページの「一致しないコアファイルのデバッグ」を参照してください。

構文

pathmap [-c] [-index] *from to* *from* から *to* への新たなマッピングを作成します。

pathmap [-c] [-index] *to* すべてのパスを *to* にマッピングします。

pathmap 既存のパスマッピングすべてを一覧表示します (インデックス別に)。

pathmap -s 前述と同じですが、出力を dbx によって読み込むことができます。

pathmap -d *from1 from2 ...* 指定のマッピングをパス単位で削除します。

pathmap -d *index1 index2 ...* 指定のマッピングをインデックス単位で削除します。

ここで

from と *to* は、ファイルパス接頭辞です。*from* は実行可能ファイルやオブジェクトファイルにコンパイルされたファイルパス、*to* はデバッグ時におけるファイルパスを示します。

from1 は、最初に削除するマッピングのファイルパスです。

from2 は、最後に削除するマッピングのファイルパスです。

index は、マッピングをリストに挿入する際に使用するインデックスを指定します。インデックスを指定しなかった場合、リスト末尾にマッピングが追加されます。

index1 は、最初に削除するマッピングのインデックスです。

index2 は、最後に削除するマッピングのインデックスです。

-c を指定すると、現在の作業用ディレクトリにもマッピングが適用されます。

-s を指定すると、dbx が読み込める出力形式で既存のマッピングがリストされます。

-d を指定すると、指定のマッピングが削除されます。

例

```
(dbx) pathmap /export/home/work1 /net/mmm/export/home/work2
# /export/home/work1/abc/test.c を /net/mmm/export/home/work2/abc/test.c ヘマッブ
(dbx) pathmap /export/home/newproject
# /export/home/work1/abc/test.c を /export/home/newproject/test.c ヘマッブ
(dbx) pathmap
(1) -c /tmp_mnt /
(2) /export/home/work1 /net/mmm/export/home/work2
(3) /export/home/newproject
```

pop コマンド

pop コマンドは、1 個または複数のフレームを呼び出しスタックから削除します。ネイティブモードでだけ有効です。

-g を使ってコンパイルされた関数の場合、フレームにポップできるだけです。プログラムカウンタは、呼び出し場所におけるソース行の先頭にリセットされます。デバッガによる関数呼び出しを越えてポップすることはできません。pop -c を使用してください。

通常、pop コマンドはポップ対象フレームに関する C++ デストラクタをすべて呼び出します。dbx 環境変数 pop_auto_destruct を off に設定すれば、この動作を変更できます(62 ページの「dbx 環境変数の設定」参照)。

構文

<code>pop</code>	現在のトップフレームをスタックからポップします。
<code>pop number</code>	<i>number</i> 個のフレームをスタックからポップします。
<code>pop -f number</code>	現在のフレーム <i>number</i> までフレームをスタックからポップします。
<code>pop -c</code>	デバッガが行なった最後の呼び出しをポップします。

ここで

number は、スタックからポップするフレームの数です。

print コマンド

ネイティブモードでは、`print` コマンドは式の値を出力します。Java モードでは、`print` コマンドは式、局所変数、パラメータの値を出力します。

ネイティブモードの構文

<code>print expression, ...</code>	式 <i>expression, ...</i> の値を出力します。
<code>print -r expression</code>	継承メンバーを含み、式 <i>expression</i> の値を出力します (C++ のみ)。
<code>print +r expression</code>	dbx 環境変数 <code>output_inherited_members</code> が on であるときは、継承メンバーを出力しません (C++ のみ)。
<code>print -d [-r] expression</code>	式 <i>expression</i> の静的型ではなく動的型を表示します (C++ のみ)。
<code>print +d [-r] expression</code>	dbx 環境変数 <code>output_dynamic_type</code> が on であるときは、式 <i>expression</i> の動的型を使用しません (C++ のみ)。
<code>print -p expression</code>	<code>prettyprint</code> 関数を呼び出します。
<code>print +p expression</code>	dbx 環境変数 <code>output_pretty_print</code> が on であるときは、 <code>prittyprint</code> 関数を呼び出しません。
<code>print -L expression</code>	出力オブジェクト <i>expression</i> が 4K を超える場合は、出力を強制実行します。
<code>print +l expression</code>	式が文字列である場合 (char *)、アドレスの出力のみを行い、文字を出力しません。

<code>print -l expression</code>	(Literal) 左側を出力しません。式が文字列である場合 (char *)、アドレスの出力は行わず、文字列内の文字だけを引用符なしで出力します。
<code>print -fformat expression</code>	整数、文字列、浮動小数点の式の形式として <i>format</i> を使用します (オンラインヘルプの <i>format</i> 参照)。
<code>print -Fformat expression</code>	指定の形式を使用しますが、左側 (変数名や式) は出力しません (オンラインヘルプの <i>format</i> 参照)。
<code>print -o expression</code>	<i>expression</i> の値を出力します。これは、序数としての列挙式でなければなりません。ここでは、形式文字列を使用することもできます (- <i>fformat</i>)。非列挙式の場合、このオプションは無視されます。
<code>print -- expression</code>	'-' は、フラグ引数の終わりを示します。これは、 <i>expression</i> がプラスやマイナスで始まる可能性がある場合に便利です。スコープ解釈処理ルールについては、 72 ページの「プログラムスコープ」 を参照してください。

ここで

expression は、出力対象の値を持つ式です。

format は、式の出力時に使用する形式です。形式が指定の型に適用しない場合は、形式文字列は無視され、内蔵出力機構が使用されます。

許可されている形式は `printf(3S)` コマンドで使用されているもののサブセットです。次の制限が適用されます。

- *n* 変換できません。
- フィールド幅または精度に * を使用できません。
- %<桁>\$ 引数を選択できません。
- 1つの形式文字列に対して1つの変換指定のみが可能です。

許可されている形式は、次の簡易文法で定義されます。

FORMAT ::= CHARS % FLAGS WIDTH PREC MOD SPEC CHARS

CHARS ::= <% を含まない任意の文字シーケンス >

| %%

| <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

```

指定した形式文字列が%を含まない場合は、dbxによって自動的に付加されます。形式文字列がスペース、セミコロン、またはタブを含んでいる場合は、形式文字列全体を二重引用符で囲む必要があります。

Java モードの構文

<code>print expression,</code>	式 <i>expression</i> , ... または識別子 <i>identifier</i> , ... の値を出力します。
<code>print -r expression identifier</code>	継承メンバーを含み、 <i>expression</i> または識別子 <i>identifier</i> の値を出力します。
<code>print +r expression identifier</code>	dbx 環境変数 <code>output_inherited_members</code> が on であるときは、継承メンバーを出力しません。
<code>print -d [-r] expression identifier</code>	式 <i>expression</i> または識別子 <i>identifier</i> の、静的型ではなく動的型を表示します。
<code>print +d [-r] expression identifier</code>	dbx 環境変数 <code>dbx output_dynamic_type</code> が on であるときは、式 <i>expression</i> の動的型または識別子 <i>identifier</i> の値は使用しないでください。
<code>print -- expression identifier</code>	'--' は、フラグ引数の終わりを示します。これは、 <i>expression</i> がプラスやマイナスで始まる可能性がある場合に便利です。スコープ解釈処理ルールについては、72 ページの「プログラムスコープ」を参照してください。

ここで

class_name は、Java クラス名で、パッケージのパス (". "(ピリオド) を修飾子として使用。たとえば `test1.extra.T1.Inner`) またはフルパス名 (# 記号で始まり、"/" (スラッシュ) や \$ を修飾子として使用。たとえば `#test1/extra/T1$Inner`) のいずれかで指定します。修飾子 \$ を使用する場合は、*class_name* を引用符で囲みます。

expression は、値を出力する Java 式です。

field_name は、クラス内のフィールド名です。

identifier は、局所変数またはパラメータです。これには、`this`、現在のクラスのインスタンス変数 (*object_name.field_name*)、クラス (static) 変数 (*class_name.field_name*) が含まれます。

object_name は、Java オブジェクトの名前です。

proc コマンド

proc コマンドは、現在のプロセスの状態を表示します。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

proc -map ロードオブジェクトのリストおよびアドレスを表示します。

proc -pid 現在のプロセス ID (pid) を表示します。

prog コマンド

prog コマンドは、デバッグ中のプログラムとその属性を管理します。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

prog -readsyms 据え置きされていた記号情報を、`dbx` 環境変数 `run_quick` を on に設定することによって読み込みます。

prog -executable - の使用がプログラムに設定されている場合、実行可能ファイルのフルパス - を出力します。

prog -argv argv[0] を含む argv 全体を出力します。

prog -args argv[0] を含まない argv を出力します。

prog -stdin <filename> を出力します。stdin が使用されている場合は、空にします。

prog -stdout >filename または >>filename を出力します。stdout が使用されている場合は、空にします。-args、-stdin、-stdout の出力は、組み合わせて run コマンドで使用できるようになっています (352 ページの「run コマンド」参照)。

quit コマンド

quit コマンドは、dbx を終了します。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

dbx がプロセスに接続されている場合、このプロセスを切り離してから終了が行われます。保留状態のシグナルは取り消されます。微調整を行うには、detach コマンドを使用します (311 ページの「detach コマンド」参照)。

構文

quit リターンコード 0 を出力して dbx を終了します。exit と同じです。

quit *n* リターンコード *n* を出力して終了します。exit *n* と同じです。

ここで

n は、リターンコードです。

regs コマンド

regs コマンドは、レジスタの現在値を出力します。ネイティブモードでだけ有効です。

構文

regs [-f] [-F]

ここで

-f には、浮動小数点レジスタ (単精度) が含まれます (SPARC プラットフォームのみ)。

-F には、浮動小数点レジスタ (倍精度) が含まれます (SPARC プラットフォームのみ)。

例 (SPARC プラットフォーム)

```
dbx[13] regs -F
現スレッド: t@1
現フレーム: [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          +0.000000000000000e+00
f2f3          +0.000000000000000e+00
f4f5          +0.000000000000000e+00
f6f7          +0.000000000000000e+00

```

replay コマンド

replay コマンドは、最後の run、rerun、または debug コマンド以降のデバッグコマンドを再現します。ネイティブモードでだけ有効です。

構文

replay [-number] 最後の run コマンド、rerun コマンド、または debug コマンド以降のコマンドすべてを再現するか、またはそれらのコマンドから *number* 個のコマンドを差し引いたコマンドを再現します。

ここで

number は、再現しないコマンドの数です。

rerun コマンド

rerun コマンドは、引数を付けずにプログラムを実行します。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

rerun 引数を付けずにプログラムの実行を開始します。

`rerun arguments` `save` コマンドで新しい引数を付けてプログラムの実行を開始します (353 ページの「[save コマンド](#)」参照)。

restore コマンド

`restore` コマンドは、以前に保存されていた状態に `dbx` を復元します。ネイティブモードでだけ有効です。

構文

```
restore [file_name]
```

ここで

`file_name` は、最後の `run` コマンド、`rerun` コマンド、または `debug` コマンドが保存されてから実行された `dbx` コマンドの実行対象ファイルの名前です。

rprint コマンド

`rprint` コマンドは、シェル引用規則を使用して式を出力します。ネイティブモードでだけ有効です。

構文

```
rprint [-r|+r|-d|+d|-p|+p|-L|-l|-f format | -Fformat | -- ] expression
```

式の値を出力します。特別な引用規則は適用されないので、`rprint a > b` の場合、`a` の値 (存在する場合) がファイル `b` に入れます (フラグの意味については [344 ページの「print コマンド」](#) 参照)。

ここで

`expression` は、出力対象の値を持つ式です。

`format` は、式の出力時に使用する形式です。詳細については、[344 ページの「print コマンド」](#) を参照してください。

rtc showmap コマンド

rtc showmap コマンドは、計測種類 (分岐またはトラップ) で分類されるプログラムのアドレス範囲をレポートします。ネイティブモードでだけ有効です。

注 - rtc showmap コマンドが実行できるのは、Solaris プラットフォームのみです。

構文

rtc showmap

このコマンドは、経験豊富なユーザー向けのコマンドで、dbx の内部デバッグを実行します。実行時チェックは、プログラムのテキストを計測してアクセスチェックを行います。計測種類として、使用可能なリソースに応じて、分岐またはトラップの命令を指定することができます。rtc showmap コマンドは、計測種類で分類されるプログラムのアドレス範囲をレポートします。このマップを使用して、パッチ領域オブジェクトファイルを追加するのに最適な場所を特定し、トラップの自動使用を回避することができます。詳細は、[155 ページの「実行時検査の制限」](#)を参照してください。

rtc skippatch コマンド

rtc skippatch コマンドは、ロードオブジェクト、オブジェクトファイル、および関数に対して、実行時検査による計測は行わないようにします。コマンドの効果は、ロードオブジェクトが明示的に除外されないかぎり各 dbx セッションで有効なまま保持されます。

このコマンドの効果により、dbx はロードオブジェクト、オブジェクトファイル、および関数でメモリアクセスを追跡しないため、スキップされない関数では、不正な rui エラーが報告されることがあります。このコマンドによって rui エラーが導かれたかどうかを dbx は特定できないため、このようなエラーは自動で抑制されません。

構文

rtc skippatch *load_object* ...

指定したオブジェクトを計測から除外します。

rtc skippatch *load_object* [-o *object_file* ...] [-f *function* ...]

指定したロードオブジェクト内の指定したオブジェクトファイルや関数を計測から除外します。

ここで

load_object はロードオブジェクトの名前またはロードオブジェクトへのパスです。

object_file は、オブジェクトファイルの名前です。

function は、関数の名前です。

run コマンド

run コマンドは引数を付けてプログラムを実行します。

Control-Cを使用すると、プログラムの実行が停止します。

ネイティブモードの構文

run 現在の引数を付けてプログラムの実行を開始します。

run *arguments* 新規の引数を付けてプログラムの実行を開始します。

run ... >|>> *output_file* 出力先の切り替えを設定します。

run ... < *input_file* 入力元の切り替えを設定します。

ここで

arguments はターゲットプロセスの実行時に使用する引数です。

input_file は、入力元ファイルの名前です。

output_file は、出力先ファイルの名前です。

注-現在、run コマンドや runargs コマンドによって stderr の出力先を切り替えることはできません。

Java モードの構文

run 現在の引数を付けてプログラムの実行を開始します。

run *arguments* 新規の引数を付けてプログラムの実行を開始します。

ここで

arguments はターゲットプロセスの実行に使用される引数です。これらの引数は、Java アプリケーション (JVM ソフトウェアではありません) に渡されます。main クラス名を引数として含めないでください。

Java アプリケーションの入力または出力を `run` コマンドでリダイレクトすることはできません。

一回の実行で設定したブレイクポイントは、それ以降の実行でも有効になります。

runargs コマンド

`runargs` コマンドは、ターゲットプロセスの引数を変更します。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

ターゲットプロセスの現在の引数を調べるには、引数を付けずに `debug` コマンドを使用します (307 ページの「`debug` コマンド」参照)。

構文

<code>runargs arguments</code>	<code>run</code> コマンドで使用する現在の引数を設定します (352 ページの「 <code>run</code> コマンド」参照)。
<code>runargs ... > >>file</code>	<code>run</code> コマンドで使用する出力先を設定します。
<code>runargs ... <file</code>	<code>run</code> コマンドで使用する入力元を設定します。
<code>runargs</code>	現在の引数をクリアします。

ここで

arguments はターゲットプロセスの実行に使用される引数です。

file は、ターゲットプロセスからの出力またはターゲットプロセスへの入力の切り替え先です。

save コマンド

`save` コマンドは、コマンドをファイルに保存します。ネイティブモードでだけ有効です。

構文

`save [-number] [file_name]` 最後の run コマンド、rerun コマンド、または debug コマンド以降のコマンドすべて、またはそれらのコマンドから *number* 個のコマンドを差し引いたコマンドを、デフォルトファイルまたは *file_name* に保存します。

ここで

number は、保存しないコマンドの数です。

filename は、最後の run コマンド、rerun コマンド、または debug コマンドのあとに実行される dbx コマンドを保存するファイルの名前です。

scopes コマンド

scopes コマンドは、活動状態にあるスコープのリストを出力します。ネイティブモードでだけ有効です。

構文

scopes

search コマンド

search コマンドは、現在のソースファイルにおいて順方向検索を行います。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

`search string` 現在のファイルの中で、*string* を順方向で検索します。

`search` 最後の検索文字列を使用して検索を繰り返します。

ここで

string は、検索対象の文字列です。

showblock コマンド

showblock コマンドは、特定のヒープブロックが割り当てられた場所を示す実行時検査結果を表示します。ネイティブモードでだけ有効です。

注 - showblock コマンドは Solaris プラットフォームでのみ利用可能です。

メモリー使用状況検査やメモリーリーク検査がオンになっているときに showblock コマンドを使用すると、指定アドレスのヒープブロックに関する詳細が表示されます。この詳細情報では、ブロックの割り当て場所とサイズを知ることができます。[292 ページの「check コマンド」](#)を参照してください。

構文

```
showblock -a address
```

ここで

address は、ヒープブロックのアドレスです。

showleaks コマンド

showleaks コマンドは、最後の showleaks コマンド実行後のメモリーリークについて報告します。ネイティブモードでだけ有効です。

注 - showleaks コマンドは Solaris プラットフォームでのみ利用可能です。

デフォルトの簡易形式では、1 行に 1 つのリークレコードを示すレポートが出力されます。実際に発生したリークのあとに、発生する可能性のあるリークが報告されます。レポートは、リークのサイズによってソートされます。

構文

```
showleaks [-a] [-m m] [-n number] [-v]
```

ここで

-a は、これまでに発生したリークすべてを表示します (最後の showLeaks コマンドを実行したあとのリークだけではなく)。

-m *m* は、複数のリークをまとめます。2 個以上のリークに対する割り当て時の呼び出しスタックが *m* 個のフレームに一致するとき、これらのリークは 1 つのリークレポートにまとめて報告されます。-m オプションを指定すると、check コマンドで指定した *m* の大域値が無効となります (292 ページの「check コマンド」参照)。

-n *number* は、最大 *number* 個のレコードをレポートに表示します。デフォルトの場合、すべてのレコードが表示されます。

-v 冗長出力を生成します。デフォルトの場合、簡易出力が表示されます。

showmemuse コマンド

showmemuse コマンドは、最後の showmemuse コマンド実行後に使用したメモリーを表示します。ネイティブモードでだけ有効です。

注 - showmemuse コマンドは Solaris プラットフォームでのみ利用可能です。

1 行に 1 つの「使用中ブロック」を示すレコードが出力されます。このコマンドは、ブロックの合計サイズに基づいてレポートをソートします。最後の showLeaks (355 ページの「showLeaks コマンド」参照) コマンド実行後にリークしたブロックもレポートに含まれます。

構文

```
showmemuse [-a] [-m <m>] [-n number] [-v]
```

ここで

-a は、使用中ブロックすべてを表示します (最後の showmemuse コマンド実行後のブロックだけではなく)。

-m *m* は、使用中ブロックレポートをまとめます。*m* のデフォルト値は 2 または check コマンドで最後に指定した大域値です (292 ページの「check コマンド」参照)。2 個以上のブロックに対する割り当て時の呼び出しスタックが *m* 個のフレームに一致するとき、これらのブロックは 1 つのレポートにまとめて報告されます。-m オプションを使用すると、*m* の大域値が無効となります。

-n *number* は、最大 *number* 個のレコードをレポートに表示します。デフォルト値は 20 です。-v は、冗長出力を生成します。デフォルトの場合、簡易出力が表示されます。

source コマンド

source コマンドは、指定ファイルからコマンドを実行します。ネイティブモードでだけ有効です。

構文

source *file_name* ファイル *file_name* からコマンドを実行します。\$PATH は検索されません。

status コマンド

status コマンドは、イベントハンドラ (ブレイクポイントなど) を一覧表示します。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

status 活動中の trace、when、および stop ブレイクポイントを出力します。

status *handler_id* ハンドラ *handler_id* のステータスを出力します。

status -h 隠れているものを含み、活動中の trace、when、および stop ブレイクポイントを出力します。

status -s 前述と同じですが、出力を dbx によって読み込むことができます。

ここで

handler_id は、イベントハンドラの識別子です。

例

```
(dbx) status -s > bpts
```

```
...
```

```
(dbx) source bpts
```

step コマンド

step コマンドは、1 ソース行または1 文をステップ実行します (-g オプションを使ってコンパイルされた呼び出しにステップインします)。

dbx 環境変数 `step_events` は、ステップ実行中にブレークポイントが使用可能であるかどうかを制御します。

dbx の環境変数 `step_granularity` は、ソース行のステップ実行のきめ細かさを制御します。

dbx の環境変数 `step_abflow` は、dbx が「異常」制御フロー変更が発生しそうになっていることを検出したときに停止するかどうかを制御します。このような制御フロー変更は、`siglongjmp()` または `longjmp()` の呼び出し、あるいは例外の送出しが原因で発生することがあります。

ネイティブモードの構文

<code>step</code>	1 行をステップ実行します (呼び出しにステップイン)。関数呼び出しがステップオーバーされるマルチスレッドプログラムの場合、デッドロック状態を避けるため、その関数呼び出し中は全スレッドが暗黙的に再開されます。非活動状態のスレッドをステップ実行することはできません。
<code>step n</code>	n 行をステップ実行します (呼び出しにステップイン)。
<code>step up</code>	ステップアップし、現在の関数から出ます。
<code>step ... -sig signal</code>	ステップ実行中に指定のシグナルを引き渡します。シグナルに対するシグナルハンドラが存在する場合、そのシグナルハンドラが -g オプション付きでコンパイルされていると、そのシグナルにステップインします。
<code>step ... thread_id</code>	指定のスレッドをステップ実行します。step up には適用されません。
<code>step ... lwp_id</code>	指定の LWP をステップ実行します。関数をステップオーバーしたときに全 LWP を暗黙に再開しません。
<code>step to [function]</code>	現在のソースコード行から呼び出された <i>function</i> へのステップインを試行します。function が指定されなかった場合は最後の関数へのステップインを試行し、step コマンドおよび step up コマンドによる長いシーケンスを防止できます。最後の関数の例としては、次のものがあります。

```
f()->s()-t()->last();
```

```
last(a() + b(c()->d()));
```

ここで

n は、ステップ実行対象の行数です。

signal はシグナル名です。

thread_id は、スレッド ID です。

lwp_id は、LWP ID です。

function は、関数名です。

明示的な *lwp_id* が指定されている場合のみ、step コマンドによる汎用のデッドロック回避策は無効となります。

step to コマンドを実行した際、最後のアセンブル呼び出し命令へのステップインや現在のソースコード行の関数 (指定されている場合) へのステップインが試行されている間、条件付き分岐があると呼び出しが受け付けられないことがあります。呼び出しが受け付けられない場合や現在のソースコード行に関数呼び出しがない場合、step to コマンドが現在のソースコード行をステップオーバーします。step to コマンドを使用する際は、ユーザー定義演算子に特に注意してください。

マシンレベルの呼び出しステップ実行については、[360 ページの「stepi コマンド」](#)も参照してください。

Java モードの構文

step	1 行をステップ実行します (呼び出しにステップイン)。メソッド呼び出しがステップオーバーされるマルチスレッドプログラムの場合、デッドロック状態を避けるため、そのメソッド呼び出し中は全スレッドが暗黙的に再開されます。非活動状態のスレッドをステップ実行することはできません。
step <i>n</i>	<i>n</i> 行をステップ実行します (呼び出しにステップイン)。
step up	ステップアップし、現在のメソッドから出ます。
step ... <i>thread_id</i>	指定のスレッドをステップ実行します。step up には適用されません。
step ... <i>lwp_id</i>	指定の LWP をステップ実行します。メソッドをステップオーバーしたときに全 LWP を暗黙的に再開しません。

stepi コマンド

stepi コマンドは、1 マシン命令をステップ実行します (呼び出しにステップイン)。ネイティブモードでだけ有効です。

構文

stepi	1つのマシン命令をシングルステップ実行します (呼び出しにステップイン)。
stepi <i>n</i>	<i>n</i> 個のマシン命令をシングルステップ実行します (呼び出しへのステップイン)。
stepi -sig <i>signal</i>	ステップ実行し、指定のシグナルを引き渡します。
stepi ... <i>lwp_id</i>	指定の LWP をステップ実行します。
stepi ... <i>thread_id</i>	指定のスレッドが活動状態である LWP をステップ実行します。

ここで

n は、ステップ実行対象の命令数です。

signal はシグナル名です。

lwp_id は、LWP ID です。

thread_id は、スレッド ID です。

stop コマンド

stop コマンドは、ソースレベルのブレークポイントを設定します。

構文

stop コマンドは、ソースレベルのブレークポイントを設定します。

stop *event_specification* [*modifier*]

指定イベントが発生すると、プロセスが停止されます。

ネイティブモードの構文

ネイティブモードで有効な構文の中で重要なものを、いくつか次に示します。これ以外のイベントについては、[266 ページの「イベント指定の設定」](#)を参照してください。

`stop [-update]`

実行をただちに停止します。when コマンドの本体内でのみ有効です。

`stop -nouupdate`

実行をただちに停止しますが、Sun Studio IDE のデバッガウィンドウは更新しません。

`stop access mode address_expression [, byte_size_expression]`

`address_expression` で指定したメモリーがアクセスされた場合に、実行を停止します。[100 ページの「特定アドレスへのアクセス時にプログラムを停止する」](#)も参照してください。

`stop at line-number`

実行を `line_number` で停止します。[96 ページの「ソースコードの特定の行に stop ブレークポイントを設定する」](#)も参照してください。

`stop change variable`

`variable` の値が変更された場合に実行を停止します。

`stop cond condition_expression`

`condition_expression` で指定した条件が真になる場合に実行を停止します。

`stop in function`

`function` が呼び出されたときに実行を停止します。[97 ページの「関数に stop ブレークポイントを設定する」](#)も参照してください。

`stop inclass class_name [-recurse | -norecurse]`

C++ のみ: `class/struct/union/template` のいずれかのクラスのメンバー関数すべてにブレークポイントを設定します。`-norecurse` はデフォルトです。`-recurse` が指定された場合、基底クラスが含まれます。[99 ページの「同じクラスのメンバー関数にブレークポイントを設定する」](#)も参照してください。

`stop infunction name`

C++ のみ: すべての非メンバー関数 `name` にブレークポイントを設定します。

`stop inmember name`

C++ のみ: すべての非メンバー関数 `name` にブレークポイントを設定します。[98 ページの「異なるクラスのメンバー関数にブレークポイントを設定する」](#)を参照してください。

`stop inobject object_expression [-recurse | -norecurse]`

C++ のみ: オブジェクト `object_expression` から呼び出された場合に、クラスおよびそのすべての基底クラスの非静的メソッドへのエントリにブレークポイントを設定します。`-recurse` はデフォルトです。`-norecurse` が指定された場合、基底クラ

スは含まれません。99 ページの「オブジェクトにブレークポイントを設定する」も参照してください。

line_number は、ソースコード行の番号です。

function は、関数の名前です。

class_name は、C++ の class、struct、union、または template クラスの名前です。

mode はメモリーのアクセス方法を指定します。次の文字 (複数可) で構成されます。

r 指定したアドレスのメモリーが読み取られたことを示します。

w メモリーへの書き込みが実行されたことを示します。

x メモリーが実行されたことを示します。

mode には、次を含めることもできます。

a アクセス後にプロセスを停止します (デフォルト)。

b アクセス前にプロセスを停止します。

name は、C++ 関数名です。

object_expression は、C++ オブジェクトを示します。

variable は、変数の名前です。

ネイティブモードでは、次の修飾子が有効です。

-if <i>condition_expression</i>	<i>condition_expression</i> が真の場合にだけ、指定したイベントが発生します。
-in <i>function</i>	指定したイベントが <i>function</i> の範囲内で発生した場合にだけ、実行が停止します。
-count <i>number</i>	カウンタが 0 で開始され、イベントの発生ごとに増分されます。 <i>number</i> に到達すると、実行が停止され、カウンタが 0 にリセットされます。
-count infinity	カウンタが 0 で開始され、イベントの発生ごとに増分されます。実行は停止されません。
-temp	イベントの発生時に削除される一時的なブレークポイントを作成します。
-disable	無効状態のブレークポイントを作成します。
-instr	命令レベルのバリエーションを実行します。たとえば、step は命令レベルのステップ実行になり、at では行番号ではなくテキストアドレスを引数として指定します。

-perm	このイベントをデバッグ中は常に有効にします。一部のイベント(ブレークポイントなど)は、常に有効にするには適していません。delete all は、常に有効なハンドラを削除しません。削除するには、delete hid を使用します。
-hidden	status コマンドからイベントを隠ぺいします。一部のインポートモジュールでこれが使用されることがあります。そのようなモジュールを表示するには、status -h を使用します。
-lwp lwp_id	指定したLWPで指定したイベントが発生した場合にだけ、実行が停止します。
-thread thread_id	指定したスレッドで指定したイベントが発生した場合にだけ、実行が停止します。

Java モードの構文

Java モードでは、次の構文が有効です。

stop access <i>mode class_name.field_name</i>	<i>class_name.field_name</i> で指定したメモリーがアクセスされた場合に、実行を停止します。
stop at <i>line_number</i>	<i>line_number</i> で実行を停止します。
stop at <i>file_name:line_number</i>	<i>file_name</i> の <i>line_number</i> で実行を停止します。
stop change <i>class_name.field_name</i>	<i>class_name</i> で <i>field_name</i> の値が変更された場合に実行を停止します。
stop classload	いずれかのクラスが読み込まれた場合に実行を停止します。
stop classload <i>class_name</i>	<i>class_name</i> が読み込まれた場合に実行を停止します。
stop classunload	いずれかのクラスが読み込み解除された場合に実行を停止します。
stop classunload <i>class_name</i>	<i>class_name</i> が読み込み解除された場合に実行を停止します。
stop cond <i>condition_expression</i>	<i>condition_expression</i> で指定した条件が真になる場合に実行を停止します。

`stop in class_name.method_name`

`class_name.method_name`に入ったあとで、`method_name`に入ったあとで、最初の行が実行される直前に実行を停止します。パラメータが指定されておらず、メソッドがオーバーロードされている場合は、メソッドのリストが表示されます。

`stop in class_name.method_name ([parameters])`

`class_name.method_name`に入ったあとで、`method_name`に入ったあとで、最初の行が実行される直前に実行を停止します。

`stop inmethod class_name.method_name`

`class_name.method_name`で指定した、すべての非メンバーメソッドでブレークポイントを設定します。

`stop inmethod class_name.method_name ([parameters])`

`class_name.method_name`で指定した、すべての非メンバーメソッドでブレークポイントを設定します。

`stop throw`

Java の例外が投げられた場合に実行を停止します。

`stop throw type`

`type`で指定した種類の Java の例外が投げられた場合に実行を停止します。

ここで

`class_name` は、Java クラス名で、パッケージのパス ("." (ピリオド) を修飾子として使用。たとえば `test1.extra.T1.Inner`) またはフルパス名 (# 記号で始まり、"/" (スラッシュ) や \$ 記号を修飾子として使用。たとえば `#test1/extra/T1$Inner`) のいずれかで指定します。修飾子 \$ を使用する場合は、`class_name` を引用符で囲みます。

`condition_expression` には、任意の式を指定できます。ただし、評価結果が整数型になる必要があります。

`field_name` は、クラス内のフィールド名です。

`file_name` は、ファイルの名前です。

`line_number` は、ソースコード行の番号です。

`method_name` は、Java メソッドの名前です。

`mode` はメモリーのアクセス方法を指定します。次の文字 (複数可) で構成されます。

r 指定したアドレスのメモリーが読み取られたことを示します。

w メモリーへの書き込みが実行されたことを示します。

`mode` には、次を含めることもできます。

b アクセス前にプロセスを停止します。

プログラムカウンタは、問題のある個所を示します。

parameters は、メソッドのパラメータです。

type は、Java の例外の種類です。 *type* には、 `-unhandled` または `-unexpected` を指定できます。

Java モードでは、次の修飾子が有効です。

- `-if condition_expression` *condition_expression* が真の場合にだけ、指定したイベントが発生します。
- `-count number` カウンタが0で開始され、イベントの発生ごとに増分されます。 *number* に到達すると、実行が停止され、カウンタが0にリセットされます。
- `-count infinity` カウンタが0で開始され、イベントの発生ごとに増分されます。実行は停止されません。
- `-temp` イベントの発生時に削除される一時的なブレークポイントを作成します。
- `-disable` 無効状態のブレークポイントを作成します。

マシンレベルのブレークポイントの設定については、[365 ページ](#)の「[stopi コマンド](#)」も参照してください。

全イベントのリストと構文については、[266 ページ](#)の「[イベント指定の設定](#)」を参照してください。

stopi コマンド

`stopi` コマンドは、マシンレベルのブレークポイントを設定します。ネイティブモードでだけ有効です。

構文

`stopi` コマンドの一般構文は、次のとおりです。

```
stopi event_specification [modifier]
```

指定イベントが発生すると、プロセスが停止されます。

次の構文が有効です。

```
stopi at address      address の場所で実行を停止します。
```

```
stopi in function      function が呼び出されたときに実行を停止します。
```

ここで

address は、アドレスとなった式またはアドレスとして使用可能な式です。

function は、関数の名前です。

全イベントのリストと構文については、[266 ページの「イベント指定の設定」](#)を参照してください。

suppress コマンド

suppress コマンドは、実行時検査中のメモリーエラーの報告を抑止します。ネイティブモードでだけ有効です。

注 - suppress コマンドが実行できるのは、Solaris プラットフォームのみです。

dbx 環境変数 `rtc_auto_suppress` が on である場合、指定場所におけるメモリーエラーは 1 度だけ報告されます。

構文

suppress

suppress コマンドと `unsuppress` コマンドの履歴 (`-d` オプションと `-reset` オプションを指定するものは含まない)。

suppress -d

デバッグ用にコンパイルされなかった関数で抑止されているエラーのリスト (デフォルト抑止)。このリストは、ロードオブジェクト単位です。これらのエラーの抑止を解除する方法は、`-d` オプションを付けて `unsuppress` を使用することだけです。

suppress -d errors

`errors` をさらに抑止することによって、全ロードオブジェクトに対するデフォルト抑止を変更します。

suppress -d errors in *load_objects*

`errors` をさらに抑止することによって、*load_objects* のデフォルト抑止を変更します。

suppress -last

エラー位置における現在のエラーを抑止します。

`suppress -reset`

デフォルト抑止としてオリジナルの値を設定します (起動時)。

`suppress -r id...`

識別子によって指定される抑止解除イベントを削除します (識別子は `unsuppress` コマンドで取得できます。380 ページの「`unsuppress` コマンド」参照)。

`suppress -r 0 | all | -all`

`unsuppress` コマンドによって指定される抑止解除イベントすべてを削除します (380 ページの「`unsuppress` コマンド」参照)。

`suppress errors`

あらゆる場所における `errors` を抑止します。

`suppress errors in [functions] [files] [load_objects]`

`functions` リスト、`files` リスト、`load_objects` リストにおける `errors` を抑止します。

`suppress errors at line`

`line` における `errors` を抑止します。

`suppress errors at "file":line`

`file` の `line` における `errors` を抑止します。

`suppress errors addr address`

`address` における `errors` を抑止します。

ここで

`address` は、メモリーアドレスです。

`errors` は空白文字で区切られた次の要素で構成されます。

`all` 全てのエラー

`aib` メモリーリークの可能性 - ブロック中のアドレス

`air` メモリーリークの可能性 - レジスタ中のアドレス

`baf` 不正解放

`duf` 重複解放

`mel` メモリーリーク

`maf` 境界整列を誤った解放

`mar` 境界整列を誤った読み取り

`maw` 境界整列を誤った書き込み

`oom` メモリー不足

`rua` 非割り当てメモリーからの読み取り

`ruj` 非初期化メモリーからの読み取り

wro 読み取り専用メモリーへの書き込み
wua 非割り当てメモリーへの書き込み
biu ブロック使用状況(割り当てられているメモリー)。**biu** はエラーではありませんが、*errors* とまったく同じように **suppress** コマンドで使用できます。

file は、ファイルの名前です。

files は、1 個または複数のファイル名です。

functions は、1 個または複数の関数名です。

line は、ソースコード行の番号です。

load_objects は、1 個または複数のロードオブジェクト名です。

エラーの抑止については、[144 ページの「エラーの抑止」](#)を参照してください。

エラーの抑止解除については、[380 ページの「**unsuppress** コマンド」](#)を参照してください。

sync コマンド

sync コマンドは、指定の同期オブジェクトに関する情報を表示します。ネイティブモードでだけ有効です。

注 - **sync** コマンドが実行できるのは、Solaris プラットフォームのみです。

構文

sync -info address address における同期オブジェクトに関する情報を表示します。

ここで

address は、同期オブジェクトのアドレスです。

syncs コマンド

syncs コマンドは、同期オブジェクト(ロック)すべてを一覧表示します。ネイティブモードでだけ有効です。

注 - syncs コマンドが実行できるのは、Solaris プラットフォームのみです。

構文

syncs

thread コマンド

thread コマンドは、現在のスレッドの表示や変更を行います。

ネイティブモードの構文

thread 現在のスレッドを表示します。

thread *thread_id* スレッド *thread_id* に切り替えます。

次の構文で *thread_id* がいない場合は、現在のスレッドが仮定されます。

thread -info [*thread_id*] 指定スレッドに関する既知情報すべてを出力します。

thread -hide [*thread_id*] 指定(または現在の)スレッドを隠ぺいします。通常のスレッドリストには表示されなくなります。

thread -unhide [*thread_id*] 指定(または現在の)スレッドを隠ぺい解除します。

thread -unhide all すべてのスレッドを隠ぺい解除します。

thread -suspend *thread_id* 指定した(または現在の)スレッドの実行を一時停止します。中断されているスレッドは、スレッドリストに "S" の文字とともに表示されます。

thread -resume *thread_id* -suspend の効果を解除します。

thread -blocks [*thread_id*] ほかのスレッドをブロックしている指定スレッドが保持しているロックすべてを一覧表示します。

`thread -blockedby [thread_id]` 指定スレッドをブロックしている同期オブジェクトがある場合、そのオブジェクトを表示します。

ここで

`thread_id` は、スレッド ID です。

Java モードの構文

`thread` 現在のスレッドを表示します。

`thread thread_id` スレッド `thread_id` に切り替えます。

次の構文で `thread_id` がいない場合は、現在のスレッドが仮定されます。

`thread -info [thread_id]` 指定スレッドに関する既知情報すべてを出力します。

`thread -hide [thread_id]` 指定 (または現在の) スレッドを隠ぺいします。通常のスレッドリストには表示されなくなります。

`thread -unhide [thread_id]` 指定 (または現在の) スレッドを隠ぺい解除します。

`thread -unhide all` すべてのスレッドを隠ぺい解除します。

`thread -suspend thread_id` 指定した (または現在の) スレッドの実行を一時停止します。中断されているスレッドは、スレッドリストに "S" の文字とともに表示されます。

`thread -resume thread_id` `-suspend` の効果を解除します。

`thread -blocks [thread_id]` `thread_id` が所有する Java モニターを表示します。

`thread -blockedby [thread_id]` `thread_id` がブロックされている Java モニターを表示します。

ここで

`thread_id` は `t@number` の dbx 形式のスレッド ID またはスレッドを指定した Java スレッド名です。

threads コマンド

threads コマンドは、すべてのスレッドを一覧表示します。

ネイティブモードの構文

threads	既知のスレッドすべてのリストを出力します。
threads -all	通常出力されないスレッド (ゾンビ) を出力します。
threads -mode all filter	全スレッドを出力するか、またはスレッドをフィルタリングするかを指定します。デフォルトではスレッドがフィルタリングされます。フィルタリングがオンになっている場合、thread -hide コマンドによって隠されているスレッドはリスト表示されません。
threads -mode auto manual	dbx デバッガで、スレッドリストの自動更新を有効にします。
threads -mode	現在のモードをエコーします。

各行は、次の項目で構成されます。

- * (アスタリスク) は、ユーザーの注意を必要とするイベントがこのスレッドで発生したことを示します。通常は、ブレークポイントに付けられます。
アスタリスクの代わりに 'o' が示される場合は、dbx 内部イベントが発生していません。
- > (矢印) は、現在のスレッドを示します。
- t@num はスレッド ID であり、特定のスレッドを指します。number は、thr_create が返す thread_t の値になります。
- b l@num は、そのスレッドが結合されていること (指定した LWP に現在割り当てられている) を示します。a l@num は、スレッドがアクティブであること (現在実行が予定されている) を表します。
- thr_create に渡されたスレッドの開始関数。?() は開始関数が不明であることを示します。
- スレッドの状態。次のいずれかになります。
 - monitor
 - 実行中
 - スリープ
 - wait
 - 未知
 - ゾンビ

スレッドが現在実行している関数

Java モードの構文

<code>threads</code>	既知のスレッドすべてのリストを出力します。
<code>threads -all</code>	通常出力されないスレッド (ゾンビ) を出力します。
<code>threads -mode all filter</code>	全スレッドを出力するか、またはスレッドをフィルタリングするかを指定します。デフォルトではスレッドがフィルタリングされます。
<code>threads -mode auto manual</code>	dbx デバッガで、スレッドリストの自動更新を有効にします。
<code>threads -mode</code>	現在のモードをエコーします。

各行は、次の項目で構成されます。

- `>` (矢印) は、現在のスレッドを示します。
- `t@number` は dbx スタイルスレッド ID を示します。
- スレッドの状態。次のいずれかになります。
 - `monitor`
 - 実行中
 - スリープ
 - `wait`
 - 未知
 - ゾンビ

単一引用符内のスレッド名

- スレッドの優先順位を示す番号

trace コマンド

`trace` コマンドは、実行したソース行、関数呼び出し、変数の変更を表示します。

トレース速度は dbx 環境変数 `trace_speed` によって設定します。

dbx が Java モードで、トレースのブレークポイントをネイティブコードで設定する場合は、`joff` コマンドを使用してネイティブモードに切り替えるか (326 ページの「`joff` コマンド」参照)、`trace` コマンドの前に `native` を追加します (339 ページの「`native` コマンド」参照)。

dbx が JNI モードで、トレースのブレイクポイントを Java コードで設定する場合は、trace コマンドの前に java を追加します (326 ページの「java コマンド」参照)。

構文

trace コマンドの一般構文は、次のとおりです。

```
trace event_specification [modifier]
```

指定イベントが発生すると、トレースが出力されます。

ネイティブモードの構文

ネイティブモードでは、次の構文が有効です。

<code>trace -file <i>file_name</i></code>	指定 <i>file_name</i> に全トレース出力を送ります。トレース出力を標準出力に戻すには、 <i>file_name</i> の代わりに - を使用します。トレース出力は常に <i>file_name</i> に追加されます。トレース出力は、dbx がプロンプト表示するたび、またアプリケーションが終了するたびにフラッシュされます。dbx 接続後にプログラムの実行を再開するか新たに実行を開始すると、 <i>file_name</i> が常に開きます。
<code>trace step</code>	各ソース行、関数呼び出し、および戻り値をトレースします。
<code>trace next -in <i>function</i></code>	指定 <i>function</i> の中で各ソース行をトレースします。
<code>trace at <i>line_number</i></code>	指定のソース <i>line_number</i> をトレースします。
<code>trace in <i>function</i></code>	指定 <i>function</i> の呼び出しとこの関数からの戻り値をトレースします。
<code>trace inmember <i>function</i></code>	<i>function</i> という名前のメンバー関数の呼び出しをトレースします。
<code>trace infunction <i>function</i></code>	<i>function</i> という名前の関数が呼び出されるとトレースします。
<code>trace inclass <i>class</i></code>	<i>class</i> のメンバー関数の呼び出しをトレースします。
<code>trace change <i>variable</i></code>	<i>variable</i> の変更をトレースします。

ここで

file_name は、トレース出力の送信先ファイルの名前です。

function は、関数の名前です。

line_number は、ソースコード行の番号です。

class は、クラスの名前です。

variable は、変数の名前です。

ネイティブモードでは、次の修飾子が有効です。

- if *condition_expression* *condition_expression* が真の場合にだけ、指定したイベントが発生します。
- in *function* 指定したイベントが関数で発生した場合にだけ、実行が停止します。
- count *number* カウンタが0で開始され、イベントの発生ごとに増分されます。*number* に到達すると、実行が停止され、カウンタが0にリセットされます。
- count infinity カウンタが0で開始され、イベントの発生ごとに増分されます。実行は停止されません。
- temp イベントの発生時に削除される一時的なブレークポイントを作成します。
- disable 無効状態のブレークポイントを作成します。
- instr 命令レベルのバリエーションを実行します。たとえば、*step* は命令レベルのステップ実行になり、*at* では行番号ではなくテキストアドレスを引数として指定します。
- perm このイベントをデバッグ中は常に有効にします。一部のイベント(ブレークポイントなど)は、常に有効にするには適していません。*delete all* は、常に有効なハンドラを削除しません。削除するには、*delete hid* を使用します。
- hidden *status* コマンドからイベントを隠ぺいします。一部のインポートモジュールでこれが使用されることがあります。そのようなモジュールを表示するには、*status -h* を使用します。
- lwp *lwpid* 指定したLWPで指定したイベントが発生した場合にだけ、実行が停止します。
- thread *thread_id* 指定したスレッドで指定したイベントが発生した場合にだけ、実行が停止します。

Java モードの構文

Java モードでは、次の構文が有効です。

`trace -file file_name`

指定 `file_name` に全トレース出力を送ります。トレース出力を標準出力に戻すには、`file_name` の代わりに `-` を使用します。トレース出力は常に `file_name` に追加されます。トレース出力は、`dbx` がプロンプト表示するたび、またアプリケーションが終了するたびにフラッシュされます。`file_name` は、接続後の新規実行時や再開時に必ずオープンし直されます。

`trace at line_number`

`line_number` をトレースします。

`trace at file_name.line_number`

指定したソース `file_name.line_number` をトレースします。

`trace in class_name.method_name`

`class_name.method_name` の呼び出しと、このメソッドからの戻り値をトレースします。

`trace in class_name.method_name([parameters])`

`class_name.method_name([parameters])` の呼び出しと、このメソッドからの戻り値をトレースします。

`trace inmethod class_name.method_name`

`class_name.method_name` という名前のメソッドの呼び出しと、このメソッドからの戻り値をトレースします。

`trace inmethod class_name.method_name[(parameters)]`

`class_name.method_name [(parameters)]` という名前のメソッドの呼び出しと、このメソッドからの戻り値をトレースします。

ここで

`class_name` は、Java クラス名で、パッケージのパス ("`.`" (ピリオド) を修飾子として使用。たとえば `test1.extra.T1.Inner`) またはフルパス名 (`#` 記号で始まり、"`/`" (スラッシュ) や `$` 記号を修飾子として使用。たとえば `#test1/extra/T1$Inner`) のいずれかで指定します。修飾子 `$` を使用する場合は、`class_name` を引用符で囲みます。

`file_name` は、ファイルの名前です。

`line_number` は、ソースコード行の番号です。

`method_name` は、Java メソッドの名前です。

`parameters` は、メソッドのパラメータです。

Java モードでは、次の修飾子が有効です。

<code>-if condition_expression</code>	<code>condition_expression</code> が真の場合にだけ、指定したイベントが発生し、トレースが出力されます。
<code>-count number</code>	カウンタが0で開始され、イベントの発生ごとに増分されます。 <code>number</code> に到達すると、トレースが出力され、カウンタが0にリセットされます。
<code>-count infinity</code>	カウンタが0で開始され、イベントの発生ごとに増分されます。実行は停止されません。
<code>-temp</code>	イベントが発生してトレースが出力されるときに削除される、一時的なブレークポイントを作成します。 <code>-temp</code> を <code>-count</code> とともに使用した場合は、カウンタが0にリセットされたときだけブレークポイントが削除されます。
<code>-disable</code>	無効状態のブレークポイントを作成します。

全イベントのリストと構文については、[266 ページ](#)の「[イベント指定の設定](#)」を参照してください。

tracei コマンド

tracei コマンドは、マシン命令、関数呼び出し、変数の変更を表示します。ネイティブモードでだけ有効です。

tracei は、`trace event-specification -instr` の省略形です。ここで、`-instr` 修飾子を指定すると、ソース行の細分性ではなく命令の細分性でトレースが行われます。イベント発生時に出力される情報は、ソース行の書式ではなく逆アセンブリの書式になります。

構文

<code>tracei step</code>	各マシン命令をトレースします。
<code>tracei next -in function</code>	指定 <code>function</code> の中で書く命令をトレースします。
<code>tracei at address_expression</code>	<code>address</code> にある命令をトレースします。
<code>tracei in function</code>	指定 <code>function</code> の呼び出しとこの関数からの戻り値をトレースします。
<code>tracei inmember function</code>	<code>function</code> という名前のメンバー関数の呼び出しをトレースします。

`tracei infunction function` *function* という名前の関数が呼び出されるとトレースします。

`tracei inclass class` *class* のメンバー関数の呼び出しをトレースします。

`tracei change variable` *variable* の変更をトレースします。

ここで

filename は、トレース出力の送信先ファイルの名前です。

function は、関数の名前です。

line は、ソースコード行の番号です。

class は、クラスの名前です。

variable は、変数の名前です。

詳細については、[372 ページの「trace コマンド」](#)を参照してください。

unchecked コマンド

unchecked コマンドは、メモリのアクセス、リーク、使用状況の検査を使用不可にします。ネイティブモードでだけ有効です。

注 - unchecked コマンドが実行できるのは、Solaris プラットフォームのみです。

構文

<code>unchecked</code>	検査の現在のステータスを出力します。
<code>unchecked -access</code>	アクセス検査を停止します。
<code>unchecked -leaks</code>	リーク検査を停止します。
<code>unchecked -memuse</code>	memuse 検査を停止します (リーク検査も停止されます)。
<code>unchecked -all</code>	unchecked -access、unchecked -memuse と同じです。
<code>unchecked [<i>functions</i>] [<i>files</i>] [<i>loadobjects</i>]</code>	<i>functions files loadobjects</i> に対する suppress all と同じです。

ここで

functions は、1 個または複数の関数名です。

files は、1 個または複数のファイル名です。

loadobjects は、1 個または複数のロードオブジェクト名です。

検査をオンにする方法については、[292 ページ](#)の「[check コマンド](#)」を参照してください。

エラーの抑止解除については、[366 ページ](#)の「[suppress コマンド](#)」を参照してください。

実行時検査の概要については、[130 ページ](#)の「[概要](#)」を参照してください。

undisplay コマンド

undisplay コマンドは、display コマンドを取り消します。

ネイティブモードの構文

undisplay *expression*, ... display *expression* コマンドを取り消します。

undisplay *n*, ... *n* 個の display コマンドを取り消します。

undisplay 0 すべての display コマンドを取り消します。

ここで

expression は、有効な式です。

Java モードの構文

undisplay *expression*, ... | *identifier*, ...
display *expression*, ... または display *identifier*, ... コマンドを取り消します。

undisplay *n*, ...
n 個の display コマンドを取り消します。

undisplay 0
すべての display コマンドを実行します。
すべての display コマンドを取り消します。

ここで

expression は、有効な Java の式です。

field_name は、クラス内のフィールド名です。

identifier は、局所変数またはパラメータです。これには、`this`、現在のクラスのインスタンス変数 (*object_name.field_name*)、クラス (static) 変数 (*class_name.field_name*) が含まれます。

unhide コマンド

unhide コマンドは、hide コマンドを取り消します。ネイティブモードでだけ有効です。

構文

unhide 0 全てのスタックフレームフィルタを削除します。

unhide *regular_expression* スタックフレームフィルタ *regular_expression* を削除します。

unhide *number* スタックフレームフィルタ番号 *number* を削除します。

ここで

regular_expression は、正規表現です。

number は、スタックフレームフィルタの番号です。

hide コマンド (285 ページの「[浮動小数点例外を捕捉する](#)」参照) は、番号を持つフィルタを一覧表示します。

unintercept コマンド

unintercept コマンドは、intercept コマンドを取り消します (C++ のみ)。ネイティブモードでだけ有効です。

構文

unintercept *intercepted_typename* [, *intercepted_typename* ...]
種類が *intercepted_typename* の送出手を intercept リストから削除します。

unintercept -a[ll]
すべての種類の送出手を intercept リストから削除します。

`unintercept -x excluded_typename [, excluded_typename ...]`
excluded_typename を `excluded` リストから削除します。

`unintercept -x -a[ll]`
 すべての種類の送出を `excluded` リストから削除します。

`unintercept`
 阻止対象の型を一覧表示します。

ここで

included_typename および *excluded_typename* は、`List <int>` や `unsigned short` などの例外仕様です。

unsuppress コマンド

`unsuppress` コマンドは、`suppress` コマンドを取り消します。ネイティブモードでだけ有効です。

注 - `unsuppress` コマンドが実行できるのは、Solaris プラットフォームのみです。

構文

`unsuppress`
`suppress` コマンドと `unsuppress` コマンドの履歴 (`-d` オプションと `-reset` オプションを指定するものは含まない)。

`unsuppress -d`
 デバッグ用にコンパイルされなかった関数で抑止解除されているエラーのリスト。このリストは、ロードオブジェクト単位です。エラーを抑止する方法は、`-d` オプションを付けて `suppress` コマンド (366 ページの「`suppress` コマンド」参照) を使用することだけです。

`unsuppress -d errors`
`errors` をさらに抑止解除することによって、全ロードオブジェクトに対するデフォルト抑止を変更します。

`unsuppress -d errors in loadobjects`
`errors` をさらに抑止解除することによって、`loadobjects` のデフォルト抑止を変更します。

`unsuppress -last`
 エラー位置における現在のエラーを抑止解除します。

`unsuppress -reset`
 デフォルト抑止マスクとしてオリジナルの値を設定します (起動時)。

`unsuppress errors`

あらゆる場所における *errors* を抑止解除します。

`unsuppress errors in [functions] [files] [loadobjects]`

functions リスト、*files* リスト、*loadobjects* リストにおける *errors* を抑止します。

`unsuppress errors at line`

line における *errors* を抑止解除します。

`unsuppress errors at "file"line`

file の *line* における *errors* を抑止解除します。

`unsuppress errors addr address`

address における *errors* を抑止解除します。

unwatch コマンド

`unwatch` コマンドは、`watch` コマンドを取り消します。ネイティブモードでだけ有効です。

構文

`unwatch expression, ...` `watch expression` コマンドを取り消します。

`unwatch n, ...` *n* で指定された番号の `watch` コマンドを取り消します。

`unwatch 0` すべての `watch` コマンドを取り消します。

ここで

expression は、有効な式です。

up コマンド

`up` コマンドは、呼び出しスタックを上方向に移動します (`main` に近づく)。このコマンドの構文および機能は、ネイティブモードと Java モードで同一です。

構文

`up` 呼び出しスタックを 1 レベル上方向に移動します。

`up number` 呼び出しスタックを *number* レベルだけ上方向に移動します。

`up -h [number]` 呼び出しスタックを上方向に移動しますが、隠しフレームをとばすことはしません。

ここで

`number` は、呼び出しスタックレベルの数です。

use コマンド

`use` コマンドは、ディレクトリ検索パスの表示や変更を行います。ネイティブモードでだけ有効です。

このコマンドは古いため、次の `pathmap` コマンドにマッピングしてあります。

`use` は、`pathmap -s` と同じです。

`use directory` は、`pathmap directory` と同じです。

watch コマンド

`watch` コマンドは、すべての停止ポイントの式を、その停止ポイントでの現在のスコープ内で評価して出力します。式は入力時に分析されないため、式の正確さをすぐに確認できません。`watch` コマンドはネイティブモードだけで有効です。

構文

`watch`

表示されている式のリストを表示します。

`watch expression, ...`

式 `expression, ...` の値を表示します。

`watch [-r|+r|-d|+d|-p|+p|-L|-fformat|-F format|--] expression, ...`

フラグの意味については、[344 ページ](#)の「`print` コマンド」を参照してください。

ここで

`expression` は、有効な式です。

`format` は、式の出力時に使用する形式です。詳細については、[344 ページ](#)の「`print` コマンド」を参照してください。

whatis コマンド

ネイティブモードでは、`whatis` コマンドは式の型または型の宣言を出力します。Java モードでは、`whatis` コマンドは識別子の宣言を出力します。識別子がクラスの場合は、クラスのメソッド (継承されたすべてのメソッドを含む) を出力します。

ネイティブモードの構文

`what is [-n] [-r] name` 型ではない *name* の宣言を出力します。

`whatis -t [-r] type` 型 *type* の宣言を出力します。

`whatis -e [-r] [-d] expression` 式 *expression* の型を出力します。

ここで

name は、型ではない名前です。

type は、型名です。

expression は、有効な式です。

`-d` は、静的型ではなく動的型を表示します (C++ のみ)。

`-e` は、式の型を表示します。

`-n` は、型ではない宣言を表示します。`-n` はオプションを付けずに `whatis` コマンドを使用したときのデフォルト値であるため、`-n` を指定する必要はありません。

`-r` は、基底クラスに関する情報を出力します (C++ のみ)。

`-t` は、型の宣言を表示します。

C++ のクラスや構造体に対して `whatis` コマンドを実行すると、定義済みメンバー関数すべて (未定義メンバー関数は除く)、静的データメンバー、クラスのフレンド、およびそのクラス内で明示的に定義されているデータメンバーのリストが表示されます。

`-r (recursive)` オプションを指定すると、継承クラスからの情報が追加されます。

`-d` フラグを `-e` フラグと併用すると、式の動的型が使用されます。

C++ の場合、テンプレート関係の識別子は次のように表示されます。

- テンプレート定義は `whatis -t` によって一覧表示されます。
- 関数テンプレートのインスタンス化は、`whatis` によって一覧表示されます。
- クラステンプレートのインスタンス化は、`whatis -t` によって一覧表示されません。

Java モードの構文

`whatis identifier` `identifier` の宣言を出力します。

ここで

`identifier` は、クラス、現在のクラス内のメソッド、現在のフレーム内の局所変数、現在のクラス内のフィールドのいずれかです。

when コマンド

when コマンドは、指定したイベントが発生したときに、コマンドを実行します。

dbx が Java モードで、when のブレイクポイントをネイティブコードで設定する場合は、`joff` コマンドを使用してネイティブモードに切り替えるか (326 ページの「`joff` コマンド」を参照)、when コマンドの前に `native` を追加します (339 ページの「`native` コマンド」参照)。

dbx が JNI モードで、when のブレイクポイントを Java コードで設定する場合は、when コマンドの前に `java` を追加します (326 ページの「`java` コマンド」を参照)。

構文

when コマンドの一般構文は、次のとおりです。

```
when event_specification [modifier]{command; ... }
```

指定イベントが発生すると、コマンドが実行されます。when コマンドで禁止されているコマンドには次のものがあります。

- `attach`
- `debug`
- `next`
- `replay`
- `rerun`
- `restore`
- `run`
- `save`
- `step`

オプションなしの `cont` コマンドは無視されます。

ネイティブモードの構文

ネイティブモードでは、次の構文が有効です。

`when at line_number { command; }` `line` に到達したら、`command(s)` を実行します。
`when in procedure { command; }` `procedure` が呼び出されたら、`command(s)` を実行します。

ここで

`line_number` は、ソースコード行の番号です。

`command` は、コマンドの名前です。

`procedure` は、手続きの名前です。

Java モードの構文

Java モードでは、次の構文が有効です。

`when at line_number`
 ソースの `line_number` に到達したときにコマンドを実行します。

`when at file_name.line_number`
`file_name.line_number` に到達したときにコマンドを実行します。

`when in class_name.method_name`
`class_name.method_name` が呼び出されたときにコマンドを実行します。

`when in class_name.method_name([parameters])`
`class_name.method_name([parameters])` が呼び出されたときにコマンドを実行します。

`class_name` は、Java クラス名で、パッケージのパス ("`.`" (ピリオド) を修飾子として使用。たとえば `test1.extra.T1.Inner`) またはフルパス名 (`#` 記号で始まり、"`/`" (スラッシュ) や `$` 記号を修飾子として使用。たとえば `#test1/extra/T1$Inner`) のいずれかで指定します。修飾子 `$` を使用する場合は、`class_name` を引用符で囲みます。

`file_name` は、ファイルの名前です。

`line_number` は、ソースコード行の番号です。

`method_name` は、Java メソッドの名前です。

`parameters` は、メソッドのパラメータです。

全イベントのリストと構文については、[266 ページの「イベント指定の設定」](#) を参照してください。

下位レベルのイベントの発生時にコマンドを実行する方法については、[386 ページ](#)の「[wheni コマンド](#)」を参照してください。

wheni コマンド

wheni コマンドは、指定した下位レベルのイベントが発生したときに、コマンドを実行します。ネイティブモードでだけ有効です。

wheni コマンドの一般構文は、次のとおりです。

構文

```
wheni wheni event_specification [modifier]{command... ; }
```

指定イベントが発生すると、コマンドが実行されます。

次の構文が有効です。

```
wheni at address_expression { command; }    address_expression に到達したら、  
command(s) を実行します。
```

ここで

address は、アドレスとなった式またはアドレスとして使用可能な式です。

command は、コマンドの名前です。

全イベントのリストと構文については、[266 ページ](#)の「[イベント指定の設定](#)」を参照してください。

where コマンド

where コマンドは、呼び出しスタックを出力します。

ネイティブモードの構文

where 手続きトレースバックを出力します。

where *number* トレースバックの上から *number* 個のフレームを出力します。

where -f *number* フレーム *number* からトレースバックを開始します。

<code>where -fp <i>address_expression</i></code>	fp レジスタに <i>address_expression</i> 値がある場合、トレースバックを出力します。
<code>where -h</code>	隠しフレームを含めます。
<code>where -l</code>	関数名を持つライブラリ名を含めます。
<code>where -q</code>	クイックトレースバック (関数名のみ)。
<code>where -v</code>	冗長トレースバック (関数の引数と行情報を含む)。

ここで

number は、呼び出しスタックフレームの数です。

これらの構文はスレッドや LWP ID を結合させることにより、指定エンティティのトレースバックを取り出すことがあります。

-fp オプションは、fp (frame pointer) レジスタが壊れていてイベント dbx が呼び出しスタックを正しく再構築できないときに役立ちます。このオプションは、値が正しい fp レジスタ値かをテストするためのショートカットを提供します。指定した正しい値が特定されたことが確認できた場合、assign コマンドや lwp コマンドを使用して設定できます。

Java モードの構文

<code>where [<i>thread_id</i>]</code>	メソッドのトレースバックを出力します。
<code>where [<i>thread_id</i>] <i>number</i></code>	トレースバックの上から <i>number</i> 個のフレームを出力します。
<code>where -f [<i>thread_id</i>] <i>number</i></code>	フレーム <i>number</i> からトレースバックを開始します。
<code>where -q [<i>thread_id</i>]</code>	クイックトレースバック (関数名のみ)。
<code>where -v [<i>thread_id</i>]</code>	冗長トレースバック (関数の引数と行情報を含む)。

ここで

number は、呼び出しスタックフレームの数です。

thread_id は、dbx 形式のスレッド ID またはスレッドを指定した Java スレッド名です。

whereami コマンド

whereami コマンドは、現在のソース行を表示します。ネイティブモードでだけ有効です。

構文

`whereami` 現在の位置 (スタックのトップ) に該当するソース行、および現在のフレームに該当するソース行を表示します (前者と異なる場合)。

`whereami -instr` 前述と同じ。ただし、ソース行ではなく現在の逆アセンブル命令が出力されます。

whereis コマンド

whereis コマンドは、特定の名前の使用状況すべて、またはアドレスの英字名を出力します。ネイティブモードでだけ有効です。

構文

`whereis name` *name* の宣言をすべて出力します。

`whereis -a address_expression` *address_expression* の場所を出力します。

ここで

name は、変数、関数、クラステンプレート、関数テンプレートといった、スコープ内の読み込み可能オブジェクトの名前です。

address は、アドレスとなった式またはアドレスとして使用可能な式です。

which コマンド

which コマンドは、指定の名前の完全修飾形を出力します。ネイティブモードでだけ有効です。

構文

`which [-n] name` *name* の完全修飾形を出力します。

`which -t type` *type* の完全修飾形を出力します。

ここで

name は、変数、関数、クラステンプレート、関数テンプレートといった、スコープ内の物の名前です。

type は、型名です。

`-n` は、型以外の完全修飾形を表示します。`-n` はオプションを指定せずに `which` コマンドを使用したときにデフォルトで設定されるため、`-n` を指定する必要はありません。

`-t` は、型の完全修飾形を表示します。

whocatches コマンド

`whocatches` コマンドは、C++ 例外が捕獲される場所を示します。ネイティブモードでだけ有効です。

構文

`whocatches type` 型 *type* の例外が現在の実行点で送出された場合にどこで捕獲されることになるかを示します (捕獲されるとしたら)。次に実行される分が `throw x` であり (*x* の型は *type*)、これを捕獲する `catch` 節の行番号、関数名、フレーム番号を表示するとします。

このとき、送出を行う関数の中に捕獲点がある場合には、"*type* is unhandled" が返されます。

ここで

type は、例外の型です。

索引

数字・記号

:: (コロンを重ねた) C++ 演算子, 75

A

access イベント, 268

alias コマンド, 51

AMD64 レジスタ, 247

array_bounds_check 環境変数, 63

assign コマンド

構文, 287-288

使用して値を変数に割り当て, 260

使用して大域変数に正しい値を再び割り当て, 167

使用して大域変数を復元, 168

使用して変数に値を代入, 123

attach イベント, 273

attach コマンド, 73, 89, 288-289

at イベント, 267

B

bcheck コマンド, 153

構文, 153

例, 153-154

bind コマンド, 252

bsearch コマンド, 289

C

C++

dbx の使用, 189-190

-g0 オプションでコンパイル, 51

-g オプションでコンパイル, 51

あいまいまたは多重定義された関数, 70

オブジェクトポインタ型, 121-122

関数テンプレートインスタンス化、リスト, 80

逆引用符演算子, 75

クラス

継承されたすべてのデータメンバーを表示, 121

継承されたメンバーを表示, 82

宣言、検索, 80-83

宣言の出力, 81

直接定義されたすべてのデータメンバーを表示, 121

定義、調べる, 81-83

表示, 80-83

継承されたメンバー, 82

コロンを重ねたスコープ決定演算子, 75

さまざまな名前, 77

出力, 121-122

テンプレート定義

修正, 168

表示, 80

テンプレートのデバッグ, 194

複数のブレークポイントの設定, 98-100

無名引数, 121-122

メンバー関数のトレース, 105

例外処理, 190-193

C++ ソースファイル、場所を指定する, 222

- call コマンド
 - インスタンス化された関数やクラステンプレートのメンバー関数を明示的に呼び出し, 198
 - 構文, 289-291
 - 使用して関数を呼び出す, 92
 - 使用して手続きを呼び出す, 92, 261
 - 使用して明示的に関数を呼び出す, 92
 - cancel コマンド, 291
 - catch コマンド, 185, 186, 291-292
 - change イベント, 268-269
 - check コマンド, 40, 131, 292-294
 - CLASSPATH 環境変数, 63, 218
 - clear コマンド, 294-295
 - collector archive コマンド, 297
 - collector dbxsample コマンド, 297
 - collector disable コマンド, 297-298
 - collector enable コマンド, 298
 - collector heaptrace コマンド, 298
 - collector hw_profile コマンド, 298-299
 - collector limit コマンド, 299
 - collector mpitrace コマンド, 299
 - collector pause コマンド, 299-300
 - collector profile コマンド, 300
 - collector resume コマンド, 300
 - collector sample コマンド, 300-301
 - collector show コマンド, 301
 - collector status コマンド, 301
 - collector store コマンド, 301-302
 - collector synctrace コマンド, 302
 - collector tha コマンド, 302
 - collector version コマンド, 303
 - collector コマンド, 295-303
 - cond イベント, 269
 - cont コマンド
 - 構文, 303
 - 使用して修正後プログラムの実行を継続, 166
 - 使用して大域変数の復元後に実行を継続, 168
 - 使用して別の行からプログラムの実行を継続, 91, 167, 262
 - 使用してマルチスレッドプログラムの実行を再開, 172
 - デバッグの情報なしでコンパイルできるファイルの制限, 164
 - プログラムの実行の継続, 91, 132
 - core_lo_pathmap 環境変数, 63
 - count イベント指定修飾子, 277
 - C ソースファイル, 場所を指定する, 222
- ## D
- dalias コマンド, 303-304
 - dbxenv コマンド, 50-51, 62, 307
 - .dbxrc ファイル, 61
 - dbx 起動時シーケンスで使用, 49
 - dbx 起動時に使用, 61
 - 作成, 62
 - 例, 62
 - dbxrc ファイル, dbx 起動時シーケンスでの使用, 49
 - dbxrc ファイル, dbx 起動時に使用, 61
 - .dbxrc ファイルの例, 62
 - dbx 環境変数, 63
 - array_bounds_check, 63
 - CLASSPATH, 63, 218
 - core_lo_pathmap, 63
 - dbxenv コマンドで設定, 62-68
 - debug_file_directory, 63
 - disassembler_version, 63
 - event_safety, 63
 - fix_verbose, 63
 - follow_fork_inherit, 63, 178
 - follow_fork_mode, 63, 146, 178
 - follow_fork_mode_inner, 64
 - input_case_sensitive, 64, 202
 - JAVASRCPATH, 64, 218
 - Java デバッグ用, 218
 - jdbx_mode, 64, 218
 - jvm_invocation, 64, 218
 - language_mode, 64
 - mt_scalable, 64
 - optim_local_vars, 64
 - output_auto_flush, 64
 - output_base, 65
 - output_class_prefix, 65
 - output_derived_type, 121
 - output_dynamic_type, 65, 190
 - output_inherited_members, 65
 - output_list_size, 65

dbx 環境変数 (続き)

output_log_file_name, 65
 output_max_string_length, 65
 output_no_literal, 65
 output_pretty_print, 65
 output_pretty_print_fallback, 65
 output_short_file_name, 65
 overload_function, 65
 overload_operator, 65
 pop_auto_destruct, 65
 proc_exclusive_attach, 66
 rtc_auto_continue, 66, 132, 154
 rtc_auto_suppress, 66, 144
 rtc_biu_at_exit, 66, 142
 rtc_error_limit, 66, 145
 rtc_error_log_file_name, 66, 132, 154
 rtc_error_stack, 66
 rtc_inherit, 66
 rtc_mel_at_exit, 66
 run_autostart, 66
 run_io, 67
 run_pty, 67
 run_quick, 67
 run_savetty, 67
 run_setpgrp, 67
 scope_global_enums, 67
 scope_look_aside, 67, 79
 session_log_file_name, 67
 stack_find_source, 67, 73
 stack_max_size, 68
 stack_verbose, 68
 step_abflow, 68
 step_events, 68, 109
 step_granularity, 68, 91
 suppress_startup_message, 68
 symbol_info_compression, 68
 trace_speed, 68, 106
 および Korn シェル, 68

dbx、起動, 43

起動オプション, 306
 コアファイル名を使用, 44-48
 プロセス ID のみ, 49

dbx コマンド, 43, 48, 304-307

dbx コマンド

Java コードのデバッグ時に利用される静的および動的情報, 229
 Java の式の評価, 228
 Java モードで構文が異なる, 230
 Java モードでだけ有効, 232
 Java モードでの使用, 228-232
 構文と機能が Java モードとネイティブモードで完全に同じコマンド, 229
 dbx セッションを終了する, 55-56
 dbx の Java コードデバッグモード, 226
 dbx のカスタマイズ, 61
 dbx の起動, 32
 dbx を終了する, 40
 debug_file_directory 環境変数, 63
 debug コマンド, 73
 構文, 307-310
 子プロセスに接続, 177
 使用して dbx を実行中のプロセスへ接続, 89
 使用してコアファイルをデバッグ, 45
 delete コマンド, 310-311
 detach イベント, 274
 detach コマンド, 55, 90, 311
 -disable イベント指定修飾子, 277
 disassembler_version 環境変数, 63
 display コマンド, 122, 312-313
 dis コマンド, 72, 236-237, 311-312
 dlopen()
 ブレークポイントの制限, 107
 ブレークポイントの設定, 107
 dlopen イベント, 269
 down コマンド, 74, 113, 314
 dump コマンド, 314
 dump コマンド, OpenMP コードの使用, 182

E

edit コマンド, 314-315
 event_safety 環境変数, 63
 examine コマンド, 72, 234-236, 315-316
 exception コマンド, 190, 316-317
 exec 関数、追跡, 178
 exists コマンド, 317
 exit イベント, 272

F

fault イベント, 270
 fflush(stdout)、dbx の呼出し後, 93
 files コマンド, 318
 file コマンド, 70, 72, 74, 317
 fix_verbose 環境変数, 63
 fixed コマンド, 319
 fix コマンド, 164, 165, 262, 318-319
 効果, 165
 デバッグの情報なしでコンパイルできるファイルの制限, 164
 follow_fork_inherit 環境変数, 63, 178
 follow_fork_mode_inner 環境変数, 64
 follow_fork_mode 環境変数, 63, 146, 178
 fork 関数、追跡, 178
 Fortran
 大文字/小文字を区別, 202
 間隔式, 211-212
 組み込み関数, 210
 構造, 213
 配列断面化の構文, 125-126
 派生型, 213
 複合式, 210
 論理演算子, 212
 割り当て可能配列, 209
 fortran_module コマンド, 319-320
 FPE シグナル、トラップする, 185-187
 frame コマンド, 113
 frame コマンド, 74, 320
 funcs コマンド, 321-322
 func コマンド, 70-71, 72, 74, 320-321

G

-g オプションを使用しないでコンパイルされたコード, 54
 -g コンパイラオプションを使用, 51
 gdb コマンド, 322

H

handler コマンド, 266, 323
 -hidden イベント指定修飾子, 278

hide コマンド, 114, 323-324

I

-if イベント指定修飾子, 276
 ignore コマンド, 184, 185, 324
 import コマンド, 324-325
 inclass イベント, 267
 infunction イベント, 267
 inmember イベント, 267
 inmember の停止コマンド, 98
 inmethod イベント, 267
 inobject イベント, 267
 In Object ブレークポイント, 99
 input_case_sensitive 環境変数, 64, 202
 -instr イベント指定修飾子, 278
 Intel レジスタ, 244
 intercept コマンド, 191, 325
 in イベント, 266
 -in イベント指定修飾子, 277

J

JAR ファイル、デバッグ, 219-220
 JAVASRCPATH 環境変数, 64, 218
 Java アプリケーション
 64 ビットライブラリを必要とする, 221
 dbx でデバッグできる種類, 218
 接続 dbx, 220
 デバッグの開始, 218
 独自のラッパーを指定する, 225
 ラッパー、デバッグ, 220
 Java アプリケーションを埋め込む C++ アプリケーション、デバッグ, 221
 Java アプリケーションを埋め込む C アプリケーション、デバッグ, 221
 Java クラスファイル、デバッグ, 219
 Java コード
 dbx の機能, 217
 dbx の使用, 217-218
 dbx の制限, 217
 dbx のデバッグモード, 226
 java コマンド, 326

Java ソースファイル、場所を指定する, 222
 Java デバッグ、環境変数, 218
 jclassess コマンド, 326
 jdbx_mode 環境変数, 64, 218
 joff コマンド, 326
 jon コマンド, 327
 jpkgs コマンド, 327
 jvm_invocation 環境変数, 64, 218
 JVM ソフトウェア
 64 ビット JVM ソフトウェアの指定, 226
 run 引数を渡す, 222, 224
 起動方法のカスタマイズ, 223
 パス名を指定する, 224

K

kill コマンド, 55-56, 139, 327-328
 Korn シェル
 dbx との違い, 251
 拡張, 252
 実装されていない機能, 251
 名前が変更されたコマンド, 252
 Korn シェルと dbx コマンドの違い, 251

L

language_mode 環境変数, 64
 language コマンド, 328
 lastrites イベント, 274
 LD_AUDIT, 150
 librttc.so の読み込み, 150
 librttc.so、読み込み, 150
 librtld_db.so, 256
 libthread_db.so, 169
 libthread.so, 169
 line コマンド, 72, 328-329
 listi コマンド, 237-238, 331
 list コマンド, 72, 74
 インスタンス化された関数のソースリストを出力, 199
 構文, 329-330
 使用してファイルまたは関数のソースリストを出力, 71

loadobject -dumpelf コマンド, 332
 loadobject -exclude コマンド, 332-333
 loadobject -hide コマンド, 333
 loadobject -list コマンド, 333-334
 loadobject -load コマンド, 334
 loadobject -unload コマンド, 334-335
 loadobject -use コマンド, 335
 loadobject コマンド, 331-335
 lwp_exit イベント, 271
 lwps コマンド, 175, 336-337
 -lwp イベント指定修飾子, 278
 LWP (軽量プロセス), 169
 情報について, 175
 情報を表示, 175
 lwp コマンド, 335-336

M

mmapfile コマンド, 337
 modules コマンド, 85, 338-339
 module コマンド, 84, 337-338
 mt_scalable 環境変数, 64

N

native コマンド, 339
 nexti コマンド, 238, 341-342
 next イベント, 272
 next コマンド, 90, 339-341

O

OpenMP アプリケーションプログラミングインタフェース, 179
 OpenMP コード
 dump コマンドの使用, 182
 shared、private、および threadprivate 変数の出力, 181
 コンパイラによる変換, 180
 実行シーケンス, 182
 シングルステップ, 181
 スタックトレースの使用, 181-182

OpenMP コード (続き)

利用可能な dbx の機能, 181
optim_local_vars 環境変数, 64
output_auto_flush 環境変数, 64
output_base 環境変数, 65
output_class_prefix 環境変数, 65
output_derived_type 環境変数, 121
output_dynamic_type 環境変数, 65, 190
output_inherited_members 環境変数, 65
output_list_size 環境変数, 65
output_log_file_name 環境変数, 65
output_max_string_length 環境変数, 65
output_no_literal 環境変数, 65
output_pretty_print 環境変数, 65
output_short_file_name 環境変数, 65
overload_function 環境変数, 65
overload_operator 環境変数, 65

P

pathmap コマンド, 85, 165, 342-343
 使用してコンパイル時ディレクトリをデバッグ
 時ディレクトリにマッピングする, 50
-perm イベント指定修飾子, 278
pop_auto_destruct 環境変数, 65
pop コマンド
 構文, 343-344
 使用して現在のスタックフレームを変更, 74
 使用してコールスタックフレームを1つ上がる,
 168
 使用して呼び出しスタックからフレームを
 ポップ, 260
 フレームを呼び出しスタックから削除, 114
pretty-print, 使用, 118
print コマンド
 C または C++ の配列の断面化の構文, 124
 Fortran の配列の断面化の構文, 125
 インスタンス化された関数やクラステンプレ
 ートのメンバーを評価, 199
 構文, 344-347
 使用して式の値を出力, 261
 使用して変数または式を評価, 118
 使用してポインタを間接参照, 122
proc_exclusive_attach 環境変数, 66

proc_gone イベント, 274
proc コマンド, 347
prog_new イベント, 274
prog コマンド, 347

Q

quit コマンド, 348

R

regs コマンド, 240-249, 348-349
replay コマンド, 56, 59, 349
rerun コマンド, 349-350
restore コマンド, 56, 58, 350
 -resumeone イベント指定修飾子, 105, 277
returns イベント, 273
rprint, コマンド, 350
rtc_auto_continue 環境変数, 66, 132, 154
rtc_auto_suppress 環境変数, 66, 144
rtc_biu_at_exit 環境変数, 66
rtc_error_limit 環境変数, 66, 145
rtc_error_log_file_name 環境変数, 66, 132, 154
rtc_error_stack 環境変数, 66
rtc_inherit 環境変数, 66
rtc_mel_at_exit 環境変数, 66
rtc showmap コマンド, 351
rtc skippatch コマンド, 351
rtld, 255-256
run_autostart 環境変数, 66
run_io 環境変数, 67
run_pty 環境変数, 67
run_quick 環境変数, 67
run_savetty 環境変数, 67
run_setpgrp 環境変数, 67
runargs コマンド, 353
run コマンド, 88, 352

S

save コマンド, 56, 353-354
scope_global_enums 環境変数, 67

- scope_look_aside 環境変数, 67, 79
- scopes コマンド, 354
- search コマンド, 354
- session_log_file_name 環境変数, 67
- showblock コマンド, 131, 355
- showleaks コマンド
 エラー制限, 145
 結果の報告, 138
 構文, 355-356
 使用してリークレポートを要求, 141
 デフォルトの出力, 142
- showmemuse コマンド, 142, 356-357
- sig イベント, 271
- source コマンド, 357
- SPARC レジスタ, 243
- stack_find_source 環境変数, 67, 73
- stack_max_size 環境変数, 68
- stack_verbose 環境変数, 68
- status コマンド, 357
- step_abflow 環境変数, 68
- step_events 環境変数, 68, 109
- step_granularity 環境変数, 68, 91
- stepi コマンド, 238, 360
- step to コマンド, 37, 90, 359
- step up コマンド, 90, 358
- step イベント, 273
- step コマンド, 90, 190, 358-359
- stop at コマンド, 96, 97
- stop change コマンド, 101
- stop inclass コマンド, 99
- stopi コマンド, 240, 365-366
- stop イベント, 274
- stop コマンド, 198
 C++ テンプレートクラスのすべてのメンバー
 関数にブレークポイントを設定, 197
 関数テンプレートのすべてのインスタンスにブ
 レークポイントを設定, 198
 構文, 360-365
 使用して C++ テンプレートクラスのすべての
 メンバー関数で停止, 197-198
- suppress_startup_message 環境変数, 68
- suppress コマンド
 構文, 366-368
 使用して RTC エラーの報告を制限, 132
- suppress コマンド (続き)
 使用して RTC エラーを管理, 146
 使用して RTC エラーを抑制, 144
 使用してデバッグ用にコンパイルされていない
 ファイル内の抑止されているエラーを表
 示, 145
- symbol_info_compression 環境変数, 68
- syncrtld イベント, 275
- syncs コマンド, 369
- sync イベント, 275
- sync コマンド, 368
- sysin イベント, 271-272
- sysout イベント, 272
- ## T
- temp イベント指定修飾子, 277
- thr_create イベント, 174, 275
- thr_exit イベント, 174, 275
- threads コマンド, 172, 371-372
- thread イベント指定修飾子, 278
- thread コマンド, 171, 369-370
- throw イベント, 275, 276
- timer イベント, 276
- trace_speed 環境変数, 68, 106
- tracei コマンド, 239, 376-377
- trace コマンド, 105, 372-376
- ## U
- uncheck コマンド, 131, 377-378
- undisplay コマンド, 123, 378-379
- unhide コマンド, 114, 379
- unintercept コマンド, 191, 379-380
- unsuppress コマンド, 144, 146, 380-381
- unwatch コマンド, 381
- up コマンド, 74, 112, 381-382
- use コマンド, 382
- ## W
- watch コマンド, 382

- whatis コマンド, 80, 81
 - 構文, 383-384
 - 使用してコンパイラによって割り当てられた関数名を取得, 122
 - テンプレートとインスタンスの定義の表示, 196-197
 - wheni コマンド, 386
 - when コマンド, 106, 261, 264, 384-386
 - when ブレークポイント、設定, 106
 - whereami コマンド, 388
 - whereis コマンド, 77, 117, 196, 388
 - where コマンド, 112, 207, 386-387
 - which コマンド, 71, 78, 117, 388-389
 - whocatches コマンド, 192, 389
- X**
- x コマンド, 234-236
- あ**
- あいまいな関数名をリストから選択する, 70
 - アクセシブルな製品マニュアル, 26-27
 - アクセス検査, 134
 - アセンブリ言語のデバッグ, 233
 - アドレス
 - 現在の, 72
 - 内容を調べる, 233-238
 - 表示形式, 235
 - アプリケーションファイルを再設定して再実行, 284
- い**
- 移動
 - 呼び出しスタックの指定フレームへ, 113
 - 呼び出しスタックを上へ, 112
 - 呼び出しスタックを下へ, 113
 - イベント
 - あいまいさ, 279
 - 解析, 279
 - 子プロセスの対話, 178
 - イベントカウンタ, 266
 - イベント固有の変数, 281
 - イベント指定, 240, 263, 265, 266-276
 - access, 268
 - at, 267
 - attach, 273
 - change, 268-269
 - cond, 269
 - detach, 274
 - dlopen, 269
 - exit, 272
 - fault, 270
 - in, 266
 - inclass, 267
 - infunction, 267
 - inmember, 267
 - inmethod, 267
 - inobject, 267
 - lastrites, 274
 - lwp_exit, 271
 - next, 272
 - prog_gone, 274
 - prog_new, 274
 - returns, 273
 - sig, 271
 - step, 273
 - stop, 274
 - sync, 275
 - syncrtld, 275
 - sysin, 271-272
 - sysout, 272
 - thr_create, 174, 275
 - thr_exit, 174, 275
 - throw, 275, 276
 - timer, 276
 - イベントのほかの型, 273
 - キーワード、定義, 266
 - システムイベントに対する, 269
 - 修飾子, 276-278
 - 進行イベント実行, 272-273
 - 設定, 266-276
 - 定義済み変数の使用, 279
 - データ変更イベント, 268-269
 - ブレークポイントイベント, 266-267

イベント指定修飾子

- count, 277
- disable, 277
- hidden, 278
- if, 276
- in, 277
- instr, 278
- lwp, 278
- perm, 278
- resumeone, 105, 277
- temp, 277
- thread, 278

イベント指定のための定義済み変数, 279

イベント発生後にブレークポイントを有効にする, 284

イベントハンドラ

- 隠す, 278
- 作成, 265
- 設定、例, 282-285
- 操作, 265
- デバッグセッション間で維持, 278

イベントハンドラの操作, 265

インスタンス、定義を表示, 194, 196

え

エディタのキーバインド、表示または変更, 252

エラーの抑止, 144-146

- 型, 144
- デフォルト値, 145
- 例, 145

演算子

- C++ コロンを重ねたスコープ決定, 75
- 逆引用符, 75
- ブロックローカル, 75

お

大文字/小文字を区別する、Fortran, 202

オブジェクトファイル

検索, 50, 85

オブジェクトポインタ型, 121-122

オンにする

- メモリアクセス検査, 40, 131
- メモリー使用状況検査, 40, 131
- メモリーリーク検査, 40, 131

か
型

- 調べる, 81-83
- 宣言、検索, 80-83
- 宣言の検索, 80-83
- 宣言の出力, 81
- 派生、Fortran, 213
- 表示, 80-83

カレントプロシージャとカレントファイル, 201
関数

- C++ コードにブレークポイントを設定, 99
- あいまいまたは多重定義された, 70

インスタンス化

ソースリストを出力, 199

評価, 199

呼び出し, 198

組み込み、Fortran, 210

クラステンプレートのメンバー、評価, 199

クラステンプレートのメンバー、呼び出し, 198

コンパイラで割り当てられた名前の取得, 122

実行中、変更, 166

実行、変更, 166

スタックにある、変更, 166

宣言の検索, 80-81

内容を表示する, 70-71

名前を特定する, 74-77

ブレークポイントの設定, 97-98

呼び出されていない、変更, 166

呼び出し, 92-93

関数テンプレートのインスタンス化

値の出力, 194

ソースコードの表示, 194

リストの出力, 194, 196

関数内のブレークポイント, 97

関数引数、無名

評価, 122

表示, 122

き

機械命令レベル

- AMD64 レジスタ, 247
- Intel レジスタ, 244
- SPARC レジスタ, 243
- アドレスにブレークポイントを設定する, 240
- アドレス、ブレークポイントを設定する, 240
- シングルステップ, 238
- すべてのレジスタの値を表示, 240
- デバッグ, 233
- トレース, 239-240

機械命令レベルでトレースする, 239-240

起動オプション, 306

逆引用符演算子, 75

共有オブジェクト

- 修正, 164
- 修正と継続, 256

共有ライブラリ

- dbx 用にコンパイル, 54
- ブレークポイントの設定, 257

切り離し

- dbx からのプロセスを, 90
- dbx からプロセスを, 55
- プロセスを dbx から切り離して停止状態にする, 90

く

クラス

継承されたすべてのデータメンバーを表示, 121

継承されたメンバーを表示, 82

調べる, 81-83

宣言の検索, 80-83

宣言の出力, 81

直接定義されたすべてのデータメンバーを表示, 121

表示, 80-83

クラステンプレートのインスタンス化、リストの出力, 194, 196

け

継承されたメンバー

表示, 81, 82

現在のアドレス, 72

検索

- オブジェクトファイル, 50, 85
- 現在位置, 112
- ソースファイル, 50, 85

こ

コアファイル

- 一致しないデバッグ, 46-48
- チェックする, 34-36
- デバッグ, 35, 44-48

コールスタック

- ポップ
- 1 フレーム, 168

子プロセス

- イベントと対話, 178
- 実行時検査を使用, 146-149
- 接続 dbx, 177
- デバッグ, 177

コマンド

- alias, 51
- assign
 - 構文, 287-288
 - 使用して値を変数に割り当て, 260
 - 使用して大域変数に正しい値を再び割り当て, 167
 - 使用して大域変数を復元, 168
 - 使用して変数に値を代入, 123

attach, 89, 288-289

bcheck, 153

bind, 252

bsearch, 289

call

- インスタンス化された関数やクラステンプレートのメンバー関数を明示的に呼び出し, 198

構文, 289-291

使用して関数を呼び出す, 92

使用して手続きを呼び出す, 92, 261

cancel, 291

コマンド (続き)

catch, 185, 186, 291-292
 check, 40, 131, 292-294
 clear, 294-295
 collector, 295-303
 collector archive, 297
 collector dbxsample, 297
 collector disable, 297-298
 collector enable, 298
 collector heaptrace, 298
 collector hw_profile, 298-299
 collector limit, 299
 collector mpitrace, 299
 collector pause, 299-300
 collector profile, 300
 collector resume, 300
 collector sample, 300-301
 collector show, 301
 collector status, 301
 collector store, 301-302
 collector synctrace, 302
 collector tha, 302
 collector version, 303
 cont, 166, 172
 構文, 303
 使用して大域変数の復元後に実行を継
 続, 168
 使用して別の行からプログラムの実行を継
 続, 91, 167, 262
 デバッグの情報なしでコンパイルできる
 ファイルの制限, 164
 プログラムの実行の継続, 91, 132
 dalias, 303-304
 dbx, 43, 48, 304-307
 dbxenv, 50-51, 62, 307
 debug
 構文, 307-310
 子プロセスに接続, 177
 使用して dbx を実行中のプロセスへ接続, 89
 使用してコアファイルをデバッグ, 45
 delete, 310-311
 detach, 55, 90, 311
 dis, 72, 236-237, 311-312
 display, 122, 312-313

コマンド (続き)

down, 113, 314
 dump, 314
 dump
 OpenMP コードの使用, 182
 edit, 314-315
 examine, 72, 234-236, 315-316
 exception, 190, 316-317
 exists, 317
 file, 70, 72, 317
 files, 318
 fix, 164, 165, 262, 318-319
 効果, 165
 デバッグの情報なしでコンパイルできる
 ファイルの制限, 164
 fixed, 319
 fortran_modules, 319-320
 frame, 113, 320
 func, 70-71, 72, 320-321
 funcs, 321-322
 gdb, 322
 handler, 266, 323
 hide, 114, 323-324
 ignore, 184, 185, 324
 import, 324-325
 inmember の停止, 98
 intercept, 191, 325
 java, 326
 jclassses, 326
 joff, 326
 jon, 327
 jpkgs, 327
 kill, 55-56, 139, 327-328
 language, 328
 line, 72, 328-329
 list, 72
 インスタンス化された関数のソースリスト
 を出力, 199
 構文, 329-330
 使用してファイルまたは関数のソースリス
 トを出力, 71
 listi, 237-238, 331
 loadobject, 331-335
 loadobject -dumpelf, 332

コマンド (続き)

loadobject -exclude, 332-333
loadobject -hide, 333
loadobject -list, 333-334
loadobject -load, 334
loadobject -unload, 334-335
loadobject -use, 335
lwp, 335-336
lwps, 175, 336-337
mmapfile, 337
module, 84, 337-338
modules, 85, 338-339
native, 339
next, 90, 339-341
nexti, 238, 341-342
pathmap, 85, 165, 342-343
 使用してコンパイル時ディレクトリをデ
 バッグ時ディレクトリにマッピング, 50
pop, 74, 114, 168, 260
 構文, 343-344
print
 CまたはC++の配列の断面化の構文, 124
 Fortranの配列の断面化の構文, 125
 インスタンス化された関数やクラステンプレートのメンバー関数を評価, 199
 構文, 344-347
 使用して式の値を出力, 261
 使用して変数または式を評価, 118
 使用してポインタを間接参照, 122
proc, 347
prog, 347
quit, 348
regs, 240-249, 348-349
replay, 56, 59, 349
rerun, 349-350
restore, 56, 58, 350
rprint, 350
rtc showmap, 351
rtc skippatch, 351
run, 88, 352
runargs, 353
save, 56, 353-354
scopes, 354
search, 354

コマンド (続き)

showblock, 131, 355
showleaks
 エラー制限, 145
 結果の報告, 138
 構文, 355-356
 使用してリークレポートを要求, 141
 デフォルトの出力, 142
showmemuse, 142, 356-357
source, 357
status, 357
step, 90, 190, 358-359
step to, 37, 90, 359
step up, 90, 358
steppi, 238, 360
stop, 198
 C++テンプレートクラスのすべてのメン
 バー関数にブレークポイントを設定, 197
 関数テンプレートのすべてのインスタ
 ンスにブレークポイントを設定, 198
 構文, 360-365
 使用してC++テンプレートクラスのすべて
 のメンバー関数で停止, 197-198
stop change, 101
stop inclass, 99
stopi, 240, 365-366
suppress
 構文, 366-368
 使用してRTCエラーの報告を制限, 132
 使用してRTCエラーを管理, 146
 使用してRTCエラーを抑止, 144
 使用してデバッグ用にコンパイルされてい
 ないファイル内の抑止されているエラー
 を表示, 145
sync, 368
syncs, 369
thread, 171, 369-370
threads, 172, 371-372
trace, 105, 372-376
tracei, 239, 376-377
uncheck, 131, 377-378
undisplay, 123, 378-379
unhide, 114, 379
unintercept, 191, 379-380

コマンド (続き)

- unsuppress, 144, 146, 380-381
 - unwatch, 381
 - up, 112, 381-382
 - use, 382
 - watch, 382
 - whatis, 80, 81
 - 構文, 383-384
 - 使用してコンパイラによって割り当てられた関数名を取得, 122
 - テンプレートとインスタンスの定義の表示, 196-197
 - when, 106, 261, 264, 384-386
 - wheni, 386
 - where, 112, 207, 386-387
 - whereami, 388
 - whereis, 77, 117, 196, 388
 - which, 71, 78, 117, 388-389
 - whocatches, 192, 389
 - x, 234-236
 - プログラムの状態を変更する, 260-262
 - プロセス制御, 87
- コンパイラで割り当てられた関数名の取得, 122
- コンパイルする
- g オプションを使用, 51
 - O オプションを使用, 51
 - 最適化コード, 51
 - デバッグを目的として, 31

さ

再開

- 特定の行からのプログラムの実行, 92
- マルチスレッドプログラムの実行, 172

最新エラーの抑止, 144

最適化コード

- コンパイルする, 51
- デバッグ, 52

削除

- 指定ブレークポイントをハンドラ ID を使用して, 108
- すべての呼び出しスタックフレームフィルタ, 114
- 阻止リストから例外型を, 191

削除 (続き)

- 呼び出しスタックから停止した関数, 113
- 呼び出しスタックフレーム, 114

作成

- .dbxrc ファイル, 62
- イベントハンドラ, 265

し式

- 値を監視, 122
- 値を出力, 118, 261
- 間隔, Fortran, 211-212
- 表示, 122
- 表示の終了, 123
- 複合, Fortran, 210
- 変更を監視, 122

式の値を監視, 122

シグナル

- dbx が受け付ける名前, 185
- FPE、トラップする, 185-187
- 現在捕獲されているシグナルのリストを表示する, 185
- 現在無視されているシグナルのリストを表示する, 185
- 自動処理, 187
- デフォルトのリストの変更, 185
- 転送, 183
- 取り消し, 183
- プログラム内で送信する, 187
- 捕獲, 184-187
- 無視, 185

システムイベント指定, 269

実験, サイズを制限, 299

実験のサイズを制限, 299

実行時検査

- アクセス検査, 134-137
- アプリケーションプログラミングインタフェース, 152
- エラー, 158-162
- エラーの抑止, 144-146
- エラー抑止のタイプ, 144
- エラーを抑止する, 144-146
- デフォルト値, 145

実行時検査, エラーを抑止する (続き)

- 例, 145
 - 子プロセス, 146-149
 - 最新エラーの抑止, 144
 - 修正と継続, 150-152
 - 使用時期, 130
 - 接続されたプロセス, 149-150
 - トラブルシューティングのヒント, 154
 - バッチモードでの使用, 152-154
 - 直接 dbx から, 154
 - 必要条件, 130-131
 - 無効化, 131
 - メモリアクセス
 - エラー, 136-137, 158-160
 - エラーの報告, 135-136
 - 検査, 134
 - メモリー使用状況検査, 142-143
 - メモリーリーク
 - エラー, 138, 160-162
 - エラーの報告, 139-142
 - 検査, 137-142
 - メモリーリークの修正, 142
 - リークの可能性, 138-139
- 修正
- C++ テンプレート定義, 168
 - 共有オブジェクト, 164
 - プログラム, 165, 262
- 修正と継続, 163
- 共有オブジェクトで使用, 256
 - 実行時検査での使用, 150-152
 - 制限, 164
 - ソースコードの修正, 164-165
 - 動作方法, 164
- 終了
- 監視中のすべての変数の表示, 123
 - 特定の変数または式の表示, 123
 - プログラム, 55-56
 - プログラムのみ, 55-56

出力

- OpenMP コードの shared、private、および threadprivate 変数, 181
- 型または C++ のクラスの宣言, 81
- 関数テンプレートのインスタンス化の値, 194
- 既知のスレッドすべてのリスト, 172

出力 (続き)

- 現在のモジュールの名前, 84
 - 式の値, 261
 - 指定のインスタンス化された関数のソースリストを出力, 199
 - シンボルの出現リスト, 77
 - すべてのクラスと関数テンプレートインスタンス化のリスト, 194, 196
 - すべてのマシンレベルレジスタの値, 240
 - ソースリスト, 71
 - 通常は出力されないスレッド (ゾンビ) のリスト, 172
 - データメンバー, 80
 - 配列, 123-128
 - フィールドの型, 81
 - 変数の型, 81
 - 変数または式の値, 118
 - ポインタ, 215
 - メンバー関数, 80
- 使用して値を変数に割り当て, 260
- 調べる
- this ポインタ, 81
 - 型の定義, 81-83
 - 関数の定義, 80-81
 - クラスの定義, 81-83
 - 変数の定義, 80-81
 - メンバーの定義, 80-81
- シングルステップ
- 機械命令レベルで, 238
 - プログラムを実行する, 91
- 進行イベント指定実行, 272-273
- シンボル
- 出現リストを出力, 77
 - 使用する dbx を決定する, 78
 - 複数存在する場合の選択, 70-71
- シンボルが複数存在する場合の選択, 70-71
- シンボル名、スコープを特定する, 74-77
- シンボル名を特定する, 74-77

す

- スコープ
 - 現在の, 69, 72
 - 検索規則、緩和, 79-80

- スコープ (続き)
 - 定義, 72
 - 表示, 73
 - コンポーネント, 73
 - 変更, 73
 - 表示の変更, 73-74
- スコープ決定演算子, 74-77
- スコープ決定検索パス, 78-79
- スタックトレース, 207
 - OpenMP コードの使用, 181
 - 表示, 114
 - 読み込み, 115
 - 例, 115
- スタックトレースの読み込み, 115
- スタックフレーム、定義, 111
- ストリップされたプログラム, 54
- スレッド
 - thread id に切り替える, 172
 - 既知のスレッドすべてのリストの出力, 172
 - 現在の、表示, 172
 - 現在のリスト、表示, 172
 - 通常出力されないスレッド (ゾンビ) リストの出力, 172
 - 表示される情報, 170-171
 - ブレークポイントに達した最初のスレッドのみを再開, 105
 - 別の、コンテキストを切り替える, 171
- スレッド作成、概要, 174

- せ
- セグメント例外
 - Fortran、原因, 205
 - 行番号の検出, 206
 - 生成, 205
- セッション、dbx
 - 開始, 43-44
 - 終了する, 55-56
- 接続
 - dbx 実行中の子プロセスへ, 177
 - dbx 実行中のプロセスへ, 48, 88-90
 - dbx が実行されていない場合, 89
 - 既存のプロセスのデバッグ中に dbx を新規のプロセスへ, 89
- 接続されたプロセス、実行時検査を使用, 149-150
- 設定
 - dbxenv コマンドによる dbx 環境変数, 62-68
 - トレース, 105
 - 非メンバー関数の複数のブレークポイント, 99
 - ブレークポイント
 - JVM ソフトウェアによって読み込まれていないコードに対する, 223
 - 同じクラスのメンバー関数, 99
 - オブジェクト内, 99-100
 - 関数テンプレートのすべてのインスタンス, 198
 - 関数呼び出しを含むフィルタ, 104
 - 異なるクラスのメンバー関数, 98
 - テンプレートクラスのメンバー関数またはテンプレート関数, 198
 - ブレークポイントのフィルタ, 103
- 宣言、検索 (表示), 80-83

- そ
- ソースファイル、検索, 50, 85
- ソースリスト、出力, 71

- た
- 断面化
 - C と C++ 配列, 124
 - Fortran 配列, 125-126
 - 配列, 126-127

- ち
- チェックポイント、一連のデバッグ実行を保存, 57

- つ
- 追跡
 - exec 関数, 178
 - fork 関数, 178

て

停止

- Ctrl+Cによってプロセスを, 94
- テンプレートクラスのすべてのメンバー関数, 197
- プログラム実行
 - 条件文が真と評価された場合, 102
 - 変数の値が変更された場合, 101
- プロセス実行, 55
- ディレクトリからディレクトリへの新たなマッピングを作成する, 50, 86
- データ変更イベント指定, 268-269
- データメンバー、出力, 80
- 手続き、呼び出し, 261
- デバッグ
 - g オプションを使用しないでコンパイルされたコード, 54
 - アセンブリ言語, 233
 - 一致しないコアファイル, 46-48
 - 機械命令レベル, 233, 238-240
 - コアファイル, 35, 44-48
 - 子プロセス, 177
 - 最適化コード, 52
 - マルチスレッドプログラム, 169
- デバッグ実行
 - 保存, 56-59
 - 保存された
 - 再現, 58-59
 - 復元, 58
- デバッグ情報
 - すべてのモジュールについての、読み込み, 84
 - モジュールについての、読み込み, 84
- デフォルト dbx 設定の調整, 61
- テンプレート
 - インスタンス化, 194
 - リスト印刷, 194, 196
- 関数, 194
- クラス, 194
 - メンバー関数内で停止, 197
- 宣言の検索, 81-83
- 定義を表示, 194, 196

と

- 動的リンカー, 255-256
- 独自のクラスローダーを使用するクラスファイルのパスの指定, 222
- 特定の型の例外の捕獲, 191
- どの変数を dbx が評価したか確認, 117
- トラブルシューティングのヒント、実行時検査, 154
- トリップカウンタ, 266
- トレース
 - 実装, 283
 - 設定, 105
 - 速度の制御, 106
 - リストの表示, 107
- トレース出力、ファイルに転送, 106
- トレース速度を制御, 106

な

- 内容を表示する
 - 関数, 70-71
 - ファイルの, 70
 - 呼び出しスタックの移動によって関数の, 71

は

配列

- Fortran, 208
- Fortran 95 割り当て可能配列, 209
- 刻み, 123, 127
- 断面化, 123, 126-127
 - CとC++の構文, 124
 - Fortran 構文, 125-126
- 断面化の構文、刻み, 124-126
- 範囲、超える, 205
- 評価, 123-128
- 配列の断面化の刻み, 127

判定

- 実行行数, 283
- 実行命令数, 284
- 使用するシンボル dbx, 78
- ソース行ステップの細分性, 91
- 浮動小数点例外 (FPE) の原因, 186

判定 (続き)

- 浮動小数点例外 (FPE) の場所, 186
- プログラムのクラッシュしている場所, 34
- ハンドラ, 263
 - 関数内で有効にする, 283
 - 作成, 264, 265
- ハンドラ ID、定義, 265

ひ

評価

- インスタンス化された関数やクラステンプレートのメンバー関数, 199

- 配列, 123-128
- 無名関数引数, 122

表示

- 型, 80-83
- 関数テンプレートのインスタンス化のソースコード, 194
- 基底クラスから継承されたすべてのデータメンバー, 121
- クラス, 80-83
- クラスで直接定義されたすべてのデータメンバー, 121
- 継承されたメンバー, 81
- シンボル、出現, 77
- スタックトレース, 114
- スレッドリスト, 172
- 宣言, 80-83
- テンプレート定義, 80
- テンプレートとインスタンスの定義, 194, 196
- 別のスレッドのコンテキスト, 171-172
- 変数, 80-83
- 変数と式, 122
- 変数の型, 81
- 無名関数引数, 122
- メンバー, 80-83
- 例外処理の型, 190
- 表示スコープ, 73
 - コンポーネント, 73
 - 変更, 73-74

ふ

- ファイル
 - 位置, 85
 - 検索, 50, 85
 - 内容を表示する, 70
 - 名前を特定する, 74-77
- フィールドの型
 - 出力, 81
 - 表示, 81
- 浮動小数点例外 (FPE)
 - 原因の判定, 186
 - 取得, 285
 - 場所の判定, 186
- ブレークポイント
 - stop 型, 96
 - 設定時期の設定, 69
 - trace 型, 96
 - when 型, 96
 - when 型
 - 行で設定, 106
 - イベント効率, 108-109
 - イベント指定, 266-267
 - イベント発生後に有効にする, 284
 - オブジェクト内, 99
 - 概要, 96-102
 - 関数内, 97
 - クリア, 107-108
 - 削除、ハンドラ ID を使用, 108
 - 制限, 107
 - 設定
 - C++ コード内での複数のブレーク, 98
 - JVM ソフトウェアによって読み込まれていないコードに対する, 223
 - あるアドレスに, 240
 - 同じクラスのメンバー関数, 99
 - オブジェクト内, 99-100
 - 関数テンプレートのインスタンス化, 197
 - 関数テンプレートのすべてのインスタンス, 198
 - 関数内, 36, 97-98
 - 関数プレートのインスタンス化, 194
 - 関数呼び出しを含むフィルタ, 104
 - 機械レベル, 240
 - 行, 36, 96-97

- ブレイクポイント, 設定 (続き)
 - 共有ライブラリ, 107, 257
 - クラステンプレートのインスタンス化, 194, 197
 - 異なるクラスのメンバー関数, 98
 - テンプレートクラスのメンバー関数またはテンプレート関数, 198
 - 明示的に読み込まれたライブラリ, 257-258
- 定義, 36, 95
- フィルタの設定, 103
- 複数、非メンバー関数に設定, 99
- 変数の変更時, 101
- 無効にする, 108
- 有効にする, 108
- リストの表示, 107-108
- ブレイクポイントをクリアする, 107-108
- フレーム、定義, 111
- プログラム
 - 実行, 87-88
 - dbx 下で、影響, 259-260
 - 実行継続, 91-92
 - 指定の行, 262
 - 修正後, 166
 - 実行する
 - すべての RTC を有効化, 132
 - 実行を停止
 - 条件文が真と評価された場合, 102
 - 変数の値が変更された場合, 101
 - 修正, 165, 262
 - 終了, 55-56
 - 状態、チェック, 285
 - シングルステップ実行, 91
 - ステップ実行, 90-94
 - ストリップされた, 54
 - 特定の行からの再開の実行, 92
 - マルチスレッド
 - 実行の再開, 172
 - デバッグ, 169
- プログラムの実行, 33-34, 87-88
 - dbx で、引数なしで, 34, 88
 - すべての RTC を有効化, 132
- プログラムの実行継続, 91-92
 - 指定の行, 91, 262
 - 修正後, 166
- プログラムをステップ実行する, 37, 90-94
- プログラムを読み込む, 32-33
- プロシージャリンクエッジテーブル, 256
- プロセス
 - Ctrl+C によって停止, 94
 - dbx から切り離して停止状態にする, 90
 - dbx からの切り離し, 55, 90
 - 子
 - 実行時検査を使用, 146-149
 - 接続 dbx, 177
 - 実行、dbx を接続する, 88-90
 - 実行を停止, 55
 - 接続された、実行時検査を使用, 149-150
 - プロセス制御コマンド、定義, 87
 - ブロックローカル演算子, 75
- へ
- ヘッダファイルの変更, 168
- ヘッダファイル、変更, 168
- 変更
 - 関数実行中, 166
 - 実行関数, 166
 - 修正後の変数, 167-168
 - スタックにある関数, 166
 - デフォルトのシグナルリスト, 185
 - 呼び出されていない関数, 166
- 変数
 - 値を出力, 118
 - 値を割り当て, 123, 260
 - イベント指定, 281
 - 修正後の変更, 167-168
 - 調べる, 39
 - スコープ外, 118
 - 宣言、検索, 80-83
 - 宣言の検索, 80-83
 - 定義された関数とファイルの表示, 117
 - どの変数を dbx が評価したか決定, 117
 - 名前を特定する, 74-77
 - 表示, 80-83
 - 表示の終了, 123
 - 変更を監視, 122
- 変数に値を割り当て, 123
- 変数の型、表示, 81

ほ

ポインタ

- 間接参照, 122

- 出力, 215

- ポインタを間接参照, 122

- 捕獲シグナルリスト, 185

- 捕獲ブロック, 190

保存

- チェックポイントとして一連のデバッグ実行を, 57

- デバッグ実行をファイルへ, 56-59

- 保存されたデバッグ実行の再現, 58-59

- 保存されたデバッグ実行の復元, 58

ポップ

- コールスタックの1フレーム, 168

- 呼び出しスタック, 113-114, 166, 260

ま

- マニュアル、アクセス, 25-28

- マニュアル索引, 25

- マルチスレッドプログラム、デバッグ, 169

む

- 無効化、実行時検査, 131

- 無視されているシグナルのリスト, 185

め

メモリー

- アドレスの内容を調べる, 233-238

- アドレス表示形式, 235

- 状態, 135

- 表示モード, 233-238

メモリーアクセス

- エラー, 136, 158-160

- エラーの報告, 135

- 検査, 134

- オンにする, 40, 131

- メモリー使用状況検査, 142-143

- オンにする, 40, 131

- メモリーの内容を調べる, 233-238

メモリーリーク

- エラー, 138, 160-162

- 検査, 137-142

- オンにする, 40, 131

- 修正, 142

- 報告, 139-142

メンバー

- 宣言、検索, 80-83

- 宣言の検索, 80-83

- 表示, 80-83

メンバー関数

- 出力, 80

- トレース, 105

- 複数のブレークポイントの設定, 98-100

- メンバーテンプレート関数, 194

も

モジュール

- 現在の、名前を出力, 84

- すでに dbx に読み取られたデバッグ情報を含む、リスト表示, 85

- すべてのプログラム、リスト表示, 85

- デバッグ情報, 84

- デバッグ情報付き、リスト表示, 85

よ

呼び出し

- インスタンス化された関数やクラステンプレートのメンバー, 198

- 関数, 92-93

- 手続き, 261

- メンバーテンプレート関数, 194

- 呼び出しオプション, 306

- 呼び出しスタック, 111

- 位置を検索, 112

- 移動, 71, 112

- down, 113

- up, 112

- 指定フレームへ, 113

- 確認する, 38

呼び出しスタック (続き)

削除

- すべてのフレームフィルタ, 114

- フレーム, 114

- 定義, 111

- 停止された関数を削除, 113

- フレーム、定義, 111

- フレームを隠す, 114

- ポップ, 113-114, 166, 260

呼び出しスタックの移動, 71, 112

呼び出しスタックフレームを隠す, 114

読み込み

- すべてのモジュールについてのデバッグ情報, 84

- モジュールについてのデバッグ情報, 84

ら

ライブラリ

- 共有、dbx 用にコンパイル, 54

- 動的なリンク、ブレークポイントを設定, 107

り

リストの表示

- トレース, 107

- ブレークポイント, 107-108

リスト表示

- 関数テンプレートインスタンス化, 80

- 現在捕獲されているシグナル, 185

- 現在無視されているシグナル, 185

- すでに dbx に読み込まれたデバッグ情報が入っているモジュールの名前, 85

- すべてのプログラムモジュールの名前, 85

- デバッグ情報付きのすべてのプログラムモジュール名, 85

- モジュールのデバッグ情報, 84

リンカー名, 77

リンクマップ, 256

れ

例外

- Fortran プログラム、検出, 206-207

- 型が捕獲される場所のレポート, 192

- 型、表示, 190

- 阻止リストから型を削除, 191

- 特定の型、捕獲, 191

- 浮動小数点、原因の判定, 186

- 浮動小数点、場所の判定, 186

例外が捕獲される場所のレポート, 192

例外処理, 190-193

- 例, 192

レジスタ

- AMD64 アーキテクチャー, 247

- Intel アーキテクチャー, 244

- SPARC アーキテクチャー, 243

- 値を出力, 240

ろ

ロードオブジェクト、定義, 255