Oracle® Solaris Studio 12.4: dbx コマンドによるデバッグ

Part No: E57216
2015 年 1 月

ORACLE

目次

このドキュメントの使用法

- **概要** – 対話形式のソースレベルのデバッグツールである `dbx` コマンド行デバッガを使用する方法について説明します。
- **対象読者** – アプリケーション開発者、システム開発者、設計者、サポートエンジニア
- **必要な知識** – Fortran、C、C++、または Java プログラミング言語に精通していること、および Oracle Solaris オペレーティングシステムまたは Linux オペレーティングシステムと UNIX® コマンドをある程度理解していること

製品ドキュメントライブラリ

この製品の最新情報や既知の問題は、ドキュメントライブラリ (https://docs.oracle.com/cd/E37069_01/) に含まれています。

フィードバック

このドキュメントに関するフィードバックを http://www.oracle.com/goto/docfeedback からお聞かせください。

♦ ♦ ♦ 第 1 章

1 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モニタリング、およびメモリーリークの検出も行えます。

dbx を使用すると、C、C++ (C++11 および C11 規格を含む)、または Fortran で記述されたアプリケーションをデバッグできます。また、いくつかの制限はありますが (241 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™ コードと C JNI (Java Native Interface) コードまたは C++ JNI コードが混在したアプリケーションをデバッグすることもできます。

dbxtool は、dbx のグラフィカルユーザーインタフェースを提供します。

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。

デバッグを目的としてコードをコンパイルする

対象のプログラムを、C コンパイラ、C++ コンパイラ、Fortran コンパイラ、および Java コンパイラによって受け入れられる -g オプションでコンパイルすることによって、dbx でのソースレベ

ルのデバッグ用に準備する必要があります。dbx はまた、C++11 および C11 規格で記述されたコードもサポートしています。詳細については、43 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 または dbxtool 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 コマンドを入力します。

```
$ dbx
```

dbxtool を起動するには、シェルプロンプトで dbxtool コマンドを入力します。

```
$ dbxtool
```

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
```

dbxtool 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。

```
$ dbxtool 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}
```

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
```

dbxtool コマンドを使用すると、プロセス ID を指定することにより、dbxtool を起動し、それを実行中プロセスに接続することができます。

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

プロセスのプロセス ID がわからない場合は、dbx コマンドに pgrep コマンドを含めてプロセスを検索し、そのプロセスに接続します。例:

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

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

dbx コマンドと起動オプションの詳細については、343 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するか、または 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
```

dbx を Java コードと C JNI (Java Native Interface) コードまたは C++ JNI コードを含む実行中プロセスに接続するには、次のように入力します。

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

詳細については、346 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。引数を付けないで 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*] [< *inputfile*] [> *output-file*]

例:

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

run コマンドを引数なしで繰り返した場合、そのプログラムは、以前の run コマンドの引数またはリダイレクトを使用して再起動します。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、396 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、394 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、正常に終了する可能性があります。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。アプリケーションにバグが含まれている場合は、メモリーフォルトまたはセグメント例外のために停止する可能性があります。

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確認します。27 ページの「コアファイルをチェックする」および 31 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。

- プログラムが正しくない結果を返している原因を特定するため。方法としては、次のものがあります。
  - プログラムの状態をチェックし、変数の値を確認できるように、実行を停止するためのブレークポイントを設定します。28 ページの「ブレークポイントの設定」および 31 ページの「変数の調査」を参照してください。
  - プログラムの状態がどのように変化するかをモニターするために、コードを 1 ソース行ずつステップ実行します。29 ページの「プログラムをステップ実行する」を参照してください。
- 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。実行時検査を使用すると、メモリーアクセスエラーやメモリーリークエラーなどの実行時エラーを検出したり、メモリー使用をモニターしたりすることができます。32 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」を参照してください。

コアファイルをチェックする

プログラムがクラッシュしている場所を特定するには、クラッシュ時のプログラムのメモリーイメージであるコアファイルの調査が必要になることがあります。`where` コマンドを使用すると、コアダンプ時にプログラムが実行されていた場所を特定できます。434 ページの「`where` コマンド」を参照してください。

注記 -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、次のように入力します。

```
$ 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。まず、`dbx` が起動され、コアファイルがロードされます。次に、`where` コマンドによってスタックトレースが表示されます。これにより、ファイル `foo.c` の 9 行目でクラッシュが発生したことが示されます。

```
% dbx a.out core
Reading a.out
```

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

コアファイルのデバッグの詳細については、36 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使用の詳細については、31 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。

注記 - プログラムがいずれかの共有ライブラリと動的にリンクされている場合は、コアファイルを、それが作成された同じオペレーティング環境で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のデバッグについては、38 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。

ブレークポイントの設定

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て dbx に制御を渡すようにするプログラム内のある場所のことです。バグが存在するのではないかと思わ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ら、プログラムの状態や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。(95 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設定すると、関数または手続き内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、99 ページの「関数へのブレークポイントの設定」および 405 ページの「stop コマンド」を参照してください。

また、停止ブレークポイントを設定して、ソースコードの特定の行で停止させることもできます。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、98 ページの「ソースコードの行へのブレークポイントの設定」および 405 ページの「stop コマンド」を参照してください。

停止させる行は、file コマンドを使用して現在のファイルを設定したあと、list コマンドを使用して停止させる関数のリストを表示することによって特定できます。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 (92 ページの「プログラムを継続する」および 342 ページの「cont コマンド」を参照)。

現在のすべて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
...
stopped in main at line 12 in file "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、プログラムの実際の状態を予測される状態と比較しながら、プログラムを 1 ソース行ずつステップ実行することもできます。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、現在のソース行内の指定された関数か、または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って特定される最後に呼び出された関数へのステップインを試みます。

一部の関数 (特に、printf などのライブラリ関数) は -g オプションでコンパイルされていない可能性があるため、dbx はこれらの関数にステップインできません。このような場合、step と next は同様の動作になります。

次の例は、step および next コマンドと、28 ページの「ブレークポイントの設定」で設定されたブレークポイントの使用を示しています。

```
(dbx) stop at 13
(3) stop at "t.c":13
(dbx) run
Running: a.out
stopped in main at line 13 in file "t.c"
   13           printit(msg);
(dbx) next
Hello world
stopped in main at line 14 in file "t.c"
   14   }

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

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、402 ページの「step コマンド」および 381 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、呼び出されたが、まだ対応する呼び出し側に戻っていない、現在アクティブなすべてのルーチンを表します。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), at 0xef763418
=>[2] printit(msg = 0x20a84 "hello world\n"), line 6 in "t.c"
  [3] main(argc = 1, argv = 0xefffe93c), line 13 in "t.c"
(dbx)
```

-g オプションでコンパイルされた関数の場合は、引数名とその型がわかっているため、正確な値が表示されます。デバッグ情報が存在しない関数の場合は、引数の 16 進数が表示されます。これらの数字に意味があるとはかぎりません。たとえば、上のスタックトレースでは、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。29 ページの「プログラムをステップ実行する」に示されていた例では printf に 2 つの引数しか渡されなかったため、意味があるのは $i0 から $i1 のレジスタの内容だけです。

-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。このような関数内で停止すると、dbx はスタックを下方向に検索して -g オプションでコンパイルされた関数 (この場合は printit()) を含む最初のフレームを見つけ、現在のスコープを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。

呼び出しスタックの詳細については、112 ページの「効率性に関する考慮事項」を参照してください。現在のスコープの詳細については、68 ページの「プログラムスコープ」を参照してください。

変数の調査

スタックトレースにはプログラムの状態を完全に表すための十分な情報が含まれている可能性がありますが、さらに多くの変数の値を確認することが必要になる場合があります。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 = 0xefffe8b4
```

データ変更ブレークポイントを使用して、変数や式の値がいつ変化したかを追跡できます (103 ページの「データ変更ブレークポイント (ウォッチポイント) の設定」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誤った使用をチェックします。*メモリー使用状況とメモリーリークの検査*では、未処理のヒープ領域をすべて追跡したあと、必要に応じて、またはプログラムの終了時に、使用可能なデータ領域をスキャンし、参照されていない領域を識別します。

メモリーアクセス検査、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。メモリーアクセス検査のみを有効にするには、次のように入力します。

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

メモリー使用状況とメモリーリークの検査を有効にするには、次のように入力します。

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

必要な種類の実行時検査を有効にしたら、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、各メモリーアクセスが実行される直前にその妥当性がチェックされるため、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。その場合は、現在のスタックトレースを表示するための `where` コマンドや、変数を調べるための `print` コマンドなどの `dbx` コマンドを使用できます。

注記 - 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。

実行時検査の使用の詳細については、第9章「実行時検査の使用」を参照してください。

dbx の終了

dbx セッションは、dbx を起動した時点から dbx を終了するまで実行されます。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、**quit** と dbx プロンプトに入力します。

```
(dbx) quit
```

dbx を起動し、プロセス ID を指定してそれを実行中プロセスに接続した場合は、デバッグセッションを終了しても、そのプロセスは終了せずに実行を継続します。dbx は、セッションを終了する前に暗黙的な detach を実行します。

dbx の終了の詳細については、50 ページの「デバッグの終了」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
```

♦ ♦ ♦ 第 2 章

2 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。

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、現在の場所、dbx で行う必要のある処理、dbx にどれだけ精通しているか、およびいずれかの dbxenv 変数を設定しているかどうかによって異なります。

dbx を完全に端末ウィンドウのコマンド行から使用することも、あるいは dbx のグラフィカルユーザーインタフェースである dbxtool を実行することもできます。dbxtool については、dbxtool のマニュアルページおよび dbxtool 内のオンラインヘルプを参照してください。

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

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
```

または

```
$ dbxtool 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}
```

Oracle Solaris 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できる 32 ビット dbx と、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できる 64 ビット dbx の 2 つの dbx バイナリが含まれています。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。

注記 - 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。32 ビットプログラムを Linux OS 上でデバッグするには、32 ビット dbx に dbx コマンドオプション -xexec32 を付けて起動するか、DBX_EXEC_32 環境変数を設定する必要があります。

64 ビット Linux OS 上で 32 ビット dbx を使用しているときに、結果として 64 ビットプログラムが実行される場合は、debug コマンドを使用したり、follow_fork_mode 環境変数を *child* に設定したりしないでください。64 ビットプログラムをデバッグするには、dbx を終了し、64 ビット dbx を起動してください。

---

dbx コマンドと起動オプションの詳細については、343 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いずれかの共有ライブラリと動的にリンクされた場合は、そのコアファイルを、それが作成された同じオペレーティング環境でデバッグしてください。dbx では、異なるバージョンまたはパッチレベルの Oracle Solaris オペレーティングシステムを実行しているシステム上で生成されたコアファイルなど、「一致しない」コアファイルのデバッグに対するサポートが制限されています。

注記 -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、次のコマンドを使用します。

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

または

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

次のコマンドを発行すると、dbx は、コアファイルからプログラム名を特定します。

```
$ dbx - core
```

または

```
$ dbxtool - core
```

dbx がすでに実行されている場合は、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
```

プログラム名を - に置き換えた場合、dbx は、コアファイルからプログラム名を抽出しようとします。コアファイル内でそのフルパス名を取得できない場合は、dbx が実行可能ファイルを見つけることができない可能性があります。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内に存在しない場合は、そのパス名 (/tmp/core など) を指定できます。

コアダンプ時にプログラムが実行されていた場所を特定するには、where コマンドを使用します。

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。シングルステップ実行はできませんが、ブレークポイントを設定し、プログラムを再実行することができます。

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 (limit(1) のマニュアルページを参照) を使用してコアファイルの最大サイズを設定できます。Bourne シェルおよび Korn シェルでは、ulimit コマンド (limit(1) のマニュアルページを参照) を使用します。シェルの起動ファイル内のコアファイルのサイズに関する制限を変更し、その起動ファイルを再度読み込んでから、コアファイルを生成したプログラムを再実行することによって、完全なコアファイルを生成できます。

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。実行時リンカーの情報がないばあいは、ロードオブジェクトのリストを使用できません。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。軽量プロセス (LWP) のリストがない場合は、スレッド情報、LWP 情報、またはスタックトレース情報を使用できません。where コマンドを実行すると、プログラムがアクティブでなかったことを示すエラーが表示されます。

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。

- コアホスト上のプログラムによって使用される共有ライブラリが、dbx ホスト上のライブラリと同じライブラリでない可能性があります。ライブラリに関する正しいスタックトレースを取得するには、これらの元のライブラリを dbx ホスト上で使用できるようにしてください。
- dbx は、/usr/lib 内のシステムライブラリを使用して、システム上の実行時リンカーやスレッドライブラリの実装に関する詳細の理解に役立てます。dbx が実行時リンカーのデータ構造やスレッドのデータ構造を理解できるように、これらのシステムライブラリをコアホストからも提供することが必要になる可能性があります。

ユーザーライブラリやシステムライブラリは、パッチや Oracle Solaris オペレーティングシステムのメジャーアップグレードで変更される場合があるため、たとえば、コアファイルが収集されたあと、そのコアファイルに対して dbx を実行する前にパッチがインストールされた場合など、この問題は同じホスト上でも発生する場合があります。

一致しないコアファイルをロードすると、dbx によって、次のエラーメッセージの 1 つまたは複数が表示される可能性があります。

```
dbx: core file read error: address 0xff3dd1bc not available
dbx: warning: could not initialize librtld_db.so.1 -- trying libDP_rtld_db.so
```

```
dbx: cannot get thread info for 1 -- generic libthread_db.so error
dbx: attempt to fetch registers failed - stack corrupted
dbx: read of registers from (0xff363430) failed -- debugger service failed
```

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では機能しないため、2 つの引数をとる from-path to-path モードを使用してください。
- コアファイルのデバッグは、`dbx` ホストの Oracle Solaris オペレーティングシステムのバージョンがコアホストと同じか、またはそれより新しい場合、より適切に機能する可能性があります。ただし、この設定が必ずしも必要とは限りません。
- 必要になる可能性のあるシステムライブラリは次のとおりです。
  - 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用 (使用している `libthread` の実装によって異なる) :

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

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

    xxx`_db.so` ライブラリはターゲットプログラムの一部としてではなく、`dbx` の一部としてロードおよび使用されるため、`dbx` が 64 ビット対応バージョンの Oracle Solaris OS 上で実行されている場合は、これらのシステムライブラリの 64 ビットバージョンが必要です。

    `ld.so.1` ライブラリは `libc.so` やその他のライブラリなどのコアファイルイメージの一部であるため、そのコアファイルを作成したプログラムに一致する 32 ビットの `ld.so.1` ライブラリまたは 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のコアファイルを調べていて、`where` コマンドでスタックが表示されない場合は、`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
  current thread: t@0
  [1] 0x0(), at 0xffffffff
  ```

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

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) が壊れている場合に役立ちます。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。

スレッドスタックがないことは、thread_db.so.1 の問題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みることも必要になる場合があります。

▼ 共有ライブラリの問題を解消し、一致しないコアファイルをデバッグするには

1. **dbxenv 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が存在する場所を示します。**

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 経由でエクスポートされ、dbx ホストマシン上の /net/core-host/ を使用してアクセス可能な場合は、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ロードします。

   ```
   (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。単に libc.so.1 が使用できることを確認してください。

プロセス ID の使用

プロセス ID を dbx コマンドまたは dbxtool コマンドへの引数として使用して、実行中プロセスを dbx に接続できます。

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

または

```
dbxtool program-name processD
```

dbx を Java™ コードと C JNI (Java Native Interface) コードまたは C++ JNI コードを含む実行中プロセスに接続するには、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 - processID
```

または

```
$ dbxtool - processID
```

dbx はプログラム名を認識できないままであるため、run コマンド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「実行中プロセスへの dbx の接続」を参照してください。

dbx の起動シーケンス

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、dbx は、インストールされている起動ファイル dbxrc をディレクトリ /*install-dir*/lib 内で探します。デフォルトのインストールディレクトリは、Oracle Solaris プラットフォームでは /opt/solstudio12.4、Linux プラットフォームでは /opt/oracle/solstudio12.4 です。Oracle Solaris Studio ソフトウェアがデフォルトのディレクトリにインストールされていない場合、dbx は、dbxrc ファイルへのパスを dbx 実行可能ファイルへのパスから取得します。

そのあと、dbx は .dbxrc ファイルを現在のディレクトリで、次に $HOME で検索します。-s オプションを使用してファイルパスを指定することにより、.dbxrc とは異なる起動ファイルを明示的に指定できます。詳細については、55 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 コマンドを含めることができますが、一般には 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ています。ただし、特定のコマンドでは、プログラムがロードされているか、またはプロセスが接続されていることが必要です。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。起動ファイルはまた、source または .(ピリオド) コマンドを使用して、ほかのファイルをソースにしている場合もあります。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ロードすると、Reading *filename* などの一連のメッセージが出力されます。

プログラムのロードが完了すると、dbx は実行可能状態になり、そのプログラムのメインブロック (C または C++ の場合は main()、Fortran の場合は MAIN()) が表示されます。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。

起動プロパティーの設定

pathmap コマンド、dbxenv コマンド、および alias コマンドを使用すると、dbx セッションの起動プロパティーを設定できます。

コンパイル時ディレクトリのデバッグ時ディレクトリへのマッピング

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。

次のコマンドは、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、385 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、.dbxrc ファイル内に格納できます。

また、dbxenv 変数を設定することもできます。54 ページの「replay を使用した保存と復元」ファイルおよびこれら変数の設定方法について詳しくは、Saving and Restoring Using replayを参照してください。

詳細については、56 ページの「dbxenv 変数の設定」および 345 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、独自の dbx コマンドを作成できます。詳細については、342 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

対象のプログラムを、-g または -g0 オプションでコンパイルすることによって、dbx でのデバッグ用に準備する必要があります。

-g オプションでのコンパイル

-g オプションは、コンパイル中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++ コンパイラ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++ コンパイラの場合:

- 最適化レベルを指定せず、-g オプションのみを指定すると、デバッグ情報の取得が有効になり、関数のインライン化が無効になります。
- -g オプションを -O オプションまたは -xO*level* オプションとともに使用すると、デバッグ情報が有効になり、関数のインライン化は無効になりません。これらのオプションにより、限定されたデバッグ情報とインライン関数が生成されます。
- -g0 (ゼロ) オプションは、デバッグ情報をオンにし、関数のインライン化には影響を与えません。-g0 オプションでコンパイルされたコードの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、プログラムによるインライン関数の使用に応じて、リンク時間や dbx の起動時間を大幅に削減できます。

最適化されたコードを dbx で使用するためにコンパイルするには、-O (大文字 O) オプションと -g オプションの両方を使用して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 では、Linux プラットフォームの objcopy コマンド、および Oracle Solaris プラットフォームの gobjcopy コマンドの各オプションを使用して、実行可能ファイルから個別のデバッグファイルにデバッグ情報をコピーし、その情報を実行可能ファイルからストリップして、これらの 2 つのファイルの間にリンクを作成することができます。

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。

1. 実行可能ファイルを含むディレクトリ。
2. 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。
3. 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。これは、dbxenv 変数 debug_file_directory がそのディレクトリのパス名に設定されている場合は表示または変更できます。環境変数のデフォルト値は、/usr/lib/debug です。

たとえば、次の手順は、実行可能ファイル a.out の個別のデバッグファイルを作成する方法を説明しています。

▼ 個別のデバッグファイルを作成する方法

1. **デバッグ情報を含む a.out.debug という名前の個別のデバッグファイルを作成します。**

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

2. **a.out からデバッグ情報をストリップします。**

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

3. **2 つのファイルの間にリンクを確立します。**

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

Oracle Solaris プラットフォームでは、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 -help コマンドを使用して、この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。

補助ファイル (Oracle Solaris のみ)

デフォルトでは、ロードオブジェクトには割り当て可能なセクションと割り当て不可のセクションの両方が含まれています。割り当て可能なセクションは、実行可能コードとそのコードが実行時に必要とするデータが含まれているセクションです。割り当て不可のセクションには、実行時のファイルの実行には必要がない補足情報が含まれています。これらのセクションは、デバッガおよびその他の可観測性ツールの動作をサポートします。オブジェクト内の割り当て不可のセクションは、実行時にオペレーティングシステムによってメモリーに読み込まれないため、どのようなサイズであっても、メモリーの使用やその他の実行時パフォーマンスの側面に影響を与えません。

利便性のため、割り当て可能なセクションと割り当て不可のセクションは、通常どちらも同じファイルに保持されます。しかし、これらのセクションを分離した方が有用な場合があります。特

に、高度に最適化されたコードのきめ細かいデバッグをサポートするには、大量のデバッグデータが必要になります。最近のシステムでは、記述されるコードよりもデバッグデータの方が簡単に大きくなります。32 ビットオブジェクトのサイズは 4G バイトに制限されています。非常に大きな 32 ビットオブジェクトでは、デバッグデータのためにこの制限を超え、オブジェクトを作成できなくなる可能性があります。

従来より、これらの問題に対処するために、ロードオブジェクトからは割り当て不可のセクションがストリップされています。除去は有効ですが、あとで必要になる可能性があるデータも破棄されます。Oracle Solaris リンカーは、代わりに、割り当て不可のセクションを補助ファイルに書き込むことができます。この機能は、`-z ancillary` コマンド行オプションを使用して有効にします。

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

補助ファイルには、デフォルトでプライマリ出力オブジェクトと同じ名前と `.anc` ファイル拡張子が付けられます。ただし、`-z ancillary` オプションに *outfile* の値を指定することで、別の名前を付けることができます。

注記 - 補助ファイルの ELF 定義では、1 つのプライマリファイルと任意の数の補助ファイルが提供されます。現時点では、Oracle Solaris リンカーは、すべての割り当て不可のセクションを含む 1 つの補助ファイルのみを生成します。これは、将来変更される可能性があります。

`-z ancillary` が指定されると、リンカーは次のことを行います。

- すべての割り当て可能なセクションがプライマリファイルに書き込まれます。さらに、`SHF_SUNW_PRIMARY` セクションヘッダーフラグが設定された 1 つ以上の入力セクションを含むすべての割り当て不可のセクションがプライマリファイルに書き込まれます。
- 残りのすべての割り当て不可のセクションが補助ファイルに書き込まれます。
- 両方の出力ファイルは、次の既知の割り当て不可のセクションの完全な同一コピーを受信します。

.shstrtab	セクション名文字列テーブル。
.symtab	完全な非動的シンボルテーブル。
.symtab	`.symtab` に関連付けられたシンボルテーブルの拡張インデックスセクション。

.strtab	`.symtab` に関連付けられた非動的文字列テーブル。
.SUNW_ancillary	プライマリオブジェクトとすべての補助オブジェクトを識別したり、調査されているオブジェクトを識別したりするために必要な情報が含まれています。

- プライマリファイルとすべての補助ファイルには、セクションヘッダーの同じ配列が含まれています。各セクションは、すべてのファイルで同じセクションインデックスを持っています。
- プライマリファイルと補助ファイルはいずれも同じセクションヘッダーを定義しますが、ほとんどのセクションのデータは、前述のように 1 つのファイルに書き込まれます。セクションのデータが特定のファイル内に存在しない場合は、`SHF_SUNW_ABSENT` セクションヘッダーフラグが設定され、`sh_size` フィールドは 0 になります。

この構成により、セクションヘッダーの完全なリスト、完全なシンボルテーブル、およびプライマリファイルと補助ファイルの完全なリストのすべてを 1 つのファイルの検査から取得することが可能になります。

`dbx` は次に、実行可能ファイルで補助ファイルを探すことにより、これらの補助ファイルを `dbx` が個別のデバッグファイルを使用するのと同様に使用できます。コンパイル時には、次のように `-z ancillary` オプションを使用します。

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

プライマリロードオブジェクトとそれに関連付けられたすべての補助ファイルには、すべてのロードオブジェクトを識別して関連付けることができる .SUNW_ancillary セクションが含まれています。

詳細については、『Oracle Solaris 11.2 リンカーとライブラリガイド 』の第 2 章「リンカー」を参照してください。

注記 - この機能は現在、Oracle Solaris 11.1 でのみ使用できます。

最適化されたコードのデバッグ

`dbx` は、最適化されたコードに対する部分的なデバッグサポートを提供しています。サポートの範囲は、プログラムをコンパイルした方法によって大幅に異なります。

最適化されたコードを分析する場合は、次のことが可能です。

- 任意の関数の開始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
- ある行から別の行までシングルステップ実行する (`next` または `step` コマンド)

プログラムが (`-O` および `-g` オプションを使用して) 最適化とデバッグを同時に有効にしてコンパイルされている場合、`dbx` は制限モードで動作します。

どのコンパイラが、どのような環境下でどの種類のシンボリック情報を発行するかについての詳細情報は、リリースごとに変更される可能性があります。

ソース行の情報は使用できますが、あるソース行のコードが最適化されたプログラムの複数の異なる場所に現れる可能性があるため、ソース行でプログラムをステップ実行すると、オプティマイザがコードをどのようにスケジュールしたかによって、現在の行がソースファイル内の別の場所に移動します。

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。

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 ディレクティブを実装するために引き続きコードを処理するため、説明された問題のいくつかは `-xopenmp=noopt` で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。

パラメータと変数

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。

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あります。`dbx` が値を発見できない場合、発見できないことが報告されます。値は一時的に消失する場合があるため、再びシングルステップおよび出力を実行してください。

SPARC ベースのシステムおよび x86 ベースのシステム向けの Oracle Solaris Studio 12.2 コンパイラやそれ以降の Oracle Solaris Studio 更新は、パラメータおよび局所変数の場所

を特定するための情報を提供しています。GNU コンパイラの最近のバージョンも、この情報を提供しています。

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が、大域変数は表示したり、値を割り当てたりできます。

インライン関数

dbx を使用すると、インライン関数にブレークポイントを設定できます。呼び出し側で、インライン関数からの最初の命令の停止を制御します。インライン関数では、非インライン関数で実行できるのと同じ dbx 操作 (step、next、list の各コマンドなど) を実行できます。

where コマンドを実行すると、呼び出しスタックがインライン関数とともに表示されます。また、インラインパラメータの場所情報がある場合は、パラメータも表示されます。

呼び出しスタックを上下に移動する up および down コマンドも、インライン関数でサポートされています。

呼び出し側からのローカル変数は、インラインフレームにはありません。

レジスタがある場合は、これらは呼び出し側のウィンドウから表示されます。

コンパイラがインライン化する関数には、C++ インライン関数、C99 インラインキーワードを持つ C 関数、およびパフォーマンスにメリットがあるとコンパイラによって判断されたその他の関数が含まれます。

『Oracle Solaris Studio 12.4: パフォーマンスアナライザ 』には、最適化されたプログラムをデバッグときに役立つ可能性のある情報が含まれています。

-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必要がありますが、dbx では引き続き、-g なしでコンパイルされたコードに対する次のレベルのサポートが提供されます。

- バックトレース (dbx where コマンド)
- 関数の呼び出し (ただし、パラメータはチェックしない)
- 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

完全なサポートを受けるには、共有ライブラリも -g オプションで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機能します。たとえば、メモリー使用状況検査とアクセス検査は、strip -x でストリップされたコードで機能しますが、strip でストリップされたコードでは機能しません。

デバッグの終了

dbx セッションは、dbx を起動した時点から dbx を終了するまで実行されます。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、quit と dbx プロンプトに入力します。

```
(dbx) quit
```

dbx を起動し、プロセス ID オプションを指定することによってそれを実行中プロセスに接続した場合は、デバッグセッションを終了しても、そのプロセスは終了せずに実行を継続します。dbx は、セッションを終了する前に暗黙的な detach を実行します。

プロセス実行の停止

プロセスの実行は、dbx を終了することなく、Ctrl + C キーを押すことによっていつでも停止できます。

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、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 から切り離すには、detach コマンドを使用します。

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。

詳細については、349 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 セッション自体を保持し、dbx を別のプログラムのデバッグが可能な状態のままにします。

プログラムの強制終了は、dbx を終了することなく、デバッグしていたプログラムの残骸を削除するための良い方法です。詳細については、367 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、デバッグ実行のすべてまたは一部を保存し、それをあとで再生するための次の 3 つのコマンドが用意されています。

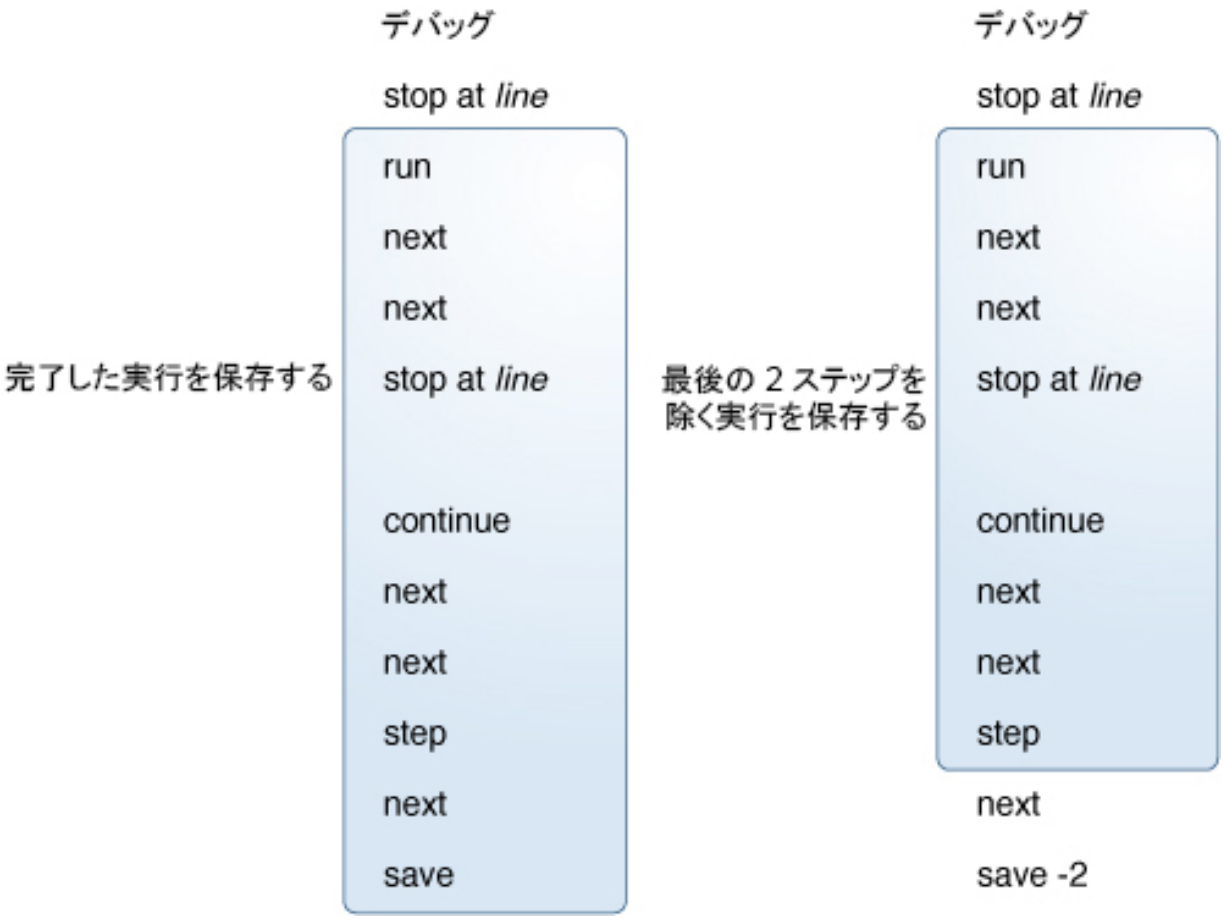
- save [-*number*] [*filename*]
- restore [*filename*]
- replay [-*number*]

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。デバッグセッションのこのセグメントは、「デバッグ実行」と呼ばれます。

発行されたデバッグコマンドのリストに加えて、save コマンドでは、実行開始時のプログラムの状態に関するデバッグ情報 (ブレークポイントや表示リストなど) が保存されます。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。



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。

注記 - デフォルトでは、save コマンドは特殊ファイルに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。53 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 コマンドは、デバッグ全体を保存する時点で発行します。

```
(dbx) save
```

デバッグ実行の一部を保存するには、*number* オプションを含めます。ここで、*number* は save コマンドから前に戻った、保存したくないコマンドの数です。

```
(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

ファイル名を指定せずにデバッグ実行を保存すると、dbx は特殊ファイルに情報を書き込みます。保存するたびに、dbx はこのファイルを上書きします。ただし、save コマンドに *filename* 引数を指定することにより、*filename* に保存されたデバッグ実行のあとにほかのデバッグ実行を保存した場合でも、あとで復元できるファイルにデバッグ実行を保存できます。

一連の実行を保存すると、それぞれがセッション内のさらに前から開始する一連の*チェックポイント*が提供されます。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。

デフォルト以外のファイルにデバッグ実行を保存するには、ファイル名を含めます。

```
(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、その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。

注記 - source コマンドも、ファイル内に格納された一連のコマンドを再発行しますが、dbx の状態はリセットしません。このコマンドは、現在のプログラムの場所から一連のコマンドを再発行するだけです。

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。

注記 - セグメントを保存してから、run、rerun、または debug コマンドを、restore を実行する前に発行すると、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
```

デフォルト以外のファイルに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 filename
```

replay を使用した保存と復元

replay コマンドは、save -1 を発行したあと、ただちに restore を発行するのと同じ、組み合わせのコマンドです。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 の部分に渡されます。デフォルトでは、-*number* の値は -1 であるため、replay コマンドは元に戻すコマンドとして機能し、最後の実行を最後に発行されたコマンドまで (ただし、このコマンドは除く) 復元します。

現在のデバッグ実行を、最後に発行されたデバッグコマンドを除いて再生するには、次のように入力します。

```
(dbx) replay
```

現在のデバッグ実行を再生し、特定のコマンドの前で実行を停止するには、-number オプションを使用します。ここで、*number* は最後のデバッグコマンドから前に戻ったコマンドの数です。

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

♦ ♦ ♦ 第 3 章

3 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 変数と、初期化ファイル .dbxrc を使用して、セッション間の変更と調整を保存する方法について説明します。

この章には次のセクションが含まれています。

dbx 初期化ファイルの使用

dbx 初期化ファイルは dbx を起動するたびに実行される dbx コマンドを保存します。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。

dbxrc ファイルには、コードを実行するコマンドを含めないでください。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 (dbx コマンドに -S オプションを指定しない限り) /*install-dir*¥/lib/dbxrc。デフォルトのインストールディレクトリは、Oracle Solaris プラットフォームでは /opt/solstudio12.4、Linux プラットフォームでは /opt/oracle/solstudio12.4 です。Oracle Solaris Studio ソフトウェアがデフォルトの *install-dir* にインストールされていない場合、dbx は dbxrc ファイルのパスを dbx 実行可能ファイル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

```
(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。

初期化ファイルのサンプル

次の例に、サンプル .dbxrc ファイルを示します。

```
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、dbxenv 変数を設定するために使用するコマンドです。dbxenv 変数の完全なリストについては、56 ページの「dbxenv 変数の設定」を参照してください。
- input_case_sensitive は、大文字と小文字の区別を制御する dbxenv 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。

dbxenv 変数の設定

dbxenv コマンドを使用して、dbx セッションをカスタマイズする dbxenv 変数を設定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

```
(dbx) dbxenv
```

変数の値を設定するには

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

表3-1「dbx 環境変数」は設定できるすべてのdbxenv 変数で構成されています。

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。
`c_array_op on | off`	C および C++ では、配列演算が可能です。たとえば、`a` と `b` が配列の場合、コマンド `print a+b` を使用できます。デフォルト値は `off` です。
`CLASSPATHX`	`dbx` にカスタムクラスローダーによってロードされる Java クラスファイルのパスを指定します。
`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_versionautodetect|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応じて、動的にモードを設定します。 x86 プラットフォーム: `dbx` の x86_32 または x86_64 用内蔵型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応じて、動的にモードを設定します。
`event_safety on | off`	`dbx` を安全でないイベント使用に対して保護します。デフォルト値は `on` です。
`filter_max_length` *num*	pretty-print フィルタによって、配列に変換されるシーケンスの最大長を *num* に設定します。
`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`	子を追跡する場合、ブレークポイントを継承するかどうかを決定します。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

dbx 環境変数	dbx 環境変数の機能
	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。
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、stop を選択した場合、この変数を設定して、cont -follow を使用する必要はありません。デフォルト値は unset です。
input_case_sensitive autodetect|true|false	autodetect に設定されている場合、dbx はファイルの言語に基づいて大文字と小文字の区別を自動的に選択します。Fortran 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。有効な設定は java、jni、または native です。
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、248 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。 ■ 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。
macro_expand on | off	on に設定した場合、選択された式のマクロ展開をグローバルに有効にします。デフォルト値は on です。
macro_source none | compiler | skim | skim_unless_compiler	dbx がマクロ情報を取得する場所を制御します。詳細については、322 ページの「スキミングエラー」を参照してください。デフォルト値は skim_unless_compiler です。
mt_resume_one on | off | auto	off に設定した場合、デッドロックを防ぐため、next コマンドによって呼び出しに対するステップを実行中にすべてのスレッドが再開されます。on に設定した場合、next コマンドによって呼び出しに対するステップを実行中に現在のスレッドのみが再開されます。auto に設定した場合、動作は off に設定した場合と同じです。ただし、プロ

dbx 環境変数	dbx 環境変数の機能
	グラムがトランザクション管理アプリケーションで、トランザクション内でステップ実行している場合は、現在のスレッドのみが再開されます。デフォルト値は auto です。
mt_scalable on|off	有効の場合、dbx はリソースの使用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。ただし、この設定により大幅に速度が低下する可能性があります。デフォルト値は off です。
mt_sync_tracking on | off	同期オブジェクトがプロセスを開始した際に、dbx が同期オブジェクトの追跡を有効にするかどうかを決定します。デフォルト値は off です。
output_auto_flush on|off	call 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。
output_base 8|10|16|automatic	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 automatic です (ポインタは 16 進文字、その他すべては 10 進)。
output_class_prefix on | off	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 1 つまたは複数の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。
output_derived_type on|off	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 -d にします。デフォルト値は off です。
output_dynamic_type on|off	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 -d にします。デフォルト値は off です。
output_inherited_members on|off	on の場合、出力、表示、および検査のデフォルト出力を -r にします。デフォルト値は off です。
output_list_size *num*	list 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。
output_log_file_name *filename*	コマンドログファイルの名前。 デフォルト値は /tmp/dbx.log.*unique-ID* です。 。
output_max_object_size *number*	変数の出力の最大バイト数を設定します。変数のサイズがこの数値より大きい場合、-L フラグを指定する必要があります。この dbxenv 変数はコマンド print、display、および watch に適用されます。デフォルト値は 4096 です。
output_max_string_length *number*	char *s で出力される文字数を設定します。デフォルト値は 096 です。

dbx 環境変数	dbx 環境変数の機能
output_no_literal on|off	有効にすると、式が文字列 (char *) 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。
output_pretty_print on|off	ウォッチポイントの出力および表示のデフォルトを -p に設定します。デフォルト値は off です。
output_pretty_print_fallback on|off	デフォルトで、問題が発生した場合、pretty-print は標準出力に戻されます。pretty-print 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。
output_pretty_print_mode call | filter | filter_unless_call	使用する pretty-print メカニズムを決定します。call に設定されている場合、呼び出し形式のプリティプリンタを使用します。filter に設定されている場合、Python ベースのプリティプリンタを使用します。filter_unless_call に設定されている場合、まず呼び出し形式のプリティプリンタを使用します。
output_short_file_name on|off	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。
overload_function on|off	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。
overload_operator on|off	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。
pop_auto_destruct on|off	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。
proc_exclusive_attach on|off	on 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。注意: プロセスに複数のツールが接続しており、それを制御しようとすると、予期しない結果が発生する可能性があります。デフォルト値は on です。
rtc_auto_continue on|off	rtc_error_log_file_name にエラーを記録して続行します。デフォルト値は off です。
rtc_auto_suppress on|off	on 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 n です。
rtc_biu_at_exit on|off|verbose	メモリー使用検査が明示的に、または check -all によって on になっている場合に使用されます。この値が on 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 verbose の場合は、プログラムの終了時に詳細メモリー使用状況レポートが作成されます。off の場合は出力は生成されません。デフォルト値は on です。
rtc_error_limit *number*	報告される RTC アクセスエラーの数。デフォルト値は 1000 です。
rtc_error_log_file_name *filename*	rtc_auto_continue 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 /tmp/dbx.errlog です。

dbx 環境変数	dbx 環境変数の機能
rtc_error_stack on|off	on に設定すると、スタックトレースは、RTC 内部メカニズムへ対応するフレームを示します。デフォルト値は off です。
rtc_inherit on|off	on 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 LD_PRELOAD が継承されます。デフォルト値は off です。
rtc_mel_at_exit on|off|verbose	リーク検査が on の場合に使用されます。この値が on 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 verbose 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。off の場合は出力は生成されません。デフォルト値は on です。
run_autostart on|off	アクティブでないプログラムで on に設定されている場合、step、next、stepi、および nexti は、暗黙的にプログラムを実行し、言語依存のmain ルーチンで停止します。on の場合、cont は必要に応じて run を暗黙指定します。デフォルト値は off です。
run_io stdio|pty	ユーザープログラムの入出力が、dbx の stdio か、または特定の pty にリダイレクトされるかどうかを指定します。pty は、run_pty によって指定します。デフォルト値は stdio です。
run_pty *ptyname*	run_io が pty に設定されているときに使用する pty の名前を設定します。Ptys はグラフィカルユーザーインタフェースのラッパーで使用されます。
run_quick on|off	on 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、prog -readsysms 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 off です。
run_savetty on | off	dbx とデバッグ対象のプログラムの間で、TTY 設定、プロセスグループ、およびキーボード設定 (-kbd 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 SIGTTIN または SIGTTOU を取得しシェルに戻る場合は、on に設定します。速度を多少上げるには off に設定します。設定は、dbx がデバッグ対象プログラムに接続されているか、Oracle Solaris Studio IDE で実行しているかに関係ありません。デフォルト値は off です。
run_setpgrp on | off	on に設定されている場合、プログラムの実行時に、フォークの直後に setpgrp(2) が呼び出されます。デフォルト値は off です。
scope_global_enums on | off	on 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します (~/ .dbxrc)。デフォルト値は off です。

dbx 環境変数	dbx 環境変数の機能
scope_look_aside on | off	on 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 on です。
session_log_file_name *filename*	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。
show_static_members	on の場合、印刷、監視、表示のデフォルトは -S になります。デフォルト値は on です。
stack_find_source on | off	on に設定した場合、デバッグ中のプログラムが -g 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 on です。
stack_max_size *number*	where 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。
stack_verbose on | off	where 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 on です。
step_abflow stop|ignore	stop に設定されていると、シングルステップ実行時に dbx が longjmp()、siglongjmp() で停止し、文を送出します。ignore に設定されていると、dbx は longjmp() および siglongjmp() の異常制御フロー変更を検出しません。デフォルト値は stop です。
step_events on |off	on に設定されている場合、ブレークポイントを許可する一方で、step および next コマンドを使用してコードをステップ実行します。デフォルト値は off です。
step_granularitystatement | line	ソース行ステップ実行の細分性を制御します。statement に設定されていると、次のコード: a(); b(); を、実行するための 2 つの next コマンドが必要です。line に設定されていると、1 つの next 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、行の細分性は特に有用です。デフォルト値は statement です。
suppress_startup_message *number*	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。
symbol_info_compression on|off	on に設定されている場合、各 include 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 on です。
trace_speed *number*	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	dbx 環境変数の機能
track_process_cwd on|off	on に設定されており、GUI が実行中のプロセスに接続されている場合、現在の作業ディレクトリが実行中のプロセスの作業ディレクトリに変更されます。デフォルト値は off です。
vdl_mode classic | lisp | xml	データ構造を dbx のグラフィカルユーザーインターフェース (GUI) に伝えるために、値記述言語 (VDL) を使用します。classic モードは、Sun WorkShop™ IDE で使用されました。lisp モードは、Sun Studio および Oracle Solaris Studio リリースの IDE で使用されます。xml モードは試験的なもので、サポートされていません。デフォルト: 値は GUI によって設定されます。

dbxenv 変数および Korn シェル

各 dbxenv 変数は、ksh 変数としてもアクセス可能です。ksh 変数名は dbxenv 変数から派生され、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 は同じ出力を生成します。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。

♦ ♦ ♦ 第 4 章

4 コードの表示とコードへの移動

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。

この章で説明する内容は次のとおりです。

コードへの移動

デバッグしているプログラムが停止するたびに、dbx は、その停止位置に関連付けられたソース行を出力します。プログラムが停止するたびに、dbx は、現在の関数の値をその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。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。移動によって、現在のスコープが設定されます (68 ページの「プログラムスコープ」を参照)。これは、stop at ブレークポイントをいつ、どのようなソース行で設定するかを判定するために役立ちます。

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するファイルであれば、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、どのファイルにも移動できます。ファイルに移動するには、次のように入力します。

```
(dbx) file filename
```

file コマンドを引数なしで使用すると、現在移動しているファイル名がエコー表示されます。

```
(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
```

詳細については、98 ページの「ソースコードの行へのブレークポイントの設定」を参照してください。

関数への移動

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。例:

```
(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、360 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++)

あいまいな名前または多重定義された関数名を持つ C++ メンバー関数に移動しようとすると、多重定義された名前を持つすべての関数を示すリストが表示されます。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。例:

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

which コマンドのコンテキストでは、出現リストから選択しても、dbx またはプログラムの状態には影響を与えません。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

ファイルまたは関数のソースリストを出力するには、list コマンドを使用します。ファイルを検索したあと、list コマンドは、上から *number* 行を出力します。デフォルトは 10 行です。関数を検索したあと、list コマンドはその行を出力します。

詳細については、369 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

動作中のプロセスが存在するときにコードに移動する別の方法として、「呼び出しスタックの移動」、つまりスタックコマンドを使用して、現在アクティブなすべてのルーチンを表す呼び出しスタック上に現在存在する関数を表示する方法があります。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「一番下」にあると見なされるため、そこから離れるには up コマンドを使用します。つまり、main または begin 関数に向けて移動します。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。

詳細については、116 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。

- 現在のアドレス。これは、dis コマンドおよび examine コマンドによって使用および更新されます。
- 現在のソースコード行。これは、list コマンドによって使用および更新されます。この行番号は、表示スコープを変更する一部のコマンドによって再設定されます。詳細については、69 ページの「表示スコープの変更」を参照してください。
- 現在の表示スコープ。これは、69 ページの「表示スコープ」で説明されている複合変数です。表示スコープは式の評価中に使用されます。これは、line コマンド、func コマンド、file コマンド、および list 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の観点で定義されたプログラムのサブセットのことです。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」ことになります。C では、関数にはグローバルまたはファイル固有のスコープが、また変数にはグローバル、ファイル固有、関数、またはブロックのスコープが存在する場合があります。

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。

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

これらの変数は、動作中のプロセス中にしか役立ちません。

表示スコープ

dbx でプログラムのさまざまな要素を検査すると、表示スコープが変更されます。dbx は、あいまいなシンボルの解決などの目的で、式の評価中に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、どの i に出力するかを決定します。

```
(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。スレッドを切り替えると、各スレッドはその表示スコープを返します。

表示スコープのコンポーネント

表示スコープの一部のコンポーネントは、次の事前に定義された ksh 変数で可視になります。

$vscope	現在の表示スコープ
$vloadobj	現在の表示ロードオブジェクト
$vfile	現在の表示ファイル
$vlineno	現在の表示行番号
$vclass	$vfunc が属するクラス
$vfunc	現在の表示関数

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` *number*
- `pop`
- `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` 環境変数が `on` に設定されている場合は、`dbx` は、ソースコードを含むスタックフレームを検索してアクティブにしようとします。

`up` コマンド、`down` コマンド、`frame` コマンド、または `pop` コマンドを使用して現在のスタックフレームを変更すると、`dbx` は、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表示スコープを設定します。

`list` コマンドによって使用される行番号位置は、`list` コマンド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、次のコマンドを実行すると、`dbx` は `my_func` のソースの先頭を表示し、表示スコープを `my_func` に変更します。

```
(dbx) list my_func
```

次のコマンドを実行すると、`dbx` は現在のソースファイル内の行 127 を表示しますが、表示スコープは変更しません。

```
(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、それに応じて表示スコープが更新されます。

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なったり、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。

このセクションでは、すべての種類のシンボル名修飾の規則について説明します。シンボル修飾規則は、すべての場合で同じです。

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
```

プログラムは、2 つの異なるファイルまたはコンパイルモジュールで同じ関数名を使用できます。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`filename`function-name
```

C++ 二重コロンスコープ決定演算子

次のような名前の C++ メンバー関数、トップレベル関数、または大域スコープを持つ変数を修飾するには、二重コロン演算子 (::) 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修飾しない場合は、dbx によって多重定義リストが表示されるため、どの関数を表示するかを選択できます。関数のクラス名がわかっている場合は、それを二重コロンスコープ決定演算子とともに使用することによって名前を修飾できます。

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

たとえば、hand がクラス名で、draw が関数名である場合は、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、特にネストされたブロック内の変数を参照できます。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。この行番号は、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の番数です。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されています。

```
(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.`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すよう特別な構文を使用します。シンボル名の前に # (ポンド記号) 文字を追加します。すべての $ (ドル記号) 文字の前に ksh のエスケープ文字 ¥ (バックスラッシュ) を使用します。

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

シンボルの検索

同じ名前が多くの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。dbx の which コマンドは、ユーザーが式の中でその名前を指定した場合に dbx がシンボルのどの出現を使用するかを表示します。

シンボルの出現を出力する

指定されたシンボルのすべての出現のリストを出力するには、whereis *symbol* を使用します。ここで、*symbol* にはユーザー定義の任意の識別子を指定できます。例:

```
(dbx) whereis table
forward: `Blocks`block_draw.cc`table
function: `Blocks`block.cc`table::table(char*, int, int, const point&)
class: `Blocks`block.cc`table
```

```
class: `Blocks`main.cc`table
variable:       `libc.so.1`hsearch.c`table
```

この出力には、プログラムが *symbol* を定義しているロード可能なオブジェクトの名前のほか、そのエンティティータイプであるクラス、関数、または変数が含まれています。

dbx シンボルテーブルからの情報は必要な場合に読み込まれるため、whereis コマンドは、すでにロードされているシンボルの出現のみを登録します。デバッグセッションが長くなると、出現リストが増大する場合があります。詳細については、79 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」を参照してください。

実際に使用されるシンボルを決定する

which コマンドは、ユーザーが式の中でその名前を完全に修飾しないで指定した場合に dbx が指定された名前を持つどのシンボルを使用するかを表示します。例:

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

指定されたシンボル名が局所的なスコープ内に存在しない場合、which コマンドは、そのシンボルの最初の出現をスコープ決定検索パスに沿って検索します。決定パスで最初に見つかった名前の完全修飾名が示されます。

検索パスに沿ったいずれかの場所で同じスコープレベルにある *symbol* の複数の出現が検索された場合、dbx は、あいまいさを報告するメッセージをコマンドペインに出力します。

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

dbx 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。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 *symbol* (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こ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。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。

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、その式の中のシンボルが次の順序で検索されます。dbx は、コンパイラが現在の表示スコープで処理するのと同様にシンボルを解決します。

1. 現在の表示スコープを使用している現在の関数のスコープ内。プログラムがネストされたブロックで停止された場合、dbx はそのブロック内で検索したあと、囲んでいる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. すぐ外側の囲んでいるモジュール。これは一般に、現在の関数を含むファイルです。
6. この共有ライブラリまたは実行可能ファイル専用に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. 上記の検索がいずれも成功しなかった場合、dbx は、ユーザーが別のファイル内の非公開 (つまり、ファイルの静的) 変数または関数を参照していると見なします。dbx はオプションで、dbxenv の設定 scope_look_aside の値に応じて、すべてのコンパイル単位内のファイルの静的シンボルを検索します。

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++ メンバー関数に対するスコープ検索規則を緩和するには、dbxenv 変数 scope_look_aside を on に設定します。

```
dbxenv scope_look_aside on
```

また、「二重逆引用符」接頭辞を使用することもできます。

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

dbxenv 変数 scope_look_aside が on に設定されている場合、dbx は次のものを探します。

- 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。

変数、メンバー、型、クラスを調べる

whatis コマンドは、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、429 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、whatis コマンドを使用します。

(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++ プログラムの場合、whatis は、関数テンプレートのインスタンス化を一覧表示します。テンプレート定義は、whatis -t で表示されます。77 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、次のコマンドを入力します。

```
(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 コマンドは、その変数の型を表示します。

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

フィールドを指定すると、whatis コマンドは、そのフィールドの型を表示します。

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

メンバー関数内で停止した場合は、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++ クラスの宣言を出力するには、次のように入力します。

```
(dbx) whatis -t type-orclassname
```

継承されたメンバーを表示するために、whatis コマンドは -r オプション (再帰を示します) を受け取ります。このオプションは、指定されたクラスの宣言を、そのクラスが基底クラスから継承するメンバーとともに表示します。

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

whatis -r の問い合わせからの出力は、クラス階層やクラスのサイズによっては長くなることがあります。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。

クラスの継承されたメンバーのルートを表示するために、whatis コマンドは、型定義のルートを表示する -u オプションを受け取ります。-u オプションを指定しないと、whatis コマンドは、値の履歴にある最後の値を表示します。これは、gdb で使用される ptype コマンドと同様です。

次の 2 つの例では、親クラス load_bearing_block の子クラスであるクラス table を使用しています。この親クラスはさらに、block の子クラスです。

-r を指定しないと、whatis は、クラス table で宣言されたメンバーを報告します。

```
(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);
};
```

次の例は、子クラスが継承するメンバーを表示するために、その子クラスに対して whatis -r が使用された場合の結果を示しています。

```
(dbx) whatis -t -r class table
class table : public load_bearing_block {
public:
  /* from base class table::load_bearing_block::block */
  block::block();
  block::block(char *name, int w, int h, const class point &pos, class load_bearing_block
 *blk);
    virtual char *block::type();
    char *block::name();
    int block::is_movable();
//  deleted several members from example protected:
    char *nm;
    int movable;
    int width;
    int height;
    class point  position;
    class load_bearing_block *supported_by;
    Panel_item panel_item;
```

```
    /* from base class table::load_bearing_block */
public:
    load_bearing_block::load_bearing_block();
    load_bearing_block::load_bearing_block(char *name, int w, int h,
        const class point &pos, class load_bearing_block *blk);
    virtual int load_bearing_block::is_load_bearing();
    virtual class list *load_bearing_block::supported_blocks();
    void load_bearing_block::add_supported_block(class block &b);
    void load_bearing_block::remove_supported_block(class block &b);
    virtual void load_bearing_block::print_supported_blocks();
    virtual void load_bearing_block::clear_top();
    virtual void load_bearing_block::put_on(class block &object);
    class point load_bearing_block::get_space(class block &object);
    class point load_bearing_block::find_space(class block &object);
    class point load_bearing_block::make_space(class block &object);
protected:
    class list *support_for;
    /* from class table */
public:
    table::table(char *name, int w, int h, const class point &pos);
    virtual char *table::type();
    virtual void table::draw(unsigned long pw);
};
```

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

最適な結果を得るには、ソースファイルを `-g` オプションを使用してコンパイルすることにより、プログラムをよりデバッグしやすくします。`-g` オプションを指定すると、コンパイラにより、デバッグ情報がプログラムのコードやデータとともにスタブまたは DWARF 形式でオブジェクトファイルに記録されます。

`dbx` は、必要に応じて、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ロードします。`module` コマンドを使用すると、いずれか特定のモジュール、またはすべてのモジュールのデバッグ情報をロードするよう `dbx` に要求できます。84 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ロード

オブジェクト (`.o`) ファイルがリンクされるとき、リンカーはオプションで、結果のロードオブジェクトにサマリー情報のみを格納できます。このサマリー情報は実行時に `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。作成される実行可能ファイルのディスク占有領域は小さくなりますが、dbx の実行時にそのオブジェクトファイルが使用可能である必要があります。

この要件は、オブジェクトファイルを -xs オプションでコンパイルして、これらの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まとめることによってオーバーライドできます。

オブジェクトファイルとともにアーカイブライブラリ (.a ファイル) を作成し、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。デバッグ情報が実行時にプロセスイメージにロードされないため、プログラムの実行が遅くなることはありません。

スタブ型式を使用した際のデフォルト動作では、コンパイラはサマリー情報のみを実行可能ファイルに入力します。

オブジェクトファイルは、-xs オプションを使用して DWARF で作成できます。詳細については、81 ページの「インデックス DWARF (-xs[={yes|no}])」を参照してください。

注記 - 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見えてしまいます。

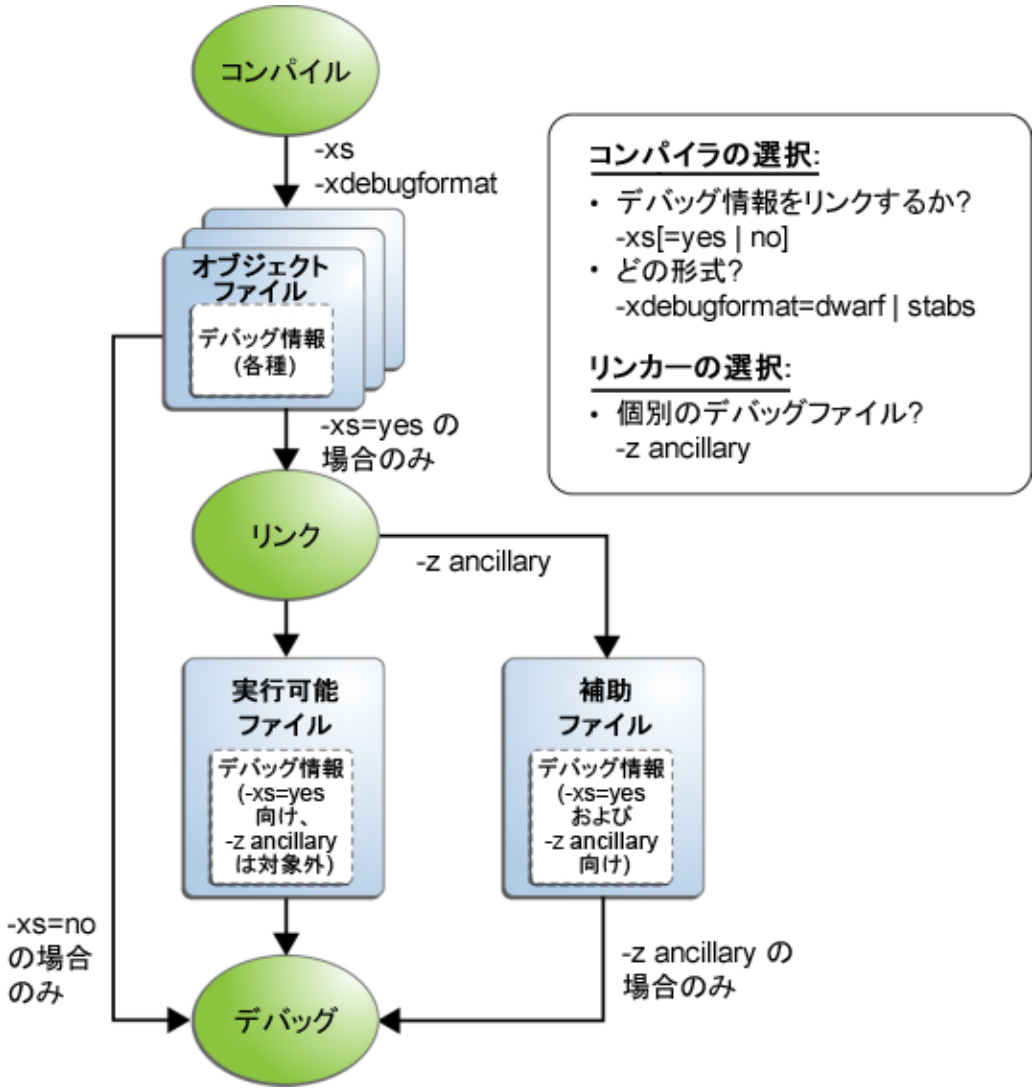
スタブのインデックスの詳細については、*install-dir*/solarisstudio12.4/READMEs/stabs.pdf のパスにあるスタブのインタフェースに関するガイドを参照してください。

デバッグをサポートするためのコンパイラおよびリンカーオプション

コンパイラおよびリンカーオプションにより、ユーザーはデバッグ情報をより自由に生成したり、使用したりできるようになります。コンパイラは、インデックススタブに似た DWARF のインデックスを生成します。このインデックスは常に存在し、これによって dbx の起動が速くなるほか、DWARF でのデバッグではその他の項目も改善されます。

次の図は、デバッグ情報のさまざまな種類と場所を示しています。ここでは、特にデバッグデータがどこに存在するかを強調しています。

図 4-1 デバッグ情報のフロー



インデックス DWARF (-xs[={yes|no}])

DWARF は、デフォルトでは実行可能ファイルにロードされます。新しいインデックスにより、-xs=no オプションで DWARF をオブジェクトファイル内に残すことが可能になります。これ

により、実行可能ファイルのサイズがより小さく、リンクがより高速になります。デバッグするには、これらのオブジェクトファイルを保持しておく必要があります。これは、スタブの動作と同様です。

個別のデバッグファイル (-z ancillary[=*outfile*])

Oracle Solaris 11.1 リンカーは、実行可能ファイルの構築中に、デバッグ情報を個別の補助ファイルに送信できます。個別のデバッグファイルは、すべてのデバッグ情報を移動、インストール、またはアーカイブする必要がある環境で役立ちます。実行可能ファイルは個別に動作できますが、その個別のデバッグファイルのコピーを使用して複数のユーザーがデバッグすることもできます。

`dbx` は、デバッグ情報を個別のファイルに抽出するための GNU ユーティリティー `objcopy` の使用を引き続きサポートしていますが、Oracle Solaris リンカーの使用には、`objcopy` に比べて次の利点があります。

- 個別のデバッグファイルはリンクの副産物として生成される
- 大きすぎて 1 つのファイルとしてリンクできなかったプログラムは 2 つのファイルとしてリンクされる

詳細については、45 ページの「補助ファイル (Oracle Solaris のみ)」を参照してください。

デバッグ情報の最小化

`-g1` コンパイラオプションは、配備されるアプリケーションのデバッグ可能性を最小限に抑えることを目的にしています。このオプションでアプリケーションをコンパイルすると、ファイルと行番号のほか、事後デバッグ中に重要であると見なされた単純なパラメータ情報が生成されます。詳細については、コンパイラのマニュアルページおよびコンパイラのユーザーガイドを参照してください。

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
```

ここでは:

`-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。

現在のモジュールの名前を出力するには、次のように入力します。

```
(dbx) module
```

モジュールの一覧表示

`modules` コマンドは、モジュール名を一覧表示することによってモジュールを追跡するのに役立ちます。

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一覧表示するには、次のように入力します。

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

デバッグ情報を含むかどうかには関係なく、すべてのプログラムモジュールの名前を一覧表示するには、次のように入力します。

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

デバッグ情報を含むすべてのプログラムモジュールを一覧表示するには、次のように入力します。

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

ここでは:

-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 は、プログラムに関連付けられたソースコードファイルの場所を認識している必要があります。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。ソースファイルを移動したか、またはそれらを新しい場所にコピーした場合は、プログラムを再リンクするか、デバッグの前に新しい場所に変更するか、または pathmap コマンドを使用する必要があります。

Sun Studio 11 以前のリリースの dbx で使用されていたスタブフォーマットでは、dbx のデバッグ情報は、オブジェクトファイルを使用してその他のデバッグ情報を読み込むことがあります。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。

ソースファイルへのパスを含むシンボリック情報は、実行ファイルに含まれています。dbx でソース行を表示する必要がある場合は、必要な分だけシンボリック情報を読み込んでソースファイルの位置を特定し、そこから行を読み取り、表示します。

シンボリック情報にはソースファイルのフルパス名が含まれていますが、dbx コマンドを入力するときは通常、ファイルのベース名のみを使用します。例:

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

dbx は、シンボリック情報内で、一致するファイルを検索します。

ソースファイルを削除している場合は、dbx でこれらのファイルのソース行を表示できませんが、スタックトレースを表示したり、変数値を出力したり、あるいは現在のソース行を特定したりすることもできます。

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ビューから実行可能イメージ内の名前へ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。

♦ ♦ ♦ 第 5 章

5 プログラム実行の制御

実行、ステップ実行、および実行の継続のため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。イベント管理コマンドとともに使用すると、dbx の下で実行されているプログラムの実行時動作を制御できます。

この章には次のセクションが含まれています。

プログラムの実行

プログラムをはじめて dbx にロードすると、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 コードの場合はメインクラス) に移動します。dbx はそのあと、ユーザーがコード内を移動するか、またはイベント管理コマンドを使用することにより、それ以上のコマンドを発行するのを待ちます。

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。

注記 - 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際に、まだロードされていないコードにブレークポイントを設定したいと考える場合があります。詳細については、247 ページの「ネイティブ (JNI) コードでブレークポイントを設定する」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。

必要に応じて、コマンド行引数や、入力に <、出力に > または >> を使用して、入力と出力のリダイレクトを追加できます。>> を使用すると、既存の出力ファイルに内容が追加されます。

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

注記 - 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。

注記 - run コマンドからの出力は、>> を使用していないかぎり (この場合は既存のファイルに追加されます)、dbx を実行しているシェルで noclobber を設定している場合でも既存の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。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。

実行中プロセスへの dbx の接続

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。実行中プロセスに接続する場合として、次のものが考えられます。

- 実行中のサーバーをデバッグしたいが、そのサーバーを停止または強制終了したくない場合。
- グラフィカルユーザーインタフェースを備えた実行中のプログラムをデバッグしたいが、そのプログラムを再起動したくない場合。
- プログラムが無限ループに入っており、そのプログラムを強制終了せずにデバッグしたい場合。

dbx を実行中のプログラムに接続するには、そのプログラムのプロセス ID 番号を dbx debug コマンドへの引数として使用します。

プログラムのデバッグを終了したら、detach コマンドを使用することにより、プロセスを終了することなくそのプログラムを dbx の制御から解放できます。

dbx を実行中プロセスに接続したあとに終了すると、dbx は、終了の前に暗黙的に切り離しを実行します。

dbx を dbx とは独立に実行されているプログラムに接続するには、attach コマンドまたは debug コマンドのどちらかを使用できます。

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

または

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

プログラム名を – (ダッシュ) に置き換えることができます。dbx は、プロセス ID に関連付けられたプログラムを自動的に検索し、それをロードします。

詳細については、346 ページの「debug コマンド」と 326 ページの「attach コマンド」を参照してください。

dbx が実行されていない場合は、次のように入力して dbx を起動します。

```
% 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。それを、dbx にロードされた任意のプログラムと同様に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 を新しい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。
- 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現在のプロセスのデバッグを開始した場合、dbx は、新しいプロセスに接続する前に現在のプロセスからの切り離しを実行します。

dbx を接続しようとしている先のプロセスが SIGSTOP シグナル、SIGTSTOP シグナル、SIGTTIN シグナル、または SIGTTOUT シグナルのために停止された場合、この接続は成功し、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
```

このプロセスは検査可能ですが、それを再開するには、cont コマンドを使用してそのプロセスに 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。158 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。それにより、切り離し時に -stop オプションを指定しないかぎり、そのプログラムは dbx とは独立した実行を再開します。

dbx に排他的アクセス権があるときにブロックされる可能性のあるほかの /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のままにすることができます。例:

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

詳細については、349 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 2 つのシングルステップ実行のための基本コマンドに加え、step up と step to という名前の step コマンドの 2 つの関連コマンドをサポートしています。next コマンドと step コマンドはどちらも、再度停止する前に 1 ソース行を実行します。

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、ユーザーが関数にステップ実行したあと、そのプログラムを呼び出し側の関数に返します。

step to コマンドは、現在のソース行内の指定された関数か、または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って特定される最後に呼び出された関数へのステップインを試みます。条件付き分岐のために関数呼び出しが実行されなかったり、現在のソース行ではどの関数も呼び出されなかったりする可能性があります。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。

next および step コマンドの詳細については、381 ページの「next コマンド」と 402 ページの「step コマンド」を参照してください。

シングルステップ動作の制御

指定された行数のコードをシングルステップで実行するには、dbx コマンド next または step を使用し、そのあとに実行したいコードの行数 [*n*] を指定します。

```
(dbx) next n
```

または

```
(dbx) step n
```

step_granularity dbxenv 変数によって、step コマンドや next コマンドがコードをステップ実行する単位が決定されます。この単位は statement または line のどちらかです。

step_events 環境変数は、ステップ実行中にブレークポイントを有効にするかどうかを制御します。

step_abflow 環境変数は、dbx が、異常な制御フロー変更が実行されようとし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。このタイプの制御フロー変更は、siglongjmp() または longjmp() の呼び出し、あるいは例外のスローが原因で発生することがあります。

詳細については、56 ページの「dbxenv 変数の設定」を参照してください。

特定の関数または最後の関数へのステップイン

現在のソースコード行から呼び出された関数にステップインするには、step to コマンドを使用します。

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

最後に呼び出された関数にステップインするには、次のように入力します。

```
(dbx) step to
```

次の 2 つの例では、step to を単独で使用すると foo にステップインします。

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

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

プログラムを継続する

プログラムがブレークポイントに達したか、または何らかのイベントが発生したあとにその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。

```
(dbx) cont
```

その変形である cont at *line-number* を使用すると、現在のプログラムの場所以外の行からプログラムの実行を再開するように指定できます。このオプションを使用すると、再コンパイルしなくても、問題の原因であることがわかっている 1 行以上のコードをスキップできます。

指定された行でプログラムの実行を継続するには、次のように入力します。

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

この行番号は、プログラムが停止されたファイルを基準にして評価されます。指定された行番号は、現在の関数のスコープ内にある必要があります。

cont at *line-number* コマンドを assign コマンドとともに使用すると、ある変数の値を誤って計算している可能性のある関数の呼び出しを含むコード行の実行を回避できます。誤って計算された値をすばやく調整するには、assign コマンドを使用してその変数に正しい値を割り当てます。その値を誤って計算することになる関数呼び出しを含む行をスキップするには、cont at *line-number* を使用します。

たとえば、プログラムが行 123 で停止したとします。行 123 は関数 how_fast() を呼び出します。この関数が変数 speed を正しく計算していません。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。次に、how_fast() の呼び出しをスキップして、行 124 でプログラムの実行を継続します。

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

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;}
```

詳細については、次を参照してください。

- 98 ページの「ソースコードの行へのブレークポイントの設定」
- 100 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」
- 101 ページの「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」

■ 101 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」
■ 431 ページの「when コマンド」

関数の呼び出し

プログラムが停止された場合は、dbx の call コマンドを使用して関数を呼び出すことができます。このコマンドは、呼び出された関数に渡す必要のあるパラメータの値を受け入れます。

手続きを呼び出すには、関数の名前を入力し、そのパラメータを指定します。例:

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

パラメータはオプションですが、関数名のあとに括弧を入力する必要があります。例:

```
(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、または stop in glyph -if animate() などの条件付き修飾子を使用することによって関数を暗黙的に呼び出したりすることができます。

C++ 仮想機能は、print コマンドか call コマンド、または関数呼び出しを実行するその他の任意のコマンドを使用することにより、ほかのすべての関数と同様に呼び出すことができます。

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。何らかのあいまいさが残る (たとえば、関数が -g でコンパイルされていない) 場合は、dbx によって、多重定義された名前のリストが表示されます。

関数が定義されているソースファイルが –g オプションでコンパイルされた場合、またはプロトタイプ宣言が現在のスコープで可視である場合、dbx は引数の数と型をチェックし、不一致があればエラーメッセージを発行します。それ以外の場合、dbx はパラメータの数をチェックせずに呼び出しを続行します。

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て、入出力バッファー内に格納されているすべての情報が確実に出力されるようにします。自動的なフラッシュを無効にするには、dbxenv 変数 output_auto_flush を off に設定します。

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。次に、where コマンドを入力すると、スタックトレースに dbx コマンドのレベルから発生した呼び出しが示されます。

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。そのあと、そのコマンドを再発行できます。あるいは、pop -c コマンドを使用して、すべてのフレームをデバッガから実行された最新の呼び出しまでポップすることもできます。

呼び出しの安全性

call コマンドを使用するか、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対象のプロセスを呼び出すと、すぐには現れない重大な障害が発生する可能性があります。例:

- 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。多くの場合は、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と、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、呼び出しを行なったスレッド以外のスレッドで副作用が現れる可能性があります。
- ブレークポイント条件で使用され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 (186 ページの「実行の再開」を参照)。

dbx によって行わ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 ... ではなく問題 (通常はセグメント例外) が検出された場合、dbx は次のいずれかのアクションを実行します。

- メモリーアクセスエラーの検出によって発生したコマンドを含む、すべての stop コマンドを無視します。
- 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 は、次の状況では安全な呼び出しを使用します。

- display コマンドによって出力される式の中で発生した呼び出し。失敗した呼び出しは、ic0->get _data() = <call failed> と表示されます。

  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、Ctrl+C (^C) を使用して停止できます。^C を使用してプロセスを停止すると、dbx は ^C を無視しますが、子プロセスがそれを SIGINT として受け入れて停止します。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。実行を再開するために、cont のオプションの修飾子 sig *signal-name* を使用する必要はありません。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。

イベント管理

イベントは、dbx への通知の原因となる、デバッグプロセス内での事象の発生です。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。イベントが発生した場合、dbx では、プロセスを停止するか、任意のコマンドを実行するか、または情報を出力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。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 (106 ページの「呼び出し元フィルタによるブレークポイントの修飾」を参照)。

イベントハンドラ、イベントの安全性、イベントの作成、イベント指定、その他のイベント管理トピックなどのイベント管理の詳細については、付録B イベント管理を参照してください。

♦ ♦ ♦ 第 6 章

6 ブレー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ークポイントです。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。

この章では、ブレー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。ブレークポイントおよびトレースの設定で使用できるイベント指定の完全な詳細については、292 ページの「イベント指定の設定」を参照してください。

この章には次のセクションが含まれています。

ブレークポイントの設定

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 – プログラムは stop コマンドで作成されたブレークポイントに到達すると停止します。このプログラムは、cont、step、または next などのほかのデバッグコマンドを発行するまで再開できません。

- when – 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停止し、dbx が 1 つまたは複数のデバッグコマンドを実行した後、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続行します。
- trace – trace 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ークポイントと異なりますが、トレースとブレークポイントは類似したイベントハンドラを共有します。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 trace 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、292 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。詳細については、第18章「機械命令レベルでのデバッグ」を参照してください。

注記 - 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際に、まだロードされていないコードにブレークポイントを設定したいと考える場合があります。これらのコードへのブレークポイントの設定の詳細については、247 ページの「ネイティブ (JNI) コードでブレークポイントを設定する」を参照してください。

ソースコードの行へのブレークポイントの設定

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できます。ここで、*n* はソースコードの行番号、*filename* はオプションのプログラムファイル名修飾子です。

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

例:

```
(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。

停止させる行を決定するには、file コマンドで現在のファイルを設定し、list コマンドで停止させる関数を使用します。次に、次の例に示すように、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 場所」イベントを指定する詳細については、293 ページの「at イベント指定」を参照してください。

関数へのブレークポイントの設定

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できます。

(dbx) **stop in** *function*

関数内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先頭でプログラムの実行を中断します。

dbx は次の状況を除いて、参照されている関数を特定できるはずです。

- 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 ` が付く関数を参照する場合
- リンカー名 (C++ でのマングル名) で関数が参照されている。この場合、dbx は、名前の前に # が付けられていれば、名前を受け付けます。詳細については、73 ページの「リンカー名」を参照してください。

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();
};
```

非メンバー関数で停止するには、次のコマンドでグローバル foo(int) にブレークポイントを設定します。

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

メンバー関数にブレークポイントを設定するには:

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

次のコマンドでは、dbx でグローバル関数 foo(int) またはグローバル関数 foo(double) のどちらを意味しているのかを判断できないため、明確にするためにオーバーロードしたメニューを表示させることができます。

```
stop in foo
```

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() と、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。

注記 - unique_member など、メンバー名が一意の場合、stop in unique_member を使用すれば十分です。メンバー名が一意でない場合は、stop in コマンドを使用して、意図するメンバーを指定するためのオーバーロードメニューに応答することができます。

関数内イベントの指定の詳細については、293 ページの「in イベント指定」を参照してください。

C++ プログラムへの複数のブレークポイントの設定

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。stop、when、または trace コマンドで、キーワード inmember、inclass、infunction、または inobject を使用して、C++ コードに複数のブレークポイントを設定できます。

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クラス固有のバリアント (同じメンバー関数名で異なるクラス) 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、295 ページの「inmember イベント指定」を参照してください。

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

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。

次のコマンドは、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します。

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

次のコマンドは、クラス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、295 ページの「inclass イベント指定」および405 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の選択により、大量のブレークポイントが挿入される可能性があるため、dbxenv 変数 step_events を必ず on に設定し、step および next コマンドの速度を上げるようにしてください。詳細については、112 ページの「効率性に関する考慮事項」を参照してください。

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 2 つのバージョンの関数が定義されていて、一方が int 型の引数、もう一方が float 型の引数を渡す場合、次のコマンドで、両方の関数にブレークポイントを配置します。

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

infunction イベントを指定する詳細については、294 ページの「infunction イベント指定」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

オブジェクト内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される操作をチェックします。

オブジェクト内ブレークポイントを使用して、特定のオブジェクトインスタンスで特定のメソッドが呼び出されたときに、プログラムの実行を停止します。たとえば、次のコードは f1->printit() が呼び出されたときにのみ stop を発生させます。

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

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

f1 に格納されるアドレスはブレークポイントを配置したオブジェクトを示します。これは、f1 内のオブジェクトがインスタンス化された後にのみ、このブレークポイントを作成できることを示します。

デフォルトで、オブジェクト内ブレークポイントは、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の実行を中断します。ブレークポイントをオブジェクトクラスのみに制限するには、-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が、オブジェクト foo の継承クラスに定義されたものには設定しない場合:

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

inobject イベントを指定する詳細については、295 ページの「inobject イベント指定」および 405 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 (ウォッチポイント) の設定

dbx でデータ変更ブレークポイント (またはウォッチポイントと呼ばれる) を使用して、変数の値または式が変更されたときに注意することができます。

特定アドレスへのアクセス時にプログラムを停止する

メモリーアドレスにアクセスされたときに実行を停止するには、stop access コマンドを使用します。

(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。有効なオプションは:

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アクセス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。記号式を指定すると、監視対象領域のサイズが自動的に推定されます。*byte-size-expression* を指定して、それをオーバーライドすることができます。さらに、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、その場合はサイズが必須です。

次の例では、コマンドはメモリーアドレス **0x4762** 以降のいずれかの 4 バイトが読み取られたあとに実行を停止します。

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

次の例では、変数 speed に書き込みが行われる前に実行が停止します。

```
(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ように指示するには、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、296 ページの「access イベント指定」および405 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、`stop change` コマンドを使用します。

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

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

- `dbx` は、指定した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、関数が初めて入力されて *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、297 ページの「change イベント指定」および405 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ながら、各ステップで値をチェックして、`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る場合、`dbx` はネストされた `user_routine2` をトレースしません。トレースでは、ライブラリの呼び出しとネストされた `user_routine2` の呼び出しがスキップされるためです。

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

variable の値の変更は、`user_routin2` 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に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、`dbx` はその操作を実行できない旨を通知するエラーメッセージを発行します。

注記 - change イベントよりも access イベントを使用した方が、データ変更の監視が高速になります。自動的にプログラムのシングルステップを実行する代わりに、access イベントはハードウェアまたはオペレーティングシステムのはるかに高速なサービスを利用します。

条件付きでプログラムを停止する

stop cond コマンドを使用して、条件文が true に評価された場合にプログラム実行を停止します。

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

条件が発生すると、プログラムは実行を停止します。

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 true に評価された行の*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。

条件イベントを指定する詳細については、297 ページの「cond イベント指定」および 405 ページの「stop コマンド」を参照してください。

ブレー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でオプションのイベントフィルタ修飾子もサポートします。もっとも単純なフィルタは、プログラムがブレークポイントかトレースハンドラに到達したあと、またはデータ変更ブレークポイントが発生したあとに、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、プログラムはブレー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。

フィルタを含むブレー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ークポイントでは、条件の構文解析を行うスコープはブレー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 (71 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。

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` にブレー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。

条件付きフィルタによるブレークポイントの修飾

フィルタを含むブレー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、コマンドの入力時に現在の言語でブール値または整数値を返す関数呼び出しも含めて、任意の有効な式を指定できます。

関数呼び出しをブレー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)
```

関数呼び出しで `-if` オプションを使用できます。

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

次に、ウォッチポイントで条件付きフィルタを使用する例を示します。

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

呼び出し元フィルタによるブレークポイントの修飾

経験の少ないユーザーは、条件付きイベントコマンド (ウォッチタイプのコマンド) の設定と、フィルタの使用を混同することがあります。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ー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。

次の例を考慮します。

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

このコマンドは、変数 `speed` をモニターするように *dbx* に指示します。変数 *speed* に書き込みが行われると (「ウォッチ」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 と等しいかどうかをチェックします。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、フィルタはブレークの「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]
```

次のようなコードのある類似の例を考慮します。

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

次のコマンドによって、open() の ENOENT などの特定のエラーで停止できます。

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

局所変数にデータ変更ブレークポイントを配置する際に、フィルタを使用すると便利です。次の例では、現在のスコープは関数 foo() 内にあり、対象となる変数 index は関数 bar() 内にあります。

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

bar`index により、関数 foo にある index() 変数や index という名称のグローバル変数ではなく、関数 bar にある index 変数が確実に取り出されます。

-in bar には、次のような意味があります。

- 関数 bar() に入ると、ブレークポイントが自動的に有効になります。
- bar() とそれが呼び出したすべての関数が有効の間は、ブレークポイントは有効の状態を保つ。
- bar() からの復帰時に、ブレークポイントは自動的に無効になります。

index に対応するスタック位置は、ほかのいずれかの関数のいずれかの局所変数によって再度利用できます。-in により、ブレークポイントが起動するのは bar`index がアクセスされた場合のみになります。

フィルタとマルチスレッド

マルチスレッドプログラムで関数呼び出しを含むフィルタのあるブレークポイントを設定すると、dbx はブレークポイントに達したときにすべてのスレッドの実行を停止し、条件を評価します。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、次のブレークポイントを、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ー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。

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

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

そのような場合に、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信できる場合は、-resumeone イベント指定修飾子を使用して、dbx に、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、次の状況で役立つ場合があります。

- 条件は再帰的に lookup() を呼び出すため、lookup() の 2 つ目のブレークポイントは最初のスレッドと同じスレッドで発生します。
- 条件実行が別のスレッドへの制御を放棄するスレッド。

詳細については、308 ページの「イベント指定修飾子」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 trace 情報行を出力したあと、処理を再開します。

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。

トレースの設定

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、418 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 (292 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

トレース出力の表示が速すぎる場合があります。`dbxenv` 変数 `trace_speed` を使用して、各トレースが出力された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。

トレース中のコードの各行の実行間隔を秒単位で設定するには:

`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、トレースの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。接続後に新たに実行するか再開すると、ファイルが常に再び開きます。

1 行での dbx コマンドの実行

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ますが、これは、ユーザーが独自のバージョンの trace トレースを書くことができることを意味します。

```
(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。例では、現在の行のソースコードを一覧表示したあと、プログラムが実行を続行します。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、431 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、308 ページの「イベント指定修飾子」を参照してください。

動的にロードされたライブラリにブレークポイントを設定する

dbx は、次のタイプの共有ライブラリと連動します。

- プログラムの実行開始時点で暗黙的にロードされたライブラリ。
- dlopen(2) を使用して明示的 (動的) にロードされたライブラリ。これらのライブラリにある名前は実行中にライブラリがロードされたあとにのみわかるため、debug または attach コマンドを使用してデバッグセッションを開始したあとに、それらにブレークポイントを配置することはできません。
- dlopen(2) を使用して明示的にロードされたフィルタライブラリ。これらのライブラリにある名前は、ライブラリがロードされて、その中の最初の関数が呼び出されたあとにのみわかります。

明示的 (動的) にロードされたライブラリにブレークポイントを設定するには、次の 2 つの方法があります。

- たとえば、mylibrary.so など、関数 myfunc() を含むライブラリがある場合、ライブラリの記号テーブルを dbx へプリロードし、次のようにブレークポイントを関数に設定できます。

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

- もっと簡単な方法は、プログラムを dbx の下で完了まで実行することです。dbx は、dlopen(2) を使用してロードされたすべての共有ライブラリを記録し、それらが

dlclose() を使用して閉じられた場合でも記憶しています。そのため、プログラムの最初の実行後に、ブレークポイントを正しく設定できるようになります。

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

ブレークポイントの一覧表示と削除

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

すべてのアクティブなブレークポイントのリストを表示するには、status コマンドを使用して、カッコまたは角カッコ内に ID 番号を表示します。これは後でほかのコマンドで使用できます。ID 番号が角カッコに囲まれている場合、これらのブレークポイントは無効にされています。さらに、アスタリスク (*) がカッコまたは角カッコの前に表示されることがあり、プログラムがそのイベントのために停止しているかどうかを示します。

dbx はキーワード inmember、inclass、および infunction キーワードで設定された複数の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、1 つのブレークポイント セットとして報告します。

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一覧表示した場合、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 (この例では 3 と 5) によって削除するには:

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

dbx に現在ロードさ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:

```
(dbx) delete all
```

詳細については、349 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各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。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 を `handler` コマンドの引数として使用し、ブレークポイントを有効または無効にできます。例:

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

詳細については、289 ページの「イベントハンドラ」を参照してください。

効率性に関する考慮事項

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。

`inclass` など数百のブレークポイントになる可能性のある複数のブレークポイントは、作成時にのみオーバーヘッドがあります。dbx は永続的ブレークポイントを使用しますが、それら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。

`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。dbx step_events 環境変数を使用して、各 step コマンドまたは next コマンドのあとにブレークポイントを取り出して再挿入するかどうかを制御します。

もっとも低速なイベントは、自動ステップ実行を使用するイベントです。このプロセスは、各ソース行をステップ実行する trace step コマンドと同様に、明示的で明確です。stop change や trace cond のようなその他のイベントは、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もあります。

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。例:

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

trace -in main は使用しないでください。trace は main によって呼び出された関数の中でも有効であるためです。lookup() 関数が変数を破壊していることが疑われる場合に、この修飾子を使用します。

♦ ♦ ♦ 第 7 章

7 呼び出しスタックの使用

この章では、dbx が*呼び出しスタック*を使用するしくみと、呼び出しスタックを操作するときに、where コマンド、hide コマンド、unhide コマンド、および pop コマンドを使用する方法について説明します。

マルチスレッドのプログラムにおいて、これらのコマンドは現在のスレッドの呼び出しスタックに対して作用します。現在のスレッドの変更方法の詳細については、414 ページの「thread コマンド」を参照してください。

この章には次のセクションが含まれています。

呼び出しスタックは、呼び出されたが、それぞれの呼び出し元にまだ戻されていない、現在アクティブなすべてのルーチンを表します。スタックフレームは、単一の関数で使用するために割り当てられる呼び出しスタックのセクションです。

呼び出しスタックは上位メモリー (上位アドレス) から下位メモリーに拡大するため、*up* は呼び出し元 (最終的には main() またはスレッドの開始関数) のフレームに向かい、*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() のような呼び出し元ルーチンは上位メモリーに位置します。

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生成したプログラムの状態を知る場合にも役立ちます。これが発生した場合、そのコアファイルを dbx にロードすることができます (36 ページの「既存のコアファイルのデバッグ」を参照)。

詳細については、434 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

スタックの上下の移動は、「スタックの移動」と呼ばれます。スタックを上下に移動して、関数を参照する場合、dbx は現在の関数とソース行を表示します。開始する位置、*ホーム*は、プログラムが実行を停止したポイントです。ホームからスタックを上下に移動するには、up コマンド、down コマンド、または frame コマンドを使用します。

dbx コマンドの up および down はともに、スタックを現在のフレームから上下に移動するフレームの数を dbxに指示する *number* 引数を受け付けます。*number* を指定しない場合、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* レベル上に (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、428 ページの「up コマンド」を参照してください。

スタックの下方向への移動

呼び出しスタックを *number* レベル下に (現在の停止ポイントに向かって) 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、353 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 コマンドで表示された番号で指定したフレームに直接移動するために使用します。

```
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、コマンドによって、その番号で指示されたフレームに直接移動できます。+ (プラス記号) または - (マイナス記号) を含めると、コマンドによって、1 レベルの増分で上 (+) または下 (-) に移動できます。プラスまたはマイナス記号と *number* を含めると、指定した数のレベルだけ上または下に移動できます。- h オプションは、非表示のフレームもカウントに含めます。

pop コマンドを使用して、特定のフレームに移動することもできます。

呼び出しスタックのポップ

呼び出しスタックから停止関数を削除し、呼び出し中の関数を新しい停止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、大域変数または静的変数、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除きます。

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:

```
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、387 ページの「pop コマンド」を参照してください。

スタックフレームの非表示

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一覧表示します。

正規表現に一致するすべてのスタックフレームを非表示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に一致し、ファイルの照合に sh または ksh 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 コマンドを使用します。

```
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す。このような関数で停止すると、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、現在のスコープをそれに設定します。この停止関数は矢印記号 (=>) で表されます。

次の例では、main() が -g オプションを付けてコンパイルされているため、引数の値に加えてシンボリック名も表示されます。main() によって呼び出されたライブラリ関数は、-g を付けてコンパイルされていないため、関数のシンボリック名は表示されますが、SPARC入力レジスタ $i0 から $i5 の 16 進値の内容は引数として表示されます。

次の例では、プログラムがセグメント例外により停止しています。原因は、SPARC 入力レジスタ $i0 内の strlen() への null 引数と考えられます。

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

CD Library Statistics:

 Titles:         1

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

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

```
(dbx)
```

スタックトレースの例については、31 ページの「呼び出しスタックを確認する」および231 ページの「呼び出しのトレース」を参照してください。

♦ ♦ ♦ 第 8 章

8 データの評価と表示

この章では、データの評価とデータの表示の 2 つのタイプのデータのチェックについて説明します。

この章には次のセクションが含まれています。

変数と式の評価

このセクションは、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 がどの変数を評価しているか確かでないときは、which コマンドを使用して dbx が使用している完全修飾名を確認します。

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、436 ページの「which コマンド」と 436 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を評価 (モニター) したい場合は、次のいずれかを行います。

- 関数の名前を特定します。70 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。例:

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

- 現在の関数を変更することにより、関数を表示します。65 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。

ネイティブコード内の変数または式を評価するには print コマンドを使用します。

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、388 ページの「print コマンド」を参照してください。

注記 - 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場合、dbx は、コンパイラで提供された特定の実行時型識別関数を呼び出します。ソースで明示的に演算子を使用していない場合、これらの関数はコンパイラで生成されていない場合があり、dbx は式の評価に失敗します。

C++ ポインタの出力

C++ で、オブジェクトポインタには、*静的な型* (ソースコードに定義される) と *動的な型* (キャストが行われる前のオブジェクト) の 2 つの型があります。dbx は、オブジェクトの動的な型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 コマンド、display コマンド、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。dbx はクラスによって直接定義されたすべてのデータメンバーと、基底クラスから継承されたものを表示します。

これらのコマンドは dbxenv 変数 output_dynamic_type のデフォルトの動作を切り替える -d または +d オプションもとります。

プロセスが実行されていないときに、-d フラグを使用するか、または dbxenv 変数 output_dynamic_type を on に設定すると、program is not active エラーメッセージが生成されます。コアファイルのデバッグ中に、プロセスがないと動的情報にアクセスすることが不可能なためです。仮想継承から動的な型の検索を試みると、「illegal cast on class pointers」という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++ で無名の引数を持つ関数を定義できます。例:

```
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名前を取得するには、関数名をそのターゲットとして、whatis コマンドを使用します。

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

詳細については、429 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:

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

ポインタの間接参照

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ため、dbx は、コマンドペインに評価を表示します。この例では、t の指す値です。

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

式のモニタリング

プログラムが停止するたびに式の値をモニターすることは、特定の式または変数がいつどのように変化するかを知る効果的な方法です。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モニターするように dbx に指示します。モニタリングは、undisplay コマンドによって停止するまで続けられます。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。

display コマンドは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ために使用します。

display *expression*, ...

一度に複数の変数をモニターできます。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、351 ページの「display コマンド」を参照してください。

watch コマンドは、すべての停止ポイントで式の値を監視するために使用します。

watch *expression*, ...

詳細については、429 ページの「watch コマンド」を参照してください。

表示の停止 (非表示)

dbx は、undisplay コマンドで表示が停止されるまで、モニターしている変数の値の表示を続行します。指定した式の表示を停止したり、現在モニターしているすべての式の表示を停止したりすることができます。

特定の変数または式の表示を停止するには:

undisplay *expression*

現在モニターされているすべての変数の表示を停止するには:

undisplay 0

詳細については、424 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

assign コマンドは変数に値を割り当てるために使用します。

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

配列の評価

配列の評価は、ほかの変数の型を評価する方法と同じです。

次の例は、サンプルの Fortran 配列です。

```
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。例:

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

dbx print コマンドを使用して、大きな配列の一部を評価することができます。配列の評価に含まれるもの:

- 配列の断面化 – 多次元配列から任意の矩形、つまり *n* 次元ボックスを出力します。

- 配列の刻み – 指定した断面 (配列全体のこともあります) 内の特定の要素のみを固定パターンで出力します。

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

配列の断面化は、C、C++、Fortran の `print`、`display`、および `watch` コマンドでサポートされています。

C と C++ での配列の断面化の構文

配列の各次元で、配列を断面化するための `print` コマンドの完全な構文は次のようになります。

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

ここでは:

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

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。

例:

```
(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

配列の各次元で、配列を断面化するための print コマンドの完全な構文は次のようになります。

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

ここでは:

array-expression	配列型に評価される式
first-expression	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
last-expression	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
stride-expression	刻み幅。デフォルトは 1 です。

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。

例:

```
(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:

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

3 行目を出力するには:

```
(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)
```

4 列目を出力するには:

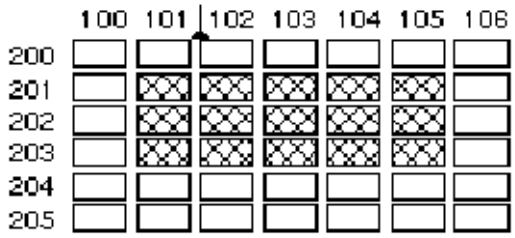
```
(dbx) print a(1:3,4:4)
'ShoSli'MAIN'a(1:3, 1:4) =
        (1,4)   14
        (2,4)   24
        (3,4)   34
(dbx)
```

断面の使用

次の例は、C++ 配列の 2 次元の矩形の断面で、デフォルトの刻み値の 1 が省略されています。

```
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。



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。次の組の式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。

刻みの使用

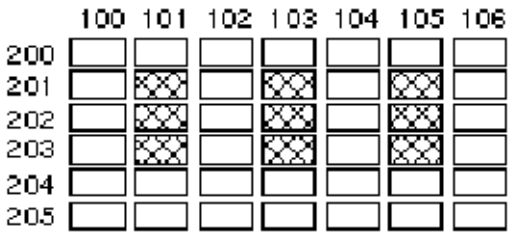
print コマンドに、配列の断面全体を*刻む*ように指示すると、dbx は断面に含まれる特定の要素だけを評価し、それぞれの間の一定の数の要素をスキップします。

配列の断面化の構文の 3 番目の式 *stride-expression* は、刻み幅の長さを指定します。*stride-expression* の値は出力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。

次の例は、前の断面の例で使用した同じ配列です。今回は、print コマンドに、第 2 次元の断面に 2 の刻み幅を含めます。

```
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。



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。次の例に、短縮形の構文の使用方法を示します。

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* の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
```

次のコマンドは、2 次元配列の第 2 次元の要素を 2 つおきに出力します。

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

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、式の値を独自に表示することができます。`dbx` は呼び出しベースの pretty-print と、Python で書かれた pretty-print フィルタの 2 つの pretty-print のメカニズムをサポートしています。古い呼び出しベースのメカニズムは、デバッグ対象内で定義されている、特定のパターンに準拠した関数を呼び出すことによって機能します。

- 131 ページの「呼び出しベースの pretty-print」
- 134 ページの「Python pretty-print フィルタ (Oracle Solaris)」

`dbx` は `dbxenv` 変数 `output_pretty_print_mode` で使用するメカニズムを決定します。`call` に設定されている場合、呼び出しベースのプリティプリンタだけが検索されます。`filter` に設定されている場合、Python ベースのプリティプリンタが検索されます。`filter_unless_call` に設定されている場合、呼び出しベースのプリティプリンタがフィルタより優先されます。

`print` コマンド、`rprint` コマンド、`display` コマンド、または `watch` コマンドに `-p` オプションを指定した場合、タイプに関係なく、プリティプリンタが呼び出されます。プリティプリンタの呼び出しの詳細については、131 ページの「pretty-print の呼び出し」を参照してください。

dbxenv 変数 output_pretty_print が on に設定されている場合、print コマンド、rprint コマンド、または display コマンドにデフォルトとして -p が渡されます。この動作をオーバーライドするには、+p を使用します。さらに、output_pretty_print は、IDE 局所変数、バルーン評価、およびウォッチの pretty-print を制御します。

pretty-print の呼び出し

pretty-print 関数は次のような場合に起動されます。

- print -p または dbxenv 変数 output_pretty_print が on に設定されている場合。
- display -p または dbxenv 変数 output_pretty_print が on に設定されている場合。
- watch -p または dbxenv 変数 output_pretty_print が on に設定されている場合。
- dbxenv 変数 output_pretty_print が on に設定されている場合のバルーン評価。
- dbxenv 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目的としているため、スクリプトは予測可能であるべきです。
- dump コマンド。dump は、where コマンドと同じ簡略化フォーマットを使用します。これは後のリリースで 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。

呼び出しベースの pretty-print

呼び出しベースの pretty-print を使用すると、アプリケーションで関数呼び出しによって、式の値を独自に表示できます。print コマンド、rprint コマンド、display コマンド、または watch コマンドに -p オプションを指定すると、dbx は const chars *db_pretty_print (const T *, int flags, const char *fmt) 形式の関数を検索して呼び出し、print または display の戻り値を置換します。

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す。

pretty-print するときは、次の情報も考慮してください。

- 133 ページの「可能性のある障害」
- 132 ページの「pretty-print 関数に関する考慮事項」
- dbx 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、そのセマンティクスの大半は 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、局所変数、およびバルーン評価で pretty-print を使用できます。
  - 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。
  - エラー回復の向上。
- 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。
- dbxenv 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に戻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。

pretty-print 関数に関する考慮事項

pretty-print 関数を使用する場合、次のことを考慮する必要があります。

- 一定/揮発性の非限定型の場合、通常は db_pretty_print(int *, ...() および db_pretty_print(const int *, ...)() などの関数は別個のものとみなされます。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 - 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。
  - 非強制 - `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。
- `db_pretty_print()` 関数は `-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` がパラメータシグニチャーにアクセスする必要があるためです。
- `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。
- `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。

可能性のある障害

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。

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、これらの障害はサイレントで、`dbx` は標準フォーマットに戻ります。ただし、`output_pretty_print_fallback` `dbxenv` 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。

`dbxenv` 変数 `output_pretty_print` を `on` に設定しないで、`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す。次に、`pop -c` コマンドを使用すると、呼び出しをクリーンアップすることができます。

Python pretty-print フィルタ (Oracle Solaris)

pretty-print フィルタ機能を使用すると、ある形式の値を別の形式に変換できるフィルタを Python で作成できます。Python ベースのプリティプリンタは Oracle Solaris でのみ使用できます。

注記 - Python pretty-print フィルタは C および C++ コードでのみ使用でき、Fortran では使用できません。

フィルタは、C++ 標準テンプレートライブラリの 4 つの実装の選択したクラスに組み込まれています。次の表に、ライブラリ名とそのライブラリのコンパイラオプションを示します。

ライブラリのコンパイラオプション	ライブラリ名
`-library=Cstd` (デフォルト)	`libCstd.so.1`
`-library=stlport4`	`libstlport.so.1`
`-library=stdcxx4`	`libstdcxx4.so.4.**`
`-library=stdcpp` (`-std=c++11` オプションを使用した場合のデフォルト)	`libstdc++.so.6.*`

次の表に、C++ 標準テンプレートライブラリで使用できるクラスフィルタと、インデックスおよび断面を出力できるかどうかを示します。

クラス	インデックスと断面が使用可能
文字列	いいえ
vector	はい
list	はい
set	いいえ

例 8-1 フィルタ付きの pretty-print

次の出力は、`dbx` の `print` コマンドを使用したリストの出力の例です。

```
(dbx) print list10
list10 = {
   __buffer_size = 32U
   __buffer_list = {
```

```
    __data_ = 0x654a8
  }
  __free_list   = (nil)
  __next_avail  = 0x67334
  __last     = 0x67448
  __node     = 0x48830
  __length   = 10U
  }
```

次は、dbx で出力された同じリストですが、pretty-print フィルタを使用しています。

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

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

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

Oracle Solaris での Python の使用

Python pretty-print フィルタと python コマンドは Oracle Solaris でのみ使用できます。組み込みの Python インタプリタを起動するには、python と入力します。Python コードを評価するには、python *python-code* と入力します。初期の Python プラグイン API を使用できます。ただし、その主な目的は、コールバックとして呼び出されるプリティプリンタフィルタを作成することです。したがって、python コマンドは主にテストと診断の目的に使用されます。

Python pretty-print API のドキュメント

Python pretty-print API ドキュメントを生成するには、python-docs コマンドを使用します。このコマンドは Oracle Solaris でのみ使用できます。

♦ ♦ ♦ 第 9 章

9 実行時検査の使用

実行時検査 (RTC) を使用すると、開発段階でネイティブコードアプリケーション内にあるメモリーアクセスエラーやメモリーリークなどの実行時エラーを自動的に検出できます。また、メモリー使用をモニターすることもできます。

この章は次の各節から構成されています。

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。

注記 - Java コードでは、実行時検査を行うことはできません。

実行時検査では、次の機能が提供されます。

- メモリー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` フラグを指定してコンパイルすると、実行時検査エラーメッセージにソースの行番号の相互関係が表示されます。また、実行時検査では、`-O` の最適化フラグで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` オプションでコンパイルされていないプログラムについては、特別な考慮事項がいくつかあります。

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

一度に多数のエラーが表示されないようにするには、プログラムを構成する個々のモジュールを開発している、開発サイクルの早い段階で実行時検査を使用します。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。この方法により、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。

実行時検査の要件

RTC を使用するには、次の要件を満たす必要があります。

- `libc` を動的にリンクしている。
- 標準の `libc malloc`、`free`、および `realloc` 関数、またはこれらの関数に基づいたアロケータの使用。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。161 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。
- 完全にはストリップされていないプログラム。`strip -x` でストリップされたプログラムは受け入れ可能です。

実行時検査の制限については、164 ページの「実行時検査の制限」を参照してください。

実行時検査の使用

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の検査の有効化

メモリー使用状況とメモリーリークの検査を有効にするには、次のコマンドを使用します。

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

メモリー使用状況検査またはメモリーリーク検査が有効になっている場合、showblock コマンドは、指定されたアドレスにあるヒープブロックに関する詳細情報を表示します。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、399 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の有効化

メモリーアクセス検査のみを有効にするには、次のコマンドを使用します。

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

すべての実行時検査の有効化

メモリーリーク検査、メモリー使用状況検査、およびメモリーアクセス検査を有効にするには、次のコマンドを使用します。

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

詳細については、330 ページの「check コマンド」を参照してください。

実行時検査の無効化

実行時検査を完全に無効にするには、次のコマンドを使用します。

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

詳細については、423 ページの「uncheck コマンド」を参照してください。

プログラムの実行

必要な種類の実行時検査を有効にしたら、ブレークポイントを使用して、または使用しないでテスト対象の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、各メモリーアクセスが実行される直前にその妥当性がチェックされるため、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。dbxenv 変数 rtc_auto_continue が on に設定されていないかぎり、制御がユーザーに戻されます。

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。そのエラーが致命的エラーでない場合は、cont コマンドを使用してプログラムの実行を継続できます。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ところまで実行されます。詳細については、342 ページの「cont コマンド」を参照してください。

rtc_auto_continue dbxenv 変数が on に設定されている場合、実行時検査では自動的に、引き続きエラーを検索して実行を継続します。エラーは、dbxenv 変数 rtc_error_log_file_name で指定されたファイルにリダイレクトされます。デフォルトのログファイル名は、/tmp/dbx.errlog.*unique-ID* です。

suppress コマンドを使用して、実行時検査エラーの報告を制限できます。詳細については、411 ページの「suppress コマンド」を参照してください。

次の単純な例は、hello.c と呼ばれるプログラムに対するメモリーアクセス検査とメモリー使用状況検査を有効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
Reading ld.so.1
Reading librtc.so
Reading libc.so.1
Reading libdl.so.1

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

 Total      Num of  Leaked     Allocation call stack
```

```
 Size       Blocks  Block
                    Address
==========  ====== ==========  ======================================
        32       1    0x21aa8  memory_leak < main

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

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

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

execution completed, exit code is 0
```

関数 access_error() は、初期化される前の変数 j を読み取ります。実行時検査では、このアクセスエラーを「非初期化領域からの読み取り」(rui) エラーとして報告します。

関数 memory_leak() は、戻る前に変数 local を解放しません。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。

このプログラムは、常にスコープ内にある大域変数 hello1 と hello2 を使用しています。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、解放の各操作をモニターすることによって、プログラムが正しくメモリーにアクセスしているかどうかをチェックします。

プログラムが誤ってメモリーの読み書きを行う状況にはさまざまなものがあります。これらはメモリーアクセスエラーと呼ばれます。たとえば、プログラムは、ヒープブロックに対する free() の呼び出しを使用して解放されたメモリーブロックを参照することがあります。または、関数が局所変数へのポインタを返すことがあり、そのポインタにアクセスするとエラーが発生します。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。ある種類のメモリーアクセスエラーは、見つけるのが非常に困難です。

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。可能性のあるメモリーの状態は次のとおりです。

- **割り当てられていない、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。
- **割り当てられているが、初期化されていない**。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。
- **割り当てられており、かつ初期化済み**。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。そのため、チェーン内の最初のエラーは「最初の原因」であり、そのエラーを修正すると以降のエラーの一部も修正される可能性があります。

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

実行時検査では、メモリーアクセスエラーに関する次の情報が提供されます。

`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

`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。

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

メモリーアクセスエラー

実行時検査では、次のメモリーアクセスエラーを検出します。

- `rui` – 170 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` – 170 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
- `rob` – 170 ページの「配列範囲外からの読み込み (rob) エラー」を参照
- `wua` – 171 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` – 171 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
- `wob` – 170 ページの「配列範囲外メモリーへの書き込み (wob) エラー」を参照
- `mar` – 169 ページの「境界整列を誤った読み取り (mar) エラー」を参照
- `maw` – 169 ページの「境界整列を誤った書き込み (maw) エラー」を参照
- `duf` – 168 ページの「重複解放 (duf) エラー」を参照
- `baf` – 168 ページの「不正解放 (baf) エラー」を参照
- `maf` – 168 ページの「境界整列を誤った解放 (maf) エラー」を参照
- `oom` – 169 ページの「メモリー不足 (oom) エラー」を参照

注記 - SPARC プラットフォームでは、実行時検査では配列境界のチェックを実行しないため、配列境界違反をアクセスエラーとして報告しません。

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、ブロックを指しているポインタがないため、プログラムはそれらのブロックを参照できず、ましてや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; /* no free of s. Once foo returns, there is no     */
            /* pointer pointing to the malloc'ed block,         */
            /* so that block is leaked.                         */
}
```

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

```
}
```

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。プログラムがすぐに終了する場合にメモリーを解放しないことは一般的なプログラミング習慣であるため、この定義はほとんど役に立ちません。プログラムがブロックへのポインタを引き続き 1 つ以上保持している場合、実行時検査ではそのブロックをリークとして報告しません。

メモリーリーク検査の使用

実行時検査では、次のメモリーリークエラーを検出します。

- `mel` – 172 ページの「メモリーリーク (mel) エラー」を参照
- `air` – 172 ページの「レジスタ中のアドレス (air)」を参照
- `aib` – 171 ページの「ブロック中のアドレス (aib)」を参照

注記 - 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。

起こり得るリーク

実行時検査では、「起こり得る」リークを 2 つのケースで報告できます。最初のケースは、ブロックの先頭を指しているポインタは見つからないが、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。このケースは、「ブロック中のアドレス」(`aib`) エラーとして報告されます。ブロックの内部を指している浮遊ポインタが実際のメモリーリークになります。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、このケースはメモリーリークになりません。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。

2 つ目のタイプの起こり得るリークは、データ領域内にはブロックへのポインタが見つからないが、レジスタ内にポインタが見つかった場合に発生します。このケースは、「レジスタ中のアドレス」(`air`) エラーとして報告されます。レジスタが誤ってブロックを指している場合や、それが以前に失われたメモリーポインタの古いコピーである場合、これは実際のリークです。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が、この場合はメモリーリークではありません。そのため、プログラムが最適化されており、そのレポートが showleaks コマンドの結果であった場合、それは実際のリークでない可能性があります。リークでない可能性があります。詳細については、400 ページの「showleaks コマンド」を参照してください。

注記 - RTC リーク検査では、標準の libc malloc/free/realloc 関数またはこれらの関数に基づいたアロケータを使用する必要があります。その他のアロケータについては、161 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有効になっている場合、メモリーリークのためのスキャンは、テスト対象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。このプログラムを、kill コマンドで強制終了してはいけません。次の例は、メモリーリークエラーの標準的なメッセージです。

```
Memory leak (mel):
Found leaked block of size 6 at address 0x21718
At time of allocation, the call stack was:
    [1] foo() at line 63 in test.c
    [2] main() at line 47 in test.c
```

UNIX プログラムには、そのプログラムの最上位のユーザー関数である main 手続き (f77 では MAIN と呼ばれます) が含まれています。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。後者の場合は、戻ったあとに main にローカルなすべての変数がスコープから外れるため、大域変数がこれらの同じブロックを指していないかぎり、それらの変数が指していたヒープブロックはすべてリークとして報告されます。

プログラムは終了しようとしており、exit() を呼び出すことなく main から復帰するため、一般的なプログラミング習慣では、main で局所変数に割り当てられたヒープブロックを解放しません。これらの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 内の最後の実行可能なソース行にブレークポイントを設定することによって、main から復帰する直前でプログラムを停止します。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、400 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

リーク検査が有効になっている場合は、プログラムが終了したときに自動リークレポートを受信します。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。レポート内の詳細レベルは、dbxenv 変数 rtc_mel_at_exit によって制御されます。デフォルトでは、簡易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。

サイズ	リークのあるブロックのサイズ
場所	リークのあるブロックが割り当てられていた場所
アドレス	リークのあるブロックのアドレス
スタック	check -frames によって制約される、割り当て時の呼び出しスタック

次に、対応する簡易メモリーリークレポートを示します。

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

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

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

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

次の例は、標準的な詳細リークレポートを示しています。

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

Memory Leak (mel):
Found 2 leaked blocks with total size 1852 bytes
```

```
At time of each allocation, the call stack was:
    [1] true_leak() at line 220 in "leaks.c"
    [2] true_leak() at line 224 in "leaks.c"

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

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

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

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できます。このコマンドは、最後の `showleaks` コマンド以降の新しいメモリーリークを報告します。詳細については、400 ページの「`showleaks` コマンド」を参照してください。

リークの結合

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。リークを結合するか、またはリークを個別に報告するかの決定は、`check -leaks` の `-match` *m* オプション、または `showleaks` コマンドの `-m` オプションで指定される `number-of-frames-to-match` パラメータによって制御されます。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`

ブロック 1	ブロック 2	ブロック 3
[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 より大きい場合、実行時検査では、すべてのリークを個別のリークとして報告します。malloc は、リークレポートには示されません。

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

メモリーリークレポートを取得したら、メモリーリークを修正するための次のガイドラインに従ってください。

- 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な場合は簡単ですが、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。デフォルトでは、showleaks コマンドは、最後の showleaks コマンド以降に作成された新しいリークのみを一覧表示します。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、400 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を使用すると、使用中のすべてのヒープメモリーを表示できます。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。この情報はまた、プログラムの動的なメモリー消費の削減にも役立つほか、パフォーマンスチューニングにも役立つ可能性があります。

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。メモリー使用状況の情報はまた、メモリー使用状況を表示させる showmemuse コマンドを使用して、プログラムの実行中にいつでも取得できます。詳細については、401 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有効にすると、リーク検査も有効になります。プログラム終了時のリークレポートに加えて、「使用中のブロック」(biu) レポートも取得されます。デフォルトでは、使用中のブロックの簡易レポートがプログラム終了時に生成されます。メモリー使用状況レポート内の詳細レベルは、dbxenv 変数 rtc_biu_at_exit によって制御されます。

次の例は、標準的なメモリー使用状況の簡易レポートを示しています。

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

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

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

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

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

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

Block in use (biu):
Found block of size 8 bytes at address 0x21978 (20.00% of total)
```

```
At time of allocation, the call stack was:
     [1] cyclic_leaks() at line 155 in "memuse.c"
     [2] main() at line 118 in "memuse.c"
The following is the corresponding verbose memory use report:
```

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。

エラーの抑制

実行時検査には、報告されるエラーの数や種類を制限するうえで高い柔軟性を提供する、強力なエラーの抑制機能が含まれています。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。

suppress コマンドを使用してエラーを抑制できます。

エラーの抑制は、unsuppress コマンドを使用して取り消すことができます。

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。

抑制のタイプ

このセクションでは、使用可能な抑制のタイプについて説明します。

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、オプションは次のとおりです。

グローバル	デフォルトであり、プログラム全体に適用されます。
ロードオブジェクト	ロードオブジェクト全体 (共有ライブラリなど) またはメインプログラムに適用されます。
ファイル	特定のファイル内のすべての関数に適用されます。
関数	特定の関数に適用されます。
行	特定のソース行に適用されます。
アドレス	あるアドレスにある特定の命令に適用されます。

最新エラーの抑止

デフォルトでは、同じエラーの報告が繰り返されないようにするために、実行時検査では最新のエラーを抑制します。この設定は、dbx 変数 rtc_auto_suppress によって制御されます。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。この設定は、たとえば、何回も実行されるループでエラーが発生した場合に同じエラー報告の複数のコピーを避けるために役立ちます。

エラー報告回数の制限

dbxenv 変数 rtc_error_limit を使用すると、報告されるエラーの数を制限できます。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、エラー上限が 5 に設定されている場合は、実行の最後に示されるリークレポートと、発行する各 showleaks コマンドの両方で、最大 5 つのアクセスエラーと 5 つのメモリーリークが表示されます。デフォルトは 1000 です。

エラーの抑制の例

次の例では、main.cc はファイル名、foo と bar は関数、a.out は実行可能ファイルの名前です。

割り当てが関数 foo で発生したメモリーリークを報告しません。

```
suppress mel in foo
```

libc.so.1 から割り当てられた使用中のブロックの報告を抑制します。

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

a.out 内のすべての関数での非初期化領域からの読み取りを抑制します。

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

ファイル main.cc での非割り当て領域からの読み取りを報告しません。

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

main.cc の行 10 での重複した解放を抑制します。

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

関数 bar 内のすべてのエラーの報告を抑制します。

```
suppress all in bar
```

詳細については、411 ページの「suppress コマンド」を参照してください。

デフォルトの抑制

実行時検査では、すべてのエラーを検出するために - g オプション (シンボリック) を使用し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 (主に rui エラー) の正確性を保証するために、シンボリック情報が必要な場合もあります。このため、シンボリック情報が使用できない場合、特定のエラー (a.out では rui、共有ライブラリでは rui、aib、および air) はデフォルトで抑制されます。この動作は、suppress コマンドと unsuppress コマンドの -d オプションを使用して変更できます。

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、426 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。段階的なアプローチをとることを検討してください。それには、suppress コマンドを使用して、報告されるエラーを管理可能な数に削減し、これらのエラーだけを修正します。そして、このサイクルを繰り返しします。これにより、繰り返すたびに、抑制するエラーを少なくすることができます。

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。通常検出されるもっとも一般的なエラーの種類は、rui、rua、および wua であり、通常はこの順序です。rui エラーは比較的軽度ですが、あとでより重大なエラーを引き起こす場合があります。これらのエラーが発生しても、プログラムはたいてい引き続き正しく機能します。rua エラーと wua エラーは、無効なメモリーアドレスへのアクセスまたはそれらのアドレスからのアクセスであり、また常にコーディングエラーを示すためより重大です。

まず rui と rua エラーを抑制し、発生する wua エラーをすべて修正したあと、rui エラーのみを抑制して、プログラムを再度実行します。発生する rua エラーをすべて修正したあと、エラーを抑制せずに、プログラムを再度実行します。すべての rui エラーを修正します。最後に、プログラムを最終的に実行して、エラーが残っていないことを確認します。

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

子プロセスで実行時検査を使用するには、dbxenv 変数 rtc_inherit が on に設定されている必要があります。デフォルトでは off になります

dbx は、親に対して実行時検査が有効になっていて、かつ dbxenv 変数 follow_fork_mode が child に設定されている場合、子プロセスの実行時検査をサポートします。

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。

次の例に示すように、実行時検査の制御下に置くことができるのは、常に 1 つのプロセスだけです。

```
% 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
Reading symbolic information for program1
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for librtc.so
Reading symbolic information for libc.so.1
Reading symbolic information for libdl.so.1
Reading symbolic information for libc_psr.so.1
(dbx) check -all
access checking - ON
memuse checking - ON
(dbx) dbxenv rtc_inherit on
(dbx) dbxenv follow_fork_mode child
(dbx) run
Running: program1
(process id 3885)
Enabling Error Checking... done
RTC reports first error in the parent, program1
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff110
    which is 104 bytes above the current stack pointer
Variable is 'parent_j'
Current function is main
  11       parent_i = parent_j;
(dbx) cont
dbx: warning: Fork occurred; error checking disabled in parent
detaching from process 3885
Attached to process 3886
Because follow_fork_mode is set to child, when the fork occurs error checking is switched from the
parent
to the child process
stopped in _fork at 0xef6b6040
0xef6b6040: _fork+0x0008:    bgeu    _fork+0x30
Current function is main
   13       child_pid = fork();
```

```
 parent: child's pid = 3886
 (dbx) cont
 child: In child
 Read from uninitialized (rui):
 Attempting to read 4 bytes at address 0xeffff108
     which is 96 bytes above the current stack pointer
```

RTC reports an error in the child

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

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

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

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

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

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

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

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

 execution completed, exit code is 0
```

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は rui を検出できないという例外を除き、接続されたプロセスに対して機能します。

Oracle Solaris を実行しているシステム上の接続されたプロセス

Oracle Solaris オペレーティングシステムを実行しているシステムでは、プロセスの開始時に、そのプロセスに rtcaudit.so がプリロードされている必要があります。接続しようとしている先のプロセスが 64 ビットプロセスである場合は、適切な 64 ビットの rtcaudit.so を使用します。これは次の場所にあります。

64 ビット SPARC プラットフォーム: /*install-dir*/lib/dbx/sparcv9/runtime/rtcaudit.so

AMD64 プラットフォーム: /*install-dir*/lib/dbx/amd64/runtime/rtcaudit.so

32 ビットプラットフォーム: /*install-dir*/lib/dbx/runtime/rtcaudit.so

rtcaudit.so をプリロードするには、次のように入力します。

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

必要な場合のみ、rtcaudit.so をプリロードするための LD_AUDIT 環境変数を設定します。これを常にロードされたままにしないでください。例:

```
% setenv LD_AUDIT...
% start-your-application
% 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。

接続する先のプログラムがほかの何らかのプログラムからフォークまたは実行されている場合は、フォークするメインプログラムに LD_AUDIT を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。32 ビットプログラムが 64 ビットプログラムをフォークまたは実行する場合、あるいは 64 ビットプログラムが 32 ビットプログラムをフォークまたは実行する場合には、この方法は機能しないことがあります。

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。Oracle Solaris OS の一部のバージョンは、LD_AUDIT_32 環境変数と LD_AUDIT_64 環境変数をサポートしています。これらは、それぞれ 32 ビットプログラムと 64 ビットプログラムにのみ影響を与えます。これらの変数がサポートされているかどうかを確認するには、実行している Oracle Solaris のバージョンの『*リンカーとライブラリガイド*』を参照してください。

Linux を実行しているシステム上の接続されたプロセス

Linux オペレーティングシステムを実行しているシステムでは、プロセスの開始時に、そのプロセスに librtc.so がプリロードされている必要があります。接続先のプロセスが AMD64 プロセッサで実行中の 64 ビットプロセスである場合、次の場所にある適切な 64 ビットの librtc.so を使用します。

64 ビット AMD64 プラットフォーム: /*install-dir*/lib/dbx/amd64/runtime/librtc.so

32 ビット AMD64 プラットフォーム: /*install-dir*/lib/dbx/runtime/librtc.so

librtc.so をプリロードするには、次のように入力します。

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

必要な場合のみ、librtc.so をプリロードするための LD_PRELOAD 環境変数を設定します。これを常にロードされたままにしないでください。例:

```
% setenv LD_PRELOAD...
% start-your-application
% unsetenv LD_PRELOAD
```

プロセスに接続したら、RTC を有効にすることができます。

接続する先のプログラムがほかの何らかのプログラムからフォークまたは実行されている場合は、フォークするメインプログラムに LD_PRELOAD を設定する必要があります。LD_PRELOAD 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。この解決方法は、32 ビットプログラムが 64 ビットプログラムをフォークまたは実行している場合や、64 ビットプログラムが 32 ビットプログラムをフォークまたは実行している場合は機能しない可能性があります。

環境変数 LC_PRELOAD 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。Linux の一部のバージョンは、LD_PRELOAD_32 環境変数と LD_PRELOAD_64 環境変数をサポートしています。これらは、それぞれ 32 ビットプログラムと 64 ビットプログラムにのみ影響を与えます。実行して

いる Linux 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

プログラミングエラーをすばやく特定して修正するために、実行時検査を `fix` および `cont` コマンドとともに使用できます。修正継続機能では、デバッグ時間を大幅に節約できる、修正と継続の強力な組み合わせが提供されます。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
Reading symbolic information for a.out
Reading symbolic information for rtld /usr/lib/ld.so.1
Reading symbolic information for librtc.so
Reading symbolic information for libc.so.1
Reading symbolic information for libintl.so.1
```

```
Reading symbolic information for libdl.so.1
Reading symbolic information for libw.so.1
(dbx) check -access
access checking - ON
(dbx) run
Running: a.out
(process id 15052)
Enabling Error Checking... done
Write to unallocated (wua):
Attempting to write 1 byte through NULL pointer
Current function is problem
    7       *s = 'c';
(dbx) pop
stopped in main at line 12 in file "bug.c"
   12       problem();
(dbx) #at this time we would edit the file; in this example just copy
the correct version
(dbx) cp bug-fixed.c bug.c
(dbx) fix
fixing "bug.c" ......
pc moved to "bug.c":14
stopped in main at line 14 in file "bug.c"
   14       problem();
(dbx) cont

execution completed, exit code is 0
(dbx) quit
The following modules in \Qa.out' have been changed (fixed):
bug.c
Remember to remake program.
```

修正継続機能の使用の詳細については、172 ページの「メモリーリーク (mel) 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。多くのアプリケーションは、`malloc()` または `free()` 関数をベースに、あるいはスタンドアロンで、独自のメモリー管理ルーチンを記述しています。独自のアロケータ (*専用アロケータ*と呼ばれます) を使用すると、実行時検査ではそれらを自動的に追跡できません。そのため、それらの誤った使用の結果としてのリークエラーやメモリーアクセスエラーを認識できません。

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 で提供され、Oracle Solaris 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。標準ヒープブロックを参照するメモリーアクセスエラーが発生した場合、エラーレポートには通常、ヒープブロック割り当ての位置が含まれます。専用アロケータがプログラムヒープを使用しない場合、エラーレポートには割り当て項目が含まれない場合があります。

実行時検査 API を使用して libumem 内のメモリーアロケータを追跡する必要はありません。RTC は libumem ヒープ管理ルーチンに割り込み、それらを対応する libc 関数にリダイレクトします。

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 の下でプログラムを実行し、デフォルトでは、実行時検査エラー出力をデフォルトファイル program.errs 内に格納します。

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。

注記 - 64 ビット Linux OS を実行しているシステムで bcheck ユーティリティーを実行するには、その前に環境変数 _DBX_EXEC_32 を設定する必要があります。

bcheck の構文

bcheck の構文は次のとおりです。

```
bcheck [-V] [-access | -all | -leaks | -memuse] [-xexec32] [-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。ファイル *script* に含まれている dbx コマンドを読み込むには、プログラムを実行する前に -s *script* オプションを使用します。*script* ファイルには通常、bcheck ユーティリティーのエラー出力を調整するための suppress や dbxenv などのコマンドが含まれています。

-q オプションは、bcheck を完全な静止ステータスにして、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のみを実行するには、次のように入力します。

```
bcheck hello
```

mach に対して、引数 5 を指定してアクセス検査のみを実行するには、次のように入力します。

```
bcheck -access mach 5
```

cc に対してメモリー使用状況検査を出力なしで実行し、通常の終了ステータスで終了するには、次のように入力します。

```
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、ブレー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 を使用して制御できます。

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

また、dbxenv 変数 rtc_auto_continue と rtc_error_log_file_name を設定することによって、dbx から直接バッチに似たモードを有効にすることもできます。

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。エラーは、dbxenv 変数 rtc_error_log_file_name で指定されたファイルにリダイレクトされます。デフォルトのログファイル名は、/tmp/dbx.errlog.*unique-ID* 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。

デフォルトでは、rtc_auto_continue は off に設定されています。

トラブルシューティングのヒント

プログラムに対してエラー検査が有効になり、そのプログラムが実行されたあと、次のいずれか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

このエラーは、接続されたプロセスで実行時検査を使用しており、LD_AUDIT を Oracle Solaris Studio dbx イメージに付属しているバージョン以外の rtcaudit.so に設定した場合に発生することがあります。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。164 ページの「実行時検査の制限」を参照してください。

実行時検査の制限

このセクションでは、実行時検査の制限について説明します。

シンボルやデバッグ情報が多いほどパフォーマンスが向上する

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。「非初期化領域からの読み取り」(rui) メモリーエラーは正しくないことがあるため、抑制されます。この抑止は unsuppress rui コマンドを使用してオーバーライドできます。ロードオブジェクト内のシンボルテーブルを保持するには、ロードオブジェクトのストリップ時に -x オプションを使用します。

RTC は、すべての配列範囲外エラーを検出できるわけではありません。静的メモリーおよびスタックメモリーに対する範囲検査は、デバッグ情報なしでは使用できません。

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すると、EINVAL エラーが発生するか、またはその試みが無視されます。

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 SEGSEGV が生成されます。このエラーが表示された場合は、rtc 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。

既存のすべてのコードから 8M バイト以内で十分なパッチ領域が使用可能な場合はパフォーマンスが向上する (SPARC プラットフォームのみ)。

既存のすべてのコードから 8M バイト以内で十分なパッチ領域が使用できない場合は、2 つの問題が発生する可能性があります。

- 遅延

  アクセス検査が有効になっている場合、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。この分岐命令には 8M バイトの範囲があります。デバッグ対象プログラムが、置き換えられている特定のロードまたはストア命令から 8M バイト以内のアドレス空間のすべてを使用していると、パッチ領域を配置するための場所は存在しません。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。トラップハンドラへの制御の移行は大幅に (最大 10 倍) 遅くなりますが、8M バイトの制限は存在しなくなります。

- V8+ モードでの出力レジスタのオーバーライドの問題

  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、デバッグ対象のプロセスが出力レジスタの上位 32 ビットを使用している場合は、アクセス検査が有効になっていると、そのプロセスが誤って実行される可能性があります。

コンパイラは、32 ビット SPARC ベースのバイナリの作成時にデフォルトでは V8+ アーキテクチャーを使用しますが、`-xarch` オプションを指定して、コンパイラに V8 アーキテクチャーを使用するよう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、トラップとともに計測されると正しく機能しないことがわかっている次の関数およびライブラリの計測機構を自動的にスキップします。

- `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上のいずれかの条件が当てはまり、かつアクセス検査を有効にしたときにプログラムが異なる動作を始める場合は、トラップハンドラの制限がそのプログラムに影響を与え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` コマンドを使用して、上に示されている関数およびライブラリを使用しているプログラム内のコードの計測機構をスキップします。一般に、問題を特定の関数まで追跡することは困難であるため、ロードオブジェクト全体の計測機構をスキップすることもできます。`rtc showmap` コマンドによって、アドレスでソートされた計測タイプのマップが表示されます。
- 32 ビット SPARC-V8 の代わりに 64 ビット SPARC-V9 を使用してみてください。

  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間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。詳細については、`rtc_patch_area`(1) のマニュアルページを参照してください。

  `dbx` は 8M バイトの制限に達すると、どのロードオブジェクトが大きすぎたかをユーザーに示し (メインプログラムまたは共有ライブラリ)、そのロードオブジェクトに必要な合計パッチ領域を表示します。

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 a.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 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ると、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。

  実行可能ファイルまたは大きなライブラリ内のオブジェクトファイルをより小さいオブジェクトファイルのグループに分割してから、それらをより小さい部分にリンクします。大きなファイルが実行可能ファイルである場合は、それをより小さい実行可能ファイルと一連の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。

  この方法により、dbx は、さまざまな共有オブジェクトの間にパッチコード用の領域を見つけることができます。

- 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。dbx は、実行時検査が有効になった状態で実行可能ファイルを起動する場合、共有ライブラリ間に余分な間隔を空けるよう実行時リンカーに依頼します。ただし、実行時検査が有効になった状態で dbx が起動していないプロセスに接続する場合は、各ライブラリが近づきすぎている可能性があります。

  実行時ライブラリが近づきすぎており、かつ dbx を使用してプログラムを起動できない場合は、rtc_patch_area スクリプトを使用して共有ライブラリを作成し、それをほかの共有ライブラリの間にあるプログラムにリンクしてみることができます。詳細については、rtc_patch_area(1) のマニュアルページを参照してください。

実行時検査エラー

実行時検査によって報告されるエラーは一般に、アクセスエラーとリークという 2 つのカテゴリに分類されます。

アクセスエラー

アクセス検査が有効になっている場合、実行時検査では、このセクションで説明されている種類のエラーを検出して報告します。

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: ヒープデータ以外のポインタを `free()` または `realloc()` に渡しています。

例:

```
char a[4];
char *b = &a[0];

free(b);                    /* Bad free (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);                    /* Duplicate free (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: 正しく境界合わせされていないポインタを `free()` または `realloc()` に渡しています。`malloc` によって返されたポインタを変更しています。

例:

```
char *ptr = (char *)malloc(4);
```

```
ptr++;
free(ptr);                    /* Misaligned free */
```

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;                    /* Misaligned read (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;                    /* Misaligned write (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);
/* Out of Memory (oom), ptr == NULL */
```

配列範囲外からの読み込み (rob) エラー

意味: 配列範囲外の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: 浮遊ポインタ、ヒープブロックの境界を越えています。

例:

```
char *cp = malloc (10);
char ch = cp[10];
```

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 *cp = malloc (10);
free (cp);
cp[0] = 0;
```

非初期化メモリーからの読み取り (rui) エラー

意味: 初期化されていないメモリー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;    /* Read from uninitialized memory (rui) */
}
```

配列範囲外メモリーへの書き込み (wob) エラー

意味: 配列範囲外のメモリーにデータを書き込もうとした。

考えられる原因: 浮遊ポインタ、またはヒープブロックの境界を越えています。

例:

```
char *cp = malloc (10);
cp[10] = 'a';
```

読み取り専用メモリーへの書き込み (wro) エラー

意味: 読み取り専用メモリー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;                /* Write to read-only memory (wro) */
}
```

非割り当てメモリーへの書き込み (wua) 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 *cp = malloc (10);
free (cp);
cp[0] = 0;
```

メモリーリークエラー

リーク検査が有効になっている場合、実行時検査では、次の種類のエラーを報告します。

ブロック中のアドレス (aib)

意味: メモリーリークの可能性がある。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++;    /* Address in Block */
}
```

レジスタ中のアドレス (air)

意味: メモリーリークの可能性がある。割り当てられたブロックが解放されておらず、そのブロックへの参照がプログラムメモリー内のどこにも存在しませんが、レジスタ内には参照が存在します。

考えられる原因: この状況は、コンパイラがプログラム変数をメモリー内には保持せず、レジスタ内にのみ保持している場合に正常な状態として発生することがあります。コンパイラは多くの場合、最適化が有効になっているときに、局所変数や関数パラメータに対してこの動作を実行します。最適化が有効になっていないときにこのエラーが発生した場合は、実際のメモリーリークである可能性があります。この状況は、割り当てられたブロックへの唯一のポインタが、そのブロックが解放される前にスコープから外れた場合に発生することがあります。

例:

```
if (i == 0) {
       char *ptr = (char *)malloc(4);
       /* ptr is going out of scope */
}
  /* Memory Leak or Address in Register */
```

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ません。

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;
/* Memory leak (mel) */
```

♦ ♦ ♦ 第 10 章

10 修正継続機能

fix コマンドを使用すると、デバッグプロセスを停止することなく、編集されたネイティブソースコードをすばやく再コンパイルできます。

この章には次のセクションが含まれています。

修正継続機能の使用

修正継続機能を使用すると、プログラム全体を再作成することなく、ネイティブソースファイルを変更して再コンパイルし、実行を継続することができます。.o ファイルを更新してプログラムに組み込むことにより、次に示すように、再リンクする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。

注記 - fix コマンドの次の制限に注意してください。

- 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。
- 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 コマンドは、Linux プラットフォームでは使用できません。

---

fix と cont の働き

fix コマンドを使用する前に、ソースを編集する必要があります。変更を保存したあと、fix コマンドを発行します。fix コマンドについては、358 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、スタックの一番上にある関数が修正されている場合は、新しい停止した関数が、新しい関数内の同じ行の先頭になります。さらに、古いファイルのブレークポイントがすべて新しいファイルに移動します。

修正継続機能はデバッグ情報あり、またはなしのどちらでコンパイルされたファイルに対しても使用できますが、最初にデバッグ情報なしでコンパイルされたファイルの場合、fix コマンドと cont コマンドの機能にはいくつかの制限があります。fix Commandの 358 ページの「fix コマンド」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) ファイルは、修正は可能ですが、特殊なモードで開く必要があります。dlopen 関数の呼び出しで RTLD_NOW|RTLD_GLOBAL または RTLD_LAZY|RTLD_GLOBAL のどちらかを使用できます。

Oracle Solaris 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を最小限に抑えるには、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。

これらのいずれかの変更を行う場合は、修正継続機能を使用するのではなく、プログラム全体を再作成してください。

プログラムの修正

`fix` コマンドを使用すると、変更を行なったあとに、プログラム全体を再コンパイルしなくてもソースファイルを再リンクできます。引き続きプログラムの実行を続けることができます。

ファイルの修正

まず、ソースへの変更を保存します。次に、`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を行わず、`.o` ファイルとメモリーイメージのみを変更します。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 が呼び出されると、コンパイル行を実行する前に、コンパイルの時点で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ます。コンパイル時からデバッグ時までの間にファイルシステムの構造が変更されると、正しいディレクトリの検索が困難になることがあります。この

問題を回避するには、あるパス名から別のパス名へのマッピングを作成する pathmap コマンドを使用します。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。

修正後の継続

cont コマンドを使用して実行を継続できます。プログラムの実行を再開する前に、このセクションで説明されている、変更の効果を決定する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内で変更を行なった場合、それらの変更は、プログラムを再度実行するか、またはその関数が次回呼び出されるまで効果がありません。

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

まだ呼び出されていない関数内で変更を行なった場合、それらの変更は、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対して変更を行なった場合、fix コマンドの影響は、その変更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。現在の関数をスタックからポップし、変更された関数が呼び出された位置から続行することによってそのコードを実行します。その関数に元に戻せない副作用 (ファイルを開く操作など) が含まれているかどうかを判断できるほど十分にコードを理解している必要があります。
- 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

停止した関数にではなく、現在スタック上にある関数に対して変更を行なった場合、変更されたコードは、その関数の現在の呼び出しには使用されません。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。

この問題は、いくつかの方法で解決できます。

- pop コマンドを使用して、変更されたすべての関数がスタックから削除されるまでスタックをポップします。コードを実行して問題が発生しないか確認する。
- cont at コマンドを使用して、別の行から続行します。
- 続行する前に、assign コマンドを使用してデータ構造を手動で修復します。
- run コマンドを使用してプログラムを再び実行する。

スタック上の変更された関数内にブレークポイントがある場合、それらのブレークポイントは新しいバージョンの関数に移動されます。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 コマンドや fix コマンドでは元に戻されません。大域変数に正しい値を手動で再割り当てするには、assign コマンドを使用します。

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
Running: testfix
(process id 4842)
signal SEGV (no mapping at the fault address) in copy at line 6 in file “testfix.cc”
    6        while ((*to++ = *from++) != ’\0’);
```

行 14 を 0 の代わりに buf に対して copy を実行するように変更し、ファイルを保存したあと、fix を実行します。

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

プログラムがここから続行された場合は、スタック上で依然として 0 ポインタがプッシュされるため、引き続きセグメント例外が発生します。pop コマンドを使用して、スタックの 1 フレームをポップします。

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

プログラムがここから続行された場合、プログラムは実行されますが、大域変数 from がすでに 1 増分されているため正しい値を出力しません。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てから、cont コマンドを使用します。それにより、プログラムは正しい文字列を出力します。

```
(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、ヘッダー (.h) ファイルの変更が必要になることがあります。変更されたヘッダーファイルが、それ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確実にアクセスされるようにするには、そのヘッダー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++ テンプレート定義を直接修正することはできません。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オーバーライドするには、`-f` オプションを使用します。

♦ ♦ ♦ 第 11 章

11 マルチスレッドアプリケーションのデバッグ

dbx は、Oracle Solaris スレッドまたは POSIX スレッドのどちらか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 を使用すると、各スレッドのスタックトレースを調べたり、すべてのスレッドを再開したり、特定のスレッドに対して step または next を実行したり、スレッド間を移動したりすることができます。

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。

マルチスレッドデバッグについて

dbx は、libthread.so が利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を認識します。プログラムは、-lthread または -mt でコンパイルされることによって明示的に、あるいは -lpthread でコンパイルされることによって暗黙的に libthread.so を使用します。

マルチスレッドプログラムを検出すると、dbx は、/usr/lib にあるスレッドデバッグのための特殊なシステムライブラリである libthread_db.so をロードしようとします。

dbx は同期的に動作するため、いずれかのスレッドまたは軽量プロセス (LWP) が停止すると、その他のスレッドおよび LWP もすべて同様に停止します。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。

注記 - マルチスレッドプログラミングと LWP については、Oracle Solaris の*マルチスレッドプログラミングに関するガイド*を参照してください。

スレッド情報

dbx では、次の例に示されているスレッド情報を使用できます。

```
(dbx) threads
    t@1 a l@1  ?()  running   in main()
    t@2      ?() asleep on 0xef751450  in_swtch()
    t@3 b l@2  ?()  running in sigwait()
    t@4     consumer()  asleep on 0x22bb0 in _lwp_sema_wait()
  *>t@5 b l@4 consumer()  breakpoint     in Queue_dequeue()
    t@6 b l@5 producer()     running       in _thread_start()
(dbx)
```

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す。
- > (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* または a l@*number* は、そのスレッドが指定された LWP にバインドされているか、またはその LWP 上でアクティブであることを示します。つまり、そのスレッドは実際にオペレーティングシステムが実行できます。
- 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。
- スレッド状態。
- 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*、dbx 形式のスレッド ID
- スレッド状態
- 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

スレッドの状態と LWP の状態

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	切り離されたスレッドが終了すると (thr_exit))、そのスレッドは、thr_join() を使用して再度参加させられるまでゾンビ状態にあります。THR_DETACHED は、スレッドの作成時 (thr_create()) に指定されたフラグです。非結合のスレッドは、再実行されるまでゾンビ状態です。
syncobj 上でスリープ中	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。libthread と libthread_db によって提供されるサポートレベルに応じて、*syncobj* は単なる 16 進数のアドレスになったり、より詳細な情報になったりします。
アクティブ	スレッドは LWP 上でアクティブですが、dbx がその LWP にアクセスできません。
不明	dbx では状態を判定できません。
lwpstate	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	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 *num*	所定のシステムコール番号の入口で LWP が停止しました。
シスコール *num* 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。
ジョブ制御	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。
ステップ動作	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。
フォルト *num*	LWP に所定の障害番号が発生しました。
シグナル *name*	LWP に所定のシグナルが発生しました。
プロセス同期	この LWP が所属するプロセスの実行が開始しました。
LWP 終了中	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
```

詳細については、414 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]]
```

既知のすべてのスレッドのリストを出力するには、次のように入力します。

```
threads
```

通常は出力されないスレッド (ゾンビ) を出力するには、次のように入力します。

```
threads -all
```

スレッドリストについては、184 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、416 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。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、-resumeone オプションを指定して call コマンドを使用することにより、1 つのスレッドを再開できます。

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たあと、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() を呼び出すときに、dbx は呼び出しの間、すべてのスレッドを再開します。これは、next コマンドでシングルステップを実行しているときの 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。

イベントの再入
第 1 イベント BPT(VID 6、TID 6、PC echo+0x8)
第 2 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。

このような場合、条件式で呼び出された関数が相互排他ロックを取得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 に強制的に 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、次の例のような場合には役立ちません。

- 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、thr_create イベントと thr_exit イベントを使用して、アプリケーションがスレッドを作成および破棄する頻度を調べることができます。

```
(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。

より広範囲な情報を得るために、別のセッションで次の例を試してみることもできます。

```
(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
```

この出力には、3 つのすべてのスレッドがスレッド t@1 によって作成されたことが示されています。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
Current function is main
216       stat = (int) thr_create(NULL, 0, consumer, q, tflags, &tid_cons2);
```

```
(dbx)
```

アプリケーションがスレッド t@1 ではなく、スレッド t@5 から新しいスレッドを生成することがある場合は、そのイベントを次のように取得できます。

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

LWP 情報について

通常、LWP を意識する必要はありません。ただし、スレッドレベルの問い合わせを完了できない場合があります。これらの場合は、LWP コマンドを使用して LWP に関する情報を表示します。

```
(dbx) lwps
    l@1 running in main()
    l@2 running in sigwait()
    l@3 running in _lwp_sema_wait()
  *>l@4 breakpoint in Queue_dequeue()
    l@5 running in _thread_start()
(dbx)
```

LWP 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 を示します。
- l@*number* は、特定の LWP を示します。
- LWP の状態。
- この LWP が現在実行している関数の名前。

現在の LWP を一覧表示または変更するには、lwp コマンドを使用します。

♦ ♦ ♦ 第 12 章

12 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) 関数を使用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。
この章には次のセクションが含まれています。

単純な接続の方法

次のいずれかの方法で、実行中の子プロセスに接続できます。

- dbx の起動時:

  ```
  $ dbx program-name process-ID
  ```

- dbx コマンド行からは次のように入力します。

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

プログラム名ではなく、- (マイナス記号) を含めると、dbx は指定されたプロセス ID に関連付けられている実行可能ファイルを自動的に見つけます。- 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。

Oracle Solaris Studio IDE で、実行中の子プロセスに接続することもできます。詳細については、IDE および dbxtool のオンラインヘルプ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 exec(2) 関数またはそのいずれかのバリアントを使用して新しいプログラムを実行すると、そのプロセス ID は変わりませんが、プロセスイメージが変更されます。dbx は exec() 関数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ロードします。

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 vfork(2)、fork1(2)、または fork(2) 関数を呼び出すと、そのプロセス ID は変わりますが、プロセスイメージは同じままです。dbx の動作は dbxenv 変数 follow_fork_mode の設定方法によって異なります。

parent	従来の動作です。dbx は fork を無視し、親プロセスを追跡します。
child	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	このモードは、Oracle Solaris Studio IDE または dbxtool から dbx を使用する場合にのみ使用できます。
ask	dbx が fork を検出するたびに、parent、child、both、または stop to investigate を選択するように求められます。stop を選択した場合、プログラムの状態を調査してから、cont を入力して続行できます。再度、続行方法を選択するように求められます。both は Oracle Solaris Studio IDE と dbxtool でのみサポートされています。

イベントとの対話

exec() または fork() プロセスでは、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。dbxenv 変数 follow_fork_inherit を on に設定して、フォークされたプロセスの削除をオーバーライドするか、-perm eventspec 修飾子を使用してイベントを永続的にすることがで

きます。イベント指定修飾子の使用方法の詳細については、288 ページの「cont at コマンド」を参照してください。

♦ ♦ ♦ 第 13 章

13 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、複数のコンピュータベンダーと共同で開発された、共有メモリー型マルチプロセッサアーキテクチャー用の移植性のある並列プログラミングモデルです。dbx で Fortran、C++、および C の OpenMP プログラムをデバッグするためのサポートは、dbx の一般的なマルチスレッドデバッグ機能に基づいています。

この章には次のセクションが含まれています。

Oracle Solaris Studio Fortran および C コンパイラによって実装される、OpenMP バージョン 4.0 アプリケーションプログラムインタフェースを構成するディレクティブ、実行時ライブラリルーチン、および環境変数については、『Oracle Solaris Studio 12.4: OpenMP API ユーザーズガイド 』を参照してください。

コンパイラによる OpenMP コードの変換

OpenMP のデバッグをより適切に説明するには、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    program example
2        integer i, n
3        parameter (n = 1000000)
4        real sum, a(n)
5
6        do i = 1, n
7        a(i) = i*i
8        end do
9
10        sum = 0
11
12    !$OMP PARALLEL DO DEFAULT(PRIVATE), SHARED(a, sum)
13
```

```
14        do i = 1, n
15        sum = sum + a(i)
16        end do
17
18    !$OMP END PARALLEL DO
19
20        print*, sum
21        end program example
```

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 `_$d1A12.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をデバッグするための通常の機能に加えて、`dbx` には、OpenMP プログラムをデバッグするための機能が用意されています。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。

並列領域へのシングルステップ

`dbx` は、並列領域にシングルステップ実行できます。並列領域は OpenMP 実行時ライブラリからアウトライン化されて呼び出されるため、実行のシングルステップでは、実際には、この目的で作成されたスレッドによって実行される何層にもわたる実行時ライブラリの呼び出しが実行されます。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、195 ページの「コンパイラによる OpenMP コードの変換」の Fortran コードで、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実行すると、スレーブ

スレッド t@2、t@3、および t@4 が作成され、実行時ライブラリの呼び出しを実行します。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。そのため、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、一般にシングルステップのあとも前と同じスレッド上にある通常のステップ実行とは異なります。

変数と式の出力

dbx は、すべての共有変数、非公開変数、およびスレッド非公開変数を出力できます。並列領域外で thread private 変数を出力しようとすると、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、並列構文内の shared 変数と private 変数のデータ共有属性を出力します。スレッド非公開変数については、並列構造内にあるかどうかには関係なく、そのデータ共有属性を出力します。例:

```
(dbx) whatis p_a
# OpenMP first and last private variable
int p_a;
```

print -s コマンドは、式 *expression* に非公開変数またはスレッド非公開変数が含まれている場合、現在の OpenMP 並列領域内のスレッドごとにその式の値を出力します。例:

```
(dbx) print -s p_a
thread t@3: p_a = 3
thread t@4: p_a = 3
```

式に private 変数または thread private 変数が含まれない場合は、1 つの値だけが出力されます。

領域およびスレッド情報の出力

現在の並列領域または指定された並列領域の説明を出力するには、omp_pr コマンドを使用します。これには、親領域、並列領域 ID、チームのサイズ (スレッドの数)、およびプログラムの場所 (プログラムカウンタアドレス) が含まれます。例:

```
(dbx) omp_pr
parallel region 127283434369843201
    team size = 4
    source location = test.c:103
    parent = 127283430568755201
```

また、現在の並列領域または指定された並列領域からそのルートまでのパスに沿ったすべての並列領域の説明を出力することもできます。例:

```
(dbx) omp_pr -ancestors
parallel region 127283434369843201
    team size = 4
    source location = test.c:103
    parent = 127283430568755201

    parallel region 127283430568755201
        team size = 4
        source location = test.c:95
        parent = <no parent>
```

また、並列領域ツリー全体も出力できます。例:

```
(dbx) omp_pr -tree
parallel region 127283430568755201
    team size = 4
    source location = test.c:95
    parent = <no parent>

    parallel region 127283434369843201
        team size = 4
        source location = test.c:103
        parent = 127283430568755201
```

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

現在のタスク領域または指定されたタスク領域の説明を出力するには、omp_tr コマンドを使用します。これには、タスク領域 ID、状態 (spawned、executing、waiting)、実行中のスレッド、プログラムの場所 (プログラムカウンタアドレス)、未完了の子、および親が含まれます。例:

```
(dbx) omp_tr
task region 65540
    type = implicit
    state = executing
    executing thread = t@4
    source location == test.c:46
    unfinished children = 0
    parent = <no parent>
```

また、現在のタスク領域または指定されたタスク領域からそのルートまでのパスに沿ったすべてのタスク領域の説明を出力することもできます。

```
(dbx) omp_tr -ancestors
task region 196611
    type = implicit
    state = executing
    executing thread = t@3
    source location - test.c:103
    unfinished children = 0
    parent = 131075

    task region 131075
```

```
        type = implicit
        state = executing
        executing thread = t@3
        unfinished children = 0
        parent = <no parent>
```

また、タスク領域ツリー全体も出力できます。例:

```
(dbx) omp_tr -tree
task region 10
    type = implicit
    state = executing
    executing thread = t@10
    source location = test.c:103
    unfinished children = 0
    parent = <no parent>
task region 7
    type = implicit
    state = executing
    executing thread = t@7
    source location = test.c:103
    unfinished children = 0
    parent = <no parent>
task region 6
    type implicit
    state = executing
    executing thread = t@6
    source location = test.c:103
    unfinished children = 0
    parent = <o parent>
task region 196609
    type = implicit
    state = executing
    executing thread = t@1
    source location = test.c:95
    unfinished children = 0
    parent = <no parent>

    task region 262145
        type = implicit
        state = executing
        executing thread = t@1
        source location = test.c:103
        unfinished children - 0
        parent = 196609
```

詳細については、385 ページの「omp_tr コマンド」を参照してください。

現在のループの説明を出力するには、omp_loop コマンドを使用します。これには、スケジューリング型 (静的、動的、ガイド付き、自動、または実行時)、順序付きかどうか、範囲、ステップ数または刻み幅、および繰り返し回数が含まれます。例:

```
(dbx) omp_loop
    ordered loop: no
    lower bound: 0
    upper bound: 3
    step: 1
    chunk: 1
    schedule type: static
    source location: test.c:49
```

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

現在のチームまたは指定された並列領域のチーム上のすべてのスレッドを出力するには、omp_team コマンドを使用します。例:

```
(dbx) omp_team
team members:
    0: t@1 state = in implicit barrier, task region = 262145
    1: t@6 state = in implicit barrier, task region = 6
    2: t@7 state = working, task region = 7
    3: t@10 state = in implicit barrier, task region = 10
```

詳細については、384 ページの「omp_team コマンド」を参照してください。

OpenMP コードをデバッグしている場合、thread -info は、現在のスレッドまたは指定されたスレッドに関する通常の情報に加えて、OpenMP スレッド ID、並列領域 ID、タスク領域 ID、および OpenMP スレッドの状態を出力します。詳細については、414 ページの「thread コマンド」を参照してください。

並列領域の実行の直列化

現在のスレッド、または現在のチーム内のすべてのスレッドについて、次に検出された並列領域の実行を直列化するには、omp_serialize コマンドを使用します。詳細については、384 ページの「omp_serialize コマンド」を参照してください。

スタックトレースの使用

並列領域内で実行が停止されると、where コマンドは、アウトラインサブルーチンを含むスタックトレースを表示します。

```
(dbx) where
current thread: t@4
```

```
=>[1] _$d1E48.main(), line 52 in "test.c"
  [2] _$p1I46.main(), line 48 in "test.c"

--- frames from parent thread ---
current thread: t@1
  [7] main(argc = 1, argv = 0xffffffff7fffec98), line 46 in "test.c"
```

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。コードが略述されているにもかかわらず、ソース行番号は依然として 15 にマップされます。

並列領域で実行が停止されたときに、関連フレームがアクティブ状態である場合、スレーブスレッドの `where` コマンドはマスタースレッドのスタックトレースを出力します。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。

また、まず `omp_team` コマンドを使用して現在のチーム内のすべてのスレッドを一覧表示し、次にマスタースレッド (OpenMP スレッド ID が `0` のスレッド) に切り替え、そのスレッドからスタックトレースを取得することによって、実行がスレーブスレッド内のブレークポイントにどのように到達したかを判定することもできます。

dump コマンドの使用

並列領域内で実行が停止されると、`dump` コマンドは、非公開変数の複数のコピーを出力する可能性があります。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` コマンドはスコープ内のすべての変数を出力するため、ホストしているルーチン内の `i` とアウトラインルーチン内の `i` の両方が表示されます。

イベントの使用

`dbx` では、OpenMP コード上で `stop`、`when`、および `trace` コマンドで使用できるイベントが提供されます。これらのコマンドとともにイベントを使用する方法については、292 ページの「イベント指定の設定」を参照してください。

同期イベント

`omp_barrier` [*type*] [*state*]

バリアーに入っているスレッドのイベントを追跡します。

type の有効な値は次のとおりです。

- `explicit` – 明示的なバリアーを追跡する
- `implicit` – 暗黙的なバリアーを追跡する

type を指定しなければ、明示的なバリアーだけが追跡されます。

state の有効な値は次のとおりです。

- `enter` – いずれかのスレッドがバリアーに入ったときにイベントを報告する
- `exit` – いずれかのスレッドがバリアーを出たときにイベントを報告する
- `all_entered` – すべてのスレッドがバリアーに入ったときにイベントを報告する

state を指定しない場合のデフォルトは `all_entered` です。

`enter` または `exit` を指定する場合は、そのスレッドのみの追跡を指定するためにスレッド ID を含めることができます。

`omp_taskwait` [*state*]

taskwait に入っているスレッドのイベントを追跡します。

state の有効な値は次のとおりです。

- `enter` – スレッドが taskwait に入ったときにイベントを報告する
- `exit` – すべての子タスクが完了したときにイベントを報告する

state を指定しない場合のデフォルトは `exit` です。

`omp_ordered` [*state*]

順序付き領域に入っているスレッドのイベントを追跡します。

state の有効な値は次のとおりです。

- `begin` – 順序付き領域が開始したときにイベントを報告する
- `enter` – スレッドが順序付き領域に入ったときにイベントを報告する
- `exit` – スレッドが順序付き領域を出たときにイベントを報告する

state を指定しない場合のデフォルトは `enter` です。

`omp_critical`

クリティカル領域に入っているスレッドのイベントを追跡します。

`omp_atomic` [*state*]

不可分領域に入っているスレッドのイベントを追跡します。

state の有効な値は次のとおりです。

- `begin` – 微細領域が開始したときにイベントを報告する
- `exit` – スレッドが微細領域を出たときにイベントを報告する

state を指定しない場合のデフォルトは `begin` です。

`omp_flush` フラッシュを実行しているスレッドのイベントを追跡します。

その他のイベント

`omp_task [`*state*`]` タスクの作成と終了を追跡します。
state の有効な値は次のとおりです。

- `create` – タスクが作成されてすぐ、その実行が開始される前にイベントを報告する
- `start` – タスクがその実行を開始したときにイベントを報告する
- `finish` – タスクがその実行を完了し、終了されようとしているときにイベントを報告する

state を指定しない場合のデフォルトは `start` です。

`omp_master` マスター領域に入るマスタースレッドのイベントを追跡します。

`omp_single` 単一領域に入っているスレッドのイベントを追跡します。

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でシングルステップを実行している場合は、実行シーケンスがソースコードのシーケンスとは同じでない可能性があります。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、オプティマイザが一般にコードを移動している最適化されたコード内でのシングルステップに似ています。

♦ ♦ ♦ 第 14 章

14 シグナルの操作

この章では、dbx を使用してシグナルを操作する方法について説明します。

この章には次のセクションがあります。

シグナルイベントについて

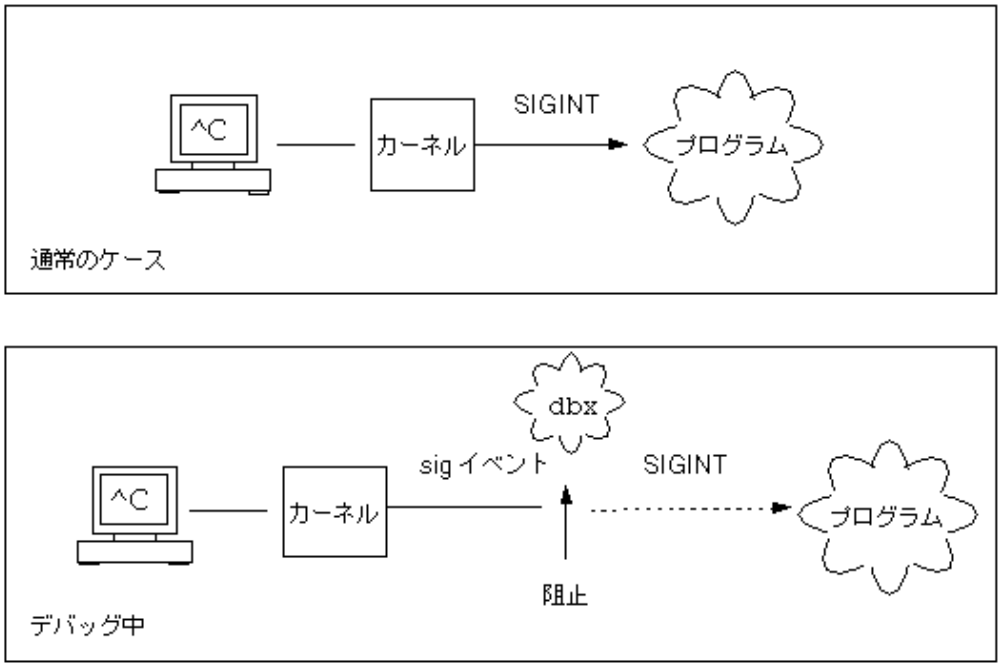
デバッグ中のプロセスにシグナルが送信されると、そのシグナルはカーネルによって dbx に送られます。通常、このことはプロンプトによって示されますが、そこでは次の 2 つの操作から 1 つを選択してください。

- プログラムが再開されたときにシグナルを取り消して (cont コマンドのデフォルトの動作です)、SIGINT (Ctrl + C) による割り込みと再開を容易にします。図14-1「SIGINT シグナルの阻止と取り消し」を参照してください。
- 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さ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*

ただし、そのシグナルは引き続きプロセスに転送されます。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 (364 ページの「ignore コマンド」を参照)。

シグナルの捕獲

dbx は、catch コマンドをサポートしています。このコマンドは、dbx が catch リストに登録されているシグナルのいずれかを検出したらプログラムを停止するよう dbx に指示します。

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の 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。

注記 - dbx が受け入れるシグナル名のリストには、dbx がサポートしているバージョンの Oracle Solaris オペレーティング環境によってサポートされるすべてのシグナル名が含まれています。そのため、dbx は、ユーザーが実行しているバージョンの Oracle Solaris オペレーティング環境ではサポートされないシグナルを受け入れる可能性があります。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 Oracle Solaris OS によってサポートされるシグナルのリストについては、`signal`(3head) のマニュアルページを参照してください。

現在トラップされているシグナルのリストを確認するには、*signal* 引数を指定せずに **catch** と入力します。

```
(dbx) catch
```

プログラムによって検出されたときに dbx によって現在*無視されているシグナル*のリストを確認するには、*signal* 引数を指定せずに **ignore** と入力します。

```
(dbx) ignore
```

デフォルトの catch リストと ignore リストを変更する

どのシグナルによってプログラムを停止させるかを制御するには、リスト間でシグナル名を移動します。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 シグナルのトラップ (Oracle Solaris のみ)

浮動小数点や整数の算術演算によって、オーバーフローや 0 による除算などの例外が発生する場合があります。このような例外は多くの場合、サイレントです。つまり、例外を発生させた操

作の結果としてシステムが (NaN などの) 妥当な答えを返します。そのため、これらの例外は dbx には認識されません。

この例外をサイレントではなく、トラップを発生させるように調整することができます。その場合、オペレーティングシステムがそのトラップを SIGFPE に変換してプロセスに配信すると、dbx はこのシグナル配信をインターセプトできます。次の点に注意してください。

- F77 は、デフォルトではどの浮動小数点例外でもトラップしません。
- F95 は、デフォルトでは無効なオペランド、0 による除算、およびオーバーフローの例外でトラップしますが、アンダーフローと不正確の例外ではトラップしません。
- C および C++ は、デフォルトでは浮動小数点例外ではトラップしません。
- 整数オーバーフローが SIGFPE を暗黙的にトリガーするプロビジョニングはありません。SPARC では、TVS (trap-on-overflow-set) アセンブリ命令を使用できます。SPARC または Intel では、類似の branch-on-overflow-set 命令を使用できます。

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
次のものを使用してトラップを有効にすることができます。

- fpsetmask – この関数は、トラップの有効化を厳密に制御します。fpsetmask(3C) のマニュアルページを参照してください。

  例:

  ```
  #include <ieeefp.h>
      int main() {
      fpsetmask(FP_X_INV|FP_X_OFL|FP_X_UFL|FP_X_DZ|FP_X_IMP);
      ...
  ```

- ieee_handler – Fortran には、psetmask(3c) の正確な類似関数はありません。代わりに、次のようにデフォルトの動作を確立することによって、トラップを有効にすることができます。

  例:

  ```
      integer*4 ieeer
      ieeeer  = ieee_handler('set', 'common', SIGFPE_DEFAULT)
  ```

  詳細については、ieee_environment(3f) および ieee_handler(3m) のマニュアルページを参照してください。

- -ftrap コンパイラフラグ – このタグは、fpsetmask()() と同様に、トラップの有効化を厳密に制御します。Fortran 95 の場合は、f95(1) のマニュアルページを参照してください。

前に説明した方法のいずれかを使用して浮動小数点トラップハンドラを有効にすると、ハードウェア浮動小数点ステータスレジスタ内のトラップ許可マスクが設定されます。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。

fpsetmask()() または ieee_handler()() の呼び出しを挿入するか、トラップハンドラを使用してプログラムをコンパイルしたら、そのプログラムを dbx にロードします。SIGFPE は、Oracle Solaris Studio 12.4 の時点ではデフォルトで捕獲されます。古いバージョンの dbx では、このシグナルが引き続き catch リスト内にあることを確認してください。

```
(dbx) catch FPE
```

catch FPE と同様の機能を持つ dbx の catch コマンドの代替コマンドを使用して fpsetmask() や ieee_handler() のパラメータを次のように調整することによって、どの特定の例外が表示されるかをさらに調整できます。

```
(dbx) stop sig FPE
(dbx) ignore SIGFPE #don't catch it twice
```

次のコードを使用すると、さらに詳細に制御できます。

```
stop sig FPE subcode
```

ここで、*subcode* には次のいずれかを指定できます。

FPE_INTDIV	0 による整数除算。
FPE_INTOVF	整数オーバーフロー。
FPE_FLTDIV	0 による浮動小数点除算。
FPE_FLTOVF	浮動小数点オーバーフロー。
FPE_FLTUND	浮動小数点アンダーフロー。
FOE_FLTRES	浮動小数点の結果不正確。
FPE_FLTINV	浮動小数点演算が無効です。
FPE_FLTSUB	添字が範囲外です。

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。次に、dbx の where コマン

ドを使用して呼び出しスタックをトレースすることにより、例外が発生したプログラムの特定の行番数を見つけることができます。

例外処理の原因追求

SPARC 上の例外の原因を特定するには、`regs -f` コマンドを使用して浮動小数点状態レジスタ (FSR) を表示します。このレジスタの発生した例外 (`aexc`) フィールドと現在の例外 (`cexc`) フィールドを確認します。これらのフィールドには、次の浮動小数点例外条件のビットが含まれています。

- 無効なオペランド
- オーバーフロー
- アンダーフロー
- 0 による除算
- 結果不正確

Intel では、浮動小数点ステータスレジスタは x87 の場合は `fstat`、SSE の場合は `mxcsr` です。

浮動小数点状態レジスタの詳細については、『*SPARC アーキテクチャーマニュアル 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。詳細な説明と例については、『Oracle Solaris Studio 12.4: 数値計算ガイド 』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` の `cont` コマンドは、`-sig` オプションをサポートしています。これを使用すると、システムシグナル *signal* を受信したかのようなプログラムの動作でプログラムの実行を再開できます。

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は、`^C` を入力してアプリケーションを停止し、`dbx` に制御を返すできます。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` コマンド、`next` コマンド、および `detach` コマンドも `-sig` オプションを受け入れます。

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 つのコマンドの効果は同じです。

```
(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 $signame;}
```

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
```

♦ ♦ ♦ 第 15 章

15 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。これらのタスクを実行するために使用するコマンドのサマリーとコード例も示します。dbx で C++ を正常にデバッグできますが、この章に説明する例外があります。

この章には次のセクションが含まれています。

- 213 ページの「C++ での dbx の使用」
- 214 ページの「dbx での例外処理」
- 219 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については、43 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 つの固有の側面を中心に説明しますが、dbx は、C++ プログラムをデバッグする場合にすべての機能を提供します。C++ プログラムでは次のタスクも実行できます。

注記 - 次のタスクはすべて以前の章で説明しています。

クラスと型の定義の詳細	77 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
継承されたデータメンバーの出力または表示	122 ページの「C++ ポインタの出力」を参照してください
オブジェクトポインタの動的情報	122 ページの「C++ ポインタの出力」を参照してください

仮想関数のデバッグ	93 ページの「関数の呼び出し」を参照してください
仮想関数のデバッグ	93 ページの「関数の呼び出し」を参照してください
実行時型情報の使用方法	122 ページの「変数、式または識別子の値を出力する」を参照してください
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	101 ページの「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」を参照してください
オーバーロード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	100 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
オーバーロードされたすべての非メンバー関数にブレークポイントを設定する	101 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」を参照してください
特定のオブジェクト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	102 ページの「オブジェクトにブレークポイントを設定する」を参照してください
オーバーロードされた関数またはデータメンバーの処理	99 ページの「関数へのブレークポイントの設定」を参照してください

この章の残りでは、C++ のデバッグの 2 つの固有の側面を中心に説明します。

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
- 予期されない例外を捕獲する
- 特定の例外を、スタックが解放される前に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

例外がスローされたポイントでの停止後、step コマンドを発行すると、スタックの解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。すべてのデスト

ラクタが実行されると、step コマンドによって、例外のスローを処理する捕獲ブロックに移動します。

例外処理コマンド

このセクションでは、例外を処理するための dbx コマンドについて説明します。

exception コマンド

exception コマンドの構文を次に示します。

```
exception [--d | -+d]
```

exception コマンドを使用して、デバッグ時にいつでも例外の型を表示します。オプションなしで exception コマンドを実行した場合に、表示される型は dbxenv 変数 output_dynamic_type の設定によって判断されます。

- 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 または +d オプションを指定すると、環境変数の設定がオーバーライドされます。

- -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、356 ページの「exception コマンド」を参照してください。

intercept コマンド

intercept コマンドの構文を次に示します。

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

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、指定した型のスローのみを阻止または除外するように設定します。

たとえば、int を除くすべての型を阻止するには:

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

Error 型の例外を阻止するには:

```
(dbx) intercept Error
```

次のコマンドで多すぎる CommonError 例外を阻止した後:

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

intercept コマンドを引数なしで入力すると、処理されていない例外と予期され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 が目的の例外のクラスではなく、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、365 ページの「intercept コマンド」を参照してください。

unintercept コマンド

unintercept コマンドの構文を次に示します。

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

- 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、425 ページの「unintercept コマンド」を参照してください。

whocatches コマンド

whocatches コマンドは、*typename* の例外が実行の現在のポイントでスローされた場合に、捕獲される場所を報告します。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す。捕獲ポイントがスロ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、「*type* is unhandled」というメッセージを表示します。

詳細については、437 ページの「whocatches コマンド」を参照してください。

例外処理の例

この例は、例外を含むサンプルプログラムを使用して、dbx での例外処理を示しています。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
(process id 304)
Stopped in bar at line 13 in file “foo.cc”
   13       c c1(3);
(dbx) whocatches int
int is caught at line 24, in function main (frame number 2)
(dbx) whocatches c
dbx: no runtime type info for class c (never thrown or caught)
(dbx) cont
Exception of type int is caught at line 24, in function main (frame number 4)
stopped in _exdbg_notify_of_throw at 0xef731494
0xef731494: _exdbg_notify_of_throw          :        jmp     %o7 + 0x8
Current function is bar
   14        throw(99);
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8         printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(3)
stopped in c::~c at line 9 in file "foo.cc"
    9       }
(dbx) step
stopped in c::~c at line 8 in file "foo.cc"
    8        printf("destructor for c(%d)\n", x);
(dbx) step
destructor for c(5)
stopped in c::~c at line 9 in file "foo.cc"
    9       )
(dbx) step
stopped in main at line 24 in file "foo.cc"
   24           printf("caught exception %d\n", i);
(dbx) step
caught exception 99
stopped in main at line 26 in file "foo.cc"
   26   }
```

注記 - このセクションで使用している例外は Oracle Solaris Studio コンパイラによって構築されています。この例は、gcc でコードをコンパイルした場合に異なることがあります。

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 にロードし、クラスや関数で使用するテンプレートで任意の dbx コマンドを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	222 ページの「stop inclass コマンド」、223 ページの「stop infunction コマンド」、および 223 ページの「stop in コマンド」を参照してください
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	221 ページの「whereis コマンド」を参照してください
テンプレートとインスタンスの定義を表示する	221 ページの「whatis コマンド」を参照してください
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	224 ページの「call コマンド」を参照してください
関数テンプレートのインスタンス化の値を出力する	224 ページの「print 式」を参照してください
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	224 ページの「list 式」を参照してください

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) です
- `Array<int>::getlength` はテンプレートクラスのメンバー関数です

- `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 コマンド

whereis コマンドは、関数またはクラステンプレートの関数またはクラスのインスタンス化のすべての発生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
member function: `Array<int>::Array(int)
member function: `Array<double>::Array(int)
class template instance: `Array<int>
class template instance: `Array<double>
class template: `a.out`template_doc_2.cc`Array
```

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
function template instance: `square<int>(__type_0,__type_0*)
function template instance: `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、436 ページの「whereis コマンド」を参照してください。

whatis コマンド

whatis コマンドは、関数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および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
To get the full template declaration, try `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
More than one identifier 'Array'.
Select one of the following:
 0) Cancel
 1) Array<int>::Array(int)
 2) Array<double>::Array(int>
> 1
Array<int>::Array(int 1);
```

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
More than one identifier 'square'.
Select one of the following:
 0) Cancel
 1) square<int(__type_0,__type_0*)
 2) square<double>(__type_0,__type_0*)
> 2
void square<double>(double num, double *result);
```

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、429 ページの「whatis コマンド」を参照してください。

stop inclass コマンド

テンプレートクラスのすべてのメンバー関数内で停止するには、次のように入力します。

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

stop inclass コマンドは、特定の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するために使用します。

```
(dbx) stop inclass Array<int>
(2) stop inclass Array<int>
```

詳細については、405 ページの「stop コマンド」および295 ページの「inclass イベント指定」を参照してください。

stop infunction コマンド

stop infunction コマンドは、指定した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するために使用します。

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

詳細については、405 ページの「stop コマンド」および294 ページの「infunction イベント指定」を参照してください。

stop in コマンド

stop in コマンドは、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するために使用します。

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、405 ページの「stop コマンド」および293 ページの「in イベント指定」を参照してください。

call コマンド

call コマンドは、スコープ内で停止した場合に、関数のインスタンス化または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ために使用します。dbx が正しいインスタンスを判断できない場合、ユーザーが選択できる番号付きの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、327 ページの「call コマンド」を参照してください。

print 式

print コマンドは、関数のインスタンス化またはクラステンプレートのメンバー関数を評価するために使用します。

```
(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、388 ページの「print コマンド」を参照してください。

list 式

list コマンドは、指定した関数のインスタンス化のソースリストを出力するために使用します。

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

詳細については、369 ページの「list コマンド」を参照してください。

♦ ♦ ♦ 第 16 章

16 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。

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 での Fortran OpenMP コードのデバッグについては、192 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、どのファイルがカレントであるかによって異なるブレークポイントを設定します。

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。以前のいくつかのバージョンのように、大文字と小文字が区別されるコマンドまたは大文字と小文字が区別されないコマンドを指定する必要はありません。

Fortran と dbx は、大文字と小文字が区別されるモードまたは大文字と小文字が区別されないモードのどちらか同じモードにする必要があります。

- 大文字と小文字が区別されないモードでは、-U オプションなしでコンパイルおよびデバッグします。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 という変数が含まれている場合、dbx では、print LAST コマンドと print last コマンドの両方が動作します。Fortran と dbx は、LAST と last を要求どおり同じであると見なします。
- 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 という変数が含まれている場合、dbx では、print last は動作しますが、print LAST は動作しません。Fortran と dbx は、LAST と last を要求どおり区別します。

注記 - dbx input_case_sensitive 環境変数を false に設定している場合でも、dbx では、ファイルまたはディレクトリ名は常に大文字と小文字が区別されます。

---

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。

デバッグのためのメインプログラム、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を実行する方法

1. **- g オプションでコンパイルおよびリンクします。**

   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。

   ■ **1 つの手順でコンパイルおよびリンクするには、次のように入力します。**

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

   ■ **個別の手順でコンパイルおよびリンクするには、次のように入力します。**

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

2. **my_program という名前の実行可能ファイルに対して dbx を起動します。**

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

3. **単純なブレークポイントを設定します。**

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
```

メインプログラム MAIN はすべて大文字である必要がありますが、サブルーチン、関数、またはブロックデータサブプログラムの名前は大文字と小文字のどちらでもかまいません。

4. **dbx の起動時に指定した実行可能ファイル内のプログラムを実行します。**

```
(dbx) run
Running: my_program
stopped in MAIN at line 3 in file "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. **値を出力します。**

n の値を出力します。

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

行列 twobytwo を出力するには、形式が変わる可能性があります。

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

array はここではなく、mkidentity でしか定義されていないため、行列 array は出力できないことに注意してください。

6. **次の行に実行を進めます。**

```
(dbx) next
stopped in MAIN at line 4 in file "a1.f"
    4             print *, determinant( twobytwo )
(dbx) print twobytwo
twobytwo =
    (1,1)       1.0
    (2,1)       0.0
    (1,2)       0.0
    (2,2)       1.0
(dbx) quit
demo%
```

next 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。

7. **dbx を終了します。**

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

セグメント例外のデバッグ

プログラムでセグメント例外 (SIGSEGV) が発生した場合、そのプログラムは、使用可能なメモリーの範囲外のメモリーアドレスを参照しています。

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

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見つけます。

プログラムを使用し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、dbx セグメント例外の行番号を見つけます。

```
demo% f95 -g -silent WhereSEGV.f
demo% a.out
Segmentation fault
demo% dbx a.out
Reading symbolic information for a.out
program terminated by signal SEGV (segmentation violation)
(dbx) run
Running: a.out
signal SEGV (no mapping at the fault address)
    in MAIN at line 4 in file "WhereSEGV.f"
    4                   a(j) = (i * 10)
(dbx)
```

例外の検出

プログラムが例外をスローする原因には多くのものが考えられます。問題を突き止めるための 1 つのアプローチとして、例外が発生したソースプログラム内の行番号を見つけたあと、その位置を調べます。

`-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
Reading symbolic information for wh
(dbx) catch FPE
(dbx) run
Running: wh
(process id 17970)
signal FPE (floating point divide by zero) in MAIN at line 2 in file “wh.f”
   2                    print *, r/s
(dbx)
```

呼び出しのトレース

プログラムがコアダンプで停止し、そこに至るまでの呼び出しのシーケンスを見つけることが必要になる場合があります。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、プログラムフロー内のどこで実行が停止し、実行がどのようにしてこの位置に到達したか、つまり呼び出されたルーチンの*スタックトレース*を表示します。

ShowTrace.f は、スタックトレースを表示するために、呼び出しシーケンス内の数レベル深い位置までコアダンプを取得するように記述されたプログラムです。

Note the reverse order:

```
demo% f77 -silent -g ShowTrace.f
demo% a.out
MAIN called calc, calc called calcb.
*** TERMINATING a.out
*** Received signal 11 (SIGSEGV)
Segmentation Fault (core dumped)
quil 174% dbx a.out
Execution stopped, line 23
Reading symbolic information for a.out
...
(dbx) run
calcB called from calc, line 9
Running: a.out
(process id 1089)
calc called from MAIN, line 3
signal SEGV (no mapping at the fault address) in calcb at line 23 in file "ShowTrace.f"
   23                   v(j) = (i * 10)
(dbx) where -V
=>[1] calcb(v = ARRAY , m = 2), line 23 in "ShowTrace.f"
  [2] calc(a = ARRAY , m = 2, d = 0), line 9 in "ShowTrace.f"
  [3] MAIN(), line 3 in "ShowTrace.f"
(dbx)
Show the sequence of calls, starting at where the execution stopped:
```

配列の操作

dbx は配列を認識し、これらの配列を出力することができます。

```
demo% dbx a.out
Reading symbolic information…
(dbx) list 1,25
    1           DIMENSION IARR(4,4)
    2           DO 90 I = 1,4
    3                   DO 20 J = 1,4
```

```
    4                           IARR(I,J) = (I*10) + J
    5   20              CONTINUE
    6   90      CONTINUE
    7           END
(dbx)  stop at 7
(1) stop at "Arraysdbx.f":7
(dbx) run
Running: a.out
stopped in MAIN at line 7 in file "Arraysdbx.f"
    7           END
(dbx) print IARR
iarr =
    (1,1) 11
    (2,1) 21
    (3,1) 31
    (4,1) 41
    (1,2) 12
    (2,2) 22
    (3,2) 32
    (4,2) 42
    (1,3) 13
    (2,3) 23
    (3,3) 33
    (4,3) 43
    (1,4) 14
    (2,4) 24
    (3,4) 34
    (4,4) 44
(dbx) print IARR(2,3)
    iarr(2, 3) = 23  - Order of user-specified subscripts ok
(dbx) quit
```

詳細については、127 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 割り当て可能配列

次の例は、dbx で割り当て可能配列への変更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
Running: a.out
(process id 10749)
 Size?
1000
stopped in main at line 6 in file "alloc.f95"
    6           ALLOCATE( buffer(n), STAT=status )
(dbx) whatis buffer
integer*4 , allocatable::buffer(:)
(dbx) next
continuing
stopped in main at line 7 in file "alloc.f95"
    7           IF ( status /= 0 ) STOP 'cannot allocate buffer'
(dbx) whatis buffer
integer*4 buffer(1:1000)
(dbx) cont
stopped in main at line 9 in file "alloc.f95"
    9           PRINT *, buffer(n)
(dbx) print n
buffer(1000) holds 1000
n = 1000
(dbx) print buffer(n)
buffer(n) = 1000
```

組み込み関数の表示

dbx は、Fortran 組み込み関数を認識します (SPARC プラットフォームおよび x86 プラットフォームのみ)。

dbx で組み込み関数を表示するには、次のように入力します。

```
demo% cat ShowIntrinsic.f
    INTEGER i
    i = -2
    END
(dbx) stop in MAIN
(2) stop in MAIN
(dbx) run
Running: shi
(process id 18019)
stopped in MAIN at line 2 in file "shi.f"
    2              i = -2
(dbx) whatis abs
Generic intrinsic function: "abs"
```

```
(dbx) print i
i = 0
(dbx) step
stopped in MAIN at line 3 in file "shi.f"
    3             end
(dbx) print i
i = -2
(dbx) print abs(1)
abs(i) = 2
(dbx)
```

複合式の表示

dbx は、Fortran 複合式も認識します。

dbx で複合式を表示するには、次のように入力します。

```
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
Running: a.out
(process id 10953)
stopped in MAIN at line 2 in file "ShowComplex.f"
    2       z = ( 2.0, 3.0 )
(dbx) whatis z
complex*8  z
(dbx) print z
z = (0.0,0.0)
(dbx) next
stopped in MAIN at line 3 in file "ShowComplex.f"
    3       END
(dbx) print z
z = (2.0,3.0)
(dbx) print z+(1.0,1.0)
z+(1,1) = (3.0,4.0)
(dbx) quit
demo%
```

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
Running: a.out
(process id 5217)
stopped in MAIN at line 2 in file "ShowInterval.f95"
    2      v = [ 37.1, 38.6 ]
(dbx) whatis v
INTERVAL*16  v
(dbx) print v
v = [0.0,0.0]
(dbx) next
stopped in MAIN at line 3 in file "ShowInterval.f95"
    3      END
(dbx) print v
v = [37.1,38.6]
(dbx) print v+[0.99,1.01]
v+[0.99,1.01] = [38.09,39.61]
(dbx) quit
demo%
```

論理演算子の表示

dbx は Fortran 論理演算子を見つけ、これらの論理演算子を出力することができます。

dbx で論理演算子を表示するには、次のように入力します。

```
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
Running: a.out
(process id 15394)
stopped in MAIN at line 5 in file "ShowLogical.f"
    5           z = .false.
(dbx) whatis y
logical*4 y
(dbx) print a .or. y
a.OR.y = true
(dbx) assign z = a .or. y
(dbx) print z
z = true
(dbx) quit
demo%
```

Fortran 派生型の表示

dbx では、構造体、Fortran 派生型を表示できます。

```
demo% f95 -g DebStruc.f95
demo% dbx a.out
(dbx) list 1,99
    1   PROGRAM Struct ! Debug a Structure
    2      TYPE product
    3         INTEGER        id
    4         CHARACTER*16   name
    5         CHARACTER*8    model
    6         REAL           cost
    7 REAL price
    8      END TYPE product
    9
   10      TYPE(product) :: prod1
   11
   12      prod1%id = 82
   13      prod1%name = "Coffee Cup"
   14      prod1%model = "XL"
   15      prod1%cost = 24.0
   16      prod1%price = 104.0
   17      WRITE ( *, * ) prod1%name
   18   END
(dbx) stop at 17
(2) stop at "Struct.f95":17
(dbx) run
Running: a.out
(process id 12326)
stopped in main at line 17 in file "Struct.f95"
   17      WRITE ( *, * ) prod1%name
(dbx) whatis prod1
product prod1
(dbx) whatis -t product
```

```
type product
    integer*4 id
    character*16 name
    character*8 model
    real*4 cost
    real*4 price
end type product
(dbx) n
(dbx) print prod1
    prod1 = (
    id    = 82
    name = 'Coffee Cup'
    model = 'XL'
    cost = 24.0
    price = 104.0
)
```

Fortran 派生型へのポインタ

dbx では、構造体、Fortran 派生型、およびポインタを表示できます。

```
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
Declare a derived type.
     2      TYPE product
     3         INTEGER        id
     4         CHARACTER*16   name
     5         CHARACTER*8    model
     6         REAL           cost
     7         REAL           price
     8      END TYPE product
     9
Declare prod1 and prod2 targets.
    10      TYPE(product), TARGET :: prod1, prod2
Declare curr and prior pointers.
    11      TYPE(product), POINTER :: curr, prior
    12
Make curr point to prod2.
    13      curr => prod2
Make prior point to prod1.
    14      prior => prod1
Initialize prior.
    15      prior%id = 82
    16      prior%name = "Coffee Cup"
    17      prior%model = "XL"
    18      prior%cost = 24.0
    19      prior%price = 104.0
```

```
Set curr to prior.
    20      curr = prior
Print name from curr and prior.
    21      WRITE ( *, * ) curr%name, " ", prior%name
    22   END PROGRAM DebStruPtr
 (dbx) stop at 21
 (1) stop at "DebStruc.f95":21
 (dbx) run
 Running: debstr
(process id 10972)
stopped in main at line 21 in file "DebStruc.f95"
   21      WRITE ( *, * ) curr%name, " ", prior%name
(dbx) print prod1
 prod1 = (
    id = 82
    name = "Coffee Cup"
    model = "XL"
    cost = 24.0
    price = 104.0
)
```

前の例で、dbx は、派生型のすべてのフィールド (フィールド名を含む) を表示しています。

構造体を使用して、Fortran 派生型の項目について照会できます。

```
Ask about the variable
(dbx) whatis prod1
 product prod1
Ask about the type (-t)
 (dbx) whatis -t product
 type product
    integer*4 id
    character*16 name
    character*8 model
    real cost
    real price
 end type product
```

ポインタを出力するには、次のように入力します。

```
dbx displays the contents of a pointer, which is an address. This address can be different with every run.
(dbx) print prior
 prior = (
     id    = 82
     name = 'Coffee Cup'
     model = 'XL'
     cost = 24.0
     price = 104.0
 )
```

オブジェクト指向 Fortran

dbx でサポートされているオブジェクト指向 Fortran 機能は、型拡張と多相ポインタで、C++ のサポートとの整合性があります。

dbxenv 変数 output_dynamic_type および output_inherited_members は、Fortran で機能します。

print および whatis コマンドで -r、+r、-d、および +d オプションを使用すると、オブジェクト指向 Fortran コード内の継承される (親) 型および動的型に関する情報を取得できます。

割り当て可能スカラー型

dbx は、Fortran の割り当て可能スカラー型をサポートしています。

♦ ♦ ♦ 第 17 章

17 dbx による Java アプリケーションのデバッグ

この章では、dbx を使用して、Java™ コードと C JNI (Java Native Interface) コードまたは C++ JNI コードが混在したアプリケーションをデバッグする方法について説明します。

この章は、次のセクションで構成されています。

dbx と Java コード

Oracle Solaris Studio dbx を使用すると、Oracle Solaris™ OS および Linux OS の下で実行されている混在コード (Java コードと C コードまたは C++ コード) をデバッグできます。

Java コードに対する dbx の機能

dbx では、いくつかの種類の Java アプリケーションをデバッグできます。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 コードをデバッグする場合、dbx には次の制限があります。

- 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

次の dbxenv 変数は、dbx での Java アプリケーションのデバッグに固有です。JAVASRCPATH、CLASSPATHX、および jvm_invocation 環境変数は、dbx を起動する前にシェルプロンプトで設定するか、または dbx コマンド行から設定できます。jdbx_mode 環境変数の設定は、アプリケーションのデバッグ中に変化します。その設定は、jon コマンドと joff コマンドを使用して変更できます。

jdbx_mode	jdbx_mode dbxenv 変数の設定は、java、jni、native のいずれかです。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、251 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: java。
JAVASRCPATH	JAVASRCPATH dbxenv 変数を使用すると、dbx で Java ソースファイルを検索するディレクトリを指定できます。この変数は、Java ソースファイルが .class または .jar ファイルと同じディレクトリ内に存在しない場合に役立ちます。詳細については、246 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 dbxenv 変数を使用すると、独自のクラスローダーによってロードされる Java クラスファイルのパスを dbx に指定できます。詳細については、247 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 dbxenv 変数を使用すると、JVM™ ソフトウェアが起動される方法をカスタマイズできます。(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、248 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 を使用すると、次の種類の Java アプリケーションをデバッグできます。

- .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

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
```

dbx を使用して、.class ファイル名拡張子を使用するファイルをデバッグできます。また、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。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、次のパス名が指定されると、dbx は pkg/Toy.class がメインクラス名であると判定し、/home/user/java をクラスパスに追加します。

```
(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。dbx を使用して、JAR ファイルをデバッグできます。.jar で終わるファイル名を持つファイルのデバッグを開始すると、dbx は、この JAR ファイルのマニフェストで指定された Main_Class 属性を使用してメインクラスを判定します。(メインクラスは、アプリケーションのエントリポイントである 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。

Main-Class 属性が含まれていない JAR ファイルをデバッグする場合は、次の例に示すように、Java 2 Platform, Standard Edition のクラス JarURLConnection で指定されている JAR URL 構文 jar:<url>!/{entry} を使用してメインクラスの名前を指定できます。

```
(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ているクラスパス (/myclass.class や /x/y/z.class など) をメインクラスとして処理します。
- JAR ファイルの名前 (./myjar.jar、/a/b/c/d/e.jar、または /a/b/c/d.jar) をクラスパスに追加します。
- 主クラスのデバッグを開始します。

注記 - 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 (248 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

Java アプリケーションには通常、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が使用されることを dbx に通知する必要があります。詳細については、248 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。

動作中の Java アプリケーションへの dbx の接続

アプリケーションを起動したときに次の例に示されているオプションを指定した場合は、dbx を実行中の Java アプリケーションに接続できます。アプリケーションを起動したあとは、実行中の Java プロセスのプロセス ID を指定して dbx コマンドを使用することによってデバッグを開始します。

```
$ java -agentlib:dbx_agent myclass.class
$ dbx - 2345
```

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 Oracle OS を実行しているシステム上の 32 ビットバージョンの JVM ソフトウェア: /*install-dir*/SUNWspro/lib/libdbx_agent.so を追加します。

- Oracle Solaris OS を実行している SPARC ベースのシステム上の 64 ビットバージョンの JVM ソフトウェア: `LD_LIBRARY_PATH` に `/`*install-dir*`/SUNWspro/lib/v9/libdbx_agent.so` を追加します。
- Linux OS を実行している x64 ベースのシステム上の 64 ビットバージョンの JVM ソフトウェア: `LD_LIBRARY_PATH` に `/`*install-dir*`/sunstudio12/lib/amd64/libdbx_agent.so` を追加します。

install-dir は、Oracle Solaris Studio がインストールされている場所です。

実行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 アプリケーションに 64 ビットのオブジェクトライブラリが必要な場合は、そのアプリケーションを起動するときに `-d64` オプションを含めます。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 -agentlib:dbx_agent
$ dbx - 2345
```

次のタスクは、プロセス ID を使用して `dbx` を特定の Java プロセスに接続する方法を説明しています。

▼ 実行中の Java プロセスに接続する

1. **前のセクションの説明に従って `LD_LIBRARY_PATH` に `libdbx_agent.so` を追加することによって、JVM™ ソフトウェアが `libdbx_agent.so` を確実に見つけることができるようにします。**
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 -agentlib:dbx_agent` *myclass*`.class`
3. **これで、プロセス ID を使用して `dbx` を起動することによってプロセスに接続できます。**
   `dbx -` *process-ID*

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。C アプリケーションまたは 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必要があります。

```
-agentlib:dbx_agent
```

JVM ソフトウェアが `libdbx_agent.so` を見つけることができるようにするには、Java アプリケーションを実行する前に `LD_LIBRARY_PATH` に適切なパスを追加する必要があります。244 ページの「動作中の Java アプリケーションへの dbx の接続」を参照してください。

install-dir は、Oracle Solaris 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、248 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

場合によっては、Java ソースファイルが `.class` または `.jar` ファイルと同じディレクトリ内に存在しないことがあります。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。次の例では、`dbx` が、指定されたディレクトリ内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すようにします。

```
JAVASRCPATH=.:/mydir/mysrc:/mydir/mylibsrc:/mydir/myutils
```

C/C++ ソースファイルの格納場所の指定

次の状況では、`dbx` が C ソースファイルまたは C++ ソースファイルを見つけることができない可能性があります。

- 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、385 ページの「`pathmap` コマンド」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 (`pathmap Command`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。このような状況では、dbx はクラスファイルを見つけることができません。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 はクラスファイルを見つけようとするときに、指定されたディレクトリ内を探します。

Java メソッドにブレークポイントを設定する

ネイティブアプリケーションとは異なり、Java アプリケーションには容易にアクセスできる名前のインデックスがありません。そのため、たとえば、単純にメソッド名を指定できません。

```
(dbx) stop in myMethod #This will not work
```

代わりに、メソッドへのフルパスを使用する必要があります。

```
(dbx) stop in com.any.library.MyClass.myMethod
```

例外は、MyClass の何らかのメソッドで停止した場合で、その場合は myMethod で十分です。

メソッドへのフルパスを含めないようにする 1 つの方法として、stop inmethod の使用があります。

```
(dbx) stop inmethod myMethod
```

ただし、このコマンドでは、myMethod という名前の複数のメソッドが停止する可能性があります。

ネイティブ (JNI) コードでブレークポイントを設定する

JNI C または C++ コードを含む共有ライブラリは JVM によって動的にロードされるため、それらのライブラリ内にブレークポイントを設定するには追加の手順がいくつか必要です。詳しく

は、110 ページの「動的にロードされたライブラリにブレークポイントを設定する」を参照してください。

JVM ソフトウェアの起動方法のカスタマイズ

特定のタスクを実行するには、dbx から JVM ソフトウェアの起動をカスタマイズすることが必要になる場合があります。カスタマイズが必要な一般的なタスクには次のものがあります。

JVM ソフトウェアの起動は、jvm_invocation 環境変数を使用してカスタマイズできます。jvm_invocation 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 -agentlib:dbx_agent=sync=process-ID
```

jvm_invocation 環境変数が定義されている場合、dbx は、その変数の値を使用して JVM ソフトウェアを起動します。

jvm_invocation 環境変数の定義に -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 -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 環境変数を適切なパス名に設定します。

```
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 -agentlib:dbx_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、これらの引数を使用して JVM ソフトウェアを起動するように jvm_invocation 環境変数を設定します。

```
jvm_invocation="java -Xdebug -Xms512 -Xmx1024 -Xcheck:jni"
```

この例では、dbx が次のように JVM ソフトウェアを起動するようにします。

```
java -agentlib:dbx_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、起動のために独自のラッパーを使用できます。アプリケーションが独自のラッパーを使用している場合は、次の例に示すように、jvm_invocation 環境変数を使用して、使用されるラッパーを指定できます。

```
jvm_invocation="/export/siva-a/forte4j/bin/forte4j.sh -J-Xdebug"
```

この例では、dbx が次のように JVM ソフトウェアを起動するようにします。

```
/export/siva-a/forte4j/bin/forte4j.sh - -agentlib:dbx_agent=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) はいくつかの環境変数を設定し、コマンド行オプションを受け入れます。

```
#!/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のためのいくつかのコマンド行オプションを受け入れます。この形式のラッパースクリプトの場合は、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 -J 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、いくつか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は受け入れません。

```
#!/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して dbx を起動することによって、ラッパースクリプト自体の内部から dbx を起動するようにスクリプトを変更します。

  ```
  #!/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。

- いくつかのコマンド行オプションを受け入れるように、スクリプトを次のように少し変更します。

  ```
  #!/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が必要なアプリケーションをデバッグする場合は、jvm_invocation 環境変数を設定するときに -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

dbx が Java モードまたは JNI (Java Native Interface) モードにある場合は、Java アプリケーション (JNI コードを含む) の状態を検査したり、コードの実行を制御したり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、または native) は、環境変数 jdbx_mode に格納されます。

Java モードでは、Java 構文を使用して dbx と対話します。dbx も Java 構文を使用し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、ネイティブのステータスばかりでなく、Java 関係のステータスも示されるた

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プログラムにのみ影響を与えるため、Java に関連したすべての機能が無効になります。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、自動的にはネイティブモードに切り替わりません。joff コマンドを使用して Java または JNI モードからネイティブモードに、また jon コマンドを使用してネイティブモードから Java モードに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

Java アプリケーションの実行を (たとえば、Ctrl + C キーを押すことによって) 中断すると、dbx はアプリケーションを安全な状態に置き、すべてのスレッドを中断することによって、自動的にモードを Java/JNI モードに設定しようとします。

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。その場合は、プログラムの状態を検査できるように、jon コマンドを使用して Java モードに切り替えることができます。

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。

- Java モードまたは JNI モードでネイティブモードの場合と同じ引数を受け入れ、同じように動作するコマンド。254 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」を参照してください。

- Java モードまたは JNI モードでのみ有効な引数や、ネイティブモードでのみ有効な引数を持つコマンド。255 ページの「Java モードで構文が異なる dbx コマンド」を参照してください。
- Java モードまたは JNI モードでのみ有効なコマンド。256 ページの「Java モードでのみ有効なコマンド」を参照してください。

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 コマンドでの Java の式の評価

ほとんどの `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。

- 修飾された `this` (<*ClassName*>`.this` など)
- クラスのインスタンス作成式
- 配列作成式
- 文字列連結演算子
- 条件演算子 ?:
- 複合代入演算子 (`x += 3` など)

Java アプリケーションの状態を調べるうえで特に有用なのは、IDE または `dbxtool` の監視機能を利用する方法です。

単なるデータの検査を超える操作を行う式では、正確な値の解釈に依存しないでください。

dbx コマンドによって使用され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。これらの情報により、dbx はユーザーがアプリケーションを実行する前に、ブレークポイントでより適切なエラー検査を実行できます。

一部の Java クラスとその属性には、クラスパス経由でアクセスできないことがあります。dbx は、これらのクラスを検査したり、ステップ実行したりできます。また、式解析プログラムは、これらのクラスが実行時にロードされたあと、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

次の表に示されている dbx コマンドは、Java モードでネイティブモードの場合と同じ構文を持ち、同じ動作を実行します。

コマンド	機能
attach	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。
cont	プロセスが実行を再開します。
dbxenv	dbxenv 変数を一覧表示または設定します。
delete	ブレークポイントとその他のイベントを削除します。
down	呼び出しスタックを下方向に移動します (main の逆方向)。
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。

コマンド	機能
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	ソースファイルの行を表示します。
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を検索するために、あるパス名を別のパス名にマップします。
proc	現在のプロセスのステータスを表示します。
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

次の表に示されている dbx コマンドは、Java のデバッグのためにネイティブコードのデバッグとは異なる構文を持ち、Java モードでの動作がネイティブモードの場合とは異なります。

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。

コマンド	ネイティブモードでの機能	Java モードでの機能
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	すべての停止点で式を評価して出力します。	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、局所変数、またはパラメータの値を出力します。
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。
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 コマンドを元に戻します。	display コマンドを元に戻します。
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

次の表に示されている dbx コマンドは、Java モードまたは JNI モードでのみ有効です。

コマンド	機能
java	dbx が 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。

コマンド	機能
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。
jpkgs	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。

♦ ♦ ♦ 第 18 章

18 機械命令レベルでのデバッグ

この章では、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する方法、指定されたアドレスにあるメモリーの内容を表示する方法、およびソース行を対応する機械命令とともに表示する方法について説明します。

この章には次のセクションが含まれています。

機械命令レベルでの dbx の使用

next コマンド、step コマンド、stop コマンド、および trace コマンドは、それぞれ対応する機械命令レベルのコマンドである nexti コマンド、stepi コマンド、stopi コマンド、および tracei コマンドをサポートしています。機械語レジスタの内容を出力するには regs コマンドを、また個々のレジスタの内容を出力するには print コマンドを使用します。

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。adb(1) (アセンブリ言語のデバッガ) から派生したコマンドを使用すると、次の項目について問い合わせることができます。

- *address* / = (等号) を使用。

- あるアドレスに格納されている *contents* - `/` (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。

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。デフォルトの *format* は、以前の `examine` コマンドで使用されたものと同じか、またはこれが指定された最初のコマンドである場合は `X` 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* までのメモリーの内容を *format* の書式で表示するには、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、アドレスを指定した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ている値を出力するには、次のように入力します。

```
examine +/ i
```

式の値を出力するには、その式をアドレスとして指定します。

```
examine address=format
examine address=
```

アドレスの使用

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。*address* は `+` (プラス記号) に置き換えることができます。これにより、次のアドレスの内容がデフォルト書式で表示されます。

次の例は有効なアドレスです。

`0xff00`	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
`&errno`	変数のアドレス
`str`	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。

書式の使用

format は、`dbx` が問い合わせの結果を表示するときのアドレス表示書式です。生成される出力は、現在の表示書式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` セッションの開始時に設定されるデフォルトの書式は `X` です。これにより、アドレスまたは値が 32 ビットワードとして 16 進数で表示されます。次のメモリー表示書式が有効です。

i	アセンブリ命令として表示
d	10 進数の 16 ビット (2 バイト) として表示
D	10 進数の 32 ビット (4 バイト) として表示
o	8 進数の 16 ビット (2 バイト) として表示
O	8 進数の 32 ビット (4 バイト) として表示
x	16 進数の 16 ビット (2 バイト) として表示
X	16 進数の 32 ビット (4 バイト) として表示 (デフォルト書式)
b	8 進数の 1 バイトとして表示
c	1 文字として表示
n	10 進数 (1 バイト) として表示。
w	1 つのワイド文字として表示
s	NULL バイトで終わる文字列として表示
W	ワイド文字列として表示

f	単精度浮動小数点数として表示
F、g	倍精度浮動小数点数として表示
E	倍精度浮動小数点数として表示
ld、lD	10 進数の 32 ビット (4 バイト) を表示 (D と同じ)
lo、lO	8 進数の 32 ビット (4 バイト) を表示 (O と同じ)
lx、LX	16 進数の 32 ビット (4 バイト) を表示 (X と同じ)
Ld、LD	10 進数の 64 ビット (8 バイト) を表示
Lo、LO	8 進数の 64 ビット (8 バイト) を表示
Lx、LX	16 進数の 64 ビット (8 バイト) を表示

カウントの使用

count は、10 進数の繰り返し回数です。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、アドレスと書式オプションを使用して、現在の停止点から始まる 5 つの連続した逆アセンブル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

dis コマンドは、examine コマンド (デフォルト表示書式を i として指定) と同じです。

dis コマンドの構文は次のとおりです。

dis [*address*] [*address1*, *address2*] [/*count*]

dis コマンドは、次のように動作します。

- 引数を指定しないと、+ から始まる 10 個の命令を表示します。
- *address* 引数のみを指定すると、*address* から始まる 10 個の命令を逆アセンブルします。
- *address* 引数と *count* を指定すると、*address* から始まる *count* 個の命令を逆アセンブルします。
- *address1* と *address2* 引数を指定すると、*address1* から *address2* までの命令を逆アセンブルします。
- *count* のみを指定すると、+ から始まる *count* 個の命令を表示します。

listi コマンドの使用

ソース行を対応するアセンブリ命令とともに表示するには、listi コマンドを使用します。これは、コマンド list -i と同等です。67 ページの「ソースリストの出力」にある list -i の説明を参照してください。

SPARC ベースのシステムの例:

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x0001083c: main+0x0014:  ld      [%fp + 0x48], %l0
0x00010840: main+0x0018:  add     %l0, 0x4, %l0
0x00010844: main+0x001c:  ld      [%l0], %l0
0x00010848: main+0x0020:  or      %l0, %g0, %o0
0x0001084c: main+0x0024:  call    0x000209e8 [unresolved PLT 7: atoi]
0x00010850: main+0x0028:  nop
0x00010854: main+0x002c:  or      %o0, %g0, %l0
0x00010858: main+0x0030:  st      %l0, [%fp - 0x8]
   14       j = foo(i);
0x0001085c: main+0x0034:  ld      [%fp - 0x8], %l0
0x00010860: main+0x0038:  or      %l0, %g0, %o0
0x00010864: main+0x003c:  call    foo
0x00010868: main+0x0040:  nop
0x0001086c: main+0x0044:  or      %o0, %g0, %l0
```

```
0x00010870: main+0x0048:  st      %l0, [%fp - 0xc]
```

x86 ベースのシステムの例:

```
(dbx) listi 13, 14
   13       i = atoi(argv[1]);
0x080488fd: main+0x000d:  movl   12(%ebp),%eax
0x08048900: main+0x0010:  movl   4(%eax),%eax
0x08048903: main+0x0013:  pushl  %eax
0x08048904: main+0x0014:  call   atoi <0x8048798>
0x08048909: main+0x0019:  addl   $4,%esp
0x0804890c: main+0x001c:  movl   %eax,-8(%ebp)
   14       j = foo(i);
0x0804890f: main+0x001f:  movl   -8(%ebp),%eax
0x08048912: main+0x0022:  pushl  %eax
0x08048913: main+0x0023:  call   foo <0x80488c0>
0x08048918: main+0x0028:  addl   $4,%esp
0x0804891b: main+0x002b:  movl   %eax,-12(%ebp)
```

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。

機械命令レベルでのシングルステップ

ある機械命令から次の機械命令へのシングルステップを実行するには、`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動作を行います。`nexti` コマンドは関数をステップ*オーバー*し、`stepi` コマンドは、次の命令から呼び出された関数にステップインして、呼び出された関数内の最初の命令で停止します。コマンドの書式も同じです。

`nexti` コマンドと `stepi` コマンドからの出力は、対応するソースレベルのコマンドとは次の 2 つの点で異なります。

- この出力には、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命令のアドレスが含まれます。
- デフォルトの出力には、ソースコード行の代わりに、逆アセンブルされた命令が含まれます。

例:

```
(dbx) func
hand::ungrasp
(dbx) nexti
ungrasp +0x18:  call support
(dbx)
```

詳細については、382 ページの「nexti コマンド」と 404 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、dbx がアドレスの実行またはトレースされている変数または式の値をチェックしている間、プログラムが各命令のあとに一瞬停止するようになります。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。

trace とそのイベント指定および修飾子の詳細については、108 ページの「トレースの実行」および 422 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。

tracei step	各命令をトレースします。
tracei next	各命令をトレースしますが、呼び出しを飛び越します。
tracei at *address*	指定のコードアドレスをトレース

詳細については、422 ページの「tracei コマンド」を参照してください。

SPARC の場合:

```
(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。このコマンドは、すべてのイベント指定を受け入れます。stopi コマンドの構文は次のとおりです。

stopi *event-specification* [*modifier*]

stopi コマンドの一般的に使用される形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、410 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、stopi コマンドを使用します。

(dbx) **stopi at** *address*

例:

```
(dbx) nexti
stopped in hand::ungrasp at 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出力できます。

regs コマンドの構文は次のとおりです。

```
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、393 ページの「regs コマンド」を参照してください。

SPARC ベースのシステムの例:

```
dbx[13] regs -F
current thread: t@1
current frame:  [1]
g0-g3    0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7    0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3    0x00000003 0x00000014 0xef7562b4 0xeffff420
o4-o7    0xef752f80 0x00000003 0xeffff3d8 0x000109b8
l0-l3    0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7    0xeffff438 0x00000001 0x00000007 0xef74df54
i0-i3    0x00000001 0xeffff4a4 0xeffff4ac 0x00020c00
i4-i7    0x00000001 0x00000000 0xeffff440 0x000108c4
y        0x00000000
psr      0x40400086
pc       0x000109c0:main+0x4    mov     0x5, %l0
npc      0x000109c4:main+0x8    st      %l0, [%fp - 0x8]
f0f1     +0.00000000000000e+00
f2f3     +0.00000000000000e+00
f4f5     +0.00000000000000e+00
f6f7     +0.00000000000000e+00
...
```

x64 ベースのシステムの例の場合:

```
(dbx) regs
current frame:  [1]
r15     0x0000000000000000
r14     0x0000000000000000
r13     0x0000000000000000
r12     0x0000000000000000
r11     0x0000000000401b58
r10     0x0000000000000000
r9      0x0000000000401c30
r8      0x0000000000416cf0
rdi     0x0000000000416cf0
rsi     0x0000000000401c18
rbp     0xfffffd7fffdff820
```

```
rbx     0xfffffd7fff3fb190
rdx     0x0000000000401b50
rcx     0x0000000000401b54
rax     0x0000000000416cf0
trapno  0x0000000000000003
err     0x0000000000000000
rip     0x0000000000401709:main+0xf9    movl $0x0000000000000000,0xfffffffffffffffc(%rbp)
cs      0x000000000000004b
eflags  0x0000000000000206
rsp     0xfffffd7fffdff7b0
ss      0x0000000000000043
fs      0x00000000000001bb
gs      0x0000000000000000
es      0x0000000000000000
ds      0x0000000000000000
fsbase  0xfffffd7fff3a2000
gsbase  0xffffffff80000000
(dbx) regs -F
current frame:  [1]
r15     0x0000000000000000
r14     0x0000000000000000
r13     0x0000000000000000
r12     0x0000000000000000
r11     0x0000000000401b58
r10     0x0000000000000000
r9      0x0000000000401c30
r8      0x0000000000416cf0
rdi     0x0000000000416cf0
rsi     0x0000000000401c18
rbp     0xfffffd7fffdff820
rbx     0xfffffd7fff3fb190
rdx     0x0000000000401b50
rcx     0x0000000000401b54
rax     0x0000000000416cf0
trapno  0x0000000000000003
err     0x0000000000000000
rip     0x0000000000401709:main+0xf9    movl     $0x0000000000000000,0xfffffffffffffffc(%rbp)
cs      0x000000000000004b
eflags  0x0000000000000206
rsp     0xfffffd7fffdff7b0
ss      0x0000000000000043
fs      0x00000000000001bb
gs      0x0000000000000000
es      0x0000000000000000
ds      0x0000000000000000
fsbase  0xfffffd7fff3a2000
gsbase  0xffffffff80000000
st0     +0.00000000000000000000e+00
st1     +0.00000000000000000000e+00
st2     +0.00000000000000000000e+00
st3     +0.00000000000000000000e+00
st4     +0.00000000000000000000e+00
st5     +0.00000000000000000000e+00
st6     +0.00000000000000000000e+00
```

```
st7     +NaN
xmm0a-xmm0d     0x00000000 0xfff80000 0x00000000 0x00000000
xmm1a-xmm1d     0x00000000 0x00000000 0x00000000 0x00000000
xmm2a-xmm2d     0x00000000 0x00000000 0x00000000 0x00000000
xmm3a-xmm3d     0x00000000 0x00000000 0x00000000 0x00000000
xmm4a-xmm4d     0x00000000 0x00000000 0x00000000 0x00000000
xmm5a-xmm5d     0x00000000 0x00000000 0x00000000 0x00000000
xmm6a-xmm6d     0x00000000 0x00000000 0x00000000 0x00000000
xmm7a-xmm7d     0x00000000 0x00000000 0x00000000 0x00000000
xmm8a-xmm8d     0x00000000 0x00000000 0x00000000 0x00000000
xmm9a-xmm9d     0x00000000 0x00000000 0x00000000 0x00000000
xmm10a-xmm10d   0x00000000 0x00000000 0x00000000 0x00000000
xmm11a-xmm11d   0x00000000 0x00000000 0x00000000 0x00000000
xmm12a-xmm12d   0x00000000 0x00000000 0x00000000 0x00000000
xmm13a-xmm13d   0x00000000 0x00000000 0x00000000 0x00000000
xmm14a-xmm14d   0x00000000 0x00000000 0x00000000 0x00000000
xmm15a-xmm15d   0x00000000 0x00000000 0x00000000 0x00000000
fcw-fsw  0x137f 0x0000
fctw-fop        0x0000 0x0000
frip     0x0000000000000000
frdp     0x0000000000000000
mxcsr    0x00001f80
mxcr_mask       0x0000ffff
(dbx)
```

プラットフォーム固有のレジスタ

このセクションの表は、式で使用できる SPARC アーキテクチャー、x86 アーキテクチャー、および AMD64 アーキテクチャーのプラットフォーム固有のレジスタ名を示しています。

SPARC レジスタ情報

次の表は、SPARC アーキテクチャーのレジスタ情報を示しています。

レジスタ	説明
`$g0 - $g7`	大域レジスタ
`$o0 - $o7`	「出力」レジスタ
`$l0 - $l7`	「局所」レジスタ
`$i0 - $i7`	「入力」レジスタ
`$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 $o6 と等価)
`$y`	Y レジスタ

レジスタ	説明
$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 - $f31	FPU「f」レジスタ
$fsr	FPU ステータスレジスタ
$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 とも呼ばれます。

次の 4 倍精度浮動小数点レジスタは、C の long double 型を持つと見なされます。これらのレジスタは、SPARC V9 ハードウェア上で使用できます。

```
$q0 $q4 through $q60
```

次のレジスタペアは、2 つのレジスタの下位 32 ビットを組み合わせたもので、SPARC V8+ ハードウェアで使用できます。

```
$g0g1 through $g6g7
$o0o1 through $o6o7
```

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。

```
$xg0 through $xg7
$xo0 through $xo7
$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

次の表は、x86 アーキテクチャーのレジスタ情報を示しています。

レジスタ	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
$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	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)。
$ps	

次の表は、80386 の下位半分 (16 ビット) のレジスタを示しています。

レジスタ	説明
$ax	汎用レジスタ
$cx	汎用レジスタ
$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)

80386 の最初の 4 つの 16 ビットレジスタは、次の表に示すように、8 ビットの部分に分割できます。

レジスタ	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

次の表は、80387 の各半分のレジスタを示しています。

レジスタ	説明
$fctrl	コントロールレジスタ
$fstat	ステータスレジスタ
$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レクタ
$fopoff	オペランドポインタオフセット

レジスタ	説明
$fopsel	オペランドポインタセレクタ
$st0 - $st7	データレジスタ

AMD64 レジスタ情報

次の表は、AMD64 アーキテクチャーのレジスタ情報を示しています。

レジスタ	説明
rax	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
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
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
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

レジスタ	説明
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
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
os	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
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
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
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

レジスタ	説明
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
ymmo	256 ビット Advanced Vector レジスタ
ymm1	256 ビット Advanced Vector レジスタ
ymm2	256 ビット Advanced Vector レジスタ
ymm3	256 ビット Advanced Vector レジスタ
ymm4	256 ビット Advanced Vector レジスタ
ymm5	256 ビット Advanced Vector レジスタ
ymm6	256 ビット Advanced Vector レジスタ
ymm7	256 ビット Advanced Vector レジスタ
ymm8	256 ビット Advanced Vector レジスタ
ymm9	256 ビット Advanced Vector レジスタ
ymm10	256 ビット Advanced Vector レジスタ
ymm11	256 ビット Advanced Vector レジスタ
ymm12	256 ビット Advanced Vector レジスタ
ymm13	256 ビット Advanced Vector レジスタ
ymm14	256 ビット Advanced Vector レジスタ
ymm15	256 ビット Advanced Vector レジスタ

Advanced Vector (AVX) レジスタ (ymm0 - ymm15) の各フィールドは、C int、float、または double 型を持つと見なすことができます。

♦ ♦ ♦ 第 19 章

19 dbx の Korn シェル機能

dbx のコマンド言語は、Korn シェル (ksh 88) の構文に基づいています (I/O リダイレクト、ループ、組み込み演算、履歴、およびコマンド行編集を含む)。この章では、ksh-88 と dbx のコマンド言語の違いについて説明します。

dbx 初期化ファイルが起動時に見つからない場合、dbx は ksh モードと見なします。

この章には次のセクションが含まれています。

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。

- 配列 *name* に値を割り当てるための `set -A` name
- `set -o` の次のオプション: `allexport bgnice gmacs markdirs noclobber nolog privileged protected viraw`
- `typeset` の `-l -u -L -R -H` の各属性
- 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 [[*expression*]] による式の評価
- `@(pattern[|pattern] …)` による拡張パターン照合
- 同時処理 (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 への拡張機能

dbx では、次の機能が追加されました。

- `$[ p? > flags ]` 言語式
- `typeset -q` (ユーザー定義関数のための特殊な引用を可能にする)
- C シェルに似た `history` と `alias` 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)
- 組み込みの `dbx` コマンド

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、特定の `dbx` コマンドの名前が変更されました。

- `dbx` の `print` コマンドは `print` という名前を維持し、`ksh` の `print` コマンドは `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` コマンドです。
- *address*`/`*format* は、`examine` *address*`/`*format* になりました。
- `/`*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` コマンドを使用すると、編集機能を再バインドできます。このコマンドでは、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。

`bind`	現在の編集機能のキーバインドを表示します。
`bind` *key*=*definition*	*key* を *definition* にバインドします。
`bind` *key*	*key* の現在の定義を表示します。
`bind` *key*=	*key* をバインド解除します。
`bind -m` *key*=*definition*	*key* を *definition* のマクロとして定義します。
`bind -m`	`bind` と同じです。

ここでは:

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。

Emacs 風のエディタでのより重要なデフォルトのキーバインドのいくつかを次に示します。

^A = beginning-of-line	^B = backward-char
^D = eot-or-delete	^E = end-of-line
^F = forward-char	^G = abort
^K = kill-to-eo	^L = redraw
^N = down-history	^P = up-history
^R = search-history	^^ = quote
^? = delete-char-backward	^H = delete-char-backward
^[b = backward-word	^[d = delete-word-forward
^[f = forward-word	^[^H = delete-word-backward
^[^[= complete	^[? = list-command

vi 風のエディタでのより重要なデフォルトのキーバインドのいくつかを次に示します。

a = append	A = append at EOL
c = change	d = delete
G = go to line	h = backward character
i = insert	I = insert at BOL
j = next line	k = previous line
l = forward line	n = next match
N = prev match	p = put after

P = put before
R = replace
u = undo
X = delete previous character
~ = transpose case
* = expand
- = previous line
sp = forward char
? = search history from beginning
/ = search history from current
r = repeat
s = substitute
x = delete character
y = yank
_ = last argument
= = list expansion
+ = next line
# = comment out command

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
^U = kill line
^H = delete character
^W = delete word

♦ ♦ ♦ 第 20 章

20 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を使用するプログラムのデバッグを完全にサポートしていますが、ライブラリが -g オプションを使用してコンパイルされている場合に限ります。

この章には次のセクションが含まれています。

動的リンカー

動的リンカーは rtld、実行時 ld、または ld.so とも呼ばれ、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) を組み込む準備をします。rtld がアクティブになる 2 つの主な領域:

- 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。これらの*プリロードされた*共有オブジェクトには、libc.so、libC.so、libX.so などがあります。ldd (1) を使用すると、プログラムによってロードされる共有オブジェクトがわかります。
- 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。

dbx では*ロードオブジェクト*という用語を使用して、共有オブジェクト (.so) や実行可能ファイル (a.out) を表します。loadobject コマンドを使用して、ロードオブジェクトのシンボリック情報を一覧表示し、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。リンクマップは、デバッグされているプログラムのメモリーに保存され、デバッガによって使用される特殊なシステムライブラリの `librtld_db.so` によって間接的にアクセスされます。

起動手順と .init セクション

`.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分です。たとえば、`.init` セクションは、C++ 実行時システムが `.so` ファイル内のすべての静的初期化関数を呼び出すために使用します。

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。その後、動的リンカーはリンクマップをトラバースし、各共有オブジェクトに対して `.init` セクションを実行します。`syncrtld` イベントはこれらの 2 つのフェーズの間に発生します。詳細については、307 ページの「`syncrtld` イベント指定」を参照してください。

プロシージャリンケージテーブル

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。詳細については、汎用およびプロセッサ固有の SVR4 ABI のリファレンスマニュアルを参照してください。

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` ハンドシェークと同時に取得されます。

修正と継続

`dlopen()` でロードされた共有オブジェクトで修正と継続を使用する場合、それらを開く方法に変更が必要です。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には、dbx はプログラムの実行時にプログラムがそのライブラリを使用することを確認する必要があり、dbxは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。新しくロードされたプログラムがその実行時に使用するライブラリを特定するため、dbx は実行時リンカーがすべての起動ライブラリをロードするために十分な長さだけ、プログラムを実行します。そして、dbx はロードされたライブラリのリストを読み取ってプロセスを強制終了します。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、プログラムが dbx コマンドによってコマンド行からロードされたか、debug コマンドで dbx プロンプトからロードされたか、または IDE でロードされたかに関係なく、ライブラリをロードするために同じ手順に従います。

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 dlclose() 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。

共有オブジェクトが dlclose() を使用してアンロードされた場合、dbx はそれに配置されていたブレークポイントを記憶しているので、たとえアプリケーションが再実行されても、dlopen() によって共有オブジェクトが再びロードされた場合に、それらを再配置します。

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れば、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()` によってロードされた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を設定できません。
- `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。そのため、`dlopen()` によってロードされたライブラリ内の `_init()` で `dbx` を停止させることはできません。

♦ ♦ ♦ 付 録 A

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。プログラムがどのコマンドによって変更される可能性があるかを理解することが重要です。

この章は、次のセクションで構成されています。

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、監視がそのプロセスに影響を与えてはいけません。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。場合によっては、簡単な監視が実行に影響を与え、間欠的なバグ症状を引き起こすことがあります。

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 のサポートライブラリ librtc.so をプログラムにロードすると、そのプログラムの動作が変わる場合があります。
- dbx の初期設定スクリプトによって、忘れている一部の環境変数が設計されている可能性があります。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。このアドレスはまた、環境や argv[] の内容によっても異なる可能性があり、それによって局所変数の割り当て方法が強制的に変更されます。これらの変数が初期化されていない場合は、別の乱数が生成されます。この問題は、実行時検査によって検出できます。

- プログラムが、malloc() で割り当てられたメモリーを使用前に初期化しません。この問題は、実行時検査によって検出できます。
- dbx は LWP 作成および dlopen イベントを捕獲する必要があり、これが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に影響を与え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、アプリケーションでシグナルを頻繁に使用している場合は、動作が異なる可能性があります。
- 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 の下での実行はアドレス空間に大きな影響を与えるため、mmap() が、dbx なしでプログラムが実行された場合と同じアドレスを返さなくなる可能性があります。これが問題であるかどうかを判定するには、mmap() のすべての使用を調べ、プログラムがハードコードされたアドレスではなく、返されたアドレスを使用していることを確認します。
- 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 の下での実行はスレッドのスケジューリングを乱すため、プログラムが通常とは異なる順序でスレッドを実行する可能性があります。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 や truss を使用して実行しても同じ問題が発生するかどうかを確認してください。

dbx によって強い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

このセクションで説明されているコマンドは、プログラムを変更する可能性があります。

assign コマンド

assign コマンドは、*式*の値を*変数*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します。

`pop`	現在のフレームをポップします。
`pop` *number*	*number* 個のフレームをポップします。
`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、387 ページの「pop コマンド」を参照してください。

call コマンド

dbx で call コマンドを使用すると、手続きが呼び出され、その手続きが指定されたとおりに実行されます。

```
call proc([params])
```

この手続きによって、プログラムが変更される可能性があります。dbx は呼び出しを、ユーザーがそれをプログラムソースに記述しているかのように実行します。

詳細については、327 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含まれている場合は、その式を出力すると call コマンドが実行されます。そのため、327 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、388 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、これらのコマンドが実行されます。どのコマンドが発行されたかに応じて、このアクションがプログラムの状態を変更する可能性があります。

詳細については、431 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、プログラムを即座に変更できます。

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。172 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、358 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。実行は、*line* 行で継続されます。ID は、プログラムがマルチスレッド化されている場合に必要です。

cont at *line* [*ID*]

このコマンドは、プログラムの結果を変更する可能性があります。

♦ ♦ ♦ 付 録 B

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。

この付録には次のセクションが含まれています。

イベントハンドラ

イベント管理は*ハンドラ*の概念に基づきます。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。各イベント管理コマンドは一般にハンドラを作成し、それは*イベント指定*と一連の副作用アクションで構成されます。(292 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 (308 ページの「イベント指定修飾子」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汎用的な形式は、when コマンドを使用することです。

```
when event-specification {action; ... }
```

この章の例では、when に関してコマンド (stop、step、ignore など) を書く方法を示します。これらの例は、when とその配下にある「ハンドラ」メカニズムの柔軟性を示すものですが、常に同じ働きをするとはかぎりません。

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 コマンド、stop コマンド、trace コマンドを使用します。(詳細については、431 ページの「when コマンド」、405 ページの「stop コマンド」、および 418 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 で、目的のイベントを指定するために使用します (292 ページの「イベント指定の設定」を参照)。

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。

すべてのコマンドは、ハンドラ ID (*hid*) と呼ばれる番号を返します。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。

イベントハンドラの操作

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、それぞれのセクションを参照してください。

表 B-1 イベントハンドラの操作

コマンド	説明	詳細情報
status	ハンドラを一覧表示します	402 ページの「status コマンド」を参照してください

コマンド	説明	詳細情報
delete	一時ハンドラを含むすべてのハンドラを削除します	349 ページの「delete コマンド」を削除してください
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	333 ページの「clear コマンド」を参照してください
handler -enable	ハンドラを有効にします	363 ページの「handler コマンド」を参照してください
handler -disable	ハンドラを無効にします	363 ページの「handler コマンド」を参照してください
cancel	シグナルを取り消し、プロセスが続行できるようにします	329 ページの「cancel コマンド」を参照してください

イベントカウンタの使用

イベントハンドラには、カウント制限を保持するトリップカウンタがあります。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、ハンドラに関連付けられたアクションが実行されるのは、カウントが制限値に達したときのみで、その時点でカウンタは自動的に 0 にリセットされます。デフォルトの制限値は 1 です。プロセスを再実行する際は常に、すべてのイベントカウンタがリセットされます。

カウント制限を設定するには、stop コマンド、when コマンド、trace コマンドで -count 修飾子を使用します。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
```

イベントの安全性

dbx では、イベントメカニズムによって、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 プログラム状態の変更と94 ページの「呼び出しの安全性」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 FLTTRACE イベントに基づきます。そのため、コマンド stop fault FLTTRACE 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。これらのフェーズに含まれるもの:

- プログラムの起動時に rtld が実行された場合 (281 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 (192 ページの「fork 機能後のプロセス追跡」および192 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 がスタックのマップされたページを確実に利用できるようにする必要があ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 ハンドシェーク時および前述のその他のフェーズで stop コマンドを無視する

例:

```
...SolBook linebreakstopped in munmap at 0xff3d503c 0xff3d503c: munmap+0x0004: ta
 %icc,0x00000008SolBook linebreak 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。

イベント指定の設定

イベント指定は、stop コマンド、stopi コマンド、when コマンド、wheni コマンド、trace コマンド、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。書式は、イベントタ

イプを表すキーワードとオプションのパラメータで構成されます。イベント指定の意味は、一般に 3 つすべてのコマンドで同じです。例外は付録 D のコマンドの説明に記載されています。

ブレークポイントイベント指定

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。このセクションでは、ブレークポイントイベントのイベント指定について説明します。

in イベント指定

in イベント指定の構文:

in*function*

関数に入り、先頭行が実行される直前です。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である場合があります。C++ のコンストラクタの場合、すべてのベースクラスのコンストラクタの実行後に実行されます。-instr 修飾子が使用された場合、それ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。例:

```
stop in mumble(int, float, struct Node *)
```

注記 - in *function* と -in *function* 修飾子とを混同しないでください。

at イベント指定

at イベント指定の構文:

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、指定したファイルの指定の行が実行される直前です。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、必要な場合があります。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。

特定のアドレスを指定することもできます。

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 コマンドまたは -instr イベント修飾子でのみ使用できます。

infile イベント指定

infile イベント指定の構文:

infile *filename*

このイベントにより、ファイルで定義されたすべての関数にブレークポイントが設定されます。stop infile コマンドは、funcs -f *filename* コマンドと同じ関数のリストを繰り返します。

.h ファイル内のメソッド定義、テンプレートファイル、または .h ファイル内のプレーン C コード (regexp コマンドで使用される種類など) は、ファイルの関数定義に寄与する場合がありますが、これらの定義は除外されます。

指定されたファイル名が、オブジェクトファイルの名前の場合 (その場合、名前は .o で終了する)、ブレークポイントは、そのオブジェクトファイルで発生する関数すべてに設定されます。

stop infile list.h コマンドは、list.h ファイルで定義されたメソッドのすべてのインスタンスにブレークポイントを設定することはしません。そうするためには、inclass または inmethod のようなイベントを使用します。

fix コマンドは、関数をファイルから削除する、または追加する場合があります。stop infile コマンドは、ファイル内の関数のすべての古いバージョンと、将来追加されるすべての関数にブレークポイントを設定します。

ネストされた関数や Fortran ファイルのサブルーチンには、ブレークポイントは設定されません。

clear コマンドを使用して、infile イベントによって作成された組にある単一のブレークポイントを無効にできます。

infunction イベント指定

infunction イベント指定の構文:

`infunction`*function*

この指定は、*function* という名前のすべての多重定義関数またはそれのすべてのテンプレートインスタンスに対する `in` *function* と同等です。

inmember イベント指定

`inmember` イベント指定の構文:

`inmember` *function*

この指定は `inmethod` イベント指定のエイリアスです。

inmethod イベント指定

`inmember` イベント指定の構文:

`inmethod` *function*

この指定は、すべてのクラスの *function* という名前のメンバーメソッドに対する `in` *function* と同等です。

inclass イベント指定

`inclass` イベント指定の構文:

`inmember` *classname* `[-recurse | -norecurse]`

この指定は、*classname* のメンバーであるが、*classname* のベースのメンバーではない、すべてのメンバー関数に対する `in` *function* と同等です。デフォルトは `-norecurse` です。`-recurse` が指定された場合、基底クラスが含まれます。

inobject イベント指定

`inobject` イベント指定の構文:

`inobject` *object-expression* `[-recurse | -norecurse]`

object-expression で示されるアドレスにある特定のオブジェクトに対して呼び出されたメンバー関数が呼び出されました。`stop inobject` *ox* はほぼ次と同じですが、`inclass` とは異なり、*ox* の動的タイプ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

このセクションでは、メモリーアドレスの内容へのアクセスや変更に関するイベントのイベント指定について説明します。

access イベント指定

`access` イベント指定の構文:

```
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。有効な値は次のいずれかまたはすべての文字です。

`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。記号式を指定すると、監視対象領域のサイズが自動的に推定されます。*byte-size-expression* を指定して、それ

をオーバーライドすることができます。さらに、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、その場合はサイズが必須です。例:

```
stop access w 0x5678, sizeof(Complex)
```

access コマンドには、2 つの一致する領域が重複しない、という制限があります。

注記 - access イベント仕様は、modify イベント仕様の代替です。

change イベント指定

change イベント指定の構文:

change *variable*

variable の値は変更されました。change イベントは、次とほぼ同等です。

```
when step { if [ $last_value !=$[variable]]
            then
                 stop
            else
                 last_value=$[variable]
            fi
          }
```

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。パフォーマンスの向上のため、access イベントを使用してください。

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 イベント指定

cond イベント指定の構文:

cond *condition-expression*

condition-expression で示された条件は true に評価されます。*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 stop コマンドとほぼ同等です。

stop step -if *conditional-expression*

システムイベント指定

このセクションでは、システムイベントのイベント指定について説明します。

dlopen および dlclose イベント指定

dlopen() および dlopen() イベント指定の構文:

dlopen [*lib-path*]

dlclose [*lib-path*]

システムイベントは、dlopen() の呼び出しまたは dlclose() の呼び出しが成功したあとに発生します。dlopen() の呼び出しまたは dlclose() の呼び出しにより、複数のライブラリがロードさ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 $dllist でいつでも入手できます。$dllist の中の最初のシェルの単語は + (プラス記号) または - (マイナス記号) で、ライブラリのリストに追加されているか、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、$dlobj にライブラリの名前が格納されます。また、$dllist 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。$dlobj は空になりますが、$dllist は有効です。

fault イベント指定

fault イベント指定の構文:

fault *fault*

fault イベントは、指定された障害が検出されたときに発生します。障害は、アーキテクチャー依存です。dbx に認識される一連の障害を次のリストに示し、proc(4) マニュアルページで定義しています。

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
FLTTRACE*	トレーストラップ (ステップ実行)
FLTACCESS	メモリーアクセス (境界合わせなど)
FLTACCESS	メモリーアクセス (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	回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

注記 - FLTBPT、FLTTRACE、および FLTWATCH は、dbx でブレークポイント、ステップ実行、ウォッチポイントを実装するために使われるため、処理されません。

これらの障害は、/sys/fault.h から抜粋されています。*fault* には上に挙げたいずれかを大文字または小文字、FLT- 接頭辞を付けるか付けないで指定するか、または実際の数値コードを指定できます。

注記 - 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 イベント指定

lwp_exit イベント指定の構文:

```
lwp_exit
```

lwp_exit イベントは、lwp が終了したときに発生します。$lwp には、イベントハンドラの継続時間中に終了した LWP (軽量プロセス) の ID が含まれます。

注記 - 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 イベント指定

sig イベント指定の構文:

sig*signal*

sig *signal* イベントは、デバッグ中のプログラムにシグナルが初めて送られたときに、発生します。*signal* は、10 進数、または大文字、小文字のシグナル名のいずれかです。接頭辞はオプションです。このイベントは、catch コマンドおよび 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装することができます。

```
function simple_catch {
    when sig $1 {
            stop;
            echo Stopped due to $sigstr $sig
            whereami
    }
}
```

注記 - 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。

または、サブコードでシグナルを指定することができます。sig イベント指定のこのオプションの構文:

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 *signalsub-code* イベントが発生します。シグナルと同様に、*sub-code* は 10 進数として、大文字または小文字で指定することができます。接頭辞はオプションです。

sysin イベント指定

sysin イベント指定の構文:

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 でサポートされるシステムコールの概念は、/usr/include/sys/syscall.h に列挙されるように、カーネルへのトラップによって提供されるものです。

この概念は、ABI のシステムコールの概念とは違います。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。

注記 - sysin イベントは、Linux プラットフォームでは使用できません。

/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、Oracle Solaris OS のプライベートインタフェースの一部であり、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 のそれらと正確に一致することはありえないため、syscall.h 内の一部の名前は使用でき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 イベント指定

sysout イベント指定の構文:

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。

注記 - 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 イベント指定

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

このセクションでは、実行進行状況に関するイベントのイベント指定について説明します。

exit イベント指定

exit イベント指定の構文:

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 イベント指定

next イベントは、ステップインしない関数を除いて、step イベントと似ています。

returns イベント指定

returns イベントは、現在*アクセスされている*関数の戻りポイントにある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 イベント指定の構文:

returns*function*

returns *function* イベントは、特定の関数がその呼び出し場所に戻るたびに実行されます。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 ベースのシステム – $o0
- x86 ベースのシステム – $eax
- x64 ベースのシステム – $rax、$rdx

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 イベント指定

step イベントは、ソース行の最初の命令が実行されたときに発生します。たとえば、次のコマンドで簡単なトレースを取得できます。

```
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。

注記 - 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 イベントに関して、大ざっぱに次のように実装します。alias step="when step -temp { whereami; stop; }; cont"

---

throw イベント指定

throw イベントの構文:

```
throw [type | -unhandled | -unexpected]
```

throw イベントは、アプリケーションによって、処理されないか、予期しない例外がスローされるたびに発生します。

throw イベントで例外のタイプが指定されている場合、そのタイプの例外だけが throw イベントを発生させます。

-unhandled オプションが指定されている場合、例外を示す特殊な例外タイプがスローされますが、それに対してハンドラが存在しません。

-unexpected オプションが指定されている場合、例外を示す特殊な例外タイプはそれをスローした関数の例外指定を満たしていません。

追跡されたスレッドイベント指定

次のセクションでは、追跡されたスレッドのイベント指定について説明します。

omp_barrier イベント指定

omp_barrier イベント指定は、追跡されたスレッドがバリアに入るか、出るタイミングです。*type* (explicit または implicit になる) と、*state* (enter、exit、または all_entered になる) を指定できます。デフォルトは explicit all_entered です。

omp_taskwait イベント指定

omp_taskwait イベント指定は、追跡されたスレッドが taskwait に入るか、出るタイミングです。*state* を指定でき、これは enter または exit になります。デフォルトは exit です。

omp_ordered イベント指定

omp_ordered イベント指定は、追跡されたスレッドが Ordered 領域に入るか、出るタイミングです。*state* を指定でき、これは begin、enter、または exit になります。デフォルトは enter です。

omp_critical イベント指定

omp_critical イベント指定は、追跡されたスレッドがクリティカル領域に入るか、出るタイミングです。

omp_atomic イベント指定

omp_atomic イベント指定は、追跡されたスレッドが不可分領域に入るか、出るタイミングです。*state* を指定でき、これは begin または exit になります。デフォルトは begin です。

omp_flush イベント指定

omp_flush イベント指定は、追跡されたスレッドが明示的なフラッシュ領域に入るか、出るタイミングです。

omp_task イベント指定

omp_task イベント指定は、追跡されたスレッドがタスク領域に入るか、出るタイミングです。*state* を指定でき、これは create、start、または finish になります。デフォルトは start です。

omp_master イベント指定

omp_master イベント指定は、追跡されたスレッドがマスター領域に入るか、出るタイミングです。

omp_single イベント指定

omp_single イベント指定は、追跡されたスレッドが Single 領域に入るタイミングです。

その他のイベント指定

このセクションでは、その他のタイプのイベントのイベント指定について説明します。

attach イベント指定

attach イベントは、dbx がプロセスに正常に接続したタイミングです。

detach イベント指定

detach イベントは dbx がデバッグ中のプログラムから正常に切り離されたタイミングです。

lastrites イベント指定

lastrites イベントは、デバッグ中のプロセスが終了する直前のタイミングです。これは次の理由によって発生する可能性があります。

- _exit(2) システムコールが、明示的な呼び出し経由か、または main() の戻り時に呼び出された。
- 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。

注記 - 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 イベント指定

proc_gone イベントは dbx がデバッグ対象のプロセスに関連付けられなくなったときに発生します。事前定義済み変数 $reason は、signal、exit、kill、または detach になります。

prog_new イベント指定

prog_new イベントは、follow exec の結果として新しいプログラムがロードされたときに発生します。

注記 - このイベントのハンドラは常に存在しています。

stop イベント指定

stop イベントは、特に stop ハンドラへの応答として、ユーザーがプロンプトを受け取るなど、プロセスが停止するたびに発生します。次に例を示します。

```
display x
when stop {print x;}
```

sync イベント指定

sync イベントは、デバッグ中のプロセスが exec() で実行された直後に発生します。a.out で指定されたメモリー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、まだメモリー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。

注記 - 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 イベント指定

syncrtld イベントは、sync、またはデバッグ中のプロセスがまだ共有ライブラリを処理していない場合は attach のあとに発生します。これは、動的リンカーの起動コードが実行され、プリロード済みのすべての共有ライブラリのシンボルテーブルがロードされたあと、ただし、.init セクション内のコードが実行される前に実行します。

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*] イベント指定

thr_create イベントは、スレッドまたは指定したスレッド ID を持つ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 イベント指定

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
```

timer イベント指定

timer イベントの構文:

timer*seconds*

timer イベントは、デバッグ中のプログラムが *seconds* 秒間実行されたときに発生します。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。修飾子はダッシュ (-) から始まります。各修飾子の構成は次のとおりです。

-if 修飾子

-if 修飾子の構文:

-if*condition*

イベント仕様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 修飾子が、in または at などの 1 つだけのソース位置が関連付けられたイベントで使用された場合、*condition* 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。

マクロ展開は、print コマンドと同じ規約に従った条件で実行されます。

-resumeone 修飾子

-resumeone 修飾子は、マルチスレッドプログラムのイベント指定で、-if 修飾子とともに使用して、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させることができます。詳細については、106 ページの「条件付きフィルタによるブレークポイントの修飾」を参照してください。

-in 修飾子

-in 修飾子の構文:

-in*function*

イベントは指定した関数の最初の命令に達したときから、関数が戻るときまでの間に発生した場合にのみトリガーします。関数の再帰は無視されます。

-disable 修飾子

-disable 修飾子は無効状態でハンドラを作成します。

-count *n*, -count infinity 修飾子

-count 修飾子の構文:

-count*n*

または

-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 (291 ページの「イベントカウンタの使用」を参照)。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 (346 ページの「debug コマンド」を参照) または attach -r コマンド (326 ページの「attach コマンド」を参照) 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 修飾子

-temp 修飾子は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す。

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 修飾子

`-instr` 修飾子はハンドラ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 修飾子

`-thread` 修飾子の構文:

`-thread`*thread-ID*

`-thread` 修飾子はイベントを発生させたスレッドが異なるスレッド ID に一致する場合にのみ、アクションが実行されることを意味します。プログラムが次から次に実行されるうちに、目的とする特定のスレッドに、異なるスレッド ID が割り当てられることがあります。

-lwp 修飾子

`-lwp` 修飾子の構文:

`-lwp`*lwp-ID*

`-lwp` 修飾子はイベントを発生させたスレッドが *lwp-ID* に一致する場合にのみ、アクションが実行されることを意味します。イベントを発生させたスレッドが *lwp-ID* と一致する場合にのみ、アクションが実行されます。プログラムが次から次に実行されるうちに、目的とする特定のスレッドに、異なる *lwp-ID* が割り当てられることがあります。

-hidden 修飾子

`-hidden` 修飾子は 通常の `status` コマンドでハンドラを非表示にします。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 修飾子

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、デバッグセッション間でハンドラが維持されます。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

イベント指定と修飾子のための構文はキーワードドリブンで、ksh 規則に基づきます。すべてのものがスペースで区切られた単語に分割されます。

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
```

最初の例では、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 構文解析プログラムは -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* という変数がない場合、エラーが発生します。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。

事前定義済み変数の使用

特定の読み取り専用の ksh 事前定義済み変数が用意されています。次の表に示す変数は常に有効です。

変数	定義
$ins	現在の命令の逆アセンブル
$lineno	現在の行番号 (10 進数)
$vlineno	現在の表示行番号 (10 進数)
$line	現在の行の内容
$func	現在の関数の名前
$vfunc	現在の表示関数の名前
$class	$func が所属するクラスの名前

変数	定義
$vclass	$vfunc が所属するクラスの名前
$file	現在のファイルの名前
$vfile	現在表示しているファイルの名前
$loadobj	現在のロードオブジェクトの名前
$vloadobj	現在表示している現在のロードオブジェクトの名前
$scope	逆引用符表記での現在の PC のスコープ
$vscope	現在表示している逆引用符表記での PC のスコープ
$funcaddr	$func のアドレス (16 進数)
$caller	$func を呼び出している関数の名前
$dllist	dlopen イベントまたは dlclose イベントのあと、ロードされた、またはアンロード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。dllist の先頭の単語は、dlopen または dlclose のどちらが発生したかによる + (プラス記号) または - (マイナス記号) です。
$newhandlerid	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。dbx 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。
$firedhandlers	停止の原因となった最近のハンドラ ID のリストです。リスト上のハンドラは、status コマンドの出力に * (アスタリスク) でマークされます。
$proc	現在デバッグ中のプロセスの ID
$lwp	現在の LWP の ID。
$thread	現在のスレッドの ID
$newlwp	新しく作成された LWP の ID。
$newthread	新しく作成されたスレッドの ID。
$prog	デバッグ中のプログラムの絶対パス名
$oprog	$prog の前の値で、これは exec() のあとにデバッグしていたものに戻るために使用され、このときにプログラムのフルパス名は - (ダッシュ) に戻ります。$prog がフルパス名に展開され、$oprog がコマンド行または debug コマンドに指定されているプログラムパスを含みます。exec() 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れません。
$exec32	dbx バイナリが 32 ビットの場合は true です。
$exitcode	プログラムの最後の実行ステータスを終了します。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。

変数	定義
$booting	イベントがブートプロセス中に発生すると、true 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、共有ライブラリのリストと位置を確認できるよう、まず実行されます。プロセスはそのあと終了します。このシーケンスは「ブート」と呼ばれます。 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、たとえばデバッグの実行中に発生する sync および syncrtld イベントと、通常の実行中に発生するイベントを区別するために使用します。
$machtype	プログラムがロードされた場合、そのマシンタイプ sparcv8、sparcv8+、sparcv9、または intel を返します。そうでない場合、unknown を返します。

$datamodel	プログラムがロードされた場合、そのデータモデル ilp32 または lp64 を返します。そうでない場合、unknown を返します。ロードしたばかりのプログラムのモデルを見つけるには、.dbxrc ファイルで次を使用します。
	
	`when prog_new -perm {`
	`  echo machine: $machtype $datamodel;`
	`}`

次の例に、whereami を実装できることを示します。

```
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

このセクションで説明する変数は、when コマンド本体内でのみ有効です。

$handlerid

本体の実行中、$handlerid は本体が属する when コマンドの ID です。次のコマンドは同等です。

```
when X -temp { do_stuff; }
when X  { do_stuff; delete $handlerid; }
```

when コマンドと特定のイベントに対して有効な変数

特定の変数は、以下の表に示すように、when コマンドの本体内および特定のイベントに対してのみ有効です。

表 B-2 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-3 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-4 dlopen および dlclose イベントに有効な変数

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-5 sysin および sysout イベントに有効な変数

変数	説明
$syscode	システムコール番号
$sysname	システムコール名

表 B-6 proc_gone イベントに固有の変数

変数	説明
$reason	signal、exit、kill、または detach のいずれか

表 B-7 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)

表 B-8 access イベントに有効な変数

変数	説明
$watchaddr	アドレスが書き込まれたり、読みだされたり、実行されたりします。
$watchmode	次のいずれかです。r は読み込み、w は書き込み、x は実行。そのあとに次のいずれかが続きます。a は後、b は前。

イベントハンドラの例

このセクションでは、イベントハンドラの設定のいくつかの例を挙げます。

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

この例は、array[99] にデータ変更ブレークポイントを設定する方法を示しています。

```
(dbx) stop access w &array[99]
(2) stop access w &array[99], 4
(dbx) run
Running: watch.x2
watchpoint array[99] (0x2ca88[4]) at line 22 in file "watch.c"
   22    array[i] = i;
```

単純なトレースを実行する

この例は、単純なトレースを実装する方法を示しています。

```
(dbx) when step { echo at line $lineno; }
```

関数内にいる間ハンドラを有効にする

次の例に、関数内にいる間ハンドラを有効にする方法を示します。

```
<dbx> trace step -in foo
```

このコマンドは次と同等です。

```
    # 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

この例は、小さなプログラムで実行された行数を確認する方法を示しています。入力:

```
(dbx) stop step -count infinity     # step and stop when count=inf
(2) stop step -count 0/infinity
(dbx) run
...
(dbx) status
(2) stop step -count 133/infinity
```

プログラムは停止することなく、プログラムが終了します。実行された行の数は 133 です。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

この例は、コードのある行で実行する命令数をカウントする方法を示しています。

```
(dbx) ...                       # get to the line in question
(dbx) stop step -instr -count infinity
(dbx) step ...
(dbx) status
(3) stop step -count 48/infinity # 48 instructions were executed
```

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、プログラムが関数 hash で、ただし 1300 番目のシンボル検索の後にのみ、正しく実行しなくなった場合、次のブレークポイントを使用します。

```
(dbx) when in lookup -count 1300 {
    stop in hash
    hash_bpt=$newhandlerid
    when proc_gone -temp { delete $hash_bpt; }
}
```

注記 - $newhandlerid は、実行されたばかりの stop in コマンドを表しています。

replay 時にアプリケーションファイルをリセットする

この例では、アプリケーションで replay 時にリセットされる必要があるファイルを処理する場合、プログラムを実行するたびにこれ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...    # during which database gets clobbered
(dbx) save
...              # implies a RUN, which implies the SYNC event which
(dbx) restore       # causes regen to run
```

プログラムのステータスのチェック

この例は、プログラムの実行中にプログラムの場所をすばやく確認する方法を示しています。入力:

```
(dbx) ignore sigint
(dbx) when sig sigint { where; cancel; }
```

次に、^C を発行して、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ます。

この例は、基本的にコレクタ側の標本モードで実行することです (これだけではありません)。^C が使われているため、プログラムに割り込むには SIGQUIT (^\) を使用します。

浮動小数点例外の捕獲

次の例に、IEEE アンダーフローなど、特定の浮動小数点例外のみを捕獲する方法を示します。

```
(dbx) ignore FPE               # disable default handler
(dbx) help signals | grep FPE  # can't remember the subcode name
...
(dbx) stop sig fpe FPE_FLTUND
...
```

ieee ハンドラを有効にする詳細については、207 ページの「FPE シグナルのトラップ (Oracle Solaris のみ)」を参照してください。

♦ ♦ ♦ 付 録 C

マクロ

デフォルトでは、選択された式は、評価される前にマクロ展開されます。これには、print、display、watch の各コマンドで指定する式、stop、trace、when の各コマンドの -if オプション、および $[] 構造も含まれます。マクロ展開はまた、IDE や dbxtool でのバルーン評価とウォッチポイントにも適用されます。

マクロ展開の追加の使用

マクロ展開は、assign コマンド内の変数と式の両方に適用されます。

call コマンドでは、マクロ展開は呼び出される関数の名前だけでなく、渡されるパラメータにも適用されます。

macro コマンドは任意の式とマクロを受け取り、そのマクロを展開します。例:

```
(dbx) macro D(1, 2)
    Expansion of: D(1, 2)
              is: d(1,2)
```

whatis コマンドにマクロを指定すると、そのマクロの定義が表示されます。例:

```
(dbx) whatis B
    #define B(x) b(x)
```

which コマンドにマクロを指定すると、スコープ内で現在アクティブなマクロがどこで定義されているかが表示されます。例:

```
(dbx) which B2
    `a.out`macro_wh.c`B2    # defined at defs2.h:3
            # included from defs1.h:3
            # included from macro_wh.c:23
```

whereis コマンドにマクロを指定すると、そのマクロが定義されているすべての場所が表示されます。リストは、dbx がすでにデバッグ情報を読み取ったモジュールに限られます。例:

```
(dbx) whereis U
    macro:          U       # defined at macro_wh.c:21
    macro:          U       # undefined at defs1.h:5
```

dbxenv 変数 macro_expand は、これらのコマンドによってマクロが展開されるかどうかを制御します。デフォルトでは on に設定されます。

通常、dbx コマンドの +m オプションにより、コマンドでのマクロ展開は省略されます。-m オプションは、dbxenv 変数 macro_expand が off に設定されている場合でも、マクロ展開を強制します。1 つの例外は、$[] 構造内の -m オプションです。その場合は、-m マクロが展開されるだけで、評価は実行されません。この例外により、シェルスクリプトでのマクロ展開が容易になります。

マクロ定義

dbx はマクロ定義を 2 つの方法で認識できます。

- デバッグ情報のデフォルトの DWARF 形式を使用する場合は、-g3 オプションでコンパイルするときにコンパイラによって定義が提供されます。ただし、コンパイル時に -xdebugformat=stabs オプションを指定した場合は提供されません。
- dbx は、ソースファイルとそのインクルードファイルのスキミングを行うことによって定義を再作成できます。正確な再作成は、元のソースとインクルードファイルへのアクセスに依存しています。また、使用されているコンパイラがパス名を使用できるかどうか、および -D や -I などのコンパイラオプションにも依存します。この情報は Oracle Solaris Studio コンパイラから DWARF 形式とスタブ形式の両方で使用できますが、GNU からは使用できません。スキミングを確実に成功させる方法については、322 ページの「スキミングエラー」および 323 ページの「pathmap コマンドを使用したスキミングの改善」を参照してください。

dbxenv 変数 macro_source (第3章「dbx のカスタマイズ」の表3-1「dbx 環境変数」 を参照) は、dbx がこの 2 つの方法のどちらを使用してマクロ定義を認識するかを制御します。

dbx で使用する方法を選択するときに、考慮すべき要因がいくつかあります。

コンパイラとコンパイラオプション

マクロ定義方法を選択する場合の 1 つの要因は、コードを構築するために使用したコンパイラやコンパイラオプションによって異なるさまざまな種類の情報が使用可能かどうかです。次の表に、コンパイラとデバッグ情報オプションに応じて選択できる方法を示します。

表 C-1 さまざまな構築オプションで使用できるマクロ定義方法

コンパイラ	-g オプション	デバッグ情報の形式	機能する方法
Oracle Solaris Studio	`-g`	DWARF	スキミング
Oracle Solaris Studio	`-g`	stabs	スキミング
Oracle Solaris Studio	`-g3`	DWARF	スキミングおよびコンパイラから
Oracle Solaris Studio	`-g3`	stabs	スキミング (`-g3` オプションと `-xdebugformat=stabs` オプションの併用はサポートされていません)
GNU	`-g`	DWARF	いずれでもなし
GNU	`-g`	stabs	該当なし
GNU	`-g3`	DWARF	コンパイラから
GNU	`-g3`	stabs	該当なし

機能におけるかね合い

マクロ定義方法を選択する場合に考慮すべきもう 1 つの要因は、どの方法を選択するかによって異なる、機能におけるかね合いです。

- **実行可能ファイルのサイズ**。スキミング方法の主な利点は、`-g` オプションでのコンパイルによって生成されたより小さい実行可能ファイルで機能するため、`-g3` オプションでコンパイルする必要がないことです。
- **デバッグの形式**。スキミングは DWARF とスタブの両方で機能します。`-g3` オプションでコンパイルしてコンパイラから定義を取得する場合は、DWARF でのみ機能します。
- **速度**。スキミングでは、`dbx` でまだデバッグ情報が読み取られていないモジュールの式が最初に評価されるときに、最大 1 秒かかります。
- **正確性**。`-g3` オプションでコンパイルするときにコンパイラによって提供される情報は、スキミングによって提供される情報よりも正確で安定しています。

- **構築環境が使用可能かどうか。**スキミングでは、デバッグ中にコンパイラ、ソースコードファイル、およびインクルードファイルが使用可能である必要があります。dbx は、これらの項目が古くなっているかどうかをチェックしないため、変更されている可能性が高い場合は、正確性が低下することがあり、スキミングに依存するよりも -g3 オプションでのコンパイルの方が適切であることがあります。
- **コードがコンパイルされたシステムとは異なるシステムでのデバッグ。**システム A でコードをコンパイルし、それをシステム B でデバッグしている場合、dbx は、pathmap コマンドをある程度利用しながら NFS を使用してシステム A 上のファイルにアクセスします。

  pathmap コマンドはまた、スキミング中のファイルアクセスを容易にするためにも役立ちます。これは、プログラムのソースファイルとインクルードファイルに対しては機能しますが、/usr/include が通常 NFS 経由では使用できないため、システムのインクルードファイルに対しては機能しない可能性があります。そのため、マクロ定義は構築システムではなく、デバッグシステム上の /usr/include から抽出されます。

  システムインクルードファイル間の不整合の可能性を認識して許容するか、-g3 オプションでコンパイルするかを選択できます。

制限事項

- Fortran コンパイラは cpp(1) 関数または fpp(1) 関数を通してマクロをサポートしますが、dbx では Fortran のマクロ展開はサポートされていません。
- dbx では、-g3 オプションおよび -xdebugformat=stabs オプションでのコンパイルによって生成されたマクロ情報は無視されます。

  スタブのインデックスの詳細については、*install-dir*/solarisstudio12.4/READMEs/stabs.pdf のパスにあるスタブのインタフェースに関するガイドを参照してください。
- スキミングは、-g オプションおよび -xdebugformat=stabs オプションでコンパイルされたコードで機能します。

スキミングエラー

コードを -g3 オプションでコンパイルせず、かつ macro_source dbxenv 変数が skim_unless_compiler または skim に設定されている場合は、マクロスキミングに依存しています。

モジュールのスキミングを正常に実行するには、次の条件が成立する必要があります。

- このモジュールは、Oracle Solaris Studio コンパイラで -g オプションを使用してコンパイルされている必要があります。
- モジュールのコンパイルに使用されたコンパイラに dbx からアクセスできることが必要です。
- このモジュールのソースファイルに、dbx からアクセスできる必要があります。
- このモジュールのソースコードによってインクルードされたファイルが使用可能である必要があります。つまり、このモジュールがコンパイルされたときに -I オプションに指定されたパスに、dbx からアクセスできる必要があります。
- ソースコードは字句的に正常でなければなりません。たとえば、終了していないコメント文字列が含まれていたり、#endif が欠けていたりしてはいけません。

ソースコードやインクルードファイルに dbx からアクセスできない場合は、pathmap コマンドを使用してそれらをアクセス可能にすることができます。

pathmap コマンドを使用したスキミングの改善

コンパイル後にソースファイルを移動した場合、あるマシンで構築したあとに別のマシンでデバッグする場合、または 84 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」で説明されているその他の状況のいずれかに当てはまる場合は、マクロスキミングでスキミング対象のファイル内にインクルードファイルが見つからないことがあります。この解決方法として、ファイルが見つからないその他の場合と同様に、pathmap コマンドを使用してマクロスキマーでインクルードディレクトリを見つけやすくします。たとえば、オプション -I/export/home/proj1/include でコンパイルし、コードに文 #include "module1/api.h" があると仮定します。その後、proj1 から proj2 に名前を変更した場合、次の pathmap コマンドにより、マクロスキマーはファイルを見つけやすくなります。

```
pathmap /export/home/proj1 /export/home/proj2
```

パスマップは、元のコードのコンパイルに使用されたコンパイラには適用されません。

マクロの作業以外では、ファイルが見つからないときに pathmap コマンドを使用してパスマップを変更すると変更はただちに有効になりますが、マクロの作業では、パスマップを有効にするにはアプリケーションを再度読み込む必要があります。

あるマシンで構築し別のマシンでデバッグする場合に、pathmap コマンドによって dbx は正しいファイルを見つけやすくなります。ただし、/usr/include/stdio.h などのシステムインクルード

ファイルは通常は構築マシンからエクスポートされないため、マクロスキマーではデバッグマシン上のファイルが使用される可能性があります。場合によっては、デバッグマシン上でステムインクルードファイルを使用できないことがあります。また、システム固有のマクロやシステムに依存するマクロの値が、デバッグマシン上と構築マシン上とでは同じでない可能性もあります。

pathmap コマンドでスキミングの問題が解決されない場合は、コードを -g3 オプションでコンパイルし、macro_source dbxenv 変数を skim_unless_compiler または compiler に設定することを検討してください。

♦ ♦ ♦ 付 録 D

コマンドリファレンス

この付録では、すべての 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*

ここでは:

expression は、*variable* に代入される値です。

Java モードの構文

assign *identifier* = *expression*

ここでは:

expression は、次のいずれかを含むことができる有効な Java の式です。

- *class-name* は、Java クラスの名前です。次のいずれかを使用できます。
  - ピリオド (.) を修飾子として使用したパッケージのパス (test1.extra.T1.Inner など)

- シャープ記号 (#) が前に付き、スラッシュ (/) とドル記号 ($) を修飾子として使用したフルパス名。たとえば `#test1/extra/T1$Inner` などです。`$` 修飾子を使用する場合は、*class-name* を引用符で囲みます。
- *field-name* は、クラス内のフィールド名です。
- *identifier* は `this` を含む局所変数またはパラメータで、現在のクラスインスタンス変数 (*object-name.field-name*) またはクラス (静的) 変数 (*class-name.field-name*) です。
- *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-name* 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 *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。
`attach -r ...`	`-r` オプションを使用すると、`dbx` は、`watch` コマンド、`display` コマンド、`when` コマンド、および `stop` 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` コマンドと `undisplay 0` コマンドが暗黙に実行されます。

ここでは:

process-ID は実行中のプロセスのプロセス ID です。

program-name は、実行中のプログラムのパス名です。

dbx を実行中の Java プロセスに接続する方法については、244 ページの「動作中の Java アプリケーションへの dbx の接続」を参照してください。

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*	現在のファイル内で、*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。

ここでは:

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。

call コマンドを使用して、関数を呼び出すこともできます。戻り値を表示するには、print コマンドを使用します。

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 コマンドを使用して続行するか、pop -c を使用して呼び出しを中止できます。後者のメソッドは、呼び出された関数がセグメント例外を引き起こした場合にも便利です。

ネイティブモードの構文

`call` *procedure* `([`*parameters*`]) [-lang` *language*`] [-resumeone] [-m] [+m]`

ここでは:

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。

-lang は呼び出す手続きの言語を指定し、指定した言語の呼び出し規則を使用するように 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 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、186 ページの「実行の再開」を参照してください。

-m は、dbxenv 変数 macro_expand が off に設定されている場合に、手続きとパラメータにマクロ展開を適用するように指定します。

+m は、dbxenv 変数 macro_expand が on に設定されている場合に、マクロ展開をスキップするように指定します。

Java モードの構文

call [*class-name*.|*object-name*.] *method-name* ([*parameters*])

ここでは:

class-name は、Java クラスの名前です。次のいずれかを使用できます。

- ピリオド (.) を修飾子として使用したパッケージのパス (test1.extra.T1.Inner など)
- シャープ記号 (#) が前に付き、スラッシュ (/) とドル記号 ($) を修飾子として使用したフルパス名。たとえば #test1/extra/T1$Inner などです。$ 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。

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 (431 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。

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` *number* *number* ...	番号が *number* のシグナルを捕獲します。
`catch` *signal* *signal* ...	*signal* によって名前を付けられた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。

ここでは:

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) の現在のステータスを出力します。ネイティブモードでだけ有効です。

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

このセクションでは、check コマンドのオプションに関する情報を提供します。

check [*functions*] [*files*] [*loadobjects*]

functions、*files*、*loadobjects* における check -all、suppress all、unsuppress all と同じです。

ここでは:

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 つ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。

注記 - RTC ですべてのエラーを検出するには、-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) 情報が必要となることがあります。このため、特定のエラー (a.out の rui と共有ライブラリの rui + 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。

-access オプション

-access オプションはチェックを有効にします。RTC は、次のエラーを報告します。

baf	不正な領域解放
duf	重複領域解放

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
rob	配列の範囲外のメモリーからの読み取り
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
wob	配列の範囲外のメモリーへの書き込み
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

デフォルトの動作は、各アクセスエラーが検出されるとプロセスを停止することです。これを変更するには、rtc_auto_continue dbxenv 変数を使用します。on に設定すると、アクセスエラーがファイルに記録されます。ログファイル名は、dbxenv 変数 rtc_error_log_file_name で制御されます。

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。この動作は、dbxenv 変数 rtc_auto_suppress を使用して変更できます。この変数のデフォルト設定は on です。

-leaks オプション

leaks オプションの構文:

```
check -leaks [-frames n] [-match m]
```

リーク検査を有効にします。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の真ん中を指しています。
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にのみ存在します
mel	メモリーリーク – ブロックへのポインタが存在しません。

リーク検査を有効にすると、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。デフォルトでは、dbxenv 変数 rtc_mel_at_exit から変更可能な簡易レポートが生成されます。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 (400 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、複数のリークを組み合わせるために使用します。複数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 8 です。

-memuse オプション

-memuse オプションの構文:

```
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、使用中ブロックのレポート (biu) も有効にします。デフォルトでは、dbxenv 変数 rtc_biu_at_exit から変更可能な使用中ブロックの簡易レポートが生成されます。プログラムの実行中にいつでも、プログラム内のメモリーがどこに割り当てられているかを確認できます (401 ページの「showmemuse コマンド」を参照)。

-frames *n* は、メモリーの使用およびリークが報告されるとき、*n* 個までのスタックフレームが個別に表示されることを示します。-match *m* を使用して、これらのレポートを組み合わせます。複数のリークの割り当て時に、呼び出しスタックが *m* 個のフレームに一致した場合、これらのリークは、単一の結合されたメモリーリークレポートで報告されます。

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 8 です。

-all オプション

-all オプションの構文:

check -all [-frames *n*] [-match *m*]

次と同等です。

check -access and check -memuse [-frames *n*] [-match *m*]

dbxenv 変数 rtc_biu_at_exit の値は check -all によって変更されないので、デフォルトで、終了時にメモリー使用状況レポートは生成されません。343 ページの「dbx コマンド」環境変数については、dbx Command を参照してください。

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。

inclass 引数、inmethod 引数、infile 引数、または infunction 引数を付けた stop コマンド、trace コマンド、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いず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。この方法でクリアすると、特定のセットに属する個々のブレークポイントを再び有効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無効にしたあと有効にすると、すべてのブレークポイントが再設定されます。

構文

clear [*filename: line*]

ここでは:

line はソースコード行の番号であるため、指定された行にあるすべてのブレークポイントがクリアされます。

filename はソースコードファイルの名前で、指定されたファイルの行 *line* にあるすべてのブレークポイントがクリアされます。

ファイルや行を指定していない場合、現在の停止点にあるすべてのブレークポイントがクリアされます。

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。

このセクションでは、コレクタコマンドを一覧表示し、それらの詳細を説明します。

構文

`collector archive` *options*	実験の終了時に実験をアーカイブするためのモードを指定します。
`collector dbxsample` *options*	dbx がターゲットプロセスを停止した場合の標本の収集を制御します。
`collector disable`	データ収集を停止し、現在の実験を閉じます。
`collector enable`	コレクタを有効にし、新しい実験を開きます。
`collector heaptrace` *options*	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。
`collector hwprofile` *options*	ハードウェアカウンタプロファイリング設定を指定します。
`collector limit` *options*	記録されるプロファイリングデータの量を制限します。
`collector pause`	パフォーマンスデータの収集を停止しますが、実験は開いたままにします。
`collector profile` *options*	呼び出しスタックプロファイリングデータの収集の設定を指定します。
`collector resume`	一時停止後、パフォーマンスデータの収集を開始します。
`collector sample` *options*	標本設定を指定します。
`collector show` *options*	現在のコレクタ設定を表示します。
`collector status`	現在の実験に関するステータスを照会します。

`collector store` *options*	実験ファイルの制御と設定。
`collector synctrace` *options*	スレッド同期待ちトレースデータの収集のための設定を指定します。
`collector tha` *options*	スレッドアナライザデータの収集の設定を指定します。
`collector version`	データを収集するために使用される `libcollector.so` のバージョンを報告します。

ここでは:

データ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。移植性のために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。この時点で標本を収集しないように指示するには、`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。

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。

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リング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 on|hi|high|lo|low|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。カウンタの解像度を `high` または `low` に設定できます。解像度を指定しない場合、標準に設定されます。これらのオプションは `collect` コマンドオプションに似ています。詳細は、`collect(1)` のマニュアルページを参照してください。
`collector hwprofile addcounter on|off`	ハードウェアカウンタオーバーフロープロファイルのためのカウンタを追加します。
`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。

ここでは:

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、使用可能なカウンタの選択は使用している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`* `| unlimited | none`

ここでは:

value はメガバイト単位であり、記録されているプロファイリングデータの量を制限し、正の数値でなければなりません。制限に到達すると、それ以上のプロファイリングデータは記録されませんが、実験は開かれたままで、標本ポイントの記録は継続されます。デフォルトでは、記録されるデータサイズに制限はありません。

制限を設定した場合、制限を削除するには `unlimited` または `none` 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、標本ポイントは記録されません。標本は一時停止の前に生成され、再開直後に別の標本が生成されます。データ収集は `collector resume` コマンドによって再開できます。

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 (338 ページの「`collector pause` コマンド」参照)。

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。

ここでは:

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	すべての設定を表示します
collector show all	すべての設定を表示します
collector show archive	アーカイブ設定を表示します
collector show profile	呼び出しスタックプロファイリング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 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。作業ディレクトリと実験名を返します。

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* `| -filename` *filename* `| -group` *string*`}`

ここでは:

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 {*microseconds*| calibrate}	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。calibrate が指定された場合、しきい値が自動的に計算されます。

ここでは:

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

collector tha コマンドは、しきい値アナライザデータの収集のオプションを指定します。

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。マルチスレッドプロセスでは、すべてのスレッドが再開されます。Ctrl-C を使用すると、プログラムの実行が停止します。
`cont ... -sig` *signal*	シグナル *signal* で実行を継続します。
`cont ...` *ID*	継続するスレッドまたは LWP を *id* で指定します。
`cont at` *line* `[`*ID*`]`	行 *line* 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、*ID* が必要です。
`cont ... -follow parent|child|both`	`dbx follow_fork_mode` 環境変数を `ask` に設定し、`stop` を選択した場合、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 Oracle Solaris Studio DE でのみ使用できます。

dalias コマンド

dalias コマンドは、dbx 形式 (csh 形式) の別名を定義します。ネイティブモードでだけ有効です。

構文

`dalias [`*name* `[`*definition*`]]`	(dbx alias) 現在定義されている別名をすべて一覧表示します。 名前を指定した場合、別名 *name* があれば定義が一覧表示されます。 定義も指定する場合、*name* を *definition* の別名として定義します。*definition* 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。

ここでは:

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。例:

```
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* [*core* | *process-ID*]	*program-name* をデバッグします。 *core* を指定した場合、コアファイル *core* によって *program-name* がデバッグされます。 *process-ID* を指定した場合、プロセス ID *process-ID* によって *program-name* がデバッグされます。
dbx *options* - {*process-ID*|*core*}	*process-ID* を指定した場合、プロセス ID *process-ID* がデバッグされ、dbx は、/proc を使用してプログラムを見つけます。 *core* を指定した場合、コアファイル *core* によってデバッグされます。
dbx *options* - *core*	コアファイル *core* を使用してデバッグします。

`dbx` *options* `-r` *program-name arguments*	引数 *arguments* で *program-name* を実行します。異常終了した場合、*program-name* のデバッグを開始し、そうでない場合は単に終了します。

ここでは:

program-name は、デバッグ対象プログラムの名前です。

process-ID は実行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、345 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` を使用してプログラムを見つけます。
`dbx` *options* `{ -r | -a}` *program-name*`{.class | .jar}` *arguments*	引数 *arguments* で *program-name* を実行します。異常終了した場合、*program-name* のデバッグを開始し、そうでない場合は単に終了します。

ここでは:

program-name は、デバッグ対象プログラムの名前です。

process-id は実行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、345 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

次の表に、ネイティブモードおよび Java モードの両方の dbx コマンドのオプションを一覧表示します。

-a *arguments*	プログラム引数 *arguments* を付けてプログラムをロードします。
-B	すべてのメッセージを抑止します。デバッグするプログラムの exit コードを返します。
-c *commands*	*commands* 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 (330 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 *file* を削除します。
-e	入力コマンドをエコーします。
-f	コアファイルが一致しない場合でも、それを強制的にロードします。
-h	dbx のヘルプを出力します。
-I *dir*	*dir* を pathmap セットに追加します (385 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 *file*	*file* を */current-dir*/.dbxrc または $HOME/ .dbxrc の代わりに起動ファイルとして使用します。
-S	初期設定ファイル */install-dir*/lib/dbxrc の読み取りを抑止します。
-V	dbx のバージョンを出力します。
-w *n*	where コマンドで *n* 個のフレームをスキップします。
-x exec32	64 ビット OS を実行しているシステム上でデフォルトで実行される 64 ビットの dbx バイナリの代わりに、32 ビットの 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、dbxenv 変数の表示や設定に使用します。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 モードで同じです。

構文

`dbxenv` [*environment-variable setting*]	`dbxenv` 変数の現在の設定値を表示します。`dbxenv` 変数を指定した場合、`dbxenv` 変数が *setting* に設定されます。

ここでは:

environment-variable は `dbxenv` 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。
`debug` *program-name core*	コアファイル *core* による *program* のデバッグを開始します。*program-name* には `-` を指定できます。`dbx` 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、36 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。
`debug` *program-name 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 -f ...`	コアファイルが一致しない場合でも、それを強制的にロードします。
`debug -r ...`	`-r` オプションを使用すると、`dbx` は `display`、`trace`、`when`、および `stop` 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。Oracle Solaris Studio IDE で実行してい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。Oracle Solaris Studio IDE で実行している場合にのみ有効です。
`debug [`*options*`] -- `*program-name*	*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。

ここでは:

core は、コアファイルの名前です。

options は、348 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。

`debug` コマンドでプログラムをロードすると、リーク検査とアクセス検査は無効になります。`check` コマンドを使用すれば、それらを有効にできます。

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` コマンド、`display` コマンド、`trace` コマンド、`when` コマンド、`stop` 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` コマンドと `undisplay 0` コマンド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。Oracle Solaris Studio IDE で実行してい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。Oracle Solaris Studio IDE で実行してい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。

ここでは:

options は、348 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*	*commands* 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`	入力コマンドをエコーします。
`-I` *directory_name*	*directory_name* を `pathmap` セットに追加します (385 ページの「`pathmap` コマンド」を参照)。
`-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 *file*	*file* を *current_directory* /.dbxrc または $HOME/.dbxrc の代わりに起動ファイルとして使用します。
-S	初期設定ファイル /*install-dir*/lib/dbxrc の読み取りを抑止します。
-V	dbx のバージョンを出力します。
-w *n*	where コマンドで *n* 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 モードで同じです。

構文

delete [-h] *handler-ID* ...	指定された *handler-ID* の trace コマンド、when コマンド、または stop コマンドを削除します。非表示のハンドラを削除するには、-h オプションを含める必要があります。
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。

ここでは:

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` [`-sig` *signal* | `-stop`]

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。

`-sig` オプションを指定した場合、指定の *signal* の転送中に切り離します。

`-stop` オプションを指定した場合、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、排他的アクセスによってブロックされる可能性のあるほかの `/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。

ここでは:

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 ]` *address* `[/`*count*`]`

アドレス *address* を始点とし、*count* 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`

+ の値から始まる 10 命令を逆アセンブルします。

ここでは:

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 コマンドで共有されます。

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。

dbx を Sun Studio 12 リリース、Sun Studio 12 Update 1 リリース、Oracle Solaris Studio 12.2 リリース、または以降の更新済みリリースの IDE または dbxtool で実行している場合、display *expression* コマンドは事実上 watch $(which *expression*) コマンドのように動作します。

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*, ...　式 *expression*, ... の値を、すべての停止ポイントで表示します。*expression* 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。

`display [-r|+r|-d|+d|-S|+S|-p|+p|-L|-f`*format*`|-F`*format*`|-m|+m|--]` *expression*`, ...`	フラグの意味については、388 ページの「`print` コマンド」を参照してください。

ここでは:

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、388 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|-f`*format*`|-F`*format*`|-F`*format*`|--]` *expression*`|`*identifier*`, ...`	フラグの意味については、388 ページの「`print` コマンド」を参照してください。

ここでは:

class-name は、Java クラスの名前です。次のいずれかを使用できます。

- ピリオド (`.`) を修飾子として使用したパッケージのパス (`test1.extra.T1.Inner` など)
- シャープ記号 (`#`) が前に付き、スラッシュ (`/`) とドル記号 (`$`) を修飾子として使用したフルパス名。たとえば `#test1/extra/T1$Inner` などです。`$` 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。有効な形式については、388 ページの「`print` コマンド」を参照してください。

identifier は this を含む局所変数またはパラメータで、現在のクラスインスタンス変数 (*object-name.field-name*) またはクラス (静的) 変数 (*class-name.field-name*) です。

object-name は、Java オブジェクトの名前です。

down コマンド

down コマンドは、呼び出しスタックを下方向に移動します (main から遠ざかる)。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 モードで同じです。

構文

down	呼び出しスタックを 1 レベル下方向に移動します。
down *number*	呼び出しスタックを *number* レベルだけ下方向に移動します。
down -h [*number*]	呼び出しスタックを下方向に移動しますが、非表示フレームをスキップしません。

ここでは:

number は、呼び出しスタックレベルの数です。

dump コマンド

dump コマンドは、手続きの局所変数すべてを出力します。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 モードで同じです。

構文

dump [*procedure*]	現在の手続きの局所変数すべてを出力します。 手続きを指定した場合、*procedure* の局所変数がすべて出力されます。

ここでは:

procedure は、手続きの名前です。

edit コマンド

edit コマンドは、ソースファイルに対して $EDITOR を起動します。ネイティブモードでだけ有効です。

dbx が Oracle Solaris Studio IDE で動作していない場合、edit コマンドは $EDITOR を使用します。そうでない場合は、該当するファイルを表示することを指示するメッセージを IDE に送信します。

構文

edit [*filename* | *procedure*]	現在のファイルを編集します。 ファイル名を指定した場合、指定したファイル *filename* を編集します。 手続きを指定した場合、関数または手続き *procedure* を含むファイルを編集します。

ここでは:

filename 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。

x コマンドは examine コマンドの別名です。

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。 前に表示された最後のアドレスの直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。

ここでは:

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。

`o,O`	8 進数 (2 または 4 バイト)
`x,X`	16 進数 (2 または 4 バイト)
`b`	8 進数 (1 バイト)
`c`	文字
`w`	ワイド文字
`s`	文字列
`W`	ワイド文字列
`f`	16 進浮動小数点数 (4 バイト、6 桁の精度)
`F`	16 進浮動小数点数 (8 バイト、14 桁の精度)
`g`	`F` と同じ

E	16 進浮動小数点数 (16 バイト、14 桁の精度)
ld,lD	10 進数 (4 バイト、D と同じ)
lo,lO	8 進数 (4 バイト、O と同じ)
lx,lX	16 進数 (4 バイト、X と同じ)
Ld,LD	10 進数 (8 バイト)
Lo,LO	8 進数 (8 バイト)
Lx,LX	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。

ここでは:

-d は動的例外の表示を有効にします。

+d は動的例外の表示を無効にします。

exists コマンド

exists コマンドは、シンボル名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。

ここでは:

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 モードで同じです。

構文

file*filename*	現在のファイルの名前を出力します。 ファイル名を指定した場合、現在のファイルを変更します。

ここでは:

filename 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 ファイルと同じディレクトリにない場合、$JAVASRCPATH 環境変数を設定しない限り dbx はそれらを発見できない場合があります。詳細については、246 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。

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。

ここでは:

regular-expression は正規表現です。

例:

```
(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一覧表示します。

`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。

構文

`fix` [*file-name file-name ...*] `[-options]`	現在のファイルを修正します。 ファイル名が一覧表示されたら、リスト内のファイルを修正します。

ここでは:

`-options` は次の有効なオプションです。

`-f`	ソースが変更されていない場合にも、ファイルの修正を強制します。
`-a`	手が加えられたファイルすべてを修正します。
`-g`	`-O` フラグを取り除き、`-g` フラグを追加します。

-c	コンパイル行を出力します (dbx による使用を目的として内部的に追加されたオプションの一部が含まれることがあります)。
-n	compile/link コマンドを実行しません (-v を付けて使用)。
v	冗長モード (dbx fix_verbose 環境変数の設定をオーバーライドします)。
+v	簡易モード (dbx fix_verbose 環境変数の設定をオーバーライドします)。

fixed コマンド

fixed コマンドは、固定ファイルすべての名前を一覧表示します。ネイティブモードでだけ有効です。

fortran_module コマンド

fortran_modules コマンドは現在のプログラムの Fortran モジュール、または、あるモジュール内の関数または変数を一覧表示します。

構文

fortran_modules [-f *module-name* | -v *module-name*]	現在のプログラムの、すべての Fortran モジュールを一覧表示します。 -m オプションを指定した場合、指定したモジュールのすべての関数が一覧表示されます。 -m オプションを指定した場合、指定したモジュールのすべての変数が一覧表示されます。

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

`frame`	現在のフレームのフレーム番号を表示します。
`frame [-h]` *number*	現在のフレームとしてフレーム *number* を設定します。
`frame [-h] +[`*number*`]`	*number* 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。
`frame [-h] -[`*number*`]`	*number* 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。
`-h`	フレームが隠されている場合でもフレームに進みます。

ここでは:

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

`func [`*procedure*`]`	現在の関数の名前を出力します。 手続きを指定した場合、現在の関数を関数または手続き *procedure* に変更します。

ここでは:

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。

ここでは:

class-name は、Java クラスの名前です。次のいずれかを使用できます。

- ピリオド (.) を修飾子として使用したパッケージのパス (`test1.extra.T1.Inner` など)
- シャープ記号 (#) が前に付き、スラッシュ (/) とドル記号 ($) を修飾子として使用したフルパス名。たとえば `#test1/extra/T1$Inner` などです。`$` 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 [-f` *filename*`] [-g] [`*regular-expression*`]`	現在のプログラム内の関数すべてを一覧表示します。 `-f` *filename* を指定した場合、ファイル内のすべての関数が一覧表示されます。`-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。*filename* が `.o` で終わる場合、コンパイラによって自動的に生成された関数を含むすべての関数が一覧表示されます。そうでない場合、ソースコードにある関数のみが一覧表示されます。 *regular-expression* を指定した場合、この正規表現に一致する関数すべてが一覧表示されます。

ここでは:

filename 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。

例:

```
(dbx) funcs [vs]print
"libc.so.1"isprint
"libc.so.1"wsprintf
"libc.so.1"sprintf
"libc.so.1"vprintf
"libc.so.1"vsprintf
```

gdb 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`

gbd on を使用すると、dbx が GDB コマンドを理解し受け付ける GDB コマンドモードになります。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。dbx コマンドは GDB コマンドモードでは受け付けられず、GDB コマンドは dbx モードでは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。

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。これらの各コマンドは、*ハンドラ ID* (*handler-ID*) と呼ばれる番号を返します。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 | -disable] *handler-ID* ...	特定のハンドラを有効または無効にし、全ハンドラを示す all として *handler-ID* を指定します。*handler-ID* 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無効となります。
handler -count *handler-ID new-limit*	特定のハンドラのトリップカウンタの値を出力します。 新しい制限パラメータを指定した場合、指定のイベントの新しいカウント制限が設定されます。
handler -reset *handler-ID*	特定のハンドラのトリップカウンタをリセットします。

ここでは:

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` *regular-expression*
: 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。
正規表現を指定した場合、*regular-expression* に一致するスタックフレームが非表示になります。正規表現は関数名またはロードオブジェクトの名前に一致し、`sh` または `ksh` ファイル照合形式の正規表現です。

ここでは:

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ように `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 [`*number* `... | `*signal* `...]`
: 無視するシグナルのリストを出力します。
シグナル番号を指定した場合、*number* の番号のシグナルが無視されます。
シグナルを指定した場合、*signal* で指定されたシグナルが無視されます。`SIGKILL` を捕獲したり無視したりすることはできません。

ここでは:

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 モードで同じです。

構文

`import` *path-name*　　`dbx` コマンドライブラリ *path-name* からコマンドをインポートします。

ここでは:

path-name 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 ...]`　　*excluded-typename* の送出を除外リストに追加します。

`intercept -a[ll] -x` *excluded-typename* `[, excluded-typename...]`　　*excluded-typename* 以外のすべての型を阻止リストに追加します。

`intercept -s[et]` `[`*intercepted-typename* `[, intercepted-typename ...]] [-x` *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。

[, *excluded-typename*]]

`intercept` 阻止対象の型を一覧表示します。

ここでは:

included-typename および *excluded-typename* は、`List <int>` や `unsigned short` などの例外型指定です。

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*

ここでは:

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、コマンド発行時に `dbx` に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 [-a]` `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。

`-a` オプションを指定した場合、システムクラスとその他の既知の Java クラスが出力されます。

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。

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。

jpkgs コマンド

jpkgs コマンドは、コマンド発行時に dbx に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。

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 [*signal*]*job* ...	一覧表示されているジョブに SIGTERM シグナルを送ります。 -signal オプションを指定した場合、特定のシグナルが一覧表示されたジョブに送信されます。

ここでは:

job はプロセス ID を指定するか、または次のいずれかの方法で指定できます。

%+	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。

ここでは:

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 環境変数によって設定された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。

ここでは:

language は c、c++、fortran、または fortran90 です。

注記 - 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 モードで同じです。

構文

`line [["`*file-name*`":] [`*number*`]]`	現在の行番号を表示します。 番号を指定した場合、現在の行番号が *number* に設定されます。 ファイル名を指定した場合、現在の行番号が *filename* の 1 行目に設定されます。 両方を指定した場合、現在の行番号が *file-name* の *number* 行目に設定されます。

ここでは:

filename は、行番号を変更するファイルの名前です。ファイル名を囲む "" 引用符はオプションです。それらはファイル名にスペースが含まれている場合に便利です。

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 モードで同じです。

デフォルトの表示行数 N は、dbx output_list_size 環境変数によって制御されます。

構文

`list`	N 行を一覧表示します。
`list` *number*	行番号 *number* を表示します。
`list +`	次の N 行を一覧表示します。
`list +`*n*	次の *n* 行を一覧表示します。

`list -`	直前の N 行を一覧表示します。
`list -`*n*	直前の *n* 行を一覧表示します。
`list` *n1*`,` *n2*	*n1* から *n2* までの行を一覧表示します。
`list` *n1*`, +`	*n1* から *n1* +N までを一覧表示します。
`list` *n1*`, +`*n2*	*n1* から *n1* + *n2* までを一覧表示します。
`list` *n1*`, -`	*n1*-N から *n1.* までを一覧表示します。
`list` *n1*`, -`*n2*	*n1* - *n2* から *n1* までを一覧表示します。
`list` *function*	*function* のソースの先頭を一覧表示します。`list` *function* は、現在のスコープを変更します。詳細については、68 ページの「プログラムスコープ」を参照してください。
`list` *filename*	ファイル *filename* の先頭を一覧表示します。
`list` *filename*`:`*n*	ファイル *filename* を *n* 行目から一覧表示します。

ここでは:

filename は、ソースコードファイルのファイル名です。

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。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。コンマはオプションです。

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 `-w`*n*	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。このオプションはプラス記号 (+) またはマイナス記号 (-) 構文と併用したり、2 つの行番号が指定されている場合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                    // list N lines starting at current line
list +5                    // list next 5 lines starting at current line
list -                  // list previous N lines
list -20                // list previous 20 lines
list 1000               // list line 1000
list 1000,$             // list from line 1000 to last line
list 2737 +24           // list line 2737 and next 24 lines
list 1000 -20           // list line 980 to 1000
list test.cc:33         // list source line 33 in file test.cc
list -w                 // list N lines around current line
list -w8 "test.cc"func1 // list 8 lines around function func1
list -i 500 +10              // list source and assembly code for line
               500 to line 510
```

listi 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。このコマンドは list -i を使用するのと同じです。

詳細については、369 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。

このセクションでは、loadobject オプションを一覧表示し、それらの詳細を説明します。

構文

loadobject -list [*regexp*] [-a]　現在ロードされたロードオブジェクトを表示します。

`loadobject -load` *loadobject*	指定したロードオブジェクトのシンボルをロードします。
`loadobject -unload` [*regexp*]	指定したロードオブジェクトをアンロードします。
`loadobject -hide` [*regexp*]	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。
`loadobject -use` [*regexp*]	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。
`loadobject -dumpelf` [*regexp*]	ロードオブジェクトのさまざまな ELF 詳細を表示します。
`loadobject -exclude` *ex-regexp*	*ex-regexp* に一致するロードオブジェクトを自動的にロードしません。
`loadobject exclude -clear`	パターンの除外リストをクリアします。

ここでは:

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。

ex-regexp はオプションではなく、指定する必要があります。

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` [*regexp*]

ここでは:

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、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、Oracle 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、`dbx` に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ロードしないように指示します。

構文

`loadobject -exclude` *ex-regexp* `[-clear]`

ここでは:

ex-regexp 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` サブコマンドでの *regexp* とは異なり、*ex-regexp* を指定しない場合は、デフォルトがすべて (all) に設定されません。*ex-regexp* を指定しない場合、このコマンドは前の `loadobject -exclude` コマンドで指定された除外パターンを一覧表示します。

`-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 ロードし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]
```

ここでは:

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。詳細については、dbx プロンプトに次を入力して、dbx ヘルプファイルを参照してください。

```
(dbx) help loadobject preloading
```

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]
```

ここでは:

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、余白部分にはステータス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「マップされていない」を意味します。
p	この文字は、プリロードされたロードオブジェクト、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。

例:

```
(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 376 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 -load *load-object*

ここでは:

load-object には、フルパス名または /usr/lib、/usr/lib/sparcv9、または /usr/lib/amd64 内のライブラリを指定します。プログラムがデバッグ中の場合は、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 [*regexp*]

ここでは:

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、コマンド行に指定された *regexp* に一致するすべてのロードオブジェクトのシンボルをアンロードします。debug コマンドでロードした主プログラムはアンロードできません。また、dbx は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アンロードも拒否する場合があります。

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 [regexp]
```

ここでは:

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。

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。

注記 - lwp コマンドは Oracle 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` *lwp-ID*	LWP *lwp-ID* に切り替えます。
`lwp -info`	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。
`lwp` [*lwp-ID*] `-setfp` *address-expression*	`fp` レジスタに *address_expression* の値が入っていることを `dbx` に伝えます。デバッグ中のプログラムの状態は変更されません。`-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。
`lwp` [*lwp-ID*] `-resetfp`	前の `lwp -setfp` コマンドの効果を元に戻して、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。

ここでは:

lwp-ID は軽量プロセスの識別子です。

コマンドに LWP ID とオプションの両方が使用された場合、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 レジスタへの変更を、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。

注記 - `lwps` コマンドは Oracle Solaris プラットフォームでのみ利用可能です。

macro コマンド

macro コマンドは、式のマクロ展開を出力します。

構文

macro *expression*, ...

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。

Oracle 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 *mmapped-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。

ここでは:

mmapped-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、*mmapped-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。

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。例:

```
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;
```

次のコマンドは、メモリーとしてデバッガーからデータベースへのアクセスを可能にします。

```
mmapfile ../DATABASE $[vaddr] $[offset] $[size]
```

データベースの内容を構造的に表示するには:

```
print *index
```

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* `| -a}`	*name* を指定した場合、*name* というモジュールのデバッグ情報を読み取ります。-a を指定した場合、すべてのモジュールのデバッグ情報を読み取ります。

ここでは:

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 はファイルが実行可能ファイルより新しい場合でも、デバッグ情報を強制的に読み取ります。このオプションは慎重に使用してください。

-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。

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

modules [-v] [-debug |-read]	すべてのモジュールを一覧表示します。 -debug を指定した場合、デバッグ情報を含むすべてのモジュールが一覧表示されます。 -read を指定した場合、すでに読み取られたデバッグ情報を含むモジュールの名前が一覧表示されます。

ここでは:

-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*

ここでは:

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 step_events 環境変数 (56 ページの「dbxenv 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を有効にするかどうかを制御します。

ネイティブモードの構文

next	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。
next *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)。
next ... -sig *signal*	ステップ実行中に指定したシグナルを引き渡します。
next ... *thread-ID*	指定した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。

ここでは:

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、汎用 next コマンドのデッドロック回避策は無効となります。

マシンレベルの呼び出しステップオーバーについては、382 ページの「nexti コマンド」も参照してください。

注記 - 軽量プロセス (LWP) の詳細については、Oracle 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

`next`	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。
`next` *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)。
`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。

ここでは:

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 *lwp-ID* が指定されている場合、汎用 `next` コマンドのデッドロック回避策は無効となります。

注記 - 軽量プロセス (LWP) の詳細については、Oracle 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。

ここでは:

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。

omp_loop コマンド

`omp_loop` コマンドは現在のループに関する説明を出力します。これには、スケジューリング (静的、動的、ガイド付き、自動、または実行時)、番号付きまたは番号なし、範囲、ステップ数または刻み幅、および繰り返し回数が含まれます。このコマンドは、ループを現在実行中のスレッドからしか発行できません。

omp_pr コマンド

`omp_pr` コマンドは、現在の並列領域または指定された並列領域に関する説明を出力します。これには、親領域、並列領域の ID、チームのサイズ (スレッド数)、およびプログラムの場所 (プログラムのカウンタアドレス) が含まれます。

構文

`omp_pr`	現在の並列領域の説明を出力します。
`omp_pr` *parallel-region-ID*	指定された並列領域の説明を出力します。このコマンドを実行しても、dbx は現在の並列領域を指定の並列領域に切り替えません。
`omp_pr -ancestors`	現在の並列領域から、現在の並列領域ツリーのルートに至るまで、パス上のすべての並列領域の説明を出力します。
`omp_pr` *parallel-region-ID* `-ancestors`	指定された並列領域から、そのルートに至るまで、パス上のすべての並列領域の説明を出力します。
`omp_pr -tree`	並列領域ツリー全体の説明を出力します。
`omp_pr -v`	チームメンバー情報とともに、現在の並列領域の説明を出力します。

omp_serialize コマンド

`omp_serialize` コマンドは、現在のスレッド、または現在のチームのすべてのスレッドで、次に検出された並列領域の実行を直列化します。直列化は、並列領域への 1 回限りのトリップに対してのみ適用され、持続はしません。

このコマンドを使用するときは、プログラム内での位置が正しいことを確認してください。論理的な位置とは、並列指令の直前です。

構文

`omp_serialize [-team]`	現在のスレッドで、次に検出された並列領域の実行を直列化します。 `-team` を指定した場合、現在のチームのすべてのスレッドに対してこれが実行されます。

omp_team コマンド

`omp_team` コマンドは、現在のチームのすべてのスレッドを出力します。

構文

omp_team [*parallel-region-ID*]	現在のチームのすべてのスレッドを出力します。 並列領域 ID を指定すると、指定された並列領域のチームのすべてのスレッドが出力されます。

omp_tr コマンド

omp_tr コマンドは、現在のタスク領域に関する説明を出力します。これには、タスク領域 ID、型 (暗黙的または明示的)、状態 (生成済み、実行中、または待機中)、実行中のスレッド、プログラムの場所 (プログラムカウンタアドレス)、未完了の子、および親が含まれます。

構文

omp_tr	現在のタスク領域の説明を出力します。
omp_tr *task-region-ID*	指定されたタスク領域の説明を出力します。このコマンドが実行されても、dbx は、現在のタスク領域を指定されたタスク領域に切り替えません。
omp_tr -ancestors	現在のタスク領域から、現在のタスク領域ツリーのルートに至るまで、パス上のすべてのタスク領域の説明を出力します。
omp_tr *task-region-ID* -ancestors	指定されたタスク領域から、そのルートに至るまで、パス上のすべてのタスク領域の説明を出力します。
omp_tr -tree	タスク領域ツリー全体の説明を出力します。

pathmap コマンド

pathmap コマンドは、ソースファイルなどを検索するために、1 つのパス名を別のパス名にマップします。マッピングは、ソースパス、オブジェクトファイルパス、および現在の作業ディレクトリに適用されます (-c を指定した場合)。マクロスキミング中、ディレクトリパスを含めるためにも適用されます。pathmap コマンドの構文および機能は、ネイティブモードと Java モードで同じです。

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。自動マウントされたファイルシステム上の現在の作業ディレクトリは不正確です。オートマウンタが原因で起こる問題を修正しようとする場合は -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、dbxenv 変数 core_lo_pathmap が on に設定されているときに、コアファイルのロードオブジェクトを検索するために使用します。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。詳細については、38 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* ...	任意のマッピングをパスごとに削除します。
pathmap -d *index1* *index2* ...	任意のマッピングをインデックスごとに削除します。

ここでは:

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パス、*to* はデバッグ時におけるパスを示します。

from1 は、最初に削除するマッピングのパスです。

index2 は、最後に削除するマッピングのパスです。

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
  # maps /export/home/work1/abc/test.c to /net/mmm/export/home/work2/abc/test.c
(dbx) pathmap /export/home/newproject
  # maps /export/home/work1/abc/test.c to /export/home/newproject/test.c
(dbx) pathmap
(1) -c /tmp_mnt /
(2) /export/home/work1 /net/mmm/export/home/work2
(3) /export/home/newproject
```

pop 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、フレームにのみポップできます。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が、pop -c を使用する必要があります。

通常 pop コマンドは、ポップされたフレームに関連付けられているすべての C++ デストラクタを呼び出します。この動作は dbx pop_auto_destruct 環境変数を off に設定してオーバーライドできます。

構文

pop　　現在のトップフレームをスタックからポップします。

`pop` *number*	*number* 個のフレームをスタックからポップします。
`pop -f` *number*	現在のフレーム *number* までフレームをスタックからポップします。
`pop -c`	デバッガが行なった最後の呼び出しをポップします。

ここでは:

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

`print` *expression*`, ...`	式 *expression*, ... の値を出力します。
`print -r` *expression*	継承メンバーを含み、式 *expression* の値を出力します。
`print +r` *expression*	`dbx output_inherited_members` 環境変数が `on` に設定されている場合は、継承メンバーを出力しません。
`print -d [-r]` *expression*	式 *expression* の静的型ではなく動的型を表示します。
`print +d [-r]` *expression*	`dbx output_dynamic_type` 環境変数が `on` に設定されている場合は、式 *expression* の動的型を使用しません。
`print -s` *expression*	式の中に private 変数または thread-private 変数が含まれる場合に、現在の OpenMP の並列領域の各スレッドの式 *expression* の値を出力します。
`print -S [-r] [-d]` *expression*	静的メンバーを含み、式 *expression* の値を出力します (C++ のみ)。
`print +S [-r] [-d]` *expression*	`dbxenv show_static_members` 環境変数が `on` に設定されている場合は、静的メンバーを出力しません (C++ のみ)。

`print -p` *expression*	`prettyprint` 関数を呼び出します。
`print +p` *expression*	`dbx output_pretty_print` 環境変数が on の場合、`prettyprint` 関数を呼び出しません。
`print -L` *expression*	出力オブジェクト *expression* が 4K を超える場合は、出力を強制実行します。
`print +l` *expression*	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。
`print -l` *expression*	('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。
`print -f`*format* *expression*	整数、文字列、浮動小数点の式の形式として *format* を使用します (オンラインヘルプの format 参照)。
`print -F`*format* *expression*	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 format 参照)。
`print -o` *expression*	*expression* 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、書式文字列を使用することもできます (`-f`*format*)。非列挙式の場合、このオプションは無視されます。
`print –m` *expression*	`dbxenv` 変数 `macro_expand` が `off` に設定されている場合、*expression* にマクロ展開を適用します。
`print +m` *expression*	`dbxenv` 変数 `macro_expand` が `on` に設定されている場合、式のマクロ展開をスキップします。
`print --` *expression*	'--' は、フラグ引数の終わりを示します。これは、*expression* がプラスまたはマイナスで開始できる場合に便利です。スコープ解決の規則については、68 ページの「プログラムスコープ」を参照してください。

ここでは:

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、内蔵出力メカニズムが使用されます。

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。
- %<digits>$ 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 ::= <any character sequence not containing a %>

| %%

| <empty>

| CHARS CHARS

FLAGS ::= + | - | <space> | # | 0 | <empty>

WIDTH ::= <decimal_number> | <empty>

PREC ::= . | . <decimal_number> | <empty>

MOD ::= h | l | L | ll | <empty>

SPEC ::= d | i | o | u | x | X | f | e | E | g | G |

c | wc | s | ws | p
```

  指定の書式文字列に % 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

`print` *expression*`, ... |` *...*	式 *expression*, ... または識別子 *identifier*, ... の値を出力します。
`print -r` *expression* `|` *identifier*	継承メンバーを含み、*expression* または識別子 *identifier* の値を出力します。
`print +r` *expression* `|` *identifier*	dbx `output_inherited_members` 環境変数が `on` に設定されている場合は、継承メンバーを出力しません。
`print -d [-r]` *expression* `|` *identifier*	*expression* または *identifier* の、静的型ではなく動的型を表示します。

`print +d [-r]` *expression* | *identifier*
: `dbx output_dynamic_type` 環境変数が `on` に設定されている場合、*expression* または *identifier* の動的型は使用しないでください。

`print --` *expression* | *identifier*
: '--' は、フラグ引数の終わりを示します。これは、*expression* がプラスまたはマイナスで開始できる場合に便利です。スコープ解決の規則については、68 ページの「プログラムスコープ」を参照してください。

ここでは:

class-name は、Java クラスの名前です。次のいずれかを使用できます。

- ピリオド (.) を修飾子として使用したパッケージのパス (`test1.extra.T1.Inner` など)
- シャープ記号 (#) が前に付き、スラッシュ (/) とドル記号 ($) を修飾子として使用したフルパス名。たとえば `#test1/extra/T1$Inner` などです。`$` 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 `this` を含む局所変数またはパラメータで、現在のクラスインスタンス変数 (*object-name.field-name*) またはクラス (静的) 変数 (*class-name.field-name*) です。

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、現在のプロセスのステータスを表示します。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 {-cwd | -map | -pid}`
: `-cwd` を指定した場合、現在のプロセスの現在の作業ディレクトリが表示されます。

`-map` を指定した場合、アドレスを含むロードオブジェクトのリストが表示されます。

`-process-ID` を指定した場合、現在のプロセス ID (process-ID) が表示されます。

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 モードで同じです。

構文

prog -readsyms	dbx run_quick 環境変数を on に設定して据え置きされていたシンボリック情報を読み取ります。
prog -executable	- を使用してプログラムに接続されている場合、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、文字列を組み合わせて run コマンドで再利用できるように設計されています。

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 モードで同じです。

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。微調整するには、detach コマンドを使用します。

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。

ここでは:

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]
```

ここでは:

`-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
current thread: t@1
current frame:  [1]
g0-g3              0x00000000 0x0011d000 0x00000000 0x00000000
g4-g7              0x00000000 0x00000000 0x00000000 0x00020c38
o0-o3              0x00000003 0x00000014 0xef7562b4 0xeffff420
o4-o7              0xef752f80 0x00000003 0xeffff3d8 0x000109b8
l0-l3              0x00000014 0x0000000a 0x0000000a 0x00010a88
l4-l7              0xeffff438 0x00000001 0x00000007 0xef74df54
i0-i3              0x00000001 0xeffff4a4 0xeffff4ac 0x00020c00
i4-i7              0x00000001 0x00000000 0xeffff440 0x000108c4
y                  0x00000000
psr                0x40400086
pc                 0x000109c0:main+0x4    mov     0x5, %l0
npc                0x000109c4:main+0x8    st      %l0, [%fp - 0x8]
f0f1               +0.00000000000000e+00
f2f3               +0.00000000000000e+00
f4f5               +0.00000000000000e+00
f6f7               +0.00000000000000e+00
```

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*]	次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。最後の run コマンド、rerun コマンド、または debug コマンド

ここでは:

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、引数を付けないで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 モードで同じです。

構文

rerun	引数を付けないで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(398 ページの「save コマンド」を参照)。

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 [`*filename* `]`

ここでは:

filename は、`dbx` コマンドの実行対象ファイルの名前です。このコマンドは、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ます。

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|-S|+S|-p|+p|-L|-l|-f` *format* `| -F`*format* `| -- ]` *expression*

式の値を出力します。特殊な引用符規則は適用されないため、`rprint a > b` は、`a` の値 (存在する場合) をファイル `b` に配置します。フラグの意味については、388 ページの「`print` コマンド」を参照してください。

ここでは:

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。有効な形式については、388 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。

このコマンドは上級ユーザー向けです。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、164 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アンロードされないかぎり各 dbx セッションで永続的になります。

dbx はこのコマンドの影響を受ける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ーアクセスを追跡しないため、スキップされなかった関数では、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、このタイプのエラーは自動で抑制されません。

構文

`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。

ここでは:

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の名前へのパスです。

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。

Ctrl-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。

ここでは:

arguments はターゲットプロセスの実行に使用される引数です。

input-file は、入力のリダイレクト元のファイルの名前です。

output-file は、出力のリダイレクト先のファイルの名前です。

注記 - 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。

ここでは:

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ークポイントは、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、引数を付けないで debug コマンドを使用します。

構文

runargs *arguments*	run コマンドで使用する現在の引数を設定します (396 ページの「run コマンド」参照)。
runargs ... >| >>*file*	run コマンドで使用する出力先を設定します。
runargs ... <*file*	run コマンドで使用する入力元を設定します。
runargs	現在の引数をクリアします。

ここでは:

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*] [*filename*]	最後の run コマンド、rerun コマンド、または debug コマンドのあとのすべてのコマンドまたは *number* 個のコマンドを引いたすべてのコマンドをデフォルトのファイルまたは *filename* に保存します。

ここでは:

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。

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。ネイティブモードでだけ有効です。

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。

ここでは:

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。

実行時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。

構文

showblock -a *address*

ここでは:

address は、ヒープブロックのアドレスです。

showleaks コマンド

注記 - showleaks コマンドは Oracle 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 つのリークレコードあたり 1 行の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]

ここでは:

-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* はリークを組み合わせます。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 オプションを指定した場合、check コマンドで指定した *m* の大域値がオーバーライドされます。

-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

1 つの使用中ブロックレコードあたり 1 行のレポートが出力されます。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 コマンドのあとのリークしたすべてのブロックもレポートに含まれます。

構文

showmemuse [-a] [-m *m*] [-n *number*] [-v]

ここでは:

-a は、すべての使用中ブロックを表示します (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 8 または check コマンドで最後に指定した大域値です。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 *filename*　　ファイル *filename* 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 (ブレークポイントなど) 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	有効な trace、when、および stop ブレー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	非表示のものを含み、有効な trace、when、および stop ブレー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。

ここでは:

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、-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 step_abflow 環境変数は、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。このタイプの制御フロー変更は、siglongjmp() または longjmp() の呼び出し、あるいは例外のスロー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

`step`	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。
`step` *n*	*n* 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、現在の関数から出ます。
`step ... -sig` *signal*	ステップ実行中に指定した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。
`step ...`*thread-ID*	指定した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 ...`*lwp-ID*	指定した 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。

`step to [` *function* `]`	現在のソースコード行から呼び出された *function* へのス テップインを試行します。*function* が指定されていない場合、最後に呼び出された関数にステップインし、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()));`

ここでは:

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。

lip-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、汎用 step コマンドのデッドロック回避策は無効となります。

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に、step to コマンドを実行した場合、条件付き分岐のために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 コマンドは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、404 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 *n*	*n* 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 ...*thread-ID*	指定した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 ...*lwp-ID*	指定した 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 つの機械命令をステップ実行します (呼び出しにステップイン)。

`stepi` *n*	*n* 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` *signal*	ステップ実行し、指定したシグナルを引き渡します。
`stepi ...`*lwp-ID*	指定の LWP をステップ実行します。
`stepi ...`*thread-ID*	指定したスレッドがアクティブである LWP をステップ実行します。

ここでは:

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

このセクションでは、ネイティブモードで有効な重要な構文をいくつか説明します。追加のイベントについては、292 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 -noupdate`	実行をただちに停止しますが、Oracle Solaris 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。103 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*	*line-number* で実行を停止します。98 ページの「ソースコードの行へのブレークポイントの設定」を参照してください。
`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 true に評価された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。99 ページの「関数へのブレークポイントの設定」を参照してください。
`stop inclass` *class-name* `[ -recurse | -norecurse ]`	C++ のみ: class、struct、union、または template 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。101 ページの「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」も参照してください。
`stop infile` *filename*	*filename* 内のいずれかの関数が呼び出されたときに、実行を停止します。
`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: すべてのメンバー関数 *name* にブレークポイントを設定します。100 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` を指定した場合、基底クラスは含まれません。102 ページの「オブジェクトにブレークポイントを設定する」を参照してください。

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 *condition-expression*	*condition-expression* が true に評価された場合にだけ、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 *function* の範囲内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、永続的にするのには適していません。delete all

は、永続的なハンドラを削除しません。永続的なハンドラを削除するには、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 *mode class-namefield-name*　*class-namefield-name* 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 *line-number*　*line-number* で実行を停止します。

stop at *filename:line-number*　*filename* の *line-number* で実行を停止します。

stop change *class-namefield-name*　*class-name* で *field-name* 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 *class-name*　*class-name* がロードさ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 *class-name*　*class-name* がアンロードされたときに実行を停止します。

stop cond *condition-expression*　*condition-expression* で指定した条件が true に評価される場合に実行を停止します。

`stop in` *class-name.method-name*	*class-name.method-name* に入り、最初の行が実行されるときに、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* に入り、最初の行が実行されるときに、実行を停止します。
`stop inmethod` *class-name.method-name*	すべての非メンバーメソッド *class-name*.*method-name* にブレークポイントを設定します。
`stop inmethod` *class-name.method-name* `(`[*parameters*]`)`	すべての非メンバーメソッド *class-name*.*method-name* に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。

ここでは:

class-name は、Java クラスの名前です。次のいずれかを使用できます。

- ピリオド (`.`) を修飾子として使用したパッケージのパス (`test1.extra.T1.Inner` など)
- シャープ記号 (`#`) が前に付き、スラッシュ (`/`) とドル記号 (`$`) を修飾子として使用したフルパス名。たとえば `#test1/extra/T1$Inner` などです。`$` 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が、整数型に評価される必要があります。

field-name は、クラス内のフィールド名です。

filename 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 例外のタイプです。*type* の有効な値は、-unhandled または -unexpected です。

Java モードでは、次の修飾子が有効です。

-if *condition-expression*	*condition-expression* が true に評価された場合にだけ、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、410 ページの「stopi コマンド」を参照してください。

全イベントのリストと構文については、292 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-expression*	*address-expression* 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。

ここでは:

address-expression は、アドレスとなる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、292 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。

`dbx rtc_auto_suppress` 環境変数 が `on` に設定されている場合、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` コマンドの履歴で、`-d` オプションと `-reset` オプションを指定するものは含みません。
`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唯一の方法は、`unsuppress` コマンドを `-d` オプションを付けて使用することです。
`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*`...`	`unsuppress` コマンドで取得可能な、ID で指定された抑止解除イベントを削除します。
`suppress -r 0 | all | -all`	`unsuppress` コマンドで指定されたすべての抑止解除イベントを削除します。
`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* のリスト、*files* のリスト、*load-objects* のリストにおける *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*	場所 *address* における *errors* を抑止します。

ここでは:

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任意の組み合わせで指定できます。

all	すべてのエラー
`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`	不正な領域解放
`duf`	重複領域解放
`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
rob	配列の範囲外のメモリーからの読み取り
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
wob	配列の範囲外のメモリーへの書き込み
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 つまたは複数のロードオブジェクト名です。

エラーの抑止の詳細については、152 ページの「エラーの抑制」を参照してください。

エラーの抑止解除については、426 ページの「unsuppress コマンド」を参照してください。

sync コマンド

sync コマンドは指定した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。

注記 - sync コマンドは、Oracle Solaris プラットフォームでのみ使用できます。

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。

ここでは:

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。

注記 - syncs コマンドは、Oracle Solaris プラットフォームでのみ使用できます。

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。

次のバリエーションで、スレッド ID が指定されていない場合は、現在のスレッドが仮定されます。

`thread -info` [*thread-ID*]　　指定したスレッドに関する既知情報すべてを出力します。OpenMP スレッドの場合、この情報には OpenMP のスレッド ID、並列領域 ID、タスク領域 ID、およびスレッドの状態が含まれます。

`thread -hide` [*thread-ID*]	指定 (または現在の) スレッドを非表示にします。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) スレッドを非表示解除します。
`thread -unhide all`	すべてのスレッドを非表示解除します。
`thread -suspend` *thread-ID*	指定した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したスレッドが保持しているすべてのロックを一覧表示します。
`thread -blockedby` [*thread-ID*]	指定した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。

ここでは:

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。

次のバリエーションで、スレッド ID が指定されていない場合は、現在のスレッドが仮定されます。

`thread -info` [*thread-ID*]	指定した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) スレッドを非表示にします。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) スレッドを非表示解除します。
`thread -unhide all`	すべてのスレッドを非表示解除します。

`thread -suspend` *thread-ID*	指定した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 モニターを一覧表示します。
`thread -blockedby` [*thread-id*]	*thread-ID* がブロックされている Java モニターを一覧表示します。

ここでは:

thread-ID は `t@`*number* の `dbx` 形式のスレッド ID またはスレッドに指定された 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`	IDE 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す。

- > (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を示します (指定した LWP に現在割り当てられている)。a l@*num* は、スレッドがアクティブであることを示します (現在実行が予定されている)。
- 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
  - running
  - sleeping
  - unknown
  - wait
  - zombie

  スレッドが現在実行している関数

Java 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。
threads -mode auto|manual	IDE 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。

- > (矢印) は、現在のスレッドを示します。
- t@*number*、dbx 形式のスレッド ID

- スレッドの状態。次のいずれかになります。
  - monitor
  - running
  - sleeping
  - unknown
  - wait
  - zombie
- 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 trace_speed 環境変数 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、traceコマンドの前に native を追加します。

dbx が JNI モードで、トレースのブレークポイントを 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。

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。

`trace -file` *filename*	指定したファイル名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、*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。接続後に新たに実行するか実行を再開すると、ファイルが常に再び開きます。
`trace step`	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。
`trace next -in` *function*	指定した関数内にいる間、各ソース行をトレースします。
`trace at` *line-number*	指定のソース *line* をトレースします。
`trace in` *function*	指定した関数の呼び出しとその関数からの戻り値をトレースします。
`trace infile` *filename*	*file_name* 内の任意の関数の呼び出しとその関数からの戻り値をトレースします。
`trace inmember` *function*	*function* 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。
`trace 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。
`trace inclass` *class*	*class* のメンバー関数の呼び出しをトレースします。
`trace change` *variable*	*variable* の変更をトレースします。

ここでは:

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-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 true に評価された場合にだけ、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 *function* 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。ブレークポイントなど一部のイベントは、永続的にするのには適していません。delete all は、永続的なハンドラを削除しません。永続的なハンドラを削除するには、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。

trace -file *filename* すべてのトレース出力を指定した *filename* に送ります。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。接続後に新たに実行するか実行を再開すると、ファイルが常に再び開きます。

trace at *line-number* *line-number* をトレースします。

`trace at` *filename.line-number*	指定したソース *filename.line-number* をトレースします。
`trace in` *class-name.method-name*	*class-name. method-name* の呼び出しとその戻り値をトレースします。
`trace in` *class-name.method-name*`(`[*parameters*]`).`	*class_name.method_name*([*parameters*]) の呼び出しとそ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が呼び出されるとトレースします。
`trace inmethod` *class-name.method-name*`[(`*parameters*`)]`	*class-name.method-name* [(*parameters*)] という名前のメソッドが呼び出されるとトレースします。

ここでは:

class_name は、Java クラスの名前です。次のいずれかを使用できます。

- ピリオド (.) を修飾子として使用したパッケージのパス (`test1.extra.T1.Inner` など)
- シャープ記号 (#) が前に付き、スラッシュ (/) とドル記号 ($) を修飾子として使用したフルパス名。たとえば `#test1/extra/T1$Inner` などです。`$` 修飾子を使用する場合は、*class-name* を引用符で囲みます。

filename 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。

`-if` *condition-expression*	*condition-expression* が true に評価された場合にだけ、指定したイベントが発生し、トレースが出力されます。
`-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、トレースが出力され、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。`-temp` を `-count` 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。
`-disable`	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、292 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

`tracei step`	各マシン命令をトレースします。
`tracei next -in` *function*	指定した関数内にいる間に各命令をトレースします。
`tracei at` *address*	*address* にある命令をトレースします。
`tracei in` *function*	指定した関数の呼び出しとその関数からの戻り値をトレースします。
`tracei inmember` *function*	*function* 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*	変数の変更をトレースします。

ここでは:

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。

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、418 ページの「`trace` コマンド」を参照してください。

uncheck コマンド

`uncheck` コマンドは、メモリーのアクセス、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。

構文

`uncheck`	検査の現在のステータスを出力します。
`uncheck -access`	アクセス検査を無効にします。
`uncheck -leaks`	リーク検査を無効にします。
`uncheck -memuse`	メモリー使用状況検査を無効にします (リーク検査も無効にされます)。
`uncheck -all`	`uncheck -access`、`uncheck -memuse` と同じです。
`uncheck` [*functions*] [*files*] [*load-objects*]	*functions files load-objects* での `suppress all` と同じです。

ここでは:

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。

load-objects は、1 つまたは複数のロードオブジェクト名です。

検査の有効化については、330 ページの「check コマンド」 を参照してください。

エラーの抑止については、411 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、137 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*, ... | *n* ...}	display *expression* コマンドまたは *n*, ... 番のすべての display コマンドを取り消します。 *n* がゼロ (0) に設定されている場合、すべての display コマンドを取り消します。

ここでは:

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*, ... 番の 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。

すべての display コマンドを実行します。

ここでは:

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 this を含む局所変数またはパラメータで、現在のクラスインスタンス変数 (*object-name.field-name*) またはクラス (静的) 変数 (*class-name.field-name*) です。

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 {*regular-expression* | *number*}	スタックフレームフィルタ *regular_expression* を削除するか、スタック フレーム フィルタ番号 *number* を削除します。 *number* がゼロ (0) に設定されている場合、すべてのスタックフレーム フィルタを削除します。

ここでは:

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 コマンドは、フィルタを番号付きで一覧表示します。

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* `... ]`	タイプが *intercepted-typename* の送出を `intercept` リストから削除します。
`unintercept -a[ll]`	すべての種類の送出を `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。

ここでは:

included-typename および *excluded-typename* は、`List <int>` や `unsigned short` などの例外型指定です。

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。

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。ほかのすべてのエラーを抑止するには、suppress コマンドに `-d` オプションを付けて使用します。

`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` *load-objects*	*errors* をさらに抑止解除することによって、*load-objects* 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*] [*filename ...*] [*load-objects*]	関数のリスト、ファイルのリスト、ロードオブジェクトのリストにおける *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 "`*filenames*`"` *line*	*filenames* の *line* における *errors* を抑止解除します。
`unsuppress` *errors* `addr` *address*	場所 *address* における *errors* を抑止解除します。

ここでは:

errors は、1 つまたは複数のエラー名です。

functions は、1 個または複数の関数名です。

filenames は、1 つまたは複数のファイル名です。

line は、行番号です。

load-objects は、1 つまたは複数のロードオブジェクト名です。

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* `|` *n*`}`	`watch` *expression* コマンドまたは *n* 番の `watch` コマンドを取り消します。 *n* がゼロ (0) に設定されている場合、すべての `watch` コマンドを取り消します。

ここでは:

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 `main` に向かって上方向に移動します。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 モードで同じです。

構文

`up [-h [`*number*`]]`	呼び出しスタックを 1 レベル上方向に移動します。 *number* を指定した場合、コールスタックの *number* レベルだけ上方向に移動します。 `-h` を指定した場合、呼び出しスタックを上方向に移動しますが、非表示フレームをスキップしません。

ここでは:

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ので、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 [-r|+r|-d|+d|-S|+S|-p|+p|-L|-f*format*|-F*format*|-m|+m|--] *expression*	式 *expression* の値を、すべての停止ポイントで表示します。フラグの意味については、388 ページの「print コマンド」を参照してください。

ここでは:

expression は、有効な式です。

format は、式の出力時に使用する形式です。有効な形式については、388 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、型の宣言、またはマクロの定義を出力します。該当する場合は、OpenMP のデータ共有属性情報も出力します。

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

`whatis [-n] [-r] [-m] [+m]` *name*	型ではない *name* の宣言を出力します。または、*name* がマクロの場合は定義を出力します。
`whatis -t [-r] [-u]` *type*	型 *type* の宣言を出力します。
`whatis -e [-r] [-u] [-d]` *expression*	式 *expression* の型を出力します。

ここでは:

name は、型ではない名前またはマクロの名前です。

type は、型名です。

expression は、有効な式です。

macro は、マクロの名前です。

`-d` は、静的型ではなく動的型を表示します。

`-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` はオプションを指定せずに `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、基底クラスおよび型に関する情報を出力します。

`-t` は、型の宣言を表示します。

`-n` は、型のルート定義を表示します。

`-m` は、`dbxenv` 変数 `macro_expand` が `off` に設定されている場合でも、マクロ展開を強制します。

`+m` はマクロ検索を無効にして、マクロで隠蔽されているシンボルが代わりに見つかるようにします。

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、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。未定義のメンバー関数は一覧表示されません。

-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す。

Java モードの構文

whatis *identifier*　　*identifier* の宣言を出力します。

ここでは:

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 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、when コマンドの前に native を追加します。

dbx が JNI モードで、when ブレークポイントを 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。

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-number* に到達したら、*command* を実行します。
when in *procedure* { *command;* }	*procedure* が呼び出されたら、*command* を実行します。

ここでは:

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 *line-number* に到達したら、コマンドを実行します。
when at *filename.line-number*	*filename.line-number* に到達したら、コマンドを実行します。

when in *class-name.method-name*	*class-name.method-name* が呼び出されたら、コマンドを実行します。
when in *class-name.method-name*([*parameters*])	*class-name.method-name*([*parameters*]) が呼び出されたら、コマンドを実行します。

class-name は、Java クラスの名前です。次のいずれかを使用できます。

- ピリオド (.) を修飾子として使用したパッケージのパス (test1.extra.T1.Inner など)
- シャープ記号 (#) が前に付き、スラッシュ (/) とドル記号 ($) を修飾子として使用したフルパス名。たとえば #test1/extra/T1$Inner などです。$ 修飾子を使用する場合は、*class-name* を引用符で囲みます。

filename 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、292 ページの「イベント指定の設定」を参照してください。

指定した下位レベルイベントでコマンドを実行することについては、433 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。

構文

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* `{` *command;* `}`	*address* に到達したときに、*command* を実行します。

ここでは:

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、292 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。OpenMP のスレーブスレッドの場合、関連するフレームがアクティブ状態であれば、マスタースレッドのスタックトレースも出力されます。

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。
`where -fp` *address-expression*	`fp` レジスタに *address-expression* 値があった場合、トレースバックを出力します。
`where -h`	非表示フレームを含めます。
`where -l`	関数名を持つライブラリ名を含めます。
`where -q`	クィックトレースバック (関数名のみ)。
`where -v`	冗長トレースバック、関数の引数と行情報を含みます。

ここでは:

address-expression は、アドレスとなる式またはアドレスとして使用可能な式です。

number は、呼び出しタックフレームの数です。

これらのオプションをスレッドや LWP ID と組み合わせると、指定したエンティティーのトレースバックを取得できます。

-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。正しい値が指定されていることを確認したら、assign コマンドや lwp コマンドを使用してそれを設定できます。

Java モードの構文

where [*thread-ID*]	メソッドのトレースバックを出力します。
where -f [*thread-ID*] *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。f を指定した場合、フレーム *number* からトレースバックを開始します。
where -q [*thread-ID*]	クィックトレースバック (関数名のみ)。
where -v [*thread-ID*]	冗長トレースバック、メソッドの引数と行情報を含みます。

ここでは:

number は、呼び出しタックフレームの数です。

thread-ID は、dbx 形式のスレッド ID またはスレッドに指定した 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`	前述と同じですが、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、指定した名前のすべての使用、またはアドレスのシンボリック名を出力します。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。

ここでは:

name は、変数、関数、クラステンプレート、または関数テンプレートなどの、スコープ内のロード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、指定した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] [-m] [+m]` *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。

ここでは:

name は、変数、関数、クラステンプレート、または関数テンプレートなどの、スコープ内のロード可能オブジェクトの名前です。

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。

-t は、型の完全修飾形を表示します。

-m は、dbxenv 変数 macro_expand が off に設定されている場合でも、マクロ検索を強制します。

+m はマクロ検索を無効にして、マクロで隠蔽されているシンボルが代わりに見つかるようにします。

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)。次に実行される文が 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、およびフレーム番号を表示するものとします。
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。

ここでは:

type は、例外の型です。

索引

さ

た

は

ま

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X