情報処理学会論文誌　プログラミング　Vol.7　No.2　11–19 (June 2014)

例外処理機能を備えたシェル言語

関口 渚[1,a)]　倉光 君郎[1]

受付日 2013年11月11日, 採録日 2014年2月28日

概要：既存のシェル言語処理では，終了ステータスや trap コマンドを利用することで，エラー処理を記述することができる．しかし，これらの手法ではエラー原因を判別することが困難であるため，詳細なエラー処理が記述できないという問題があった．我々は，この問題を解決するため，例外処理機能を備えたシェル言語処理系 D-Shell を提案する．D-Shell はコマンドが発行するシステムコールを追跡する．コマンドが異常終了した際，システムコールエラーを基にエラー原因を推定し，例外として発行する機能を備える．例外に対する処理を記述することで，エラー原因に応じたエラー処理が実現できる．評価を行ったところ，エラー原因を識別し適切な例外が発行されたことを確認した．本論文では，例外機構を中心に D-Shell の設計，実装について述べる．

キーワード：シェルスクリプト，エラー処理，例外処理

A Shell Language with Exception Handling

Nagisa Sekiguchi[1,a)]　Kimio Kuramitsu[1]

Received: November 11, 2013, Accepted: February 28, 2014

Abstract: We can describe the error handling in the existing shell language, by making use of the trap command and exit status. However, there is a problem specific error handling can not be described, because it is difficult to determine the cause of the error in these technique. To address this problem, we propose *D-Shell*, a shell language with exception handling. D-Shell keeps track of the system call issued by the command. It has a function that estimates the cause of the error based on the system call error and issues it as an exception when a command failed. By describing the processing for the exception, the error processing corresponding to the cause of the error can be realized. When we evaluated D-Shell, we confirmed that the appropriate exception is issued to identify the cause of the error. In this paper, we describe the design and implementation of the D-Shell around the exception mechanism.

Keywords: shell script, error handling, exception handling

1. はじめに

シェル言語は，コマンドラインツールやファイルシステムとの親和性が高く，システムの操作を記述しやすい言語設計 [1] であり，システム運用に不可欠な存在である．

近年，シェルスクリプトの不具合・誤動作により，システム障害に至る事例 [2] が報告されている．たとえば，証券取引システムの障害では，バックアップサーバの起動スクリプトが動作し，本来，起動すべきでないエラー状態で起動されたため，機能不全を起こした事例が報告されている．

このように，運用システムはスクリプト開発者の予想どおりに処理が進まないケースがあり，コマンド実行の異常状態の検知，それに対応するエラー処理を適切に記述することがシェルスクリプト，さらにそれによって運用されるシステムの高信頼化につながると期待される．

しかしながら，既存のシェル言語処理は，十分なエラー処理機能を提供していなかった．我々は，詳細なエラー処理を実現するためのシェル言語処理系として，D-Shell を提案する．D-Shell はコマンド内部で発生したシステムコールエラーからエラー原因を推定し，それを例外として発行する機能を備える．例外に対応する処理を記述することで，詳細なエラー処理が可能になる．

[1] 横浜国立大学
Yokohama National University, Yokohama, Kanagawa 240–8501, Japan
[a)] sekiguchi-nagisa-wr@ynu.jp

なお，本論文におけるエラー処理は，エラーが発生したとき，エラーに対し処理を書くこと，と定義する．本論文におけるエラー原因は，言語処理系がエラーを検出した際のエラー発生の原因や状態の詳細情報である．より精度の高いエラー処理の決め手となる．

本論文は以下の構成になっている．2 章で既存のシェル言語におけるエラー処理の問題点を述べる．3 章で D-Shell の概要を述べる，4 章で例外機構の設計，5 章で例外機構の実装を述べる．6 章で D-Shell の評価を述べる．7 章で課題を述べる．8 章で関連研究を述べる．9 章でまとめを述べる．

2. 既存のシェル言語処理系におけるエラー処理の問題点

本章では，代表的なシェル言語処理系である，bash でエラー処理を実現する手法とそれらの問題点について述べる．

bash では，エラー処理を実現する手法が 2 つ存在する．1 つ目は終了ステータスチェック，2 つ目は trap コマンドである．どちらの手法も異常終了の際に処理を実行することができるものの，エラー原因に応じた処理を記述するのは困難である．

なお，これらの手法は bash 以外のシェル言語処理系でも用いることができる．**表 1** は，代表的なシェル言語処理系で利用できるエラー処理の手法をまとめたものである．tcsh では trap コマンドを用いることができないものの，他のシェル言語処理系では，bash と同様に前述の 2 つの手法を利用できることが分かる．

2.1 終了ステータスチェック

コマンドは実行を終了したとき，終了ステータスと呼ばれる整数型の値をシェルに返す．UNIX システムでは，コマンドの終了ステータスが 0 のときに正常終了，0 以外のときに異常終了を表す．また，コマンドによっては，終了ステータスの値でエラー原因を表すこともある．

しかし，終了ステータスが表すエラー原因は，コマンドによって粒度が大きく異なり，統一されていないという問題がある [3]．したがって，終了ステータスを利用する手法では，エラー原因に応じた処理を記述するのは困難である．

例として，rsync コマンドを用いて，別のサーバ上にファイルをバックアップするスクリプトを**図 1** にあげる．図 1 の 1 行目で syslog にバックアップの開始を通知し，2 行目で rsync コマンドを用いて，/var/www 以下のファイルをバックアップサーバに転送する．3 行目で rsync コマンドの終了ステータスを取得し，それをもとにエラー処理を行う．4 行目から 8 行目のエラー処理では，バックアップに失敗したことを syslog に通知するとともに，システム管理者に終了ステータスを通知する．

rsync コマンドでバックアップをする際，このコマンドにエラーが発生しバックアップに失敗することがある．rsync コマンドで発生するエラーの原因としては，「バックアップ元のファイルの問題」や「ネットワークの問題」，「バックアップ先のサーバの問題」があげられる．バックアップスクリプトの失敗に対処するためには，これらのエラー原因を識別し，原因に応じた詳細なエラー情報をシステム管理者に通知する処理が求められる．

しかし，エラー原因の識別が困難であるため，図 1 の 4 行目から 8 行目のように，エラーが発生したことを単に通知する処理しか記述できない．

表 1　代表的なシェル言語処理系におけるエラー処理
Table 1　Error handling in typical shell.

手法	bash	tcsh	ksh	zsh
終了ステータスチェック	○	○	○	○
trap コマンド	○	×	○	○

2.2 trap コマンド

trap コマンド [3] はシグナルハンドラを登録するコマンドである．trap コマンドは OS が発行するシグナルのほかにも，シェル言語処理系が発行する擬似シグナルも扱うことができる．コマンドが異常終了した際に発生する疑似シグナル ERR に対するハンドラを登録することで，エラー処理を記述できる．しかし，trap コマンドではエラー原因を識別できないため，以下のように簡略なエラー処理しか記述できない．

```
errorHandler() {
    ## エラー処理
    logger -ip local0.err "backup failed"
}
logger -ip local0.notice "start backup"
trap "errorHandler" ERR
rsync -av /var/www rsync://server/backup/
```

また，trap コマンドで設定したシグナルハンドラの中で終了ステータスチェックを行うこともできるが，2.1 節で述べたことと同様の問題が残る．

3. D-Shell

本章では，D-Shell の概要を述べる．D-Shell の例外機構

```
1 logger -ip local0.notice "start backup"
2 rsync -av /var/www rsync://server/backup/
3 ret=$?
4 if (( $ret != 0 )) ;then
5     ## エラー処理
6     logger -ip local0.err "backup failed"
7     echo $ret | mail -s "error" admin@ex.com
8 fi
```

図 1　bash における記述例
Fig. 1　Example of bash.

```
1  logger -ip local0.notice "start backup"
2  try {
3      rsync -av /var/www rsync://server/backup/
4  } catch(UnreachableNetworkException e) {
5      ## ネットワークに到達できない場合の処理
6      logger -ip local0.err "network problem"
7      echo $e | mail -s "error" admin1@ex.com
8  } catch(FileNotFoundException e) {
9      ## ファイルが存在しない場合の処理
10     logger -ip local0.err "disk problem"
11     echo $e | mail -s "error" admin2@ex.com
12 } catch(ConnectionRefusedException e) {
13     ## 接続先でサーバが動作していない場合の処理
14     logger -ip local0.err "server problem"
15     echo $e | mail -s "error" admin3@ex.com
16 }
```

図 2　D-Shell における記述例
Fig. 2　Example of D-Shell.

の設計については 4 章で述べる．

3.1　D-Shell の記述例

図 2 に図 1 のスクリプトを D-Shell で書き直した例を示す．D-Shell は，コマンドが異常終了した際，システムコールエラーをもとにエラー原因を推定し，それを例外として発行する機能を備える．図 2 の 2 行目から 16 行目のように Java などが備える try-catch 文を用いて例外処理（エラー処理）を記述する．3 行目の rsync コマンドでエラーが発生した際，D-Shell はエラー原因を推定しそれに応じた例外を発行する．「ネットワークに到達できない場合」に対しては，UnreachableNetworkException が発行され，「ファイルが存在しない場合」に対しては，FileNotFoundException，「接続先でサーバが動作していない場合」に対しては，ConnectionRefusedException が発行される．

例外に対する処理を catch 節に記述することで，エラー原因に応じたエラー処理を記述できる．図 2 の例では，例外に応じたメッセージを syslog に書き出すとともに，mail コマンドで例外情報をシステム管理者に送信している．D-Shell を用いることで，図 1 の例よりも詳細なエラー通知処理を実現することができる．

3.2　言語の特徴

D-Shell は Java によく似た文法および言語機能を持つ，静的型付けオブジェクト指向言語である．既存のシェル処理系と異なり，ローカルスコープを備え，関数の戻り値に整数型以外の値もとることができる．以下に D-Shell の特徴を記す．

- int 型および，float 型と四則演算，比較演算
- boolean 型と論理演算
- String 型と文字列結合演算

表 2　DShellException クラスが持つメソッド一覧
Table 2　List of methods DShellException class has.

メソッド名	取得できる情報
getCommandName	例外を発行したコマンドの名前
getErrorMessage	コマンドの標準エラー出力
getStackTrace	例外発生箇所までのスクリプトの実行過程

- ローカルスコープを備えた変数宣言
- 関数定義
- 整数型以外の値も扱える return 文
- 条件分岐文（if-else 文，switch-case 文）
- ループ文（while 文，do-while 文，for 文）
- try-catch 文
- クラス定義文，メソッド定義文

3.3　コマンドとの連携

既存のシェル言語処理系と同様に，スクリプト上にコマンドを直接記述することで，コマンドを実行することができる．また，パイプやリダイレクト，バックグラウンド実行も既存のシェル言語処理系と同様に記述できる．

D-Shell におけるコマンドの実行は，Task 型のオブジェクトを戻り値とする関数の呼び出しとして扱われる．Task 型のオブジェクトは，コマンドの終了ステータス，標準出力，標準エラー出力といった情報を保持するオブジェクトである．これらの情報にアクセスするメソッドとして，Task#getExitStatus メソッド，Task#getOutMessage メソッド，Task#getErrorMessage メソッドが用意されている．

3.4　例外オブジェクト

D-Shell では例外をオブジェクトとして扱っており，コマンドが発行する例外は，DShellException を継承したクラスとして設計されている．したがって，以下のように記述することで，コマンドから発行されたすべての種類の例外を捕捉することもできる．また，**表 2** に示すメソッドを用いて，catch 節の中で例外オブジェクトが保持する情報にアクセスすることができる．

```
try {
    rsync -av /var/www rsync://server/backup/
} catch(DShellException e) {
    ## 例外処理
}
```

3.1 節で述べた，UnreachableNetworkException や FileNotFoundException といった，システムコールエラーに基づいた例外クラスは表 2 のメソッドに加えて，以下のようなメソッドを持つ．

- getSyscallName
  エラー原因となったシステムコールの名前

- getSyscallParams
  エラー原因となったシステムコールの引数
- getErrno
  エラー原因に対応するエラーコード

3.5　パイプで結合されたコマンドの例外処理

D-Shell はコマンドの処理が終了した後に例外発行を行う．これはパイプで結合された複数のコマンドについても同様である．パイプ内の複数のコマンドは，まとめて 1 つの処理と見なすことができる．したがって，すべてのコマンドの終了を待ち，各コマンドで発行された例外をまとめて 1 つの例外（MultipleException）として扱う．

なお，パイプ内のコマンドで例外が発行された際，他のコマンドの処理を中止するという設計も考えられる．しかし，パイプ内のコマンドの失敗が処理全体の失敗に結び付くとは限らないため，このような設計は選択しなかった．

MultipleException には，getExceptions メソッドが用意されており，以下のように各コマンドで発行された例外オブジェクトにアクセスすることができる．また，例外オブジェクトの判別には，instanceof 演算子を用いる．

```
try {
    find /var/www -newer sample.html | rsync -av
        --files-from=- /var/www rsync://server/
        backup
} catch(MultipleException e) {
    for(DShellException e1 : e.getExceptions()) {
        if(e1 instanceof
            UnreachableNetworkException) {
            ## ネットワークに到達できない場合の処理
        } else if(e1 instanceof
            FileNotFoundException) {
            ## ファイルが存在しない場合の処理
        }
          .....
    }
}
```

3.6　バックグラウンド実行されたコマンドの例外処理

バックグラウンド実行されたコマンドについては，コマンドの終了を待ってから例外発行が行われる．以下のように Task#join メソッドを用いてコマンドの終了を待つ．

```
Task t = rsync -av /var/www rsync://server/
    backup/ &
try {
    t.join()
} catch(UnreachableNetworkException e) {
    ## ネットワークに到達できない場合の処理
}
  ..........
```

D-Shell では，サーバプログラムのように実行終了を待たないコマンドに対しては，例外処理の対象外としている．このようなコマンドの例外処理（エラー処理）は，運用監視ツールで実現できるため実用上の問題はない．

3.7　処理系

D-Shell は，Java で実装されており，Linux システム上で動作する．また，我々が開発しているスクリプト言語である libBun のパーサを利用して構文解析を行っている．

D-Shell および，libBun はオープンソース・ソフトウェアとして開発が進められている[*1,*2]．

4.　例外機構の設計

本章では，D-Shell の例外機構の設計について述べる．

D-Shell では，システムコールエラーによって引き起こされる異常終了に対し，その原因に応じた例外発行を行う．システムコールエラーは，システム自体の異常との関連性が高く，システムの障害対応を行う際に重要になる．したがって，システムコールエラーに基づいた例外発行は，システム運用の観点から必要性が高いといえる．

一方，テキスト処理や数値処理を行うコマンドのように，コマンドの異常終了とシステムコールエラーが無関係なものに対しては，システム自体の異常と無関係な場合が多いため，詳細な例外発行の対象外としている．また，コマンドに与える引数の間違いのようなエラーも対象外である．

これらのエラーが発生した場合でも例外発行が行われ，一律に NotRelatedSyscallException（システムコールエラーに起因しないことを表す例外）が発行される．この例外を捕捉することで，これらのエラーに対しても例外処理を行うことができるため，実用上は問題ないといえる．

4.1　エラー原因とするシステムコールエラーの特定

コマンドは複数のシステムコールエラーを引き起こすことがあるため，どのシステムコールエラーが異常終了の原因であるか判断する必要がある．複数のシステムコールエラーが発生した場合でも，それらが異常終了の原因でなければ，コマンドは処理を継続する．異常終了を引き起こすシステムコールエラーが発生した際，コマンドは標準エラー出力へメッセージを出力し，exit 関数を呼び出し終了する．標準エラー出力への出力が失敗することはほとんどないため，exit 関数を呼び出す直前に発生したシステムコールエラーが異常終了の原因であると判断できる．

4.2　システムコールエラーに基づいた例外発行

システムコールエラーが発生すると，エラーコードが返される．UNIX システムにおいて，システムコールによる操作時に発生するエラーを報告する値である．エラーコードはシステムコールエラーを一意に示している．たとえば，

*1　https://github.com/libbun/libbun
*2　https://github.com/konoha-project/dshell

エラーコード ENOENT については，「File Not Found」というシステムコールエラーを示す．

```
ENOENT → File Not Found
```

エラーコードと例外の対応表を用意することで，システムコールエラーから例外への変換を実現した．

5. 例外機構の実装

本章では，D-Shell の例外機構の実装について述べる．

図 3 に示すように，D-Shell の例外機構は 3 つの部分からなり，トレーサ，エラー原因推定機構，例外発行機構がある．トレーサは，コマンドが発行するシステムコールを追跡しログに書き出す．コマンドが異常終了した際，エラー原因推定機構がログファイルを読み込み，システムコールログから異常終了の原因を推定する．その後，例外発行機構で異常終了の原因に対応する例外を発行する．

5.1 トレーサ

トレーサは，コマンドが実行を開始してから，終了するまでに発行するシステムコールを追跡しログに書き出す．以下のような情報を取得する．

- 実行されたシステムコールの名前
- システムコールの引数と終了ステータス
- システムコールのエラーコード
- システムコールの呼び出し時のコールスタック

システムコールトレースを行うツールはいくつか存在する [4], [5], [6], [7] が，D-Shell では，strace+ コマンド [8] を用いる．strace+ コマンドは，システムコールトレースの対象となるコマンドを，自身の子プロセスとして実行する．したがって，D-Shell ではコマンドを直接実行する代わりに，strace+ コマンドを実行し，その子プロセスとしてコマンドを実行する．

strace+ コマンドで取得したシステムコールのログは**図 4** のようになる．ログにはコマンドの初期化（動的リンクライブラリのロード）中に発生するシステムコール，コマンドの本体（main 関数）の実行中に発生するシステムコール，終了処理（exit 関数の呼び出し後）中に発生するシステムコールが時系列順に記録されている．

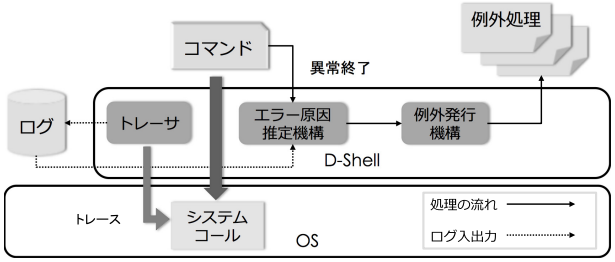


図 3　D-Shell の例外機構
Fig. 3　Exception mechanism of D-Shell.

5.2 エラー原因推定機構

エラー原因推定機構では，コマンドが異常終了した際，トレーサで取得したログをもとにエラー原因の推定を行う．ログに対してフィルタを適用することによって，exit 関数を呼び出す直前に発生したシステムコールエラーを取り出す．以下のようなフィルタを順に適用し，システムコールの抽出を行う．

(1) 初期化処理中に発生したシステムコールを取り除くフィルタ
(2) exit 関数を呼び出した後に発生したシステムコールを取り除くフィルタ
(3) システムコールエラーを取得するフィルタ
(4) gettext 関数の中で発生したシステムコールエラーを取り除くフィルタ

5.2.1 初期化処理中に発生したシステムコールの除去

5.1 節で述べたように，トレーサで取得したログには，main 関数が実行される前の初期化処理で発生するシステムコールも含まれている．初期化処理で発生するシステムコールは，コマンドの異常終了とは無関係であるため，これを取り除く．**図 5** のように，システムコール呼び出し時のコールスタック情報を用いることで，初期化処理中のシステムコールと main 関数中のシステムコールを判別することができる．

5.2.2 終了処理中に発生したシステムコールの除去

一般にプログラムは exit 関数を呼び出すことで終了す

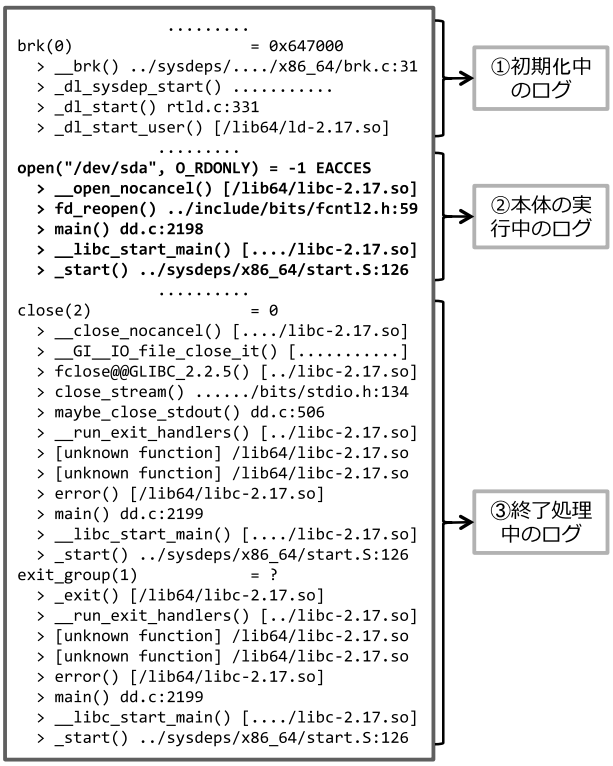


図 4　トレーサで取得したログの抜粋
Fig. 4　The extract of log obtained by tracer.

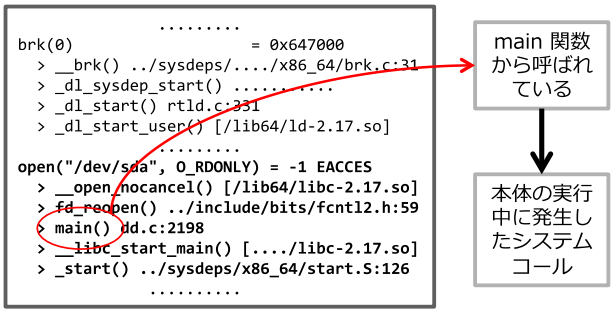


図 5　初期化処理中に発生したシステムコールの除去

Fig. 5　Removal of system calls that occurred during the initialization.

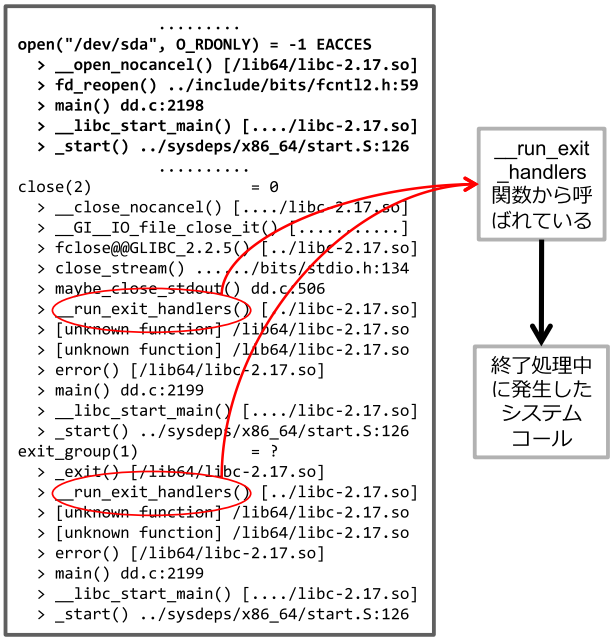


図 6　終了処理中に発生したシステムコールの除去

Fig. 6　Removal of system calls that occurred during termination.

る．exit 関数を呼び出した後，ただちに終了するのではなく，何らかの終了処理が呼ばれる．atexit 関数などを用いることで任意の終了処理を実行することもできる．したがって，終了処理中に何らかのシステムコールエラーが発生することがありうる．Linux システムにおいては，exit 関数はシステムコールではないため，単にシステムコールログを用いただけでは exit 関数が呼び出されたことが分からない．ゆえに，exit 関数が呼ばれる直前に発生したシステムコールエラーと終了処理中に発生したシステムコールエラーを区別することができない．

しかしながら，システムコールのコールスタック情報を用いることで，終了処理中に発生したシステムコールを判別することができる．図 6 のように，終了処理中に発生したシステムコールは，__run_exit_handlers 関数から呼ばれている．この関数から呼び出されたシステムコールを取り除くことで，終了処理中のシステムコールを取り除くことができる．

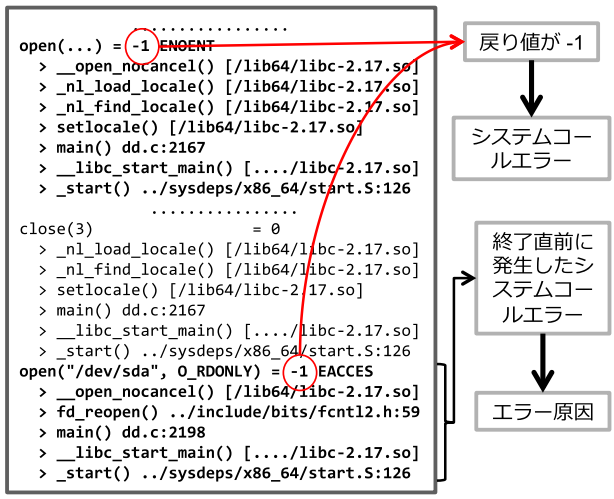


図 7　システムコールエラーの取得

Fig. 7　Get system call error.

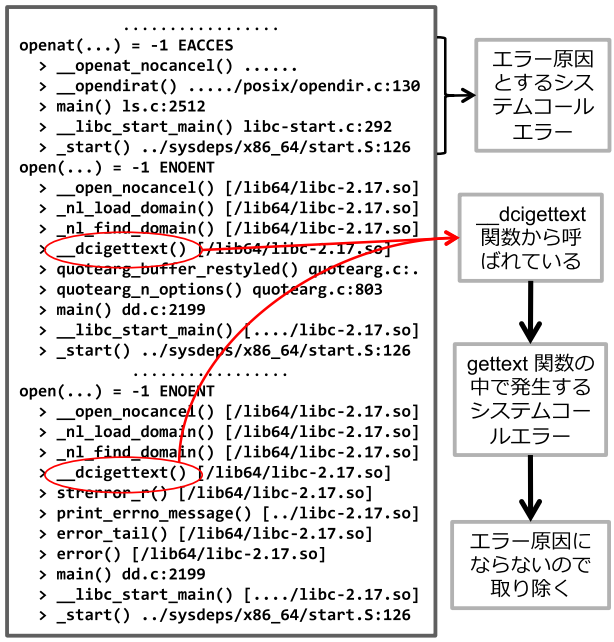


図 8　gettext 関数の中で発生したシステムコールの除去

Fig. 8　Removal of system calls that occurred in the gettext function.

5.2.3　システムコールエラーの取得

図 7 のように戻り値が −1 であるシステムコールを取り出すことで，システムコールエラーを取得できる．取得したシステムコールエラーのうち最後に発生したものが，exit 関数が呼ばれる直前に発生したシステムコールエラー，つまり，エラー原因として扱うシステムコールエラーとなる．

5.2.4　gettext 関数の中で発生したシステムコールエラーの除去

フィルタ (1) から (3) を適用しただけでは，エラー原因とするシステムコールエラーを特定できないことがある．コマンドのエラーと無関係なシステムコールエラーが exit

表 3　エラーコードと例外の対応表の抜粋
Table 3　Extract of table of error code and exception.

エラーコード	エラーの説明	対応する例外
EACCES	許可がない	NotPermittedException
ENOENT	ファイルが存在しない	NotFoundException
EROFS	ファイルシステムが読み書き専用	ReadOnlyException
ENOMEM	メモリの空きがない	NoFreeMemoryException
ENETUNREACH	ネットワークが到達できない	UnreachableException
EINTR	関数呼び出しが割り込まれた	InterrupttedBySignalException

関数を呼び出す直前に発生すると，エラー原因を特定することができなくなる．

このようなシステムコールエラーとして，gettext 関数 [9] から発生するものがあげられる．gettext 関数は文字列の翻訳を行う関数であり，翻訳対象の文字列を引数にとり，翻訳した文字列を返す．

この関数は，翻訳対象の文字列に対応する翻訳結果をローカライズファイルから取得することで，文字列の翻訳を行う．翻訳の際，この関数の中でいくつかのシステムコールエラー（ローカライズファイルが存在しない）が発生し，翻訳に失敗することがある．このとき，gettext 関数は，引数として与えれた翻訳対象の文字列をそのまま返す．一般的に gettext 関数はメッセージ出力の際に用いられるため，文字列の翻訳に失敗してもプログラムの動作に影響を及ぼすことはない．

ほとんどのコマンドは，システムコールエラーが発生し異常終了する際，エラーメッセージを出力する．このとき，gettext 関数を用いてエラーメッセージを翻訳することがある．gettext 関数を呼び出した際にシステムコールエラーが発生すると，エラー原因の特定を正しく行うことができない．したがって，**図 8** のように，__dcigettext 関数（gettext 関数の内部で呼ばれる関数）から呼び出されたシステムコールを取り除くことで，この問題に対処している．

5.3　例外発行機構

例外発行機構では，エラー原因に対応する例外を発行する．4.2 節で述べたようにエラーコードと例外の対応表を用いることで，システムコールエラーから例外への変換を実現した．文献 [10] にあげられてあるエラーコードについて対応表を作成した．**表 3** に対応表の一部を示す．

なお，コマンドが異常終了した際，システムコールエラーが発生していなかった場合は，NotRelatedSyscallException を発行する．

例外が発行された後，例外の種類に応じた例外ハンドラに処理が移る．

6.　評価

本章では，D-Shell のオーバヘッドの測定と例外発行の評価について述べる．以下のような 64 bit の Linux システムで評価を行った．

- OS: OpenSUSE 12.3 Linux Kernel 3.7.10 x86_64
- glibc 2.17
- strace+ 20131014
- coreutils 8.17 (ls, dd)
- rsync 3.0.9
- net-tools 1.6 (ifconfig)
- iputils 20101006 (ping)

表 4　ls のベンチマーク
（単位はミリ秒，括弧内は倍率）
Table 4　ls command benchmark.

	正常終了	異常終了
トレース無効	145 (1)	234 (1)
トレース有効	12,164 (84)	67,398 (288)

6.1　システムコールトレースのオーバヘッドの測定

D-Shell はコマンドの実行時にシステムコールトレースを行っているためオーバヘッドが生じる．**表 4** に ls コマンドを D-Shell 上で実行した際のベンチマークを示す．

ls コマンドが正常終了した場合と異常終了した場合について，100 回実行した際にかかった時間を測定した．それぞれの場合について，システムコールトレースの有無による実行時間を比較した．

なお，D-Shell は以下のようにコマンドの先頭に @untrace オプションをつけることで，そのコマンドに対するシステムコールトレースを無効にすることができる．この場合でも例外発行を行うことができ，DShellExeception が発行される．

```
@untrace rsync -av /var/www rsync://server/backup/
```

表 4 で示したように，システムコールトレースを行うことによるオーバヘッドは非常に大きい．システムコールトレースを行うと，コマンドの正常終了時で約 80 倍，異常終了時で約 300 倍のオーバヘッドがあることが確認された．

実行速度の低下は，サーバプログラムのように長時間動作するものでは致命的な欠点になりうる．しかしながら，シェルで実行するコマンドは実行時間が短く，また，前述のようにシステムコールトレースの有無をコマンドごとに

表 5　例外発行の評価結果
Table 5　Evaluation result of an exception issue.

コマンド	発生させたエラー	D-Shell が発行した例外	対応	終了ステータス
ls	ファイルへのアクセス権限がない	NotPermittedException	○	2
ls	ファイルが存在しない	FileNotFoundException	○	2
dd	ディスクの空きがない	NoFreeSpaceException	○	1
dd	ファイルシステムが読み込み専用	ReadOnlyException	○	1
ifconfig	インタフェースが存在しない	DeviceNotFoundException	○	−1
ifconfig	インタフェースへの権限がない	NotPermittedOperateException	○	−1
ping	ホストが存在しない	UnreachableHostException	○	1
ping	サブネットワークが存在しない	InterruptedBySignalException	×	1
rsync	ネットワークが到達できない	UnreachableNetworkException	○	10
rsync	接続先でサーバが動作していない	ConnectionRefusedException	○	10

切替え可能であるため，実行速度の低下はそれほど問題にならない．詳細な例外発行と実行速度はトレードオフの関係にあるため，状況に応じた選択が必要である．

6.2　例外発行の評価

システムコールをともなう典型的なコマンドである，ls コマンド，dd コマンド，ifconfig コマンド，ping コマンド，rsync コマンドを対象とし，それぞれのコマンドに対して 2 種類のエラーを意図的に発生させた．このとき，発生させたエラーの原因と D-Shell が発行する例外が対応するか調べた．また，エラーが発生した際のコマンドの終了ステータスとの比較も行った．

評価結果は**表 5** のとおりである．発生させた 10 個のエラーのうち 9 個については，適切な例外が発行されたことを確認した．また，D-Shell を用いることで終了ステータスよりも詳細にエラー原因を判別できることが確認された．

ping コマンドで発生させた，サブネットワークが存在しないというエラーは，D-Shell が発行した例外と対応していなかった．本来であれば，D-Shell はネットワークに関連する例外を発行するはずである．

しかし，このとき D-Shell によって発行された例外は InterruptedBySignalException であった．この例外はシステムコールがシグナルで割り込まれたことを表すものである．ping コマンドは，一定時間が経ってもパケットが返ってこないと，SIGALRM が発生しタイムアウトするという実装になっている．ゆえに，InterruptedBySignalException が発行されたと考えられる．

7.　課題

現在，達成されていない課題がいくつか存在する．本章では，D-Shell の課題について述べる．

7.1　不適切な例外の発行

D-Shell の例外発行は，コマンド中でクリティカルなシステムコールエラーが発生した後，exit 関数を呼び出して終了する，という前提に基づいている．したがって，異常終了と無関係なシステムコールエラーが終了直前に発生すると，誤った例外が発行されてしまうという問題がある．5.2.4 項で述べたように，エラー原因と無関係であることが明らかなものに関しては対処することができる．しかし，エラー原因との関連性が明らかでないシステムコールエラーについては対処するのが困難である．この問題を解決するために，コマンドの標準エラー出力を例外発行に利用する手法を検討している．

7.2　オーバヘッドの削減

6.1 節で述べたように，システムコールトレースによるオーバヘッドが非常に大きいことが分かっている．オーバヘッドの原因は，システムコールトレーサ（strace+ コマンド）の実装に起因する．strace+ コマンドは ptrace システムコールを用いてシステムコールトレースを行っており，これのオーバヘッドは大きいことが知られている．

したがって，ptrace システムコールを利用しない手法を用いてトレーサを再実装することで，オーバヘッドの削減が可能になる．今後は，環境変数 LD_PRELOAD を用いたシステムコールトレーサの実装を検討している．

8.　関連研究

例外処理機能を持つシェル言語処理系がいくつか提案されている．本章では例外処理機能を持つ既存の処理系と D-Shell の比較を行う．

PowerShell [11] は .NET Framework を基盤とするシェルスクリプト処理系である．PowerShell 上で実行されるコマンドは .NET Framework のクラスとして実装されているため，例外を発行することができる．例外の種類に応じた処理を記述すれば，エラー処理が実現できる．しかし，PowerShell は .NET Framework の例外処理機能を利用しているため，.NET Framework 上で動作しないコマンドは詳細な例外を発行できない．一方，D-Shell は，コマンドの内部で発生するシステムコールエラーをもとに例外発行を

行うため，既存のコマンドをそのまま用いることができ，コマンドの修正や再実装が必要ない．

BeanShell [12] は JavaVM 上で動作するシェル言語処理系である．BeanShell では Java と互換性のある文法をサポートする．BeanShell も例外処理機能を持つが，PowerShell と同様の問題をかかえている．

DCL [13] は OpenVMS オペレーティングシステム用のシェル言語処理系である．OpenVMS のシステム管理に用いられている．DCL では，特定のコマンドを用いることで，コマンドが異常終了した際に例外ハンドラを実行することができる．しかし，異常終了の原因が分からないため，原因に応じたエラー処理を記述することができない．一方，D-Shell は異常終了の原因を推定し，適切な例外を発行するため，原因に応じたエラー処理を記述することができる．

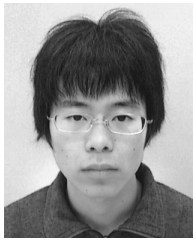
9. まとめ

本論文では，詳細なエラー処理を実現するために，例外処理機能を備えたシェル言語処理系 D-Shell を提案した．D-Shell は，コマンドが発行するシステムコールを追跡し，コマンドが異常終了した際，システムコールエラーを基にエラー原因を推定し，例外発行を行う．評価を行ったところ，既存の終了ステータスを用いる方法よりも詳細にエラー原因を識別できることを確認した．

謝辞　本研究は，JST/CREST「実用化を目指した組込みシステム用ディペンタブル・オペレーティングシステム」領域の研究課題「実行時の安全性を確保する SecurityWeaver と P-SCRIPT」の一部として行われた．本論文の作成にあたり，井出真広氏，志田駿介氏にさまざまなご教示をいただいたことを深感する．

参考文献

[1] Robbins, A. and Beebe, N.H.F.: *Classic Shell Scripting*, O'Reilly Media, Inc. (2005).
[2] 東京証券取引所グループ：株式売買システムの障害発生に関する再発防止措置等について (2012)，入手先 ⟨http://www.tse.or.jp/news/03/b7gje6000002bfdr-att/b7gje6000002bfkq.pdf⟩.
[3] Newham, C.: *Learning the bash shell, 3rd edition*, O'Reilly Media, Inc. (2005).
[4] Cantrill, B.M., Shapiro, M.W. and Leventhal, A.H.: Dynamic instrumentation of production systems, *Proc. Annual Conference on USENIX Annual Technical Conference*, *ATEC '04*, p.2, USENIX Association (2004) (online), available from ⟨http://dl.acm.org/citation.cfm?id=1247415.1247417⟩.
[5] Red Hat Inc., IBM Corp. and Intel Corporation: SystemTap Language Reference (2007), available from ⟨http://sourceware.org/systemtap/langref/⟩.
[6] Kranenburg, P., Lankester, B. and Sladkey, R.: strace (1991), available from ⟨http://sourceforge.net/projects/strace/⟩.
[7] Rostedt, S.: ftrace - Function Tracer (2008), available from ⟨https://www.kernel.org/doc/Documentation/trace/ftrace.txt⟩.
[8] Kranenburg, P., Lankester, B. and Sladkey, R.: strace+: An improved version of strace that collects stack traces (1991), available from ⟨http://code.google.com/p/strace-plus/⟩.
[9] Free Software Foundation, Inc.: gettex (1998), available from ⟨https://www.gnu.org/software/gettext/⟩.
[10] Linux man-pages project: Man Page of ERRNO (2008), available from ⟨http://linuxjm.sourceforge.jp/html/LDP_man-pages/man3/errno.3.html⟩.
[11] Holmes, L.: *Windows powershell quick reference*, *1st edition*, O'Reilly (2006).
[12] Niemeyer, P.: BeanShell Lightweight Scripting for Java (2000), available from ⟨http://www.beanshell.org/⟩.
[13] Hewlett - Packard Development Company, L.P: Simplifying Maintenance with DCL (2007), available from ⟨http://h71000.www7.hp.com/openvms/journal/v9/simplifying_maintenance_with_dcl.pdf⟩.



関口 渚

1991 年生．2014 年横浜国立大学工学府博士課程前期入学．言語処理系の研究に従事．

倉光 君郎　（正会員）

1972 年生．愛知県出身．2000 年東京大学大学院理学系研究科情報科学専攻博士課程中途退学．同年東京大学大学院情報学環助手．2007 年より横浜国立大学工学部准教授．ユビキタスコンピューティング応用から，プログラミング言語 Konoha の研究開発に着手する．博士（理学），2008 年山下研究記念賞受賞．ACM，IPSJ，日本ソフトウェア科学会各会員．